Computational Intelligence

Assignment 1:

Generation of fuzzy rules from data

MUSI - Máster Universitario en Sistemas Inteligentes

Group members:

López Muñoz, Iván
Martí Picó, Pedro
Noboa Rivera, Jason
Suárez Campione, David

Table of content

Table of figures

1. Introduction

Fuzzy logic has emerged as a powerful framework for handling uncertainty and imprecision in decision-making and control systems. Fuzzy logic, introduced by Lotfi A. Zadeh in 1965 [1], is a mathematical approach that models and simulates human reasoning by incorporating linguistic terms and degrees of membership, allowing for the representation of vague and ambiguous information. This paradigm has found applications in diverse fields such as artificial intelligence, control systems, pattern recognition, and decision support.

The foundation of fuzzy logic lies in the use of linguistic terms, which provide a natural way to describe and manipulate imprecise information. These linguistic terms, often defined using fuzzy sets, allow us to express concepts like "very hot," "quite cold," or "medium speed" with a high degree of flexibility. This linguistic approach enhances the interpretability of systems and bridges the gap between human perception and computational methods [2].

Linguistic terms, as a fundamental aspect of fuzzy logic, enable us to convert qualitative, subjective judgments into quantitative, computational models. This linguistic perspective has become integral in the development of fuzzy systems, where membership functions, operators, and fuzzy rules are constructed to capture the nuances of human language and reasoning.

2. Objective

The objective of this assignment is to implement the algorithm for generating fuzzy rules from numerical data proposed in this paper [3] in a Python environment. To achieve this, we will follow the following sections of the paper, implementing step by step the discussed models and linguistic norms. To ensure proper implementation, we will perform some of the examples provided in the paper.

Once the algorithm is developed, we will assess its performance by proposing a problem that can be modeled using a fuzzy rule-based system.

Finally, we will analyze the algorithm's behavior based on the obtained results, depending on the parameter values.

3. Algorithm implementation

The algorithm presented in the paper [3] is a potent heuristic approach for automatically generating fuzzy if-then rules from numerical data. The primary advantage of this algorithm lies in its simplicity, as it avoids time-consuming iterative learning procedures and intricate rule generation mechanisms. Additionally, it proposes a linguistic representation technique to transform fuzzy if-then rules with consequent real numbers into linguistic rules.

3.1 Functions

In this section, we will present the primary functions employed for the computation of the formulas as outlined in the paper [3].

3.1.1 Membership Function

In the context of a database, there are two fundamental factors that play a pivotal role in its functionality: a fuzzy partition of the input space and the membership functions associated with antecedent fuzzy sets. The main paper [3] focuses on an assumption where each input variable 'x' is evenly divided into 'Ki' fuzzy sets, denoted as 'Ai1, Ai2, ..., AiK,' where 'i' ranges from 1 to 'n.' This division results in the creation of n-dimensional input space, which is subsequently subdivided into 'K1 * K2 * ... * Kn' fuzzy subspaces, represented as (A1j, A2j, ..., Anj), where 'j' can take values from 1 to 'Ki,' and 'i' can range from 1 to 'n.'

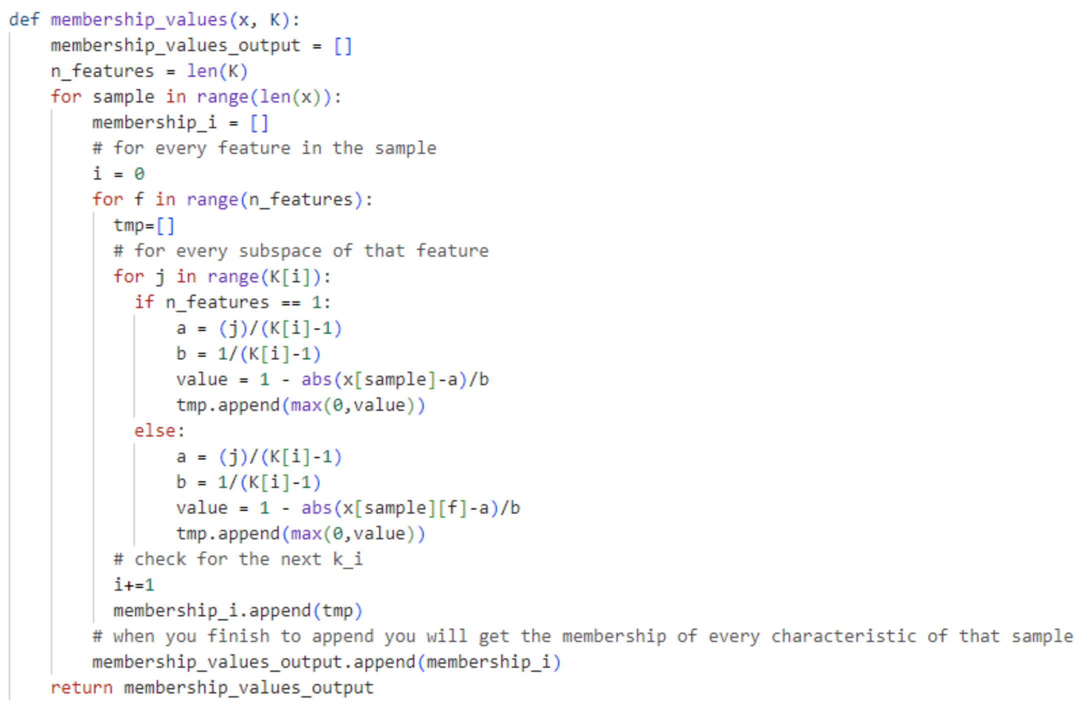
While it's worth noting that various types of membership functions, such as triangle-shaped, trapezoid-shaped, and bell-shaped, can be utilized for antecedent fuzzy sets, the paper specifically employs symmetric triangle-shaped fuzzy sets, denoted as 'Aij,' [3] with the following predefined membership functions:

$$\mu_{ij}(x) = max\{1 - |x - a_{ji}^{Ki}| / b^{Ki}, 0\}, \qquad ji = 1, 2,...., Ki \tag{1}$$

