Umelá inteligencia

Zadanie 3a – klasifikácia

Tomáš Brček

FIIT STU, Bratislava

2023/2024

Obsah

Zadanie

Máme 2D priestor, ktorý má rozmery v intervaloch od -5000 do +5000. V tomto priestore sa nachádzajú body, pričom každý bod má určenú polohu pomocou súradníc X a Y. Každý bod má unikátne súradnice, pričom patrí do jednej zo 4 tried, tieto triedy sú: red (R), green (G), blue (B) a purple (P). Na začiatku sa v priestore nachádza 5 bodov pre každú triedu. Súradnice počiatočných bodov sú:

R: [-4500, -4400], [-4100, -3000], [-1800, -2400], [-2500, -3400] a [-2000, -1400]

G: [+4500, -4400], [+4100, -3000], [+1800, -2400], [+2500, -3400] a [+2000, -1400]

B: [-4500, +4400], [-4100, +3000], [-1800, +2400], [-2500, +3400] a [-2000, +1400]

P: [+4500, +4400], [+4100, +3000], [+1800, +2400], [+2500, +3400] a [+2000, +1400]

Úlohou je naprogramovať klasifikátor pre nové body, teda funkciu *classify(int X, int Y, int k)*, ktorá klasifikuje nový bod so súradnicami X a Y, pridá tento bod do 2D priestoru a vráti triedu, ktorú pridelila pre tento bod.

Súradnice nových bodov sú generované náhodne, pričom nový bod má zakaždým inú triedu:

- R body by mali byť generované s 99% pravdepodobnosťou s X < +500 a Y < +500
- G body by mali byť generované s 99% pravdepodobnosťou s X > -500 a Y < +500
- B body by mali byť generované s 99% pravdepodobnosťou s X < +500 a Y > -500
- P body by mali byť generované s 99% pravdepodobnosťou s X > -500 a Y > -500

(Zvyšné jedno percento bodov je generované v celom priestore.)

Je vhodné využiť nejaké optimalizácie na zredukovanie zložitosti:

- Pre hľadanie k najbližších bodov si môžeme rozdeliť plochu na viaceré menšie štvorce, do ktorých umiestňujeme body s príslušnými súradnicami, aby sme nemuseli vždy porovnávať všetky body, ale len body vo štvorci, kde sa nachádza aktuálny bod a susedných štvorcoch.
- Je možné tiež využiť len základnú plochu a vyhľadávať k najbližších len z bodov, ktoré sú v nejakej vzdialenosti od klasifikovaného bodu. Či už je to ako kružnica alebo štvorec. Počiatočnú veľkosť kružnice/štvorca volíme podľa množstva bodov, ktoré sú aktuálne v priestore. Kružnica by mala obsahovať aspoň k bodov, štvorec aspoň 2k, aby to bolo určite k najbližších.
- Je tiež možné využiť algoritmus na hľadanie práve k najmenších hodnôt.
- Na hľadanie najbližších susedov v dvojrozmernom priestore sú vhodné aj k-d stromy.
- PyPy je implementácia programovacieho jazyka Python. PyPy často beží rýchlejšie ako štandardná implementácia CPython, pretože PyPy používa just-in-time kompilátor. Pypy nepodporuje niektoré grafické knižnice.

kNN algoritmus

kNN, čo znamená k-najbližších susedov, je jednoduchý algoritmus strojového učenia, používaný predovšetkým pre klasifikáciu a regresiu. Jeho základným princípom je určiť triedu alebo hodnotu neznámeho príkladu na základe jeho podobnosti s príkladmi zo vstupných dát.

Princíp fungovania algoritmu je nasledovný:

- **Výber hodnoty k**: Definuje sa počet najbližších susedov (k), ktoré budú použité na rozhodovanie. Táto hodnota sa väčšinou volí experimentálne alebo na základe znalostí o dátach.
- **Vzdialenostná metrika**: Určuje, ako sa meria podobnosť medzi príkladmi. Najčastejšie používanou metrikou je euklidovská vzdialenosť, ale môžu sa používať aj iné, ako napríklad Manhattanova vzdialenosť.
- **Nájdenie najbližších susedov**: Pre každý neznámy príklad sa vypočíta jeho vzdialenosť od všetkých príkladov v množine. Potom sa vyberie k príkladov s najmenšou vzdialenosťou.
- **Výber triedy**: V prípade klasifikácie sa rozhoduje na základe tried najbližších susedov väčšinovým hlasovaním.

KNN je relatívne jednoduchý a intuitívny algoritmus, ale jeho úspešnosť môže byť ovplyvnená voľbou vhodnej hodnoty k a vhodnej vzdialenostnej metriky. Taktiež môže byť citlivý na šum v dátach a vyžaduje uchovávanie celej množiny, čo môže byť neefektívne v prípade veľkých datasetov.

Moja implementácia

2D plocha

Celú 2D plochu som si rozdelil na 400 menších štvorčekov, ktorých rozmery sú 500x500. Každý z týchto štvorčekov je reprezentovaný triedou *Square*, v ktorej sú uložené všetky body, ktoré patria do tohto štvorca.

