An Interactive CBR System (AICS)

Betanzos, Miguel
gaurwraith@gmail.com

Blanco, Guillem
gblanco92@gmail.com

Moreno, Jonathan
jmoreno@lsi.upc.edu

1 Introduction

This CBR system aims to be an interactive system that the user can manage step by step. This project is open-ended as it exposes the infrastructure and few specific features are provided. We do this in order to be as general as possible, this way the user can implement its own method to deal with the peculiarities of their problems. While many options can be configured via the input console, the inner structure of our CBR is intended to facilitate the addition of features that can further expand the system.

2 A guided example

Let's go through the basic functionality of our CBR system. In order to do so we will use the Iris data set as guided example.

To start AICS the user needs to execute the CBR.py script using a python interpreter:

```
>> ipython ./CBR.py
```

This executes an interactive shell where the user can interact with the system:

```
CBR tool

(Cmd)
```

Typing `help` we get a list of the available commands

```
(Cmd) help

Documented commands (type help <topic>)
========================================
executeCBR  hist             printCurrentCase  setRetrieveParameter
exit        loadCasesFromCsv printMemory       setWeights
help        newCase          setMemoryModel
```

For more information about a command just type: `help <commandName>`. Let's begin loading the Iris data set from a CSV file into the system. To load a set of cases from a CSV file use the `loadCasesFromCsv` command:

```
(Cmd) loadCasesFromCsv datasets/iris.data.csv
```

The command takes the file path as a string and starts an interactive process so the user can set the variable types and names. The following message will pop up in the screen:

```
We need to know the kind of attributes present in the CSV
file. Introduce such types of variables. Example:
    r,r,r,c
In this case the csv file will have 4 columns, being the
three first real values and the last one a categorical one.

Introduce the kind of variables:
```

In this case we have 5 columns with 4 real valued attributes and the categorical labels:

```
Introduce the kind of variables: r,r,r,r,c
```

The next step is to introduce the name of all the attributes including the label

```
Introduce the names of the variables: sepal-length,
                                      sepal-width,
                                      petal-length,
                                      petal-width,
                                      specie
```

Finally, we need to introduce the position of the label within the attributes, in this case it is the 5th attribute:

```
Now we need to know which parameter specifies the solution of
the case. Indicate that through a number
Introduce the variable that is the solution (number): 5
```

This last step finishes the load of a CSV.

The next step the user may want to take is to set the weights for the attributes, this step is optional and if the user does not specify any custom weights all the attributes will be given equal weight. Let's add some random weights using the `setWeights` command to illustrate the process:

```
(Cmd) setWeights
Introduce petal-width:
3.14
Introduce sepal-width:
2.71
Introduce sepal-length:
1.414
Introduce petal-length:
1.618
```

Before introducing new cases and running the CBR loop the user can set some more settings to customize the CBR process. For instance, the user can set the number of similar cases that will be fetched in the retrieve step:

```
(Cmd) setRetrieveParameter 5
```

We have set the retrieve parameter to 5, this step is also optional and the default value is 1.

In addition the user can also select the library structure that is used to store and retrieve the case: flat memory or hierarchical memory. To change from one to another use the `setMemoryModel (flat | hierarchical)` command.

Once the CBR system has been set the user can start introducing new cases. We can input them using the `newCase` command:

```
(Cmd) newCase
Introduce petal-width:
3.0
Introduce sepal-width:
3.5
Introduce sepal-length:
 4.0
 Introduce petal-length:
 4.1
```

