Exploration of parallel implementation of Machine Learning algorithms for shared memory architecture

Team #1
Siddhant Kulkarni
Ritesh Sangurmath

Motivation

Over the past couple of decades, machine learning has seen an exponential growth in research interest from scholars in all domains. As the value of information becomes increasingly evident, this growth does not show signs of slowing down. The fact that machine learning can be applied to problems in almost every domain has opened doors that we could have never imagined. Today, it plays a role in everything from research on particle accelerators to simply finding out if a person who bought diapers will buy beer. This appeal has also introduced several new challenges in the field of machine learning. Introduction of World Wide Web and Business Analytics shifted the focus from "how do we deal with this lack of data?" to "how do we deal with all this data?!". To deal with the large volumes and extreme velocity of the data being produced from billions of data sources has become a major area of focus for several researchers. Specific applications require these huge amounts of data to be processed in near-real time (for example, self-driving cars need to take split second decisions).

To address this problem, researchers have started looking at parallel and distributed computing [1] as an alternative to traditional single processors single node computing. The reason being that the traditional systems are simply no longer capable of handling such computations. As part of this project we intend to implement and analyze machine learning algorithms and the applicability of parallel computing to improve the performance of the same.

Primary objective of this project is to develop serial and parallel implementations of two machine learning algorithms and analyze the challenges and possible solutions.

Introduction to Supervised learning

Supervised learning has always been a crucial part of day to day life for humans as well as computers. The concept of supervised learning stems from the way in which humans gain their knowledge and learn to identify different objects and conditions. For example, we know that there is a good chance of snow in the month of January. We know this based on the knowledge we have gathered by living in Colorado over the years. Here, the event of snowing is related to a certain time based on the knowledge we gathered from our experience. This concept can also be applied to computers using algorithms capable of learning from labelled data and using the extracted knowledge to predict the label of some input unlabelled data. These algorithms are called Classifiers. Classifiers make using of labelled data (aka Training data) to extract knowledge and then apply this knowledge to unlabelled data (aka Testing data) to predict the labels. Following diagram shows the conventional classification process.

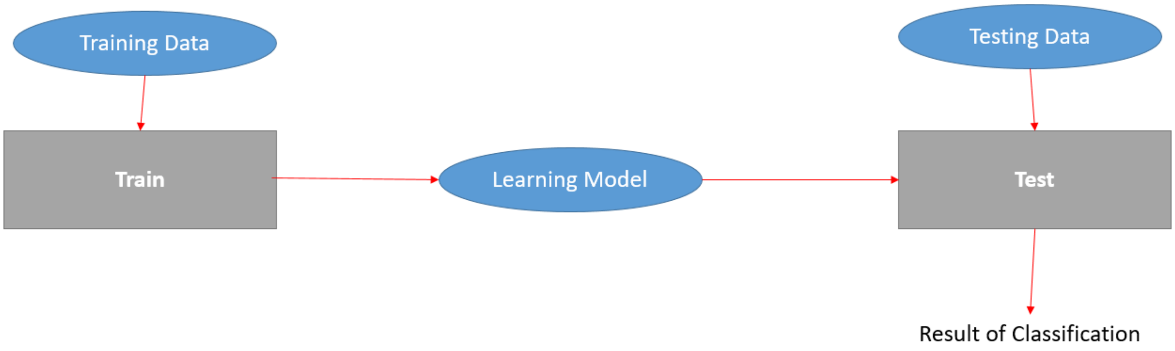


Fig 1. Conventional classification process.

This project focuses on one classifier which is popularly known as “Naïve Bayesian” classifier.

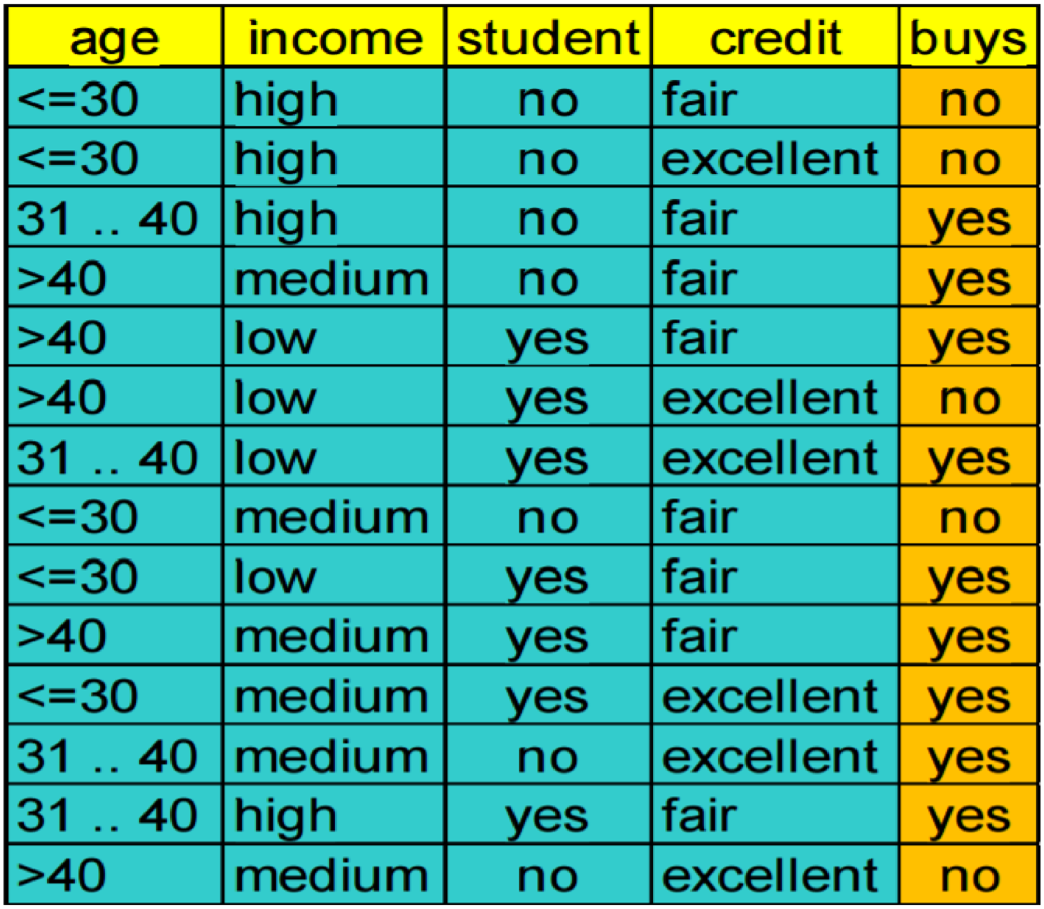
Naïve Bayesian Classifier

Naïve Bayesian classifier is a simple probabilistic classifier which works by applying the Baye’s theorem along with naïve assumptions about feature independence. Studies comparing classification algorithms have found Naive Bayesian comparable in performance with Decision Tree and few neural network classifiers. Naive Bayesian has also exhibited high accuracy and speed when applied to huge amounts of data [X]. It assumes value of any feature is independent of values of other features. This assumption is also known as Conditional Independence. Despite the naïve assumption and over simplification, Naïve Bayesian classifiers have proved to be quite useful in complex real world conditions. Probabilistic model of Naïve Bayesian Classifier is as follows:

$$P(X|C_i) = \prod_{k=1}^{n} P(x_k|C_i)$$

Naïve Bayesian considers the probability of a tuple X belonging to a class C_i is equal to the multiplication of probabilities that each attribute of the tuple X belongs to the class C_i where the probabilities $P(x_1|C_i)$, $P(x_1|C_i)$ and so on can be easily calculated from the training data sets. Once these probabilities are calculated for each attribute in X, the probabilities with respect to individual classes are multiplied. The class with maximum probability is selected as the result of the classification. The theoretical training and testing time complexity of Naive Bayesian is O(N*P) where N is number of records to be tested and P is the number of features. But while considering the practical implementation of Naive Bayesian classifier, we can see that the training complexity is 2 * O(N*P) as the probabilities must be calculated for each attribute value after getting the joint distribution counts and testing complexity is O(N*P*C) where C is the number of possible class values.

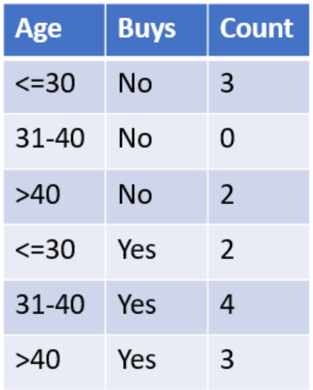
For example, Consider the following dataset which describes information about customers in a computer shop along with a label indicating whether they bought a computer in said shop.



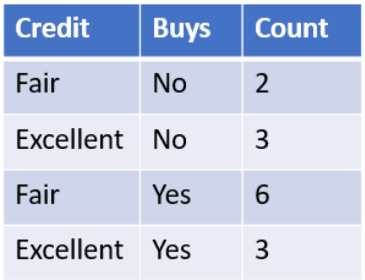
age	income	student	credit	buys
<=30	high	no	fair	no
<=30	high	no	excellent	no
31 .. 40	high	no	fair	yes
>40	medium	no	fair	yes
>40	low	yes	fair	yes
>40	low	yes	excellent	no
31 .. 40	low	yes	excellent	yes
<=30	medium	no	fair	no
<=30	low	yes	fair	yes
>40	medium	yes	fair	yes
<=30	medium	yes	excellent	yes
31 .. 40	medium	no	excellent	yes
31 .. 40	high	yes	fair	yes
>40	medium	no	excellent	no

Fig 2. Sample Dataset

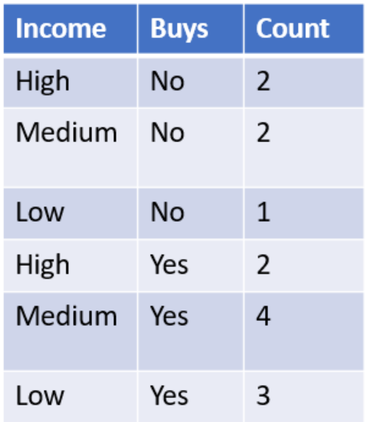
Given this dataset, naïve Bayesian will start by gathering frequency counts of the following combinations:



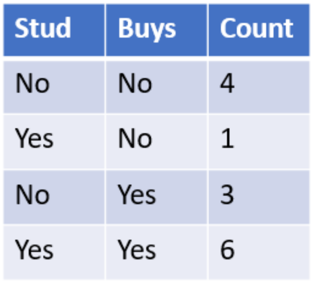
Age	Buys	Count
<=30	No	3
31-40	No	0
>40	No	2
<=30	Yes	2
31-40	Yes	4
>40	Yes	3



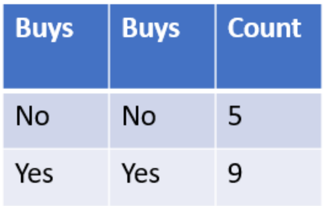
Credit	Buys	Count
Fair	No	2
Excellent	No	3
Fair	Yes	6
Excellent	Yes	3



