Rzeszów, 27.05.2024

# Sztuczna Inteligencja

## Raport z wykonania projektu:

Porównanie algorytmów wstecznej propagacji błędu, LVQ, sieci LSTM oraz drzewa decyzyjnego jako klasyfikatorów danych ze zbioru Amphibians


**Autor:**

Michał Mroczka

173677

2 EF-DI L06

# Spis treści

# 1 Wstęp

Niniejsza praca będzie dotyczyła problematyki klasyfikacji wieloetykietowej. W przypadku tego projektu przedstawiać ona będzie klasyfikację gatunków płazów. Szczegóły dotyczące problemu znajdują się na stronie [1]. W skrócie celem jest przewidywanie obecności gatunków płazów w pobliżu zbiorników wodnych na podstawie cech uzyskanych z systemów GIS i zdjęć satelitarnych. Uczenie na podstawie tych danych będzie wykonywane za pomocą algorytmów: wstecznej propagacji błędu, LVQ, LSTM oraz drzewa decyzyjnego.

## Czym jest klasyfikacja wieloetykietowa?

Jest to zadanie z zakresu uczenia maszynowego i analizy danych, które polega na przypisaniu jednemu obiektowi lub przykładów wiele etykiet jednocześnie. W odróżnieniu od klasyfikacji jednoetykietowej, gdzie każdy przykład należy do jednej z określonych klas, w klasyfikacji wieloetykietowej każdy przykład może być przypisany do wielu klas jednocześnie.

## Środki pomocne w realizacji projektu:

Do realizacji projektu zostały wykorzystane następujące narzędzia:

Microsoft Excel 2022 – arkusz kalkulacyjny, który został użyty do wstępnej analizy danych oraz modyfikacji (o których więcej będzie w rozdziale Dane) potrzebnych do późniejszej obróbki

Python – główny język programowania w którym były pisane skrypty

Wraz z Pythonem następujące biblioteki:

Tensorflow, keras i scikit-learn - budowa, trenowanie oraz testowanie modeli sieci neuronowych

Numpy – efektywne przetwarzanie danych liczbowych

Pandas – zarządzanie oraz analiza danych

Matplotlib – wizualizacja danych, w tym tworzenie wykresów oraz wizualizacja wyników

# 2 Dane

Dane do realizacji tego projektu zostały pobrane ze strony [1]. Zawierały one pojedynczy plik dataset.csv. Do otwarcia, analizy oraz modyfikacji tej bazy wykorzystałem program Microsoft Excel 2022, jak wspomniałem we wcześniejszym rozdziale. W oryginalnej wersji plik zawierał 23 kolumny oraz 191 wierszy. Pierwszy wiersz zawierał informację dla każdej kolumny jakiego typu będzie dana zmienna. Można było wyróżnić 5 typów danych: Integer(liczba całkowita), Numerical (zmienna numeryczna), Categorical (zmienna kategoryczna), Ordinal (zmienna porządkowa), Label (etykieta). Drugi wiersz określał nazwę zmiennej w każdej kolumnie. Trzeci i następne wiersze zawierały wartości dla każdych zmiennych.

Oto jak wyglądała oryginalna baza danych:

| | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Integer | Categorica | Numerical | Numerical | Categorica | Categorica | Categorica | Categorica | Categorica | Categorica | Categorica | Numerical | Ordinal | Ordinal | Categorica | Categorica | Label 1 | Label 2 | Label 3 | Label 4 | Label 5 | Label 6 | Label 7 |
| | ID | Motorway | SR | NR | TR | VR | SUR1 | SUR2 | SUR3 | UR | FR | OR | RR | BR | MR | CR | Green frog | Brown frog | Common t | Fire-bellie | Tree frog | Common r | Great crested |
| | 1 | A1 | 600 | 1 | 1 | 4 | 6 | 2 | 10 | 0 | 0 | 50 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | 2 | A1 | 700 | 1 | 5 | 1 | 10 | 6 | 10 | 3 | 1 | 75 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 0 |
| | 3 | A1 | 200 | 1 | 5 | 1 | 10 | 6 | 10 | 3 | 4 | 75 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 0 |
| | 4 | A1 | 300 | 1 | 5 | 0 | 6 | 10 | 2 | 3 | 4 | 25 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| | 5 | A1 | 600 | 2 | 1 | 4 | 10 | 2 | 6 | 0 | 0 | 99 | 0 | 5 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 |
| | 6 | A1 | 200 | 1 | 5 | 1 | 6 | 6 | 10 | 1 | 0 | 50 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | 7 | A1 | 500 | 1 | 5 | 0 | 6 | 6 | 10 | 1 | 2 | 25 | 0 | 0 | 0 | 2 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 0 | 8 | A1 | 700 | 1 | 5 | 2 | 10 | 6 | 9 | 0 | 0 | 100 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 |
| 1 | 9 | A1 | 750 | 2 | 5 | 1 | 6 | 1 | 2 | 0 | 1 | 50 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| | 10 | A1 | 200 | 1 | 12 | 4 | 2 | 7 | 6 | 0 | 0 | 75 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |

*2.1 Fragment oryginalnej bazy danych*

Jednakże pozwoliłem sobie na modyfikacje. Na początku zamieniłem miejscami wiersze pierwsze z drugim, aby łatwiej było w skrypcie przygotowującym dane do obróbki (o którym będzie w późniejszym etapie rozdziału) odczytać nazwy zmiennych. Następnym krokiem w modyfikacji było usunięcie niektórych kolumn.

Usunięte kolumny:

- TR – zmienna kategoryczna, oznaczała ona rodzaj zbiorników wodnych.

- FR – zmienna kategoryczna, oznaczała obecność połowów

- SUR1, SUR2, SUR3 – tak jak poprzednie również zmienne kategoryczne, odpowiednio to były dominujący, drugi pod względem dominujący oraz trzeci pod względem dominujący

Podane zmienne zostały usunięte z powodu niespójności z oryginalnym opisem danych znajdującym się na stronie [1].

Wygląd bazy danych w Excelu po modyfikacji:

| ID | Motorway | SR | NR | VR | UR | OR | RR | BR | MR | CR | Green frog | Brown frog | Common t | Fire-bellie | Tree frog | Common r | Great crested newt |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Integer | Categorica | Numerical | Numerical | Categorica | Categorica | Numerical | Ordinal | Ordinal | Categorica | Categorica | Label 1 | Label 2 | Label 3 | Label 4 | Label 5 | Label 6 | Label 7 |
| 1 | A1 | 600 | 1 | 4 | 0 | 50 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 2 | A1 | 700 | 1 | 1 | 3 | 75 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 0 |
| 3 | A1 | 200 | 1 | 1 | 3 | 75 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 0 |
| 4 | A1 | 300 | 1 | 0 | 3 | 25 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 5 | A1 | 600 | 2 | 4 | 0 | 99 | 0 | 5 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 |
| 6 | A1 | 200 | 1 | 1 | 1 | 50 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 7 | A1 | 500 | 1 | 0 | 1 | 25 | 0 | 0 | 0 | 2 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 8 | A1 | 700 | 1 | 2 | 0 | 100 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 |
| 9 | A1 | 750 | 2 | 1 | 0 | 50 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| 10 | A1 | 200 | 1 | 4 | 0 | 75 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |

*2.2 Baza danych po obróbce*

Opis danych wykorzystanych w projekcie:

ID - identyfikator wektora (nieuwzględniona w obliczeniach), zmienna Integer

MV - autostrada (nieuwzględniona w obliczeniach), wartości: A1 lub S52, zmienna Categorical

SR - powierzchnia zbiornika wodnego (m2), zmienna Numerical

NR - liczba zbiorników wodnych, zmienna Numerical

VR – zmienna Categorical, występowanie roślinności w obrębie zbiorników, w nawiasach zawarte są wartości w Excelu:

A. brak roślinności (0)

B. wąskie plamy na krawędziach (1)

C. tereny silnie zarośnięte (2)

D. bujna roślinność zbiornika, której część jest pozbawiona roślinności (3)

E. zbiorniki całkowicie zarośnięte zanikającym zwierciadłem wody (4)

UR – Korzystanie ze zbiorników wodnych, zmienna Categorical:

A. nieużytkowany przez człowieka (bardzo atrakcyjny dla płazów) – wartość 0

B. rekreacyjne i widokowe (wykonywane są prace opiekuńcze) – wartość 1

C. wykorzystywane gospodarczo (często hodowla ryb) – wartość 2

D. techniczny – wartość 3

OR - Procentowy dostęp od krawędzi zbiornika do terenów niezabudowanych (proponowane przedziały procentowe stanowią liczbowe odzwierciedlenie wyrażeń: brak dostępu, niski dostęp, średni dostęp, duży dostęp do wolnej przestrzeni), zmienna Numerical:

A. 0–25% – brak dostępu lub słaby dostęp – wartość

B. 25–50% — niski dostęp

C. 50–75% - średni dostęp,

D. 75–100%–duży dostęp do siedlisk lądowych, linia brzegowa styka się z lądowym siedliskiem płazów.

RR - Minimalna odległość zbiornika wodnego od dróg, zmienna Ordinal:

A. poniżej 50 m – wartość 0

B. 50–100 m – wartość 1

C. 100–200 m – wartość 2

D. 200–500 m – wartość 5

mi. 500–1000 m – wartość 9

F. powyżej 1000 m – wartość 10

BR – Zabudowa – Minimalna odległość od budynków, zmienna Ordinal:

A. poniżej 50 m – wartość 0

B. 50–100 m – wartość 1

C. 100–200 m – wartość 2

D. 200–500 m – wartość 5

mi. 500–1000 m – wartość 9

F. powyżej 1000 m – wartość 10

MR - Stan utrzymania zbiornika, wartość w Excelu w nawiasie, zmienna Categorical:

A. Czysty (0)

B. lekko zaśmiecone (1)

C. zbiorniki silnie lub bardzo mocno zaśmiecone (2)

CR - Rodzaj brzegu, wartość w Excelu w nawiasie, zmienna Categorical:

A. Naturalny (1)

B. Beton (2)

Ostatnie siedem kolumn zawierały etykiety, przyjmują one wartości 0 i 1:

Green frog – obecność żab zielonych

Brown frog – obecność żab brunatnych

Common toad – obecność ropuchy szarej

Fire-bellied toad – obecność ropuchy plamistej

Tree frog – obecność żaby drzewnej

Common newt – obecność traszki zwyczajnej

Great crested newt – obecność traszki grzebieniastej

## Implementacja przygotowania danych do uczenia

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, OneHotEncoder, OrdinalEncoder
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline

# Wczytanie danych, pomijając drugi wiersz, który zawiera typy danych
data = pd.read_csv('dataset_improved.csv', delimiter=';', skiprows=[1])

