POLITECHNIKA RZESZOWSKA
im. IGNACEGO ŁUKASIEWICZA

WYDZIAŁ ELEKTROTECHNIKI I INFORMATYKI
POLITECHNIKI RZESZOWSKIEJ

Projekt

TEMAT:

Learning Vector Quantization(LVQ)

Kierunek: Informatyka
2EF – DI
Semestr letni
Grupa laboratoryjna: L08

Wykonała: Vanivska Veronika
24.05.2024

Spis treści

1 Opis projektu

Tematem projektu jest stworzenie sieci neuronowej LVQ uczącej się rozpoznawać przeżywalność osób,który były chore na zapalenie wątroby. Należy dokonać przygotowania danych, napisać program w języku Python oraz preprowadzić eksperymenty dla znależenia najbardziej optymalnych parametrów sieci,które będą zapewniać najwyższą poprawność klasyfikacji.

2 Wstęp teorytyczny

2.1 Sztuczna sieć neuronowa

Sztuczna sieć neuronowa to system obliczeniowy inspirowany biologicznymi sieciami neuronowymi, które tworzą podstawę działania mózgu człowieka. Jest to model matematyczny zaprojektowany do przetwarzania informacji w sposób podobny do tego, jak robi to ludzki mózg, przez zbiorowe działanie wielu prostych jednostek przetwarzających zwanych neuronami. Sztuczne neurony są zorganizowane w warstwy, w tym warstwę wejściową, warstwy ukryte i warstwę wyjściową. Warstwa wejściowa przyjmuje dane wejściowe, warstwy ukryte przetwarzają te dane, a warstwa wyjściowa generuje ostateczne wyniki. Każdy neuron w sieci odbiera sygnały wejściowe, które są następnie modyfikowane przez przypisane wagi. Te wagi określają siłę połączeń między neuronami, wpływając na przekazywanie i przetwarzanie sygnałów w całej sieci. Wagi są kluczowym elementem, który pozwala sieci uczyć się i adaptować do różnych zadań. W sztucznych sieciach neuronowych połączenia między neuronami mogą mieć różne struktury, zależnie od typu sieci i jej architektury. Model wybranej sieci decyduje o ilości warstw ukrytych, których może istnieć dowolna ilość. W sieciach w pełni połączonych, każdy neuron w jednej warstwie jest połączony z każdym neuronem w warstwie następnej, co umożliwia kompleksowe przetwarzanie informacji. Taka struktura jest typowa dla klasycznych sieci neuronowych, zwanych również gęsto połączonymi.

2.1.1 Model neuronu

Sztuczne neurony to są fundamendalne komponenty,które odpowiedzialne za budowę struktury sztucznej sieci neuronowej. W uproszczeniu neuron może znajdować się w dwóch stanach: pobudzony(aktywny) lub w stanie spoczynku. Aby go pobudzić,potrzebujemy sygnałów wejściowych. Zwykle neuron ma wiele wejść i jedno wyjście. Pobiera on sygnały wejściowe, przetwarza je i przekazuje na wyjście. Sygnały wejściowe reprezentowane są jako wektor wejściowy X, który przyjmuje wartości $< x_1, x_2, ..., x_n >$. Wartość sygnału wejściowego obliczamy w dwóch etapach.

1. Mnożenie sygnałów wejściowych przez odpowiadające im wagi i sumujemy wraz z przesunięciem b.
$$z = \sum_{j=1}^{n} w_j x_j + b. \tag{1}$$

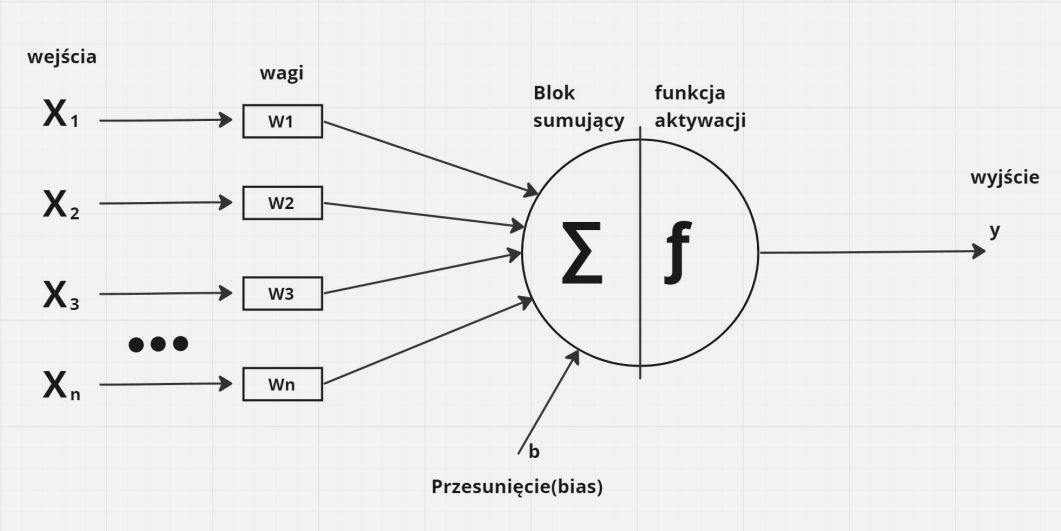
2. Działanie określonej funkcji wejścia-wyjścia zwanej funkcją aktywacji.

Sygnał wyjściowy neuronu y określony jest zależnością:

$$y = f\left(\sum_{j=1}^{n} w_j x_j + b\right), \tag{2}$$

gdzie:

- x_j - wartość j-tego sygnału wejśćiowego
- w_j - wartość współczynnika wagowego
- b - przesunięcie
- n - liczba sygnałów wejściowych
- y - sygnał wyjściowy neuronu



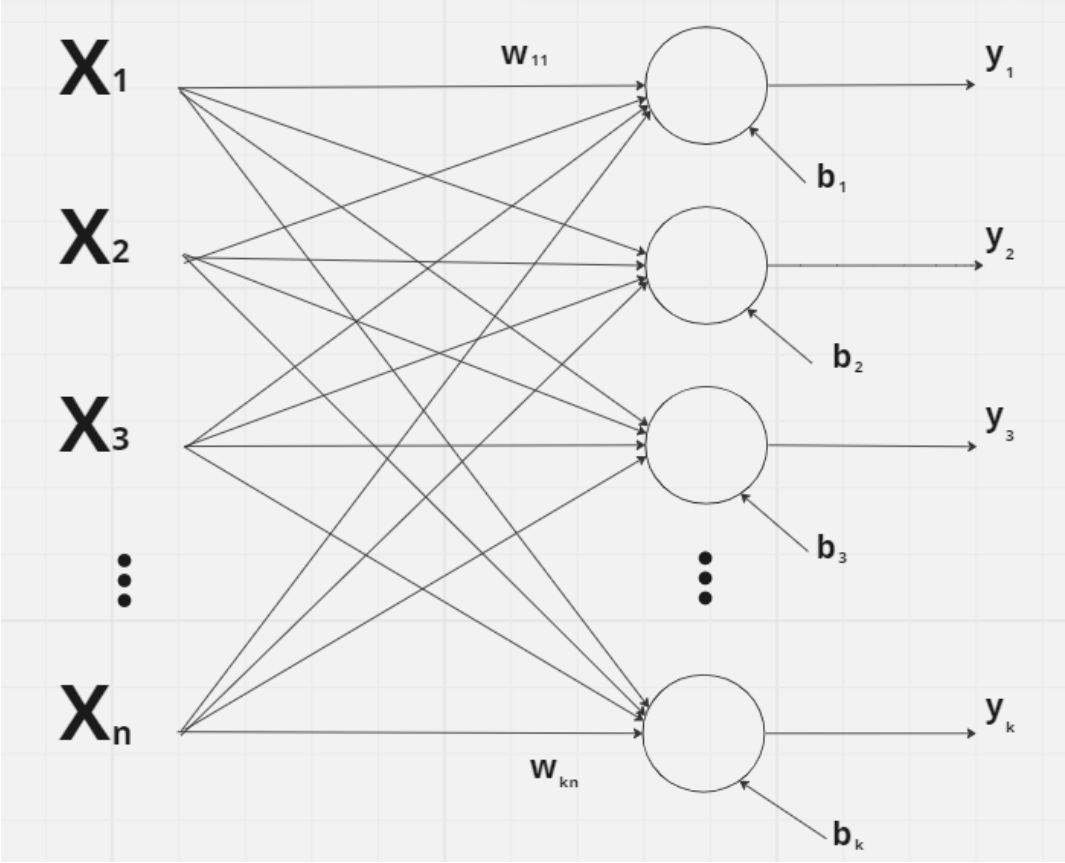
Rysunek 1: Model sztucznego neuronu

2.1.2 Sieć jednokierunkowa

Sieć neuronowa jednokierunkowa ma ułożone neurony w taki sposób,że przepływ sygnałów odbywa się wyłącznie w kierunku od wejścia (poprzez ewentualne warstwy ukryte) do wyjścia. Wykluczony jest przepływ sygnałów w drugą stronę. Jednym z najbardziej popularnych typów takiej sieci jest perceptron wielowarstwowy - MLP, jak i również LVQ.

2.1.3 Sieć jednowarstwowa

Architektuta takiej sieci jest prosta. Sieć jednowarstwowa jest zbudowana z jednej warstwy neuronów oraz sygnałów wejściowych i wyjściowych. Neurony działają niezależnie od siebie,przez co możliwości działania takiej sieci są ograniczone do możliwości pojedycznych neuronów.



Rysunek 2: Schemat sieci jednowarstwowej

2.1.4 Sieć wielowarstwowa

Sieci wielowarstwowe są jednymi z najczęściej wykorzystywanych architektur wśród sztucznych sieci neuronowych. Zbudowane są z neuronów rozmieszczonych w co najmniej trzech warstwach: warstwie wejściowej, warstwie wyjściowej oraz jednej lub więcej warstwach ukrytych.

W tego typu sieciach każdy neuron w warstwie wejściowej jest połączony z każdym wejściem sieci. Wejścia każdego neuronu w warstwach ukrytych i wyjściowej są z kolei połączone z wyjściami wszystkich neuronów z poprzedniej warstwy. Wyjścia neuronów z warstwy wyjściowej stanowią globalne wyjścia sieci.

Proces przetwarzania danych w sieci wielowarstwowej zaczyna się od wprowadzenia sygnałów wejściowych na wejścia neuronów warstwy wejściowej. Następnie przetworzone dane są przekazywane na wyjścia i trafiają na wejścia neuronów kolejnej warstwy. Ten proces powtarza się tyle razy, ile jest warstw ukrytych. Na końcu, sygnały wyjściowe z ostatniej warstwy ukrytej pełnią rolę sygnałów wejściowych dla warstwy wyjściowej.

