STA 663 Final Project

Nicole Solomon

April 30, 2015

1 Introduction

Model building and selection has applications across countless fields, including medicine, biology, and life sciences. The primary interest often is to identify those most strongly predictive of the outcome via model building. When datasets are abundant in potential covariates there may be concern of admitting or retaining possibly meaningless variables. To account for and control this "false" variable selection, Boos et al (2009) developed the fast false selection rate (FFSR) technique. This is an algorithm which performs variable selection, model selection, and model-sizing under the context of forward selection. These three aspects are determined by controlling for the 'false selection rate' (FSR) or the rate at which unimportant variables are added to the model.

1.1 Fast False Selection Rate Procedure

Typical forward selection starts with fitting all covariates to univariate models of the outcome and a single predictor: $Y \sim X_i$, $i = 1, \ldots, k$. A pre-specified α level is chosen as the cutoff p-value level for inclusion vs exclusion from the model. The covariate with the smallest p-value, "p-to-enter", less than α is kept and then the process is repeated, with the aforementioned covariate inclusively fixed in future models. The sequence of p-values for all k covariates is called the *forward addition sequence*.

In the context of forward selection, the FSR algorithm requires a monotonically increasing forward addition sequence. Hence, if a sequence is not monotone, the sequence it altered by carrying the largest p forward to give a monotonized sequence of p-values: $\tilde{p}$. These are then used to compute the FSR (γ) level for the model size corresponding to each $\tilde{p}$.

FSR is defined as

$$\gamma = \mathrm{E}\left\{\frac{U(Y,X)}{1 + I(Y,X) + U(Y,X)}\right\}$$

where $U(Y,X)$ and $I(Y,X)$ are the number of uninformative and informative variables in the given model respectively. Hence, $U + I = S$, the size of the current model. The expected value is with respect to repeated sampling of the true model, and a 1 is included in the denominator to avoid division by 0 and account for an intercept. The goal of the FSR procedure is to pre-specify an initial FSR, γ_0, and determine the appropriate α-to-enter level to meet this FSR; i.e. what must the cutoff (α) for any variable to be included in the model be, in order to restrict the rate at which unimportant variables enter the model.

Now for a given α the number of uninformative variables in the model is $U(\alpha) = U(Y,X)$. This quantity is estimated by $U(\alpha) \approx (k - S(\alpha))\hat{\theta}(\alpha)$, where $S(\alpha)$ is the model size at level α, k is the total number of possible predictors, and so $(k - S(\alpha))$ is an estimate of the number of uninformative variables in the dataset. $\hat{\theta}(\alpha)$ is the estimated rate at which uninformative variables enter the model. The original FSR method developed by the same authors estimates $\hat{\theta}$ by simulating 'phony' variables (unrelated to the outcome) and computing the rate at which these enter the model. The new 'fast' method found via simulations that $\hat{\theta}(\alpha) = \alpha$ is an acceptable substitute and considerably faster. It was found to be produce more accurate predictions than the former FSR when coupled with a bagging procedure (Boos et al, 2009). Hence, the fast FSR expression is:

$$\hat{\gamma}(\alpha) = \frac{(k - S(\alpha))\alpha}{1 + S(\alpha)}$$

In this manner one can build a table structured as follows:

Size	Variable	p-value	$\tilde{p}$-value	$\hat{\alpha}(\tilde{p})$	$\hat{\gamma}(\tilde{p})$
1	V1	1e-05	1e-05	0.002	0.0004
2	V2	0.005	0.005	0.005	0.0120
3	V3	0.021	0.021	0.017	0.2040
4	V4	0.009	0.021	0.028	0.2040
5	V5	0.053	0.053	0.033	0.4241

where the expression for $\hat{\alpha}$ is

$$\hat{\alpha}(\tilde{p}) = \frac{\gamma_0(1 + S(\tilde{p}))}{k - S(\tilde{p})}$$

In this manner it is easy to select a model size: the size where $\hat{\gamma} < \gamma_0$. Alternatively, if one utilizes a model of size $S(\tilde{p})$ with corresponding $\hat{\alpha}(\tilde{p})$ then one can look to the table to determine what the FSR of the chosen model is.