# Usunięcie kolumn 'ID' i 'Motorway'
data.drop(['ID', 'Motorway'], axis=1, inplace=True)

# Listy kolumn dla różnych typów danych
numeric_features = ['SR', 'NR', 'OR']
categorical_features = ['VR', 'UR', 'MR', 'CR']
ordinal_features = ['RR', 'BR']

# Dane wejściowe i etykiety
X = data.drop(data.columns[-7:], axis=1)  # Zakładamy, że ostatnie 7 kolumn to
etykiety

y = data[data.columns[-7:]]  # Etykiety

# Tworzenie transformatorów dla numerycznych, kategorycznych i porządkowych
danych
preprocessor = ColumnTransformer(
    transformers=[
        ('num', StandardScaler(), numeric_features),
        ('cat', OneHotEncoder(), categorical_features),
        ('ord', OrdinalEncoder(), ordinal_features)  # Użycie OrdinalEncoder dla
danych porządkowych
    ])

# Pipeline do przetwarzania i modelowania danych
pipeline = Pipeline(steps=[('preprocessor', preprocessor)])

# Przetwarzanie danych
X_processed = pipeline.fit_transform(X)

# Podział danych na zestaw treningowy i testowy
X_train, X_test, y_train, y_test = train_test_split(X_processed, y,
test_size=0.2, random_state=42)

print("Przetworzone dane treningowe:", X_train.shape)
print("Przetworzone dane testowe:", X_test.shape)

# Zapis danych treningowych i testowych do plików CSV
pd.DataFrame(X_train).to_csv('X_train.csv', index=False)
pd.DataFrame(X_test).to_csv('X_test.csv', index=False)
pd.DataFrame(y_train).to_csv('y_train.csv', index=False)
pd.DataFrame(y_test).to_csv('y_test.csv', index=False)
```

*2.1 Listing przygotowania danych do uczenia*

## Omówienie działania kodu

Rozpocząłem najpierw od wczytania danych z pliku w którym była już zmodyfikowana baza danych. Pominąłem wiersz, który zawierał typy danych.

```
# Wczytanie danych, pomijając drugi wiersz, który zawiera typy danych
data = pd.read_csv('dataset_improved.csv', delimiter=';', skiprows=[1])
```

*2.2 Listing - Wczytanie danych*

Następnie usunąłem kolumny „ID" i „Motorway", gdyż one nie będą używane podczas uczenia.

```
# Usunięcie kolumn 'ID' i 'Motorway'
data.drop(['ID', 'Motorway'], axis=1, inplace=True)
```

*2.3 Listing – Usunięcie kolumn*

Aby później ułatwić process transformacji danych to stworzyłem listę poszczególnych typów dla danych kolumn.

```
# Listy kolumn dla różnych typów danych
numeric_features = ['SR', 'NR', 'OR']
categorical_features = ['VR', 'UR', 'MR', 'CR']
ordinal_features = ['RR', 'BR']
```

*2.4 Listing - Wczytanie danych*

Zmiennymi X i y określiłem odpowiednio dane wejściowe i etykiety

```
# Dane wejściowe i etykiety
X = data.drop(data.columns[-7:], axis=1)  # Zakładamy, że ostatnie 7 kolumn to
etykiety
y = data[data.columns[-7:]]  # Etykiety
```

*2.5 Listing – dane wejściowe i etykiety*

Utworzyłem transformatory dla danych numerycznych, kategorycznych i porządkowych. Zdefiniowałem najpierw preprocesor, który korzysta z funkcji 'ColumnTransformer'. Pozwala ona na zastosowanie różnych przekształceń do różnych kolumn w zbiorze danych, co jest szczególnie przydatne, gdy mamy do czynienia z danymi mieszanymi (numerycznymi, kategorycznymi, porządkowymi). Jako argument funkcji zostaje przekazana lista transformers, która zawiera krotki definujące transformacje dla różnych typów danych.

Każda krotka ma trzy elementy:

- Pierwszy element: Nazwa transformacji (dowolny string).
- Drugi element: Transformator, który ma być zastosowany.
- Trzeci element: Lista kolumn, do których ma być zastosowany transformator.

W pierwszej krotce nastepuje przekształcenie cech numerycznych. Odbywa się ono poprzez funkcję StandardScaler(), który standaryzuje cechy poprzez usunięcie średniej i skalowanie do jednostkowej wariancji. Druga krotka przekształca cechy kategoryczne poprzez funkcje OneHotEncoder() na format one-hot encoding. Natomiast cechy porządkowe, które są przekształcane w trzeciej krotce, za ich obróbkę odpowiada funkcja OrdinalEncoder() przekształcając cechy na wartości liczbowe zachowując ich porządek

```
# Tworzenie transformatorów dla numerycznych, kategorycznych i porządkowych
danych
preprocessor = ColumnTransformer(
    transformers=[
        ('num', StandardScaler(), numeric_features),
        ('cat', OneHotEncoder(), categorical_features),
        ('ord', OrdinalEncoder(), ordinal_features)  # Użycie OrdinalEncoder dla
danych porządkowych
    ])
```

*2.6 Listing – utworzenie transformatorów*

Potem utworzyłem pipeline korzystając z obiektu biblioteki scikit-learn, umożliwia łączenie kilku etapów przetwarzania danych oraz modelowania w jeden proces. Parametr ‘steps’ przyjmuje listę krotek, gdzie każda krotka zawiera nazwę kroku i transformator lub model. W naszym przypadku wykorzystujemy krok ‘('preprocessor', preprocessor)’, który wykorzystuje wcześniej zdefiniowany transformator kolumnowy ‘preprocesor’. Po utworzeniu pipeline’u następuje proces przetwarzania danych. Utworzony został w tym przypadku zbiór danych ‘X_processed’ po zastosowaniu wszystkich transformacji zdefiniowanych w ‘preprocessor’. Metoda ‘fit_transform’ jest odpowiedzialna za dopasowania do danych ‘X’. Następstwem przetworzenia danych jest ich podział na zestaw treningowy i testowy. Wykorzystałem do tego funkcję ‘train_test_split’ z biblioteki scikit-learn. Jest ona używana do losowego podziału danych na zestaw treningowy i testowy.

Parametry funkcji ‘train_test_split’:

- X_processed: Przetworzony zbiór cech (features).
- y: Zbiór etykiet (labels) odpowiadający X.
- test_size=0.2: Wskazuje, że 20% danych ma być przeznaczone na zestaw testowy, a 80% na zestaw treningowy.
- random_state=42: Ustawia losowy seed, aby podział danych był powtarzalny (każde uruchomienie kodu da ten sam podział danych).

```
# Pipeline do przetwarzania i modelowania danych
pipeline = Pipeline(steps=[('preprocessor', preprocessor)])

# Przetwarzanie danych
X_processed = pipeline.fit_transform(X)

# Podział danych na zestaw treningowy i testowy
X_train, X_test, y_train, y_test = train_test_split(X_processed, y, test_size=0.2,
random_state=42)
```

*2.7 Listing – Przetwarzanie i podział danych*

Końcowym eteapem przygotowania danych do obróbki jest zapisanie danych treningowych i testowych do plików z rozszerzeniem .csv korzystając z metod biblioteki Pandas: ‘DataFrame’ i ‘to_csv’.

```
# Zapis danych treningowych i testowych do plików CSV
pd.DataFrame(X_train).to_csv('X_train.csv', index=False)
pd.DataFrame(X_test).to_csv('X_test.csv', index=False)
pd.DataFrame(y_train).to_csv('y_train.csv', index=False)
pd.DataFrame(y_test).to_csv('y_test.csv', index=False)
```

*2.8 Listing – Zapis danych do pliku*

# 3 Algorytm wstecznej propagacji błędu

## Teoria

### Wprowadzenie

Algorytm wstecznej propagacji, znany również jako backpropagation, jest jednym z najważniejszych algorytmów stosowanych w sztucznej inteligencji, szczególnie w kontekście sieci neuronowych. Jego głównym celem jest optymalizacja wag w sieciach neuronowych w celu minimalizacji błędu predykcji. Algorytm ten jest kluczowy dla procesu uczenia się nadzorowanego i jest powszechnie stosowany w wielu aplikacjach, takich jak rozpoznawanie obrazów, przetwarzanie języka naturalnego czy systemy rekomendacyjne.

### Podstawy teoretyczne

Wsteczna propagacja opiera się na metodzie gradientu prostego, która jest techniką optymalizacyjną służącą do minimalizacji funkcji kosztu. Podstawowym celem algorytmu jest aktualizacja wag sieci neuronowej w taki sposób, aby zmniejszyć błąd między rzeczywistym wynikiem a wynikiem przewidywanym przez sieć. Proces ten odbywa się poprzez propagowanie błędu wstecz przez sieć, stąd nazwa 'wsteczna propagacja'.

### Kroki algorytmu wstecznej propagacji

1. Inicjalizacja wag:  Na początku wszystkie wagi w sieci są inicjalizowane losowymi wartościami.

2. Propagacja w przód:  Dla każdego wejścia w zbiorze treningowym obliczane są wyjścia dla każdej warstwy sieci aż do uzyskania końcowego wyniku predykcji.

3. Obliczenie błędu:  Różnica między przewidywanym wynikiem a rzeczywistym wynikiem jest używana do obliczenia błędu.

4. Propagacja wstecz: Błąd jest propagowany wstecz przez sieć w celu obliczenia gradientów dla wag. Gradienty te pokazują, jak bardzo każda waga wpływa na błąd.

5.  Aktualizacja wag: Wagi są aktualizowane w kierunku przeciwnym do gradientu błędu, co zmniejsza wartość funkcji kosztu. Proces ten jest powtarzany dla każdego przykładu w zbiorze treningowym.

## Implementacja uczenia sieci neuronowej

```
import numpy as np
import pandas as pd

# Załadowanie danych
X_train = pd.read_csv('X_train.csv').values
X_test = pd.read_csv('X_test.csv').values
y_train = pd.read_csv('y_train.csv').values
y_test = pd.read_csv('y_test.csv').values

#Normalizacja danych
X_train = X_train / np.max(X_train)
X_test = X_test / np.max(X_test)

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

def sigmoid_derivative(x):
    return x * (1 - x)

input_layer_neurons = X_train.shape[1]
hidden_layer1_neurons = 64
hidden_layer2_neurons = 32
output_layer_neurons = y_train.shape[1]

