Metody inteligencji obliczeniowej
Sprawozdanie 1
Klasyfikacja z użyciem sztucznych neuronów

Yuliya Zviarko

19.03.2025

Spis treści

1 Wprowadzenie

W ramach drugiego laboratorium zapoznaliśmy się z podstawowymi pojęciami dotyczącymi uczenia maszynowego. Omówione zagadnienia obejmowały klasyfikację oraz jej biologiczne podstawy, a także budowę perceptronu, jego schemat działania, strukturę warstw oraz sposób wyznaczania wag. Ponadto poznaliśmy techniki wstępnego przetwarzania danych wejściowych, w tym metody wykrywania wartości odstających. Są to kluczowe zagadnienia teoretyczne, które umożliwiły sprawniejsze wykonanie części praktycznej, szczegółowo opisanej w kolejnej sekcji sprawozdania.

2 Przebieg laboratorium

2.1 Zadanie 1

2.1.1 Opis zadania

W pierwszym zadaniu należało stworzyć zestaw punktów należących do dwóch klas: K1 i K2. Punkty z klasy K1 powinny być losowane z rozkładu normalnego o średniej $[0, -1]$ i wariancji 1. Punkty z klasy K2 powinny pochodzić z rozkłdu normalnego o średniej $[1, 1]$ i wariancji 1. Łącznie zbiór powinien zawierać 200 punktów, po 100 dla każdej klasy.

Należało wybrać zbiory uczące o następującej liczebności: 5, 10, 20 oraz 100. Dla każdego wariantu podziału należało znaleźć równanie prostej, która najlepiej oddziela klasy K1 i K2. Uzyskaną prostą należało zaprezentować wraz z punktami testowymi oraz linią (hiperpłaszczyzną), która rozdziela klasy. Następnie należało ocenić skuteczność klasyfikatora w zależności od proporcji danych uczących i testujących.

2.1.2 Implementacja

W celu rozwiązania zadania zaimplementowano klasyfikację perceptronem w następujących krokach:

1. Wygenerowano dwa zbiory punktów: K1 i K2, po 100 punktów każdy, losowane z rozkładu normalnego o różnych średnich i wariancjach.

```
K1 = np.random.normal([0, -1], [1, 1], [100, 2])
K2 = np.random.normal([1, 1], [1, 1], [100, 2])
K = np.concatenate((K1, K2))
y = np.concatenate((np.ones(100), np.full(100, 2)))
```

Listing 1: Generowanie punktów K1 i K2

2. Zbiory uczące o różnych rozmiarach (5, 10, 20, 100) zostały wygenerowane przy użyciu funkcji `train_test_split`.

```
X_train, X_test, y_train, y_test = train_test_split(K, y,
    train_size=rozmiar, stratify=y, random_state=42)
```

Listing 2: Podział danych na zbiory uczące i testowe

3. Utworzono model perceptronu, który następnie trenowano na wybranych zbiorach uczących.

```
    neuron = Perceptron(tol=1e-3, max_iter=1000,
        random_state=42)
    neuron.fit(X_train, y_train)
```

Listing 3: Trenowanie modelu perceptronu

4. Na podstawie wytrenowanego modelu, dla każdego zbioru uczącego wyznaczono granicę decyzyjną oraz oceniono dokładność na zbiorze testowym.

```
    dokladnosc = neuron.score(X_test, y_test)
    x1 = np.linspace(x_min, x_max, 100)
    if neuron.coef_[0][1] != 0:
        x2 = -(neuron.coef_[0][0] * x1 + neuron.intercept_[0])
            / neuron.coef_[0][1]
        ax.plot(x1, x2, '-k', linewidth=2, label='Granica
            decyzyjna')
```

