Wydział Inżynierii Mechanicznej i Robotyki

Katedra Robotyki i Mechatroniki

Podstawy Sztucznej Inteligencji I Uczenia Głębokiego

Inżynieria Mechatroniczna

Instrukcja 3:

Liniowa i nieliniowa klasyfikacja

Nauczysz się: Jak napisać swój własny klasyfikator od podstaw – poczynając od klasyfikacji liniowej i rozszerzając ją do klasyfikacji nieliniowej. Przetestujemy też skuteczność różnych algorytmów opracowanych w ramach pierwszej instrukcji w rozwiązywaniu tego wielowymiarowego problemu optymalizacyjnego.

Dodatkowe materiały:

- Wykład nr 1
- Wykład nr 2

Ta instrukcja została przetłumaczona przez ChatGPT na podstawie angielskiego oryginału – materiałów z przedmiotu *Basic of Artificial Intelligence and Deep Learning*

Koordynator przedmiotu:
Krzysztof Holak, holak@agh.edu.pl

Autor instrukcji:
Ziemowit Dworakowski, zdw@agh.edu.pl

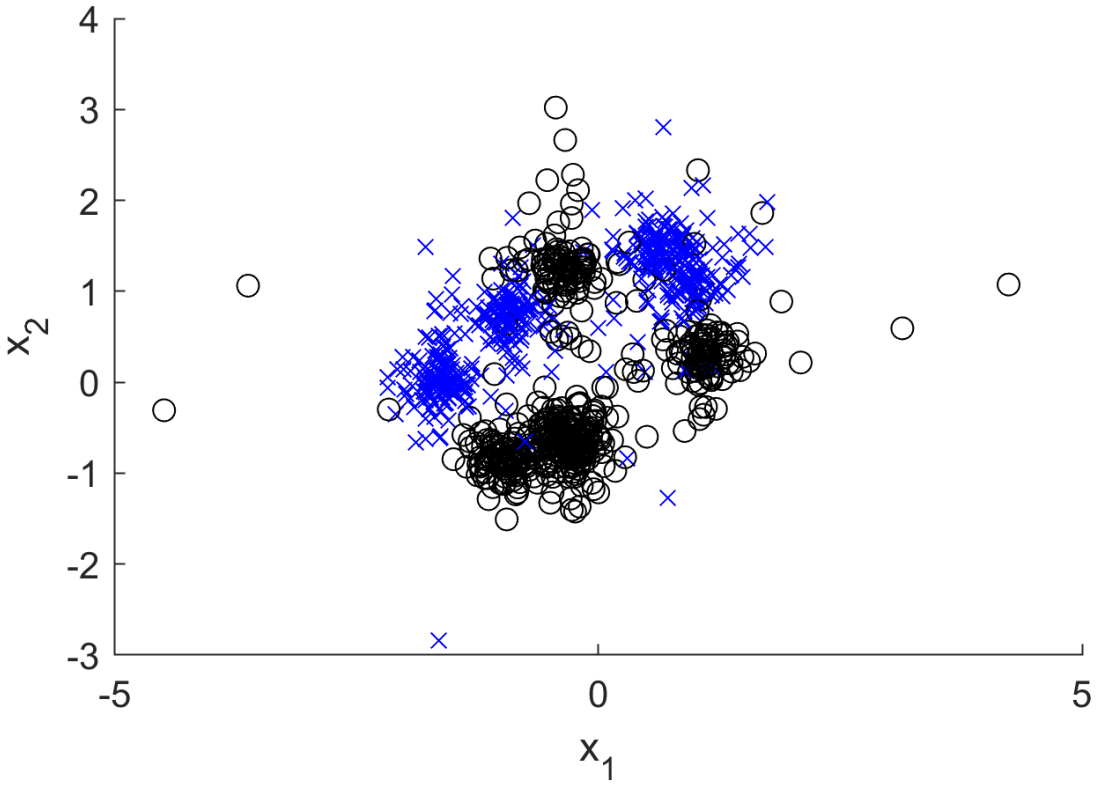
Klasyfikacja liniowa

Podobnie jak wcześniej, potrzebujemy zbioru danych. Tym razem wygenerujemy go z predefiniowanego rozkładu klastrów. Użyję do tego następującego kodu:

```
load Clusters_10

Samples = 1000;          % How many data samples there are?
DataDivision = 0.5;      % How many data samples fall into which class?
v = 2;                   % v parameter of T Student's distribution

% Definition of data
for k = 1:Samples
   if(rand()>DataDivision)
       DATA(1,k) = 1;
       Ind = randi(Clusters.ClustersA);
       DATA(2,k) = Clusters.ACoordinates(1,Ind)+random('T',v)*0.15;
       DATA(3,k) = Clusters.ACoordinates(2,Ind)+random('T',v)*0.15;
   else
       DATA(1,k) = 0;
       Ind = randi(Clusters.ClustersB);
       DATA(2,k) = Clusters.BCoordinates(1,Ind)+random('T',v)*0.15;
       DATA(3,k) = Clusters.BCoordinates(2,Ind)+random('T',v)*0.15;
   end
end

for k = 1:Samples
    if(DATA(1,k) == 1)
        plot(DATA(2,k),DATA(3,k),'ok'); hold on
    else
        plot(DATA(2,k),DATA(3,k),'xb'); hold on
    end
end
xlabel('x');
ylabel('y');
ylim([-3 4])


save DATA DATA
```

Zbiory klastrów znajdują się w naszym zajęciowym zbiorze funkcji i danych (StudentToolbox2025). Zwróć uwagę, że wartość *DataDivision* określa podział danych na klasy (możemy mieć więcej przykładów w jednej klasie niż w drugiej), natomiast wartość *v* określa, jak ciężkie są ogony rozkładu danych – im niższa wartość tego parametru, tym cięższe ogony rozkładu (używamy tutaj rozkładu T-Studenta).

