

Wstęp do Sztucznej Inteligencji

Ewolucyjne podejście do optymalizacji klasyfikatora opartego na zestawie reguł rozmytych

Celem laboratorium jest implementacja klasyfikatora opartego na regułach rozmytych. Postać reguł rozmytych może być optymalizowana za pomocą algorytmu ewolucyjnego.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import copy
```

Zacniemy od prostego problemu klasyfikacyjnego irysów. Dane te opisują 150 przykładów kwiatów, każdy należący do jednego z trzech gatunków. Mamy zatem do czynienia z problemem klasyfikacyjnym z trzema klasami.

Każdy przykład opisany jest czterema atrybutami (cechami). Na ich podstawie nasz klasyfikator ma podejmować decyzję, do której klasy przypisać dany przykład.

Proces uczenia takiego modelu to przykład uczenia z nauczycielem (ang. supervised learning).

```
from sklearn.preprocessing import LabelEncoder

data_path = 'iris.csv'
data = pd.read_csv(data_path, header=None, delimiter=r"\s+")
data.columns = ['SepalLength', 'SepalWidth', 'PetalLength',
                'PetalWidth', 'Species']
print(data.head())

# encode as np matrix
X = features_matrix = data.iloc[:, 0:4].to_numpy()
print(X.shape)
Y = data['Species'].to_numpy()
label_encoder = LabelEncoder()
Y = label_encoder.fit_transform(Y)
print(Y)
```

	SepalLength	SepalWidth	PetalLength	PetalWidth	Species
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa

(150, 4)

[illegible]

Po wczytaniu danych, warto się im przyjrzeć i ocenić trudność problemu.

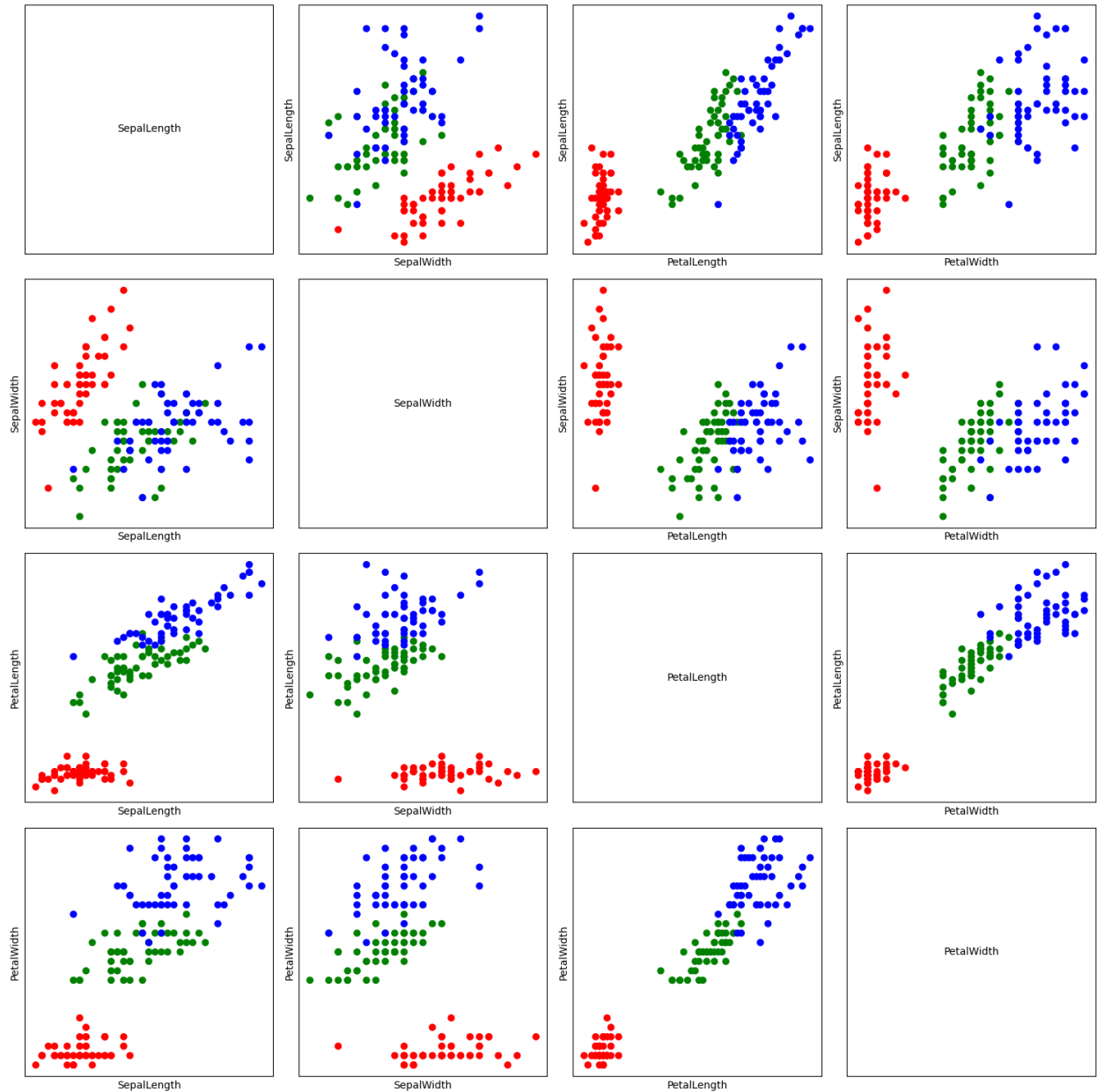
Które atrybuty wydają się szczególnie przydatne w tym problemie? Zwróć uwagę, że przy takiej wizualizacji danych nie widzimy dokładnie jak dobrze te dane są odseparowane w oryginalnej przestrzeni (w tym przypadku 4D).

