

Dokumentacja projektu “System rekomendacji klientów banku”

Autorzy: Paweł Skrok w67266, Karolina Rydzik w67264

Wstęp

System rekomendacji klientów banku został zaprojektowany w celu wsparcia procesu podejmowania decyzji przez instytucje finansowe. Jego głównym zadaniem jest ułatwienie **klasyfikacji klientów** pod kątem **prawdopodobieństwa rozwiązania umowy z bankiem**. System ten opiera się wyłącznie na otwartoźródłowych narzędziach (open source) i nie korzysta z zewnętrznych modeli ani komercyjnych rozwiązań do uczenia maszynowego.

Aplikacja oferuje prosty i intuicyjny interfejs użytkownika, dzięki któremu pracownik banku może wprowadzić dane dotyczące klienta. Po wprowadzeniu informacji model sieci neuronowej przetwarza je, **analizując ryzyko odejścia klienta**. Pierwszym etapem prac nad systemem było przygotowanie odpowiedniego zbioru danych, zawierającego wybrane informacje o klientach, które mogą mieć istotny wpływ na decyzję o rezygnacji z usług banku. Uwzględniono dane demograficzne oraz informacje finansowe, natomiast celowo pominięto wszelkie dane poufne i wrażliwe, takie jak imię czy nazwisko, aby zapewnić ochronę prywatności klientów. W związku z tym wykorzystywane pliki CSV nie zawierają danych osobowych.

Model działa w oparciu o klasyfikację binarną - przewiduje, czy klient pozostanie, czy odejdzie. Wynik modelu podawany jest w formie procentowej, określającej prawdopodobieństwo odejścia klienta. Na przykład wynik 34% oznacza, że klient ma 34% szans na rozwiązanie umowy w najbliższym czasie. Im wartość wyniku jest bliższa 1 (czyli 100%), tym większe ryzyko odejścia. Takie informacje mogą być wykorzystane przez bank do podejmowania działań prewencyjnych, takich jak oferta dodatkowych benefitów mających na celu zatrzymanie klienta.

Opis technologii

W projekcie zastosowano szereg nowoczesnych technologii open source, głównie w języku Python:

- **Tkinter** – biblioteka do tworzenia graficznego interfejsu użytkownika (GUI), umożliwiająca intuicyjną obsługę aplikacji.
- **Pandas** – narzędzie do zaawansowanego przetwarzania i analizy danych, wykorzystywane do manipulacji zbiorami danych oraz podziału na zbiory treningowe i testowe.

- **NumPy** – biblioteka zapewniająca szybkie i efektywne operacje numeryczne na macierzach i tablicach danych.
- **Matplotlib** – służy do wizualizacji danych i monitorowania procesu uczenia modelu.
- **JSON** – wykorzystywany do zapisu i odczytu konfiguracji oraz parametrów transformacji danych.
- **Logging** – moduł do logowania zdarzeń i pomiaru wydajności funkcji w trakcie działania aplikacji.

Całość systemu została zaprojektowana w sposób modularny i obiektowy, co daje możliwość rozwoju i umożliwia modyfikację kodu. Wprowadzono również mechanizmy logowania i pomiaru czasu wykonywania funkcji przy użyciu dekoratorów, a w warstwie sieci neuronowej zastosowano metaklasę umożliwiającą dynamiczne inicjowanie parametrów w zależności od wybranego algorytmu optymalizacji.

Opis modelu i architektury sieci neuronowej

Model predykcyjny oparty jest na podejściu opisanym w artykule "[A Step by Step Backpropagation Example](#)" (link <https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/>).

Dane wejściowe pochodzą z repozytorium zawierającego zbiór informacji o klientach banku (link: <https://github.com/Yorko/mlcourse.ai/blob/main/data/bank.csv>).

Sieć neuronowa składa się z **trzech warstw**: dwóch ukrytych (z 12 neuronami każda) oraz warstwy wyjściowej z pojedynczym neuronem binarnym. Model realizuje klasyfikację binarną, ucząc się za pomocą mechanizmu propagacji wstecznej (backpropagation). Architektura jest elastyczna i umożliwia implementację różnych technik optymalizacji, takich jak Adam, AdaDelta czy Momentum oraz uczenie wsadowe (batch learning).

Diagram klas

Poniższy diagram przedstawia strukturę klas zastosowanych w systemie rekomendacji. Ukazuje wzajemne zależności między głównymi komponentami oraz sposób organizacji kodu w podejściu obiektowym.



1. Przygotowanie danych

Dane wykorzystane do trenowania modelu pochodzą z dwóch zbiorów CSV, które zostały połączone:

- bank_full.csv
- DataSet_for ANN-checkpoint.csv

Model zawiera 1000 wierszy oraz 15 kolumn opisujących różnorodne cechy klientów, takie jak wiek, wynagrodzenie, status zatrudnienia, informacje demograficzne i finansowe. Zmienna decyzyjna „y” przyjmuje wartości „yes” lub „no”, wskazujące, czy klient odejdzie z banku.

Dane zostały odpowiednio przekształcone — kolumny binarne zamieniono na wartości numeryczne, a kolumny kategoryczne zakodowano za pomocą One-Hot Encoding. Dodatkowo, cechy numeryczne poddano normalizacji lub standaryzacji (w zależności od wybranej metody) w celu zapewnienia efektywnego uczenia modelu.

Atrybuty opisujące

Specyfikacja pól danych wejściowych

- **CreditScore:**
Typ: `int` lub `float`
Zakres: dowolne liczby, z dokładnością do 2 miejsc po przecinku
- **Country:**
Typ: `wybór`
Możliwe wartości: `Spain, Germany, France etc.`
- **Gender:**
Typ: `kategoria`
Możliwe wartości: `Male, Female`
- **Tenure:**
Typ: `int`
Możliwe wartości: `0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10`
- **HasCrCard** (posiadanie karty kredytowej):
Typ: `binarny`
Możliwe wartości: `1` (tak), `0` (nie)
- **IsActiveMember** (czy klient jest aktywnym członkiem):
Typ: `binarny`
Możliwe wartości: `1` (tak), `0` (nie)
- **EstimatedSalary:**
Typ: `int` lub `float`
Zakres: dowolne liczby, z dokładnością do 2 miejsc po przecinku
- **Age:**
Typ: `int`
Zakres wartości: `18+`
- **Job:**
Typ: `kategoria`
Możliwe wartości:
`management, technician, entrepreneur, blue-collar, unknown, retired, admin., services, self-employed, unemployed, housemaid, student`
- **Marital:**
Typ: `kategoria`

Mögliche Werte: married, single, divorced

- **Education:**

Typ: kategoria

Mögliche Werte: tertiary, secondary, unknown, primary

- **Balance:**

Typ: int

Zakres: beliebige ganze Zahlen, auch negativer

- **Loan:**

Typ: kategoria

Mögliche Werte: yes, no

Zmienna decyzyjna:

'y': zmienna decyzyjna posiadająca 2 etykiety „yes”, „no” która wystąpiła dla danego rekordu opisującą decyzję klienta w momencie opisanym pewne powyższe atrybuty decyzja „yes” oznacza opuszczenie banku przez klienta, decyzja „no” oznacza że klient pozostał w banku.

Implementacja i funkcjonalności aplikacji

Aplikacja zawiera klasy i metody do przetwarzania danych, takie jak:

- **multiply** – funkcja augmentująca dane poprzez powielanie i wprowadzanie losowych zmian,
- **Transformations** – klasa realizująca normalizację min-max oraz standaryzację Z-score,
- **SplitData** – podział danych na zbiory treningowe, walidacyjne i testowe,
- **OneHotEncoder** – kodowanie zmiennych kategorycznych do postaci binarnej.

Dzięki zastosowaniu tych narzędzi, dane są przygotowywane w sposób umożliwiający efektywne i poprawne działanie modelu sieci neuronowej.

