10.15 Variable Selection and the Bias-Variance Trade-Off

2020-10-02

1 Introduction

We consider the linear model with a univariate responce and p covariates, $Y = X\beta + \epsilon, \epsilon \sim N(0, \sigma^2)$. We denote a training dataset by $\mathcal{T} = (y_t, x_t)_{t=1:N}$ and the least squares estimate of β is the minimiser of the RSS (residual sum of squares) over the training set, $\hat{\beta}^{LS}(\mathcal{T}) = (\mathbf{x}^T\mathbf{x})^{-1}\mathbf{x}^T\mathbf{y}$, where x_t is a $1 \times p$ row vector representing and observation. Alternatively, we may employ matrix notation, letting $\mathbf{y} = (y_t)_{t=1:N}$ be a row vector and $\mathbf{x} = (x_{ti})$ an $N \times p$ matrix where each row represents an observation.

Under the assumption $N > p + 1$ we will refer to a subset of the covariates as a *model* $\mathcal{M} \subseteq \{1, ..., p\}$. The LS estimate for $\mathcal{M}$ is computed on a reduced dataset $\mathcal{T}^{\mathcal{M}} = (y_t, (x_{ti})_{i \in \mathcal{M}})_{t=1:N}$. For computational ease we will instead represent the LS estimate for $\mathcal{M}$ in the p-dimensional space of the original model. We denote this representation by $\hat{\beta}^{\mathcal{M}}$, where

$$\hat{\beta}^{\mathcal{M}}_j(\mathcal{T}) = \begin{cases} \hat{\beta}^{\mathcal{M}}_{\pi(j)}(\mathcal{T}^{\mathcal{M}}), & \text{if } j \in \mathcal{M} \\ 0, & \text{otherwise.} \end{cases}$$

Here $\pi : \mathcal{M} \to 1, ..., ||\mathcal{M}||$ maps indicies of covariates in the model to their respectve indicies in the reduced dataset $\mathcal{T}^{\mathcal{M}}$, so that $\mathbf{x}\hat{\beta}^{\mathcal{M}}(\mathcal{T})$ is a notation for $\mathbf{x}^{\mathcal{M}}\hat{\beta}^{LS}(\mathcal{T}^{\mathcal{M}})$.

2 The Bias-Variance tradeoff

2.1 Question 1

We start with model (1) and show the expected squared predictor error at a fixed, arbitrary location u can be deposed as follows (noting $y \perp \mathcal{T} \implies \epsilon \perp \hat{\beta}$):

$$\begin{aligned}
\mathbb{E}_{\mathcal{T},y|u}(y - u\hat{\beta})^2 &= \mathbb{E}_{\mathcal{T},y|u}(y^2 - 2yu\hat{\beta} + (u\hat{\beta})^2) \\
&= \mathbb{E}_{y|u}(y^2) - \mathbb{E}_{\mathcal{T}}(2u\beta\ u\hat{\beta}) - 2\mathbb{E}(\epsilon)\mathbb{E}_{\mathcal{T}}(u\hat{\beta}) + \mathbb{E}_{\mathcal{T}}((u\hat{\beta})^2) \\
&= \mathrm{Var}_{y|u}(y) + (u\beta)^2 - 2u\beta\mathbb{E}_{\mathcal{T}}(u\hat{\beta}) + (\mathbb{E}_{\mathcal{T}}(u\hat{\beta}))^2 + \mathrm{Var}_{\mathcal{T}}(u\hat{\beta}) \\
&= \sigma^2 + (u\beta - \mathbb{E}_{\mathcal{T}}(u\hat{\beta}))^2 + \mathrm{Var}_{\mathcal{T}}(u\hat{\beta}).
\end{aligned}$$

The summands on the right-hand side are reffered to as the *irreducible variance*, *squared estimation bias* (SEB) and *estimation variance*, respectively. Now consider the effect of removing the p^{th} covariate on each of these for $\hat{\beta} = \hat{\beta}^{LS}$. Under the assumption that $\mathbf{x}^T\mathbf{x} = I_p$ we see $\mathbf{x}_{\mathcal{M}}^T\mathbf{x}_{\mathcal{M}} = I_{p-1}$, so $u\hat{\beta} = u\hat{\beta}^{\mathcal{M}} + u_p\hat{\beta}_p$. Clearly, σ remains unchanged. Now for the SEB:

$$\begin{aligned}
SEB_{\mathcal{T}^{\mathcal{M}}} &= (u\beta - \mathbb{E}_{\mathcal{T}^{\mathcal{M}}}(u\hat{\beta}^{\mathcal{M}}))^2 \\
&= (u\beta - \mathbb{E}_{\mathcal{T}}(u\hat{\beta}) + \mathbb{E}_{\mathcal{T}}(u_p\hat{\beta}_p))^2 \\
&= (u_p\beta_p)^2 \\
&\geq 0 = SEB_{\mathcal{T}} \text{ (by the fact bias}(\hat{\beta}^{LS}) = 0),
\end{aligned}$$

with equality if and only if $u_p\beta_p = 0$. For the estimation variance:

$$\begin{aligned}
\mathrm{Var}_{\mathcal{T}^{\mathcal{M}}}(u\hat{\beta}^{\mathcal{M}}) &= \mathrm{Var}_{\mathcal{T}^{\mathcal{M}}}(u\hat{\beta} - u_p\hat{\beta}_p) \\
&= \mathbb{E}\left[\left(u\hat{\beta} - u_p\hat{\beta}_p - \mathbb{E}(u\hat{\beta} - u_p\hat{\beta}_p)\right)^2\right] \\
&= \mathrm{Var}_{\mathcal{T}}(u\hat{\beta}) - 2\mathbb{E}\left[(u\hat{\beta} - \mathbb{E}(u\hat{\beta}))(u_p\hat{\beta}_p - \mathbb{E}u_p\hat{\beta}_p)\right] + \mathrm{Var}_{\mathcal{T}}(u_p\hat{\beta}_p) \\
&= \mathrm{Var}_{\mathcal{T}}(u\hat{\beta}) - 2\mathbb{E}\left[\left(\sum_{i=1}^{p} u_i\hat{\beta}_i\right)\left(u_p\hat{\beta}_p - \mathbb{E}u_p\hat{\beta}_p\right)\right] + \mathrm{Var}_{\mathcal{T}}(u_p\hat{\beta}_p)
\end{aligned}$$

As we assume $\mathbf{x}^T\mathbf{x} = I_p$ then, $\hat{\beta}_i, \hat{\beta}_j$ indep for $i \neq j$.

$$\begin{aligned}
\mathrm{Var}_{\mathcal{T}^{\mathcal{M}}}(u\hat{\beta}^{\mathcal{M}}) &= \mathrm{Var}_{\mathcal{T}}(u\hat{\beta}) - 2\mathbb{E}\left[(u_p\hat{\beta}_p)(u_p\hat{\beta}_p - \mathbb{E}u_p\hat{\beta}_p)\right] + \mathrm{Var}_{\mathcal{T}}(u_p\hat{\beta}_p) \\
&= \mathrm{Var}_{\mathcal{T}}(u\hat{\beta}) - \mathrm{Var}_{\mathcal{T}}(u_p\hat{\beta}_p) \leq \mathrm{Var}_{\mathcal{T}}(u\hat{\beta})
\end{aligned}$$

