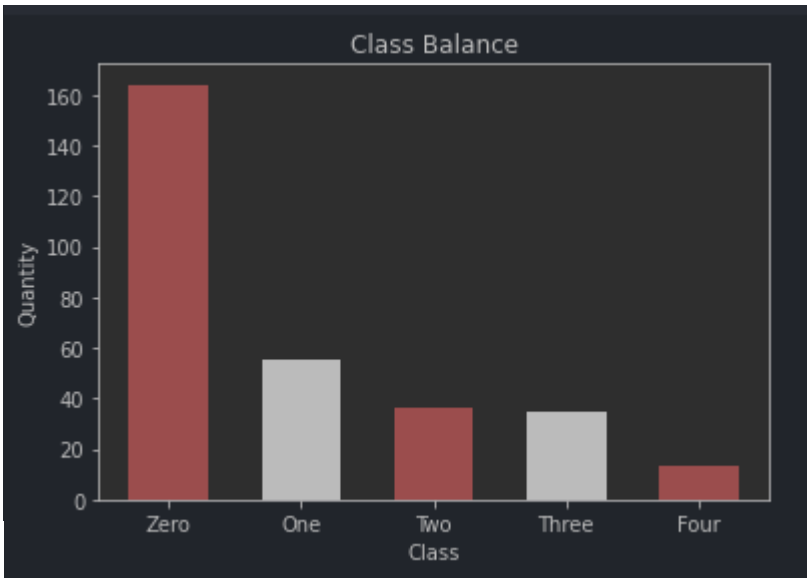
Raport 1

Krzysztof Maciejewski 260449

Analiza eksploracyjna

Czy zbiór jest zbalansowany pod względem liczby próbek na klasy?



1. Wykres zależności liczby próbek od klasy

Po przeprowadzeniu analizy liczby próbek należących do danej klasy, zauważyłem że najwięcej próbek przynależy do klasy 0, która mówi o jej braku. Pozostałe cztery klasy mają podobnie zbalansowane wartości.

Jakie są średnie i odchylenia cech liczbowych?

age	54.438944
trestbps	131.689769
chol	246.693069
thalach	149.607261
oldpeak	1.039604
ca	0.672241

Tabela przedstawia średnie wartości cech liczbowych

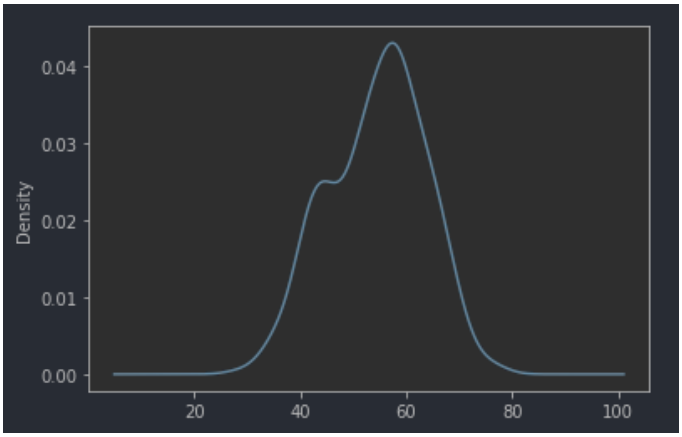
Średnie wartości cech liczbowych znacznie się od siebie różnią co pozwala stwierdzić, że w przyszłości trzeba będzie je znormalizować aby umożliwić wzajemne porównywanie i dalszą analizę.

age	9.038662
trestbps	17.599748
chol	51.776918
thalach	22.875003
oldpeak	1.161075
ca	0.937438

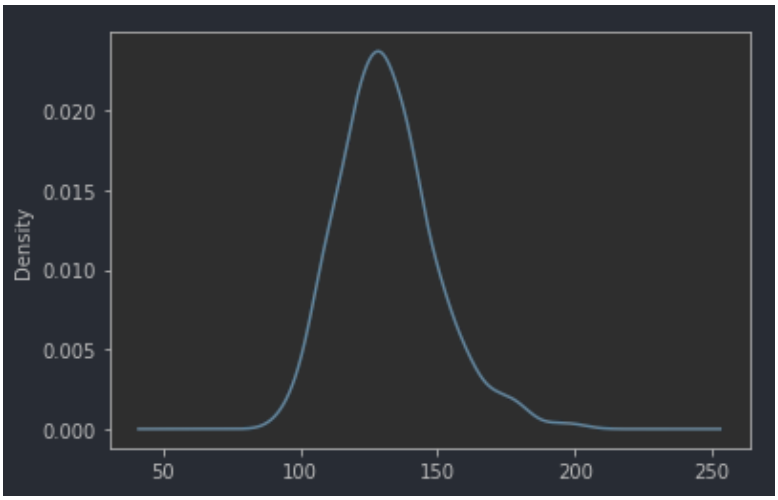
Tabela przedstawia odchylenia cech liczbowych

Wartości pozwalają stwierdzić, że takie dane jak np. wskaźnik cholesterolu mają bardzo zróżnicowane wartości i są mocno rozproszone.