$$a_{ji}^{Ki} = (ji - 1)/(Ki - 1), \qquad ji = 1, 2,...., Ki \tag{2}$$

$$b^{Ki} = 1/(Ki - 1) \tag{3}$$

We have implemented this membership function to get the membership value of every characteristic of that input sample, as shown in Figure 1:



```python
def membership_values(x, K):
    membership_values_output = []
    n_features = len(K)
    for sample in range(len(x)):
        membership_i = []
        # for every feature in the sample
        i = 0
        for f in range(n_features):
          tmp=[]
          # for every subspace of that feature
          for j in range(K[i]):
            if n_features == 1:
                a = (j)/(K[i]-1)
                b = 1/(K[i]-1)
                value = 1 - abs(x[sample]-a)/b
                tmp.append(max(0,value))
            else:
                a = (j)/(K[i]-1)
                b = 1/(K[i]-1)
                value = 1 - abs(x[sample][f]-a)/b
                tmp.append(max(0,value))
          # check for the next k_i
          i+=1
          membership_i.append(tmp)
        # when you finish to append you will get the membership of every characteristic of that sample
        membership_values_output.append(membership_i)
    return membership_values_output
```

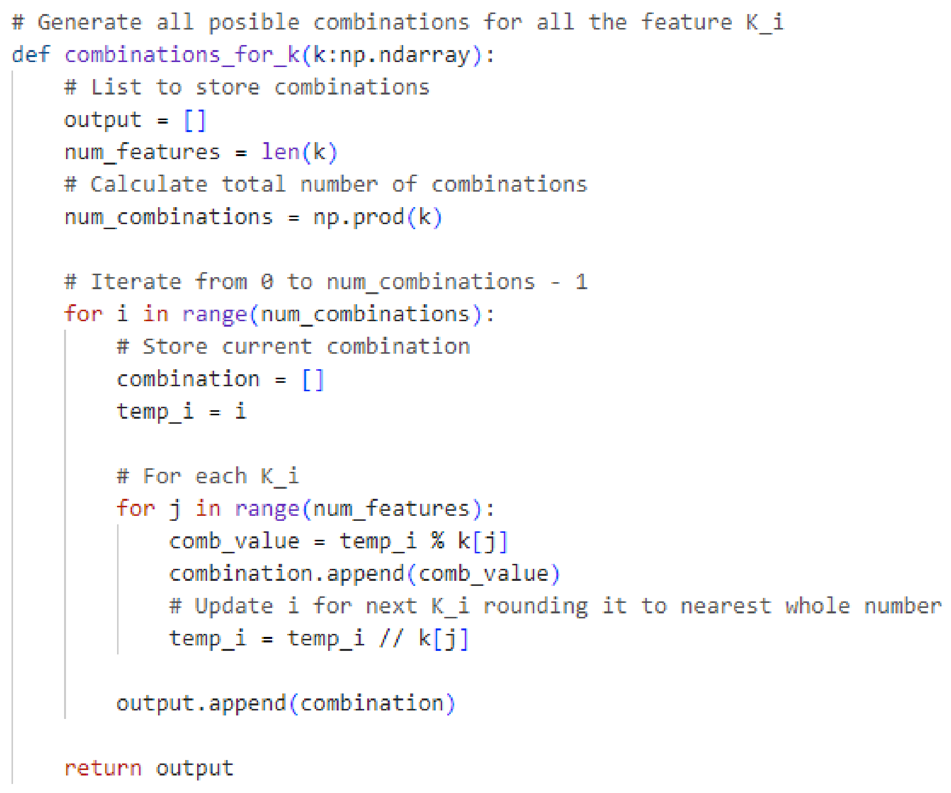
Figure 1: Membership value function

3.1.2 Combinations of linguistic labels

In each of the fuzzy subspaces as defined in the function before represented as (A1j, A2j, ..., Anj), a single fuzzy if-then rule is generated. This means that the total number of fuzzy if-then rules in the fuzzy rule-based system is equal to the number of fuzzy subspaces.

For instance, in the scenario of a two-input single-output fuzzy rule-based system, where we have K1 = 5 and K2 = 5, a total of 25 fuzzy if-then rules are generated. This implies that there will be 25 distinct rules governing the system's behavior, each associated with a specific region within the input space.

We have done a function that is capable of generate and store all possible combinations for all Ki features, as shown in Figure 2:



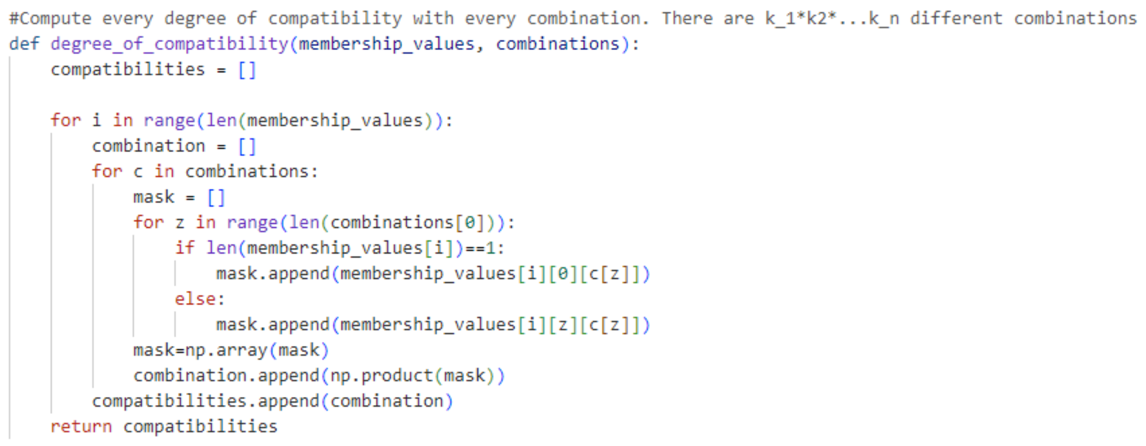
```python
# Generate all posible combinations for all the feature K_i
def combinations_for_k(k:np.ndarray):
    # List to store combinations
    output = []
    num_features = len(k)
    # Calculate total number of combinations
    num_combinations = np.prod(k)

    # Iterate from 0 to num_combinations - 1
    for i in range(num_combinations):
        # Store current combination
        combination = []
        temp_i = i

        # For each K_i
        for j in range(num_features):
            comb_value = temp_i % k[j]
            combination.append(comb_value)
            # Update i for next K_i rounding it to nearest whole number
            temp_i = temp_i // k[j]

        output.append(combination)

    return output
```