Przedstawienie działania poszczególnych warstw:

- Dla pierwszej warstwy:

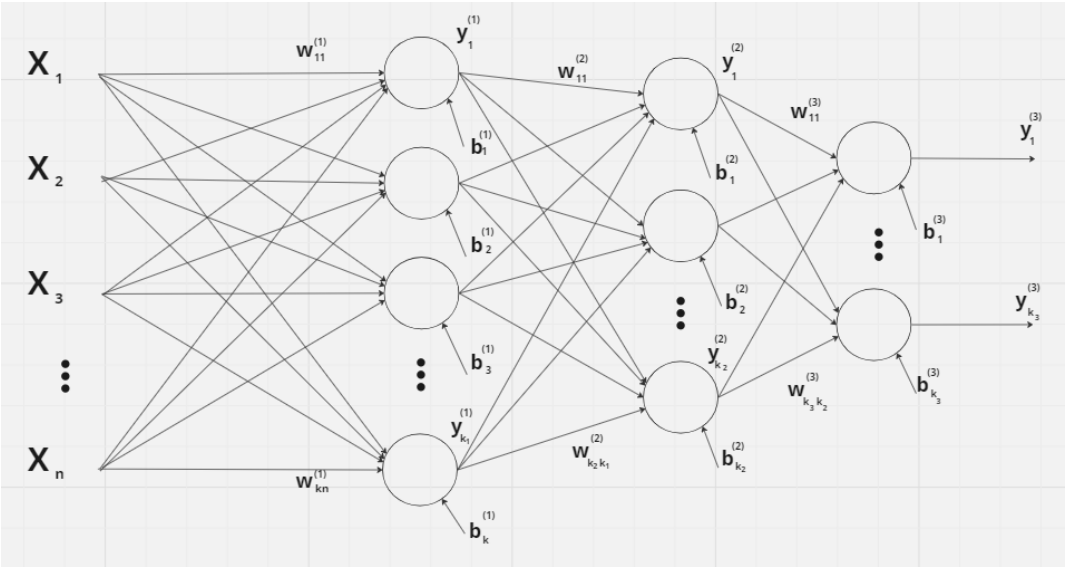
$$y^{(n)} = f^{(n)}(w^{(n)}x + b^{(n)}), n = 1 \tag{3}$$

- Dla pozostałych warstw:

$$y^{(n)} = f^{(n)}(w^{(n)}y^{(n-1)} + b^{(n)}), n \neq 1 \tag{4}$$

gdzie:

- $y^{(n)}$ - wyjście neuronów w warstwie n.
- $y^{(n-1)}$ - wyjście neuronów z poprzedniej warstwy (n - 1).
- $f^{(n)}$ - funkcja aktywacji używana w warstwie n.
- $w^{(n)}$ - macierz wag dla warstwy n.
- x - wektory wejściowe dla sieci (sygnały wejściowe dla pierwszej warstwy).
- $b^{(n)}$ - wektory przesunięcia dla warstwy n.



Rysunek 3: Przykładowa struktura sieci trójwarstwowej

2.2 Struktura sieci LVQ

Algorytm Learning Vector Quantization (LVQ) został zaproponowany przez Teuvo Kohonena jako technika klasyfikacji oparta na prototypach. Jest to jednokierunkowa, wielowarstwowa sieć neuronowa, będąca rozszerzeniem metody samoorganizujących się map (Self-Organizing Maps, SOM) do formy nadzorowanej.

Został opracowany jako technika klasyfikacji oparta na prototypach wykorzystująca dane treningowe z pożądaną klasą informacji do klasyfikacji danych.LVQ ma konkurencyjne podejście do uczenia się oparte na zasadzie "zwycięzca bierze wszystko".

Dane, które wprowadzamy do algorytmu LVQ w celu przeprowadzenia klasyfikacji, nazywane są wektorem wejściowym. Natomiast używane przez algorytm wektory do klasyfikacji nazywane są wektorami referencyjnymi,wektorami prototypowymi lub księgą kodów.

W architekturze sieci LVQ można wyróżnić dwie istotne warstwy: warstwę liniową z nadzorowanym uczeniem oraz warstwę Kohonena, znana również jako konkurencyjna. Warstwa Kohonena uczy się klasyfikować wektory wejściowe, identyfikując najbliższy neuron dla każdego wektora wejściowego. Klasy nauczone przez tę warstwę nazywane są podklasami.

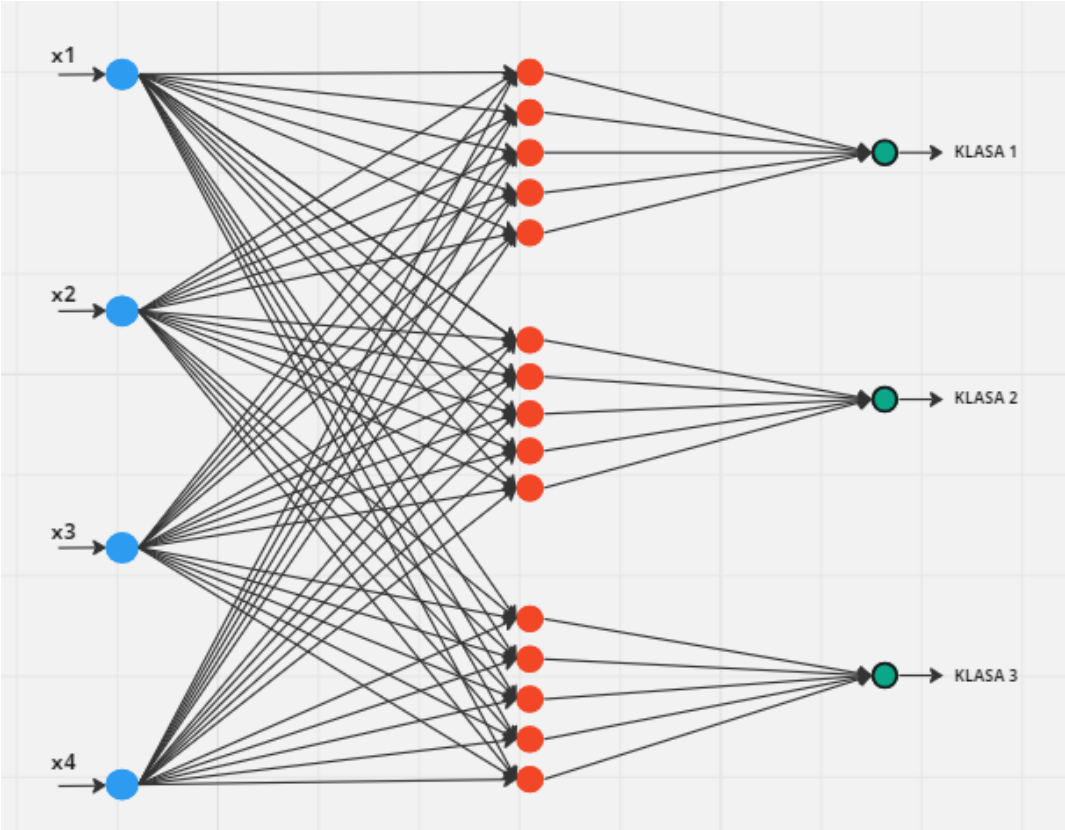
Warstwa liniowa przekształca nauczone klasy przez warstwę konkurencyjną w docelowe klasyfikacje zdefiniowane przez użytkownika. Klasy tej warstwy to klasy docelowe. Algorytm LVQ działa na zasadzie przyciągania i odpychania wektorów prototypowuch, które reprezentują typowe przykłady różnych klas w zestawie danych. Jeśli klasyfikacja jest poprawna, wektor wag zwycięskiego neuronu jest dostosowywany do kierunku wektora wejściowego, a błąd klasyfikacji regulowany jest w przeciwnym kierunku.

Dzięki temu procesowi algorytm uczy się dopasowywać prototypy dla wzorców danych i poprawnej klasyfikacji.

2.3 Rodzaje sieci LVQ

Istnieją rożne rodzaje sieci LVQ,w danej pracę skupimy się na 4 podstawowych algorytmach. Są to:

- LVQ1
- LVQ2
- LVQ2.1
- LVQ3



Rysunek 4: Przykładowa arcitektura sieci LVQ

2.3.1 LVQ1

LVQ1 - ten algorytm jest bazowym dla calej grupy. Wektory książki kodów są przypisywane do klas. Następnie szukamy prototypu, który ma najmniejszą odległość euklidesową od wybranego wektora wejściowego.Jeżeli znaleziony wektor prototypowy jest najbliszym danemu wektoru wejściowemu, uznajemy, że należą do tej samej klasy.

Odległość tych wektorów obliczmy za pomocą wzoru który ma postać:

$$d(x, w) = \sqrt{\sum_{i=1}^{n}\sum_{j=1}^{m}(x_i - w_{ij})^2} \tag{5}$$

gdzie:

- w_{ij} - to i-ta cecha j-tego prototypu
- x_i -to i-ta cecha wektora x (wektora wejściowego)
- d - odległość Euklidesowa między wektorem wejściowym, a prototypowym
- n - to liczba cech wektora wejściowego
- m - to liczba prototypów

Następnie, w warstwie konkurencyjnej, algorytm "zwycięzca bierze wszystko" jest stosowany. Oznacza to, że tylko jeden neuron z warstwy konkurencyjnej zostanie aktywowany, mając wartość 1, a reszta neuronów będzie miała wartość wyjścia równą 0. Algorytm uczenia nadzorowanego, nagradza poprawne klasyfikacje,jeżeli wektor prototypowy oraz wejściowy znajdują się w tej samej klasie, to prototyp jest przesuwany w kierunku do wektora wejściowego. Jeśli wektor należy to używamy wzoru:

$$w_i'(t+1) = w_i(t) + \alpha(t)(x_i(t) - w_i(t)) \tag{6}$$

gdzie:

- $w_i'(t+1)$ - nowy i-ty prototyp
- $w_i(t)$ - aktualny i-ty prototyp
- $\alpha(t)$ - współczynnik uczenia w danym kroku czasowym t
- $x_i(t)$ - i-ty wektor wejściowy

Jeśli wektor prototypowy i wektor wejściowy są w różnych klasach, to prototyp jest przesuwany w przeciwnym kierunku do wektora wejściowego, to używamy wzoru:

$$w_i'(t+1) = w_i(t) - \alpha(t)(x_i(t) - w_i(t)) \tag{7}$$

gdzie:

- $w_i'(t+1)$ - nowy i-ty prototyp
- $w_i(t)$ - aktualny i-ty prototyp
- $\alpha(t)$ - współczynnik uczenia w danym kroku czasowym t
- $x_i(t)$ - i-ty wektor wejściowy

2.3.2 Algorytm uczenia LVQ1

1. Inicjalizacja wektora prototypowego.
2. Wybór losowego wektora wejściowego x
3. Wykonuj kroki 4 - 8 , dopóki warunek stopu nie będzie fałszywy.
4. Wykonuj kroki 5 - 6 dla każdego wektora wejściowego
5. Oblicz odległość euklidesową, według wzoru (5).
6. Aktualizuj wagi neuronu prototypowego.Jeżeli wektor prototypowy i wejściowy w tej samiej klasie użyj wzoru (6), a jeśli w różnych (7).
7. Zmniejsz współczynnik uczenia
8. Sprawdż czy warunek stopu został spełniony.

   Warunkiem stopu może być liczba epoch.

2.3.3 LVQ2

LVQ2 został opracowany w celu poprawy skuteczności standardowego modelu LVQ. W szczególności sieć LVQ2 stara się zapobiec błędowi klasyfikacji przy wartościach granicznych klas. Oprócz standardowego LVQ, w trakcie treningu ta sieć proponuje jednoczesną zmianę dwóch najbliższych wektorów prototypowych do wektora wejściowego.

Aby do tej zmiany doszło muszą być spełnione warunki:

- Odległość między wektorem wejściowym, a zwycięskim neuronem, oraz drugim wektorem prototypowym muszą być prawie równe do siebie.
- Wektory prototypowe należą do różnych klas oraz drugi najbiższy wektor prototypowy należy do tej samej klasy co wektor wejściowy, a pierwszy nie.

