ZDANIE SPRAWY

Projekt - ITO (N2)

Przez
Krystian Sereda

Wydział Elektryczny PW
Informatyka Stosowana

Spis treści

1 Treść Zadania

Zaprojektować i wykonać system wnioskowania w logikach rozmytych typu 1 i typu 2, służący do podejmowania decyzji o wiarygodności klienta Banku. System ma wspomagać pracownika Banku podejmującego decyzje o udzieleniu kredytu (klientowi Banku).

Minimalny zbiór typów decyzji:

h_1 „udziel kredytu do 100 000 zł (lub do 200 000 zł, lub do 400 000 zł) klientowi X"

h_2 „udziel kredytu o racie spłaty 2 000 zł (lub 4 000 zł, lub 6 000 zł) klientowi Y"

h_3 „udziel kredytu na okres 10, 20 lub 30 lat"

Kryteria decyzyjne (zmienne rozmyte):

- średnie miesięczne wpływy na konto klienta w okresie ostatniego pół roku,
- suma zarobków klienta i współmałżonka,
- liczba osób na utrzymaniu,
- wiek kredytobiorcy itd.

Generowanie funkcji przynależności:
Program ma automatycznie tworzyć funkcje przynależności dla każdej zmiennej rozmytej na podstawie przedziałów wartości podanych przez użytkownika.

Interfejs użytkownika:

- wprowadzanie obserwacji (faktów),
- wybór rodzaju decyzji (h_1, h_2 lub h_3),
- prezentacja wyników pośrednich (wartości przynależności, stopnie pewności) oraz końcowego wyniku procesu wnioskowania.

Porównanie wyników:
Przeprowadzić analizę porównawczą wyników uzyskanych za pomocą logiki rozmytej typu 1 oraz typu 2.

2 Część teoretyczna

Logika rozmyta umożliwia przetwarzanie wiedzy niedokładnej, w której przynależność obiektów do relacji nie jest binarna (0 lub 1), lecz opisywana funkcją przynależności $\mu_A(x) \in [0,1]$. Oznacza to, że element może należeć do zbioru w pewnym stopniu, a nie tylko zero-jedynkowo.

2.1 Operacje na wartościach w przedziale $[0,1]$

Zamiast klasycznych operacji na zbiorach $\{0,1\}$ stosuje się ich uogólnienia w logice rozmytej:

$$AND(a,b) \;=\; \min(a,b), \qquad OR(a,b) \;=\; \max(a,b), \qquad NOT(a) \;=\; 1-a,$$

gdzie $a, b \in [0,1]$. Dzięki temu można formalnie operować pojęciami typu „mało", „dużo", „prawie" czy „być może" i uzyskiwać wnioski opierając się na stopniach przynależności.

2.2 Zbiory rozmyte

Zbiór rozmyty A na uniwersum X definiuje się jako

$$A \;=\; \{\,(x, \mu_A(x)) : x \in X,\ \mu_A(x) \in [0,1]\}.$$

Jeśli $\mu_A(x) = 1$, mówimy, że x należy w pełni do A; jeśli $\mu_A(x) = 0$, nie należy wcale; a gdy $\mu_A(x) \in (0,1)$, należy w pewnym stopniu.

2.3 Etapy wnioskowania rozmytego

Typowy system wnioskowania rozmytego składa się z czterech kroków:

1. **Rozmywanie (fuzzyfikacja)**
   Każda zmienna wejściowa X_i ma zdefiniowane etykiety lingwistyczne (np. „niski", „średni", „wysoki") oraz odpowiadające im funkcje przynależności $\mu_{A_i^{(j)}}(x)$. W momencie wprowadzenia wartości x_i^* oblicza się stopnie $\mu_{A_i^{(j)}}(x_i^*) \in [0,1]$, co daje wektor przynależności opisujący, jak mocno x_i^* należy do każdej etykiety.

2. **Reguły rozmyte (IF–THEN)**
   Reguły mają postać

   $$R_k: \quad (X_{i_1} jest A_{i_1}^{(k)})\ AND/OR\ (X_{i_2} jest A_{i_2}^{(k)})\ \ldots \implies Y jest B^{(k)}.$$

   Dla każdej reguły oblicza się stopień aktywacji α_k przez:

   - jeżeli warunki są połączone **AND**, to $\alpha_k = \min(\ldots)$,
   - jeżeli są połączone **OR**, to $\alpha_k = \max(\ldots)$.

   Tym sposobem otrzymuje się dla każdej reguły $\alpha_k \in [0,1]$ oraz odpowiadającą jej etykietę wyjściową $B^{(k)}$. Reguły z $\alpha_k = 0$ są pomijane.

3. **Agregacja**
   Po obliczeniu α_k dla aktywnych reguł każdą funkcję przynależności wyjścia $\mu_{B^{(k)}}(y)$ przycina się do wysokości α_k:

   $$\mu'_{B^{(k)}}(y) \;=\; \min(\mu_{B^{(k)}}(y),\, \alpha_k).$$

   Następnie, dla każdej etykiety wyjściowej C_j, grupuje się wszystkie $\mu'_{B^{(k)}}$ z $B^{(k)} = C_j$ i w każdym punkcie y bierze maksimum:

   $$\mu^{\text{ag}}_{C_j}(y) \;=\; \max_{\{k:B^{(k)}=C_j\}} \mu'_{B^{(k)}}(y).$$

   W ten sposób uzyskujemy „rozmyty rozkład" funkcji przynależności Y dla każdej etykiety.