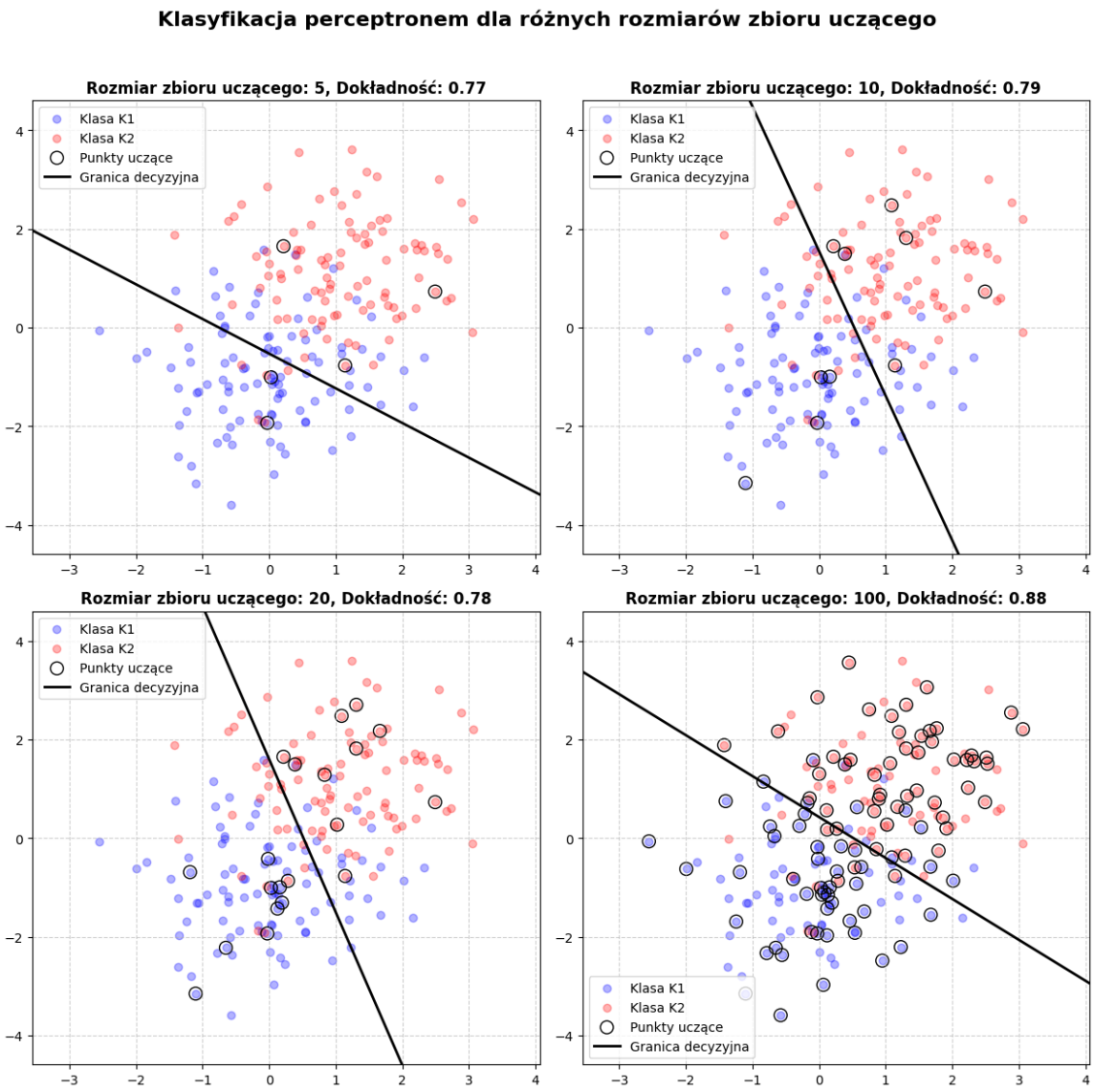
Listing 4: Ocena dokładności i rysowanie granicy decyzyjnej

5. Na samym końcu wygenerowano wykresy przedstawiające punkty klas K1 i K2 oraz wyznaczoną prostą decyzyjną.

2.1.3 Wyniki

Wyniki uzyskane dla różnych rozmiarów zbioru uczącego przedstawiają się następująco:

Rozmiar zbioru uczącego	Dokładność
5	0.77
10	0.79
20	0.78
100	0.88



Rysunek 1: Klasyfikacja perceptronem dla różnych rozmiarów zbioru uczącego

Z wykresu widać, że wraz ze wzrostem rozmiaru zbioru uczącego dokładność klasyfikatora rośnie. Dla mniejszych zbiorów (5 i 10 punktów) dokładność waha się w przedziale 77%-79%, a dla większych zbiorów (100 punktów) osiąga wartość 88%. Takie wyniki są efektem coraz lepszego dopasowania modelu do danych, co pozwala na bardziej precyzyjne oddzielenie klas. Większa liczba danych pozwala modelowi lepiej rozróżniać klasy.

Choć dla małego zbioru uczącego, przy wyniku 77%, może wydawać się, że osiągamy wystarczający poziom trafności, patrząc na wykres, widać, że model nadal nie jest idealny w kontekście rozdzielania klas K1 i K2. Dla przypadku z największym rozmiarem zbioru uczącego widać, jak prosta dobrze rozdziela klasy K1 i K2, co dodatkowo przekłada się na wyższą dokładność.

2.2 Zadanie 2

2.2.1 Opis zadania

W następnym zadaniu skorzystano z pliku `fuel.txt`, w którym pierwsze trzy kolumny to właściwości fizykochemiczne próbek, czwarta kolumna - klasa czystości. Należało sprawdzić skuteczność sieci opartej o pojedynczy neuron do klasyfikacji w tym problemie, porównując wyniki dla pięciokrotnego uczenia sieci. Wszystkie dane zostały potraktowane jako uczące.