Te wartości parametrów pozwoliły na uzyskanie zbioru danych widocznego na rys. 1. Warto zauważyć, że użycie rozkładu T-Studenta zamiast rozkładu Gaussa umożliwiło wygenerowanie kilku wartości odstających, a także naturalne połączenie klastrów, jednocześnie zachowując odrębne centra klastrów. Dzięki temu otrzymaliśmy ciekawy dwuwymiarowy problem klasyfikacyjny.



Rys 1 – Problem klasyfikacyjny stworzony za pomocą rozkładu klastrów danego w pliku Clusters_10. Zwróć uwagę na fakt, że generując dane do klastrów za każdym razem otrzymasz nieco inny rozkład punktów, ale same centra klastrów znajdować się będą w tych samych miejscach.

Dane uczące, walidacyjne i testowe

Nasze dane powinny być teraz przygotowane do trenowania i testowania klasyfikatorów. Podzielimy je losowo na trzy podzbiory: treningowy (50% danych), walidacyjny (25% danych) i testowy (25% danych), co w naszym przypadku oznacza odpowiednio 500, 250 i 250 próbek. Do tego celu możemy użyć następującego kodu:

```
Indices = randperm(length(DATA));
DATA_permutated = DATA(:,Indices)

TR_number = ceil(length(DATA)*0.5);
VA_number = ceil(length(DATA)*0.25);
TE_number = ceil(length(DATA)*0.25);

TR_DATA = DATA_permutated(:,1:TR_number);
VA_DATA = DATA_permutated(:,TR_number+1:TR_number+VA_number);
TE_DATA = DATA_permutated(:,TR_number+VA_number+1:end);

save TR_DATA TR_DATA
save VA_DATA VA_DATA
save TE_DATA TE_DATA
```

Zadanie 3.1: Używając swojej **indywidualnej** struktury Clusters, wygeneruj zbiór danych zawierający 1000 próbek, podzielonych równo między dwie klasy, z wartością v=2. Zapisz wygenerowany zbiór danych na dysku, aby można go było używać do trenowania i testowania klasyfikatorów. Zapisz zarówno oryginalny zbiór danych (DATA), jak i jego podzielone podzbiory. Zapisz także skrypt użyty do generowania danych, aby później można było go wykorzystać do tworzenia innych zbiorów danych.

Projekt prostego klasyfikatora liniowego

Generując dane upewniliśmy się, że nasz zbiór danych nie jest liniowo separowalny. Niemniej jednak spróbujemy podzielić te dane za pomocą prostej linii. Aby zaprojektować klasyfikator, zadamy proste pytanie dla każdego punktu danych: „Czy ten punkt znajduje się powyżej czy poniżej wcześniej zdefiniowanej linii?"

Sformułujmy równanie dla tej linii:

W1 * x1 + W2 * x2 + b > 0 (1)

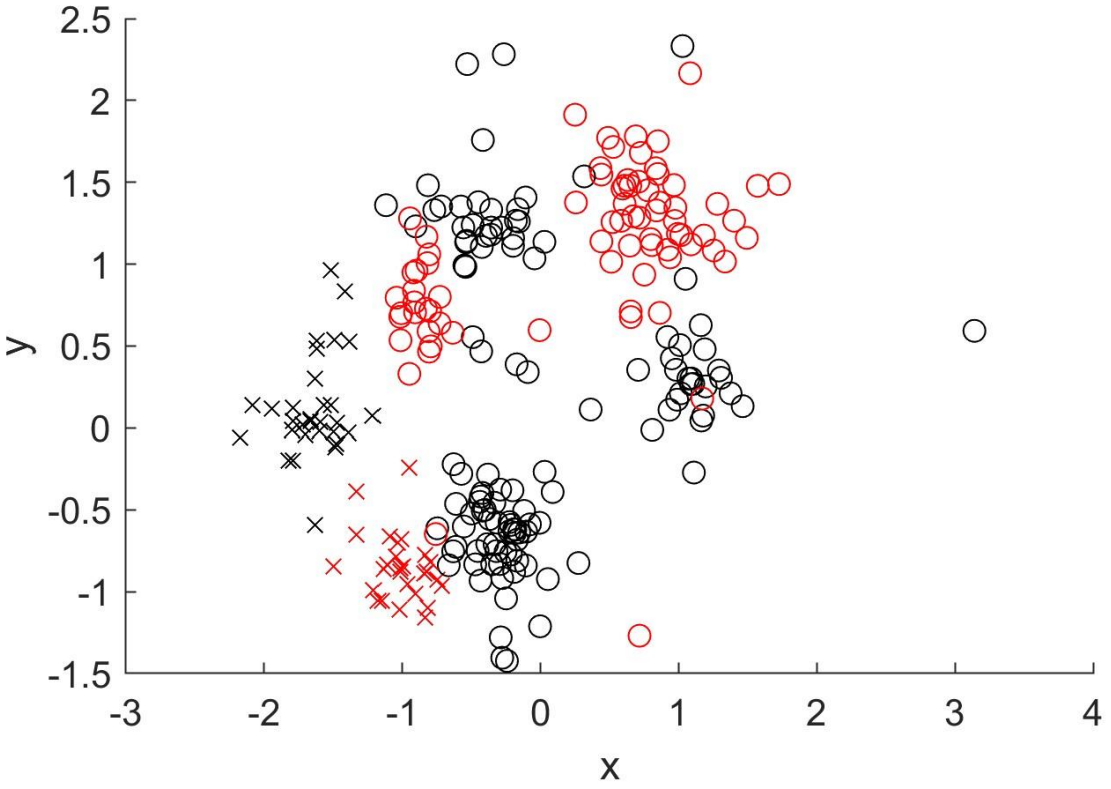
I teraz będziemy poszukiwać takich parametrów W1, W2 i B by zmaksymalizować skuteczność klasyfiacji Nasz klasyfikator powinien wyglądać tak:

```
function [ClassLabel] = InitialClassifier(x,y,Parameters)
    if(Parameters.W1*x + Parameters.W2*y + Parameters.B > 0)
        ClassLabel = 1;
    else
        ClassLabel = 0;
    end
end
```