4. **Wyostrzanie (defuzzyfikacja)**
   Aby otrzymać jedną ostrą wartość y^*, stosujemy jedną z dwóch metod (w naszym przypadku):

Typ 1 Dla każdej aktywnej reguły R_k, z $\alpha_k > 0$, przypisuje się reprezentanta z_k (liczbę odpowiadającą etykiecie $B^{(k)}$). Ostateczna wartość to

$$y^* = \frac{\sum_{k:\,\alpha_k>0} \alpha_k\, z_k}{\sum_{k:\,\alpha_k>0} \alpha_k}.$$

Typ 2 Przeskalowuje się każdą oryginalną funkcję przynależności $\mu_{B^{(k)}}(y)$ przez α_k, a następnie sumuje wszystkie te przeskalowane krzywe w jedną:

$$\mu_{\text{sum}}(y) = \sum_{k:\,\alpha_k>0} \alpha_k\, \mu_{B^{(k)}}(y).$$

Wartość y^* oblicza się jako środek masy tej sumarycznej funkcji:

$$y^* = \frac{\int y\, \mu_{\text{sum}}(y)\, dy}{\int \mu_{\text{sum}}(y)\, dy}.$$

3 Analiza problemu

Przyznanie kredytu w banku nie jest jedynie decyzją „tak/nie", lecz pytaniem „ile" i „na jaki okres" udzielić środków. Dane o kliencie (np. średnie wpływy, suma dochodów, liczba osób na utrzymaniu, wiek) są często nieprecyzyjne, dlatego zastosujemy *logikę rozmytą*, która pozwala modelować stopnie przynależności do pojęć typu „niski", „średni" i „wysoki".

Główne zadania algorytmu wnioskowania rozmytego to:

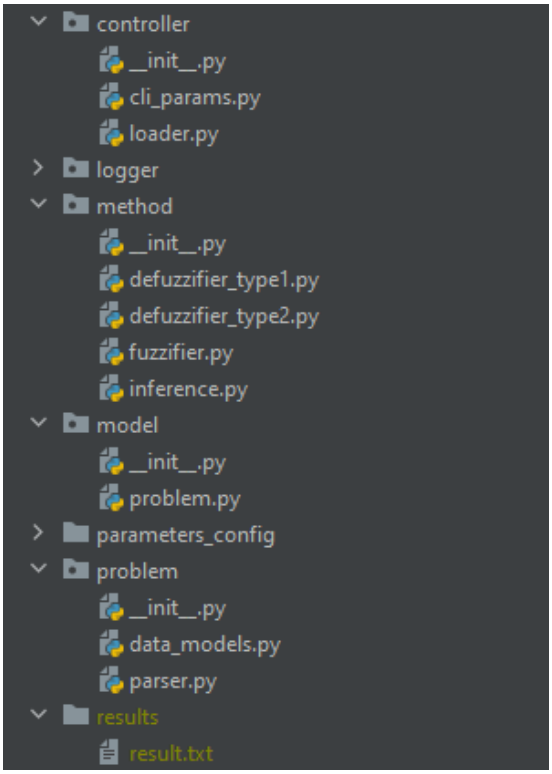
1. **Przypisywanie wartości liczbowych do etykiet** Każda wartość (np. dochód 6000 zł) jest automatycznie przetłumaczona na wektor stopni przynależności $(\mu_{niski}, \mu_{redni}, \mu_{wysoki}) \in [0, 1]$. Dzięki temu „6000 zł dochodu" może być częściowo „niski" i częściowo „średni".
2. **Wnioskowanie przez reguły IF–THEN** Na podstawie uzyskanych stopni przynależności system przechodzi przez zdefiniowaną bazę reguł rozmytych. Każda reguła otrzymuje stopień aktywacji $\alpha_k \in [0, 1]$ metodą min/max. W efekcie wiadomo, w jakim stopniu dana reguła „działa" w ocenie wiarygodności klienta.
3. **Agregacja wyników** Wszystkie aktywne reguły, każda z wagą α_k, przycinają odpowiadające im funkcje przynależności etykiet wyjściowych (np. „mały kredyt", „średni kredyt", „duży kredyt"). Następnie dla każdej etykiety wybiera się maksimum z przyciętych wykresów, tworząc rozmyty rozkład decyzji.
4. **Ostateczna wartość przez defuzzyfikację** System konwertuje rozmyty rozkład na jedną, ostrą rekomendację. W Typie 1 stosuje się ważoną średnią reprezentantów (np. 100 000 zł, 200 000 zł, 400 000 zł), a w Typie 2 wyznacza się centroid sumarycznej funkcji przynależności.

W efekcie algorytm nie zwróci jedynie „udzielamy lub nie udzielamy kredytu", lecz wskaże konkretną kwotę, wysokość raty lub okres spłaty, uwzględniając niuanse wszystkich wprowadzonych kryteriów.

4 Wstępna koncepcja projektu

Projekt zostanie zorganizowany w następujący sposób:

- **Pliki konfiguracyjne:**
  - *membership_functions.json* – definicje funkcji przynależności
  - *rules.json* – lista reguł IF–THEN
  - *representatives.json* – wartości reprezentantów etykiet wyjściowych
- **Moduły aplikacji:**
  - *Moduł komunikacji (CLI)* – odpowiada za parsowanie parametrów przekazanych z linii poleceń oraz wyświetlanie wyników i ewentualnych komunikatów użytkownikowi.
  - *Moduł danych* – przy starcie wczytuje pliki z regułami, funkcjami przynależności i reprezentantami, konstruując wewnętrzne struktury (np. słowniki, tablice).
  - *Moduł logiki biznesowej* – realizuje wszystkie etapy wnioskowania rozmytego:
    - fuzzyfikacja (przypisywanie wartości liczbowych do etykiet),
    - ocena reguł (obliczanie stopni aktywacji α_k),
    - agregacja (przyczenie i łączenie funkcji wyjściowych),
    - defuzzyfikacja (Typ 1: ważona średnia, Typ 2: centroid sumarycznej funkcji).
- **Uruchamianie:**
  - Program uruchamiany będzie z poziomu linii poleceń (CLI), podając jako argumenty ścieżki do plików konfiguracyjnych oraz wartości wejściowe (np. wpływy, dochód, liczba osób).
  - Po wczytaniu konfiguracji i danych wejściowych, moduł biznesowy wykona cały proces wnioskowania rozmytego, a wyniki zostaną zwrócone w formie tekstowej do konsoli.