Toto rozdelenie plochy som sa rozhodol urobiť preto, aby som znížil náročnosť môjho riešenia, keďže nebude potrebné vypočítavať vzdialenosť so všetkými bodmi v priestore, ale len v aktuálnom štvorci a vo všetkých susedných štvorcoch. V prípade, že sa v týchto štvorcoch nachádza menej ako k bodov, vzdialenosti sú počítané pre všetky body v priestore.

```
#inicializácia gridu
def initialize_grid():
    global grid
    #rozdeľenie gridu na 400 malých štvorcov veľkosti 500x500
    grid = [
        [Square(x, y, x + 500, y + 500) for x in range(-5000, 5000, 500)]
        for y in range(-5000, 5000, 500)
    ]
    #naplnenie gridu počiatočnými bodmi
    for point in [RP1,RP2,RP3,RP4,RP5,GP1,GP2,GP3,GP4,GP5,BP1,BP2,BP3,BP4,BP5,PP1,PP2,PP3,PP4,PP5]:
        sq = find_square(point)
        grid[sq[0]][sq[1]].add_point(point)
```

Na začiatku si inicializujem mriežku, rozdelím si ju na menšie štvorčeky a vložím tam všetky počiatočné body do príslušných štvorčekov, ktorých súradnice zisťujem vo funkcií *find_square()*:

```
def find_square(point):
    x = point.x
    y = point.y
    if y < 0:
        y = 4999 + abs(y)
    else:
        y = 4999 - y
    if x < 0:
        x = 5000 - abs(x)
    else:
        x = 5000 + x
    return (min(y//500,19), min(x//500,19))
```

Triedy

1. **Square** – trieda reprezentujúca jednotlivé štvorce v mriežke

```python
class Square:
    #konštruktor triedy
    def __init__(self, min_x, max_x, min_y, max_y):
        self.min_x = min_x
        self.max_x = max_x
        self.min_y = min_y
        self.max_y = max_y
        self.points = []

    #pridanie bodu do daného štvorca
    def add_point(self, point):
        self.points.append(point)
```

- atribúty:
  - *min_x* – dolná hranica na osi x
  - *max_x* – horná hranica na osi x
  - *min_y* – dolná hranica na osi y
  - *max_y* – horná hranica na osi y
  - *points* – pole bodov nachádzajúcich sa v danom štvorci
- metódy
  - *add_point()* – vloží do poľa bodov nový bod

2. **Point** – trieda reprezentujúca jednotlivé body v mriežke

```python
class Point:
    #konštruktor triedy
    def __init__(self, x, y, color):
        self.x = x
        self.y = y
        self.color = color

    #definícia, kedy sa dva Pointy rovnajú
    def __eq__(self, Point):
        if self.x == Point.x and self.y == Point.y and self.color == Point.color:
            return True
        return False

    #stringová reprezentácia triedy Point
    def __str__(self):
        return f"Point({self.x}, {self.y}, \"{self.color}\")"
```

- atribúty:
  - *x* – x-ová súradnica
  - *y* – y-ová súradnica
  - *color* – farba bodu
- metódy:
  - *__eq__()* – definícia, kedy sa dva body rovnajú
  - *__str__()* – stringová reprezentácia triedy Point

Generovanie bodov

Na generovanie bodov slúži funkcia *generate_points(n, min_x, max_x, min_y, max_y, color)*, ktorá vráti pole bodov zo zadaného intervalu.

```
#funkcia, ktorá generuje n bodov v zadanom rozmedzí bodov, ktorým priradí zadanú farbu
def generate_points(n, min_x, max_x, min_y, max_y, color):
    points = []
    #while-cyklus, ktorý beží až kým nie je počet bodov n
    while len(points) != n:
        #s pravdepodobnosťou 99% sú body generované zo zadaného intervalu
        if random() < 0.99:
            x_coordinate = randint(min_x, max_x)
            y_coordinate = randint(min_y, max_y)
        #s pravdepodobnosťou 1% sú body generované z celého priestoru
        else:
            x_coordinate = randint(-5000, 5000)
            y_coordinate = randint(-5000, 5000)
        new_point = Point(x_coordinate, y_coordinate, color)        #vytvorenie bodu
        #ak sa v poli nenachádza bod na tých istých súradniciach, vloží sa tam
        if new_point not in points:
            points.append(new_point)
    return points
```

Parametre:

- *n* – počet bodov na generovanie z daného intervalu
- *min_x, max_x* – interval na osi x
- *min_y, max_y* – interval na osi y
- *color* – farba bodu

V tejto funkcií sa generujú body tak, že s 99% pravdepodobnosťou sú body generované z intervalu, ktorý je zadaný ako parameter funkcie a s pravdepodobnosťou 1% sú generované z celého priestoru, teda <-5000, 5000>.

Následne sa kontroluje, či už bod s danými súradnicami nebol vygenerovaný, aby som zaručil podmienku zo zadania.

Po vygenerovaní všetkých potrebných bodov sa zavolá funkcia *mix_arrays().*

```
#vygenerovanie COUNT bodov pre každú z farieb
red_points.extend(generate_points(COUNT, -5000, 499, -5000, 499, "red"))
green_points.extend(generate_points(COUNT, -500, 5000, -5000, 499, "green"))
blue_points.extend(generate_points(COUNT, -5000, 499, -500, 5000, "blue"))
purple_points.extend(generate_points(COUNT, -500, 5000, -500, 5000, "purple"))

#zmiešanie polí
all_points = mix_arrays(red_points, green_points, blue_points, purple_points)
```

Funkcia *mix_arrays()* slúži na zmiešanie všetkých bodov tak, aby nikdy nenasledovali za sebou dva body z rovnakého intervalu.

Vytvorím si pole *values*, do ktorého vložím i-tu hodnotu z každého poľa a z nich náhodne vyberám nasledujúcu hodnotu, až kým nie sú vybraté všetky hodnoty z tohto poľa. Poslednú vybranú hodnotu si ukladám do premennej *last_selected*, ktorá slúži na to, keď zvýšime i, aby sme nezobrali tú istú hodnotu, ktorá bola posledná.

```python
#funkcia, ktora zmieša polia všetkých farieb tak, aby nikdy nešli po sebe 2 body rovnakej farby
def mix_arrays(arr1, arr2, arr3, arr4):
    mixed_array = []
    last_selected = None

    for i in range(len(arr1)):
        values = [arr1[i], arr2[i], arr3[i], arr4[i]]       #uloženie i-tych prvkov zo všetkých polí
        #vyprázdňovanie poľa values
        while len(values) != 0:
            chosen = choice(values)     #náhodný výber prvku z poľa values
            #kontrola, či táto farba nebola posledná
            while chosen.color == last_selected:
                chosen = choice(values)
            values.pop(values.index(chosen))    #odstránenie vybraného prvku z poľa values
            mixed_array.append(chosen)          #vloženie do zmiešaného poľa
            last_selected = chosen.color

    return mixed_array
```

