STAT34800 HW7

Sarah Adilijiang

Problem A: HMM

(1) EM algorithm

In our two-state simple HMM model:

$$X_t|Z_t = k \sim N(\mu_k, \sigma^2), \quad \textit{where } k \in \{1,2\} \textit{ and } \sigma^2 \textit{ is known}$$

EM algorithm:

1. Initialize $\mu^0 = \{\mu_1^0, \mu_2^0\}$, and evaluate the incomplete data log-likelihood with the parameters:

$$l(\mu^0) = \log\left(p(X_1, ..., X_T|\mu^0)\right) = \log\left(\sum_{k=1}^{K} \alpha_{Tk}(\mu^0)\right)$$

2. **E-step**: Evaluate the posterior probabilities $\gamma_{Z_i}(k)$ using the current values of the $\mu_k's$:

$$\gamma_{Z_t}(k) = p(Z_t = k|X_1, ..., X_T) = \frac{p(X_1, ..., X_T; Z_t = k)}{\sum_{k=1}^{K} p(X_1, ..., X_T; Z_t = k)} = \frac{\alpha_{tk}\beta_{tk}}{\sum_{k=1}^{K} \alpha_{tk}\beta_{tk}}$$

Then, use the current values of $\gamma_{Z_t}(k)$ to find the expectation of the complete data log-likelihood $Q(\mu, \mu^0)$, evaluated at an arbitrary μ.

Since the complete data likelihood is:

$$p(X_1, ..., X_T; Z_1, ..., Z_T) = p(Z_1)p(X_1|Z_1)p(X_2, ..., X_T; Z_2, ..., Z_T|Z_1)$$

$$= p(Z_1)p(X_1|Z_1)p(Z_2|Z_1)p(X_2|Z_2)p(X_3, ..., X_T; Z_3, ..., Z_T|Z_2) = \cdots = p(Z_1)\prod_{t=2}^{T} p(Z_t|Z_{t-1})\prod_{t=1}^{T} p(X_t|Z_t)$$

So:

$$Q(\mu, \mu^0) = E_{Z|X,\mu^0}\ [\log(P(X, Z|\mu))] = E_{Z|X,\mu^0}\ \left[\log(p(Z_1)) + \sum_{t=2}^{T}\log(p(Z_t|Z_{t-1})) + \sum_{t=1}^{T}\log(p(X_t|Z_t, \mu))\right]$$

$$= C + \sum_{t=1}^{T}\sum_{k=1}^{K}\log(p(X_t|Z_t = k, \mu))\ p(Z_t = k|X_1, ..., X_T)\ =\ C + \sum_{t=1}^{T}\sum_{k=1}^{K}\gamma_{Z_t}(k)\ \log(p(X_t|Z_t = k, \mu))$$

$$=\ C + \sum_{t=1}^{T}\sum_{k=1}^{K}\gamma_{Z_t}(k)\ \left(-\frac{1}{2}\log(2\pi\sigma^2) - \frac{(X_t - \mu_k)^2}{2\sigma^2}\right)$$

where C part does not involve μ.

3. **M-step**:

By setting the derivative to zero:

$$\frac{\partial Q(\mu, \mu^0)}{\partial \mu_k} = 0$$

We can get the MLE estimate of new parameter $\hat{\mu}_k$ with the current values of $\gamma_{Z_t}(k)$ that maximizes the complete data log-likelihood $Q(\mu, \mu^0)$, i.e. $\hat{\mu} = argmax_\mu \ Q(\mu, \mu^0)$:

$$\hat{\mu}_k = \frac{1}{N_k} \sum_{t=1}^{T} \gamma_{Z_t}(k) \ X_t, \qquad where \quad N_k = \sum_{t=1}^{T} \gamma_{Z_t}(k)$$

4. Evaluate the incomplete data log-likelihood with the new parameter estimates:

$$l(\hat{\mu}) = \log\left(p(X_1, ..., X_T|\hat{\mu})\right) = \log\left(\sum_{k=1}^{K} \alpha_{Tk}(\hat{\mu})\right)$$

If the log-likelihood has changed by less than some small ϵ, stop. Otherwise, go back to E-step.

