Contents

Foreword

I have provided a detailed explanation of the code in this report and presented the output results. Simultaneously incorporating personal perspectives and discoveries on different algorithms and output results. The order of report writing follows the steps in the requirement document, step by step. I would greatly appreciate it if you could take the time to read this report.

Step 1: Select Pivots

Firstly, I need to read ten-dimensional data from the file, separated by spaces. I use a list to store the data.

```
def load_data(self, file_path):
    # save the 10d data to list, split using 'space'
    data = []
    with open(file_path, 'r') as file:
        for line in file:
            data.append(list(map(float, line.strip().split())))
    return np.array(data)
```

Then, create a function **'euclidean_distance'** to calculate the Euclidean distance.

```
def euclidean_distance(self, a, b):
    return np.sqrt(np.sum((a - b) ** 2))
```

Then it's time to calculate the pivot. First, use **data [0] as the seed** and select the object farthest from the seed as the first pivot. This can be solved through a loop.

```
# seed as data[0]
seed = self.data[0]
max_dist = -1
first_pivot = 0
for i in range(n):
    dist = self.euclidean_distance(seed, self.data[i])
    if dist > max_dist:
        max_dist = dist
        first_pivot = i
self.pivots.append(first_pivot)
```

The selection of subsequent pivot can still be solved using a loop. But the boundary for determining a pivot is the sum of the distances between the object and all previous pivot points.

```
sum_dist = sum(self.euclidean_distance(self.data[i], self.data[p]) for p in self.pivots)
if sum_dist > max_sum_dist:
    max_sum_dist = sum_dist
    next_pivot = i
```

After selecting the pivot, calculate the **distance** from each object to each pivot for future use.

```
# filling distances list
for i in range(n):
    for j, pivot in enumerate(self.pivots):
        self.distances[i][j] = self.euclidean_distance(self.data[i], self.data[pivot])
```

Finally, save all data in a picklel file. The reasons can be seen in the Appendix.

```
def save_pivot_info(self):
    with open("../data/pivots_info.pkl", "wb") as file:
        pickle.dump({
            "pivots": self.pivots,
            "distances": self.distances,
            "data": self.data
        }, file)
    print("Pivot information saved to 'pivots_info.pkl'")
```

Capture user input in the main function to customize the number of pivot points(numpivots).

When entering 5, the running result is the same as the example.

```
5
Pivots: [1109, 6878, 5514, 5026, 5338]
Distances array:
 [[2.14741915 0.5215314  1.63665604 1.85131602 1.55808568]
 [1.00349938 2.05578525 1.73131078 1.17123738 1.51913693]
 [1.47941103 1.65377417 1.8444899  1.25890786 1.73481959]
 ...
 [1.40572864 1.7997097  1.84825079 1.63050391 1.4779753 ]
 [1.23651526 1.45405846 1.62076032 1.57421091 1.32089137]
 [1.25951141 1.71555793 1.96059047 1.21385213 1.41869059]]
```

Step 2: iDistance

First, read two files and retrieve the pivot and distance information stored in the previous step. Read the location information of the query point.

```python
def load_pivot_info(file_path):

    with open(file_path, "rb") as file:
        details = pickle.load(file)

    pivots = details["pivots"]
    distances = details["distances"]
    data = details["data"]
    return pivots, distances, data
```

```python
def load_queries(file_path):

    queries = []
    with open(file_path, 'r') as file:
        for line in file:
            queries.append(list(map(float, line.strip().split())))
    return np.array(queries)
```

Then, following the steps of the iDistance method, we need to calculate the specific value of iDistance based on the **global maximum** distance from the point to the pivot that contains it. This requires determining which pivot the object is included in based on the distances between each object and each pivot calculated in the previous step, in order to calculate the **maximum range of each pivot** and save it in an array. Finally, the **maximum value of the array** is calculated as the global maximum value.

```python
# assign objects, calculate distances, and find global_maxd
for oid in range(n):
    # nearest pivot
    nearest_pivot_idx = np.argmin(distances[oid])
    nearest_pivot_dist = distances[oid][nearest_pivot_idx]

    # refresh maxd_per_pivot
    if nearest_pivot_dist > maxd[nearest_pivot_idx]:
        maxd[nearest_pivot_idx] = nearest_pivot_dist

# max(maxd) -> global_maxd
return max(maxd), maxd
```

After obtaining the global maximum value, the iDistance of each object can be calculated. That is: $i * global\_maxd + dist(o_i, p_{nearest})$

```python
# nearest pivot
nearest_pivot_idx = np.argmin(distances[oid])
nearest_pivot_dist = distances[oid][nearest_pivot_idx]

iDist = nearest_pivot_idx * global_maxd + nearest_pivot_dist
```

After calculating iDistance, I added the index of the nearest pivot to the object in the pivot array (i.e., the i^{th} pivot) and the index of the pivot in the original data to the array. For the convenience of subsequent calculations and determining conditions. Avoid double counting.

```
iDist_array.append((iDist, oid, nearest_pivot_idx, pivot_id))
```

Finally, sort and save it in the picklel file as well.

Step 3: Range similarity queries

After reading the picklel file that stored iDistance_array in the previous step from the file, there are three methods to implement range query.

Naïve approach

The naive method is equivalent to linear scanning, which only requires traversing all objects, calculating their distance to the current query point, and determining whether they meet the query range.

And because it is a traversal, the average distance calculation times are equal to the length of the dataset.

```
for query in queries:
    result = []
    for oid, obj in enumerate(data):
        dist = euclidean_distance(query, obj)
        if dist <= epsilon:
            result.append(oid)
    result.sort()
    results.append(result)
    totalcount += len(data)
```

Pivot-based search approach

For pivot-based methods, it is necessary to pre calculate the distance between the query point and each pivot. So we need to add the distance calculation times here to the total calculation times.

```
# calculate the distance between pivots and query
query_to_pivot_distances = [euclidean_distance(query, data[p]) for p in pivots]
totalcount += len(pivots)
```

In this way, for objects that meet the pruning conditions, where the absolute value of the difference between the distance from the object to the pivot and the distance from the pivot to the query point is greater than the query range, the object is pruned without the need to calculate the actual distance from the object to the query point. And the distance from the object to the pivot has already been calculated in the first step of selecting pivot.

```
for oid, obj in enumerate(data):
    # prune criteria
    skip = False
    for p_idx, pivot_dist in enumerate(query_to_pivot_distances):
        if abs(distances[oid][p_idx] - pivot_dist) > epsilon:
            skip = True
            break

    # calculate real distance
    if not skip:
        dist = euclidean_distance(query, obj)
        totalcount += 1
        if dist <= epsilon:
            result.append(oid)
```

iDistance-based search approach

Similar to pivot-based, it is necessary to calculate the distance between the query point and each pivot. Add the number of calculations to the total count. But **unlike pivot-based**, here we can directly **prune the pivot**.