kNN algoritmus

Moja implementácia kNN algoritmu sa nachádza vo funkcií *classify(x, y, k):*

```python
def classify(x, y, k):
    distances = []
    #počítanie počtu susedov z jednotlivých farieb
    red_count = 0
    green_count = 0
    blue_count = 0
    purple_count = 0

    #nájdenie všetkých bodov zo všetkých susedných štvorcov, s ktorými sa bude počítať vzdialenosť
    all_p = []
    for row, col in neigh:
        all_p.extend(grid[row][col].points)

    #ak je počet bodov v susedných štvorcoch menší ako k, vzdialenosti sa počítajú so všetkými bodmi v priestore
    if len(all_p) < k:
        all_p = []
        for i in range(19):
            for j in range(19):
                all_p.extend(grid[i][j].points)

    #počítanie vzdialeností pre všetky body v poli all_p
    for point in all_p:
        distance = math.sqrt((x-point.x)**2 + (y-point.y)**2)
        distances.append((distance, point.color))

    #zoradenie vzdialeností od najmenšej po najväčšiu
    distances = sorted(distances, key=lambda x: x[0])

    #spočítanie farieb pre k najbližších susedov
    for i in range(k):
        color = distances[i][1]
        if color == "red":
            red_count += 1
        elif color == "green":
            green_count += 1
        elif color == "blue":
            blue_count += 1
        else:
            purple_count += 1

    #nájdenie najväčšieho počtu a vrátenie prislúchajúcej farby
    max_colors = sorted((("red", red_count), ("green", green_count), ("blue", blue_count), ("purple", purple_count)), key=lambda x: x[1], reverse=True)
    max_count = max_colors[0][1]
    #v prípade, že viac farieb má rovnaký počet bodov
    if max_colors[1][1] == max_count:
        for _, nearest_color in distances:
            for color, count in max_colors:
                if color == nearest_color:
                    if count == max_count:
                        return nearest_color

    return max_colors[0][0]
```

Parametre:

- *x* – x-ová súradnica bodu
- *y* – y-ová súradnica bodu
- *k* – počet susedov

Vo funkcií si najskôr do poľa *all_p* vložím všetky body, s ktorými sa bude vypočítavať vzdialenosť od aktuálneho bodu. Tieto body sa vyberajú z aktuálneho štvorčeka a zo všetkých susedných. V prípade, že počet týchto bodov je menší ako k, do poľa *all_p* sa vložia všetky body v priestore.

Následne sa pre každý bod z poľa *all_p* vypočíta euklidovská vzdialenosť od aktuálneho bodu:

$$d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

Takto vypočítaná vzdialenosť sa vloží spolu s farbou bodu do poľa, ktoré sa v nasledujúcom kroku zoradí podľa vzdialenosti od najmenšej po najväčšiu.

Následne sa prejde usporiadané pole a zoberie sa prvých k hodnôt a spočíta sa počet susedov každej farby a maximálna hodnota sa vráti ako výstup funkcie.

V prípade, že je viacero bodov rovnakej farby, funkcia vráti farbu najbližšieho bodu k aktuálnemu bodu.

Spúšťanie programu

Program sa spúšťa tak, že najskôr sa spustí program *main.py* pomocou interpretera **PyPy**. Tento program zapíše všetky body do 4 textových súborov (1 súbor pre každé k).

Následne je potrebné spustiť program *plot.py* pomocou klasického **Python** interpretera. Tento program prečíta body z textových súborov a pomocou knižnice *matplotlib*, tieto body zobrazí v grafe.

Vizualizácia

Na vizualizovanie výsledkov používam knižnicu *matplotlib*.

```
def plot_four_measurements(measurement1, measurement2, measurement3, measurement4):
    #vytvorenie 2x2 mriežky pre výpisy
    fig, axs = plt.subplots(2, 2, figsize=(10, 8))

    # Plot each measurement in its respective subplot
    plot_measurement(axs[0, 0], measurement1, 'k = 1')
    plot_measurement(axs[0, 1], measurement2, 'k = 3')
    plot_measurement(axs[1, 0], measurement3, 'k = 7')
    plot_measurement(axs[1, 1], measurement4, 'k = 15')

    #nastavenie vlastností pre jednotlivé časti mriežky
    for ax_row in axs:
        for ax in ax_row:
            ax.set_xlim(-5000, 5000)
            ax.set_ylim(-5000, 5000)
            ax.set_xticks(np.arange(-5000, 6000, 2000))
            ax.set_yticks(np.arange(-5000, 6000, 2000))
            ax.set_xlabel("X")
            ax.set_ylabel("Y")

    fig.tight_layout()

    plt.suptitle('KNN')
    plt.savefig("knn")      #uloženie ako obrázok

    #zobrazenie
    plt.show()

#funkcia, ktorá vypíše body jedného merania do grafu
def plot_measurement(ax, measurement, title):
    for point_array in measurement:
        if len(point_array) == 0:
            continue
        x, y, color = zip(*[(point.x, point.y, point.color) for point in point_array])
        ax.scatter(x, y, marker='o', color=color[0], s=50)
    ax.set_title(title)
```