Taki klasyfikator może być zapisany jako funkcja i wykorzystany do sklasyfikowania podzbioru danych w następujący sposób:

```
load VA_DATA

Parameters.W1 = 1;
Parameters.W2 = 0.3;
Parameters.B = 1;

ErrorsA = 0;
ErrorsB = 0;

for k = 1:length(VA_DATA)
   if(InitialClassifier(VA_DATA(2,k),VA_DATA(3,k),Parameters) == 1)
      % Data point classified as A
      if(VA_DATA(1,k) == 1)
          % Data point classified correctly!
          plot(VA_DATA(2,k),VA_DATA(3,k),'ok'); hold on
      else
          plot(VA_DATA(2,k),VA_DATA(3,k),'xr') ; hold on
          ErrorsA = ErrorsA + 1;
      end
   else
       % Data point classified as B
       if(VA_DATA(1,k) == 0)
        % Data point classified correctly!
          plot(VA_DATA(2,k),VA_DATA(3,k),'xk'); hold on
       else
          plot(VA_DATA(2,k),VA_DATA(3,k),'or') ; hold on
          ErrorsB = ErrorsB + 1;
       end
   end
end
xlabel('x');
ylabel('y');
ErrorsA
ErrorsB
ErrorsA+ErrorsB
```

Powyższy skrypt wygenerował wyniki przedstawione na rys. 2. Wystąpiło 76 błędów w klasie A oraz 27 błędów w klasie B – wynik nie jest idealny, ale jak dotąd parametry W1,W2, i B zostały wybrane dość losowo.



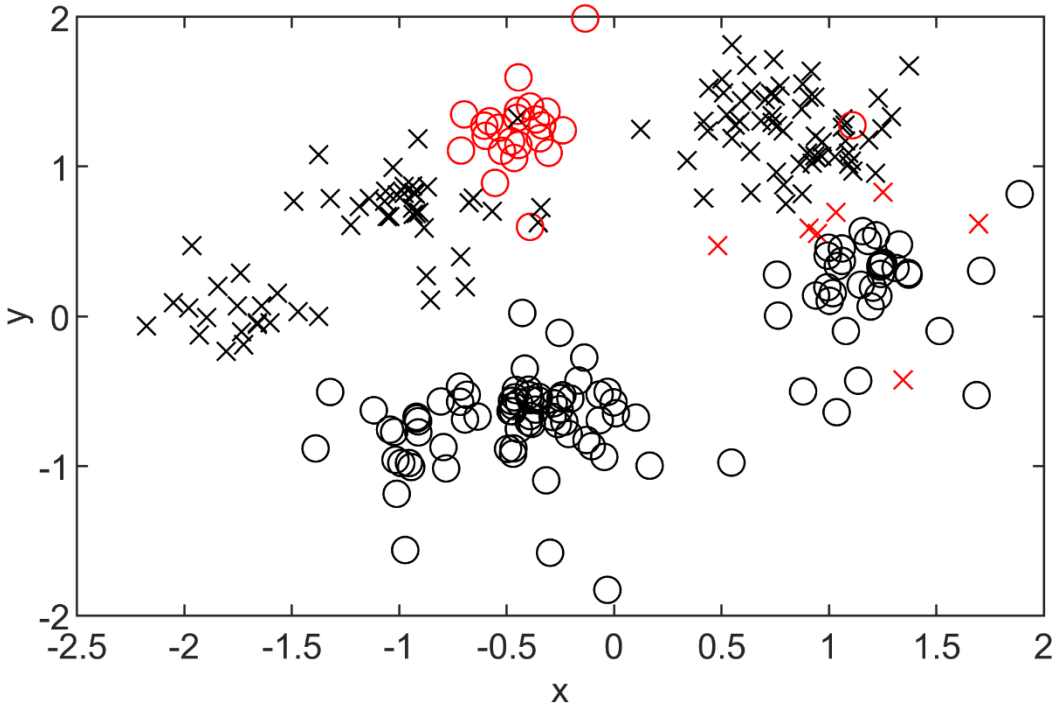
Rys 2 – Wyniki klasyfikacji w dwuwymiarowej przestrzeni cech. Na czerwono zaznaczono punkty przypisane przez klasyfikator do niewłaściwej klasy.

