UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

Master in Artificial Intelligence

Supervised and Experiential Learning
Final Project

A Case Based Reasoning System for Cocktail Recommendations

Submitted by
Andrea Corda
Hrizantema Stancheva
Haoran Mo
Johannes Heidecke

January 19, 2018

Contents

List of Figures

1 Introduction

In this final project, we used a case based reasoning (CBR) system for the purpose of recommending cocktails to users based on a problem description containing information such as desired ingredients or the alcoholic level of the drink. The system has access to cases which already cover 108 different cocktail recipes. It will retrieve a similar case to a given query, adapt it, and propose it to the user, giving capabilities to change and repair the proposed solution.

In this section we will introduce the application domain and the CBR principles applied for the project. In section 2 we describe the requirements of the project, both from the user and technical perspective. This is followed by a description of the functional architecture in section 3. The implementation of the project is detailed in section 4, split up into the different stages of the CBR cycle. In section 5 we will present a system evaluation, followed by discussion and outlook in section 6.

1.1 Application Domain

The application domain of this CBR system is the space of delicious cocktail recipes. A cocktail is a drink containing two or more ingredients, usually with at least one of them containing alcohol. Most bars offer a wide selection of different cocktails on their menu and they are a popular choice, especially on weekends. In Spain, almost one in ten persons stated that they were drinking a cocktail at least once a week in 2016. This number was previously decreasing, but went back up in the last years (see figure 1.1).

Cocktails may contain a wide variety of ingredients, ranging from different liquors, juices, or sodas over fruits and vegetables to more exotic aromas such as wasabi or basil. When confronted with a list of multiple pages of cocktails in a bar, guests who are not frequent drinkers are often overwhelmed by the huge variety of options and are not sure what to pick. While they usually know their preferences for some ingredients, they will struggle to find the subset of cocktails which best matches their preferences.

For this purpose, our CBR system can be applied to quickly find a well suited cocktail based on the specification of the user's preferences and it can even carefully adapt existing recipes in a way that special requirements of the user are met. The system could be deployed on smart phones or as a website and can be used by any bar guest. In addition, the feedback provided by the users can then be used to add new knowledge (cases) to the system and to iteratively improve it.

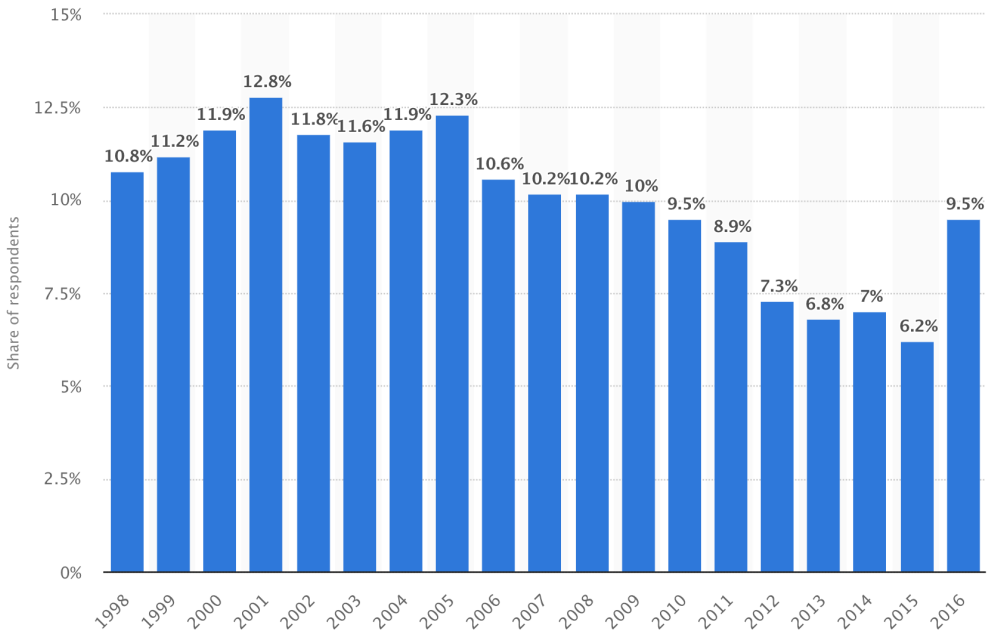

Figure 1.1: Share of individuals that consume cocktails weekly in Spain between 1998 and 2016. Source: statista

