2025

Bid3000

EKSAMEN

OSCAR, ALBERT OG SARA

Navn	kandidatnr
OSCAR	7026
SARA	7008
ALBERT	7009

Table of content

Task 1

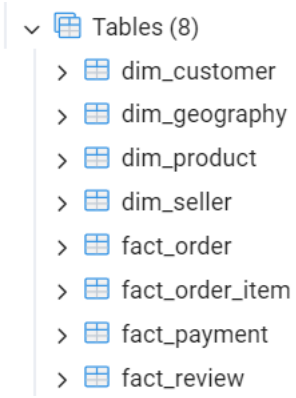
Warehouse design decisions:

Our data Warehouse has a star schema design, it includes:

Dimension tables that hold the business context, which are denormailized and descriptive. Fact tables which store measurable business events, and we have surrogate keys for joining facts to dimensions, ensuring consistency and slowly changing handling (SCD). Our design priorities:

- Query performance, because it is denormalized lookups with fewer joins
- Historical tracking, dimension changes do not break fact associations
- Flexibility for OLAP queries

These are all the tables:



Each dimension and fact table has a surrogate key generated via CREATE SEQUENCE. Surrogate keys are used instead of business keys for example, seq_customer_sk instead of customer_id. (In the interest of time we created this in SQL, but it's possible to do this directly in Pentaho as well.)

```
CREATE SEQUENCE seq_customer_sk START 1;
CREATE SEQUENCE seq_product_sk START 1;
CREATE SEQUENCE seq_seller_sk START 1;
CREATE SEQUENCE seq_date_sk START 1;
CREATE SEQUENCE seq_geo_sk START 1;
CREATE SEQUENCE seq_order_sk START 1;
CREATE SEQUENCE seq_order_item_sk START 1;
CREATE SEQUENCE seq_payment_sk START 1;
CREATE SEQUENCE seq_review_sk START 1;
```

This is because:

- Buisness keys may change or be inconsistent.
- Surrogate keys ensures stable joins over time.

- They enable support for slowly changing dimensions like if a customer moves city, but it will keep the old history intact.

Grain of fact tables

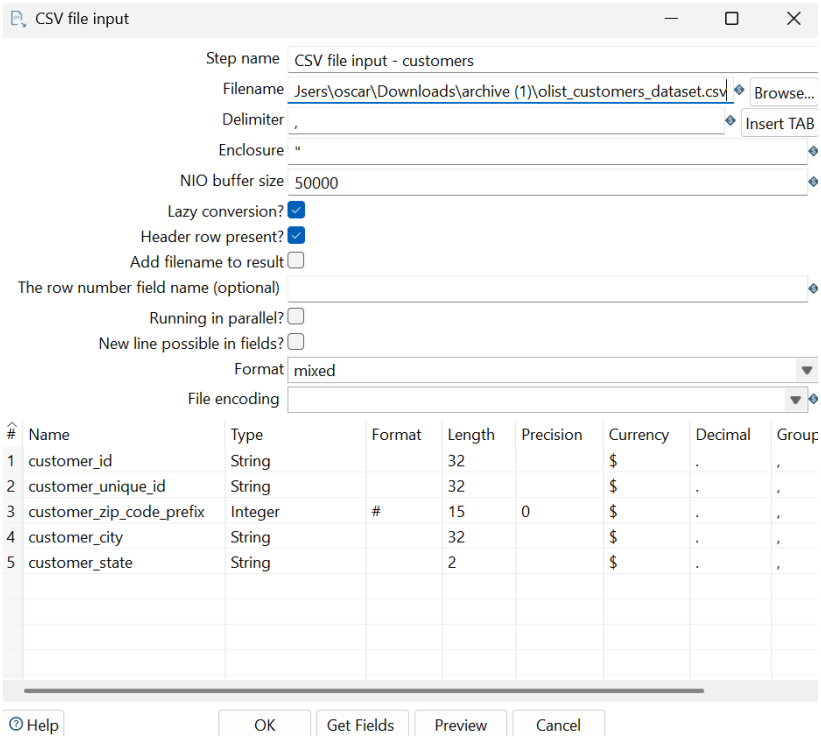
- Fact_order – one row per order
- Fact_order_item – One row per order item
- Fact_payment – one row per payment transaction per order
- Fact_review – one review per order
- Order can be analyzed independently. Items, payments and reviews can be “rolled” up to orders.

Design Rationale

- Fact-dimension separation makes it flexible:
    - Multiple dimensions can be combined with facts
- Surrogate keys prevent broken relationships if business ID’s change.
- Grain choices
    - Order, item, payment and review kept separate, prevents data duplication and supports detailed drill-down.

Step-by-step ETL process

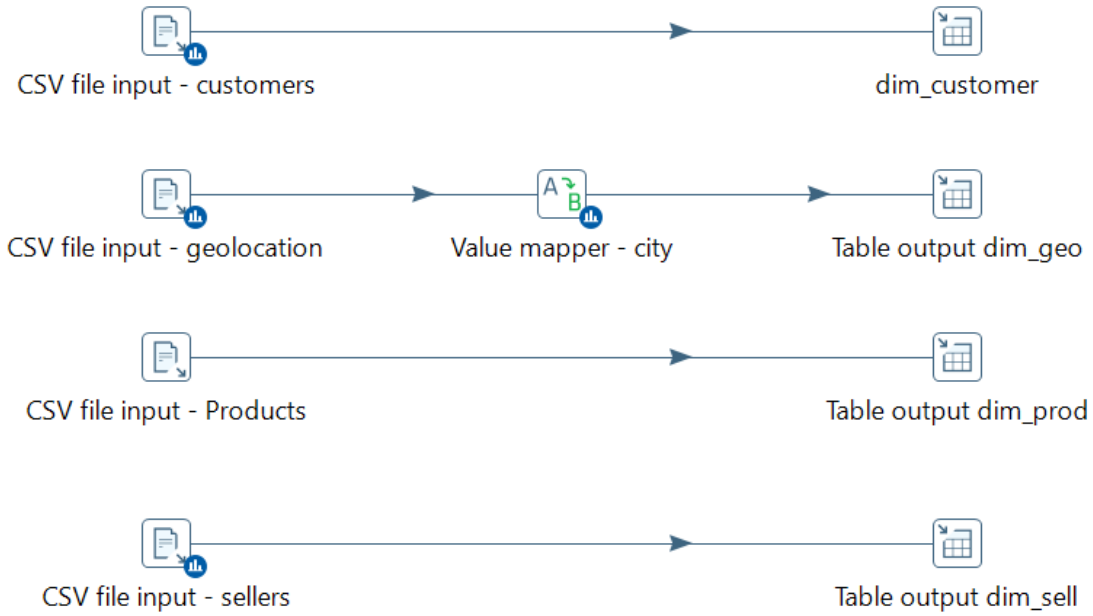
We started by importing all the data from the CSV files. We did this for all our tables, as shown below.



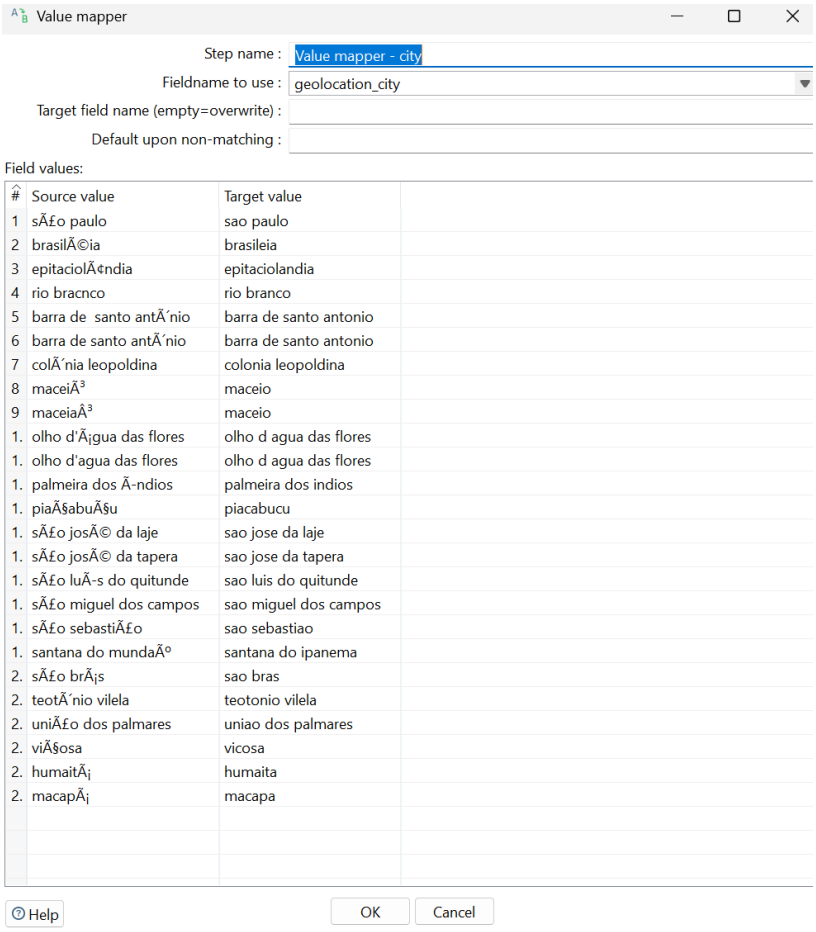
Then we separated our dimension tables from the fact tables, by creating a spoon transformation for each fact table, and one combined transformation for the dimensional tables.

We created a job that runs all the transformations in our chosen order. We did this because all the dimensional tables must be uploaded first (but in no strict order). Then we chose which fact tables to upload in which order based on the FK keys.

In the T_dim_tables you can see that for the geolocation, we have a “value mapper” to sort out some spelling errors in the dataset. We realized that the dataset had way too many spelling errors in geolocation for us to manually fix with a value mapper, so we accept some faulty city names for this task. We could use javascript to fix more city names automatically, but we did not implement it.

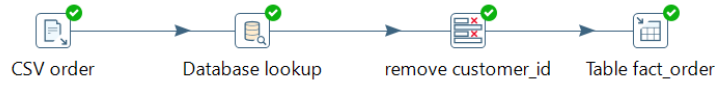


The inconsistency in the data is because of the Portuguese letters as shown below, which creates multiple of the same cities. Another option is to replace the Portuguese letters to corresponding English letters, but this also created a few problems as many were written differently.

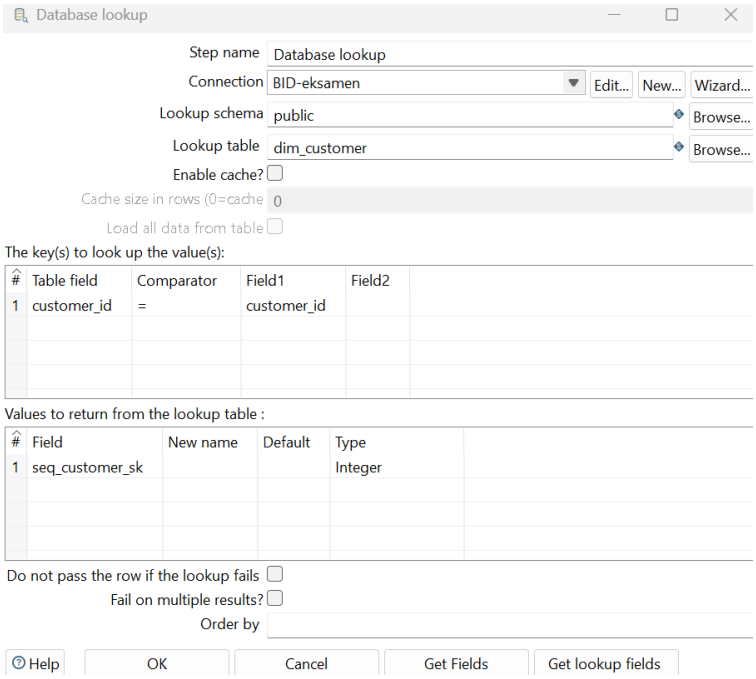


#	Source value	Target value
1	sÃ£o paulo	sao paulo
2	brasilÃ©ia	brasileia
3	epitaciolÃ¢ndia	epitaciolandia
4	rio bracnco	rio branco
5	barra de santo antÃ´nio	barra de santo antonio
6	barra de santo antÃ´nio	barra de santo antonio
7	colÃ´nia leopoldina	colonia leopoldina
8	maceiÃ³	maceio
9	maceiaÂ³	maceio
1.	olho d'Ã¡gua das flores	olho d agua das flores
1.	olho d'agua das flores	olho d agua das flores
1.	palmeira dos Ã-ndios	palmeira dos indios
1.	piaÃ§abuÃ§u	piacabucu
1.	sÃ£o josÃ© da laje	sao jose da laje
1.	sÃ£o josÃ© da tapera	sao jose da tapera
1.	sÃ£o luÃ-s do quitunde	sao luis do quitunde
1.	sÃ£o miguel dos campos	sao miguel dos campos
1.	sÃ£o sebastiÃ£o	sao sebastiao
1.	santana do mundaÃº	santana do ipanema
2.	sÃ£o brÃ¡s	sao bras
2.	teotÃ´nio vilela	teotonio vilela
2.	uniÃ£o dos palmares	uniao dos palmares
2.	viÃ§osa	vicosa
2.	humaitÃ¡	humaita
2.	macapÃ¡	macapa

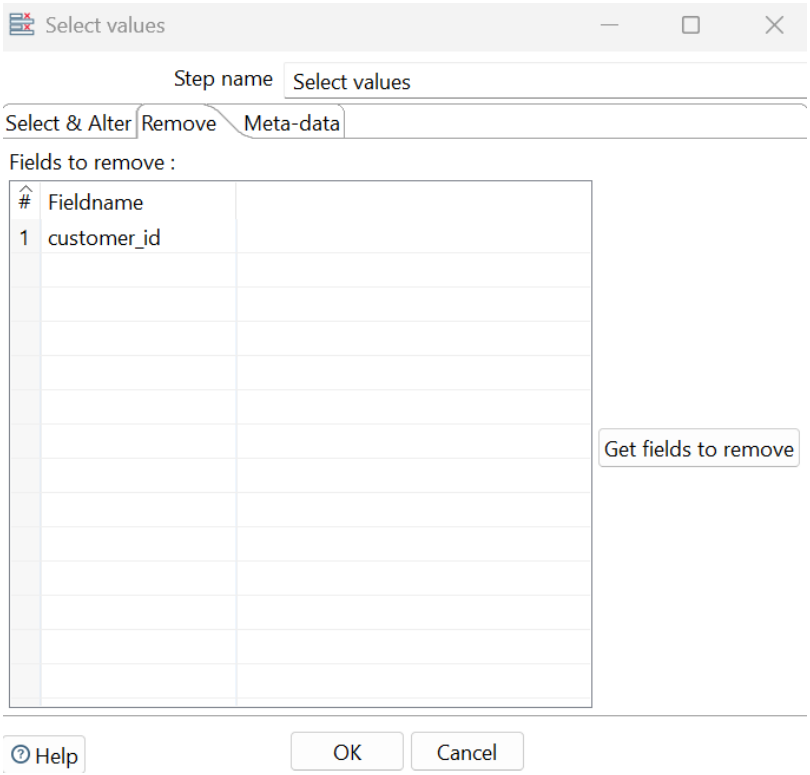
In our fact tables transformations, we used “database lookup” and “select values” to establish a surrogate key and remove the obsolete id (business id) it was based on. The simplest example of this can be seen in our T_fact_order transformation. Here we are looking up the customer_id and giving it the corresponding surrogate key, then we use a “select values” which we named remove customer_id. This one makes sure to remove the customer_id so that we only have the surrogate key (seq_customer_sk) in the fact_order table.



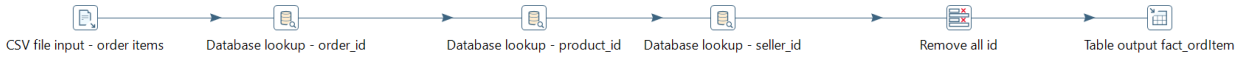
Below shows the logic in the Database Lookup. As you can see, we find the customer_id that we have in our CSV file (order), and we find the same customer_id in the database that belongs to the dim_customer table. Then we return a seq_customer_sk key (surrogate), which means at this point we have both the surrogate key and the business key.

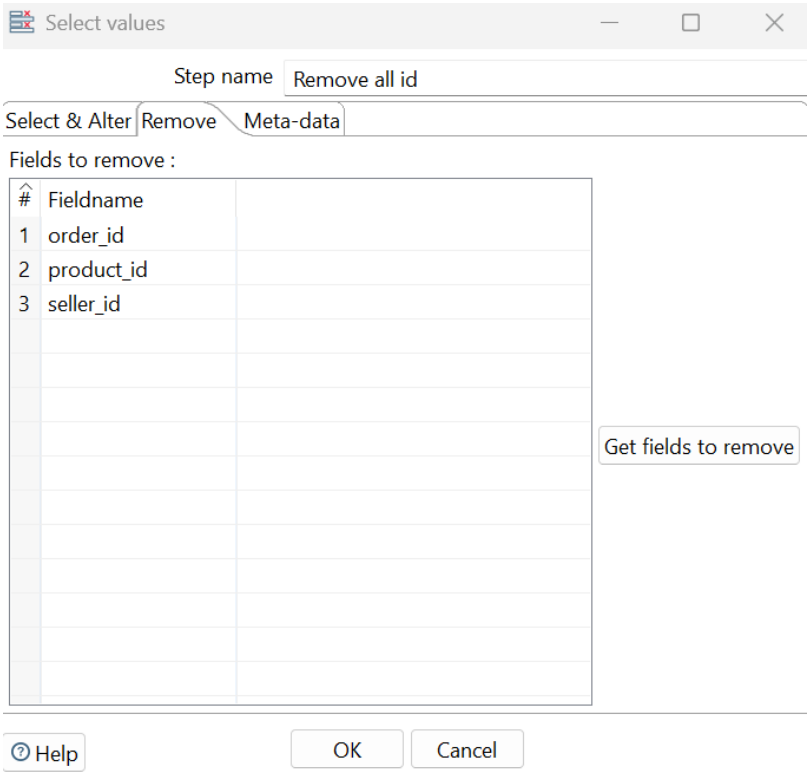


Now we will use the "Select Values" to remove the business key (customer_id) in the fact_order table. This is very simple, we just chose the fieldname we wanted to remove which is customer_id, and then it will remove it, leaving us with just the surrogate key. Then in the last step it will upload to the data warehouse with our changes.

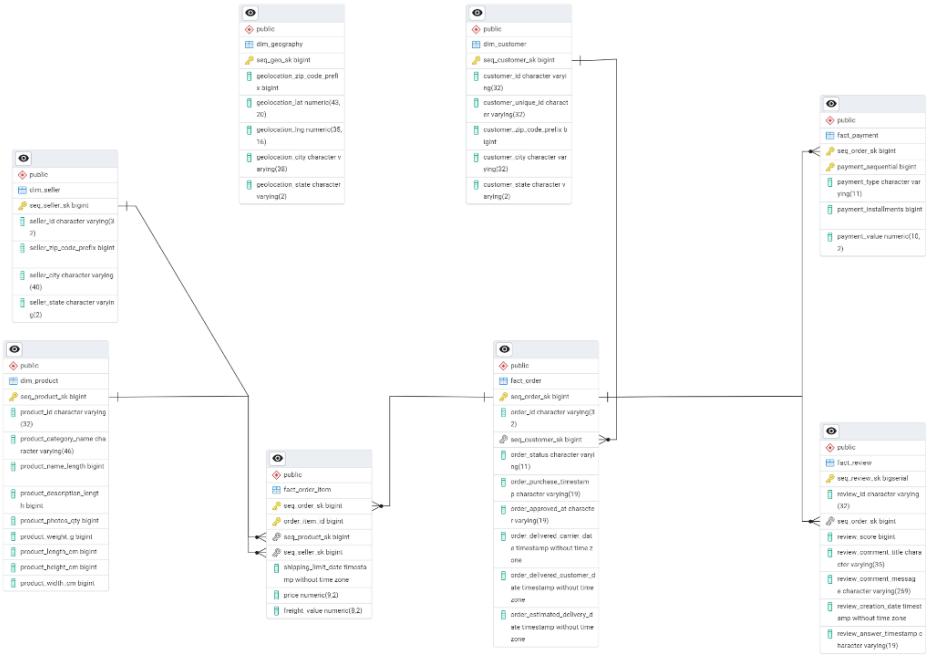


As mentioned, we use the same logic for the rest of the fact tables. But for the fact_order_item we had to use 3 lookups, and one select value:





ERD


public
dim_geography
seq_geo_sk bigint
geolocation_zip_code_prefix bigint
geolocation_lat numeric(48, 26)
geolocation_lng numeric(38, 16)
geolocation_city character varying(30)
geolocation_state character varying(2)
public
dim_customer
seq_customer_sk bigint
customer_id character varying(32)
customer_unique_id character varying(32)
customer_zip_code_prefix bigint
customer_city character varying(32)
customer_state character varying(2)
public
fact_payment
seq_order_sk bigint
payment_sequential bigint
payment_type character varying(11)
payment_installments bigint
payment_value numeric(10, 2)
public
dim_seller
seq_seller_sk bigint
seller_id character varying(32)
seller_zip_code_prefix bigint
seller_city character varying(40)
seller_state character varying(2)
public
dim_product
seq_product_sk bigint
product_id character varying(32)
product_category_name character varying(46)
product_name_length bigint
product_description_length bigint
product_photos_qty bigint
product_weight_g bigint
product_length_cm bigint
product_height_cm bigint
product_width_cm bigint
public
fact_order_item
seq_order_sk bigint
order_item_id bigint
seq_product_sk bigint
seq_seller_sk bigint
shipping_limit_date timestamp without time zone
price numeric(9,2)
freight_value numeric(8,2)
public
fact_order
seq_order_sk bigint
order_id character varying(32)
seq_customer_sk bigint
order_status character varying(11)
order_purchase_timestamp character varying(19)
order_approved_at character varying(19)
order_delivered_carrier_date timestamp without time zone
order_delivered_customer_date timestamp without time zone
order_estimated_delivery_date timestamp without time zone
public
fact_review
seq_review_sk bigserial
review_id character varying(32)
seq_order_sk bigint
review_score bigint
review_comment_title character varying(35)
review_comment_message character varying(289)
review_creation_date timestamp without time zone
review_answer_timestamp character varying(19)

Task 2

A) SQL Analytical queries

All the SQL queries are in analytical_queries.sql in the database folder with detailed comments.

