Data Engineering — Final Report

Daan Brugmans

ABSTRACT

This document is the final report for the Radboud University's Data Engineering course and its corresponding project. Every section was written consecutively as a weekly assignment. The report describes the design and construction of a data pipeline. The code for the implementation of the pipeline described here can be found at the following URL: https://github.com/daanbrugmans/ru-data-engineering-23-24/tree/main/code

1 INTRODUCTION

For my Data Engineering project, I built the basics of a pipeline that serves data to data scientists working on a beer recommendation system. Within the context of this project's use case, this beer recommendation system is to be built into an existing platform where end users engage in the rating of beers that they have tried. We want to use this end user data to recommend beers to end users, which we can then sell to them through our platform.

In design, the pipeline should collect from multiple sources that serve data(sets) of beer and their attributes, and serve this data to the data scientists. Examples of attributes that should be served to the data scientists are name, brewery, alcohol by volume (ABV), international bitterness unit (IBU), category/style, textual description, flavor profile, personal ratings, and a timestamp. The pipeline should serve two sets of data to the data scientists: a big general dataset containing beer data from varying publicly available sources that can (and should) be updated, and a dataset of end users of the platform that includes personal ratings of beers the end user has had before. The general dataset then serves as a pool of beers that could potentially be recommended to an end user, and the end user dataset then serves as the basis upon which recommendations are made. These datasets will be served tabular.

In practice, in order to contain the scope of this project to what is feasible and realistic, I have implemented only the parts of the pipeline that are needed to serve the dataset end users. Though not the complete pipeline described above, this long-thin approach does allow me to showcase all necessary steps for getting the pipeline running. For example, the data quality assessment and data wrangling I perform is complete, but is only applied on the dataset of end users.

For the dataset of end users, I have provided my own data of beer ratings that I have collected on Untappd. Within the context of the use case, Untappd can be seen as the platform into which our beer recommendation system will be integrated. Untappd [1] is a social medium where users rate beers they try and share their ratings with friends by registering their rating on the medium ("check-in"). I participate in Untappd and have collected a dataset of these check-ins with ratings. This dataset contains information about a beer's name, category/style, brewery, check-in location, purchasing location, flavor profile experienced by the end user, rating, and timestamp. The data will be provided in CSV format and contain almost 400 check-ins, slightly over 300 of which being unique, so while I do not expect for missing data to be a major issue, duplicates will be more prominent.

2 DATA QUALITY

Before wrangling, I will talk about the data quality of the end user data. Quantitative measures of data quality were defined using code. A class called the `DataQualityAssessor` assesses the data's quality using a few metrics. A code snippet of this class can be found in appendix A. This class is used in a Jupyter notebook that allows easy visualizations of data distributions and qualities. This notebook can be found at the following URL: https://github.com/daanbrugmans/ru-data-engineering-23-24/blob/main/code/analysis.ipynb

The Untappd data contains all of the features I need. Most of these features do not have missing data: beer name, brewery name, beer type, ABV, IBU, personal rating, and check-in timestamp have 0% data missing.

The flavor profiles experienced by the user have 1% missing data, but since that's so little, I expect to be able to handle that issue easily. The missing flavor profile data may be caused due to how a user selects flavors: although a user can select any combination of flavors from a set list, most users will often only select from the top 5 most chosen flavors for a beer, since Untappd suggests those automatically. Flavors that are not in the top 5 require more user interaction, and many users will refrain from manually searching for specific flavors. This may mean that some beers with missing flavor profile data do have certain flavors as experienced by the user, but none of which being in the top 5 most chosen flavors, and the user did not manually search for the less common flavors.

More problematic is the feature for textual descriptions: 77% of the check-ins do not have a textual comment, and of the remaining 23%, not all comments are semantically relevant to the beer itself. I expect that this feature is not one I can make good use of from the side of the personal dataset. However, the general dataset will likely contain less missing descriptions, and the descriptions will likely be more factual and relevant to the beer itself. The data scientists may use the textual descriptions from the general dataset for check-ins in the personalized dataset.

Based on box-and-whiskers-plot and barplot visualizations, there seem to be no outliers present in the data. Two features are units of measure, ABV and IBU, and they are formatted correctly: ABV as a percentage, and IBU as a float. The check-in timestamp is formatted to the standard ISO format, and floats use periods as the decimal mark. The dataset is in an unnormalized form.