Figure 2: Linguistic labels combination function

3.1.3 Degree of compatibility

The degree of compatibility (3) measures the extent to which an element belongs to a particular fuzzy set. In this case, as shown in the function developed in Figure 3, we are computing the degree of compatibility of the input vector, which are the membership values for each linguistic label in the corresponding feature, and all possible combinations of linguistic labels, this function generates products for the different possible spaces for every sample.

$$\mu_{ji} \dots \mu_{jn}(x_p). \;=\; \mu 1_{j1}(x_{p1}) \cdot \dots \cdot \mu n_{jn}(x_{pn}) \qquad (3)$$



```python
#Compute every degree of compatibility with every combination. There are k_1*k2*...k_n different combinations
def degree_of_compatibility(membership_values, combinations):
    compatibilities = []

    for i in range(len(membership_values)):
        combination = []
        for c in combinations:
            mask = []
            for z in range(len(combinations[0])):
                if len(membership_values[i])==1:
                    mask.append(membership_values[i][0][c[z]])
                else:
                    mask.append(membership_values[i][z][c[z]])
            mask=np.array(mask)
            combination.append(np.product(mask))
        compatibilities.append(combination)
    return compatibilities
```

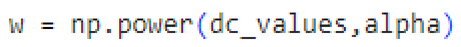
Figure 3: Degree of compatibility function

3.1.4 Weight of input-output pairs

Weight is based on the previously calculated degree of compatibility. By applying a power function with a user-selected alpha, the value obtained determines the extent of concentration or dilation in the resulting function. This approach allows us to fine-tune the impact of different input-output pairs on the system's behavior.

As it is not a very complicated operation (4), we have not made a function to calculate the weight. However, with a simple line, as shown in Figure 4, within the predict function, that we will explain later in subsection 3.1.6, we have managed to obtain the weight values.

$$W_{j1...jn}(x_p) = \{\mu_{ji} ... \mu_{jn}(x_p)\}^{\alpha} \tag{4}$$



```python
w = np.power(dc_values,alpha)
```

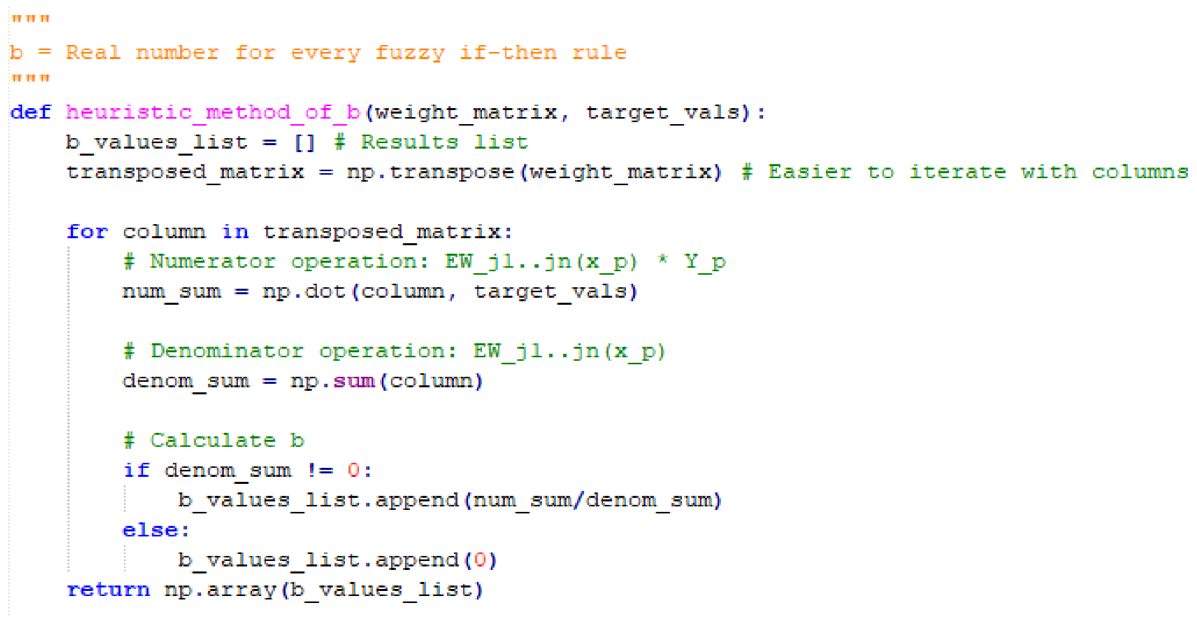
Figure 4: Calculation of Weight

3.1.5 Heuristic method

Once we have calculated the weight of each input pair, we are able to calculate, using the heuristic method developed in the paper [3], the consequent real number value of every fuzzy if-then rule, defined by the next equation (5):

$$b_{j1...jn} = \sum_{p=1}^{m} W_{j1...jn}(x_p) \cdot y_p \,/\, \sum_{p=1}^{m} W_{j1...jn}(x_p) \tag{5}$$

Our function, as shown in Figure 5, has for inputs the weight values calculated previously and the final results of our problem, and the output will be an array with the calculation of the value of each rule as defined by the equation (5).



```
"""
b = Real number for every fuzzy if-then rule
"""
def heuristic_method_of_b(weight_matrix, target_vals):
    b_values_list = [] # Results list
    transposed_matrix = np.transpose(weight_matrix) # Easier to iterate with columns

    for column in transposed_matrix:
        # Numerator operation: EW_j1..jn(x_p) * Y_p
        num_sum = np.dot(column, target_vals)

        # Denominator operation: EW_j1..jn(x_p)
        denom_sum = np.sum(column)

        # Calculate b
        if denom_sum != 0:
            b_values_list.append(num_sum/denom_sum)
        else:
            b_values_list.append(0)
    return np.array(b_values_list)
```

