Sprawozdanie z Metod Inteligencji Obliczeniowej – Lab 2

Małgorzata Makieła

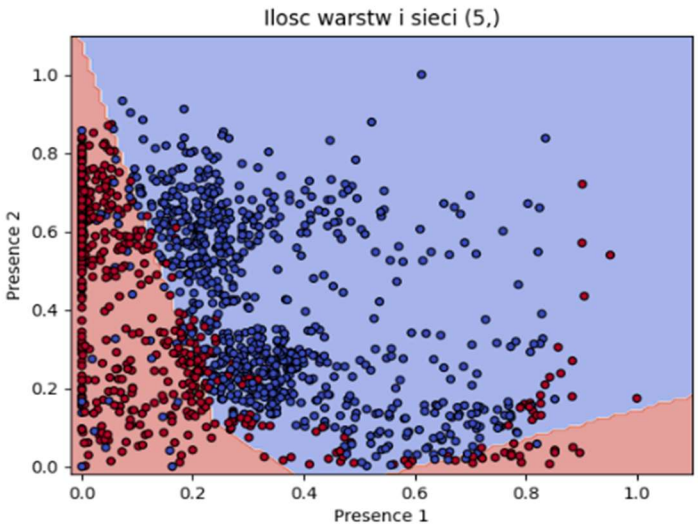
16.03.2024

Zadanie 1.

Dane z pliku *medicine.txt* wczytałam za pomocą pakietu pandas i funkcji *read_csv()*. Dokonałam normalizacji danych funkcją *sklearn.preprocessing.MinMaxScaler().fit_transform()* i podziału na zbiory uczące i testujące funkcją *train_test_split()*. Następnie z użyciem modelu MLPClassifier(*max_iter=2000*) z różną ilością warstw i neuronów w tych warstwach utworzyłam wielowarstwową sieć neuronową. Wyniki z tego zadania przedstawione w postaci wykresów zawierają dwa główne elementy – mapkę konturową z widocznym podziałem na dwie klasy, oraz wszystkie punkty z pliku *medicine.txt,* nałożone na wspomnianą mapkę.

Wyniki:

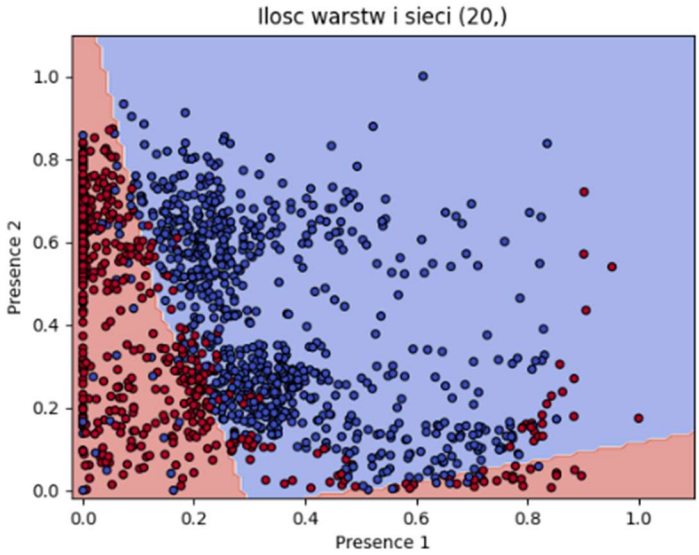
- 1-warstwowe modele
  - hidden_layer_sizes=(5,)




```
Results for the train set:
[[580  40]
 [ 95 361]]
Accuracy =  0.8745353159851301

Results for the test set:
[[145  11]
 [ 19  95]]
Accuracy =  0.8888888888888888
```

