Laboratorium z Metod Inteligencji Obliczeniowej
Sprawozdanie

Autor Łukasz Gut - WFiIS, Informatyka Stosowana, Rok 2.
20 października 2018

Laboratorium 1 - Logika Rozmyta

Cel laboratorium:
Pierwsze laboratorium miało na celu zapoznanie nas z podstawami jednej z głównych metod inteligencji obliczeniowej: logiki rozmytej. Naszym zadaniem było napisanie prostego własnego rozmytego systemu wnioskującego na podstawie projektu demonstracyjnego odnośnie wysokości napiwku w restauracji. W tym sprawozdaniu przedstawię mój sposób implementacji rozmytego systemu wnioskującego na podstawie programu obliczającego optymalną ilość nauki w godzinach.

Wprowadzenie:
Logika rozmyta to jedna z logik wielowartościowych. Stanowi ona uogólnienie klasycznej dwuwartościowej logiki. Jej pomysłodawcą jest Lotfi Zadeh, amerykański automatyk pochodzenia azerskiego. Logika rozmyta opiera się na zbiorach rozmytych, czyli zbiorach, w których każdy obiekt ma jasno zdefiniowaną dla siebie funkcję przynależności, której wartości zawierają się w przedziale [0,1].

Formalnie:
$A = \{(x, \mu_A(x)) \mid x \varepsilon X\}$, gdzie $\mu_A : X \rightarrow [0, 1]$

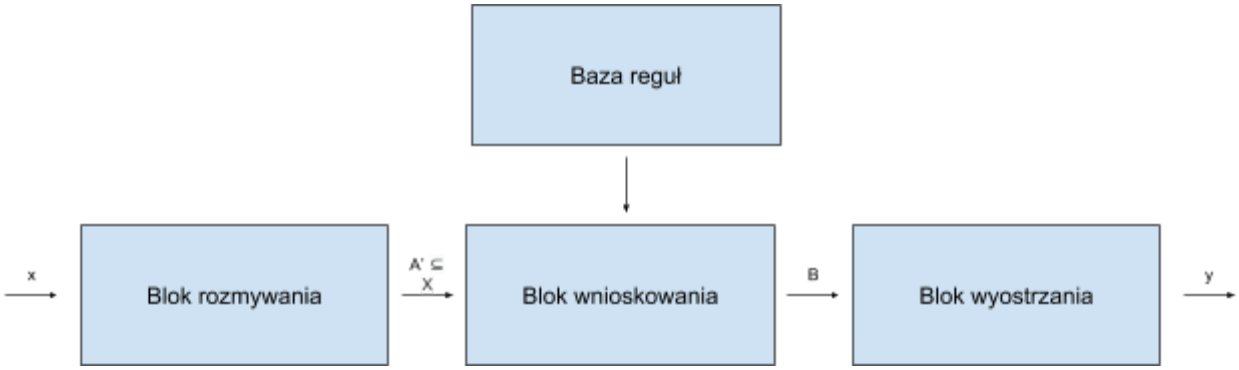
Konsekwencją tego jest fakt, iż w logice rozmytej pomiędzy stanem 0 (fałszem) a stanem 1 (prawdą) rozciąga się szereg wartości pośrednich, które określają stopień przynależności elementu do danego zbioru.

Koncept zbiorów rozmytych, a w szczególności wynikającej z nich logiki rozmytej pasuje wręcz idealnie do tworzenia tak zwanych rozmytych systemów wnioskujących. System wnioskujący składa się z następujących elementów:

- Baza reguł (lingwistyczny model, zbiór reguł rozmytych)
- Blok rozmywania (blok, w którym konkretna wartość sygnału wejściowego jest poddawana operacji rozmywania, w wyniku której zostaje odwzorowana w zbiór rozmyty)

- Blok wnioskowania (blok, który określa rozmyty zbiór wyjściowy na podstawie przyjętych reguł wnioskowania)
- Blok wyostrzania (blok, który odwzorowuje wyjściowy zbiór rozmyty w konkretną wartość końcową)

Ogólny schemat rozmytego systemu wnioskującego przedstawiam poniżej:



Moim zadaniem, tak jak opisałem to w paragrafie opowiadającym o celu laboratorium, było stworzenie mojego własnego rozmytego systemu wnioskowania oraz sprawdzenie jak ilość funkcji przynależności dla każdej z danych wejściowych wpływa na "przewidywany" przez program wynik. Z tego powodu przyjrzymy się dokładnie 3, 5 oraz 7 funkcjom przynależności oraz odpowiadającym im zasadom, a następnie wyciągniemy z tego wnioski.

