# Computing Science


## CS4047: Computational Intelligence

Darren Coutts

No portion of the work contained in this document has been submitted in support of an application for a degree or qualification of this or any other university or other institution of learning. All verbatim extracts have been distinguished by quotation marks, and all sources of information have been specifically acknowledged.

Signed:

Date: November 4, 2015

**CS4047 Report**

**Department of Computing Science**
University of Aberdeen
King's College
Aberdeen AB24 3UE

**November 2015**

# CS4047: Computational Intelligence

Darren Coutts

Department of Computing Science
University of Aberdeen

November 4, 2015

# 1 Fuzzy Logic

Fuzzy Logic, sometimes referred to as Multi-Valued logic is an area of artificial intelligence which can be used to relate the subjective world which we live in, with the rigid world of Boolean Logic. Fuzzy Logic is a problem solving system, which takes use of If Then statements, in order to reason about the world, and prove a real world outcome, even when the exact definition of how such an answer could be formed without using a mathematical approach. Each rule, within the Rule Base adds a piece of knowledge about the mappings from the real world, into the model of the world in which we reason about. The rules can be as straightforward, or as complex as is needed to fully explain the conditions, however each rule must have a single response output.

For example, if we take the following rule;

**If the RoomTemp is Cold AND the RoomLight is Dark Then the curtains should be closed.**

In this rule, we can compare this as simple Boolean logic $\text{Cold} \wedge \text{Dark} \rightarrow \text{Curtains(Closed)}$. This provides us with a simple understanding of how we could crudely create the system to analyse if the curtains in the room should be open or closed. We can imagine that we would need a number of other rules, to explain what should happen to the curtains if the room is not cold and dark. As we are using Boolean logic, we would need $2^2 = 4$ rules to explain all of the possibilities for the system.

What happens when an input does not fall into Boolean values? This can be the case when we are using real world inputs, which could be grouped in a number of different ways, some people could group temperatures simply as Hot or Cold, but others may include intermediate groupings, such as Warm, Mild and Average. In this case, the number of rules we would need under normal conditions increases as the number of categories in these inputs to the power of the number of inputs. For a system of two inputs, with three groups each, High, Medium and Low, there would be 9 rules needed to explain all of the possible input combinations.

Not only does this model require a vast number of different rules incoming, but it is also required to have strict bounds as to which category a value fits into. This creates a very rigid system, and one that does not provide a dynamic response to the real world input values. For example if the temperature in a room is sitting on the cut-off point between two groups, but is varying slightly, the output of the system, when using strict Boolean logic will not take into account the small change, and could cause the output to wildly change. This is deemed acceptable when there are only two possible output values.

If we are to allow constant input values, we must allow a level of cross over between a number falling into one category. For example, you may have a 5% leeway either side of a cut-off point, where it is ambiguous as to which group the input value should fall into. We require a definition of how we appoint a value into one or more groups, which can be used to ensure consistency throughout the system. The way which we represent these groupings in Fuzzy Logic is to use curves, which contains a central portion where inputs are considered to be fully within that group, and the value of its membership is 1. To either side of this, we have a section which tails off to a membership of 0, which may or may not overlap another group. The membership of a group is the percentage (scaled

down to 0-1) of which a value falls completely within the curve of an input graph. An example of such a graph is shown in Figure 1