Income	Buys	Count
High	No	2
Medium	No	2
Low	No	1
High	Yes	2
Medium	Yes	4
Low	Yes	3



Stud	Buys	Count
No	No	4
Yes	No	1
No	Yes	3
Yes	Yes	6



Buys	Buys	Count
No	No	5
Yes	Yes	9

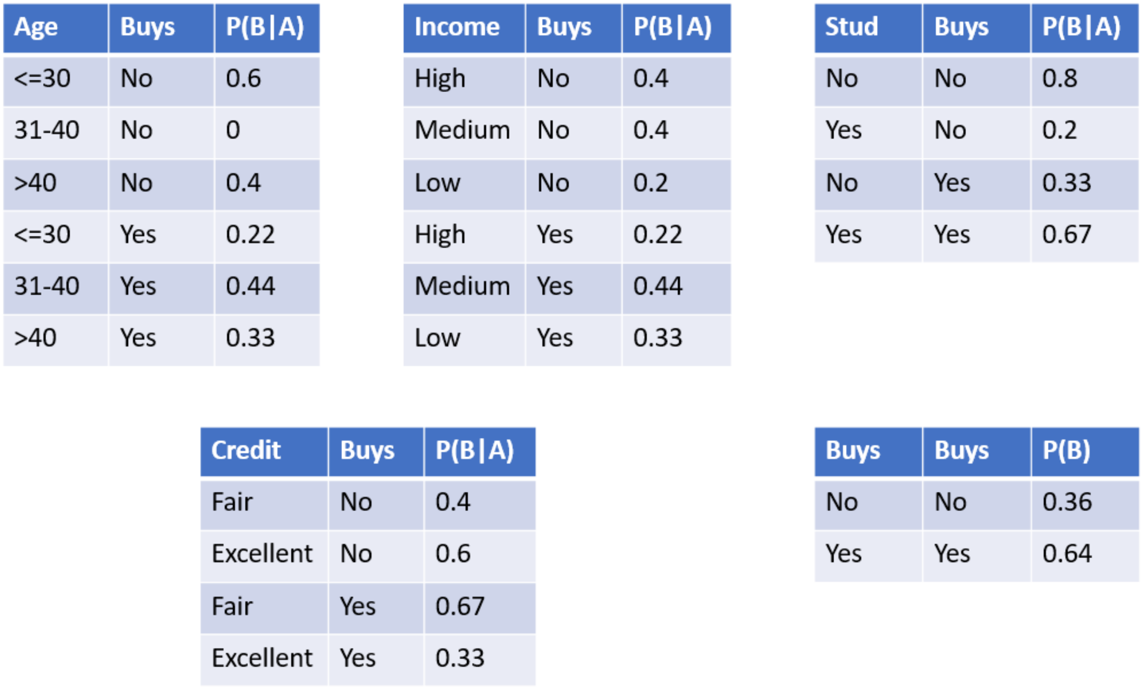
These frequency counts indicate the number of times a value of each of the columns has occurred along with each possible class. Simply put, these counts are spread across the Cartesian products of the domains of each of the non-class attribute and the domain of the class attribute. These counts allow Naïve Bayesian to build a probabilistic model by dividing the frequency counts of (value, class label) pairs by the frequency count of respective (class label, class label) pairs. For example, with the frequency counts given above, we can calculate the probability of the pairs (<=30, Yes) and (Fair, Yes) as follows:

$$P(\leq 30|Yes) = \frac{FC(\leq 30, Yes)}{FC(Yes, Yes)} = \frac{3}{9} = 0.33$$

And

$$P(Fair|Yes) = \frac{FC(Fair, Yes)}{FC(Yes, Yes)} = \frac{6}{9} = 0.67$$

Using this type of calculations, Naïve Bayesian comes up with the following probabilistic learning model:



Age	Buys	P(B|A)
<=30	No	0.6
31-40	No	0
>40	No	0.4
<=30	Yes	0.22
31-40	Yes	0.44
>40	Yes	0.33

Income	Buys	P(B|A)
High	No	0.4
Medium	No	0.4
Low	No	0.2
High	Yes	0.22
Medium	Yes	0.44
Low	Yes	0.33

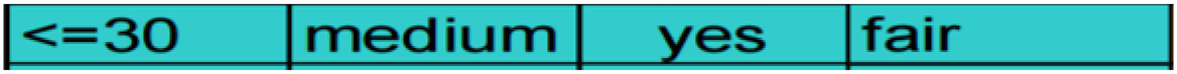
Stud	Buys	P(B|A)
No	No	0.8
Yes	No	0.2
No	Yes	0.33
Yes	Yes	0.67

Credit	Buys	P(B|A)
Fair	No	0.4
Excellent	No	0.6
Fair	Yes	0.67
Excellent	Yes	0.33

Buys	Buys	P(B)
No	No	0.36
Yes	Yes	0.64

Fig 3. Learning model of Naïve Bayesian

Once these have been calculated, the classifier is ready to classify a record. For example, let us assume a new customer record is provided to the classifier as follows:



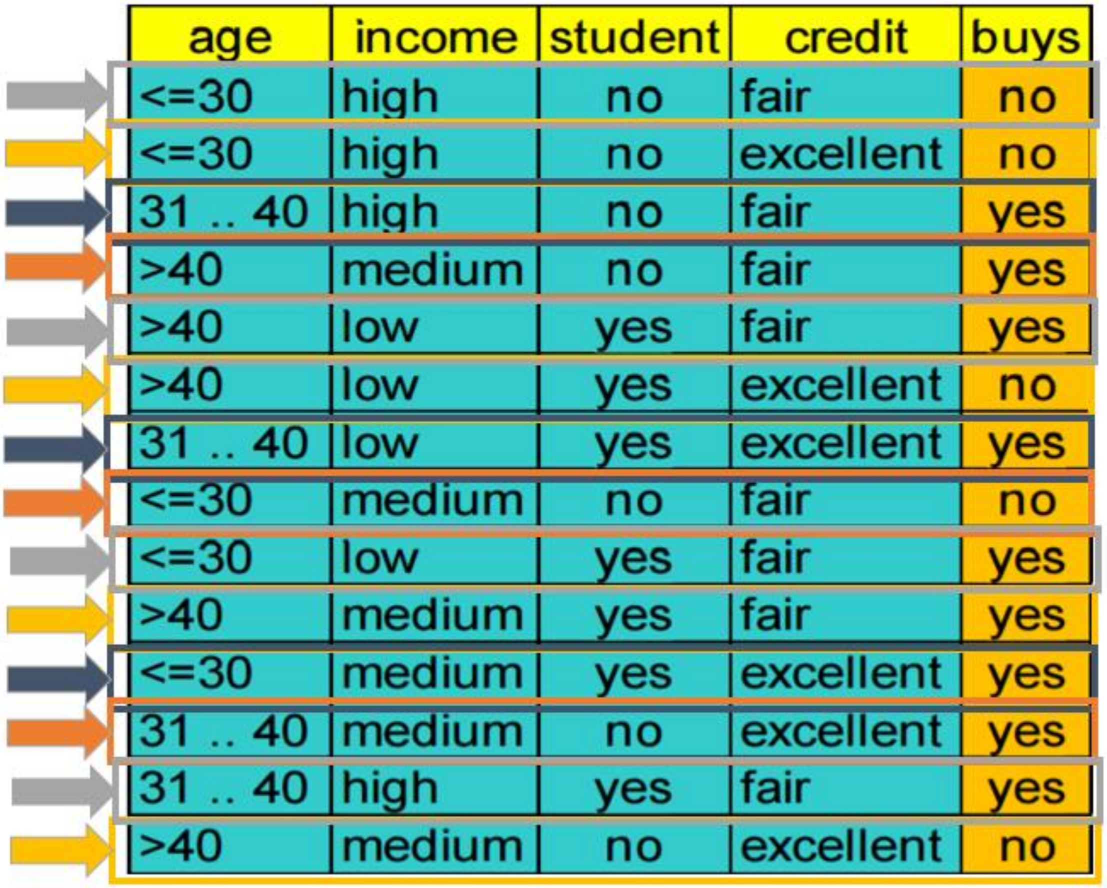
<=30	medium	yes	fair

If we call this customer X, we can use the learning model to predict whether they will buy a computer as follows,

- P (X|"yes") = 0.22*0.44*0.67*0.67*0.64 = 0.028
- P (X|"no") = 0.6*0.4*0.2*0.4*0.36 = 0.007
- Since, P(X|"yes") > P(X|"no")
  - We predict that this customer WILL buy

Parallel Naïve Bayesian

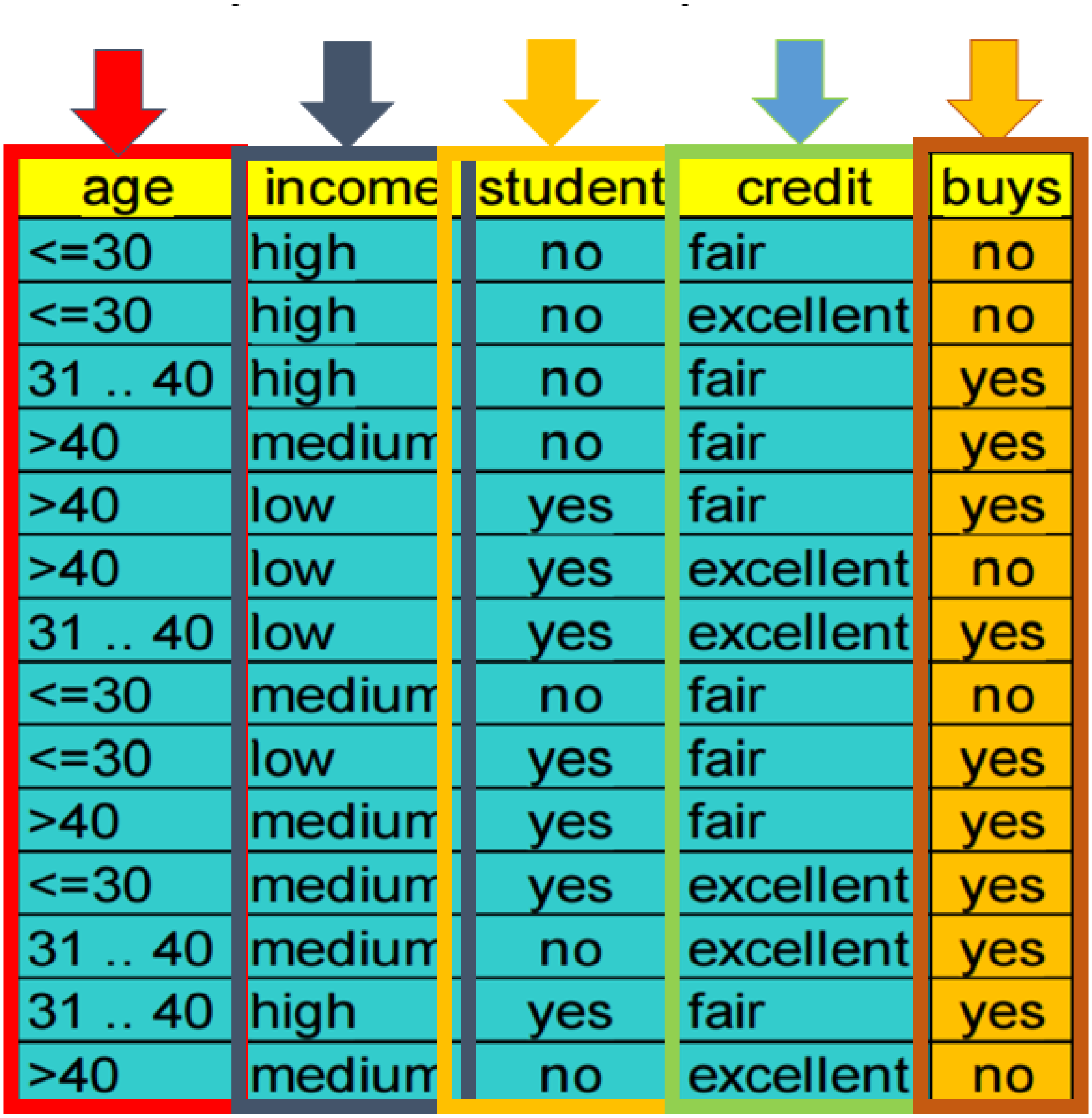
Naïve Bayesian is one of the few classifiers which are very well built for parallel computation. So much so that we might even be able to call it an embarrassingly parallelizable algorithm. Taking the concept of the classifier into consideration, there are two ways in which it can be parallelized. For now, let us focus only on the training part of the classifier, the first way to parallelize this algorithm is as follows:



age	income	student	credit	buys
<=30	high	no	fair	no
<=30	high	no	excellent	no
31 .. 40	high	no	fair	yes
>40	medium	no	fair	yes
>40	low	yes	fair	yes
>40	low	yes	excellent	no
31 .. 40	low	yes	excellent	yes
<=30	medium	no	fair	no
<=30	low	yes	fair	yes
>40	medium	yes	fair	yes
<=30	medium	yes	excellent	yes
31 .. 40	medium	no	excellent	yes
31 .. 40	high	yes	fair	yes
>40	medium	no	excellent	no

Fig 4. Row-major parallelization of Naïve Bayesian

As show in the above diagram, we can make use of cyclic work distribution with multiple threads to be able to cover several rows of the dataset at a time. However, because Naïve Bayesian maintains counts of combinations of values and classes, this type of an approach will take a significant hit on the performance side as once each row has been processed for training, the classifier will need to go in and update the frequency count. This means that multiple threads might have to wait for their turn to update the frequency counts while another thread is doing the same. A better way to do this will be to capitalize on the conditional independence assumption of the Naïve Bayesian classifier and follow a Column-major parallelization approach.



age	income	student	credit	buys
<=30	high	no	fair	no
<=30	high	no	excellent	no
31 .. 40	high	no	fair	yes
>40	medium	no	fair	yes
>40	low	yes	fair	yes
>40	low	yes	excellent	no
31 .. 40	low	yes	excellent	yes
<=30	medium	no	fair	no
<=30	low	yes	fair	yes
>40	medium	yes	fair	yes
<=30	medium	yes	excellent	yes
31 .. 40	medium	no	excellent	yes
31 .. 40	high	yes	fair	yes
>40	medium	no	excellent	no

Fig 5. Column-major parallelization of Naïve Bayesian

We can apply both approaches to training as well as testing phases. These approaches can be applied to testing only if two critical assumptions are true:

1. We have all the training and testing data captured and stored
2. We are not making use of an incremental training model

These assumptions present challenges on two fronts, assumptions regarding data as well as the way the learning model is designed.

Let us look at what implications the falsification of these assumptions has on Naïve Bayesian:

Incremental Learning Model

Many times, with machine learning applications, the training data turns out to be either insufficient or in-accurate. In this case, an incremental model can help us update the learning model as we test unlabelled data points. Incremental model adds a feedback loop to the conventional classification model to consistently update the learning model as results of classification are produced. Following figure shows how this can be achieved:

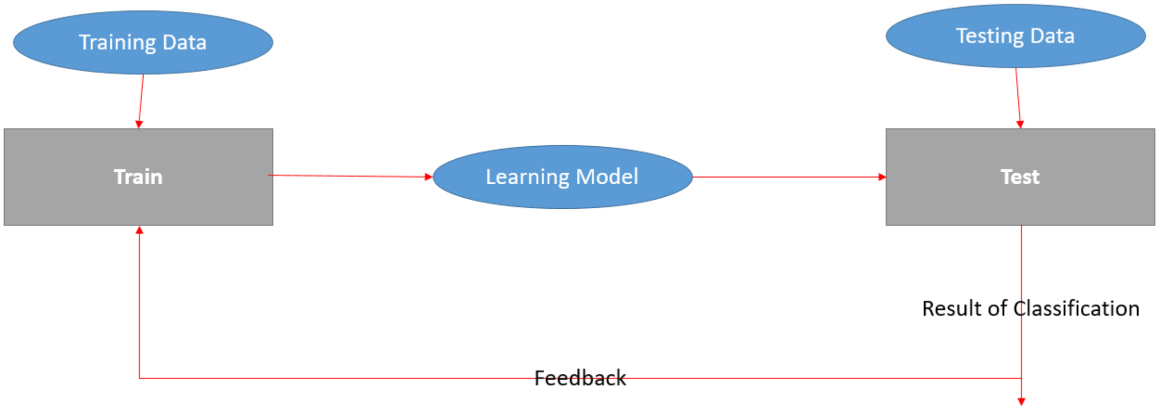
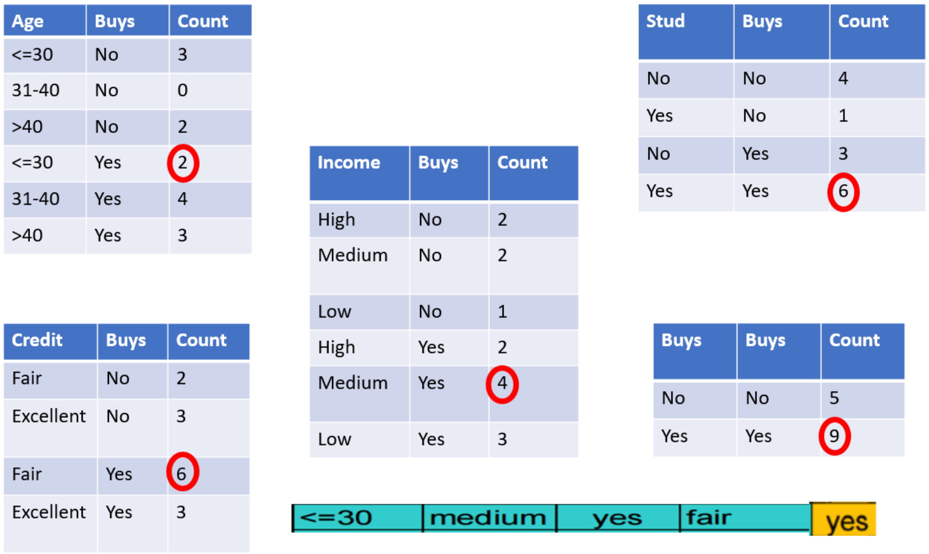


Fig 6. Incremental Learning model

Going back to our example, let us look at how we can update the learning model for classification:

After we are done classifying the new unlabelled record, we can update the learning model as follows:



Age	Buys	Count
<=30	No	3
31-40	No	0
>40	No	2
<=30	Yes	2
31-40	Yes	4
>40	Yes	3

Stud	Buys	Count
No	No	4
Yes	No	1
No	Yes	3
Yes	Yes	6

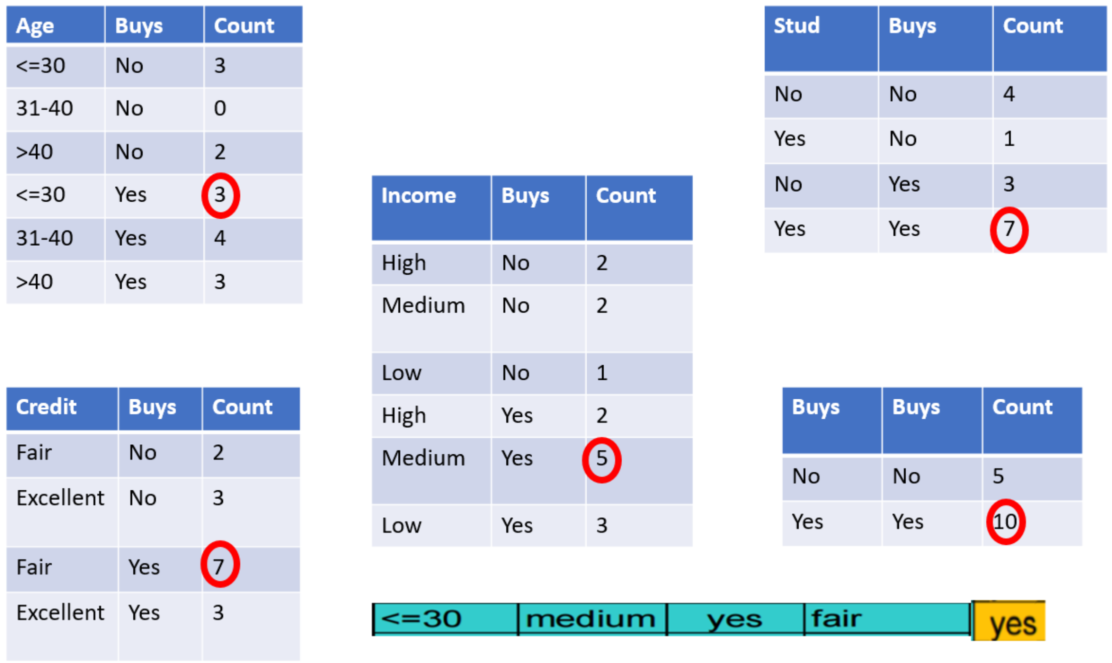
Income	Buys	Count
High	No	2
Medium	No	2
Low	No	1
High	Yes	2
Medium	Yes	4
Low	Yes	3

Credit	Buys	Count
Fair	No	2
Excellent	No	3
Fair	Yes	6
Excellent	Yes	3

Buys	Buys	Count
No	No	5
Yes	Yes	9

<=30	medium	yes	fair	yes

Fig 7. Before Incremental learning



Age	Buys	Count
<=30	No	3
31-40	No	0
>40	No	2
<=30	Yes	3
31-40	Yes	4
>40	Yes	3

Credit	Buys	Count
Fair	No	2
Excellent	No	3
Fair	Yes	7
Excellent	Yes	3

Income	Buys	Count
High	No	2
Medium	No	2
Low	No	1
High	Yes	2
Medium	Yes	5
Low	Yes	3

