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

```
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 = 4 and dimensions = 2, a possible generated dataset could be:

$$C = \begin{bmatrix} 23.45 & 12.34 \\ 65.23 & 43.67 \\ 32.98 & 77.54 \\ 54.21 & 11.29 \end{bmatrix}$$

Assuming $Q = C$, we have:

$$Q = \begin{bmatrix} 23.45 & 12.34 \\ 65.23 & 43.67 \\ 32.98 & 77.54 \\ 54.21 & 11.29 \end{bmatrix}$$

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);
}
```

Given our dataset C and query set Q, we can compute the distance matrix D as follows:

$$D = \begin{bmatrix} 0.00 & 68.56 & 60.17 & 5.26 \\ 68.56 & 0.00 & 96.54 & 52.16 \\ 60.17 & 96.54 & 0.00 & 66.30 \\ 5.26 & 52.16 & 66.30 & 0.00 \end{bmatrix}$$

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

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 essence, the quickselect algorithm partitions the array around a pivot, recursively narrowing the search to find the k smallest elements.

Assuming we are finding the 2 nearest neighbors (k = 2) for each point based on the distance matrix D:

$$D = \begin{bmatrix} 0.00 & 68.56 & 60.17 & 5.26 \\ 68.56 & 0.00 & 96.54 & 52.16 \\ 60.17 & 96.54 & 0.00 & 66.30 \\ 5.26 & 52.16 & 66.30 & 0.00 \end{bmatrix}$$

The resulting matrix N, containing the indices of the 2 nearest neighbors for each query point, would look like this:

$$N = \begin{bmatrix} 3 & 2 \\ 4 & 1 \\ 4 & 1 \\ 1 & 2 \end{bmatrix}$$

In this matrix: - The first row indicates that for the first query point (corresponding to $C[0]$), the 2 nearest neighbors are $C[3]$ and $C[2]$. - The second row shows that for the second query point, the nearest neighbors are $C[4]$ and $C[1]$, and so forth.

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.