Table of Contents

This table of contents has been made using generative AI

For these queries, we also ran them in a jupyter notebook to get a better grasp of how to use it. Below, you can also see screenshots of the imports used and database setup, along with a simple function that accepts queries as parameters.

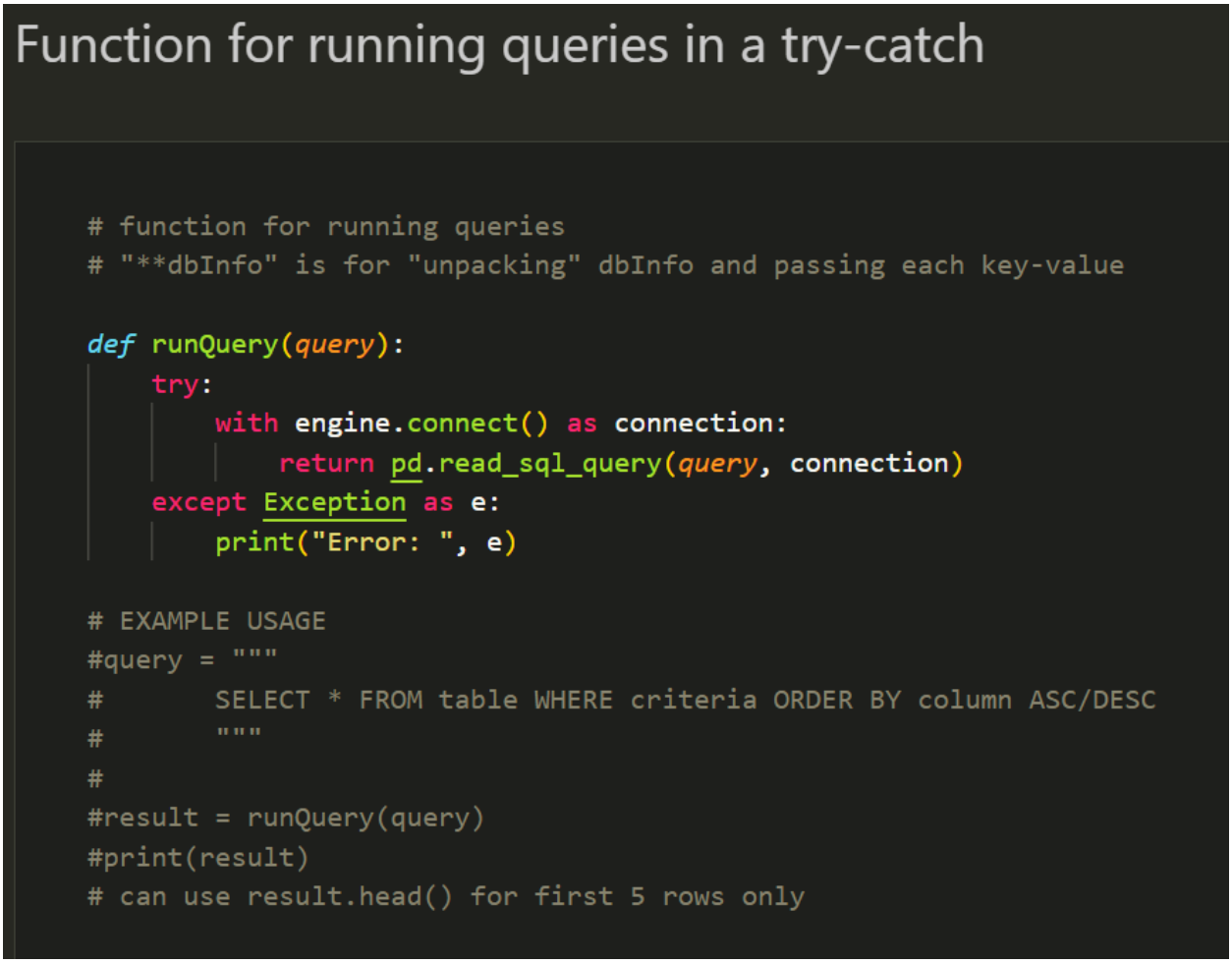
Imports

```
# import pandas and sqlalchemy
import pandas as pd
from sqlalchemy import create_engine

# Visualization
import matplotlib.pyplot as plt  # For creating tree visualizations
import seaborn as sns            # For statistical plots

# to run this jupyter book, run:
# pip install ipykernel
# pip install pandas sqlalchemy matplotlib seaborn
```

Database and pandas setup

```
# connection params
pd.set_option('display.max_rows', 100)        # Default is 10
pd.set_option('display.max_columns', 100)     # Default is 20

engine = create_engine("postgresql+psycopg2://postgres:2424@localhost:5432/bid3000")
```

The images above show imports used for running and displaying some graphs, though visualization was not used for these queries. The database setup depends on the users database information and has to be adjusted accordingly.



We wanted a simple way of running queries in the jupyter notebook without typing too much for each query. This function performs the query with the database config specified earlier, but in a try-catch as you should. Some instructions are also included in the code comments.

Here are the results of the queries:

1. Time-based Trend Analysis

Year-over-year growth analysis:

	year numeric	total_orders bigint
1	2016	329
2	2017	45101
3	2018	54011

A very simple query that can be expanded on if needed, but with this query you can see the year-to-year growth and do calculations based on that. For example, you can find how much growth in percentage

you have from last year. Or if you have more historical data you could see trends by the preferred interval like each 5th year or even each decade.

Seasonal pattern identification:

	month text	total_orders bigint
1	2016-09	4
2	2016-10	324
3	2016-12	1
4	2017-01	800
5	2017-02	1780
6	2017-03	2682
7	2017-04	2404
8	2017-05	3700
9	2017-06	3245
10	2017-07	4026
11	2017-08	4331
12	2017-09	4285
13	2017-10	4631
14	2017-11	7544
15	2017-12	5673
16	2018-01	7269
17	2018-02	6728
18	2018-03	7211
19	2018-04	6939
20	2018-05	6873
21	2018-06	6167
22	2018-07	6292
23	2018-08	6512
24	2018-09	16
25	2018-10	4

Another simple query that has the same logic, but now with months. This makes it possible to find patterns in for example seasons like summer, winter, automn and spring. You can also have over quarters of the year, or things like Christmas, haloween, summerbreak etc.

1. Time-based Trend Analysis

Year-over-year growth analysis:

we do not have a dim table for dates, so we need to extract and convert

```
# year-over-year revenue
query = """
        SELECT
            EXTRACT(YEAR FROM order_purchase_timestamp::timestamp) AS year,
            SUM(foi.price) AS total_revenue
        FROM fact_order fo
        JOIN fact_order_item foi
            ON fo.seq_order_sk = foi.seq_order_sk
        GROUP BY year
        ORDER BY year;
        """

result = runQuery(query)
print(result)
```

✓ 0.2s

	year	total_revenue
0	2016.0	49785.92
1	2017.0	6155806.98
2	2018.0	7386050.80

A query that returns a total_revenue for the three years we have data on. You can see that we're using SQL aliases, such as "year" and "foi", in this query, and we will continue to do so in almost every query going forward. Aliases make queries more compact and readable at the same time. As commented in the jupyter notebook, we can use SQL EXTRACT() to get the years out of the date info. I did initially think freight_value was something to be added to total_revenue, and had it included in the SUM() function, but that has been removed from all queries. You will probably notice that not all queries have properly been cleaned up after removal of freight_value.

Seasonal pattern identification:

```
# seasonal pattern identification
# shows avg revenue by month
query = """
        SELECT
            TO_CHAR(order_purchase_timestamp::timestamp, 'Month') AS month_name,
            EXTRACT(MONTH FROM order_purchase_timestamp::timestamp) AS month_number,
            ROUND(AVG(foi.price), 2) AS avg_monthly_revenue
        FROM fact_order fo
        JOIN fact_order_item foi
            ON fo.seq_order_sk = foi.seq_order_sk
        GROUP BY month_name, month_number
        ORDER BY month_number;
        """
result = runQuery(query)
print(result)
```

✓ 0.2s

	month_name	month_number	avg_monthly_revenue
0	January	1.0	116.81
1	February	2.0	113.42
2	March	3.0	121.03
3	April	4.0	127.27
4	May	5.0	124.58
5	June	6.0	121.77
6	July	7.0	120.02
7	August	8.0	117.51
8	September	9.0	129.15
9	October	10.0	125.55
10	November	11.0	116.59
11	December	12.0	117.91

In this query, we select month name and number, along with a calculated average monthly revenue. This query does not discriminate on year, which in hindsight it should probably do. If we were to go back on it, we would change the query to also discriminate on year and return averages for all months since 2016.

2. Drill-down and Roll-up Operations

Multi-level aggregation queries

Roll-up is "zooming out" and summarizing roll-up revenue by day

```
# revenue by each individual day
query = """
        SELECT
            EXTRACT(YEAR FROM order_purchase_timestamp::timestamp) AS year,
            EXTRACT(MONTH FROM order_purchase_timestamp::timestamp) AS month,
            EXTRACT(DAY FROM order_purchase_timestamp::timestamp) AS day,
            SUM(foi.price) AS total_revenue
        FROM fact_order fo
        JOIN fact_order_item foi
            ON fo.seq_order_sk = foi.seq_order_sk
        GROUP BY year, month, day
        ORDER BY year, month, day;
        """
result = runQuery(query)
print(result.head()) # returns 600 rows, so we're not gonna print all in this example
```

✓ 0.3s

	year	month	day	total_revenue
0	2016.0	9.0	4.0	72.89
1	2016.0	9.0	5.0	59.50
2	2016.0	9.0	15.0	134.97
3	2016.0	10.0	2.0	100.00
4	2016.0	10.0	3.0	463.48

In this result specifically, we only printed the first five results, this was mostly as to test the .head() function, but also to not print 600 rows. I later found out it does limit itself, as specified in the database config. Anyways, this query returns daily revenues, which we can see were very low in their first five days of 2016. We will see this later as well, as the dataset shows very large growths from 2016 to 2017.

We also ran a query for monthly revenues:

Roll-up revenue by month

```
# revenue by each month
query = """
        SELECT
            EXTRACT(YEAR FROM order_purchase_timestamp::timestamp) AS year,
            EXTRACT(MONTH FROM order_purchase_timestamp::timestamp) AS month,
            SUM(foi.price) AS total_revenue
        FROM fact_order fo
        JOIN fact_order_item foi
            ON fo.seq_order_sk = foi.seq_order_sk
        GROUP BY year, month
        ORDER BY year, month;
        """
result = runQuery(query)
print(result)
```

✓ 0.2s

```
    year  month  total_revenue
0  2016.0    9.0         267.36
```

This is mostly the same as the one above, but on a month-by-month instead.

Results would look something like this:

	year	month	total_revenue
0	2016.0	9.0	267.36
1	2016.0	10.0	49507.66
2	2016.0	12.0	10.90
3	2017.0	1.0	120312.87
4	2017.0	2.0	247303.02
5	2017.0	3.0	374344.30
6	2017.0	4.0	359927.23
7	2017.0	5.0	506071.14
8	2017.0	6.0	433038.60
9	2017.0	7.0	498031.48
10	2017.0	8.0	573971.68
11	2017.0	9.0	624401.69
12	2017.0	10.0	664219.43
13	2017.0	11.0	1010271.37
14	2017.0	12.0	743914.17
15	2018.0	1.0	950030.36
16	2018.0	2.0	844178.71
17	2018.0	3.0	983213.44
18	2018.0	4.0	996647.75
19	2018.0	5.0	996517.68
20	2018.0	6.0	865124.31
21	2018.0	7.0	895507.22
22	2018.0	8.0	854686.33
23	2018.0	9.0	145.00

Roll-up revenue by year

```
# revenue by year
query = """
        SELECT
            EXTRACT(YEAR FROM order_purchase_timestamp::timestamp) AS year,
            SUM(foi.price) AS total_revenue
        FROM fact_order fo
        JOIN fact_order_item foi
            ON fo.seq_order_sk = foi.seq_order_sk
        GROUP BY year
        ORDER BY year;
        """
result = runQuery(query)
print(result)
```

✓ 0.1s

```
     year  total_revenue
0  2016.0       49785.92
1  2017.0     6155806.98
2  2018.0     7386050.80
```

The yearly revenues would also show to reflect the low numbers of 2016.

We also tried some “drill-down” by making queries for revenue by city

Seasonal pattern identification (drill-down)

drill-down is for example to inspect details revenue by city

Revenue by city

```
# revenue by city
query = """
        SELECT
            dc.customer_state,
            dc.customer_city,
            SUM(foi.price) AS total_revenue
        FROM fact_order fo
        JOIN dim_customer dc
            ON fo.seq_customer_sk = dc.seq_customer_sk
        JOIN fact_order_item foi
            ON fo.seq_order_sk = foi.seq_order_sk
        GROUP BY dc.customer_state, dc.customer_city
        ORDER BY total_revenue, dc.customer_state, dc.customer_city DESC
        -- LIMIT 10;
        """
result = runQuery(query)
print(result) # can use limit 10 in this example to not display 4300 rows
```

✓ 0.1s

This query should return about 4300 rows, which is not too crazy considering Brazil has more than 5000 cities. Some of these rows should be a result of faulty input data, for example typos in city names and so on. The results returned for this query:

```
     customer_state                customer_city  total_revenue
0                RS  polo petroquimico de triunfo           5.60
1                PR                    sabaudia           5.90
2                MG   santo antonio do rio abaixo           6.00
3                PA         senador jose porfirio           6.00
4                MG             jenipapo de minas           7.48
...             ...                         ...            ...
4295             PR                    curitiba      211738.06
4296             DF                    brasilia      301920.25
4297             MG              belo horizonte      355611.13
4298             RJ              rio de janeiro      992538.86
4299             SP                   sao paulo     1914924.54

[4300 rows x 3 columns]
```

We also have a similar query for revenue by state, as that could be easier data to work in a business setting:

Revenue by state

```
# revenue by state
query = """
        SELECT
            dc.customer_state AS state,
            SUM(foi.price) AS total_revenue
        FROM fact_order fo
        JOIN dim_customer dc
            ON fo.seq_customer_sk = dc.seq_customer_sk
        JOIN fact_order_item foi
            ON fo.seq_order_sk = foi.seq_order_sk
        GROUP BY dc.customer_state
        ORDER BY dc.customer_state
        -- LIMIT 10;
        """
result = runQuery(query)
print(result) # can use limit 10 in this example to not display 27 rows
```

✓ 0.1s

	state	total_revenue
0	AC	15982.95
1	AL	80314.81
2	AM	22356.84
3	AP	13474.30
4	BA	511349.99
5	CE	227254.71
6	DF	302603.94
7	ES	275037.31
8	GO	294591.95
9	MA	119648.22
10	MG	1585308.03
11	MS	116812.64
12	MT	156453.53
13	PA	178947.81
14	PB	115268.08
15	PE	262788.03
16	PI	86914.08
17	PR	683083.76
18	RJ	1824092.67
19	RN	83034.98
20	RO	46140.64
21	RR	7829.43
22	RS	750304.02
23	SC	520553.34
24	SE	58920.85
25	SP	5202955.05
26	TO	49621.74

3. Advanced Window Functions

Ranking and percentile calculations

For ranking and percentile calculations, we used SQL functions such as OVER() and RANK()

Rank sellers by total revenue

```
# ranking sellers by total revenue, with percentile rank
query = """
        SELECT
            ds.seller_id,
            SUM(foi.price) AS total_revenue,
            RANK() OVER (ORDER BY SUM(foi.price) DESC) AS revenue_rank,
            PERCENT_RANK() OVER (ORDER BY SUM(foi.price) DESC) AS revenue_percentile
        FROM fact_order fo
        JOIN fact_order_item foi
            ON fo.seq_order_sk = foi.seq_order_sk
        JOIN dim_seller ds
            ON foi.seq_seller_sk = ds.seq_seller_sk
        WHERE EXTRACT(YEAR FROM fo.order_purchase_timestamp::timestamp) BETWEEN 2016 AND 2018
        GROUP BY ds.seller_id
        -- LIMIT 10;
        """
result = runQuery(query)
sellers_ranked_by_revenue = result
print(result) # can use limit 10 in this example to not display 3100 rows
```

✓ 0.4s

```
                          seller_id  total_revenue  revenue_rank  \
0     4869f7a5dfa277a7dca6462dcf3b52b2      229472.63             1
1     53243585a1d6dc2643021fd1853d8905      222776.05             2
2     4a3ca9315b744ce9f8e9374361493884      200472.92             3
3     fa1c13f2614d7b5c4749cbc52fecda94      194042.03             4
4     7c67e1448b00f6e969d365cea6b010ab      187923.89             5
...                                ...            ...           ...
```

This is for sure a bigger query than those above, and it returns a list of seller_id's, total_revenue (by seller) and that sellers "rank" among other sellers. We get these ranks by using the RANK() and PERCENT_RANK() in SQL along with OVER() with an ordering specified in the OVER() function.

Moving averages and cumulative measures

At first, I tried something by just getting amount of orders per day in a given year, that query is still included in the jupyter notebook, and possibly the SQL file too. We realized that was not the correct way to proceed, so we came up with a query using a common table expression (CTE). CTE's are incredibly useful to improve readability and to break down larger queries into more understandable queries.

Here is the common table expression used, and its results alone:

moving averages and cumulative measures take 2

```
# we're gonna use this as a CTE(common table expression)
# in the next query using WITH().
# this mini-query finds amount of sales per day
query = """
        SELECT
            DATE(fo.order_purchase_timestamp) AS order_date,
            COUNT(*) AS daily_order_count
        FROM fact_order fo
        WHERE EXTRACT(YEAR FROM fo.order_purchase_timestamp::timestamp) = 2017
        GROUP BY DATE(fo.order_purchase_timestamp)
        """
result = runQuery(query)
print(result)
```

✓ 0.2s

	order_date	daily_order_count
0	2017-01-05	64
1	2017-01-06	8
2	2017-01-07	8
3	2017-01-08	12
4	2017-01-09	10

Much like the first attempt, it counts an amount of orders per day.

```
# this is a better answer for the task, as we properly
# user the AVG() and OVER() functions for moving averages
# this query shows us the daily amount of orders in 2017,
# and "moving average" which tells us the average of the last seven days
query = f"""
        -- CTE for daily amount of orders
        WITH daily_orders AS (
            SELECT
                DATE(fo.order_purchase_timestamp) AS order_date,
                COUNT(*) AS daily_order_count
            FROM fact_order fo
            WHERE EXTRACT(YEAR FROM fo.order_purchase_timestamp::timestamp) = 2017
            GROUP BY DATE(fo.order_purchase_timestamp)
        )
        SELECT
            order_date,
            daily_order_count,
            -- 7 day moving average of daily order count
            ROUND(AVG(daily_order_count) OVER (
                ORDER BY order_date
                ROWS BETWEEN 6 PRECEDING AND CURRENT ROW
            ), 2) AS moving_avg_week,

            -- cumulative total orders up to current date
            --
            SUM(daily_order_count) OVER (
                ORDER BY order_date
            ) AS cumulative_order_count
        FROM daily_orders
        ORDER BY order_date;
        """

result = runQuery(query)
print(result)
```

✓ 0.1s

order_date	daily_order_count	moving_avg_week	cumulative_order_count

This is probably the most wordy query in this report. It returns daily orders in 2017 along with a moving seven day average, which means the average amount of orders for a day, within the last seven days. It also has a cumulative count of total orders along the way. A very nice query, which we will use in the descriptive analysis later, with a small adjustment to the year selection.