Because in the iDistance method, we assign objects to different pivot points, allowing us to calculate the maximum distance that each pivot contains. If the distance from the query point to the pivot is **greater than** the maximum distance within the range of the pivot **plus** the query range, it means that no object within this pivot can be included in the query range, thus achieving pruning operation on the pivot.

```
for query in queries:
    result = []
    # traverse pivots
    for pivot_idx, pivot in enumerate(pivots):
        # query to pivot
        dist_to_pivot = euclidean_distance(query, data[pivot])
        totalcount += 1

        # prune
        if dist_to_pivot - maxd_per_pivot[pivot_idx] > epsilon:
            continue
```

For the unpruned pivot, we need to check its content. But the query range can still be used for filtering, thereby reducing the calculation of actual distance. The scanning upper and lower bounds are defined as follows:

```
# not pruned
lower_bound = pivot_idx * global_maxd + max(0, dist_to_pivot - epsilon)
upper_bound = pivot_idx * global_maxd + min(maxd_per_pivot[pivot_idx], dist_to_pivot + epsilon)
```

Reason: The pivot includes a circular range, and in 'iDistance_array', the data range included in different pivots is: **starting** from the global maximum value multiplied by the ID of the pivot(pid), and **ending** at the global maximum value multiplied by the ID of the pivot(pid) plus the maximum value of the pivot's included range.

Furthermore, after obtaining the upper and lower bounds of the scan, we can use the **binary search** to quickly find the scan range in iDistance_array.

```
# binary search
iDist_values = [item[0] for item in iDistance_array]
start_idx = bisect.bisect_left(iDist_values, lower_bound)
end_idx = bisect.bisect_right(iDist_values, upper_bound)
```

Attention: When scanning within the range, a judgment condition must be attached to check whether the current scanned object belongs to the same pivot as the currently inspected pivot.

```
# scan from lower to upper
for i in range(start_idx, end_idx):
    iDist, oid, pivot_idx_in_array, pivot_id = iDistance_array[i]

    # avoid distances mistakes
    if pivot_idx_in_array == pivot_idx:
        dist = euclidean_distance(query, data[oid])
        totalcount += 1
        if dist <= epsilon:
            result.append(oid)
```

Reason: When computers calculate and save floating-point numbers, there may be accuracy issues, so there may be situations where two pivots with adjacent ID have **overlapping boundaries**. For example, the upper bound of the pixel with ID 2 coincides with the upper bound of the pixel with ID 3 (i.e., the iDistance value is the same). This will result in some **overlap** and errors in the calculation of the **starting and ending** id of the scanning range.

Partial results (ε=0.2) are as follows, consistent with the example file.

```
Average distance computations per query (Naive) (ε = 0.2)= 10000.0
Total time for Naive method (ε = 0.2) = 6.289201 seconds
Average distance computations per query (Pivots) (ε = 0.2)= 16.71
Total time for Pivots method (ε = 0.2) = 1.250262 seconds
Average distance computations per query (iDistance) (ε = 0.2)= 2074.24
Total time for iDistance method (ε = 0.2) = 1.625184 seconds
```

Query information to file

Due to the need to evaluate the results of running different parameter values in the future, we need to save the running results.

In my program, you can choose to save as a txt file or as an html file. As a demonstration, I chose an HTML file with better readability for explanation.

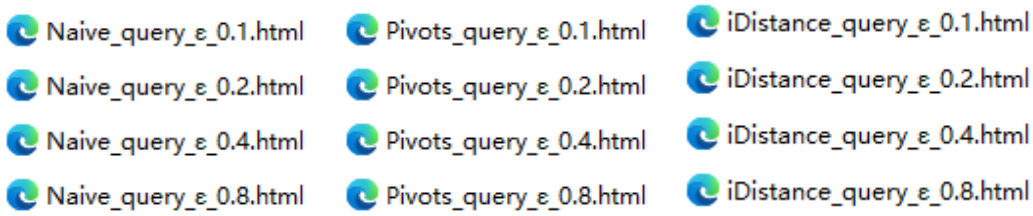
Build function **'savehtml'**. Pass the parameter Method (capturing user input) into the function. So that customize different file names and save their corresponding results through different method parameters and query range parameters.

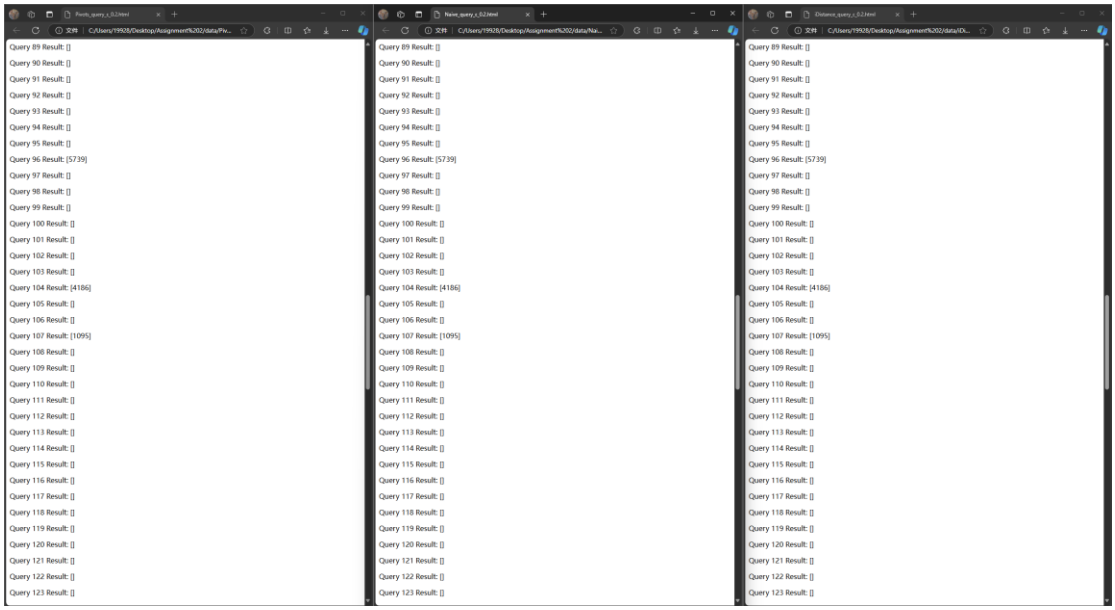
```
def savehtml(method, results, time, avg_count, param_name=None, param_value=None):

    method_dict = {1: 'Naive', 2: 'Pivots', 3: 'iDistance'}
    Method = method_dict.get(method, 'Unknown')


    file_name = f"../data/{Method}_query"
    if param_name and param_value is not None:
        file_name += f"_{param_name}_{param_value}"
    file_name += ".html"
```

