Documentation for Wine.MS

GROUP IV
AMNA MUBASHAR
LEON JEVRIC
LEONARD SENSMEIER
VUK STOJKOVIC
ZORAN LISOVAC

All code for this project is available at:
https://gitlab.com/DexterJettster/wines.ms

1 IDEA

Many people like wine. German per capita wine consume rose to 20.7 liters in 2020 [1]. A lot of this wine is bought off the shelf, which has major drawbacks. On one hand information on wine labels is not standardized so a customer has to learn how to read wine labels. On the other hand a customer has to use proxy information like label design or price to judge quality.
To solve these issues a customer could use a local store and get personal recommendations from the seller. This however takes time as the customer has to visit a physical store. There is also a burden to enter a physical store as a customer might not know what to expect and how to interact with a wine seller. If the online stores of local sellers are used there are usually no personalized recommendations. Users might resolve to using reviews on the website of the seller but these are limited in amount and not matched to the users taste. Therefore, a user might use a big wine seller like Vivino[2] to buy their wine. On one hand these sellers do not provide personal service which might help to leverage existing knowledge about wine for advanced personalization. They also use opaque recommender systems so the user does not know how a wine is recommended. This could lead to unwanted behavior. There is, for example, evidence of an upselling system in the generated cookies from Vivino[3]. Other than that a large service is more likely to procure mass produced wines to meet high demands. This could lead to less diverse offerings and in turn a decline in diversity in the wine industry. Our website tackles these problems as it recommends wine sellers from Münster and their individual wines, while taking advantage of the huge amounts of data that Vivino provides. The website is designed free from upselling and provides data on the wine. It also visually shows how the wines relate to the personalized profile of the user, based on the industry standard flavor wheel.

2 DATA SOURCES AND APPROACH

To create this website, we used two types of data sources. We integrated popular local wine sellers that offer an online shop. The three local wine sellers that we decided to use are Divino, Wein-Direktimport and Jacques as they have solid databases of their products. The main problem that occurred, when getting the necessary information about wines, was the dimensionality of data. All wine sellers provide different useful information, however format and presentation of information vary considerably. In addition to that, none of the wine sellers provide standardized taste data. For this reason we use the local sellers to get information about availability and basic information about the wines.
To get a better basis for our recommendations we also integrate the Scandinavian wine service Vivino. Vivino provides a huge amount of information for a variety of different wines and calls itself the "largest marketplace of wine". A big benefit of Vivino is that they provide a JSON file for all of their wines. The stored information is standardized in terms of structure and datatype. They provide mostly precise and complete data and have less missing values than the other sources we examined and most importantly provide data on taste. Vivino has a large user base that is encouraged to leave reviews about the wines they tasted. Based on keywords from these Vivino computes the taste of the wine which is provided to the user. We use these taste profiles to recommend wines that are matching the taste of our users.
To achieve an integrated database the data from the local wine sellers and Vivino is scraped, features are extracted, cleaned and saved in a PostgreSQL database. The scraping process is described in section 3 and the database is discussed in section 4.4. After that the local wines are matched to their counterpart in the Vivino dataset. The wines that have no counterpart are re-scraped using the search function of Vivino and then matched. This process is described in section 4.3. The shared database is accessed by the application programming interface (API). The API is used to serve data from the database, create user profiles based on selected wines and compute recommendations. The functionality of the API including recommendations is described in section 6. The API is then accessed by the user interface(UI) which lets the user interact with the backend. Here the user can create a user profile and view recommendations. The user experience design(UX) and UI are described in section 5 in detail.

3 SCRAPERS

Scraping websites was the primary way of obtaining data for this project. To get the data we created a scraper for each of the four sources. The approach varies from source to source. Some data

[1] https://de.statista.com/statistik/daten/studie/150008/umfrage/weinkonsum-pro-kopf-in-deutschland-seit-2003/
[2] https://www.vivino.com/DE/de/
[3] If http://www.vivino.com/ is opened a cookie with the name eeny_meeny_personalized_upsell_module_v2 is set

Data Integration, Winter Term 2021/2022, WWU Münster
2022.

sources were easy to scrape with BeautifulSoup4[4] (bs4) in combination with the python request library, while others required a browser automation tool such as Selenium. bs4 parses the entire HTML string of a page as a network of nested objects that can be traversed both up and down. We chose bs4 as our primary library for scraping as it is on one side very fast and robust but also easy to learn and use. However, its limitation is that it converts a entire HTML string into accessible objects. For this reason another library has to be used to load the page which might be detected by the server. Besides that, as requests only load the basic html document of the page, JavaScript is not executed. If there is some data that is appended to the Document Object Model(DOM) dynamically, requests and bs4 falls short and this data cannot be scraped. To counteract this problem, a browser can be controlled. To achieve this we used Selenium in combination with Google Chrome. In these cases the website is scraped with all browser features such as JavaScript or HTML executions.

