k-Nearest Neighbors (k-NN) Implementation

Ioannis Michalainas, Iasonas Lamprinidis

October 2024

Abstract

This project involves implementing the k-NN algorithm to find the nearest neighbors of a set of query points Q relative to a corpus set C in a high-dimensional space. Given a set of M-point corpus data and N-point query data, both in D-dimensional space, the algorithm identifies the k-nearest neighbors of each query point. Using optimized matrix operations and the CBLAS library, our implementation efficiently handles high-dimensional distance calculations.

1 Problem Statement

The objective of this project is to implement a subroutine that computes the **k-nearest neighbors (k-NN)** of each query point in Q based on their distances to the data points in C. In this problem, we assume that $Q = C$.

To calculate the distances, we use the following formula:

$$D = \sqrt{C^2 - 2CQ^T + (Q^2)^T} \tag{1}$$

where:

- C is the set of data points (corpus).
- Q is the set of query points.
- D is the distance matrix containing distances between each pair of points from C and Q.

Each row of the $N \times M$ (queries $\times$ corpus) matrix D contains distances from a query point to all corpus points. We then use the quickselect algorithm to retrieve the k smallest distances in $O(n)$ time.

2 Example

In this section, we illustrate the process of generating random data points and calculating the distance matrix for k-NN using C code snippets. For clarity, let's assume we have sets C and Q in d-dimensional space.

2.1 Random Data Generation

We begin by generating random data points for both the dataset C and query set Q. The following function creates a dataset with a specified number of points and dimensions:

Listing 1: Generating random data points

```c
void random_data(Mat* dataset, size_t points, size_t dimensions) {

  srand(time(NULL) + (uintptr_t)dataset);

  dataset->data = (double*)malloc(points*dimensions*sizeof(double));
  dataset->rows = points;
  dataset->cols = dimensions;
```

```
  for(size_t i=0; i<points*dimensions; i++) {
    dataset->data[i] = ((double)rand()/RAND_MAX)*100.0;
  }
}
```

For example, with points = 5 and dimensions = 2, a possible generated dataset could be:

$$C = \begin{bmatrix} 13.65 & 37.00 \\ 37.98 & 48.09 \\ 42.54 & 9.01 \\ 77.50 & 62.99 \\ 90.04 & 94.99 \end{bmatrix}$$

And the query dataset Q with 4 points could be:

$$Q = \begin{bmatrix} 60.71 & 9.11 \\ 6.63 & 22.71 \\ 81.77 & 65.85 \\ 83.09 & 57.47 \end{bmatrix}$$

2.2 Distance Calculation

We compute the distance matrix D using the following function, which leverages CBLAS for efficient matrix operations:

Listing 2: Calculating distances using CBLAS

```
void calculate_distances(const Mat *C, const Mat *Q, Mat *D) {

  int c = (int)C->rows;
  int q = (int)Q->rows;
  int d = (int)C->cols;

  double *C2 = (double *)malloc(c * sizeof(double));
  for (int i = 0; i < c; i++) {
    C2[i] = 0;
    for (int j = 0; j < d; j++) {
      C2[i] += C->data[i * d + j] * C->data[i * d + j];
    }
  }

  double *Q2 = (double *)malloc(q * sizeof(double));
  for (int i = 0; i < q; i++) {
    Q2[i] = 0;
    for (int j = 0; j < d; j++) {
      Q2[i] += Q->data[i * d + j] * Q->data[i * d + j];
    }
  }

  double *CQ = (double *)malloc(c * q * sizeof(double));
  cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasTrans, c, q, d, -2.0, C->data,
              d, Q->data, d, 0.0, CQ, q);

  for (int i = 0; i < c; i++) {
    for (int j = 0; j < q; j++) {
      CQ[i * q + j] += C2[i] + Q2[j];
      D->data[j * c + i] = sqrt(CQ[i * q + j]);
    }
  }

  free(C2);
  free(Q2);
  free(CQ);
}
```

Using this data, the computed distance matrix D between each point in C and Q is:

$$D = \begin{bmatrix} 54.70 & 45.11 & 18.17 & 56.43 & 90.75 \\ 15.92 & 40.34 & 38.44 & 81.52 & 110.37 \\ 73.98 & 47.25 & 69.07 & 5.14 & 30.29 \\ 72.40 & 46.07 & 63.19 & 7.85 & 38.16 \end{bmatrix}$$

Each entry $D[i][j]$ in this matrix represents the distance between the i-th point in Q and the j-th point in C.

2.3 Finding k-Nearest Neighbors

To find the k-nearest neighbors, we use the **quickselect** algorithm, which efficiently identifies the smallest k elements in $O(n)$ time.

Listing 3: Quickselect algorithm for k smallest distances

```
void swap(double* arr, int i, int j) {
  double temp = arr[i];
  arr[i] = arr[j];
  arr[j] = temp;
}

int partition(double* arr, int left, int right) {
  double pivot = arr[right];
  int i = left;
  for (int j = left; j < right; j++) {
    if (arr[j] < pivot) {
      swap(arr, i, j);
      i++;
    }
  }
  swap(arr, i, right);
  return i;
}

void quickSelect(double* arr, int left, int right, int k, double* result) {
  if (left <= right) {
    int pivotIndex = partition(arr, left, right);

    if (pivotIndex == k - 1) {
      for (int i = 0; i < k; i++) {
        result[i] = arr[i];
      }
      return;
    } else if (k - 1 < pivotIndex) {
      quickSelect(arr, left, pivotIndex - 1, k, result);
    } else {
      quickSelect(arr, pivotIndex + 1, right, k, result);
    }
  }
}
```

In this example, we find the 3 nearest neighbors (k = 3) for each point in Q based on the distance matrix D:

$$N = \begin{bmatrix} 18.17 & 45.11 & 54.70 \\ 15.92 & 38.44 & 40.34 \\ 5.14 & 30.29 & 47.25 \\ 7.85 & 38.16 & 46.07 \end{bmatrix}$$

In this matrix: - The first row indicates that for the first query point, the 3 nearest neighbors are 18.17, 45.11, and 54.70. - The second row shows the 3 nearest neighbors for the second query point, and so forth.

This modified section now accurately represents your example data and the computed nearest neighbors using the quickselect algorithm. Let me know if you need further customization!

3 Summary

The following steps summarize the process:

a. Calculated squared terms for data points C and query points Q.

b. Computed the dot product CQ^T to facilitate the distance calculation.

c. Combined results to obtain the distance matrix D.

d. Identified the k-nearest neighbors for each query point using quickselect.

4 Conclusion

This project demonstrates an efficient k-NN algorithm implementation that uses advanced matrix operations and sorting techniques. This approach enables accurate neighbor searches in high-dimensional spaces, achieving a balance between clarity and computational efficiency.