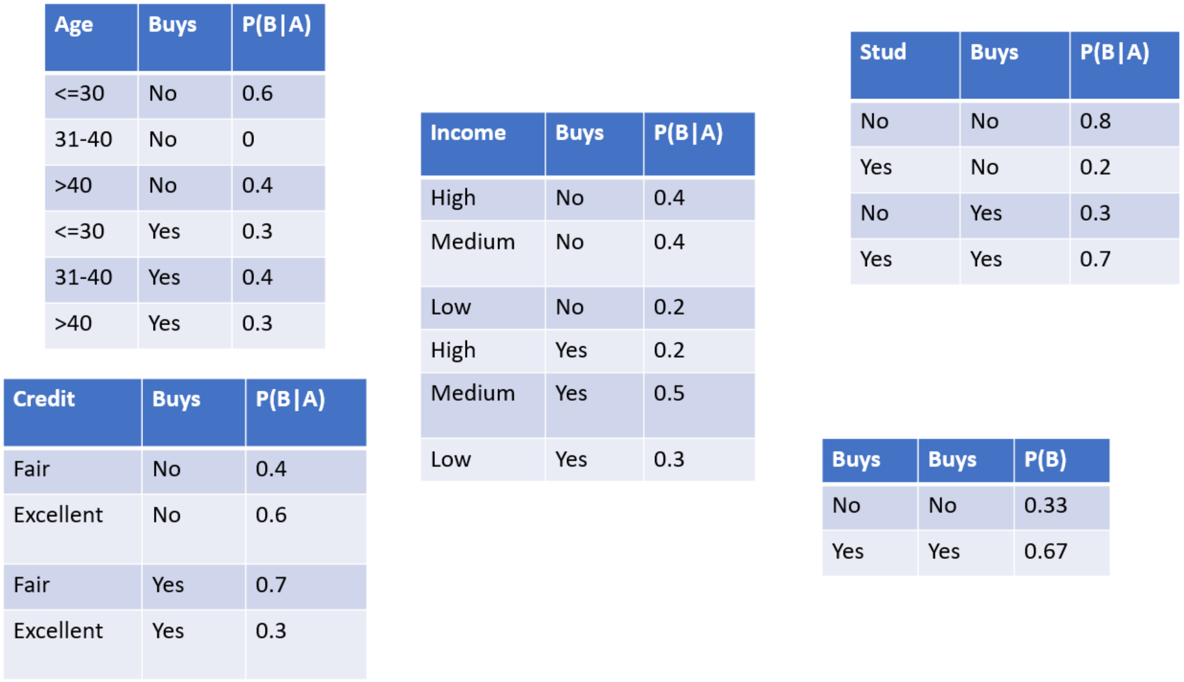
Stud	Buys	Count
No	No	4
Yes	No	1
No	Yes	3
Yes	Yes	7

Buys	Buys	Count
No	No	5
Yes	Yes	10

<=30	medium	yes	fair	yes

Fig 8. After incremental learning

As we can see, the frequency counts are updated using the classified record. We can then use these updated frequency counts to calculate the probabilities as follows:



Age	Buys	P(B|A)
<=30	No	0.6
31-40	No	0
>40	No	0.4
<=30	Yes	0.3
31-40	Yes	0.4
>40	Yes	0.3

Credit	Buys	P(B|A)
Fair	No	0.4
Excellent	No	0.6
Fair	Yes	0.7
Excellent	Yes	0.3

Income	Buys	P(B|A)
High	No	0.4
Medium	No	0.4
Low	No	0.2
High	Yes	0.2
Medium	Yes	0.5
Low	Yes	0.3

Stud	Buys	P(B|A)
No	No	0.8
Yes	No	0.2
No	Yes	0.3
Yes	Yes	0.7

Buys	Buys	P(B)
No	No	0.33
Yes	Yes	0.67

Fig 9. Updated Probabilities

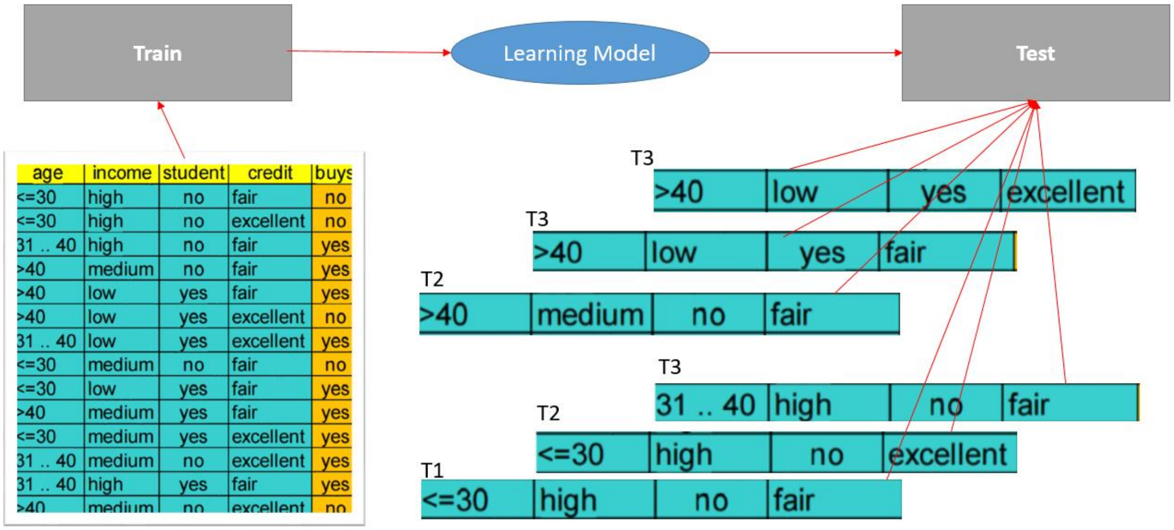
We can then make use if these probabilities to classify the next unlabelled record.

Incremental learning poses challenges on two fronts, machine learning and parallel programming. If we consider machine learning, we cannot make use of each classified record to update the learning model because that will mean that records that were incorrectly classified will also end up updating the learning model. This means that we will be updating the learning model using incorrect information. This will end up making the learning model worse whereas the initial goal of this type of an approach was to make it better. On the parallel programming front, there are two issues. First, the operation of updating the learning model needs to be an atomic operation. We cannot have threads classifying unlabelled records while another thread is halfway through updating the learning model. Second, if we were to implement this operation, we cannot use each classified record for updating the learning model, because this will mean that every time a thread needs to update the learning model, other threads will need to wait for it to be done before beginning classification of the next unlabelled record.

To address these problems, we can apply two solutions. First, we can specify certain constraints on the classified records for them to be valid for updating the learning model. For example, in our case, we can apply a constraint that says that a record is only valid if the probability of the predicted class is X more than the probability of the class with second highest probability. This faces two challenges at once. Such constraint affords us the luxury to say that we are only using records that we accurately classified to update the learning model. On the other hand, this will reduce the number of records that update learning model and effectively reduce the amount of time threads spend waiting on other threads updating the learning model. Second way to target these issues is to delay the update of learning model until we face a certain number of valid records and then updating the learning model at once for all of them. This will reduce the waiting time for threads by delaying the process to update the learning model.

Stream processing

In real life applications of machine learning, the data is never statically available for classification. It can be flowing into our classification system at different rates and from different sources. This type of data presents a unique challenge to parallel programming the work distribution that it follows. To address this type of data, we need to be able to buffer the incoming records as we cannot know the quantity and velocity of the data that we need to classify. Even with the fastest classification model, we cannot be sure that our model will be able to classify the incoming records in real time. This breeds the need for buffering incoming records. This targets the issue of stream processing on two fronts. First, this allows for handling records that are too much to be classified in real time. Second, it will allow for dynamic work assignment to be carried over a paradigm like thread pools where each thread waits for the next task to be assigned. Following diagram shows the stream processing based classification model and records flowing in at different times indicated by T_i.



age	income	student	credit	buys
<=30	high	no	fair	no
<=30	high	no	excellent	no
31 .. 40	high	no	fair	yes
>40	medium	no	fair	yes
>40	low	yes	fair	yes
>40	low	yes	excellent	no
31 .. 40	low	yes	excellent	yes
<=30	medium	no	fair	no
<=30	low	yes	fair	yes
>40	medium	yes	fair	yes
<=30	medium	yes	excellent	yes
31 .. 40	medium	no	excellent	yes
31 .. 40	high	yes	fair	yes
>40	medium	no	excellent	no

Fig 10. Stream processing based classification model

As a part this project we have implemented two versions of the serial Naïve Bayesian classifier: with and without incremental learning model. We have also implemented column-major parallelized training and row-major parallelized testing.

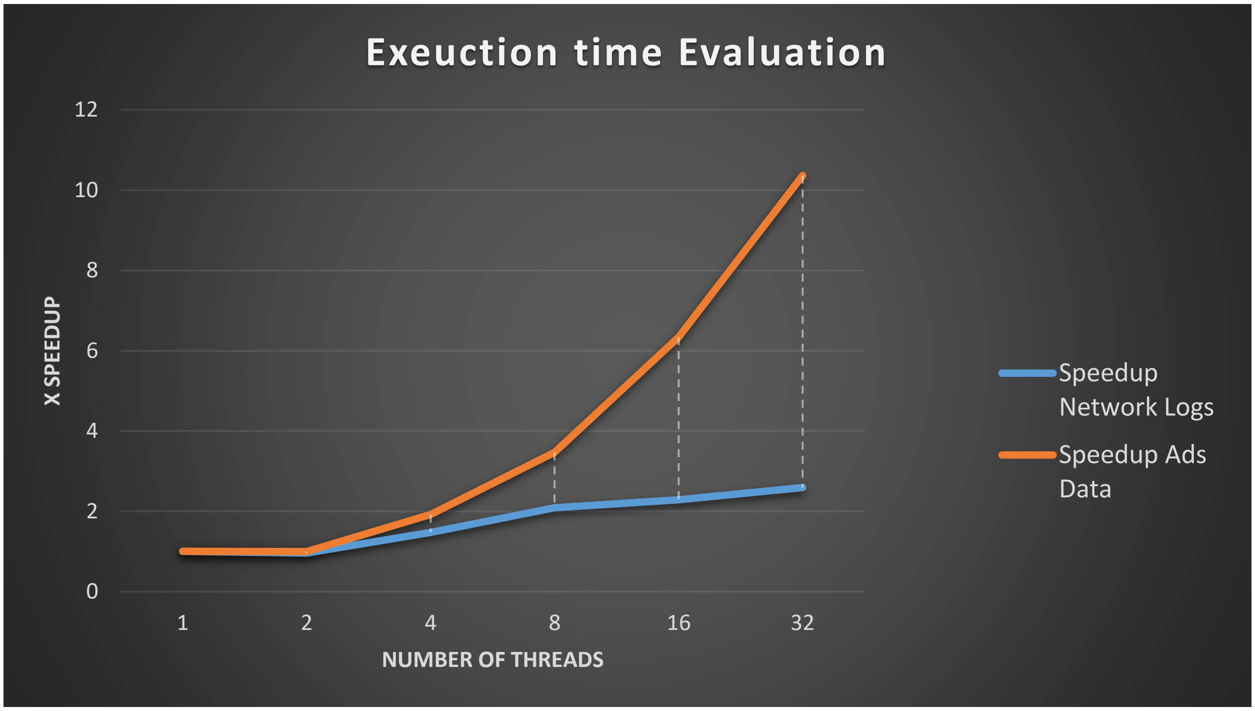


Fig 11. Speedup with parallel Naïve Bayesian (non-incremental model with static data)

Introduction to K-Means

K-means algorithm is one of the simplest unsupervised learning algorithm which solves the clustering problem. K-means algorithm is the example of the partitional clustering, in which all the data points will be divided into the K number of clusters. The data points will be assigned to cluster based in the minimum distance between the data point and the cluster centroid. Initially the distance will be measure between the data point and all the cluster centroid and based upon the minimum distance, the data point will be assigned to that cluster.

