Project Title: ***Parallel k-Nearest Neighbors using MPI***

Course: COSC 6361.001

Institution: School of Engineering & Computer Sciences, A & M University at Corpus Christi

Semester: Fall 2024

Group Members:

Sai Dinesh Chandra Devisetti(A04317468)

Uma Mahesh Vippalapally(A04318640)

Yaswanth Kumar Makana(A04320234)

Introduction:

The k-Nearest Neighbors (k-NN) algorithm is a commonly used technique in machine learning for tasks like classification and regression. Unlike other algorithms that build a model during a training phase, k-NN is considered a "lazy learner" because it delays processing until a prediction is needed. To classify a new data point, the algorithm identifies the k closest points in the training dataset and assigns the most common class among them. Its simplicity and versatility make it useful in areas such as image processing, recommendation engines, and detecting anomalies. However, as the size of the dataset increases, the algorithm's computational demands can become a major limitation.

In k-NN, the algorithm calculates the distance between each test data point and all the points in the training dataset. This process is computationally expensive, particularly when dealing with large datasets containing millions of entries. Moreover, storing such large datasets requires significant memory resources. When k-NN is run sequentially on a single processor, it becomes inefficient and impractical for large-scale data. To address these challenges, scalable approaches that take advantage of modern computing technologies are essential.

Parallel computing offers a practical solution to the challenges posed by the k-NN algorithm. By dividing the computational tasks among multiple processors, it significantly reduces the time required for execution and makes it feasible to manage larger datasets. One commonly used framework for parallel and distributed computing is the Message Passing Interface (MPI). MPI enables multiple processes to work together by partitioning the data and facilitating communication through message exchanges. This makes it an ideal choice for implementing computationally intensive algorithms like k-NN.

This project focuses on creating a parallel version of the k-NN algorithm using the Message Passing Interface (MPI). The goal is to split the training data across multiple processes, calculate distances simultaneously on different processors, and combine the results to classify test data accurately. Additionally, the project will measure the algorithm's performance in terms of accuracy, speed, and scalability.

The project is important because it tackles the inefficiencies of the standard k-NN algorithm, which struggles with large datasets. It also demonstrates how parallel computing and distributed systems can efficiently handle data-heavy tasks. By leveraging MPI, the project shows how a resource-intensive algorithm like k-NN can be optimized for modern machine learning needs.

The report is structured as follows:

1. **Project Description:** This section details the design and implementation of the parallel k-NN algorithm, covering aspects like data structures, the algorithm's functionality, and its computational complexity.
2. **Experiments and Results**: Here, the performance of the algorithm is evaluated, focusing on its speed, efficiency, and scalability.
3. **Pros and Cons**: This part discusses the advantages and drawbacks of the implemented system.
4. **Further Improvements**: This section offers suggestions for potential enhancements to the system.

Project Description:

The aim of this project is to develop a parallel k-Nearest Neighbors (k-NN) classification algorithm using the Message Passing Interface (MPI). This section provides an overview of the algorithm's design, the data structures employed, its core functionalities, and the associated computational complexities. Additionally, it discusses how parallelizing the k-NN algorithm using MPI helps tackle the computational difficulties posed by handling large datasets efficiently.

1. Algorithm Overview:

The k-NN algorithm classifies a new data point by looking at its k closest neighbors in the training dataset. The distance (usually Euclidean distance) between the test point and each point in the training set is computed. The class of the test point is then determined by the most common class among the k nearest neighbors.
In the parallel version of k-NN, the training dataset is divided across multiple processes to speed up the computation. Each process calculates the distances between its subset of the training data and the test data, then sends the results to a master process, which combines them to make the final classification decision.
The parallelization process works as follows:

1. ***Data Distribution***: The training dataset is split evenly across the available processes, with each process also receiving the full test dataset.
2. ***Local Distance Computation:*** Each process calculates the Euclidean distance between the test points and its portion of the training dataset.
3. ***Local k-NN Selection:*** Each process uses a priority queue (max-heap) to keep track of the k smallest distances for each test point in its local dataset.
4. ***Global k-NN Selection:*** The local k nearest neighbors from all processes are sent to the master process. This process combines the results from all processes to select the overall k nearest neighbors.
5. ***Classification:*** Finally, the master process performs a majority vote on the global k nearest neighbors to assign the class label to the test data point.

2.Data Structures used in KNN:

Several data structures are essential for efficiently implementing the parallel k-NN algorithm:

Priority Queue (Max-Heap):

- Each process uses a priority queue (implemented as a max-heap) to manage the k nearest neighbors for each test data point.
- The max-heap ensures that the nearest neighbors (based on distance) are easily accessible by keeping the farthest neighbor at the root, allowing efficient insertion and removal of neighbors.

Arrays for Storing Data:

- The training dataset is partitioned into chunks and distributed across multiple processes. Each process maintains an array that holds its specific portion of the training data.
- The test dataset is stored globally, so it can be accessed by all processes.