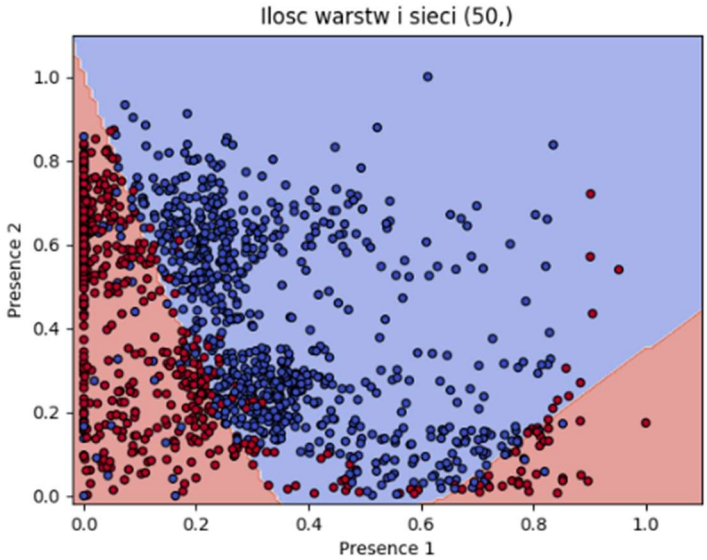
- hidden_layer_sizes=(20,)




```
Results for the train set:
[[571  49]
 [ 94 362]]
Accuracy =  0.8671003717472119

Results for the test set:
[[152   4]
 [ 19  95]]
Accuracy =  0.9148148148148149
```

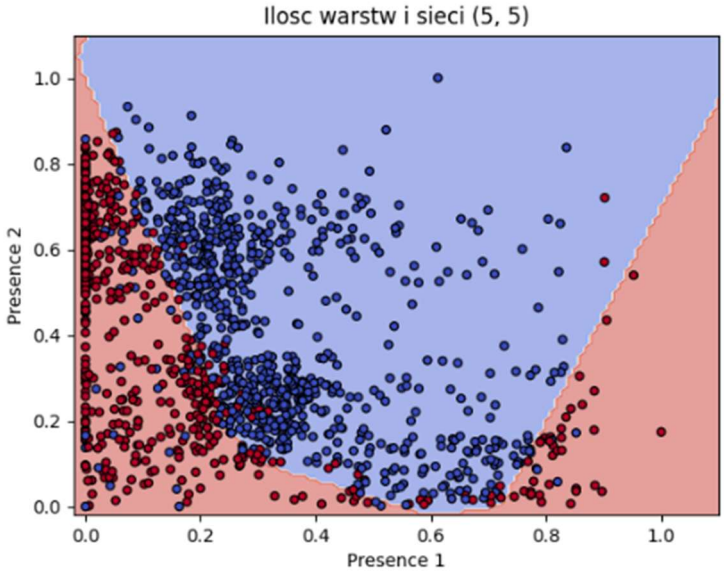
- hidden_layer_sizes = (50,)




```
Results for the train set:
[[565  55]
 [ 65 391]]
Accuracy =  0.8884758364312267

Results for the test set:
[[144  12]
 [ 15  99]]
Accuracy =  0.9
```

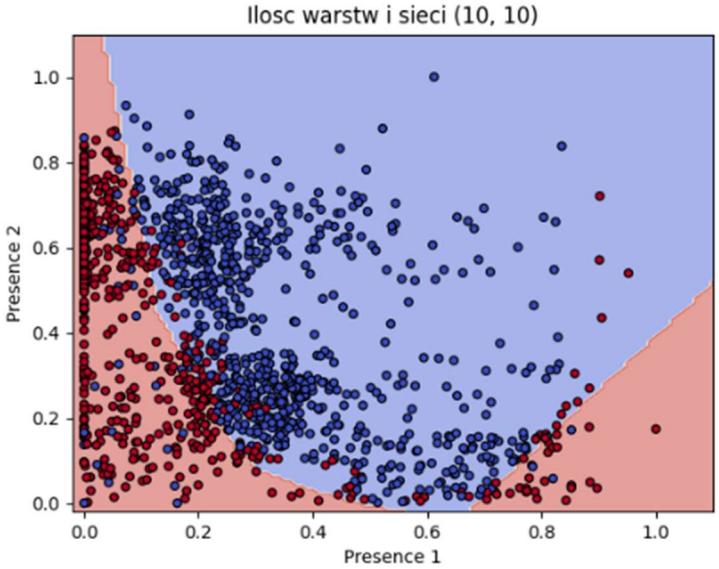
- 2-warstwowe:
  - hidden_layer_sizes = (5,5)




```
Results for the train set:
[[563  57]
 [ 45 411]]
Accuracy =  0.9052044609665427

Results for the test set:
[[139  17]
 [  9 105]]
Accuracy =  0.9037037037037037
```

