Wojciech Gomułka, Adrian Furman, Tomasz Gajda

Sztuczne Sieci Neuronowe

Projekt z wykorzystaniem sieci rekurencyjnych

SPIS TREŚCI

Sztuczne Sieci Neuronowe **1**

1. Informacje techniczne 3

1.1 Repozytorium projektu 3

1.2 Zawartość repozytorium 4

1.3 Technologie użyte w projekcie 5

1.4 Podział pracy 5

2. Wprowadzenie 5

2.1 Informacje wstępne 5

2.2 Przyjęte nazewnictwo i oznaczenia 6

3. Przetwarzanie danych 7

3.1 Ładowanie i inicjalizacja danych wejściowych 7

3.2 Podział danych i normalizacja 9

3.3 Podział danych na pakiety chwil czasowych 9

4. Analiza danych 9

4.1 Statystyki ogólne 10

4.2 Średnie 10

4.3 Mediany 13

4.4 Odchylenia 16

4.5 Korelacje 18

4.6 Histogramy 19

5. Podstawowa architektura sieci 21

6. Dobieranie parametrów 22

6.1 Learning rate 22

6.2 Rozmiar sekwencji czasu (time chunk size) 22

7. Testowanie innych topologii 23

8. Walidacja krzyżowa 34

9. Najlepsza architektura sieci 35

10. Przetrenowane modele 36

11. Podsumowanie 37

1. Informacje techniczne

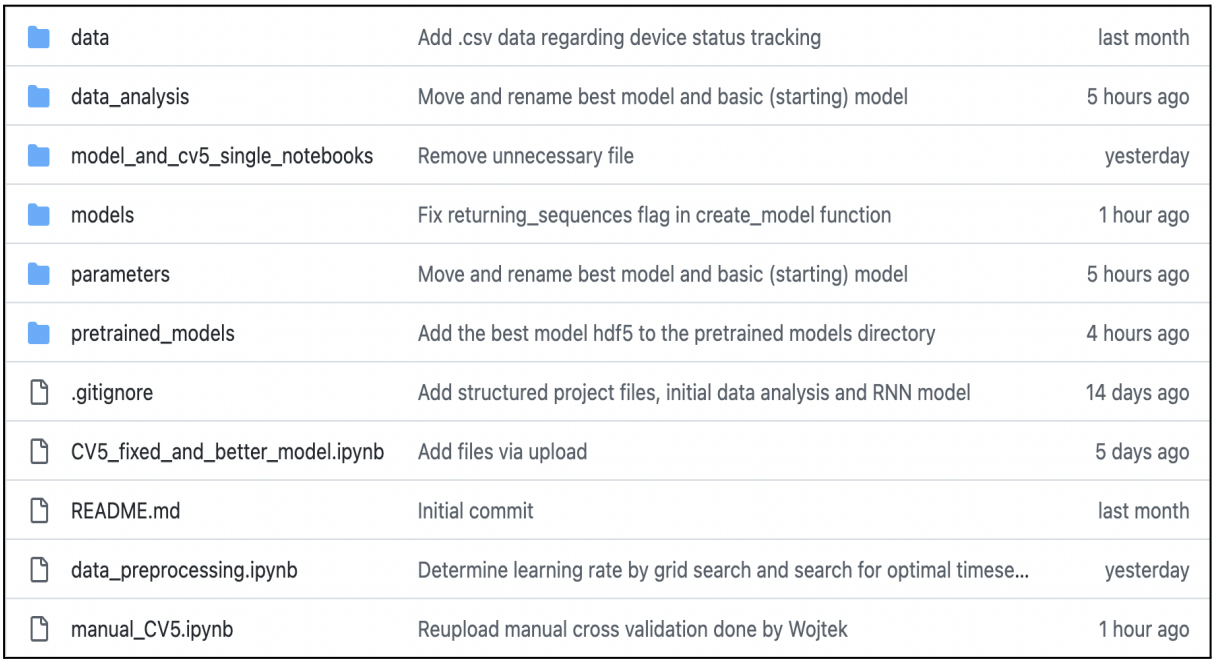
1.1 Repozytorium projektu

Pliki projektu dostępne są w repozytorium dostępnym pod URL:

https://github.com/nerooc/device-downtime-detection

Zawarte tam są wszystkie wykorzystane w tym sprawozdaniu wykresy, modele oraz rezultaty.

1.2 Zawartość repozytorium



data	Add .csv data regarding device status tracking	last month
data_analysis	Move and rename best model and basic (starting) model	5 hours ago
model_and_cv5_single_notebooks	Remove unnecessary file	yesterday
models	Fix returning_sequences flag in create_model function	1 hour ago
parameters	Move and rename best model and basic (starting) model	5 hours ago
pretrained_models	Add the best model hdf5 to the pretrained models directory	4 hours ago
.gitignore	Add structured project files, initial data analysis and RNN model	14 days ago
CV5_fixed_and_better_model.ipynb	Add files via upload	5 days ago
README.md	Initial commit	last month
data_preprocessing.ipynb	Determine learning rate by grid search and search for optimal timese...	yesterday
manual_CV5.ipynb	Reupload manual cross validation done by Wojtek	1 hour ago

Rys. 1 - Struktura repozytorium projektu

- W folderze o nazwie **data_analysis** zawarta jest analiza danych wejściowych w poszczególnych plikach znajdują się statystyki określone przez ich nazwę,-

- W folderze **data** znajdują się wszystkie pliki .csv z danymi wejściowymi,

- W folderze **parameters** znajdują się pliki przedstawiające poszukiwanie optymalnych parametrów,

- W folderze **models** mamy modele - początkowy model, od którego zaczynaliśmy, model najlepszy, który przynosił najlepsze wyniki oraz plik z wieloma testowanymi topologiami,

- Folder **pretrained_models** zawiera przetrenowane modele, które można przetestować w pliku **test_pretrained_models.ipynb**,

- W pliku **data_preprocessing** znajdują się funkcje dotyczące przetwarzania i oporządzania danych wejściowych,

1.3 Technologie użyte w projekcie

- **tensorflow** - biblioteka programistyczna napisana przez Google Brain Team. Wykorzystywana jest w uczeniu maszynowym i głębokich sieciach neuronowych, przez co idealnie sprawdziła się w przypadku naszego projektu.

- **matplotlib** - jedną z potężniejszych bibliotek Pythona, która służy do tworzenia różnego rodzaju wykresów.

- **seaborn** - efektywna biblioteka, pozwalająca na szybkie tworzenie najbardziej “atrakcyjnych” wykresów. Została, zbudowana na bazie biblioteki Matplotlib, jednocześnie wzbogacona o dodatkowe typy wykresów.

- **pandas** - jest jednym z najbardziej rozbudowanych pakietów, do analizy danych, w Python. Możemy za jego pomocą, wczytywać dane, czyścić, modyfikować, a nawet analizować.

- **scikit-learn** - ta biblioteka udostępnia wiele algorytmów z dziedziny nadzorowanego i nienadzorowanego uczenia maszynowego w postaci spójnego interfejsu programistycznego.

1.4 Podział pracy

Wojciech Gomułka - Wstępna obróbka danych, przygotowanie pierwszego modelu, walidacja krzyżowa

Adrian Furman - Dobór parametrów (liczba chwil czasowych, *learning rate*), testowanie topologii

Tomasz Gajda - Zaawansowana analiza danych wejściowych, dokumentacja

2. Wprowadzenie

2.1 Informacje wstępne

Projekt dotyczy przewidywania wystąpień awarii pewnej maszyny, na podstawie zestawu bloków danych zarówno w trakcie regularnej pracy jak i awarii. Dane zapisane są w postaci plików ***.csv*** i są ukształtowane w następujący sposób:

	1	31.1	14	21.7	25	120	5.8	65.9	36.1	15.1
01	2	31.1	14	21.8	25	120	6	55.5	35.5	15.3
02	3	30.9	14	21.8	25	60	6.3	47.4	35.1	15.9
03	4	30.6	15	21.7	25	120	6.6	42.1	34.9	16.6
04	5	30.5	15	21.7	25	660	6.6	63.7	39.6	16.8
05	6	30.7	15	21.7	25	660	6.4	70.9	42.1	16.2
06	7	31.2	15	21.7	25	600	6	72.6	42.5	15.9
07	8	31.6	15	21.8	25	120	5.8	68.5	40.6	15.7
08	9	31.7	14	21.8	25	120	6	57.4	39.7	15.8
09	10	31.5	14	21.8	25	120	6.3	49.3	39	16.4
10	11	31.2	15	21.7	25	180	6.4	43.9	38.4	17.2
11	12	31.1	15	21.7	26	660	6.5	67.3	42	17.1
12	13	31.4	15	21.7	26	600	6.3	72.7	43.1	16.8
13	14	31.9	15	21.7	26	660	5.9	73.9	43.3	16.6
14	15	32.4	15	21.7	26	240	5.7	72.2	41.8	16.3
15	16	32.5	14	21.7	26	120	5.9	60.8	40.5	16.2
16	17	32.3	14	21.7	26	60	6.1	51.6	39.8	16.6
17	18	32.1	15	21.7	26	120	6.3	45.6	39.2	17.4
18	19	31.8	15	21.7	26	600	6.5	63.4	41.6	17.8
19	20	32	15	21.7	26	600	6.3	72.8	43.3	17.2
20	21	32.4	15	21.7	26	660	5.9	74.5	43.5	16.9
21	22	32.9	15	21.7	26	480	5.7	75.2	43.6	16.6
22	23	33.1	14	21.8	26	60	5.6	65.9	41.1	16.4
23	24	33	14	21.8	26	120	5.9	55.1	40.3	16.6
24	25	32.7	14	21.8	26	120	6.2	47.9	39.6	17.3
25	26	32.3	14	21.8	26	300	6.4	48.1	39.8	18.1
26	27	32.3	15	21.8	26	660	6.4	71.5	43.1	17.6
27	28	32.6	14	21.8	26	660	6.1	74.7	43.6	17.1
28	29	33	14	21.8	26	600	5.8	75.6	43.8	16.8
29	30	33.5	14	21.8	26	180	5.6	72.3	41.9	16.5
30	31	33.4	14	21.8	26	120	5.8	60.4	40.8	16.5

Rys. 2 - Przykładowy blok danych

Pierwsza kolumna przedstawia czas w minutach (od 1 do 31), pozostałe natomiast są wartościami pomiarowymi, które otrzymaliśmy z badanej maszyny. Korzystając z 53 bloków danych w czasie poprawnej pracy i 54 w czasie awarii, otrzymać mieliśmy model przewidujący na podstawie otrzymywanych wartości czy urządzenie jest w trakcie awarii czy też pracuje poprawnie. W naszym przypadku wykorzystać do tego celu mieliśmy **Rekurencyjne Sieci Neuronowe (RNN)**. Cały proces rozpoczęliśmy od przetwarzania i analizy danych.

2.2 Przyjęte nazewnictwo i oznaczenia

W danych wejściowych na każdą minutę przypada zestaw ośmiu wartości pomiarowych, odpowiadających wartościom zwracanym przez naszą obserwowaną maszynę. W ramach następnych rozdziałów (między innymi do analizy danych) posługujemy się tymi wartościami by przedstawić pewne ich statystyki bądź zależności.

W następnych rozdziałach sprawozdania wartości te będą nazywane indeksami na jakich się znajdują (nie wliczając w to kolumny z minutami).

31.1	14	21.8	25	120	6	55.5	35.5	15.3
0	1	2	3	4	5	6	7	8

Rys. 3 - Przykładowy zestaw danych z jednej minuty i oznaczenia poszczególnych wartości

Przez większość obliczeń i analiz dane są również przechowywane w ramach obiektów **Dataframe** - jest to podstawowy kontener danych wykorzystywany w ramach biblioteki **pandas**.

3. Przetwarzanie danych

Otrzymane dane wejściowe musimy następnie wczytać i przetworzyć. Funkcje opisywane w podrozdziałach znajdują się w pliku [data_preprocessing.ipynb]

3.1 Ładowanie i inicjalizacja danych wejściowych

W zależności od miejsca użycia, dane są importowane z katalogu (gdy uruchamiane lokalnie) bądź z repozytorium (jeśli uruchamiane za pomocą Google Colab). Funkcja **load_data_from_file** odczytuje i zwraca dane z naszego pliku .csv usuwając przy tym kolumnę z indeksami jako że jej nie będzie potrzebować - będziemy korzystać z naszego własnego indeksowania.

