
# Wydział Informatyki

Metody sztucznej inteligencji

## Laboratorium 01 IUz-22 Urbaniak

Sprawozdanie

Autor: Sergiusz Urbaniak
Grupa: IUz-22
Data: 14 grudnia 2009

# Spis treści

# 1 Bramki logiczne

## 1.1 Bramka OR

Bramka OR posiada tablice prawdy pokazaną w tablicy 1.

| Wejście | Wyjście |
|---|---|
| 0 0 | 0 |
| 0 1 | 1 |
| 1 0 | 1 |
| 1 1 | 1 |

Tablica 1: Tablica prawdy bramki OR

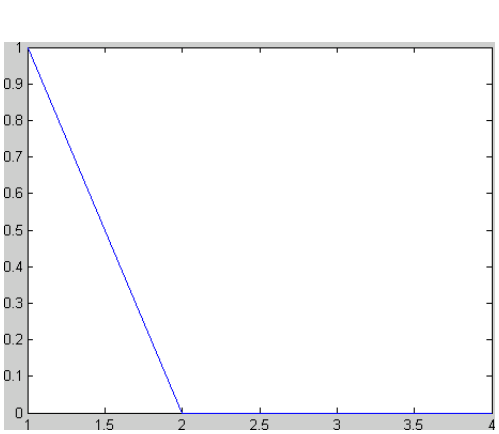
Wartości wejściowe są wpisane do zmiennej `we`. Dane wyjściowe według tablicy prawdy 1 są wpisane w zmienną `wy`. Następnie jest tworzona "sieć" jednego perceptronu i przeprowadzona symulacja działania sieci bez uczenia perceptronu. Kod źródłowy jest dostępny w listingu 1.

```
we = [0 0 1 1; 0 1 0 1];
wy = [0 1 1 1];

net = newp(minmax(we), 1);
y = sim(net, we);

figure(1)
plot(abs(y-wy));

figure(2)
plotpv(we, wy);
plotpc(net.iw{1,1}, net.b{1});
```

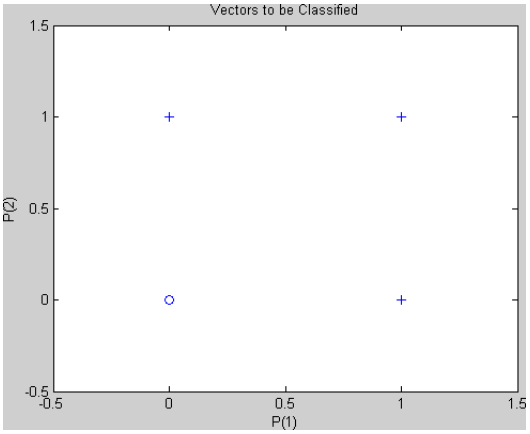
Listing 1: Kod nie uczonej bramki OR

Wykres błędu jak i danych generowany kodu 1 jest widoczny na rysunkach 1.

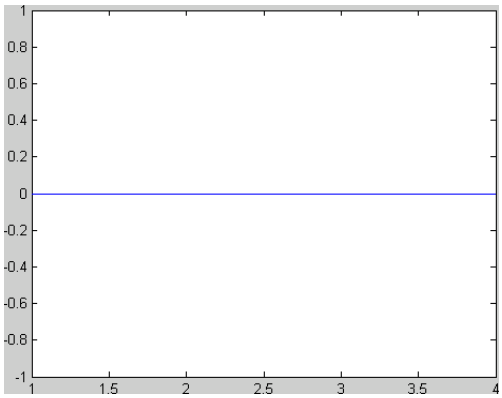
Kod zmieniony na uczenie perceptronu jest widoczny w listingu 2. Dodane zostały polecenia `init` i `train`.



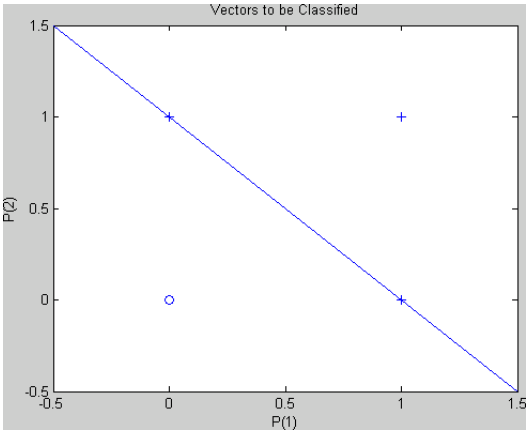
(a) Wykres błędu (nie uczone)




(b) Linia podziału (nie uczone)



(c) Wykres błędu (uczone)




(d) Linia podziału (uczone)

Rysunek 1: Wykresy bramki OR

```
we = [0 0 1 1; 0 1 0 1];
wy = [0 1 1 1];

net = newp(minmax(we), 1);
net = init(net);
net.trainParam.epochs = 50;
net = train(net, we, wy);

y = sim(net, we);
figure(1)
plot(abs(y-wy));

figure(2)
plotpv(we, wy);
plotpc(net.iw{1,1}, net.b{1});
```

Listing 2: Kod uczonej bramki OR

Można zaobserwować że uczenie nie pozostawiło żadnego błędu. Perceptron poprawnie nauczył się bramki OR.

## 1.2 Bramka XOR

Bramka XOR posiada tablice prawdy pokazaną w tablicy 2.

| Wejście | Wyjście |
|---|---|
| 0 0 | 0 |
| 0 1 | 1 |
| 1 0 | 1 |
| 1 1 | 0 |

Tablica 2: Tablica prawdy bramki XOR

Kod perceptronu nie uczonego jest widoczny w listingu 3. Jak widać została tylko zmieniona zmienna `wy`.

```
we = [0 0 1 1; 0 1 0 1];
wy = [0 1 1 0];

net = newp(minmax(we), 1);
y = sim(net, we);

figure(1)
plot(abs(y-wy));

figure(2)
plotpv(we, wy);
plotpc(net.iw{1,1}, net.b{1});
```

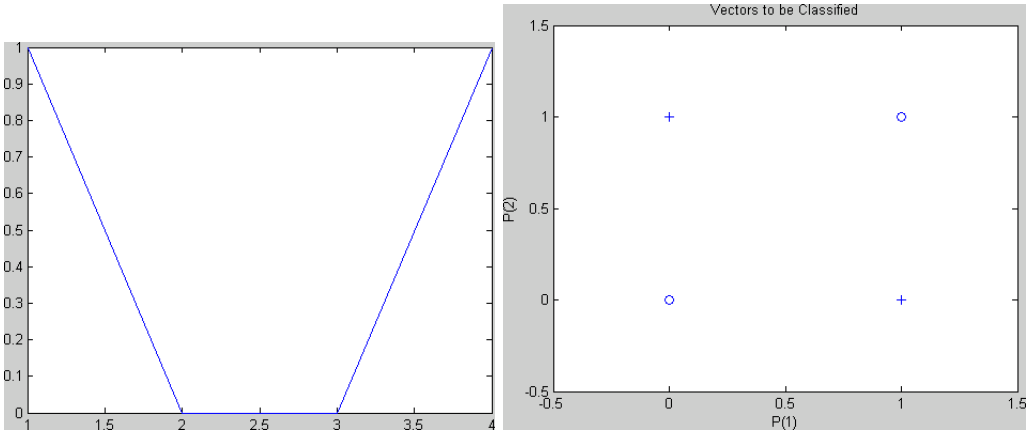
Listing 3: Kod nie uczonej bramki XOR

Wykres błędu jak i danych generowany kodu 3 jest widoczny na rysunkach 2.

Kod zmieniony na uczenie perceptronu jest widoczny w listingu 4. Znów dodane zostały polecenia `init` i `train`.

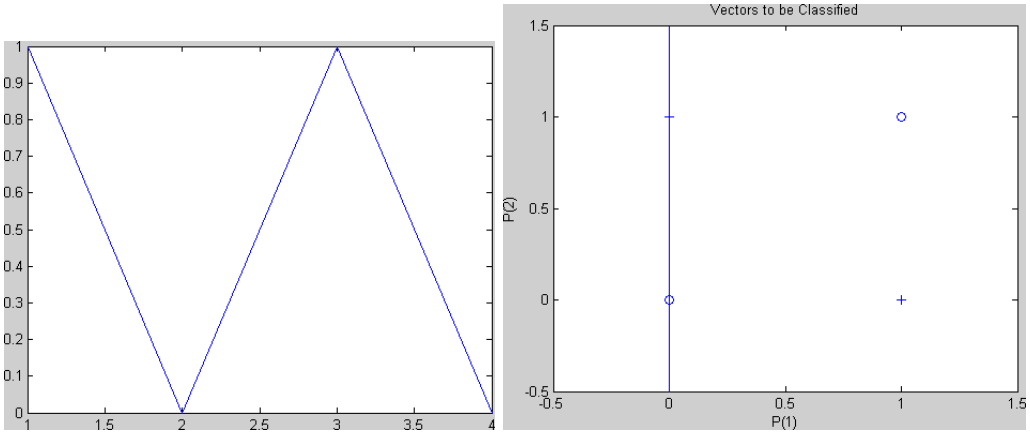
```
we = [0 0 1 1; 0 1 0 1];
wy = [0 1 1 0];

net = newp(minmax(we), 1);
net = init(net);
net.trainParam.epochs = 50;
net = train(net, we, wy);

y = sim(net, we);
figure(1)
plot(abs(y-wy));

figure(2)
plotpv(we, wy);
plotpc(net.iw{1,1}, net.b{1});
```

Listing 4: Kod uczonej bramki XOR




(a) Wykres błędu (nie uczone)

(b) Linia podziału (nie uczone)




(c) Wykres błędu (uczone)

(d) Linia podziału (uczone)

Rysunek 2: Wykresy bramki XOR

Można zaobserwować że uczenie nie udało się. Nadal istnieją błędy. Powodem jest fakt że podział danych bramki XOR nie jest lineowo separowalny.

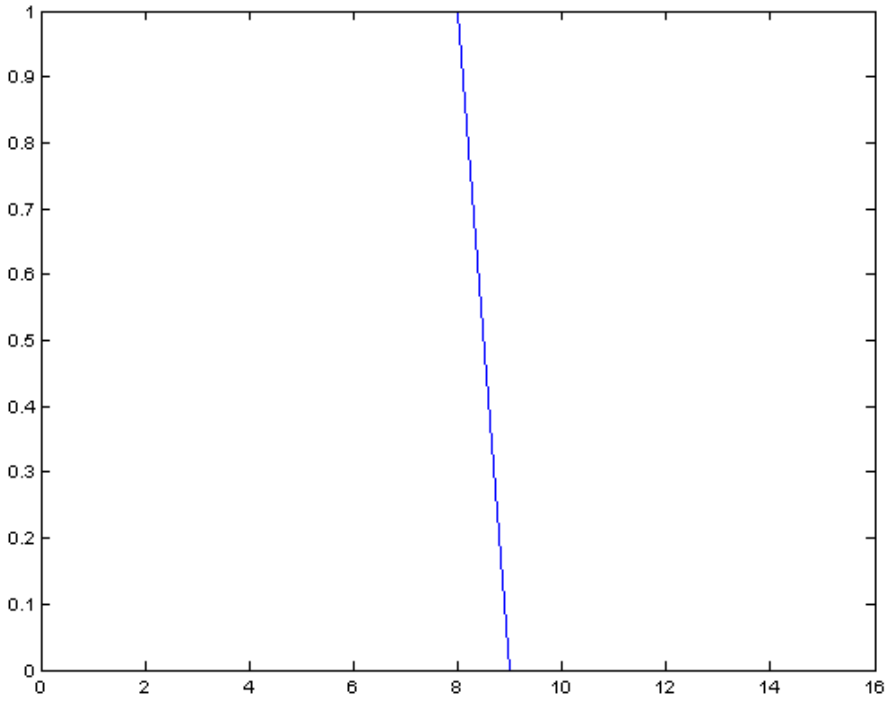
## 2 Dane wczytane z plików

Aby ułatwić generowanie i wczytanie dużej ilości wykresów został stworzony nowy skrypt widoczny w listingu 5. Ten skrypt bierze jako parametr prefiks sprawdzanych danych `prefix`. Na podstawie prefiksu są wczytane dane z plików `[prefix '_i.txt']` i `[prefix '_o.txt']`. Zmienna `learn` podaje czy neuron ma być uczony czy nie. W końcu zmienna `epochs` specyfikuje ile epok uczenia mają być uwzględnione.

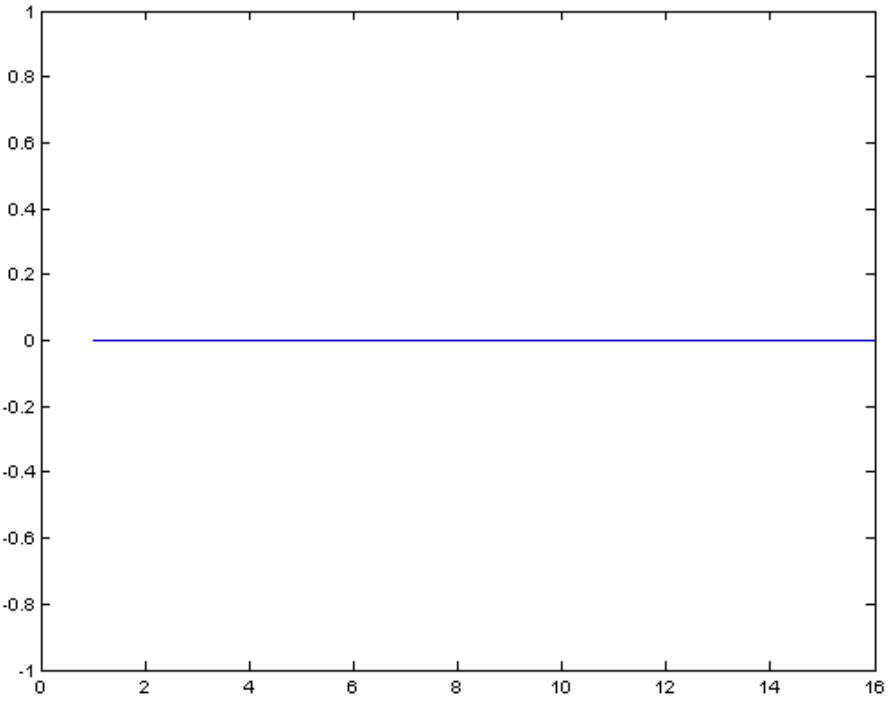
```
function load_learn_plot(prefix, learn, epochs)
    we = load([prefix '_i.txt']);
    wy = load([prefix '_o.txt']);

    net = newp(minmax(we), 1);

    if learn
        net = init(net);
        net.trainParam.epochs = epochs;
        net = train(net, we, wy);

        suffix = '_learned';
    else
        suffix = '_unlearned';
    end

    y = sim(net, we);

    figure(1)
    plot(abs(y-wy));
    print('-dpng', '-r75', [prefix '_error' suffix '.png'])

    figure(2)
    plotpv(we, wy);
    plotpc(net.iw{1,1}, net.b{1});
    print('-dpng', '-r75', [prefix '_data' suffix '.png'])
end
```

Listing 5: `load_learn_plot.m`: Kod do wczytania, uczenia i generowania wykresów

Wykresy generowane są widoczne w następujących stronach.