The goal of this project was to implement this algorithm under linear regression in Python. Specifically, functions were be written for the Fast FSR technique for three contexts: in its simplest form for pure variable selection, allowing for forced inclusion of a subset of variables, and with bagging. The code was tested on a dataset available from NCSU in order to demonstrate the technique and its efficiency. The algorithm was also compared to an equivalent R version of the algorithm (Boos, ncsu.edu) on the same dataset to demonstrate the correctness of the Python function.

2 Implementation & Pseudocode

Traditional forward selection can be slow in sizeable datasets. In order to produce an efficient algorithm, the `regsubsets` function from the `leaps` package in `R` was utilized to conduct the forward selection aspect of the FFSR algorithm. This function utilizes a branch-and-bound technique to quickly identify optimal subsets in the linear prediction of y by x, rather than fitting all possible models at each iteration (Lumley, 2015). The remainder of the algorithm was implemented in a modular programming fashion in Python. These modules perform the following tasks:

2.1 Pseudocode

```
1. Perform forward selection via the ‘regsubsets‘ function called in Python via the RPY2
   library
2. Store covariate order of entry into the model
3. Compute p-values for each sequential covariate entering the model
4. Monotonize p-values by taking sequential max
5. Compute gamma values for each step in model size
6. Compute alpha values for each monotonized p-value
7. Compute alpha for a pre-specified gamma (optional)
8. Estimate parameters for the final fitted model
```

Additional functions were written to neatly compile the results, as well as check for appropriate data input type. All of these functions are called within the primary ‘ffsr’ function. A streamlined version of this function was written without the data check or table compilation for the sake of bagging. An additional ‘bag_fsr’ function was written for implementation of FFSR with bagging; this function iteratively runs the ffsr algorithm on a duplicate of the original data built from randomly selecting the original rows with replacement. In this manner the resulting parameter estimates can be averaged, akin to model averaging. This produces predictions more accurate than those obtained with just one application of the FFSR algorithm (Boos, 2009).

3 Testing, Profiling, Optimization

Unit tests were drafted to assure that functions failed when appropriate, and raise specific warnings. The functions passed all unit tests as seens below. The code was also profiled to allow for optimization to improve

efficiency and speed. The primary bottleneck in the primary ffsr function is within the forward selection modular function. This procedure requires more than 75% of the time necessary to run the functions. Utilizing an R function rather than a Python or C function appears to be the reason for the slow speed. This issue is addressed in more detail in the next section.

```
In [1]: import os
        os.chdir('/home/bitnami/STA-663-Nicole-Solomon-Project/Tests')
        !py.test
        os.chdir('/home/bitnami/STA-663-Nicole-Solomon-Project/Report')
```

```
============================= test session starts ==============================
platform linux2 -- Python 2.7.9 -- py-1.4.25 -- pytest-2.6.3
collected 31 items

test_alpha.py ...
test_alphag.py ...
test_bagfsr.py ...
test_beta.py ...
test_covnames.py .
test_df_type.py ..
test_ffsr.py ..
test_gamma.py ...
test_pvals.py .

========================== 31 passed in 2.77 seconds ===========================
```

4 Application and Comparison

The Python algorithm was applied to a NCAA basketball dataset obtained from NCSU (Boos, ncsu.edu). The R FFSR algorithm was also applied to this data in order to compare the efficiency of the Python algorithm.

