D&D Case Base Reasoning System in Java

The System

The system is a case based reasoning (CBR) system[2][3] built in Java using the android API. Java is the chosen language because it is the programming language best suited for the system because the CBR system can be implemented using SQLite[1] with an android GUI front end.

The system itself emulates the class definitions found in games like 'Dungeons and Dragons' (D&D), where users would assign stat points to their avatar and the avatar would be given a classification. The CBR system takes into account these stat points and predicts what 'class' the avatar is.

There are three stat points and three potential classes in this system. The stat points are Strength, Agility and Intelligence. The classes are Warrior, which has high Strength (Str) medium Intelligence (Int) and low Agility (Agi), Rogue, which has high Agi, medium Str and low Int, and Wizard, which has high Int, medium Agi and low Str. All avatars have 100 points to spend, a Warrior for example will have 70 points in Str, 25 int and 5 Agi.

The system auto generates 150 cases of training data and takes ten from those cases as a training set. The avatar in question is then compared to the existing cases and a similarity percentage is produced. The system uses 3NN to determine what class the avatar should be. Each stat has a weighting attached to it which can manipulated by the user to get obtain desired data.

The output is displayed on the GUI in the android application.

Inputs and Outputs

The case base system is handled by the SQLite database. The database stores 150 cases which are auto generated. The cases are created using a random number generator with the algorithm in the genHero() method. For each class the most important stat is generated between 50 and 90, the second highest stat is then randomly generated between the remainder and then the last and lowest stat is the total remainder. 10 is then added to the important stat to ensure that the total is out of 100 which avoids various errors in random number generating.

An example for Warrior:

1. 70 is generated by the random number generator (RNG) between 50 and 90. This will be stored in the Str stat.
2. There is 20 left so another RNG value is generated giving the value of 15. This will be stored in the Int stat.
3. There is 5 left over, this will be stored in the Agi stat. 10 is then added to the Str stat to equal 100.
4. The case is now a Warrior with 80 Str, 15 Agi and 5 Int.

This case is added to the database. The method's parameters take into account how many values are generated as well as the desired class and if they should be printed to logs. For example If 'Wizard' is passed instead of 'Warrior' then the highest value is stored in Int rather than Str. This creates a case base that can be queried by the user's target case.

To query the case base the user would input values into the input fields to create a target case, for example: Str 60, Agi 40 Int 0. The user would input this into the GUI, the SQLite backend then takes these values and finds the closest three values (3NN) and calculates the most probable case while taking into account the weighting. The lower the weighting the more important the stat is when querying the case base due to producing a lower similarity value, as shown in Figure Input.

