Multivariate Analysis and Statistical Learning PC Algorithm's implementation

Authors: Alex Foglia, Tommaso Puccetti

Università degli Studi di Firenze

21/12/2018

Theoretical references (1)

- Bayesian Networks can be rappresented as a **directed acyclic graph (DAG)**
- ”acyclic” means that there are no paths starting from a node v that ends with v itself, $\forall v \in G$

Theoretical references (2)

Let $G = (V, E)$ be a DAG relative to a finite set $X = \{X_v, v \in V\}$ of casual variables, then:

$$\forall u, v \in V \; non \; adjacent \; | v \in nd(u) \Rightarrow$$

$$\Rightarrow u \perp\!\!\!\perp v | nd(u) - v$$

Where $nd(u)$ is the set of **non-descendant** n of n, that are all those nodes u' for which there is no path from u to u'.

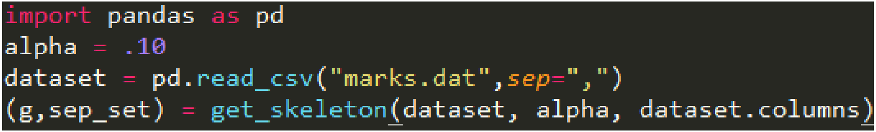
PC-Algorithm

Given a set of variables with a joint Gaussian probability distribution, it is possible to learn the DAG closer to the sample through the use of **PC-Algorithm**.
It is composed of two sub-functions that solve two different problems:

1. The construction of the skeleton (or **Moral Graph**)
2. The construction of the DAG from a given skeleton

Step one: read the dataset

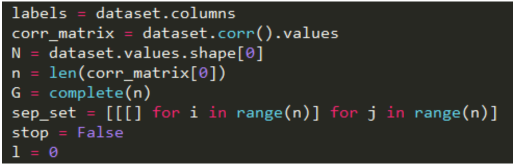
- import **pandas** library
- call **pandas.read_csv()** function to read dataset
- define **alpha**
- call **get_skeleton** on dataset and alpha as arguments



```
import pandas as pd
alpha = .10
dataset = pd.read_csv("marks.dat",sep=",")
(g,sep_set) = get_skeleton(dataset, alpha, dataset.columns)
```

Step two: initialization

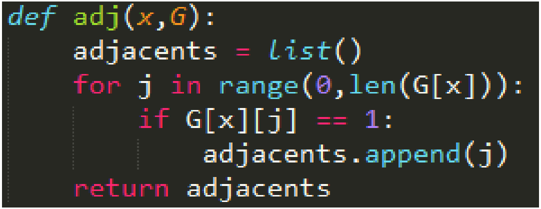
- read names of the dataset variables accessing **dataset.columns** field
- retrieve the correlation matrix of the given dataset with **dataset.corr().values**
- initialize **N,n** as the number of sampling and the number of variables
- initialize **G** as the complete graph of dimension n
- initalize the **separation_set** as a list of list
- initialize **l = 0**, **stop = false**



```
labels = dataset.columns
corr_matrix = dataset.corr().values
N = dataset.values.shape[0]
n = len(corr_matrix[0])
G = complete(n)
sep_set = [[[] for i in range(n)] for j in range(n)]
stop = False
l = 0
```

Step three: define adj function

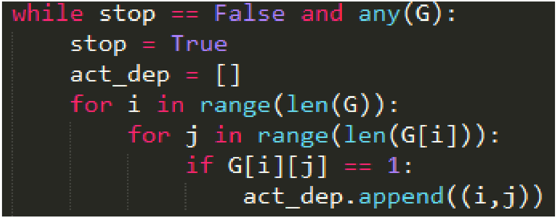
- define the **adj()** function in order to get the adjacents of a node in a given graph.



```
def adj(x,G):
    adjacents = list()
    for j in range(0,len(G[x])):
        if G[x][j] == 1:
            adjacents.append(j)
    return adjacents
```

Step four: how many variables are actually dependent?