Czy możemy osiągnąć lepszy wynik? Zauważmy, że mamy tu problem optymalizacyjny z trzema parametrami i jednym kryterium. Naszą funkcją celu, którą chcemy zminimalizować, jest suma błędów w obu klasach. Dysponujemy już narzędziami do rozwiązywania tego rodzaju problemów, czyli różnymi algorytmami optymalizacji opracowanymi w ramach pierwszego laboratorium.

Aby wykorzystać wcześniej opracowane rozwiązania (np. *1+1* lub *grid search*), musimy zapisać skrypt klasyfikatora jako funkcję, która:

- Jako weście przyjmuje wartości parametrów (oznaczone w kodzie na zielono).
- Zwraca sumę błędów (oznaczoną na niebiesko).
- Zakomentować linie odpowiedzialne za rysowanie wykresów (oznaczone na szaro).
- Wykorzystuje podzbiór *TR_DATA* zamiast *TR_DATA*, ponieważ teraz trenujemy algorytm, a nie oceniamy jego działanie.

Jeśli wszystko zostanie wykonane poprawnie, algorytm optymalizacji powinien znaleźć wyraźny podział liniowy minimalizujący błędy. Dla clusters_10 otrzymany obraz powinien wyglądać jak na rys. 3



Rys 3 – Rezultat optymalizacji klasyfikatora liniowego dla danych wygenerowanych na podstawie Clusters10. Na zbiorze walidacyjnym widzimy wyraźny, czerwony klaster błędów – to punkty, które są niemożliwe do separacji za pomocą klasyfikatora liniowego.

Zadanie 3.2: Używając skryptu opracowanego w ramach laboratorium 1, zoptymalizuj parametry swojego klasyfikatora liniowego. Wykorzystaj algorytm przeszukiwania siatkowego (Grid Search). Do optymalizacji parametrów użyj danych treningowych (TR_DATA). Po pomyślnej optymalizacji przetestuj klasyfikator na danych walidacyjnych (VA_DATA). Zapisz skrypt, aby mógł zostać sprawdzony przez prowadzącego. Zapisz wynik tej optymalizacji w Tabeli 3.1 na końcu instrukcji.

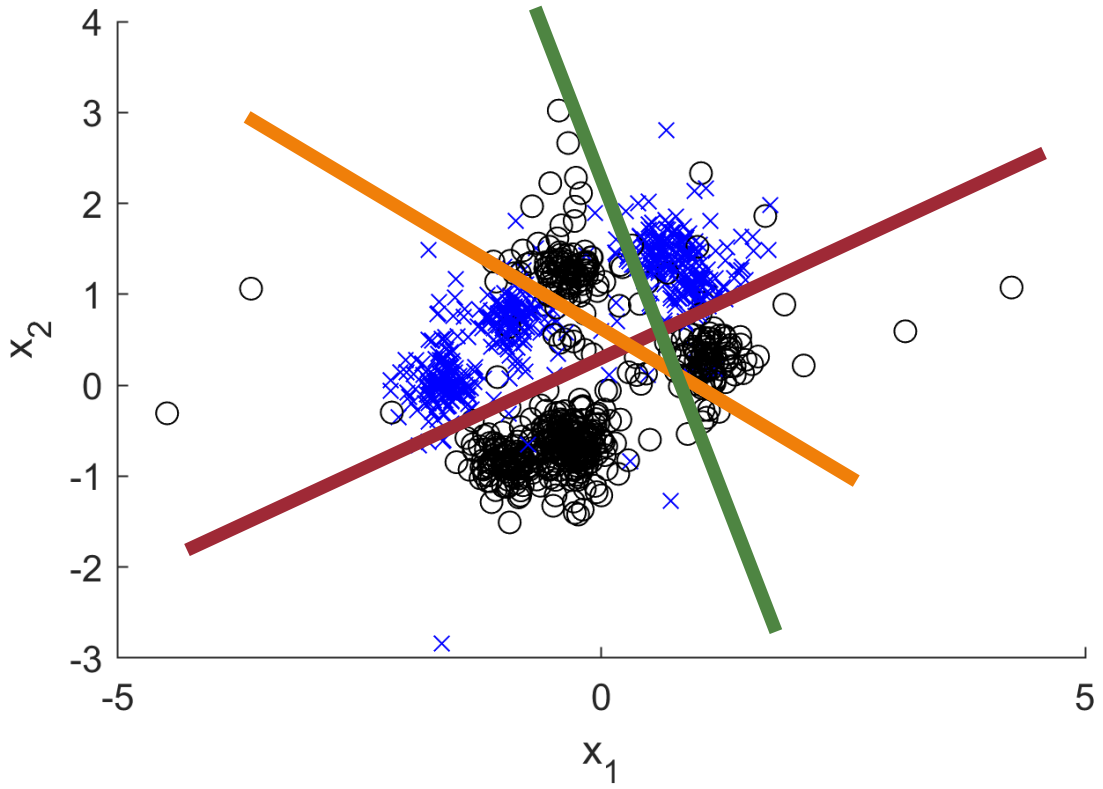
Zwróć uwagę, że masz trzy parametry do optymalizacji, więc Twój skrypt GridSearch będzie wymagał dodatkowej pętli do modyfikacji trzeciego parametru.

Zadanie 3.3: Używając skryptu opracowanego w ramach laboratorium 1, zoptymalizuj parametry swojego klasyfikatora liniowego. Wykorzystaj algorytm 1+1 z adaptacyjnym krokiem. Do optymalizacji parametrów użyj danych treningowych (TR_DATA). Po pomyślnej optymalizacji przetestuj klasyfikator na danych walidacyjnych (VA_DATA). Skonfiguruj algorytm optymalizacji tak, aby konsekwentnie znajdował globalne minimum. Zapisz skrypt, aby mógł zostać sprawdzony przez prowadzącego. Zapisz wynik tej optymalizacji w Tabeli 3.1 na końcu instrukcji.