2.2.2 Implementacja

1. Pierwszym krokiem było wczytanie danych z pliku `fuel.txt`, gdzie kolumny przedstawiają właściwości fizykochemiczne próbek, a czwórka zawiera klasę czystości.

```
    X = pd.read_csv("data/fuel.txt", sep=",", header=0)
    y = X['purity_class'].apply(lambda x: 0 if x == 'A' else 1)
```

2. Następnie usunięto kolumnę `purity_class`, ponieważ stanowiła ona etykiety klasowe, a my potrzebujemy jedynie cech fizykochemicznych jako danych wejściowych do modelu.

```
    X.drop(columns=['purity_class'], inplace=True)
```

3. Aby poprawić działanie sieci neuronowej, dane wejściowe zostały znormalizowane przy użyciu `MinMaxScaler`, co zapewniało, że wszystkie cechy miały wartości w tym samym zakresie.

4. Zainicjowano model perceptronowy z odpowiednimi parametrami:

```
    neuron = Perceptron(tol=1e-3, max_iter=1000,
       random_state=42)
```

5. Proces uczenia sieci został powtórzony pięciokrotnie, gdzie w każdej iteracji sieć była trenowana na tych samych danych wejściowych. W każdej iteracji wyliczono dokładność oraz wygenerowano macierz pomyłek.

```
    for i in range(5):
        print(f"\nIteracja {i+1}:")
        neuron.fit(X_scaled, y)
        predictions = neuron.predict(X_scaled)
        accuracy = neuron.score(X_scaled, y)
        print(f"Dokladnosc w tej iteracji: {accuracy:.2f}")
```

6. Dla każdej iteracji generowana była macierz pomyłek, która pokazywała, ile próbek zostało poprawnie sklasyfikowanych, a ile zostało błędnie przypisanych do niewłaściwej klasy. Macierz była wyświetlana w postaci wykresu.

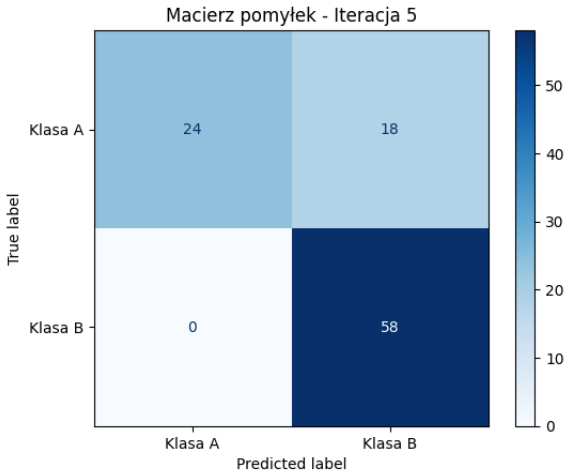
```
cm = confusion_matrix(y, predictions, labels=[0, 1])
disp = ConfusionMatrixDisplay(confusion_matrix=cm,
    display_labels=['Klasa A', 'Klasa B'])
disp.plot(cmap='Blues')
plt.title(f"Macierz pomylek - Iteracja {i+1}")
plt.show()
```

2.2.3 Wyniki

Po przeprowadzeniu pięciokrotnego uczenia sieci perceptronowej uzyskano następujące wyniki w macierzy pomyłek:

- Klasa A - klasa A = 24
- Klasa A - Klasa B = 18
- Klasa B - Klasa A = 0
- Klasa B - klasa B = 58

Wygenerowany wykres macierzy pomyłek na podstawie powyższych wyników wygląda następująco:



Rysunek 2: Macierz pomyłek po 5 iteracjach.

Po przeprowadzeniu pięciokrotnego uczenia sieci perceptronowej, uzyskano wysoką dokładność klasyfikacji (83%), co wynikało z tego, że wszystkie dane zostały użyte jako dane uczące. Model osiągnął bardzo stabilne wyniki, które nie zmieniały się pomiędzy iteracjami, co sugeruje, że sieć dobrze "zapamiętuje"dane, które jej dostarczamy.
W przypadku klasy B, model uzyskał bardzo wysoką precyzję, poprawnie klasyfikując wszystkie próbki tej klasy, natomiast w przypadku klasy A pojawiły się błędy klasyfikacyjne. Model przypisał 18 próbek klasy A do klasy B, co sugeruje, że może istnieć trudność w rozróżnieniu tych dwóch klas. Wyniki te pokazują, że sieć perceptronowa jest stabilna, ale może wymagać dalszej optymalizacji, szczególnie w zakresie poprawy klasyfikacji klasy A.
Ostatecznie, model osiągał wysoką dokładność w każdej iteracji, co może sugerować, że sieć jest zbyt "dopasowana"do danych treningowych, a tym samym mało generalizowalna.

