

submit

March 10, 2016

```
In [48]: import os
import sys
import glob
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from scipy import stats
import statsmodels.api as sm
import scipy.stats as ss
from functools import partial
from pandas import Series, DataFrame, Panel
from sklearn.linear_model import Lasso
from sklearn.linear_model import Ridge
from numba import jit, int32, int64, float32, float64

import multiprocessing
%matplotlib inline
%precision 4
```

```
Out[48]: u'%.4f'
```

```
In [49]: %load_ext rpy2.ipython
from rpy2.robj.ckit import importr
p1=importr('leaps')
p2=importr('stats')
```

The rpy2.ipython extension is already loaded. To reload it, use:

```
%reload_ext rpy2.ipython
```

```
In [50]: %load_ext cythonmagic
```

The cythonmagic extension is already loaded. To reload it, use:

```
%reload_ext cythonmagic
```

1 Background

I selected the Fast FSR Variable Selection research paper by Yujun Wu, Leonard A. Stefanski and Dennis D. Boos in 2009. Many variable selection procedures have been developed in the literature for linear regression models. A new and general approach, the False Selection Rate (FSR) method, to control variable selection is applicable to a broader class of regression models. The algorithm Fast FSR is a type of forward selection method and sequentially selects variables with fixed False Selection Rate (Usually target rate 0.05).

The earlier version of FSR variable selection method by Wu, Boos, and Stefanski (2007) requires the generation of the phony explanatory variables and the rate at which they enter a variable selection procedure is monitored as a function of a tuning parameter like α -to-enter of forward selection. This rate function is then

used to estimate the appropriate tuning parameter so that the average rate that uninformative variables enter selected models is controlled to be γ_0 , usually 0.05. However, the Fast FSR developed in this paper requires no phony variable generation, but achieves the same result. Basically, it depends on this mathematical formula to select variables:

$$K(\gamma_0) = \max\{i : \tilde{p}_i \leq \frac{(1+S) * \gamma_0}{k_T - S}, \text{ and, } \tilde{p}_i \leq \alpha_{max}\}$$

Fast FSR has competitive advantages among model selection method. It can give the parsimony model when the number of variables are bigger than the number of observations. Although lasso regression performs well in high dimension model selection, it is not competitive based on some criteria, where Fast FSR can compensate these disadvantages. Fast FSR has lower False Selection Rate than Lasso regression and have similar Model Error as lasso, which will be shown in the simulation study part. In addition, I implemented the optimization of the Fast FSR, Fast FSR bagging which is useful when a large of high correlated predictors are suspected in real case. Normal forward selection's and lasso prediction performance can degrade with highly correlated predictors.

- Flow of this project
 - Implement the Fast FSR and bagging Fast FSR
 - Two unit tests on the forward selection and get_fsr functions
 - Profile the performance of the algorithm: Pure python and vectorized python of function are written and the vectorized version improved the speed twice
 - High performance programming : Parallel programming by using multiple core. The result improves the speed twice compared the vectorized version.
 - Application and comparison: Compare the lasso and Fast FSR on the four simulated data set models. Based on the two criteria, False selection rate and Model error. Lasso and Fast FSR has similar Model error, however, the False Selection rate is lower in Fast FSR
 - Method optimization: Bagging Fast FSR: more applicable for the general data set: highly correlated predictors are suspected.
 - Reproducible analysis: I applied the Fast FSR and Bagging Fast FSR to the NCAA data, the result of which can be verified by the “Boos-Stefanski Variable Selection Home” <http://www4.stat.ncsu.edu/~boos/var.select/ncaa.data.orig.txt>. <http://www4.stat.ncsu.edu/~boos/var.select/bag.ncaa.ex.txt>

2 Implement

- Implement main functions: Fast FSR
- Description of fsr_fast_vectorized:
 - Input observation array and reponse y
 - Based on the forward selection function and selection rule as follows, variables are selected

$$K(\gamma_0) = \max\{i : \tilde{p}_i \leq \frac{(1+S) * \gamma_0}{k_T - S}, \text{ and, } \tilde{p}_i \leq \alpha_{max}\}$$

