MultiSampleSplitting

December 10, 2023

1 CONTEST MASL 2023: MULTI SAMPLE SPLITTING, Christian Mancini.

L'obbiettivo di questo Notebook è implementare i criteri per l'analisi in grandi dimensioni definite nel paper: High-Dimensional Inference: Confidence Intervals, p-Values and R-Software hdi.

Attraverso una simulazione MonteCarlo, proviamo a verificare ciò che è stato proposto.

[1]:
```
import numpy as np
import pandas as pd
import math
from sklearn.linear_model import ElasticNetCV
import statsmodels.api as sm
import warnings
warnings.filterwarnings("ignore")

seed = 420
np.random.seed(seed)
```

1.1 Data generating process

La generazione dei dati è eseguita campionando da una Normale. Specificando il vero modello da usare, possiamo costruire la variabile di risposta.

[2]:
```
def generate_data(dimention:int,observations:int)->pd.DataFrame:
    variables = np.array(range(dimention),dtype=float)
    data = pd.DataFrame()
    for idx, variable in enumerate(variables):
        mu, sigma = variable, math.sqrt(variable + 1)
        generated = np.random.normal(mu,sigma,observations)
        data[variable] = generated
    return data.copy()

def build_response(data:pd.DataFrame,colums_to_include:list,values:list,name =␣
 ↪"y")->pd.Series:
    if np.shape(colums_to_include) != np.shape(values):
        raise ValueError('Columns and values have different shapes')
    response = np.random.normal(0,1,np.shape(data)[0])
```

```
    for i, column in enumerate(colums_to_include):
        response += data[column]*values[i]
    return pd.Series(name=name,data = response)
```

Il vero modello è definito come $y = 16\beta_5 + 8\beta_{10} + 4\beta_{15} + 2\beta_{20} + \varepsilon$.

La i-esima variabile è campionata da $N\ (i, \sqrt{i})$.

Specifichiamo la dimensione p del dataset e quante osservazioni vogliamo.

Di seguito dichiariamo anche il vero modello per generare la risposta.

[3]:
```
p = 80
observations = 40
toUse = [5,10,15,20]
values = [16,8,4,2]

data = generate_data(dimention = p,
                     observations = observations)
y = build_response(data,toUse,values)
data["y"] = y
data.columns = data.columns.astype(str)
```

Le seguenti funzioni servono per eseguire la divisione del DataSet in due parti a seconda di una percentuale fornita.

Di default è 0.5, che è il valore riportato nel paper a cui facciamo riferimento.

[4]:
```
rng = np.random.default_rng(seed)

def sample_DataFrame(df:pd.DataFrame,sample_percentage_size:float)->pd.
 ↪DataFrame:
    """Funzione per campionare una percentuale dei dati dal DataSet."""
    if not (0.0 < sample_percentage_size < 1):
        raise AttributeError(f"sample_percentage_size must be in (0,1), got␣
 ↪{sample_percentage_size} instead.")
    sample_size = int(df.shape[0] * sample_percentage_size)
    sample_idx = rng.integers(low=0,high=df.shape[0],size=sample_size)
    return df.iloc[sample_idx]

def split_DataFrame(df:pd.DataFrame, sample_percentage_size=0.5, response_name␣
 ↪= "y")->tuple[pd.DataFrame,pd.DataFrame]:
    """Funzione per dividere il Dataset in 2 parti. Il primo Dataset ha␣
 ↪elementi pari alla percentuale fornita."""
    y = df[response_name]
    df = df.drop(response_name, axis=1)
    #Si prendono le osservazioni che hanno indice in sample_percentage_size
    df1 = sample_DataFrame(df,sample_percentage_size)
    df1[response_name] = y.iloc[df1.index]
    #Si prendono le osservazioni rimanenti
```

```
    df2 = df.iloc[~df1.index]
    df2[response_name] = y.iloc[~df1.index]
    return df1,df2
```

1.2 Passi della simulazione:

Ripeti B volte da 1 a 4

1. Dividere il DataSet in due parti uguali I_1 e I_2
2. Applicare uno stimatore regolarizzato (LASSO) a I_1 ed estrarre i coefficienti diversi da zero
3. Applicare a I_2 un OLS con soli i parametri ottenuti da I_2
4. Definito $s = |\{\beta \neq 0, \forall \beta \in I_1\}|$ e $P_{raw,j}$ il j-esimo p-value

- Correggere i p-value ottenuti con $P_{corr,j} = min(P_{raw,j} * s, 1)$

5. Aggregare i P_{corr} della simulazione utilizzando:

- media
- mediana
- min-max (solo per intervalli di confidenza)

  Gli intervalli di confidenza possono essere aggregati con gli stessi metodi del punto 5.