3.1 Local Vendors

Divino[5] and Wein-Direktimport[6], share the same approach. As there are no complex dynamically loaded DOM objects, it was possible to scrape all the data using requests and bs4. Due to the nature of the website, it was easy to navigate through the DOM, get the list of wines to be scraped, and fetch details for all wines via the URL redirect to the wine detail's page. Different wine categories were scraped individually and then combined into one large working data set for each vendor.

Jacques[7] was a slightly more difficult source to scrape. Some data was easily scraped using bs4, however, some additional wine data had to be scraped using Selenium. This is because not all available wines are listed upon page load. In order for all wines to be shown, the user has to scroll to the bottom for more data to be appended to the DOM.

3.2 Vivino

We scraped the data from Vivino in the following three stages. First we got the links of the wine detail page. This was either done by scraping the Vivino toplists[8] or via search terms[9] for links to wine detail pages. Afterwards, the wine detail pages were scraped. For our dataset this was done for multiple toplists and search terms. We chose toplists that were filtered by type and ordered by highest rating and highest popularity because we assume that those are the wines that users would be most familiar with. Our search terms were chosen after matching local wines and wines from Vivino. They include all names of wines that are present in the data set from local sellers, but do not match any wines from our Vivino dataset.

To access data, we first tried to use the internal API of Vivino but due to probelms with access and bot detection, we then moved to scraping. As Vivino uses JavaScript to load data dynamically we could not simply use bs4, therefore only using Selenium. While this adds computing and also network overhead, the scraper is able to access all data that a user would see when using the website. While Selenium is also less prone to bot detection we still had to add a delay after every call to avoid error responses from the website. To optimize this, using a proxy or Tor[10] were discussed during our final presentation, but not considered at the time of implementation. To extract taste data from the wine detail pages we additionally used Selenium Wire[11], which acts as a local proxy to intercept the requests to the website and relevant response data. This was necessary because this data is dynamically loaded and displayed, and cannot be extracted from the DOM.

4 DATA PROCESSING

This section describes the process, scraping results underwent in order to form a basis for our recommendation application. It includes data exploration, cleaning, matching, and fusion. We then describe how the resulting data set is loaded into and stored in our application database. The data processing from scraping results to data matching was done in a Python Jupyter notebook[12], given our prior familiarity with the language and its general use throughout this course.

4.1 Data Exploration/Extraction

The first challenges with our retrieved data became immediately apparent when examining the results of our scraper.

Vivino dataset: The data scraped from Vivino was very well-organized. Each wine had its own folder with separate files for attributes and taste profile. We needed to decide which features would be essential for our recommendations and simply iterate through the JSON files in our script. This allowed to have comprehensive wine profiles.

Local sellers' dataset: The data obtained scraped from sites of local sellers was in a comparatively disorganized form. Feature extraction was a challenging task and so was combining this data from different sources in a homogenized form. Data processing is discussed in detail in upcoming sections.

4.2 Data Cleaning

Before performing any sort of data matching, the results of our independent scrapers needed to be cleaned and standardized where possible.

4.2.1 Removing unusable items. Some items were unusable for the purposes of our applications. This group consisted of three categories:

(1) Wine sets offered by local vendors, which include more than one bottle
(2) Wines offered by local vendors for which there are none of the relevant attributes (e.g. type, country, region)
(3) Wines for which Vivino offered no taste data

The first is an obvious issue when recommending wines, as it is often impossible to identify and/or separate the individual bottles offered in the set. Though it is possible to match the local vendor

[4]https://www.crummy.com/software/BeautifulSoup/
[5]https://www.divino.de
[6]https://wein-direktimport.de
[7]https://www.jacques.de
[8]https://www.vivino.com/explore
[9]https://www.vivino.com/search/wines?q=SEARCHTERM
[10]https://www.torproject.org/
[11]https://github.com/wkeeling/Selenium-wire
[12]Found at ./data matching/data_matching.ipynb

wines with missing data on name only and to ameliorate the missing values using data fusion, this presents a likely danger of producing mismatches and resulting unfitting recommendations. Further explanation of this expectation is given in section 4.3.1. Lastly, since our recommendations rely on the taste data provided by Vivino's large user base, those wines missing this information are unable to be recommended and thus excluded from our data set.