Although the Untappd data provides a unique ID for each record, for our purposes, that is not the primary key of our data: that would be beer name + brewery name. We will assume that a brewery does not make multiple different beers that share the exact same name. Given this primary key, the Untappd dataset consists of 18% duplicates. This can be fixed during data wrangling: duplicates are different check-ins of the same beer. Features like beer name and brewery will be consistent across duplicates. For some features, we must apply some transformation. For example, personal ratings can

be averaged, and flavor profiles of all check-ins can be merged into one list.

3 DATA WRANGLING

The pipeline should serve two different sets of data, the general dataset and the end users dataset with ratings. We will store these datasets as parquet files. These datasets should share the same features, with the exception of the personal rating for the personal dataset. This list of features should be as follows:

- Beer Name
- Brewery Name
- Beer Category (General)
- Beer Type (Specific)
- Alcohol By Volume (ABV)
- International Bitterness Unit (IBU)
- Textual Description
- Flavor Profile

We want to represent these features in a tabular manner. Most of these features already fit this structure. However, flavor profiles pose a problem, as they are currently stored as a list of varying size for every record in the Untappd dataset. My proposed solution is to wrangle the flavors into binary features: every flavor found in any flavor profile becomes a feature. If a beer's flavor profile contains that flavor, the feature's value is 1, otherwise it is 0. If we want to use these binary features for the general dataset as well, then we will have to extract that knowledge somehow, since individually experienced flavor profiles may not reflect commonly experienced flavors of the beer. This can be solved by using the top 5 most commonly selected flavors on Untappd for the general dataset.

The Untappd dataset only has one column about beer category, but it actually contains both beer category and beer type: beer type is the full value, while beer category is the substring prior to the "-" (if it is present). We thus parse this column into two to get a category feature and a type feature.

Since duplicates in the end user dataset represent beers that an end user has rated multiple times, duplicates should be handled in a way that fits the use case. Duplicates shouldn't be ignored, since all ratings are valuable towards determining an end user's preferences, but we do not want to serve data of multiple ratings of the same beer by the same user. Our strategy is to merge multiple ratings from the same person for the same beer into one, taking the mean of all rating scores for a final rating. We could even weigh this mean rating by the timestamp, since we assume that recent check-ins are more important than older check-ins. However, for now, we simply calculate the mean rating without weighting. When creating the flavor profile features, we will also look at duplicates, since different ratings may contain different flavors experienced by an end user.

The Untappd data source is currently represented in a tabular form and is stored in a CSV file. I wrangle it into the desired representation using Python and Pandas, applying further operations on dataframes. Since all data is stored in a single CSV of manageable size, I can perform this process in bulk.

I have implemented the data wrangling as described above in a class called `DataWrangler`, whose full code can be found in appendix B. I have defined four functions that individually perform one of the wrangling steps described above:

(1) `handle_duplicates` handles duplicate records from the dataset as described above. This means that multiple ratings are merged into one. To that end, we calculate the mean rating score over all duplicate ratings, we concatenate all descriptions into one with a ";" as a delimiter, and we concatenate the variable length flavor profile lists so that we can later generate the binary flavor features with flavors experienced across multiple ratings.
(2) `parse_category_and_type_features` takes the existing beer type feature from Untappd and splits it into two: the general beer category and the specific beer type.
(3) `generate_flavor_features` generates the binary flavor features. If a user experienced a flavor for any rating of a beer, then that flavor feature is set to true, otherwise it is false.
(4) `save_as_parquet` saves the dataset as a parquet file.

I designed my data wrangling process in a way where every individual wrangling step is a method call. This way, we can easily alter the order of the data wrangling steps, and decide to include or exclude certain steps. An example of how the wrangled dataset which is served to the data scientists looks can be found in the notebook at the following URL: https://github.com/daanbrugmans/ru-data-engineering-23-24/blob/main/code/analysis.ipynb.

4 DATA SERVING

Serving of the data should be kept simple and designed to fit the stakeholders' needs. Since these stakeholders are data scientists that want to have full access to all data, and since the data is not to be accessed publicly, an API does not seem to be a good fit. Currently, there is also no business logic that should be implemented when serving the data, and the dataset size is small-scale and very manageable, so a QL API seems like an overengineered design for the current use case. Since the dataset contains data that could be traced back to specific users, we should serve it with security precautions and refrain from storing information about end users that we do not need.

A single parquet file should fit the data scientists' needs and would be preferred over CSV, but if the data scientists cannot perform their work with a parquet file, the data can be served as a CSV instead. Assuming that the business can and is willing to cover the costs, serving the data using AWS S3 would be preferred over self-hosted alternatives, for the sake of the data lifecycle and infrastructure management.