Modyfikacja pozwalająca na klasyfikację nieliniową

Pamiętasz, że nasze klastry nie były liniowo separowalne? Teraz znajdziemy sposób, aby faktycznie je rozdzielić. Pomysł, który zastosujemy, polega na wykonaniu wielu klasyfikacji przy użyciu różnych linii, a następnie wykorzystaniu tych cząstkowych wyników do wyprowadzenia ostatecznej decyzji o przynależności do klasy.

Spójrz na rys. 4. Możemy na przykład przyjąć, że punkt należy do klasy „niebieskie kółko", jeśli znajduje się poniżej czerwonej linii **lub** powyżej pomarańczowej i jednocześnie poniżej zielonej. Taka reguła pozwoliłaby nam znacząco poprawić dokładność klasyfikacji, pozostawiając jedynie niewielką liczbę błędnie sklasyfikowanych punktów wynikających z nakładania się klastrów i obecności wartości odstających.



Rys 4 – Dodanie dwóch dodatkowych linii pozwala na skuteczną klasyfikację naszego zbioru

Zanim zbudujemy klasyfikator, musimy najpierw zdecydować, jak będzie działał nasz model. Będziemy używać struktury *Parameters*, która przechowa współczynniki dla wszystkich linii. Kolejne współczynniki mogą być kodowane przez kolejne elementy wektorów naszej struktury, jak w poniższym kodzie. Współczynniki definiujące te same linie są oznaczone tym samym kolorem:

```
Parameters.W1 = [ 0.2, 0.3,-0.5];
Parameters.W2 = [ 1.3, 1.1, 2.1];
Parameters.B =  [ 3.0, 0.1,-0.7];
```

Użyjemy naszego wcześniej wykorzystywanego kodu *InitialClassifier* i zmodyfikujemy go by przechodził po wszystkich liniach naszej struktury za pomocą następującego kodu:

```
ClassPresence = 0;
for k = 1:length(Parameters.W1)
   if(Parameters.W1(k)*x + Parameters.W2(k)*y + Parameters.B(k) > 0)
        ClassPresence = ClassPresence + 1;
   else
        ClassPresence = ClassPresence - 1;
   end
end
```

Jeśli zaczniemy od *ClassPresence = 0* i po przejściu w pętli przez wszystkie linie wynik będzie 0 lub większy, klasyfikator powinien zwrócić *ClassLabel = 1*. W przeciwnym razie *ClassLabel* powinien wynosić 0. Zwróć uwagę na zmienną *k* – śledzi ona numer linii, którą aktualnie analizujemy.

Nazwiemy tę zmodyfikowaną funkcję *MultilineClassifier*. Teraz jedyne, co musimy zrobić, to zastosować rozwiązanie optymalizacyjne do faktycznego wytrenowania klasyfikatora. Potrzebujemy wektora parametrów, którego długość odpowiada liczbie optymalizowanych zmiennych (trzy na linię). Wprowadzimy go do naszego optymalizatora w taki sposób:

```
linesUsed = 9;
dimensions = linesUsed * 3;
Range = zeros(dimensions,2)
% Here we state what range we want to draw initial solutions from:
Range = Range + [-10,10]

% And a starting point:
Point = Range(:,1)' + rand(1,dimensions).*(Range(:,2)-Range(:,1))';
```

Zwróć uwagę, że ponieważ zdecydowaliśmy się na reprezentację klasyfikatora opartą na strukturze (aby ułatwić jej zrozumienie), musimy ją zbudować na podstawie używanej w naszej optymalizacji reprezentacji wektorowej. Możemy to zrobić w prosty sposób, używając poniższego kodu:

```
Parameters.W1 = OptimizerOutput(1:linesUsed);
Parameters.W2 = OptimizerOutput(linesUsed +1:2* linesUsed);
Parameters.B  = OptimizerOutput(2*linesUsed +1:end);
```

W tym kodzie *OptimizerOutput* to wektor optymalizowanych współrzędnych – czyli zawartość zmiennej *Point* lub *Result.* Jeśli z jakiegoś powodu chcielibyśmy powyższą czynność odwrócić (budując reprezentację wektorową ze struktury), możemy to zrobić tak:

```
OptimizerInput = [Parameters.W1,Parameters.W2,Parameters.B]
```

Jeśli ciekawi Cię, gdzie w przestrzeni cech znajdują się zoptymalizowane linie, możesz użyć tego fragmentu kodu po narysowaniu sklasyfikowanych punktów. Kod ten narysuje linie w przestrzeni cech zgodnie z zawartością struktury *Parameters*:

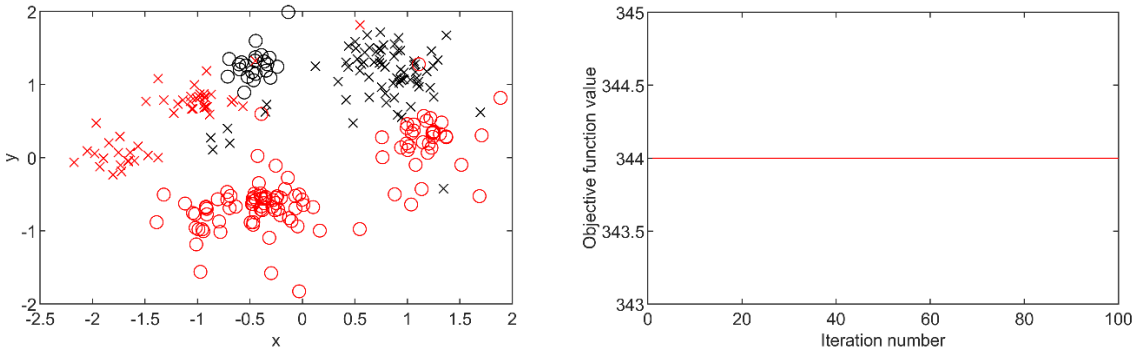
```
X = [-10,10]
Y(:,1) = -(Parameters.W1(:)*X(1) + Parameters.B(:))./Parameters.W2(:);
Y(:,2) = -(Parameters.W1(:)*X(2) + Parameters.B(:))./Parameters.W2(:);
for(p = 1:linesUsed)    line([X],[Y(p,:)]); hold on;  end
```

Zadanie 3.4: Korzystając z Twojego indywidualnego zbioru danych do klasyfikacji:

(a) Zaimplementuj trening klasyfikatora 5-liniowego przy użyciu metody treningowej *1+1*.
(b) Skonfiguruj metaparametry rozwiązania *1+1* tak, aby zwracało powtarzalne wyniki.
(c) Oceń swoje ostateczne rozwiązanie statystycznie (co najmniej 20 prób) i zapisz wyniki w strukturze wygenerowanej w zadaniu 2.11.
(d) Zapisz oszacowania statystyczne swoich prób w Tabeli 3.1 na końcu instrukcji.

Wprowadzenie informacji o gradiencie do zadania klasyfikacji

Jak zapewne zauważyłeś, do optymalizacji nie używaliśmy algorytmu gradientowego. Sprawdźmy, dlaczego. Z czystej ciekawości wróćmy do naszego klasyfikatora jednoliniowego z zadania 3.3, przekażmy naszą funkcję celu do algorytmu gradientowego i zobaczmy, co się stanie (rys. 5):



Rys 5 – Zastosowanie algorytmu gradientowego do optymalizacji funkcji celu z zadania 3.3

Najwyraźniej metoda w swojej obecnej postaci nie potrafi poprawnie wytrenować naszego klasyfikatora… Dlaczego tak się dzieje? Warto zauważyć, że mimo iż mamy problem ciągły (możemy przypisywać wartości zmiennoprzecinkowe do W1, W2 i B), nie jesteśmy jeszcze w stanie wyznaczyć gradientu tej funkcji. Powód jest następujący: jeśli przesuniemy linię separacji i przekroczymy lokalizację jakiegokolwiek punktu danych, wartość funkcji celu nie będzie zmieniać się stopniowo – zamiast tego zostanie zwiększona lub zmniejszona skokowo. Bardzo mała zmiana dowolnego parametru prawdopodobnie nie spowoduje żadnej zmiany wartości funkcji celu.

Moglibyśmy to zaakceptować i po prostu nie używać metod gradientowych, ale potraktujmy to jako wyzwanie i zastanówmy się, co możemy zrobić, aby umożliwić tutaj podejście gradientowe. Na początek potrzebujemy podfunkcji, która nie tylko określi, czy nasza linia klasyfikacyjna „przekroczyła" punkt danych, ale także jak daleko znajduje się linia od klasyfikowanego punktu w danej chwili.

Nazwiemy tę funkcję **funkcją aktywacj**i (*activation function*) i zdefiniujemy ją w następujący sposób:

$$u = \frac{2}{1 + e^{-y}} - 1$$

gdzie:

$$y = w_1 x_1 + w_2 x_2 + b$$

Więc wykonajmy implementację:

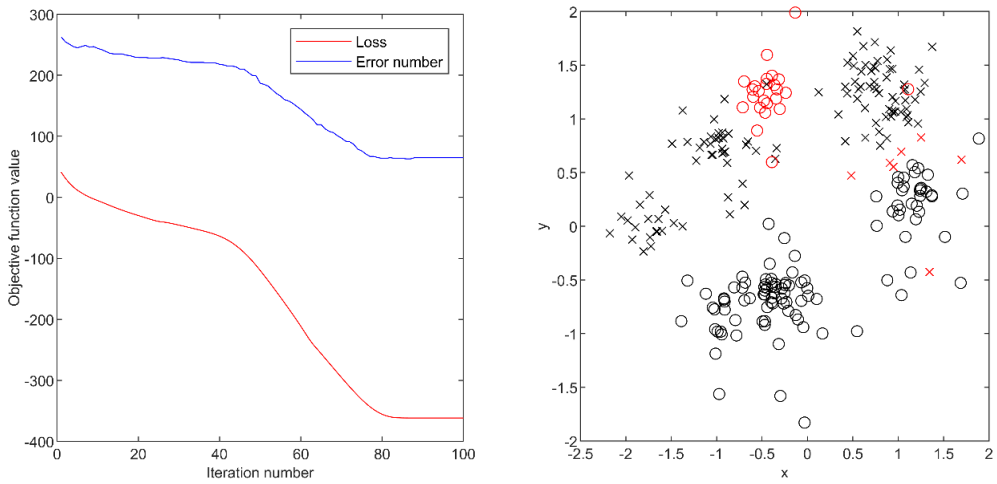
```
function[AF] = ActivationFunction(x,y,Parameters)
   AF = 2/(1+exp(-(Parameters.W1*x + Parameters.W2*y + Parameters.B))) - 1;
end
```

