SDS - Homework 2

Group 00: Michele Meo, Lorenzo Ceccomancini

Whack a MoG

EM algorithm generalization

We generalized the EM algorithm proposed in handmade code from lecture on gaussian mixture, in order to estimate the set of parameters $\theta = (p, \mu, \sigma)$ for each Gaussian in a K Gaussians mixture model.

First of all, given the set of parameters for a specific MoG model and the data, we need a function to evaluate the likelihood at each step of the EM algorithm.

input: X data, k, p, μ, σ Gaussians number, weights, mean, variance

output: L vector of Likelihood values on each datapoint

```
# Function to evaluate the likelihood at each step of EM algo
likelihood = function(X, k, p, mu, sigma){

  L = c(rep(0, length(X)))

  for(c in 1:k){
    L = L + p[c]*dnorm(X, mu[c], sigma[c])
  }

  return(L)
}
```

In order to get the responsibilities vector for each Gaussian in each model, we perform the E-step of the algorithm in the following function.

input: X data, k, p, μ, σ Gaussians number, weights, mean, variance

output: r vector of responsibilities for each Gaussian, given a specific model

```
# Function to evaluate responsibilities of each Normal component
E_step = function(X, k, p, mu, sigma){

  d = matrix(NA, k, length(X))

  for(c in 1:k){
    d[c,] = p[c]*dnorm(X, mu[c], sigma[c])
  }

  r = matrix(NA, k, length(X))

  for(c in 1:k){
    r[c,] = d[c,]/colSums(d)
  }

  return(r)
}
```

Instead, we use the following function to generalize the DataFrame structure to a generic number k of Gaussian components.

```
# Function to names parameters in DataFrame with variable k
names_parameters = function(k){
  parameters = c()
  parameters[1] = "iteration"
  parameter_names = c("p", "mu", "sigma")

  i = 2

  for(par in parameter_names){
    for(c in as.character(1:k)){
      parameters[i] = paste(par, c, sep="")
      i = i+1
    }
  }

  parameters[i] = "deviance"

  return(parameters)
}
```

Finally, we implement the generalized EM algorithm. The number of steps are not fixed but they depends on the procedure convergence: in particular, we decided to introduce a threshold t needed to check the objective function convergence, the deviance in this case. When the absolute difference between the deviance evaluated at the previous step and the actual step is under the threshold, we stop the procedure and we get the last inferred values of (p, μ, σ) as Gaussian parameter estimators.

Since our code had an overflow error, we decided to exclude the possibility of an over-convergence problem by adding the threshold t: since defining a convergence criterion linked to the threshold did not solve these problems, we decided to leave a small constant of 0.1 in the estimator denominators of the μ and σ parameters, in order to avoid this error.

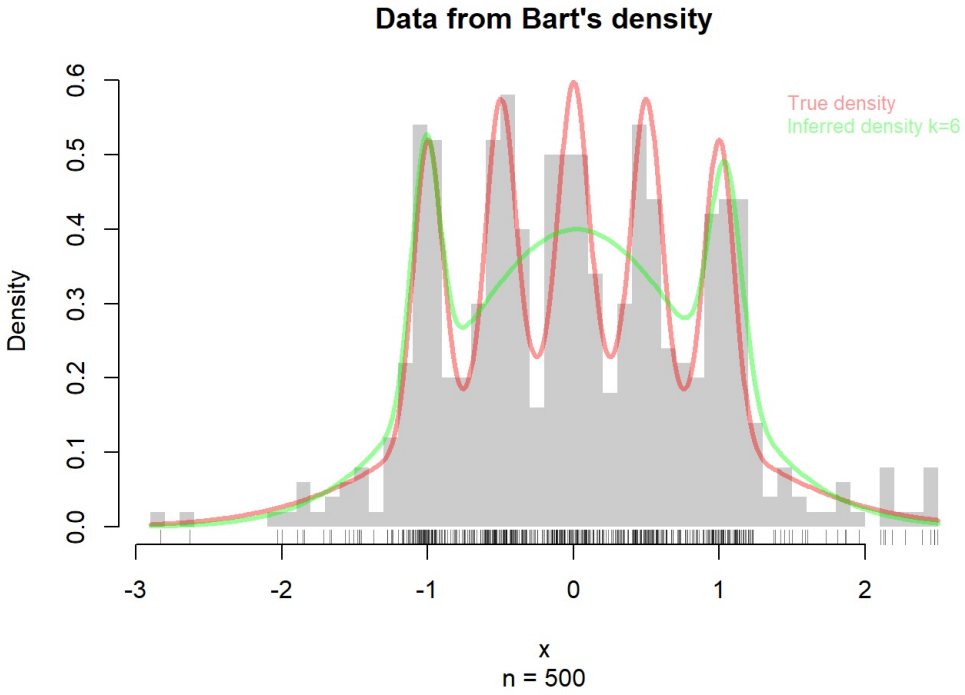
input: X data, k, p, μ, σ Gaussians number, weights, mean, variance vectors, t threshold for convergence