Warunek sprawdzający tą odległość wyrażamy wzorami:

$$\frac{d_i}{d_j} > (1-\varepsilon) \wedge \frac{d_j}{d_i} < (1+\varepsilon) \tag{8}$$

gdzie:

- d_i - odległość między wektorem wejściowym, a najbliższym wektorem prototypowym
- d_j - odległość między wektorem wejściowym, a drugim najbiższym wektorem prototypowym
- ε - zmienna regulująca rozmiar okna

Aktualizacja prototypu, który jest "zwycięzcą", oddalając go od wektora wejściowego,jest definiowana wzorem:

$$w_i'(t+1) = w_i(t) - \alpha(t)(x(t) - w_i(t)) \tag{9}$$

gdzie:

- w_i' - zaktualizowany prototyp
- w_i - aktualny prototyp
- α - aktualny współczynnik uczenia
- x - wektor wejściowy

Dla neuronu będącego najbliższym sąsiadem,jego wagi modyfikowane w taki sposób, aby przybliżyć go od wektora wejściowego. Aktualizacja wag jest definiowana wzorem:

$$w_j'(t+1) = w_j(t) + \alpha(t)(x(t) - w_j(t)) \tag{10}$$

gdzie:

- w_j' - zaktualizowany prototyp
- w_j - aktualny prototyp
- α - aktualny współczynnik uczenia
- x - wektor wejściowy

2.3.4 LVQ2.1

W algorytmie LVQ2.1 klasyfikacja odbywa się analogicznie do LVQ1, czyli poprzez minimalizację wybranej metryki. W tym algorytmie modyfikowane są jednocześnie dwa najbliższe wektory z książki kodowej,które są najbliższymi sąsiadami wektora wejściowego.

Modyfikowanie wektorów zachodzi tylko wtedy gdy zostaną spełnione warunki:

1 Warunki dotyczące klas prototypów:

- Klasa wektora wejściowego x jest inna niż klasa prototypu w_i i taka sama jak klasa prototypu w_j.
- Klasa wektora wejściowego x jest inna niż klasa prototypu w_j i taka sama jak klasa prototypu w_i.

  Warunek ten jest spełniony, gdy jeden z powyższych przypadków jest prawdziwy.

2 Wektor wejściowy musi leżeć w tzw. oknie, które jest zdefiniowane wokół środkowej płaszczyzny między prototypami. Definicję względnej szerokości okna możemy zdefiniować za pomocą wzoru:

$$min\left[\frac{d_i}{d_j}, \frac{d_j}{d_i}\right] > (1-\varepsilon) \wedge max\left[\frac{d_i}{d_j}, \frac{d_j}{d_i}\right] < (1+\varepsilon) \tag{11}$$

gdzie:

- d_i - odległość od wektora wejściowego x do prototypu w_i
- d_j - odległość od wektora wejściowego x do prototypu w_j
- ε - zmienna regulująca rozmiar okna

Jeżeli powyższe warunki są spełnione, i załóżmy, że w_i należy do poprawnej klasy, aktualizacja prototypów przebiega następująco.

Prototyp który należy do tej samej klasy co wektor wejściowy jest przesuwany w kierunku wektora wejściowego x zgodnie z następującym wzorem:

$$w_i'(t+1) = w_i(t) + \alpha(t)(x(t) - w_i(t)) \tag{12}$$

gdzie:

- w_i' - zaktualizowany prototyp
- w_i - aktualny prototyp
- α - aktualny współczynnik uczenia
- x - wektor wejściowy

Prototyp, który nie należy do tej samej klasy co wektor wejściowy, jest przesuwany z dala od wektora wejściowego x zgodnie z następującym wzorem:

$$w_j'(t+1) = w_j(t) - \alpha(t)(x(t) - w_j(t)) \tag{13}$$

gdzie:

- w_j' - zaktualizowany prototyp
- w_j - aktualny prototyp
- α - aktualny współczynnik uczenia
- x - wektor wejściowy

2.3.5 LVQ3

Ten algorytm umożliwia naukę dwóch wektorów prototypowych najbliższych wektorowi wejściowemu,jeśli zostanie spełniony warunek:

$$min\left[\frac{d_i}{d_j}, \frac{d_j}{d_i}\right] > (1-\varepsilon)(1+\varepsilon) \tag{14}$$

gdzie:

- d_i - odległość od wektora wejściowego x do prototypu w_i
- d_j - odległość od wektora wejściowego x do prototypu w_j
- ε - zmienna regulująca rozmiar okna

Jeśli jeden prototyp należy do klasy wektoru wejściowego, a drugi nie nalezy,to modufikacja prototypów przybiega jak w przypadku LVQ2.1.

Modyfikacja prototypów, w przypadku gdy obydwa należą do klasy odpowiadającej wektorowi wejściowemu:

$$w'(t+1) = w(t) + m\alpha(t)(x(t) - w(t)) \tag{15}$$

gdzie:

- w' - zaktualizowany prototyp
- w - aktualny prototyp
- α - aktualny współczynnik uczenia
- x - wektor wejściowy
- m - parametr względnego tempa uczenia(odpowiednie wartości zawierają się w przedziale 0.1 do 0.5)

2.4 Walidacja krzyżowa

Metoda walidacji krzyżowej jest często wykorzystywana do oceny zdolności sieci.

Ta metoda polega na podziale zbioru danych wejściowych na serię podzbiorów. Z nich jeden jest wybierany jako zbiór walidacyjny, natomiast pozostałe podzbiory służą do uczenia sieci. Walidacja krzyżowa jest szczególnie przydatna, kiedy liczba dostępnych danych wejściowych jest ograniczona. W projekcie została zastosowana walidacja n - krotna. Sposób ten dzieli zbiór danych uczących na n podzbiorów, które zazwyczaj są równoliczne. Jeden z tych podzbiorów jest wykorzystywany do testowania sieci, podczas gdy pozostałe służą do jej uczenia.

Procedura ta jest powtarzana n razy, tak aby każdy podzbiór mógł pełnić rolę zbioru walidacyjnego. Skuteczność sieci jest zwykle określana jako średnia błędów klasyfikacji uzyskanych dla każdego z osobnych zbiorów walidacyjnych.

3 Analiza zbioru danych

3.1 Przedstawienie zbioru

Dane używane w projekcie pochodzą z bazy danych Hepatitis Data Set. Zbiór ten zawiera informacje medyczne dotyczące pacjentów z zapaleniem wątroby (hepatitis). Celem analizy jest klasyfikacja pacjentów na podstawie ich cech fizycznych, aby przewidzieć, czy pacjent przeżył chorobę czy zmarł.Zestaw zawiera 155 rekordów przed odpowiednim przygotowaniem danych.

Baza danych zawiera 19 atrybutów opisujących stan zdrowia pacjentów z zapaleniem wątroby. Poniżej znajduje się lista atrybutów :

1. Wiek
2. Płeć
3. Stosowanie steroidów
4. Stosowanie leków przeciwwirusowych
5. Zmęczenie
6. Osłabienie
7. Anoreksja
8. Powiększona wątroba
9. Twarda wątroba
10. Wyczuwalna śledziona
11. Naczyniaki pajątkowe
12. Wodobrzusze
13. Żylaki
14. Poziom bilirubiny
15. Poziom fosfatazy alkanicznej
16. Poziom transamizany glutaminowej
17. Poziom albuminy
18. Czas protrombinowy
19. Wyniki histologiczne

Zbiór danych zawiera brakujące wartości dla niektórych atrybutów.

Dane zostały podzielone na dwie klasy:

- Przeżył
- Zmarł

3.2 Przygotowanie oraz normalizacja danych

Przed rozpocięciem eksperymentów potrzebujemy przygotować dane. Zbiór ma brakujące dane i nie jest znormalizowany.

```
import hickle as hkl
import numpy as np
import matplotlib.pyplot as plt # Import bibliotek

filename = open("hepatitis.txt") # Otwieranie pliku z danymi
data = np.loadtxt(filename, delimiter=',', dtype=str) # Wczytanie danych z
    pliku jako tablica numpy z elementami typu string

data = data[(data=='?').any(axis=1)==0] #usunięcie rekordów z brakującymi
    danymi

x = data[:, 1:].astype(float).T # Ustawienie cech wejściowych
y_t = data[:,0].astype(float) #ustawienie wyjścia pożądanego
y_t = y_t.reshape(1,y_t.shape[0]) #zmiana na jednowierszową tablicę

# Wypisanie zakresu wartości dla każdej cechy przed normalizacją
print(np.transpose([np.array(range(x.shape[0])), x.min(axis=1),
x.max(axis=1)]))

x_min = x.min(axis=1) # Minimalne wartości dla każdej cechy
x_max = x.max(axis=1) # Maksymalne wartości dla każdej cechy
x_norm_max = 1 # Maksymalna wartość po normalizacji
x_norm_min = -1 # Minimalna wartość po normalizacji
x_norm = np.zeros(x.shape) # Inicjalizacja znormalizowanej tablicy z zerami

# Normalizacja danych do zakresu [-1, 1]
for i in range(x.shape[0]):
  x_norm[i,:] = (x_norm_max-x_norm_min)/(x_max[i]-x_min[i])* \
  (x[i,:]-x_min[i]) + x_norm_min

print(np.transpose([np.array(range(x.shape[0])), x_norm.min(axis=1),
x_norm.max(axis=1)])) #Wypisanie znormalizowanych zbiorów

y_t_s_ind = np.argsort(y_t)  # Indeksy sortujące y_t
x_n_s = np.zeros(x.shape)  # Inicjalizacja posortowanej tablicy cech
y_t_s = np.zeros(y_t.shape) # Inicjalizacja posortowanej tablicy wyjść

# Sortowanie danych na podstawie posortowanych wyjść
for i in range(x.shape[1]):
  y_t_s[0,i] = y_t[0,y_t_s_ind[0,i]] # Przypisanie posortowanych wartości wyjść
  x_n_s[:,i] = x_norm[:,y_t_s_ind[0,i]] # Przypisanie posortowanych cech

plt.plot(y_t_s[0]) # Wykres posortowanego zbioru wyjść
plt.show()  # Wyświetlenie wykresu

# Zapisanie wyników do pliku hepatitis2.hkl
hkl.dump([x,y_t,x_norm,x_n_s,y_t_s],"hepatitis2.hkl")
# Wczytanie wyników z pliku hepatitis2.hkl
x,y_t,x_norm,x_n_s,y_t_s = hkl.load("hepatitis2.hkl")
```

Listing 1: Skrypt dla przygotowania oraz normalizacji danych

Pierwszym krokiem w zbiorze danych było usunięcie wierszy z brakującymi danymi.Po usunieciu zestaw zawierał już nie 155 rekordów, a 80, widzimy że nastąpiło ich istotne zmniejszenie.

Kolejnym krokiem była normalizacja danych, dzięki czemu będzie możlwość ich dalszej analizy.Atrybuty poza klasami zostały przekształcone tak, aby ich nowa wartość znajdowała się w zakresie od -1 do 1.

Wzór do wykorzystania:

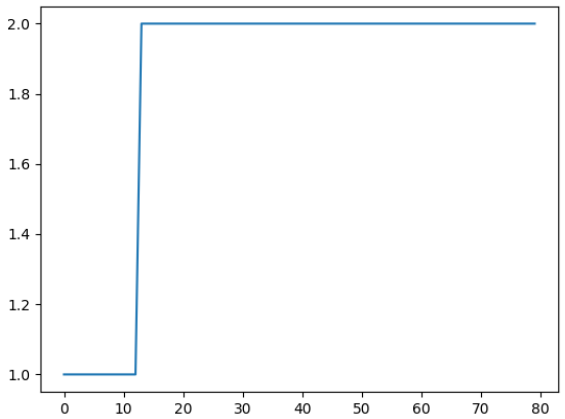
$$x_{norm} = \frac{x_{normMax} - x_{normMin}}{x_{max} - x_{min}} \cdot (x - x_{min}) + x_{normMin} \tag{16}$$

gdzie:

- x_{norm} - wartość znormalizowana
- $x_{normMax}$ - maksymalna wartość cechy po normalizacji
- $x_{normMin}$ - minimalna wartość cechy po normalizacji
- x - oryginalna wartość, którą chcemy znormalizować
- x_{max} - maksymalna wartość cechy przed normalizacją
- x_{minx} - minimalna wartość cechy przed normalizacją

Po normalizacji, minimalne i maksymalne wartości każdej cechy zostały wyświetlone, aby potwierdzić poprawność przekształceń. Następnie dane zostały posortowane według wartości wyjść (klas).