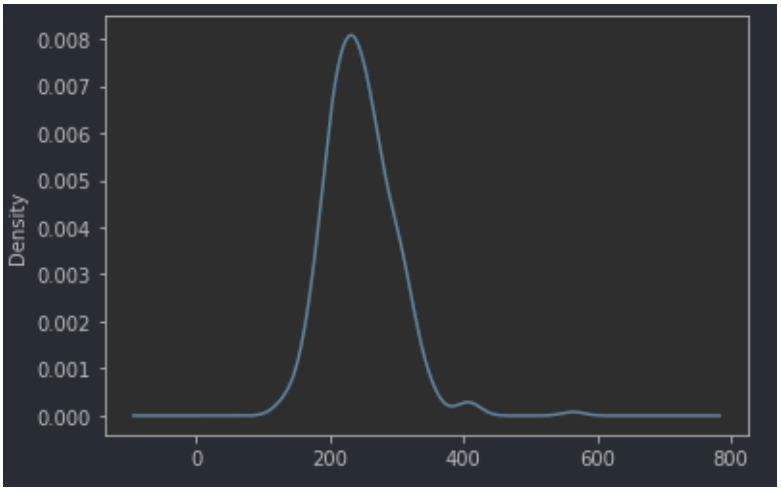
Dla cech liczbowych: czy ich rozkład jest w przybliżeniu normalny?