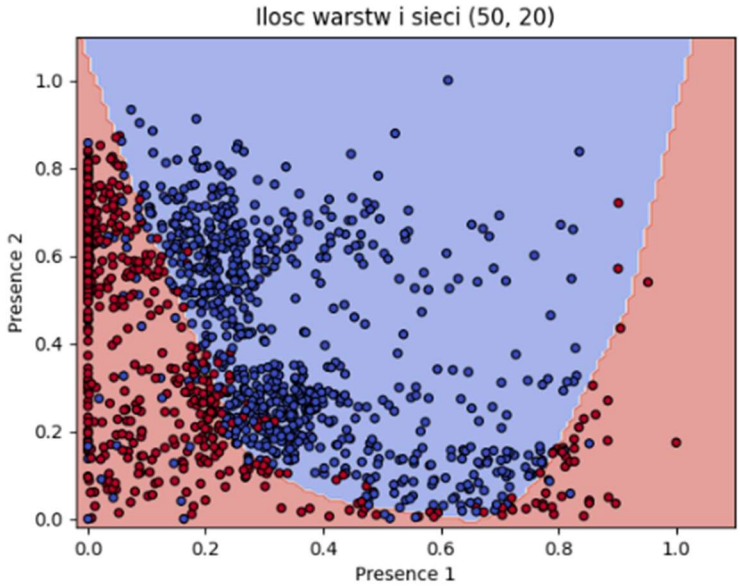
- hidden_layer_sizes = (10,10)




```
Results for the train set:
[[582  38]
 [ 83 373]]
Accuracy =  0.887546468401487

Results for the test set:
[[143  13]
 [ 14 100]]
Accuracy =  0.9
```

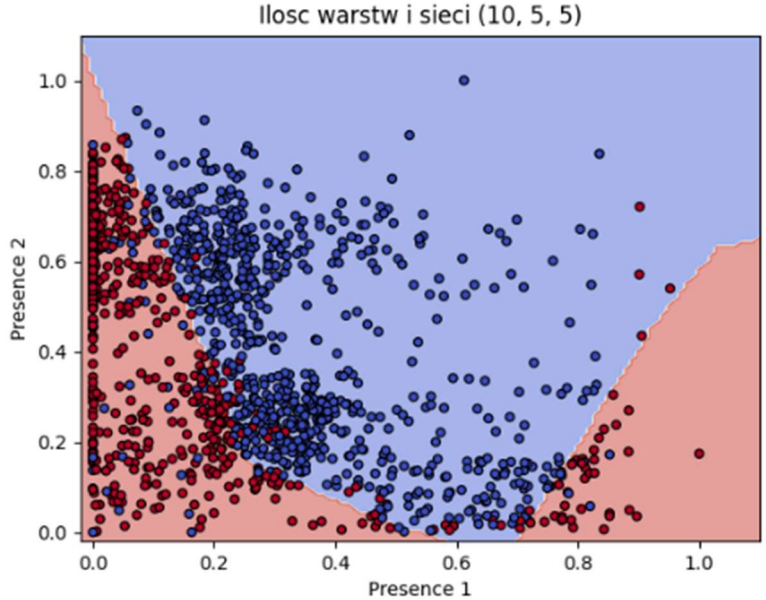
- hidden_layer_sizes = (50,20)




```
Results for the train set:
[[560  60]
 [ 37 419]]
Accuracy =  0.9098513011152416

Results for the test set:
[[141  15]
 [ 10 104]]
Accuracy =  0.9074074074074074
```

