



Sieci neuronowe

Raport ćw. 1-4

AUTOR

Szymon Sawczuk

nr albumu: **260287**

kierunek: **Informatyka Stosowana**

17 listopada 2023

Spis treści

1	Ćwiczenie 1 - Analiza danych	2
1.1	Biblioteki użyte w tym ćwiczeniu	2
1.2	Eksploracja danych	2
1.3	Przygotowanie macierzy cech liczbowych	6
1.4	Wnioski	8
2	Ćwiczenie 2 - Model sieci neuronowej wykorzystującej regresję logistyczną	9
2.1	Implementacja modelu	9
2.2	Test modelu na bazie danych z ćwiczenia 1	11
2.3	Podsumowanie wyników	12
3	Ćwiczenie 3 - Model wielowarstwowy z propagacją wsteczną	13
3.1	Implementacja modelu	13
3.2	Badanie modelu na zbiorze heart disease	18
3.3	Podsumowanie wyników	21
4	Ćwiczenie 4 - Model wielowarstwowy przy użyciu narzędzi	22
4.1	Odtworzenie architektury sieci	22
4.2	Badanie modelu na zbiorze heart disease	24
4.3	Podsumowanie wyników	29
	Literatura	30

1 Ćwiczenie 1 - Analiza danych

Celem ćwiczenia było zapoznanie (bądź przypomnienie) się z bibliotekami i narzędziami, które wykorzystywane są do uczenia maszynowego, eksploracji danych oraz ewaluacji sieci neuronowych, a także analiza zbioru danych wykorzystywanych do tego i dalszych ćwiczeń. (*Heart Disease Dataset*, 1988)

1.1 Biblioteki użyte w tym ćwiczeniu

W tym ćwiczeniu wykorzystałem 3 biblioteki dostępne dla języka python:

- pandas - biblioteka pozwalająca na łatwe tworzenie zbiorów danych oraz ich eksplorację i modyfikację
- matplotlib - do tworzenia wykresów danych
- seaborn - dla zaawansowanych wizualizacji danych np. mapy ciepła

Zapoznałem się także z bibliotekami, które będą potrzebne do kolejnych ćwiczeń: numpy (biblioteka do operacji na wielowymiarowych tabelach/macierzach), Scikit-learn (dająca implementacje algorytmów do preprocessing'u oraz algorytmów uczenia maszynowego).

1.2 Eksploracja danych

Po załadowaniu danych poprzez prosty skrypt podany na stronie zbioru danych (*Heart Disease Dataset*, 1988), uzyskałem 14 kolumn w zbiorze danych:

- age (liczbowa) - wiek osoby (lata)
- sex (kategoryczna) - płeć osoby (0 - kobieta, 1 - mężczyzna)
- cp (kategoryczna) - typ bólu klatki piersiowej (wartości 1-4)
- trestbps (liczbowa) - ciśnienie krwi w spoczynku (mmHg)
- chol (liczbowa) - poziom cholesterolu w surowicy (mg/dl)
- fbs (kategoryczna) - poziom cukru we krwi na czczo (0 - nie, 1 - tak)
- restecg (kategoryczna) - wynik elektrokardiografii w spoczynku (0 - normalny, 1 - ST-T anormalność, 2 - hipertrofia)
- thalach (liczbowa) - maksymalne tętno osiągnięte podczas testu wysiłkowego
- exang (kategoryczna) - dławica wysiłkowa (0 - nie, 1 - tak)
- oldpeak (liczbowa) - depresja odcinka ST wywołana przez ćwiczenia w stosunku do odpoczynku
- slope (kategoryczna) - nachylenie odcinka ST podczas ćwiczeń (0 - wnoszące, 1 - płaskie, 2 - opadające)
- ca (liczbowa) - liczba głównych naczyń (0-3), podczas badania fluoroskopowego
- thal (kategoryczna) - rodzaj defektu (3 - normalny, 6 - uleczony defekt, 7 - odwracalny defekt)
- num - obecność choroby serca (0 - brak, 1,2,3,4 - obecność (czym większa wartość tym poważniejsza choroba))

Dane składają się z 303 próbek oraz 13 cech, kolumna num określa nam obecność choroby serca, albo jej brak.

```
heart_data.sample(10)
```

	age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal	num
261	58	0	2	136	319	1	2	152	0	0.0	1	2.0	3.0	3
169	45	0	2	112	160	0	0	138	0	0.0	2	0.0	3.0	0
40	65	0	4	150	225	0	2	114	0	1.0	2	3.0	7.0	4
241	41	0	2	126	306	0	0	163	0	0.0	1	0.0	3.0	0
12	56	1	3	130	256	1	2	142	1	0.6	2	1.0	6.0	2
259	57	1	2	124	261	0	0	141	0	0.3	1	0.0	7.0	1
201	64	0	4	180	325	0	0	154	1	0.0	1	0.0	3.0	0
246	58	1	4	100	234	0	0	156	0	0.1	1	1.0	7.0	2
99	48	1	4	122	222	0	2	186	0	0.0	1	0.0	3.0	0
255	42	0	3	120	209	0	0	173	0	0.0	2	0.0	3.0	0

Rysunek 1: 10 losowych przykładowych danych po wczytaniu

Pierwszą rzeczą jaką zbadałem było zbalansowanie danych względem liczby próbek w klasie.

```
heart_data["num"].value_counts()
```

num	count
0	164
1	55
2	36
3	35
4	13

Rysunek 2: Liczba próbek w klasach zbioru

Wyniki wskazują na brak zbalansowania danych pod względem liczby próbek na klasy. 164 próbki (około 54%) są próbkami zdrowych pacjentów (bez wykazanych problemów z sercem). Natomiast osób z zdiagnozowanymi najpoważniejszymi chorobami serca (klasa 4) jest tylko 4%. Rozwiązanie tego problemu wytłumaczone zostanie w następnym podrozdziale.

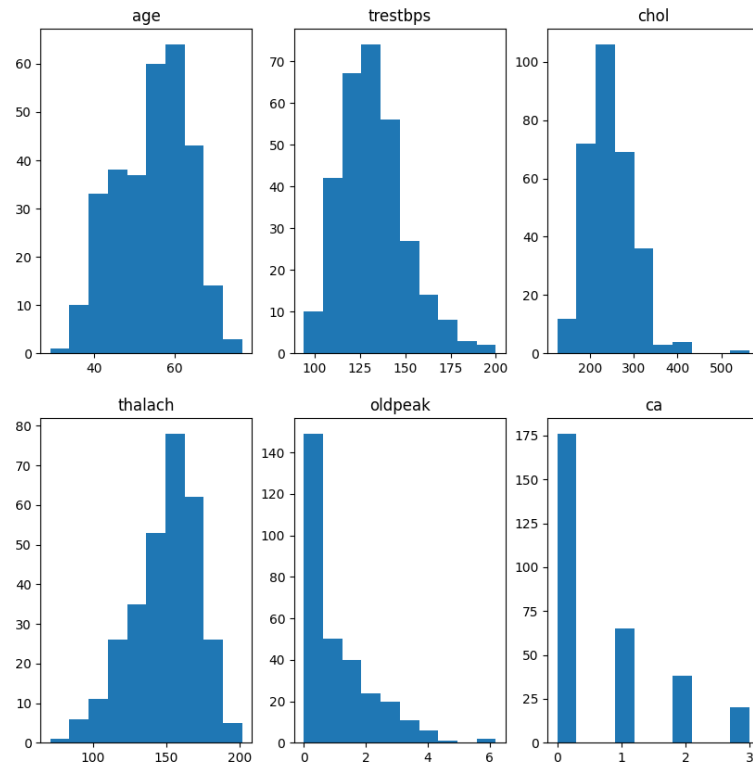
Następnym elementem badanym były wartości średnie oraz odchylenia standardowe cech liczbowych zbioru.

<pre>heart_data[num_features].mean()</pre> <table> <thead> <tr> <th></th><th>mean</th></tr> </thead> <tbody> <tr> <td>age</td><td>54.438944</td></tr> <tr> <td>trestbps</td><td>131.689769</td></tr> <tr> <td>chol</td><td>246.693069</td></tr> <tr> <td>thalach</td><td>149.607261</td></tr> <tr> <td>oldpeak</td><td>1.039604</td></tr> <tr> <td>ca</td><td>0.672241</td></tr> </tbody> </table>		mean	age	54.438944	trestbps	131.689769	chol	246.693069	thalach	149.607261	oldpeak	1.039604	ca	0.672241	<pre>heart_data[num_features].std()</pre> <table> <thead> <tr> <th></th><th>std</th></tr> </thead> <tbody> <tr> <td>age</td><td>9.038662</td></tr> <tr> <td>trestbps</td><td>17.599748</td></tr> <tr> <td>chol</td><td>51.776918</td></tr> <tr> <td>thalach</td><td>22.875003</td></tr> <tr> <td>oldpeak</td><td>1.161075</td></tr> <tr> <td>ca</td><td>0.937438</td></tr> </tbody> </table>		std	age	9.038662	trestbps	17.599748	chol	51.776918	thalach	22.875003	oldpeak	1.161075	ca	0.937438
	mean																												
age	54.438944																												
trestbps	131.689769																												
chol	246.693069																												
thalach	149.607261																												
oldpeak	1.039604																												
ca	0.672241																												
	std																												
age	9.038662																												
trestbps	17.599748																												
chol	51.776918																												
thalach	22.875003																												
oldpeak	1.161075																												
ca	0.937438																												

Rysunek 3: Wartości średnie oraz odchylenia standardowe cech liczbowych

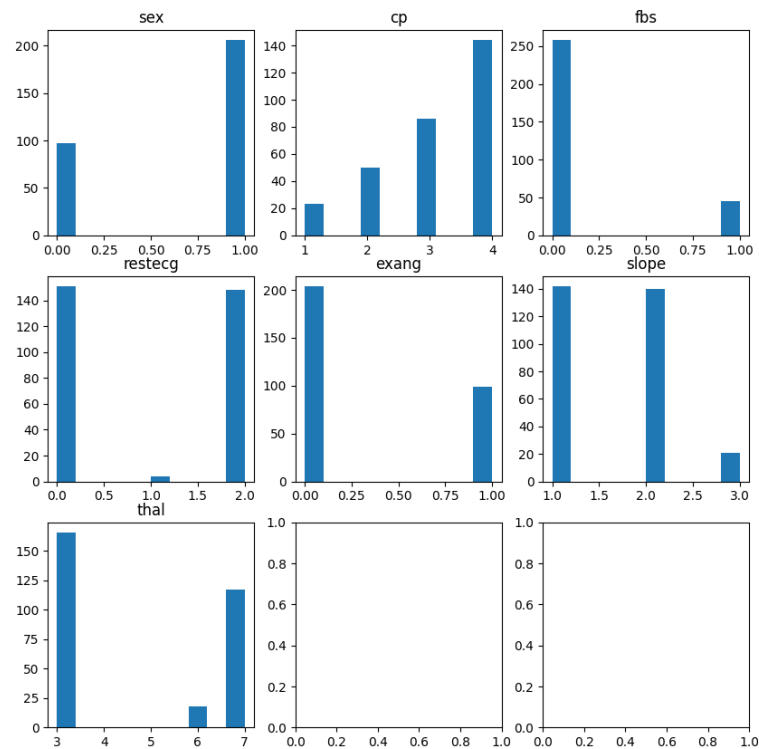
Dla przykładu średnia wartość wieku w zbiorze danych wynosi 54 lata, a około 70% danych mieści się między wiekiem 45, a 63. Widać już tutaj potencjalne rozkłady niektórych cech np. wieku. Natomiast histogramy wykazane poniżej wykazują, że cechy wieku, ciśnienia krwi w spoczynku, poziomu cholesterolu oraz maksymalnego osiągniętego tętna układają się w przybliżeniu zgodnie z wykresem

Gausa, zatem posiadają one rozkłady normalne. Dane depresji odcinka ST (oldpeak) oraz liczba naczyń zaobserwowanych poprzez fluoroskopię (ca) nie wykazują rozkładu normalnego, bardziej rozkład wykładniczy.



