CO395 Machine Learning
CBC #3
Case Based Reasoning

Group 1

Yong Wen Chua, ywc110

Thomas Morrison, tm1810

Marcin Baginski, mgb10

Marcin Kadziela, mk4910

Contents

Implementation Details

Overview

We implemented a case as a MATLAB struct with the following fields:

- `au` - which represents the AUs which are active
- `class` - which represents the emotion (can be 0 for a case which does not have a solution yet)
- `typicality` - which represents the number of times this exact case has been encountered during training and subsequently during the RETAIN phase

Our Case Base is a column vector where each cell represents a case.

Description of the MATLAB functions

- `CBRInit(x,y)` - this function initialises the case base. It takes as arguments the examples `x` and classifications `y` and returns the case base in the format described in the previous subsection
- `retrieve(cbr, newCase)` - this function implements a modified version of the k-NN algorithm to return a set of cases which closest match the `newCase`. Along with the cases, the function reports a distance from each returned case to the `newCase`. The modification of the original k-NN algorithm comes from the fact, that if at least two largest distances returned from the function are equal, the function returns `k+n` cases such that `n` is the smallest natural number which makes sure that no case which have the distance equal to the largest one in the original `k` cases is omitted. This modification is useful because in situations where there are more than `k` cases with the same distance to the `newCase`, we do not want to restrict ourselves to considering only the ones, which appear first in our case base. We implemented the Manhattan and Euclidean distances as our similarity measures.
- `reuse(cases, newCase)` - this function takes as arguments the cases returned by the `retrieve` function as well as the new case without a solution. It performs the majority-voting weighted by the distance of the existing `cases` in order to determine the most probable solution to the `newCase`. If there are two or more emotions which have the equal number of 'votes', then the preferred 'vote' is given to the case with a higer value in the `typicality` field. If there is still ambiguity even after this step, the function removes one of the existing `cases` which has the largest distance and repeats all the steps. The algorithm will terminate in the worst-case scenario when it will be left with just a single case
- `retain(cbr, newCase)` - this function updates the Case Base with the new case. If the new case does not already exist in the case base, it simply appends it at the end. Otherwise, it increments the `typicality` field in an identical, existing case
- `makeCase(AUs, class)` - creates a single case struct as described above. It can take either of the two representations of the active AUs as an argument
- `nFoldValidate(examples, classifications, n)` - preforms the n-fold cross-validation and returns the confusion matrices for each fold
- `testCBR(cbr,x2)` - takes the Case Base and a matrix of examples to be classified and returns a column vector of predictions

Evaluation

The results presented in this section are for the modified (as explained in the Implementation Details) distance-weighted **5-NN** algorithm on the *clean* dataset and **9-NN** on the *noisy* one. We chose these algorithms because they performed best during our tests. Other variations of the algorithm are analysed in the next section under 'Comparison of similarity measures'.

Clean dataset

		Predicted class					
		1	2	3	4	5	6
Actual class	1	**97**	18	4	4	8	1
	2	7	**178**	1	6	5	1
	3	3	3	**96**	0	2	15
	4	0	6	1	**207**	0	2
	5	4	16	2	5	**103**	2
	6	1	1	6	6	0	**193**

Table 1: Confusion Matrix for the modified 5-NN distance-weighted algorithm on the *clean* dataset

		Recall	Precision	F_1
Actual class	1	73%	87%	80%
	2	90%	80%	85%
	3	81%	87%	84%
	4	96%	91%	93%
	5	78%	87%	82%
	6	93%	90%	92%

Table 2: Recall, precision and F_1 measure for the modified 5-NN distance-weighted algorithm on the *clean* dataset

$$C = \frac{874}{1004} = 87.1\%$$

Figure 1: Classification rate for the modified 5-NN distance-weighted algorithm on the *clean* dataset

Noisy dataset

		Predicted class					
		1	2	3	4	5	6
Actual class	1	**15**	11	31	4	22	5
	2	0	**160**	15	3	6	3
	3	4	10	**133**	9	6	25
	4	3	8	11	**170**	4	13
	5	5	5	14	2	**74**	10
	6	1	1	8	7	9	**194**

