

Projekt Perceptronu 2

5. Wyświetlono wyniki przeliczone przez perceptron:

W tym projekcie zamiast własnej klasy perceptronu oraz SLP zastosowano gotową klasę z biblioteki

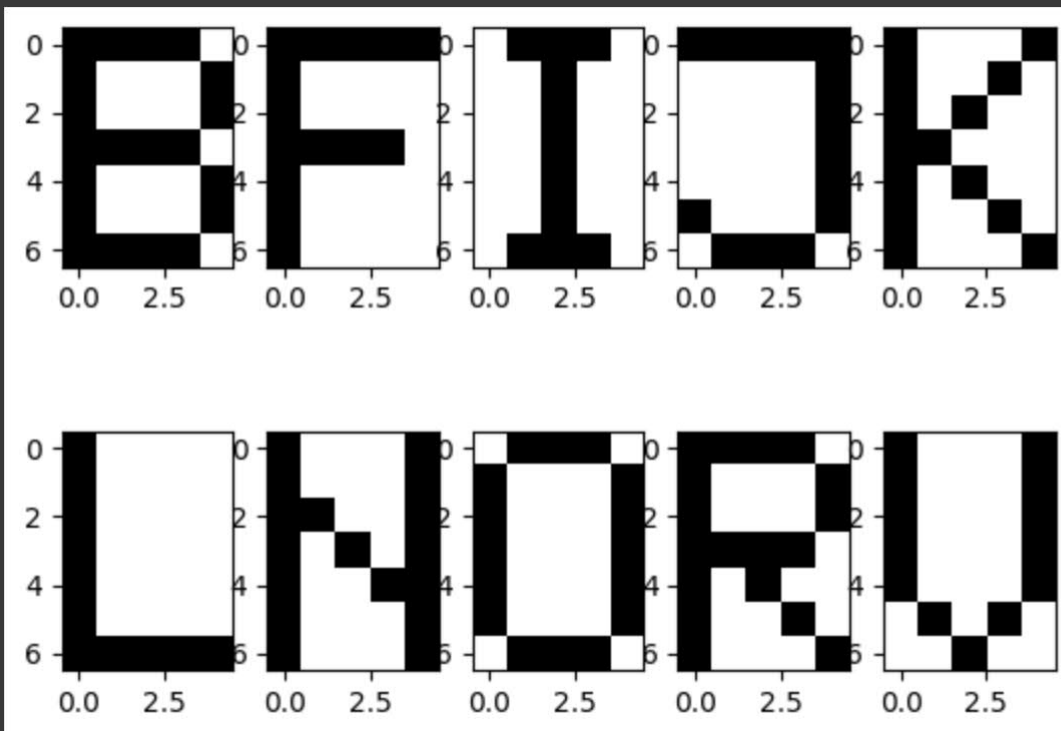
1. Pierwsza część projektu jest bardzo podobna do poprzedniego poprzedniego. Plik z danymi zostaje wpisany i obsługiwany w ten sam sposób. Tak samo działa też metoda `show()`, ale dla kontekstu pozostawię jej zrzuty ekranu. Inny jest natomiast zbiór `y`, ponieważ klasa z biblioteki przyjmuje jednowymiarową tablicę.

Aby więc `y` zawierało unikalne wyniki wpisałem indeksy liter: `y = [1, 5, 8, 9, 10, 11, 13, 14, 17, 21]`

Tablica `X` wygląda jak w poprzednim przykładzie.