# Inicjalizacja wag
wh1 = np.random.uniform(size=(input_layer_neurons,
hidden_layer1_neurons))
bh1 = np.random.uniform(size=(1, hidden_layer1_neurons))
wh2 = np.random.uniform(size=(hidden_layer1_neurons,
hidden_layer2_neurons))
bh2 = np.random.uniform(size=(1, hidden_layer2_neurons))
wo = np.random.uniform(size=(hidden_layer2_neurons,
output_layer_neurons))
bo = np.random.uniform(size=(1, output_layer_neurons))

learning_rate = 0.01
epochs = 1000

for epoch in range(epochs):
    # Forward Propagation
    hidden_layer1_input = np.dot(X_train, wh1) + bh1
    hidden_layer1_output = sigmoid(hidden_layer1_input)

    hidden_layer2_input = np.dot(hidden_layer1_output, wh2) + bh2
    hidden_layer2_output = sigmoid(hidden_layer2_input)

    output_layer_input = np.dot(hidden_layer2_output, wo) + bo
    predicted_output = sigmoid(output_layer_input)

    # Backpropagation
    error = y_train - predicted_output
    d_predicted_output = error * sigmoid_derivative(predicted_output)

    error_hidden_layer2 = d_predicted_output.dot(wo.T)
```

```
    d_hidden_layer2 = error_hidden_layer2 *
sigmoid_derivative(hidden_layer2_output)

    error_hidden_layer1 = d_hidden_layer2.dot(wh2.T)
    d_hidden_layer1 = error_hidden_layer1 *
sigmoid_derivative(hidden_layer1_output)

    # Aktualizacja wag i biasów
    wo += hidden_layer2_output.T.dot(d_predicted_output) * learning_rate
    bo += np.sum(d_predicted_output, axis=0, keepdims=True) *
learning_rate

    wh2 += hidden_layer1_output.T.dot(d_hidden_layer2) * learning_rate
    bh2 += np.sum(d_hidden_layer2, axis=0, keepdims=True) * learning_rate

    wh1 += X_train.T.dot(d_hidden_layer1) * learning_rate
    bh1 += np.sum(d_hidden_layer1, axis=0, keepdims=True) * learning_rate

    if epoch % 100 == 0:
        loss = np.mean(np.square(y_train - predicted_output))
        print(f'Epoch {epoch}, Loss: {loss}')
```

*3.1 Implementacja learnbp*

# Omówienie działania kodu

## Importowanie bibliotek i wczytywanie danych

```
import numpy as np
import pandas as pd

# Załadowanie danych
X_train = pd.read_csv('X_train.csv').values
X_test = pd.read_csv('X_test.csv').values
y_train = pd.read_csv('y_train.csv').values
y_test = pd.read_csv('y_test.csv').values
```

*3.2 Importowanie bibliotek i wczytanie danych*

## Normalizacja danych

```
#Normalizacja danych
X_train = X_train / np.max(X_train)
X_test = X_test / np.max(X_test)
```

*3.3 Normalizacja*

Normalizacja danych: Skaluje dane wejściowe, dzieląc je przez maksymalną wartość w zbiorze treningowym, aby wartości mieściły się w przedziale [0, 1].

### Definicje funkcji aktywacji i ich pochodnych

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

def sigmoid_derivative(x):
    return x * (1 - x)
```

*3.4 Funkcje aktywacji i ich pochodnych*

- Funkcja sigmoid: Stosowana jako funkcja aktywacji, przekształca wejścia w przedział (0, 1).
- Pochodna funkcji sigmoid: Używana do obliczeń podczas backpropagation.

### Inicjalizacja parametrów sieci

```
input_layer_neurons = X_train.shape[1]
hidden_layer1_neurons = 64
hidden_layer2_neurons = 32
output_layer_neurons = y_train.shape[1]

# Inicjalizacja wag
wh1 = np.random.uniform(size=(input_layer_neurons,
hidden_layer1_neurons))
bh1 = np.random.uniform(size=(1, hidden_layer1_neurons))
wh2 = np.random.uniform(size=(hidden_layer1_neurons,
hidden_layer2_neurons))
bh2 = np.random.uniform(size=(1, hidden_layer2_neurons))
wo = np.random.uniform(size=(hidden_layer2_neurons,
output_layer_neurons))
bo = np.random.uniform(size=(1, output_layer_neurons))
```

*3.5 Inicjalizacja parametrów sieci*

- **Definicja liczby neuronów**: Ustalanie liczby neuronów w każdej warstwie (wejściowa, dwie ukryte, wyjściowa).
- **Inicjalizacja wag i biasów**: Losowa inicjalizacja wag i biasów dla każdej warstwy.

### Parametry treningowe

```
learning_rate = 0.01
epochs = 1000
```

*3.6 Parametry treningowe*

- **Learning rate**: Współczynnik uczenia, określający krok aktualizacji wag.
- **Epochs**: Liczba epok treningowych, czyli ile razy cały zbiór treningowy zostanie przetworzony.

**Trening modelu**

```
for epoch in range(epochs):
    # Forward Propagation
    hidden_layer1_input = np.dot(X_train, wh1) + bh1
    hidden_layer1_output = sigmoid(hidden_layer1_input)

    hidden_layer2_input = np.dot(hidden_layer1_output, wh2) + bh2
    hidden_layer2_output = sigmoid(hidden_layer2_input)

    output_layer_input = np.dot(hidden_layer2_output, wo) + bo
    predicted_output = sigmoid(output_layer_input)

    # Backpropagation
    error = y_train - predicted_output
    d_predicted_output = error * sigmoid_derivative(predicted_output)

    error_hidden_layer2 = d_predicted_output.dot(wo.T)
    d_hidden_layer2 = error_hidden_layer2 *
sigmoid_derivative(hidden_layer2_output)

    error_hidden_layer1 = d_hidden_layer2.dot(wh2.T)
    d_hidden_layer1 = error_hidden_layer1 *
sigmoid_derivative(hidden_layer1_output)

    # Aktualizacja wag i biasów
    wo += hidden_layer2_output.T.dot(d_predicted_output) * learning_rate
    bo += np.sum(d_predicted_output, axis=0, keepdims=True) *
learning_rate

    wh2 += hidden_layer1_output.T.dot(d_hidden_layer2) * learning_rate
    bh2 += np.sum(d_hidden_layer2, axis=0, keepdims=True) * learning_rate

    wh1 += X_train.T.dot(d_hidden_layer1) * learning_rate
    bh1 += np.sum(d_hidden_layer1, axis=0, keepdims=True) * learning_rate

    if epoch % 100 == 0:
        loss = np.mean(np.square(y_train - predicted_output))
        print(f'Epoch {epoch}, Loss: {loss}')
```

*3.7 Trening modelu*

- Forward Propagation:
    1. Obliczanie wartości wyjściowych dla pierwszej warstwy ukrytej.
    2. Obliczanie wartości wyjściowych dla drugiej warstwy ukrytej.
    3. Obliczanie wartości wyjściowych dla warstwy wyjściowej.

- Backpropagation:
    1. Obliczanie błędu wyjściowego (różnica między wartością oczekiwaną a przewidywaną).
    2. Obliczanie pochodnych błędów dla każdej warstwy.
    3. Aktualizacja wag i biasów na podstawie obliczonych błędów i pochodnych.
- Aktualizacja wag i biasów: Uaktualnianie wag i biasów przy użyciu obliczonych pochodnych i współczynnika uczenia.
- Monitorowanie straty: Co 100 epok obliczanie i wyświetlanie wartości funkcji straty (średni kwadrat błędu).

# 4 Algorytm LVQ

## Teoria

Algorytm LVQ (Learning Vector Quantization) to metoda nadzorowanego uczenia maszynowego, która służy do klasyfikacji danych. Jest to technika, która łączy zalety sieci neuronowych i wektorów prototypowych. Algorytm LVQ został opracowany przez Teuvo Kohonena na początku lat 80. i jest szczególnie użyteczny w sytuacjach, gdy dane są w dużej mierze nieliniowe, a klasyfikacja wymaga wyraźnych granic decyzji.

Podstawowym założeniem algorytmu LVQ jest wykorzystanie prototypów, które reprezentują różne klasy w przestrzeni cech. Proces uczenia polega na optymalizacji położenia tych prototypów w taki sposób, aby minimalizować błąd klasyfikacji. Prototypy są aktualizowane na podstawie przykładów treningowych, co pozwala im dostosowywać się do struktury danych.

Algorytm LVQ działa według następujących kroków:

1. **Inicjalizacja prototypów**: Na początku ustala się liczbę prototypów oraz ich początkowe położenie w przestrzeni cech. Prototypy mogą być inicjalizowane losowo lub na podstawie wybranych przykładów z każdej klasy.
2. **Uczenie prototypów**: Dla każdego przykładu z zestawu treningowego algorytm identyfikuje najbliższy prototyp (zwykle za pomocą metryki euklidesowej). Następnie, w zależności od tego, czy prototyp poprawnie klasyfikuje dany przykład, jest on przesuwany w kierunku lub od tego przykładu:
    - Jeśli przykład jest poprawnie klasyfikowany (należy do tej samej klasy co prototyp), prototyp jest przesuwany bliżej przykładu.
    - Jeśli przykład jest błędnie klasyfikowany (należy do innej klasy niż prototyp), prototyp jest przesuwany dalej od przykładu.

Proces ten jest regulowany przez współczynnik uczenia, który zwykle zmniejsza się w miarę postępu uczenia, aby stabilizować położenie prototypów.

3. **Iteracja**: Krok 2 jest powtarzany dla wszystkich przykładów w zestawie treningowym przez określoną liczbę epok lub do momentu, gdy zmiany w położeniach prototypów staną się znikome.
4. **Klasyfikacja**: Po zakończeniu etapu uczenia, algorytm jest gotowy do klasyfikacji nowych przykładów. Nowe dane są przypisywane do klasy najbliższego prototypu.

Zaletą algorytmu LVQ jest jego zdolność do tworzenia wyraźnych granic decyzji, co jest szczególnie przydatne w przypadkach, gdy klasy są dobrze zdefiniowane i oddzielne. Dodatkowo, LVQ jest stosunkowo prosty do zrozumienia i implementacji, co czyni go popularnym narzędziem w praktycznych zastosowaniach.

Jednak LVQ ma również pewne wady. Jego wydajność zależy od odpowiedniego doboru liczby prototypów oraz ich początkowego położenia. Zbyt mała liczba prototypów może prowadzić do niedostatecznego dopasowania modelu, podczas gdy zbyt duża liczba może prowadzić do przetrenowania. Ponadto, algorytm może być wrażliwy na wartości początkowe współczynnika uczenia i jego harmonogram zmniejszania.