4.2.2 Standardize attributes. Where reasonable, we aimed to modify the local vendor data to fit the standards given by the considerably larger set of wines obtained from Vivino. We began by considering attributes like wine type and country. Though one might expect these attributes to be identical for each data source already, there were a number of differences that needed to be amended. For one, the wine type categorization was being done differently by different vendors. For example, some discerned between sparkling and dessert wines, while others did not. Thus, we modified the attributes to be identical in each data source by either formatting or reclassifying each as needed.
It is of significant importance here to talk about the absence of the year attribute for our wines. While the topic of the specific influence the specific vintage (wine year) has on its flavor came up during flash talks, Vivino does not provide taste data for each indivdual vintage. Instead, all vintages of a specific wine are grouped. This tied our hands when it came to differentiation between vintages for the purposes of our application. Thus, the year attributes were kept, but not used for data matching. Some of the attributes considered for data matching however had to be extracted from the wine name strings. For example, the year attribute was often only found in the wine name and not as a separate attribute. Through heavy use of regular expressions, much of the available information in the strings was able to be separated and used as individual attributes. Though it was similarly attempted to split the winery and wine name where possible, not all sources could be modified in this way, thus forcing us to consider the full name string only.
The raw taste data only contained the number of mentions for each taste group. As the number of total mentions varied significantly between each wine, we averaged the values for each individual group by dividing over the total. This leaves us with data between zero and one, with one indicating the most prominent notes as indicated by Vivino users. Structural wine data, like fizziness or presence of tannins was given in a range from zero to five. For purposes of consistency, we scaled these to range of zero to one as well.

4.2.3 Deduplication. The removal of vintage differentiation also had an impact on duplicates in our local vendor data sets. Along with sellers carrying different sizes of bottles, different years presented an opportunities for what would be considered duplicates in our application, meaning that for all available vintages of the same wine, there would only be one set of taste data. It should be noted that while it would have been possible to reflect the connection between individual vintages and sizes and recommend the set as one wine, we thought this endeavor to be outside of the this project's scope. For this reason, we used the recordlinkage[13] Python library to detect duplicates in the local set after removing bottle size indicators to the best of our ability. Through this, 127 duplicates were removed. It should be noted that while the usage of this library was possible for the purpose of deduplication, it proved incapable for further data matching, see section 8.2.

[13] https://recordlinkage.readthedocs.io/

4.3 Data Matching & Fusion

In the following we will discuss our process to match the inventory of local vendors to the larger data set provided by Vivino. To better explain some of the choices made in this regard, we will first describe some of the specific challenges encountered during this process.

4.3.1 Challenges of the wine domain. The item domain of wines proved difficult to work with. While a restaurant usually has a distinct name across multiple sites and a song's artist rarely changes, no such properties can be expected when it comes to the naming of wines or their other attributes. Thus, data matching proved to be quite challenging. Even when perfectly matching on region, country and wine type, the names alone can prove enough to hinder any method of comparison, specifically string distance measures. While we encountered numerous differences in naming schemata between vendors and wineries, two of the more prevalent issues are described in the following.

(1) Pietro Dal Cero Amarone della Valpolicella Cà dei Frati
(2) Cà dei Frati Amarone della Valpolicella DOC Pietro dal Cero

Consider the wine names shown in the list above. The entries represent the names given to a single wine by two different vendors. To the human eye, these two immediately appear as a match, since aside from the quality indicator DOC and the order of the individual tokens they are identical. However, these two strings have a Levenshtein distance of thirty, making it improbable to be considered a match by a record linkage library. In contrast, consider the following pair of names, in which the first entry has been rearranged manually to mirror the second:

(1) Cà dei Frati Amarone della Valpolicella Pietro Dal Cero
(2) Cà dei Frati Amarone della Valpolicella DOC Pietro dal Cero

While even more obvious now to the eye, this pair would also be easily be identified as a match by a computer since the Levenshtein distance has shrunk to only five. The issue, however, is that such reordering of wine names is impossible to perform automatically, given that there is seldom a consistent schema to be found even for single vendors. This problem heavily limited our options when it came to the use of record linking packages, with some notable eliminated examples described further in section 8.2.

Seller A	Seller B
Rhône	Châteauneuf-du-Pape
Mosel	Bernkastel
Languedoc	Pays d'Oc
Loire	Crémant de Loire

Table 1. Attribute grain difference in wine regions, selected examples.

