

Laboratorio 5: K Nearest Neighbors

Alumno: Josue Samuel Philco Puma

15 de noviembre de 2024

1. Definición del K Nearest Neighbors

El **k-nearest neighbors** o también conocido como el algoritmo de los **k** vecinos más cercanos es un método que utiliza principalmente para tareas de clasificación y regresión basándose en la premisa de que los puntos de datos similares se encuentran cerca de otros en el espacio multidimensional. Es una herramienta poderosa en el ámbito del aprendizaje automático, ideal para situaciones donde la simplicidad y la interpretabilidad son cruciales, aunque su rendimiento puede verse afectado por la elección del valor **k** y la dimensionalidad del espacio de características.

2. Implementación

Vamos a realizar el algoritmo de **k Nearest Neighbor** de dos formas. La primera implementación será la **solución por fuerza bruta**, los datos son buscados uno por uno sin ninguna estructura, y la segunda implementación será usando un **KD-Tree** con su función de inserción, este es mucho más eficiente en las búsquedas.

- **Solución por fuerza bruta:** Para saber cuales son los vecinos más cercanos al punto será de acuerdo a la distancia y mediante un **k** que será la cantidad de vecinos que estamos buscando. Para esto usaremos la distancia euclidiana en datos de 3 dimensiones.

```
1 # Implementacion del K nearest neighbor por fuerza bruta
2
3 def euclidean_distance(point1, point2):
4     return ((point1[0] - point2[0])**2 + (point1[1] - point2[1])**2 + (
5         point1[2] - point2[2])**2)**0.5
6
7 def bruce_force_knn(points, query_point, k):
8     distances = []
9     for point in points:
10         distances.append((point, euclidean_distance(point, query_point)))
11     distances = sorted(distances, key=lambda x: x[1])
12     return distances[:k]
13
14 points = [
15     (2, 3, 1),
16     (5, 4, 7),
17     (9, 6, 3),
18     (4, 7, 8),
19     (8, 1, 5),
20     (7, 2, 6)
21 ]
22
23 query_point = (9, 2, 4)
24 k = 3
25
26 nearest_neighbors = bruce_force_knn(points, query_point, k)
27 print("k-Nearest Neighbors:")
28 for neighbor, distance in nearest_neighbors:
29     print(f"Point: {neighbor}, Distance: {distance:.2f}")
```

Esta implementación es considerada de **fuerza bruta** debido a que debemos recorrer todos los puntos que tengamos para calcular su **distancia euclidiana**. Al ejecutar el código nos mostrará cual son los vecinos más cercanos y a cuanta distancia están del punto **query_point**.

```
> python KNN_Nearest_Neighbor.py
k-Nearest Neighbors:
Point: (8, 1, 5), Distance: 1.73
Point: (7, 2, 6), Distance: 2.83
Point: (9, 6, 3), Distance: 4.12
```

Figura 1: Ejecución de Ejemplo

Para ver como es que se comporta con más datos, tenemos los archivos 1000.csv, 10000.csv y 20000.csv, estos archivos como indica el nombre contiene el número de datos. Entonces modificaremos el código para que empiece a leer los archivos, además generaremos los puntos de búsqueda de forma aleatoria para este caso, estos serán distintos en cada archivo. Además iremos viendo como es que va en tiempo de ejecución en encontrar todos los vecinos más cercanos al query_point.