EM function

```
# this is the function Pr(X_t|Z_t=k) for our example
emit = function(mu_k, x){
  dnorm(x, mean=mu_k, sd=sd)   #sd is known
}

pi = c(0.5,0.5)     # Assumed prior distribution on Z_1

# EM algorithm function
HMM.EM = function(X, mu.init) {
    T = length(X)
    K = length(mu.init)

    # initialize parameters
    mu.curr = mu.init

    # EM steps & checks for convergence
    log_liks = c()
    delta.ll = 1
    while(delta.ll > 1e-5) {

        # (1) Compute Forwards Probabilities
        alpha = matrix(nrow=T, ncol=K)
        # Initialize alpha[1,]
        for(k in 1:K){
            alpha[1,k] = pi[k] * emit(mu.curr[k], X[1])
        }
        # Forward algorithm computing alpha[t,]
        for(t in 1:(T-1)){
            m = alpha[t,] %*% P
            for(k in 1:K){
                alpha[t+1,k] = m[k] * emit(mu.curr[k], X[t+1])
            }
        }

        # (2) Compute Backwards Probabilites
```

```
        beta = matrix(nrow=T, ncol=K)
        # Initialize "boundary" beta[T,]
        for(k in 1:K){
            beta[T,k] = 1
        }
        # Backwards algorithm computing beta[t,]
        for(t in (T-1):1){
            for(k in 1:K){
                beta[t,k] = sum(beta[t+1,]*P[k,]*emit(mu.curr[1:K], X[t+1]))
            }
        }


        # (3) E-step: compute posterior p(Zt=k|X1,...,XT)
        ab = alpha*beta   # element-wise matrix multiplication
        z_tk = ab/rowSums(ab)


        # (4) M-step: update estimates of mu_k
        N_k = colSums(z_tk)
        mu.curr = as.vector( t(z_tk) %*% X / N_k )


        # (5) compute & store incomplete data log-likehoods p(X1,...,XT) for each iteration
        # and checks for convergence
        ll       = log(sum(alpha[T,]))
        log_liks = c(log_liks, ll)
        if(length(log_liks)>1){
            delta.ll = abs(log_liks[length(log_liks)]  - log_liks[length(log_liks)-1])
        }
    }

    return(list(mu=mu.curr, Z=z_tk, logliks=log_liks))
}
```

