RDC

Roberto Alcover Couso

20/9/2018

Introduction

In this document we will explain how to implement RDC in python and the reasons behind the implementation decisions aswell as some hints of the theory behind it which is necessary to understand in order to implement this method.

Theory behind it

Estimation of Marginal Observation Ranks

Using uniform variables

Consider a random vector X = (X1 , . . . , Xd) with continuous marginal cumulative distri- bution functions (cdfs) Pi, 1≤i≤d. **U**=(U1,...,Ud):=(P1(X1),...,Pd(Xd)) has uniform marginals.(*Probability Integral Transform [Appendix 1.1]*)

Using the ecdfs

Let X1,...,Xn be iid random variables qith common cumulative distribution function P. Then, the empirical cumulative distribution function, defined as

$$P_n := \frac{1}{n}\sum_{i=1}^{n} I(X_i \leq x)$$

converges uniformly to P:

$$\|P_n - P\|_\infty = \sup_{x\in R} |P_n(x) - P(x)| \rightarrow^{a.s} 0$$

(*Appendix 1.2*)

Generation of Random Non-Linear Proyections

Thanks to Rahimi-Brecht axiom one can achive regressors of the form $\sum_{i=1}^{K} \alpha_i \phi(wi)$ when we randomize over the weights wi inside the non-linearities ϕ but optimize the lineat mixing coefficients α. Let $w_i \sim \mathcal{N}(\mathbf{0}, s\mathbf{I}), b_i \sim \mathcal{U}[-\pi, \pi]$. Given a data collection **X** = (x1,...,xn), we will denote by

$$\Phi(\mathbf{X}; k, s) := \begin{pmatrix} \phi(w_1^T x_1 + b_1) & ... & \phi(w_k^T x_1 + b_k) \\ . & . & . \\ . & . & . \\ . & . & . \\ \phi(w_1^T x_n + b_1) & ... & \phi(w_k^T x_n + b_k) \end{pmatrix}$$

Canonical Correlation Analysis
We will use CCA to compute the linear combinations of the augmented empirical copula that have maximal correlation. CCA searches for vectors a and b which maximize the correlation $\rho_{a^T X b^T Y}$
(*Appendix 1.3*)

Appendixes

Appendix 1.1 Probability Integral Transform

Let X with CDF F and Y = F(X), then Y follows a uniform distribution.
$F_Y(y) = P(Y) = P(F_X(x) \leq y) = P(X_X^{-1}(y)) = F_X(F_X^{-1}(y)) = y$

Appendix 1.2 Glivenko-Cantelli Theorem

$$F_n := \frac{1}{n} \sum_{i=1}^{n} I(X_i \leq x)$$

converges uniformly to P: Let X = $(x_0, ..., x_m)$ such that
$-\infty = x_0 < x_1 < ... < x_m = \infty$ and $F(x_j) - F(x_{j-1}) \leq \frac{1}{m}$
$F_n(x) - F(x) \leq F_n(x_j) - F(x_{j-1}) = F_n(x_j) - F(x_j) + \frac{1}{m}$
$F_n(x) - F(x) \geq F_n(x_{j-1}) - F(x_j) = F_n(x_{j-1}) - F(x_{j-1}) - \frac{1}{m}$

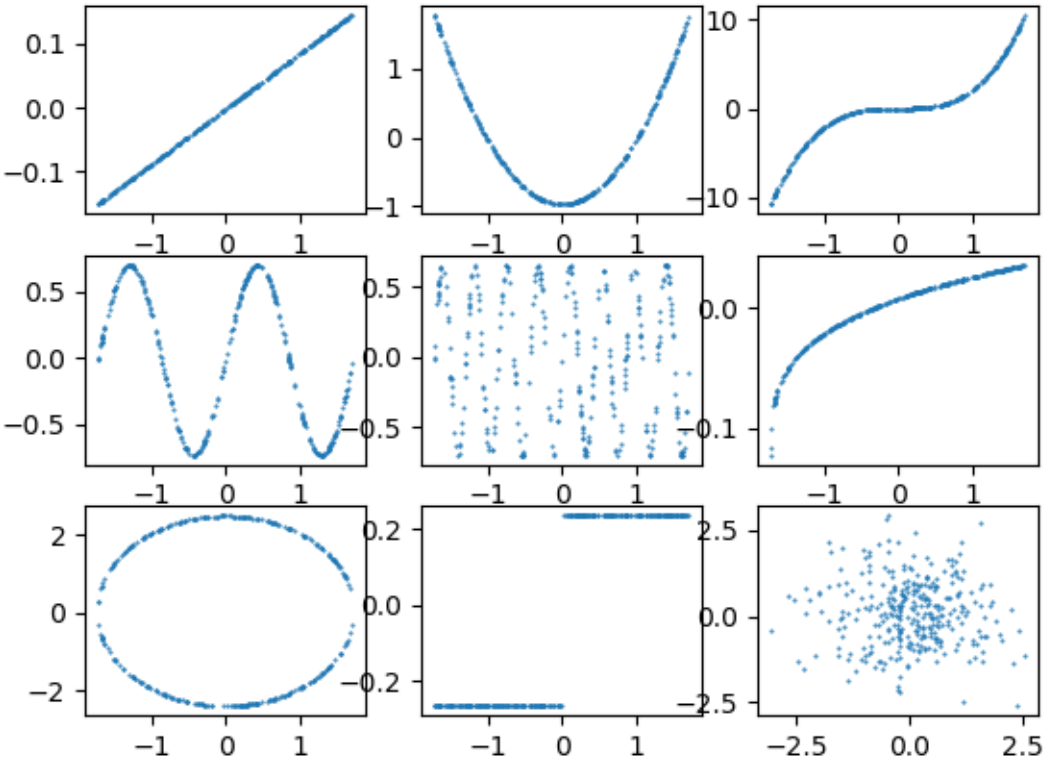
$$\|F_n - F\|_\infty = \sup_{x \in R} |F_n(x) - F(x)| \leq \max_{j \in 1,..,m} |F_n(x_j) - F(x_j)| + \frac{1}{m} \to 0(a.s)$$