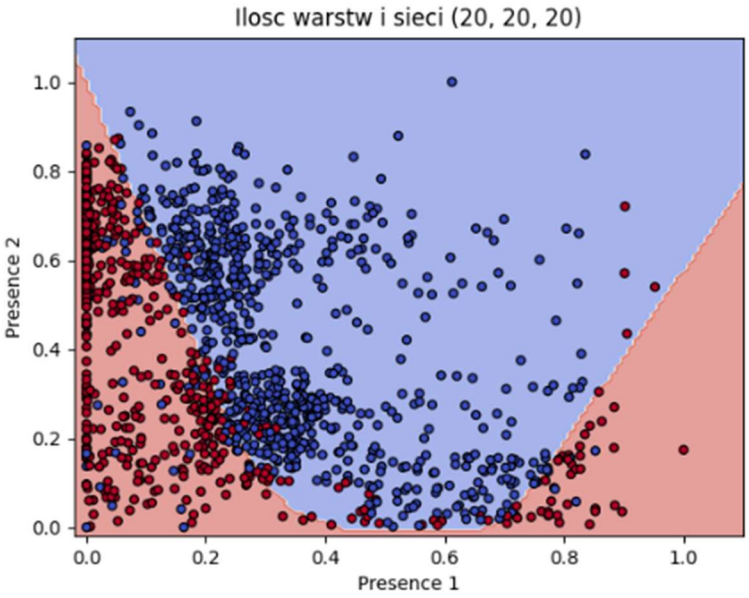
- 3-warstwowe
  - hidden_layer_sizes=(10, 5, 5)




```
Results for the train set:
[[556  64]
 [ 40 416]]
Accuracy =  0.9033457249070632

Results for the test set:
[[148   8]
 [  8 106]]
Accuracy =  0.9407407407407408
```

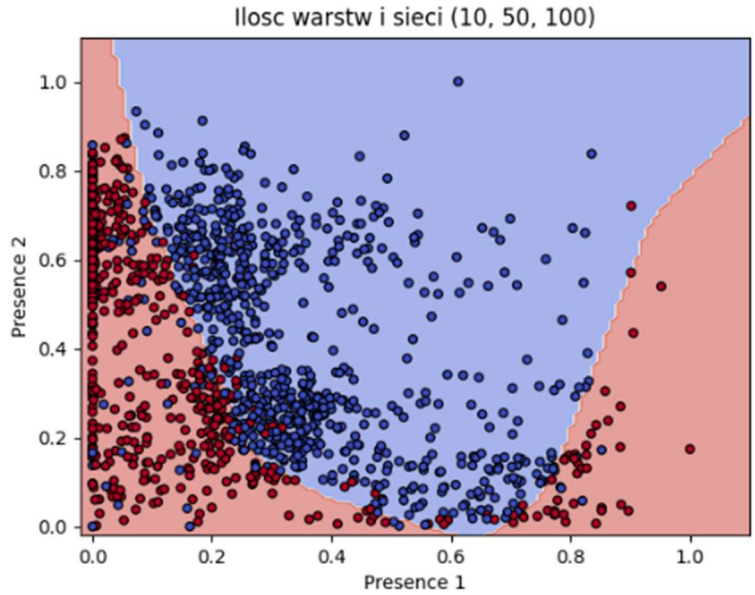
- hidden_layer_sizes=(20, 20, 20)




```
Results for the train set:
[[560  60]
 [ 51 405]]
Accuracy =  0.8968401486988847

Results for the test set:
[[141  15]
 [  8 106]]
Accuracy =  0.9148148148148149
```

- hidden_layer_sizes=(10, 50, 100)




```
Results for the train set:
[[554  66]
 [ 31 425]]
Accuracy =  0.9098513011152416

Results for the test set:
[[137  19]
 [  5 109]]
Accuracy =  0.9111111111111111
```

Wnioski: Właściwie wszystkie zaproponowane powyżej sieci dobrze poradziły sobie z zadaniem, ich skuteczność jest bardzo podobna, utrzymuje się na poziomie około 90%. Najwyższy wynik uzyskany został przy hidden_layer_sizes=(20, 20, 20) dla zbioru testowego – było to 94% skuteczności.

Zadanie 2.