K-Mean Algorithm Process: [1]

Step 1: Select the K number of random points from the set of data point which will be considered as the cluster centroid.

Step 2: Find the distance between each data point to the K number of cluster centroid. Select the Cluster centroid to which the data point has minimum distance and assign the data point to that cluster. Euclidean distance Function will be used for calculating the distance between the data point and the cluster centroid.

Step 3: Perform step 2 for all the data points and at end of the step 3 all the data points should be assigned to their respective clusters.

Step 4: After the end of the step 3, for each K number of cluster calculate the mean of all the data points present in that cluster and the mean value represents the new location for the cluster centroid. Perform this step for all the clusters.

$$\boldsymbol{M_j} = \sum_{i_l \in Cj} \boldsymbol{i_l} / \, |\boldsymbol{C_j}|$$

Above is the formula used for calculating the mean of all the data points present in the respective cluster, were "j" value represent the K number of cluster centroid, i_l represent the represent the data point in the respective cluster and Cj represent the number of data point in that cluster.

Step 5: After getting the new location for K number of cluster centroid repeat step 2, step 3 and step 4 until the cluster centroid location does not change their respective location or the error function which is mentioned below does not changes.

$$\mathrm{E} = \sum_{j=1}^{k} \sum_{i_l \in Cj} \boxed{|\, \boldsymbol{i_l} - \boldsymbol{w_j} |^2}$$

Euclidean distance between the data point and the cluster centroid

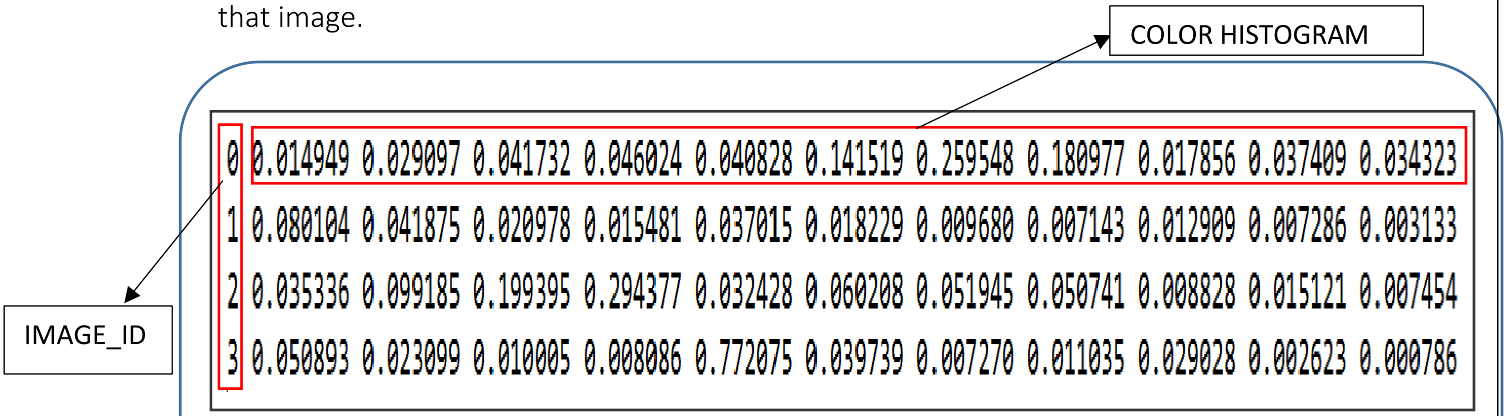
Step 6: At this step the final cluster centroid and the list of data points in their respective cluster will be returned. The final cluster centroid location completely varies based upon the selection of the random cluster centroid from the data points in the step 1.

Dataset Used

The main aim for this project is to perform the clustering using the K-means algorithm on the image retrieval, so the dataset used for this implementation is the set of images were each image consist of vector of float value which represent the color histogram of the image. The set of values in each image are the color density in that images. In the single dataset, there are 68,040 set of images with various category [2]. Below are the sample images present in this dataset.

FIG 12

In the dataset, each line indicates the single image, were the first value in the starting of the line indicates the image_ID and the rest other value is the color histogram values of that image.



```
0 0.014949 0.029097 0.041732 0.046024 0.040828 0.141519 0.259548 0.180977 0.017856 0.037409 0.034323
1 0.080104 0.041875 0.020978 0.015481 0.037015 0.018229 0.009680 0.007143 0.012909 0.007286 0.003133
2 0.035336 0.099185 0.199395 0.294377 0.032428 0.060208 0.051945 0.050741 0.008828 0.015121 0.007454
3 0.050893 0.023099 0.010005 0.008086 0.772075 0.039739 0.007270 0.011035 0.029028 0.002623 0.000786
```

FIG 13

Serial Implementation

Procedure:

1. The user input will be the text file which consists of images and its respective features and the K value which will represent the number of cluster required.
2. Fetch all the images from the text file and insert all the data in the 2D array called image[ID][FEATURE]. The ID is the first value in every line which represent the image_id and the feature represent the rest other value in the line which represent the color density in the image
3. After getting the image file and the K from the user, initialize the first K number of image from the text file as the cluster centroid and save the values in the 2D array called Cluster[ID][FEATURE].
4. Invoke the function KMeans by passing 7 parameters:
    A. Images[ID][FEATURE]: consists of all the images and its features
    B. NumFea : consists of count of features present in each images
    C. NumImg: consists of number of images in the text file (in this case it will be 68,040)
    D. NumCluster: specify the K value (number of cluster)
    E. Threshold: decide the end of the program
    F. Cluster_id[image_id]: specify the membership of the images
    G. Numiteration: specify the number of iteration required to get the final cluster.
5. Before starting of the first iteration the “Cluster_id[image_id]” for each image will be initialized to the value -1.
6. In the Kmean function “do-while” condition will be used were if the value (change/N) is greater than threshold then the program must stop.
7. The “change” variable value will be initialized to 0 before entering the first iteration.
8. For each image, we need to find the index or the cluster_id in which the image belongs, so cluster_index function will be invoked by passing 4 parameters:
    A. NumCluster: specify the K value
    B. NumFea : specify the number of feature in each images
    C. Images[id]: consist of single image
    D. Cluster: the cluster centroid.
9. In the cluster_index function the Euclidean distance will be determined between the single image and with the K number of cluster centroids and then the index of the cluster centroid will be returned, to which the image is near.
10. After receiving the index that image will be checked with its previous index, if the index has been changed then the “change” variable will be incremented by 1 and then the cluster_id[image_id] will be equal to the index value.
11. Then the variable newClusterSize[index] will be increment 1 if an image falls in that cluster and then the images will be added to that cluster by adding the image value to the “newcluster[index][Feature]”.

12. Then using the variable newClusterSize and the newCluster we can find the mean of all the data-point in the respective cluster and then the mean value which represent the new cluster centroid location will replace the old cluster centroid location in the cluster variable.
13. Then the condition will be checked, were if the value of (Changes/N > threshold) and the number of iteration is less than 500 then the program will be stopped. The new cluster centroid location will be the final cluster centroid location.

Below is the flow chart for the above explained procedure:

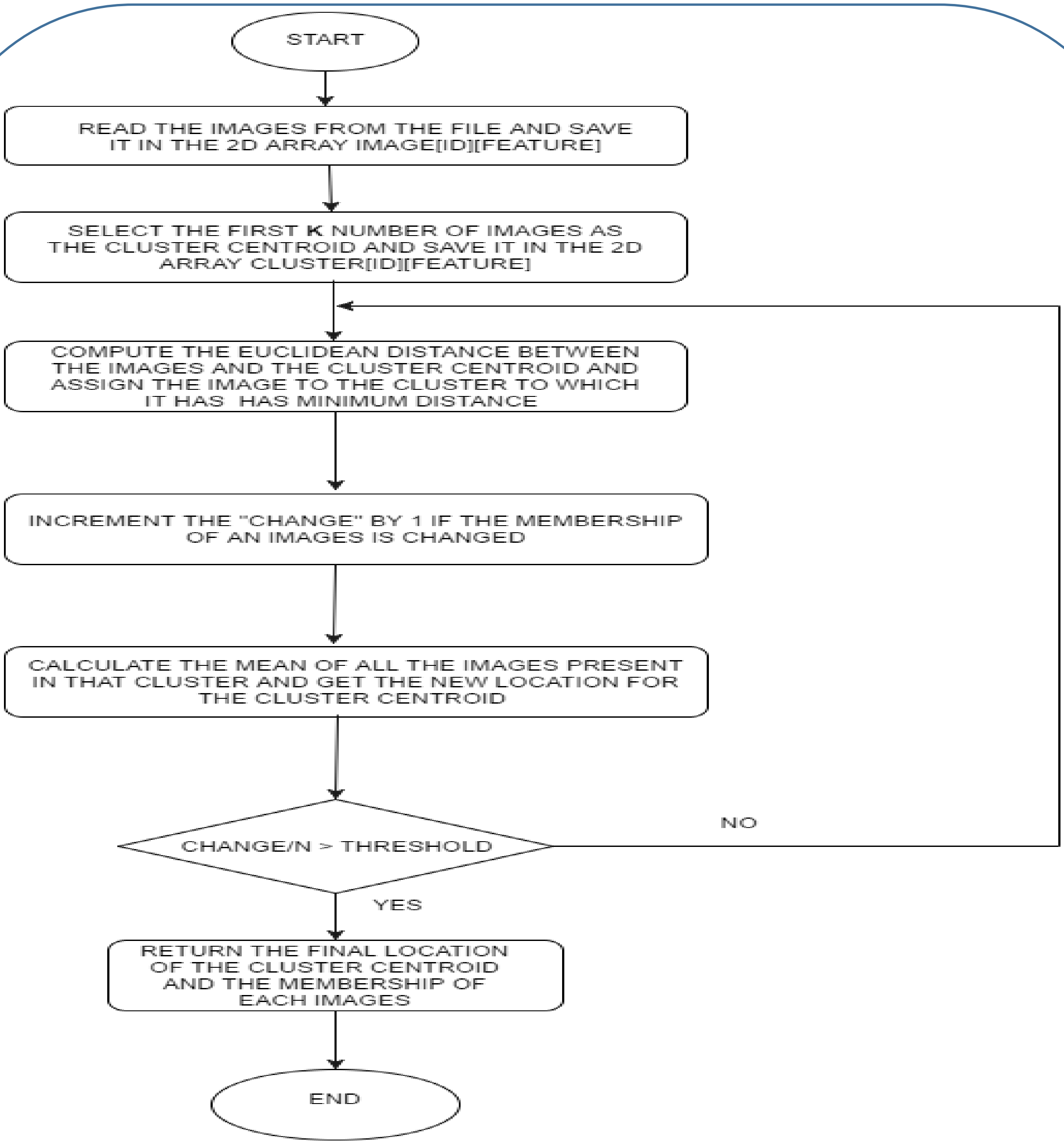

START
READ THE IMAGES FROM THE FILE AND SAVE IT IN THE 2D ARRAY IMAGE[ID][FEATURE]
SELECT THE FIRST K NUMBER OF IMAGES AS THE CLUSTER CENTROID AND SAVE IT IN THE 2D ARRAY CLUSTER[ID][FEATURE]
COMPUTE THE EUCLIDEAN DISTANCE BETWEEN THE IMAGES AND THE CLUSTER CENTROID AND ASSIGN THE IMAGE TO THE CLUSTER TO WHICH IT HAS HAS MINIMUM DISTANCE
INCREMENT THE "CHANGE" BY 1 IF THE MEMBERSHIP OF AN IMAGES IS CHANGED
CALCULATE THE MEAN OF ALL THE IMAGES PRESENT IN THAT CLUSTER AND GET THE NEW LOCATION FOR THE CLUSTER CENTROID
CHANGE/N > THRESHOLD
NO
YES
RETURN THE FINAL LOCATION OF THE CLUSTER CENTROID AND THE MEMBERSHIP OF EACH IMAGES
END