Rysunek 1:

5 Realizacja

Projekt został zaimplementowany w języku Python. Moduły zostały nieco zmienione względem wstępnej koncepcji — zmieniło się nazewnictwo i zakres odpowiedzialności. Poniżej opisano strukturę i główne komponenty.

5.1 Controller

Moduł `controller` odpowiada za parsowanie i walidację parametrów wprowadzanych z linii poleceń oraz za wczytywanie plików konfiguracyjnych. Program uruchamia się przy pomocy czterech flag: `-inflow`, `-income_sum`, `-dependents` i `-age`. Każdy parametr ma zdefiniowany dopuszczalny zakres wartości, a w razie podania nieprawidłowej wartości użytkownik otrzymuje komunikat o błędzie, na przykład:

```
--age must be in [18,100]
```

W ramach tego modułu zaimplementowano także funkcje wczytujące definicje z plików JSON znajdujących się w katalogu `parameters_config/`. Przykład fragmentu z `input_mfs.json`:

```
{
        "inflow": {
                "low": {
                        "type":   "trapezoid",
                        "points": [
                        [   1, 1],
                        [   1, 1],
                        [3000, 1],
                        [4500, 0]
                        ]
                },
                "medium": {
                        "type":   "trapezoid",
                        "points": [
                        [3000, 0],
                        [4500, 1],
                        [6000, 1],
                        [8000, 0]
```

```
                        ]
                },
                "high": {
                        "type":   "trapezoid",
                        "points": [
                        [6000,  0],
                        [8000,  1],
                        [10000, 1],
                        [10000, 1]
                        ]
                }
        }
}
```

Fragment `output_mfs.json` definiuje kształty funkcji wyjściowych, na przykład:

```
{
        "up_to_100k": {
                "type":   "trapezoid",
                "points": [
                [     0, 0],
                [     1, 1],
                [100000, 1],
                [400000, 0]
                ]
        }
}
```

W pliku `representatives.json` znajdują się wartości reprezentantów używane w defuzyfikacji typu 1:

```
{
        "h1": {
                "up_to_100k": 100000,
                "up_to_200k": 200000,
                "up_to_400k": 400000
        }
}
```

Baza reguł IF–THEN jest przechowywana w `rules.json`. Przykładowa reguła dla decyzji `h1` wygląda następująco:

```
{
        "if": {
                "or": [
                { "and": [ { "inflow":"low" },       { "income_sum":"low" } ] },
                { "and": [ { "dependents":"high" },  { "age":"old" } ] }
                ]
        },
        "then": { "h1": "up_to_100k" }
}
```

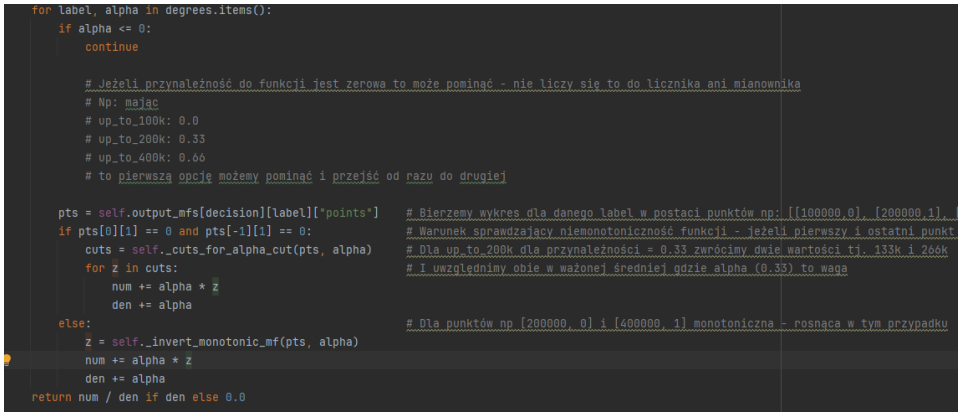
5.2 Model (Problem)

Moduł `model.problem` przygotowuje zbiór reguł dla każdego z zadań decyzyjnych (`h1`, `h2`, `h3`) i udostępnia je do dalszej analizy.

5.3 Method

Moduł `method` realizuje wszystkie cztery etapy standardowego algorytmu wnioskowania rozmytego. W kodzie zawarto komentarze wskazujące kluczowe aspekty w implementacjach **Typu 1** oraz **Typu 2**.

`fuzzifier.py` odpowiada za *fuzzyfikację* — każde wejście x_i jest interpolowane liniowo względem jego przedziałów wierzchołkowych, aby obliczyć stopień przynależności $\mu_A(x_i) \in [0, 1]$ dla wszystkich etykiet lingwistycznych.

