STAT 6100 - Advanced Regression

Spring 2020

FINAL PROJECT

From-Scratch Implementation of the Least Angle Regression Algorithm for LASSO Variable Selection

Jared Hansen

April 26, 2020

1 Introduction

The chief goal of this project was to gain a deeper understanding and appreciation of the Least Angle Regression (LAR hereafter) algorithm as applied to the Lasso, detailed by Efron, Hastie, Johnstone, and Tibshirani in their 2004 paper [1]. Having consulted with you, Dr. Cutler, I figured that there was no better way to understand this topic than coding it up from scratch. Doing so required a sufficient understanding of every step in the algorithm, as well as being able to translate mathematics into code. It also deepended my knowledge of the algorithm itself, making me aware of things which I didn't fully understand prior to undertaking this project (see Section 3 in particular).

My original intent was to code this up from scratch in both Python and C++. Due to time constraints, as well as much more difficulty than I'd anticipated, I have only done this in Python (with "done" being a somewhat generous characterization of my work). For comparison, I also used pre-built implementations in SAS, R, and Python. This also gave me a nice measuring stick to verify I was proceeding in the right direction.

2 Data

After some initial attempts at replicating the LAR-selected Lasso with a large data set, I ended up using the very small (and thus manageable) Hald cement data set to develop my code. This allowed for much easier prototyping, prevented computational load from being an issue, and made for easy comparison of my implementation to existing ones to ensure I was proceeding correctly. Interestingly, even with pre-built packages in the most common statistical programming languages, there was still surprising variability in the final results (see Section 4).

It's worth giving a short background of this dataset to highlight characteristics relevant when applying Lasso regression to it. First, the data is comprised of four predictors: aluminate, dicalcium, ferrite, and silicate. All four of these are chemical compounds used in cement. The response variable is heat, meaning heat given off in calories per gram when the concrete is hardening. The dataset contains only 13 observations, making it very manageable computationally.

Variable selection (via methods such as the Lasso) is usually of more importance when we have a very large number of predictors and are interested in performing selection to create a parsimonious model. While a model with just four variables is quite parsimonious already, our predictors exhibit high collinearity, potentially suggesting we might remove one or two of them and have nearly as good a model (where "good" is quantified by a metric like PRESS or MSE). Figure 1 shows high collinearity between a couple pairs of variables. Aluminate and ferrite have a large-magnitude correlation at -0.824. Dicalcium and silicate have an even larger-magnitude correlation at -0.973, indicating near-perfect collinearity. From these diagnostics we might expect that a variable selection method would remove either dicalcium or silicate from the model. As we'll see in Section 4, this does not end up being the case!

