Data Cleansing for Food Composition Data

Data Cleansing for Food Composition Data

Master Thesis

Alexandra Hochuli
<hochulia@student.ethz.ch>

Prof. Dr. Moira C. Norrie
David Weber

Global Information Systems Group
Institute of Information Systems
Department of Computer Science

7th April 2014

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

globis

Abstract

This Master thesis gives an overview of data cleansing problems by analysing the Swiss Food Composition Database (SFCD). Databases can contain incorrect, incomplete or duplicate entries. Such problems need to be cleaned. Existing problems of the SFCD data are presented and solutions using data cleansing methods and approaches are discussed to solve them. Additionally, open source tools for data cleaning are presented that could be useful to clean data automatically. Furthermore, a Food Matching Tool is presented which was developed during this thesis. The tool matches food items of two food sets by only using the names of the foods and food classifications. A food can for example be classified with categories like fruit, vegetable or milk product. This is useful information to match foods. The matches proposed by the tool can then be used to find duplicates in a food database or to enhance food data of one data source with food data of another source.

Contents

1 Introduction

Data cleansing is about correction and improvement of data in a database. In an ETL (Extract-Transform-Load) process, where data from different data sources is extracted, transformed and then loaded into a new database, data cleansing is of high importance. An example is shown in Figure 1.1 where customer data from two data sources is extracted and loaded into a new database table. The person James Black appears in both data sources, but the last name is written differently. The birthday is the same date in both sources, but the date format is different. In the new database table the customer James Black appears now twice, but once with a typing error in the last name and the date format is not consistent.

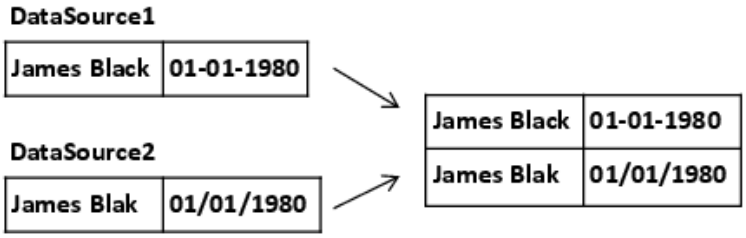

Figure 1.1: Duplicates in ETL process

The goal of data cleansing in this example would be to correct the typing error and to transform the dates into the same format, such that duplicate customers can be detected and eliminated. The duplicate elimination of customers reduces the costs of a company because less letters to the customers have to be sent. Additionally, customers are not annoyed by receiving the same letter several times. Another advantage of duplicate elimination is that statistics about the customers can be improved, as for example the average age of customers is not correct with duplicates in the database.
Data cleansing operates on the data which is already in the database. To avoid inconsistencies in the format, such as the date of birth in Figure 1.1, the text fields where the data is entered into the system or database should be validated, such that only a certain format of the text is

allowed. Another validation could be, that only numbers or only letters are allowed in a text field. But even if the data validation is realised carefully, certain errors are not avoided, such as typing errors or if data is entered several times to the database which results in duplicates within the database. Therefore, data cleansing is still needed even though data validation is present.

1.1 Motivation for Data Cleansing

As the customer example shows, it is important that there are no duplicates in a database, i.e. that a real world item, for example a customer, should not be represented by more than one database entry. Usually, the database entries that represent the same real world object are not exactly the same. They can contain for example spelling errors in the street name or the telephone numbers do not have the same format.
In [7] data cleansing problems and anomalies are discussed. The following list presents some of them and possible solutions are mentioned:

- Typographical errors: As shown in the introduction example in Figure 1.1 misspellings can happen when the data is entered into the system. Typing errors could be found by comparing the words to a reference dictionary.
- Standardisation, harmonisation: In the whole data set that should be cleaned, the same dictionaries and abbreviations should be used for same real world objects. An example is shown in Figure 1.2 where the same unit is represented with two different words, namely 'mg'/'milligram' and 'g'/'gram'.

Nutrient	Value	Unit
nutrient1	100	mg
nutrient2	224	milligram
nutrient3	4	gram
nutrient4	1	g

Figure 1.2: Units are not standardised

- Uniformity: The same units and currencies should be used within a database table column, such that they could be compared directly. In Figure 1.2 for example, some values are given in milligrams, others in grams. These values need to be transformed into the same unit first, before they can be compared.
- Data enhancement: Additional data is added to a database entry. This data could be obtained from external sources by identifying that it is the same real world item, or by merging information with another database entry, which represents the same real world item.
- Analysis of database schema: Data types of columns should be appropriate, e.g. if the field expects a number, the type should be number, not text. If values should be within

a certain range or part of a certain set or should not be null, corresponding database constraints should exist. For each column or combination of columns which should be unique, a unique constraint should exist. All possible foreign keys should be set.

- Patterns: If a field should only accept values of a certain pattern, a regular expression can be created to ensure this pattern.
- Cross-field validation, attribute dependencies: These are constraints including several columns of a database table. Figure 1.3 shows an example where the values of the columns 'Birthday' and 'Age' are correlated.

Birthday	Age
01.01.1990	24
01.01.1992	24

Figure 1.3: Birthday and age are correlated

- Outlier detection: Outliers, as 'mg' or '10134' of Figure 1.4, are often indicators for errors in a database. Therefore, one of the goals of data cleansing is to find outliers.

Value
124
189
348
mg
799
...

Value
2
8
10134
4
10
...

Figure 1.4: Outliers

1.2 Motivation for this Master Thesis

There exist quite a lot of data cleansing tools with focus on address data. This master thesis focuses on food composition data by analysing the SFCD and proposing solutions how to clean the data cleansing issues and anomalies that were found. The tool developed during this master thesis is a Food Matching tool, which matches food items of two food sets. The resulting matches could be used to find duplicates in the database, for example after an integration of multiple food sets, or to enhance the food data of one set with the information provided by the other set.

1.3 Structure

In Chapter 2 the theory about data cleansing, including methods and tools used for data cleansing, and theory about food composition data is explained. In Chapter 3 the SFCD is analysed and data cleansing problems of the SFCD data are presented. Chapter 4 describes the approach of the Food Matching tool, which was developed during this master thesis. Chapter 5 shows implementation issues of the Food Matching tool and Chapter 6 explains how the tool could be extended. Chapter 7 presents evaluation results of matching two food sets using the Food Matching tool. In Chapter 8 methods and decisions which were needed for the realisation of the tool are justified. Finally, Chapter 9 summarises the work done during the master thesis, explains the contribution again, and presents future work.

2 Background

In this chapter general theory about data cleansing is presented, including the data cleansing process, methods and tools for data cleansing. Furthermore, an introduction to food composition data and the SFCD is given.

2.1 Data cleansing process

In this section the data cleansing process is described. To clean data of a database, several steps and tasks are needed, each responsible for one data cleansing issue of the data. The order in which these tasks are executed is essential to optimise the process and to not introduce new problems in the data. In [8] and [11] the data cleansing process is explained which is summarised in the following.

2.1.1 Data auditing

At the beginning of the process, the data in the database is analysed to find problems in data quality. For this analysis, for example statistical methods like mean and standard deviation could be used to find outliers in a database column. Another method to analyse the data is to use SQL queries on the database. The goal of this process step is to find anomalies or contradictions in the data.

2.1.2 Workflow specification

As soon as the problems of the data are detected, data cleansing steps can be defined for each problem. The sequence in which these steps should be executed, also called the workflow, needs to be specified. The optimal workflow can be found by clearly analysing the cause of the errors and the anomalies. When the data, that should be cleaned, comes from multiple databases, it is recommended to clean first the data within each database, then to integrate the data from the multiple databases, and finally to clean the integrated database [8].

2.1.3 Workflow execution and verification

The specified workflow is executed on samples of the data or a copy of the data to test the workflow. During the cleansing steps the data can be replaced, modified or deleted. The results of the execution are verified and analysed. If the results are not satisfying, the workflow is improved by specifying new or adapted transformations and is executed again. This is done iteratively until the results are good. The workflow should also be efficient as for large data sets the cleansing process takes a lot of time.

2.1.4 Post-processing and backflow of data

Finally the workflow is executed on the whole data set. The results are also verified and can be corrected manually if necessary. The data of the original database should be replaced with the cleaned data, such that applications that are using the original data have access to the cleaned data. This also avoids re-cleaning of the same data in future data extractions.

2.2 Methods

In this section possible data transformations and machine learning methods that can be used for data cleansing are presented.

2.2.1 Parsing

Regular expression parsing can be used to find syntax or format errors in database entries [11]. When dealing with schema integration, parsing can be used to match columns of different databases or tables, for example by comparing the words or values that are written in a column, or by comparing the labels of different columns. Regular expressions can be learnt by 'learning by example' techniques as described in Potter's Wheel [9]. Hidden Markov models can be used to learn how probable a regular expression is in the training data [3]. However, to capture all the possible syntax and format errors of the database entries, a huge amount of regular expressions is needed, which is expensive in the learning phase and the evaluating phase of new data.

2.2.2 Data transformations

In data cleansing, decomposing and reassembling data is an important part [6]. Data transformations are applied to get a more unified form of the data, such that the data entries are better comparable. This is useful for example to find duplicates in the data set or to get more representative statistics from the data. Possible transformations are for example to map data into a new format [11], to split free-form attributes if several attributes were written into the same field, or to split or merge, also called unfold and fold, attributes [8]. Other transformations are standardisation, such that standard codes are used, and normalisation where values are transformed for example into the same unit.

2.2.3 Duplicate elimination

Duplicate elimination, also called deduplication, is the process of finding and eliminating duplicate data entries. An ideal database should contain exactly one representation of a real

world object. If a database contains more than one representation of one real world object, these representations are called duplicates. To find duplicates in a database, first similar entries have to be found [8]. Often this is done by mapping the data to keys, such that similar entries are mapped to the same key [11]. If the keys are chosen well, it is assumed that duplicates only occur among the entries which were mapped to the same key, such that the search space for duplicates becomes much smaller. In [3] the same principle is described as blocking, where entries are hashed, based on their attributes, and duplicates are searched only within entries of the same hash bucket. In [4] the basic sorted neighbourhood method is described, where keys for example are created by some letters from each attribute, then the keys are sorted and compared within a sliding window. For comparison some rules using equational theory can be defined to find out, if the records of the corresponding keys are similar. In [4] a multi-pass approach is also described, which generates several keys for one record and uses the transitive closure to find duplicate candidates.
If duplicates could be found, the records have to be merged somehow into one new record. One possibility is to pick the value that appears more frequently or the value of a more authentic source [3].

2.2.4 Statistical methods

Statistical methods like mean, standard deviation, ranges or clustering can be used to find unexpected values, correct values, or fill in an average value in records where the value is missing [11]. To find outliers, the Chebyshev's theorem can be used, which says, that values should lie within a certain number of standard deviations from the mean [6]. Another method to find outliers is using association rules [6]. Association rules are rules that define dependencies between several attributes. An ordinal association rule, which describes dependencies between attributes with the operators $<=$, $=$, $>=$, has confidence c if for $c\%$ of the records the rule holds. To find outliers rules with high confidence, e.g. 98%, are considered.

2.2.5 Similarity methods

To compute the similarity of strings or sets, different similarity methods and metrics exist . In [3] some of them are presented. In the following these similarity methods are summarized:

- The **Jaccard similarity** is a common method to measure the similarity of two sets A and B. It uses the intersection and the union of the two sets. Formally, the Jaccard similarity is defined as $sim(A, B) = \frac{|A \cap B|}{|A \cup B|}$.
- The **Levenshtein distance**, also called edit distance, is a similarity method for strings. The number of operations is counted to transform one string into the other one using insertion, deletion and replacement of characters [10].
- **N-grams** are used to measure the similarity of strings. The strings are divided in shorter strings of length N. These shorter strings, also called N-grams, build a set, such that the original string is represented by a set of strings. Using 3-grams for example, the string 'apple' is transformed into the set {'app','ppl','ple'}.
- Soundex, which determines the similarity of strings based on the phonetic, or the cosine distance are other similarity methods.

2.3 Data cleansing tools

While analysing the different data cleansing techniques, also several data cleansing tools were investigated in the context of this Master thesis. The focus was on open source tools, because if the SFCD turned out to be cleansable with a data cleansing tool, one of these tools could have been used or extended to clean th SFCD data. Therefore, the functionalities they provide and the data cleansing problems they deal with were studied.

2.3.1 SQL Power DQguru

The SQL Power DQguru[1] is an open source tool for data cleansing. The user can select a database table he wants to clean. The tool provides several data transformations to transform the values of a database column, such as Lowercase, Uppercase, Empty String to Null, Boolean to String, Trim Spaces, Word Count, Concatenation, Substring and Sort Words. Deduplication is also provided. For the deduplication, the user has to specify which columns of the database table should be considered. If the values of two database entries are equal in all of these columns, the entries are duplicates according to the tool. Additionally, SQL Power DQguru provides address cleansing. After the cleansing, the tool writes the cleaned data back into the same database table, such that the original table is modified. If the original table should not be modified, the user should copy the table first and then apply the data cleansing tool onto this copied table.

2.3.2 OpenRefine

OpenRefine[2] is an open source data cleansing tool developed by Google. The data can be imported from files with comma separated values amongst others, but it is not possible to directly load the data from a database. However, after importing the data it is represented in columns. The tool provides several data transformations, such as Lowercase, Uppercase or Trim whitespaces. The values of a column can be clustered using different distance methods which the user can choose. By using this clustering functionality, the user could detect near duplicates. However, the clustering can only be applied on the values of one column at the time, such that it is not possible to find duplicates considering multiple columns as in the SQL Power DQguru. OpenRefine does not provide deduplication. Nevertheless, the tool provides other interesting functionality, such as enhancement. The data can be enhanced by using web services or Freebase[3], which is an online database that anyone can use. After the data has been transformed by the OpenRefine tool it is stored to a file again.