2. Przy pomocy nowoutworzonego obiektu klasy `Perceptron` wyświetlono wczytane dane:

```
[8] net.show(X)
```



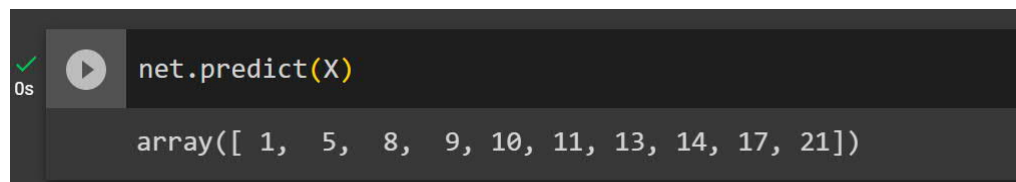
3. W odróżnieniu od ręcznie napisanej klasy `Perceptron`, klasa z biblioteki nie ma ręcznie ustawionej ilości generacji, ale można ograniczyć maksymalną ilość przy pomocy zmiennej `max_iter`, toteż ustawiona została na tę samą wartość co ilość generacji w poprzedniej wersji projektu(10):

```
net.fit(X,y)
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/linear_model/_stochastic_gradient.py:702: ConvergenceWarning: Maximum number of iteration reached before  
warnings.warn(  
    Perceptron  
    Perceptron(eta=0.05, max_iter=10, random_state=1)
```

Wyświetla się ostrzeżenie, w związku z tym, że obliczenia zakończyły się przed osiągnięciem konwergencji. Program zaleca zwiększyć ilość iteracji, jednak dla dobrego porównania z poprzednią wersją pozostawię je w takiej formie.

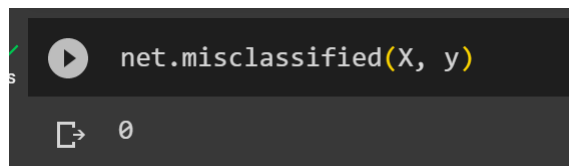
4. Wyświetlono wyniki przeliczone przez perceptron:



```
net.predict(X)
```

```
array([ 1,  5,  8,  9, 10, 11, 13, 14, 17, 21])
```

5. Następnie obliczono ilość różnic z oczekiwanymi wynikami:



```
net.misclassified(X, y)
```

```
0
```

Metoda `misclassified` dodana do klasy `Perceptron` działa podobnie jak w poprzednim projekcie, ale jest dopasowana do innego formatu tablicy `y`. Nadal zlicza różnice między wynikami perceptronu a oczekiwanymi wynikami, z tym, że tutaj ich maksymalna ilość wynosi 10. W powyższym przykładzie nie ma różnic.

W tej klasie prakuje właściwości `errors_` przechowującej ilość błędów w każdej iteracji. Z tego względu, ta tablica zostaje pominięta w tym segmencie, jednak wyniki metody `misclassified()` są wystarczająco informatywne.

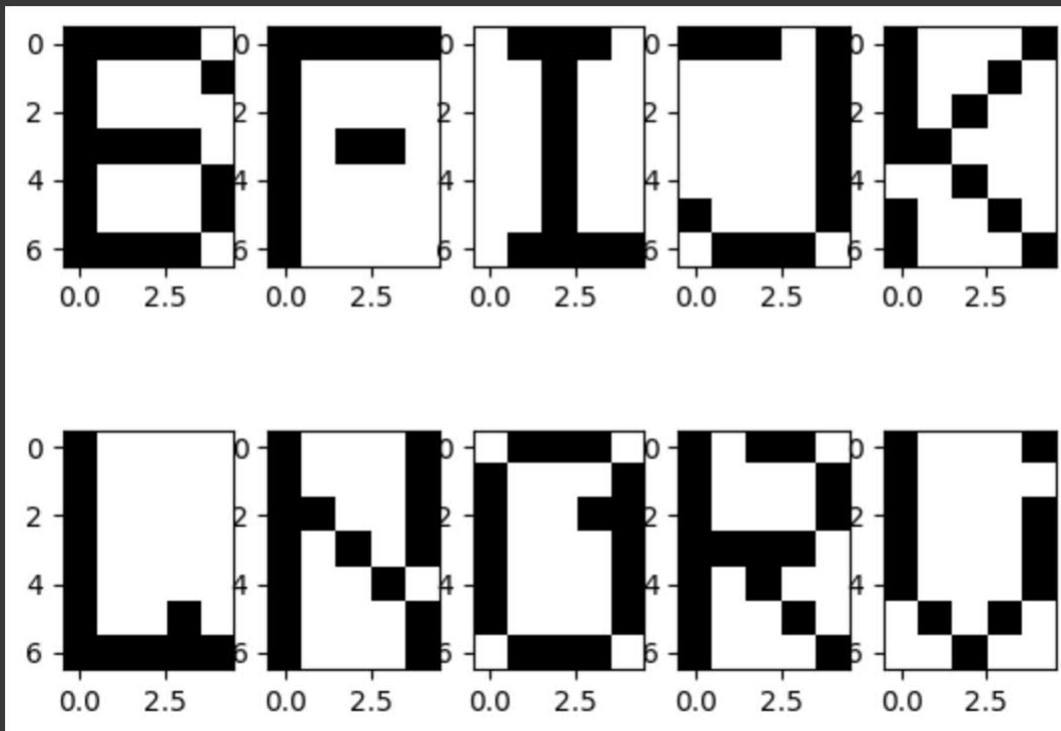
6. Następnie ponownie uszkadzamy dane w ten sam sposób co poprzednio.