Optimalizácie

1. Rozdelil som plochu na viaceré menšie štvorce, do ktorých umiestňujem body s príslušnými súradnicami, aby som nemusel vždy porovnávať všetky body, ale len body vo štvorci, kde sa nachádza aktuálny bod a susedných štvorcoch.
2. PyPy je implementácia programovacieho jazyka Python. PyPy často beží rýchlejšie ako štandardná implementácia CPython, pretože PyPy používa just-in-time kompilátor. Pypy nepodporuje niektoré grafické knižnice.
   - keďže PyPy nepodporuje niektoré grafické knižnice, teda ani matplotlib, rozdelil som moju implementáciu do dvoch súborov
     - v jednom súbore je implementácia algoritmu a všetkých tried, tento program spúšťam pomocou PyPy a výsledky ukladám do textových súborov
     - v druhom súbore sa nachádza kód pre vizualizáciu bodov pomocou knižnice matplotlib a tento program spúšťam klasicky cez Python, program načíta body zo súborov a vypíše ich

Testovanie

Na demonštráciu môjho klasifikátora postupne generujem nové body a klasifikujem ich volaním funkcie classify. Celkovo vygenerujem 40 000 nových bodov (teda 10 000 z každej triedy). Súradnice nových bodov generujem náhodne, pričom nový bod má zakaždým inú triedu:

- R body by mali byť generované s 99% pravdepodobnosťou s X < +500 a Y < +500
- G body by mali byť generované s 99% pravdepodobnosťou s X > -500 a Y < +500
- B body by mali byť generované s 99% pravdepodobnosťou s X < +500 a Y > -500
- P body by mali byť generované s 99% pravdepodobnosťou s X > -500 a Y > -500

Testovanie vykonávam 4-krát, pričom zakaždým klasifikátor použije iný parameter k (pre k = 1, 3, 7 a 15) a vygenerované body sú pre každý experiment rovnaké.

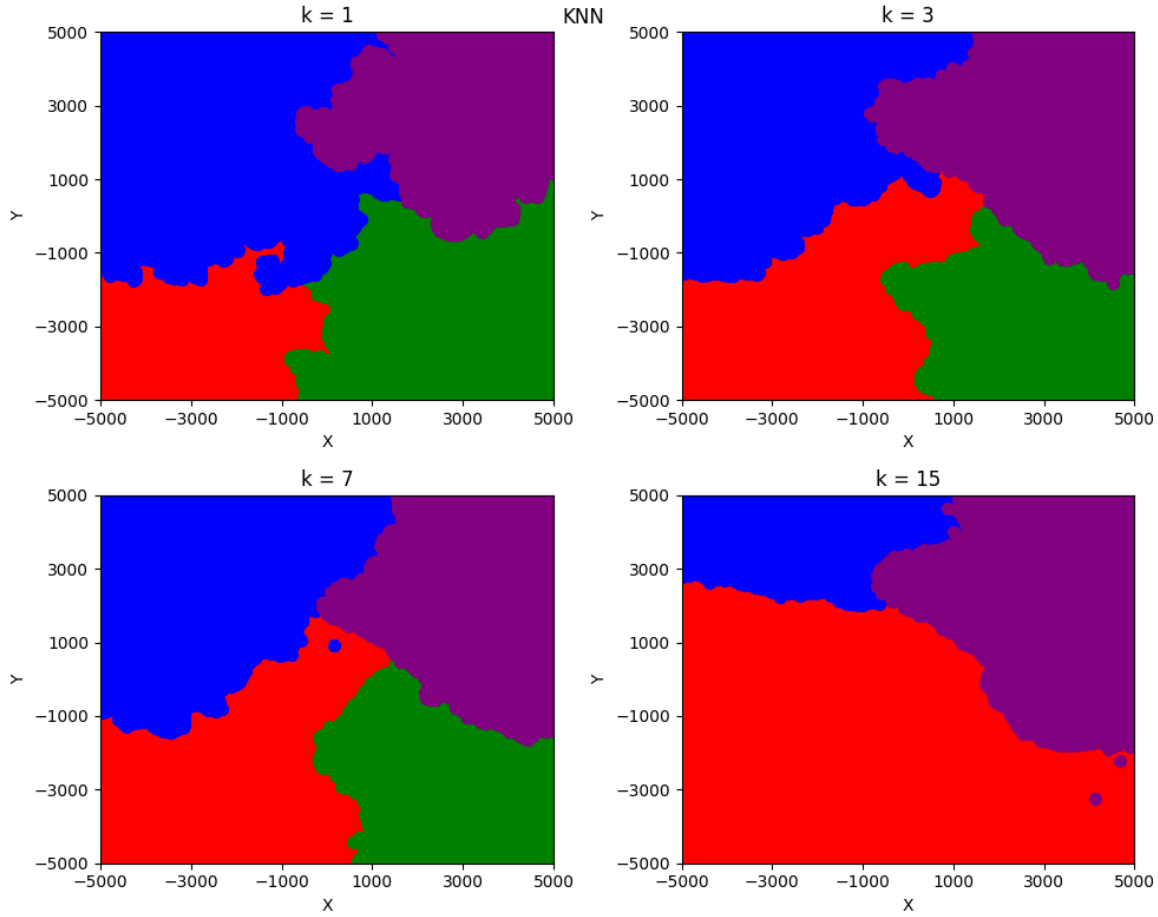
Návratovú hodnotu funkcie *classify* porovnávam s triedou vygenerovaného bodu. Na základe týchto porovnaní vyhodnocujem úspešnosť môjho klasifikátora pre daný experiment a to tak, že:

- počítam všetky body, ktoré boli preklasifikované na inú farbu
- následne vypíšem, koľko percent bodov každej farby sa nachádza v danom intervale

```
#vypísanie štatistiky
print("#########################################################")
print(f"k = {k}")
print("---------------------------------------------------------")
print("Red:", COUNT, ",", red_wrong)
print("Red-correct:", round((1-red_wrong/COUNT)*100, 2), "%")
print("---")
print("Green:", COUNT, ",", green_wrong)
print("Green-correct:", round((1-green_wrong/COUNT)*100, 2), "%")
print("---")
print("Blue:", COUNT, ",", blue_wrong)
print("Blue-correct:", round((1-blue_wrong/COUNT)*100, 2), "%")
print("---")
print("Purple:", COUNT, ",", purple_wrong)
print("Purple-correct:", round((1-purple_wrong/COUNT)*100, 2), "%")
print("#########################################################")
```

Experiment vykonám 5-krát s náhodnými bodmi.

