POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

# Systems and Methods for Big and Unstructured Data Project

Author: **Matteo Vitali (10800443)**

Group Number: **114**

Academic Year: 2024-2025

# Contents

# 1 | Introduction

In this chapter, the reader is presented with a high-level description of the project work delivered for the *Systems and Methods for Big and Unstructured Data course* at Politecnico di Milano.

## 1.1. Objective

The objective of this project is to design and perform 10 queries on a publicly available dataset employing a No-SQL technology (Neo4j, MongoDB or Elasticsearch). In this way, valuable insights could be extracted from the collected data and, at the same time, knowledge acquired during the course on one of these Database Management Systems (DBMSs) could be applied in practice.

## 1.2. Dataset overview

The following analysis is based on Yelp dataset that can been found on Kaggle or on the official Yelp's page.

It's a comprehensive collection of information about businesses, reviews, users, and other related data, designed for academic research and learning purposes, organized as follows:

- Reviews: $6,990,280$ in total, complete with full text, star ratings, and metadata such as helpfulness votes and dates.
- Businesses: $150,346$ in total, each described with attributes like location, hours, categories, and reviews.
- Users: $1,987,897$ in total, with review activity, social connections, and received compliments.
- Check-ins: over $200,100$ check-ins for $131,930$ different businesses.

The original dataset contained, also, tips (brief reviews written by users) and pictures, but they haven't been included in the analysis for the sake of simplicity.

Data span multiple metropolitan areas, including cities like Montreal, Toronto, Phoenix, Las Vegas, and Pittsburgh.

Here, the reader can find an example of each document:

- Sample object of `businesses.json`:


```
{
    "business_id": "tnhfDv5Il8EaGSXZGiuQGg",
    "name": "Garaje",
    "address": "475 3rd St",
    "city": "San Francisco",
    "state": "CA",
    "postal_code": "94107",
    "latitude": 37.7817529521,
    "longitude": -122.39612197,
    "stars": 4.5,
    "review_count": 1198,
    "is_open": 1,
    "attributes": {
        "RestaurantsTakeOut": true,
        "BusinessParking": {
            "garage": false,
            "street": true,
            "validated": false,
            "lot": false,
            "valet": false
        }
    },
    "categories": ["Mexican", "Burgers", "Gastropubs"],
    "hours": {
        "Monday": "10:00-21:00",
        "Tuesday": "10:00-21:00",
        "Friday": "10:00-21:00",
        "Wednesday": "10:00-21:00",
        "Thursday": "10:00-21:00",
        "Sunday": "11:00-18:00",
        "Saturday": "10:00-21:00"
```

```
    }
}
```

- Sample object of `users.json`:

```
{
    "user_id": "Ha3iJu77CxlrFm-vQRs_8g",
    "name": "Sebastien",
    "review_count": 56,
    "yelping_since": "2011-01-01",
    "friends": [
        "wqoXYLWmpkEH0YvTmHBsJQ",
        "KUXLLiJGrjtSsapmxmpvTA",
        "6e9rJKQC3n0RSKyHLViL-Q"
    ],
    "useful": 21,
    "funny": 88,
    "cool": 15,
    "fans": 1032,
    "elite": [2012, 2013],
    "average_stars": 4.31,
    "compliments": {
        "hot": 339,
        "funny": 99,
        "cool": 91
    }
}
```

- Sample object of `reviews.json`:

```
{
    "user_id": "Ha3iJu77CxlrFm-vQRs_8g",
    "name": "Sebastien",
    "review_count": 56,
    "yelping_since": "2011-01-01",
    "friends": [
```

```
        "wqoXYLWmpkEH0YvTmHBsJQ",
        "KUXLLiJGrjtSsapmxmpvTA",
        "6e9rJKQC3n0RSKyHLViL-Q"
    ],
    "useful": 21,
    "funny": 88,
    "cool": 15,
    "fans": 1032,
    "elite": [2012, 2013],
    "average_stars": 4.31,
    "compliments": {
        "hot": 339,
        "funny": 99,
        "cool": 91
    }
}
```


- Sample object of `checkins.json`:


```
{
    "business_id": "tnhfDv5Il8EaGSXZGiuQGg",
    "date": "2016-04-26 19:49:16, 2016-08-30 18:36:57, 2016-10-15 02:45:18, 2016-11-18 01:54:50, 2017-04-20 18:39:06, 2017-05-03 17:58:02"
}
```


## 1.3. Methodology

In this project, an organizational approach is adopted to analyse the dataset: the ultimate goal, namely, consists of extracting actionable insights to support decision-making within the businesses and having a deeper understanding of customer behaviour.

The project began with a comprehensive examination of the dataset's structure and features. This initial phase was essential for uncovering significant elements worthy of further investigation. After this, first batch of queries (1 to 7 and 9) were formalized and drafted

in code. As last step, sentiment analysis was performed on reviews, DB updated and remaining queries (leveraging previous results) written.

## 1.4. DBMS Technology

For this project, MongoDB was used.

### 1.4.1. MongoDB overview

MongoDB is a document-oriented, open source database system that organizes data using a JSON-like document structure. It offers speed, scalability, and performance when handling large datasets by leveraging its distributed architecture. Its compatibility with JSON / CSV makes it an excellent choice for managing web data or scientific dataset.

MongoDB supports flexible document structure that is necessary when dealing with changing data requirements. It also offers powerful aggregation pipelines and geospatial data processing. As a document-based database, MongoDB performs data retrieval without the need for resource-intensive join operations commonly used in relational databases.

### 1.4.2. Technology selection criteria

As the reader will see, I heavily relied on MongoDB's flexible schema design for data analysis. Namely, diverse data structures can be stored in each document, without enforcing rigid constraints: think, for example, to business attributes, user profiles that are have naturally different formats and values. Moreover, when new features are added or queries' objectives change, I can immediately start storing and analysing new fields without schema migrations.

Instead of splitting related data across multiple collections, MongoDB can store a business entity along with its reviews, check-ins, and other related data in a single document, reducing the need for `$lookup`s and making the queries more straightforward to write.

Under the hood, MongoDB's WiredTiger storage engine caches frequently accessed data in memory while the rest on disk. Especially for large datasets, like the one in analysis, this is a key point for guaranteeing faster query execution and better resource utilization.

Neo4j has been considered as an alternative to MongoDB. Namely, the most important relations that emerge from the model are: `(:User) -> [:REVIEWS] -> (:Business)`, `(:User) -> [:ENTERS] -> (:Business)`, `(:User) -> [:HAS_FRIEND] -> (:User)`. For queries to be performed, Neo4j wouldn't be the optimal tool for several reasons. First,

Neo4j's strength lies in handling complex relationships and path queries, which aren't the primary focus of business and review analysis. Then, Neo4j doesn't provide the same level of flexibility in document structure that MongoDB offers, making it more difficult to handle varying business attributes and review formats. Moreover, Neo4j is not optimized for running aggregation queries over the entire graph, so performing historical analysis over such a model isn't suitable.

Elasticsearch is excellent for full-text search and real-time log analysis, but it presents several limitations when it comes to analyse businesses and reviews data. First, its primary storage architecture is optimized for search operations rather than analytical queries: it excels at finding and retrieving documents based on text content, but it's way more inefficient in aggregations or joins that are common in my analysis. Vice versa, MongoDB's aggregation pipeline provides a more natural and efficient approach. The requirement to define mappings for all fields in Elasticsearch is fundamental for search optimization, but adds an extra layer of complexity that isn't necessary for pure data analysis tasks. Finally, it's true that Elasticsearch can handle structured queries, but they often require more verbose and less intuitive structures compared to MongoDB.

# 2 | Data wrangling

With *"data wrangling"* we refer to the process of cleaning and reshaping raw input data, to enhance its quality and usability for analytical purposes.

## 2.1. Data collection

As mentioned before, data was downloaded from Yelp's dataset official page. It was organized in four different JSON files: `buinsesses.json`, `users.json`, `reviews.json` and `checkins.json`. All of them were converted in Pandas DataFrames using `pd.read_json()` method with `lines = True` to guarantee line by line reading and `convert_dates = [...]` for dates attributes.

## 2.2. Data inspection

After having loaded the data, it was inspected to understand better the type of each attribute, the number of records and the size of each collection.

## 2.3. Data cleaning

The primary objective of data cleaning phase was to search for `null` values and deal with them. Luckily, before publishing the dataset, Yelp team analysed it and started to clean it. There were no `null` values in `reviews`, `users` and `checkins` DataFrames. Vice versa, `attributes`, `categories` and `hours` contained some `null` values distributed as follows:

```
business_id         0
name                0
address             0
city                0
state               0
postal_code         0
latitude            0
longitude           0
stars               0
review_count        0
is_open             0
attributes      13744
categories        103
hours           23223
dtype: int64
```

Figure 2.1: Distribution of `null` values in `businesses`

This is not surprising at all because some businesses might not have specified opening hours or attributes. For 103 businesses no category is specified. It seemed like that having one of them `null` was uncorrelated with the others being `null`, so I let MongoDB handle the missing fields.

## 2.4. Data normalization

In this part of the preprocessing, I dived deeper into the DataFrames to prepare them for the following shaping phase.

First, uninteresting columns were dropped and complex JSON structures (like the one for `attributes`) flattened using `flatten()` method of `flatten_json` library. In `elite` column there were `null` values and not coherent or repeated years (like 20 for 2020).

It emerged that 44.19% of total users didn't have any friends: this seemed uncorrelated with average star ratings or number of compliments received, so no rows were dropped. Also, upon having verified that `users["friends"]` $\not\subset$ `users["userd_id"]`, I have filtered out them with the method `filter_list()`.

The same problem arose in `businesses` and `reviews` collection: there were users who have written some reviews but their `user_id` was not in `users["user_id"]`: I dropped

them from `reviews` DataFrame.

Luckily, `reviews["buiness_id"]` $\subset$ `businesses["business_id"]`.

For each business, check-ins were stored in the form of an array of dates. Pandas is not able to directly convert dates in it during reading phase, so the following was employed:

```
checkins["date"] = checkins["date"].str.split(", ")
checkins["date"] = checkins["date"].apply(lambda x : pd.to_datetime(x,
format = '%Y-%m-%d %H:%M:%S'))
```

Then, to make the dates compatible with MongoDB:

```
checkins["date"] = checkins["date"].apply(lambda x:
date.to_pydatetime() for date in x])
```

## 2.5. Data shaping

If I loaded users, businesses, reviews and check-ins in DBMS and then query the DB with `$lookup` (the MongoDB counterpart of SQL `join`), I wouldn't have been leveraging the main feature of MongoDB that is embedding documents inside others.

