HW1

Conti Andrea - Napoli Mario - Mascolo Davide

Notation:

- $D_n = (i_1, i_2, \ldots, i_k, \ldots, i_n)$
- x_j = {number of indices i_k in D_n equal to j} = $|\{k : i_k = j\}|, j \in \{1, \ldots, d\}$
- $y^k = L \cdot x^k$, with L that is a random projection matrix of dimension $p \times d$ whose entries are drawn *independently* as $N_1(0, \frac{1}{p})$, and y^k is the *p-dimensional* vector.
- The *Johnson-Lindenstrauss lemma (JL)* says that, for every tolerance $\epsilon > 0$ we pick: $\mathrm{P}((1-\epsilon) \cdot ||x|| \leq (1+\epsilon) \cdot ||x||) \geq 1 - e^{-e^2 \cdot p}$.

Exercise 1

1.1

Show the validity of the update step: increasing by 1 the j^{th} coordinate of x^{k-1} corresponds to add the j^{th} column of L to y^{k-1}.

Let's assume that we have:

$$L = \begin{pmatrix} a_{11} & \cdots & a_{1d} \\ \vdots & & \vdots \\ a_{p1} & \cdots & a_{pd} \end{pmatrix} \qquad X = \begin{pmatrix} b_{11} \\ \vdots \\ b_{d1} \end{pmatrix}$$

Suppose that, at time i, we get a new packet b_{k1}, with $1 \leq k \leq d$, so we upload the correspondent element in the respective position in $X^{(i-1)}$ in the frequency vector.

$$X^{(i)} = \begin{pmatrix} b_{11} \\ \vdots \\ b_{k1} + 1 \\ \vdots \\ b_{d1} \end{pmatrix}$$

After the X update, we have to compute $L \cdot X^{(i)}$ in order to get $Y^{(i)}$.

$$Y^{(i)} = L \cdot X^{(i)} = \begin{pmatrix} a_{11} & \cdots & a_{1k} & \cdots & a_{1d} \\ \vdots & & \vdots & & \vdots \\ a_{p1} & \cdots & a_{pk} & \cdots & a_{pd} \end{pmatrix} \cdot \begin{pmatrix} b_{11} \\ \vdots \\ b_{k1} \\ \vdots \\ b_{d1} \end{pmatrix} = \begin{pmatrix} a_{11}b_{11} + \cdots + a_{1k}b_{k1} + \cdots + a_{1d}b_{d1} \\ \vdots \\ a_{p1}b_{11} + \cdots + a_{pk}b_{k1} + \ldots a_{pd}b_{d1} \end{pmatrix}$$

Given that no b elements but b_{k1} have been updated we have that all the multiplications of the type $a_{ij}b_{j1}$ have the same values they had at time $(i-1)$, except the elements multiplied by b_{k1}, for this reason we can say that the real update is due to the k-th part of each sum because it adds exactly one time $a_{ik}, \forall i \in \{1, \ldots, p\}$ at the total summation, in other words we have added the k-th column of L to $Y^{(i-1)}$.