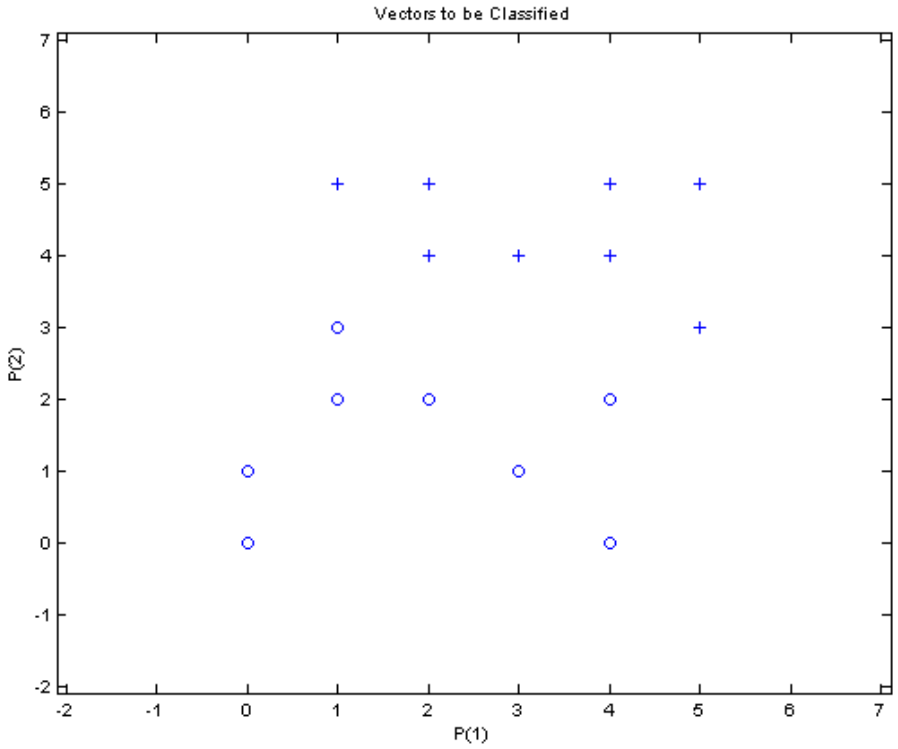



(a) Wykres błędu (nie uczone)

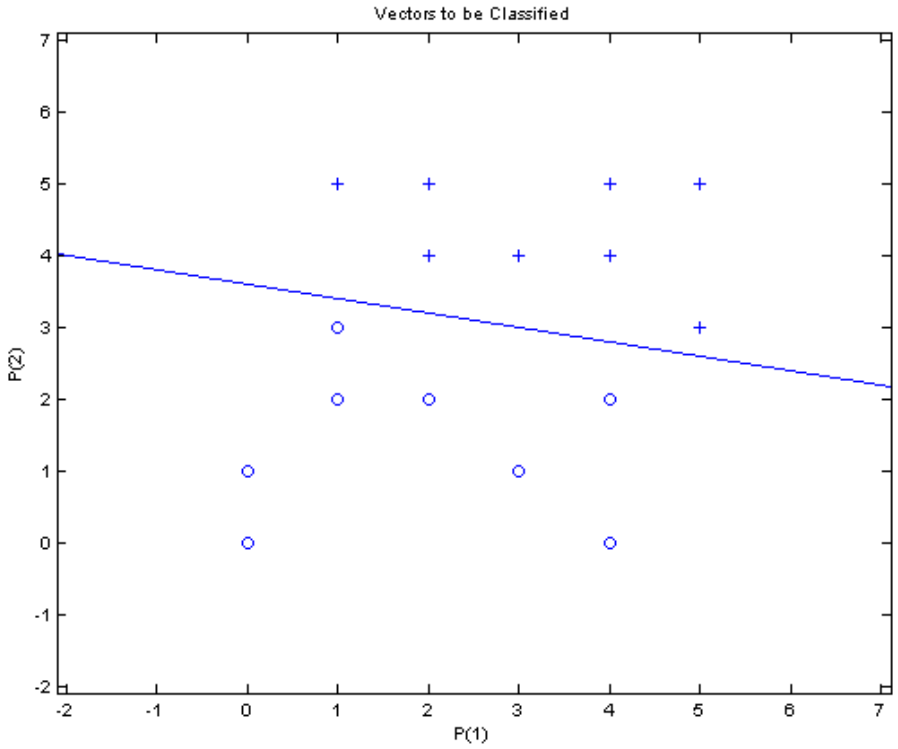



(b) Wykres błędu (uczone)

Rysunek 3: Wykresy błędu zestawu danych `percep`

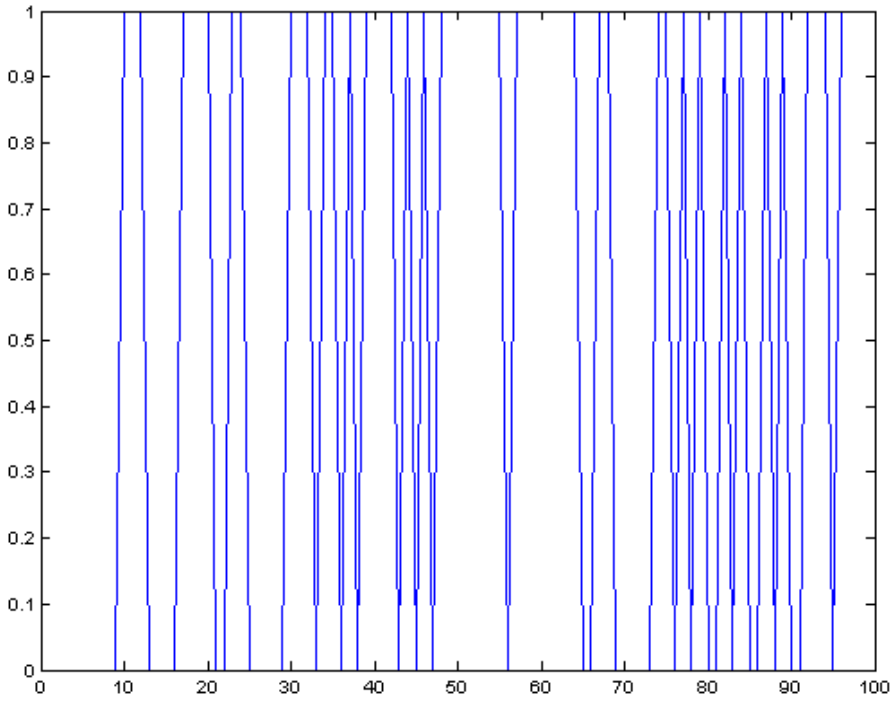



(a) Linia podziału (nie uczone)

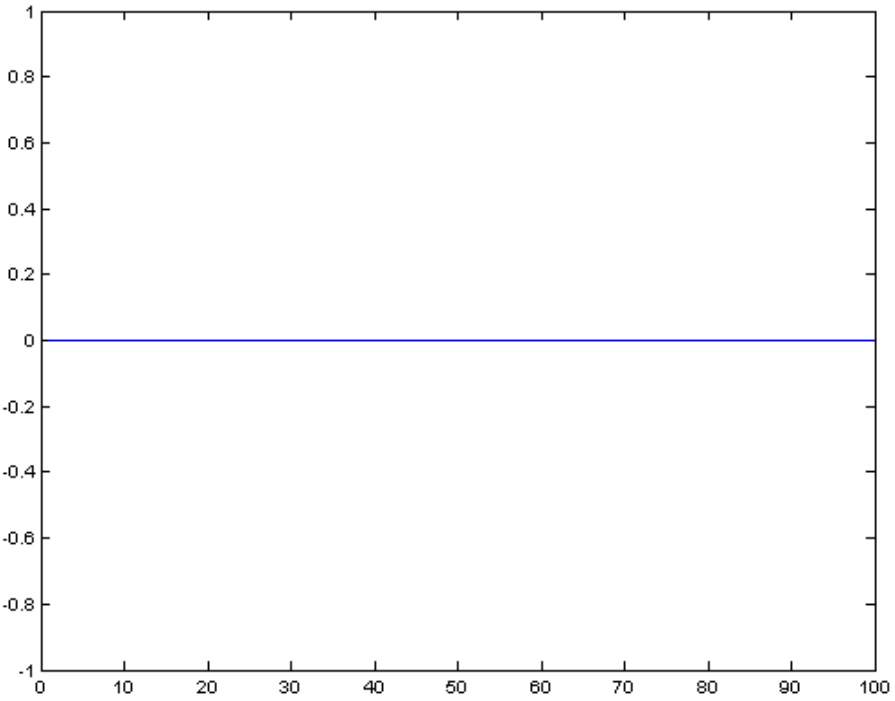



(b) Linia podziału (uczone)

Rysunek 4: Wykresy przedziału zestawu danych `percep`

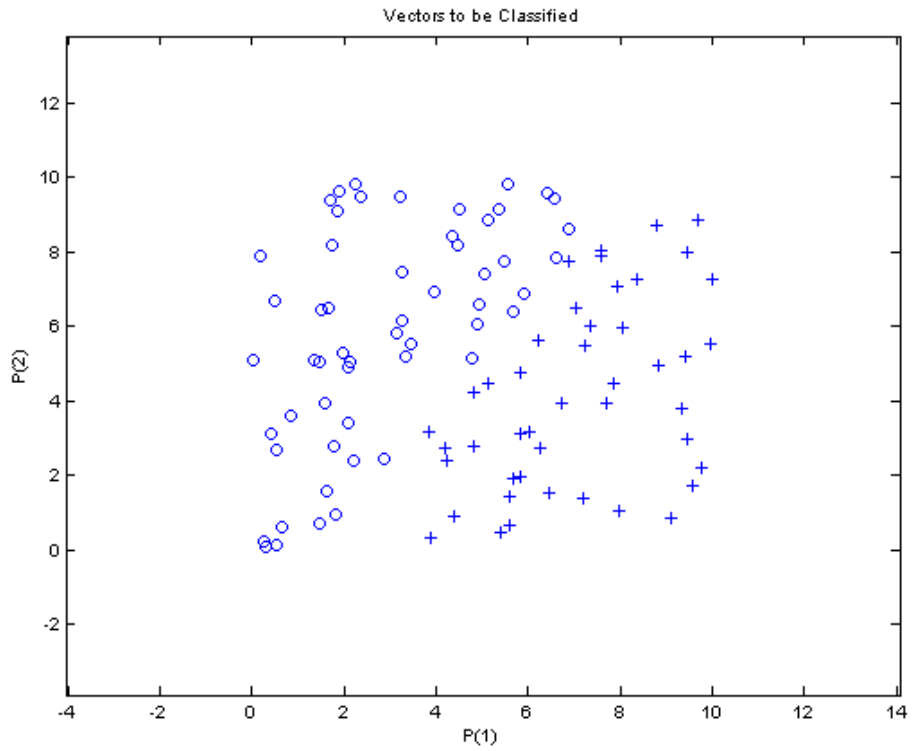



(a) Wykres błędu (nie uczone)

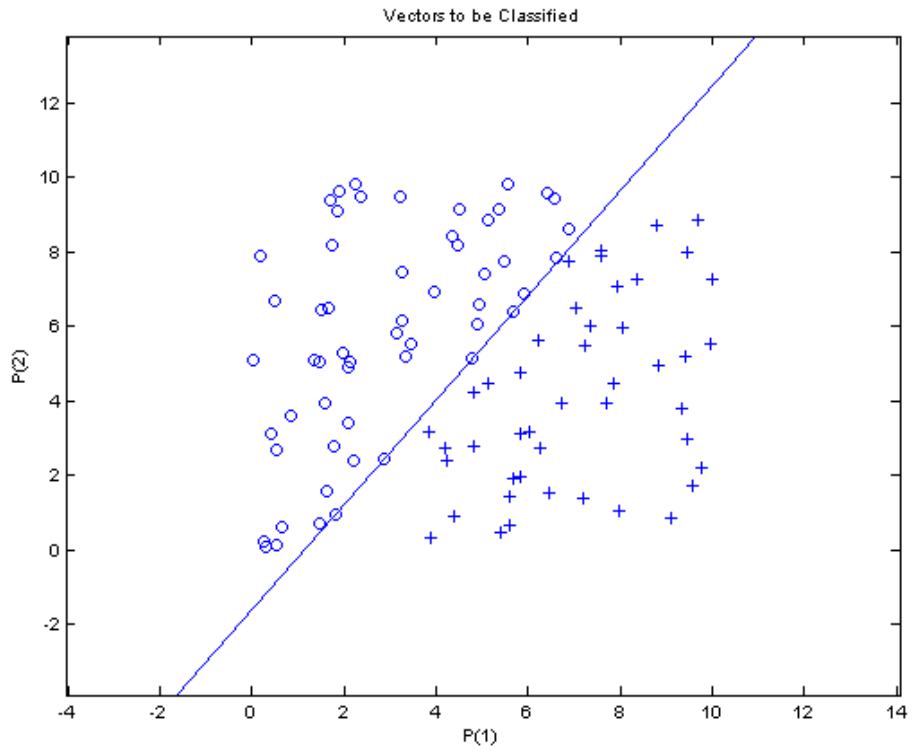



(b) Wykres błędu (uczone)

