Fuzzy Socks* : A fuzzy problem solved fuzzily by fuzzy logic using fuzzy rules

11753 - Intel · ligència Computacional

Balandis, Alina Gelabert Gómez, Martí

November 2022

Contents

1 Introduction

Fuzzy logic was created to represent ambiguity and uncertainty in mathematical problems. This includes interpreting linguistic terms such as ”bad” and ”good”. These inputs are interpreted through a fuzzy rule based system. Such systems define the rules that link input variables together and produce the outputs of a system. Such rules are often created by human experts using human knowledge. Nozaki et al. 1995 proposed a heuristic method to generate fuzzy rules from numerical data. This paper implements this method with a new problem and the corresponding data.

1.1 Problem

The problem is predicting the **quality of a sewn garment**. If a seamstress makes a garment, the quality of the finished piece of clothing will depend on two things. First, the **experience** of the seamstress. We decided to encode the experience in a value between 1 and 6. We decided against using years of experience because someone who sews every day for a year is going to have more experience than someone who sews only a few times a year but has done that for three years. Second, the **complexity** of the garment. A garment may have very straightforward instructions and an easy pattern. Or it may have a very complex pattern that uses advanced sewing techniques. This complexity is often encoded in a six-star system, so the seamstress knows how difficult to make the garment is. Therefore, we have encoded the difficulty in a value between 1 and 6. Through these two inputs, we want to predict a crisp value representing the **quality of the garment**. With **fuzzy logics** we should be able to compute what the quality of the finished garment will be, as we have seen previously in the course.

1.2 Fuzzyfication

As in Nozaki et al. 1995, we will also work directly with **numerical crisp values**. This means that fuzzification of the input variables is not required in this model. The idea is to be able to output numerical values from a set of rules generated with the original numerical crisp inputs.

1.3 Structure

This paper will implement the method from Nozaki et al. 1995 with the problem outlined below. To do this, first we will introduce the python implementation of the method proposed by Nozaki et al. Then we will show the dataset we used and explain the application of these functions on our problem. Finally, we will discuss the results.

2 Implementation

2.1 Functions

In this section we will introduce the main functions used to compute the formulas from the paper Nozaki et al. 1995. The functions related to data visualization will not be explained in the documentation but will be present in the code files with their respective comments.

2.1.1 Membership Function

The membership functions in Listing 1 is extracted from the paper, using the values computed from:

$$\mu_{ij_i}(x) = max\{1 - \frac{|x - a_{j_i}^{K_i}|}{b^{K_i}}, 0\}, \qquad j_i = 1, 2, ..., K_i \tag{1}$$

$$a_{j_i}^{K_i} = \frac{(j_i - 1)}{K_i - 1}, \qquad j_i = 1, 2, ..., K_i \tag{2}$$

$$b^{K_i} = \frac{1}{K_i - 1} \tag{3}$$

The paper assumes that each linguistic label for each fuzzy space is represented by a **triangular membership function**. And for each fuzzy space, it's divided in K triangular membership functions.

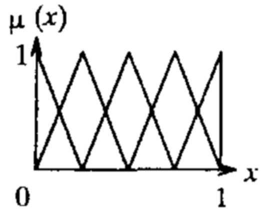


Figure 1: Partition of K=5

```python
def mu(x,j,k) -> np.ndarray:
  a = (j - 1)/(k-1)
  b = 1/(k-1)
  value = 1 - abs(x-a)/b
  if value <=0:
    return 0
  else:
    return value
```

Listing 1: Membership value function

```python
# For each feature you will have a k lenght vector with the
# membership degree of that place.
def membership_for_all_features(x: np.ndarray, k:np.ndarray):
  """For each feature you will have a k lenght vector with the membership degree of that
    place."""
  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:
          tmp.append(mu(x[sample],j+1,k[i]))
        else:
          tmp.append(mu(x[sample][f],j+1,k[i]))
      # 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
    output.append(membership_i)
  return output
```