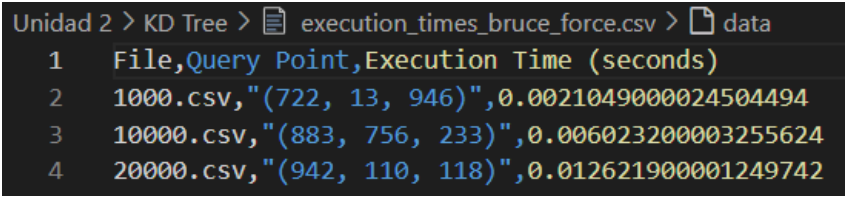
```
1 # Lectura de archivos con query_point aleatorio y tiempo de ejecucion
2
3 import time
4 import random
5 import csv
6
7 def euclidean_distance(point1, point2):
8     return ((point1[0] - point2[0])**2 + (point1[1] - point2[1])**2 + (
9         point1[2] - point2[2])**2)**0.5
10
11 def bruce_force_knn(points, query_point, k):
12     distances = []
13     for point in points:
14         distances.append((point, euclidean_distance(point, query_point)))
15     distances = sorted(distances, key=lambda x: x[1])
16     return distances[:k]
17
18 def load_points(filename):
19     points = []
20     with open(filename, "r") as file:
21         for line in file:
22             x, y, z = map(float, line.strip().split(','))
23             points.append((x, y, z))
24     return points
25
26 def generate_random_query():
27     return (random.randint(0, 999), random.randint(0, 999), random.randint(
28         0, 999))
29
30 file_names = ["1000.csv", "10000.csv", "20000.csv"]
31 k = 600
32
33 with open("execution_times_bruce_force.csv", mode="w", newline="") as
34     csv_file:
35         csv_writer = csv.writer(csv_file)
36         csv_writer.writerow(["File", "Query Point", "Execution Time (seconds)"
37             ])
38
39         for file_name in file_names:
40             print(f"Processing {file_name}...")
41             points = load_points(file_name)
42
43             start_time = time.perf_counter()
44             query_point = generate_random_query()
45             nearest_neighbors = bruce_force_knn(points, query_point, k)
46             end_time = time.perf_counter()
```

```

44     exec_time = end_time - start_time
45     csv_writer.writerow([file_name, query_point, exec_time])
46
47     print("Query Point:", query_point)
48     print("k-Nearest Neighbors:")
49     for neighbor, distance in nearest_neighbors:
50         print(f"Point: {neighbor}, Distance: {distance:.2f}")
51     print(f"Execution Time: {exec_time:.5f} seconds\n")
52
53 print("Execution times saved to execution_times_bruce_force.csv")

```

Al ejecutar el código leerá todos los datos de los 3 archivos correspondientes, para cada archivo generará un punto aleatorio para empezar la búsqueda, en este caso estamos usando un $k = 600$ que indica que buscará los 600 vecinos más cercanos al `query_point`. Los resultados generados están presentes en un archivo csv.



File	Query Point	Execution Time (seconds)
1000.csv	(722, 13, 946)	0.0021049000024504494
10000.csv	(883, 756, 233)	0.006023200003255624
20000.csv	(942, 110, 118)	0.012621900001249742

Figura 2: Resultados de la búsqueda

Podemos ver que el tiempo de ejecución logra estar variando demasiado, esto se debe a la generación del punto aleatorio y que recorre todos los datos comparando para encontrar cuáles son los puntos vecinos más cercanos al `query_point`.

- **Usando un KD-Tree:** Como vimos en el anterior punto, hacerlo por **fuerza bruta** puede a veces tener un tiempo de ejecución mayor en cuanto incrementamos el número de datos. Entonces para optimizar estos tiempos en la búsqueda de los vecinos más cercanos usaremos la estructura **KD-Tree**, que va a permitir realizar búsquedas más rápidas al dividir recursivamente el espacio en regiones más pequeñas. Para esto usaremos datos en 3 dimensiones.

```

1  # Implementacion del KD-Tree para K Nearest Neighbors
2
3  import math
4  import heapq
5
6  class KDTree:
7      class Node:
8          def __init__(self, point):
9              self.point = point
10             self.left = None
11             self.right = None
12
13         def __init__(self, k):
14             self.k = k
15             self.root = None
16
17         def insert_recursive(self, node, point, depth):
18             if node is None:
19                 return self.Node(point)
20
21             cd = depth % self.k
22
23             if point[cd] < node.point[cd]:
24                 node.left = self.insert_recursive(node.left, point, depth + 1)
25             else:
26                 node.right = self.insert_recursive(node.right, point, depth + 1)

