**Serial and parallel implementation’s performance evaluation of**

**K-Nearest Neighbors algorithm.**

**Content**

1. **Goal**
2. **Background**
   1. **Machine learning**
   2. **Supervised Learning**
   3. **Classification**
   4. **K-nearest neighbors (KNN) classiﬁer**
3. **Methodology**
   1. **Sequential algorithm**
   2. **Parallel Algorithm**
4. **Results**
5. **Data Distribution & Load Balancing**
6. **Conclusion**
7. **Future Work**
8. **References**
9. **Codes:**
   1. **Serial code**
   2. **Parallel1 & Parallel2 code**

**Goal**

The k-Nearest Neighbors (KNN) is one of the statistical supervised machine learning algorithm used for classification. It classifies a given observation to a class to which the majority of given observation's "k" closest neighbors belongs to. It has been observed that the algorithm becomes computationally extensive and made the classification task slow when the data size becomes large. Many attempts have been made to parallelize it on GPUs by using their natural parallel architecture. I tried to increase the performance of it by parallelizing it using FORTRAN, MPI & open MP on HPC cluster.

**Background**

**Machine Learning:**

It is a subfield of computer science that evolved from the study of pattern recognition and computational learning theory in artificial intelligence.

**Supervised learning:**

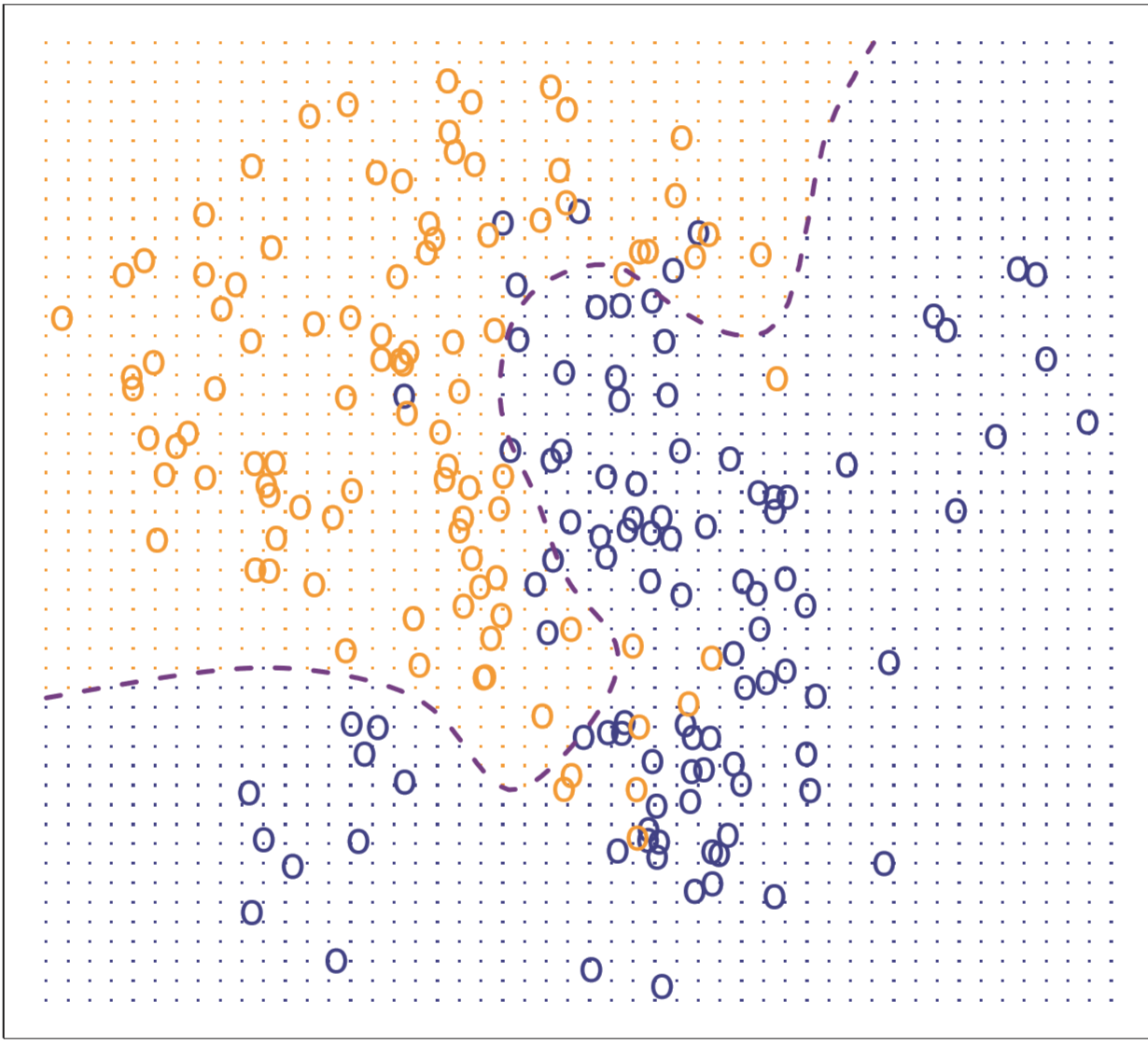
Supervised statistical learning involves building a statistical model for predicting, or estimating, an output based on one or more inputs. Problems of this nature occur in ﬁelds as diverse as business, medicine, astrophysics, and public policy.

In other words, For each observation of the predictor measurement(s) xi, i =1,...,n there is an associated response measurement yi. We wish to ﬁt a model that relates the response to the predictors, with the aim of accurately predicting the response for future observations (prediction) or better understanding the relationship between the response and the predictors (inference).

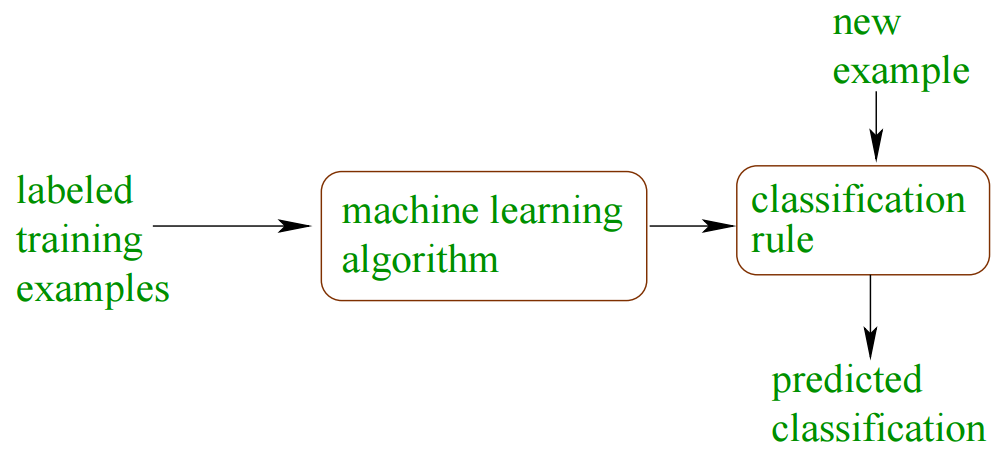
**Classifications:**

Predicting a qualitative response (non-numerical value—that is, a categorical or qualitative output), Yi for an observation can be referred to as classifying the observation, since it involves assigning the observation to a category, or class. On the other hand, often the methods used for classiﬁcation ﬁrst predict the probability of each of the categories of a qualitative variable, as the basis for making the classiﬁcation.

E.g. On the basis of DNA sequence data for a number of patients with and without a given disease, a biologist would like to ﬁgure out which DNA mutations are deleterious (disease-causing) and which are not.



