Supervised and Experiential Learning

CBR for a cocktail recipe creator

Practical Work 3 - A CBR prototype for a synthetic task

Xavier Cucurull Salamero
xavier.cucurull@estudiantat.upc.edu
Daniel Hinjos García
daniel.hinjos@estudiantat.upc.edu
Fernando Vázquez Novoa
fernando.vazquez.novoa@estudiantat.upc.edu
Estela Vázquez-Monjardín Lorenzo
estela.vazquez-monjardin@estudiantat.upc.edu

Master in Artificial Intelligence
Universitat Politècnica de Catalunya
June 2021

Contents

1 Introduction

In this project we have implemented a Case-Based Reasoning system for a cocktail recipe creator. We have implemented a CBR engine in Python using a case library formatted as an structured XML. In order to obtain an efficient system, a hybrid structure has been used to index the cases, as will be explained in following sections. From the beginning, it has been clear that our CBR engine needed to implement the basic *retrieval*, *adaptation*, *evaluation* and *learning* phases from a Case-Based Reasoning system [1], so each of these phases have been implemented as methods of our CBR class. In the learning phase our method will be able to learn from experience, both from successful experiences and failures. In addition, various methods are proposed as a way for the user to interact with the system in order to obtain and evaluate recipes that fulfill their constraints.

In this report we present the technical details of our CBR system along with the tests that have been performed to assess its performance and robustness. In addition, a User Manual can be found apart from this document to obtain information about the use of the system.

1.1 Application domain

The application domain chosen for this practical work has been that of cocktail creation. For this project we got inspired by the mixology challenge of the Computer Cooking Contest 2017 [2] and the Cocktails dataset from Kaggle [3] which is in `.csv` format. Based on it, we have created a system that is able to suggest a tasty cocktail recipe that matches the user preferences, which may include the desired or/and unwanted ingredients, the available type of glass or the desired basic taste, among others.

The Cocktails dataset contains 473 cocktails indexed by name, it provides the required ingredients, measures and the recipe's preparation steps. Below we show a summary of the possible values of all the attributes except for ingredients.

- *Categories* (9): Cocktail, Shot, Beer, Milk/Float/Shake, Ordinary Drink, Other/Unknown, Punch/Party Drink, Coffee/Tea, Soft Drink/Soda.
- *Alcohol Type* (25): Creamy Liqueur, Vodka, Triple Sec, Sweet Liqueur, Rum, Tequila, Beer, Bitters, Schnapps, Whisky, Gin, Brandy, Vermouth, Champagne, Cider, Campari, Wine, Sambuca, Prosecco, Cachaca, Port, Pisco, Aperol, Absinthe, Ouzo.
- *Basic Taste* (9): Sour, Sweet, Cream, Bitter, Water, Mint, Egg, Salty, Spicy.
- *Glass Type* (35): Highball glass, Old-fashioned glass, Beer Glass, Beer mug, White wine glass, Shot glass, Collins glass, Collins Glass, Cocktail glass, Irish coffee cup, Highball Glass, Champagne flute, Martini Glass, Margarita/Coupette glass, Coffee mug, Pousse cafe glass, Punch Bowl, Punch bowl, Beer pilsner, Hurricane glass, Whiskey sour glass, Margarita glass, Champagne Flute, Pint glass, Mason jar, Shot Glass, Old-Fashioned glass, Parfait glass, Cocktail Glass, Coffee Mug, Pitcher, Wine Glass, Cordial glass, Brandy snifter, Copper Mug.

The cocktails in the dataset contain 303 unique ingredients, which can be alcoholic drinks, non alcoholic drinks and food (ice cream, cinnamon, orange peel, etc.). Each alcoholic drink has a corresponding *alcohol type* and the rest of ingredients have the attribute *basic taste*. Note that while *alcohol type* and *basic taste* are attributes that correspond to certain ingredients, *category* and *glass type* are features referring to the cocktails. A cocktail in the database will only belong to one *category* and it will be served in a *glass* of certain type. However, it will contain various ingredients which can belong to a wide variety of types. Alcohol types whithin a cocktail are non-exclusive, for example a certain cocktail may contain 3 different ingredients whose alcohol type is Rum.

The original dataset also contains a column that specifies the garnish used in the cocktail. However, since for most drinks this attribute has missing values, it has been discarded for not being relevant enough.

In the following Figures we present some bar charts summarizing information about the dataset. Figure 1 shows, in descendant order, the number of cocktails belonging to each of the *categories*.

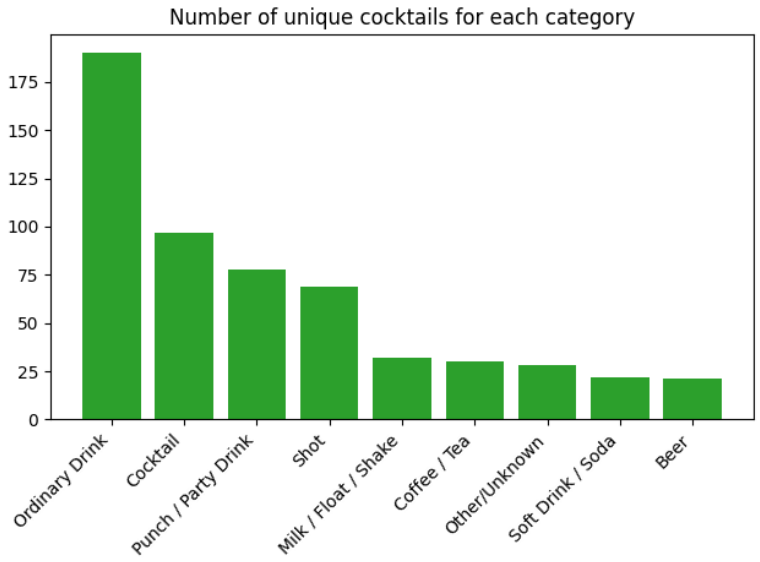

Figure 1: Unique cocktails by category

Figure 2 shows, in descendant order, the number of different ingredients that belong to each *alcohol type*. For example, we only have one type of Absinthe, Cachaça or Pisco, but we have up to 17 types or brands of Vodka and 26 different Sweet Liquors.

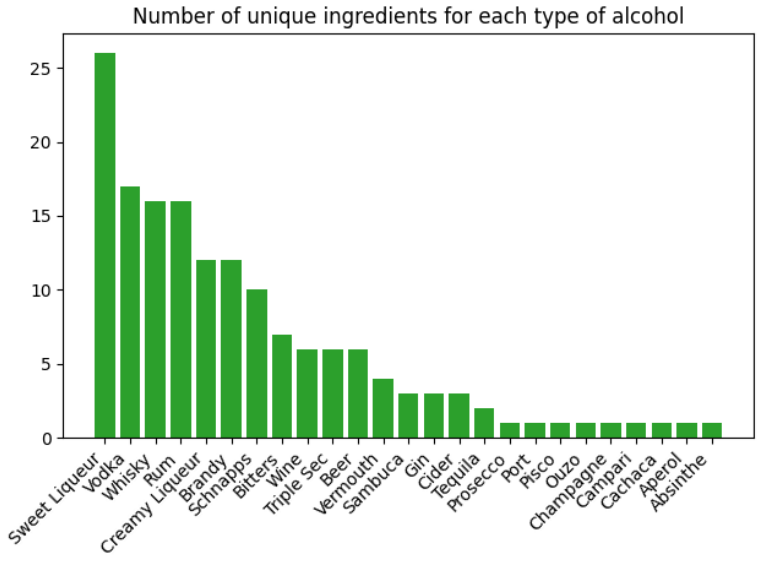

Figure 2: Unique alcohols by type

Figure 3 shows, again in descendant order, the number of different ingredients that belong to each *basic taste* class. We encounter a clearly dominant class, which are the sweet ingredients. This was to be expected since, in most of the cases when preparing a cocktail we mix up some spirit drink with some juice or soda, which are sweet drinks. Moreover there is a wide variety of these kind of ingredients (67 different sweet ingredients), while there are not so many spicy or salty ingredients that you may add to a cocktail.