```

```

27         return node
28
29     def insert(self, point):
30         self.root = self.insert_recursive(self.root, point, 0)
31
32     def search_recursive(self, node, point, depth):
33         if node is None:
34             return False
35
36         if node.point == point:
37             return True
38
39         cd = depth % self.k
40
41         if point[cd] < node.point[cd]:
42             return self.search_recursive(node.left, point, depth + 1)
43         else:
44             return self.search_recursive(node.right, point, depth + 1)
45
46     def search(self, point):
47         return self.search_recursive(self.root, point, 0)
48
49     def euclidean_distance(self, point1, point2):
50         return math.sqrt(sum((x - y) ** 2 for x, y in zip(point1, point2)))
51
52     def knn_recursive(self, node, query_point, k, depth, heap):
53         if node is None:
54             return
55
56         distance = self.euclidean_distance(query_point, node.point)
57
58         if len(heap) < k:
59             heapq.heappush(heap, (-distance, node.point))
60         else:
61             if -heap[0][0] > distance:
62                 heapq.heappushpop(heap, (-distance, node.point))
63
64         cd = depth % self.k
65         next_branch = node.left if query_point[cd] < node.point[cd] else
66             node.right
67         opposite_branch = node.right if next_branch == node.left else node.
68             left
69
70         self.knn_recursive(next_branch, query_point, k, depth + 1, heap)
71
72         if len(heap) < k or abs(query_point[cd] - node.point[cd]) < -heap
73             [0][0]:
74             self.knn_recursive(opposite_branch, query_point, k, depth + 1,
75                 heap)
76
77     def knn_nearest_neighbor(self, query_point, k):
78         heap = []
79         self.knn_recursive(self.root, query_point, k, 0, heap)
80         return [(-distance, point) for distance, point in sorted(heap,
81             reverse=True)]
82
83     def to_dot(self):
84         def node_to_dot(node):
85             if not node:
86                 return ""
87             node_str = f'"{node.point}" [label="{node.point}"]; \n'
88             if node.left:
89                 node_str += f'"{node.left.point}" -> "{node.point}" [label

```

```

85         ="L"];\n'
86         node_str += node_to_dot(node.left)
87     if node.right:
88         node_str += f'"{node.point}" -> "{node.right.point}" [label
89         ="R"];\n'
90         node_str += node_to_dot(node.right)
91     return node_str
92
93     return "digraph KDTree {\n" + node_to_dot(self.root) + "}\n"
94
95 kdtree = KDTree(3)
96 points = [
97     (2, 3, 1),
98     (5, 4, 7),
99     (9, 6, 3),
100    (4, 7, 8),
101    (8, 1, 5),
102    (7, 2, 6)
103 ]
104
105 for p in points:
106     kdtree.insert(p)
107
108 query_point = (9, 2, 4)
109 k = 3
110 nearest_neighbors = kdtree.knn_nearest_neighbor(query_point, k)
111
112 print("Query Point:", query_point)
113 print("k-Nearest Neighbors:")
114 for distance, neighbor in nearest_neighbors:
115     print(f"Point: {neighbor}, Distance: {distance:.2f}")
116
117 dot_representation = kdtree.to_dot()
118 print("\nGraphviz DOT representation:\n")
119 print(dot_representation)

```

Entonces, si ejecutamos este código lo que obtendremos es la misma búsqueda de los vecinos más cercanos pero usando un **KD-Tree**, además para comprobar si estamos insertando bien se hizo una figura para ver como se distribuyeron los puntos.

```

> python .\KNN_Nearest_Neighbor_KDTree.py
Query Point: (9, 2, 4)
k-Nearest Neighbors:
Point: (8, 1, 5), Distance: 1.73
Point: (7, 2, 6), Distance: 2.83
Point: (9, 6, 3), Distance: 4.12

Graphviz DOT representation:

digraph KDTree {
"(2, 3, 1)" [label="(2, 3, 1)"];
"(2, 3, 1)" -> "(5, 4, 7)" [label="R"];
"(5, 4, 7)" [label="(5, 4, 7)"];
"(5, 4, 7)" -> "(8, 1, 5)" [label="L"];
"(8, 1, 5)" [label="(8, 1, 5)"];
"(8, 1, 5)" -> "(7, 2, 6)" [label="R"];
"(7, 2, 6)" [label="(7, 2, 6)"];
"(5, 4, 7)" -> "(9, 6, 3)" [label="R"];
"(9, 6, 3)" [label="(9, 6, 3)"];
"(9, 6, 3)" -> "(4, 7, 8)" [label="R"];
"(4, 7, 8)" [label="(4, 7, 8)"];
}