4.1 Standard FFSR function

```
In [2]: %run ffsr_r_run
        %run -t -m ffsr_p_run
```

```
NCSU R Results:

    S var   pval  pvmax  alpha      g
1   1   2 0.0000 0.0000 0.0056 0.0000
2   2   3 0.0001 0.0001 0.0088 0.0004
3   5   5 0.0116 0.0116 0.0214 0.0270
4   5   4 0.0053 0.0116 0.0214 0.0270
5   5   7 0.0025 0.0116 0.0214 0.0270
6   6  17 0.0433 0.0433 0.0269 0.0804
7   7  15 0.0527 0.0527 0.0333 0.0791
8  10   6 0.1056 0.1056 0.0611 0.0864
9  10   9 0.0826 0.1056 0.0611 0.0864
10 10   8 0.0536 0.1056 0.0611 0.0864
11 11  12 0.2350 0.2350 0.0750 0.1566
12 12  10 0.2864 0.2864 0.0929 0.1542
13 14  13 0.3163 0.3163 0.1500 0.1054
14 14  18 0.2697 0.3163 0.1500 0.1054
```

```
15 15  11 0.4953 0.4953 0.2000 0.1238
16 16   1 0.6326 0.6326 0.2833 0.1116
17 17  14 0.7056 0.7056 0.4500 0.0784
18 18  19 0.8605 0.8605 0.9500 0.0453
19 19  16 0.9032 0.9032    Inf 0.0000
   user  system elapsed
  0.007   0.000   0.007

Python Results:

     S  Var       p     p_m alpha_F gamma_F
0    1   x2  0.0000  0.0000  0.0056  0.0000
1    2   x3  0.0001  0.0001  0.0088  0.0004
2    5   x5  0.0116  0.0116  0.0214  0.0270
3    5   x4  0.0053  0.0116  0.0214  0.0270
4    5   x7  0.0025  0.0116  0.0214  0.0270
5    6  x17  0.0433  0.0433  0.0269  0.0804
6    7  x15  0.0527  0.0527  0.0333  0.0791
7   10   x6  0.1056  0.1056  0.0611  0.0864
8   10   x9  0.0826  0.1056  0.0611  0.0864
9   10   x8  0.0536  0.1056  0.0611  0.0864
10  11  x12  0.2350  0.2350  0.0750  0.1566
11  12  x10  0.2864  0.2864  0.0929  0.1542
12  14  x13  0.3163  0.3163  0.1500  0.1054
13  14  x18  0.2697  0.3163  0.1500  0.1054
14  15  x11  0.4953  0.4953  0.2000  0.1238
15  16   x1  0.6326  0.6326  0.2833  0.1116
16  17  x14  0.7056  0.7056  0.4500  0.0784
17  18  x19  0.8605  0.8605  0.9500  0.0453
18  19  x16  0.9032  0.9032  1.0000  0.9032

IPython CPU timings (estimated):
  User   :       0.03 s.
  System :       0.00 s.
Wall time:       0.03 s.
```