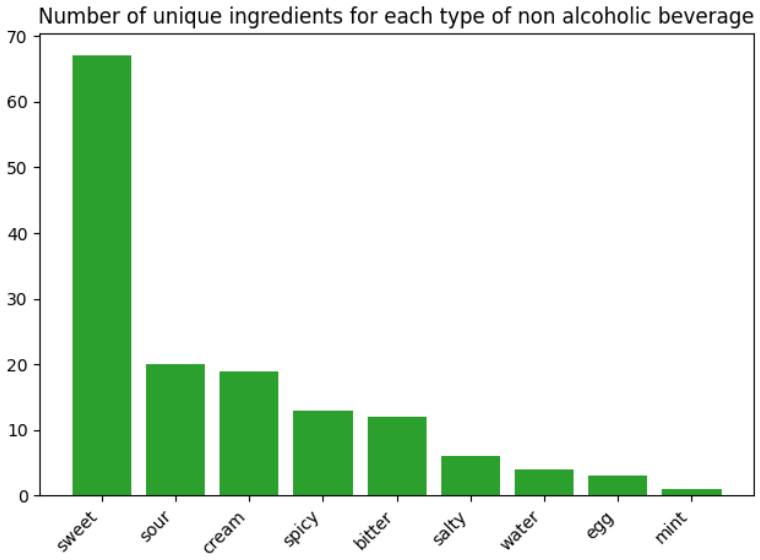

Figure 3: Unique non alcoholic ingredients by taste

Finally, Figures 4 and 5 show the number of times an ingredient appears in a recipe depending on its type. Figure 4 shows the number of times that a type of alcoholic drink appears in a recipe. In the dataset there are 473 cocktails and observing this image we can easily conclude that a high proportion of them contain more than just one alcoholic beverage, since the height

of the bars sum up to 931. For example, a cocktail (Milk/Float/Shake) such as *151 Florida Bushwacker* contains 3 different types of Rum (Light Rum, Malibu Rum and 151 Proof Rum). Note that, again the sweet flavour is the most common one, being Sweet Liqueur the most abundant type of alcoholic drink in the recipes of the dataset.

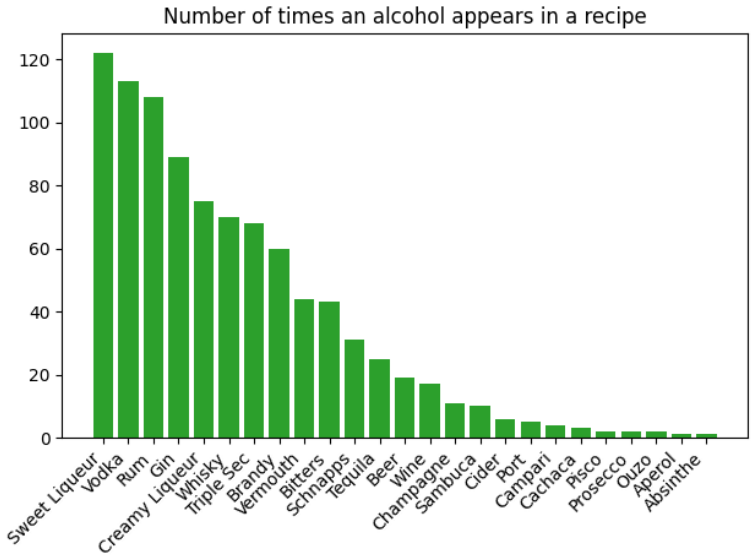

Figure 4: Popular alcohol types

Figure 5 shows how many times a non-alcoholic ingredient appears in the recipe of a cocktail depending on its basic taste. Similar that in the previous figure, there are a lot more non-alcoholic ingredients than cocktail recipes, since most of the cocktails have more than one non-alcoholic ingredient. In this case the height of the bars sum up to 850, meaning that in all our database there are exactly 850 (not unique) non-alcoholic ingredients through all the recipes. Same as before, the most popular basic taste between the 9 we have is the sweet taste, not only there are more types of sweet ingredients but they are also the most used in absolute numbers.

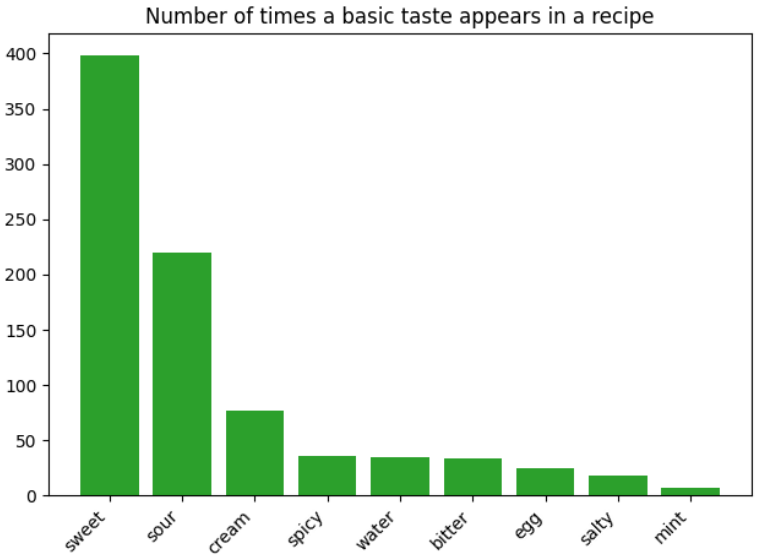

Figure 5: Popular basic tastes

2 Requirement Analysis

The main functionality of our system is to adapt recipes of cocktails in order to meet a list of constraints that is provided by the user. For that matter, the system needs a case library where the original recipes of some cocktails are stored. From this library an already existing recipe should be found and then adapted in order to meet the requirements of the user. The system will improve its performance as it generates more cases because it learns from its experience, whether the generated solutions are successful or not.

In order to accomplish all of this, the system will need the retrieval, adaptation, evaluation and learning functionalities. In addition to these four functionalities the user is going to need a way to interact with the system. Three options were developed for this task: a Graphical User Interface (GUI), an interactive menu and the possibility of using a JSON formatted file as input both by a Command Line Interface (CLI)

Regarding project execution, a Python interpreter or virtual environment will be necessary. The system was developed using Python 3.8 as programming language and PyCharm as IDE. Within the different methods of the system several Python libraries are used, and so the user will need to have them installed on the virtual environment or in the interpreter used. The libraries needed are stored in the file `requirements.txt` and their versions are: `lxml` v4.6.3, `numpy` v1.20.2, `PySide2` v5.15.2, `matplotlib` v3.4.2 and `pandas` v1.2.3.

The time response of the system was analyzed performing several automated tests in order to measure the mean time response of the system and also how the increase of the case library size affects to the time consumed by the system. As we will show in the Testing and Evaluation section, the time needed for solving one new case with out CBR system is extremely low. In general, less than a few seconds are needed for solving new cases when working with case libraries of small sizes. When the size of our case library grows, this time might be increased for concrete complicated cases but this growth is not faster than linear. Therefore the maximum time response of the system grows linearly with the size of the case library.

In the same way that the execution time grows as the case library is expanded with more new cases, the memory size needed by the system is going to grow as well. Due to the fact that the system learns from all new adapted cases and stores them, there is no maximum memory size consumed by the system and it will depend on the size of the case library.

3 Functional Architecture

In order to describe the functional architecture of the system, its different components and the relationships among them, this section is divided into: the explanation of the principal CBR scheme followed, as well as the system architecture from a software engineering superficial point of view, providing a class diagram and a sequence diagram of the principal flow.

3.1 CBR scheme

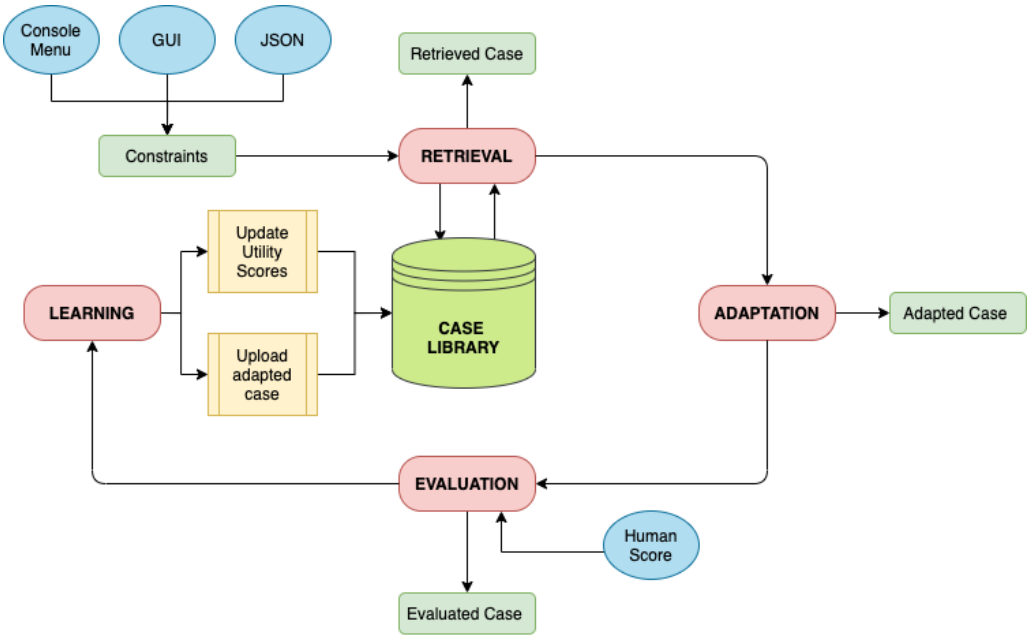

Figure 6: CBR general diagram

Our Case-Based Reasoning system parts from the typical CBR scheme and it's adapted according to our decisions as a group of experts. As it can be appreciated in the previous Figure, the process flow followed starts in the definition of constraints by the user and finishes with the update of the case library, after going through the retrieval, adaptation, evaluation and learning phases.