The idea, here, was to embed `reviews` and `checkins` inside `businesses`. To do this, first reviews and check-ins were grouped on `business_id` obtaining relations of the form:

$$\texttt{business\_id} \longrightarrow \texttt{[reviews for that business\_id]}$$

$$\texttt{business\_id} \longrightarrow \texttt{[checkins for that business\_id]}$$

Then, a Pandas `merge` was performed:

```
businesses.merge(reviews_grouped.rename("reviews"),
                 on = 'business_id', how = 'left')\
          .merge(checkins_grouped.rename("checkins"),
                 on = 'business_id', how = "left")
```

Finally, redundant `business_id` column used in `merge` was removed. The resulting DataFrame (`buinesses_merged`) was the following:

| | business_id | name | address | city | state | postal_code | latitude | longitude | stars | review_count | attributes | categories | hours | reviews | checkins |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | Pns2l4eNsfO8kk83dixA6A | Abby Rappoport, LAC, CMQ | 1616 Chapala St. Ste 2 | Santa Barbara | CA | 93101 | 34.426679 | -119.711197 | 5.0 | 7 | {'ByAppointmentOnly': 'True'} | [Doctors, Traditional Chinese Medicine, Naturo... | None | [{'review_id': '9vwYDBVI3ymdqcyJSWW2Tg', 'user... | [2018-09-21 20:51:31] |
| 1 | mpf3x-BjTdTEA3yCZrAYPw | The UPS Store | 87 Grasso Plaza Shopping Center | Affton | MO | 63123 | 38.551126 | -90.335695 | 3.0 | 15 | {'BusinessAcceptsCreditCards': 'True'} | [Shipping Centers, Local Services, Notaries, M... | {'Monday': '0:0-0:0', 'Tuesday': '8:0-18:30', ... | [{'review_id': '-WXMS4p3D9NQsAPw4YPEyw', 'user... | [2011-12-12 23:30:26, 2014-05-23 20:31:34, 201... |
| 2 | tUFrWirKiKi_TAnsVWINQQ | Target | 5255 E Broadway Blvd | Tucson | AZ | 85711 | 32.223236 | -110.880452 | 3.5 | 22 | {'BikeParking': 'True', 'BusinessAcceptsCredit... | [Department Stores, Shopping, Fashion, Home & ... | {'Monday': '8:0-22:0', 'Tuesday': '8:0-22:0', ... | [{'review_id': 'lOmiYoBPtQsY_fh5uA4mXg', 'user... | [2010-07-25 04:30:06, 2010-08-31 04:07:22, 201... |
| 3 | MTSW4McQd7CbVtyjqoe9mw | St Honore Pastries | 935 Race St | Philadelphia | PA | 19107 | 39.955505 | -75.155564 | 4.0 | 80 | {'RestaurantsDelivery': 'False', 'OutdoorSeati... | [Restaurants, Food, Bubble Tea, Coffee & Tea, ... | {'Monday': '7:0-20:0', 'Tuesday': '7:0-20:0', ... | [{'review_id': 'BXQcBN0iAi1IAUxibGLFzA', 'user... | [2010-08-18 17:05:36, 2010-11-25 17:45:31, 201... |

Figure 2.2: DataFrame `businesses_merged`

Same procedure was applied for `users` and `reviews`. I started by grouping reviews, this time, on `user_id`, then I merged each group with the `users` DataFrame and finally I dropped redundant joining attribute `user_id` from the resulting table. The resulting DataFrame `users_merged` is the following:

| | user_id | name | review_count | yelping_since | useful | funny | cool | elite | friends | fans | ... | compliment_profile | compliment_cute | compliment_list | compliment_note | compliment_plain | compliment_cool | compliment_funny | compliment_writer | compliment_photos | reviews |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | qVc8ODYU5SZjKXVBgXdI7w | Walker | 585 | 2007-01-25 16:47:26 | 7217 | 1259 | 5994 | [2007] | [NSCy54eWehBJyZdG2iE84w, 6EDA76ffkaTyOeuxkg0Fm... | 267 | ... | 55 | 56 | 18 | 232 | 844 | 467 | 467 | 239 | 180 | [{'review_id': 'Egy2a4qZeXGr2aY6KMixbg', 'busi... |
| 1 | j14WgRoU_-2ZE1aw1dXrJg | Daniel | 4333 | 2009-01-25 04:35:42 | 43091 | 13066 | 27281 | [2009, 2010, 2011, 2012, 2013, 2014, 2015, 201... | [ueRPE0CX75ePGMqOFVj6IQ, E_GAXhVA1_lVC2aFpMQEl... | 3138 | ... | 184 | 157 | 251 | 1847 | 7054 | 3131 | 3131 | 1521 | 1946 | [{'review_id': 'iMpqDa0Oyukiw5406KYNRw', 'busi... |
| 2 | 2WnXYQFK0hXEoTxPtV2zvg | Steph | 665 | 2008-07-25 10:41:00 | 2086 | 1010 | 1003 | [2009, 2010, 2011, 2012, 2013] | [pypZb3V5TXHOnlTj-qLSrw, irkRHMqg9oSt7lv3OSiNk... | 52 | ... | 10 | 17 | 3 | 66 | 96 | 119 | 119 | 35 | 18 | [{'review_id': 'vILRJrV_Qxhroxgm4_cstw', 'busi... |
| 3 | SZDeASXq7o05mMNLshsdIA | Gwen | 224 | 2005-11-29 04:38:33 | 512 | 330 | 299 | [2009, 2010, 2011] | [enx1vVPnfdNUdPho6PH_wg, 1OocYCAZixwbAXueW75FM... | 28 | ... | 1 | 6 | 2 | 12 | 16 | 26 | 26 | 10 | 9 | [{'review_id': 'nCIgkXNs7JxJmSQAftGKyw', 'busi... |
| 4 | hASIMy-EnncsH4JoR-hFGQ | Karen | 79 | 2007-01-05 19:40:59 | 29 | 15 | 7 | NaN | [tjm3b3Ts8msuGiYRybHx0Q, Z1MFQFgn-O9ADIOr6g-wE... | 1 | ... | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | [{'review_id': 'ux2ssSDD8i8mNeMtvQF1KQ', 'busi... |

Figure 2.3: DataFrame `users_merged`

# 3 | Data ingestion and schema design

This chapter focuses on the process of importing data into the database and defining its structure. It explains how the data was loaded into MongoDB and the schema design used to organize and store it effectively. By the end of the chapter, the dataset will be ready for efficient querying and analysis.

## 3.1. Data loading

Loading that amount of documents directly through MongoDB GUI was infeasible, so PyMongo and pre-computed Pandas DataFrames were leveraged.

First, a connection was established with the server with the following statement: `client = pymongo.MongoClient("mongodb://localhost:27017/")`; then, a DB instance was created `db = client["yelp"]`. To speed up the injection process, I employed batch loading: at each iteration a chunck (of size `batch_size = 1000`) of original DataFrame was inserted in the DB using `db[collection_name].insert_many(records)`. A TQDM progress bar was displayed for each collection.

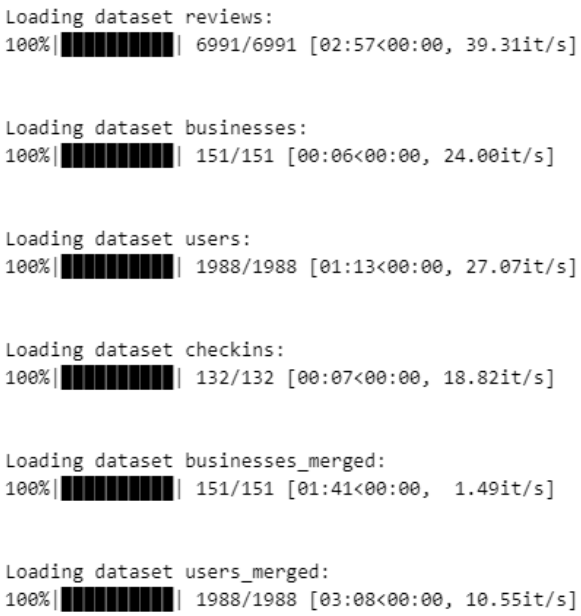


Figure 3.1: Loading collections in DB instance

## 3.2. Schema definition

As the reader will surely have noticed, not only the aggregated collections but also the individual collections resulting from the analysis and cleaning of the JSON files were loaded into the database. This approach was intentional because some queries required working on single entities independently of their relationships with others.

While one might argue that this created redundancy, it was a deliberate trade-off to improve performance. By having individual collections readily available, queries that do not depend on relationships can be executed faster and more efficiently.

Individual collections have the following shape:

- `businesses`

| Field Name | Type | Description |
|---|---|---|
| `business_id` | `string` | Unique identifier for the business (22 characters). |
| `name` | `string` | The name of the business. |
| `address` | `string` | Full address of the business. |
| `city` | `string` | City where the business is located. |

| | | |
|---|---|---|
| state | string | State code (2 characters). |
| postal_code | string | Postal code of the business. |
| latitude | float | Latitude of the business location. |
| longitude | float | Longitude of the business location. |
| stars | float | Star rating, rounded to half-stars. |
| review_count | integer | Number of reviews for the business. |
| is_open | integer | Indicates whether the business is open (1 for open, 0 for closed). |
| attributes | dict | Key-value pairs of business attributes (e.g., parking availability). |
| categories | array[string] | Array of strings representing business categories. |
| hours | dict | Key-value pairs of operating hours by day. |

Table 3.1: businesses collection

- reviews

| Field Name | Type | Description |
|---|---|---|
| review_id | string | Unique identifier for the review (22 characters). |
| user_id | string | Unique identifier for the user who wrote the review. |
| business_id | string | Unique identifier for the business the review is written for. |
| stars | integer | Star rating given in the review. |
| date | DateTime | Date of the review (format: YYYY-MM-DD HH:MM:SS). |
| text | string | The text of the review. |
| useful | integer | Number of useful votes received by the review. |
| funny | integer | Number of funny votes received by the review. |
| cool | integer | Number of cool votes received by the review. |

Table 3.2: reviews collection

- `users`

| Field Name | Type | Description |
|---|---|---|
| `user_id` | `string` | Unique identifier for the user (22 characters). |
| `name` | `string` | User's first name. |
| `review_count` | `integer` | Number of reviews written by the user. |
| `yelping_since` | `DateTime` | Date the user joined Yelp (format: YYYY-MM-DD HH:MM:SS). |
| `friends` | `array[string]` | Array of user IDs representing the user's friends. |
| `useful` | `integer` | Number of useful votes sent by the user. |
| `funny` | `integer` | Number of funny votes sent by the user. |
| `cool` | `integer` | Number of cool votes sent by the user. |
| `fans` | `integer` | Number of fans the user has. |
| `elite` | `array[int]` | Years the user was marked as elite. |
| `average_stars` | `float` | User's average star rating across all reviews. |

Table 3.3: `users` collection

- `checkins`

| Field Name | Type | Description |
|---|---|---|
| `business_id` | `string` | Unique identifier for the business (22 characters). |
| `date` | `array[DateTime]` | Comma-separated list of timestamps (format: YYYY-MM-DD HH:MM:SS). |

Table 3.4: `checkins` collection

The structure of `businesses_merged` and `users_merged` is not shown because the reader can easily infer it from the previously presented tables.

## 3.3. Indexing

MongoDB assigns by default to each object a unique ID. With such a large amount of data, it is fundamental to define custom indices to keep good queries performance. I did

it directly in the notebook using PyMongo `create_index()` method:

```
#For "businesses", we need only "business_id" as index
db["businesses"].create_index(["business_id"], unique = True)

#For "users", we need only "user_id" as index
db["users"].create_index(["user_id"], unique = True)

#For "checkins", we need only "business_id" as index
db["checkins"].create_index(["business_id"], unique = True)

#For "reviews", we need "business_id", "user_id" and "review_id" as index
db["reviews"].create_index(["business_id"])
db["reviews"].create_index(["user_id"])
db["reviews"].create_index(["review_id"], unique = True)

#For "businesses_merged", we need "business_id" as index
db["businesses_merged"].create_index(["business_id"], unique = True)

#For "users_merged", we need "user_id" as index
db["users_merged"].create_index(["user_id"], unique = True)
```

where `unique = True` parameter tells MongoDB that index values are unique (it is not the case of `user_id` in `reviews` because one user might appear in more than one document).

# 4 | Sentiment analysis

In this chapter, the reader is presented with the preprocessing work done for sentiment analysis phase (extra).