Príklad 1



k	X < 500 a Y < 500 -> RED		X > -500 a Y < 500 -> GREEN		X < 500 a Y > -500 -> BLUE		X > -500 a Y > -500 -> PURPLE	
	nesprávne	úspešnosť	nesprávne	úspešnosť	nesprávne	úspešnosť	nesprávne	úspešnosť
1	4243	**57,57%**	1504	**84,96%**	485	**95,15%**	3271	**67,29%**
3	2143	**78,57%**	3650	**63,50%**	1686	**83,14%**	2466	**75,34%**
7	1829	**81,71%**	3560	**64,40%**	1536	**84,64%**	3013	**69,87%**
15	34	**99,66%**	10000	**0%**	5657	**43,43%**	2340	**76,60%**

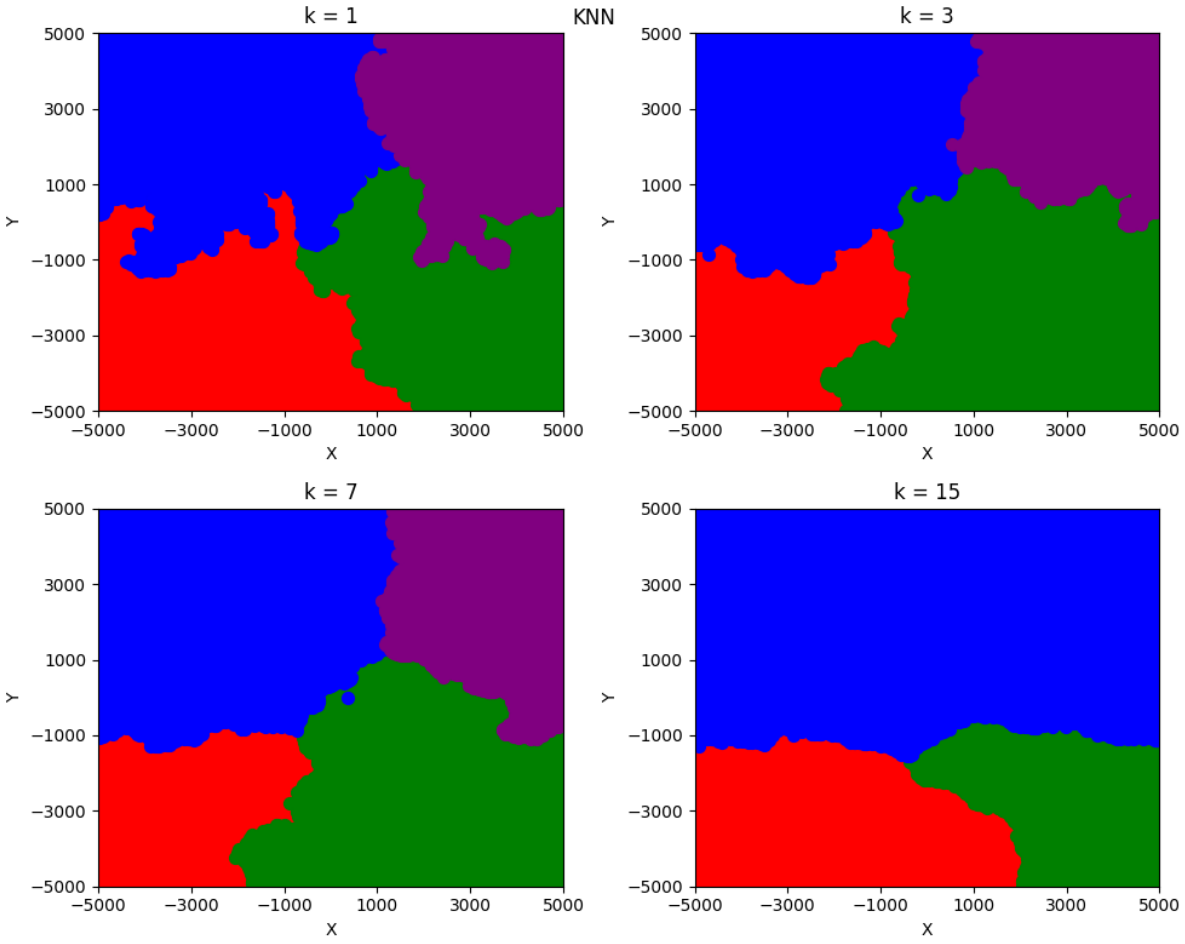
Čas vykonávania: 27,765s

Vyhodnotenie:

Z tabuľky vyššie môžeme vidieť, že z hľadiska úspešnosti dopadol najlepšie algoritmus pre k=1. Po sčítaní úspešností pre jednotlivé intervaly bolo toto číslo najväčšie a teda úspešnosť tohto k je najlepšia. Najhoršie dopadol algoritmus pre k=15.

V riešení však môžeme vidieť aj niekoľko outlierov, ktorí sú pravdepodobne spôsobení tým, že tieto body boli klasifikované medzi prvými, kedy ešte na ploche nebol dostatočný počet bodov.