Another issue present in the data is the different grains our data sources use when describing properties. A shining example of this is the region attribute. Very often different vendors disagreed on how

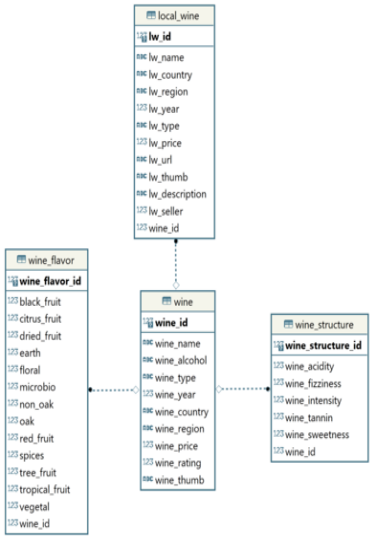
to describe a wine's regional origins with one referring to the greater region, while others indicated specific villages or winery collectives. Additionally, regions sometimes include official quality indicators for regional products. Some examples can be seen in table 1, with many more present in the data. Nevertheless, the region attribute was included for matching as it still most often carried valuable information and appeared usable by fuzzy matching methods.

4.3.2 Matching. Though not a perfect solution (see section 7), we ultimately found the python library fuzzymatcher[14]. The major advantage it has compared to the other considered options is the use of SQLite3's Full Text Search (FTS) to find candidate pairs. While we do not claim to understand the search method in its entirety, what we can discern is that FTS works independent of token order, and does so in a very fast manner. In combination with its probabilistic approach, this helps mitigate the issues discussed previously. Thus, we used the package to match our set of locally available wines to the set obtained from Vivino, with join attributes being the full name string, country of origin, region and wine type. Though mostly available, other attributes like price and alcohol percentage proved to be either irrelevant or with negative impact in testing.

4.3.3 Fusion. With the matching results available, we are able to fulfill our goal of extending the wines available from local sellers with data about their taste and structure. However, instead of simply merging these data points into one large table, we decided to deliberately keep the data about local wines separate from our Vivino data. Consequently, matches were identified via foreign key relations in the database. This was done for multiple reasons, including so that information like the specific name, price, etc. are available for each vendor separately in case a wine is available from multiple sources, and easier searching of the distinct parts (local vs. Vivino) of our data. However, some concerns regarding a more efficient structure for the database can be found in section 7. Also, though the Vivino dataset offers an immense amount of information on most wines (e.g. region-typical grape composition), we deemed most of these attributes irrelevant for the purposes of this project.

4.4 Database

We store all our data in a PostgreSQL database[15]. There are multiple reasons why we chose a SQL database. First the strict structural constraints provide security and limits the amount of error handling necessary in the API. Secondly our recommender system takes advantage of SQL and does not require the additional speed that NoSQL offers over security. We chose PostgreSQL because it is a powerful, reliable and industry standard database management system. The ER model of our system went through multiple revisions. Initially, we started with a relatively complex model with multiple m:n relationships and intermediate tables for joins in the schema. As the project developed, it seemed more feasible to keep the ER model simple as it facilitated the access for API and User Interface. The final ERD model consists of four tables.



4.4.1 wine. : This is the central table of our schema with unique keys for each wine and contains our set of wines obtained from Vivino. It is related to all other tables linking the structure and flavor attributes to individual wine items. The table contains all the substantial information about each wine differentiating them from other similar items.

4.4.2 wine_flavor. : Flavor data of each wine extracted from Vivino contains 13 main flavor groups based on which wine taste profiles are built and the count of this group represents the strength of each flavor in the wine. "wine_id" from wine table is used to connect the flavor profile to each wine.

4.4.3 wine_structure. : This table represents additional features to be used in generating recommendations like intensity, sweetness etc. of each wine in wine table and are linked through foreign keys.[16]

4.4.4 local_wine. : All the local wine data obtained from data sources (Wein-Direktimport, Jacques, Divino) is represented in this table. "lw_seller" represents the belonging of entry to one of the sellers. The data collected from these local sources suffered from missing values which were sometimes omitted on purpose. Thus, each local wine is mapped to corresponding entry in wine table to have comprehensive profile for each wine. These datasets were disoriented (without any matchable name or identifying attribute) and it was a challenge to map them as discussed already in section 4.3.1.

4.4.5 Integration with Python. : The SQLAlchemy toolkit[17] in Python is used to inject data into PostgreSQL.[18] This toolkit allows direct conversion of DataFrames or CSV files to tables in PostgreSQL. All the extracted features from JSON files are added to lists in Python which are then added to DataFrames. DataFrames represents a table and lists are individual columns.