## 4.1. Sentiment analysis model

The model used for performing sentiment analysis is `distilbert-base-uncased-finetuned-sst-2-english` that is a lightweight, fine-tuned version of DistilBERT. It has been derived from the `distilbert-base-uncased` architecture by fine-tuning it on the Stanford Sentiment Treebank (SST-2) dataset. It classifies text into positive or negative categories.

I chose it because it is one of the most used models for social media monitoring, customer feedback analysis or product review assessments and it is able to achieve high accuracy despite its compact size. Furthermore, the fine-tuning process on SST-2 enhances the model's capability to identify nuanced emotional expressions, such as irony or strong sentiments.

It's said to be "uncased" because it treats upper / lower case letters in the same manner.

As all the other transformers architectures, the model employs self-attention mechanisms to process input sequences and capture context.

Once having installed *PyTorch* and `transformers` module from *HuggingFace*, it's API is very simple to use:

```
from transformers import pipeline

classifier = pipeline("sentiment-analysis",
                      model = """distilbert-base-uncased-finetuned-sst-
                      2-english""")

text = "This product works amazingly well!"
```

```
result = classifier(text)
print(result)
```

The output is:

```
Device set to use cuda:0
```

```
[{'label': 'POSITIVE', 'score': 0.9998623132705688}]
```

## 4.2. Objective

Sentiment analysis was not the first thing done after the dataset was loaded. The majority of the queries were performed, and, upon having analysed their results, the idea of applying sentiment analysis came up.

The ultimate goal was to bridge the gap between quantitative (stars) and qualitative (text) feedback to provide insights into user reviews. For example, by exploring the relationship between review sentiment and star ratings, one could assess how well the sentiment in text aligns with the corresponding star rating; cases where positive sentiment accompanies a low star rating, or vice versa, might provide insights of biases into user behaviour. Or, by segmenting users based on the typical sentiment of their reviews, one could assist businesses in better interpreting customer feedback.

## 4.3. Methodology

In this section, I present how sentiment analysis pipeline was integrated with MongoDB existing collections.

### 4.3.1. Data extraction

Firstly, data had to be retrieved and put in a DataFrame. A simple MongoDB query was written and executed:

```
client = pymongo.MongoClient("mongodb://localhost:27017/")
db = client["yelp"]

reviews = db["reviews"].find({}, {
    "review_id" : 1,
    "text" : 1
```

```
})

reviews = pd.DataFrame(reviews)
reviews = reviews.set_index("review_id")
reviews
```

Note that `_id` column wasn't kept, but only `review_id` column because as mentioned in previous chapter, I created an index on it.

### 4.3.2. Model usage

The main class is `SentimentAnalyzer`.

First, text was preprocessed: each review text was mapped into numerical tokens that the model is able to understand. Naturally, text varies in length, so padding and truncation were used to ensure consistent input lengths:

```
tokenizer(texts,
          max_length = self.max_length,
          padding = True,
          truncation = True,
          return_tensors = "pt")
```

To optimize memory usage, the model operated in half-precision mode when using GPU:

```
if self.device == "cuda":
    self.model = self.model.half()
```

Dataset consisted of about 7 million reviews and all of them needed to be processed to have good results, so batching and chunking strategies were applied:

- Chunks: large portions of the dataset (e.g. $10,000$ reviews) that were loaded into memory and stored in CSV file at once;
- (Mini-) Batches: smaller groups (e.g. 512 reviews) within each chunk that were processed together by the GPU.

This was implemented as nested loops:

```
for start_idx in range(0, total_rows, chunk_size):
    chunk_df = df.iloc[start_idx : end_idx]

    for batch_start in range(0, len(chunk_df), self.batch_size):
```

```
        batch_df = chunk_df.iloc[batch_start : batch_end]
```

Once having defined the `analyzer`, I used it:

```
analyzer = SentimentAnalyzer("""distilbert-base-uncased-finetuned
                                -sst-2-english""",
                                batch_size = 512, max_length = 256)

analyzer.process_reviews(reviews,
                            "sentiment_results.csv", 10000)
```

where `max_length` is the maximum number of tokens in which text was split up by the tokenizer. The model was trained and fine-tuned with sequences of maximum length of 512, so all values below should be fine. By inspecting the distribution of number of tokens per reviews, 256 seemed a good compromise between performance and expressiveness:

```
count    6.990247e+06
mean     1.327057e+02
std      1.218545e+02
min      3.000000e+00
25%      5.400000e+01
50%      9.500000e+01
75%      1.680000e+02
max      1.862000e+03
dtype: float64
```

Figure 4.1: Distribution of number of tokens

The resulting CSV file contained on each line a triplet: `(review_id, sentiment, score)`

### 4.3.3. Collections update

Both `reviews`, `users_merged` and `businesses_merged` had to be updated by adding the new fields. To do this in an efficient way, I converted the resulting DataFrame to a dictionary and then, I leveraged `pyMongo.updateOne()` along with `bulk_write()`. I chose to use `bulk_write()` to reduce as much as possible the number of transactions; namely, this method is able to perform multiple write operations (inserts, updates or deletes) in a single request to the database.

Different collections required different update strategies:

- In `reviews` collection, each document contains a single review and an index was present on `review_id`, so it was sufficient to filter on it:

```
db["reviews"].bulk_write([
    pymongo.UpdateOne({
          "review_id" : review_id
      },
      {
          "$set" : data
      })
      for review_id, data in batch_items
])
```

- In `businesses_merged`, each `business_id` mapped a set of reviews. So, I needed a dictionary `review_id` $\longrightarrow$ `business_id`
and an additional index on `reviews.review_id` that is sparse because some documents might not have `reviews` sub-collection:

```
db["businesses_merged"].create_index(["reviews.review_id"],
                                     unique = True, sparse = True)
```

In the filter condition of `updateOpne()`, the review is located by looking at `business_id` and `reviews.review_id`. Then, in `$set`, `$` is used to catch the matching document(s) to insert the new fields into.

```
db["businesses_merged"].bulk_write([
      pymongo.UpdateOne({
            "business_id" :review_to_business_map[review_id],
            "reviews.review_id" : review_id
        },
        {
            "$set" : {
                "reviews.$.sentiment" : data["sentiment"],
                "reviews.$.confidence" : data["confidence"]
            }
        })
        for review_id, data in batch_items
```

```
])
```

- In `users_merged` same challenges of `businesses_merged` were faced. I repeated what is shown in the previous case.

# 5 | Queries

In this chapter, the user is presented with the queries performed in MongoDB. All of them were executed within a Jupyter Notebook connected to MongoDB instance using `PyMongo`. Some interesting results were visualized using `matplotlib.pyplot`, `seaborn` libraries.

## 5.1. Query 1: how many businesses and reviews for each category?

The purpose of this query was to count the total number of businesses and reviews for each category of businesses.

```
db["businesses_merged"].aggregate([
    {
        "$project" : {
            "categories" : 1,
            "reviews" : 1,
            "business_id" : 1
        }
      },
    {
        "$unwind" : "$categories"
    },
    {
        "$unwind" : "$reviews"
    },
    {
        "$group" : {
            "_id" : "$categories",
            "num_reviews" : {
                "$sum" : 1
```

```
            },
            "ids" : {
                "$addToSet" : "$business_id"
            }
        }
    },
    {
        "$project" : {
            "_id" : 0,
            "category" : "$_id",
            "num_businesses" : {
              "$size" : "$ids"
            },
            "num_reviews" : 1
        }
    },
    {
        "$sort" : {
            "num_businesses" : -1,
            "num_reviews" : -1
        }
    }
])
```

This query is a clear example of applying NoSQL principles (in particular, document embedding), as it uses documents with sub-collections and deconstructs nested arrays using `$unwind`. In contrast, the following query is more aligned with traditional SQL principles, as it relies on `$lookup` to join collections:

```
db["reviews"].aggregate([
    {
        "$lookup" : {
            "from" : "businesses",
            "let" : {
                "id" : "$business_id"
            },
            "pipeline" : [{
                            "$match" : {
                                "$expr" : {
```

```
                                    "$eq" : ["$business_id", "$$id"]
                                }
                            }
                        }],
            "as" : "business"
        }
    },
    {
        "$unwind" : "$business"
    },
    {
        "$unwind" : "$business.categories"
    },
    {
        "$group" : {
            "_id" : "$business.categories",
            "num_reviews" : {
                "$sum" : 1
            },
            "ids" : {
                "$addToSet" : "$business.business_id"
            }
        }
    },
    {
        "$project" : {
            "_id" : 0,
            "category" : "$_id",
            "num_businesses" : {
                "$size" : "$ids"
            },
            "num_reviews" : 1
        }
    },
    {
        "$sort" : {
            "num_businesses" : -1,
            "num_reviews" : -1
        }
```

```
    }
])
```

In the notebook, under `Query 1` subsection, the reader will find another version of this query (here not shown for the sake of shortness) that combines MongoDB and Pandas.

Although taking different times, all the queries produce the same resulting DataFrame:

| | num_reviews | category | num_businesses |
|---|---|---|---|
| 0 | 4724464 | Restaurants | 52268 |
| 1 | 1813588 | Food | 27781 |
| 2 | 523251 | Shopping | 24395 |
| 3 | 238255 | Home Services | 14356 |
| 4 | 370120 | Beauty & Spas | 14292 |
| ... | ... | ... | ... |
| 1306 | 5 | DUI Schools | 1 |
| 1307 | 5 | Faith-based Crisis Pregnancy Centers | 1 |
| 1308 | 5 | Fencing Clubs | 1 |
| 1309 | 5 | Town Hall | 1 |
| 1310 | 5 | Hepatologists | 1 |

Figure 5.1: Number businesses and reviews per category

The result show that there are $1,311$ different categories. *"Restaurants"* and *"Food"* categories dominate in terms of both the number of reviews (4.7M and 1.8M, respectively) and businesses ($52,268$ and $27,781$), making them highly competitive sectors.

On the other hand, categories like *"Home Services"* ($238,255$ reviews, $14,356$ businesses) and *"Beauty & Spas"* ($370,120$ reviews, $14,292$ businesses) are subject to moderate competition with still a quite significant demand.

A possible conclusion is that, even if the first categories may still offer growth opportunities, new investments would probably have more success in the others. For niche opportunities, categories with fewer businesses but steady demand, such as "Shopping", could also be worth exploring.

## 5.2. Query 2: for each category, find the most polarizing business

A business is said to be *"polarizing"* if it is one with high variance in star rating. The objective of this query was to identify, for each category, the most polarizing business.