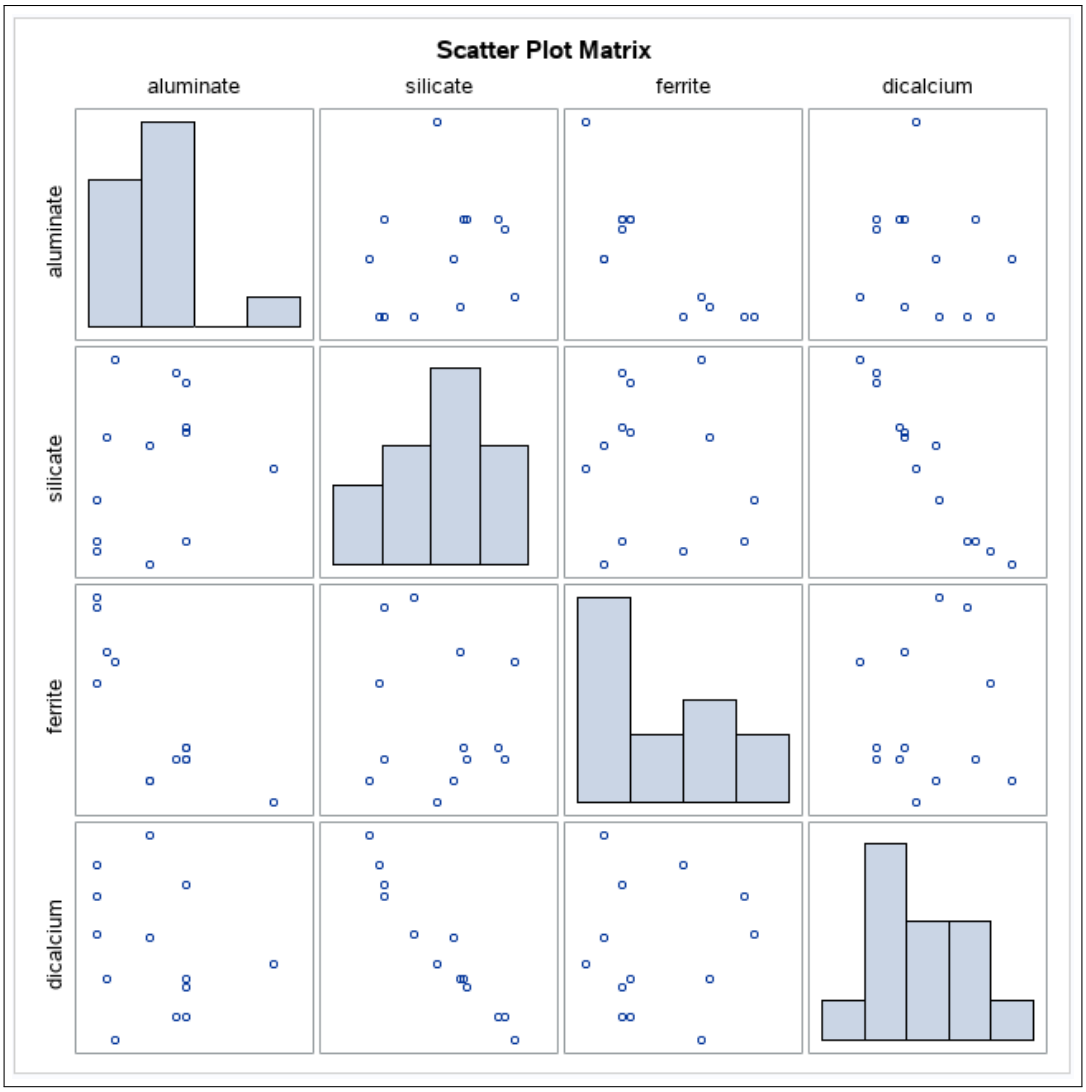


Figure 1: Pairwise scatterplots of the Hald cement data predictors.

3 LAR and Lasso Background

3.1 Pertinent Mathematical Details of LAR and Lasso

While ridge regression has a nice analytic solution thanks to the differentiability of the penalty term in the objective function (and resulting differentiability of the entire objective function), Lasso regression does not possess this nice property. To clarify, let's examine the objective function being optimized for Lasso [2, p. 68]:

$$\hat{\beta}^{lasso} = \underset{\beta}{\operatorname{argmin}} \left\{ \frac{1}{2} \sum_{i=1}^{N} \left(y_i - \beta_0 - \sum_{j=1}^{p} x_{ij}\beta_j \right)^2 + \lambda \sum_{j=1}^{p} |\beta_j| \right\}, \lambda \geq 0 \tag{1}$$

Notice the term at the end of the expression in the braces with the lambda in front of it. This quantity is the penalty term, determining how much (or how little) we choose to restrict large coefficients in our model. A very large lambda value essentially makes all β coefficients zero, while a lambda value of zero eliminates the penalty term entirely and results in the OLS $\hat{\beta}$ coefficient estimates.

As alluded to above, since the absolute value term in the penalty is non-differentiable, we cannot use calculus to carry out a clean, closed-form mathematical optimization of the Lasso objective function. Intuitively, we can still see that our Equation 1 function is the sum of two convex functions. The first portion of the equation is a quadratic, which is certainly convex (we could use the second derivative test to verify this if we wanted to). Although the absolute value function is not differentiable, it is also obviously convex. It is a well-characterized property that the sum of two convex functions is convex, therefore Equation 1 is still convex. However, since it is non-differentiable we have to rely on numerical optimization techniques, such as coordinate descent, to compute the β coefficients which minimize this function.