- Returned linear regression model on the selected variables, model size, name of the selected variables and false selection rate.

```
In [51]: def fsr_fast_vectorized(x,y):
          gam0=0.05
          digits = 4
          m = x.shape[1]
          n = x.shape[0]
          if(m >= n):
              m1=n-5
```

```

else:
    m1=m
    vm = range(m1)
    pvm = np.zeros(m1) # to create pvm below
    out_x = p1.regsbsets(x,y,method="forward") # foward selection by r function
    rss = out_x[9]
    nn = out_x[26][0]
    n_rss = np.array(range(len(rss)-1))
    q = [(rss[i]-rss[i+1])*(nn-i-2)/rss[i+1] for i in n_rss]
    rvf = ss.f(1,nn-n_rss-2)
    orig = np.array(1-rvf.cdf(q))
    for i in range(m1): # sequentially get max of pvalues
        pvm[i] = np.max(orig[0:i+1])
    alpha = [0]+pvm
    S = np.zeros(len(alpha)) # Include number of true entering in orig.
    for ia in range(1,len(alpha)): #loop through alpha values for S=size and size of models
        S[ia] = sum([pvm[i] <= alpha[ia] for i in range(len(pvm))])
    ghat = (m-S)*alpha/(1+S)
    alphamax = alpha[np.argmax(ghat)] # Got index of largest ghat
    ind = np.zeros(len(ghat))
    ind = np.where((ghat<gam0)&(alpha <=alphamax),1,0)
    Sind = S[np.max(np.where(np.array(ind)>0))] # model size with ghat just below gam0
    alphahat_fast = (1+Sind)*gam0/(m-Sind)
    size1=np.sum(np.array(pvm)<=alphahat_fast)+1 # size of model including intercept
    x=x[list(x.columns.values[list((np.array(out_x[7])-2)[1:size1]))]]
    x=sm.add_constant(x) # linear regression on the selected variables
    if(size1>1):
        x_ind=(np.array(out_x[7])-1)[1:size1]
    else:
        x_ind=0
    if (size1==1):
        mod = np.mean(y)
    else:
        mod = sm.OLS(y, x).fit()

    return mod,size1-1,x_ind,alphahat_fast

```

- FSRR_vectorized function description:
- This function returns the false selection rates for n iterations.
- Input: - target: true predictors under the simulated model. method: lasso variable selection or Fast FSR model selection. model : four true models can be selected. n : the number of iterations.
- By calling model number,the corresponding true model is generated with observation matrix and reponse variables. Then, it will generated all quadratic term for all the variables. By applying the lasso or Fast FSR, it will select some important variables. Then,get_fsr function can calculated the false selection rate by comparing the true variables and the selected variables.
- Saved each iteration false selection rate

```

In [52]: def FSRR_vectorized (target,method,model,n):
    l =[]
    for i in range(n):
        x = np.array(np.random.normal(1, 1, 21*1500).reshape(1500,21))
        if (model==1):
            y = x[:,0]
        if (model==2):

```

```

        b = np.array([9,4,1,9,4,1])
        y = np.dot(x[:,0:6],b)
    if (model==3):
        b = np.array([25,16,9,4,1,25,16,9,4,1])
        y = np.dot(x[:,0:10],b)
    if (model==4):
        b = np.array([45,36,25,16,9,4,1,45,36,25,16,9,4,1])
        y = np.dot(x[:,0:14],b)
    if (model==5):
        x[:,2] = x[:,3]
        x[:,4] = x[:,5]
        b = np.array([9,4,1,9,4,1])
        y = np.dot(x[:,0:6],b)
    quad = (x[:,0:20])**2
    x = np.concatenate((x,quad),axis=1)
    x = pd.DataFrame(x)
    m = method(x,y)
    L = get_fsr(target,method,x,y)
    l.append(L)
return l

```

3 Unit Test

- Unit Tests on functions forward_selection and get_fsr
- Unit test of forward_selection: The true linear relationship should be $y = b + 10x_1 + 200x_2 + 0.5x_3 + 0.01x_4 + 0.001x_5$. This forward selection function should select variables from the most related to less related. Thus, the returned result should be 1,3,2,4,5,6 where 1 denotes the intercept

```

In [17]: def foward_selection(x,y):
    out_x = p1.regsbsets(x,y,method="forward")
    rss = out_x[9]
    nn = out_x[26][0]
    r_7 = out_x[7]
    q = [(rss[i]-rss[i+1])*(nn-i-2)/rss[i+1] for i in range(len(rss)-1)]
    rvf = [ ss.f(1,nn-i-2) for i in range(len(rss)-1)]
    orig = [1-rvf[i].cdf(q[i]) for i in range(len(rss)-1)]
    return orig,r_7
    x = pd.DataFrame(np.random.normal(1, 1,5*1000).reshape(1000,5))
    b = np.array([10,200,0.5,0.01,0.001])
    y = np.dot(x,b)
    print foward_selection(x,y)[1]