A simulated data set consisting of 100 observations in each of two groups, indicated in blue and in orange. The orange background grid indicates the region in which a test observation will be assigned to the orange class, and the blue background grid indicates the region in which a test observation will be assigned to the blue class.



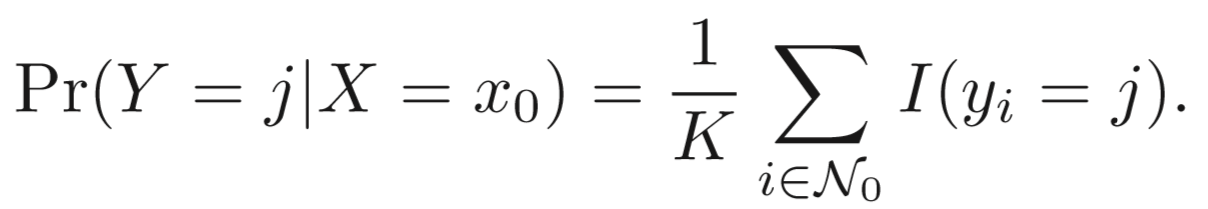
Applications of classification:

1. text categorization (e.g., spam filtering)
2. fraud detection
3. optical character recognition]
4. machine vision (e.g., face detection)
5. natural-language processing (e.g., spoken language understanding)
6. market segmentation (e.g.: predict if customer will respond to promotion)

**K-nearest neighbors (KNN) classiﬁer:**

In real data, we do not know the conditional distribution of Y given X. K-nearest neighbors (KNN) classiﬁer is one among various approaches attempt to estimate the conditional distribution of Y given X, and then classify a given observation to the class with highest estimated probability.

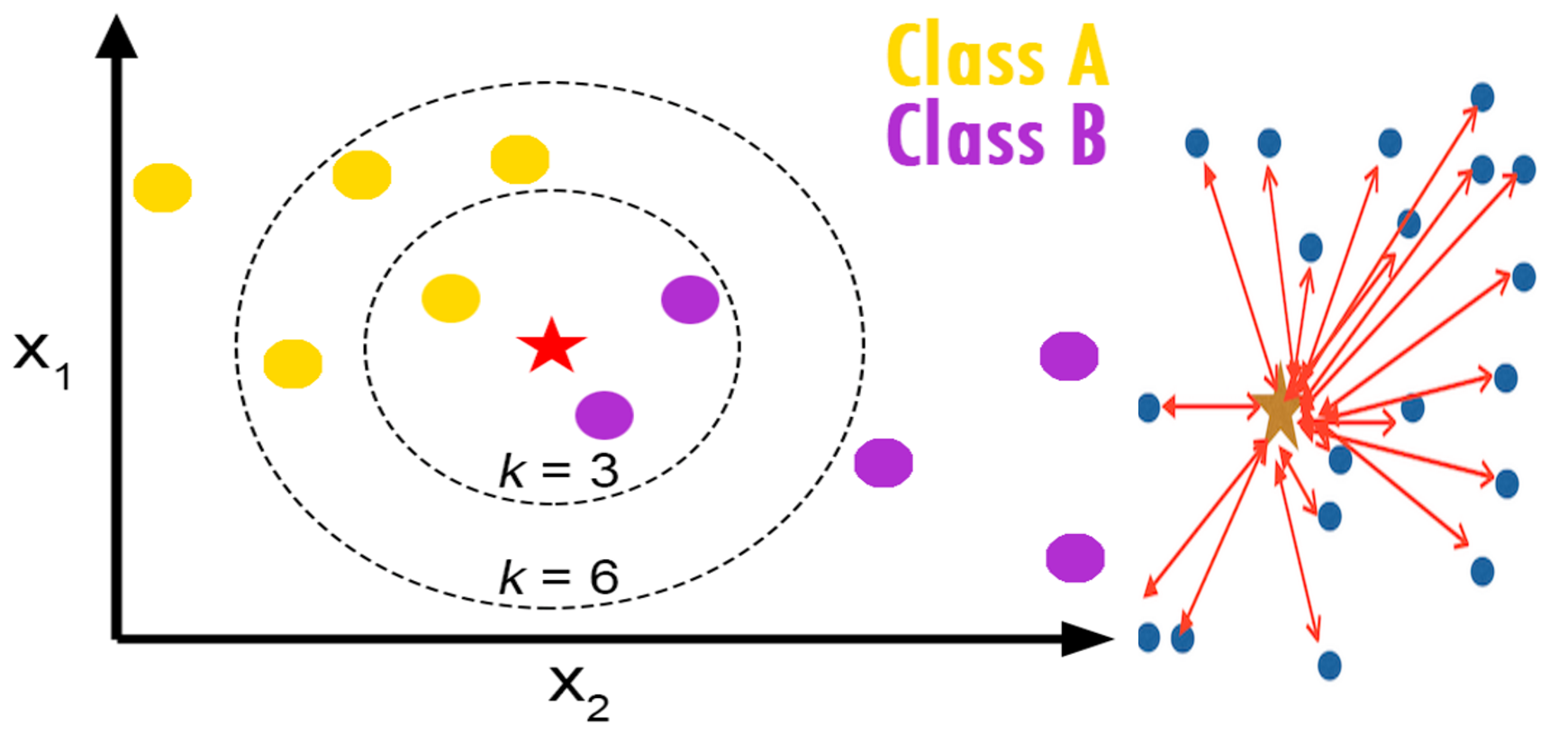
Given a positive integer K and a test observation x0, the KNN classiﬁer ﬁrst identiﬁes the K points in the training data that are closest to x0, represented by N0. It then estimates the conditional probability for class j as the fraction of points in N0 whose response values equal j:



Finally, KNN applies Bayes rule and classiﬁes the test observation x0 to the class with the largest probability. According to Bayes Rule, a test observation with predictor vector x0 is assigned to the class j for which Pr(Y=j |X=X0) is largest.

In practice, k is usually chosen to be odd, so as to avoid ties. The k = 1 rule is generally called the nearest-neighbor classification rule.

Our goal is to make a prediction for the point labeled by the “STAR” {test observation}. Suppose that we choose K = 3. Then KNN will ﬁrst identify the three observations that are closest to the star. This neighborhood is shown as a circle. It consists of two purple points and one yellow point, resulting in estimated probabilities of 2/3 for the purple class and 1/3 for the yellow class. Hence KNN will predict that the star belongs to the purple class I.e. class B.



K-nearest neighbor (KNN) is a widely used classification technique and has significant applications in various domains, especially in text classification. The computational-intensive nature of KNN requires a high performance implementation.

**Picking an optimal K:**

The choice of K has a drastic eﬀect on the KNN classiﬁer obtained

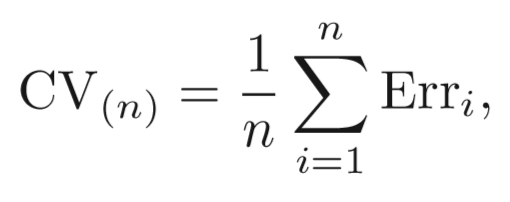
In general, we do not really care how well the method works training MSE on the training data. Rather, we are interested in the accuracy of the predictions that we obtain when we apply our method to previously unseen test data.

