

Projekt
drogi dla



„Kierunki –
gospodarki”

Tytuł: Struktury danych wykorzystywane
w algorytmach sztucznej inteligencji

Przedmiot: Algorytmy i struktury danych

Kierunek studiów: Matematyka

Autor: Marcin Kowalewski



Fundusze Europejskie
dla Rozwoju Społecznego



Dofinansowane przez
Unię Europejską

Spis treści

1	Wektory i macierze	2
1.1	Wprowadzenie	2
1.2	Definiowanie wektora	2
1.3	Definiowanie macierzy	3
1.3.1	Operacje na macierzach	3
1.4	Tworzenie symulowanego zbioru danych	7
2	Tensory	9
2.1	Tensory w PyTorch	9
2.2	Operacje na tensorach	12
3	Drzewa decyzyjne	14
3.1	Trenowanie klasyfikatora drzewa decyzyjnego	14
3.2	Trenowanie regresora drzewa decyzyjnego	16
3.3	Zastosowanie algorytmów decyzyjnych	17

1 Wektory i macierze

1.1 Wprowadzenie

W niniejszym rozdziale skupimy się na pojęciach wektorów i macierzy oraz na najważniejszych operacjach na tych strukturach, ilustrowanych praktycznymi przykładami z użyciem biblioteki NumPy.

NumPy (*Numerical Python*), to fundamentalna biblioteka języka Python, która umożliwia efektywną pracę z tablicami wielowymiarowymi oraz oferuje zestaw funkcji matematycznych. Jest podstawą dla wielu innych bibliotek, takich jak Pandas, SciPy czy TensorFlow, które są szeroko wykorzystywane w pracy z algorytmami sztucznej inteligencji.

1.2 Definiowanie wektora

Podstawową strukturą danych biblioteki NumPy jest tablica wielowymiarowa. Aby zdefiniować wektor, wystarczy zdefiniować tablicę jednowymiarową.

```
1  # Utworzenie wektora przedstawiającego wiersz.
2  vector_row = np.array([1, 2, 3])
3
4  # Utworzenie wektora przedstawiającego kolumnę.
5  vector_column = np.array([[1], [2], [3]])
6
7  # Wyświetlenie wymiarów
8  print("Wymiary vector_row:", vector_row.shape)
9  print("Wymiary vector_column:", vector_column.shape)
```

```
Wymiary vector_row: (1, 3)
Wymiary vector_column: (3, 1)
```

1.3 Definiowanie macierzy

```
1  import numpy as np
2  # Tworzenie tablicy dwuwymiarowej (macierzy) 3x3
3  matrix = np.array([[1, 2, 3],
4  [4, 5, 6],
5  [7, 8, 9]])
6  # Wyświetlenie wymiarów tablicy
7  print("Wymiary:", matrix.shape)
```

Wymiary: (3, 3)

1.3.1 Operacje na macierzach

Operacje arytmetyczne na dwóch macierzach

```
1  import numpy as np
2  # Definiowanie dwóch macierzy
3  A = np.array([[1, 2], [3, 4]])
4  B = np.array([[2, 0], [1, 3]])
5  # Mnożenie macierzy za pomocą np.dot()
6  C = np.dot(A, B)
7  # Dodanie dwóch macierzy.
8  np.add(A, B)
9  # Odjęcie jednej macierzy od drugiej.
10 np.subtract(A, B)
```

Wartość najmniejsza i największa

```
1  import numpy as np
2  matrix = np.array([[1, 2, 3],
3  [4, 5, 6],
4  [7, 8, 9]])
5  np.max(matrix) # 9
6  np.min(matrix) # 1
```