FIG 14

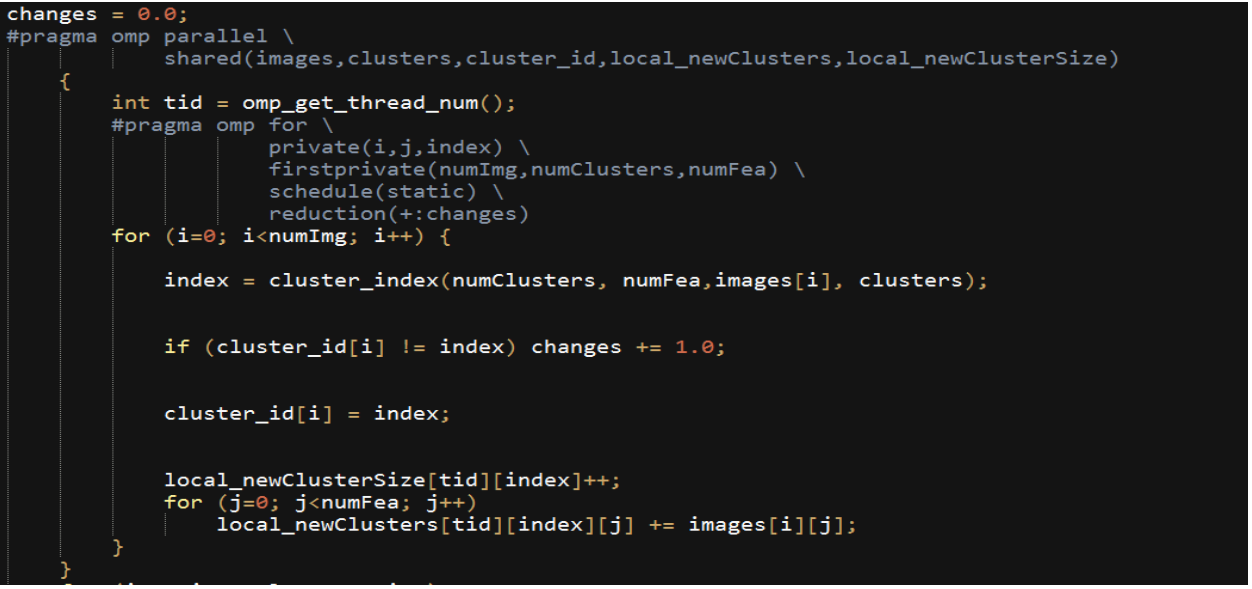
Parallel Implementation

There are two parts in the parallel implementation:

1. Parallelly accessing the images and finding their respective index value.
2. Parallelizing the calculation of the Euclidean distance between the two vector

Parallelly accessing the images:

The primary objective in this part, is parallelizing the for loop for accessing the images. For each thread an image will be assigned and that thread will be performing all the operation like finding the Euclidean distance and comparing with the other distance to get the minimum distance and returning the index value.



```
changes = 0.0;
#pragma omp parallel \
            shared(images,clusters,cluster_id,local_newClusters,local_newClusterSize)
    {
        int tid = omp_get_thread_num();
        #pragma omp for \
                    private(i,j,index) \
                    firstprivate(numImg,numClusters,numFea) \
                    schedule(static) \
                    reduction(+:changes)
        for (i=0; i<numImg; i++) {

            index = cluster_index(numClusters, numFea,images[i], clusters);


            if (cluster_id[i] != index) changes += 1.0;


            cluster_id[i] = index;


            local_newClusterSize[tid][index]++;
            for (j=0; j<numFea; j++)
                local_newClusters[tid][index][j] += images[i][j];
        }
    }
```

FIG 15

For each thread the variables i, j and the index will be local and the variables numImg, numCluster and numFea will be declared as the first private were these variables will be local to each thread but they will be assigned with their global values. And for each images the respective "change" variable will be assigned, so we have used reduction for the "change" variable with the "+" operator, so if the image changes its membership then the "change" value will be incremented and at the end of the parallel for loop, all the thread with their "change" value will be added to update to the global thread copy.

In the parallel region, we have declared two new variable local_newclusters and the local_newclustersize which is the shared variables. The local_newclusters[tid][index][j] is the 3D array were tid is the thread_id, index will be the cluster_id and the j value will be the feature present in the images and the second variable local_newclusterSize[tid][index] is the 2D array were tid is the thread_id and the index will be the cluster_id. The reason for using these two variables is to find out index for each thread_id. At the end of the first iteration, he values of local_newclusters and the local_newclustersize will be assigned to the new_cluster and the new_clusterSize variables respectively for calculating the location of the cluster centroid and then the variables local_newclusters and the local_newclustersize will be reassigned to 0 for the second iteration.

Parallelizing the calculation of the Euclidean distance

In this part, we will be paralyzing the calculation of the Euclidean distance between the two array.

Below is the pictorial representation of calculating the Euclidean distance between two array.

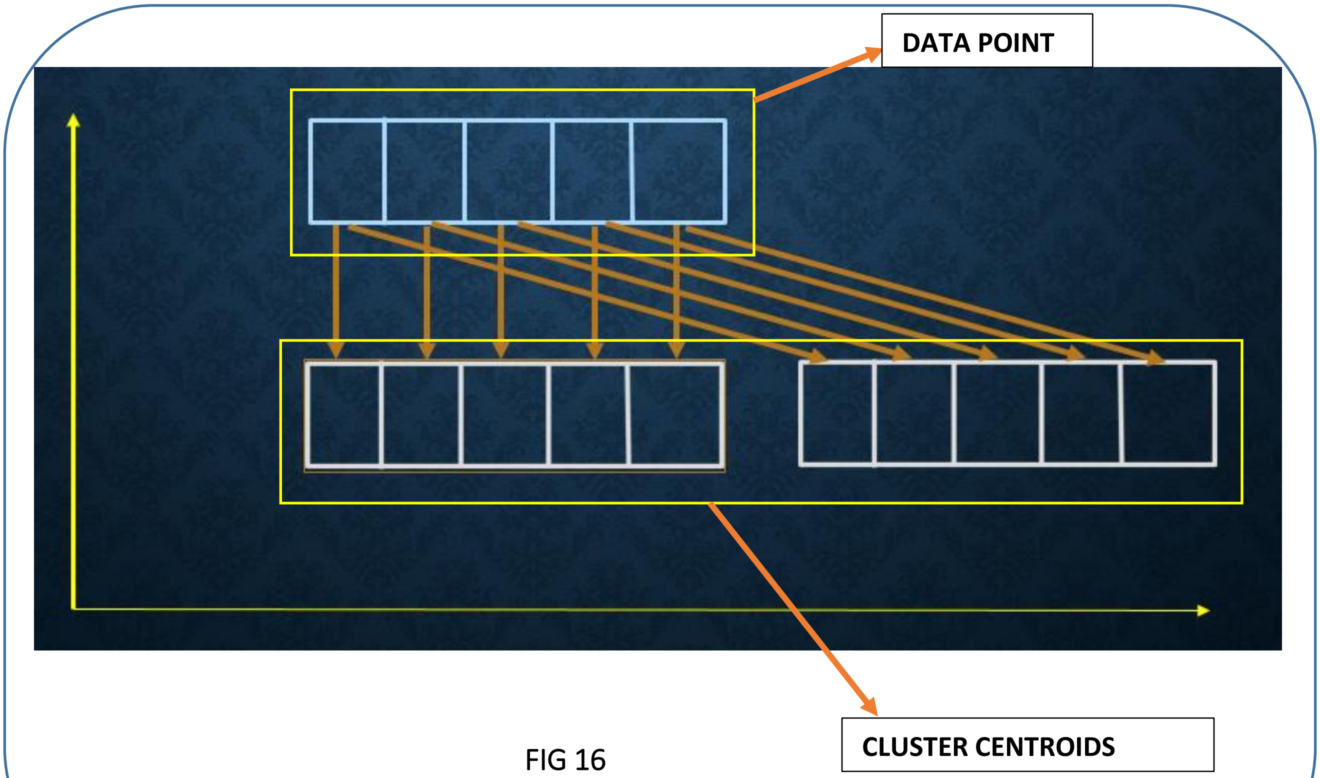
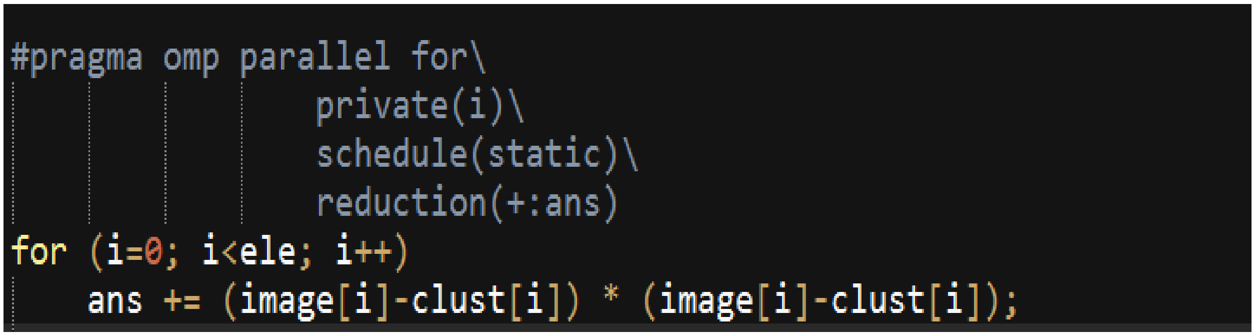