```

```
[1] 1 3 2 4 5 6
```

```

In [53]: def get_fsr (target,method,x,y):
    m = method(x,y)
    if (len(m) == 4 and m[1]==0):
        m = []
    if (len(m) == 4 and m[1]!=0):
        m = m[2]
    I = set(target)&set(m)
    L = (len(m)-len(I))/(1.0+len(m))
    return L

```

4 Profiling: Naive Version and Vectoried Verison

- First two naive version functions are written. Then I profile the naive functions and find that it is unnecessary to do the Cythoned verion. So, I did the vectorized version of the functions. The speed of the vectorized verison improves twice as much as the pure python.

```
In [24]: ! pip install --pre line-profiler &> /dev/null
! pip install psutil &> /dev/null
! pip install memory_profiler &> /dev/null
```

```
In [19]: def Naive_Fast_FSR(x,y):
    gam0=0.05
    digits = 4
    pl = 1
    m = x.shape[1]
    n = x.shape[0]
    if(m >= n):
        m1=n-5
    else:
        m1=m
    vm = range(m1)
    # if only partially named columns corrects for no colnames
    pvm = [0] * m1
    out_x = pl.regsbsets(x,y,method="forward")
    rss = out_x[9]
    nn = out_x[26][0]
    q = [(rss[i]-rss[i+1])*(nn-i-2)/rss[i+1] for i in range(len(rss)-1)]
    rvf = [ ss.f(1,nn-i-2) for i in range(len(rss)-1)]
    orig = [1-rvf[i].cdf(q[i]) for i in range(len(rss)-1)]
    # sequential max of pvalues
    for i in range(m1):
        pvm[i] = max(orig[0:i+1])
    alpha = [0]+pvm
    ng = len(alpha)
    # will contain num. of true entering in orig
    S = [0] * ng
    # loop through alpha values for S=size
    for ia in range(1,ng):
        S[ia] = sum([pvm[i] <= alpha[ia] for i in range(len(pvm))])
        ghat = [(m-S[i])*alpha[i]/(1+S[i]) for i in range(len(alpha))]
        alphamax = alpha[np.argmax(ghat)]
        ind = [0]*len(ghat)
        ind = [ 1 if ghat[i]<gam0 and alpha[i]<=alphamax else 0 for i in range(len(ghat))]
        Sind = S[np.max(np.where(np.array(ind)>0))]
        alphahat_fast = (1+Sind)*gam0/(m-Sind)
        size1=np.sum(np.array(pvm)<=alphahat_fast)+1
        x=x[list(x.columns.values[list((np.array(out_x[7])-2)[1:size1])])]
        x=sm.add_constant(x)
        if(size1>1):
            x_ind=(np.array(out_x[7])-1)[1:size1]
        else:
            x_ind=0
        if (size1==1):
            mod = np.mean(y)
        else:
```

```

        mod = sm.OLS(y, x).fit()
    return mod,size1-1,x_ind,alpha_hat_fast

```

```

In [20]: def Naive_FSRR (target,method,model,n):
    l = []
    for i in range(n):
        x = pd.DataFrame(np.random.normal(1, 1, 21*1500).reshape(1500,21))
        if (model==1):
            y = x.ix[:,1]
        if (model==2):
            y = 9*x.ix[:,0]+4*x.ix[:,1]+x.ix[:,2]+9*x.ix[:,3]+4*x.ix[:,4]+x.ix[:,5]
        if (model==3):
            y = 25*x.ix[:,0]+16*x.ix[:,1]+9*x.ix[:,2]+4*x.ix[:,3]+1*x.ix[:,4]+25*x.ix[:,5]+16*x.ix[:,6]
        if (model==4):
            y = 45*x.ix[:,0]+36*x.ix[:,1]+25*x.ix[:,2]+16*x.ix[:,3]+9*x.ix[:,4]+4*x.ix[:,5]+x.ix[:,6]
        if (model==5):
            x.ix[:,2] = 2*x.ix[:,3]
            x.ix[:,4] = 3*x.ix[:,5]
            y = 9*x.ix[:,5]+4*x.ix[:,6]+x.ix[:,7]+9*x.ix[:,12]+4*x.ix[:,13]+x.ix[:,14]
        quad = (x.ix[:,0:20])**2
        x = np.concatenate((x,quad),axis=1)
        x = pd.DataFrame(x)
        m = method(x,y)
        L = get_fsr(target,method,x,y)
        l.append(L)
    return l

```

```

In [21]: %load_ext line_profiler

```

```

In [22]: x = pd.DataFrame(np.random.normal(1, 1, 21*150000).reshape(150000,21))
        y = 45*x.ix[:,0]+36*x.ix[:,1]+25*x.ix[:,2]+16*x.ix[:,3]+9*x.ix[:,4]+4*x.ix[:,5]+x.ix[:,6]+45*x.ix[:,7]