Now that we have loaded a new case we can call the `executeCBR`:

```
====== CASE ======
-- Attributes
petal-width :  1.5
sepal-width :  3.0
sepal-length :  5.4
petal-length :  4.5
-- Label
specie :  Iris-versicolor
==================
sim:  0.312895672112

====== CASE ======
-- Attributes
petal-width :  1.7
sepal-width :  2.5
sepal-length :  4.9
petal-length :  4.5
-- Label
specie :  Iris-virginica
==================
sim:  0.31409347541

====== CASE ======
-- Attributes
petal-width :  1.6
sepal-width :  3.4
sepal-length :  6.0
petal-length :  4.5
-- Label
specie :  Iris-versicolor
==================
sim:  0.314331159615

====== CASE ======
-- Attributes
```

```
   petal-width :  2.3
   sepal-width :  3.4
   sepal-length :  6.2
   petal-length :  5.4
   -- Label
   specie :  Iris-virginica
   ==================
   sim:  0.31532658149

   ====== CASE ======
   -- Attributes
   petal-width :  2.4
   sepal-width :  3.4
   sepal-length :  6.3
   petal-length :  5.6
   -- Label
   specie :  Iris-virginica
   ==================
   sim:  0.324688551

   ADAPTATION - Result (Solution):
   Iris-virginica
```

The solution of the new case is then `Iris-virginica`.

After the system has presented what it considers the best solution, the user is asked to input the correct solution value for the new case, so that the system will know if it has given the right or wrong answer.

```
-- Label
specie :  Iris-versicolor
Utility:  0
Evaluation:  True
==================
sim:  0.433615819209
ADAPTATION - Result (Solution):
Iris-versicolor
Introduce the real solution:Iris-setosa
```

After this step, the system takes control of the remaining evaluation/retain phase of the CBR, and loops back to the beginning state, ready to admit new cases, while its memory is now, ideally, more capable of providing a correct answer to a new case, and even more with each iteration of the CBR process.

Furthermore, there is one functionality that tests the CBR with the dataset loaded to check its accuracy by executing several tests. After the execution of such tests it outlines the results giving to the user the accuracy reached for each one. Thus, the user could know which are the best settings of the method for the given dataset.

3 Implementation details

We want to remark that one of the pros of this implementation is how it is structured. It maintains a modular structures allowing its extension. For example, if a new similarity distance for real values is needed we only have to add such similarity distance in the class Attributes. This programming approach is applied along all the CBR tool implementation.

As we can see in the guided example, we allow to the user to load any csv that have a dataset being the CBR the one that adapts to the dataset once it knows the order of the attributes. It is not limited neither by the number of attributes nor its nature.

3.1 Case structure

A case in our system is represented as a hash map between the attribute names and the information contained in an attribute. This way the case structure can be general enough to hold any kind of attribute. A single case also holds a solution (label) that is also a hash map. This allows the user to encode anything as a solution: real/categorical value, a multi-label problem or even setting as label all the attributes which can be useful in recommendation systems. A case also holds some meta data about its utility and its evaluation.

A single attribute has its own type that encapsulates the information about itself together with a similarity function. This way real attributes that contain real values but have different similarities have a different type. This first version of AICS ships with the following attribute types: `RealAttribute`, `CategoricalAttribute`, `StringAttribute`. The user can expand the system by adding more attribute types that fit its needs.

3.2 Library structure

We have implemented two types of library structure: a flat memory and a hierarchical memory. The first one is a contiguous memory representation of the cases. The second uses a k-d tree to make a hierarchical representation of the cases. The k-d tree uses inter-quantile spread as attribute selection method. However, k-d tree can not handle categorical attributes. To fix this problem we have decided to split only real attributes and when we cannot split further we leave batches of instances in the labels. This way if the data set has only real attributes we get the usual k-d tree and if the data set has only categorical attributes we get a flat memory representation. How to retrieve from this kind of k-d tree is explained in the next section.

3.3 Retrieval

The retrieval phase in the flat memory case is pretty straightforward. We need to compute the similarity between the new case and all the cases stored in memory. This requires iterating over all the stored instance and can be expensive if the number of instances is large. Hierarchical memory representation solves this problem partially by using a k-d tree in which retrieval can be done in $\log n$ operations (where n is the number of stored instances).