Opis Klas

1. Metoda Multiply

```
def multiply(data, times, rate):
    """
    :param data is x and y DataFrame with data to multiply
    :param times defines how many rows like given are created
    :param rate random + - percentage threshold of a current point
    :return [data] + created_data X [times] with [rate]% of threshold in pd.DataFrame"""
    return for_pdframe_data
```

Przykład wykorzystania na danych w pd.DataFrame powielony 4 razy z maksymalnie +/- 50% zmianą wartości liczby w każdej kolumnie w wierszu.

```
from Multiple_points import multiply
data = pd.DataFrame([[1,2,3,0]],columns=["a","b","c","y"])
print("data before: ",data.values)
print("data after multiply 4 times with error 0.5")

xmolt = multiply(data,4,0.5)
print(xmult.values)
```

```
data before:  [[1 2 3 0]]
data after multiply 4 times with error 0.5
[[1.          2.          3.          0.          ]
 [1.23372798  1.40525975  1.7240945   0.        ]
 [0.66998655  1.59926913  2.62213266  0.        ]
 [0.61513526  2.97355579  2.865465    0.        ]
 [0.65698411  2.94836783  1.99622415  0.        ]]
```

Obiekty na danych które mają zmianę wartości o 0 %, **czyli rate = 0**.

2. Klasa Transformations

```
data before: [[1 2 3 0]]  
data after multiply 4 times without error  
[[1. 2. 3. 0.]  
 [1. 2. 3. 0.]  
 [1. 2. 3. 0.]  
 [1. 2. 3. 0.]  
 [1. 2. 3. 0.]]
```

Przyjmuje dane w postaci **obiektu pd.DataFrame** oraz zdefiniowany przy pomocy obiektu **Enumerable StandardizationType** rodzaju standaryzacji który jest stosowany.

```
class StandardizationType(Enum):  
    """  
    Attributes:  
    - MEAN_SCORE: Standardization by centering data around the mean (mean centering).  
    - Z_SCORE: Z-score standardization (subtract mean, divide by standard deviation).  
    - LOG_SCALING: Logarithmic scaling to reduce skewness or compress large values.  
    - NORMALIZATION: Rescales data to a fixed range, typically [0, 1].  
    """  
    MEAN_SCORE = "mean_score"  
    Z_SCORE = "standaryzacja_z_score"  
    LOG_SCALING = "log_scaling"  
    NORMALIZATION = "normalization"
```

Następnie wywołujemy metodę **klasy standarization_of_data** przekazując obiekt **DataFrame** która transformuje obiekty zgodnie z wytycznymi przyjętymi przez dany algorytm standaryzacji i zwraca obiekt **typu DataFrame**.

```
def standarization_of_data(self,data)->pd.core.frame.DataFrame:  
    """  
    :param data: sequence in pd.DataFrame  
    :return new_Data pd.DataFrame  
    """  
    self.data=data  
    self.minimums = data.min().values.tolist()  
    self.maximums = data.max().values.tolist()  
    new_data = pd.DataFrame(columns= data.keys())  
    klucze = list(data.keys())  
  
    if self.std_type.value=="normalization":  
        for i, (MIN,MAX) in enumerate(zip(self.minimums,self.maximums)):  
            list_containing_column_values = data.iloc[:,i].values.tolist()
```

Normalizacja **min-max** (zwykle nazywana skalowaniem cech) wykonuje liniową transformację oryginalnych danych na zakres **np. 0,1**.

Wzór na osiągnięcie tego jest następujący: (Normalizacja min-max zachowuje relacje między oryginalnymi wartościami danych).

Wzór:

$$x_{norm} = \frac{x - x_{min}}{x_{max} - x_{min}}$$

Gdzie:

- x — pojedyncza wartość,
- x_{min} — najmniejsza wartość w zbiorze,
- x_{max} — największa wartość w zbiorze,
- x_{norm} — wartość po normalizacji (w zakresie [0, 1]).

Poniżej przedstawiony jest przykład zastosowania funkcji wykorzystującej **typ standaryzacji poprzez normalizację min max** i 5 przykładowych rekordów, a jeszcze niżej - obiekt który został przeskalowany.

	CreditScore	Tenure	IsActiveMember	EstimatedSalary	age	balance
0	619	2	1	101348.88	58	6429
1	608	1	1	112542.58	44	29
2	502	8	0	113931.57	33	2
3	699	1	0	93826.63	47	4518
4	850	2	1	79084.10	33	1

	CreditScore	Tenure	IsActiveMember	EstimatedSalary	age	balance
0	0.512658	0.2	1.0	0.506524	0.923077	0.041690
1	0.489451	0.1	1.0	0.562674	0.564103	0.005443
2	0.265823	0.8	0.0	0.569642	0.282051	0.005290
3	0.681435	0.1	0.0	0.468791	0.641026	0.030867
4	1.000000	0.2	1.0	0.394840	0.282051	0.005284

Klasa Transformations posiada metodę, która wykonuje transformację pojedynczej obserwacji na podstawie wcześniejszej zapisanych w polach klasy parametrów.

- W przypadku **transformacji min-max**, w polach klasy przechowywane są minimalne i maksymalne wartości dla każdej kolumny. Te wartości służą do skalowania nowej danej.
- W przypadku **standaryzacji z-score**, w polach klasy zapisana jest średnia oraz odchylenie standardowe dla każdej kolumny. Te parametry są wykorzystywane do przeskalowania pojedynczej obserwacji.

```
def standarization_one_point(self, point: list) -> list:
    """
    :param point: apply normalization with saved data for normalization
    :return: ***
    if not (isinstance(point, list)):
        raise "element podany do zbioru musi być listą"

    if self.std_type == "normalization":
        assert len(point) == len(self.minimums) == len(self.maximums), f"data {len(point)} == {len(self.minimums)}"
        return [round((x - MIN) / (MAX - MIN), 6) for x, MIN, MAX in zip(point, self.minimums, self.maximums)]

    elif self.std_type == "mean_score":
        assert len(point) == len(self.minimums) == len(
            self.maximums) == len(self.srednie), f"data {len(point)} == {len(self.minimums)} == {len(self.maximums)}"
        return [(x - srednia_kolumny) / (MAX - MIN) for x, srednia_kolumny, MIN, MAX in zip(point, self.srednie, self.minimums, self.maximums)]

    elif self.std_type == "standaryzacja_z_score":
        assert len(point) == len(self.srednie) == len(self.minimums) == len(self.maximums)
        return [(x - srednia_kolumny) / odchylenie_kolumny for x, srednia_kolumny, odchylenie_kolumny, MIN, MAX in zip(point, self.srednie, self.minimums, self.maximums, self.odchylenie)]
    
```

Dane można zapisać w **formacie JSON** w dowolnej lokalizacji oraz wczytać je z tego pliku:

- dzięki temu możliwe jest utworzenie instancji klasy **Transformations** i automatyczne załadowanie zapisanych wartości podczas inicjalizacji.
- metoda ta automatycznie rozpoznaje, jaki typ standaryzacji (**StandardizationType**) został zastosowany.
- dodatkowo istnieje możliwość zdefiniowania ścieżki, pod którą plik JSON zostanie zapisany.

```
from Data.Transformers import StandardizationType, Transformations
# std zawiera informacje które są wykorzystywane przy transformacji punktu
std = Transformations(data_containing_required_columns, StandardizationType.Z_SCORE)
trained_data = std.standarization_of_data(data_containing_required_columns)
## save data
std.save_data() # file_path we can define path to set

new_std = Transformations.load_data() # we need to give path we used to save_data
return trained_data
```

Wykorzystanie Funkcjonalności:

Dane są przechowywane w plikach **.csv**. Do otwierania plików służy **funkcja load_data**, która przyjmuje bezwzględną ścieżkę do pliku.