2.3 Zadanie 3

2.3.1 Opis zadania

W tym zadaniu należało załadować zbiór danych Iris, a następnie samodzielnie dokonać podziału na dane uczące i testujące w proporcji 80%/20%. Celem było zbudowanie sieci z pojedynczą warstwą perceptronów, której zadaniem była jak najdokładniejsza klasyfikacja gatunków irysów na podstawie ich pomiarów. Przeprowadzono analizę macierzy pomyłek dla 10 uruchomień algorytmu.

2.3.2 Implementacja

1. Zainicjowano model perceptronu z odpowiednimi parametrami:

```
perceptron_layer = Perceptron(tol=1e-3, max_iter=20,
    early_stopping=True)
```

2. Przeprowadzono 10 prób, w każdej z nich dokonano podziału danych na zbiór uczący i testowy w proporcji 80%/20%, a następnie wytrenowano model perceptronu.

```
for i in range(10):
    X_train, X_test, Y_train, Y_test =
        train_test_split(data_iris['data'], data_iris['target'],
        test_size=0.2)
    perceptron_layer.fit(X_train, Y_train)
```

3. Dla każdej próby obliczono macierz pomyłek oraz dokładność modelu.

```
confusion_matrix_model = confusion_matrix(Y_test, Y_predicted)
accuracy = np.trace(confusion_matrix_model) /
    np.sum(confusion_matrix_model)
accuracies.append(accuracy)
```

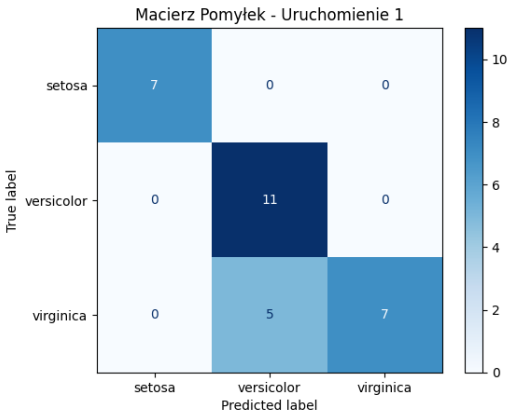
4. Na podstawie macierzy pomyłek wygenerowano wykresy dla każdej próby, które ilustrują poprawność klasyfikacji.

```
disp =
    ConfusionMatrixDisplay(confusion_matrix=confusion_matrix_model,
    display_labels=data_iris['target_names'])
disp.plot(cmap=plt.cm.Blues)
plt.title(f"Macierz Pomylek - Uruchomienie {i+1}")
plt.show()
```

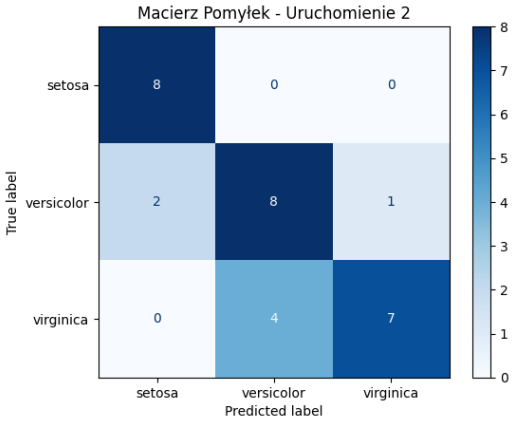
5. Na koniec obliczono średnią dokładność po 10 uruchomieniach.

```
mean_accuracy = np.mean(accuracies)
print(f"Srednia dokladnosc po 10 uruchomieniach:
    {mean_accuracy:.2f}")
```

