UNIVERSITÀ
DI PAVIA

Open MPI implementation of K-Nearest Neighbors

F. Amato[1] and D. Ligari[1]

[1] *University of Pavia, Department of Computer Engineering (Data Science), Pavia, Italy*

Date: February 12, 2023

Abstract— The objective of this project is providing an efficient implementation of the K-Nearest Neighbors (Machine Learning) algorithm, that exploits multiple CPUs architectures, using Open MPI (on C++). Starting from a serial implementation, the code has been modified to work in parallel to obtain an improvement in terms of performances, and so a significant speedup (compared with the standard serial version). An a priori study of available parallelism has been conducted, to understand why and which parts of the serial code could have been parallelized. To better understand how this implementation scales, two dimensions have been varied: the datasets sizes, and processors number. In so doing, the code has been runned onto Google Cloud Platform virtual instances, to exploit the whole computational power of a cluster of machines cooperating one with each other.

Keywords— K-Nearest Neighbors • Machine Learning • Parallelization • Open MPI • Performances • Google Cloud Platform

1. What is K-Nearest Neighbors

Nowadays Machine Learning (ML), a branch of Artificial Intelligence (AI) that automates the construction of analytical models based on the idea that systems can learn from data without the need for human intervention, is becoming used in many different application fields, especially to build efficient models to help humans in taking decisions, understanding interesting patterns, and so on.

The problems that Machine Learning can efficiently solve belong in two different macro-groups: supervised learning (classification, regression) and unsupervised learning (clustering, dimensionality reduction). The former group uses labeled datasets to train algorithms to find which are the most impactful features for a given phenomenon outcome (categorical or continuous values). The latter, instead, uses unlabeled datasets to analyze and discover hidden patterns or data groupings.

The K-Nearest Neighbors algorithm (KNN) is falling into the supervised learning methods, and is based on proximity to make classifications or predictions of an individual data point. It works particularly well when the assumption that similar points can be found near one another is true.

The report threaten version can solve classification problems, but it can be easily extended (modifying properly it) to solve also regression.

Its basic working principle is to assign the class label to a given point on the basis of a majority vote, considering its K nearest neighbors. Those K points are obtained by performing a selected distance function (e.g: Euclidean, Manhattan, Minkowski or Hamming).

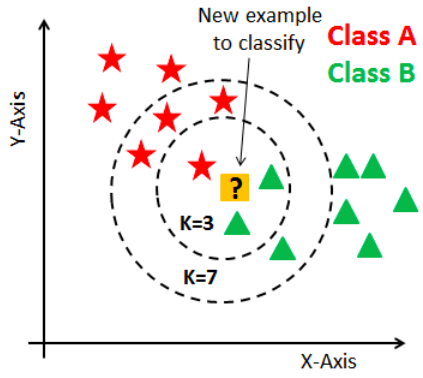

Fig. 1: Simple K-Nearest Neighbors problem

Obviously, each distance function provides different results. In this implementation, the Euclidean distance is the standard (and only) one.

In order to display its behavior, a simple example, where both the classes and feature number are just two, has been chosen [1]. A question that might come to your mind is:

- What is the class of the unclassified point?

Based on what stated previously, if the number of considered neighbors (K) is equal to three the most occurrent class is the green one, otherwise considering six it would have been red. The described working flow is the same considering a more complex example.

Even the choice of the value of the parameter K is a critical aspect, since it can lead to over-fitting (low K) or under-fitting (high K). It should be properly tuned, since it will largely depend on the input data, as data with more outliers or noise will likely perform better with higher values of K. Clearly it should have some sense considering very far (high K) points.

Contact data: F. Amato, francesco.amato01@universitadipavia.it

Parameter	Description
trainFile	Path of train file
nTrainSamples	Number of train points to use
testFile	Path of test file
nTestSamples	Number of test points to use
K	Number of neighbors
nFeatures	Number of features for each sample
nClasses	Number of classes

Table 1: Parameters of the serial version

2. Sample datasets generation

Even though the implementations that will be discussed in details in the following sections are capable of perform classification even on real case datasets (as will be shown on the well-known Iris dataset), the objectives of this analysis are more biased on execution performances with respect to Machine Learning in a strict sense (the objective is not to find the best configuration for a KNN classifier in a specific field of study).

So, even a general purpose dataset (with not a real meaning of the features representing a specific characteristic for that sample of a given class) can be considered as good as benchmark to test how the code behaves in terms of performances by considering different aspects such as: the number of samples (dataset dimension), the number of classes, the number of features for each class (still related to the dataset dimension), and whether to apply Min-Max scaling.

In order to cover this aspect, a specific Python script has been implemented, and it can be found at the following GitHub link:

- KNN-OpenMPI/Datasets/generate.py

3. Serial implementation

In the section, it will be discussed how the serial version of the KNN algorithm has been implemented from scratch, using C++.

The code can be entirely analyzed at the following link:

- KNN-OpenMPI/Serial

The code can be described as a set of consequent steps, in order to perform classification on a general dataset. The final objective is, given the parameters described in the above table (a more detailed description can be found in the project's README file), to obtain train and test accuracy of a KNN model.

More precise classification performance metrics should have been used (e.g., to understand whether the selected model is more biased on recognizing a class with respect to another), but this is not the actual purpose of the project.