A peculiarity of our system is that the user can provide the input constraints in three different ways, attending to different levels of intuitiveness and interaction in order to cover different personal preferences: a Graphical User Interface (GUI), a interactive menu and a JSON formatted file, both through CLI. The first two are designed as to arrange a more thorough and wholesome experience from the point of view of the user. The last one is oriented towards more professional and experienced user profiles, who would have to write their desires in the JSON file and introduce it to the system by command line. This last method has also been useful to execute a batch of tests without requiring human interaction.

Once the constraints are defined by the user, they are parsed as a Python dictionary. This structure will be then used as input for the retrieval phase, which will return as output the retrieved case among the case library that best suits the constraints specified by the user. Using both the retrieved case and the user constraints as inputs, the adaptation step will transform the former and its solution into a new case that fulfills the user desires with a new adapted solution. This new case along with its solution will be the the output of this phase.

Next, the adapted case will be provided as input to the evaluation step, which will consist in assessing the quality of the adapted case based on the decisions of the user, who will act as a human oracle. This way, the user will give, by interacting with the corresponding interface, a grade ranging from 0.0 to 10.0 to the adapted case proposed by the system. Based on this number and a predefined evaluation threshold, the system will decide if that adapted solution

is a failure or a success. The output of this step is the adapted case with the evaluation field updated and the score provided by the human oracle.

Finally, the learning step is carried out using the output of the previous phase as arguments. On this phase, the utility measure of the retrieved case of the iteration is updated based on the total number of successes and failures derived from it along the lifetime of the CBR system. Moreover, a new utility measure for the adapted case is computed based on the score provided by the human oracle. Either the evaluation resulted in a success or a failure, the adapted case will be stored in the case library so that the system can learn from both perspectives.

3.2 System architecture

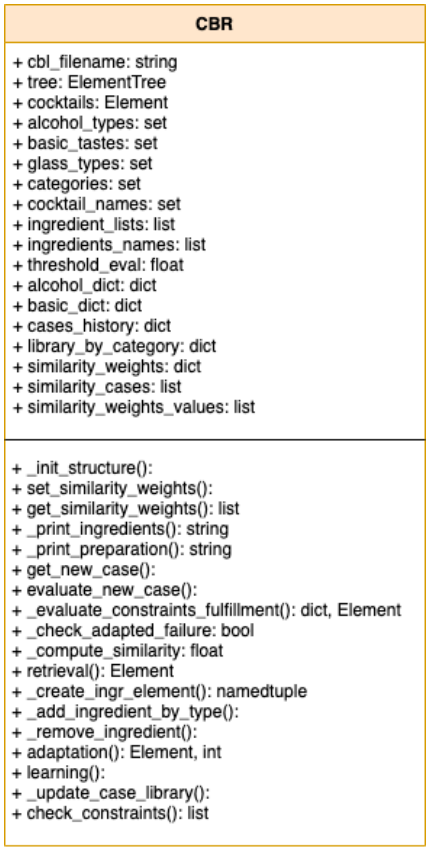

Figure 7: CBR class

Our CBR system project is composed by many files. However, the key part is the CBR class, created to wrap and manage all relevant aspects and phases of the system. Its class diagram can be visualized above in order to have a wider vision of the components involved. This class has several global variables which comprise crucial information about the dataset and different parts of the process. Among the amount of methods constructed, four of them can be specially highlighted, as they correspond to the essential phases of the Case-Based Reasoning system: retrieval, adaptation, evaluation and learning. We have considered that it is not really necessary for the goals of this report to delve into the functionality of all these methods. Instead, the execution flow of the system involving this class will be discussed in the following sequence diagram:

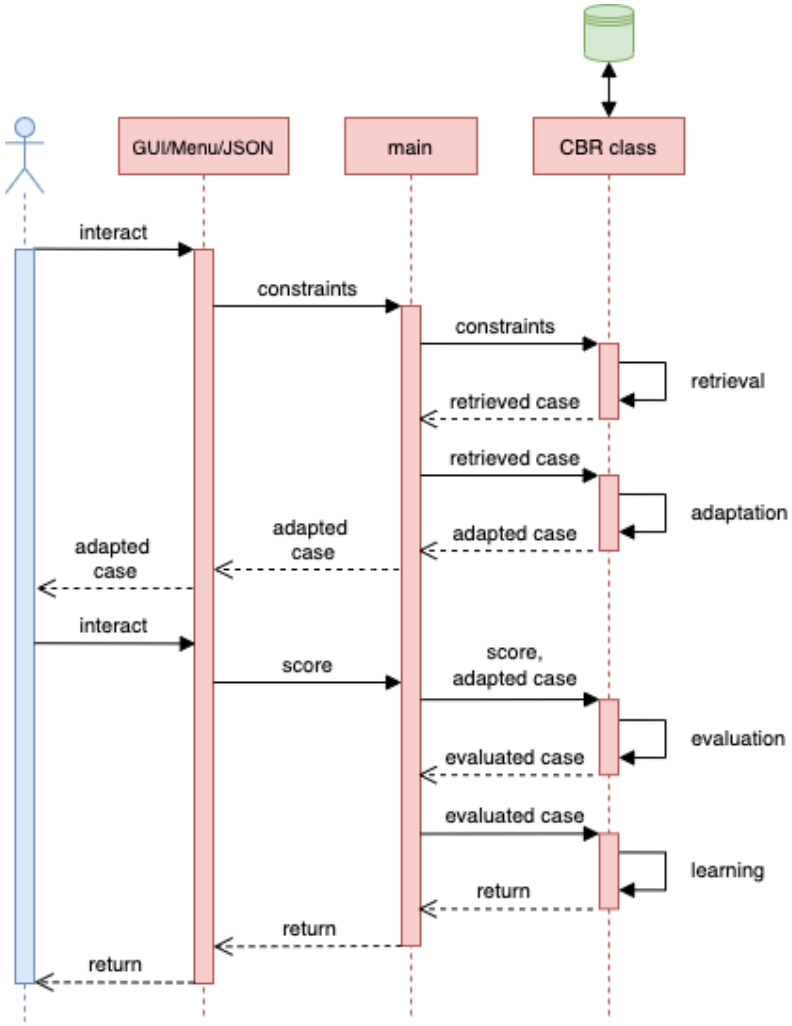

Figure 8: Sequence diagram of the principal flow

As mentioned, when the user interacts with the preferred interface form, the information collected will be parsed into a dictionary and will be passed to the main file of the project. This file will be responsible for handling the CBR's flow, instantiating the class and calling the methods relative to the principal phases, providing the different inputs for them.

In the retrieval phase, the CBR class accesses the case library in order to measure similarities and select the most similar case to the user constraints. After the adaptation phase, the adapted case will be shown to the human oracle, which will provide a grade necessary for its evaluation, as it has already been mentioned. The main file will handle this procedure as well, invoking the request to the user in the corresponding interface form. When obtained, this score will be passed along the different components until reaching the evaluation method. During the learning phase, the class handles the update of the case library. Finally, after this step, the execution flow will end and the current iteration for the CBR system will have finished.

4 Proposed CBR engine Project solution design

4.1 Description of the Case Structure and Case Library structure designed.

The Case Structure or Case Representation is the way in which the system stores and manages each single case. As commented in the application domain section, the dataset used can be found in [3] in `.csv` format. For a more suitable management of the data with respect to the necessities of the CBR system, this `.csv` file was parsed as a `.xml` file, which will be our case library. Note that, when parsing, some initial manual changes were made in the preparation steps in order to keep the cocktail recipes as generic as possible and to maintain the robustness of the library.

Once we have a clean case library in `.xml` format with a clear structure of the objects, we can say that our Cases are formed by sets of reasonable features. We will use XML Elements as the principal structures to be stored, accessed and modified through the project. An example of a case (cocktail) in our `.xml` file would be:

```
<cocktail>
  <name>Kool-Aid Shot</name>
  <category>shot</category>
  <glasstype>old-fashioned glass</glasstype>
  <ingredients>
    <ingredient id="ingr1" alc_type="" basic_taste="sour" measure="" quantity="" unit="">cranberry juice</ingredient>
    <ingredient id="ingr2" alc_type="vodka" basic_taste="" measure="1 shot" quantity="25.0" unit="ml">vodka</ingredient>
    <ingredient id="ingr3" alc_type="triple sec" basic_taste="" measure="1 shot" quantity="25.0" unit="ml">triple sec</ingredient>
    <ingredient id="ingr4" alc_type="gin" basic_taste="" measure="1 shot" quantity="25.0" unit="ml">sloe gin</ingredient>
    <ingredient id="ingr5" alc_type="sweet liqueur" basic_taste="" measure="1 shot" quantity="25.0" unit="ml">amaretto</ingredient>
  </ingredients>
  <preparation>
    <step>Pour into a large glass with ice and stir</step>
    <step>Add a little ingr1 to taste.</step>
  </preparation>
  <utility>1.0</utility>
  <derivation>Original</derivation>
  <evaluation>Success</evaluation>
</cocktail>
```

Figure 9: Example of a case

The features that our cases have are the following:

- Case identifier: in our case library the `<name>` parameter is unique for each case (cocktail).
- Description of the case: attribute-value vector formed by:
  - `<category>` of the cocktail
  - `<glasstype>` in which the cocktail is served
  - `<ingredients>` is a feature whose value is a collection of objects `<ingredient>`. Each one of them has the name of the ingredient it refers to and is an attribute-value vector with the following attributes:

```
- id            - measure
- alc_type      - quantity
- basic_taste   - unit
```

- Solution of the problem: in our case the `<preparation>` of the cocktail is considered to be the solution of a case, to be adapted for new cases. It is divided in different steps, `<step>`, note that the quality of the solutions is not the best. Sometimes all the ingredients appear in the preparation but it is not the most common situation. In the case that not all of them appear explicitly, where the steps of the preparation are that general, it might be more difficult to adapt the solution for a new case.
- Utility measure of the case: `<utility>` is a measure of the usefulness of a case for generating successful cases from it. We decided to initialize the utility measure of all the original cases to 1.0, and adapt them each time a CBR cycle is finished, depending on which case was retrieved for generating a new case and on the evaluation of the adapted case.
- Derivation of the case: `<derivation>` informs about where the case comes from. It takes the value 'original' when referring to a cocktail in the initial database and, if it is a cocktail generated by our CBR system, the name of the cocktail from which it comes from (retrieved cocktail).
- Evaluation of the solution: `<evaluation>` can take values *success* or *failure*. Initially all the cases in the Case Library will be *success*, since they are real recipes for the cocktails in the database. Once we adapt and generate a new recipe, it will only be classified as a *success* if the human expert evaluates it as a valid recipe, this is, if the numerical evaluation of the generated cocktail is larger than a fixed threshold.

With respect to the structure of our Case Library of our CBR system we decided to use a mixed organisation, using a hierarchical structure at first and flat memory structure later. We decided to divide the cases in the Case Library constructing a discriminant tree that has 9 branches: one for each *category* of cocktail. Then, the cases in each category are structured following a flat memory structure, since because of the structure of the data itself another hierarchical split is complicated and might not be adequate. This way, when a new case (user constraints) enters the system to generate a solution on the basis of our case library, the system will only check the recipes of the cocktails that belong to the same category or categories the user asks for. In the situation where the user does not provide a specific category, the CBR system searches all over the Case Library, over the 9 branches of the discriminant tree, for the most similar case. This first step of structured organisation makes the matching and retrieval steps more efficient, since only a few cases are considered for similarity assessment purposes. However, it is important to notice that this decision can lead us to sub-optimal solutions because the system might miss some optimal cases in the retrieval step.

4.2 Description of the Methods implementing each CBR cycle step

4.2.1 Retrieval

The retrieval phase is the step of the CBR cycle in which, given a new case, we look for the most similar case(s) to it in the case library. The retrieving process strongly depends on the case library organization and tries to maximize the similarity between the new case and the retrieved one(s).

Because of the structure of our case library, as a hybrid scheme between hierarchical and flat structures, the retrieval process is at the same time efficient (because the first hierarchical division) and effective when finding the best retrieved case (because of the flat memory of each branch). However, in our CBR system a non-optimal case might be found when the user asks for a particular category of cocktail and the most similar one is inside another category.

In a CBR system the retrieval process is divided into two main substeps:

- *Searching*: this step serves as an initial filter to obtain a set of promising similar cases for the next step. In order to accomplish that, it returns only the cases that correspond to the categories specified in the constraints. For that matter, it uses a global up-to-date dictionary containing all cases of the case library grouped by category, so that it's not necessary to iterate over the whole case library every time that the retrieval phase is called. Afterwards, another filter is passed so that we are not selecting failure cases nor cases that derived into failures. Note that this last step of learning from failures is not present when initializing the CBR system, as there will be no failures to learn from. We will explain more about it in the Learning step.
- *Selecting*: among the best cases found in the previous searching step, the relevance (i.e. the similarity $s(x, y)$) of each case with respect to the user constraints is computed. The best, most similar one is chosen to be the retrieved case. If there is a tie, a random selection is carried out.

The system computes the similarity between the new case (given by the user constraints) and the case library cases using domain dependent similarities defined by us. For computing this similarity between cases, we assigned each attribute of a case a determined importance value or weight, $\omega_k \in [0, 1]$, which are:

- `glass_type`: the weight of the constraints associated with the glass type is $\omega_{glass} = 0.4$ since it is not that relevant to have the right glass when asking for a cocktail.
- `ingredients`: when the user asks for a concrete ingredient, it is essential that the cocktail the system offers them contains this ingredient. Therefore, the weight of the constraints associated with concrete ingredients is $\omega_{ingr} = 1.0$.
- `alc_type` and `basic_taste`: the weight of the constraints associated to the desired types of ingredients is $\omega_{type} = 0.8$ since it is not as relevant as asking for a concrete ingredient but, at the same time is quite important.
- `exc_ingredient`, `exc_alc_type` and `exc_basic_taste`: when the user requests for an ingredient or a type of ingredient that they don't not want in the cocktail, we considered extremely important that these ingredients were not in the offered cocktail. Therefore, the weight of these constraints is $\omega_{excluding} = 1.0$.

After defining the importance of each comparison that the system needs to perform between a new case and the cases in the case library, we need to define a similarity measure that captures the essence of these constraints.

Among the 7 possible schemes of constraints that our system can handle, there are five of them that are considered to be binary:

- A cocktail may be served (*similar*) or not (*not similar*) in one of the desired glass types requested by the user.
- A cocktail may contain (*similar*) or not (*not similar*) each alcohol type requested by the user.
- A cocktail may contain (*similar*) or not (*not similar*) each basic taste requested by the user.
- A cocktail may contain (*not similar*) or not (*similar*) each alcohol type the user does not want in their cocktail.
- A cocktail may contain (*not similar*) or not (*similar*) each basic taste the user does not want in their cocktail.

Apart from that, the constraints that refer to concrete ingredients (that need to be included in or excluded of the resulting cocktail) are considered to be deeper than the others. When searching for a concrete ingredient in a case of the case library we may not find it, but the case may contain another ingredient of the same type that the requested one.

If the requested ingredient is in the case we are comparing, the constraint is completely fulfilled and the cases are *similar*. If the requested included ingredient is an alcoholic (analogously non-alcoholic) ingredient that is not in the case we are comparing, we look for ingredients with its same alcoholic type (analogously basic taste). If there is one of those 'similar' ingredients, the system determines that the constraint is met at some level, the cases are *somewhat similar*. If none of this cases occur, the system determines that the constraint is not met and the cases are *not similar*.

Something similar happens with the constraint referring to requested excluded ingredients: if the excluded ingredient is in the case we are comparing then the constraint is not met and the cases are *dissimilar*; if the excluded ingredient is not in the case but some ingredient of the same type is, the cases are *somewhat dissimilar* and, in another case, the cases are *not dissimilar*.

Therefore, the similarity $sim_k(x, y)$ between cases with respect to different constraints can take the following values:

- Binary constraints: $sim_k(x, y) = \begin{cases} 0, & \text{if constraint is not met} \\ 1, & \text{if constraint is met} \end{cases}$
- Included ingredients constraint: $sim_k(x, y) = \begin{cases} 0, & \text{if cases are not similar} \\ 0.6, & \text{if cases are somewhat similar} \\ 1, & \text{if cases are similar} \end{cases}$
- Excluded ingredients constraint: $sim_k(x, y) = \begin{cases} 0, & \text{if cases are not dissimilar} \\ -0.6, & \text{if cases are somewhat dissimilar} \\ -1, & \text{if cases are dissimilar} \end{cases}$

Finally, with all the weights of each constraint and the possible values of (dis)similarity of each constraint defined, we can compute the normalized similarity between two cases as follows:

$$s(x,y) = \frac{\sum_{k=1}^{K} \omega_k \; sim_k(x,y)}{\sum_{k=1}^{K} \omega_k}$$

where $s(x,y) \in [-1,1]$ is the similarity measure between the cases x and y; ω_k is the relevance of each comparison (comparing that the type of glass matches is less relevant that an alcoholic ingredient matches); K is the number of constraints checked between cases and $sim_k(x,y)$ is the similarity metric described above, different for each type of constraint.

As a last step in the similarity computation, the normalized similarity of each case is multiplied by its utility measure. This way, even if a case is very similar to the constraints defined, its possibilities of being selected as the retrieved case decrease with a low utility score. This way, we put in practice the real idea behind utility measures, as we are forgetting not useful cases.

4.2.2 Adaptation

After retrieving the best (partial) matching case from the case library, the old solution (retrieved solution) needs to be adapted to fit the new case, this is, to fulfill the constraints given by the user. This reusing process takes place during the solution formulation, inside the adaptation step.

We made use of various **structural adaptation methods**, which apply the adaptation directly over the solution of the best retrieved case. Structural adaptation methods that our system uses can be divided into:

- **Transformation methods**: these methods use common sense transformations rules over the solution, such as adding or deleting components from the retrieved solution to generate a valid solution for the new case.