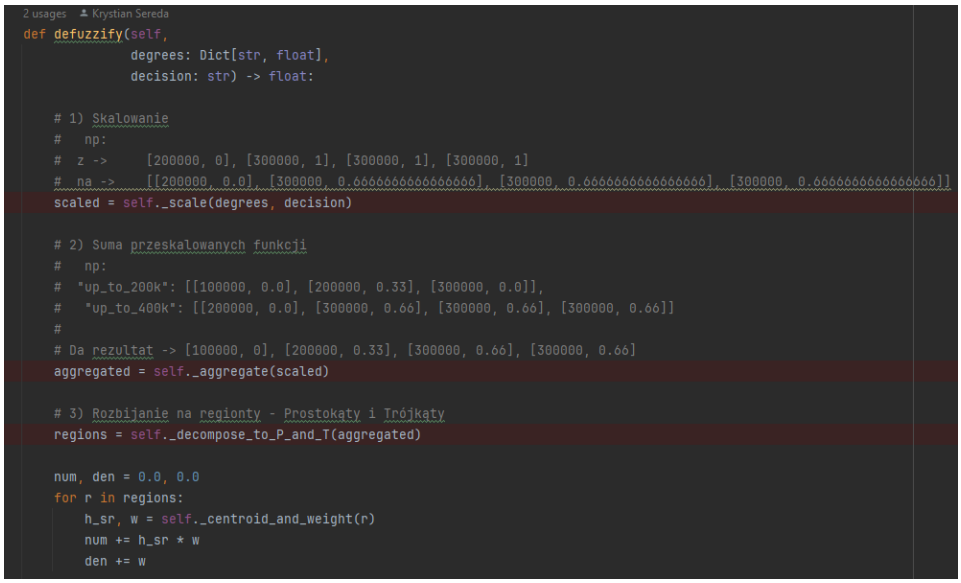


```
for label, alpha in degrees.items():
    if alpha <= 0:
        continue

        # Jeżeli przynależność do funkcji jest zerowa to może pominąć - nie liczy się to do licznika ani mianownika
        # Np: mając
        # up_to_100k: 0.0
        # up_to_200k: 0.33
        # up_to_400k: 0.66
        # to pierwszą opcję możemy pominąć i przejść od razu do drugiej

    pts = self.output_mfs[decision][label]["points"]    # Bierzemy wykres dla danego label w postaci punktów np: [[100000.0], [200000.1],
    if pts[0][1] == 0 and pts[-1][1] == 0:              # Warunek sprawdzający niemonotoniczność funkcji - jeżeli pierwszy i ostatni punkt
        cuts = self._cuts_for_alpha_cut(pts, alpha)     # Dla up_to_200k dla przynależności = 0.33 zwrócimy dwie wartości tj. 133k i 266k
        for z in cuts:                                  # I uwzględnimy obie w ważonej średniej gdzie alpha (0.33) to waga
            num += alpha * z
            den += alpha
    else:                                               # Dla punktów np [200000, 0] i [400000, 1] monotoniczna - rosnąca w tym przypadku
        z = self._invert_monotonic_mf(pts, alpha)
        num += alpha * z
        den += alpha
return num / den if den else 0.0
```

Rysunek 2: Przykład komentarzy w kodzie



```
2 usages  Krystian Sereda
def defuzzify(self,
              degrees: Dict[str, float],
              decision: str) -> float:

    # 1) Skalowanie
    #   np:
    #  z ->     [200000, 0], [300000, 1], [300000, 1], [300000, 1]
    #  na ->    [[200000, 0.0], [300000, 0.6666666666666666], [300000, 0.6666666666666666], [300000, 0.6666666666666666]]
    scaled = self._scale(degrees, decision)

    # 2) Suma przeskalowanych funkcji
    #   np:
    #  "up_to_200k": [[100000, 0.0], [200000, 0.33], [300000, 0.0]],
    #   "up_to_400k": [[200000, 0.0], [300000, 0.66], [300000, 0.66], [300000, 0.66]]
    #
    # Da rezultat -> [100000, 0], [200000, 0.33], [300000, 0.66], [300000, 0.66]
    aggregated = self._aggregate(scaled)

    # 3) Rozbijanie na regiony - Prostokąty i Trójkąty
    regions = self._decompose_to_P_and_T(aggregated)

    num, den = 0.0, 0.0
    for r in regions:
        h_sr, w = self._centroid_and_weight(r)
        num += h_sr * w
        den += w
```

Rysunek 3: Przykład komentarzy w kodzie

`inference.py` realizuje *wnioskowanie* poprzez analizę każdego obiektu `Rule`:

- warunki połączone **AND** zwracają $\alpha_k = \min(\ldots)$,
- warunki połączone **OR** zwracają $\alpha_k = \max(\ldots)$.

Wynikiem jest słownik aktywacji $\{\alpha_k\}$ dla tych reguł, które odnoszą się do wybranej decyzji.

`defuzzifier_type1.py` przeprowadza *defuzyfikację typu 1*. Dla każdej reguły z $\alpha_k > 0$ pobiera wartość reprezentanta z_k z pliku `representatives.json` i wylicza:

$$y^* = \frac{\sum_{k:\,\alpha_k>0} \alpha_k\, z_k}{\sum_{k:\,\alpha_k>0} \alpha_k}.$$

Jednym z wymagań tej metody jest *monotoniczność* funkcji przynależności: etykiety o kształcie „górki” (np. `medium`) muszą zostać przed defuzyfikacją podzielone na dwie monotoniczne części (`mid_low`, `mid_high`), aby odwzorowanie $\mu^{-1}(\alpha)$ było jednoznaczne.