Obliczanie średniej, wariancji i odchylenia standardowego

```
1  import numpy as np
2  matrix = np.array([[1, 2, 3],
3  [4, 5, 6],
4  [7, 8, 9]])
5
6  # Wartość średnia
7  np.mean(matrix) # 5.0
8  # Wariancja
9  np.var(matrix)   # 6.666666666666667
10 # Odchylenie standardowe
11 np.std(matrix)
12 # Wyszukanie wartości średniej w każdej kolumnie
13 np.mean(matrix, axis=0) # array([ 4., 5., 6.]
```

Zmiana kształtu tablicy

Chcemy zmienić kształt (liczbę wierszy i kolumn) tablicy bez modyfikowania wartości jej elementów.

```
1  import numpy as np
2  matrix = np.array([[1, 2, 3],
3  [4, 5, 6],
4  [7, 8, 9],
5  [10, 11, 12]])
6  # Zmiana kształtu macierzy z 4x3 na 2x6.
7  matrix.reshape(2, 6)
8  # array([[ 1,  2,  3,  4,  5,  6],
9  #        [ 7,  8,  9, 10, 11, 12]])
```

Metoda *reshape()* pozwala na zmianę rozmiaru tablicy w taki sposób, aby zachowała te same dane, ale przechowywane za pomocą innej liczby wierszy i kolumn. Jedynym wymaganiem jest, że obie tablice muszą mieć tę samą wielkość.

Transponowanie wektora lub macierzy

```
1  import numpy as np
2  matrix = np.array([[1, 2, 3],
3  [4, 5, 6],
4  [7, 8, 9]])
5  # Transponowanie macierzy.
6  matrix.T
7  # array([[1, 4, 7],
8  #        [2, 5, 8],
9  #        [3, 6, 9]])
```

Spłaszczanie macierzy

Czasami zachodzi potrzeba przekształcenia macierzy w tablicę jednowymiarową (wektor).

```
1  import numpy as np
2  matrix = np.array([[1, 2, 3],
3  [4, 5, 6],
4  [7, 8, 9]])
5  # Spłaszczenie macierzy
6  matrix.flatten()
7  # array([1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Znajdowanie rzędu macierzy

```
1  import numpy as np
2  matrix = np.array([[1, 1, 1],
3  [1, 1, 10],
4  [1, 1, 15]])
5  # Rząd macierzy
6  np.linalg.matrix_rank(matrix) # 2
```

Obliczanie śladu macierzy

```
1  import numpy as np
2  matrix = np.array([[1, 2, 3],
3  [2, 4, 6],
4  [3, 8, 9]])
5  # Ślad macierzy
6  matrix.trace()    # 14
```

Iloczyn skalarny dwóch wektorów

```
1  import numpy as np
2  # Utworzenie dwóch wektorów.
3  vector_a = np.array([1, 2, 3])
4  vector_b = np.array([4, 5, 6])
5  # Obliczenie iloczynu skalarnego.
6  np.dot(vector_a, vector_b)    # 32
```

Odwracanie macierzy

```
1  import numpy as np
2  matrix = np.array([[1, 4],
3  [2, 5]])
4  # Utworzenie macierzy odwrotnej
5  np.linalg.inv(matrix)
6  # array([[-1.66666667,  1.33333333],
7  #        [ 0.66666667, -0.33333333]])
```

1.4 Tworzenie symulowanego zbioru danych

Biblioteka `scikit-learn` udostępnia zestaw narzędzi przeznaczonych do generowania sztucznych zbiorów danych, które można wykorzystać w eksperymentach z algorytmami uczenia maszynowego. Spośród nich szczególnie użyteczne są trzy metody:

`make_regression()`, `make_classification()` oraz `make_blobs()`.

Jeżeli celem jest utworzenie zbioru danych do pracy z regresją liniową, dobrym wyborem jest metoda `make_regression()`.

