

ACIT5900
MASTER THESIS

in

**Applied Computer and Information
Technology (ACIT)**

May 2021

Applied Artificial Intelligence

**A Comparative Study of Data Transformation
Tools: An Investigation of Functionalities Supported
in Common Tools and Case Study of Declarative and
Procedural Data Manipulation Languages**

Tine-Lovise Storvoll

Department of Computer Science

Faculty of Technology, Art and Design

OSLOMET

Acknowledgements

I would like to thank my supervisors, Ahmet Soylu and Francisco Martin-Recuerda, for their help in getting me through this master's thesis. Thank you for guiding and encouraging me throughout the semester.

I would also like to thank Leif-Harald Karlsen for providing the use case with datasets for my case study and giving me advice on solving the use case. I appreciate you taking the time.

Francisco Martin-Recuerda was supported by the project Skytrack funded by The Research Council of Norway under the grant No.309714.

Tine-Lovise Storvoll

Tine-Lovise Storvoll

Oslo, May 16, 2022

Abstract

Today, organizations are collecting and storing huge amounts of data that could potentially be very valuable. Finding trends and patterns in historic data can allow businesses to make more informed decision. Data scientists are therefore working to extract meaning from the massive amount of data. However, 80% of the time in data science projects is spent preparing the data for analysis. Selecting an efficient tool for the job can contribute to reducing the time spent on data transformation. Thus, this thesis will provide some insights into existing tools and their performance.

A selection of common tools is made in Chapter 3. The tools are reviewed with regards to a framework to identify the support of common data preparation tasks and an evaluation of the tools are given at the end of the chapter. In Chapter 4, one declarative and one procedural Data Manipulation Language (DML) are selected from the common data transformation tools. Python pandas, a procedural language, and SQL, a declarative language, are evaluated and compared in a case study. The case study delves deeper into the tools through a use case and the comparative analysis at the end will provide some insights into the differences in the two DMLs. Thus, the first contribution of this thesis is a review of the support of common data preparation tasks provided by a selection of some prevalent data transformation tools. The second contribution is an analysis of the differences in a declarative vs procedural approach to data manipulation through a case study comparing two popular DMLs.

The findings of the review of tools in Chapter 3, revealed that the most prevalent data transformation tools support the majority of the common data preparation tasks. This review gives some general insight into which tasks are supported, which tasks needs more effort to perform, and which are not supported at all. The review is exclusively based on information found in technical documentation of the tools, and no further experimentation is done to investigate the support. The case study in Chapter 4 revealed that the procedural DML, Python pandas, is better suited for data manipulation as it is less time-consuming and provides higher flexibility and usability. Python pandas is also considered to have high readability and expressiveness, although SQL seems to beat pandas in these areas.

Table of Contents

| | |
|---|----|
| Acknowledgements | 1 |
| Abstract | 2 |
| List of Figures..... | 6 |
| List of Tables..... | 8 |
| 1 Introduction | 9 |
| 1.1 Problem Description | 10 |
| 1.2 Research Questions | 10 |
| 1.3 Research Design..... | 11 |
| 1.3.1 Framework | 11 |
| 1.3.2 Use Case | 13 |
| 1.4 Thesis Structure | 13 |
| 2 Background and Related Works..... | 14 |
| 2.1 The Raw Data..... | 14 |
| 2.2 Data Transformation | 15 |
| 2.3 Data Transformation Tools | 17 |
| 2.3.1 Declarative vs. Procedural..... | 18 |
| 2.3.2 Review of Data Transformation Tools..... | 18 |
| 2.4 Related Works | 21 |
| 2.5 Ethical considerations..... | 25 |
| 3 Review of Data Transformation Tools | 26 |
| 3.1 Functionalities of the different tools..... | 28 |
| 3.1.1 Python pandas..... | 28 |
| 3.1.2 SQL..... | 28 |
| 3.1.3 R..... | 29 |

| | | |
|-------|---|-----|
| 3.1.4 | RapidMiner..... | 29 |
| 3.1.5 | KNIME..... | 30 |
| 3.1.6 | Trifacta | 30 |
| 3.1.7 | OpenRefine..... | 30 |
| 3.1.8 | Talend Data Preparation | 30 |
| 3.1.9 | Tableau Prep | 30 |
| 3.2 | Results of the review | 31 |
| 4 | Case Study: Comparing Declarative vs. Procedural Data Transformation Tools... | 32 |
| 4.1 | Python pandas library..... | 32 |
| 4.1.1 | Importing pandas and loading the data..... | 32 |
| 4.1.2 | Agricultural Properties | 33 |
| 4.1.3 | Meat Deliveries | 36 |
| 4.1.4 | Dairy Deliveries | 44 |
| 4.1.5 | Egg Deliveries | 47 |
| 4.1.6 | Grain Deliveries | 52 |
| 4.1.7 | Nutrients Table..... | 57 |
| 4.1.8 | Data Mapping and Integration..... | 65 |
| 4.1.9 | Summary of the case study of Python pandas..... | 73 |
| 4.2 | MySQL..... | 74 |
| 4.2.1 | Creating a database and loading the data | 74 |
| 4.2.2 | Data Discovery..... | 76 |
| 4.2.3 | Data Preparation | 76 |
| 4.2.4 | Data Mapping and Integration..... | 98 |
| 4.2.5 | Summary of the case study of MySQL | 102 |
| 4.3 | Results..... | 103 |
| 4.3.1 | Time-consumption | 104 |

| | | |
|-------|---|-----|
| 4.3.2 | Flexibility | 105 |
| 4.3.3 | Expressiveness..... | 105 |
| 4.3.4 | Usability..... | 106 |
| 4.3.5 | Readability..... | 106 |
| 5 | Conclusion..... | 108 |
| 5.1 | Discussion | 108 |
| 5.1.1 | What is the support for common data preparation tasks provided by some of the most prevalent data transformation tools? | 108 |
| 5.1.2 | How does declarative vs. procedural DMLs differ in terms of time- consumption, flexibility, expressiveness, usability, and readability?..... | 109 |
| 5.2 | Future work | 113 |
| 6 | References | 114 |
| | List of Appendices | 117 |

List of Figures

| | |
|--|----|
| Figure 2.1 The steps of Data Transformation | 16 |
| Figure 2.2 Illustration of long and wide data formats (Zach, 2021)..... | 17 |
| Figure 4.1 A sample of the agricultural properties data | 34 |
| Figure 4.2 Output of the current subset of the agricultural properties data | 35 |
| Figure 4.3 Output of the prepared agricultural properties data | 36 |
| Figure 4.4 Comparing the size of the initial and final agricultural properties dataset | 36 |
| Figure 4.5 Meat deliveries data after unpivoting..... | 39 |
| Figure 4.6 Meat deliveries data after removing rows where amount is 0 | 39 |
| Figure 4.7 Lists of parts of the "type" column of the meat deliveries data | 40 |
| Figure 4.8 A sample of the current Meat deliveries data. | 41 |
| Figure 4.9 Dictionary of simplified meat types | 42 |
| Figure 4.10 A sample of the final Meat deliveries data. | 44 |
| Figure 4.11 Dairy deliveries data after dropping irrelevant columns | 45 |
| Figure 4.12 Dairy deliveries data after unpivoting..... | 46 |
| Figure 4.13 Dairy deliveries data after splitting the "type" column | 47 |
| Figure 4.14 The final Dairy deliveries data..... | 47 |
| Figure 4.15 Egg deliveries data after dropping irrelevant rows..... | 49 |
| Figure 4.16 Null values in the egg deliveries dataset..... | 49 |
| Figure 4.17 Egg deliveries data after adding the "type" column..... | 50 |
| Figure 4.18 Egg deliveries data after renaming the "egg_kg" column to "amount" ... | 51 |
| Figure 4.19 Egg deliveries data after rearranging the columns | 51 |
| Figure 4.20 Final Egg deliveries data after preparation | 52 |
| Figure 4.21 List of column labels of the grain deliveries data | 53 |
| Figure 4.22 Relevant columns of the grain deliveries data..... | 54 |
| Figure 4.23 Grain deliveries data after removing irrelevant columns | 55 |
| Figure 4.24 Grain deliveries data after unpivoting | 55 |
| Figure 4.25 Grain deliveries data after splitting the "type" column | 56 |
| Figure 4.26 A sample of the current Grain deliveries data. | 57 |
| Figure 4.27 A small sample of the initial nutrient table data..... | 57 |

| | |
|---|----|
| Figure 4.28 A sample of the nutrients table data with the new column headers | 58 |
| Figure 4.29 Nutrients table data after removing redundant rows | 60 |
| Figure 4.30 Nutrients table data after extracting only relevant columns | 60 |
| Figure 4.31 Lists of substrings of the type of food from the Nutrients Table data. | 61 |
| Figure 4.32 Nutrients table data after splitting the "Matvare" column. | 62 |
| Figure 4.33 Nutrients table data after renaming the columns to "type" and "kcal" ... | 62 |
| Figure 4.34 Comparing the size of the dataframes after removing missing values. ... | 63 |
| Figure 4.35 Checking the size of the Nutrients Table data after grouping the data. .. | 64 |
| Figure 4.36 Nutrients table data after performing all data preparation tasks | 64 |
| Figure 4.37 First five rows of concatenated food production data | 66 |
| Figure 4.38 Last five rows of the food production data..... | 66 |
| Figure 4.39 List of unique food types in the food production data | 67 |
| Figure 4.40 Finding the length of the list of unique food types..... | 68 |
| Figure 4.41 Dictionary containing the fuzzy matched food types | 69 |
| Figure 4.42 Nutrients table data after matching the food types..... | 70 |
| Figure 4.43 Nutrients table data after mapping the food types..... | 70 |
| Figure 4.44 Final data with all datasets integrated..... | 72 |
| Figure 4.45 Sample of the data after adding a column "produced_kcal"..... | 72 |
| Figure 4.46 Final data answering the use case question | 73 |
| Figure 4.47 An example of the import of CSV files. | 75 |
| Figure 4.48 Sample of the Agricultural Properties Data | 78 |
| Figure 4.49 A sample of the Agricultural Properties data..... | 80 |
| Figure 4.50 A sample of the current Meat Deliveries data..... | 84 |
| Figure 4.51 A sample of the Dairy Deliveries data before making any changes..... | 85 |
| Figure 4.52 A sample of the Dairy Deliveries data with a "type" column..... | 86 |
| Figure 4.53 A sample of the Dairy Deliveries table after adding the type column..... | 86 |
| Figure 4.54 A sample of the Dairy Deliveries data after setting the type..... | 87 |
| Figure 4.55 A sample of the Dairy Deliveries data with an "amount" column | 88 |
| Figure 4.56 A sample of the Dairy Deliveries data. | 89 |
| Figure 4.57 A sample of the Dairy Deliveries data after grouping and aggregating.... | 90 |
| Figure 4.58 A sample of the Egg Deliveries data before making any changes | 91 |

| | |
|--|-----|
| Figure 4.59 A sample of the Egg Delivery data with a “type” column | 92 |
| Figure 4.60 A sample of the Egg Delivery data after grouping and aggregating | 93 |
| Figure 4.61 A sample of the Grain Deliveries data..... | 95 |
| Figure 4.62 A sample of the loaded nutrients table data | 96 |
| Figure 4.63 A sample of the Nutrients Table data. | 96 |
| Figure 4.64 A sample of the Nutrients Table data after adding a type column..... | 97 |
| Figure 4.65 A sample of the Nutrients Table data with new “type” column..... | 97 |
| Figure 4.66 A sample of the final Nutrients Table data. | 98 |
| Figure 4.67 A sample of the joined data. | 101 |
| Figure 4.68 Results of the final calculation. | 102 |
| Figure 4.69 Results of final calculation using Python pandas | 103 |
| Figure 4.70 Results of final calculation using MySQL..... | 103 |

List of Tables

| | |
|--|-----|
| Table 1.1 Examples of data preparation tasks in the data transformation process | 11 |
| Table 2.1 Pros and Cons of Applications vs. code for data transformation (White, 2020)..... | 18 |
| Table 2.2 Characteristics of Data Manipulation Languages..... | 18 |
| Table 2.3 Characteristics of Data Transformation Applications | 20 |
| Table 2.4 Results of the comparison of OpenRefine and Trifacta (Petrova-Antonova & Tancheva, 2020) | 22 |
| Table 2.5 Advantages and Disadvantages of OpenRefine and Trifacta (Petrova-Antonova & Tancheva, 2020)..... | 23 |
| Table 3.1 Overview of the support of common data preparation tasks..... | 27 |
| Table 5.1 Advantages and disadvantages of Python pandas and MySQL..... | 110 |
| Table 5.2 Reviewing the functionalities of MySQL and Python pandas..... | 110 |

1 Introduction

Today organizations are collecting and storing huge amounts of data. In the field of Data Science, scientists are attempting to extract insight from the massive volume of data in order to make more informed business decisions. The large volume of data is often referred to as Big Data. Big Data was characterized by Doug Laney in 2001 by three V's (Laney, 2001). *Volume*, as the volume of data is large in many environments. *Variety*, as the data comes in a wide variety of types. And *Velocity*, as the data often is collected at a high rate. Big Data has since been characterized by more V's, such as *Veracity* and *Value*. *Veracity* refers to the accuracy and truthfulness of the data. *Value* refers to the value that this data can provide.

The large amount of data allows for organizations to extract insight into historical trends in the data, which can put them in a better position to make decisions. It can also be fed to machine learning algorithms in order for the algorithms to learn trends and predict the future. However, the raw data that is collected cannot be used for data analytics. The data comes in different formats and can be structured, semi-structured or unstructured. Because it comes in so many different formats and structures, it is very difficult to make sense of the raw data. Therefore, there is a need for data engineering processes to be applied in advance.

The raw data must go through a process in order to be prepared for data analysis. This process might consist of several steps, and they are often implemented in a data pipeline. In the pipeline, each step takes data as an input and produces data as an output, which in turn is inputted to the next step. This continues until the data arrives at the desired format and structure. Data pipelines must be designed specifically for the particular case, but some common steps used in data pipelines are data transformation, augmentation, enrichment, filtering, grouping, and aggregation (Snowflake, 2022).

A crucial phase in data analytic projects is the data transformation phase. This is where the messy raw data is transformed into clean data that we are able to perform analytics on. As this is a very time-consuming and often tedious task, a lot of researchers are focusing on the challenges connected to optimizing the efficiency of this process. The efficiency of the process is obviously connected to the efficiency of the tools used in the

process. Thus, data transformation tools will be investigated in this thesis. Prevalent data transformation tools will be reviewed to provide some insights into what functionalities they support. In a case study, declarative and procedural Data Manipulation Languages (DML) will be compared. The aim is to provide an overview of popular data transformation tools and DMLs and analyze their efficiency. The results of this thesis can be used by data scientists and data engineers to get insights into what tools are available and when to choose them over other tools.

1.1 Problem Description

The aim in this thesis is to compare some of the most popular data transformation tools, their strengths and weaknesses. Some prevalent data transformation tools will be reviewed using their technical documentation, to find out if the tools support common data preparation tasks. The results will give data scientists and engineers an overview of which tools provide the support needed for their projects.

In a case study, two tools will be selected and compared through a data transformation use case. One declarative and one procedural tool will be compared, and the results will provide insight into what the differences of the two approaches are.

Finding a tool that makes the transformation task simple and effective is essential for companies and organizations in order to gain valuable insight from their data without the process becoming too resource-demanding and time-consuming. Structured Query Language (SQL) (W3Schools, n.d.) and Python Pandas (Pandas, n.d.-a) will be used to perform a data transformation task and they will be compared using a framework for comparison. This will give insight into the functionality they provide, and the ease of use of each of the Data Manipulation Languages and will put data scientists at a better position to choose a tool that fits their need.

1.2 Research Questions

The main research questions that will be answered in this thesis are:

- What is the support for common data preparation tasks provided by some of the most prevalent data transformation tools?

- How does declarative vs. procedural Data Manipulation Languages differ in terms of time-consumption, flexibility, expressiveness, usability, and readability?

1.3 Research Design

In order to answer the research questions defined in Section 1.2, a conceptual framework is defined for the comparison of data transformation tools. The framework used in this thesis is developed partly from the framework used in (Hameed & Naumann, 2020) to compare commercial tools for data preparation. Some additional data preparation tasks will be added, and some less relevant ones will be removed. The framework will be used to review some of the most prevalent data transformation tools, including DMLs and applications. This review will be based exclusively on reviewing the technical documentation of these tools. Later, a case study will be conducted to further investigate a selection of tools. The research questions will be investigated by defining a use case and considering some dimensions to compare one declarative and one procedural Data Manipulation Language, namely SQL and Python Pandas. This is done in order to analyze their functionalities and demonstrate the most important differences, limitations and advantages in different contexts. SQL and Python are two of the most frequently used Data Manipulation Languages and are therefore highly relevant to consider in such an analysis (Convertino & Echenique, 2017).

1.3.1 Framework

The core preparation tasks were initially defined in (Hameed & Naumann, 2020) for evaluating commercial data preparation tools. Some of the initial tasks from the paper has been removed and some new ones has been added. These tasks will be used in the evaluation of data transformation tools, and the comparison of declarative and procedural DMLs in this thesis. The data preparation tasks and their corresponding step in the process are listed in Table 1.1.

Table 1.1 Examples of data preparation tasks in the data transformation process

| Step in the process | Preparation tasks |
|----------------------------|--------------------------|
| Data discovery | Find null values |

| | |
|-------------------------|-------------------------------------|
| | Find outliers |
| | Search by pattern |
| | Sort data |
| Data validation | Compare values (selection and join) |
| | Check data range |
| | Check permitted characters |
| | Check column uniqueness |
| | Find type-mismatched data |
| | Find type-mismatched datatypes |
| Data structuring | Change column data type |
| | Delete column |
| | Detect & change encoding |
| | Pivot / unpivot |
| | Rename column |
| | Split column |
| | Transform by example |
| Data enrichment | Assign semantic data type |
| | Calculate column using expressions |
| | Discover & merge external data |
| | Duplicate column |
| | Generate primary key column |
| | Join & Union |
| | Merge columns |
| | Normalize numeric values |
| Data filtering | Delete / Keep filtered rows |
| | Delete empty and invalid rows |
| | Extract value parts |
| | Filter with regular expressions |
| Data cleaning | Change date & time format |
| | Change letter case |
| | Change number format |
| | Deduplicate data |
| | Delete by pattern |
| | Edit & replace cell data |
| | Fill empty cells |
| | Remove extra whitespace |
| | Remove diacritics |

1.3.2 Use Case

To evaluate SQL vs Python Pandas, datasets regarding food production will be used. These datasets come in a couple of formats, namely CSV and XLSX. The datasets describe agricultural enterprises, agricultural properties, deliveries for slaughterhouses, dairies, egg packaging businesses, grain buyers, and seed businesses, and the energy and nutrients contained in the most frequently eaten foods in Norway. In order to evaluate the two DMLs, a question will be defined and attempted answered by performing the necessary steps in the transformation process to get the data in a format where meaning can be extracted from it. Performing these transformation steps will give insights into whether or not SQL and Python Pandas provides the necessary functionalities for the process, and also how easy the process is using the two DMLs.

1.4 Thesis Structure

The thesis structure is presented below.

- Chapter 2 Background and Related Works: The background theory will be presented and some related work is summarized in order to provide an understanding of why this thesis work is necessary.
- Chapter 3 Review of Data Transformation Tools: Some prevalent data transformation tools will be reviewed in order to provide insight into which tools offer which functionalities.
- Chapter 4 Case Study: Comparing Declarative vs. Procedural Data Transformation Tools: Comparing SQL and Python Pandas: A declarative DML will be compared to a procedural DML in order to provide some insight into the advantages and disadvantages posed when working with the two approaches.
- Chapter 5 Conclusion: The results of the thesis work will be discussed and concluded and some suggestions for future work in this field will be given.

2 Background and Related Works

In this chapter the theoretical background needed to understand the issue of efficiency in the data transformation phase will be presented. The first section will discuss the data, how it actually looks initially when it is being collected and the challenges posed when working with this raw data. In the next section the solution to the challenges will be presented, namely data transformation. Here, the data transformation steps from Table 1.1 will be explained. Lastly, some tools typically used in data transformation will be presented and reviewed and some of their characteristics will be highlighted.

2.1 The Raw Data

During data extraction raw data is typically retrieved from several different sources. Companies and organizations can use both internal and external sources to extract data from. Some examples of internal sources are transactional data, such as purchases made by the company or organization, or purchases made by customers. Customer Relationship Management systems (CRMs) can also add insights into geographical details of the customers, and this data can be combined with the purchase history of a customer, for instance. Companies and organizations often have internal documents containing information about things like business activities, policies and processes. (aunalytics, n.d.) With the increase of Internet of Things (IoT) solutions, companies and organizations can also collect data from sensors and devices. Examples of external sources are social media, official records and publicly available data on the web.

Gathering data from all these different sources gives us datasets that are often inconsistent. Some records can be incomplete, containing null values or missing values. Sometimes records are duplicated in the dataset. Datasets can also contain comments that are meant for humans to read in order to interpret the data table. These are often irrelevant for computers and can even cause problems in an analysis. Thus, in order for data scientists to be able to perform data analysis, the messy data has to be cleaned. This entails for instance removing missing values, fixing data type mismatches, deduplicating data. The structure of the data should also be tidy. In (Wickham, 2014), the author presents three characteristics of tidy data. These are:

1. Each variable is a column
2. Each observation is a row
3. Each type of observational unit is a table

The author states that this standard structure of data facilitates data analysis, and the three characteristics will therefore be used to define clean data throughout this thesis. The process of transforming data from the initial format and structure to a structure where we are able to apply data analysis, is called data transformation. This process will be explained in the next section.

2.2 Data Transformation

Data transformation is any process that takes data as an input and produces data as an output. Since the data can come from different sources it can have all kinds of different formats, structures, and values. Data transformation enables us to get the data in a format and structure that is understandable both for humans and computers, and especially for systems and applications that requires a certain format or structure. It also enables us to check for, and improve, data quality, which prevents problems later, for instance when analysis is applied. Things like duplicate data, abnormalities and missing values can be taken care of through data transformation. The data transformation process typically includes some common steps such as data discovery, data validation, data structuring, data enrichment and data filtering (Hameed & Naumann, 2020). These steps can be performed in various orders and some steps are performed several times. Figure 2.1 illustrates the steps of data transformation that will be considered in this thesis. The steps will be described in a bit more detail below.



Figure 2.1 The steps of Data Transformation

Data discovery is the first step of the process. As the raw data is often messy and inconsistent, the first step is to look at the data and try to identify the format and structure that the source data has (TIBCO, n.d.). In order to be able to change the data to meet the requirements of the target system, it is necessary to understand where the data is at the moment. This step is typically done using tools for data profiling.

Data validation entails checking for data quality. This can include the correctness and completeness of the data, for instance (Hameed & Naumann, 2020). The goal is to ensure the data's consistency and quality. Types of validations could for instance be allowed character checks, data type checks, format checks and uniqueness checks (contributors, 2021). This is a step that might be conducted several times throughout the process.