Rysunek 4: Histogramy cech liczbowych

Weźmy teraz pod lupę cechy katagoryczne i czy są one w przybliżeniu równomierne.



Rysunek 5: Histogramy cech katagorycznych

Na powyższych histogramach cech kategorycznych nie widać, aby jakkolwiek cecha miała zrównoważone dane. Najbliżej jednak takiego rozkładu równomiernego są cechy danych elektrokardiograficznych (restecg) oraz nachylenie odcinka ST (slope). Z danych nierównomiernych widać np. że większą ilością badanych byli mężczyźni.

W zbiorze odnalazłem 2 cechy, które posiadają wartości puste jest to ca oraz thal. Łącznie wartości pustych jest 6.

```
heart_data[heart_data["ca"].isnull()]
```

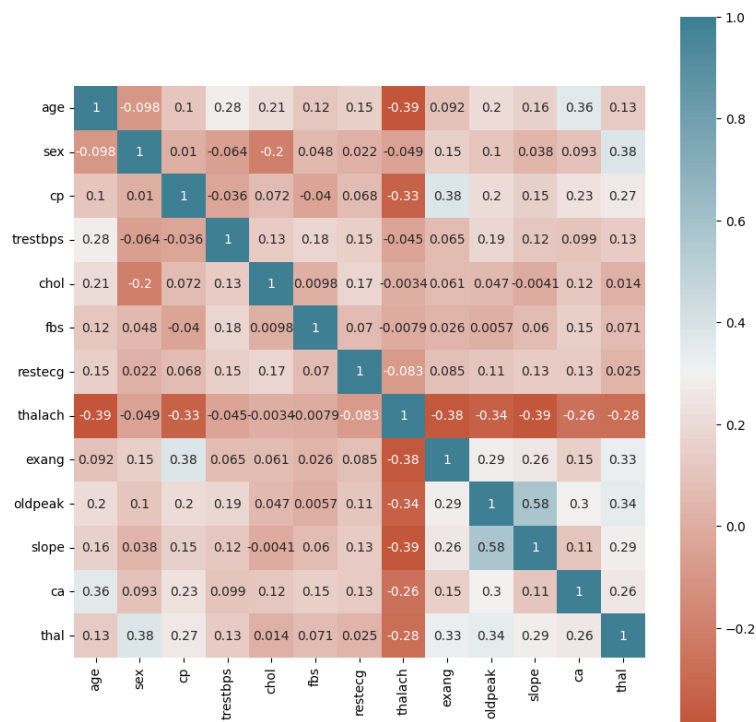
age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal	num
166	52	1	3	138	223	0	0	169	0	0.0	1	NaN	3.0
192	43	1	4	132	247	1	2	143	1	0.1	2	NaN	7.0
287	58	1	2	125	220	0	0	144	0	0.4	2	NaN	7.0
302	38	1	3	138	175	0	0	173	0	0.0	1	NaN	3.0


```
heart_data[heart_data["thal"].isnull()]
```

age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal	num
87	53	0	3	128	216	0	2	115	0	0.0	1	0.0	NaN
266	52	1	4	128	204	1	0	156	1	1.0	2	0.0	NaN

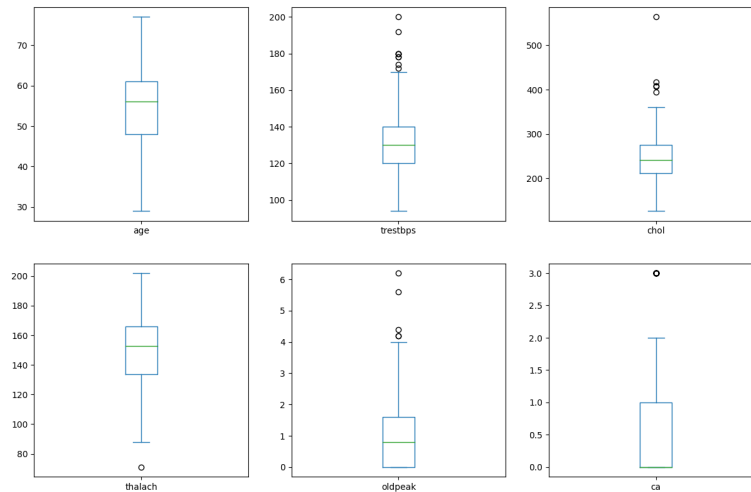
Rysunek 6: Wartości puste zbioru

Jest wiele sposobów na rozwiązanie tego problemu np. uzupełnienie brakujących danych w sposób sztuczny używając mediany, albo algorytmu k-najbliższych sąsiadów (*K najbliższych sąsiadów*, 2022). Natomiast z powodu małej ilości danych brakujących (około 2% danych), możemy najprościej usunąć te dane ze zbioru, bez znaczącej utraty informacji.



Rysunek 7: Wykres ciepła dla korelacji między cechami

Mapy ciepła zamieszczone powyżej pokazują poziom korelacji cech danych. Przy wysokich wartościach korelacji możnaby rozważyć usunięcie jednej z tych cech np. (slope i oldpeak), mogłoby to pomóc w uzyskaniu lepszych wyników nauczania. Warto jednak rozważyć także sens merytoryczny tych dwóch cech, czy jednak nie są one znaczące dla całego modelu.



Rysunek 8: Wykresy pudełkowe dla cech liczbowych

Powyższe wykresy pudełkowe wskazują nam na rozłożenie wartości danych cech. Widzimy, że dane `ca`, `oldepeak` są w mniejszym zakresie niż np. wiek. Takie dane o małych zakresach mogą zostać przykryte w niektórych modelach przez cechy o większych zakresach. Warto też przyrzeć się danym odbiegającym od kwartyli cechy (można rozważyć ich usunięcie).

1.3 Przygotowanie macierzy cech liczbowych

Po wyciągnięciu cech liczbowych ze zbioru danych zająłem się rozwiązaniem problemu braku zbilansowania próbek względem klasyfikacji zbioru. Zdecydowałem na naprawę braku zbalansowania próbek poprzez zmniejszenie klasyfikacji do klasyfikacji binarnej (0 - zdrowy, 1 - choroba serca), ponieważ klasy 1-4 oznaczały inne stopnie problemów z sercem, które można na potrzeby modelu budowanego zmniejszyć do tej samej klasyfikacji. Owe rozwiązanie pozwoliło także na zmniejszenie ilości danych potrzebnych do usunięcia, aby klasyfikacje były zbilansowane. W wyniku uzyskałem zmniejszony zbiór do 274 próbek, ale ze zbilansowanymi próbkami względem klas.

```
# repairing of the imbalance in classification and removing null values
df["num"] = df["num"].replace([2, 3, 4], 1) #change classes to binary classification
print(df["num"].value_counts())

#get null values of ca and remove them
null_idx = df[df["ca"].isnull()].index
print(null_idx)
df = df.drop(null_idx)
df = df.reset_index(drop=True)
print(df["num"].value_counts())

#get null values of thal and remove them
null_idx = df[df["thal"].isnull()].index
print(null_idx)
df = df.drop(null_idx)
df = df.reset_index(drop=True)
print(df["num"].value_counts())

# balance classes to same amount 138
random_idx = df.query("num == 0").sample(df["num"].value_counts()[0] - df["num"].value_counts()[1]).index
df = df.drop(random_idx)
df = df.reset_index(drop=True)
print(df["num"].value_counts())
```

Rysunek 9: Kod naprawiający zbilansowanie próbek

Usunięcie tych danych pozwoliło także na pozbycie się wartości pustych dla cechy `ca` oraz `thal`.

Wykorzystując metodę `get_dummies` z biblioteki `pandas` zamieniłem dane katagoryczne (pomijając kategorie z wartościami 0 i 1) na rozdzielone kolumny danych o wartościach 0 lub 1.

```
def category_to_dummy(column: str, targets: list[str]):
    result_categories = pd.get_dummies(column).astype(int)
    result_categories.columns = targets
    return result_categories

df_without_num = pd.concat([df,
                             category_to_dummy(df["cp"], ["typical angina", "atypical angina", "non-anginal pain", "asymptomatic"]),
                             category_to_dummy(df["restecg"], ["normal", "having ST-T wave abnormality", "left ventricular hypertrophy by Estes' criteria"]),
                             category_to_dummy(df["slope"], ["upsloping", "flat", "downsloping"]),
                             category_to_dummy(df["thal"], ["normal", "fixed defect", "reversable defect"]),
                             ], axis=1)

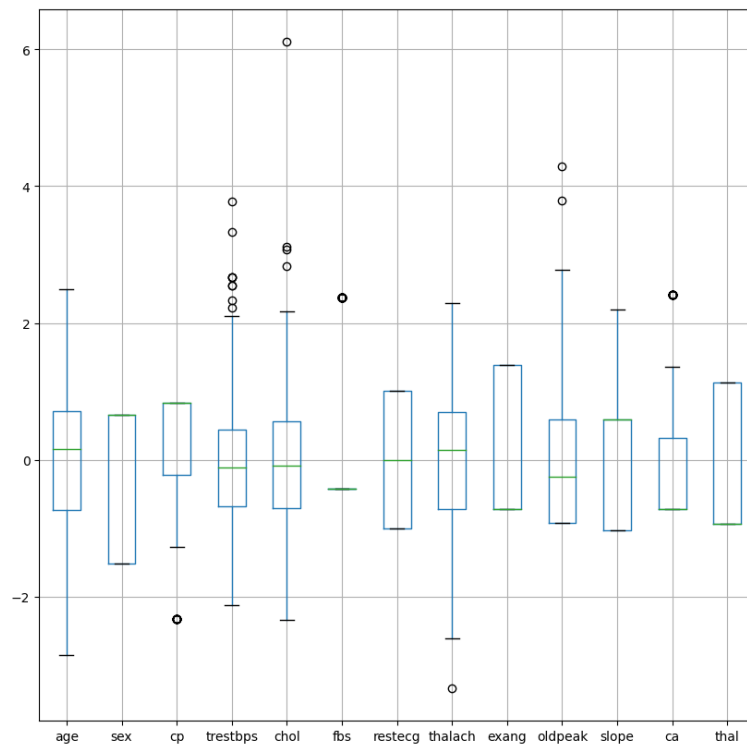
df_without_num = df_without_num.loc[:, df_without_num.columns != "cp"]
df_without_num = df_without_num.loc[:, df_without_num.columns != "restecg"]
df_without_num = df_without_num.loc[:, df_without_num.columns != "slope"]
df_without_num = df_without_num.loc[:, df_without_num.columns != "thal"]
df_without_num = df_without_num.loc[:, df_without_num.columns != "num"]
```

Rysunek 10: Zamiana wartości katagorycznych na liczbowe

Następnie rozwiązałem problem różnych zakresów cech liczbowych poprzez standaryzację cech, w taki sposób zachowane pozostały rozkłady owych cech. Standaryzację wykonałem za pomocą wzoru

$$z = \frac{x - \mu}{\sigma}, \text{ gdzie:}$$

- x – zmienna niestandaryzowana,
- μ – średnia z populacji,
- σ – odchylenie standardowe populacji.