2.3.3 Other tools

Another tool that may be useful for data cleansing tasks is DataWrangler [5], which can be used to reorder and transform data. Possible transformations are proposed to the user based on a ranking of previous transformations. In Chapter 8 of [1] more data cleansing tools are discussed.

[1] http://www.sqlpower.ca/page/dqguru
[2] http://openrefine.org
[3] http://www.freebase.com

2.4 Food Composition Data

Food Composition Data is the data which describes what nutrients are contained in a food, and how much of a nutrient is contained in the food. Examples for nutrients are carbohydrate, fat, protein, water, alcohol, vitamins, mineral nutrients.

2.4.1 Swiss Food Composition Database

The following list shows an overview of the relevant SFCD tables for this master thesis:

- tblcomponent: This table contains the nutrients, or also called components.
- tblsinglefood: This table contains basic foods, such as apple, pear or yoghurt. These foods are called single foods in the SFCD.
- tblsinglefoodcomponent: This table is the mapping table between single foods and components. It contains the information of what components are contained in a single food and how much of a component is contained in the single food.
- tblaggrfood: This table contains more complex foods, such as fruits. These foods are called aggregated foods in the SFCD. An aggregated food consists of several single foods.
- tblaggrcontributingvalue: This table is the mapping table between the aggregated foods and the single foods of which they consist.
- tblaggrfoodcomponent: This table is the mapping table between aggregated foods and components. It contains the information of what components are contained in an aggregated food and how much of a component is contained in the aggregated food.

2.5 Total Diet Study

Total Diet Studies analyse contaminants in the food.[4] The contaminants are also called substances in the following.

2.6 FoodCASE

FoodCASE is a research project at the ETH Zurich as well as a software for the management of scientific food composition information.[5] The SFCD is managed by this software.

2.6.1 Data Quality Analysis Tool

The Data Quality Analysis (DQA) Tool is a tool of FoodCASE. Running the tool on the SFCD data, checks data quality requirements for the data, which were specified previously. As a result the tool shows statistics which requirements are fulfilled by the data, and which ones are not.

[4] http://www.tds-exposure.eu
[5] http://www.foodcase.ethz.ch/index_EN

3

Data quality issues of the Swiss Food Composition Database

The SFCD was analysed using SQL queries directly on the database tables and using the DQA Tool described in Section 2.6.1.

3.1 General problems

In general, the data quality of the SFCD is good. However, some data cleansing issues could be found which are presented in the following.

3.1.1 Rounding problems

In the table tblsinglefoodcomponent for example, the selected value (singlefoodcompselectedvalue) is rounded, but the minimum value (singlefoodcompminimum) and maximum value (singlefoodcompmaximum) are not rounded. If the selected value and the minimum value are originally the same values, and then the selected value is rounded down, the selected value gets smaller than the minimum value. The same problem occurs when other values, such as mean, median, standard deviation or standard error of this table, are combined.
These rounding inconsistencies also exist in the table tblaggrfoodcomponent.

3.1.2 Quotation marks as values

In the samples table (tblsinglefoodcomponentsample) for example, some attributes of type 'character varying' contain two quotation marks (' ') instead of a description of sample reason (singlefoodcompsamplereason) or sample handling (singlefoodcompsamplehandling). Probably this indicates that the value of such an attribute is the same as the value of the item which was entered into the system before.

The same phenomenon occurs in the table tblsinglefoodcomponent for the attribute aggrfoodcompidconfidencecode.

3.1.3 English names missing

In the tables tblsinglefood and tblaggrfood there are foods which do not have an English name even though this attribute is mandatory.

3.1.4 Different formats

In the table tblreference, the publication date (referencepublicationdate) is not in a consistent format. Sometimes the date is written with timezone, sometimes without.
In the table tblaggrcontributingvalue, the value weights (aggrcontrvalueweight) are not in the same format. Some values are given as '1', others as '1.0'.

3.1.5 Null in relationship table

In some tables which represent a relationship between two other tables, like tblsinglefoodcomponentreference which maps singlefoodcomponents to references and vice versa, there exist some rows where one side of the mapping is null as shown in Figure 3.1. The same problem occurs in the table tblaggrcontributingvalue, which maps singlefoodcomponents to aggrfoodcomponents. The origin of this problem is probably based on half cascading deletion of singlefoodcomponents or aggrfoods.

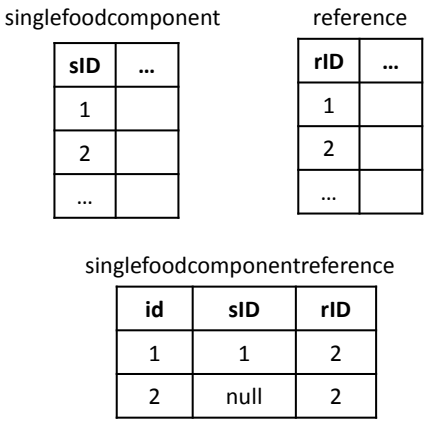

Figure 3.1: Null entry in a relationship table

3.1.6 Redundancy of samples and methods

In the samples table (tblsinglefoodcomponentsample) and the methods table (tblmethod) the same real world objects are stored several times. The reason for this redundancy is the design of the FoodCASE database schema. As shown in Figure 3.2 a sample is referred only to one singlefoodcomponent. Therefore, even if the samples for two different singlefoodcomponents

are the same, they have to be entered twice into the samples table. The same problem exists for the methods, which are also only referred to one singlefoodcomponent.

singlefoodcomponentsample

id	place	samplehandling	cID
1	Spain	freeze-drying, milling	1
2	Spain	freeze-drying, milling	2

singlefoodcomponent

cID	...
1	
2	
...	

Figure 3.2: Database schema problem for samples

3.1.7 Misfielded data

In the recipe table (tblrecipe) a recipe has a reference attribute which indicates where the measurements of the corresponding recipe are documented. For some recipes this reference attribute is left empty and the reference is added to the recipe name like 'siehe ...' (German for 'see ...').

3.1.8 Legacy data

The analysis of the SFCD with the DQA Tool described in Section 2.6.1 shows that some database fields are null, even though they should not be empty. The reason for this is legacy data, which means that older data was imported to the newer database from another database where this information was not stored. To get the missing information, someone has to extract the required information from the research papers and documentation of the food measurements.

3.2 Analysis using the Data Quality Analysis Tool

The DQA Tool, which is described in Section 2.6.1, was run on the SFCD data. If requirements were fulfilled only partially or not at all, the reason for that was analysed. In the following, SFCD problems are presented, analysed and possible solutions how to clean the data are discussed.

3.2.1 Automatic correction

Only few problems were found, that can be cleaned automatically. The rounding problems described in Section 3.1.1 can be detected and corrected automatically. This can be done by finding entries where for example the selected value is larger than the minimum value, but only if the difference is based on the last decimal of the rounded value, and the minimum and maximum are the same. Then the minimum and maximum can be rounded to the selected value. Another requirement of the DQA Tool was that the selected values are rounded to a certain precision. This can be cleaned automatically as well by rounding the numbers to the required precision.

3.2.2 Manual correction

Most of the problems need to be cleaned manually. As described in Section 3.1.8 some database fields are null because of legacy data. The missing information has to be extracted from documentations manually.

3.2.3 Correction with choice

Some problems could be cleaned by proposing a few suggestions to the user. He then can decide if an appropriate solution is proposed and select it to clean the problem. For example, to fill in the missing English names mentioned in Section 3.1.3 a dictionary or translating service could be used to propose some translations of the food name to the user who can choose the correct translation if it is available.
For some foods the values of protein, carbohydrate and fat components are missing. A possible solution could be to propose similar food items to the user. He then can choose one and the values of this similar food would be used for the values of the other food.

3.2.4 Correction by a tool

To use a tool for the data cleansing, the problems need to be automatically correctable or at least with choices. For manual corrections a tool does not help so much, because the user has still a lot of work to do as he has to look up the missing information in research papers. In the SFCD most of the problems require manual correction which is why no tool to clean the SFCD was developed in the context of this Master thesis.

3.3 Approaches for duplicate detection

In this section, deduplication approaches for some of the SFCD tables are presented.

3.3.1 Table tblreference

Citation (referencecitation), title (referencetitle) and authors (referenceauthors) can be used to deduplicate the references by comparing the strings with string similarity algorithms.

3.3.2 Table tblsinglefoodcomponentsample

A sample is referenced to one singlefoodcomponent as explained in Section 3.1.6. If the same sample of food is used for several singlefoodcomponents, a new sample entry is added for each singlefoodcomponent, which leads to duplicates in the sample table. If most of the fields are the same for two sample entries, it can be assumed that the two samples are the same. The fields 'idsinglefoodcompsample' and 'singlefoodcompsampleidsinglefoodcomp' should be ignored, the 'singlefoodcompsampledate' values should be close together. Additionally, the singlefoodcomponents of two samples that are assumed to be duplicates should belong to the same singlefood. Some fields are left empty or contain the empty string "" instead of containing the same value as before, as described in Section 3.1.2. In this case the missing information has to be inserted first, before the samples can be compared.
However, to deduplicate the samples another database schema is needed as depicted in Figure

3.3, where one table (sample) only contains data about the samples and one table (singlefoodcomponentsample) contains the relationships between sample items and singlefoodcomponent items.

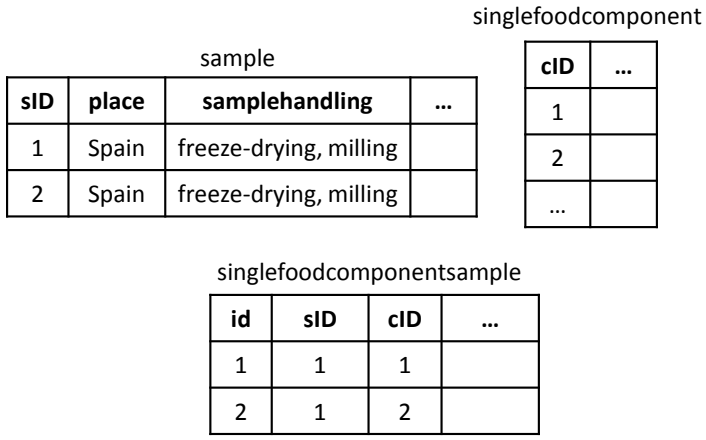

Figure 3.3: Proposed changes of database schema for samples

3.3.3 Table tblmethod

The duplicates detection can be handled analogeous to tblsinglefoodcomponentsample.

3.3.4 Table tblsinglefoodcomponent

The singlefoodcomponents with the same idimportsource are duplicate candidates. The idimportsource contains the ID of the previous database. However, it is possible that the two singlefoodcomponents come from different data sources and the idimportsource is incidentally the same.

3.3.5 Table tblsinglefood

If two singlefoods have many components in common and their measured values (singlefoodcomponents) are similar, they could be duplicates.

4 Approach: Food Matching Tool

4.1 Motivation

The goal of the Food Matching Tool is to match two sets of food items as shown in Figure 4.1. The food data comes from different food data sources, such as the SFCD, a Total Diet Study (TDS) or a food consumption study. Not all data sources provide the same information about food items, which leads to difficulties in food matching. Common food information which is provided by most data sources are the food name in any language and the food category of the food such as fruit, vegetable or milk products.

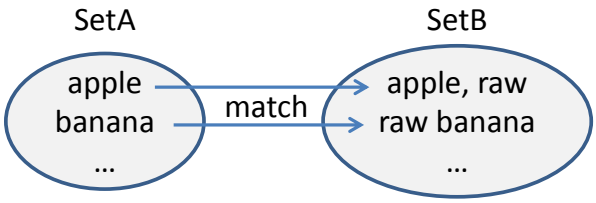

Figure 4.1: Two sets of foods are matched

To find out which food items of the two sets are the same or similar, a matching of the food items is required. The Food Matching Tool can be used to find the best matches for the food items of one set (SetA in Figure 4.1) from the items of the other food set (SetB in Figure 4.1). The idea of the Food Matching is to compare the foods of the SetA with the foods of the SetB, using the food names and the food categories, and to propose the most similar SetB foods as matches for the SetA foods. In the example depicted in Figure 4.2 two sets of foods are illustrated and possible matches of SetB foods are proposed for the food11, which belongs to SetA. The order of the proposed foods indicates a ranking, such that the food with the best ranking is proposed first. The user can then choose one of the proposed foods as the correct

match for the food of SetA.
The Food Matching Tool can be used to enhance the information of a SetA food with the information of the corresponding food item of SetB, or to combine the information. Furthermore, the matching of food items can also be used to find duplicates of two food sets before the sets are merged into a new database.

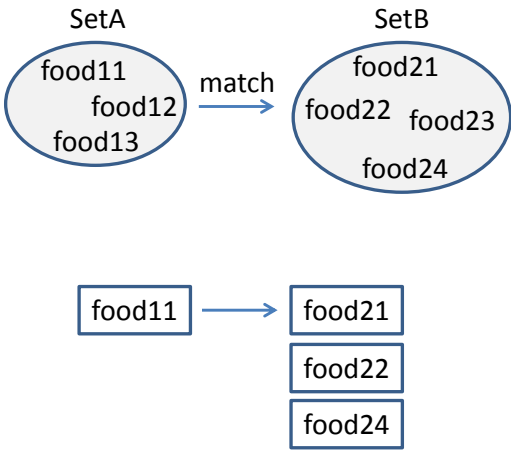

Figure 4.2: Food items of two food sets and proposed foods for food11

4.2 Overview

Figure 4.3 shows an overview of how the food matching process of the Food Matching Tool works. At the beginning the user chooses the two food sets that should be matched, SetA and SetB. The Food Matching consists of an Exploitation step, where stored matches from a previous run of the tool are retrieved from the matching storage, and several matching steps where the food names and other food information, such as food categories, is used to match the foods. In the following it is defined what a matching step is and two scenarios of the Food Matching Tool are explained, called sequence matching and weighted matching.