So, $\mathrm{Var}_{\mathcal{T}^{\mathcal{M}}}(u\hat{\beta}^{\mathcal{M}}) \leq \mathrm{Var}_{\mathcal{T}}(u\hat{\beta})$, with equality if and only if

$$\mathrm{Var}_{\mathcal{T}}(u_p\hat{\beta}_p) = u_p^2\mathrm{Var}_{\mathcal{T}}(\mathbf{e}_p^T\mathbf{x}^T\mathbf{y}(\mathcal{T})) = u_p^2\mathbf{e}_p^T\mathbf{x}^T\mathrm{Cov}_{\mathcal{T}}(\mathbf{y}(\mathcal{T}))\mathbf{x}\mathbf{e} = \sigma^2u_p^2 = 0.$$

Thus, removing the p^{th} covariate will generally increase the bias but decrease the variance. Combining the results above we see $\mathbb{E}_{\mathcal{T}^{\mathcal{M}},y|u}(y - u\hat{\beta}^{\mathcal{M}})^2 = \mathbb{E}_{\mathcal{T},y|u}(y - u\hat{\beta})^2 + u_p^2(\beta_p^2 - \sigma^2)$, and so the expected square error falls if $\sigma^2 > \beta_p$ and rises if $\sigma^2 < \beta_p$, when removing the p^{th} co-variate (under our simplifying assumptions).

2.2 Question 2

Use the same model as above with $p = 10, \sigma^2 = 1$, and $X \sim N(0, I_p)$, and set $\beta = (-0.5, 0.45, -0.4, 0.35, -0.3, 0.25, -0.2, 0.15, -0.1, 0.05)^T$. We then consider $\mathcal{M}_1 = \{1\}, ..., \mathcal{M}_p = \{1, ..., p\}$, trained on a data set size N_{tr} and test data set of size $N_{te} = 1000$. Program 1 (page 13) completes this simulation and produces plots of the relative training and testing RSS averaged over 100 experiments graphed against the respective M_j. The following shows this run for $N_{tr} = 30$ and 200 with the respective tables below and graphs produced in Figures 1 and 2 (pages 11-12).

Listing 1: Completing statistical training 1

```
>>>true_beta = [-0.5,0.45,-0.4,0.35,-0.3,0.25,-0.2,0.15,-0.1,0.05]
>>>train_and_test(30, 1000, 10, 1, true_beta)
```

M_j	Average Testing RSS	Average Training RSS
1	1.7763	1.6314
2	1.6015	1.4006
3	1.4854	1.2210
4	1.4021	1.0673
5	1.3570	0.9589
6	1.3629	0.8529
7	1.3845	0.7851
8	1.4129	0.7283
9	1.4658	0.6819
10	1.5120	0.6482

Listing 2: Completing statistical training 2

```
>>>train_and_test(200, 1000, 10, 1, true_beta)
```

M_j	Average Testing RSS	Average Training RSS
1	1.7269	1.6973
2	1.5130	1.4946
3	1.3620	1.3325
4	1.2584	1.2071
5	1.1616	1.1021
6	1.1046	1.0356
7	1.0726	0.9910
8	1.0620	0.9667
9	1.0547	0.9531
10	1.0456	0.9474

It is first important to note that $RSS(\hat{\beta};\mathcal{T}) = \frac{1}{N}\sum_{t=1}^{N}(y_t - x_t\hat{\beta})^2$, so in the ideal scenario as $\hat{\beta} \to \beta$, $\mathbb{E}(RSS) \to \mathbb{E}(\epsilon^2) = \sigma^2 = 1$. The average RSS values approximate $\mathbb{E}(RSS)$, which in both cases is showing increased error to optimal, thus there are inaccuracies in the estimates of β. Furthermore, in both cases we are seeing the over training of data for high j as these give a training RSS less than 1 (though much less in the $N_{tr} = 200$ case). The negative affects of this on the accuracy of the model on test data can be seen in the respective tables and plots which show the optimal testing RSS values to occur for $j = 5$ and $j = 10$ for $N_{tr} = 30$ and 200 respectively. The first shows consistency with the results of Question 1 (this time for a less restricted model, $X^TX \neq I_p$ necessarily) where a balance of bias and variance must be met by managing the number of co-variate terms considered. It is also worth noting that in the $N_{tr} = 200$ case we see little benefit by considering fewer co-variates and that for $j > 6$ the models give very close approximations of the optimal expected RSS. This is because with this larger training data size our program is able to better estimate β, ignoring the noise produced by ϵ, thus, also avoiding over training. We then see as expected a larger training set increases the training RSS and reduces the testing RSS, with both approaching the optimal value: 1.

3 Variable selection methods

3.1 Subset Selection

3.1.1 Question 3: Best subsets selection

Program 2 shows a procedure which takes the input of a training dataset $\mathcal{T}$ and outputs a $p \times p$ matrix B, whose j^{th} column contains $\hat{\beta}^{\mathcal{M}_j}$ for the best (RSS) performing model of size j on the training data. The program assumes the same model as used within Question 2 but with some slight tweaks could be made to deal with any more general linear model. To run an example below the 30×10 training matrix was first formed as before then used as the input for Program 2 (page 14) to produce the following:

Listing 3: Best subsets selection on Question 2 model and training dataset

```
>>>training_dataset = form_list(30,10)
>>>bestsubset(training_dataset)
#Output matrix B transcribed into nicer format below
```

$$B = \begin{bmatrix} 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & -0.29 & -0.32 & -0.32 & -0.29 & -0.3 \\ 0.0 & 0.54 & 0.61 & 0.73 & 0.72 & 0.8 & 0.83 & 0.78 & 0.73 & 0.72 \\ -0.46 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & -0.17 & -0.23 & -0.24 \\ 0.0 & 0.42 & 0.47 & 0.6 & 0.53 & 0.49 & 0.52 & 0.52 & 0.56 & 0.56 \\ 0.0 & 0.0 & -0.4 & -0.59 & -0.61 & -0.55 & -0.5 & -0.47 & -0.51 & -0.49 \\ 0.0 & 0.0 & 0.0 & 0.53 & 0.49 & 0.46 & 0.43 & 0.41 & 0.46 & 0.45 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.24 & 0.26 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.3 & 0.31 & 0.33 & 0.3 & 0.22 & 0.23 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & -0.16 & -0.18 & -0.26 & -0.27 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.07 \end{bmatrix}$$

$$RSS_{training} = \begin{bmatrix} 1.43 & 1.29 & 1.13 & 0.94 & 0.88 & 0.81 & 0.80 & 0.78 & 0.76 & 0.76 \end{bmatrix}$$

$$RSS_{test} = \begin{bmatrix} 1.79 & 1.62 & 1.55 & 1.69 & 1.64 & 1.50 & 1.50 & 1.33 & 1.48 & 1.49 \end{bmatrix}$$

The RSS_{test} vector ($N_{te} = 1000$) was also produced to give insight into the relative accuracy at each value of j, and showed consistency with the theory discussed in Question 1 and results of Question 2. Interestingly, B is nearly diagonal which supports what we would expect seeing as the terms of β decrease in magnitude for increasing indexes (discrepancies arise from the stochastic nature of the program). It is important to note that this method is very computationally expensive, with a model space of size $|\{\mathcal{M} : \mathcal{M} \subseteq \{1, ..., p\}| = 2^p - 1$ (formed on the logic that creating a subset has two options for each element of the set, to include or not include in the subset, then ignore the empty set) individual cases to find the optimal $\mathcal{M}_j$ for all j. Set $K_j = |\{\mathcal{M} : \mathcal{M} \subseteq \{1, ..., p\}, ||\mathcal{M}|| = j\}|$ for $j \in \{1, ..., p\}$, we know the program runs under acceptable time frames for $\max_j(K_j) < 10^5$. Then note by the combinatoric definition of the binomial coefficient we see $K_j = \binom{p}{j}$. It is clear that this is at a maximum for $j = \lfloor \frac{p}{2} \rfloor$ by considering the equality $\binom{n}{r+1} = \frac{n-r}{r+1}\binom{n}{r}$, and the respective values of r for which $\binom{n}{r}$ increases/decreases. Then note, $\binom{19}{9} = 92378 < 10^5 < 184756 = \binom{20}{10}$, thus the smallest such p is 20.