1.2

```
## Constraint
## p << n << d
p_vec <- c(5, 25, 100)
n_vec <- c(40, 200, 1000)
d_vec <- c(125, 550, 5000)
M       <- 1000 # Number of simulations

for (s in 1:3){
  p       <- p_vec[s] # Size of projection
  n       <- n_vec[s] # Stream size
  d       <- d_vec[s] # Alphabet length
  ## Although you have asked us to compute the value of p starting from epsilon we do the reverse operation in or
der to have an integer (well defined) number.
  epsilon <- 1/sqrt(p)
  pr <- 1 - exp(-epsilon^2 * p)

  # Algorithm doesn't see them but...I'm God and I can
  D_n = sample(1:d, size = n, replace = TRUE) # Input stream
  # Frequency vector and its update
  X = rep(0, d)
  for (i in 1:n){
    X[D_n[i]] = X[D_n[i]] + 1
  }
  X_norm = sqrt(sum(X^2)) # Norm of X

  cnt <- 0

  # Simulations cycle
  beg <- Sys.time()
  for (i in 1:M){
    # Create a random L
    L <- matrix(rnorm(p*d, mean = 0, sd = 1/sqrt(p)),
                nrow = p,
                ncol = d,
                byrow = T)

    #Initialize Y
    Y = rep(0, p)

    ## Cycle
    for (i_k in 1:n){
      ## Received packet
      j = D_n[i_k]
      Y = Y + L[,j]
    }

    Y_norm = sqrt(sum(Y^2))
    ## We updated cnt if the condition on the left side of the JL lemma is verified.
    cnt = cnt + as.integer(((1 - epsilon) * X_norm <= Y_norm) && (Y_norm <= (1 + epsilon) * X_norm))
  }

  ## Print
  cat('Values: p = ', p,
      'n =', n,
      'd =', d,
      'epsilon =', epsilon, "\n", sep = " ")
  ## Print True if the JL lemma is verified.
  print(cnt / M >= pr)
  fin <- Sys.time() - beg
  print(fin)
}
```

```
## Values: p =  5 n = 40 d = 125 epsilon = 0.4472136
## [1] TRUE
## Time difference of 0.1096981 secs
## Values: p =  25 n = 200 d = 550 epsilon = 0.2
## [1] TRUE
## Time difference of 1.10902 secs
## Values: p =  100 n = 1000 d = 5000 epsilon = 0.1
## [1] TRUE
## Time difference of 36.35729 secs
```

1.3

As you can see above, we can use $p \sim \Theta(\frac{log(n)}{\epsilon^2})$ words of space to store y but we have not achieved our goal because the algorithm stores also the matrix L. Can we avoid to store L in order to use only $O(log(n))$ space? Yes, we can generate (*using normal distribution*) on the fly a vector of dimension p and sum this vector to y. The problem is that starting by y built in this way we can not return to the original space. A possible

implementation of this idea is the following:

```
M       <- 1000 # Number of simulations
p       <- 10 # Size of projection
n       <- 100 # Stream size
d       <- 1000 # Alphabet length
epsilon <- 1/sqrt(p)
pr <- 1 - exp(-epsilon^2 * p)

# Algorithm doesn't see them but...I'm God and I can
D_n = sample(1:d, size = n, replace = TRUE) # Input stream
X = rep(0, d) # Frequency vector and its update
for (i in 1:n){
  X[D_n[i]] = X[D_n[i]] + 1
}
X_norm = sqrt(sum(X^2)) # Norm of X

cnt <- 0

# Simulations cycle
for (i in 1:M){
  #Initialize Y
  Y = rep(0, p)

  ## Cycle
  for (i_k in 1:n){
    L = rnorm(p, mean = 0, sd = 1/sqrt(p))
    j = D_n[i_k]
    Y = Y + L
  }

  Y_norm = sqrt(sum(Y^2))
  cnt = cnt + as.integer(((1 - epsilon) * X_norm <= Y_norm) && (Y_norm <= (1 + epsilon) * X_norm))
}

cnt / M >= pr
```

```
## [1] TRUE
```

Exercise 2

2.1

The probability density function is used to calculate the probability of events defined in terms of the corresponding continuous v.c. X. For example, for $a < b$, it is: $P(a < X \leq b) = \int_a^b f_X(x)dx$

A function $f(.)$ with a domain on the real line and the codomain R_0^+ is defined as a density function if and only if:

- $f(x) \geqslant 0 \forall x \in R$
- $\int_{-inf}^{inf} f(x)dx = 1$

So, in this case, I have to impose that the approximating density is:

- $\hat{f}(x;\theta) \geqslant 0 \forall x \in S$, with $S \in [0, 1]$
- $\int_0^1 \hat{f}(x;\theta)dx = 1$