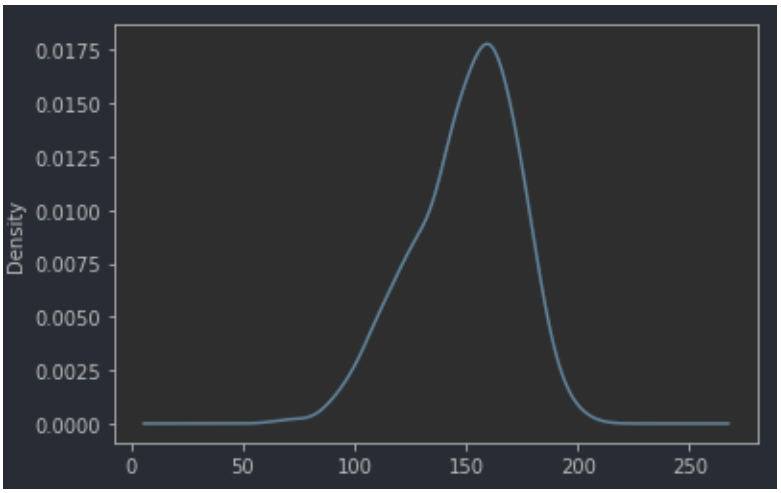
Rozkład wieku



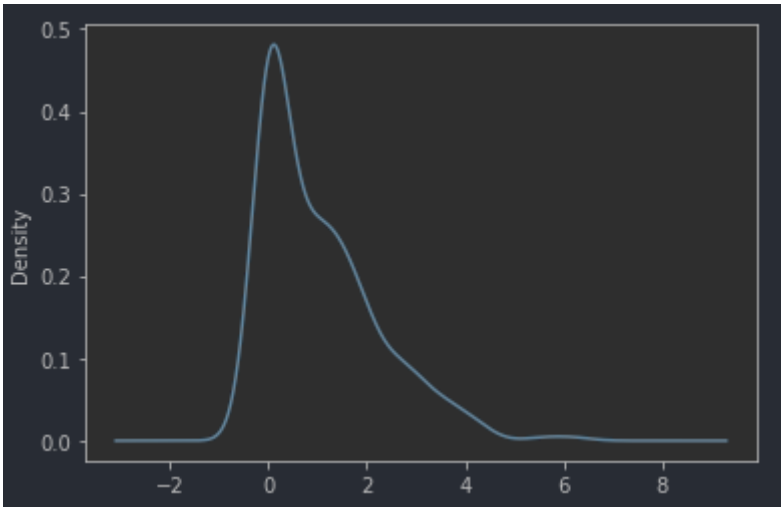
Rozkład resting blood pressure



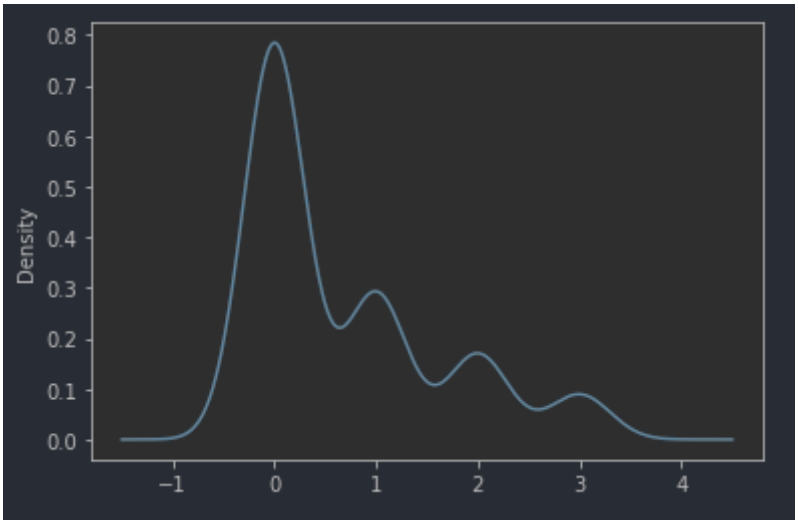
Rozkład cholesterolu



Rozkład maximum heart rate



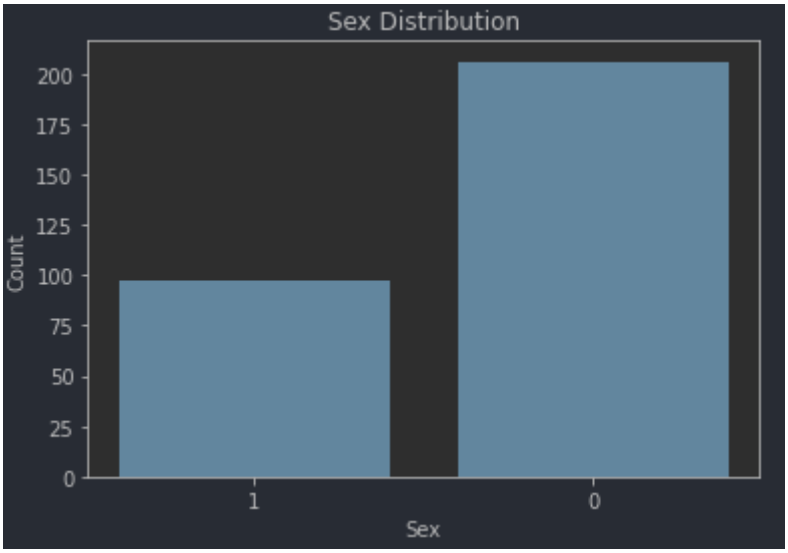
Rozkład depression induced by exercise



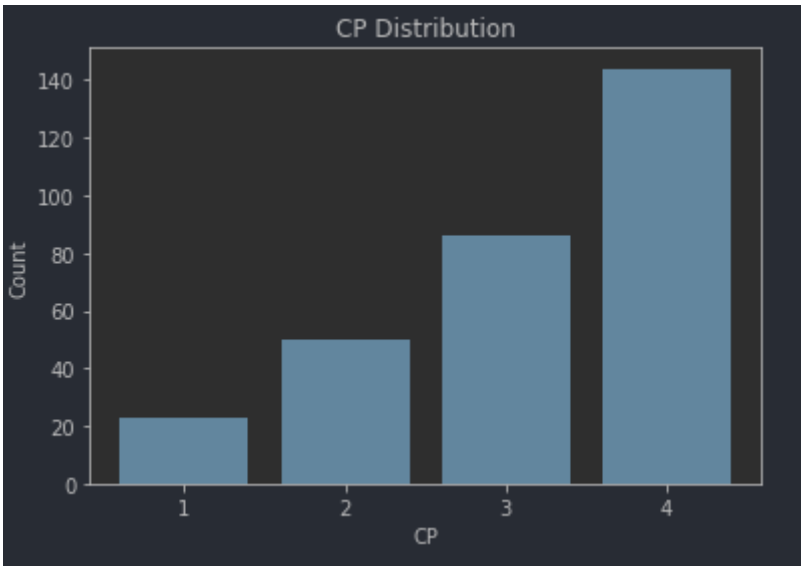
Rozkład cechy ca

Większość rozkładów dla cech liczbowych jest w przybliżeniu normalnych. Najbardziej od kształtu „dzwona” odbiegają dwa ostatnie wykresy. Wartości koncentrują się w nich wokół 0.

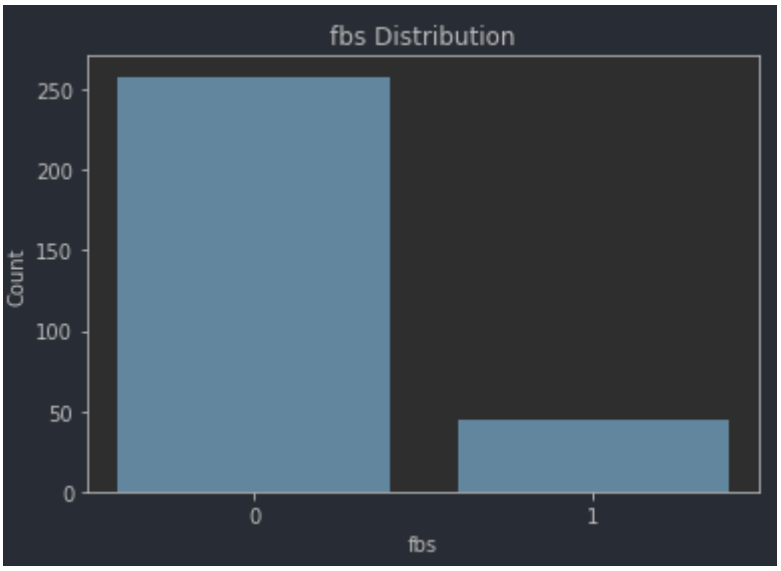
Dla cech kategorycznych: czy rozkład jest w przybliżeniu równomierny?