As good as some convex optimizers may be, Efron et al. showed that the LAR algorithm is a fantastic optimizer of the Lasso objective in their LAR paper [1]. Let's build a bit of intuition about this optimization problem so we can understand why the LAR approach is so good. A naive approach to Lasso coefficient estimation (read "objective function minimization") would be to systematically vary the regression coefficients and the lambda parameter and record the resulting objective function value. After performing this search we would simply select the coefficients and lambda values which result in the minimum objective function value. This brute force and combinatorics-esque approach quickly becomes intractable as the number of predictors in the dataset grows. (It might actually be a computationally feasible approach for a dataset as small as the Hald cement data though.)

A more intelligent approach is to use some kind of descent approach, such as coordinate descent as mentioned above (this is a close cousin of other approaches like gradient descent). This approach is much faster than a brute force global search, and takes advantage of past iterations to update subsequent iterations intelligently.

Better than either the brute force or the descent approach is the LAR method of optimization. Two methods of convex optimization often taught in machine learning and optimization courses are gradient descent and Newton's method. In my experience, as the initial guess of the decision variables (the β vector in regression) grows farther and farther from the optimum, gradient descent takes far more iterations to converge than does Newton's method. However, the quick convergence of Newton's method comes at a cost. Equation 2 shows the update step of Newton's method [2, p. 120].

$$\beta_{k+1} = \beta_k - \left(\left[\nabla^2 \beta_k \right]^{-1} \left[\nabla \beta_k \right] \right) \tag{2}$$

Notice the inverse Hessian in this formula. Computing the Hessian, and especially inverting it, is quite computationally expensive. However, this expense is compensated by arriving at the optimum in much fewer iterations. Similarly, the LAR algorithm relies on more precise computations (primarily vector and matrix multiplication) which result in much fewer iterations. Like Newton's Method, LAR also involves taking the inverse of a matrix (the inverse of the $\mathcal{G}_\mathcal{A} = \left[X'_\mathcal{A} \right] \left[X_\mathcal{A} \right]$ matrix in the paper). In fact, if performing stagewise variable selection, the LAR algorithm only takes m iterations to complete (where m is the number of predictor variables). For certain datasets, this also holds when using LAR for the Lasso, although in some cases it may take a few steps more if variables' coefficients come into the model and then are removed after reaching 0.

Rather than reiterating the entire LAR algorithm as found in the paper, I will highlight an aspect of the algorithm I wasn't familiar with prior to undertaking this project. In class we'd talked about finding the variable most highly correlated with the response (which are also the residual values when all β values begin at 0). After finding this variable, we increase its coefficient estimate in the direction of its correlation (either positive or negative) until another variable has the same correlation with the residuals as the first variable in. Then we adjust both of their coefficients in the respective directions of their correlations until a third variable has the same correlation with the residuals as the first two variables in. Prior to reading the paper, I didn't know that these adjustments to the coefficients aren't uniform. In other words, we aren't adjusting the betas equally according to some formula like $\beta_{i,new} = \beta_{i,old} + (sign(correlation_i))(0.05)$.

The reason that we can't adjust coefficients uniformly is because it doesn't result in variables maintaining the same correlation with the residuals, which is what allows us to find the next variable to bring into the model. The uniform adjustment approach is actually the method I naively tried at the start. After realizing that it wasn't working I had to go back to the paper and dig into the specific mechanics of how we adjust the coefficients to maintain the correlations of the in-model variables with the residuals at each step. In short, the way this is achieved is by use of the *equiangular vector*, denoted as $\mathbf{u}_\mathcal{A}$ in the paper [1, p. 7]. This vector projects the "active" (in-the-model) variables in the correct directions (according to sign of the correlation, the $X_\mathcal{A}$ matrix) and along the correct angles (the "least angle" using the $w_\mathcal{A}$ vector) to ensure we maintain equal correlations with the residuals of the active variables while allowing the next-in variable to have this same correlation in the update step. This precise geometric calculation is why the LAR algorithm only needs m iterations to complete and also why it doesn't require searching in the same manner that a descent algorithm does. In short, The *equiangular vector* is the key to the LAR algorithm!