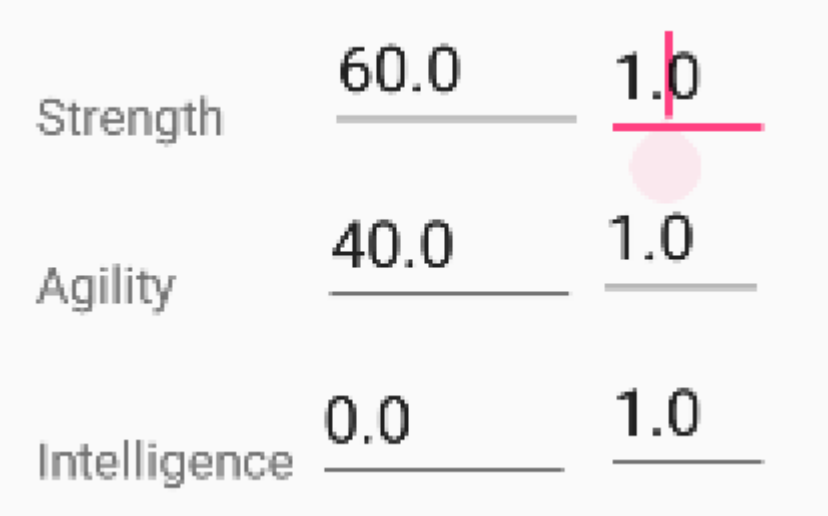
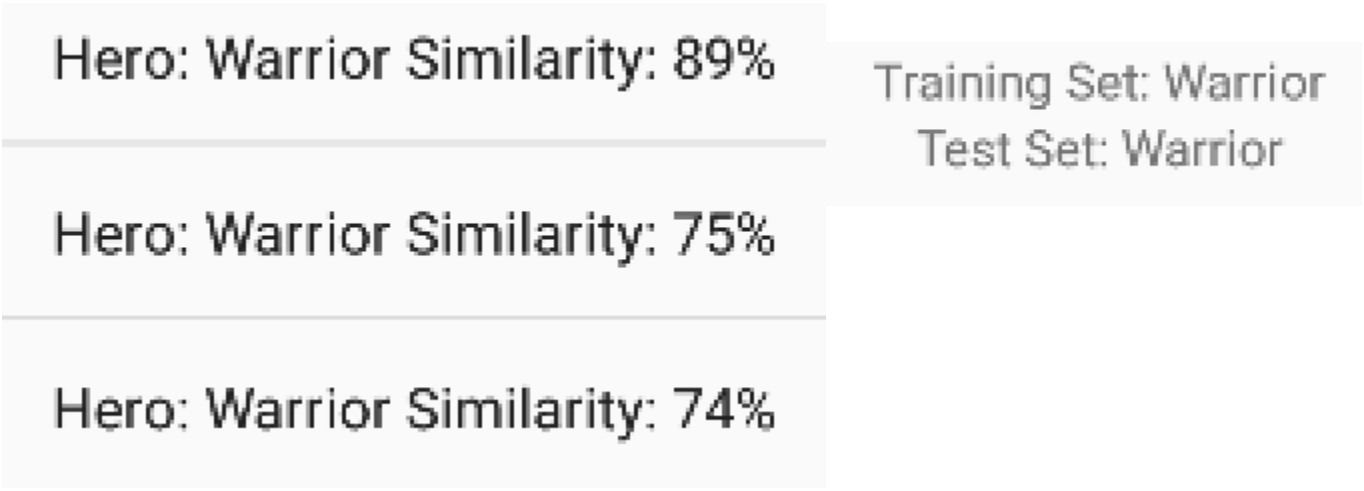


Figure Input

The database is iterated through and the similarity values are calculated for each case and they are given a percentage ranging from 100 to zero. 100% is the most similar and 0% is the least similar. Ten entries in the test base are taken randomly and this produces a training set of data which is displayed in a list view in the GUI. Both of the sets of data are arranged by the similarity percentage and 3NN is calculated for both the training set of data (ten cases) and the test set of data (140 cases) and the results are displayed as shown in Figure Output.

Figure Output



The result for this example is Warrior which was to be expected and thus the unknown classification for the query is now Warrior, this can be stored in the database for future use.

Real World Problem

The real world problem is that many classification systems are set in stone and are not very intuitive. A classification system may just take the stat with the highest point and categorize it to a class. For example with an entry like 51 Str, 49 Agi and 0 Int the system could classify it as a Warrior just by taking the highest value, but what if the case had tied values like Str 40, Agi 40 int 20. How would a system categorize this?

The CBR system looks at past cases and makes the best judgement based on the closest values. If the test base holds a lot of Rogues with similar values then the system judges the query that it's probably a Rogue. This allows a much more flexible system in determining the classification and it allows the system to grow.

Success of the System

To measure the success of the system multiple tests will be ran. At first testing the "Sure" data which consists of the points heavily in one stat. For example a 'Sure' Warrior will have 90 Str, 10 Agi and 0 Int. This is to provide a baseline for the rest of the testing for the test data and the training data.

The testing will then test 'Probable' cases which would be more typical entries such as determining a Rogue with 60 Agi, 20 Str and 20 Int. This testing will take place across all three categories.