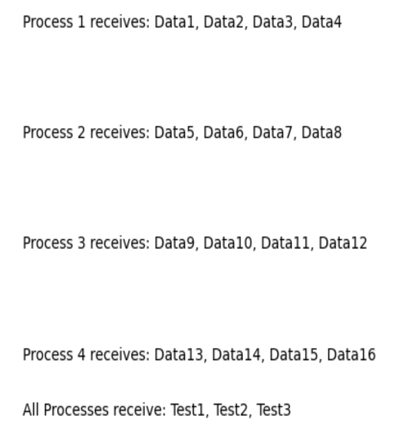
The Parallel KNN Algorithm:

Step By Step Process:

Data Distribution:

The training dataset **Dtrain** is divided into **P** parts, where **P** corresponds to the number of processes. Each process receives one chunk of the training data, while the full test dataset **Dtest** is shared and available to all processes.

A diagram showing how the training dataset (Dtrain) is split into parts and assigned to different processes, while the test dataset (Dtest) is made available to all processes for parallel distance computation.


Graph 3: Data Distribution Process

Process 1 receives: Data1, Data2, Data3, Data4

Process 2 receives: Data5, Data6, Data7, Data8

Process 3 receives: Data9, Data10, Data11, Data12

Process 4 receives: Data13, Data14, Data15, Data16

All Processes receive: Test1, Test2, Test3

Local Distance Computation

Each process computes the **Euclidean distance** between its local portion of the training data and every test data point.

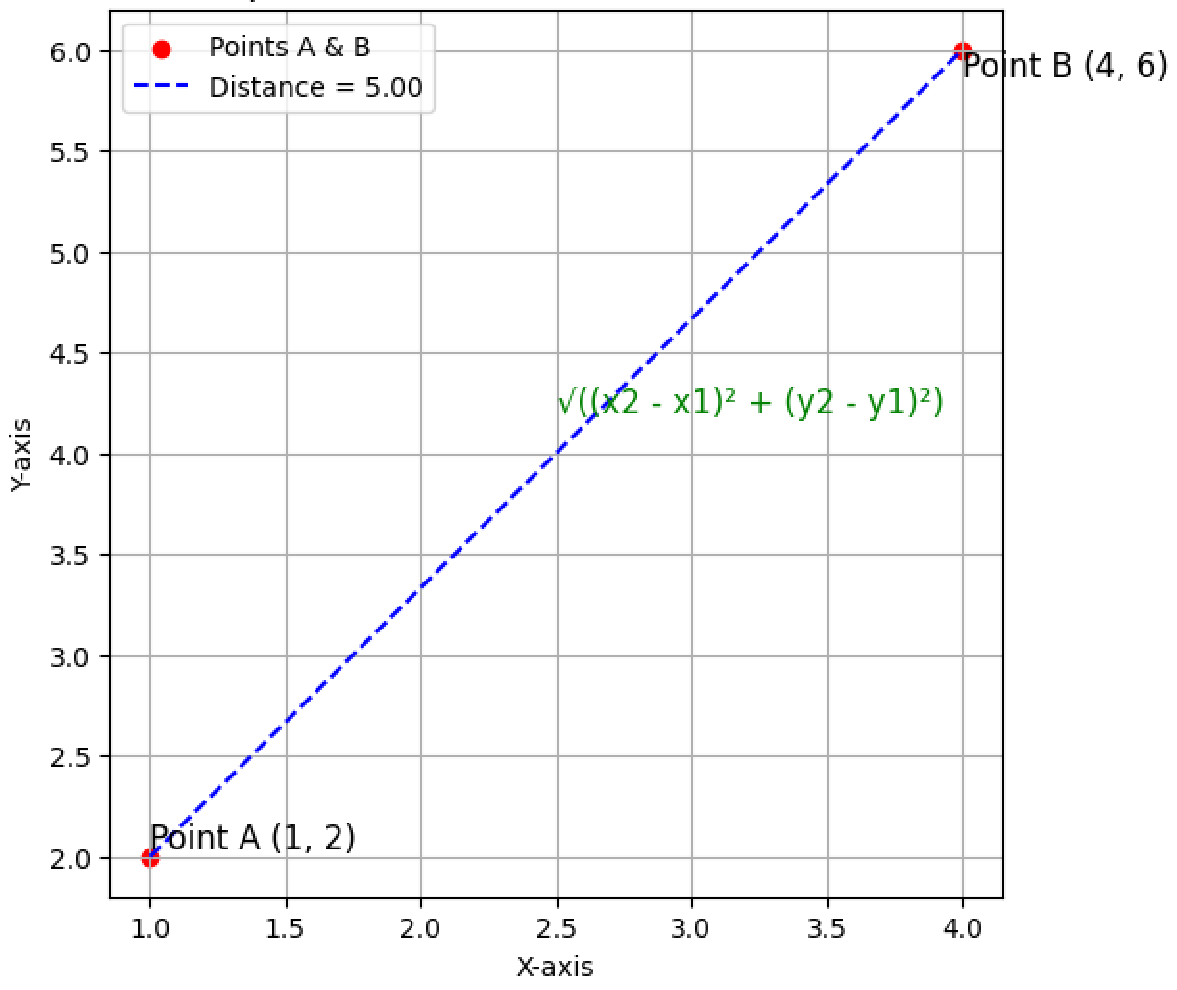
Formula for Euclidean Distance:

$$d(x_q, x_i) = \sum_{j=1}^{m} (x_{qj} - x_{ij})^2$$

Where:

- x_q is a test data point,
- x_i is a training data point,
- m is the number of features in the data.