```
1  from sklearn.datasets import make_regression
2  # Wygenerowanie macierzy cech, wektora docelowego
3  # i prawdziwych współczynników.
4  features, target, coefficients = make_regression(
5      n_samples=100,
6      n_features=3,
7      n_informative=3,
8      n_targets=1,
9      noise=0.0,
10     coef=True,
11     random_state=1)
12 # Wyświetlenie macierzy cech i wektora docelowego.
13 print('Macierz cech\n', features[:3])
14 print('Wektor docelowy\n', target[:3])
```

Metoda `make_regression()` zwraca macierz cech zawierającą wartości zmiennoprzecinkowe oraz wektor docelowy również złożony z liczb zmiennoprzecinkowych.

Jeżeli interesuje nas symulowany zbiór danych przeznaczony do klasyfikacji, wówczas należy skorzystać z metody `make_classification()`.

```
1  from sklearn.datasets import make_classification
2  # Wygenerowanie macierzy cech i wektora docelowego.
3  features, target = make_classification(
4      n_samples=100,
5      n_features=3,
6      n_informative=3,
7      n_redundant=0,
8      n_classes=2,
9      weights=[.25, .75],
10     random_state=1)
11 # Wyświetlenie macierzy cech i wektora docelowego.
12 print('Macierz cech\n', features[:3])
13 print('Wektor docelowy\n', target[:3])
```

Metody `make_classification()` i `make_blobs()` generują macierz cech z wartościami zmiennoprzecinkowymi oraz wektor liczb całkowitych reprezentujący przynależność obserwacji do klasy lub klastra.

Symulowane zbiory danych oferują szeroki zakres parametrów wpływających na charakter generowanych danych. W metodach `make_regression()` oraz `make_classification()` parametr `n_informative` określa liczbę cech wykorzystywanych do wyznaczenia wektora docelowego. Jeżeli jego wartość jest mniejsza niż `n_features`, zbiór danych zawiera cechy nadmiarowe, które mogą zostać wyeliminowane za pomocą technik selekcji cech.

Metoda `make_classification()` udostępnia parametr `weights`, który umożliwia symulowanie zbiorów danych z nie zrównoważonymi klasami, np. `weights = [.15, .85]` oznacza, że 15% obserwacji należy do jednej klasy, a 85% do drugiej.

2 Tensory

Tensory stanowią uogólnienie pojęć skalarów, wektorów i macierzy na dowolny wymiar i są podstawową strukturą danych wykorzystywaną w algorytmach uczenia głębokiego. Umożliwiają one efektywną reprezentację oraz przetwarzanie złożonych danych, takich jak obrazy, sekwencje czasowe czy dane tekstowe. W praktyce wszystkie obliczenia wykonywane w sieciach neuronowych opierają się na operacjach na tensorach, takich jak dodawanie, mnożenie czy przekształcanie kształtu danych.

Podobnie jak biblioteka NumPy stanowi podstawowe narzędzie do wykonywania operacji na danych w stosie uczenia maszynowego, tak framework PyTorch jest kluczowym narzędziem do pracy z tensorami w obszarze uczenia głębokiego.

2.1 Tensory w PyTorch

Utworzenie tensora

Pod wieloma względami tensor przypomina tablicę wielowymiarową, którą łatwo operować za pomocą narzędzi z biblioteki NumPy. Podobnie jak w przypadku wektorów i tablic, tensory mogą być przedstawione poziomo (np. wiersz) lub pionowo (np. kolumna).

```
1  import torch
2  # Utworzenie wektora przedstawiającego wiersz.
3  tensor_row = torch.tensor([1, 2, 3])
4  # Utworzenie wektora przedstawiającego kolumnę.
5  tensor_column = torch.tensor([
6  [1],
7  [2],
8  [3]
9  ])
```

Możemy też utworzyć tensor na podstawie wektora z biblioteki NumPy.