Rysunek 5: Wykresy błędu zestawu danych `dane_a`

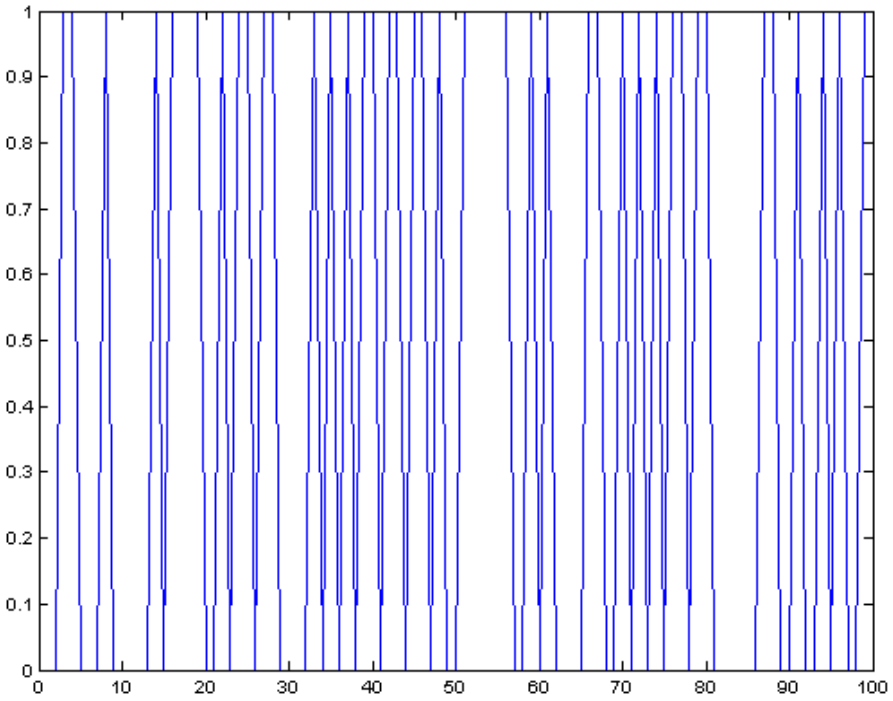



(a) Linia podziału (nie uczone)

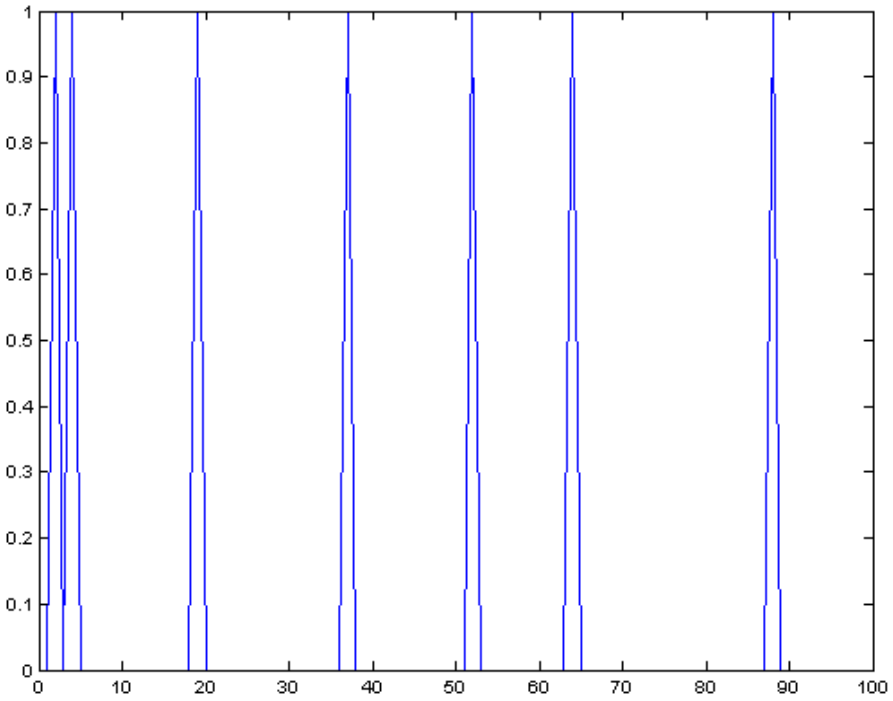



(b) Linia podziału (uczone)

Rysunek 6: Wykresy przedziału zestawu danych `dane_a`

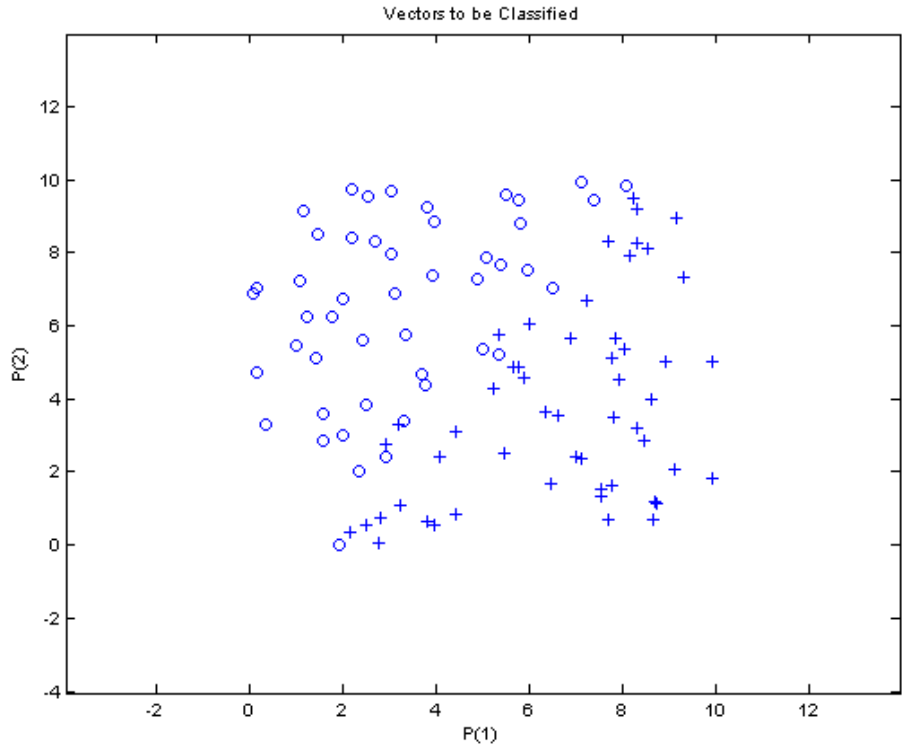



(a) Wykres błędu (nie uczone)

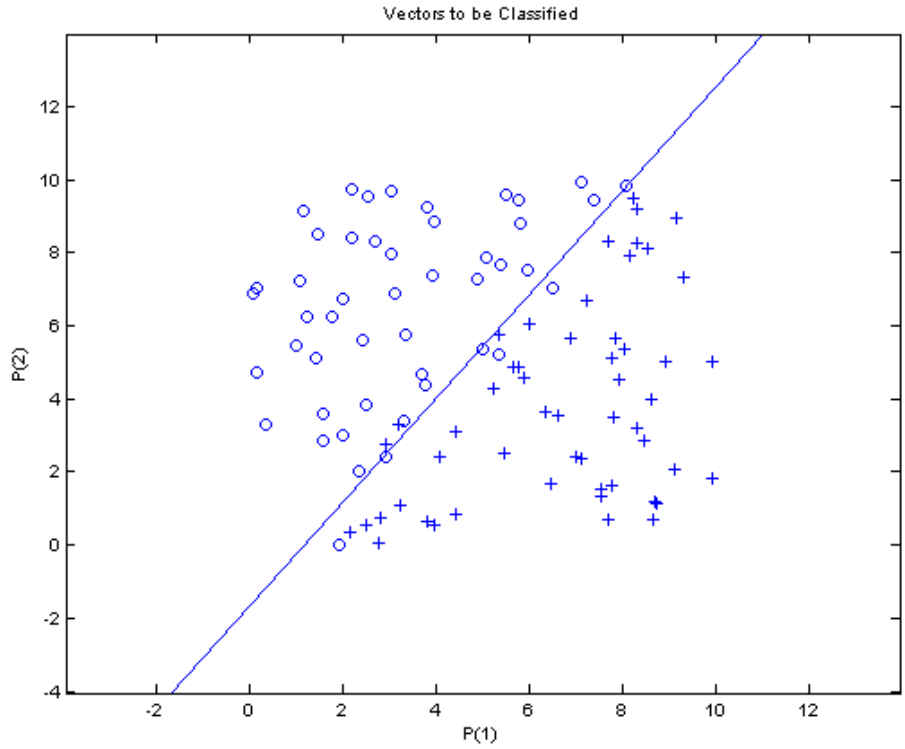



(b) Wykres błędu (uczone)

Rysunek 7: Wykresy błędu zestawu danych `dane_1`

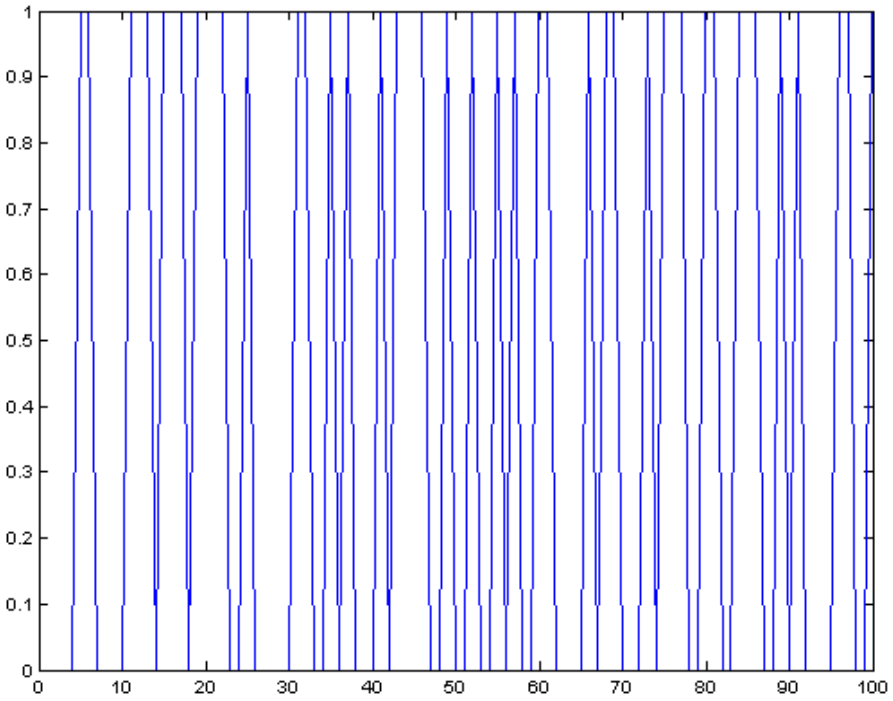



(a) Linia podziału (nie uczone)

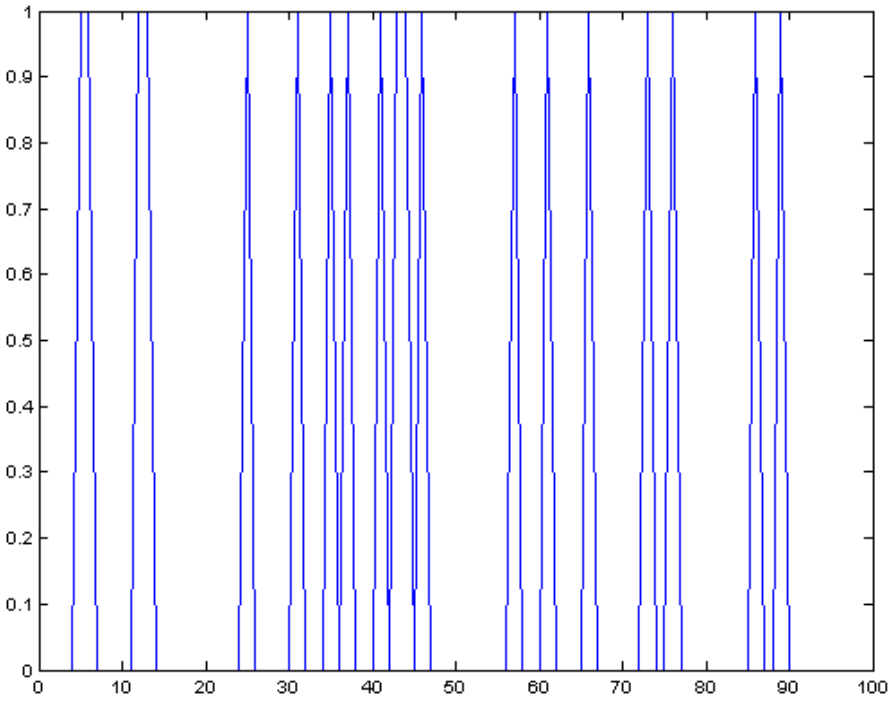



(b) Linia podziału (uczone)

Rysunek 8: Wykresy przedziału zestawu danych `dane_1`

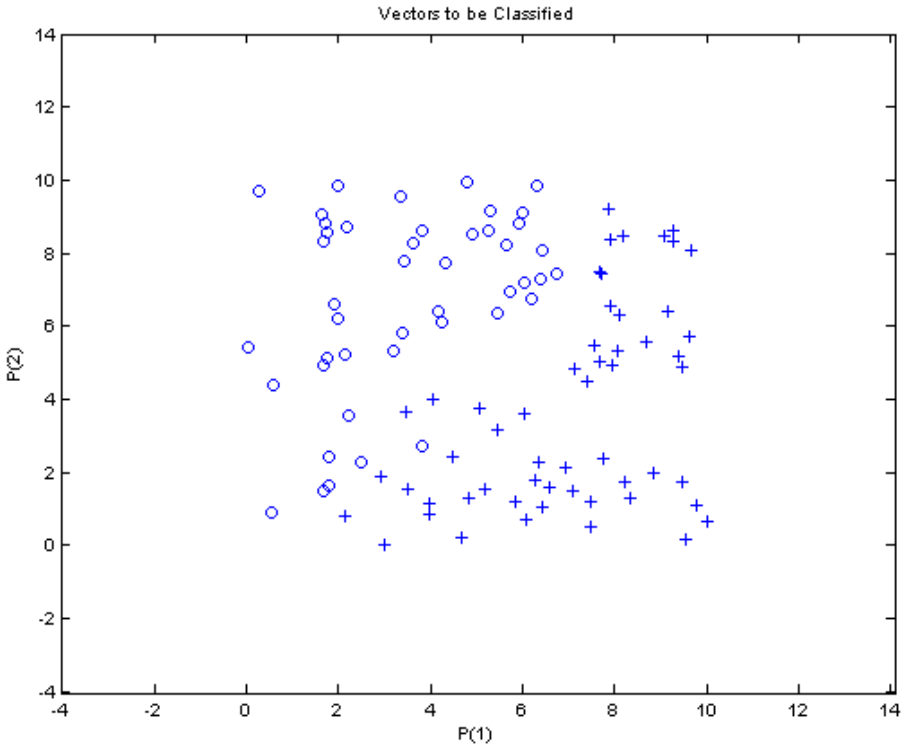



(a) Wykres błędu (nie uczone)