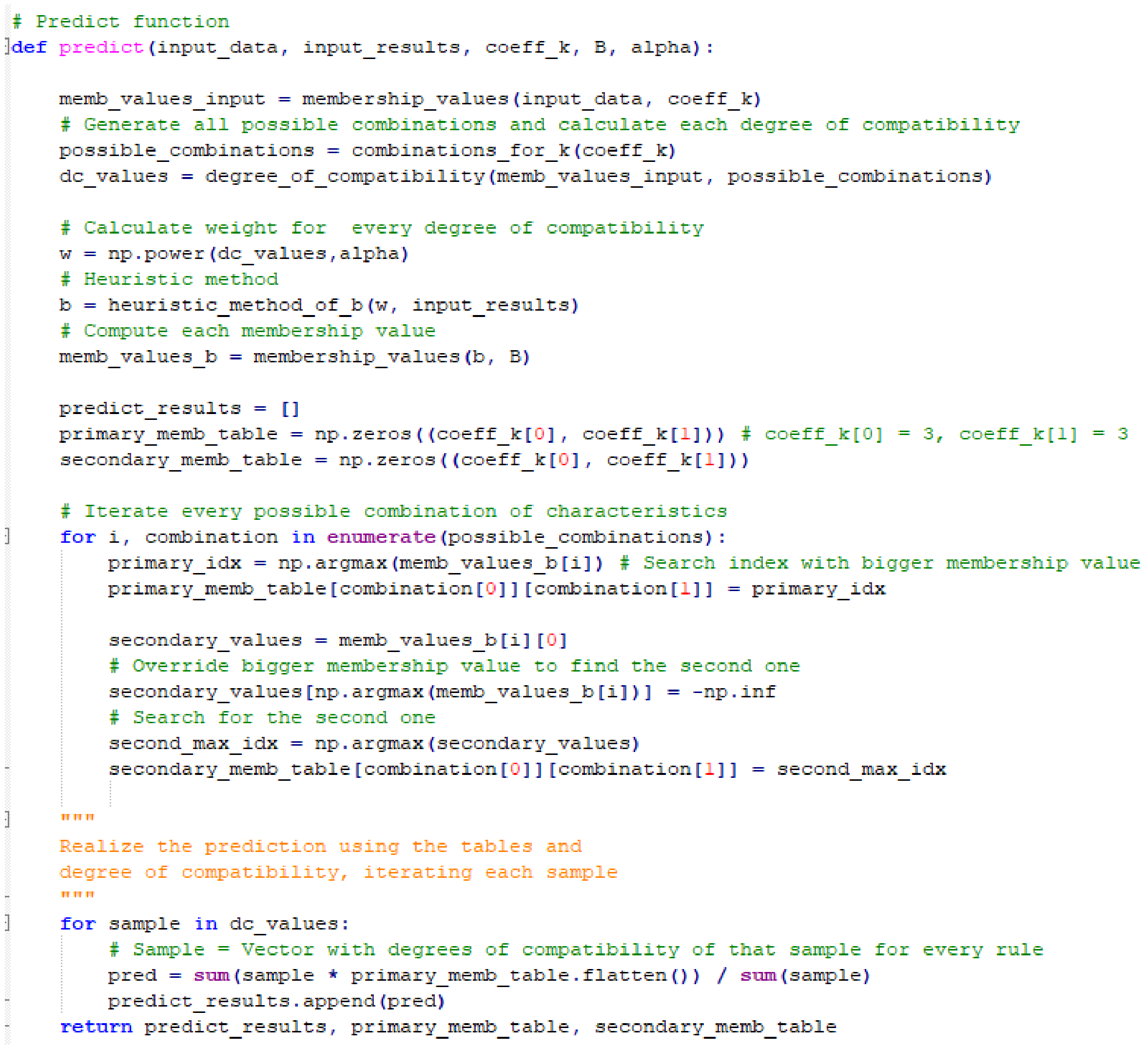
Figure 5: Heuristic method function to calculate b

3.1.6 Prediction

The main objective of this function, once we have calculated the real number value for each fuzzy if-then rule, is to do a linguistic label interpretation for the whole output domain interval.

To do that, the given fuzzy if-then rules have been separated into two linguistic rule tables, in which the membership value of two consequent fuzzy sets that overlaps, the rule with biggest membership value will be in the main linguistic rule table and the lowest one in the secondary rule table.

The prediction function, as shown in Figure 6, uses the different functions described in this document to be able to iterate for every possible combination the membership values to generate the two different rule tables with the dimensions according to the number K. It also provides the calculation of the real number value of every fuzzy if-then rule.



```python
# Predict function
def predict(input_data, input_results, coeff_k, B, alpha):

    memb_values_input = membership_values(input_data, coeff_k)
    # Generate all possible combinations and calculate each degree of compatibility
    possible_combinations = combinations_for_k(coeff_k)
    dc_values = degree_of_compatibility(memb_values_input, possible_combinations)

    # Calculate weight for  every degree of compatibility
    w = np.power(dc_values,alpha)
    # Heuristic method
    b = heuristic_method_of_b(w, input_results)
    # Compute each membership value
    memb_values_b = membership_values(b, B)

    predict_results = []
    primary_memb_table = np.zeros((coeff_k[0], coeff_k[1])) # coeff_k[0] = 3, coeff_k[1] = 3
    secondary_memb_table = np.zeros((coeff_k[0], coeff_k[1]))

    # Iterate every possible combination of characteristics
    for i, combination in enumerate(possible_combinations):
        primary_idx = np.argmax(memb_values_b[i]) # Search index with bigger membership value
        primary_memb_table[combination[0]][combination[1]] = primary_idx

        secondary_values = memb_values_b[i][0]
        # Override bigger membership value to find the second one
        secondary_values[np.argmax(memb_values_b[i])] = -np.inf
        # Search for the second one
        second_max_idx = np.argmax(secondary_values)
        secondary_memb_table[combination[0]][combination[1]] = second_max_idx

    """
    Realize the prediction using the tables and
    degree of compatibility, iterating each sample
    """
    for sample in dc_values:
        # Sample = Vector with degrees of compatibility of that sample for every rule
        pred = sum(sample * primary_memb_table.flatten()) / sum(sample)
        predict_results.append(pred)
    return predict_results, primary_memb_table, secondary_memb_table
```