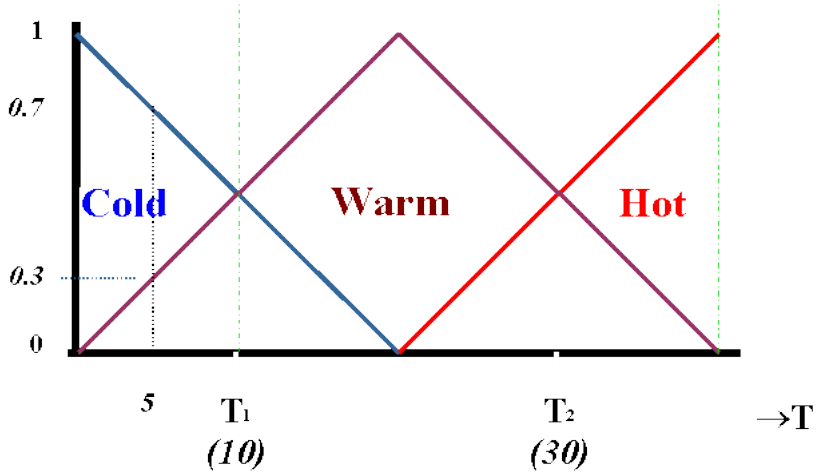


**Figure 1:** Example Graph Containing Membership Curves.

Figure 1 Clearly slows how three groups, Cold, Warm and Hot overlap each other across the baseline of the graph. That said, if you analyse the value of T to be 10 (T1), it is directly on the cross over point of Cold and Warm. At this point, the membership value for T(10) = Cold( 0.5 ) + Warm ( 0.5 ) + Hot ( 0.0 ), that is T1 falls 50% of the way into the cold Curve and 50% of the way into the Warm curve. The membership of any input value must have a total value across all curves of 1.0, representing 100%. If we were to look at T(20), it falls wholly into the Warm curve, and would therefore have a membership of 100% in Warm. Similarly T(5) = Cold( 0.7 ) + Warm ( 0.3 ) + Hot ( 0.0 ) = 1.0.

If we look back on the example rule above, we would need to create a graph of membership curves for RoomTemp, RoomLight and Curtains. The reason for the two input graphs is obvious, as we would need to find out the membership of the real world values, but we would also require one for the output as typically this would also fall into one of many output curves. In the example given it is a simple Boolean output, which would be drawn as a straight line sigmoidal curve, which would simply have a cut-off point, which would alternate the output state of the system.

The above process is known as Fuzzification. Fuzzification is the first part of a Fuzzy Logic system, which will take each of the known real life input values, and calculate the membership of each of these with their respective membership curves. The way in which the actual membership value is calculated is simply done through the use of linear mathematics, using trigonometry, and the equation of a straight line. Once the calculations of the memberships have been completed, we can move onto the next section of the Fuzzy Logic, Rule Firing, which takes into account the rules which have been provided in the rule base, in order to examine which of these rules have their conditions met.

For Rule Firing, we must examine each of the rules, to calculate the likelihood that the rule will

fire. The main part of the rule which we must look at is the logical operator in the rule (and or or). Depending on which of these operations are in the rule, the mathematical function will change. For an AND operation, we must use the minimum of each component of the rule, and if we are using an OR operation, we use the maximum of the components in the rule.

This can be linked back to Boolean logic, when we have an or operation on a simple two input OR gate, we only need one input to be high, and on the truth table for the operation we have 3 possible combinations which will produce a positive output. The Maximum of a list of numbers will produce an output which is higher than a majority of the inputs, the same as an OR gate. The same is likewise for an AND gate and the Minimum operation, we have more input positives than we have output positives. Therefore we use these mathematical operators when we see conditionals in the Fuzzy Logic Rule Firing system.

The system will calculate the overall firing likelihood for each of the rules in the system, and these are related to the output class. Each rule will have a membership curve to which it is related, in our example this would be the curve for Curtains(open). Similarly for any other value which can be set as an output. Rules will be assigned to them. A particular output value may have more than one rule which can provide it as an output, in this case, each of the output classes should contain a list of values which will be handled after all of the rules have been parsed in turn.

Before we can move onto the final stage of the Fuzzy Logic system, we need to have a single value for each of the output classes. In order to calculate this, it is necessary to understand how the rule base is structured. Each of the rules in the rule base are thought to be separate and disconnected from each other, that is the rules have to dependency on one another. In a Boolean sense, each of these rules can be thought of as ORed together. Moving back to the previous explanation as to Boolean vs Fuzzy Logic operators, it was shown that an OR operation is a mathematical Max function. Therefore in order to compute the final value for each of the output classes, it is necessary to calculate the maximum of the list of values generated from the rules. By doing this, we will have a single value for each output class, for example Open or Closed in our above rule base.