Plik jest wczytywany za pomocą **biblioteki Pandas**, która tworzy **obiekt DataFrame** z odczytanych danych. Dla poprawnego rozdzielenia kolumn, jako **delimiter** (separator) używany jest **znak średnika** ;

```
1 usage new *
def load_data(file_path=r"C:\Program Files\Pulpit\Data_science\Gui_szkolenie_4\TrainData\new_data.csv"):
    merged_data = pd.read_csv(file_path,
                               delimiter=";")
    #for key in merged_data.columns:
    #    print(merged_data[key].unique())
    return merged_data
```

Po wczytaniu danych następuje ich przetworzenie: - jeśli kolumna zawiera wartości numeryczne **0 i 1** zapisane jako **stringi** (czyli teksty, a nie liczby typu int czy float), są one **konwertowane na wartości liczbowe**.

W przypadku kolumn zawierających dwie kategorie, np. **loan (yes, no)** lub **gender (male, female)**, wartości te są zamieniane na liczby **0 i 1**.

Kolejnym krokiem jest podział kolumn na dwie grupy, które poddane zostaną różnym transformacjom:

- **col1_one_hot** — to kolumny, które zostaną zakodowane za pomocą *one hot encoder*.
- **col2_norm** — to kolumny, które zostaną poddane standaryzacji.

```
def data_preprocessing():
    merged_data = load_data()

    merged_data["HasCrCard"] = pd.to_numeric(merged_data.loc[:, "HasCrCard"])
    merged_data["loan"] = np.where(merged_data["loan"] == "yes", 1, 0)
    merged_data["Gender"] = np.where(merged_data["Gender"] == "Male", 1, 0)
    #print(pd.unique(merged_data["HasCrCard"]))

    ## dane dla one hot encodera
    col1one_hot = [2, 9, 10, 11]
    # kolumna za pomocą normalizacji danych
    col2_norm = [1, 4, 6, 7, 8, 12]
```

3. Klasa SplitData

Klasa **SplitData** służy do dzielenia danych na zbiory wykorzystywane podczas trenowania sieci neuronowej. Główną funkcją jest **przetasowanie danych** oraz podział na 3 zbiory: **treningowy, testowy i walidacyjny**, które są następnie wykorzystywane w modelu.

```
class SplitData:  
    __dict__ = ["train", "valid", "test"]  
    """  
        A class to split a dataset into training, validation, and test sets.  
        Parameters:  
            train (float): Default is 0.4.  
            valid (float): Default is 0.4.  
            test (float): Default is 0.2.  
            The sum of train, valid, and test must be 1.0."""  
    train = 0.4  
    valid = 0.4  
    test = 0.2
```

Funkcja posiada również metodę tasowania danych typu DataFrame oraz umożliwia konkatenację danych zawierających cechy uczące wraz ze zmiennej decyzyjną.

```
@classmethod  
def tasowanie(cls, data, f= False):  
    """  
        :param data is pd.DataFrame object  
        :param f when is set to False it splits data for x and y  
        :param f is set to True it returns pd.DataFrame"""  
    shuffled_data = data.sample(frac=1).reset_index(drop=True)  
    if f: return shuffled_data  
    return shuffled_data.iloc[:, :-1].values, shuffled_data.loc[:, "y"]  
  
@classmethod  
def merge(cls, x, y) -> pd.core.frame.DataFrame:  
    """  
        :return x and y merged into one dataframe with last column named y"""  
    data = pd.DataFrame(x, columns=[f"x{i}" for i in range(x.shape[1])])  
    data["y"] = y  
    return data
```

4. Klasa OneHotEncoder

Jej zadaniem jest kodowanie danych w sposób zrozumiały dla modelu, ponieważ modele dobrze radzą sobie ze zmiennymi numerycznymi, ale zmienne kategoryczne trzeba odpowiednio przygotować.

Jeżeli zmienna kategoryczna ma charakter porządkowy, można stosować metody kodowania porządkowego, takie jak **Ordinal Encoding** lub **Integer Encoding**.

Jeżeli zmienna kategoryczna nie ma charakteru porządkowego, a narzucenie takiego porządku nie byłoby korzystne dla modelu, stosujemy metody polegające na przekształceniu kategorii do postaci **binarnego wektora** (ang. One-Hot Encoding).

```
class OneHotEncoder:  
    __slots__ = ["decoded_set", "label_code", "number_of_coded_keys", "data_code"]  
    __zdrapiek-jpg *  
    def __init__(self):  
        self.decoded_set = {}  
        self.label_code={}  
        self.number_of_coded_keys=0  
        self.data_code={}
```

Dla każdej kolumny i każdej kategorii muszą zostać ustalone wartości, które będą podstawiane, oraz musi powstać klucz kodowania.

```
def code_keys(self,data):  
    """  
        :argument data in pd.DataFrame (one column)! to transform  
        :return data frame with keys() from code_keys and data splitted with sepecific labels  
    """  
    if isinstance(data, pd.Series):  
        data = data.to_frame()  
    for column_name in data.columns:  
        unique_elements =list(set(data.loc[:,column_name].values.tolist()))  
  
        self.label_code = {i: unique_element for i, unique_element in enumerate(unique_elements)}  
  
        self.decoded_set ={unique_element:None for unique_element in self.label_code.values()}  
        self.number_of_coded_keys = self.decoded_set.__len__()  
        numpy_zeros_shape_like_num_of_coded_keys=np.zeros((self.number_of_coded_keys,self.number_of_coded_keys))  
        for i,(row,unique_value) in enumerate(zip(numpy_zeros_shape_like_num_of_coded_keys[:,],self.decoded_set.keys())):  
            row[i]=1  
            self.decoded_set[unique_value]=list(row)  
        self.data_code[column_name]={"label_code":self.label_code,
```

Zakodowane dane wyglądają następująco:

Pole **self.decoded_set** zawiera wartości zapisane w słowniku, gdzie kluczem jest nazwa kategorii, a wartością - odpowiadający jej wektor binarny, np.:

- "Cat": [1, 0, 0, 0]
- "Dog": [0, 1, 0, 0]
- "Turtle": [0, 0, 1, 0]
- "Fish": [0, 0, 0, 1]

W powyższym przykładzie w kolumnie **Pet** występowały 4 zmienne Cat,Dog, Turtle, Fish. Na podstawie tych danych tworzymy listy zawierające wartości **0** i **1**, o długości równej liczbie kategorii.

Następnie, za pomocą odpowiedniej funkcji, możemy zamienić jedną kolumnę z danymi kategorycznymi na jej binarną reprezentację, gdzie każda kategoria jest reprezentowana przez wektor zawierający zera i jedynki.