Implementacja:
Zadaniem mojego programu jest pobranie od użytkownika dwóch argumentów, które później przekażemy na wejście naszego systemu rozmytego. Argumenty te to subiektywna ocena ilości czasu wolnego oraz subiektywna ocena ilości materiałów do przerobienia. Na podstawie tych dwóch danych system podpowiada nam optymalny czas nauki.

We wszystkich wersjach mojego programu deklaracja zmiennych przebiega następująco:

```
# New Antecedent/Consequent objects hold universe variables and membership
# functions
free_time = ctrl.Antecedent(np.arange(0, 11, 1), 'free_time')
material_amount = ctrl.Antecedent(np.arange(0, 11, 1), 'material_amount')
study_time = ctrl.Consequent(np.arange(0, 5, 1), 'study_time')
```

Tworzę 2 zmienne wejściowe: **x_free_time** oraz **x_material_amount**, których uniwersum zawiera się w przedziale **[0,10]**. Następnie tworzę jedną zmienną wyjściową **x_study_time**, której uniwersum zawiera się w przedziale **[0,4]**.

Następnie tworzę funkcje przynależności. W tym konkretnym problemie zdecydowałem się wykorzystać tzw. trójkątną funkcję przynależności (dostarcza ją metoda **trimf** z biblioteki **scikit-fuzzy**).

Uwaga: Korzystanie z innego rodzaju funkcji przynależności np. Trapezoidalnych lub Gaussowskich w przypadku naszego problemu, nie wpływa w sposób znaczący na otrzymywany wynik.

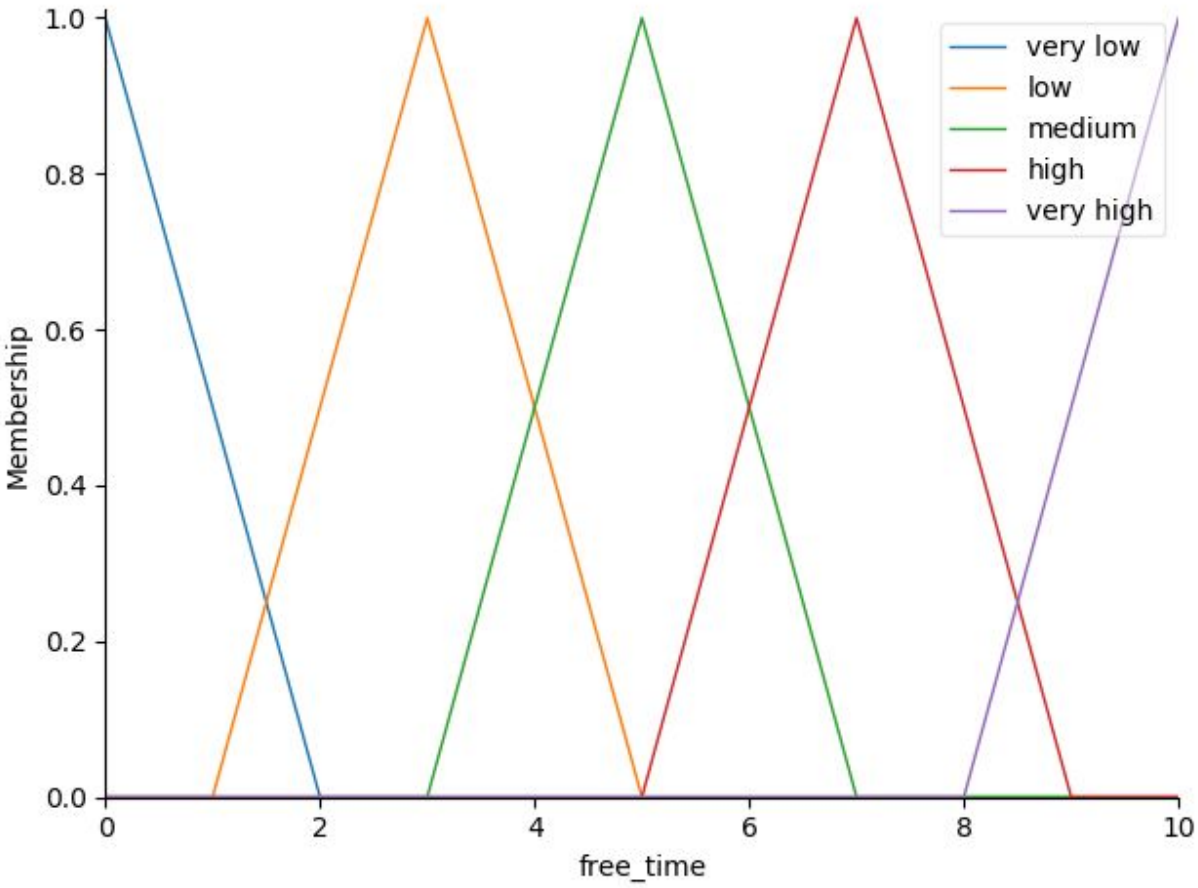
Przykład dla 5 funkcji przynależności

```
free_time['very low'] = fuzz.trimf(free_time.universe, [0, 0, 2])
free_time['low'] = fuzz.trimf(free_time.universe, [1, 3, 5])
free_time['medium'] = fuzz.trimf(free_time.universe, [3, 5, 7])
free_time['high'] = fuzz.trimf(free_time.universe, [5, 7, 9])
free_time['very high'] = fuzz.trimf(free_time.universe, [8, 10, 10])

material_amount['very low'] = fuzz.trimf(material_amount.universe, [0, 0, 2])
material_amount['low'] = fuzz.trimf(material_amount.universe, [1, 3, 5])
material_amount['medium'] = fuzz.trimf(material_amount.universe, [3, 5, 7])
material_amount['high'] = fuzz.trimf(material_amount.universe, [5, 7, 9])
material_amount['very high'] = fuzz.trimf(material_amount.universe, [8, 10, 10])

study_time['very low'] = fuzz.trimf(study_time.universe, [0, 0, 1])
study_time['low'] = fuzz.trimf(study_time.universe, [0, 1, 2])
study_time['medium'] = fuzz.trimf(study_time.universe, [1, 2, 3])
study_time['high'] = fuzz.trimf(study_time.universe, [2, 3, 4])
study_time['very high'] = fuzz.trimf(study_time.universe, [3, 4, 5])
```