Currently, the only data served is a small bulk file of the personal end user data of a single Untappd user, which fits the serving design described above. However, in the future, there may be many users whose check-ins we want to analyze. If this number grows to be very big, we may want to partition the personal user data. I propose that this partitioning is performed by a user's region/country. This is because where a user lives has a major impact on what beers they may have access to; recommending local beers from the USA generally does not fit the needs of a German. We should then also perform partitioning by region/country for the big general dataset,

which is more likely to grow to a size too large for a single partition. By using this partitioning scheme, we can recommend beers from the regions where people live, which makes it more likely that they can act upon (purchase) the recommendations.

If the size of the collected data is approaching a level where simply loading a single parquet file is not feasible anymore, only then should we consider implementing an API. Such an API can map onto the partitions, serving a partition of the data for the region/country specified.

5 DATA LIFECYCLE

It is expected that the pipeline described above will face challenges regarding the data lifecycle. For example, we must anticipate that the general dataset is bound to grow. Breweries will continue to make new beers on a regular basis, and these should be added to the general dataset. Preferably, we try to synchronize the rate at which we update the general dataset with the pace at which breweries release new beers. This could, for example, be weekly, fortnightly, or monthly, but should be based on general brewery release schedules. By frequently adding to the general dataset, we can make sure that the end users of the beer recommendation system are able to purchase products that fit the current offerings.

Additionally, we should also frequently remove beers from the general dataset, since some beers have a limited production time, and it is useless to recommend beers to users when they cannot purchase them. These deletions can occur less frequently than the additions, since beers going out of production is a less occurring phenomenon than new beers going into production. For example, we could try a monthly or seasonal data destruction strategy. This also goes for breweries, as sometimes, entire breweries go out of business. When destroying beer data, we should also destroy data for all beers produced by a brewery that cannot offer them anymore. Aside from these reasons for data destruction, we can assume that data in the general dataset will remain relevant indefinitely. This is because most breweries have a core offering of products that they will always continue to produce.

The personal dataset of user check-ins and ratings has a higher need for being updated in a shorter time span than the general dataset. This is because this data changes more frequently: an end user's list of checked-in beers can update weekly or even (nearly) daily, and we want to use that data for their recommendations. We should attempt to also match this pace, updating end user data weekly or even daily in smaller batches. This includes updating the ratings of existing check-ins if changed and deleting check-ins if a users chooses to, since the removal of check-ins reflects the apparent preferences of the end user. We should also make sure that we can delete all check-ins of a user for GDPR compliance, but when using a pseudonymized ID for end user data, this should not be very difficult.

Furthermore, the timestamp of a user check-in should be considered. This is because we may assume that more recent check-ins weigh more heavily than old check-ins, as a user's preferred tastes change over time. We may still want to consider old check-ins nonetheless, so while we may not want to destroy old check-in data, it would be important for the data scientists to incorporate the time passed since a check-in into the recommendation system.

Given this information on the type and frequency of updates, I expect that a λ-Architecture would fit the pipeline best. We want to update the data fairly frequently, but there is no need to do it very frequently (microbatching) or continuously (streaming). I also expect that a row layout fits the data better than a columnar layout, since our updates will almost exclusively consist of data changes, and not schema changes. We only expect to add new columns when a newly added beer contains a flavor in its flavor profile that does not exist in our dataset already. We then have to add the flavor column to both datasets and set the newly added beer's value for that flavor to true and all other beers' value for that flavor to false. Unfortunately, especially as the size of our data grows, this will be an expensive operation to make. A potential solution would be to use Untappd's existing exhaustive list of flavors and adding all of these in advance. Then, we would never have to apply schema changes, unless our data need changes.

Since we are serving parquet files, we can opt for a "Parquet S3 Zoo" strategy, where we host parquet files on AWS S3 that contain the data, and we keep adding new parquet files to S3 that contain information on how the existing parquet data has been deleted or updated. We can use the DeltaLake or Iceberg framework as a tool to assist us in this in this strategy.