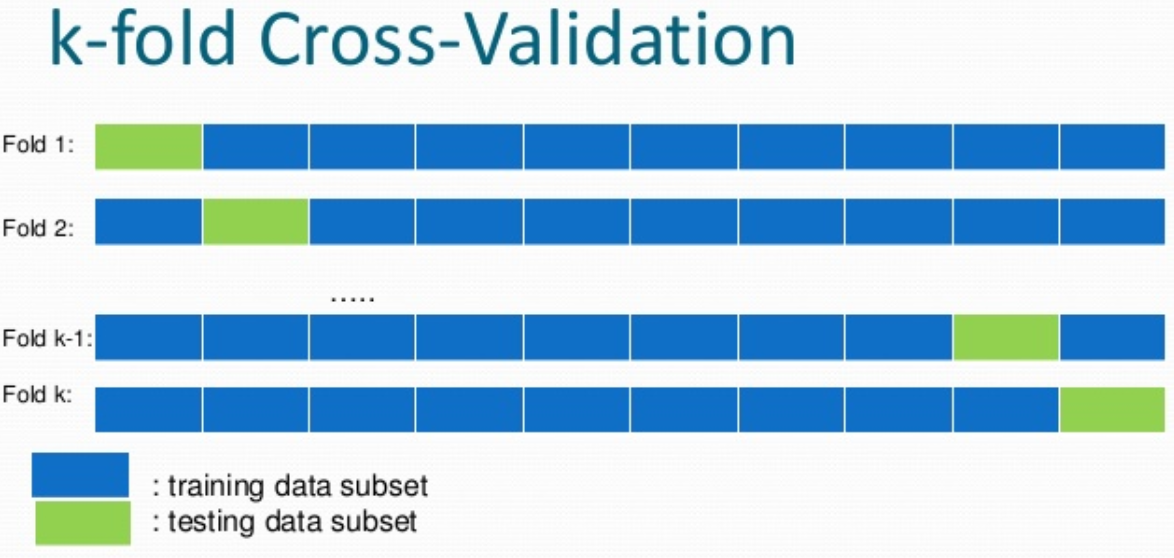
In other words, suppose that we have clinical measurements (e.g. weight, blood pressure, height, age, family history of disease) for a number of patients, as well as information about whether each patient has diabetes. We can use these patients to train a statistical learning method to predict risk of diabetes based on clinical measurements. In practice, we want this method to accurately predict diabetes risk for future patients based on their clinical measurements.

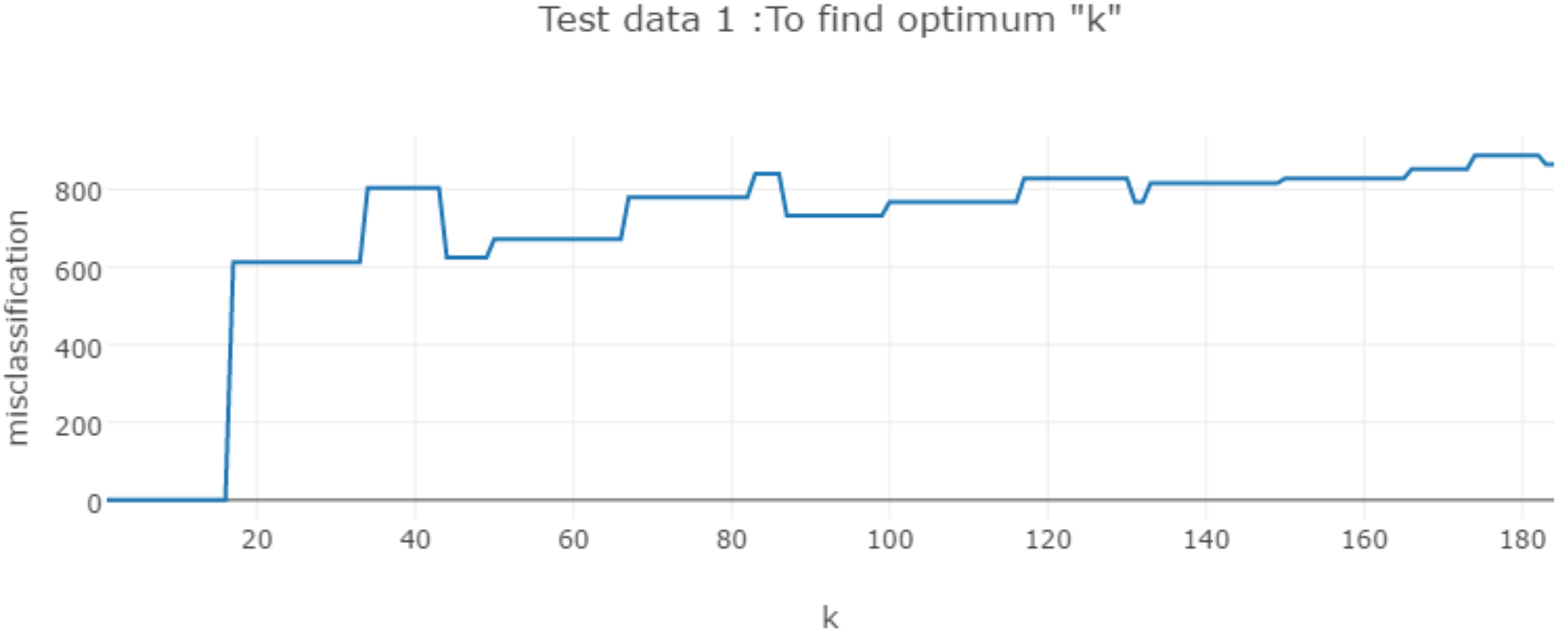
One such approach to pick an optimal “K” is Cross-validation:

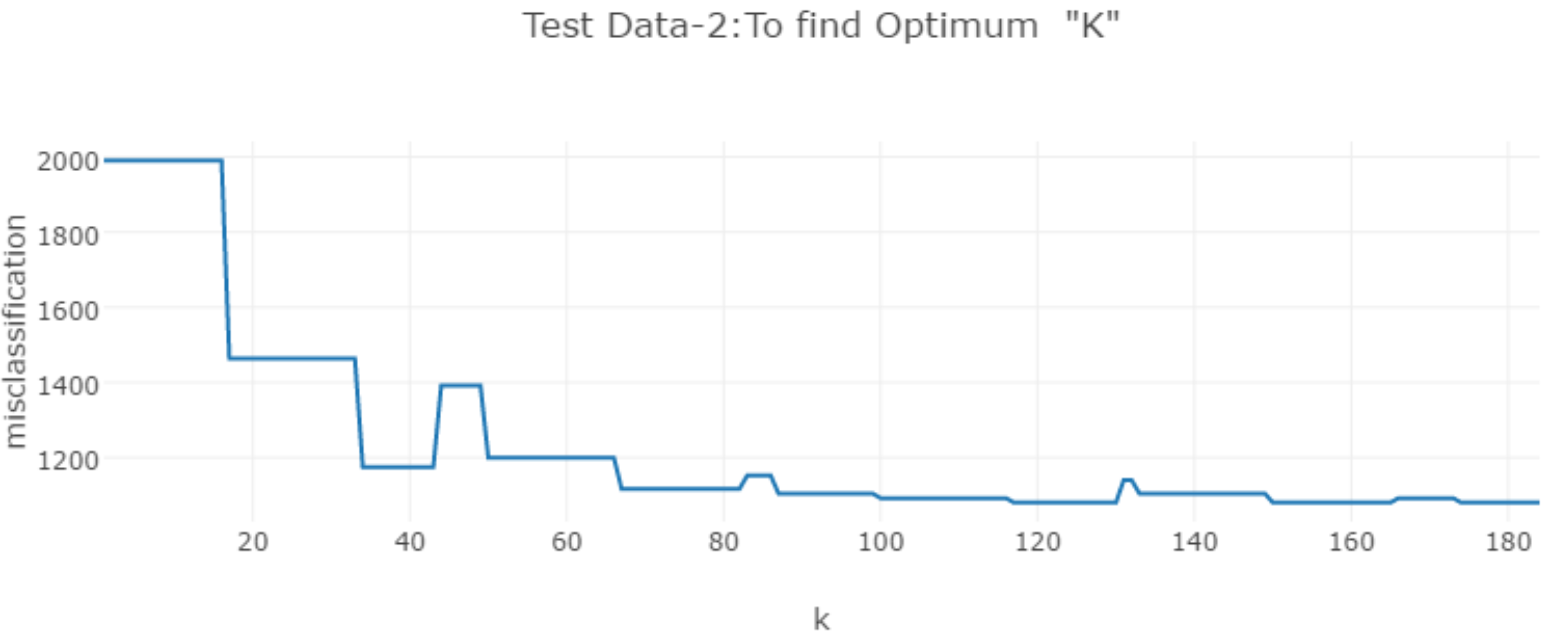
**Cross-validation Approach:**

A single observation (x1,y1) is used for the validation set, and the remaining observations {(x2,y2),...,(xn,yn)} make up the training set. The statistical learning method is ﬁt on the n − 1 training observations, and a prediction ˆ y1 is made for the excluded observation, using its value x1. Since (x1,y1) was not used in the ﬁtting process, error rate 1 = (y1 − ˆ y1)2 provides an approximately unbiased estimate for the test error. But even though error rate1 is unbiased for the test error, it is a poor estimate because it is highly variable, since it is based upon a single observation (x1,y1). We can repeat the procedure by selecting (x2,y2) for the validation data, training the statistical learning procedure on the n−1 observations {(x1,y1),(x3,y3),...,(xn,yn)}, and computing error rate 2 =( y2−ˆ y2)2. Repeating this approach n times produces n squared errors, error rate 1,...,error rate n. The LOOCV estimate for the test MSE is the average of these n test error estimates:





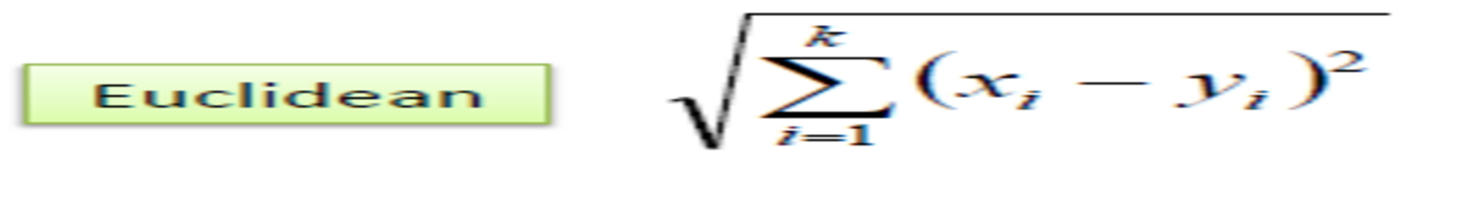




KNN approach for classiﬁcation, as a function of the value of K (which in this context indicates the number of neighbors used in the KNN classiﬁer, rather than the number of CV folds used). Again the training error rate declines as the method becomes more ﬂexible, and so we see that the training error rate cannot be used to select the optimal value for K. Though the cross-validation error curve slightly underestimates the test error rate, it takes on a minimum very close to the best value for K.

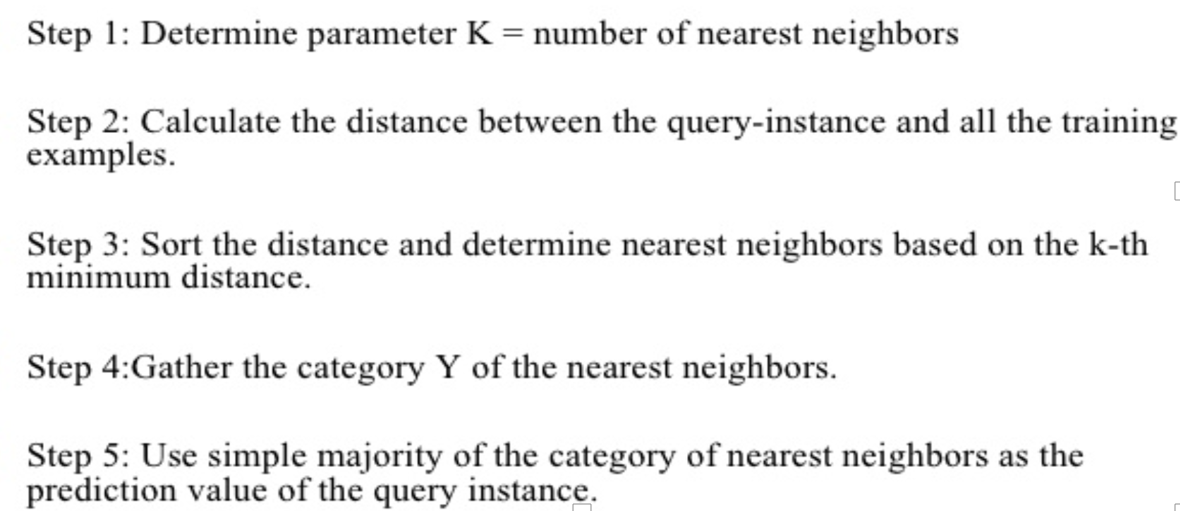
**Picking a Distance Metric:**

I am using Euclidian distance to computer the distance between a test observation and a training observation.