```

```

In [23]: %lprun -f fsr_fast_vectorized fsr_fast_vectorized(x,y)

```

```

In [32]: %lprun -f Naive_Fast_FSR Naive_Fast_FSR(x,y)

```

```

In [33]: %lprun -f FSRR_vectorized FSRR_vectorized([0,1,2,3,4,5,6,7,8,9,10,11,12,13],fsr_fast_vectorized(x,y))

```

```

In [24]: %lprun -f Naive_FSRR Naive_FSRR([0,1,2,3,4,5,6,7,8,9,10,11,12,13],Naive_Fast_FSR,4,100)

```

- Time comparison for pure python and vectorized python

```

In [25]: n =100
        %timeit Naive_FSRR([0,1,2,3,4,5,6,7,8,9,10,11,12,13],Naive_Fast_FSR,4,n)

```

1 loops, best of 3: 17.7 s per loop

```

In [26]: %timeit FSRR_vectorized([0,1,2,3,4,5,6,7,8,9,10,11,12,13],fsr_fast_vectorized,4,n)

```

1 loops, best of 3: 9.15 s per loop

5 High Performance : Parallel Programming

- In this part, I use the parallel programming and it improves the speed twice as much as vectorized python and fourth times as the pure python

```
In [54]: def pi_multiprocessing1(target,method,model,n):
        """Split a job of length n into num_procs pieces."""
        import multiprocessing
        m = multiprocessing.cpu_count()
        pool = multiprocessing.Pool(m)
        mapfunc = partial(FSRR_vectorized,target,method,model)
        results = pool.map(mapfunc,[n/m]*m)
        pool.close()
        return np.mean(results)
```

- The multiple core programming based on the vectorized python improves the speed twice compared to the vectorized python and four times compared to the pure python.

```
In [39]: %timeit pi_multiprocessing1([0,1,2,3,4,5,6,7,8,9,10,11,12,13],fsr_fast_vectorized,4,n)
```

1 loops, best of 3: 4.62 s per loop

6 Application and comparison

- Comparison among lasso, Fast_FSR based on two criteria: Model Error Ratio and False Selection Rate by the simulated data
- In this simulation study, I simulated 100 data points with 42 variables. Four models are simulated: H1: First variable is non-zero. H2: 6 variables are non-zeros at variables 1-6 with values (9,4,1,9,4,1). H3: 10 variables are non-zeros at variables 1-9 with values (25,16,9,4,1,25,16,9,4,1). H4: 14 variables are non-zeros at variables 1-14 with value (49, 36, 25, 16, 9, 4, 1,49, 36, 25, 16, 9, 4, 1)
- Result: Under each model, False Selection rate of Fast_FSR is higher than that of Lasso and the Model Error for lasso and Fast_FSR are close.

```
In [55]: def lasso_fit (x,y):
        alpha =0.5
        lasso = Lasso(alpha=alpha, tol=0.001)
        y_coef_lasso = lasso.fit(x, y).coef_
        lasso_index = np.where(y_coef_lasso != 0)[0]+1
        return lasso_index
```

- Model 1

```
In [56]: target =[1]
        n = int(100)
```

```
print "LASSO False Selection Rate is",pi_multiprocessing1(target,lasso_fit,1,n)
print "False Selection Rate of Fast FSR is",pi_multiprocessing1(target,fsr_fast_vectorized,1,n)
```

LASSO False Selection Rate is 0.423333333333
False Selection Rate of Fast FSR is 0.12

- Model 2

```
In [57]: n = int(100)
        target = [0,1,2,3,4,5]
```

```
print "LASSO False Selection Rate is",pi_multiprocessing1(target,lasso_fit,2,n)
print "False Selection Rate of Fast FSR is",pi_multiprocessing1(target,fsr_fast_vectorized,2,n)
```

LASSO False Selection Rate is 0.537610722611
False Selection Rate of Fast FSR is 0.190285714286

- Model 3

```
In [58]: n = int(100)
         target = [0,1,2,3,4,5,6,7,8,9]

         print "LASSO False Selection Rate is",pi_multiprocessing1(target,lasso_fit,3,n)
         print "False Selection Rate of Fast FSR is",pi_multiprocessing1(target,fsr_fast_vectorized,3,n)

LASSO False Selection Rate is 0.523977443609
False Selection Rate of Fast FSR is 0.154335664336
```