## Implementacja uczenia sieci neuronowej

```
import numpy as np
import pandas as pd
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score


class LVQ:
    def __init__(self, n_prototypes_per_class=1, learning_rate=0.01,
n_epochs=100):
        self.n_prototypes_per_class = n_prototypes_per_class
        self.learning_rate = learning_rate
        self.n_epochs = n_epochs
        self.prototypes = None
        self.prototype_labels = None

    def _initialize_prototypes(self, X, y):
        n_classes = len(np.unique(y))
        self.prototypes = np.zeros((n_classes *
self.n_prototypes_per_class, X.shape[1]))
        self.prototype_labels = np.zeros(n_classes *
self.n_prototypes_per_class)

        for i in range(n_classes):
            class_indices = np.where(y == i)[0]
            selected_indices = np.random.choice(class_indices,
self.n_prototypes_per_class, replace=False)
            self.prototypes[i * self.n_prototypes_per_class:(i + 1) *
self.n_prototypes_per_class] = X[selected_indices]
            self.prototype_labels[i * self.n_prototypes_per_class:(i + 1)
* self.n_prototypes_per_class] = y[
                selected_indices]

    def _update_prototype(self, x, y):
        distances = np.linalg.norm(self.prototypes - x, axis=1)
        winner_index = np.argmin(distances)
```

```
        winner_label = self.prototype_labels[winner_index]

        if winner_label == y:
            self.prototypes[winner_index] += self.learning_rate * (x -
self.prototypes[winner_index])
        else:
            self.prototypes[winner_index] -= self.learning_rate * (x -
self.prototypes[winner_index])

    def fit(self, X, y):
        self._initialize_prototypes(X, y)
        for epoch in range(self.n_epochs):
            for i in range(len(X)):
                self._update_prototype(X[i], y[i])

    def predict(self, X):
        y_pred = np.zeros(X.shape[0])
        for i in range(X.shape[0]):
            distances = np.linalg.norm(self.prototypes - X[i], axis=1)
            winner_index = np.argmin(distances)
            y_pred[i] = self.prototype_labels[winner_index]
        return y_pred


# Wczytaj dane z plików CSV
X_train = pd.read_csv('X_train.csv').values
X_test = pd.read_csv('X_test.csv').values
y_train = pd.read_csv('y_train.csv').values
y_test = pd.read_csv('y_test.csv').values

# Standaryzacja danych
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Konwersja one-hot encoded etykiet do jednowymiarowych etykiet
y_train_1d = np.argmax(y_train, axis=1)
y_test_1d = np.argmax(y_test, axis=1)

# Inicjalizacja i trenowanie modelu LVQ
lvq = LVQ(n_prototypes_per_class=1, learning_rate=0.01, n_epochs=100)
lvq.fit(X_train_scaled, y_train_1d)

# Predykcja na danych testowych
y_pred = lvq.predict(X_test_scaled)

# Ewaluacja modelu
accuracy = accuracy_score(y_test_1d, y_pred)
print(f'Accuracy: {accuracy}')
```

*4.1 Implementacja sieci LVQ*

# Omówienie działania kodu

## Importowanie bibliotek

```
import numpy as np
import pandas as pd
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score
```

*4.2 Importowanie bibliotek*

## Definicja klasy LVQ

```
class LVQ:
    def __init__(self, n_prototypes_per_class=1, learning_rate=0.01,
n_epochs=100):
        self.n_prototypes_per_class = n_prototypes_per_class
        self.learning_rate = learning_rate
        self.n_epochs = n_epochs
        self.prototypes = None
        self.prototype_labels = None

    def _initialize_prototypes(self, X, y):
        n_classes = len(np.unique(y))
        self.prototypes = np.zeros((n_classes *
self.n_prototypes_per_class, X.shape[1]))
        self.prototype_labels = np.zeros(n_classes *
self.n_prototypes_per_class)

        for i in range(n_classes):
            class_indices = np.where(y == i)[0]
            selected_indices = np.random.choice(class_indices,
self.n_prototypes_per_class, replace=False)
            self.prototypes[i * self.n_prototypes_per_class:(i + 1) *
self.n_prototypes_per_class] = X[selected_indices]
            self.prototype_labels[i * self.n_prototypes_per_class:(i + 1)
* self.n_prototypes_per_class] = y[
                selected_indices]

    def _update_prototype(self, x, y):
        distances = np.linalg.norm(self.prototypes - x, axis=1)
        winner_index = np.argmin(distances)
        winner_label = self.prototype_labels[winner_index]

        if winner_label == y:
            self.prototypes[winner_index] += self.learning_rate * (x -
self.prototypes[winner_index])
        else:
            self.prototypes[winner_index] -= self.learning_rate * (x -
self.prototypes[winner_index])

    def fit(self, X, y):
        self._initialize_prototypes(X, y)
        for epoch in range(self.n_epochs):
            for i in range(len(X)):
                self._update_prototype(X[i], y[i])

    def predict(self, X):
```

```
        y_pred = np.zeros(X.shape[0])
        for i in range(X.shape[0]):
            distances = np.linalg.norm(self.prototypes - X[i], axis=1)
            winner_index = np.argmin(distances)
            y_pred[i] = self.prototype_labels[winner_index]
        return y_pred
```

*4.3 Listing - Definicja klasy*

### Omówienie klasy LVQ

- __init__: Inicjalizuje instancję klasy LVQ z określoną liczbą prototypów na klasę, współczynnikiem uczenia i liczbą epok.
- _initialize_prototypes: Inicjalizuje prototypy dla każdej klasy. Dla każdej klasy losowo wybiera przykłady z danych treningowych jako początkowe prototypy.
- _update_prototype: Aktualizuje prototypy. Oblicza odległości euklidesowe między danym przykładem a wszystkimi prototypami. Znajduje najbliższy prototyp (winner). Jeśli etykieta winnera jest zgodna z etykietą przykładu, aktualizuje prototyp w kierunku przykładu, w przeciwnym razie aktualizuje w przeciwnym kierunku.
- fit: Trenuje model LVQ. Inicjalizuje prototypy, a następnie przez określoną liczbę epok aktualizuje prototypy na podstawie wszystkich przykładów treningowych.
- predict: Klasyfikuje nowe przykłady. Oblicza odległości między każdym przykładem a prototypami i przypisuje etykietę najbliższego prototypu.

### Wczytywanie danych

```
# Wczytaj dane z plików CSV
X_train = pd.read_csv('X_train.csv').values
X_test = pd.read_csv('X_test.csv').values
y_train = pd.read_csv('y_train.csv').values
y_test = pd.read_csv('y_test.csv').values
```

*4.4 Listing - Wczytywanie danych*

Wczytuje dane treningowe i testowe z plików CSV do tablic numpy.

### Standaryzacja danych

```
# Standaryzacja danych
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

*4.5 Listing - Standaryzacja danych*

Standaryzuje dane treningowe i testowe, aby miały średnią 0 i odchylenie standardowe 1.

### Konwersja etykiet

```
# Konwersja one-hot encoded etykiet do jednowymiarowych etykiet
y_train_1d = np.argmax(y_train, axis=1)
y_test_1d = np.argmax(y_test, axis=1)
```

*4.6 Listing - Konwersja etykiet*

Konwertuje etykiety z formatu one-hot encoded do formatu jednowymiarowego.

### Trenowanie i ewaluacja modelu LVQ

```
# Inicjalizacja i trenowanie modelu LVQ
lvq = LVQ(n_prototypes_per_class=1, learning_rate=0.01, n_epochs=100)
lvq.fit(X_train_scaled, y_train_1d)

# Predykcja na danych testowych
y_pred = lvq.predict(X_test_scaled)

# Ewaluacja modelu
accuracy = accuracy_score(y_test_1d, y_pred)
print(f'Accuracy: {accuracy}')
```

*4.7 Listing - Trenowanie i ewaluacja modelu LVQ*

- Inicjalizuje model LVQ z jednym prototypem na klasę, współczynnikiem uczenia 0.01 i 100 epokami.
- Trenuje model na danych treningowych.
- Przeprowadza predykcję na danych testowych.
- Oblicza i drukuje dokładność modelu na danych testowych.

# 5 Sieć Long Short-Term Memory

## Teoria

LSTM (Long Short-Term Memory) to rodzaj sieci neuronowej zaprojektowanej w celu skutecznego przetwarzania i przewidywania sekwencji danych. Została opracowana przez Hochreitera i Schmidhubera w 1997 roku jako rozwiązanie problemów związanych z tradycyjnymi rekurencyjnymi sieciami neuronowymi (RNN), które mają trudności z uczeniem się długoterminowych zależności w sekwencjach. Kluczową innowacją LSTM jest wprowadzenie specjalnych komórek pamięci oraz bramek, które pozwalają na kontrolowanie przepływu informacji wewnątrz sieci. Komórki pamięci są zdolne do przechowywania informacji przez długie okresy, co pozwala sieci na skuteczne zarządzanie informacjami istotnymi dla długoterminowych zależności.

Architektura LSTM składa się z trzech głównych bramek: bramki wejściowej, bramki zapominania i bramki wyjściowej. Bramki te regulują przepływ informacji do i z komórki pamięci. Bramki wejściowe decydują, które nowe informacje powinny zostać dodane do komórki pamięci, bramki zapominania kontrolują, które informacje powinny zostać usunięte z komórki pamięci, a bramki wyjściowe określają, które informacje z komórki pamięci powinny być używane do obliczenia wyjścia sieci.

Działanie LSTM zaczyna się od przetworzenia sekwencji danych przez warstwę wejściową, która przekształca dane na wektory o odpowiednich rozmiarach. Następnie wektory te są przekazywane do komórek pamięci przez bramki wejściowe, które decydują, które elementy sekwencji są istotne i powinny być zapamiętane. Jednocześnie bramki zapominania usuwają nieistotne lub przestarzałe informacje z komórek pamięci. Proces ten jest powtarzany dla każdej jednostki czasowej w sekwencji, umożliwiając sieci LSTM gromadzenie i aktualizowanie informacji w sposób dynamiczny.

Po przetworzeniu całej sekwencji, wyjście z ostatniej jednostki czasowej jest przepuszczane przez bramki wyjściowe, które decydują, które informacje z komórek pamięci powinny zostać użyte do generowania ostatecznego wyjścia sieci. Dzięki tej strukturze LSTM jest zdolne do efektywnego uczenia się zarówno krótkoterminowych, jak i długoterminowych zależności w danych sekwencyjnych, co czyni je szczególnie użytecznymi w zadaniach takich jak rozpoznawanie mowy, tłumaczenie maszynowe, analiza sentymentu czy przewidywanie szeregów czasowych.

## Implementacja uczenia sieci neuronowej

```
import numpy as np
import pandas as pd
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense
from sklearn.preprocessing import StandardScaler

# Załadowanie danych
X_train = pd.read_csv('X_train.csv').values
X_test = pd.read_csv('X_test.csv').values
y_train = pd.read_csv('y_train.csv').values
y_test = pd.read_csv('y_test.csv').values
```