4.2.1 Matching step definition

Figure 4.4 shows an abstract matching step for a foodA of SetA. As input the matching step gets a candidate set of the foodA, which contains the foods of SetB which are possible matches for the foodA. During the matching step all foods of the candidate set are compared with foodA and a similarity score is computed which indicates how similar foodA and the candidate food are. The candidate foods with the best scores, namely with a score larger than some threshold t, remain in the candidate set, the other candidate foods are excluded from the candidate set such that the candidate set is reduced.

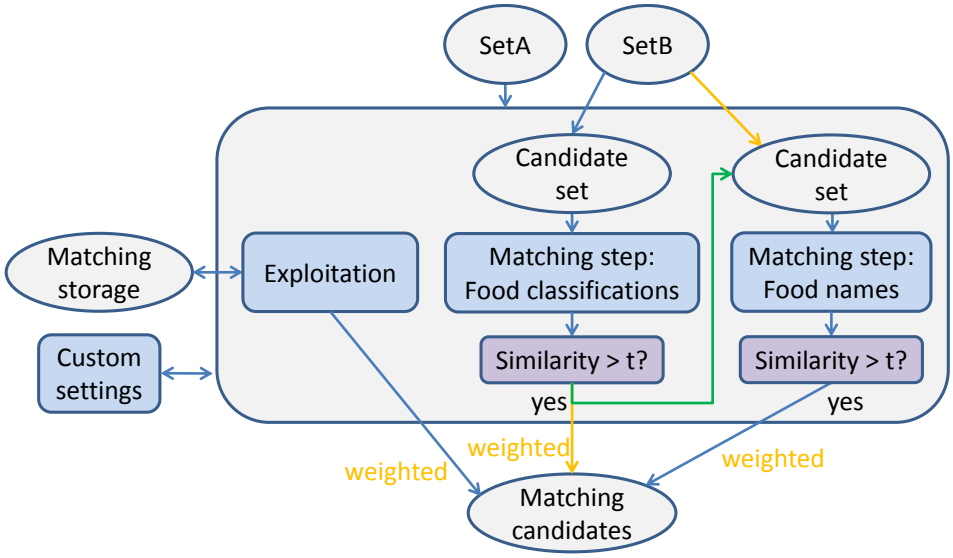

Figure 4.3: Overview of weighted matching (orange) and sequence matching (green)

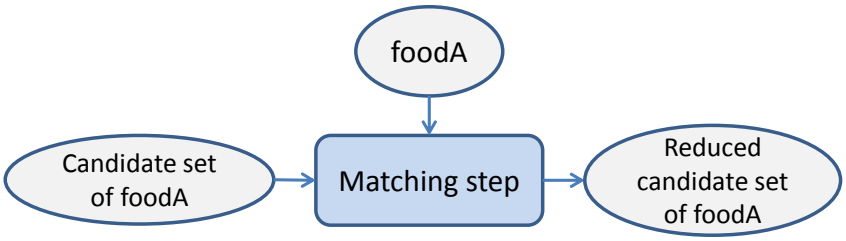

Figure 4.4: Abstract matching step

4.2.2 Sequence matching

The two scenarios of the matching process differ in how the matching steps are combined. In the sequence matching, the matching steps are executed in sequence, such that the reduced candidate set of a matching step is the candidate set of the next matching step for a food of SetA. The candidate set for the first matching step is initialized by the whole SetB. Figure 4.5 shows the initial candidate set for food11 of the example in Figure 4.2, containing all foods of SetB. The sequence matching scenario is shown with the green line in Figure 4.3. The goal of this scenario is to reduce the candidate set after each step, such that in further matching steps a food of SetA has to be compared with less foods of SetB. The remaining candidate foods of the last matching step are proposed as matching candidates to the user together with the matches found in the Exploitation step.

4.2.3 Weighted matching

In the weighted matching, the matching steps are executed separately, such that each matching step gets the whole SetB as input candidate set. The best matches of the matching steps are combined, where the similarity scores of matches of different steps can be weighted dif-

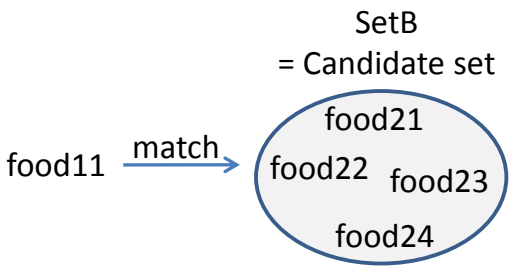

Figure 4.5: Initial candidate set

ferently, and are then proposed to the user as matching candidates. In Figure 4.3 the weighted matching is indicated with orange. The goal of this scenario is that foods of SetB are not excluded from the candidate set if the matching in one step is very bad and would exclude most of the foods.
In the following, first the basic matching steps using food names, food categories and other food classifications are described in more detail, and then the Exploitation step is discussed. In the end of this section the two scenarios are explained in more detail with examples.

4.3 Facet matching

In this section the matching steps that use the food classifications are presented. First it is explained what food classifications are. Afterwards, the 'single facet matching' and the 'multiple facet matching' are described.

4.3.1 Food classification theory

Foods are classified according to their food categories like milk products or fruits, but also according to their wrapping or cooking methods.
The classes that can be chosen to classify a food are called facets. In Figure 4.7 some facets describing food categories are shown, in Figure 4.6 some facets of the LanguaL classification for cooking methods are listed. Facets are represented either by names like 'Fine bakery wares' or by a code like 'A009T'.

There exist two types of food classification: single facet classification and multiple facet classification. If a food is classified by a single facet classification it means that only one facet is assigned to the food. Single facets usually describe the food categories like milk product or fruit. Using a multiple facet classification, one or more facets are assigned to a food. These facets describe for example cooking methods or the wrapping of a food.
In some classifications the facets are organised hierarchically as shown in Figure 4.7 for the FoodEx2 classification, such that facets can have parent facets and child facets. Not only the leaves can be chosen to classify the food, but also the parent facets.
Single facet classifications:

- **EuroFIR classification:** The EuroFIR[1] classification describes only food categories,

[1] http://www.eurofir.org

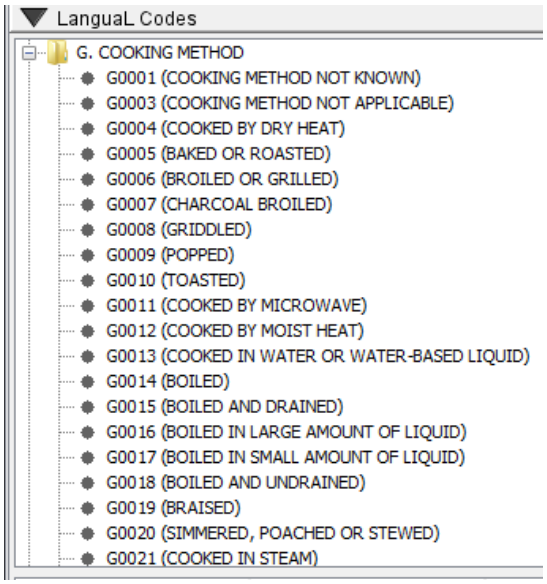

Figure 4.6: Facets of LanguaL classification

no cooking methods or wrappings. The facets are organised in a hierarchical structure.

- **FoodEx2 classification (main facets):** One part of the FoodEx2[2] classification is a single facet classification, which describes the food category of a food. The classification is hierarchical as shown in Figure 4.7.

Multiple facet classifications:

- **Own classification:** Each country has its own classification of food categories. The Swiss classification is non-hierarchical.
- **FoodEx2 classification (sub facets):** The other part of the FoodEx2 classification, which describes cooking methods among others, is a multiple facet classification. The facets are organised hierarchical. Examples for parent facets and their child facets are 'Characterising Ingredient' ('Fruit and fruit products', 'Tap water'), 'Cooking-method' ('Baked', 'Microwave-cooked'), 'Preservation-technique facet' ('Frozen').
- **LanguaL classification:** The facets of the LanguaL classification are organised in groups as shown in Figure 4.8. From the facet group A (Product Type) several facets can be selected, for the other facet groups B to Z only one facet is selected from each group. Within the facet groups the facets are organised hierarchical, even though in the FoodCASE application the facets are not structured hierarchically.

4.3.2 Single facet matching

The single facet matching is the matching step that uses single facets to match food items. As described in Section 4.3.1, in a single facet classification a food item is classified with only

[2]http://www.efsa.europa.eu/de/datex/datexfoodclass.htm

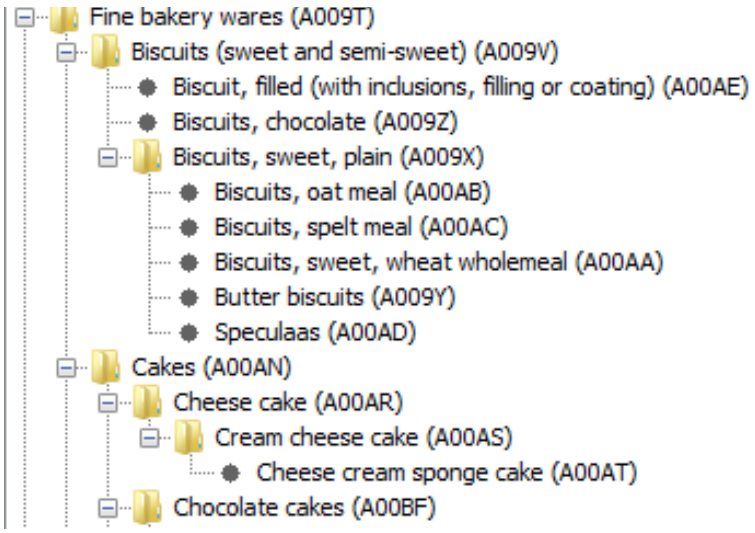

Figure 4.7: Hierarchical facet structure of FoodEx2 classification

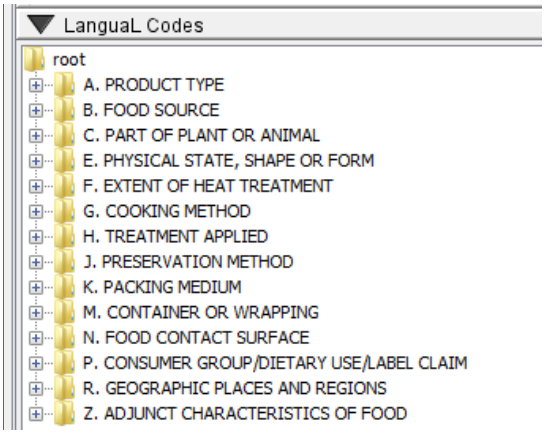

Figure 4.8: Facet groups of the LanguaL classification

one facet and this facet describes the food category. This means if two foods are classified with the same single facet, the food belongs to the same food category. In this matching step, foods with the same single facets are considered to be similar, whereas foods with different single facets are considered to be less similar. For single food classifications which are hierarchical, the parent-child structure is also taken into account. Figure 4.9 shows an example, where a food11 of SetA is classified with the facet 'Fish', and the foods of SetB are classified with 'Fish', 'Seafood' and 'Meat'. The foods with facet 'Fish' remain in the candidate set, the food with facet 'Meat' is excluded. The facet 'Seafood' is a parent of the facet 'Fish' which is why the food22 is not excluded from the candidate set.
In more detail, a similarity function is needed to compute a similarity score for two foods. The function should take the hierarchical structure into account. A high similarity score means that the two foods are very similar, whereas a low score means that the two foods are dissimilar. The concrete similarity function used for the Food Matching Tool is described in Section 4.3.4.

Custom settings: The user can specify what single facet classifications are available for the SetA, such that only these classifications are used to match the foods. The user can also specify the threshold that is used to determine which foods remain in the candidate set and which ones are excluded from the candidate set.

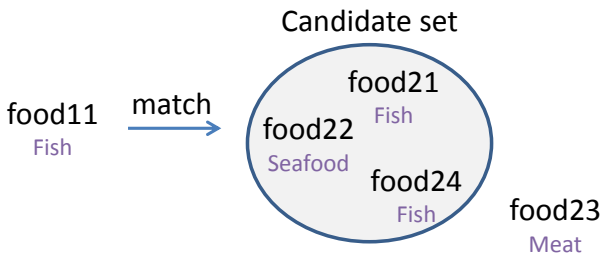

Figure 4.9: Candidate set after using single facets

4.3.3 Multiple facet matching

The multiple facet matching is the matching step that uses multiple facets to match food items. The SetA foods are compared to the foods of their candidate sets. In Figure 4.10 food11 has the multiple facets A0120, B1205, F1314 and is compared to other foods. To compare two food items using multiple facets, a similarity measure needs to be defined to compute a ranking of the food items. Several approaches can be considered for this measure:

- Intuitively, SetB foods with more facets in common with a SetA food are more likely to be a match for this SetA food. But if the level of the facets in the facet tree is considered, it can also be said, that facets with a lower level are more accurate and should therefore contribute more to the score than facets with a higher level.
- For hierarchical multiple facet classifications, the parent facets of facets of a food should also be taken into account. For example, if a facet of foodA is a parent of a foodB facet this should also contribute to the similarity score, because it means that at least at some level the foods are similar. Another approach of facet weighting is presented in Section 9.2.2.

The concrete similarity function used for the multiple facet matching in the Food Matching Tool is the same as for the single facet matching and is described in Section 4.3.4. When the similarities between a SetA food and its candidate foods are computed, a ranking of the candidate foods is established. The foods with the highest similarity get the best rank, the foods with the lowest similarity get the worst rank. There are the following possibilities how to continue with the candidate food ranking:

- The SetB foods with the most facets in common remain in the candidate set of a SetA food. For example keep the best k=5 matches. The disadvantage of this approach is that some SetB foods are excluded from the candidates even though they have the same similarity as some other SetB foods that remain in the set, such that the order of the foods is relevant.