3.1.2 Question 4: Greedy subset selection

We now look at a procedure *greedysubset*, using the same input-output format as before, that incrementally builds the model sequence $\mathcal{M}_j$ at each iteration as follows:

$$\mathcal{M}_0 = \varnothing, \mathcal{M}_{d+1}(\mathcal{T}) = \mathcal{M}_d(\mathcal{T}) \cup \left\{ l : l = \arg\min_j RSS\left(\hat{\beta}^{\mathcal{M}_d(\mathcal{T}) \cup \{j\}}(\mathcal{T}); \mathcal{T}\right) \right\}.$$

This procedure is implemented in Program 3 (page 16), producing the following:

Listing 4: Greedy subsets selection on Question 2 model and training dataset

```
>>>greedysubset(training_dataset)
#using the same training dataset as before
#Output matrix B transcribed into nicer format below
```

$$B = \begin{bmatrix} 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & -0.28 & -0.29 & -0.32 & -0.29 & -0.3 \\ 0.0 & 0.0 & 0.42 & 0.51 & 0.65 & 0.72 & 0.75 & 0.78 & 0.73 & 0.72 \\ -0.46 & -0.5 & -0.37 & -0.31 & -0.22 & -0.22 & -0.15 & -0.17 & -0.23 & -0.24 \\ 0.0 & 0.32 & 0.42 & 0.46 & 0.59 & 0.55 & 0.49 & 0.52 & 0.56 & 0.56 \\ 0.0 & 0.0 & 0.0 & -0.37 & -0.55 & -0.49 & -0.52 & -0.47 & -0.51 & -0.49 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.5 & 0.47 & 0.44 & 0.41 & 0.46 & 0.45 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.24 & 0.26 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.28 & 0.3 & 0.22 & 0.23 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & -0.18 & -0.26 & -0.27 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.07 \end{bmatrix}$$

$$RSS_{training} = \begin{bmatrix} 1.43 & 1.31 & 1.22 & 1.09 & 0.92 & 0.86 & 0.80 & 0.78 & 0.76 & 0.76 \end{bmatrix}$$

$$RSS_{test} = \begin{bmatrix} 1.79 & 1.69 & 1.47 & 1.37 & 1.49 & 1.33 & 1.34 & 1.33 & 1.48 & 1.49 \end{bmatrix}$$

Again the RSS_{test} vector ($N_{te} = 1000$, note the use of $np.random.seed(10)$ to match results with Question 3) is also provided to give an insight into the accuracy of this models. The first thing to note is the algorithm is much faster in producing the matrix B. Though the algorithm does not always choose the minimal training RSS at each stage, this appears to have little effect on the realisation of testing RSS. In fact greedy subset appears to outperform best subset for M_j with $3 \leq j \leq 7$ for test RSS in this seed. This may be by random chance or that by choosing and sticking with parameters earlier *greedysubset* may priorities true patterns in the training data over 'false patterns' as a result of randomness.

We note that the computational complexity could be even further improved upon by noting the the family of models is nested allowing us to more efficiently calculate $(X^T_{\mathcal{M}_{j+1}} X_{\mathcal{M}_{j+1}})^{-1}$, normally being $\mathcal{O}(N^2_{tr}(j+1)^2)$ for the multiplication and then $\mathcal{O}((j+1)^3)$ for the inverse. Then work from the assumption $M_j = \{1, ..., j\}$ (if not, the following still applies simply under a change of basis):

$$X = X^T_{\mathcal{M}_{j+1}} X_{\mathcal{M}_{j+1}} = \begin{bmatrix} X^T_{\mathcal{M}_j} X_{\mathcal{M}_j} & X^T_{\mathcal{M}_j} x_{j+1} \\ x^T_{j+1} X_{\mathcal{M}_j} & |x_{j+1}|^2 \end{bmatrix},$$

$$X' = \begin{bmatrix} A(X^T_{\mathcal{M}_j} X_{\mathcal{M}_j})^{-1} & \mathbf{b} \\ \mathbf{a}^T & c \end{bmatrix},$$

with A some $j \times j$ matrix, $\mathbf{a}, \mathbf{b} \in \mathbb{R}^j$ and $c \in \mathbb{R}$. Set $X'X = I_{j+1}$, giving:

$$A + \mathbf{b} x^T_{j+1} X_{\mathcal{M}_j} = I_j \tag{1}$$

$$A(X^T_{\mathcal{M}_j} X_{\mathcal{M}_j})^{-1} X^T_{\mathcal{M}_j} x_{j+1} + \mathbf{b}|x_{j+1}|^2 = \mathbf{0} \tag{2}$$

$$\mathbf{a}^T X^T_{\mathcal{M}_j} X_{\mathcal{M}_j} + c x^T_{j+1} X_{\mathcal{M}_j} = \mathbf{0}^T \tag{3}$$

$$\mathbf{a}^T X^T_{\mathcal{M}_j} x_{j+1} + c|x_{j+1}|^2 = 1 \tag{4}$$

Now this can be solved for $A(X_{\mathcal{M}_j}^T X_{\mathcal{M}_j})^{-1}, \mathbf{a}, \mathbf{b}$ and c, assuming the existence of a solution (which occurs almost surely), starting with (1) and (3) for A and $\mathbf{a}$ respectively, then using (2) and (4) to give $\mathbf{b}$ and c in terms of $X_{\mathcal{M}_j}, x_{j+1}$ and $(X_{\mathcal{M}_j}^T X_{\mathcal{M}_j})^{-1}$.

$$c = \frac{1}{|x_{j+1}|^2 - x_{j+1}^T X_{\mathcal{M}_j}(X_{\mathcal{M}_j}^T X_{\mathcal{M}_j})^{-1} X_{\mathcal{M}_j}^T x_{j+1}}$$

$$\mathbf{b} = -c(X_{\mathcal{M}_j}^T X_{\mathcal{M}_j})^{-1} X_{\mathcal{M}_j}^T x_{j+1}$$

$$A(X_{\mathcal{M}_j}^T X_{\mathcal{M}_j})^{-1} = (X_{\mathcal{M}_j}^T X_{\mathcal{M}_j})^{-1} - \mathbf{b} x_{j+1}^T X_{\mathcal{M}_j}(X_{\mathcal{M}_j}^T X_{\mathcal{M}_j})^{-1}.$$

$$\mathbf{a} = -c x_{j+1}^T X_{\mathcal{M}_j}(X_{\mathcal{M}_j}^T X_{\mathcal{M}_j})^{-1}$$

This reduces the complexity to $\mathcal{O}(N_{tr} j + j^3)$, which is particularly significant for large N_{tr}.