```
1  import numpy as np
2  import torch
3
4  # Utworzenie tablicy biblioteki NumPy.
5  vector_row = np.array([1, 2, 3])
6
7  # Utworzenie tensora na podstawie tablicy NumPy.
8  tensor_row = torch.from_numpy(vector_row)
```

Wybór elementów tensora

```
1  import torch
2  vector = torch.tensor([1, 2, 3, 4, 5, 6])
3  matrix = torch.tensor([
4  [1, 2, 3],
5  [4, 5, 6],
6  [7, 8, 9]
7  ])
8
9  # Pobranie trzeciego elementu wektora.
10 vector[2]      # tensor(3)
11
12 # Pobranie drugiego wiersza, drugiej kolumny.
13 matrix[1, 1]   # tensor(5)
```

Podobnie jak tablice biblioteki NumPy oraz większość struktur danych w języku Python, tensory biblioteki PyTorch wykorzystują indeksowanie rozpoczynające się od zera.

Obsługiwane jest zarówno odwoływanie się do pojedynczych elementów, jak i tworzenie wycinków danych. Istotna różnica polega na tym, że w przypadku indeksowania tensora PyTorch w celu uzyskania pojedynczego elementu, wartością zwrótną nadal jest obiekt typu tensor, a nie bezpośrednio liczba całkowita lub zmiennoprzecinkowa.

Składnia wycinków jest zgodna ze składnią znaną z biblioteki NumPy, a rezultatem każdej takiej operacji w PyTorch pozostaje obiekt typu tensor.

```
1  # Pobranie wszystkich elementów wektora.
2  vector[:]
3
4  # Pobranie wszystkich elementów aż do trzeciego (bez niego)
5  vector[:3]      # tensor([1, 2, 3])
6
7  # Pobranie wszystkich elementów po trzecim
8  vector[3:]      # tensor([4, 5, 6])
9
10 # Pobranie ostatniego elementu
11 vector[-1]      # tensor(6)
12
13 # Pobranie dwóch pierwszych wierszy i wszystkich kolumn
14 matrix[:2, :]   # tensor([[1, 2, 3],
15 #               [4, 5, 6]])
16
17 # Pobranie wszystkich wierszy i drugiej kolumny
18 matrix[:, 1:2]  # tensor([[2],
19 #               [5],
20 #               [8]])
```

2.2 Operacje na tensorach

Operacje arytmetyczne na wszystkich elementach tensora

```
1  import torch
2  tensor = torch.tensor([1, 2, 3])
3  # Rozgłaszanie operacji we wszystkich elementach tensora
4  tensor * 100      # tensor([100, 200, 300])
```

Możemy je wykonywać dla operatorów matematycznych obsługiwanych przez Python (+, -, *, /) oraz funkcji oferowanych przez PyTorch.

Wyszukiwanie wartości minimalnej i maksymalnej

```
1  import torch
2  tensor = torch.tensor([1, 2, 3])
3
4  # Wyszukanie największej wartosci.
5  tensor.max()      # tensor(3)
6  # Wyszukanie najmniejszej wartosci.
7  tensor.min()      # tensor(1)
```

Zmiana kształtu, transpozycja oraz spłaszczanie

```
1  # Zmiana kształtu tensora na 2x6.
2  tensor.reshape(2, 6)
3
4  # Transponowanie tensora.
5  tensor.mT
6
7  # Spłaszczenie tensora
8  tensor.flatten()
```

Obliczanie iloczynu skalarnego oraz mnożenie dwóch tensorów