The result file rendering is shown in the following figure





Simultaneously write the performance analysis results (Methods, parameter value, total running time, average distance calculation times) into a separate file. Used for subsequent performance analysis.

```python
def save_time(timing_data, file_name='time_info.txt'):
    try:
        with open(file_name, 'w') as file:
            for entry in timing_data:
                method_name = entry['method_name']
                epsilon = entry['epsilon']
                total_time = entry['total_time']
                avg_counts = entry['avg_counts']
                file.write(f"{method_name},{epsilon},{total_time:.6f},{avg_counts}\n")
    except Exception as e:
        print(f"Error saving timing info: {e}")
```

Main function design

Design a loop to capture user input, 0: run all parameter values that need to be tested at once, 1: naive method, 2: pivot method, 3: iDistance method. Simultaneously handle illegal inputs.

```python
# capture input
method = int(input("enter method (0 for all methods): "))

time_info = []

if method == 0:

    # do all
    for epsilon in [0.1, 0.2, 0.4, 0.8]:
        for m in range(1, 4):
            run_query(m, data, queries, pivots, distances, iDistance_array, global_maxd, maxd_per_pivot, ep

    save_time(time_info, file_name='../data/Range_Query_Timeinfo.txt')

elif method in [1, 2, 3]:

    epsilon = float(input("enter ε: "))
    # do certain
    run_query(method, data, queries, pivots, distances, iDistance_array, global_maxd, maxd_per_pivot, epsil
```

```python
f run_query(method, data, queries, pivots, distances, iDistance_array, global_maxd, maxd_per_pivot, epsilon,

 if method == 1:
     start_time = time.time()
     results, avg_counts = naive_range_query(data, queries, epsilon)
     total_time = time.time() - start_time
     method_name = "Naive"

 elif method == 2:
     start_time = time.time()
     results, avg_counts = pivot_range_query(data, queries, pivots, distances, epsilon)
     total_time = time.time() - start_time
     method_name = "Pivots"

 elif method == 3:
     start_time = time.time()
     results, avg_counts = iDistance_range_query(data, queries, pivots, iDistance_array, epsilon, global_ma
     total_time = time.time() - start_time
     method_name = "iDistance"

 else:
     raise ValueError("Invalid method")

 # save file
 savehtml(method, results, total_time, avg_counts, param_name='ε', param_value=epsilon)

 # print info
 print(f"Average distance computations per query ({method_name}) (ε = {epsilon})= {avg_counts}")
 print(f"Total time for {method_name} method (ε = {epsilon}) = {total_time:.6f} seconds")
```

Result correctness testing (Additional Works)

Due to the need to **verify the correctness** of our designed **algorithm**, we need to

compare the output results of the algorithm.

And since the naive method is a simple linear scan, there are no algorithm design issues, and the results are definitely correct. So it is necessary to compare the running results of the other two methods with those of the naive method. If the results are different, it proves that the algorithm has errors and outputs differential information.

Therefore, design 'compact_three_html_files' function to compare whether the results obtained by different methods are the same when the query parameters are the same.

```
def compare_three_html_files(file1, file2, file3):
    try:
        with open(file1, 'r') as f1, open(file2, 'r') as f2, open(file3, 'r') as f3:

            line1 = f1.readline().strip()
            line2 = f2.readline().strip()
            line3 = f3.readline().strip()
            line1 = f1.readline().strip()
            line2 = f2.readline().strip()
            line3 = f3.readline().strip()
            line1 = f1.readline().strip()
            line2 = f2.readline().strip()
            line3 = f3.readline().strip()
            line1 = f1.readline().strip()
            line2 = f2.readline().strip()
            line3 = f3.readline().strip()

            line_number = 1  # track line numbers

            while True:
                line1 = f1.readline().strip()
                line2 = f2.readline().strip()
                line3 = f3.readline().strip()

                # end and the same
                if not line1 and not line2 and not line3:
                    print("All three HTML files are identical.")
                    return True

                # some diff in certain line or len(file) diff
                if line1 != line2 or line2 != line3:
                    print(f"Difference found at line {line_number}:")
                    if line1 != line2:
                        print(f"File 1 and File 2 differ.\nFile 1: {line1}\nFile 2: {line2}")
                    if line2 != line3:
                        print(f"File 2 and File 3 differ.\nFile 2: {line2}\nFile 3: {line3}")
                    if line1 != line3:
                        print(f"File 1 and File 3 differ.\nFile 1: {line1}\nFile 3: {line3}")
                    return False

                line_number += 1
    except FileNotFoundError as e:
        print(f"Error: {e}")
        return False
```

We only need to input different file path parameters to perform file comparison

Step 4: KNN similarity queries

I use the maximum heap to store the K nearest neighbors in KNN. Import the **'heapq'** package in Python, but typically **heapq** is used to implement the **minimum heap**, where the smallest element in the heap is located at the top of the heap. Therefore, when saving in the heap, we need to take the **opposite number** of distances. So the element at the top of the heap is always the farthest nearest neighbor. Here the minimum heap comes to a maximum heap. If we scan closer neighbors next, we can quickly and conveniently **replace the top of the heap**.