Finally, we must revert these output classes back to an overall output for the system. The output from this system will be a numeric value, which will relate to a point on the membership graph for the output variable. The method that will be used in this paper is the dilatation of the aggregate. This creates an approximation of the area of the graph represented up to the value of that particular output class value. For example if the output value for a particular class is 0.5, then we calculate the area of the rhombus which is shown on the graph, and then scale the apex of this down to a half. This does not necessarily provide a 100% accurate definition, but provides a reasonable definition of the area.

In order to calculate the area, we are required to multiply the base length of the curve (the distance between the two zero points on the curve) against the output value for that curve, multiplied by a half. The extra half is included as we are essentially considering the rhombus to be a triangle, and this is the standard area of a triangle equation.

Finally we must use an equation to combine these areas together. We also need to know the

centre of each of the rhombuses, which is equivalent to half of the base, plus the distance from the origin to the first zero point on the curve. The equation which is used is as follows.

$$\chi = \frac{\sum_{i=1}^{n} A_i X_i}{\sum_{i=1}^{n} A_i}$$

Where $A_i$ is the area of each curve, $X_i$ is the centres of the curves and $n$ is the number of output classes. $\chi$ is the overall output.

Fuzzy Logic proves a very wide open system which can be used in a variety of applications, from calculating the fare for a taxi which is based on distance and time of day, or safety critical systems which may monitor the heat and power draw of an electrical system, determining if the system requires any external factors to reduce risk. In basic terms, Fuzzy Logic is used when it is not possible to hard-code values directly into a system, as a particular value could vary over time, or be classified in more than one way. The dynamic nature of this algorithm can be confusing at first instance when one is more familiar to simple Boolean logic, however the possibilities are endless.

This paper will attempt to provide a complete system, implemented in Python, which will calculate the output Fuzzy Logic value for a particular rule base, as specified in the assessment criteria.

# 2 System Design

## 2.1 File Parser

The file parser will be the first part of the application to run, and is responsible for reading the input rule base file, and converting it into the various different parts which are required for the remainder of the system to function correctly. The format of the input file uses empty lines in order to separate the file into the following sections:

- RuleBaseName - The name of the particular rule base.
- Rules - An array of strings representing each individual rule in the rule base.
- Curves - The membership curves of the rule base system.
- RealWorld - The real world values which will used in order to run the rule base system on.

## 2.2 CurveClass

The curve class is used to store the; a, b, alpha and beta value of the membership categories. The class contains a membershipOf method which calculates, on a scale of 0-1, the membership of a certain value on this curve. This is used to find out the degree to which a real world value fits into a particular curve.

## 2.3 Rule Parser

The rule parser handles the parsing of the text based rules, and converts the rule into a format which can be handled by the remainder of the system. The parser used Regular Expressions to separate the rule into its various different parts. These parts, which are all handled in a directory, are:

**Algorithm 1:** File Parser

```
Data: .txt File to be parsed
Result: Rule base parsed into rules, membership curves and real world values
begin
    Read Text File
    Split Text File at Empty Line Breaks
    Process Each Section of the File
    RuleBaseName ← First Section
    while not at end of second section do
        Array of Rules ← This Rule
    while not at the last section of the file do
        read current section as a membership group
        GroupName ← First Line in Section
        while no at the last line in the section do
            Create membership curve for the current membership value
            Array of Membership Curves ← This Membership Curve
        Array of MembershipGroups ← Current Membership Groups
    RealWorldValues ← Final Section of File
```

**Algorithm 2:** Membership Function

```
Data: a, b, α, β, Value
Result: Membership Value from Curve
if Value ≤ a - α then
    return 0
else if Value ∈ [ a - α , a ] then
    return (Value - a + α) / α
else if Value ∈ [a, b] then
    return 1
else if Value ∈ [b, b + β ] then
    return (b + β - Value) / β
else if Value ≥ b + β then
    return 0
else
    return False
```

- RuleID - The numeric ID of the rule.
- Conditions - An array of Variable/Value pairs which make up the rule.
- Conjunction - AND/OR/Null, depending of the makeup of the rule.
- Output - The real world output of this rule.