4.2 FFSR function with forced in variables

```
In [3]: %run ffsr_force_r_run
        %run -t -m ffsr_force_p_run

NCSU R Results:

    S var   pval  pvmax  alpha      g
1   3  12 0.0000 0.0000 0.0125 0.0000
2   3   3 0.0000 0.0000 0.0125 0.0000
3   3   5 0.0000 0.0000 0.0125 0.0000
4   4   2 0.0000 0.0000 0.0167 0.0000
5   6   4 0.0043 0.0043 0.0269 0.0079
6   6   7 0.0028 0.0043 0.0269 0.0079
7   8  17 0.0539 0.0539 0.0409 0.0659
8   8  15 0.0458 0.0539 0.0409 0.0659
9  11   6 0.0976 0.0976 0.0750 0.0651
10 11   9 0.0962 0.0976 0.0750 0.0651
11 11   8 0.0281 0.0976 0.0750 0.0651
```

```
12 12  10 0.2864 0.2864 0.0929 0.1542
13 14  13 0.3163 0.3163 0.1500 0.1054
14 14  18 0.2697 0.3163 0.1500 0.1054
15 15  11 0.4953 0.4953 0.2000 0.1238
16 16   1 0.6326 0.6326 0.2833 0.1116
17 17  14 0.7056 0.7056 0.4500 0.0784
18 18  19 0.8605 0.8605 0.9500 0.0453
19 19  16 0.9032 0.9032    Inf 0.0000
   user  system elapsed
  0.009   0.001   0.008

Python Results:

     S  Var       p     p_m alpha_F gamma_F
0    3  x12  0.0000  0.0000  0.0125  0.0000
1    3   x3  0.0000  0.0000  0.0125  0.0000
2    3   x5  0.0000  0.0000  0.0125  0.0000
3    4   x2  0.0000  0.0000  0.0167  0.0000
4    6   x4  0.0043  0.0043  0.0269  0.0079
5    6   x7  0.0028  0.0043  0.0269  0.0079
6    8  x17  0.0539  0.0539  0.0409  0.0659
7    8  x15  0.0458  0.0539  0.0409  0.0659
8   11   x6  0.0976  0.0976  0.0750  0.0651
9   11   x9  0.0962  0.0976  0.0750  0.0651
10  11   x8  0.0281  0.0976  0.0750  0.0651
11  12  x10  0.2864  0.2864  0.0929  0.1542
12  14  x13  0.3163  0.3163  0.1500  0.1054
13  14  x18  0.2697  0.3163  0.1500  0.1054
14  15  x11  0.4953  0.4953  0.2000  0.1238
15  16   x1  0.6326  0.6326  0.2833  0.1116
16  17  x14  0.7056  0.7056  0.4500  0.0784
17  18  x19  0.8605  0.8605  0.9500  0.0453
18  19  x16  0.9032  0.9032  1.0000  0.9032

IPython CPU timings (estimated):
  User   :       0.04 s.
  System :       0.00 s.
Wall time:       0.04 s.
```

4.3 FFSR with bagging

```
In [4]: %run ffsr_bag_r_run
        %run -t -m ffsr_bag_p_run

NCSU R Results:

   user  system elapsed
  0.733   0.010   0.749

[1] "Mean of estimated alpha-to-enter: 0.0454"

[1] "Mean size of selected model: 7.69"

Python Results:
```

```
Mean of estimated alpha-to-enter: 0.0503

Mean size of selected model: 7.095

IPython CPU timings (estimated):
  User   :       6.71 s.
  System :       0.27 s.
Wall time:       6.01 s.
```

As seen in the results above, the Python algorithm yields results identical to those returned by the R algorithm. The bagging results differ somewhat due to the random resampling in Python and R that cannot be matched via setting identical seeds. These results overall demonstrate the validity of the Python FFSR function. However, the Python algorithm is significantly slower than the R function. This is due to the overhead time spent in calling R, running the R function `regsubsets`, and then returning the results for processing.

This procedure would be significantly faster and competitive with the equivalent R algorithm if the Fortran code upon which `regsubsets` is based were to be wrapped in C and thereby made directly callable in Python. This is a daunting task however as the original Fortran code is of the type Fortran 77 and was written decades ago (Lumley, github.com). At the very least, an alternative, but performance-wise competitive, forward selection procedure is necessary to improve the Python algorithm beyond its current speed.

5 References

1. Boos D, Stefanski L, and Wu Y. "Fast FSR Variable Selection with Applications to Clinical Trials." *Biometrics*. 2009; 65(3): 692-700.
2. Boos D and Stefanski L. "Fast FSR: Controlling the False Selection Rate without Simulation." NCSU. http://www4.stat.ncsu.edu/~boos/var.select/fast.fsr.html.
3. Thomas Lumley. "Package 'leaps' ". R Foundation for Statistical Computing, 2015. Version 2.9.
4. Thomas Lumley and Alan Miller. "cran/leaps." https://github.com/cran/leaps/tree/master/src.