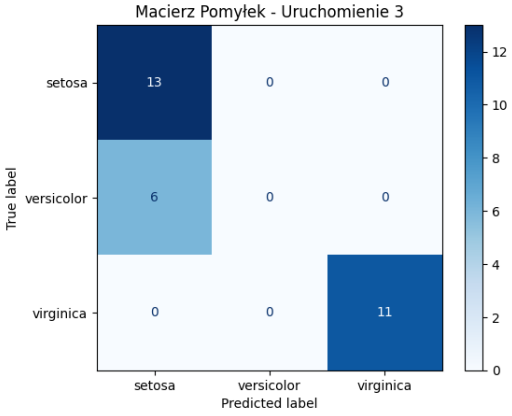
2.3.3 Wyniki



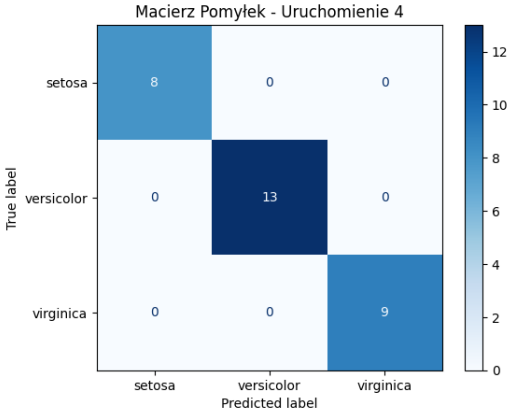
Rysunek 3: Iteracja 1



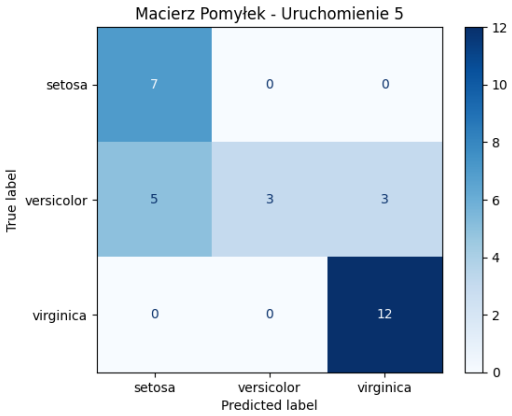
Rysunek 4: Iteracja 2



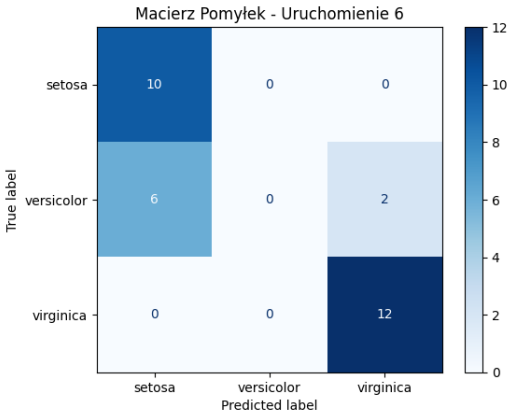
Rysunek 5: Iteracja 3



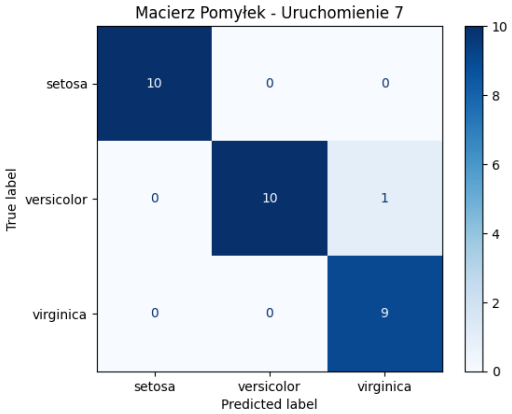
Rysunek 6: Iteracja 4



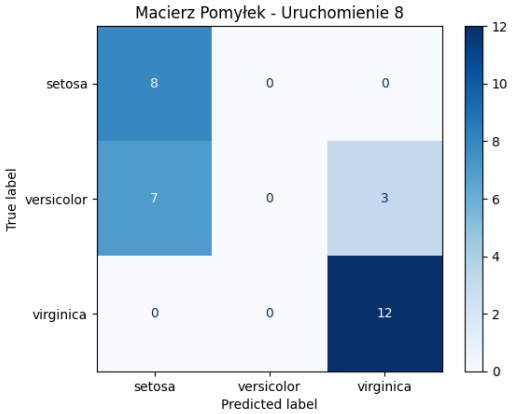
Rysunek 7: Iteracja 5



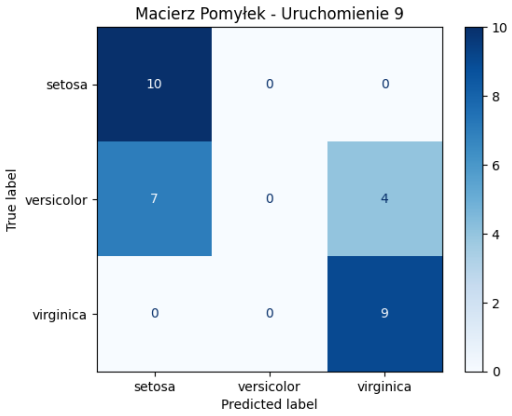
Rysunek 8: Iteracja 6



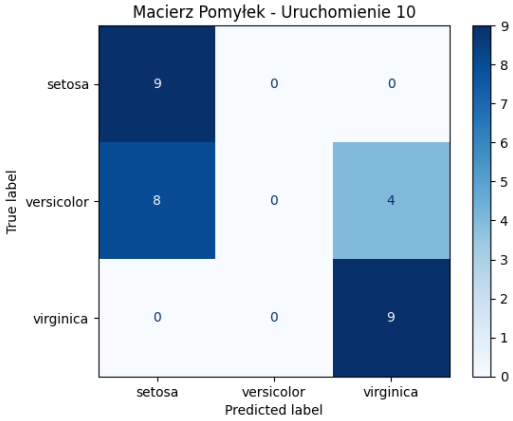
Rysunek 9: Iteracja 7



Rysunek 10: Iteracja 8



Rysunek 11: Iteracja 9



Rysunek 12: Iteracja 10

Zauważmy, że perceptron jest modelem liniowym, który stara się znaleźć liniowe separatory między klasami w danych. Gdy dane są liniowo separowalne, perceptron działa skutecznie, osiągając wysoką trafność klasyfikacji. W przypadku zestawu danych irys możemy założyć, że niektóre kombinacje cech będą separowalne liniowo, jednak inne mogą sprawić trudności modelowi, jeśli dane nie są idealnie rozdzielone linią.
Analizując wyniki, zauważamy, że najwyższa osiągnięta trafność to 93%, jednak większość wyników znajduje się poniżej tej wartości. Taki wynik może sugerować, że cechy w zbiorze danych irys nie są w pełni liniowo separowalne, co wpływa na skuteczność perceptronu. Średnia trafność uzyskana po 10 uruchomieniach wynosi około 77%, co wskazuje na stabilność wyników, ale również na ich zmienność w zależności od podziału danych. Z tego wynika, że perceptron jest odpowiedni do klasyfikacji danych, które są w dużej mierze separowalne liniowo, ale może mieć trudności w bardziej złożonych przypadkach, gdzie separacja liniowa nie jest możliwa.

2.4 Zadanie 4

2.4.1 Opis zadania

Kontynuowaliśmy pracę z zadania 3, tym razem dzieląc zbiór irysów na zbiór uczący i testujący na co najmniej trzy różne sposoby. Wykonałam to, stosując różne wartości parametru `test_size` w funkcji `train_test_split`, czyli: `test_size_arr = [0.1, 0.3, 0.5, 0.8]`.

2.4.2 Implementacja

1. Na początku załadowano zbiór irysów przy pomocy funkcji `load_iris()`.