1.2 CBR Principles

Case based reasoning (CBR) was introduced in the late 80s and early 90s of the last century [1] [2] [3] and is based on the observation that human problem solving depends heavily of the memorization of experiences, which can be retrieved when facing a new problem and adapted to the given circumstances. Instead of solving a new problem from scratch, similar instances of the given problems are retrieved from the case library and adapted to fit the special situation. CBR systems become better over time since each new experience can be stored as a case to be used in the future and they can learn from mistakes and prevent from making them again. In this way, using the CBR system in the real world is a way of learning.

In this section we will describe some of the main principles we decided for the implementation of our CBR system.

- **Case library structure**: Given that we expect the number of cases to stay relatively small, we use a *flat memory structure* for our case library. This will enable fast adding of new cases and a relatively easy implementation of the case library. If the system grows significantly in size, a change to a hierarchical structure might be considered (e.g. splitting up cases by their alcohol strength category into alcohol free, low alcohol, ...).

- **Case structure**: The cases will contain a field of attributes with the problem description and another field of attributes containing the corresponding solution. For each case we will persist a utility measure, containing information about how useful the case has been in the past whenever it was retrieved. The structure will be explained in more detail in section 4.1.1.

- **Distance metric**: We define a custom distance metric to retrieve the best matching cases to a new problem using a simple nearest neighbor (NN) algorithm. The distance metric can be parameterized with custom weights for the different features, according to some expert estimation of which preference features are more important. It can deal with empty values and different types of data. Since we have chosen a flat memory structure, the retrieval will always be based on the entire case library, calculating distances to all instances.

- **Adaption strategy**: We use a *structural adaption strategy*, enhancing the found solution in a way that it conforms better to the preferences stated in the new problem. For more details see section 4.3. In some cases with a good fit even a *null adaption strategy* could be used.

- **Evaluation method**: The users will be asked for direct feedback for found solutions to their problems. Since this step depends on the subjective experience of human users, it can't be simulated and we will require the user's expert feedback to evaluate the system and to decide if new cases will be stored and if and how they have to be repaired.

- **Learning by observation**: The case base will not start empty but will already contain a set of cases, one example case for each of the 108 known recipes of our data base. While these cases are not observed from real human behavior, they will give a good baseline for the system to be usable from the beginning on. Initial cases which were regularly evaluated to be bad solutions will be automatically pruned from the case library.

- **Learning by experience**: The system will learn from each interaction with users. After retrieving and adapting a solution to a given case, the user will be asked for both a quality evaluation and for proposals how the solution could be enhanced or repaired. Only solutions which were evaluated to be good by the user will be added to the case base and only if they are significantly different enough from existing cases. If an obtained solution was evaluated to be bad, this will be stored for the case it was based on. In this way, 'bad' cases of the library can regularly be automatically removed. At the same time, the user feedback will enable the system engineers to retroactively change and enhance the adaption rules.

2 | Requirement Analysis

2.1 User Requirements

The users of our system will regularly be faced with having to choose a cocktail from many different options without knowing exactly which ones fit their preferences best. The user have the following requirements:

- **Specifying alcoholic degree**: The user wants to specify roughly how strong he wants the cocktail to be. This can range from alcohol free over low up to strong cocktails. This will often be a very important criterion for the user since it will influence the level of intoxication. Moreover the user may want to have a non-alcoholic cocktail. In this case the system have to suggest a non-alcoholic cocktail, because the reasons behind the user's decision may be that he or she is supposed to drive after that, they may concern his or her health or other reason.
- **Specifying desired ingredients**: The user wants to give a list of ingredients he or she prefers and would like to have them in the cocktail. No information on the ingredient quantities should be provided from the user, the original quantities will be used to prepare the new cocktail's recipes. The ingredient's names are provided by the system.
- **Inclusion of similar ingredients**: The user expects the system to suggest him ingredients similar to the ones he wants. A suggested ingredient should be from the same category as the wanted one. For example if the user wants an orange fruit to be included in his cocktail, the system will not suggest orange juice, but a lemon fruit, which is from the same category and is the most similar one from all the others that are offered.
- **Flexible level of preference detail**: The system is supposed to offer a flexible specification of the ingredients to be included in the cocktail. The user may specify the ingredients he wants for one or more of the existing categories. Also the number of the wanted ingredients for each category is unlimited. For example the user can specify that he wants a cocktail with rum and also he can specify that he wants a cocktail with rum, lemon, orange, coconut and something else.

2.2 Technical Requirements

- **Maximum time response of the system**: Since this software is supposed to be used in a non-formal setting, the requirements for the time response of

the system are not that strict. However the maximum satisfactory response time to be experienced for this project is 5 seconds.

- **Maximum memory size of the system**: The limit for the maximum memory size is 50 MB since we want to run the software it on mobile devices because of its nature.
- **Open source** Another technical requirement for the software is to be an open source so that it can participate in the annual Cocktail challenge on making real cocktails competition.

3 | Functional Architecture

In this section we will describe the architecture of the CBR system and how it was designed to interact with the user. In figure 3.1 we can see a schematic diagram of the interaction between the system's different modules and human input. The entire case library is persisted in a XML file which is updated whenever cases are changed, added, or removed.

The user interacts with the system in different parts, depicted in the diagram with the orange boxes. Towards the beginning of each CBR cycle, the user will specify his preferences as a problem, which will be passed to the retrieval module. Later in the cycle, the user will serve as an expert to evaluate the quality of the found case and adaption and to propose possible repairing before any new case is stored to the data base. More details about the case library can be found in section 4.1.

The retrieval module will be triggered upon receiving a new problem description from the user. It has access to all the cases stored in the case library and calculates the distance between the new problem and all of them to find the nearest neighbor. For the distance metric is uses domain knowledge such as the similarity of different ingredients (e.g. orange-lemon: 0.9) and the weighting of importance of different features (e.g. alcohol-free problems should not be close to cocktails with strong alcohol). The retrieval module is described in more detail in section 4.2.

The retrieved case is then passed to the adaption module. This module also uses domain knowledge to enhance the found solution and make it more suitable for the preferences defined in the new problem. The adaption module also has access to the domain knowledge base. More details about the adaption process are described in section 4.3.

Finally, the retrieved and adapted solution is shown to the user. The user is asked to serve as a human expert to judge and evaluate the quality of the found solution and its adaption. This feedback and the adapted case are passed to the evaluation module which takes care of storing new cases and removing bad ones. More information on this part is written in section 4.4.

The user interacts with the system with simple shell inputs via a keyboard. No graphical interface was implemented, but the programmatic interfaces are available to easily extend the system with a GUI at a later point.

