KNN Report

Tong Liu

2024-12-17

1. KNN Model Overview

The **K-Nearest Neighbors (KNN)** algorithm is primarily used for classification and regression tasks. Its core idea can be summarized as follows: for a sample to be predicted, the distances between the sample and all training samples are calculated. The k closest samples are then selected, and the final prediction is determined either by majority voting or by averaging. While the KNN algorithm is simple and intuitive, it suffers from high computational complexity. This limitation becomes particularly pronounced when dealing with large-scale datasets, where computational bottlenecks are a significant concern.

For the large-scale dataset in this project, the traditional KNN method faces the following challenges:

1. **High Computational Complexity**: For each test sample, the Euclidean distance must be calculated with all training samples, resulting in a complexity of O(N×M×D) (number of test samples × number of training samples × feature dimension).
2. **High Memory and Time Overhead**: High-dimensional data increases the time required for distance calculations, while a large number of samples imposes additional memory and sorting burdens.

2. Optimization Methods

To improve the performance of the KNN model for large-scale datasets, I adopted the following optimization methods:

1. **Feature Dimensionality Reduction**: Utilize **PCA** (Principal Component Analysis) for numerical variables and **MCA** (Multiple Correspondence Analysis) for binary categorical variables to reduce dimensionality. This minimizes the complexity of distance calculations and improves the computational efficiency of the KNN algorithm.
2. **Distance Calculation Optimization**: Since the Euclidean distance formula requires calculating the distance between each test sample and all training samples across every feature dimension, we expand its squared term. By precomputing and storing the sum of squares of the feature values for each training sample, we avoid redundant summations during each distance calculation, thereby improving efficiency.
3. **High-Performance Computing with C++**: Leverage **C++** for computationally intensive operations, while **R** is responsible for orchestration and management. The `nth_element` function is used to perform partial sorting instead of full sorting, efficiently identifying the top k nearest neighbors and reducing time complexity.
4. **Parallel Processing**: Utilize **parLapply** to process test data chunks in parallel, significantly reducing the overall runtime.

3. KNN Modeling Process

1. **Data Processing and Dimensionality Reduction**: Use functions such as **preProcess()** and **MCA()** to perform dimensionality reduction on continuous and binary data, respectively.
2. **Determining the Optimal K Value**: Randomly select 5000 samples from the training dataset. Using a step size of 20, calculate the accuracy of the KNN model for K values ranging from 50 to 500 with the **knn** function. The K value corresponding to the highest accuracy is selected as the optimal K value.
3. **KNN Model Prediction Using the Optimized Algorithm**

   a. Precompute the sum of squares of the training samples.

   b. **Implement the `knn_euclidean_optimized` Function in C++**: Write the function to handle core computations, including **distance calculation**, **nearest neighbor selection**, and **label voting**, to accelerate complex operations.

   c. **Apply the `knn_parallel_rcpp` Function**: Partition the test set into chunks of 1000 samples each. Use the **parLapply** function to compute predictions in parallel for each chunk, and finally integrate the prediction results.
4. **Results Evaluation and Visualization**:

   a. Calculate Accuracy and Confusion Matrix.

   b. Plot the ROC Curve and Calculate AUC.

   c. Use the `microbenchmark` Function to Compare the Execution Time Before and After Optimization, and Visualize the Results.

4. Results

By applying dimensionality reduction to the sample data, the original 20-dimensional features were reduced to 9 dimensions, effectively decreasing computational complexity and significantly improving the efficiency of distance calculations. Additionally, experiments determined the optimal number of neighbors, $\mathbf{K = 190}$, which was subsequently used for constructing and predicting with the KNN model.

The model evaluation results show that the optimized KNN model achieved an accuracy of **72%** on the test set.

Furthermore, compared to the manually implemented KNN distance calculation algorithm, the performance advantages are significant:

- The optimized algorithm achieved an **average runtime of 32 seconds**.
- The manually implemented algorithm had an **average runtime of 85 seconds**.

The optimized KNN algorithm achieved a significant improvement in runtime, with efficiency increased by approximately **62%**, demonstrating the effectiveness and practicality of the optimization strategies, including dimensionality reduction, efficient C++ implementation, and parallel scheduling.