Príklad 2



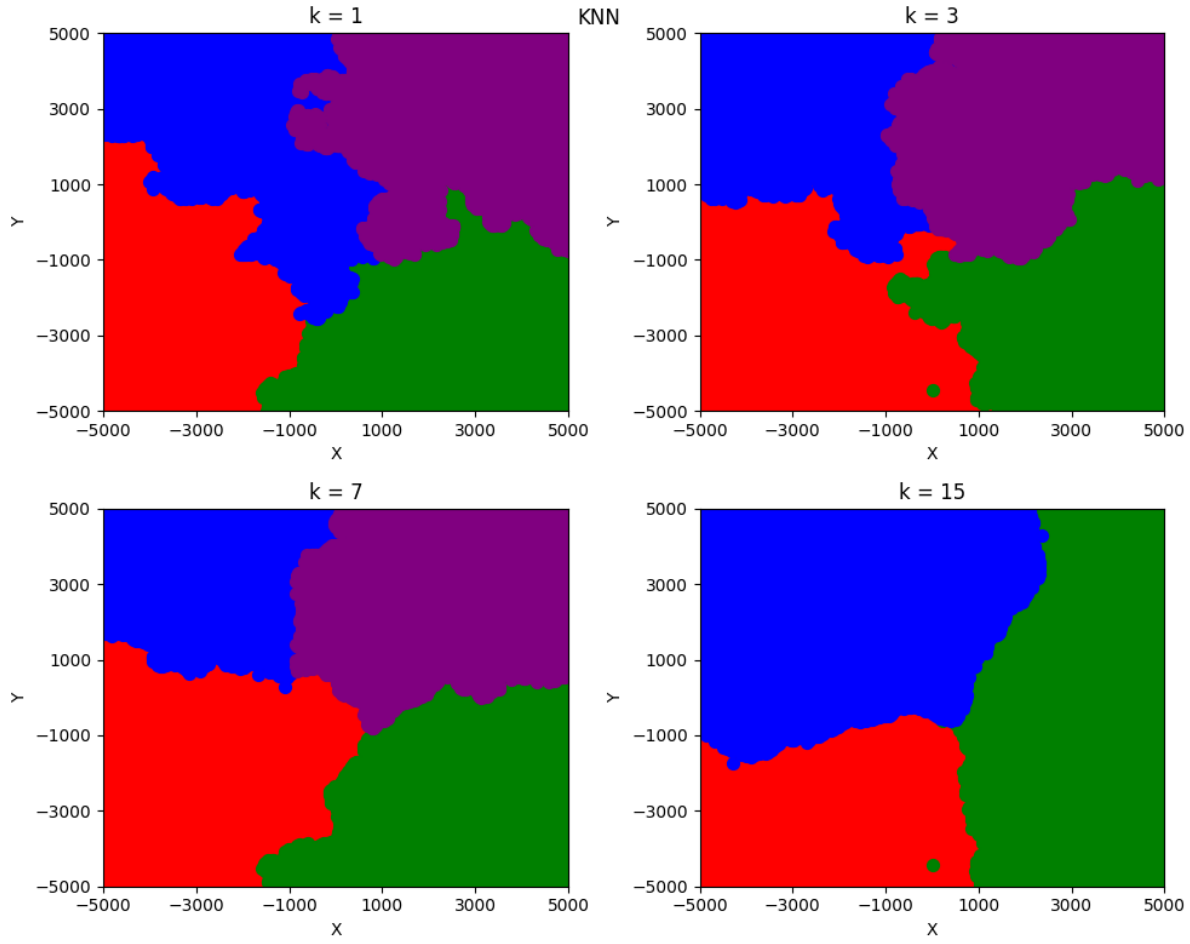
k	X < 500 a Y < 500 -> RED		X > -500 a Y < 500 -> GREEN		X < 500 a Y > -500 -> BLUE		X > -500 a Y > -500 -> PURPLE	
	nesprávne	úspešnosť	nesprávne	úspešnosť	nesprávne	úspešnosť	nesprávne	úspešnosť
1	1761	**82,39%**	2277	**77,23%**	1224	**87,76%**	4132	**58,86%**
3	4365	**56,35%**	256	**97,44%**	775	**92,25%**	4597	**54,03%**
7	4655	**53,45%**	871	**91,29%**	329	**96,71%**	4335	**56,65%**
15	3255	**67,45%**	4541	**54,59%**	41	**99,59%**	10000	**0,00%**

Čas vykonávania: 27,719s

Vyhodnotenie:

Z tabuľky vyššie môžeme vidieť, že z hľadiska úspešnosti dopadol najlepšie algoritmus pre k=1. Po sčítaní úspešností pre jednotlivé intervaly bolo toto číslo najväčšie a teda úspešnosť tohto k je najlepšia.

Príklad 3



k	X < 500 a Y < 500 -> RED		X > -500 a Y < 500 -> GREEN		X < 500 a Y > -500 -> BLUE		X > -500 a Y > -500 -> PURPLE	
	nesprávne	úspešnosť	nesprávne	úspešnosť	nesprávne	úspešnosť	nesprávne	úspešnosť
1	3010	**69,90%**	1991	**80,09%**	2699	**73,01%**	2082	**79,18%**
3	1232	**87,68%**	2791	**72,09%**	3052	**69,48%**	1788	**82,12%**
7	1197	**88,03%**	1799	**82,01%**	4079	**59,21%**	1562	**84,38%**
15	2562	**74,38%**	2446	**75,54%**	113	**98,87%**	10000	**0,00%**