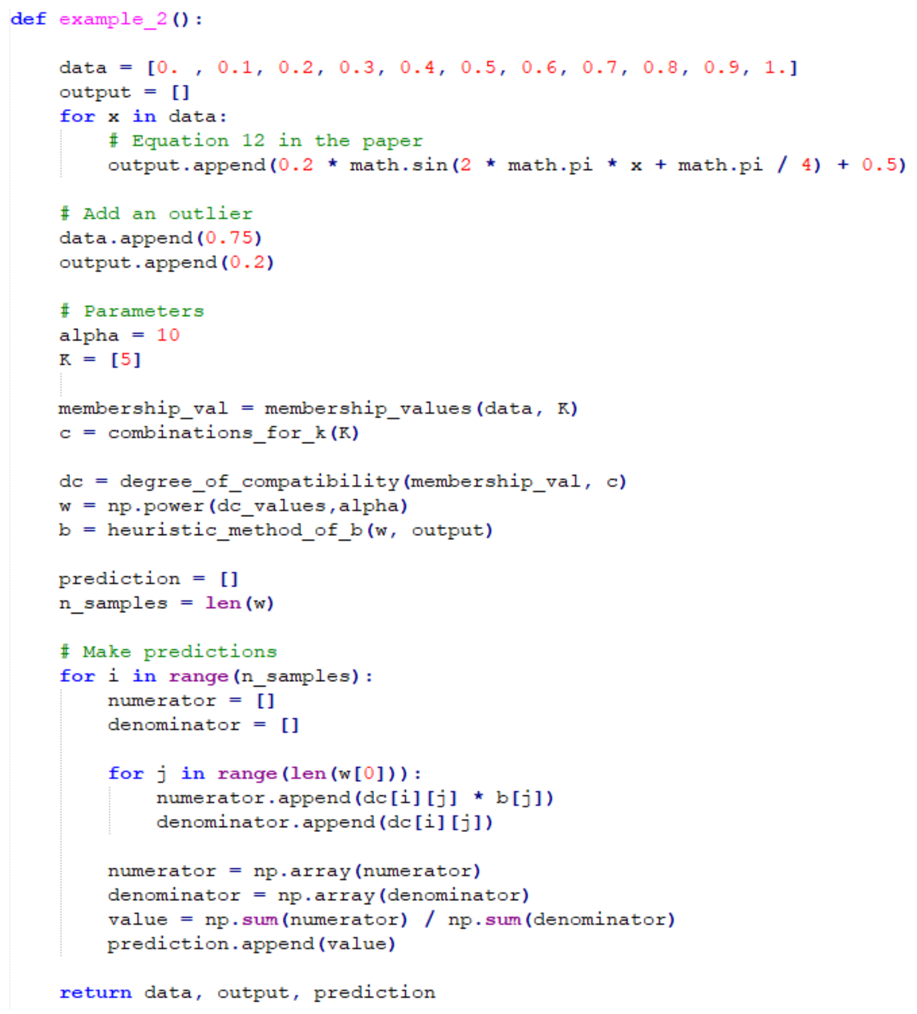
Figure 6: Prediction function for linguistic interpretation

3.2 Examples

In this section we are going to do different examples discussed in the main paper [3] to verify that the implemented algorithm works correctly and then be able to apply it to a problem proposed by us.

3.2.1 Example 2

This example performs a test using a synthetic dataset. The goal is to illustrate how the defined functions can be used to make predictions and evaluate the performance of the fuzzy system.



```python
def example_2():

    data = [0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.]
    output = []
    for x in data:
        # Equation 12 in the paper
        output.append(0.2 * math.sin(2 * math.pi * x + math.pi / 4) + 0.5)

    # Add an outlier
    data.append(0.75)
    output.append(0.2)

    # Parameters
    alpha = 10
    K = [5]

    membership_val = membership_values(data, K)
    c = combinations_for_k(K)

    dc = degree_of_compatibility(membership_val, c)
    w = np.power(dc_values,alpha)
    b = heuristic_method_of_b(w, output)

    prediction = []
    n_samples = len(w)

    # Make predictions
    for i in range(n_samples):
        numerator = []
        denominator = []

        for j in range(len(w[0])):
            numerator.append(dc[i][j] * b[j])
            denominator.append(dc[i][j])

        numerator = np.array(numerator)
        denominator = np.array(denominator)
        value = np.sum(numerator) / np.sum(denominator)
        prediction.append(value)

    return data, output, prediction
```

Figure 7: Example 2 function

This example has parameters alpha = 10 and K = [5]. The membership values are calculated with the input of data and K (Figure 1), also combinations_for_k (Figure 2) are calculated with input of K.

The degree_of_compatibility (Figure 3) uses membership values and c as input, once we have the information we calculate the weights (Figure 4) and the heuristic method of b (Figure 5) with the weights and the output as inputs

Once all of this is calculated, compute the predictions and return data, output and the prediction.

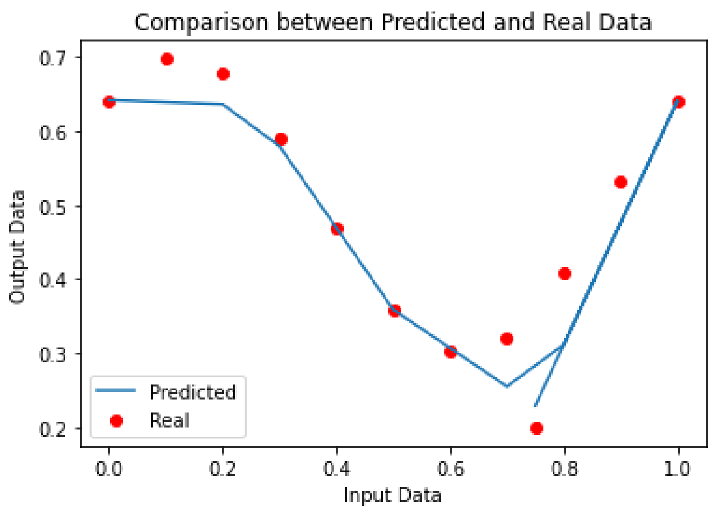


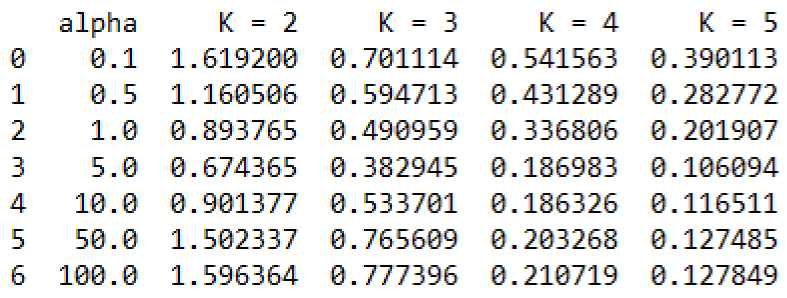
Figure 8: Comparison between Predicted and Real Data

Figure 8 shows the quality of predictions made by the fuzzy inference system compared to the actual values, which is crucial for understanding the effectiveness of the proposed approach. As we can see the predicted data is quite similar to the real data.