Data structuring entails tasks that change the structure of the data. In some cases, we want data to be in a long format, where data values repeat in the columns. In other cases, we want a wide format, where values do not repeat in the columns (Zach, 2021). These formats are illustrated in Figure 2.2. The structuring of data can include tasks such as pivoting, changing data types, deleting and renaming columns etc. (Hameed & Naumann, 2020). This is done in order to reach a structure where the data is better understood by data analysis tools.

| Wide Format | | | | Long Format | | |
|-------------|--------|---------|----------|-------------|----------|-------|
| Team | Points | Assists | Rebounds | Team | Variable | Value |
| A | 88 | 12 | 22 | A | Points | 88 |
| B | 91 | 17 | 28 | A | Assists | 12 |
| C | 99 | 24 | 30 | A | Rebounds | 22 |
| D | 94 | 28 | 31 | B | Points | 91 |
| | | | | B | Assists | 17 |
| | | | | B | Rebounds | 28 |
| | | | | C | Points | 99 |
| | | | | C | Assists | 24 |
| | | | | C | Rebounds | 30 |
| | | | | D | Points | 94 |
| | | | | D | Assists | 28 |
| | | | | D | Rebounds | 31 |

Figure 2.2 Illustration of long and wide data formats (Zach, 2021)

Data enrichment means adding value or supplementary information to existing data from separate sources, and it typically means augmenting existing data with new or derived data values using data lookups, primary key generation, and inserting metadata (Hameed & Naumann, 2020). Enrichment can be achieved by combining first party data from internal sources with either disparate data from other internal systems, or third party data from external sources (Trifacta, n.d.-a).

Data filtering is used to create a subset of data from a dataset and can for instance be used to look at data for a particular period of time or exclude erroneous observations from an analysis (Facer, n.d.). It helps improve data quality by using predefined criteria, such as removing records that contain empty values or that do not conform to some user-defined pattern (Hameed & Naumann, 2020).

Data cleaning is a process that is done in order to reach data quality. This is typically done several times throughout the data transformation. Data cleaning may include tasks such as deduplication of data, editing and replacing cell data and removing whitespace (Hameed & Naumann, 2020). The goal is to ensure any corrupt or erroneous records are corrected or removed entirely from the data, and thus, improving data quality.

2.3 Data Transformation Tools

In this section tools used in the process of data transformation will be discussed. First, the two different approaches to data transformation tools will be presented, namely applications and code. Then, a review of data manipulation languages and data transformation applications will be presented in Section 2.3.2.

Data transformation is typically done through the use of either applications or code. What to choose depends on the prior knowledge and the given context, and both ways have advantages and disadvantages. Data Manipulation Languages (DML) such as Python Pandas, SQL, and R, are used to perform the data transformation through writing the code yourself. Applications lets you do transformations using a graphical user interface to make the job faster. However, there are some pros and cons of the two approaches. Nick White highlights

some of the pros and cons of applications in an article on LinkedIn (White, 2020) and these are listed in Table 2.1.

Table 2.1 Pros and Cons of Applications vs. code for data transformation (White, 2020)

| Pros | Cons |
|--|---|
| Quicker to learn, programming languages has a steep learning curve | Cost money (often significant amounts) |
| Easier to understand (not as advanced as code) | Executes slower than code |
| Have the potential to be easily integrated into a wider data governance eco-system | Do not integrate easily (or at all) into CI/CD workflow |

2.3.1 Declarative vs. Procedural

Data Manipulation Languages (DML) can be declarative or procedural. Declarative languages are high-level languages, while a procedural language is a low-level language embedded in a general-purpose programming language. Declarative DMLs specify properties of the data that is to be retrieved from the database, while procedural languages specify how to access the data (Özsu, 2017). Examples of popular declarative and procedural DMLs are SQL and Python Pandas library, respectively.

2.3.2 Review of Data Transformation Tools

In this section, some data manipulation languages and some data transformation applications will be reviewed. The tools’ characteristics will be highlighted.

Data Manipulation Languages

Table 2.2 shows the characteristics of Data Manipulation Languages (DML). R is not in itself a DML, however, it has additional packages that adds capabilities for data manipulation. As can be seen in Table 2.2, the packages *dplyr* and *tidyr* provides functionality to support data profiling, data restructuring and data integration.

Table 2.2 Characteristics of Data Manipulation Languages

| Language | Characteristics |
|----------------------|--|
| R (Foundation, n.d.) | Data analysis and statistical language Provides several Data Wrangling packages |

| | |
|--|---|
| | <p>Packages such as dplyr and tidyr include functionality to support data profiling, data restructuring and data integration</p> <p>Packages such as ggplot2 provide powerful data visualization functions</p> <p>Many operations that allow for populating new columns and operations to combine datasets</p> <p>Lacks native support for Date/Time manipulation operations</p> <p>Main limitation is the steep learning curves for non-professionals and the lack of Data Wrangling functions provided in native R, finding the required functions across several packages can be time-consuming</p> |
| <p>Python (Pandas) (Pandas, n.d.-a)</p> | <p>Python is a production ready language used in a wide range of industries, research and engineering workflows</p> <p>Supports extraction of data from many sources, such a plain text files and CSV files, or the web</p> <p>Provides functionality to also do analysis, such as regression tests, time series manipulation, and statistical analysis</p> <p>Also offers machine learning frameworks for developing ML algorithms to apply on the transformed data, and is particularly well suited for deploying machine learning at a large scale</p> <p>Iterative, readable, portable, broadly applicable approach to data manipulation, fast</p> <p>Easy to join data from several sources</p> <p>In-memory processing (can be disadvantageous with large datasets)</p> <p>Supported by a large community, continuously extended libraries and tools</p> <p>Can be used with notebooks to collaborate</p> |
| <p>SQL (W3Schools, n.d.)</p> | <p>Supports extraction of data from databases</p> <p>Simple functions, but also a narrow range of functions compared to Python</p> <p>Limited functions for processing, analyzing or experimenting with data</p> <p>Can run some data processes, but may be inefficient or complicated because the ability to perform calculations efficiently is not part of the languages design</p> |

Specifically designed to query and extract data, and one of its main strengths is merging data from multiple tables within a database

Higher-level data manipulation, such as statistical analysis, regression tests, and time-series data manipulation are very difficult to achieve using SQL exclusively

Data Transformation Applications

Table 2.3 shows the characteristics of Data Transformation Applications.

Table 2.3 Characteristics of Data Transformation Applications

| Tool | Characteristics |
|---|--|
| Altair Monarch Data Preparation (Altair, n.d.) | Provides common data preparators for structured data, but also transforms tables from within PDF and text files to tabular data Extracted tables can be merged with other tables using a variety of join and union operations |
| Paxata Self-Service Data Preparation (DataRobot, n.d.) | Provides features for organizing and preparing structured data Deals efficiently with semi-structured data User experience is designed to fit non-experts |
| SAP Agile Data Preparation | Runs on top of SAP's HANA database system (SAP, n.d.) Provides common data preparators with some specific system features such as Schedule Snapshot, interactive suggestions to help users navigate and prepare data efficiently and multi-user access that enables collaboration |
| SAS Data Preparation (SAS, n.d.) | Part of SAS Viya System Management, which runs its operations with distributed in-memory processing Has common data preparation features, but also offers code-based transformations for users to write and share custom code to transform data, supporting re-usability of preparation pipeline |
| Tableau Prep (Tableau, n.d.) | Implements a workflow approach to organize and prepare messy data Users can perform multiple operations simultaneously Tableau Prep Builder is designed to develop workflows, manage data and apply operations on data Tableau Prep Conductor is designed to share, schedule, and monitor the flows |
| Tabula (Tabula, n.d.) | Web-based Data Wrangling tool developed with focus on data format transformation |

| | |
|---|---|
| | Generates Excel, CSV or JSON data file format via extracting data tables uploaded in PDF data file formats |
| Mr Data Converter (thdoan, n.d.) | <p>Similar to Tabula in terms of data format transformations</p> <p>Does not support extraction from PDF files</p> <p>Supports uploading files in either TSV or CSV</p> <p>Generates output files in a number of formats, such as Actionscript, ASP/VB Script, MySQL, Ruby, HTML, XML, or JSON</p> |
| Trifacta (Trifacta, n.d.-b) | <p>Supports a comprehensive set of data profiling and data restructuring functionality</p> <p>Does not support extraction of data from PDF files</p> <p>Can convert files from CSV or TSV to JSON</p> <p>Allows for integration with a wide range of data science and data ingestion technologies</p> |
| OpenRefine (OpenRefine, n.d.) | <p>Supports data profiling, data cleaning and data restructuring</p> <p>Offers operations for data cleaning and string manipulations, but lacks advanced statistical and restructuring operations</p> <p>Supports moving and dropping columns, date/time operations and array operations</p> <p>Limited support for extracting data from PDF files</p> <p>Supports converting data formats from JSON or TSV to CSV</p> <p>Its big data capabilities have shown limitations to scale Gigabyte order of magnitude</p> |
| Talend Data Preparation (Talend, n.d.) | <p>Designed for importing, structuring and transforming data</p> <p>Web-based visual interface that enables users to develop data preparation workflows</p> <p>Offers common data cleaning and restructuring functionality, such as column manipulation operators, like rename, create or drop, and fill in missing data</p> <p>Provides functionality for string manipulation</p> <p>Supports few operations for Date/Time manipulation compared to Trifacta and OpenRefine</p> |

2.4 Related Works

To address the issue of efficiency in the data transformation phase, several researchers have taken a similar approach to provide insights into the functionality of different data transformation tools. In (Petrova-Antonova & Tancheva, 2020), the authors presented a comparative analysis of widely used tools for data cleaning. They also perform a

case study where they compare OpenRefine and Trifacta Wrangler in order to address the issues they face using these tools. Both tools provide similar functionalities. However, the results of applying the functions on the same dataset don't necessarily match between the tools. In Table 2.4 the results of applying the different techniques using the two tools are shown. As seen in the table, OpenRefine for instance removed 3 duplicated rows, which is 100% of the existing duplicated rows. Trifacta only removed 2 out of 3 existing duplicated rows. OpenRefine also performs better in clustering of text facets. Except for these two techniques, the tools perform the same.

Table 2.4 Results of the comparison of OpenRefine and Trifacta (Petrova-Antonova & Tancheva, 2020)

| Technique | Results (Number) | OpenRefine | Trifacta |
|--------------------------------------|--------------------------------|-------------------|-----------------|
| Duplicate rows removal | Existing duplicate rows | 3 | 3 |
| | Removed duplicate rows | 3 (100%) | 2 (66.66%) |
| Cleaning of structural errors | Fields with removed whitespace | 413 | 413 |
| | Fields with incorrect format | 6 | 6 |
| | Fields with corrected format | 6 (100%) | 6 (100%) |
| Clustering of text facets | Unified fields | 413 | 98 |
| Outliers | Existing outliers | 1 | 1 |
| | Removed outliers | 1 (100%) | 1 (100%) |
| Missing values | Existing missing values | 14123 | 14123 |
| | Identified missing values | 14123 (100%) | 14123 (100%) |

The authors also highlight some advantages and disadvantages of the two tools. These are shown in Table 2.5.

Table 2.5 Advantages and Disadvantages of OpenRefine and Trifacta (Petrova-Antonova & Tancheva, 2020)

| | Advantages | Disadvantages |
|-------------------|---|--|
| OpenRefine | <ul style="list-style-type: none"> (1) Easy modification of a single field (2) The field is not necessary to satisfy a given criterion to be changed (3) Multiple records processing (4) Statistics on the number of records processed with a given operation | <ul style="list-style-type: none"> (1) Doesn't support processing of large datasets |
| Trifacta | <ul style="list-style-type: none"> (1) Record the applied data processing steps as a "Recipe" procedure (2) Non-blocking processing of large datasets (3) Supports easier to use functions (4) Different method for filling the missing values | <ul style="list-style-type: none"> (1) Single field cannot be modified (2) Allows only modification of multiple fields that meet a given criterion (3) Missing information for the number of records processed with a given operation |

In (Patil & Hiremath, 2018), the authors argue that in order for data preparation tools to be efficient, self-service solutions using machine learning approaches are required. In this paper, the Trifacta tool's functionalities are tested through a data wrangling case study to demonstrate the efficiency. The case study shows that Trifacta allows for human machine interactive transformation of real world data, and that it enables business analysts to iteratively explore predictive transformation scripts with the help of highly trained learning algorithms (Patil & Hiremath, 2018).

A survey of commercial tools for data preparation is presented in (Hameed & Naumann, 2020). The authors collected 42 commercial tools and listed their data preparation capabilities. Later they picked out seven tools which they investigated further. A preparator matrix showed each preparator's availability for each tool. They selected three examples of preparators to demonstrate their function. Then they gave examples of how

one tool solved each of the preparators. Through the investigation of the tools, the authors found that all the tools needed the data that was being inputted to be clean beforehand.

They list a few assumptions that most tools make (Hameed & Naumann, 2020):

- Single table file (no multi-table files)
- Specific file encoding
- No preambles, comments, footnotes, etc.
- No intermediate headers
- Specific line-ending symbol
- Homogenous delimiters
- Homogenous escape symbols
- Same number of fields per row
- Relational data (no nested or graph-structured data, such as XML, JSON, or RDF)

The authors of (Hameed & Naumann, 2020) also emphasizes the need for automated and intelligent solutions for the data preparation tasks. They also found that IT- and domain knowledge is needed due to the shortcomings of the tools. Although the paper focused on preparing structured data, the authors also highlight the lack of basic preparation steps for unstructured data, such as textual data.

The comparative analysis in (Petrova-Antonova & Tancheva, 2020) only focuses on a few activities of the data cleaning process. In addition, it only considers two applications for data transformation, namely OpenRefine and Trifacta Wrangler. The case study in (Patil & Hiremath, 2018) only considers Trifacta and is focused on the need for self-service in data wrangling tools. The survey in (Hameed & Naumann, 2020) is focused on commercial tools and the lack of automation and intelligent solutions. The reviews of data transformation tools only consider a few tools and a few common data preparation tasks. In this thesis, more tools and preparation tasks will be considered in order to provide a larger overview of prevalent tools and their support of data preparation tasks. None of the case studies are comparing declarative and procedural data manipulation languages either. This area of research does not seem to have been explored. Thus, these two things are what will be investigated in this thesis.

2.5 Ethical considerations

When dealing with data, and particularly personally identifiable information (PII), there are some guides on measures in information security. For instance, the CIA triad. CIA is an abbreviation of Confidentiality, Integrity, and Availability. In information technology the three things that typically are attacked by hackers are the confidentiality of information, the integrity of information, and also the availability of information. In the data transformation phase all three of these aspects should be considered. Making sure the data is not altered in a way that makes it lose integrity is one measure to ensure the process is conducted ethically. Another thing to consider is whether the temporary storage of PII, from source to target, is safe from attacks. This involves confidentiality particularly, as the data probably still is available in the source.

The focus on the rights of individuals and obligations of organizations has increased over the years, and laws that regulate both the gathering and storage of PII have been developed. The General Data Protection Regulation (GDPR), which applies to countries within the European Union (EU), has a goal of protecting personal data and give users control over their data (Consulting, n.d.). China has adopted a lot of GDPR's principles in their People's Republic of China (PRC) Personal Information Protection Law (PIPL) (Laird, 2021). In the U.S several laws have been implemented, but none of them are like GDPR and PIPL. To mention a couple of them, they have The Healthcare Insurance Portability and Accounting Act (HIPAA) that governs health information collection and The Children's Online Privacy Protection Act (COPPA) that governs collection of minors information and prohibits online companies from collecting personal information from children under the age of 12 unless parents have given verifiable consent (Green, 2021).

Another thing to consider is the ethics around what the transformed data is used for. Whether the data is to be used in an ethical manner or not. For instance, many tech giants are gathering information about how their consumers are behaving on their applications. Often their activities are sold to companies that send targeted commercials to the consumer. Consumers are often not aware of the effect that this data gathering has on them. This consideration is not specifically important in the data transformation phase, but it is still something to consider.

3 Review of Data Transformation Tools

In this chapter, some popular data transformation tools and languages will be reviewed. The review will provide some insights into what features each of the tools provide and what they are lacking. The results of the review will give some general guidance for data engineers and data scientists that are looking for a tool for the data transformation process. In Table 3.1 an overview of the available data preparation tasks in each of the selected tools is shown. In the table, the different data preparation tasks' availability is indicated by "✓", "*", or "-". "✓" indicates the data preparation task can be performed using a specific method or component/block. "*" indicates the data preparation task can be performed, but it requires some extra effort to create a script or workflow that performs the tasks, and there is no single method or component/block that solves the task. "-" indicates that the data preparation task is not available. This review is based exclusively on reading the technical documentation and without any exploration of the tools beyond this. The availability of preparation tasks for the tools will be provided in a table and the results for each tool will be discussed in Sections 3.1.1 to 3.1.9. Lastly, the results of the review will be presented in Section 3.2.

Table 3.1 Overview of the support of common data preparation tasks. The data preparation tasks supported by each of the data transformation languages and tools are shown. “✓” means that the data preparation task is available in a simple method/function or component of the language/tool. “*” means the data preparation task can be performed but it requires additional libraries or some kind of workaround, meaning there is no specific method or component that solves the task alone. “-” means the data preparation task is not available.

| | Python pandas | SQL | R | RapidMiner | KNIME | Trifacta | OpenRefine | Talend | Tableau Prep |
|-----------------------------|---------------|-----|---|------------|-------|----------|------------|--------|--------------|
| Data profiling | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Find missing or null values | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Find outliers | * | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Sort data | ✓ | ✓ | * | ✓ | ✓ | ✓ | ✓ | - | ✓ |
| Filter data | ✓ | ✓ | * | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Check permitted characters | * | * | * | - | ✓ | ✓ | ✓ | ✓ | - |
| Check column uniqueness | ✓ | ✓ | ✓ | - | ✓ | ✓ | - | ✓ | - |
| Change column data type | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Delete column | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Pivot / Unpivot | ✓ | * | * | ✓ | ✓ | ✓ | ✓ | - | ✓ |
| Rename column | ✓ | ✓ | * | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Split column | ✓ | * | * | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Grouping data | ✓ | ✓ | * | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Aggregating data | ✓ | ✓ | * | ✓ | ✓ | ✓ | - | ✓ | ✓ |
| Fuzzy matching | * | * | * | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Duplicate column | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Join and Union | ✓ | ✓ | * | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Merge columns | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Delete / Keep filtered rows | ✓ | ✓ | * | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |

| | | | | | | | | | |
|-------------------------------------|---|---|---|---|---|---|---|---|---|
| Delete empty or invalid rows | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Change date and time format | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | - |
| Change letter case | ✓ | ✓ | * | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Deduplicate data | ✓ | ✓ | * | ✓ | ✓ | ✓ | ✓ | ✓ | - |
| Edit and replace cell data | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Remove diacritics | ✓ | * | * | - | ✓ | - | ✓ | ✓ | - |
| Remove whitespace | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | * | ✓ | ✓ |

3.1 Functionalities of the different tools

3.1.1 Python pandas

The Python pandas library offer most of the data preparation tasks in simple methods. However, some of the data preparation tasks must be solved by creating custom scripts. This makes the job a little more time-consuming and requires a bit more programming experience than what using simple methods provided in the library would. Although some of the tasks requires a bit of programming experience, Python is not a very challenging programming language to learn. As previously mentioned in Table 2.2 of Section 2.3.2, Python provides functionalities for data analysis as well as data manipulation. It also offers machine learning frameworks and is particularly well suited for deploying machine learning algorithms at scale. Considering these factors, Python and the pandas library are strong tools when working with data science. The wide range of possibilities these tools provide makes it an effective and applicable option for the entire data science project.

3.1.2 SQL

SQL solves most tasks using relatively simple queries. Some of the tasks, however, require a complicated workaround in order to be solved. Examples are pivoting data or removing diacritics. These tasks therefore become very time-consuming to perform, especially for non-experts. Thus, SQL is efficient for performing most of the tasks but the ones that do not have an obvious solution, gets very complicated and requires experience.

3.1.3 R

R provides some data preparation tasks using simple functions. However, R is made for data analysis and additional packages are required to get the necessary data preparation functionalities. Using these additional packages, all the data preparation tasks in Table 3.1 are possible to achieve with R. The drawbacks of R as a data manipulation language is the inefficiency of the process due to the required additional packages. Over half of the data preparation tasks required, or was at least more efficiently performed with, additional packages. The *dplyr* package provide many functions for data preparation tasks, e.g., for sorting, filtering, aggregating, and deduplicating data. The *tidyr* package provide functionalities for pivoting data and splitting columns, for instance. In addition, the *grepl* package was mentioned for checking permitted characters, and the *stringi* package for removing diacritics. Some of the data preparation tasks are possible to perform without additional packages, but not in a simple and efficient way. These are also marked as “*” in Table 3.1, as the additional packages provide more efficient ways of solving the tasks. The large set of packages required to do all the data preparation tasks makes the process more time-consuming, as a data scientist would have to look up the different packages needed to perform the data preparation process. However, this probably is only a problem the first few times. Once the required packages are identified they provide the necessary functionalities. As mentioned in Table 2.2 of Section 2.3.2, R has a steep learning curve which is also worth considering when choosing which tool to use for a data science project. Overall, the lack of native support for data preparation tasks and the steep learning curve for non-experts makes R a poorer choice than for instance Python pandas.

3.1.4 RapidMiner

RapidMiner is one of the applications reviewed in this chapter. As seen in Table 3.1, RapidMiner provides most of the preparation tasks. However, three of them are marked as unavailable. There might be ways to solve these tasks, but it is not clear from the documentation, and it is therefore assumed that these tasks either are completely unavailable or at least requires a workaround in order to solve. RapidMiner still provides a simple solution to the data preparation tasks that are available, and therefore is considered a highly efficient tool for data preparation.

3.1.5 KNIME

KNIME is another application which provides all of the data preparation tasks. In KNIME, the components which perform the different tasks are called nodes. Some of the tasks require a few nodes in order to be solved, but they are still easy to perform. With all the data preparation tasks available in the application, KNIME seems to provide even more functionalities than RapidMiner, and is therefore considered a slightly more efficient tool.

3.1.6 Trifacta

The Trifacta application only misses functionality for removing diacritics. The rest of the data preparation tasks are easily performed in the application. As with RapidMiner and KNIME, this makes the application a highly efficient tool for data preparation. The simple user interface and the availability of data preparation tasks makes Trifacta a great tool to consider, especially for non-experts.

3.1.7 OpenRefine

The OpenRefine application is missing a few of the data preparation tasks, as seen in Table 3.1. Most important is the grouping and aggregation of data. These data preparation tasks are included in all the previously reviewed applications and languages. Although OpenRefine offers many of the preparation tasks in Table 3.1, the lack of data preparation tasks should be considered when choosing a tool for the data transformation process as it might affect the efficiency of the process. Spending time on workarounds or having to use additional tools to perform the data transformation makes the process more time-consuming.

3.1.8 Talend Data Preparation

As shown in Table 3.1, Talend Data Preparation is missing a couple of data preparation tasks but provides most of them. As with OpenRefine, the lack of data preparation tasks has to be taken into consideration when choosing a tool, as workarounds or use of additional tools is time-consuming and inefficient.

3.1.9 Tableau Prep

Tableau Prep are missing quite a few data preparation tasks, as seen in Table 3.1. It is the tool that provides the least tasks. However, what tasks are needed in the particular context has to be considered when choosing a tool. If the missing preparation tasks are not relevant in that context, Tableau can still be an efficient tool.

3.2 Results of the review

Table 3.1 provides a quick overview of which data preparation tasks are provided from the different tools. This can be considered when selecting a tool for the data transformation process, in order to get some insight into the efficiency of the tool. Many missing data preparation tasks might indicate that the tool is missing important functionality and therefore is inefficient. It's also worth mentioning that the tools are evolving, and suggestions from the community often are taken into account in new releases of the tools. Thus, there is no guarantee that the tool that currently provides the most data preparation tasks will always be the better choice.