For this query, `$stdDevPop` was employed.

```
db["reviews"].aggregate([
    {
        "$project" : {
            "business_id" : 1,
            "categories" : 1,
            "reviews.stars" : 1,
            "name" : 1
        }
    },
    {
        "$unwind" : "$categories"
    },
    {
        "$unwind" : "$reviews"
    },
    {
        "$group" : {
            "_id" : {
                "r_id" : "$business_id",
                "category" : "$categories"
            },
            "name" : {
                "$first" : "$name"
            },
            "avg_stars" : {
                "$avg": "$reviews.stars"
            },
            "variance" : {
                "$stdDevPop": "$reviews.stars"
            },
            "num_reviews" : {
```

```
32                  "$sum" : 1
33              }
34          }
35      },
36      {
37          "$sort" : {
38              "category" : 1,
39              "variance" : -1
40          }
41      },
42      {
43          "$group" : {
44              "_id" : "$_id.category",
45              "name" : {
46                  "$first" : "$name"
47              },
48              "variance" : {
49                  "$first" : "$variance"
50              },
51              "avg_stars" : {
52                  "$first": "$avg_stars"
53              },
54              "num_reviews" : {
55                  "$first": "$num_reviews"
56              }
57          }
58      },
59      {
60          "$project" : {
61              "_id" : 0,
62              "category" : "$_id",
63              "name" : 1,
64              "avg_stars" : 1,
65              "num_reviews" : 1,
66              "variance" : 1,
67          }
68      },
69      {
70          "$sort" : {
```

```
            "variance" : -1
        }
    }
])
```

For some categories ("Medical Centers", "Restaurants", "Pets", "Optometrists", "Aquarium Services", "Somali"), number of reviews along with average and variance were plotted:

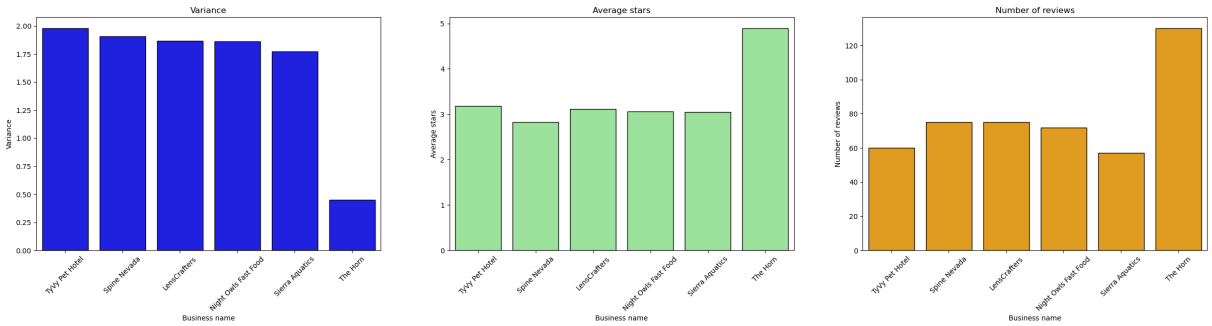


Figure 5.2: Most polarizing businesses for some categories

The entire DataFrame is the following:

| | name | variance | avg_stars | num_reviews | category |
|---|---|---|---|---|---|
| 0 | TyVy Pet Hotel | 1.978987 | 3.183333 | 60 | Veterinarians |
| 1 | TyVy Pet Hotel | 1.978987 | 3.183333 | 60 | Pet Groomers |
| 2 | TyVy Pet Hotel | 1.978987 | 3.183333 | 60 | Pet Boarding |
| 3 | TyVy Pet Hotel | 1.978987 | 3.183333 | 60 | Pet Services |
| 4 | TyVy Pet Hotel | 1.978987 | 3.183333 | 60 | Pets |
| ... | ... | ... | ... | ... | ... |
| 1036 | Tremendez Jewelry and Repair | 0.133609 | 4.981818 | 55 | Engraving |
| 1037 | Gators Parasail | 0.000000 | 5.000000 | 66 | Sport Equipment Hire |
| 1038 | Music City Cats | 0.000000 | 5.000000 | 54 | House Sitters |
| 1039 | Marc Bosserman Pianist & Vocalist | 0.000000 | 5.000000 | 55 | Music Production Services |
| 1040 | AllVitae Health & Chiropractic | 0.000000 | 5.000000 | 67 | Health Coach |

Figure 5.3: DataFrame of most polarizing businesses for some categories

In the query, only businesses with at least 50 reviews were selected to minimize potential bias in the variance. Notably, the *"restaurants"* category stands out as one of those

with the highest variance: this is reasonable because restaurants cater to a wide range of customer preferences that leads to greater variability in reviews. Also, the categories *"pets"* and *"medical centers"* show high variance. For the first, this could be due to the emotional weight of customers experience with pet services. For the latter, it might come from from the critical nature of healthcare services, where individual experiences can vary dramatically based on the quality of care, wait times, or outcomes.

Note that the number of reviews and the average star ratings are relatively consistent across categories: this is important because it means that a uniform sample was chosen.

## 5.3. Query 3: find in which city a chain of businesses performs the best

A chain of businesses is a set of `business_id` with the same `name` but different `city`. The goal of this query was to discover in which city a chain of businesses performs the best. Note that in a city there could be more than one `business_id` with the same name.

In the first step of the query, all the necessary documents were retrieved using MongoDB:

- `more_than_one_city`: all chains of businesses were collected.

```
db["businesses"].aggregate([{
        "$group" : {
            "_id" : {
                "name" : "$name",
            },
            "cities" : {
                "$addToSet" : "$city"
            }
        }
    },
    {
        "$match" : {
            "$expr" : {
                "$gt" : [{"$size" : "$cities"}, 1]
            }
        }
    },
    {
        "$project" : {
```

```
            "_id" : 0,
            "name" : "$_id.name",
            "num_cities" : {"$size" : "$cities"}
        }
    }
])
```

- `ratings`: for each business name and each city, the average rating was computed.

```
db["businesses_merged"].aggregate([{
    "$project" : {
        "name" : 1,
        "city" : 1,
        "reviews.stars" : 1,
    }
},
{
        "$unwind" : "$reviews"
    },
    {
        "$group" : {
            "_id" : {
                "name" : "$name",
                "city" : "$city"
            },
            "avg_stars" : {
                "$avg" : "$reviews.stars"
            }
        }
    },
    {
        "$project" : {
            "_id" : 0,
            "name" : "$_id.name",
            "city" : "$_id.city",
            "avg_stars" : 1
        }
    },
    {
```

```
        "$sort" : {
            "name" : 1,
            "avg_stars" : -1
        }
    },
    {
        "$group" : {
            "_id" : "$name",
            "city" : {
                "$first" : "$city"
            },
            "avg_stars" : {
                "$first" : "$avg_stars"
            }
        }
    },
    {
        "$project" : {
            "_id" : 0,
            "name" : "$_id",
            "city" : 1,
            "avg_stars" : 1
        }
    }
])
```

In the second step, the intersection between `more_than_one_city` and `ratings` was computed using Pandas:

```
df = pd.DataFrame(ratings)
df = df[df["name"].isin(more_than_one_city["name"])]
```

I opted for a composite query because computing and intersecting `more_than_one_city` and `ratings` in a single query wasn't trivial (it could be done, but it would require the usage of `$facet`).

The results are summarized in the following DataFrame:

| | city | avg_stars | name |
|---|---|---|---|
| 32 | Boise | 3.878261 | Grind Modern Burger |
| 43 | Tampa | 5.000000 | Walesby Vision Center |
| 51 | Shrewsbury | 4.297872 | Pizza World |
| 57 | Tampa | 4.006711 | ProntoWash |
| 70 | Clearwater | 4.000000 | Tampa Bay Grand Prix |
| ... | ... | ... | ... |
| 114035 | West Point | 3.903846 | Boyd's Cardinal Hollow Winery |
| 114037 | Saint Charles | 4.714286 | Ziebart |
| 114098 | Glenside | 3.469388 | Luigi's Pizza & Pasta |
| 114099 | Saint Louis | 3.928571 | Nail Club |
| 114103 | St. Petersburg | 5.000000 | Restore Hyper Wellness |

Figure 5.4: Best city for chains of businesses

## 5.4. Query 4: establish whether local competitions influences average and variance review stars

Two businesses $b_1$ and $b_2$ are said to be competitors if they share at least one category (i.e. $b_1$.category $\cap$ $b_2$.category $\neq \{\emptyset\}$ and they work in the same city (i.e. $b_1$.city = $b_2$.city).

With this query, I studied whether there was a relation between the average (variance) of reviews' stars and the number of competitors for the most widespread businesses.

As done in the previous query, I proceeded with a two-steps query. In the first one, all relevant documents were shaped and retrieved:

- `competitors`: for each `name` and `city`, the number of competitors was computed.

```
db["businesses"].aggregate([{
        "$unwind" : "$categories"
    },
    {
        "$group" : {
            "_id" : {
                "city" :"$city",
```

```
8                "category" : "$categories"
9            },
10           "businesses_names" : {
11               "$addToSet" : "$name"
12           }
13       }
14   },
15   {
16       "$addFields" : {
17           "businesses_names_copy" : "$businesses_names"
18       }
19   },
20   {
21       "$unwind" : "$businesses_names"
22   },
23   {
24       "$group" : {
25           "_id" : {
26               "name" : "$businesses_names",
27               "city" : "$_id.city"
28           },
29           "possible_competitors" : {
30               "$addToSet" : "$businesses_names_copy"
31           }
32       }
33   },
34   {
35       "$project" : {
36           "_id" : 1,
37           "possible_competitors" : {
38               "$filter" : {
39                   "input" : "$possible_competitors",
40                   "as" : "pc",
41                   "cond" : {
42                       "$ne" : ["$$pc", "$_id.name"]
43                   }
44               }
45           }
46       }
```

```
47      },
48      {
49          "$addFields" : {
50              "num_competitors" : {
51                  "$size" : "$possible_competitors"
52              }
53          }
54      },
55      {
56          "$project" : {
57              "_id" : 0,
58              "city" : "$_id.city",
59              "name" : "$_id.name",
60              "num_competitors" : 1
61          }
62      }
63  ])
```

The reasoning behind this query is quite complex; to fully understand it, it's better to outline the structure oof documents after each stage:

1. In the `$unwind` stage documents with multiple categories were split, creating separate documents for each category.
2. Documents were grouped by `(city, category)` pairs using `$group` and, for each group, the unique business names were collected into `businesses_names`.
3. A copy of `businesses_names` was created as `businesses_names_copy` for later reference.
4. The `businesses_names` array was unwound: in each document a certain `(business_name, category)` is associated with all the competitors `business_name` for that category.
5. Then, documents were regrouped by `(name, city)`, collecting all businesses that share any category from `businesses_names_copy` arrays (i.e. potential competitors).
6. With `$project` and `$filter`, each business was removed from its own list of competitors.
7. Number of competitors for each business was computed using `$size`.

