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 uses 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. *This implementation also works when $Q = C$.*

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 (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 (or .hdf5). 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. In larger datasets, we typically compute matrix D in chunks of size 300, as this size was found to be optimal for balancing parallelism on our machine while avoiding unnecessary overhead. Further details on the calculation of matrix D for larger datasets will be provided in the following chapter. For smaller input datasets, matrix D is computed in its entirety.

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

  int c = C->rows;
  int d = C->cols;
  int batch_size = end_idx - start_idx;

  double* C2 = (double*)malloc(c*sizeof(double));
  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* Q2 = (double*)malloc(batch_size*sizeof(double));
  memory_check(Q2);
  #pragma omp parallel for
  for(int i=0; i<batch_size; i++) {
    double sum = 0.0;
    for(int j=0; j<d; j++) {
      sum += Q->data[(start_idx+i)*d + j] * Q->data[(start_idx+i)*d + j];
    }
    Q2[i] = sum;
  }

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

  cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasTrans, c, batch_size, d,
              1.0, C->data, d, Q->data + start_idx * d, d, 0.0, CQ, batch_size);

  #pragma omp parallel for collapse(2)
  for(int i=0; i<c; i++) {
    for(int j=0; j<batch_size; j++) {
      D[j*c + i] = C2[i] - 2.0*CQ[i*batch_size + j] + Q2[j];
      if(D[j*c + i] < 0.0) {
        D[j*c + i] = 0.0;
      }
    }
  }

  free(C2);
  free(Q2);
  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.

```
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 3: Quick Select algorithm implementation

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 (chunks), use each chunk to find the k-NN of the corresponding queries, and then discard the chunk 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 chunk and running quickselect on it. We also use parallelism in order to calculate distances quicker. Chunk size is set to be small enough to be easily stored in memory and large enough to avoid unnecessary overhead.

```
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;

  int const chunk_size = 300;
  int num_chunks = (q + chunk_size-1) / chunk_size;

  #pragma omp parallel for
  for(int chunk = 0; chunk<num_chunks; chunk++) {

    int start_idx = chunk*chunk_size;
    int end_idx   = (start_idx+chunk_size > q) ? q : start_idx+chunk_size;

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

    calculate_distances(C, Q, start_idx, end_idx, D);

    for(int i=start_idx; i<end_idx; i++) {

      int query_idx = i - start_idx;

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

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

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

      free(indices);
    }

    free(D);
  }
}
```

Listing 4: 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, with minimal error.

The Johnson–Lindenstrauss Lemma asserts that, given a set of points in a high-dimensional space, it is possible to project them onto a lower-dimensional space such, that the Euclidean distances between the points are approximately preserved. Specifically, for a set of n points in d-dimensional space, we can project them onto a subspace of t-dimensions where

$$t = O\left(\frac{\log n}{\epsilon^2}\right),$$

and the distances between the points will be distorted by no more than a factor of $1 \pm \epsilon$, with high probability. This dimensionality reduction is crucial for large datasets, as it allows us to reduce the size of the problem without a significant loss in the accuracy of distance calculations.

```
void random_projection(Mat* C, Mat* Q, int t, Mat* C_RP, Mat* Q_RP) {

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

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

  C_RP->rows = c;
  C_RP->cols = t;
  Q_RP->rows = q;
  Q_RP->cols = t;

  C_RP->data = (double*)malloc(c*t*sizeof(double));
  Q_RP->data = (double*)malloc(q*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, c, t, d, 1.0, C->data, d, R, t,
      0.0, C_RP->data, t);
  cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, q, t, d, 1.0, Q->data, d, R, t,
      0.0, Q_RP->data, t);