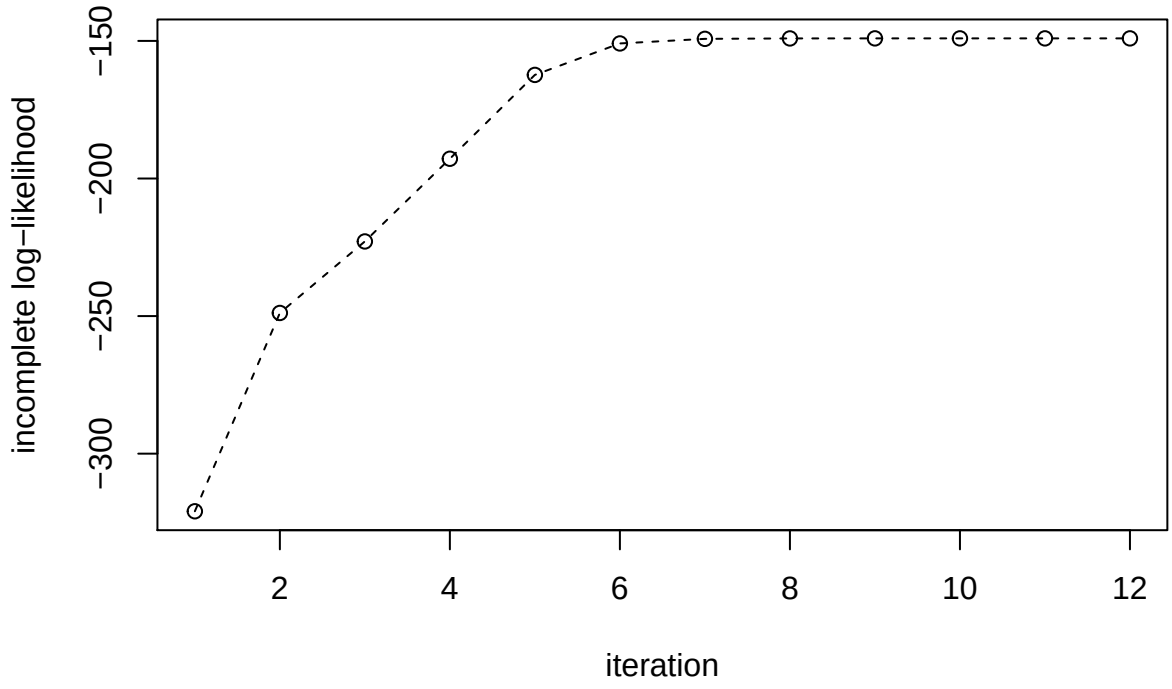
(2) Demonstration by simulation

```
# Simulate data from an HMM
set.seed(1)
T = 200
K = 2     # two states: Z= 1 or 2
sd = 0.4
P = cbind(c(0.9,0.1),c(0.1,0.9))  # transition matrix P

# Simulate the latent (Hidden) Markov states: Z
Z = rep(0,T)
Z[1] = 1
for(t in 1:(T-1)){
    Z[t+1] = sample(K, size=1, prob=P[Z[t],])    # Z= 1 or 2
}

# Simulate the emitted/observed values: X
X = rnorm(T, mean=Z, sd=sd)    # true X|Z=k ~ N(k,sd^2), so true mu=c(1,2)
```

```r
# perform EM for HMM
HMM1 = HMM.EM(X, mu.init=c(0,1))

# estimated mu
HMM1$mu
```

```
## [1] 0.9865166 2.0221746
```

```r
# inspect the evolution of the incomplete log-likelihood
plot(HMM1$logliks, ylab='incomplete log-likelihood', xlab='iteration')
lines(HMM1$logliks, lty=2)
```



Comments:

We can see that the estimated $\hat{\mu} = (0.9865166, 2.0221746)$ is very close to the true value $\mu = (1, 2)$. And the incomplete data log-likelihood strictly increases every iteration. Therefore, our EM algorithm for the HMM model works well here.

(3) Different initial values & local optima

```r
# perform EM with different initial values
HMM1 = HMM.EM(X, mu.init=c(0,1))
HMM2 = HMM.EM(X, mu.init=c(-1,1))
HMM3 = HMM.EM(X, mu.init=c(-1,2))
HMM4 = HMM.EM(X, mu.init=c(0.5,0.5))
HMM5 = HMM.EM(X, mu.init=c(1.5,1.5))
```

```r
# compare estimates with true values
rbind(HMM1$mu,HMM2$mu,HMM3$mu,HMM4$mu,HMM5$mu)
```

```
##           [,1]     [,2]
## [1,] 0.9865166 2.022175
## [2,] 0.9865172 2.022176
## [3,] 0.9865209 2.022185
## [4,] 1.2995776 1.299578
## [5,] 1.2995776 1.299578
```

```r
# compare log-likelihood
rbind(HMM1$logliks[length(HMM1$logliks)], HMM2$logliks[length(HMM2$logliks)],
      HMM3$logliks[length(HMM3$logliks)], HMM4$logliks[length(HMM4$logliks)],
      HMM5$logliks[length(HMM5$logliks)])
```

```
##           [,1]
## [1,] -149.0804
## [2,] -149.0804
## [3,] -149.0804
## [4,] -244.7537
## [5,] -244.7537
```

Comments:

When using different initial values for μ_1 and μ_2, the estimated $\hat{\mu}$ are very close to each other in different settings, and they are also very close to the true value $\mu = (1, 2)$. Besides, the final incomplete data log-likelihood are the same with each other. Therefore, in this case, the EM algorithm here is not sensitive to the initial values, so it does not get stuck in local optima of the log-likelihood.

However, when using the same initial values for μ_1 and μ_2, the estimated $\hat{\mu}$ are exactly the same with each other, however, they are not close to the true value $\mu = (1, 2)$ any more. Besides, the final incomplete data log-likelihood are also exactly the same with each other.