Rysunek 11: Wykresy pudełkowe cech po standaryzacji

Wynikiem wszystkich tych operacji jest gotowa macierz cech liczbowych z przykładami, którą można wykorzystać do dalszych ćwiczeń.

	age	sex	trestbps	chol	fbs	thalach	exang	oldpeak	ca	typical angina	...	normal	having ST-T wave abnormality	left ventricular hypertrophy by Estes' criteria	upsloping	flat	downsloping	normal	fixed defect	reversible defect	heart_disease
0	1.369345	0.680840	1.581474	0.716989	-0.406635	-1.782080	1.427229	0.346002	2.399814	-0.256723	...	-0.976550	-0.121494	1.005486	-0.907551	1.042890	-0.272466	0.921002	-0.231624	-0.829961	1
1	1.369345	0.680840	-0.668682	-0.360461	-0.406635	-0.868494	1.427229	1.272156	1.351092	-0.256723	...	-0.976550	-0.121494	1.005486	-0.907551	1.042890	-0.272466	-1.081812	-0.231624	1.200479	1
2	-1.933477	0.680840	-0.106143	0.036494	-0.406635	1.654743	-0.698101	2.029918	-0.746354	-0.256723	...	1.020276	-0.121494	-0.990914	-0.907551	-0.955375	3.656784	0.921002	-0.231624	-0.829961	0
3	-1.493101	-1.463414	-0.106143	-0.833026	-0.406635	1.002182	-0.698101	0.261806	-0.746354	-0.256723	...	-0.976550	-0.121494	1.005486	1.097844	-0.955375	-0.272466	0.921002	-0.231624	-0.829961	0
4	0.158310	0.680840	-0.668682	-0.228142	-0.406635	1.263206	-0.698101	-0.243369	-0.746354	-0.256723	...	1.020276	-0.121494	-0.990914	1.097844	-0.955375	-0.272466	0.921002	-0.231624	-0.829961	0
...
269	0.268405	-1.463414	0.456396	-0.133629	-0.406635	-1.129518	1.427229	-0.748544	-0.746354	-0.256723	...	1.020276	-0.121494	-0.990914	-0.907551	1.042890	-0.272466	-1.081812	-0.231624	1.200479	1
270	-1.052724	0.680840	-1.231221	0.301131	-0.406635	-0.737982	-0.698101	0.093414	-0.746354	3.881040	...	1.020276	-0.121494	-0.990914	-0.907551	1.042890	-0.272466	-1.081812	-0.231624	1.200479	1
271	1.479439	0.680840	0.681411	-1.040955	2.450235	-0.346445	-0.698101	1.945722	1.351092	-0.256723	...	1.020276	-0.121494	-0.990914	-0.907551	1.042890	-0.272466	-1.081812	-0.231624	1.200479	1
272	0.268405	0.680840	-0.106143	-2.212918	-0.406635	-1.477551	1.427229	0.093414	0.302369	-0.256723	...	1.020276	-0.121494	-0.990914	-0.907551	1.042890	-0.272466	-1.081812	-0.231624	1.200479	1
273	0.268405	-1.463414	-0.106143	-0.228142	-0.406635	1.089190	-0.698101	-0.918935	0.302369	-0.256723	...	-0.976550	-0.121494	1.005486	-0.907551	1.042890	-0.272466	0.921002	-0.231624	-0.829961	1

Rysunek 12: Wynikowa macierz cech liczbowych

1.4 Wnioski

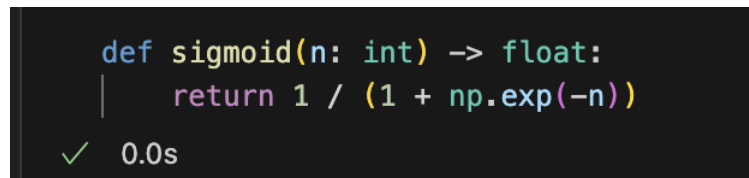
- Analiza danych pozwala nam na lepsze zrozumienie zbioru, a także naprawę problemów w zbiorze, które mogą spowodować gorsze wyniki naszego modelu.
- Warto zwrócić uwagę na zbalansowanie klasyfikacji w próbkach, gdyż brak owego zbalansowania może nauczyć model rozpoznawania poprawnej klasyfikacji tylko w kilku z nich (tych klas, których jest najwięcej).
- Innymi wartymi uwagi problemami jakie mogą pojawić się w zbiorze są brakujące dane, nierówne zakresy cech, bądź zbyt duża korelacja danych.
- Warto zwrócić uwagę na rozkłady cech w zbiorze, ponieważ może nam to pomóc w wyborze odpowiedniego modelu do rozwiązania naszego zadania.
- Analiza danych to proces iteracyjny, który nie raz wymaga wielu kroków, warto wspomóc się bibliotekami np. dla języka python, które pomagają nam na np. szybszą operację na danych oraz różne wizualizacje zbioru danych.

2 Ćwiczenie 2 - Model sieci neuronowej wykorzystującej regresję logistyczną

Celem ćwiczenia było przygotowanie własnoręcznego modelu sieci neuronowej wykorzystującej regresję logistyczną, a później przetestowanie działania wykorzystując dane przygotowane w ćwiczeniu 1. Do zadania wykorzystałem biblioteki numpy (do operacji na macierzach), pandas (do przygotowania danych jak w ćwiczeniu 1), matplotlib (wykonanie wykresów przedstawiający wyniki uczenia się modelu) oraz scikit learn (do łatwej weryfikacji wyników sklearn.metrics)

2.1 Implementacja modelu

Wyjście modelu opiera się o funkcję sigmoidu:



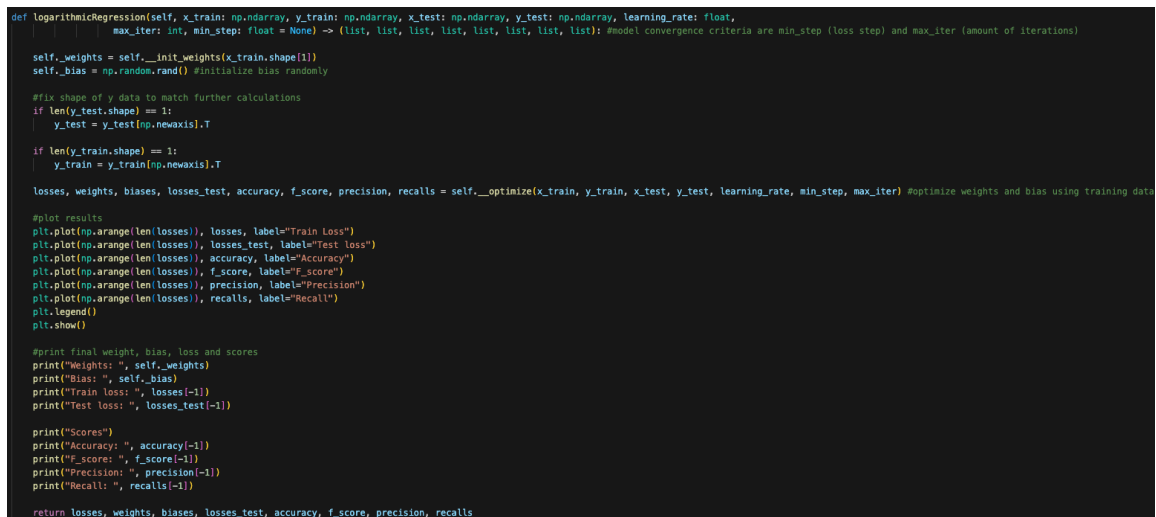
```
def sigmoid(n: int) -> float:
    return 1 / (1 + np.exp(-n))
```

✓ 0.0s

Rysunek 13: Funkcja sigmoidu

Warto jednak zauważyć, że z powodu limitu typu float przy większym wejściu (np. 42) funkcja ta zwróci nam wartość 1, co może powodować problemy przy obliczaniu dalszym logarytmu naturalnego z $(1 - \text{sigmoid})$, ponieważ $\ln(0)$ jest nieokreślony. Przy danych z poprzedniego ćwiczenia, nie pojawi się ten problem z powodu wcześniejszej standaryzacji danych (dane mają małe wartości).

Zaimplementowałem klasę `LogarithmicRegression`, która posiada atrybuty prywatne wag oraz bias'u. Główna metoda wywołująca uczenie modelu nazywa się `logarithmicRegression` i przyjmuje ona dane treningowe oraz testowe, współczynnik uczenia, oraz warunki zbieżności modelu (pewna maksymalna liczba iteracji i wystarczająco mała zmiana funkcji kosztu w danej iteracji i)



```
def logarithmicRegression(self, x_train: np.ndarray, y_train: np.ndarray, x_test: np.ndarray, y_test: np.ndarray, learning_rate: float,
                           max_iter: int, min_step: float = None) -> (list, list, list, list, list, list, list, list): #model convergence criteria are min_step (loss step) and max_iter (amount of iterations)

    self._weights = self._init_weights(x_train.shape[1])
    self._bias = np.random.rand() #initialize bias randomly

    #fix shape of y data to match further calculations
    if len(y_test.shape) == 1:
        y_test = y_test[np.newaxis].T

    if len(y_train.shape) == 1:
        y_train = y_train[np.newaxis].T

    losses, weights, biases, losses_test, accuracy, f_score, precision, recalls = self._optimize(x_train, y_train, x_test, y_test, learning_rate, min_step, max_iter) #optimize weights and bias using training data

    #plot results
    plt.plot(np.arange(len(losses)), losses, label="Train Loss")
    plt.plot(np.arange(len(losses)), losses_test, label="Test Loss")
    plt.plot(np.arange(len(losses)), accuracy, label="Accuracy")
    plt.plot(np.arange(len(losses)), f_score, label="F_score")
    plt.plot(np.arange(len(losses)), precision, label="Precision")
    plt.plot(np.arange(len(losses)), recalls, label="Recall")
    plt.legend()
    plt.show()

    #print final weight, bias, loss and scores
    print("Weights: ", self._weights)
    print("Bias: ", self._bias)
    print("Train loss: ", losses[-1])
    print("Test loss: ", losses_test[-1])

    print("Scores")
    print("Accuracy: ", accuracy[-1])
    print("F_score: ", f_score[-1])
    print("Precision: ", precision[-1])
    print("Recall: ", recalls[-1])

    return losses, weights, biases, losses_test, accuracy, f_score, precision, recalls
```

Rysunek 14: Główna metoda klasy `LogarithmicRegression`

W pierwszym kroku inicjalizuję wartości wag oraz bias'u poprzez wartości losowe. Następnie po przygotowanie wymiarów macierzy wartości y (wyniki przykładów) dokonuję optymalizacji wag i bias'u:

```
def __optimize(self, x_train: np.ndarray, y_train: np.ndarray, x_test: np.ndarray, y_test: np.ndarray, learning_rate: float,
               min_step: float, max_iter: int) -> (list, list, list, list, list, list, list, list):
    losses = []
    weights = []
    biases = []
    losses_test = []

    accuracy = []
    f_score = []
    precision = []
    recalls = []
    for index in range(max_iter): #learn for max_iter

        old_train_loss = self.__cross_entropy_loss(self.weights, x_train, y_train, self.bias)

        #calculate new weight and bias using step
        self.weights = self.weights - learning_rate * self.__gradient_weights(self.weights, x_train, y_train, self.bias)
        self.bias = self.bias - learning_rate * self.__gradient_bias(self.weights, x_train, y_train, self.bias)

        #calculate loss of training and testing data
        new_train_loss = self.__cross_entropy_loss(self.weights, x_train, y_train, self.bias)
        new_test_loss = self.__cross_entropy_loss(self.weights, x_test, y_test, self.bias)

        #append to helper lists
        losses.append(new_train_loss)
        weights.append(self.weights)
        biases.append(self.bias)
        losses_test.append(new_test_loss)

        #calculate scores for each iteration
        y_pred = self.predict(x_test)
        accuracy.append(metrics.accuracy_score(y_test, y_pred))
        f_score.append(metrics.f1_score(y_test, y_pred))
        precision.append(metrics.precision_score(y_test, y_pred))
        recalls.append(metrics.recall_score(y_test, y_pred))

        if min_step is not None and index > 0 and abs(old_train_loss - new_train_loss) <= min_step: #if change of loss is smaller or equal than min_step when stop learning
            break

    return losses, weights, biases, losses_test, accuracy, f_score, precision, recalls
```

Rysunek 15: Metoda optymalizacji

Wagi zostają przesunięte o wartość gradientu wyliczanego dla całego zbioru danych zgodnie z wzorem $w' = w - learning_rate * gradient_wag$:

```
def __gradient_weights(self, weights: np.ndarray, x: np.ndarray, y: np.ndarray, bias: float) -> np.ndarray:
    return -np.dot(x.T, (y - sigmoid(np.dot(x, weights) + bias))) / x.shape[0] #calculate gradient of weights using whole training data
```

Rysunek 16: Gradient wag

Dla bias'u natomiast dane x wynoszą 1 więc możemy uprościć wyliczenie gradientu do:

```
def __gradient_bias(self, weights: np.ndarray, x: np.ndarray, y: np.ndarray, bias: float) -> np.ndarray:
    return np.sum(-(y - sigmoid(np.dot(x, weights) + bias))) / x.shape[0] #calculate gradient of bias using whole training data
```

Rysunek 17: Gradient bias'u

$x.shape[0]$ oznacza ilość przykładów w macierzy danych treningowych.