3.1.3 Question 5: Forward F-test

We now want to consider the use of an F-statistic ($F \sim F_{1,N_{tr}-d-1}$)

$$F = \frac{RSS(\hat{\beta}^{\mathcal{M}_d}) - RSS(\hat{\beta}^{\mathcal{M}_{d+1}})}{RSS(\hat{\beta}^{\mathcal{M}_{d+1}})/(N_{tr} - d - 1)}$$

to force our program to stop when increasing d does not significantly improve the fit at the $p = 0.05$ level. This is implemented in *greedysubset2* (Program 4, page 18), with use of the *scipy.stats.f.cdf* function in python. Program 4 is set up return matrix B however, this is the same as that for Question 4. So for conciseness below, only the final vector of $\hat{\beta}^{\mathcal{M}_j}$ and its RSS will be displayed. It is also important to note, in the case that $RSS(\hat{\beta}^{\mathcal{M}_j}) < RSS(\hat{\beta}^{\mathcal{M}_{j+1}})$ the algorithm will terminate and return B.

Listing 5: Greedy subsets 2 selection on Question 2 model and training dataset

```
>>>greedysubset2(training_dataset)
RSS_{test} = 1.49
RSS_{training} = 0.92
[0, 0.65, -0.22, 0.58, -0.55, 0.50, 0, 0, 0, 0]
```

This same method could not be used in for the best subset selection method, this is because in the general case F no longer follows an $F \sim F_{1,pN_{tr}-d-1}$ distribution. We can see this by considering in the *greedysubset* method

$$RSS(\hat{\beta}^{\mathcal{M}_d}) - RSS(\hat{\beta}^{\mathcal{M}_{d+1}}) = ||(P_{d+1} - P_d)Y||^2 \overset{*}{\sim} \chi_1^2, \tag{5}$$

$$RSS(\hat{\beta}^{\mathcal{M}_{d+1}}) = ||(I - P_{d+1})Y||^2 \overset{**}{\sim} \chi_{N_{tr}-d-1}^2, \tag{6}$$

as $\sigma^2 = 1$, for the respective projection matrices P_d and P_{d+1}. In general, ie. *bestsubset*, (**) holds, however, (*) holds only if $(P_{d+1} - P_d)$ is a projection

matrix. Note $(P_{d+1} - P_d)^2 = P_{d+1}^2 - P_{d+1}P_d - P_dP_{d+1} + P_d^2$, but $\mathcal{M}_d$ need not be a subset of $\mathcal{M}_{d+1}$, in fact they could be disjoint, which would imply $(P_{d+1} - P_d)^2 = P_{d+1} + P_d \neq P_{d+1} - P_d$. Thus, it is not a projection (as it isn't idempotent), so (5) or in particular (*) does not hold in general for *bestsubset*, and so the F test is invalid here.

3.1.4 Question 6

We consider now a new estimate of $\beta \approx \hat{\beta}^{CV}$ which selects one of the solutions of a given sparse estimator ($\hat{\beta}^{(j)}$ is the j^{th} candidate with precisely $p-j$ zeros), chosen on the basis of estimated predicted error $\hat{PE}$:

$$\hat{\beta}^{CV}(\mathcal{T}) = \hat{\beta}^{j^*}(\mathcal{T}), \text{ where } j^* = \arg\min_j \left\{ \hat{PE}(j, \mathcal{T}) \right\}.$$

The prediction error can be estimated using 10-fold cross-validation as

$$\hat{PE}(j, \mathcal{T}) = \frac{1}{10} \sum_{k=1}^{10} RSS\left(\hat{\beta}^{(j)}(\mathcal{T}^{-k}); \mathcal{T}^k \right),$$

where $\mathcal{T}^k$ is the k^{th} fold of the training set and $\mathcal{T}^{-k}$ its complement:

$$\mathcal{T}^k = \left\{ (y_{\pi(n)}, x_{\pi(n)}) | k - 1 < \frac{10n}{N_{tr}} \leq k \right\}$$
$$\mathcal{T}^{-k} = \left\{ (y_{\pi(n)}, x_{\pi(n)}) | \frac{10n}{N_{tr}} \leq k - 1 \text{ or } \frac{10n}{N_{tr}} > k \right\},$$

where $\pi(n)$ is a random permutation of $\{1, ..., N_{tr}\}$, calculated uniquely for each k, each run through the algorithm (using *np.random.permutation*). The running of this algorithm requires this procedure to take a function as an input argument, in python this is no different to using a variable as it treats both objects very similarly. This is put together in Program 5 (page 20), taking an input $\mathcal{T}$ and a sparse estimator, and giving the output $\hat{\beta}^{CV}(\mathcal{T})$. Some examples read below:

Listing 6: Finding the cross-validation estimator for β

```
>>>crossval(training_dataset, greedysubset)
array([ 0,  0.67, -0.15,  0.53, -0.59, 0.47,  0,  0.27,  0,  0])
>>>crossval(training_dataset, bestsubset) #long run time
array([0, 0.73, 0, 0.60, -0.59, 0.53, 0, 0, 0, 0])
```

3.2 The Lasso estimator

3.2.1 Question 7

The lasso estimator penalises the RSS as follows:

$$\hat{\beta}^{(L,\lambda)}(\mathcal{T}) = \arg\min_{\hat{\beta}} \left\{ RSS\left(\hat{\beta}; \mathcal{T} \right) + \lambda \sum_{j=1}^{p} \left| \hat{\beta}_j \right| \right\}. \tag{7}$$

This can be transformed into a quadratic program with linear constraints as follows (using the fact that λ is fixed),

$$\hat{\beta}^{(L,\lambda)}(\mathcal{T}) = \arg\min_{\hat{\beta}} \left\{ RSS\left(\hat{\beta}; \mathcal{T}\right) + \lambda \left(\sum_{j=1}^{p} \left|\hat{\beta}_j\right| + z(\lambda) - k(\lambda) \right) : z, k \geq 0 \right\}$$

We recognise this as the Lagrangian for the problem ($\mathcal{T} = (\mathbf{y}, \mathbf{x})$):

$$\begin{aligned}\hat{\beta}^{(L,\lambda)}(\mathcal{T}) &= \arg\min_{\hat{\beta}} \left\{ (\mathbf{y} - \mathbf{x}\hat{\beta})^T(\mathbf{y} - \mathbf{x}\hat{\beta}) : \sum_{j=1}^{p} \left|\hat{\beta}_j\right| \leq k(\lambda) \right\} \\ &= \arg\min_{\hat{\beta}} \left\{ \hat{\beta}^T(\mathbf{x}^T\mathbf{x})\hat{\beta} - (2\mathbf{y}^T\mathbf{x})\hat{\beta} : \sum_{j=1}^{p} \left|\hat{\beta}_j\right| \leq k(\lambda) \right\}\end{aligned}$$