La ripetizione (solitamente con $B = 50$ o $B = 100$) serve per avere dei risultati riproducibili indipendentemente dal seme usato.

1.2.1 LASSO

Di seguito abbiamo la funzione per lo svolgimento del passo 2, ovvero applicare il LASSO e fornire i coefficienti diversi da zero.

[5]:
```
def applyLasso(data:pd.DataFrame,response = "y")-> tuple[int,float]:
    """Applica il LASSO e restituisce i coefficenti diversi da zero"""
    X = data.drop(response,axis=1)
    y = data[response]
    lasso = ElasticNetCV(cv=5,random_state=seed,l1_ratio = 1)
    lasso.fit(X, y)
    #print(f"Best alpha is: {lasso.alpha_}")
    return nonZeroCoeffiecients(X.columns,lasso.coef_)

def␣
 ↪nonZeroCoeffiecients(dataColumns,regression_coefficients)->list[tuple[str,float]]:
 ↪
    """Restituisce una lista di tuple contenente (nome,valore) dei coefficienti␣
 ↪diversi da zero"""
    coefficients = list(zip(dataColumns,regression_coefficients))
    coefficients_copy = coefficients.copy()
    for coefficient in coefficients_copy:
        if np.isclose(coefficient[1],0.0):
            coefficients.remove(coefficient)
    return coefficients
```

1.2.2 OLS

Con le funzioni sottostanti si risolve il passo 3, ovvero si utilizzano i coefficienti diversi da zero e si applica OLS.

[6]:
```
def useNonZeroCoefficient(data:pd.DataFrame,nonZeroCoefficients:
 ↪tuple[str,float],response = "y"):
    coefficients_name = [coefficient[0] for coefficient in nonZeroCoefficients]
    X = data.drop(response,axis=1)
    y = data["y"]
    X = X[coefficients_name]
    return X, y

def applyOLS(data:pd.DataFrame,nonZeroCoefficients:tuple[str,float],response =␣
 ↪"y"):
    X, y = useNonZeroCoefficient(data,nonZeroCoefficients)
    X = sm.add_constant(X)
    least_square = sm.OLS(y, X)
    least_square.fit()
    return least_square.fit()
```

1.2.3 Correzione dei p-values

Con le seguenti funzioni possiamo correggere il p-value della simulazione e salvarne il risultato.

Lo stesso viene fatto per gli intervalli di confidenza al 95%.

In entrambi i metodi il valore riferito alla costante è stato messo in coda per semplicità.

[7]:
```
def correct_pvalue(row,correction):
    return min(row*correction,1.0)

def updadte_pvalues(matrix,values,correction,simulation_number):
    pvalues_indexs = values.index.tolist()
    for index in pvalues_indexs:
        if index == "const":
            matrix[simulation_number][-1] =␣
 ↪correct_pvalue(values[index],correction)
        else:
            matrix[simulation_number][int(index)] =␣
 ↪correct_pvalue(values[index],correction)
    return matrix

def␣
 ↪update_confidence_intervals(lower_bound,upper_bound,values,simulation_number):
 ↪
    confidence_intervals_index = values.index.tolist()
    for index in confidence_intervals_index:
        if index == "const":
```

```
            lower_bound[simulation_number][-1] = values[0][index]
            upper_bound[simulation_number][-1] = values[1][index]
        else:
            lower_bound[simulation_number][int(index)] = values[0][index]
            upper_bound[simulation_number][int(index)] = values[1][index]

    return lower_bound,upper_bound
```

1.2.4 Aggregazione

Ottenuti i risultati della simulazione dobbiamo aggregare i risultati secondo il passo 5.

Verranno quindi usati le aggregazioni tramite:

- media
- mediana
- min - max (sensato solo per intervalli di confidenza)

[8]:
```
def aggregate_confidence_intervals(lower_bound,upper_bound,criteria =␣
↪"average"):
    if criteria == "average":
        return pd.DataFrame(
            [np.average(lower_bound,axis=0),
            np.average(upper_bound,axis=0)]).T.rename(columns={0:␣
↪"Average_Lower-bound",
                                                               1:␣
↪"Average_Upper-bound"})
    elif criteria == "median":
        return pd.DataFrame(
            [np.median(lower_bound,axis=0),
            np.median(upper_bound,axis=0)]).T.rename(columns={0:␣
↪"Median_Lower-bound",
                                                              1:␣
↪"Median_Upper-bound"})
    elif criteria == "min-max":
        return pd.DataFrame([np.min(lower_bound,axis=0),
                             np.max(lower_bound,axis=0),
                             np.min(upper_bound,axis=0),
                             np.max(upper_bound,axis=0)]).T.rename(columns={0:␣
↪"min-Lower-bound",
                                                                            1:␣
↪"max-Lower-bound",
                                                                            2:␣
↪"min-Upper-bound",
                                                                            3:␣
↪"max-Upper-bound"})
    else:
```

```
        raise ValueError(f"Criteria must be in ['average','median','min-max'],␣
↪got {criteria} instead.")

def aggregate_pvalues(pvalues,criteria = "average"):
    if criteria == "average":
        return pd.DataFrame(np.average(pvalues,axis=0)).rename(columns={0:␣
↪"Average"})
    elif criteria == "median":
        return pd.DataFrame(np.median(pvalues,axis=0)).rename(columns={0:␣
↪"Median"})
    elif criteria == "min-max":
        return pd.DataFrame([np.min(pvalues,axis=0), np.max(pvalues,axis=0)]).T.
↪rename(columns={0: "Min",
                                                                        ␣
↪                1: "Max"})
    else:
        raise ValueError(f"Criteria must be in ['average','median','min-max'],␣
↪got {criteria} instead.")
```