3.2 Ambiguities in the LAR Paper and Difficulties in Implementation

In reading through the paper and then implementing the mathematics from scratch in Python I came across a few issues. It is easiest to simply list them and discuss briefly.

- Gamma for the final update step: for each update step we must compute a γ value. This value is defined as $\gamma = \min_{j \in \mathcal{A}^C}{}^{+} \left\{ \frac{\hat{C} - \hat{c_j}}{A_\mathcal{A} - a_j}, \frac{\hat{C} + \hat{c_j}}{A_\mathcal{A} + a_j} \right\}$. The key thing to notice here is the $j \in \mathcal{A}^C$ qualifier on the minimum condition. This means that we are only considering the positive minimum values for indices (variables) which are in the complement of the active set (variables not in the active set, in other words). Because of this, when we get to the last step, there

is still another update to do, but there is ambiguity on the part of how γ is calculated since the complement of the active set is now the empty set (since all variable are active at this point). The best answer I could find for how to handle this came from someone who'd done a similar project [3]. Their approach was to simply calculate γ without accounting for the j indexing, meaning that we're simply left with $\gamma = \left\{ \frac{\hat{C}}{A_{\mathcal{A}}} \right\}$. This seemed to work in practice, but I was surprised that this was left underspecified in the original paper. It may be that my calculation of gamma was done too early in each iteration, meaning that I was doing this step "off by one." However, I looked through things several times (both the paper and my code) and am fairly convinced that this was a small oversight which wasn't precisely cleared up in the paper, requiring the assumption made by Perkins in [3].

- Issues of $\mathcal{G}_{\mathcal{A}}$ matrix invertibility: One of the required calculations made in the algorithm is $\left[\mathcal{G}_{\mathcal{A}}\right]^{-1}$. Some issues occur when this matrix is singular. This was easily circumvented in Python's `NumPy` library (sub-library `linalg`), which has a psuedo-inverse (`pinv`) function. However, I'm not sure if this results in small numerical inaccuracies, and I'm also not sure exactly how existing packages handle this issue.

- Numerical Precision: When calculating new correlations to determine the next variable to enter the model, the first couple of iterations worked splendidly. The correlation with the residuals of the already-in and next-in variable(s) were almost exactly the same. However, as we get to the fourth and final variable in the Hald cement data (ferrite), these numbers are just off enough that it prevents the intuitive for loop (or recursive) automation of the model. We can't presume a number of decimal places to which we should round correlations, and even if we do, sometimes the correlations are far enough off that the correct variable doesn't enter the model. This is actually the reason I didn't put the iterations of the LAR in a for loop in my "from-scratch code" in the appendix. There was too much additional logic to write to fully - or even partially - account for precision issues. This is one of the issues that I would've liked to get around had I had more time and patience. I believe that this becomes more of an issue the more variables we have in the model; since correlations with the residuals begin to drop, it requires a very high degree of numerical precision to get correct results using strictly the LAR algorithm with no accompanying 'safety net' logic.

- Obtaining actual coefficient values at each step: The LAR algorithm actually only provides updated predictions at each step ($\hat{\mu}_i$ in the paper) for the standardized response vector. It doesn't ever describe how to get actual β values at each step. Since we need these values to create the coefficient progression plots in Section 4 below, I once again had to use a little bit of linear algebra to do this. It wasn't very hard, as we know that $\hat{y} = X\beta \longrightarrow \beta \approx X^{-1}\hat{y}$. Again, we run into the issue of taking an inverse, and here we certainly have to take a pseudo-inverse since X is not a square matrix. While I did use this approach to get β values that created a reasonable from-scratch coefficient progression plot, I'm not sure that it was the right way to do this, and don't know how pre-built packages compute the coefficients.