  free(R);
}
```

Listing 5: Routine for applying Random Projection

When $c \gg d$, we cannot reduce the problem with random projection, as the target dimension t would exceed the original dimensionality d. Instead, we truncate the matrix C, retaining a set of representative rows, using k-means++ initialization and k-means clustering.

k-means clustering: Clustering is a technique that aims to group a set of data points into clusters, such that points within a cluster are similar. The similarity is typically measured using a distance metric, in this case the Euclidean distance. The idea is to partition the dataset into clusters in a way that maximizes the cohesion within clusters and the separation between different clusters.

1. **Initialization**: Randomly select k points from the dataset to serve as initial cluster centroids.
2. **Assignment step**: Each data point is assigned to the nearest centroid based on Euclidean distance, thus forming k clusters.
3. **Update step**: The centroids are updated by calculating the mean of all points in each cluster. This new mean becomes the new centroid for that cluster.
4. **Repeat**: The assignment and update steps are repeated until convergence, which occurs when the centroids no longer change significantly.

k-means++ initialization: The main challenge in k-means is selecting good initial centroids, as poor initialization can lead to suboptimal clustering results. To address this, k-means++ was introduced as an enhancement to k-means to select better initial centroids and improve the algorithm's convergence rate.

1. **First centroid selection**: The first centroid is chosen randomly from the dataset.
2. **Subsequent centroids selection**: For each remaining centroid, the probability of selecting a data point is proportional to its squared distance from the nearest already selected centroid. This ensures that the selected centroids are spread out across the data, capturing the overall structure of the dataset.
3. **Repeat**: This process is repeated until k centroids have been selected.

```
void truncMat(Mat* C, int r, Mat* C_TR) {

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

  C_TR->rows = r;
  C_TR->cols = d;
  C_TR->data = (double*)malloc(r*d*sizeof(double));

  srand(time(NULL));

  int first_idx = rand()%c;
  memcpy(C_TR->data, C->data + first_idx*d, d*sizeof(double));

  double* distances = (double*)malloc(c*sizeof(double));
  #pragma omp parallel for
  for(int i=0; i<c; i++) {
    double dist = 0.0;
    for(int j=0; j<d; j++) {
      double diff = C->data[i*d + j] - C_TR->data[j];
      dist += diff*diff;
    }
    distances[i] = dist;
  }

  for(int i=1; i<r; i++) {
    double total_dist = 0.0;

    #pragma omp parallel for reduction(+:total_dist)
    for(int j=0; j<c; j++) {
      total_dist += distances[j];
    }

    double rand_dist = ((double)rand()/RAND_MAX) * total_dist;
    double cumulative_dist = 0.0;
    int    next_idx = 0;

    for(int j=0; j<c; j++) {
      cumulative_dist += distances[j];
      if(cumulative_dist >= rand_dist) {
        next_idx = j;
        break;
      }
    }

    memcpy(C_TR->data + i*d, C->data + next_idx*d, d*sizeof(double));

    #pragma omp parallel for
    for(int j=0; j<c; j++) {
      double dist = 0.0;
      for(int k=0; k<d; k++) {
        double diff = C->data[j*d + k] - C_TR->data[i*d + k];
        dist += diff*diff;
      }
      if(dist < distances[j]) {
        distances[j] = dist;
      }
    }
  }

  free(distances);
}
```

Listing 6: Matrix Truncation

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. Moreover, the number of representative rows selected when truncating the matrix C affects the accuracy of our calculations. A heuristic metric that balances both speed and accuracy is given by $r = 100 \cdot \log(c) + 10d$.

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

```
double const e = 0.3;
int    const t = log((double)c) / (e*e);
if(t<d && (c>1000 && d>50)) {

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

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

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

  Mat C_TR;
  int const r = (int)(100*log(c)) + 10*d;
  printf("Representative rows (r): %d\n", r);
  truncMat(&C, r, &C_TR);

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

  free(C_TR.data);
} else {

  printf("Exact calculation\n");

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

Listing 7: Approximation Check

3.3 Details

To improve computational efficiency, we avoid calculating the square root of distances in the ‘calculate-distances‘ function. Instead, squared distances are compared directly within the ‘quickselect‘ function. This is mathematically valid because, for any $a > 0$ and $b > 0$, the inequality $a > b$ is equivalent to $a^2 > b^2$. The square root is computed only for the final k-nearest neighbors. This approach reduces the number of square root calculations from $c \times q$ (one for each distance) to $k \times q$, yielding a significant performance improvement.