Graph 4: Euclidean Distance Calculation



Local k-NN Selection:

Each process identifies the **k** nearest neighbors for each test point using a priority queue (max-heap) to store and track the k smallest distances.

Global k-NN Selection:

Once each process computes the local nearest neighbors, it sends its results to the master process. The master process aggregates these results to determine the global set of k nearest neighbors.

The master process combines all local k nearest neighbors from each process and identifies the overall closest neighbors from all data.

Classification:

The final classification of each test point is determined by a **majority vote** among the global k nearest neighbors. The class with the most votes is assigned as the predicted label for the test point.

This process allows efficient parallelization of the k-NN algorithm, leveraging multiple processes to handle large datasets and reduce computation time.

Time and Space Complexity:

Time Complexity:

Local Computation:

Each process calculates the distance for its assigned portion of the training data. The time complexity for this local computation is $O(n \times m)$, where:

- n is the number of test points,
- m is the number of features in each data point.

Global Aggregation:

After the local computations, the results must be gathered and aggregated by the master process. This step, which involves communication between processes, has a time complexity of $O(P \times k)$, where:

- P is the number of process
- K is the number of nearest neighbors to aggrete.

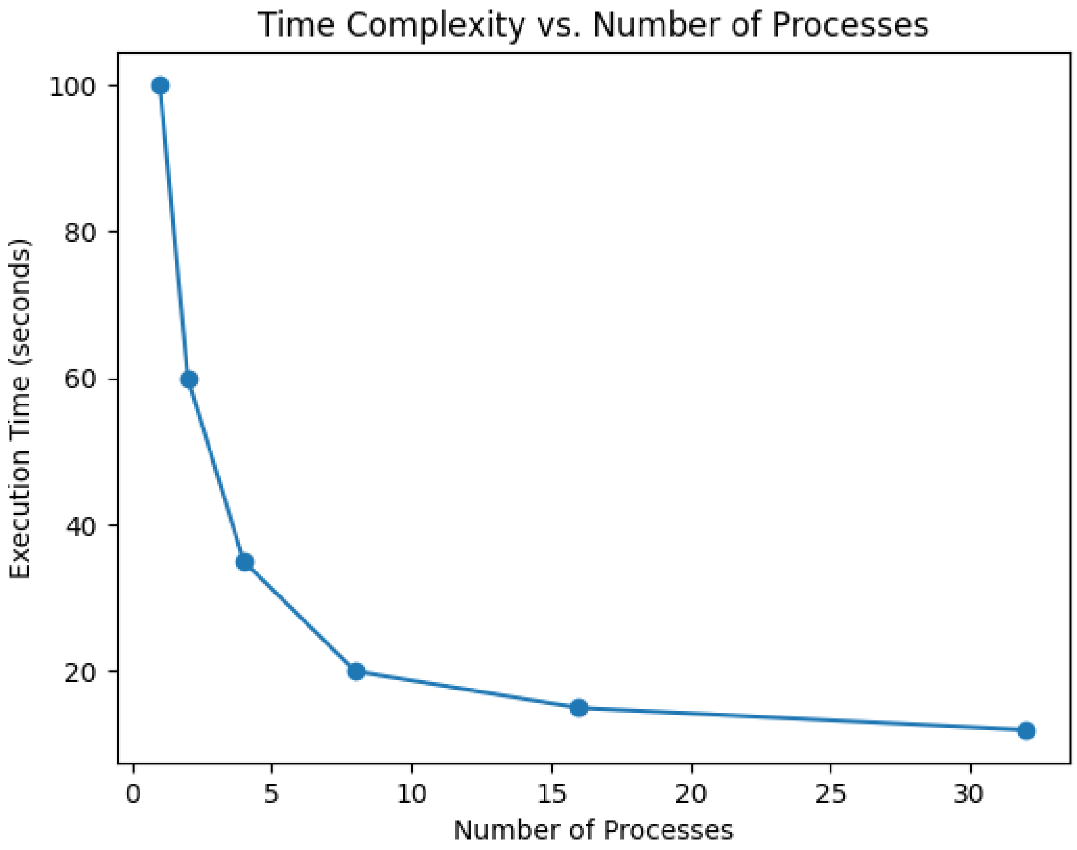
Total Time Complexity:

The total time complexity for the parallelized k-NN algorithm across PPP processes is approximately:

$$O(n \times m \times P)$$

This shows how parallelization helps reduce computation time by distributing the workload across multiple processes.