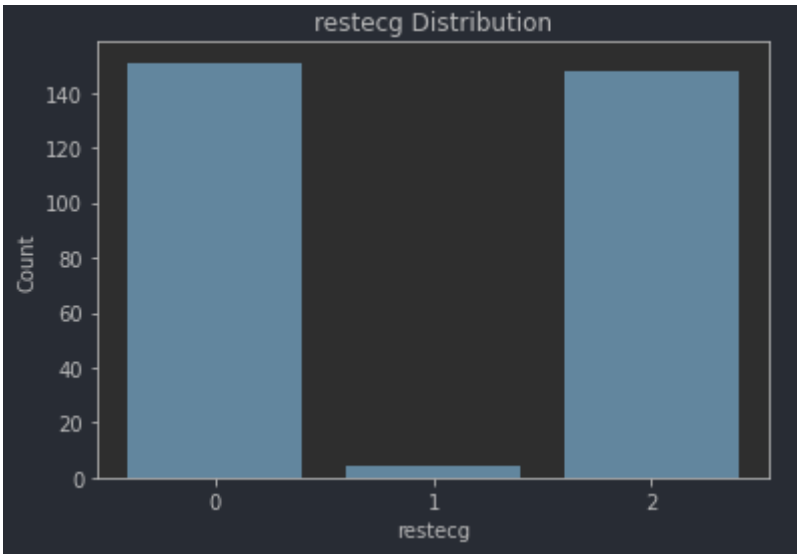
Rozkład cechy kategorycznej płeć



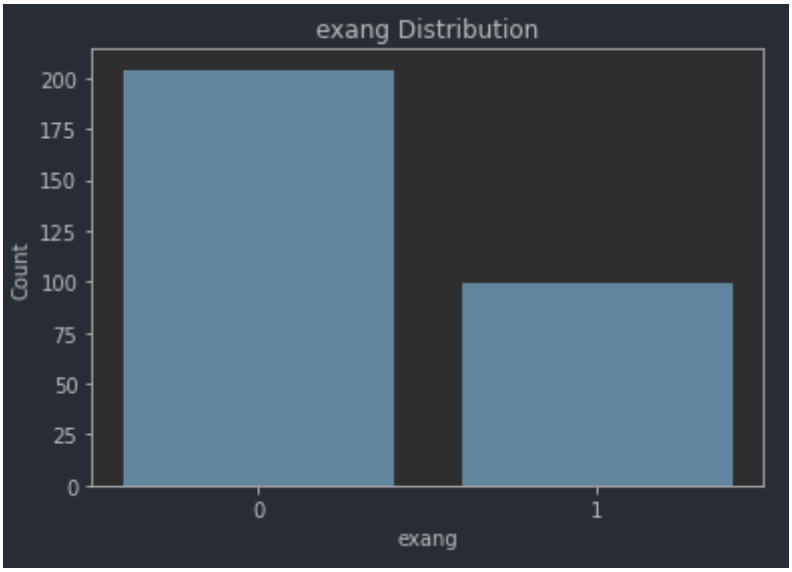
Rozkład cechy kategorycznej CP



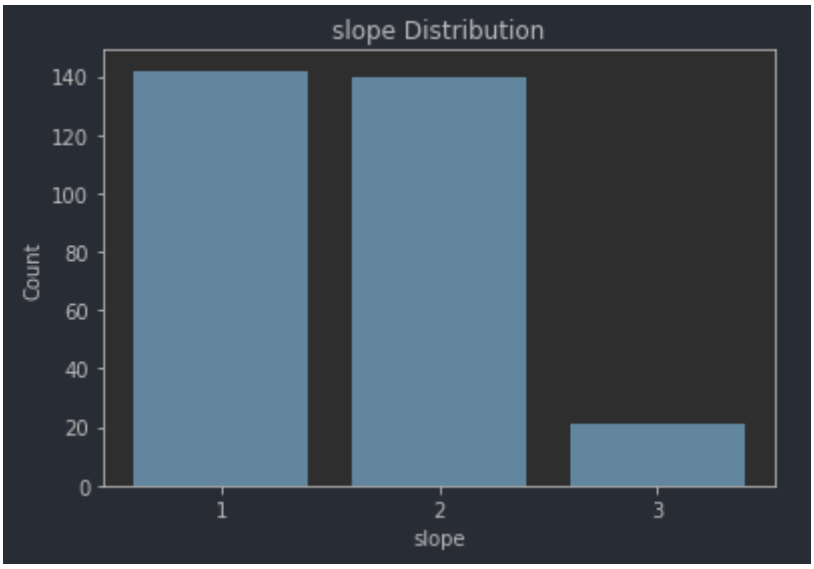
Rozkład cechy kategorycznej fbs



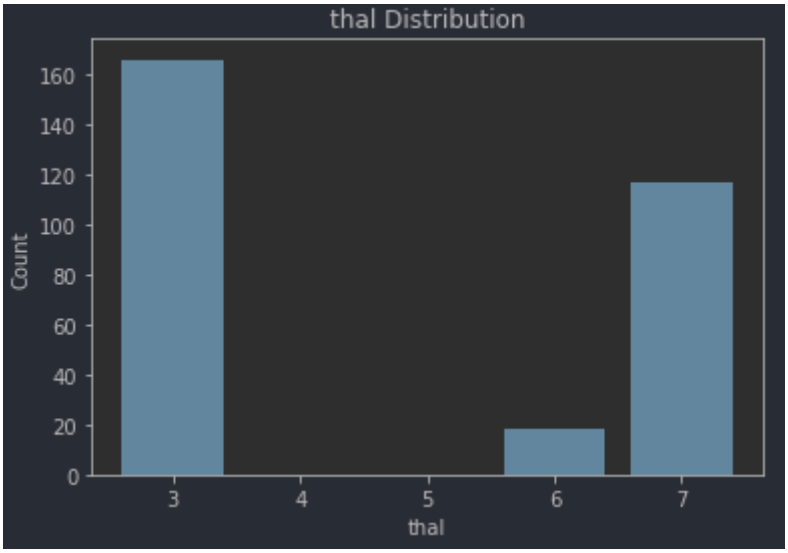
Rozkład cechy kategorycznej restecg



Rozkład cechy kategorycznej exang



Rozkład cechy slope



Rozkład cechy thal

Jak pokazują przedstawione rozkłady cech kategorycznych, żaden z nich nie jest rozkładem jednostajnym. Wartości tych cech występują nierównomiernie.

Czy występują cechy brakujące i jaką strategię możemy zastosować żeby je zastąpić?

W danych następujące cechy mają brakujące wartości: ca (cecha liczbowa) oraz thal (cecha kategoryczna). Cecha ca to liczba głównych naczyń zabarwionych za pomocą fluoroskopii. Po sprawdzeniu ile wartości jest brakujących, okazało się że dla ca są to 4 wartości, a dla thal 2 wartości. Stosunkowo jest to ilość, która nie jest duża więc nie warto byłoby w tych przypadkach odrzucać całą kolumnę. Ponieważ liczba wierszy z brakującymi wartościami jest mała, można by usunąć te wiersze.

Dla cechy liczbowej w przypadku rozkładu normalnego skuteczną strategią pozwalającą na nieodrzucanie wierszy byłoby wstawienie w miejsce brakującej wartości średniej z danej kolumny. Jednakże jak wynika z wykresu rozkładu cechy ca jej rozkład nie przypomina rozkładu normalnego,

dlatego w tym przypadku lepsze okazałoby się wstawienie mediany która jest mniej wrażliwa na wartości odstające.

Inną strategią było by wypełnienie brakujących kolumn w wierszach losowo wybraną wartością spośród wierszy posiadających wszystkie wartości. Technika ta jest odpowiednia zarówno dla cech liczbowych jak i kategorycznych.