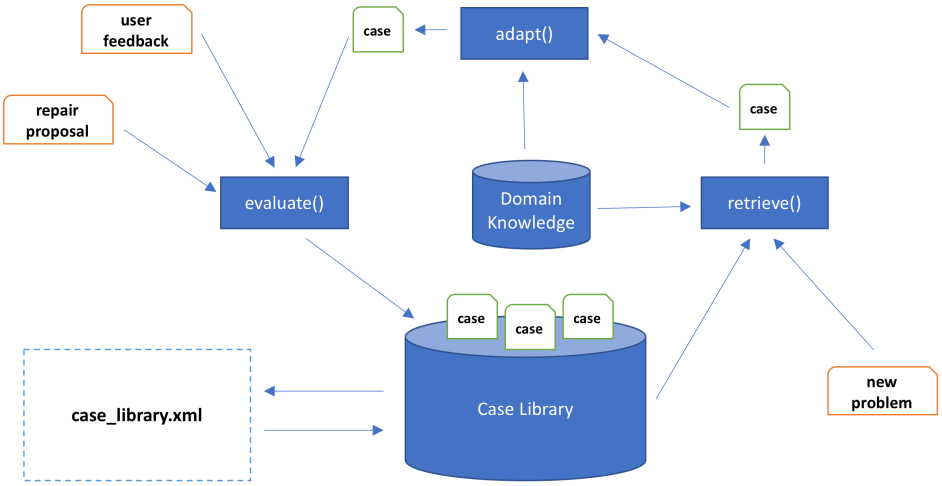

Figure 3.1: Functional architecture of the CBR system

4 | Implementation

4.1 Case Base

4.1.1 Case Structure

A case is composed by a problem and a solution. The problem is used to specify the characteristics required for the solution. We identified 6 possible types of specifications:

- Alcoholic liquor
- Non alcoholic liquor
- Fruit
- Vegetables
- Alcohol content
- Other

Based on these it is possible to express the requirement of a cocktail. The solution is indeed the recipe of a cocktail. It contains informations about the ingredients, their types and their quantity. Moreover it encludes the necessary steps for the preparation of the cocktail, together with the alcoholic percentage of the drink. This has been calculated starting from the alcoholic level of each drink, the formula used is:

$$d = \text{drink}$$

$$n_d = \text{number of liquid ingredients for the drink}$$

$$q_i = \text{amount of ingredient i in ml}$$

$$q_t = \text{drink's total quantity}$$

$$alcohol\_content(drink) = \sum_{k=1}^{n} \frac{q_i \cdot alcohol\_percentage(ingredient_i)}{q_t}$$

The alcohol quantity calculated using this method has been stored in the dataset xml file in order to be able to use this information. Moreover, the alcohol percentage is categorized for the definition of the problem. The categories identified are four:

1. None: alcohol free drink
2. Low: alcohol ≤ 15 %
3. Medium: alcohol $> 15 \wedge$ alcohol ≤ 20 %
4. High: alcohol > 20 %

4.1.2 Case Library

Case library is the manager of all the cases. It uses a flat memory to store the cases. More precisely cases are stored in a list, while problems and solutions are stored separately into dictionaries in order to make the retrieval process faster and more reliable. It exposes functionalities to the overlying layers, such as:

- Store new problem
- Store new case
- Update solution parameters

Each change will be loaded in memory and saved to file. This way the learning algorithm can retrieve information about the past usage and be able to improve the solutions. Beside the data related to cases in general, we decided to extend the purpose of the Case Library including in it also the similarities between ingredients. We started from the assumption that an ingredient is only comparable with another belonging to the same category. Similarities are exposed too as dictionaries, divided per category. So, in order to access the similarity information, it is enough to access the dictionary this way:

$$category = \{c | c \in \{"alcoholicLiqueurs", "nonalcoholicLiqueurs",$$
$$"fruit", "vegetables", "tasteEnhancers", "other"\}\}$$
$$(ing_1, ing_2) = \{(ing_1, ing_2) | ing_1 \in category_1 \wedge$$
$$ing_2 \in category_2 \wedge category_1 = category_2\}$$
$$sim(ing_1, ing_2) = category["ing_1"]["ing_2"]$$

4.1.3 Creation of Initial Case Base

The creation of the case base has been conducted starting from data already present in an existing project [4]. Using these data we extended the information they contained, adding the alcoholic percentage and organizing the cases. Moreover we enlarged the datasets adding new ingredients and the similarities.