The results look something like this:

	order_date	daily_order_count	moving_avg_week	cumulative_order_count
0	2017-01-05	64	64.00	64.0
1	2017-01-06	8	36.00	72.0
2	2017-01-07	8	26.67	80.0
3	2017-01-08	12	23.00	92.0
4	2017-01-09	10	20.40	102.0
..	...	...	...	...
356	2017-12-27	334	242.00	89298.0
357	2017-12-28	292	242.86	89590.0
358	2017-12-29	270	249.71	89860.0
359	2017-12-30	194	246.29	90054.0
360	2017-12-31	148	250.57	90202.0

[361 rows x 4 columns]

We can see the date, and that days amount of incoming orders. The moving average then tells us what the average daily orders amount is for that 7-day period. The cumulative count can be explained by this formula, where x is the row.

$$\text{Cumulative\_order\_count} = \text{result}[x-1]\text{cumulative\_order\_count} + \text{result}[x].\text{daily\_order\_count}$$

4. Complex Filtering and Subqueries

Multi-dimensional filtering with EXISTS/IN clauses

For the next query, we use a SubQuery.

Multi-dimensional filtering with EXISTS/IN clauses

```
# we will use this subquery in the next task,
# to find product SK-s, joined to dim_customer
# to link each order item to its customer
# and groups by product
# for each product, it counts the number of
# unique cities with DISTINCT() to see where its sold
# HAVING() is used to count and say
# it should only return the products
# that are sold in more than 150 different cities
query = """
        SELECT foi.seq_product_sk
        FROM fact_order_item foi
        JOIN fact_order fo ON foi.seq_order_sk = fo.seq_order_sk
        JOIN dim_customer dc ON fo.seq_customer_sk = dc.seq_customer_sk
        GROUP BY foi.seq_product_sk
        HAVING COUNT(DISTINCT dc.customer_city) > 150
        """
result = runQuery(query)
print(result)
```

✓ 0.1s

	seq_product_sk
0	1750
1	4599
2	5823
3	8291
4	9662
5	13431
6	14052
7	29130
8	30294
9	32099

This small query really shows the power of having these SK's, as it makes it very easy to JOIN on multiple tables. For this query, we wanted to join fact_order_item with fact_order and dim_customer, and such queries are made really simple by including these keys in our tables. Shows one strength of the star schema. Anyways, The result is a list of products that has been sold in more than 150 cities.

This SubQuery is used in:

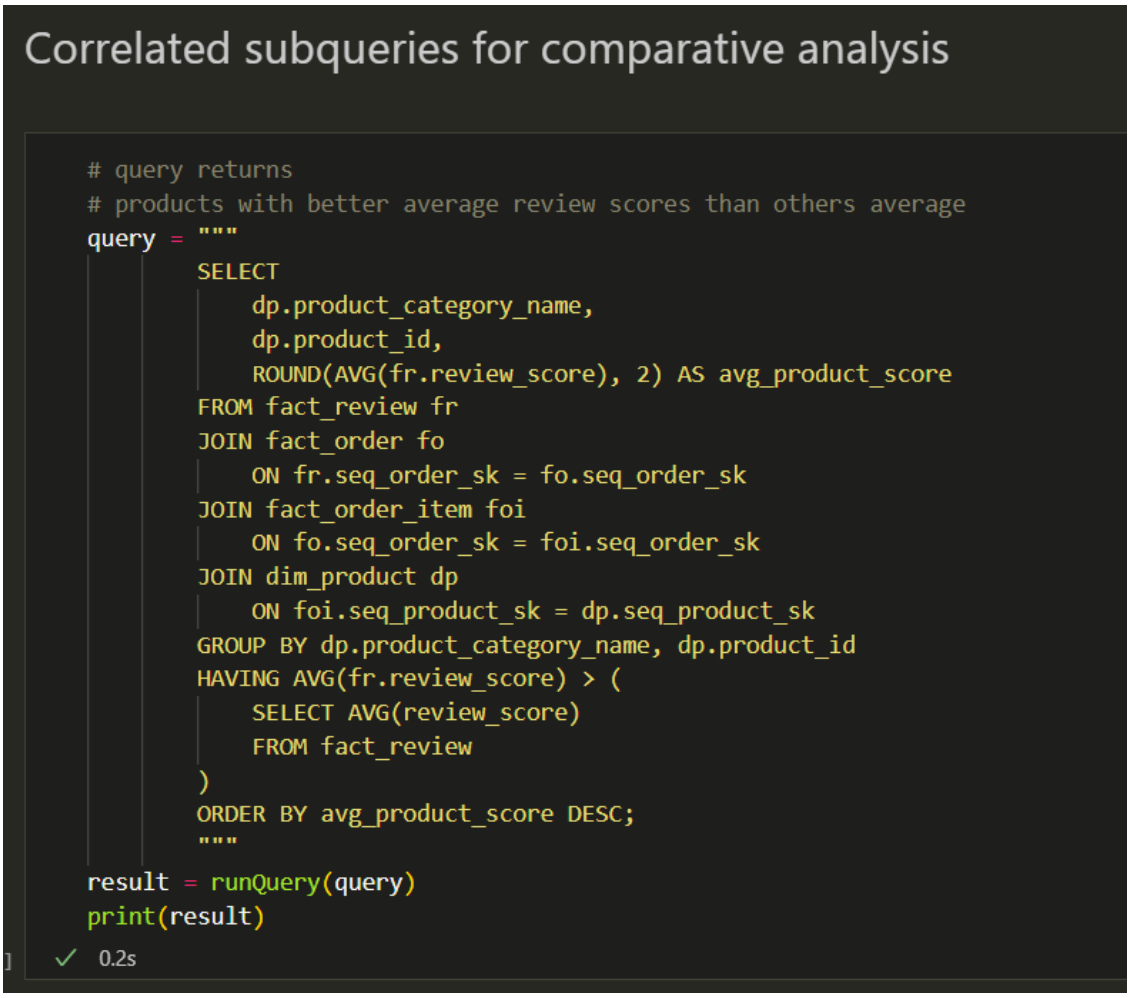
```
# this query returns product that were sold in
# more than 150 different cities
#
# it uses the IN() function with the subquery above
#
query = """
    SELECT dp.product_id, dp.product_category_name
    FROM dim_product dp
    WHERE dp.seq_product_sk IN (
        SELECT foi.seq_product_sk
        FROM fact_order_item foi
        JOIN fact_order fo ON foi.seq_order_sk = fo.seq_order_sk
        JOIN dim_customer dc ON fo.seq_customer_sk = dc.seq_customer_sk
        GROUP BY foi.seq_product_sk
        HAVING COUNT(DISTINCT dc.customer_city) > 150
    );
        """
result = runQuery(query)
products_sold_in_150p_cities = result
print(result)
```

✓ 0.1s

	product_id	product_category_name
0	154e7e31ebfa092203795c972e5804a6	beleza_saude
1	368c6c730842d78016ad823897a372db	ferramentas_jardim
2	2b4609f8948be18874494203496bc318	beleza_saude
3	d1c427060a0f73f6b889a5c7c61f2ac4	informatica_acessorios
4	99a4788cb24856965c36a24e339b6058	cama_mesa_banho
5	aca2eb7d00ea1a7b8ebd4e68314663af	moveis_decoracao
6	422879e10f46682990de24d770e7f83d	ferramentas_jardim
7	3dd2a17168ec895c781a9191c1e95ad7	informatica_acessorios
8	389d119b48cf3043d311335e499d9c6b	ferramentas_jardim
9	53759a2ecddad2bb87a079a1f1519f73	ferramentas_jardim

As you can see, this really just builds on the subquery, by asking for more information on the product, with the subquery in the WHERE clause.

Correlated subqueries for comparative analysis



Again, we take advantage of the keys, to make it simple for ourselves when JOINing tables. This query returns a list of products that all have an average review score higher than the average review score of all products overall. The results looks like:

```
                  product_category_name                        product_id  \
0                         cama_mesa_banho  c55ee2882c5321eb3c9277e5c92e50c7
1                            beleza_saude  54f52360a2b571e7d7b54225b74bbccd
2                      relogios_presentes  d76e09e3182a68025076542af2726322
3      construcao_ferramentas_construcao  486e577f635477c920c95e95dc243bc8
4                               perfumaria  8c10db7e2bc84a4697eaed54db164c7d
...                                    ...                               ...
18680                  relogios_presentes  f819f0c84a64f02d3a5606ca95edd272
18681                   moveis_escritorio  b73f6899a58fe7a37e55149e9a11c717
18682                   moveis_decoracao   28b4eced95a52d9c437a4caf9d311b95
18683                              bebes   920840f7899b13c467d223950c89e9e9
18684                  relogios_presentes  6f3b5b605d91b7439c5e3f5a8dffeea7

       avg_product_score
0                   5.00
1                   5.00
2                   5.00
3                   5.00
4                   5.00
...                  ...
18680               4.09
18681               4.09
18682               4.09
18683               4.09
18684               4.09

[18685 rows x 3 columns]
```

We will also show in the descriptive analysis and on the PowerBI dashboard, that products have really good review scores in general in this dataset.

5. Business Intelligence Metrics

Customer/Product profitability analysis

Customer/Product profitability analysis

```
# profits for each product category
query = """
        SELECT
            dp.product_category_name,
            ROUND(SUM(foi.price), 2) AS total_revenue,
            ROUND(SUM(foi.freight_value), 2) AS total_freight_cost,
            ROUND(SUM(foi.price - foi.freight_value), 2) AS profitability
        FROM fact_order_item foi
        JOIN dim_product dp
            ON foi.seq_product_sk = dp.seq_product_sk
        GROUP BY dp.product_category_name
        ORDER BY profitability DESC;
        """
result = runQuery(query)
product_category_profitabiliy = result
print(result)
```

[66] ✓ 0.0s

Here, at this point we came to realize that the freight_value might be a cost, and not an income. The query returns a list of each product category and that categories profits. Included is also the price and freight_value included in the proficts calculations. The results would look like the following:

product_category_name	total_revenue	total_freight_cost	profitability
relogios_presentes	1205005.68	100535.93	1104469.75
beleza_saude	1258681.34	182566.73	1076114.61
cama_mesa_banho	1036988.68	204693.04	832295.64
esporte_lazer	988048.97	168607.51	819441.46
informatica_acessorios	911954.32	147318.08	764636.24
moveis_decoracao	729762.49	172749.30	557013.19
cool_stuff	635290.85	84039.10	551251.75
automotivo	592720.11	92664.21	500055.90
utilidades_domesticas	632248.66	146149.11	486099.55
brinquedos	483946.60	77425.95	406520.65

Performance KPI calculations specific to your domain

The exam task specified it wanted some information about delivery times for this dataset, so the next query returns an average amount of delivery days for that month. Along with that, there is also a ratio(percentage) of how many deliveries were on-time that month.

Performance KPI calculations specific to your domain

```
# query returns average delivery time and a ratio for on-time delivery
# ordered by year, month so we can see progress and eventually use it in
# a dashboard for business intelligence purposes
# it extracts total seconds with EPOCH() and turns them into days by
# dividing by (60s * 60m * 24h)
# this is then averaged and rounded and aliased as avg_delivery_days
# for the ratio, we use CASE to return a 1 when orders are delivered on time
# otherwise we get a 0
# we sum and and divide by count to get the ratio
query = """
        SELECT
            EXTRACT(YEAR FROM fo.order_purchase_timestamp::timestamp) AS year,
            EXTRACT(MONTH FROM fo.order_purchase_timestamp::timestamp) AS month,
            ROUND(AVG(EXTRACT(EPOCH FROM (fo.order_delivered_customer_date - fo.order_purchase_timestamp::timestamp)) / (60*60*24)),
            ROUND(100.0 * SUM(CASE WHEN fo.order_delivered_customer_date <= fo.order_estimated_delivery_date THEN 1 ELSE 0 END)
                / COUNT(*), 2) AS on_time_delivery_percent
        FROM fact_order fo
        WHERE fo.order_status = 'delivered'
        GROUP BY year, month
        ORDER BY year, month;
        """
result = runQuery(query)
print(result)
```

✓ 0.7s Python

	year	month	avg_delivery_days	on_time_delivery_percent
0	2016.0	9.0	54.81	0.00
1	2016.0	10.0	19.60	98.87
2	2016.0	12.0	4.69	100.00
3	2017.0	1.0	12.65	96.93
4	2017.0	2.0	13.17	96.79
5	2017.0	3.0	12.95	94.42
6	2017.0	4.0	14.92	92.14
7	2017.0	5.0	11.32	96.36

The results tell us that the on-time ratio has gotten slightly worse since the end of 2016, but that might be due to a small amount of orders during those few months. They have a nice overall on-time ratio with nothing below 92,14%.

Exports:

Some of these queries were used in the dashboard, and could be exported to CSV files.

Exporting results to CSV

```python
import os

os.makedirs("exports", exist_ok=True)

# List all result variables to export for advanced analytics
advanced_exports = {
    "product_category_profitabiliy": "exports/product_category_profitability.csv",
    "sellers_ranked_by_revenue": "exports/sellers_ranked_by_revenue.csv",
    "products_sold_in_150p_cities": "exports/products_sold_in_150p_cities.csv",
    "monthly_profit": "exports/monthly_profitability_reviews.csv"
}

try:
    for var, path in advanced_exports.items():
        if var in globals():
            globals()[var].to_csv(path, index=False)
        else:
            print(f"{var} not found.")
    print("Export done.")
except Exception as e:
    print("Error during export:", e)
```

✓ 0.0s

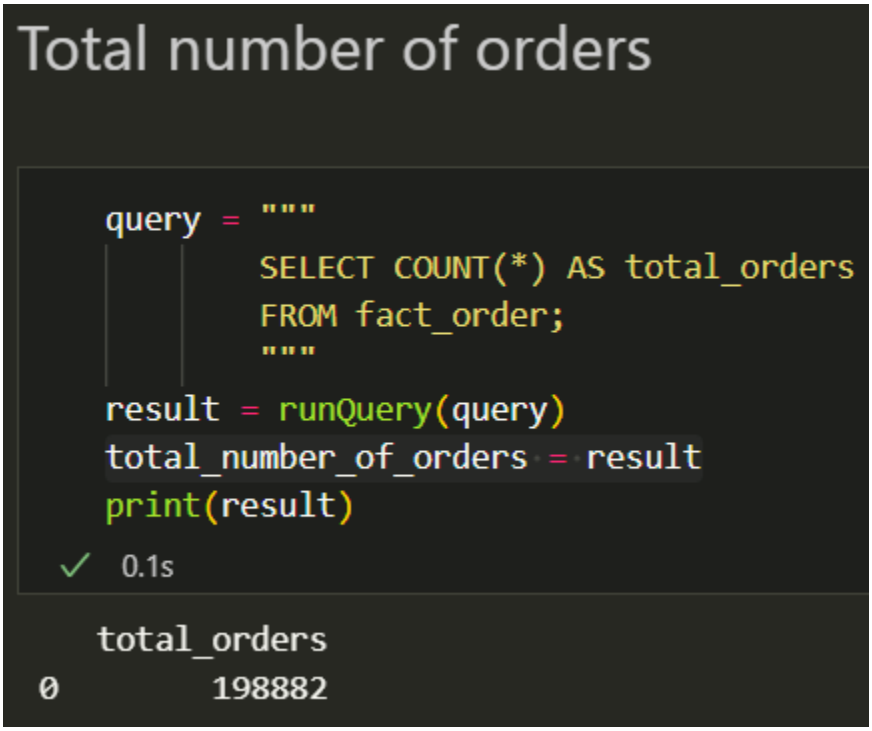
```
Export done.
```

B) Python Analytics Integration

Descriptive Analytics

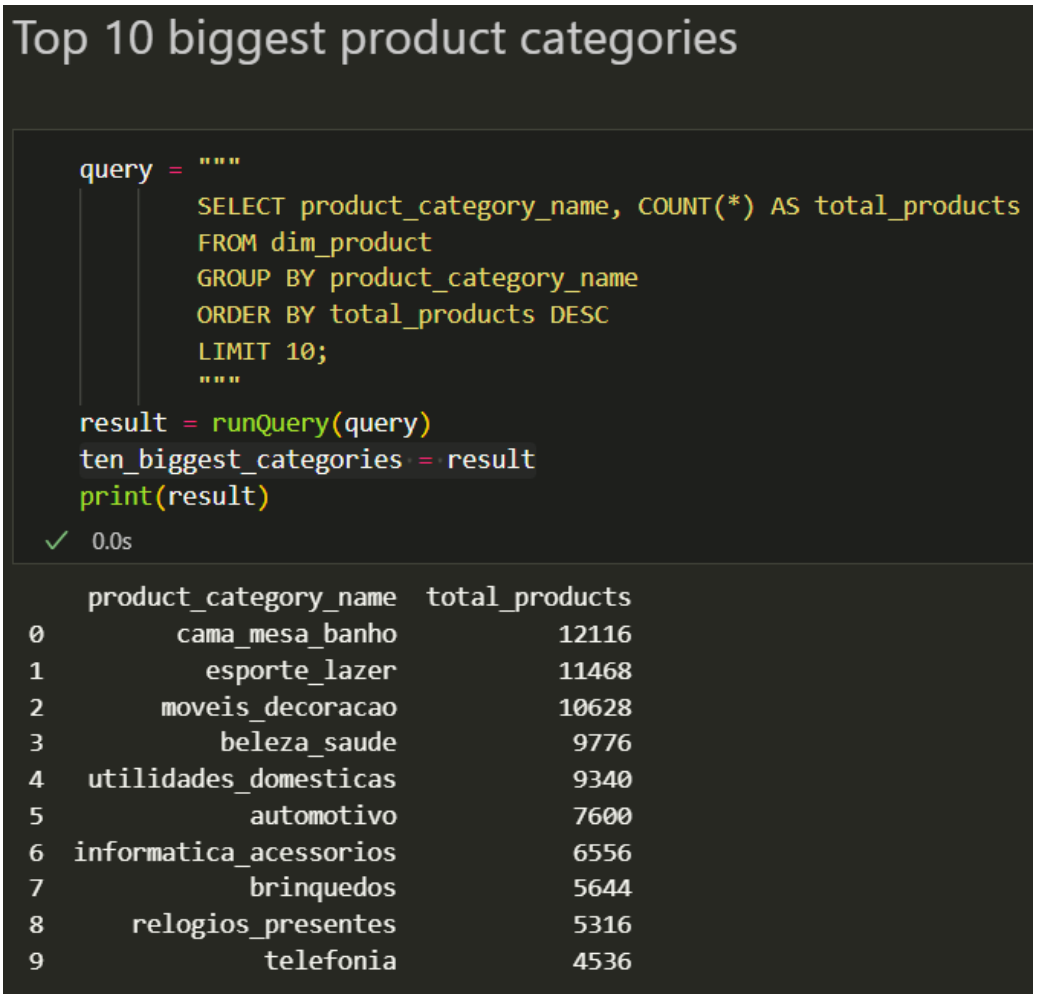
The descriptive analytics give us statistical summaries and other insightful data. Please note that the jupyter notebook for descriptive analytics contains the same logic for imports, database setup and running queries.

Total number of orders



This was originally just a test-query, but we figured this could be a cool number to show on a dashboard.

Top 10 biggest product categories



This is a table of the total amount of products in the ten largest categories (product-wise).

Total Revenue by payment type

Total revenue by payment type

```
query = """
        SELECT payment_type, ROUND(SUM(payment_value), 2) AS total_revenue
        FROM fact_payment
        WHERE payment_type != 'not_defined'
        GROUP BY payment_type
        ORDER BY total_revenue DESC;
        """
result = runQuery(query)
revenue_by_payment_type = result
print(result)
```

✓ 0.0s

	payment_type	total_revenue
0	credit_card	12542084.19
1	boleto	2869361.27
2	voucher	379436.87
3	debit_card	217989.79

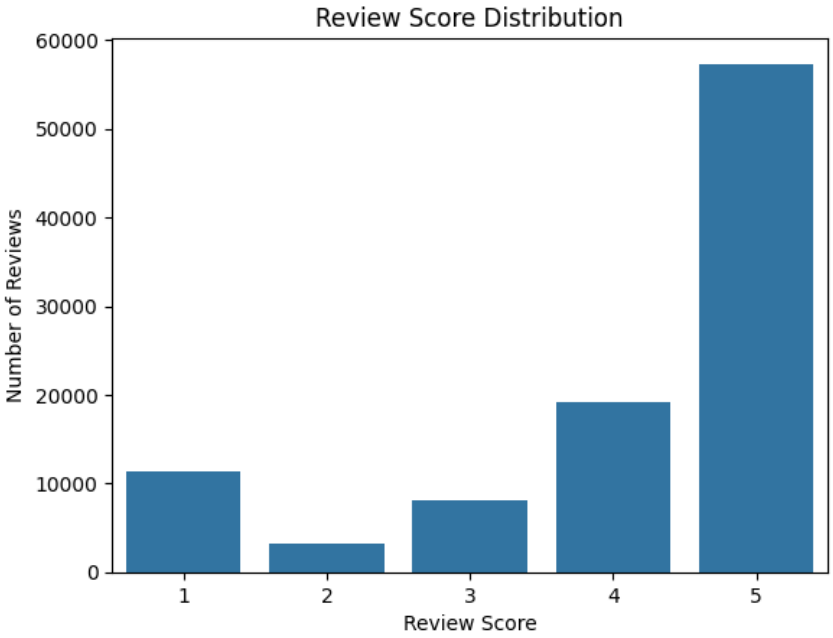
In this query, we chose to exclude “not_defined” from payment types. In hindsight, this could probably be done in PowerBI instead, because there might be a scenario where you would want to see that type as well.