- Keep the SetB foods in the candidate set for which the similarity with the SetA food is higher than a certain threshold. In Figure 4.10 the threshold is set to 0.5, which means that candidates with a lower similarity, such as food24, are excluded from the candidate set. This approach is chosen in the Food Matching Tool.

Custom settings: The user can specify which multiple facet classifications are available for the SetA foods, such that only these classifications are used to match the foods. The user can also specify the threshold or the value of k, which are used to determine which foods remain in the candidate set and which ones are excluded.

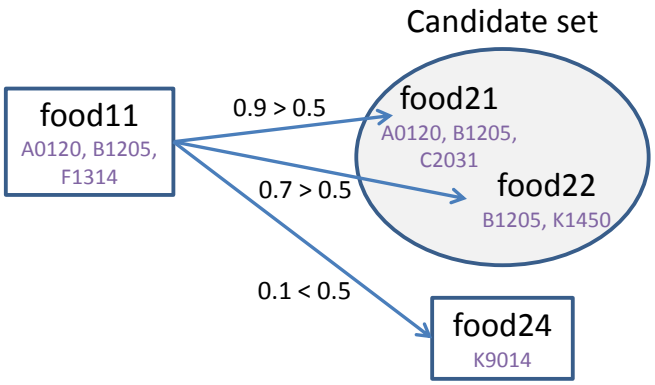

Figure 4.10: Candidate set after using multiple facets and the similarities between the food items

4.3.4 Similarity for facet matching

For both, the single facets and the multiple facets, the same method is used to compute the similarity scores of two foods. The method makes use of the Jaccard similarity, which is described in Section 2.2.5. In Figure 4.11 and Figure 4.12 examples are shown how to compute the similarity scores for facets. A facet set of a food is defined here as the set of all facets the food is classified with, plus all their parent facets. Figure 4.11 shows an example to compute the similarity of foodA and foodB using single facets. foodA is classified with facet $f8$, foodB with the facet $f9$. The facet set of foodA consists of foodA's single facet $f8$ and all the parent facets of $f8$, namely $f5, f2, f1$. The facet set for foodB is built analogeous. The Jaccard similarity of the two facet sets is then used as the score for foodA and foodB.

Figure 4.12 shows an example for a matching using multiple facets. FoodA is classified with the facets $f4$ and $f8$, foodB with $f7$ and $f9$. The facet set of foodA consists of foodA's multiple facets $f4$ and $f8$, plus all their parents, namely $f5$, which is a parent of $f8$, and $f2$ and $f1$, which are parents of both, $f4$ and $f8$. The facet set for foodB is built analogeous. The similarity score for the multiple facet matching of foodA and foodB is then computed using the Jaccard similarity of their facet sets.

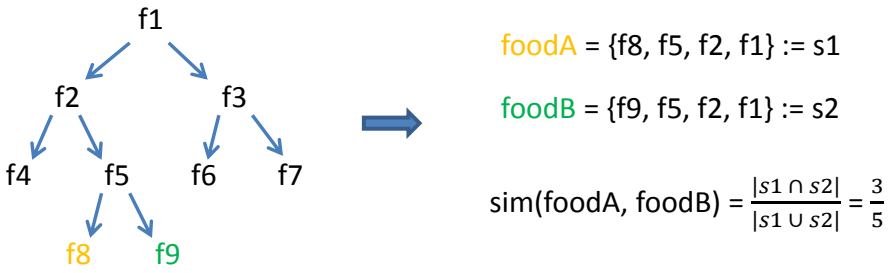

Figure 4.11: Similarity of single facet matching

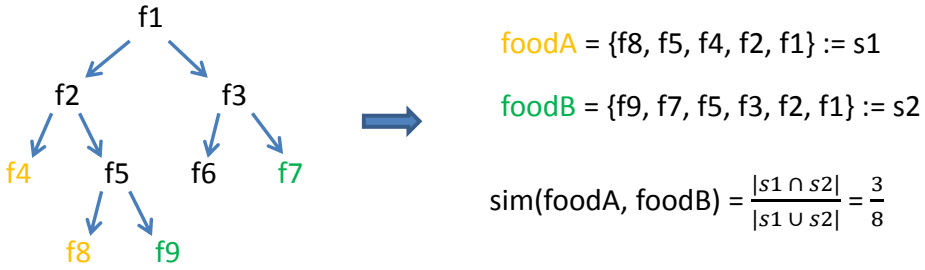

Figure 4.12: Similarity of multiple facet matching

In the examples of Figure 4.11 and Figure 4.12 the facets of foodA and foodB are leaves of the facet tree, but they could also be inner nodes of the tree.

4.4 Food name matching

In the food name matching step, the food names are used to find the most similar foods of SetB for a food of SetA. As name the English name of a food is used in the Food Matching Tool, but also other names, such as synonyms or names of other languages, could be considered for a food name matching. This is discussed later in Section 8.3. The English name of a food in SetA is compared to the English names of the foods in its candidate set. Also in this matching step a similarity measure is needed to compute the similarity of two food names and therefore of the corresponding foods. As similarity measure different string similarity methods can be used as described in 2.2.5. In the Food Matching Tool the Levenshtein distance and Jaccard similarity with N-grams are used. The resulting similarity scores are used to compute a ranking of the candidate foods. The foods of the top ranks are proposed as matches for the SetA food.

4.5 Exploitation

In the Exploitation step of the food matching, the goal is to exploit matches that were stored in previous runs of the Food Matching Tool. To understand this exploitation, the version and study concept of food data is described in the following before the process of the Exploitation

is explained.

4.5.1 Version and study theory

In the context of Food Composition Data, from time to time foods that are already stored in the database are measured again. Reasons for this could be that through new research some components or substances of the food composition data change or new nutrient or contaminant values are measured, such that the food data set needs to be updated. Instead of adding the new information to the already stored food item, a new food item of the food is stored, such that the old food item is not changed. To relate the two items, a food has a foodID and a version, as shown in Figure 4.13.
In the context of Total Diet Studies, for each study that is carried out, a set of foods is chosen which are investigated in this study. If foods were already investigated in an earlier study, the stored foods also have a foodID to relate foods among each other. Instead of the version as in the Food Composition Data, TDS foods have a study as attribute. In the following, the expression 'version' is used for both, version and study.

ID	foodID	englishname	version	...
1	10	apple, raw	5	
2	11	raw banana	5	
...	...	...	...	...
203	10	apple, raw	7	
204	11	banana, raw	7	

Figure 4.13: Example for versions of foods

4.5.2 Exploitation process

In the Exploitation step, the SetA and SetB foods are looked up in the matching storage using their foodIDs and the current and previous versions of the food sets. If matches between SetA and SetB foods of any version combination are found, they are proposed to the user as possible matching candidates. In the following, some examples are presented with different version combinations. The version of SetA is denoted by versionA, the version of SetB with versionB. A food with foodID XX is called foodXX. In Figure 4.14 and 4.15 Set1 is a previous version of Set2 and Set3 is a previous version of Set4. To indicate that the set versions are not all related, the sets on the right hand side of the figure have a column study instead of version.

- versionA, previous versionB: Set2 and Set4 should be matched in Figure 4.14. Set2 and Set3 were matched in a previous run of the Food Matching Tool, where food11 of Set2 was matched with food9 of Set3. This match was stored in the matching storage and can be retrieved in the Exploitation step. In Set4 there is also a food with foodID

9, and since Set3 is a previous version of Set4 food9 is proposed as matching candidate of food11.

- previous versionA, versionB: Set2 and Set3 should be matched in Figure 4.14. Set1 and Set3 were matched in a previous run of the Food Matching Tool, where food10 of Set1 was matched with food8 of Set3. In the Exploitation step of food10 of Set2 this match is retrieved from the matching storage and food8 of Set3 is proposed as matching candidate of food10.
- previous versionA, previous versionB: Set2 and Set4 should be matched in Figure 4.15. Set1 and Set3 were matched in a previous run of the Food Matching Tool, where food10 was matched with food8 and food11 was matched with food9. These matches were stored in the matching storage. In the Exploitation step of Set2, the foodIDs 10 and 11 are looked up in the matching storage together with previous versions of Set2 and Set4, which are the sets Set1 and Set3 in this example. The matches of the previous run are retrieved and therefore food8 is proposed as matching candidate for food10 and food9 is proposed for food11.

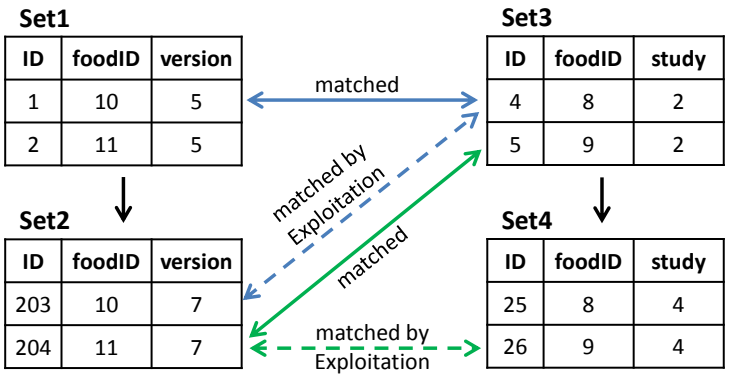

Figure 4.14: Examples of Exploitation

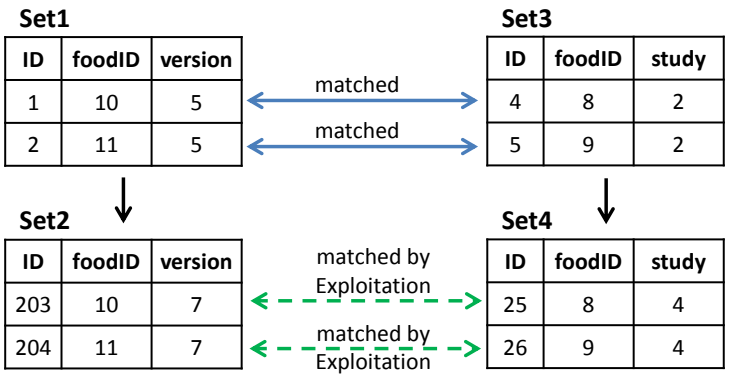

Figure 4.15: Examples of Exploitation

The version combination of versionA, versionB is not considered as relevant, because this would mean that the two sets already have been matched. A version combination with subsequent versions of SetA and SetB is also not considered because in practice the new versions should be matched, and not older versions.
The benefit of the Exploitation step is that the user who runs the Food Matching Tool can see which SetB foods were chosen in previous runs as the correct matches of the SetA foods.

4.6 Matching scenarios

4.6.1 Sequence matching

In the sequence of steps scenario, the matching steps 'single facet matching', 'multiple facet matching' and 'food name matching' are executed in sequence, where in each step some of the SetB foods are excluded from the candidate set as explained in Section ... . To determine which foods are excluded, a threshold t is used after each matching step, such that if the computed score s of a foodA of SetA and a foodB of SetB is smaller than t, foodB is excluded from the candidate set of foodA. The matches that are found in the Exploitation step do not have a similarity score. They are just added without a score to the final candidate set of foodA that is presented to the user.

4.6.2 Weighted matching

In the weighted matching scenario, the matching steps 'single facet matching', 'multiple facet matching' and 'food name matching' are executed independently, such that for each step the candidate set of a foodA consists of all SetB foods. After each step some foods of the candidate set are excluded using a threshold as in the sequence of steps scenario. Then the remaining candidates of all steps are combined using their scores and a weight for each matching step. The following example in Table 4.16 shows how to compute the weighted similarity of two foods foodA and foodB. The table contains example values for the similarity scores and the weights of each matching step. The similarity score of the Exploitation is defined by 1. The weights of the matching steps are user-defined. For each matching step, the similarity of a matching step is multiplied with the step's weight. Then these weighted similarities are added up to one final similarity. The candidate foods of the foods of SetA are proposed to the user together with their final similarities.
Custom settings: The user can specify the weights that are used to combine the matching candidates of the different matching steps.

matching step	similarity score	weight of step	weighted similairty (score * weight)
single facet matching	0.6	0.5	0.3
multiple facet matching	0.7	1	0.7
food name matching	0.4	2	0.8
exploitation	1.0	1	1.0
final weighted similarity			**2.8**

Figure 4.16: Example for a weighted matching scenario

5 Implementation

5.1 Food Matching Java classes

In Figure 5.1 an overview of the Java classes are shown that were implemented or used for the Food Matching Tool. In the following the purpose and the main functionality of the classes are described. In general the classes are located on the client side in package ch.ethz.inf.tds.client.gui.food.matching as a part of the FoodCASE distribution. If this is not the case, it is mentioned in the following. In the appendix in Section A.1, a more detailed class diagram is shown with local variables and methods of the classes. In the implementation of the name matching, the library DéjàVu Stringmetrics[1] is used. which was developed during the Information Systems Lab 2013 in the Globis Group by students.

- `FoodMatching`: This class is the main class of the Food Matching Tool. The user can choose the food sets that have to be matched and can select the matching scenario, namely matchWithWeights() or matchWithSequence(). These two methods call the methods `findExploitationCandidates()`, `findSingleFacetCandidates()`, `findMultipleFacetFoodEx2Candidates()` and `findNameCandidates()` of the `FoodMatching` class to retrieve the candidates of each matching step. The candidates for each food of SetA are maintained in the list `candidates`.

- `FoodObject`: The foods of the two food sets are objects of the classes `AggregatedFood` and `TdsFood`. This class represents the food items in a shared representation. More details about this class can be read in Section 5.2.

- `SingleFacetMatching`: An instance of this class is created in the method findSingleFacetCandidates(). The class preprocesses SetB and computes the single facet

[1]The DéjàVu Stringmetrics library was developed by students during the Information Systems Lab of the Globis Group in spring 2013

matching of each food of SetA and its candidates. More details are described in Section 5.3.1.

