Ocena skuteczności modeli klasyfikacyjnych w zakresie rozpoznawania spamu

Spis treści

Opis projektu

Celem projektu jest znalezienie najlepszej metody pod względem weryfikacji czy dana wiadomość jest spamem lub czy nią nie jest. Do projektu została zastosowana zbiór danych z podane linku: https://archive.ics.uci.edu/dataset/94/spambase. Zbiór posiada 4601 próbek oraz 58 cech.

Pierwsze 48 kolumn w zbiorze zwracają postać liczbową jak często występuje dane słowo w wiadomości. np.

word_freq_make = 0.75

Oznacza to że słowo **'make'** stanowi 0.75% całej wiadomości.

Tak samo dla pozostałych kolumn od 49 do 54 kolumny, istnieje ten sam system zliczania znaków, natomiast w tych kolumnach pojawiają się znaki specjalne.

Przykład:

char_freq_$ = 0.1

Czyli że znak **'$'** stanowi 0.1% całej wiadomości.

Przygotowanie danych

Na początku przeglądnąłem dane ręcznie w sprawdzeniu czy nie ma tam cech tekstowych oraz brakujących wartości. W dokumentacji dotyczącej tego zbioru danych zostało napisane że posiada brakujące dane, natomiast obok każdej cechy wszędzie było widoczne że nie posiada brakujących danych, dlatego musiałem się upewnić czy faktycznie tak jest.

Początkowo sprawdziłem wartości typu 'null'

```
# sprawdzenie czy nie ma brakujacych danych
missing_values = dataset.isnull().sum()
print("Brakujące dane:\n", missing_values.sum())
```

```
[5 rows x 58 columns]
Brakujące dane:
 0
```

Następnie sprawdziłem czy w całym zbiorze są kolumny o typie 'object' co mogło by oznaczać że posiadają wartość tekstową.

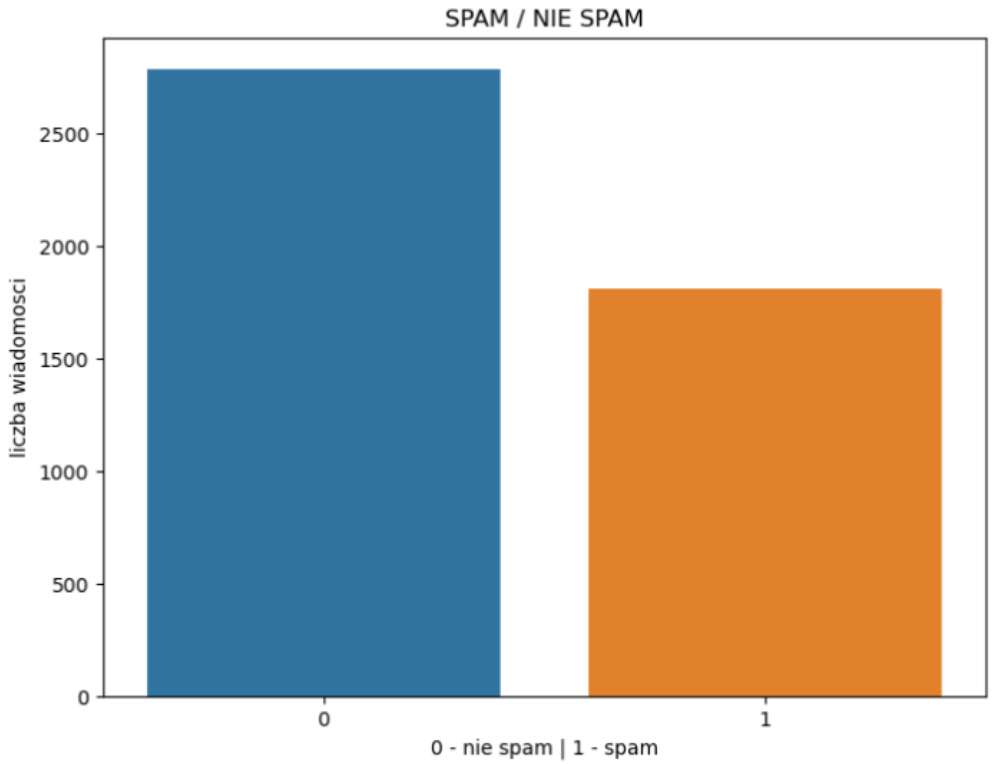
```python
# zliczenie cech o typie 'object'
number_object = (dataset.dtypes == 'object').sum()
print(f"Liczba kolumn z typem object: {number_object}")

if number_object > 0:
    # wyświetlenie kolumn które sa obiektami
    object_columns = dataset.dtypes[dataset.dtypes == 'object']
    print(object_columns.sum())
```

```
Liczba kolumn z typem object: 0
```

Finalnie cały zbiór nie zawierał brakujących danych, oraz nie posiadał typów ‘object’. Z wszystkich danych to 55 kolumn było typem float64 oraz 3 kolumny typu int64.

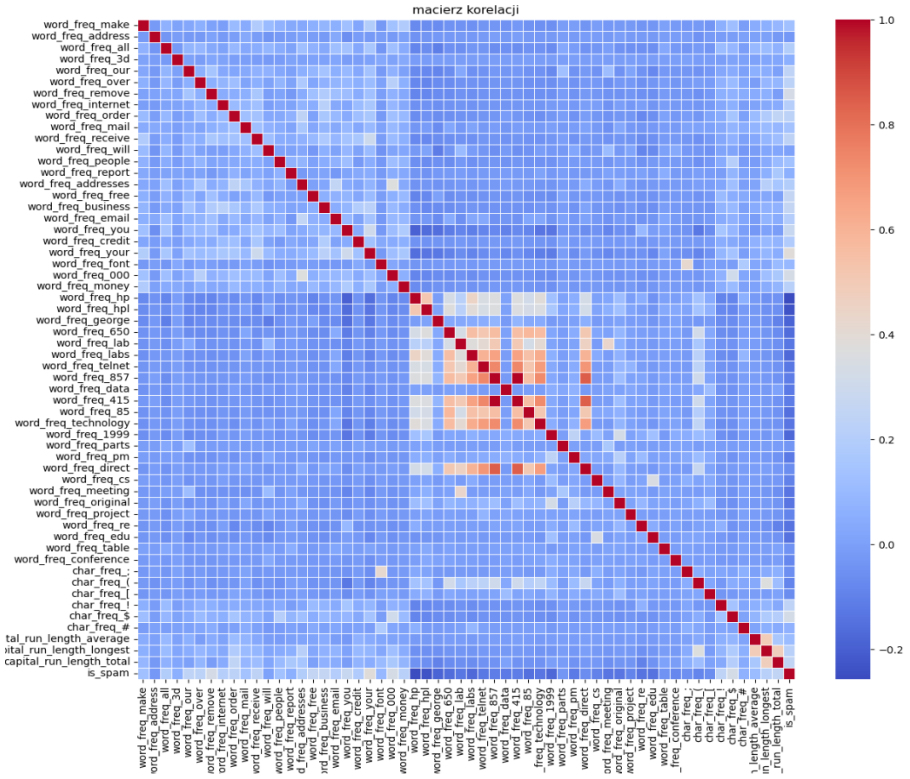
Grupowanie etykiet

Następnie etykiety zostały zgrupowane i policzone aby sprawdzić czy dana etykieta nie odstaje zbyt dużo od drugiej. Ważne jest aby każdej z etykiet było w miarę równo, aby nie było sytuacji że jest proporcja np. 90% to spam a 10% to nie spam, gdyż wtedy będziemy mieć problem np. z weryfikacja wiadomości która nie jest spamem.

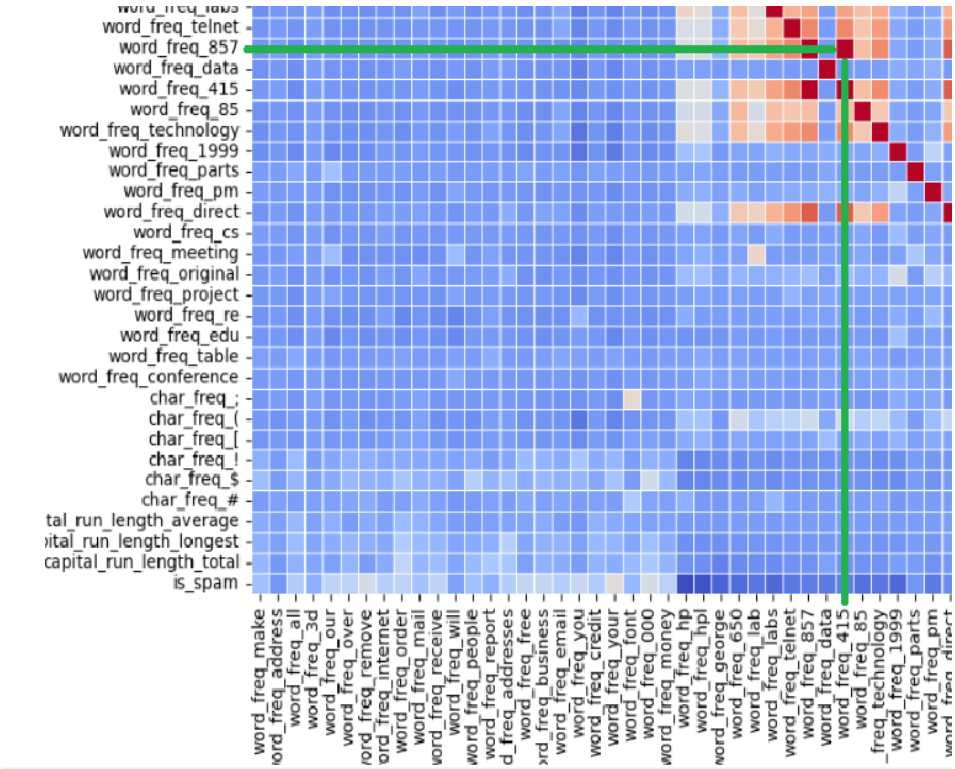


Jak widać dane nie odstają od siebie zbyt dużo.
Nie spam: **2788 (60.60%),** spam: **1813 (39.40%)**

Macierz korelacji



Na wykresie można zobaczyć że kolumna word_freq_857 oraz word_freq_415 są silnie zależne od siebie. Usunięci kolumny word_freq_857 nie powinno wpłynąć na końcowy wynik.



Aby za każdym razem nie szukać ręcznie jaka kolumna mocny koreluje z inną, zrealizowałem metodę, która wypisuje kolumny która mogę zostać usunięte, w zależności od podanego progu.

```
# skanowanie kolumn ktore maja wysoka korelacje
# prog korelacji kiedy dana kolumna moze zostac wyrzucona
threshold = 0.9
columns_to_drop = [column for column in upper.columns if any(upper[column] > threshold)]
print(f"kolumna ktora moze zostac wyrzucona:\n {columns_to_drop}")
```

```
kolumna ktora moze zostac wyrzucona:
 ['word_freq_415']
```

W sekcji z trenowaniem modeli, będę testował czy usunięcie kolumny wpłynie pozytywnie lub negatywnie na trenowanie modelu.

Odchylenie standardowe

Finalnym etapem będzie sprawdzenie odchylenia standardowego, który wykaże nam czy nasze dane nie są zbyt bardzo od siebie oddalone. Jeżeli jakieś kolumny będą posiadać zbyt dużą różnice wtedy można je poddać standaryzacji lub normalizacji.

Poniżej wybrałem próg odchylenia równy 3:

```
# odchylenie standardowe
std_dev = dataset.std()
threshold = 3
std_columns = std_dev[std_dev > threshold].index
print("kolumny z odchyleniem standardowym:\n", std_columns)
```

```
kolumny z odchyleniem standardowym:
 Index(['word_freq_george', 'capital_run_length_average',
       'capital_run_length_longest', 'capital_run_length_total'],
      dtype='object')
```

Następnie kolumny które przekraczają wartość zostają poddane standaryzacji.

```
scaler = StandardScaler()
#scaled_data = scaler.fit_transform(dataset)
dataset[std_columns] = scaler.fit_transform(dataset[std_columns])
```

Przygotowane dane zostaną teraz wykorzystane do trenowania różnych modeli. Skalowanie danych przed podziałem na zbiory testowe, treningowe może prowadzić do wycieku danych. Pod koniec sprawozdania uwzględniłem to i testowałem bez skalowania konkretnych kolumn.

Trenowanie modeli

Do aktualnego projektu zastosowałem nastepujące modele:

XGBClassifier – wmocnienie gradientowe oparte na drzewach decyzyjnych
https://xgboost.readthedocs.io/en/stable/get_started.html

LogisticRegression - regresja logistyczna
https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html

DecisionTreeClassifier - drzewa decyzyjne
https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html

RandomForestClassifier – lasy losowe
https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html

SVC - Support Vector Classifier - klasyfikacja
https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html

KNeighborsClassifier – klasyfikacja na podstawie większości sąsiadów
https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html

GradientBoostingClassifier – wmocnienie gradientowe
https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingClassifier.html

Sieć neuronowa
https://pytorch.org/docs/stable/optim.html
https://pytorch.org/docs/stable/nn.html
https://pytorch.org/docs/stable/torch.html
https://pytorch.org/docs/stable/generated/torch.optim.Adam.html
https://pytorch.org/docs/stable/generated/torch.optim.SGD.html
https://pytorch.org/docs/stable/generated/torch.nn.BCELoss.html

Funkcje aktywacji
https://pytorch.org/docs/stable/generated/torch.nn.ReLU.html
https://pytorch.org/docs/stable/generated/torch.nn.Sigmoid.html
https://pytorch.org/docs/stable/generated/torch.nn.Tanh.html

Do testowania modelu stworzyłem sobie pomocniczą funkcje, która ograniczy powielanie tekstu. Funkcja **train_model**, przyjmuje dwa parametry,

- Pierwszy jest odpowiedzialny za przekazywanie obiektu z modelem,
- Drugi parametr to nazwa użytego modelu
- Funkcja zwraca DataFrame z następującymi kolumnami [‘y_pred', 'accuracy', 'f1', 'roc_score', 'r2']

Metryki, wzory

Y_pred – przewidywana wartość

Accuracy – dokładność modelu

https://scikit-learn.org/stable/modules/model_evaluation.html#accuracy-score

$$\texttt{accuracy}(y, \hat{y}) = \frac{1}{n_{\text{samples}}} \sum_{i=0}^{n_{\text{samples}}-1} 1(\hat{y}_i = y_i)$$

F1 - średnia harmoniczna precyzji i czułości

https://scikit-learn.org/stable/modules/generated/sklearn.metrics.f1_score.html#f1-score

$$\mathrm{F1} = \frac{2 * \mathrm{TP}}{2 * \mathrm{TP} + \mathrm{FP} + \mathrm{FN}}$$

ROC – krzywa ROC

https://scikit-learn.org/stable/modules/generated/sklearn.metrics.roc_auc_score.html

FN - false negative

FP - false positive

TPR - true positive rate

FPR - false positive rate

TPR = TP / (TP + FN) - oś Y

FPR = FP / (FP + TN) - oś X

R2 – jakość dopasowania modelu do danych

https://en.wikipedia.org/wiki/Coefficient_of_determination

R2 = 1 - ((sum(y - y_pred))^2 / (sum(y – y_pred)^2))

$$R^2 = 1 - \frac{SS_{\text{res}}}{SS_{\text{tot}}}$$

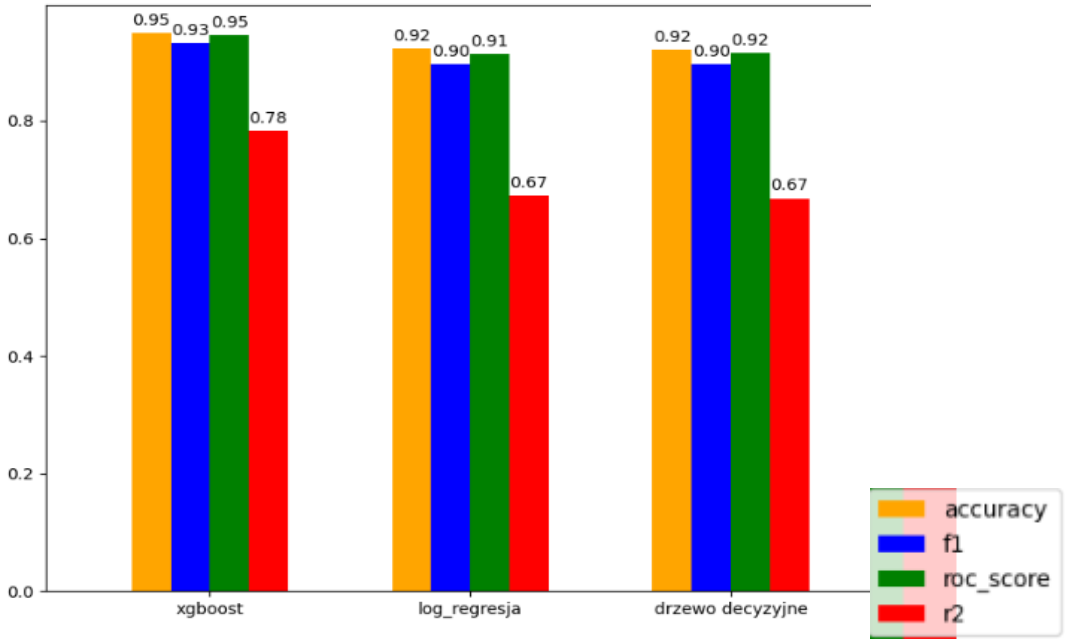
$$SS_{\text{res}} = \sum_i (y_i - f_i)^2 = \sum_i e_i^2 \qquad SS_{\text{tot}} = \sum_i (y_i - \bar{y})^2$$

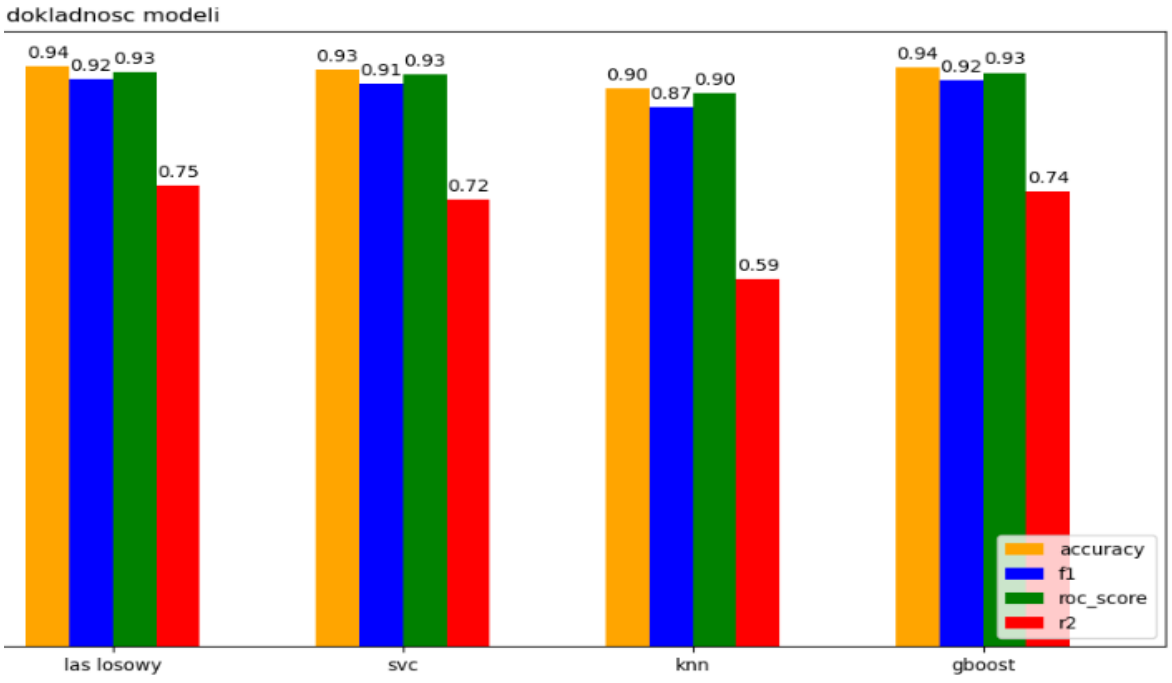
Modelowanie wszystkich kolumn metodą ‘train_test_split’

Zanim zacznę testować modele, muszę podzielić je na część testową i część treningową. Do tego celu zastosowałem metode **train_test_split.** Z biblioteki *sklearn.model_selection*, gdzie dzielę na **80% danych treningowych** oraz **20% danych testowych.**

```
# dzielenie na czesc testowa i czesc do uczenia
X_train, X_test, y_train, y_test = train_test_split( *arrays: X, y, test_size=0.2)
```

Pierwsze trenowanie zostało wykonane na wszystkich kolumnach.



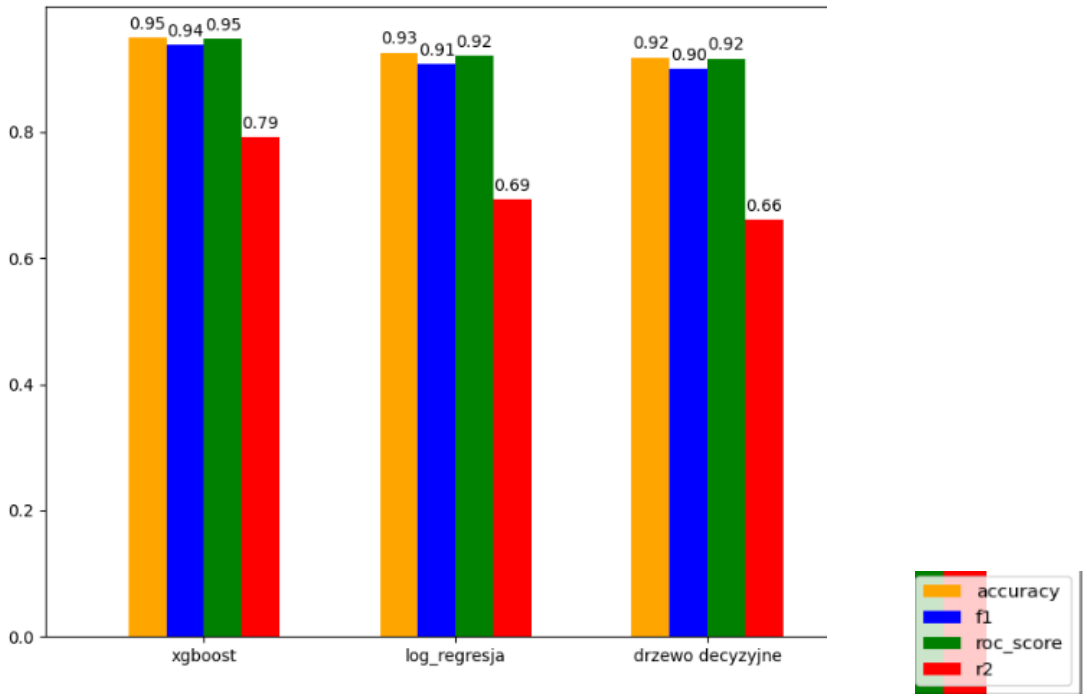


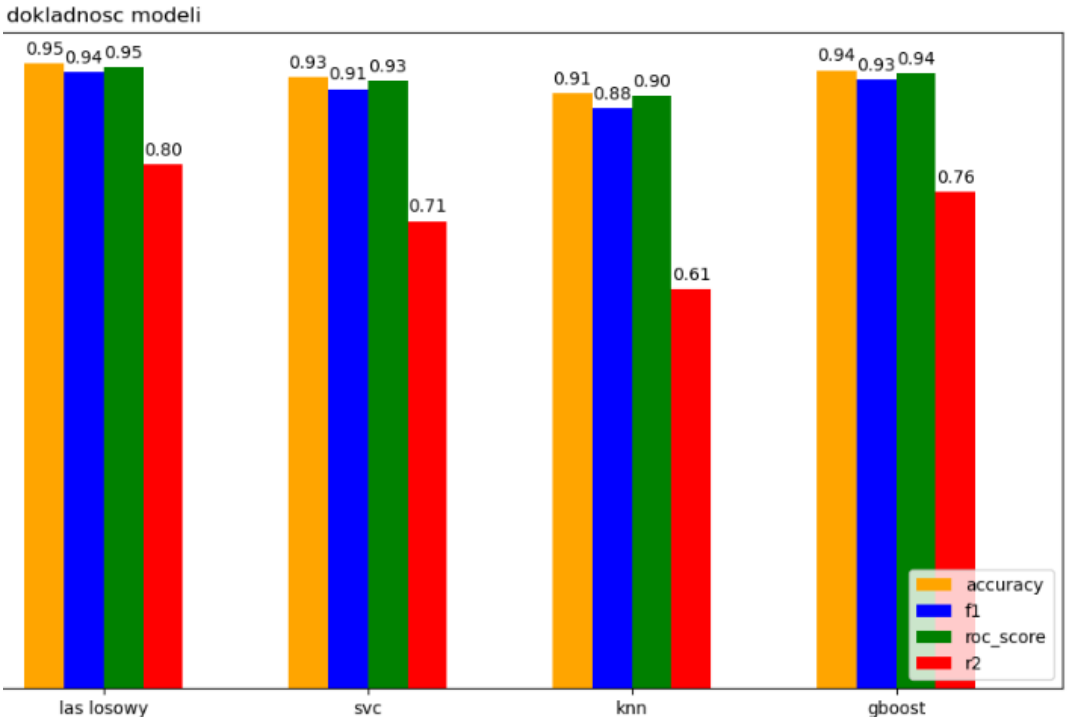
	accuracy	f1	roc_score	r2
xgboost	0.948969	0.932568	0.945021	0.783403
log_regresja	0.92291	0.896047	0.9135	0.6728
drzewo decyzyjne	0.921824	0.896552	0.915942	0.668191
las losowy	0.940282	0.918759	0.930275	0.746535
svc	0.934853	0.912281	0.92645	0.723493
knn	0.904452	0.873199	0.896955	0.594456
gboost	0.938111	0.916545	0.92963	0.737318

Dla wszystkich kolumn oraz podziału zbioru na **80% danych treningowych i 20% testowych**, najlepiej reprezentuje się model XGBoost. Osiągnął on najwyższą dokładność, wynoszącą około 0.95%. Wysoka dokładność pozwoli na poprawne klasyfikowanie instancji. Wynik F1 jest średnią harmoniczną precyzji. Wysoki wynik F1 wskazuje na to, że model XGBoost **robi bardzo mało błędów** wiec też jest pod tym względem najlepszy. Wynik ROC-AUC (0.945021) pokazuje jego zdolność do skutecznego rozróżniania między klasami przy różnych progu klasyfikacji.R2, czyli współczynnik determinacji, mierzy, jak dobrze przyszłe próbki będą prawdopodobnie przewidywane przez model. Wysoka wartość R2 świadczy o tym że model dobrze dopasowuje się do danych.