Funkcja **load_all_data** korzystając z powyższej funkcji w pętli iteruje po wszystkich plikach i pakuje ich dane do jednej wspólnej tablicy **date_blocks**, która jest na koniec zwracana. Przenosząc dane z plików do tablicy, w miejsce danego bloku danych oprócz tablicy z pomiarami dorzucamy **jeden bit** wskazujący czy jest to odczyt z działającego czy też niedziałającego urządzenia.

Ostatecznie tablica naszych danych prezentuje się w następujący sposób:

```
[
      [ [ TABLICA Z POMIARAMI Z 1 MINUTY ],
        [ TABLICA Z POMIARAMI Z 2 MINUTY ],
        .
        .   31 pomiarów
        .
        [ TABLICA Z POMIARAMI Z 31 MINUTY ] ], 0],

      [ [ TABLICA Z POMIARAMI Z 1 MINUTY ],
        [ TABLICA Z POMIARAMI Z 2 MINUTY ],
        .
        .   31 pomiarów
        .
        [ TABLICA Z POMIARAMI Z 31 MINUTY ] ], 1],
      .
```

```
. 54 bloki pomiarowe
.
[ [ TABLICA Z POMIARAMI Z 1 MINUTY ],
  [ TABLICA Z POMIARAMI Z 2 MINUTY ],
  .
  .   31 pomiarów
  .
  [ TABLICA Z POMIARAMI Z 31 MINUTY ] ], 0],
]
```

3.2 Podział danych i normalizacja

Funkcja o nazwie **split_working_faulty_blocks** korzystając z tablicy zwróconej w poprzednim punkcie, przetwarza ją i zwraca dwa **dataframe'y** - jeden z blokami działającymi i jeden z blokami awaryjnymi. Te posłużą nam w dalszej analizie statystycznej jak i przy tworzeniu idealnego modelu.

Funkcja ta ma również opcjonalny parametr decydujący czy dane użyte w dataframe'ach mają być **znormalizowane**. Parametr ten domyślnie jest oznaczony jako **False.**

Do normalizacji wykorzystana jest funkcja **normalize_data**, która przyjmuje dane w postaci dataframe'u i zwraca je w tej samej postaci lecz znormalizowane.

3.3 Podział danych na pakiety chwil czasowych

By przygotować dane do trenowania, musimy je podzielić na **pakiety chwil czasowych**. W następnych rozdziałach ustalimy ile tych szeregów stanowi optymalną liczbę do trenowania, jednak na etapie przetworzenia przygotowaliśmy funkcję **time_chunks_split**, która zwraca nam przetworzone zestawy danych treningowych i testowych.

4. Analiza danych

W ramach projektu mieliśmy dokonać analizy danych wejściowych, prezentując najważniejsze statystyki i wnioski. Większość oczekiwanych statystyk została obliczona na dwa sposoby - **bezpośredni** za pomocą funkcji przeznaczonych do poszczególnych statystyk i ogólniejszy, korzystając z przygotowanej do tego funkcji **describe** z biblioteki **Pandas.** Wszystkie elementy dotyczące analizy danych dostępne są w plikach wewnątrz folderu **data_analysis.**

4.1 Statystyki ogólne

Wynik większości statystyk otrzymaliśmy za pomocą wspomnianej wcześniej metody.

	0	1	2	3	4	5	6	7	8
count	1643.000000	1643.000000	1643.000000	1643.000000	1643.000000	1643.000000	1643.000000	1643.000000	1643.000000
mean	30.412538	16.251978	21.732136	27.356056	308.764455	6.004504	57.479002	36.323798	16.064455
std	0.778284	2.482789	0.482308	4.134368	231.419114	0.562396	11.119428	1.721049	1.205308
min	27.200000	13.000000	20.900000	22.000000	60.000000	5.200000	30.800000	31.400000	14.600000
25%	29.900000	14.000000	21.400000	23.000000	120.000000	5.600000	47.300000	34.900000	15.300000
50%	30.400000	16.000000	21.500000	27.000000	180.000000	5.900000	59.700000	35.900000	15.900000
75%	30.900000	18.000000	22.200000	29.000000	600.000000	6.200000	68.400000	37.900000	16.600000
max	32.600000	24.000000	22.900000	38.000000	660.000000	9.000000	72.800000	40.200000	22.600000

Rys. 4 - Tabela otrzymana w wyniku metody **describe** dla bloków działających

	0	1	2	3	4	5	6	7	8
count	1674.000000	1674.000000	1674.000000	1674.000000	1674.000000	1674.000000	1674.000000	1674.000000	1674.000000
mean	31.103405	18.659498	21.439367	32.762246	322.831541	5.839606	59.551792	38.608602	16.287157
std	1.214770	2.801451	0.742500	4.837056	237.406512	0.420602	11.291858	2.304267	0.908331
min	27.200000	13.000000	19.100000	23.000000	60.000000	4.800000	30.600000	31.300000	13.800000
25%	30.400000	17.000000	21.100000	29.000000	120.000000	5.600000	48.800000	36.900000	15.700000
50%	31.200000	19.000000	21.400000	33.000000	240.000000	5.800000	61.400000	38.600000	16.200000
75%	31.900000	20.000000	22.000000	36.000000	600.000000	6.100000	70.400000	40.400000	16.800000
max	34.800000	25.000000	22.900000	42.000000	660.000000	8.600000	77.400000	43.900000	22.600000

Rys. 5 - Tabela otrzymana w wyniku metody **describe** dla bloków awaryjnych

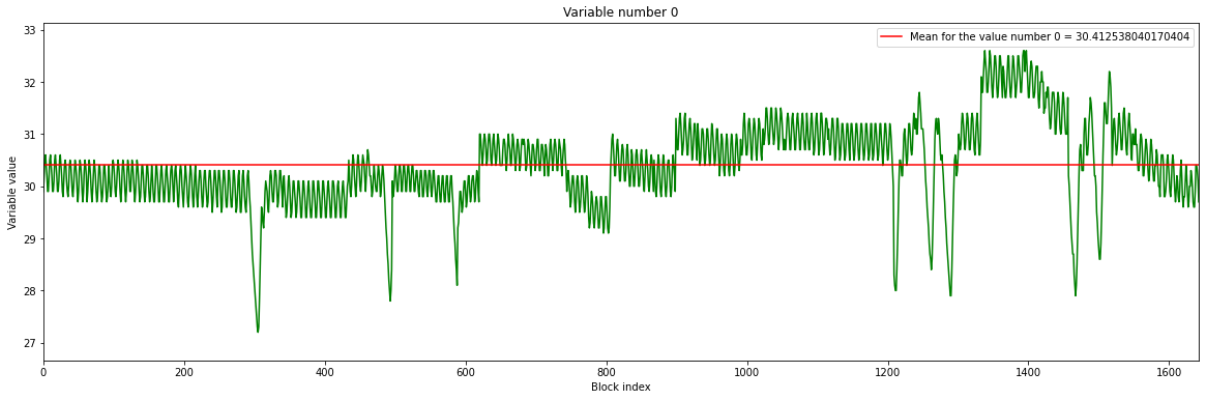
4.2 Średnie

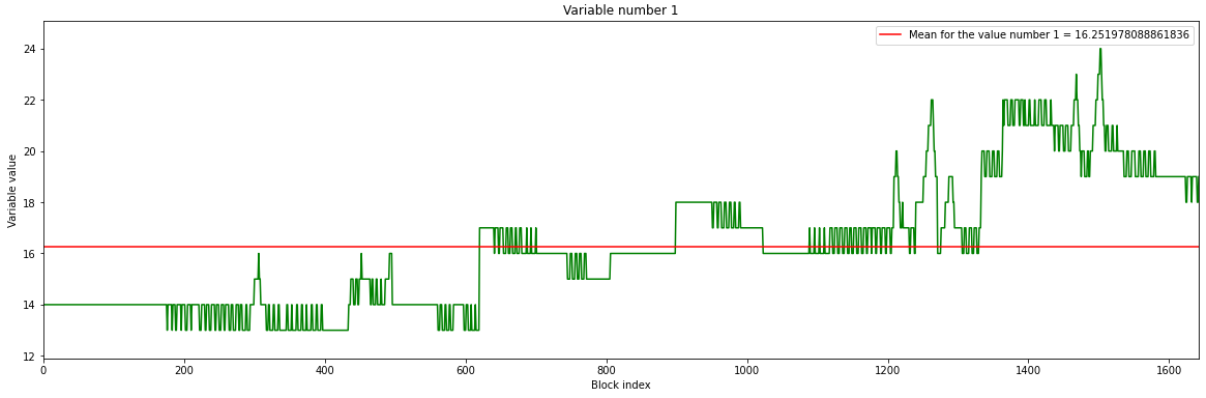
W ramach obliczania średnich w sposób bezpośredni, wyniki pokrywają się z tymi otrzymanymi z metody **describe**.

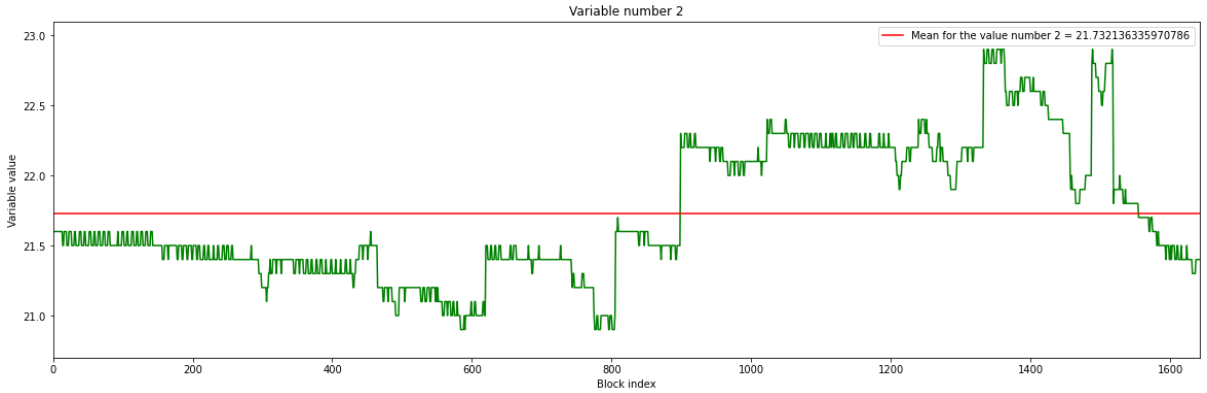
```
Średnie dla wszystkich dostępnych bloków działających:

[ 30.41253804  16.25197809  21.73213634  27.356056   308.76445526
   6.00450396  57.47900183  36.32379793  16.06445526]
```

```
Średnie dla wszystkich dostępnych bloków awaryjnych:

[ 31.10340502  18.65949821  21.43936679  32.76224612 322.83154122
   5.83960573  59.55179211  38.60860215  16.28715651]
```

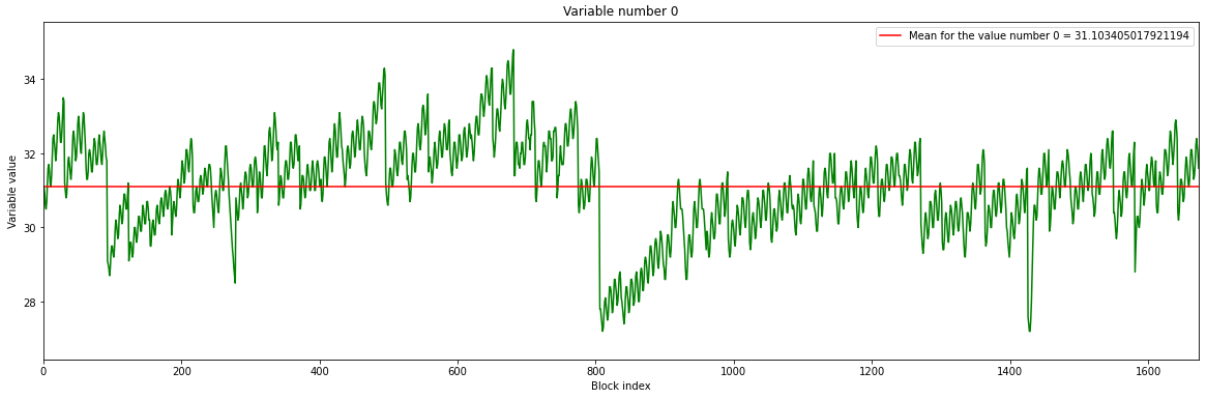
Dla każdej wartości otrzymanej w ramach danych z jednej minuty policzyliśmy średnią ze wszystkich istniejących pomiarów i kroków czasowych. Przedstawiliśmy je również na wykresach. Rozpoczynając od bloków działających:

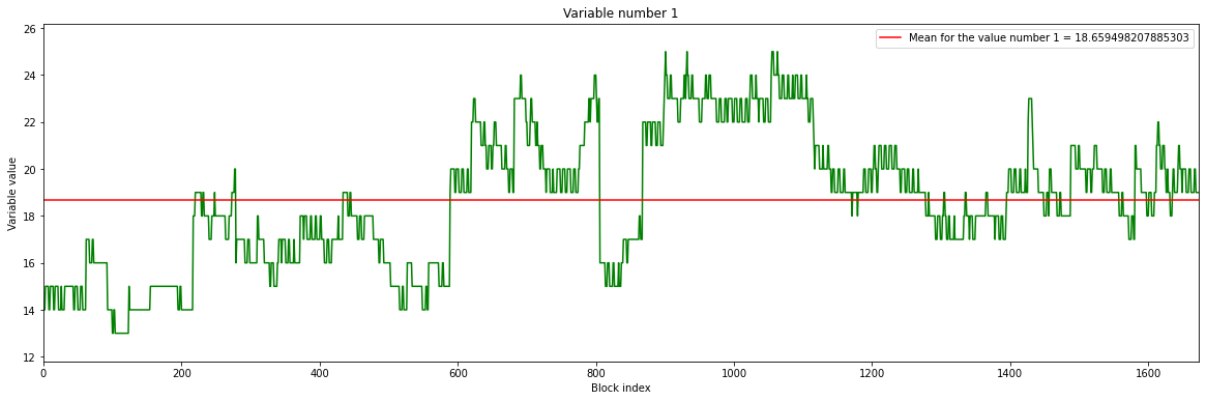


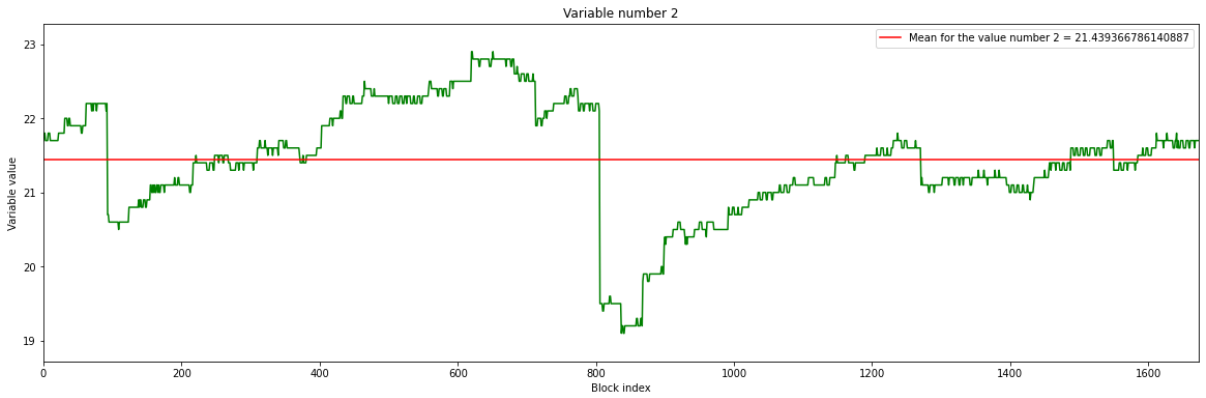




Rys. 6 - Wykresy wartości trzech zmiennych w czasie oraz ich medianą dla bloku działającego







Rys. 7 - Wykresy wartości trzech zmiennych w czasie oraz ich średnią dla bloku awaryjnego

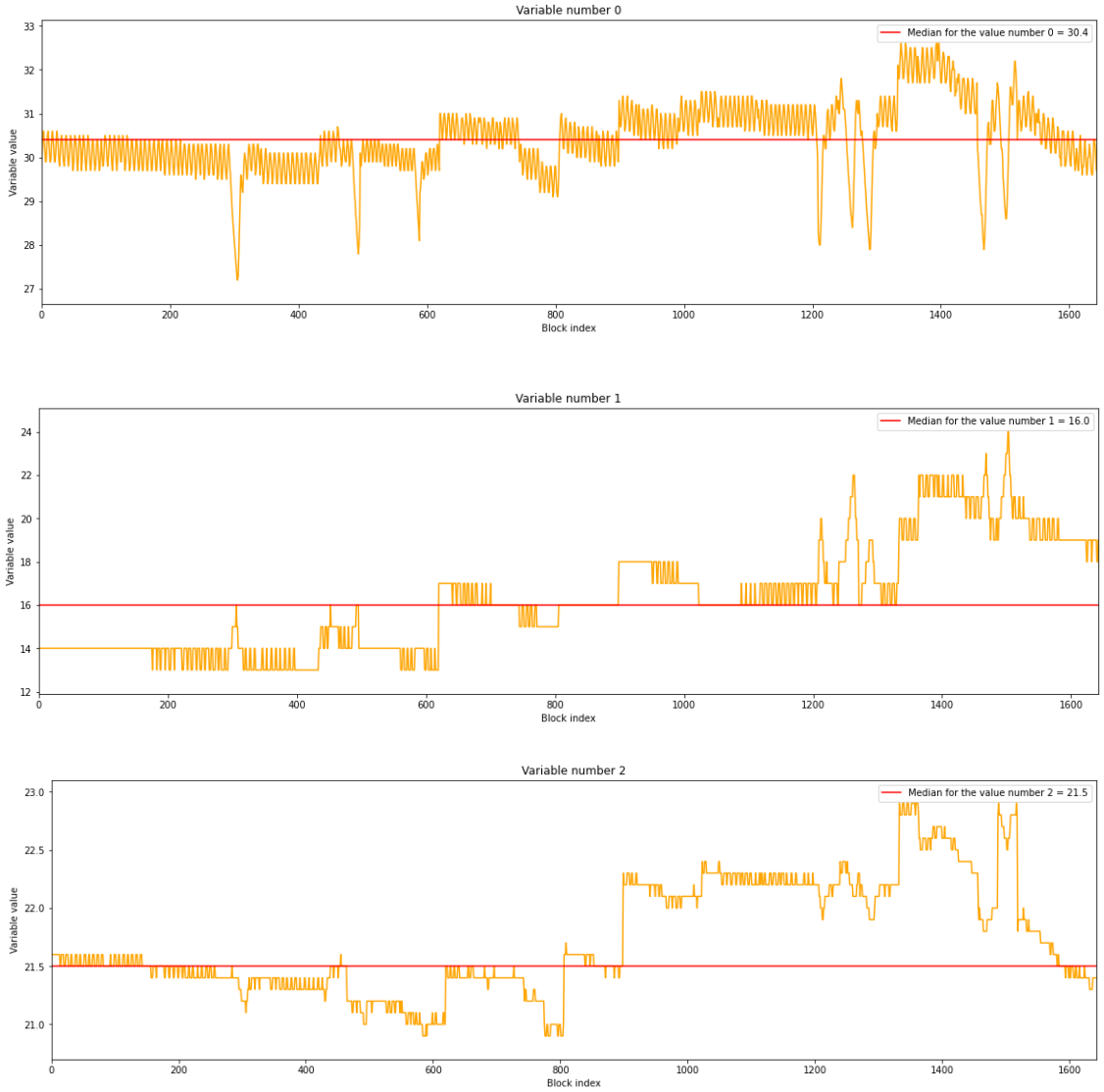
Wykresy ze średnimi pozostałych zmiennych są dostępne tutaj.

4.3 Mediany

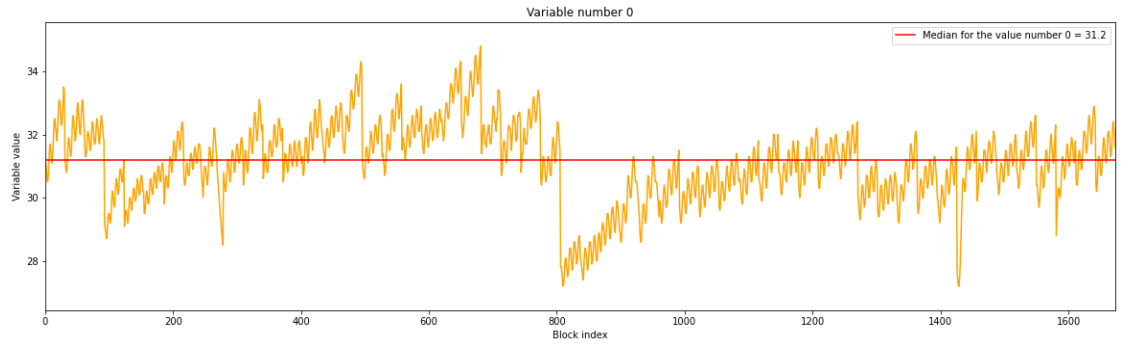
Podobnie jeśli chodzi o obliczanie median w sposób bezpośredni, wyniki pokrywają się z tymi otrzymanymi z metody **describe**.

```
Mediany dla wszystkich dostępnych bloków działających:

[ 30.4   16.    21.5   27.   180.     5.9   59.7   35.9   15.9]

Mediany dla wszystkich dostępnych bloków awaryjnych:

[ 31.2   19.    21.4   33.   240.     5.8   61.4   38.6   16.2]
```

Dla każdej wartości otrzymanej w ramach danych z jednej minuty policzyliśmy też medianę ze wszystkich istniejących pomiarów i kroków czasowych. Przedstawiliśmy je również na wykresach. Rozpoczynając od bloków **działających**:



Rys. X - Wykresy wartości trzech zmiennych w czasie oraz ich medianą dla bloku działającego