Następnie obliczony jest koszt po obliczeniu nowych wag i bias'u:

```
def __cross_entropy_loss(self, weights: np.ndarray, x: np.ndarray, y: np.ndarray, bias: float) -> np.ndarray:
    return np.sum(-y*np.log(sigmoid(np.dot(x, weights) + bias)) - (1 - y)*np.log(1 - sigmoid(np.dot(x, weights) + bias))) / x.shape[0] #calculate loss using whole training data
```

Rysunek 18: Wzór kosztu funkcji

Dla każdej iteracji obliczony zostaje także wynik, względem predykcji wyników dla danych testowych:

Sigmoid daje nam prawdopodobieństwo w przedziale (0, 1), więc wyniki należy zaokrąglić do pełnego 0, albo 1 (próg 0.5).

```
def predict(self, x_test: np.ndarray) -> np.ndarray:

    y_pred = sigmoid(np.dot(x_test, self._weights) + self._bias) #calculate sigmoid for input data

    #change possibility from sigmoid to 1 or 0
    y_pred[y_pred >= 0.5] = 1
    y_pred[y_pred < 0.5] = 0

    return y_pred
```

Rysunek 19: Metoda predykcji danych

Na koniec sprawdzany jest warunek zbieżności modelu - wystarczająco mała zmiana funkcji kosztu w danej iteracji i.

Wyniki optymalizacji oraz przebieg kosztów w trakcie iteracji są przedstawiane za pomocą wykresu oraz zwracane jako wynik metody `logarithmicRegression`.

2.2 Test modelu na bazie danych z ćwiczenia 1

Po wczytaniu danych zgodnie z przebiegiem ćwiczenia 1, należy jeszcze podzielić dane na dane uczące i testowe. Wykonałem to w sposób losowy, dla pewnej ilości danych uczących względem testowych (np. 70% danych treningowych, 30% testowych):

```
def train_test_split(features, targets, percentage):
    choices = np.random.choice(range(features.shape[0]), size=(int(features.shape[0] * percentage/100),), replace=False)
    split = np.zeros(features.shape[0], dtype=bool)
    split[choices] = True

    return features[split], targets[split], features[~split], targets[~split]

features = result.loc[:, result.columns != "heart_disease"].to_numpy()
targets = result["heart_disease"].to_numpy()

x_train, y_train, x_test, y_test = train_test_split(features, targets, 70)
```

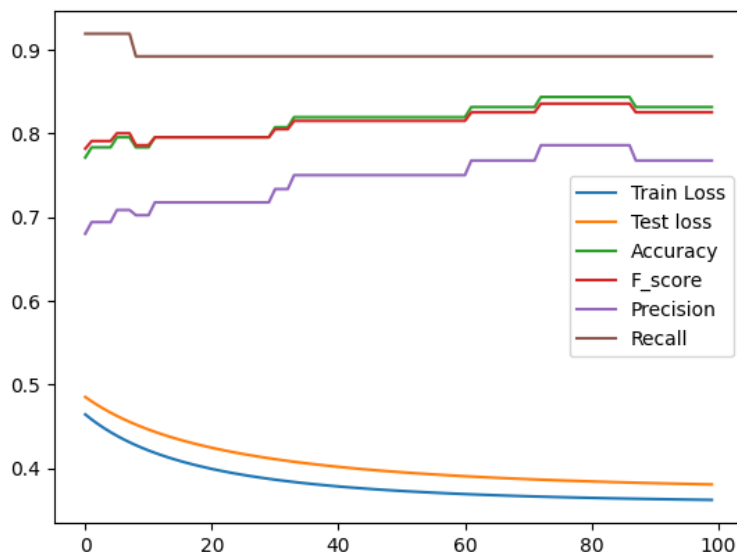
Rysunek 20: Funkcja dzieląca zbiór na dwa zbiory względem pewnego stosunku

Dla takich podzielonych danych oraz wartości współczynnika uczenia - 0.2, ilości iteracji - 200 oraz wartości progu małego spadku funkcji kosztu - 0.0001 uzyskujemy wyniki:

```
Weights: [[-0.11786781]
 [ 0.65462996]
 [ 0.43297219]
 [ 0.25495492]
 [ 0.04797949]
 [-0.10275015]
 [ 0.18658021]
 [-0.28018111]
 [ 0.51133107]
 [ 0.40773473]
 [ 0.40899077]
 [ 1.09021618]
 [ 0.66263916]]
Bias: 0.38287959027390495
Train loss: 0.3619743755584501
Test loss: 0.3804651798847365
Scores
Accuracy: 0.8313253012048193
F_score: 0.825
Precision: 0.7674418604651163
Recall: 0.8918918918918919
```

Rysunek 21: Wyniki testu modelu

Przebieg iteracji przedstawiony jest na poniższym wykresie:



Rysunek 22: Przebieg nauczania modelu

2.3 Podsumowanie wyników

Jak widać na powyższym wykresie funkcja kosztu przy nauczaniu modelu jest w każdej iteracji coraz mniejsza (minimalizacja funkcji kosztu). A miary takie, jak dokładność, precyzja i średnia harmoniczna zwiększają się wraz z przebiegiem uczenia (model coraz lepiej przewiduje nieznane nowe dane).

Dokładność, która sprawdza stosunek poprawnie przewidywanych wyników do wszystkich prób testujących, może nie być najlepszym wskaźnikiem jakości dla tego zadania (rozpoznawanie chorób serca), ponieważ te wyniki nieodgadnięte mogą należeć do osób chorych, albo model źle wskazał na chore osoby, które są tak naprawdę zdrowe. Lepiej zwrócić uwagę na precyzję oraz recall, które zwracają uwagę w wyliczeniu na klasy odgadnięte poprawnie oraz błędnie. Miara f1 natomiast jest średnią harmoniczną recall'a oraz precyzji. (*Precision, recall i F1 – miary oceny klasyfikatora*, 2019)

Model utworzony w tym zadaniu osiąga wyniki ok. 76% poprawnych przewidzianych wyników pozytywnych (precision), natomiast ilość elementów poprawnie rozpoznanych przez model wynosi 89% (recall). Średnia harmoniczna (f_score) wskazuje, że uśredniony wynik dla naszego modelu to 82% poprawnie rozpoznanych przykładów.

3 Ćwiczenie 3 - Model wielowarstwowy z propagacją wsteczną

Celem ćwiczenia było przygotowanie własnoręcznego wielowarstwowego modelu sieci neuronowej wykorzystującego propagację wsteczną, a później przetestowanie działania wykorzystując dane przygotowane w ćwiczeniu 1. Do zadania wykorzystałem te same biblioteki co w Ćwiczeniu 2.

3.1 Implementacja modelu

```
class Neuron:
    weights: np.ndarray
    bias: float
    _err: np.ndarray

    def init_params(self, std_dev: float, weights_size: float) -> None:
        self.weights = np.random.normal(scale=std_dev, size=weights_size) # initialize
        weights
        self.bias = np.random.normal(scale=std_dev) #initialize bias

    def forward(self, x: np.ndarray) -> np.ndarray:
        return sigmoid(np.dot(x, self.weights) + self.bias)

    def backward(self, x: np.ndarray, err: np.ndarray, is_output: bool, curr_weights:
        np.ndarray = np.array([])) -> np.ndarray:
        if not is_output:
            err = err.T @ curr_weights
        self._err = err * sigmoid_der(np.dot(x, self.weights) + self.bias)
        return self._err

    def update(self, learning_rate: float, x: np.ndarray) -> None:
        self.weights = self.weights - learning_rate * np.dot(x.T, self._err)
        self.bias = self.bias - learning_rate * np.sum(self._err)

class Layer:
    _cache_x: np.ndarray
    neurons: list[Neuron]

    def __init__(self, neurons_amount: int):
        self.neurons = []
        for _ in range(neurons_amount):
            self.neurons.append(Neuron())

    def init_params(self, std_dev: float, weights_size: float) -> None:
        for neuron in self.neurons:
            neuron.init_params(std_dev, weights_size)

    def forward(self, x: np.ndarray) -> np.ndarray:
        self._cache_x = x
        return np.array([neuron.forward(x) for neuron in self.neurons]).T

    def backward(self, err: np.ndarray, is_output: bool, curr_weights: np.ndarray =
        np.array([])) -> np.ndarray:
        return np.array([neuron.backward(self._cache_x, err, is_output,
            curr_weights[neuron_index] if not is_output else None) for neuron_index, neuron
            in enumerate(self.neurons)])

    def get_weights(self) -> np.ndarray:
        return np.array([neuron.weights for neuron in self.neurons]).T

    def update(self, learning_rate: float) -> None:
```

```

        for neuron in self.neurons:
            neuron.update(learning_rate, self._cache_x)

class MultilayerNetwork:
    _layers : list[Layer]
    _batch_size: int

    def __init__(self, hidden_layers_sizes: tuple):
        self._layers = []
        for layer_size in hidden_layers_sizes:
            self._layers.append(Layer(layer_size))

        # output layer
        self._layers.append(Layer(1))

    def __cross_entropy_loss(self, y, y_pred) -> np.ndarray:
        return np.sum(-y * np.log(y_pred) - (1 - y) * np.log(1 - y_pred), axis=1)

    def __cross_entropy_loss_der(self, y, y_pred) -> np.ndarray:
        return np.sum((1 - y) / (1 - y_pred) - y / y_pred, axis=1)

    def forward(self, x: np.ndarray) -> np.ndarray:
        for layer in self._layers:
            x = layer.forward(x)
        return x

    def backward(self, y_train: np.ndarray, y_pred: np.ndarray) -> None:
        err = self._layers[-1].backward(self.__cross_entropy_loss_der(y_train, y_pred),
            True) # calculate error on output layer
        curr_weights = self._layers[-1].get_weights()

        for layer in reversed(self._layers[:-1]):
            err = layer.backward(err, False, curr_weights)
            curr_weights = layer.get_weights()

    def predict(self, x: np.ndarray) -> np.ndarray:
        y_pred = self.forward(x)

        y_pred[y_pred >= 0.5] = 1
        y_pred[y_pred < 0.5] = 0

        return y_pred

    def optimize(self, x_train: np.ndarray, y_train: np.ndarray, x_test: np.ndarray,
        y_test: np.ndarray, batch_size: int, learning_rate: float,
            max_iter: int, std_dev: float, verb = False) -> (list, list, list,
                list, list):
        self._batch_size = batch_size

        # init helper lists
        losses = []
        losses_test = []

        accuracy = []
        precision = []
        f_score = []
        recalls = []

        # init weights and biases
        curr_size = x_train.shape[1]