  - When the system receives a negative constraint that refers to an ingredient or type of ingredient that the user does not want in their cocktail, it proceeds in the following way. When it encounters an ingredient in the retrieved solution that is inside the *exc_ ingredients* list, it directly **deletes** it from the recipe. To delete this component the system removes the component from the ingredients list and, if there is any step where this ingredient appears, the system removes the step from the recipe or removes only that excluded ingredient in case there are more ingredients in it. The same process occurs when dealing with *exc_ alc_ type* and *exc_ basic_ taste* constraints. If any ingredient from an excluded alcohol type or basic taste is in the retrieved recipe, the system simply removes it along with the steps corresponding to it.

  - When the system receives a positive constraint that refers to a type of ingredients that the user explicitly desires in their cocktail, it proceeds in the following way. It first

checks whether the desired type (in *alc_ type* or *basic_ taste* lists) is already present in the recipe. In that case, it validates that the constraint is fulfilled and continues by examining the following constraint available. If the desired type of ingredient is not in the retrieved recipe, the system searches over the database of ingredients of that type and randomly choose one that is not in the *exc_ ingredients* list. The chosen ingredient is **added** to the recipe, along with a step of the preparation referring to this addition.

- **Substitution methods**: these methods alter the solution of the retrieved case by searching for appropriate components computed from the original components in the retrieved solution.
  - When receiving a positive constraint about a concrete ingredient that the user explicitly wants in their cocktail (in the *ingredients* list), the system checks if that ingredient is not already in the recipe. If the constraint is not yet satisfied, we encounter two different cases depending on the ingredients already present in the recipe: the system may directly **add** the ingredient or it may **substitute** another ingredient by the desired one.
    i. If the desired ingredient is an alcoholic beverage, (`alc_type` $\neq$ `""`), and there is no other alcoholic beverage in the recipe that has the same *alc_ type*, the system directly **adds** the desired ingredient to the recipe along with its corresponding step in the preparation process. Same happens when the ingredient present in the positive constraint is a non-alcoholic ingredient. If `basic_taste` $\neq$ `""` and there is no other ingredient in the recipe that has the same *basic_ taste*, the system directly **adds** the desired ingredient to the recipe along with its corresponding step in the preparation process.

    ii. If the desired ingredient is an alcoholic beverage, (`alc_type` $\neq$ `""`), and the system finds other alcoholic beverage in the recipe that has the same *alc_ type* (which we will call 'similar' ingredient) it will need to perform an extra step. The system needs to check whether that 'similar' ingredient is in the list of desired *ingredients*. In that case, if the 'similar' ingredient was explicitly requested by the user, the system will directly **add** the desired one. If the 'similar' ingredient is not present in the *ingredients* constraints, the system will **substitute** it by the desired one.
    Note that, inside the substitution process, the step of checking whether the currently desired ingredient is one of the explicitly requested ingredients is vital. If we do not validate this property the system may substitute one desired ingredient by another if, for example, the user asks for a cocktail with two different whiskies: bourbon and scotch. Once the system finds the first constraint, it will add bourbon in case it is not in the recipe. When checking the second constraint corresponding to the scotch whisky, if the scotch whisky is not in the recipe, the system would substitute bourbon by scotch. This way, the system would damage a constraint when trying to meet another.
    This process of adding or substituting is performed analogously for the non-alcoholic ingredients and this substitution method is called **abstraction and respecialization**.

4.2.3 Evaluation

The evaluation step is carried out after adapting the solution, as a third step. The purpose of this step is to qualify and determine the performance or the quality of the proposed solution. Note that only "derivated" cases are evaluated because "orignal" cases, those that come from the original dataset or knowledge base, are considered to be successes that need no further evaluation.

Between the adaptation and the evaluation step several constraints are checked. In order to not assess cases with a high probability of being a failure, the similarity of the adapted case with respect to the cases in the library is measured. In case that the similarity with a case that has been a failure is higher than a threshold, the query is directly assigned as a failure and it passes to the learning phase and a new retrieval phase starts.

In addition, the constraints provided by the user are checked and, if any of them is not fulfilled by the adapted case a new retrieval-adaptation phase is started.

There are different available methods for performing an evaluation. For our system we decided to use the evaluation by a human expert and also to perform the solution in the real world. The option of running a simulation in order to evaluate the proposed solution was discarded.

Before starting the evaluation step our system first shows to the user the ingredients needed on the preparation and the preparation steps (adapted solution). Then it asks to the user to provide a qualification between 0.0 and 10.0 for the new recipe. After setting an input score to the cocktail the evaluation step starts. The evaluation step takes into account the score provided by the user and decides if the case evaluation is "Success" or if it is "Failure" comparing the score with a threshold. In our case, this threshold was set as 8.0 as we wanted to watch the performance of the system from a point of view of an exigent user, but it could be manually modified as desired. Once the evaluation of the new case along with its adapted solution is decided, this evaluation parameter is added to the cocktail and the evaluated cocktail is returned and sent to the learning phase.

4.2.4 Learning

After performing the evaluation of the case and as the last phase of the system, the learning step is carried out. On the learning step the system uses the new cases in order to learn from them and improve its performance, facilitating the possibility of producing better solutions in the future.

There are several ways in which a CBR system can learn. In this case, it was decided to emphasize a learning by experience. This way, the CBR system gains knowledge after each cycle of execution, learning from its own behavior and results. Specifically, the system can learn from successes or from failures, as both approaches have been covered in our methodology. In addition, the learning phase is responsible for managing the utility measures, which are very important metrics whose purpose is "forgetting" the useless cases. In order to accomplish that aim, they condition the similarity between cases in the retrieval phase.

First of all, in this phase the utility measure of the retrieved case is updated by following the next formula:

$$Utility = \frac{\#Successes - \#Failures + 1}{2}$$

For that matter, a historical of the count of successes and failures of each case in the case library is preserved, consulted and updated through the whole lifetime of the system. Note that each success or failure added to the historical of each case will be weighted by the score provided by the human expert, so that the utility measure of each retrieved case represents the real usefulness of that case for the matter.

Afterwards, the utility measure of the adapted and evaluated case will be also updated by multiplying by 0.1 the human oracle score provided as well. This way, this metric represents the level of satisfaction with the case. It is relevant to highlight that, if the utility measure of a retrieved case (which is always going to be successful) is decreased to 0, it is set as "Failure" and will not be considered in further retrievals.

In our system, both learning from failure and learning from success have been implemented. So all adapted and evaluated cases, either success or failure, are stored in our case library with its corresponding utility measure. The successes will be useful for solving future situations similar to this one, whereas the failures will prevent making the same or similar mistakes in future iterations.

5 Testing and evaluation of the CBR system

Different tests and evaluations of the CBR system were performed in order to assess that the system execution occurs with no errors, to measure execution times and how the behaviour of the system evolves when the case library grows. Some testing was done during the development of the system in order to ensure that the system was working and to test the different changes applied to it. The majority of these tests (the ones performed during the development of the project) were done through the GUI as a manual testing. After finishing the project a set of final tests were also performed in an automated fashion using JSON files as inputs.

5.1 Manual Testing

On this section several tests performed with the GUI are going to be showed. These tests are presented as to analyze in a more detailed way the behavior of the system, how it adapts the recipes, if there are no changes needed and what are the returns of the system.

All the user constraints can be specified using the GUI. Among these constraints the user can select the desired category/categories of the drink, ingredients, alcohol types, basic tastes or glass types and also the negative constraints (for example which ingredients you wish to not have on your cocktail).