Modelowanie bez kolumn korelacyjnych

Na tych samych modelach i takich samym podziale danych na część testowa i treningową wytestuje zbiór danych bez kolumny 'word_freq_415'.





	accuracy	f1	roc_score	r2
xgboost	0.950054	0.938005	0.947706	0.792555
log_regresja	0.926167	0.907609	0.921602	0.693343
drzewo decyzyjne	0.918567	0.899866	0.916527	0.661775
las losowy	0.95114	0.93844	0.94645	0.797065
svc	0.93051	0.912807	0.925679	0.711381
knn	0.905537	0.883845	0.902998	0.607659
gboost	0.941368	0.926431	0.936953	0.756478

Dla zbioru danych bez kolumny 'word_freq_415' oraz podziału zbioru na **80% danych treningowych** i **20% danych testowych** nie zauważyłem negatywnego efektu. Modele nadal radzą sobie w podobny sposób. W tej sytuacji **RandomForest** (las losowy) wypada najlepiej. Dokładność tego modelu wynosi w przybliżeniu 0.95, dodatkowo inne metryki tj. krzywa ROC wskazują że model dobrze radzi sobie z klasyfikowaniem klas. Różnica między poprzednim testem gdzie były wszystkie kolumn a między tym gdzie brakuje kolumny która była mocno skorelowana, nie jest duża. Jeżeli usunięcie jeden kolumny ze zbioru nie wpływa negatywnie na wynik końcowy to ten zbiór zostanie finalnie zastosowany do kolejnych testów na innych modelach.

Testowanie zbioru danych w proporcji 40% dane testowe i 60% dane treningowe.

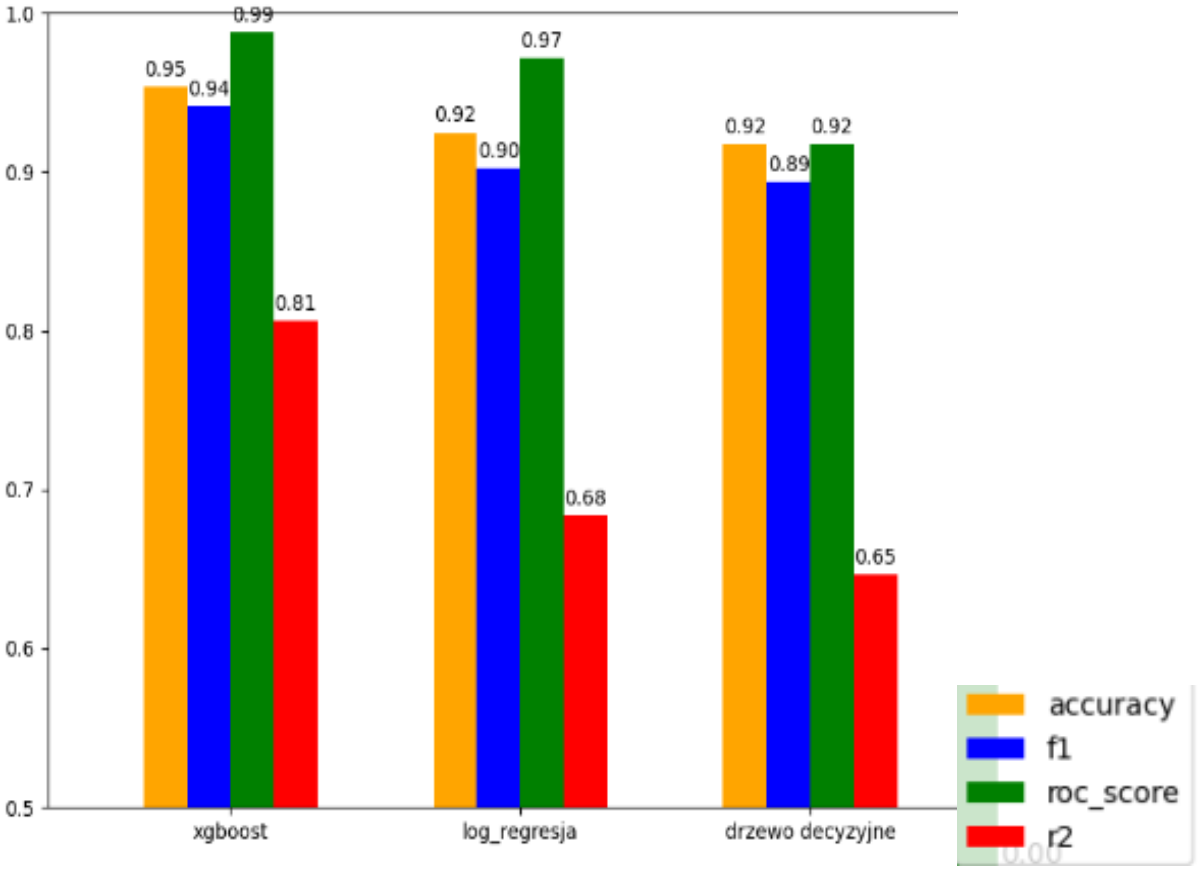
	accuracy	f1	roc_score	r2
xgboost	0.953286	0.940361	0.947928	0.805324
log_regresja	0.916893	0.892932	0.908537	0.653658
drzewo decyzyjne	0.909832	0.884722	0.902428	0.62423
las losowy	0.949484	0.935192	0.943173	0.789478
svc	0.917436	0.892045	0.906721	0.655922
knn	0.900054	0.872576	0.892695	0.583484
gboost	0.939163	0.921238	0.930946	0.746469

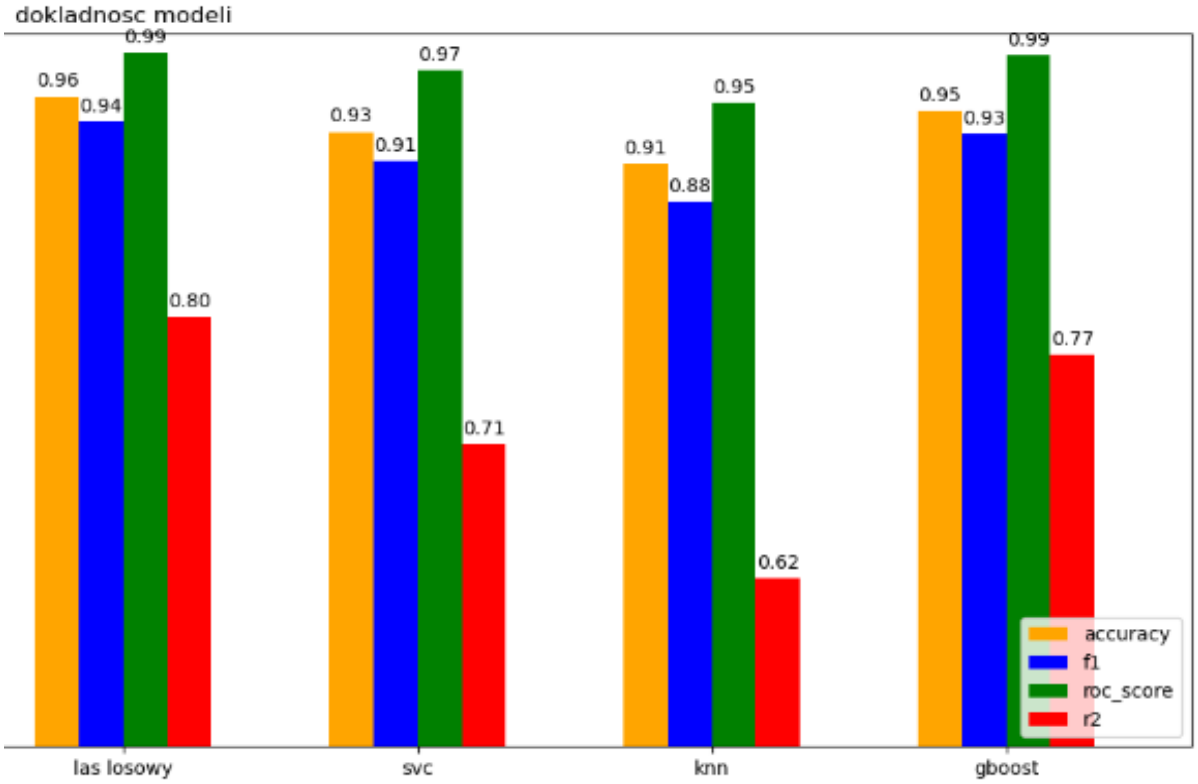
czas trwania: 2.29 sekund

Nie wypada najgorzej, mimo mniejszej ilości danych treningowych model jest w stanie w miarę dobrze przewidywać wartości na podstawie cech.

Modelowanie danych metodą KFold

Metoda KFold która dzieli dany zbiór na podaną X części.
W tym przykładzie podzielę na 5 części.





	accuracy	f1	roc_score	r2
xgboost	0.953922	0.941066	0.987573	0.806638
log_regresja	0.924583	0.902305	0.971744	0.68391
drzewo decyzyjne	0.91741	0.893465	0.917362	0.646532
las losowy	0.956096	0.938483	0.986669	0.801201
svc	0.931319	0.910702	0.974262	0.711983
knn	0.908719	0.882196	0.951407	0.617602
gboost	0.945881	0.930003	0.98526	0.774902

Krzywa ROC wskazuje że różnica między klasami nie jest duża co oznacza że model bardzo dobrze, wydajnie radzi sobie z tym problemem. Dokładność dla **KFold** i **train_test_split** jest bardzo podobna natomiast dzięki krzywej ROC możemy zdecydowanie stwierdzić że uczenie modelu za pomocą KFold'ów będzie bardziej stabilne i mniej wrażliwe na wahania wyników. Podział danych na mniejsze części prowadzi do bardziej efektywnej nauki. Natomiast idzie to z jednym minusem jakim jest czas. Podział danych np. na 10 części może poprawić nasze wyniki, ale kosztem dłuższego czasu trenowania. Nie zawsze jest sens dzielić dane na części. Jeżeli nasz zbiór posiada bardzo dużo próbek (5tys. +), to podział danych na część testową lub treningową może być całkiem sensowy i dawać satysfakcjonujące wyniki.

Testowanie KFold = 10

	accuracy	f1	roc_score	r2
xgboost	0.955228	0.942858	0.98898	0.811916
log_regresja	0.926103	0.903681	0.972095	0.688968
drzewo decyzyjne	0.921539	0.899838	0.924454	0.665393
las losowy	0.954359	0.941928	0.987066	0.803388
svc	0.931103	0.909832	0.974731	0.709584
knn	0.906111	0.878257	0.953608	0.604911
gboost	0.946967	0.931039	0.985457	0.776763

Mimo większego podziału, nie uzyskuje lepszego efektu. Natomiast czas trwania programu wzrósł. Ostatnia próbą będzie wytestowanie dla KFold-3

Testowanie KFold = 3

	accuracy	f1	roc_score	r2
xgboost	0.951097	0.937151	0.98683	0.794813
log_regresja	0.925236	0.902911	0.97069	0.686728
drzewo decyzyjne	0.910021	0.883964	0.903182	0.61758
las losowy	0.949795	0.935308	0.98431	0.788518
svc	0.929798	0.908443	0.972417	0.705588
knn	0.899807	0.870575	0.948449	0.580168
gboost	0.946534	0.930333	0.983972	0.773047

Zarówno jak dla **KFold = 3/5/10** dokładność, krzywa ROC i F1 są bardzo zbliżone natomiast czas modelowania dramatycznie wrasta.

- Czas trwania dla KFold = 3

```
czas trwania: 34.61 sekund
```

- Czas trwania dla KFold = 5

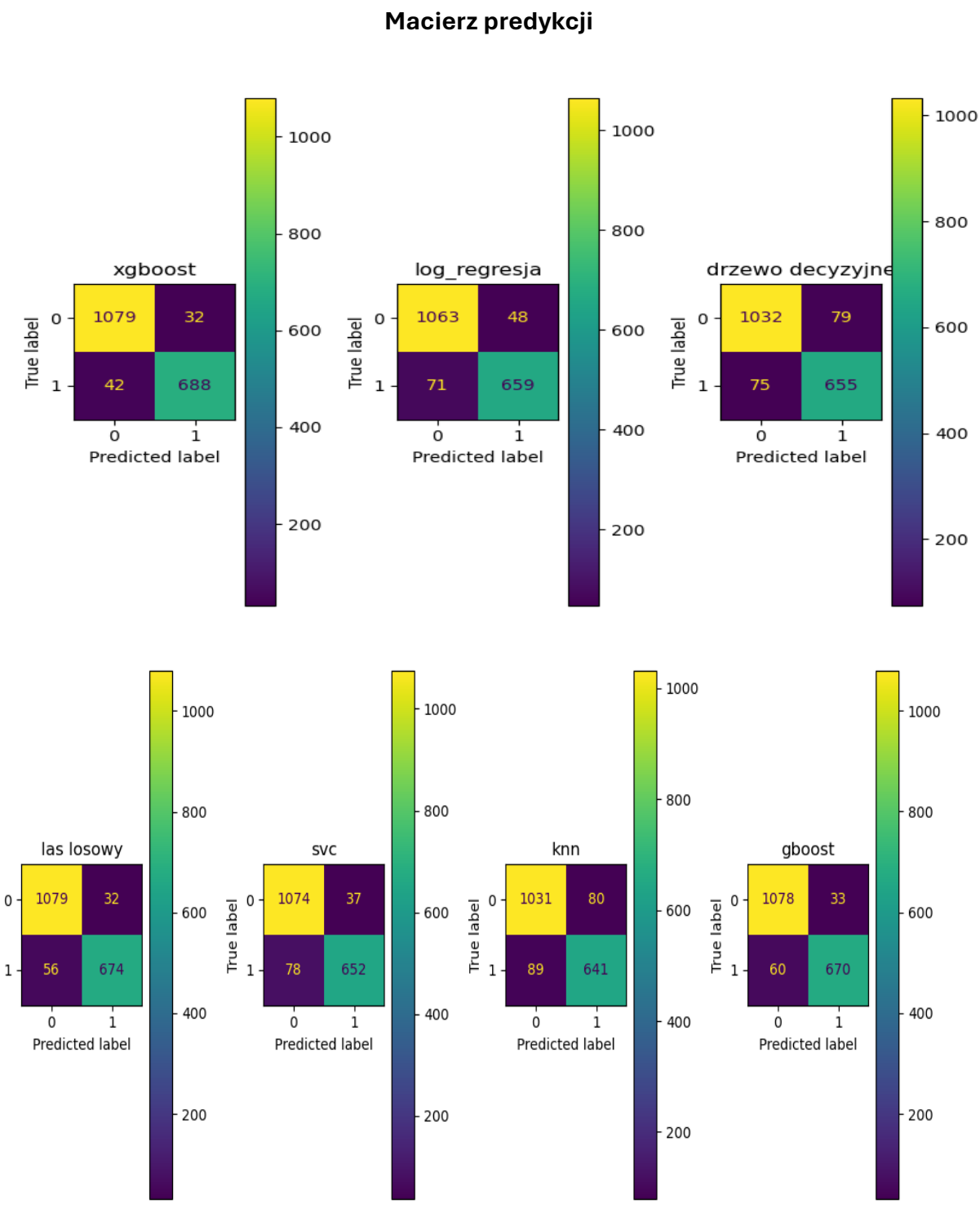
```
czas trwania: 66.97 sekund
```

- Czas trwania dla KFold =10

```
czas trwania: 139.69 sekund
```

Dlatego uważam ze KFold równe 3 jest dobrym modelem. Natomiast porównując to do podziału zbioru za pomoca **train_test_split** to czas przetworzenia wszystkich modeli to około 2-3 sekundy. Uważam że różnica miedzy dokładnością dla podziału danych na **KFold** i **train_test_split** nie jest zbyt duża i że lepiej zastosować metodę na podział zbioru na dane testowe i treningowe.

Macierz predykcji



Sieci neuronowe

Do sieci neuronowych zostanie wykorzystana biblioteka torch. Sieci będzie testowana na danych bez kolumny ‘word_freq_415’. Do trenowania zrealizowałem następującą funkcje (*networks/network_manager.py*) **network_train** z nastepującymi parametrami:

1. Nazwa sieci
2. Model sieci (wszystkie modele znajdują się w folderze (networks/network_templates)
3. X_train – zbiór do trenowania X
4. Y_train - zbiór do trenowania etykiet
5. X_test - zbiór testowy
6. Y_test - zbiór testowy etykiet
7. Liczba epok
8. Optimizer – model optymalizujący wagi
9. Criterion – funkcja błędu modelu

Cała funkcja na podstawie parametrów podany model sieci a następnie testuje. Cała funkcja zwraca DataFrame z następującymi metrykami:

Index(nazwa sieci) = [Accuracy, F1, ROC, R2, Time]

Pierwsze testy realizuje na podziale zbióru na testowy i treningowy

```
X_train, X_test, y_train, y_test = train_test_split( *arrays: X, y, test_size=0.2, random_state=42)
```

Przygotowanie sieci

```
network_model_1 = nn2.NeuralNetwork(input_size, hidden_size_1: 4, hidden_size_2: 4, output_size: 1,
                                    nn.Tanh(), nn.Tanh(), nn.Sigmoid())
network_model_2 = nn2.NeuralNetwork(input_size, hidden_size_1: 8, hidden_size_2: 8, output_size: 1,
                                    nn.Tanh(), nn.Tanh(), nn.Sigmoid())
network_model_3 = nn2.NeuralNetwork(input_size, hidden_size_1: 32, hidden_size_2: 64, output_size: 1,
                                    nn.Tanh(), nn.Tanh(), nn.Sigmoid())

"""
funkcja straty - jak bardzo model myli sie w przewidywaniach
"""
criterion = nn.BCELoss()
"""
algorytm optymalizacji - aktualizuje wagi modelu
poprawa modelu
"""
optimizer = optim.Adam(network_model_1.parameters(), lr=learning_rate)
```

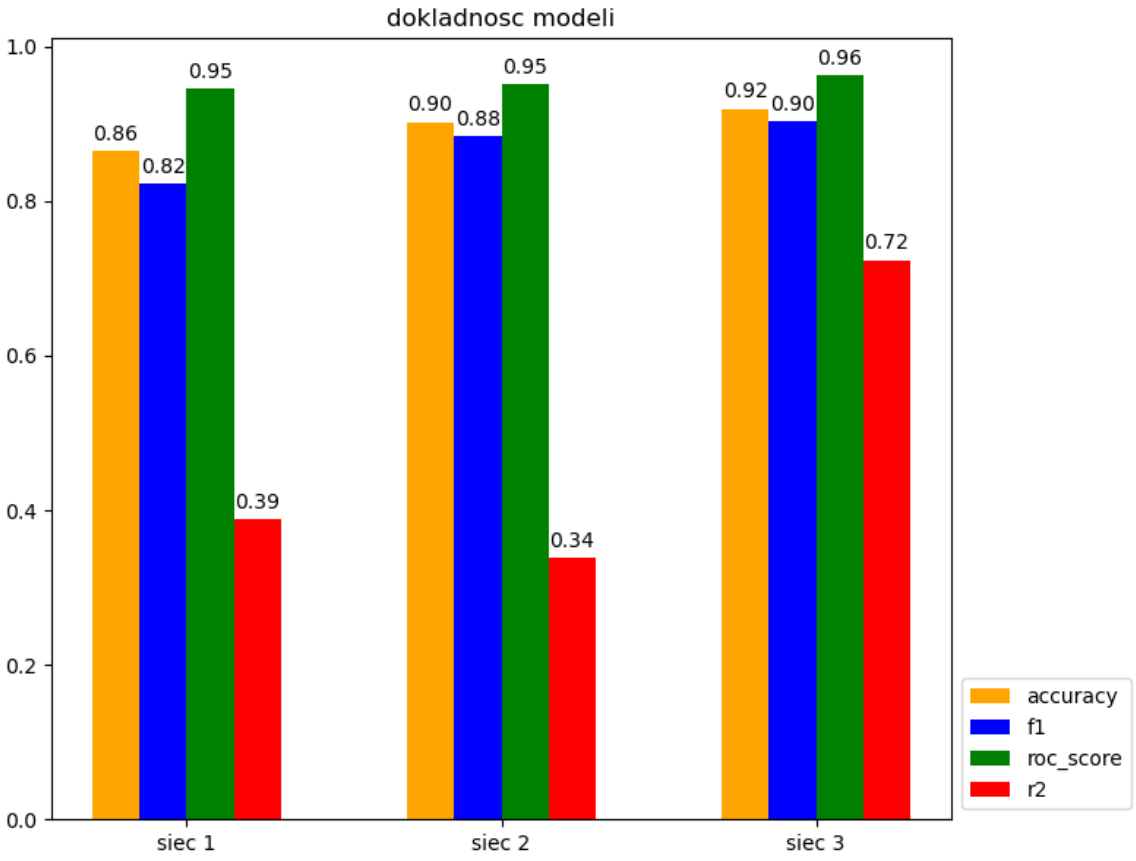
Następnie stosuje dla wszystkich modeli sieci funkcje która opisałem wcześniej **‘network_train’**. Pomoże to mi w szybości testowania wszystkich modeli.

Trenowanie sieci na zbiorze danych treningowym i testowym

```
#[ 1 ]#############################################################################
"""
algorytm optymalizacji - aktualizuje wagi modelu
poprawa modelu
"""
optimizer_1 = optim.Adam(network_model_1.parameters(), lr=learning_rate)
network_1 = network_train( name: 'siec 1', network_model_1, X_train, y_train, X_test, y_test,
                          num_epochs, optimizer_1, criterion)

time_1 = network_1['time']
print(f"czas trwania: {time_1} sekund")

#[ 2 ]#############################################################################
optimizer_2 = optim.Adam(network_model_2.parameters(), lr=learning_rate)
network_2 = network_train( name: 'siec 2', network_model_2, X_train, y_train, X_test, y_test,
                          num_epochs, optimizer_2, criterion)

time_2 = network_2['time']
print(f"czas trwania: {time_2} sekund")
#[ 3 ]#############################################################################
optimizer_3 = optim.Adam(network_model_3.parameters(), lr=learning_rate)
network_3 = network_train( name: 'siec 3', network_model_3, X_train, y_train, X_test, y_test,
                          num_epochs, optimizer_3, criterion)

time_3 = network_3['time']
print(f"czas trwania: {time_3} sekund")
```

Różnica tylko w ilości neuronów w dwóch warstwach.

	Sieć 1	*Sieć 2*	*Sieć 3*
Funkcja strat	BCELoss	BCELoss	BCELoss
Algorytm optymalizacji	Adam	Adam	Adam
Liczba epok	10	10	10
Współczynnik uczenia	0.01	0.01	0.01
Warstwa ukryta 1	4	8	32
Warstwa ukryta 2	4	8	64
Funkcja aktywacji warstwa 1	Tanh	Tanh	Tanh
Funkcja aktywacji warstwa 2	Tanh	Tanh	Tanh
Funkcja aktywacji wyjście	Sigmoid	Sigmoid	Sigmoid
Accuracy	0.86	0.90	0.92
F1	0.82	0.88	0.90
ROC	0.95	0.95	0.96
R2	0.39	0.34	0.72
Czas (sekundy)	0.03	0.02	0.04



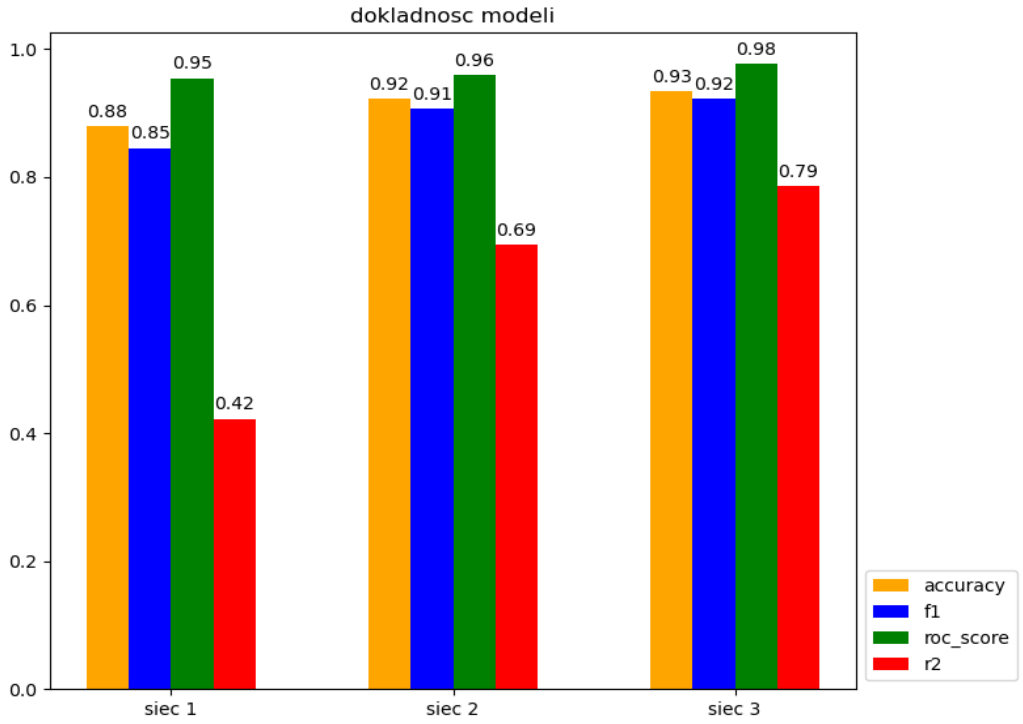
Podsumowanie:

Sieć 3 osiągnęła najwyższą dokładność (0.92), co sugeruje, że większa liczba neuronów w warstwach ukrytych (32 i 64) pozwala na lepsze modelowanie złożonych wzorców w danych. Sieć 2 również osiągnęła wysoką dokładność (0.90), co wskazuje na dobrą równowagę między złożonością a efektywnością. Natomiast sieć 1 uzyskała nieco niższą dokładność (0.86), ale nadal jest to wynik zadowalający w porównaniu do innych sieci. Siec 3 posiada najwyższą wartość dla R2 co wskazuje na to że model bardzo dobrze dopasowuje się do danych. Jednakże nie można tego powiedzieć o sieci 2, gdzie wartość R2 jest niska. Dana nie dopasowują się dobrze do modelu. Dlatego teraz przetestuje te same sieci na różnych epokach o różnych współczynnikach uczenia. Zwiększenie liczy epok wpłynie negatywnie na czas uczenia się sieci, natomiast czy poprawi metryki? Poniżej postaram się przedstawić kilka testów oraz podsumowań do jakich uda mi się dojść.