```

```

for layer in self._layers:
    layer.init_params(std_dev, curr_size)
    curr_size = len(layer.neurons)

#fix shape of y data to match further calculations
if len(y_test.shape) == 1:
    y_test = y_test[np.newaxis].T

if len(y_train.shape) == 1:
    y_train = y_train[np.newaxis].T

# check if batch_size is not larger than size of training data
if batch_size > x_train.shape[0]:
    batch_size = x_train.shape[0]

for index in range(max_iter): #learn for max_iter
    # shuffle training data
    shuffle = np.random.permutation(x_train.shape[0])
    x_train_shuffled = x_train[shuffle]
    y_train_shuffled = y_train[shuffle]

    # for each batch perform learning
    for batch_start_index in range(0, x_train.shape[0], batch_size):
        x_train_batch =
            x_train_shuffled[batch_start_index:batch_start_index+batch_size]
        y_train_batch =
            y_train_shuffled[batch_start_index:batch_start_index+batch_size]

        y_pred = self.forward(x_train_batch)
        self.backward(y_train_batch, y_pred)

        for layer in self._layers:
            layer.update(learning_rate)

        new_train_batch_loss = np.array([])
        new_train_batch_loss = np.append(new_train_batch_loss,
            self.__cross_entropy_loss(y_train_batch, y_pred))

    new_train_loss = np.mean(new_train_batch_loss)

    test_pred = self.forward(x_test)
    new_test_loss = np.mean(self.__cross_entropy_loss(y_test, test_pred))

    #append to helper lists
    losses.append(new_train_loss)
    losses_test.append(new_test_loss)

    # predict test values as 0 or 1
    test_pred = self.predict(x_test)

    #calculate scores for each iteration
    accuracy.append(metrics.accuracy_score(y_test, test_pred))
    precision.append(metrics.precision_score(y_test, test_pred))
    f_score.append(metrics.f1_score(y_test, test_pred))
    recalls.append(metrics.recall_score(y_test, test_pred))

    if verb and index % 100 == 0:
        print("----- Iteration " +
            str(index))
        print("Train loss on " + str(index) + " iteration: ", losses[index])
        print("Test loss on " + str(index) + " iteration: ", losses_test[index])

```



```

        print("Accuracy on " + str(index) + " iteration: ", accuracy[index])
        print("Precision on " + str(index) + " iteration: ", precision[index])
        print("Recall on " + str(index) + " iteration: ", recalls[index])
        print("Fscore on " + str(index) + " iteration: ", f_score[index])
        print("-----\n")

    print("Result of learning process for " + str(max_iter) + " iterations")
    print("-----\n")
    print("Train loss: ", losses[-1])
    print("Test loss: ", losses_test[-1])
    print("-----\n")
    print("Scores")
    print("Accuracy: ", accuracy[-1])
    print("Precision: ", precision[-1])
    print("F_score: ", f_score[-1])
    print("Recall: ", recalls[-1])
    return losses, losses_test, accuracy, f_score, recalls

def plot_learning(self, losses: list, losses_test: list, title: str, axs = None) ->
None:
    if axs == None:
        plt.plot(np.arange(len(losses)), losses, label="Train Loss")
        plt.plot(np.arange(len(losses)), losses_test, label="Test loss")
        plt.plot(np.convolve(losses, np.ones(self._batch_size)/self._batch_size,
            mode='valid'), label="Train loss smooth") #rolling avarage to smooth line
        plt.title(title)
        plt.legend()
        plt.show()
    else:
        axs.plot(np.arange(len(losses)), losses, label="Train Loss")
        axs.plot(np.arange(len(losses)), losses_test, label="Test loss")
        axs.plot(np.convolve(losses, np.ones(self._batch_size)/self._batch_size,
            mode='valid'), label="Train loss smooth") #rolling avarage to smooth line
        axs.set_title(title)
        axs.legend()

```

Rozwiązanie składa się z trzech klas:

- Neuron - klasa reprezentująca pojedynczy neuron w sieci, każdy neuron posiada własne wagi oraz bias, a także zapisuje błąd o jaki ma zmienić wartości wag w trakcie uczenia. Posiada cztery metody:
 - init_params - inicjalizuje wagi oraz bias neuronu w oparciu na parametr ilości wag oraz odchylenia standardowego przy losowaniu wag
 - forward - wywołuje przejście w przód dla danego neuronu
 - backward - wywołuje propagację wstecz dla danego neuronu wyliczając wartość błędu dla danego przejścia w przód dla sieci
 - update - aktualizuje wagi oraz bias neuronu
- Layer - klasa reprezentująca warstwę ukrytą oraz wyjściową sieci. Posiada listę neuronów w owej warstwie oraz zapisuje wartości wejściowe dla tej warstwy. Posiada poza konstruktorem za pomocą którego wiemy ile jest neuronów w danej sieci, także 5 metod:
 - init_params - inicjalizuje dla każdego neuronu w warstwie jego wagi oraz bias'y.
 - forward - zapisuje wartości wejściowe dla tej warstwy, a następnie wykonuje przejście w przód dla neuronów w warstwie
 - backward - dla każdego neuronu w warstwie wykonuje propagację wsteczną
 - get_weights - zwraca wagi dla każdego neuronu

- update - aktualizuje wagi oraz bias dla każdego neuronu warstwy
- MultilayerNetwork - klasa reprezentująca wielowarstwową sieć, posiada warstwy neuronów po których wykonywana jest propagacja wstecz oraz przejście w przód po sieci:
 - Przy utworzeniu obiektu klasy, ustalany jest rozmiar sieci ukrytych oraz ich ilość w postaci krotki np. (2,3) daje nam 2 warstwy ukryte, gdzie pierwsza posiada 2 neurony druga 3
 - W sieci wykorzystywana jest entropia krzyżowa jako funkcja błędu, zdefiniowana została także jej pochodna potrzebna przy propagacji wstecz
 - Metoda forward wywołuje przejście w przód po sieci
 - Metoda backward wywołuje propagację wstecz, wylicza błąd dla wyniku przejścia w przód za pomocą pochodnej funkcji kosztu. Następnie oblicza błąd warstwy wyjściowej (tutaj nie jest potrzebna wiedza o wagach następnej warstwy ponieważ ta warstwa jest ostatnia). Po zapisaniu wag warstwy wyjściowej przechodząc od ostatniej warstwy ukrytej do pierwszej, wyliczane oraz zapisywane zostają błędy neuronów na kolejnych warstwach ukrytych. Błąd obliczany jest wykorzystując błąd wcześniej obliczony, owy błąd przemnażamy przez wagi następnej warstwy oraz przez pochodną funkcji aktywacji, aby cofnąć wcześniej wykonaną aktywację przy przejściu w przód
 - Metoda predict zwraca nam dane predykowane przez sieć
 - Metoda optimize dzieli dane treningowe na paczki o danych rozmiarze (batch_size) (przy każdej iteracji procesu nauczania sieci permutujemy dane treningowe, aby polepszyć wyniki sieci) i przy każdej iteracji, przy każdej paczce wykonujemy przejście w przód po sieci a następnie wykonujemy propagację wstecz dla wyliczonego wyniku przy przejściu w przód. Następnie aktualizujemy wagi i bias'y neuronów sieci i obliczamy oraz zapisujemy koszty, dla dalszej wizualizacji nauczania sieci.
 - Klasa posiada jeszcze metodę do wizualizacji procesu nauczania (plot_learning)

Tak jak dla poprzedniego zadania funkcją aktywacji sieci jest funkcja sigmoidu.

```
def sigmoid(n):
    return 1 / (1 + np.exp(-n))

def sigmoid_der(n):
    return sigmoid(n) * (1 - sigmoid(n))
```

3.2 Badanie modelu na zbiorze heart disease

Dane zostały przygotowane tak samo jak w Ćwiczeniu 1 oraz 2. Jediną różnicą jest przygotowanie zbioru bez standaryzacji danych (#4 n_std_result).

Następnie przygotowałem zmienne reprezentujące różne parametry sieci, które będą badane.

```
#1. Różna wymiarowość warstwy ukrytej
hidden1 = (4, 3)
hidden2 = (20, 10)
hidden3 = (100, 50)

#2. Różne wartości współczynnika uczenia
learning1 = 0.001
learning2 = 0.01

#3. Różne odchylenia standardowych przy inicjalizacji wag
std_dev1 = 0.8
std_dev2 = 0.3

#5. Różne liczby warstw
hidden4 = (20,)
hidden5 = (20, 20)
hidden6 = (20, 20, 20)

max_iter = 1000
verbose = False

options_title = ["hidden_layer " + str(hidden1), "hidden_layer " + str(hidden2), "hidden_layer " + str(hidden3),
                 "learning " + str(learning1), "learning " + str(learning2),
                 "std_dev " + str(std_dev1), "std_dev " + str(std_dev2),
                 "layer size " + str(hidden4), "layer size " + str(hidden5), "layer size " + str(hidden6),
                 "stadarized data", "not standarized data"]

options = [[hidden1, learning1, std_dev2], [hidden2, learning1, std_dev2], [hidden3, learning1, std_dev2],
           [hidden2, learning1, std_dev2], [hidden2, learning2, std_dev2],
           [hidden2, learning1, std_dev1], [hidden2, learning1, std_dev2],
           [hidden4, learning1, std_dev2], [hidden5, learning1, std_dev2], [hidden6, learning1, std_dev2]]
```

Rysunek 23: Przygotowanie zmiennych różnych parametrów

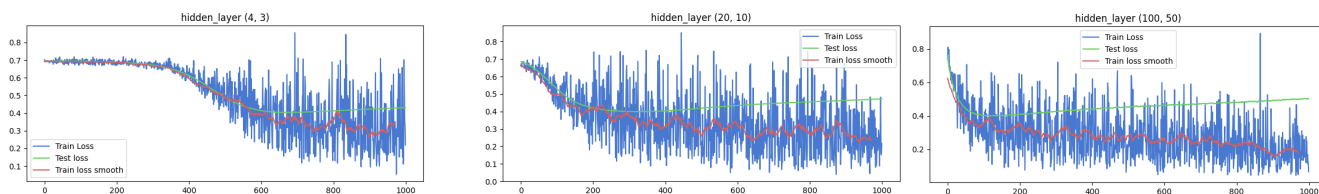
Dla czystego wyświetlania wyników ukrywam ostrzeżenia wyświetlane z powodu zbyt małej precyzji dla pierwszych iteracji nauczania sieci.

```
import warnings
warnings.filterwarnings('ignore') #hide warning for precision (first iterations have no
precision and python gives a warning, to clear output hide it)
```

Dla poniższych testów paczkowanie wynosi 30, a testy przeprowadzone dla 1000 iteracji nauczania. Do zobrazowania kosztu treningu w formie spłaszczonej linii wykorzystuję średnią kroczącą (*Rolling Averages: What They Are and How To Calculate Them*, 2022)

Różne wymiarowości warstwy ukrytej (odchylenie 0.3, współczynnik nauczania 0.001):