Naïve KNN

The simplest **linear scan** and maintaining a maximum heap. Maintain continuous insertion when the number of elements in the heap does not meet K. After satisfying K (i.e. finding the current K nearest neighbors), scan the next objects and **replace the heap top** if a closer neighbor is found. Finally, sort the results and store them in the result list.

```
for query_idx, query in enumerate(queries):
    # Actually minheap
    max_heap = []

    for oid, obj in enumerate(data):
        dist = euclidean_distance(query, obj)
        totalcount += 1

        if len(max_heap) < k:
            # heap push
            heapq.heappush(max_heap, (-dist, oid))
        else:
            # replace heap top element
            if dist < -max_heap[0][0]:
                heapq.heapreplace(max_heap, (-dist, oid))

    # sort so that knn
    knn_result = sorted([(-dist, oid) for dist, oid in max_heap])
    results.append(knn_result)
```

Pivot-based KNN

There is a pruning operation similar to the previous range query here. But the pruning range here is **dynamically updated**, that is, after finding a closer neighbor, the query range is

updated to the **top element** of the heap, and subsequent objects are scanned and pruned based on the updated query range. This **dynamic query range update** can ensure fewer distance calculations.

```
for oid, obj in enumerate(data):
    # prune
    skip = False
    for pivot_idx, pivot_dist in enumerate(query_to_pivot_distances):
        # dynamic epsilon
        if len(max_heap) == k:
            epsilon = -max_heap[0][0]  # search epsilon
            if abs(distances[oid][pivot_idx] - pivot_dist) > epsilon:
                skip = True
                break

    # not pruned
    if not skip:
        dist = euclidean_distance(query, obj)
        totalcount += 1

        if len(max_heap) < k:
            heapq.heappush(max_heap, (-dist, oid))
        elif dist < -max_heap[0][0]:
            heapq.heapreplace(max_heap, (-dist, oid))
```

iDistance KNN

1. Nearest pivot scan

In the initial stage, need to first scan the nearest pivot to the current query point. Because the pivot closest to the query point is most likely to contain more nearest neighbors.

Similar to range queries, in iDistance, we need to maintain a scanning **upper and lower bound** to determine the starting the ending point of the scan.

```
# nearest pivot
nearest_pivot_idx = np.argmin(query_to_pivot_distances)
nearest_pivot_dist = query_to_pivot_distances[nearest_pivot_idx]

# dynamic scan for the nearest pivot
lower_bound = nearest_pivot_idx * global_maxd
upper_bound = nearest_pivot_idx * global_maxd + maxd_per_pivot[nearest_pivot_idx]

# binary search
iDist_values = [item[0] for item in iDistance_array]
start_idx = bisect.bisect_left(iDist_values, lower_bound)
end_idx = bisect.bisect_right(iDist_values, upper_bound)
```

Here, we also use the range query method of **dynamic range update**. Since no nearest neighbors were found in the initial stage, we set the query range to infinite.

```
# initial epsilon and dynamic update
epsilon = float('inf')
```

Furthermore, we can perform pruning operations on the object here to further reduce the number of distance calculations.

```
# prune obj
if abs(distances[oid][nearest_pivot_idx] - nearest_pivot_dist) > epsilon:
    continue
```

The next step is a **secondary verification** similar to iDistance range query, ensuring that the scanned object is consistent with the current scanning pivot. Then maintain a maximum heap to store K nearest neighbors and **dynamically update the pruning range**.

```
# secondary validate
if pivot_idx_in_array == nearest_pivot_idx:
    dist = euclidean_distance(query, data[oid])
    totalcount += 1

    if len(max_heap) < k:
        heapq.heappush(max_heap, (-dist, oid))
        epsilon = -max_heap[0][0] if len(max_heap) == k else float('inf')
    elif dist < -max_heap[0][0]:
        heapq.heapreplace(max_heap, (-dist, oid))
        epsilon = -max_heap[0][0]
```

2. Subsequent pivot scanning

After scanning the nearest pivot, we still need to scan objects of other pivots to ensure that no nearest neighbors are missed.

For other pivots, **dynamic epsilon updating** range scanning is also used. After scanning the nearest pivot, update the pruning range.

```
# update epsilon
epsilon = -max_heap[0][0] if len(max_heap) == k else float('inf')
```

Firstly, skip the nearest pivot as it has already been scanned.

```
# scan and prune pivot
for pivot_idx, pivot_dist in enumerate(query_to_pivot_distances):

    if pivot_idx == nearest_pivot_idx:
        continue  # already scaned the nearest
```

Then, we can traverse all the pivot points and prune them. For the pivot points that have not been pruned, we calculate their scanning upper and lower bounds and scanning start and end points based on the dynamically updated query range.

During the scanning process, prune the object, and the actual distance is further calculated for the uncut object. And **dynamically update the pruning range.**

```
# prune pivots
if pivot_dist - maxd_per_pivot[pivot_idx] <= epsilon:
    # not pruned
    lower_bound = pivot_idx * global_maxd + max(0, pivot_dist - epsilon)
    upper_bound = pivot_idx * global_maxd + min(maxd_per_pivot[pivot_idx], pivot_dist + epsilon)

    start_idx = bisect.bisect_left(iDist_values, lower_bound)
    end_idx = bisect.bisect_right(iDist_values, upper_bound)

    for i in range(start_idx, end_idx):
        iDist, oid, pivot_idx_in_array, pivot_id = iDistance_array[i]
        # prune obj
        if abs(distances[oid][pivot_idx] - pivot_dist) > epsilon:
            continue

        if pivot_idx_in_array == pivot_idx:

            dist = euclidean_distance(query, data[oid])
            totalcount += 1

            if len(max_heap) < k:
                heapq.heappush(max_heap, (-dist, oid))
                epsilon = -max_heap[0][0] if len(max_heap) == k else float('inf')
            elif dist < -max_heap[0][0]:
                heapq.heapreplace(max_heap, (-dist, oid))
                epsilon = -max_heap[0][0]
```

The **partial running results** (k = 5) are shown in the following figure, which is consistent with the example file.

```
Average distance computations per KNN_query (Naive) when K = 5 = 10000.0
Total time for Naive method when K = 5 = 8.332942 seconds
Average distance computations per KNN_query (Pivots) when K = 5 = 3156.76
Total time for Pivots method when K = 5 = 6.686431 seconds
Average distance computations per KNN_query (iDistance) when K = 5 = 6115.185
Total time for iDistance method when K = 5 = 5.630343 seconds
```

Query information to file & Main function & Result correctness testing (Additional)

Similar to the step3.