4.1.4 Cases for Testing

To test and evaluate our CBR system, we asked 10 individual human users to generate one problem for us and have them specified their expected solution. For this purpose, the users were given access to the list of all recipes, were asked to select one of the cocktails that they would

4.2 Retrieval

The task of retrieving cases in the case library is difficult, because the very nature of the structure of the case library, therefore a partial-matching strategy should be used. A retrieval method should try to maximize the similarity between the actual case and the retrieved one(s). This task implies most of the time the use of general

domain knowledge. The retrieving process of a case from the system's memory strongly depends on the case library organization. Our case library structure is a flat memory. Flat memories have an intrinsic problem of bad performance in time, so that the retrieval time is proportional to the size of the case library.

Selecting the best similar case is usually performed by means of some evaluation heuristic functions or distances, possibly domain dependent. They are usually named as nearest neighbor algorithms. The evaluation function usually combines all the partial matching through a dimension or attribute of the cases, into an aggregate or full-dimensional partial-matching between the searched cases and the new case. Commonly, each attribute or dimension of a case has a determined importance value (weight), which is incorporated in the evaluation function. The evaluation function computes an absolute match score (a numeric value).

A distance function is a function that defines a distance between each pair of elements of a set.

A metric on a set X is a function $d : X \times X \to [0, \infty)$ where $[0, \infty)$ is the set of non-negative real numbers and for all $x, y, z \in X$ the following 4 conditions should be satisfied:

1. $d(x, y) \geq 0$
2. $d(x, y) = 0 \Leftrightarrow x = y$
3. $d(x, y) = d(y, x)$
4. $d(x, z) \leq d(x, y) + d(y, z)$

The most common distance measuring method is based on the location of objects in Euclidean space, in which the distance is calculated as the square root of the sum of the arithmetical differences between the corresponding coordinates of two objects. We made a research which distance metric will be the most appropriate one for our domain. Initially we decided to implement a Gower's similarity measure.

Gower's General Similarity Coefficient is one of the most popular measures of proximity for mixed data types.

Gower's General Similarity Coefficient s_{ij} compares two cases x_i , x_j, and is defined as the average score over all partial comparisons:

$$s_{i,j} := s(x_i, x_j) = \frac{\sum_{k=1}^{d} w_k s_{ijk} \delta_{ijk}}{\sum_{k=1}^{d} w_k \delta_{ijk}}, x_i, x_j \in X$$

For any two vector objects x_i , x_j to be compared on the basis of feature k, a score s_{ijk} is defined, described next:

1. Set $\delta_{ijk} = 0$ when the comparison of x_i , x_j cannot be performed on the basis of feature k for some reason; for example, by the presence of missing values, by the feature semantics, etc.
2. Set $\delta_{ijk} = 1$ when such comparison is meaningful
3. If $\delta_{ijk} = 0$ for all the features, then $s(x_i, x_j)$ is undefined

It should be noted that the effect of the denominator is to divide the sum of the similarity scores by the number of variables; or if variable weights have been specified, by the sum of their weights.

The variable weight w_{ijk} for the comparison on the k^{th} variable is usually 1 or 0. However, if you assign differential weights to your variables, then w_{ijk} is either the weight of the k^{th} variable or 0, depending upon whether the comparison is valid or not. This allows larger weights to be given to important variables, or for another type of external scaling of the variables to be specified. If the weight of any variable is zero, then the variable is effectively ignored for the calculation of proximities.

There are some established rules for calculating s_{ijk} depending on the type of the variable, whether it is ordinal or continuous, binary or nominal but we had to develop a specific solution to serves our needs, because of the structure of our problems. A problem consist of 7 different sections: *'fruits', 'vegetables', 'alcoholicLiqueurs', 'nonalcoholicLiqueurs', 'tasteEnhancers', 'other', 'alcoholic percentage'* from which the values of the first 6 are lists of different ingredients. This means that we have to define a function to compare two different lists. Our solution is for each of the elements of the first list to add the similarity value of the most similar element of the second list and to divide this to the length of the list, then we do the same for the elements of the second list. Then we find the average of the two values and the number we get when we subtract this from one is the actual distance. For the 'alcoholic percentage' attribute it was only considered whether the value is the same or not, since the possibilities for it are *"None", "Low", "Medium", "Strong"*.