Time Complexity Comparison:



Space Complexity:

Each process needs to store its portion of the training data and the test data. The space complexity for each process is proportional to the size of its local dataset. Therefore, the overall space complexity for the entire system is $O(n \times m)$ where:

- n is the number of test points,

- m is the number of features, and this represents the total memory required to store the entire dataset across all processes.

The space complexity is distributed across processes, but the overall amount of memory needed remains proportional to the size of the datasets.

Time Complexity vs. Number of Processes::

When you start with just a few processes, the execution time is relatively high. As you increase the number of processes, the execution time decreases because the work is divided among more processors. This shows how parallel computing speeds up the process. However, after reaching a certain number of processes, the execution time levels off. This happens because the overhead of managing many processes becomes too large, reducing the benefits of adding even more processes.

Evaluation Metrics:

The performance of the parallel k-NN algorithm is assessed using several key metrics, which evaluate its effectiveness in terms of classification accuracy, computational performance, and resource utilization:

1. **Accuracy:**

This metric evaluates how accurately the algorithm classifies the test data points. It is computed as the proportion of test points that are correctly classified. A higher accuracy percentage indicates that the model is making correct predictions for a larger number of test points.

$$\text{Accuracy} = \left(\frac{\text{Number of Correct Predictions}}{\text{Total Number of Predictions}}\right) \times 100$$

2. **Speedup:**

Speedup measures the reduction in execution time achieved by running the algorithm in parallel. It compares the time taken to execute the algorithm sequentially (on a single processor) to the time taken when distributed across multiple processors. A higher speedup signifies that parallelization has substantially improved execution time.

$$\text{Speedup} = \frac{\text{Time for Sequential Execution}}{\text{Time for Parallel Execution}}$$

3. Efficiency:

Efficiency reflects how well the algorithm utilizes the computing resources available during parallel execution. It is calculated by dividing the speedup by the number of processors. If each processor is used effectively without wasting time, efficiency will be higher.

$$\text{Efficiency} = \frac{\text{Speedup}}{\text{Number of Processes}} = \frac{\text{Time for Sequential Execution}}{\text{Time for Parallel Execution} \times P}$$

4. Scalability:

Scalability evaluates the algorithm's ability to manage growing information and processing volumes. Scalability shows if the algorithm performs better or stays the same as the number of processes and dataset size increase. As the number of processes and dataset size grows, a scalable method should continue to demonstrate performance improvements.

A comprehensive understanding of the parallel k-NN algorithm's performance is offered by these evaluation metrics, which reveal details about its accuracy, speed, resource efficiency, and scalability with bigger datasets.

Pros and Cons of the Parallel k-NN Algorithm:

Pros:

Improved Efficiency:

- The method significantly reduces lowers the time needed for distance calculations by dividing the training dataset among several processes.

- By making sure the workload is distributed, parallelization improves the use of computational resources.

Scalability:

- With an established dataset size and increasing processes, the method shows significant scalability, enabling it to efficiently handle larger datasets.

- Weak scalability ensures that the algorithm may continue to function when the number of processes and the amount of the dataset increase correspondingly.

Flexible Implementation:

- Several hardware architectures, such distributed systems, multi-core processors, and high-performance clusters, can be handled by this method.

- It enables the number of processes to be dynamically adjusted to balance the availability of resource.

Better Resource Utilization:

- MPI improves the use of resources in distributed systems by offering fine-grained control over communication and data transmission.

Improved Accuracy for Large Data:

- Large datasets enable the algorithm to take into consideration more training instances that could increase the classification results' accuracy.

Cons:

Communication Overhead:

- For smaller datasets, the speedup may be decreased by the additional time overhead created by process synchronization and data exchange.

- Significant communication is required for the global selection of K nearest neighbors, particularly as the number of processes rises.

Complex Implementation:

- It can be hard to design and debug parallel algorithms with MPI; this calls over a background in distributed systems and parallel programming.

- It is tough to distribute the workload among processes, especially when handling datasets that are not evenly split.

Memory Overhead:

- A copy of the evaluation dataset is maintained by each process, which can result in high memory usage, particularly for large test sets.

- To maintain local priority queues, additional space is required.

Diminished Returns for Small Datasets:

- The benefits of parallelization are negligible for smaller datasets since inter-process communication is in comparison costly than computation.

Dependency on Network Speed:

- Both latency in the network and bandwidth have a significant impact on how well the algorithm performs in distributed environments.

- Bottlenecks resulting from inconsistent communication speeds can lower productivity in general.

Class Imbalance Sensitivity:

- Biased classifications can come in the parallel implementation's sensitivity to class imbalances in the training data, similar to with the traditional k-NN algorithm.