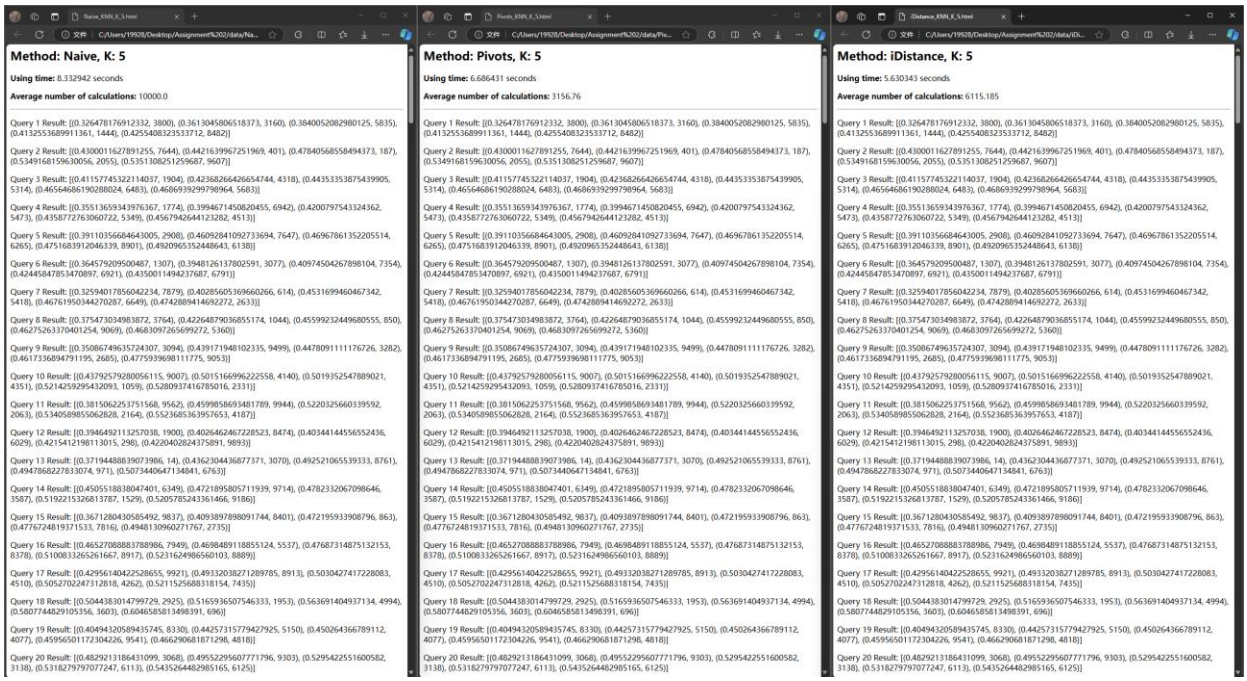
Capture user input in the main function, and if it is 0, test all parameter values.

```
if method == 0:


    for k in [1, 5, 10, 50, 100]:
        for m in range(1, 4):
            run_KNN(m, data, queries, pivots, distances, iDistance_array, global_maxd, maxd_per_pivot, k, time_info)

    save_time(time_info, file_name='../data/KNN_Timeinfo.txt')
```

Save the HTML file of the results.



Method: Naive, K: 5

Using time: 8.332942 seconds

Average number of calculations: 10000.0

Query 1 Result: [(0.3264781769123332, 3800), (0.3613045806518373, 3160), (0.3840052082980125, 5835), (0.4132553689911361, 1444), (0.4255408323533712, 8482)]

Query 2 Result: [(0.4300011627891255, 7644), (0.4421639967251969, 401), (0.4784056855894373, 187), (0.5349168159630056, 2055), (0.5351308251259687, 9607)]

Query 3 Result: [(0.4115774532211037, 1904), (0.4236826642665474, 4318), (0.4435335387543990, 5314), (0.4656468619028802, 6483), (0.4686939299798964, 5683)]

Query 4 Result: [(0.3551365934397636, 1774), (0.3994671450820455, 6942), (0.4200797543324362, 5473), (0.4358772763060722, 5349), (0.4567942644123282, 4513)]

Query 5 Result: [(0.3911035668464300, 2908), (0.4609284109273369, 7647), (0.4696786135220551, 6265), (0.4751683912046339, 8901), (0.4920965352448643, 6138)]

Query 6 Result: [(0.3645792095004870, 1307), (0.3948126137802591, 3077), (0.4097450426789810, 7354), (0.4244584785347089, 6921), (0.4350011494237687, 6791)]

Query 7 Result: [(0.3259401785604223, 7879), (0.4028560536966026, 614), (0.4531699460467342, 5418), (0.4676195034427028, 6649), (0.4742889414692272, 2633)]

Query 8 Result: [(0.3754730349833872, 3764), (0.4226487903685517, 1044), (0.4559923244968055, 850), (0.4627526337040125, 9069), (0.4683097265699272, 5360)]

Query 9 Result: [(0.3508674963572430, 3094), (0.4391719481023335, 9499), (0.4478091111176726, 3282), (0.4617336894791195, 2685), (0.4775939698111775, 9053)]

Query 10 Result: [(0.4379257928005611, 9007), (0.5015166996222558, 4140), (0.5019352547889021, 4351), (0.5214259295432093, 1059), (0.5280937416785016, 2331)]

Query 11 Result: [(0.3815062253751568, 9562), (0.4599858693481789, 9944), (0.5220325660339592, 2063), (0.5340589855062828, 2164), (0.5523685363957653, 4187)]

Query 12 Result: [(0.3946492113257038, 1900), (0.4026462467228523, 8474), (0.4034414455655243 6029), (0.4215412198113015, 298), (0.4220402824375891, 9893)]

Query 13 Result: [(0.3719448883907398 6, 14), (0.4362304436877371, 3070), (0.4925210655393 33, 8761), (0.4947868227833074, 971), (0.5073440647134841, 6763)]

Query 14 Result: [(0.4505518838047401, 6349), (0.4721895805711939, 9714), (0.4782332067098646, 3587), (0.5192215326813787, 1529), (0.5205785243361466, 9186)]

Query 15 Result: [(0.3671280430585492, 9837), (0.4093897898091744, 8401), (0.4721959339087 96, 863), (0.4776724819371533, 7816), (0.4948130960271767, 2735)]

Query 16 Result: [(0.4652708888378898 6, 7949), (0.4698489118855124, 5537), (0.4768731487513215 3, 8378), (0.5100833265261667, 8917), (0.5231624986560103, 8889)]

Query 17 Result: [(0.4295614042252865 5, 9921), (0.4933203827128978 5, 8913), (0.5030427417228083, 4510), (0.5052702247312818, 4262), (0.5211525688318154, 7435)]

Query 18 Result: [(0.5044383014799729, 2925), (0.5165936507546333, 1953), (0.5636914049371 34, 4994), (0.5807744829105356, 3603), (0.6046585813498391, 696)]

Query 19 Result: [(0.4049432058943574 5, 8330), (0.4425731577942792 5, 5150), (0.4502643667891 12, 4077), (0.4595650117230422 6, 9541), (0.4662906811871298, 4818)]