```

colors = ['red', 'green', 'blue']
data['Species'] = Y
# Rysowanie wykresów dla każdej pary cech
fig, axs = plt.subplots(4, 4, figsize=(15, 15)) # Ustawienie siatki
# wykresów
for i in range(4):
    for j in range(4):
        if i != j:
            axs[i, j].scatter(data.iloc[:, j], data.iloc[:, i],
c=data['Species'].map(lambda x: colors[x]))
            axs[i, j].set_xlabel(data.columns[j])
            axs[i, j].set_ylabel(data.columns[i])
        else:
            axs[i, j].text(0.5, 0.5, data.columns[i],
horizontalalignment='center', verticalalignment='center')
            axs[i, j].set_xlabel('')
            axs[i, j].set_ylabel('')
            axs[i, j].set_xticks([])
            axs[i, j].set_yticks([])

plt.tight_layout()
plt.show()

```



Dane dzielimy na dane trenujące i testowe. Klasyfikator powinien być przetestowany na danych, z którymi nie miał styczności w czasie procesu uczenia.

```
test_perc = 0.3
idxs = np.random.permutation(X.shape[0])
ii = int(test_perc*X.shape[0])
Xtest = X[idxs[:ii]]
Ytest = Y[idxs[:ii]]
X = X[idxs[ii:]]
Y = Y[idxs[ii:]]
print(X.shape, Xtest.shape)
```

```
(105, 4) (45, 4)
```

Do zdefiniowania funkcji przynależności potrzebne będą zakresy wartości zmiennych.

```
mins = X.min(axis=0)
maxs = X.max(axis=0)
print(mins)
print(maxs)

[4.3 2.  1.  0.1]
[7.9 4.4 6.9 2.5]
```

Nasz przykładowy klasyfikator rozmyty będzie korzystał z trójkątnych funkcji przynależności.

```
def trfun(x, a, b, c):
    return 0 if (x < a or x > c) else ((x-a)/(b-a) if (x >= a and x <=
b) else (c-x)/(c-b))
```

Klasyfikator rozmyty podejmuje decyzję na podstawie zbioru reguł rozmytych w postaci:

JEŚLI x1 jest MAŁE i x2 jest ŚREDNIE i x4 jest DUŻE wtedy KLASA_1

Zwróć uwagę, że w powyższym przykładzie nie jest wykorzystane x3.

Obiekt poniższej klasy FuzzyClassifier przechowuje dla każdego wymiaru (x1 do x4 w przypadku irysów) zadaną liczbę wartości lingwistycznych reprezentowanych przez trójkątne funkcje przynależności definiowane przez trzy parametry. Dla każdego wymiaru może to być różna liczba (parametr fnums).

Reguły rozmyte zakodowane są w polu rules. Każda reguła to lista, której kolejne wartości wskazują, do której funkcji przynależności odnosi się ta reguła w danym wymiarze, a ostatni element to indeks klasy z wniosku reguły. Jeśli występuje wartość -1, to ten atrybut nie jest brany pod uwagę przez daną regułę.

Przykład:

```
[-1, -1, 0, 2, 2]
```

Reguła ta odnosi się do x3 (pierwsza funkcja przynależności) oraz x4 (trzecia funkcja przynależności). We wniosku wskazuje klasę o indeksie 2 (trzecią).

Podczas klasyfikacji sprawdzane są wartości odpowiednich funkcji przynależności dla konkretnych wartości x1 do x4. Stopień spełnialności części warunkowej danej reguły określany jest za pomocą odpowiedniej t-normy (minimum w tym przykładzie). Jeśli więcej niż jedna reguła ma we wniosku daną klasę, wartości stopni spełnialności warunków tych reguł agregowane są za pomocą odpowiedniej s-normy (maksimum w tym przykładzie). Ostatecznie wybierana jest klasa, na którą wskazuje maksymalna wartość.