Now to linearise the constraints we must remove the absolute values, this can be done using the $2^p \times p$ matrix A with rows consisting of all permutations of ± 1's:

$$A = \begin{bmatrix} 1 & \dots & 1 & -1 \\ 1 & \dots & -1 & 1 \\ 1 & \dots & -1 & -1 \\ \vdots & & \vdots & \vdots \\ -1 & \dots & -1 & -1 \end{bmatrix}$$

Noting $|a| = \max\{a, -a\}$, we arrive at the required quadratic program with linear constraints as follows:

$$\hat{\beta}^{(L,\lambda)}(\mathcal{T}) = \arg\min_{\hat{\beta}} \left\{ \hat{\beta}^T(\mathbf{x}^T\mathbf{x})\hat{\beta} - (2\mathbf{y}^T\mathbf{x})\hat{\beta} : A\hat{\beta} \leq k(\lambda) \right\}.$$

3.3 Question 8

The program *linear_model.lars_path* from the (Python) machine learning package *sklearn* can be used to produce a function equivalent in practice to that of the provided *monotonic_lars.m*. This can then be used as an input for *crossval* via Program (REFERENCE). We note here that the use of *np.rand.seed*() is important for reproducibility of the crossval function with input *monotonic_lars*, as different permutations $\pi()$ will lead to differing outputs of *crossval*. The following displays some examples:

Listing 7: Using *monotonic_lars*

```
>>>monotonic_lars(training_data)
#transcribed below
>>>crossval(training_data, monotonic_lars)
array([-0.17,  0.46, -0.11,  0.31, -0.33, 0.22,  0,  0.20,  0,  0])
```

$$B = \begin{bmatrix} 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & -0.06 & -0.17 & -0.23 & -0.23 & -0.3 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.08 & 0.17 & 0.46 & 0.57 & 0.59 & 0.72 \\ 0.0 & 0.0 & 0.0 & -0.03 & -0.06 & -0.08 & -0.11 & -0.13 & -0.14 & -0.24 \\ 0.02 & 0.03 & 0.03 & 0.05 & 0.1 & 0.14 & 0.31 & 0.39 & 0.41 & 0.56 \\ 0.0 & -0.01 & -0.02 & -0.04 & -0.1 & -0.14 & -0.33 & -0.38 & -0.4 & -0.49 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.22 & 0.28 & 0.31 & 0.45 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.03 & 0.26 \\ 0.0 & 0.0 & 0.01 & 0.03 & 0.07 & 0.11 & 0.2 & 0.23 & 0.23 & 0.23 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & -0.06 & -0.08 & -0.27 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.07 \end{bmatrix}$$

Then we split up the *prostate.dat*, with an additional 4 rows of random values ($\sim N(0,1)$), into a training data set of size 70 and testing set of size 27 as follows.

Listing 8: Data set up

```
>>>training_set, testing set = setup_data()
```

4 Appendices

4.1 Figures

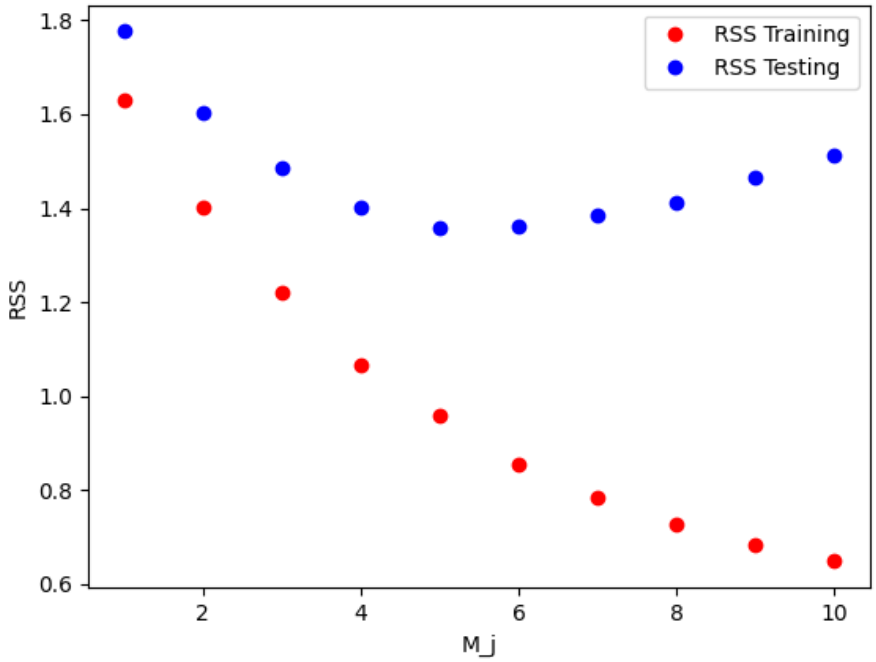

Figure 1: Average RSS in training and test data sets for LS estimator with $N_{tr} = 30$ and $N_{te} = 1000$.

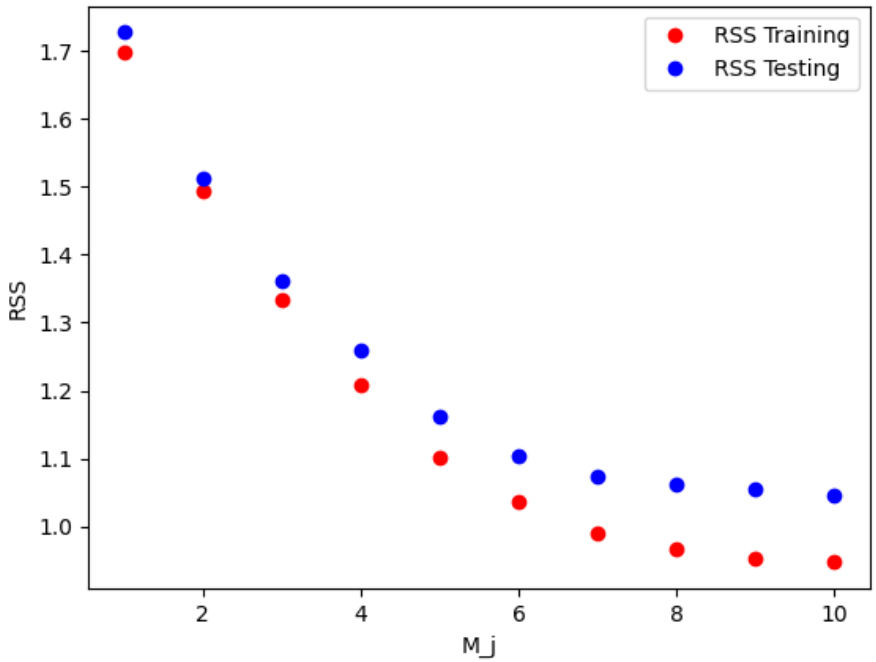

Figure 2: Average RSS in training and test data sets for LS estimator with $N_{tr} = 200$ and $N_{te} = 1000$.