Table 3: Confusion Matrix for the modified 9-NN distance-weighted algorithm on the *noisy* dataset

		Recall	Precision	F_1
Actual class	1	17%	54%	26%
	2	86%	82%	84%
	3	71%	63%	67%
	4	81%	87%	84%
	5	67%	61%	64%
	6	88%	78%	83%

Table 4: Recall, precision and F_1 measure for the modified 9-NN distance-weighted algorithm on the *noisy* dataset

$$C = \frac{746}{1001} = 74.5\%$$

Figure 2: Classification rate for the modified 9-NN distance-weighted algorithm on the *noisy* dataset

Discussion of results

Questions

Two or more best matches

Our `retrieve` function returns a set of `k` or `k+n` cases which best match the new case so this is not an issue at this point. However, those returned cases might have the same distance. This issue is dealt with in the function `reuse` which computes majority-voting weighted by the inverse distances of the returned cases from the `retrieve` function. The initial algorithm proceeds as follows:

`votes` $\leftarrow [0, 0, 0, 0, 0, 0]$
for all `cases` **do**
  votes(`cases.class`) $+= \frac{1}{\texttt{distance(case,newCase)}}$
end for
return `index(max(votes))` ▷ return the index of the bucket with maximum number of votes

The reason behind using the inverse distance of the `cases` is that the closest a given case to the `newCase`, the bigger weight of the vote it should have.

If the above algorithm is inconclusive, we additionally weigh the votes by the `typicality` field of the `cases`. Should this fail as well, we remove the case with the smallest distance and repeat all the steps. Ultimately, we will be left with just a single case and we will assign its class to the `newCase`.

Addition of an existing case to the Case Base

Since we implemented a `typicality` field in our Case Base which counts the number of times an identical case has been encountered, we simply increment this value. This field is sometimes used in the `reuse` function, as described in the previous question.

Comparison of the similarity measures

Our similarity measures are the Euclidean:

$$D_E = \sum_{AU=1}^{45} |newCase(AU) - existingCase(AU)| \tag{1}$$

and Manhattan distances:

$$D_M = \sqrt{\sum_{AU=1}^{45} (newCase(AU) - existingCase(AU))^2} \tag{2}$$

for which the average classification rates are presented in Tables 5 and 6. We evaluated the distances on two versions of the k-NN algorithm (sweeping `k` in the range from 1 to 10):

- Simple k-NN - which performs the majority voting without weighing the votes of each returned case by its distance to the new case
- Distance-weighted k-NN - which performs the majority voting weighing each vote by the distance of the case as described in the previous section

Obviously, since in our problem the Euclidean distance is simply a square root of the Manhattan distance, the same cases are going to be returned in each itaration. Therefore, the Simple k-NN algorithm performs identically for both types of distances. There is a slight difference in the performance of the Distance-weighted k-NN for the Euclidean distance because the weights assigned to each vote are essentially square roots of the original Manhattan distances (i.e. the cases with bigger distances are not penalised as heavily).

	Simple k-NN		Distance-weighted k-NN	
	clean	*noisy*	*clean*	*noisy*
1-NN	80.4%	64.6%	80.4%	64.6%
2-NN	84.1%	69.3%	84.6%	69.9%
3-NN	**86.2%**	72.5%	86.6%	72.2%
4-NN	86.1%	72.1%	86.8%	72.8%
5-NN	85.9%	72.6%	**87.1%**	73.5%
6-NN	85.2%	73.6%	86.5%	74.2%
7-NN	85.0%	**73.7%**	86.3%	74.2%
8-NN	85.0%	73.4%	86.2%	74.2%
9-NN	84.9%	73.5%	86.1%	**74.5%**
10-NN	84.9%	73.6%	86.1%	74.5%

Table 5: Comparison of the different versions of the simple and distance-weighted k-NN algorithms for the *Manhattan* distance

