

Politechnika Śląska  
Wydział Automatyki, Elektroniki i Informatyki

# Podstawy Programowania Komputerów

Sąsiedzi

---

autor	Michał Brodziak
prowadzący	dr inż. Adam Gudyś
rok akademicki	2021/2022
kierunek	informatyka
rodzaj studiów	SSI
semestr	1
termin laboratorium	wtorek, 08:00 – 09:30
sekcja	11
termin oddania sprawozdania	2022-02-01

---

## Spis treści:

Spis treści: .....	2
1 Treść zadania .....	3
2 Analiza zadania .....	3
2.1 Plik wyjściowy .....	3
2.2 Struktury danych.....	4
2.3 Algorytmy.....	4
3 Specyfikacja zewnętrzna .....	4
4 Specyfikacja wewnętrzna.....	6
4.1 Ogólna struktura programu .....	6
4.2 Szczegółowy opis typów i funkcji.....	7
5 Testowanie .....	7
5.1 Test na przykładowych danych, gdzie punkt testowy jest w obszarze spornym .....	7
5.2 Test na trójwymiarowej przestrzeni. ....	9
6 Wnioski .....	12
Dodatek. Szczegółowy opis funkcji i typów .....	13

# 1 Treść zadania

Napisać program klasyfikujący dane algorytmem k najbliższych sąsiadów. Dane treningowe zapisane są w pliku wejściowym w następującym formacie (znak % rozpoczyna komentarz do końca linii):

100 % liczba punktów treningowych N

2 % liczba wymiarów D

% poniżej znajduje się lista N punktów danych, każdy opisany przez D współrzędnych

% oraz etykietę klasy będącą łańcuchem tekstowym

7.6 5.3 klasa\_B % punkt 1

45.3 4.6 klasa\_C % punkt 2

-3.7 22.9 klasa\_A % punkt 3

...

5.7 -0.8 klasa\_C % punkt 100

Dane testowe zapisane są w analogicznym pliku, ale bez etykiet klas:

10 % liczba punktów testowych M

2 % liczba wymiarów D

% poniżej znajduje się lista M punktów danych, każdy opisany przez D współrzędnych

1.5 12.3 % punkt 1

4 7.6 % punkt 2

0.2 -0.9 % punkt 3

...

-6 3.15 % punkt 10

## 2 Analiza zadania

Wykorzystując zbiór treningowy, program przyporządkowuje każdemu punktowi ze zbioru testowego etykietę klasy metodą k najbliższych sąsiadów (k to parametr algorytmu).

### 2.1 Plik wyjściowy

Plik wyjściowy posiada format analogiczny do pliku testowego, ale z uzupełnionymi etykietami klas:

10 % liczba punktów testowych

2 % liczba wymiarów D

1.5 12.3 klasa\_C % punkt 1

4 7.6 klasa\_A % punkt 2

0.2 -0.9 klasa\_A % punkt 3

...

-6 3.15 klasa\_C % punkt 10

## 2.2 Struktury danych

W programie wykorzystano tablice alokowane dynamicznie, do których zapisywane są współrzędne punktów testowych, treningowych oraz dystanse między nimi. W tablicach dynamicznych zapisywane są zarówno podane ogólnie klasy punktów treningowych, jak i określone w wyniku działania programu klasy punktów testowych. Wykorzystana została także mapa do zliczania wystąpień konkretnych klas.

## 2.3 Algorytmy

Program bazuje głównie na algorytmie sortowania tablicy. Wykorzystany jest też algorytm obliczania odległości euklidesowej. Ponadto przy zapisywaniu danych z pliku do tablicy używany jest dostęp do pliku oraz dostęp do tablicy. Marginalnie program używa także dostępu do mapy.

# 3 Specyfikacja zewnętrzna

Program uruchamiany jest z linii poleceń z wykorzystaniem następujących przełączników:

-train plik treningowy

-test plik testowy

-out plik wyjściowy

-k liczba najbliższych sąsiadów k ze zbioru treningowego

Przykładowe uruchomienie programu:

```
projekt.exe -train treningowe.txt -test testowe.txt -out wyjscie.txt -k 4
```

Uruchomienie programu bez któregoś z przełączników spowoduje wypisanie krótkiej instrukcji odnośnie tego co powinniśmy zrobić. Jeśli jeden z przełączników podanych przez nas będzie nie prawidłowy, program nas o tym poinformuje.

Przykładowe struktury plików dla:

- Punktów treningowych

```
20      %liczba punktow treningowych N
3       %liczba wymiarow D
%ponizej znajduje sie lista n punktow danych, opisanych przez d wspolrzecznych
%oraz etykiety klasy bedacych lancuchem tekstowym
7.6 5.3 1.3 klasa_A
45.3 4.6 -2 klasa_A
-3.7 22.9 -8 klasa_C
2.4 12.2 -16 klasa_A
38.3 -3.5 -23 klasa_A
-4.7 -9.9 66 klasa_A
0.1 0.1 0.1 klasa_A
-0.1 -0.1 -0.1 klasa_B
0.1 -0.1 0.1 klasa_B
-0.1 0.1 -0.1 klasa_C
5.0 5.0 5.0 klasa_A
2.2 -14.3 2.2 klasa_A
7.3 16.8 7.3 klasa_C
23.0 12.5 -10.0 klasa_C
6.9 6.9 -19 klasa_C
42.0 0.42 -16 klasa_B
21.37 -19.39 -2.2 klasa_C
-22.22 11.11 1.1 klasa_C
6.3 3.3 22.22 klasa_A
4.3 2.6 69.69 klasa_C
```

