Sprawozdanie z Metod Inteligencji Obliczeniowej

Małgorzata Makieła

10.03.2024

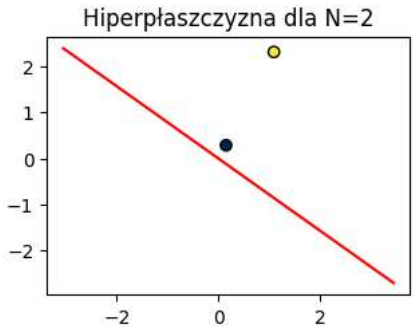
Zadanie 1.

Na początku trzeba było stworzyć po 400 2-wymiarowych punktów przypisanych do dwóch klas K1 i K2, pochodzących z rozkładów normalnych N([0,-1],1) i N([1,1],1)

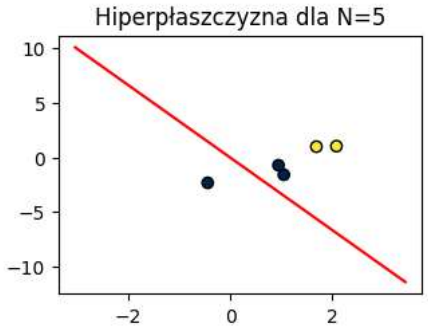
```
import numpy as np
from sklearn.linear_model import Perceptron
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix
import matplotlib.pyplot as plt

# 2D points
X = np.concatenate((np.random.normal([0, -1], [1, 1], [400, 2]), np.random.normal([1, 1], [1, 1], [400, 2])))

# K1 and K2 classes
Y = np.concatenate((np.array([[0, 0]] * 400), np.array([[0, 1]] * 400)))
```

Następnie podzieliłam dane na zbiory uczące i testujące w proporcji 400 - N do N. Sprawdziłam średnią dokładność klasyfikacji (accuracy) oraz odchylenie standardowe (std) dla kazdego n po 10 powtórzeniach. Dla każdego n, dla jednego przykładowego wywołania pętli ustaliłam wzór hiperpłaszczyzny oraz przedstawiłąm ją na wykresie razem z danymi testującymi.

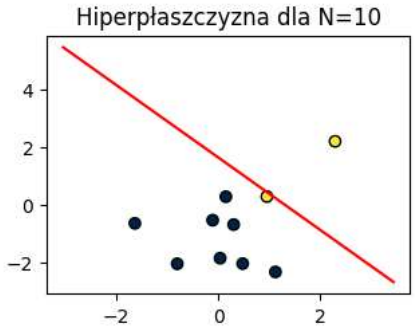
```
N = [2, 5, 10, 20, 100]

# dla każdego n
for i in N:
    accuracy_array = []
    printData = True

    # 10 powtórzeń dla każdego n
    for _ in range(10):

        # dzielimy na zbiory uczące i testujące
        X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=i)
        neuron = Perceptron(max_iter=20)
        neuron.fit(X_train, Y_train[:, 1])
        neuron.predict(X_train)
        Y_predicted = neuron.predict(X_test)

        # macierz pomyłek
        confusion_matrix_model = confusion_matrix(Y_test[:, 1], Y_predicted, labels=[True, False])

        # dokładność klasyfikacji - ilość elementów na diagonali / wszystkie elementy
        accuracy = (confusion_matrix_model[0, 0] + confusion_matrix_model[1, 1]) / np.sum(confusion_matrix_model)
        accuracy_array.append(accuracy)

        # plot once for a N value
        if printData:
            plt.figure(figsize=(3.5, 2.5))
            plt.scatter(X_test[:, 0], X_test[:, 1], c=Y_test[:, 1], cmap='cividis', edgecolors='k')
            plt.title(f"Hiperpłaszczyzna dla N={i}")
            x1 = np.linspace(np.min(X[:, 0]), np.max(X[:, 0]), 100)

            # hyperplane equation
            x2 = -(1. / neuron.coef_[0][1]) * (neuron.coef_[0][0] * x1 + neuron.intercept_[0])
            plt.plot(x1, x2, color='red')
            plt.show()
            printData = False

    print(f"\nN = {i}")
    print("Średnia dokładność klasyfikacji = " + str(np.mean(accuracy_array)))
    print("Odchylenie standardowe " + str(np.std(accuracy_array)))
```



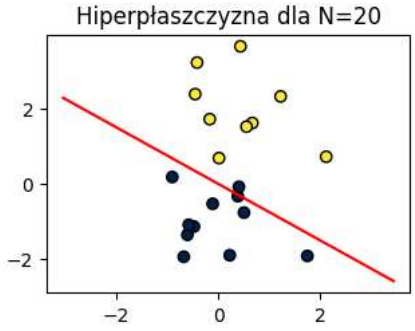
N = 2
Średnia dokładność klasyfikacji = 0.85
Odchylenie standardowe 0.22912878474779202