After the 'Sure' and 'Probable' testing has taken place the unpredictable testing will take place where cases that are unsure of the outcome will be tested three times to eliminate some of the RNG. An example of this would be the example mentioned above with 40 Str, 40 Agi and 20 Int. This testing will involve manipulating the weights to try and force the results to swing a certain way. If the testing goes as expected then the system is deemed successful.

Both the training set and the test set of data will be tested in both cases and the results will be analysed.

Challenges of building the system

SQLite was implemented into Java to create the CBR system. The system itself required a lot of code to emulate CBR systems in different frameworks. The challenges of the system were to create a database that would hold generated data.

Drawing data from the database and inserting the data into case ‘Objects’ was the prefered method as it would store every single entry as a separate object which meant data could be pulled from the ArrayList tuned to the user's specification if a future feature would be implemented, for example if only the Str values were going to be measured then the objects could be queried to do so.

This object oriented approach required a lot of extra code but the benefits from implementing the system outweighed the time and learning that it took to implement them.

Implementation

Initially when the application is loaded the user is presented with three fields in which to enter data alongside the associated weighting for each field. These will be used in the algorithm to enter data and determine the similarity.

The user enters the relative data, for example leaving the weights as default on 1.0, the data 70 Str, 20 Agi and 10 Int are entered. This is predicted to be a Warrior classification due to the high Str value but the user is unsure, so the user polls the CBR system with this target set of data, this is done by pressing the ‘Start’ button.

The system takes the values in the text views for the stats and the associated weighting and stores them in variables. The database is then created as well as generating 150 cases which are inserted into the database shown in Figure Gen Hero

Figure Gen Hero

```
int min = 50;
int max = 90;
Random r = new Random();

for (int i = 0; i < entries; i++) {

    int i1 = r.nextInt(max - min + 1) + min;

    int i2 = r.nextInt(100 - i1);

    int i3 = r.nextInt(100 - i1 - i2);

    i1 += (100 - (i1 + i2 + i3));

    switch (hero) {
        case "Warrior":

            //Warrior highest value Str, then Int, then Agi
            cbrDB.execSQL("INSERT INTO heroes (name, str, agi, int) VALUES ('Warrior', " +
                    String.valueOf(i1) + ", " +
                    String.valueOf(i3) + ", " +
                    String.valueOf(i2) + ") ");

            Log.i("Result: " + hero, i1 + ", " + i2 + ", " + i3);
            break;
```

The algorithm used in Gen Hero fills the database with expected case data which can be used to determine what class the target data belongs to. The alternatives to this algorithm were to hard code every entry which would take far too much time, and as the database stands as a proof of concept in a theoretical problem, there is no real data to pull data from.

Once the database has been created there are two arraylists to hold the data, one for the training set of data which is available to the user and the test set which is hidden and only written in the backend code.

Firstly the test set of data is generated by iterating through the database for every entry and taking the Str, Agi and Int values from the data as well as the weightings. This is compared to the target data and given a similarity percentage. 100% being the most similar.

To calculate the similarity the difference between the target stat and the case start is calculated, this is then multiplied by the weighting for each stat. This is done for all three stats and then they are totalled and divided by the total possible difference as shown in Figure Similarity.

Figure Similarity

```
double divider = ((strW * 100) + (agiW * 100) + (intW * 100) );
double total = 0;
double a = (agi-agiT) * agiW;
double b = (str-strT) * strW;
double c = (intel-intelT) * intW;
```

For example using the target data, if the case in question has 75 Str, this would calculate 5 as the difference and multiply it by the Str weight which in this case is 1. This is then repeated for the other values and totalled. The algorithm then works out the maximum difference possible and divides the total, then the result is worked into a percentage. For example the case in question might be 82% similar.

This similarity value is then stored in the arraylist along with the class that was polled. An example case would be Warrior, 82% Similar as stored as a ‘Hero’ object. Initially the value didn’t have a percentage and just a number, this was changed as the similarity percentage gave some context to the value whilst a value of 9 did not offer much insight.