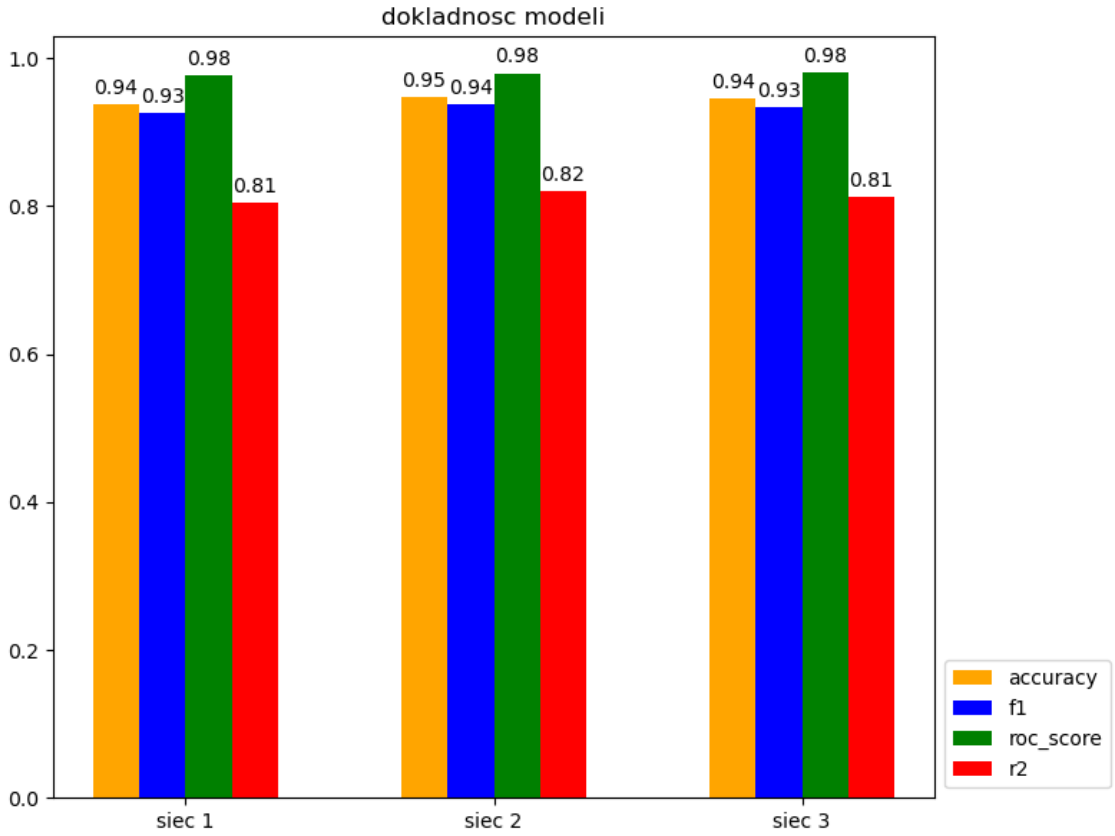
	Siec 1	*Siec 2*	*Siec 3*
Funkcja strat	BCELoss	BCELoss	BCELoss
Algorytm optymalizacji	Adam	Adam	Adam
Funkcja aktywacji warstwa 1	Tanh	Tanh	Tanh
Funkcja aktywacji warstwa 2	Tanh	Tanh	Tanh
Funkcja aktywacji wyjście	Sigmoid	Sigmoid	Sigmoid

	ETA		**20 epok**	**100 epok**	**1000 epok**	**2000 epok**
Sieć 1	0.01	Accuracy	0.88	0.94	0.93	0.94
		F1	0.85	0.93	0.92	0.93
		ROC	0.95	0.98	0.98	0.97
		R2	0.42	0.81	0.77	0.80
		Czas (sek)	0.05	0.19	1.82	3.59
Sieć 2		Accuracy	0.92	0.95	0.92	0.92
		F1	0.91	0.94	0.91	0.90
		ROC	0.96	0.98	0.96	0.94
		R2	0.69	0.82	0.71	0.69
		Czas (sek)	0.05	0.19	2.02	4.10
Sieć 3		Accuracy	0.93	0.94	0.92	0.94
		F1	0.92	0.93	0.91	0.93
		ROC	0.98	0.98	0.97	0.97
		R2	0.79	0.81	0.71	0.76
		Czas (sek)	0.11	0.25	9.65	19.00

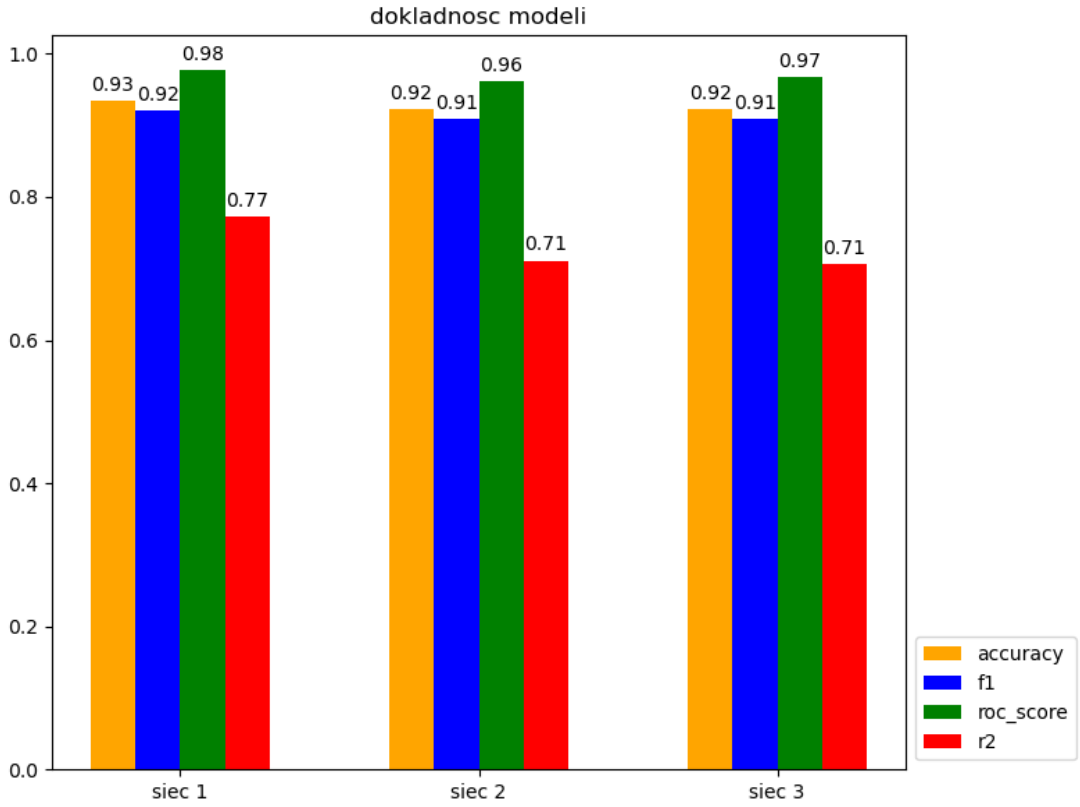
20 epok – dwie warstwy ukryte, funkcje aktywacji (Tanh, Tanh, Sigmoid)



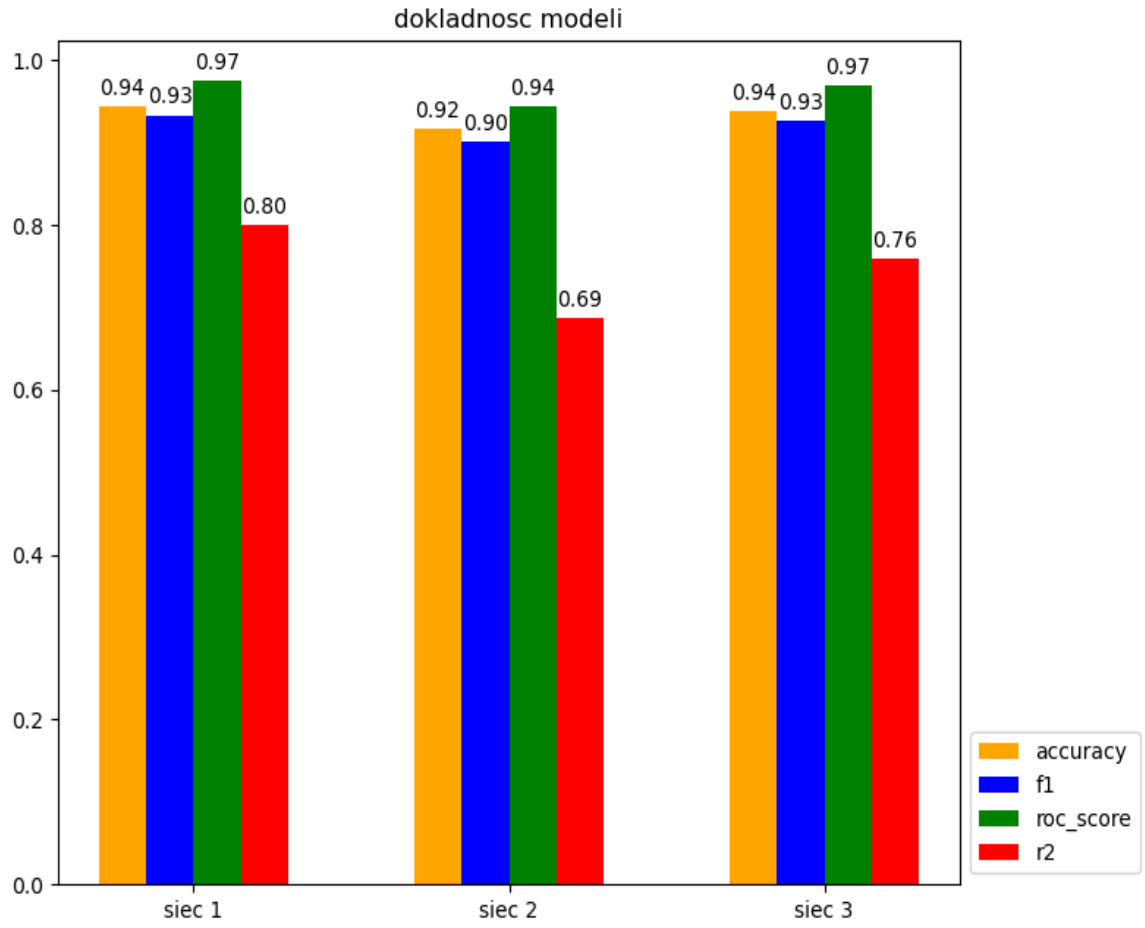
100 epok - dwie warstwy ukryte, funkcje aktywacji (Tanh, Tanh, Sigmoid)



1000 k - dwie warstwy ukryte, funkcje aktywacji (Tanh, Tanh, Sigmoid)



2000 epok - dwie warstwy ukryte, funkcje aktywacji (Tanh, Tanh, Sigmoid)



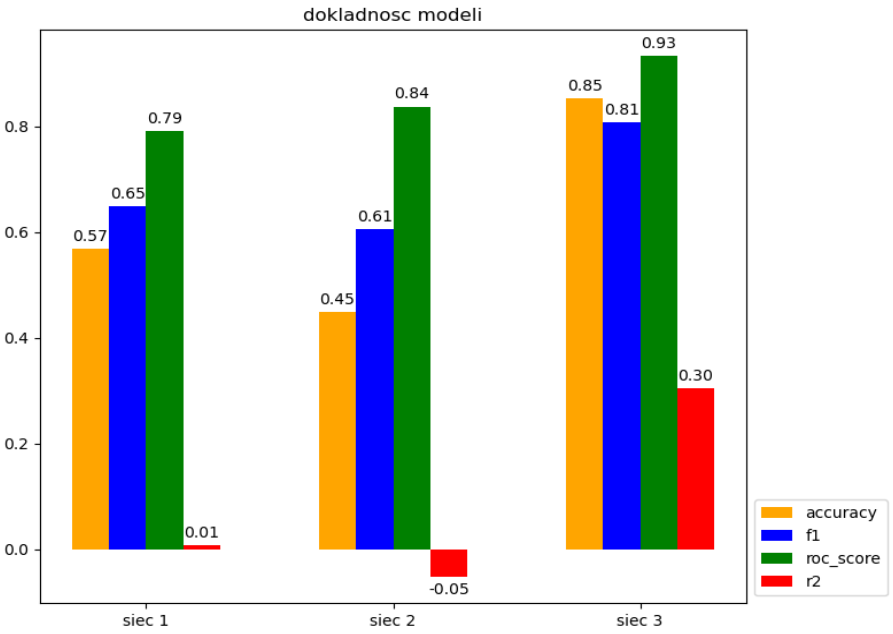
Wszystkie sieci poprawiają swoje wyniki (Accuracy, F1, ROC, R2) wraz ze zwiększeniem liczby epok, ale po 100 epokach przyrosty są mniejsze, co wskazuje, że model osiąga stabilność. Dla wszystkich sieci największa poprawa w wynikach metryk następuje między 20 a 100 epoką, po czym wyniki się stabilizują. Sieć 3, choć ma najdłuższy czas trenowania (do 19 sekund przy 2000 epokach), to pokazuje stabilne wyniki na wszystkich poziomach epok. Czas trenowania wzrasta wraz ze wzrostem liczby epok, szczególnie dla bardziej złożonych sieci, co widać w przypadku Sieci 3. Sieć ta zawiera 32 neurony dla pierwszej warstwy oraz 64 dla drugiej, dlatego czas trenowania dramatycznie wzrasta przy niewielkim powiększeniu liczby epok. Natomiast Sieć 2 pokazuje dobre wyniki we wszystkich metrykach i osiąga je w stosunkowo krótkim czasie, co czyni ją efektywnym wyborem przy zachowaniu równowagi między jakością a czasem trenowania.

Kolejny testy przeprowadzę nadal na tych samych sieciach ale z różnym współczynnikiem uczenia

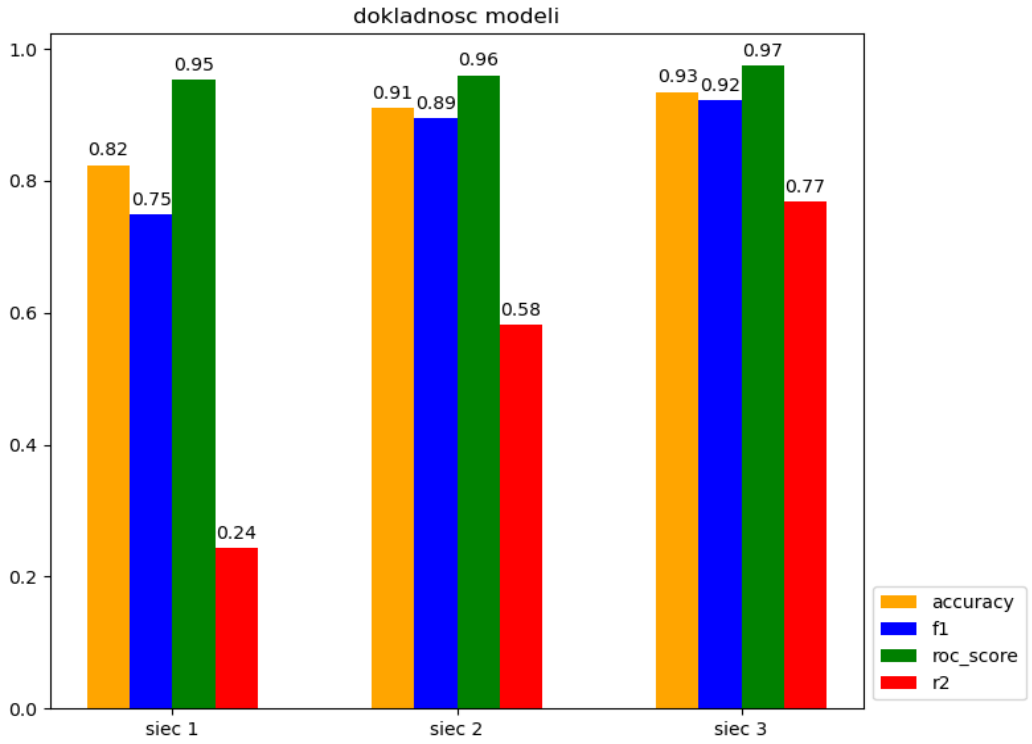
	Siec 1	*Siec 2*	*Siec 3*
Funkcja strat	BCELoss	BCELoss	BCELoss
Algorytm optymalizacji	Adam	Adam	Adam
Funkcja aktywacji warstwa 1	Tanh	Tanh	Tanh
Funkcja aktywacji warstwa 2	Tanh	Tanh	Tanh
Funkcja aktywacji wyjście	Sigmoid	Sigmoid	Sigmoid

	ETA		**20 epok**	**100 epok**	**1000 epok**	**2000 epok**
Sieć 1	0.001	Accuracy	0.57	0.82	0.94	0.94
		F1	0.65	0.75	0.93	0.93
		ROC	0.79	0.95	0.98	0.98
		R2	0.01	0.24	0.81	0.81
		Czas (sek)	0.05	0.19	1.79	3.60
Sieć 2		Accuracy	0.45	0.91	0.95	0.94
		F1	0.61	0.89	0.94	0.93
		ROC	0.84	0.96	0.98	0.98
		R2	-0.05	0.58	0.83	0.82
		Czas (sek)	0.04	0.20	1.96	4.09
Sieć 3		Accuracy	0.85	0.93	0.92	0.93
		F1	0.81	0.92	0.91	0.92
		ROC	0.93	0.97	0.97	0.97
		R2	0.30	0.77	0.75	0.75
		Czas (sek)	0.17	0.72	9.06	16.20

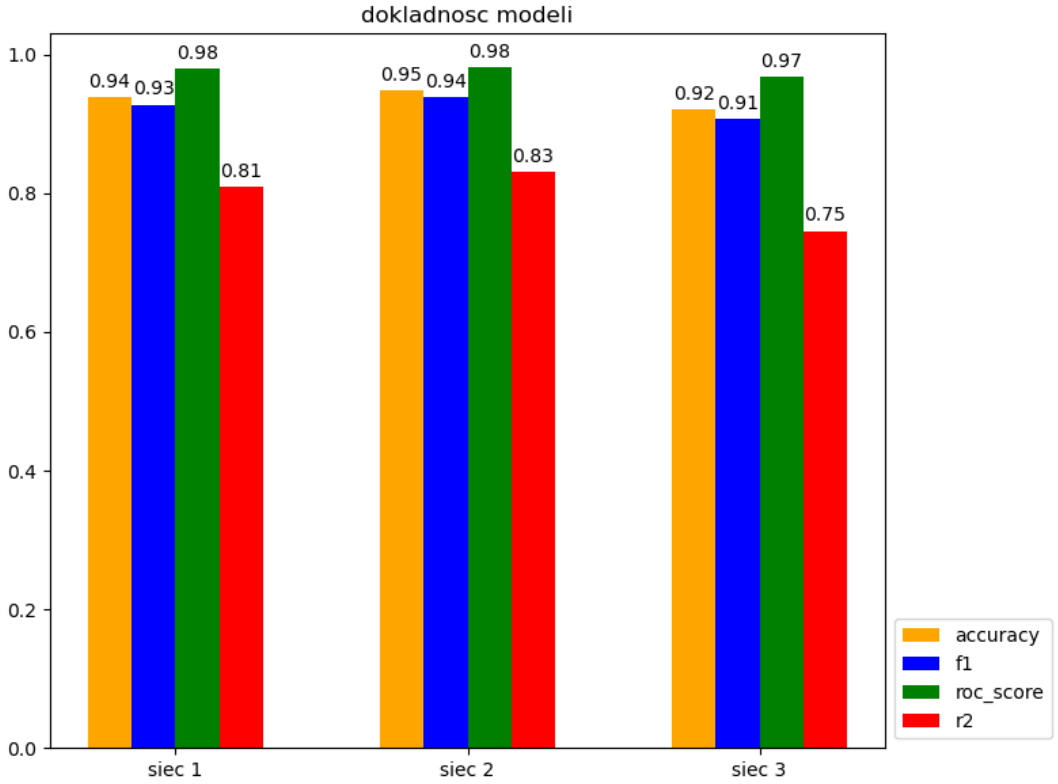
Epok: 20 – dwie warstwy ukryte . funkcja aktywacji (Tanh, Tanh, Sigmoid)



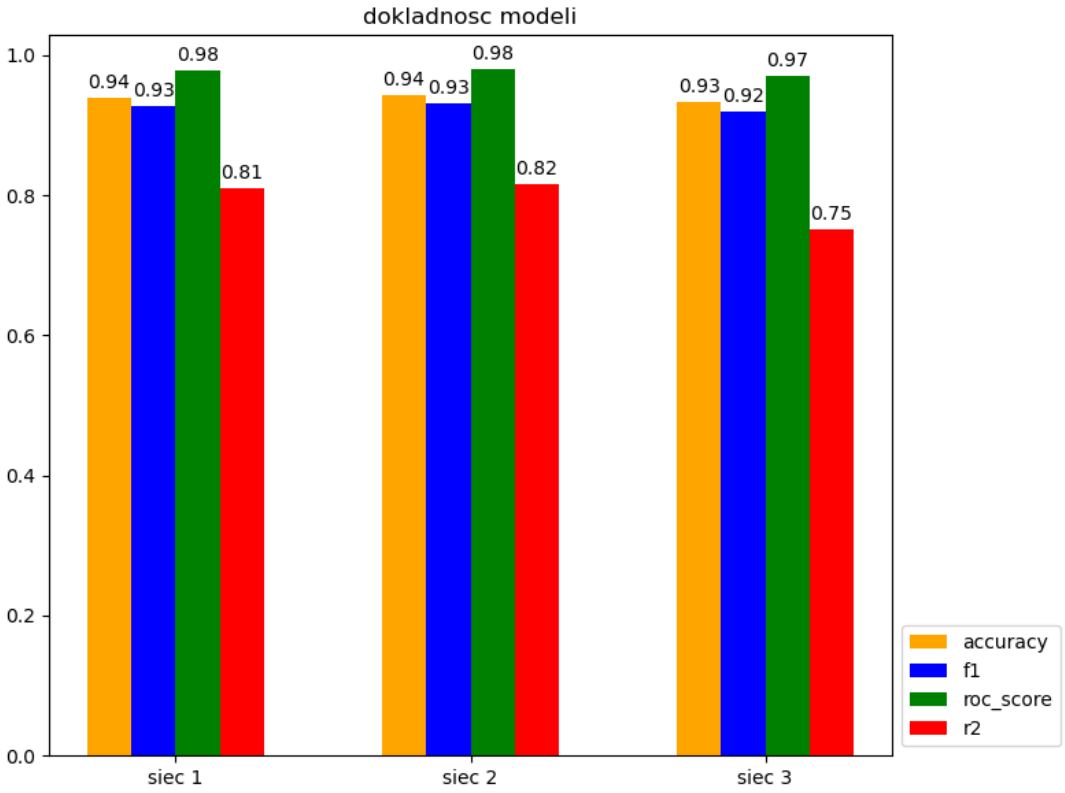
Epok: 100 – dwie warstwy ukryte . funkcja aktywacji (Tanh, Tanh, Sigmoid)



Epok: 1000 – dwie warstwy ukryte . funkcja aktywacji (Tanh, Tanh, Sigmoid)



Epok: 2000 – dwie warstwy ukryte . funkcja aktywacji (Tanh, Tanh, Sigmoid)



Wszystkie sieci poprawiają swoje wyniki (Accuracy, F1, ROC, R2) wraz ze zwiększeniem liczby epok, w poprzednim przypadku przy niższym współczynniku uczenia była to liczba epok 100, teraz jest to około 1000 epoki. Poprawa metryki sieci występuje między setną a tysięczną epoką. Wtedy sieć już w miarę jest stabilna. Sieć 3 ma najdłuższy czas trenowania co oznaczałoby że najlepiej sobie poradzi, natomiast w tej sytuacji jak przedstawiłem na wykresie nie jest najlepszą siecią. Sieć 1 pokazuje najlepsze wyniki we wszystkich metrykach i osiąga je w stosunkowo krótkim czasie, dlatego uważam że jest najlepsza dla tych parametrów. Sieć 3 dla epok (20, 100) jest jedną z lepszych ze względu na czas i efektywność. Natomiast jak dojdziemy do tysięcznej epoki to sieć nie ulega poprawie, wręcz przeciwnie delikatnie zmniejsza się jej dokładność. Jak widać współczynnik uczenia jak i ilość epok jest dość ważna, czasami zbyt małe wartości mogą od razu dać nam oczekiwany rezultat a zwiększenie liczby epok nie zawsze może wpłynąć pozytywnie na sieć ponieważ może się ona przetrenować.