The review of the technical documentation revealed that that the languages Python pandas, SQL, and R often requires workarounds or additional packages and libraries. Therefore, the decision should be made also considering the context it is being used in. For instance, if the data is going to be used for Machine Learning, Python is a good choice, as previously mentioned. If the purpose is to do data analysis, R is specifically designed for that and is particularly well suited. And as previously mentioned, the languages also require some programming experience. The applications, on the other hand, all provide a user interface which makes the process much easier for non-experts. However, the flexibility of the applications should be considered, as the user interface might limit the user to only a set of specific operations.

4 Case Study: Comparing Declarative vs. Procedural Data Transformation Tools

In this chapter a case study of a selection of two data transformation tools from the review in Section 3 will be conducted. One of the tools is declarative and the other is procedural. The case study considers a situation where a data scientist is handed a set of data files that needs to be transformed in order to answer a question. The question is:

How many calories worth of food did each municipality in Norway produce per square meter in 2019?

The question is to be answered using six different data files. The case will be solved using Python pandas library, which is a procedural approach to data transformation, and SQL, which is the declarative approach to the data transformation. The necessary data preparation tasks will be performed and each of the tools will be evaluated based on whether they provide the functionality for performing the tasks or not. In addition, the tools will be evaluated with regards to some measures, namely, time-consumption, flexibility, expressiveness, usability, and readability.

4.1 Python pandas library

In this chapter, the process of preparing the data for an analysis where the answer to the question can be found will be demonstrated using the Python library *pandas* (*Pandas, n.d.-a*). The process contains two general steps: Data Discovery and Data Preparation. During the Data Discovery the raw data will be investigated, and the necessary data preparation steps will be identified. Later, during the Data Preparation step, the necessary data preparation tasks will be performed in order to reach the desired format and structure. In the end the different fields of the prepared datasets can be mapped before the data can be integrated. The final data can be used to answer the question.

4.1.1 Importing pandas and loading the data

In this case study the Anaconda distribution has been installed beforehand, and the pandas library has also been installed using pip (pypi, n.d.). Pandas library is then easily imported into a Jupyter Notebook (Jupyter, 2022) as shown in the code line below:

```
1 import pandas as pd
```

The data is loaded simply by using pandas' reader functions. CSV files are read using the **read_csv()** function (pandas, n.d.-u) where the **sep** argument specifies the delimiter used in the files. In this case all the CSV files use “;” as delimiter, therefore **sep** is set equal to “;”. The XLSX file is read using the **read_excel** function (pandas, n.d.-v). The code lines loading the different data files are shown below:

```
1 # Load the data
2 ac_businesses_init = pd.read_csv("foretak.csv", sep=";")
3 ac_properties_init = pd.read_csv("grunneiendommer.csv", sep=";")
4 meat_deliveries_init = pd.read_csv("slakteri.csv", sep=";")
5 dairy_deliveries_init = pd.read_csv("meieri.csv", sep=";")
6 egg_deliveries_init = pd.read_csv("eggpakkeri.csv", sep=";")
7 grain_deliveries_init = pd.read_csv("korn.csv", sep=";")
8 nutrients_table_init = pd.read_excel("matvaretabellen.xlsx")
```

4.1.2 Agricultural Properties

Data Discovery

This data file is from *Felles Datakatalog* and contains information about all the agricultural properties in Norway¹. Properties in Norway have different identification numbers called “gardsnummer”, “bruksnummer” and “festenummer”. These are irrelevant for this case study, but they are some of the columns in the raw data. In addition, the properties have a column which holds the identification number of the municipality the property is in. This column is named “komnr” and is essential for this case study. The rest of the columns provides information about the area of land each of the properties has. Some of the land is categorized as farmland, some is categorized as forest, along with a few other categories. Most of these are not important in this case study. The column that is interesting

¹ <https://data.norge.no/datasets/0dabd5e4-514c-4667-adfc-aa4fe741142b>

is the one named “jordbruksareal”, as this is the one that tells us the area of farmland of the property.

To explore the data the **head()** method (pandas, n.d.-h) can be used. This method outputs a small sample of a Series or DataFrame. The default number of elements in the sample is 5 but adding a number inside the parentheses will output the given number of elements.

```
1 ac_properties_init.head()
```

This outputs the data shown in Figure 4.1.

| | komnr | gardsnr | bruksnr | festenr | bruksnavn | jordbruksareal | skogareal | annet_areal | fulldyrket | overflatedyrket | inmarksbeite | l_komnr | l_gardsnr | l_bruksnr | l_festenr |
|---|-------|---------|---------|---------|------------------|----------------|-----------|-------------|------------|-----------------|--------------|---------|-----------|-----------|-----------|
| 0 | 301 | 1 | 1 | 0 | Bygdøy kongsgård | 730.0 | 1000.0 | 301.0 | 674.0 | 40.0 | 16.0 | 301 | 1 | 1 | 0.0 |
| 1 | 301 | 2 | 8 | 0 | Bygdøy kongsgård | 0.0 | 1.0 | 19.0 | 0.0 | 0.0 | 0.0 | 301 | 1 | 1 | 0.0 |
| 2 | 301 | 1 | 3 | 0 | NaN | 0.0 | 0.0 | 5.0 | 0.0 | 0.0 | 0.0 | 301 | 1 | 3 | 0.0 |
| 3 | 301 | 11 | 42 | 0 | NaN | 0.0 | 12.0 | 0.0 | 0.0 | 0.0 | 0.0 | 301 | 1 | 3 | 0.0 |
| 4 | 301 | 29 | 115 | 0 | NaN | 0.0 | 27.0 | 11.0 | 0.0 | 0.0 | 0.0 | 301 | 1 | 3 | 0.0 |

Figure 4.1 A sample of the agricultural properties data

What needs to be done

What is needed from this dataset is the total area of farmland of each municipality.

To get this, the following data preparation tasks are necessary:

- Drop irrelevant columns
- Remove null data
- Group the data by Municipality ID and calculate the total area of farmland for each municipality

Data Preparation

Drop irrelevant columns

In this case, the only columns that are relevant are Municipality ID (komnr) and area of farmland (jordbruksareal). Since only two columns is needed and 13 columns are irrelevant, it is easier to create a subset of the data only containing the desired columns. This is done in the following line of code:

```
1 ac_properties_subset = ac_properties_init[['komnr', 'jordbruksareal']]
```

Remove null data

The properties in this dataset has area of farmland, forests and area of other types. Some of the properties in the data only has forests, and the area of farmland is therefore 0.0. As mentioned in the previous section, these properties are not interesting in this case and can therefore be removed. They are removed by creating another subset of data where the rows that has area of farmland greater than 0 is included, and the rest is filtered out. This is shown in the following line of code:

```
1 ac_properties = ac_properties_subset[ac_properties_subset['jordbruksareal' > 0]]
```

Looking at the current data, it only contains the two columns we desired and the rows where the area of farmland seem to be gone. This is shown in Figure 4.2.

| | komnr | jordbruksareal | |
|-------------------------|--------|----------------|-------|
| | 0 | 301 | 730.0 |
| | 15 | 301 | 23.0 |
| | 19 | 301 | 39.0 |
| | 20 | 301 | 37.0 |
| | 23 | 301 | 950.0 |
| | ... | ... | ... |
| | 314825 | 5444 | 22.0 |
| | 314828 | 5444 | 16.0 |
| | 314830 | 5444 | 34.0 |
| | 314831 | 5444 | 20.0 |
| | 314832 | 5444 | 25.0 |
| 235262 rows × 2 columns | | | |

Figure 4.2 Output of the current subset of the agricultural properties data

Group data by Municipality ID

Now, to find the area of farmland per municipality the **groupby()** (pandas, n.d.-g) and **agg()** (pandas, n.d.-c) methods can be used. The **groupby()** method groups the data by the column that is added in the parentheses. In this case, the data needs to be grouped by

Municipality ID (komnr). Since the total area of farmland per municipality is what is needed, the **agg()** method is used to specify which aggregation functions to use on which columns. Here, the area of farmland needs to be summarized for each municipality. This results in the following line of code:

```
1 area_of_farmland_per_municipality = ac_properties.groupby(['komnr'], as_index=False)
    .agg({'jordbruksareal': 'sum'})
```

Looking at the current dataset, it now only contains the municipality ID and area of farmland columns. This can be seen in Figure 4.3.

| | komnr | jordbruksareal |
|---|-------|----------------|
| 0 | 301 | 9246.0 |
| 1 | 412 | 2.0 |
| 2 | 419 | 3.0 |
| 3 | 513 | 1.0 |
| 4 | 522 | 44.0 |

Figure 4.3 Output of the prepared agricultural properties data

Comparing the initial dataset and the final dataset using the **shape** attribute (pandas, n.d.-o), it can be seen that the data has been significantly reduced, with about 314,500 rows and 13 columns. The code lines and output are shown in Figure 4.4. The output shows the number of rows and columns, respectively, in parentheses.

```
1 ac_properties_init.shape
```

```
(314833, 15)
```

```
1 area_of_farmland_per_municipality.shape
```

```
(372, 2)
```

Figure 4.4 Comparing the size of the initial and final agricultural properties dataset

4.1.3 Meat Deliveries

This data file contains information about the deliveries of meat from each farm². It contains columns for organization ID, name of the agricultural business/owner of the farm and the municipality ID. In addition, each meat type has a column giving the amount of each type of meat produced by each farm.

Data Discovery

The next dataset is explored in the same way using the **head()** method. This dataset contains the amount of produced meat per organization. The amount is also linked to a municipality ID. From this dataset, the amount of each type of meat for each municipality is needed. The necessary steps to reach this is listed below:

- Drop irrelevant columns
- Unpivot data
- Remove null data
- Split column
- Simplify meat types
- Group by municipality ID and type of meat

Data Preparation

Drop irrelevant columns

The data contains three columns that are irrelevant for the case study, and these are removed in the code line below using the **drop()** method (pandas, n.d.-f). The *axis* parameter is set equal to 1, which means it will drop columns with the labels specified in the list.

```
1 meat_deliveries_dropped = meat_deliveries_init.drop(['navn', 'orgnr', 'ull_kg'], axis=1)
```

Unpivot data

The data is in a wide format, meaning that it has many columns of data. This is because for each organization, there is one column for each meat type. However, one organization typically produced one or a few types of meat. Therefore, in one row of data

² <https://data.norge.no/datasets/713abd23-2247-4287-a969-cf0079318685>

there is one or a few cells giving the amount of produced meat, but most of the cells will be 0 since no organization produces all types of meat. Referring back to section 2.1, the characteristics of tidy data described in (Wickham, 2014) was:

- Each variable is a column
- Each observation is a row
- Each type of observational unit is a table

Starting from the bottom, the data table is one observational unit as this is a table of the amount of meat produced by each organization. The variables needed in this data is the type of meat and the amount, so these should be the columns. Each observation should in this case be the amount of each meat type produced in each municipality. Therefore, the data is unpivoted into a long format, keeping the municipality ID column and naming the new columns “type” and “amount”.

The unpivoting of the data is done using the **melt()** method (pandas, n.d.-l). In the method, identifier variables (`id_vars`) and measured variables (`value_vars`) can be set. Identifier variables are the ones that are kept the way they are, while measured variables will be unpivoted. The line of code is shown below:

```
1 meat_deliveries_melted = pd.melt(meat_deliveries_dropped, id_vars=['komnr'],
                                var_name='type',
                                value_name='amount')
```

The data now consists of the columns “komnr”, “type” and amount, as shown in Figure 4.5.

| | komnr | type | amount |
|----------|--------------|-------------|---------------|
| 0 | 137 | and_kg | 0.0 |
| 1 | 1119 | and_kg | 0.0 |
| 2 | 528 | and_kg | 0.0 |
| 3 | 1101 | and_kg | 0.0 |
| 4 | 1426 | and_kg | 0.0 |

Figure 4.5 Meat deliveries data after unpivoting

Remove rows where amount = 0

As there are many rows now having an amount of 0, these will be removed. This is done by filtering the dataset and creating a subset of data only containing rows that has an amount greater than 0. The code line is shown below:

```
1 meat_deliveries_subset = meat_deliveries_melted[meat_deliveries_melted['amount'] > 0]
```

The data looks pretty much the same, but as shown in Figure 4.6 the first 5 rows printed no longer has amount of 0.

| | komnr | type | amount |
|-------------|--------------|-------------|---------------|
| 29 | 716 | and_kg | 51238.19 |
| 269 | 710 | and_kg | 10136.97 |
| 476 | 710 | and_kg | 16105.45 |
| 1609 | 712 | and_kg | 88126.59 |
| 2276 | 716 | and_kg | 12269.17 |

Figure 4.6 Meat deliveries data after removing rows where amount is 0

Split columns

The type of meat also contains the unit of the amount in the name. Since the unit is the same for all rows, the unit can be removed from the name. The column will be split and the rows will be renamed to only the type of meat. This is done by first splitting the different parts of the name by the underscore as shown in the code line below using the **split()** method (pandas, n.d.-x):


```
1 meat_deliveries_variables = meat_deliveries_subset['type'].str.split('_')
```

This creates a series where each row is a list of the two parts of the name, the meat type and the unit, as shown in Figure 4.7.

```
29          [and, kg]
269         [and, kg]
476         [and, kg]
1609        [and, kg]
2276        [and, kg]
...
686083     [ungku, kg]
686084     [ungku, kg]
686086     [ungku, kg]
686087     [ungku, kg]
686088     [ungku, kg]
Name: type, Length: 87601, dtype: object
```

Figure 4.7 Lists of parts of the "type" column of the meat deliveries data

Now, the next code lines first makes a copy of the previous subset of data using the **copy()** method (pandas, n.d.-e), drops the *type* column where the unit is still in the type name, add a new *type* column where only the type is added. Lastly, the newly added *type* column will be added to the end of the table. To keep the same order for all datasets, the columns are rearranged in the last line of code. All the code lines are shown below:

```
1 # Copy the old subset
2 meat_deliveries_subset_copy = meat_deliveries_subset.copy()
3 # Drop the old types
4 meat_deliveries_split = meat_deliveries_subset_copy.drop(['type'], axis=1)
5 # Add a new type column with the values of the splitted types
6 meat_deliveries_split['type'] = meat_deliveries_variables.str.get(0)
7 # Rearrange the order of the columns to match the other datasets
8 meat_deliveries_split = meat_deliveries_split[['komnr', 'type', 'amount']]
```

The data now looks as shown in Figure 4.8.

| | komnr | type | amount |
|--|--------------|-------------|---------------|
| | 29 | 716 and | 51238.19 |
| | 269 | 710 and | 10136.97 |
| | 476 | 710 and | 16105.45 |
| | 1609 | 712 and | 88126.59 |
| | 2276 | 716 and | 12269.17 |
| | 7989 | 710 and | 115180.90 |
| | 11507 | 123 and | 73974.89 |

Figure 4.8 A sample of the current Meat deliveries data. The unit has been removed from the types in the “type” column. **Simplify meat types**

The meat types are very specific and will not match the nutrients table that will be described later. Therefore, some of the categories should be added together into one broader category. An example is the category “sheep”, which is categorized as “sau”, “ungsau” and “vaer”. The difference is the age and sex of the sheep. For instance, “ungsau” is a young sheep, and “vaer” is a male sheep. These three categories could be given the broader category “sheep”. In addition, some of the meat types are not in the nutrients table at all and there is no substitute for it. For instance, horse and goat meat. The meat deliveries data also contains the amount of wool produced, which is not relevant in this case. These types will be removed from the data.

To do this, the new and broader categories of meat is first created by creating lists of the types that are to be considered the same category. Then the **fromkeys()** method (Python, n.d.-b) is used to create dictionaries from the lists. The **fromkeys()** method takes in the keys and value and returns a dictionary and it is used to create a dictionary where each element in the lists are set as keys and the specified value is set as value. The specified value is in this case the simplified types of meat. A new dictionary named *simplified_meat_types* can now be created, and all the dictionaries can be added to this. The code lines are shown below:

```

1 L1 = ['gris', 'purke', 'raane']
2 d1 = dict.fromkeys(L1, 'svin')
3 L2 = ['okse', 'ungokse', 'ku', 'ungku', 'kvige']
4 d2 = dict.fromkeys(L2, 'okse')
5 L3 = ['hons', 'hane']
6 d3 = dict.fromkeys(L3, 'høne')
7 L4 = ['lam', 'lam_villsau']
8 d4 = dict.fromkeys(L4, 'lam')
9 L5 = ['vaer', 'sau', 'ungsau']
10 d5 = dict.fromkeys(L5, 'sau')
11 L6 = ['kalv']
12 d6 = dict.fromkeys(L6, 'kalv')
13 L7 = ['kylling']
14 d7 = dict.fromkeys(L7, 'kylling')
15 L8 = ['kalkun']
16 d8 = dict.fromkeys(L8, 'kalkun')
17 L9 = ['and']
18 d9 = dict.fromkeys(L9, 'and')
19 L10 = ['gaas']
20 d10 = dict.fromkeys(L10, 'gås')
21
22 # Create a dictionary and add all dictionaries to it (d1 to d10)
23 simplified_meat_types = {**d1, **d2, **d3, **d4, **d5, **d6, **d7,
                          **d8, **d9, **d10}

```

The final dictionary now looks as shown in Figure 4.9.

```

{'gris': 'svin',
 'purke': 'svin',
 'raane': 'svin',
 'okse': 'okse',
 'ungokse': 'okse',
 'ku': 'okse',
 'ungku': 'okse',
 'kvige': 'okse',
 'hons': 'høne',
 'hane': 'høne',
 'lam': 'lam',
 'lam_villsau': 'lam',
 'vaer': 'sau',
 'sau': 'sau',
 'ungsau': 'sau',
 'kalv': 'kalv',
 'kylling': 'kylling',
 'kalkun': 'kalkun',
 'and': 'and',
 'gaas': 'gås'}

```

Figure 4.9 Dictionary of simplified meat types

The final dictionary can be used to map the simplified meat types to the initial meat types in the meat deliveries dataset. This is done using the **map()** method, which maps the

values according to a input mapping (pandas, n.d.-r). In this case, the values are mapped according to the dictionary. The line of code is shown below:

```
1 meat_deliveries_split['type'] = meat_deliveries_split['type'].map(simplified_meat_types)
```

The **map()** method will set all values that it cannot find in the dictionary as keys to NaN, which means that in this case wherever e.g. categories horse and goat are in the data, there will be a NaN value. Therefore, a subset of data is created from the previous dataset, where the data is filtered using the **notnull()** method (pandas, n.d.-s). The **notnull()** method detects existing values and returns a boolean object that indicates if the value is null or not. Non-missing values are mapped to True, while missing values (for instance NaN) is mapped to False. In this way, the subset of data will only contains the values that are not NaN. The code line is shown below:

```
1 meat_deliveries_subset = meat_deliveries_split[meat_deliveries_split['type'].notnull()]
```

Group by municipality ID and type of meat

To get the amount of each type of meat delivered for each municipality, the data is grouped by both municipality ID (komnr) and type of meat (type). The amount is then summarized per municipality and meat type using the aggregation function **sum** and pandas **agg()** method.

```
1 meat_deliveries_per_municipality = meat_deliveries_split_subset.groupby(['komnr', 'type'],
                                                                    as_index=False)
                                                                    .agg({'amount': 'sum'})
```

| | komnr | type | amount |
|-----------|--------------|-------------|---------------|
| 0 | 101 | kalv | 1053.7 |
| 1 | 101 | kylling | 1317918.0 |
| 2 | 101 | lam | 5463.6 |
| 3 | 101 | okse | 246022.2 |
| 4 | 101 | sau | 3086.4 |
| 5 | 101 | svin | 1129388.2 |
| 6 | 104 | kalv | 1211.0 |
| 7 | 104 | lam | 3137.8 |
| 8 | 104 | okse | 946.0 |
| 9 | 104 | sau | 386.6 |
| 10 | 105 | høne | 17332.8 |
| 11 | 105 | kalkun | 776190.9 |
| 12 | 105 | kalv | 8946.5 |
| 13 | 105 | kylling | 488286.1 |
| 14 | 105 | lam | 15183.3 |
| 15 | 105 | okse | 224471.9 |
| 16 | 105 | sau | 3202.7 |
| 17 | 105 | svin | 1092929.4 |

Figure 4.10 A sample of the final Meat deliveries data. All the necessary data preparation tasks have been performed, and the data now shows the amount of each meat type for each municipality.

4.1.4 Dairy Deliveries

The data file from *Felles Datakatalog* contains information about deliveries of cow and goat milk³. It contains columns for organization ID (orgnr), name of the owner of the organization, municipality ID and amount of cow milk and goat milk.

Data Discovery

Since the goal is to find the amount of each type of milk produced in each municipality, the necessary data preparation tasks are:

³ <https://data.norge.no/datasets/ce1f1dfc-704f-43c8-b133-51f5a20ee406>

- Drop irrelevant columns
- Unpivot data
- Remove null data
- Split column
- Group by municipality ID and type of milk

Data Preparation

Drop irrelevant columns

The organization ID and name of the owner is not relevant in this case, therefore these columns should be dropped. Columns are dropped from a pandas DataFrame using the **drop()** method, which drops the specified labels from rows or columns. The **axis** parameter is set to 1, which means it drops the columns with the specified labels in the inputted list.

The code line is shown below:

```
1 dairy_deliveries_dropped = dairy_deliveries_init.drop(['navn', 'orgnr'], axis=1)
```

The data now looks as shown in Figure 4.11.

| | komnr | kumelk_liter | geitemelk_liter |
|----------|--------------|---------------------|------------------------|
| 0 | 101 | 92957 | 0 |
| 1 | 101 | 533955 | 0 |
| 2 | 105 | 500496 | 0 |
| 3 | 105 | 147071 | 0 |
| 4 | 105 | 203632 | 0 |

Figure 4.11 Dairy deliveries data after dropping irrelevant columns

Unpivot data

To get the dairy deliveries data in a similar format and structure as the meat deliveries data, it is unpivoted using the **melt()** method. The municipality ID (komnr) columns is kept as is, and the cow and goat milk columns are unpivoted. The new columns are named “type” and “amount”. The code line is shown below:

```
1 dairy_deliveries_melted = pd.melt(dairy_deliveries_dropped,
                                   id_vars=['komnr'],
                                   var_name='type',
                                   value_name='amount')
```

The unpivoted data looks as shown in Figure 4.12.

| | komnr | type | amount |
|---|-------|--------------|--------|
| 0 | 101 | kumelk_liter | 92957 |
| 1 | 101 | kumelk_liter | 533955 |
| 2 | 105 | kumelk_liter | 500496 |
| 3 | 105 | kumelk_liter | 147071 |
| 4 | 105 | kumelk_liter | 203632 |

Figure 4.12 Dairy deliveries data after unpivoting

Remove null data

Since most farms only produce one type of milk, there will be a lot of rows having an amount of 0. These rows are redundant and can be removed. This is done in the following line of code:

```
1 dairy_deliveries_subset = dairy_deliveries_melted[dairy_deliveries_melted['amount'] > 0.0]
```

Split column

Similarly as in the meat deliveries dataset, the type of milk also contains the unit. The unit is liter for all entries in the data table, so it can be removed from this dataset as well. This is done in the same way as before and will not be described again in detail. See section 4.1.3 for details on how this was done for the meat deliveries data.

