k-Nearest Neighbors (k-NN) Implementation

Ioannis Michalainas, Iasonas Lamprinidis

October/November 2024

Abstract

This C project implements the k-NN algorithm to identify the nearest neighbors of a set of query points Q relative to a corpus set C in a high-dimensional space. Given a corpus of c points and q query points, both in d-dimensional space, the algorithm efficiently determines the k-nearest neighbors for each query point. The implementation leverages optimized matrix operations, parallel processing, and approximation techniques to handle large datasets and high-dimensional distance calculations efficiently.

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.

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

Each row of the $q \times c$ matrix D contains distances from a query point to all corpus points. We then use the quickselect algorithm to retrieve the k smallest distances and their corresponding indices in $O(n)$ time.

2 Example

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

2.1 Input

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

```c
void random_input(Mat* matrix, size_t points, size_t dimensions) {

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

  matrix->data = (double*)malloc(points*dimensions*sizeof(double));
  memory_check(matrix->data);

  matrix->rows = points;
  matrix->cols = dimensions;

  for (size_t i = 0; i < points * dimensions; i++) {
    // Scale to [0, 200], then shift to [-100, 100]
    matrix->data[i] = ((double)rand()/RAND_MAX)*200.0 - 100.0;
  }
}
```

Listing 1: Generating random data points

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

$$C = \begin{bmatrix} 98.87 & 77.36 \\ 85.86 & 21.03 \\ -61.65 & -54.45 \\ -57.33 & 76.06 \\ 30.87 & 66.55 \end{bmatrix}$$

And the query dataset Q with 4 points could be:

$$Q = \begin{bmatrix} 92.90 & 21.38 \\ -76.71 & -29.80 \\ -83.48 & -40.61 \\ -46.21 & 64.69 \end{bmatrix}$$

2.2 Distance Calculation

For smaller datasets, computing the distance matrix is straightforward. Using the formula

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

we compute matrix D (queries $\times$ corpus) that represents the Euclidean distance for each query q to each corpus point c.

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

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

  double* C2 = (double*)malloc(c*sizeof(double));
  double  Q2;
  memory_check(C2);

  #pragma omp parallel for
  for(int i=0; i<c; i++) {
    double sum = 0.0;
    for(int j=0; j<d; j++) {
      sum += C->data[i*d + j]*C->data[i*d + j];
    }
    C2[i] = sum;
  }

  double sum = 0.0;
  for(int j=0; j<d; j++) {
    sum += Q->data[j]*Q->data[j];
  }
  Q2 = sum;

  double* CQ = (double*)malloc(c*sizeof(double));
  memory_check(CQ);

  cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, c, 1, d, 1.0, C->data, d, Q->data,
      1, 0.0, CQ, 1);

  #pragma omp parallel for
  for(int i=0; i<c; i++) {

    D[i] = C2[i] - 2*CQ[i] + Q2;
    if(D[i] < 0.0) {
      D[i] = 0.0;
    }
  }

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

Listing 2: Calculating distances using CBLAS

The computed distance matrix D between each point in C and Q is:

$$D = \begin{bmatrix} 56.30 & 7.05 & 76.74 \\ 28.89 & 107.62 & 144.42 \\ 25.85 & 119.57 & 156.71 \\ 15.91 & 77.10 & 120.14 \end{bmatrix}$$

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 and returns their values, as well as their indices. The 'findKNN' function calculates the distance from every query to every corpus point, finds the k-nearest neighbors, and stores them in matrix N.

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

$$N = \begin{bmatrix} 56.30 - id0 & 7.05 - id1 & 76.74 - id4 \\ 28.89 - id2 & 107.62 - id3 & 144.42 - id4 \\ 25.85 - id2 & 119.57 - id3 & 156.71 - id4 \\ 15.91 - id3 & 77.10 - id4 & 120.14 - id2 \end{bmatrix}$$

In this matrix: - The first row indicates that for the first query point, the 3 nearest neighbors are 56.30, 7.05, and 76.74, with indexes 0, 1, and 4 respectively. - The second row shows the 3 nearest neighbors for the second query point, and so forth.