```
# Normalizacja danych
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Przekształcenie danych 2D na 3D (liczba próbek, liczba kroków
czasowych, liczba cech)
X_train = np.reshape(X_train, (X_train.shape[0], 1, X_train.shape[1]))
X_test = np.reshape(X_test, (X_test.shape[0], 1, X_test.shape[1]))

# Budowanie modelu LSTM
model = Sequential()
model.add(LSTM(64, input_shape=(X_train.shape[1], X_train.shape[2]),
return_sequences=True))
model.add(LSTM(32))
model.add(Dense(y_train.shape[1], activation='sigmoid'))

# Kompilacja modelu
model.compile(loss='binary_crossentropy', optimizer='adam',
metrics=['accuracy'])

# Wyświetlenie podsumowania modelu
model.summary()

# Trenowanie modelu
history = model.fit(X_train, y_train, epochs=100, batch_size=32,
validation_data=(X_test, y_test), verbose=2)

# Ocena modelu
loss, accuracy = model.evaluate(X_test, y_test)
print(f'Test Loss: {loss}')
print(f'Test Accuracy: {accuracy}')
```

*5.1 Listing - Implementacja sieci LSTM*

### Importowanie bibliotek

```
import numpy as np
import pandas as pd
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense
from sklearn.preprocessing import StandardScaler
```

*5.2 Listing - Importowanie bibliotek*

- **Sequential** z Keras do tworzenia sekwencyjnego modelu sieci neuronowej.
- **LSTM** i **Dense** z Keras do dodawania warstw LSTM i Dense do modelu.
- **StandardScaler** ze scikit-learn do normalizacji danych.

### Normalizacja danych

```
# Normalizacja danych
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

*5.3Listing - Normalizacja danych*

Normalizuje dane wejściowe (X_train i X_test) tak, aby miały średnią 0 i odchylenie standardowe 1. Jest to ważne, ponieważ normalizacja przyspiesza proces uczenia i może poprawić dokładność modelu.

### Przekształcenie danych z 2D na 3D

```
# Przekształcenie danych 2D na 3D (liczba próbek, liczba kroków
czasowych, liczba cech)
X_train = np.reshape(X_train, (X_train.shape[0], 1, X_train.shape[1]))
X_test = np.reshape(X_test, (X_test.shape[0], 1, X_test.shape[1]))
```

*5.4 Listing - Przekształcenie danych z 2D na 3D*

Dane są przekształcane z formatu 2D (liczba próbek, liczba cech) na format 3D (liczba próbek, liczba kroków czasowych, liczba cech). LSTM (Long Short-Term Memory) sieci neuronowe wymagają danych w formacie 3D, gdzie każda próbka ma określoną liczbę kroków czasowych i cech. W tym przypadku, każda próbka ma tylko jeden krok czasowy.

### Budowanie modelu LSTM

```
# Budowanie modelu LSTM
model = Sequential()
model.add(LSTM(64, input_shape=(X_train.shape[1], X_train.shape[2]),
return_sequences=True))
model.add(LSTM(32))
model.add(Dense(y_train.shape[1], activation='sigmoid'))
```

*5.5 Listing - Budowanie modelu LSTM*

Model LSTM jest tworzony jako sekwencyjny model Keras:

- Pierwsza warstwa LSTM ma 64 jednostki, przyjmuje dane wejściowe w formacie (1, liczba cech) i zwraca sekwencje (**return_sequences=True**).
- Druga warstwa LSTM ma 32 jednostki i zwraca ostatni stan ukryty.
- Warstwa Dense (gęsta) jest warstwą wyjściową z funkcją aktywacji sigmoid, co sugeruje, że problem jest klasyfikacją binarną lub wieloklasową (one-vs-all).

### Kompilacja modelu

```
# Kompilacja modelu
model.compile(loss='binary_crossentropy', optimizer='adam',
metrics=['accuracy'])
```

*5.6 Listing - Kompilacja modelu*

Model jest kompilowany z funkcją straty **binary_crossentropy**, optymalizatorem **adam** oraz metryką **accuracy**. **binary_crossentropy** jest odpowiednia dla problemów klasyfikacji binarnej lub wieloklasowej z kodowaniem one-hot.

### Wyświetlenie podsumowania modelu

```
# Wyświetlenie podsumowania modelu
model.summary()
```

*5.7 Lisitng - Wyświetlenie podsumowania modelu*

Wyświetla podsumowanie modelu, w tym informacje o każdej warstwie, liczbie parametrów itp.

### Trenowanie modelu

```
# Trenowanie modelu
history = model.fit(X_train, y_train, epochs=100, batch_size=32,
validation_data=(X_test, y_test), verbose=2)
```

*5.8 Listing - Trenowanie modelu*

Model jest trenowany na danych treningowych przez 100 epok, z batch size równym 32. validation_data to zestaw testowy używany do walidacji modelu w trakcie treningu, co pozwala monitorować jego wydajność na nieznanych danych. verbose=2 oznacza, że będą wyświetlane szczegółowe informacje o procesie treningu.

### Ocena modelu

```
# Ocena modelu
loss, accuracy = model.evaluate(X_test, y_test)
print(f'Test Loss: {loss}')
print(f'Test Accuracy: {accuracy}')
```

*5.9 Listing - Ocena modelu*

Model jest oceniany na zestawie testowym, a następnie drukowane są wyniki straty (loss) i dokładności (accuracy) na danych testowych.

# 6 Drzewo decyzyjne

## Teoria

Drzewo decyzyjne to popularny algorytm uczenia maszynowego stosowany w zadaniach klasyfikacji i regresji. Jego podstawową ideą jest dekompozycja problemu na mniejsze, bardziej zarządzalne fragmenty poprzez sekwencyjne podejmowanie decyzji, co przypomina strukturę drzewa. Drzewa decyzyjne są łatwe do interpretacji i wizualizacji, co czyni je użytecznym narzędziem w eksploracyjnej analizie danych oraz w podejmowaniu decyzji.

Drzewo decyzyjne rozpoczyna działanie od korzenia, który reprezentuje całą przestrzeń cech i wszystkie dostępne dane. Następnie, poprzez serię pytań lub podziałów, dane są dzielone na mniejsze podzbiory, które są reprezentowane przez węzły drzewa. Każdy węzeł odpowiada pewnemu warunkowi nałożonemu na jedną z cech danych, a krawędzie wychodzące z tego węzła reprezentują możliwe odpowiedzi na to pytanie (np. "tak" lub "nie"). Proces ten jest rekurencyjnie powtarzany, tworząc kolejne poziomy drzewa, aż do osiągnięcia liści. Liście drzewa reprezentują ostateczne decyzje lub przewidywania algorytmu.

Konstrukcja drzewa decyzyjnego opiera się na kryteriach podziału, które mierzą, jak dobrze dany podział danych w danym węźle poprawia homogeniczność wynikowych podzbiorów. Dla problemów klasyfikacji, często stosowanymi miarami są entropia i zysk informacyjny, które oceniają zmniejszenie niepewności w podziałach. W problemach regresji, popularną miarą jest z kolei błąd średniokwadratowy (MSE), który ocenia różnice między przewidywanymi a rzeczywistymi wartościami.

Drzewo decyzyjne rośnie poprzez iteracyjne wybieranie cech i progów podziału, które maksymalizują poprawę kryterium podziału. Proces ten jest kontynuowany, dopóki nie zostaną spełnione określone warunki zakończenia, takie jak minimalna liczba przykładów w liściu, maksymalna głębokość drzewa lub brak dalszej poprawy jakości podziału.

Jedną z głównych zalet drzew decyzyjnych jest ich interpretowalność. Struktura drzewa przypomina sposób podejmowania decyzji przez człowieka, co ułatwia zrozumienie i interpretację modelu przez użytkowników. Dodatkowo, drzewa decyzyjne są w stanie radzić sobie z danymi o mieszanych typach (kategoryczne i numeryczne) oraz nie wymagają specjalnej normalizacji lub skalowania cech.

Jednak drzewa decyzyjne mają także pewne wady. Są podatne na przetrenowanie, szczególnie w przypadku głębokich drzew, które mogą zbytnio dopasować się do szumów w danych treningowych. Przetrenowanie można ograniczyć poprzez zastosowanie technik przycinania (pruning), które polegają na usuwaniu zbędnych węzłów z drzewa po jego wstępnej budowie. Ponadto, drzewa decyzyjne mogą być niestabilne, co oznacza, że niewielkie zmiany w danych treningowych mogą prowadzić do znacznych zmian w strukturze drzewa.

Aby przezwyciężyć te ograniczenia, często stosuje się zespoły drzew decyzyjnych, takie jak lasy losowe (random forests) i gradient boosting. Metody te łączą wyniki wielu drzew decyzyjnych, co poprawia ogólną wydajność modelu i zmniejsza ryzyko przetrenowania.

## Implementacja uczenia sieci neuronowej

```
import numpy as np
import pandas as pd
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score

# Załadowanie danych
X_train = pd.read_csv('X_train.csv').values
X_test = pd.read_csv('X_test.csv').values
y_train = pd.read_csv('y_train.csv').values
y_test = pd.read_csv('y_test.csv').values

# Normalizacja danych
X_train = X_train / np.max(X_train)
X_test = X_test / np.max(X_test)

# Upewnienie się, że y_train i y_test są jednowymiarowe
y_train = np.squeeze(y_train)
y_test = np.squeeze(y_test)

# Inicjalizacja modelu drzewa decyzyjnego
tree = DecisionTreeClassifier(max_depth=5)

# Trenowanie modelu
tree.fit(X_train, y_train)

# Predykcja na zbiorze testowym
y_pred = tree.predict(X_test)

# Obliczenie dokładności
accuracy = accuracy_score(y_test, y_pred)
print(f'Test Accuracy: {accuracy}')
```

*6.1 Listing - Ocena modelu*

## Omówienie działania kodu

### Importowanie bibliotek

```
import numpy as np
import pandas as pd
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score
```

*6.2 Listing – Importowanie bibliotek*

- DecisionTreeClassifier ze scikit-learn do tworzenia modelu drzewa decyzyjnego.
- accuracy_score ze scikit-learn do obliczenia dokładności modelu.

### Normalizacja danych

```
# Normalizacja danych
X_train = X_train / np.max(X_train)
X_test = X_test / np.max(X_test)
```

*6.3 Listing – Normalizacja danych*

Normalizuje dane wejściowe (X_train i X_test), dzieląc je przez maksymalną wartość w zbiorze treningowym. To skaluje wartości danych do zakresu [0, 1], co może poprawić wydajność modelu, szczególnie jeśli dane mają różne zakresy wartości.