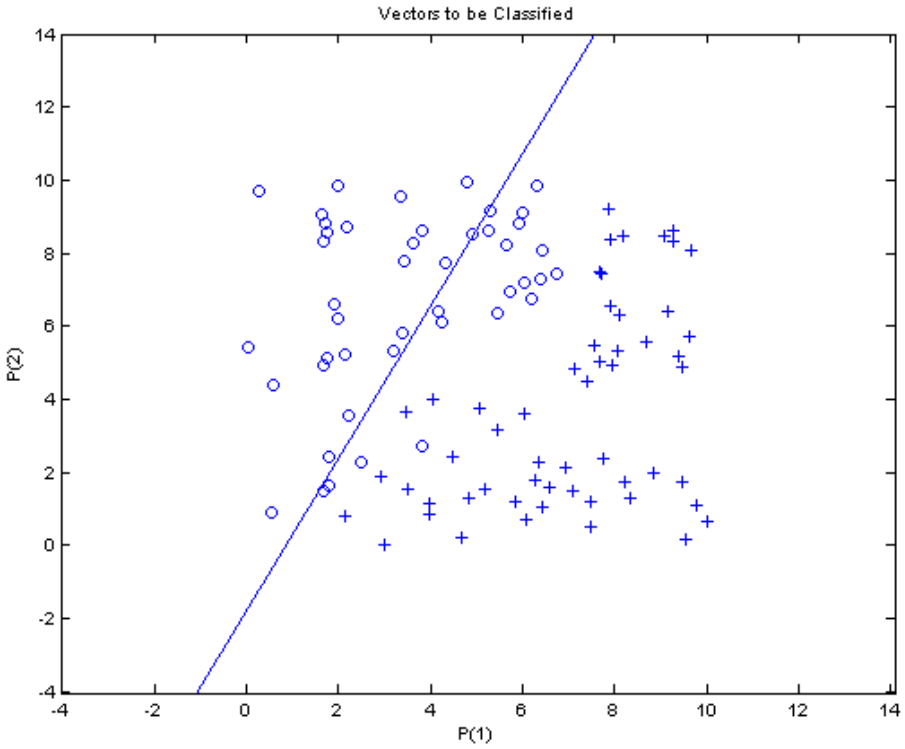



(b) Wykres błędu (uczone)

Rysunek 9: Wykresy błędu zestawu danych `dane_2`

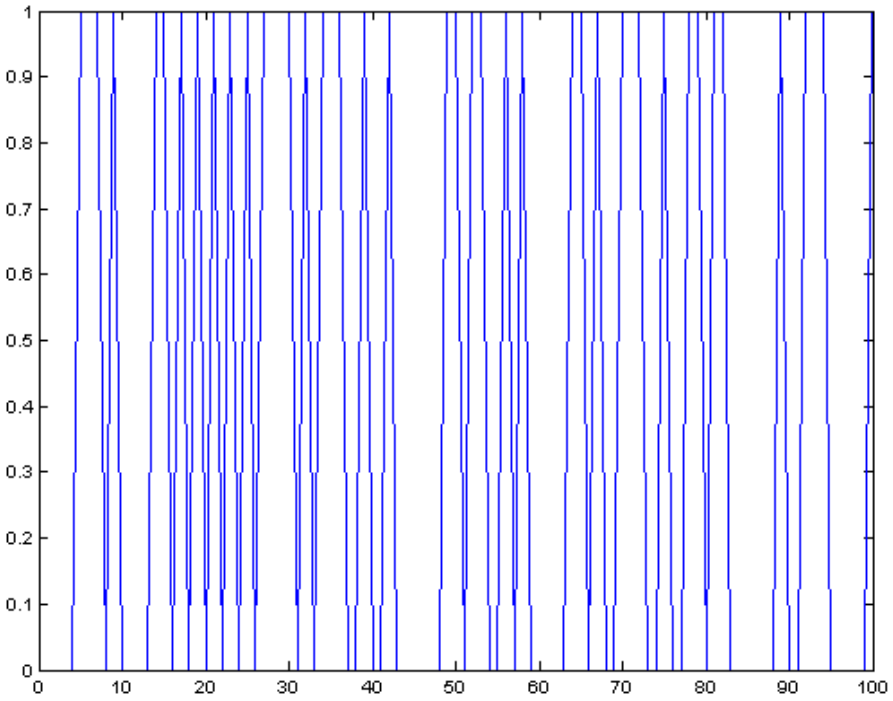



(a) Linia podziału (nie uczone)

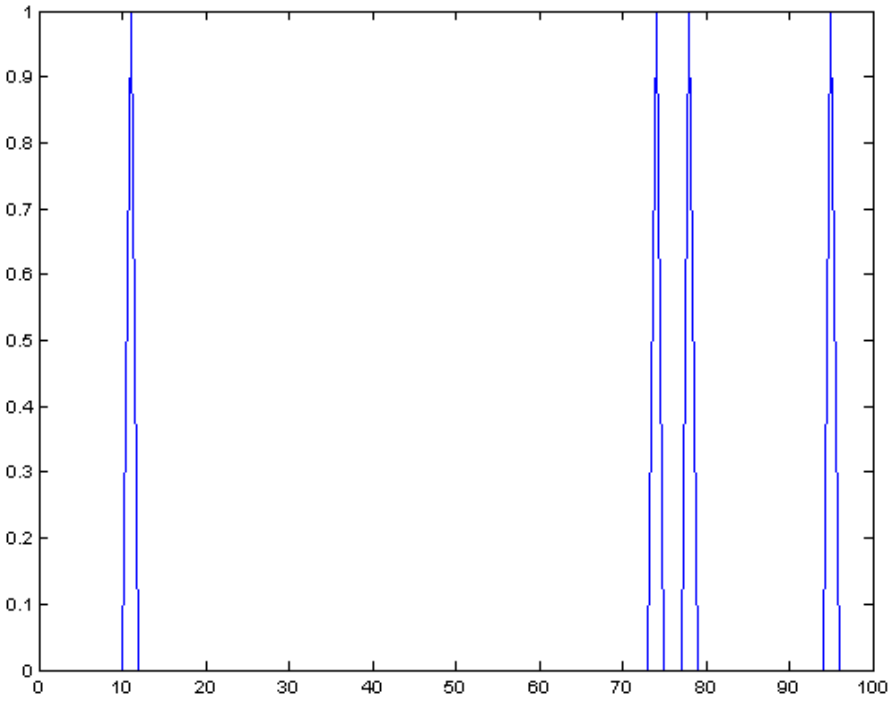



(b) Linia podziału (uczone)

Rysunek 10: Wykresy przedziału zestawu danych `dane_2`

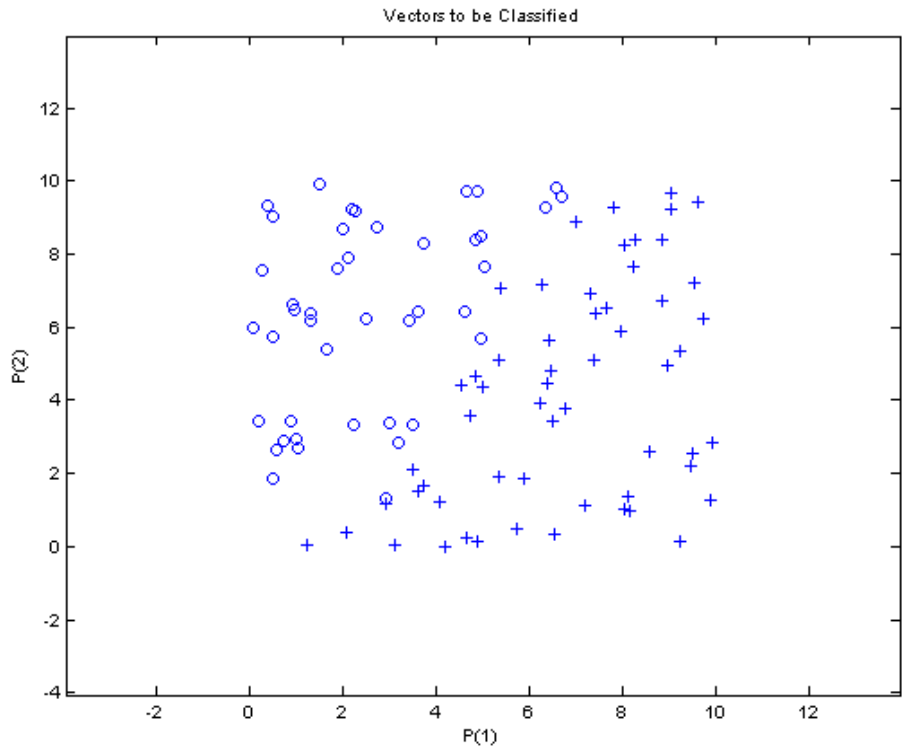



(a) Wykres błędu (nie uczone)

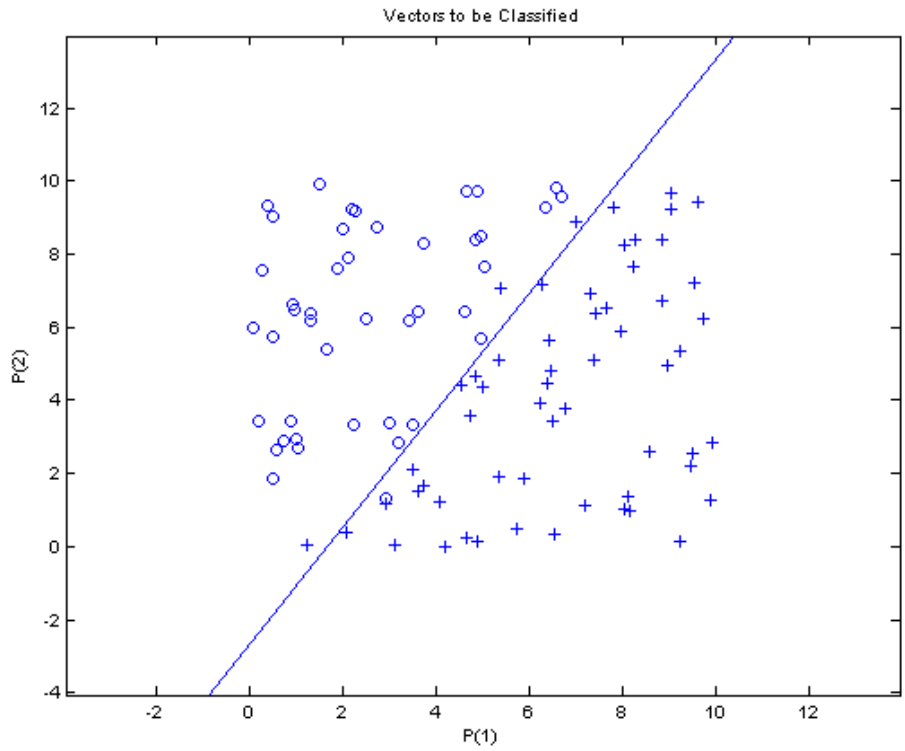



(b) Wykres błędu (uczone)

Rysunek 11: Wykresy błędu zestawu danych `dane_3`

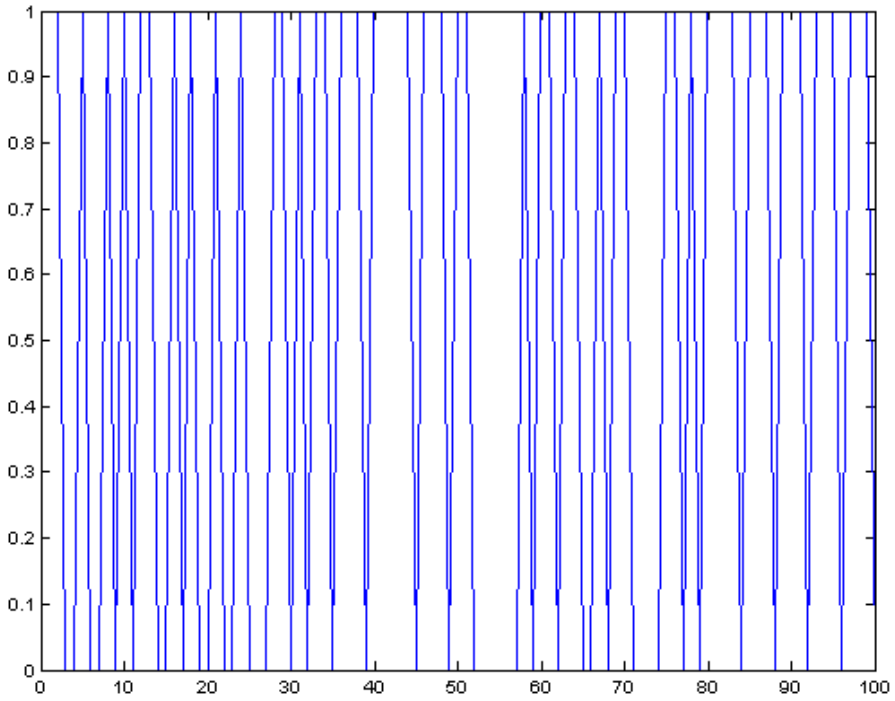



(a) Linia podziału (nie uczone)