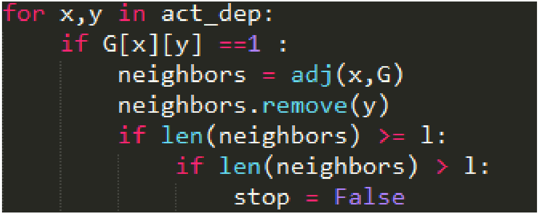
- set stop condition to true
- retrieve dependent variables: i,j are actually dependent if the adjacence matrix[i][j] is equal to 1
- call the set of dependent variables **act_dep**



```
while stop == False and any(G):
    stop = True
    act_dep = []
    for i in range(len(G)):
        for j in range(len(G[i])):
            if G[i][j] == 1:
                act_dep.append((i,j))
```

Step five: variables needed for independence test

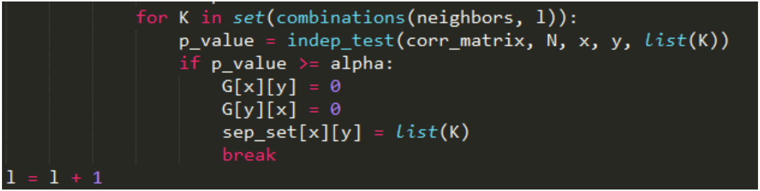
- for **x,y** in **act_dep**
- retrieve the **neighbors** of **x** calling the **adj()** function
- remove y from the **neighbors** set
- if **neighbors** set has dimension ≥ **l** then
    - if **neighbors** set has dimension > **l** go ahead



```
for x,y in act_dep:
    if G[x][y] ==1 :
        neighbors = adj(x,G)
        neighbors.remove(y)
        if len(neighbors) >= l:
            if len(neighbors) > l:
                stop = False
```

Step six: conditional independence test

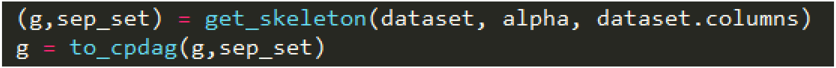
- foreach set **K** of neighbors of dimension **l**
- test independence of **x** and **y** given **K**
- if the p value is greater than **alpha**:
  - remove the edge x,y setting **G[x][y] = 0**
  - set **K** as the **separation_set[x][y]**



```
            for K in set(combinations(neighbors, l)):
                p_value = indep_test(corr_matrix, N, x, y, list(K))
                if p_value >= alpha:
                    G[x][y] = 0
                    G[y][x] = 0
                    sep_set[x][y] = list(K)
                    break
l = l + 1
```

Step eight: from the skeleton to the CPDAG

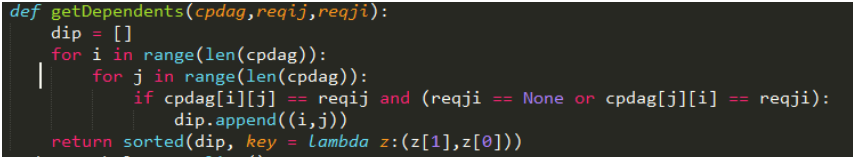
- return **G** and **separation_set**
- call **to_cpdag(G, separation_set)**



```
(g,sep_set) = get_skeleton(dataset, alpha, dataset.columns)
g = to_cpdag(g,sep_set)
```

Step eight: define the getIndependents() function

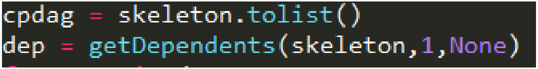
- define **getIndependents(adj_matrix,reqij, reqji)**
- this function retrieve all the variables i,j such that: **adj_matrix[i][j] == reqij** and **adj_matrix[j][i] == reqji**



```python
def getDependents(cpdag,reqij,reqji):
    dip = []
    for i in range(len(cpdag)):
        for j in range(len(cpdag)):
            if cpdag[i][j] == reqij and (reqji == None or cpdag[j][i] == reqji):
                dip.append((i,j))
    return sorted(dip, key = lambda z:(z[1],z[0]))
```

Step nine: CPDAG initialization

- set the **cpdag** as the skeleton
- set **dip** as the set of variables i,j for which exists an edge from i to j

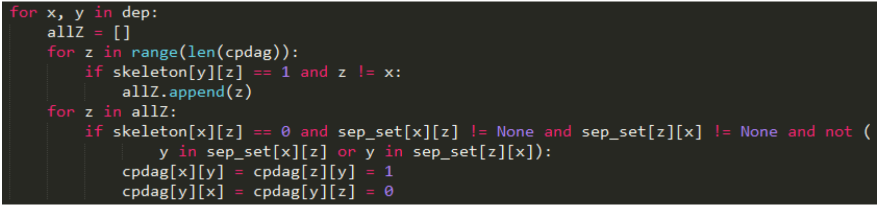


```
cpdag = skeleton.tolist()
dep = getDependents(skeleton,1,None)
```

Step ten: rule ”zero” (1)

- foreach pair x,y in **dip**:
- add to **allZ** all the variables **z** for which exists an egde from **z** to **j** and **z is not x**
- if:
  there is no edge between x and z
  there is a separation set between x and z
  there is a separation set between z and x
  y is not in separation set between x and z or **in separation set between z and x**, then:
- remove the edge from y to x and from z to y