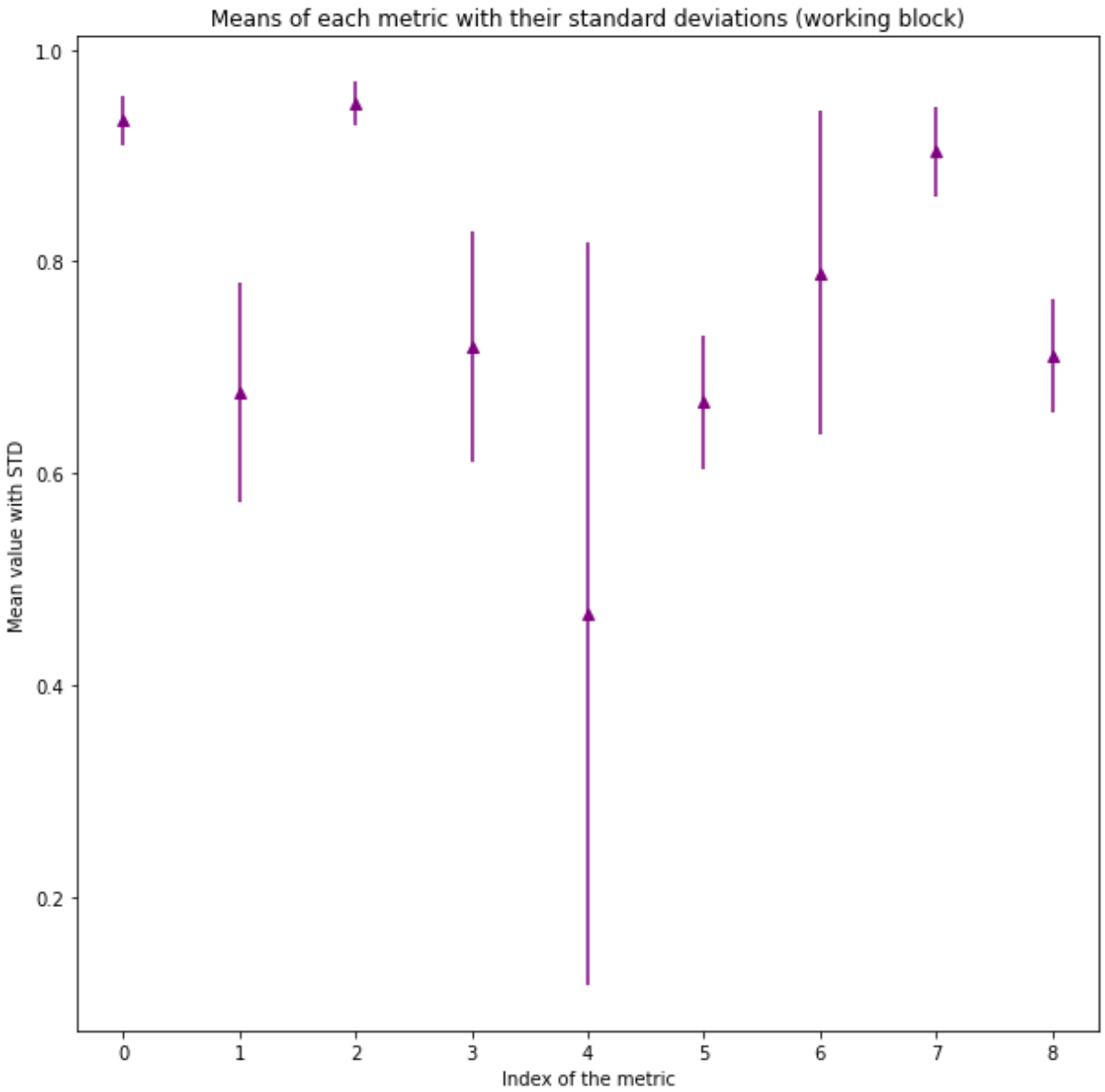


Rys. 8 - Wykresy wartości trzech zmiennych w czasie oraz ich medianą dla bloku awaryjnego

Wykresy z medianami pozostałych zmiennych są dostępne tutaj.

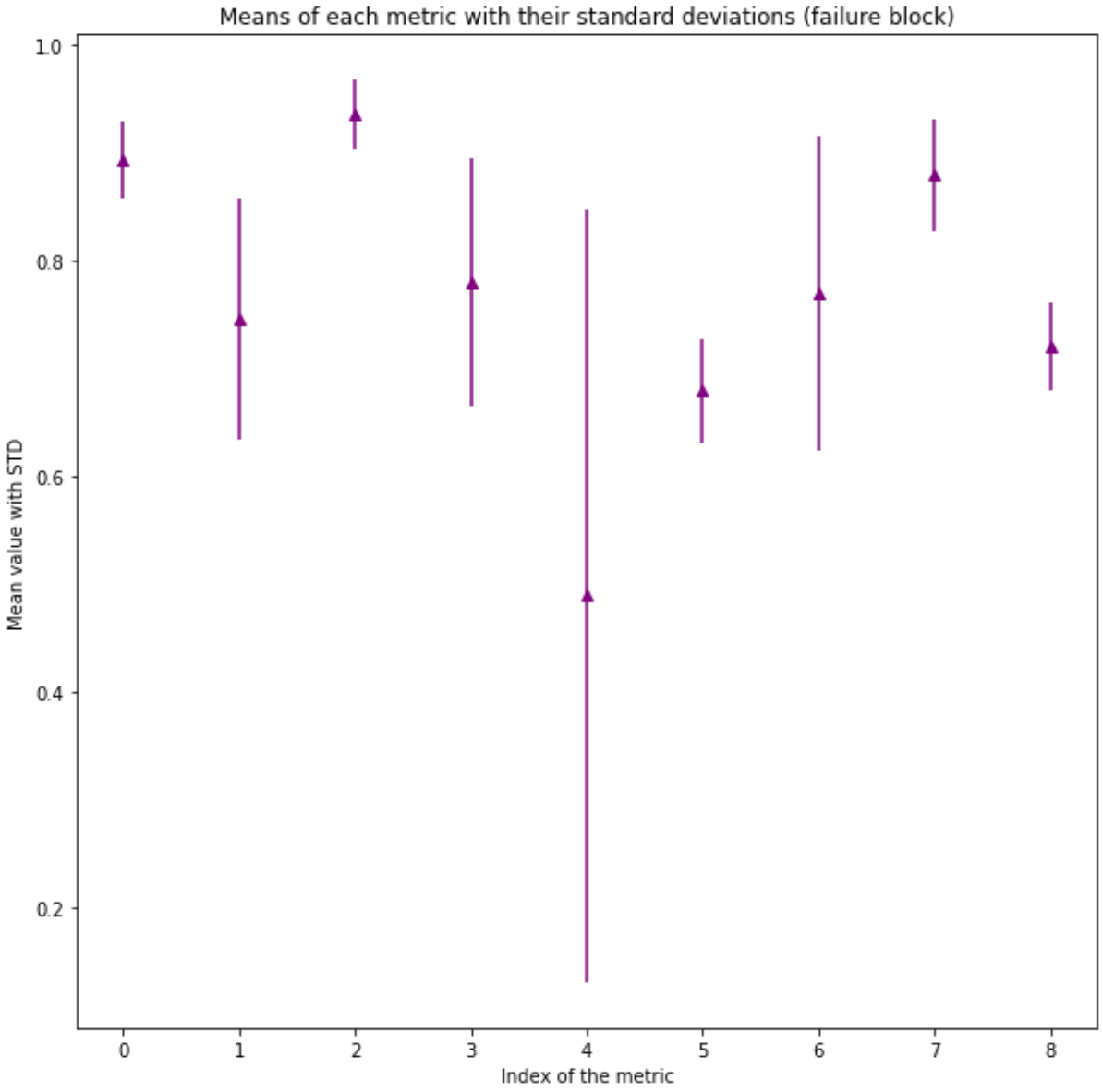
4.4 Odchylenia

W ramach statystyk obliczyliśmy również odchylenia poszczególnych wartości obu bloków i przedstawiliśmy je na wykresach. Możemy na nich zobaczyć zarówno średnią wartość danej zmiennej jak i otaczające ją możliwe odchylenie. Rozpoczynając od bloków **działających**:



Rys. 9 - Wykres średnich wartości i ich odchylenia standardowego w blokach standardowych

Podobny wykres stworzony został również dla danych **awaryjnych**:



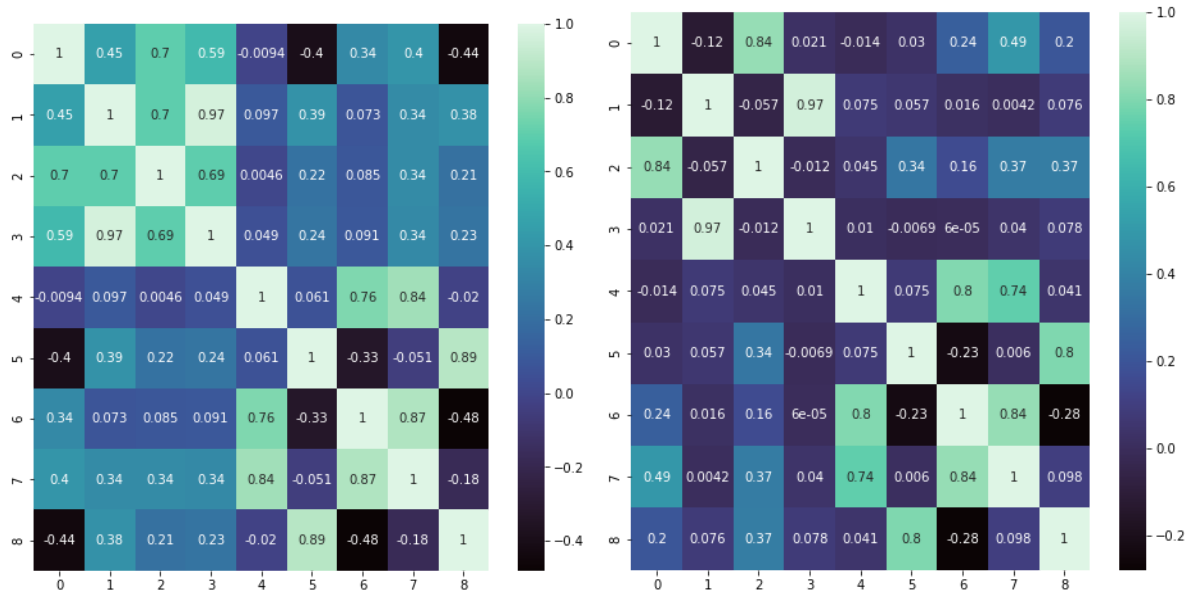
Rys. 10 - Wykres średnich wartości i ich odchylenia standardowego w blokach awaryjnych

Kod dotyczący generowania wykresów odchylenia standardowego jest dostępny w pliku o ścieżce **/data_analysis/deviations.ipynb**.

4.5 Korelacje

Następną ważną statystyką którą mogliśmy wyciągnąć z naszych danych wejściowych to macierze korelacji. Otrzymaliśmy je za pomocą funkcji **corr**, z której możemy skorzystać dzięki obiektowi Dataframe. Możemy zauważyć spore różnice w korelacjach jeśli chodzi o dane dla bloków działających i awaryjnych. Możemy również określić które zmienne są na tyle mało skorelowane, że moglibyśmy je pominąć przygotowując dane do uczenia.

Wartości te przedstawiliśmy za pomocą wykresów:



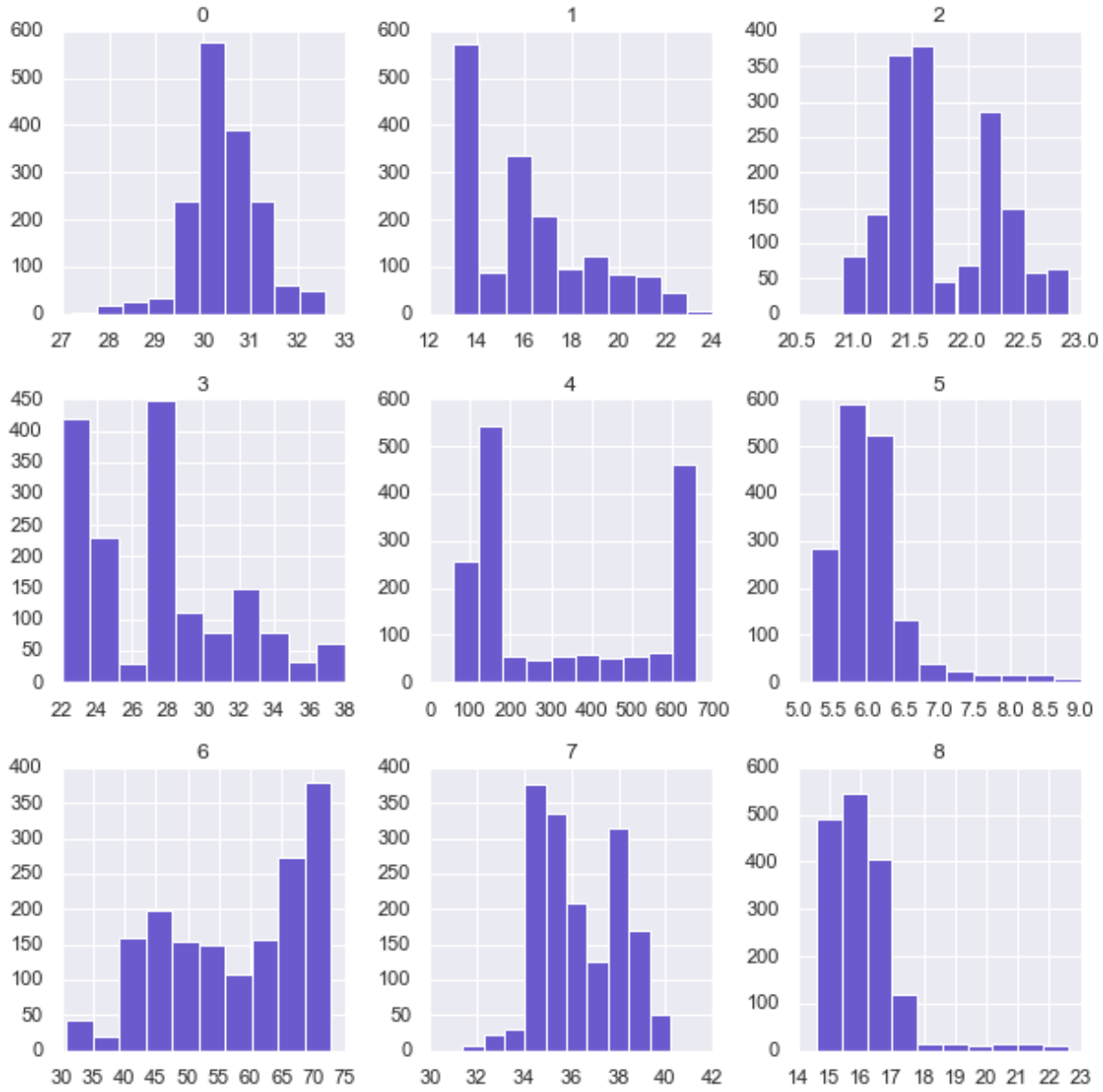
Rys. 11 - Wykresy korelacji pomiędzy zmiennymi dla bloków (od lewej) działających i awaryjnych

Kod dotyczący generowania wykresów macierzy korelacji jest dostępny w pliku o ścieżce **/data_analysis/correlations.ipynb**.