**Algorithm 3:** Rule Parsing

**Data**: Rule in Text Format
**Result**: Dictionary containing ID, Conditions, Conjunction and Output

```
1 begin
2     Find Matches with Text Rule against Regular Expression as matches
3     if matches then
4         RuleID ← matches.RuleID
5         if matches.conjunction then
6             Conjunction ← matches.conjunction
7         forall the matches.condition do
8             Conditions[] ← This Condition
9         Output ← matches.outputValue
```

### 2.4 RuleFiring

This function in the system will examine the firing of the rules, by calculate the value of each variable throughout its membership curves by using the real world values. Depending on the conjunction of the rule, the system will either find the minimum value (for AND) or the maximum value (for OR). This provides, for each class of output, a membership value. This is expressed as an array.

### 2.5 DeFuzzification

The De-fuzzification of the system calculates the final output value based on the values for each of the output classes. The system will use the dilation of the aggregate in order to produce an approximate output for the system. This is done by estimating the area under a certain point on the output curve. The final, de-fuzzified value is then returned to the user as a float.

## 3 Part Implimentation

### 3.1 File Parser

The partial implementation of the Rule Parser can be found in the 1.*FileParser.py* file to be run from Python. The script will ask for a filename to be entered, which should be a plain text file with a file extension.

When the script is run, a Python dictionary will be printed to screen, which contains elements with the Rules, Curves, RulebaseName and RealWorldValues. Example outputs for the input files *example.txt* and *example2.txt* can be found in the 1.*FileParserDemo* directory.

**Algorithm 4:** RuleFiring

**Data**: List of All Rules
**Result**: Final Value for Each Output Class

```
1  begin
2      Initialization
3      Create array of output classes as Outputs. Each element should be an empty list.
4      forall the Rules do
5          Find conditions in rule
6          Find operation in rule (and—or)
7          Values ← Values of each of the conditions of the rule, as computed with algorithm 3.
           if Operation is AND then
8              RuleValue ← MIN(Values)
9          else
10             RuleValue ← MAX(Values)
11         Add the RuleValue to the array of Outputs against the output class of this rule.
12     forall the Output Classes do
13         if Output Class Values is Array then
14             FinalOutputs.Class ← MAX(OutputValues)
15         else
16             FinalOutputs.Class ← OutputValue
17     return FinalOutputs
```

**Algorithm 5:** DeFuzzification

**Data**: List of Output Categories with Associated Values
**Result**: Final Output of Defuzzified Value

```
1  begin
2      Initialization
3      forall the Output Values do
4          if Output Value is not 0.0 then
5              Calculate Approximate Area
6              Base ← b + β − a − α
7              Area ← 0.5 ∗ Base ∗ Value
8              Center ← Base/2
9              A_i ← Area
10             X_i ← Center
11     FinalValue ← (Sum of Areas * Centers) / (Sum of Areas)
12     return FinalValue
```

| $a$ | $b$ | $\alpha$ | $\beta$ | $Input$ | $Output$ |
|---|---|---|---|---|---|
| 50 | 50 | 20 | 20 | 45 | 0.75 |
| 50 | 50 | 20 | 20 | 30 | 0.0 |
| 50 | 50 | 20 | 20 | 30 | 1.0 |
| 100 | 100 | 50 | 50 | 20 | 0.0 |
| 100 | 100 | 50 | 50 | 60 | 0.2 |
| 100 | 100 | 50 | 50 | 80 | 0.6 |

**Table 1:** Example values computed using the Partial Implementation of CurveClass.

## 3.2 CurvesClass

The partial implementation of the Rule Parser can be found in the 2.*CurveClass.py* file to be run from Python. The script will ask for the $a$, $b$, $\alpha$, $\beta$ and $input$ for the curve to be generated. The membership value for the $input$ will be generated using the function described in Algorithm 2 above. Example vales are shown in Table 1.