**Serial implementation:**

**Pseudo Code:**

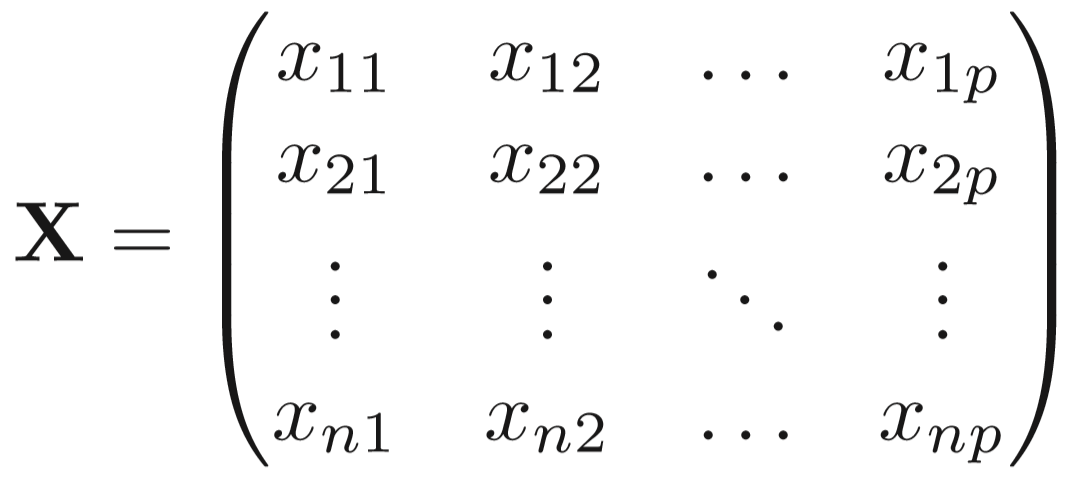


**Describing Code:**

**Dataset understanding:**

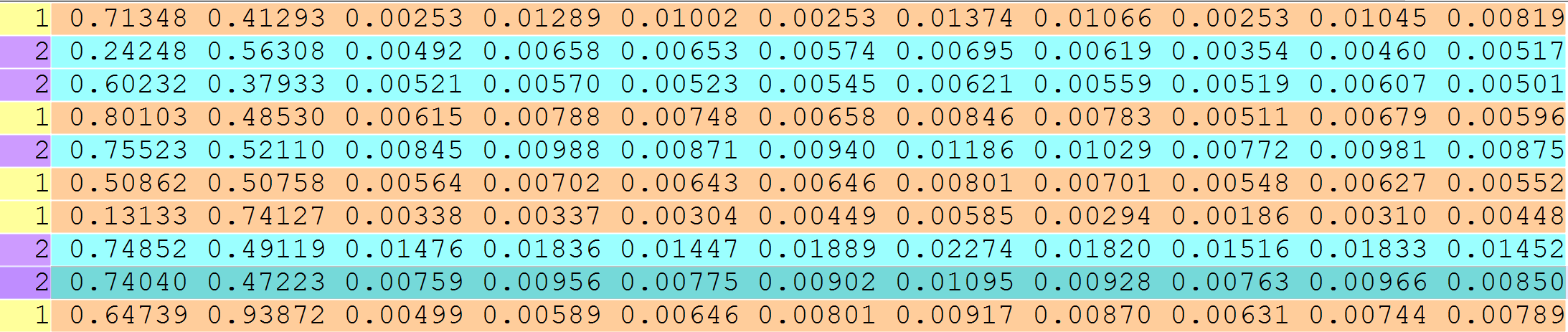
 represent the value of the jth variable for the ith observation, where i =1 ,2,...,nand j =1 ,2,...,p.

where, i will be used to index the samples or observations (from 1 to n) and j will be used to index the variables (from 1 to p). We let X denote a n×p matrix whose (i,j)th element is 



**Variables used:**

Train (n,m): Training Dataset {n is number of rows & m is the number of columns}



test(n,m) : Test Dataset{n is number of rows & m is the number of columns}



trainC(n): Contains Train class labels

testC(n): Contains test class label{simulated data, practically this is not available as shown above in test data }(just to verify my results with the predicted classification.)

class\_1: total count of observations in train datasets of class 1.

class\_2: total count of observations in train datasets of class 2.

TN:True Negatives

FP : false positives is count of observations incorrectly not assigned into category

FN :False Negatives

TP: true positive is count of correctly classified observations.

no\_class: observation not classified to any class

prior, p1 : calculate the conditional probabilities for each value of class\_1

prior, p2 : calculate the conditional probabilities for each value of class\_2

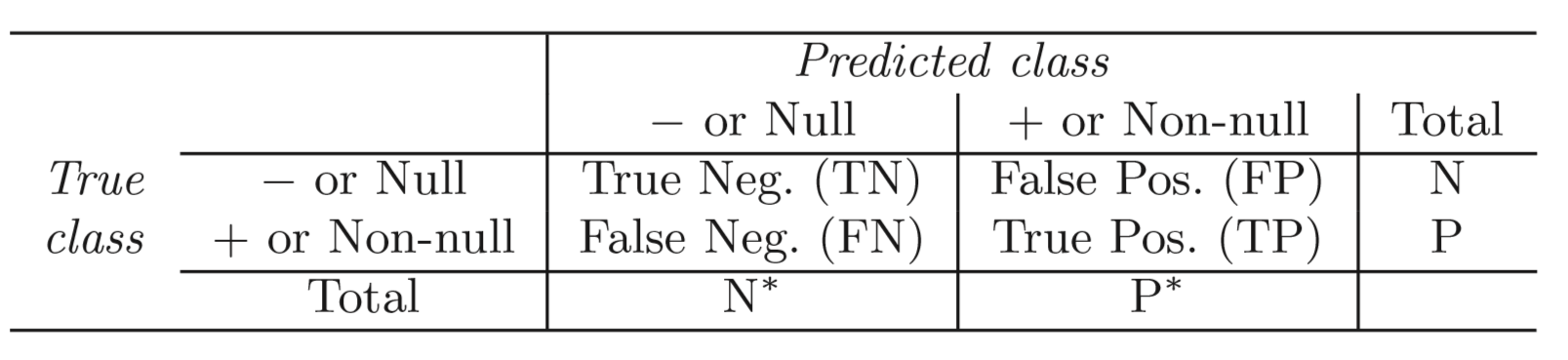
sensitivity: percentage of class2 that are identiﬁed correctly.

specificity: percentage of non-defaulters that are correctly identiﬁed

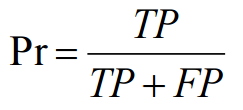
class\_assgn(n):Predicted class labels{compared with true test class labels(simulated once)}

**Efficiency of classification:**

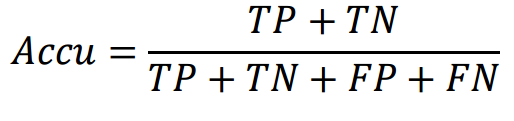
We can better understand the classification by creating a confusion matrix:



Precision (Pr) is defined as a probability that selected observations are classified correctly



Accuracy:



**Main program (main.f90):**

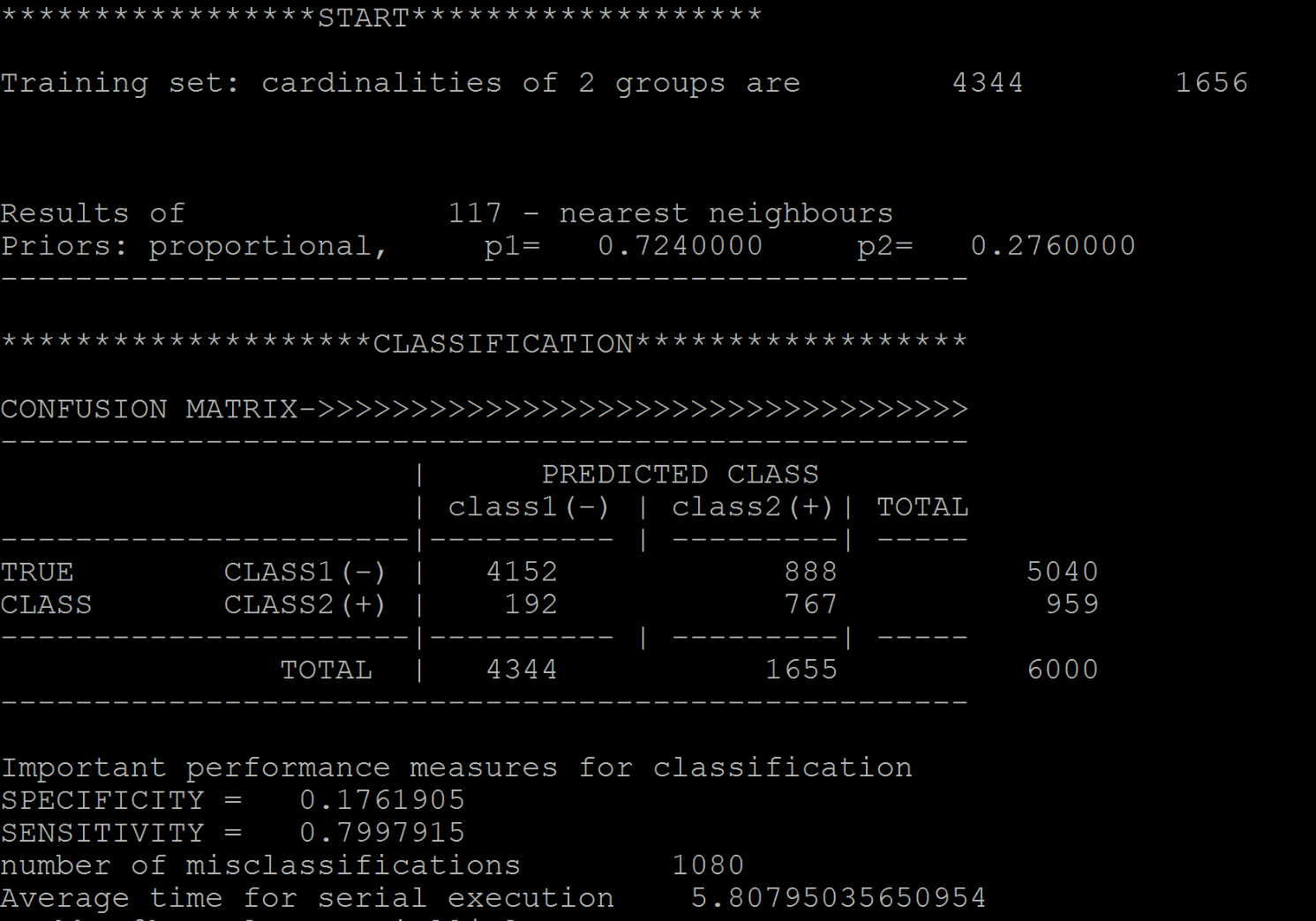
1. Taking data from Test and training files and segregating there class labels and saving them in trainC and TestC. Also, simultaneously creating the test and train matrix of size n\*m.
2. Calculating the probabilities of class1 and class2 training samples.
3. Calculating the class probabilities- PRIORS.
4. Sending priors, Train matrix, test matrix, value of k, and TrainC to **nearest\_neighbours subroutine.**
5. **……..{calling and receiving}…..**
6. Receiving the received valued in array CLASS\_asgn and comparing its values with testC to construct confusion matrix.
7. Analysing the confusion matrix as result.