In the case where we have a mixture of real and non real data we can retrieve in the k-d tree by traversing the tree using only the real attributes. The tree then returns a batch of instance and we can do a linear iteration over the returned instances to find the ones with lower similarity (being the number of returned instances much lower than the total number of instances).

3.4 Similarities

As we have seen, the similarity calculation between the cases stored in memory and the new case introduced by the user is done by means of the attributes encapsulated in the case class. Since a case may be composed by attributes of different nature, that might need different similarity measures, these measures are calculated locally at each attribute.

Hence we compute a straightforward Euclidean distance for real valued attributes, normalized to return a value between 0 and 1, being 0 the most similarity possible and 1 the most dissimilarity possible. The total similarity for a given case is the sum of each of its attributes' similarity. Furthermore, we can set different weights to each attribute independently in those cases that some attributes are more important than others.

In the case of categorical attributes, the similarity is just checking whether the two of them are an exact match or not, a Boolean indicating exact match with 0 and no match with 1.

For the case of strings we have implemented a function that measures the Jarow-Winkler distance between two strings. This distance takes into account string length, character matches, position of the matching character, and several permutations of the string characters.

The structure of the attributes and cases makes it quite easy to add different similarity measures to existing attribute types, so, for instance, using cosine distance or other desired distance measure, just implies building it into a simple function and calling it from within the attributes.

3.5 Adaptation

For the adaptation phase, first of all we collect as many cases as we have specified to the retrieval tool. Then, from this set of cases that are similar to the new case proposed, if there is a consensus, that is, if all retrieved cases possess the same class label, we conclude that the new case is also of that class.

When there are several classes to choose from, we implemented an approach that takes into account the utility and evaluation of the retrieved cases.

Each cases have assigned an utility and an evaluation. A high utility means that such case have been useful lots of times and consequently, the solution that it proposes have more relevance. The evaluation says if the cases is well resolved since we can have stored at memory wrong-classified instances.

Hence, our approach relies into a punctuation system. For each of the retrieved cases we give some punctuation for the solution that it proposes. This punctuation is weighted proporcionally to the utility. Moreover, this punctuation could be positive or negative depending if the case is well or wrong classified. For example, we can have one wrong-classified similar case with high utility. This will lead into a decreasing on the punctuation of the solution that proposes such case. This approach is based in the idea that if we store also the wrong classified cases they can help us to not repeat mistakes. Hence, we take a decision based on the utility attribute of the classes, choosing the one with the highest utility.

Finally, if there there is no possibility of discrimination, the case is adapted randomly and we pass it to the next phase to be evaluated and retained.

Furthermore, as commented above, the code allows to implement new adapting approaches in a modular way. We can modify the attributes classes by adding new adapt function without need of modify the rest of the code, the program is thought to be extensible.

3.6 Retain

Our retain phase is actually more dedicated to evaluation of the cases retrieved and the case presented. Each time that the CBR system offers a solution to a new case, the user is asked to input what would be the correct solution to that case. If answer given by the system was correct, the case is retained as a successful one, and the cases that served to adapt its solution are given +1 utility. Whereas if the answer given by the system does not match the answer provided by the user, the case is retained as a failed solution, and the cases that were retrieved to adapt it are flagged with a -1 to utility.

The system actually retains each of the new cases presented, and this increases noticeably its performance. The number of cases retained could be capped if necessary, to avoid computational overhead during the retrieval phase, but since we are not working with any massive dataset it does not become necessary.

Again, the implementation allows to add new retain policies without effort.

4 Testing

As said in the guided example we could execute one function for testing the CBR. In order to give some results here in this document we just executed twice our tool using two different datasets.

In the first case we loaded the famous iris dataset giving to the CBR the right structure. After that we just executed the test. That test gives to us the number of errors made en each iteration. What this test does is split the loaded dataset into a train dataset (which will be the ones that will be introduced in the case library) and a test dataset. However, it make different partitions, *i.e.*, using a ratio it uses some part to belong to the train dataset and the rest to the test. Furthermore, it samples the original dataset in a balanced way.