In these expressions, we have that h is a length so it is always greater than 0 and the indicator function may take value 0 or value 1. So, setting the first constraint we have that:

$\sum_{j=1}^{N} \frac{\pi_j}{h} 1(\mathrm{x} \in \mathrm{B_j}) \geqslant 0 \Rightarrow \pi_j \geqslant 0 \ \forall j \in \{1, \ldots, N\}$

Setting the second constraint we have that:
$1 = \int_0^1 \sum_{j=1}^{N} \frac{\pi_j}{h} 1(\mathrm{x} \in \mathrm{B_j})dx = \int_0^h \sum_{j=1}^{N} \frac{\pi_j}{h} 1(\mathrm{x} \in \mathrm{B_j})dx + \int_h^{2h} \sum_{j=1}^{N} \frac{\pi_j}{h} 1(\mathrm{x} \in \mathrm{B_j})dx + \ldots + \int_{1-h}^{1} \sum_{j=1}^{N} \frac{\pi_j}{h} 1(\mathrm{x} \in \mathrm{B_j})dx = \frac{\pi_1}{h} \cdot h + \frac{\pi_2}{h} \cdot h + \ldots + \frac{\pi_N}{h} \cdot$ using the additive property of the integrals and exploiting the length of the interval equal to h.

In conclusion, each parameter π_j is greater than zero and their sum is equal to one.

- $\pi_j \geq 0, \forall j \in \{1, \ldots, N\}$
- $\sum_{j=1}^{N} \pi_j = 1$

2.2

```
## Approximating density
proxy_dens <- function(x, pi_greco, h){
  ## Number of bins
  N <- ceiling(1/h)

    ## Cycle
    for (j in 0:N-1){
      if ((x >= h*j) & (x < h*(j+1))){
        f <- pi_greco[j+1]/h
        return(f)
      }else{
        if (x == 1){
          l_pi <- length(pi_greco)
          f <- pi_greco[l_pi] / h
          return(f)}}}}
```

At this point you test the function implemented above fixing various values for the parameters and respecting the constraints.

```
## Test
## Sequence of x
x <-seq(0, 1, 0.01)
h        <- 0.1
pi_greco <- c(0.02, 0.08, 0.1, 0.2, 0.05,
              0.05, 0.06, 0.04, 0.3, 0.1)

## Check pi_greco
print(sum(pi_greco))
```

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

```
print(pi_greco > 0)
```