### Inicjalizacja modelu drzewa decyzyjnego

```
# Inicjalizacja modelu drzewa decyzyjnego
tree = DecisionTreeClassifier(max_depth=5)
```

*6.4 Listing – Inicjalizacja modelu drzewa decyzyjnego*

Inicjalizuje model drzewa decyzyjnego z maksymalną głębokością drzewa ustawioną na 5. Ograniczenie głębokości drzewa decyzyjnego może zapobiec przeuczeniu (overfitting), szczególnie na małych zestawach danych.

### Trenowanie modelu

```
# Trenowanie modelu
tree.fit(X_train, y_train)
```

*6.5 Listing - Trenowanie modelu*

Trenuje model drzewa decyzyjnego na danych treningowych (X_train, y_train). Model uczy się reguł decyzyjnych, aby jak najlepiej klasyfikować dane na podstawie cech wejściowych.

### Predykcja na zbiorze testowym

```
# Predykcja na zbiorze testowym
y_pred = tree.predict(X_test)
```

*6.6 Listing – Predykcja na zbiorze testowym*

Model predykuje etykiety klas dla danych testowych (X_test). Wynik (y_pred) jest wektorem przewidywanych etykiet dla każdej próbki w zbiorze testowym.

### Obliczenie dokładności

```
# Obliczenie dokładności
accuracy = accuracy_score(y_test, y_pred)
print(f'Test Accuracy: {accuracy}')
```

*6.7 Listing – Obliczenie dokładności*

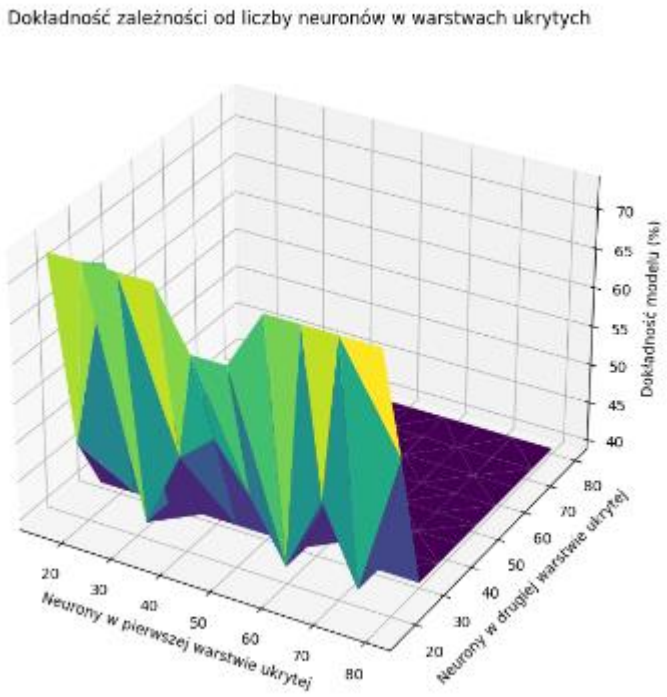
Oblicza dokładność modelu na danych testowych, porównując przewidywane etykiety (y_pred) z rzeczywistymi etykietami (y_test). Funkcja accuracy_score zwraca stosunek liczby poprawnych przewidywań do całkowitej liczby próbek. Dokładność jest następnie drukowana.

# 7 Badania – algorytm wstecznej propagacji

Do zbadania algorytmu wstecznej propagacji wykonane zostały dwa eksperymenty. Pierwszy z nich będzie dotyczył badania dokładności zależności od liczby neuronów. Z racji tego, że mamy dwie warstwy ukryte to wykres będzie w 3D. Drugi z nich będzie dotyczył dokładności modelu w zależności od współczynnika uczenia. Przyjęta tam będzie liczba neuronów w obydwu warstwach taka sama, jak podczas uczenia sieci, czyli 64 neurony w pierwszej wartswie ukrytej i 32 neurony w drugiej warstwie ukrytej.

## 7.1 Eksperyment I – dokładność zależności od liczby neuronów w wartstwach ukrytych

Zainicjalizowałem tablicę obrazującą generującą dwie listy neuron_counts_1 i neuron_counts_2, które zawierają 10 równomiernie rozmieszczonych wartości całkowitych w zakresie od 16 do 80. Do badań przyjąłem learning_rate = 0.01 oraz liczbę epok równą 2000.




**Kluczowe obserwacje**

1. **Wartości dokładności**:
    - **Fioletowe obszary**: Reprezentują najniższe wartości dokładności modelu. Sugeruje to, że dla niektórych kombinacji liczby neuronów w warstwach ukrytych model działa bardzo słabo.
    - **Zielone i żółte obszary**: Reprezentują wyższe wartości dokładności modelu. W tych obszarach model działa lepiej.

2. **Wpływ liczby neuronów w pierwszej warstwie ukrytej**:
    - Zmiana liczby neuronów w pierwszej warstwie ukrytej wpływa na dokładność modelu. Widać pewne kombinacje, gdzie zmniejszenie lub zwiększenie liczby neuronów prowadzi do spadku dokładności.
3. **Wpływ liczby neuronów w drugiej warstwie ukrytej**:
    - Podobnie jak w przypadku pierwszej warstwy, zmiana liczby neuronów w drugiej warstwie ukrytej ma wpływ na dokładność. Wartości dokładności różnią się w zależności od liczby neuronów.

## 7.2 Eksperyment II – dokładność modelu w zależności od współczynnika uczenia

Przyjąłem tutaj dwie warstwy w warstwie ukrytej. Jedna posiada 64 neurony a druga posiada 32 neurony. Liczba epok jest równa 1000. Wygenerowałem tutaj listę 30 wartości współczynnika uczenia się (learning rate) równomiernie rozmieszczonych na skali logarytmicznej od $10^{-4}$ do $10^{0}$ (czyli 1).

**Obserwacje**

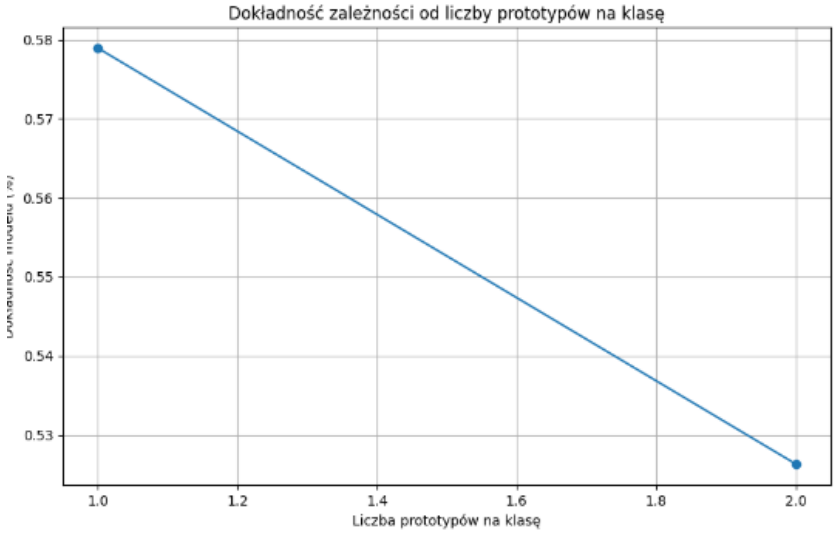
1. **Dokładność dla małych współczynników uczenia**:
    - Dla bardzo małych wartości współczynnika uczenia (np. $10^{-4}$ do $10^{-2}$), dokładność modelu jest stosunkowo niska i wynosi około 40%. To może sugerować, że model nie jest w stanie odpowiednio się nauczyć, ponieważ zmiany wag są zbyt małe, aby zrobić znaczący postęp w nauce.
2. **Dokładność dla średnich współczynników uczenia**:
    - Przy współczynnikach uczenia rzędu $10^{-1}$, dokładność modelu zaczyna się poprawiać, osiągając wyższe wartości. Widzimy kilka pików, co wskazuje, że dla niektórych wartości współczynnika uczenia model uczy się lepiej.
    - Najwyższa dokładność jest osiągana dla wartości współczynnika uczenia w zakresie od $10^{-1}$ do $10^{0}$ (0.1 do 1).
3. **Dokładność dla dużych współczynników uczenia**:
    - Dla bardzo dużych wartości współczynnika uczenia (np. $10^{0}$), dokładność modelu spada. Może to sugerować, że model uczy się zbyt agresywnie, co prowadzi do przeskoków nad optymalnymi rozwiązaniami (zjawisko przeuczenia).

# 8 Badania – algorytm LVQ

W ramach pracy z algorytmem LVQ będą przeprowadzone dwa eksperymenty. Obydwa z nich będą wizualizowane na wykresie 2D. Pierwszy z nich będzie dotyczył zależności od liczby prototypów na klasę, a drugi dokładności zależności od współczynnika uczenia.

## 8.1 Eksperyment I – zależność od liczby prototypów na klasę

Do tego eksperymentu zostały przetestowane zależności od liczby prototypów w wartościach od 1 do 5 liczby prototypów na klasę.

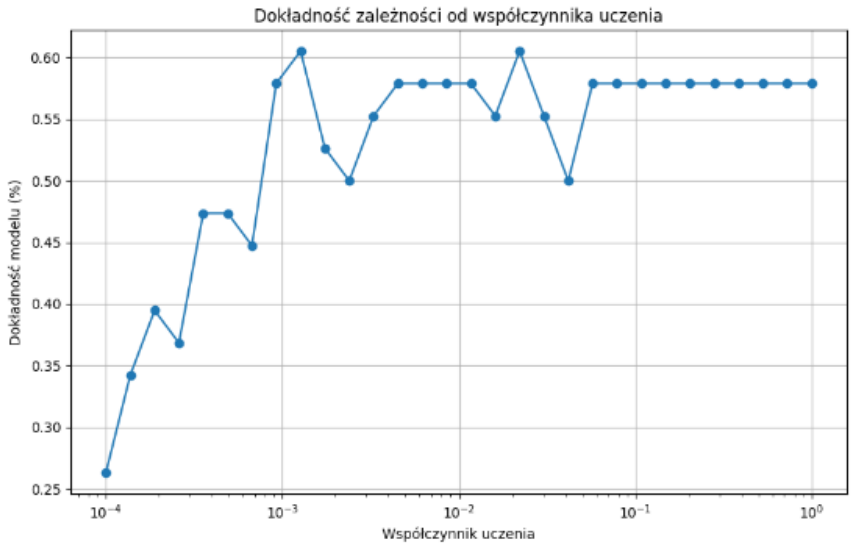


**Obserwacje**

1. **Trend malejący**:
    - Wykres pokazuje, że wraz ze wzrostem liczby prototypów na klasę, dokładność modelu LVQ spada.
    - Dokładność zaczyna się od około 0.58 (58%) dla jednego prototypu na klasę i spada do około 0.53 (53%) dla dwóch prototypów na klasę.