Ostatnią strategią jest technika zwana Multiple Imputation, która pozwala na wyznaczenie brakujących wartości biorąc pod uwagę pozostałe wartości kolumn. Na początku zastępujemy brakujące wartości prostą strategią jak np. średnią danej kolumny. Brakujące wartości są wyznaczane na podstawie modelu regresji, w którym brakująca zmienna jest zmienną zależną, a pozostałe zmienne są zmiennymi niezależnymi. Następnie kolejna brakująca wartość jest używana jako zmienna zależna, a pozostałe jako niezależne. Proces trwa dopóki wszystkie brakujące wartości nie zostaną uwzględnione jako zmienne zależne. Po wyznaczeniu ich wartości początkowe tymczasowe wartości wyznaczone za pomocą prostej strategii są zastępowane przewidywaniami z modelu regresji. Proces zastępowania jest wykonywany kilka razy i wartości są aktualizowane po każdym z nich, aż do momentu gdy najlepiej odzwierciedlają relacje zidentyfikowane w danych.

Raport 2

Krzysztof Maciejewski 260449

Wstęp

Przy pomocy modelu regresji logistycznej wykorzystującej entropię krzyżową jako funkcję straty będę klasyfikował dane binarnie. Na początku pobieram dane i wypełniam brakujące wartości kolumn modą i medianą w zależności od tego czy jest to cecha kategoryczna czy liczbowa. Ponieważ będę klasyfikował dane binarnie zamieniłem klasy 1-4 na jedną wspólną klasę 1 (chory).

```
for index, row in Y.iterrows():
    if row[0] != 0:
        row[0] = 1 #jeżeli nie ma klasy 0 to ma klasę 1

y = Y.to_numpy()
X = (X-X.min())/(X.max()-X.min())
x = X.to_numpy()
```

Potem znormalizowałem dane i zamieniłem na np array.

Kolejno podzieliłem dane na zbiory treningowe i testowe oraz losowo zainicjowałem tablicę wag i bias.

```
x_train, x_test, y_train, y_test = train_test_split(x,y, test_size= 0.2)

number_of_features = 13
#losowa inicjalizacja wag
weights = np.random.uniform(0.0, 100.0, size=number_of_features)
bias= np.random.rand(1)
```

Implementacja matematycznych funkcji

Następnie stworzyłem funkcję służącą jako sigmoid do obliczania wartości wyjściowych neuronów. Aby lepiej dostosować funkcję sigmoidalną (rozciągnąć ją) dodałem parametr, sprawiło to, że wartości neuronów stały się bardziej zróżnicowane.

```
def sigmoid(z):
    return 1 / (1 + np.exp(-z/200))
    # return 1 / (1 + np.exp(-(z-200)))
```