**Subroutines:**

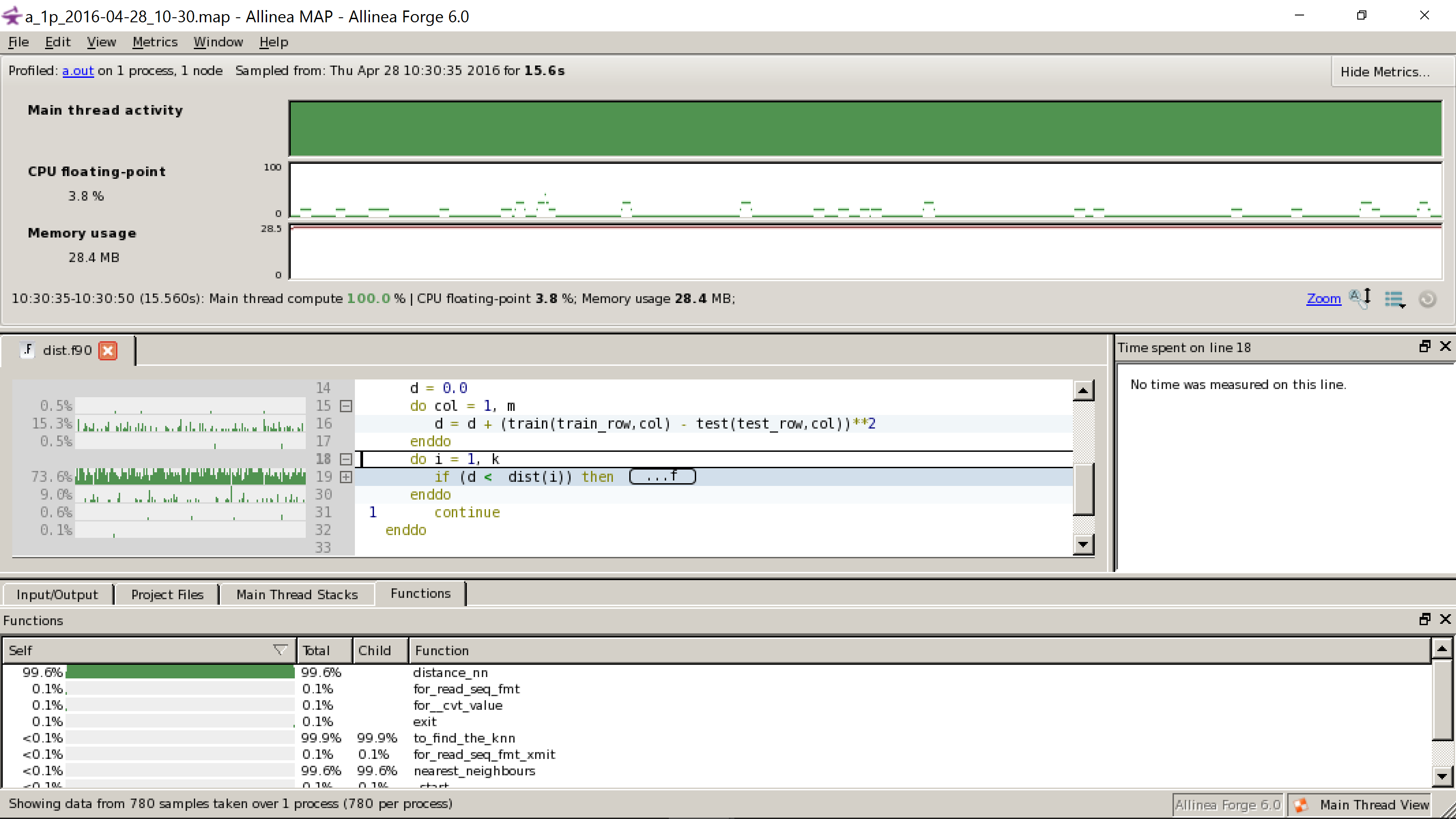
1. **nearest\_neighbours (knn.f90):**
   * 1. For each test row, the Train matrix, test matrix, value of k, and TrainC , are passed to subroutine distance\_NN.
     2. ….. {Calling and receiving}….
     3. Receiving the k-nearest training observation class label values in Class\_NN array corresponding to particular test row.
     4. Calculating the posterior probability (the predicted probabilities of observations of class\_1 and class\_2).
     5. Based upon probability calculation, assigning the test observations to a particular class label.
     6. Collecting and preparing all iteration over test rows in the array class\_assgn.
     7. Sending class\_assgn array back to main.f90 in order to create confusion matrix.
2. **distance\_NN (dist.f90):** 
   1. Initializing my dist array to maximum value and class\_NN value to 0.
   2. Calculated the distance from the particular test row to all the training rows iteratively and keep the “k” minimum (closest) distances in dist array and correspondingly taking the train class labels trainC values in class\_NN array.
   3. Finally the class\_NN array for single test row will be returned to nearest\_neighbours subroutine and the procedure gets repeated for next iteration of test row sent from nearest\_neighbours subroutine.

**Results:**

**Run for 6000 test rows and train rows.**



**Serial Code Profiling:**



**Serial Optimization:**

* Different Compiling options- -xHost,-o3,-ofast
* Replaced divisions by multiplications.