Sortowane dane zostały zwizualizowane na wykresie, co umożliwiło sprawdzenie rozkładu wartości klas po przetwarzaniu. Na koniec przetworzone dane (oryginalne cechy, wyjścia, znormalizowane cechy oraz posortowane zbiory) zostały zapisane do pliku w formacie Hickle "hepatitis2.hkl".



Rysunek 5: Graficzne przedstawienie danych posortowanych

4 Skrypt programu

Algorytm realizujący sieć neuronową LVQ został napisany w języku Python.Skrypt jest zaprojektowany do optymalizacji metaparametrów sieci LVQ takich jak:

1. Step (krok):
   - Parametr step określa wielkość kroku aktualizacji wag podczas procesu uczenia.
   - Im większa wartość parametru step, tem większe zmiany wag mogą zachodzić podczas aktualizacji.
   - Wartość step powinna być odpowiednio dobrana, aby uniknąć nadmiernego drgania wag(oscylacji) lub zbyt małej szybkości uczenia.
2. Minstep (minimalny krok):
   - Parametr minstep to minimalna wartość, jaką krok może przyjąć.
   - Jest to używane w celu kontrolowania stopnia zmiany wag.
   - Gdy krok staje się mniejszy niż wartość minstep, przestaje się zmieniać, zapobiegając nadmiarnemu dopasowaniu modelu do danych treningowych.
3. n_updates_to_stepdrop (liczba aktualizacji do zmniejszania kroku):
   - Parametr n_updates_to_stepdrop określa, co ile aktualizacji wag następuje zmniejszenie wartości kroku.
   - Jest to technika, która pozwala na zmniejszenie kroku w miarę postępu w uczeniu.
   - Po upływie określonej liczby aktualizacji wag, wartość kroku może zostać zmniejszona, aby precyzyjnej dopasować model.

Kod realizujący sieć neuronową LVQ:

```
from sklearn.model_selection import StratifiedKFold
import hickle as hkl
from timeit import default_timer as timer
import matplotlib.pyplot as plt
import numpy as np
from neupy import algorithms

# Odczyt danych z pliku i przypisanie ich do zmiennych
x,y_t, x_norm ,x_n_s, y_t_s = hkl.load('hepatitis.hkl')
y_t -= 1 # Dostosowanie etykiet klas
x = x.T # Transpozycja macierzy x
y_t = np.squeeze(y_t) # Usuwanie jednowymiarowych osi z y_t
data = x_norm.T # Przypisanie transponowanej macierzy 'x_norm' do zmiennej data
target = y_t # Przypisanie wektora zawierającego etykiety do zmiennej target

epoch = 500 # Ustalenie liczby epok do treningu modelu

# Definiowanie metaparametrów
# Wartości dla parametru 'step'
step_vec = np.array([0.5, 0.1, 1e-3, 1e-6, 1e-9, 1e-17, 1e-25])
```

```
# Wartości dla parametru 'n_updates_to_stepdrop'
n_updates_to_stepdrop_vec = np.array([10, 100, 500, 1000, 2500, 5000, 10000])
# Wartości dla parametru 'minstep'
minstep_vec = np.array([0.1, 1e-5, 1e-9, 1e-16, 1e-19, 1e-27])

start = timer() # Rozpoczynanie pomiaru czasu
CVN = 10 # Ustalenie liczby foldow dla walidacji krzyżowej
skfold = StratifiedKFold(n_splits=CVN) # Przygotowanie do wykonania walidacji
   krzyżowej

# Inicjalizacja zmiennych, które przechowują najlepsze parametry
best_step = 0 # Najlepsza wartośc 'step'
best_n_updates_to_stepdrop = 0 # Najlepsza wartość 'n_updates_to_stepdrop'
best_minstep = 0 # Najlepsza wartość 'minstep'
best_PK = 0 # Najlepsza wartość poprawności klasyfikacji
best_PK_minstep = 0 # Najlepsza wartość poprawności klasyfikacji, używana
   podczas iteracji 'minstep'
temp_minstep = 1e-40 #Tymczasowa wartość 'minstep', użyta dla testowania 'step'
    oraz 'n_updates_to_stepdrop'

# Macierz przechowująca wartości poprawności klasyfikacji dla kombinacji 'step'
    oraz 'n_updates_to_stepdrop'
PK_values = np.zeros((len(step_vec),len(n_updates_to_stepdrop_vec)))

# Pętla przechodząca przez różne wartości parametru 'step'
for steps in range(len(step_vec)):
  # Pętla przechodząca przez różne wartości parametru 'n_updates_to_stepdrop'
  for n_updates_to_stepdrop_ in range(len(n_updates_to_stepdrop_vec)):
    print("Step: ", step_vec[steps], "n_updates_to_stepdrop_vec: ",
    n_updates_to_stepdrop_vec[n_updates_to_stepdrop_]) # Wyświetlanie aktualnej
    kombinacji 'step' i 'n_updates_t _stepdrop'
    # Wektor przechowujący wyniki poprawności klasyfikacji  każdego folda
    PK_vec = np.zeros(CVN)
      # Pętla po wszystkich foldach walidacji krzyżowej
      for i, (train, test) in enumerate(skfold.split(data, target),start = 0):
        # Definicja zbioru treningowego i testowego dla danych wejściowych
        x_train , x_test = data[train], data[test]
        # Definicja zbioru treningowego i testowego dla danych wyjściowych
        y_train,  y_test = target[train], target[test]
        # Tworzenie instancji algorytmu LVQ z określonymi parametrami
        lvqnet = algorithms.LVQ(
          n_inputs = x_train.shape[1], # Liczba jednostek wejściowych
          n_classes = np.unique(y_train).shape[0], # Liczba klas w zestawie
   danych
          step = step_vec[steps], # Współczynnik uczenia
          n_updates_to_stepdrop = n_updates_to_stepdrop_vec[
   n_updates_to_stepdrop_], # Liczba aktualizacji zmniejszania kroku
          minstep = temp_minstep # Testowy 'minstep'
        )
        lvqnet.train(x_train,y_train, epochs=epoch) # Trening sieci na zbiorze
   treningowym
        result = lvqnet.predict(x_test) # Przewidywanie etykiet dla zestawu
   testowego
        n_test_samples = test.size # Obliczenie liczby próbek w zestawie
   treningowym
        PK_vec[i] = (np.sum(result == y_test) / n_test_samples) * 100 #
```

```
    Obliczanie precyzji poprawności klasyfikacji

        # Wypisanie na ekran aktualnych danych dla biezącego podziału
    kroswalidacji
        print("Test #{:<2}: PK_vec {} test_size   {}".format(i, PK_vec[i],
    n_test_samples))

     PK = np.mean(PK_vec) # Średnia poprawność klasyfikacji dla wszystkich
    podziałów kroswalidacji
     PK_values[steps,n_updates_to_stepdrop_] = PK # Przypisanie średniej PK do
    tablicy PK_values na odpowiadające kombinacje 'step' oraz '
    n_updates_to_stepdrop'

     # Czy obliczona średnia jest większa niż obecnie najlepsze PK
     if PK > best_PK:
       best_PK = PK # Jeżeli tak, przypisywana jest najwyższa średnia wartość PK
     do 'best_PK'
       best_step = step_vec[steps] # Przypisanie najlepszego kroku, dla najwyż
    szego PK
       best_n_updates_to_stepdrop = n_updates_to_stepdrop_vec[
    n_updates_to_stepdrop_] # Najlepsze 'n_updates_to_stepdrop'

     print("Srednie PK dla wykonanej iteracji: {}\n".format(PK))
print("Najlepsze parametry:\nPoprawnosc Klasyfikacji: {}\nstep: {}\
    nn_updates_to_stepdrop: {}".format(best_PK,best_step,
    best_n_updates_to_stepdrop))
print("Czas wykonania:", timer()-start) # Wyświetlanie najlepszych parametrów

# Rysowanie 3D wykresu
fig = plt.figure(figsize = (8,8)) # Tworzy nową figurę do rysowania wykresu, o
    rozmiarze 8x8
ax = fig.add_subplot(111, projection='3d') # Tworzy osie do rysowania 3D na
    objekcie figury. '111' , oznacza, że jest utworzony pojedynczy wykres na
    figurze
X,Y = np.meshgrid(np.log10(step_vec), n_updates_to_stepdrop_vec) #Tworzy siatkę
     współrzędnych dla wykresów 3D dla 'step_vec', którego wartości są
    logarytmowane i 'n_updates_to_stepdrop'
surf = ax.plot_surface(X,Y,PK_values.T, cmap='viridis') # Tworzy powierzchnie 3
    D używając X,Y i transpozycji macierzy PK_values jako z. 'Viridis' to mapa
    kolorów kolorujących przestrzeń
ax.set_xlabel('log10(step)') # Ustawienie etykiety osi x jako 'log10(step)'
ax.set_ylabel('n_updates_to_stepdrop') # Ustawienie etykiety osi y na '
    n_updates_to_stedrop'
ax.set_zlabel('PK') # Ustawienie etykiety osi z na 'PK'
plt.show() # Wyświetlanie utworzonego wykresu

start = timer() #Rozpoczęcie pomiaru czasu

PK_values_minstep = np.zeros(len(minstep_vec)) # Tablica używana do
    przechowywania poprawności klasyfikacji zależnej od 'minstep'
# Pętla przechodząca przez różne wartości parametru 'minstep'
for minstep_index in range(len(minstep_vec)):
  print("Minstep: ",minstep_vec[minstep_index]) # Wyświetlenie aktualnej wartoś
    ci 'minstep'
  PK_vec = np.zeros(CVN) # Wektor przechowujący wyniki poprawności klasyfikacji
     kazdego folda
```

```
  # Pętla po wszystkich foldach walidacji krzyżowej
  for i,(train,test) in enumerate(skfold.split(data, target), start = 0):
    x_train,x_test = data[train], data[test] # Definicja zbioru treningowego i
   testowego dla danych wejściowych
    y_train,y_test = target[train], target[test] # Definicja zbioru
   treningowego i testowego dla danych wyjściowych
    # Tworzenie instancji algorytmu LVQ z określonymi parametrami
    lvqnet = algorithms.LVQ(
      n_inputs = x_train.shape[1],# Liczba jednostek wejściowych
      n_classes = np.unique(y_train).shape[0],# Liczba klas w zestawie danych
      step = best_step, # Learning rate dla którego osiągnięto najwyższe PK
      n_updates_to_stepdrop = best_n_updates_to_stepdrop, # Liczba aktualizacji
    zmniejszania kroku dla którego osiągnięto najwyższe PK
      minstep = minstep_vec[minstep_index] # Minimalna wartość kroku
    )
    lvqnet.train(x_train,y_train,epochs=epoch) # Trening sieci na zbiorze
   treningowym
    result = lvqnet.predict(x_test) # Przewidywanie etykiet dla zestawu
   testowego
    n_test_samples = test.size  # Obliczenie liczby próbek w zestawie
   treningowym
    PK_vec[i] = (np.sum(result == y_test)/n_test_samples) * 100 # Obliczanie
   precyzji poprawności klasyfikacji

    print("Test #{:<2}: PK_vec {} test_size {} minstep: {}".format(i,PK_vec[i],
   n_test_samples,minstep_vec[minstep_index]))

  PK = np.mean(PK_vec) # Średnia poprawność klasyfikacji dla wszystkich foldów
  PK_values_minstep[minstep_index] = PK # Średnia poprawność klasyfikacji dla
    aktualnej wartości 'minstep'
  # Czy obliczona średnia jest większa niż obecnie najlepsze PK
  if PK > best_PK_minstep:
    best_PK_minstep = PK # Jeżeli tak to aktualizowana jest najlepsza wartość
   PK
    best_minstep = minstep_vec[minstep_index] # Aktualizacja 'minstep' dla
   najlepszej poprawności

# Rysowanie wykresu zależności między 'minstep' i PK
plt.figure(figsize=(8,6)) # Tworzy nową figurę do rysowania wykresu o rozmiarze
    8x6
plt.plot(minstep_vec,PK_values_minstep, 'o-') # Rysuje liniowy wykres '
   minstep_vec' na osi x i 'PK_values_minstep' na osi y. Punkty danych są okrę
   gami, a linia jest ciągła
plt.xscale('log') # Skala osi x jest logarytmiczna
plt.title('Zależność PK od minstep') # Ustalanie tytułu
plt.xlabel('minstep') # Ustalanie etukiety osi x na 'minstep'
plt.ylabel('PK') # Ustalenie etykiety osi y na 'PK'
plt.grid(True) # Dodawanie siatki do wykresu
plt.show() #Wyświetlanie utworzonego wykresu


print("Najlepsze parametry:\nPoprawność Klasyfikacji: {} minstep: {}".format(
   best_PK_minstep,best_minstep))
print("Czas wykonania:", timer()-start) # Wyświetlenie najlepszych parametrów
   dla eksperymetu
```