After the test set of 150 cases is entered, the training set is generated and stored in a separate arraylist. The training set takes ten random index values and stores them in a separate arraylist and removes them from the test set as shown in Figure Training.

Figure Training

```
public void fillTestList() {

    testList.clear();
    Random r = new Random();

    for (int i = 0; i < 10; i++) {

        //Adds data to the test list, removes it from the training set

        int x = r.nextInt(heroList.size());

        testList.add(heroList.get(x));
        heroList.remove(x);
    }
}
```

There is now a test set of data with 140 cases and a training set of data of ten values randomly pulled from the test set with the correct class names and the associated similarity percentage.

After this the getAnswer() method is called which passes the arraylist of hero objects to be sorted and the 3NN found. The algorithm sorts the data by it's percentage and lists the highest percentage first, meaning that the most similar values are listed on top as shown in Figure Answer. The algorithm then loops over the top three and gets the class name and inputs this on a counter. The counter with the highest number is selected and this is output as the predicted class.

Figure Answer

```
Collections.sort(list, (Comparator) (lhs, rhs) → {
        return Double.compare(rhs.getSimilarity(), lhs.getSimilarity());
});

//Log.i("X", list.toString());

for (int i = 0; i < 3; i++) {

    Log.i("Top " + (i+1) + ": ", list.get(i).getHero());

    switch (list.get(i).getHero()) {

        case "Rogue":
            rogCounter++;
            break;
```

The algorithm has finished once both sets of data have been polled, creating separate methods allows for great flexibility when deciding where the data would be shown on the GUI.

The 3NN has been found for both sets of data and displayed in the Text View for the user to view the results.

Changes

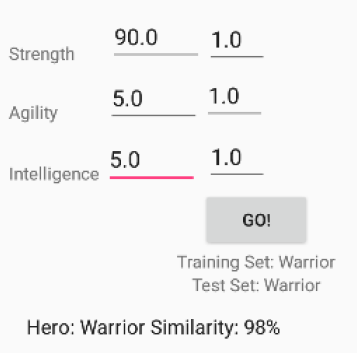
There were three major changes made to the system. The first is that the original system did not have enough data. It only featured 16 hard coded data sets, not enough for a comprehensive CBR system. The this was tackled by creating a method to auto-generate 150 data sets which could be used in the CBR system.

The second change was the percentage similarity instead of a flat numerical value. This change was simple to implement as it required dividing the similarity value over the largest possible difference and then subtract from 100. This was to add context as 99% similar presents more information than '3' similar.

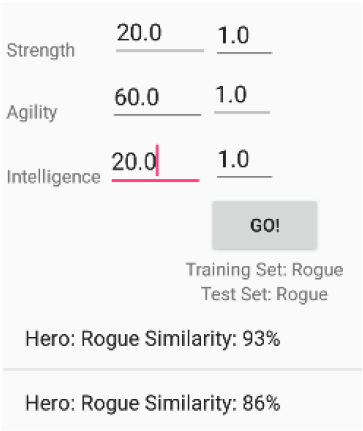
The last change was to implement a training set of data, this was so the data could be judged on its efficiency as all of the test data was shown. It also allowed the weights to be tweaked if a different result is desired.

Testing

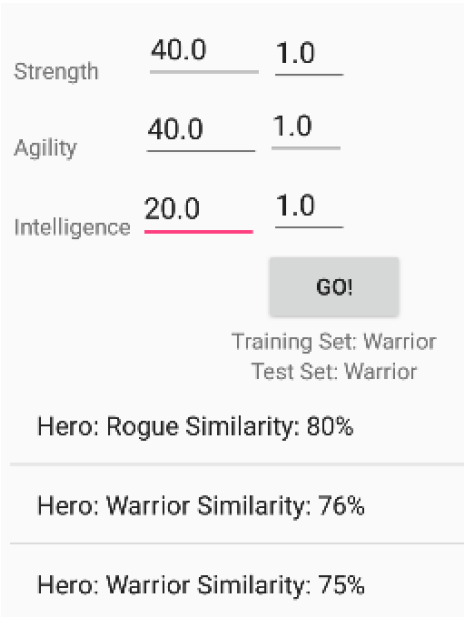
Sure (Testing 90 Str, 5 Agi, 5 Int), expected to be Warrior in both cases, weighting the same.