Następnie zdefiniowałem funkcje odpowiedzialne za wyliczanie funkcji straty i aktualizacje wag. Do wzoru entropii krzyżowej dodałem małą wartość która, zapobiega zwracaniu przez funkcję logarytmiczną -inf. W funkcji update_weights wykorzystuje wzór pochodnej wyprowadzony na wykładzie.

```
def loss_fun(y, y_pred):
    epsilon = 1e-15  # zapobieganie log(0)
    loss = - (y * np.log(y_pred + epsilon) + (1 - y) * np.log(1 - y_pred +
epsilon))
    return loss


def update_weights(X, y, y_pred, weights, bias, learning_rate):
    arX = np.squeeze(np.asarray(X))
    ary = np.squeeze(np.asarray((y_pred - y)))
    gradient_weights = np.dot(arX, ary)
    gradient_bias = np.sum(y_pred - y)
    #aktualizacja wag
    weights -= learning_rate * gradient_weights
    bias -= learning_rate * gradient_bias

    return weights, bias
```

Uczenie modelu

Model uczy się po jednym przykładzie. Zbieżność modelu zdefiniowałem jako wystarczająco małą zmianę funkcji kosztu (procentowo) i maksymalną liczbę epok.

```
learning_rate = 0.3
epochs = 1200
prev_loss = 1
avg_loss = 1
for epoch in range(epochs):
    loss_arr = []
    for i in range(len(x_train)):
        X = x_train[i]
        y = y_train[i]
        y_pred = sigmoid(np.dot(X, weights) + bias)
        loss = loss_fun(y, y_pred)
        loss_arr.append(loss)
        weights, bias = update_weights(X, y, y_pred, weights, bias,
learning_rate)
    if epoch % 100 == 0:
        prev_loss = avg_loss
        avg_loss = sum(loss_arr) / len(loss_arr)
        print(f"Epoch {epoch}: Average loss = {avg_loss}")
        #if (1-(avg_loss/prev_loss))*100 < 2: break #zbyt mała zmiana
f.kosztu procentowo

print("Trained Weights:", weights)
print("Trained Bias:", bias)
```

Następnie stworzyłem funkcję, która na podstawie przewidzianej wartości przyporządkowuje do niej klasę 0 lub 1.

```
def predict(x_data):
    y_preds = []
    for i in range(len(x_data)):
        X = x_data[i]
        y_pred = sigmoid(np.dot(X, weights) + bias)
```

```
            binary_prediction = 1 if y_pred >= 0.5 else 0
            y_preds.append(binary_prediction)
        return np.array(y_preds)
```

Ocena działania modelu

Dane treningowe

Wyniki:

Accuracy: 0.8388429752066116

Confusion: [[117 16]

[23 86]]

klasa	precision	recall	f1-score	support
0	0,84	0, 88	0,86	133
1	0,84	0,79	0,82	109

Dane testowe

```
y_pred = predict(x_test)
accuracy = accuracy_score(y_test, y_pred)
confusion = confusion_matrix(y_test, y_pred)
report = classification_report(y_test, y_pred)
print(f'Accuracy: {accuracy}')
print(f'Confusion: {confusion}')
print(f'Report: {report}')
```

Wyniki:

Accuracy: 0.8524590163934426

Confusion: [[28 3]

[6 24]]

klasa	precision	recall	f1-score	support
0	0,82	0,90	0,86	31
1	0,89	0,80	0,84	30

Wnioski

Jak wynika z przedstawionych powyżej wyników accuracy jest wyższe w danych testowych, co jest niespodziewane. Może to prawdopodobnie wynikać z niereprezentatywnego podzielenia danych na treningowe i testowe. Właściwie wszystkie metryki oceny dla danych testowych wypadły lepiej albo porównywalnie do danych treningowych. Confusion matrix w obydwóch zbiorach wygląda podobnie i znacznie przeważają w niej wartości TP i TN co jest pożądanym rezultatem.

Raport 3

Krzysztof Maciejewski 260449

Wstęp

Zadanie polegało na zbudowaniu modelu wielowarstwowego sieci neuronowej. Jako funkcję aktywacji wykorzystałem sigmoidę, a na funkcję kosztu wybrałem entropię krzyżową.

Implementacja algorytmu

```
def initialize_network(n_inputs, hidden_layers, n_outputs):
    network = []

    for num_neurons in hidden_layers:
        hidden_layer = [{'weights': [uniform(1, 100) for i in
range(n_inputs + 1)]} for i in range(num_neurons)]
        network.append(hidden_layer)
        n_inputs = num_neurons

    output_layer = [{'weights': [uniform(1, 100) for i in range(n_inputs +
1)]} for i in range(n_outputs)]
    network.append(output_layer)

    return network
```

Sieć neuronowa została zaimplementowana jako lista słowników wag i biasów, które wypełniłem losowymi wartościami float od 1 do 100. Ta implementacja pozwala przekazać listę ukrytych warstw z ilością neuronów w każdej z nich.

```
def forward_propagate(network, row):
   inputs = row
   for layer in network:
      new_inputs = []
      for neuron in layer:
         activation = activate(neuron['weights'], inputs)
         neuron['output'] = transfer(activation)
         new_inputs.append(neuron['output'])
      inputs = new_inputs
   return inputs
```