Query 20 Result: [(0.4829213186431099, 3068), (0.4955229560771796, 9303), (0.5295422551600582, 3138), (0.5318279797077247, 6113), (0.5435264482985165, 6125)]

Method: Pivots, K: 5

Using time: 6.686431 seconds

Average number of calculations: 3156.76

Query 1 Result: [(0.3264781769123332, 3800), (0.3613045806518373, 3160), (0.3840052082980125, 5835), (0.4132553689911361, 1444), (0.4255408323533712, 8482)]

Query 2 Result: [(0.4300011627891255, 7644), (0.4421639967251969, 401), (0.4784056855894373, 187), (0.5349168159630056, 2055), (0.5351308251259687, 9607)]

Query 3 Result: [(0.4115774532211037, 1904), (0.4236826642665474, 4318), (0.4435335387543990, 5314), (0.4656468619028802, 6483), (0.4686939299798964, 5683)]

Query 4 Result: [(0.3551365934397636, 1774), (0.3994671450820455, 6942), (0.4200797543324362, 5473), (0.4358772763060722, 5349), (0.4567942644123282, 4513)]

Query 5 Result: [(0.3911035668464300, 2908), (0.4609284109273369, 7647), (0.4696786135220551, 6265), (0.4751683912046339, 8901), (0.4920965352448643, 6138)]

Query 6 Result: [(0.3645792095004870, 1307), (0.3948126137802591, 3077), (0.4097450426789810, 7354), (0.4244584785347089, 6921), (0.4350011494237687, 6791)]

Query 7 Result: [(0.3259401785604223, 7879), (0.4028560536966026, 614), (0.4531699460467342, 5418), (0.4676195034427028, 6649), (0.4742889414692272, 2633)]

Query 8 Result: [(0.3754730349833872, 3764), (0.4226487903685517, 1044), (0.4559923244968055, 850), (0.4627526337040125, 9069), (0.4683097265699272, 5360)]

Query 9 Result: [(0.3508674963572430, 3094), (0.4391719481023335, 9499), (0.4478091111176726, 3282), (0.4617336894791195, 2685), (0.4775939698111775, 9053)]

Query 10 Result: [(0.4379257928005611, 9007), (0.5015166996222558, 4140), (0.5019352547889021, 4351), (0.5214259295432093, 1059), (0.5280937416785016, 2331)]

Query 11 Result: [(0.3815062253751568, 9562), (0.4599858693481789, 9944), (0.5220325660339592, 2063), (0.5340589855062828, 2164), (0.5523685363957653, 4187)]

Query 12 Result: [(0.3946492113257038, 1900), (0.4026462467228523, 8474), (0.4034414455655243 6029), (0.4215412198113015, 298), (0.4220402824375891, 9893)]

Query 13 Result: [(0.3719448883907398 6, 14), (0.4362304436877371, 3070), (0.4925210655393 33, 8761), (0.4947868227833074, 971), (0.5073440647134841, 6763)]

Query 14 Result: [(0.4505518838047401, 6349), (0.4721895805711939, 9714), (0.4782332067098646, 3587), (0.5192215326813787, 1529), (0.5205785243361466, 9186)]

Query 15 Result: [(0.3671280430585492, 9837), (0.4093897898091744, 8401), (0.4721959339087 96, 863), (0.4776724819371533, 7816), (0.4948130960271767, 2735)]

Query 16 Result: [(0.4652708888378898 6, 7949), (0.4698489118855124, 5537), (0.4768731487513215 3, 8378), (0.5100833265261667, 8917), (0.5231624986560103, 8889)]

Query 17 Result: [(0.4295614042252865 5, 9921), (0.4933203827128978 5, 8913), (0.5030427417228083, 4510), (0.5052702247312818, 4262), (0.5211525688318154, 7435)]

Query 18 Result: [(0.5044383014799729, 2925), (0.5165936507546333, 1953), (0.5636914049371 34, 4994), (0.5807744829105356, 3603), (0.6046585813498391, 696)]

Query 19 Result: [(0.4049432058943574 5, 8330), (0.4425731577942792 5, 5150), (0.4502643667891 12, 4077), (0.4595650117230422 6, 9541), (0.4662906811871298, 4818)]

Query 20 Result: [(0.4829213186431099, 3068), (0.4955229560771796, 9303), (0.5295422551600582, 3138), (0.5318279797077247, 6113), (0.5435264482985165, 6125)]

Method: iDistance, K: 5

Using time: 5.630343 seconds

Average number of calculations: 6115.185

Query 1 Result: [(0.3264781769123332, 3800), (0.3613045806518373, 3160), (0.3840052082980125, 5835), (0.4132553689911361, 1444), (0.4255408323533712, 8482)]

Query 2 Result: [(0.4300011627891255, 7644), (0.4421639967251969, 401), (0.4784056855894373, 187), (0.5349168159630056, 2055), (0.5351308251259687, 9607)]

Query 3 Result: [(0.4115774532211037, 1904), (0.4236826642665474, 4318), (0.4435335387543990, 5314), (0.4656468619028802, 6483), (0.4686939299798964, 5683)]

Query 4 Result: [(0.3551365934397636, 1774), (0.3994671450820455, 6942), (0.4200797543324362, 5473), (0.4358772763060722, 5349), (0.4567942644123282, 4513)]

Query 5 Result: [(0.3911035668464300, 2908), (0.4609284109273369, 7647), (0.4696786135220551, 6265), (0.4751683912046339, 8901), (0.4920965352448643, 6138)]

Query 6 Result: [(0.3645792095004870, 1307), (0.3948126137802591, 3077), (0.4097450426789810, 7354), (0.4244584785347089, 6921), (0.4350011494237687, 6791)]

Query 7 Result: [(0.3259401785604223, 7879), (0.4028560536966026, 614), (0.4531699460467342, 5418), (0.4676195034427028, 6649), (0.4742889414692272, 2633)]

Query 8 Result: [(0.3754730349833872, 3764), (0.4226487903685517, 1044), (0.4559923244968055, 850), (0.4627526337040125, 9069), (0.4683097265699272, 5360)]

Query 9 Result: [(0.3508674963572430, 3094), (0.4391719481023335, 9499), (0.4478091111176726, 3282), (0.4617336894791195, 2685), (0.4775939698111775, 9053)]

Query 10 Result: [(0.4379257928005611, 9007), (0.5015166996222558, 4140), (0.5019352547889021, 4351), (0.5214259295432093, 1059), (0.5280937416785016, 2331)]

Query 11 Result: [(0.3815062253751568, 9562), (0.4599858693481789, 9944), (0.5220325660339592, 2063), (0.5340589855062828, 2164), (0.5523685363957653, 4187)]