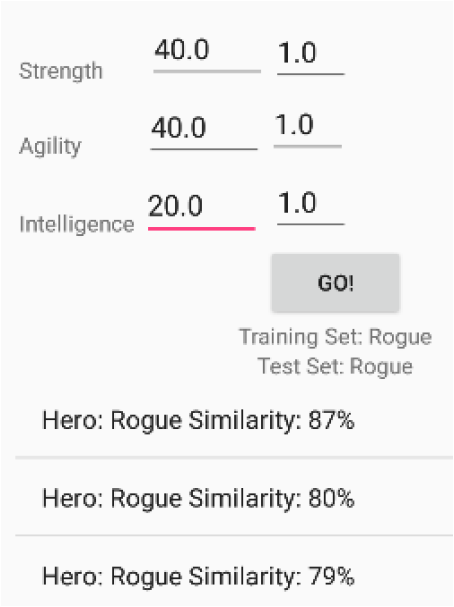


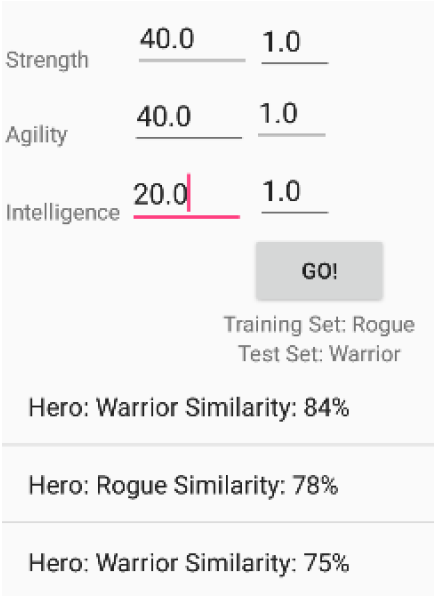
Probable (Testing 20 Str, 60 Agi, 20 Int), expected to be Rogue in both cases, weighting the same.



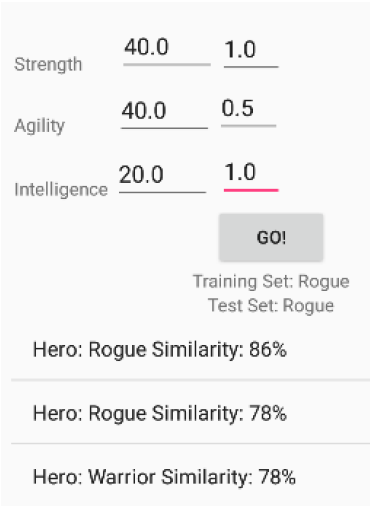
Unsure (Testing 40 Str, 40 Agi, 20 Int), expecting Rogue or Warrior, running test three times, weighting the same.







Unsure (Weighting Tweak), Same tests as above but with a weighting decrease on Agi, Expecting more Rogue results



Conclusion

The CBR system created in Java works as expected and when tested delivers the expected results.

References

[1] (2016). *Saving Data in SQL Databases.* Available: https://developer.android.com/training/basics/data-storage/databases.html. Last accessed 27th May 2016

[2] Aamodt, Agnar, and Enric Plaza. "Case-Based Reasoning: Foundational Issues, Methodological Variations, and System Approaches" *Artificial Intelligence Communications* 7, no. 1 (1994): 39-52. CBR1

[3] Leake, David. "CBR in Context: The Present and Future", In Leake, D., editor, Case-Based Reasoning: Experiences, Lessons, and Future Directions. AAAI Press/MIT Press, 1996, 1-30. CBR2