Test

1 Introduction

The goal of this analysis is to evaluate the quality and consistency of sales and metadata tables before uploading them into the system. For this, I have to find at least 8 errors in the data analysis part.

2 Data analysis assignment

2.1 Data

We have 4 files: *zones.csv* (2[1]), *sites.csv* (5), *articles.csv* (15) and *sales.csv* (17997). These files contain all the information for the assignment. In particular, sample sales are based on real store data and are a cutout of the annual aggregation of 15 products that were sold at 5 stores within two different groups.

2.2 Methodology

The analysis focused on four main areas:

- Referential integrity
- Missing values
- Internal consistency
- Anomalies

2.3 The errors

The following 9 errors / inconsistencies were identified:

1. Referential integrity: missing **SiteId** in sites
2. Missing values: 27 missing **CostPrice** references and 13 missing **Price** references
3. Internal consistency: 358 negative margins
4. Internal consistency: 492 duplicate sales
5. Internal consistency: negative quantities (4.6%)
6. Internal consistency: different prices in the same zone (788 cases)
7. Internal consistency: Sales assigned to different zones
8. Anomalies: **Quantity** stored as float, **Date** stored as object
9. Anomalies: extreme outlier quantities (e.g., 432 items sold in one day)

[1] Number of entries in the file.

2.3.1 Referential integrity

1. Extra reference **SiteId** in *sales* not matching with *sites*:

   We started analyzing the data and checking first that all attributes in *sales* are in accordance to the rest. Then it was found an extra **SiteId** reference in *sales*: **SiteId = 9** (4,5,6,7,8,and 9) or missing in *sites* (4,5,6,7 and 8).

   Since *sites* contains stores with the same price over one product and there is information only about 5 stores, the inclusion of 9 would give rise to a new store, which was not counted. Also we found

   ```
   sum(sales["SiteId"]== 9)
   253
   ```

   253 out of 17997, roughly a 1.4%. We also noticed when those entries are from:

   ```
   missing_sites = sales[~sales["SiteId"].isin(sites["SiteId"])]
   missing_sites["Date"].min(), missing_sites["Date"].max()
   ('2018-11-01', '2018-11-30')

   sales["Date"].min(), sales["Date"].max()
   ('2018-07-01', '2019-07-31')
   ```

   This reference only occurs for sales from '2018-11-01' to '2018-11-30', i.e., only in November 2018.

   Suggestion: Confirm with the client whether **SiteId = 9** is a new store with missing metadata or a data entry typo. Update the sites table accordingly. Although, since there are only 5 stores, it is probably a typo.

   As extra information, no new **ArticleId** was found in *sales*.

2.3.2 Missing values

In *sales* it was found:

1. 27 missing **CostPrice** references (NaN), between '2019-03-01' and '2019-03-15'. This happends only for **SiteId = 5** and **ZoneId = 1** (Hypermarket in Belgium).

2. 13 missing **Price** references (NaN) between '2019-04-01' and '2019-04-15'. This happends only for **SiteId = 4** and **ZoneId = 1** (Hypermarket in London).

2.3.3 Internal consistency

These are internal logic errors — the numbers contradict business rules.
In *sales* it was found:

1. 358 sales (≈ 2%) with negative margin during '2019-05-13'-'2019-05-20' for various **SiteId** and **ZoneId** (See Fig.1). Selling under cost price? This suggests potential data errors or exceptional pricing events.

2. 492 duplicate sales, according to articles, site, zone and date:

   ```
   dupes = sales.duplicated(
           subset=["ArticleId", "SiteId", "ZoneId", "Date"],
           keep=False
           )
   len(sales[dupes])
   492
   ```