```

Figura 3: Ejecución con el KD-Tree

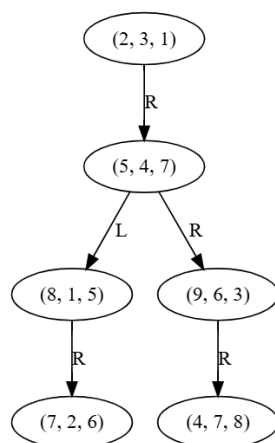


Figura 4: Inserción en tres dimensiones en el KD-Tree

Entonces vemos que la búsqueda de los vecinos más cercanos funciona correctamente. Ahora con esto debemos hacer lo mismo que el punto anterior, tenemos que leer los archivos 1000.csv, 10000.csv y 20000.csv (La visualización no la pondremos debido a la cantidad de datos), también haremos lo mismo con el query_point para que sea de forma aleatoria para los diferentes archivos, también el parámetro k será el mismo que el anterior para sacar las comparaciones.

```

1  # Lectura de archivos con query_point aleatorio y tiempo de ejecucion
2
3  import math
4  import heapq
5  import time
6  import random
7  import csv
8
9  class KDTree:
10     class Node:
11         def __init__(self, point):
12             self.point = point
13             self.left = None
14             self.right = None
15
16     def __init__(self, k):
17         self.k = k
18         self.root = None
19
20     def insert_recursive(self, node, point, depth):
21         if node is None:
22             return self.Node(point)
23
24         cd = depth % self.k
25
26         if point[cd] < node.point[cd]:
27             node.left = self.insert_recursive(node.left, point, depth + 1)
28         else:
29             node.right = self.insert_recursive(node.right, point, depth +
30             1)
31         return node
32
33     def insert(self, point):
34         self.root = self.insert_recursive(self.root, point, 0)
35
36     def euclidean_distance(self, point1, point2):
37         return math.sqrt(sum((x - y) ** 2 for x, y in zip(point1, point2)))
38
39     def knn_recursive(self, node, query_point, k, depth, heap):

```

```

39         if node is None:
40             return
41
42         distance = self.euclidean_distance(query_point, node.point)
43
44         if len(heap) < k:
45             heapq.heappush(heap, (-distance, node.point))
46         else:
47             if -heap[0][0] > distance:
48                 heapq.heappushpop(heap, (-distance, node.point))
49
50         cd = depth % self.k
51         next_branch = node.left if query_point[cd] < node.point[cd] else
52             node.right
53         opposite_branch = node.right if next_branch == node.left else node.
54             left
55
56         self.knn_recursive(next_branch, query_point, k, depth + 1, heap)
57
58         if len(heap) < k or abs(query_point[cd] - node.point[cd]) < -heap
59             [0][0]:
60             self.knn_recursive(opposite_branch, query_point, k, depth + 1,
61                 heap)
62
63     def knn_nearest_neighbor(self, query_point, k):
64         heap = []
65         self.knn_recursive(self.root, query_point, k, 0, heap)
66         return [(-distance, point) for distance, point in sorted(heap,
67             reverse=True)]
68
69     def read_points(file_name):
70         points = []
71         with open(file_name, "r") as file:
72             for line in file:
73                 points.append(tuple(map(int, line.strip().split(','))))
74         return points
75
76     def find_coordinate_ranges(points):
77         min_values = [float('inf')] * len(points[0])
78         max_values = [-float('inf')] * len(points[0])
79
80         for point in points:
81             for i in range(len(point)):
82                 min_values[i] = min(min_values[i], point[i])
83                 max_values[i] = max(max_values[i], point[i])
84
85         return min_values, max_values
86
87     def generate_random_query(min_values, max_values):
88         return tuple(random.randint(min_value, max_value) for min_value,
89             max_value in zip(min_values, max_values))
90
91     file_name = ['1000.csv', '10000.csv', '20000.csv']
92     results = []
93
94     for file in file_name:
95         print(f"\nProcessing file: {file}")
96         kdtree = KDTree(3)
97         points = read_points(file)
98         for p in points:
99             kdtree.insert(p)
100
101         min_values, max_values = find_coordinate_ranges(points)

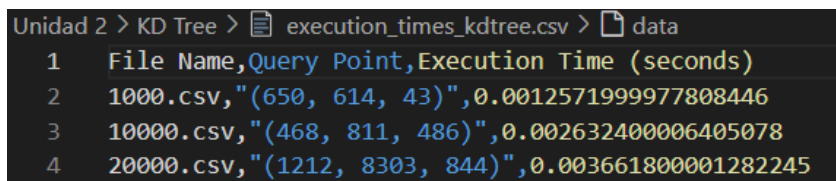
```

```

96
97     query_point = generate_random_query(min_values, max_values)
98     k = 600
99
100    start_time = time.perf_counter()
101    nearest_neighbors = kdtree.knn_nearest_neighbor(query_point, k)
102    end_time = time.perf_counter()
103
104    execution_time = end_time - start_time
105    print(f"Execution time: {execution_time:.10f} seconds")
106    results.append([file, query_point, execution_time])
107
108    print("Query Point:", query_point)
109    print("k-Nearest Neighbors:")
110    for distance, neighbor in nearest_neighbors:
111        print(f"Point: {neighbor}, Distance: {distance:.2f}")
112
113    with open('execution_times_kdtree.csv', 'w', newline='') as csvfile:
114        writer = csv.writer(csvfile)
115        writer.writerow(["File Name", "Query Point", "Execution Time (seconds)"
116                        ])
117        writer.writerows(results)
118    print("\nExecution times have been saved to 'execution_times_kdtree.csv'.")