Review scores in a bar plot

Review scores in a bar plot

```
query = """
        SELECT review_score, COUNT(*) AS total_reviews
        FROM fact_review
        GROUP BY review_score
        ORDER BY review_score DESC;
        """

result = runQuery(query)
review_scores_distribution = result

sns.barplot(data=result, x='review_score', y='total_reviews')
plt.title("Review Score Distribution")
plt.xlabel("Review Score")
plt.ylabel("Number of Reviews")
plt.show()
```

✓ 0.0s



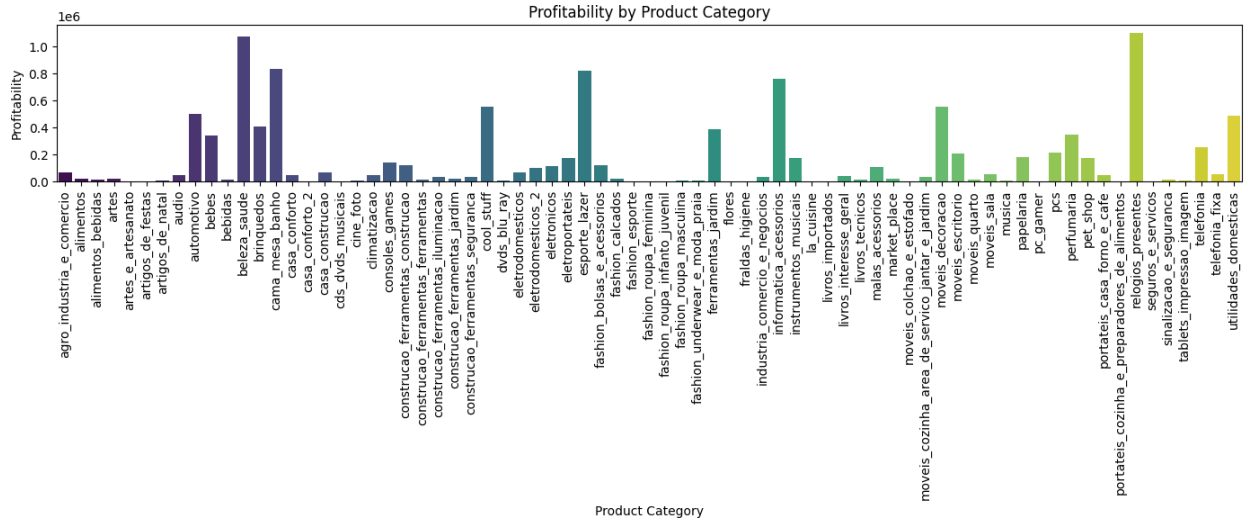
This is the first of the queries we visualize in jupyter. A bar plot is a pretty good fit for this kind of data. We also see that the dataset has an incredible amount of five star reviews. Useful for the PowerBI dashboard later.

Profitability by product category in a distribution plot

Profitability by product category in a bar plot

```
#
query = """
        SELECT
            dp.product_category_name,
            ROUND(SUM(foi.price), 2) AS total_revenue,
            ROUND(SUM(foi.freight_value), 2) AS total_freight_cost,
            ROUND(SUM(foi.price - foi.freight_value), 2) AS profitability
        FROM fact_order_item foi
        JOIN dim_product dp
            ON foi.seq_product_sk = dp.seq_product_sk
        GROUP BY dp.product_category_name
        """

result = runQuery(query)
print(result.describe())

plt.figure(figsize=(14,6))
sns.barplot(data=result, x="product_category_name", y="profitability", palette="viridis")
plt.title("Profitability by Product Category")
plt.xlabel("Product Category")
plt.ylabel("Profitability")
plt.xticks(rotation=90)
plt.tight_layout()
plt.show()
```

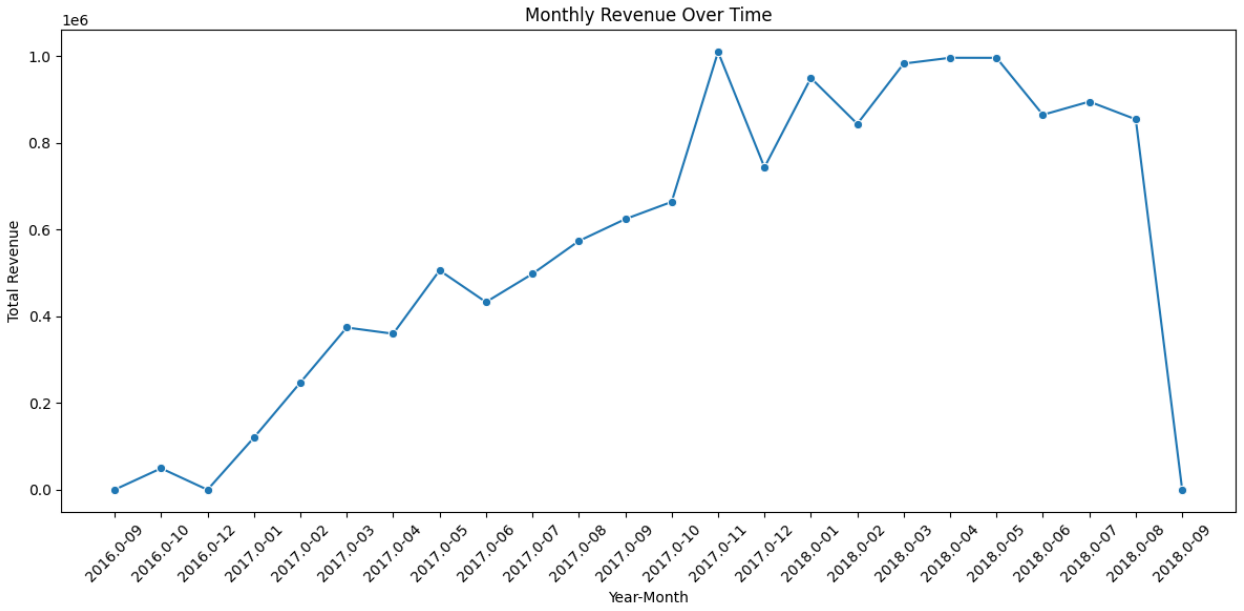
✓ 0.5s



Bar plot of product category profits. It shows the unique product categories on the x-axis and their profitability on the y-axis. With this plot, we can easily see which categories generate the most revenue.

Revenue by month in a line plot

```
# revenue by each month
query = """
        SELECT
            EXTRACT(YEAR FROM order_purchase_timestamp::timestamp) AS year,
            EXTRACT(MONTH FROM order_purchase_timestamp::timestamp) AS month,
            SUM(foi.price) AS total_revenue
        FROM fact_order fo
        JOIN fact_order_item foi
            ON fo.seq_order_sk = foi.seq_order_sk
        GROUP BY year, month
        ORDER BY year, month;
        """
result = runQuery(query)
revenue_by_month = result
# print(result)

result['year_month'] = result['year'].astype(str) + '-' + result['month'].astype(int).astype(str).str.zfill(2)
plt.figure(figsize=(12,6))
sns.lineplot(data=result, x='year_month', y='total_revenue', marker='o')
plt.xticks(rotation=45)
plt.title('Monthly Revenue Over Time')
plt.xlabel('Year-Month')
plt.ylabel('Total Revenue')
plt.tight_layout()
plt.show()
✓ 0.3s
```



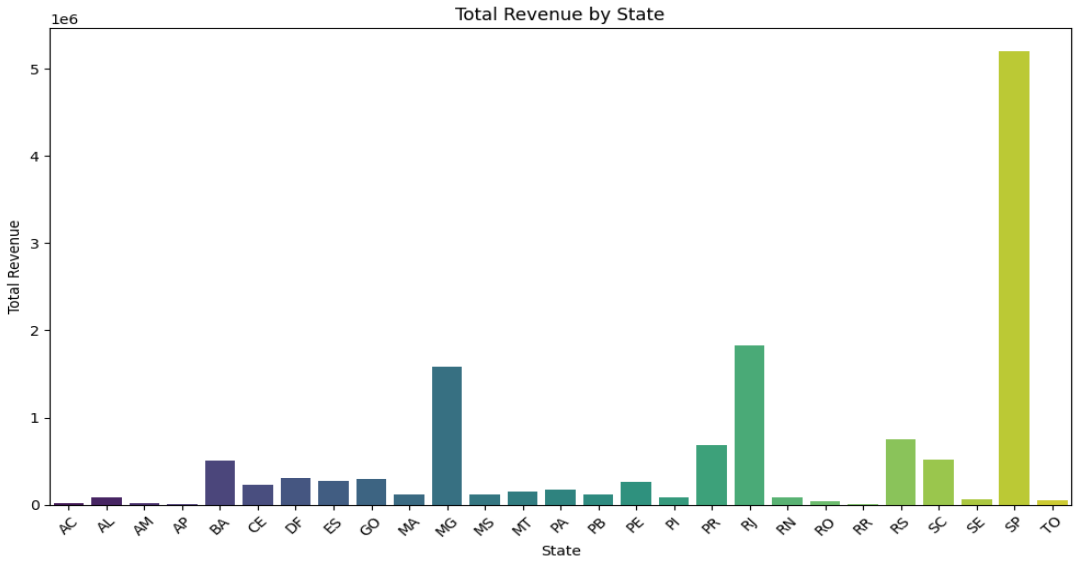
Next up is the monthly revenue over time. The x-axis represents time, while the y-axis is revenue for that particular month. We can see a good development along the time-axis, and the reason for the dip is most likely that the data for the last month is incomplete.

Revenue by state in a bar plot

Revenue by state in a bar plot

```
# revenue by state
query = """
        SELECT
            dc.customer_state AS state,
            SUM(foi.price) AS total_revenue
        FROM fact_order fo
        JOIN dim_customer dc
            ON fo.seq_customer_sk = dc.seq_customer_sk
        JOIN fact_order_item foi
            ON fo.seq_order_sk = foi.seq_order_sk
        GROUP BY dc.customer_state
        ORDER BY dc.customer_state
        -- LIMIT 10;
        """
result = runQuery(query)
revenue_by_state = result
# print(result)

plt.figure(figsize=(10,6))
sns.barplot(data=result, x='state', y='total_revenue', palette='viridis')
plt.title('Total Revenue by State')
plt.xlabel('State')
plt.ylabel('Total Revenue')
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```

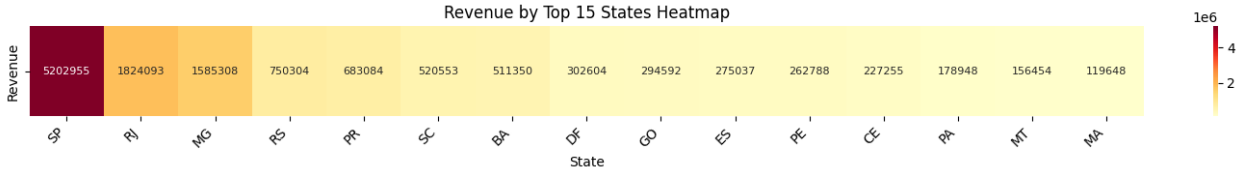
✓ 0.3s



Here we see the different states of Brazil along the x-axis, with their revenue on the y-axis. We can see that the state Sao Paulo generates by far the most revenue.

Revenue by state in a heatmap

```
plt.figure(figsize=(15,2))
sns.heatmap(
    np.array([top_states['total_revenue']]),
    annot=True,
    fmt=".0f",
    cmap='YlOrRd',  # <-- brighter color map
    xticklabels=top_states['state'],
    yticklabels=['Revenue'],
    annot_kws={"size":8}
)
plt.title('Revenue by Top 15 States Heatmap')
plt.xlabel('State')
plt.ylabel('')
plt.xticks(rotation=45, ha='right', fontsize=10)
plt.tight_layout()
plt.show()
```

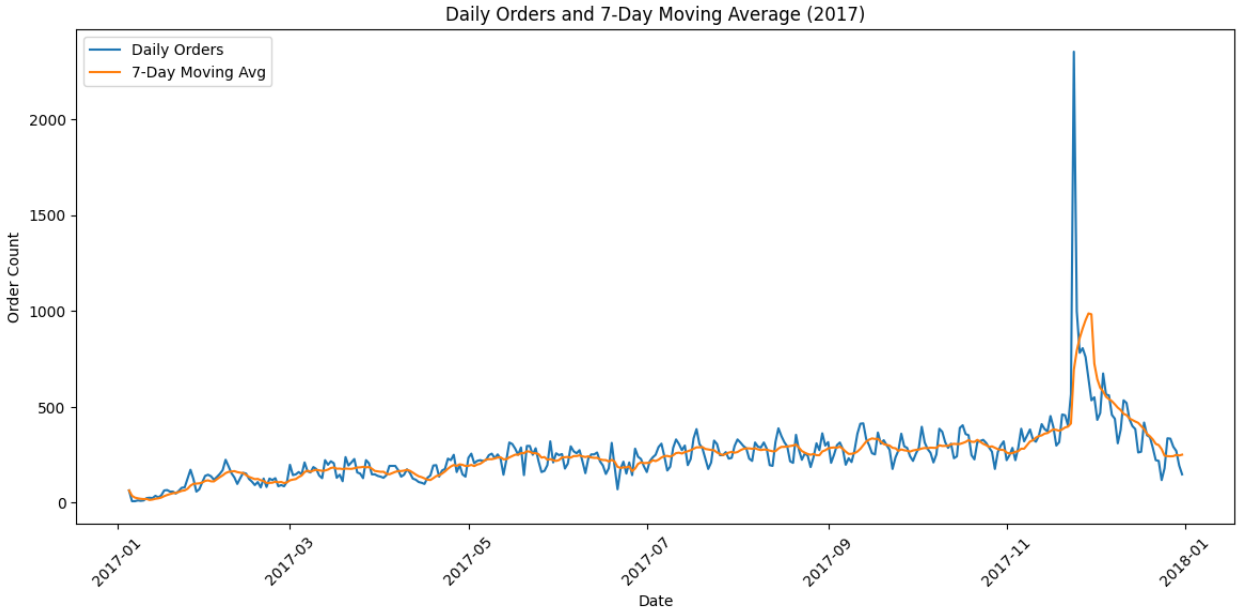


An alternative could be to use a heatmap to display the differences using colors instead. Other than that, it's the same data as the previous query.

Amount of orders and moving average + cumulative amount in a plot

```
    # this is a better answer for the task, as we properly
    # user the AVG() and OVER() functions for moving averages
    # this query shows us the daily amount of orders in 2017,
    # and "moving average" which tells us the average of the last seven days
    query = f"""
            -- CTE for daily amount of orders
            WITH daily_orders AS (
                SELECT
                    DATE(fo.order_purchase_timestamp) AS order_date,
                    COUNT(*) AS daily_order_count
                FROM fact_order fo
                WHERE EXTRACT(YEAR FROM fo.order_purchase_timestamp::timestamp) = 2017
                GROUP BY DATE(fo.order_purchase_timestamp)
            )
            SELECT
                order_date,
                daily_order_count,
                -- 7 day moving average of daily order count
                ROUND(AVG(daily_order_count) OVER (
                    ORDER BY order_date
                    ROWS BETWEEN 6 PRECEDING AND CURRENT ROW
                ), 2) AS moving_avg_week,

                -- cumulative total orders up to current date
                --
                SUM(daily_order_count) OVER (
                    ORDER BY order_date
                ) AS cumulative_order_count
            FROM daily_orders
            ORDER BY order_date;
            """

    result = runQuery(query)
    daily_amount_of_orders_w_moving_avg = result
    # print(result)
```

```
fig, ax1 = plt.subplots(figsize=(14,6))

ax1.plot(result['order_date'], result['daily_order_count'], label='Daily Orders', color='tab:blue')
ax1.plot(result['order_date'], result['moving_avg_week'], label='7-Day Moving Avg', color='tab:orange')
ax1.set_xlabel('Date')
ax1.set_ylabel('Order Count')
ax1.legend(loc='upper left')
ax1.set_title('Daily Orders and 7-Day Moving Average (2017)')
plt.xticks(rotation=45)
✓ 0.2s
```

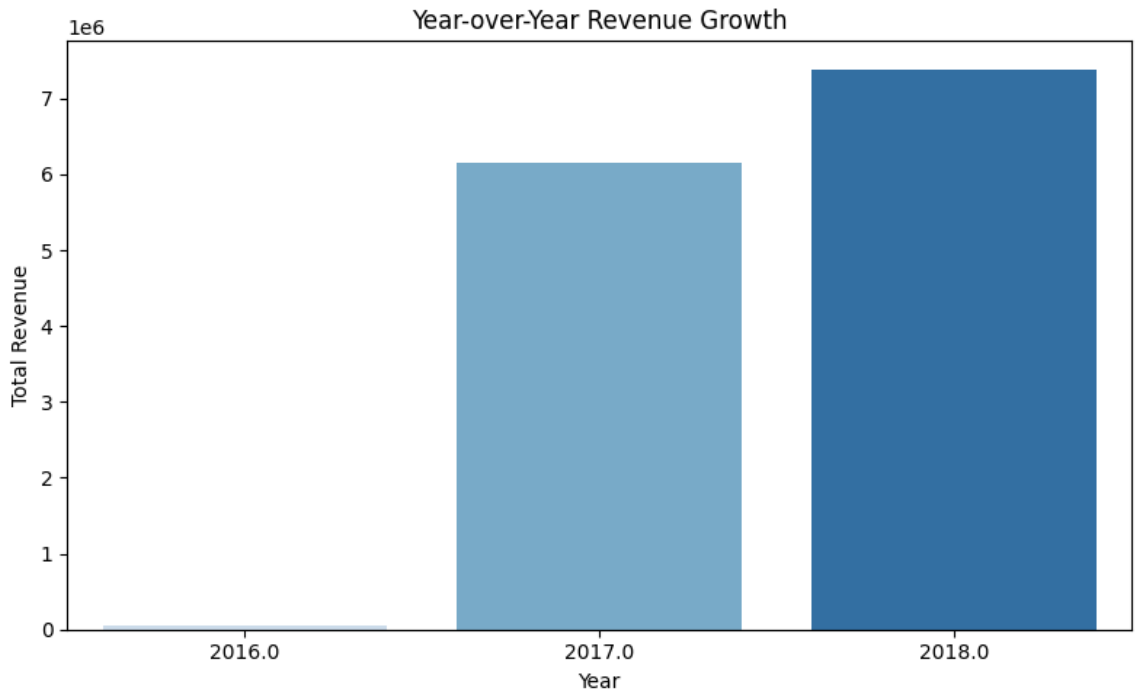


This interesting graph shows time on the x-axis, and two different plots on the y-axis. We have the blue representing the number of orders that day, and orange for the seven-day moving average. The massive spike in sales might be an error, a merging of companies or just a phenomenal day at the office. For sure an interesting graph to include in a dashboard.

Year-over-year revenue growth in a bar plot

Year-over-year revenue growth in a bar plot

```
# year-over-year revenue
query = """
        SELECT
            EXTRACT(YEAR FROM order_purchase_timestamp::timestamp) AS year,
            SUM(foi.price) AS total_revenue
        FROM fact_order fo
        JOIN fact_order_item foi
            ON fo.seq_order_sk = foi.seq_order_sk
        GROUP BY year
        ORDER BY year;
        """

result = runQuery(query)
year_over_year_revenue = result
# print(result)

plt.figure(figsize=(8,5))
sns.barplot(data=result, x='year', y='total_revenue', palette='Blues')
plt.title('Year-over-Year Revenue Growth')
plt.xlabel('Year')
plt.ylabel('Total Revenue')
plt.tight_layout()
plt.show()
```

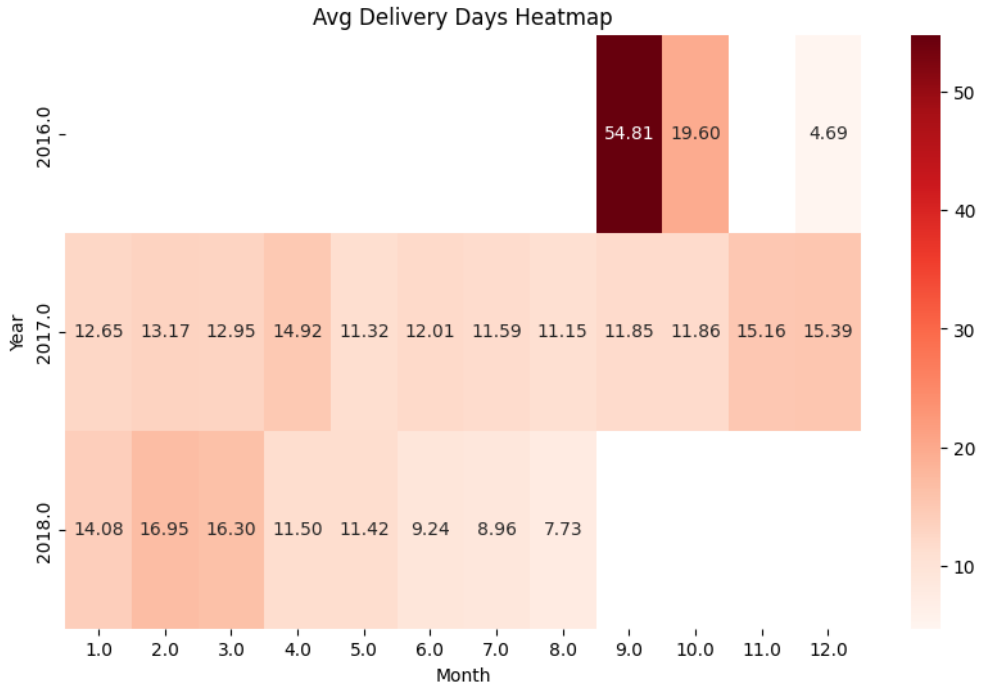


Don’t worry about the incredibly low values of 2016, as the dataset only contains data from the last couple of months. On the other hand, we see an increase in revenue, a growth from 2017 to 2018.

Average delivery time and ratio for on-time delivery in a heatmap

Average delivery time and ratio for on-time delivery in a heatmap