Kolejne testy będą realizowane na poniższej sieci z jedną warstwą ukrytą

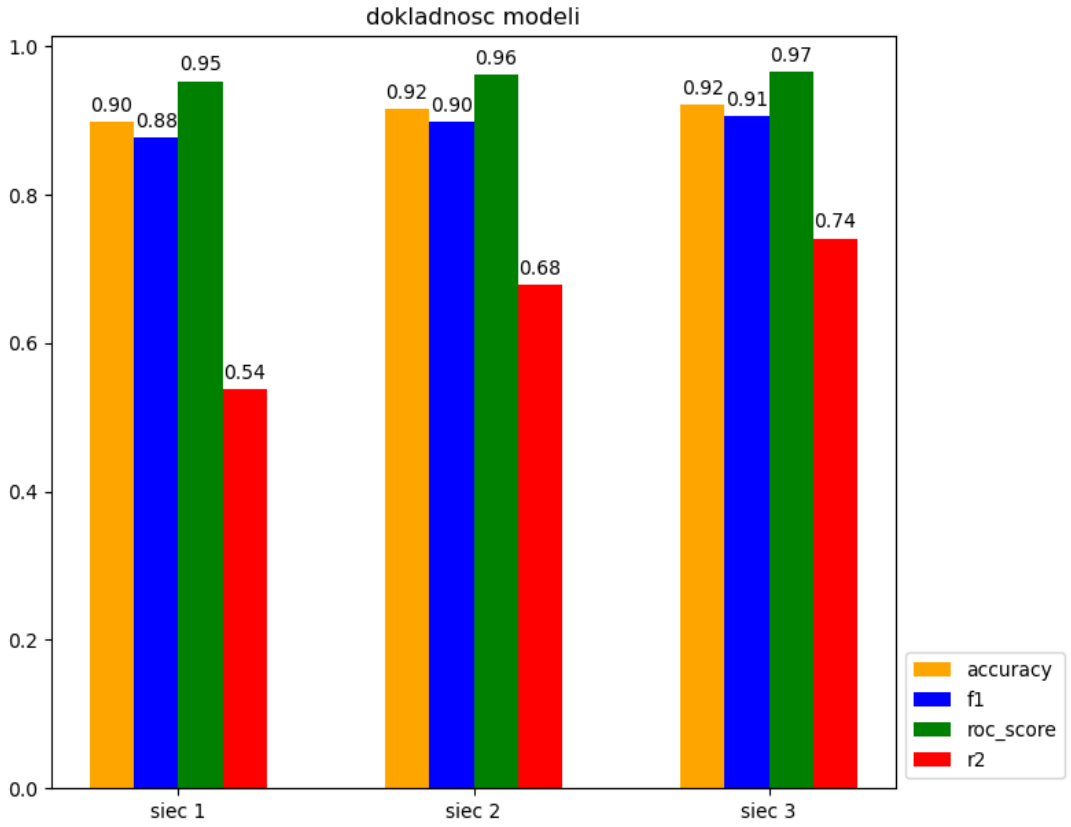
```
"""
siec z jedna warstwa ukrta
"""
network_model_1 = nn1.NeuralNetwork(input_size, hidden_size_1: 4, output_size: 1,
                                    nn.Tanh(), nn.Sigmoid())
network_model_2 = nn1.NeuralNetwork(input_size, hidden_size_1: 8, output_size: 1,
                                    nn.Tanh(), nn.Sigmoid())
network_model_3 = nn1.NeuralNetwork(input_size, hidden_size_1: 32, output_size: 1,
                                    nn.Tanh(), nn.Sigmoid())
```

	Siec 1	*Siec 2*	*Siec 3*
Funkcja strat	BCELoss	BCELoss	BCELoss
Algorytm optymalizacji	Adam	Adam	Adam
Funkcja aktywacji warstwa 1	Tanh	Tanh	Tanh
Funkcja aktywacji wyjście	Sigmoid	Sigmoid	Sigmoid

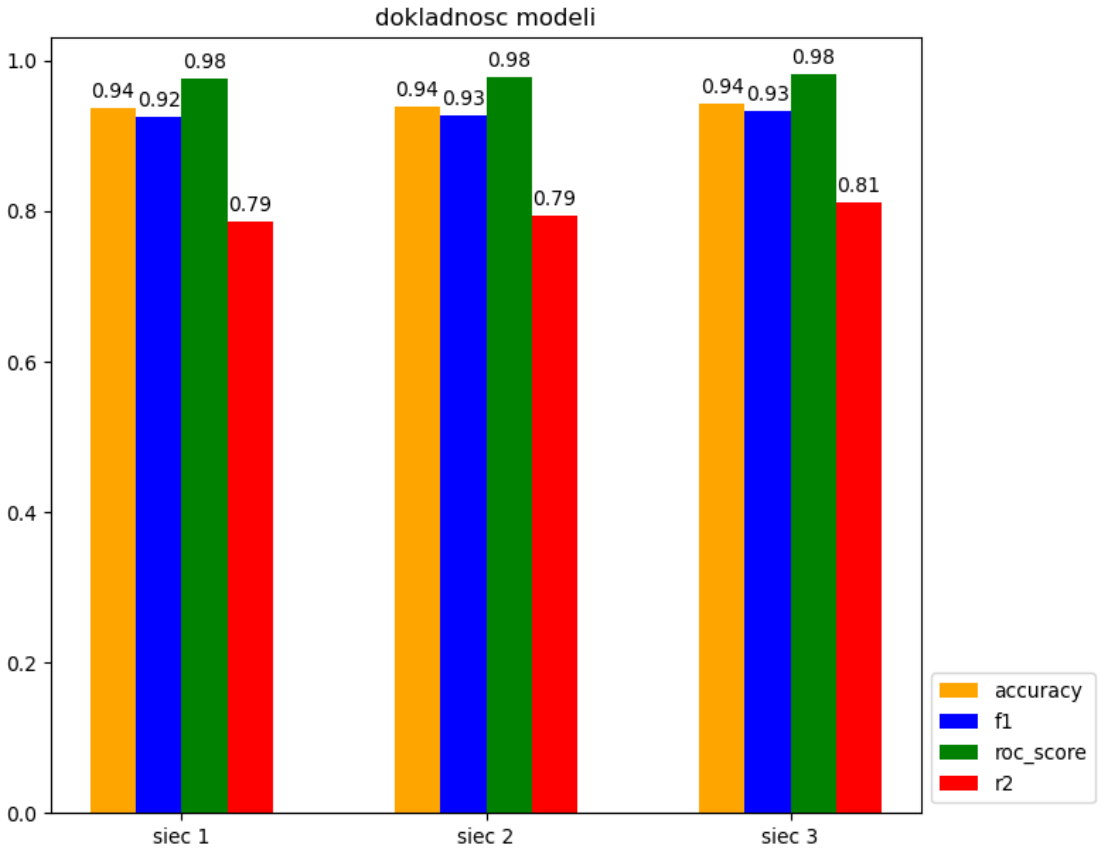
	ETA		**20 epok**	**100 epok**	**1000 epok**	**2000 epok**
Sieć 1	0.01	Accuracy	0.90	0.94	0.94	0.94
		F1	0.88	0.92	0.93	0.93
		ROC	0.95	0.98	0.98	0.98
		R2	0.54	0.79	0.81	0.81
		Czas (sek)	0.04	0.15	1.47	2.91
Sieć 2		Accuracy	0.92	0.94	0.95	0.94
		F1	0.90	0.93	0.94	0.93
		ROC	0.96	0.98	0.98	0.98
		R2	0.68	0.79	0.82	0.77
		Czas (sek)	0.03	0.14	1.49	2.96
Sieć 3		Accuracy	0.92	0.94	0.93	0.92
		F1	0.91	0.93	0.91	0.91
		ROC	0.97	0.98	0.97	0.97
		R2	0.74	0.81	0.76	0.73
		Czas (sek)	0.04	0.23	1.82	5.24

Przy zastosowaniu jednej warstwy od razu można zobaczyć duża różnice w czasie trenowania. W poprzednich modelach sieci, gzie było dwie warstwy czas był znacznie większy, dla przypadku Sieci 3 dla 2000 epok sieć w poprzednim modelu uczyła się około 19 sekund. Tutaj można zaobserwować różnice 14/15 sekund.

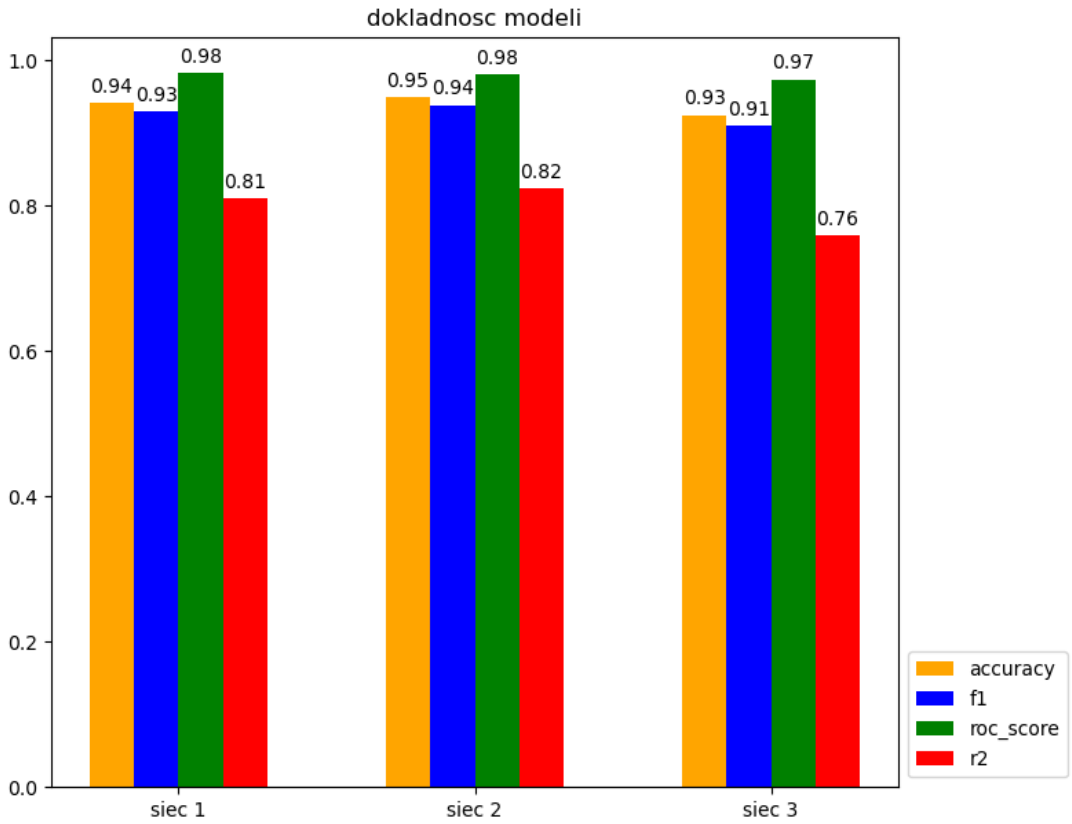
20 epok – Eta 0.01 - 1 warstwa ukryta, funkcja aktywacji (Tanh, Sigmoid)



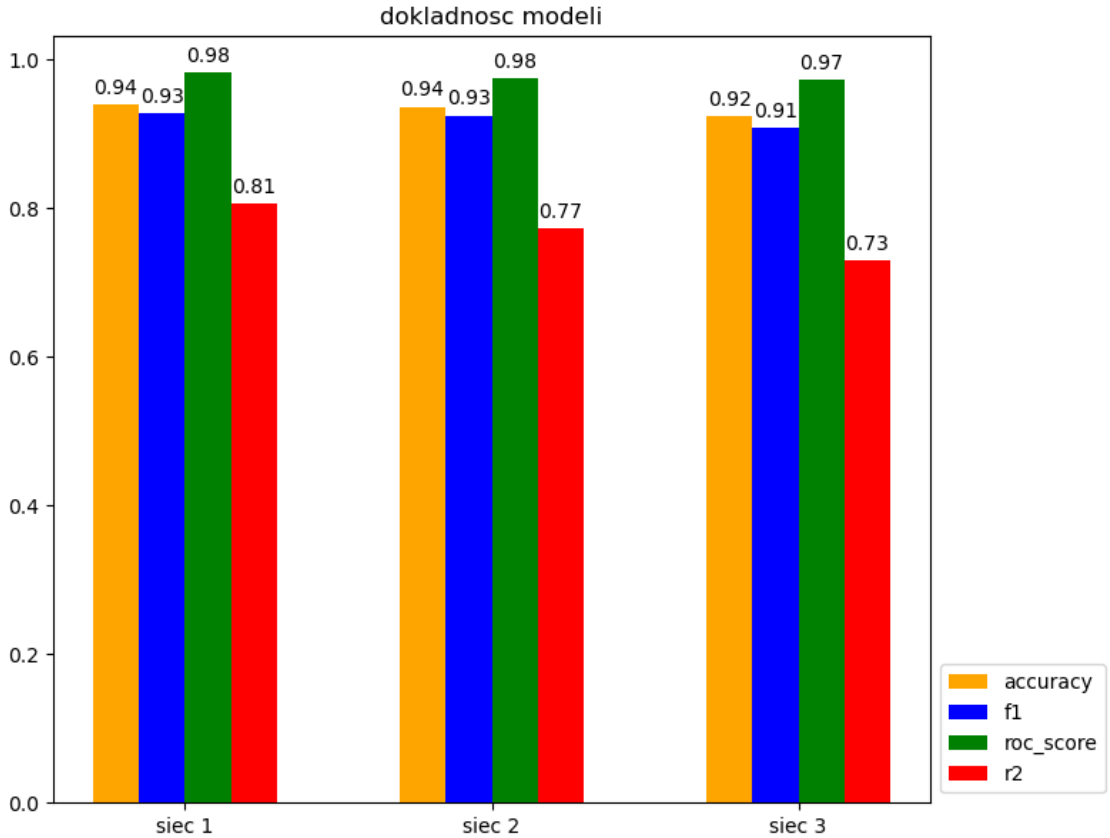
100 epok – Eta 0.01 - 1 warstwa ukryta, funkcja aktywacji (Tanh, Sigmoid)



1000 k – Eta 0.01 - 1 warstwa ukryta, funkcja aktywacji (Tanh, Sigmoid)



2000 epok – Eta 0.01 - 1 warstwa ukryta, funkcja aktywacji (Tanh, Sigmoid)



Tak jak w poprzednich przypadkach przy zwiększeniu ilości epok poprawia się dokładność, przy około 1000 epokach mamy dobrze już wyuczone sieci. Natomiast Sieć 3 już przy 100 epokach bardzo dobrze sobie radzi. Dodatkowo sieci osiągają wysokie wartości ROC co oznacza że sieci bardzo dobrze rozróżniają klasy. W tym przypadku czas uczenia jest zdecydowanie mniejszy ponieważ zamiast dwóch warstw mamy jedną, co przyspiesza uczenie sieci nie tracąc na jakości.

Zmiana funkcji aktywacji dla wszystkich sieci
z jedną warstwą ukrytą i z dwiema warstwami ukrytymi

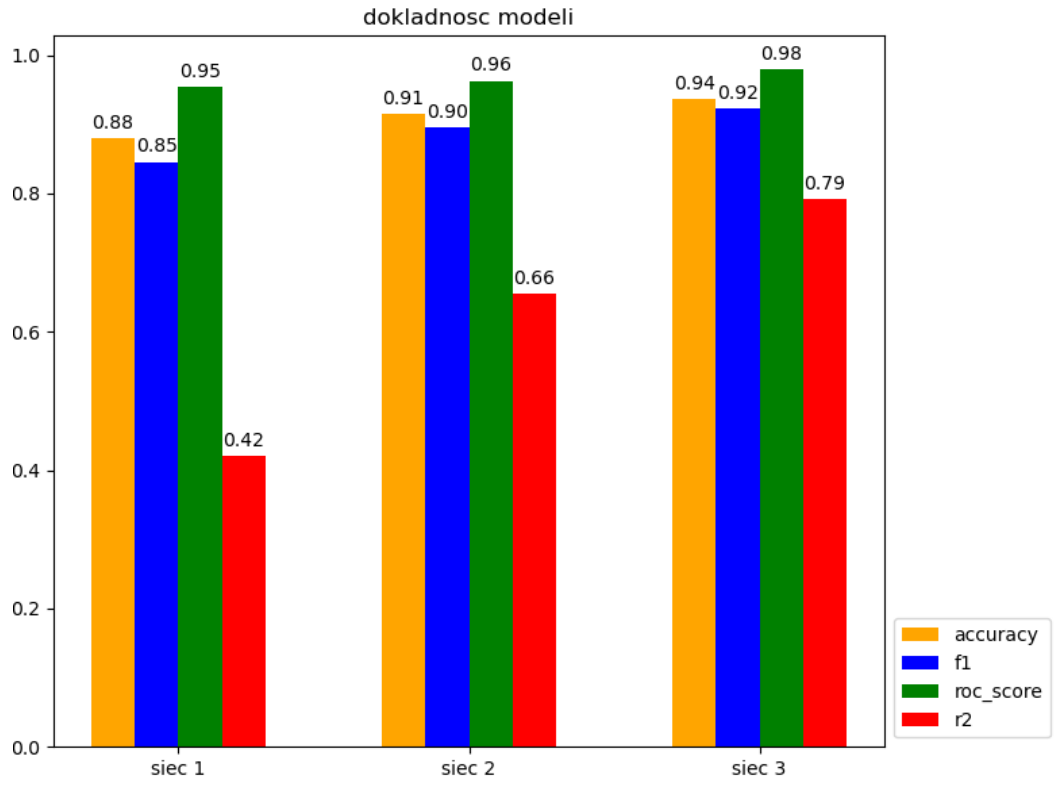
```
network_model_1 = nn2.NeuralNetwork(input_size, hidden_size_1: 4, hidden_size_2: 4, output_size: 1,
                                    nn.ReLU(), nn.ReLU(), nn.Sigmoid())
network_model_2 = nn2.NeuralNetwork(input_size, hidden_size_1: 8, hidden_size_2: 8, output_size: 1,
                                    nn.ReLU(), nn.ReLU(), nn.Sigmoid())
network_model_3 = nn2.NeuralNetwork(input_size, hidden_size_1: 32, hidden_size_2: 64, output_size: 1,
                                    nn.ReLU(), nn.ReLU(), nn.Sigmoid())
```

Dla funckji aktywacji:

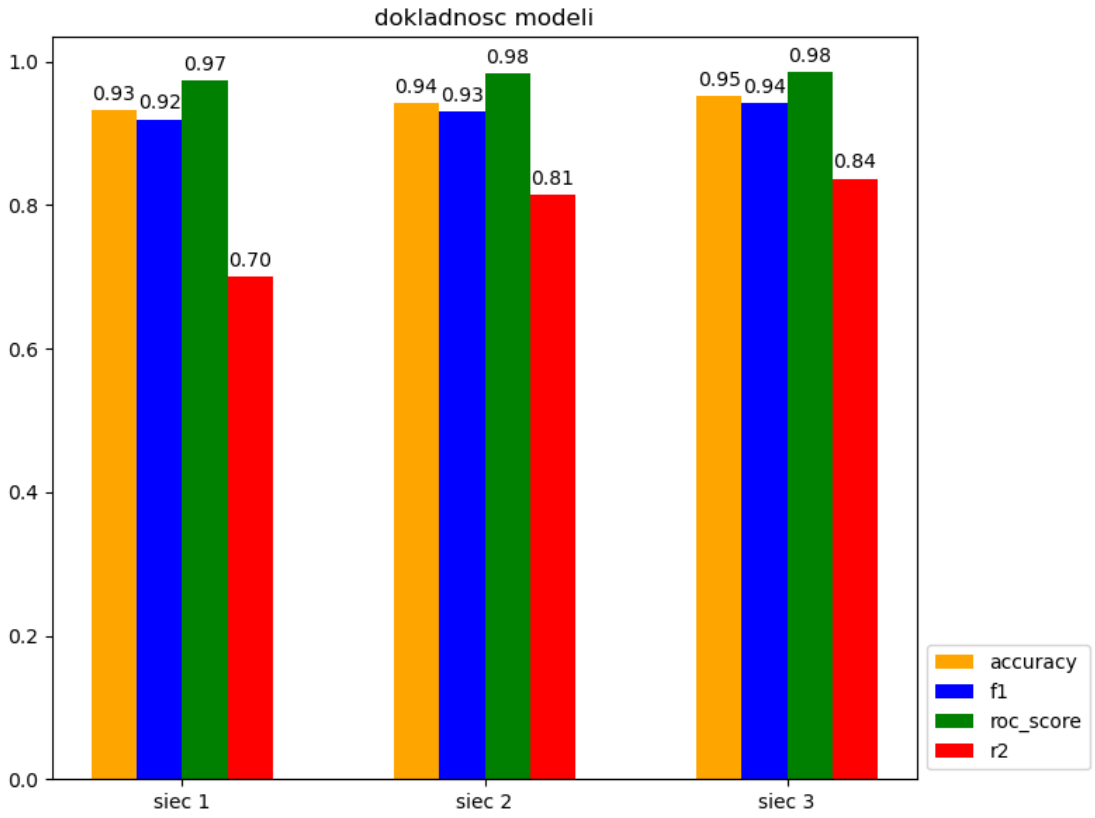
- 1 warstwa – ReLU,
- 2 warstwa – ReLU
- wyjście - Sigmoid

	ETA		20 epok	100 epok	1000 epok	2000 epok
Sieć 1	0.01	Accuracy	0.88	0.93	0.94	0.94
		F1	0.85	0.92	0.93	0.93
		ROC	0.95	0.97	0.98	0.97
		R2	0.42	0.70	0.81	0.80
		Czas (sek)	0.05	0.16	1.24	2.66
Sieć 2		Accuracy	0.91	0.94	0.94	0.93
		F1	0.90	0.93	0.93	0.91
		ROC	0.96	0.98	0.98	0.97
		R2	0.66	0.81	0.81	0.74
		Czas (sek)	0.05	0.18	1.76	3.80
Sieć 3		Accuracy	0.94	0.95	0.95	0.94
		F1	0.92	0.94	0.94	0.93
		ROC	0.98	0.98	0.98	0.98
		R2	0.79	0.84	0.80	0.78
		Czas (sek)	0.06	0.25	4.21	11.03

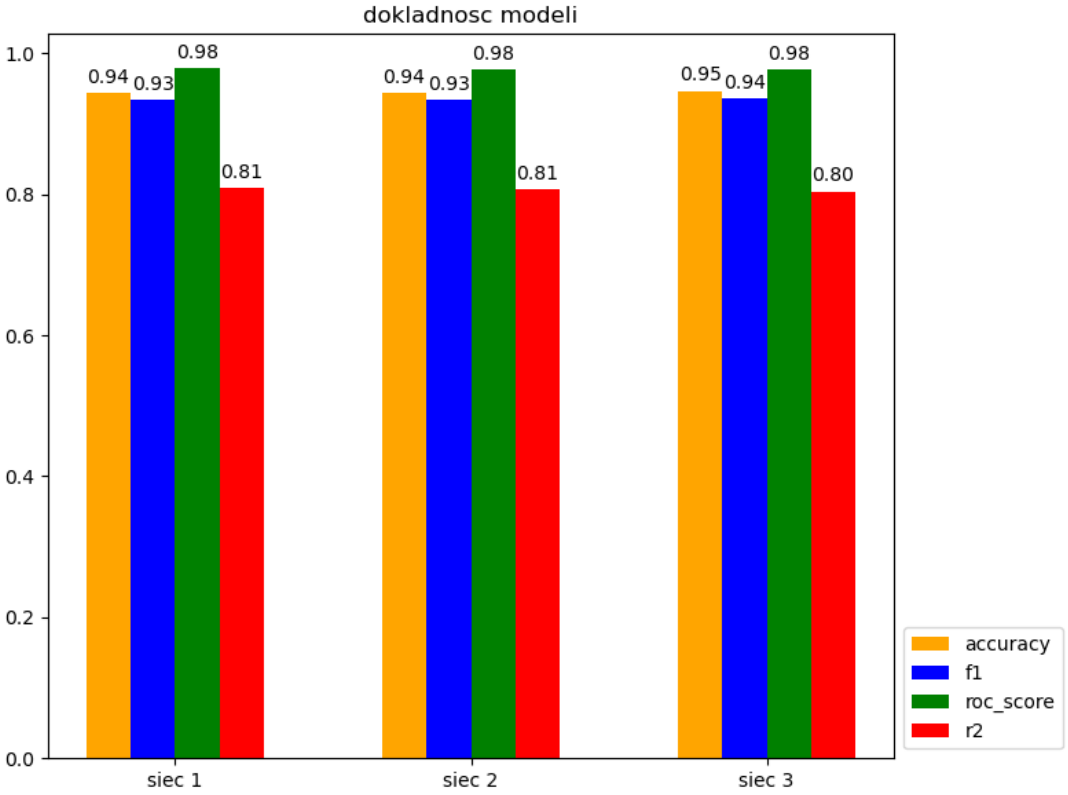
20 epok – Eta 0.01 - 2 warstwy ukryte (Funkcja aktywacji: ReLU, ReLU, Sigmoid)



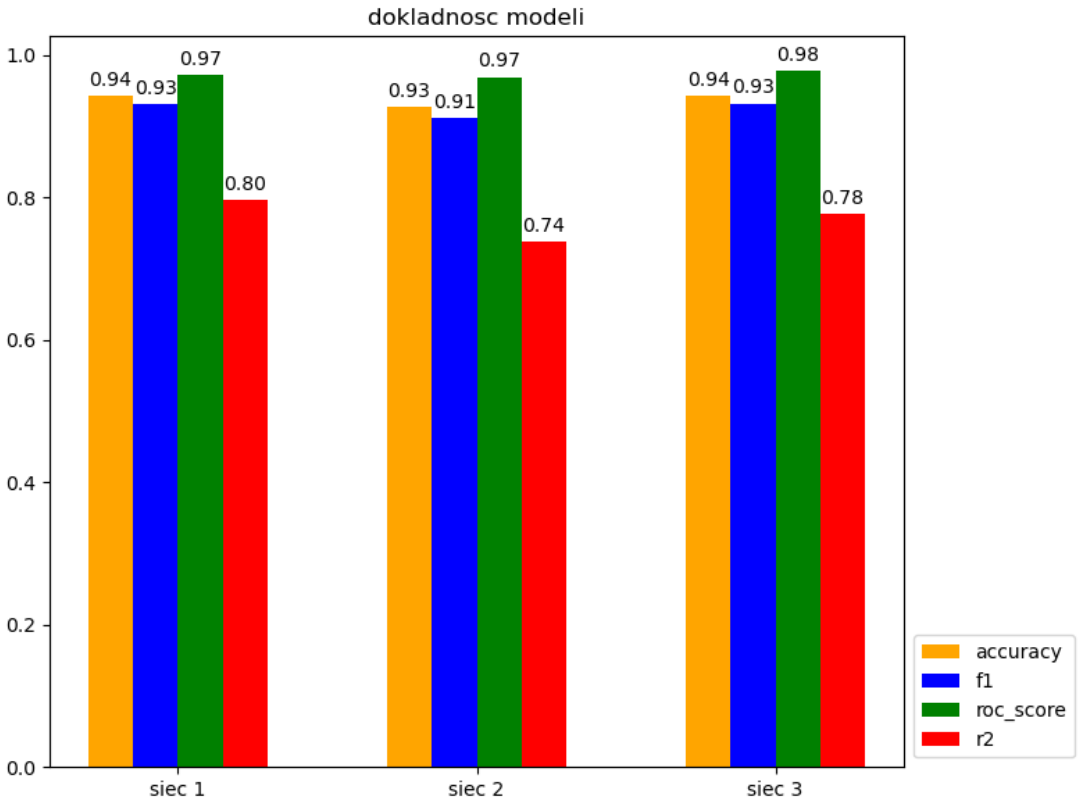
100 epok – Eta 0.01 - 2 warstwy ukryte (Funkcja aktywacji: ReLU, ReLU, Sigmoid)



1000 epok – Eta 0.01 - 2 warstwy ukryte (Funkcja aktywacji: ReLU, ReLU, Sigmoid)



2000 epok – Eta 0.01 - 2 warstwy ukryte (Funkcja aktywacji: ReLU, ReLU, Sigmoid)



Sieć 1 osiągnęła maksymalną dokładność 0.94 po 1000 epokach, co jest podobne do wcześniejszych wyników, ale już przy 20 epokach dokładność wynosiła 0.88, co wskazuje na to że sieć szybciej osiąga lepsza wydajność. Sieć 2 poprawiła dokładność do 0.94 już po 100 epokach, co sugeruje efektywniejsze uczenie się w porównaniu do wcześniejszych testów. Ostatnia sieć przy 100 epokach osiągnęła najlepszy wynik dodatkowo w krótkim czasie. Uważam że zmiana funkcji aktywacji **poprawiła sieć**, szczególnie w osiągnieciu stabilnych wyników. Funkcja ReLU przyspiesza trening i jest bardzo dobrze efektywna. Dowodem na to że ReLU jest tutaj lepsze, to wystarczy sprawdzić wartości R2 które pokazują jak dobrze sieć dopasowuje się do danych.