## 3.3 Rule Parser

The partial implementation of the Rule Parser can be found in the 3.*RuleParser.py* file to be run from Python. When the script loads, it will ask you to enter in a rule to be parsed. This rule should follow the pattern as set out for this assessment, if the rule can not be parsed, the system will return *This rule does not compute*.

For example, when running the script on the rule: **Rule 1 If the driving is good and the journey time is short then the tip will be big**, the system will return a Python dictionary as highlighted in Figure 2.

Further examples of the outputs from the system can be found in the 3.*RuleParserDemo.txt* file.

```
The Rule has been parsed as.
{'Output': {'tip': 'big'}, 'Conjunction': 'and', 'Conditions': {'journey_time': 'short', 'driving': 'good'}, 'ID': '1'}
```

**Figure 2:** Example output computed using the Partial Implementation of the Rule Parser.

### 3.4 RuleFiring

In order to complete a partial implementation of the RuleFiring subsystem, it is necessary that the previous three parts are combined together. This is required as we need a way to hold rules and curves before it is possible to examine the Rule Firing.

The script will ask for a filename to be entered, which should be a plain text file with a file extension. The rules, curves and real world values from this file will be used to complete the rule firing of the rule base. An example, when run on the file *example.txt* (which was provided through the assessment) is shown in Figure 3.

```
Please enter the rule file name.
Loading the currentRulebase Rule Base
{'reduce': 0.7, 'same': 0.3333333333333333}
```

**Figure 3:** Example output computed using the Partial Implementation of the RuleFiring System.

Further examples of the outputs from the system can be found in the 4.*RuleFiring* directory.

### 3.5 DeFuzzification

As with the above, in order to implement the DeFuzzification sub system, it is necessary to use the previous *RuleFiring* script as a base to work from, as it is not possible to compute the DeFuzzified value without this.

The script will ask for a filename to be entered, which should be a plain text file with a file extension. The rules, curves and real world values from this file will be used to complete the final de-fuzzified value for the real world values, in the rule base. An example, when run on the file *example.txt* (which was provided through the assessment) is shown in Figure 4.

```
Please enter the rule file name.
Loading the tippingRulebase Rule Base
The Answer is...
105.55555555555556
```

**Figure 4:** Example output computed using the Partial Implementation of the De-Fuzzification System.

## 4 Overall System Production

The full implementation of the system is essentially exactly the same as the partial implementation of the DeFuzzification system. The only changes which have been made at this point are some general housekeeping for the Python scripts to ensure that they run in the most optimal way possible.

The first test that was completed with the system is the tippingRulebase which was provided as part of this assessment, and is included in the file example.txt. It was also necessary to prepare a

| RuleBase File | Known Output | Given Output (LHS) | Adjusted Output |
|---|---|---|---|
| example.txt | $\approx 108$ | $\approx 106$ | $\approx 10$ |
| example2.txt | $\approx 69$ | $\approx$ -133 | $\approx 67$ |
| example3.txt | $\approx 2.1$ | $\approx 2.8$ | $\approx 2.8$ |

**Table 2:** Adjusted output values for three test rule bases.

number of other rule bases which could be used to evaluate the system. These are *example*2*.txt* and *example*3*.txt*.

All three of these rule bases contain real world values, and have known answers. The known answers have been compared to the values given from the complete Fuzzy Logic system in table 2.

The implementation of the system calculates the value from the Left hand side point of the graph, which does not always equal zero. The adjusted value gives the true value, on the $x$ axis of the graph at which the output value corresponds to. One of the main reasons for the difference between the expected and adjusted outputs from the system are the methods which are used to DeFuzzyfy the output value. The expected output is done using the standard method as using throughout the course, however the actual output is calculated using the dilatation of the aggregate, which gives more of an estimation as to the output value.