As a result, considering the whole picture that includes both these two cases, we may say that the EM algorithm may get stuck in local optima of the log-likelihood.

Problem B: HMM

Here in our two-state simple HMM model:

$$X_t|Z_t = k \sim Bernoulli(p_{kt}), \quad where\ k \in \{1, 2\}\ and\ p_{kt}\ is\ the\ frequency\ of\ allele\ 1$$

```r
# Simulate data from an HMM
set.seed(1)
T = 1000   # longer length of Markov chain
K = 2      # two states: Z= 1 or 2
P = rbind(c(0.9,0.1),c(0.1,0.9))   # transition matrix P

# simulate the matrix of allele frequencies in each of the K populations at each of T loci
p = matrix(runif(K*T),nrow=K,ncol=T)     # p_kt = Pr(X_t=1|Z_t=k)

# Simulate the latent (Hidden) Markov states: Z
Z = rep(0,T)
Z[1] = 1
for(t in 1:(T-1)){
  Z[t+1] = sample(K, size=1, prob=P[Z[t],])    # Z= 1 or 2
```

```
}

# Work out the corresponding Bernoulli probability for each state: p_kt
prob = rep(0,T)
for(i in 1:T){
  prob[i] = p[Z[i],i]
}

# Simulate the emitted/observed values: X
X = rbinom(n=T, size=1, prob=prob)  # true X_t|Z_t=k ~ Bernoulli(p_kt)
```

(1) Posterior p(Zt=k|X1,. . . ,XT)

```
# this is the Bernoulli probability p_kt = Pr(X_t=1|Z_t=k) for our example
emit = function(p, x){
  ifelse(x==1, p, 1-p)   #p is given in the simulation
}

pi = c(0.5,0.5)    # Assumed prior distribution on Z_1

# (1) Compute Forwards Probabilities
alpha = matrix(nrow=T, ncol=K)

# Initialize alpha[1,]
for(k in 1:K){
    alpha[1,k] = pi[k] * emit(p[k,1], X[1])
}
alpha[1,] = alpha[1,]/sum(alpha[1,])      #normalize

# Forward algorithm computing alpha[t,]
for(t in 1:(T-1)){
    m = alpha[t,] %*% P
    for(k in 1:K){
        alpha[t+1,k] = m[k] * emit(p[k,t+1], X[t+1])
    }
    alpha[t+1,] = alpha[t+1,]/sum(alpha[t+1,])    #renormalize every iteration
}


# (2) Compute Backwards Probabilites
beta = matrix(nrow=T, ncol=K)

# Initialize "boundary" beta[T,]
for(k in 1:K){
    beta[T,k] = 1
}
beta[T,] = beta[T,]/sum(beta[T,])         #normalize

# Backwards algorithm computing beta[t,]
for(t in (T-1):1){
    emit_probs = rep(0,K)      # p_k(t+1) = Pr(X_t+1=1|Z_t+1=k)
    for (k in 1:K) {
        emit_probs[k] = emit(p[k,t+1], X[t+1])
```

```
    }
    for(k in 1:K){
        beta[t,k] = sum(beta[t+1,]*P[k,]*emit_probs)
    }
    beta[t,] = beta[t,]/sum(beta[t,])     #renormalize every iteration
}


# (3) compute posterior p(Zt=k|X1,...,XT)
ab = alpha*beta    # element-wise matrix multiplication
z_tk = ab/rowSums(ab)
```

(2) True error rate when knowing the true Z

```
z_t = apply(z_tk, 1, which.max)    #Z= 1 or 2
error_rate = mean(z_t!=Z);   error_rate
```

```
## [1] 0.186
```

Answer:

When assigning each Z_t to its most probable value and comparing it with the corresponding true values, we get that the error rate is 0.186.

(3) Estimate error rate when not knowing the true Z

```
p_t = apply(z_tk, 1, max)
error_rate = mean(1-p_t);   error_rate
```

```
## [1] 0.1831756
```

Answer:

Here we are assigning each Z_t to its most probable value, i.e. $Z_t = argmax_k Pr(Z_t = k)$, so we can estimate the probability of getting wrong prediction for each Z_t by using the value $1 - max_k\{Pr(Z_t = k)\}$. Taking the mean of this value over t, we get that the error rate is 0.1831756. It is very close to the true error rate 0.186 when we know the true values of Z_t.