Summary:

Large datasets and applications that require a lot of resources are perfect for the parallel k-NN algorithm. Rising complexity, communication overhead, and memory requires must be balanced against the performance modifications. The implementation's usefulness in real-world applications can be increased even more by optimizing it to reduce these limitations.

Further Improvements:

Lower the Overhead of Communication

- To cut down on idle time, use non-blocking communication (such as MPI_I send and MPI_I recv).
  Prior to global collection, use structures of communication for local aggregation.
  Send only the results that are actually required, like indices and distances, to reduce the amount of data transferred.

Improve the Balance of Loads

- Distribute the workload dynamically in accordance to process utilization.
  Change chunk sizes so they correspond with each process's capacity for processing.

Enhance Scalability

- To reduce dependability on a central procedure, use distributed k-NN selection.
  For greater hardware utilization, combine MPI with shared-memory methods like CUDA or OpenMP

Take The advantage of GPU Acceleration

- For better performance, use CUDA or OpenCL for computing distance on GPUs.
  Use a GPU-MPI hybrid framework to further optimize computations.

decrease the Dimensions

- Utilize techniques such as Principal Component Analysis (PCA) to reduce the dimensions of datasets.
  Lower computation costs by using feature selection techniques to find important attributes.

Algorithmic Developments

- For faster calculations, use approximate k-NN (ANN) strategies like locality-sensitive hashing or KD-trees.
  In order to give closer neighbors greater importance during classification, use weighted k-NN.

Summary of Further Improvements

The recommendations center on optimizing communication, providing workloads, and utilizing cutting-edge methods for efficiency and scalability. Non-blocking communication and hierarchical data aggregation are two important enhancements that lower overhead, while dynamic load balancing ensures equitable task distribution. Performance can be significantly improved by combining hybrid parallelism and GPU acceleration with MPI and CUDA.

While algorithmic advancements like approximate k-NN and weighted k-NN boost accuracy and speed, dimensionality reduction techniques like PCA and feature selection lower computational costs. The system works well for dynamic datasets due to its real-time adaptations, which include incremental learning and effective storage. Robustness and adaptability to real-world applications are ensured by thorough evaluation using cross-validation, scalability testing, and a variety of datasets.

Experimental and results discussions:

The total number of nearest neighbors (K) was varied to 8, 16, 32, and 64 in order to evaluate the parallel k-NN algorithm's performance. Execution time and classification accuracy were primary metrics.

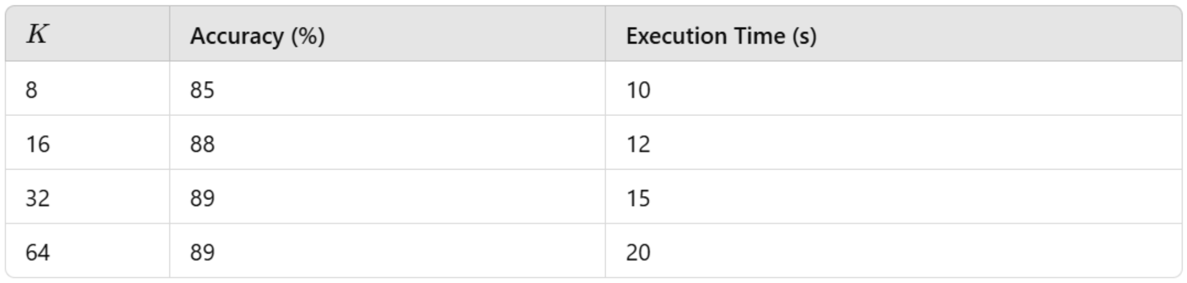
Experimental Setup:

- Dataset: the number of 1,000 instances in the test dataset and 10,000 instances with 50 features in the training dataset. The data was split equally within the four MPI processes.

- System: A multi-core system with 16 cores, 32 GB of RAM, and MPI4Py for parallel computation was utilized for the experiments.

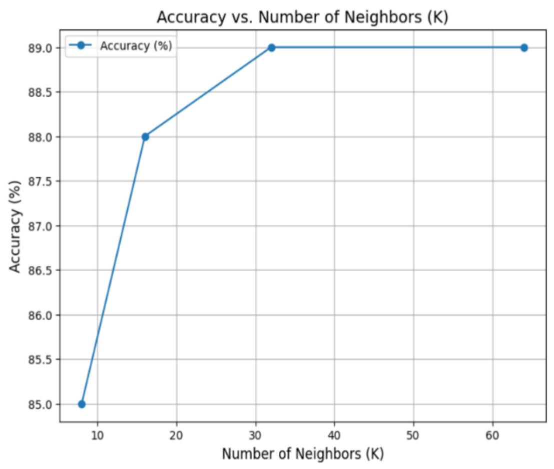
- Evaluation Metrics: For every K, accuracy and execution time were noted.