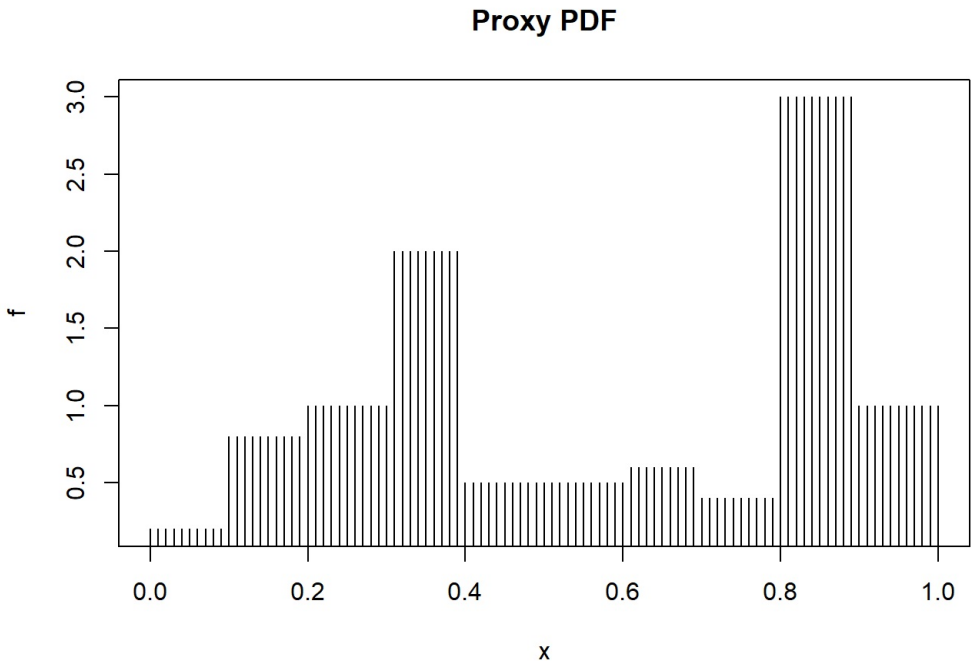
```
##  [1] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
```

```
## Apply function
f_hat <- sapply(x, function(x) proxy_dens(x, pi_greco, h))

## Result
f_hat
```

```
##   [1] 0.2 0.2 0.2 0.2 0.2 0.2 0.2 0.2 0.2 0.2 0.8 0.8 0.8 0.8 0.8 0.8 0.8 0.8
##  [19] 0.8 0.8 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 2.0 2.0 2.0 2.0 2.0
##  [37] 2.0 2.0 2.0 2.0 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5
##  [55] 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.6 0.6 0.6 0.6 0.6 0.6 0.6 0.6 0.6 0.4 0.4
##  [73] 0.4 0.4 0.4 0.4 0.4 0.4 0.4 0.4 3.0 3.0 3.0 3.0 3.0 3.0 3.0 3.0 3.0 3.0
##  [91] 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0
```

```
## Plot
plot(x, f_hat,
     type = 'h',
     main = 'Proxy PDF',
     xlab = 'x',
     ylab = 'f')
```



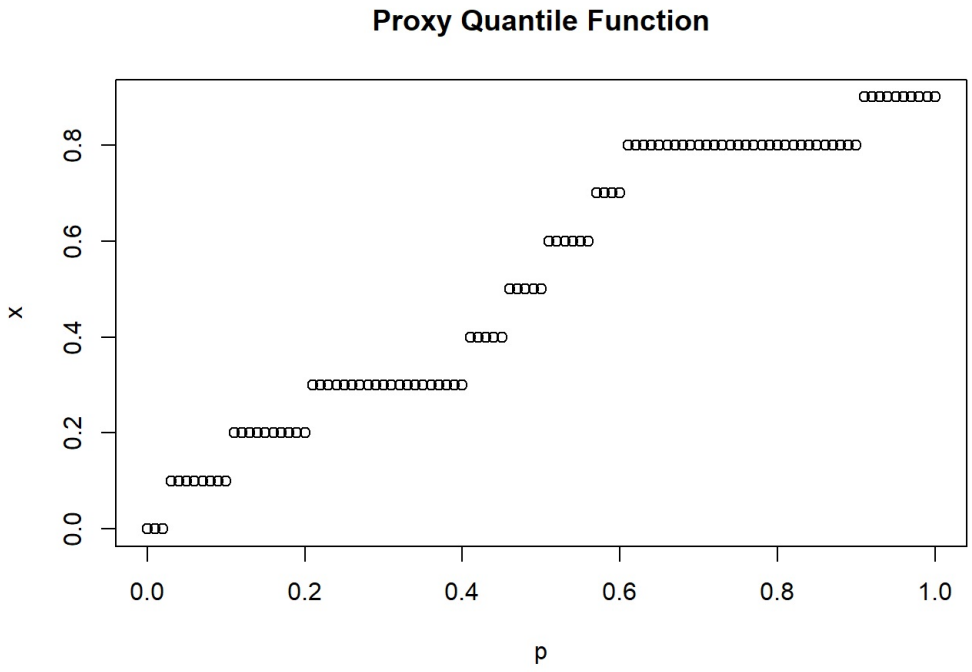
X is a continuous random variable with *CDF* $F_X(.)$. For any given level (*tail area*) $p \in [0; 1]$, the quantile function is defined as:
$Q(p) = inf\ \{x\ such\ that\ F_x(x) = p\}$ = the smallex value x such that $F_x(x) = P(X \leq x)$ is at least p.

```
## Quantile Function
proxy_quant <- function(p, pi_greco, h){
  ## Number of bins
  N <- ceiling(1/h)
  ## CDF
  cdf <- cumsum(pi_greco)

  ## Cumsum of the length of the bins
  new_h <- cumsum(rep(h, N))

  ## Cycle
  if (pi_greco[1] >= p){
    return(0)
  }else{
    for (j in 1:(N-1)){
      if((cdf[j] < p) && (cdf[j+1] >= p)){
        return(new_h[j])}}}}
```

```
## Test
q = sapply(x, function(x) proxy_quant(x, pi_greco, h))
plot(x, q,
     type = 'p',
     main = 'Proxy Quantile Function',
     xlab = 'p',
     ylab = 'x')
```