Dane podzieliłam na zbiory trenujące i testujące w proporcji 80:20 z zachowaniem proporcji klas, i utworzyłam model
model = MLPClassifier(hidden_layer_sizes=(10, 50, 100)) z trzema warstwami po 10, 50, I 100 neuronów kolejno. Uzyskany wynik dokładności modelu jest bardzo wysoki – dla kilku wywołań programu utrzymywał się na poziomie ~ 95%.

```
[[35  0  0  0  0  1  0  0  0  0]
 [ 0 36  0  0  1  0  0  0  0  0]
 [ 0  0 35  0  0  0  0  0  0  0]
 [ 0  0  0 34  0  1  0  0  0  2]
 [ 0  0  0  0 36  0  0  0  0  0]
 [ 0  0  0  0  0 36  0  0  0  0]
 [ 0  0  0  0  0  1 35  0  0  0]
 [ 0  0  0  0  1  0  0 35  0  0]
 [ 0  0  0  0  0  2  0  0 33  0]
 [ 0  0  0  2  0  0  0  0  0 34]]
Accuracy =  0.9694444444444444
```

Zadanie 3.

Dla wszystkich poniższych przykładów zmiana ilości warstw na 2, i ilości na neuronów na mniejszą, około (10, 5) powodowała minimalne tylko zmiany, dlatego używam wszędzie hidden_layer_sizes = (10, 20, 50)

1) Model = MLPClassifier(
   hidden_layer_sizes = (10, 20, 50),
   max_iter=20000,
   activation='relu',
   solver='sgd',
   learning_rate='invscaling')
   Dla takiego zestawu danych macierz pomyłek najbardziej mnie zaskoczyła – wynik był bardzo słaby i dane źle dopasowane. Bardzo podobna macierz wychodzi również, gdy użyjemy funkcji aktywacji ‘tanh’ i logistic – natomiast dla activation=’identity’ dane nieco wyniki nieco się poprawiają, skuteczność wynosi ~ 70%

```
[[ 1  5  0 30  0  0  0  0  0  0]
 [ 0 23  0  3  0  0  1  0  0  9]
 [ 0  5  8 13  0  4  3  0  0  2]
 [ 6 13  1 16  0  0  0  0  0  1]
 [ 0 14  0 17  0  2  0  0  0  3]
 [ 1  2  4 19  0  5  2  0  0  4]
 [ 0 13  0 18  0  0  0  0  0  5]
 [ 0  5  8  7  0  1  9  0  0  6]
 [ 0 27  0  4  0  1  0  0  0  3]
 [ 1 21  0  5  0  0  2  0  1  6]]
Accuracy =  0.1638888888888889
```

2) Jeśli w powyższym zestawie danych zmienimy tylko solver na ‘adam’ – model działa już bardzo dobrze, niezależnie od wyboru funkcji aktywującej i współczynnika uczenia.
   Model = MLPClassifier(
   hidden_layer_sizes = (10, 20, 50),
   max_iter=20000,
   activation='relu',
   solver='adam',
   learning_rate='invscaling')

```
[[36  0  0  0  0  0  0  0  0  0]
 [ 0 35  1  0  0  0  1  0  0  0]
 [ 0  0 35  0  0  0  0  0  0  0]
 [ 0  0  1 34  0  0  0  0  2  0]
 [ 0  0  0  0 35  0  1  0  0  0]
 [ 0  0  0  0  0 34  0  0  0  2]
 [ 1  0  0  0  0  1 34  0  0  0]
 [ 0  0  0  0  0  1  0 34  0  1]
 [ 0  1  0  2  0  1  1  0 30  0]
 [ 0  0  0  0  0  1  0  1  0 34]]
Accuracy =  0.9472222222222222
```