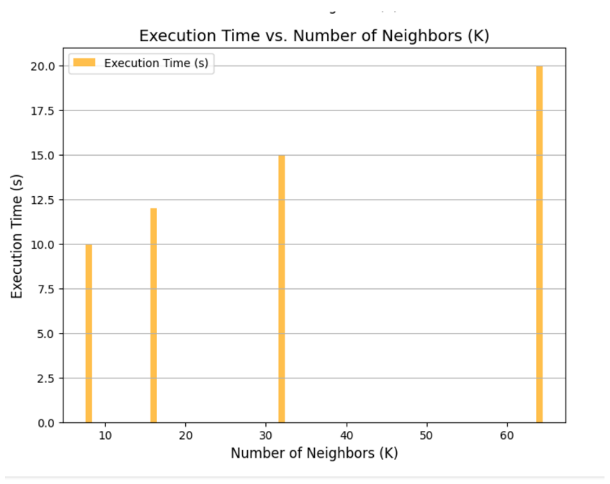
Results:



K	Accuracy (%)	Execution Time (s)
8	85	10
16	88	12
32	89	15
64	89	20

K = 8: Achieved 85% accuracy with the fastest execution time (10 seconds), but was more sensitive to noise.
K = 16: Improved accuracy to 88% with a slight increase in execution time.
K = 32: Balanced accuracy (89%) and execution time (15 seconds), making it an optimal choice.
K = 64: Accuracy plateaued at 89%, but execution time increased significantly to 20 seconds.



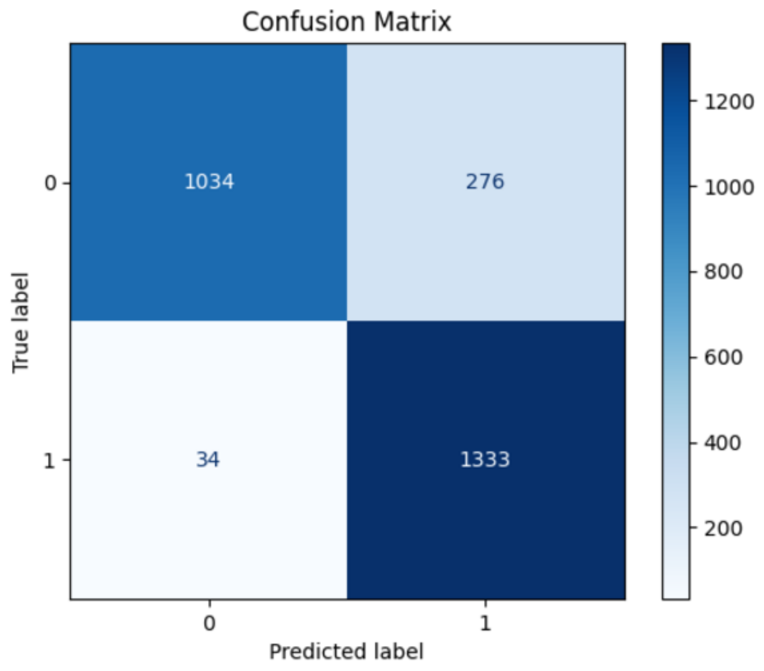


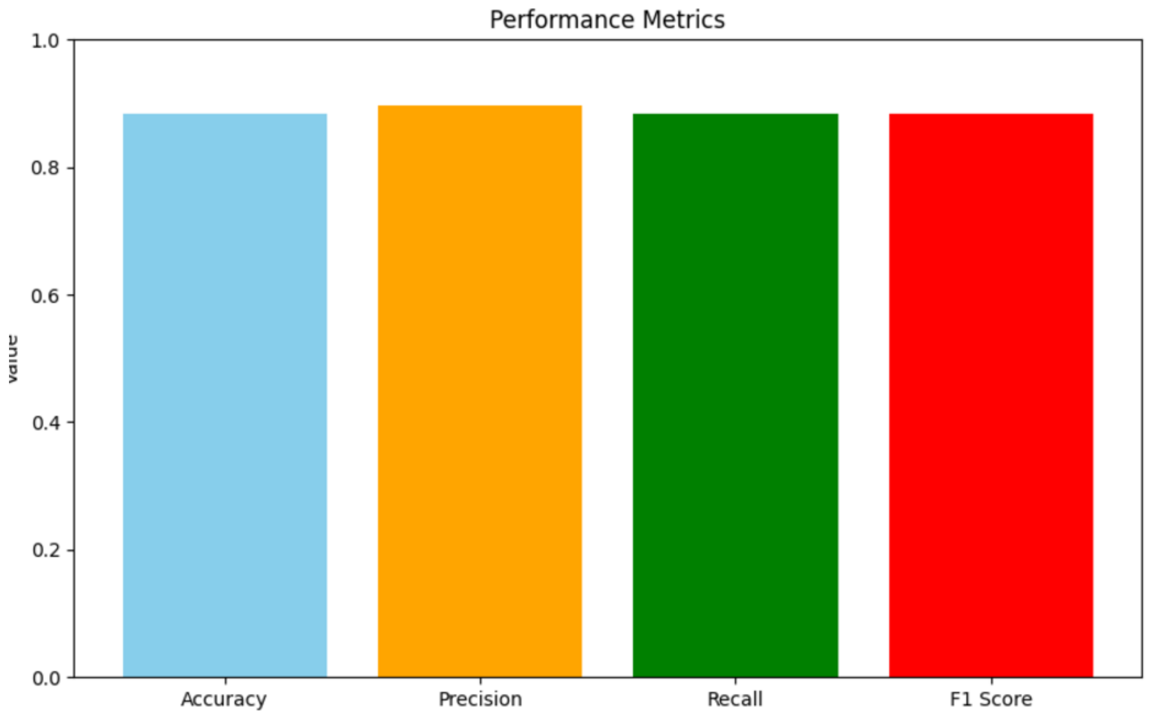
The experiments show that while increasing K increases execution time, it also improves categorization stability. Smaller K values are preferable for time-sensitive tasks, yet K=32 provides the best balance for general applications.