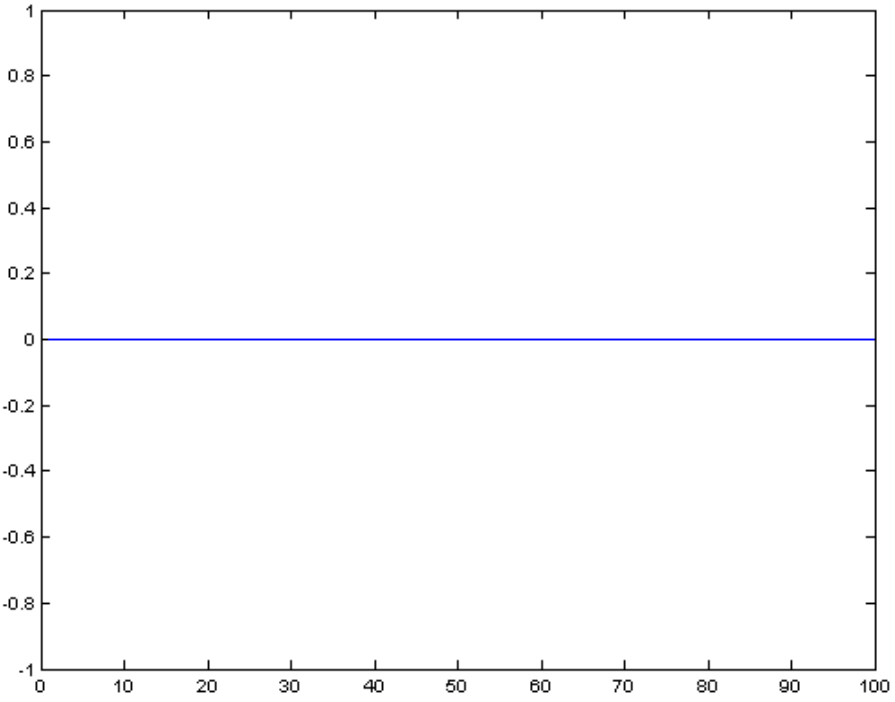



(b) Linia podziału (uczone)

Rysunek 12: Wykresy przedziału zestawu danych `dane_3`

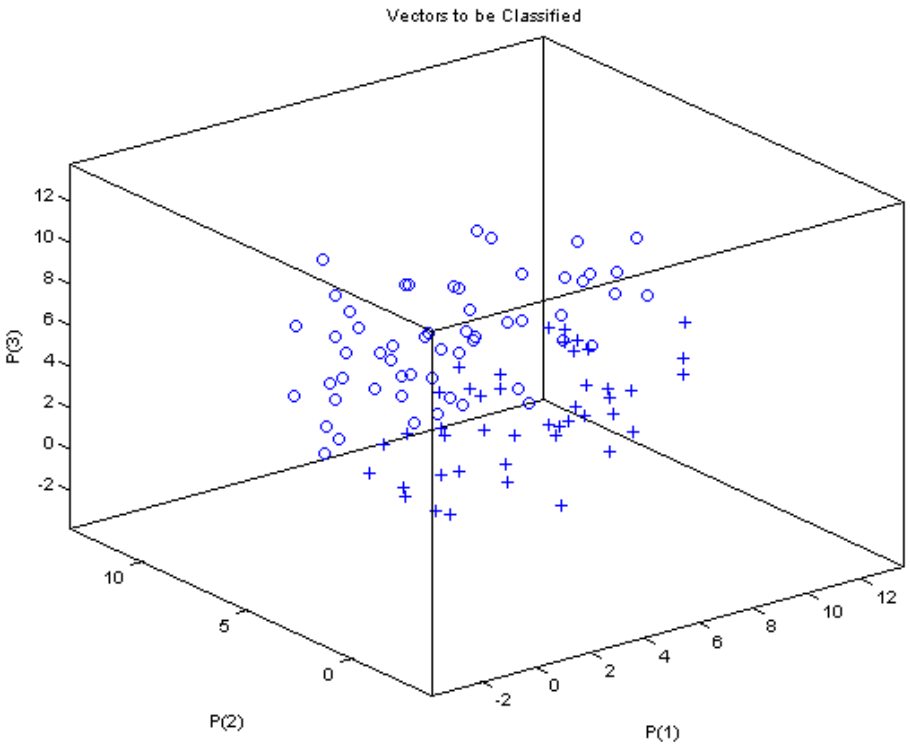



(a) Wykres błędu (nie uczone)

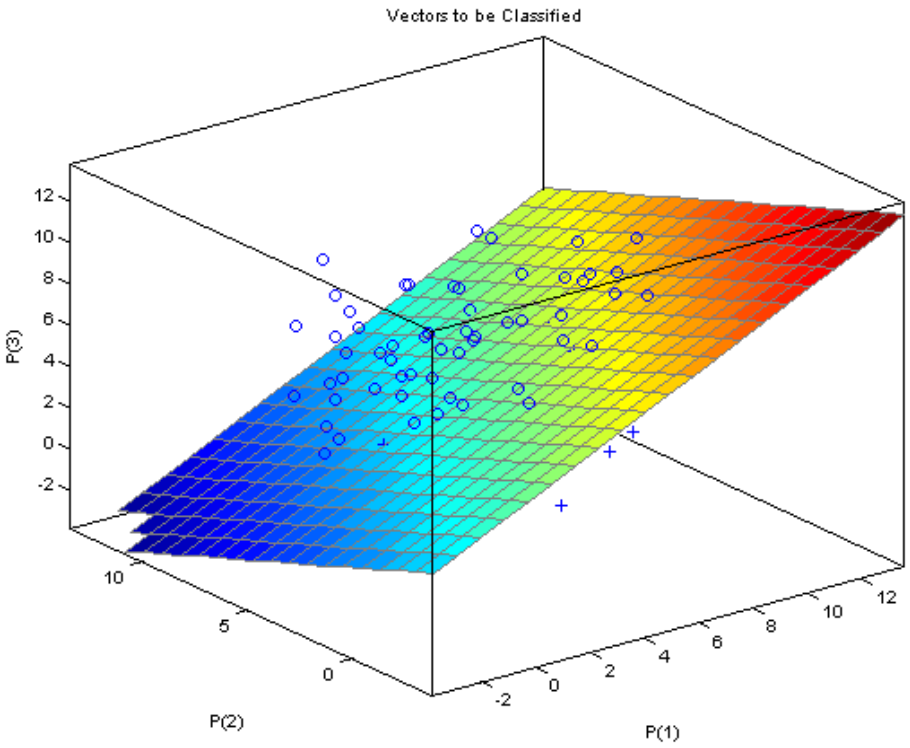



(b) Wykres błędu (uczone)

Rysunek 13: Wykresy błędu zestawu danych `dane3d_a`

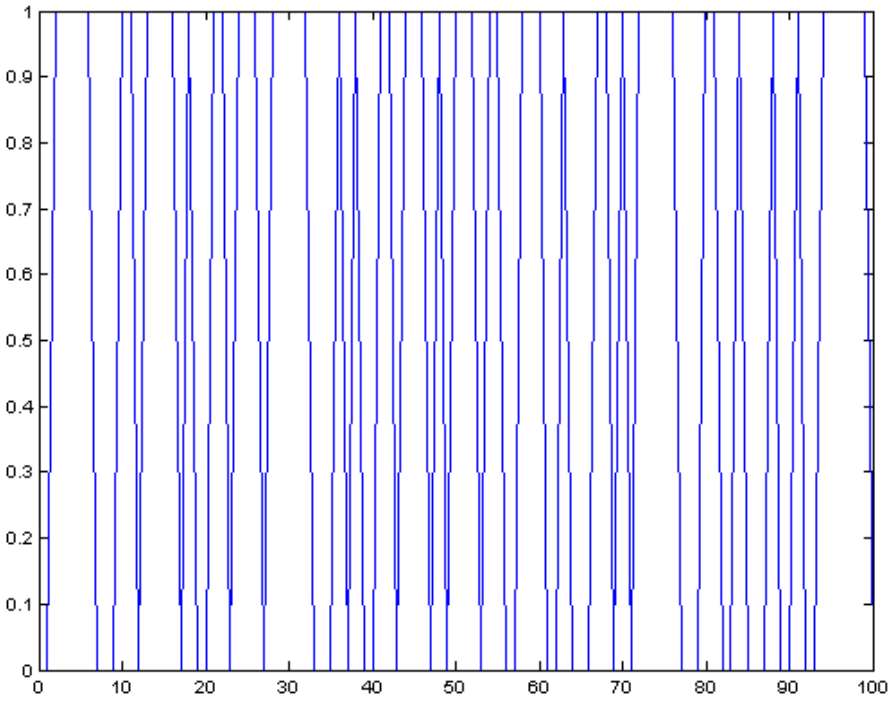



(a) Linia podziału (nie uczone)

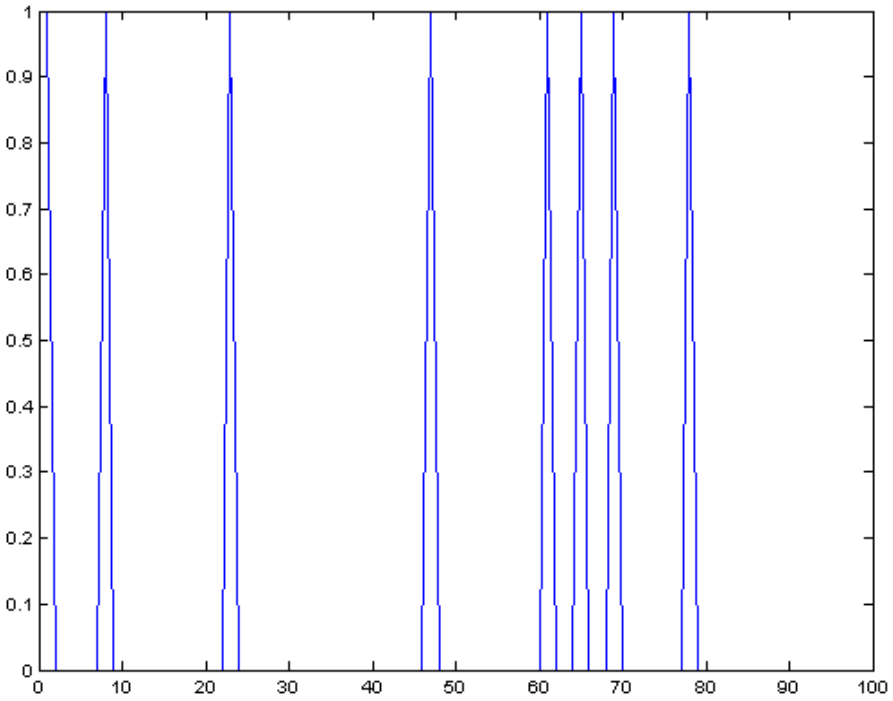



(b) Linia podziału (uczone)

Rysunek 14: Wykresy przedziału zestawu danych `dane3d_a`

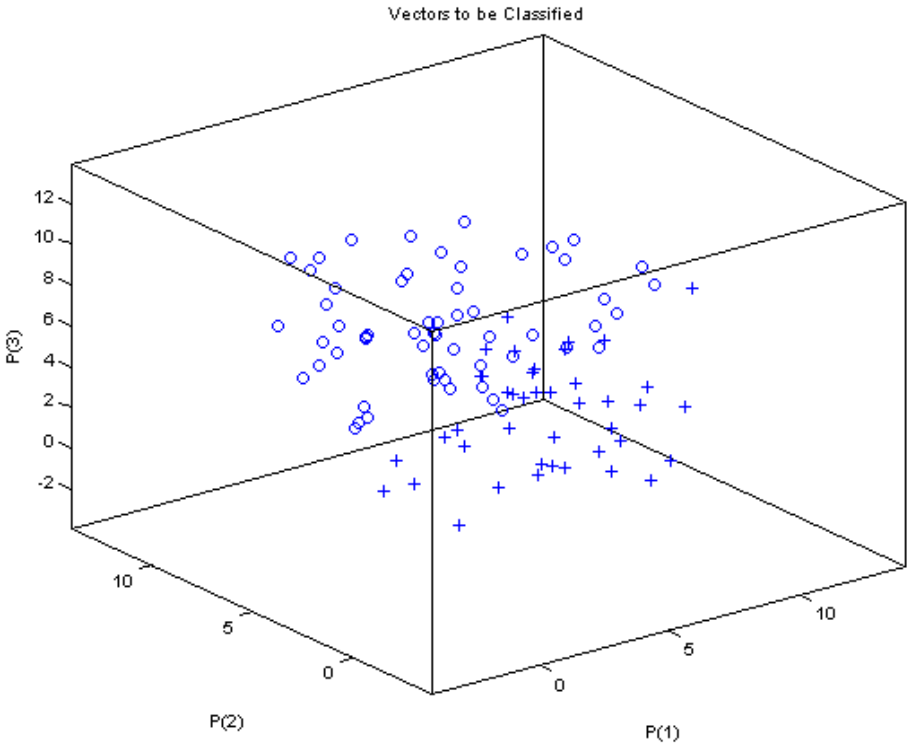



(a) Wykres błędu (nie uczone)