3.2.2 Example 4.2

This example has the data from the paper "A Fuzzy-Logic-Based Approach to Qualitative Modeling". Y is calculated with the data of x1 and x2, this data has been normalized and uses a MinMaxScaler function with K = [5,5], B = [5] and alpha = 5 as parameters.

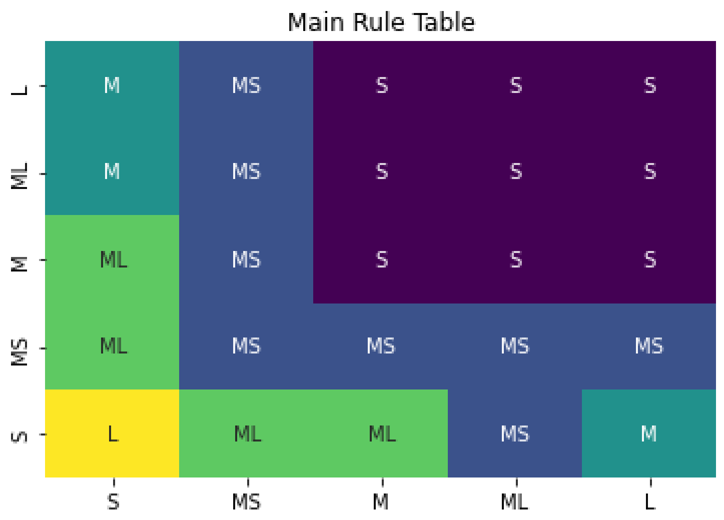
The performanceTable compares the model's performance based on different parameter configurations and select the configuration that best suits the needs of the problem.



```
   alpha     K = 2     K = 3     K = 4     K = 5
0    0.1  1.619200  0.701114  0.541563  0.390113
1    0.5  1.160506  0.594713  0.431289  0.282772
2    1.0  0.893765  0.490959  0.336806  0.201907
3    5.0  0.674365  0.382945  0.186983  0.106094
4   10.0  0.901377  0.533701  0.186326  0.116511
5   50.0  1.502337  0.765609  0.203268  0.127485
6  100.0  1.596364  0.777396  0.210719  0.127849
```

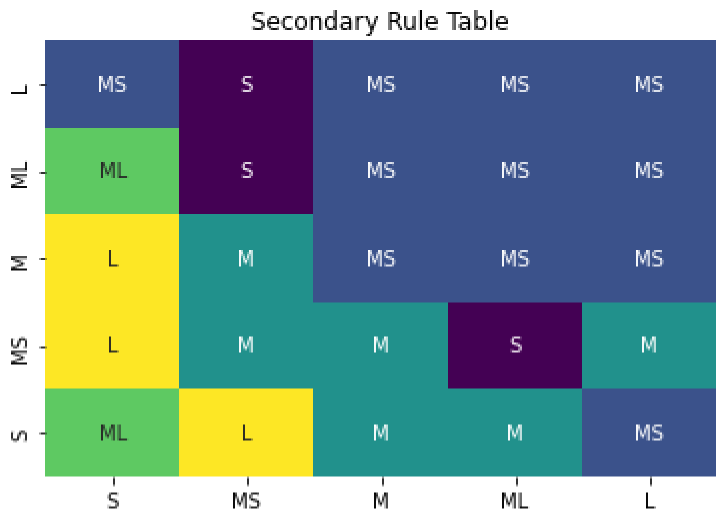
Figure 9: Performance Table

This function represents two tables which are the Main Rule Table and the Secondary table. The main rule table specifies the mapping from input combination to output labels based on the rules of the system, the secondary considers alternative output labels when the primary is already chosen.