Query 12 Result: [(0.3946492113257038, 1900), (0.4026462467228523, 8474), (0.4034414455655243 6029), (0.4215412198113015, 298), (0.4220402824375891, 9893)]

Query 13 Result: [(0.3719448883907398 6, 14), (0.4362304436877371, 3070), (0.4925210655393 33, 8761), (0.4947868227833074, 971), (0.5073440647134841, 6763)]

Query 14 Result: [(0.4505518838047401, 6349), (0.4721895805711939, 9714), (0.4782332067098646, 3587), (0.5192215326813787, 1529), (0.5205785243361466, 9186)]

Query 15 Result: [(0.3671280430585492, 9837), (0.4093897898091744, 8401), (0.4721959339087 96, 863), (0.4776724819371533, 7816), (0.4948130960271767, 2735)]

Query 16 Result: [(0.4652708888378898 6, 7949), (0.4698489118855124, 5537), (0.4768731487513215 3, 8378), (0.5100833265261667, 8917), (0.5231624986560103, 8889)]

Query 17 Result: [(0.4295614042252865 5, 9921), (0.4933203827128978 5, 8913), (0.5030427417228083, 4510), (0.5052702247312818, 4262), (0.5211525688318154, 7435)]

Query 18 Result: [(0.5044383014799729, 2925), (0.5165936507546333, 1953), (0.5636914049371 34, 4994), (0.5807744829105356, 3603), (0.6046585813498391, 696)]

Query 19 Result: [(0.4049432058943574 5, 8330), (0.4425731577942792 5, 5150), (0.4502643667891 12, 4077), (0.4595650117230422 6, 9541), (0.4662906811871298, 4818)]

Query 20 Result: [(0.4829213186431099, 3068), (0.4955229560771796, 9303), (0.5295422551600582, 3138), (0.5318279797077247, 6113), (0.5435264482985165, 6125)]

KNN_Timeinfo.txt

文件 编辑 查看

Naive,1,8.186104,10000.0
Pivots,1,4.169530,1325.075
iDistance,1,4.296897,4640.855
Naive,5,8.332942,10000.0
Pivots,5,6.686431,3156.76
iDistance,5,5.630343,6115.185
Naive,10,8.271145,10000.0
Pivots,10,8.166499,4139.145
iDistance,10,6.390158,6732.42
Naive,50,8.259766,10000.0
Pivots,50,11.027671,6653.57
iDistance,50,7.725077,8167.965
Naive,100,8.597272,10000.0
Pivots,100,12.388435,7663.865
iDistance,100,8.174539,8729.285

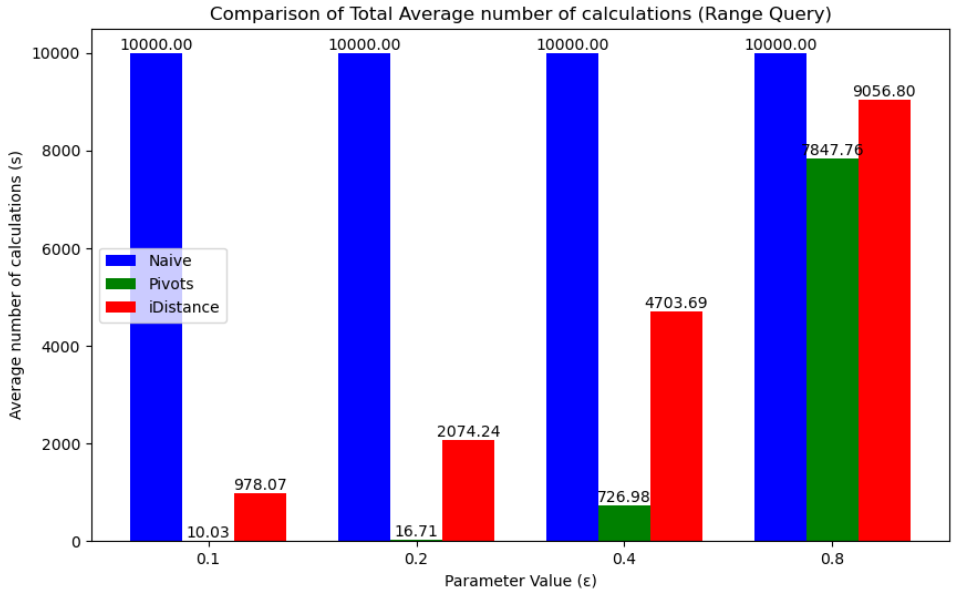
Step 5: Evaluation and Observations

Since we have previously saved the performance parameters of different methods in a file, we can directly read the performance parameters in the file for plotting.

Time and Calculation Times Comparison (Range Query)

1. Calculation Times Comparison

The **average number of distance calculations** required for different methods to perform range queries under different parameter values is shown in the following figure.

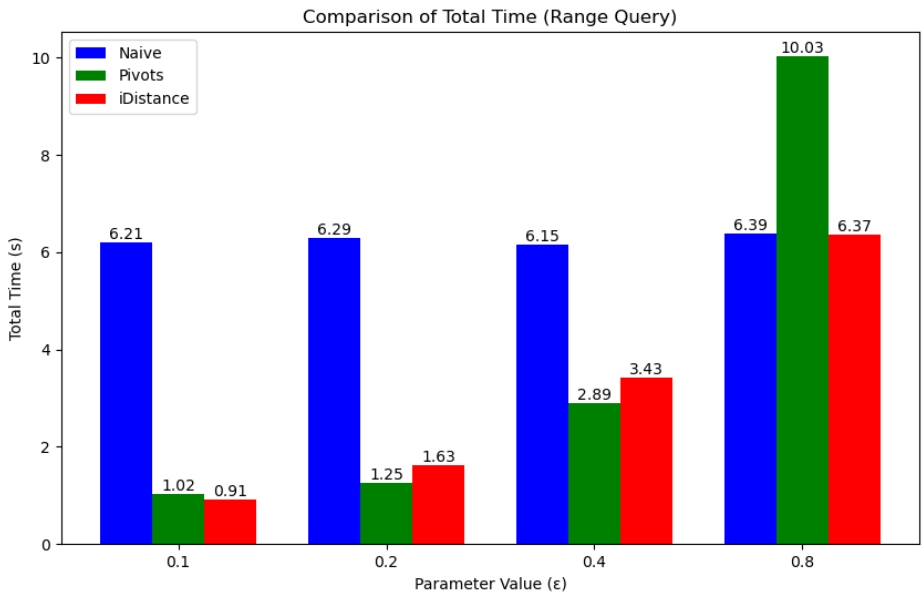


It is not difficult to observe that, both the pivot-based method and the iDistance-based method have significantly fewer average distance calculations compared to the naive method.

