Contents

[Foreword 2](#_Toc180415947)

[Step 1: Select Pivots 2](#_Toc180415948)

[Step 2: iDistance 4](#_Toc180415949)

[Step 3: Range similarity queries 6](#_Toc180415950)

[Naïve approach 6](#_Toc180415951)

[Pivot-based search approach 6](#_Toc180415952)

[iDistance-based search approach 7](#_Toc180415953)

[Query information to file 9](#_Toc180415954)

[Main function design 11](#_Toc180415955)

[Result correctness testing (Additional Works) 11](#_Toc180415956)

[Step 4: KNN similarity queries 13](#_Toc180415957)

[Naïve KNN 13](#_Toc180415958)

[Pivot-based KNN 13](#_Toc180415959)

[iDistance KNN 14](#_Toc180415960)

[Query information to file & Main function & Result correctness testing (Additional) 17](#_Toc180415961)

[Step 5: Evaluation and Observations 19](#_Toc180415962)

[Time and Calculation Times Comparison (Range Query) 19](#_Toc180415963)

[Time and Calculation Times Comparison (KNN) 21](#_Toc180415964)

[Appendix 22](#_Toc180415965)

# 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.

文本

描述已自动生成

Then, create a function **‘euclidean\_distance’** to calculate the Euclidean distance.

图形用户界面, 文本

描述已自动生成

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.

文本

描述已自动生成

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.

文本

描述已自动生成

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

文本

描述已自动生成

Finally, save all data in a picklel file. [The reasons can be seen in the Appendix.](#appendix1)

文本

描述已自动生成

**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.

文本

描述已自动生成

# 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.

文本

描述已自动生成 文本

描述已自动生成

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.

文本

描述已自动生成

After obtaining the global maximum value, the iDistance of each object can be calculated. That is:

文本

描述已自动生成

After calculating iDistance, I added the index of the nearest pivot to the object in the pivot array (i.e., the ith 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.



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.

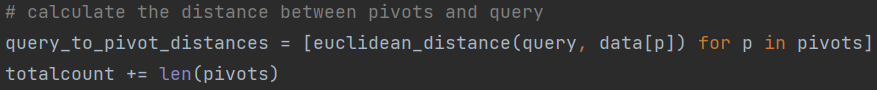
And because it is a traversal, the average distance calculation times are equal to the length of the dataset.

文本

描述已自动生成

## 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.



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.

文本

描述已自动生成

## iDistance-based search approach

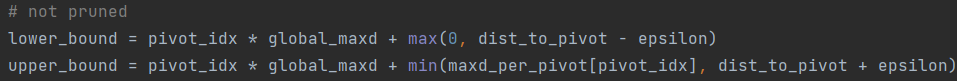
Similar to pivot-based, it is necessary to calculate the distance between the query point and each pivot. Add the number of calculations to the total count. But **unlike pivot-based**, here we can directly **prune the pivot**.

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.

文本

描述已自动生成

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:



**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.

文本

描述已自动生成

**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.

文本

描述已自动生成

**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** (ɛ=0.2) are as follows, consistent with the example file.

文本

描述已自动生成

## 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.

屏幕的截图

描述已自动生成

The result file rendering is shown in the following figure

文本

描述已自动生成 文本

描述已自动生成手机屏幕截图

描述已自动生成

电脑萤幕的截图

描述已自动生成

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.

文本

描述已自动生成

图形用户界面, 文本, 应用程序

描述已自动生成

## 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.

文本

描述已自动生成

文本

描述已自动生成

## 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.

文本

描述已自动生成

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.

文本

描述已自动生成

## 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.

文本

描述已自动生成

## 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.

文本

描述已自动生成

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.

文本

描述已自动生成

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

文本

描述已自动生成

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**.

文本

描述已自动生成

1. 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.



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

文本

描述已自动生成

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.**

文本

描述已自动生成

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

电脑屏幕的照片

低可信度描述已自动生成

## 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.

文本

描述已自动生成

Save the HTML file of the results.