8. Lastly, the output was formatted to show `city`, `name`, and `num_competitors` for each business.

- `ratings`: for each business, average and standard deviation of reviews' stars were computed.

```
db["businesses_merged"].aggregate([
    {
        "$unwind" : "$reviews"
    },
    {
        "$group" : {
            "_id" : {
                "name" : "$name",
                "city" : "$city"
            },
            "avg_stars" : {
                "$avg" : "$reviews.stars"
            },
            "star_variance" : {
                "$stdDevPop" : "$reviews.stars"
            }

        }
    },
    {
        "$project" : {
            "_id" : 0,
            "name" : "$_id.name",
            "city" : "$_id.city",
            "avg_stars" : 1,
            "star_variance" : 1
        }
    }
])
```

In the second step of the query, the two resulting DataFrames were merged using Pandas:

```
ratings_df = pd.DataFrame(ratings)
competitors_df = pd.DataFrame(competitors)
df = pd.merge(ratings_df, competitors_df,
              left_on = ["name", "city"], right_on = ["name", "city"])
```

Some results:

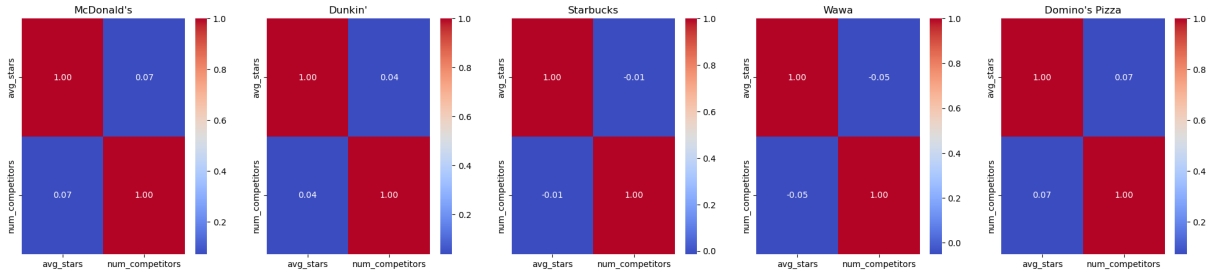


Figure 5.5: Average - variance vs competition for 1st to 5th most widespread businesses

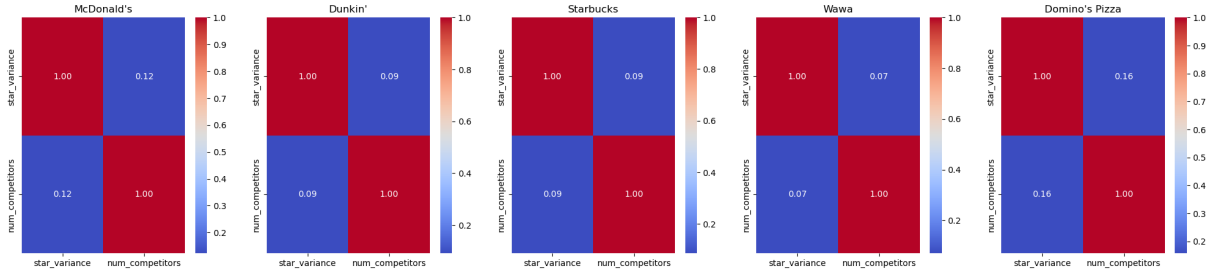


Figure 5.6: Variance vs competition for 1st to 5th most widespread businesses

This results reveal mixed trends. For instance, *Wawa* in Souderton has a high average rating (4.19) with 8 competitors, while *Domino's Pizza* in Normandy and *McDonald's* in Lawrence, with fewer competitors (3 and 5, respectively), exhibit much lower ratings (1.56 and 1.72).

Further analysis was executed on the 50th to 55th most widespread businesses and it confirmed the complexity. *Waffle House* in Tampa, despite 10 competitors, maintains a strong average rating (3.57), while *Walmart* in Exton has 6 competitors and is rated 1.87.

The star variance also varies widely, For example, *Dairy Queen Grill & Chill* in Spring Hill shows high variance (1.77), whereas *Waffle House* in Lawrence has a more consistent satisfaction level (1.23).

I concluded that competition may influence ratings, but this is not an universal rule because the relationship is shaped, also, by local factors and customer expectations.

## 5.5. Query 5: are day and night reviews biased?

The aim of this query was to investigate whether there was a discrepancy in the reviews of the day and night for each user. Such discrepancy was estimated using average star rating.

```
db["reviews"].aggregate([{
        "$project" : {
            "user_id" : 1,
            "date" : 1,
            "stars" : 1
        }
    },
    {
        "$addFields" : {
            "reviewHour" : {
                "$hour": "$date"
            }
        }
    },
    {
        "$addFields" : {
            "is_night" : {
                "$or" : [
                    {"$gte": ["$reviewHour", 22]},
                    {"$lt": ["$reviewHour", 5]}
                ]
            }
        }
    },
    {
        "$group" : {
            "_id" : "$user_id",
            "num_night_reviews" : {
                "$sum" : {
                    "$cond" : ["$is_night", 1, 0]
```

```
                }
            },
            "num_day_reviews" : {
                "$sum" : {
                    "$cond" : ["$is_night", 0, 1]
                }
            },
            "avg_night_ratings" : {
                "$avg" : {
                    "$cond" : ["$is_night", "$stars", None]
                }
            },
            "avg_day_ratings" : {
                "$avg" : {
                    "$cond" : ["$is_night", None, "$stars"]
                }
            }
        }
    },
    {
        "$match": {
            "num_night_reviews" : {
                "$gte": 5
            },
            "num_day_reviews" : {
                "$gte": 5
            },
        }
    },
    {
        "$project" : {
            "_id": 0,
            "user_id": "$_id",
            "num_night_reviews": 1,
            "num_day_reviews": 1,
            "avg_night_ratings": 1,
            "avg_day_ratings": 1
        }
    }
```

```
70 ])
```

In the notebook, under `Query 5` subsection, the reader will find another version of this query (here not shown for the sake of shortness) that combines MongoDB and Pandas.

Some results:

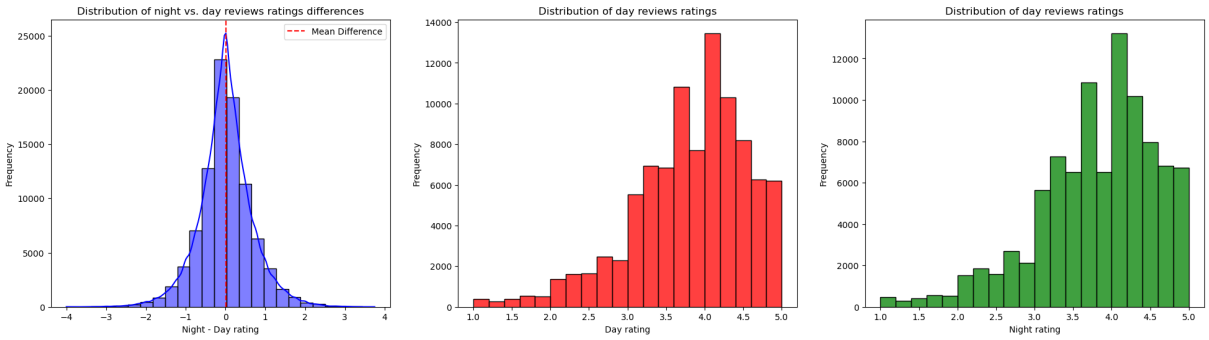


Figure 5.7: Distribution of night vs. day reviews ratings differences

The distribution of reviews is stable across day and night. The mean difference is 0: this means that the majority of users give similar ratings during day and night. However, a part of users seems to be stricter during night because bars below 0 are higher than the symmetric ones (w.r.t $x = 0$). A deeper statistical analysis could determine if these patterns are meaningful or coincidental.

## 5.6. Query 6: friends have similar ratings?

This query was designed to examine the similarity in rating behaviours among friends by focusing on businesses they have both reviewed. Such similarity was quantified in terms of difference between average review stars.

As done in the previous queries, I proceeded in two steps. In the first one, all relevant documents were shaped and retrieved:

- `avg_rating`: for each `user_id` and `business_id`, average star rating was computed.

```
1 db["businesses_merged"].aggregate([{
2         "$unwind" : "$reviews"
3     },
4     {
```

```
        "$group" : {
            "_id" : {
                "user_id" : "$reviews.user_id",
                "business_id" : "$business_id"
            },
            "avg_rating" : {
                "$avg" : "$reviews.stars"
            }
        }
    },
    {
        "$match" : {
            "avg_rating" : {
                "$ne" : None
            }
        }
    },
    {
        "$project" : {
            "_id" : 0,
            "user_id" : "$_id.user_id",
            "business_id" : "$_id.business_id",
            "avg_rating" : 1
        }
    }
])
```

- `common_ratings`: for each `user_id` and `user_id` of friends, `business_id` of common reviewed businesses were found.

```
db["users_merged"].aggregate([{
        "$project" : {
            "user_id" : 1,
            "reviews" : "$reviews.business_id",
            "friends" : 1
        }
    },
    {
        "$lookup" : {
```

```
10              "from" : "users_merged",
11              "localField" : "friends",
12              "foreignField" : "user_id",
13              "pipeline" : [{
14                  "$project" : {
15                      "user_id" : 1,
16                      "reviews" : "$reviews.business_id"
17                  }
18              }],
19              "as" : "friend_reviews"
20          }
21      },
22      {
23          "$project" : {
24              "friends" : 0
25          }
26      },
27      {
28          "$addFields" : {
29              "shared_reviews" : {
30                  "$map" : {
31                      "input" : "$friend_reviews",
32                      "as" : "f_r",
33                      "in" : {
34                          "friend_id" : "$$f_r.user_id",
35                          "businesses" : {
36                              "$setIntersection" : ["$reviews",
    "$$f_r.reviews"]
37                          }
38                      }
39                  }
40              }
41          }
42      },
43      {
44          "$project" : {
45              "reviews" : 0,
46              "friend_reviews" : 0
47          }
```

```
    },
    {
        "$addFields" : {
            "shared_reviews" : {
                "$filter" : {
                    "input" : "$shared_reviews",
                    "as" : "s_r",
                    "cond" : {
                        "$gt": [{"$size": "$$s_r.businesses"}, 0]
                    }
                }
            }
        }
    },
    {
        "$unwind" : "$shared_reviews"
    },
    {
        "$project": {
            "_id" : 0,
            "user_id" : 1,
            "shared_reviews" : 1
        }
    }
])
```

To better understand the reasoning behind this query, the structure of the documents is presented step by step:

- `$project`: `user_id field`, `business_id` from the reviews array and `friends` are preserved.

```
{
    "user_id": "string",
    "reviews": ["business_id1", "business_id2", ...],
    "friends": ["friend_id1", "friend_id2", ...]
}
```

- `$lookup`: self-join operation with the `users_merged` collection on `friends`

array and `user_ids`. The document is expanded into:

```
{
    "user_id": "string",
    "reviews": ["business_id1", "business_id2", ...],
    "friends": ["friend_id1", "friend_id2", ...],
    "friend_reviews": [
        {
            "user_id": "friend_id1",
            "reviews": ["business_id1", "business_id3
               ", ...]
        },
        ...
    ]
}
```

– `$project`: now-redundant `friends` array is removed.

– `$addFields`: with `$map` operator, the intersection of reviewed businesses between the user and each friend was computed.

```
{
    "user_id": "string",
    "reviews": ["business_id1", "business_id2", ...],
    "friend_reviews": [...],
    "shared_reviews": [
        {
            "friend_id": "friend_id1",
            "businesses": ["shared_business_id1", ...
               ]
        },
        ...
    ]
}
```

– `$project`: `reviews` and `friend_reviews` arrays are removed.

– `$addFields`: with `$filter` entries where users share no common reviews are removed. Structure is same as previous.