Listing 2: Membership calculator with vector as input

And the function from Listing 2 computes the membership for each feature of the input in each K_i possible for that characteristic. It will output a vector with a **list for each sample** with the following structure:

```
output[
        Sample 1 [
                [mu(x11,1,k1),mu(x11,2,k1),..., mu(x11,,k1,k1)],
                [mu(x12,1,k2),mu(x12,2,k2),..., mu(x12,k2,k2)]
            ]
            .
            .
            .
        Sample p [
                [mu(xp1,1,k1),mu(xp1,2,k1),..., mu(xp1,,k1,k1)],
                [mu(xp2,1,k2),mu(xp2,2,k2),..., mu(xp2,k2,k2)]
            ]
      ]
```

2.1.2 Combinations of linguistic labels

Here we output an array for each possible combination between linguistic labels from different features. It is possible to use up to three features as input, but it is recommended to use only two.

```
# Here we just generate all posible combinations for all the feature k_i
def combinations_for_k(k:np.ndarray):
  """ Here we just generate all posible combinations for all the feature k_i
      for example k = [2,2] will return
      [[1,1],[1,2],[2,1],[2,2]]
  """
  output = []
  labels = len(k)

  if len(k)==3:
    for j in range(k[0]):
      for r in range(k[1]):
        for m in range(k[2]):
          output.append([j,r,m])
    return output
  if len(k)==2:
    for j in range(k[0]):
      for r in range(k[1]):
        output.append([j,r])
    return output
  else:
    for j in range(k[0]):
      output.append([j])
```

```python
    return output
```
Listing 3: K_i combinations

It will be used for accessing to all possible partitions in the different array vectors of the assignment.

2.1.3 Degree of compatibility

In this case we are computing a vector for convenience. The input of this functions are all the membership values possible for each K_i linguistic label in the corresponding feature of the sample. It will generate a product for each combinations between the different spaces possible in each sample.

$$\mu_{j_i \cdots j_n}(x_p) = \mu_{1j_1}(x_{p1}) \times \cdots \times \mu_{nj_n}(x_{pn}) \tag{4}$$

```python

# now we need to compute every degree of compatibility with every combination
# we will have k_1*k2*...K_n diferent combinations
# if one sample is provided use [[0,3  0,2  0,3]]
def degree_of_compatibility(membership_values, combinations):
  output = []
  n_features = len(combinations[0])
  n_samples = len(membership_values)

  for i in range(n_samples):
    general = []
    for c in combinations:
      tmp = []
      for j in range(n_features):
        if len(membership_values[i])==1:
            tmp.append(membership_values[i][0][c[j]])
        else:
            tmp.append(membership_values[i][j][c[j]])
      tmp=np.array(tmp)
      general.append(np.product(tmp))
    output.append(general)
  return output
```
Listing 4: Calculation of degree of compatibility

For example, in the case of $K_1 = 2$ and $K_2 = 2$ we would have 4 different combinations possible and we would be computing our values this way :

$$\begin{aligned}
\mu_{1,1}(x_p) &= \mu_{1,1}(x_{p1}) \times \mu_{2,1}(x_{p2}) \\
\mu_{1,2}(x_p) &= \mu_{1,1}(x_{p1}) \times \mu_{2,2}(x_{p2}) \\
\mu_{2,1}(x_p) &= \mu_{1,2}(x_{p1}) \times \mu_{2,1}(x_{p2}) \\
\mu_{2,2}(x_p) &= \mu_{1,2}(x_{p1}) \times \mu_{2,2}(x_{p2})
\end{aligned} \tag{5}$$

We will be generating a vector for each sample with the combinations possible calculated. In the case of this example, for each sample, we would have 4 values.