3) Również biorąc pierwszy zestaw danych, ale zmieniając tylko współczynnik uczenia (learning_rate) na ‘adaptive’ lub ‘constant’, model działa bardzo dobrze.
Model = MLPClassifier(
hidden_layer_sizes = (10, 20, 50),
max_iter=20000,
activation='relu',
solver='sgd',
learning_rate='adaptive')

```
[[36  0  0  0  0  0  0  0  0  0]
 [ 0 35  0  0  0  0  0  0  0  1]
 [ 0  0 35  0  0  0  0  0  0  0]
 [ 0  0  0 37  0  0  0  0  0  0]
 [ 0  2  0  0 33  0  0  0  1  0]
 [ 0  0  0  0  0 36  0  1  0  0]
 [ 1  0  0  0  0  0 35  0  0  0]
 [ 0  0  0  0  0  0  0 35  0  1]
 [ 0  0  0  1  0  0  1  0 33  0]
 [ 0  0  0  0  0  0  0  1  0 35]]
Accuracy =  0.9722222222222222
```

4) Ciekawy wynik wychodził też przy użyciu funkcji aktywacyjnej ‘logistic’ w takim zestawie danych: model = MLPClassifier(
hidden_layer_sizes=(10, 20, 50),
max_iter=20000,
activation='logistic',
solver='sgd',
learning_rate='invscaling'). Dla kilku kolejnych wywołań programu dane zawsze przypisywało jednej klasie (zmieniającej się w kolejnych wywołaniach). Nie pomagała tu zmiana współczynnika uczenia. Dopiero zmiana solvera na ‘adam’ lub ‘lbfgs’ powodowała powrót dobrych wyników z accuracy ~ 90%.

```
[[ 0 36  0  0  0  0  0  0  0  0]
 [ 0 36  0  0  0  0  0  0  0  0]
 [ 0 35  0  0  0  0  0  0  0  0]
 [ 0 37  0  0  0  0  0  0  0  0]
 [ 0 36  0  0  0  0  0  0  0  0]
 [ 0 37  0  0  0  0  0  0  0  0]
 [ 0 36  0  0  0  0  0  0  0  0]
 [ 0 36  0  0  0  0  0  0  0  0]
 [ 0 35  0  0  0  0  0  0  0  0]
 [ 0 36  0  0  0  0  0  0  0  0]]
Accuracy =  0.1
```

Zadanie 4.

Do pobrania danych użyłam funkcji *ucimlrepo.fetch_ucirepo(id=110)* , a następnie dane ‘Y’ zawierające przydział danych do klas zamieniłam ze stringów na liczby przy użyciu LabelEncoder(). Podzieliłam zbiór na trenujący i testowy w proporcji 80:20.

1. model = MLPClassifier(
hidden_layer_sizes=(10, 20, 50),
max_iter=20000,
activation='identity',
solver='lbfgs',
learning_rate='adaptive')
jest to jedna z najszybszych wersji trenowania modelu. Bardzo podobnie działa dla learning_rate ‘constant’ i ‘invscaling’

```
Training time: 3.4575746059417725 seconds
Results for the train set:
[[256   0   0   0   2   2  29  80   1   0]
 [  0   4   0   0   0   0   0   0   0   0]
 [  2   0  15   4   3   0   3   1   0   0]
 [  0   0   6  26   2   0   1   0   0   0]
 [  6   0   2   5  18   3   4   3   0   0]
 [  7   0   0   0   2 110   1  11   0   0]
 [ 54   0   1   1   6   6 120   6   1   0]
 [147   0   1   0   2   9  22 162   0   0]
 [  5   0   0   0   0   0   1   0  10   0]
 [ 11   0   1   0   1   7   3   1   0   0]]
Accuracy =  0.607413647851727

Results for the test set:
[[71  0  0  0  1  0  6 15  0  0]
 [ 0  1  0  0  0  0  0  0  0  0]
 [ 2  0  4  0  0  0  1  0  0  0]
 [ 0  0  2  3  1  2  1  0  0  0]
 [ 1  0  1  1  4  1  2  0  0  0]
 [ 5  0  0  0  0 24  0  3  0  0]
 [21  0  0  0  0  2 24  2  0  0]
 [35  0  0  0  0  4  7 40  0  0]
 [ 1  0  0  0  1  0  2  0  0  0]
 [ 2  0  1  1  0  0  1  1  0  0]]
Accuracy =  0.5757575757575758
```

```
Training time: 44.99068284034729 seconds
Results for the train set:
[[370   0   0   0   0   0   0   0   0   0]
 [  0   4   0   0   0   0   0   0   0   0]
 [  0   0  28   0   0   0   0   0   0   0]
 [  0   0   0  35   0   0   0   0   0   0]
 [  0   0   0   0  41   0   0   0   0   0]
 [  0   0   0   0   0 131   0   0   0   0]
 [  0   0   0   0   0   0 195   0   0   0]
 [  0   0   0   0   0   0   0 343   0   0]
 [  0   0   0   0   0   0   0   0  16   0]
 [  0   0   0   0   0   0   0   0   0  24]]
Accuracy =  1.0

Results for the test set:
[[41  0  1  0  3  2  9 36  0  1]
 [ 1  0  0  0  0  0  0  0  0  0]
 [ 1  0  3  2  1  0  0  0  0  0]
 [ 0  0  0  8  1  0  0  0  0  0]
 [ 2  0  0  1  2  0  4  1  0  0]
 [ 3  0  0  0  2 22  3  2  0  0]
 [ 7  0  0  2  1  0 26 10  0  3]
 [32  0  0  0  0  5 10 39  0  0]
 [ 1  0  0  0  0  0  0  0  3  0]
 [ 3  0  0  1  0  2  0  0  0  0]]
Accuracy =  0.48484848484848486
```