The result after splitting the column and setting the type to the first part, is shown in Figure 4.13.

| | komnr | type | amount |
|----------|--------------|-------------|---------------|
| 0 | 101 | kumelk | 92957 |
| 1 | 101 | kumelk | 533955 |
| 2 | 105 | kumelk | 500496 |
| 3 | 105 | kumelk | 147071 |
| 4 | 105 | kumelk | 203632 |

Figure 4.13 Dairy deliveries data after splitting the "type" column

Group by municipality ID and grain type

The dairy delivery data is grouped in the same way as the meat delivery data, by using the **groupby()** method. It is grouped by both municipality ID (komnr) and type of milk (type). The code line is shown below:

```
1 dairy_deliveries_per_municipality = dairy_deliveries_split.groupby(['komnr', 'type'],
                                                                    as_index=False)
                                                                    .agg({'amount': 'sum'})
```

The dairy deliveries data now looks as shown in Figure 4.14.

| | komnr | type | amount |
|----------|--------------|-------------|---------------|
| 0 | 101 | kumelk | 3612980 |
| 1 | 105 | kumelk | 4743006 |
| 2 | 106 | kumelk | 2162467 |
| 3 | 118 | kumelk | 623344 |
| 4 | 119 | kumelk | 1882796 |

Figure 4.14 The final Dairy deliveries data. All data preparation steps have now been performed.

4.1.5 Egg Deliveries

This data file contains information about the amount of eggs delivered from each organization⁴. It contains the columns organization ID (orgnr), name of organization owner (navn), municipality ID (komnr), and the amount of eggs in kilograms (egg_kg).

Data Discovery

To find the amount of eggs delivered in each municipality, the necessary data preparation tasks are:

- Drop irrelevant columns
- Check for null values
- Add a “type” column
- Rename column
- Group by municipality ID

Data Preparation

Drop irrelevant columns

Again using the **drop()** method, the organization ID (orgnr) and owner (navn) is dropped. The code line is shown below:

```
1 egg_deliveries_dropped = egg_deliveries_init.drop(['orgnr', 'navn'], axis=1)
```

The data now only contains the columns municipality ID (komnr) an amount of eggs (egg_kg), as shown in Figure 4.15.

⁴ <https://data.norge.no/datasets/d58a8898-6162-43cd-9df8-5ae0b5fa9ebc>

| | komnr | egg_kg |
|----------|--------------|---------------|
| 0 | 1120 | 131770 |
| 1 | 5004 | 129940 |
| 2 | 1120 | 133703 |
| 3 | 1445 | 121256 |
| 4 | 1432 | 8052 |

Figure 4.15 Egg deliveries data after dropping irrelevant rows

Check for null values

There are several ways to check for null values. One way is using the **info()** method (pandas, n.d.-j). This will output how many non-null values are in each column. If the number of non-null values are not equal to the total number of values in the column, there are null values in the column. Another way is using the **isnull()** (pandas, n.d.-k) and **any()** (pandas, n.d.-d) methods can be used in combination to check for null values and filter out any null values from the data. The **any()** method returns whether any element is True. In this case over axis 1, which is the columns. This is done as shown in the code line below:

```
1 null_data = egg_deliveries_dropped[egg_deliveries_dropped.isnull().any(axis=1)]
```

Since there are no null values in this dataset, the output of *null_data* is an empty data table, as shown in Figure 4.16.

```
1 null_data
Out[59]:
komnr egg_kg
```

Figure 4.16 Null values in the egg deliveries dataset

Add a new column

To match the other datasets another column is added named “type”. Since this data only contains amount of egg, the type will be set to “egg” for all rows. The code line is shown below:

```
1 egg_deliveries_dropped['type'] = «egg»
```

The data now looks as shown in Figure 4.17.

| | komnr | egg_kg | type |
|----------|--------------|---------------|-------------|
| 0 | 1120 | 131770 | egg |
| 1 | 5004 | 129940 | egg |
| 2 | 1120 | 133703 | egg |
| 3 | 1445 | 121256 | egg |
| 4 | 1432 | 8052 | egg |

Figure 4.17 Egg deliveries data after adding the “type” column

Rename column

The “egg_kg” column should be renamed to match the other datasets. This column is renamed to “amount” using the **rename()** method (pandas, n.d.-m), which takes in a dictionary containing the columns to rename and the new name to apply. The line of code is shown below:

```
1 egg_deliveries = egg_deliveries_dropped.rename(columns={'egg_kg': 'amount'})
```

The data now looks as shown in Figure 4.18.

| | komnr | amount | type |
|----------|--------------|---------------|-------------|
| 0 | 1120 | 131770 | egg |
| 1 | 5004 | 129940 | egg |
| 2 | 1120 | 133703 | egg |
| 3 | 1445 | 121256 | egg |
| 4 | 1432 | 8052 | egg |

Figure 4.18 Egg deliveries data after renaming the "egg_kg" column to "amount"

Rearrange columns

To get the egg deliveries data to match the other dataset, the order of the columns is rearranged by extracting the columns in the desired order and saving them to the dataset *egg_deliveries*. This is done in the following line of code:

```
1 egg_deliveries = egg_deliveries[['komnr', 'type', 'amount']]
```

The columns are now in the desired order, as shown in Figure 4.19.

| | komnr | type | amount |
|----------|--------------|-------------|---------------|
| 0 | 1120 | egg | 131770 |
| 1 | 5004 | egg | 129940 |
| 2 | 1120 | egg | 133703 |
| 3 | 1445 | egg | 121256 |
| 4 | 1432 | egg | 8052 |

Figure 4.19 Egg deliveries data after rearranging the columns

Group by municipality ID

Lastly, the egg deliveries data needs to be grouped by municipality ID and the amount is summarized. This is done in the code line below:

```
1 egg_deliveries_per_municipality = egg_deliveries.groupby(['komnr', 'type'],
                                                         as_index=False)
                                                         .agg({'amount': 'sum'})
```

This results in the dataset only containing one row of the amount of egg produced for each municipality ID. A sample of the data is shown in Figure 4.20, where you can see that there is only one row for each municipality ID (komnr).

| | komnr | type | amount |
|----------|--------------|-------------|---------------|
| 0 | 101 | egg | 246774 |
| 1 | 105 | egg | 867467 |
| 2 | 106 | egg | 107854 |
| 3 | 118 | egg | 146253 |
| 4 | 119 | egg | 291154 |

Figure 4.20 Final Egg deliveries data after preparation

4.1.6 Grain Deliveries

This data file contains information about the deliveries of different types of grain⁵. It contains the columns organization ID (orgnr), name of owner of organization (navn) and municipality ID (komnr). In addition, each type of grain is divided into three categories: animal feed, food and seeds. Each of the types of grain, and each category of the type, has its own column.

Data Discovery

The data has a wide format and each grain type has its own column, similar to the meat and egg deliveries data. A lot of these categories are irrelevant or not in the nutrients table and should therefore be removed. All the necessary data preparation tasks are listed below:

- Drop irrelevant columns
 - Unpivot data
 - Remove null data
 - Split column
-

⁵ <https://data.norge.no/datasets/2cda2089-8629-4098-835c-59b473e4a785>

- Group by municipality ID

Data Preparation

Drop irrelevant columns

The only relevant columns in the grain deliveries dataset are the municipality ID (komnr) and amount of each type of food. Since there are some columns of animal feed and seeds, these can be removed first. This is done by getting the column labels, converting them to a list using the **tolist()** method (pandas, n.d.-q), and storing them in a list named *grain_deliveries_columns* as shown in the code line below:

```
1 grain_deliveries_cols = grain_deliveries_init.columns.tolist()
```

The list now looks as shown in Figure 4.21.

```
['orgnr',  
'navn',  
'komnr',  
'bygg_for_kg',  
'bygg_saakorn_kg',  
'bygg_mat_kg',  
'erter_for_kg',  
'erter_mat_kg',  
'erter_saakorn_kg',  
'havre_for_kg',  
'havre_saakorn_kg',  
'hvete_for_kg',  
'hvete_mat_kg',  
'hvete_saakorn_kg',  
'oljefro_kg',  
'oljefro_saakorn_kg',  
'rug_for_kg',  
'rug_mat_kg',  
'rug_saakorn_kg',  
'rughvete_for_kg',  
'rughvete_saakorn_kg']
```

Figure 4.21 List of column labels of the grain deliveries data

To remove the columns that are not relevant, the **remove()** method (Python, n.d.-a) is used. The columns that are needed are the municipality ID and all types of grain that is food. This means the “komnr” and all columns containing the word “mat” (food in Norwegian). The irrelevant column labels are removed from the list using a while loop that loops as long as *i* is less than the length of the list of columns labels. An If statement is used

to check if the column label does not consist of the substring “mat” and is not equal to “komnr”. These elements in the list are removed. Else, the *i* is increased by 1. The *i* will increase until each element has been checked and the irrelevant column labels are removed.

The code lines are shown below:

```
1 # Extract only column headers containing the substring "mat"
2 substring = "mat"
3 i = 0
4 while i < len(grain_deliveries_cols):
5     if (substring not in grain_deliveries_cols[i] & (grain_deliveries_cols[i] != "komnr")):
6         print(f"{grain_deliveries_cols[i]}" does not contain the substring "mat")
7         grain_deliveries_cols.remove(grain_deliveries_cols[i])
8     else:
9         i += 1
10    print(f"{grain_deliveries_cols[i]}" does contain the substring "mat")
```

The list now looks as shown in Figure 4.22.

```
1 grain_deliveries_cols
['komnr', 'bygg_mat_kg', 'erter_mat_kg', 'hvete_mat_kg',
'rug_mat_kg']
```

Figure 4.22 Relevant columns of the grain deliveries data

Now, the relevant columns can be extracted from the dataset. This is done in the code line below:

```
1 grain_deliveries_dropped = grain_deliveries_init[grain_deliveries_cols]
```

A sample of the current data can be seen in Figure 4.23.

| | komnr | bygg_mat_kg | erter_mat_kg | hvete_mat_kg | rug_mat_kg |
|----------|--------------|--------------------|---------------------|---------------------|-------------------|
| 0 | 5038 | 0 | 0 | 0 | 0 |
| 1 | 412 | 0 | 0 | 0 | 0 |
| 2 | 124 | 0 | 0 | 0 | 0 |
| 3 | 624 | 0 | 0 | 0 | 0 |
| 4 | 712 | 0 | 0 | 0 | 0 |

Figure 4.23 Grain deliveries data after removing irrelevant columns

Unpivot

The data is unpivoted to match the other datasets, creating a column “type” containing the type of grain and a column “amount” containing the amount of grain in kilograms. The code line is shown below:

```
1 grain_deliveries_melted = pd.melt(grain_deliveries_dropped,
                                   id_vars=['komnr'],
                                   var_name='type',
                                   value_name='amount')
```

The unpivoted data is shown in Figure 4.24.

| | komnr | type | amount |
|----------|--------------|-------------|---------------|
| 0 | 5038 | bygg_mat_kg | 0 |
| 1 | 412 | bygg_mat_kg | 0 |
| 2 | 124 | bygg_mat_kg | 0 |
| 3 | 624 | bygg_mat_kg | 0 |
| 4 | 712 | bygg_mat_kg | 0 |

Figure 4.24 Grain deliveries data after unpivoting

Remove null data

There are now a lot of rows with amount of 0. These are removed in the code line below:

```
1 grain_deliveries_subset = grain_deliveries_melted[grain_deliveries_melted['amount'] > 0]
```


Split column

The type is formatted like this: [type of grain] _ [mat] _ [kg]. All rows are grain used for food and are in kilograms, so these parts of the name can be removed. The columns strings are split on underscore and the first part of the string is assigned as the value of the rows of the “type” column. The code line is shown below:

```
1 grain_deliveries_variables = grain_deliveries_subset['type'].str.split('_')
2 # Copy the old subset
3 grain_deliveries_subset_copy = grain_deliveries_subset.copy()
4 # Drop the old types
5 grain_deliveries_split = grain_deliveries_subset_copy.drop(['type'], axis=1)
6 # Add a new type column with the values of the splitted types
7 grain_deliveries_split['type'] = grain_deliveries_variables.str.get(0)
8 # Rearrange the order of the columns to match the other datasets
9 grain_deliveries_split = grain_deliveries_split[['komnr', 'type', 'amount']]
```

The data now looks as shown in Figure 4.25.

| | komnr | type | amount |
|--------------|-------|-------|--------|
| 130 | 5037 | bygg | 90000 |
| 19503 | 231 | hvete | 54771 |
| 19506 | 417 | hvete | 51845 |
| 19512 | 119 | hvete | 16718 |
| 19516 | 214 | hvete | 53754 |

Figure 4.25 Grain deliveries data after splitting the "type" column

Group by municipality ID and type of grain

To get the amount of each type of grain per municipality, the data needs to be grouped by municipality ID (komnr) and type of grain (type). This is done in the code line below:

```
1 grain_deliveries_per_municipality = grain_deliveries_split.groupby(['komnr', 'type'],
                                                                    as_index=False)
                                                                    .agg({'amount': 'sum'})
```

A sample of the grouped data is shown in Figure 4.26.

| | komnr | type | amount |
|---|-------|-------|---------|
| 0 | 101 | hvete | 3901835 |
| 1 | 105 | hvete | 3797783 |
| 2 | 105 | rug | 15895 |
| 3 | 106 | hvete | 2945609 |
| 4 | 106 | rug | 5265 |

Figure 4.26 A sample of the current Grain deliveries data. All the necessary data preparation tasks have now been performed

4.1.7 Nutrients Table

This data file contains the nutritional contents of different types of food⁶. As this file has 116 columns, they won't all be described here, but there are for instance columns for the type of food, an ID for each type of food, and one column for each type of nutrient.

Data Discovery

This is a very messy dataset with many empty rows of data and one row dedicated to the title. It also has comments and in some cells the unit of the value is given. In Figure 4.27 you can see a small sample of the initial nutrients table data, where only 10 out of 116 columns and 5 out of 2152 rows are shown. There is a lot of NaN values, comments and generally irrelevant data that should be removed.

| | Unnamed: 0 | Den norske matvaretabellen 2021 | Unnamed: 2 | Unnamed: 3 | Unnamed: 4 | Unnamed: 5 | Unnamed: 6 | Unnamed: 7 | Unnamed: 8 | Unnamed: 9 |
|---|------------|---|--------------|------------|------------|------------|------------|------------|--------------|------------|
| 0 | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN |
| 1 | MatvareID | Matvare | Spiselig del | Ref | Vann | Ref | Kilojoule | Ref | Kilokalorier | Ref |
| 2 | NaN | Forkortelser: M = manglende verdi. Ref = refer... | % | NaN | g | NaN | kJ | NaN | kcal | NaN |
| 3 | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN |
| 4 | 1 | Melk og melkeprodukter | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN |

Figure 4.27 A small sample of the initial nutrient table data

The necessary steps to get the desired data format and structure is:

⁶ <https://data.norge.no/datasets/9d082918-e3d4-4ae2-8efd-e7d025dfd52d>

- Change column header
- Drop redundant rows
- Extract only relevant columns
- Split column

Data Preparation

Change column header

Since the header row only contains a title in one of the columns and the rest of the columns are named “Unnamed: 0”, “Unnamed: 1”, and so on, there is no point in keeping this row. The first row only contains NaN values, as seen in Figure 4.27, and this can also be removed. The actual column headers are in row 2 (with index 1), so this row should be set as column header. This is done by using the pandas indexer `.iloc[]` (pandas, n.d.-i), which gets the row(s) of the inputted index. In this case, the row with index 1 should be set as column header. This is done by setting the column labels equal to the row as shown in the code line below:

```
1 nutrients_table_init.columns = nutrients_table_init.iloc[1]
```

A sample of the current data is shown in Figure 4.28.

| 1 | MatvareID | Matvare | Spiselig del | Ref | Vann | Ref | Kilojoule | Ref | Kilokalorier | Ref |
|---|-----------|---|--------------|-----|------|-----|-----------|-----|--------------|-----|
| 0 | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN |
| 1 | MatvareID | Matvare | Spiselig del | Ref | Vann | Ref | Kilojoule | Ref | Kilokalorier | Ref |
| 2 | NaN | Forkortelser: M = manglende verdi. Ref = refer... | % | NaN | g | NaN | KJ | NaN | kcal | NaN |
| 3 | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN |
| 4 | 1 | Melk og melkeprodukter | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN |

Figure 4.28 A sample of the nutrients table data with the new column headers

Drop redundant rows

There is still a lot of redundant rows. As seen in Figure 4.28, the first row (with index 0) consist of only NaN values. The second row (with index 1) is set as the column header and

should be removed. The third row (with index 2) has a comment in the second column and the rest of the columns are units and NaN values. The foods are also divided into categories, such as “Milk and dairy products”. There are rows that only contain the category in the “Matvare” columns, but the rest of the columns consists of NaN values. None of the above-mentioned rows are relevant in this case and should be removed. One thing to note is that the comment tells me that any missing values are denoted “M”, which is important to know in order to remove this data later. Rows of index 0-3 are removed in the code line below:

```
1 # Drop rows containing redundant information or null values
2 nutrients_table_dropped = nutrients_table_init.drop([0, 1, 2, 3])
```

This removed the first four rows. Now, all the rows containing the categories of food can be removed. Since these rows only has a value in the “Matvare” column, it will have a NaN value in the rest of the columns. Choosing one of these columns, a subset of data can be made by filtering out the rows where the column contains a NaN value. In this case, the “Spiselig del” column are chosen. Filtering the rows by using the **notnull()** method results in a subset of data where all values are non-missing values (not NaN values). The code line is shown below:

```
1 # Remove food category rows
2 nutrients_table_subset = nutrients_table_dropped[nutrients_table_dropped['Spiselig del']
                                                    .notnull()]
```

A sample of the current data is shown in Figure 4.29.

| 1 | MatvareID | Matvare | Spiselig del | Ref | Vann | Ref | Kilojoule | Ref | Kilokalorier | Ref |
|----|-----------|----------------------------------|--------------|-----|------|--------|-----------|--------|--------------|--------|
| 6 | 01.013 | Geitmelk, langtidsholdbar | 100 | 0 | 89 | MI0142 | 286 | MI0114 | 69 | MI0115 |
| 7 | 01.272 | Helmelk, 3,5 % fett, laktosefri | 100 | 0 | 90 | MI0142 | 249 | MI0114 | 60 | MI0115 |
| 8 | 01.001 | Helmelk, 3,5 % fett, Tine | 100 | 0 | 89 | MI0142 | 264 | MI0114 | 63 | MI0115 |
| 9 | 01.235 | Helmelk, 3,9 % fett, Q-meieriene | 100 | 0 | 88 | 132 | 279 | MI0114 | 67 | MI0115 |
| 10 | 01.283 | Helmelk, 4,1 % fett, økologisk | 100 | 0 | 88 | MI0142 | 279 | MI0114 | 67 | MI0115 |

Figure 4.29 Nutrients table data after removing redundant rows

Extract relevant columns

In order to get the kilocalories of each type of food, the food type column “Matvare” and the kilocalories column named “Kilokalorier” should be extracted from the dataset. This is done in the code line below:

```
1 nutrients_table = nutrients_table_subset[['Matvare', 'Kilokalorier']]
```

A sample of the current data is shown in Figure 4.30.

| 1 | Matvare | Kilokalorier |
|----|----------------------------------|--------------|
| 6 | Geitmelk, langtidsholdbar | 69 |
| 7 | Helmelk, 3,5 % fett, laktosefri | 60 |
| 8 | Helmelk, 3,5 % fett, Tine | 63 |
| 9 | Helmelk, 3,9 % fett, Q-meieriene | 67 |
| 10 | Helmelk, 4,1 % fett, økologisk | 67 |

Figure 4.30 Nutrients table data after extracting only relevant columns

Split column

The type of food in the column “Matvare” not only provides the type of food, but also some additional information about the food type, such as whether it is cooked or not, how many percent of fat it contains, the manufacturer of the food, etc. This information is not relevant in this case. The rest of the datasets of food production data are not very specific. For instance, the meat deliveries data only says pig meat, and does not specify

which part of the pig the meat is from. In the nutrients table, this information is included. However, as the information is not consistent in both datasets, there is no point in considering which part of the pig the meat is from. Thus, the type column (Matvare) will be split on comma and space (“, ”) and the first string will be the value that is assigned to the type column.

Splitting the string on underscore is done in the code line below:

```
1 # Split the string on underscore
2 nutrients_table_variables = nutrients_table['Matvare'].str.split(', ')
```

This gives a Series of lists containing the parts of the string, as shown in Figure 4.31.

```
6           [Geitmelk, langtidsholdbar]
7       [Helmelk, 3,5 % fett, laktosefri]
8           [Helmelk, 3,5 % fett, Tine]
9       [Helmelk, 3,9 % fett, Q-meieriene]
10          [Helmelk, 4,1 % fett, økologisk]
...
2142          [Sinlac spesialgrøt, pulver, Nestlé]
2143          [Sinlac spesialgrøt, spiseklar, Nestlé]
2144          [Tilskuddsblanding, fra 6 mnd, drikkeklar]
2145 [Tilskuddsblanding, fra 6 mnd, ferdig utblandet]
2146          [Tilskuddsblanding, fra 6 mnd, pulver]
Name: Matvare, Length: 1983, dtype: object
```

Figure 4.31 Lists of substrings of the type of food from the Nutrients Table data.

Next, a copy of the previous dataset is made, the old types of food are dropped and a new column with the first part of the string is created. This is shown in the code lines below:

```
1 # Copy the old subset
2 nutrients_table_copy = nutrients_table.copy()
3 # Drop the old types
4 nutrients_table_split = nutrients_table_copy.drop(['Matvare'], axis=1)
5 # Add a new type column with the values of the splitted types
6 nutrients_table_split['Matvare'] = nutrients_table_variables.str.get(0)
```

The data now looks as shown in Figure 4.32.

| 1 | Kilokalorier | Matvare |
|----|--------------|----------|
| 6 | 69 | Geitmelk |
| 7 | 60 | Helmelk |
| 8 | 63 | Helmelk |
| 9 | 67 | Helmelk |
| 10 | 67 | Helmelk |

Figure 4.32 Nutrients table data after splitting the "Matvare" column. Only the first part of the string has been assigned as the value of the column

Rename columns

To get this dataset to match the rest, the columns are renamed to "type" and "kcal". This is done using the **rename()** method. The code line is shown below:

```
1 nutrients_table_renamed = nutrients_table_split.rename(columns={
    'Matvare': 'type',
    'Kilokalorier': 'kcal'
})
```

The data now looks as shown in Figure 4.33.

| 1 | kcal | type |
|----|------|----------|
| 6 | 69 | Geitmelk |
| 7 | 60 | Helmelk |
| 8 | 63 | Helmelk |
| 9 | 67 | Helmelk |
| 10 | 67 | Helmelk |

Figure 4.33 Nutrients table data after renaming the columns to "type" and "kcal"

Remove missing values

As previously mentioned, a comment in the data said that missing values were denoted "M". A subset of the data should therefore be made, filtering out the rows where kilocalories have a "M" as value. This is done in the code line below:

```
1 nutrients_table_reduced = nutrients_table_renamed[nutrients_table_renamed['kcal'] != 'M']
```

Using the **shape** attribute the number of rows of the datasets can be checked to see if it has been reduced. As seen in Figure 4.34, the reduced nutrients table data has 1980 rows and the old dataset had 1983. This means three rows of data were removed from the nutrients table data.

```
1 nutrients_table_reduced.shape
(1980, 2)

1 nutrients_table_renamed.shape
(1983, 2)
```

Figure 4.34 Comparing the size of the dataframes after removing missing values.