output: a DataFrame containing all parameters, deviance for each iteration

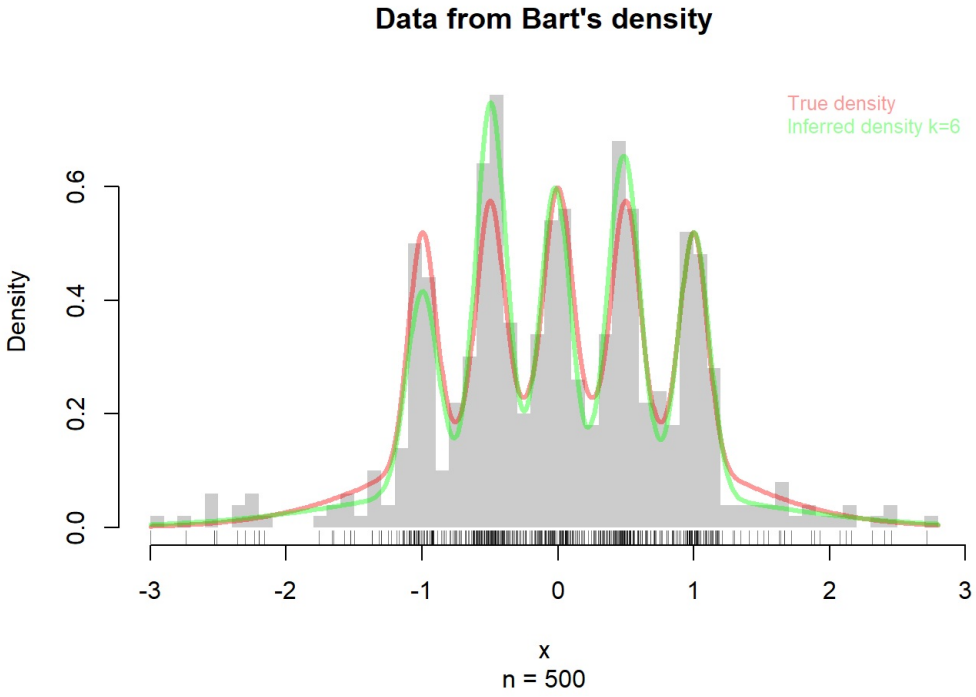
```
# EM algorithm function
EM_algo = function(X, k, p, mu, sigma, t){

  res = matrix(NA, nrow = 1, ncol = 3*k + 2)

  like = likelihood(X, k, p, mu, sigma)
  deviance = -2*sum(log(like))
  i = 1
  res[i,] = c(i, p, mu, sigma, deviance)

  while(TRUE){

    # E step
    r = E_step(X, k, p, mu, sigma)

    # M step

    for(c in 1:k){
      p[c] = mean(r[c,])
      mu[c] = sum(r[c,]*X)/(sum(r[c,])+0.1)
      sigma[c] = sqrt( sum(r[c,]*(X^2))/(sum(r[c,])+0.1)-(mu[c])^2)
    }

    old_deviance = deviance
    like = likelihood(X, k, p, mu, sigma)
    deviance = -2*sum(log(like))

    if( abs(deviance - old_deviance) < t ){
      break
    }

    res = rbind(res, c(i, p, mu, sigma, deviance))
    i = i+1
  }

  res = data.frame(res)
  names(res) = names_parameters(k)
  out = list(parameters = c(p = p, mu = mu, sigma = sigma), deviance = deviance, res = res)
  return(out)
}
```

Given that the EM algorithm strongly depends from the initialization, so from the first choice of the K Gaussians parameters (p, μ, σ), we decided to initialize the set of parameters from the true one and adding a normal variability to each of them. In particular, we have seen that the algorithm is very sensible to variation on the $\{\sigma\}$ initialization because, often, adding a variability bigger than 0.01 is enough to get an OverFlow Error.

We can see from the plot below that adding a Normal noise with mean 0 and standard deviation 4 to initialization of parameter μ for a model with $k = 6$, the EM algorithm converge to a not precise MoG.



Given the consideration above, we decided to add a small Normal noise to the initialization of the model parameters for each model selector we are going to implement. In particular, in the following plot we show the inferred density from an EM algorithm running with an applied Normal variability to parameters μ and σ, respectively, of standard deviations 0.1 and 0.01.



Model selection

Given our computational resources and after some empirical attempts, we decided to adopt $n_1 = 100$ as a clearly non-asymptotic size and $n_2 = 1000$ as a reasonably asymptotic size for sampling from the Bart distribution. We decided to fix the choice of $k_{max} = 11$, $t = 0.01$ threshold for EM algorithm convergence, $M = 100$ number of iterations of each model selector, considering our computational power and balancing the precision of each method for a future comparison among them. All the model selectors, a part for the Wasserstein based score, are scores linked to the evaluation of the likelihood function, so we are going to maximize each score in order to take the best choice of k for each sample. In the Wasserstein case, we have a loss function so we need to minimize it. Moreover, we decided to take the mean over all the M iteration for each set of k generated by each model as a good estimator predicted by that specific model.

AIC and BIC

In the following function we implement both *AIC* and *BIC* score.

input: sample_size size of dataset generated from Bart density, K_max maximum number of gaussian in a model, t convergence threshold for EM algorithm, M number of samples, type AIC or BIC flag method

output: a matrix containing all the AIC or BIC score, for each model and for each sample

```
# Implementing Akaike and BIC method

AIC_or_BIC = function(sample_size, k_max, t, M, type){

  tmp_results = matrix(NA, M, k_max)

  for(m in 1:M){
    XX = rnormmix(sample_size,
                  lambda = c(0.5, rep(0.1,5)),
                  mu     = c(0, ((0:4)/2)-1),
                  sigma  = c(1, rep(0.1,5)) )

    for(k in 1:k_max){

      if(type=="AIC"){
        reg = k*3} else{
          reg = (log(sample_size)/sample_size)*3*k}

      # parameters initialization
      p = c(rep(1/k, k))
      mu = c(0, ((0:(k-2))/2)-1)
      sigma  = c(1, rep(0.1, k-1))
      if (k == 1){
        mu = c(0)
        sigma = c(1)}

      mu = mu + rnorm(k, 0, 0.1)
      sigma = sigma + rnorm(k, 0, 0.01)

      EM = EM_algo(XX, k, p, mu, sigma, t)
      n = length(EM[["res"]][["deviance"]])
      loglike = -(EM[["res"]][["deviance"]][n])/2
      tmp = 2*(loglike - reg)
      tmp_results[m,k] = tmp
    }
    #print(m)
  }

  return(tmp_results)
}
```

We can see that AIC seems to capture the real k parameter in the reasonable asymptotic case, while it reaches an estimate very far from the real value in the non-asymptotic case.