While this method enhances runtime efficiency, it requires careful handling of numerical limits. The IEEE 754 double-precision format can reliably represent numbers up to approximately 10^{308}. However, squaring large elements in matrices C and Q may cause intermediate values to approach or exceed this limit, potentially resulting in overflow. To mitigate this, we use the ‘long double‘ data type, which provides a wider range (up to approximately 10^{4932} in most implementations) to safely handle large squared norms.

4 Testing

To evaluate the performance and validity of our implementation, we employed three metrics: **Accuracy**, **Queries per Second**, and **Execution Time**. We also used Matlab’s kNN routine for further testing.

4.1 Accuracy

To assess the algorithm's accuracy, we selected a random sample of queries and calculated their exact k-nearest neighbors. The accuracy was determined by computing the deviation of approximate results from exact results using the formula:

$$\text{Error} = \frac{|d_{\text{approx}} - d_{\text{exact}}|}{|d_{\text{exact}}|}$$

The cumulative error for all samples was averaged, and the accuracy was computed as:

$$\text{Accuracy} = 1 - \text{Average Error}$$

We avoided recall-based metrics that rely on index comparisons because the minimization techniques (e.g., representative rows) modify the indexing, making recall unsuitable. This adjustment ensures accuracy is evaluated based solely on the deviation in distances.

```c
double recall(Mat* C, Mat* Q, Neighbor* N, int k) {

  double accuracy = 0.0;

  Mat Q_TEST;
  Q_TEST.rows = SAMPLE;
  Q_TEST.cols = Q->cols;
  Q_TEST.data = (double*)malloc(SAMPLE * Q->cols * sizeof(double));
  memory_check(Q_TEST.data);
  memcpy(Q_TEST.data, Q->data, SAMPLE * Q->cols * sizeof(double));

  Neighbor* N_TEST = (Neighbor*)malloc(SAMPLE * k * sizeof(Neighbor));
  memory_check(N_TEST);
  findKNN(C, &Q_TEST, N_TEST, k);

  for(int i=0; i<SAMPLE; i++) {
    qsort(N + i*k, k, sizeof(Neighbor), compare);
    qsort(N_TEST + i*k, k, sizeof(Neighbor), compare);
  }

  double error = 0.0;
  for(int i=0; i<SAMPLE; i++) {
    for(int j=0; j<k; j++) {
      double approx = N[i*k + j].distance;
      double exact  = N_TEST[i*k + j].distance;

      if(exact != 0) {
        error += fabs(approx-exact) / fabs(exact);
      }
    }
  }

  double average = error / (SAMPLE*k);
  accuracy = 1.0 - average;

  free(Q_TEST.data);
  free(N_TEST);

  return accuracy;
}
```

Listing 8: Routine for Calculating Accuracy

4.2 Queries per Second

This metric measures the algorithm's throughput by dividing the total number of queries processed by the elapsed time. It reflects how many queries can be resolved per second.

```
double qps(struct timespec start, size_t q) {

  double elapsed = duration(start);
  return q/elapsed;
}
```

Listing 9: Routine for Calculating Queries per Second

4.3 Execution Time

Execution time is the total time taken by the algorithm to process the dataset, from the start of computation to the end of processing. It is a straightforward yet critical metric for benchmarking performance.

```
double duration(struct timespec start) {

  struct timespec end;
  clock_gettime(CLOCK_MONOTONIC, &end);

  double elapsed = (end.tv_sec-start.tv_sec) + (end.tv_nsec-start.tv_nsec)/1e9;
  return elapsed;
}
```

Listing 10: Routine for Calculating Elapsed Time

5 Summary

This project implements an efficient k-NN algorithm by combining advanced matrix operations, sorting techniques, and approximation methods. By sacrificing a small degree of accuracy, the algorithm achieves substantial speedups, especially in high-dimensional datasets. Additionally, parallelism is leveraged to further improve scalability. This balanced approach demonstrates the practicality of neighbor searches in large-scale and high-dimensional scenarios, ensuring both computational efficiency and flexibility.

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, as well as Matlab.
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

C Data Types:

- https://en.wikipedia.org/wiki/Long_double
- https://www.tutorialspoint.com/cprogramming/c_data_types.htm

Random Projection:

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

k-means Clustering:

- https://en.wikipedia.org/wiki/K-means_clustering

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