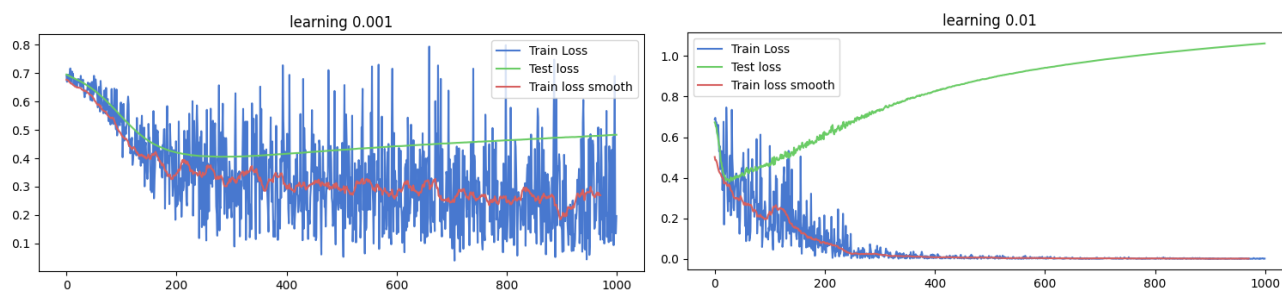
	(4,3)	(20,10)	(100,50)
Dokładność	0.86	0.80	0.83
Precyzja	0.95	0.85	0.94
F score	0.85	0.81	0.83
Recall	0.77	0.77	0.73



Rysunek 24: Różne wymiarowości warstwy ukrytej

Różne wartości współczynnika uczenia ((20,10), odchylenie 0.3):

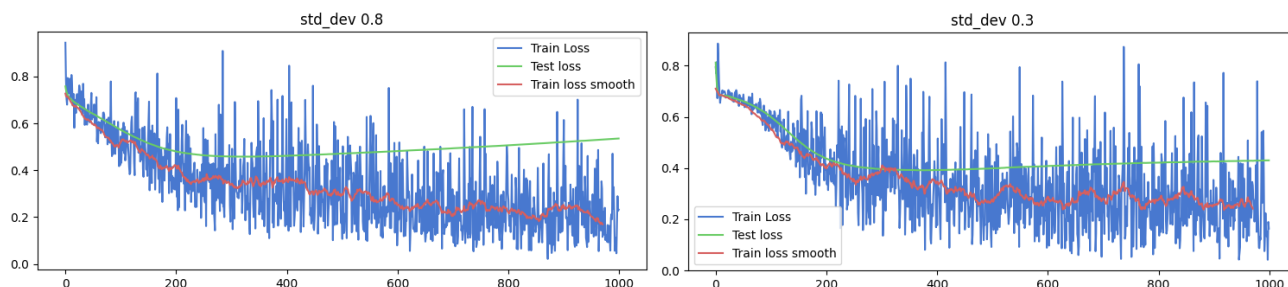
	0.001	0.01
Dokładność	0.82	0.83
Precyzja	0.89	0.92
F score	0.82	0.83
Recall	0.76	0.76



Rysunek 25: Różne wartości współczynnika uczenia

Różne odchylenia standardowych przy inicjalizacji wag ((20,10), współczynnik nauczania 0.001):

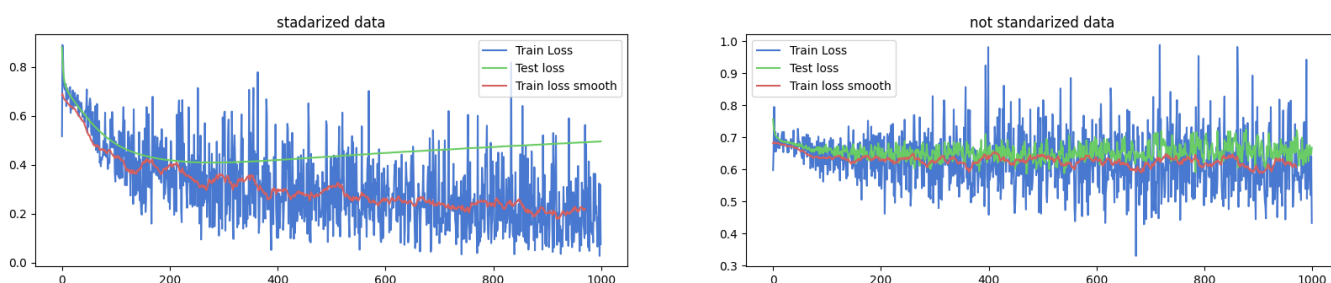
	0.8	0.3
Dokładność	0.78	0.86
Precyzja	0.83	0.95
F score	0.79	0.86
Recall	0.76	0.76



Rysunek 26: Różne odchylenia standardowych przy inicjalizacji wag

Dane znormalizowane i nieznormalizowane ((20,10), współczynnik nauczania 0.001, odchylenie 0.5):

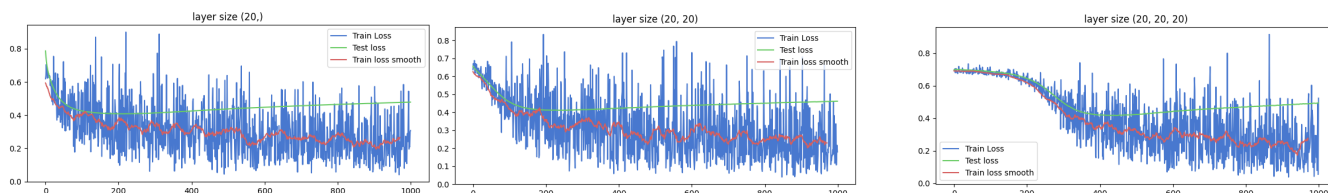
	Standaryzowane	Niestandaryzowane
Dokładność	0.80	0.70
Precyzja	0.91	0.63
F score	0.80	0.71
Recall	0.71	0.82



Rysunek 27: Dane znormalizowane i nieznormalizowane

Różne liczby warstw (współczynnik nauczania 0.001, odchylenie 0.3):

	(20,)	(20,20)	(20,20,20)
Dokładność	0.82	0.83	0.84
Precyzja	0.94	0.90	0.92
F score	0.81	0.83	0.84
Recall	0.71	0.78	0.78



Rysunek 28: Różne liczby warstw

3.3 Podsumowanie wyników

- Wymiary warstwy ukrytej mogą wpłynąć na wyniki sieci, w trakcie badania sieci widać, że gdy zwiększaliśmy ilości neuronów w warstwach model szybciej się uczył, ale łatwiej pojawiał się problem przeuczenia, dlatego też precyzja była gorsza dla sieci ze zbyt Dużą ilością neuronów niż przy tej ilości mniejszej
- Podobnie jak wymiary warstw to i współczynnik nauczania większy przyspiesza nam proces uczenia, ale tutaj także widzimy nagły wzrost kosztu testowego, co wskazuje, że model nie daje rady z uogólnieniem swojej wiedzy
- Gdy odchylenie standardowe przy inicjalizacji wag jest zmniejszane, spowalniamy proces uczenia, co może pomóc w uniknięciu przeuczenia, ale zbyt małe odchylenie może spowodować proces niedouczenia sieci
- Gdy dane są nieznormalizowane sieć nie daje rady uczyć się, widać to także po wynikach miar, gdzie dla danych niestandardyzowanych mamy je najslabsze
- Zwiększenie ilości warstw pomogło ustabilizować koszt treningu, co przejawia się także w lepszych wynikach miar

Z powyższych testów najlepsze wyniki przejawia sieć (20, 10), współczynnik nauczania 0.001, odchylenie standardowe 0.3, gdzie precyzja wynosi 95%. Najgorsze zaś posiadają dane niestandardyzowane, które nie były w stanie się uczyć i wartość precyzji zatrzymała się na poziomie 63%.

4 Ćwiczenie 4 - Model wielowarstwowy przy użyciu narzędzi

Celem ćwiczenia było odtworzenie architektury sieci zaimplementowanej w poprzednim ćwiczeniu wykorzystując jedno z dostępnych narzędzi do budowania sieci neuronowych. W tym ćwiczeniu wykorzystujemy bibliotekę tensorflow w odrębnie keras. Reszta bibliotek jak w poprzednich ćwiczeniach.

4.1 Odtworzenie architektury sieci

Do odtworzenia architektury sieci z poprzedniego ćwiczenia tworzę klasę dziedziczącą po klasie `tf.keras.Model`. Klasa w konstruktorze przyjmuje wymiary warstw ukrytych w formie krotki, wielkość warstwy wejściowej oraz wyjściowej. Każdą warstwę ukrytą oraz wyjściową przechowuje w `_layers`, której elementy to `tf.keras.layers.Dense` (oblicza wyjście wykorzystując wzór $\text{output} = \text{activation}(\text{dot}(\text{input}, \text{kernel}) + \text{bias})$, identycznie jak robiliśmy to w poprzednim ćwiczeniu). Funkcja aktywacji w warstwach to `sigmoid`. Klasa posiada także metodę `call`, która wykonuje przejście w przód po sieci.

```
class MultilayerNetwork(tf.keras.Model):
    def __init__(self, hidden_layers_sizes: tuple, input_size: int, output_size: int):
        super(MultilayerNetwork, self).__init__()

        self._layers = []
        curr_size = input_size
        for hidden_layer_size in hidden_layers_sizes:
            layer = tf.keras.layers.Dense(hidden_layer_size, activation="sigmoid",
                                           input_shape=(curr_size,))
            self._layers.append(layer)
            curr_size = hidden_layer_size
        output_layer = tf.keras.layers.Dense(output_size, activation="sigmoid",
                                              input_shape=(curr_size,))
        self._layers.append(output_layer)

    def call(self, inputs):
        x = inputs
        for layer in self._layers:
            x = layer(x)
        return x
```

Po utworzeniu klasy architektury sieci, należy zdefiniować funkcje procesu uczenia sieci.

```
def train_model(model, x_train: np.ndarray, y_train: np.ndarray, x_test: np.ndarray,
               y_test: np.ndarray, batch_size: int,
               optimizer, loss, max_iter, verb=False):

    losses = []
    losses_test = []

    accuracy = []
    precision = []
    f_score = []
    recalls = []

    #fix shape of y data to match further calculations
    if len(y_test.shape) == 1:
        y_test = tf.transpose(y_test)

    if len(y_train.shape) == 1:
        y_train = tf.transpose(y_train)
```

```

if batch_size > x_train.shape[0]:
    batch_size = x_train.shape[0]

for index in range(max_iter): #learn for max_iter
    # shuffle training data
    shuffle = np.random.permutation(x_train.shape[0])
    x_train_shuffled = tf.gather(x_train, shuffle)
    y_train_shuffled = tf.gather(y_train, shuffle)

    # for each batch perform learning
    for batch_start_index in range(0, x_train.shape[0], batch_size):
        x_train_batch = x_train_shuffled[batch_start_index:batch_start_index+batch_size]
        y_train_batch = y_train_shuffled[batch_start_index:batch_start_index+batch_size]

        with tf.GradientTape() as tape:
            y_pred = model(x_train_batch)
            y_pred = tf.squeeze(y_pred)
            loss_value = loss(y_train_batch, y_pred)
            loss_value = tf.reduce_mean(loss_value)

            gradient = tape.gradient(loss_value, model.trainable_variables)
            optimizer.apply_gradients(zip(gradient, model.trainable_variables))

        losses.append(loss_value.numpy())
        test_pred = model(x_test)
        test_pred = tf.squeeze(test_pred)
        loss_test_value = loss(y_test, test_pred)
        loss_test_value = tf.reduce_mean(loss_test_value)
        losses_test.append(loss_test_value.numpy())

        test_pred = tf.where(test_pred >= 0.5, 1.0, test_pred)
        test_pred = tf.where(test_pred < 0.5, 0.0, test_pred)

    #calculate scores for each iteration
    accuracy.append(metrics.accuracy_score(y_test, test_pred))
    precision.append(metrics.precision_score(y_test, test_pred))
    f_score.append(metrics.f1_score(y_test, test_pred))
    recalls.append(metrics.recall_score(y_test, test_pred))

if verb and index % 100 == 0:
    print("----- Iteration " + str(index))
    print("Train loss on " + str(index) + " iteration: ", losses[index])
    print("Test loss on " + str(index) + " iteration: ", losses_test[index])
    print("Accuracy on " + str(index) + " iteration: ", accuracy[index])
    print("Precision on " + str(index) + " iteration: ", precision[index])
    print("Recall on " + str(index) + " iteration: ", recalls[index])
    print("Fscore on " + str(index) + " iteration: ", f_score[index])
    print("-----\n")

print("Result of learning process for " + str(max_iter) + " iterations")
print("-----\n")
print("Train loss: ", losses[-1])
print("Test loss: ", losses_test[-1])
print("-----\n")
print("Scores")
print("Accuracy: ", accuracy[-1])
print("Precision: ", precision[-1])
print("F_score: ", f_score[-1])
print("Recall: ", recalls[-1])
return losses, losses_test, accuracy, f_score, recalls