- Extracting the intercept term of the model(s): One thing that confused me a bit was that the pre-built Lasso-using-LAR implementations in R, SAS, and Python all had an intercept term in their respective models. I figured that since we standardize all predictors and the response that there would be no need for an intercept term. In reading the LAR paper I was never able to figure out how to get the intercept, or where SAS, R, and Python are getting their

intercept terms. It was interesting to me that despite not having an intercept term, my model coefficients were still similar to those of the pre-built packages (see Table 1).

- Obtaining MSE values for each model: Another thing which proved difficult was obtaining MSE values for each model in the LAR steps. The main reason I wanted to do this was to have something to compare my 3-variable best model to the pre-built models seen in Table 1. However, I simply could not get calculations to work out such that I was getting reasonably comparable MSE values. This was mainly because I was trying to back-transform from the standardized, unit-length values of the predicted response to the original scale of the response values. I think this is another issue which simply would've been solved with more time on the project. Since it wasn't a critical issue (the coefficient progression plot indicated to me that I'd essentially implemented the algorithm right), I simply had to accept defeat here.

4 Results and Comparison

In order to compare my from-scratch implementation to others, I carried out LAR-Lasso in R using `glmnet` [4], in SAS using `proc glmselect` [5], and in Python using the `LarsLassoCV` function from the `linear_model` module of the `scikit-learn` machine learning library [6]. To facilitate valid comparisons, I used k-fold, cross-validated PRESS to select the best model for each package.

As an admitted Python devotee, I was quite let down with the Python offerings for LASSO. There exist a large number of different Python packages and functions relating to LASSO regression, but none of them encapsulated all of the characteristics which I was looking for, and which are contained in R's `glmnet` and SAS's `glmselect`. The best Python package I could find for comparison was the `LarsLassoCV` function. It did end up giving a fairly similar model to what R and SAS returned (see Table 1), but the similarities did not hold for the coefficient progression plot. Although the coefficients reported in the final model are for standardized data, I was entirely unable to have the progression plot reflect standardized values. Also, I had to create a legend on my own, which was annoying relative to how easily legends were made in R and SAS.

4.1 Notes of Interest for Coefficient Progression Plots

I largely determined how close I was to replicating the LAR-Lasso was by plotting coefficient progression and comparing my plot to the output of R, Python, and SAS packages. In retrospect, I should have saved one or two of the plainly wrong plots that I created when using my "uniform adjustment approach" for coefficients. To anyone familiar with the Lasso, these plots would provide some quality comedic relief, with coefficients bouncing all over the place in odd geometric patterns.

- Plot structure inconsistency: There didn't seem to be a uniform or accepted way of structuring coefficient plots in each language. For example, in R (see Figure 2) the x-axis is the log of the lambda penalty parameter in Equation 1 while in Python (see Figure 3) the x-axis is the negative log of the penalty parameter (alpha in Python, lambda in R). Different still, SAS (see Figure 4) appears to simply uniformly space the horizontal distance between each step in the Lasso. For my implementation (see Figure 5), I used an x-axis measured by the sum of the absolute value of the coefficients, $\sum_{j=1}^{m} \left|\hat{\beta}_j\right|$. Although the x-axis measure could be adjusted, there was a heavily-preferred measure in each language.
  The scale of the y-axes in the plots also differed. The R, SAS, and from-scratch Python plots all had relatively small (and standardized) coefficients, while the pre-built Python package used much larger value (potentially in un-standardized units).

- Final variable in the model: The R and pre-built Python plots never include ferrite (the final variable) at any point, while this variable just squeaks in at the last step in the SAS and from-scratch Python plots. I made sure to plot things fully (meaning I didn't truncate the R and Python plots at the best model which only had three variables), so I was surprised that two of the three plots broke the well-known Lasso rule that all variables have a chance to enter the model. I'm sure that the ferrite variable was included numerically and in computation for R and Python, but somehow this didn't make it into plots.