```
# query returns average delivery time and a ratio for on-time delivery
# ordered by year, month so we can see progress and eventually use it in
# a dashboard for business intelligence purposes
# it extracts total seconds with EPOCH() and turns them into days by
# dividing by (60s * 60m * 24h)
# this is then averaged and rounded and aliased as avg_delivery_days
# for the ratio, we use CASE to return a 1 when orders are delivered on time
# otherwise we get a 0
# we sum and and divide by count to get the ratio
query = """
        SELECT
            EXTRACT(YEAR FROM fo.order_purchase_timestamp::timestamp) AS year,
            EXTRACT(MONTH FROM fo.order_purchase_timestamp::timestamp) AS month,
            ROUND(AVG(EXTRACT(EPOCH FROM (fo.order_delivered_customer_date - fo.order_purchase_timestamp::timestamp)) / (60*60*24)),
            ROUND(100.0 * SUM(CASE WHEN fo.order_delivered_customer_date <= fo.order_estimated_delivery_date THEN 1 ELSE 0 END)
                / COUNT(*), 2) AS on_time_delivery_percent
        FROM fact_order fo
        WHERE fo.order_status = 'delivered'
        GROUP BY year, month
        ORDER BY year, month;
        """
result = runQuery(query)
# print(result)

pivot = result.pivot(index='year', columns='month', values='avg_delivery_days')
plt.figure(figsize=(10,6))
sns.heatmap(pivot, annot=True, fmt=".2f", cmap='Reds')
plt.title('Avg Delivery Days Heatmap')
plt.xlabel('Month')
plt.ylabel('Year')
plt.show()
```



We tried a heatmap of each month, which worked out pretty well for this task. We can see a strong dark red on the worst month, where the average delivery time was very high relative to the other months. The issue with this visualization is that it does not have a place to display the ratio of on-time deliveries, which is arguably the most important statistic from this query.

Exports

```
import os

os.makedirs("results", exist_ok=True)

# List all result variables to export for descriptive analytics
descriptive_results = {
    "revenue_by_month": "results/revenue_by_month.csv",
    "revenue_by_state": "results/revenue_by_state.csv",
    "total_number_of_orders": "results/total_number_of_orders.csv",
    "ten_biggest_categories": "results/ten_biggest_categories.csv",
    "revenue_by_payment_type": "results/revenue_by_payment_type.csv",
    "review_scores_distribution": "results/review_scores_distribution.csv",
    "daily_amount_of_orders_w_moving_avg": "results/daily_amount_of_orders_w_moving_avg.csv",
    "year_over_year_revenue": "results/year_over_year_revenue.csv",
    "avg_delivery_time_and_ratio": "results/avg_delivery_time_and_ratio.csv"
}

# for-loop in a try-catch to export each of the elements
# in the object defined above
try:
    for var, path in descriptive_results.items():
        if var in globals():
            globals()[var].to_csv(path, index=False)
        else:
            print(f"{var} not found.")
    print("Export done.")
except Exception as e:
    print("Error during export:", e)
```

```
Export done.
```

Exports are done the same way here, as in the “task_2_queries” section. The only change is the directory, which changed to “results” instead of “exports” due to this exams file structure.

Predictive Analytics

This analysis grouped customers into clusters based on their purchasing behavior to be able to predict their potential future behavior.

Perspective Analytics:

In this analysis we find recommendations for each customer based on what they have purchased before. This will make it easier to plan which products to advertise, put on sale, give coupons or prioritize etc. for each customer.

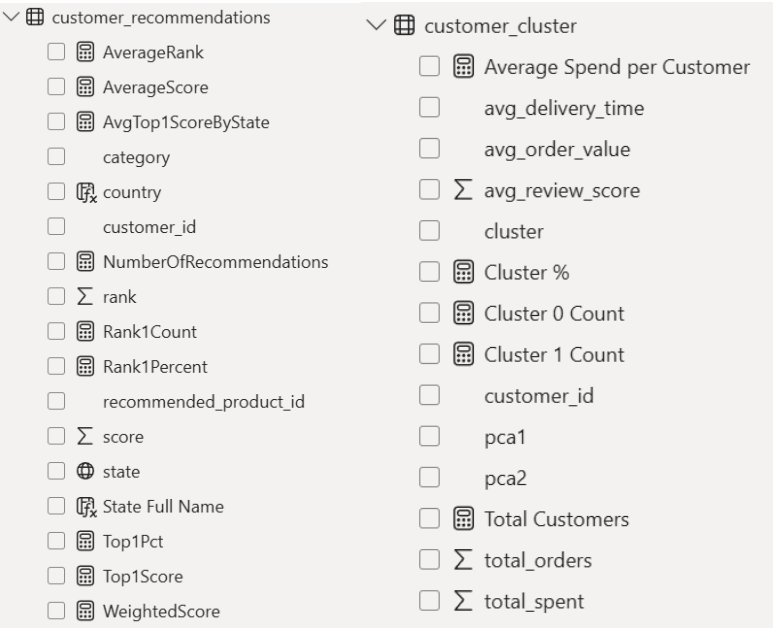
Task 3

Dashboard

The dashboard is based on the result of the 3 analysis we made in python, descriptive, predictive and prescriptive. We have chosen to create different dashboard pages to give more separation in the dashboard.

Calculated measures (DAX)

We created several calculations to help support our dashboard analysis. Here is an overview:



Prescriptive

Gives the average rank for recommendations

```
AverageRank = AVERAGE(customer_recommendations[rank])
```

Gives the average score for recommendations

```
AverageScore = AVERAGE(customer_recommendations[score])
```

Gives the average score for recommendations by the best rank (1)

```
AvgTop1ScoreByState =
CALCULATE(
    AVERAGE(customer_recommendations[score]),
    customer_recommendations[rank] = 1
)
```

Counts the total number of recommendations

```
NumberOfRecommendations = COUNTROWS(customer_recommendations)
```

Counts all the rank 1

```
Rank1Count =
CALCULATE(
    COUNTROWS(customer_recommendations),
    customer_recommendations[rank] = 1
)
```

Gives the percentage of rank 1

```
Rank1Percent =
DIVIDE(
    COUNTROWS(FILTER(customer_recommendations, customer_recommendations[rank] = 1)),
    COUNTROWS(customer_recommendations)
)
```

More advanced, gives a weighted score based on rank and score

```
WeightedScore =
DIVIDE(
    SUMX(customer_recommendations, customer_recommendations[score] /
customer_recommendations[rank]),
    COUNTROWS(customer_recommendations)
)
```

Prediction:

Average money spent per customer

```
Average Spend per Customer =
DIVIDE(
    SUM('customer_cluster'[total_spent]),
    DISTINCTCOUNT('customer_cluster'[customer_id]),
    0
)
```

Cluster percentage

```
Cluster % =
DIVIDE(
    COUNTROWS('customer_cluster'),
    [Total Customers],
    0
)
```

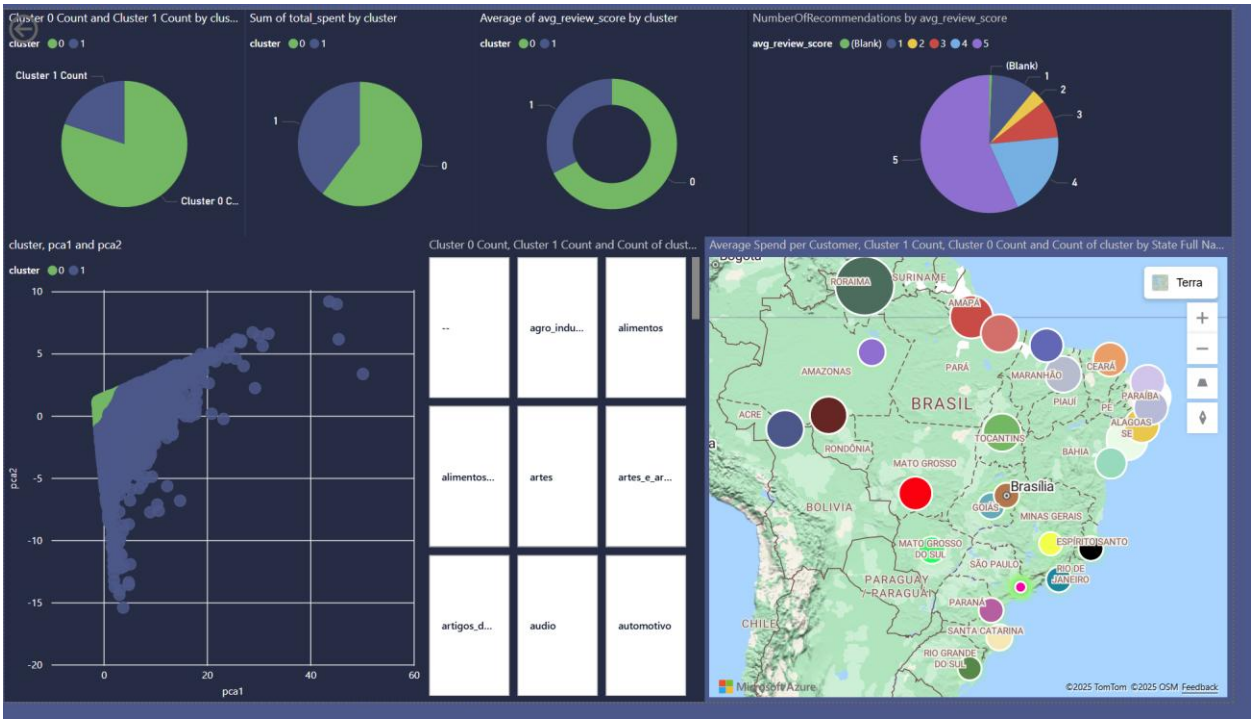
Cluster count

```
Cluster 0 Count = COUNTROWS(FILTER('customer_cluster', 'customer_cluster'[cluster] = 0))
Cluster 1 Count = COUNTROWS(FILTER('customer_cluster', 'customer_cluster'[cluster] = 1))
```

Total amount of customers

```
Total Customers = COUNTROWS('customer_cluster')
```

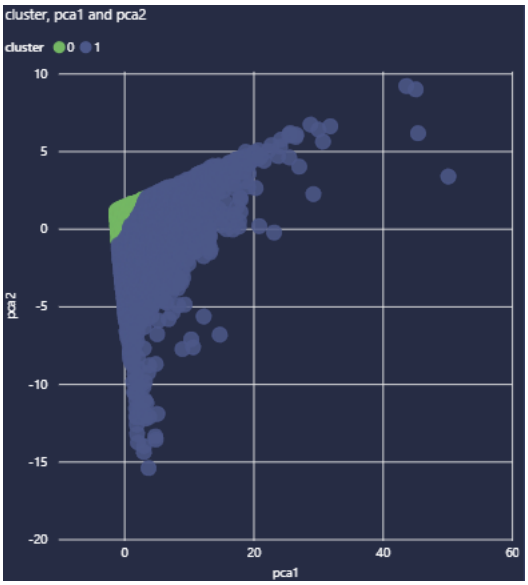
Predictive page



This page is for predicting the behavior of customers, the data is taken from the predictive analysis. On this page we look at the two different clusters of customers, and what the difference is between them. This will help us identify how customers will spend their money. Each chart has drill-down capabilities, so by clicking on the charts you can drill down in the data.

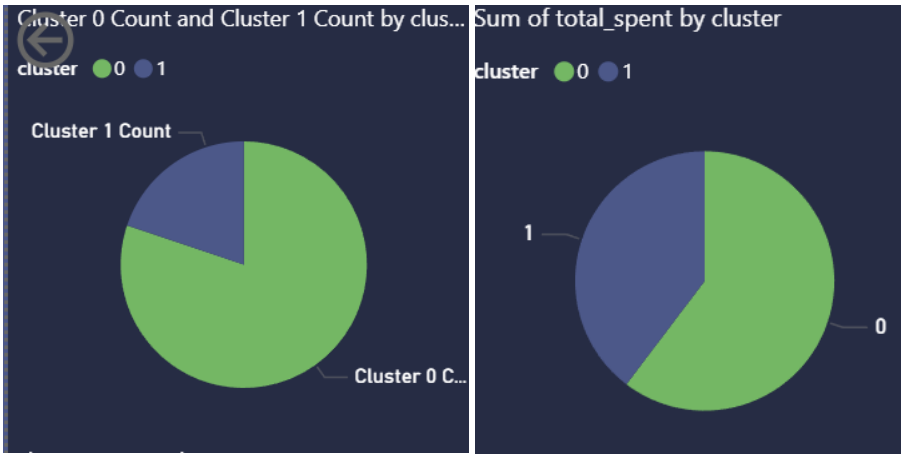
Cluster

We begin by looking at the scatter chart, this shows us the visual cluster result from the python analysis. Looking at the chart in isolation isn't very useful, however, approaching the analysis of other charts and reports from the dashboard, we can gain deeper knowledge about our analysis. It does however show us that cluster 0 is very concentrated, and that cluster 1 is more spread.



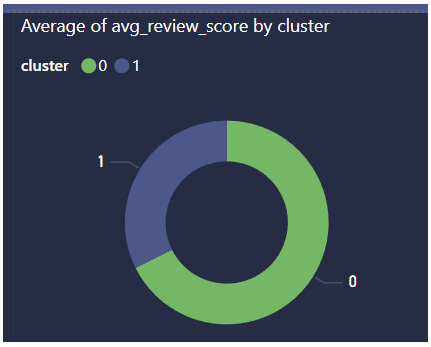
Count and total spent by cluster

These charts give us valuable insight into the two clusters. The left represents total count of customers in each cluster, where we can see that cluster 0 has 77 340 (80%) and cluster 1 has 19 138 (20%). And on the right, we can see total money spent on each cluster. This is very interesting because even though cluster 0 has 80% of the customers, it only has 60% of the total money spent.



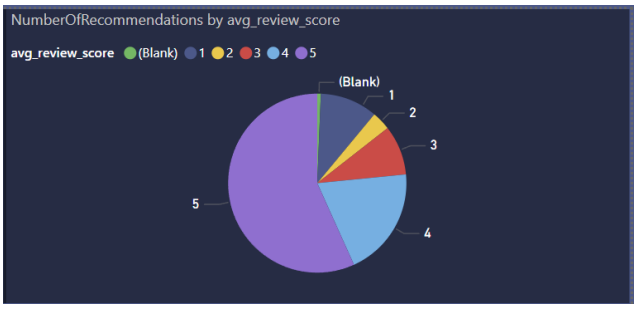
Average review by cluster

A donut chart that gives us the average review score by each cluster. If we filter only Cluster 1, we can see that we received mostly bad reviews. On the other hand, cluster 0 gave mostly very positive feedback.



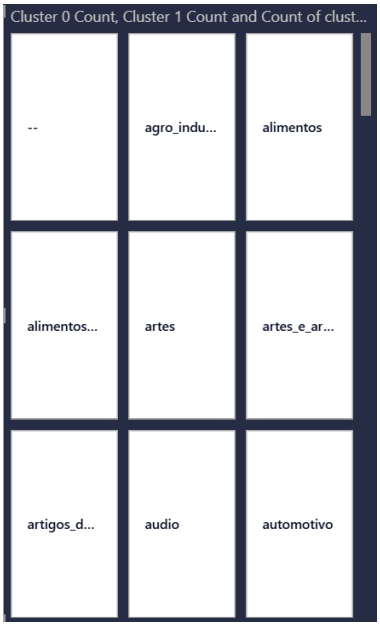
Number of Recommendations by average review score

In this pie chart, we can see the number of recommendations by average review score of our customers from dataset. With this analysis, we can analyze the reviews that we received from our customers by review score. It also shows difference between the clusters if we filter by cluster.



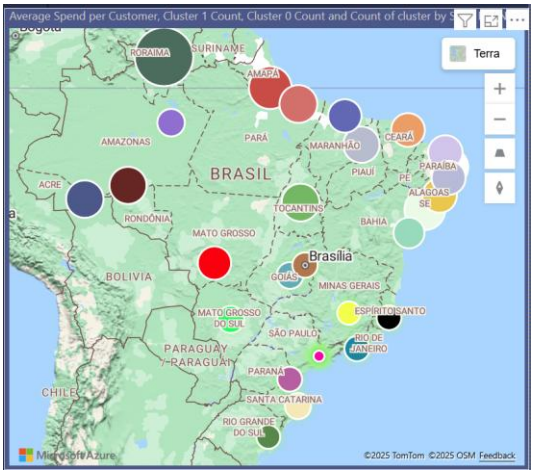
Catergory filter

This table represents interactive buttons that are connected to all the others reports in. By clicking on any button, we can see a drill-down analysis of clusters from the dataset, and we can look more closely at how each clusters have different patterns in the categories.



Interactive map

We also added an interactive map where we can see the analysis of average spending of customers in each state in Brazil. If we want to visualize data only for Cluster 0, we can filter it in one of the top charts and get the results. Comparing these results with our review chart, we can see that customers from cluster 1 gave us negative reviews but at the same time they spent the most on their shoppings.

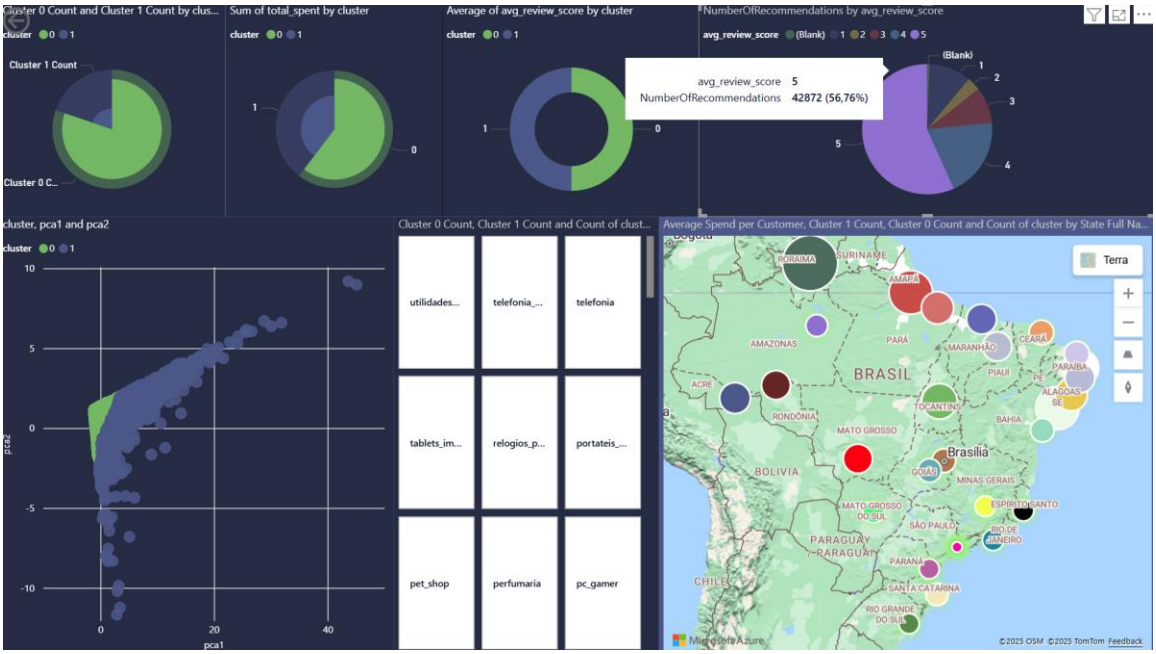


There was some problem getting the interactive map to work, so we made a new column named Full State Name, and then we got AI to get the two letter names to full ones. Otherwise, it was showing states in other countries, for example, PA was Pennsylvania in the US.