In Figure 1 and Figure 2 we can see how the errors predicting the class behaves in function of the ratio. However, Figure 2 takes into account the number of cases. The iris dataset contains 150 items. Hence, if the ratio

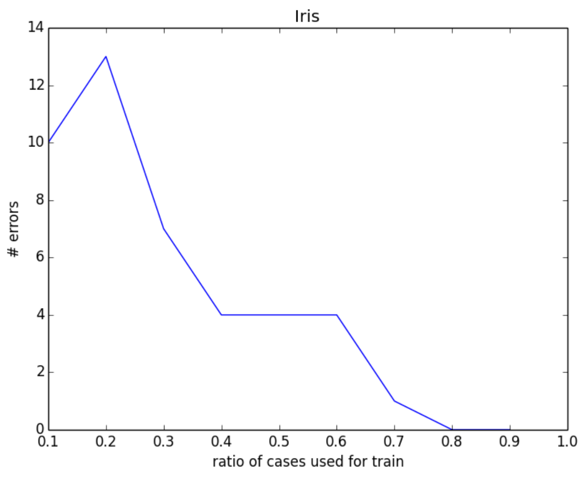

Figure 1: Iris

is 0.1 the training set will have 15 items while the test set 135. Hence in Figure 1 we have that 13 errors of 135 instances. Figure 2 takes into account the amount of cases that falls into each dataset and outlines the percentage of error.

This charts allows the user to use the best setting. Moreover, we can observe that the results are good, even with a very short ratio of the instances we can predict with a 90% of accuracy

Furthermore we also tested the titanic dataset. The procedure was the same but in this case we give to the CBR the structure of this dataset (without changing the code). Figure 3 and Figure 4 shows the corresponding charts.

In the titanic case we have more instances (891) and each instance is more complex (it have more attributes and also more heterogeneous). In this case, hence, we infer that for this dataset we need a bigger ammount of cases stored in the case library in order reach better accuracy.

In Figures 1,2,3 and 4 the K parameter was fixed to 5. However, we can play, among others, with this parameters. For example, Figure 5 and 6 draw the same Titanic results but executed with K equal to 10.

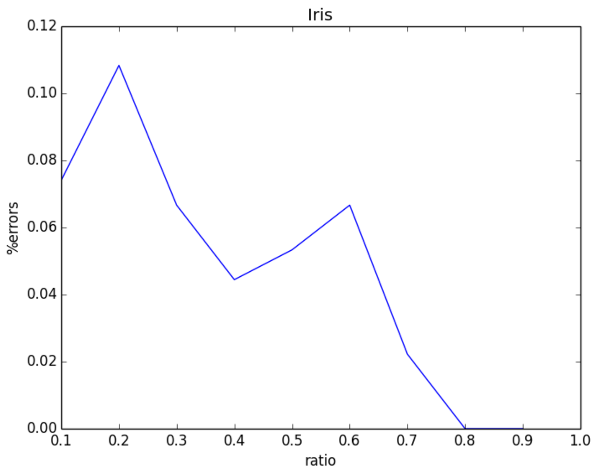

Figure 2: Iris

5 Conclusion and Future work

We built an interactive CBR that allows the user to execute it with any dataset since it can provide the structure by calling the implemented functions. It is programmed taking care about modularity, scalability and extensibility. With that characteristics it is quite straightforward to add new similarity functions of a particular kind of attribute without mesh up the rest of the processes, we only need to add the function and the program will handle it.

As a future work, we started an implementation aiming to accept a list as a kind of attribute. We had to rollback in order to have a working version for delivering. However, we want to remark the code is thought to incorporate this feature.

We believe that the incorporation of the list as a kind of attribute and consequently nested attributes will generate an exceptional generality. This is because we will be able to select lists as a solution (being able to recommend any kind of structure) covering different CBR tasks in a general model, have lists of lists of attributes wining a huge versatility or both.