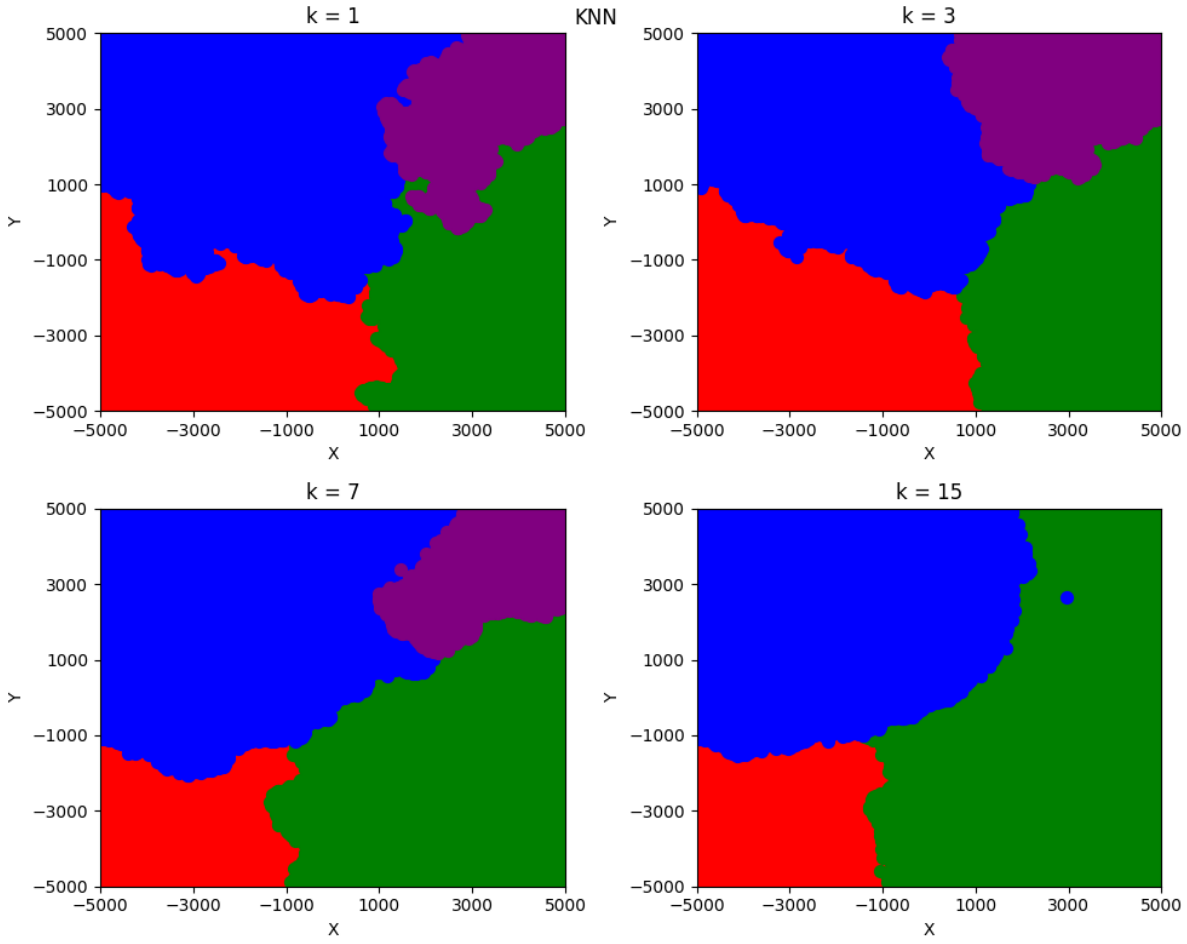
Čas vykonávania: 28,072s

Vyhodnotenie:

Z tabuľky vyššie môžeme vidieť, že z hľadiska úspešnosti dopadol najlepšie algoritmus pre k=7. Po sčítaní úspešností pre jednotlivé intervaly bolo toto číslo najväčšie a teda úspešnosť tohto k je najlepšia. Najhoršie dopadol algoritmus pre k=15.

V riešení však môžeme vidieť aj outlierov, ktorí sú pravdepodobne spôsobení tým, že tieto body boli klasifikované medzi prvými, kedy ešte na ploche nebol dostatočný počet bodov.

Príklad 4



k	X < 500 a Y < 500 -> RED		X > -500 a Y < 500 -> GREEN		X < 500 a Y > -500 -> BLUE		X > -500 a Y > -500 -> PURPLE	
	nesprávne	úspešnosť	nesprávne	úspešnosť	nesprávne	úspešnosť	nesprávne	úspešnosť
1	2741	**72,59%**	3112	**68,88%**	421	**95,79%**	6185	**38,15%**
3	2139	**78,61%**	2851	**71,49%**	708	**92,92%**	5645	**43,55%**
7	5370	**46,30%**	334	**96,66%**	142	**98,58%**	6799	**32,01%**
15	5083	**49,17%**	321	**96,79%**	139	**98,61%**	10000	**0,00%**