Main Rule Table

	S	MS	M	ML	L
L	M	MS	S	S	S
ML	M	MS	S	S	S
M	ML	MS	S	S	S
MS	ML	MS	MS	MS	MS
S	L	ML	ML	MS	M

Figure 10: Main Rule Table Example 4.2


Secondary Rule Table

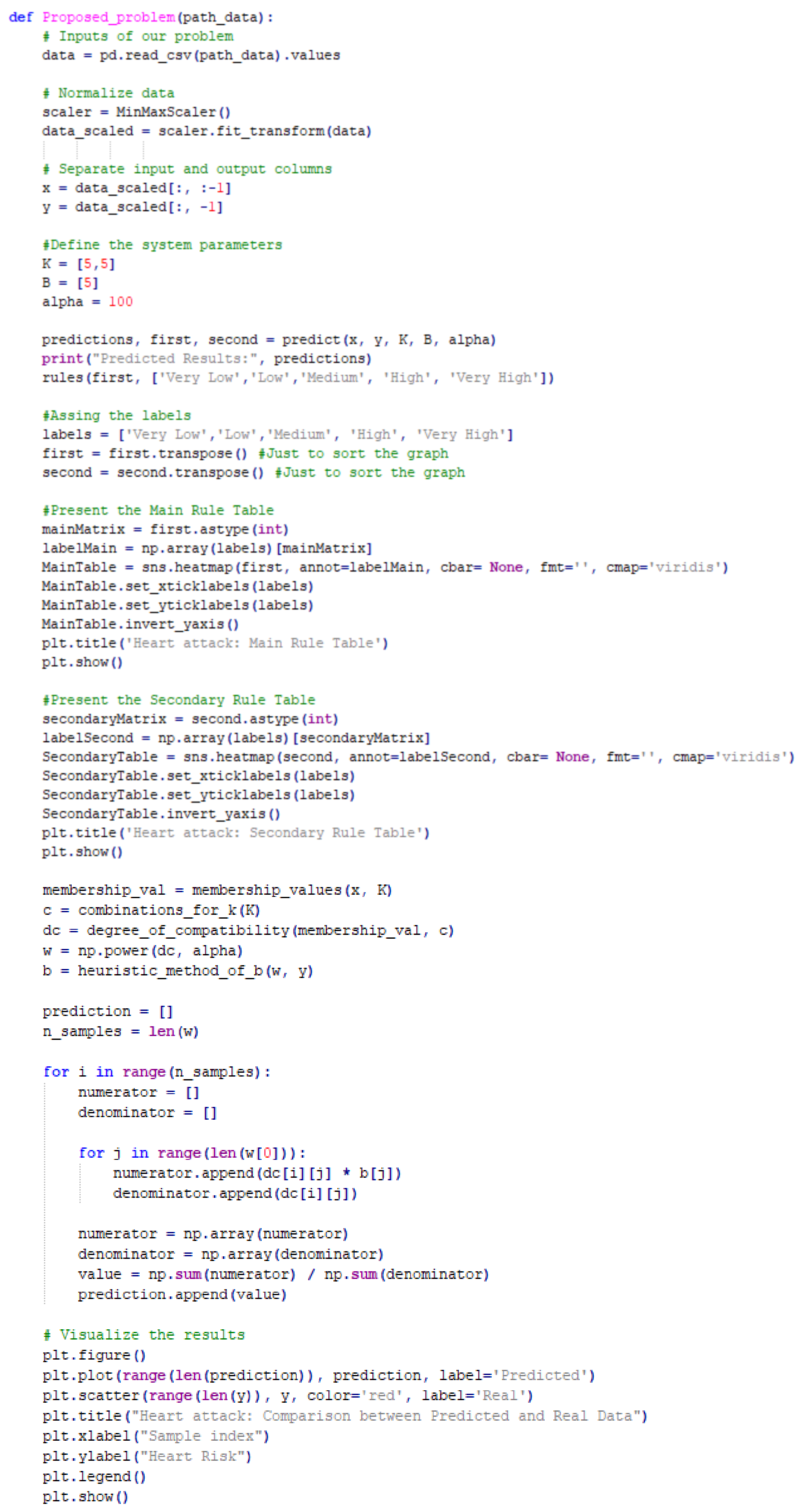
	S	MS	M	ML	L
L	MS	S	MS	MS	MS
ML	ML	S	MS	MS	MS
M	L	M	MS	MS	MS
MS	L	M	M	S	M
S	ML	L	M	M	MS

Figure 11: Secondary Rule Table Example 4.2

As we can see in the figure 10 and 11, the results obtained are exactly the same as the example represented on the paper in section 4.2 [3].

4. Proposed problem

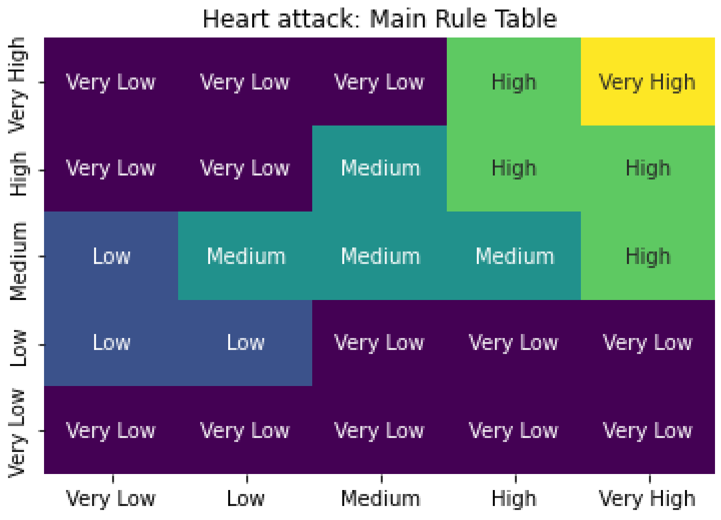
Our proposed problem is about the probability of having a heart problem risk with cholesterol and blood pressure as inputs. A database with the values of the inputs and outputs is provided called (data.csv).



```python
def Proposed_problem(path_data):
    # Inputs of our problem
    data = pd.read_csv(path_data).values

    # Normalize data
    scaler = MinMaxScaler()
    data_scaled = scaler.fit_transform(data)

    # Separate input and output columns
    x = data_scaled[:, :-1]
    y = data_scaled[:, -1]

    #Define the system parameters
    K = [5,5]
    B = [5]
    alpha = 100

    predictions, first, second = predict(x, y, K, B, alpha)
    print("Predicted Results:", predictions)
    rules(first, ['Very Low','Low','Medium', 'High', 'Very High'])

    #Assing the labels
    labels = ['Very Low','Low','Medium', 'High', 'Very High']
    first = first.transpose() #Just to sort the graph
    second = second.transpose() #Just to sort the graph

    #Present the Main Rule Table
    mainMatrix = first.astype(int)
    labelMain = np.array(labels)[mainMatrix]
    MainTable = sns.heatmap(first, annot=labelMain, cbar= None, fmt='', cmap='viridis')
    MainTable.set_xticklabels(labels)
    MainTable.set_yticklabels(labels)
    MainTable.invert_yaxis()
    plt.title('Heart attack: Main Rule Table')
    plt.show()

    #Present the Secondary Rule Table
    secondaryMatrix = second.astype(int)
    labelSecond = np.array(labels)[secondaryMatrix]
    SecondaryTable = sns.heatmap(second, annot=labelSecond, cbar= None, fmt='', cmap='viridis')
    SecondaryTable.set_xticklabels(labels)
    SecondaryTable.set_yticklabels(labels)
    SecondaryTable.invert_yaxis()
    plt.title('Heart attack: Secondary Rule Table')
    plt.show()

    membership_val = membership_values(x, K)
    c = combinations_for_k(K)
    dc = degree_of_compatibility(membership_val, c)
    w = np.power(dc, alpha)
    b = heuristic_method_of_b(w, y)

    prediction = []
    n_samples = len(w)

    for i in range(n_samples):
        numerator = []
        denominator = []

        for j in range(len(w[0])):
            numerator.append(dc[i][j] * b[j])
            denominator.append(dc[i][j])

        numerator = np.array(numerator)
        denominator = np.array(denominator)
        value = np.sum(numerator) / np.sum(denominator)
        prediction.append(value)

    # Visualize the results
    plt.figure()
    plt.plot(range(len(prediction)), prediction, label='Predicted')
    plt.scatter(range(len(y)), y, color='red', label='Real')
    plt.title("Heart attack: Comparison between Predicted and Real Data")
    plt.xlabel("Sample index")
    plt.ylabel("Heart Risk")
    plt.legend()
    plt.show()
```