Group by type of food

Previously, the “Matvare” column (now named “type”) was split, and the first part of the string was set as the value of the column. This means there will be a lot of rows with the same food type. Previously there were more information about the food type that specified the difference between these types, however, as this additional information is now removed these types can be grouped together. Since these types in the initial datasets were different, they all have different values in the kilocalories column. As the food production datasets are not specific enough to distinguish between these types, the kilocalories values will be aggregated for each type, returning the mean of the kilocalories for each type of food. First the column data type has to be changed to dtype *float*. This is done in the code line below:

```
1 # Change data type (dtype) of the kilocalories column
2 nutrients_table_numeric = nutrients_table_reduced.astype({'kcal': float})
```

Then the data can be grouped by the food type as shown in the code line below:

```
1 # Group by type of food
2 nutrients_table_grouped = nutrients_table_numeric.groupby(['type'],
                                                            as_index=False)
                                                            .agg({'kcal': 'mean'})
```

This further reduces the data by more than 1,000 rows, as shown in Figure 4.35.


```
1 nutrients_table_grouped.shape
(942, 2)
```

Figure 4.35 Checking the size of the Nutrients Table data after grouping the data.

Changing letter case

The food types in the nutrients table data have a capitalized first letter. This does not match the other datasets, and the letter case will therefore be changed. This is done by first creating a copy of the previous dataset as shown in the code line below:

```
1 kcal_of_foods_copy = nutrients_table_grouped.copy()
```

Then the “type” column is dropped from the copied data as shown in the code line below:

```
1 kcal_of_foods = kcal_of_foods_copy.drop(['type'], axis=1)
```

Next, a new “type” column is added and set equal to the “type” column of the previous nutrients table data, applying the **lower()** method to convert the strings to lowercase. This is done in the code line below:

```
1 kcal_of_foods['type'] = nutrients_table_grouped['type'].str.lower()
```

This results in the data shown in Figure 4.36.

| | kcal | type |
|---|-------|--------------|
| 0 | 310.0 | adzukibønner |
| 1 | 11.0 | agurk |
| 2 | 37.0 | agurker |
| 3 | 23.0 | agurksalat |
| 4 | 728.0 | aioli |

Figure 4.36 Nutrients table data after performing all data preparation tasks

4.1.8 Data Mapping and Integration

The datasets have now been individually prepared for integration. The types of datasets that are available are:

1. One dataset containing information about area of farmland of each municipality
2. Four datasets containing information about the amount of food produced in each municipality
3. One dataset containing information about how many kilocalories each type of food contains

In order to answer the question from the use case, the data needs to be integrated.

The necessary steps in order to integrate the data is:

- The food production datasets need to be concatenated to form one large datasets of all the produces food of each municipality
- The food types of the nutrients table data and the food production data needs to be mapped
- Do the final calculation

Concatenating the Food Production datasets

The food production datasets are listed below:

- *meat_deliveries_per_municipality*
- *dairy_deliveries_per_municipality*
- *egg_deliveries_per_municipality*
- *grain_deliveries_per_municipality*

These datasets can now be concatenated using the **concat()** method (pandas, n.d.-b).

This method concatenates pandas objects along a particular axis. In this case, the axis is not set in the code line, which means it will be set to the default which is 0 or index. The code line is shown below:

```
1 food_produced_per_municipality = pd.concat([meat_deliveries_per_municipality,
                                             dairy_deliveries_per_municipality,
                                             egg_deliveries_per_municipality,
                                             grain_deliveries_per_municipality])
```

The **tail()** method (pandas, n.d.-p) is used to output a sample of the data and by default returns the five last rows of the data frame. As shown in Figure 4.37 and Figure 4.38, the five first and last rows of data in the concatenated food production data are meat types and grain types, respectively. This is consistent with the code line above, where the meat deliveries data was in the first element of the list given to the **concat()** method, and the grain deliveries data was the last element of this list.

| | komnr | type | amount |
|----------|--------------|-------------|---------------|
| 0 | 101 | kalv | 1053.7 |
| 1 | 101 | kylling | 1317918.0 |
| 2 | 101 | lam | 5463.6 |
| 3 | 101 | okse | 246022.2 |
| 4 | 101 | sau | 3086.4 |

Figure 4.37 First five rows of concatenated food production data

| | komnr | type | amount |
|-------------|--------------|-------------|---------------|
| 3437 | 5037 | bygg | 90000.0 |
| 3438 | 5037 | hvete | 192099.0 |
| 3439 | 5038 | hvete | 21757.0 |
| 3440 | 5053 | hvete | 103184.0 |
| 3441 | 5054 | hvete | 8575.0 |

Figure 4.38 Last five rows of the food production data

Mapping food types of the Nutrients Table and Food Production dataset

In order to integrate the nutrients table data and food production data, the food types in both the datasets needs to be mapped. This is done by first getting the unique food types from the food production data by using the **unique()** method (pandas, n.d.-y), which returns all unique values in a Series. The **tolist()** method is also used to create a list of the unique values. This is done in the code line below:

```
1 unique_food_types = food_produced_per_municipality['type'].unique().tolist()
```

After some exploration using a python library named thefuzz (seatgeek, n.d.) to do fuzzy matching between the food types in the two datasets, a few food types did not get a high enough matching score and therefore had to be changed manually before the rest of the types was matched using fuzzy matching. The types of food that had to be manually mapped are three types of flour, where in the nutrients table it said the flour type and “flour” at the end. An example is “hvetemel”, which is wheat flour. In the nutrients table, the food type is “hvetemel” (wheat flour in English), but in the food production data the food type is only “hvete” (wheat in English). To solve this, these types of food is added to the list of unique food types from the food production data. This way, the food types are written in

the exact same way in both datasets. This shouldn't be necessary when doing fuzzy matching, but as a few characters was missing from the type in the food production dataset, the threshold set to select a match had to be set too low. When the threshold is too low, some of the other types got several matches that were not a good match at all. After the food types are added to the list, the food types that could not be fuzzy matched has to be removed from the list. This is done by creating a list with the food types, iterating over the list using a for loop and checking whether the element is in the list of unique food types or not using an if statement. If the element is in the list of unique food types, it is removed using the **remove()** method. The code lines are shown below:

```
1 unique_food_types += ['hvetemel', 'rugmel', 'byggmel', 'melk', 'okserull']
2 x = ['hvete', 'rug', 'bygg', 'kumelk', 'okse']
3 for f in x:
4     if f in unique_food_types:
5         unique_food_types.remove(f)
```

The list now contains the food types that match the nutrients table, and the ones that could not be fuzzy matched are removed. In total, the food production data now contains 16 unique food types, as shown in Figure 4.39.

```
['kalv',
 'kylling',
 'lam',
 'sau',
 'svin',
 'høne',
 'kalkun',
 'and',
 'gås',
 'geitemelk',
 'egg',
 'hvetemel',
 'rugmel',
 'byggmel',
 'melk',
 'okserull']
```

Figure 4.39 List of unique food types in the food production data

A list of unique values is created for the nutrients table in the same way as shown in the code line below:

```
1 nutrients_types_unique = kcal_of_foods['type'].unique().tolist()
```

Using the `len()` method the length of the list can be found. As seen in Figure 4.40, the nutrients table data has 942 types of food.

```
1 len(nutrients_types_unique)
942
```

Figure 4.40 Finding the length of the list of unique food types

To map the food production data's food types to the matching nutrients table data food types, fuzzy matching is used. The Python library *thefuzz* is imported in the line below:

```
1 from thefuzz import fuzz
```

An empty dictionary is created first. Then the matching food types are found using a nested for loop to check two elements, one from each list of food types. If the two elements gets a score that is greater than 90, the food type from the nutrients table is set as key and the food type from the food production dataset is set as value in the dictionary. The code lines are shown below:

```
1 # Fuzzy matching food types
2 food_types = {}
3 for f1 in unique_food_types:
4     for f2 in nutrients_types_unique:
5         if fuzz.ratio(f1, f2) > 90:
6             food_types[f2] = f1
```

This resulted in the dictionary shown in Figure 4.41.

```
{'kalv': 'kalv',  
'kylling': 'kylling',  
'lam': 'lam',  
'sau': 'sau',  
'svin': 'svin',  
'høne': 'høne',  
'kalkun': 'kalkun',  
'and': 'and',  
'gås': 'gås',  
'geitemelk': 'geitemelk',  
'egg': 'egg',  
'hvetemel': 'hvetemel',  
'rugmel': 'rugmel',  
'byggmel': 'byggmel',  
'melk': 'melk',  
'okserull': 'okserull'}
```

Figure 4.41 Dictionary containing the fuzzy matched food types

Now, the food types in the nutrients table that is in the dictionary can be replaced by the food type from the food production data, such that food types match between the datasets. This is done in the code line below:

```
1 kcal_of_foods_mapped = kcal_of_foods.replace({'type': food_types})
```

As previously mentioned, the nutrients table data has 942 unique food types. The only ones that are relevant in this case, are the 16 types that are also in the food production data. Therefore, the rest of the 926 food types in the nutrients table data can be removed. This is done in the code line below:

```
1 # Remove irrelevant food types  
2 kcal_of_foods_reduced = kcal_of_foods_mapped.loc[kcal_of_foods_mapped['type']  
                                                    .isin(unique_food_types)]
```

Lastly, the column's order is rearranged in the code line below:

```
1 # Rearrange the order of columns  
2 kcal_of_foods_final = kcal_of_foods_reduced[['type', 'kcal']]
```

As shown in Figure 4.42, the nutrients table now seems to have the desired food types.

| | type | kcal |
|-----|-----------|------------|
| 12 | and | 307.750000 |
| 98 | byggmel | 334.000000 |
| 154 | egg | 166.333333 |
| 226 | geitemelk | 69.000000 |
| 265 | gås | 333.750000 |

Figure 4.42 Nutrients table data after matching the food types

Since the index of the rows are now all messed up, the **reset_index()** method (pandas, n.d.-n) can be used to reset the indexes. Setting the drop parameter to True makes sure the current index is not added as a new column. The code line is shown below:

```
1 kcal_of_foods_final.reset_index(drop=True)
```

This results in the data frame shown in Figure 4.43.

| | type | kcal |
|----|-----------|------------|
| 0 | and | 307.750000 |
| 1 | byggmel | 334.000000 |
| 2 | egg | 166.333333 |
| 3 | geitemelk | 69.000000 |
| 4 | gås | 333.750000 |
| 5 | hvetemel | 338.666667 |
| 6 | høne | 178.800000 |
| 7 | kalkun | 134.666667 |
| 8 | kalv | 112.000000 |
| 9 | kylling | 155.904762 |
| 10 | lam | 234.000000 |
| 11 | melk | 45.000000 |
| 12 | okserull | 166.000000 |
| 13 | rugmel | 323.000000 |
| 14 | sau | 371.000000 |
| 15 | svin | 267.612903 |

Figure 4.43 Nutrients table data after mapping the food types

The final processing step is to merge the three current datasets. This is done using the **merge()** method (pandas, n.d.-t), which merges the data frames with a database-style

join. The code line that merges the food production data and the nutrients table data is shown below:

```
1 final_data = pd.merge(food_produced_per_municipality, kcal_of_foods_final)
```

This data is then merged with the area of farmland data in the code line below:

```
1 final_data2 = pd.merge(final_data, area_of_farmland_per_municipality)
```

Now, all the data is integrated into one final data frame and the final calculation can be done to answer the question.

Final calculation

During the exploration of how to calculate the answer for the question, the units of some of the columns needed to be changed. The kilocalories are given in kilocalories per hectogram, instead it should be in kilocalories per kilograms, as the food amount is given in kilograms. The area of farmland is given in square kilometers and should be in square meters. The code lines doing the conversions is shown below:

```
1 # Converting kcal/hg to kcal/kg
2 final_data3 = final_data2.drop('kcal', axis=1)
3 final_data3['kcal'] = final_data2['kcal'].div(0.1)
4 # Convert area of farmland from km^2 to m^2
5 final_data3['jordbruksareal'] = final_data2['jordbruksareal'].mul(1000000)
```

The resulting data looks as shown in Figure 4.44.

| | komnr | type | amount | jordbruksareal | kcal |
|------------|--------------|-------------|---------------|-----------------------|-------------|
| 0 | 301 | kalv | 254.8 | 9.246000e+09 | 1120.000000 |
| 1 | 301 | lam | 2540.3 | 9.246000e+09 | 2340.000000 |
| 2 | 301 | sau | 2172.0 | 9.246000e+09 | 3710.000000 |
| 3 | 301 | svin | 60496.5 | 9.246000e+09 | 2676.129032 |
| 4 | 412 | kalv | 5545.1 | 2.000000e+06 | 1120.000000 |
| ... | ... | ... | ... | ... | ... |
| 857 | 4202 | egg | 11677.0 | 1.882700e+10 | 1663.333333 |
| 858 | 4203 | egg | 2572.0 | 1.835600e+10 | 1663.333333 |
| 859 | 4207 | egg | 1348.0 | 1.657500e+10 | 1663.333333 |
| 860 | 4223 | egg | 2627.0 | 1.392300e+10 | 1663.333333 |
| 861 | 5438 | egg | 4425.0 | 2.442000e+09 | 1663.333333 |

Figure 4.44 Final data with all datasets integrated

The amount of produced kilocalories are found by adding a column to the data named “produced_kcal” and multiplying the amount with the kilocalories of each type of food produced by each municipality. A sample of the result is shown in Figure 4.45.

| | komnr | type | amount | jordbruksareal | kcal | produced_kcal |
|----------|--------------|-------------|---------------|-----------------------|-------------|----------------------|
| 0 | 301 | kalv | 254.8 | 9.246000e+09 | 1120.000000 | 285376.0 |
| 1 | 301 | lam | 2540.3 | 9.246000e+09 | 2340.000000 | 5944302.0 |
| 2 | 301 | sau | 2172.0 | 9.246000e+09 | 3710.000000 | 8058120.0 |
| 3 | 301 | svin | 60496.5 | 9.246000e+09 | 2676.129032 | 161896440.0 |
| 4 | 412 | kalv | 5545.1 | 2.000000e+06 | 1120.000000 | 6210512.0 |

Figure 4.45 Sample of the data after adding a column “produced_kcal”. The column shows the amount of produced kilocalories of each type of food by each municipality

The “type”, “amount” and “kcal” columns are now removed as the calculation has been done. The code line is shown below:

```
1 final_data4 = final_data3.drop(['amount', 'type', 'kcal'], axis=1)
```

Then the data is grouped by municipality ID (komnr), the produced kilocalories (produced_kcal) is summarized for each municipality, and the area of farmland (jordbruksareal) gets the first value of the group. This means the area of farmland gets the first area of farmland belonging to each municipality. As the area of farmland is the same for all rows in this column, it can be set to the first value. The code line is shown below:

```
1 final_data5 = final_data3.groupby(['komnr'], as_index=False)
   .agg({'produced_kcal': 'sum', 'jordbruksareal': 'first'})
```

Now, a new data frame can be made by taking the old one and dropping the produced kilocalories and the area of farmland. A new column named “kcal_produced per_m2” is added to the new data frame and for each row the produced kilocalories is divided by the area of farmland. This is shown in the code lines below:

```
1 final_data6 = final_data5.drop(['produced_kcal', 'jordbruksareal'], axis=1)
2 final_data6['kcal_produced_per_m2'] = final_data5['produced_kcal'] /
   final_data5['jordbruksareal']
```

The kilocalories produced per square meter for each municipality can be rounded to 4 decimal places using the **round()** method (pandas, n.d.-w).

```
1 final_data6['kcal_produced_per_m2'] = final_data6['kcal_produced_per_m2'].round(4)
```

Using the **head()** method to output a sample of the data frame it can be seen in Figure 4.46 that the answer can be found in the outputted table.

| | komnr | kcal_produced_per_m2 |
|---|-------|----------------------|
| 0 | 301 | 0.0191 |
| 1 | 412 | 17611.4081 |
| 2 | 419 | 433.0908 |
| 3 | 513 | 2858.4526 |
| 4 | 522 | 94.9475 |

Figure 4.46 Final data answering the use case question

4.1.9 Summary of the case study of Python pandas

The data files were easily read using pandas’ reader methods. Most tasks were efficiently solved using simple methods provided by the library. Some methods are native Python method, namely the **remove()** and **fromkeys()** methods. A few of the tasks had to be solved creating some custom scripts, and fuzzy matching required importing an additional library called *thefuzz*. This makes the tool a bit less efficient. However, overall, Python

pandas provides a wide range of methods for solving the necessary tasks to get the data in the desired format and structure, and to perform the final calculation.

4.2 MySQL

In this chapter, the process of preparing the data for an analysis will be performed using Structured Query Language. In this case study, MySQL Workbench (MySQL, 2021) is used to create a database, load and prepare the data. After the data has been prepared, the question defined in the introduction to Chapter 4 can be answered. As MySQL Database Management System (DBMS) is investigated in this case study, some functions, clauses and statements mentioned in this chapter are specific to MySQL. Thus, when describing the solutions to the data preparation tasks, MySQL will be referred to throughout the case study. Other DBMSs might provide different functionalities than described here. However, only MySQL will be considered in this study.

4.2.1 Creating a database and loading the data

First, a database needs to be created. The first line drops the database if it already exists, the second line creates the database, and the third line accesses the database. This is done as shown below:

```
1 DROP DATABASE IF EXISTS `test_db`;  
2 CREATE DATABASE `test_db`;  
3 USE `test_db`;
```

MySQL Workbench allows for simple import of data from the CSV files through the *Table Data Import Wizard*. The data can either be loaded into an already existing table, or a new table can be created when the data is loaded. The Import Wizard lets you choose which columns in the CSV files should be included, and each column in the source data is linked to a column in the table. An example of the import of data is shown in Figure 4.47, where the source and destination columns has been linked and a preview of the data is shown at the bottom of the window. One of the datafiles has file format XLSX, which cannot be imported directly into MySQL database. This file had to be exported as a CSV file before importing it into the database.

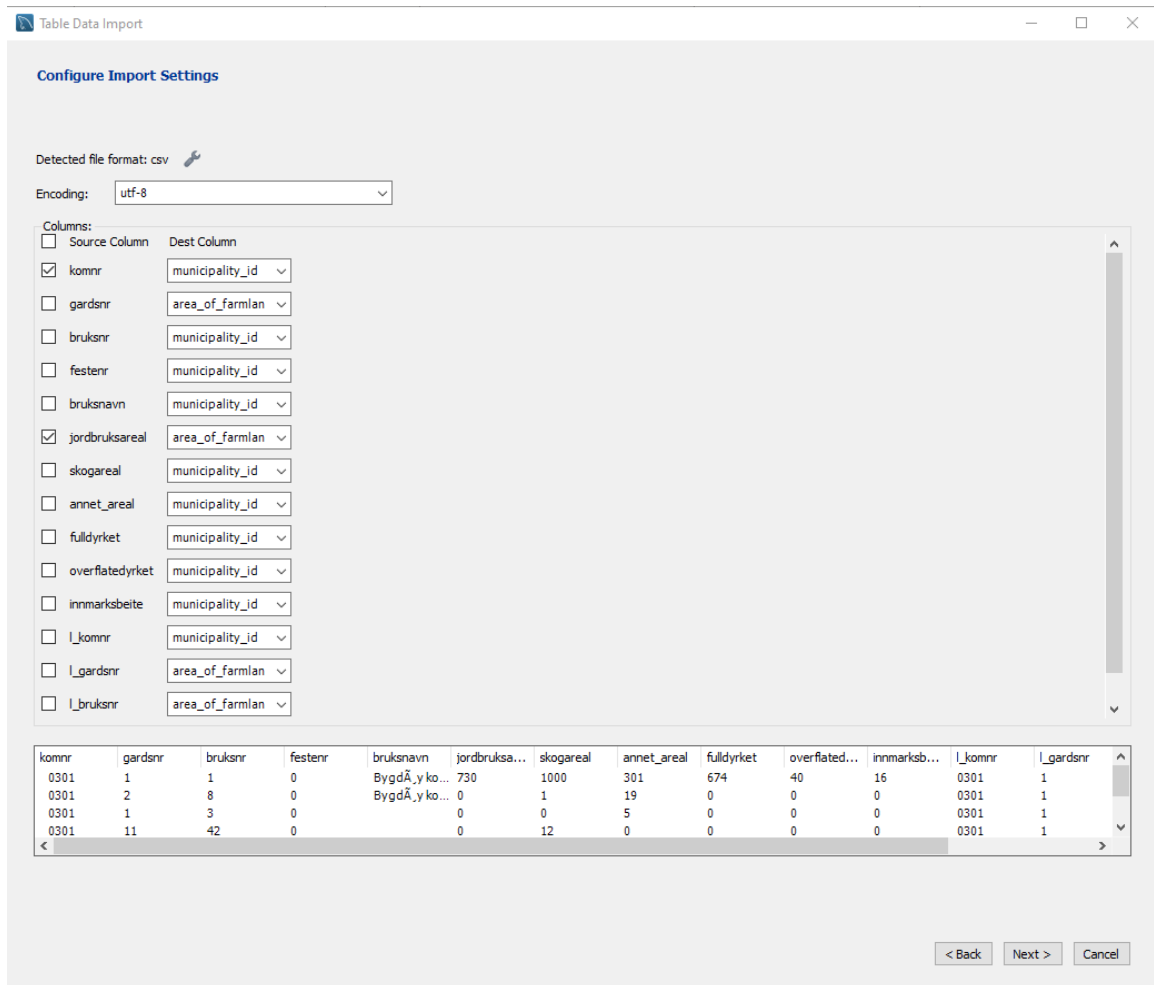


Figure 4.47 An example of the import of CSV files. The Table Data Import Wizard in MySQL Workbench is used. Here the agricultural properties data is shown. The relevant columns are selected and linked to the correct columns in the table ("komnr" is linked to "municipality_id" and "jordbruksareal" is linked to "area_of_farmland").

After the source and destination columns has been linked, the data is imported into the table. This process is done in a similar way for each of the six data files. Tables are created for the data in advance and the data is imported using Table Data Import Wizard. An example of how the tables are created is shown below:

```

1 CREATE TABLE ac_properties(
2     municipality_id INT NOT NULL,
3     area_of_farmland INT NOT NULL
4 );

```

Loading the data in MySQL can be a bit challenging and large amounts of data takes time to load using MySQL Workbench's Table Data Import Wizard. However, data can also be loaded using the LOAD DATA Statement (MySQL, n.d.-e). An example of a simple load data statement is shown below:

```
1 LOAD DATA INFILE 'file_name.csv' INTO TABLE table_name;
```

More clauses can be added for specifying which line or fields terminator is used in the data, for instance. This way of loading the CSV files caused some problems, mainly regarding missing values and wrong data types in the CSV files. No solution to these problems were found, and therefore the Import Wizard was used instead. The problem with the Import Wizard was that it takes a lot more time to import the data. As an example, the agricultural properties CSV file had about 300,000 rows of data and it took about 3 hours to load using Import Wizard. Some people therefore suggest using the LOAD DATA statement instead as they claim it is faster (Stackoverflow, 2019). The problems using the LOAD DATA Statement could have been solved by preprocessing the data using another tool before loading it. Removing missing values and fixing data type mismatches. However, *Table Data Import Wizard* handles this automatically.

4.2.2 Data Discovery

When the data has been loaded into the different tables, the SELECT statement can be used to retrieve the data from one or more tables (MySQL, n.d.-f). Using aggregate functions (MySQL, n.d.-j), some statistics can be retrieved from the data as well. For instance, finding out the number of rows of the table can be done using the COUNT() aggregate function (MySQL, n.d.-j). An example showing the use of the SELECT statement and COUNT() aggregate function is shown below:

```
1 SELECT COUNT(area_of_farmland)
2 FROM ac_properties;
```

The expression inside the COUNT() function, *area_of_farmland*, is one of the columns in the *ac_properties* table. Other aggregate functions available are e.g. AVG(), MAX() and MIN(), which returns the average, maximum and minimum values, respectively (MySQL, n.d.-j). These can be used in the same way as COUNT() was in the example query shown above.