- `MultipleFacetMatchingFoodEx2`: An instance of this class is created in the method findMultipleFacetFoodEx2Candidates(). The class preprocesses SetB and computes the multiple facet matching of each food of SetA and its candidates using the FoodEx2 classification. More details are described in Section 5.3.1.

- `MultipleFacetMatchingOwn`: An instance of this class is created in the method findMultipleFacetOwnCandidates(). The class preprocesses SetB and computes the multiple facet matching of each food of SetA and its candidates using the own classification. More details are described in Section 5.3.1.

- `FacetMatchingUtils`: This class contains methods to compute the Jaccard similarity of two sets. They are used for the facet matching and are described in more detail in Section 5.3.1.

- `FoodNameMatching`: An instance of this class is created in the method findNameCandidates(). The class matches the name of a food of SetA with the names of its candidates using different similarity methods. More details are described in Section 5.3.2.

- `NGramWordTokenizer`: This class is used for the name matching. It extends the class `Tokenizer` of the string matching library DéjàVu. More details can be found in Section 5.3.2.

- `Tokenizer`, `WordTokeninzer`, `NGramTokenizer`: These classes belong to the string matching library DéjàVu and are used for the implementation of class `NGramWordTokenizer`.

- `Exploitation`: An instance of this class is created in the method findExploitationCandidates(). The class retrieves matches for a food of SetA from the matching storage or matches for a previous version of the food. More details are described in Section 5.5.

- `MatchingStorage`: This class represents the matching storage and therefore the entries of the database table, where the matches of previous runs of the Food Matching Tool are stored, and where the Exploitation gets the matches from. More details about the matching storage can be found in Section 5.4.

- `FoodObjectSchema`: `FoodObjectSchema` is an enumeration type and represents the data source of the two food sets. Foods of the database table 'tblaggrfood' get the FoodObjectSchema 'TBL_AGGRFOOD', foods of the database table 'tdsfood' get the FoodObjectSchema 'TDS_FOOD'.

- `MatchingStorageBean`: This class is located on the server side in package ch.ethz.inf.foodcomp.server.sessionbeans. The class is used by the `Exploitation` class to access the database. The `MatchingStorageBean` class provides methods for retrieving, storing and deleting matches of the matching storage. More information about the `MatchingStorageBean` can be found in Section 5.4.

- `FoodMatchingTest`: This test class contains the tests for the Food Matching Tool.

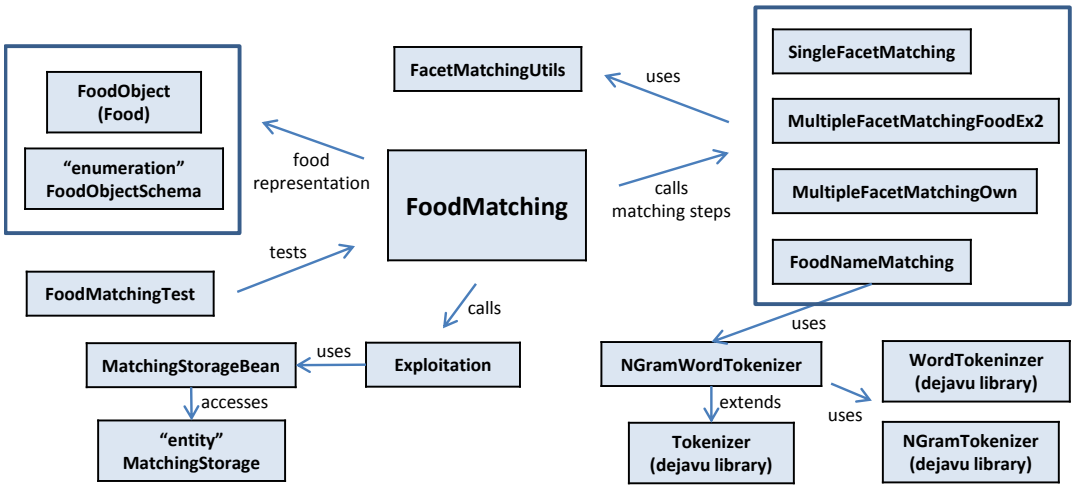

Figure 5.1: Overview of Food Matching classes

5.2 Food data representation

In the implementation of the Food Matching Tool, food data from two different sources can be matched, namely Food Composition Data from the FoodCASE database table 'tblaggrfood' and TDS data from the FoodCASE database table 'tdsfood'. The food data of these two sources is represented in a common food data representation, which is an object of the class `FoodObject`, such that during the matching process the foods from both sources can be handled equally. Only the information which is necessary for the food matching is extracted from the food data: the English name, the single and multiple facets of the classifications FoodEx2, LanguaL and the own classification, the foodID and the database schema, which is 'tblaggrfood' or 'tdsfood'. In Figure 5.2 it is shown how the information of an `AggregatedFood` object is extracted.

5.3 Matching steps

5.3.1 Facet matching

In the Food Matching Tool the single food matching is implemented using the main facets of the FoodEx2 classification described in Section 4.3.1, the multiple facet matching is implemented using the sub facets of the FoodEx2 classification or the own Swiss classification also described in Section 4.3.1. Since both classifications are hierarchical, the parent facets can be used for the facet matchings. In the method `fillFacetsB()` of the single and multiple facet matching, the foods of SetB are preprocessed by building the facet set of each food in SetB. As an example, the `fillFacetsB()` method of the `SingleFacetMatching` class is shown in Figure 5.3.

```
public FoodObject(AggregatedFood food) {
    this.id = food.getId();
    this.foodId = Integer.parseInt(food.getPublicId());
    this.originalName = food.getName();
    this.englishName = food.getnameenglish();
    if (food.getFoodEx2() != null) {
        this.foodEx2Single = food.getFoodEx2();
    }
    foodEx2Multiple = new HashSet<FoodEx2Facet>();
    for (AggrFoodEx2Facet aggrFacet : food.getFoodEx2Facets()) {
        foodEx2Multiple.add(aggrFacet.getFoodEx2Facet());
    }
    langualCodes = new ArrayList<Langualcode>();
    for (AggrFoodLangualcode aggrFoodLangualcode : food.getlangualcodelist()) {
        langualCodes.add(aggrFoodLangualcode.getLangual());
    }
    ownCategories = new ArrayList<Category>();
    for (AggregatedFoodCategory aggrCat : food.getCategoryList()) {
        ownCategories.add(aggrCat.getCategory());
    }
}
```

Figure 5.2: `FoodObject` from an `AggregatedFood`

```
        private void fillFacetsB(List<FoodObject> setB) {
            for (FoodObject foodB : setB) {
                Integer foodBId = foodB.getId();
                HashSet<Integer> facetSet = new HashSet<Integer>();
                //get the single facet of foodB
                if (foodB.getFoodEx2Single() != null) {
                    FoodEx2 f = foodB.getFoodEx2Single();
                    Integer facet = f.getId();
                    facetSet.add(facet);
                    //find parent facets
                    FoodEx2 currentFacet = f;
                    while (currentFacet.getParent() != null) {
                        FoodEx2 parent = currentFacet.getParent();
                        facetSet.add(parent.getId());
                        currentFacet = parent;
                    }
                }
                facetSetsB.put(foodBId, facetSet);
            }
        }
```

Figure 5.3: Builds facet set of foods in SetB

Figure 5.4 and 5.5 show how the similarity of the facet sets of a foodA and its candidates is computed

```
//compute similarity between foodA and foodB's of candidate set
for (Integer foodBId : previousCandidates) {
    Set<Integer> facetSetB = facetSetsB.get(foodBId);
    //use Jaccard similarity
    Double similarity = FacetMatchingUtils.
            computeJaccardSimilarity(facetSetA, facetSetB);
    if (!candidateScores.containsKey(similarity)) {
        candidateScores.put(similarity, new HashSet<Integer>());
    }
    candidateScores.get(similarity).add(foodBId);
}
```

Figure 5.4: Computes the similarity of the facet sets of foodA and its candidates. `candidateScores` is of type `TreeMap<Double, Set<Integer>>` and is empty at the beginning.

```
static public Double computeJaccardSimilarity(Set<Integer> facetsFoodA,
        Set<Integer> facetsFoodB) {
    //addAll for union, retainAll for intersection, removeAll for minus
    //create union set
    Set<Integer> union = new HashSet<Integer>(facetsFoodA);
    union.addAll(facetsFoodB);
    //create intersection set
    Set<Integer> intersection = new HashSet<Integer>(facetsFoodA);
    intersection.retainAll(facetsFoodB);
    //compute similarity
    Double sizeIntersection = Integer.
            valueOf(intersection.size()).doubleValue();
    Double sizeUnion = Integer.valueOf(union.size()).doubleValue();
    Double similarity = sizeIntersection / sizeUnion;
    return similarity;
}
```

Figure 5.5: Method to compute Jaccard similarity of two sets

5.3.2 Name matching

For the food name matching, the Jaccard similarity is used again to compute the similarity of the English names of the foods. Here, the sets which are compared using the Jaccard similarity are sets of 3-grams. As explained in Section 2.2.5, a food name 'apple' is transformed into a set of 3-grams {'app','ppl','ple'}. For the implementation of the food name matching, classes of the library DéjàVu were used and extended. The library allows to build the 3-grams of the food names and then to compute the Jaccard similarity of their sets. In the Food Matching Tool a variant of this method is implemented in the class `NGramWordTokenizer`. It extends the class `Tokenizer` and combines the `WordTokeninzer` and the `NGramTokenizer` of DéjàVu. The variant consists of splitting the food names into words first and then to build the 3-grams of the words. If for example the name contains several words, like 'apple raw', then the 3-grams which contain

the whitespace are ignored and not part of the 3-grams set of the food. This approach helps to ignore the ordering of the words, which should not be relevant in the name matching. After the 3-grams sets of two foods are built, the Jaccard similarity of the two sets can be computed, which is used as the score of the food name matching of the two foods.
Figure 5.6 and 5.7 show how the food name matching is implemented. The method `matchFoodNames()` computes the similarities of foodA and its candidates using the `SimilarityMetric` that is given as argument. `SimilarityMetric` is a class of DéjàVu and is a super class of `JaccardSimilarity` and `JaccardSimilarityWithTokenizer`. `JaccardSimilarityWithTokenizer` is the similarity metric that is used in Figure 5.7. The method of this figure shows an example of how to instantiate a similarity metric of the DéjàVu library and how `matchFoodNames()` is called.
The DéjàVu library provides also other similarity methods besides the Jaccard similarity. One of them is the Levenshtein distance, which is described in Section 2.2.5. This method is also provided by the Food Matching Tool for the food name matching, as well as the original Jaccard and 3-grams method and its extended version of the class `NGramWordTokenizer`. The user can choose one of these methods to match the food names.

```
private TreeMap<Double, Set<Integer>> matchFoodNames(FoodObject foodA,
        Set<Integer> candidateSet, SimilarityMetric simMethod) {
    //TreeMap storing similarities and foodBIds
    TreeMap<Double, Set<Integer>> candidateScores =
            new TreeMap<Double, Set<Integer>>();
    //get name of foodA
    String nameA = foodA.getEnglishName();
    if (nameA != null) {
        //compute similarity for each candidate food
        for (Integer foodBId : candidateSet) {
            FoodObject foodB = hashedSetB.get(foodBId);
            String nameB = foodB.getEnglishName();
            if (nameB != null) {
                Double similarity = simMethod.computeSimilarity(
                        nameA.toLowerCase(), nameB.toLowerCase());
                if (!candidateScores.containsKey(similarity)) {
                    candidateScores.put(similarity, new HashSet<Integer>());
                }
                candidateScores.get(similarity).add(foodBId);
            }
        }
    }
    //if nameA is null -> no entries in tree
    //if a nameB is null -> foodB is not added to tree (is no candidate)
    return candidateScores;
}
```

Figure 5.6: Similarity of food names is computed using the given similarity metric

```
public TreeMap<Double,Set<Integer>> matchFoodNamesJaccardNGramWord(
        FoodObject foodA, Set<Integer> candidateSet){
    //create JaccardSimilarityWithTokenizer instance
    NGramWordTokenizer tokenizer = new NGramWordTokenizer(3);
    JaccardSimilarityWithTokenizer jaccSim =
            JaccardSimilarityWithTokenizer.getBuilder().
                    tokenizer(tokenizer).takeTokenizer(true).build();
    //match food names using this instance
    TreeMap<Double, Set<Integer>> candidateScores =
            matchFoodNames(foodA, candidateSet, jaccSim);
    return candidateScores;
}
```

Figure 5.7: Similarity of food names is computed using the given similarity metric

5.4 Matching storage

The database table of the matching storage is shown on the left hand side of Figure 5.8. It consists of an item ID and the foodID, schema and version of the SetA and SetB foods, where the schema is the database table 'tblaggrfood' or 'tdsfood' which the matched food belongs to. A food set SetA or SetB is uniquely defined by the schema and the version. If a match of a SetA food and a SetB food is stored, the foodID, schema and version of the SetA food are stored in the columns foodid1, schema1 and version1. The information of the SetB food is stored in foodid2, schema2 and version2.

Matching storage

ID	foodid1	schema1	version1	foodid2	schema2	version2
1	10	tblaggrfood	5	8	tdsfood	2
...	...	...	...	...	...	...
20	10	tblaggrfood	7	8	tdsfood	2
...	...	...	...	...	...	...
60	8	tdsfood	4	10	tblaggrfood	7
61	9	tdsfood	4	11	tblaggrfood	7

SetA	SetB
Set1	Set3
Set2	Set3
...	...
Set4	Set2
Set4	Set2

Figure 5.8: Matching Storage example

5.5 Exploitation

The Exploitation step of the Food Matching Tool is implemented in a more general form than described in the approach of Section 4.5. For the SetA foods the version of SetA, called versionA, and the previous versions of versionA are considered as explained in the approach. For the SetB foods not only previous versions of versionB are considered as mentioned in the approach, but all versions.