- Punktów testowych (na wejściu)

```
10      %liczba punktow testowych
3       %liczba wymiarow D
1.5 12.3 4.5
4 7.6 12.7
0.2 -0.9 17.8
0.0 0.0 0.0
-2.0 8.9 6.9
-0.5 3.9 2.1
10.5 10.5 3.7
-10 -10 666
6.4 3.9 888
1.5 21.37 919
```

- Punktów testowych (na wyjściu)

```

10      %liczba punktow testowych
3      %liczba wymiarow D
1.5 12.3 4.5 klasa_C    %punkt 1
4 7.6 12.7 klasa_A     %punkt 2
0.2 -0.9 17.8 klasa_A  %punkt 3
0 0 0 klasa_B    %punkt 4
-2 8.9 6.9 klasa_C    %punkt 5
-0.5 3.9 2.1 klasa_A  %punkt 6
10.5 10.5 3.7 klasa_A %punkt 7
-10 -10 666 klasa_A   %punkt 8
6.4 3.9 888 klasa_A   %punkt 9
1.5 21.37 919 klasa_A %punkt 10

```

## 4 Specyfikacja wewnętrzna

Program został napisany z zasadami podziału na pliki nagłówkowe oraz na pliki zawierające kod. Wyróżniony jest także plik .cpp zawierający funkcje main() i w tym właśnie pliku są wywoływane wszystkie inne funkcje zawarte w innym pliku.

### 4.1 Ogólna struktura programu

W funkcji głównej wywoływana jest funkcja argInfo(), która sprawdza czy ilość argumentów podanych przez użytkownika jest zgodna z ogólnymi założeniami programu oraz przyporządkowuje odpowiednie argumenty do odpowiednich zmiennych. Następnie argumenty przechodzą walidację w funkcji argCheck(), która sprawdza najczęstsze błędy użytkowników w związku z plikami wejściowymi i plikiem wyjściowym. Jeśli w tych dwóch funkcjach program znajdzie jakiś błąd, poinformuje nas o tym. Następnie wywoływana jest funkcja readData(), która czytuje współrzędne punktów testowych i treningowych i zapisuje je do tablic w odpowiedniej kolejności. Tablice mają wymiary ilość punktów testowych na ilość wymiarów oraz ilość punktów treningowych na ilość wymiarów. Funkcja ta czytuje z pliku także klasy punktów treningowych do jednowymiarowej tablicy. Następnie funkcja additionalValid() sprawdza czy argument k podany przez użytkownika nie jest większy od liczby punktów treningowych. W przypadku gdy k jest większe, program wypisuje błąd, gdyż jest to sprzeczne z ogólnymi założeniami. Następnie funkcja checkDistances tworzy tabelę, w której przechowywane są odległości obliczone metodą euklidesową. Później funkcja findTestClasses, dla każdego punktu testowego, znajduje k najbliższe punkty treningowe i odpowiadające im klasy, po czym zapisuje klasy do mapy. Klasa, która dla danego punktu testowego pojawiła się w mapie najczęściej, zostaje przyporządkowana jako klasa danego punktu testowego w tablicy jednowymiarowej testPointClasses. Następnie funkcja writeToFile() wypisuje dane w sposób podobny do pliku wejściowego z punktami testowymi, jednak na końcu dodaje uzyskane w wyniku wcześniejszych operacji klasy punktów testowych. Ostatnim krokiem jest zwolnienie pamięci, czym zajmuje się funkcja deleteTables(), która usuwa wszystkie tablice zaalokowane dynamicznie.

## 4.2 Szczegółowy opis typów i funkcji

Szczegółowy opis typów i funkcji zawarty jest w dodatku na końcu pliku.

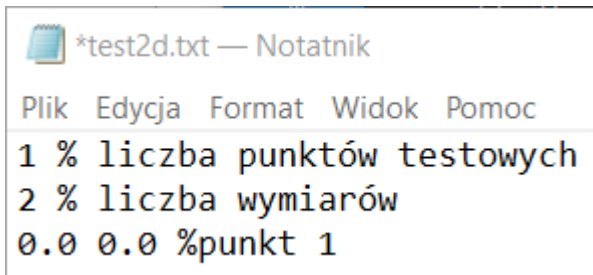
# 5 Testowanie

Program został przetestowany pod kątem różnych argumentów wejściowych. Sprawdzone zostały takie sytuacje jak wpisanie przełączników z błędami, zadeklarowanie zbyt wielkiego argumentu  $k$ , uruchomienie pliku ze zbyt małą ilością przełączników. Sprawdzone zostały także same dane w plikach wejściowych. Zostały one zmodyfikowane tak, aby w jak najlepszy sposób ukazać działanie programu. Wyniki testów przedstawiam poniżej.