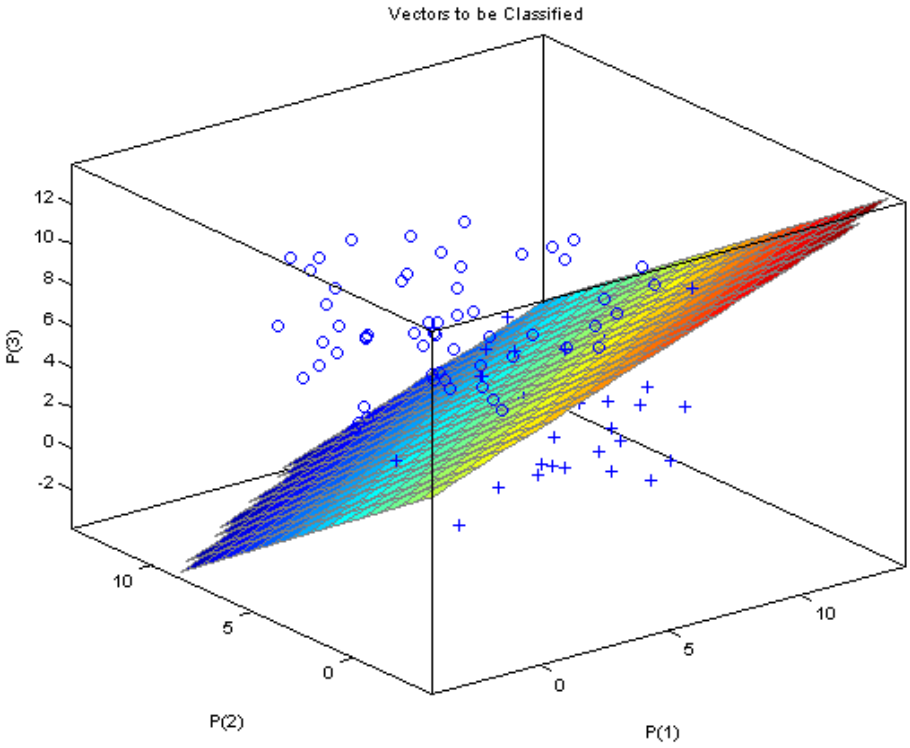



(b) Wykres błędu (uczone)

Rysunek 15: Wykresy błędu zestawu danych `dane3d_1`

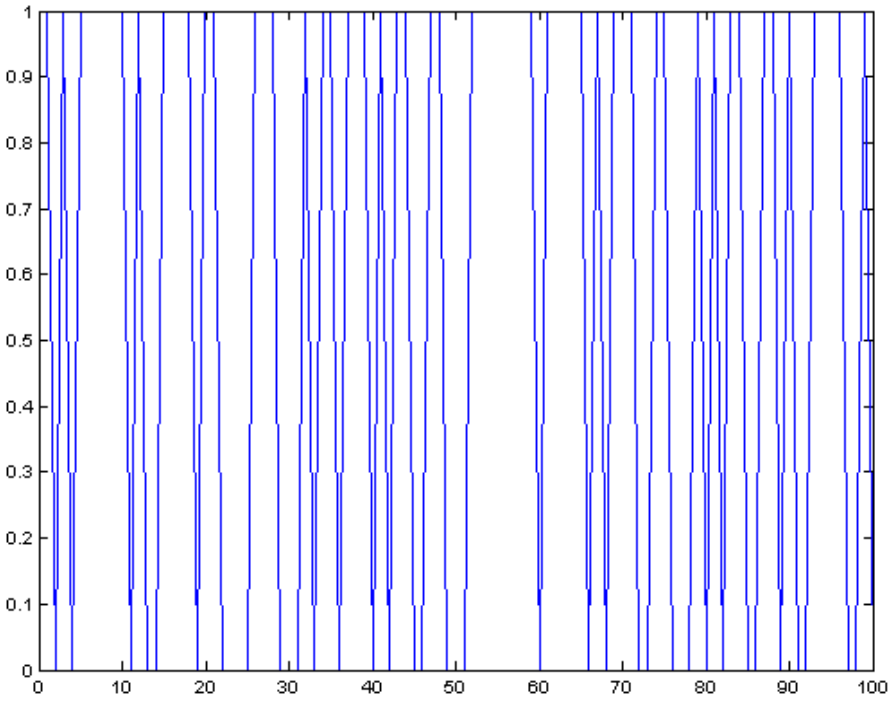



(a) Linia podziału (nie uczone)

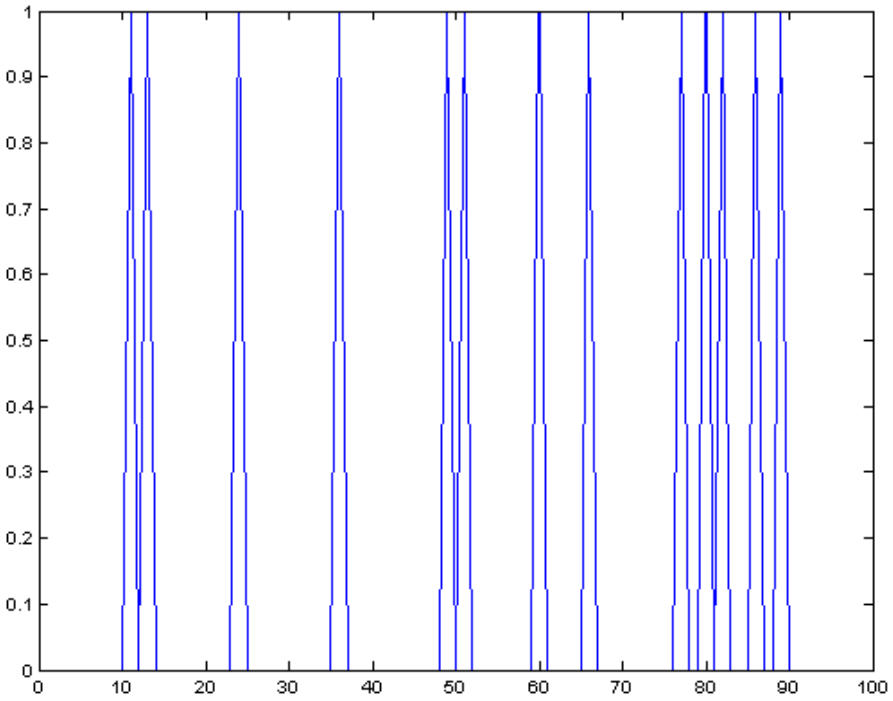



(b) Linia podziału (uczone)

Rysunek 16: Wykresy przedziału zestawu danych `dane3d_1`

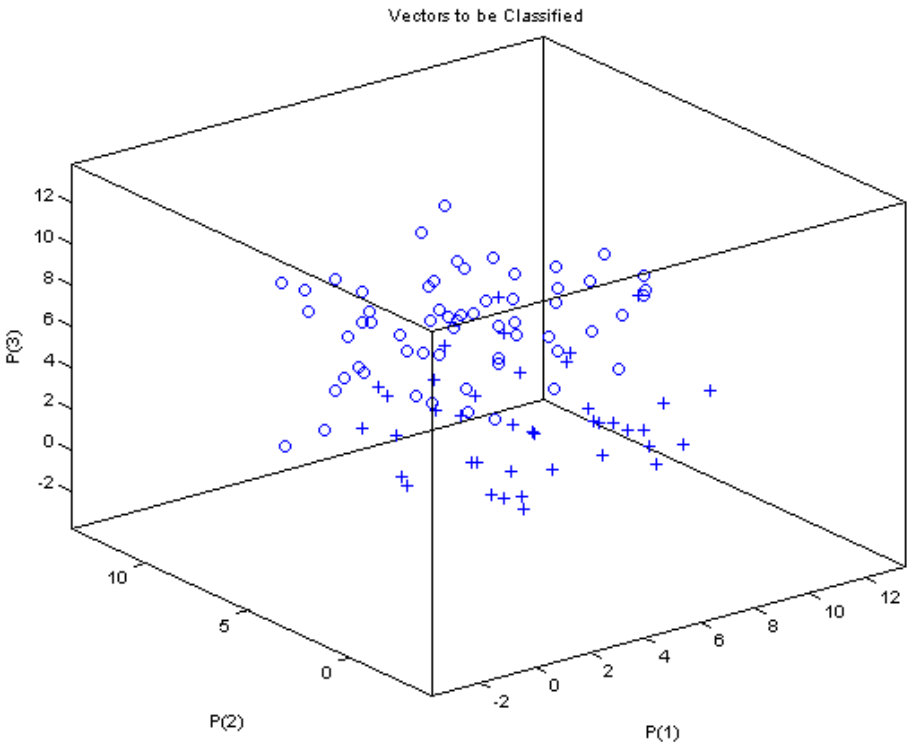



(a) Wykres błędu (nie uczone)

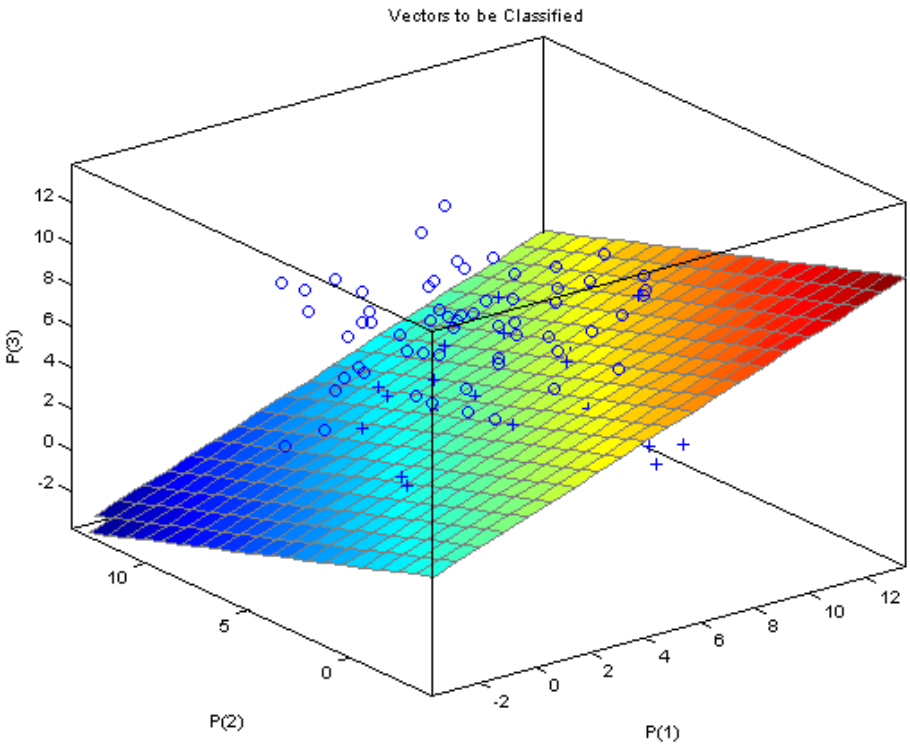



(b) Wykres błędu (uczone)

Rysunek 17: Wykresy błędu zestawu danych `dane3d_2`

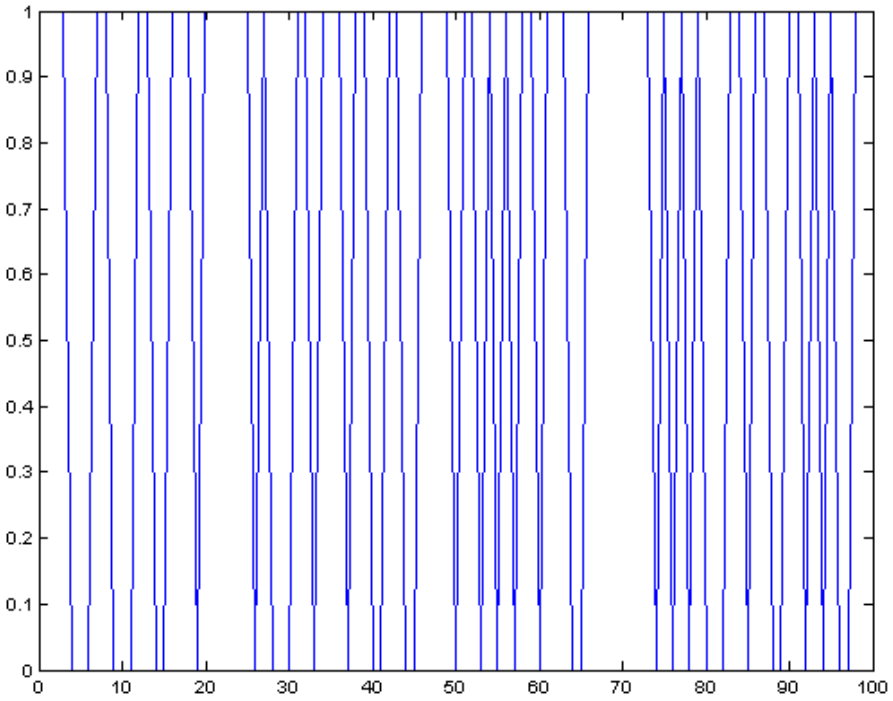



(a) Linia podziału (nie uczone)

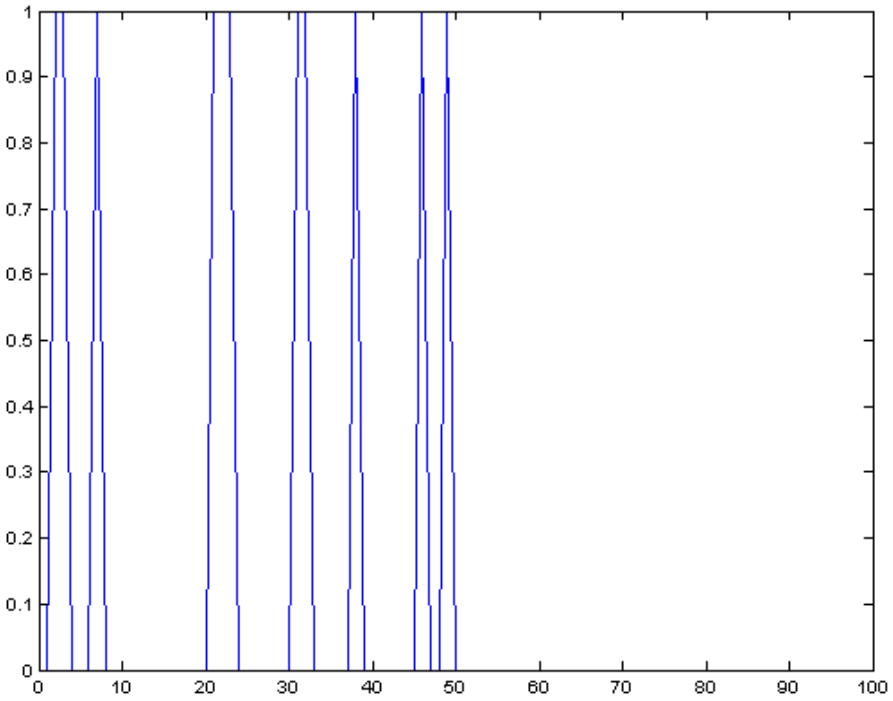



(b) Linia podziału (uczone)