```


Dla każdej iteracji tworzone są paczki danych treningowych (wcześniej permutowanych dla lepszych wyników). Dla każdej paczki wykonujemy przejście w przód po naszej sieci z użyciem `tf.GradientTape()`, a następnie po obliczeniu błędu wyniku względem danych treningowych, przeliczamy gradienty parametrów uczących się w naszej sieci (wagi, bias), a następnie aktualizujemy parametry neuronów za pomocą owych gradientów z użyciem optimizera. Reszta funkcji jest podobna do rozwiązania optimize w poprzednim zadaniu.

Przygotowana została także funkcja pomocnicza do wizualizacji wyników.

```
def plot_learning(losses: list, losses_test: list, title: str, batch_size: int, axs = None):
    if axs == None:
        plt.plot(np.arange(len(losses)), losses, label="Train Loss")
        plt.plot(np.arange(len(losses)), losses_test, label="Test loss")
        plt.plot(np.convolve(losses, np.ones(batch_size)/batch_size, mode='valid'),
                 label="Train loss smooth") #rolling avarage to smooth line
        plt.title(title)
        plt.legend()
        plt.show()
    else:
        axs.plot(np.arange(len(losses)), losses, label="Train Loss")
        axs.plot(np.arange(len(losses)), losses_test, label="Test loss")
        axs.plot(np.convolve(losses, np.ones(batch_size)/batch_size, mode='valid'),
                 label="Train loss smooth") #rolling avarage to smooth line
        axs.set_title(title)
        axs.legend()
```

4.2 Badanie modelu na zbiorze heart disease

Dane zostały przygotowane identycznie, jak w poprzednich ćwiczeniach. Jako, że tensorflow działa na tensorach, dane zamieniam na tensor.

```
x_train = tf.convert_to_tensor(x_train, dtype=tf.float32)
y_train = tf.convert_to_tensor(y_train, dtype=tf.float32)

y_test = tf.convert_to_tensor(y_test, dtype=tf.float32)
x_test = tf.convert_to_tensor(x_test, dtype=tf.float32)
```

A także, jak w poprzednim zadaniu ukrywam ostrzeżenia wynikające z precyzji.

```
import warnings
warnings.filterwarnings('ignore') #hide warning for precision (first iterations have no
precision and python gives a warning, to clear output hide it)
```

Sieć będzie badana przez 1000 iteracji, funkcja kosztu taka sama, jak w poprzednim zadaniu (entropia krzyżowa). Wymiary warstw ukrytych sieci (10, 10, 10), a warstwa wyjściowa składa się z 1 neuronu.

```
#2. Rozmiar batcha
batch_size_small = 10
batch_size_middle = 30
batch_size_large = 120

#3. Wartość współczynnika uczenia dla różnych optimizerów
learning_rate_1 = 0.0001
learning_rate_2 = 0.01

optimizer_SGD_lr_2 = tf.keras.optimizers.legacy.SGD(learning_rate_2)
optimizer_Adam_lr_2 = tf.keras.optimizers.legacy.Adam(learning_rate_2)
optimizer_Nadam_lr_2 = tf.keras.optimizers.legacy.Nadam(learning_rate_2)

#1. Wybrany optimizer (SGD i dwa inne)
optimizer_SGD = tf.keras.optimizers.legacy.SGD(learning_rate_1)
optimizer_Adam = tf.keras.optimizers.legacy.Adam(learning_rate_1)
optimizer_Nadam = tf.keras.optimizers.legacy.Nadam(learning_rate_1)

max_iter = 1000
loss = tf.keras.losses.binary_crossentropy
verbose = False

options_title = ["Optimizer SGD lr: " + str(learning_rate_1) + " batch: " + str(batch_size_middle),
                 "Optimizer Adam lr: " + str(learning_rate_1) + " batch: " + str(batch_size_middle),
                 "Optimizer Nadam lr: " + str(learning_rate_1) + " batch: " + str(batch_size_middle),
                 "Optimizer Adam lr: " + str(learning_rate_1) + " batch: " + str(batch_size_small),
                 "Optimizer Adam lr: " + str(learning_rate_1) + " batch: " + str(batch_size_large),
                 "Optimizer SGD lr: " + str(learning_rate_2) + " batch: " + str(batch_size_middle),
                 "Optimizer Adam lr: " + str(learning_rate_2) + " batch: " + str(batch_size_middle),
                 "Optimizer Nadam lr: " + str(learning_rate_2) + " batch: " + str(batch_size_middle)]

options = [(optimizer_SGD, batch_size_middle), (optimizer_Adam, batch_size_middle), (optimizer_Nadam, batch_size_middle),
           (optimizer_Adam, batch_size_small), (optimizer_Adam, batch_size_large),
           (optimizer_SGD_lr_2, batch_size_middle), (optimizer_Adam_lr_2, batch_size_middle), (optimizer_Nadam_lr_2, batch_size_middle)]
```

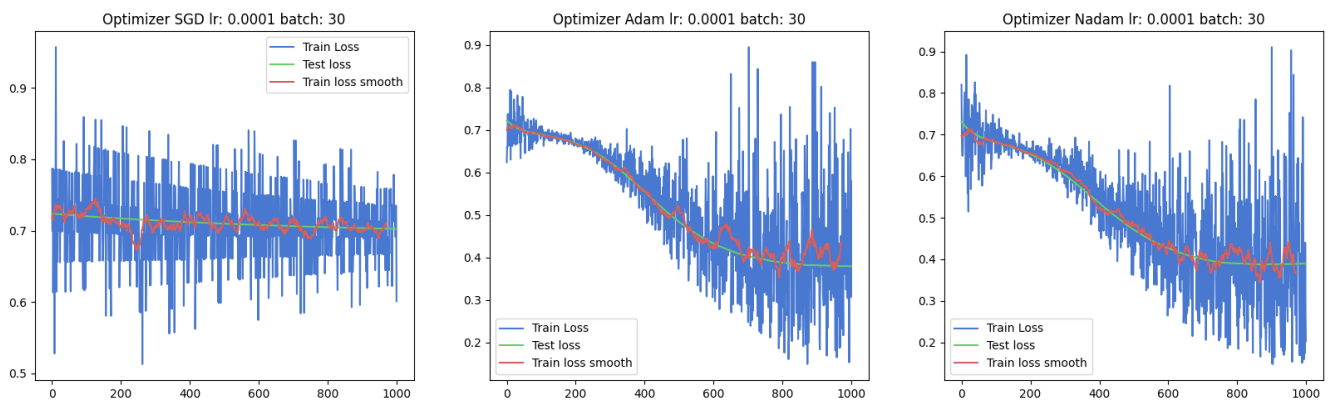
Rysunek 29: Przygotowanie zmiennych różnych parametrów

Badane będą 3 optimizer'y:

- SGD (Stochastic Gradient Descent) - SGD jest podstawowym algorytmem optymalizacji stosowanym w uczeniu maszynowym. Działa na zasadzie aktualizacji wag sieci poprzez iteracyjne minimalizowanie funkcji kosztu, poruszając się w kierunku przeciwnym do gradientu funkcji kosztu. Każda aktualizacja wag następuje na podstawie jednego przykładu treningowego (stochastycznie), co może powodować chaotyczne skoki w przestrzeni wag.
- Adam (Adaptive Moment Estimation) - Adam łączy zalety algorytmów SGD z adaptacyjnym podejściem do aktualizacji wag. Wykorzystuje ruchome średnie gradientów oraz ich kwadratów, dostosowując tempo uczenia dla każdego wagowego parametru. Adaptacyjność pozwala na dostosowanie się do różnych gradientów w różnych warstwach i w różnych okresach uczenia. (*Adam: A Method for Stochastic Optimization*, 2014)
- Nadam (Nesterov-accelerated Adaptive Moment Estimation) - Nadam jest wariantem algorytmu Adam, który łączy cechy algorytmu Nesterova z Adamem. Korzysta z techniki akceleracji Nesterova, która dodaje poprawki do aktualizacji wag, aby lepiej uwzględniać przyszłe położenie wag. (*Nesterov momentum*, 2019)

Poniższe wyniki są dla różnych optimizerów, gdzie współczynnik uczenia wynosi 0.0001, a rozmiar batch'a 30.

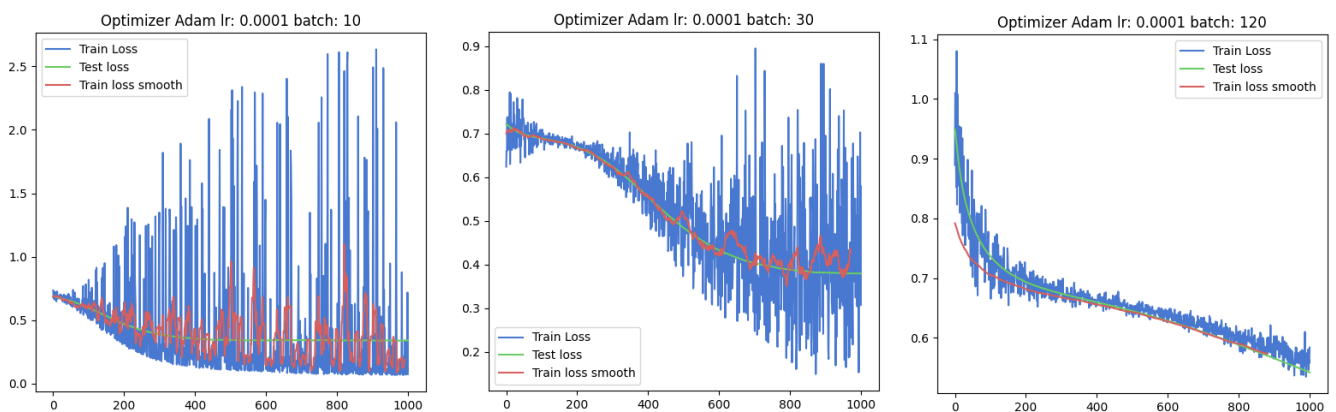
	SGD	Adam	Nadam
Dokładność	0.50	0.86	0.83
Precyzja	0.50	0.81	0.78
F score	0.66	0.86	0.84
Recall	1.0	0.93	0.93