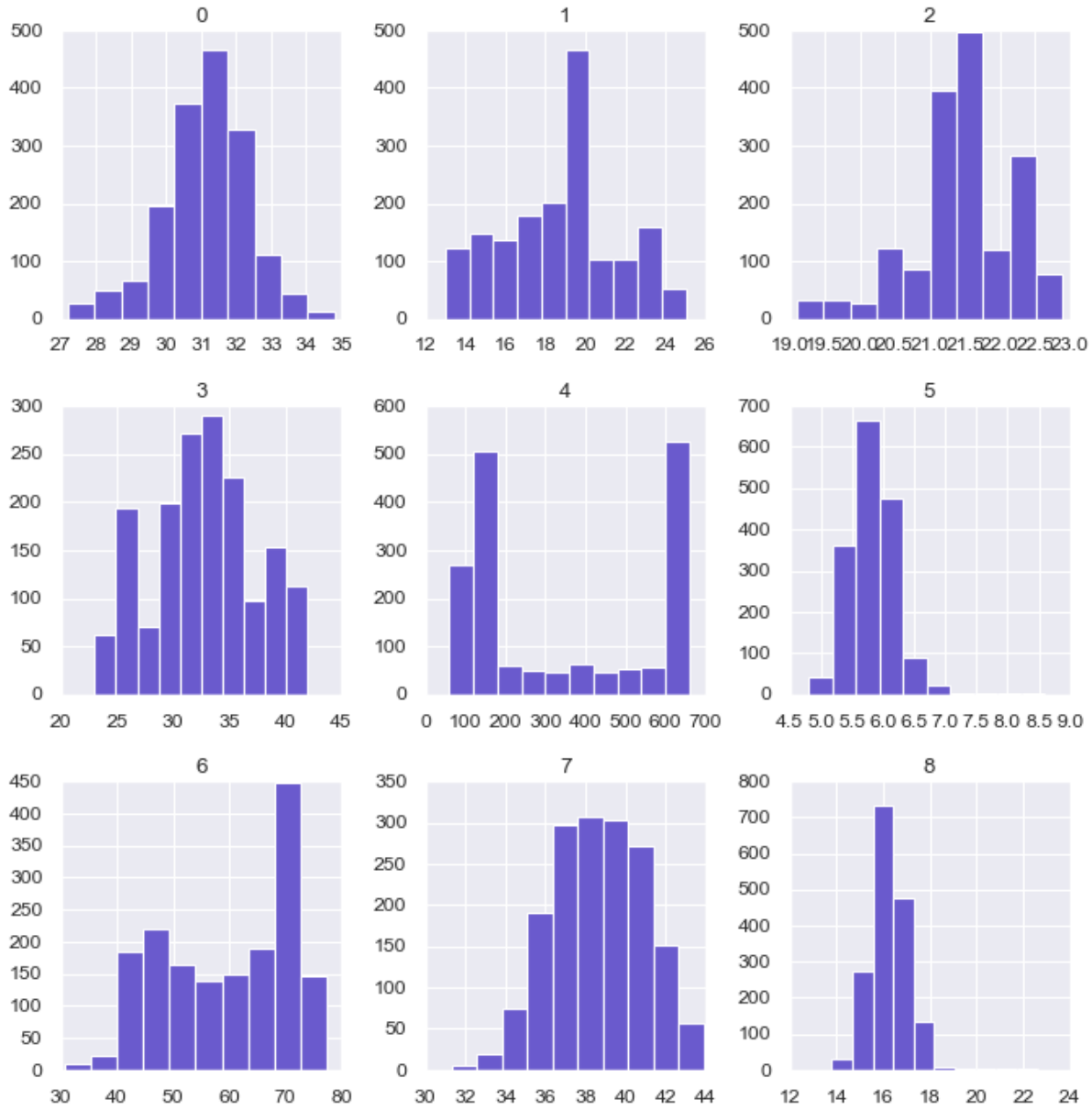
4.6 Histogramy

Na koniec przedstawiliśmy jeszcze nasze dane za pomocą histogramów. Zostały one stworzone za pomocą wbudowanej metody **hist**.

Na bazie wyrysowanych histogramów jesteśmy w stanie stwierdzić że niektóre z naszych zapisywanych zmiennych będą ważniejsze jeśli chodzi o **wykrywanie awarii** niż pozostałe.



Rys. 12 - Histogramy przedstawiające dane i ich zależności dla bloków działających



Rys. 13 - Histogramy przedstawiające dane i ich zależności dla bloków awaryjnych

Kod dotyczący generowania powyższych histogramów jest dostępny w pliku o ścieżce **/data_analysis/visualizations.ipynb**.

5. Podstawowa architektura sieci

Rozpoczęliśmy od stworzenia naszej podstawowej sieci, którą potem będziemy mogli modyfikować w ramach testowania i poszukiwania lepszych topologii. Kod tej architektury dostępny jest w pliku **/models/basic_model.ipynb**.

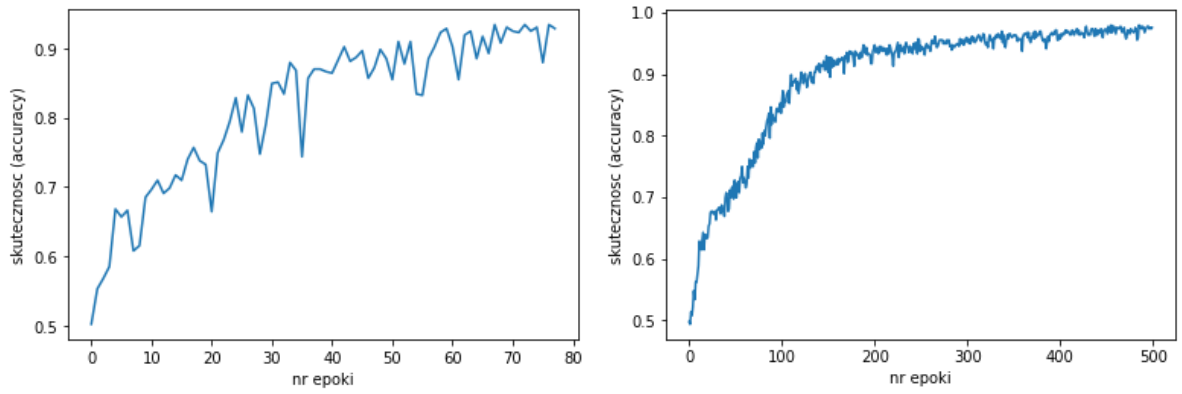
```
def get_model(lr=0.0001):
  # typowy sekwencyjny model
  model = Sequential()
  # dodajemy warstwe rekurencyjna, input_shape w naszym przypadku bedzie NUMBER_OF_TIMESTAMPS (minut) x 9
  # (nie liczac usunietego uprzednio indeksu wskazujacego z ktora minuta mamy do czynienia)
  model.add(SimpleRNN(75, input_shape=(NUMBER_OF_TIMESTAMPS, 9), activation='relu', return_sequences=True))
  model.add(Dropout(0.2))
  model.add(SimpleRNN(45, activation='relu', return_sequences=False))
  # warstwa gesta na wyjscie - najbardziej typowe rozwiazanie

  # niezmiernie istotne -> binarna entropia krzyzowa wymaga jednego neuronu na wyjsciu
  model.add(Dense(units=1, activation=tf.keras.activations.sigmoid))
  opt = tf.optimizers.Adam(learning_rate=lr)
  model.compile(loss='binary_crossentropy', optimizer=opt, metrics=['accuracy'])

  return model
```

Blok 1 - Fragment kodu reprezentujący generację naszego początkowego modelu

W funkcji o nazwie **get_model** zawarte jest zdefiniowanie naszego podstawowego modelu. Jest on typowym modelem sekwencyjnym, który rozpoczyna się od warstwy rekurencyjnej z **75 neuronami**, gdzie **input_shape** w naszym przypadku będzie to np. **8** (minut - można wziąć też więcej lub mniej - w następnym rozdziale dowiemy się jak odnaleźć najlepszą ilość minut) **x 9** parametrów (po usunięciu minut z bloków) oraz funkcji aktywacyjnej **ReLU**. Następnie mamy dropout o wartości **0.2** oraz kolejną warstwę rekurencyjną, o tej samej funkcji aktywacyjnej, lecz z wyłączonym zwracaniem sekwencji.

Niezmiernie istotne jest dodanie na końcu **warstwy gęstej** z jednym neuronem, jako że binarna entropia krzyżowa wymaga jednego neuronu na wyjściu. Warstwa gęsta posiada **sigmoid** jako funkcję aktywacji**.**

Pierwotnie korzystaliśmy z **tangensa hiperbolicznego** lecz niestabilne wyniki jak i otrzymane porady przekonały nas do zmiany. Aktualizacja znacznie ustabilniła wyniki co upewniło nas że był to dobry wybór.



Rys. 14 - Wykresy przedstawiające skuteczność podstawowegomodelu - (od lewej) z mniejszą ilością epok oraz **tangensem hiperbolicznym** jako funkcją aktywacji warstwy gęstej oraz z większą ilością epok i **sigmoidem** jako funkcją aktywacji warstwy gęstej

6. Dobieranie parametrów

By jak najlepiej wytrenować nasz model, musimy dobrze dobrać **parametry uczenia**.