2.3

```
## Implementation
pi_greco <- function(h, a, b){

  N <- ceiling(1/h)

  bins <- c(0, cumsum(rep(h, N)))

  pi_vec <- rep(0, N)
  for (i in 1:N) {
    pi_vec[i] <- pbeta(bins[i+1], a, b) - pbeta(bins[i], a, b)
  }
  return(pi_vec)
}
```

```
## Check
## Quantile Beta
qbeta(pi_greco(h, 2, 2),2, 2)
```

```
##  [1] 0.1000000 0.1689651 0.2082082 0.2315342 0.2425928 0.2425928 0.2315342
##  [8] 0.2082082 0.1689651 0.1000000
```

```
## Check
qbeta(p = 0.3, 2, 2) - proxy_quant(h = 0.001, 0.3,
                                   pi_greco(h = 0.001, 2, 2))
```

```
## [1] 0.0002574911
```

```
qbeta(p = 0.5, 5, 2) - - proxy_quant(h = 0.01, 0.5,
                                   pi_greco(h = 0.01, 5, 2))
```

```
## [1] 1.46555
```

```
qbeta(p = 0.7, 2, 3) - proxy_quant(h = 0.1, 0.7,
                                   pi_greco(h = 0.1, 2, 3))
```

```
## [1] 0.008404755
```

```
## Check
abs(qbeta(p = 0.3, 2, 2) - proxy_quant(h = 0.001, 0.3,
                                       pi_greco(h = 0.001,
                                                2, 2)))
```

```
## [1] 0.0002574911
```

```
## Distance
wass_dist <- function(h, a ,b) {

  diff <- function(x, h, a, b) abs(qbeta(p = x ,a,b) - proxy_quant(h = h, x, pi_greco(h = h, a,b)))

  dist <- integrate(diff, 0,1, a = a, b = b, h = h)
  return(dist)
}
```

```
## Largest Binwidth
largest_h <- function(h_vec, a, b, epsilon){
  ws_dist <- rep(0, length(h_vec))

  for (i in 1:length(h_vec)){
    ws_dist[i] <- wass_dist(h_vec[i],a, b)$value
  }
  idx <- which(ws_dist <= epsilon)
  return(max(h_vec[idx]))
}
```

```
## Check
h_vec <- c(0.01, 0.02, 0.03, 0.04,
           0.05, 0.06, 0.07, 0.08, 0.09)
epsilon <- c(0.01, 0.02, 0.03, 0.04,
             0.002, 0.003,0.004, 0.005)
l_epsilon <- length(epsilon)
h_large <- rep(0, l_epsilon)

for(j in 1:l_epsilon){
  h_large[j] <- largest_h(h_vec, 2, 2, epsilon[j])
}

# Plot
plot(epsilon, h_large,
     type = 'b',
     main = "Largest binwidth(in function of epsilon)",
     xlab = "epsilon",
     ylab = "h")
```