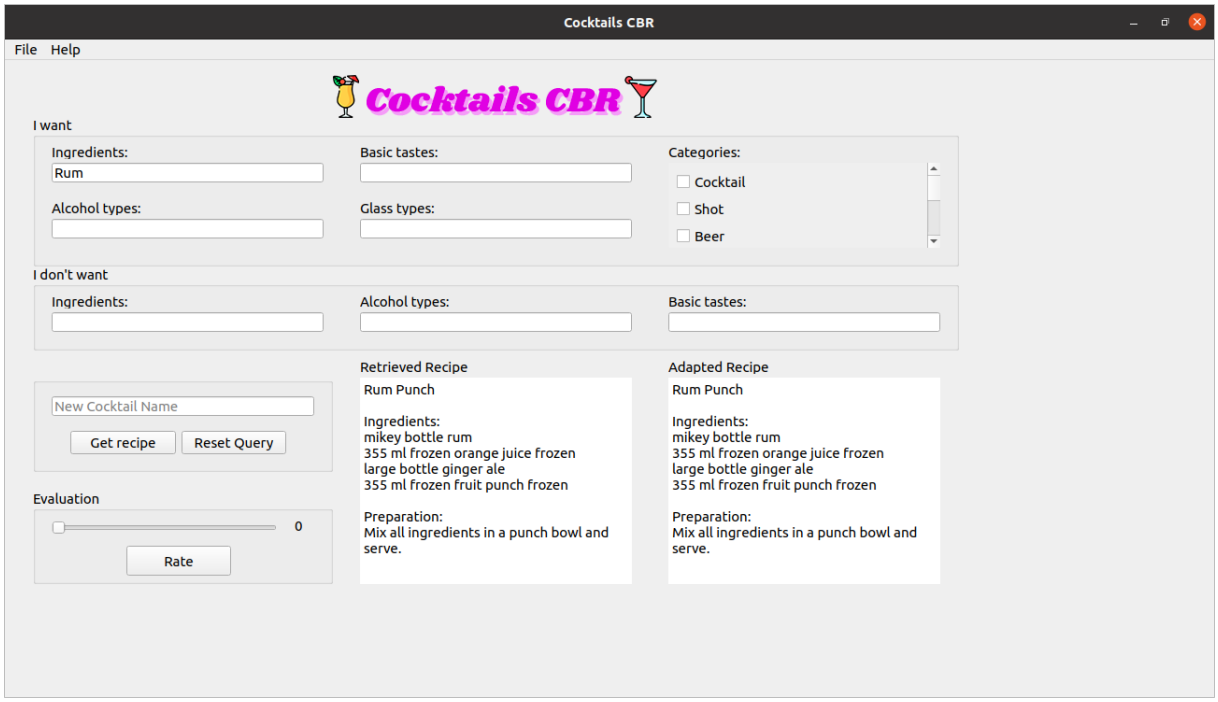

Figure 10: GUI Interface, Rum ingredient as constraint

```
$ [CBR] Retrieved case: Rum Punch
$ [CBR] Similarity between constraints and retrieved case: 1.0
$ [CBR] Ingredient rum is already in the recipe, no changes needed
```

The last three lines are printed on the terminal (many others are printed too, but we will highlight these lines on this section).

The constraints specified on this case are very simple and matches completely an already existing cocktail. As it can be appreciated in the debug message printed on the terminal, a similarity of 1.0 is obtained with another cocktail meaning that no adaptation is needed to fulfill all the constraints. In the GUI it can be seen that the original recipe and the adapted one are the same recipe.

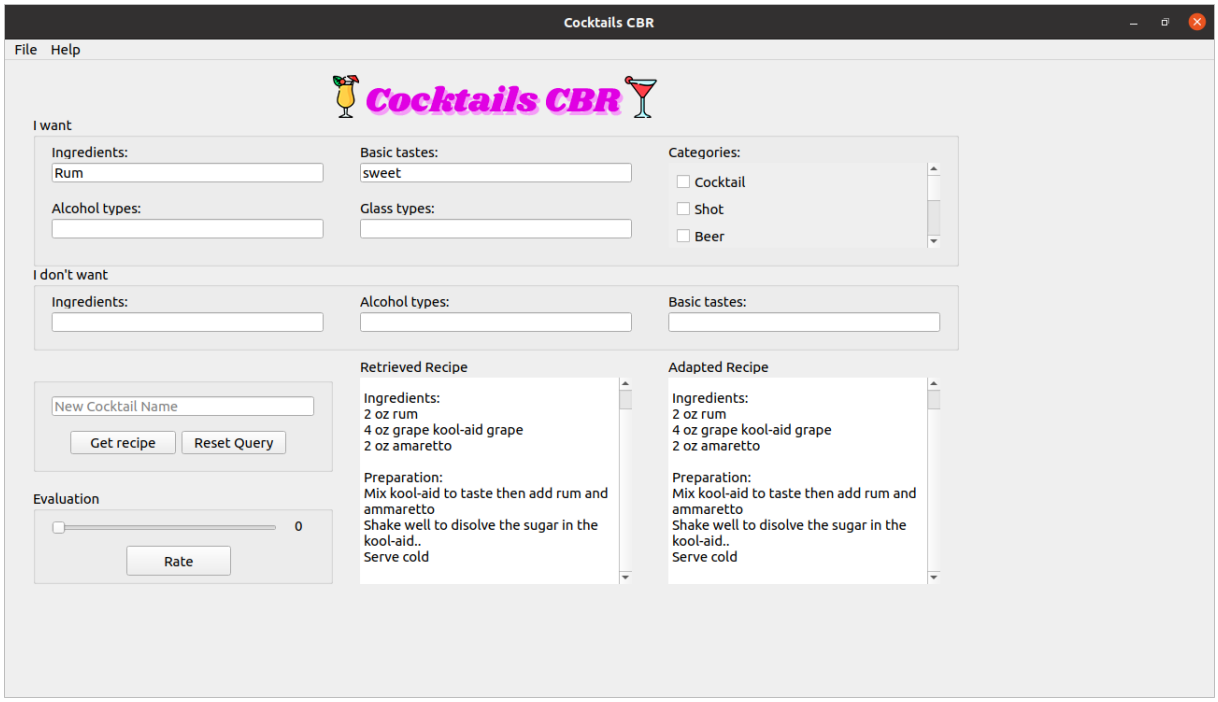

Figure 11: GUI Interface, Rum ingredient and Sweet taste as constraints

```
$ [CBR] Retrieved case: Zippy s Revenge
$ [CBR] Similarity between constraints and retrieved case: 1.0
$ [CBR] Ingredient rum is already in the recipe, no changes needed
```

The same happens as in the previous case. The constraints are easy to fulfill, no adaptation is needed and a cocktail that already exists on the case library is returned as the most similar case.

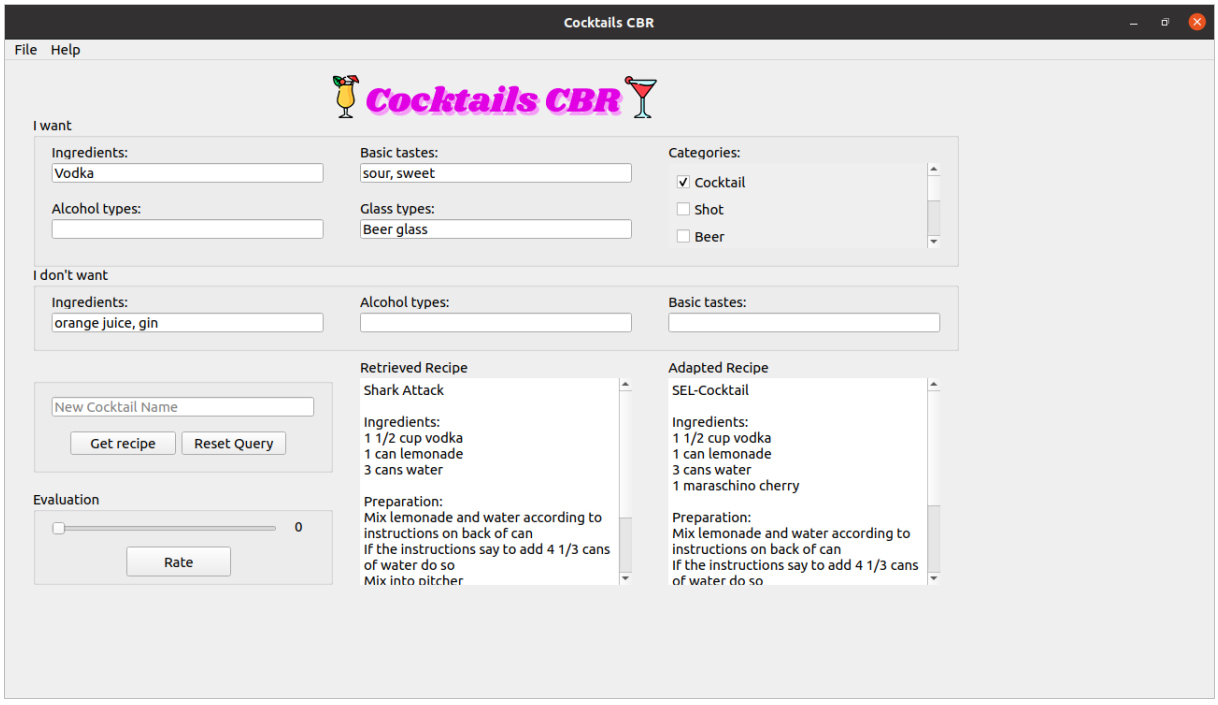

Figure 12: GUI Interface, recipe that needs adaptation

When more constraints are added in the elaboration of this recipe, the terminal output for the case is the following one:

```
$ [CBR] Retrieved case: Shark Attack
$ [CBR] Similarity between constraints and retrieved case: 0.391304347826087
$ [CBR] I served your cocktail in a beer glass to fulfill your constraint
$ [CBR] I added maraschino cherry to the recipe to fulfill your sweet positive
        constraint
$ [CBR] Ingredient vodka is already in the recipe, no changes needed
```

As it can be seen, several changes are performed on the original recipe in order to fulfill all the constraints. The similarity is smaller than 1.0 as the original recipe does not fulfill the constraints. On the terminal the user can see some debug messages that explain the changes applied to the recipe and in the GUI the *Adapted Recipe* text box shows the new recipe with the added, removed or replaced ingredients as well as the adapted preparation steps.

Moreover, by using a slider in the GUI, the user can set a score for the cocktail. By pressing *Rate*, the evaluation phase will start with that score.