2.1.4 Calculation of $W_{j_1 \dots j_n}$

This operation allows getting values with a concentration or dilation applied. In this case, we use the degree of compatibility we already calculated and execute a *power* function with an alpha selected by the user. Depending on the alpha value we will be dilating or concentrating our function.

```python
# calculating all combinations for W
def Wj1_jn(dc,k:np.ndarray, alpha=1):
    return np.power(dc,alpha)
```

Listing 5: Calculation of $W_{j_1 \dots j_n}$

We introduce the vector obtained on Listing 4 as input and apply a power operation to all of the arrays with a selected α. The output will be the same shape as the input.

2.1.5 Calculation of $b_{j_1 .. j_n}$

Calculation of b values using the following formula:

$$b_{j_1 .. j_n} = \frac{\sum_{p=1}^{m} W_{j_1 \dots j_n (x_p)} \cdot y_p}{\sum_{p=1}^{m} W_{j_1 \dots j_n (x_p)}} \tag{6}$$

The code of the Listing 6 will output a 1-D array with a length equivalent to the amount of combination between spaces.

```python
# Now we need to compute bj1..jn, we will get a vector with the number of combinations
# posible
def bj1_jn(w, y_real):
  output = []
  n_samples=len(w)
  for j in range(len(w[0])):# these are the number of combinations we have
    numerator = []
    denominator = []
    for i in range(n_samples):
      numerator.append(w[i][j]*y_real[i])
      denominator.append(w[i][j])
    numerator=np.array(numerator)
    denominator=np.array(denominator)
    #print(denominator)
    value = np.sum(numerator)/np.sum(denominator)
    output.append(value)
  return output
```

Listing 6: Calculation of vector $b_{j_1 .. j_n}$

2.1.6 Prediction functions

In the Listing 7 we have the first implementation for calculating the y present on the section 2 of the paper Nozaki et al. 1995 equation 7:

```python
#y(x_p)
def prediction(dc,b,w):
    output = []
    n_samples=len(w)

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

        for j in range(len(w[0])):# these are the number of combinations we have
            numerator.append(dc[i][j]*b[j])
            denominator.append(dc[i][j])

        numerator=np.array(numerator)
        denominator=np.array(denominator)
        #print(denominator)
        value = np.sum(numerator)/np.sum(denominator)
        output.append(value)
    return output
```

Listing 7: First version of the prediction function (Not used)

The second prediction function is computed with equation 24 from Nozaki et al. 1995. In the Listing 8 is shown how is calculated the predictions and the **main** and **second** linguistic tables. Their dimensions will be matching the number of (K_1,K_2) shapes, and will contain linguistic variables encoded in the range $1, 2, ..., B$.

There are two auxiliary tables appearing as raw_first or raw_second. They have the same shapes as the anterior tables, but in this case, are used for storing the membership raw values from b. They will be used later for computational purposes.

```python
# This method allows to predict the values with the method shown in the section 4 of the
    papers
# Here we will be using the two tables
def predictV2(x,b:np.ndarray, B:np.ndarray,k:np.ndarray):

    membership_values = membership_for_all_features(b,B)

    membership_values_x = membership_for_all_features(x,k)
    combinations = combinations_for_k(k)

    # U_ji...jn
    dc = degree_of_compatibility(membership_values_x, combinations)

    # this tables only will contain the linguistic labels
    first_table = np.zeros((k[0],k[1]))
    second_table = np.zeros((k[0],k[1]))

    # this will contain the raw membership functions values
    raw_first = np.zeros((k[0],k[1]))
    raw_second = np.zeros((k[0],k[1]))

    output_table = np.zeros((k[0],k[1]))

    space = k_partitions(B[0], 0, 1)


    for i in range(len(combinations)):

        first_table[combinations[i][0]][combinations[i][1]]= np.argmax(membership_values[
    i])
```

```python
        raw_first[combinations[i][0]][combinations[i][1]] = membership_values[i][0][np.argmax(membership_values[i])]


    for i in range(len(combinations)):
        #####
        tmp = membership_values[i][0]
        tmp[np.argmax(membership_values[i])]=-42 # Giving a low value to find the second
        second_table[combinations[i][0]][combinations[i][1]]= np.argmax(tmp)

        raw_second[combinations[i][0]][combinations[i][1]]= membership_values[i][0][np.argmax(tmp)]

    prediction = []
    for sample in range(len(x)):
        numerator = 0
        denominator = 0
        for i in range(len(combinations)):
            # What is B**

            # Getting the first and the second membership space for the b fuzzy
            # in Bx[1] we will find the peak values
            B1 = space[int(first_table[combinations[i][0]][combinations[i][1]])]
            B2 = space[int(second_table[combinations[i][0]][combinations[i][1]])]

            numerator   += dc[sample][i]*B1[1]*raw_first[combinations[i][0]][combinations[i][1]] + dc[sample][i]*B2[1]*raw_second[combinations[i][0]][combinations[i][1]]
            denominator += dc[sample][i]*raw_first[combinations[i][0]][combinations[i][1]] + dc[sample][i]*raw_second[combinations[i][0]][combinations[i][1]]

        prediction.append(numerator/denominator)

    return prediction, first_table, second_table
```
Listing 8: Second version of the prediction function