- Differently-shaped paths: Another thing which is very striking when comparing all of these plots: the variables paths of the plots bear surprisingly little resemblance to each other. In the R plot (figure 2), all paths are very smooth and seem to level off quickly. In the pre-built Python plot (figure 3) we can see that coefficients only increase in magnitude in a very linear fashion (with the exception of dicalcium at the third step). The SAS plot (Figure 4) shows a much more nuanced behavior, with coefficient values moving both up and down depending on the step. The from-scratch Python plot (Figure 5) shows largely linear paths similar to the pre-built Python plot, while also showing some similarities to the SAS plot. Overall, I'd say that the SAS plot and the from-scratch Python plot bear the most resemblance of the four plots. For such a simple dataset and such a well-studied algorithm and regression method I was very surprised to see so many differences between the final coefficient progression plots.

- Order of variable entry: One point of consistency among all four plots was the order in which variables entered into the model. All approaches included dicalcium first, then aluminate, silicate, and ferrite. This makes sense, as the underlying LAR algorithm unifies each of these implementations.

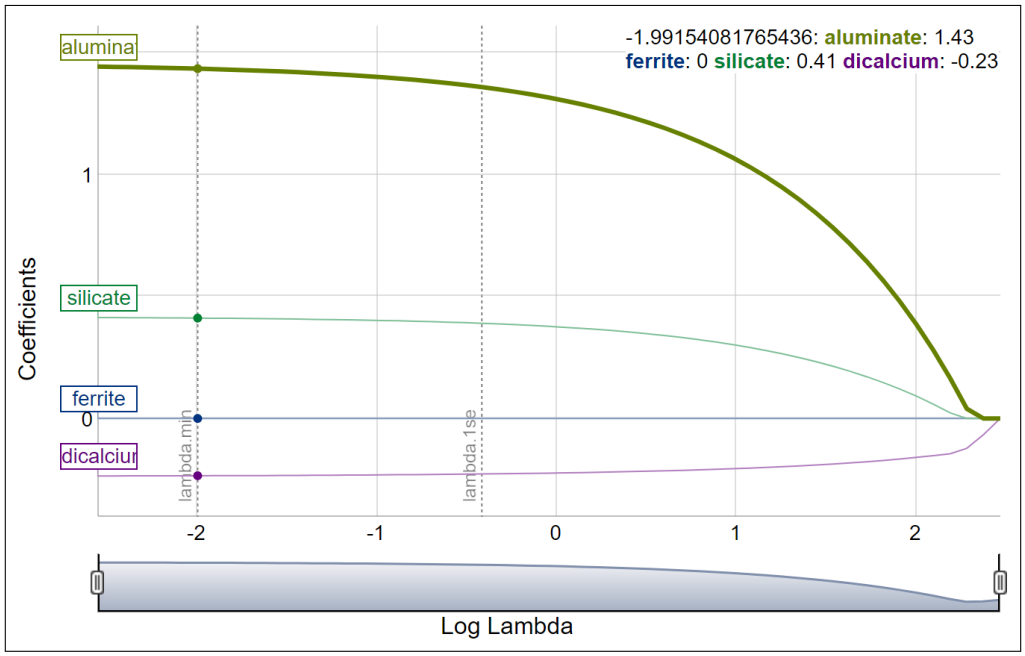


Figure 2: Hald cement coefficient progression plot for LASSO regression in R (using `glmnet` package).