Teraz możemy po prostu podstawić to równanie w miejsce, gdzie wcześniej sprawdzaliśmy, czy punkt znajduje się powyżej czy poniżej linii. Teraz zamiast wartości 0 lub 1, otrzymamy ciągły wynik zależny od odległości punktu od linii – przy czym wartość będzie dążyć do -1 lub +1 dla punktów znajdujących się daleko od linii. Cel pozostaje taki sam jak wcześniej: minimalizacja funkcji celu (*ClassificationLossCumulated*), ale tym razem nie będziemy zliczać błędów, tylko wyniki zwracane przez funkcję aktywacji dla wszystkich punktów.

Każda poprawna klasyfikacja powinna dodawać wartości ujemne do sumy (zmniejszając sumę), a błędna klasyfikacja – wartości dodatnie (powiększając ją). Zwróć uwagę, że możemy to łatwo osiągnąć, zmieniając odpowiedni znak – chcemy, aby wszystkie próbki miały wartość funkcji aktywacji równą „1” dla klasy „A” i -1 dla klasy „B”:

```
function[Loss] = ClassificationLossCumulated(Input)

    load TR_DATA
    Parameters.W1 = Input(1);
    Parameters.W2 = Input(2);
    Parameters.B =  Input(3);
    Loss = 0;

    for k = 1:length(TR_DATA)
     if(TR_DATA(1,k) == 1)    % For class A
       Loss = Loss - ActivationFunction(TR_DATA(2,k),TR_DATA(3,k),Parameters);
     else  % For class B
       Loss = Loss + ActivationFunction(TR_DATA(2,k),TR_DATA(3,k),Parameters);
     end
    end
end
```

Jeśli to zrobimy, zauważymy, że nasz klasyfikator faktycznie uczy się wzorca w danych. Choć czasami może utknąć w minimum lokalnym, krzywa zbieżności powinna wykazywać poprawę przynajmniej przez kilka początkowych iteracji (zob. rys. 6).



Rys 6 – Klasyfikator liniowy dopasowany z zastosowaniem metody gradientowej wykorzystującej funkcję aktywacji

Zadanie 3.5: Dodaj sigmoidalną funkcję aktywacji, aby umożliwić użycie optymalizacji gradientowej w naszym klasyfikatorze. Następnie naucz (na podstawie zbioru TR_DATA) i przetestuj (VA_DATA) ten klasyfikator. Zapisz wyniki w Tabeli 3.1 po wykonaniu trzech prób metody.

Teraz uogólnijmy nasze rozwiązanie, aby umożliwiało klasyfikację wieloliniową. Ponownie potrzebujemy tylko kilku drobnych zmian. Będziemy używać kodu optymalizacji gradientowej z pierwszej instrukcji, wprowadzając zmiany podobne do tych z zadania 3.5.
Zmiany wystąpią w samym klasyfikatorze oraz w funkcji oceniającej oceniająca jakość naszego rozwiązania.
Aby ocenić, czy punkt powinien zostać przypisany do jednej czy drugiej klasy, potrzebujemy następującego klasyfikatora:

```
function [CumulativeActivation] = MultilineClassifierDifferentiable(x,y,Parameters)
   CumulativeActivation = 0;

   for k = 1:length(Parameters.W1)
      AF = 2/(1+exp(-(Parameters.W1(k)*x + Parameters.W2(k)*y + Parameters.B(k)))) - 1;
      CumulativeActivation = CumulativeActivation + AF;
   end
    % In order to cast the solution to the (0,1) range maintaining differentiability:
   CumulativeActivation = 1/(1+exp(-(CumulativeActivation)));
end
```

Kod działa w zasadzie tak samo jak *MultilineClassifier* – jedyną różnicą jest zastosowanie funkcji sigmoidalnej w zakresie (-1,1) (oznaczonej na niebiesko) zamiast prostego sprawdzenia, czy punkt znajduje się powyżej lub poniżej linii. Następnie, ponieważ oczekujemy, że klasyfikator zwróci wartości z przedziału (0,1) (wskazując, czy punkt należy do klasy 0 czy 1), musimy przekształcić wynik na pożądany zakres przy użyciu kolejnej funkcji sigmoidalnej – oznaczonej na żółto.

Teraz właściwa funkcja treningowa, która przechodzi przez wszystkie punkty w zbiorze treningowym, może być bardzo prosta:

```
function[Loss] = TrainTheClassifierGradient(Parameters)

load TR_DATA
Loss = 0;
     for k = 1:length(TR_DATA)
       if(TR_DATA(1,k) == 1)     % For class A
         Loss = Loss + (1 - MultilineClassifierDifferentiable(TR_DATA(2,k),TR_DATA(3,k),Parameters));
       else  % For class B
         Loss = Loss + MultilineClassifierDifferentiable(TR_DATA(2,k),TR_DATA(3,k),Parameters);
       end
    end
end
```

Zwróć uwagę, że zmienna *Loss* zlicza błędy poprzez dodawanie rezultatów dla klasy B (ponieważ powinny mieć wartość 0) oraz dodawanie odwrotności rezultatów dla klasy A (ponieważ te powinny mieć wartość 1). W ten sposób *Loss* powinien wzrastać z każdym błędem i mieć wartość 0 jeśli błędów nie będzie.