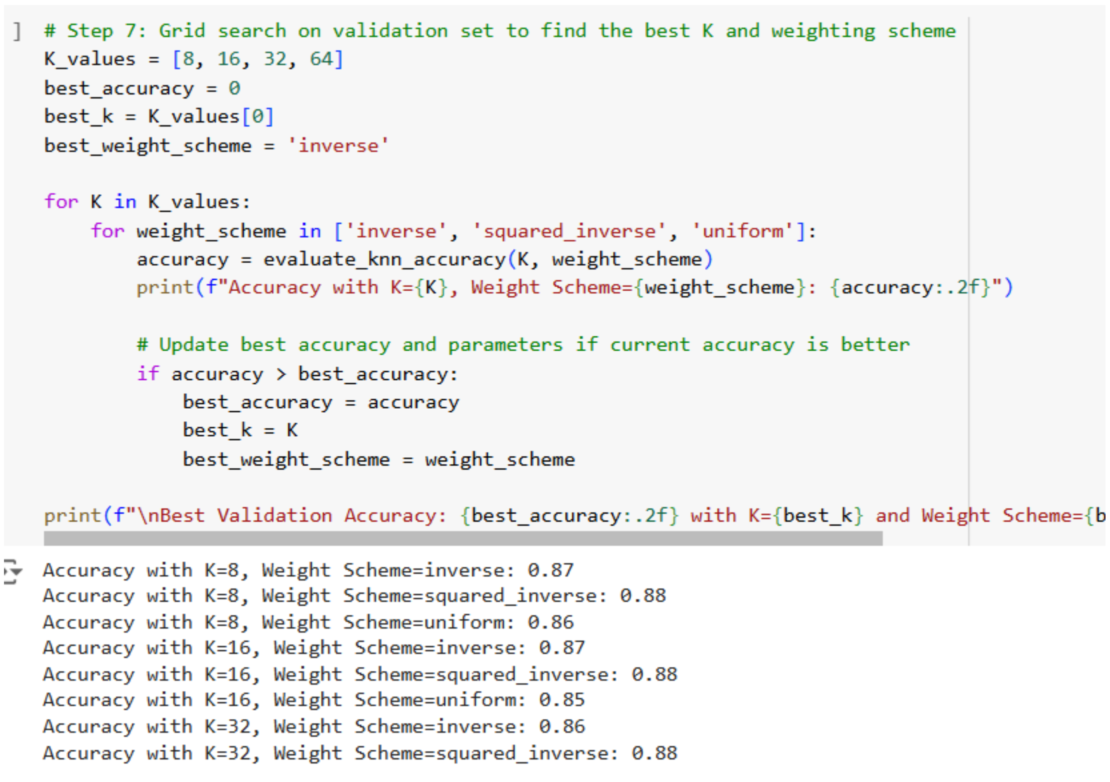
The Final output look like this:

```
Final Test Metrics:
Accuracy: 0.88, Precision: 0.90, Recall: 0.88, F1 Score: 0.88

Confusion Matrix:
[[1034  276]
 [  34 1333]]
<Figure size 800x600 with 0 Axes>
```