2.Ciekawe rozwiązanie uzyskałam dla danych:
model = MLPClassifier(
hidden_layer_sizes=(10, 20, 50),
max_iter=20000,
activation='tanh',
solver='lbfgs',
learning_rate='invscaling')

To jedyny zestaw dla którego trafność dla zbioru trenującego wyniosła 100%, jest to również zestaw który wykonywał się 10 razy dłużej niż większość innych.

3. model = MLPClassifier(
hidden_layer_sizes=(10, 20),
max_iter=20000,
activation='relu',
solver='adam',
learning_rate='invscaling')

Najwyższa dokładność jaką udało mi się uzyskać na zbiorze testującym (trenującym też, poza przykładem wyżej)

```
Training time: 6.578698635101318 seconds
Results for the train set:
[[220   0   0   0   1   4  30 115   0   0]
 [  0   3   0   0   1   0   0   0   0   0]
 [  4   0  16   4   2   0   1   1   0   0]
 [  0   1   4  27   2   0   1   0   0   0]
 [  2   0   2   5  19   3   4   6   0   0]
 [  4   0   0   0   1 112   2  12   0   0]
 [ 48   0   0   1   4   6 117  18   1   0]
 [ 91   0   1   0   3  14  19 215   0   0]
 [  6   0   0   0   0   0   3   0   7   0]
 [ 10   0   2   0   1   6   2   3   0   0]]
Accuracy =  0.620050547598989

Results for the test set:
[[68  0  0  0  1  0  3 21  0  0]
 [ 0  0  0  0  0  0  0  1  0  0]
 [ 0  0  4  1  0  0  2  0  0  0]
 [ 0  1  0  7  0  1  0  0  0  0]
 [ 2  0  1  3  2  1  1  0  0  0]
 [ 3  0  0  0  0 26  0  3  0  0]
 [16  0  1  0  1  1 27  3  0  0]
 [28  0  0  0  0  0  3 55  0  0]
 [ 1  0  1  0  0  0  0  0  2  0]
 [ 3  0  0  0  0  1  0  2  0  0]]
Accuracy =  0.6430976430976431
```

4. Połączenie solvera ‘sgd’ z learning_rate ‘invscaling’ (i z dowolną funkcją aktywacji) doprowadzało do najgorszych wyników, z trafnością na poziomie ~ 30%. Natomiast larning_rate ‘constant’ oraz ‘adaptive’ radziły sobie podobnie, trafność wynosiła ~ 57%

Podsumowując, trafność na poziomie 50% jest dość przeciętna dla tego zbioru. Da się osiągnąć trafność bliżej 60% na kilka sposobów. Jednak wyżej niż 64% nie udało mi się osiągnąć, więc na taką skalę 50% nie wypada źle.