4.2 Code

4.2.1 Program 1

Listing 9: Program 1

```
def train_and_test(N_tr, N_te, p, sigma, true_beta):
    #Repeat 100 times
    RSS_test_total = np.array([0] * p)
    RSS_training_total = np.array([0] * p)
    repeat = 1
    while repeat <= 100:
        #training
        vector_x = np.random.normal(0, 1, p)
        scalar_y = np.dot(vector_x, true_beta) + np.random.normal(0,sigma,1)
        matrix_x = np.array([vector_x])
        vector_y = np.array([scalar_y])
        i=2
        while i <= N_tr:
            vector_x = np.random.normal(0,1,p)
            scalar_y = np.dot(vector_x, true_beta) + np.random.normal(0,sigma,1)
            matrix_x = np.append(matrix_x, [vector_x],axis=0)
            vector_y = np.append(vector_y, scalar_y)
            i=i+1

        #now complete testing for each of M_j
        RSS_test = [0] * p
        RSS_training = [0] * p
        j=1
        while j <= p:
            reduced_matrix_x = np.zeros([N_tr,j])
            for repeat1 in range(N_tr):
                for repeat2 in range(j):
                    reduced_matrix_x[repeat1][repeat2] = matrix_x[repeat1][repeat2]

            #Calculate reduced_beta by LS
            product1 = np.linalg.inv(np.dot(reduced_matrix_x.T, reduced_matrix_x))
            product2 = np.dot(product1, reduced_matrix_x.T)
            reduced_beta = np.dot(product2, vector_y)
            #fix size
            while len(reduced_beta) < p:
                reduced_beta = np.append(reduced_beta, 0)

            #now do N_te tests
            vector_x_test = np.random.normal(0, sigma, p)
            rand = np.random.normal(0, sigma, 1)
            scalar_y_true = np.dot(vector_x_test, true_beta) + rand
            scalar_x_beta_test = np.dot(vector_x_test, reduced_beta)
            vector_y_true = np.array([scalar_y_true])
            vector_x_beta_test = np.array([scalar_x_beta_test])
            k = 2
            while k <= N_te:
                vector_x_test = np.random.normal(0, 1, p)
                rand = np.random.normal(0, sigma, 1)
                scalar_y_true = np.dot(vector_x_test, true_beta) + rand
                scalar_x_beta_test = np.dot(vector_x_test, reduced_beta)
                vector_y_true = np.append(vector_y_true, scalar_y_true)
                vector_x_beta_test = np.append(vector_x_beta_test, scalar_x_beta_test)
                k = k + 1
```

```
            #Calc RSS
            RSS_vector = np.subtract(vector_y_true, vector_x_beta_test)
            RSS_test[j-1] = (np.dot(RSS_vector, RSS_vector))/ N_te

            #Now run the N_tr as tests
            vector_x_training = matrix_x[0]
            scalar_y_true_training = vector_y[0]
            scalar_x_beta_training = np.dot(vector_x_training, reduced_beta)
            scalar_y_training = scalar_x_beta_training + (scalar_y_true_training - np.dot(vect
            vector_y_true_training = np.array([scalar_y_true_training])
            vector_x_beta_training = np.array([scalar_x_beta_training])
            k = 2
            while k <= N_tr:
                vector_x_training = matrix_x[k-1]
                scalar_y_true_training = vector_y[k-1]
                scalar_x_beta_training = np.dot(vector_x_training, reduced_beta)
                vector_y_true_training = np.append(vector_y_true_training, scalar_y_true_traini
                vector_x_beta_training = np.append(vector_x_beta_training, scalar_x_beta_traini
                k=k+1
            RSS_vector = np.subtract(vector_y_true_training, vector_x_beta_training)
            RSS_training[j - 1] = (np.dot(RSS_vector, RSS_vector)) / N_tr

            j=j+1
        RSS_test_total = np.add(RSS_test_total, RSS_test)
        RSS_training_total = np.add(RSS_training_total, RSS_training)
        repeat = repeat +1

    #Produce RSS vectors
    RSS_test_total = np.divide(RSS_test_total,100)
    RSS_training_total = np.divide(RSS_training_total,100)
    jay = [1,2,3,4,5,6,7,8,9,10]

    #Create Plots
    plt.plot(jay, RSS_training_total, 'ro', label='RSS Training')
    plt.plot(jay, RSS_test_total, 'bo', label='RSS Testing')
    plt.xlabel('M_j')
    plt.ylabel('RSS')
    plt.legend()
    #plt.title('Average RSS in training and test data sets for LS estimator')
    plt.show

    return
```

Listing 10: Program 2

4.2.2 Program 2

```
from itertools import chain, combinations

def powerset(iterable):
    "powerset([1,2,3]) --> () (1,) (2,) (3,) (1,2) (1,3) (2,3) (1,2,3)"
    s = list(iterable)
    return chain.from_iterable(combinations(s, r) for r in range(1, len(s)+1))

def form_list(N_tr, p):
    #forms (N_tr by p) matrix training set like Q2
    vector_x = np.random.normal(0,1,p)
```

```
    matrix_x = np.array([vector_x])
    true_beta = np.array([-0.5,0.45,-0.4,0.35,-0.3,0.25,-0.2,0.15,-0.1,0.05])
    i=2
    while i<=N_tr:
        vector_x = np.random.normal(0, 1, p)
        matrix_x = np.append(matrix_x, [vector_x], axis=0)
        i=i+1

    # Complete training
    vector_y = np.array([])
    for i in range(N_tr):
        vector_y = np.append(vector_y, np.dot(matrix_x[i], true_beta) + np.random.normal(0, 1,

    training_dataset = [vector_y, matrix_x]
    return training_dataset

def bestsubset(training_dataset):
    matrix = training_dataset[1]
    vector_y = training_dataset[0]

    p = len(matrix[0])
    true_beta = np.array([-0.5,0.45,-0.4,0.35,-0.3,0.25,-0.2,0.15,-0.1,0.05])
    N_tr = len(matrix)
    N_te = 1000
    M_full = []
    for i in range(1,p+1):
        M_full = M_full + [i]
    M_powerset = list(powerset(M_full))
    #print(M_full)
    #print(M_powerset)

    #set up RSS arrays
    RSS_test = []
    RSS_training = []
    for i in range(p):
        RSS_test = RSS_test + [[100,'', [0]*10]]
        RSS_training = RSS_training + [[100,'', [0]*10]]

    for M_j in M_powerset:
        jay = len(M_j)
        M_j_array = []
        reduced_beta = [0] * p
        for a in M_j:
            M_j_array = M_j_array + [a]

        #Calculate reduced_beta LS
        left_out = [0] * p
        reduced_matrix_x = np.zeros([N_tr, jay])
        for repeat1 in range(N_tr):
            for repeat2 in range(jay):
                index = M_j_array[repeat2]
                reduced_matrix_x[repeat1][repeat2] = matrix[repeat1][index-1]
                left_out[index-1] = 1
        product1 = np.linalg.inv(np.dot(reduced_matrix_x.T, reduced_matrix_x))
        product2 = np.dot(product1, reduced_matrix_x.T)
        reduced_beta_initial = np.dot(product2, vector_y)
        #fix size
```

```
        count=0
        for i in range(p):
            if left_out[i]==0:
                reduced_beta[i]=0
            else:
                reduced_beta[i]=reduced_beta_initial[count]
                count=count+1

        #Complete testing
        np.random.seed(10)
        vector_x_test = np.random.normal(0, 1, p)
        rand = np.random.normal(0, 1, 1)
        scalar_y_true = np.dot(vector_x_test, true_beta) + rand
        scalar_x_beta_test = np.dot(vector_x_test, reduced_beta)
        vector_y_true = np.array([scalar_y_true])
        vector_x_beta_test = np.array([scalar_x_beta_test])
        k = 2
        while k <= N_te:
            vector_x_test = np.random.normal(0, 1, p)
            rand = np.random.normal(0, 1, 1)
            scalar_y_true = np.dot(vector_x_test, true_beta) + rand
            scalar_x_beta_test = np.dot(vector_x_test, reduced_beta)
            vector_y_true = np.append(vector_y_true, scalar_y_true)
            vector_x_beta_test = np.append(vector_x_beta_test, scalar_x_beta_test)
            k = k + 1

        #RSS values
        vector_x_beta_tr = np.dot(matrix, reduced_beta)
        RSS_vector = np.subtract(vector_y, vector_x_beta_tr)
        RSS_scalar = (np.dot(RSS_vector, RSS_vector)) / N_tr
        RSS_vector2 = np.subtract(vector_y_true, vector_x_beta_test)
        RSS_scalar2 = (np.dot(RSS_vector2, RSS_vector2)) / N_te
        if RSS_scalar < RSS_training[jay-1][0]:
            RSS_training[jay-1][0] = RSS_scalar
            RSS_training[jay-1][1] = M_j_array
            RSS_training[jay-1][2] = reduced_beta
            RSS_test[jay - 1][0] = RSS_scalar2
            RSS_test[jay - 1][1] = M_j_array
            RSS_test[jay - 1][2] = reduced_beta

    print(RSS_test)
    print(RSS_training)

    #Create matrix B
    B=np.zeros([p,p])
    for i in range(p):
        B[i] = RSS_training[i][2]

    #print(B)
    return(B.T)
```