## 5.1 Test na przykładowych danych, gdzie punkt testowy jest w obszarze spornym

Przykładowe pliki tekstowe:

- Testowy:



```
*test2d.txt — Notatnik
Plik  Edycja  Format  Widok  Pomoc
1 % liczba punktów testowych
2 % liczba wymiarów
0.0 0.0 %punkt 1
```

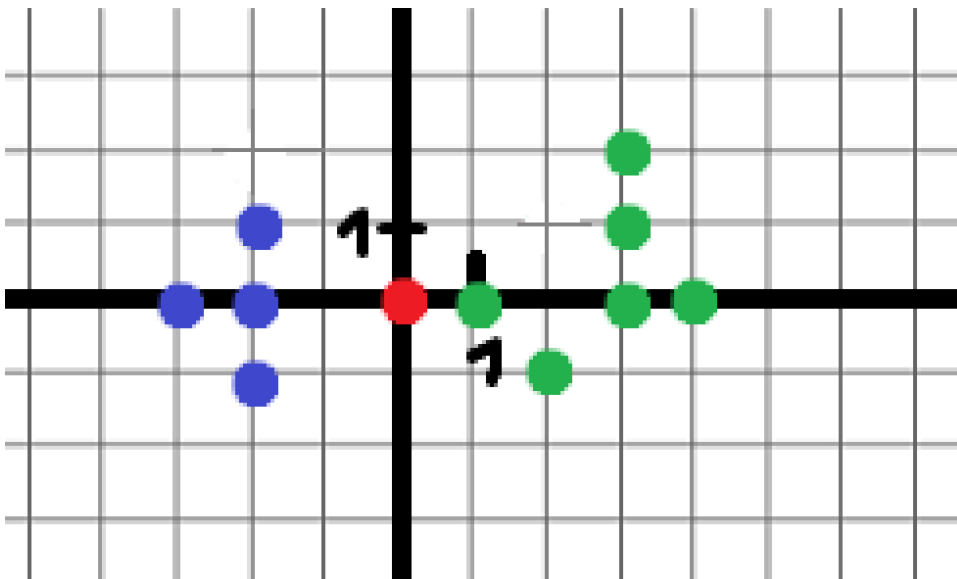
- Treningowy:

trening2d.txt — Notatnik

Plik Edycja Format Widok Pomoc

```
10 % liczba punktów treningowych
2 % liczba wymiarów D
1.0 0.0 zielony %punkt 1
3.0 0.0 zielony
2.0 -1.0 zielony
3.0 1.0 zielony
3.0 2.0 zielony
4.0 0.0 zielony
-2.0 0.0 niebieski
-2.0 1.0 niebieski
-2.0 -1.0 niebieski
-3.0 0.0 niebieski
```

- Jak te punkty wyglądają na płaszczyźnie:



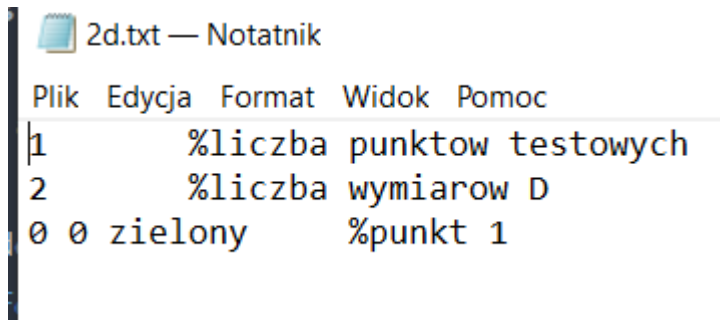
Kolorem czerwonym zaznaczony jest punkt testowy, natomiast kolory reszty punktów to klasy w jakich się znajdują. Widzimy w ten sposób, że punkt testowy może zostać przyporządkowany kolejno do klasy niebieskiej lub zielonej. Widać także, że dla  $k=1$  punkt testowy zostanie dodany do klasy zielonej, gdyż najbliższy punkt należy do tej klasy. Natomiast dla  $k=5$  punkt powinien zostać dodany do klasy niebieskiej, gdyż będziemy mieli wtedy sytuację, gdzie 3 punkty najbliższe testowemu należą do niebieskiej klasy, a 2 do zielonej. Sprawdźmy to zatem.

Dla  $k=1$  mamy polecenie:

```
|-train trening2d.txt -test test2d.txt -out 2d.txt -k 1
```

I plik wynikowy:





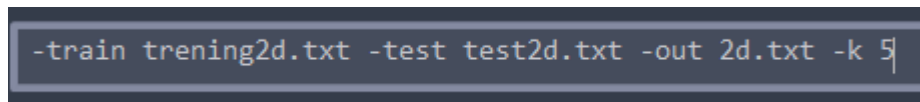
```

2d.txt — Notatnik
Plik  Edycja  Format  Widok  Pomoc
1      %liczba punktow testowych
2      %liczba wymiarow D
0 0 zielony      %punkt 1

```

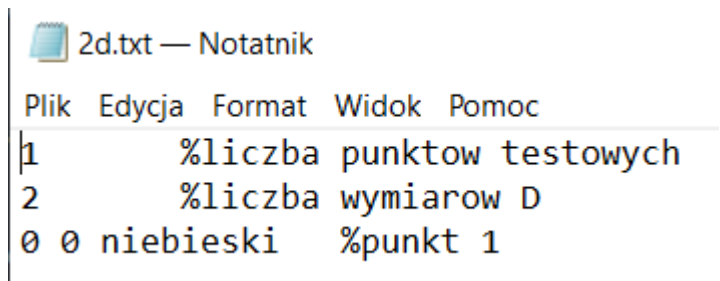
Czyli tak jak prognozowaliśmy.

Natomiast dla  $k = 5$  mamy polecenie:



```
-train trening2d.txt -test test2d.txt -out 2d.txt -k 5
```

I plik wynikowy:



```

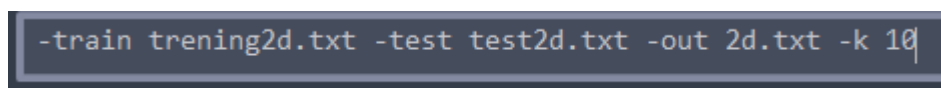
2d.txt — Notatnik
Plik  Edycja  Format  Widok  Pomoc
1      %liczba punktow testowych
2      %liczba wymiarow D
0 0 niebieski      %punkt 1

```

Czyli również zgodnie z prognozami.