Problem C: Spatial Gaussian Processes

Deal with 0 counts and perform the transformation of $\hat{f}$: $x = \log(\hat{f}/(1-\hat{f}))$, where $\hat{f} \in [0, 1]$.

```
# load data
ccr5 = read.table("CCR5.freq.txt", header=TRUE)
ccr5[,1] = ifelse(ccr5[,1]>180,ccr5[,1]-360,ccr5[,1])  # changes longitudes>180 to negative

# compute pairwise distances
geo.dist = geosphere::distm(ccr5[,1:2])/1000   # distance in km
# distance matrix, d_{ij} = |y_i - y_j|

# deal with 0 counts
ccr5$count = round(ccr5$Freq * ccr5$SampleSize * 2)
ccr5$f_hat = (ccr5$count + 1)/(ccr5$SampleSize*2 + 2)
```

```r
# logit transformation of f_hat
ccr5$Z = log(ccr5$f_hat/(1-ccr5$f_hat))
```

(1) Compute the covariance matrix

```r
# function for computing the covariance matrix
compute_Sigma_SE = function(a){
    d = geo.dist
    Sigma_SE = a[1] * exp(-(d/a[2])^2)  # Squared-Exponential covariance function
    return(Sigma_SE)
}
```

```r
# try different a's and check for positive semi-definite by Cholesky decomposition
Sigma_SE = compute_Sigma_SE(a=c(1,1))
L = t(chol(Sigma_SE)); L[1:5,1:5]
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    0    0    0    0
## [2,]    0    1    0    0    0
## [3,]    0    0    1    0    0
## [4,]    0    0    0    1    0
## [5,]    0    0    0    0    1
```

```r
Sigma_SE = compute_Sigma_SE(a=c(5,20))
L = t(chol(Sigma_SE)); L[1:5,1:5]
```

```
##               [,1]     [,2]          [,3]     [,4]     [,5]
## [1,]  2.236068e+00 0.000000  0.000000e+00 0.000000 0.000000
## [2,]  0.000000e+00 2.236068  0.000000e+00 0.000000 0.000000
## [3,]  0.000000e+00 0.000000  2.236068e+00 0.000000 0.000000
## [4,] 9.900165e-272 0.000000  0.000000e+00 2.236068 0.000000
## [5,]  0.000000e+00 0.000000 7.905050e-323 0.000000 2.236068
```

```r
Sigma_SE = compute_Sigma_SE(a=c(0.1,100))
L = t(chol(Sigma_SE)); L[1:5,1:5]
```

```
##               [,1]          [,2]          [,3]          [,4]      [,5]
## [1,]  3.162278e-01  0.000000e+00  0.000000e+00  0.000000e+00 0.0000000
## [2,]  5.568291e-83  3.162278e-01  0.000000e+00  0.000000e+00 0.0000000
## [3,]  7.698395e-84  8.962687e-21  3.162278e-01  0.000000e+00 0.0000000
## [4,]  4.424318e-12 -7.790552e-94 -1.077076e-94  3.162278e-01 0.0000000
## [5,] 1.353454e-148  1.289156e-58  3.984590e-14 1.357157e-107 0.3162278
```

Comments:

Here we see that all the Cholesky decompositions succeeded when using different values of a, so the resulting covariance matrix is valid, that is, it is positive semi-definite.

(2) Compute log-likelihood

$$x^{\text{obs}}|m, a \sim N_r(\mu, \Sigma), \quad \text{where} \;\; \mu = \text{rep}(m, r) \;\; \text{and} \;\; \Sigma = \Sigma(a_1, a_2)$$

```r
# function for computing log-likelihood of a Gaussian Process (multivariate normal)
minus_loglik = function(x, pars){   # pars = c(m,a1,a2)
    m = pars[1]
    a = pars[2:3]
```

```
    r = length(x)
    mu = rep(m,r)
    Sigma_SE = compute_Sigma_SE(a)
    loglik = mvtnorm::dmvnorm(x, mean=mu, sigma=Sigma_SE, log=TRUE)
    return(-loglik)
}
```