2. Zdefiniowano perceptron z parametrami:

```
    perceptron_layer = Perceptron(tol=1e-4, max_iter=20,
       early_stopping=True)
```

Listing 5: Inicjalizacja perceptronu

3. Dla każdego rozmiaru zbioru testowego zdefiniowanego w `test_size_arr`, przeprowadzono podział na zbiór uczący i testowy przy użyciu funkcji `train_test_split()`.

```
    for i in test_size_arr:
        X_train, X_test, Y_train, Y_test =
           train_test_split(data_iris['data'],
           data_iris['target'], test_size=i)
```

Listing 6: Podział na zbiory uczący i testowy

4. Model perceptronu został wytrenowany na zbiorze uczącym za pomocą funkcji `fit()`.
5. Po wytrenowaniu modelu, dokonano predykcji dla zbioru testowego przy pomocy funkcji `predict()`.
6. Na podstawie rzeczywistych etykiet (`Y_test`) oraz przewidywanych etykiet (`Y_predicted`), obliczono macierz pomyłek przy pomocy funkcji `confusion_matrix()`.
7. Dokładność modelu obliczono dzieląc sumę elementów diagonalnych macierzy pomyłek przez sumę wszystkich elementów macierzy. Wynik zapisywano w zmiennej `accuracies`.

```
    accuracy = np.trace(confusion_matrix_model) /
       np.sum(confusion_matrix_model)
    accuracies.append(accuracy)
```

Listing 7: Obliczanie dokładności

8. Została wyświetlona macierz pomyłek, a także wizualizacja wyników z pomocą biblioteki `matplotlib`:

```
    disp =
        ConfusionMatrixDisplay(confusion_matrix=confusion_matrix_model,
        display_labels=data_iris['target_names'])
    disp.plot(cmap=plt.cm.Blues)
    plt.title(f"Macierz Pomylek - Rozmiar Testu {i}")
    plt.show()
```

Listing 8: Wizualizacja macierzy pomyłek

9. Na końcu wygenerowano wykres pokazujący zależność między rozmiarem zbioru testowego a dokładnością klasyfikacji.

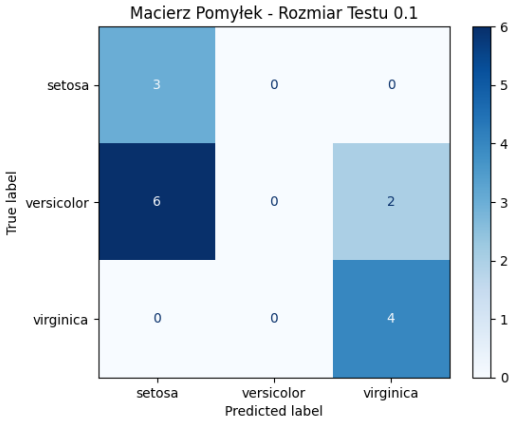
```
    plt.plot(test_size_arr, accuracies, marker='o',
        linestyle='-', color='b')
    plt.title('Wplyw rozmiaru zbioru testowego na dokladnosc
        klasyfikacji')
    plt.xlabel('Rozmiar zbioru testowego (%)')
    plt.ylabel('Dokladnosc')
    plt.grid(True)
    plt.show()
```