For example if Set2 and Set4 should be matched in Figure 4.14 of Chapter 4, the Set2 foods food10 and food11 are looked up in the matching storage. For food10 this means that the tuple {10, tblaggrfood, 7}, which corresponds to {foodid, schema, version}, is looked up in the matching storage. Since Set2 could have been SetA or SetB in previous runs of the tool, the foods of Set2 are looked up twice in the matching storage: Once in the columns of {foodid1, schema1, version1}, whereas schema2 is the schema of Set2, and once in the columns of {foodid2, schema2, version2}, whereas schema1 is the schema of Set2. This procedure is implemented in the method `getMatchesForVersionA()` depicted in Figure 5.9. All matches that are retrieved by this method are proposed as matching candidates of food10 if the foodid2, or foodid1 respectively, is a foodID of SetB, which is Set4 in this example. The version of the SetB foods is not considered as relevant because all versions of SetB are considered as mentioned before.
After retrieving matches for versionA, matches for previous versions of versionA are retrieved. For the example of Figure 4.14 this means that the tuple {10, tblaggrfood, 5} is looked up in the matching storage, because version 5 is the previous version of version 7. The retrieving of the matches for this version is analogeous to the retrieving of matches for versionA. The retrieved SetB foods are also proposed as matching candidates. Figure 5.10 shows the method where `getMatchesForVersionA()` is called with previous versions of versionA.

```
private void getMatchesForVersionA(FoodObject foodA, Integer version,
        Set<FoodObject> matches) {
    //get all matches for given version of A
    //where foodA is food1 in matching storage
    List<MatchingStorage> matches1 = BeanBag.getMatchingStorageBean().
            getMatchesPart1(foodA.getFoodId(), schemaA.toString(),
                    version, schemaB.toString());
    //check if foodIDs of retrieved matches are part of SetB
    for (MatchingStorage match : matches1) {
        Integer foodId2 = match.getFoodid2();
        if (foodIdsB.containsKey(foodId2)) {
            matches.add(foodIdsB.get(foodId2));
        }
    }
    //get all matches for given version of A
    //where foodA is food2 in matching storage
    List<MatchingStorage> matches2 = BeanBag.getMatchingStorageBean().
            getMatchesPart2(foodA.getFoodId(), schemaA.toString(),
                    version, schemaB.toString());
    //check if foodIDs of retrieved matches are part of SetB
    for (MatchingStorage match : matches2) {
        Integer foodId1 = match.getFoodid1();
        if (foodIdsB.containsKey(foodId1)) {
            matches.add(foodIdsB.get(foodId1));
        }
    }
}
```

Figure 5.9: Retrieve matches from matching storage

```
public Set<FoodObject> getMatches(FoodObject foodA) {
    Set<FoodObject> matches = new HashSet<FoodObject>();
    //find matches for current version of A
    getMatchesForVersionA(foodA, versionA, matches);
    //find matches of previous versions of A
    TreeSet<Integer> versionListA;
    if (schemaA.equals(FoodObjectSchema.TBL_AGGRFOOD)) {
        versionListA = tblVersions;
    } else {
        versionListA = tdsVersions;
    }
    Integer previousVersionA = versionListA.lower(versionA);
    if (previousVersionA != null) {
        getMatchesForVersionA(foodA, previousVersionA, matches);
    }
    return matches;
}
```

Figure 5.10: Get exploitation matches for versionA and its previous versions

5.6 Matching scenarios

5.6.1 Sequence matching

The sequence matching is implemented in the method `matchWithSequence()` shown in Figure 5.11. It calls the matching steps in sequence and adapts the candidate set of the foods of SetA after each step. If the returned candidate set of a matching step for a SetA food, which is the reduced candidate set, is empty, the candidate set of this food is not changed after this matching step. This is realised in the method `setEmptyCandidateSetToPreviousSet`. The reason for this is, that if a food of SetA is not classified with the classification of the single or multiple facet matching, the facet matching cannot find any candidates and the empty set is returned. Therefore, the next matching step can also not find any candidates because the input candidate set is empty. To avoid this case, where all following matching steps are useless, the candidate set is only changed if it is not empty. As the final candidates for a SetA food are stored in a `TreeMap` together with their similarities, the matches that are found in the Exploitation step are also stored in this `TreeMap` in the end with similarity `NaN` such that all candidates are stored in one data structure.

5.6.2 Weighted matching

The weighted matching is implemented in the method `matchWithWeights()` shown in Figure 5.12. It calls all the matching steps independently, such that the input candidate set for each matching step is the whole SetB. Afterwards, the returned candidates of each step for a food of SetA are combined.

```
public void matchWithSequence() {
    initialiseCandidates(setB);
    List<Set<FoodObject>> exploitationCandidates =
            findExploitationCandidates();
    //single facet matching
    List<TreeMap<Double,Set<Integer>>> singleCandidates =
            findSingleFacetCandidates();
    //if no candidates were found, use previous candidate set for next step
    setEmptyCandidateSetToPreviousSet(candidates, singleCandidates);
    candidates = singleCandidates;
    //multiple facet matching
    List<TreeMap<Double,Set<Integer>>> multipleCandidates =
            findMultipleFacetFoodEx2Candidates();
    setEmptyCandidateSetToPreviousSet(candidates, multipleCandidates);
    candidates = multipleCandidates;
    //name matching
    List<TreeMap<Double,Set<Integer>>> nameCandidates = findNameCandidates();
    candidates = nameCandidates;

    //add exploitation matches to candidates with similarity NaN
    Iterator<Set<FoodObject>> itExploit = exploitationCandidates.iterator();
    for (TreeMap<Double,Set<Integer>> candidatesA : candidates) {
        candidatesA.put(NaN, new HashSet<Integer>());
        Set<FoodObject> explMatchesA = itExploit.next();
        for (FoodObject match : explMatchesA) {
            candidatesA.get(NaN).add(match.getId());
        }
    }
}
```

Figure 5.11: Sequence matching method

```
public void matchWithWeights() {
    List<TreeMap<Double,Set<Integer>>> finalCandidates =
            new ArrayList<TreeMap<Double,Set<Integer>>>();
    initialiseCandidates(setB);
    List<Set<FoodObject>> exploitationCandidates = findExploitationCandidates();
    List<TreeMap<Double,Set<Integer>>> singleCandidates = findSingleFacetCandidates();
    List<TreeMap<Double,Set<Integer>>> multipleCandidates =
            findMultipleFacetFoodEx2Candidates();
    List<TreeMap<Double,Set<Integer>>> nameCandidates = findNameCandidates();

    //combine and weight candidates for each foodA
    Iterator<Set<FoodObject>> itExploitation = exploitationCandidates.iterator();
    Iterator<TreeMap<Double,Set<Integer>>> itSingle = singleCandidates.iterator();
    Iterator<TreeMap<Double,Set<Integer>>> itMultiple = multipleCandidates.iterator();
    Iterator<TreeMap<Double,Set<Integer>>> itName = nameCandidates.iterator();
    while (itSingle.hasNext()) { //for each foodA
        Hashtable<Integer,Double> candidatesA = new Hashtable<Integer, Double>();
        //add exploitation candidates
        Set<FoodObject> explCandA = itExploitation.next();
        for (FoodObject candidate : explCandA) {
            candidatesA.put(candidate.getId(), weightExploitation);
        }
        //add candidates of matching steps
        TreeMap<Double,Set<Integer>> singleCandA = itSingle.next();
        extractCandidatesFromTree(singleCandA, candidatesA, weightSingleMatching);
        TreeMap<Double,Set<Integer>> multipleCandA = itMultiple.next();
        extractCandidatesFromTree(multipleCandA, candidatesA, weightMultipleMatching);
        TreeMap<Double,Set<Integer>> nameCandA = itName.next();
        extractCandidatesFromTree(nameCandA, candidatesA, weightNameMatching);

        //transform hashtable of candidates into tree of candidates
        TreeMap<Double,Set<Integer>> treeCandA = new TreeMap<Double,Set<Integer>>();
        for (Entry<Integer,Double> entry : candidatesA.entrySet()) {
            Integer foodBId = entry.getKey();
            Double similarity = entry.getValue();
            if (!treeCandA.containsKey(similarity)) {
                treeCandA.put(similarity, new HashSet<Integer>());
            }
            treeCandA.get(similarity).add(foodBId);
        }
        finalCandidates.add(treeCandA);
    }
    candidates = finalCandidates;
}
```

Figure 5.12: Weighted matching method

6 Extensibility

6.1 LanguaL classification

The hierarchical structure of the LanguaL classification is not captured in the SFCD, such that the parents of a facet are not known. As soon as this hierarchy is available, a new matching step can be introduced using the LanguaL classifiaction facets. This matching step is analogous to the multiple facet matching of the FoodEx2 classification or the own classification. In the `fillFacetsB()` method of Figure 5.3, it is shown how the parent facets for the single facets are retrieved. The `fillFacetsB()` for the multiple facet matching using FoodEx2 facets looks similar, but with a list of facets. For the LanguaL classification the method would look like the method for the multiple FoodEx2 facets, except replacing the FoodEx2 classification by the LanguaL classification. The similarity scores would be computed in the same manner.

6.2 Combination of multiple facet classifications

Instead of only using one multiple facet matching to match the foods, a combination of several multiple facet matchings using different classifications could be implemented. As an example, in one matching step the similarity of two foods is computed using the FoodEx2 classification and in another matching step the similarity is computed using the Swiss classification. The method `matchWithWeights()` in Figure 5.12, which represents the weighted matching scenario, shows how several matching steps can be executed independently and can then be combined using weights. A similar scenario, or rather subscenario, can be implemented to combine several multiple facet matchings.

6.3 Combination of food name matchings

Instead of only using the English names of the foods, the food names in other languages or synonyms could be used to match the foods. As in the Section 6.2, where the idea of combining several multiple facet matchings is described, several food name matchings could be combined. Instead of using the class `FoodNameMatching`, which compares the English names, a new class could be implemented which replaces the English names by the scientific names for example. The two name matching steps could then be combined as in the previous section.

6.4 Similarity methods for name matching

Instead of using the Levenshtein distance or the Jaccard similarity, other similarity metrics could be used provided by the DéjàVu library such as `SmithWatermanDistance`. A new method must be implemented in the `FoodNameMatching` that replaces the `JaccardSimilarityWithTokenizer` in the method depicted in Figure 5.7 by the new similarity metric.

6.5 Additional food data sources

If not only foods of the Food Composition Database or of Total Diet Studies should be matched, but for example foods of food consumption studies, a new Java class, e.g. `ConsumptionFood`, can be generated which maps the food data from this food consumption database table to Java objects of this class. A new constructor has to be implemented in the `FoodObject` class, similar to the constructor shown in Figure 5.2, where the necessary information of the consumption foods for the food matching is extracted and mapped to `FoodObject` instances. Then food sets of the different data sources can be matched, because they are all represented in the same way.

7 Evaluation

7.1 Test data

As test data, food composition data of Denmark and France was used. The France set used for testing contained about 1500 foods, the set of Denmark about 1000. In both food sets about half of the food items are classified with the FoodEx2 classification. Also food sets of UK and the SFCD data were used for testing during the development process of the Food Matching Tool.

7.2 Parameters

For the evaluation the similarity thresholds of Figure 7.1 were chosen. For the weighted matching evaluation the weights of all matching steps were set to 1 as also shown in Figure 7.1. The similarity thresholds of both facet matching steps are chosen to 0.5, but the threshold for the name matching is 0.3. The threshold of 0.5 was too high for the `NGramWordTokenizer` similarity, such that quite similar names were not found.

matching step	similarity threshold	weight of step
single facet matching	0.5	1
multiple facet matching	0.5	1
food name matching	0.3	1
exploitation	1.0	1

Figure 7.1: Similarity thresholds and weights for the evaluation

The Levenshtein distance for the name matching does not work well for food names which is why the `NGramWordTokenizer` was used for the evaluation.

7.3 Matching of fruits

The Food Matching Tool was evaluated with fruits of the France food set. The fruits were selected manually and put into a new set, called the fruit set here. Afterwards this fruit set was matched with the Denmark food set, which still contained all types of foods, not only the fruits. The fruit set contains 134 foods which are related to fruits, including for example fruit juices or fruit cakes. 77 of them are classified with FoodEx2 facets.

7.4 Evaluation scheme

For the evaluation, the results were assigned to the properties listed below. For the evaluation of the Food Matching Tool, the expected matches of the fruit set and the Denmark set must be known. For each food of the fruit set, the best food of the Denmark set was chosen manually by deciding with human knowledge if two foods are similar. The matches are distinguished by perfect matches and best matches. A perfect match means that there is a food in the Denmark set that represents the same food as the food of the fruit set. For example, 'apple, raw' and 'apple, fresh' is considered as a perfect match. If no perfect match is available, best matches of the foods are considered. For example, if no general apple item is available in one of the sets, then a best match for 'apple, raw' could be 'apple, danish, raw'.
After executing the weighted matching and the sequence matching on the fruit set and the Denmark set, the proposed matches were analysed. For each food of the fruit set it was examined, if the perfect match, or the best match respectively, was proposed in the matching candidates. The following list shows the evaluation criteria:

- Perfect match found: The perfect food of the Denmark set is proposed for a food of the fruit set by the Food Matching Tool.
- Perfect match not found: The perfect food of the Denmark set is not proposed for a food of the fruit set.
- Perfect match not available: There is no perfect match for a food of the fruit set available in the Denmark set.
- Best match found: The best food of the Denmark set is proposed for a food of the fruit set.
- Best match not found: The best food of the Denmark set is not proposed for a food of the fruit set.
- Best match not available: There is no best match for a food of the fruit set available in the Denmark set.