4.2.3 Program 3

Listing 11: Program 3

```
def greedysubset(training_dataset):
    matrix = training_dataset[1]
    vector_y = training_dataset[0]
```

```
p = len(matrix[0])
true_beta = np.array([-0.5,0.45,-0.4,0.35,-0.3,0.25,-0.2,0.15,-0.1,0.05])
N_tr = len(matrix)
N_te = 1000
M_full = []
M_not = []
for i in range(p):
    M_not = M_not + [i+1]

# set up RSS arrays
RSS_test = []
RSS_training = []
for i in range(p):
    RSS_test = RSS_test + [[100, '', [0] * 10]]
    RSS_training = RSS_training + [[100, '', [0] * 10]]

#start subset testing
for i in range(p):
    for j in M_not:
        M_try = M_full + [j]
        M_try.sort()

        jay = len(M_try)
        reduced_beta = [0] * p

        # Calculate reduced_beta LS
        left_out = [0] * p
        reduced_matrix_x = np.zeros([N_tr, jay])
        for repeat1 in range(N_tr):
            for repeat2 in range(jay):
                index = M_try[repeat2]
                reduced_matrix_x[repeat1][repeat2] = matrix[repeat1][index - 1]
                left_out[index - 1] = 1
        product1 = np.linalg.inv(np.dot(reduced_matrix_x.T, reduced_matrix_x))
        product2 = np.dot(product1, reduced_matrix_x.T)
        reduced_beta_initial = np.dot(product2, vector_y)

        # fix size
        count = 0
        for i in range(p):
            if left_out[i] == 0:
                reduced_beta[i] = 0
            else:
                reduced_beta[i] = reduced_beta_initial[count]
                count = count + 1

        # Complete testing
        np.random.seed(10)
        vector_x_test = np.random.normal(0, 1, p)
        rand = np.random.normal(0, 1, 1)
        scalar_y_true = np.dot(vector_x_test, true_beta) + rand
        scalar_x_beta_test = np.dot(vector_x_test, reduced_beta)
        vector_y_true = np.array([scalar_y_true])
        vector_x_beta_test = np.array([scalar_x_beta_test])
        k = 2
        while k <= N_te:
```

```
                vector_x_test = np.random.normal(0, 1, p)
                rand = np.random.normal(0, 1, 1)
                scalar_y_true = np.dot(vector_x_test, true_beta) + rand
                scalar_x_beta_test = np.dot(vector_x_test, reduced_beta)
                vector_y_true = np.append(vector_y_true, scalar_y_true)
                vector_x_beta_test = np.append(vector_x_beta_test, scalar_x_beta_test)
                k = k + 1

            # Calculate and update RSS_test vectors
            # RSS values
            vector_x_beta_tr = np.dot(matrix, reduced_beta)
            RSS_vector = np.subtract(vector_y, vector_x_beta_tr)
            RSS_scalar = (np.dot(RSS_vector, RSS_vector)) / N_tr
            RSS_vector2 = np.subtract(vector_y_true, vector_x_beta_test)
            RSS_scalar2 = (np.dot(RSS_vector2, RSS_vector2)) / N_te
            if RSS_scalar < RSS_training[jay - 1][0]:
                RSS_training[jay - 1][0] = RSS_scalar
                RSS_training[jay - 1][1] = j
                RSS_training[jay - 1][2] = reduced_beta
                RSS_test[jay - 1][0] = RSS_scalar2
                RSS_test[jay - 1][1] = M_try
                RSS_test[jay - 1][2] = reduced_beta

        length = len(M_not)
        k=0
        while k < length:
            if M_not[k] == RSS_training[jay-1][1]:
                M_full.append(M_not[k])
                M_not.remove(M_not[k])
                k=10
            k=k+1

    print(RSS_test)
    print(RSS_training)

    # Create matrix B
    B = np.zeros([p, p], dtype=float)
    for i in range(p):
        B[i] = RSS_training[i][2]

    return(B.T)
```