Reason: Pruning operation can effectively reduce the number of actual distance calculations.

2. Total Time Comparison

The **Total running Time** required for different methods to perform range queries under different parameter values is shown in the following figure.



We found that:

Although the latter two methods require many fewer distance calculations than the naive method, the time used increases with the increase of the query range, and ultimately even the pivot-based method takes more time than the naive method.

Moreover, as the query (ε) scope continues to expand, the increase in time consumption of iDistance-based methods is far less than that of pivot-based methods

Reason:

Both pivot-based and iDistance-based methods introduce **pruning operations**, which involve **a large number of judgment statements** (if) to determine whether pruning conditions are met. And it is precisely conditional judgment statements that cause a significant amount of **CPU clock overhead**. Because modern CPUs use **branch prediction** methods to process conditional judgment statements, that is, to guess the execution path of the program in advance, in order to avoid the processor waiting for the actual condition judgment result at the branch point, thereby reducing the waste of clock cycles. If the branch prediction is correct, it will consume less time (1-3 clock cycles), but if the **branch prediction is incorrect**, branch correction is required, resulting in **additional clock cycle overhead** (10-20 clock cycles).

At the same time, when the **query scope (ε) is relatively large**, it is not possible to prune many objects, which results in retaining many actual distance calculations while introducing a large number of conditional judgment statements.

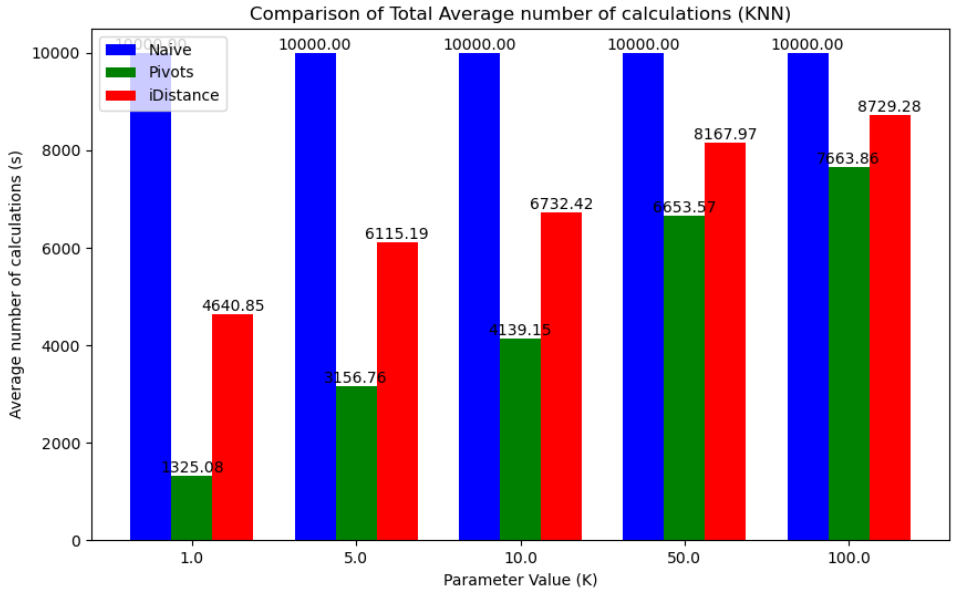
Moreover, Due to the fact that iDistance-based methods **not only prune objects, but**

also prune pivots, this can result in significantly fewer actual distance calculations and conditional statement judgments for iDistance-based methods compared to pivot-based methods.

Time and Calculation Times Comparison (KNN)

1. Calculation Times Comparison

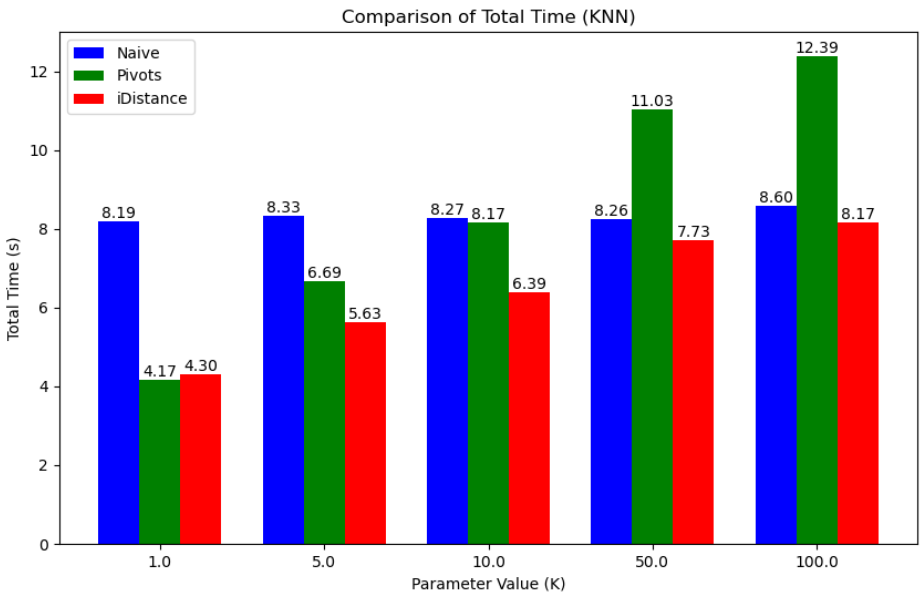
The **average number of distance calculations** required for different methods to perform KNN search under different parameter values (k) is shown in the following figure.



The Observations and Reasons is similar to the Range Query.

2. Total Time Comparison

The **Total running Time** required for different methods to perform KNN search under different parameter values (k) is shown in the following figure.



The Observations and Reasons is also similar to the Range Query.

Appendix

The main benefits of using **'pickle'** files to store data include:

1. Supports complex objects: Can serialize and deserialize complex data structures and custom objects in Python, such as class instances and nested data structures.

2. Easy to use: Easily save and load data through ‘pickle. Dump’ and ‘pickle. Load’.

3. Efficient Performance: Using binary format, fast read and write speed, and small file size.

4. Integrity preservation: After serialization, it can faithfully restore object state and reference relationships.

Pickle is particularly suitable for storing models, temporary data, or large datasets that require frequent reading and writing.