7.5 Sequence of steps vs. Weighted matching

The results of matching the fruit set and the Denmark set using the parameters mentioned above are presented in this section. In the table of Figure 7.2 the absolute values of the weighted matching and the sequence matching are shown. For example, for 53 foods of the fruit set, which contains 134 foods in total, the perfect match of the Denmark set was proposed by the Food Matching Tool. For the perfect and the best matches that were found, the average rank of the foods in the matching candidates ranking is shown in the table. For example, if a perfect match was proposed by the Food Matching Tool its rank was 1.21 in average, where rank 1 is the best rank and means that the food was proposed as best match. The table of Figure 7.2 shows further the difference between the weighted matching and the sequence matching. The number of best and perfect matches that are not found is twice as much in the sequence matching as in the weighted matching. However, if the perfect or best match is found, the average ranking of this match is higher in the sequence matching than in the weighted matching.

	perfect match found (rank)	perfect match not found	perfect match not available	best match found (rank)	best match not found	best match not available
Weighted matching	53 (1.21)	1	80	38 (2.03)	16	26
Sequence matching	46 (1.04)	8	80	26 (1.46)	28	26

Figure 7.2: Absolute results of the fruit matching

In the following, relative values of the evaluation are presented. If the foods of the fruit set are excluded, for which no best matches and therefore no perfect matches are available in the Denmark set, the number of remaining fruit foods is 108. The relative values are listed below and illustrated in Figure 7.3.

The results of the weighted matching:
Relative value of perfect matches found: $53/108 = 0.49$
Relative value of best matches found: $38/108 = 0.35$
Relative value of perfect and best matches not found: $17/108 = 0.16$

The results of the sequence matching:
Relative value of perfect matches found: $46/108 = 0.43$
Relative value of best matches found: $26/108 = 0.24$
Relative value of perfect and best matches not found: $36/108 = 0.33$

In the sequence matching, the case where for a food of SetA no match is proposed, appeared more often than in the weighted matching. In the sequence matching, for 26 foods no match was proposed, even though there exists a best match. In the weighted matching, only for

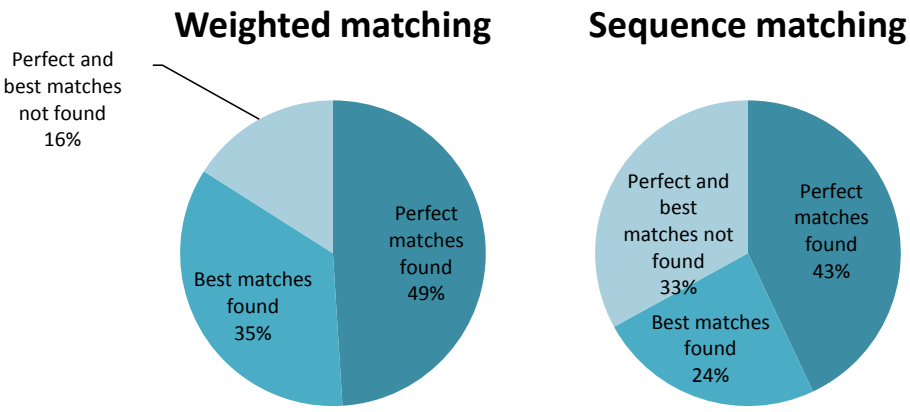

Figure 7.3: Relative results of the fruit matching only considering the fruits for which a perfect or best match is available

3 foods no match was proposed. The reason is that in the facet matching also the foods are excluded, that are not classified with the classification used in the facet matching. As mentioned above, only half of the data in both test sets is classified with FoodEx2 facets. Therefore, it happens that the perfect or best matches are excluded in the facet matching of the sequence matching, such that for the name matching step only foods are left with a name similarity smaller than the given threshold. This results in an empty candidate set and no match is proposed.

7.6 Exploitation

For the Denmark set and the France set, no data of previous versions was available for the evaluation. To test the Exploitation step, the food sets were matched twice, such that the top ranked matches of the first run could be stored as matches into the matching storage and in the second run these matches could be retrieved by the Exploitation step. In the sequence matching, the matches found by the Exploitation are just added to the proposed matching candidates. In Figure 7.4 the proposed foods for 'Pineapple, canned' are shown. The top ranked match is 'Pineapple, canned' with similarity 1.0. The match found by the Exploitation step is indicated with the similarity *NaN*. It is the same as the top ranked food, which is what is expected as we previously stored this top ranked food in the matching storage.

foodA: Pineapple, canned
NaN Pineapple, canned
1.0 Pineapple, canned
0.6666666666666666 Pineapple juice, canned
0.42105263157894735 Apple juice, canned or bottled
0.38461538461538464 Pear, canned
0.35714285714285715 Peach, canned

Figure 7.4: Proposed matches for sequence matching with Exploitation

In the weighted matching, the matches found by the Exploitation are combined with the matching candidates of the other matching steps. In Figure 7.5 on the right, the proposed matches for 'Apricot, dry' are shown after the first execution of the matching. On the left, the proposed matches of the second execution are shown. As the top ranked match of the first execution is retrieved as Exploitation match in the second run, the top ranked food gets a higher similarity score, as expected.

foodA: Apricot, dry	foodA: Apricot, dry
2.475 Apricot, dried	1.475 Apricot, dried
1.0 Raisin , seedless	1.0 Raisin , seedless
0.75 Apricot, raw	0.75 Apricot, raw
0.6 Strawberry, jam	0.6 Strawberry, jam
0.6 Fig, dried	0.6 Fig, dried
0.6 Apple, dried	0.6 Apple, dried
0.3 Apricot, canned, light syrup pack	0.3 Apricot, canned, light syrup pack

Figure 7.5: Proposed matches for weighted matching with Exploitation on the left, without Exploitation on the right

8 Discussion

8.1 Justification of facet matching

The similarity function that is used to compute the similarity scores of two foods in the single and multiple facet matching, was described in Section 4.3.4. Here it is shown that this similarity function fulfils the requirements for a similarity function that were listed in Section 4.3.2 and 4.3.3.

8.1.1 Common facets and their levels

If two foods of SetB have common facets with the foodA, the SetB food with more facets in common or with common facets of a lower lever should be ranked higher than the other food. In Figure 8.1 an example is shown where these two approaches are mixed. foodA of SetA has the multiple facets {f6, f7, f8}. food1 of SetB has the multiple facets {f6, f7}. It has two facets in common with foodA. food2 of SetB has only one multiple facet, namely $f8$. It has one facet in common with foodA. Even though food1 has more facets in common with foodA, the similarity score is equal as shown in Figure 8.1, because the level of the facet of food2 is lower than the levels of the facets of food1. The similarity function realises this correctly.

8.1.2 Parent facets

If two foods of SetB have common facets with the parent facets of facets of foodA, the parent facets of lower level should contribute more to the similarity score. In Figure 8.2 an example is shown to illustrate this case. foodA of SetA has the multiple facets {f6, f7, f8}, food1 of SetB the facets {f6, f8}, food2 of SetB the facets {f5, f6} and food3 of SetB the facets {f2, f6}. $f5$ and $f2$ are both parents of $f8$, but $f5$ has a lower level in the facet tree than $f2$. The similarity function treats this case correctly, because the common parents of the facet sets are

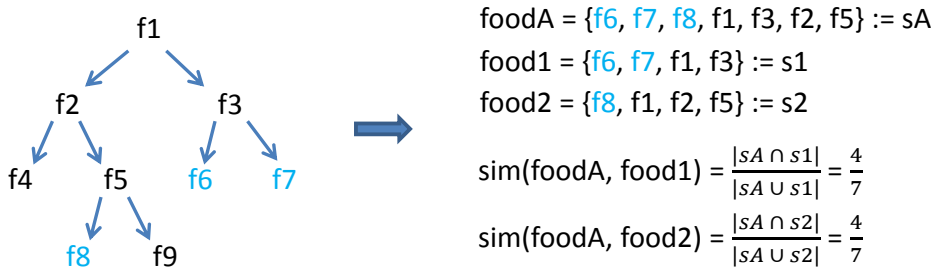

Figure 8.1: Common facets of different levels

captured in the intersection of the facet sets and for parents on a higher level this intersection is smaller.

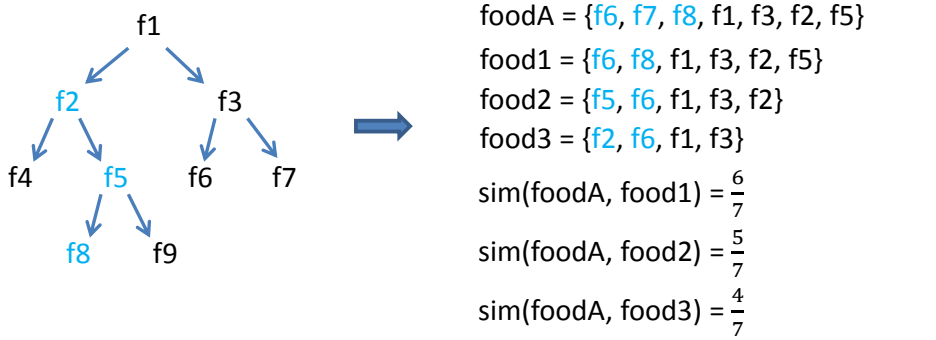

Figure 8.2: Common parent facets

8.1.3 Other facet matching approaches

Other approaches to compare the facets were considered before coming up with the approach above. In Chapter 2 of [2] the graph matching problem is discussed and several approaches are summarized. The goal of the graph matching is to determine which vertex of one graph corresponds to which vertex in the other graph. As the facets are organized in a tree structure and therefore in a graph, this approach was considered. The facet tree, which consists of all facets of a classification, is considered in the following as the base tree. The facet tree of a food is here defined as the tree consisting of the facets the food was classified with, and their parent facets. Defining the trees like that, the facet trees of foods are subtrees of the base tree. Because of the base tree, each facet has its unique position in the facet tree. Therefore, if two facet trees are matched, it is already defined which facet of one tree is matched to which facet in the other tree. The graph edit distance and maximum common subgraph techniques, which are mentioned in [2], were considered to match the facet trees. The graph edit distance is similar to the edit distance for text. The number of operations are counted to transform one graph into the other by inserting, deleting or relabeling vertices and edges. For the facet trees only the inserting and deleting of vertices must be considered, as the position is determined. The maximum common subgraph technique was considered as the two facet trees are both rooted in the root of the base tree. Starting from the root and following

the branches downwards the tree, the maximum common subgraph could be determined. The size of the maximum common subgraph could be interpreted as the shared facets of the two foods. The approach which was described above was used finally because it is simple to implement and captures the properties that were required for the similarity measure.

8.2 Justification of Exploitation

In this section the implementation of the Exploitation step is discussed. As mentioned in Section 5.5 the implementation of the Exploitation does not consider the version of SetB. If the Exploitation finds matches with a newer version as the version of SetB, their foodIDs can also be considered as matches.
In the Exploitation it is assumed that all foods of schemaB are related, such that a food of schemaB belongs to the SetB itself or to a previous version of SetB. If food sets which are not related with SetB are present in the same database table as SetB, it cannot be assumed anymore that foods with the same foodIDs represent the same food, because foodIDs of foods of not related sets are not related either. However, the aim of the Exploitation step is to propose possible candidates which the user has to investigate to determine the correct match.

8.3 Variants of food name matching

In the Food Matching Tool implementation only the English names are used for the food name matching. But also other names can be used, such as the original name, which is the name of the food in the language of the country, where this food was stored originally. For example the food names of a food consumption study in France are stored in French, or the food names of a food composition database in Germany are in German. For the matching also synonyms of the food names, such as peanut and groundnut or aubergine and eggplant, can be taken into account, scientific names, or the food names in different languages, such as Italian name. Even the brand name of a food could be useful because sometimes the brand name is contained in the original name, as for example 'Nutella'.
In the following an approach of the food name matching is described which uses a combination of all food names. Each food name of a food in SetA is compared to all names of the foods in its candidate set. Figure 8.3 shows an example: Both names of food11, namely 'original name' and 'synonym', are compared to all names of food21. Also in this approach a similarity measure is needed to compute the similarity of two foods.
There are several possibilities how to define the similarity measure and how to compute the ranking of the candidate foods. To describe these possibilities the following example and definitions are used. At the beginning of the food name reduction step, food21 and food22 are in the candidate set of food11, such that each name of food11 is compared with all names of food21 and all names of food22. In Figure 8.3 it is shown how each name of food11, namely the original name and a synonym, are compared with all names of food21. In the following, a 'pair of names' consists of two names that are compared, e.g. {orig. name11, orig. name21} or {orig. name11, synonym21}. For each of these pairs the score is computed with the similarity methods explained in the Section 5.3.2. The scores could then be combined as listed in the following to get the ranking score of the candidate foods.

- To get the ranking score of a candidate food, take the score of the pair that has the

highest score of this candidate food. A problem of this scoring is that if the brand names of two foods are the same, {brand name, brand name} gives the highest score of all pairs, but maybe the foods are only of the same brand but are not the same foods.

- An average of all name pair scores of a candidate food is computed as score of the candidate food. A problem of this scoring is that if two names that have nothing in common, e.g. {orig. name (in French), Engl. name}, are compared, this name pair score has negative influence on the score of the candidate food.
- An average of the name pair scores is computed for the score of the candidate food, but a name pair score is only included in the average if the name pair score is higher than a certain threshold. Name pairs, where the names have nothing in common, have a very low score. Therefore, these pairs are not included in the candidate food score. It still can happen, that for example only the brand name pair passes this threshold, such that the candidate food gets a high score even the foods are not the same. To avoid this case the brand name pairs could be weighted less than other pairs.

Custom settings: The user can specify in which languages the food names of the SetA are available, such that not all name combinations have to be compared. The user can also specify the threshold, which is used to determine which foods remain in the candidate set and which ones are excluded.