4.2.3 Data Preparation

When the data has been loaded into tables and data discovering is done, the data preparation tasks can be performed. The next sections will consider each of the datasets and their necessary data preparation steps.

Agricultural Properties

Create a dummy table for the data

The table for the agricultural properties data has already been created and Import Wizard has been used to load the data into the table. The query used to create the table is shown below:

```
1 CREATE TABLE ac_properties(  
2   municipality_id INT NOT NULL,  
3   area_of_farmland INT NOT NULL  
4 );
```

Now, the current data can be investigated using the query below:

```
1 SELECT * FROM ac_properties;
```

A sample of the output is shown in Figure 4.48. Since the table columns had to be defined when the table was created, only the relevant columns were added during import. This means that no columns need to be dropped in this case.

| Result Grid | | Filter Rows: |
|-----------------|------------------|--------------|
| municipality_id | area_of_farmland | |
| 301 | 730 | |
| 301 | 23 | |
| 301 | 39 | |
| 301 | 37 | |
| 301 | 950 | |
| 301 | 26 | |
| 301 | 25 | |
| 301 | 17 | |
| 301 | 30 | |
| 301 | 34 | |
| 301 | 5 | |
| 301 | 8 | |
| 301 | 16 | |
| 301 | 3 | |
| 301 | 5 | |
| 301 | 9 | |
| 301 | 7 | |
| 301 | 11 | |
| 301 | 40 | |
| 301 | 8 | |
| 301 | 35 | |
| 301 | 17 | |
| 301 | 48 | |
| 301 | 19 | |
| 301 | 22 | |
| 301 | 81 | |

Figure 4.48 Sample of the Agricultural Properties Data

Investigating the data a bit more revealed that the table only has 215,413 rows, while the original CSV had 314,866 rows. This was done using the query below to count the number of rows:

```
1 SELECT COUNT(municipality_id) FROM ac_properties;
```

Inspecting the data a bit more revealed that the specific row where the Import Wizard had stopped importing. This way, a new CSV file could be created and the remaining rows from the original CSV could be copied over to the new CSV file. Another table named *ac_properties2* was created, and the new CSV with the remaining rows were loaded into the new table in the same way as previously. Loading the original CSV file was attempted several times, without all the rows being successfully imported. Thus, this seemed to be the only way to load the entire file into the database. The LOAD DATA Statement might have solved his problem, but this would have required the data to be preprepared, which would leave only a few preparation tasks for the case study investigating the functionalities provided in MySQL.

Checking for and removing null values

What is needed from this data is the area of farmland per municipality, as this will be used to answer the analysis question. What can be difficult to see in the retrieved data, especially with large datasets, is the null values. However, they can be found using the query below:

```
1 SELECT * FROM ac_properties WHERE area_of_farmland=0;
```

This resulted in an empty table, meaning that the data does not contain any null values. However, removing them could have been done by replacing “SELECT *” with “DELETE” in the previous query, as shown below:

```
1 DELETE FROM ac_properties WHERE area_of_farmland=0;
```

This is done for both the *ac_properties* and the *ac_properties2* tables.

Unioning the two tables of Agricultural properties data

Since the data had to be loaded into two different tables, they have to be unioned before the rest of the data preparation tasks are performed. A new table is created named *ac_properties3* and the *ac_properties* and *ac_properties2* tables are unioned as shown in the line below using the UNION clause (MySQL, n.d.-g):

```
1 INSERT INTO ac_properties3 (municipality_id, area_of_farmland)
2 SELECT * FROM ac_properties
3 UNION
4 SELECT * FROM ac_properties2;
```

Grouping and aggregating the data

The sample shown in Figure 4.48 revealed that the data needs to be aggregated and grouped by municipality ID in order to get the area of farmland per municipality. This can be done first in a test query using the SELECT clause, as shown below:

```
1 SELECT municipality_id, SUM(area_of_farmland) AS total_area FROM ac_properties
2 GROUP BY municipality_id;
```

In order for the grouped data to be stored somewhere, the only solution found was creating a new table and inserting the data from the old table into the new table using the

GROUP BY (MySQL, n.d.-b) clause to group the data by municipality ID and the SUM() aggregate function (MySQL, n.d.-j) to summarize the total area of farmland per municipality. Since the data is in two datasets, the new table is created in the exact same way as the previous one, as shown below:

```

1 CREATE TABLE area_per_municipality(
2     municipality_id INT NOT NULL,
3     area_of_farmland INT NOT NULL
4 );
5 SELECT * FROM area_per_municipality;
6
7 ## Insert the grouped data into the new table
8 INSERT INTO area_per_municipality (municipality_id, area_of_farmland)
9 SELECT municipality_id, SUM(area_of_farmland)
10 FROM ac_properties3
11 GROUP BY municipality_id;

```

In lines 1 to 4, the table is created. In lines 8 to 11 the data is grouped and aggregated and inserted into the new table. Using the SELECT clause, the new table with the aggregated data can be viewed. A sample of the data is shown in Figure 4.49.

| | municipality_id | area_of_farmland |
|---|-----------------|------------------|
| ▶ | 301 | 9246 |
| | 412 | 2 |
| | 419 | 3 |
| | 3440 | 34678 |
| | 528 | 3 |
| | 633 | 2 |
| | 829 | 19 |
| | 904 | 26 |
| | 919 | 4 |
| | 940 | 6 |
| | 1101 | 52571 |
| | 1103 | 91813 |
| | 1106 | 8672 |
| | 1108 | 90687 |
| | 1111 | 16791 |
| | 1112 | 24794 |
| | 1114 | 61720 |
| | 1119 | 119595 |
| | 1120 | 74832 |
| | 1121 | 83813 |
| | 1122 | 55535 |

Figure 4.49 A sample of the Agricultural Properties data. The data has been aggregated and grouped by municipality ID.

Now this data is ready to be used in the final calculation. The dummy tables can be removed, as it is no longer needed. This is done in the queries below:

```
1 DROP TABLE ac_properties;
2 DROP TABLE ac_properties2;
3 DROP TABLE ac_properties3;
```

Meat Deliveries

Creating a dummy table for the data

The meat deliveries data is loaded into a table with the following structure:

```
1 CREATE TABLE meats(
2     municipality_id INT NOT NULL,
3     and_kg INT NOT NULL,
4     kje_kg INT NOT NULL,
5     gris_kg INT NOT NULL,
6     purke_kg INT NOT NULL,
7     raane_kg INT NOT NULL,
8     gaas_kg INT NOT NULL,
9     hane_kg INT NOT NULL,
10    hons_kg INT NOT NULL,
11    kalkun_kg INT NOT NULL,
12    kalv_kg INT NOT NULL,
13    kylling_kg INT NOT NULL,
14    lam_kg INT NOT NULL,
15    lam_villsau_kg INT NOT NULL,
16    sau_kg INT NOT NULL,
17    ungsau_kg INT NOT NULL,
18    vaer_kg INT NOT NULL,
19    ku_kg INT NOT NULL,
20    kvige_kg INT NOT NULL,
21    okse_kg INT NOT NULL,
22    ungotse_kastrat_kg INT NOT NULL,
23    ungotse_kg INT NOT NULL
24 );
```

The final table should contain three columns: municipality ID, amount and type of meat. It should also be grouped by municipality ID and the sum of the amount for each type of food for each municipality should be calculated. This is a wide table with 21 columns. It needs to be unpivoted to get the desired structure, which means some columns will be converted to rows.

Grouping, aggregating and unpivoting the data

The data is first grouped and aggregated to simplify the types of meat. Because the nutrients table (which will be discussed later) contains only some of these meat types, as previously mentioned in Section 4.1.3, the meat types will be simplified. Some of them will be removed, as they have no equivalent in the nutrients table. Others will be merged into

one broader category, as they are basically the same type. Again, before the grouping of data can happen, a new table has to be created. This is done as shown below:

```
1 CREATE TABLE meat(  
2     municipality_id INT NOT NULL,  
3     type VARCHAR(100),  
4     amount INT NOT NULL  
5 );
```

Now, the data can be both grouped and unpivoted in the same query. The grouping and aggregation is done using the SUM() and GROUP BY clauses. The unpivoting is done by selecting the old columns and setting a new column name and value for it. An *amount* column gets the value from the different meat type columns and the *type* column is manually set to a new type. This is to create some new and broader categories for some of the types of meat.

```

1 INSERT INTO meat (municipality_id, amount, type)
2 SELECT municipality_id, (SUM(gris_kg) + SUM(purke_kg) + SUM(raane_kg)) amount, 'svin' type
3 FROM meats
4 GROUP BY municipality_id, type
5 UNION ALL
6 SELECT municipality_id, (SUM(hane_kg) + SUM(hons_kg)) amount, 'hone' type
7 FROM meats
8 GROUP BY municipality_id, type
9 UNION ALL
10 SELECT municipality_id, (SUM(vaer_kg) + SUM(sau_kg) + SUM(ungsau_kg)) amount, 'sau' type
11 FROM meats
12 GROUP BY municipality_id, type
13 UNION ALL
14 SELECT municipality_id, (SUM(lam_kg) + SUM(lam_villsau_kg)) amount, 'lam' type
15 FROM meats
16 GROUP BY municipality_id, type
17 UNION ALL
18 SELECT municipality_id, (SUM(ku_kg) + SUM(kvige_kg) +
                            SUM(okse_kg) + SUM(ungokse_kastrat_kg) +
                            SUM(ungku_kg)) amount,
                            'okse' type
19 FROM meats
20 GROUP BY municipality_id, type
21 UNION ALL
22 SELECT municipality_id, SUM(gaas_kg) amount, 'gaas' type
23 FROM meats
24 GROUP BY municipality_id, type
25 UNION ALL
26 SELECT municipality_id, SUM(and_kg) amount, 'and' type
27 FROM meats
28 GROUP BY municipality_id, type
29 UNION ALL
30 SELECT municipality_id, SUM(kalkun_kg) amount, 'kalkun' type
31 FROM meats
32 GROUP BY municipality_id, type
33 UNION ALL
34 SELECT municipality_id, SUM(kalv_kg) amount, 'kalv' type
35 FROM meats
36 GROUP BY municipality_id, type
37 UNION ALL
38 SELECT municipality_id, SUM(kylling_kg) amount, 'kylling' type
39 FROM meats
40 GROUP BY municipality_id, type;

```

Checking for and removing null values

This results in a lot of rows where amount is equal to 0, and these rows can be removed as they are redundant. This is done using the DELETE Statement (MySQL, n.d.-d) as shown below:

```

1 DELETE FROM meat WHERE amount=0;

```

The current data is in the desired structure and can be used for the final calculation. A sample of the data is shown in Figure 4.50.

| | municipality_id | type | amount |
|---|-----------------|------|---------|
| ▶ | 1119 | svin | 1232322 |
| | 528 | svin | 80553 |
| | 1101 | svin | 191434 |
| | 542 | svin | 6082 |
| | 5032 | svin | 118790 |
| | 1141 | svin | 29377 |
| | 1211 | svin | 6544 |
| | 1134 | svin | 101795 |
| | 125 | svin | 129726 |
| | 236 | svin | 332915 |
| | 701 | svin | 49505 |
| | 1102 | svin | 117155 |
| | 1114 | svin | 279021 |
| | 1260 | svin | 2448 |
| | 231 | svin | 175903 |
| | 227 | svin | 24993 |
| | 211 | svin | 12990 |
| | 716 | svin | 180387 |

Figure 4.50 A sample of the current Meat Deliveries data. The data has been unpivoted, grouped and aggregated, and rows where amount=0 has been removed.

Drop dummy table

Lastly, the dummy table *meats* is dropped as shown in the query below:

```
1 DROP TABLE meats;
```

Dairy Deliveries

Creating a dummy table for the data

The dairy deliveries data is loaded into a table with the following structure:

```
1 CREATE TABLE milk(
2     municipality_id INT NOT NULL,
3     melk INT NOT NULL,
4     geitmelk INT NOT NULL
5 );
```

The columns are municipality ID, cow milk (“melk”) and goat milk (“geitmelk”). A sample of the data is shown in Figure 4.51.

| | municipality_id | melk | geitmelk |
|---|-----------------|--------|----------|
| ▶ | 101 | 92957 | 0 |
| | 101 | 533955 | 0 |
| | 105 | 500496 | 0 |
| | 105 | 147071 | 0 |
| | 105 | 203632 | 0 |
| | 105 | 174900 | 0 |
| | 106 | 244342 | 0 |
| | 122 | 118158 | 0 |
| | 122 | 379979 | 0 |
| | 127 | 91421 | 0 |
| | 127 | 167587 | 0 |

Figure 4.51 A sample of the Dairy Deliveries data before making any changes

Adding a “type” column

In order to get this data to match the other tables, it needs a *type* column which shows what type of milk and the amount. First, a test query is written in order to ensure the results are as desired before the changes to the table is made permanently. The query shown below selects the *municipality_id*, *melk*, *geitmelk* and *type* columns. However, for the *type* column a CASE statement (MySQL, n.d.-i) is used to determine what type of milk should be assigned to it. The CASE statement says that if the *melk* column is equal to 0 then the type is set to “geitmelk” (or goat milk in English), else the type is set to “melk” (meaning cow milk). In the table, each row represents one organization’s production of milk. As each organization only produces one type of milk, the rows will contain a 0 in either the “melk” or “geitmelk” column. Thus, using the CASE statement the type for each row in the table can be determined by using the 0’s as shown in the query below:

```

1 SELECT municipality_id, melk, geitmelk,
      CASE WHEN melk = 0 THEN 'geitmelk'
      ELSE 'melk'
      END AS type
FROM milk;

```

As can be seen in Figure 4.52 this returns a table where the type is set to “melk” when the *melk* column is not 0 (from the ELSE clause in the query above).

| | municipality_id | melk | geitmelk | type |
|---|-----------------|--------|----------|------|
| ▶ | 101 | 92957 | 0 | melk |
| | 101 | 533955 | 0 | melk |
| | 105 | 500496 | 0 | melk |
| | 105 | 147071 | 0 | melk |
| | 105 | 203632 | 0 | melk |
| | 105 | 174900 | 0 | melk |
| | 106 | 244342 | 0 | melk |
| | 122 | 118158 | 0 | melk |
| | 122 | 379979 | 0 | melk |

Figure 4.52 A sample of the Dairy Deliveries data with a "type" column

Now, the previous query only printed out temporary results. To make the changes to the table, the ALTER TABLE Statement (MySQL, n.d.-c) is first used to add a type column of type VARCHAR with a fixed length of 100.

```
1 ALTER TABLE milk ADD type VARCHAR(100);
```

The *type* column is now added, but it only contains null values still, as shown in Figure 4.53.

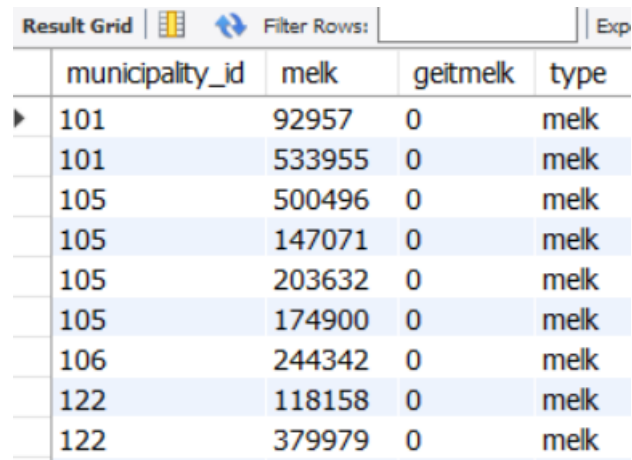
| | municipality_id | melk | geitmelk | type |
|---|-----------------|--------|----------|------|
| ▶ | 101 | 92957 | 0 | NULL |
| | 101 | 533955 | 0 | NULL |
| | 105 | 500496 | 0 | NULL |
| | 105 | 147071 | 0 | NULL |
| | 105 | 203632 | 0 | NULL |
| | 105 | 174900 | 0 | NULL |
| | 106 | 244342 | 0 | NULL |
| | 122 | 118158 | 0 | NULL |
| | 122 | 379979 | 0 | NULL |

Figure 4.53 A sample of the Dairy Deliveries table after adding the type column

The next thing to do is to update the *type* column with the correct milk type using the UPDATE Statement (MySQL, n.d.-h), and the CASE Statement previously tested. This is done in the query below:

```
1 UPDATE milk
2 SET type = CASE
3     WHEN melk = 0 THEN 'geitmelk'
4     ELSE 'melk'
5     END;
```

The *milk* table now looks similar to the table that was outputted when the test query was executed, as can be seen in Figure 4.54.



| | municipality_id | melk | geitmelk | type |
|---|-----------------|--------|----------|------|
| ▶ | 101 | 92957 | 0 | melk |
| | 101 | 533955 | 0 | melk |
| | 105 | 500496 | 0 | melk |
| | 105 | 147071 | 0 | melk |
| | 105 | 203632 | 0 | melk |
| | 105 | 174900 | 0 | melk |
| | 106 | 244342 | 0 | melk |
| | 122 | 118158 | 0 | melk |
| | 122 | 379979 | 0 | melk |

Figure 4.54 A sample of the Dairy Deliveries data after setting the type

Merging columns into an “amount” column

Now that the type of milk is in the *type* column, the columns *melk* and *geitmelk* are redundant. To match the other data tables, these two columns should be merged into one column named *amount*.

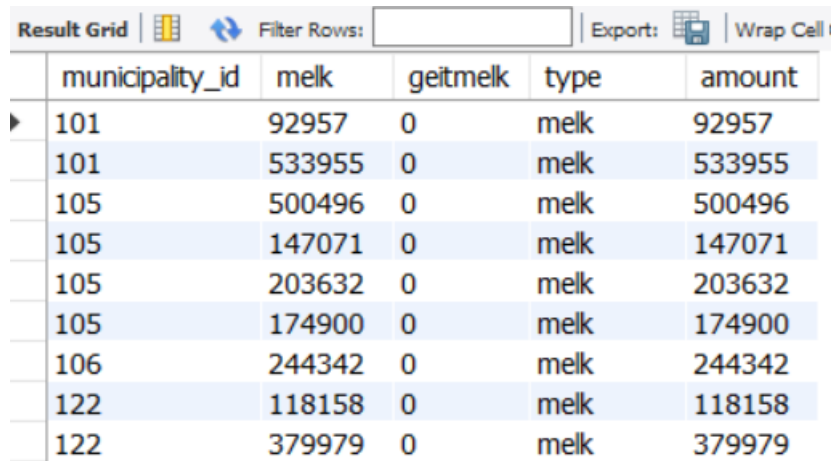
The *amount* column is first added to the *milk* table, by again using the ALTER TABLE Statement as shown below:

```
1 ALTER TABLE milk ADD amount INT NOT NULL;
```

Two UPDATE Statements are used to get the values from the *melk* and *geitmelk* columns and add them to the *amount* columns. The first one, shown in line 1 to 3 below, updates the *milk* table and sets the amount equal to the value in the *melk* column for all rows where the *geitmelk* column contains a 0. The next one, shown in line 5 to 7 below, updates the *milk* table and sets the amount equal to the values in the *geitmelk* column for all rows where the *melk* column contains a 0.

```
1 UPDATE milk
2 SET amount=melk
3 WHERE geitmelk = 0;
4
5 UPDATE milk
6 SET amount=geitmelk
7 WHERE melk = 0;
```


A sample of the current data is shown Figure 4.55.



| | municipality_id | melk | geitmelk | type | amount |
|---|-----------------|--------|----------|------|--------|
| ▶ | 101 | 92957 | 0 | melk | 92957 |
| | 101 | 533955 | 0 | melk | 533955 |
| | 105 | 500496 | 0 | melk | 500496 |
| | 105 | 147071 | 0 | melk | 147071 |
| | 105 | 203632 | 0 | melk | 203632 |
| | 105 | 174900 | 0 | melk | 174900 |
| | 106 | 244342 | 0 | melk | 244342 |
| | 122 | 118158 | 0 | melk | 118158 |
| | 122 | 379979 | 0 | melk | 379979 |

Figure 4.55 A sample of the Dairy Deliveries data with an “amount” column

Drop redundant columns

Now that the amount of milk of the different types are in the *amount* column, the *melk* and *geitmelk* columns are no longer needed. To remove them, the ALTER TABLE and DROP clauses are used. In MySQL several DROP clauses can be used in a single ALTER TABLE clause, so both columns can be removed as shown below:

```
1 ALTER TABLE milk
2 DROP COLUMN melk,
3 DROP COLUMN geitmelk;
```

A sample of the current data is shown in Figure 4.56.

| | municipality_id | type | amount |
|---|-----------------|------|--------|
| ▶ | 101 | melk | 92957 |
| | 101 | melk | 533955 |
| | 105 | melk | 500496 |
| | 105 | melk | 147071 |
| | 105 | melk | 203632 |
| | 105 | melk | 174900 |
| | 106 | melk | 244342 |
| | 122 | melk | 118158 |
| | 122 | melk | 379979 |

Figure 4.56 A sample of the Dairy Deliveries data. Redundant columns has been removed.