Step ten: rule ”zero” (2)



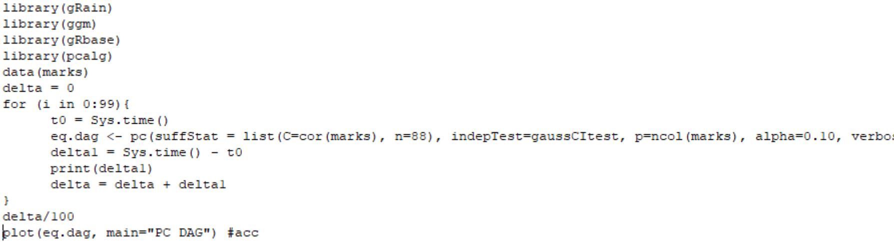
```
for x, y in dep:
    allZ = []
    for z in range(len(cpdag)):
        if skeleton[y][z] == 1 and z != x:
            allZ.append(z)
    for z in allZ:
        if skeleton[x][z] == 0 and sep_set[x][z] != None and sep_set[z][x] != None and not (
                y in sep_set[x][z] or y in sep_set[z][x]):
            cpdag[x][y] = cpdag[z][y] = 1
            cpdag[y][x] = cpdag[y][z] = 0
```

Step eleven: apply rules

- using the same logic we apply the known rules 1,2 and 3
- return the resulting cpdag
- using **matplotlib** and **networkx** we are able to plot the resulting cpdag

Python vs R

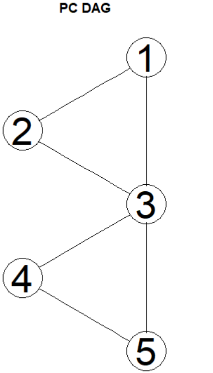
- consider this R code

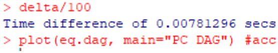


```
library(gRain)
library(ggm)
library(gRbase)
library(pcalg)
data(marks)
delta = 0
for (i in 0:99){
      t0 = Sys.time()
      eq.dag <- pc(suffStat = list(C=cor(marks), n=88), indepTest=gaussCItest, p=ncol(marks), alpha=0.10, verbos
      delta1 = Sys.time() - t0
      print(delta1)
      delta = delta + delta1
}
delta/100
plot(eq.dag, main="PC DAG") #acc
```

- it gives

Python vs R

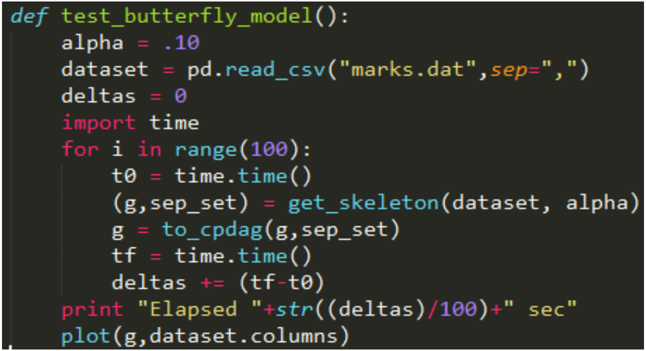




```
> delta/100
Time difference of 0.00781296 secs
> plot(eq.dag, main="PC DAG") #acc
```

Python vs R

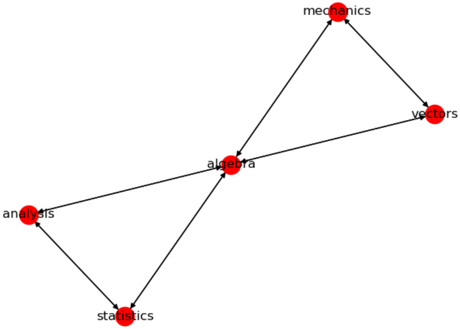
- consider this python code

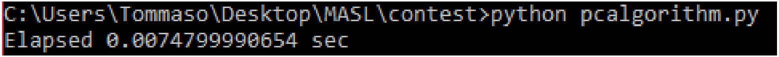


```python
def test_butterfly_model():
    alpha = .10
    dataset = pd.read_csv("marks.dat",sep=",")
    deltas = 0
    import time
    for i in range(100):
        t0 = time.time()
        (g,sep_set) = get_skeleton(dataset, alpha)
        g = to_cpdag(g,sep_set)
        tf = time.time()
        deltas += (tf-t0)
    print "Elapsed "+str((deltas)/100)+" sec"
    plot(g,dataset.columns)
```

- it gives:

Python vs R





```
C:\Users\Tommaso\Desktop\MASL\contest>python pcalgorithm.py
Elapsed 0.0074799990654 sec
```