[14] https://fuzzymatcher.readthedocs.io/
[15] https://www.postgresql.org/
[16] wine, wine_flavor and wine_structure could be combined as one table with many columns. But for clear distinction of features used in recommendation algorithm, data was split into separate tables.
[17] https://www.sqlalchemy.org/
[18] Found at ./db/ wine_sql_schema.py

5 USER EXPERIENCE & USER INTERFACE

To design our UI we first designed the user flow and translated all steps into different views. These were then designed and modelled in Figma[19] and after that implemented.
On the website there are four different views. The user starts in the Wine search where one or multiple wines can be selected. We decided to start with a wine search as we expect the user to know at least one wine that can be used as a starting point for the user profile. After that follows the profile customization. Here the user profile we computed from the wines can be customized. Users who know which wine they prefer can use this to fine tune their profile while users who do not know a lot about wine do not need to enter anything. We used a radar chart to illustrate taste as it visually represents taste in a way that places similar tastes close to each other. This visualization is based on typical wine flavor wheels like from Winefolly[20]. After that follows the recommendation view which shows twelve single wines from the local wine sellers and also the three local wine sellers ranked in terms of taste overlap. When clicking on the single wines the wine detail page is opened. This contains the description, availability and taste of the wine. We use a radar chart to visually show the overlap between the user profile and the taste of the wine. There is also a table with information that is used to calculate similarity of different wines like acidity or fizziness. We decided to exclude this information from the profile customization as a user would not have any point of reference for a value like acidity 0.7.
Our UI is implemented in React.js and served by webpack. We chose React.js because we can leverage the state management system to create a more predictable UI that updates automatically if data is changed. It also provides great debugging capabilities and huge open source community. As React libraries we used material.ui[21] for basic components. Material.ui provides a modern look based on the material design guidelines by Google[22]. This allows the user to intuitively use our website and us to save time in development. Integration with the backend is done by swagger-js[23]. This package allows us to read the swagger.json file provided by the API. Swagger-js translates API endpoints to JavaScript functions which can be called directly. This cleans the source code and speeds up integration in development. We used Chart.js[24] to display the radar charts as it provides an intuitive design. The map to display recommended wine sellers is integrated with Google Maps[25] through gmaps[26]. This is because most of our users will have used Google Maps in the past as it is one of the largest maps services. We also use react-cookie[27] to store the profile persistently until the user decides to reset his recommendations.

[19] https://www.figma.com/
[20] https://winefolly.com/tips/wine-aroma-wheel-100-flavors/
[21] https://mui.com/
[22] https://material.io/
[23] https://github.com/swagger-API/swagger-js
[24] https://www.chartjs.org/
[25] https://www.google.de/maps
[26] https://github.com/MicheleBertoli/react-gmaps
[27] https://github.com/reactivestack/cookies

6 API & RECOMMENDER

In order to further separate the functionalities of the recommender engine, an API on top of Django was built. Given the course's focus on Python and our previous experience with the language, Django REST framework was chosen, which supports implementations in Python. The API offers 4 GET endpoints:

/search_wines/criteria:
This endpoint has one parameter, criteria, which is of type string. The criteria is then used when searching through the Wine table which has the data from all the available wines from the large international re-seller, Vivino.

/details/id:
This endpoint also expects one parameter, id, which represents the id of the local wine user wishes to obtain more detailed information about. Id is of data type integer and must be greater than zero. The details about the local wine are obtained from the table local_wine.

/profile/wine_ids:
This endpoint expects a list of of wine IDs. Data from the respective wines are queried from the database, and the taste profile is constructed based on taste data for those wines. Taste data is queried for each of the wines from the wine_flavor and wine_structure tables. The taste data for each taste (there are thirteen taste notes taken into account) are averaged across multiple wines and the constructed profile is sent back to the user.

/recommendations/profile:
This endpoint expects a tuned wine taste profile from the user as an input. Profile is received in the same form as it was returned from the /profile/wine_ids endpoint. Difference is that user is given the chance to further refine their taste profile in the implemented UI (see UI section). Refined taste data is then used by the recommender engine logic to produce a list of local wines, which are then returned to the user as a recommendation result.

All the endpoints are consumed via HTTP GET request. The reason for this is that the communication between the client and the API is unidirectional in the respect that the client cannot alter the state of the data and database within our project scope. If detailed user profiles outside of cookie-based session management would need to be kept, adequate endpoints would need to be implemented. There are example files for all four API Endpoints in the folder ./api/examples.