4.2.4 Program 4

Listing 12: Program 4

```
def greedysubset2(training_dataset):
    matrix = training_dataset[1]
    vector_y = training_dataset[0]

    p = len(matrix[0])
    true_beta = np.array([-0.5,0.45,-0.4,0.35,-0.3,0.25,-0.2,0.15,-0.1,0.05])
    N_tr = len(matrix)
    N_te = 1000
    M_full = []
    M_not = []
    for i in range(p):
        M_not = M_not + [i+1]
```

```
# set up RSS arrays
RSS_test = []
RSS_training = []
for i in range(p):
    RSS_test = RSS_test + [[100, '', [0] * 10]]
    RSS_training = RSS_training + [[100, '', [0] * 10]]

#start subset testing
m=0
tf = True
while m < p and tf:
    for j in M_not:
        M_try = M_full + [j]
        M_try.sort()

        jay = len(M_try)
        M_j_array = []
        reduced_beta = [0] * p
        for a in M_try:
            M_j_array = M_j_array + [a]

        # Calculate reduced_beta LS
        left_out = [0] * p
        reduced_matrix_x = np.zeros([N_tr, jay])
        for repeat1 in range(N_tr):
            for repeat2 in range(jay):
                index = M_j_array[repeat2]
                reduced_matrix_x[repeat1][repeat2] = matrix[repeat1][index-1]
                left_out[index - 1] = 1
        product1 = np.linalg.inv(np.dot(reduced_matrix_x.T, reduced_matrix_x))
        product2 = np.dot(product1, reduced_matrix_x.T)
        reduced_beta_initial = np.dot(product2, vector_y)
        # fix size
        count = 0
        for i in range(p):
            if left_out[i] == 0:
                reduced_beta[i] = 0
            else:
                reduced_beta[i] = reduced_beta_initial[count]
                count = count + 1

        # Complete testing
        vector_x_test = np.random.normal(0, 1, p)
        rand = np.random.normal(0, 1, 1)
        scalar_y_true = np.dot(vector_x_test, true_beta) + rand
        scalar_x_beta_test = np.dot(vector_x_test, reduced_beta)
        vector_y_true = np.array([scalar_y_true])
        vector_x_beta_test = np.array([scalar_x_beta_test])
        k = 2
        while k <= N_te:
            vector_x_test = np.random.normal(0, 1, p)
            rand = np.random.normal(0, 1, 1)
            scalar_y_true = np.dot(vector_x_test, true_beta) + rand
            scalar_x_beta_test = np.dot(vector_x_test, reduced_beta)
            vector_y_true = np.append(vector_y_true, scalar_y_true)
            vector_x_beta_test = np.append(vector_x_beta_test, scalar_x_beta_test)
```

```
            k = k + 1

        # Calculate and update RSS_test vectors
        np.random.seed(10)
        vector_x_beta_tr = np.dot(matrix, reduced_beta)
        RSS_vector = np.subtract(vector_y, vector_x_beta_tr)
        RSS_scalar = (np.dot(RSS_vector, RSS_vector)) / N_tr
        RSS_vector2 = np.subtract(vector_y_true, vector_x_beta_test)
        RSS_scalar2 = (np.dot(RSS_vector2, RSS_vector2)) / N_te
        if RSS_scalar < RSS_training[jay - 1][0]:
            RSS_training[jay - 1][0] = RSS_scalar
            RSS_training[jay - 1][1] = j
            RSS_training[jay - 1][2] = reduced_beta
            RSS_test[jay - 1][0] = RSS_scalar2
            RSS_test[jay - 1][1] = M_try
            RSS_test[jay - 1][2] = reduced_beta
    length = len(M_not)
    k = 0
    while k < length:
        if M_not[k] == RSS_training[jay - 1][1]:
            M_full.append(M_not[k])
            M_not.remove(M_not[k])
            k = 10
        k = k + 1

    #F-testing and whether to repeat again
    if jay > 1:
        F = (RSS_training[jay-2][0] - RSS_training[jay-1][0]) / (RSS_training[jay-1][0] /
        p_value = 1 - scipy.stats.f.cdf(F, 1, N_tr - jay)
        if p_value < 0.05 or F < 0:
            tf = False
    m=m+1

print(RSS_test)
print(RSS_training)

# Create matrix B
B = np.zeros([p, p], dtype=float)
for i in range(p):
    B[i] = RSS_training[i][2]

return(B.T)
```

4.2.5 Program 5

Listing 13: Program 5

```
def crossval(training_dataset, function):
    N_tr = len(training_dataset[1])
    p = len(training_dataset[1][0])
    matrix = training_dataset[1]
    vector_y = training_dataset[0]

    # form the estimator matrices
    B_k = [0]*p
    test_sets = [0]*p
    np.random.seed(100)
    for k in range(1,p+1):
```

```
    #calculate random permuation for each k
    pi = np.random.permutation(N_tr)

    lower_bound = N_tr * (k-1) / p
    upper_bound = N_tr * k / p
    i=1
    n=[]
    not_n=[]
    while i <= N_tr:
        if lower_bound < i <= upper_bound:
            n = n + [i-1]
        else:
            not_n = not_n +[i-1]
        i = i+1
    #print(n)
    #print(not_n)
    pi_n = pi[n]
    not_pi_n = pi[not_n]
    #print(pi_n)
    #print(not_pi_n)

    vector_y_n = vector_y[pi_n]
    vector_y_not_n = vector_y[not_pi_n]
    matrix_x_n = matrix[pi_n]
    matrix_x_not_n = matrix[not_pi_n]


    training_k = [vector_y_n, matrix_x_n]
    training_not_k = [vector_y_not_n, matrix_x_not_n]

    #function must take training dataset as the input
    single_matrix = function(training_not_k)
    B_k[k-1] = single_matrix
    test_sets[k-1] = training_k

RSS_min = 0
for j in range(p):
    RSS_total = 0
    for k in range (1,p+1):
        N_te = len(test_sets[k-1][0])
        vector_y_true = test_sets[k-1][0]
        matrix_x_test = test_sets[k-1][1]
        B = B_k[k-1]
        B = B.T
        vector_xbeta_test = []
        for i in range(N_te):
            scalar_xbeta_test = np.dot(B[j], matrix_x_test[i])
            vector_xbeta_test = np.append(vector_xbeta_test, scalar_xbeta_test)

        RSS_vector = vector_y_true - vector_xbeta_test
        RSS = np.dot(RSS_vector, RSS_vector) / N_te
        RSS_total = RSS_total + RSS

    RSS_total = RSS_total / p
    #print('j=' + str(j))
    #print(RSS_total)
```

```
        if RSS_min == 0:
            RSS_min = RSS_total
            count = 0
        elif RSS_total < RSS_min:
            RSS_min = RSS_total
            count = j

    B_needed = function(training_dataset)
    B_needed = B_needed.T
    return(B_needed[count])
```

4.2.6 Program 6

Listing 14: Program 6

```
def monotonic_lars(training_data):
    matrix_x = training_data[1]
    vector_y = training_data[0]
    N_tr = len(matrix_x)
    p = len(matrix_x[0])
    alphas, active, coef_path =
sklearn.linear_model.lars_path(matrix_x,vector_y)
    coef_path = coef_path.T
    output = np.zeros([p,p])
    for i in range(p):
        output[i] = coef_path[i+1]
    output = output.T
    return output

def setup_data():
    #READ data
    dat_file = r"C:\Austin\II_Maths\CATAM\
10.15_Variable_Selection_and_the_Bias-Variance_Tradeoff\test\
prostatedata.dat"
    data = pandas.read_csv(dat_file, sep="\t", header=None)

    #Add 4 extra covariate columns
    np.random.seed(1)
    col1 = np.random.normal(0,1,97)
    col2 = np.random.normal(0,1,97)
    col3 = np.random.normal(0,1,97)
    col4 = np.random.normal(0,1,97)
    data[9],data[10],data[11],data[12] = col1, col2, col3, col4

    #Now get in training_data format
    y_vector = np.array(data[0])
    x_matrix = data.T[1:]
    x_matrix = np.array(x_matrix.T)
    full_data = [y_vector, x_matrix]

    #Subsection into training and testing data
    np.random.seed(11)
    y_vector_training = [0]*70
    y_vector_testing = [0]*27
    x_matrix_training = [0]*70
    x_matrix_testing = [0]*27
    rows = [0]*97
    for i in range(97):
```

```
    rows[i] = i
rand_rows = np.random.permutation(rows)
for i in range(70):
    index = rand_rows[i]
    y_vector_training[i] = y_vector[index]
    x_matrix_training[i] = x_matrix[index]
for i in range(27):
    index = rand_rows[70+i]
    y_vector_testing[i] = y_vector[index]
    x_matrix_testing[i] = x_matrix[index]

training_data = [np.array(y_vector_training), np.array(x_matrix_training)]
testing_data = [np.array(y_vector_testing), np.array(x_matrix_testing)]
return training_data, testing_data
```