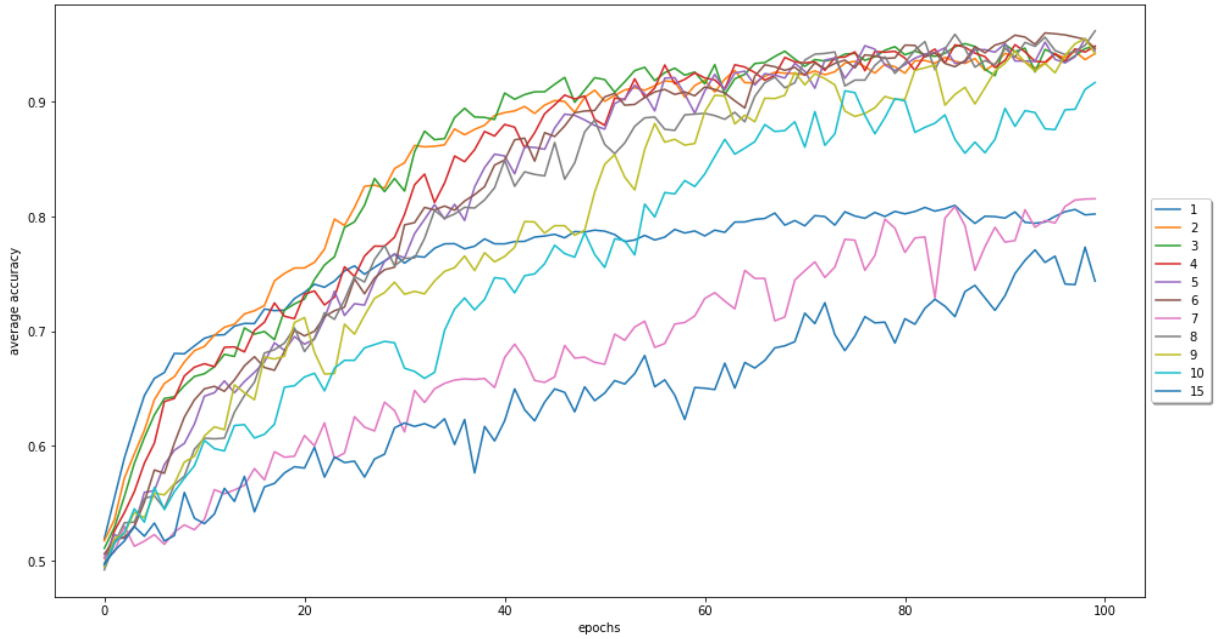
6.1 Learning rate

Skorzystaliśmy z funkcji **GridSearchCV**, która oferuje wyczerpujące wyszukiwanie określonych wartości parametrów dla estymatora. Podając tablicę testowanych wartości learning rate ([0.0005, 0.001, 0.003, 0.005, 0.007, 0.01]) jako wynik otrzymaliśmy informację że to **0.0005** jest najlepiej pasującym parametrem. Implementacja dostępna w pliku [parameters/learning_rate.ipynb]

6.2 Rozmiar sekwencji czasu (time chunk size)

Kolejną ważną do ustalenia zmienną jest wielkość sekwencji, których będziemy używać do trenowania naszego modelu. Choć nie jesteśmy jednoznacznie w stanie określić który rozmiar jest najlepszy (ponieważ wykres zawiera pewne

niestabilności i nie za każdym razem prezentuje takie same wyniki) to na podstawie kilku powtórzeń możemy wskazać że **8** to właściwy rozmiar który pozwoli nam na skuteczne trenowanie. Implementacja dostępna w pliku **parameters/time_chunks.ipynb**



Rys. 15 - Wykres średniej dokładności na epokę dla poszczególnych rozmiarów sekwencji

7. Testowanie innych topologii

W ramach projektu mieliśmy również przetestować różne topologie w celu zaobserwowania zmian jak i odnalezienia tej najdokładniejszej. Wszystkie testowane topologie znajdują się w pliku **/models/topologies.ipynb.**

Według ustaleń z poprzedniego rozdziału, jako **learning_rate** przyjmujemy wartość **0.005**, a jako **chunk_size** wykorzystujemy **8** wartości.

Za pomocą stworzonej metody **test_model** mogliśmy w dość prosty sposób testować modele o zmienionych parametrach.

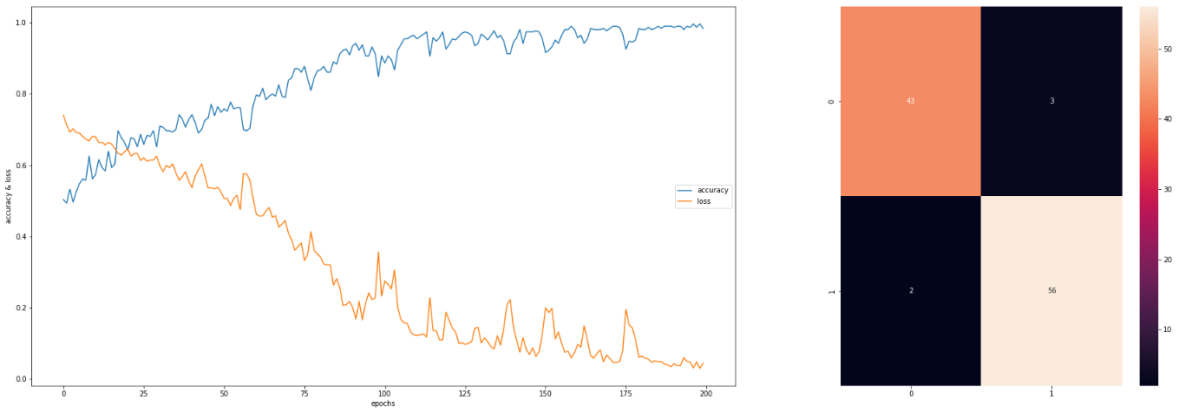
7.1 Topologia pierwsza

Train accuracy: 0.9838709831237793

Test accuracy: 95.192307692307

```
model = Sequential([
  SimpleRNN(64, input_shape=(chunk_size, 9), activation='relu', return_sequences=True),
  Dropout(0.4),
  SimpleRNN(8, activation='relu', return_sequences=False),
  Dense(units=1, activation=tf.keras.activations.sigmoid),
])

opt = tf.optimizers.Adam(learning_rate=learning_rate)

test_model(model, opt)
```

Blok 2 - Fragment kodu reprezentujący generację pierwszej topologii



Rys. 16 - Wykresy przedstawiające dokładność i błąd na epokę oraz macierz pomyłek dla pierwszej topologii

7.2 Topologia druga

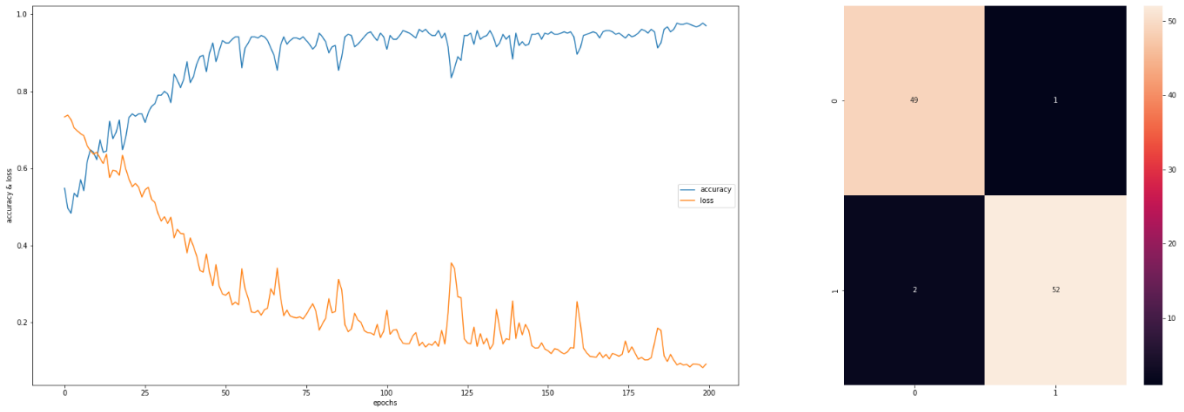
Zmiana funkcji aktywacji z relu na tanh przełożyła się na nieznaczną poprawę stabilności modelu oraz jego skuteczności - zarówno tej mierzonej na danych testowych jak i danych uczących.

Train accuracy: 0.9709677696228027

Test accuracy: 97.11538461538461

```
model = Sequential([
  SimpleRNN(64, input_shape=(chunk_size, 9), activation='tanh', return_sequences=True),
  Dropout(0.4),
  SimpleRNN(8, activation='tanh', return_sequences=False),
  Dense(units=1, activation=tf.keras.activations.sigmoid),
])

opt = tf.optimizers.Adam(learning_rate=learning_rate)

test_model(model, opt)
```

Blok 3 - Fragment kodu reprezentujący generację drugiej topologii



Rys. 17 - Wykresy przedstawiające dokładność i błąd na epokę oraz macierz pomyłek dla drugiej topologii

7.3 Topologia trzecia

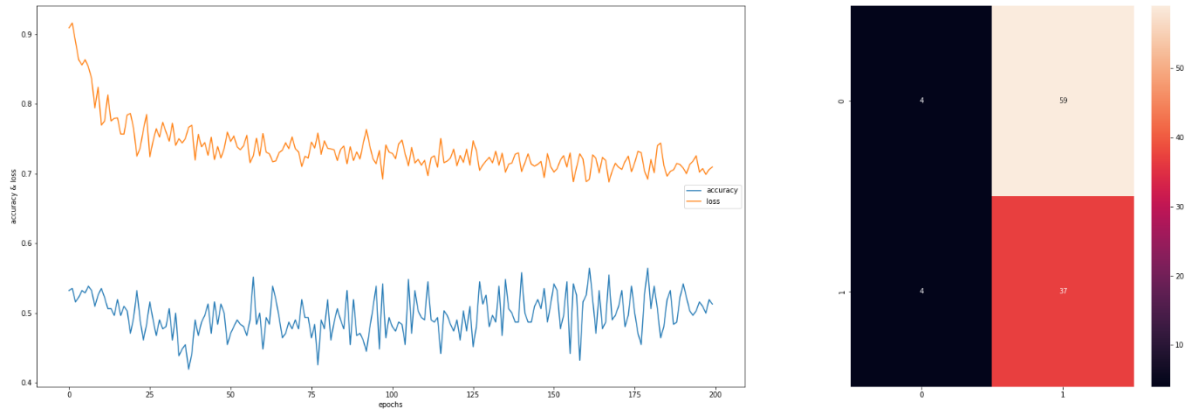
Zmiana optymalizatora na SGD spowodowała całkowity rozpad zbieżności modelu.

Train accuracy: 0.5129032135009766

Test accuracy: 39.42307692307692

```
model = Sequential([
  SimpleRNN(64, input_shape=(chunk_size, 9), activation='tanh', return_sequences=True),
  Dropout(0.4),
  SimpleRNN(8, activation='tanh', return_sequences=False),
  Dense(units=1, activation=tf.keras.activations.sigmoid),
])

opt = tf.optimizers.SGD(learning_rate=learning_rate)

test_model(model, opt)
```

Blok 4 - Fragment kodu reprezentujący generację trzeciej topologii



Rys. 18 - Wykresy przedstawiające dokładność i błąd na epokę oraz macierz pomyłek dla trzeciej topologii

7.4 Topologia czwarta

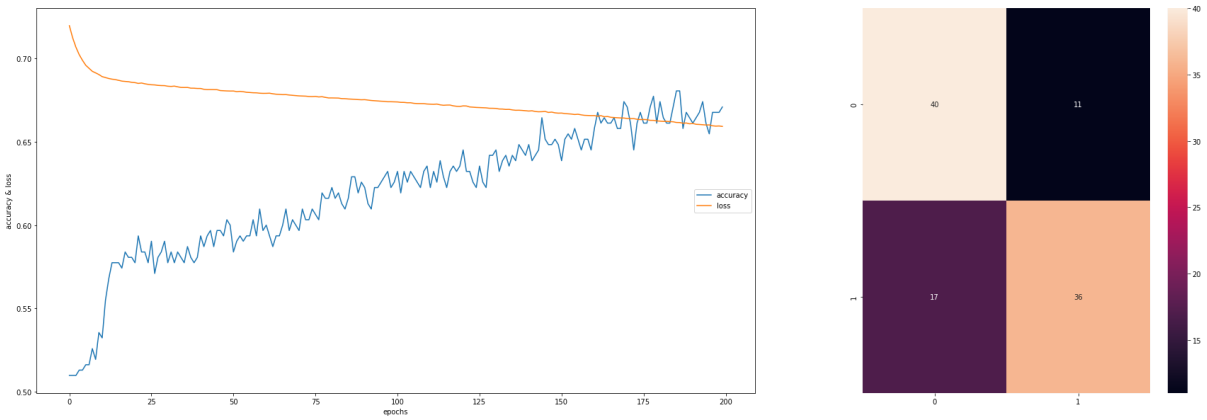
Pominięcie warstwy Dropout znacznie pogorszyło skuteczność modelu, sprowadzając ją do wysoce niesatysfakcjonujących wartości.