Given the expression of the regularization factor in the AIC and BIC scores, it can be noted that AIC always penalizes the more complex models while BIC penalizes the complexity based on the sample size. For the same complexity of the model, BIC will penalize less a given model for a greater data availability.

This consideration seems confirmed by the results obtained, since in the non-asymptotic case BIC seems to be closer to the real target while in the asymptotic case AIC give a better estimation of k real value while BIC seems to favor the more complex models, giving an higher k estimate than the non-asymptotic case.

```
selection_n1_AIC = AIC_or_BIC(100, 11, 0.01, 100, "AIC")
selection_n1_BIC = AIC_or_BIC(100, 11, 0.01, 100, "BIC")
K_AIC_n1 = argmax(selection_n1_AIC)
K_BIC_n1 = argmax(selection_n1_BIC)
print("The AIC and BIC best models for not asymptotic sample are:")
```

```
## [1] "The AIC and BIC best models for not asymptotic sample are:"
```

```
print(c(K_AIC_n1 = mean(K_AIC_n1), K_BIC_n1 = mean(K_BIC_n1)))
```

```
## K_AIC_n1 K_BIC_n1
##     3.00     8.24
```

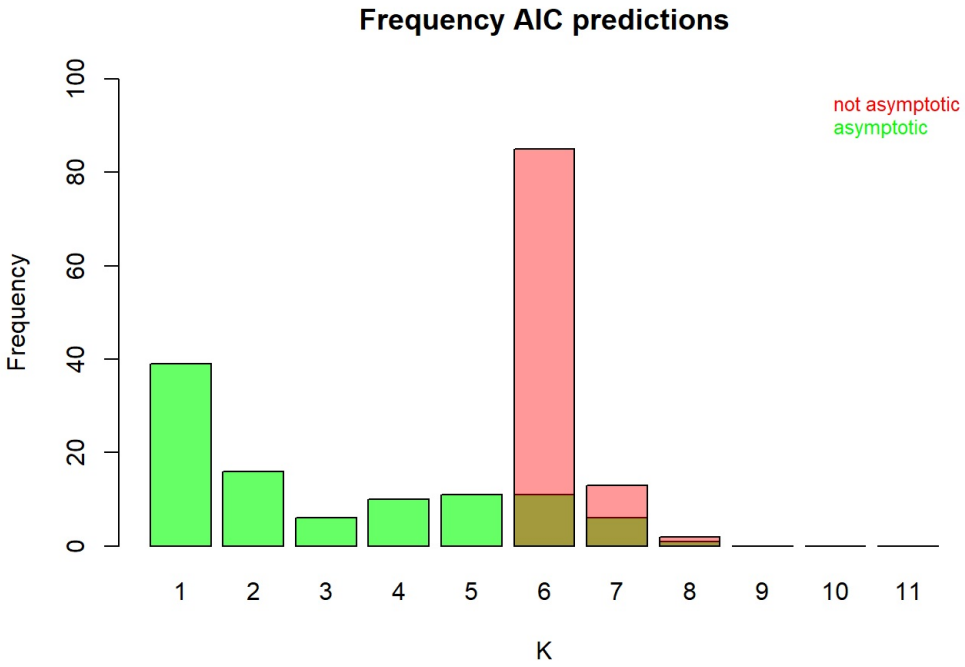
```
selection_n2_AIC = AIC_or_BIC(1000, 11, 0.01, 100, "AIC")
selection_n2_BIC = AIC_or_BIC(1000, 11, 0.01, 100, "BIC")
K_AIC_n2 = argmax(selection_n2_AIC)
K_BIC_n2 = argmax(selection_n2_BIC)
print("The AIC and BIC best models for reasonably asymptotic sample are:")
```

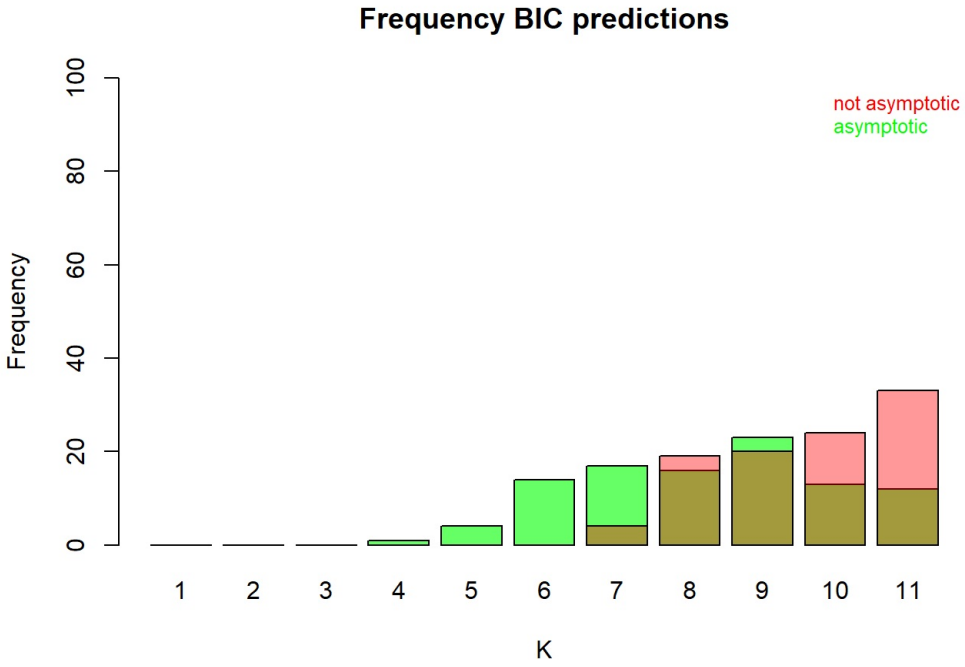
```
## [1] "The AIC and BIC best models for reasonably asymptotic sample are:"
```

```
print(c(K_AIC_n2 = mean(K_AIC_n2), K_BIC_n2 = mean(K_BIC_n2)))
```

```
## K_AIC_n2 K_BIC_n2
##     6.17     9.63
```

```
bar = function(method){
  arr = c(rep(0,11))
  for(i in method){
    arr[i] = arr[i] + 1
  }
  return(arr)
}
```