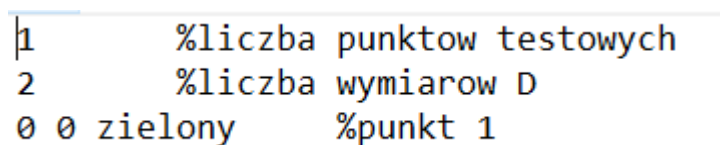
Łatwo również określić to, że skoro punktów zielonych jest więcej niż niebieskich (zielonych 6, a niebieskich 4), to dla maksymalnej wartości  $k = 10$ , punkt testowy powinien zostać dodany do zielonej klasy.

Dla  $k=10$  polecenie:



```
-train trening2d.txt -test test2d.txt -out 2d.txt -k 10
```

I plik wyjściowy:



```

2d.txt — Notatnik
Plik  Edycja  Format  Widok  Pomoc
1      %liczba punktow testowych
2      %liczba wymiarow D
0 0 zielony      %punkt 1

```

Czyli także zgodnie z naszymi domysłami.

## 5.2 Test na trójwymiarowej przestrzeni.

Dla plików:

- Testowego:

```
testowe.txt — Notatnik
Plik Edycja Format Widok Pomoc
10      %liczba punktów testowych
3      %liczba wymiarów D
1.5 12.3 4.5
4 7.6 12.7
0.2 -0.9 17.8
0.0 0.0 0.0
-2.0 8.9 6.9
-0.5 3.9 2.1
10.5 10.5 3.7
-10 -10 666
6.4 3.9 888
1.5 21.37 919
```

- Treningowego:

```
*treningowe.txt — Notatnik
Plik Edycja Format Widok Pomoc
20      %liczba punktów treningowych N
3      %liczba wymiarów D
%poniżej znajduje się lista n punktów danych, opisanych przez d współrzędnych
%oraz etykiety klasy będących łańcuchem tekstowym
7.6 5.3 1.3 klasa_A
45.3 4.6 -2 klasa_A
-3.7 22.9 -8 klasa_C
2.4 12.2 -16 klasa_A
38.3 -3.5 -23 klasa_A
-4.7 -9.9 66 klasa_A
0.1 0.1 0.1 klasa_A
-0.1 -0.1 -0.1 klasa_B
0.1 -0.1 0.1 klasa_B
-0.1 0.1 -0.1 klasa_C
5.0 5.0 5.0 klasa_A
2.2 -14.3 2.2 klasa_A
7.3 16.8 7.3 klasa_C
23.0 12.5 -10.0 klasa_C
6.9 6.9 -19 klasa_C
42.0 0.42 -16 klasa_B
21.37 -19.39 -2.2 klasa_C
-22.22 11.11 1.1 klasa_C
6.3 3.3 22.22 klasa_A
4.3 2.6 69.69 klasa_C
```

Nie skupiamy się na wszystkich punktach. Można natomiast łatwo zauważyć, że jeden z punktów testowych to początek układu współrzędnych i są cztery punkty treningowe bardzo mu bliskie. Na tym przykładzie widać, że dla  $k=4$  punkt testowy 0,0,0 powinien otrzymać klasę B.

Więc dla polecenia:

```
-train treningowe.txt -test testowe.txt -out wyjscie.txt -k 4
```

Plik wyjściowy:

```
wyjscie.txt — Notatnik
Plik Edycja Format Widok Pomoc
10      %liczba punktow testowych
3        %liczba wymiarow D
1.5 12.3 4.5 klasa_A    %punkt 1
4 7.6 12.7 klasa_A    %punkt 2
0.2 0.9 17.8 klasa_A    %punkt 3
0 0 0 klasa_B    %punkt 4
-2 8.9 6.9 klasa_A    %punkt 5
-0.5 3.9 2.1 klasa_B    %punkt 6
10.5 10.5 3.7 klasa_A    %punkt 7
-10 -10 666 klasa_A    %punkt 8
6.4 3.9 888 klasa_A    %punkt 9
1.5 21.37 919 klasa_A    %punkt 10
```

Testowanie walidacji argumentów:

- Brak .txt

```
-train treningowe.txt -test testowe -out wyjscie.txt -k 4
```

Plik testowy zostal podany bez rozszerzenia .txt

- Zbyt duży argument k (plik treningowe.txt ma 20 punktów)

```
-train treningowe.txt -test testowe.txt -out wyjscie.txt -k 40
```

liczba k jest wieksza niz ilosc punktow treningowych

- Zbyt mała ilość argumentów

```
-train treningowe.txt -test testowe.txt -out wyjscie.txt
```

Liczba parametrow jest niewystarczajaca, sprobuj uruchomic program z przełącznikami:

```
-train (sciezka pliku z punktami treningowymi)
-test (sciezka pliku z punktami testowymi)
-out (nazwa pliku z punktami testowymi i ich klasami)
-k (liczba najblizszych sasiadow)
```

To kończy testy programu.