文本

描述已自动生成文本

描述已自动生成文本

描述已自动生成

电脑屏幕截图

描述已自动生成

文本

描述已自动生成

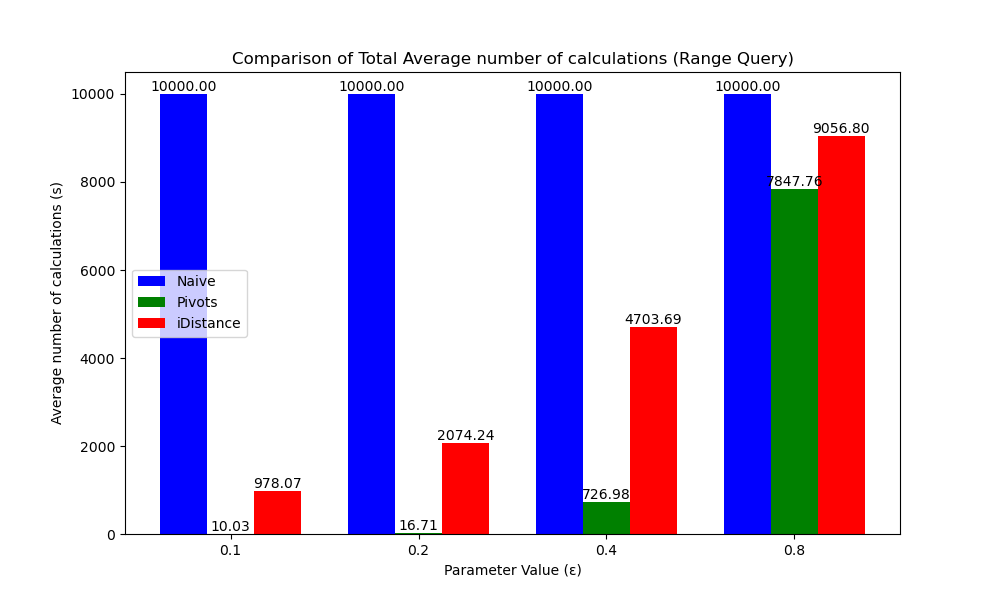
# Step 5: Evaluation and Observations

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.

1. 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.

图表, 条形图

描述已自动生成

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 (ɛ) 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 judgmentstatements 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).

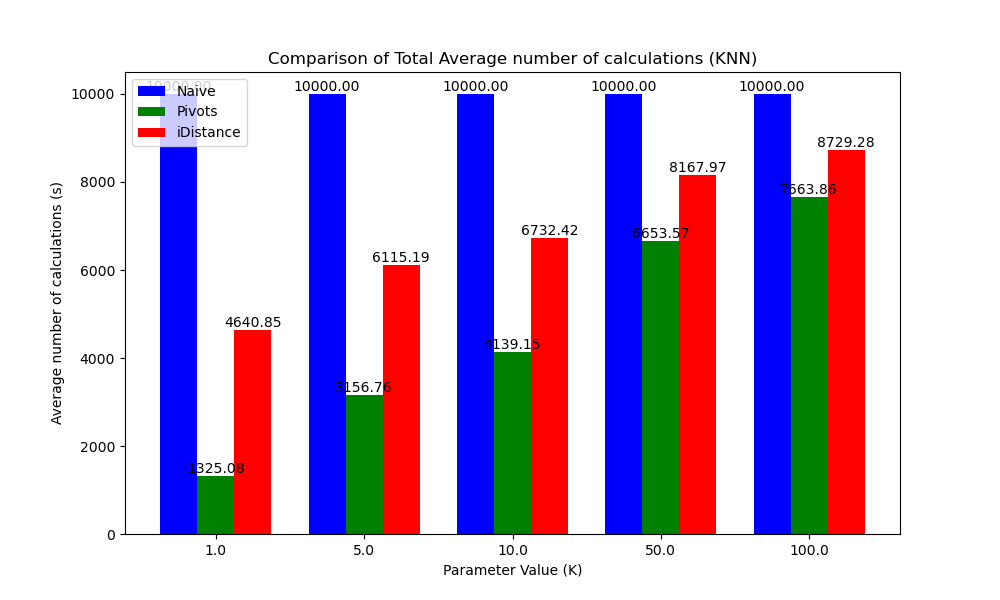
At the same time, when the **query scope (ɛ) is relatively large**, it is not possible to prune many objects, which results in retaining many actual distance calculations while introducing a large number of conditional judgment statements.

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.

1. 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.

图表, 条形图

描述已自动生成

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.