(3) Estimate a and m by maximizing log-likelihood

```
# function for computing MLEs using R function optim()
# using different initial values of parameters
MLEs = function(x, pars.init){
    MLEs = optim(par=pars.init, minus_loglik, x=x)   # pars = c(m,a1,a2)
    m_hat = MLEs$par[1]
    a_hat = MLEs$par[2:3]
    minus_maxloglik = MLEs$value
    return(list(m.hat=m_hat, a.hat=a_hat, max.loglik=-minus_maxloglik))
}
```

```
# find MLEs for our data using different initial values for optim() function
                        # pars = c(m,a1,a2)
MLE1 = MLEs(x=ccr5$Z, pars.init=c(0,1,1))
MLE2 = MLEs(x=ccr5$Z, pars.init=c(1,5,10))
MLE3 = MLEs(x=ccr5$Z, pars.init=c(10,10,100))
MLE4 = MLEs(x=ccr5$Z, pars.init=c(20,100,1000))

# mu
c(MLE1$m.hat, MLE2$m.hat, MLE3$m.hat, MLE4$m.hat)
```

```
## [1] -2.605855 -2.605827 -2.720352 -2.720147
```

```
# a=(a1, a2)
rbind(MLE1$a.hat, MLE2$a.hat, MLE3$a.hat, MLE4$a.hat)
```

```
##           [,1]       [,2]
## [1,] 0.5309045   3.747659
## [2,] 0.5309019  14.030665
## [3,] 0.5494890 146.764326
## [4,] 0.5467214 145.360103
```

```
# maximized log-likelihood
c(MLE1$max.loglik, MLE2$max.loglik, MLE3$max.loglik, MLE4$max.loglik)
```

```
## [1] -78.26564 -78.26564 -72.98569 -72.98857
```

Answer:

Here we see that when using different initial values of parameters for R function optim(), it will get roughly stable estimated values for $\hat{m}$ between $(-2.6, -2.7)$ and for $\hat{a}_1$ between $(0.53, 0.55)$, but quite different estimated values for a_2. By comparing the maximized log-likelihood values obtained from the optim() function, we see that the log-likelihood roughly reaches its maximum when the initial values of $a_2 = 100$ or 1000, and provides the roughly stable estimated values for $\hat{a}_2$ between $(145.4, 146.8)$.

Finally, I take the values of estimation where I get the maximum log-likelihood between these four settings: $\hat{m} = -2.720352$ and $\hat{a} = (0.5494890146.764326)$.

(4) Kriging: imputation of missing values

(a) Compute E(X_i|X_{-i})

Suppose $X = (X_1, ..., X_r) \sim N_r(\mu, \Sigma)$, and we partition X into two parts $X_i \in R$ and $X_{-i} \in R^{r-1}$, then the distribution of X_i conditional on $X_{-i} = a$ is univariate normal:

$$X_i|X_{-i} \sim N(\bar{\mu}, \bar{\Sigma})$$

where

$$E(X_i|X_{-i}) = \bar{\mu} = \mu_i + \Sigma_{i(-i)}\Sigma^{-1}_{(-i)(-i)}(X_{-i} - \mu_{-i})$$

```
# function for computing the conditional expectation: E(X_i|X_{-i})
GP_conditional_expect = function(x, i, mu, Sigma){
    cm = mu[i] + Sigma[i,-i] %*% solve(Sigma[-i,-i]) %*% (x[-i] - mu[-i])
    return(as.numeric(cm))
}
```

(b) Compute E(x(y_1)|x(y_2),...,x(y_r)) for our data

```
m = MLE3$m.hat
a = MLE3$a.hat
r = nrow(ccr5)
mu = rep(m,r)
Sigma = compute_Sigma_SE(a)
```

```
x1_con.exp = GP_conditional_expect(x=ccr5$Z, i=1, mu, Sigma)
x1_con.exp
```

```
## [1] -2.689484
```

Answer:

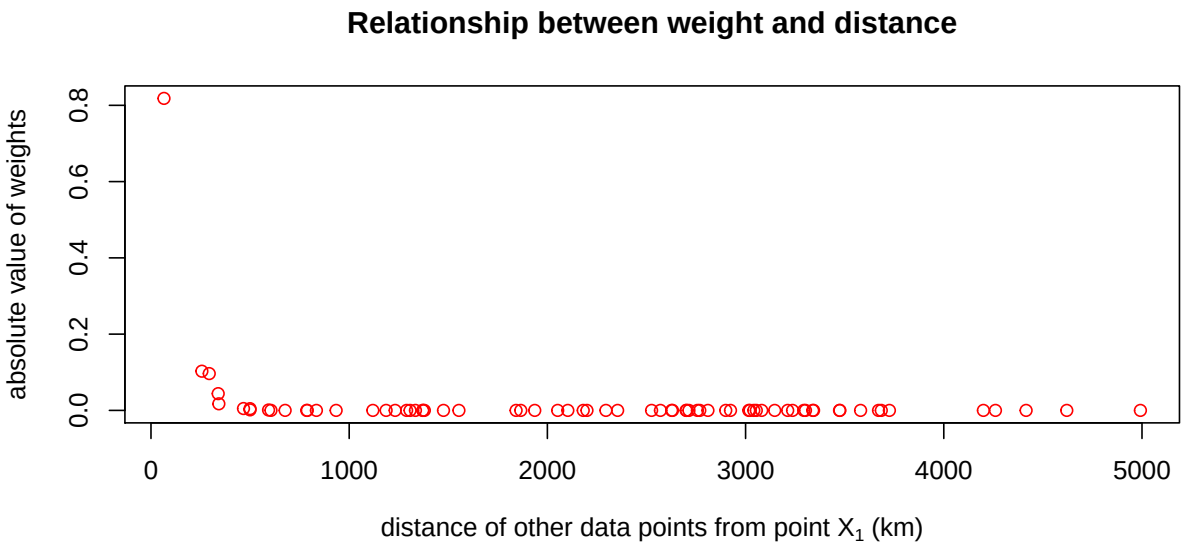
Here we get the conditional expectation $E(x(y_1)|x(y_2), ..., x(y_r)) = -2.689484$.

Note that the conditional expectation $E(X_i|X_{-i})$ is a weighted linear combination of the other data points X_{-i}, where the weights are:

$$w_{-i} = \Sigma_{i(-i)}\Sigma^{-1}_{(-i)(-i)}$$

Now we check if these weights weight the nearby data points more, that is, the data points spatially closer to X_i will have the larger weight values.

```
# check for data point X20
i = 1
w = rep(0,r)
w[-i] = Sigma[i,-i] %*% solve(Sigma[-i,-i])
index_w = order(abs(w[-i]), decreasing=TRUE)       # large to small
index_d = order(geo.dist[i,-i], decreasing=FALSE)  # close to far
# distance matrix, d_{ij} = |y_i - y_j|, distance in km

# relationship between weight and distance
plot(geo.dist[i,-i][index_d], abs(w[-i][index_w]), xlab=expression(paste("distance of other data points
```



Comments:

From the plot we can conclude that the closer the data point is to X_1 spatially, the larger the weight is of that data point. And after the closest 10 data points, the weights of data points have decayed to almost zero values.

(c) Repeat for each of the r datapoints

```
# imputation with conditional expectation
x_con.exp = rep(NA,r)
for (i in 1:r) {
    x_con.exp[i] = GP_conditional_expect(x=ccr5$Z, i, mu, Sigma)
}
```

(d) Imputation by Gaussian Process Regression vs Mean

```
# imputation with mean
x = ccr5$Z
x_mean = rep(NA,r)
for (i in 1:r) {
    x_mean[i] = mean(x[-i])
}
```

```
# compare accuracy
MSE_con.exp = mean((x_con.exp - x)^2)
MSE_mean = mean((x_mean - x)^2)
data.frame(MSE_con.exp=MSE_con.exp, MSE_mean=MSE_mean)
```

```
##   MSE_con.exp  MSE_mean
## 1   0.4282045 0.5461608
```

Comments:

Comparing the two imputation methods, the imputation using Gaussian Process Regression (conditional expectation) provides more accurate estimation for missing values than the imputation method using mean values.