## 6 Wnioski

Klasyfikator KNN to stosunkowo prosty program oparty w całości prawie jedynie na tablicach alokowanych dynamicznie. Najbardziej wymagającym wydawało mi się zapisywanie współrzędnych punktów dla większej ilości wymiarów, jednak okazało się to łatwiejsze niż przypuszczałem. Trudności sprawiło mi natomiast znajdowanie indeksów najbliższych punktów i zapisywanie ich do mapy. Sam projekt dał mi dużo odnośnie samodzielnego rozwiązywania problemów i znajdowania błędów. Myślę też, że lepiej niż wcześniej analizuję kod. Jednak muszę przyznać, że gdyby nie laboratorium, często ciężko byłoby ruszyć z miejsca.

Dodatek.  
Szczegółowy opis funkcji  
i typów

## KNN CLASSIFIER

Generated by Doxygen 1.9.3



<b>1 File Index</b>	<b>1</b>
1.1 File List	1
<b>2 File Documentation</b>	<b>3</b>
2.1 functions.cpp File Reference	3
2.1.1 Function Documentation	3
2.1.1.1 additionalValid()	3
2.1.1.2 argInfo()	4
2.1.1.3 argValidate()	4
2.1.1.4 checkDistances()	5
2.1.1.5 deleteTables()	5
2.1.1.6 findTestClasses()	6
2.1.1.7 readData()	6
2.1.1.8 writeToFile()	7
2.2 functions.h File Reference	7
2.2.1 Function Documentation	8
2.2.1.1 additionalValid()	8
2.2.1.2 argInfo()	8
2.2.1.3 argValidate()	9
2.2.1.4 checkDistances()	10
2.2.1.5 deleteTables()	10
2.2.1.6 findTestClasses()	10
2.2.1.7 readData()	11
2.2.1.8 writeToFile()	12
2.3 functions.h	12
2.4 main.cpp File Reference	12
2.4.1 Function Documentation	13
2.4.1.1 main()	13
<b>Index</b>	<b>15</b>





# Chapter 1

## File Index

### 1.1 File List

Here is a list of all documented files with brief descriptions:

<a href="#">functions.cpp</a>	3
<a href="#">functions.h</a>	7
<a href="#">main.cpp</a>	12



## Chapter 2

# File Documentation

### 2.1 functions.cpp File Reference

```
#include <iostream>
#include <fstream>
#include <algorithm>
#include <sstream>
#include <map>
#include <vector>
#include "functions.h"
```

#### Functions

- int [argInfo](#) (int &argc, char \*argv[], string &addressTrain, string &addressTest, string &addressOut, int &k)
- int [argValidate](#) (const string &addressTest, const string &addressTrain, const string &addressOut)
- int [additionalValid](#) (int k, int trainingAmount)
- void [readData](#) (const string &addressTrain, const string &addressTest, int &dimensions, int &trainingAmount, int &testAmount, double \*\*&trainingPoints, double \*\*&testPoints, string \*&trainingPointClasses)
- void [checkDistances](#) (double \*\*&testPoints, double \*\*&trainingPoints, double \*\*&distances, int testAmount, int trainAmount, int dimensions)
- void [findTestClasses](#) (map< string, int > &classes, double \*\*&distances, string \*&testPointClasses, string \*&trainingPointClasses, int testAmount, int k, int trainAmount)
- void [writeToFile](#) (const string &address, string \*&testPointsClasses, double \*\*&testPoints, int testAmount, int dimensions)
- void [deleteTables](#) (string \*&testPointClasses, string \*&trainingPointClasses, double \*\*&testPoints, double \*\*&trainingPoints, double \*\*&distances)

#### 2.1.1 Function Documentation

Zawiera instrukcje wszystkich funkcji programu.

##### 2.1.1.1 additionalValid()

```
int additionalValid (
    int k,
    int trainingAmount )
```

Dodatkowa walidacja argumentu k, który nie może być większy od ilości punktów treningowych

**Parameters**

<i>k</i>	ilość k najbliższych sąsiadów
<i>trainingAmount</i>	ilość punktów treningowych

**Returns**

zwraca błąd i kończy program jeśli k jest większe od ilości punktów treningowych

**2.1.1.2 argInfo()**

```
int argInfo (
    int & argc,
    char * argv[],
    string & addressTrain,
    string & addressTest,
    string & addressOut,
    int & k )
```

Funkcja przyporządkowuje wartości a tablicy argv odpowiednim zmiennym które będą używane w całym programie

**Parameters**

<i>argc</i>	ilość argumentów podanych przez użytkownika
<i>argv</i>	tablica z argumentami podanymi przez użytkownika
<i>addressTrain</i>	adres pliku z punktami treningowymi podany przez użytkownika
<i>addressTest</i>	adres pliku z punktami testowymi podany przez użytkownika
<i>addressOut</i>	adres pliku wyjściowego wybranego przez użytkownika
<i>k</i>	ilość najbliższych sąsiadów służąca do dalszej klasyfikacji

**Returns**

jeśli ilość argumentów jest odpowiednia to po wykonaniu funkcji zwrócone zostaną wypełnione zmienne, jeśli nie to funkcja zwróci błąd

**2.1.1.3 argValidate()**

```
int argValidate (
    const string & addressTest,
    const string & addressTrain,
    const string & addressOut )
```

Funkcja odpowiada za prostą walidację argumentów podanych przez użytkownika

Sprawdza najczęstsze błędy związane z niepodaniem rozszerzenia pliku

## Parameters

<i>addressTest</i>	adres pliku z punktami testowymi
<i>addressTrain</i>	adres pliku z punktami treningowymi
<i>addressOut</i>	adres pliku wyjściowego

## Returns

funkcja kończy się bez błędu jeśli wszystkie pliki mają rozszerzenie txt, jeśli pojawia się błąd - program przerywa się