```
class FuzzyClassifier:
    def __init__(self, nclass, dim, fnums, mins, maxs):
```

```

self.nclass = nclass #number of classes
self.dim = dim #number of input features
self.fnums = fnums #list: number of linguistic values for each
feature
self.mins = mins #minimum values of features
self.maxs = maxs #maximum values of features
self.funs = None #parametry funkcji przynależności (tu:
trójkątnej)
self.rules = None #reguły
self.tnorm_condition = None #t-norma używana do oceny części
warunkowej reguły
self.snorm_aggregation = None #s-norma używana do agregacji

def init_triangular_funs(self):
    """
    Dla każdego wymiaru generuje zadaną liczbę równomiernie
    rozłożonych trójkątnych funkcji przynależności
    """
    self.funs = []
    for i in range(self.dim):
        step = (self.maxs[i] - self.mins[i]) / (self.fnums[i] - 1)
        tmp = []
        for j in range(self.fnums[i]):
            center = self.mins[i] + j * step
            tmp.append((center - step, center, center + step))
        self.funs.append(tmp)

def gen_random_rules(self, n):
    """
    Generuje zadaną liczbę losowych reguł
    """
    return [[np.random.randint(-1, num) for num in self.fnums] +
            [np.random.randint(0, nclass)] for i in range(n)]

def classify(self, x):
    """
    Klasyfikuje pojedynczy przykład x
    """
    votes = [[] for i in range(self.nclass)]
    for r in self.rules:
        conditions = []
        for i, findx in enumerate(r[:-1]):
            if findx == -1:
                continue
            a, b, c = self.funs[i][findx]
            conditions.append(trfun(x[i], a, b, c))
        condition_sat_level = self.tnorm_condition(conditions) if
len(conditions) > 0 else 0
        votes[r[-1]].append(condition_sat_level)

```

```

        votes = [self.snorm_aggregation(v) if len(v) > 0 else 0 for v
in votes]
        return np.argmax(votes)

    def classify_all(self, X):
        """
        Klasyfikuje wszystkie przykłady - wiersze macierzy X
        """
        return np.array([self.classify(x) for x in X])

    def test(self, X, Y):
        """
        Klasyfikuje przykłady z X i porównuje z prawidłowymi
etykietami klas z Y

        Zwraca accuracy - odsetek poprawnych klasyfikacji
        """
        return (Y == self.classify_all(X)).sum()/X.shape[0]

```

Ustawienia dla problemu irysów:

```

nclass = 3 #liczba klas
dim = 4 #liczba atrybutów
fnums = [3, 3, 3, 3] #liczba funkcji przynależności (wartości
lingwistycznych) dla każdego atrybutu