REFERENCES

[1] Untappd. 2024. Untappd. https://untappd.com/

A CODE: DATA_QUALITY_ASSESSOR

```python
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

class DataQualityAssessor:
    def __init__(self, dataset: pd.DataFrame) -> None:
        self.dataset = dataset

    def get_null_rates(self) -> pd.DataFrame:
        df_is_null = self.dataset.copy().isna()
        null_rates = {}

        for column_name in df_is_null.columns:
            row_count = df_is_null[column_name].size
            null_row_count = df_is_null[df_is_null[column_name] == True][column_name].size

            null_rates[column_name] = round(null_row_count / row_count, 2)

        null_rates_df = pd.DataFrame([null_rates]).T
        null_rates_df.columns = ["NULL Rate"]

        return null_rates_df

    def get_duplicate_rate(self, primary_key: list[str]) -> float:
        row_count = len(self.dataset)
        row_count_without_duplicates = len(self.dataset.drop_duplicates(subset=primary_key))

        return round(1 -  row_count_without_duplicates / row_count, 2)

    def draw_boxplots(self) -> None:
        sns.boxplot(data=pd.melt(self.dataset.select_dtypes(include="number")), x="value", y="variable")
        plt.xscale("log")

    def draw_categorical_feature(self, feature_name: str) -> None:
        sns.countplot(data=self.dataset, y=feature_name, order=self.dataset[feature_name].value_counts().index)

    def draw_continuous_feature(self, feature_name: str) -> None:
        sns.histplot(data=self.dataset, x=feature_name)
```

B CODE: DATA_WRANGLER

```python
from datasets.beer_dataset import BeerDataset

from pathlib import Path

import pandas as pd

class DataWrangler:
    def __init__(self, dataset: BeerDataset) -> None:
        self.dataset = dataset
        self.dataset.df = self.dataset.df[["beer_name", "brewery_name", "beer_type", "beer_abv", "beer_ibu", "comment", "flavor_profiles", "rating_score"]]
        self.dataset.df = self.dataset.df.rename(columns={
            "beer_name": "beer_name",
            "brewery_name": "brewery_name",
            "beer_type": "beer_type",
            "beer_abv": "abv",
            "beer_ibu": "ibu",
            "comment": "description",
            "flavor_profiles": "flavor_profile",
            "rating_score": "rating"
        })

    def handle_duplicates(self):
        grouped_df = self.dataset.df.groupby(by=["beer_name", "brewery_name", "beer_type", "abv", "ibu"], as_index=False)
        grouped_df = grouped_df.agg({
```

```
            "description": lambda x: pd.NA if pd.isna(x.all()) else ("; ".join(x.dropna().to_list()).replace("\n", ". ")
      if ("".join(x.dropna().to_list()) != "") else pd.NA),
            "rating": "mean",
            "flavor_profile": lambda x: pd.NA if pd.isna(x.all()) else ",".join(x.dropna().to_list())
        })

        self.dataset.df = grouped_df

    def generate_flavor_features(self):
        found_flavors = set()

        for _, row in self.dataset.df.iterrows():
            if pd.isna(row.flavor_profile):
                continue

            beer_flavor_profile = set(row.flavor_profile.split(","))
            found_flavors = found_flavors.union(beer_flavor_profile)

        found_flavors = sorted(found_flavors)

        for flavor in found_flavors:
            self.dataset.df[flavor] = False

        for index, (_, row) in enumerate(self.dataset.df.iterrows()):
            if pd.isna(row.flavor_profile):
                continue

            beer_flavor_profile = row.flavor_profile.split(",")

            for flavor in beer_flavor_profile:
                self.dataset.df.loc[index, flavor] = True

        self.dataset.df = self.dataset.df.drop(["flavor_profile"], axis=1)

    def parse_category_and_type_features(self):
        self.dataset.df["beer_category"] = ""
        self.dataset.df.insert(2, "beer_category", self.dataset.df.pop("beer_category"))

        for index, (_, row) in enumerate(self.dataset.df.iterrows()):
            if " - " in row.beer_type:
                category_name = row.beer_type.split(" - ")[0].strip()
                type_name = row.beer_type.split(" - ")[1].strip()

                self.dataset.df.loc[index, "beer_category"] = category_name
                self.dataset.df.loc[index, "beer_type"] = type_name + " " + category_name
            else:
                self.dataset.df.loc[index, "beer_category"] = row.beer_type

    def save_as_parquet(self, filename: str):
        if str(Path.cwd()).endswith("code"):
            cwd = Path.cwd().parents[0]
        else:
            cwd = Path.cwd()

        path_to_parquet_file = Path.joinpath(cwd, "data", "wrangled", f"{filename}.parquet")

        self.dataset.df.to_parquet(path_to_parquet_file)

        print(f"Saved dataset as parquet file to {path_to_parquet_file}")
```