## 2.1.1.4 checkDistances()

```
void checkDistances (
    double **& testPoints,
    double **& trainingPoints,
    double **& distances,
    int testAmount,
    int trainAmount,
    int dimensions )
```

Funkcja sprawdza odległość każdego punktu testowego do każdego z punktów treningowych za pomocą metody euklidesowej

Odległości zapisywane są do tablicy distances która ma wielkość ilość punktów testowych na ilość punktów treningowych

## Parameters

<i>testPoints</i>	tablica zawierająca współrzędne punktów testowych
<i>trainingPoints</i>	tablica zawierające współrzędne punktów treningowych
<i>distances</i>	tablica zawierająca odległości kolejnych punktów testowych od każdego punktu treningowego
<i>testAmount</i>	ilość punktów testowych
<i>trainAmount</i>	ilość punktów treningowych
<i>dimensions</i>	ilość wymiarów D

## 2.1.1.5 deleteTables()

```
void deleteTables (
    string *& testPointClasses,
    string *& trainingPointClasses,
    double **& testPoints,
    double **& trainingPoints,
    double **& distances )
```

Funkcja usuwa wszystkie tablice zaalokowane wcześniej dynamicznie

## Parameters

<i>testPointClassess</i>	tablica klas punktów testowych
<i>trainingPointClassess</i>	tablica klas punktów treningowych
<i>testPoints</i>	talica współrzędnych punktów testowych
<i>trainingPoints</i>	tablica współrzędnych punktów treningowych
<i>distances</i>	tablica dystansów kolejnych punktów testowych od wszystkich punktów treningowych

## 2.1.1.6 findTestClasses()

```
void findTestClasses (
    map< string, int > & classes,
    double **& distances,
    string *& testPointClasses,
    string *& trainingPointClasses,
    int testAmount,
    int k,
    int trainAmount )
```

Funkcja wypełnia tablicę klas punktów testowych na podstawie k najbliższych punktów treningowych względem kolejnych punktów testowych, klasy zapisywane są do mapy classes

Na podstawie ilości wystąpień wybranych klas, ta która ma największą wartość w mapie przyporządkowywana jest do wybranego punktu testowego

## Parameters

<i>classes</i>	mapa do której zapisywane będą klasy k najbliższych punktów treningowych do kolejnych punktów testowych
<i>distances</i>	tablica odległości kolejnych punktów testowych od każdego punktu treningowego
<i>testPointClasses</i>	tablica klas punktów testowych, którą wypełnia ta funkcja
<i>trainingPointClasses</i>	tablica klas punktów treningowych
<i>testAmount</i>	ilość punktów testowych
<i>k</i>	liczba k najbliższych sąsiadów (w tym przypadku najbliższych, względem kolejnych punktów testowych, punktów treningowych)
<i>trainAmount</i>	liczba punktów treningowych

## 2.1.1.7 readData()

```
void readData (
    const string & addressTrain,
    const string & addressTest,
    int & dimensions,
    int & trainingAmount,
    int & testAmount,
```

```
double **& trainingPoints,
double **& testPoints,
string *& trainingPointClasses )
```

Funkcja czytuje współrzędne punktów treningowych i testowych z odpowiednich plików

Po czytaniu danych zapisywane są one do odpowiednich tablic

#### Parameters

<i>addressTrain</i>	adres pliku z punktami treningowymi
<i>addressTest</i>	adres pliku z punktami testowymi
<i>dimensions</i>	liczba wymiarów D w których zapisane są wszystkie punkty
<i>trainingAmount</i>	ilość punktów treningowych
<i>testAmount</i>	ilość punktów testowych
<i>trainingPoints</i>	tablica D wymiarowa zawierająca współrzędne wszystkich punktów treningowych (trainingAmount x dimensions)
<i>testPoints</i>	tablica D wymiarowa zawierająca współrzędne wszystkich punktów testowych (testAmount x dimensions)
<i>trainingPointClasses</i>	tablica jednowymiarowa stringów zawierająca klasy punktów treningowych

#### 2.1.1.8 writeToFile()

```
void writeToFile (
    const string & address,
    string *& testPointsClasses,
    double **& testPoints,
    int testAmount,
    int dimensions )
```

Funkcja wypisuje informacje do pliku w sposób podobny do pliku z punktami testowymi, lecz w tym wypadku każdemu z punktów przyporządkowuje ona klasę dobraną w wyniku funkcji [findTestClasses\(\)](#)

#### Parameters

<i>address</i>	adres pliku do którego zostaną wypisane informacje
<i>testPointsClasses</i>	tablica z klasami punktów testowych
<i>testPoints</i>	tablica z współrzędnymi punktów testowych
<i>testAmount</i>	ilość punktów testowych
<i>dimensions</i>	ilość wymiarów D

## 2.2 functions.h File Reference

```
#include <iostream>
#include <fstream>
#include <algorithm>
```



```
#include <sstream>
#include <map>
```

## Functions

- int [argInfo](#) (int &argc, char \*argv[], string &addressTrain, string &addressTest, string &addressOut, int &k)
- int [argValidate](#) (const string &addressTest, const string &addressTrain, const string &addressOut)
- int [additionalValid](#) (int k, int trainingAmount)
- void [readData](#) (const string &addressTrain, const string &addressTest, int &dimensions, int &trainingAmount, int &testAmount, double \*\*&trainingPoints, double \*\*&testPoints, string \*&trainingPointClasses)
- void [checkDistances](#) (double \*\*&testPoints, double \*\*&trainingPoints, double \*\*&distances, int testAmount, int trainAmount, int dimensions)
- void [findTestClasses](#) (map< string, int > &classes, double \*\*&distances, string \*&testPointClasses, string \*&trainingPointClasses, int testAmount, int k, int trainAmount)
- void [writeToFile](#) (const string &address, string \*&testPointsClasses, double \*\*&testPoints, int testAmount, int dimensions)
- void [deleteTables](#) (string \*&testPointClasses, string \*&trainingPointClasses, double \*\*&testPoints, double \*\*&trainingPoints, double \*\*&distances)

### 2.2.1 Function Documentation

Zawiera nagłówki wszystkich funkcji programu.