Cross-Validation

In the following function we implement Cross-Validation with different type of splitting. In particular we focus on three different proportions that can be chosen, setting type argument of the function to 1, 2 or 3 corresponding to 50-50, 70-30 or 30-70.

input: sample_size size of dataset generated from Bart density, K_max maximum number of gaussian in a model, t convergence threshold for EM algorithm, M number of samples, type 1,2 or 3.

output: a matrix containing all the Cross_Validation score, for each model and for each sample

```
CV = function(sample_size, k_max, t, M, type){

  tmp_results = matrix(NA, M, k_max)
  K_estimator = c()
  for(m in 1:M){
    XX = rnormmix(sample_size,
                  lambda = c(0.5, rep(0.1,5)),
                  mu     = c(0, ((0:4)/2)-1),
                  sigma  = c(1, rep(0.1,5)) )

    if(type==1){
      train = XX[1:(sample_size%/%2)]
      test = XX[(sample_size%/%2+1):sample_size]
    } else if(type==2){
      train = XX[1:floor(sample_size*0.7)]
      test = XX[(floor(sample_size*0.7)+1):sample_size]
    } else if(type==3){
      train = XX[1:floor(sample_size*0.3)]
      test = XX[(floor(sample_size*0.3)+1):sample_size]
    }


    for(k in 1:k_max){

      # parameters initialization
      p = c(rep(1/k, k))
      mu = c(0, ((0:(k-2))/2)-1)
      sigma  = c(1, rep(0.1, k-1))
      if (k == 1){
        mu = c(0)
        sigma = c(1)}

      mu = mu + rnorm(k, 0, 0.1)
      sigma = sigma + rnorm(k, 0, 0.01)


      EM_params = EM_algo(train, k, p, mu, sigma, t)
      n = length(EM_params[["res"]][["deviance"]])

      p = as.numeric(as.vector(EM_params[["res"]][n, 2:(k*3+1)][1:k]))
      mu = as.numeric(as.vector(EM_params[["res"]][n, 2:(k*3+1)][(k+1):(2*k)]))
      sigma = as.numeric(as.vector(EM_params[["res"]][n, 2:(k*3+1)][(2*k+1):(3*k)]))

      like = likelihood(test, k, p, mu, sigma)

      loglike = sum(log(like))
      tmp_results[m,k] = loglike
    }
    #print(m)
  }
  return(tmp_results)
}
```

As we can see the 70-30 splitting is the best proportion in the not asymptotic case and it is reasonable as the EM-Algorithm is trained on the most of the data while the score evaluation is implemented on the thirty percent. As we could expect the score improve increasing the sample size and it gives an acceptable result. We notice that in the not asymptotic case the cross validation predicts a low k while in the reasonably asymptotic case the estimate is closer to the target in the 50-50 and 30-70 sample splitting.

```
selection_n1_CV_50_50 = CV(100, 11, 0.01, 100, 1)
selection_n1_CV_70_30 = CV(100, 11, 0.01, 100, 2)
selection_n1_CV_30_70 = CV(100, 11, 0.01, 100, 3)
K_CV_50_50_n1 = argmax(selection_n1_CV_50_50)
K_CV_70_30_n1 = argmax(selection_n1_CV_70_30)
K_CV_30_70_n1 = argmax(selection_n1_CV_30_70)
print("The cros validation best model for not asymptotic sample is:")
```

```
## [1] "The cros validation best model for not asymptotic sample is:"
```

```
print(c(train_50_test_50 = mean(K_CV_50_50_n1), train_70_test_30 = mean(K_CV_70_30_n1), train_30_test_70 = mean(K
_CV_30_70_n1)))
```

