Systemy inteligentne Projekt

Kod źródłowy

Plik 1

Zapisywanie danych do plików json

import csv  
import json  
import random  
import numpy as np  
from sklearn import preprocessing  
  
file\_1 = open('train\_set.json', "r")  
data1 = json.load(file\_1)  
file\_1.close()  
  
file\_2 = open('test\_set.json', "r")  
data2 = json.load(file\_2)  
file\_2.close()  
  
file\_3 = open('unnormed.json', "r")  
data3 = json.load(file\_3)  
file\_3.close()  
  
parameters\_list = []  
exemplary\_outputs = []  
list\_of\_id = []  
list\_to\_json3 = []  
with open("Raisin\_Dataset2.csv") as csv\_file:  
 csv\_reader = csv.reader(csv\_file, delimiter=';')  
 id = 0  
 for row in csv\_reader:  
 if id == 0:  
 # print(f'Column names are {", ".join(row)}')  
 id += 1  
 else:  
 row[0] = row[0]  
 row[1] = row[1].replace(',', '.')  
 row[2] = row[2].replace(',', '.')  
 row[3] = row[3].replace(',', '.')  
 row[4] = row[4]  
 row[5] = row[5].replace(',', '.')  
 row[6] = row[6].replace(',', '.')  
 if row[7] == 'Kecimen':  
 row[7] = [1, 0]  
 if row[7] == 'Besni':  
 row[7] = [0, 1]  
 parameters\_list.append([float(row[0]), float(row[1]), float(row[2]), float(row[3]), float(row[4]), float(row[5]), float(row[6])])  
 exemplary\_outputs.append(row[7])  
 list\_of\_id.append(id)  
 list\_to\_json3.append([float(row[0]), float(row[1]), float(row[2]), float(row[3]), float(row[4]), float(row[5]), float(row[6]), row[7], id])  
 id += 1  
  
  
  
x = np.array(parameters\_list)  
# x\_normed = x / x.max(axis=0)  
z = preprocessing.normalize(x, norm='max', axis=0)  
x\_normed = z  
  
  
parameters\_list2 = []  
k = 0  
for i in x\_normed:  
 auxiliary\_list = []  
 for j in i:  
 auxiliary\_list.append(j)  
 auxiliary\_list.append(exemplary\_outputs[k])  
 auxiliary\_list.append(list\_of\_id[k])  
 parameters\_list2.append(auxiliary\_list)  
 k += 1  
  
  
random.shuffle(parameters\_list2)  
  
list\_to\_json1 = []  
list\_to\_json2 = []  
counter = 0  
for i in range(len(parameters\_list2)):  
 if counter < 700:  
 list\_to\_json1.append(parameters\_list2[i])  
 counter += 1  
 else:  
 list\_to\_json2.append(parameters\_list2[i])  
  
  
data1["train\_set.json"] = list\_to\_json1  
data2["test\_set.json"] = list\_to\_json2  
data3["unnormed.json"] = list\_to\_json3  
  
g = open("train\_set.json", "w")  
json.dump(data1, g)  
g.close()  
  
g = open("test\_set.json", "w")  
json.dump(data2, g)  
g.close()  
  
g = open("unnormed.json", "w")  
json.dump(data3, g)  
g.close()

Plik 2

Klasyfikacja rodzynek za pomocą dwuwarstwowej sieci neuronowej

#CTM  
from random import random  
from math import exp  
import json  
  
  
file\_1 = open('weight\_neural\_network.json', "r")  
data = json.load(file\_1)  
file\_1.close()  
  
file\_6 = open('besni\_set.json', "r")  
data\_6 = json.load(file\_6)  
file\_6.close()  
  
file\_3 = open('train\_set.json', "r")  
data\_3 = json.load(file\_3)  
train\_dataset = data\_3["train\_set.json"]  
file\_3.close()  
  
w = {  
 'w1': [random()-0.5 for \_ in range(7)],  
 'w2': [random()-0.5 for \_ in range(7)],  
 'w3': [random()-0.5 for \_ in range(7)],  
 'w4': [random()-0.5 for \_ in range(7)],  
 'w5': [random()-0.5 for \_ in range(7)],  
  
 'w6': [random()-0.5 for \_ in range(5)],  
 'w7': [random()-0.5 for \_ in range(5)],  
  
  
 'w10': [random()-0.5],  
 'w20': [random()-0.5],  
 'w30': [random()-0.5],  
 'w40': [random()-0.5],  
 'w50': [random()-0.5],  
  
 'w60': [random()-0.5],  
 'w70': [random()-0.5]  
}  
  
v = {  
 'v1': [],  
 'v2': [],  
 'v3': [],  
 'v4': [],  
 'v5': []  
}  
  
y = {  
 'y1': [],  
 'y2': []  
}  
all\_gradients = {}  
  
error = {  
 'error': []  
}  
  
  
def activate(weights, inputs):  
 s = 0  
 for i in range(len(weights)):  
 s += weights[i] \* inputs[i]  
 return s  
  
  
def sigmoid(s, beta):  
 return 1.0 / (1.0 + exp(-s \* beta))  
  
  
def forward\_propagate(w, row, beta):  
 x = row  
 b = [1]  
 v['v1'] = activate(w['w1'], x[:-2])  
 v['v1'] += activate(w['w10'], b)  
 v['v1'] = sigmoid(v['v1'], beta)  
  
 v['v2'] = activate(w['w2'], x[:-2])  
 v['v2'] += activate(w['w20'], b)  
 v['v2'] = sigmoid(v['v2'], beta)  
  
 v['v3'] = activate(w['w3'], x[:-2])  
 v['v3'] += activate(w['w30'], b)  
 v['v3'] = sigmoid(v['v3'], beta)  
  
 v['v4'] = activate(w['w4'], x[:-2])  
 v['v4'] += activate(w['w40'], b)  
 v['v4'] = sigmoid(v['v4'], beta)  
  
 v['v5'] = activate(w['w5'], x[:-2])  
 v['v5'] += activate(w['w50'], b)  
 v['v5'] = sigmoid(v['v5'], beta)  
  
 y['y1'] = activate(w['w6'], [v['v1'], v['v2'], v['v3'],v['v4'], v['v5']])  
 y['y1'] += activate(w['w60'], b)  
 y['y1'] = sigmoid(y['y1'], beta)  
  
 y['y2'] = activate(w['w7'],[v['v1'], v['v2'], v['v3'], v['v4'], v['v5']])  
 y['y2'] += activate(w['w70'], b)  
 y['y2'] = sigmoid(y['y2'], beta)  
  
 y\_ = [y['y1'], y['y2']]  
 v\_ = [v['v1'],v['v2'],v['v3'],v['v4'],v['v5']]  
 return v\_, y\_  
  
  
def sigmoid\_derivative(f\_s, beta):  
 return beta \* f\_s \* (1.0 - f\_s)  
  
  
def backward\_propagate(d, beta, x,v,y):  
 # liczenie gradientów dla warstwy ukryto-wyjsciowej  
  
 e = 0.0  
 for j in range(len(y)):  
 e += (d[j]-y[j]) \*\* 2  
 gradient\_vy\_list = []  
 gradient\_vy\_bias = []  
 for k in v:  
 gradient\_vy\_list.append(-(d[j]-y[j])\*sigmoid\_derivative(y[j], beta) \* k)  
 gradient\_vy\_bias.append(-(d[j]-y[j])\*sigmoid\_derivative(y[j], beta) \* 1)  
 all\_gradients[f'g{j+6}'] = gradient\_vy\_list  
 all\_gradients[f'g{j\*10+60}'] = gradient\_vy\_bias  
 e = 0.5 \* e  
 e = round(e,4)  
  
  
 # wagi warstwy ukryto-wyjsciowej  
 weights\_v\_y = [w['w6'], w['w7']]#, w['w8']]  
  
 # liczenie gradientów dla warstwy wejsciowo-ukrytej  
 for i in range(len(v)):  
 gradient\_xv\_list = []  
 gradient\_xv\_bias\_list = []  
 for j in range(len(x)):  
 gradient\_xv = 0  
 for m in range(len(y)):  
 gradient\_xv += -(d[m]-y[m])\*sigmoid\_derivative(y[m], beta) \* weights\_v\_y[m][i]  
 gradient\_xv = gradient\_xv\*sigmoid\_derivative(v[i], beta)\*x[j]  
 gradient\_xv\_bias = gradient\_xv\*sigmoid\_derivative(v[i], beta)\*1  
 gradient\_xv\_list.append(gradient\_xv)  
 gradient\_xv\_bias\_list.append(gradient\_xv\_bias)  
 all\_gradients[f'g{i + 1}'] = gradient\_xv\_list  
 all\_gradients[f'g{i \* 10 + 10}'] = gradient\_xv\_bias\_list  
  
  
def update\_weight(w, gamma, all\_gradients):  
 # listy do aktualizacji wag wejsciowo-ukrytych  
 weights\_updat\_vy = [w['w6'], w['w7']]  
 weights\_updat\_bias\_vy = [w['w60'], w['w70']]  
  
 gradients\_vy = [all\_gradients['g6'], all\_gradients['g7']]  
 gradients\_bias\_vy = [all\_gradients['g60'], all\_gradients['g70']]  
  
 # aktualizacja wag ukryto-wyjsciowych  
 for i in range(len(weights\_updat\_vy)):  
 for j in range(len(weights\_updat\_vy[i])):  
 weights\_updat\_vy[i][j] = weights\_updat\_vy[i][j] - gamma \* gradients\_vy[i][j]  
 weights\_updat\_vy[i][j] = round(weights\_updat\_vy[i][j],4)  
  
 # aktualizacja wag ukryto-wyjsciowych bias  
 for i in range(len(weights\_updat\_bias\_vy)):  
 for j in range(len(weights\_updat\_bias\_vy[i])):  
 weights\_updat\_bias\_vy[i][j] = weights\_updat\_bias\_vy[i][j] - gamma \* gradients\_bias\_vy[i][j]  
 weights\_updat\_bias\_vy[i][j] = round(weights\_updat\_bias\_vy[i][j], 4)  
  
 # listy do aktualizacji wag wejsciowo-ukrytych  
 weights\_updat\_xv = [w['w1'], w['w2'], w['w3'], w['w4'], w['w5']]  
 weights\_updat\_bias\_xv = [w['w10'], w['w20'], w['w30'], w['w40'], w['w50']]  
  
 gradients\_xv = [all\_gradients['g1'], all\_gradients['g2'], all\_gradients['g3'], all\_gradients['g4'], all\_gradients['g5']]  
 gradients\_bias\_xv = [all\_gradients['g10'], all\_gradients['g20'], all\_gradients['g30'], all\_gradients['g40'], all\_gradients['g50']]  
  
 # aktualizacja wag wejsciowo-ukrytych  
 for i in range(len(weights\_updat\_xv)):  
 for j in range(len(weights\_updat\_xv[i])):  
 weights\_updat\_xv[i][j] = weights\_updat\_xv[i][j] - gamma \* gradients\_xv[i][j]  
 weights\_updat\_xv[i][j] = round(weights\_updat\_xv[i][j],4)  
  
 # aktualizacja wag wejsciowo-ukrytych bias  
 for i in range(len(weights\_updat\_bias\_xv)):  
 for j in range(len(weights\_updat\_bias\_xv[i])):  
 weights\_updat\_bias\_xv[i][j] = weights\_updat\_bias\_xv[i][j] - gamma \* gradients\_bias\_xv[i][j]  
 weights\_updat\_bias\_xv[i][j] = round(weights\_updat\_bias\_xv[i][j],4)  
  
def train\_network(train\_data, gamma, beta, e, iterations, number\_epochs, value\_of\_difference):  
 flag = True  
 list\_of\_error\_in\_epoch = []  
 while flag:  
 list\_y = []  
 list\_d = []  
 iterations += 1  
 for row in train\_data:  
 v, y = forward\_propagate(w, row, beta)  
 # print('y',y)  
 max\_out = y.index(max(y))  
 y\_2 = [0, 0]  
 y\_2[max\_out] = 1  
 y\_ = y\_2  
 list\_y.append(y\_)  
  
 d = row[-2]  
 list\_d.append(d)  
 # print('d', d)  
 backward\_propagate(d, beta, row[:-2], v, y)  
 update\_weight(w, gamma, all\_gradients)  
 print(f'Current epoch {iterations}')  
 scores\_of\_error = 0  
 if len(list\_y) >= len(train\_dataset):  
 for i in range(len(list\_y[-700:])):  
 if list\_y[-700 + i] != list\_d[-700 + i]:  
 scores\_of\_error += 1  
 print('scores\_of\_error', scores\_of\_error)  
 list\_of\_error\_in\_epoch.append(scores\_of\_error)  
  
 significant\_change = [0]  
 if len(list\_of\_error\_in\_epoch) >= number\_epochs:  
 for i in range(len(list\_of\_error\_in\_epoch[:-1])):  
 if abs(list\_of\_error\_in\_epoch[i] - list\_of\_error\_in\_epoch[i+1]) >= value\_of\_difference:  
 significant\_change.append(1)  
 if sum(significant\_change) == 0:  
 data["koncowe\_wartosci\_wag"] = w  
 flag = False  
 break  
 else:  
 del list\_of\_error\_in\_epoch[0]  
  
  
iterations = 0  
gamma = 0.7  
beta = 0.5  
number\_epochs = 25  
value\_of\_difference = 6  
  
  
  
def testing\_neural\_network(w, row, beta, score, besni\_set):  
 d = row[-2]  
 v, y = forward\_propagate(w, row, beta)  
 max\_out = y.index(max(y))  
 y\_2 = [0, 0]  
 y\_2[max\_out] = 1  
  
 d\_3 = 0  
 y\_3 = 0  
 if y\_2 == [1, 0]:  
 y\_3 = 'Kecimen'  
 elif y\_2 == [0, 1]:  
 y\_3 = 'Besni'  
 besni\_set.append(row)  
 if d == [1, 0]:  
 d\_3 = 'Kecimen'  
 elif d == [0, 1]:  
 d\_3 = 'Besni'  
  
 print('Oczekiwane wyjście:', d\_3)  
 print('Wyjście sieci neuronowej: ', y\_3)  
 y = y\_2  
 if y == d:  
 score.append(score[-1]+1)  
 print('--->Sklasyfikowano poprawnie')  
 else:  
 print('--->Sklasyfikowano nie poprawnie')  
  
  
  
answer = "train"# str(input('Czy chcesz trenować [train] czy testować [test] sieć neuronową: '))  
  
# answer = 'test'  
answer1 = '1'  
  
if answer =='train':  
 train\_network(train\_dataset, gamma, beta, error, iterations, number\_epochs, value\_of\_difference)  
  
besni\_set = []  
  
if answer == 'test':  
 test\_dataset = []  
 if answer1 == '1':  
 file\_4 = open('test\_set.json', "r")  
 data\_4 = json.load(file\_4)  
 test\_dataset = data\_4["test\_set.json"]  
 file\_4.close()  
  
 score = [0]  
  
 for i in range(len(test\_dataset)):  
 print('Numer zestawu: ', i+1)  
 testing\_neural\_network(data["koncowe\_wartosci\_wag"], test\_dataset[i], beta, score, besni\_set)  
 print('Liczba poprawnie sklasyfikowanych zestawów: ',score[-1])  
 data\_6["besni\_set"] = besni\_set  
  
  
g = open("weight\_neural\_network.json", "w")  
json.dump(data, g)  
g.close()  
  
g2 = open("besni\_set.json", "w")  
json.dump(data\_6, g2)  
g2.close()

Plik 3

Rozwiązanie problemu podejmowania decyzji które rodzynki należy wziąć do badań za pomocą algorytmu tabu search.

import json  
  
file\_1 = open('besni\_set.json', "r")  
data = json.load(file\_1)  
file\_1.close()  
  
file\_2 = open('unnormed.json', "r")  
data2 = json.load(file\_2)  
file\_2.close()  
  
unnormed\_set = data2["unnormed.json"]  
besni\_set1 = data["besni\_set"]  
  
besni\_set = []  
for i in besni\_set1:  
 for j in unnormed\_set:  
 if i[8] == j[8]:  
 besni\_set.append(j)  
# print(besni\_set)  
  
Value = []  
Weight = []  
capacity = 800000  
for i in besni\_set:  
 Value.append(i[6])  
 Weight.append(i[0])  
  
  
def begin\_solution\_fun(set, Value, Weight, capacity):  
 item = {}  
 capacity = capacity  
 for i in range(len(set)):  
 item[i] = Value[i]/Weight[i], Value[i], Weight[i], i  
 item = sorted(item.values(), reverse=True)  
 value = 0  
 weight = 0  
 solution = []  
 for i in range(len(Value)):  
 if item[i][2] <= capacity:  
 capacity -= item[i][2]  
 weight += item[i][2]  
 solution.append(item[i][3])  
 value += item[i][1]  
 begin\_solution = [0 for i in range(len(Value))]  
 for i in solution:  
 begin\_solution[i] = 1  
  
 return begin\_solution, weight  
  
def count\_value(begin\_solution, item, capacity): # liczenie wartości plecaka dla pewnego rozwiązania  
 max\_value = 0  
 for i in range(len(item)):  
 if begin\_solution[i] == 1 and capacity >= 0:  
 max\_value += item[i][0]  
 capacity -= item[i][1]  
 if capacity < 0:  
 max\_value = -1  
 return max\_value  
  
def raisin\_id(best\_solution, item):  
 raisin\_id = []  
 for i in range(len(item)):  
 if best\_solution[i] == 1:  
 raisin\_id.append(item[i][2])  
 return raisin\_id  
  
  
begin\_solution, current\_weight = begin\_solution\_fun(besni\_set, Value, Weight, capacity)  
  
  
def tabu\_search(begin\_solution, value, weight, capacity, set):  
 item = {}  
 for i in range(len(set)):  
 item[i] = value[i], weight[i], set[i][8]  
 tabu = []  
 iteration = 0  
 iteration\_bs\_sol = 0  
 best\_solution = begin\_solution[:]  
 capacity2 = capacity  
 interim\_list = best\_solution[:]  
 while iteration - iteration\_bs\_sol <= 200:  
 iteration += 1  
 save\_sol = interim\_list[:]  
 value\_capacity = -1  
 iter = -1  
 for i in range(len(item)):  
 if interim\_list[i] == 1:  
 interim\_list[i] = 0  
 else:  
 interim\_list[i] = 1  
  
 solution\_fract = interim\_list[:]  
 if count\_value(solution\_fract, item, capacity2) > value\_capacity and not (i in tabu):  
 save\_sol = solution\_fract[:]  
 value\_capacity = count\_value(save\_sol, item, capacity)  
 iter = i  
 if count\_value(solution\_fract, item, capacity2) > count\_value(best\_solution, item, capacity2):  
 best\_solution = solution\_fract[:]  
 iteration\_bs\_sol = iteration  
  
 if interim\_list[i] == 1:  
 interim\_list[i] = 0  
 else:  
 interim\_list[i] = 1  
 if iter != -1:  
 if not (iter in tabu):  
 if len(tabu) == 20:  
 del tabu[0]  
 tabu.append(iter)  
 interim\_list = save\_sol[:]  
  
 print('Całkowity zsumowany obwód rodzynek do testowania', count\_value(best\_solution, item, capacity))  
 print('Lista id rodzynek które zostaną wzięte do badań', raisin\_id(best\_solution,item))  
  
 return raisin\_id(best\_solution, item)  
  
  
  
x = tabu\_search(begin\_solution, Value, Weight, capacity, besni\_set)  
  
  
end\_value\_area = 0  
for i in besni\_set:  
 for j in x:  
 if i[8] == j:  
 end\_value\_area += i[0]  
  
print('Całkowite pole powierzchni rodzynek do przetestowania', end\_value\_area)