3 Application to our problem

3.1 Data

To apply the implementation described above to our problem, at first we need data. As mentioned above, the problem is to predict the quality of a garment from the experience of the seamstress and the complexity of the pattern. The table below shows the input data in the columns ”Experience” and ”Complexity” and the output data in the column ”Quality”. The Quality is calculated through the next formula:

$$y_p = \frac{x_{p1}}{2} * x_{p_2} \tag{7}$$

Experience	**Complexity**	**Quality**
1.72287697	4.600602612	5.980389918
5.961742898	4.962609901	11.70053385

2.846869904	1.157335809	2.167970164
2.673053245	5.390372876	8.575964222
5.136766389	3.860893808	9.963476763
2.132178663	5.444342229	7.637390142
4.906378381	4.327792082	10.2599743
1.352626712	1.141006277	1.125640023
3.144253203	4.765588258	8.746040334
2.595661398	3.165771591	6.136976203
3.663848109	1.537841892	4.498191525
5.534987276	1.169565132	5.101110435
2.265245207	3.313063558	5.743149921
5.030462591	1.359218619	5.338672429
3.58116681	4.741897453	9.28946713
5.471198636	2.457227251	8.274978098
5.831809069	2.346039745	8.351086666
3.778286378	5.423857462	10.10573145
5.671019474	3.447902538	9.901861241
1.193523039	2.558211505	1.83737259
5.670601602	5.52179151	11.94679145
5.922895253	3.724739407	10.4259995
5.760038606	2.30537271	8.221365466
4.552207391	3.487821222	8.997462597
1.115684227	3.374462005	2.747158401
5.907126888	1.879426353	7.443716164
4.576478415	4.654678384	10.27391101
5.691051082	1.373488034	5.919873681
2.652502122	5.520260312	8.64585302
2.801583158	2.395515443	5.257724971
5.588194006	5.713351837	12.03132449
3.700036798	2.357097525	6.395535994
5.743809011	3.306323333	9.77515317
1.588273132	5.772336009	6.612467971
3.76670727	2.325199247	6.413920483
1.4333055	4.486877887	5.072530219
5.638319124	3.454023373	9.884449293
2.072490486	3.430100138	5.507693489
3.703140168	4.447999339	9.157048692
5.730757037	2.587132732	8.700007113
5.212867492	3.497502939	9.59804778
2.286207407	2.079788165	3.761047345
3.751797176	1.805569495	5.298235653
1.531557385	4.344099451	5.220030373
3.148859567	3.068217881	6.840094933
1.49626555	2.397050204	2.536558112

