1-.Theoretical/Kernel density function on a histogram

(Example 1.1-Exercises_Bank)

Histogram:

```
hist( data, breaks = whatever, xlab = "", ylab = "", main = "",
      prob = TRUE)
```

How to plot theoretical density function on a histogram:

```
data = r… (…)
sequence = seq( min(data), max(data), length = length(data))
y.values = d…(sequence,…)
lines(sequence, y.values, lwd = , col = )
```

d – Density

r – Realizations

p – probability

q - quantile

Ex)

```
data = rexp(1000, rate = 1/2)
sequence = seq( min(data), max(data), length = length(data))
y.values = dexp(sequence, rate = 1/2)
# Note: seq can have the parameter by= …
```

How to plot the Kernel Density Function on a histogram:

```
lines( density(data, bw = value),
      lwd = 2, col = "red")
```

Different ways for bw:

```
fx = density(data)
1-. density(data, bw = fx$bw*2)
2-. density(data, bw = fx$bw*0.5)
3-. density(data, bw = sd(data)/2)
```

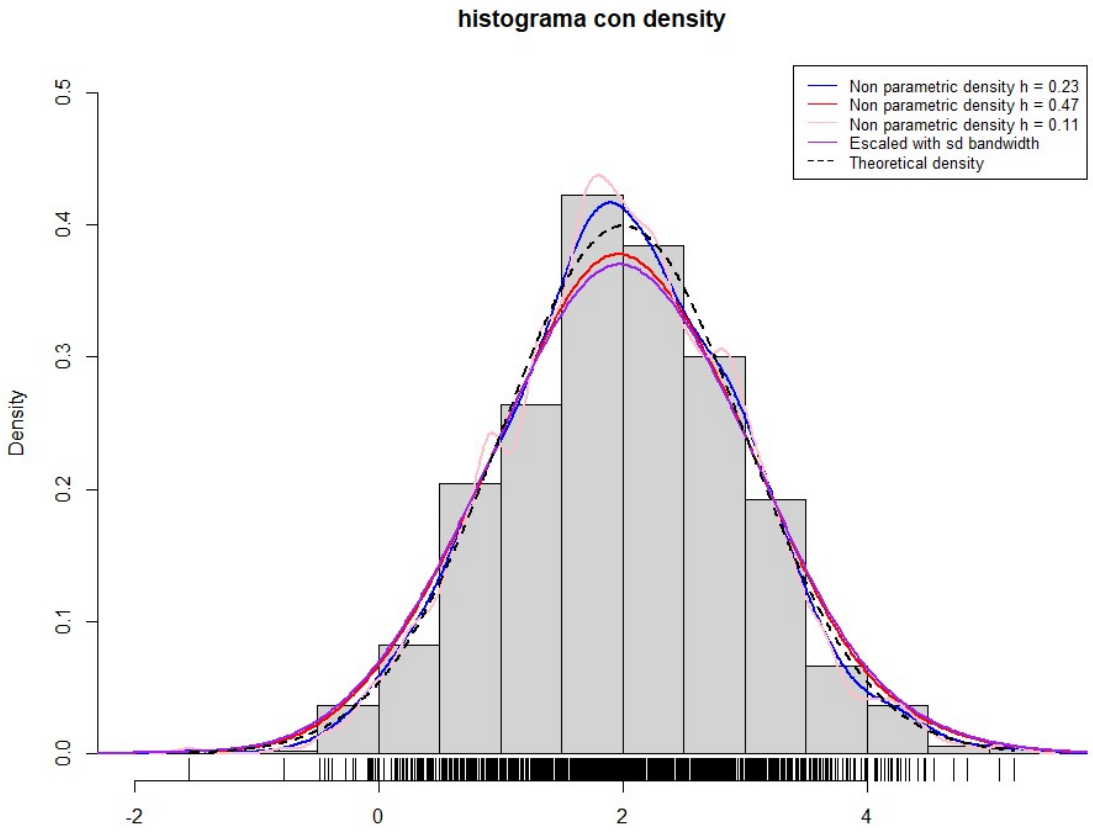
Note: more bw smoother, less bw squigglier..

