```
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
```

```
u'%.4f'
```

```
%load_ext rpy2.ipython
from rpy2.robjects.packages import importr
p1=importr('leaps')
p2=importr('stats')
```

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

```
%load_ext cythonmagic
```

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

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 verison 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 γ0, usually 0.05. However, the Fast FSR developed in this paper requires no phony variable generation, but achieves the same result. Bascially, it depends on this mathematical formula to select variables:

$$ K(\gamma_0) = max \{i :\tilde{p_i} <= \frac{(1 + S)* \gamma_0}{k_{T} - S}, and, \tilde{p_i} <= \alpha_{max}\}$$

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 higly 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 : Parellel 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 to the NCAA data which is provided by the "Boos-Stefanski Variable Selection Home" http://www4.stat.ncsu.edu/~boos/var.select/ncaa.data.orig.txt.

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
    $$ K(\gamma_0) = max \{i :\tilde{p_i} <= \frac{(1 + S)* \gamma_0}{k_{T} - S}, and, \tilde{p_i} <= \alpha_{max}\}$$
    - Returned linear regression model on the selected variables, model size, name of the selected variables and false selection rate.

```
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.regsubsets(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 at alpha[ia], S[1]=0
        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 quadritic 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

```
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
```

Unit Test

- Unit Tests on functions forward_selection and get_fsr

  - Unit test of forward_selection: The true linear relationship should be y = b + 10*x1+ 200*x2+ 0.5*x3 + 0.01*x4 + 0.001*x5. 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

```
def foward_selection(x,y):
    out_x = p1.regsubsets(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
```

```
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
```

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 verision improves twice as much as the pure python.

```
! pip install --pre line-profiler &> /dev/null
! pip install psutil &> /dev/null
! pip install memory_profiler &> /dev/null
```

```
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 = p1.regsubsets(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))])        # size of models at alpha[ia], S[1]=0
    ghat = [(m-S[i])*alpha[i]/(1+S[i]) for i in range(len(alpha))]              # gammahat_ER
    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,alphahat_fast


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]+9*x.ix[:,7]+4*x.ix[:,8]+1*x.ix[

        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]+45*x.ix[:,7]+36*x.ix[:,8]+25*x.ix

        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


%load_ext line_profiler


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]+36*x.ix[:,8]+25*x.ix
```

```
%lprun -f fsr_fast_vectorized fsr_fast_vectorized(x,y)
```

```
%lprun -f Naive_Fast_FSR Naive_Fast_FSR(x,y)
```

```
%lprun -f FSRR_vectorized FSRR_vectorized([0,1,2,3,4,5,6,7,8,9,10,11,12,13],fsr_fast_vectorized,4,100)
```

```
%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 comparsion for pure python and vectorized python

```
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
```

```
%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
```

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

```
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 he pure python.

```
%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
```

Application and comparison

- Comparsion 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.

```
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

```
target =[1]
n = int(100)

print "LASSO FSR is",pi_multiprocessing1(target,lasso_fit,1,n)
print "FAST FSR is",pi_multiprocessing1(target,fsr_fast_vectorized,1,n)
```

```
LASSO FSR is 0.443333333333
FAST FSR is 0.0833333333333
```

- Model 2

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

print "LASSO FSR is",pi_multiprocessing1(target,lasso_fit,2,n)
print "FAST FSR is",pi_multiprocessing1(target,fsr_fast_vectorized,2,n)
```

```
LASSO FSR is 0.540920745921
FAST FSR is 0.189285714286
```

- Model 3

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

print "LASSO FSR is",pi_multiprocessing1(target,lasso_fit,3,n)
print "FAST FSR is",pi_multiprocessing1(target,fsr_fast_vectorized,3,n)

LASSO FSR is 0.526709273183
FAST FSR is 0.138478188478
```

- Model 4

```
n = int(100)
target = [0,1,2,3,4,5,6,7,8,9,10,11,12,13]
print "LASSO FSR is",pi_multiprocessing1(target,lasso_fit,4,n)
print "FAST FSR is",pi_multiprocessing1(target,fsr_fast_vectorized,4,n)

LASSO FSR is 0.517102718482
FAST FSR is 0.112006535948
```