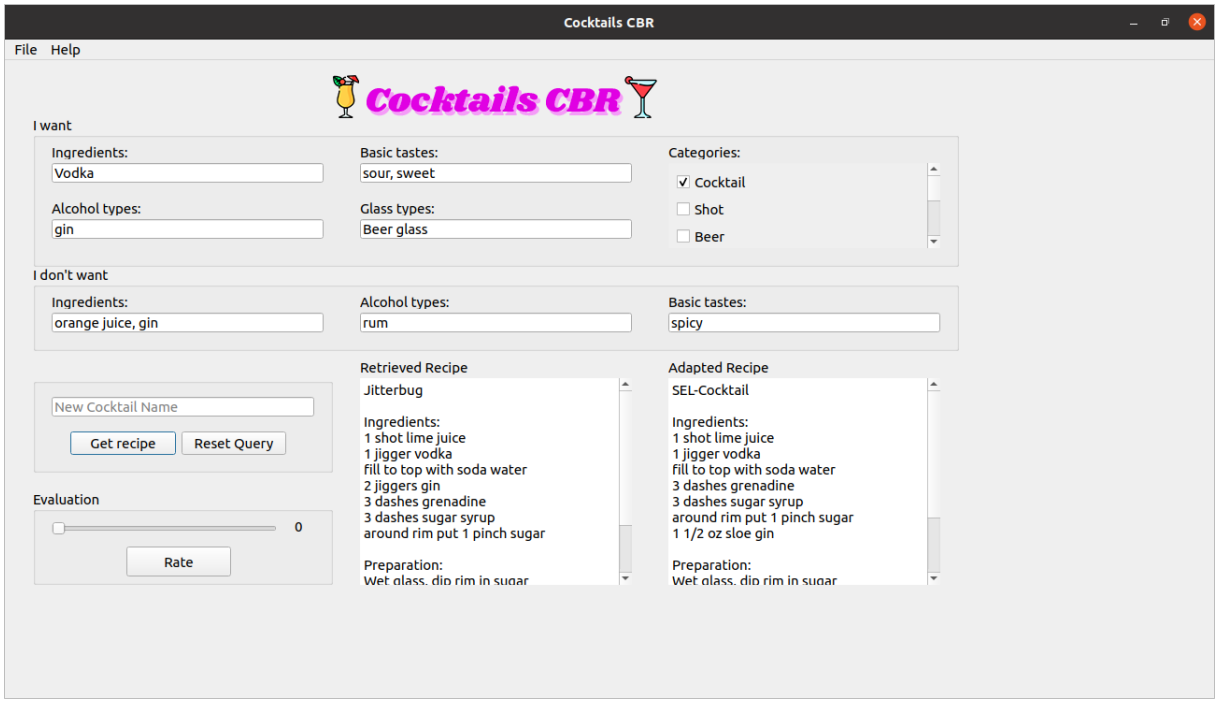

Figure 13: GUI Interface, recipe that needs adaptation

```
$ [CBR] Retrieved case: Jitterbug
$ [CBR] Similarity between constraints and retrieved case: 0.24324324324324326
$ [CBR] I served your cocktail in a beer glass to fulfill your constraint
$ [CBR] I removed gin from the recipe to fulfill your negative constraint
$ [CBR] I added sloe gin to the recipe to fulfill your gin positive constraint
$ [CBR] Ingredient vodka is already in the recipe, no changes needed
```

As it is shown on this last step more constraints can be specified and the system will adapt the cocktail to the needs of the user. The similarity case drops down as more constraints are specified because the original recipes are going to be less similar when growing the number of

constraints and more changes are going to be needed. These changes are again listed by CLI and shown on the GUI.

On this last case two constraints should be highlighted. In the wanted alcohol types "gin" alcohol type is specified; on the other hand, "gin" ingredient is also indicated as an ingredient that is not wanted on the cocktail. Two opposite constraints can not be defined as an input, but in this case these two constraints are not opposite, but there is an ingredient named as gin and there is an alcohol type named as gin. More ingredients than the "gin" ingredient are of the "gin" alcohol type; for example, on the adapted case the ingredient "sloe gin" is added in order to fulfill the alcohol type constraint. Then, the ingredient "gin" is removed from the recipe in order to fulfill the constraint specified.

These tests aren't valid in order to assess the time complexity of the system or other technical aspects but they are a tool in order to check that the system is performing well. Also they return results in a very visual way and errors can be seen easily.

5.2 Automatic Testing

In order to automatically test the robustness and performance of our system, a set of batch tests that use a JSON file as input have been developed. First, we created JSON files with 100, 500 and 1000 different constraints combinations, generated in order to test the robustness of our system to different random but plausible inputs. This was done in order to search possible errors or situations that our system could not handle and for evaluating the impact on time that the addition of new cases to the library has. In order to generate the JSON files with the constraints we wrote a python script called `preparing_test.py` and for performing these tests we used the `performing_tests.py` script.

To generate the JSON files with random constraints we took into account various details:

- Since the number of possible constraints (ignoring the name) are 7, we generated 7 random integers between 0 and 3 to establish the number of constraints of each type we create. For example, if this generated number is 0 for alcohol types, there will not be any constraint referring to desired alcohol type and, if the number generated for excluded ingredients is 2, there will be 2 concrete ingredients that can not be in the obtained cocktail.
- For the positive constraints, we simply searched in the database for random categories, glass types, alcohol types, basic tastes and ingredients depending of the random number of desired constraints we obtained with the previously explained process.
- Finally, for the negative constraints, we generated the appropriate number of random elements from each type but removed from the constraints the ones which generated contradictions with the positive ones. For example: if a positive constraint states that there must be *Bourbon* in the solution to the new case, we would remove *Whisky* if it appears as a negative constraint of alcohol type.

It must be mentioned that, in order to be able to automatize the testing process, without needing the help of a human expert which evaluates each solution found by the CBR system,

we agreed that the score given for all the generated solutions will be 8.0, which will lead to a successful query in all the cases.

To measure the time needed by the CBR system to generate a solution for a new case, we measured separately the time needed for performing the Retrieval and Adaptation steps, and the Evaluation and Learning steps for each new case or query entering the system. After initializing the case library, each time a new case is solved and learnt, the size of the case library gets larger and we expect that the time needed to solve a new case gets also larger. Therefore the goal of these experiments is not only to check that this time gets bigger as the case library grows, but also to observe the evolution of the time needed to solve new queries.

In the following images we can observe the times needed for the CBR system to manage each part of the process for the different sizes of JSON files:

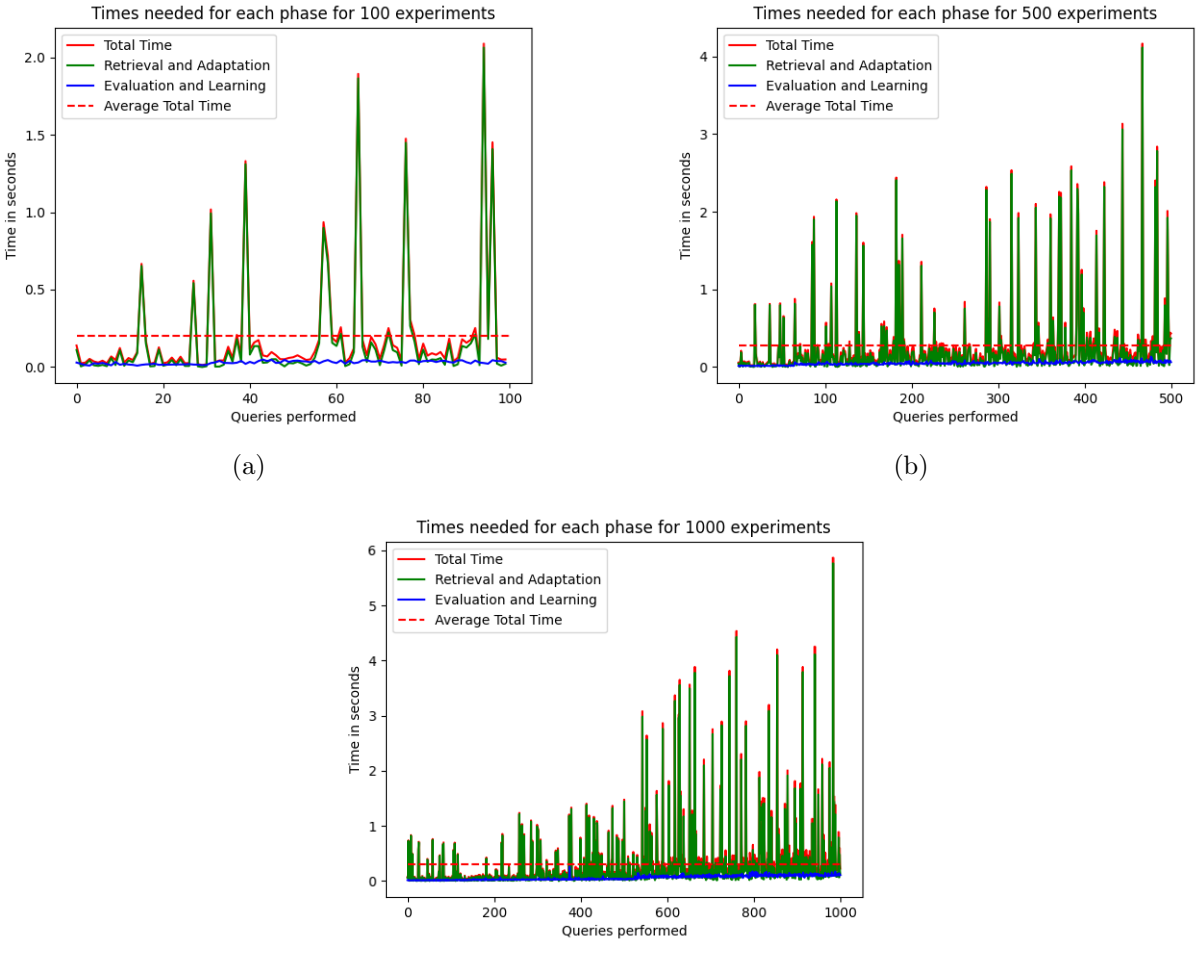

Figure 14: Different times needed for solving new cases from an initialized case library