`defuzzifier_type2.py` wykonuje *defuzyfikację typu 2* w trzech jasno wyodrębnionych krokach:

1. **Skalowanie:**

$$\mu'_{B^{(k)}}(y) = \alpha_k\, \mu_{B^{(k)}}(y).$$

2. **Agregacja:**
$$\mu_{\text{sum}}(y) = \sum_{k:\,\alpha_k>0} \mu'_{B^{(k)}}(y).$$

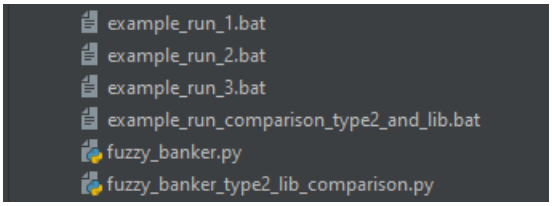
3. **Defuzyfikacja centroidalna:** rozkłada się krzywą $\mu_{\text{sum}}(y)$ na kolejne regiony — fragmenty o stałej wartości μ (prostokąty) oraz fragmenty o liniowej zmianie μ (trójkąty) — i dla każdej figury oblicza centroidę h_{sr} i pole w:
   Prostokąt:
$$h_{\text{sr}}(P) = h_{\min} + \frac{h_{\max} - h_{\min}}{2}, \quad w(P) = \mu\,(h_{\max} - h_{\min}),$$
   Trójkąt:
$$h_{\text{sr}}(T) = h_{\min} + 23\,(h_{\max} - h_{\min}), \quad w(T) = 12\,\Delta\mu\,(h_{\max} - h_{\min}).$$

Głównym wyzwaniem było automatyczne wykrycie, które odcinki są prostokątami (stała przynależność) a które trójkątami (zmieniające się μ), a także rozróżnienie, czy nachylenie trójkąta jest rosnące czy malejące, aby dobrać odpowiednie wzory na centroidę.

5.4 Logger

Moduł `logger.logger` zapisuje do pliku `results/result.txt` oraz wyświetla w konsoli: parametry wejściowe wraz ze stopniami przynależności, stopnie aktywacji reguł, przynależności wyjścia oraz końcowe wyniki defuzyfikacji obu typów.

6 Działanie i obsługa programu



Rysunek 4:

Główny skrypt `fuzzy_banker.py`, wspomagający pracownika Banku w podejmowaniu decyzji o przyznaniu kredytu, uruchamia się z poziomu wiersza poleceń. Przykładowe wywołanie:

```
python fuzzy_banker.py \
--inflow     6500 \
--income_sum 11100 \
--dependents 1 \
--age        46
```

Alternatywnie można dwukrotnie kliknąć jeden z przygotowanych plików wsadowych (`example_run_1.bat`, `example_run_2.bat`, `example_run_3.bat`), które zawierają różne zestawy parametrów.

Po uruchomieniu program:

1. Weryfikuje wartości wejściowe, zgłaszając błąd i przerywając działanie w przypadku przekroczenia dozwolonych zakresów.
2. Ładuje pliki konfiguracyjne z katalogu `parameters_config`
3. Wykonuje kolejne etapy algorytmu wnioskowania:
   - *fuzzyfikację*
   - *wnioskowanie*
   - *agregację*
   - *defuzyfikację* w obu wariantach (Typ 1 i Typ 2).
4. Prezentuje w konsoli szczegółowy przebieg: stopnie przynależności wejść, stopnie aktywacji reguł, przynależności wyjść oraz końcowe wartości decyzji.
5. Zapisuje informacje do pliku `results/result.txt` w postaci dziennika przebiegu wraz z informacją o czasie wykonania.

Dzięki temu każdorazowe uruchomienie pozwala zarówno na przejrzyste zobaczenie wszystkich pośrednich wyników w konsoli, jak i na późniejszą analizę kompletnego logu w pliku wynikowym.

6.1 Przykładowe uruchomienie

Aby zobrazować działanie systemu, wykonano następujące polecenie w konsoli:

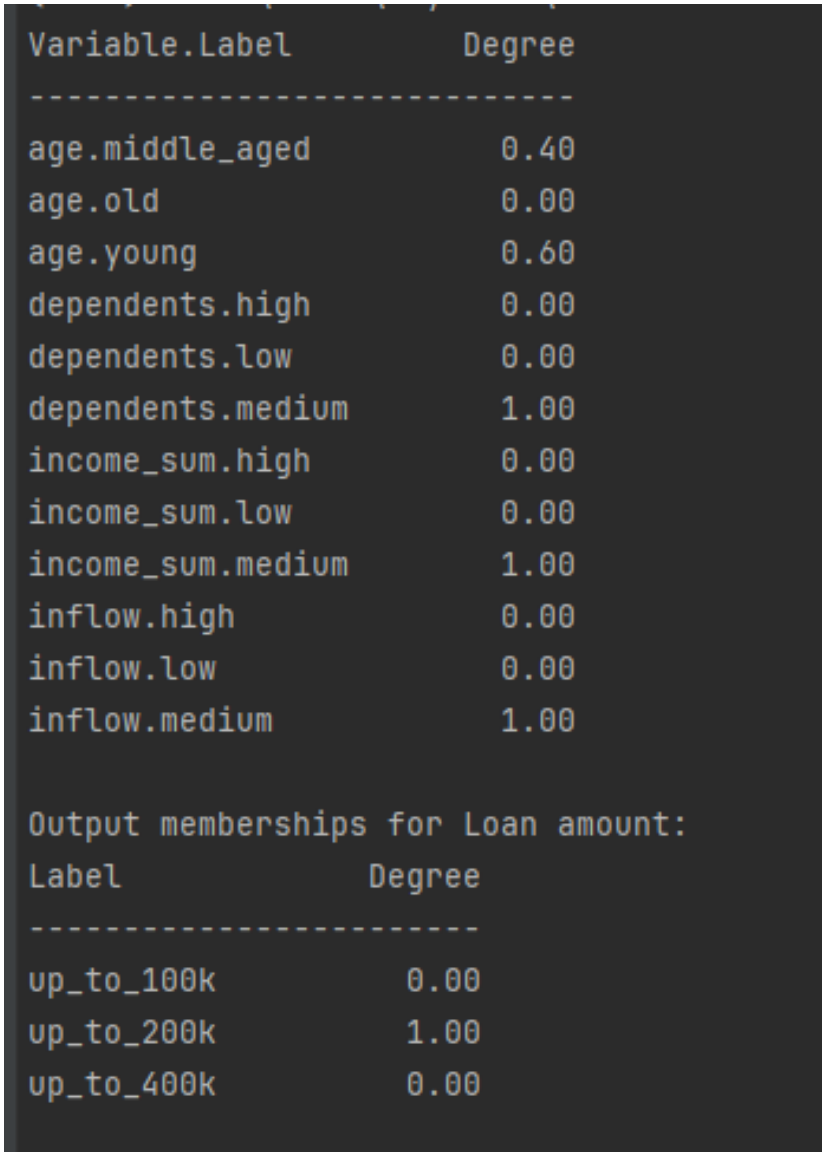
```
python fuzzy_banker.py --inflow 4500 --income_sum 5100 --dependents 2 --age 36
```