– `$unwind`: a single user-friend relationship is now represented in a document.

```
{
    "user_id": "string",
    "shared_reviews": {
        "friend_id": "friend_id1",
        "businesses": ["shared_business_id1", "
           shared_business_id2", ...]
    }
}
```

– `$project`: `_id` field is removed, resulting in the following:

```
{
    "user_id": "string",
    "shared_reviews": {
        "friend_id": "friend_id1",
        "businesses": ["shared_business_id1", "
           shared_business_id2", ...]
    }
}
```

- In the second stage of the query, I combined the results collected before. Because DataFrame indexing is slow, I transformed `avg_rating` in a dict of dicts:

```
avg_rating.reset_index()\
          .groupby("user_id")\
          .apply(lambda g: dict(zip(g["business_id"], g["avg_rating"])))\
          .to_dict()
```

Then, for each document in `common_reviewed`, I retrieved ratings for both the user and one of his / her friend (there could be more than one commonly reviewed businesses) and their averages were computed:

```
result = []

for doc in common_reviewed:
    user_id = doc["user_id"]
```

```
        friend_id = doc["shared_reviews"]["friend_id"]
        businesses = doc["shared_reviews"]["businesses"]

        avg_user_ratings = [avg_rating_dict[user_id][b] for b in businesses]
        avg_friend_ratings = [avg_rating_dict[user_id][b] for b in businesses]

        result.append({
            "user_id" : user_id,
            "friend_id" : friend_id,
            "avg_rating_user" : pd.Series(avg_user_ratings).mean(),
            "avg_rating_friend" : pd.Series(avg_friend_ratings).mean()
        })

    result_df = pd.DataFrame(result)
    result_df
```

The following histogram shows the results:

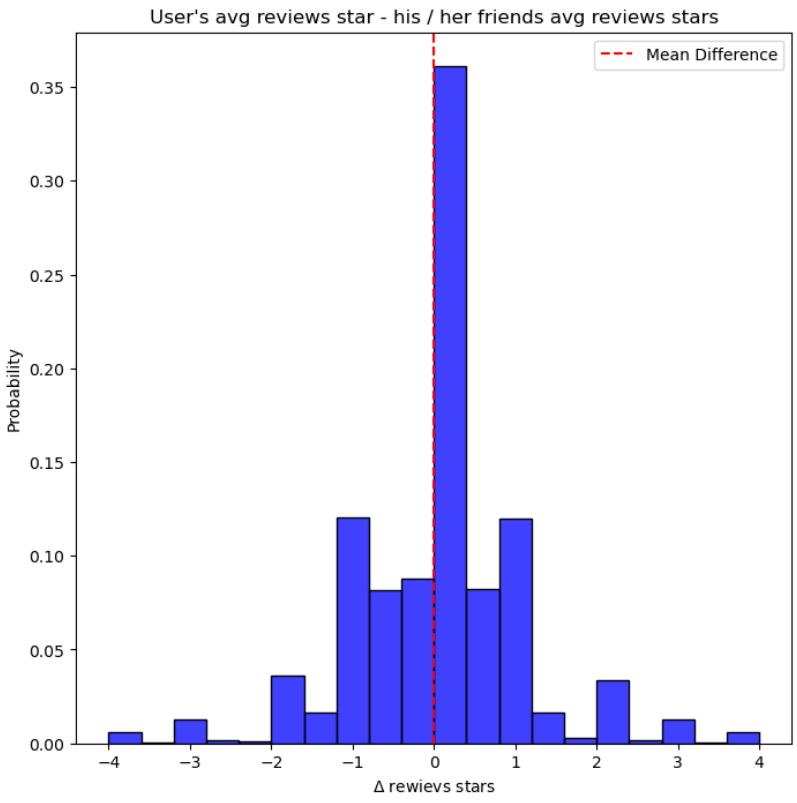


Figure 5.8: User's average reviews star - his / her friends average reviews stars

The distribution is centered around zero, which suggests that users and their friends tend to give similar ratings: with probability of 0.35, the $\Delta$ it's exactly in 0-bin and if we consider $-1$ to 1 bins, about 0.75 of total friend pairs are covered. There are, however, a few cases where the differences in ratings are more extreme, ranging from $-4$ to $+4$ (users either rated a business significantly higher or lower than their friends). Despite these deviations, the red dashed line, representing the mean difference, lies at zero.

## 5.7. Query 7: yearly growth rate of businesses

The growth rate of a business is defined as $g_{B,i} = \frac{n_{B,i+1} - n_{B,i}}{n_{B,i}}$ where $n_{B,i}$ is the number of check-ins for business $B$ in year $i$. The objective of this query was to compute the growth rate of all businesses such that $\exists i, j \in [\texttt{start\_date}, \texttt{end\_date}] \mid i < j \wedge \forall k \in [i, j] \mid n_{B,k} > 0$ with `start_date` and `end_date` query parameters.

```
db["businesses_merged"].aggregate([{
        "$unwind" : "$checkins"
    },
    {
        "$addFields" : {
            "year_checkin" : {
                "$year" : "$checkins"
            }
        }
    },
    {
        "$match" : {
            "$and" : [{"year_checkin" : {"$gte" : start}},
                    {"year_checkin" : {"$lte" : end}}]
        }
    },
    {
        "$group" : {
            "_id" : {
                "name" : "$name",
                "city" : "$city",
                "year" : "$year_checkin"
            },
            "num_checkins" : {
                "$sum" : 1
```

```
            }
        }
    },
    {
        "$sort" : {
            "_id.name" : 1,
            "_id.city" : 1,
            "_id.year" : 1
        }
    },
    {
        "$group" : {
            "_id" : {
                "name" : "$_id.name",
                "city" : "$_id.city"
            },
            "checkins_sequence" : {
                "$push" : {
                    "year" : "$_id.year",
                    "num_checkins" : "$num_checkins"
                }
            }
        }
    },
    {
        "$project" : {
            "_id" : 0,
            "name" : "$_id.name",
            "city" : "$_id.city",
            "checkins_sequence" : 1
        }
    }
])
```

There were businesses with $n_{B,i} = 0$ and $n_{B,i-1} \neq 0 \ \wedge \ n_{B,i+1} \neq 0$, so the results of this query had to be filtered and the growth had to be computed. The Python code is available under `Query 7` subsection of `Queries` notebook. In particular, note the `calculate_and_filter_trend` method that takes in input a `lambda` function `filter`

to specify conditions on growth rate.

For the business with the longest sequence of registered check-ins, both $n_{B,i}$ and $g_{B,i}$ were plotted:

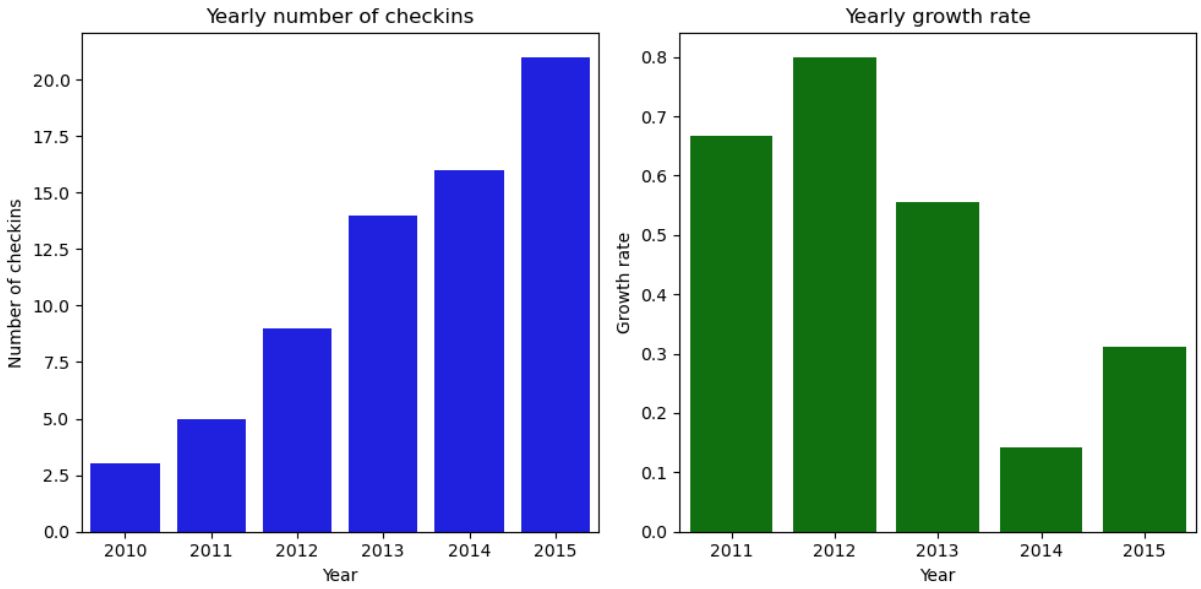


Figure 5.9: Check-ins and yearly growth rate sequence for the selected business

## 5.8. Query 8: cities with lowest sentiment for each category

For each category, the cities with the lowest sentiment for that category were determined. To have a less biased result, only categories with sufficient number of positive and negative reviews (at least 50 for each one) were considered.

```
db["businesses_merged"].aggregate([{
        "$unwind": "$reviews"
    },
    {
        "$unwind" : "$categories"
    },
    {
        "$group" : {
            "_id" : {
```

```
                "city" : "$city",
                "category": "$categories"
            },
            "num_positive_reviews" : {
                "$sum" : {
                    "$cond" : [{"$eq" : ["$reviews.sentiment", "positive"]},
    1, 0]
                }
            },
            "num_negative_reviews" : {
                "$sum" : {
                    "$cond" : [{"$eq" : ["$reviews.sentiment", "negative"]},
    1, 0]
                }
            }
        }
    },
    {
        "$match" : {
            "num_positive_reviews" : {
                "$gte": 50
            },
            "num_negative_reviews" : {
                "$gte" : 50
            }
        }
    },
    {
        "$addFields" : {
            "neg_to_pos_ratio" : {
                "$divide" : ["$num_negative_reviews", "$num_positive_reviews"]
            }
        }
    },
    {
        "$sort" : {
            "city" : 1,
            "name" : 1,
            "neg_to_pos_ratio" : -1
```

```
        }
    },
    {
        "$project" : {
            "_id" : 0,
            "city" : "$_id.city",
            "category" : "$_id.category",
            "tot_reviews" : {
                "$sum" : ["$num_positive_reviews", "$num_negative_reviews"]
            },
            "num_negative_reviews" : 1,
            "num_positive_reviews" : 1,
            "neg_to_pos_ratio" : 1
        }
    }

])
```

Upon having all documents, using Pandas, widespread categories (i.e. categories in resulting DataFrame present in more than 15 lines) were computed and for the top 3 an histogram was plotted:

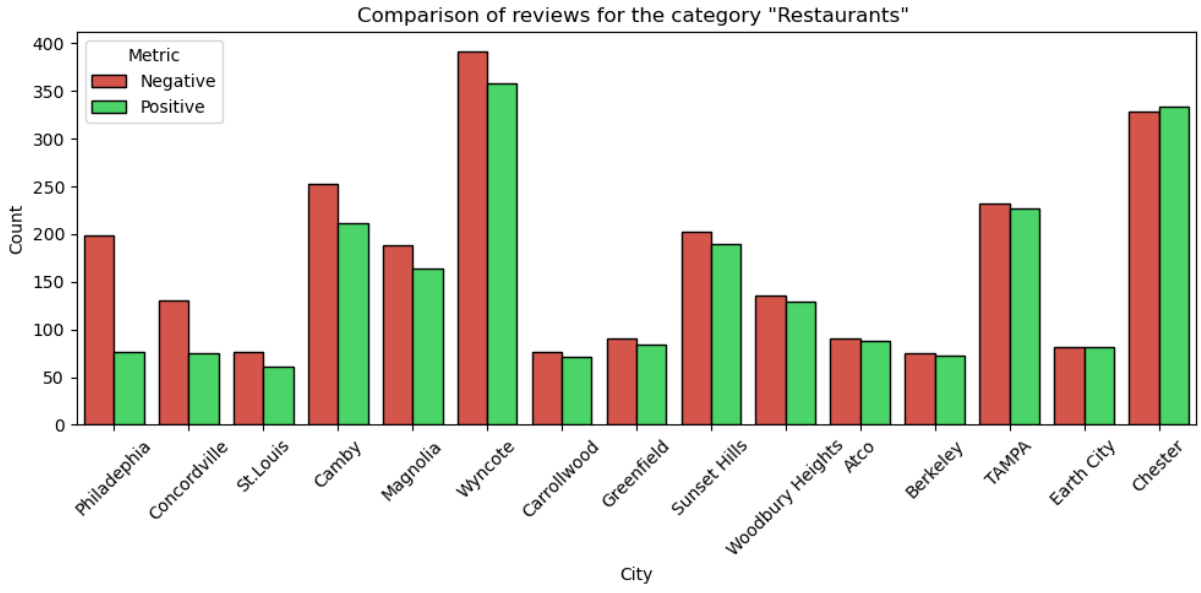


Figure 5.10: Comparison of reviews sentiment for the category "Restaurants"