Rysunek 18: Wykresy przedziału zestawu danych `dane3d_2`

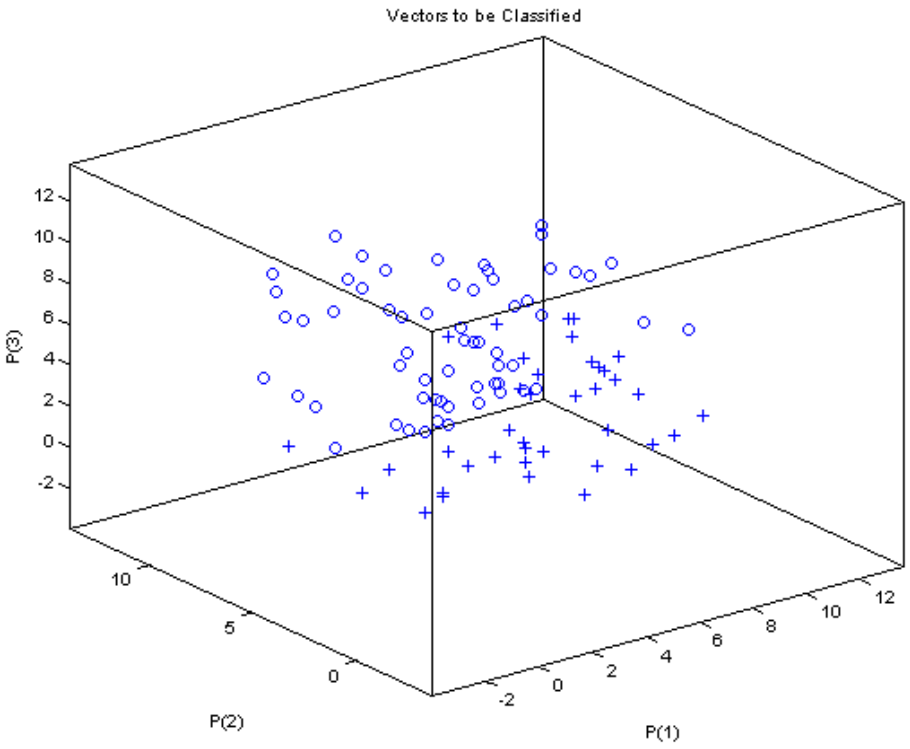



(a) Wykres błędu (nie uczone)

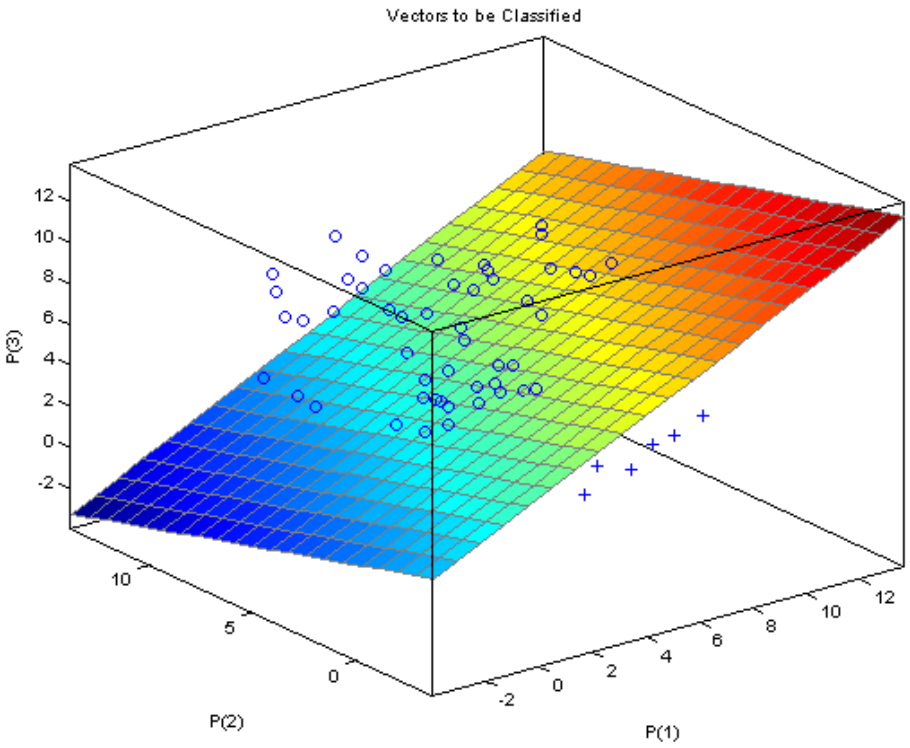



(b) Wykres błędu (uczone)

Rysunek 19: Wykresy błędu zestawu danych `dane3d_3`

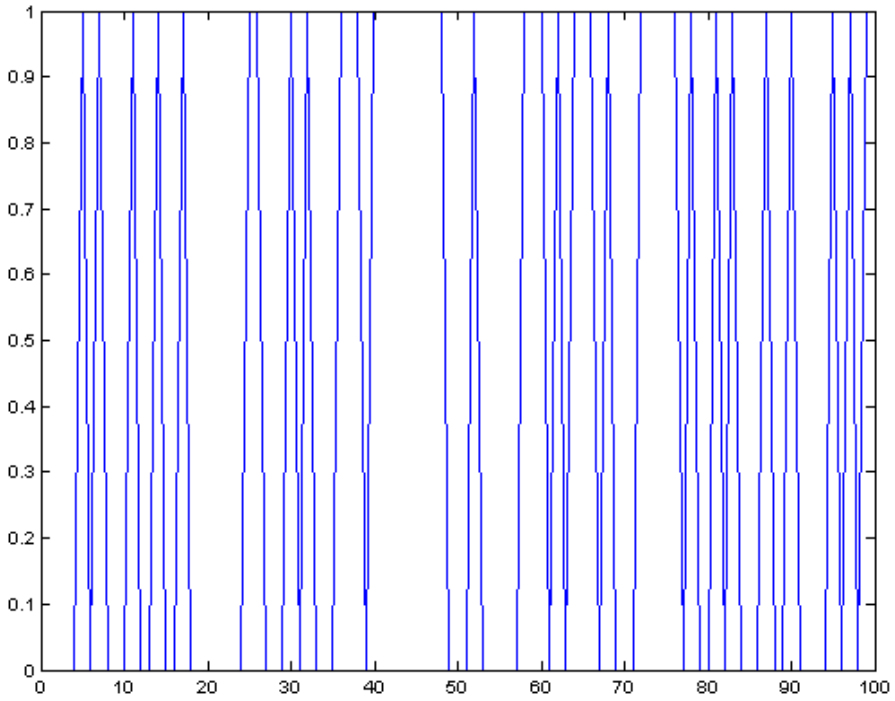



(a) Linia podziału (nie uczone)

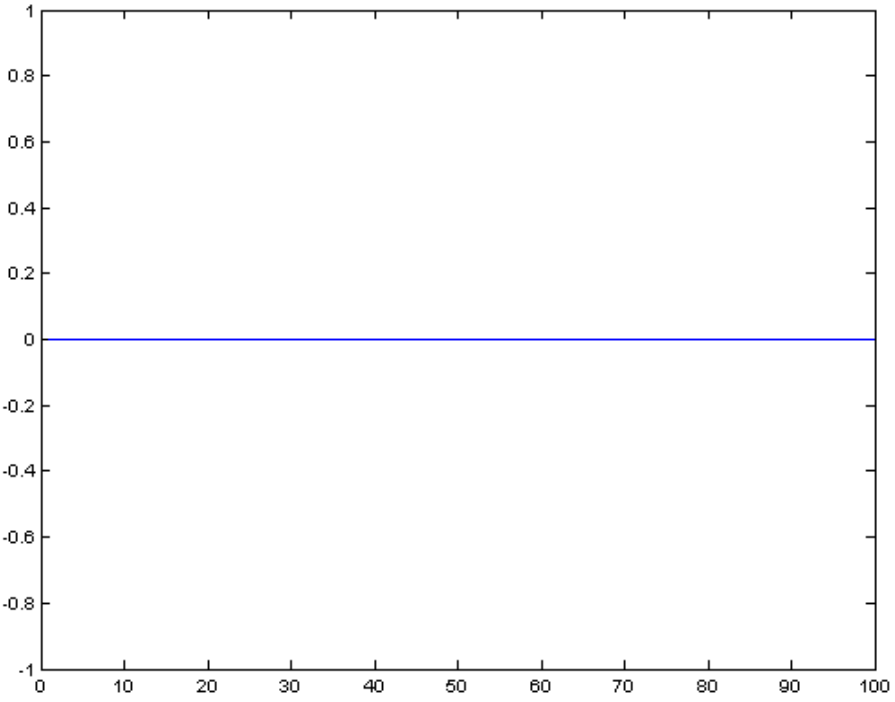



(b) Linia podziału (uczone)

Rysunek 20: Wykresy przedziału zestawu danych `dane3d_3`

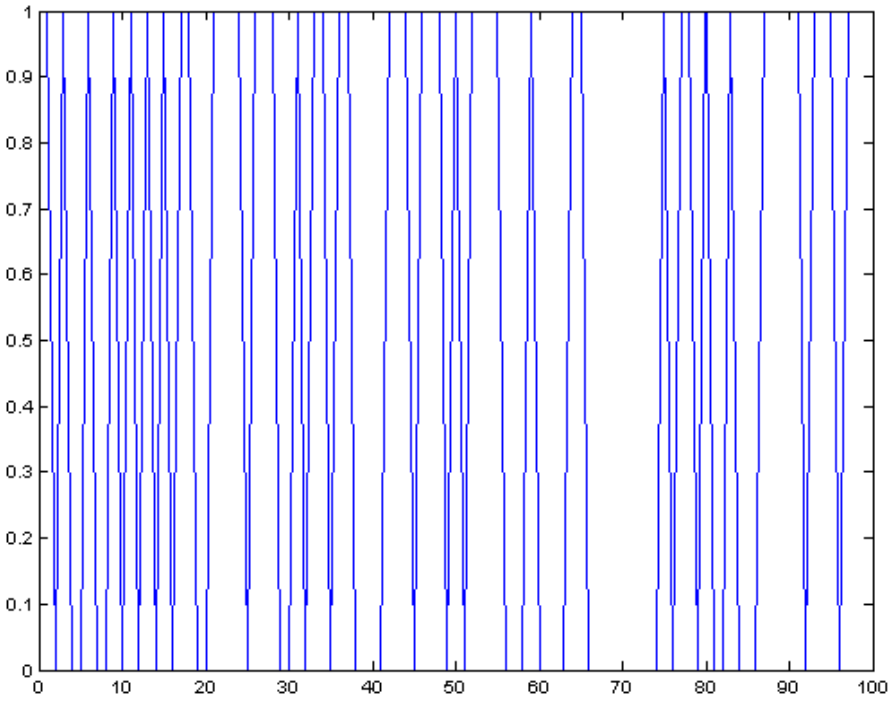



(a) Wykres błędu (nie uczone)

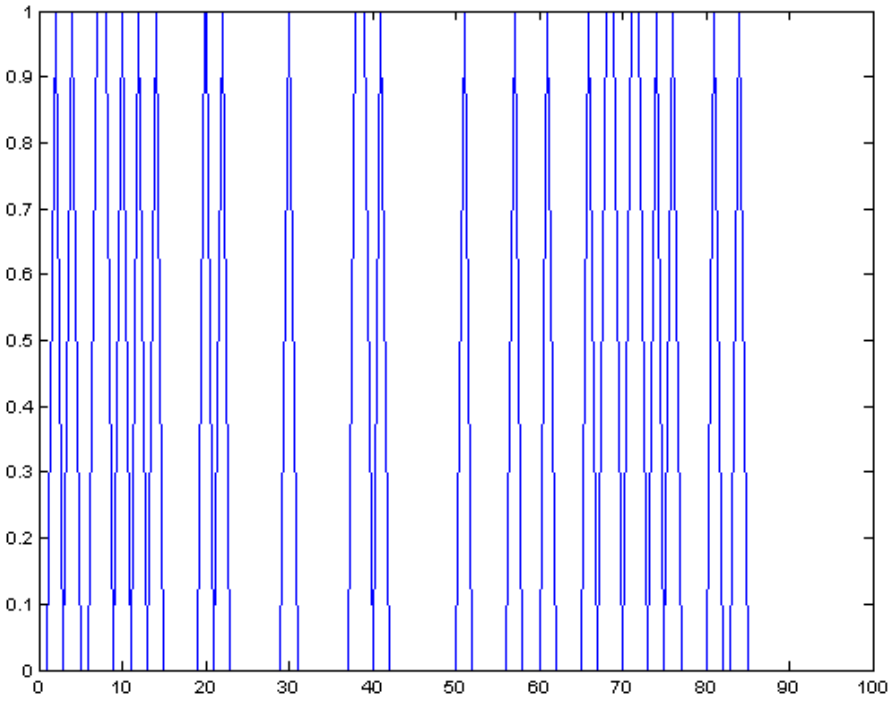



(b) Wykres błędu (uczone)

Rysunek 21: Wykresy błędu zestawu danych `dane8d_a`

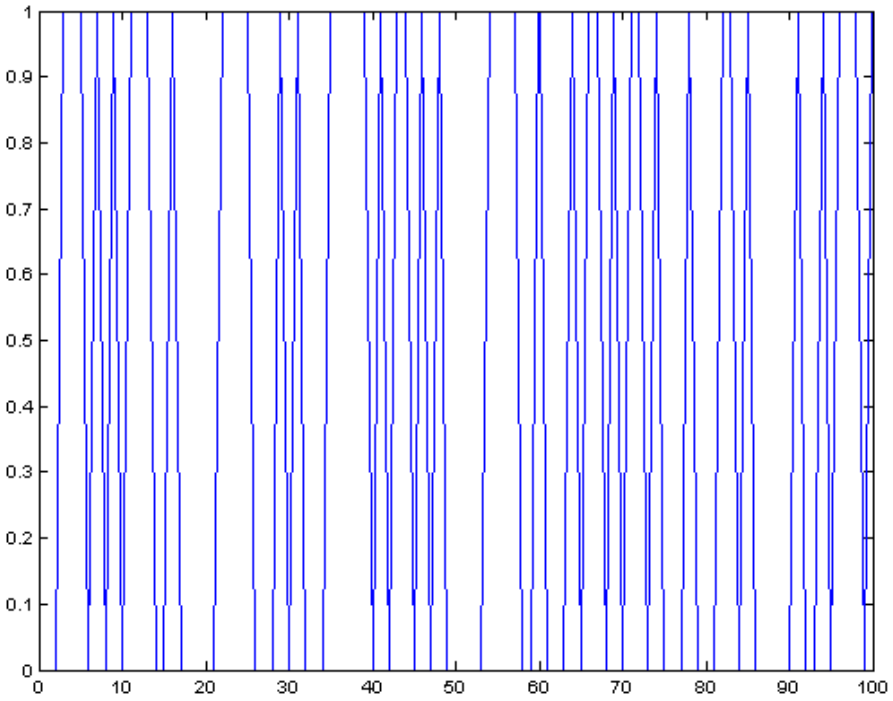



(a) Wykres błędu (nie uczone)