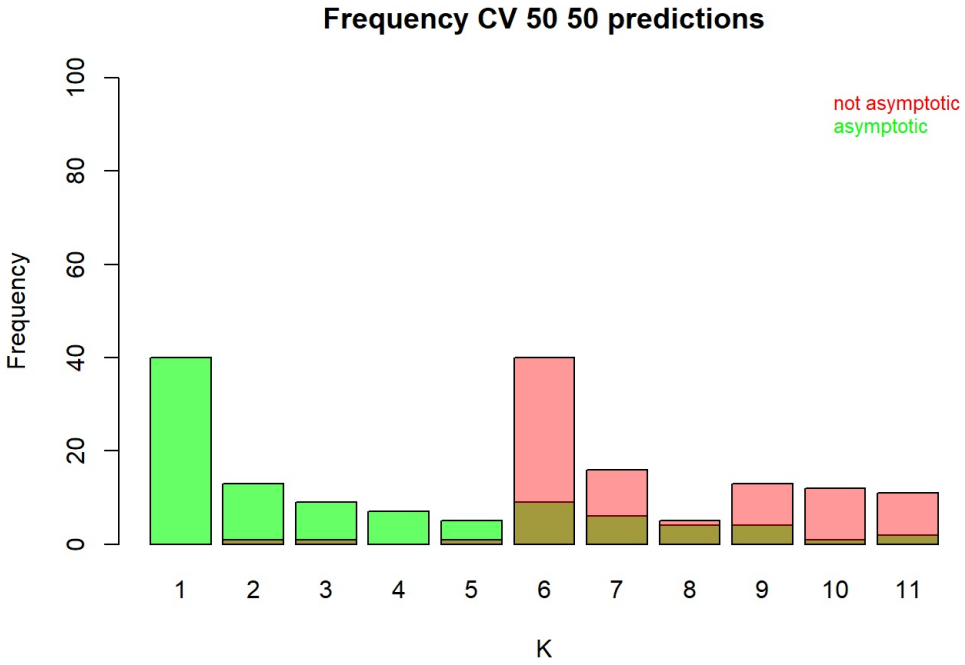
```
## train_50_test_50 train_70_test_30 train_30_test_70
##             3.42             4.51             1.99
```

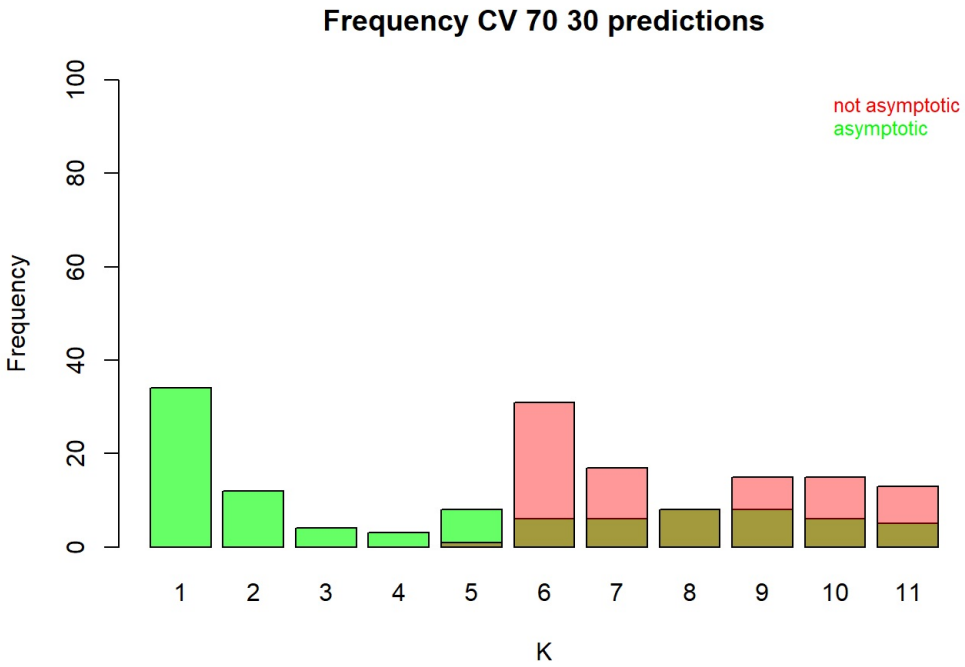
```
selection_n2_CV_50_50 = CV(1000, 11, 0.01, 100, 1)
selection_n2_CV_70_30 = CV(1000, 11, 0.01, 100, 2)
selection_n2_CV_30_70 = CV(1000, 11, 0.01, 100, 3)
K_CV_50_50_n2 = argmax(selection_n2_CV_50_50)
K_CV_70_30_n2 = argmax(selection_n2_CV_70_30)
K_CV_30_70_n2 = argmax(selection_n2_CV_30_70)
print("The cros validation best model for reasonably asymptotic sample is:")
```

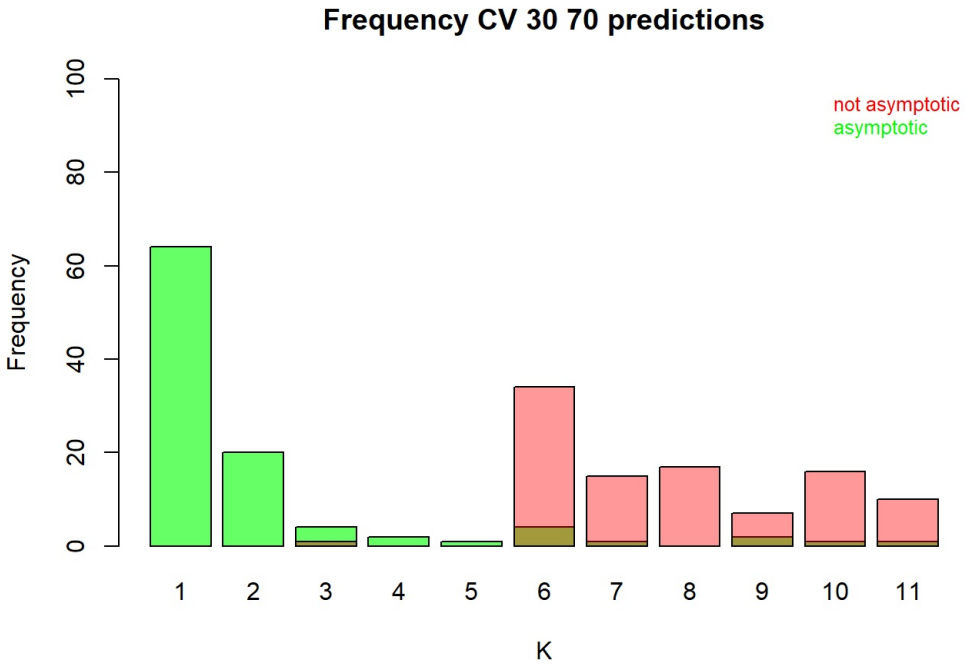
```
## [1] "The cros validation best model for reasonably asymptotic sample is:"
```

```
print(c(train_50_test_50 = mean(K_CV_50_50_n2), train_70_test_30 = mean(K_CV_70_30_n2), train_30_test_70 = mean(K
_CV_30_70_n2)))
```

```
## train_50_test_50 train_70_test_30 train_30_test_70
##             7.60             8.02             7.81
```