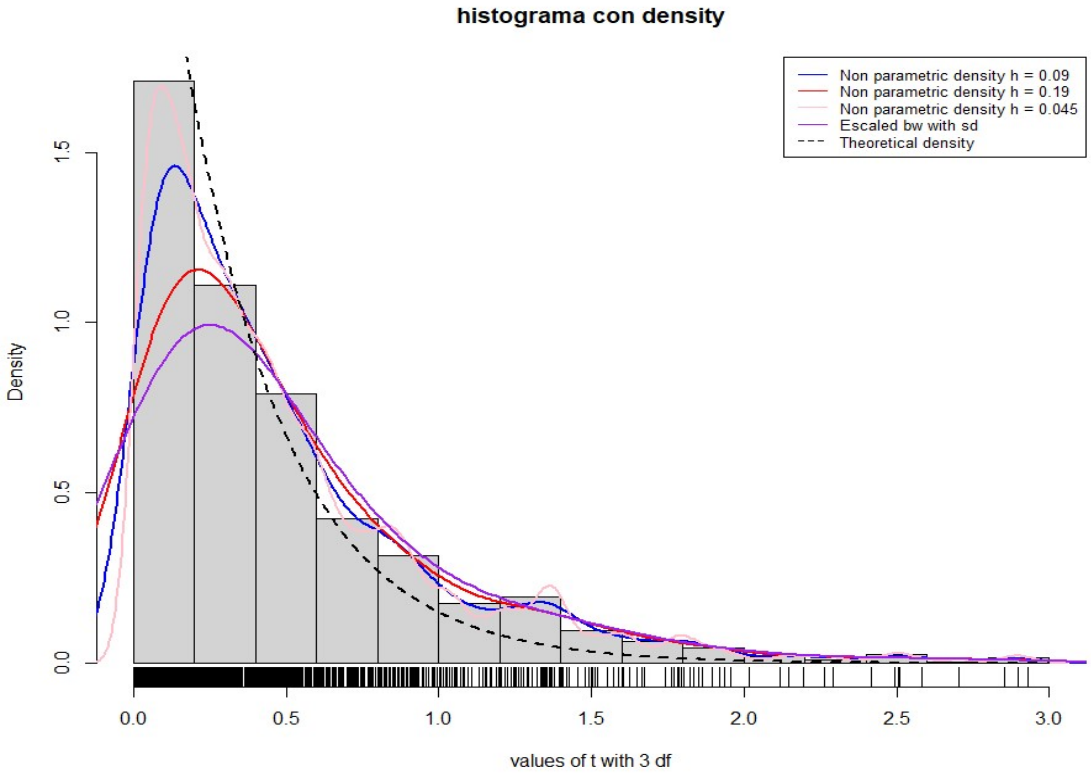
(Check Exercise 3 Chapter 1)

How to add a legend

```
legend( "position_legend",
      legend = c(names_in_legend),
      col = c(color_of_lines),
      lty = c(lines_types),
      cex = size_legend)
```

Note: Remember KDE is worse for distrubution like the Exponential(picture 2). That's why the exponential converges to infinity when it tends to 0. Conversely, it works very good for function such as normal (picture 1).





1-. Montecarlo:

Montecarlo Integral:

$$\int_a^b f(x)dx$$

1-. realizations = **runif**(M, a , b)

2-. We evaluate these realizations

```
for (i in 1:M){
      sum = 0
      sum = sum + f(realizations[i])
}
```

Montecarlo: we know ahead of calculating anything the underlying distribution of the data.

Basic Montecarlo Example:

```
M = 1000
storage = numeric(M) # Storing results for each montecarlo iteration
for (i in 1:M) {
      # We need to generate different noise in each iteration
      noise = rnorm(1000, mean = 10, sd = 2)
      # We add that noise to whatever, for example
      Y = true_theta*x + noise
      # And now we calculate what we are interested in…
      …
}
```

Note: remember Montecarlo is based on CLT (Central Limit Theorem). So that, the more M or the more N we have the better our estimation will be. (Check exercise 3 b) 2019)

Moreover, if we are trying to estimate the sd of a distribution

Sd(storage) = sd(distribution) / root(N) **(CLT)** (check 02_Rscrip_L14)

(We should know ahead of calculating anything sd of the distribution)

How to make a good boxplot

```
boxplot( data_1, data_2,… ,
      names = c("name_data_1", "name_data_2", ….),
      col = c("col_data_1","col:data_2",…),
      ylab = "",
      xlab = "",
      main = "")
```

Statistics of interest for Montecarlo

1-. Bias: estimated value – real value

2-. Percentage of bias: (estimated value – real value) / real value

3-. Variance/sd: var(storage) or sd(storage)

4-. MSE: Bias^2 + Variance or mean((estimated – real)^2)

Using abline

```
abline(h = horizontal_line, v = vertical_line, lwd = 2, col =  "red")
```

Storages variables

```
1-. storage = numeric(K)
2-. Storage = matrix(NA, nrow = nrow, ncol = ncol)
```

Non-parametric confidence interval

Ex) for a 95%

quantile(data, prob = c(0.025, 0.975))

2-. Bootstrap: We learn from the data without assiming an underlying distribution.