Listing 9: Wykres dokładności

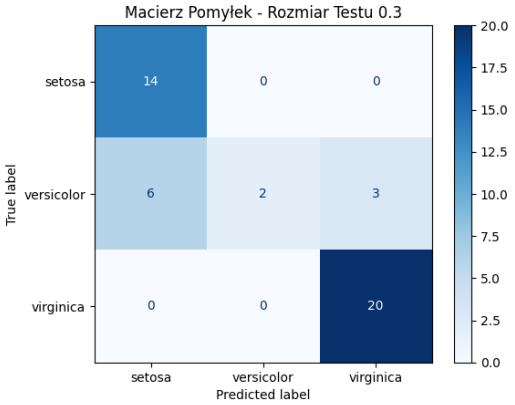
Po wykonaniu wszystkich uruchomień, obliczono średnią dokładność na podstawie wyników uzyskanych z różnych rozmiarów zbioru testowego.

2.4.3 Wyniki

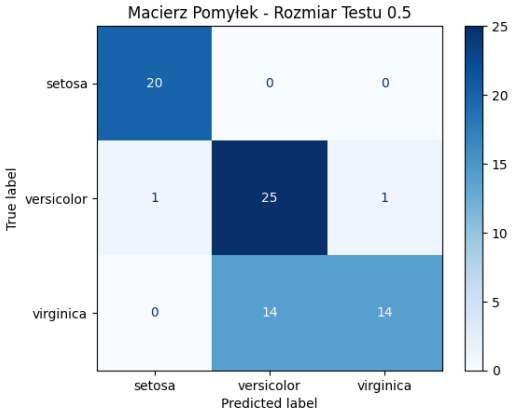
Analizując wyniki, widać, że większy zbiór danych treningowych prowadzi do wyższej dokładności modelu. Z kolei im większy zbiór testowy, tym precyzyjniej można ocenić dokładność modelu, obliczając ją na podstawie macierzy pomyłek.



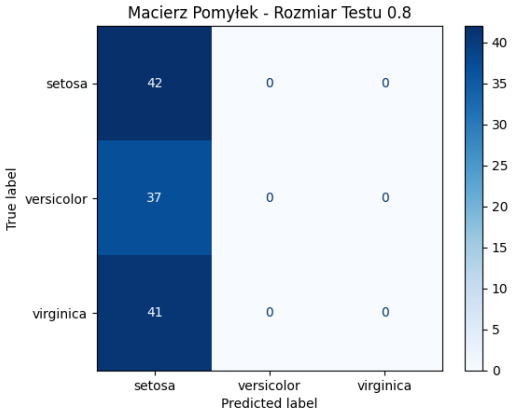
Rysunek 13: Macierz pomyłek - rozmiar zbioru testowego 0.1



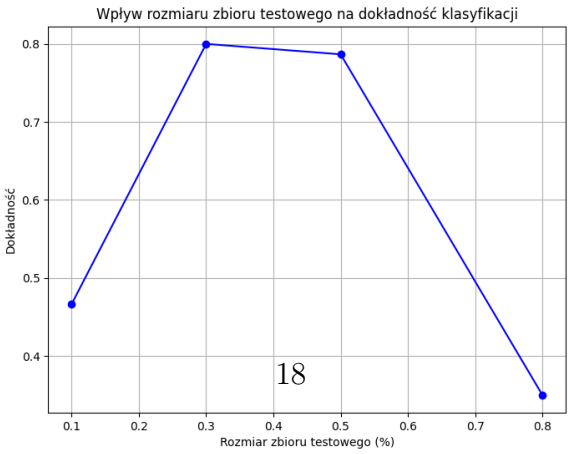
Rysunek 14: Macierz pomyłek - rozmiar zbioru testowego 0.3



Rysunek 15: Macierz pomyłek - rozmiar zbioru testowego 0.5



Rysunek 16: Macierz pomyłek - rozmiar zbioru testowego 0.8



Rysunek 17: Wpływ rozmiaru zbioru testowego na dokładność klasyfikacji

2.5 Zadanie 5

2.5.1 Opis zadania

Celem ostatniego zadania było zbadanie wpływu liczby epok na poprawność klasyfikacji zbioru irysów, analizowanego w poprzednich zadaniach. Wyniki należało przedstawić w formie wykresu zależności średniej trafności klasyfikacji od liczby epok.