Frequency CV 50 50 predictions
not asymptotic
asymptotic
Frequency
0
20
40
60
80
100
1
2
3
4
5
6
7
8
9
10
11
K


Frequency CV 70 30 predictions
not asymptotic
asymptotic
Frequency
0
20
40
60
80
100
1
2
3
4
5
6
7
8
9
10
11
K


Frequency CV 30 70 predictions
not asymptotic
asymptotic
Frequency
0
20
40
60
80
100
1
2
3
4
5
6
7
8
9
10
11
K

K-folds Cross-Validation

In the following function we implement k-folds-Cross-Validation with 5 and 10 folds.

input: sample_size size of dataset generated from Bart density, K_max maximum number of gaussian in a model, t convergence threshold for EM algorithm, M number of samples, n_fold 5 or 10.

output: a matrix containing all the k-Cross_Validation score, for each model and for each sample

```
k_CV = function(sample_size, k_max, t, M, n_fold){

  final_results = matrix(NA, M, k_max)

  for(m in 1:M){
    XX = rnormmix(sample_size,
                  lambda = c(0.5, rep(0.1,5)),
                  mu     = c(0, ((0:4)/2)-1),
                  sigma  = c(1, rep(0.1,5)) )

    for(k in 1:k_max){

      tmp_results = c()

      for(n in 1:n_fold){

        fold_size = floor(sample_size/n_fold)

        test = XX[((n-1)*fold_size+1):(n*fold_size)]
        if (n==1){
          train = XX[(fold_size+1):sample_size]
        } else if (n == n_fold){
          train = XX[1:((n-1)*fold_size)]
        } else{
            train = XX[c(1:((n-1)*fold_size),((n*fold_size)+1):sample_size)]
        }


        # parameters initialization
        p = c(rep(1/k, k))
        mu = c(0, ((0:(k-2))/2)-1)
        sigma  = c(1, rep(0.1, k-1))
        if (k == 1){
          mu = c(0)
          sigma = c(1)}

        mu = mu + rnorm(k, 0, 0.1)
        sigma = sigma + rnorm(k, 0, 0.01)


        EM_params = EM_algo(train, k, p, mu, sigma, t)
        len = length(EM_params[["res"]][["deviance"]])

        p = as.numeric(as.vector(EM_params[["res"]][len, 2:(k*3+1)][1:k]))
        mu = as.numeric(as.vector(EM_params[["res"]][len, 2:(k*3+1)][(k+1):(2*k)]))
        sigma = as.numeric(as.vector(EM_params[["res"]][len, 2:(k*3+1)][(2*k+1):(3*k)]))

        like = likelihood(test, k, p, mu, sigma)

        loglike = sum(log(like))
        tmp_results[n] = loglike
      }

      final_results[m, k] = mean(tmp_results)
    }
    #print(m)
  }

  return(final_results)
}
```

We can see that for the not asymptotic case we are far from the real value of k and both the versions converge to the same result. In the asymptotic case, where the sample size is bigger, the two versions converge to the same estimate for k that is closer to the real value.

```
selection_n1_5_FOLDS = k_CV(100, 11, 0.01, 100, 5)
selection_n1_10_FOLDS = k_CV(100, 11, 0.01, 100, 10)
K_5_FOLDS_n1 = argmax(selection_n1_5_FOLDS)
K_10_FOLDS_n1 = argmax(selection_n1_10_FOLDS)
print("The 5 k-fold and 10 k-folds best models for not asymptotic sample are:")
```

```
## [1] "The 5 k-fold and 10 k-folds best models for not asymptotic sample are:"
```

```
print(c(FIVE_K_FOLDS = mean(K_5_FOLDS_n1), TEN_K_FOLDS= mean(K_10_FOLDS_n1)))
```

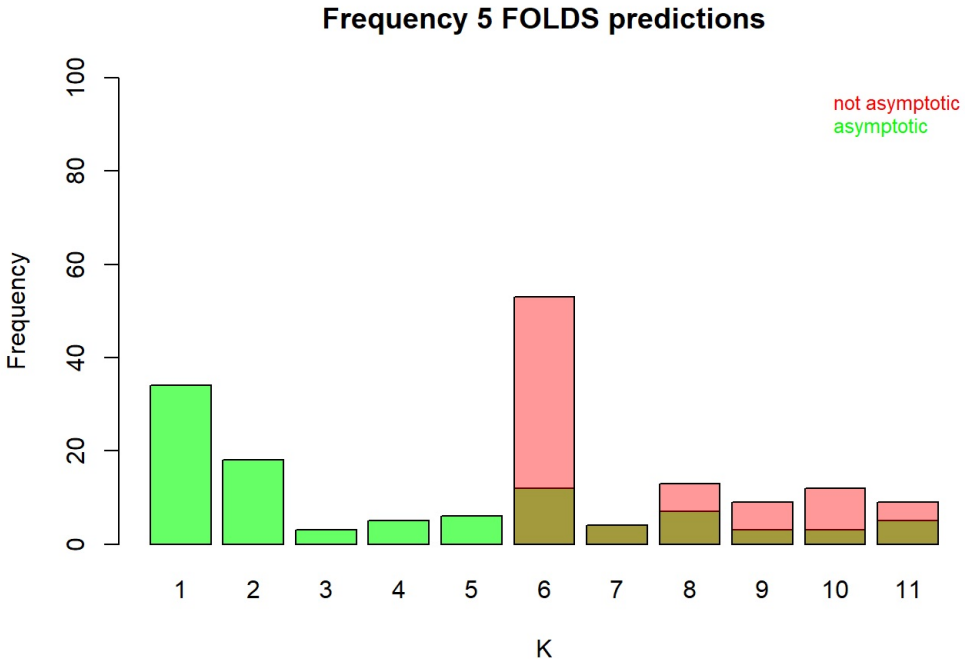
```
## FIVE_K_FOLDS  TEN_K_FOLDS
##         3.97         3.78
```