W przypadku wielu kolumn funkcja wykonuje tę operację kodowania osobno dla każdej z nich, korzystając z pętli.

```
def code_y_for_network(self, data):
    """
    :argument data in pd.DataFrame (one column)! to transform
    :return data frame with keys() from code_keys and data splited with sepecific labels
    """
    new_data_Frame_coded = pd.DataFrame(columns=self.decoded_set.keys())
    #print(new_data_Frame_coded)
    if isinstance(data, str):
        new_data_Frame_coded.loc[0] = self.decoded_set[data]
        return new_data_Frame_coded
    for i in range(len(data)):
        new_data_Frame_coded.loc[i] = self.decoded_set[data[i]]
    #print(new_data_Frame_coded)
    #zwraca dataframe który ma rozmiar ilość wierszy X ilość klas
    return new_data_Frame_coded
```

Pet	Cat	Dog	Turtle	Fish
Cat	1	0	0	0
Dog	0	1	0	0
Turtle	0	0	1	0
Fish	0	0	0	1
Cat	1	0	0	0

Konkretny zdekodowany wiersz możemy przetworzyć za pomocą funkcji **decode_keys**, która na podstawie wartości w **label_code** odnajduje odpowiadającą kategorię i zwraca oryginalną wartość przed binaryzacją, np. wektor [0, 0, 1, 0] zostanie zmieniony na "Turtle".

```
def decode_keys(self,y_pred):
    """
    :param y_pred is list - sequence of soft max predictions of classes
    max argument (1) and return dict :{"a": [1,0,0], "b":[0,1,0] ....}
    :return for [0,1,0,0] ->b
    """
    index_max =np.argmax(y_pred)
    #print(index_max)
    if 0 <= index_max <1: # Avoid IndexError
        return self.label_code[index_max]
    else: return self.label_code[index_max]
    raise Exception
    # {"a": [1,0,0], "b":[0,1,0] ....} odkodowanie etykiet klas jak model przewiduje do porównania
```

Wyświetlenie danych dotyczących ćwiczenia modelu:

Wyświetlanie danych dotyczących treningu modelu jest bardzo ważne. Podczas uczenia modelu istotne jest obserwowanie, jak zmieniają się jego wyniki w kolejnych epokach. Monitorujemy wartości na zbiorze treningowym, takie jak strata (loss), a także wyniki na zbiorze walidacyjnym. Informacje z walidacji pomagają sprawdzić, czy model nie przeucza się ani nie niedoucza, oraz czy jakość modelu jest zadowalająca.

```
def show_training_process(train_acc: list, train_loss: list, valid_acc: list, valid_loss: list, test_acc: float,
                           test_loss: float):
    """
    Visualizes the training, validation, and test performance of a neural network over training epochs.

    This function plots training and validation loss and accuracy across epochs using a dual y-axis format.

    :param train_acc: List of training accuracy values .
    :param train_loss: List of training loss values .
    :param valid_acc: List of validation accuracy values .
    :param valid_loss: List of validation loss values .
    :param test_acc: Final test set accuracy (a single float value).
    :param test_loss: Final test set loss (a single float value).
    :raises AssertionError: If the lengths of input lists for training and validation metrics are not equal.
    """

    # Your implementation here
    pass
```

Wyświetlanie danych dotyczących czasu działania funkcjonalności:

Ze względu na brak dodatkowych wymagań najwygodniejszym rozwiązaniem było użycie dekoratora, który zapisuje czas rozpoczęcia i zakończenia wykonania funkcji oraz zapisuje ten czas w pliku razem z logami działania aplikacji.

```
def log_execution_time(function_to_measure):
    new =
    @wraps(function_to_measure)
    def wrapper(*args, **kwargs):
        start = time.time()
        result = function_to_measure(*args, **kwargs)
        end = time.time()

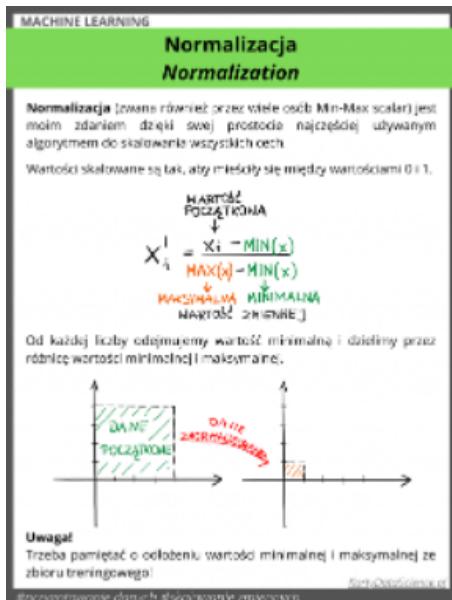
        # Calculate execution time
        time_execution = end - start
        if time_execution > 120:
            time_execution /= 60
            time_execution = f"{time_execution:.2f} minutes"
        else:
            time_execution = f"{time_execution:.2f} seconds"

        # Log execution time
        logging.info(f"Executed function: {function_to_measure.__name__} in: {time_execution}")

    return wrapper
```

- Standaryzacja danych:

Standaryzacja poprzez normalizację min-max:



- Algorytm standaryzacji danych z-score:

Wzór:

$$z = \frac{x - \mu}{\sigma}$$

Gdzie:

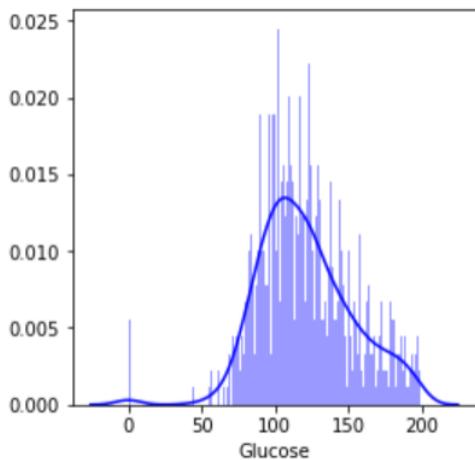
- x — pojedyncza wartość (dana),
- μ — średnia (mean) zbioru danych,
- σ — odchylenie standardowe (standard deviation),
- z — z-score (wartość po standaryzacji).

STANDARDIZE function
returns a normalized value from a distribution based on mean and standard_dev.

Item	Sale	Z-score
Chair	64	-0.54
Desk	71	0.06
Laptop	77	0.57
Keyboard	68	-0.20
Monitor	72	0.14
Mouse	54	-1.39
Lamp	60	-0.88
Notebook	98	2.36
Gel pen	69	-0.11

Mean Standard Deviation

70.33 11.71



StandarizationType.Z SCORE określa jaki typ standaryzacji zostanie wykonany w funkcji standarization_of_data, a następnie w celu zapisania informacji.

```
usage: new*
def process_standarization(data_containing_required_columns):
    from Data.Transformers import StandardizationType, Transformations
    # std zawiera informacje które są wykorzystywane przy transformacji punktu
    std = Transformations(data_containing_required_columns, StandardizationType.Z_SCORE)
    trained_data = std.standarization_of_data(data_containing_required_columns)
    # print(trained_data)

    ## save data
    std.save_data()
    return trained_data
```

Metoda `save_data()` zapisuje dane w pliku `.json` w taki sposób aby było możliwe wczytanie danych i utworzenie instancji:

```
{"minimums": [376.0,
  0.0, 0.0,
  371.05,
  22.0,
  -932.0],
 "maximums": [850.0,
  10.0,
  1.0,
  199725.39,
  61.0,
  175632.0],
 "srednie": [648.84316,
  5.06893,
  0.50849,
  98393.45195,
  42.58641,
  1411.03996],
 "odchylenia_w_kolumnach": [98.26106623930913,
  2.9263766121551695, 0.49992788908582975,
  57154.99859917158, 9.062044723472626,
  7990.937598598606],
 "type": "standaryzacja_z_score",
```

Dane zostaną w późniejszym etapie odczytane za pomocą metody **load_data**, która utworzy instancję klasy **Transformations** i zapisze w jej polach wartości odczytane z pliku. Te wartości posłużą do standaryzacji danych przekazanych przez użytkownika w metodzie **standardization_one_point()**, której jako argument przekazujemy listę danych do standaryzacji.