Bootstrap **relies on** the assumption that the sample appropiately represents the underlying distribution.

Bootstrap **takes an average of 63.4%** of the data in each iteration.

Example of an easy bootstrap:

```
B = 100
storage = numeric(B) or matrix(NA, nrow = nrow, ncol = ncol)
n = nrow(data)
for (b in 1:B){
      # We are generating a sample WITH REPLACEMENT
      temporal.index = sample(1:n, n, replace = TRUE)
      # We use the indeces now..
      temporal.data = data[temporal.index,]
      # We calculate whatever we wanted now…
      …
}
```

Caution: We generate the indeces just ONCE in each iteration

Useful command if they are asking for lm (inside the loop)

```
temp.lm = lm(y ~ x , data = dat, subset = temp.index)
```

Function apply

```
apply(whatever, 2(columns) or 1(rows), mean/sd/median…)
```

Statistics of interest for Bootstrap

1-. Bias: estimated value – **value (taking all data)**

2-. Percentage of bias: (estimated value – value_all_data) / value_all_ data

3-. Variance/sd: var(storage) or sd(storage)

4-. MSE: Bias^2 + Variance or mean((estimated – real)^2)

Bootstrap confidence intervals

- **Naïve or Non-Parametric interval:**

  (q_L, q_M) = quantile(storage, c(0.025,0.975)) – 95%

- **Studentized CI**

  Bootstrap_estimate +- $t_{N-1,1-alpha/2} * se(boostrap_{estimate})$

- **Bootstrap CI**

  $(2 * value_{all_{data}} - q_M, \quad 2 * value_{all_{data}} - q_L)$

  (check exercise 3 part c) Assignment 1)

3-. Cross Validation:

1-. LOO: using as many folds as your data's rows has

- We use only one point for testing in each iteration and n-1 points for training.
- We are OVERFITTING
- Each point is tested just once.

2-. K-FOLD CV: (most important)

- We have k-folds of the same lengths
- Each point is just tested once
- The samleer sample, the smallet k-fold
- We'd like to have 70% training 30% testing

An easy K-fold Cross Validation

```
n = nrow(data)
```

Shuffle the data

```
data = data[sample(1:n, n, replace = F),]
```

Number of folds

```
K = 10

folds = cut(1:n, K, labels = F)

storage = numeric/matrix…

# Loop all over folds

for (k in 1:K){

      train.set = which(folds != k)

      test.set = which(folds == k)

      # And now we make whatever…

      # But usually we fit a model with the training data and we check errors with the test set (predictions…)
```

3-. Repeated K-fold CV:

An example is as follows:

```
n = nrow(data)
R = 10 # repetitions
K = 10 # Number of folds
folds = cut(1:n, K, labels = F) # folds
storages =…

# First loop
for (r in 1:R){
      # Shuffle the data en each repetition
      Data = data[sample(1:n, n, replace = F),]
      for (k in 1:K){
            train.set = which(folds != k)
            test.set = which(folds == k)
            # And now we make whatever…
            # But usually we fit a model with the training data and we check errors with the test set (predictions…)
            # To store something changes
            storage[k + (r-1)*K] = whatever…
```

(Check exercise 3.7 Exercises Bank)

4-.Out-of-Bag Bootstrap:

Since in bootstrap we are generally taking 63.4% of the data. We will use 63.4 training set and the remaining for test.

(Example 03_Out_of_Bag_Boots_Kfold – Chapter 03)

1-. Polynomial Regression

We are constructing here a polynomial with ideally (N-1) degrees

R code:

```
polynomial = lm(y ~ x + I(x^2) + … + I(x^r))
```

Plotting it

```
lines(x, fitted(polynomial), …) # it has fitted values
```

Predicting with the polynomial

We use the coefficients

```
cc = coef(polynomial)
new = seq(min, max , length = l)
y.pred = cc[1] + cc[2]*new + cc[3]*new^2 + … + cc[r+1]*new^r
```

2-. NLS

R code:

```
model.nls = nls(y ~ formula,
      start = list(a = a1, b = b1, …)) # Parameters involved in the
formula
```

Formula ex:

```
y ~ a + b*exp(c*x)
y ~ log(a*b)/c
```

How to plot it

1-. lines(x, **fitted(nls.model)**,…)

2-. curve(coeficients*x, …)

Predict with it

We use the coefficients

cc = coef(nls.model)

new = seq(min, max , length = l)

y.pred = coefficients*x

3-. OPTIM

(Example in 02_Optim.R – Chapters)

It should have three functions associated:

1. **Model:** it will have the expression of your model.