Testowanie jeszcze dla jednej warstwy

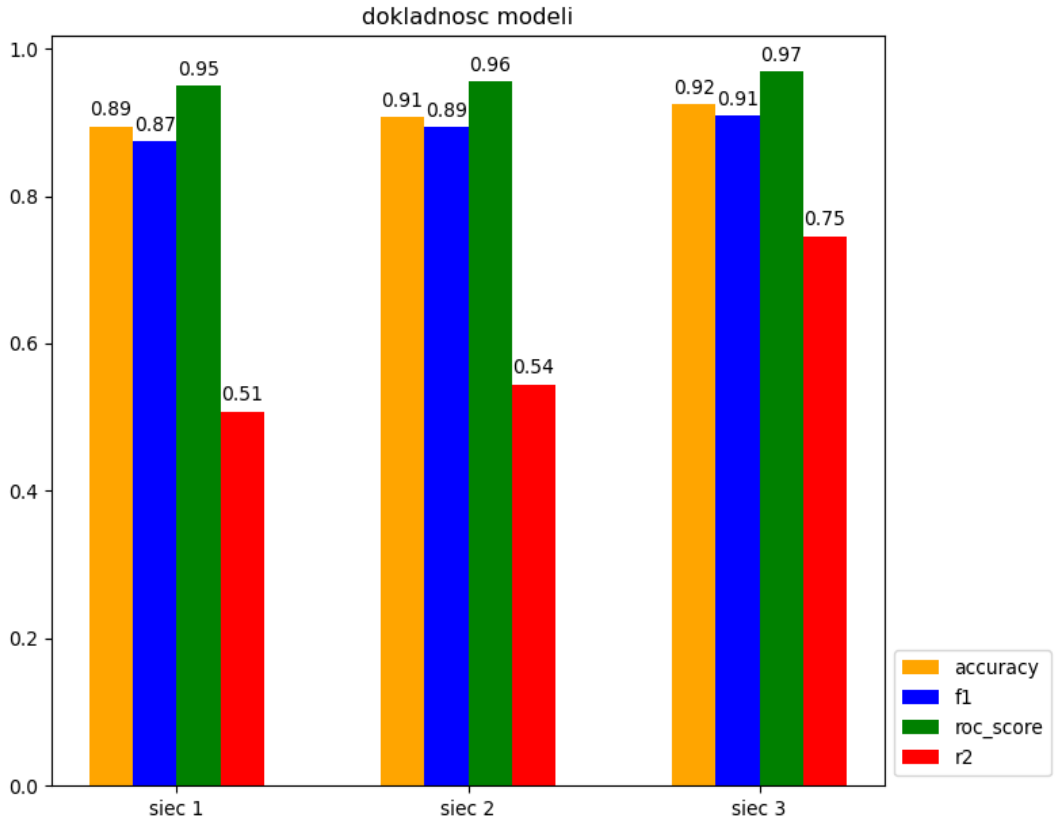
```
network_model_1 = nn1.NeuralNetwork(input_size, hidden_size_1: 4, output_size: 1,
                                    nn.ReLU(), nn.Sigmoid())
network_model_2 = nn1.NeuralNetwork(input_size, hidden_size_1: 8, output_size: 1,
                                    nn.ReLU(), nn.Sigmoid())
network_model_3 = nn1.NeuralNetwork(input_size, hidden_size_1: 32, output_size: 1,
                                    nn.ReLU(), nn.Sigmoid())
```

Dla funkcji aktywacji:

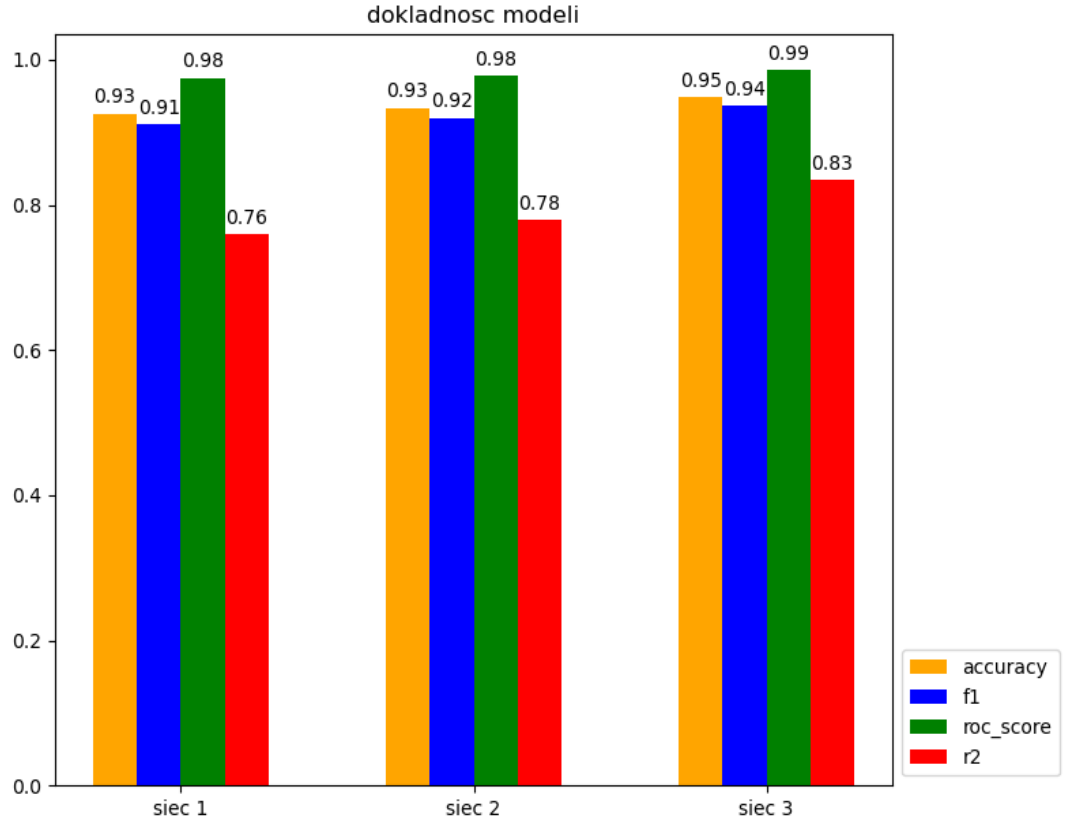
- 1 warstwa – ReLU,
- wyjście - Sigmoid

	ETA		20 epok	100 epok	1000 epok	2000 epok
Sieć 1	0.01	Accuracy	0.89	0.93	0.94	0.94
		F1	0.87	0.91	0.92	0.93
		ROC	0.95	0.98	0.98	0.98
		R2	0.51	0.76	0.82	0.81
		Czas (sek)	0.04	0.14	0.99	2.22
Sieć 2		Accuracy	0.91	0.93	0.95	0.93
		F1	0.89	0.92	0.94	0.92
		ROC	0.96	0.98	0.98	0.97
		R2	0.54	0.78	0.81	0.77
		Czas (sek)	0.03	0.14	1.41	2.62
Sieć 3		Accuracy	0.92	0.95	0.94	0.93
		F1	0.91	0.94	0.92	0.92
		ROC	0.97	0.99	0.98	0.97
		R2	0.75	0.83	0.78	0.76
		Czas (sek)	0.04	0.18	2.07	4.81

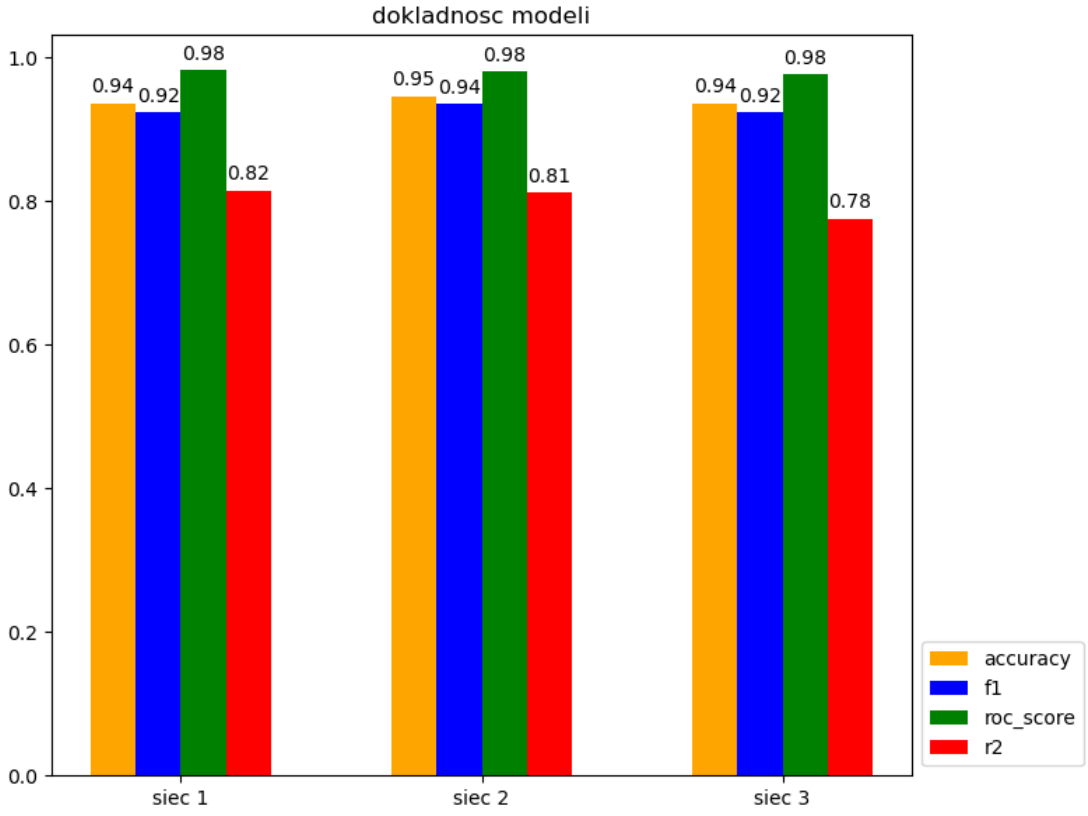
20 Epok – Eta 0.01 - 1 warstwa ukryta (Funkcja aktywacji: ReLU, Sigmoid)



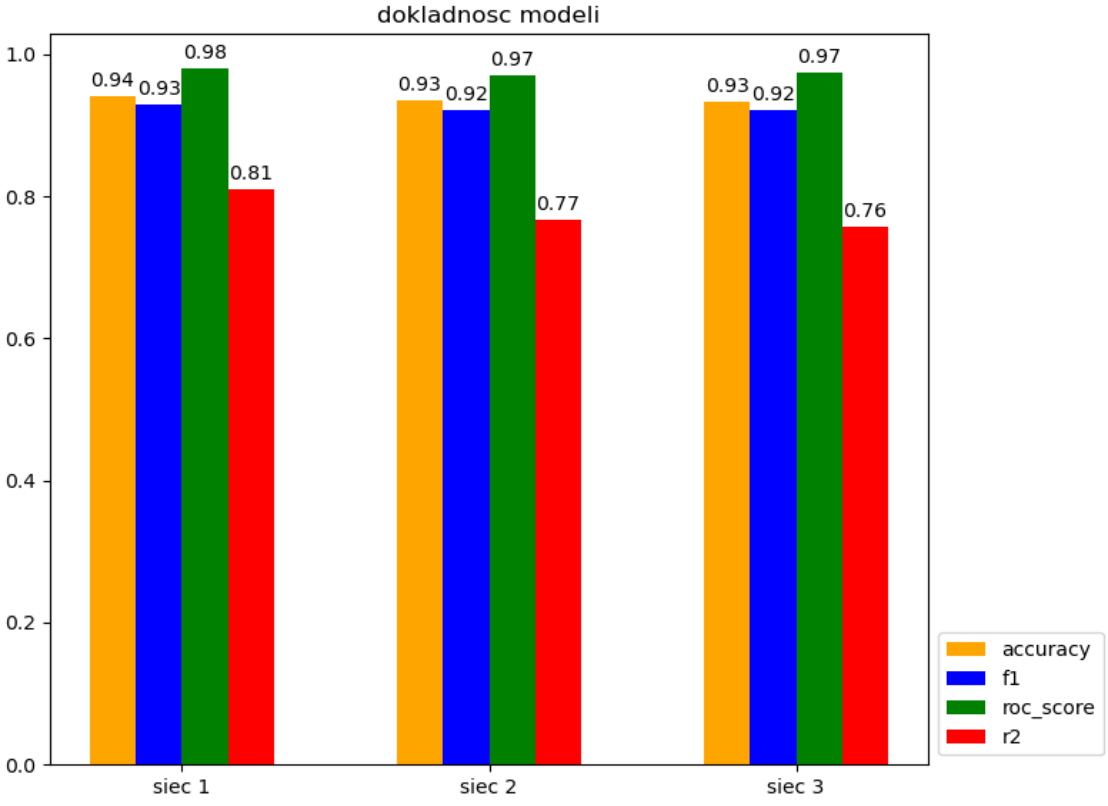
1000 epok – Eta 0.01 - 1 warstwa ukryta (Funkcja aktywacji: ReLU, Sigmoid)



1000 epok – Eta 0.01 - 1 warstwa ukryta (Funkcja aktywacji: ReLU, Sigmoid)



2000 epok – Eta 0.01 - 1 warstwa ukryta (Funkcja aktywacji: ReLU, Sigmoid)



Wszystkie sieci osiągają dobre wyniki przy użyciu jednej warstwy ukrytej z funkcją aktywacji ReLU i funkcją Sigmoid na wyjściu. Największe korzyści widać do około 1000 epoki, po czym wyniki się stabilizują. Sieć 3 jest najbardziej efektywna w osiąganiu najwyższych wartości metryk, chociaż wymaga więcej czasu na trening. Sieć 1 i Sieć 2 osiągają wysoką wydajność przy krótszym czasie trenowania, jeżeli bierzemy pod uwagę czas, to te metody mogą być jedne z lepszych. Jest to bardzo dziwne ponieważ zazwyczaj zwiększenie warstw oraz liczby neuronów powinno wpłynąć pozytywne na uczeni się sieci. Wykorzystany zbiór danych posiada dużo cech bo jest ich około 56, dlatego zwiększenie liczby neuronów pozytywny wpływa na naukę. Można to zaobserwować na większości dokonanych testów, gdzie sieć 3 osiąga zazwyczaj lepsze rezultaty, lepsze metryki.

	ETA		20 epok	100 epok	1000 epok	2000 epok
Sieć 1	0.01	Accuracy	0.89	0.93	0.94	0.94
		F1	0.87	0.91	0.92	0.93
		ROC	0.95	0.98	0.98	0.98
		R2	0.51	0.76	0.82	0.81
		Czas (sek)	0.04	0.14	0.99	2.22
Sieć 2		Accuracy	0.91	0.93	0.95	0.93
		F1	0.89	0.92	0.94	0.92
		ROC	0.96	0.98	0.98	0.97
		R2	0.54	0.78	0.81	0.77
		Czas (sek)	0.03	0.14	1.41	2.62
Sieć 3		Accuracy	0.92	0.95	0.94	0.93
		F1	0.91	0.94	0.92	0.92
		ROC	0.97	0.99	0.98	0.97
		R2	0.75	0.83	0.78	0.76
		Czas (sek)	0.04	0.18	2.07	4.81

Podsumowując wszystkie dotychczas sieci, uważam że Sieć z jedną warstwą ukrytą oraz funkcja aktywacji Relu, Sigmoid, działa najlepiej ponieważ wymaga tylko 100 epok a osiąga najlepsze wyniki. Bardzo dobrze to widać na powyższej grafice, gdzie dokładność wynosi 0.95 oraz krzywa ROC jest bliska równej jeden, bo wynosi 0.99 co oznacza że model bardzo dobrze rozróżnia instancje klas. Otrzymany wynik dokładności jest bardzo podobny do modelu XGBoost bo przy zaokrągleniu do liczb dwóch miejsc po przecinku wynoszą taka samą dokładność.

Trenowanie sieci z użyciem walidacji krzyżowej KFold

Uczenie trzech sieci z podziałem na **KFoldy = 5**

```
input_size = X.shape[1]
num_epochs = 20
learning_rate = 0.01

network_model_1 = nn2.NeuralNetwork(input_size, hidden_size_1: 4, hidden_size_2: 4, output_size: 1,
                                    nn.ReLU(), nn.ReLU(), nn.Sigmoid())
network_model_2 = nn2.NeuralNetwork(input_size, hidden_size_1: 8, hidden_size_2: 8, output_size: 1,
                                    nn.ReLU(), nn.ReLU(), nn.Sigmoid())
network_model_3 = nn2.NeuralNetwork(input_size, hidden_size_1: 32, hidden_size_2: 64, output_size: 1,
                                    nn.ReLU(), nn.ReLU(), nn.Sigmoid())
```

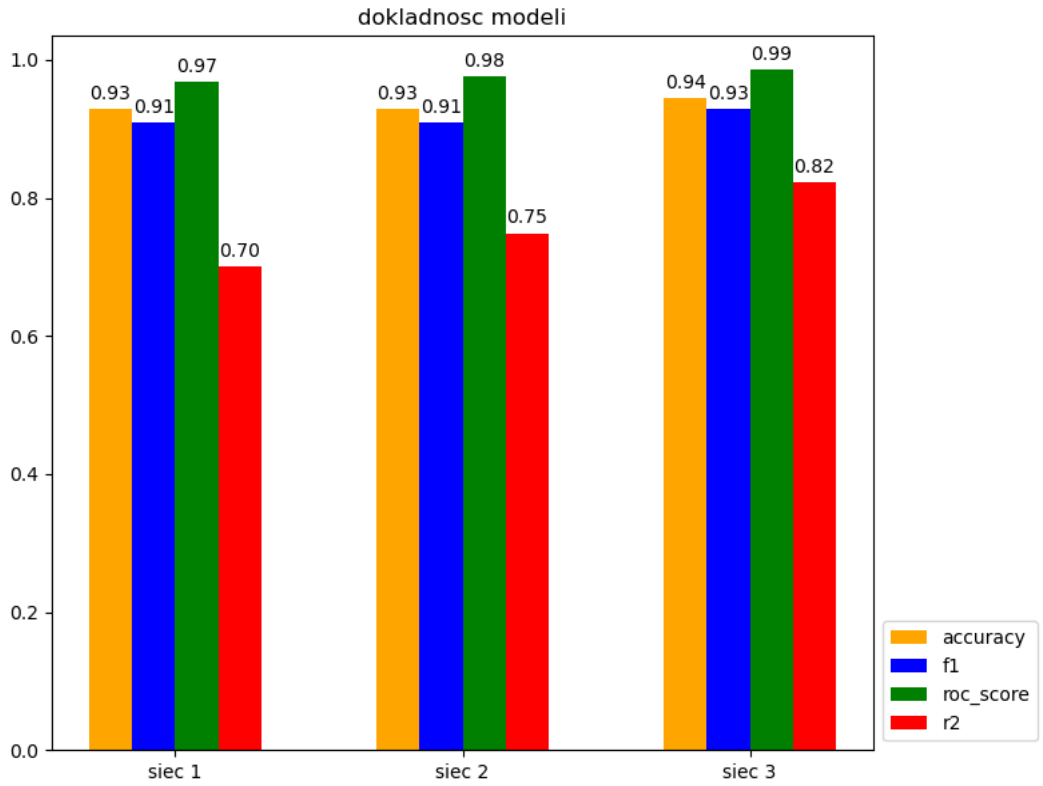
```
# funkcja straty - jak bardzo model myli sie w przewidywaniach
criterion = nn.BCELoss()
# algorytm optymalizacji - aktualizuje wagi modelu
# poprawa modelu
optimizer_1 = optim.Adam(network_model_1.parameters(), lr=learning_rate)
optimizer_2 = optim.Adam(network_model_2.parameters(), lr=learning_rate)
optimizer_3 = optim.Adam(network_model_3.parameters(), lr=learning_rate)

kf = KFold(n_splits=5, shuffle=True, random_state=42)
```

Dla funkcji aktywacji:

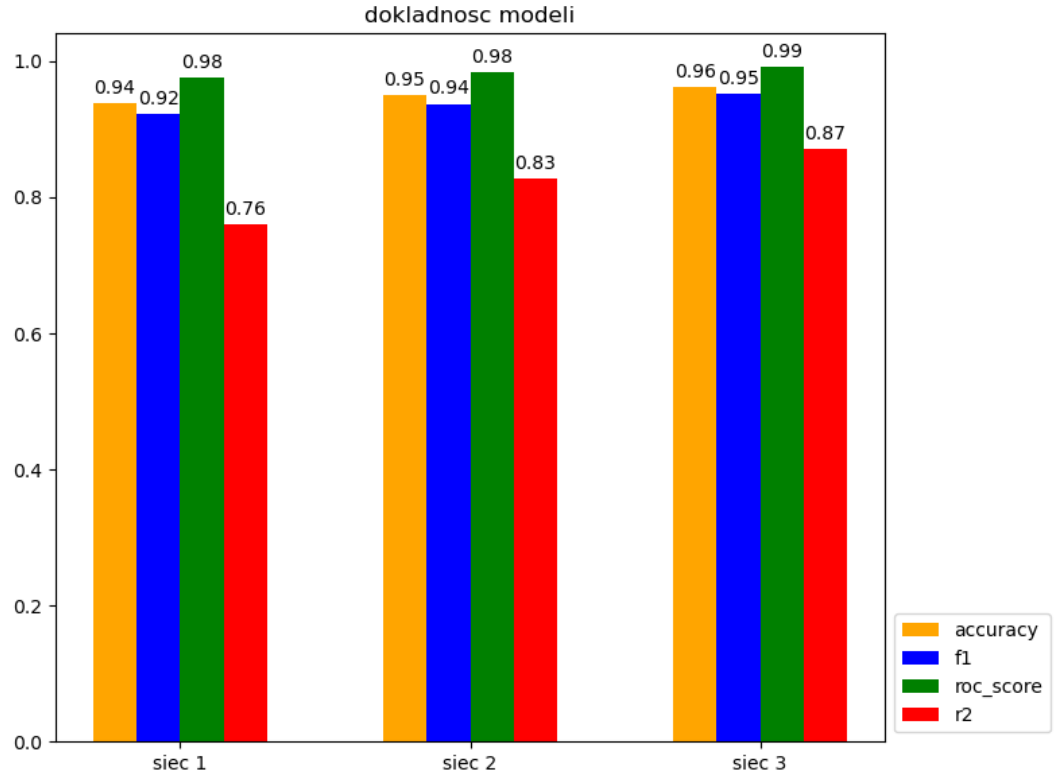
- 1 warstwa – ReLU,
- 2 warstwa – ReLU
- wyjście - Sigmoid

	ETA		20 epok	100 epok	1000 epok	2000 epok
Sieć 1	0.01	Accuracy	0.93	0.94	0.94	0.94
		F1	0.91	0.92	0.93	0.92
		ROC	0.97	0.98	0.98	0.98
		R2	0.70	0.76	0.80	0.79
		Czas (sek)	0.22	0.80	6.43	12.63
Sieć 2		Accuracy	0.93	0.95	0.95	0.95
		F1	0.91	0.94	0.93	0.93
		ROC	0.98	0.98	0.98	0.98
		R2	0.75	0.83	0.80	0.79
		Czas (sek)	0.26	0.95	10.31	19.75
Sieć 3		Accuracy	0.94	0.96	0.96	0.96
		F1	0.93	0.95	0.95	0.95
		ROC	0.99	0.99	0.99	0.99
		R2	0.82	0.87	0.86	0.85
		Czas (sek)	0.30	1.69	23.3	60.42

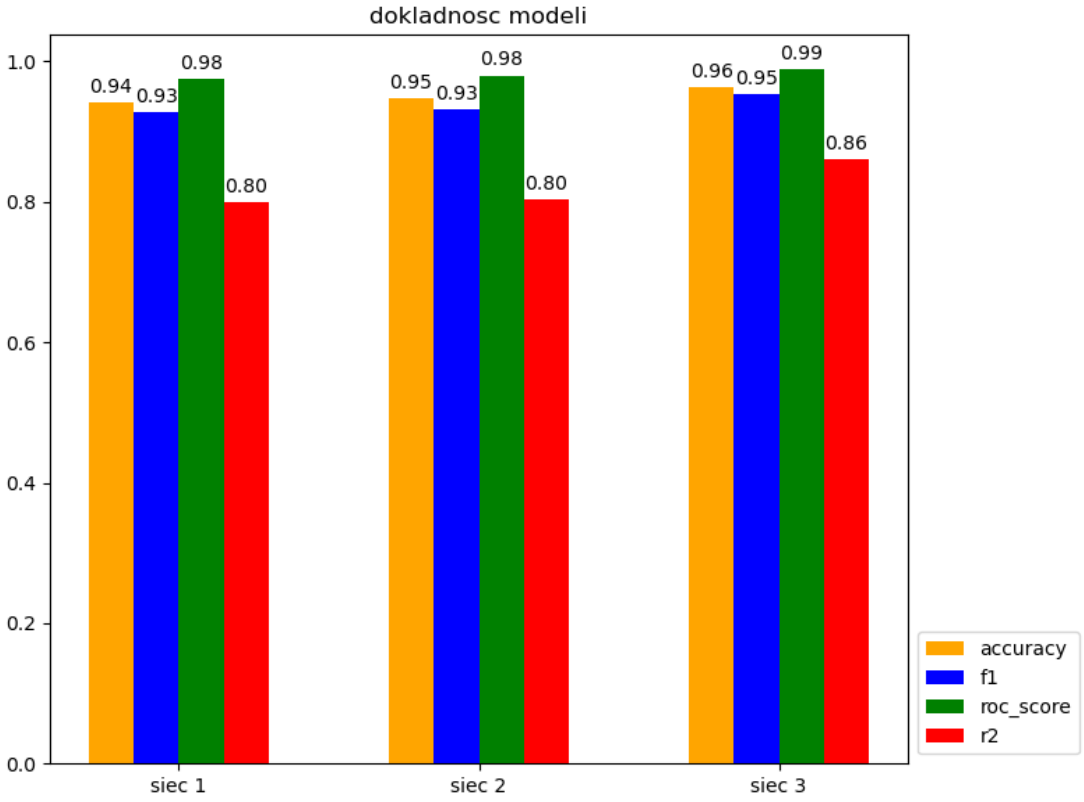
20 Epok - Eta 0.01 - 2 warstwy ukryte (Funkcja aktywacji: ReLU, ReLU, Sigmoid)



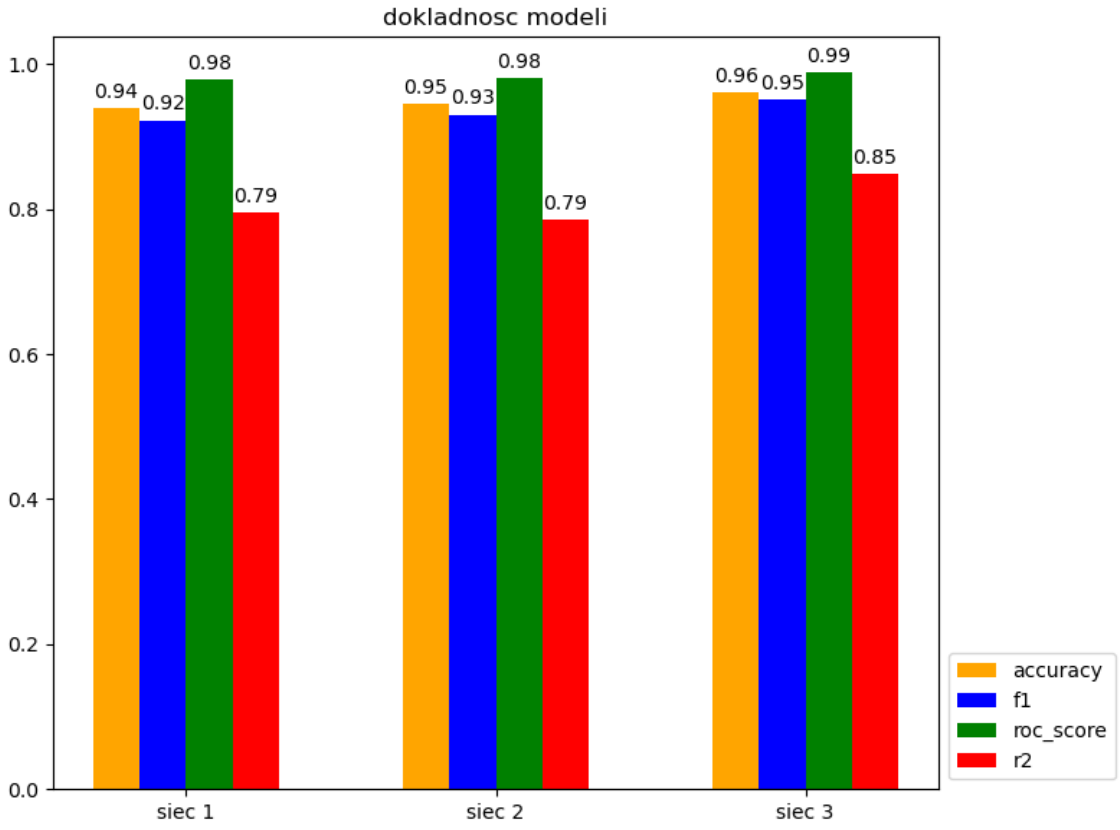
100 Epok – Eta 0.01 - 2 warstwy ukryte (Funkcja aktywacji: ReLU, ReLU, Sigmoid)



1000 Epok - Eta 0.01 - 2 warstwy ukryte (Funkcja aktywacji: ReLU, ReLU, Sigmoid)



2000 Epok - Eta 0.01 - 2 warstwy ukryte (Funkcja aktywacji: ReLU, ReLU, Sigmoid)



Do około 1000 epok sieć się uczy, poprawia swoją jakość, natomiast już po 2 tysiącach nie widać różnicy pod względem jakości. Jedynie co się zwiększyło to czas np. Dla Sieci 3 z 20 sekund na około 60 sekund. Sieć 3 osiągnęła najlepszy jakość (0.96) oraz F1 równe 0.95 co wskazuje na to że sieć posiada dobrą zdolność do klasyfikacji. Wszystkie sieci posiadały duże ROC co świadczy o tym że nie miały problemu z rozróżnianiem klas. Ogólnie Siec 1 oraz 2 osiągnęła dobre wyniki w krótkim czasie.