Listing 2: Implementacja algorytmu

Skrypt zaczyna się od ładowania danych z pliku hepatitis.hkl oraz przetwarzania ich w celu przygotowania do treningu. Dane są dzielone na zbiory treningowe i testowe przy użyciu walidacji krzyżowej (StratifiedKFold) z dziesięcioma foldami. Następnie, dla różnych wartości metaparametrów step i n_updates_to_stepdrop, algorytm LVQ jest trenowany i testowany na danych, a dokładność klasyfikacji (PK) jest obliczana dla każdej kombinacji.

W celu znalezienia najlepszych wartości metaparametrów, średnie dokładności klasyfikacji są zapisywane i analizowane. Najlepsze wartości metaparametrów są wybierane na podstawie najwyższej średniej dokładności klasyfikacji. Po znalezieniu optymalnych wartości dla step i n_updates_to_stepdrop, skrypt przeprowadza dodatkowe testy, aby znaleźć najlepszą wartość minstep.

Wyniki eksperymentów są wizualizowane na wykresach, co ułatwia analizę wpływu różnych wartości metaparametrów na dokładność klasyfikacji. Na koniec, skrypt wyświetla najlepsze znalezione wartości metaparametrów oraz czas wykonania eksperymentów.

5 Eksperymenty

5.1 Eksperyment pierwszy

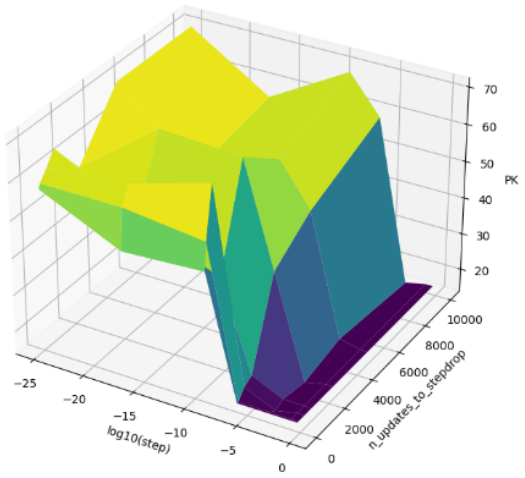
Pierwszy eksperyment polegał na znalezeniu parametrów step i n_updates_to_stepdrop, dla których poprawność klasyfikacji będzie największa.

Wartości badanych parametrów wynoszą:

- step : 0.5, 0.1, 1e-3, 1e-6, 1e-9, 1e-17, 1e-25
- n_updates_to_stepdrop : 10, 100, 500, 1000, 2500, 5000, 10000
- epoch : 10, 100, 500, 1000

Algorytm LVQ1

Liczba epok = 1000

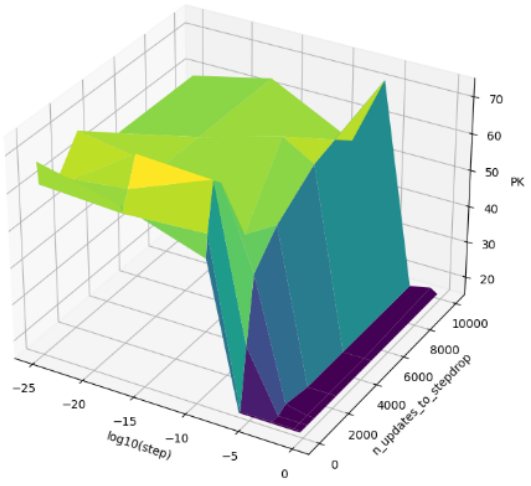


Rysunek 6: Przedstawienie wyników eksperymentu pierwszego dla LVQ1, liczba epok - 1000

Optymalne parametry:

- Poprawność klasyfikacji: 71.25%
- step: 1e-17
- n_updates_to_stepdrop: 2500

Liczba epok = 500

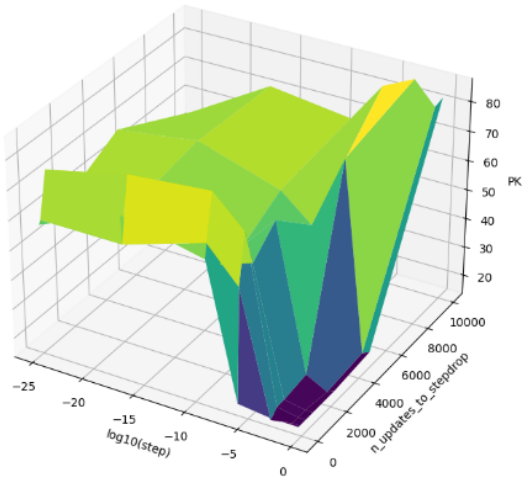


Rysunek 7: Przedstawienie wyników eksperymentu pierwszego dla LVQ1, liczba epok - 500

Optymalne parametry:

- Poprawność klasyfikacji: 73.75%
- step: 1e-9
- n_updates_to_stepdrop: 500

Liczba epok = 100

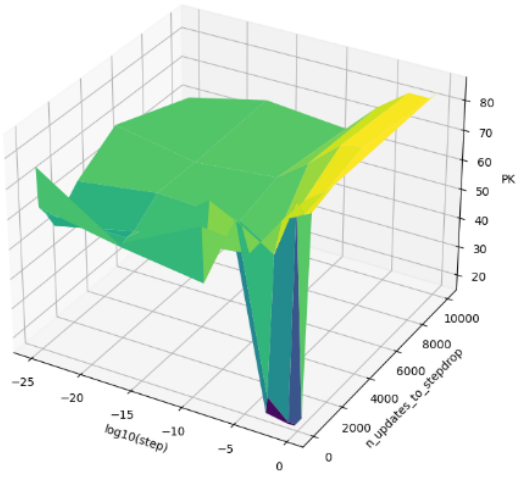


Rysunek 8: Przedstawienie wyników eksperymentu pierwszego dla LVQ1, liczba epok - 100

Optymalne parametry:

- Poprawność klasyfikacji: 86.25%
- step: 1e-3
- n_updates_to_stepdrop: 10000

Liczba epok = 10



Rysunek 9: Przedstawienie wyników eksperymentu pierwszego dla LVQ1, liczba epok - 10

Optymalne parametry:

- Poprawność klasyfikacji: 86.25%
- step: 0.1
- n_updates_to_stepdrop: 500

Podsumowanie najlepszych parametrów dla LVQ1

epoch	step	n_updates_to_stepdrop	Poprawność klasyfikacji
1000	1e-17	2500	71.25%
500	1e-9	500	73.75%
100	1e-3	10000	86.25%
10	0.1	500	86.25%

Tabela 1: Zestawienie najlepszych wyników dla algorytmu LVQ1

Najlepsze wyniki dla algorytmu LVQ1 osiągnięto przy 100 oraz 10 epokach, z poprawnością klasyfikacji wynoszącą 86.25%. Dla 100 epok optymalne wartości to step wynoszący 1e-3 oraz n_updates_to_stepdrop wynoszący 10000, natomiast dla 10 epok parametry wynosiły odpowiednio 0.1 oraz 500.

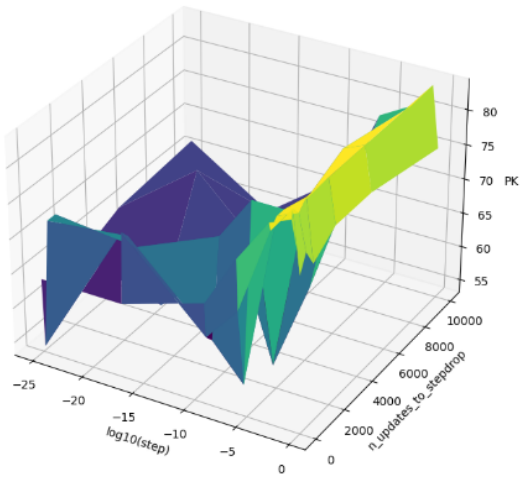
Możemy zauważyć że mniejsza liczba epok prowadzi do wyższej poprawności klasyfikacji, co może wynikać z szybszej konwergencji algorytmu oraz redukcji ryzyka przeuczenia modelu. Wartości parametru step są znacznie mniejsze przy większej liczbie epok, co sugeruje, że dłuższy trening wymaga bardziej precyzyjnych aktualizacji wag. Natomiast przy mniejszej liczbie epok, większe wartości step pozwalają na szybsze dostosowanie modelu do danych.

Parametr n_updates_to_stepdrop wykazuje tendencję do dostosowywania liczby aktualizacji przed zmniejszeniem kroku uczenia się w zależności od liczby epok. Przy większej liczbie epok (1000 i 500), optymalne wartości były mniejsze (2500 i 500), co sugeruje potrzebę częstszego dostosowywania kroku uczenia się w dłuższych treningach, aby uniknąć zbyt szybkiego zmniejszania kroku. Natomiast przy mniejszej liczbie epok (100 i 10), optymalne wartości były większe (10000 i 500), co pozwala na bardziej stopniowe dostosowanie kroku uczenia się przy krótszym treningu.

Podsumowując, wyniki eksperymentu sugerują, że algorytm LVQ1 osiąga najlepszą poprawność klasyfikacji przy mniejszej liczbie epok, co jest związane z szybszą konwergencją i unikaniem przeuczenia. Optymalne wartości parametrów step i n_updates_to_stepdrop są zależne od liczby epok i powinny być dostosowywane do specyfiki problemu klasyfikacyjnego oraz charakterystyki danych.