	Simple k-NN		Distance-weighted k-NN	
	clean	*noisy*	*clean*	*noisy*
1-NN	80.4%	64.6%	80.4%	64.6%
2-NN	84.1%	69.3%	84.4%	69.0%
3-NN	**86.2%**	72.5%	86.3%	72.3%
4-NN	86.1%	72.1%	86.5%	72.6%
5-NN	85.9%	72.6%	**86.8%**	73.3%
6-NN	85.2%	73.6%	86.1%	73.9%
7-NN	85.0%	**73.7%**	85.8%	73.9%
8-NN	85.0%	73.4%	85.8%	73.9%
9-NN	84.9%	73.5%	85.8%	74.2%
10-NN	84.9%	73.6%	85.9%	**74.3%**

Table 6: Comparison of the different versions of the simple and distance-weighted k-NN algorithms for the *Euclidean* distance

Analysis of Tables 5 and 6 shows that the Manhattan distance performs slightly better for each version of the Distance-weighted k-NN algorithm.

Initialisation of the Case Base

We initialise the Case Base by creating a struct (described in the Implementation Details section) for each new case and adding it to the column vector of such structs which represents our entire Case Base. If we encounter the same case multiple times, we increment the `typicality` field.

Eager vs. lazy types of algorithms

CBR belongs to a lazy learning class of algorithms. As opposed to the eager learning algorithms which construct an explicit description of the target function as soon as the training examples are provided, the lazy learning algorithms only store the data and postpone the classification of the new examples until an explicit request is made. The decision tress and neural networks belong to the eager learning methods. As soon as the tree/network is trained, we can get rid of the training examples and never use them again. The trained tree/network is enough to classify any new, incoming example. In CBR however, the training examples are stored in a Case Base, which might increase its size with each new, correctly solved example. One of the most important advantages of the lazy learning method is the fact, that it can estimate the

target function locally for each encountered example, instead of creating a single, global and unchangeable description of it.