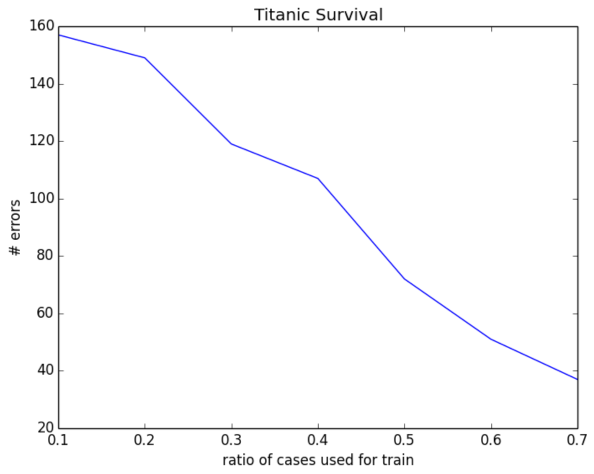

Figure 3: Titanic

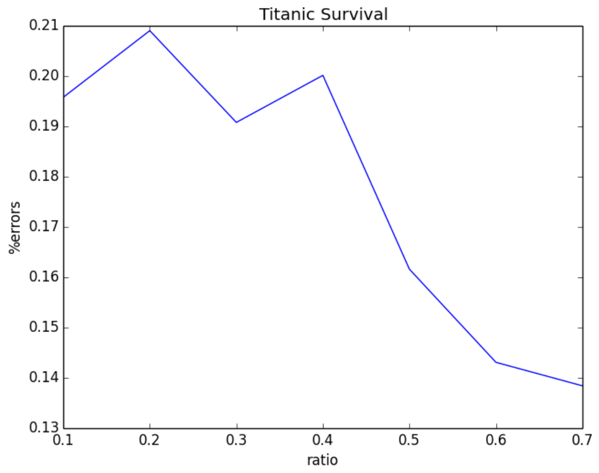

Figure 4: Titanic

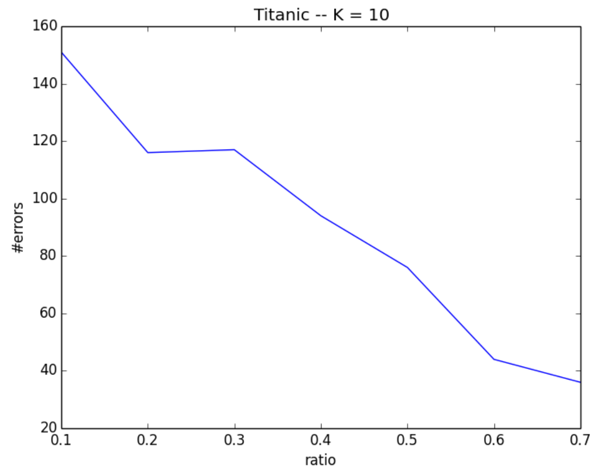

Figure 5: Titanic

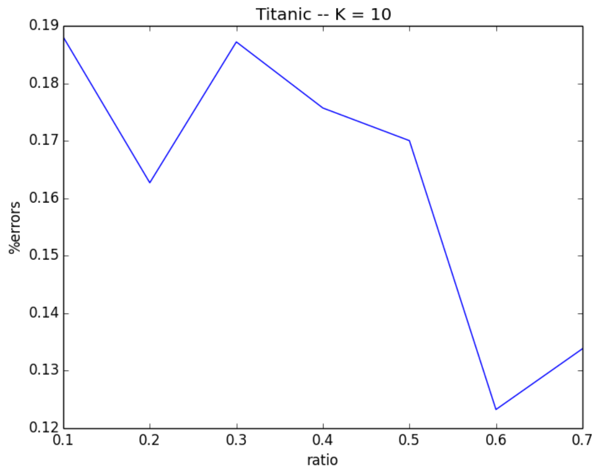

Figure 6: Titanic