## 8.2 Eksperyment II – dokładność zależności od współczynnika uczenia

Do tego eksperymentu zostały ustawione współczynniki uczenia w wartościach od $10^{-4}$ do $10^{0}$.




**Obserwacje**

1. **Niskie współczynniki uczenia ($10^{-4}$ do $10^{-3}$)**:
    - Dokładność modelu jest stosunkowo niska dla bardzo małych wartości współczynnika uczenia. Dokładność w tym zakresie wynosi około 0.25 do 0.45. Może to sugerować, że model uczy się zbyt wolno i nie osiąga optymalnego rozwiązania w zadanej liczbie epok.
2. **Średnie współczynniki uczenia ($10^{-3}$ do $10^{-2}$)**:
    - Dokładność modelu znacząco wzrasta w tym zakresie, osiągając wartość około 0.6 (60%). Jest to punkt, w którym model osiąga swoją najlepszą wydajność. Wartości współczynnika uczenia w tym zakresie wydają się być optymalne dla tego modelu i danych.
3. **Wysokie współczynniki uczenia ($10^{-2}$ do $10^{0}$)**:
    - Dokładność modelu stabilizuje się w zakresie około 0.55 do 0.6, ale wykazuje pewne fluktuacje. Chociaż model nadal osiąga stosunkowo dobrą wydajność, nie ma już znaczącego wzrostu dokładności, a w niektórych przypadkach nawet lekki spadek.

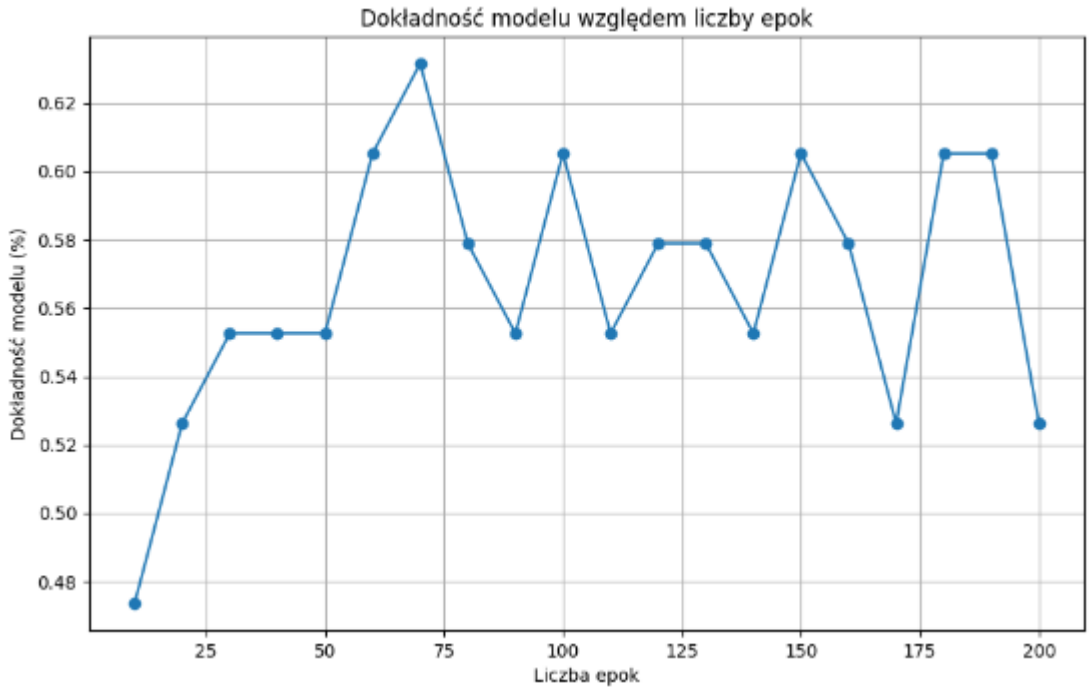
- Najwyższe współczynniki uczenia (powyżej $10^{-1}$) prowadzą do stabilizacji dokładności na poziomie około 0.55, co może sugerować, że model uczy się zbyt agresywnie, co prowadzi do oscylacji i braku konwergencji do optymalnego rozwiązania.

# 9 Badania – sieć LSTM

W ramach uczenia sieci neuronowej Long Short-Term Memory zostaną przeprowadzone dwa eksperymenty. Obydwie wizualizacje wyników zostaną przeprowadzone na wykresach 2D. Pierwszy eksperyment będzie polegał na badaniu dokładności modelu względem liczby epok, a drugi z nich będzie dotyczył dokładności modelu względem batch size.

## 9.1 Eksperyment I – dokładność modelu względem liczby epok

Do tego eksperymentu zostało przyjęte 20 losowych wartości uporządkowanych rosnąco.



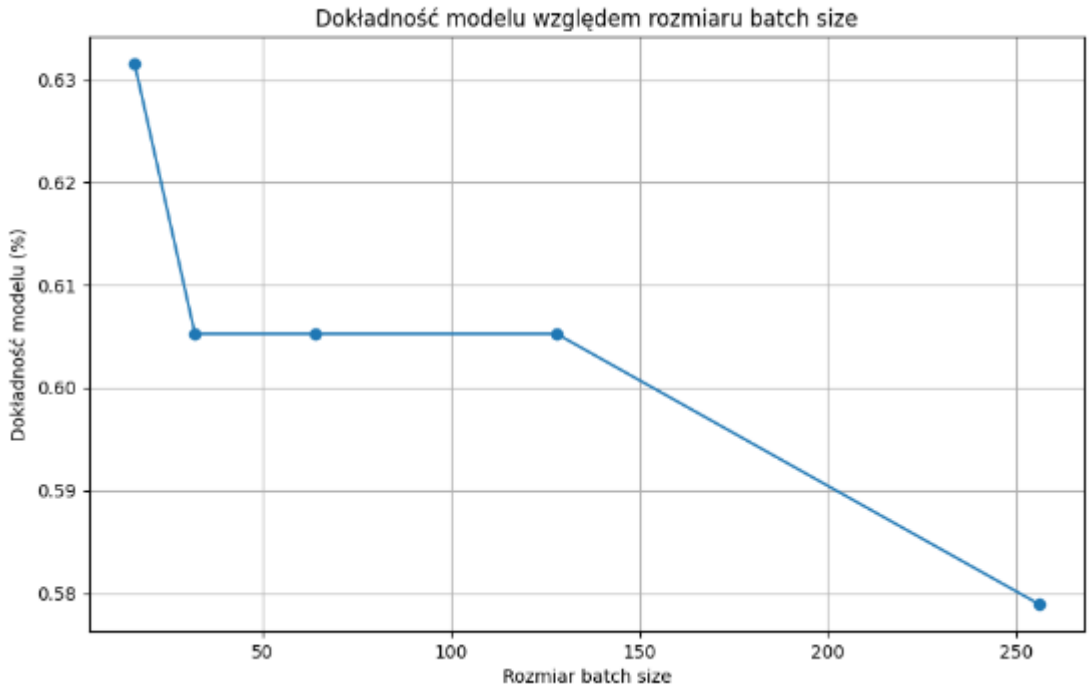

## Obserwacje

1. **Początkowy wzrost dokładności**: W pierwszych 50 epokach można zaobserwować znaczny wzrost dokładności modelu, co sugeruje, że model szybko uczy się i poprawia swoje wyniki na początku treningu.
2. **Wahania dokładności**: Po osiągnięciu około 75 epok, dokładność zaczyna wykazywać duże wahania. Jest to oznaka niestabilności w procesie uczenia, która może wynikać z różnych czynników, takich jak zbyt duża wartość współczynnika uczenia, brak regularizacji lub nadmierne przetrenowanie na pewnych etapach.
3. **Brak wyraźnego trendu wzrostowego po 50 epokach**: Po osiągnięciu szczytu w okolicach 75 epok, nie ma wyraźnego trendu wzrostowego. Dokładność oscyluje wokół różnych wartości, ale nie wykazuje stałej poprawy ani spadku.

4. **Najwyższa dokładność**: Najwyższa wartość dokładności na wykresie wydaje się być osiągnięta w okolicach 75 epok, gdzie dokładność przekracza 62%. W dalszych epokach, dokładność nie osiąga ponownie tego poziomu.
5. **Potencjalne przetrenowanie**: Wahania dokładności po około 75 epokach mogą sugerować, że model zaczyna się przetrenowywać, co oznacza, że dobrze dopasowuje się do danych treningowych, ale traci zdolność do generalizacji na dane testowe.

## 9.2 Eksperyment II – dokładność modelu względem rozmiaru batch size

W tym eksperymencie zostało przyjęte pięć wartości batch size(16, 32, 64, 128, 256).




**Obserwacje z wykresu dotyczącego wpływu rozmiaru batch size na dokładność modelu LSTM**

1. **Najwyższa dokładność przy najmniejszym batch size (16)**:
    - Najwyższa dokładność modelu (około 63%) została osiągnięta przy najmniejszym rozmiarze batch size, który wynosi 16.
    - Sugeruje to, że mniejsze wartości batch size mogą prowadzić do lepszego dostosowania modelu do danych treningowych.
2. **Spadek dokładności przy większych wartościach batch size**:
    - Widać wyraźny spadek dokładności wraz ze wzrostem rozmiaru batch size.
    - Dokładność modelu spada do około 60% przy batch size wynoszącym 64 i jeszcze bardziej spada do około 58% przy batch size wynoszącym 256.
    - Większe batch size powodują rzadsze aktualizacje wag, co może prowadzić do gorszego dostosowania modelu do danych