7. Obliczamy wynik dla takich danych wejściowych:

8. Wynik dla danych wejściowych uszkodzonych 5%:

✓
2s

```
[13] # 5% damaged:  
net.show(damaged5)
```



✓
0s

```
[13] net.predict(damaged5)  
  
array([ 1,  5,  8,  9, 10, 11, 13, 14, 17, 21])
```

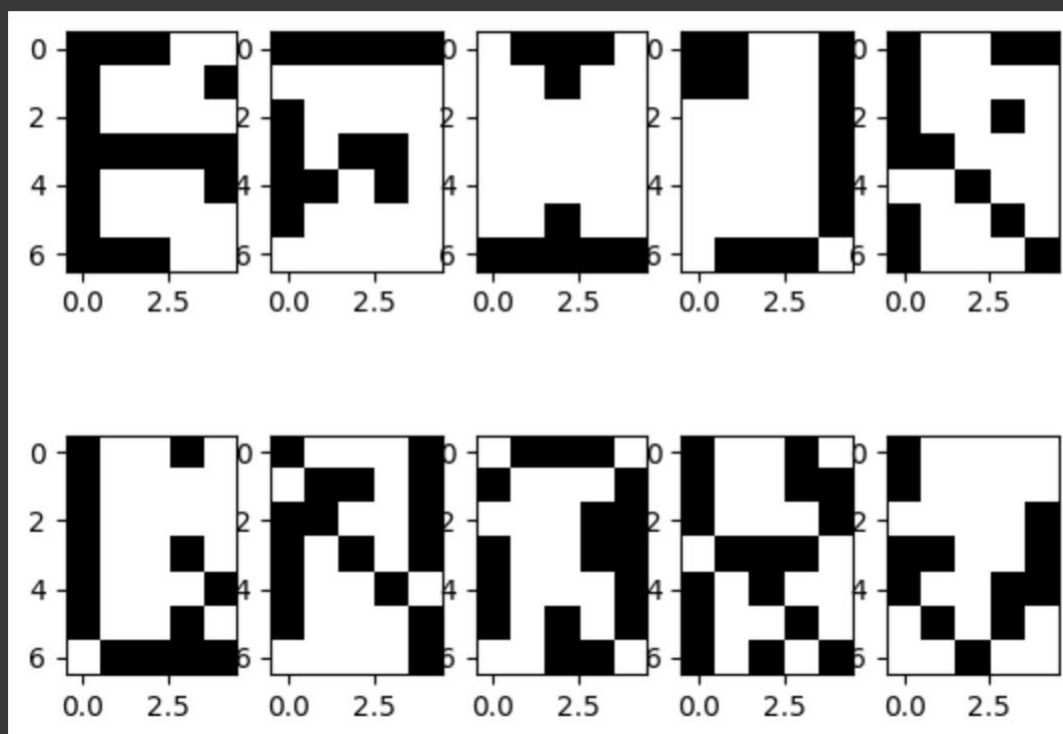
✓
0s

```
[14] net.misclassified(damaged5, y)  
  
0
```