In order to obtain the test accuracy (the same line of reasoning follows for train accuracy), the following steps are performed one after the other:

- Consider each test sample and compute the Euclidean distance (1) with respect to all the train samples
- Consider the most occurrent class of K-Nearest Neighbors points of each test sample and assign it to the predicted test sample class
- Compare the actual class of the test sample with the predicted one, if the values are equal the sample has been correctly classified (classification accuracy consequently increases), otherwise it is interpreted as a misclassified point (accuracy decreases)
- Divide the number of correctly classified test points by the total number of test points, multiply for 100 and this is the test accuracy

N-dimensional Euclidean distance formula The used distance formula is the depicted below.

$$d(p,q) = \sqrt{\sum_{i=1}^{n} (q_i - p_i)^2} \tag{1}$$

Where p and q are respectively a train and test sample, whereas n is the number of features that the two considered samples have.

The discussed serial algorithm has been applied also on the already cited Iris dataset (considering a train-test split of 0.8/0.2 and fine-tuning the K parameter - using a random search approach), in order to show that it can work properly also with datasets used daily by the data scientist community. For this simple classification problem, both train and test accuracies are almost 97% (neither over-fitting nor under-fitting, since the discrepancy between the two is negligible).

Example on how the script can be executed, once compiled, on the Iris dataset:

```
./knn-serial ../Datasets/train.txt
120 ../Datasets/test.txt 30 10 4 3
```

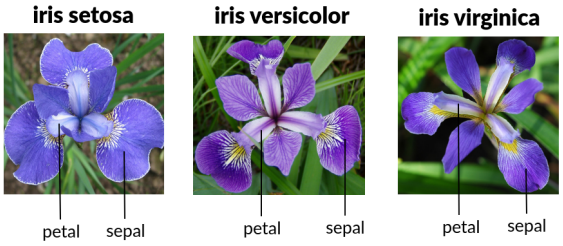

Fig. 2: Visual representation of the different iris classes, considering the two more distinguishable features between all those iris types, sepal and petal (PCA)

Cost associated with each task of the serial code, in terms of number of instructions For what concerns the serial implementation an a priori estimate on the number of instructions needed to conclude a specific tasks has been conducted, in order to better understand which were the functions that needed more (if possible) parallelization, since they were more computationally expensive.

The results of that analysis are shown in the following table, which has been obtained by looking directly at the code of the serial version (they can also be checked more precisely by using a specific profiler tool such as Valgrind or Gprof).

Task	Cost (# of instructions)
Euclidean distance	$NFeatures+2$
Sorting	$nTrainSamples*log(nTrainSamples)$
KNN prediction	$nTrainSamples*EuclDist+Sort+3*nClasses+K$
Array calculus	$nTestSamples*KNNPrediction+2$
Read train samples	$1+nTrainSamples*(2+nFeatures)$
Read test samples	$1+nTestSamples*(2+nFeatures)$
Initialization	18

Table 2: Estimate of the number of instruction executed, for each function of the serial implemented version

4. AVAILABLE PARALLELISM

In this section, it will be briefly discussed the expected improvement in terms of performances of parallelizing the serial solution described previously.

The serial version code has been analyzed discarding the less relevant (neither critical nor crucial) parts, such as: the checks on the parameters that can be passed to the `main()` function; the import of the libraries, and the both global variables and the sample (`struct`) definition. The only fraction of code that cannot be efficiently, nor simply, be parallelized is the reading of the two datasets (train and test). All the other functions can exploit multiple CPUs architectures (how they have been parallelized is explained in the next section).

Amdalh's law In order to understand the plausible speedup which can be obtained parallelizing the serial version, the Amhdal's law has been used. Its formula is the following one:

$$Speedup(N) = \frac{1}{S+\frac{P}{N}} \tag{2}$$

Where N is the number of CPUs used, S is the fraction of the code that cannot be parallelized (serial execution), while P the fraction that can be parallelized (recall that $P+S=1$).

By looking at the previous above table it can be noticed that the number of iterations for each task, among other things, is strongly correlated with the dataset dimension (the number of samples to be considered both in train and test).

Since the speedup depends on the portion of serializable (and non) fraction of code and those two depend on the number of samples it can be noticed that the speedup does not change only with the number of cores (which can equally spread the work), but also on the number of samples.

The serializable code fraction diminishes with the increasing of the dataset dimension (tends to zero, as shown on the next graph), so consequently the speedup increases linearly with the core number.

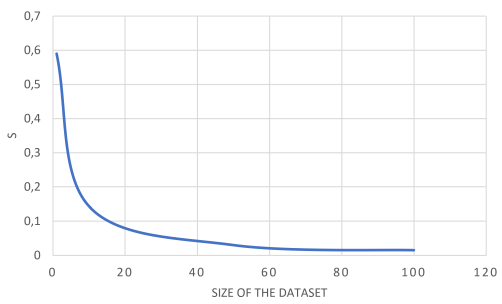

Fig. 3: Theoretical fraction of code that cannot be parallelized, in relation to the size of the dataset

In reality, differently from what discussed from the "theoretical" view point, what is expect is that the dataset dimension impacts negatively (to be valued how much) the execution time, since the data transmission between master and slaves depends (grows proportionally) on the amount of train and test of samples.

5. OPEN-MPI IMPLEMENTATION

6. PERFORMANCE ANALYSIS

7. CONCLUSIONS