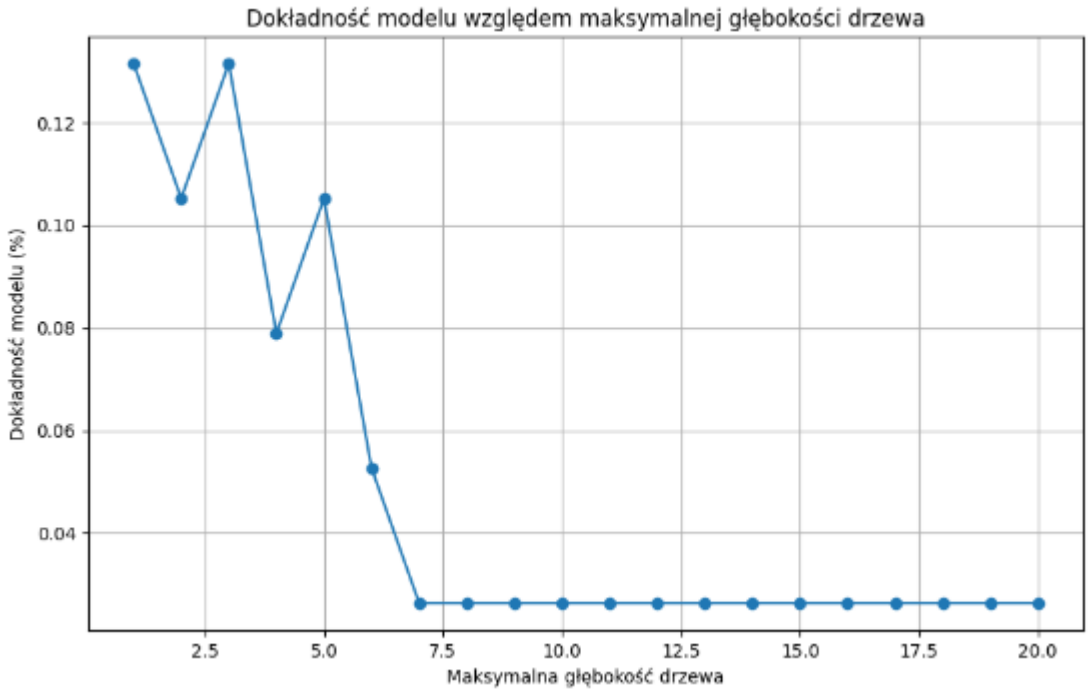
3. **Stabilizacja dokładności przy średnich wartościach batch size (32, 64)**:
    - Dokładność modelu jest stosunkowo stabilna przy batch size wynoszącym 32 i 64, gdzie dokładność wynosi około 61%.
    - Sugeruje to, że wartości batch size w tym przedziale mogą być wystarczające do osiągnięcia przyzwoitej dokładności bez znacznego spadku wydajności.

# 10 Badania – drzewo decyzyjne

Praca nad algorytmem drzewa decyzyjnego tak jak poprzednie algorytmy, będzie skupiona na dwóch eksperymentach i również obydwie wziualizacje wyników odbędą się na wykresie 2D. Jako pierwszy eksperment zostanie zbadana dokładność modelu względem maksymalnej głębokości drzewa. Drugi eksperyment będzie skupiony na analizie dokładności modelu względem minimalnej liczby próbek w liściu.

## 10.1 Eksperyment I – dokładność modelu względem maksymalnej głębokości drzewa

Do tego eksperymentu została zainicjalizowana tablica z liczbami naturalnymi od 1 do 20, która przechowywała informacje na temat maksymalnej głębokości drzewa.



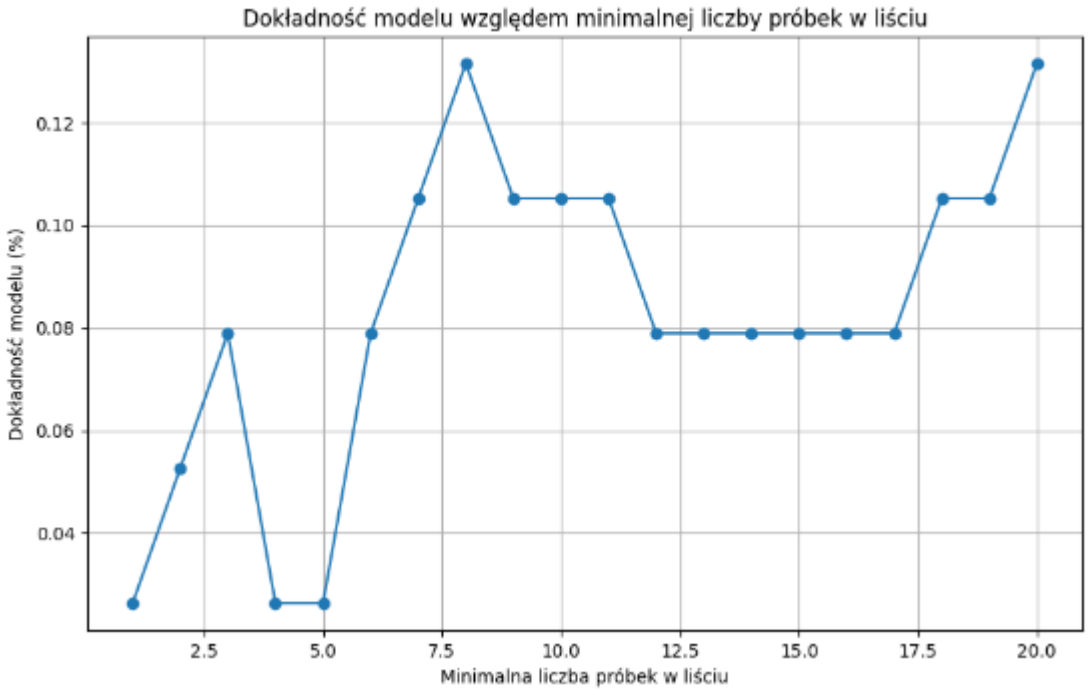

**Obserwacje**

1. **Najlepsza dokładność przy małej głębokości**:
    - Dokładność modelu jest najwyższa przy bardzo małych wartościach maksymalnej głębokości drzewa (1-3).
    - Po osiągnięciu pewnego maksimum (ok. 0.12), dokładność zaczyna spadać.

2. **Spadek dokładności przy większej głębokości**:
    - Dokładność modelu drastycznie spada dla wartości **max_depth** większych niż 5, co wskazuje na przeuczenie modelu. Przeuczenie (overfitting) oznacza, że model jest zbyt skomplikowany i dopasowuje się do szumów w danych treningowych, co skutkuje gorszą generalizacją na danych testowych.
    - Wartości głębokości drzewa powyżej 5 prowadzą do bardzo niskiej dokładności, co sugeruje, że drzewo staje się zbyt skomplikowane i traci zdolność do poprawnej klasyfikacji danych testowych.
3. **Stabilizacja dokładności przy większych głębokościach**:
    - Po osiągnięciu głębokości 7, dokładność modelu stabilizuje się na bardzo niskim poziomie, co sugeruje, że dalsze zwiększanie głębokości drzewa nie poprawia jego wydajności i utrzymuje model w stanie przeuczenia.

## 10.2 Eksperyment II – dokładność modelu względem minimalnej liczby próbek w liściu

W tym eksperymencie została zainicjalizowana tablica z liczbami naturalnymi od 1 do 20, która przechowywała informacje na temat minimalnej liczby próbek w liściu.



**Obserwacje**

1. **Niska dokładność przy bardzo małej liczbie próbek w liściu**:
    - Dla bardzo małych wartości **min_samples_leaf** (1-2), dokładność modelu jest niska. Może to sugerować, że model jest zbyt złożony i dopasowuje się do szumów w danych treningowych, co prowadzi do przeuczenia (overfitting).
2. **Optymalna wartość minimalnej liczby próbek w liściu**:
    - Dokładność modelu osiąga maksimum (około 0.13) dla wartości **min_samples_leaf** równej 7. W tym punkcie model prawdopodobnie jest wystarczająco złożony, aby dobrze dopasować się do danych, ale jednocześnie na tyle prosty, aby uniknąć przeuczenia.
    - Po osiągnięciu maksimum dla wartości 7, dokładność spada, a następnie ponownie wzrasta dla wartości 20, co może sugerować, że wartość **min_samples_leaf** równa 20 również może być korzystna w kontekście tego zestawu danych.
3. **Wahania dokładności**:
    - Wykres pokazuje pewne wahania dokładności dla różnych wartości **min_samples_leaf**. Nie ma jednoznacznego trendu, co sugeruje, że wpływ **min_samples_leaf** może być silnie zależny od charakterystyki danych.

# 11 Podsumowanie i wnioski

Podsumowanie i wnioski projektu wskazują na złożoność i wszechstronność przeprowadzonych badań nad algorytmami klasyfikacyjnymi. W trakcie realizacji projektu zbadano trzy różne algorytmy: wstecznej propagacji błędu LVQ, sieci LSTM oraz drzewa decyzyjnego, aby porównać ich skuteczność w klasyfikacji danych dotyczących płazów. Każdy z tych algorytmów ma swoje specyficzne zastosowania i ograniczenia, które zostały szczegółowo przeanalizowane.

Algorytm LVQ okazał się skuteczny w tworzeniu wyraźnych granic decyzji, co jest korzystne w przypadku dobrze zdefiniowanych i oddzielnych klas. Jednakże, jego wydajność była wrażliwa na liczbę prototypów i wartości początkowe współczynnika uczenia. Optymalne ustawienie tych parametrów było kluczowe dla uzyskania wysokiej dokładności klasyfikacji, co wymagało dokładnych eksperymentów.

Sieci LSTM, dzięki swojej zdolności do uczenia się zarówno krótkoterminowych, jak i długoterminowych zależności, sprawdziły się dobrze w przetwarzaniu sekwencyjnych danych. Badania wykazały, że odpowiedni dobór liczby epok i rozmiaru batch size miał istotny wpływ na dokładność modelu. Najwyższą dokładność uzyskano przy mniejszym rozmiarze batch size, co sugeruje, że częstsze aktualizacje wag mogą prowadzić do lepszego dostosowania modelu do danych treningowych.

Drzewa decyzyjne, z kolei, były łatwe do interpretacji i wizualizacji, co czyni je użytecznymi narzędziami w eksploracyjnej analizie danych. Eksperymenty wykazały, że drzewo decyzyjne osiągało najlepsze wyniki przy małej głębokości drzewa, a zbyt duża głębokość prowadziła do przetrenowania modelu. Również minimalna liczba próbek w liściu miała znaczący wpływ na dokładność modelu, co sugeruje konieczność balansowania złożoności modelu w celu uniknięcia przeuczenia.

Ogólnie rzecz biorąc, każdy z badanych algorytmów ma swoje mocne i słabe strony, a wybór odpowiedniego algorytmu zależy od specyfiki danych oraz celów analizy. Przeprowadzone eksperymenty dostarczyły cennych informacji na temat parametryzacji i optymalizacji tych algorytmów, co może być wykorzystane w przyszłych badaniach i zastosowaniach praktycznych w dziedzinie klasyfikacji danych.

# 12 Bibliografia

[1] https://archive.ics.uci.edu/dataset/528/amphibians

[2] https://web.archive.org/web/20160309093826/http://ac-it.pl/algorytm-wstecznej-propagacji-bledow

[3] https://www.geeksforgeeks.org/learning-vector-quantization/

[4] https://www.geeksforgeeks.org/understanding-of-lstm-networks/

[5] https://www.geeksforgeeks.org/decision-tree-algorithms/