Grouping and aggregating the data

The last thing to do is to aggregate the data, to get the total amount of the different types of milk produced in each municipality. This is done similarly to the meat deliveries data, by creating a new table for the aggregated data and inserting the aggregated data from the old table into the new table. The table is created as shown below:

```

1 CREATE TABLE dairy(
2     municipality_id INT NOT NULL,
3     amount INT NOT NULL,
4     type VARCHAR(100)
5 );

```

The data is then aggregated, grouped and inserted into the new table using the query shown below:

```

1 INSERT INTO dairy (municipality_id, amount, type)
2 SELECT municipality_id, SUM(amount), type
3 FROM milk
4 GROUP BY municipality_id;

```

The data now looks like the sample shown in Figure 4.57. It now shows the amount of each type of milk produced in each municipality.

| | municipality_id | amount | type |
|---|-----------------|---------|------|
| ▶ | 101 | 1565163 | melk |
| | 105 | 2310494 | melk |
| | 106 | 592213 | melk |
| | 122 | 1036258 | melk |
| | 127 | 524531 | melk |
| | 128 | 2063640 | melk |
| | 135 | 353571 | melk |

Figure 4.57 A sample of the Dairy Deliveries data after grouping and aggregating

Check for and remove null values

Before moving on to the next dataset, the amount column can be checked for null values using the query below. The second line of the query deletes the rows where amount is equal to 0.

```
1 SELECT * FROM dairy WHERE amount=0;
2 DELETE FROM dairy WHERE amount=0;
```

Drop dummy table

Lastly, the dummy table *milk* can be dropped as shown below:

```
1 DROP TABLE milk;
```

Egg Deliveries

Creating a dummy table for the data

The egg deliveries data is loaded into a table with the structure shown below:

```
1 CREATE TABLE eggs(
2     municipality_id INT NOT NULL,
3     amount INT NOT NULL
4 );
5 SELECT * FROM eggs;
```

A sample of the current data can be seen in Figure 4.58.

| Result Grid | | Filter Rows: |
|-------------|-----------------|--------------|
| | municipality_id | amount |
| ▶ | 1120 | 131770 |
| | 5004 | 129940 |
| | 1120 | 133703 |
| | 1445 | 121256 |
| | 1432 | 8052 |
| | 5023 | 144487 |
| | 417 | 88236 |
| | 605 | 85683 |
| | 701 | 89572 |
| | 1121 | 9382 |
| | 1511 | 107880 |
| | 1563 | 103647 |
| | 128 | 145481 |
| | 427 | 206329 |
| | 415 | 131255 |
| | 415 | 114099 |
| | 528 | 147247 |
| | 1133 | 127874 |
| | 1141 | 96817 |
| | 1520 | 72010 |
| | 5004 | 147266 |
| | 5037 | 122470 |
| | 427 | 13455 |
| | 520 | 106216 |
| | 528 | 125488 |
| | 1102 | 108097 |
| | 1141 | 99965 |
| | 1141 | 95695 |
| | 1127 | 86227 |

Figure 4.58 A sample of the Egg Deliveries data before making any changes

Check for and remove null values

As the figure only shows a sample of the data in the table, the data should be checked for null values. In this case, any rows where amount is equal to 0 should be removed. Using the first query shown below, the rows where amount equals 0 will be outputted. Using the second query shown below, the rows where amount equals 0 is deleted from the table.

```
1 SELECT * FROM eggs WHERE amount=0; -- outputs rows where amount=0
2 DELETE FROM eggs WHERE amount=0; -- deletes the rows where amount=0
```

Add a “type” column

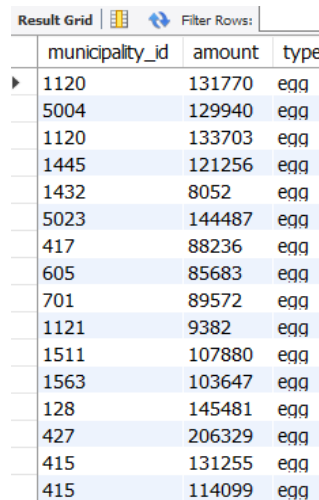
The data currently has a *municipality ID* and *amount* column. The other datasets contain different types of foods, and therefore also has a *type* column. To match the other datasets, a *type* column is added to the egg deliveries data using the query below:

```
1 ALTER TABLE eggs ADD type VARCHAR(100);
```

Since this table only contains records of amount of eggs delivered, the *type* column will be set equal to “egg”. This is done in the following query:

```
1 UPDATE eggs SET type = 'egg';
```

The data now looks as shown in Figure 4.59.



| municipality_id | amount | type |
|-----------------|--------|------|
| 1120 | 131770 | egg |
| 5004 | 129940 | egg |
| 1120 | 133703 | egg |
| 1445 | 121256 | egg |
| 1432 | 8052 | egg |
| 5023 | 144487 | egg |
| 417 | 88236 | egg |
| 605 | 85683 | egg |
| 701 | 89572 | egg |
| 1121 | 9382 | egg |
| 1511 | 107880 | egg |
| 1563 | 103647 | egg |
| 128 | 145481 | egg |
| 427 | 206329 | egg |
| 415 | 131255 | egg |
| 415 | 114099 | egg |

Figure 4.59 A sample of the Egg Delivery data with a “type” column

Grouping and aggregating the data

What is missing now is the aggregation of the data. First, a new table for the grouped and aggregated data is created as shown below:

```
1 CREATE TABLE egg(  
2     municipality_id INT NOT NULL,  
3     amount INT NOT NULL,  
4     type VARCHAR(100)  
5 );
```

The amount of egg delivered for each municipality is what is needed from the data; therefore, the data needs to be grouped by municipality ID and the amount of eggs for each municipality needs to be summarized. This is done in the query below:

```
1 INSERT INTO egg (municipality_id, amount, type)  
2 SELECT municipality_id, SUM(amount), type  
3 FROM eggs  
4 GROUP BY municipality_id;
```

Comparing the previous data in Figure 4.59 and the data in Figure 4.60, we can see that the two rows with municipality ID of 415 in Figure 4.59 is reduced to one row in Figure 4.60.

| | municipality_id | amount | type |
|---|-----------------|---------|------|
| ▶ | 1120 | 3755978 | egg |
| | 5004 | 4035393 | egg |
| | 1445 | 368264 | egg |
| | 1432 | 8052 | egg |
| | 5023 | 726388 | egg |
| | 417 | 378638 | egg |
| | 605 | 1188705 | egg |
| | 701 | 194652 | egg |
| | 1121 | 2019251 | egg |
| | 1511 | 734764 | egg |
| | 1563 | 292697 | egg |
| | 128 | 608609 | egg |
| | 427 | 419188 | egg |
| | 415 | 832303 | egg |
| | 528 | 2057376 | egg |
| | 1133 | 792935 | egg |
| | 1141 | 3133597 | egg |

Figure 4.60 A sample of the Egg Delivery data after grouping and aggregating

Drop dummy table

Now that the new table *egg* contains the data, the dummy table *eggs* can be dropped as it is no longer needed. This is done in the query below:

```
1 DROP TABLE eggs;
```

Now the egg deliveries data is prepared as well, and it is ready for the final calculation.

Grain Deliveries

Creating a dummy table for the data

The structure of the table where the grain deliveries data is loaded is shown below:

```
1 CREATE TABLE grains(
2     municipality_id INT NOT NULL,
3     bygg INT NOT NULL,
4     ertter INT NOT NULL,
5     hvete INT NOT NULL,
6     rug INT NOT NULL
7 );
```

Grouping, aggregating and unpivoting the data

This table also needs to be unpivoted so that the types of grain is in a *type* column, and the amount is in an *amount* column. It should also be grouped and aggregated as the other datasets. Another table is created first. This table has the municipality ID, amount, and type columns and is named *grain*, as shown below:

```
1 CREATE TABLE grain(  
2     municipality_id INT NOT NULL,  
3     amount INT NOT NULL,  
4     type VARCHAR(100)  
5 );
```

The query for grouping, aggregating and unpivoting is similar to the one used for the meat deliveries data, as can be seen below:

```
1 INSERT INTO grain (municipality_id, amount, type)  
2 SELECT municipality_id, SUM(bygg) amount, 'byggmel' type  
3 FROM grains  
4 GROUP BY municipality_id, type  
5 UNION ALL  
6 SELECT municipality_id, SUM(erter) amount, 'erter' type  
7 FROM grains  
8 GROUP BY municipality_id, type  
9 UNION ALL  
10 SELECT municipality_id, SUM(hvete) amount, 'hvetemel' type  
11 FROM grains  
12 GROUP BY municipality_id, type  
13 UNION ALL  
14 SELECT municipality_id, SUM(rug) amount, 'rugmel' type  
15 FROM grains  
16 GROUP BY municipality_id, type
```

Removing null values

This results in a lot of rows where amount is equal to 0, as with the meat deliveries data. These rows are deleted as shown below:

```
1 DELETE FROM grain WHERE amount=0;
```

The data is prepared for the final calculation. A sample of the current grain deliveries data is shown in Figure 4.61.

| | municipality_id | amount | type |
|---|-----------------|--------|----------|
| ▶ | 231 | 54771 | hvetemel |
| | 417 | 51845 | hvetemel |
| | 119 | 16718 | hvetemel |
| | 214 | 53754 | hvetemel |
| | 106 | 36189 | hvetemel |
| | 605 | 14440 | hvetemel |
| | 125 | 11204 | hvetemel |
| | 716 | 43069 | hvetemel |
| | 229 | 29403 | hvetemel |
| | 234 | 33288 | hvetemel |
| | 819 | 56201 | hvetemel |

Figure 4.61 A sample of the Grain Deliveries data. The data have been grouped, aggregated and unpivoted, and rows where amount=0 has been removed.

Drop dummy table

The dummy table *grains* can be dropped as shown below:

```
1 DROP TABLE grains;
```

Nutrients Table

Create a dummy table for the data

The data is first loaded into a dummy table with the following structure:

```
1 CREATE TABLE nutrients_table(
2     food_type VARCHAR(100),
3     kcal INT NOT NULL
4 );
```

Remove null data

As can be seen in Figure 4.62, there are several rows where *kcal* is equal to 0. The first rows of the data contain a title and categories of food, or they are empty. These rows are not relevant and should be removed. This is done in the query below:

```
1 DELETE FROM nutrients_table WHERE kcal=0;
```


| food_type | kcal |
|---------------------------------------|------|
| Den norske matvaretabelen 2021 | 0 |
| | 0 |
| | 0 |
| Melk og melkeprodukter | 0 |
| Melk og melkebasert drikke | 0 |
| Geitmelk, langtidsholdbar | 69 |
| Helmelk, 3,5 % fett, laktosefri | 60 |
| Helmelk, 3,5 % fett, Tine | 63 |
| Helmelk, 3,9 % fett, Q-meieriene | 67 |
| Helmelk, 4,1 % fett, Å_kologisk | 67 |
| Helmelk, uspesifisert | 62 |
| Kaffemelk, 3,5 % fett | 61 |
| Kakao, med lettmelk, tilberedt | 80 |
| Lettmelk, 0,5 % fett, vitamin D, l... | 33 |
| Lettmelk, 0,5 % fett, vitamin D, ... | 37 |

Figure 4.62 A sample of the loaded nutrients table data

The data in the *nutrients_table* now looks as shown in Figure 4.63.

| food_type | kcal |
|---|------|
| Geitmelk, langtidsholdbar | 69 |
| Helmelk, 3,5 % fett, laktosefri | 60 |
| Helmelk, 3,5 % fett, Tine | 63 |
| Helmelk, 3,9 % fett, Q-meieriene | 67 |
| Helmelk, 4,1 % fett, Å_kologisk | 67 |
| Helmelk, uspesifisert | 62 |
| Kaffemelk, 3,5 % fett | 61 |
| Kakao, med lettmelk, tilberedt | 80 |
| Lettmelk, 0,5 % fett, vitamin D, lakto... | 33 |
| Lettmelk, 0,5 % fett, vitamin D, Q-m... | 37 |
| Lettmelk, 0,5 % fett, vitamin D, Tine | 37 |
| Lettmelk, 0,5-0,7 % fett, vitamin D, ... | 38 |
| Lettmelk, 0,7 % fett, Å_kologisk | 37 |
| Lettmelk, 1,0 % fett, laktosefri | 37 |
| Lettmelk, 1,0 % fett, laktoseredusert | 43 |

Figure 4.63 A sample of the Nutrients Table data. Rows where kcal=0 has been removed.

Split column and changing letter case

The food types in the nutrients table contains additional information which is not necessary in this case. This information will be removed using the `SUBSTRING_INDEX()` function, which returns a substring from a string before the specified number of occurrences of the delimiter (MySQL, n.d.-a). The food type should be set equal to the first substring. First, a *type* column is added to the table as shown below:

```
1 ALTER TABLE nutrients_table ADD type VARCHAR(100);
```

Now, the *type* column only contains NULL, as shown in Figure 4.64. This column should be filled with the first part of the string in the initial *food_type* column. In addition, the LOWER() function (MySQL, n.d.-a) is used to convert the string to lowercase. This is done in the query below:

```
1 UPDATE nutrients_table
2 SET type=LOWER(SUBSTRING_INDEX(food_type, ' ', 1));
```

| food_type | kcal | type |
|----------------------------------|------|------|
| Geitmelk, langtidsholdbar | 69 | NULL |
| Helmelk, 3,5 % fett, laktosefri | 60 | NULL |
| Helmelk, 3,5 % fett, Tine | 63 | NULL |
| Helmelk, 3,9 % fett, Q-meieriene | 67 | NULL |
| Helmelk, 4,1 % fett, Å, kologisk | 67 | NULL |

Figure 4.64 A sample of the Nutrients Table data after adding a type column

After filling the *type* column with the lowercase substring, the initial *food_type* column can be removed as shown in the query below:

```
1 ALTER TABLE nutrients_table
2 DROP COLUMN food_type;
```

The nutrients table data now looks as shown in Figure 4.65.

| kcal | type |
|------|-----------|
| 69 | geitmelk |
| 60 | hmelk |
| 63 | hmelk |
| 67 | hmelk |
| 67 | hmelk |
| 62 | hmelk |
| 61 | kaffemelk |
| 80 | kakao |
| 33 | lettmelk |
| 37 | lettmelk |

Figure 4.65 A sample of the Nutrients Table data with new "type" column

Grouping and aggregating the data

The data now contains several rows where the food type is the same, but with different kilocalorie values. This is because the initial data said something about whether the food was prepared, how it was prepared, how much fat it contained, what part of the animal the meat was from, etc. This information was removed from the initial *food_type* column in

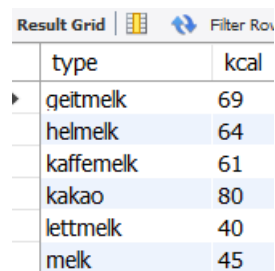
order to match the food types of the food production datasets. As the food production datasets don't give any information about these kinds of things, a simplification has to be made. The data is grouped by type and the average of the kilocalories of all records of each type is calculated. First, a new table is created to insert the data into. This is done similarly as before in the query below:

```
1 CREATE TABLE kcal_per_food(  
2     type VARCHAR(100),  
3     kcal INT NOT NULL  
4 );
```

Then, the data from the old table can be grouped by *type* and the average of the *kcal* can be calculated. This is done in the query below:

```
1 INSERT INTO kcal_per_food (type, kcal)  
2 SELECT type, AVG(kcal)  
3 FROM nutrients_table  
4 GROUP BY type;
```

The data now looks as shown in Figure 4.66. There is now only one row for each type of food in the Nutrients Table, and the *kcal* value is an average of all the similar food types in the initial data.



| type | kcal |
|-----------|------|
| geitmelk | 69 |
| helmelk | 64 |
| kaffemelk | 61 |
| kakao | 80 |
| lettmelk | 40 |
| melk | 45 |

Figure 4.66 A sample of the final Nutrients Table data. The data has been grouped by type and the average of the kilocalories of each type of food has been calculated.

4.2.4 Data Mapping and Integration

The current SQL tables are listed below:

1. area_per_municipality: Total area of farmland per municipality
2. grain: The amount of different types of grain produced in each municipality
3. egg: The amount of eggs produced in each municipality
4. milk: The amount of cow and goat milk produced in each municipality

5. meat: The amount of different kinds of meat produced in each municipality
6. kcal_per_food: The kilocalories contained in each type of food

Union Food Production data

The food production data (from point 2 to 5 in the list above) should be unioned into a table of all the produced food. First, a new table is made for the unioned data as shown in the query below:

```
1 CREATE TABLE food_production(  
2     municipality_id INT NOT NULL,  
3     type VARCHAR(100),  
4     amount INT NOT NULL  
5 );
```

Then the data is unioned. While unioning, the data is grouped by municipality ID and type and the total sum is calculated. This results in a table showing the amount of each type of food produced in each municipality. The query is shown below:

```
1 INSERT INTO food_production (municipality_id, type, amount)  
2 SELECT municipality_id, type, SUM(amount)  
3 FROM eggs  
4 GROUP BY municipality_id, type  
5 UNION ALL  
6 SELECT municipality_id, type, SUM(amount)  
7 FROM milk  
8 GROUP BY municipality_id, type  
9 UNION ALL  
10 SELECT municipality_id, type, SUM(amount)  
11 FROM meat  
12 GROUP BY municipality_id, type  
13 UNION ALL  
14 SELECT municipality_id, type, SUM(amount)  
15 FROM grain  
16 GROUP BY municipality_id, type;
```

Joining Food Production data with the rest of the data

In order to do the final calculation, the food production table should be joined with the remaining tables (from point 1. and 6. in the list above). As before, a new table is created for the joined data. The query is shown below:

```

1 CREATE TABLE results(
2 municipality_id INT NOT NULL,
3 amount INT NOT NULL,
4 type VARCHAR(100),
5 kcal INT NOT NULL,
6 area_of_farmland BIGINT NOT NULL
7 );

```

Then the data is joined by selecting the relevant columns from the different datasets and joining on some conditions. The conditions are:

- `kcal_per_food` is joined on the condition that the type of food in this data is equal to the type of food in the food production data table
- `area_per_municipality` is joined on the condition that the municipality ID of this data is equal to the municipality ID of the food production data table

Since the kilocalories of the food types are given in kilocalories per hectogram, the values in the `kcal` column need to be converted to kilocalories per kilogram. This is done by dividing the value by 0.1. The area of farmland is given in square kilometers, and as the final calculation will find the kilocalories worth of food produced per square meter for each municipality, this should be converted to square meters. This is done by multiplying the values in the `area_of_farmland` column by 1,000,000. The query that joins the data, converts kilocalories and area of farmland, and inserts it into the `results` table is shown below:

```

1 INSERT INTO results (municipality_id, amount, type, kcal, area_of_farmland)
2 SELECT food_production.municipality_id,
3        food_production.amount AS amount,
4        food_production.type,
5        kcal_per_food.kcal/0.1 AS kcal,
6        area_per_municipality.area_of_farmland*1000000 AS area_of_farmland
7 FROM food_production
8 JOIN kcal_per_food ON kcal_per_food.type = food_production.type
9 JOIN area_per_municipality ON
10      area_per_municipality.municipality_id = food_production.municipality_id

```

The joined data in the `results` table now looks as shown in Figure 4.67.

| Result Grid | Filter Rows: | Export: | Wrap Cell Content | |
|-----------------|--------------|----------|-------------------|------------------|
| municipality_id | amount | type | kcal | area_of_farmland |
| 301 | 82021 | melk | 450 | 9246000000 |
| 301 | 36 | sau | 3710 | 9246000000 |
| 301 | 23850 | hvetemel | 3390 | 9246000000 |
| 412 | 6587491 | melk | 450 | 2000000 |
| 412 | 1189244 | egg | 1660 | 2000000 |
| 412 | 323791 | kylling | 1560 | 2000000 |
| 412 | 4675 | lam | 2340 | 2000000 |
| 412 | 1805 | sau | 3710 | 2000000 |
| 412 | 719 | kalv | 1120 | 2000000 |
| 412 | 616733 | svin | 2680 | 2000000 |

Figure 4.67 A sample of the joined data. The Food production data table have been with the area of farmland table and the kilocalories per food table.

Final Calculation

From the joined data, the final calculation can either be done using the SELECT clause to output and look at the results, or changes can be made to the existing table to simplify the table to only the data necessary to answer the question. Using the query below, the data can be viewed temporarily:

```

1 SELECT municipality_id, (amount*kcal/area_of_farmland) AS kcal_produced_per_m2
2 FROM results
3 GROUP BY municipality_id;

```

This results in the output shown in Figure 4.68, where the kilocalories worth of food produced per municipality can be found. For instance, the municipality with ID of 513 has produced 1101 kilocalories worth of food.

| | municipality_id | kcal_produced_per_m2 |
|---|-----------------|----------------------|
| ▶ | 301 | 0.0127509 |
| | 412 | 10763.5 |
| | 419 | 993.948 |
| | 513 | 1209.75 |
| | 522 | 49.4251 |
| | 528 | 2810.92 |
| | 544 | 24.7939 |
| | 623 | 29.4106 |

Figure 4.68 Results of the final calculation. The kilocalories worth of food produced per square meter in each municipality can now be found from the table.

To change the *results* table to match this output, a new column *kcal_produced_per_m2* has to be added to the table and the *amount*, *type*, *kcal*, and *area_of_farmland* columns need to be dropped. This is all done in the queries shown below:

```

1 # Add a new column
2 ALTER TABLE results
3 ADD kcal_produced_per_m2 INT NOT NULL;
4
5 # Insert data in the column by updating the table
6 UPDATE results
7 SET kcal_produced_per_m2=amount*kcal/area_of_farmland;
8
9 # Drop irrelevant columns
10 ALTER TABLE results
11 DROP COLUMN amount,
12 DROP COLUMN type,
13 DROP COLUMN kcal,
14 DROP COLUMN area_of_farmland;

```

4.2.5 Summary of the case study of MySQL

The importing of the CSV files caused some problems. It seems the data has to be structured before it can be loaded into the database. This means the data has to be preprocessed before the data preparation can be done in MySQL. This is a weakness of working with data transformation inside a database. However, when the data is loaded, it means the data already has some structure and the rest of the data preparation process should be quite simple. MySQL comes with many simple functions, clauses and statements which makes the process easy. However, it lacks the capabilities for updating a table with the aggregated data. In fact, grouping, aggregating and unpivoting the data requires a new table to be created and the modified data to be inserted into the new table. This is a bit tedious. Dummy tables have to be created just to hold the data until the necessary

preparation steps have been performed, before the tables can be dropped again. Overall, the language is very readable, as it is very similar to the way that humans would explain a task. The declarative approach makes it easy for non-experts to read and understand the queries.

4.3 Results

In this section, the results of the case study will be presented. The two Data Manipulation Languages, Python pandas and SQL, studied in section 4.1 and 4.2 will be evaluated in terms of some dimensions. The dimensions are time-consumption, readability, usability, flexibility, and expressiveness. This evaluation can provide some insight into the differences and advantages/disadvantages with the two programming paradigms.

In general, both languages provide functionality for most of the data preparation tasks necessary in the case study. MySQL required some preprocessing of the data before the data could be loaded into the database, for instance, converting the XLSX file to a CSV file. In Python pandas, fuzzy matching was used to match the food types of the food production and nutrients table data. While in MySQL, the food types were manually changed to match. The categories included in the calculation therefore ended up being different. This also resulted in the final calculation being different using the two languages. This can be seen in Figure 4.69 and Figure 4.70.

| | komnr | kcal_produced_per_m2 |
|----------|--------------|-----------------------------|
| 0 | 301 | 0.0191 |
| 1 | 412 | 17611.4081 |
| 2 | 419 | 433.0908 |
| 3 | 513 | 2858.4526 |
| 4 | 522 | 94.9475 |

Figure 4.69 Results of final calculation using Python pandas

| Result Grid | | |
|-------------|------------------------|-----------------------------|
| | municipality_id | kcal_produced_per_m2 |
| | 301 | 0.0127509 |
| | 412 | 10763.5 |
| | 419 | 993.948 |
| | 513 | 1209.75 |
| | 522 | 49.4251 |
| | 528 | 2810.92 |
| | 544 | 24.7939 |
| | 623 | 29.4106 |

Figure 4.70 Results of final calculation using MySQL

The results still seem to be in the same range. This inconsistency could be fixed by matching the food types in the same way for both languages. Investigating the problems further by checking the original CSV files to see what the final values should be, could reveal where in the data preparation process the DMLs produces different results.

4.3.1 Time-consumption

The differences in time-consumption are clear from the start of the case study, as MySQL requires a local instance to be installed before the data preparation can begin. However, this is not always the case as MySQL can also be used in cloud environment where a ready-to-use MySQL data management service. Python pandas can easily be installed and used either in a notebook or an Integrated Development Environment (IDE). In the case study, Jupyter Notebook was used and installing and importing the library took only a few minutes. The Python pandas library was installed in the terminal using *pip*, which is a Python package installer (pypi), but it could also have been installed directly in the Notebook using also using *pip*. Python pandas works in-memory which compared to MySQL is faster.

Python pandas provides simple methods for reading the data files, which works seamlessly and fast. However, in MySQL loading the messy data files into the database is a challenge. Since the data is messy, the LOAD DATA INFILE statement raises several errors regarding missing values and wrong data types, and therefore fails to load the data. It seems the data has to be cleaned somewhat in advance in order to use this statement. Using the Table Data Import Wizard in MySQL Workbench, the data was loaded into the database without having to clean it in advance. However, this was very time-consuming, as mentioned in section 4.2.1. One of the six data files took 3-4 hours to load into the database. Some people suggest using LOAD DATA INFILE instead, as they claim it is faster. Cleaning the data a bit in advance and using the LOAD DATA INFILE statement can therefore be considered to decrease the time-consumption.