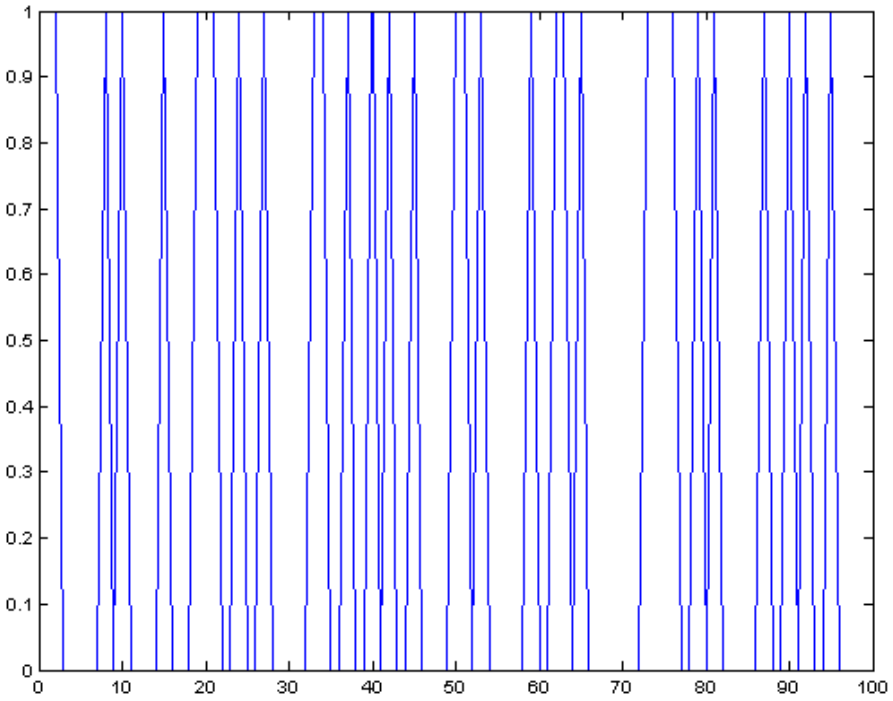



(b) Wykres błędu (uczone)

Rysunek 22: Wykresy błędu zestawu danych `dane8d_2`

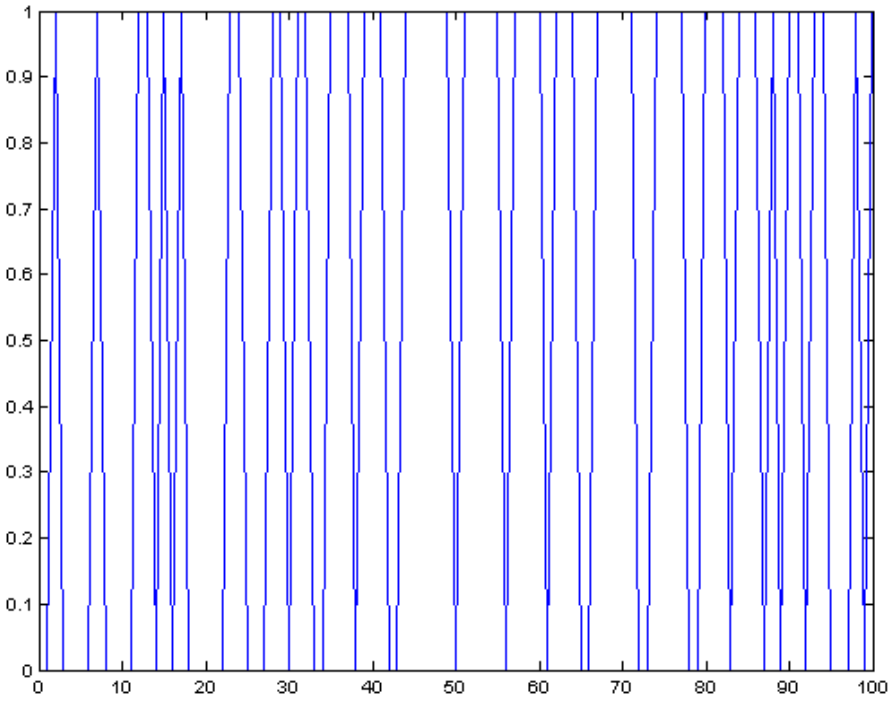



(a) Wykres błędu (nie uczone)

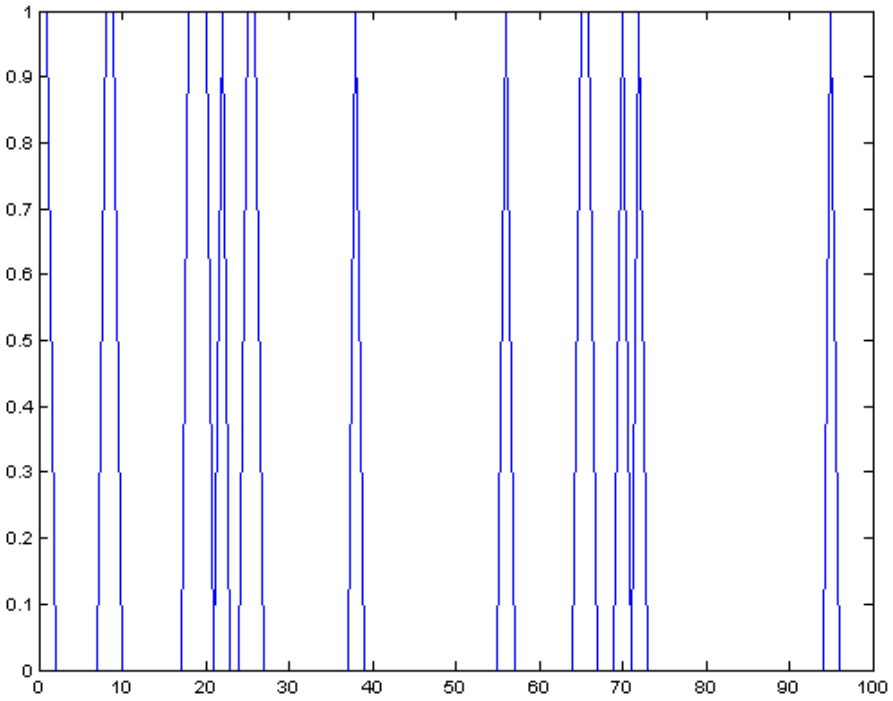



(b) Wykres błędu (uczone)

Rysunek 23: Wykresy błędu zestawu danych `dane8d_3`




(a) Wykres błędu (nie uczone)




(b) Wykres błędu (uczone)

Rysunek 24: Wykresy błędu zestawu danych `dane8d_1`

# 3 Opracowanie własnej uczącą neuron z wykorzystaniem reguły delta

## 3.1 Opis funkcji `delta`

Zrealizowany algorytm opiera się na neuronie według wzoru 1 i jest widoczny w listingu 7. Dane wejściowe $x_0, x_1, \cdots, x_n$ są wczytane zmienną `in`.

Neuron posiada na wyjściu wartość $y = 1$ albo $y = 0$ kiedy funkcja aktywacyjna $\varphi$ jest większa lub równa albo mniejsza od wartości $\theta$. Oczekiwane wartości wyjściowe są wczytane zmienną `out`. Uczone wartości wyjściowe są oddane zmienną `learned`.

$$\begin{array}{rcll} y & = & \begin{cases} 1 & \varphi \geq \theta \\ 0 & \varphi < \theta \end{cases} & \text{gdzie} \\ \\ \theta & = & 0 & \\ \varphi & = & w_0 x_0 + w_1 x_1 + \cdots + w_n x_n + w_{n+1} b & \\ b & = & 1 & \end{array} \tag{1}$$

Wartości początkowe wag $W_0 = (w_0, w_1, \cdots, w_n)$ są wybrane przypadkowo i leżą między $[0, 1]$ (linia 20 listingu 7). W każdej iteracji epoki uczenia są korygowane wartości wag według wzoru 2 (linia 55 listingu 7). Zmienna $0 > \eta \geq 1$ stanowi parametr uczący i jego wartość została w tym przypadku wybrana jako $\eta = 0.2$.

$\delta$ jest różnicą między uczonymi wartościami $t$ danej epoki i wartościami wejściowymi $y$ (linia 49 listingu 7). Uczone dane $t$ w bierzącej epoce są wyliczane za pomocą aktualnej sumy wag i zastosowania wzoru 1 (linie 36-46 listingu 7).

$$\begin{array}{rcll} W_i & = & W_{i-1} + \Delta W & \text{gdzie} \\ \\ \Delta W & = & \eta \; \delta \; x & \\ \\ \eta & = & 0.2 & \\ \delta & = & t - y & \\ x & = & (x_0, x_1, \cdots, x_n, b) & \end{array} \tag{2}$$

Iteracyjnie wagi $W$ w każdej epoce są korygowane aż do momentu kiedy $t = y$ lub kiedy aktualna epoka jest równa maksymalnej epoce danej jako parametr `epochs` (linia 26/59 listingu 7).

Następnie pomocnicza funkcja została zaimplementowana aby ułatwić współprace z plikami. Kod jest widoczny w listingu 6.

Kod został napisany w środowisku Octave 3.0.5 pod Linux ze względu na fakt że to jest środowisko Open Source i za tym o wiele łatwej dostępne od pakietu Matlab.

```
function load_delta(prefix, epochs)
    we = load([prefix '_i.txt'])';
    wy = load([prefix '_o.txt'])';
```

```
    [learned, weights, epochs_learned, success] = delta(we, wy, epochs);

    figure(1)
    plot(abs(learned - wy));
    weights
    epochs_learned

    if success
        printf("SUCCESS :-)\n");
    else
        printf("FAILURE :-(\n");
    end
endfunction
```

Listing 6: `load_delta.m`: Funkcja pomocnicza dla reguły delta

| Dane (prefiks) | Wagi | Epoki | Udane uczenie |
|---|---|---|---|
| `percep` | 0.15562 0.95492 -3.27467 | 15 | Tak |
| `dane_a` | 11.6389 -8.2880 -13.2825 | 48 | Tak |
| `dane_1` | 10.2720 -8.7994 -15.0422 | N/A | Nie |
| `dane_2` | 10.9469 -5.8266 -13.2175 | N/A | Nie |
| `dane_3` | 10.4504 -5.2809 -17.7526 | N/A | Nie |
| `dane3d_a` | 7.2781 -4.3285 -11.2311 32.8859 | 122 | Tak |
| `dane3d_1` | 6.4942 -6.7073 -10.0775 45.8484 | N/A | Nie |
| `dane3d_2` | 6.9544 -4.0883 -11.7865 27.7291 | N/A | Nie |
| `dane3d_3` | 8.3543 -3.3688 -12.0120 38.4670 | N/A | Nie |
| `dane8d_a` | 10.4484 12.2477 11.1905 12.3418 11.5381 13.3448 10.0944 9.6301 -448.6094 | 3995 | Tak |
| `dane8d_1` | 9.7304 5.4700 7.8741 10.3601 7.2305 7.1548 9.8148 4.8561 -342.2336 | N/A | Nie |
| `dane8d_2` | 5.7690 6.5841 9.0588 13.3850 6.2696 9.8061 3.6560 11.1488 -361.6779 | N/A | Nie |
| `dane8d_3` | 6.9226 7.9210 4.8611 8.3132 9.8582 4.9881 11.4248 11.2508 -333.1923 | N/A | Nie |

Tablica 3: Wyniki badan własną implementacją reguły delta

Można zauważyć w badaniach własną funkcją delta że ilość epok (iteracji algorytmu) zwiększa się z większą ilością parametrów.

## 3.2 Kod funkcji `delta`

```
function [learned,weights,epochs_learned,success]=delta(in,out,epochs)
    % debug input parameters
    % in
    % out
    % epochs

    % specify learning parameter
    eta = 0.2

    % specify the bias. for our case, only 1 and 0 makes sense
    bias = 1

    % specify the threshold (theta)
    % in our case 0 makes since since we have boolean functions
    threshold = 0

    % initialize empty weight vector
    % we have as many weights as rows in "in" + 1
    % the additional entry is the weight for the bias
    weights = rand(1, size(in, 1)+1);

    % the learned variable has the same dimensions as the expected out
        variable
    learned = zeros(size(out,1), size(out,2));

    % loop through all epochs and try to learn
    for current_epoch=1:epochs
        % iterate through all input values in the current epoch
        for i=1:size(in,2)
            % transpose the current values
            cur_in_transposed = in(:,i)';

            % add the bias
            cur_in_and_bias = [cur_in_transposed, bias];

            % calculate the weighted inputs (including the bias)
            weighted_ins = weights .* cur_in_and_bias;

            % sum the weighted inputs
            % this sum is the current output value
            sum_weighted_ins = sum(weighted_ins);

            if (sum_weighted_ins >= threshold)
                learned(i) = 1;
            else
                learned(i) = 0;
            end

            % error in the current epoch
            err = out(i) - learned(i);

            % the delta of the weights
            delta_weight = eta * err * cur_in_and_bias;

            % updates the current weight values with the calculated delta
            weights = weights + delta_weight;
```

```
56          end
57
58          % if the learned vector equals the expected vector, then we are
                finished with learning
59          if (learned == out)
60              break;
61          end
62      end
63
64      epochs_learned = current_epoch;
65
66      if (learned == out)
67          success = true
68      else
69          success = false
70      end
71  end
```

Listing 7: `delta.m`: Funkcja ucząca neuron regułą delta