```

Esto al ejecutarlo, nos dará el resultado del tiempo de ejecución en un archivo csv para mostrar cuanto se demora en encontrar los vecinos más cercanos de los 3 archivos.



```

Unidad 2 > KD Tree > execution_times_kdtree.csv > data
1  File Name,Query Point,Execution Time (seconds)
2  1000.csv,"(650, 614, 43)",0.0012571999977808446
3  10000.csv,"(468, 811, 486)",0.002632400006405078
4  20000.csv,"(1212, 8303, 844)",0.003661800001282245

```

Figura 5: Resultados con el KD-Tree

Entonces, si comparamos con los resultados de la solución por **fuerza bruta** son muchos más óptimos para la búsqueda de los vecinos más cercanos al `query_point`.

3. Costo computacional

Ahora haremos el análisis computacional de las funciones implementadas, en el caso del **KD-Tree** haremos un análisis más profundo que viene a ser las funciones de **inserción** y **búsqueda**. Para ambos se hará el análisis de la función de **K Nearest Neighbor**.

3.1. Análisis en el KD-Tree

Como se menciono, haremos un análisis de costo computacional para las funciones de inserción y de búsqueda. Así que vamos a analizar ambas funciones:

- **Función de inserción:** Esta función se encarga de ingresar los puntos en el árbol y en plano. En este caso estamos hablando de una inserción en 3 puntos, por lo que al insertar en cada dimensión se compara el nuevo punto con el punto actual en la dimensión actual. Primero en *x*, luego en *y* y por último en *z*, luego se repite en *x* y así sucesivamente. Para esta función tendremos 3 casos para el costo computacional:
 - **Mejor Caso:** Para este caso debemos contar con la mejor suerte y este vendría a ser insertar un solo punto en el árbol, esto es muy conveniente ya que el árbol está vacío e insertar el punto no tendríamos que tener que hacer comparaciones. Entonces el costo computacional de esta función vendría a ser tiempo constante, es decir, es $O(1)$.

- **Peor Caso:** Considerando el peor de los casos, debemos tener en cuenta que esta estructura puede ser óptima pero hay veces donde las inserciones de los puntos sean como una lista enlazada haciendo que al insertar un nodo nuevo vayamos recorriendo todo el árbol como lista enlazada. Entonces el costo computacional de esta función vendría a ser de tiempo lineal, es decir, es $O(n)$.
- **Caso Promedio:** En este caso es que vayamos insertando los puntos haciendo que el árbol esté balanceado, el **KD-Tree** en si no siempre garantiza un árbol balanceado. Pero en el caso en el árbol este de esa manera la inserción sería en tiempo logarítmico. Entonces el costo computacional vendría a ser $O(\log n)$.
- **Función de búsqueda:** Esta función simplemente buscará un punto en el árbol. Este seguirá la misma lógica que la inserción al ir comparando en cada dimensión para encontrar el punto. Si el punto es encontrado, se obtiene y se dice que el punto si existe en el árbol; de caso contrario, se dice que el punto no existe en el árbol. Al igual que la inserción seguiremos 3 casos:
 - **Mejor Caso:** En el mejor de los casos es que el punto que estamos buscando se encuentre en la raíz del árbol, lo que va a evitar hacer las comparaciones en las dimensiones. Entonces el costo computacional será de tiempo constante, es decir, es $O(1)$.
 - **Peor Caso:** En el peor caso de los casos es que el punto que buscamos se encuentre abajo al final y que la estructura del árbol se parezca a una lista enlazada, lo que ocasiona que estemos buscando por todos los puntos en el árbol. Entonces, el costo computacional será de tiempo lineal, es decir, es $O(n)$.
 - **Caso Promedio:** Para el caso promedio consideramos que el árbol se encuentre balanceado, lo que simplifica la búsqueda del punto en el árbol y no hacer muchas comparaciones. Entonces el costo computacional sería de tiempo logarítmico, es decir, es $O(\log n)$.