6.0.1 Swagger. Swagger is an Interface Description Language for describing RESTful APIs expressed using JSON. Using the this package, three endpoints were generated. Hitting the baseurl/swagger endpoint, swagger UI will be generated based on the underlying API code, and the API can be examined with the swagger UI easily then. Otherwise, swagger information schema can also be exported to a .json or .yaml file by hitting the endpoints /swagger.json or /swagger.yaml for JSON or YAML configuration respectively.

6.1 Recommendations

For our recommendations, we thought to make use of the invaluable repository of user-contributed taste data provided by Vivino. The averaged signal values for each flavor group are used to find those wines that match the given user preferences as closely as possible. While in its nature this is a content-based recommendation system, we like to think of it as a hybrid approach given that it uses user-provided data. Additionally, it borrows inspiration from collaborative aspects as it could be considered to treat wines like user ratings which the user's input is matched to.

To be more specific, we have previously seen that taste and structure data is stored as values between zero and one. When a user now selects one or multiple wines as an input to get similar recommendations for, the API queries the database for their data and averages their values to generate the user's profile, which the user can modify on the front-end. Combined with the user's selected categorical filters, the resulting profile is then passed back to the API.

Now, all locally available wines matching the filter criteria are scored on their similarity to the given profile. This is done using the following formula:

$$\text{score} = (p_s * \text{cs}(s_{\text{wine}}, s_{\text{profile}}) + p_t * \text{cs}(t_{\text{wine}}, t_{\text{profile}})) + (p_r * r_{\text{avg}})$$

Here, s, t and r represent the structure values vector, taste values vector and average user rating for the wine, respectively, while cs stands for the cosine similarity function. In other words, we add up the cosine similarities between the wine's and the user's structure and taste vectors respectively, and then add the resulting number by the wine's average rating on a zero to one scale. This is done to incorporate the average user ratings provided by Vivino, which are a valuable resource when it comes to determining a wine's quality. p_s, p_t and p_r describe parameter values which are used to give weights to the individual parts of the formula. While these could be used to further fine-tune recommendations, an initial configuration of 50%, 100% and 50% appears to perform enough for the purposes of our minimum viable product. We deliberately chose to give less weight to the structural values and average rating because while they represent influential characteristics of their respective wines, the focus of our recommendations as well as most of the user interaction focuses on their taste descriptors. Of course, the parameters could be modified to lay greater focus specific types of recommendations. For example, one could recommend unpopular, but similarly tasting wines by simply giving the ratings parameter a negative value.
Finally, the resulting scores are sorted, and a configurable number of best scoring wines are returned to the user with their respective information added, and the scores for each local vendor within this set are aggregated to return a corresponding seller ranking to the user.

7 LIMITATIONS

At the moment our project has some limitations which we present below. These can be categorized into limitations that stem from the context of a school projects and limitations that evolved from difficulties of the domain of data about wine.

7.1 Concept

At the moment we only recommend wine from three local sellers. To provide a accurate picture of the wine sellers of Münster we would have to integrate more. Especially the recommendations of wine sellers would benefit if not only the same three sellers would be shown over and over again. We decided to not include more sellers because we see this project as a prototype and we think that we have shown that we can integrate local wine sellers. We also wanted to include collaborative filtering into our recommendations. In the process of cleaning the data and developing the matching system we realized that we would not have time to develop or test this and therefore decided against it. Another limitation is hosting. We designed our system with hosting in mind. The UI, API and database are ready to be transformed into docker containers. However we did not have time in the end to do it.

7.2 Matching results

Fuzzy matching introduces a number of mismatches. As a specific example, given items with large amounts of overlaps in their name strings but slight differences, the matcher might confuse wines of the same winery and type if their names only differ by few characters, such as the following examples:

(1) Weingut Emmerich Knoll Grüner Veltliner Smaragd „Ried Schütt“
(2) Weingut Emmerich Knoll Grüner Veltliner Smaragd „Ried Loibenberg“
(3) Weingut Emmerich Knoll Riesling Smaragd „Ried Schütt“
(4) Weingut Emmerich Knoll Riesling Smaragd „Ried Kellerberg“

These show how small differences in naming between Vivino and the local vendor can lead to mismatches between these. In fact, the first and second entry were both matched to the same wine, while the third and fourth were correctly differentiated.

Furthermore, not all locally available wines have gotten good matches despite re-scraping Vivino for their counterparts. Manual confirmation of each match would have taken an unreasonable amount of time, thus allowing these to exist within our data. However, as seen in the example above, we expect most wines to be extended with flavor data of at least a very similar wine, which for the purposes of our system as a prototype was deemed acceptable. Additionally, given the size of our data set, we decided against the inclusion of a specific threshold of matching score, much for the same reasons. Of course, this would need to avoided in a production system.