Funkcje te na wykresie prezentują się następująco:



Oczywiście dla kolejnych programów tych funkcji jest odpowiednio mniej lub więcej.

Teraz przyszła kolej na implementację zasad, czyli rdzenia całego rozmytego systemu wnioskującego. Implementacja wygląda następująco:

3 funkcje przynależności

```
#rule 1: IF AMOUNT OF TIME LOW AND MATERIAL AMOUNT LOW => OPTIMAL STUDY TIME LOW
activate_rule1 = np.fmax(free_time_level_lo, material_amount_level_lo)

study_time_activation_lo = np.fmin(activate_rule1,study_time_lo)

#rule 2: IF MATERIAL AMOUNT IS MID => OPTIMAL STUDY TIME MID
study_time_activation_md = np.fmin(material_amount_level_md,study_time_md)

#rule 3: IF FREE TIME IS HIGH AND MATERIAL AMOUNT IS HIGH => OPTIMAL STUDY TIME HIGH
activate_rule3 = np.fmax(free_time_level_hi,material_amount_level_hi)

study_time_activation_hi = np.fmin(activate_rule3,study_time_hi)
```

5 funkcji przynależności

```
free_time['very low'] = fuzz.trimf(free_time.universe, [0, 0, 2])
free_time['low'] = fuzz.trimf(free_time.universe, [1, 3, 5])
free_time['medium'] = fuzz.trimf(free_time.universe, [3, 5, 7])
free_time['high'] = fuzz.trimf(free_time.universe, [5, 7, 9])
free_time['very high'] = fuzz.trimf(free_time.universe, [8, 10, 10])

material_amount['very low'] = fuzz.trimf(material_amount.universe, [0, 0, 2])
material_amount['low'] = fuzz.trimf(material_amount.universe, [1, 3, 5])
material_amount['medium'] = fuzz.trimf(material_amount.universe, [3, 5, 7])
material_amount['high'] = fuzz.trimf(material_amount.universe, [5, 7, 9])
material_amount['very high'] = fuzz.trimf(material_amount.universe, [8, 10, 10])

study_time['very low'] = fuzz.trimf(study_time.universe, [0, 0, 1])
study_time['low'] = fuzz.trimf(study_time.universe, [0, 1, 2])
study_time['medium'] = fuzz.trimf(study_time.universe, [1, 2, 3])
study_time['high'] = fuzz.trimf(study_time.universe, [2, 3, 4])
study_time['very high'] = fuzz.trimf(study_time.universe, [3, 4, 5])
```