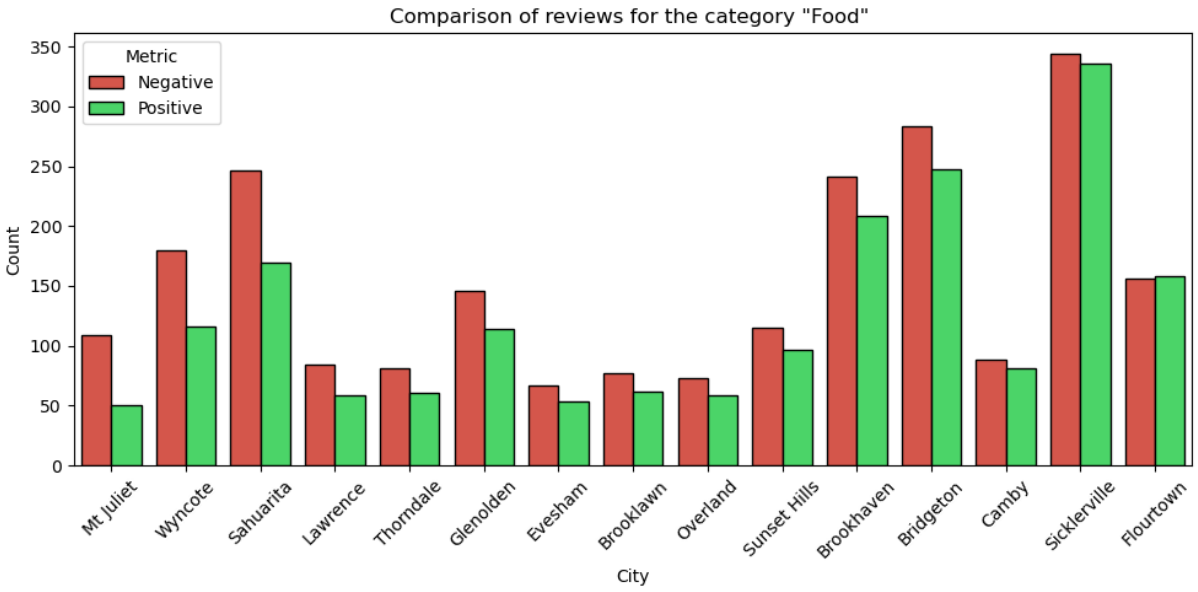


Figure 5.11: Comparison of reviews sentiment for the category "Food"

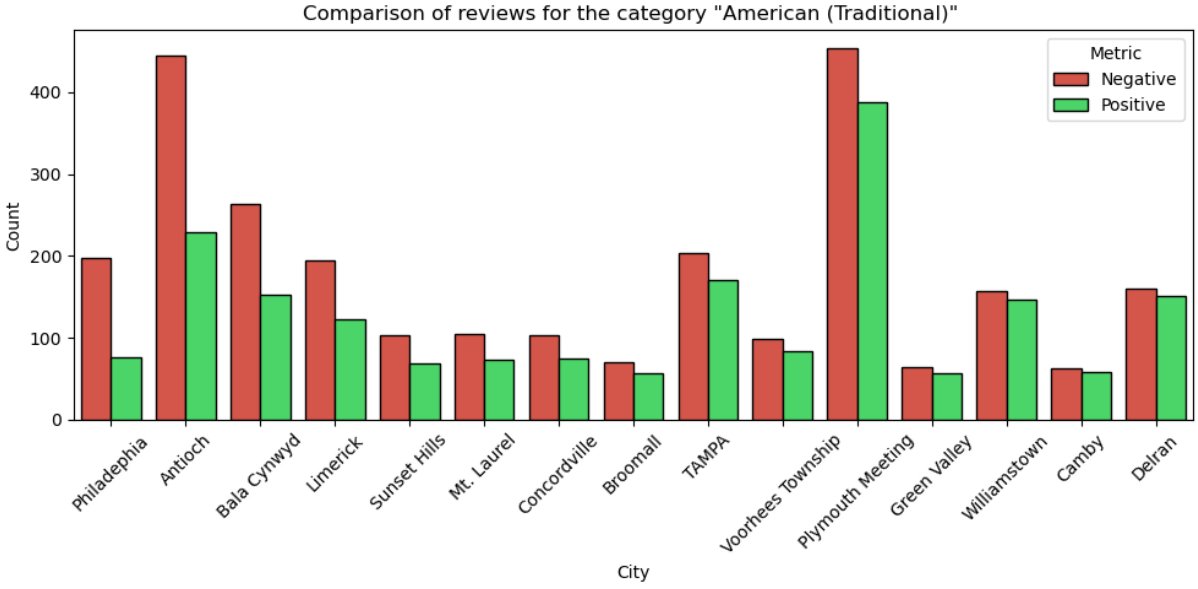


Figure 5.12: Comparison of reviews sentiment for the category "American (traditional)"

For all categories, the sentiment distribution varies significantly between cities. Some cities show a clear dominance of negative reviews, while others have a more balanced distribution or even a higher number of positive reviews: this indicates a localized sentiment trend for lower / higher quality in certain areas.

In addition, cities with higher total reviews often have larger disparities between positive and negative sentiments, suggesting that review counts and quality perceptions may correlate, with popular locations potentially attracting more polarized opinions.

## 5.9. Query 9: number of businesses for each bucket of possible reviews ratings

The aim of this query was to count the number of businesses for each bucket (or bin) of possible review ratings. In particular, I defined:

$$\texttt{buckets} = [0, 0.5, ..., 4.5, 5]$$

and counted how many businesses had their average review rating in [`buckects[i]`, `buckets[i + 1]`), using `$bucket` operator.

```
db["businesses_merged"].aggregate([{
        "$project" : {
            "business_id" : 1,
            "reviews.stars" : 1
        }
    },
    {
        "$unwind" : "$reviews"
    },
    {
        "$group" : {
            "_id" : "$business_id",
            "avg_rating" : {
                "$avg" : "$reviews.stars"
            },
            "num_reviews" : {
                "$sum" : 1
            }
        }
    },
    {
        "$match" : {
            "num_reviews" : {
```

```
                "$gte" : 25
            }
        }
    },
    {
        "$bucket" : {
            "groupBy" : "$avg_rating",
            "boundaries" : boundaries,
            "default" : "other",
            "output" : {
                "count" : {
                    "$sum" : 1
                }
            }
        }
    },
    {
        "$densify" : {
            "field" : "_id",
            "range" : {
                "step" : step,
                "bounds" : [0.5, 5.0 + 0.5]
            }
        }
    },
    {
        "$set" : {
            "count" : {
                "$cond" : [{"$not" : ["$count"]}, 0, "$count"]
            }
        }
    },
    {
        "$project" : {
            "boundary" : "$_id",
            "_id" : 0,
            "count" : 1
        }
    }
```

```
63 ])
```

The following histogram shows the results:

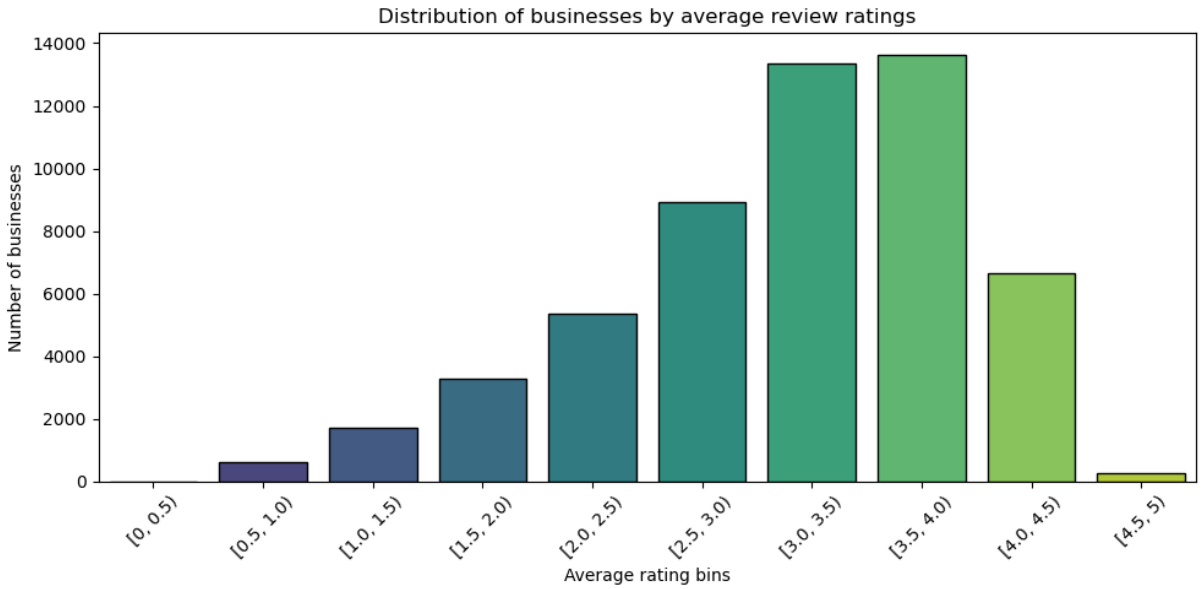


Figure 5.13: Distribution of businesses by average review ratings

Most businesses have average review ratings in the range of $3.0 - 4.0$, with about $14,000$ for both $[3.0, 3.5)$ and $[3.5, 4.0)$. As ratings decrease below 3.0 or increase above 4.0, the number of businesses drops significantly: very few businesses have average ratings below 0.5 or above 4.5.

## 5.10. Query 10: investigate the relation between sentiment and stars of reviews

As mentioned in 4.1, DistilBert's outputs come with a confidence measure (also called *score*).

For each star value, I counted the number of reviews with positive and negative sentiments. Then, I computed the minimum confidence $m = \min_i \text{confidence}_i$ and $M = \max_i \text{confidence}_i$ and defined:

$$\left[m, m + \frac{1}{4}(M-m), m + \frac{1}{2}(M-m), m + \frac{3}{4}(M-m), M\right]$$

For each star and bin, I counted the number of reviews with the that number of stars and confidence in that bin.

Two steps were performed for this query:

- `min_max_confidence`: minimum and maximum confidence were computed for all reviews.

```
    db["reviews"].aggregate([{
        "$project" : {
            "_id" : 1,
            "confidence" : 1
        }
    },
    {
        "$group" : {
            "_id" : True,
            "max_confidence" : {
                "$max" : "$confidence"
            },
            "min_confidence" : {
                "$min" : "$confidence"
            }
        }
    },
    {
        "$project" : {
            "_id" : 0
        }
    }
])
```

- From `min_max_confidence`, vector of bins was defined using Python (here not shown for the sake of shortness).
- `query_results`: bucket count of `confidence` was performed.

```
db["reviews"].aggregate([{
        "$project" : {
            "sentiment" : 1,
```

```
            "confidence" : 1,
            "stars" : 1
        }
    },
    {
        "$group" : {
            "_id" : {
                "stars" : "$stars",
                "sentiment" : "$sentiment"
            },
            "num_reviews" : {
                "$sum" : 1
            },
            **bins_fields
        }
    },
    {
        "$project" : {
            "stars" : "$_id.stars",
            "sentiment" : "$_id.sentiment",
            "_id" : 0,
            "max_confidence" : 1,
            "min_confidence" : 1,
            "bin_1" : 1,
            "bin_2" : 1,
            "bin_3" : 1,
            "bin_4" : 1,
            "num_reviews": 1
        }
    }
])
```

In particular, to avoid repeating very similar lines of code different times, I opted for dict unpacking (`**bin_fields`) inside the `$group` stage (the reader can look at specific code in the `Query 9` section of the `Queries` notebook).