Następnie przechodzimy do transformacji za pomocą one hot encoder. W tym kroku potrzebujemy zarówno informacji o kodowaniu, jak i o sposobie dekodowania wartości.

```
@log_execution_time
def process_one_hot_encoder(data_containing_required_columns, result_dict, key):
    one_hot = OneHotEncoder()
    one_hot.code_keys(data_containing_required_columns)
    data_one_hot = one_hot.code_y_for_network(data_containing_required_columns)
    one_hot.save_data()
    result_dict[key] = data_one_hot
```

One hot encoder działa w następujący sposób:

Data_containing_required_columns zwraca kolumny z etykietami

Jednocześnie wykonuje się jedna kolumna potrzebujemy jej nazwy: **for_one_hot.keys()** = zwróci np. wartości dla kolumny country -> „Country”

Metoda **one_hot.code_keys(data)** zapisze że dane w kolumnie będą miały przypisane wartości:

-> {„France”:[0,0,1], „Germany”:[0,1,0], „Spain”:[1,0,0]}

label_code -> {1: „France”, 2: „Germany”, 3: „Spain”}

numer_of_coded_keys -> 3 jest to ilość elementów kodowanych

Po zapisaniu następuje zamiana danych z tabeli głównej według kodu powstają podtabele, takie jak:

	Spain	France	Germany	services	entrepreneur	retired	...	management	blue-collar	unknown
0	0.0	1.0	0.0	0.0	0.0	0.0	...	1.0	0.0	0.0
1	1.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0
2	0.0	1.0	0.0	0.0	0.0	1.0	...	0.0	0.0	0.0
3	0.0	1.0	0.0	0.0	0.0	0.0	...	0.0	1.0	0.0
4	1.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	1.0
...
996	1.0	0.0	0.0	0.0	0.0	0.0	...	0.0	1.0	0.0
997	1.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	1.0
998	0.0	0.0	1.0	0.0	0.0	0.0	...	0.0	0.0	0.0
999	0.0	1.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0
1000	0.0	0.0	1.0	0.0	0.0	0.0	...	0.0	1.0	0.0

Te dane również są zapisywane odpowiednio za pomocą metod:

- save_data(wszystkie zakodowane klucze każdej kolumny)
 - load_data(wczytujedane i tworzy instancje klasy) do listy

Zapisany plik wygląda następująco:

```
"education": {
    "secondary": [1.0,
        0.0,
        0.0,
        0.0],
    "unknown": [0.0,
        1.0,
        0.0,
        0.0],
    "tertiary": [0.0,
        0.0,
        1.0,
        0.0],
    "primary": [0.0,
        0.0,
        0.0,
        1.0]
},
"label_code": {
    "0": "secondary",
    "1": "unknown",
    "2": "tertiary",
    "3": "primary"
}
```

Gdy dane liczbowe i ciągłe zostały już ustandaryzowane, a dane dyskretne przekształcone na wektory binarne (zawierające wartości 0 i 1), **łączymy je w jedną tabelę** za pomocą funkcji **pd.concat** z biblioteki pandas. Do pierwotnego zbioru danych (`data_set`) dołączamy w ten sposób dane zakodowane przez one hot encoder, dane ustandaryzowane oraz te, które zostały ręcznie zamienione na wartości binarne (0 i 1).

Ponieważ sieci neuronowe są metodą uczenia nadzorowanego, konieczne jest posiadanie zmiennej decyzyjnej, która służy jako punkt odniesienia dla procesu uczenia. To właśnie na jej podstawie model uczy się, porównując swoją predykcję z oczekiwana wartością i obliczając błąd, a następnie jego pochodną, aby zoptymalizować działanie.

Na końcu do tabeli dodajemy kolumnę **ze zmienną decyzyjną**, którą oznaczamy jako **y**. Zmienna ta zostaje **zakodowana binarnie**: wartość "no" zostaje zamieniona na 0, a "yes" na 1. W ten sposób otrzymujemy kompletny zbiór danych gotowy do trenowania modelu.

```
    data_set = pd.concat((data_set, data_modified), axis=1)
one_hot.save_data(code)

data_set["HasCrCard"] = merged_data["HasCrCard"].values
data_set["loan"] = merged_data["loan"].values
data_set["Gender"] = merged_data["Gender"].values
#print(code, sep="\n")

data_set = pd.concat((data_set, normalized), axis=1)
y.replace(("yes", "no"), (1, 0), inplace=True)
data_set["y"] = y
return data_set
```

Kolejnym etapem jest **trening sieci neuronowej**. W oparciu o dostępne dane oraz różne konfiguracje i układy warstw, wybrana została jedna z efektywniejszych architektur - sieć neuronową składającą się z dwóch warstw ukrytych, z których każda zawiera po 12 neuronów, oraz jednej warstwy wyjściowej z pojedynczym neuronem.

Jako funkcję aktywacji zastosowano **funkcję ReLU lub ELU**, natomiast uczenie sieci odbywa się z wykorzystaniem **propagacji wstecznej błędu** (backpropagation) oraz metody **Stochastic Gradient Descent (SGD)**, która polega na aktualizacji wag sieci po każdej iteracji, czyli po przetworzeniu każdego pojedynczego wiersza danych. Do oceny błędu modelu w trakcie treningu używana jest funkcja straty **Mean Squared Error (MSE)**.

Przed rozpoczęciem treningu importujemy klasę sieci neuronowej oraz klasę odpowiedzialną za podział danych na zbiory **treningowe, testowe i walidacyjne**.

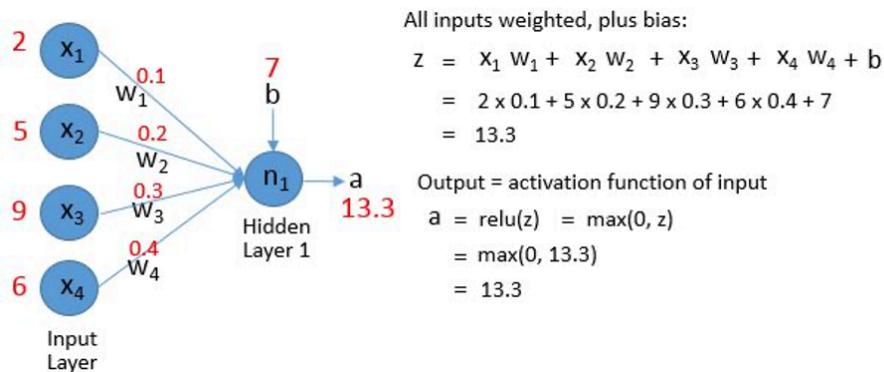
```
from Data.SPLIT_test_valid_train import SplitData  
from NeuralNetwork.Network_single_class import NNetwork
```

Najpierw tworzymy instancję klasy **SplitData**, która odpowiada za podział danych na odpowiednie zbiory. Klasa ta posiada metodę **set(train, valid, test)**, która pozwala określić, w jakich proporcjach dane mają zostać podzielone.

Np., dla zbioru zawierającego 1000 próbek, ustawienie proporcji jako 0.6 dla zbioru treningowego oraz 0.2 dla walidacyjnego oznacza, że odpowiednio 600 próbek trafi do zbioru treningowego, a 200 do walidacyjnego. Pozostałe 200 próbek zostaną przydzielone do zbioru testowego.

```
2 usages  zdrapka-jpg *  
def training(data):  
    # gotowe dane dzielimy  
    Split = SplitData()  
    # ustawiamy wartości danych  
    Split.set(train=0.6, valid=0.2, test=0.2)  
    x_train,y_train,x_valid,y_valid,x_test,y_test =Split.split_data(data)
```

Metoda `split_data(data)` dzieli podane dane w taki sposób ze tasuje zbiór a więc za każdym razem dostaniemy inne dane, i dzieli je jako wszystkie wiersze poza ostatnim są traktowane jako x a ostatnia kolumna jako y i zwraca gotowe dane czyli `x_train,y_train` które zostaną wykorzystane do uczenia modelu , `x_valid,y_valid` są potrzebne w warunku który sprawdza czy model może mieć zmniejszone wagи gdy osiągnie odpowiednią skuteczność.