Kolejne testowanie jakości sieci odbędzie się dla następujących danych

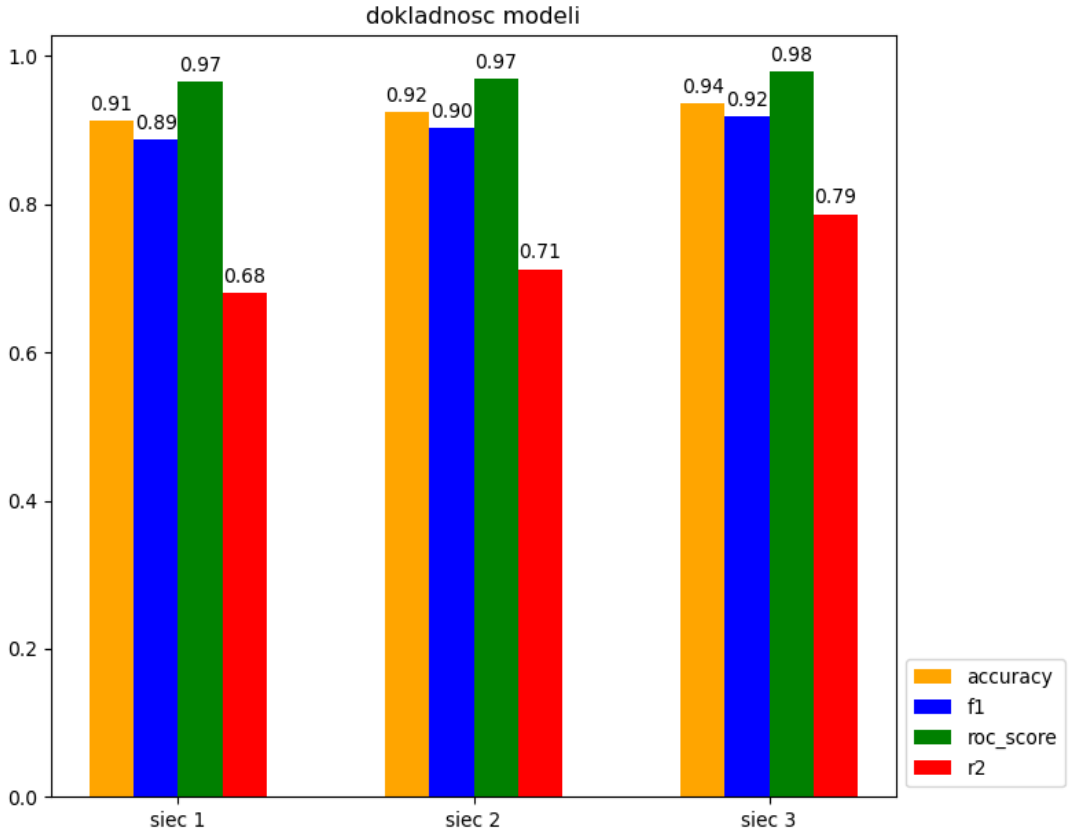
```
"""
siec z jedna warstwa ukrta
Funkcje aktywacji:
* ReLU
* Sigmoid
"""
network_model_1 = nn1.NeuralNetwork(input_size, hidden_size_1: 4,  output_size: 1,
                                    nn.ReLU(), nn.Sigmoid())
network_model_2 = nn1.NeuralNetwork(input_size, hidden_size_1: 8,  output_size: 1,
                                    nn.ReLU(), nn.Sigmoid())
network_model_3 = nn1.NeuralNetwork(input_size, hidden_size_1: 32,  output_size: 1,
                                    nn.ReLU(), nn.Sigmoid())
```

Dla funckji aktywacji:

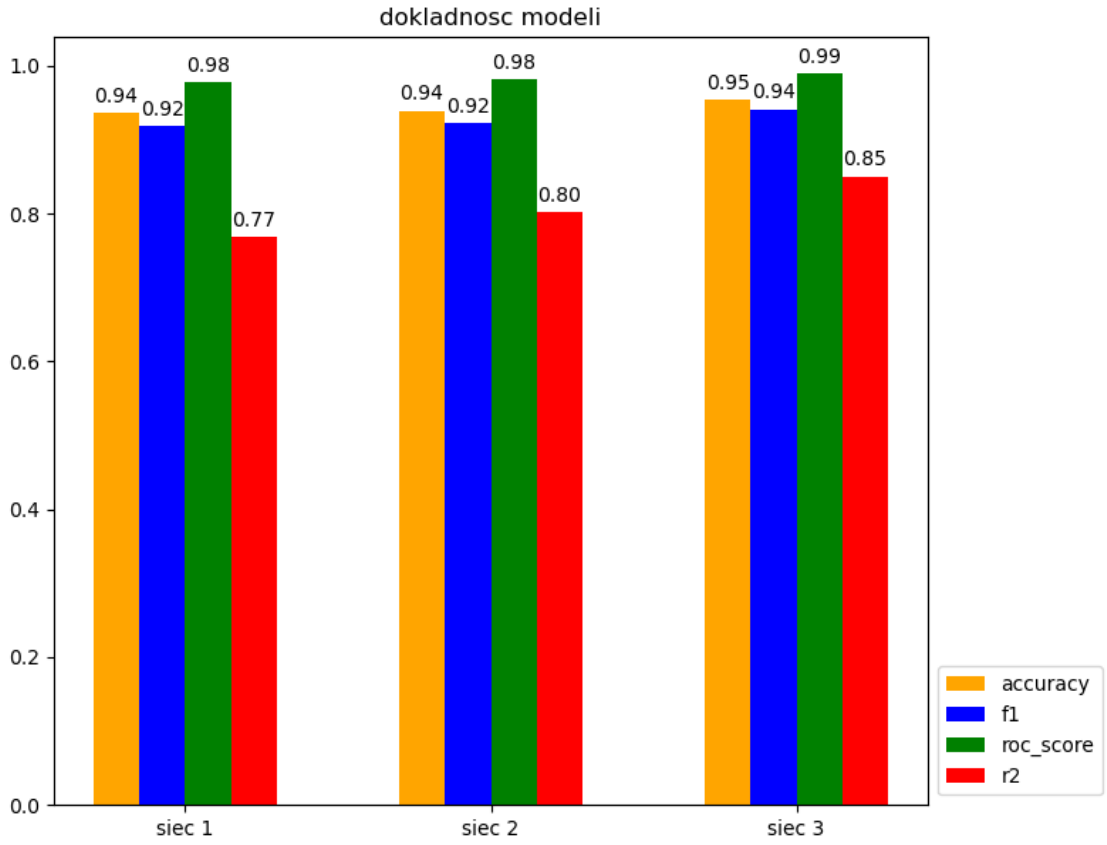
- 1 warstwa – ReLU,
- wyjście - Sigmoid

	ETA		20 epok	100 epok	1000 epok	2000 epok
Sieć 1	0.01	Accuracy	0.91	0.94	0.94	0.94
		F1	0.89	0.92	0.93	0.93
		ROC	0.97	0.98	0.98	0.98
		R2	0.68	0.77	0.81	0.81
		Czas (sek)	0.19	0.69	5.13	10.58
Sieć 2		Accuracy	0.92	0.94	0.94	0.94
		F1	0.90	0.92	0.92	0.92
		ROC	0.97	0.98	0.98	0.98
		R2	0.71	0.80	0.79	0.78
		Czas (sek)	0.19	0.71	6.87	14.01
Sieć 3		Accuracy	0.94	0.95	0.97	0.97
		F1	0.92	0.94	0.96	0.96
		ROC	0.98	0.99	0.99	0.99
		R2	0.79	0.85	0.87	0.87
		Czas (sek)	0.2	0.92	13.64	34.51

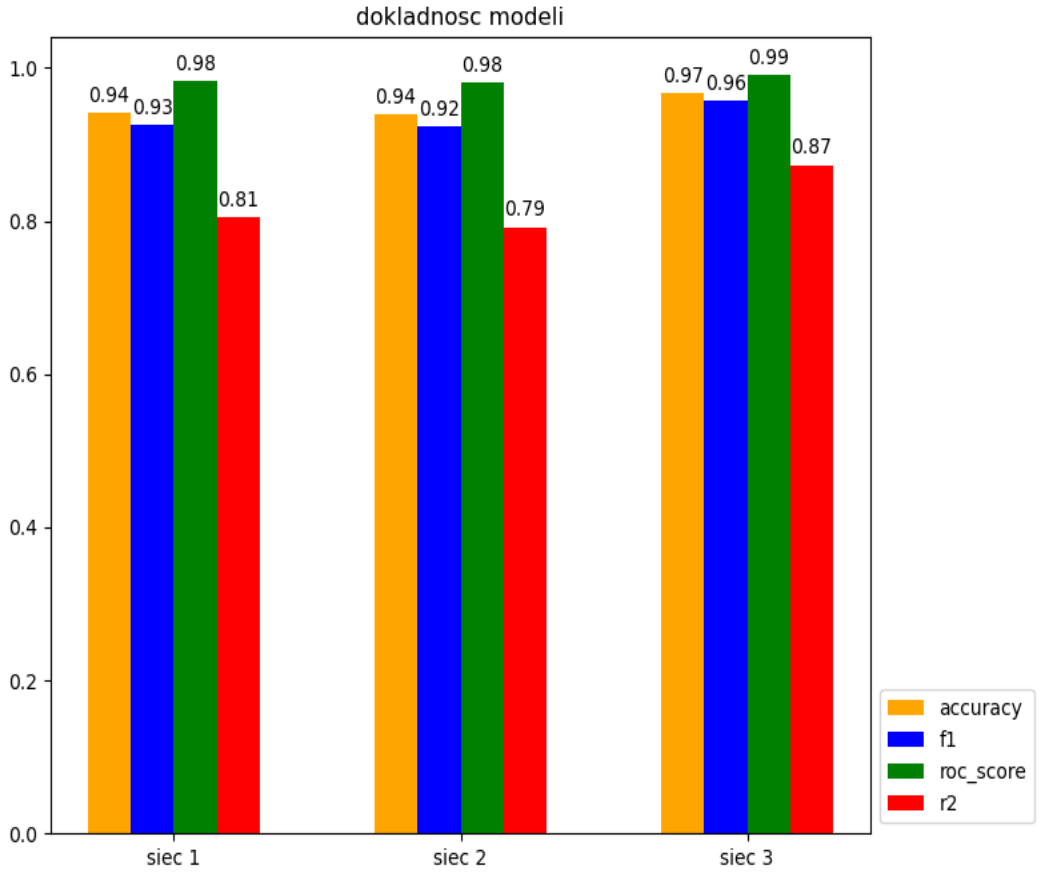
20 epok – Eta 0.01 - 1 warstwa ukryta (Funkcja aktywacji: ReLU, Sigmoid)



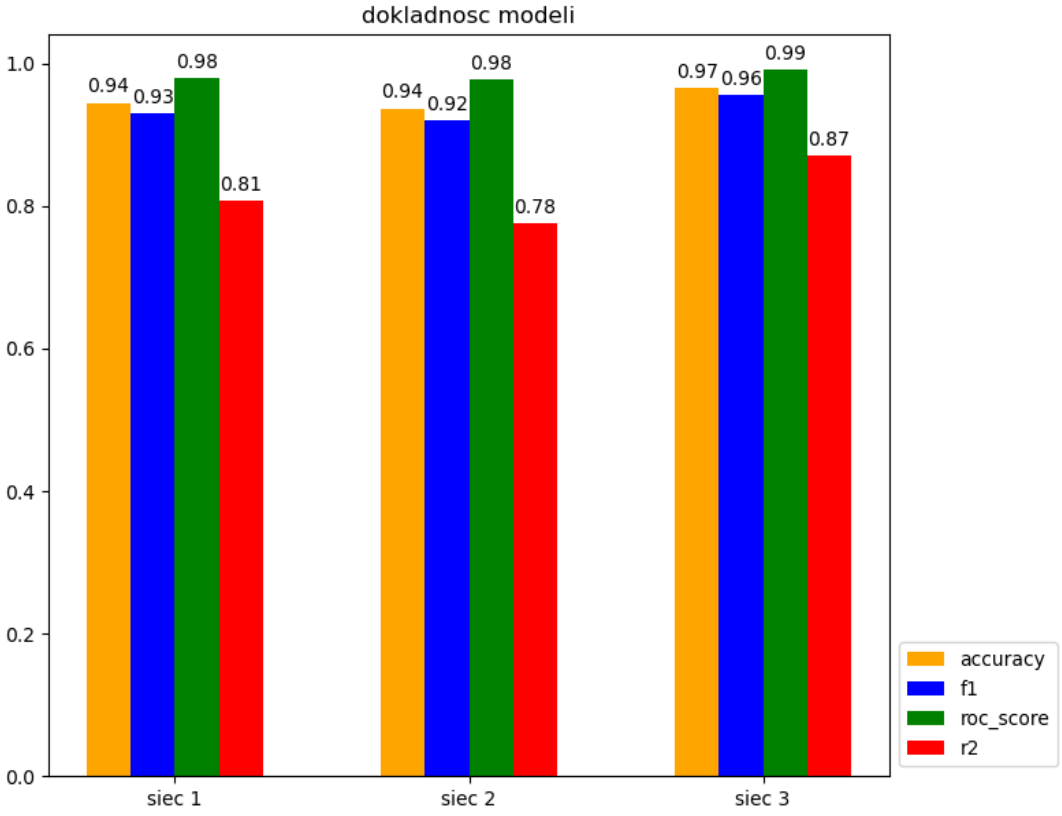
100 epok – Eta 0.01 - 1 warstwa ukryta (Funkcja aktywacji: ReLU, Sigmoid)



1000 epok – Eta 0.01 - 1 warstwa ukryta (Funkcja aktywacji: ReLU, Sigmoid)



2000 epok – Eta 0.01 - 1 warstwa ukryta (Funkcja aktywacji: ReLU, Sigmoid)



Sieć 3 osiągnęła najlepszy wynik dokładności dla wszystkich dotychczas testowanych modeli. Sieć jest bardzo podobna do poprzednia z dwiema warstwami. Też posiada wysoką wartość ROC która oznacza że dobrze sobie radzi z rozróżnianiem klas. Tak samo jak poprzednia sieć pierwsza i druga, bardzo szybko osiąga dobry rezultat pod względem jakości i czasu.

Wszystkie dotychczas modele były testowane na funkcji strat **BCELoss** (głownie stosowana właśnie do przypadków binarnych), tak samo algorytmem optymalizacji był algorytm **'Adam'**. Dlatego poniżej postaram się przetestować najlepszą sieć pod względem innej funkcji start oraz algorytmu optymalizacji.

```
criterion = nn.MSELoss()

optimizer_1 = optim.SGD(network_model_1.parameters(), lr=learning_rate)
optimizer_2 = optim.SGD(network_model_2.parameters(), lr=learning_rate)
optimizer_3 = optim.SGD(network_model_3.parameters(), lr=learning_rate)
```

Funkcja strat: MSELoss (różnica do kwadratu)

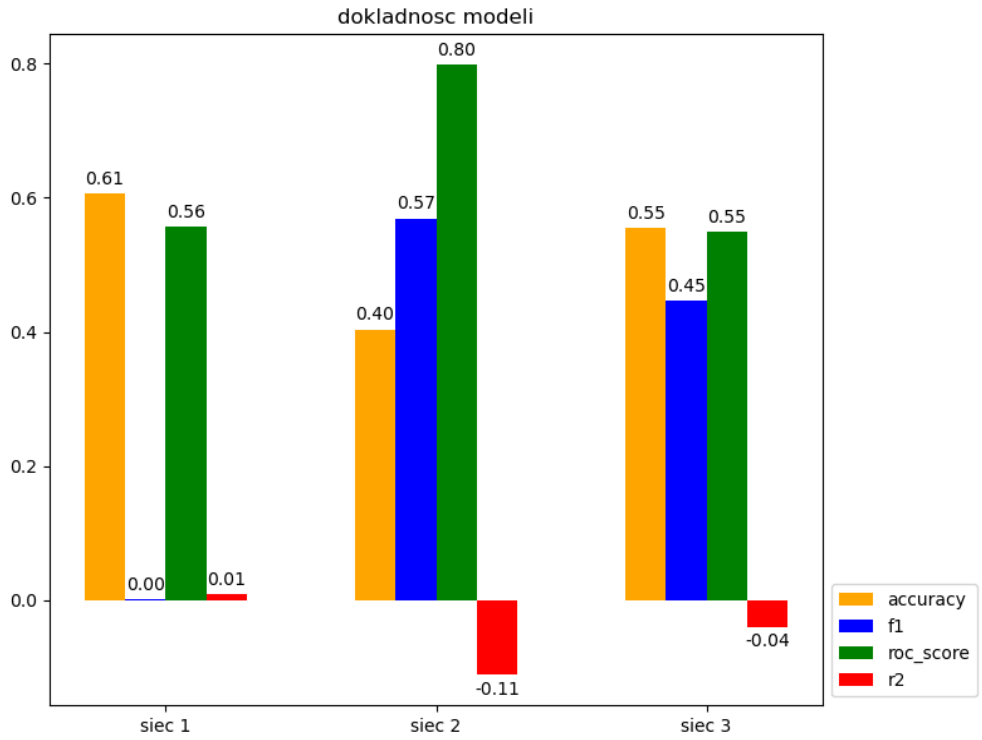
Algorytm optymalizacji: SGD (gradient)

Dla funkcji aktywacji:

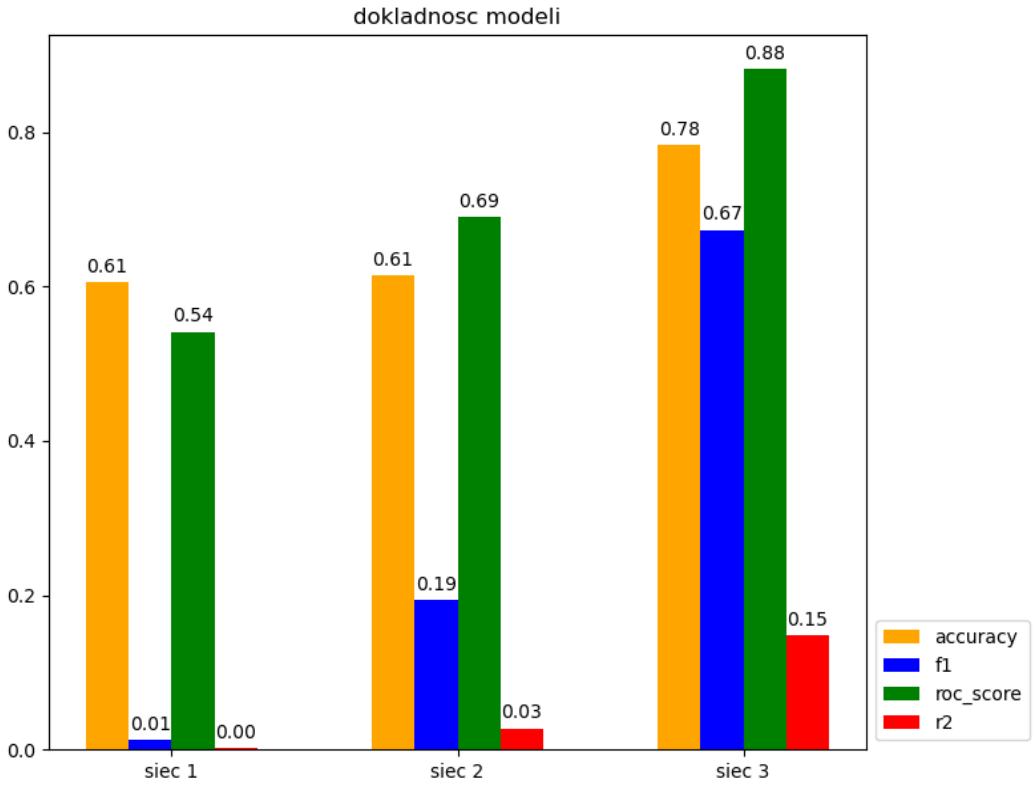
- 1 warstwa – ReLU,
- wyjście - Sigmoid

	ETA		20 epok	100 epok	1000 epok	2000 epok
Sieć 1	0.01	Accuracy	0.61	0.61	0.88	0.91
		F1	0.00	0.01	0.85	0.89
		ROC	0.56	0.54	0.94	0.96
		R2	0.01	0.00	0.55	0.68
		Czas (sek)	0.16	0.58	4.0	7.74
Sieć 2		Accuracy	0.40	0.61	0.90	0.82
		F1	0.57	0.19	0.86	0.89
		ROC	0.80	0.69	0.95	0.96
		R2	-0.11	0.03	0.60	0.68
		Czas (sek)	0.15	0.62	5.89	11.37
Sieć 3		Accuracy	0.55	0.78	0.90	0.91
		F1	0.45	0.67	0.87	0.89
		ROC	0.55	0.88	0.95	0.96
		R2	-0.04	0.15	0.59	0.69
		Czas (sek)	0.15	0.75	6.92	13.85

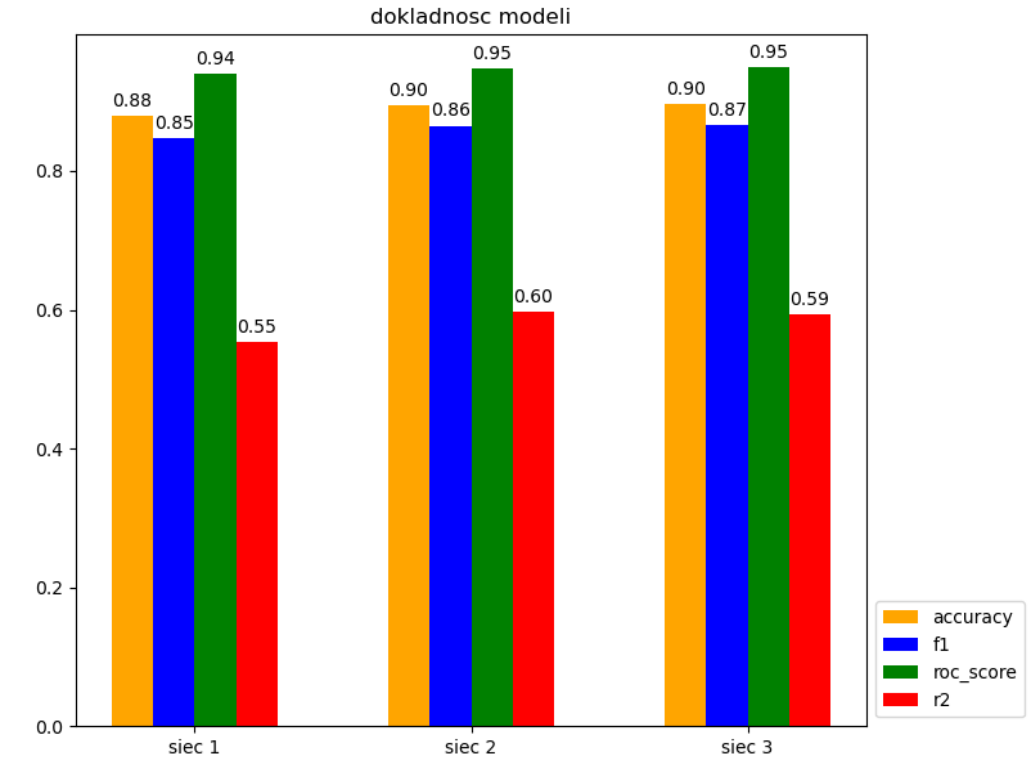
20 epok – Eta 0.01 - 1 warstwa ukryta (Funkcja aktywacji: ReLU, Sigmoid)
Algorytm: SGD, Funkcja strat: MSELoss



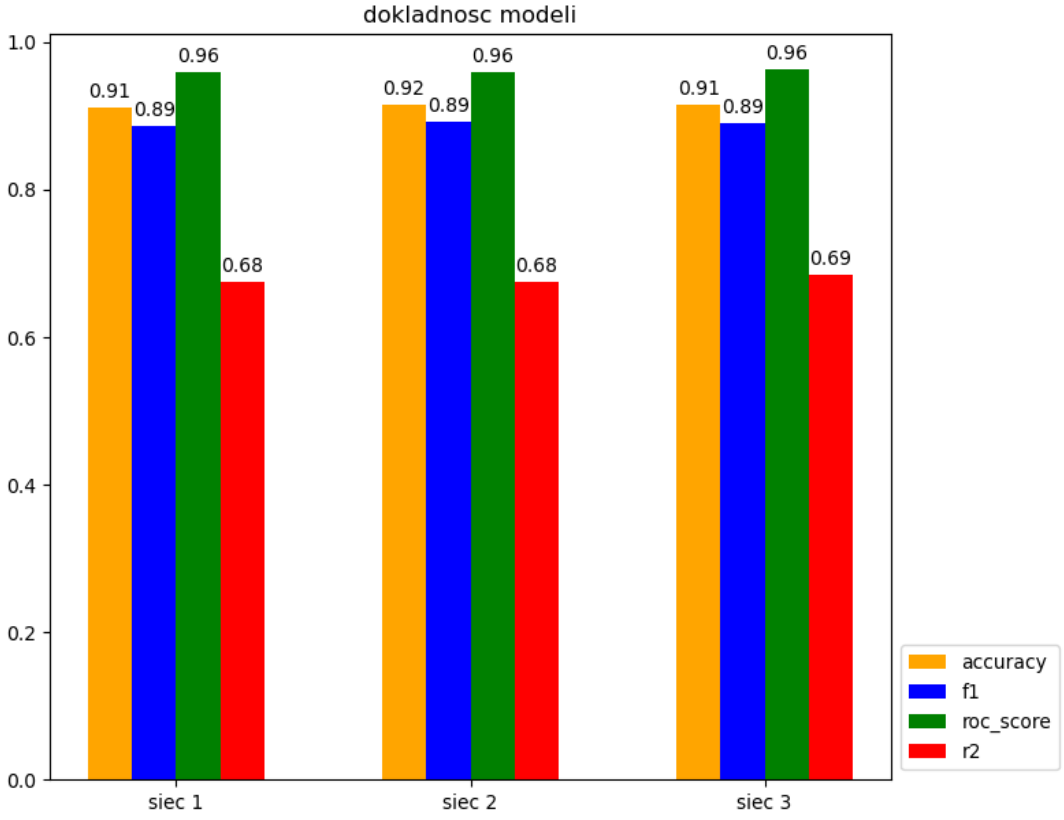
100 epok – Eta 0.01 - 1 warstwa ukryta (Funkcja aktywacji: ReLU, Sigmoid)
Algorytm: SGD, Funkcja strat: MSELoss



1000 epok – Eta 0.01 - 1 warstwa ukryta (Funkcja aktywacji: ReLU, Sigmoid)
Algorytm: SGD, Funkcja strat: MSELoss



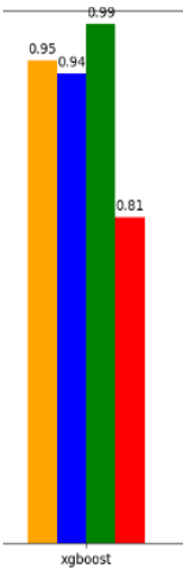
2000 epok – Eta 0.01 - 1 warstwa ukryta (Funkcja aktywacji: ReLU, Sigmoid)
Algorytm: SGD, Funkcja strat: MSELoss



Sieć 1 jest najlepsza po 2000 epokach, osiągając najwyższe wartości metryk. Sieć 2 pokazuje dużą zmienność, ale osiąga dobre wyniki po 1000 epokach. Sieć 3 ma stabilny wzrost, ale nie osiąga najwyższych wyników jak Sieć 1. Większa liczba epok jest kluczowa dla osiągnięcia wysokich wyników przy użyciu **MSELoss** i **SGD**. Zmiana algorytmu i funkcji nie wpływa na lepsze, sieć co prawda w miarę dobrze się nauczy około 92% dokładności, natomiast dopiero po tysiącu epok. Poprzednie metody były bardziej wydajne pod względem czasu oraz dokładności. Jak możemy zobaczyć sieć dla 20 i 100 epok osiąga słabe rezultaty. Metryka F1 jest bardzo słaba, więc model bardzo słabo waliduje klasy. Przy zastosowaniu algorytmy optymalizacji SGD, oraz funkcji strat MSELoss, możemy zauważyć charakterystyczną rzecz, a mianowicie każda z sieci uzyskuje bardzo zbliżone wartości, bardziej niż to było w poprzednich modelach sieci. Przykład dla R2 przy dwóch tysiącach epok, gdzie wartości się delikatnie różnią, jakbyśmy to porównali do poprzednich modeli to różnica w poprzednich jest większa. Czasami przeskakiwała nawet o wartość równą 0.1, tutaj jest różnica tylko dla sieci 3 i to dosłownie o 0.01, jest to bardzo mało.