Algorytm LVQ2

Liczba epok = 1000

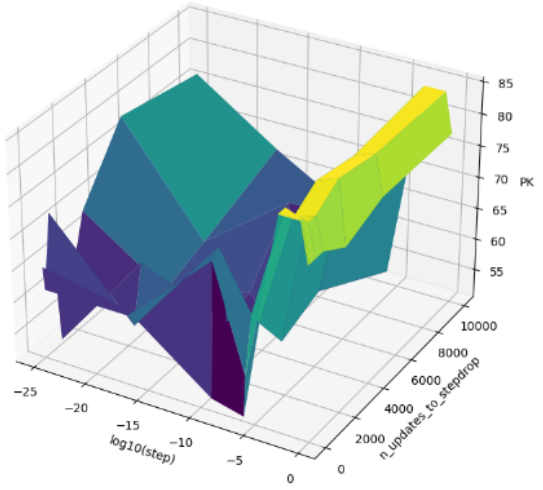


Rysunek 10: Przedstawienie wyników eksperymentu pierwszego dla LVQ2, liczba epok - 1000

Optymalne parametry:

- Poprawność klasyfikacji: 83.75%
- step: 0.1
- n_updates_to_stepdrop: 100

Liczba epok = 500

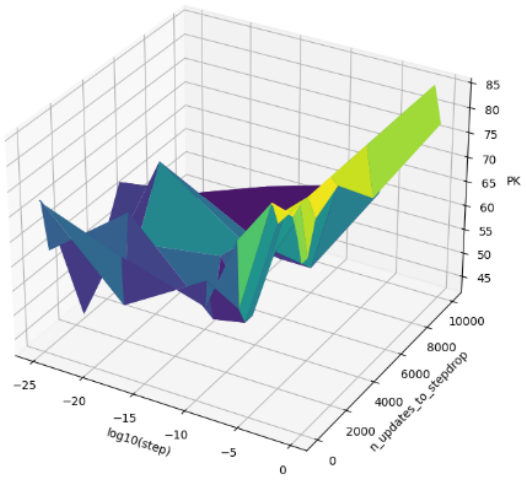


Rysunek 11: Przedstawienie wyników eksperymentu pierwszego dla LVQ2, liczba epok - 500

Optymalne parametry:

- Poprawność klasyfikacji: 85%
- step: 0.1
- n_updates_to_stepdrop: 2500

Liczba epok = 100

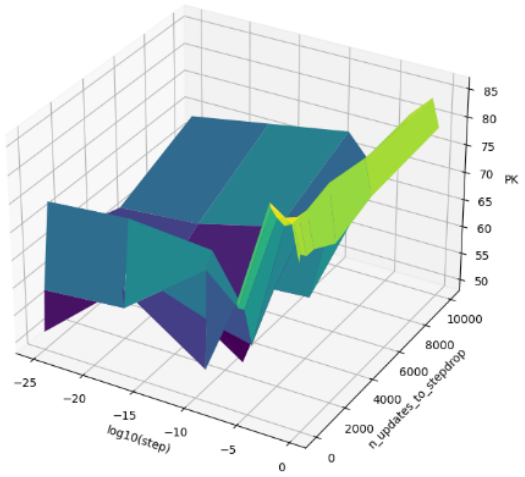


Rysunek 12: Przedstawienie wyników eksperymentu pierwszego dla LVQ2, liczba epok - 100

Optymalne parametry:

- Poprawność klasyfikacji: 85%
- step: 0.1
- n_updates_to_stepdrop: 500

Liczba epok = 10



Rysunek 13: Przedstawienie wyników eksperymentu pierwszego dla LVQ2, liczba epok - 10

Optymalne parametry:

- Poprawność klasyfikacji: 86.25%
- step: 1e-3
- n_updates_to_stepdrop: 10

Podsumowanie najlepszych parametrów dla LVQ2

epoch	step	n_updates_to_stepdrop	Poprawność klasyfikacji
1000	0.1	100	83.75%
500	0.1	2500	85%
100	0.1	500	85%
10	1e-3	10	86.25%

Tabela 2: Zestawienie najlepszych wyników dla algorytmu LVQ2

Wyniki pokazują, że poprawność klasyfikacji nieznacznie różni się między różnymi liczbami epok. Dla liczby epok równych 1000, 500 i 100 uzyskano poprawność na poziomie 83.75%, 85% oraz 85% odpowiednio. Najlepszy wynik poprawności (86.25%) osiągnięto przy tylko 10 epokach.

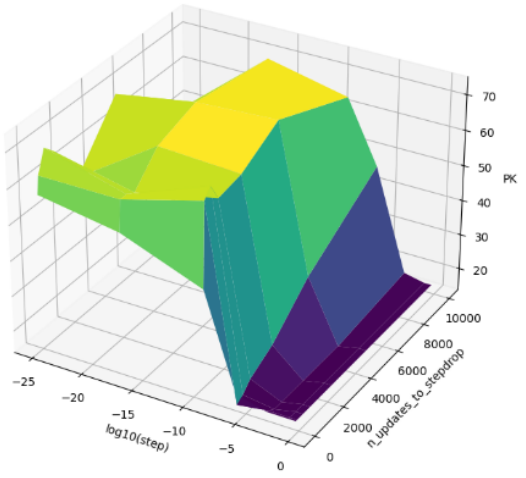
W porównaniu do algorytmu LVQ1, wyniki te sugerują, że LVQ2 może być bardziej stabilny, ponieważ osiąga podobną lub lepszą poprawność klasyfikacji przy różnych liczbach epok, co świadczy o jego zdolności do radzenia sobie z różnymi warunkami treningowymi.

Przyglądając się optymalnym parametrom dla algorytmu LVQ2, możemy zauważyć pewne trendy. Parametry te, takie jak step i n_updates_to_stepdrop, wykazują pewną zależność od liczby epok. Na przykład, dla mniejszej liczby epok (10), optymalne wartości step wynoszą 1e-3, a n_updates_to_stepdrop wynosi 10, co sugeruje potrzebę mniejszej liczby aktualizacji przy krótszym procesie uczenia się.

Podsumowując, algorytm LVQ2 wykazuje stabilność i zdolność do dostosowywania się do różnych ustawień parametrów.

Algorytm LVQ2.1

Liczba epok = 1000

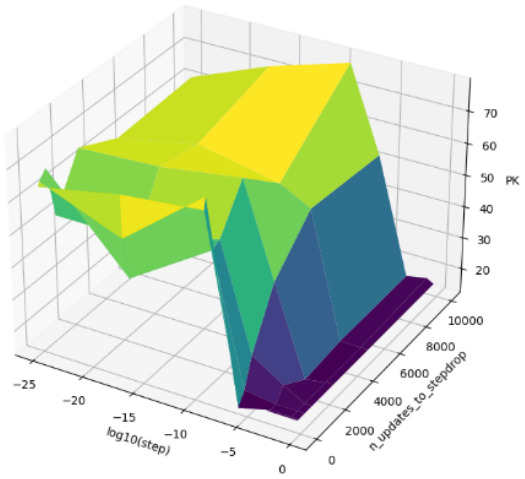


Rysunek 14: Przedstawienie wyników eksperymentu pierwszego dla LVQ2.1, liczba epok - 1000

Optymalne parametry:

- Poprawność klasyfikacji: 73.75%
- step: 1e-9
- n_updates_to_stepdrop: 5000

Liczba epok = 500

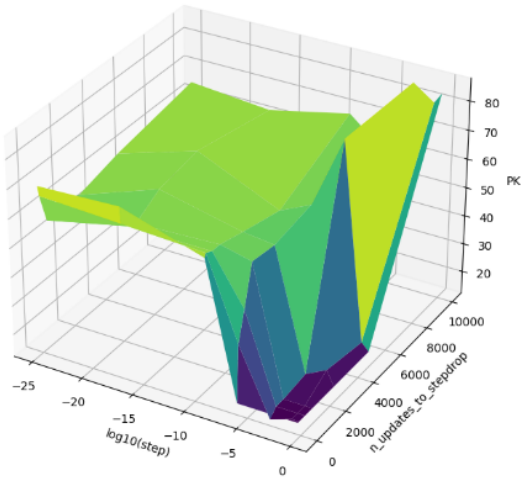


Rysunek 15: Przedstawienie wyników eksperymentu pierwszego dla LVQ2.1, liczba epok - 500

Optymalne parametry:

- Poprawność klasyfikacji: 78.75%
- step: 1e-9
- n_updates_to_stepdrop: 10000

Liczba epok = 100

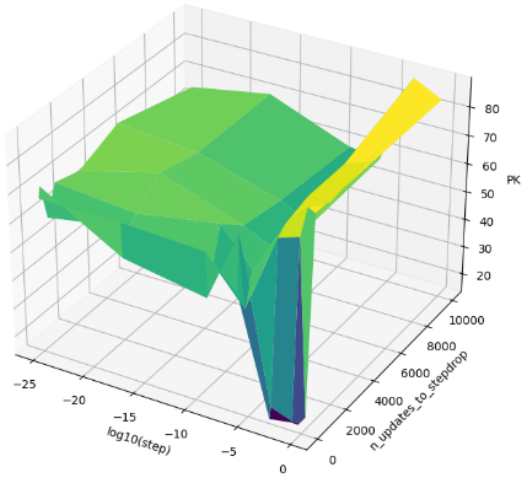


Rysunek 16: Przedstawienie wyników eksperymentu pierwszego dla LVQ2.1, liczba epok - 100

Optymalne parametry:

- Poprawność klasyfikacji: 86%
- step: 1e-3
- n_updates_to_stepdrop: 5000

Liczba epok = 10



Rysunek 17: Przedstawienie wyników eksperymentu pierwszego dla LVQ2.1, liczba epok - 10

Optymalne parametry:

- Poprawność klasyfikacji: 88.75%
- step: 1e-3
- n_updates_to_stepdrop: 10000

Podsumowanie najlepszych parametrów dla LVQ2.1

epoch	step	n_updates_to_stepdrop	Poprawność klasyfikacji
1000	1e-9	5000	73.75%
500	1e-9	10000	78.75%
100	1e-3	5000	86%
10	1e-3	10000	88.75%

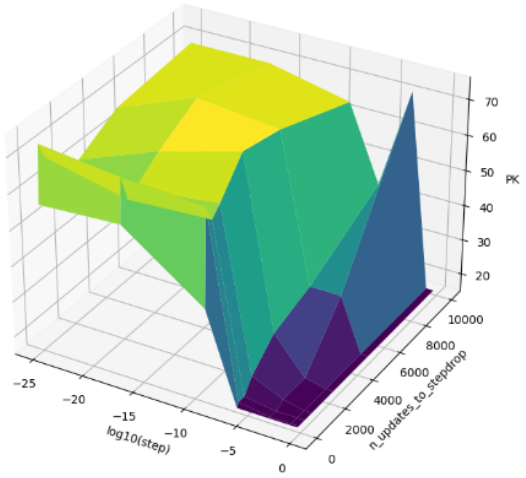
Tabela 3: Zestawienie najlepszych wyników dla algorytmu LVQ2.1

Dla LVQ2.1 poprawność klasyfikacji zmienia się znacząco wraz ze zmianą liczby epok. Najlepsze wyniki osiągnięto przy najmniejszej liczbie epok 10, gdzie uzyskano poprawność klasyfikacji na poziomie 88.75%.

Wartości parametrów step i n_updates_to_stepsrop również mają istotny wpływ na wyniki, zarówno dla 100, jak i 10 epok, optymalne wartości step wynoszą 1e-3, co sugeruje, że większe wartości kroku przyczyniają się do uzyskania lepszych wyników. Natomiast wartości parametru n_updates_to_stepdrop są zróżnicowane, co wskazuje na potrzebę dostosowania tej wartości w zależności od liczby epok.