```
State Full Name =
SWITCH(
    TRUE(),
    'customer_recommendations'[State] = "AC", "Acre",
    'customer_recommendations'[State] = "AL", "Alagoas",
    'customer_recommendations'[State] = "AM", "Amazonas",
    'customer_recommendations'[State] = "AP", "Amapá",
    'customer_recommendations'[State] = "BA", "Bahia",
    'customer_recommendations'[State] = "CE", "Ceará",
    'customer_recommendations'[State] = "DF", "Distrito Federal",
    'customer_recommendations'[State] = "ES", "Espírito Santo",
    'customer_recommendations'[State] = "GO", "Goiás",
    'customer_recommendations'[State] = "MA", "Maranhão",
    'customer_recommendations'[State] = "MT", "Mato Grosso",
    'customer_recommendations'[State] = "MS", "Mato Grosso do Sul",
    'customer_recommendations'[State] = "MG", "Minas Gerais",
    'customer_recommendations'[State] = "PA", "Pará",
    'customer_recommendations'[State] = "PB", "Paraíba",
    'customer_recommendations'[State] = "PR", "Paraná",
    'customer_recommendations'[State] = "PE", "Pernambuco",
    'customer_recommendations'[State] = "PI", "Piauí",
    'customer_recommendations'[State] = "RJ", "Rio de Janeiro",
    'customer_recommendations'[State] = "RN", "Rio Grande do Norte",
    'customer_recommendations'[State] = "RS", "Rio Grande do Sul",
    'customer_recommendations'[State] = "RO", "Rondônia",
    'customer_recommendations'[State] = "RR", "Roraima",
    'customer_recommendations'[State] = "SC", "Santa Catarina",
    'customer_recommendations'[State] = "SP", "São Paulo",
    'customer_recommendations'[State] = "SE", "Sergipe",
    'customer_recommendations'[State] = "TO", "Tocantins",
    BLANK()
)
```

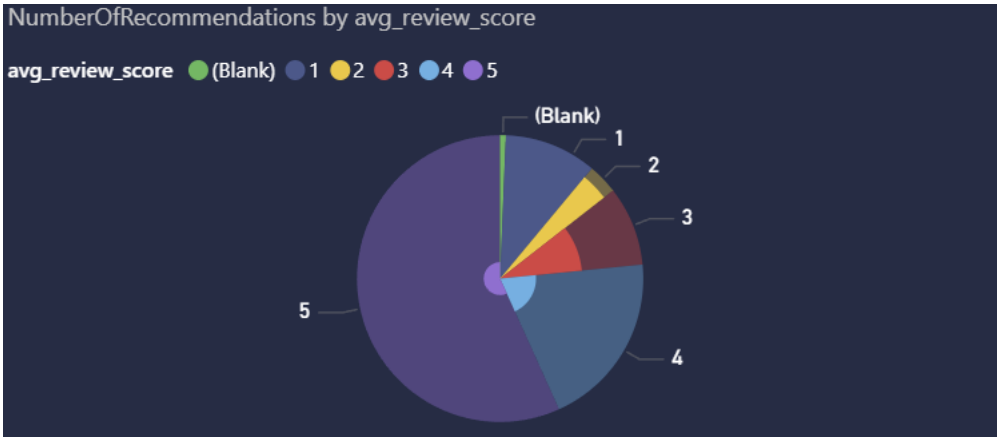
Page demo

In this example we are clicking on the top right pie chart, we click on the average score 5. Now we see all the other charts change based on this.

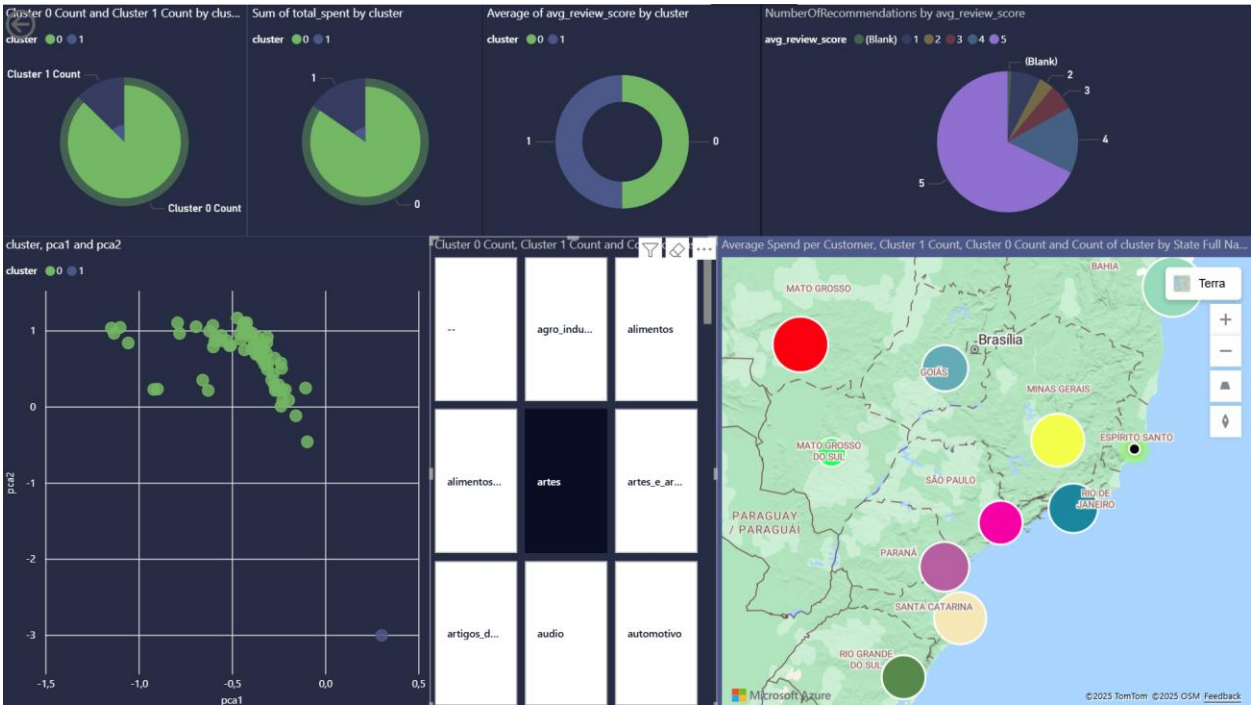


The two charts on the top left show us that the overwhelming majority of cluster 0 gives 5 stars for their products, and the opposite is true for cluster 1. We can also see that most of the money spent from cluster 0 is products that were given 5 stars, whereas around half of the total money spent by cluster 1 is on products given 5 stars.

Furthermore, if we only filter by cluster 1 we see another interesting result.

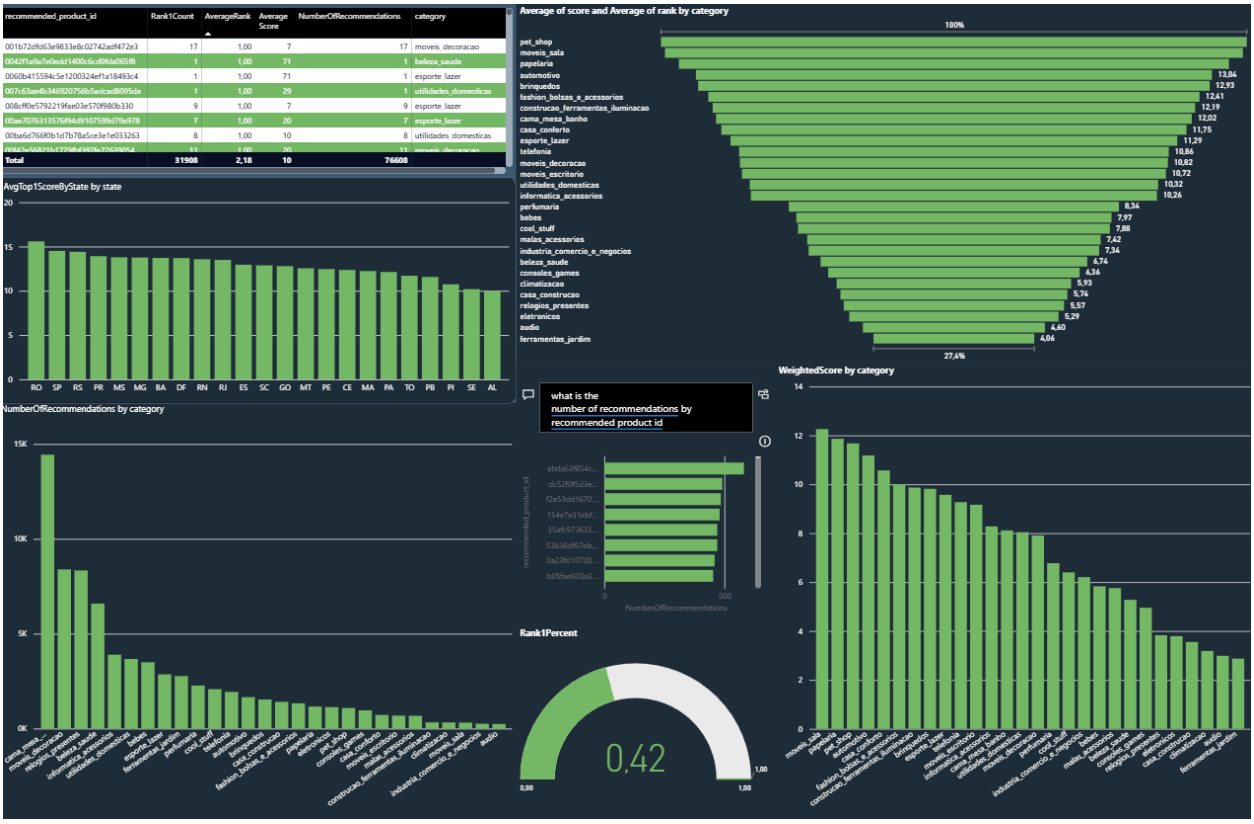


Here we can see that most of them give 1 star, followed by 2, 3 and 4, and then 5 stars are the least given. And if we click on each of the review score we can also see how much money spent by reviews. The reviews might be a bit misleading as it seems that the cluster 1 customers might be giving less reviews which also brings the score down.



Above we have selected the 5-star reviews and drilled further down by also selecting the category artes. Now we can get insight into the specific category by review. We also wanted to include this slicer in the other pages dynamically, but instead of showing the category it removed it.

Recommendation page:



In this page we try to give a clear picture of what to do next. The data is the output from our prescriptive analysis, we exported the data to an excel file. Here is how it looks after we transformed the score, we rounded it to two decimal places and multiplied it by 100 to make it more readable.

customer_id	recommended_product_id	rank	score	category	state
ffffa3172527f765de70084a7e53aae8	e7593e3c84b3302e1531a715f1ded8b2	1	16,01	moveis_decoracao	MG
ffffa3172527f765de70084a7e53aae8	8aa6223e400af9c97b07c75993142721	2	4,05	bebes	MG
ffff42319e9b2d713724ae527742af25	ba92b5a0701d2f820ba6ca8f8c86294f	3	11,18	cama_mesa_banho	SP
ffff42319e9b2d713724ae527742af25	9a0620ddaf3055b7e588e936e036a529	4	7,91	cama_mesa_banho	SP
ffff42319e9b2d713724ae527742af25	b91a91c07533369ef426985f93170068	2	14,43	cama_mesa_banho	SP
ffff42319e9b2d713724ae527742af25	ddb9028236525b8ab493fc24f99e8d1a	1	15,08	cama_mesa_banho	SP
fffb97495f78be80e2759335275df2aa	40a4dfa74a737d541b3c2413edf9cdf6	1	9,28	utilidades_domestic	MG
fffb97495f78be80e2759335275df2aa	09dafaa79c17733a900ce34c9a1f5be8	2	9,28	cama_mesa_banho	MG
fffb97495f78be80e2759335275df2aa	64dd98c857401579d7f88ddb97821b4c	5	9,28	cama_mesa_banho	MG
fffb97495f78be80e2759335275df2aa	51edddbbc47a477259e672bd291feed6	4	9,28	utilidades_domestic	MG

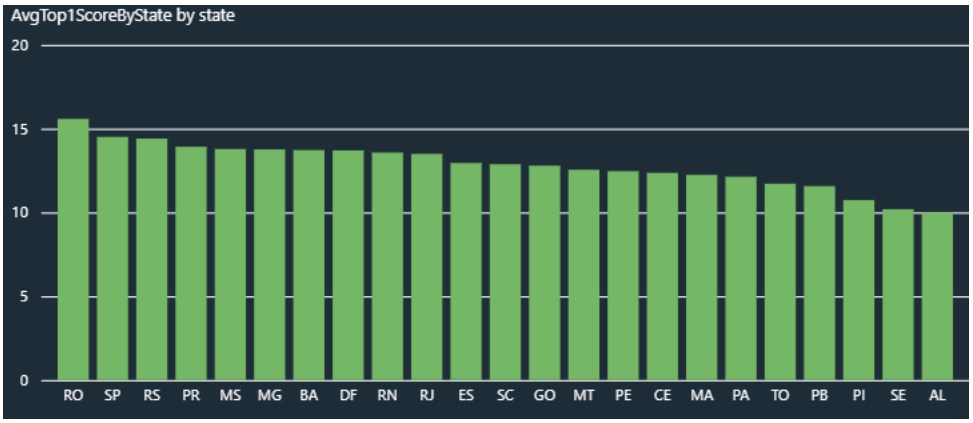
Recommended product table

recommended_product_id	Rank1Count	AverageRank	Average Score	NumberOfRecommendations	category
35afc973633aaeb6b877ff57b2793310	430	1,26	10	470	casa_conforto
7fb7c9580222a2af9eb7a95a6ce85fc5	429	1,01	2	430	utilidades_domesticas
a6be3b3eaae040735e86e4376ea77515	325	1,17	5	346	bebes
0fbf220f61720a67fc0d6432cb32a5b4	320	1,00	6	321	ferramentas_jardim