Resumiendo los costos en las funciones de inserción y búsqueda en el **KD-Tree**, tendremos:

Función	Mejor Caso	Peor Caso	Caso Promedio
Inserción	$O(1)$	$O(n)$	$O(\log n)$
Búsqueda	$O(1)$	$O(n)$	$O(\log n)$

Cuadro 1: Análisis de costo computacional para inserción y búsqueda en un KD-Tree

3.2. Análisis para K Nearest Neighbor

Esta función la vamos a analizar tanto en un **KD-Tree** y en su forma por **fuerza bruta**:

- **Por Fuerza Bruta:** En el método de fuerza bruta, se calculan las distancias de todos los puntos en el conjunto de datos al punto de consulta y luego se seleccionan los k más cercanos. Este enfoque es simple pero implica un costo computacional alto, ya que no se utilizan estructuras de datos que optimicen la búsqueda. Para sacar el costo computacional pues en todos los casos viene a ser el mismo porque debemos buscar los vecinos más cercanos en la mayoría de datos, por lo que el costo sería $O(n)$.
- **En un KD-Tree:** Para buscar los k vecinos más cercanos, el algoritmo recorre el árbol desde la raíz hacia las hojas, eligiendo el subárbol que contiene el punto objetivo y almacenando los candidatos más cercanos en una estructura de datos auxiliar. Luego, se retrocede para verificar otras ramas que podrían contener puntos más cercanos. En este lo haremos en los 3 casos:
 - **Mejor Caso:** En el mejor de los casos, esto solo ocurrirá si es que el árbol se encuentra balanceado y que nuestra búsqueda sea eficiente. Entonces el costo sería de $O(\log n + k)$ donde $\log n$ es el recorrido en profundidad en el árbol y k la recopilación de los vecinos más cercanos.
 - **Peor Caso:** En este caso solo ocurre si es que el árbol se encuentra desbalanceado como si fuera una lista enlazada o si la dimensionalidad es muy alta. Entonces el costo puede llegar a ser $O(n)$.
 - **Caso Promedio:** Para esto vamos a asumir que el árbol se encuentre balanceado y que las consultas son distribuidas de forma uniforme. Entonces el costo también es de $O(\log n + k)$.

Resumiendo, los costos para el **K Nearest Neighbor** tenemos lo siguiente:

Método	Mejor Caso	Peor Caso	Caso Promedio
KD-Tree (k-NN)	$O(\log n + k)$	$O(n)$	$O(\log n + k)$
Fuerza Bruta (k-NN)	$O(n)$	$O(n)$	$O(n)$

Cuadro 2: Análisis de costo computacional para la búsqueda de k vecinos más cercanos (k-NN)

4. Tiempo de ejecución y relación de incremento de k

Ahora que vimos como es que trabaja el algoritmo, tenemos que ver que pasa con el tiempo de ejecución si es que vamos incrementando el valor de k, para esto iremos contando desde un k igual a 100 y aumentándolo de 100 en 100 hasta llegar a tener un k igual a 900, lo haremos tanto para **fuerza bruta** y para el **KD-Tree**:

■ Fuerza Bruta

```

1  import time
2  import random
3  import csv
4
5  def euclidean_distance(point1, point2):
6      return ((point1[0] - point2[0])**2 + (point1[1] - point2[1])**2 + (
7          point1[2] - point2[2])**2)**0.5
8
9  def bruce_force_knn(points, query_point, k):
10     distances = []
11     for point in points:
12         distances.append((point, euclidean_distance(point, query_point)))
13     distances = sorted(distances, key=lambda x: x[1])
14     return distances[:k]
15
16 def load_points(filename):
17     points = []
18     with open(filename, "r") as file:
19         for line in file:
20             x, y, z = map(float, line.strip().split(','))
21             points.append((x, y, z))
22     return points
23
24 def generate_random_query():
25     return (random.randint(0, 999), random.randint(0, 999), random.randint(
26         0, 999))
27
28 file_names = ["1000.csv", "10000.csv", "20000.csv"]
29 k_values = [100, 200, 300, 400, 500, 600, 700, 800, 900]
30
31 with open("execution_times_bruce_force_k_diferent.csv", mode="w", newline="
32 ") as csv_file:
33     csv_writer = csv.writer(csv_file)
34     csv_writer.writerow(["File Name", "K", "Execution Time (seconds)"])
35
36     for file_name in file_names:
37         points = load_points(file_name)
38
39         for k in k_values:
40             print(f"Processing {file_name} with k = {k}...")
41             start_time = time.perf_counter()
42             query_point = generate_random_query()
43             nearest_neighbors = bruce_force_knn(points, query_point, k)
44             end_time = time.perf_counter()