Apart from the loading of data, both Python pandas and MySQL performs the data preparation tasks fast. What might take some time, especially for non-experts, is finding the solutions for the different tasks from the documentation and forums online. Both Python

pandas and MySQL are well-documented and have large communities providing help for the inexperienced.

4.3.2 Flexibility

The pandas library does not have methods for all the data preparation tasks performed in this case study. However, Python still provides enough flexibility to solve the tasks by creating our own scripts. This requires a bit of work, but it is still quite simple to accomplish. Python also offers a lot of other libraries that can be used in combination with pandas, which adds functionality and makes the DML even more flexible. This includes for instance libraries that allows us to develop machine learning algorithms or perform regression tests and statistical analysis, which are common next steps after the data manipulation.

MySQL provides less flexibility, as it requires the data to be structured before it can be loaded into the database. Missing values and inconsistencies in data types causes problems. During the case study using Python pandas, there were no indications of problems with the data files. Missing values, for instance, caused no problems at all. Thus, Python seems to handle unstructured data much better than MySQL. This makes Python pandas more flexible than MySQL. Moreover, MySQL does not provide much more functionality beyond the data manipulation and simple calculations, such as the final calculation in the case study.

4.3.3 Expressiveness

The declarative approach of MySQL is more expressive, as a query specifies what data is wanted very clearly. This is fairly easy to understand even for inexperienced MySQL users. The procedural approach of Python pandas is a bit less expressive, as a line of code specifies how to get the data that is wanted. For an expert with programming experience, this approach might be obvious. However, non-experts might find it harder to understand what is happening in a line of code. If, for instance, we want to drop a column from a MySQL table or pandas DataFrame, this would be done using the query below in MySQL:

```
1 ALTER TABLE table_name
2 DROP COLUMN col1;
```

The MySQL query's function is clear: alter the table by dropping the column named "col1". Compared to the MySQL query, the Python pandas code is not as obvious, as seen in the line of code below. One might understand that something will be dropped from the DataFrame *df* in this line, but the *axis* parameter is not necessarily automatically interpreted as determining column or index, especially by non-experts.

```
1 df.drop('col1', axis=1)
```

More advanced Python pandas methods get even more difficult to understand for non-experts.

4.3.4 Usability

As previously mentioned, the setup of the two DMLs is different, and MySQL requires a bit more effort than Python pandas. This also affects the usability of the two DMLs. Installing a local instance of MySQL, creating tables for the data, loading the data, running one-time queries to prepare the data, and then deleting the tables again can be a bit tedious. Python pandas is more usable as it is easier to set up. It works in-memory, which also makes it faster and increases the usability. The methods provided in the pandas library are simple and often self-explanatory. MySQL also provides some simple clauses to perform data preparation tasks, but some of the tasks requires quite complex and long queries. For instance, grouping, aggregating and unpivoting data in MySQL requires a long query of several lines. Python pandas, on the other hand, uses one simple method to do each of these tasks which requires only one line of code. In MySQL, the grouped and aggregated data has to be loaded into a new table in order to get the data in the desired format permanently. The table cannot be updated with the grouped and aggregated data, which is also a bit tedious as dummy tables has to be created and dropped. However, this functionality might be available in other DBMS.

4.3.5 Readability

Both the procedural and declarative approach is somewhat easy to read. The simplest queries in MySQL might be considered a bit easier to read than Python pandas code, especially for non-programming-experts. The example in Section 4.3.3 comparing a SQL query and Python pandas code line for dropping a column also shows the differences in readability of the two DMLs. The example showed that the MySQL query is very human-readable, while the Python pandas code line required some more programming experience to interpret.

5 Conclusion

In this chapter the thesis work will be concluded. The conclusion will consider the contributions and research questions and discuss the findings from the review and analysis of DMLs. Some suggestions for future work will also be added at the end of the chapter.

5.1 Discussion

The research questions defined in Section 1.2 has been investigated through a review of common data transformation tools and a case study comparing declarative vs. procedural DMLs. In the following sections each research question will be discussed.

5.1.1 What is the support for common data preparation tasks provided by some of the most prevalent data transformation tools?

The results of the review in Chapter 3 showed that all the selected tools provide most of the data preparation tasks. The selected tools are some of the most prevalent tools for data manipulation, it is therefore not surprising that most of the data preparation tasks are available. The key findings of the review were the following:

- The programming languages (R, SQL, Python) more often requires workarounds and custom scripts for solving tasks, which is more time-consuming and requires more experience
- The applications, on the other hand, are very easy to understand and provide simple components/blocks that performs the data preparation tasks. No coding is required for most tasks, which makes it suitable for non-experts.
- Programming languages provide more flexibility, as tasks that are not supported by applications are sometimes impossible to accomplish in the applications. Programming languages allows for creating custom solutions to the tasks even if there are no simple methods available to solve them.
- The context of which the tool will be used can also be considered when selecting a tool. This might be interesting because where the data is stored and what tools and

technologies the organization already uses can affect what is the wisest choice of tool for data manipulation. If the data is stored in a relational database, such as MySQL, using SQL to perform the manipulation might be a good choice. Another alternative might be to use Python libraries which enables us to run SQL queries in a Jupyter notebook, for instance. In this case, SQL or Python might be better choices than R, for instance. Another thing to mention is that learning a tool from scratch is time-consuming. If the tool that is chosen can also be used for other parts of the data science project, such as developing and deploying ML algorithms, it might be a wise choice to spend time learning a tool that provides libraries that has these capabilities.

To summarize, the main factors to consider is experience and context. A non-expert might find an application a better choice if the necessary functionalities is available. An expert might find it is better to use tools which are more flexible, even for simple tasks, because they also offer functionalities for developing ML algorithms or are particularly well-suited for data analysis.

5.1.2 How does declarative vs. procedural DMLs differ in terms of time-consumption, flexibility, expressiveness, usability, and readability?

Based on the case study, Python pandas seems to be a better choice for data manipulation. An overview of the main advantages and disadvantages of the DMLs found in the case study are shown in Table 5.1. The main reasons why Python pandas is considered a better choice than MySQL is that the time consumption is less, and the flexibility and usability is better. This is due to the ease of setting up Python pandas compared to MySQL, the wide range of methods provided with the library, and the availability of additional libraries offering more functionalities and increasing the flexibility.

The expressiveness of MySQL queries is good, and especially well-suited for non-experts. However, the DML is less flexible, provides less functionalities, and is not suitable for complex computations. The requirement to install a local instance of MySQL also decreases the usability of the DML.

Table 5.1 Advantages and disadvantages of Python pandas and MySQL

| | Advantages | Disadvantages |
|---------------|--|---|
| Python pandas | <ul style="list-style-type: none"> (1) Setup is simple and fast (2) Relatively easy to learn | <ul style="list-style-type: none"> (1) Requires some programming experience (2) Not very readable, especially for non-experts |
| MySQL | <ul style="list-style-type: none"> (1) Expressiveness is good, especially for non-experts (2) Readability is good for simple queries, but becomes poorer the more complex the query gets (3) Relational databases provide powerful data management capabilities | <ul style="list-style-type: none"> (1) Installation of local instance is required (2) Provides less functions for data preparation tasks and for the execution of complex computation compared to pandas (3) Requires some preprocessing in order to load the data into the database |

Considering the framework defined in Table 1.1 in Section 1.3.1, the data preparation tasks performed in the case study is reviewed to provide some insights into the differences of the DMLs. The results are shown in Table 5.2.

Table 5.2 Reviewing the functionalities of MySQL and Python pandas

| Data preparation tasks | Comparison |
|--------------------------|--|
| Drop irrelevant columns | Both Python pandas and MySQL provide simple functions for dropping a column. Python requires some parameters to specify which column to drop, and also whether a column of row should be dropped. |
| Remove null/missing data | Null/missing values can be removed easily using both languages. The expressiveness of SQL is better than Python pandas, as previously mentioned. However, Python pandas also provides a simple way to filter the data. |
| Group data | Python pandas allows for grouping the data permanently, while SQL requires that a new table is created and the grouped data is inserted into the new table. This is a bit more tedious than the Python pandas approach. |
| Aggregate data | The aggregation in SQL also requires a new table to be created and the aggregated data has to be inserted into the new table. Python pandas provides a simple method agg() that take in what column to aggregate and what aggregation |

| | |
|-----------------|---|
| | function to apply. It can all be done in one code line, compared to SQL that requires several lines of code. |
| Unpivot | Python pandas provides a simple method, melt() , to unpivot the data. The method takes in the columns to keep as they are, the columns to convert to rows, and new column labels for the unpivoted data. In SQL this task is much more complicated, as each of the columns has to be selected one by one and then be unioned to create an unpivoted table. This requires several lines of code, as compared to Python pandas that solves this in one code line. |
| Split columns | This task is solved in a similar way in Python pandas and MySQL. Python pandas has a method, <code>split()</code> , which takes in what to split on and returns a list of the substrings. Similarly, in MySQL, <code>SUBSTRING_INDEX</code> takes in the column to split, the string to split on, and an index of the desired substring. This function returns the selected substring directly, as opposed to Python pandas' <code>split()</code> method. Both languages require that a new column is created and the column is set equal to the desired substring. |
| Merging columns | This task can be performed similarly in Python pandas and MySQL. In the case study, the merging of columns is done only in the MySQL part. This was done by adding a new column and updating this column with values from the old columns based on a condition. This can be accomplished in Python pandas as well, for instance by setting a new column equal to the two old columns added together simply using the '+' operator. However, this requires the data types to be the same. If they are not, the data type needs to be changed. |
| Rename column | This task has only been performed using Python pandas but is performed just as easily by using MySQL's <code>RENAME</code> function and providing the old and new name of the column. |
| Add column | Both Python pandas and SQL provides simple functions to accomplish this. The only difference is the expressiveness in the declarative and procedural approach, as mentioned previously. |
| Drop rows | In Python pandas rows can be filtered out and the filtered data is stored in a new dataframe. In SQL the <code>DELETE</code> and <code>WHERE</code> clause can be used to delete the data where a specific column is equal to a specific value. Other operators |

| | |
|----------------------|--|
| | such as greater than, less than and not equal to, can also be used. |
| Changing letter case | In Python pandas, changing the letter case requires adding a new column, extracting the string from the old column and applying the lower() method to the string, and saving this new lower case string to the new column. In SQL, this is done in a similar way by adding a new column, using the LOWER() String function to change letter case of the old column and add the lower case string to the new column. |
| Join/Union data | Python pandas provides several methods for joining and unioning data. In the case study, the concat() method is used to union data. This method can be used to join data as well, by changing the <i>axis</i> parameter. Only one code line is required. In MySQL, the same process has to be done in several steps and lines of code. A new table has to be created, the data can then be unioned or joined using the UNION or JOIN clauses. Lastly, the joined/unioned data has to be inserted into the new table. This approach is much more tedious than Python pandas. |

Most of the data preparation tasks in Table 5.2 are solved in similar and simple ways for both DMLs. However, there are differences in the grouping, aggregation, unpivoting, joining and unioning data. These tasks are easily performed in Python pandas using one simple function and only requires one code line. MySQL, however, requires the manipulated data to be inserted into a new table, and the query becomes quite long. This is a bit more time-consuming and it also reduces the readability a bit. Readability and expressiveness are what seems to be advantages of MySQL. However, Python pandas provides simpler solutions for more of the data preparation tasks. Based on this, Python pandas is considered slightly better than MySQL.

It's worth mentioning that these tools are often used in combination, and the decision should be made considering the system in which the data science project will be conducted. However, this thesis gives an overview of the strengths and weaknesses of the DMLs. Thus, using this work, a data scientist or data engineer can get an overview of which steps in the data preparation process the DMLs are suited for.

5.2 Future work

In future work one might consider applications as well in a case study comparing DMLs to applications. Some of the literature reviewed in the background chapter have already compared different applications' performance and features (Hameed & Naumann, 2020), (Petrova-Antonova & Tancheva, 2020). Evaluating the usability of applications compared to DMLs could be interesting, as they seem very user-friendly and doesn't require users to be experienced programmers. The flexibility should also be considered, as the applications might limit the functionality of the tools to a set of preparation tasks. Complex data preparation tasks might be impossible to perform. This could reveal if the high usability reduces the flexibility of the applications.

In the case study, only a few simple data preparation tasks are evaluated. Testing the usability and flexibility further by reviewing some of the more complex data preparation tasks could be considered in future work. This would provide some additional insights into the performance of the tools in more advanced data preparations. One example is fuzzy matching in SQL. In Python the library *thefuzz* was used to do fuzzy matching. Creating an algorithm for calculating the *Levenshtein distance* (Wikipedia, 2022) using SQL could be interesting.

6 References

- Altair. (n.d.). Altair Monarch. Retrieved from <https://web.altair.com/monarch-free-trial>
- aunalytics. (n.d.). Understanding Analytics Part 1: Top Internal Sources of Big Data. Retrieved from <https://www.aunalytics.com/understanding-analytics-part-1-top-internal-sources-of-big-data/>
- Consulting, I. (n.d.). General Data Protection Regulation. Retrieved from <https://gdpr-info.eu/>
- contributors, W. (2021). Data Validation. In: Wikipedia, The Free Encyclopedia.
- Convertino, G., & Echenique, A. (2017). *Self-service data preparation and analysis by business users: New needs, skills, and tools*. Paper presented at the Proceedings of the 2017 CHI Conference Extended Abstracts on Human Factors in Computing Systems.
- DataRobot. (n.d.). Data Preparation. Retrieved from https://www.datarobot.com/platform/dataprep/?redirect_source=paxata.com
- Facer, C. (n.d.). What is Data Filtering? Retrieved from <https://www.displayr.com/what-is-data-filtering/>
- Foundation, T. R. (n.d.). Documentation. Retrieved from <https://www.r-project.org/other-docs.html>
- Green, A. (2021). Complete Guide to Privacy Laws in the US. Retrieved from <https://www.varonis.com/blog/us-privacy-laws>
- Hameed, M., & Naumann, F. (2020). Data preparation: A survey of commercial tools. *ACM SIGMOD Record*, 49(3), 18-29.
- Jupyter. (2022). Jupyter Notebook. Retrieved from <https://jupyter.org/>
- Laird, J. (2021). The GDPR vs China's PIPL. Retrieved from <https://www.privacypolicies.com/blog/gdpr-vs-pipl/>
- Laney, D. (2001). 3D data management: Controlling data volume, velocity and variety. *META group research note*, 6(70), 1.
- MySQL. (2021). MySQL Workbench 8.0. Retrieved from <https://www.mysql.com/products/workbench/>
- MySQL. (n.d.-a). 12.8 String Functions and Operators. *MySQL 8.0 Reference Manual*. Retrieved from <https://dev.mysql.com/doc/refman/8.0/en/string-functions.html>
- MySQL. (n.d.-b). 12.20.2 GROUP BY Modifiers. Retrieved from <https://dev.mysql.com/doc/refman/8.0/en/group-by-modifiers.html>
- MySQL. (n.d.-c). 13.1.9 ALTER TABLE Statement. Retrieved from <https://dev.mysql.com/doc/refman/8.0/en/alter-table.html>
- MySQL. (n.d.-d). 13.2.2 DELETE Statement. *MySQL 8.0 Reference Manual*. Retrieved from <https://dev.mysql.com/doc/refman/8.0/en/delete.html>
- MySQL. (n.d.-e). 13.2.7 LOAD DATA Statement. *MySQL 8.0 Reference Manual*. Retrieved from <https://dev.mysql.com/doc/refman/8.0/en/load-data.html>
- MySQL. (n.d.-f). 13.2.10 SELECT Statement. *MySQL 8.0 Reference Manual*. Retrieved from <https://dev.mysql.com/doc/refman/8.0/en/select.html>
- MySQL. (n.d.-g). 13.2.10.3 UNION Clause. *MySQL 8.0 Reference Manual*. Retrieved from <https://dev.mysql.com/doc/refman/8.0/en/union.html>
- MySQL. (n.d.-h). 13.2.13 UPDATE Statement. *MySQL 8.0 Reference Manual*. Retrieved from <https://dev.mysql.com/doc/refman/8.0/en/update.html>
- MySQL. (n.d.-i). 13.6.5.1 CASE Statement. Retrieved from <https://dev.mysql.com/doc/refman/5.7/en/case.html>
- MySQL. (n.d.-j). Aggregate Function Descriptions. *MySQL 8.0 Reference Manual*. Retrieved from <https://dev.mysql.com/doc/refman/8.0/en/aggregate-functions.html>
- OpenRefine. (n.d.). OpenRefine. Retrieved from <https://github.com/OpenRefine/OpenRefine>
- Özsu, M. T. (2017). Data Manipulation Language (DML). In L. Liu & M. T. Özsu (Eds.), *Encyclopedia of Database Systems* (pp. 1-2). New York, NY: Springer New York.

Pandas. (n.d.-a). pandas documentation. Retrieved from <https://pandas.pydata.org/docs/>

pandas. (n.d.-b). pandas.concat. Retrieved from <https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.concat.html?highlight=concat#pandas.concat>

pandas. (n.d.-c). pandas.DataFrame.agg. Retrieved from <https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.agg.html?highlight=agg#pandas.DataFrame.agg>

pandas. (n.d.-d). pandas.DataFrame.any. Retrieved from <https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.any.html?highlight=any#pandas.DataFrame.any>

pandas. (n.d.-e). pandas.DataFrame.copy. Retrieved from <https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.copy.html?highlight=copy#pandas.DataFrame.copy>

pandas. (n.d.-f). pandas.DataFrame.drop. Retrieved from <https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.drop.html?highlight=drop#pandas.DataFrame.drop>

pandas. (n.d.-g). pandas.DataFrame.groupby. Retrieved from <https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.groupby.html?highlight=groupby#pandas.DataFrame.groupby>

pandas. (n.d.-h). pandas.DataFrame.head. Retrieved from <https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.head.html?highlight=head#pandas.DataFrame.head>

pandas. (n.d.-i). pandas.DataFrame.iloc. Retrieved from <https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.iloc.html?highlight=iloc#pandas.DataFrame.iloc>

pandas. (n.d.-j). pandas.DataFrame.info. Retrieved from <https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.info.html?highlight=info#pandas.DataFrame.info>

pandas. (n.d.-k). pandas.DataFrame.isnull. Retrieved from <https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.isnull.html?highlight=isnull#pandas.DataFrame.isnull>

pandas. (n.d.-l). pandas.DataFrame.melt. Retrieved from <https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.melt.html?highlight=melt#pandas.DataFrame.melt>

pandas. (n.d.-m). pandas.DataFrame.rename. Retrieved from <https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.rename.html?highlight=rename#pandas.DataFrame.rename>

pandas. (n.d.-n). pandas.DataFrame.reset_index. Retrieved from https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.reset_index.html?highlight=reset_index#pandas.DataFrame.reset_index

pandas. (n.d.-o). pandas.DataFrame.shape. Retrieved from <https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.shape.html?highlight=shape#pandas.DataFrame.shape>

pandas. (n.d.-p). pandas.DataFrame.tail. Retrieved from <https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.tail.html?highlight=tail#pandas.DataFrame.tail>

pandas. (n.d.-q). pandas.DataFrame.tolist. Retrieved from <https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Series.tolist.html?highlight=tolist#pandas.Series.tolist>

pandas. (n.d.-r). pandas.Index.map.

pandas. (n.d.-s). pandas.Index.notnull. Retrieved from <https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Index.notnull.html?highlight=notnull#pandas.Index.notnull>

pandas. (n.d.-t). pandas.merge. Retrieved from <https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.merge.html?highlight=merge#pandas.merge>

pandas. (n.d.-u). pandas.read_csv. Retrieved from https://pandas.pydata.org/docs/reference/api/pandas.read_csv.html

pandas. (n.d.-v). pandas.read_excel. Retrieved from https://pandas.pydata.org/docs/reference/api/pandas.read_excel.html

pandas. (n.d.-w). pandas.Series.round. Retrieved from <https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Series.round.html?highlight=round#pandas.Series.round>

pandas. (n.d.-x). pandas.Series.str.split. Retrieved from <https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Series.str.split.html?highlight=str%20split#pandas.Series.str.split>

pandas. (n.d.-y). pandas.Series.unique. Retrieved from <https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Series.unique.html?highlight=unique>

Patil, M. M., & Hiremath, B. N. (2018). A systematic study of data wrangling. *Int. J. Inf. Technol. Comput. Sci. (IJITCS)*, 1, 32-39.

Petrova-Antonova, D., & Tancheva, R. (2020). *Data Cleaning: A Case Study with OpenRefine and Trifacta Wrangler*. Paper presented at the International Conference on the Quality of Information and Communications Technology.

pip. (n.d.). pip 22.0.4. Retrieved from <https://pypi.org/project/pip/>

Python. (n.d.-a). array.remove(x). Retrieved from <https://docs.python.org/3/library/array.html?highlight=remove#array.array.remove>

Python. (n.d.-b). classmethod fromkeys(iterable[,value]). Retrieved from <https://docs.python.org/3/library/stdtypes.html?highlight=fromkeys#dict.fromkeys>

SAP. (n.d.). SAP HANA. Retrieved from <https://www.sap.com/norway/products/hana.html>

SAS. (n.d.). SAS Data Preparation. Retrieved from https://www.sas.com/en_us/software/data-preparation.html

seatgeek. (n.d.). TheFuzz. Retrieved from <https://github.com/seatgeek/thefuzz>

Snowflake. (2022). What is a data pipeline? Retrieved from <https://www.snowflake.com/guides/data-pipeline>

Stackoverflow. (2019). MySQL workbench table data import wizard extremely slow. Retrieved from <https://stackoverflow.com/questions/33296569/mysql-workbench-table-data-import-wizard-extremely-slow>

Tableau. (n.d.). Tableau Prep. Retrieved from <https://www.tableau.com/products/prep>

Tabula. (n.d.). Retrieved from <https://tabula.technology/>

Talend. (n.d.). Talend. Retrieved from <https://www.talend.com/>

thdoan. (n.d.). Mr Data Converter. Retrieved from <https://github.com/thdoan/mr-data-converter>

TIBCO. (n.d.). What is Data Discovery? Retrieved from <https://www.tibco.com/reference-center/what-is-data-discovery>

Trifacta. (n.d.-a). Data Enrichment. Retrieved from <https://www.trifacta.com/7-ways-data-enrichment-boosts-your-business/>

Trifacta. (n.d.-b). Trifacta. Retrieved from <https://www.trifacta.com/>

W3Schools. (n.d.). SQL Tutorial. Retrieved from <https://www.w3schools.com/sql/>

White, N. (2020). Data Transformation Tools: Application or Code? Retrieved from <https://www.linkedin.com/pulse/data-transformation-tools-application-code-nick-white/>

Wickham, H. (2014). Tidy data. *The American Statistician*, 14. doi:10.18637/jss.v059.i10

Wikipedia. (2022). Retrieved from https://en.wikipedia.org/w/index.php?title=Levenshtein_distance&oldid=1082661551

Zach. (2021). Long vs. Wide Data: What's the Difference. Retrieved from <https://www.statology.org/long-vs-wide-data/>

List of Appendices

The appendices are included in the .zip folder submitted with the thesis. Appendix A is the Jupyter Notebook with all the Python pandas code from the case study. Appendix B is all the MySQL-queries used in the case study. The appendices with the file names are listed below:

- Appendix A Case_study_of_Python_pandas.ipynb
- Appendix B mysql_review.sql