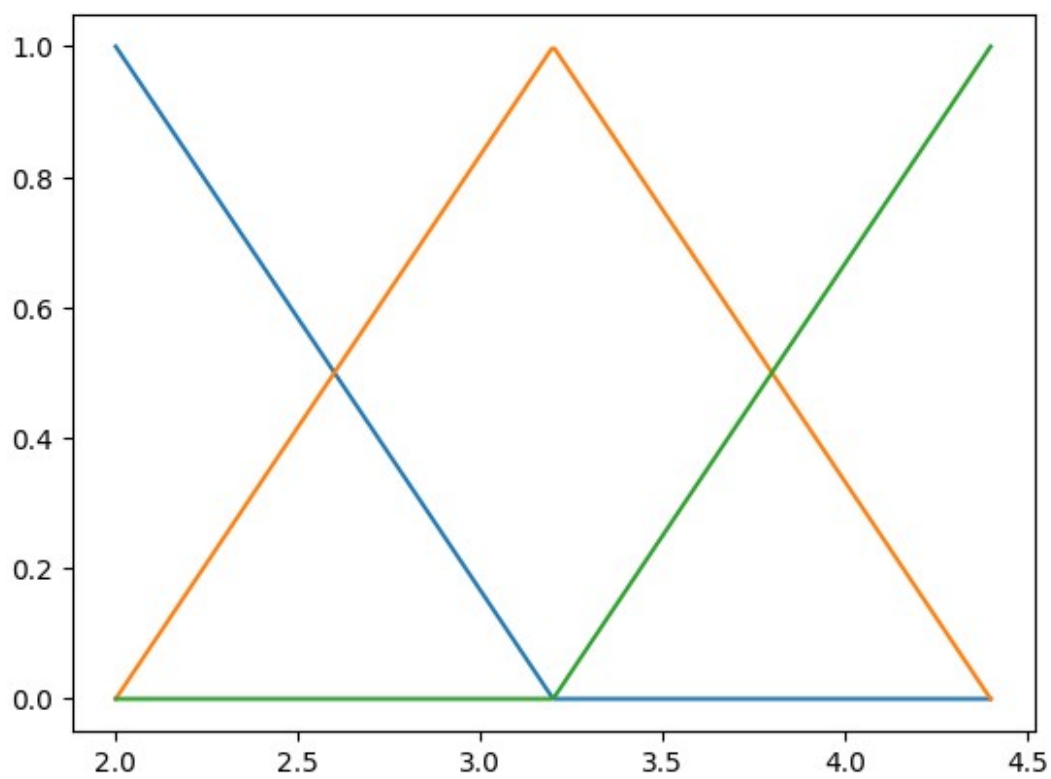
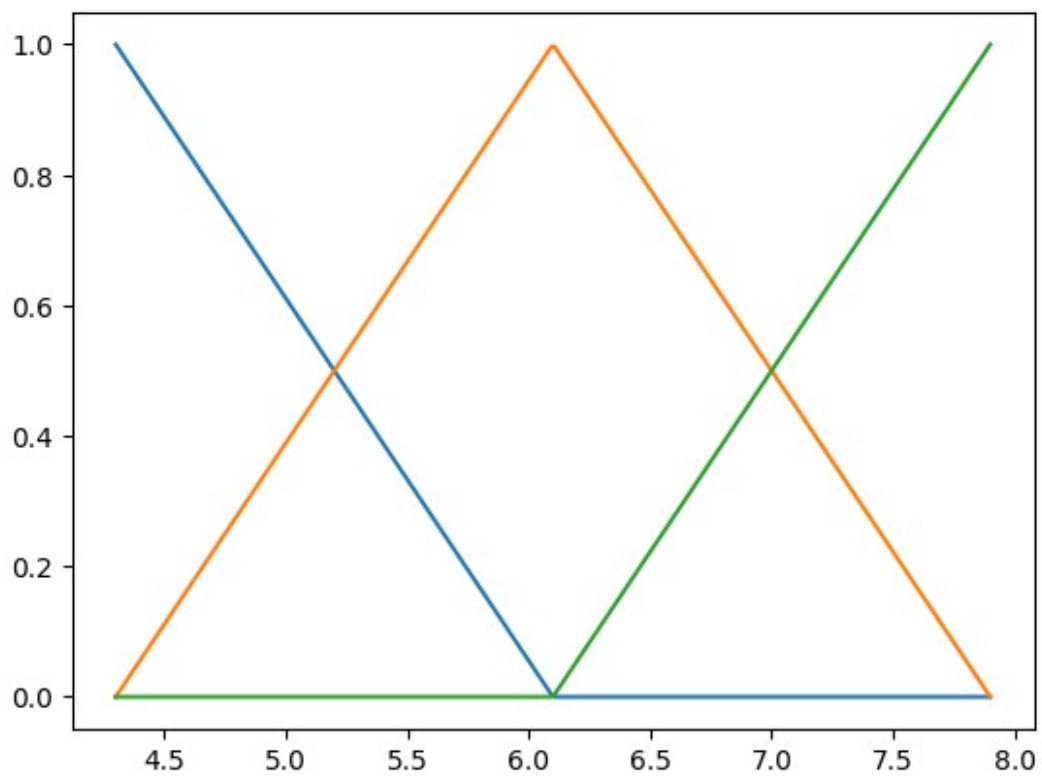
```

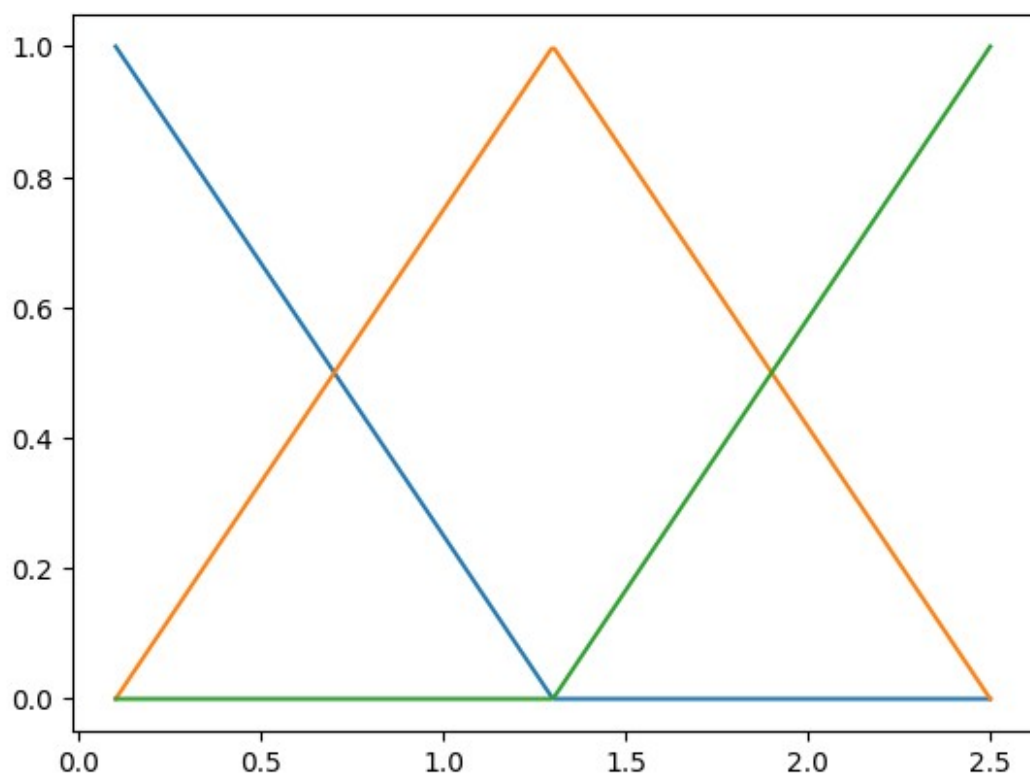
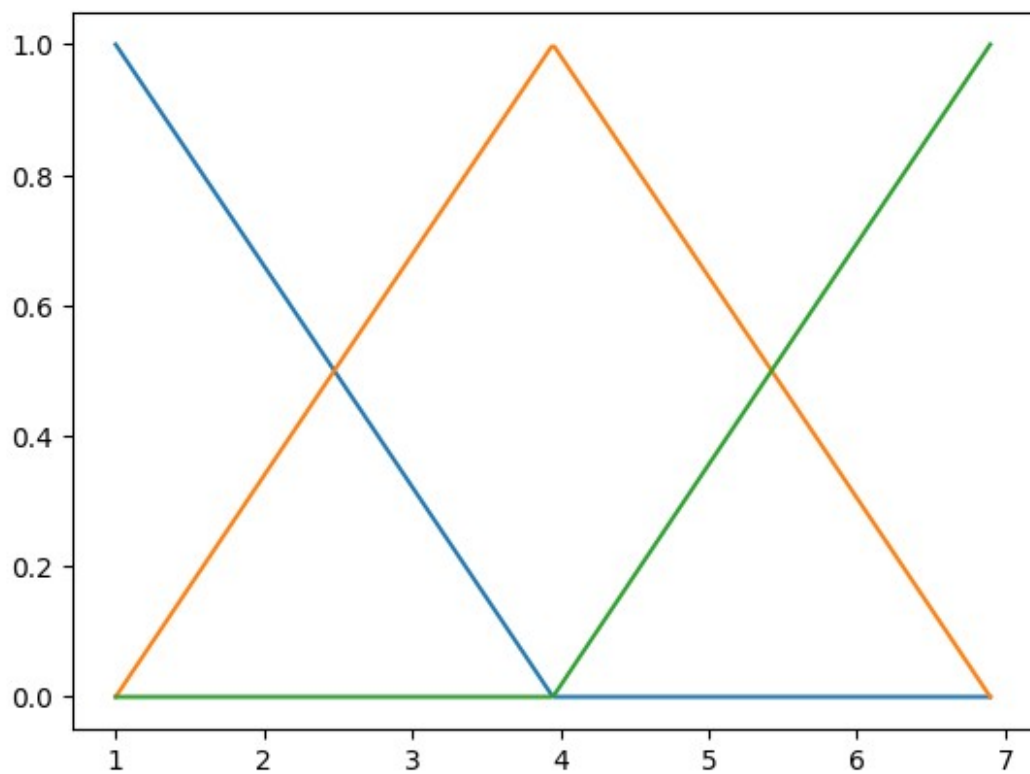
Tworzymy obiekt FuzzyClassifier i wizualizujemy funkcje przynależności.