9. Wynik dla danych wejściowych uszkodzonych 15%:

✓
1s

```
[16] # 15% damaged:  
net.show(damaged15)
```



✓
0s

```
[16] net.predict(damaged15)  
  
array([ 5,  5,  8,  9,  5, 11, 13, 14, 17, 21])
```

✓
0s

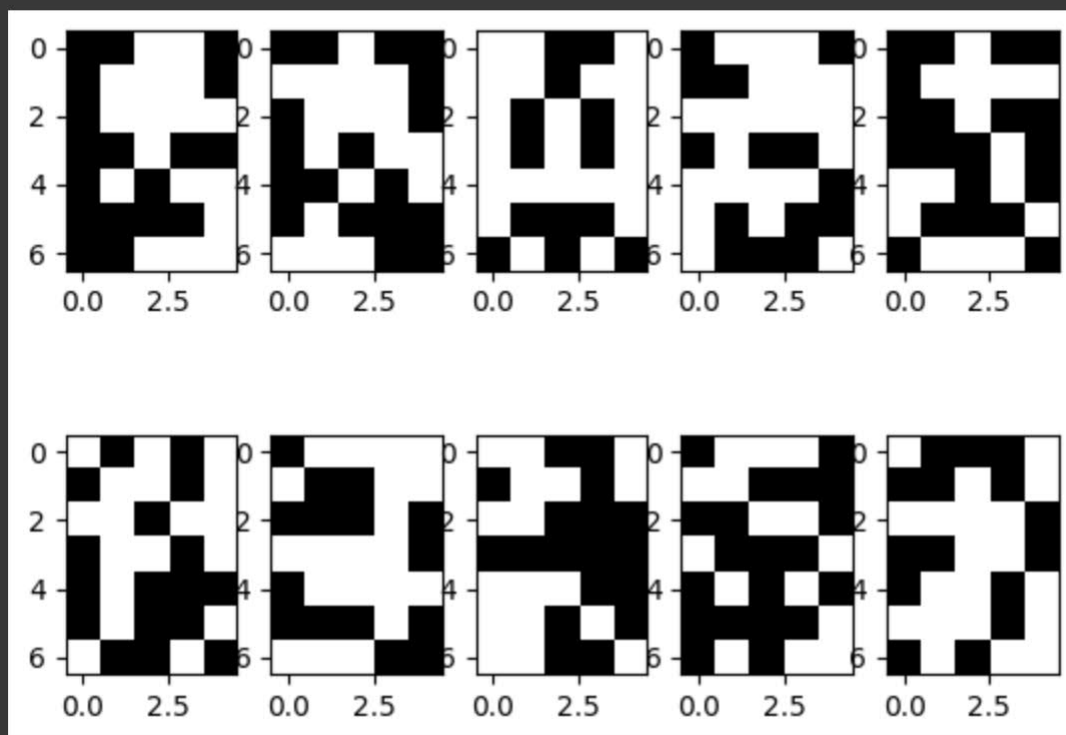
```
[17] net.misclassified(damaged15, y)
```

2

10. Wynik dla danych wejściowych uszkodzonych 40%:

✓
1s

```
[19] # 40% damaged:  
net.show(damaged40)
```



✓
0s

```
[19] net.predict(damaged40)  
  
array([ 5, 17,  5,  5, 21,  8, 13,  1, 21,  5])
```

✓
0s

```
[20] net.misclassified(damaged40, y)  
  
9
```

Jak widać w tym przypadku tylko jedna odpowiedź była poprawna, a reszta, przez uszkodzenie danych, jest błędna.