Extension for Fast_FSR: Bagging Fast FSR

- Bagging FSR: As you can see, if you use the normal Fast FSR, the rate will be very high. However, by implementing the Bagging Fast FSR,it will improve the False Selection rate to the target level

```
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,para_sd,pred,amean,sizem
```

```
x = pd.DataFrame(np.random.normal(1, 1, 21*1500).reshape(1500,21))
y = x.ix[:,0]+2*x.ix[:,1]
x.ix[:,3]= x.ix[:,0]
get_fsr ([0,1],fsr_fast_vectorized,x,y)
```

```
Reordering variables and trying again:
```

```
0.7500
```

```
B = 100
bag_fsr(x,y,B,0.05,fsr_fast_vectorized,4)
```

```
(0     0.746201
 1     0.535000
 2     2.003511
 3    -0.000070
 4     0.245000
 5     0.000919
 6    -0.003615
 7    -0.003380
 8     0.002279
 9    -0.001958
 10    0.001221
 11   -0.004208
 12   -0.001020
 13    0.000113
 14   -0.000528
 15    0.001096
 16    0.001677
 17    0.001543
 18    0.002455
 19   -0.000005
 20    0.003468
 21    0.002427
 dtype: float64, 0     0.346211
 1     0.431596
 2     0.012387
 3     0.005481
 4     0.349964
 5     0.006129
 6     0.013239
 7     0.011831
 8     0.008719
 9     0.010197
 10    0.010997
 11    0.015030
 12    0.006239
 13    0.007332
 14    0.007100
 15    0.009812
 16    0.008185
 17    0.008579
 18    0.010391
 19    0.006459
 20    0.012542
 21    0.015034
 dtype: float64, matrix([[ 7.7414],
         [ 2.0401],
         [ 7.529 ],
         ...,
         [ 6.5569],
         [ 3.021 ],
         [ 5.1156]]), 0.0515, 9.2900)
```

Reproducible analysis

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

```
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()
```

```
<table class="simpletable">
<caption>OLS Regression Results</caption>
<tr>
  <th>Dep. Variable:</th>            <td>y</td>        <th>  R-squared:         </th> <td>   0.811</td>
</tr>
<tr>
  <th>Model:</th>                   <td>OLS</td>       <th>  Adj. R-squared:    </th> <td>   0.800</td>
</tr>
<tr>
  <th>Method:</th>             <td>Least Squares</td>  <th>  F-statistic:       </th> <td>   75.50</td>
</tr>
<tr>
  <th>Date:</th>             <td>Thu, 30 Apr 2015</td> <th>  Prob (F-statistic):</th> <td>2.49e-30</td>
</tr>
<tr>
  <th>Time:</th>                 <td>18:09:42</td>     <th>  Log-Likelihood:    </th> <td> -315.88</td>
</tr>
<tr>
  <th>No. Observations:</th>      <td>    94</td>      <th>  AIC:               </th> <td>   643.8</td>
</tr>
<tr>
  <th>Df Residuals:</th>          <td>    88</td>      <th>  BIC:               </th> <td>   659.0</td>
</tr>
<tr>
  <th>Df Model:</th>              <td>     5</td>      <th>                     </th>    <td> </td>
</tr>
</table>
<table class="simpletable">
<tr>
    <td></td>       <th>coef</th>     <th>std err</th>      <th>t</th>      <th>P>|t|</th> <th>[95.0% Conf. Int.]
</th>
</tr>
<tr>
  <th>const</th> <td>  -42.1069</td> <td>    8.990</td> <td>   -4.684</td> <td> 0.000</td> <td>  -59.972
-24.242</td>
</tr>
<tr>
  <th>x2</th>    <td>    3.4714</td> <td>    0.467</td> <td>    7.428</td> <td> 0.000</td> <td>    2.543
4.400</td>
</tr>
<tr>
  <th>x3</th>    <td>    0.2391</td> <td>    0.076</td> <td>    3.163</td> <td> 0.002</td> <td>    0.089
0.389</td>
</tr>
<tr>
  <th>x5</th>    <td>    0.2787</td> <td>    0.078</td> <td>    3.582</td> <td> 0.001</td> <td>    0.124
0.433</td>
</tr>
<tr>
  <th>x4</th>    <td>    0.6770</td> <td>    0.195</td> <td>    3.475</td> <td> 0.001</td> <td>    0.290
1.064</td>
</tr>
<tr>
  <th>x7</th>    <td>   -2.5913</td> <td>    0.832</td> <td>   -3.115</td> <td> 0.002</td> <td>   -4.245
-0.938</td>
</tr>
</table>
<table class="simpletable">
<tr>
  <th>Omnibus:</th>       <td> 5.624</td> <th>  Durbin-Watson:     </th> <td>   1.718</td>
</tr>
<tr>
  <th>Prob(Omnibus):</th> <td> 0.060</td> <th>  Jarque-Bera (JB):  </th> <td>   3.905</td>
</tr>
<tr>
  <th>Skew:</th>          <td> 0.351</td> <th>  Prob(JB):          </th> <td>   0.142</td>
</tr>
<tr>
  <th>Kurtosis:</th>      <td> 2.290</td> <th>  Cond. No.          </th> <td>    620.</td>
</tr>
</table>
```