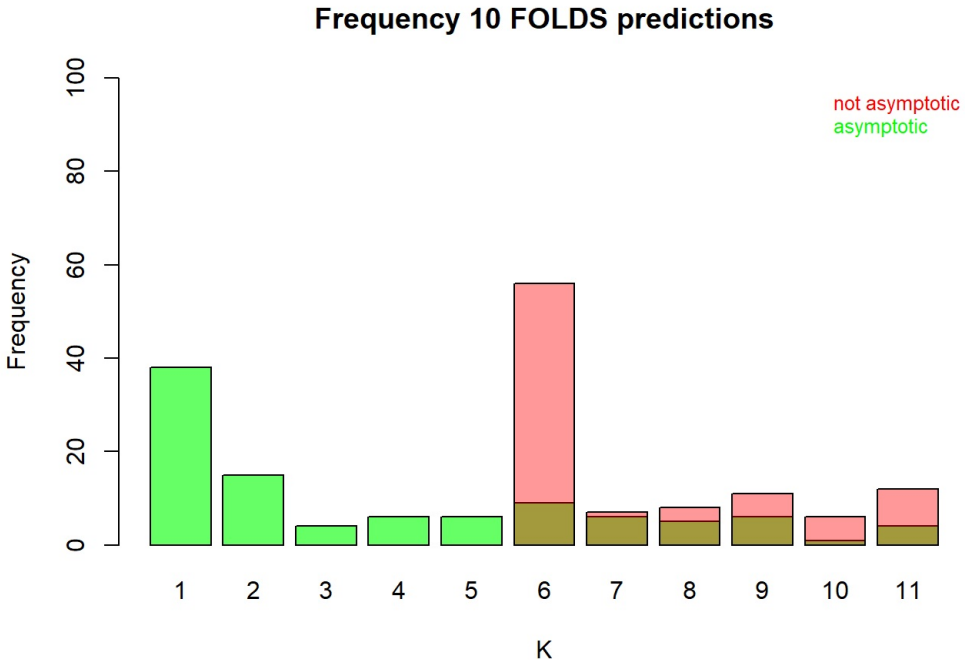
```
selection_n2_5_FOLDS = k_CV(1000, 11, 0.01, 100, 5)
selection_n2_10_FOLDS = k_CV(1000, 11, 0.01, 100, 10)
K_5_FOLDS_n2 = argmax(selection_n2_5_FOLDS)
K_10_FOLDS_n2 = argmax(selection_n2_10_FOLDS)
print("The 5 k-fold and 10 k-folds best models for reasonably asymptotic sample are:")
```

```
## [1] "The 5 k-fold and 10 k-folds best models for reasonably asymptotic sample are:"
```

```
print(c(FIVE_K_FOLDS = mean(K_5_FOLDS_n2), TEN_K_FOLDS= mean(K_10_FOLDS_n2)))
```

```
## FIVE_K_FOLDS  TEN_K_FOLDS
##          7.5          7.4
```





Wasserstein based score

As last method we implement the Wass score, we decide to evaluate the integral that gives the score numericaly, aprroximating it with the following sum: $\hat{W} \approx \sum_{i=1}^{n} |F_{Tr}^{-1}(z_i|\theta) - \hat{F}_{Te}^{-1}(z_i)| \cdot 10^{-3}$, where each z_i is a level of quantile function. We evaluate the function on 1000 different levels in the $(0, 1)$ interval, for this reason we have a 10^{-3} width for every rectangle with which we approximate the integral. Unlike the others methods, where the score is evaluated relying on likelihood function, here we use a loss function so in this case we have to minimize it.

input: sample_size size of dataset generated from Bart density, K_max maximum number of gaussian in a model, t convergence threshold for EM algorithm, M number of samples.

output: a matrix containing all the Wass score, for each model and for each sample

```
wass = function(sample_size, k_max, t, M){

  tmp_results = matrix(NA, M, k_max)

  for(m in 1:M){
    XX = rnormmix(sample_size,
                  lambda = c(0.5, rep(0.1,5)),
                  mu     = c(0, ((0:4)/2)-1),
                  sigma  = c(1, rep(0.1,5)) )


    train = XX[1:(sample_size%/%2)]
    test = XX[(sample_size%/%2+1):sample_size]


    for(k in 1:k_max){

      # parameters initialization
      p = c(rep(1/k, k))
      mu = c(0, ((0:(k-2))/2)-1)
      sigma  = c(1, rep(0.1, k-1))
      if (k == 1){
        mu = c(0)
        sigma = c(1)}

      mu = mu + rnorm(k, 0, 0.1)
      sigma = sigma + rnorm(k, 0, 0.01)

      EM_params = EM_algo(train, k, p, mu, sigma, t)
      len = length(EM_params[["res"]][["deviance"]])

      p = as.numeric(as.vector(EM_params[["res"]][len, 2:(k*3+1)][1:k]))
      mu = as.numeric(as.vector(EM_params[["res"]][len, 2:(k*3+1)][(k+1):(2*k)]))
      sigma = as.numeric(as.vector(EM_params[["res"]][len, 2:(k*3+1)][(2*k+1):(3*k)]))

      levels = seq(0.001, 0.999, by=.001)
      q_train = qmixnorm(levels,mu,sigma,p)
      q_test = quantile(test, levels)
      #print(q_train)
      #print(q_test)
      approx_wass = sum(abs(q_train-q_test))*0.001

      tmp_results[m,k] = approx_wass
    }
    #print(m)
  }
  return(tmp_results)
}
```