After experimenting a little bit with this distance function the collected results were not satisfactory, therefore weights for each of the categories were included. From the point of view of the user, the alcohol content is the most important feature when choosing a cocktail, therefore we chose to put the highest weight to the alcohol content with value of 10. Furthermore we considered the difference between an alcoholic and a nonalcoholic cocktail which affected the decision for the distance between the value *"None"* and the other three values *"Low", "Medium" and "Strong"*, which imply that a cocktail contains alcohol. This reasons made us specifically define the distances between all the possible values for the alcohol content. We also took into account that the preferences for the alcoholic liquors and nonalcoholic liquids are more important than the other features, and that the vegetables and other ingredients have the least importance. After including this considerations in the implementation of the distance function we achieved satisfactory results in testing the similarity between different problems in the given domain, while keeping the 4 conditions for the distance metric fulfilled.

Since the structure of our case library is flat in order to find the most similar case we have to compare the one we have to all the cases in the library. After making all the comparisons we simply return the case which has the smallest distance to the given one. We decided to include the support measure in our calculations. The support represents the number of common features between the two cases for which we calculate the distance. In case we have more than one case with the lowest distance we pick the one with higher support, namely the one that has more features in common with the problem we need to solve.

4.3 Adaption

Our adaption component was inspired by the ReMind Cognitive Systems, and be implemented especially for the cocktail problem.

The inputs of this function are the original problem, and original solution 1. The function has 4 steps, the first step is to filtrate some field like alcoholic percentages, which has a clear range from None, Low, Medium, to Strong. The second step is to discriminate if the solution obtained currently satisfy the problem or not. If it satisfies all requirements, then just return the original solution, otherwise, go to step three. The third step will find out the unsatisfied part, for instance, one wanted a drink with the fruit orange, but there was no orange in the solution. Retrieve the unsatisfied part and find out the best solution named solution 2. If the new solution reached what user wanted, then merge solution 1 and solution 2 as the final solution 3, otherwise, add the required ingredient without the details of how to process this ingredient.

In a word, this component finds the difference between input case and retrieved cases to obtain the adjustment, and then improve the solution. The program flow of adaption is shown below.

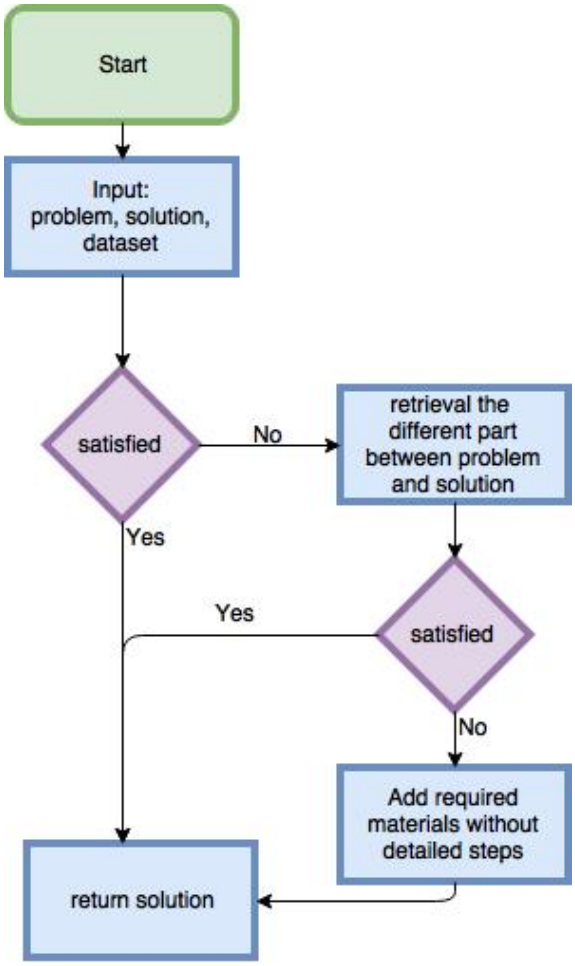

Figure 4.1: Adaption Diagram

4.4 Evaluation and Learning

The last step in the CBR cycle is concerned with evaluating the obtained solution and using this to learn and improve the system. As was stated in section 1.2, we will use direct feedback of the user to judge the quality of obtained solutions. For this purpose, the user will be asked three questions after being provided with an adapted solution:

- **Usefulness of adapted solution**: The user will be asked if he found the final, adapted solution for his problem description useful. This information will be used to keep track of the utility of cases in the library and as well to decide if a new case will be stored.
- **Usefulness of the adaption**: The user is asked if the adaption improved the solution. This statistic can be used by the system's engineer to revise adaption methods and fine tune their parameters.
- **Proposal for improvement or repairing**: Some solutions might not be sufficiently good. However, the user has the possibility to define possible repairs and enhancements in free text. These proposals can later be processed in order to make even failed cases a useful part of the case library.

4.4.1 Criteria for storing a new case

New cases are stored whenever they were judged to be useful by the human expert and are not a simple duplicate of another existing case. This is important since in the flat memory structure we need to keep the case library sufficiently small to ensure acceptable response times.

4.4.2 Criteria for removing existing cases

Given the sensitivity of the CBR system to growing too large, and the negative impact of having bad cases stored which produce solutions of bad quality, we need to regularly 'clean up' the library. While this method has not been completely implemented for our purpose (the case library was always far from having too many cases), all relevant information and statistics for this step are already stored in the case library and persisted to the XML file.

5 | Evaluation

To evaluate the implemented CBR system, a test set of ten cases was manually created by different human users, see section 4.1.4. In this section, we will present the obtained results and analyze them qualitatively. First of all, let's have a look at some of the problem instances that our users created. The problems are encoded as a basic string containing key-value pairs such as this one:
`alcoholicPercentage:None,nonalcoholicLiqueurs:apple juice,vegetables:ginger,other:ice cube`
The corresponding desired solution to this problem is a *Cocktail KidiCana*.
The script `evaluate.py` can be run to find how many of the test cases returned the exact intended solution which was originally specified by the user. In the initial state of the case base with 108 cases, 60% of problems could be solved exactly as the users intended it. In the other 40%, in three out of four cases the users were still content with the found solution, even though it was not the intended one.
`alcoholicPercentage:None,nonalcoholicLiqueurs:apple juice,vegetables:ginger,other:ice cube`
The corresponding desired solution to this problem is a *Cocktail KidiCana*.

6 | Discussion and Outlook

For this project we implemented a fully functional CBR system which is very successful at finding cocktails matching to userś preferences. Especially for the retrieval part we used specific domain knowledge to construct an efficient distance metric with high success rate.
The systemś performance is limited by the quality of the initial cases, which were extracted from a non-perfect data set with some missing ingredients and other inconsistencies. Most shortcomings of the system can reasonably be attributed to missing or compromised data in the case library. For this reason, we expect big quality improvements to be possible by simply enhancing the case library and manually adding more cases, as well as enhancing the domain specific knowledge such as similarities between ingredients.
For future iterations of the CBR system, a very promising enhancement would be to add disliked ingredients to the problem description. In some of our tests we could observe that solutions were proposed which contained one or more ingredients that are disliked by the user. Obviously this is not a desired behavior, thus enhancing the problem description could significantly improve the quality of results

Bibliography

[1] Agnar Aamodt and Enric Plaza. Case-based reasoning: Foundational issues, methodological variations, and system approaches. *AI Communications. IOS Press*, 7:1:39–59, 1994.

[2] Janet Kolodner. *Case-based Reasoning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[3] Christopher K. Riesbeck and Roger C. Schank. *Inside Case-Based Reasoning*. L. Erlbaum Associates Inc., Hillsdale, NJ, USA, 1989.

[4] Vizkids. Case based reasoning using python, 2016.