1.3 Simulazione con alta dimensionalità

Costruiti tutti i metodi di utilità possiamo effettuare la simulazione vera e propria con il codice nella cella sottostante

[9]:
```
# B governa il numero di simulazioni da effettuare (tipicamente 50 o 100).
B = 100

p_values = np.ones(shape=(B,p+1), dtype=float)
lower_bound = np.zeros(shape=(B,p+1), dtype=float)
upper_bound = np.zeros(shape=(B,p+1), dtype=float)

for simulation_number in range(B):
    I_1, I_2 = split_DataFrame(data)
    coefficients = applyLasso(I_1)
    #print(f"There are {len(coefficients)} in simulation {simulation_number}.")
    results = applyOLS(I_2,coefficients)
    p_values = updadte_pvalues(p_values,results.
↪pvalues,len(coefficients),simulation_number)
    lower_bound,upper_bound =␣
↪update_confidence_intervals(lower_bound,upper_bound,results.
↪conf_int(),simulation_number)
```

Adesso aggreghiamo gli intervalli di confidenza e i p-values per confrontarli ai veri valori scelti per la simulazione.

[10]:
```
CI = aggregate_confidence_intervals(lower_bound,upper_bound,criteria="average")
Pvalues = aggregate_pvalues(p_values,criteria="median")
```

```
print("---P-values---")
print(Pvalues.iloc[toUse])
print("------CI------")
print(CI.iloc[toUse])
```

```
---P-values---
    Median
5      1.0
10     1.0
15     1.0
20     1.0
------CI------
    Average_Lower-bound  Average_Upper-bound
5              0.000000             0.000000
10            -0.052591             0.216636
15            -0.070633             0.379283
20             0.000000             0.000000
```

Per le simulazioni effettuate, questo metodo non sembra riuscire a cogliere i veri valori dei parametri quando si tratta di grandi dimensioni.

Tuttavia, se effettuiamo la stessa simulazione riducendo la dimensione e aumentando le osservazioni, questo metodo riesce a cogliere i veri valori dei parametri.

1.4 Simulazione NON in alta dimensionalità

[11]:
```
p = 21
observations = 50
toUse = [5,10,15,20]
values = [16,8,4,2]

data = generate_data(dimention = p,
                     observations = observations)
y = build_response(data,toUse,values)
data["y"] = y
data.columns = data.columns.astype(str)

B = 100

p_values = np.ones(shape=(B,p+1), dtype=float)
lower_bound = np.zeros(shape=(B,p+1), dtype=float)
upper_bound = np.zeros(shape=(B,p+1), dtype=float)

for simulation_number in range(B):
    I_1, I_2 = split_DataFrame(data)
    coefficients = applyLasso(I_1)
    results = applyOLS(I_2,coefficients)
```

```
    p_values = updadte_pvalues(p_values,results.
 ↪pvalues,len(coefficients),simulation_number)
    lower_bound,upper_bound =␣
 ↪update_confidence_intervals(lower_bound,upper_bound,results.
 ↪conf_int(),simulation_number)

CI = aggregate_confidence_intervals(lower_bound,upper_bound,criteria="average")
Pvalues = aggregate_pvalues(p_values,criteria="median")

print("---P-values---")
print(Pvalues.iloc[toUse])
print("------CI------")
print(CI.iloc[toUse])
```

```
---P-values---
          Median
5   4.566518e-19
10  1.432201e-17
15  1.052908e-14
20  6.280502e-12
------CI------
    Average_Lower-bound  Average_Upper-bound
5             15.689360            16.295953
10             7.935545             8.358973
15             3.622208             3.954967
20             1.821724             2.144155
```

Per come sono state implementati i passi e per come sono stati generati i dati, i metodi proposti sembrerebbero non funzionare in dimensinalità alta, ma in dimensionalità "normali", colgono perfettamente i parametri scelti per la simulazione.