#### 2.2.1.1 additionalValid()

```
int additionalValid (
    int k,
    int trainingAmount )
```

Dodatkowa walidacja argumentu k, który nie może być większy od ilości punktów treningowych

##### Parameters

<i>k</i>	ilość k najbliższych sąsiadów
<i>trainingAmount</i>	ilość punktów treningowych

##### Returns

zwraca błąd i kończy program jeśli k jest większe od ilości punktów treningowych

#### 2.2.1.2 argInfo()

```
int argInfo (
    int & argc,
    char * argv[],
```

```
string & addressTrain,  
string & addressTest,  
string & addressOut,  
int & k )
```

Funkcja przyporządkowuje wartości a tablicy argv odpowiednim zmiennym które będą używane w całym programie

#### Parameters

<i>argc</i>	ilość argumentów podanych przez użytkownika
<i>argv</i>	tablica z argumentami podanymi przez użytkownika
<i>addressTrain</i>	adres pliku z punktami treningowymi podany przez użytkownika
<i>addressTest</i>	adres pliku z punktami testowymi podany przez użytkownika
<i>addressOut</i>	adres pliku wyjściowego wybranego przez użytkownika
<i>k</i>	ilość najbliższych sąsiadów służąca do dalszej klasyfikacji

#### Returns

jeśli ilość argumentów jest odpowiednia to po wykonaniu funkcji zwrócone zostaną wypełnione zmienne, jeśli nie to funkcja zwróci błąd

### 2.2.1.3 argValidate()

```
int argValidate (  
    const string & addressTest,  
    const string & addressTrain,  
    const string & addressOut )
```

Funkcja odpowiada za prostą walidację argumentów podanych przez użytkownika

Sprawdza najczęstsze błędy związane z niepodaniem rozszerzenia pliku

#### Parameters

<i>addressTest</i>	adres pliku z punktami testowymi
<i>addressTrain</i>	adres pliku z punktami treningowymi
<i>addressOut</i>	adres pliku wyjściowego

#### Returns

funkcja kończy się bez błędu jeśli wszystkie pliki mają rozszerzenie txt, jeśli pojawia się błąd - program przerywa się

#### 2.2.1.4 checkDistances()

```
void checkDistances (
    double **& testPoints,
    double **& trainingPoints,
    double **& distances,
    int testAmount,
    int trainAmount,
    int dimensions )
```

Funkcja sprawdza odległość każdego punktu testowego do każdego z punktów treningowych za pomocą metody euklidesowej

Odległości zapisywane są do tablicy distances która ma wielkość ilość punktów testowych na ilość punktów treningowych

##### Parameters

<i>testPoints</i>	tablica zawierająca współrzędne punktów testowych
<i>trainingPoints</i>	tablica zawierające współrzędne punktów treningowych
<i>distances</i>	tablica zawierająca odległości kolejnych punktów testowych od każdego punktu treningowego
<i>testAmount</i>	ilość punktów testowych
<i>trainAmount</i>	ilość punktów treningowych
<i>dimensions</i>	ilość wymiarów D

#### 2.2.1.5 deleteTables()

```
void deleteTables (
    string *& testPointClasses,
    string *& trainingPointClasses,
    double **& testPoints,
    double **& trainingPoints,
    double **& distances )
```

Funkcja usuwa wszystkie tablice zaalokowane wcześniej dynamicznie

##### Parameters

<i>testPointClassess</i>	tablica klas punktów testowych
<i>trainingPointClassess</i>	tablica klas punktów treningowych
<i>testPoints</i>	talica współrzędnych punktów testowych
<i>trainingPoints</i>	tablica współrzędnych punktów treningowych
<i>distances</i>	tablica dystansów kolejnych punktów testowych od wszystkich punktów treningowych

#### 2.2.1.6 findTestClasses()

```
void findTestClasses (
```

```

map< string, int > & classes,
double **& distances,
string *& testPointClasses,
string *& trainingPointClasses,
int testAmount,
int k,
int trainAmount )

```

Funkcja wypełnia tablicę klas punktów testowych na podstawie k najbliższych punktów treningowych względem kolejnych punktów testowych, klasy zapisywane są do mapy classes

Na podstawie ilości wystąpień wybranych klas, ta która ma największą wartość w mapie przyporządkowywana jest do wybranego punktu testowego

#### Parameters

<i>classes</i>	mapa do której zapisywane będą klasy k najbliższych punktów treningowych do kolejnych punktów testowych
<i>distances</i>	tablica odległości kolejnych punktów testowych od każdego punktu treningowego
<i>testPointClasses</i>	tablica klas punktów testowych, którą wypełnia ta funkcja
<i>trainingPointClasses</i>	tablica klas punktów treningowych
<i>testAmount</i>	ilość punktów testowych
<i>k</i>	liczba k najbliższych sąsiadów (w tym przypadku najbliższych, względem kolejnych punktów testowych, punktów treningowych)
<i>trainAmount</i>	liczba punktów treningowych