Rysunek 30: Różne optimizer'y

Różne rozmiary batch'y (współczynnik uczenia wynosi 0.0001, optimizer Adam):

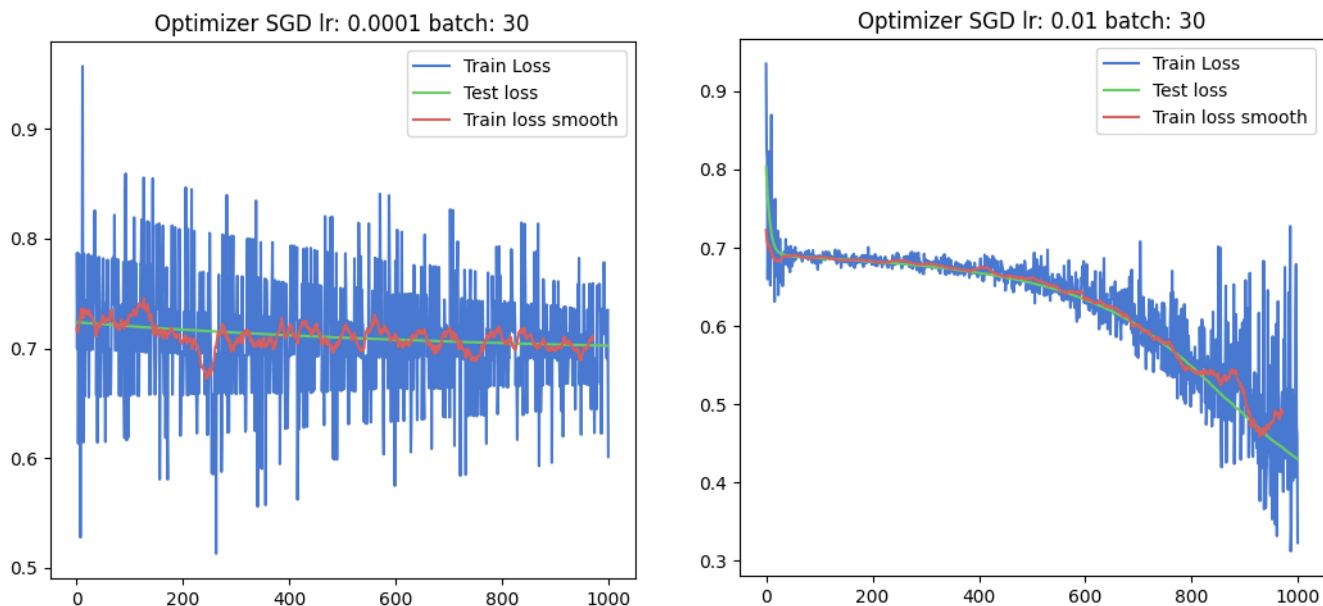
	10	30	120
Dokładność	0.87	0.86	0.83
Precyzja	0.83	0.81	0.76
F score	0.87	0.86	0.84
Recall	0.93	0.93	0.95



Rysunek 31: Różne rozmiary batch'y

Różne wartości współczynnika uczenia (batch 30):
SGD:

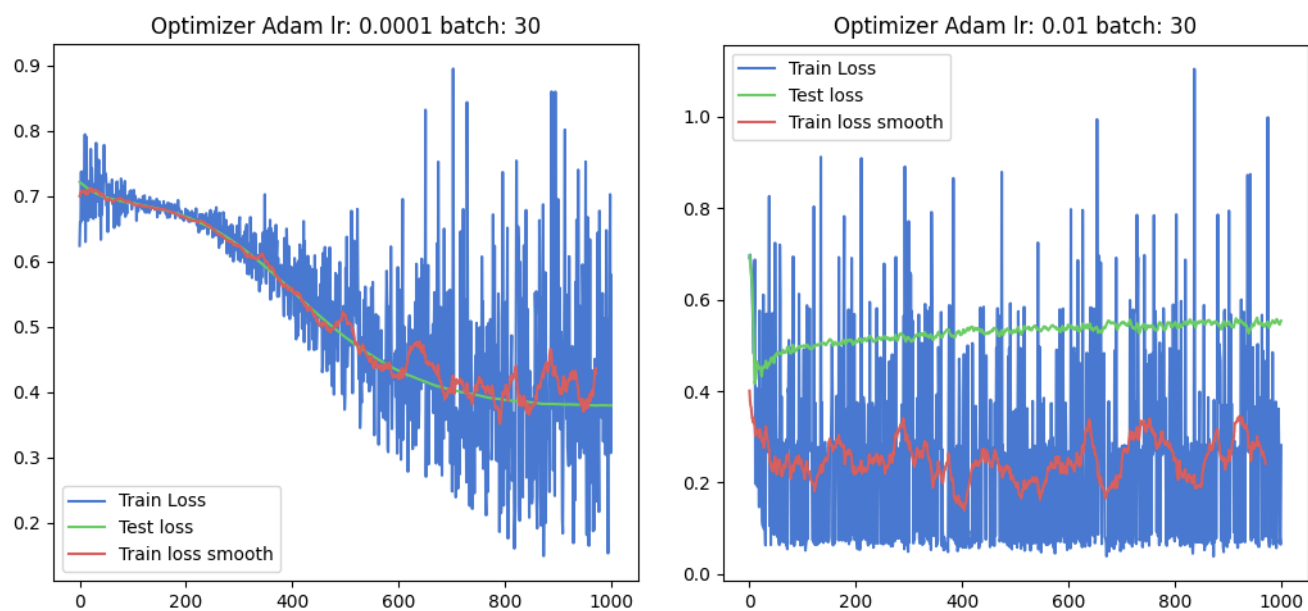
	0.0001	0.01
Dokładność	0.50	0.84
Precyzja	0.50	0.79
F score	0.66	0.86
Recall	1.0	0.93



Rysunek 32: Różne wartości współczynnika uczenia SGD

Adam:

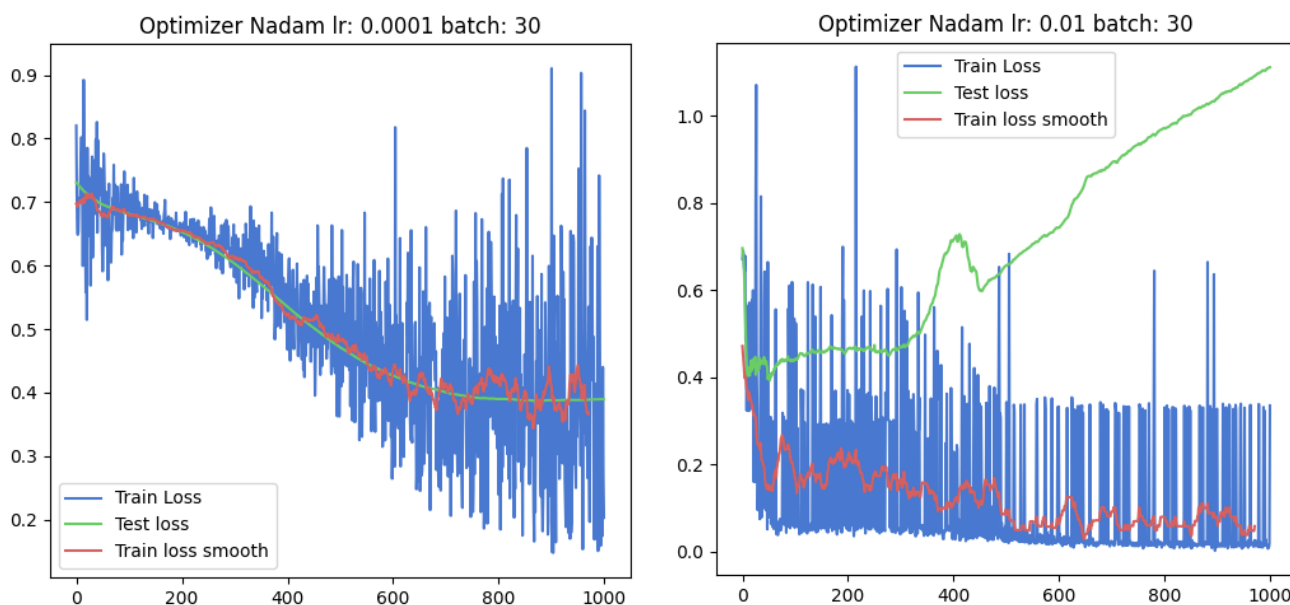
	0.0001	0.01
Dokładność	0.86	0.80
Precyzja	0.81	0.74
F score	0.86	0.81
Recall	0.93	0.90



Rysunek 33: Różne wartości współczynnika uczenia Adam

Nadam:

	0.0001	0.01
Dokładność	0.83	0.77
Precyzja	0.78	0.76
F score	0.84	0.77
Recall	0.93	0.78



Rysunek 34: Różne wartości współczynnika uczenia Nadam

4.3 Podsumowanie wyników

- Dla współczynnika nauczania 0.0001 optimizer SGD nie jest w stanie się uczyć, jest to spowodowane zbyt małym współczynnikiem. Adam oraz Nadam przejawiają podobne wyniki z podobnymi miarami oceny na poziomie 80% precyzji.
- Zwiększenie batch'a ustabilizowało w przypadku optimizer'a Adam proces nauczania, wyniki ocen pogarszały się, ale Recall natomiast wzrastał, wskazuje to na większą ilość pozytywnie predykowanych wyników, z wszystkich możliwych pozytywnych wyników.
- Gdy współczynnik nauczania wynosił 0.0001 SGD nie był w stanie się uczyć, natomiast zwiększenie współczynnika do 0.01 pozwoliło, aby model się nauczył z precyzją na poziomie 80%
- Zwiększenie tego współczynnika w przypadku Adam'a oraz Nadam'a pogorszył ich wyniki, a zarazem doprowadził do gorszego uogólnienia predykcji (przeuczenia modelu)
- Proces nauczania przy większym współczynniku dla Adam'a i Nadam'a spowodował także mniejszą stabilność nauki

Modele z tego zadania posiadają lepsze wartości miary Recall w porównaniu do poprzedniego zadania, oznacza to, że model posiada większą ilość pozytywnie predykowanych wyników, z wszystkich możliwych pozytywnych wyników. Najlepsze wyniki uzyskał optimizer Adam dla współczynnika nauczania 0.0001 oraz rozmiaru batch'a 30 (Dokładność 86%, Precyzja 81%, F score 86%, Recall 93%). Precyzja dla tego modelu jest mniejsza niż dla modelu z poprzedniego zadania, może to wynikać z małego współczynnika nauczania i model możliwe, że potrzebuje więcej iteracji, aby zwiększyć swoją precyzję (przy czym nie przeuczając się).

Literatura

- Adam: A Method for Stochastic Optimization.* (2014, grudzień). Retrieved from <https://arxiv.org/abs/1412.6980>
- Heart Disease Dataset.* (1988, czerwiec). Retrieved from <https://archive.ics.uci.edu/dataset/45/heart+disease>
- K najbliższych sąsiadów.* (2022, czerwiec). Retrieved from https://pl.wikipedia.org/wiki/K_najbli%C5%BCszych_s%C4%85siad%C3%B3w
- Nesterov momentum.* (2019, czerwiec). Retrieved from https://golden.com/wiki/Nesterov_momentum-YX9WPE5
- Precision, recall i F1 – miary oceny klasyfikatora.* (2019, listopad). Retrieved from <https://ksopyla.com/data-science/precision-recall-f1-miary-oceny-klasyfikatora/>
- Rolling Averages: What They Are and How To Calculate Them.* (2022, czerwiec). Retrieved from <https://www.indeed.com/career-advice/career-development/what-is-rolling-average>