Train accuracy: 0.6548386812210083

Test accuracy: 64.42307692307693

```
model = Sequential([
  SimpleRNN(64, input_shape=(chunk_size, 9), activation='tanh', return_sequences=True),
  SimpleRNN(8, activation='tanh', return_sequences=False),
  Dense(units=1, activation=tf.keras.activations.sigmoid),
])

opt = tf.optimizers.SGD(learning_rate=learning_rate)

test_model(model, opt)
```

Blok 5 - Fragment kodu reprezentujący generację czwartej topologii



Rys. 19 - Wykresy przedstawiające dokładność i błąd na epokę oraz macierz pomyłek dla czwartej topologii

7.5 Topologia piąta

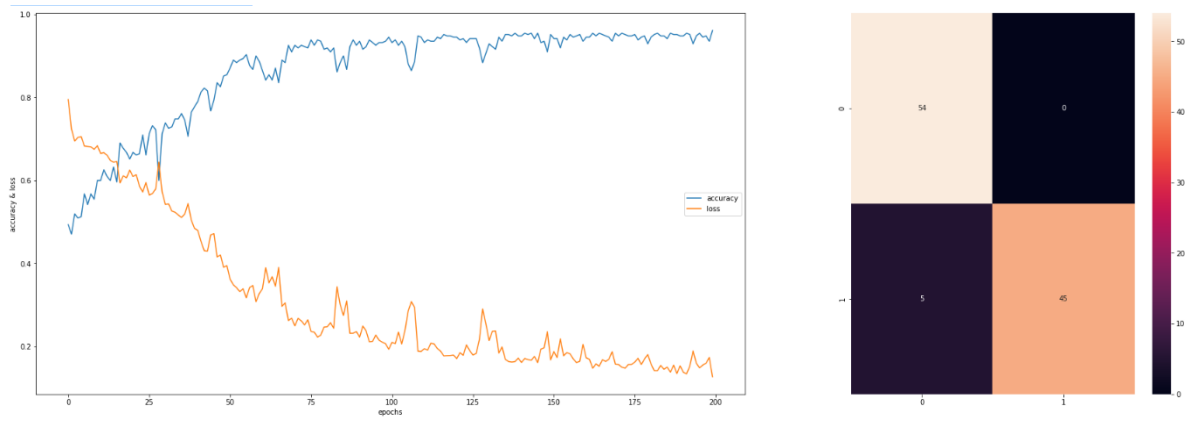
Zmiana odsetka odrzucanych wartości (parametr warstwy Dropout) na wartość niższą - 0.2 daje porównywalnie dobre rezultaty z wartością 0.4.

Train accuracy: 0.9612902998924255

Test accuracy: 95.1923076923077

```
model = Sequential([
  SimpleRNN(64, input_shape=(chunk_size, 9), activation='tanh', return_sequences=True),
  Dropout(0.2),
  SimpleRNN(8, activation='tanh', return_sequences=False),
  Dense(units=1, activation=tf.keras.activations.sigmoid),
])

opt = tf.optimizers.Adam(learning_rate=learning_rate)

test_model(model, opt)
```

Blok 6 - Fragment kodu reprezentujący generację piątej topologii



Rys. 20 - Wykresy przedstawiające dokładność i błąd na epokę oraz macierz pomyłek dla piątej topologii

7.6 Topologia szósta

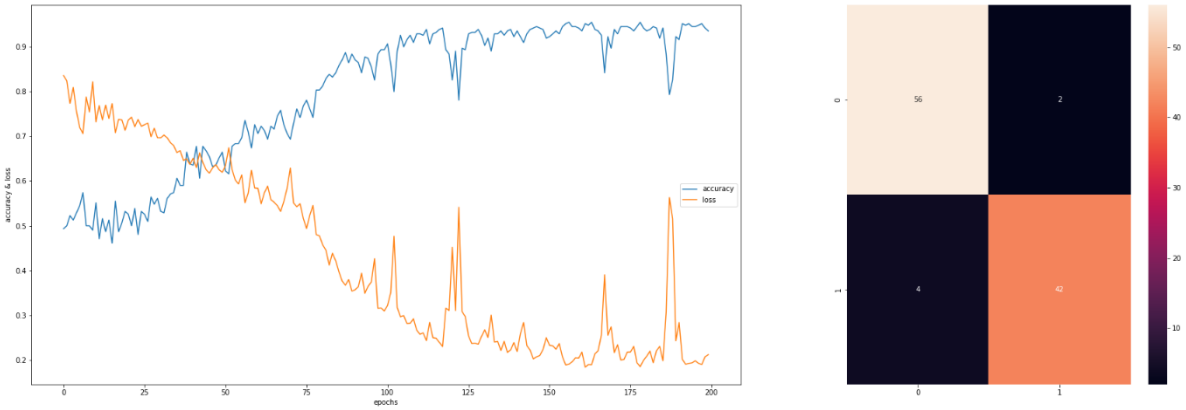
Zbyt wysoka wartość warstwy Dropout obniża wyjściową skuteczność modelu oraz powoduje częstsze i większe skoki parametrów modelu w późniejszych etapach uczenia.

Train accuracy: 0.9354838728904724

Test accuracy: 94.23076923076923

```
model = Sequential([
  SimpleRNN(64, input_shape=(chunk_size, 9), activation='tanh', return_sequences=True),
  Dropout(0.8),
  SimpleRNN(8, activation='tanh', return_sequences=False),
  Dense(units=1, activation=tf.keras.activations.sigmoid),
])

opt = tf.optimizers.Adam(learning_rate=learning_rate)

test_model(model, opt)
```

Blok 7 - Fragment kodu reprezentujący generację szóstej topologii



Rys. 20 - Wykresy przedstawiające dokładność i błąd na epokę oraz macierz pomyłek dla szóstej topologii

7.7 Topologia siódma

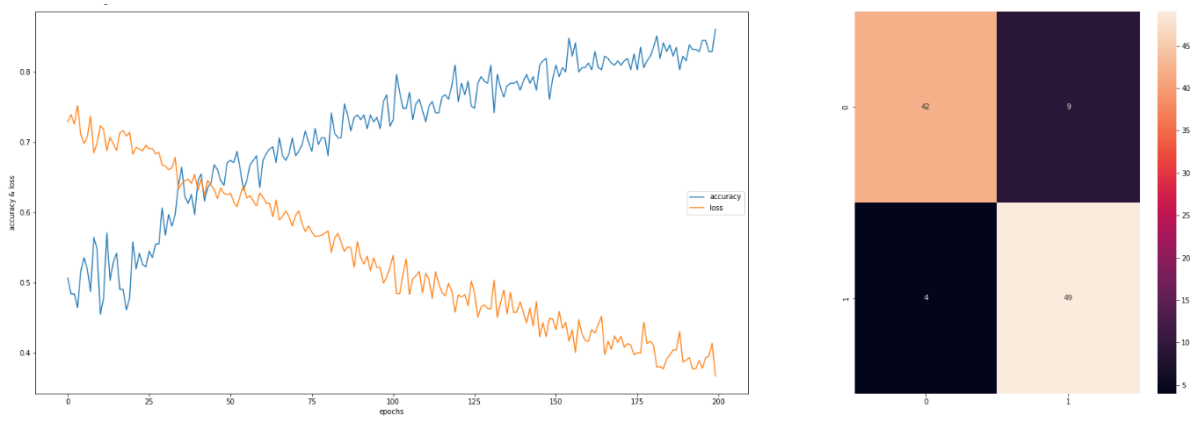
Dołożenie dodatkowej warstwy ukrytej oraz zmiana architektury na stopniowo narastające liczby neuronów nie przyniosły pozytywnych zmian do wyniku.

Train accuracy: 0.8612903356552124

Test accuracy: 87.5

```
model = Sequential([
  SimpleRNN(8, input_shape=(chunk_size, 9), activation='tanh', return_sequences=True),
  Dropout(0.3),
  SimpleRNN(16, activation='tanh', return_sequences=True),
  Dropout(0.3),
  SimpleRNN(32, activation='tanh', return_sequences=False),
  Dense(units=1, activation=tf.keras.activations.sigmoid),
])

opt = tf.optimizers.Adam(learning_rate=learning_rate)

test_model(model, opt)
```

Blok 7 - Fragment kodu reprezentujący generację siódmej topologii



Rys. 21 - Wykresy przedstawiające dokładność i błąd na epokę oraz macierz pomyłek dla siódmej topologii

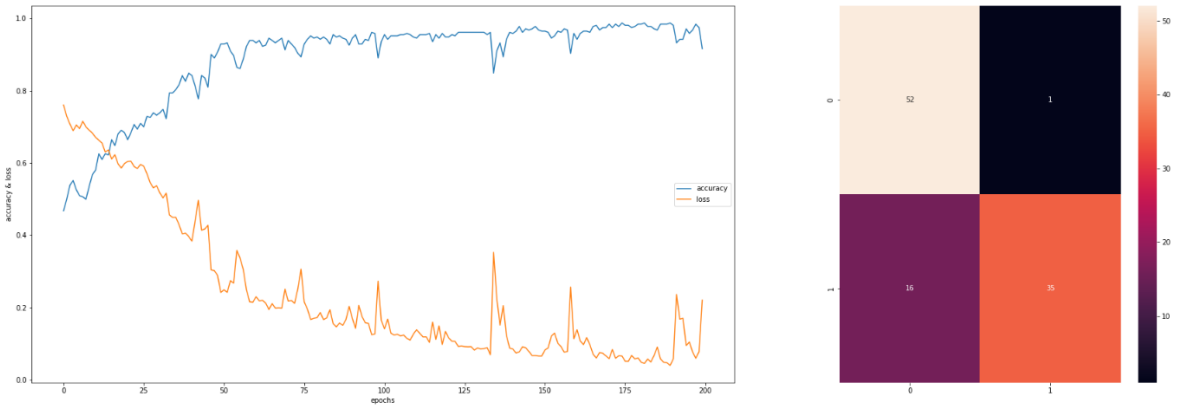
7.8 Topologia ósma

Train accuracy: 0.9161290526390076

Test accuracy: 83.65384615384616

```
model = Sequential([
  SimpleRNN(64, input_shape=(chunk_size, 9), activation='tanh', return_sequences=True),
  Dropout(0.3),
  SimpleRNN(32, activation='tanh', return_sequences=True),
  Dropout(0.2),
  SimpleRNN(16, activation='tanh', return_sequences=False),
  Dense(units=1, activation=tf.keras.activations.sigmoid),
])

opt = tf.optimizers.Adam(learning_rate=learning_rate)

test_model(model, opt)
```

Blok 8 - Fragment kodu reprezentujący generację ósmej topologii



Rys. 22 - Wykresy przedstawiające dokładność i błąd na epokę oraz macierz pomyłek dla ósmej topologii

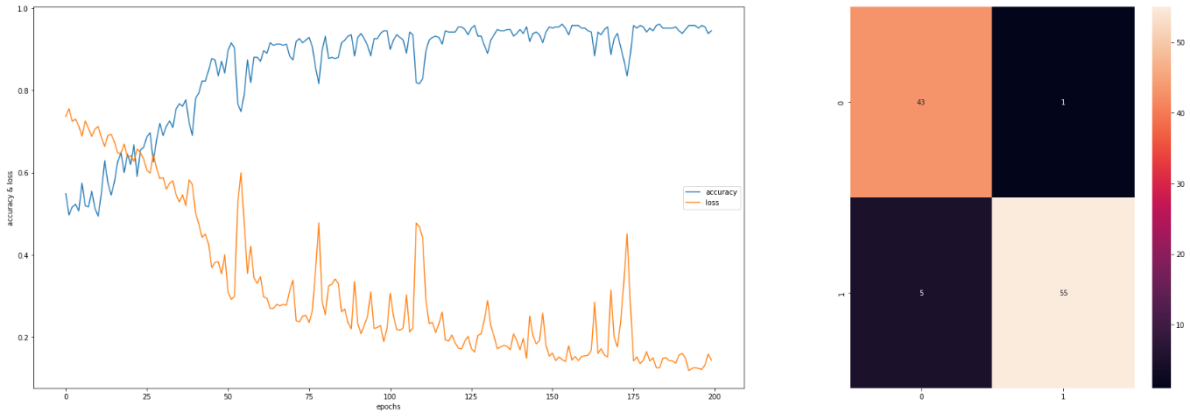
7.9 Topologia dziewiąta

Train accuracy: 0.9451612830162048

Test accuracy: 94.23076923076923

```
model = Sequential([
  SimpleRNN(64, input_shape=(chunk_size, 9), activation='tanh', return_sequences=True),
  Dropout(0.4),
  SimpleRNN(32, activation='tanh', return_sequences=True),
  Dropout(0.3),
  SimpleRNN(16, activation='tanh', return_sequences=True),
  Dropout(0.2),
  SimpleRNN(8, activation='tanh', return_sequences=False),
  Dense(units=1, activation=tf.keras.activations.sigmoid),
])

opt = tf.optimizers.Adam(learning_rate=learning_rate)

test_model(model, opt)
```