3. Different prices in same zone.

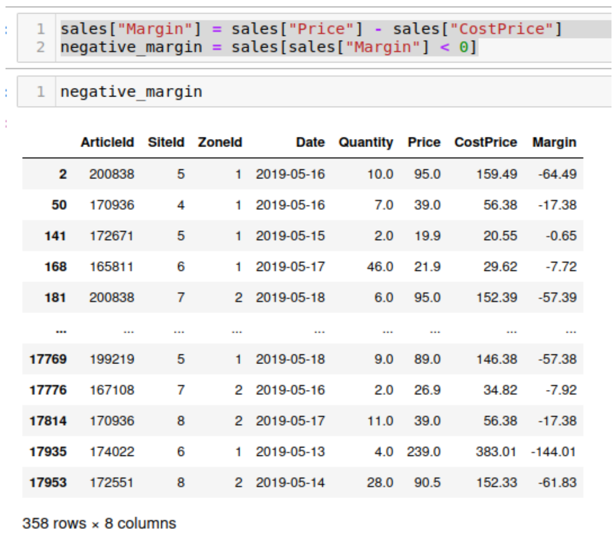
   ```
   zone_price_check = (
   sales
   .groupby(["ArticleId", "Date", "ZoneId"])["Price"]
   .nunique()
   ```

```
)

zone_price_check[zone_price_check > 1]

ArticleId  Date        ZoneId_x
165811     2018-09-20  1           2
2018-10-18  2           2
2018-11-01  2           2
2018-11-02  2           2
2018-11-03  2           2
..
200838     2019-03-22  2           2
2019-03-26  2           2

len(zone_price_check[zone_price_check > 1])
788
```



```
sales["Margin"] = sales["Price"] - sales["CostPrice"]
negative_margin = sales[sales["Margin"] < 0]
```

```
negative_margin
```

	ArticleId	SiteId	ZoneId	Date	Quantity	Price	CostPrice	Margin
2	200838	5	1	2019-05-16	10.0	95.0	159.49	-64.49
50	170936	4	1	2019-05-16	7.0	39.0	56.38	-17.38
141	172671	5	1	2019-05-15	2.0	19.9	20.55	-0.65
168	165811	6	1	2019-05-17	46.0	21.9	29.62	-7.72
181	200838	7	2	2019-05-18	6.0	95.0	152.39	-57.39
...	...	...	...	...	...	...	...	...
17769	199219	5	1	2019-05-18	9.0	89.0	146.38	-57.38
17776	167108	7	2	2019-05-16	2.0	26.9	34.82	-7.92
17814	170936	8	2	2019-05-17	11.0	39.0	56.38	-17.38
17935	174022	6	1	2019-05-13	4.0	239.0	383.01	-144.01
17953	172551	8	2	2019-05-14	28.0	90.5	152.33	-61.83

358 rows × 8 columns

Figure 1: Margin of *sales*

4. Negative **Quantity** references. After checking the description of *sale*, as well as its histogram, it is easy to notice that there are negative values, In particular a total of

```
(sales["Quantity"] <= 0).sum()
820
```

This value represents roughly 4.6% of sales and happens for all articles in different **ZoneId** and **SiteId** only from '2019-02-01' to '2019-02-18' (See Fig. 2).

These might represent product returns or a data import issue. For demand modeling, we exclude these negative quantities to avoid distortion in forecasting.

No internal inconsistencies were found for **CostPrice** or **Price** within the same store and zone on the same date.

```
price_check = (
sales
.groupby(["ArticleId", "SiteId", "ZoneId", "Date"])["CostPrice"]
.nunique()
)


```

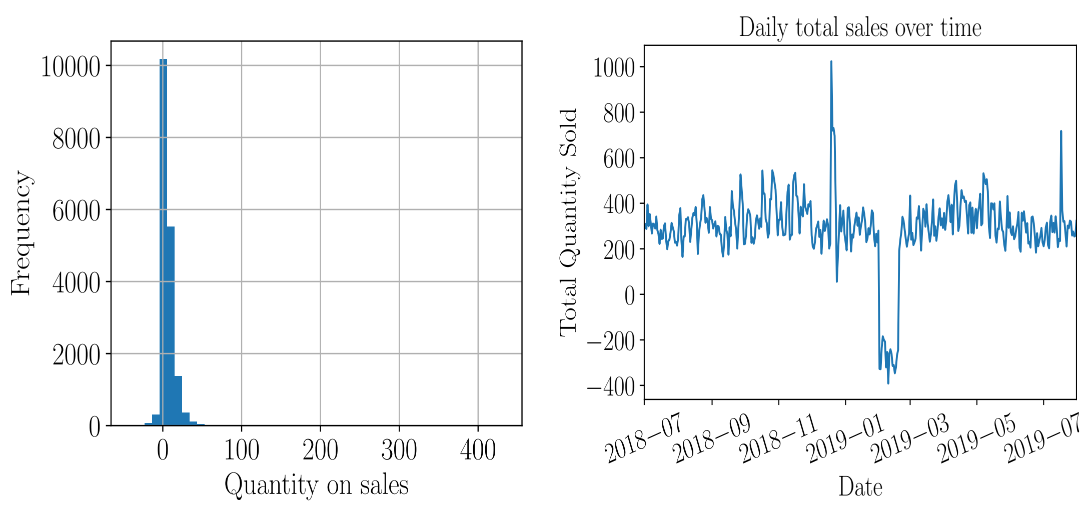


Figure 2: Distribution of **Quantity** on the left and Daily total sales over time on the right.

```python
inconsistent_cost = price_check[price_check > 1]


len(inconsistent_cost)
0

# --- time-series total Quantity sold:
sales["Date"] = pd.to_datetime(sales["Date"], errors="coerce")

daily_sales = (
sales
.sort_values("Date")
.groupby("Date")["Quantity"]
.sum()
.reset_index()
)
```