FIG 16

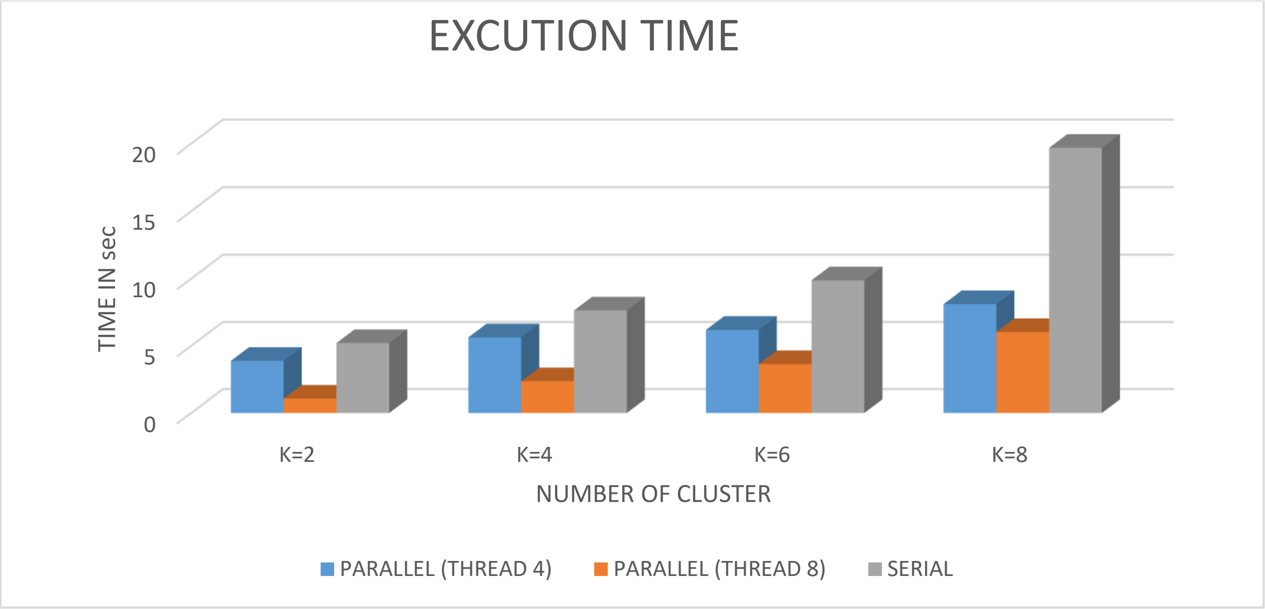


```
#pragma omp parallel for\
                private(i)\
                schedule(static)\
                reduction(+:ans)
for (i=0; i<ele; i++)
    ans += (image[i]-clust[i]) * (image[i]-clust[i]);
```

FIG 17

The above code will be paralyzing the for loop which will be calculating the Euclidean distance between the two array, In the above code the "image" variable is the data-point and the "clust" variable is the cluster centroid. The variable "i" will be local for each thread and the "ans" variable will be used for the reduction. After the end of the for loop the value of "ans" which is local to each thread will be added to update the global copy of the thread.

Results



Conclusion:

We have explored and evaluated the Naïve Bayesian classifier as part of supervised learning. We have implemented a parallelized version of the classifier to understand the speedup that it provides. We have also explored the different data and learning paradigms that can possibly present challenges to the parallelization of the classifier as well as possible solutions to these problems and their impact.

There was a lot of issue while implementing the parallel part of the K-Means algorithm. Introduction of two new variables that is local_newcluster and the local_newclsuterSize, helped to assign the image to the right cluster_id. As we keep on increasing the number of thread the performance kept on increasing and showed a lot of improvement compared to the serial part of the implementation.

References:

[1] Tapas Kanungo, David M. Mount, Nathan S. Netanyahu, Christine D. Piatko, Ruth Silverman, and Angela Y. Wu. 2002. *An Efficient k-Means Clustering Algorithm: Analysis and Implementation*. IEEE Trans. Pattern Anal. Mach. Intell. 24, 7 (July 2002), 881-892

[2]https://archive.ics.uci.edu/ml/machine-learning-databases/CorelFeatures-mld/CorelFeatures.data.html

Appendix A Source Code

Serial Non-Incremental Naïve Bayesian

```
#include <iostream>
#include <fstream>
#include <string>
#include <cstring>
#include <sstream>
#include <fstream>
#include <array>
#include <cstdlib>
#include <vector>
#include <omp.h>
#include <time.h>
#define MAX 10000
using namespace std;
string filename="largeData.csv";
string testfile="largeData.csv";
const int attrcount=5;
const int rowcount=28000;
int hashcount[attrcount];
string data[rowcount][attrcount];
string testData[rowcount][attrcount];
string countlabels[MAX][attrcount];
float counts[MAX][attrcount];
int entrycounters[attrcount];
int classcount=2;
string classes[2]={"yes","no"};
void readFileToData()
{
    char temp[1024];
    strcpy(temp,filename.c_str());
    std::ifstream file(temp);
        for(int row = 0; row < rowcount; ++row)
        {
        std::string line;
        std::getline(file, line);
        if ( !file.good() )
            break;
        std::stringstream iss(line);
        for (int col = 0; col < attrcount; ++col)
        {
            std::string val;
            std::getline(iss, val, ',');
            if ( !iss.good() )
                break;
            std::stringstream convertor(val);
            convertor >> data[row][col];
        }
        }
}
void readTestFileToData()
{
    char temp[1024];
    strcpy(temp,testfile.c_str());
    std::ifstream file(temp);
        for(int row = 0; row < rowcount; ++row)
        {
        std::string line;
        std::getline(file, line);
        if ( !file.good() )
            break;
        std::stringstream iss(line);
        for (int col = 0; col < attrcount; ++col)
        {
            std::string val;
            std::getline(iss, val, ',');
            if ( !iss.good() )
                break;
            std::stringstream convertor(val);
            convertor >> testData[row][col];
        }
        }
}
void printData()
{
    for(int row=0;row<rowcount;row++)
    {
        cout<<"\n";
        for(int col=0;col<attrcount;col++)
        {
            cout<<data[row][col]<<"\t";
        }
    }
    cout<<"\n";
```

```
}

void printTestData()
{
    for(int row=0;row<rowcount;row++)
    {
        cout<<"\n";
        for(int col=0;col<attrcount;col++)
        {
            cout<<testData[row][col]<<"\t";
        }

    }
    cout<<"\n";
}
int isInLearningModel(string key,int attr)
{
    for(int i=0;i<entrycounters[attr];i++)
    {
        if(key==countlabels[i][attr])
            return i;
    }
    return -1;
}

void insertInLearningModel(string key, int attr)
{
    int index=isInLearningModel(key,attr);
    if(index>=0)
    {
        counts[index][attr]=counts[index][attr]+1;
    }
    else
    {
        countlabels[entrycounters[attr]][attr]=key;
            counts[entrycounters[attr]][attr]=1;
        entrycounters[attr]=entrycounters[attr]+1;
    }
}
```

```
void split(const std::string &s, char delim, std::vector<std::string> &elems) {
    std::stringstream ss;
    ss.str(s);
    std::string item;
    while (std::getline(ss, item, delim)) {
        elems.push_back(item);
    }
}
std::vector<std::string> split(const std::string &s, char delim) {
    std::vector<std::string> elems;
    split(s, delim, elems);
    return elems;
}
void learn()
{
    for(int row=0;row<rowcount;row++)
    {
        for(int col=0;col<attrcount;col++)
        {
                if(data[row][col]=="")
                        continue;
            insertInLearningModel(data[row][col]+","+data[row][attrcount-1],col);
        }
    }
//add barrier
    for(int i=0;i<attrcount;i++)
    {
        for(int j=0;j<entrycounters[i];j++)
        {
                        if(i==attrcount-1)
                        {
                                    counts[j][i]=(float)counts[j][i]/(float)rowcount;
                                    continue;
                        }

            counts[j][i]=(float)counts[j][i]/(float)(counts[isInLearningModel(split(countlabels[j][i],',')
[1]+","+split(countlabels[j][i],',')[1],attrcount-1)][attrcount-1]);
        }
    }
}
```

```
int getComboIndex(string combo,int attrind)
{
        for(int i=0;i<entrycounters[attrind];i++)
        {
                if(countlabels[i][attrind]==combo)
                        return i;
        }
        return -1;
}

int getClassIndex(string classname)
{
        for(int i=0;i<classcount;i++)
        {
                if(classname==classes[i])
                        return i;
        }
        return -1;
}
string maxProbClass(double arr[])
{
        double max=-1;
        int maxind=-1;
        for(int i=0;i<classcount;i++)
        {
                if(arr[i]>max)
                {
                        max=arr[i];
                        maxind=i;
                }
        }
        return classes[maxind];
}
void test_entire()
{
        double probs[classcount];
        double acc=0;
        for(int row=0;row<rowcount;row++)
        {
                //initiallize class prob array to 1
                for(int i=0;i<classcount;i++)
                {
                        probs[i]=0;
```

```
            }
            for(int col=0;col<attrcount;col++)
            {

                  for(int classind=0;classind<classcount;classind++)
                  {
                        if(probs[classind]==0)

      probs[classind]=counts[col][getComboIndex(testData[row][col]+","+classes[classind],col
)];
                        else

      probs[classind]=probs[classind]*counts[col][getComboIndex(testData[row][col]+","+cla
sses[classind],col)];
                  }
            }
            if(testData[row][attrcount-1]==maxProbClass(probs))
                  acc++;
      }
      cout<<"Accuracy="<<acc/rowcount*100<<"%\n";
}

int main()
{
  double runtime;
  readFileToData();
  runtime = omp_get_wtime();
  learn();
  runtime = omp_get_wtime() - runtime;
  cout<< "Learning runs in "  << runtime << " seconds\n";
  readTestFileToData();
   runtime = omp_get_wtime();
  test_entire();
  runtime = omp_get_wtime() - runtime;
  cout<< "Testing runs in "  << runtime << " seconds\n";
  return 0;
}
```