**Problem Defined (Reason to parallelize it?):**

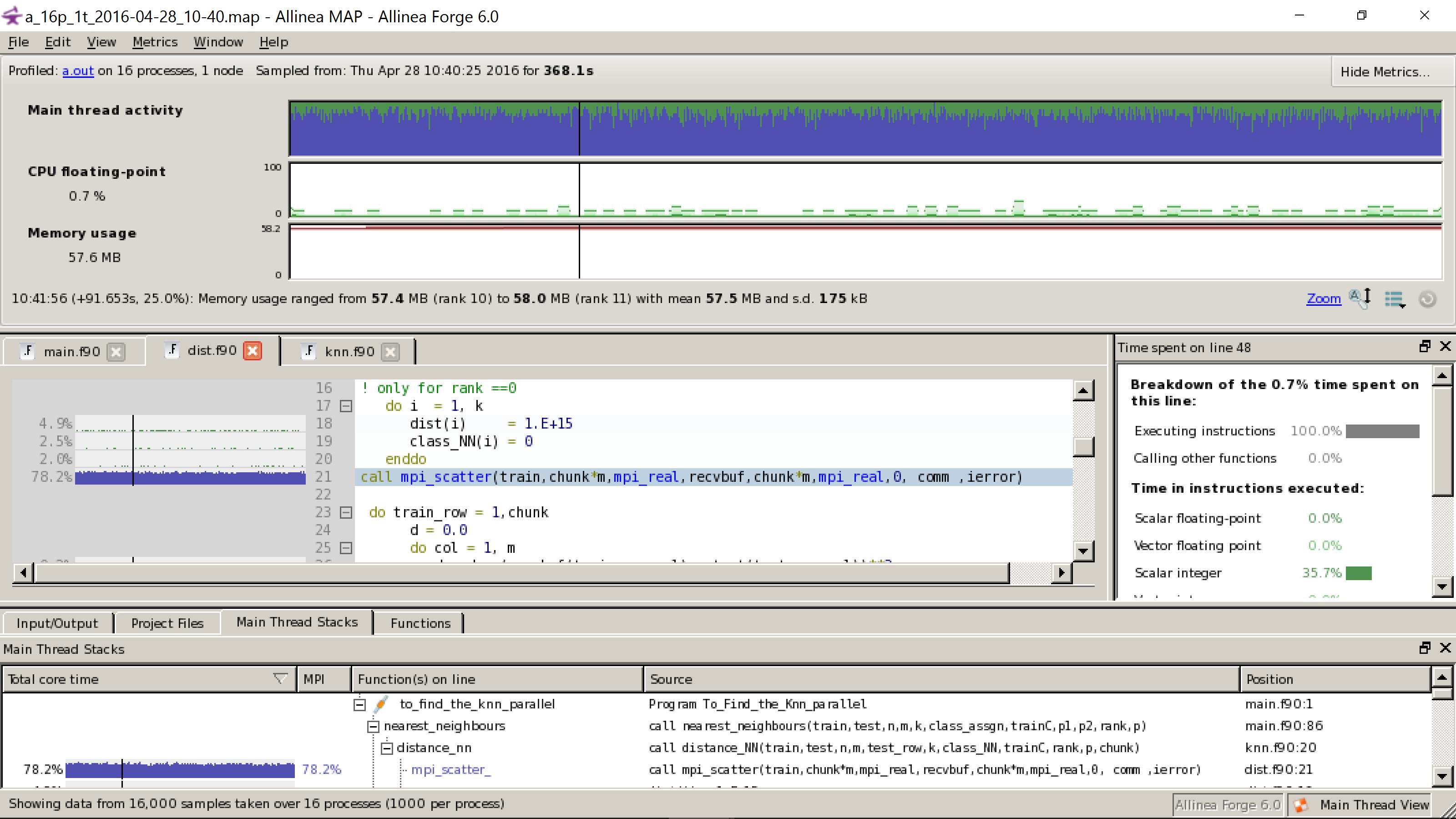
* Computationally expensive especially
* When the size of the training set becomes large
* When the number of dimension increases

Both cause the classification task to become very slow.

**Parallel Implementation:**

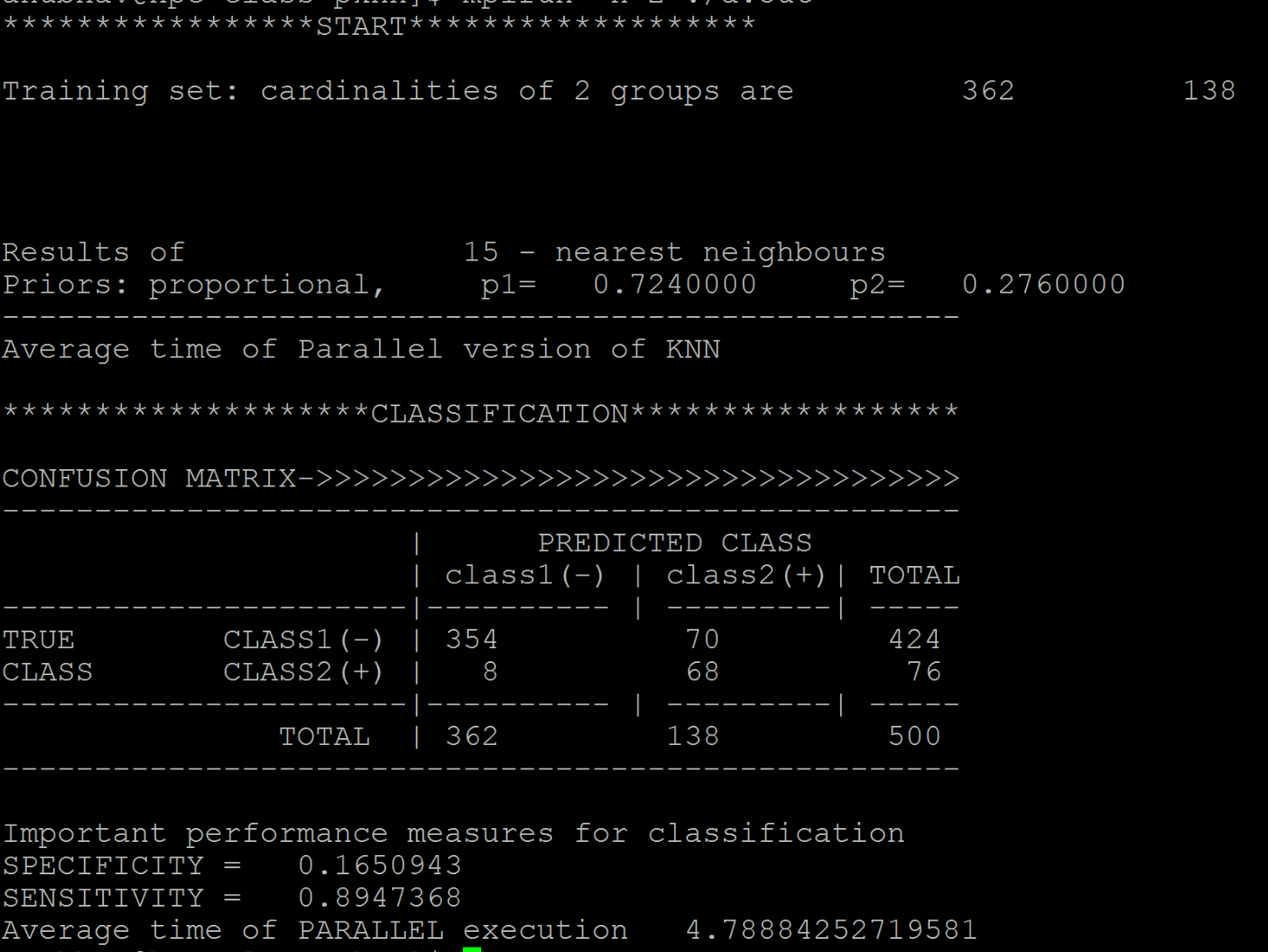
**Pseudo Code:**

* Step1: Considering 0th processor as master and and rest as slaves.
* Step2: P0 divides the training samples to equal chunks, and distributes all processors, keeping 1 chunk for local processing.
* Step3: Each individual processor now computes the distance measures independently and storing the computes measures in a local array “ppdist(chunk)”
* Step4: When each processor terminates distance calculation, the processor transmits a message to the master indicating end of processing
* Step5: Master then notes the end of processing for the sender processor and acquires the computes measures by copying them into its own array “gatherelements(n)”
* Step6: After the master has claimed all distance measures from all processors, the following steps are performed:
* Sort all distance measures in ascending order
* Select top “k” measures
* Count the number of classes in the top “k” measures
* The input element’s class will belong the class having the higher count among top k measures