Zadanie 3.6: Powtórz zadanie 3.4 dla swojego indywidualnego zbioru danych, tym razem używając metody gradientowej.

(a) Zaimplementuj trening klasyfikatora 5-liniowego przy użyciu metody gradientu.
(b) Skonfiguruj metaparametry algorytmu optymalizacyjnego, aby uzyskać stabilne i powtarzalne wyniki.
(c) Oceń swoje końcowe rozwiązanie statystycznie (co najmniej 20 prób) i zapisz wyniki w strukturze wygenerowanej podczas poprzedniego laboratorium.
(d) Zapisz oszacowania statystyczne swoich prób w Tabeli 3.1 na końcu instrukcji.

Table 3.1: Linear classification

	W1	W2	B		
Grid search				**Suma błędów:**	
1+1				**Suma błędów:**	
Wieloliniowy, 1+1	**-**				
Gradient (1 próba)					
Gradient (2 próba)					
Gradient (3 próba)					
Gradient wieloliniowy	**-**			**Średnia:**	
				Odchylenie std:	

Zadania dodatkowe:

Co dzieje się z trudnością treningu, gdy zwiększamy wymiarowość naszego problemu?

OK, mamy działające rozwiązanie dla nieliniowej klasyfikacji Co jednak się stanie, jeśli skonfigurujemy klasyfikator, aby używał znacznie większej liczby linii? Czy sprawi to, że problem stanie się trudniejszy? Jeśli tak – o ile trudniejszy?

Przetestujmy to i zobaczmy, co się wydarzy. Zwiększymy liczbę linii w klasyfikatorze i porównamy powtarzalność rozwiązań opartych na metodach 1+1 oraz gradientu z wielokrotnym startem. Aby uprościć eksperyment, ustalimy stałą liczbę startów w metodzie wielokrotnego startu (np. 5). Żeby porównanie było uczciwe, przyznamy wszystkim algorytmom tę samą liczbę obliczeń funkcji celu.

Teraz przetestujmy ich wydajność i zobaczmy, co się stanie…

Zadanie 3.7: Przeprowadź statystyczną ocenę klasyfikatorów 5, 9, 13, 17, 21 i 25-liniowych przy użyciu algorytmów 1+1 oraz gradientu z wielokrotnym startem, w najlepszej możliwej konfiguracji. Zapisz wyniki w naszej strukturze danych, uwzględniając oszacowania statystyczne. Na podstawie wyników sformułuj wnioski.

Czy problem wieloliniowej klasyfikacji jest trudny ze względu na minima lokalne czy trudności w eksploracji?

Powinniśmy dokładniej zbadać ten wielowymiarowy problem optymalizacji. Na początek wybór liczby startów w metodzie wielokrotnego startu nie jest oczywisty w przestrzeniach o wysokiej wymiarowości. Choć intuicyjnie może się wydawać, że im więcej wymiarów, tym trudniejszy problem, czasami nie jest to takie oczywiste…

Zadanie 3.8: Skonfiguruj algorytm treningowy gradientu z wielokrotnym startem, który wykonuje 1500 obliczeń funkcji celu dla problemu klasyfikacji 25-liniowej.

Przetestuj liczbę startów równą 1, 3, 5, 10, 15 i 30, dostosowując pozostałe parametry tak, aby uzyskać jak najbardziej spójne i dobre wyniki. Następnie wykonaj ten sam eksperyment dla problemu klasyfikacji 5-liniowej. Na podstawie wyników sformułuj wnioski.

Dodanie momentu do algorytmu gradientowego

Nasz algorytm gradientowy nie wykorzystuje jeszcze informacji o momencie (bezwładności). Dodajmy ją do naszego kodu. Kolejne punkty sprawdzane przez algorytm będziemy chcieli wyznaczyć za pomocą następującego wzoru:

$$v_t = \beta v_{t-1} + (1-\beta)\nabla f(\theta_t)$$

Gdzie:
v_t jest prędkością zmian parametrów (tj. Faktycznym krokiem optymalizacji)
β jest wagą momentu (jeśli równa 0, algorytm zamienia się w zwykły algorytm gradientowy)

$\nabla f(\theta_t)$ jest gradientem funkcji celu w punkcie θ_t

Teraz v_t może być dodane do naszych współrzędnych używając kroku uczenia η :

$$\theta_{t+1} = \theta_t - \eta v_t$$

Prawdopodobnie dobrym pomysłem byłoby ponowne rozpoczęcie od prostej funkcji optymalizacyjnej 2D, aby upewnić się, że metoda działa zgodnie z oczekiwaniami. Gdy już uzyskamy działające rozwiązanie dla problemu optymalizacji 2D, będziemy mogli łatwo zaimplementować je również dla naszego wielowymiarowego problemu treningu klasyfikatora.

Jak powinniśmy wybrać wagę momentu? Musimy oczywiście wybrać wartość w zakresie od 0 (brak wpływu momentu) do prawie 1 (całkowita dominacja momentu). Dobrym punktem startowym (wartością domyślną) będzie β = 0.9. Zwróć uwagę, że teraz nasza wartość współczynnika uczenia może być znacznie niższa, ponieważ algorytm sam przyspieszy dzięki momentowi.

Zadanie 3.9: Zmodyfikuj algorytm optymalizacyjny do postaci wykorzystującej informację o momencie, skonfiguruj go a następnie powtórz zadanie 3.6 i porównaj wyniki z wcześniej otrzymanymi.