```

```

43         exec_time = end_time - start_time
44         csv_writer.writerow([file_name, k, exec_time])
45
46         print("Query Point:", query_point)
47         print(f"Execution Time for k={k}: {exec_time:.5f} seconds\n")
48
49     print("Execution times for different k values saved to
        execution_times_bruce_force_k_diferent.csv")

```

■ KD-Tree

```

1  import math
2  import heapq
3  import time
4  import random
5  import csv
6
7  class KDTree:
8      class Node:
9          def __init__(self, point):
10             self.point = point
11             self.left = None
12             self.right = None
13
14      def __init__(self, k):
15          self.k = k
16          self.root = None
17
18      def insert_recursive(self, node, point, depth):
19          if node is None:
20              return self.Node(point)
21
22          cd = depth % self.k
23
24          if point[cd] < node.point[cd]:
25              node.left = self.insert_recursive(node.left, point, depth + 1)
26          else:
27              node.right = self.insert_recursive(node.right, point, depth + 1)
28          return node
29
30      def insert(self, point):
31          self.root = self.insert_recursive(self.root, point, 0)
32
33      def search_recursive(self, node, point, depth):
34          if node is None:
35              return False
36
37          if node.point == point:
38              return True
39
40          cd = depth % self.k
41
42          if point[cd] < node.point[cd]:
43              return self.search_recursive(node.left, point, depth + 1)
44          else:
45              return self.search_recursive(node.right, point, depth + 1)
46
47      def search(self, point):
48          return self.search_recursive(self.root, point, 0)
49
50      def euclidean_distance(self, point1, point2):
51          return math.sqrt(sum((x - y) ** 2 for x, y in zip(point1, point2)))

```

```

52
53     def knn_recursive(self, node, query_point, k, depth, heap):
54         if node is None:
55             return
56
57         distance = self.euclidean_distance(query_point, node.point)
58
59         if len(heap) < k:
60             heapq.heappush(heap, (-distance, node.point))
61         else:
62             if -heap[0][0] > distance:
63                 heapq.heappushpop(heap, (-distance, node.point))
64
65         cd = depth % self.k
66         next_branch = node.left if query_point[cd] < node.point[cd] else
            node.right
67         opposite_branch = node.right if next_branch == node.left else node.
            left
68
69         self.knn_recursive(next_branch, query_point, k, depth + 1, heap)
70
71         if len(heap) < k or abs(query_point[cd] - node.point[cd]) < -heap
            [0][0]:
72             self.knn_recursive(opposite_branch, query_point, k, depth + 1,
                heap)
73
74     def knn_nearest_neighbor(self, query_point, k):
75         heap = []
76         self.knn_recursive(self.root, query_point, k, 0, heap)
77         return [(-distance, point) for distance, point in sorted(heap,
            reverse=True)]
78
79     def read_points(file_name):
80         points = []
81         with open(file_name, "r") as file:
82             for line in file:
83                 points.append(tuple(map(int, line.strip().split(','))))
84         return points
85
86     def find_coordinate_ranges(points):
87         min_values = [float('inf')] * len(points[0])
88         max_values = [-float('inf')] * len(points[0])
89
90         for point in points:
91             for i in range(len(point)):
92                 min_values[i] = min(min_values[i], point[i])
93                 max_values[i] = max(max_values[i], point[i])
94
95         return min_values, max_values
96
97     def generate_random_query(min_values, max_values):
98         return tuple(random.randint(min_value, max_value) for min_value,
            max_value in zip(min_values, max_values))
99
100     file_name = ['1000.csv', '10000.csv', '20000.csv']
101     k_values = [100, 200, 300, 400, 500, 600, 700, 800, 900]
102     results = []
103
104     for file in file_name:
105         print(f"\nProcessing file: {file}")
106         kdtree = KDTree(3)
107         points = read_points(file)
108         for p in points:

```

```

109         kdtree.insert(p)
110
111     min_values, max_values = find_coordinate_ranges(points)
112
113     for k in k_values:
114         query_point = generate_random_query(min_values, max_values)
115
116         start_time = time.perf_counter()
117         nearest_neighbors = kdtree.knn_nearest_neighbor(query_point, k)
118         end_time = time.perf_counter()
119
120         execution_time = end_time - start_time
121         results.append([file, k, execution_time])
122
123         print(f"File: {file}, k: {k}, Execution time: {execution_time:.10f}
124               seconds")
125
126     with open('execution_times_vs_k.csv', 'w', newline='') as csvfile:
127         writer = csv.writer(csvfile)
128         writer.writerow(["File Name", "k", "Execution Time (seconds)"])
129         writer.writerows(results)
130
131     print("\nExecution times for varying k have been saved to '
132           execution_times_vs_k.csv'.")