Podsumowując po odpowiedniej walidacji danych najlepiej działa sieć dla następujących parametrów:

* 1 warstwa ukryta
* Funkcja aktywacji (ReLU – dla wartstwy ukrytej, oraz Sigmoid dla wyjścia)
* W warstwie ukrytej 32 neurony
* Wyjście 1
* KFold = 5
* Funkcja straty: BCELoss
* Algorytm optymalizacji: Adam
* Ilość epok: 1000



Walidacja krzyżowa KFold gdzie K = 5 dla modelu XGBoost, była najlepszym otrzymanym wynikiem do tej chwili. Jak można zauważyć obok po lewej stronie, aktualnie do tego modelu w miarę dorasta sieć opisana powyżej. Jest bardzo podobno jeżeli chodzi o jej dokładność. Jedyna różnica jaka jest to dla krzywej ROC gdzie model XGBoost wypada o 0.01 lepie. Natomiast pozostałe parametry są takie same.

Podsumowując wszystkie modele i sieci do tego momentu, to powyższe dwa modele reprezentują się najlepiej.

Testowanie wycieku danych

```
scaler = StandardScaler()
#scaled_data = scaler.fit_transform(dataset)
dataset[std_columns] = scaler.fit_transform(dataset[std_columns])
print(dataset.head())
```

Zastosowanie skalowania danych przed ich podziałem może prowadzić do wycieku danych. Dlatego poniżej dam porównania z użyciem skalowania przed podziałem jak i po podziale.

Testy z potencjalnym wyciekiem danych:

```
średnia dla całego zbioru: 1.235457919165573e-17
Średnia dla zbioru treningowego: -0.013252780938278948
Średnia dla zbioru testowego: 0.019868373378408456
średnia po skalowaniu dla zbioru treningowego: -2.0852884636437723e-16
średnia po skalowaniu dla zbioru testowego: 0.051138200250916796
Odchylenie standardowe dla:
* oryginalny zbiór: 1.0001086897454596
* zbiór treningowy: 0.6476793112423511
* zbiór testowy: 1.3673464137522908
* zbiór treningowy po skalowaniu: 0.999999999999998
* zbiór testowy po skalowaniu: 2.1111472761566246
```

Testy bez wycieku danych:

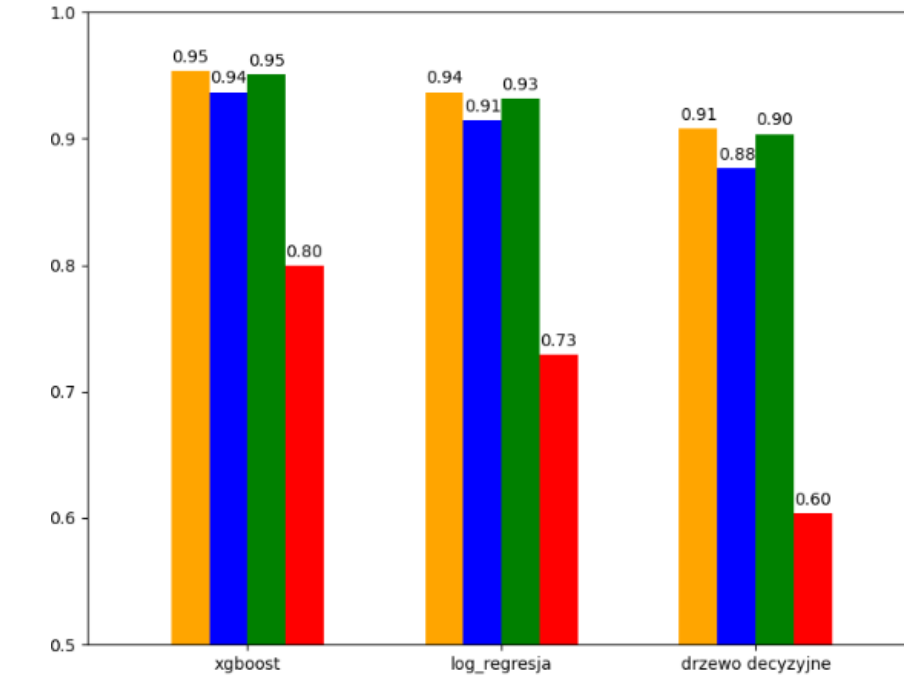
```
średnia dla całego zbioru: 52.17278852423386
Średnia dla zbioru treningowego: 50.63659420289855
Średnia dla zbioru testowego: 54.475828354155354
średnia po skalowaniu dla zbioru treningowego: 6.436075505073371e-18
średnia po skalowaniu dla zbioru testowego: 0.029095282148371687
Odchylenie standardowe dla:
* oryginalny zbiór: 194.89130952646204
* zbiór treningowy: 131.95383814044496
* zbiór testowy: 262.2834329987646
* zbiór treningowy po skalowaniu: 1.000000000000004
* zbiór testowy po skalowaniu: 1.98769082199492
```

Bardzo podobne średnie przed skalowaniem danych mogą sugerować że dane mogłyby być skalowane przed podziałem na dane testowy i treningowe. Dodatkowo większe odchylenie standardowe danych po skalowaniu może również sugerować że dane były skalowane przed podziałem.

Poniżej porównam kilka modeli bez wycieku danych oraz z potencjalnym wyciekiem danych.

List porównywaych modeli:
XGBClassifier, LogisticRegression, DecisionTreeClassifier, RandomForestClassifier, SVC, KNeighborsClassifier, GradientBoostingClassifier, Sieć neuronowa

Wykres dla 20% danych testowych i 80% danych treningowych



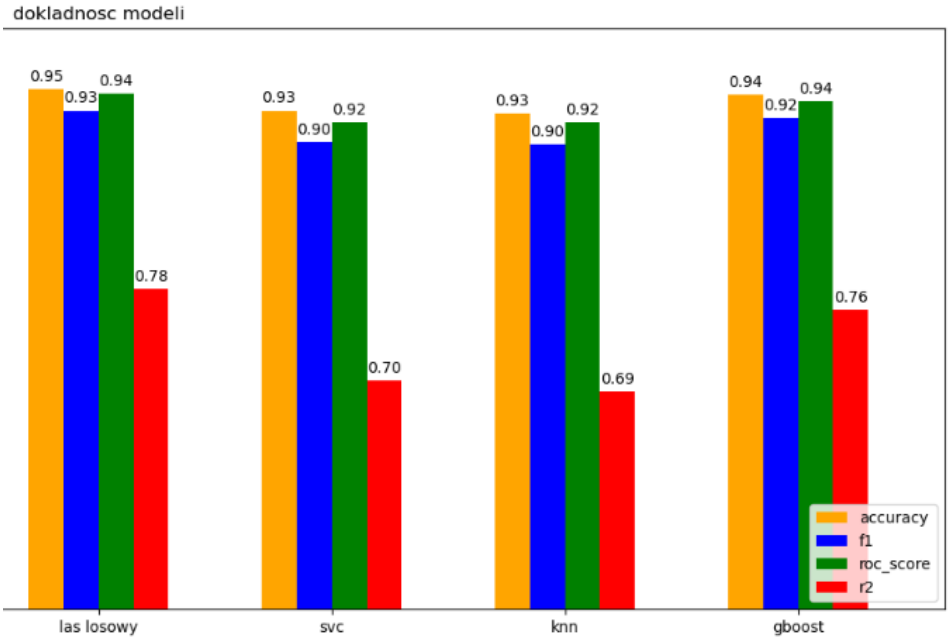
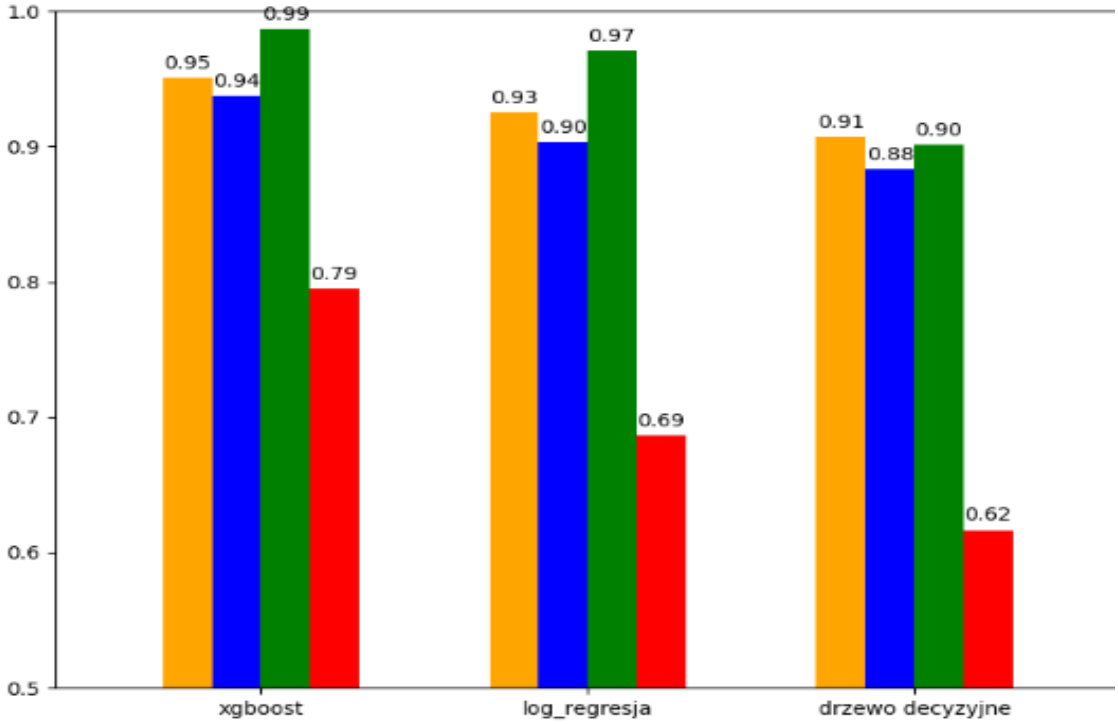


Tabela porównująca metryki wydajności modeli z możliwym wyciekiem jak i bez wycieku danych. Dla 20% danych testowych oraz 80% danych treningowym.

	metryki	POTENCJALNY WYCIEK	BEZ WYCIEKU DANYCH
xgboost	accuracy	0.948969	0.953312
	f1	0.932568	0.93
	roc	0.945021	0.950795
	r2	0.783403	0.799519
Log_regressja	accuracy	0.92291	0.937025
	f1	0.896047	0.914454
	roc	0.9135	0.931786
	r2	0.6728	0.729584
Drzewo decyzyjne	accuracy	0.921824	0.907709
	f1	0.896552	0.876633
	roc	0.915942	0.90367
	r2	0.668191	0.603701
Las losowy	accuracy	0.940282	0.947883
	f1	0.918759	0.929412
	roc	0.930275	0.944052
	r2	0.746535	0.776207
svc	accuracy	0.934853	0.929425
	f1	0.912281	0.900446
	roc	0.92645	0.919662
	r2	0.723493	0.696947
knn	accuracy	0.904452	0.927253
	f1	0.873199	0.900446
	roc	0.896955	0.919771
	r2	0.594456	0.687623
gboost	accuracy	0.938111	0.94354
	f1	0.916545	0.922849
	roc	0.92963	0.937559
	r2	0.736318	0.757558

Jeżeli wystąpił wyciek danych to był on niewielki. Jest on mało zauważalny. Jak widać na powyższej tabeli w niektórych sytuacjach dane z kolumny ‘BEZ WYCIEKU DANYCH’ mają lepsze metryki, co nie powinno tak być w sytuacji gdy byłbym wyciek danych. Zazwyczaj gdy jest wyciek danych kolumna ‘POTENCJALNY WYCIEK’ powinna mieć lepsze wyniki, natomiast tak nie jest a jak już jest to tylko dla niektórych modeli.

Porównanie modeli na KFold równym 5 z wyciekiem i bez wycieku danych:



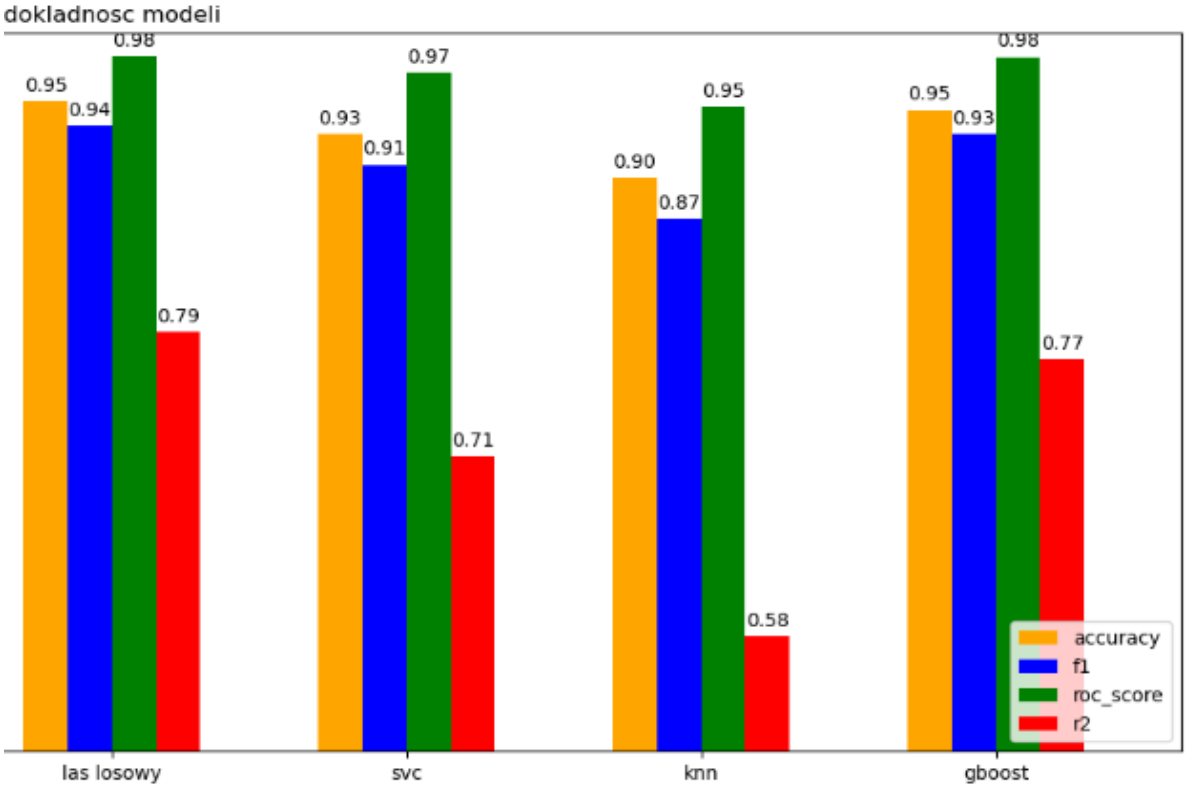


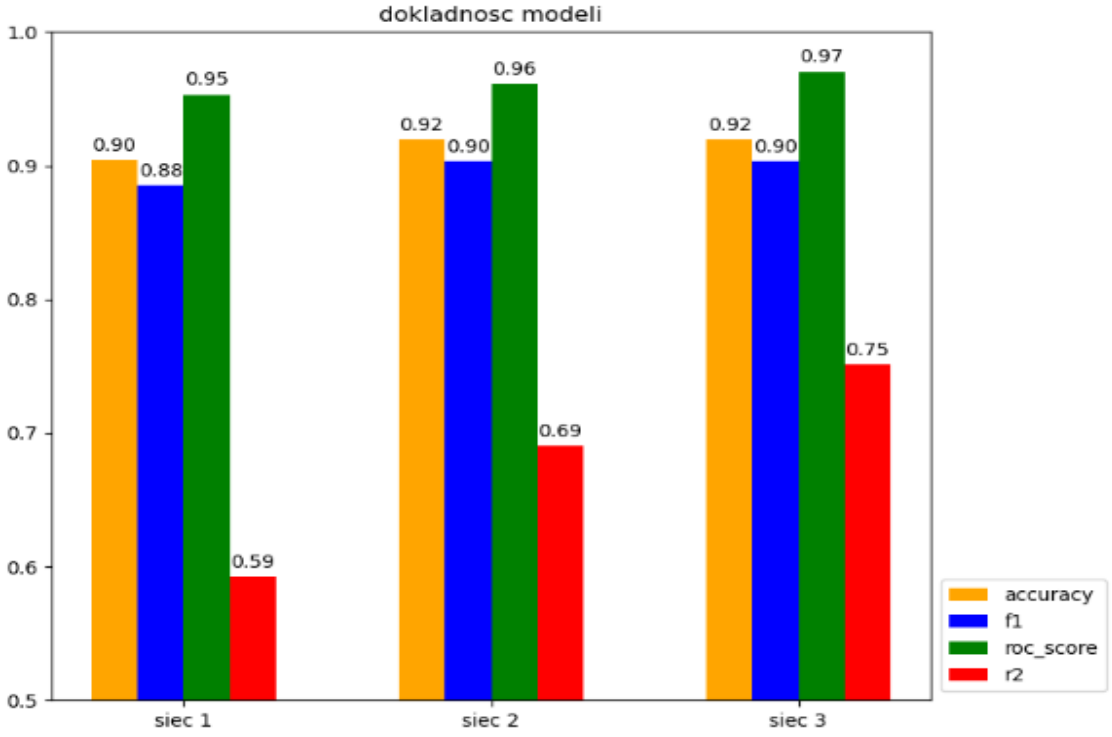
Tabela porównująca metryki wydajności modeli z możliwym wyciekiem jak i bez wycieku danych dla metody KFold = 5.

	metryki	POTENCJALNY WYCIEK	BEZ WYCIEKU DANYCH
Xgboost	accuracy	0.953922	0.951097
	f1	0.941066	0.937151
	roc	0.987573	0.98683
	r2	0.806638	0.784813
Log regresja	accuracy	0.924583	0.925236
	f1	0.902305	0.902911
	roc	0.915013	0.97069
	r2	0.68391	0.686728
Drzewo decyzyjne	accuracy	0.918497	0.907195
	f1	0.89588	0.88361
	roc	0.915013	0.901682
	r2	0.643669	0.616574
Las losowy	accuracy	0.95327	0.952403
	f1	0.939684	0.935529
	roc	0.986362	0.984324
	r2	0.802074	0.792154
SVC	accuracy	0.931319	0.929798
	f1	0.910702	0.908443
	roc	0.974262	0.872417
	r2	0.711983	0.705588
KNN	accuracy	0.908719	0.899807
	f1	0.882196	0.870575
	roc	0.951407	0.948504
	r2	0.617602	0.580168
Gboost	accuracy	0.945881	0.946534
	f1	0.930307	0.930077
	roc	0.985081	0.984322
	r2	0.774924	0.772341

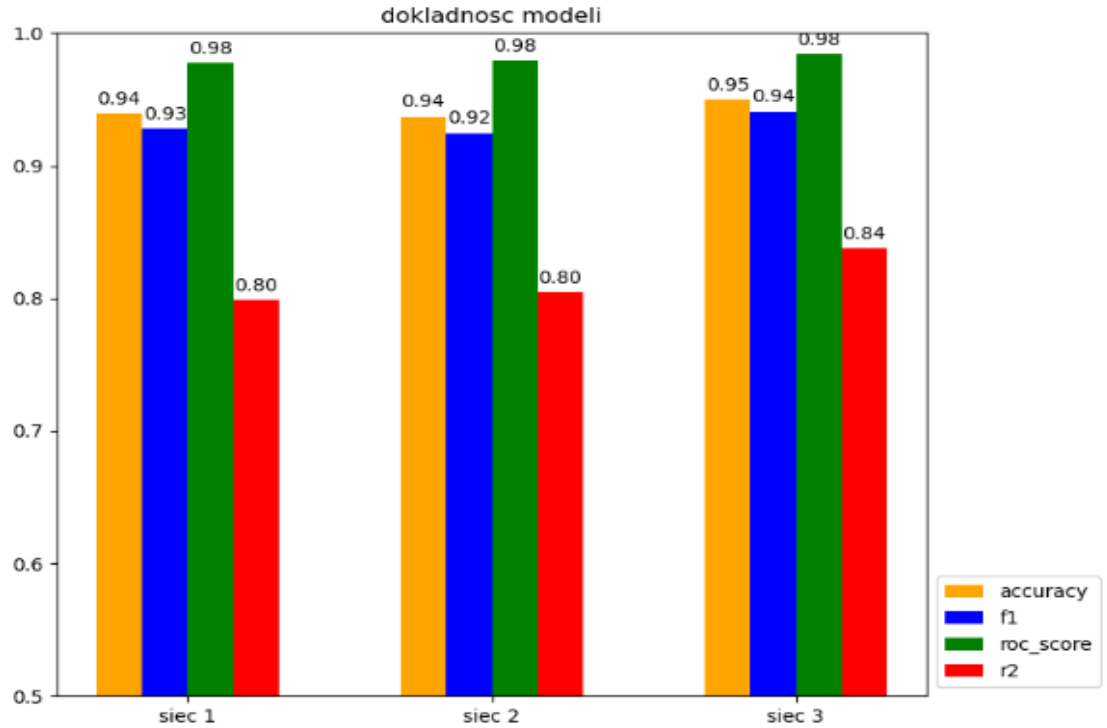
Uważam że tutaj delikatnie lepiej można zobaczyć wyciek danych, ponieważ dokładność dla kolumny ‘POTENCJALNY WYCIEK’ jest w większości minimalnie większa co mogło by faktycznie sugerować że był wyciek danych.