Comparing each of the expected values, and the produced values, it does entail that the system is completing the defuzzification process to within a small degree of change, and therefore provides a complete system for parsing Fuzzy Rule Base Files.

Further analysis of the completeness and correctness of the system is provided in the next section of this paper.

# 5 Testing

In order to provide a better test of the output values of the rule base, based on the real world input values, the tippingRuleBase (file *example.txt*), has been run a number of times, varying the real world input variables in steps each time. The results are shown in table 3. These tests were all carried out automatically through the Python system, and exported to a CSV file which is provided.

The data which has been produced through this helps to see just how the system can vary throughout its lifetime, and that the output values are far from standard Boolean logic. There are a number of values which are capped at the top end of the output value 100.0, however there is also a clear understanding how each change to input values from the real world can create subtle changes in the output, in this case the tip that the taxi driver will receive.

## 5.1 System Limitations

The system has a number of limitations, which effect the style of rules which can be parsed.

The system has a number of limitations, which effect the style of rules which can be parsed. For example a rule base can only be parsed if it is presented in the format that is specified in the assignment criteria. Other limitations which are posed on the rules base are as follows.

- Only one type of conditional can be used in a rule (AND or OR).
- Class names must have the word *the* prepending them in the rules.
- Rules must begin with Rule ID
- All inputs and outputs must be numeric, however they can be floats.
- The output can only be parsed on one graph. There cannot be two separate outputs being handled in the same rule base.
- The rule base can exhibit undesirable behaviour when using negative numbers.
- Any mistake in the rule file will result in the system terminating without a response. There is no mechanism to attempt to correct issues.
- There is no visualisation which can explain the calculations, and graphs.

As future work, the above limitations would be mitigated and resolutions sought for them.

## 5.2 Conclusion

The system as it stands provides the end user with a reasonably well calculated output value for a Fuzzy Logic rule base. The value which is calculated falls within a small margin of error from the expected output value, however given the fuzzy nature of Fuzzy Logic, it is well within permissible bounds which constitute a complete and successful system.

| Driving Style | Journey Time | Output |
|---|---|---|
| 1 | 1 | 100.0 |
| 11 | 1 | 100.0 |
| 21 | 1 | 100.0 |
| 31 | 1 | 100.0 |
| 41 | 1 | 100.0 |
| 51 | 1 | 100.0 |
| 61 | 1 | 102.63157894736842 |
| 71 | 1 | 118.96551724137932 |
| 81 | 1 | 123.6842105263158 |
| 91 | 1 | 123.6842105263158 |
| 1 | 6 | 100.0 |
| 11 | 6 | 100.0 |
| 21 | 6 | 100.0 |
| 31 | 6 | 100.0 |
| 41 | 6 | 100.0 |
| 51 | 6 | 100.0 |
| 61 | 6 | 105.0 |
| 71 | 6 | 121.05263157894737 |
| 81 | 6 | 114.28571428571429 |
| 91 | 6 | 114.28571428571429 |
| 1 | 11 | 94.44444444444444 |
| 11 | 11 | 94.44444444444444 |
| 21 | 11 | 94.44444444444444 |
| 31 | 11 | 94.44444444444444 |
| 41 | 11 | 94.44444444444444 |
| 51 | 11 | 100.0 |
| 61 | 11 | 100.0 |
| 71 | 11 | 100.0 |
| 81 | 11 | 100.0 |
| 91 | 11 | 100.0 |
| 1 | 16 | 50.0 |
| 11 | 16 | 50.0 |
| 21 | 16 | 50.0 |
| 31 | 16 | 53.84615384615385 |
| 41 | 16 | 77.50000000000001 |
| 51 | 16 | 100.0 |
| 61 | 16 | 100.0 |
| 71 | 16 | 100.0 |
| 81 | 16 | 100.0 |
| 91 | 16 | 100.0 |

**Table 3:** Example Varied Values for tippingRuleBase. Values are raw from Python.