- Model 4

```
In [59]: n = int(100)
         target = [0,1,2,3,4,5,6,7,8,9,10,11,12,13]
         print "LASSO False Selection Rate is",pi_multiprocessing1(target,lasso_fit,4,n)
         print "False Selection Rate of Fast FSR is",pi_multiprocessing1(target,fsr_fast_vectorized,4,n)

LASSO False Selection Rate is 0.514347746762
False Selection Rate of Fast FSR is 0.113767973856
```

7 Extension for Fast_FSR: Bagging Fast FSR

- Bagging FSR: When the data are highly correlated, the False selection rate will be very high for normal Fast FSR. However, by implementing the Bagging Fast FSR, it will reduce the False Selection rate to the target level which makes the method functional. See the reproducible analysis on the real data set by both normal Fast FSR and bagging Fast FSR.

```
In [45]: def bag_fsr(x,y,B,gam0,method,digits):
         m = x.shape[1]
         n = x.shape[0]
         hold = np.zeros((B,m+1))      # holds coefficients
         hold = pd.DataFrame(hold)
         alphahat = [0] * B             # holds alphahats
         size = [0] * B
         for i in range(B):
             index = np.random.choice(n, n)
             out = method(x.ix[index,:],y.ix[index])
             if out[1]>0:
                 hold.iloc[i,out[2]] = np.array(out[0].params)[1:(len(out[2])+1)]
                 hold.iloc[i,0] = out[0].params[0]
                 alphahat[i] = out[3]
                 size[i] = out[1]
         hold[np.isnan(hold)]=0
         para_av = np.mean(hold,0)
         para_sd = [0]*(m+1)
         para_sd = np.var(hold,0)**0.5
         amean = np.mean(alphahat)
         sizem = np.mean(size)
         pred = np.matrix(x)*np.transpose(np.matrix(para_av[1:]))+para_av[0]
         return para_av,amean,sizem
```

8 Reproducible analysis

- NCCA data: Result here is same as the result on the original website which can be referred to <http://www4.stat.ncsu.edu/~boos/var.select/fsr.fast.ncaa.ex.txt>


```
In [46]: df = pd.read_csv('ncaa.data2.txt',delim_whitespace=True)
x = df.ix[:,0:19]
y = df.ix[:,19]
fsr_fast_vectorized(x,y)[0].summary()
```

```
Out[46]: <class 'statsmodels.iolib.summary.Summary'>
```

```
"""
                                OLS Regression Results
=====
Dep. Variable:                  y    R-squared:                0.811
Model:                            OLS    Adj. R-squared:         0.800
Method:                    Least Squares    F-statistic:            75.50
Date:                Thu, 30 Apr 2015    Prob (F-statistic):    2.49e-30
Time:                21:05:17    Log-Likelihood:        -315.88
No. Observations:                94    AIC:                   643.8
Df Residuals:                    88    BIC:                   659.0
Df Model:                        5
=====
                        coef    std err          t      P>|t|      [95.0% Conf. Int.]
-----
const                -42.1069      8.990     -4.684      0.000     -59.972    -24.242
x2                   3.4714      0.467      7.428      0.000      2.543      4.400
x3                   0.2391      0.076      3.163      0.002      0.089      0.389
x5                   0.2787      0.078      3.582      0.001      0.124      0.433
x4                   0.6770      0.195      3.475      0.001      0.290      1.064
x7                  -2.5913      0.832     -3.115      0.002     -4.245     -0.938
=====
Omnibus:                    5.624    Durbin-Watson:           1.718
Prob(Omnibus):              0.060    Jarque-Bera (JB):         3.905
Skew:                      0.351    Prob(JB):                 0.142
Kurtosis:                   2.290    Cond. No.                  620.
=====
"""
```

- Bagging Fast FSR on the NCCA data. My bagging Fast FSR is the same as the result on the original website, which can be reached at <http://www4.stat.ncsu.edu/~boos/var.select/bag.ncaa.ex.txt>

```
In [47]: x = df.ix[:,0:19]
y = df.ix[:,19]
result = bag_fsr(x,y,100,0.05,fsr_fast_vectorized,4)
print "Coefficient estimates",result[0]
print "Mean of estimated alpha-to-enter",result[1]
print "Mean size of selected model",result[2]
```

```
Coefficient estimates 0    -45.305210
1      0.019519
2      2.894766
3      0.199894
4      0.763531
5      0.207777
6      0.110935
7     -2.570658
8     -0.266395
9      1.303161
10     -0.000067
```

```
11      0.098644
12     -0.138039
13      0.033502
14      0.035500
15     -0.000000
16      0.054006
17     -0.033240
18     -0.020751
19      0.015739
dtype: float64
Mean of estimated alpha-to-enter 0.0417760931879
Mean size of selected model 7.45
```

```
In [ ]:
```