Po chwili program zwrócił szczegółowy przebieg obliczeń. Na Rysunku 5 widać fragment wyjścia konsoli, zawierający:

- stopnie przynależności wartości wejściowych do etykiet lingwistycznych,
- stopnie aktywacji poszczególnych reguł rozmytych α_k,

Następnie, podsumowanie końcowych wartości defuzyfikacji obu typów zostało wyświetlone w formie tabeli (Rysunek 6):

Dla celów audytu i analizy szczegółowej, cała sesja została zapisana do pliku `results/result.txt`. Na Rysunku 7 pokazano wycinek tego dziennika, zawierający pełne logi wszystkich etapów wykonywania programu.



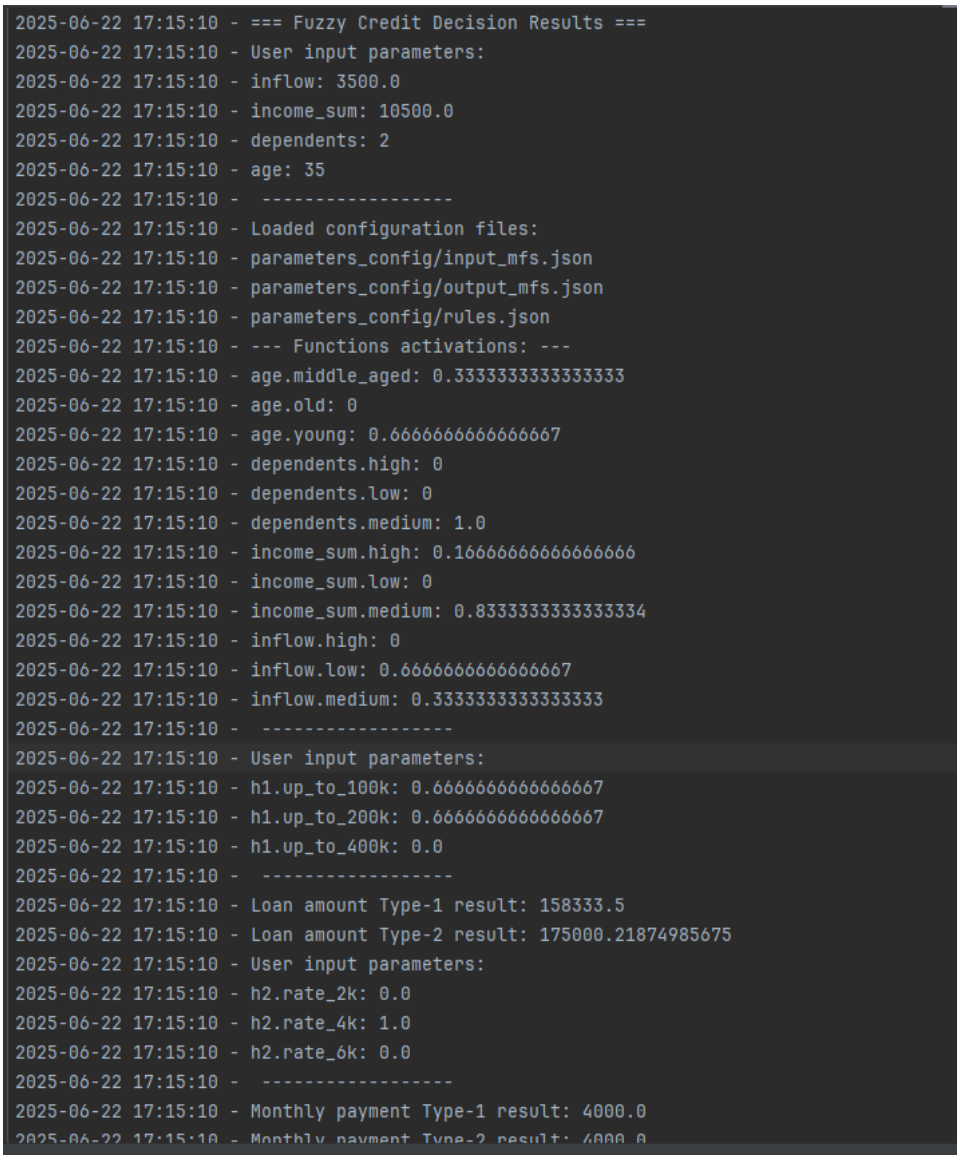
```
Variable.Label          Degree
-----------------------------
age.middle_aged           0.40
age.old                   0.00
age.young                 0.60
dependents.high           0.00
dependents.low            0.00
dependents.medium         1.00
income_sum.high           0.00
income_sum.low            0.00
income_sum.medium         1.00
inflow.high               0.00
inflow.low                0.00
inflow.medium             1.00

Output memberships for Loan amount:
Label              Degree
------------------------
up_to_100k           0.00
up_to_200k           1.00
up_to_400k           0.00
```

Rysunek 5: Fragment wyjścia konsoli pokazujący stopnie przynależności i aktywacji reguł.