Appendix 1.3 CCA

Let us consider the correlation $\phi(a, b)$ between the two prohections in more detail.
Suppose that:
$$\begin{pmatrix} X \\ Y \end{pmatrix} \sim \left(\begin{pmatrix} \mu \\ \nu \end{pmatrix} \quad \begin{pmatrix} \Sigma_{XX} & \Sigma_{XY} \\ \Sigma_{YX} & \Sigma_{YY} \end{pmatrix} \right)$$
Then:
$\phi(a, b) = \frac{a^T \Sigma_{XY} b}{(a^T \Sigma_{XX} a)^{1/2} (b^T \Sigma_{YY} b)^{1/2}}$
Which is easy to see that: $\phi(a, b) = \phi(ca, b)$ for any c. Therefore the problem of maximizing ϕ can be realiced under the constrains:
$a^T \Sigma_{XX} a = 1$ $b^T \Sigma_{YY} b = 1$.
First we need to define: $K = \Sigma_{XX}^{-1/2} \Sigma_{XY} \Sigma_{YY}^{-1/2}$ which its eigenvalues will be the canonical correlation coefficients.
$k = rank(K) = rank(\Sigma_{XY})$*, we will use this in the implementation*

Appendix 1.4 Python Code

In order to create the data which we will be applying RDC we will base on the numpy library. X $\sim \mathcal{U}[0,1]$ which will be implemented thoughout the function np.random.rand(). In order to generate different samples each execution we will provide a seed which is going to change every time the program is executed(for example current time). Y will be generated by applying different functions to the random vector X.



The code for the rdc itself its pretty simple:
First we will apply the ecdf function to the vectors (which is needed to be implemented separately in python) and concatenate the column full of ones in order to generate the bi later on.

```
lx = len(x)
ly = len(y)
x = np.concatenate((ecdf(x).reshape(-1,1),
np.ones(lx).reshape(-1,1)),axis=1)
y = np.concatenate((ecdf(y).reshape(-1,1),
np.ones(ly).reshape(-1,1)),axis=1)
nx = x.shape[1]
ny = y.shape[1]
```

Then we generate a matrix with the samples of the $w \sim \mathcal{N}(0,1)$ *in order to*

multiply it and generate the desired matrix explaned in the section Generation of Random Non-Linear Proyections
wx = norm(nx*k,s).reshape(nx,k)
wy = norm(ny*k,s).reshape(ny,k)
wxs = np.matmul(x,wx)
wys = np.matmul(y,wy)
we apply the function ϕ which is in this case is a sin
fX = np.sin(wxs)
fY = np.sin(wys)
cancor function will be explained later on in this brief
res = cancor(fX,fY,k)

Cancor function

As we see in the paper by David Lopez-Paz, Philip Hennig and Bernhard Schlkopf the correlation coefficient will be tightly related to the f,k and s parameters. While implementing this some problems related to numerical problems while calculating the inverse of Cxx and Cyy showed up. If the value of k is too big then the rank of fX or fY could be smaller than k, therefore we need to find the largest k which makes the eigenvalues real-valued. We will implement this through a binary search of k, although is not the most efficient way of solving this problem is the simplest way.

First we will calculate the covariance matrix
C = np.cov(np.hstack([x, y]).T)
Now we will apply the binary search for the optimum k value in which the Cxx and Cyy are not singular
k0 = k lb = 1 ub = k
Due to calculations of the optimum k sometimes becomes a double, therefore not being able to index a matrix
while True:
    k = int(k)
We will calculate the new cancor using the new values of k leaving behind some variations which due to numerical aproximation of functions by the computers may create some rows equal
    Cxx = C[:k,:k]
    Cyy = C[k0:k0+k, k0:k0+k]
    Cxy = C[:k, k0:k0+k]
    Cyx = C[k0:k0+k, :k]
    eigs = np.linalg.eigvals(np.dot(np.dot(np.linalg.inv(Cxx), Cxy),
    np.dot(np.linalg.inv(Cyy), Cyx)))
If k is too big, the values of the eigenvalues won't be real between 0 and 1
    if not (np.all(np.isreal(eigs)) and
        0 ¡= np.min(eigs) and
        np.max(eigs) ¡= 1):
            ub -= 1
            k = (ub + lb) / 2
            continue
    if lb == ub: break
    lb = k
    if ub == lb + 1:
            k = ub

else:

k = (ub + lb) / 2

return np.sqrt(np.max(eigs))

$$\begin{pmatrix} 1 & 0.5 \\ 0.5 & 1 \end{pmatrix}$$