Model posiada wbudowany wybór funkcji aktywacji przy uczeniu jednak najlepsze wyniki i najszybszą zbieżność posiada aktywacja relu i elu :

```
zdrapiek.jpg
def activation(self, suma_wazona): # [macierz wynikowa jednego wymiary tyle ile jest neuronów]
    """
    :return activation of product according to chosen self.activation_layer
    """
    if self.activation_layer == "softmax":
        return self.softmax(suma_wazona)
    if self.activation_layer == "sigmoid":
        z = lambda x: 1 / (1 + np.exp(-x))
        return z(suma_wazona)
    if self.activation_layer == "Elu":
        return np.where(suma_wazona > 0, suma_wazona,
                       np.where(suma_wazona < 0, 0.01 * (np.exp(suma_wazona) - 1), 0))
    if self.activation_layer == "Relu":
        return np.maximum(0, suma_wazona)
```

Microsoft

Ponieważ dane przygotowane do modelu są rozmiaru data.shape-> (31,1000)

Sieć będzie posiadać 31 wejść , po 12 neuronów w pierwszej i drugiej warstwie oraz 1 neuron na wyjściu.

```
class NNetwork():
    __slots__ = ["epoki", "alpha", "LineOne", "LineTwo", "LineThree", "loss", "train_accuracy", "valid_accuracy", "valid_loss"]
    zdrapiek.jpg
    def __init__(self, w=None, epoki=None, alpha=0.02, batche = None):
        self.epoki = epoki
        self.alpha = alpha
        if epoki is None:
            self.epoki = 2

        self.LineOne = LayerFunctions(len_data=31, wyjscie_ilosc=12, activation_layer="Elu")
        self.LineTwo = LayerFunctions(len_data=12, wyjscie_ilosc=12, activation_layer="Elu")
        self.LineThree = LayerFunctions(len_data=12, wyjscie_ilosc=1, activation_layer="sigmoid")
        self.LineOne.start(self.alpha*1.6)
        self.LineTwo.start(self.alpha)
        self.LineThree.start(self.alpha)
```

Dzięki oddzieleniu logiki warstw od struktury całej sieci możliwe jest swobodne modyfikowanie liczby wejść i wyjść w poszczególnych warstwach. Należy jednak pamiętać, że liczba wyjść jednej warstwy musi odpowiadać liczbie wejść kolejnej. Na przykład, jeśli pierwsza warstwa zawiera 12 neuronów, to druga warstwa musi mieć 12 wejść, ale może posiadać dowolną liczbę wyjść.

Możliwe jest również indywidualne określanie funkcji aktywacji dla każdej warstwy, takich jak **ReLU** czy **ELU**. W przypadku problemów klasyfikacyjnych, w których decyzja ma postać wartości binarnej (**0 lub 1**), zaleca się użycie funkcji **ReLU w warstwach ukrytych** - zapewnia ona dobrą zbieżność uczenia. Dla **warstwy wyjściowej** najlepszym wyborem jest funkcja aktywacji **typu sigmoid** lub inna, która przekształca wynik do przedziału [0, 1], umożliwiając interpretację wyniku jako prawdopodobieństwo klasy pozytywnej.

```
network = NNetwork(epoki=15, alpha=0.4)
network.train(x_train, y_train, x_valid, y_valid)
network.epoki=12
network.alpha=0.08
network.train(x_train, y_train, x_valid, y_valid)
#listy danych predykcji
out, loss = network.perceptron(x_test, y_test)
print("test accuracy:", out)
net_loss = network.loss
net_acc = network.train_accuracy
valid_loss = network.valid_loss
valid_accuracy=network.valid_accuracy
print(len(valid_loss), len(valid_accuracy))
print(len(net_loss), len(net_acc))
print(net_loss)
print(net_acc)
```

Dodatkowo, po piątej epoce treningu sprawdzany jest warunek, który można opisać następująco:

Jeśli skuteczność modelu w bieżącej epoce jest o co najmniej 1% wyższa niż w epoce poprzedniej lub przekracza wartość 0.55, lub skuteczność na danych walidacyjnych wynosi więcej niż 0.56, to współczynnik uczenia (alfa) zostaje zaktualizowany — jego wartość jest zmniejszana o połowę (mnożona przez 0.5).

```

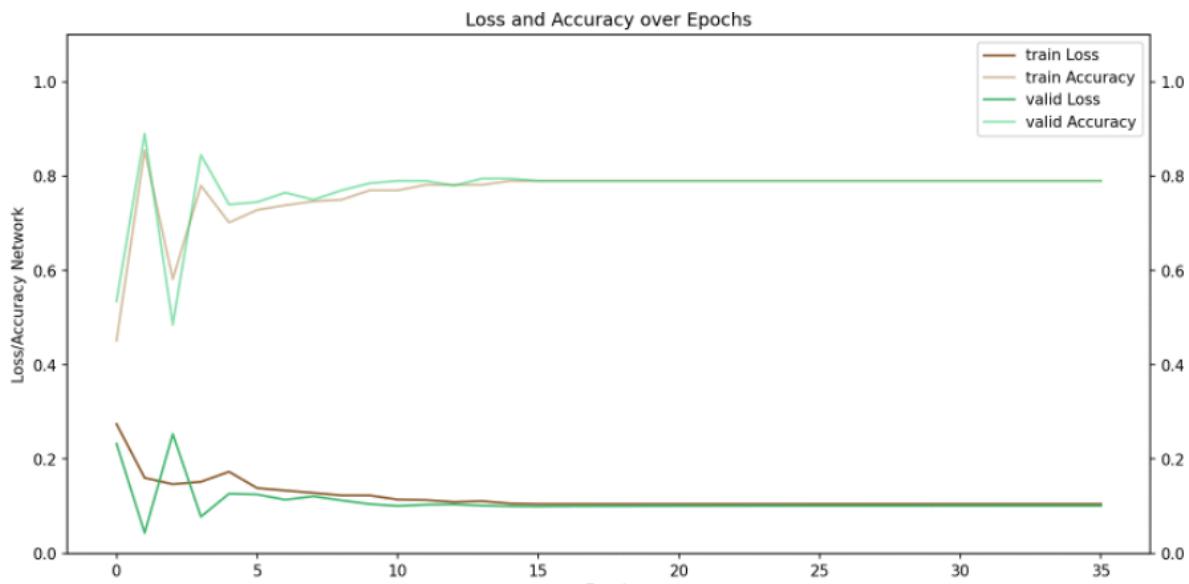
try:
    if j%5==0 and (train_acc / self.train_accuracy[-3]) >= 1.01 and train_acc >= 0.56 and valid_acc > 0.55 and help<=6:
        print("zmnieszenie wagi",j)
        self.alpha = self.alpha *0.5
        help+=1
    for layer in self._Network:
        layer.alfa = self.alpha

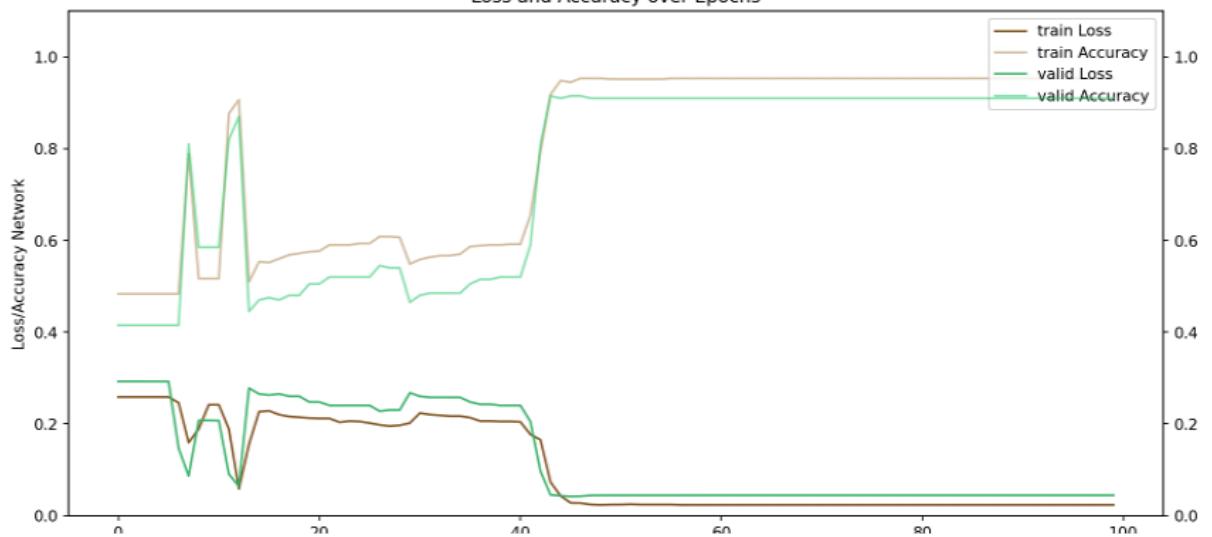
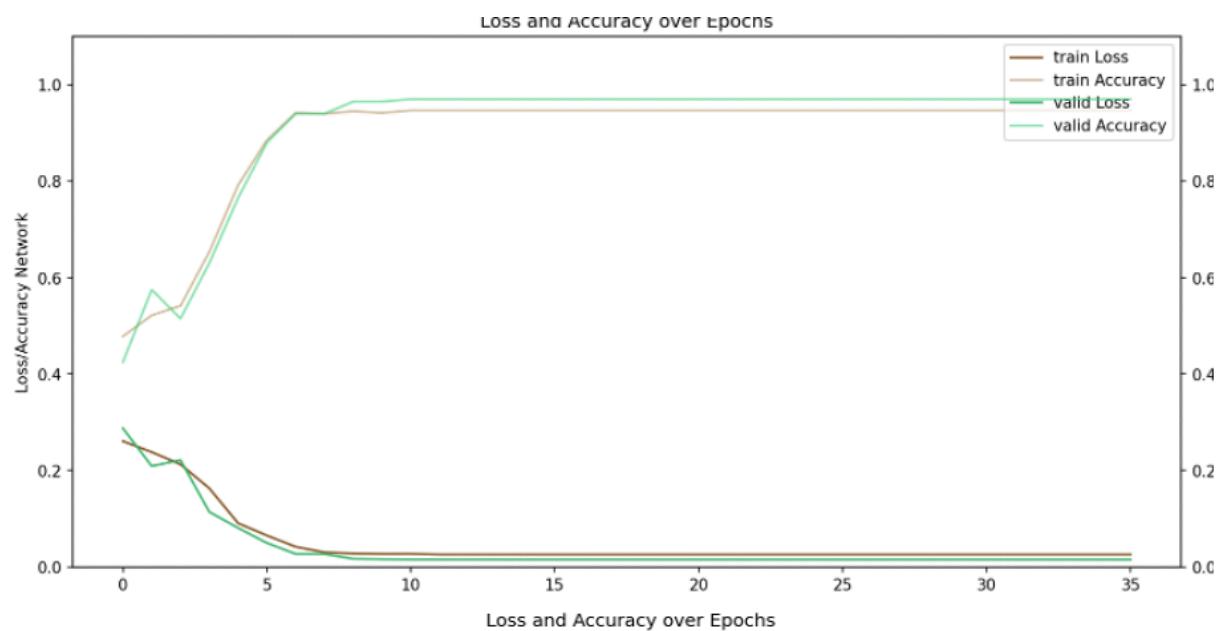
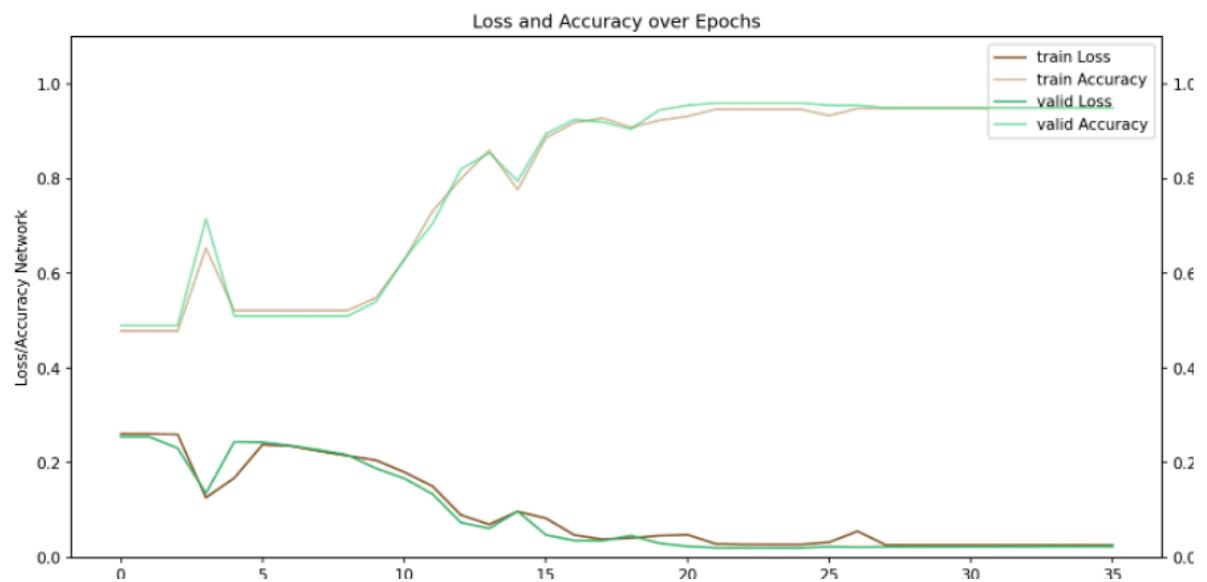
    if j%2==0:
        czy_oba_wieksze_od80 = train_acc>0.85 and valid_acc>0.85
        czy_modelowi_nie_spada_jakosc =self.train_accuracy[-2]>train_acc and self.valid_accuracy[-2]>valid_acc
        czy_modelowi_nie_rosnie_blood = self.loss[-2]-self.loss[-1]>=0.013 and self.valid_loss[-2]-self.valid_loss[-1]>=0.013
        if (czy_oba_wieksze_od80 and(czy_modelowi_nie_spada_jakosc and czy_modelowi_nie_rosnie_blood )) or train_acc>=0.9:
            break
    if j >=50 and j%50==0:
        loss_greater_than01 = self.loss[-1] >= 0.1 and self.valid_loss[-1] >= 0.1
        accuracys_loss_than = train_acc < 0.60 or valid_acc < 0.60
        if loss_greater_than01 or accuracys_loss_than :
            print("zmiana parametrów", j)
            for layer in self._Network:
                layer.start(0.5)

```

Poniżej przedstawiono wyniki dla kilku przykładowych modeli, w których parametry (takie jak wagi i biasy) są **inicjalizowane losowo**. Dla każdego modelu ustalony jest współczynnik uczenia, liczba epok oraz identyczny zestaw danych wejściowych. Dane te są jednak za każdym razem tasowane, co sprawia, że **podział na zbiory treningowy, testowy i weryfikacyjny** jest inny przy każdym uruchomieniu.

Dodatkowo, ze względu na brak zastosowania bardziej zaawansowanych algorytmów uczenia, takich jak **batch gradient descent** czy **optimizatorów typu Adam**, wyniki modeli mogą się znacznie różnić między poszczególnymi uruchomieniami.