Figure 12: Proposed Problem Function

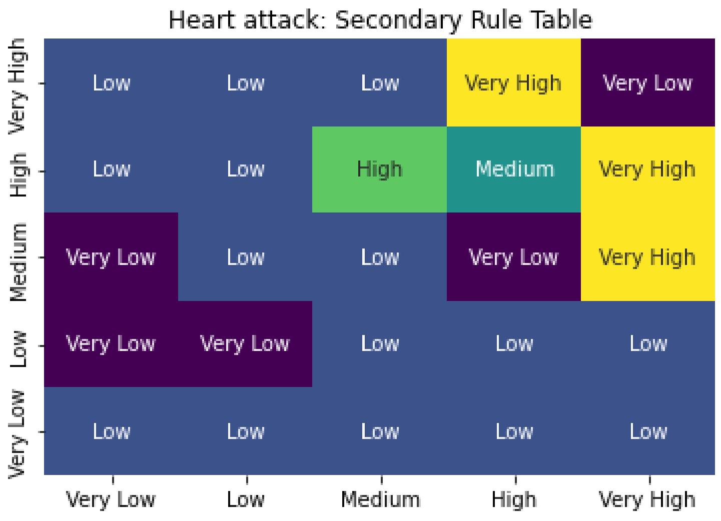
The proposed problem has K = [5,5], B = 5, and alpha = 100 as parameters and uses 5 labels for the inputs and output (Very Low, Low, Medium, High, Very High).



Heart attack: Main Rule Table

	Very Low	Low	Medium	High	Very High
Very High	Very Low	Very Low	Very Low	High	Very High
High	Very Low	Very Low	Medium	High	High
Medium	Low	Medium	Medium	Medium	High
Low	Low	Low	Very Low	Very Low	Very Low
Very Low	Very Low	Very Low	Very Low	Very Low	Very Low

Figure 13: Proposed Problem Main table



Heart attack: Secondary Rule Table

	Very Low	Low	Medium	High	Very High
Very High	Low	Low	Low	Very High	Very Low
High	Low	Low	High	Medium	Very High
Medium	Very Low	Low	Low	Very Low	Very High
Low	Very Low	Very Low	Low	Low	Low
Very Low	Low	Low	Low	Low	Low

Figure 14: Proposed Problem Secondary table

Predict function (Figure 6) is calculated with the values of the dataset and the parameters mentioned. We can see the result of this prediction in the figures 13 and 14.

To visualize the results the functions membership_values, combinations_for_k, degree_of_compatibility, weights and heuristic_method_of_b (Figure 1, 2, 3 ,4 and 5 respectively) are used.

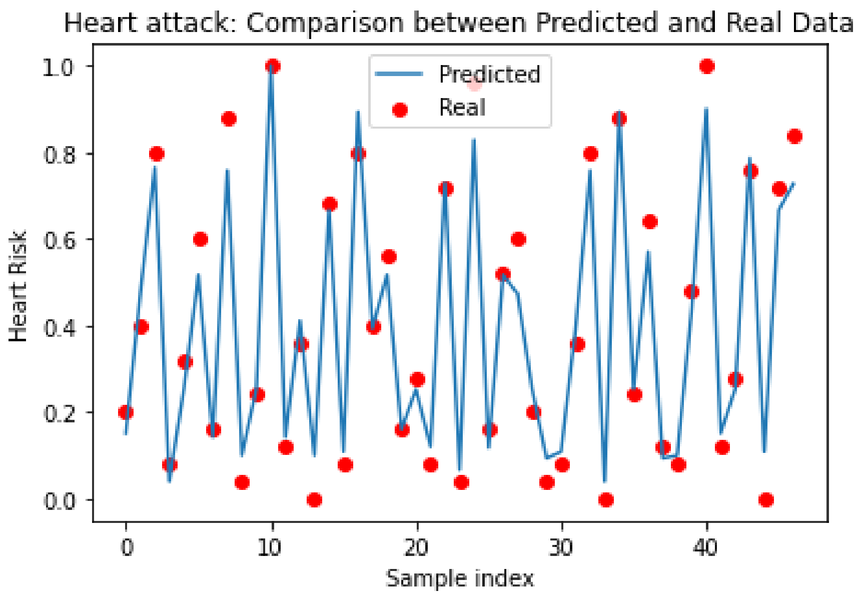


Figure 15: Proposed Problem Comparison Predicted and Real Data

Figure 15 shows the results of this problem comparing our predicted data and the data provided in the dataset (data.csv). It can be seen that there is not much difference between the predicted and the real data.

5. Conclusion

Observing the results of the examples and the proposed problem, we can say that we have achieved the proposed objectives for this task, noting that the proposed algorithm despite being based solely on degrees of membership, statistics and probabilities turns out to be quite accurate when delivering results, as well as its similar ones which use iterative processes to train the model and generate the rules.

We successfully implemented an algorithm for generating fuzzy rules from numerical data. Such an algorithm enables us to model systems using linguistic terms and degrees of membership, which has great potential in various applications.

As shown in Figure 13, the main rule table follows an ascending trend in terms of the risk of suffering a heart attack depending on whether cholesterol and blood pressure are high.

We have detected that if the database does not have logical values, the rules table does not match reality, which can lead to an error in the interpretation of the results and in the development of the algorithm.

It is also worth highlighting the computational speed when calculating all the predictions, since the results and tables are obtained immediately.

It has been interesting to be able to do this task to learn more about the behaviour of fuzzy sets, serving as an introduction for future projects.

6. References

[1] L. A. Zadeh, "Fuzzy Sets," Information and Control, vol. 8, no. 3, pp. 338-353, 1965. DOI: 10.1016/S0019-9958(65)90241-X.

[2] L. A. Zadeh, "The Concept of a Linguistic Variable and its Application to Approximate Reasoning—I," Information Sciences, vol. 8, no. 3, pp. 199-249, 1975. DOI: 10.1016/0020-0255(75)90036-5.

[3] K. Nozaki, H. Ishibuchi, and H. Tanaka, ‘A simple but powerful heuristic method for generating fuzzy rules from numerical data’, Fuzzy Sets and Systems, vol. 86, no. 3, pp. 251–270, 1997.