Serial Naïve Bayesian with Incremental learning

```
#include <iostream>
#include <fstream>
#include <string>
#include <cstring>
#include <sstream>
#include <fstream>
#include <array>
#include <cstdlib>
#include <vector>
#include <omp.h>
#include <time.h>
#define MAX 10000
using namespace std;
string filename="largeData.csv";
string testfile="largeData.csv";
const int attrcount=5;
const int rowcount=28000;
const int testrows=28000;
int hashcount[attrcount];
string data[rowcount][attrcount];
string testData[rowcount][attrcount];
string countlabels[MAX][attrcount];
float counts[MAX][attrcount];
float probabilities[MAX][attrcount];
int entrycounters[attrcount];
int classcount=2;
string classes[2]={"yes","no"};
void readFileToData()
{

    char temp[1024];
    strcpy(temp,filename.c_str());
    std::ifstream file(temp);

        for(int row = 0; row < rowcount; ++row)
        {
        std::string line;
        std::getline(file, line);
        if ( !file.good() )
            break;
        std::stringstream iss(line);

        for (int col = 0; col < attrcount; ++col)
        {

            std::string val;
            std::getline(iss, val, ',');
            if ( !iss.good() )
                break;

            std::stringstream convertor(val);
            convertor >> data[row][col];
        }
        }

}

void readTestFileToData()
{

    char temp[1024];
    strcpy(temp,testfile.c_str());
    std::ifstream file(temp);

        for(int row = 0; row < rowcount; ++row)
        {
        std::string line;
        std::getline(file, line);
        if ( !file.good() )
            break;

        std::stringstream iss(line);

        for (int col = 0; col < attrcount; ++col)
        {

            std::string val;
            std::getline(iss, val, ',');
            if ( !iss.good() )
                break;

            std::stringstream convertor(val);
            convertor >> testData[row][col];
        }
        }

}

void printData()
{
    for(int row=0;row<rowcount;row++)
    {
        cout<<"\n";
        for(int col=0;col<attrcount;col++)
        {
            cout<<data[row][col]<<"\t";
        }

    }
    cout<<"\n";
}

void printTestData()
{
    for(int row=0;row<rowcount;row++)
    {
        cout<<"\n";
        for(int col=0;col<attrcount;col++)
        {
            cout<<testData[row][col]<<"\t";
        }

    }
    cout<<"\n";
}
int isInLearningModel(string key,int attr)
{
    for(int i=0;i<entrycounters[attr];i++)
    {
        if(key==countlabels[i][attr])
            return i;
    }
    return -1;
}

void insertInLearningModel(string key, int attr)
```

```
{
    int index=isInLearningModel(key,attr);
    if(index>=0)
    {
        counts[index][attr]=counts[index][attr]+1;
    }
    else
    {
        countlabels[entrycounters[attr]][attr]=key;
            counts[entrycounters[attr]][attr]=1;
        entrycounters[attr]=entrycounters[attr]+1;
    }
}

void split(const std::string &s, char delim, std::vector<std::string> &elems) {
    std::stringstream ss;
    ss.str(s);
    std::string item;
    while (std::getline(ss, item, delim)) {
        elems.push_back(item);
    }
}


std::vector<std::string> split(const std::string &s, char delim) {
    std::vector<std::string> elems;
    split(s, delim, elems);
    return elems;
}
void calculateProbabilities(){
            for(int i=0;i<attrcount;i++)
    {
        for(int j=0;j<entrycounters[i];j++)
        {
                        if(i==attrcount-1)
                        {
                                    probabilities[j][i]=(float)counts[j][i]/(float)rowcount;
                                    continue;
                        }

            probabilities[j][i]=(float)counts[j][i]/(float)(counts[isInLearningModel(split(countlabels[j][i],',')[1]+","+split(countlabels[j][i],',')[1],attrcount-1)][attrcount-1]);
        }
```

```
  }

}
void learn()
{
  for(int row=0;row<rowcount;row++)
  {
    for(int col=0;col<attrcount;col++)
    {
        if(data[row][col]=="")
            continue;
      insertInLearningModel(data[row][col]+","+data[row][attrcount-1],col);
    }
  }
//add barrier
  calculateProbabilities();
}
int getComboIndex(string combo,int attrind)
{
      for(int i=0;i<entrycounters[attrind];i++)
      {
            if(countlabels[i][attrind]==combo)
                  return i;
      }
      return -1;
}

int getClassIndex(string classname)
{
      for(int i=0;i<classcount;i++)
      {
            if(classname==classes[i])
                  return i;
      }
      return -1;
}
string maxProbClass(double arr[])
{
      double max=-1;
      int maxind=-1;
      for(int i=0;i<classcount;i++)
      {
            if(arr[i]>max)
        {
            max=arr[i];
            maxind=i;
        }
    }
    return classes[maxind];
}
void test_entire()
{
    double probs[classcount];
    double acc=0;
    for(int row=0;row<testrows;row++)
    {
        //initiallize class prob array to 1
        for(int i=0;i<classcount;i++)
        {
            probs[i]=0;
        }
        for(int col=0;col<attrcount;col++)
        {
            for(int classind=0;classind<classcount;classind++)
            {
                if(probs[classind]==0)

    probs[classind]=probabilities[col][getComboIndex(testData[row][col]+","+classes[classi
nd],col)];
                else

    probs[classind]=probs[classind]*probabilities[col][getComboIndex(testData[row][col]+",
"+classes[classind],col)];
            }

        }
        string maxClass=maxProbClass(probs);
        if(testData[row][attrcount-1]==maxClass)
            acc++;

            for(int col=0;col<attrcount;col++)
       {
             if(testData[row][col]=="")
                continue;
            insertInLearningModel(testData[row][col]+","+maxClass,col);
       }
```

```
                calculateProbabilities();
        }
        cout<<"Accuracy="<<acc/rowcount*100<<"%\n";
}

int main()
{
    double runtime;
    readFileToData();
    runtime = omp_get_wtime();
    learn();
    runtime = omp_get_wtime() - runtime;
    cout<< "Learning runs in "  << runtime << " seconds\n";
    readTestFileToData();
     runtime = omp_get_wtime();
    test_entire();
    runtime = omp_get_wtime() - runtime;
    cout<< "Testing runs in "  << runtime << " seconds\n";
    return 0;
}
```

Parallel Naïve Bayesian

```
#include <iostream>
#include <fstream>
#include <string>
#include <cstring>
#include <sstream>
#include <fstream>
#include <array>
#include <cstdlib>
#include <vector>
#include <omp.h>
#include <time.h>
#define MAX 10000
using namespace std;
string filename="largeData.csv";
string testfile="largeData.csv";
const int attrcount=5;
const int rowcount=28000;
int hashcount[attrcount];
string data[rowcount][attrcount];
string testData[rowcount][attrcount];
string countlabels[MAX][attrcount];
float counts[MAX][attrcount];
int entrycounters[attrcount];
int classcount=2;
int numThreads=32;
string classes[2]={"yes","no"};
void readFileToData()
{

    char temp[1024];
    strcpy(temp,filename.c_str());
    std::ifstream file(temp);

        for(int row = 0; row < rowcount; ++row)
        {
        std::string line;
        std::getline(file, line);
        if ( !file.good() )
            break;

        std::stringstream iss(line);
        for (int col = 0; col < attrcount; ++col)
        {

            std::string val;
            std::getline(iss, val, ',');
            if ( !iss.good() )
                break;

            std::stringstream convertor(val);
            convertor >> data[row][col];
        }
        }

}

void readTestFileToData()
{

    char temp[1024];
    strcpy(temp,testfile.c_str());
    std::ifstream file(temp);

        for(int row = 0; row < rowcount; ++row)
        {
        std::string line;
        std::getline(file, line);
        if ( !file.good() )
            break;

        std::stringstream iss(line);

        for (int col = 0; col < attrcount; ++col)
        {

            std::string val;
            std::getline(iss, val, ',');
            if ( !iss.good() )
                break;

            std::stringstream convertor(val);
            convertor >> testData[row][col];
        }
        }

}

void printData()
{
    for(int row=0;row<rowcount;row++)
    {
        cout<<"\n";
        for(int col=0;col<attrcount;col++)
        {
            cout<<data[row][col]<<"\t";
        }

    }
    cout<<"\n";
}

void printTestData()
{
    for(int row=0;row<rowcount;row++)
    {
        cout<<"\n";
        for(int col=0;col<attrcount;col++)
        {
            cout<<testData[row][col]<<"\t";
        }

    }
    cout<<"\n";
}
int isInLearningModel(string key,int attr)
{
    for(int i=0;i<entrycounters[attr];i++)
    {
        if(key==countlabels[i][attr])
            return i;
    }
    return -1;
}

void insertInLearningModel(string key, int attr)
{
    int index=isInLearningModel(key,attr);
    if(index>=0)
    {
        counts[index][attr]=counts[index][attr]+1;
    }
    else
    {
        countlabels[entrycounters[attr]][attr]=key;
    counts[entrycounters[attr]][attr]=1;
        entrycounters[attr]=entrycounters[attr]+1;
    }
}

void split(const std::string &s, char delim, std::vector<std::string> &elems) {
    std::stringstream ss;
    ss.str(s);
    std::string item;
    while (std::getline(ss, item, delim)) {
        elems.push_back(item);
    }
}


std::vector<std::string> split(const std::string &s, char delim) {
    std::vector<std::string> elems;
    split(s, delim, elems);
    return elems;
}
void learn(string data[][attrcount])
{
    int row,col;
    #pragma omp parallel private(row,col) shared(counts,data)
    #pragma omp for schedule(static)
    for(col=0;col<attrcount;col++)
        {
        for(row=0;row<rowcount;row++)
        {
                if(data[row][col]=="")
                continue;
                insertInLearningModel(data[row][col]+","+data[row][attrcount-1],col);
        }
    }
    int i,j;
    for( i=0;i<attrcount;i++)
    {
        for( j=0;j<entrycounters[i];j++)
        {
        if(i==attrcount-1)
        {
            counts[j][i]=(float)counts[j][i]/(float)rowcount;
            continue;
        }


counts[j][i]=(float)counts[j][i]/(float)(counts[isInLearningModel(split(countlabels[j][i],',')[1]+","
+split(countlabels[j][i],',')[1],attrcount-1)][attrcount-1]);
        }

    }
}
int getComboIndex(string combo,int attrind)
{
    for(int i=0;i<entrycounters[attrind];i++)
    {
        if(countlabels[i][attrind]==combo)
            return i;
    }
    return -1;
}

int getClassIndex(string classname)
{
    for(int i=0;i<classcount;i++)
    {
        if(classname==classes[i])
            return i;
    }
    return -1;
}
string maxProbClass(double arr[])
{
    double max=-1;
    int maxind=-1;
    for(int i=0;i<classcount;i++)
    {
        if(arr[i]>max)
        {
            max=arr[i];
            maxind=i;
        }
    }
    return classes[maxind];
}
void test_entire()
{
    double  probs[classcount];
    double acc=0;
    int row,col;
    #pragma omp parallel private(probs,row,col) shared(acc)
    #pragma omp for schedule(dynamic)
    for(int row=0;row<rowcount;row++)
    {
        //initiallize class prob array to 1
        for(int i=0;i<classcount;i++)
        {
            probs[i]=0;
        }
        for(int col=0;col<attrcount;col++)
        {

            for(int classind=0;classind<classcount;classind++)
            {
                if(probs[classind]==0)

probs[classind]=counts[col][getComboIndex(testData[row][col]+","+classes[classind],col)];
                else

probs[classind]=probs[classind]*counts[col][getComboIndex(testData[row][col]+","+classes[cla
ssind],col)];
            }
        }

    }

}

int main()
{
    double runtime;
    omp_set_num_threads(numThreads);
    readFileToData();
    runtime = omp_get_wtime();
    learn(data);
    runtime = omp_get_wtime() - runtime;
    cout<< "Learning runs in "  << runtime << " seconds\n";
    readTestFileToData();
    runtime = omp_get_wtime();
    test_entire();
    runtime = omp_get_wtime() - runtime;
    cout<< "Testing runs in "  << runtime << " seconds\n";

    return 0;
}
```