Key	Decision	Type-1	Type-2
h1	Loan amount	230000.00	233333.33
h2	Monthly payment	4000.00	4000.00
h3	Loan term	22.10	23.33

Rysunek 6: Końcowe wyniki defuzyfikacji typu 1 i typu 2 dla trzech decyzji (`h1`, `h2`, `h3`).



Rysunek 7: Fragment pliku `result.txt` z kompletnym dziennikiem wykonania.

7 Analiza wyników

Przeprowadzono serię eksperymentów, w których kolejno zmieniano pojedyncze wartości wejściowe — średni przychód (`inflow`), sumę dochodów (`income_sum`), liczbę osób na utrzymaniu (`dependents`) oraz wiek (`age`) — przy czym pozostałe parametry pozostawały niezmienne. Wyniki zestawiono dla obu metod defuzyfikacji w Tabeli 1.

	Parametry wejściowe				Wyniki (h_1, kwota kredytu)	
Scenariusz	inflow	income_sum	dependents	age	Typ 1	Typ 2
1. bazowy	5500	5500	1	25	225 000	233 333
2. obniżony `inflow`	3500	5500	1	25	153 572	168 966
3. podwyższony `income_sum`	3500	10 500	1	25	175 000	191 228
4. więcej `dependents`	3500	10 500	2	25	160 185	180 000
5. większy `age`	3500	10 500	2	35	158 334	175 000

Tabela 1: Wpływ pojedynczych zmian parametrów na wartość rekomendowanej kwoty kredytu h_1.

					Wyniki					
	Parametr wejściowy				Typ 1			Typ 2		
	Przychód	Przychód łączny	Ilość osób na utrzymaniu	Wiek	h1 (wysokość)	h2 (rata)	h3 (termin)	h1 (wysokość)	h2 (rata)	h3 (termin)
Wartość	5500	5500	1	25	225000.00	6000.00	35.00	233333.33	5333.33	32.22
	3500	5500	1	25	**153571.62**	6000.00	35.00	**168965.75**	5333.33	32.22
	3500	**10500**	1	25	**175000.13**	**5697.53**	35.00	**191228.27**	**4333.33**	32.22
	3500	10500	**2**	25	**160185.33**	**4000.00**	**34.17**	**180000.21**	**4000.00**	32.22
	3500	10500	2	**35**	**158333.50**	4000.00	**28.89**	**175000.22**	4000.00	**29.17**

Rysunek 8: Wpływ pojedynczych zmian parametrów na wartość rekomendowanych h1 h2 h3 porównując typ 1 oraz 2.

Wpływ poszczególnych zmiennych

- Obniżenie średniego przychodu z 5500 do 3500 zł skutkuje znacznym spadkiem obu rekomendacji, przy czym *Typ 2* wykazuje łagodniejszą redukcję (–28%) niż *Typ 1* (–32%).
- Zwiększenie sumy dochodów do 10 500 zł podnosi wartość h_1 aż do około 175 000–191 000 zł, co ilustruje silny wpływ `income_sum` w obu schematach.
- Rozrost liczby osób na utrzymaniu powoduje spadek rekomendowanej kwoty i wysokości raty; oba typy dają zbliżone wyniki, choć *Typ 2* jest nieco bardziej konserwatywny.
- Wzrost wieku klienta z 25 do 35 lat obniża proponowany okres spłaty (h_3) i wpływa na zmniejszenie h_1, co widać wyraźniej w *Typie 2*.

Różnice między Typem 1 i Typem 2 *Typ 1* stosuje średnią ważoną reprezentantów etykiet i często daje wartości „skokowe” zgodnie z aktywowanymi regułami. *Typ 2* agreguje skalowane funkcje przynależności i wylicza centroid sumarycznej krzywej, co skutkuje łagodniejszymi zmianami i pewnym „wygładzeniem” rekomendacji.

Wnioski Z przeprowadzonych badań empirycznych wynika, że kluczowym czynnikiem wpływającym na poprawność i stabilność rekomendacji jest *staranna konstrukcja bazy reguł*. W praktyce odpowiedzialność za definiowanie właściwych formuł IF–THEN spoczywa na ekspertach dziedzinowych, a zadaniem programisty jest zapewnienie, by mechanizm wnioskowania przetwarzał je bezbłędnie. Dobre reguły pozwalają uzyskać wiarygodne wyniki niezależnie od wybranej metody defuzyfikacji przy czym typ 2 wydaje się dawać częściej bardziej dokładne wyniki tj. wartości niecałkowite.

8 Porównanie własnej implementacji z biblioteką

Dla zweryfikowania poprawności i efektywności naszej implementacji *defuzyfikacji typu 2* przygotowano oddzielny skrypt `fuzzy_banker_type2_lib_comparison.py`, działający analogicznie do głównego programu `fuzzy_banker.py`, jednak zamiast porównywać typ 1 i typ 2, od razu zestawia wnioski z naszej własnej metody z wynikami otrzymanymi za pomocą biblioteki `scikit-fuzzy`. (Niestety porównanie typu 1 okazało się niemożliwe z powodu braku dostępnych gotowych metod w zewnętrznych bibliotekach.) Program przyjmuje te same argumenty:

```
python fuzzy_banker_type2_lib_comparison.py \
-inflow 12000 -income_sum 12000 -dependents 0 -age 24
```

lub uruchomienie z pliku wsadowego `example_run_comparison_type2_and_lib.bat`. Przykładowy rezultat:

```
Key  Decision            Custom T2   Library T2
-------------------------------------------------
h1   Loan amount        322222.22    322222.22
h2   Monthly payment       777.78       779.33
h3   Loan term              7.78        12.22
```

Rysunek 9: Przykładowy rezultat wywołania programu fuzzy_banker_type2_lib_comparison.py

W tabeli 2 zestawiono wartości wyjściowe dla kilku zestawów danych wejściowych:

Dane wejściowe				**Typ 2 (własna)**			**Typ 2 (scikit-fuzzy)**		
Inflow	Income sum	Dependents	Age	h_1	h_2	h_3	h_1	h_2	h_3
5500	5500	1	25	233333.33	5333.33	32.22	233332.89	5333.33	32.22
10500	12500	1	25	307407.41	2600.00	32.22	307407.41	2603.12	32.22
10500	12500	1	45	322222.22	3641.98	20.00	322222.22	3642.77	20.00
14000	20000	1	80	322222.22	2600.00	7.78	322222.22	2603.12	12.22
14000	20000	8	35	322222.22	777.78	7.78	322222.22	779.33	12.22

Tabela 2: Porównanie wyników defuzyfikacji typu 2: własna implementacja kontra `scikit-fuzzy`

					Wyniki					
	Parametr wejściowy				Typ 2 (Własna implementacja)			Typ 2 (Biblioteka - skfuzzy)		
	Przychód	Przychód łączny	Ilość osób na utrzymaniu	Wiek	h1 (wysokość)	h2 (rata)	h3 (termin)	h1 (wysokość)	h2 (rata)	h3 (termin)
Wartość	5500	5500	1	25	**233333.33**	5333.33	32.22	**233332.89**	5333.33	32.22
	10500	12500	1	25	307407.41	**2600.00**	32.22	307407.41	**2603.12**	32.22
	10500	12500	1	45	322222.22	**3641.98**	20.00	322222.22	**3642.77**	20.00
	14000	20000	1	80	322222.22	**2600.00**	**7.78**	322222.22	**2603.12**	**12.22**
	14000	20000	8	35	322222.22	**777.78**	**7.78**	322222.22	**779.33**	**12.22**

Rysunek 10: Tabela z wynikami porównania typu 2 we własnej implementacji oraz bibliotecznej

Obie metody wykorzystują centroidalną formę defuzyfikacji:

$$y^* = \frac{\int y\,\mu_{\text{sum}}(y)\,\mathrm{d}y}{\int \mu_{\text{sum}}(y)\,\mathrm{d}y},$$

jednak różnią się w szczegółach numerycznej realizacji:

- **Własna implementacja** dzieli zagregowaną krzywą na segmenty i wyznacza środek masy każdego prostokąta i trójkąta analitycznie,
- `scikit-fuzzy` oblicza wartości węzłowe $\mu_{\text{sum}}(y)$ na jednorodnej siatce 1001 (typowo) punktów, a następnie stosuje wbudowaną funkcję `fuzz.defuzz(...,'centroid')` – de facto aproksymując całki sumą prostokątów.

Wyniki z biblioteki i własnej metody zgadzają się w większości przypadków z dokładnością rzędu setnych części jednostki, odchylenia zaś wynikają głównie z różnej gęstości próbkowania osi y.

Podsumowując, biblioteka `scikit-fuzzy` i nasza implementacja dają porównywalne wyniki – różnice pozostają na poziomie błędu numerycznego. W praktyce to jakość i kompletność bazy reguł (a nie sam algorytm defuzyfikacji) ma największy wpływ na końcową rekomendację.

9 Podsumowanie

Logika rozmyta stanowi interesującą alternatywę dla klasycznych metod podejmowania decyzji — pozwala operować na stopniach przynależności zamiast na sztywnych progach binarnych, co bliższe jest ludzkiemu sposobowi rozumowania w warunkach niepewności. W ramach projektu powstało narzędzie CLI, które przyjmuje wartości wejściowe użytkownika (`-inflow`, `-income_sum`, `-dependents`, `-age`), umożliwia elastyczną modyfikację funkcji przynależności i bazy reguł przez pliki JSON, a następnie krok po kroku prezentuje przebieg obliczeń — od fuzzyfikacji, przez wnioskowanie i agregację, aż po defuzyfikację. Szczegółowy log procesu oraz końcowe wyniki zapisywane są automatycznie w pliku `results/result.txt`.

Podczas implementacji największym wyzwaniem było zapewnienie monotoniczności funkcji wyjściowych, co wymagało podziału „górkowych" etykiet na monotoniczne segmenty w metodzie typu 1 oraz precyzyjne rozłożenie pola pod zagregowaną krzywą na fragmenty prostokątne i trójkątne w metodzie typu 2. Porównanie z centroidalną defuzyfikacją oferowaną przez bibliotekę `scikit-fuzzy` wykazało, że otrzymywane różnice mieszczą się w akceptowalnym marginesie błędu numerycznego.

Doświadczenia przeprowadzone w trakcie pracy nad projektem potwierdziły, że kluczową rolę odgrywa jakość i kompletność bazy reguł. To właśnie dobrze dobrane warunki w pliku `rules.json`, odzwierciedlające rzeczywiste scenariusze decyzyjne, decydują o tym, czy system wygeneruje użyteczną rekomendację. W praktyce oznacza to, że specjaliści domenowi powinni projektować reguły, pozostawiając programistom jedynie implementację algorytmów.

Reasumując, projekt dowiódł, że logikę rozmytą można skutecznie zaimplementować w Pythonie, uzyskując narzędzie pozwalające na szybkie definiowanie i testowanie funkcji przynależności oraz reguł, a jednocześnie osiągające wyniki porównywalne z bibliotekami referencyjnymi.