```

fc = FuzzyClassifier(nclass, dim, fnums, mins, maxs)
fc.init_triangular_funs()
for i, f in enumerate(fc.funs):
    xx = np.linspace(mins[i], maxs[i], 300)
    plt.figure()
    for ff in f:
        a, b, c = ff
        mfun = np.vectorize(lambda x: trfun(x, a, b, c),
otypes=[float])
        yy = mfun(xx)
        plt.plot(xx, yy)
plt.show()

```





Generujemy przykładowe reguły


```

r = fc.gen_random_rules(3)
print(r)

# Przy ustawieniach fnums = [5, 5, 5, 5] następujące reguły dają
całkiem niezłe działający klasyfikator
# fc.rules = [[-1, -1, -1, 4, 2], [-1, -1, -1, 3, 2], [-1, -1, -1, 2,
1], [-1, -1, 1, -1, 0]]

[[1, 0, 1, 1, 0], [1, 2, 0, 2, 0], [0, 1, -1, 2, 1]]

```

Ustawiamy t-normę i s-normę

```

fc.tnorm_condition = lambda x: min(x)
fc.snorm_aggregation = lambda x: max(x)

```

Test klasyfikatora

```

fc.rules = r
acc = fc.test(X, Y)
print(acc)
Z = fc.classify_all(X)
print(Z)

0.37142857142857144
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0
0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0
0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0]

```

Mutacja będzie polegać na zmianie aktualnej wartości na losową, dozwoloną na danej pozycji (z pewnym prawdopodobieństwem).

```

def mutate(rule_set, probab_mut):
    """
    Mutacja poprzez zmianę na losową wartość dopuszczalną na danej
    pozycji.
    Klasa z wniosku reguły również może się zmienić.
    """
    for r in rule_set:
        for i in range(fc.dim):
            if np.random.rand() < probab_mut:
                r[i] = np.random.randint(-1, fc.fnums[i])
            if np.random.rand() < probab_mut:
                r[-1] = np.random.randint(0, fc.nclass)

```

Generujemy losową początkową populację.

Każdy osobnik koduje cały zestaw reguł rozmytych.

Wszystkie osobniki korzystać będą z tego samego zestawu funkcji przynależności.