Clean vs. noisy datasets

Code Flowchart

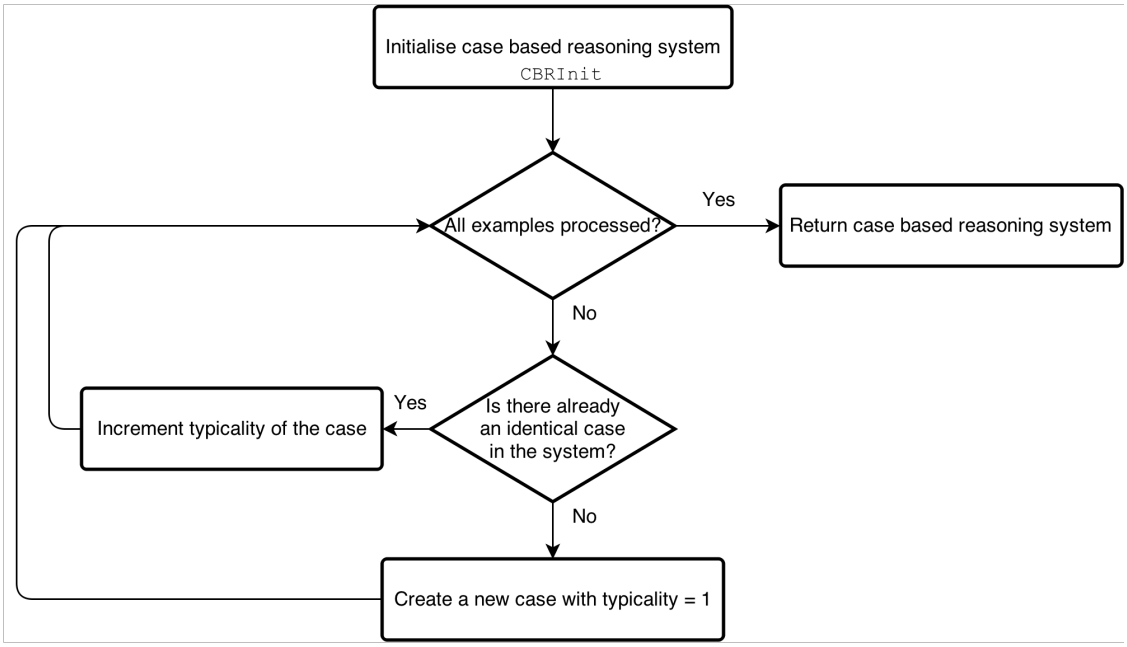

Figure 3: Flowchart of the `CBRInit` function

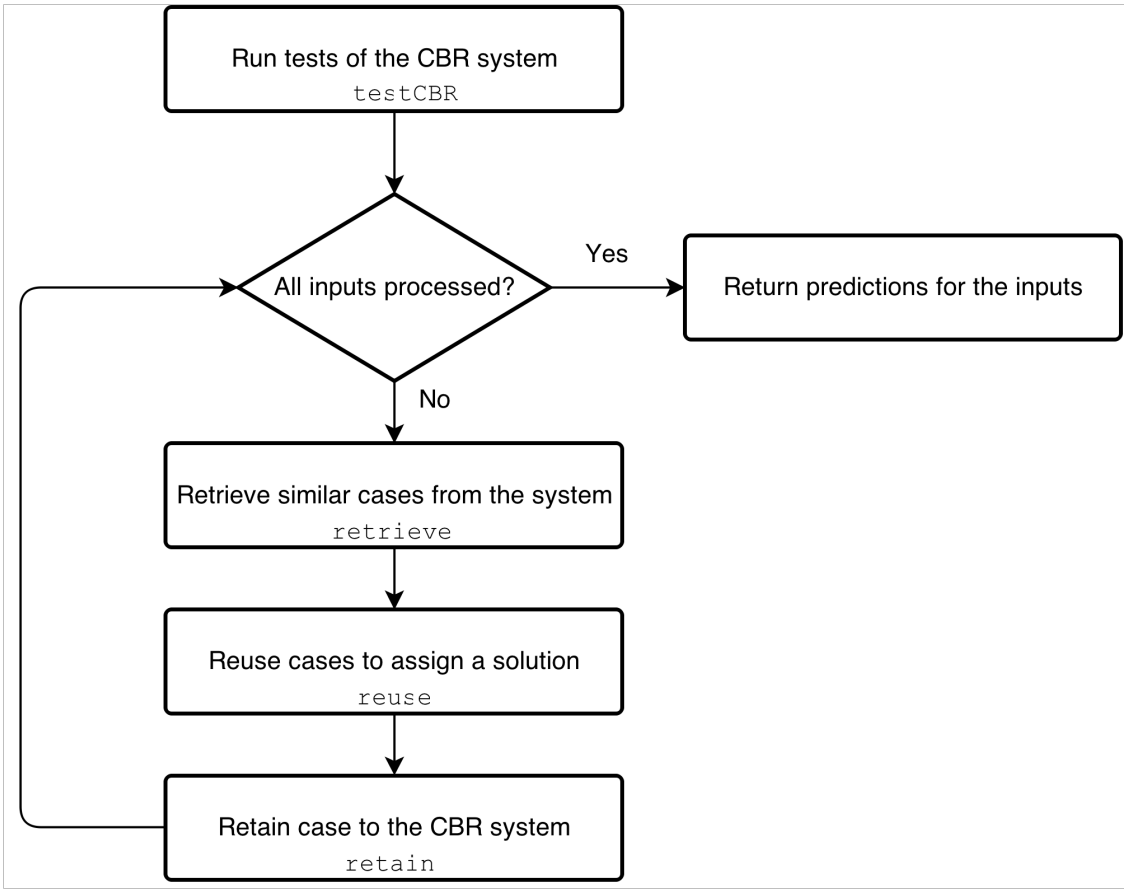

Figure 4: Flowchart of the `testCBR` function