12667a519a4a3094eab02f5841b07409	314	1,12	3	334	relogios_presentes
87838c210d9c925acac2619548950502	295	1,11	6	306	moveis_decoracao
0b13080e2bf193c4ff096b09e0758c7d	277	1,12	6	288	moveis_decoracao
d979289e78afc0a6bee40e1f9dabe520	270	1,07	6	285	casa_construcao
Total	**31908**	**2,18**	**10**	**76608**	

This is just a table to show each individual recommended product, it's not very advanced and on the fields on the bottom (total) are not what's important. What makes this table very useful is it gives us valuable insight into a specific product. We can filter each of the columns, by default it will be on rank 1 count which means the product that has been recommended the most amount of time as rank 1.

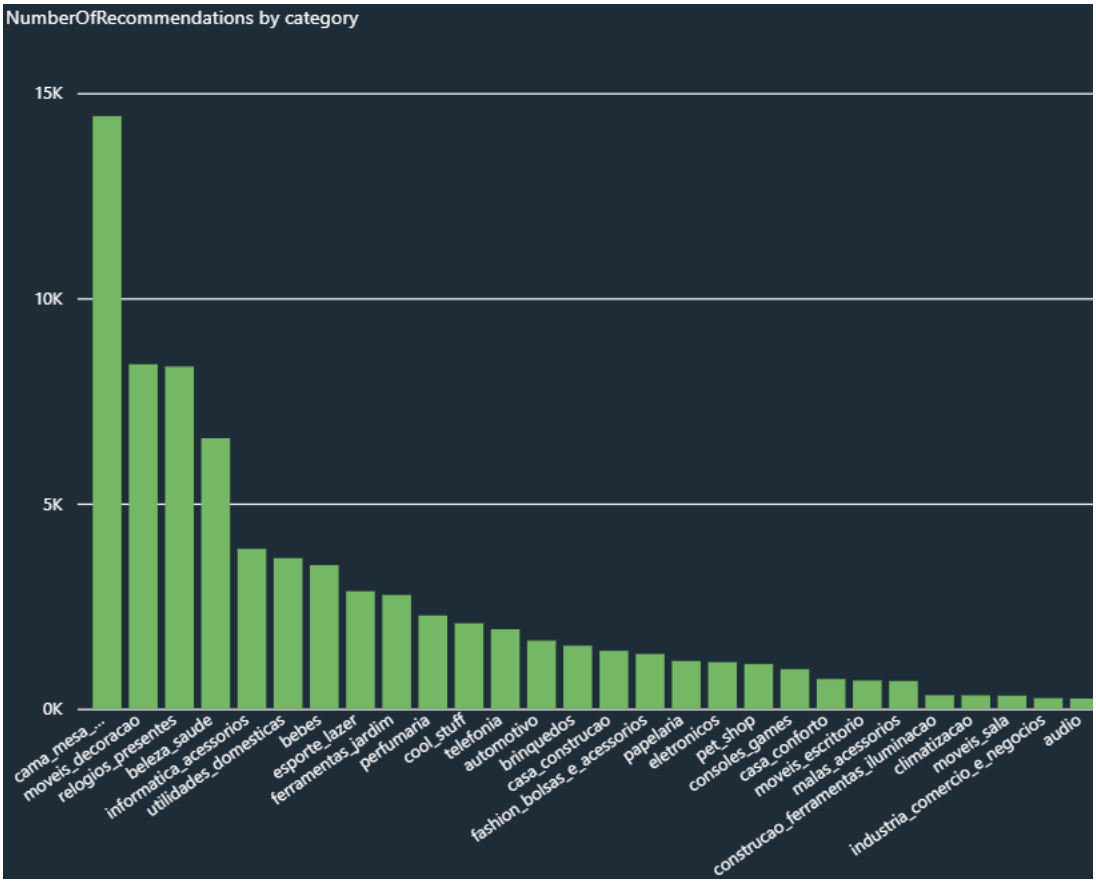
AvgTop1ScoreByState by state

This stacked column chart simply shows the average score of recommended products to customers but only the ones ranked as 1, and then it splits it in each state.



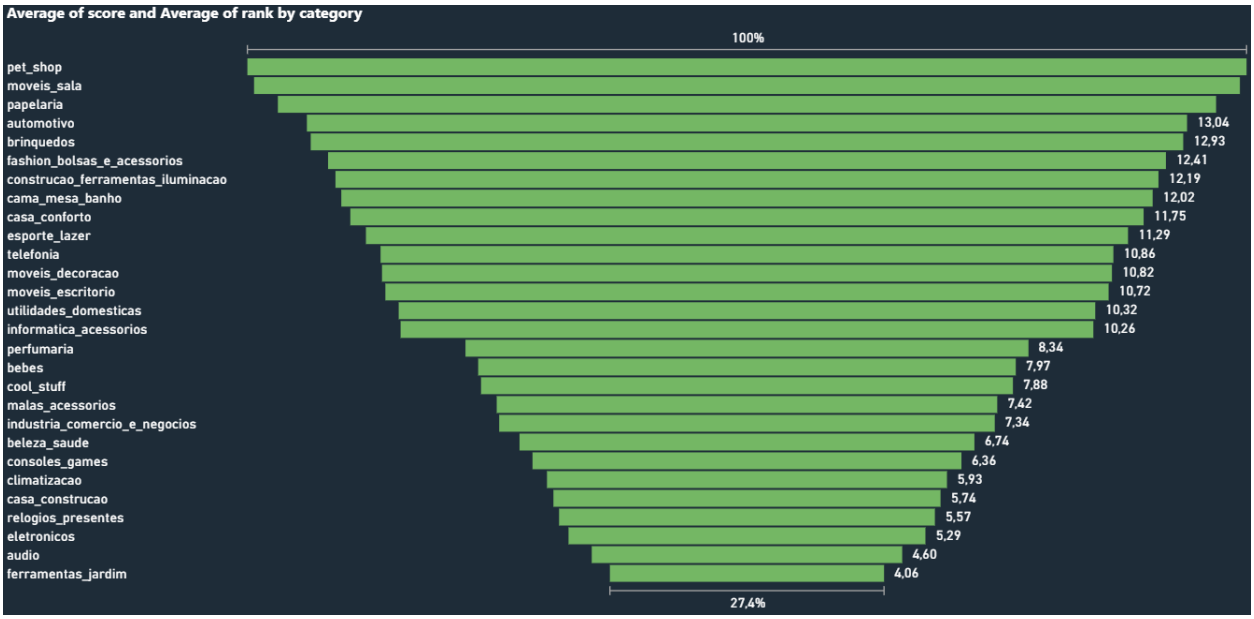
Number Of Recommendations by category

Here we have another stacked column chart, we can see the total of recommendations in each category, it includes all the recommendations, and it shows which categories it may be beneficial to invest in.



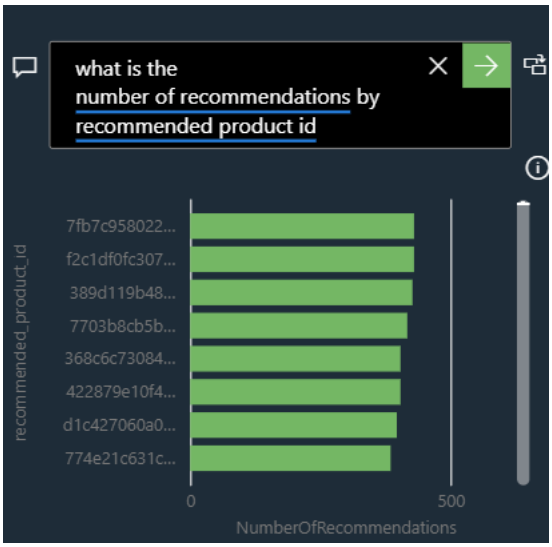
Average of score and Average of rank by category

This is a funnel visual, here we make it clearer which category on average has a high rank and score.



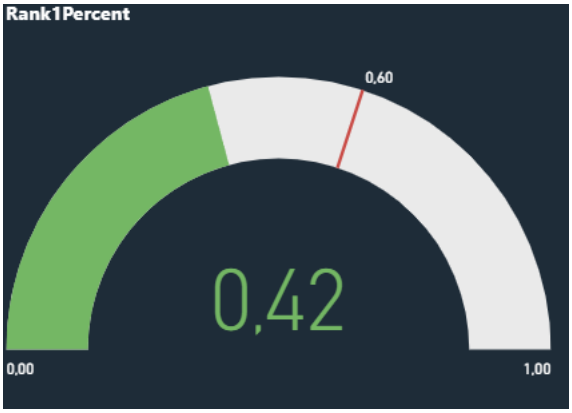
Q&A

This visual is a built in AI tool where you can ask questions, for now we have “what is the number of recommendations by recommended product id”. This one is very useful if you want insight on something that isn’t included in the dashboard. (Seems to not be possible to change color on all the text).



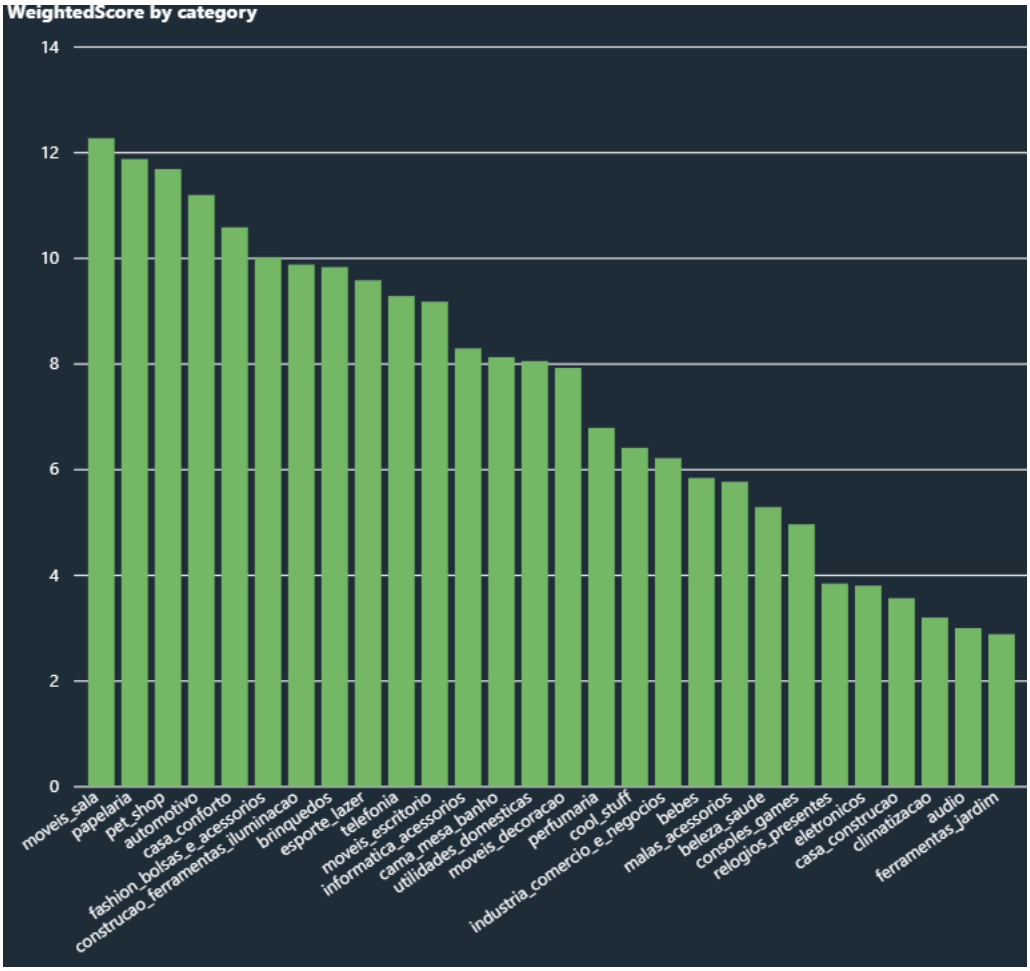
Rank1Percent

A gauge that shows the percentage of recommended rank 1, useful when clicking on different charts, as everything is interactive. For now, we have the target at 0.6, which could say that, we are interested in products or categories where the percentage of rank 1 is at least 60%. This can be adjusted based on needs. It goes from 0 to 1, where 1 represents 100%.



WeightedScore by category

A stacked column chart, it shows us more weight to top ranked recommendations, so we can compare categories/states more meaningfully.

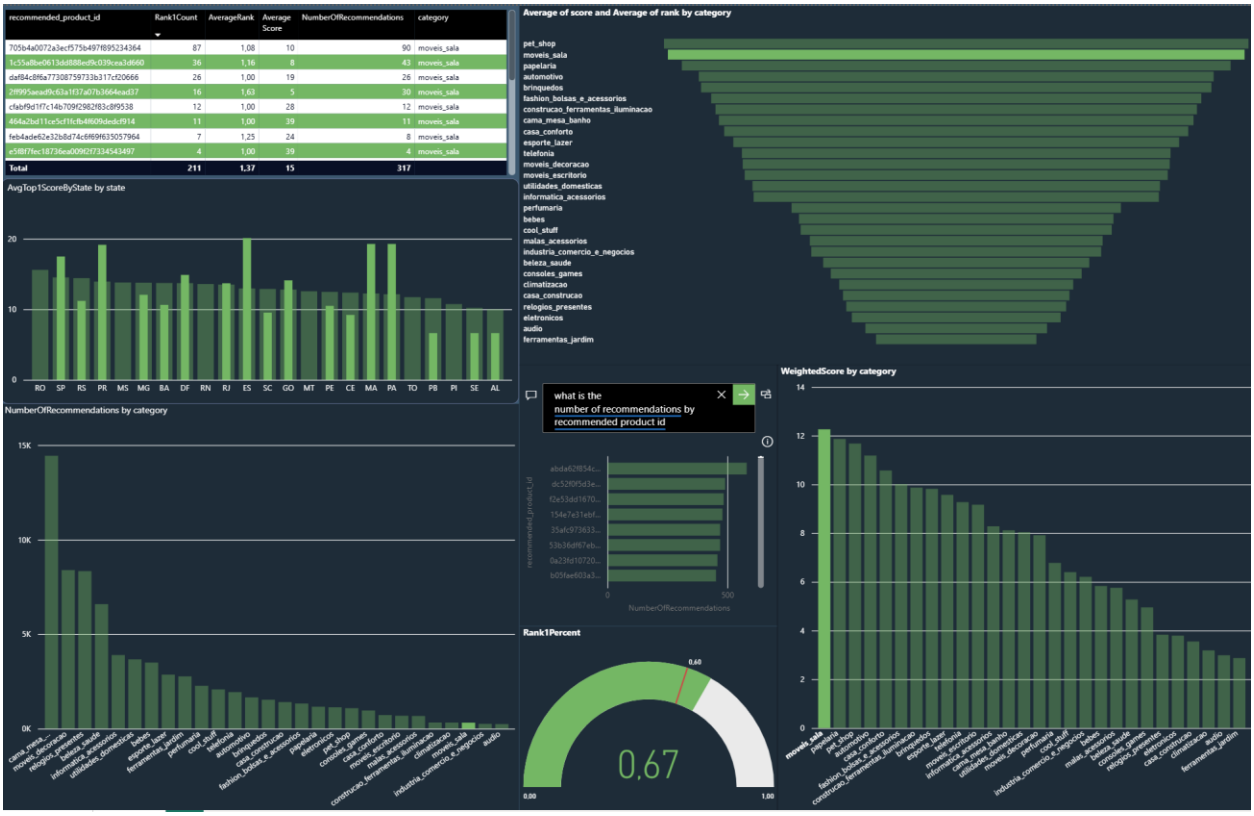


Page demo

Now let's look at some examples to show off the whole Page.

Example 1

I see that the highest column on the chart "WeightedScore by category" is very high, and I want to explore that more, so I click it.



Now we get an overview of the category "moveis_sala", we see in the gauge of Rank1Percent is at 67%, which means we hit the target we wanted. Then we look at the funnel chart, and we see the average score/rank and it's very high, which is good. But at the bottom on the page, we can see chart "NumberOfrecommendations by category" has a low number (only 317 recommendations). Above that we see which states it's recommended, and which is not. And finally, we see exactly which products have the most recommendations on the table, we can scroll through this if we want. Based on these results we can make a decision, maybe we want to push this category in specific states, or maybe we want to push just one or two of the products in this category. Either way, this dashboard gives us information to help us decide. It's also worth mentioning that we filtered out some categories that didn't contain enough recommendations, this can be toggled on and off or customized further if needed.

Example 2

In this example we clicked on the column with the highest value in the “NumberOfRecommendations by category” which is at the bottom left.

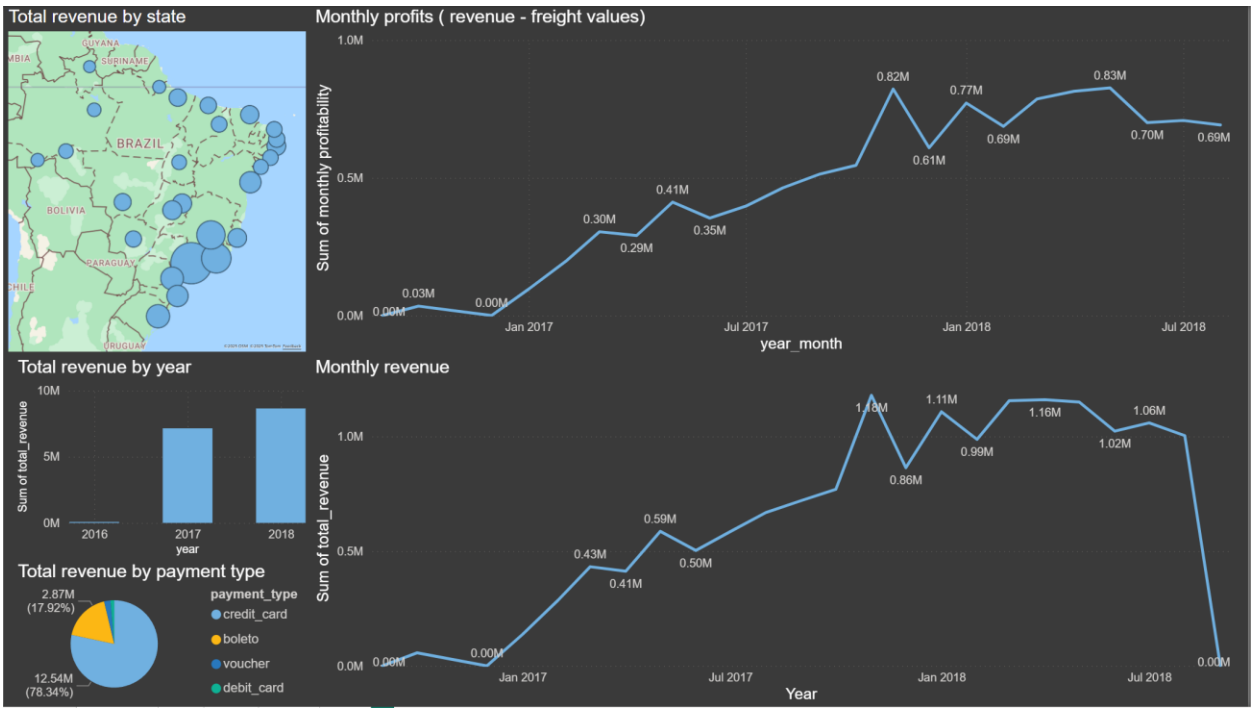
This is the category with the most overall recommendations, and what we see in the gauge is that 32% of them are ranked nr 1. This does not hit our target, but that doesn’t have to be a bad thing. Because of the high number of recommendations, it means that this category will potentially sell a lot of products but also means we may have to dig a bit deeper to find which products in this category we want to prioritize. So, what we can do now is look at the table in the top left corner, here we can look at each individual product in the category and get insight into the ones we wish. For this example, we click on the product with the most nr 1 rankings.

This will tell us that 99% of the recommendations for this product ranks at nr 1. Furthermore, we see which states to potentially focus on. If we wanted even more information, we could ask in the AI Q&A visual.

Descriptive page:

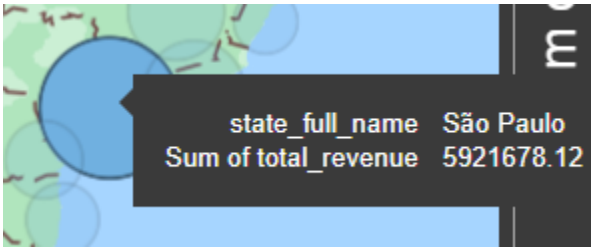
The descriptive analytics has been scattered among two – three unique pages. We have a page dedicated to revenues, and one for products. There is also one page dedicated to one massive graph for daily orders. The last-mentioned graph was difficult to place and ended up in a standalone page.

Revenues page:



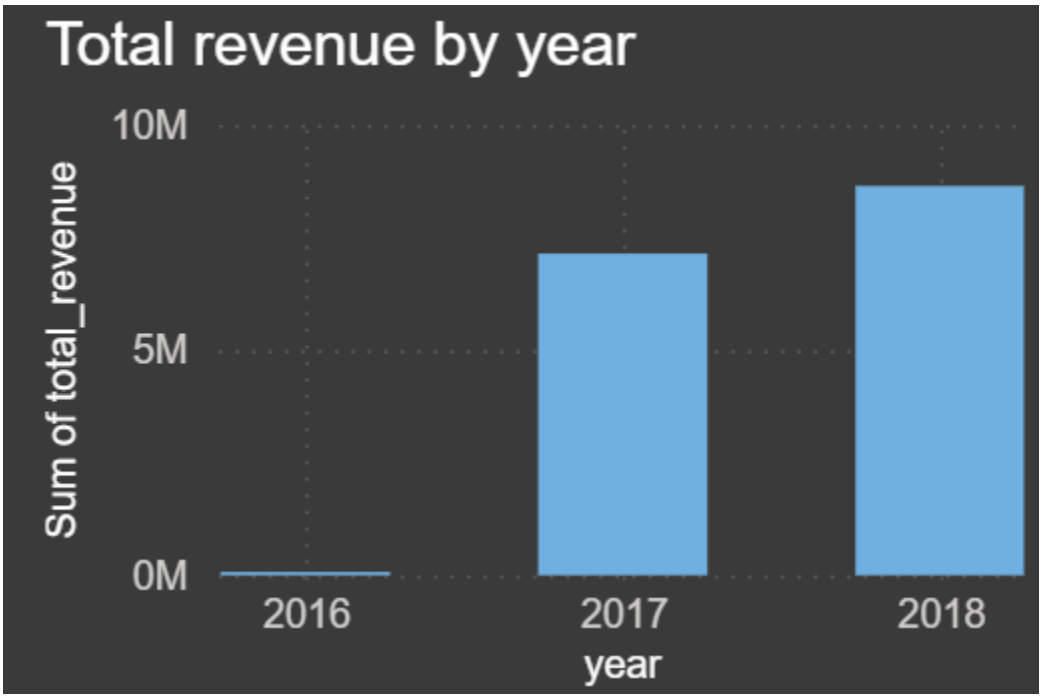
We will continue to go through each of these components one by one.

Total revenue by state



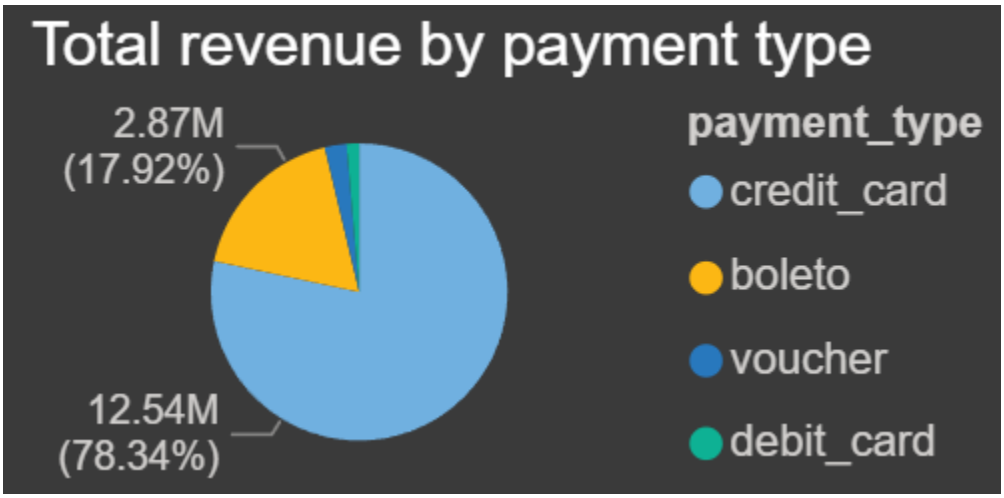
This component lets you see a visual of revenues generated by state on a map you can interact with. You can zoom in and out, mouse-drag to move or click on a bubble to see its revenue sum. As of now, it is a total of all years, but I imagine it should've been limited to a chosen period of time. You can also see the bubbles vary in size relative to their respective sums.

Total revenue by year



This is a simple bar plot of the yearly revenue. You can interact with it to see the specific numbers of the chosen year. Interacting with it also changes the dates on the x-axis of other graphs on the dashboard.

Total revenue by payment type

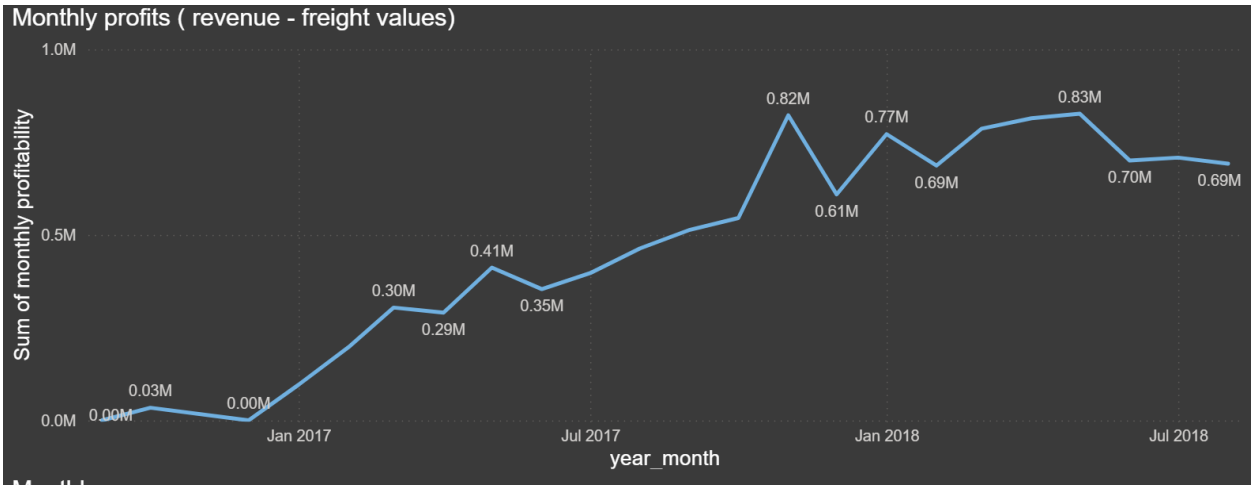


An interesting statistic in a pie chart.

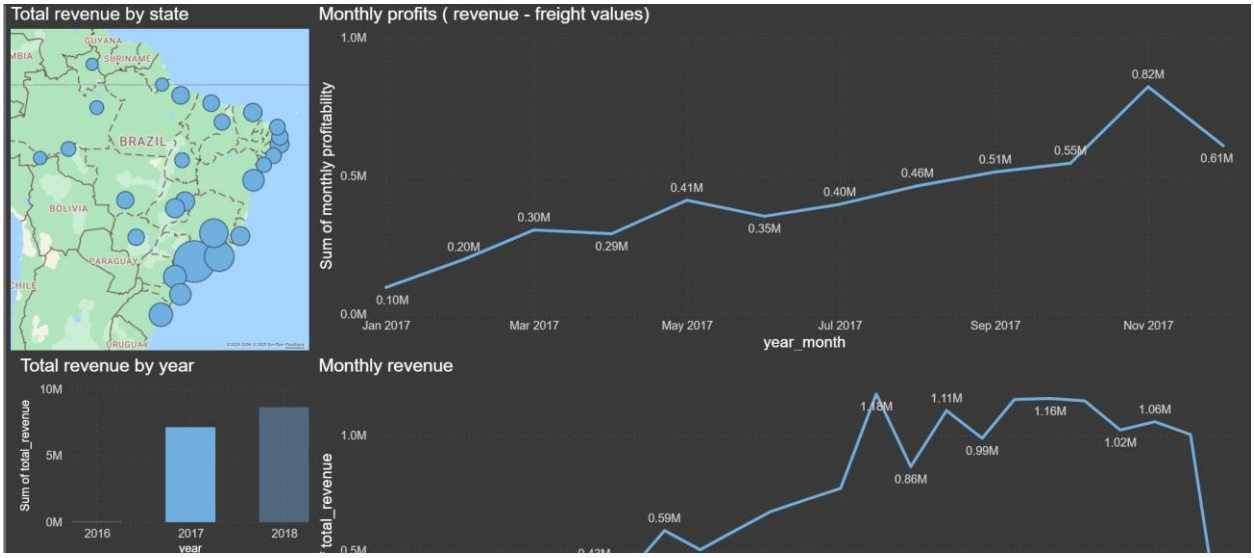
The pie chart shows the distribution of toral revenue by payment type. The chart shows that credit cards account for the majority of total revenue. Credit cards dominate with around 78% of the total revenue, while boleto accounts for approximately 18% of total revenue. While voucher and debit card represent a very small share of the pie chart. This indicates that most customers prefer cards while online shopping. For business, this suggests the importance of maintaining a secure and reliable credit card payment

system, while also exploring strategies to increase the use of alternative payment methods of diversify revenue streams.

Monthly profits and revenue



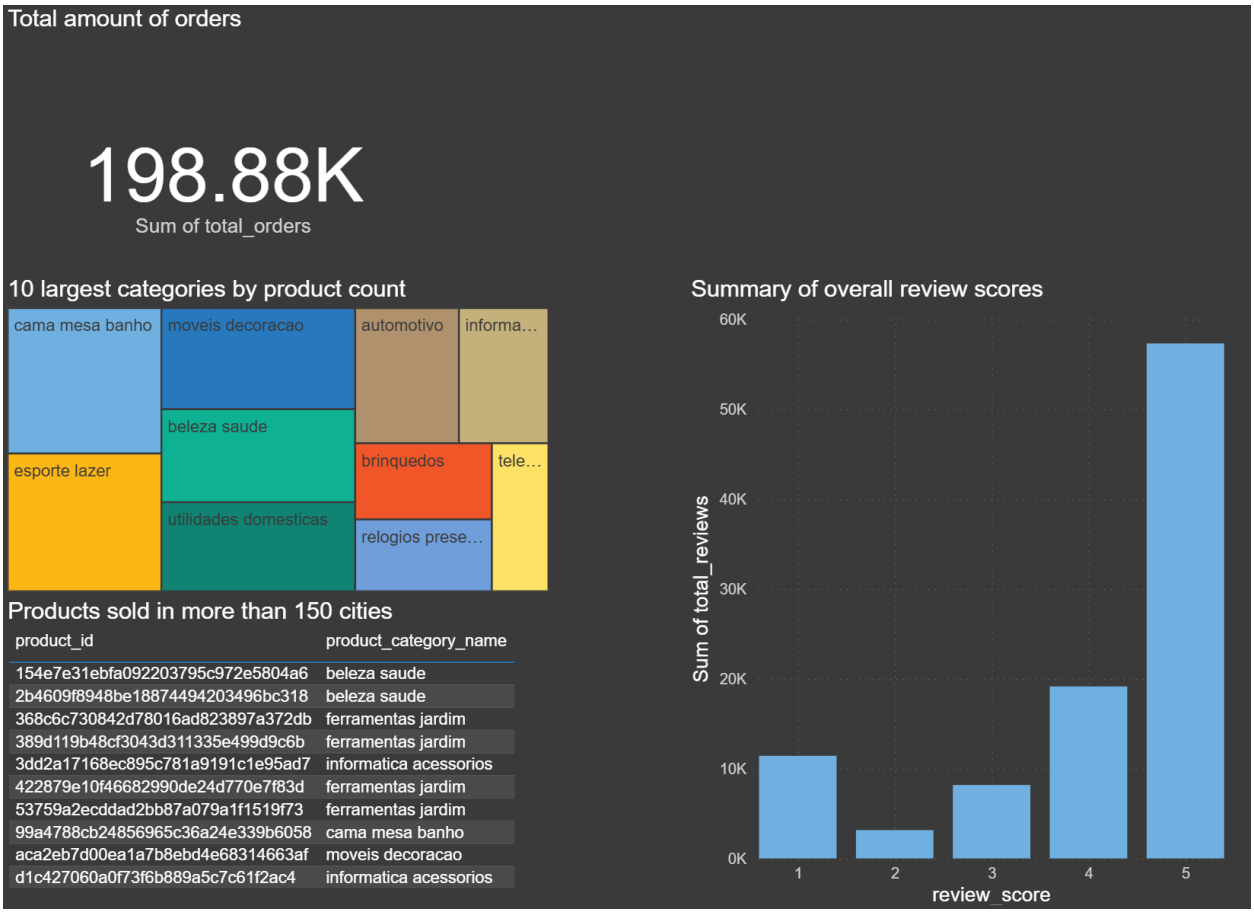
This graph shows monthly profits, and is affected by the users selection of year in the bar plot. For example, if the user wants to see the data for 2017 only:



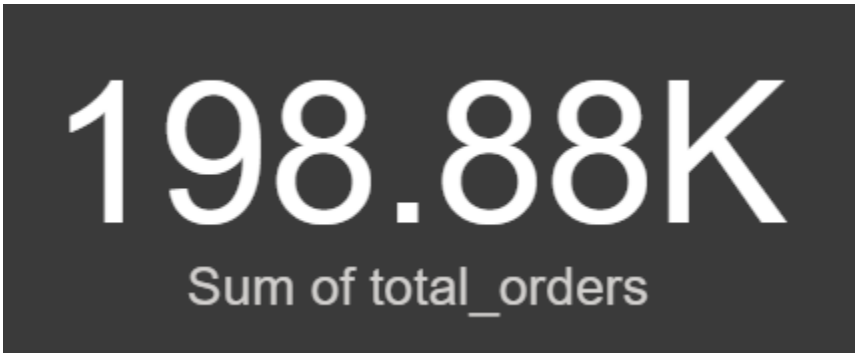
You can see that this changes the graph to zoom in on only values in 2017.

Products:

The product page is not too interesting as of now, but it contains some useful information from previous queries.

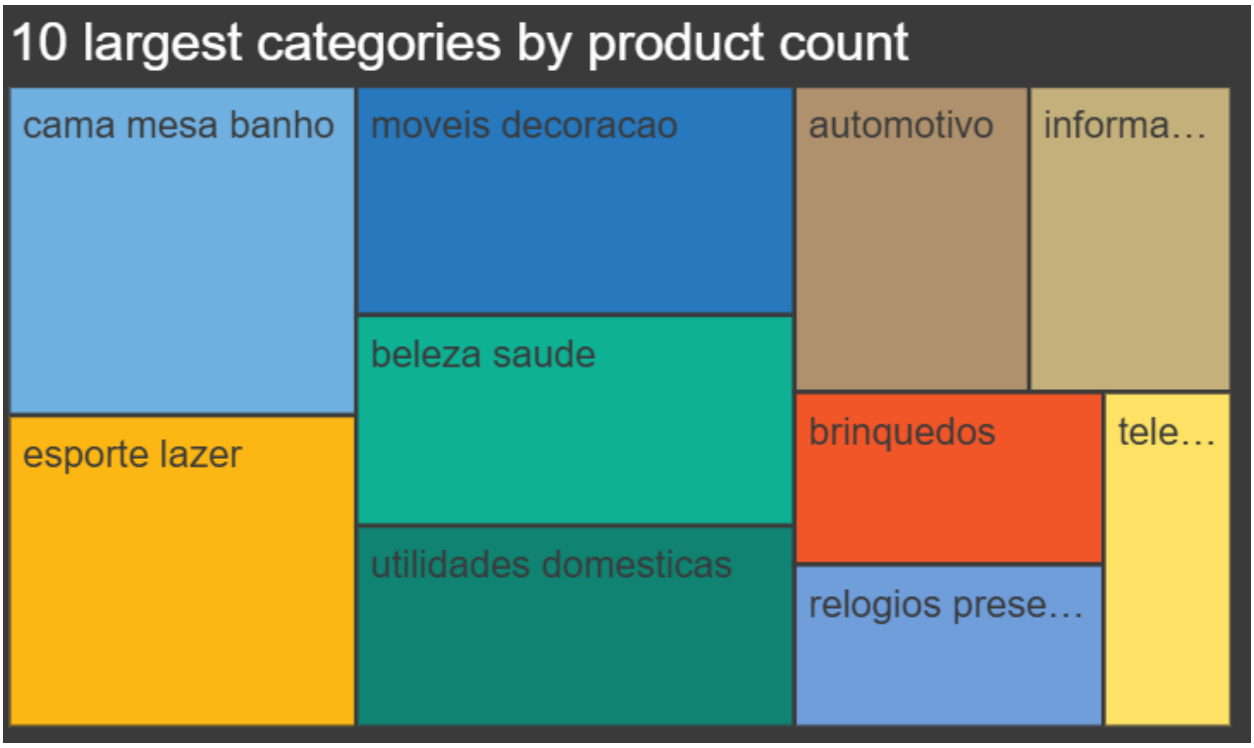


Total amount of orders



This component is just a card to flex on the dashboard. Nothing more than a count of all orders.

Ten largest categories by product count



This treemap displays the ten largest categories by their amount of products. As per now the functionality of interacting with it does not do anything. It could be a feature to select one category from this treemap to see that categories review scores for example.

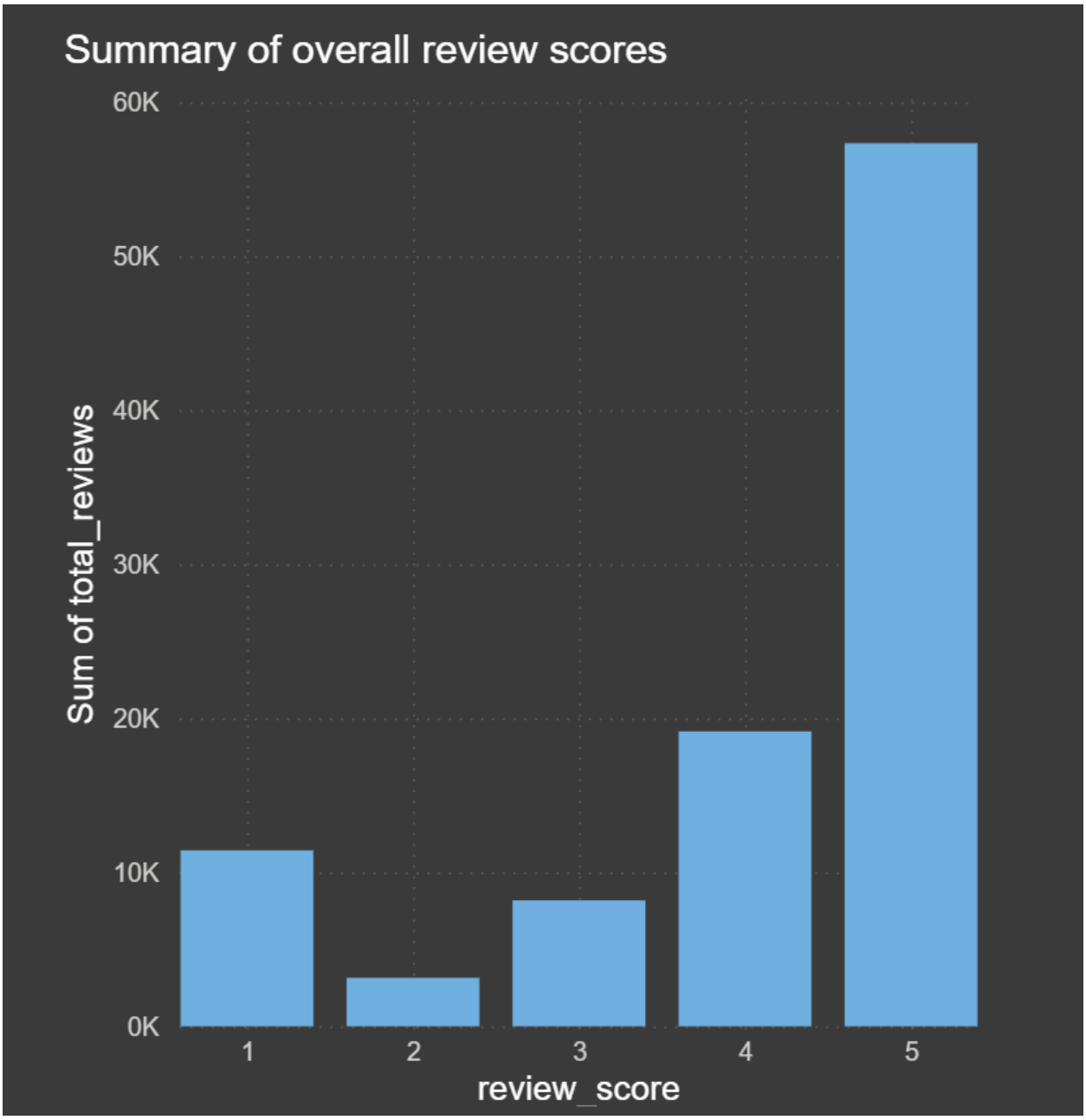
Products sold in more than 150 cities

Products sold in more than 150 cities

product_id	product_category_name
154e7e31ebfa092203795c972e5804a6	beleza saude
2b4609f8948be18874494203496bc318	beleza saude
368c6c730842d78016ad823897a372db	ferramentas jardim
389d119b48cf3043d311335e499d9c6b	ferramentas jardim
3dd2a17168ec895c781a9191c1e95ad7	informatica acessorios
422879e10f46682990de24d770e7f83d	ferramentas jardim
53759a2ecddad2bb87a079a1f1519f73	ferramentas jardim
99a4788cb24856965c36a24e339b6058	cama mesa banho
aca2eb7d00ea1a7b8ebd4e68314663af	moveis decoracao
d1c427060a0f73f6b889a5c7c61f2ac4	informatica acessorios

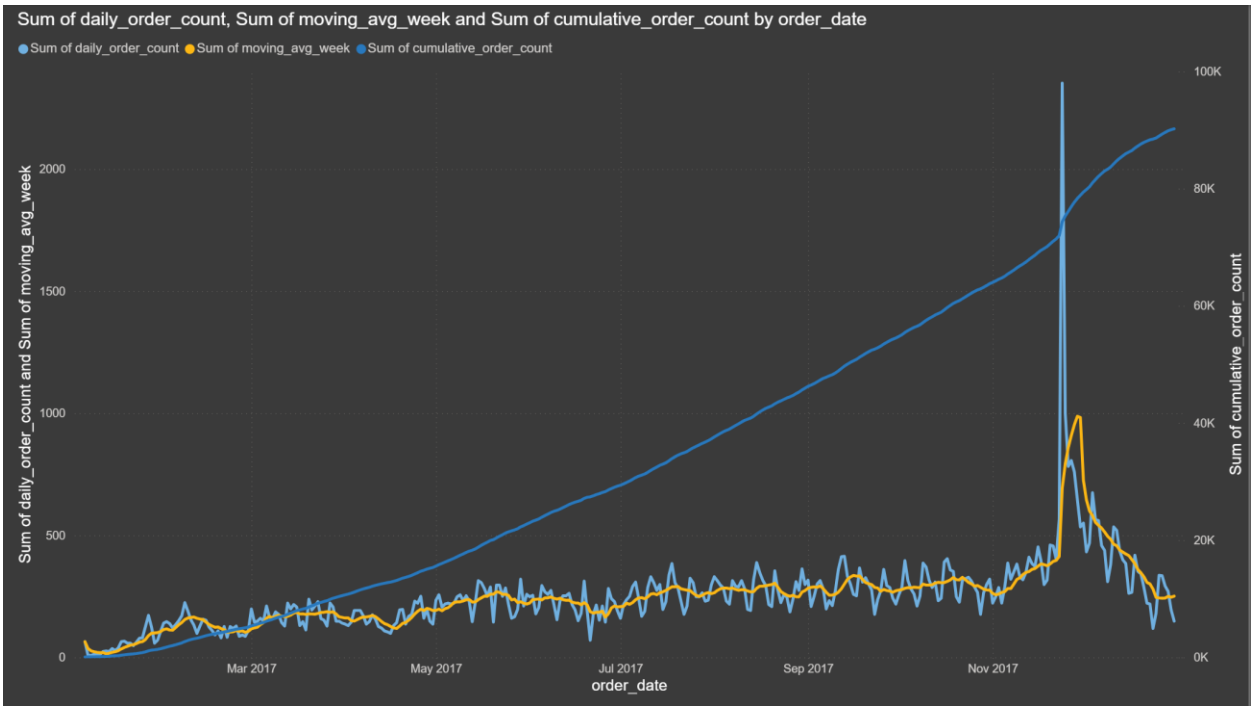
A table of products sold in more than 150 cities. In other words, a list of the most popular products.

Overall review scores



Almost identical to the bar plot of review scores displayed earlier in this report.

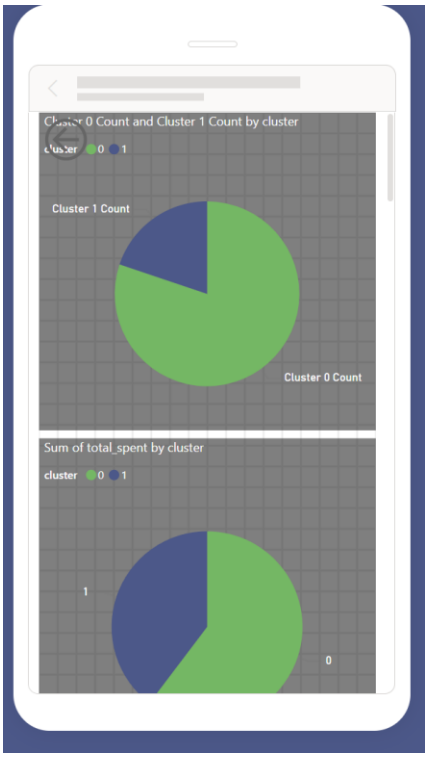
Daily order count, moving seven-day average, and a cumulative order count

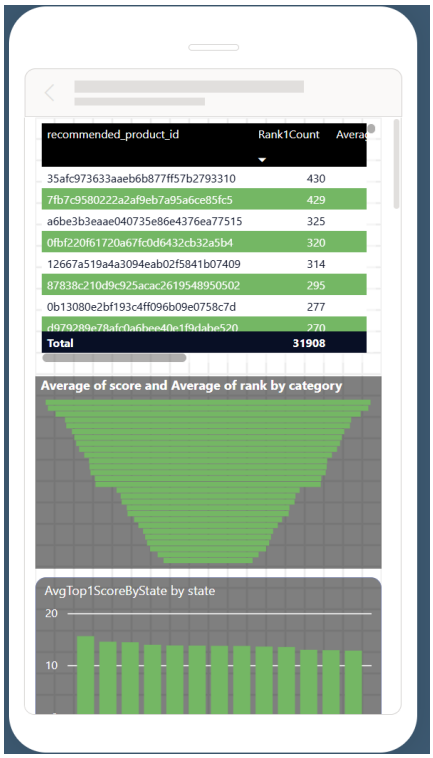


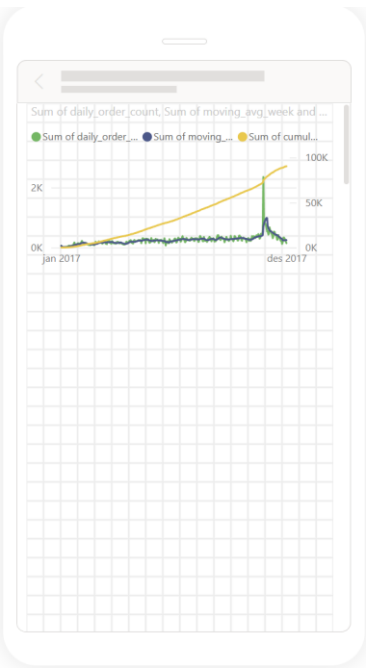
Again, almost identical to the graph seen in task 2. Displays the daily order count as light blue, the seven day moving average as orange, and the cumulative count as a darker blue.

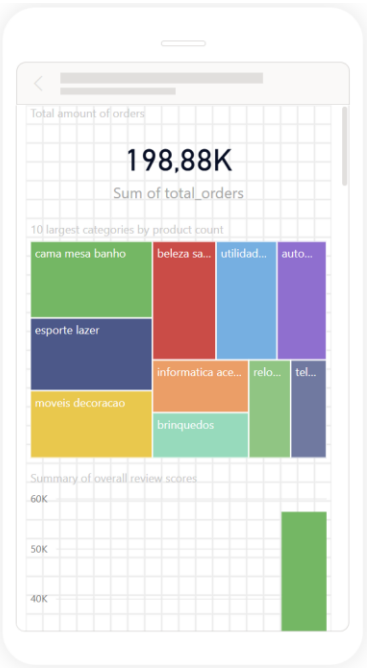
Mobile Layouts

We have mobile layouts for each page; here is a small snippet from each page.









Use of AI

We have been using AI as an assistant, mostly python analytics, especially with co-pilot. We also used AI for troubleshooting when we got errors in Pentaho and python. The tables in the database was made with the built-in-tool for Pentaho to make SQL for each table, and then we connected them with our surrogate keys. The SQL-queries was also with help of AI, mostly to find things like EXTRACT (YEAR, MONTH etc.) from timestamp.