Algorytm LVQ3

Liczba epok = 1000

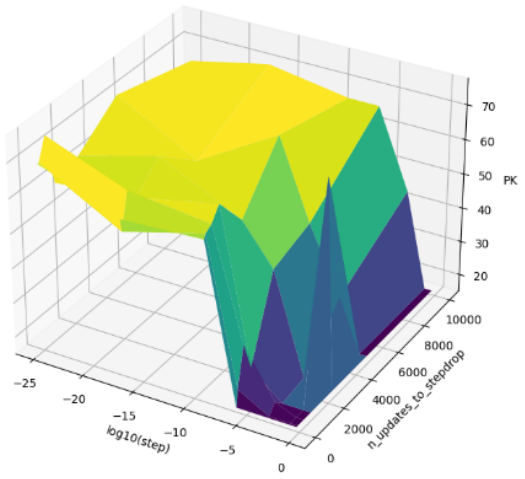


Rysunek 18: Przedstawienie wyników eksperymentu pierwszego dla LVQ3, liczba epok - 1000

Optymalne parametry:

- Poprawność klasyfikacji: 75%
- step: 1e-9
- n_updates_to_stepdrop: 2500

Liczba epok = 500

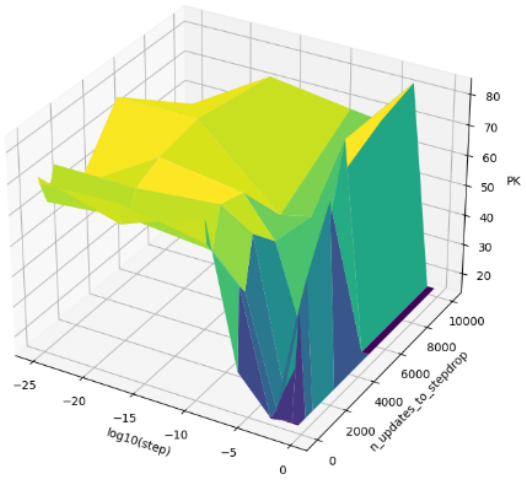


Rysunek 19: Przedstawienie wyników eksperymentu pierwszego dla LVQ3, liczba epok - 500

Optymalne parametry:

- Poprawność klasyfikacji: 76.25%
- step: 0.1
- n_updates_to_stepdrop: 2500

Liczba epok = 100

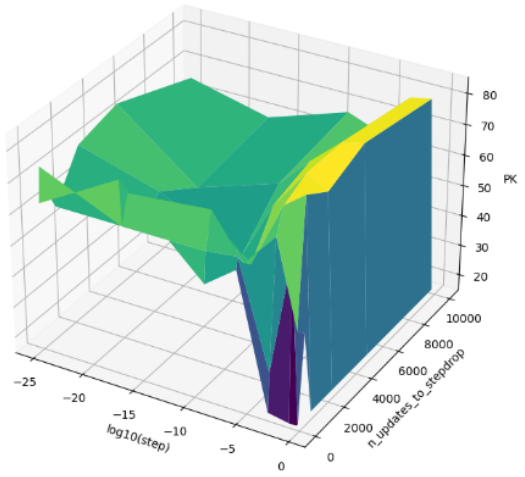


Rysunek 20: Przedstawienie wyników eksperymentu pierwszego dla LVQ3, liczba epok - 100

Optymalne parametry:

- Poprawność klasyfikacji: 83.75%
- step: 1e-3
- n_updates_to_stepdrop: 5000

Liczba epok = 10



Rysunek 21: Przedstawienie wyników eksperymentu pierwszego dla LVQ3, liczba epok - 10

Optymalne parametry:

- Poprawność klasyfikacji: 83.75%
- step: 0.1
- n_updates_to_stepdrop: 1000

Podsumowanie najlepszych parametrów dla LVQ3

epoch	step	n_updates_to_stepdrop	Poprawność klasyfikacji
1000	1e-9	2500	75%
500	0.1	2500	76.25%
100	1e-3	5000	83.75%
10	0.1	1000	83.75%

Tabela 4: Zestawienie najlepszych wyników dla algorytmu LVQ3

Analizując wyniki eksperymentów dla algorytmu LVQ3, można zauważyć, że poprawność klasyfikacji różni się w zależności od liczby epok oraz ustawień parametrów step i n_updates_to_stepdrop. Przy mniejszej liczbie epok (100 i 10) uzyskano lepsze wyniki, osiągając poprawność klasyfikacji na poziomie 83.75%.

Optymalne wartości kroku również różnią się w zależności od liczby epok. Przy większej liczbie epok (1000) krok wynosił 1e-9, co wskazuje na bardzo małe i precyzyjne aktualizacje wag. Dla 500 epok krok wynosił 0.1, a dla 100 i 10 epok optymalne wartości kroku były odpowiednio 1e-3 i 0.1. Te większe wartości kroku przy mniejszej liczbie epok sugerują, że szybsze dostosowanie modelu do danych może być korzystne.

Parametr n_updates_to_stepdrop pokazuje, że przy większej liczbie epok (1000 i 500) optymalne wartości wynosiły 2500, co może sugerować potrzebę częstszych dostosowań kroku uczenia. Natomiast dla mniejszej liczby epok (100 i 10) wartości te były odpowiednio 5000 i 1000, co pozwala na bardziej stopniowe dostosowanie kroku uczenia się przy krótszym treningu.

5.2 Eksperyment drugi

Drugi eksperyment miał na celu zbadanie zależności między minstep, a poprawnością klasyfikacji. Do tego zbadania użyłam optymalne wartości z eksperymentu pierwszego,ale tylko te, których różnica poprawności klasyfikacji najbarzdziej uległa zmianiam.

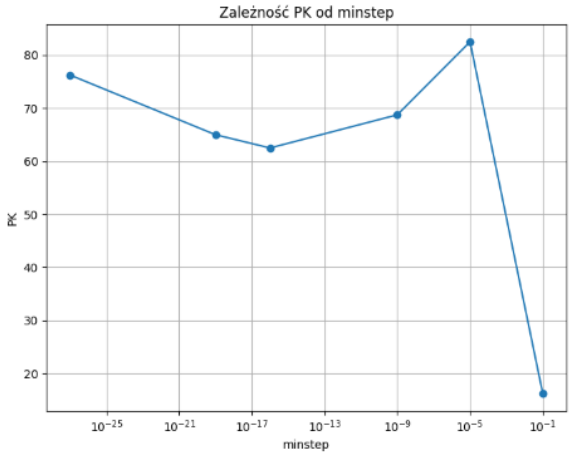
Algorytm	**step**	**n_updates_to_stepdrop**	**epoch**
LVQ1	1e-17	500	500
LVQ2	0.1	100	1000
LVQ2.1	1e-3	10000	10
LVQ3	1e-3	5000	100

Tabela 5: Dane użyte do badania parametru minstep

Badane wartości minstep wynoszą : 0.1, 1e-5, 1e-9, 1e-16, 1e-19, 1e-27

Algorytm LVQ1

Liczba epok: 500



Rysunek 22: Przedstawienie wyników eksperymentu drugiego dla LVQ1, liczba epok - 500

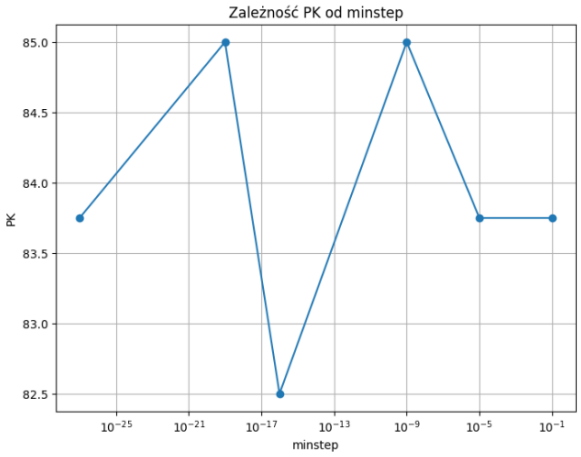
Optymalny parametr minstep: 1e-5

Poprawność klasyfikacji: 82.5%

Algorytm LVQ1 osiąga najwyższą poprawność klasyfikacji 82.5% dla wartości minstep równej 1e-5. Oznacza to, że po dodaniu tego parametra dla najlepszych wartości poprzedzniego elementu poprawność kwalifikacji wzrosła z 73.75%. Dla wartości mniejszych niż 1e-5 poprawność klasyfikacji zaczyna się pogarszać.

Algorytm LVQ2

Liczba epok: 1000



Rysunek 23: Przedstawienie wyników eksperymentu drugiego dla LVQ2, liczba epok - 1000

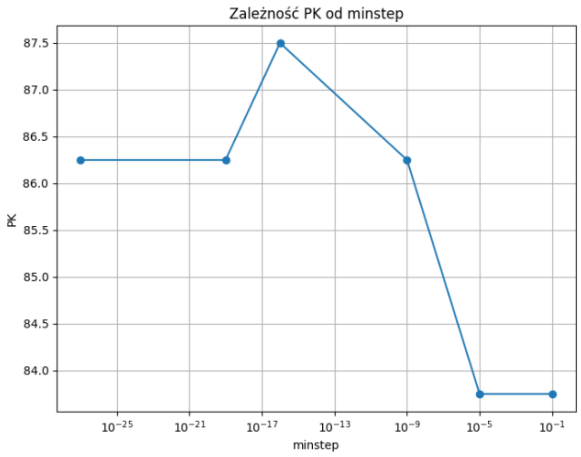
Optymalny parametr minstep: 1e-9

Poprawność klasyfikacji: 85%

Wyniki dla algorytmu LVQ2 pokazują, że parametr minstep nie ma dużego wpływu na poprawność klasyfikacji. Wszystkie testy dla różnych wartości zawierały się w rzedziale od 82% do 85%, co nadal pokazuje stabilność danego algorytmu.

Algorytm LVQ2.1

Liczba epok: 10



Rysunek 24: Przedstawienie wyników eksperymentu drugiego dla LVQ2.1, liczba epok - 10

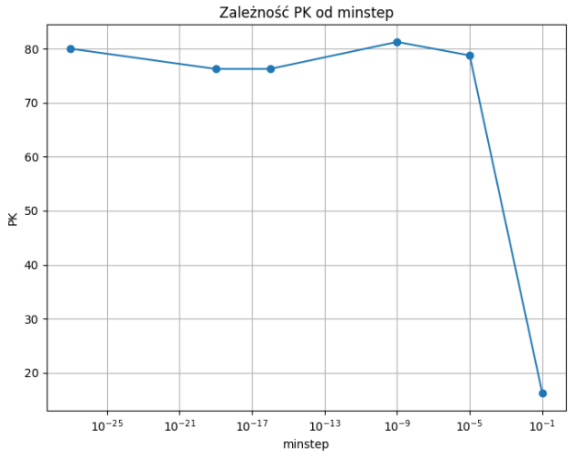
Optymalny parametr minstep: 1e-16

Poprawność klasyfikacji: 87.5%

Patrząc na wyniki lgorytmu LVQ2.1 możemy zobaczyć,że parametr minstep ma wpływ na poprawność klasyfikacji, ale nie jest on zbyt duży.Najwyższą poprawność klasyfikacji osiągnięto na poziomie 87.5% dla minstep 1e-16. Ale minstep nie polepszył poprawność z eksperymentu pierwszego, a wręcz była ona niższa(88.75% w pierwszym eksperymencie).

Algorytm LVQ3

Liczba epok: 100



Rysunek 25: Przedstawienie wyników eksperymentu drugiego dla LVQ3, liczba epok - 100

Optymalny parametr minstep: 1e-9

Poprawność klasyfikacji: 81.25%

Możemy zauważyć, że parametr minstep ma wpływ na poprawność klasyfikacji w algorytmie LVQ3. Najwyższą poprawność klasyfikacji osiągnięto na poziomie 81.25% dla minstep 1e-9. Dla wartości mniejszych niż 1e-9 poprawność klasyfikacji pozostaje praktycznie niezmieniona, jednak dla wartości wyższych niż 1e-9 znacząco spada.