Ex: (for a model y ~ exp(theta*x)

```
model <- function(x, theta){
  # Here if we had more than one parameter we could use
  # a = theta[1]
  # b = theta[2]
  return(theta*x)
}
```

2. **Criterion:** it will have the criterion to minimize such as MSE, RMSE or whatever…

Important: this function **must have theta as 1st parameter and return a single value**

Ex:

```
crit <- function(theta,x,y){
  # RSS in this case…
  return( sum( (y-model(x,theta))^2 )
}
```

3. **Optim:** we use optim function

Ex:

```
(optim.out <- optim(par=c(3), fn=crit, x=x, y=y,
                    method="L-BFGS-B",
                    lower=c(0.01), upper=c(3.05)))
```

Note: par could have been **par = c(theta_1, theta_2, …),** as many parameters as we have.

(See an example with more than one parameter in *Exercise-11 Shiny Apps)*

Result: we can obtain the estimations for the parameters using

optim.out$par

Plot it

Since we have the coefficients we can use curve

```
curve(expression using par coefficients, lwd = …, col = "")
```

4-. Grid Search

(Example 03_02_Grid_Search – Chapter 03)

In this scenario we are going to build a grid and then we are gonna evaluate or criterion to see which point in the grid **has the minimum value associated.**

R Code:

2-D grid search: when we have to optimize two parameters (i.e a and b)

L = length

```
a.grid = seq(min_value, max_value, length=L)
b.grid = seq(min_value, max_value, length=L)
```

```
# Storage
crt = matrix( NA , nrow = L, ncol = L)
# Loop all over the grid
for(i in 1:L){
  for(j in 1:L){
    crit[i,j] = crit(c(a.grid[i],a.grid[j], x, y)
  }
}
# We find the minimum
ij.best = which.min(crit.values)
ij.best = arrayInd(ij.best, .dim=dim(crt))
# grid search solutions:
a.best = a.grid[ij.best[1]]
b.best = b.grid[ij.best[2]]
# 3-D Plot grid search
persp(a.grid, b.grid, crt, col = "..", theta = angle)
contour(a.grid, b.grid, crt)
```

Note: this grid search is not only useful with optim it **can be used for splines** as well…

(Example 02_2018_2019 – Winter 2018)

5-. Regularization: they are techniques to reduce the overfitting of glm/lm. (i.e. they will increase a little the train_error but will reduce test_error)

(Exercise 07_Exercise.R Exercises Bank Chapter 3)

We are going to stand out **three regularization techniques.**

R Code FOR ALL OF THEM:

They **DO NOT HAVE FITTED VALUES**

```
library(glmnet)
```

CAUTION: all of these three model requires **x (predictors) TO BE A MATIRX**

```
xm = as.matrix(x)
```

We can tune the parameter lamda by grid search or using CV:

```
cv.l = cv.glmnet(xm[i.train,],y[i.train],grouped=FALSE)
```

Plot them

They have coefficients so go ahead with them with curve

Calculate training errors (since they do no have fitted values)

```
yh.r = predict(lasso/ridge/elastic ,newx=xm[train.set,])
```

Predict with them

```
yh.p = predict( lasso/ridge/elastic , newx = xm[-train.set,])
```

1-. Ridge

$$J_1(\theta) = \sum_{i=1}^{n}(Y_i - \theta_0 - \theta_1 X_{1,i} - \cdots - \theta_p X_{p,i})^2 + \lambda \sum_{j=1}^{p} \theta_j^2$$

This technique **does not drop from the model any variables.**

R Code:

```
ridge = glmnet(xm[train.set], y[train.set] ,
                    lambda=lam$lambda.min,
                    alpha = 0)
```

2-. LASSO

$$J_2(\theta) = \sum_{i=1}^{n}(Y_i - \theta_0 - \theta_1 X_{1,i} - \cdots - \theta_p X_{p,i})^2 + \lambda \sum_{j=1}^{p} |\theta_j|$$

This technique is **ABLE TO RULE OUT VARIABLES FROM THE MODEL.**

R Code:

```
lasso = glmnet(xm[train.set], y[train.set] ,
                    lambda=lam$lambda.min,
                    alpha = 1 # it is the default value)
```