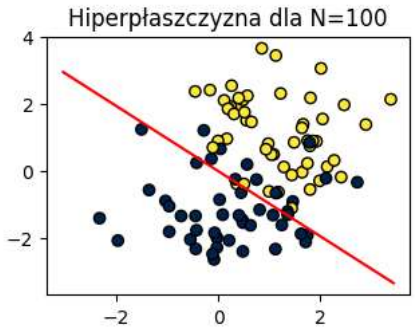
N = 5
Średnia dokładność klasyfikacji = 0.74
Odchylenie standardowe 0.1562049935181331



N = 10
Średnia dokładność klasyfikacji = 0.8400000000000001
Odchylenie standardowe 0.19595917942265423



N = 20
Średnia dokładność klasyfikacji = 0.8800000000000001
Odchylenie standardowe 0.06403124237432847



```
N = 100
Średnia dokładność klasyfikacji = 0.8
Odchylenie standardowe 0.08955445270895245
```

Wnioski

- Dla małych n - czyli dla bardzo małego zbioru testującego (n = 2, 5, 10) odchylenie standrdowe jest wyższe niż dla większych n, jest tu duża rozbieżność pomiędzy kilkoma wywołaniami programu - dla małej liczby elementów testujących wyniki są mocno zmienne.
- Zwiększenie n - dla większego zbioru testującego (n = 50, 100), powoduje mniejszą zmienność wyników - rozwiązania są bardziej stabilne i przewidywalne. Do tego osiągają dość wysoką dokładność (zazwyczaj między 0.8 - 0.85), więc dla większego zbioru model powinien być dość dokładny.

Zadanie 2 i 3

```
import numpy as np
import sklearn.datasets
from sklearn.linear_model import Perceptron
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix
import random

# pobranie zbioru irysów
data_set = sklearn.datasets.load_iris()

# 150 irysów, każdy opisują 4 cechy (feature_names)
X = data_set.data

# klasy irysów - jest ich 3 o nazwach target_names
Y = data_set.target

feature_names = data_set.feature_names
print(feature_names)

target_names = data_set.target_names
print(target_names)
```

```
['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal width (cm)']
['setosa' 'versicolor' 'virginica']
```

Po załadowaniu zbioru irysów podzieliłam zbiory na uczące i testujące na 3 sposoby:

1. Sposób pierwszy - z użyciem gotowej funkcji sklearn.model_selection.train_test_split, która określoną ilość elementów z całego zbioru X przypisuje danym uczącym i testującym. A więc w tych dwóch zbiorach mogą znaleźć się różne ilości kwiatów z każdej z klas.

```
# test_size = 30, bo 20% z 150 to 30
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=30)

# dodanie danych to tablicy przechowujacej dane z kolejnych 3 podziałów
train_test_sets = [(X_train, Y_train, X_test, Y_test)]
```

2. Sposób drugi - branie kolejnych 10 elementów z każdej z 3 klas irysów. Oryginalne dane Y podzielone są klasami - pierwsze 50 to klasa 0, kolejne 50 to klasa 2, a jeszcze kolejne 50 to klasa 3. Tak więc losujemy liczbę od 0 do 40 - a następnie z każdej klasy dodajemy po 10 kolejnych elementów do danych testujących - a resztę do danych uczących.

```
X_test_2 = []
Y_test_2 = []
X_train_2 = []
Y_train_2 = []

r = random.randint(0, 40)
for i in range(150):
    if r <= i < r+10 or r+50 <= i < r+60 or r+100 <= i < r+110:
        X_test_2.append(X[i])
        Y_test_2.append(Y[i])
    else:
        X_train_2.append(X[i])
        Y_train_2.append(Y[i])

train_test_sets.append((X_train_2, Y_train_2, X_test_2, Y_test_2))
```

3. Sposób trzeci - podobnie jak w sposobie drugim, losujemy z każdej klasy irysów po 10 elementów, jednak tu robimy to całkiem losowo używając funkcji random.sample, która produkuje nam 10 unikatowych liczb z podanego zakresu. Dane z wylosowanych indeksów dodajemy do zbioru testującego, a pozostałe do uczącego

```
random_nums_0 = random.sample(range(0, 49), 10)
random_nums_1 = random.sample(range(50, 99), 10)
random_nums_2 = random.sample(range(100, 149), 10)

X_test_3 = []
Y_test_3 = []
X_train_3 = []
Y_train_3 = []

for i in range(150):
    if i in random_nums_0 or i in random_nums_1 or i in random_nums_2:
        X_test_3.append(X[i])
        Y_test_3.append(Y[i])
    else:
        X_train_3.append(X[i])
        Y_train_3.append(Y[i])

train_test_sets.append((X_train_3, Y_train_3, X_test_3, Y_test_3))
```