2.127177768	4.379981052	6.682460214
4.617833921	5.096214346	10.70656
4.993791945	2.326269732	7.640605187
1.180645953	3.113405038	2.64325311
1.13020032	3.669704533	3.167565285
4.976839848	5.067065593	11.00680196
4.908624937	1.089309347	4.27081084
4.586098427	1.888017719	6.364194414
1.929418838	4.889466468	6.736579909
5.019869309	1.029315653	4.12210992
1.151788468	2.459792195	1.512411494
5.730412396	4.573336926	11.1738908
4.573013356	3.53759877	9.078810483
1.146203981	3.637874027	3.190795731

Table 1: Set of samples used in the problem

The data will be **completely normalized** in a scale from 0 to 1, respecting the paper preprocessing of the data.

3.2 Computation

To build the fuzzy rule-based system, we used the data in the functions introduced in Section 2.2. The analysis of this process will be done in the next section.

4 Analysis and Conclusion

To analyze the effects of different numbers of linguistic labels and the effects of alpha on the performance of the fuzzy rule-based system, we ran the implementation with the Ks and alphas are seen in the table below. The performance is calculated by computing the summation of square errors between the real value y_p and the predicted value $y(x_p)$.

$$PI = \frac{\sum_{p=1}^{m} \{y(x_p) - y_p\}^2}{2} \tag{8}$$

First, we will look how the number of linguistic labels affects the performance. As we can see in the table, the performance index gets lower and therefore better the more linguistic labels we use. We can therefore assume that the number of linguistic labels gives the output better precision. This means that using more linguistic labels leads to more precise predicted output variables.

Second, we will look how the choice of alpha affects the performance. As we can see in the table, we used alphas between 0.1 and 100. From the table, we can conclude that different alphas work best for different Ks. But on a closer look, we can see that alphas above 10 perform worse than alpha = 10 and alphas below 5 always perform worse than alpha = 5.

Another variable that influences the performance of the algorithm is the division of B. We used two different divisions of B to see whether they would influence the PI final values. As seen in

alpha	K = 2	K = 3	K = 4	K = 5
0.1	1.807837	0.469818	0.209347	0.114915
0.5	1.221603	0.310282	0.129857	0.055851
1.0	0.845653	0.206260	0.080668	0.033214
5.0	0.279654	0.070386	0.034614	0.021762
10.0	0.241999	0.066506	0.038014	0.024087
50.0	0.253302	0.081356	0.049043	0.027729
100.0	0.262145	0.084890	0.052636	0.028507

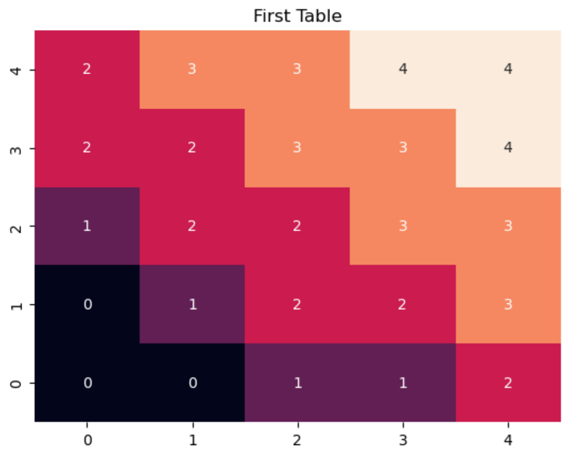
Table 2: Performance index of a fuzzy rule-based system. Method section 4. B = 5 and $K_1 = K_2$. The lower the better.

the table below, with $K_1 = K_2 = 5$ and $\alpha = 5$, a higher number of divisions will assure better performance:

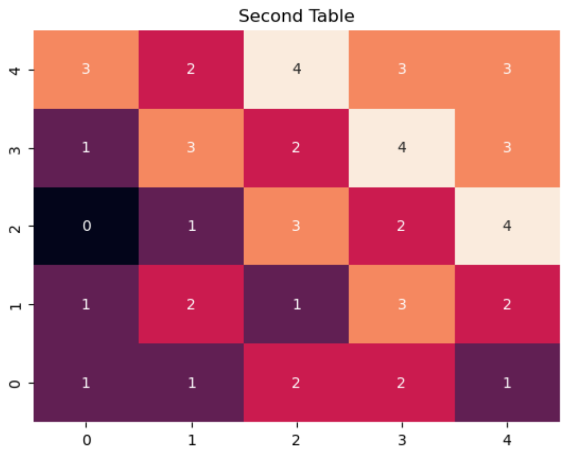
B	IP
10	0.0218157033353934422
5	0.0218157033353934453

Table 3: Performance index of changing B.

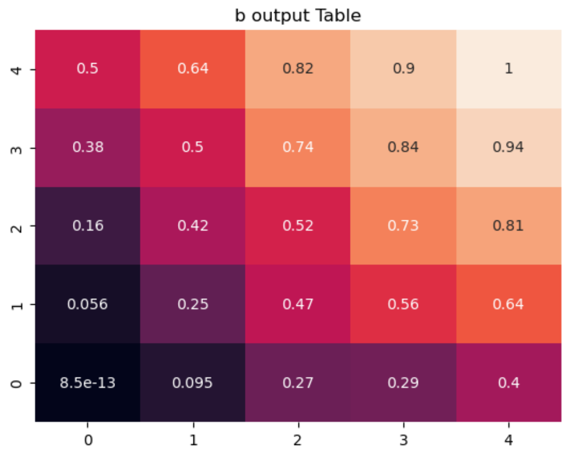
The higher B obtains a slightly better performance. In case we keep increasing the number of partitions, the performance will eventually decrease. However, the difference is not really perceptible in this case.



(a) Main rule table



(b) Secondary rule table



(c) b table

Figure 2: output from basic execution where $k_1 = K_2 = 5$, $B = 5$ and $\alpha = 5$. x-axis is representing **experience** and y-axis is representing **complexity**.

After running our data through the functions introduced in section 2, we obtained a main rule table and a secondary rule table, which can be seen in the figure above.

As we can see in the main table, the complexity and the experience of the seamstress influence the quality of the garment. Low experience and complexity will also lead to low quality of the finished garment. When the experience and complexity increase, the quality of the garment will increase as well. When we have either a complex design or an experienced seamstress, our quality will be mediocre. This tendencies can also be observed in the b-table.

The algorithm derived the rules that we expected. We have been able to generate these rules from a relatively small amount of input data via the heuristic method introduced in Nozaki et al. 1995.

This shows that the application of this method can be applied to a real world problem and can derive simple rules for a simple problem. In this case, we have been able to generate rules which have a similar mapping as a basic fuzzy if-then rule system.

5 User Guide

The code implementation will be available inside a jupyter notebook file with the content **already executed**. All the functions and imports are placed in the first part of the file, separated by markdown headers. The user should **check only** the content below the cell '**Execution Playground**' where the code is prepared to be easily accessible. The function 'loadDataset()' will load the csv file where the data is stored and the user is allowed to change their values at will. This enables the user to **tinker** with some of the parameters of K_i, B and α, therefore they have been established as variables to allow the user to test.

The user has to take into count that if used with a large number ($K_i > 8$) the membership values could come close to zero, and be rounded to zero. Then some b_{ij} values will get set to NaN. This could be solved by adding more samples, but the recommendation is to try to tinker with some values and check the outputs. Also it is not allowed to introduce a $K_i = 1$.

There are other functions extracted from the original paper created merely for testing, they can be also executed, but the dimensionality of x_p must be taken into account. The vector of k should have the same number of elements as the number of features. In case of having one dimensional x initialize the vector of k as $[[K_i]]$.

The representation of the matrix is only prepared to expect two inputs. They are allowed to have different K_i values and they can still be output as raw NumPy arrays.

References

[1] L. A. Zadeh, "Fuzzy sets," 1965.

[2] 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, 1997.