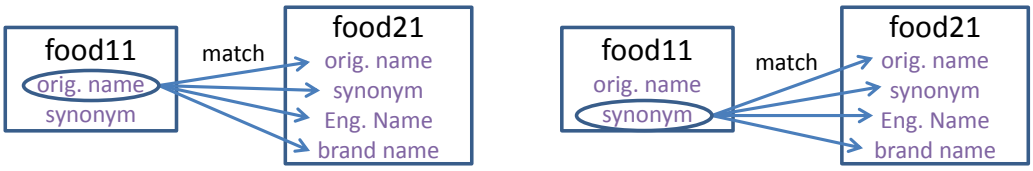

Figure 8.3: Food name comparison

8.4 Brute force matching

Originally, there was a brute force matching planned, which should be applied, if the sequence matching would exclude too many candidates from the candidate set, such that in the end no matching candidate is left. The idea was to use the food name matching again, but with a non reduced candidate set, that still contains all foods of SetB. For the sequence matching scenario this would make sense. Later, the idea of the weighted matching came up, such that the matching steps are executed with independant candidate sets. In this scenario the brute force matching would just be a repetition of the food name matching and would not make sense. Since the weighted matching turned out to be more useful than the sequence matching, it is recommended to use the weighted matching anyway and therefore a brute force matching is not needed. If a sequence matching does not yield results that are good enough, the weighted matching has to be executed instead of a brute force matching.

9 Conclusion

9.1 Contribution

The analysis of the Swiss Food Composition Database showed that in general the quality of the data is good. However, some data cleansing issues could be revealed. Most of the existing issues of the data cannot be automatically cleaned such that a tool that automatically cleans the data cannot be used. The data cleansing of the SFCD requires manual cleansing, such as reading through documentation to get missing information.
The Food Matching Tool, which was developed during this Master thesis, uses food names and food classifications to compute similarity scores between the foods of the two sets. The tool provides two matching scenarios, the weighted matching and the sequence matching. In the weighted matching, different matching steps are applied independently and the resulting similarities are combined to get a ranking of which foods of one set are the best matches for foods of the other set. In the sequence matching, the matching steps are applied in sequence, such that after each matching step some foods are excluded from being the best match of a food. Only the similarities of the last matching step are used for the ranking. In the evaluation the weighted matching achieved better results. The architecture of the tool allows several possibilities how to extend the tool. Additional ideas how to further improve the tool are discussed in the following future work.

9.2 Future Work

9.2.1 Classification mapping

If the two food sets that should be matched do not have a classification in common, the facets of a classification of SetA could be mapped to facets of a classification of SetB. For example, if the foods of SetA are classified with the FoodEx2 classification and the foods of SetB with the LanguaL classification, a mapping of the two classifications could be defined that maps a FoodEx2 facet, such as 'Biscuits (sweet and semi-sweet)', to a LanguaL facet, such as

'Biscuits/Cookies'. Then this mapping can be used to find common or at least similar facets of two foods.

9.2.2 Facet matching

As described in Section 4.3.3, in the facet matching common parent facets do also contribute to the similarity score of two foods, but not as much as common facets. Another approach could be to go manually through the facets of a classification and distribute weights individually for each facet, so that facets that seem to be more important for the matching get higher weights than facets that are less relevant.

9.2.3 LanguaL classification

As mentioned in Section 6.1 the LanguaL classification could also be used to match food items. As soon as the hierarchical structure of the LanguaL classification is captured in FoodCASE, the multiple facet matching can be extended with this classification and the resulting matching candidates of this step can be included in the weighted matching scenario.

9.2.4 Food Consumption Data

As described in Section 6.5 additional data sources can be added, such that food items of this source can be matched as well. The Food Consumption Data is data about what food items people consume and how much of it. It could be interesting to match this data with the SFCD data, for example to compute how much of a certain nutrient people eat in average.

9.2.5 Combination of food names

In the Section 8.3, an additional approach was discussed how the food names could be used for the food matching. The approach of combining different types of food names, as for example the English names and the brand names, could be implemented as future work. In the Food Matching Tool it is only possible to match English names with English names. In Section 6.3 it is described how to extend the tool to match for example scientific names with scientific names. The goal of this future work idea is that any type of food name can be matched with any type of food name.

A Appendix

A.1 Java classes

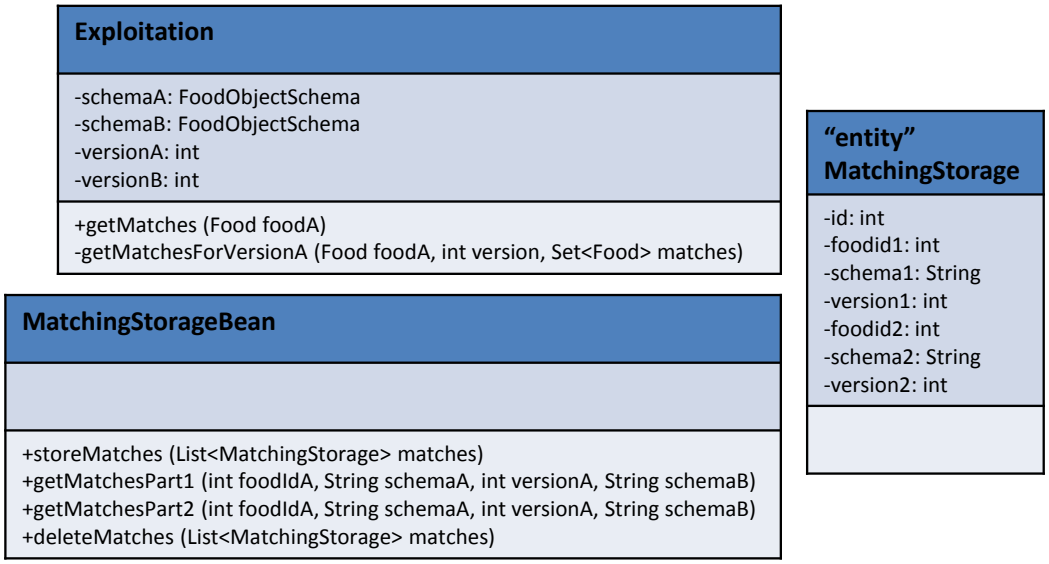

Figure A.1: Classes used for the Exploitation

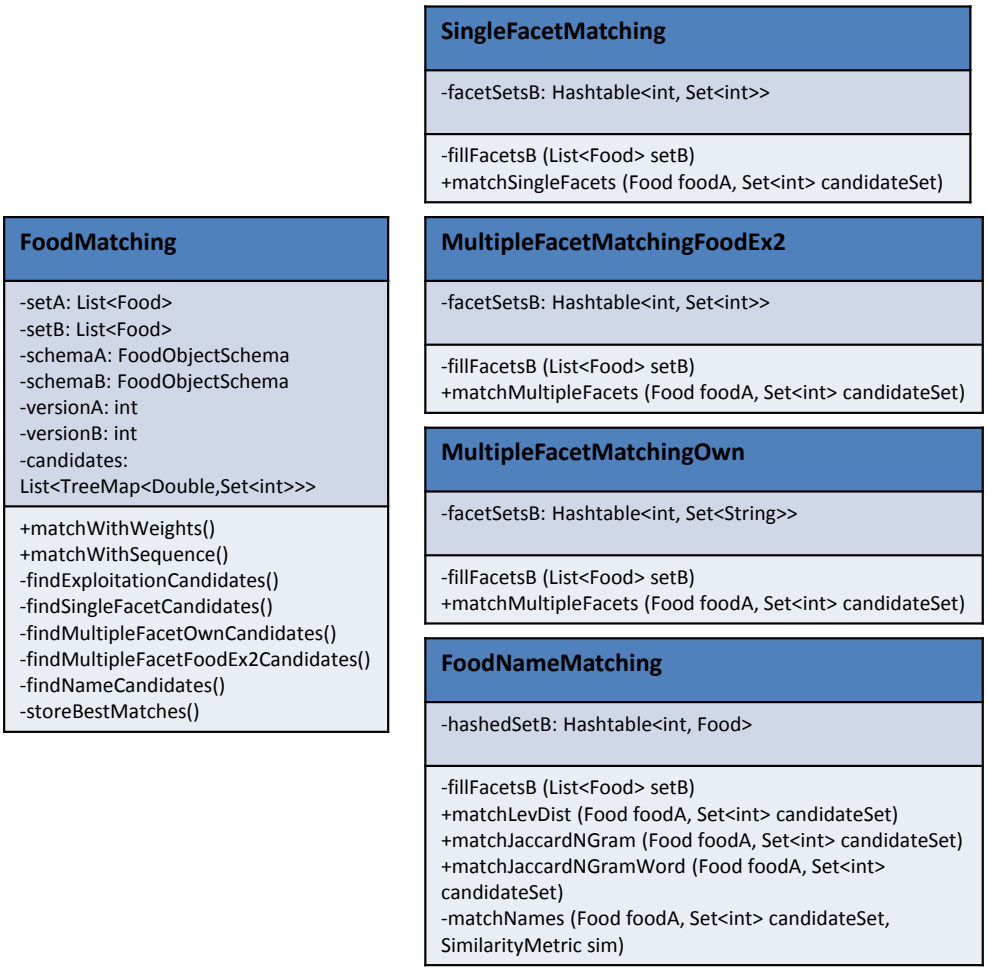

Figure A.2: Main class `FoodMatching` and the classes of the matching steps

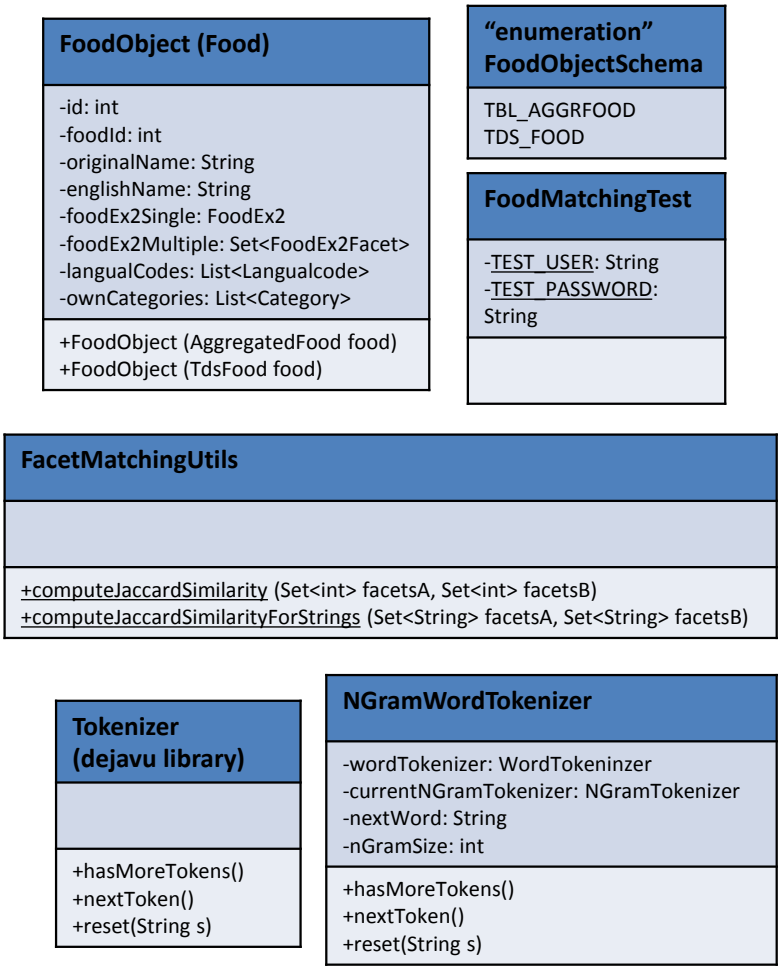

Figure A.3: Other classes

Acknowledgments

I would like to thank my supervisor David Weber for supporting me during my Master Thesis. Also I would like to thank Karl Presser for the interesting and helpful discussions.

B
Abbreviations

SFCD Swiss Food Composition Database

DQA Data Quality Analysis

TDS Total Diet Study

Bibliography

[1] C. Batini and M. Scannapieco. *Data quality: concepts, methodologies and techniques*. Springer, 2006.

[2] E. Bengoetxea. Inexact graph matching using estimation of distribution algorithms. *Ecole Nationale Supérieure des Télécommunications, Paris*, 2002.

[3] V. Ganti and A. D. Sarma. Data cleaning: A practical perspective. *Synthesis Lectures on Data Management*, 5(3):1–85, 2013.

[4] M. A. Hernández and S. J. Stolfo. Real-world data is dirty: Data cleansing and the merge/purge problem. *Data mining and knowledge discovery*, 2(1):9–37, 1998.

[5] S. Kandel, A. Paepcke, J. Hellerstein, and J. Heer. Wrangler: Interactive visual specification of data transformation scripts. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 3363–3372. ACM, 2011.

[6] J. I. Maletic and A. Marcus. Data cleansing: Beyond integrity analysis. In *IQ*, pages 200–209, 2000.

[7] H. Müller and J.-C. Freytag. *Problems, methods, and challenges in comprehensive data cleansing*. Technical Report HUB-IB-164, Humboldt University Berlin, 2003.

[8] E. Rahm and H. H. Do. Data cleaning: Problems and current approaches. *IEEE Data Engineering Bulletin*, 23(4):3–13, 2000.

[9] V. Raman and J. M. Hellerstein. Potter's wheel: An interactive data cleaning system. In *VLDB*, volume 1, pages 381–390, 2001.

[10] R. A. Wagner and M. J. Fischer. The string-to-string correction problem. *Journal of the ACM (JACM)*, 21(1):168–173, 1974.

[11] Wikipedia: Data cleansing. `http://en.wikipedia.org/wiki/Data_cleansing/`, August 2013. [Online; accessed 7-October-2013].