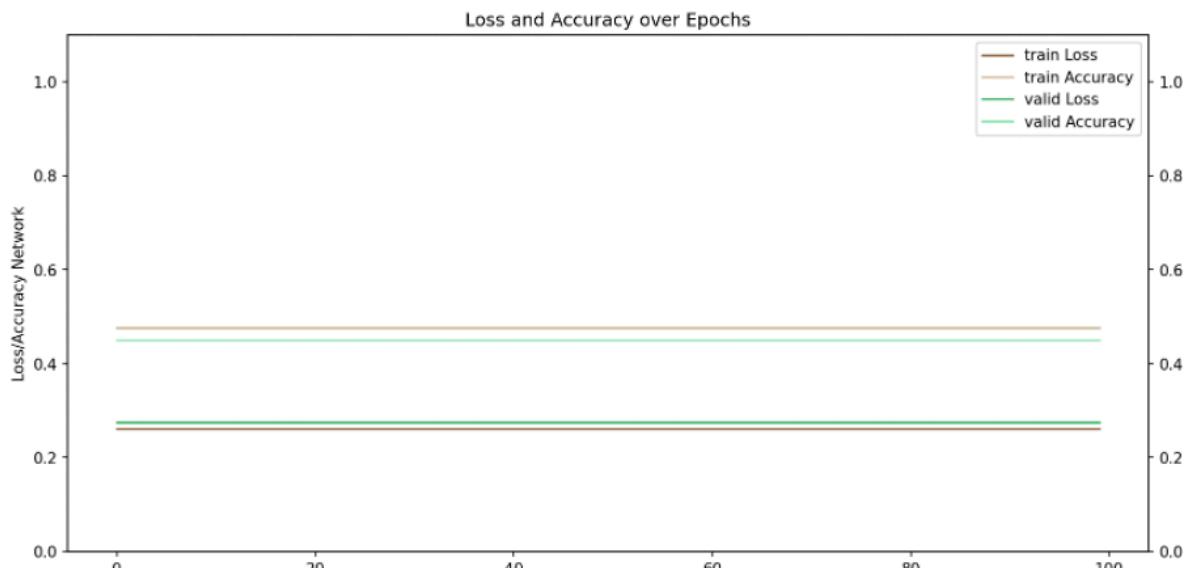
Wyniki dla ostatniego wykresu wyglądają następująco:

```
zmniejszenie wag
test accuracy: 0.9303482587064676
train loss: 0.023003322111152833 train acc: 0.9533333333333334
valid loss: 0.04406615188153421 valid acc: 0.91
test loss: 0.0334589876986713 test acc 0.9303482587064676
model saved
```

Po zakończeniu procesu uczenia sieci i osiągnięciu satysfakcyjującego wyniku - w tym przypadku skuteczności powyżej 90% na zbiorze testowym - parametry modelu, takie jak **wagi i biasy**, zostają zapisane do pliku **w formacie JSON**. Plik ten może zostać później załadowany, tworząc nową instancję modelu z tymi samymi parametrami, gotową do dalszego wykorzystania bez konieczności ponownego trenowania.

Warto jednak zauważyć, że model może wpaść w **tzw. minimum lokalne**. Oznacza to, że pomimo kolejnych epok treningu i aktualizacji wag, nie następuje poprawa skuteczności ani spadek wartości błędu. Dzieje się tak z powodu braku zastosowania bardziej zaawansowanych algorytmów optymalizacji, takich jak np. **Adam**, co jest typową wadą klasycznego algorytmu **Stochastic Gradient Descent (SGD)**.

Poniżej przedstawiono przykład modelu, który nie nauczył się poprawnie mimo przejścia przez 100 epok - skuteczność i błąd pozostały praktycznie bez zmian.



Obsługa danych przychodzących z zewnątrz:

```
def modify_user_input_for_network(data, klucze=['Surname', 'CreditScore',
                                              'Country', 'Gender', 'Tenure', 'HasCrCard',
                                              'IsActiveMember', 'EstimatedSalary', 'age', 'job',
                                              'marital', 'education', 'balance', 'loan',]):
    ## ['CreditScore', 'Country', 'Gender', 'Tenure', 'HasCrCard', 'IsActiveMember',
    # 'EstimatedSalary', 'age', 'job', 'marital', 'education', 'balance', 'loan']
    merged_data = pd.DataFrame(
        [data],
        columns=klucze
    )
    merged_data["loan"] = np.where(merged_data["loan"] == "yes", 1, 0)

    merged_data["HasCrCard"] = pd.to_numeric(merged_data.loc[:, "HasCrCard"])
    merged_data["Gender"] = np.where(merged_data["Gender"] == "Male", 1, 0)
    colone_hot = [2, 9, 10, 11]
```

Pierwszy etap pracy z danymi wejściowymi dostarczonymi z zewnątrz przebiega identycznie jak w przypadku przygotowywania danych do uczenia modelu. Należy ponownie podzielić kolumny na te, które będą przetwarzane za pomocą standaryzacji, oraz te, które zostaną zakodowane przy użyciu One-Hot Encodingu.

Różnica polega na tym, że tym razem nie tworzymy nowego modelu ani nie przeprowadzamy procesu uczenia. Zamiast tego, korzystamy z wcześniejszych zapisanych parametrów — wczytujemy je z plików JSON, co pozwala na zastosowanie wcześniejszej wytrenowanego modelu do przetwarzania nowych danych.

```
def load():
    global normalization_instance, one_hot_instance, model_instance
    normalization_instance = Transformations.load_data() # Load the transformation data
    one_hot_instance = OneHotEncoder.load_data() # Load one-hot encoding data
    model_instance = NNetwork.create_instance() # Create the model instance
    print("Data loaded successfully!")
```

Obiekt jest wczytywany przez funkcję `load`, tworzymy instancje i zapisujemy je globalnie w programie, za pomocą `thread` funkcja jest wykonywana w międzyczasie gdy uruchamiane jest okienko GUI :

```
# Load data when the application starts in a separate thread
threading.Thread(target=load).start()

window.mainloop()
```

Dane z formularza pobieramy i transformujemy :

```
def make_prediction():
    user_data = collect_data()
    data_forNN = modify_user_input_for_network(user_data, one_hot_instance, normalization_instance)

    prediction = model_instance.pred(data_forNN[0])

    # Update the UI with the prediction using the after() method
    window.after(0, update_prediction_label, prediction[0])
    # Re-enable the submit button after prediction
    window.after(0, enable_submit_button)
```

Modyfikowanie pojedynczego zbioru danych, czyli jednego wiersza, to funkcja standaryzacji i One-Hot-Encoder do którego przekazujemy listę danych.

Z pliku JSON tworzymy instancję klasy OneHotEncoder, w której umieszczamy wartości odczytane z pliku pod odpowiednimi kluczami. Następnie transformujemy te dane na obiekt typu DataFrame, gdzie unikatowe wartości znajdujące się w wierszach stają się nazwami kolumn, a wartości wypełniane są zgodnie z przypisany kodowaniem.

Dla przykładowej kolumny education uzyskamy następującą strukturę:

	secondary	unknown	tertiary	primary
0	0.0	0.0	1.0	0.0

Następnie łączymy wszystkie dane, pomijając tym razem wartość decyzyjną, której już nie posiadamy. Tworzymy instancję modelu, do której wczytujemy parametry — wagi, biasy oraz informacje o funkcjach aktywacji.

Metoda **y_predict** zwraca wartość typu float, gdzie wynik bliższy 1 oznacza większe prawdopodobieństwo, że klient opuści bank, natomiast wynik bliższy 0 wskazuje na większe szanse, że klient pozostanie w banku.

```
from json import loads, dumps
from NeuralNetwork.Network_single_class import NNNetwork

if __name__ == "__main__":
    #input_json = input_list = ["Henryk", 619, "France", "Female", 2, 1, 1, 101348.88, 58, "management", "married", "tertiary", 6429, "no"]

    input_json = sys.argv[1]
    input_list = loads(input_json)
    user_input_for_network = modify_user_input_for_network(input_list)[0]
    model = NNNetwork.create_instance()
    y_predicted = model.pred(user_input_for_network)
    print(dumps(y_predicted)) # Output result as JSON|
```

GUI: