Parallelization of Expectation Maximization (EM) algorithm

Sanaz Mohammadjafari 500916726

December 2018

1. Introduction

Since online business and computer technology is developing so fast, large amounts of data are collected and its analysis is getting important. Many data mining algorithms suffer from scalability and increasing size of databases is effecting them. Data clustering is one of the principle data mining algorithms. Clustering organize datasets into a set of groups or clusters, based on data similarities. These similarities are measured by different distance metrics. Data clustering applications are in document categorization, customer/market segmentation and scientific data analysis [5]. Expectation Maximization (EM) is one the clustering algorithms which, leading experts from data mining community recently voted to be among top ten algorithms having most impact on data mining research [1]. Dempster et al. [2] first introduced the EM algorithm for parameter estimation in a Gaussian Mixture Model (GMM). Based on the iterative nature of EM algorithm it can be shown that it finds a solution of any desired precision in a finite number of steps. This makes EM to be able to apply to fuzzy data clustering and probabilistic data modeling for classification. Since K-means can be regarded as a special case of EM, EM can be considered as the most widespread clustering algorithm.

2. Goal

In the project, we have tried to parallelize the EM algorithm. First, we have implemented a sequential version of the EM and then after parallelizing some parts of the code, we have compared them to find out how much speed up we can achieve.

3. Related literature

Because of widespread use of EM, there are several attempts for parallelizing the algorithm, in this paper, we review some of them. In [3] Zhang et al. discusses several center-based clustering algorithms including k-means and EM. The paper explains how partitioning the data between different nodes and using a central node as an aggregator can help parallelizing the algorithm. Sufficient statistics (SS) of each partition is briefly discussed, which is collected form each node and is sent to the central unit, which is responsible for making a decision to update the model. In [4] the parallelization procedure is graphically described in Figure 1, in a first level approach, a distribution of the processing by column (samples) is done in the Expectation step of EM.

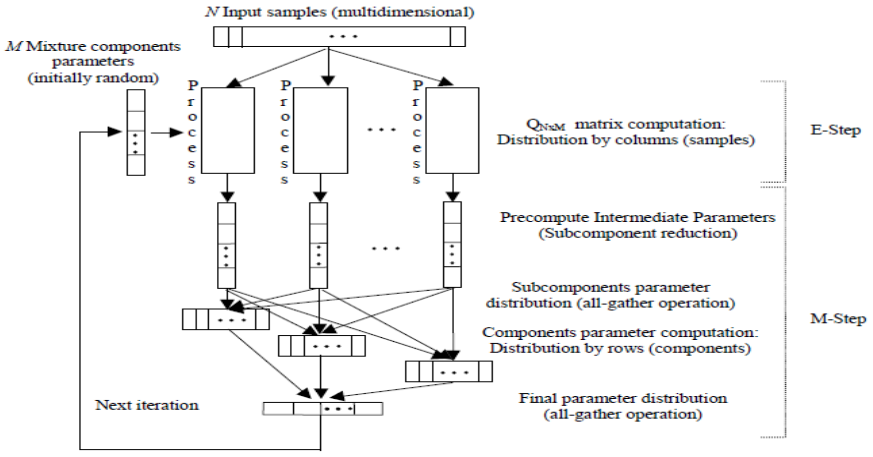

Figure 1: parallel implementation of EM for mixture models [4]

In the second level, in the M-step of the algorithm, after producing the results of first approach and distributing them between processes, a new data partition is done by rows to compute the final components of mixture model. The experimental results of this approach are gained by using a computer vision example.

In [5], authors are investigating a strategy to asynchronously update the model and accelerate the convergence of the EM algorithm. This approach is suitable for distributed environments. In their proposed model, different agents run local instances of EM with immediate model updates on their data in parallel and collect the results. When it is necessary, one agent performs a global model update and send the results to all the agents.

4. Method

In this part we first introduce the original sequential form of the EM algorithm and then discuss the parallelizable part of the code.

4.1. Classical EM

The conventional EM clustering algorithm has two main separated phases which are executed sequentially until convergence:

- **Expectation step**: Based on the probability density function of the model clusters, objects are associated to clusters.
- **Maximization step**: Parameters of the new probability density function are recalculated based on the objects that are associated with each cluster.

Clusters are modeled by Gaussian probability density function, so we can formulate the conditional probability of an object x given a cluster C as below:

(1) $$P(x|C_k) = \frac{1}{\sqrt{(2\pi)^d.|\Sigma|}}.e^{-\frac{1}{2}(x-\mu)^T.\Sigma^{-1}.(x-\mu)}$$

Where input data $x \in D$ has n rows and d features and we are trying to find k clusters $C = \{C_1, \dots, C_k\}$. $\mu = (\mu_1, \dots, \mu_d)^T$ is the mean vector of input features for cluster k. Σ is the covariance matrix input features. We assume the covariance vector to be diagonal, meaning that features are independent. Based on this we have:

(2) $$P(x|C_k) = \prod_{1\le i\le d} N(\mu_i, \sigma_i^2, x_i) = \prod_{1\le i\le d} \frac{1}{\sqrt{2\pi.\sigma_i^2}}.e^{-\frac{(x_i-\mu_i)^2}{2\sigma_i^2}}$$

Each cluster is defined by (μ, Σ, p), where p is the prior probability of the cluster k in the overall model. Overall probability of x be generated by these k clusters is formulated as:

(3) $$P(x) = \sum_{(\mu,\Sigma,p)\epsilon C} p.P(x|(\mu, \Sigma, p))$$

According to Bayes rule, the probability density of cluster k given the data is given as:

(4) $$P(C_k|x) = \frac{p_k.P(x|C_k)}{P(x)}$$

In the maximization phase, the model parameters of each cluster is determined based on the data we have from expectation step:

(5) $$p_k = \frac{1}{n}\sum_{x\in D} P(C_k|x)$$

(6) $$\mu_i = \frac{\sum_{x \in D} x_i . P(C_k|x)}{\sum_{x \in D} P(C_k|x)} \qquad \sigma_i^2 = \frac{\sum_{x \in D} (x_i - \mu_i)^2 . P(C_k|x)}{\sum_{x \in D} P(C_k|x)} \qquad 1 \leq i \leq d$$

The objective function of the model that is used to evaluate the convergence is log-likelihood of the data D given the current model C, denoted by $LL(D, C)$ is shown below:

(7) $$LL(D, C) = \sum_{x \in D} \log(P(x))$$

The summery of the algorithm is described in Figure 2.

Algorithm EM
data set D, integer k, double ϵ: GMM $\mathcal{C} = \{C_1, ..., C_k\}$

```
//initialization
 determine initial model C = {C_1, ..., C_k} by
 randomly initializing each cluster C with a point in D:
   μ := rand(D), σ := 1.0, w := k/n;
repeat
  // E-step: estimate likelihood
    for each object x ∈ D
      for each cluster C ∈ C
        compute P(x|C), P(x) and P(C|x); (cf. Eq. 1-3)
  // M-step: update model
    compute new model C = {C_1, ..., C_k} by
    updating μ, σ and w; (cf. Eq.4 - 6)
until LL(D, C_old) − LL(D, C_new) < ε;
end EM
```

Figure 2: Simple EM algorithm pseudocode [5].

4.2. parallelized EM

In order to parallelize the classical sequential EM, we decided to parallelize the E-step part of the algorithm, since computing the variables are independent and can be executed as concurrent tasks. We have done the parallelization by partitioning the data to several parts and assigning each part to a processor. Finally we gather the result of each part and concatenate them together.

Algorithm 1 Parallelizing the E-step

Input: Data n * d, number of partitions = N

```
1: for i = 1 to N do
2:     Do the E-step
3:     concatenate the results and pass it to M-step
4: end for
```

5. Results

In the chart below, we have the sequential timing and timing for 10,5,4 number of processors. As the number of processors increase the timing decreases. But because of the overhead costs we can see that the difference between timing for different number of processors decreases.

Dimension of the data		6		
Number rows of data		10000		
Number of clusters		20		
Sequential timing (Sec)		218. 43		
Number of processors			ψ	e
Parallel	10	52.18	4.186	0.154
Parallel	5	55.59	3.929	0.068
Parallel	4	58.60	3.727	0.024

We calculated the Karp-Flatt metric for each number of processors. The formula of metric is given below:

Karp-Flatt metric:

(8) $$\psi = \frac{T(1)}{T(s)} = \frac{T(sequential)}{T(parallel)} \qquad e = \frac{\frac{1}{\psi} - \frac{1}{n}}{1 - \frac{1}{n}}$$

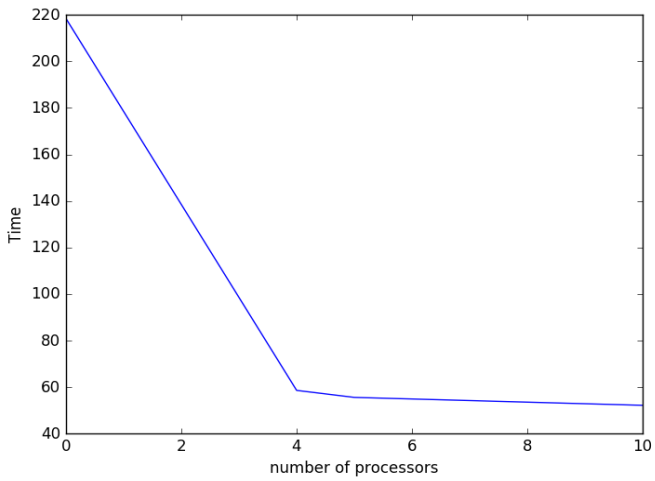

Figure 3: Time versus number of processors

6. Summary

Expectation maximization clustering is one of the most principle data mining algorithms, therefore there has been a lot of attempts to parallelize the algorithm. In this project we first explained the classical sequential EM algorithm and we considered the parallelizable parts of the code. We decided to parallelize the E-step part of code by partitioning the data into N parts and run each partition of the data in one processors an concatenate the result of each part to gather the final result of E-step. The results show significance reduction in the code running time.

7. References

[1]. Wu, Xindong, et al. "Top 10 algorithms in data mining." *Knowledge and information systems* 14.1 (2008): 1-37.

[2]. Dempster, Arthur P., Nan M. Laird, and Donald B. Rubin. "Maximum likelihood from incomplete data via the EM algorithm." *Journal of the royal statistical society. Series B (methodological)* (1977): 1-38.

[3]. Zhang, Bin, Meichun Hsu, and George Forman. "Accurate recasting of parameter estimation algorithms using sufficient statistics for efficient parallel speed-up." *European Conference on Principles of Data Mining and Knowledge Discovery*. Springer, Berlin, Heidelberg, 2000.

[4]. López-de-Teruel, Pedro E., José M. García, and Manuel E. Acacio. "The Parallel EM Algorithm and its Applications in Computer Vision." *PDPTA*. 1999.

[5]. Plant, Claudia, and Christian Bohm. "Parallel EM-clustering: Fast convergence by asynchronous model updates." *Data Mining Workshops (ICDMW), 2010 IEEE International Conference on*. IEEE, 2010.

Code Listing:

Sequential version

```
import numpy as np
import random
import E_sequential, readingData,testing,EM_parallelize
import time

filename="C:/Users/sami.rodrigue/Desktop/Ryerson/Algorithm/Data3.txt"
nclust = 20
maxiter = 5
epsilon = 0.01

X= readingData.dataPrep(filename)
n = len(X)
d = len(X[0])
# print(n,d)
#initialization
clusters =[]
mu = []
sigma=[]
W =[]
for i in range(nclust):
    clusters.append(X[random.randint(0,len(X)-1)])
    mu.append(X[random.randint(0,len(X)-1)])
    sigma.append(np.identity(d))
```

```
    W.append(nclust/n)
W1 = nclust/n
# print(sigma)
Go =True
start_time = time.time()
for itr in range(maxiter):
    P_x_C = []
    P_x_C_data =[]
    P_C_X_data =[]
    P_C_x =[]
    P_x =[]
    P_x_data=[]
    for index,data in enumerate(X):
        P = []
        p_x = 0
        for j in range(len(clusters)):
            p = 1
            for i in range(d):
                temp2 = sigma[j][i][i]
                temp = np.exp(-1*pow(data[i]-mu[j][i],2)/(2*temp2))
                p *= (1/(pow(2*np.pi*sigma[j][i][i],0.5)))*temp
            P.append(p)
            p_x += W[j]*p
        P_x_C.append(P)
        P_x.append(p_x)
```

```
    P_C_X_data =[]
    for j in range(len(clusters)):
        P_C_X_data.append(W[j]*P[j]/p_x)
    P_C_x.append(P_C_X_data)
    # for j in range(len(clusters)):
    #     P_C_x.append(P_x_C[j]*W[j]/P_x)

# ########## M-Step
####################
for j in range(len(clusters)):
    temp = 0
    for index,data in enumerate(X):
        temp += P_C_x[index][j]
    W[j] = temp/n

for j in range(len(clusters)):
    for i in range(d):
        temp = 0
        temp1 = 0
        for index, data in enumerate(X):
            temp += P_C_x[index][j]
            temp1 += data[i] * P_C_x[index][j]
        mu[j][i] =  temp1/temp
for j in range(len(clusters)):
    for i in range(d):
```

```
        temp = 0
        temp1 = 0
        for index, data in enumerate(X):
            temp += P_C_x[index][j]
            temp1 += pow(data[i]-mu[j][i],2) * P_C_x[index][j]
        sigma[j][i][i] = temp1/temp

# LL(C_old)
LL_old =0
LL_new = 0
for index,data in enumerate(X):
    LL_old += np.log(P_x[index])
for index,data in enumerate(X):
    P = []
    p_x = 0
    for j in range(len(clusters)):
        p = 1
        for i in range(d):
            temp2 = sigma[j][i][i]
            temp = np.exp(-1*pow(data[i]-mu[j][i],2)/(2*temp2))
            p *= (1/(pow(2*np.pi*sigma[j][i][i],0.5)))*temp
        P.append(p)
        p_x += W[j]*p
    LL_new += np.log(p_x)
```

```
    if np.abs(LL_new-LL_old) < epsilon:
        break

averageTraningTime = time.time() - start_time
print(averageTraningTime)
```

(Parallel version)

```
import numpy as np
import random
import readingData
import time
import multiprocessing

def multi_run_wrapper(args):
   return work(*args)
# def work(P_x_C, P_C_x,P_x,X,clusters,sigma,mu):
def work(X, clusters, sigma, mu,d,W):
    P_x_C = []
    P_C_x = []
    P_x = []
    for index, data in enumerate(X):
        P = []
        p_x = 0
        for j in range(len(clusters)):
            p = 1
            for i in range(d):
                temp2 = sigma[j][i][i]
                temp = np.exp(-1 * pow(data[i] - mu[j][i], 2) / (2 * temp2))
```

```
                p *= (1 / (pow(2 * np.pi * sigma[j][i][i], 0.5))) * temp
            P.append(p)
            p_x += W[j] * p
        P_x_C.append(P)
        P_x.append(p_x)
        P_C_X_data = []
        for j in range(len(clusters)):
            P_C_X_data.append(W[j] * P[j] / p_x)
        P_C_x.append(P_C_X_data)
    return P_x_C,P_C_x,P_x

if __name__ == "__main__":
    filename = "C:/Users/sami.rodrigue/Desktop/Ryerson/Algorithm/Data3.txt"
    nclust = 20
    maxiter = 5
    epsilon = 0.01

    X= readingData.dataPrep(filename)
    n = len(X)
    d = len(X[0])
    # print(n,d)
    #initialization
    clusters =[]
    mu = []
    sigma=[]
```

```
W =[]
for i in range(nclust):
    clusters.append(X[random.randint(0,len(X)-1)])
    mu.append(X[random.randint(0,len(X)-1)])
    sigma.append(np.identity(d))
    W.append(nclust/n)
W1 = nclust/n
# print(sigma)
# Go =True
start_time = time.time()
for itr in range(maxiter):
    Result= []
    # P_x_C_array = []
    # P_C_x_array =[]
    # P_x_array =[]
    parts = 10
    N = int(n/parts)

    args = []
    # for k in range(5):
    #     P_x_C.append(multiprocessing.Array('i', N))
    #     P_C_x_.append(multiprocessing.Array('i', N))
    #     P_x_array.append(multiprocessing.Array('i', N))
    for k in range(parts):
        data = X[k*N: (k+1)*N]
```

```
    args.append((data,clusters,sigma,mu,d,W))

p = multiprocessing.Pool(parts)
Result =  p.map(multi_run_wrapper, args)
P_x_C = []
P_C_x = []
P_x = []
for k in range(parts):
    P_x_C += (Result[k][0])
    P_C_x +=(Result[k][1])
    P_x += (Result[k][2])
    # for j in range(len(clusters)):
    #     P_C_x.append(P_x_C[j]*W[j]/P_x)

# ########## M-Step
####################
for j in range(len(clusters)):
    temp = 0
    for index,data in enumerate(X):
        temp += P_C_x[index][j]
    W[j] = temp/n

for j in range(len(clusters)):
    for i in range(d):
        temp = 0
```

```
        temp1 = 0
        for index, data in enumerate(X):
            temp += P_C_x[index][j]
            temp1 += data[i] * P_C_x[index][j]
        mu[j][i] =  temp1/temp
for j in range(len(clusters)):
    for i in range(d):
        temp = 0
        temp1 = 0
        for index, data in enumerate(X):
            temp += P_C_x[index][j]
            temp1 += pow(data[i]-mu[j][i],2) * P_C_x[index][j]
        sigma[j][i][i] = temp1/temp

# LL(C_old)
LL_old =0
LL_new = 0
for index,data in enumerate(X):
    LL_old += np.log(P_x[index])
for index,data in enumerate(X):
    P = []
    p_x = 0
    for j in range(len(clusters)):
        p = 1
        for i in range(d):
```

```
            temp2 = sigma[j][i][i]
            temp = np.exp(-1*pow(data[i]-mu[j][i],2)/(2*temp2))
            p *= (1/(pow(2*np.pi*sigma[j][i][i],0.5)))*temp
        P.append(p)
        p_x += W[j]*p
    LL_new += np.log(p_x)

  if np.abs(LL_new-LL_old) < epsilon:
    break

averageTraningTime = time.time() - start_time
print(averageTraningTime)
```