Blok 9 - Fragment kodu reprezentujący generację dziewiątej topologii



Rys. 23 - Wykresy przedstawiające dokładność i błąd na epokę oraz macierz pomyłek dla dziewiątej topologii

7.10 Topologia dziesiąta

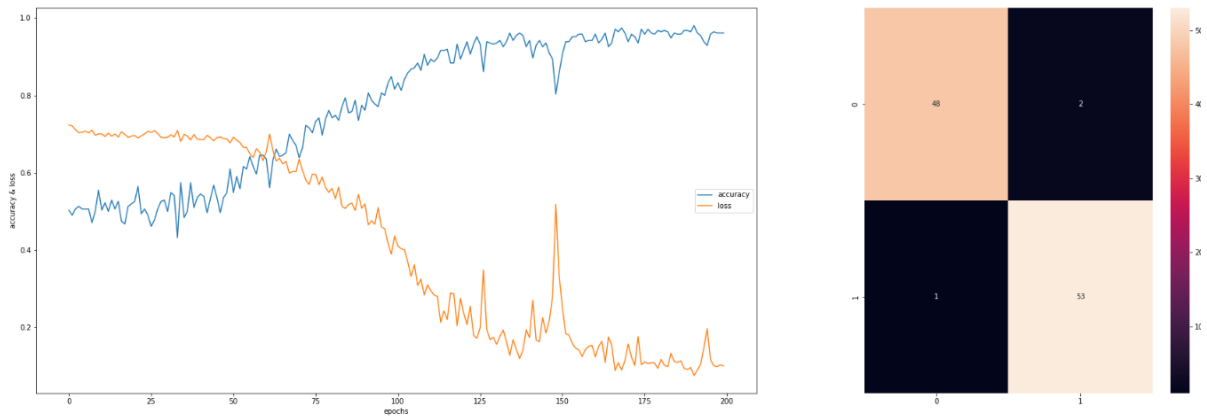
W ostatnim eksperymencie zaimplementowano architekturę zwężającą się w centrum do 2 warstw po 32 neurony każda aby następnie symetrycznie do wyjścia poszerzyć sieć do 64 neuronów. Pomiędzy każdą parą warstw ukrytych zastosowany został stopniowo malejący Dropout. Skuteczność tak zbudowanej sieci okazała się zadowalająca jednak po drodze nastąpiło jednorazowe znaczne zaburzenie procesu uczenia.

Train accuracy: 0.9612902998924255

Test accuracy: 97.11538461538461

```
model = Sequential([
  SimpleRNN(64, input_shape=(chunk_size, 9), activation='relu', return_sequences=True),
  Dropout(0.4),
  SimpleRNN(32, activation='tanh', return_sequences=True),
  Dropout(0.2),
  SimpleRNN(32, activation='tanh', return_sequences=True),
  Dropout(0.1),
  SimpleRNN(64, activation='relu', return_sequences=False),
  Dense(units=1, activation=tf.keras.activations.sigmoid),
])

opt = tf.optimizers.Adam(learning_rate=learning_rate)

test_model(model, opt)
```

Blok 10 - Fragment kodu reprezentujący generację dziesiątej topologii



Rys. 24 - Wykresy przedstawiające dokładność i błąd na epokę oraz macierz pomyłek dla dziesiątej topologii

8. Walidacja krzyżowa

Następnym etapem naszego projektu było przeprowadzenie walidacji krzyżowej na przetworzonych danych wejściowych z wykorzystaniem naszego modelu.

Źródło: https://www.section.io/engineering-education/how-to-implement-k-fold-cross-validation/

Rys. 24 - Wizualizacja walidacji krzyżowej z 5 podzbiorami

Na początku podjęliśmy się próby przeprowadzenia ręcznej walidacji krzyżowej bez użycia metody z biblioteki (próba dostępna jest tutaj), lecz przez pewne błędy w naszym rozumowaniu nie zadziałała ona tak jakbyśmy tego oczekiwali.

Dlatego też przy drugim podejściu użyta została metoda **cross_validate** zaimportowana z biblioteki **sklearn.** Umożliwia ona przeprowadzenie walidacji krzyżowej na przekazanych danych i parametrach. Możemy doprecyzować dane, model, metryki oraz liczbę tzw. **foldów** czyli pomniejszych zbiorów, na które podzielone zostaną nasze dane.

Implementacja poprawnej walidacji krzyżowej znajduje się w pliku **/model_and_cv5_single_notebooks/CV5_example.ipynb**

Przykładowe metryki dla uruchomienia walidacji krzyżowej z podziałem na 5 **foldów** przedstawiono w tabeli poniżej:

FOLD WALIDUJĄCY / METRYKA	Skuteczność (*accuracy*)	Precyzja (*precision*)	Czułość (*recall / sensitivity*)	Metryka F1	ROC AUC (*receiver operating characteristics area under curve*)
Pierwszy	0.84337349	0.83333333	0.85365854	0.84337349	0.85133566
Drugi	0.92771084	1.	0.85365854	0.92105263	0.94773519
Trzeci	0.74698795	0.70833333	0.82926829	0.76404494	0.82694541
Czwarty	0.90361446	0.90243902	0.90243902	0.90243902	0.93031359
Piąty	0.86585366	0.94117647	0.7804878	0.85333333	0.9369423

Tabela 1. Metryki dla kolejnych zbiorów (foldów) testujących (150 epok na uczenie) w każdym z przypadków

9. Najlepsza architektura sieci

Poniższa sieć jest wynikiem testów różnych topologii oraz parametrów, które dały razem nasz najlepszy wynik. Jest to topologia druga z rozdziału 7.

```
from keras.models import Sequential
from keras.layers import SimpleRNN, Dense, Dropout
import tensorflow as tf

def create_model(learning_rate=0.0005, chunk_size=8):
  model = Sequential()
  model.add(SimpleRNN(64, input_shape=(chunk_size, 9), activation='tanh', return_sequences=True))
  model.add(Dropout(0.4))
  model.add(SimpleRNN(8, activation='tanh', return_sequences=False))

  model.add(Dense(units=1, activation=tf.keras.activations.sigmoid))
  opt = tf.optimizers.Adam(learning_rate=learning_rate)
  model.compile(loss='binary_crossentropy', optimizer=opt, metrics=['accuracy'])

  return model
```

Blok 11 - Fragment kodu prezentujący najlepszą z testowanych architekturę sieci RNN

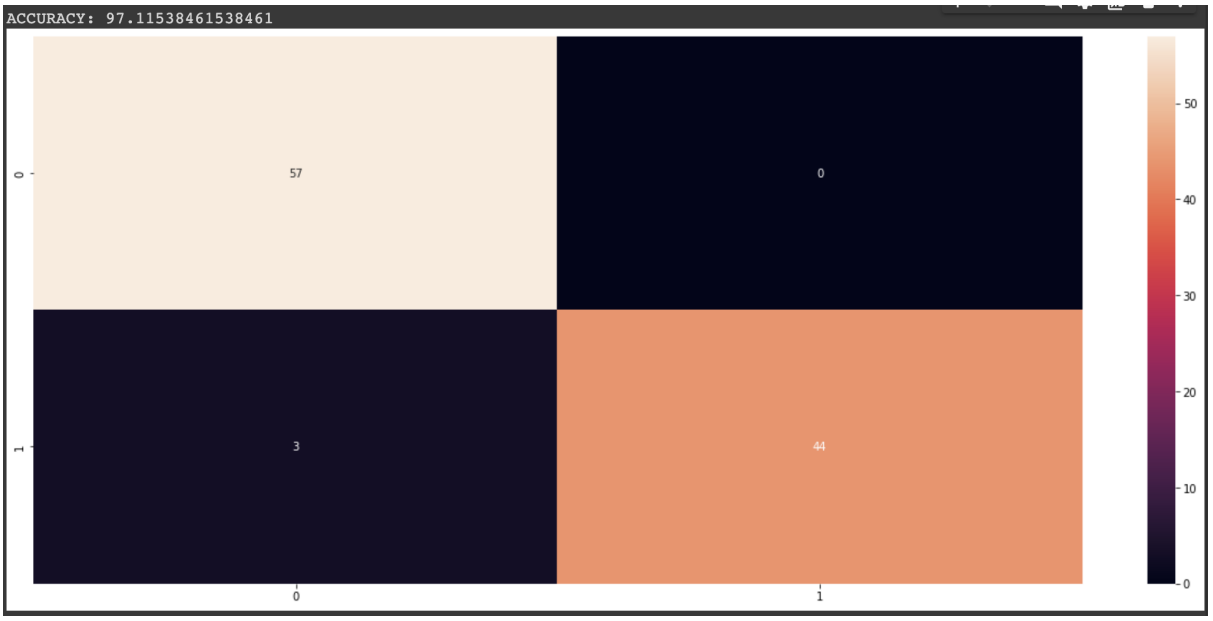
Poniższa sieć jest wynikiem testów różnych topologii oraz parametrów, które dały razem nasz najlepszy wynik. Wciąż jest to typowy model sekwencyjny, który rozpoczyna się od warstwy rekurencyjnej. **Input_shape** wynosi liczbę szeregów czasowych x 9 parametrów oraz funkcji aktywacyjnej będącej **tangensem hiperbolicznym**. Następnie mamy dropout o wartości **0.4**, czyli o 0.2 większej od inicjalnego. Kolejną warstwę rekurencyjna, o tej samej funkcji aktywacyjnej, lecz z wyłączonym zwracaniem sekwencji. Jako że dalej korzystamy z binarnej entropii krzyżowej jako funkcji strat, na końcu wciąż znajduje się **warstwa gęsta** z jednym neuronem i **sigmoidem** za funkcję aktywacji.

Funkcja zwracająca model z najlepszą z testowanych architektur znajduje się w pliku **models/best_model.ipynb**.

10. Przetrenowane modele

W folderze o nazwie **pretrained_models** znajdują się nauczone już modele, które możemy wykorzystać bezpośrednio do predykcji awarii urządzenia. Mamy tam model stworzony z optymalnej architektury oraz modele z testowanych w rozdziale 7 topologii.

W głównym katalogu projektu znajduje się również plik o nazwie **test_pretrained_models.ipynb**, w którym możemy przetestować nasze gotowe modele ładując je do pliku. Wystarczy jedynie zmienić ścieżkę do danego modelu wewnątrz metody **load** i pozwoli nam to na zobaczenie przewidywań przygotowanych przez dany model. By skorzystać z tego pliku możemy wrzucić go do aplikacji **Google Collab** wraz z danymi, ponieważ ma on zaimplementowane w sobie wszystkie funkcje potrzebne do koniecznego przetworzenia danych wejściowych pochodzące z pliku **data_processing.py**.



Rys. 25 - Wizualizacja skuteczności przetrenowanych modeli za pomocą macierzy pomyłek

11. Podsumowanie

Udało się nam spełnić wszystkie określone wymagania projektu - dane wejściowe zostały przeanalizowane za pomocą różnych statystyk, udało nam się otrzymać dość dobrze przewidujący model korzystający z **Rekurencyjnych Sieci Neuronowych**.

Skorzystaliśmy z walidacji krzyżowej oraz zaprezentowaliśmy dane wyjściowe na wykresach. Mogliśmy nauczyć się **praktycznego zastosowania sieci neuronowych** w realnej sytuacji - dowiedzieliśmy się jak zmiany pewnych parametrów wewnątrz modelu wpływają na jego dokładność i wyniki.