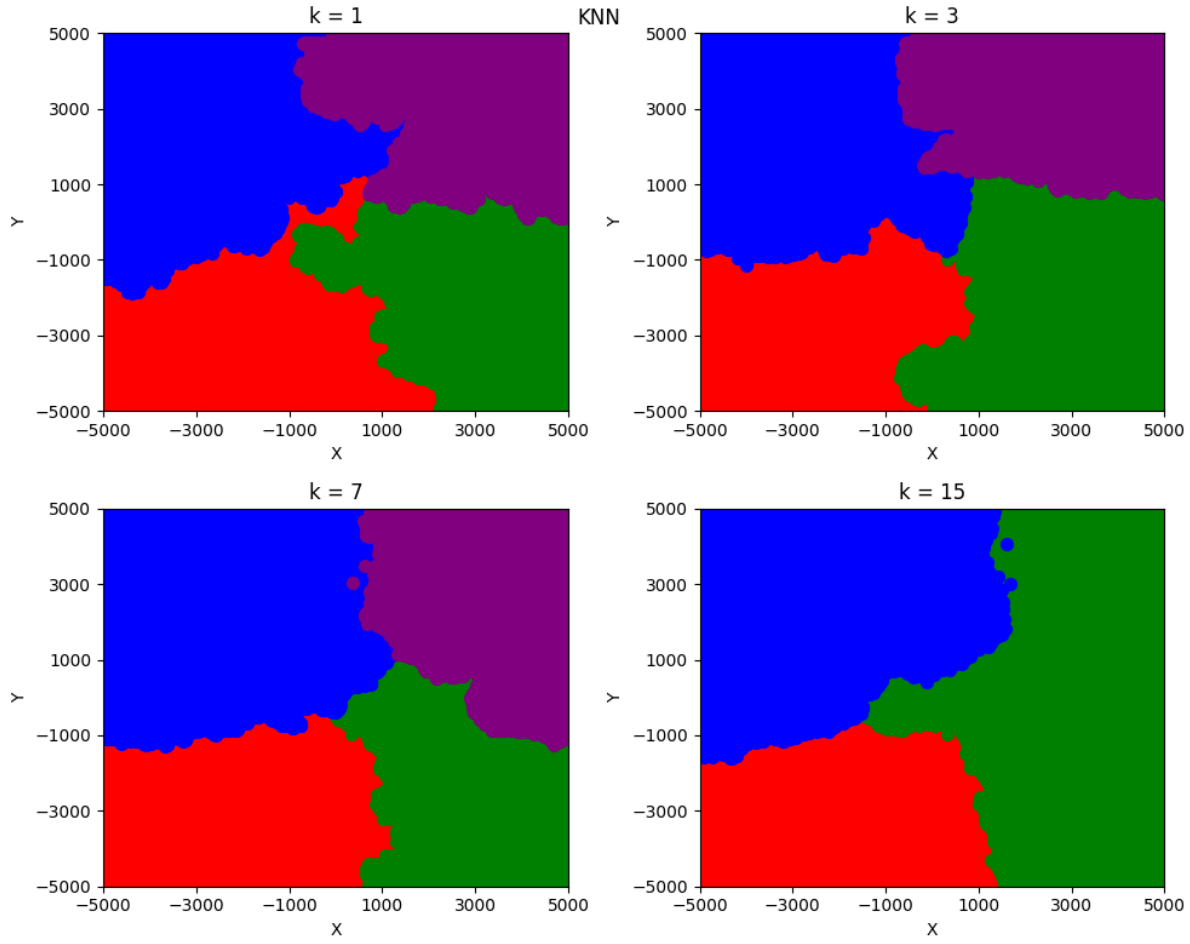
Čas vykonávania: 27,717s

Vyhodnotenie:

Z tabuľky vyššie môžeme vidieť, že z hľadiska úspešnosti dopadol najlepšie algoritmus pre k=3. Po sčítaní úspešností pre jednotlivé intervaly bolo toto číslo najväčšie a teda úspešnosť tohto k je najlepšia. Najhoršie dopadol algoritmus pre k=15.

V riešení však môžeme vidieť aj outlierov, ktorí sú pravdepodobne spôsobení tým, že tieto body boli klasifikované medzi prvými, kedy ešte na ploche nebol dostatočný počet bodov.

Príklad 5



k	X < 500 a Y < 500 -> RED		X > -500 a Y < 500 -> GREEN		X < 500 a Y > -500 -> BLUE		X > -500 a Y > -500 -> PURPLE	
	nesprávne	úspešnosť	nesprávne	úspešnosť	nesprávne	úspešnosť	nesprávne	úspešnosť
1	2390	**76,10%**	2714	**72,86%**	1555	**84,45%**	3177	**68,23%**
3	2534	**74,66%**	1693	**83,07%**	1249	**87,51%**	3329	**66,71%**
7	2490	**75,10%**	3373	**66,27%**	209	**97,91%**	3404	**65,96%**
15	2732	**72,68%**	2298	**77,02%**	791	**92,09%**	10000	**0,00%**

Čas vykonávania: 28,185s

Vyhodnotenie:

Z tabuľky vyššie môžeme vidieť, že z hľadiska úspešnosti dopadol najlepšie algoritmus pre k=3. Po sčítaní úspešností pre jednotlivé intervaly bolo toto číslo najväčšie a teda úspešnosť tohto k je najlepšia. Najhoršie dopadol algoritmus pre k=15.

V riešení však môžeme vidieť aj outlierov, ktorí sú pravdepodobne spôsobení tým, že tieto body boli klasifikované medzi prvými, kedy ešte na ploche nebol dostatočný počet bodov.

Záver

V rámci tohto projektu som vytvoril klasifikátor pre 2D priestor pomocou k-NN algoritmu na základe poskytnutého zadania. Cieľom v tejto úlohe bolo kategorizovať body do štyroch tried: red (R), green (G), blue (B) a purple (P) na základe ich súradníc. Experiment som opakoval štyrikrát s rôznymi hodnotami parametra k (1, 3, 7 a 15). A testovanie som vykonal 5-krát s náhodnými bodmi.

Výsledky experimentov:

Z experimentov môžeme vidieť, že:

- 2-krát bol algoritmus najúspešnejší pre k=1
- 2-krát bol algoritmus najúspešnejší pre k=3
- 1-krát bol algoritmus najúspešnejší pre k=7
- vo všetkých prípadoch bol najhorší pre k=15

Keď si však sčítame všetky chyby pre každé k, z každého príkladu:

k	Celkovo chýb	Celková úspešnosť
1	50974	**74,51%**
3	48949	**75,53%**
7	50886	**74,56%**
15	82353	**58,82%**

V tabuľke vyššie môžeme vidieť vyhodnotenie všetkých úspešností spolu. Z tejto tabuľky vyplýva, že najlepšie dopadol klasifikátor pre **k=3**. Najhoršie dopadol pre **k=15**.

Zistil som, že voľba parametra k ovplyvňuje výsledky klasifikácie. Nižšie hodnoty k môžu viesť k presnejším výsledkom v niektorých situáciách, ale zároveň môžu byť citlivejšie na odľahlé hodnoty.

V niektorých prípadoch sme mohli vidieť aj takzvaných „outlierov“, teda body, ktoré ležia mimo ostatných zo svojej triedy. Tieto odchýlky boli spôsobené pravdepodobne tým, že tieto body boli klasifikované ešte kým nebol klasifikovaný dostatočný počet bodov, a preto mohlo dôjsť ku chybnej klasifikácií bodu.

Vytvorený klasifikátor pomocou k-NN algoritmu je schopný úspešne kategorizovať body v 2D priestore do zadaných tried. Príklady ukázali, že správna voľba hodnôt parametrov môže viesť k optimálnym výsledkom.