7.3 Data set size

As mentioned previously, our data set is limited in size. The difficulties in scraping Vivino's dynamic website made it impossible to access their entire collection of wine data. Despite this, we were able to catalogue more than ten thousand wines. While this appears to be a reasonable amount, the sheer amount of available wines in the world is of course much larger, allowing only a small slice to be used as a base for recommendations. On the local side, our we managed to achieve a data set size of approximately twelve hundred wines over three vendors. Given the amount of effort required to

scrape, clean and adapt data for further vendors, we declined to include more of them in our project, though it might have given more varied recommendations.

7.4 Database model

Our data model as described in section 4.4 is not optimal. In a production environment it might be better not to separate the wines by their data source origin (local vs. Vivino), but rather just keep one large table of wines with another relating them to their availability at certain vendor. However, in the interest of ease of access through the Django Object-Relational Mapper (ORM) as well as time constraints, we decided to keep the model as is. Similarly, the given structure makes access to structure and taste data easier by using separate tables, as iterating over their attributes appears much simpler this way. Though possibly less efficient, our comparatively small data set size allowed this trade-off not to impact application performance significantly enough to be noticed in our testing and demonstration.

8 FURTHER APPROACHES & EXPERIMENTATION

In this section we describe some of the approaches that were tried when attempting to solve the challenges of our project. These did not end up in the final implementation that was demonstrated during our presentation, but nevertheless appear relevant to discuss in this documentation.

8.1 Data Imputation

To amend a number of wines that were eliminated due to missing values, the following attempts were made at data imputation:

8.1.1 Predicting missing wine type attributes. Initially, it appeared that some of the Vivino scraping data was missing their wine type attribute (e.g. red, white etc.). To remedy this, a model was trained to predict wine type based on taste profiles. The approach was based on converting the taste profiles into numerical data by using one-hot encoding mapping to a list of all possible flavors in our data. We then trained our model using the linear regression function from the Scikit-learn Python package[28], fit the model and predicted the wine types of the missing values. By splitting the data into training data and validation data, results showed that the model was 86.3% correct.[29]. However, we later discovered another wine type indicator in the rather complex Vivino JSON structure that was originally missed. This indicator was reliably present for all items. Consequently, this approach was excluded from the final project.

8.1.2 Predicting missing taste data. Another case of missing data was given by items which had no taste and/or structural data. As described earlier in section 4.3.1, this presents a problem when considering our goal of using these exact values to calculate recommendations. Thus, we attempted to find a model that could make use of wine reviews and their included notes on flavors to estimate the result Vivino's users would have generated. For this, we employed scikit-learn's CountVectorizer[30], which generates matrices of token counts. The vectorizer was outfitted with a vocabulary that included all tasting notes present in the Vivino data set to ensure compatibility.
In addition, the natural language processing (NLP) library spaCy[31] was used to lemmatize the words present in our set of reviews taken from the vendor descriptions. This was done in an effort to overcome issues such as We would have preferred stemming the words, but the required tools to stem German language appeared inadequate or unavailable.
Ultimately, when testing this system it quickly became apparent that the effort required to overcome issues with natural language processing in this context far outreached the scope of this course and project. One significant issue would have been to translate all taste descriptors into multiple languages to achieve coverage for both German and English reviews found on vendor websites and review aggregators. Furthermore, early testing results indicated it would have likely required extending and categorizing the vocabulary with synonyms for individual descriptors in multiple languages, complex custom stemming procedures and an entirely custom vectorizer to make use of any such vocabulary. As a result, this attempt was left out of the final project. Nevertheless, remnants of early testing are included in the code as a Jupyter notebook. [32]

8.2 Data Matching

Before settling on the data matching process as described in section 4.3, numerous attempts using other methods were made. Since they are of interest to illustrate the challenges at hand within our project, we will briefly describe them in the following.

8.2.1 recordlinkage. As presented in the lecture, our first attempts to match local vendor wines with their Vivino entries were made using the recordlinkage package[33]. Using Indexing on attributes like country and region could speed up the calculations despite the large number of possible candidate pairs. However, data exploration had previously shown that regions are not a standardized attribute and vary between vendors, even for identical wines. Additionally, despite requesting the German version of its website, Vivino provided the country and region attributes in English only for some of the items with no discernable pattern. Despite partial manual corrections of this issue, the recordlinkage library could not reasonably benefit from its attribute blocking capabilities and thus had to deal with large runtimes. Additionally, even when blocking worked, the present challenges with the available string distance measure often failed to produce matches that otherwise appear obvious to the human eye. Considering lower matching thresholds did not fix this issue, but increased the number of false positive matches instead.