```
1  import torch
2  tensor_1 = torch.tensor([1, 2, 3])
3  tensor_2 = torch.tensor([4, 5, 6])
4
5  # Obliczenie iloczynu skalarnego obu tensorow.
6  tensor_1.dot(tensor_2)      # tensor(32)
7
8  # Mnożenie dwóch tensorów
9  tensor_1 * tensor_2      # tensor([ 4, 10, 18])
```

3 Drzewa decyzyjne

Oparte na drzewach algorytmy uczące stanowią rozbudowaną i popularną rodzinę nadzorowanych metod klasyfikacji i regresji, które nie wymagają ręcznego doboru parametrów. Ich podstawowym elementem jest drzewo decyzyjne, zbudowane z sekwencji reguł decyzyjnych, na przykład: „jeżeli temperatura powietrza jest niższa niż 0°C, to...”. Struktura takiego modelu przypomina odwrócone drzewo, w którym pierwsza reguła decyzyjna znajduje się na najwyższym poziomie, a kolejne reguły rozgałęziają się na coraz niższych poziomach. W drzewie decyzyjnym każda reguła umieszczona jest w węźle decyzyjnym, a wynik jej spełnienia lub niespełnienia prowadzi do kolejnych gałęzi i węzłów. Gałęzie, które nie zawierają dalszych reguł decyzyjnych, kończą się w tzw. liściach, reprezentujących ostateczną decyzję modelu.

3.1 Trenowanie klasyfikatora drzewa decyzyjnego

Klasyfikator drzewa decyzyjnego dąży do znalezienia takiej reguły decyzyjnej, która prowadzi do możliwie największego zmniejszenia niejednorodności danych w danym węźle. Istnieje kilka miar służących do oceny niejednorodności, jednak domyślnie klasa `DecisionTreeClassifier` wykorzystuje wskaźnik Giniego. Miara ta może zostać zdefiniowana następującym wzorem:

$$G(t) = 1 - \sum_{i=1}^c p_i^2,$$

gdzie $G(t)$ oznacza niejednorodność (wskaźnik Giniego) w węźle t , natomiast p_i jest udziałem obserwacji należących do klasy i w tym węźle.

Proces wyszukiwania reguł decyzyjnych, prowadzący do powstawania kolejnych rozgałęzień drzewa, jest wykonywany rekurencyjnie. Celem tego procesu jest stopniowe zwiększanie jednorodności w węzłach potomnych. Algorytm kończy działanie, gdy wszystkie liście drzewa są jednorodne (na przykład zawierają obserwacje tylko jednej klasy) lub gdy spełnione zostaną zadane kryteria zatrzymania.

W bibliotece `scikit-learn` obiekt klasy `DecisionTreeClassifier` obsługiwany jest w sposób analogiczny do innych algorytmów uczenia maszynowego. Po wytrenowaniu modelu za pomocą metody `fit()` można wykorzystać go do prognozowania klas nowych obserwacji.

```
1  # Utworzenie nowej obserwacji.
2  observation = [[5, 4, 3, 2]]
3  # Prognozowanie klasy obserwacji.
4  model.predict(observation)
5  # array([1])
```

Istnieje również możliwość wyznaczenia prawdopodobieństw przynależności obserwacji do poszczególnych klas.

```
1  # Wświetlenie prawdopodobieństw prognozowanych klas.
2  model.predict_proba(observation)
3  # array([[0., 1., 0.]])
```

Jeżeli zachodzi potrzeba zastosowania innej miary niejednorodności, można skorzystać z parametru `criterion`. Przykładowo, zamiast wskaźnika Giniego można użyć entropii.

```
1  from sklearn.tree import DecisionTreeClassifier
2  # Utworzenie klasyfikatora drzewa decyzyjnego
3  # z wykorzystaniem entropii.
4  decisiontree_entropy = DecisionTreeClassifier(
5      criterion='entropy', random_state=0)
6  # Wytrenowanie modelu.
7  model_entropy = decisiontree_entropy.fit(features, target)
```

3.2 Trenowanie regresora drzewa decyzyjnego

Regresja drzewa decyzyjnego działa w sposób analogiczny do klasyfikatora drzewa decyzyjnego, jednak zamiast miar takich jak wskaźnik Giniego czy entropia, potencjalne podziały danych oceniane są domyślnie na podstawie stopnia redukcji błędu średniokwadratowego (*mean squared error*, MSE). Miara ta określa, jak bardzo wartości prognozowane różnią się od rzeczywistych wartości zmiennej docelowej i może zostać zapisana w postaci wzoru:

$$\text{MSE} = \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2,$$

gdzie y_i oznacza rzeczywistą wartość zmiennej docelowej, natomiast $\hat{y}_i$ jest wartością prognozowaną przez model.

W bibliotece `scikit-learn` regresję drzewa decyzyjnego realizuje się za pomocą klasy `DecisionTreeRegressor`. Po wytrenowaniu modelu można go wykorzystać do prognozowania wartości docelowej dla nowych obserwacji, analogicznie jak w przypadku innych algorytmów regresyjnych.