Następnie dla trzech powyższych przykładów podziału danych trenujemy model.

```
for (X_train, Y_train, X_test, Y_test) in train_test_sets:
    neuron = Perceptron(early_stopping=False, max_iter=20)
    neuron.fit(X_train, Y_train)
    neuron.predict(X_train)
    Y_predicted = neuron.predict(X_test)
    confusion_matrix_model = confusion_matrix(Y_test, Y_predicted)
    print(confusion_matrix_model)
    accuracy = (confusion_matrix_model[0, 0] + confusion_matrix_model[1, 1] + confusion_matrix_model[2, 2]) / np.sum(
        confusion_matrix_model)
    print(accuracy)
```

```
 [[11  0  0]
  [ 0 10  1]
  [ 0  0  8]]
 0.9666666666666667
 [[ 9  1  0]
  [ 0 10  0]
  [ 0  9  1]]
 0.6666666666666666
 [[ 9  1  0]
  [ 0 10  0]
  [ 0  5  5]]
 0.8
```

Wnioski

- Po kilku wywołaniach programu mogę stwierdzić, że dla wszystkich 3 sposobów podziału, wyniki są podobne. Poza pojedynczymi przypadkami niskich dokładności - (0.4 - 0.5), oscylują one w wartościach 0.7 - 0.9
- Podobnie jak w poprzednim zadaniu, zrobiłam eksperyment, i dla każdego zbioru (z 3 sposobów podziału) wykonałam 100 powtórzeń pętli poszukując średniej dokładności. W 3 kolejnych wywołaniach programu otrzymałam:

1. Zbiór pierwszy: 0.766666666666666, 0.7000000000000013, 0.7000000000000013
2. Zbiór drugi: 0.7000000000000013, 0.733333333333334, 0.7000000000000013
3. Zbiór trzeci: 0.9666666666666676, 0.8666666666666651, 0.9666666666666676

Tak więc zbiór trzeci pozwolił na najdokładniejsze wytrenowanie modelu, podczas gdy pierwszy i drugi zbiór osiągnęły podobną dokładność.

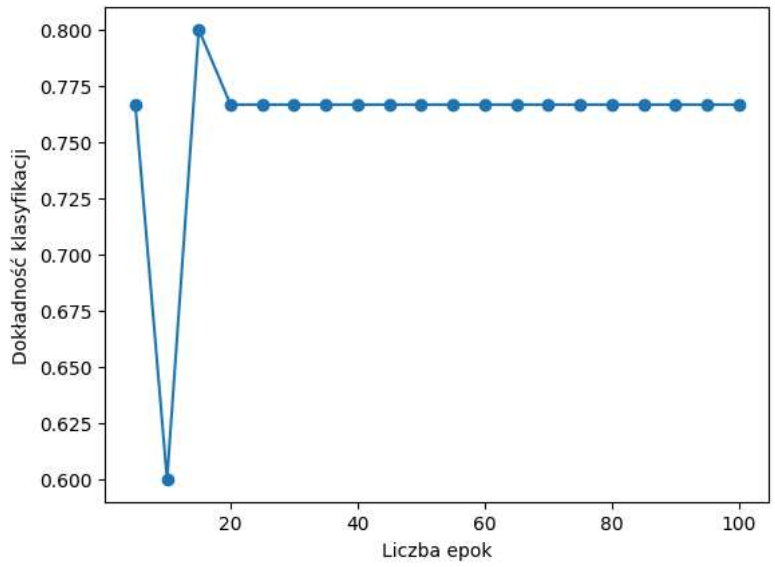
Zadanie 4.

Zapożyczając kod z zadania powyższego, przetestowałam, jak dla różnej ilości epok zmienia się dokładność klasyfikacji zbioru irysów.

```
import numpy as np
import sklearn.datasets
from sklearn.linear_model import Perceptron
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix
import matplotlib.pyplot as plt

data_set = sklearn.datasets.load_iris()
X = data_set.data
Y = data_set.target

X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=30)
epoki = [5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 70, 75, 80, 85, 90, 95, 100]
accuracy_array = []
for e in epoki:
    neuron = Perceptron(max_iter=e, early_stopping=False)
    neuron.fit(X_train, Y_train)
    neuron.predict(X_train)
    Y_predicted = neuron.predict(X_test)
    confusion_matrix_model = confusion_matrix(Y_test, Y_predicted)
    # print(confusion_matrix_model)
    accuracy = (confusion_matrix_model[0, 0] + confusion_matrix_model[1, 1] + confusion_matrix_model[2, 2]) / np.sum(
        confusion_matrix_model)
    accuracy_array.append(accuracy)
```

Następnie przedstawiłam zależność dokładności klasyfikacji od ilości epok.

```
plt.plot(epoki, accuracy_array, marker='o')
plt.xlabel('Liczba epok')
plt.ylabel('Dokładność klasyfikacji')
plt.show()
```



Wnioski: dokładność klasyfikacji rośnie wraz z ilością epok - jednak przy ponad 20 epokach już się nie zmienia, więc 20 epok jest optymalnym wyborem.