Testowanie na danych bez wycieku na sieciach neuronowych (te same wagi)

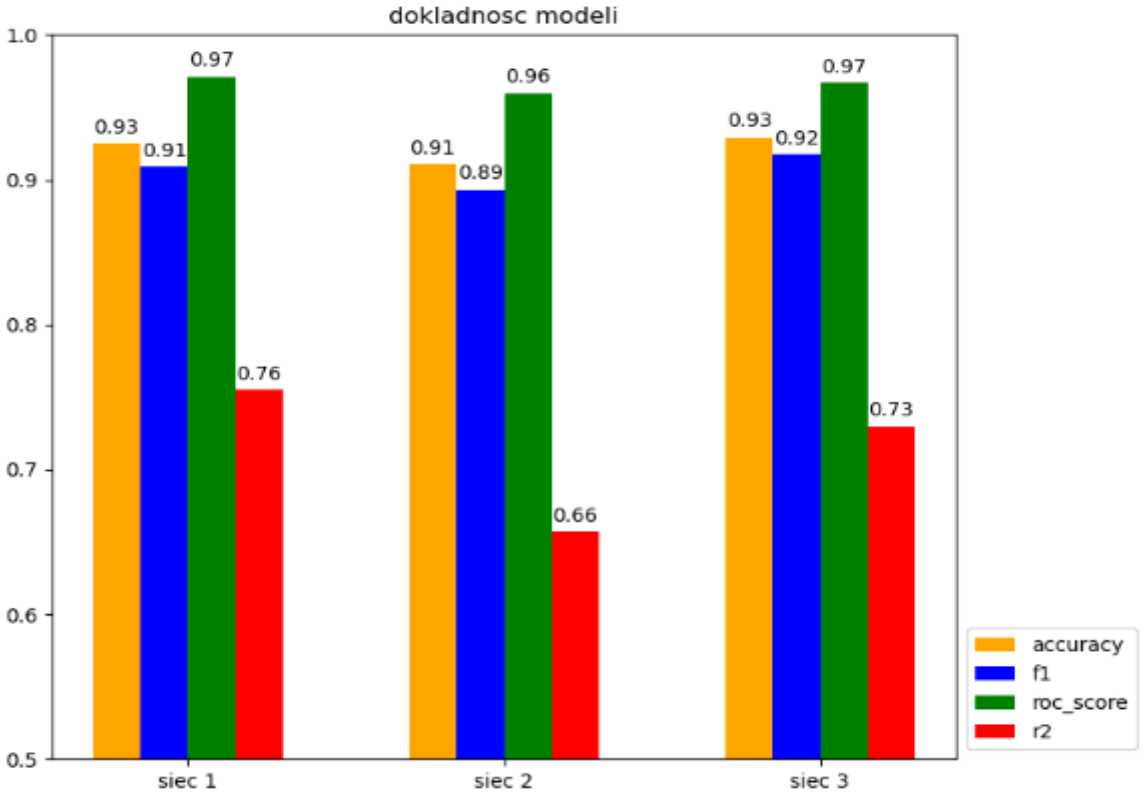
20 epok, dwie warstwy ukryte, 20% dane testowe, 80% dane treningowe funkcja aktywacji (Tanh, Tanh, Sigmoid), eta = 0.01 (te same wagi)



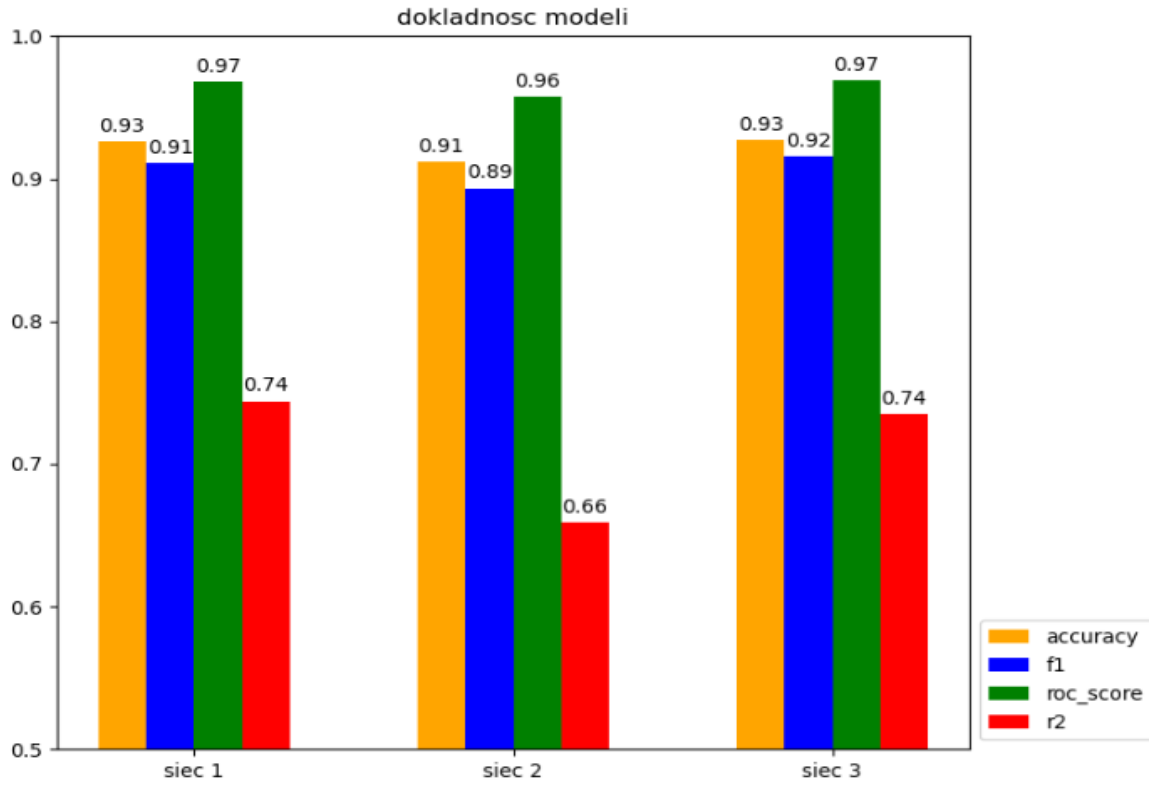
100 epok, dwie warstwy ukryte, 20% dane testowe, 80% dane treningowe funkcja aktywacji (Tanh, Tanh, Sigmoid), eta = 0.01 (te same wagi)



1000 epok, dwie warstwy ukryte, 20% dane testowe, 80% dane treningowe funkcja aktywacji (Tanh, Tanh, Sigmoid), eta = 0.01 (te same wagi)



2000 epok, dwie warstwy ukryte, 20% dane testowe, 80% dane treningowe funkcja aktywacji (Tanh, Tanh, Sigmoid), eta = 0.01 (te same wagi)



	Siec 1	*Siec 2*	*Siec 3*
Funkcja strat	BCELoss	BCELoss	BCELoss
Algorytm optymalizacji	Adam	Adam	Adam
Funkcja aktywacji warstwa 1	Tanh	Tanh	Tanh
Funkcja aktywacji warstwa 2	Tanh	Tanh	Tanh
Funkcja aktywacji wyjście	Sigmoid	Sigmoid	Sigmoid

	ETA		**20 epok**	**100 epok**	**1000 epok**	**2000 epok**
Sieć 1	0.01	Accuracy	0.90	0.94	0.93	0.93
		F1	0.88	0.93	0.91	0.91
		ROC	0.95	0.98	0.97	0.97
		R2	0.59	0.80	0.76	0.74
		Czas (sek)	0.05	0.20	1.87	3.90
Sieć 2		Accuracy	0.92	0.94	0.91	0.91
		F1	0.90	0.92	0.89	0.89
		ROC	0.96	0.98	0.96	0.96
		R2	0.69	0.80	0.66	0.66
		Czas (sek)	0.04	0.20	2.09	4.25
Sieć 3		Accuracy	0.92	0.95	0.93	0.93
		F1	0.90	0.94	0.92	0.92
		ROC	0.97	0.98	0.97	0.97
		R2	0.75	0.84	0.73	0.74
		Czas (sek)	0.16	0.33	5.87	11.55

Każda z sieci osiąga najlepszy rezultat po około 100 epokach. Dla tysiąca i dwóch tysięcy epok sieć nie poprawia się, nawet delikatnie dokładność spada, dodatkowo czas trwania jest znacznie większy niż dal 100 epok. Najlepszym wyborem w tej sytuacji będzie wybranie 100 epok oraz sieci o numerze 3, ponieważ ma najlepsza dokładność. **Porównując do poprzednich danych** gdzie mógłby być wyciek danych, wygląda to bardzo podobnie, przy 100 epokach sieć osiąga maksymalna jakość, wartości z poprzednich testów są minimalnie mniejsze.

Trenowanie sieci na tych samych wagach

Wagi dla Sieci 1

```
Layer: fc1.weight | Weights: tensor([[ 0.1022,  0.1109, -0.0313,  0.1228, -0.0293,  0.0270, -0.0651,  0.0785,
          0.1178, -0.0980,  0.1162,  0.0250,  0.0987,  0.0181,  0.0644, -0.0189,
          0.1030,  0.0198, -0.0624,  0.0341, -0.0616, -0.0157, -0.0543,  0.0886,
         -0.1055, -0.0616, -0.0377, -0.0803,  0.0126, -0.1320,  0.1207, -0.1135,
          0.1032,  0.0222, -0.0434,  0.0826,  0.0208,  0.1080,  0.0146, -0.0421,
          0.0359, -0.0362,  0.0562,  0.1193,  0.0772, -0.0584,  0.0771,  0.0239,
          0.0679, -0.0814, -0.1323, -0.0516, -0.1025,  0.1096,  0.0385,  0.0554],
```

Wagi dla Sieci 2

```
Layer: fc1.weight | Weights: tensor([[ 0.1022,  0.1109, -0.0313,  0.1228, -0.0293,  0.0270, -0.0651,  0.0785,
          0.1178, -0.0980,  0.1162,  0.0250,  0.0987,  0.0181,  0.0644, -0.0189,
          0.1030,  0.0198, -0.0624,  0.0341, -0.0616, -0.0157, -0.0543,  0.0886,
         -0.1055, -0.0616, -0.0377, -0.0803,  0.0126, -0.1320,  0.1207, -0.1135,
          0.1032,  0.0222, -0.0434,  0.0826,  0.0208,  0.1080,  0.0146, -0.0421,
          0.0359, -0.0362,  0.0562,  0.1193,  0.0772, -0.0584,  0.0771,  0.0239,
          0.0679, -0.0814, -0.1323, -0.0516, -0.1025,  0.1096,  0.0385,  0.0554],
```

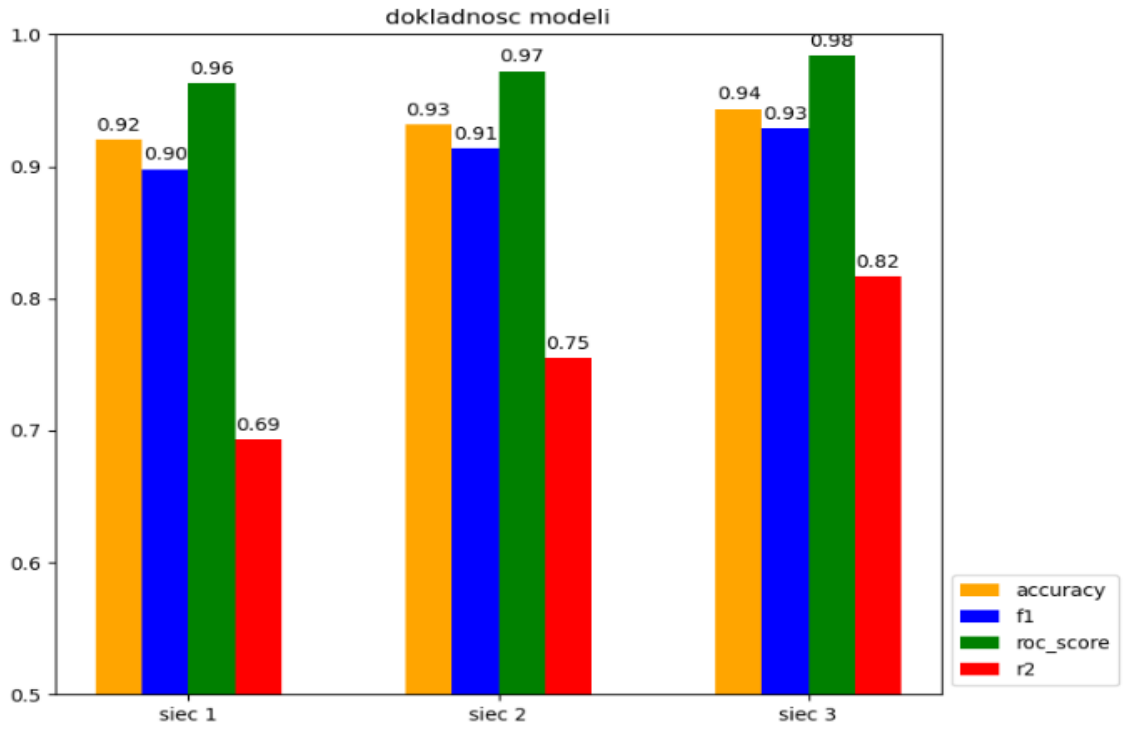
Powyżej przedstawiłem dwie grafiki dla różnych sieci aby potwierdzić że wagi wejściowe od teraz są za każdym razem takie same. Przed każdym utworzeniem instancji sieci ustawiam **seed** (ziarno generowania licz) na taką samą liczbę, dzięki czemu za każdym razem generator losuje te same liczby. Poniżej przedstawiam metodę do resetowania.

```
def set_seed(seed):
    random.seed(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    if torch.cuda.is_available():
        torch.cuda.manual_seed(seed)
        torch.cuda.manual_seed_all(seed)
    torch.backends.cudnn.deterministic = True
    torch.backends.cudnn.benchmark = False
```

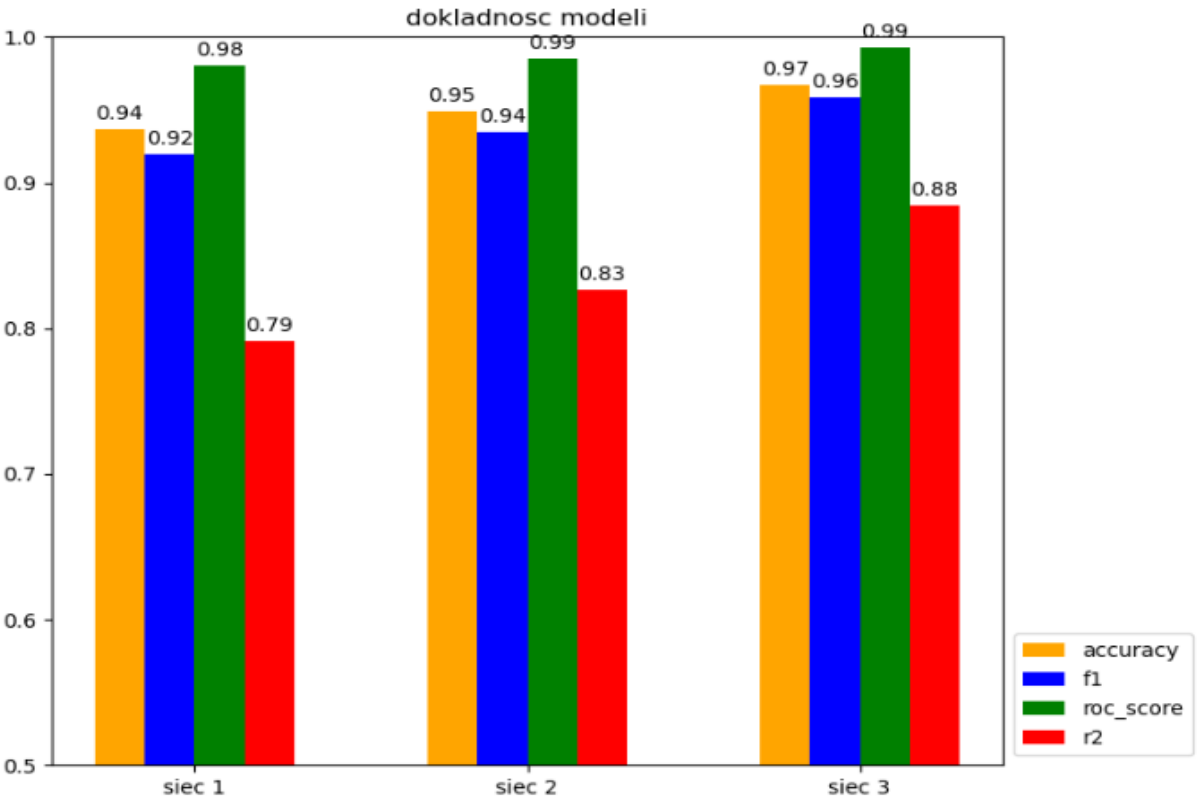
Jest ona wykorzystywana tak jak już wspomniałem przed każdym nowo utworzonym obiektem z sieci. Przykład zastosowania funkcji przedstawiam poniżej na grafice.

```
# resetowanie seed'a (domyślne wagi początkowe)
set_seed(seed)
network_model_2 = nn2.NeuralNetwork(input_size, hidden_size_1: 8, hidden_size_2: 8, output_size: 1,
                                    nn.Tanh(), nn.Tanh(), nn.Sigmoid())
```

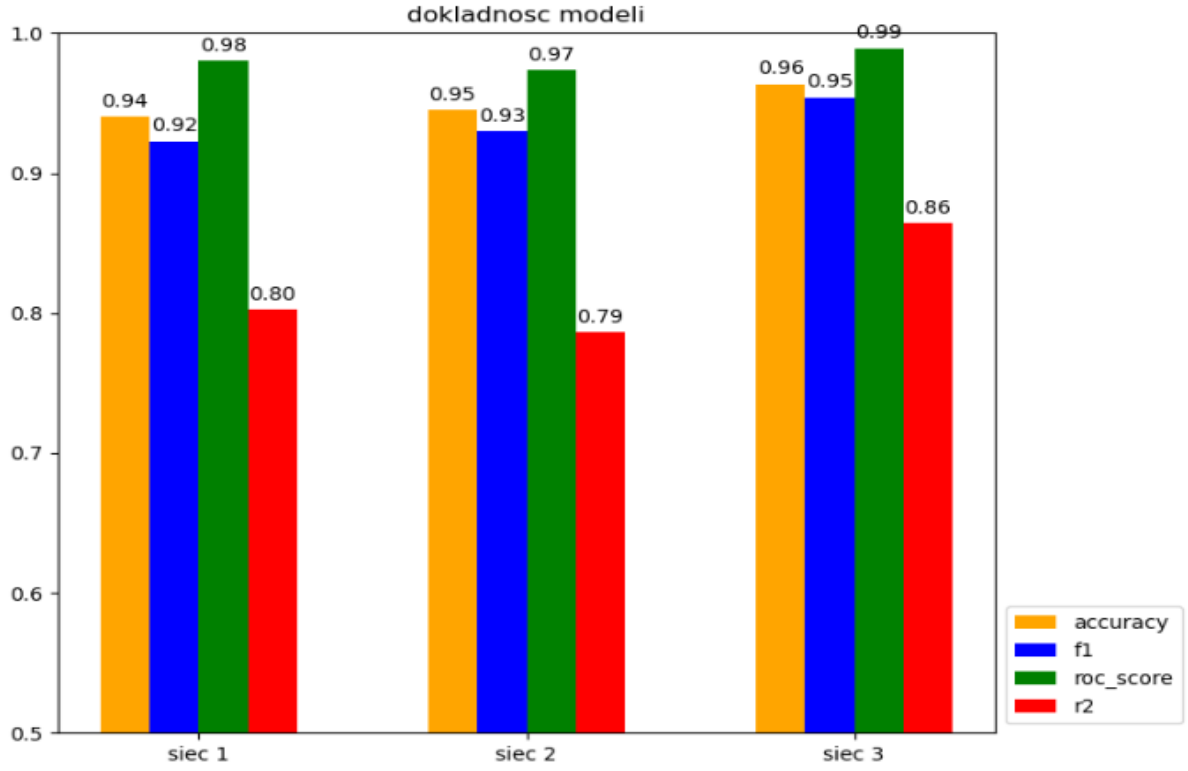
20 epok, dwie warstwy ukryte, KFold = 5
funkcja aktywacji (Tanh, Tanh, Sigmoid), eta = 0.01 (te same wagi)



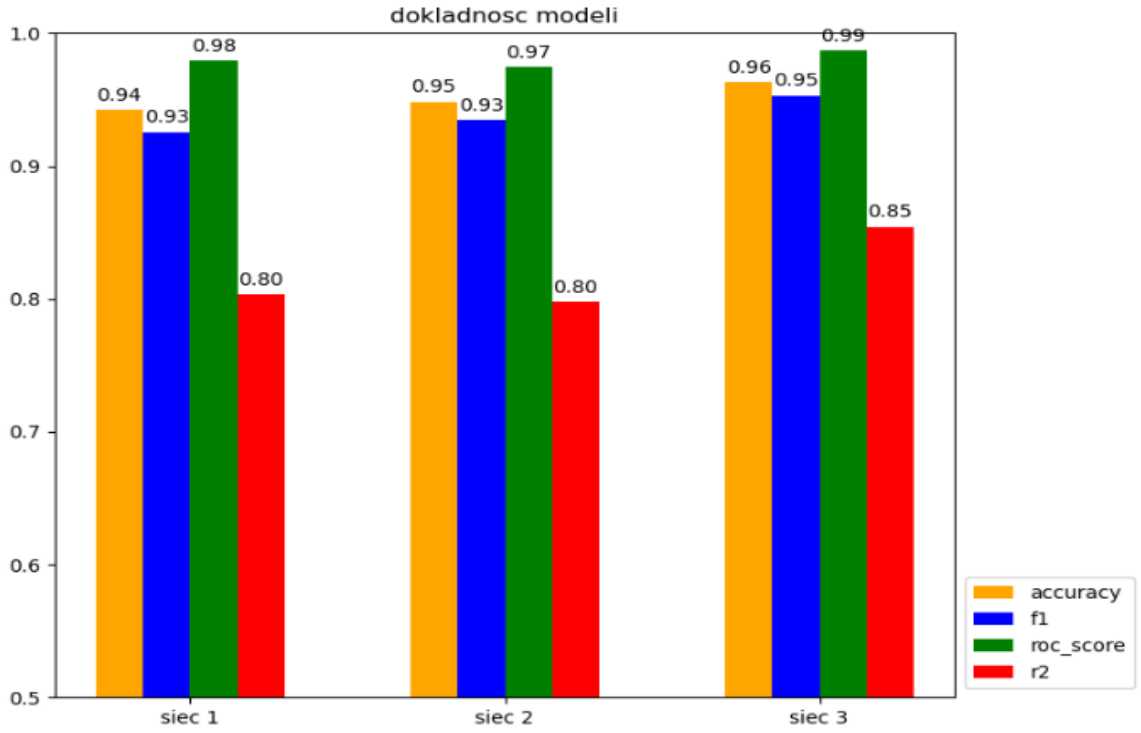
100 epok, dwie warstwy ukryte, KFold = 5
funkcja aktywacji (Tanh, Tanh, Sigmoid), eta = 0.01 (te same wagi)



1000 epok, dwie warstwy ukryte, KFold = 5
funkcja aktywacji (Tanh, Tanh, Sigmoid), eta = 0.01 (te same wagi)



2000 epok, dwie warstwy ukryte, KFold = 5
funkcja aktywacji (Tanh, Tanh, Sigmoid), eta = 0.01 (te same wagi)



	Siec 1	*Siec 2*	*Siec 3*
Funkcja strat	BCELoss	BCELoss	BCELoss
Algorytm optymalizacji	Adam	Adam	Adam
Funkcja aktywacji warstwa 1	ReLU	ReLU	ReLU
Funkcja aktywacji warstwa 2	ReLU	ReLU	ReLU
Funkcja aktywacji wyjście	Sigmoid	Sigmoid	Sigmoid

	ETA		**20 epok**	**100 epok**	**1000 epok**	**2000 epok**
Sieć 1	0.01	Accuracy	0.92	0.94	0.94	0.94
		F1	0.90	0.92	0.92	0.93
		ROC	0.96	0.98	0.98	0.98
		R2	0.69	0.79	0.80	0.80
		Czas (sek)	0.22	0.8	6.56	12.81
Sieć 2		Accuracy	0.93	0.95	0.95	0.95
		F1	0.91	0.94	0.93	0.93
		ROC	0.97	0.99	0.97	0.97
		R2	0.75	0.83	0.79	0.80
		Czas (sek)	0.22	0.93	8.85	17.36
Sieć 3		Accuracy	0.94	0.97	0.96	0.96
		F1	0.93	0.96	0.95	0.95
		ROC	0.98	0.99	0.99	0.99
		R2	0.82	0.88	0.86	0.85
		Czas (sek)	0.28	1.8	24.38	50.96

Sieć przy około 100 epokach osiąga dobrą jakość w stosunkowych krótkim czasie, kolejny epoki np. Dla tysiąca lub dwóch tysięcy epok, sieć nie uzyskuje lepszy rezultatów. Krzywa ROC dla sieci 3 od samego początku pokazuje wysoki współczynnik, to znaczy że nie ma problemu z rozróżnianiem klas. Tak samo miara R2 dla sieci 3 jest też w miarę blisko, co potwierdza że dane są w miarę dobrze dopasowane do modelu.

Podsumowanie

Porównując do poprzednich testów z potencjalnym wyciekiem danych, mogę stwierdzić że ta sieć poradziła sobie lepiej niż poprzednia. Natomiast poprzednia sieć nie była testowana na tych samych wagach. Tutaj możemy zaobserwować jakość równą 0.97 dla sieci 3 która uczyła się 1.8 sekundy. F1 dla tej sieci również jest wysoki, sugerując że sieć dobrze radzi sobie zarówno w pozytywnych przypadkach jak bardzo dobrze w unikaniu fałszywych przypadków.

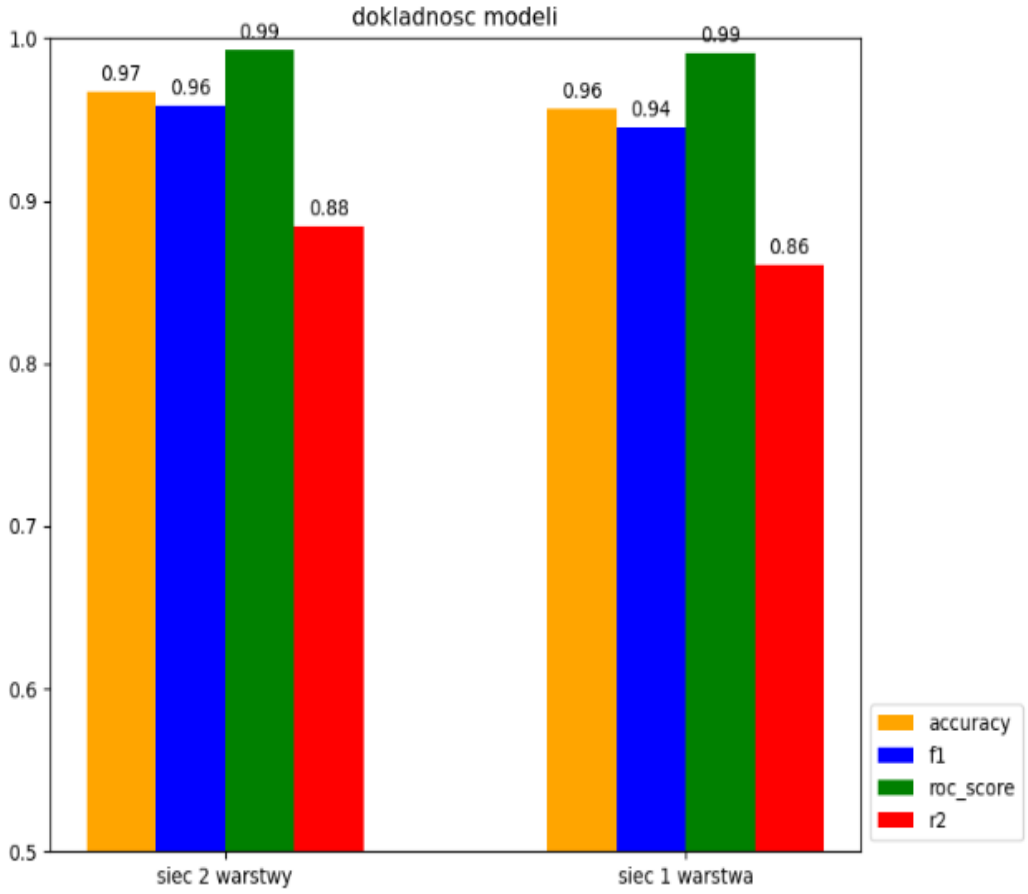
Najlepsza sieć jeżeli dane będą skalowane prawidłowo.

- KFold = 5
- Eta: 0.01
- Epok: 100
- Dwie warstwy ukryte: 1 warstwa (32 neurony) druga warstwa (64 neurony)
- Funkcje aktywacji: 1 warstwa ReLU, 2 warstwa ReLU, wyjście Sigmoid
- Funkcja straty: BCELoss
- Algorytm optymalizacji: Adam

Testowanie poprzedniej najlepszej sieci.

- 1 warstwa ukryta
- Funkcja aktywacji (ReLU – dla wartstwy ukrytej, oraz Sigmoid dla wyjścia)
- W warstwie ukrytej 32 neurony
- Wyjście 1
- KFold = 5
- Funkcja straty: BCELoss
- Algorytm optymalizacji: Adam
- Ilość epok: 1000

Zestawienie dwóch najlepszych sieci




```
czas trwania: siec 2 warstwy    1.82
czas trwania: siec 1 warstwa    0.89
```

Podsumowanie ogólne, im więcej warstw tym teoretycznie sieć głębiej się uczy co powinno wpłynąć na lepszy rezultat, nie zawsze tak się dzieje. Czasami sieć potrafi się przetrenować i dokładność zacznie delikatnie maleć. Gdy natomiast użyjemy więcej neuronów nasza sieć bardziej zwraca uwagę na dane cechy, dzięki czemu sieć może lepiej rozwiązać większe dane, gdy ich jest dużo. Zwiększenie ilości neuronów wymaga od siebie większych obliczeń, co za tym idzie sieć się długo uczy. Większość testów wypadała w miarę okay, natomiast czasami gdy współczynnik uczenia był zbyt mały oraz mała ilość epok, sieć nie mogła się dobrze nauczyć. Finalnie mogę stwierdzić że spośród tych dwóch ostatnich sieci przedstawionych powyżej, sieć z dwiema warstwami poradziła sobie lepiej. Może trenowała trochę dłużej ale wydaje mi się że około 1 sekundy dłużej to nie jest aż tak dużo.