It is important to notice that the three plots have different scales in the vertical axis representing the time in seconds needed for each process. As the number of queries in the JSON file grows, the time needed to solve new queries after having learnt from the previous ones also grows. This is to be expected since, in the retrieval and adaptation steps the system needs to

traverse the category subsets of the case library in order to find the most similar case. The more cases from which to measure similarities, the more time spent in this step. This fact can be clearly seen in the images, where we can observe that the Retrieval and Adaptation time is almost equal to the total time needed for each case.

We can clearly observe some peaks in the curves of times. This peaks are no more than "difficult cases" that need to be adapted after the retrieval step. While simple cases (low number of constraints) may lead to straightforward solutions, when the number of constraints is higher the system needs more time to compute the similarity between this new case and the cases in the case library. Moreover, if the retrieved cocktail needs to be adapted, the system will also need time for performing that step according to the constraints.

However, we plotted in the three cases the average total time needed to solve a new case and we can state that, although the maximum time needed for solving *difficult* cases grows linearly with the size of the case library, the average total time does not grow so much, being in our 3 experiments lower than 0.31 seconds.

Below we include a table with the average times needed for each of the parts of the CBR process corresponding to the automatic testing with the 3 different JSON files of constraints:

	100 Queries	**500 Queries**	**1000 Queries**
Retrieval and Adaptation	0.1752172	0.2342375	0.2436897
Evaluation and Learning	0.0290111	0.0480779	0.0583207
Total	0.2042283	0.2823154	0.3020104
% R.A. / Total	85.79 %	82.97 %	80.96 %

Table 1: Average times in seconds

In the last row of this table we show the proportion of the total time needed to overcome Retrieval and Adaptation steps, which decreases as the number of queries increases. This can be explained by looking at how the Retrieval-Adaptation times evolve compared to the Evaluation-Learning. The latter increases faster than the former. This means that having more cases in the library has a higher impact on the execution time of the evaluation and learning phases, probably because in order to update the utility score of a case an unstructured search is performed.

6 Discussion of results

Many tests have been performed along the development of the CBR system. By doing so it has been possible to catch bugs and fix them as fast as possible. After performing the final tests that are presented in the previous section, we can conclude that we have effectively developed a system that can retrieve an adapt a cocktail recipe to match the constraints given by a user, as well as to learn from its experiences.

We have seen that our system can correctly deal with the addition, replacement and deletion of ingredients to fulfill the set of required constraints. However, the adaptation of the preparation has not given optimal results. As previously mentioned, the preparation steps of all the recipes are not standard. In some cases a step would say "mix all the ingredients" instead of mentioning them one by one. In others, a slightly different name would appear in the preparation step compared to the ingredients list (for example using *cinnamon* in the ingredients and *cinnamon stick* in the preparation step). These variations can result in suboptimal results and would need to be addressed in a more thorough data preprocessing step.

Regarding the performance of the system, we have seen that using a hybrid library architecture has given us some benefits and as a result the mean retrieval time has not dramatically increased with the size of the library. Given that this system needs the interaction of the user to give an evaluation to the obtained recipes, the current time performance is sufficient for a real life scenario. However, it must be mentioned that as observed in the automatic tests performed, the system can take up to 6s to present the user with a solution, which is not too problematic for this kind of application but can negatively affect the user experience.

Overall, the implemented system has fulfilled the initial requirements and the development of this CBR engine has proved to be a useful way to gain further knowledge about Cased-Based Reasoning systems.

7 Future work

Some ideas emerged during the development of the project to improve the CBR system, they were left as future work. Some of those ideas are:

- Using Wu-Palmer similarity, or any other Natural Language metric, to measure the differences between types of alcohol or basic tastes. For example, Vodka and Gin are quite similar since they are white distilled drinks with such a high percentage of alcohol, however they are not that similar to some sweet liquor which is more soft in flavour and alcohol level. This way, we could take into account those metrics when measuring the similarities between two cases and even in the adaptation step to substitute one drink by other with similar characteristics eventhough they have different alcohol type or basic taste.
- In the retrieval step our CBR system only returns to the main flow of the system the best case found. A future approach would be to store a list of cases ordered by similarity. This way, if we find out that the best retrieved case is really similar to a failure case in the case library, we can learn from the failures by avoiding using that case again in the adaptation step and returning the second best similar.
- For performing the automatic testing of the system we created various JSON files which randomly generated constraints. However, since our system is based on human expert knowledge we could not automatize that evaluation step and decided to give a mark of 8.0 to all the solutions generated by the system. An approach to try in the future would be to generate a random number between 0.0 and 10.0 for each new case end evaluate it this way. However, this random evaluation may lead out CBR system to a wrong learning phase, eventually not being able to generate *good* solutions.

- Regarding the evaluation step, sometimes the fact that a human expert must intervene obstructs the CBR cycle and is not as optimal as it could be. In order to address this issue, some rules based on logical and domain knowledge should be modelled so that the evaluation of adapted cases could be done automatically.

References

[1] Janet Kolodner. *Case-based reasoning*. Morgan Kaufmann, 2014.

[2] ICCBR. Computer Cooking Contest 2017 Challenges. https://ccc2017.wordpress.com/rules/, 05 2017.

[3] Svetlana Gruzdeva. Cocktails data. https://www.kaggle.com/svetlanagruzdeva/cocktails-data, 12 2020.

[4] Miquel Sànchez-Marrè. Principles of case-based reasoning. https://www.cs.upc.edu/~miquel/sel/CBR-intro.pdf, 2021.

[5] Miquel Sànchez-Marrè. Part I – Fundamentals of CBR Academic Demonstrators. https://www.cs.upc.edu/~miquel/sel/SEL-CBR-1-Fund&AcadDem-MAI-2021.pdf, 2021.

[6] Miquel Sànchez-Marrè. Part II – CBR system components. https://www.cs.upc.edu/~miquel/sel/SEL-CBR-2-Components-MAI-2021.pdf, 2021.

[7] Miquel Sànchez-Marrè. Part III –CBR Application. https://www.cs.upc.edu/~miquel/sel/SEL-CBR-3-CBR-Application-MAI-2021.pdf, 2021.

[8] Miquel Sànchez-Marrè. Part IV – CBR Development Problems. https://www.cs.upc.edu/~miquel/sel/SEL-CBR-4-CBR-DevelopmentProblems-MAI-2021.pdf, 2021.

[9] Miquel Sànchez-Marrè. Part VII – Evaluation. https://www.cs.upc.edu/~miquel/sel/SEL-CBR-7-CBR-Evaluation-MAI-2021.pdf, 2021.

A E-Portfolio

The initial distribution of the tasks was made in a general way, assigning one of the principal phases to each member of the group and the case library parse and structure to the person with the weakest phase assigned.

1. Dani - Retrieval
2. Estela - Adaptation
3. Fernando - Evaluation (+ case library)
4. Xavier - Learning

However, through the implementation process different subparts of the original ones became individual tasks that had to be addressed. Additionally, more and more subtasks started to be conformed as the project advanced and new one such as the CLI and the GUI were created. As a result, we all collaborated as a team and helped in many different chores. For example, Fernando who had the simplest part of the CBR scheme, was participating as equal with Estela in the adaptation step. And Xavier, who was initially in charge of the learning step developed himself all the GUI interface while collaborating when needed was help in any other task. The participation of each member in each task, as well as an estimation of the hours dedicated to the project by each of us can be observed in the following table:

	Estela	**Fernando**	**Daniel**	**Xavier**
Case Library and Case Structure	X		X	X
Retrieval			X	
Similarity Function	X		X	
Adaptation	X	X		
Evaluation		X		X
Learning			X	
GUI				X
CLI/JSON	X	X		X
Report and Presentation	X	X	X	X
Hours Estimated	70	50	55	60

Table 2: Task assignation and hours dedicated by each member

In order to collaborate efficiently, we used GitHub for the source control of the software and also to manage tasks. In addition to that, every evening since we started working on the project we scheduled a follow-up meeting to check the status of the project, discuss the the blockers that each member could have encountered and to discuss the next steps.