7 funkcji przynależności

```
r1 = ctrl.Rule(antecedent=((free_time['vvl'] & material_amount['vvl']) |
                           (free_time['vvl'] & material_amount['vl']) |
                           (free_time['vl'] & material_amount['vvl'])),
               consequent=study_time['vvl'], label='rule vvl')

r2 = ctrl.Rule(antecedent=((free_time['vl'] & material_amount['vl']) |
                           (free_time['vl'] & material_amount['l']) |
                           (free_time['l'] & material_amount['vl'])),
               consequent=study_time['vl'], label='rule vl')

r3 = ctrl.Rule(antecedent=((free_time['l'] & material_amount['l']) |
                           (free_time['l'] & material_amount['m']) |
                           (free_time['m'] & material_amount['l'])),
               consequent=study_time['l'], label='rule l')

r4 = ctrl.Rule(antecedent=((free_time['m'] & material_amount['m']) |
                           (free_time['l'] & material_amount['m']) |
                           (free_time['m'] & material_amount['l'])),
               consequent=study_time['m'], label='rule m')

r5 = ctrl.Rule(antecedent=((free_time['h'] & material_amount['h']) |
                           (free_time['vh'] & material_amount['h']) |
                           (free_time['h'] & material_amount['vh'])),
               consequent=study_time['h'], label='rule h')

r6 = ctrl.Rule(antecedent=((free_time['vh'] & material_amount['vh']) |
                           (free_time['vh'] & material_amount['vvh']) |
                           (free_time['vvh'] & material_amount['vh'])),
               consequent=study_time['vh'], label='rule vh')

r7 = ctrl.Rule(antecedent=((free_time['vvh'] & material_amount['vvh'])),
               consequent=study_time['vvh'], label='rule vvh')
```

Uwaga: Warto tutaj zwrócić uwagę na sposób implementacji, który różni się ze względu na stosowane przeze mnie API. Do rozwiązania problemu z użyciem 3 funkcji przynależności wykorzystałem starszą wersję API, która wymagała nieco więcej kodu. Nowsze wersje tego

samego API robią praktycznie wszystko za nas, dzięki czemu potencjalnie wiele skomplikowanych reguł można zapisać w krótki i przejrzysty sposób.

Przyjrzyjmy się teraz blokowi wnioskowania. Dzięki prostocie najnowszej wersji **scikit-fuzzy** całość sprowadza się właściwie do jednej linijki.

Przykład dla 5 funkcji przynależności

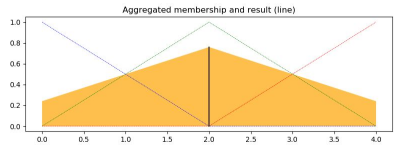
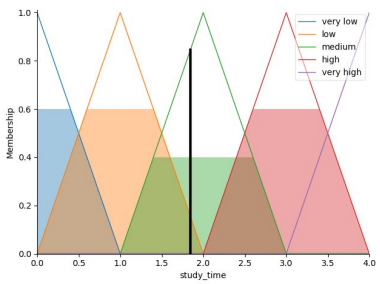
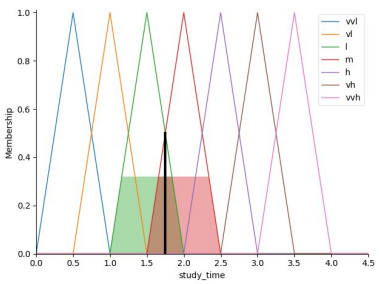
```
study_time_ctrl = ctrl.ControlSystem([r1,r2,r3,r4,r5,r6,r7,r8,r9])
```

Na samym końcu wystarczy już tylko podać wartości wejściowe i za pomocą metody **compute()** obliczyć wszystko co potrzebne, aby dostać interesujące nas dane wyjściowe. A zatem spróbujmy i przekonajmy się jak ilość funkcji przynależności wpływa na nasz wynik.

Input:

free_time_input = 6.2
material_amount_input = 3.8

Output:

3 funkcje przynależności	5 funkcji przynależności	7 funkcji przynależności
2.0	**1.8457142857142856**	**1.75**
		

Jak widzimy wynik różni się w zależności od użytych przez nas funkcji przynależności. Im jest ich więcej, tym bliżej jesteśmy wyniku, którego oczekiwaliśmy, jednak stopień skomplikowania problemu rośnie diametralnie. Wystarczy spojrzeć na stopień skomplikowania reguł przy korzystaniu z 7 funkcji przynależności, a następnie porównać to ze stopniem skomplikowania reguł w przypadku 3 funkcji przynależności.

Wniosek: Dużą ilość funkcji przynależności należy stosować tylko wtedy, gdy chcemy otrzymać wynik jak najbardziej dokładny. W naszym przypadku 5 funkcji przynależności to moim zdaniem za dużo, ponieważ cały system i tak opiera się na subiektywnej ocenie ilości czasu wolnego oraz materiałów do przerobienia. Gdybyśmy chcieli jednak zasymulować coś znacznie bardziej ciekawego, wtedy korzystanie z większej ilości funkcji przynależności zda egzamin, jednak w dalszym ciągu należy pamiętać, aby nie dodawać ich zbyt dużo.