```
1  from sklearn.tree import DecisionTreeRegressor
2  from sklearn import datasets
3
4  # Wczytanie danych.
5  diabetes = datasets.load_diabetes()
6  features = diabetes.data
7  target = diabetes.target
8
9  # Utworzenie obiektu regresora drzewa decyzyjnego.
10 decisiontree = DecisionTreeRegressor(random_state=0)
11
12 # Wytrenowanie modelu.
13 model = decisiontree.fit(features, target)
```

3.3 Zastosowanie algorytmów decyzyjnych

Poniżej przedstawiono przykład prostego algorytmu w języku Python, który przewiduje, czy klient zrealizuje zakup na podstawie wybranych cech opisujących jego profil.

```
1      from sklearn.tree import DecisionTreeClassifier
2
3      # Dane o klientach: [wiek, liczba zakupów, średni wydatek]
4      data = [[25, 5, 100], [40, 15, 200], [30, 10, 150]]
5      labels = [0, 1, 1] # 0: brak zakupu, 1: zakup
6
7      # Model decyzyjny
8      clf = DecisionTreeClassifier()
9      clf.fit(data, labels)
10
11     # Przewidywanie zakupu dla nowego klienta
12     print(clf.predict([[35, 8, 180]])) # Wynik: [1]
```

Drzewo decyzyjne – dane o klientach

W rozpatrywanym przykładzie wykorzystano dane dotyczące trzech klientów, opisanych trzema cechami:

- **Wiek,**
- **Liczba zakupów,**
- **Średni wydatek.**

Każdemu klientowi przypisano etykietę klasy:

- Klasa 0 – brak zakupu,
- Klasa 1 – zakup.

Przykładowe dane

Zestaw danych treningowych:

- Klient 1: [25, 5, 100] – klasa 0,
- Klient 2: [40, 15, 200] – klasa 1,
- Klient 3: [30, 10, 150] – klasa 1.

Nowy klient podlegający klasyfikacji:

- Wiek: 35 lat,
- Liczba zakupów: 8,
- Średni wydatek: 180 jednostek.

Trenowanie modelu drzewa decyzyjnego

Model drzewa decyzyjnego uczy się na podstawie danych treningowych, tworząc strukturę hierarchiczną złożoną z węzłów decyzyjnych. Każdy węzeł odpowiada warunkowi opartemu na jednej z cech klienta, a celem podziałów jest możliwie najlepsze rozdzielenie klas „brak zakupu” i „zakup”.

Korzystanie z wytrenowanego modelu

Po zakończeniu procesu treningu drzewo decyzyjne może zostać wykorzystane do przewidywania klasy dla nowych obserwacji. Model przechodzi przez kolejne węzły drzewa, sprawdzając warunki logiczne, aż dotrze do liścia, który odpowiada ostatecznej decyzji.

Proces klasyfikacji nowego klienta

Proces klasyfikacji przebiega według następujących kroków:

1. Sprawdzenie warunku w pierwszym węźle (np. wartości średniego wydatku),
2. Przejście do odpowiedniej gałęzi drzewa,
3. Weryfikacja kolejnych warunków w następnych węzłach,
4. Dotarcie do liścia i przypisanie klasy decyzyjnej.

Przykładowy przebieg klasyfikacji

Założmy, że drzewo decyzyjne zawiera następujące reguły:

- Jeżeli **średni wydatek** ≤ 125 , przejdź do lewej gałęzi,
- W przeciwnym przypadku przejdź do prawej gałęzi.

Dla nowego klienta o średnim wydatku równym 180 algorytm przechodzi do prawej gałęzi, a następnie przypisuje klasę końcową.

Przypisanie klasy

Po dotarciu do liścia drzewa model przypisuje nowemu klientowi klasę:

[1],

co oznacza, że klient prawdopodobnie dokona zakupu.

Dane do wizualizacji drzewa decyzyjnego