```
nrules = 5
pop_size = 10
pop = [fc.gen_random_rules(nrules) for i in range(pop_size)]
print(pop)

[[[1, 2, 0, 1, 0], [2, 0, -1, 2, 2], [1, 2, 1, -1, 2], [2, 1, -1, 1, 2], [1, -1, 2, 0, 2]], [[1, 0, -1, -1, 2], [-1, 0, -1, 1, 0], [-1, 2, -1, -1, 0], [-1, 1, 2, 0, 2], [0, 1, 0, 2, 2]], [[-1, 1, 2, 2, 1], [-1, 0, 2, 1, 2], [-1, 0, -1, 2, 1], [1, 0, 0, -1, 0], [1, 1, 1, 0, 0]], [[2, 1, 0, -1, 2], [1, 1, 0, 1, 0], [1, 0, 2, -1, 1], [0, 0, 0, -1, 0], [1, 2, 2, 1, 0]], [[0, 2, 1, -1, 2], [1, 2, 2, 1, 0], [-1, 1, 0, 1, 1], [1, -1, -1, 1, 1], [2, 1, -1, 2, 1]], [[1, 0, 0, 0, 2], [0, 1, 1, 0, 0], [0, 0, -1, 2, 1], [2, -1, -1, -1, 1], [-1, 0, 0, 2, 2]], [[0, 2, -1, 1, 2], [0, 0, -1, 1, 2], [1, -1, 1, -1, 1], [2, 2, 1, 2, 1], [2, 1, 0, 1, 0]], [[0, 2, 0, 1, 1], [-1, 1, 2, -1, 2], [1, 1, -1, 0, 0], [1, 0, 2, 1, 0], [1, -1, 0, -1, 2]], [[-1, 2, 2, 1, 0], [-1, 0, -1, 2, 1], [1, -1, 1, 2, 1], [0, 0, 2, 0, 2], [2, -1, 2, -1, 0]], [[0, -1, 0, 2, 0], [-1, 1, 1, 0, 1], [-1, 2, 0, 2, 2], [0, 0, 0, 2, 0], [1, 1, 2, 1, 2]]]
```

Ewaluacja osobników i zapamiętanie najlepszego.

Funkcją przystosowania jest odsetek poprawnych klasyfikacji na zbiorze trenującym.

```
evals = []
for rule_set in pop:
    fc.rules = rule_set
    evals.append(fc.test(X, Y))
print(evals)

srt = sorted(zip(evals, pop))[:, -1]
best_rs = copy.deepcopy(srt[0][1])
best_eval = srt[0][0]
print('best ', best_eval)
print(best_rs)

[0.4857142857142857, 0.3333333333333333, 0.4380952380952381,
0.5904761904761905, 0.3523809523809524, 0.45714285714285713,
0.29523809523809524, 0.5619047619047619, 0.4380952380952381,
0.37142857142857144]
best 0.5904761904761905
[[2, 1, 0, -1, 2], [1, 1, 0, 1, 0], [1, 0, 2, -1, 1], [0, 0, 0, -1, 0], [1, 2, 2, 1, 0]]
```

Przykładowy algorytm ewolucyjny jest bardzo prosty.

Jego jedyny operator to mutacja.

Każdy z osobników z aktualnej populacji jest kopiowany a następnie mutowany.

Stara i nowa (zmutowana) populacja są łączone.

Pozostawiamy pop_size najlepszych osobników (nacisk selekcyjny).

Przetestujmy jego działanie.

```
iters = 100
prob_mut = 0.1

for i in range(iters):
    # kopiowanie każdego osobnika
    pop2 = [copy.deepcopy(rs) for rs in pop]
    # mutacja
    for rule_set in pop2:
        mutate(rule_set, prob_mut)
    # ocena nowych rozwiązań
    evals2 = []
    for rule_set in pop2:
        fc.rules = rule_set
        evals2.append(fc.test(X, Y))
    # łączenie obu populacji
    pop = pop + pop2
    evals = evals + evals2
    # pozostawienie stałej liczby najlepszych osobników
    srt = sorted(zip(evals, pop))[:-1]
    pop = [s[1] for s in srt][:pop_size]
    evals = [s[0] for s in srt][:pop_size]
    # zapamiętanie najlepszego rozwiązania
    if evals[0] > best_eval:
        print('better found', evals[0], 'in', i+1)
        best_eval = evals[0]
        best_rs = copy.deepcopy(pop[0])
print('end')

better found 0.7714285714285715 in 3
better found 0.780952380952381 in 5
better found 0.819047619047619 in 7
better found 0.8476190476190476 in 9
better found 0.8952380952380953 in 11
better found 0.9047619047619048 in 14
better found 0.9142857142857143 in 17
better found 0.9333333333333333 in 22
better found 0.9619047619047619 in 60
end
```

Spradzamy odpowiedzi najlepszego osobnika na zbiorze trenującym:

```
print(best_eval)
print(best_rs)
fc.rules = best_rs
```

```

z = fc.classify_all(X)
print(z)
print(Y)
print(fc.test(X, Y))

0.9619047619047619
[[1, -1, 1, 1, 1], [-1, 1, 2, 1, 2], [1, -1, -1, 0, 0], [0, 1, 2, 2,
2], [1, -1, 1, -1, 2]]
[0 1 1 0 1 2 2 1 2 0 0 2 2 0 0 0 0 1 2 0 2 2 0 1 1 1 2 0 2 2 0 1 2 2 1
1 0
 2 1 0 2 2 0 0 2 0 1 1 1 0 2 2 2 2 0 1 1 0 2 2 0 1 1 2 2 0 2 2 1 0 0 0
1 0
 2 2 0 1 1 1 0 2 2 2 2 1 0 1 0 0 2 2 0 0 1 1 1 1 1 0 1 0 0 2 0]
[0 1 1 0 1 2 2 1 2 0 0 2 2 0 0 0 0 1 2 0 2 2 0 1 1 1 2 0 2 2 0 1 2 1 1
1 0
 2 1 0 2 2 0 0 2 0 1 1 1 0 2 2 2 2 0 1 1 0 2 2 0 2 1 2 2 0 2 2 1 0 0 0
1 0
 2 2 0 1 1 1 0 1 2 2 2 1 0 1 0 0 2 2 0 0 1 1 1 1 1 0 1 0 0 1 0]
0.9619047619047619

```

...oraz testowym.

```

# testing
ans = fc.classify_all(Xtest)
print(ans)
print(Ytest)
print(fc.test(Xtest, Ytest))

[1 1 0 0 0 0 2 1 2 1 2 2 1 0 0 2 1 2 1 2 2 1 2 0 0 0 1 0 2 2 1 2 0 2 2
2 1
 2 1 1 1 1 0 0 2]
[1 1 0 0 0 0 2 1 2 1 1 2 1 0 0 2 1 2 1 2 2 1 2 0 0 0 1 0 2 2 2 2 0 1 2
1 2
 2 1 1 1 1 0 0 2]
0.8888888888888888

```

Wykonaj przykładowe obliczenia dla różnych ustawień (liczba funkcji przynależności, liczba reguł, wielkość populacji itd.)

Zadanie (10 punktów)

```

nr_indeksu = 142706

print(r"""
Zaimplementuj własny algorytm ewolucyjny do optymalizacji
klasyfikatora rozmytego.

Przetestuj go na trudniejszym problemie klasyfikacyjnym diagnozy
cukrzycy (baza pima - ostatnia kolumna zawiera etykietę klasy).

```

Ile minimalnie reguł jest potrzebnych?

Wnioski zaprezentuj na podstawie średniej z co najmniej 10 uruchomień.

W algorytmie dodaj możliwość ewolucji parametrów użytych funkcji przynależności (a więc nie są one wspólne dla wszystkich rozwiązań lecz podlegają optymalizacji).

Porównaj działanie swojego algorytmu z przykładowym algorytmem z zajęć.

Algorytm powinien mieć następujące funkcjonalności (zastosuj operator krzyżowania nawet jeśli nie występuje on w oryginalnym opisie algorytmu):

```
"""
```

```
algorytm = ['algorytm genetyczny', 'strategia ewolucyjna u+s',  
'strategia ewolucyjna u,s', 'programowanie ewolucyjne']  
selekcja = ['koło ruletki', 'turniejowa', 'rankingowa']  
xover = ['jednopunktowe', 'dwupunktowe', 'jednostajne', 'uśrednianie  
ze współczynnikiem a']  
mutacja = ['rozkład jednostajny', 'rozkład Gaussa']  
memfun = ['klasa pi', 'trapezoidalna', 'dzwonowa', 'Gaussa']  
tnorm = ['minimum', 'iloczyn']  
snorm = ['maksimum', 'suma-iloczyn']  
  
print('Algorytm: ', algorytm[nr_indeksu%len(algorytm)])  
print('Rodzaj selekcji (jeśli dotyczy): ', selekcja[nr_indeksu  
%len(selekcja)])  
print('Krzyżowanie: ', xover[nr_indeksu%len(xover)])  
print('Mutacja wartości rzeczywistych (jeśli nie jest określona przez  
algorytm): ', mutacja[nr_indeksu%len(mutacja)])  
print('Rodzaj funkcji przynależności: ', memfun[nr_indeksu  
%len(memfun)])  
print('Rodzaj t-normy: ', tnorm[nr_indeksu%len(tnorm)])  
print('Rodzaj s-normy: ', snorm[nr_indeksu%len(snorm)])  
  
print(r"""  
W razie wątpliwości - ustal szczegóły z prowadzącym zajęcia.  
""")
```

Zaimplementuj własny algorytm ewolucyjny do optymalizacji klasyfikatora rozmytego.

Przetestuj go na trudniejszym problemie klasyfikacyjnym diagnozy cukrzycy (baza pima - ostatnia kolumna zawiera etykietę klasy).

Ile minimalnie reguł jest potrzebnych?

Wnioski zaprezentuj na podstawie średniej z co najmniej 10 uruchomień.

W algorytmie dodaj możliwość ewolucji parametrów użytych funkcji przynależności
(a więc nie są one wspólne dla wszystkich rozwiązań lecz podlegają optymalizacji).

Porównaj działanie swojego algorytmu z przykładowym algorytmem z zajęć.

Algorytm powinien mieć następujące funkcjonalności
(zastosuj operator krzyżowania nawet jeśli nie występuje on w oryginalnym opisie algorytmu):

Algorytm: strategia ewolucyjna u,s
Rodzaj selekcji (jeśli dotyczy): rankingowa
Krzyżowanie: jednostajne
Mutacja wartości rzeczywistych (jeśli nie jest określona przez algorytm): rozkład jednostajny
Rodzaj funkcji przynależności: dzwonowa
Rodzaj t-normy: minimum
Rodzaj s-normy: maksimum

W razie wątpliwości - ustal szczegóły z prowadzącym zajęcia.

Twój kod

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import copy
import random
from sklearn.preprocessing import LabelBinarizer
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split

data =
pd.read_csv('https://raw.githubusercontent.com/jbrownlee/Datasets/
master/pima-indians-diabetes.data.csv', header=None)
x = data.iloc[:, :-1].values
y = data.iloc[:, -1].values

xTrain, xTest, yTrain, yTest = train_test_split(x, y, test_size=0.3,
random_state=42)

def bell(x, a, b, c):
    return 1 / (1 + abs((x - c) / a) ** (2 * b))

class FuzzyClassifier:
    def __init__(self, rules):
```

```

        self.rules = rules

    def classify(self, x):
        results = []
        for rule in self.rules:
            antecedents = [bell(x[i], *params) for i, params in
enumerate(rule['antecedents'])]
            antecedent_value = np.min(antecedents)
            consequent_value = antecedent_value * rule['consequent']
            results.append(consequent_value)
        return np.argmax(np.bincount([int(r > 0.5) for r in results]))

def evolve(population, fitness_fn, mutation_rate=0.1):
    new_population = []
    fitness_scores = [fitness_fn(ind) for ind in population]
    total_fitness = sum(fitness_scores)

    def select_parent():
        pick = random.uniform(0, total_fitness)
        current = 0
        for ind, fitness in zip(population, fitness_scores):
            current += fitness
            if current > pick:
                return ind

    for _ in range(len(population)):
        parent1 = select_parent()
        parent2 = select_parent()
        child = {'rules': []}
        for rule1, rule2 in zip(parent1['rules'], parent2['rules']):
            new_rule = {
                'antecedents': [
                    tuple(random.choice(params) for params in
zip(rule1['antecedents'][i], rule2['antecedents'][i]))
                    for i in range(len(rule1['antecedents']))
                ],
                'consequent': random.choice([rule1['consequent'],
rule2['consequent']])
            }
            if random.random() < mutation_rate:
                for i in range(len(new_rule['antecedents'])):
                    new_rule['antecedents'][i] = tuple(param +
random.uniform(-0.1, 0.1) for param in new_rule['antecedents'][i])
            child['rules'].append(new_rule)
        new_population.append(child)

    return new_population

def fitness_fn(individual):
    classifier = FuzzyClassifier(individual['rules'])

```

```

    predictions = [classifier.classify(x) for x in xTrain]
    return accuracy_score(yTrain, predictions)

def generate_initial_population(size):
    population = []
    for _ in range(size):
        rules = []
        for _ in range(random.randint(1, 10)):
            antecedents = [(random.uniform(0.1, 1.0),
random.uniform(1, 5), random.uniform(0, 1)) for _ in
range(xTrain.shape[1])]
            consequent = random.randint(0, 1)
            rules.append({'antecedents': antecedents, 'consequent':
consequent})
        individual = {'rules': rules}
        population.append(individual)
    return population

population_size = 10
generations = 20
mutation_rate = 0.1

average_accuracies = []
for run in range(10):
    population = generate_initial_population(population_size)
    for gen in range(generations):
        population = evolve(population, fitness_fn, mutation_rate)

    best_individual = max(population, key=fitness_fn)
    best_classifier = FuzzyClassifier(best_individual['rules'])
    test_predictions = [best_classifier.classify(x) for x in xTest]
    accuracy = accuracy_score(yTest, test_predictions)
    average_accuracies.append(accuracy)
    print(f'Run {run + 1}/10')

mean_accuracy = np.mean(average_accuracies)
print(f'Mean Accuracy over 10 runs: {mean_accuracy}')

```

```

Run 1/10
Run 2/10
Run 3/10
Run 4/10
Run 5/10
Run 6/10
Run 7/10
Run 8/10
Run 9/10
Run 10/10

```

```

Mean Accuracy over 10 runs: 0.6536796536796536

```


Twoje sprawozdanie

Katedra Informatyki, Politechnika Krakowska