As the previous methods we can see that in a not asymptotic case it doesn't work. Instead, in the asymptotic case we are pleasantly surprised, given the simplicity of the method, because it seems to provide a good estimate for the parameter k also with respect to some of the previous methods.

```
final_was_n1 = wass(100, 11, 0.01, 100)
K_estimator_n1 = argmin(final_was_n1)
print("The wass best model for not asymptotic sample is")
```

```
## [1] "The wass best model for not asymptotic sample is"
```

```
print(c(k_est_n1 = mean(K_estimator_n1)))
```

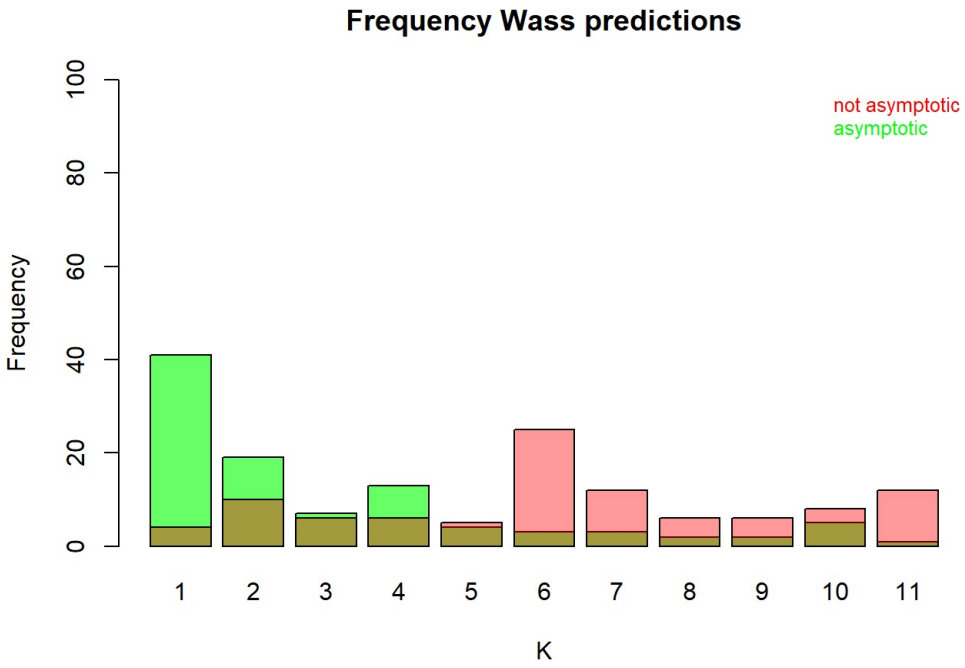
```
## k_est_n1
##     3.06
```

```
final_was_n2 = wass(1000, 11, 0.01, 100)
K_estimator_n2 = argmin(final_was_n2)
print("The wass best model for reasonably asymptotic sample is")
```

```
## [1] "The wass best model for reasonably asymptotic sample is"
```

```
print(c(k_est_n2 = mean(K_estimator_n2)))
```

```
## k_est_n2
##     6.39
```



Model selectors comparison

Given that we have the ground trouth, in this case the number of Gaussian in the model is $k = 6$, we can compute the MSE for each model selection tecnique from the estimated set of k. We can see that AIC is our best model selector tecnique, while the BIC is the worst. As not expected, the MSE for the sample splitting tecnique, with dimensions 70-30 is worse than the 50-50 and 30-70 splitting. The K-fold tecniques seems to be second best option and they give similar results. Differently from the previous consideration on Wasserstein method, we discover that its MSE is pretty high even if its bias is near to zero.

```
actual = c(rep(6,100))
AIC_MSE = mse(actual,K_AIC_n2)
BIC_MSE = mse(actual,K_BIC_n2)
MSE_50_50 = mse(actual,K_CV_50_50_n2)
MSE_70_30 = mse(actual,K_CV_70_30_n2)
MSE_30_70 = mse(actual,K_CV_30_70_n2)
MSE_5_FOLDS = mse(actual,K_5_FOLDS_n2)
MSE_10_FOLDS = mse(actual,K_10_FOLDS_n2)
WASS_MSE = mse(actual,K_estimator_n2)
print(c(AIC_MSE = AIC_MSE,BIC_MSE = BIC_MSE,MSE_50_50 = MSE_50_50, MSE_70_30 = MSE_70_30, MSE_30_70 = MSE_30_70,
        MSE_5_FOLDS = MSE_5_FOLDS,MSE_10_FOLDS = MSE_10_FOLDS, WASS_MSE = WASS_MSE))
```

```
##      AIC_MSE      BIC_MSE    MSE_50_50    MSE_70_30    MSE_30_70  MSE_5_FOLDS
##         0.21        14.69         6.46         7.50         6.61         5.54
## MSE_10_FOLDS     WASS_MSE
##         5.34         8.61
```