Results were collected in the following DataFrame:

| | num_reviews | bin_1 | bin_2 | bin_3 | bin_4 | stars | sentiment |
|---|---|---|---|---|---|---|---|
| 0 | 165723 | 10071 | 11164 | 16784 | 127873 | 5 | negative |
| 1 | 193316 | 10358 | 12123 | 17959 | 153098 | 4 | negative |
| 2 | 1259600 | 10593 | 12917 | 21023 | 1215289 | 4 | positive |
| 3 | 3065884 | 10369 | 12838 | 21359 | 3021536 | 5 | positive |
| 4 | 345863 | 9665 | 11760 | 18706 | 305931 | 3 | negative |
| 5 | 346071 | 9734 | 11129 | 17400 | 308004 | 3 | positive |
| 6 | 1013635 | 4514 | 5835 | 9713 | 993680 | 1 | negative |
| 7 | 456246 | 5240 | 6547 | 11243 | 433329 | 2 | negative |
| 8 | 87994 | 5028 | 5527 | 8228 | 69315 | 2 | positive |
| 9 | 55915 | 4130 | 4559 | 6310 | 40989 | 1 | positive |

Figure 5.14: DataFrame of distribution of confidence score for each review star

From it, two histograms were derived:

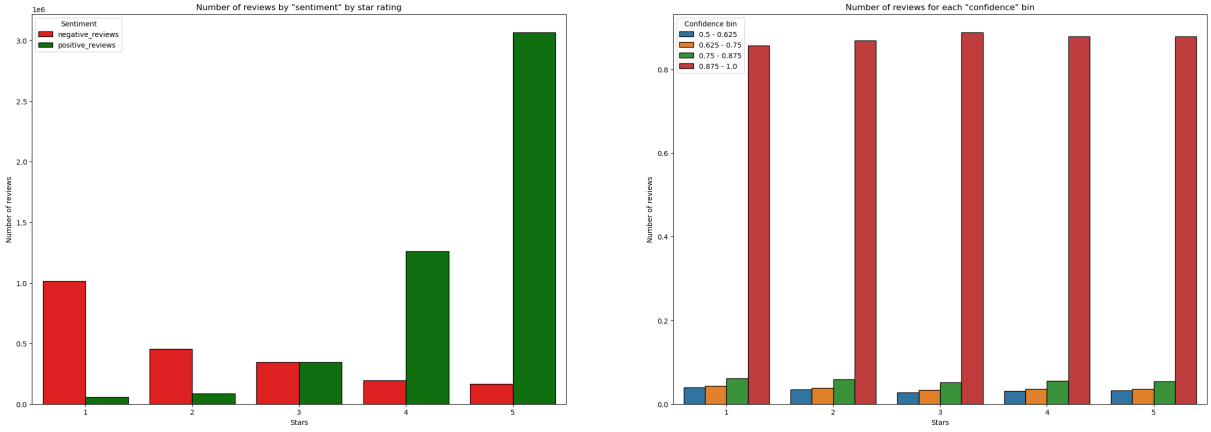


Figure 5.15: Histogram of distribution of confidence score for each review star

The analysis reveals a clear relationship between sentiment and review stars. Reviews with 4 or 5 stars are predominantly classified as positive, while those with 1 or 2 stars are mostly negative, as one could intuitively think. Three-star reviews are equally classified as positive or negative: this means that I can use 3 as threshold to discriminate outlayer reviews (see 5.11).

## 5.11. Query 11: search for reviews outlayers

The objective of this query was to search for reviews outlayers, i.e. reviews with positive (negative) sentiment, but low (high) number of stars under the assumption that positive (negative) sentiments are typically associated with stars $\geq 3.0$ (stars $\leq 2.0$). Moreover, the impact of this outlayers on the average reviews score was estimated.

To guarantee low bias in the analysis, only businesses with at least 50 reviews were considered.

```
db["businesses_merged"].aggregate([{
        "$match" : {
            "$expr" : {
                "$gte" : [{"$size" : "$reviews"}, 50]
            }
        }
    },
    {
        "$unwind" : "$reviews"
    },
    {
        "$group" : {
            "_id" : "$business_id",
            "name" : {
                "$first" : "$name"
            },
            "city" : {
                "$first" : "$city"
            },
            "measured_avg_rating" : {
                "$avg" : "$reviews.stars"
            },
            "measured_var_rating" : {
                "$stdDevPop" : "$reviews.stars"
            },
            "reviews" : {
                "$addToSet" : "$reviews"
            },
            "num_reviews" : {
                "$sum" : 1
```

```
            }
        }
    },
    {
        "$unwind" : "$reviews"
    },
    {
        "$match" : {
            "$expr" : {
                "$or" : [
                    {
                        "$and" : [{
                            "$gte" : ["$reviews.stars", 4]
                        },
                        {
                            "$eq" : ["$reviews.sentiment", "positive"]
                        }]
                    },
                    {
                        "$and" : [{
                            "$lte" : ["$reviews.stars", 2]
                        },
                        {
                            "$eq" : ["$reviews.sentiment", "negative"]
                        }]
                    }
                ]
            }
        }
    },
    {
        "$group" : {
            "_id" : "$_id",
            "name" : {
                "$first" : "$name"
            },
            "city" : {
                "$first" : "$city"
            },
```

```
70              "measured_avg_rating" : {
71                  "$first" : "$measured_avg_rating"
72              },
73              "measured_var_rating" : {
74                  "$first" : "$measured_var_rating"
75              },
76              "num_reviews" : {
77                  "$first" : "$num_reviews"
78              },
79              "filtered_avg_rating" : {
80                  "$avg" : "$reviews.stars"
81              },
82              "filtered_var_rating" : {
83                  "$stdDevPop" : "$reviews.stars"
84              },
85              "num_filtered_reviews" : {
86                  "$sum" : 1
87              }
88          }
89      },
90      {
91          "$addFields" : {
92              "abs_avg_diff" : {
93                  "$abs" : {
94                      "$subtract" : ["$filtered_avg_rating",
    "$measured_avg_rating"]
95                  }
96              },
97              "abs_var_diff" : {
98                  "$abs" : {
99                      "$subtract" : ["$filtered_var_rating",
    "$measured_var_rating"]
100                 }
101             },
102             "biased_ratio" : {
103                 "$divide" : [{"$subtract" : ["$num_reviews",
    "$num_filtered_reviews"]}, "$num_reviews"]
104             }
105         }
```

```
    },
    {
        "$sort" : {
            "biased_ratio" : -1,
            "abs_var_diff" : -1,
            "abs_avg_diff" : -1
        }
    },
    {
        "$project" : {
            "business_id" : "$_id",
            "_id" : 0,
            "name" : 1,
            "city" : 1,
            "measured_avg_rating" : 1,
            "filtered_avg_rating" : 1,
            "measured_var_rating" : 1,
            "filtered_var_rating" : 1,
            "biased_ratio" : 1
        }
    }
])
```

Results were collected in the following DataFrame:

| | name | city | measured_avg_rating | measured_var_rating | filtered_avg_rating | filtered_var_rating | biased_ratio | business_id |
|---|---|---|---|---|---|---|---|---|
| 0 | Asia Supermarket | Philadelphia | 3.362069 | 0.941354 | 3.521739 | 1.280950 | 0.603448 | Vj28PSL29SUCUzGfkZpaDg |
| 1 | Riverwalk Marketplace | New Orleans | 2.753247 | 0.913902 | 2.468750 | 1.249609 | 0.584416 | gaP8UpEO1tC-78P4wUA68A |
| 2 | Ajax Auto Glass | Philadelphia | 4.622642 | 0.758012 | 4.625000 | 0.856957 | 0.547170 | JiXca3iiIV4FewV2zTo0TA |
| 3 | Sweet Tomatoes | Creve Coeur | 3.400000 | 0.894427 | 3.478261 | 1.137104 | 0.540000 | GlrliN9NUVCy4sRhbh3ajA |
| 4 | Shops At Liberty Place | Philadelphia | 3.473684 | 0.904283 | 3.931818 | 1.008999 | 0.536842 | Az1XI-Wz_VXgswlW4OyAGQ |
| ... | ... | ... | ... | ... | ... | ... | ... | ... |
| 30176 | Truong & Company Jeweler | Santa Barbara | 5.000000 | 0.000000 | 5.000000 | 0.000000 | 0.000000 | oDnnlCOYpkMZwOv0oFSp4A |
| 30177 | Progressive Business Publications | Malvern | 1.202703 | 0.853921 | 1.202703 | 0.853921 | 0.000000 | aIgUPN4a2nzq9DK0k2gYzA |
| 30178 | Ernie's Tours | New Orleans | 4.903846 | 0.563629 | 4.903846 | 0.563629 | 0.000000 | FbGjCCc7bbJVWq3Kt5IamQ |
| 30179 | Noble Dentistry | Jenkintown | 4.600000 | 1.200000 | 4.600000 | 1.200000 | 0.000000 | L1h8vTlc_GO8riuZBtmeLw |
| 30180 | First Advantage Corporation | Saint Petersburg | 1.000000 | 0.000000 | 1.000000 | 0.000000 | 0.000000 | Snjy6RdQlkwVMLyaq5Lq1A |

Figure 5.16: DataFrame of measured vs filtered average /variance rating stars

From it, two bar plots representing the differences between measured and filtered average / variance of reviews stars were derived:

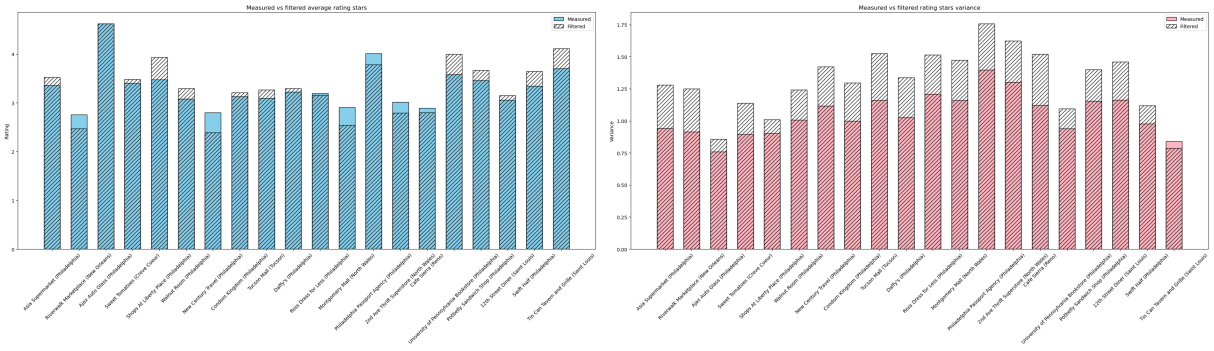


Figure 5.17: Histograms of measured vs filtered average / variance rating stars

For many businesses, the filtering process slightly increased the average rating: this mean that outliers often skew ratings downward. Some other cases display minimal impact. Variance decreased significantly in most cases after excluding outliers because this review contribute to greater inconsistency in perceived business quality.

The conclusion is that identifying and accounting for sentiment-star discrepancies can improve the reliability of aggregated ratings and better reflect actual customer experiences.

# List of Figures

# List of Tables