#### 2.2.1.7 readData()

```

void readData (
    const string & addressTrain,
    const string & addressTest,
    int & dimensions,
    int & trainingAmount,
    int & testAmount,
    double **& trainingPoints,
    double **& testPoints,
    string *& trainingPointClasses )

```

Funkcja czytuje współrzędne punktów treningowych i testowych z odpowiednich plików

Po czytaniu danych zapisywane są one do odpowiednich tablic

#### Parameters

<i>addressTrain</i>	adres pliku z punktami treningowymi
<i>addressTest</i>	adres pliku z punktami testowymi
<i>dimensions</i>	liczba wymiarów D w których zapisane są wszystkie punkty
<i>trainingAmount</i>	ilość punktów treningowych
<i>testAmount</i>	ilość punktów testowych
<i>trainingPoints</i>	tablica D wymiarowa zawierająca współrzędne wszystkich punktów treningowych (trainingAmount x dimensions)
<i>testPoints</i>	tablica D wymiarowa zawierająca współrzędne wszystkich punktów testowych (testAmount x dimensions)
<i>trainingPointClasses</i>	tablica jednowymiarowa stringów zawierająca klasy punktów treningowych

### 2.2.1.8 writeToFile()

```
void writeToFile (
    const string & address,
    string *& testPointsClasses,
    double **& testPoints,
    int testAmount,
    int dimensions )
```

Funkcja wypisuje informacje do pliku w sposób podobny do pliku z punktami testowymi, lecz w tym wypadku każdemu z punktów przyporządkowuje ona klasę dobraną w wyniku funkcji [findTestClasses\(\)](#)

#### Parameters

<i>address</i>	adres pliku do którego zostaną wypisane informacje
<i>testPointsClasses</i>	tablica z klasami punktów testowych
<i>testPoints</i>	tablica z współrzędnymi punktów testowych
<i>testAmount</i>	ilość punktów testowych
<i>dimensions</i>	ilość wymiarów D

## 2.3 functions.h

[Go to the documentation of this file.](#)

```
1
2 //
3 // Created by Michin on 28.11.2021.
4 //
5 #ifndef PROJEKT_FUNCTIONS_H
6 #define PROJEKT_FUNCTIONS_H
7 #include <iostream>
8 #include <fstream>
9 #include <algorithm>
10 #include <sstream>
11 #include <map>
12 using namespace std;
13 int argInfo(int &argc, char * argv[], string &addressTrain, string &addressTest, string &addressOut, int
    &k);
14
15 int argValidate(const string &addressTest, const string &addressTrain, const string &addressOut);
16 int additionalValid(int k, int trainingAmount);
17 void readData(const string &addressTrain, const string &addressTest, int &dimensions, int
    &trainingAmount, int &testAmount, double ** &trainingPoints, double ** &testPoints, string *
    &trainingPointClasses);
18 void checkDistances(double ** &testPoints, double ** &trainingPoints, double ** &distances, int
    testAmount, int trainAmount, int dimensions);
19 void findTestClasses(map<string, int> &classes, double ** &distances, string * &testPointClasses, string
    * &trainingPointClasses, int testAmount, int k, int trainAmount);
20 void writeToFile(const string &address, string * &testPointsClasses, double ** &testPoints, int
    testAmount, int dimensions);
21 void deleteTables(string * &testPointClasses, string * &trainingPointClasses, double ** &testPoints,
    double ** &trainingPoints, double ** &distances);
22 #endif //PROJEKT_FUNCTIONS_H
```

## 2.4 main.cpp File Reference

```
#include <iostream>
#include <fstream>
```

```
#include <algorithm>
#include <sstream>
#include <map>
#include "functions.h"
```

## Functions

- `int main (int argc, char *argv[ ])`

### 2.4.1 Function Documentation

Główny plik programu.

#### 2.4.1.1 main()

```
int main (
    int argc,
    char * argv[ ] )
```

Główna funkcja programu.

##### Parameters

<i>argc</i>	ilość argumentów podanych przez użytkownika
<i>argv</i>	tablica z argumentami



# Index

- additionalValid
  - functions.cpp, [3](#)
  - functions.h, [8](#)
- argInfo
  - functions.cpp, [4](#)
  - functions.h, [8](#)
- argValidate
  - functions.cpp, [4](#)
  - functions.h, [9](#)
- checkDistances
  - functions.cpp, [5](#)
  - functions.h, [9](#)
- deleteTables
  - functions.cpp, [5](#)
  - functions.h, [10](#)
- findTestClasses
  - functions.cpp, [6](#)
  - functions.h, [10](#)
- functions.cpp, [3](#)
  - additionalValid, [3](#)
  - argInfo, [4](#)
  - argValidate, [4](#)
  - checkDistances, [5](#)
  - deleteTables, [5](#)
  - findTestClasses, [6](#)
  - readData, [6](#)
  - writeToFile, [7](#)
- functions.h, [7](#)
  - additionalValid, [8](#)
  - argInfo, [8](#)
  - argValidate, [9](#)
  - checkDistances, [9](#)
  - deleteTables, [10](#)
  - findTestClasses, [10](#)
  - readData, [11](#)
  - writeToFile, [12](#)
- main
  - main.cpp, [13](#)
- main.cpp, [12](#)
  - main, [13](#)
- readData
  - functions.cpp, [6](#)
  - functions.h, [11](#)
- writeToFile
  - functions.cpp, [7](#)
  - functions.h, [12](#)