5. Sales assigned to different zones. By checking the **ZoneId** in *sales* and *sites*,

```python
merged = sales.merge(sites, on="SiteId", how="left")
zone_mismatch = merged["ZoneId_x"] != merged["ZoneId_y"]
zone_mismatch.sum()
253
```

   For the same **SiteId**, *sales* and *sites* disagree on ZoneId for 253 sales. That is, some stores are assigned to different zones in *sales* and *sites*, leading to inconsistent pricing groups.

Only as an observation. I though some prices were too high or inflated by 10. Although, it makes sense to think that prices are given in CZK. However, the currency was not mentioned.

2.3.4 Anomalies

These are statistical or formatting anomalies, which could break models.

1. In *sales* file **Quantity** stored as float instead of integer and **Dates** stored as string (object) rather than datetime. While all values are valid, these formatting issues could affect downstream analysis and model processing.

2. There might be some outliers in **Quantities**. In particular, JACOBS PORCE CAF.LATTÉ 250G was sold 432 times in a Hypermarket in London on the 2019-07-17, and also 273 times in Belgium

on the 2018-12-19. This is at least suspicious for me. I would require business validation. These quantities cause the two peaks observed in Fig. 2.

2.4 Analysis workflow and discussion

The analysis was conducted as an iterative process, where initial findings guided subsequent validation steps and informed further investigation.

First, I explored the structure of all tables to understand their relationships and identify key identifiers. This allowed me to verify referential integrity before analyzing numerical values.

Next, I performed basic validation checks, including missing values, data types, and date parsing, since such issues can invalidate further analysis if not addressed early.

After ensuring basic consistency, I analyzed statistical properties and distributions of key variables (minimum, maximum, quartiles, mean, and standard deviation). This step helped identify outliers and illogical values, such as negative quantities and extreme sales volumes. Time-series visualization was used to detect clustered anomalies, revealing periods with unusually high (possible) returns and extreme daily spikes.

Based on these findings, I focused on business-related consistency checks described in the assignment, including duplicate sales records, pricing consistency within zones, and margin validation.

Throughout the process, each detected anomaly was evaluated in terms of its possible business meaning (e.g., returns, historical metadata gaps, or input errors) and its potential impact on downstream models.

Each detected issue was then evaluated according to its potential business meaning and modeling impact, and classified as critical, correctable (mismatching, typos), or informational (outliers, formatting). Critical issues (e.g., inconsistent references, duplicate transactions, negative margins) would require resolution before model deployment.

Based on this analysis, the dataset contains incomplete or inconsistent metadata that should be verified with the client or data provider. Additionally, negative quantities likely represent returns and should be treated separately to improve demand modeling.

Several detected issues could significantly distort demand forecasts and pricing optimization if left unresolved. I would certainly recommend implementing automated validation rules and periodic data quality monitoring as part of the ingestion pipeline.

Further business knowledge would allow for more precise interpretation and stronger domain-specific validation rules.

3 Scripting assignment

Code below is a Python implementation of one famous algorithm. Could you briefly walk us through its steps and tell us what the result would be?

```python
1 def bb(arr):
2 n = len(arr)
3 for i in range(n-1):
4 for j in range(0, n-i-1):
5 if arr[j] > arr[j+1] :
6         arr[j], arr[j+1] = arr[j+1], arr[j]
```

Sure.

line 1: define function bb, which receive an array as an argument.

line 2: defines n as the length of the array.

line 3: creates a loop running with the size of the array for i between 0 and n-1, with a step 1.

line 4: creates another loop to check all elements from the first in the array until the (last one -i)

line 5: Condition. Checks the following statement "j-th element of the array is greater than the next element in it"

line 6: If the previous statement is True, swap if left element is greater than right.

Overall, this function, arranges the value of the array from the lowest to the largest, by passing checking the condition of larger than the next in line 6, passing though all elements in the array, provided line 3, the checking for the rest until all numbers are in order, line 5. Notices that line 3 guarantees that if the largest number is at the beginning, it will end as the last element in the array, so for every loop i, it puts the largest to the front, the the second largest behind the largest and so on.