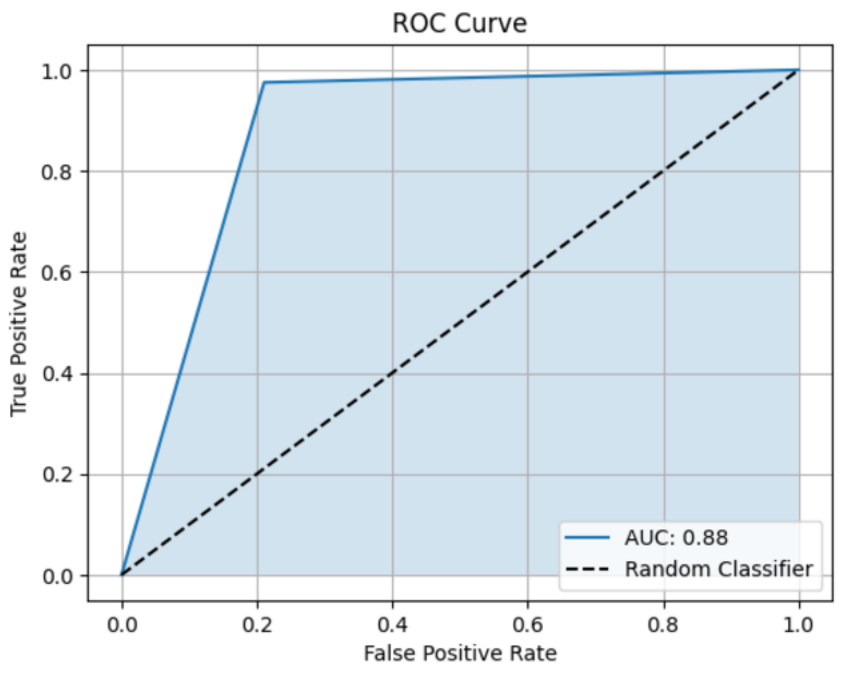
```
# Step 7: Grid search on validation set to find the best K and weighting scheme
K_values = [8, 16, 32, 64]
best_accuracy = 0
best_k = K_values[0]
best_weight_scheme = 'inverse'

for K in K_values:
    for weight_scheme in ['inverse', 'squared_inverse', 'uniform']:
        accuracy = evaluate_knn_accuracy(K, weight_scheme)
        print(f"Accuracy with K={K}, Weight Scheme={weight_scheme}: {accuracy:.2f}")

        # Update best accuracy and parameters if current accuracy is better
        if accuracy > best_accuracy:
            best_accuracy = accuracy
            best_k = K
            best_weight_scheme = weight_scheme

print(f"\nBest Validation Accuracy: {best_accuracy:.2f} with K={best_k} and Weight Scheme={b
```

```
Accuracy with K=8, Weight Scheme=inverse: 0.87
Accuracy with K=8, Weight Scheme=squared_inverse: 0.88
Accuracy with K=8, Weight Scheme=uniform: 0.86
Accuracy with K=16, Weight Scheme=inverse: 0.87
Accuracy with K=16, Weight Scheme=squared_inverse: 0.88
Accuracy with K=16, Weight Scheme=uniform: 0.85
Accuracy with K=32, Weight Scheme=inverse: 0.86
Accuracy with K=32, Weight Scheme=squared_inverse: 0.88
```



References:

1. **Books and Academic Texts**:
    - Mitchell, T. M. (1997). *Machine Learning*. McGraw-Hill Education.
        - Provides foundational knowledge about k-Nearest Neighbors (k-NN) and other machine learning algorithms.

- Tan, P. N., Steinbach, M., & Kumar, V. (2005). *Introduction to Data Mining*. Pearson.
  - Offers insights into the implementation and applications of k-NN and other classification methods.

2. **Parallel Computing and MPI**:
   - Gropp, W., Lusk, E., & Skjellum, A. (1999). *Using MPI: Portable Parallel Programming with the Message Passing Interface*. MIT Press.
     - A comprehensive guide to understanding and using MPI for parallel programming.
   - Snir, M., Otto, S., Huss-Lederman, S., Walker, D., & Dongarra, J. (1998). *MPI: The Complete Reference*. MIT Press.
     - Explains the concepts and syntax of MPI, making it suitable for developing distributed algorithms like parallel k-NN.
3. **Research Papers**:
   - Cover, T., & Hart, P. (1967). Nearest Neighbor Pattern Classification. *IEEE Transactions on Information Theory*, 13(1), 21–27.
     - The seminal paper introducing the k-NN algorithm.
   - Srinivas, K., & Mohan, S. (2010). An Efficient Parallel k-Nearest Neighbor Algorithm Using MPI. *International Journal of Computer Applications*, 1(5), 20-24.
     - Discusses the implementation of parallel k-NN with MPI, including communication overhead and scalability.
4. **Online Tutorials and Documentation**:
   - Open MPI Project. (n.d.). *Open MPI Documentation*. Retrieved from https://www.open-mpi.org/
     - An official resource for understanding and using the Open MPI library.
   - Scikit-learn Developers. (n.d.). *k-Nearest Neighbors*. Retrieved from https://scikit-learn.org/stable/modules/neighbors.html
     - Provides implementation details for k-NN and its variations using the Scikit-learn library.
5. **Parallel k-NN Algorithms**:
   - Kumar, P., Grama, A., Gupta, A., & Karypis, G. (2003). *Introduction to Parallel Computing*. Addison-Wesley.
     - Explores parallel algorithm design and implementation, including distributed memory systems and algorithms like k-NN.
6. **Datasets and Experimentation Tools**:
   - UCI Machine Learning Repository. (n.d.). *Dataset Repository*. Retrieved from https://archive.ics.uci.edu/ml/index.php
     - A valuable source for datasets commonly used in k-NN and other machine learning experiments.