3-. Elastic Net

$$J_3(\theta) = \sum_{i=1}^{n}(Y_i - \theta_0 - \theta_1 X_{1,i} - \cdots - \theta_p X_{p,i})^2 + \lambda_1 \sum_{j=1}^{p} \theta_j^2 + \lambda_2 \sum_{j=1}^{p} |\theta_j|$$

This technique **is capable to drop variables from the model.**

Is less strict than LASSO

R Code:

```
elastic = glmnet(xm[train.set], y[train.set] ,
                    lambda=lam$lambda.min,
                    alpha = values between 0 and 1)
```

In this chapter we are trying to estimate the underlying distribution. So all the techninques are **NON-PARAMETRIC.**

These models **WON'T HAVE COEFFICIENTS** since we are estimates the underlying distribution…

1-. Local Polynomial Regression

Although both of them has the capability of predict, **their predictions will be PLAIN** and so we would consider Splines insetead. (i.e. their prediction **are far away from reality**)

(03_B-Splines_P-Splines (and predicting from polynomial regression).R)

Lowess (1 vs 1)

CAUTION: lowess **sorts the x values**

It has fitted values lw$y

R code:

```
lw = lowess(x, y)
```

Plot it

```
# Remember that lowess sorts the x values…
lines(sort(x), low$y # fitted values, col='red', lwd=3)
```

Predict with lowess

We need to use approx since lowess is **not capabale to predict values outside its range**

Interpolation

```
pred.low = approx(low$x, low$y, xout=newx, rule=2)$y
```

(don't care about warnings)

Loess (1 vs many)

CAUTION: check **output values are in the same order as input values**

```
lo$x - x ; lo$y - y ; # It should give u all 0s
```

Span

By 0 to jittery

0.5 suitable

1 to smooth

It has fitted values lo$fitted

R code:

```
lo = loess(y ~ x1 + x2 + … + x_n, span = value, degree = degree)
```

Plot it

```
# Remember that lowess sorts the x values…
points(x, lo$fitted, col='navy', pch=15)
```

Predict with loess (Important)

Interpolation

```
pred.low = approx(lo$x, lo$fitted, xout=newx, rule=2)$y
```

(don't care about warnings)

Note: we could have used for loess predict

```
pred.lo = predict(lo, newdata=data.frame(x=newx))
```

2-. Splines

They are **very common when it comes to predict values outside our predictors' range** such as temporal series… . Since their **predictions are more realistic** than the local polynomial.

(Check that difference in: *03_B-Splines_P-Splines (and predicting from polynomial regression).R)*

There are mainly 2 types of splines: **B and P Splines**

B-Splines

Here is a cubic B-Spline with 3+J knots.

$$S(x) = \beta_0 + \beta_1 x + \beta_2 x^2 + \beta_3 x^3 + \sum_{j=1}^{J} \alpha_j (x - x_j)^{3+}$$

Total knots = degree of the spline + internal knots

R Code

```
library(splines)
```

Process:

1-. Set the B Spline (basis)

```
BM = bs(x, knots = c(knot_1, knot_2,…), df = degree_Bspline # default 3)
```

With the **knots** we **can control the tailance.**

Note: usually the **knots are the quantiles of x**

```
Ex) knots = quantile(x, prob = c(0.25,0.5,0.75))
```

Plot the basis

```
matplot(x,BM,xlab="x",ylab="Spline basis",main="total_knots-basis", t
= 'b')
```

2-Fit a linear model y ~ BM

```
lm.out = lm(mM~BM) # of course it has fitted values
coef(lm.out) # Will have as many coefficients as knots has
```

Plot it

```
points(x,fitted(lm.out),t='l',col="red", lwd = 3)
```

Predict with it

```
pred = predict(lm.out, newdata = data_frame)
```

P-splines

They give us even a better tailance than B-Splines.

R Code

They have fitted values

```
# Smooth spline with spar set with CV
p1 = smooth.spline(age, mF, cv = TRUE, df = degrees_freedom)
par = p1$spar
p2 = smooth.spline(age, mF, spar = par/2) # different spar
```

CAUTION: We need to check that they have the same order than x.

```
p1$x - x # should give u all 0s
```

If not we just need to **sort x**

Plot it

```
points(x, fitted(P.spline), pch=15, col='navy')
```

Predicting

```
pred.pspline = predict(P.spline, x=newx)
```

Other models

lm / glm

The unique difference between both is that **lm uses least squares** and **glm uses maximum likelihood.**

Predict

```
pred = predict(lm/glm.model , newdata = should_be_dataframe)
```