3 Fine Lines

3.1 Parallelism

If matrices C and Q are very large, there is a chance that matrix D (queries $\times$ corpus) does not fit in memory. To combat this, we calculate matrix D in segments (slices), use each slice to find the k-NN of the corresponding query, and then discard the slice to make room in memory. This way, we prevent running out of memory. To boost execution speed, we parallelize the workload of computing the D slice and running quickselect on it.

```
void findKNN(Mat* C, Mat* Q, Neighbor* N, int k) {

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

  #pragma omp parallel for
  for(int i=0; i<q; i++) {

    long double* D = (long double*)malloc(c*sizeof(long double));
    memory_check(D);

    calculate_distances(C, &(Mat){.rows = 1, .cols = d, .data = Q->data + i*d}, D);

    int* indices = (int*)malloc(c*sizeof(int));
    memory_check(indices);

    for(int j=0; j<c; j++) {
      indices[j] = j;
    }

    quickSelect(D, indices, 0, c-1, k, N + i*k);
    free(D);
    free(indices);
  }
}
```

Listing 3: Routine for computing the k smallest distances

3.2 Problem Minimization

When the datasets become arbitrarily large, we encounter some problems: A problem that arises as we scale up is computational complexity. To remedy the extreme computational cost and the curse of high dimensions, we use random projection, based on the Johnson–Lindenstrauss lemma, which proves we can reduce dimensions by projecting our matrices to lower dimensions.

```c
void random_projection(Mat* M, int t, Mat* RP) {

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

  int n = (int)M->rows;
  int d = (int)M->cols;

  RP->rows = n;
  RP->cols = t;
  RP->data = (double*)malloc(n*t*sizeof(double));

  double* R = (double*)malloc(d*t*sizeof(double));
  for (int i=0; i<d*t; i++) {
    // Rademacher distribution (-1 or +1)
    R[i] = (rand()%2 == 0 ? -1 : 1) / sqrt((double)t);
  }

  cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, n, t, d, 1.0, M->data, d, R, t,
      0.0, RP->data, t);

  free(R);
}
```

Listing 4: Routine for applying Random Projection

It is of great importance to choose the target dimensionality t carefully when using random projection. Based on the lemma, $t = \frac{\log(c)}{\epsilon^2}$, where ϵ is the allowed error and c is the number of corpus points, is a good approximation to retain accuracy while significantly reducing computational cost. Note that target dimensionality t does not depend on the actual dimensionality d. In cases when c ¿¿ d, we cannot minimize the problem using random projection, instead we truncate matrix C, keeping a percentage of its points.

```c
  double const e = 0.3;
  int    const t = log((double)c) / (e*e);
  if(t<d && (c>1000 && d>100)) {

    Mat C_RP, Q_RP;
    printf("Target dimension (t) for random projection: %d\n", t);
    random_projection(&C, t, &C_RP);
    random_projection(&Q, t, &Q_RP);

    findKNN(&C_RP, &Q_RP, N, k);
    print_neighbors(N, q, k);

    free(C_RP.data);
    free(Q_RP.data);
  } else if(c>100000) {

    Mat C_TR;
    double const perc = 0.7;
    printf("Truncation percentage: %.1f\n", 1-perc);
    truncMat(&C, &C_TR, perc);

    findKNN(&C_TR, &Q, N, k);
    print_neighbors(N, q, k);

    free(C_TR.data);
  } else {

    findKNN(&C, &Q, N, k);
    print_neighbors(N, q, k);
  }
```

Listing 5: Approximation Check

Applying these minimization techniques is valuable only when the dataset is large. If the dataset is relatively small, we proceed with exact calculation.

3.3 Details

We avoid computing the square root of the distances we calculate in 'calculate$_d$*istances*'.*Instead*, *wecomparesquareddistances* 0 and $b > 0$: $a > b \iff a^2 > b^2$) and then return their square root. This way, instead of $c \times q$ sqrt calculations, we make $k \times q$ (substantially less).

```
void quickSelect(long double* arr, int* indices, int left, int right, int k, Neighbor*
    result) {

  if(left <= right) {
    int pivotIndex = partition(arr, indices, left, right);

    if(pivotIndex == k-1) {

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

Listing 6: Routine for applying Random Projection

4 Summary

The following steps summarize the process:

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

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

c. Combine results to obtain the distance matrix D.

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

5 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.

6 Tools

In this project, we used the following tools:

1. The C programming language, its compiler, and standard libraries.
2. The linear algebra library OpenBLAS.

3. The file reading libraries HDF5 and Matio.
4. The parallel programming libraries OpenMP, OpenCilk, and Pthreads.
5. The version controll sftware Git.
6. The AI assistant Github Copilot.
7. The GNU/Linux OS.
8. The Neovim text editor.

7 Sources

Random Projection:

- `https://en.wikipedia.org/wiki/Random_projection`

Johnson-Lindenstrauss Lemma:

- `https://en.wikipedia.org/wiki/Johnson%E2%80%93Lindenstrauss_lemma`
- `https://cs.stanford.edu/people/mmahoney/cs369m/Lectures/lecture1.pdf`
- `https://www.youtube.com/watch?v=j9qbuGSjzeE`

ANN Benchmarks:

- `https://github.com/erikbern/ann-benchmarks?tab=readme-ov-file`