W funkcji forward propagate iterujemy warstwa po warstwie. Dla każdego neuronu pobieramy jego wagi i obliczamy wartość wyjściową za pomocą funkcji transfer która w tym przypadku jest sigmoidem. Wartość wyjściową zapisujemy w słowniku, a wszystkie wartości wyjściowe warstwy zapisuje w liście będącą wartościami wejściowymi kolejnej warstwy.

```
def backward_propagate_error(network, expected):
   for i in reversed(range(len(network))):
      layer = network[i]
      errors = list()
      if i != len(network)-1:
         for j in range(len(layer)):
            error = 0.0
            for neuron in network[i + 1]:
               error += (neuron['weights'][j] * neuron['delta'])
            errors.append(error)
      else:
         for j in range(len(layer)):
            neuron = layer[j]
```

```
            errors.append(neuron['output'] - expected[j])

        for j in range(len(layer)):
            neuron = layer[j]
            neuron['delta'] = errors[j]*transfer_derivative(neuron['output'])
```

W tej funkcji implementuję propagację wsteczną błędu. Przechodzimy po neuronach od końcowych warstw modelu i obliczamy różnicę wartości wyjściowej i oczekiwanej. Dla warstw ukrytych również obliczam błąd mnożąc wagi razy deltę poprzedniej warstwy. Wartości delta są obliczane jako iloczyn błędu i pochodnej sigmoida biorącej jako argument wartość wyjściową neuronu.

```
def update_weights(network, row, l_rate):
    for i in range(len(network)):
        inputs = row[:-1] # bo ostatni jest bias
        if i != 0:
            inputs = [neuron['output'] for neuron in network[i - 1]]
        for neuron in network[i]:
            for j in range(len(inputs)):
                #aktualizacja wag w tablicy neuronu danej warstwy
                neuron['weights'][j] -= l_rate * neuron['delta'] * inputs[j]
            neuron['weights'][-1] -= l_rate * neuron['delta']
```

W funkcji aktualizacji wag wartość wyjściowa poprzedniej warstwy to wejście obecnej. Przechodzę neurony warstw i aktualizuję wagi odejmując od nich iloczyn delty (czyli iloczyn błędu i pochodnej sigmoida biorącej jako argument wartość wyjściową neuronu) i wartości wejściowych pomnożonych razy learning rate. Na koniec aktualizuję bias.

Jak model zachowuje się na zbiorze heart disease dla:

• Różnej wymiarowości warstwy ukrytej

Dla modelu z 3 warstwami ukrytymi o wymiarach:

- 5, 3, 2

Accuracy: 0.8360655737704918

Confusion: [[28 4]

[6 23]]

- 5, 5, 5

Accuracy: 0.7704918032786885

Confusion: [[21 11]

[3 26]]

-1, 1, 1

Accuracy: 0.7868852459016393

Confusion: [[23 9]

[4 25]]

- 20, 7, 3

Accuracy: 0.7540983606557377

Confusion: [[20 12]

[3 26]]

Najlepsze wartości okazały się być dla takich wymiarowości warstw ukrytych: 5, 3, 2. Dodanie większej liczby neuronów w warstwach ukrytych sprawiło, że sieć miała lepsze zdolności do modelowania zależności w danych. Jednak dużym zaskoczeniem było to, że sieć o wymiarach 1, 1, 1 miała również dobre wyniki.

• Różnej wartości współczynnika uczenia

Współczynnik uczenia	Accuracy	Confusion matrix
0,1	0.7213114754098361	[[18 14] [3 26]]
0,3	0.7704918032786885	[[23 9] [5 24]]
0,4	0.7704918032786885	[[23 9] [5 24]]
0,7	0.8360655737704918	[[28 4] [6 23]]

Prawdopodobnie przez małą liczbę epok użytą do tego eksperymentu modele z najwyższym współczynnikiem uczenia poradziły sobie najlepiej i zdążyły się więcej nauczyć.

• Różnych odchyleń standardowych przy inicjalizacji wag

Aby sprawdzić różne odchylenia przy inicjalizacji wag korzystam z tej funkcji:

np.random.normal(15, std_dev, n_inputs + 1)

Po analizie danych założyłem, że odchylenia standardowe będą miały środek rozkładu = 15.

Odchylenie standardowe	Accuracy	Confusion Matrix
0,1	0.7868852459016393	[[32 0] [13 16]]
25	0.6721311475409836	[[31 1] [19 10]]
100	0.5573770491803278	[[27 5] [22 7]]

Przy różnych odchylenia standardowych ważne jest też poprawne dobranie środka rozkładu.

• Danych znormalizowanych i nieznormalizowanych

Dla danych nieznormalizowanych skale poszczególnych cech są nieprzygotowane do modelowania. Niektóre cechy z większymi wartościami mogą dominować inne przez co model dłużej się uczy i tak w rzeczywistości się stało.

Dane nieznormalizowane:

Accuracy: 0.5081967213114754

Confusion: [[1 28]

[2 30]]

Dane znormalizowane:

Accuracy: 0.7213114754098361

Confusion: [[37 0]

[17 7]]

• Różnej liczby warstw

Dobre dostosowanie liczby warstw ma istotny wpływ na zdolność modelowania zależności przez sieć. W moim przypadku płytsza sieć okazała się być lepszym wyborem, ponieważ mamy tutaj do czynienia z dość prostym problemem, który nie wymaga głębokiej hierarchii cech.