```

```

1 import pandas as pd
2 import matplotlib.pyplot as plt
3
4 brute_force_data = pd.read_csv("execution_times_brute_force_k_diferent.csv")
5 kd_tree_data = pd.read_csv("execution_times_vs_k.csv")
6
7 datasets = brute_force_data['File Name'].unique()
8
9 for dataset in datasets:
10     bf_subset = brute_force_data[brute_force_data['File Name'] == dataset]
11     kd_subset = kd_tree_data[kd_tree_data['File Name'] == dataset]
12
13     # Plot
14     plt.figure(figsize=(10, 6))
15     plt.plot(bf_subset['K'], bf_subset['Execution Time (seconds)'], label="
16             Brute Force", marker='o')
17     plt.plot(kd_subset['k'], kd_subset['Execution Time (seconds)'], label="KD-
18             Tree", marker='s')
19     plt.title(f"Execution Time vs K for Dataset: {dataset}")
20     plt.xlabel("K")
21     plt.ylabel("Execution Time (seconds)")
22     plt.legend()
23     plt.grid(True)
24     plt.show()

```

Esto nos generará los gráficos de como va el tiempo:

■ 1000.csv



Figura 6: Para 1000 datos

■ 10000.csv

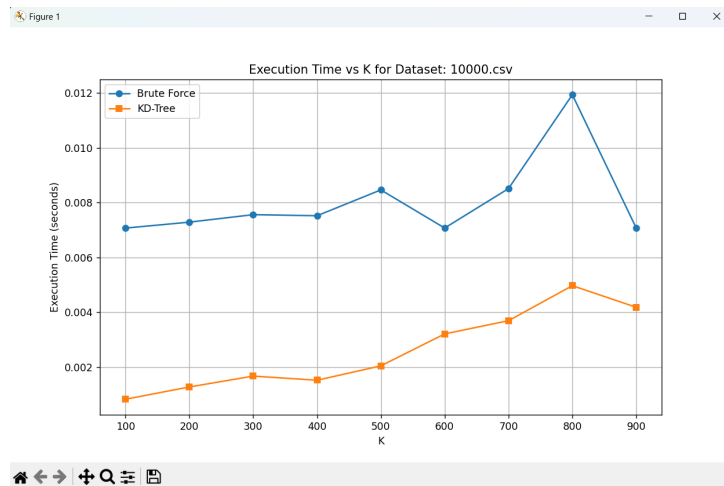


Figura 7: Para 10000 datos

■ 20000.csv

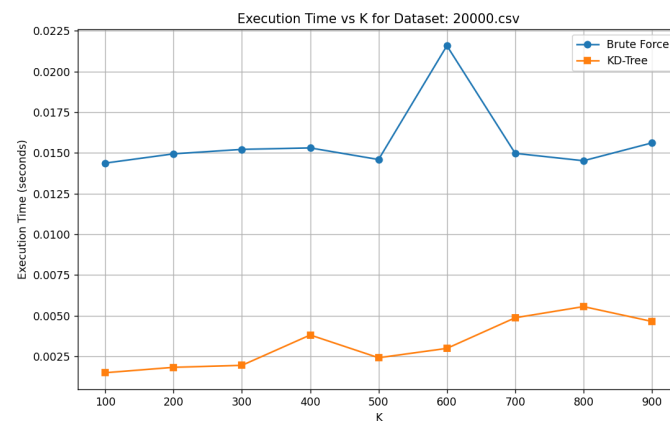


Figura 8: Para 20000 datos

Entonces viendo los gráficos generados podemos decir lo siguiente:

- **1000 puntos:** El tiempo de ejecución para ambos algoritmos se mantiene bajo, aunque el Brute Force muestra una mayor variabilidad en tiempos al aumentar k .
- **10000 puntos:** Aquí se observa un incremento en los tiempos de Brute Force, mientras que KD-Tree mantiene tiempos bajos y más consistentes.
- **20000 puntos:** El algoritmo Brute Force experimenta una mayor carga de tiempo, especialmente al incrementar k , mientras que KD-Tree sigue siendo más eficiente y muestra una curva más estable.

Esta comparación muestra cómo el KD-Tree gestiona mejor los tiempos de ejecución a medida que el tamaño del conjunto de datos y el número de vecinos aumentan, destacándose como una opción más escalable que el Brute Force para conjuntos de datos grandes.