```
1 data = [  
2 [25, 5, 100, 10],  
3 [40, 15, 200, 30],  
4 [30, 10, 150, 20],
```

```
5      [22, 3, 80, 8],
6      [35, 20, 300, 25],
7      [45, 10, 250, 35],
8      [50, 2, 75, 5],
9      [55, 8, 100, 12],
10     [33, 12, 180, 22],
11     [29, 6, 120, 15],
12     [38, 5, 90, 11],
13     [31, 10, 160, 18],
14     [28, 4, 85, 9],
15     [43, 17, 230, 29],
16     [47, 9, 210, 34],
17     [36, 14, 170, 21],
18     [39, 11, 140, 23],
19     [34, 7, 130, 17],
20     [32, 9, 115, 16],
21     [37, 6, 105, 19]
22 ]
```

Tworzenie i wizualizacja drzewa decyzyjnego

```
1      from sklearn import tree
2      import matplotlib.pyplot as plt
3
4      clf = tree.DecisionTreeClassifier()
5      clf = clf.fit(data, labels)
6      plt.figure(figsize=(10, 8))
7      tree.plot_tree(clf, filled=True)
8      plt.show()
```

Algorytm k-NN w klasyfikacji klientów

Algorytm *k-Nearest Neighbors* (k-NN) klasyfikuje obiekty na podstawie ich najbliższych sąsiadów w przestrzeni cech.

```
1     from sklearn.neighbors import KNeighborsClassifier
2     data = [[25, 5, 100], [40, 15, 200], [30, 10, 150]]
3     labels = [0, 1, 1]
4     knn = KNeighborsClassifier(n_neighbors=3)
5     knn.fit(data, labels)
6     print(knn.predict([[33, 7, 160]])) # Wynik: [1]
```

Odległość euklidesowa

Najczęściej stosowaną miarą odległości w algorytmie k-NN jest odległość euklidesowa:

$$d(A, B) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2}.$$

Obliczenia odległości

Dla nowego klienta obliczono odległości do klientów z zestawu treningowego, uzyskując:

- Klient 1: 60.56,
- Klient 2: 41.39,
- Klient 3: 10.86.

Wybór sąsiadów i klasyfikacja

Dla $k = 3$ większość najbliższych sąsiadów należy do klasy „zakup”, dlatego nowy klient zostaje zaklasyfikowany jako dokonujący zakupu.

Bibliografia

- [1] Timofiejew Aleksander, *Algorytmy i struktury danych w językach programowania*, Wydawnictwo Akademii Podlaskiej, 2008.
- [2] Debudaj-Grabysz Agnieszka, Deorowicz Sebastian, Widuch Jacek, *Algorytmy i struktury danych : wybór zaawansowanych metod*, Wydawnictwo Politechniki Śląskiej, 2012.
- [3] Chrzęszczuk Andrzej, *Algorytmy teorii liczb i kryptografii w przykładach*, Wydawnictwo BTC, 2010.
- [4] Deisenroth Marc Peter, Faisal A. Aldo, Ong Cheng Soon, *Matematyka w uczeniu maszynowym*, Helion, 2022.
- [5] Lannelongue Loïc, Grealey Jason, Inouye Michael, *Green Algorithms: Quantifying the carbon footprint of computation*, <https://arxiv.org/pdf/2007.07610>.