Liczba warstw	Accuracy	Confusion Matrix
1	0.7377049180327869	[[37 0] [16 8]]
2	0.7213114754098361	[[37 0] [17 7]]
3	0.7213114754098361	[[37 0] [17 7]]

Wnioski

Dużym problemem podczas przeprowadzania sprawdzenia jakości modelu okazało się być dostosowanie wartości na podstawie której stwierdzam czy dane wyjście powinno być 0 czy 1. Jego błędne dostosowanie prowadziło do niepoprawnych confusion matrix np. takich jak ta:

[[0 32]

[0 29]]

Raport 3

Krzysztof Maciejewski 260449

Wstęp

Zadanie polegało na odtworzeniu zaimplementowanego modelu wielowarstwowego sieci neuronowej korzystając z gotowego rozwiązania do budowania sieci neuronowych. Zdecydowałem się na skorzystanie z biblioteki PyTorch.

Implementacja

Mój model sieci został zaimplementowany poprzez klasę NeuralNetwork rozszerzającą nn.Module z jedną warstwą ukrytą.

```
class NeuralNetwork(nn.Module):
    def __init__(self):
        super(NeuralNetwork, self).__init__()
        self.input_layer = nn.Linear(x_train.shape[1], 64)
        self.hidden_layer = nn.Linear(64, 32)
        self.output_layer = nn.Linear(32, 1)
        self.sigmoid = nn.Sigmoid()
```

```
    def forward(self, x):
        x = self.input_layer(x)
        x = self.sigmoid(x)
        x = self.hidden_layer(x)
        x = self.sigmoid(x)
        x = self.output_layer(x)
        return x

model = NeuralNetwork()
```

Trenowanie sieci

Podczas pracy nad raportem postanowiłem sprawdzić 3 optymalizatory z biblioteki PyTorch: Stochastic Gradient Descend, Adam oraz Adagard. Używam entropii krzyżowej jako funkcji kosztu klasyfikacji binarnej. Pętla treningowa składa się z kroków forward pass w którym to jest obliczany wynik z modelu. Obliczam funkcję kosztu przekazując wyniki z modelu i oczekiwane wartości. Następnie obliczam gradienty wykorzystując propagację wsteczną. Funkcja *step()* aktualizuje parametry modelu za pomocą wybranego optymalizatora i przyjętej reguły uczenia.

```
criterion = nn.BCEWithLogitsLoss()
optimizer = optim.SGD(model.parameters(), lr=0.1)
# optimizer = optim.Adam(model.parameters(), lr=0.1)
# optimizer = optim.Adagrad(model.parameters(), lr=0.1)


num_epochs = 10000

for epoch in range(num_epochs):
    outputs = model(x_train)
    loss = criterion(outputs, y_train)

    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    if (epoch + 1) % 100 == 0:
        print(f'Epoch [{epoch+1}/{num_epochs}], Loss: {loss.item():.4f}')
```

Sprawdzenie wpływu:

Badania przeprowadzone dla 10000 epok.

- wybranego optimizera (SGD, Adam, Adagard)

Optimizer	Accuracy	Confusion matrix
SGD Learning rate = 0.1	81.97%	[[26 2] [9 24]]
Adam Learning rate = 0.1	80.33%	[[31 10] [2 18]]
Adagard Learning rate = 0.1	75.41%	[[28 5] [10 18]]

Adam prawdopodobnie zaczął się przeuczać. Dla Adagarda learning rate mógł być zbyt mały.

• rozmiaru batcha

Rozmiar batcha	Accuracy	Confusion matrix
32	85.25%	[[38 2] [7 14]]
64	88.52%	[[31 1] [6 23]]
128	81.97%	[[32 3] [8 18]]

• wartości współczynnika uczenia dla różnych optimizerów

Learning rate = 0.1

Optimizer	Accuracy	Confusion matrix
SGD	81.97%	[[26 2] [9 24]]
Adam	80.33%	[[31 10] [2 18]]
Adagard	75.41%	[[28 5] [10 18]]

Learning rate = 0.2

Optimizer	Accuracy	Confusion matrix
SGD	95.08%	[[31 3] [0 27]]
Adam	55.74%	[[34 0] [27 0]]
Adagard	83.61%	[[24 10] [3 24]]

Learning rate = 0.3

Optimizer	Accuracy	Confusion matrix
SGD	59.02%	[[31 0] [25 5]]
Adam	73.77%	[[22 9] [7 23]]
Adagard	72.13%	[[21 10] [7 23]]

Interesujące są wyniki dla learning rate = 0.2, ponieważ są one najbardziej zróżnicowane SGD osiągnął tam najlepszy wynik równy 95%, natomiast Adam najgorszy równy 55%.

Podsumowanie

Na podstawie przeprowadzonych badań można stwierdzić, że największy wpływ na końcowy wynik miało odpowiednie dobranie parametru uczenia i optimizera. To właśnie ta kombinacja pozwoliłą w przypadku learning rate = 0.2 dla SGD osiągnąć accuracy = 95%.