2.5.2 Implementacja

1. Ponownie załadowano zbiór danych irysów:

```
data_iris = load_iris()
```

2. Zdefiniowano listę liczby epok oraz listę do przechowywania dokładności modelu:

```
epochs = [10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 120, 140, 160,
    180, 200]
accuracies = []
```

3. Iteracja po różnych wartościach epok:

```
for i in epochs:
```

a) Utworzono warstwę perceptronu z określoną liczbą epok i wyłączonym wczesnym zatrzymaniem:

```
    perceptron_layer = Perceptron(tol=1e-4, max_iter=i,
        early_stopping=False)
```

b) Podzielono zbiór danych na treningowy i testowy:

```
    X_train, X_test, Y_train, Y_test =
        train_test_split(data_iris['data'], data_iris['target'],
        test_size=0.2)
```

c) Przeprowadzono trenowanie modelu:

```
    perceptron_layer.fit(X_train, Y_train)
```

d) Obliczono dokładność klasyfikacji i zapisano wynik:

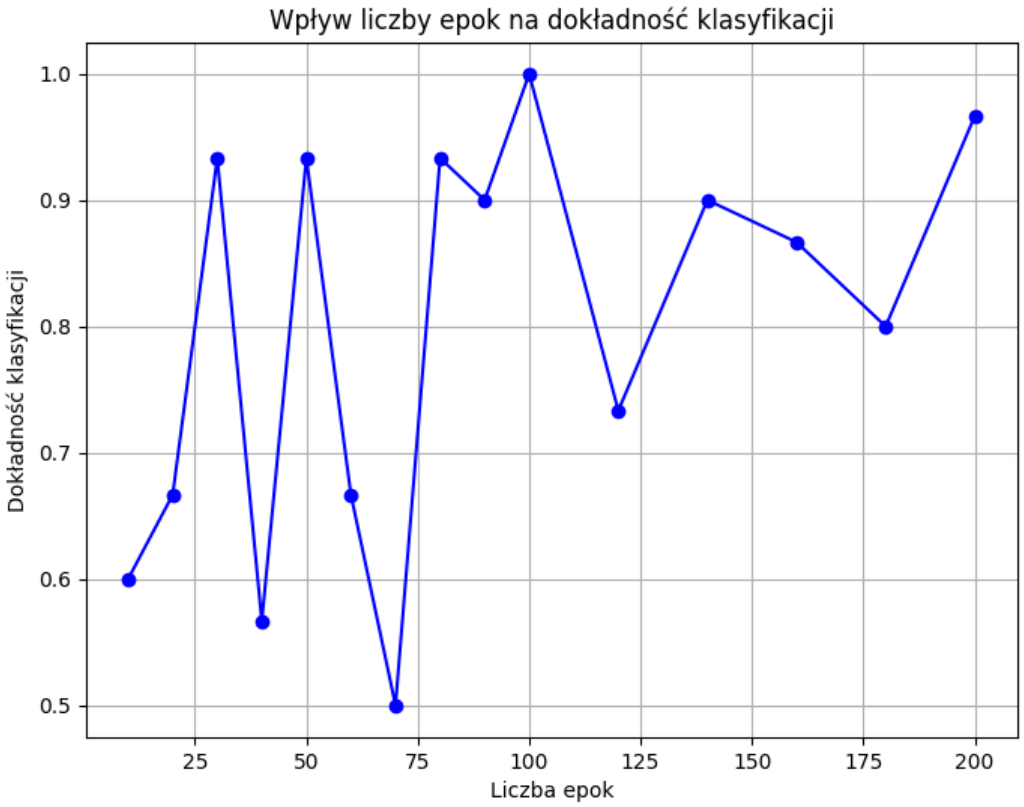
```
    accuracy = perceptron_layer.score(X_test, Y_test)
    accuracies.append(accuracy)
```

5. Narysowano wykres przedstawiający wpływ liczby epok na dokładność klasyfikacji:

```
plt.figure(figsize=(8, 6))
plt.plot(epochs, accuracies, marker='o', linestyle='-', color='b')
plt.title('Wplyw liczby epok na dokladnosc klasyfikacji')
```

```
plt.xlabel('Liczba epok')
plt.ylabel('Dokladnosc klasyfikacji')
plt.grid(True)
plt.show()
```

2.5.3 Wyniki



Rysunek 18: Wpływ liczby epok na dokładność klasyfikacji

Analiza wyników pokazuje, że zwiększenie liczby epok może poprawić dokładność klasyfikacji na zbiorze testowym, jednak do pewnego momentu. Po przekroczeniu określonej liczby epok dalsze trenowanie modelu nie prowadzi do istotnej poprawy wyników. Zbyt mała liczba epok skutkuje niedouczeniem modelu, co prowadzi do niskiej trafności klasyfikacji. Natomiast zbyt duża liczba epok może spowodować przetrenowanie modelu, co oznacza jego zbyt dobre dopasowanie do danych uczących, a w konsekwencji gorszą generalizację na zbiorze testowym.