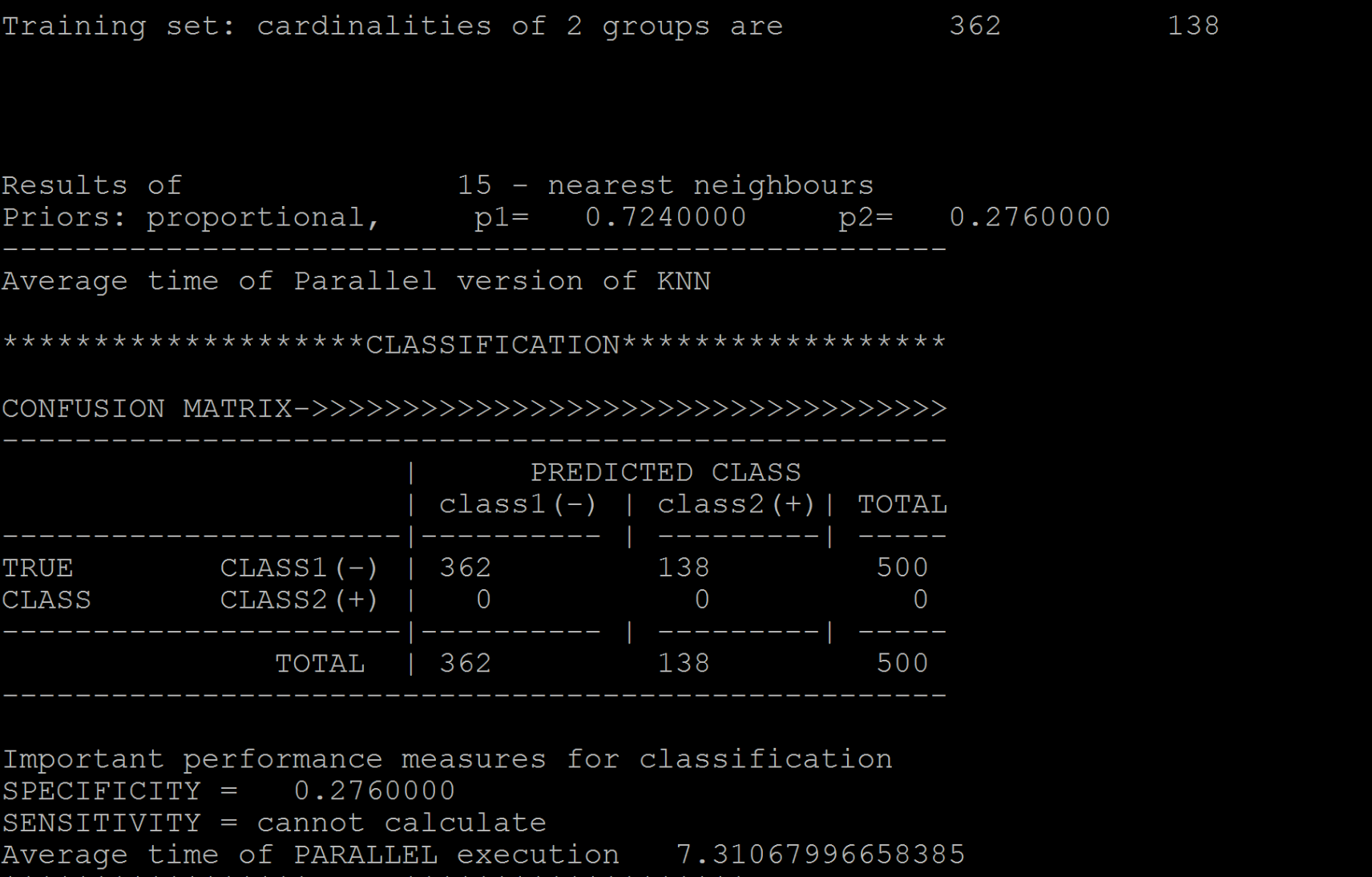
**Code profiling of Parallel Code:**

**Results:**

**Run for 500 training and test rows.(Parallel one)**



Parallel2 results:



The results varies in parallel execution.

For less data is gives correct classification but as the data size increases the number of misclassification and time increases instead of decreasing.

I tried with second data distribution but unable to reach to the results because of time constraints and hence kept in future improvements. I have tried the approach and attached the code of it. It needs few improvements.

**Data Distribution and load balancing:**

1. I have distributed my training data to all the processors and kept whole test data on every processor. The ideology behind choosing this distribution are
2. In practice, the algorithm/ model is trained on large number of training data in order to predict future test observations. So, test observations are usually less that the training observations. This way load on every processors will be of calculation.
3. In each iteration, only one test row is used to compute the distance from n train observation so it is better to distribute the load (train) in equal chuck size over all processors for computations.

**Parallel 2 approach**:

I tried to distance computation and sorting on the same processors to reduce the load on 0th processor and tried to send the changed distance and class\_NN array to subsequent processors in a round robin fashion. On single processor, I tried to distribute the work of distance calculation and sorting of each chunk with the help of threads using OPENMP.

**Conclusion:**

By implementing the parallel K-NN, the running time of the algorithm on CPU has been reduced which would make the algorithm faster, more efficient than the serial K-NN.

Parallel implementation greatly increased the speed of the KNN algorithm by reducing its time complexity from O(N) ,where

N is the number of records, to O(N/p) where

p is the number of processors.

**Future work:**

* Evaluate classification by using **different Distance metrics** and respectively checking their prediction.
* And concluding which distance Metric suits best to this particular dataset.
* Add **parallelization of CV approach** in the program.
* Use parallel K-NN over **datasets** which have **more dimensions(columns)** and evaluate the **prediction accuracy**.
* Implementing the second parallelizing approach to make it more efficient.

**References**

1. <http://www-bcf.usc.edu/~gareth/ISL/>
2. Parallel Implementation and Investigation of Ensemble KNN, Firat Tekiner1, Mike Pettipher1, Ian Whittley2, Tony Bagnall2 ,1: Manchester Computing, Kilburn Building, University of Manchester, M13 9PL, UK 2: School of Computing Sciences, University of East Anglia, Norwich, UK
3. http://en.wikipedia.org/wiki/Parallel\_computing
4. Chapman B., Jost G., Van Der Pas R. Using Open MP: portable shared memory parallel programming. MIT press, 2008
5. Zhang M. L., Zhou Z. H. ML-KNN: A lazy learning approach to multi-label learning. Pattern recognition, 2007, 40(7): 2038-2048.
6. Weinberger K., Blitzer J., Saul L. Distance metric learning for large margin nearest neighbor classification. Advances in neural information processing systems, 2006, 18: 1473
7. http://en.wikipedia.org/wiki/Levenshtein\_distance
8. Open MP API Specification, [www.OpenMp.org](http://www.openmp.org/)
9. Jim Beveridge, Robert Wiener. Multithreading Applications in Win32. Addison-Wesley, 1997