Podsumowanie wyników eksperymentu

Algorytm	**Początkowa poprawność klasyfikacji**	**Poprawność klasyfikacji z minstep**	**Liczba epok**	**Optymalna wartość minstep**
LVQ1	73.75%	82.5%	500	1e-5
LVQ2	83.75%	85%	1000	1e-9
LVQ2.1	88.75%	87.5%	10	1e-16
LVQ3	83.75%	81.25%	100	1e-9

Tabela 6: Przedstawienie wyników dla eksperymentu drugiego

5.3 Eksperyment trzeci

Eksperyment trzeci polegał na zbadaniu długości czasu, jakiej potrzebował algorytm LVQ do treningu, wzależności do liczby epok.

Dane zostały zarejestrowane przy wykonaniu pierwszego algorytmu. Czas został podany w sekundach.

Algorytm	**Liczba epok**			
	10	**100**	**500**	**1000**
LVQ1	46	622	2464	4784
LVQ2	63	484	2649	4440
LVQ2.1	63	602	2583	6206
LVQ3	64	558	2999	5980

Tabela 7: Przedstawienie wyników dla eksperymentu trzeciego

- Algorytm LVQ1 charakteryzuje się stosunkowo stałym wzrostem czasu treningu wraz z zwiększaniem złożoności zadania. Jest to spowodowane prostotą samego algorytmu, co sprawia, że jego czas treningu rośnie proporcjonalnie do liczby epok. Jest najszybszy przy niskiej liczbie epok, ale staje się coraz mniej efektywny w miarę wzrostu liczby epok.
- Algorytm LVQ2 wykazuje najbardziej zrównoważony wzrost czasu treningu w porównaniu do innych wariantów. Jest trochę wolniejszy niż LVQ1 przy niskiej liczbie epok, ale szybkość treningu utrzymuje się na względnie stałym poziomie przy większej złożoności zadania. Jest efektywny w przypadku średniej liczby epok.
- Algorytm LVQ2.1 wykazuje podobne trendy co LVQ2, ale czasem treningu jest nieco wydłużony. Jest to spowodowane wprowadzeniem pewnych modyfikacji w algorytmie w celu poprawy jego skuteczności, co jednak może prowadzić do nieznacznego wzrostu czasu treningu.
- Algorytm LVQ3 charakteryzuje się najdłuższym czasem treningu ze wszystkich wariantów. Pomimo że może osiągnąć dobre wyniki, jego wydajność maleje w porównaniu z innymi algorytmami przy większej liczbie epok.

6 Podsumowanie i wnioski

Celem projektu było stworzenie sieci neuronowej Learning Vector Quantization (LVQ), która mógłaby być wykorzystana do prognozowania przeżywalności osób cierpiących na zapalenie wątroby. Algorytm został zaimplementowany w języku Python, korzystając z dokumentacji biblioteki neupy oraz częściowo gotowej implementacji dostępnej na stronie materialy.prz-rzeszow.pl.

W ramach projektu przeprowadzono również rozległe badania mające na celu optymalizację algorytmu poprzez eksplorację parametrów: step, minstep,n_updates_to_stepdrop oraz wizualizację wyników za pomocą dwuwymiarowych i trójwymiarowych grafów.

W pierwszym eksperymencie przetestowano cztery wersje algorytmu Learning Vector Quantization (LVQ) analizując wpływ różnych parametrów: liczba epok(epoch), krok uczenia(step) i liczba aktualizacji do zmniejszenia kroku(n_updates_to_stepdrop) na poprawność klasyfikacji.

Celem było znalezienie optymalnych ustawień dla każdej wersji algorytmu, aby maksymalizować ich wydajność. Algorytm LVQ2.1 osiągnął najwyższą poprawność klasyfikacji (88.75%), co wskazuje na jego wyższość nad pozostałymi wersjami LVQ w badanym zestawie danych.

Krok uczenia 1e-3 okazał się istotny dla osiągnięcia wysokiej skuteczności klasyfikacji we wszystkich testowanych wersjach algorytmu. Wskazuje to na kluczową rolę tego parametru w procesie uczenia.

Liczba epok i liczba aktualizacji do zmniejszenia kroku mają różny wpływ na skuteczność poszczególnych wersji algorytmu. LVQ2.1 i LVQ2 osiągnęły wysoką poprawność przy mniejszej liczbie epok (10), podczas gdy LVQ1 wymagał większej liczby epok (100) do osiągnięcia podobnych wyników. LVQ3 wykazał tendencję do osiągania poprawności klasyfikacji w podobnym zakresie co inne wersje algorytmu LVQ. Jednakże, nie osiągnął wyższej skuteczności klasyfikacji w porównaniu z innymi wariantami.Różnice te sugerują, że różne wersje algorytmu mogą mieć różne optymalne ustawienia parametrów, które są specyficzne dla struktury i działania każdego z nich. Najbadziej stabilnym algorytmem okazał się LVQ2.

W drugim eksperymencie przetestowano wpływ parametru minstep na skuteczność klasyfikacji dla każdej z czterech wersji algorytmu LVQ. Dla LVQ1 optymalna wartość minstep (1e-5) przyczyniła się do wzrostu poprawności klasyfikacji z 73.75% do 82.5%. W przypadku LVQ2, stabilność algorytmu została potwierdzona, osiągając poprawność między 82% a 85%. Optymalna wartość minstep dla LVQ2 wyniosła 1e-9, przynosząc poprawę do 85%.

Jednakże, dla LVQ2.1 i LVQ3, zmiana parametru minstep nie przyniosła oczekiwanych efektów. Dla LVQ2.1, optymalna wartość minstep (1e-16) spowodowała poprawę do 87.5%, jednak wynik ten był niższy niż poprawność uzyskana w pierwotnym eksperymencie (88.75%). Podobnie, dla LVQ3, optymalna wartość minstep (1e-9) doprowadziła do poprawy do 81.25%, ale wynik ten był niższy niż w eksperymencie pierwotnym (83.75%).

Wnioskiem z drugiego eksperymentu jest to, że zmiana parametru minstep może mieć różnorodne skutki dla różnych wersji algorytmów LVQ.

W trzecim eksperymencie porównano czasy treningu dla różnych wariantów algorytmu LVQ przy różnych liczbach epok. Algorytm LVQ1 charakteryzował się stałym, proporcjonalnym wzrostem czasu treningu wraz z liczbą epok, podczas gdy LVQ2 wykazywał stabilny wzrost czasu treningu. LVQ2.1 i LVQ3 potrzebowały więcej czasu na trening niż pozostałe warianty, przy czym LVQ3 wymagał najdłuższego czasu treningu ze wszystkich.

7 Załączniki

Różnice w kodzie, który realizuje algorytm LVQ2

```
lvqnet = algorithms.LVQ2(
  epsilon = 0.3, # Podwójna aktualizacja wag jest wykonana kiedy różnica między
    dwoma prototypami wynosi 0.3 lub mniej
  n_inputs = x_train.shape[1], # Liczba jednostek wejściowych
  n_classes = np.unique(y_train).shape[0], # Liczba klas w zestawie danych
  step = step_vec[steps], # Współczynnik uczenia
  n_updates_to_stepdrop = n_updates_to_stepdrop_vec[n_updates_to_stepdrop_], #
   Liczba aktualizacji zmniejszania kroku
  minstep = temp_minstep # Testowy 'minstep'
  )

lvqnet = algorithms.LVQ2(
  epsilon = 0.3, # Podwójna aktualizacja wag jest wykonana kiedy różnica między
    dwoma prototypami wynosi 0.3 lub mniej
  n_inputs = x_train.shape[1], # Liczba jednostek wejściowych
  n_classes = np.unique(y_train).shape[0], # Liczba klas w zestawie danych
  step = step_vec[steps], # Współczynnik uczenia
  n_updates_to_stepdrop = n_updates_to_stepdrop_vec[n_updates_to_stepdrop_], #
   Liczba aktualizacji zmniejszania kroku
  minstep = temp_minstep # Testowy 'minstep'
)
```

Listing 3: Rożnica w implementacji algorytmu LVQ1 a LVQ2

Różnice w kodzie, który realizuje algorytm LVQ2.1

```
lvqnet = algorithms.LVQ21(
  epsilon = 0.3, # Podwójna aktualizacja wag jest wykonana kiedy różnica między
    dwoma prototypami wynosi 0.3 lub mniej
  n_inputs = x_train.shape[1], # Liczba jednostek wejściowych
  n_classes = np.unique(y_train).shape[0], # Liczba klas w zestawie danych
  step = step_vec[steps], # Współczynnik uczenia
  n_updates_to_stepdrop = n_updates_to_stepdrop_vec[n_updates_to_stepdrop_], #
   Liczba aktualizacji zmniejszania kroku
  minstep = temp_minstep # Testowy 'minstep'
)

lvqnet = algorithms.LVQ21(
  epsilon = 0.3, # Podwójna aktualizacja wag jest wykonana kiedy różnica między
    dwoma prototypami wynosi 0.3 lub mniej
  n_inputs = x_train.shape[1], # Liczba jednostek wejściowych
  n_classes = np.unique(y_train).shape[0], # Liczba klas w zestawie danych
  step = step_vec[steps], # Współczynnik uczenia
  n_updates_to_stepdrop = n_updates_to_stepdrop_vec[n_updates_to_stepdrop_], #
   Liczba aktualizacji zmniejszania kroku
  minstep = temp_minstep # Testowy 'minstep'
)
```

Listing 4: Rożnica w implementacji algorytmu LVQ1 a LVQ2.1

Różnice w kodzie, który realizuje algorytm LVQ3

```
lvqnet = algorithms.LVQ3(
  epsilon = 0.3, # Podwójna aktualizacja wag jest wykonana kiedy różnica między
    dwoma prototypami wynosi 0.3 lub mniej
  n_inputs = x_train.shape[1], # Liczba jednostek wejściowych
  n_classes = np.unique(y_train).shape[0], # Liczba klas w zestawie danych
  step = step_vec[steps], # Współczynnik uczenia
  n_updates_to_stepdrop = n_updates_to_stepdrop_vec[n_updates_to_stepdrop_], #
   Liczba aktualizacji zmniejszania kroku
  minstep = temp_minstep # Testowy 'minstep'
)

lvqnet = algorithms.LVQ3(
  epsilon = 0.3, # Podwójna aktualizacja wag jest wykonana kiedy różnica między
    dwoma prototypami wynosi 0.3 lub mniej
  n_inputs = x_train.shape[1], # Liczba jednostek wejściowych
  n_classes = np.unique(y_train).shape[0], # Liczba klas w zestawie danych
  step = step_vec[steps], # Współczynnik uczenia
  n_updates_to_stepdrop = n_updates_to_stepdrop_vec[n_updates_to_stepdrop_], #
   Liczba aktualizacji zmniejszania kroku
  minstep = temp_minstep # Testowy 'minstep'
)
```

Listing 5: Rożnica w implementacji algorytmu LVQ1 a LVQ3

Bibliografia

[1] https://archive.ics.uci.edu/dataset/46/hepatitis

[2] Zajdel R., SI_P2_Procedura przygotowania danych.pdf

http://materialy.prz-rzeszow.pl/pracownik/pliki/34/SI_P2__Procedura%20przygotowania%20danych.pdf

[3] Zajdel R.,Przykładowe skrypty 3.pdf

http://materialy.prz-rzeszow.pl/pracownik/pliki/34/Przyk%C5%82adowe%20skrypty%203.pdf

[4] https://studfile.net/preview/5083084/page:7/

[5] Rifat Aşlıyan, Examining Variants of Learning Vector Quantizations According to Normalization and Initialization of Vector Positions

https://dergipark.org.tr/en/download/article-file/2844839

[6] https://studfile.net/preview/5083085/page:14/

[7] neupy.algorithms.LVQ

http://neupy.com/modules/generated/neupy.algorithms.LVQ.html

Spis rysunków

Spis tabel

Listings