8.2.2 rapidfuzz. After failed attempts at using the recordlinkage library, we investigated multiple other packages to find an alternative. The rapidfuzz packagerapidfuzz appeared as a promising option, mostly due to its speed enabling the possibility of brute-forcing match results in a reasonable timeframe. While recordlinkage was unable to work with issues in our data like the inconsistent naming of regions, we thought it possible that a brute-force fuzzy-matching

[28] https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html
[29] Found at ./Data Imputation/wine_prediction.py
[30] https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.CountVectorizer.html
[31] https://spacy.io/
[32] Found at ./Data Imputation/winerec.ipynb
[33] https://recordlinkage.readthedocs.io/en/latest/

approach might be able to deliver usable results. Though it processes the entire matrix of candidate pairs, rapidfuzz is significantly faster than other fuzzy matching libraries. Additionally, it is possible to enable multi-threading, which more than halved the computation times in our testing. Ultimately, it fell prey to the same issues with string edit distances that recordlinkage did, resulting in many missing matches or mismatches despite its brute-force approach.

8.2.3 TF-IDF. The TF-IDF string comparison method provided by the python package py_stringsimjoin[34] did not yield any usable results. It generated matches when the similarity score was set to a threshold of at most 60%, but as the low threshold suggests, closer inspection of the results showed very few correct matches. Wine names' individual tokens matched on certain tokens, like wineries, or regions, but the overall entity was not the same. Conversely, when the similarity score threshold was set to values of 70% or higher, there were no potential matches produced.

9 CONTRIBUTION STATEMENTS

This section contains self-written contribution statements to reflect on each individual's contributions to the overall project.

Amna - I participated in discussions to formulate a joint agreement about scope of project. In the initial phase, I assisted in designing UI prototype screens to visualize what the project would ultimately look like. In the development phase, I was responsible for the data extraction, ERM and database formulation. I did basic work on feature selection from scraped data and worked with machine learning techniques like regression imputation to predict some missing values. I constantly adjusted ERM as per needs of the recommender/API until it was finalized and wrote scripts for insertion of cleaned data in PostgreSQL. Besides this, throughout the project, I actively worked on content description and presentations.

Leon - In early ideation meetings I pitched and campaigned for the idea of focusing on recommendations based on a flavor wheel. Later, I worked to get everyone on the same page regarding the required data and functions to implement our vision, as well as shape the process model of our application. During project implementation work, I was originally assigned to focus on the recommendation script. With the nature of our data making the recommendations relatively simple to achieve, I was also able to author our data matching notebook, which contains our data processing sans DB-related tasks. During testing I was also able to contribute by working closely with the front-end side to integrate the components, as well as debug and streamline the API code itself. In addition, I believe to have been a helpful contact for the other team members when it came to questions regarding the content and structure of our data set, as well as how our system components might interface with it.

Leonard - During this project I worked on multiple parts. In the beginning I developed the core concept of our project which included developing a general idea and also distinguish our project from other solutions to buy wine. I then developed the user flow and helped designing the individual views of the UI. After that, I developed the UI which included choosing technology, minor redesign of the views, implementing and testing. As part of that I worked on the outgoing data structure of the API and finally integrated the API in the UI. Besides that I also developed the Vivino scraper and did all the scraping for that on my computer.

Vuk - I believe that my effort invested in scraping the data from local wine vendors improved our data integration methods, and affected the entire system with its working components. I strived to give useful advice, and tried to help other colleagues in processing the data in generally – through extraction, finding the missing values, cleaning and to importing it to our database. I think that my supportive philosophy really reached some of my colleagues to, with more confidence, commitment and security fulfill their tasks. Finally, I offered an help during the search of best possible solution for our recommendation engine, and I hope that some of ideas, if was not helpful at least left inspirational trace.

Zoran - In the beginning of the project, I helped shape some ideas that eventually led to formulating the topic of our project. I supported the idea that we should have some kanban-like board to track the tasks all of us had, and the progress of those tasks. When we decided on the data sources, I wrote scrapers for 2 of the data sources using python and bs4, as I already had experience with that framework. When the data sources were scraped, I tried few duplicate detection methods using the edit distance and TF-IDF methods, but as they have proved unreliable we would later on switch to fuzzymatcher. I proposed to use scalable architecture and to separate the recommender engine from its presentation. This gave birth to our independent rest API service. Using Django Rest framework I developed the API that connected to the PostgreSQL Database and transformed the data to the UI readable data model.

[34] https://github.com/anhaidgroup/py_stringsimjoin