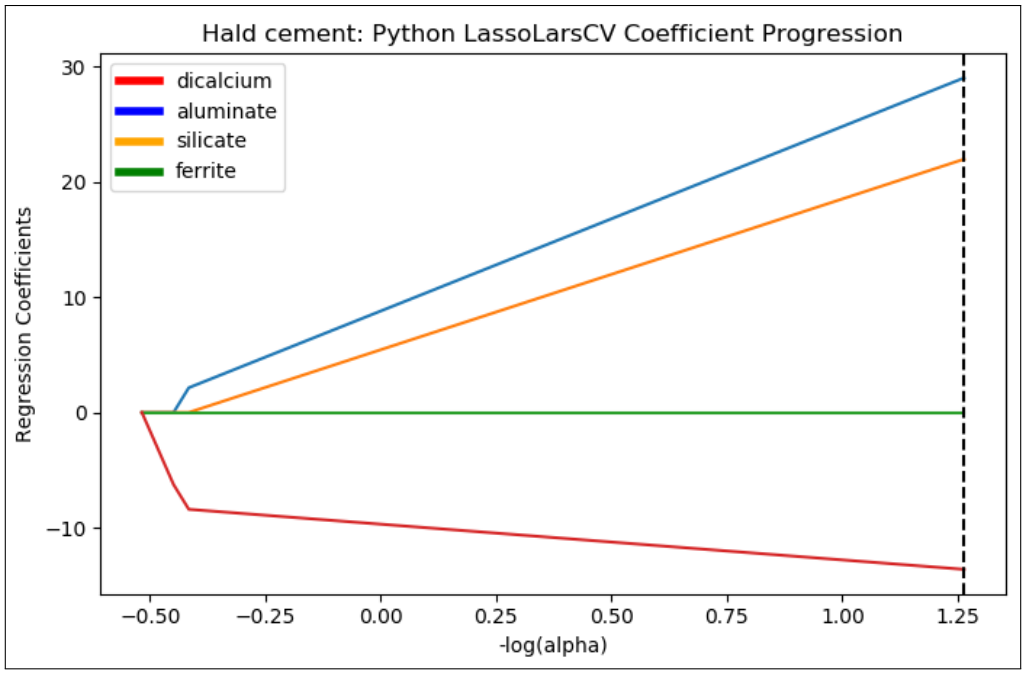


Figure 3: Hald cement coefficient progression plot for LASSO regression in Python (using `LassoLarsCV` package).

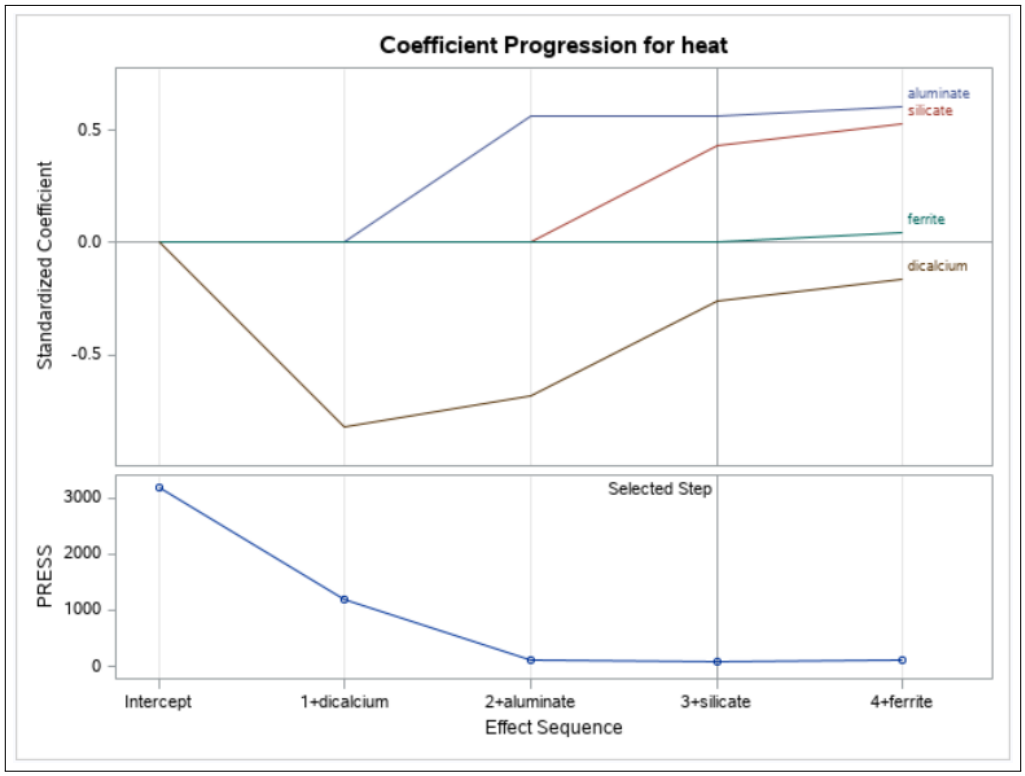


Figure 4: Hald cement coefficient progression plot for SAS regression in Python (using `glmselect` procedure).

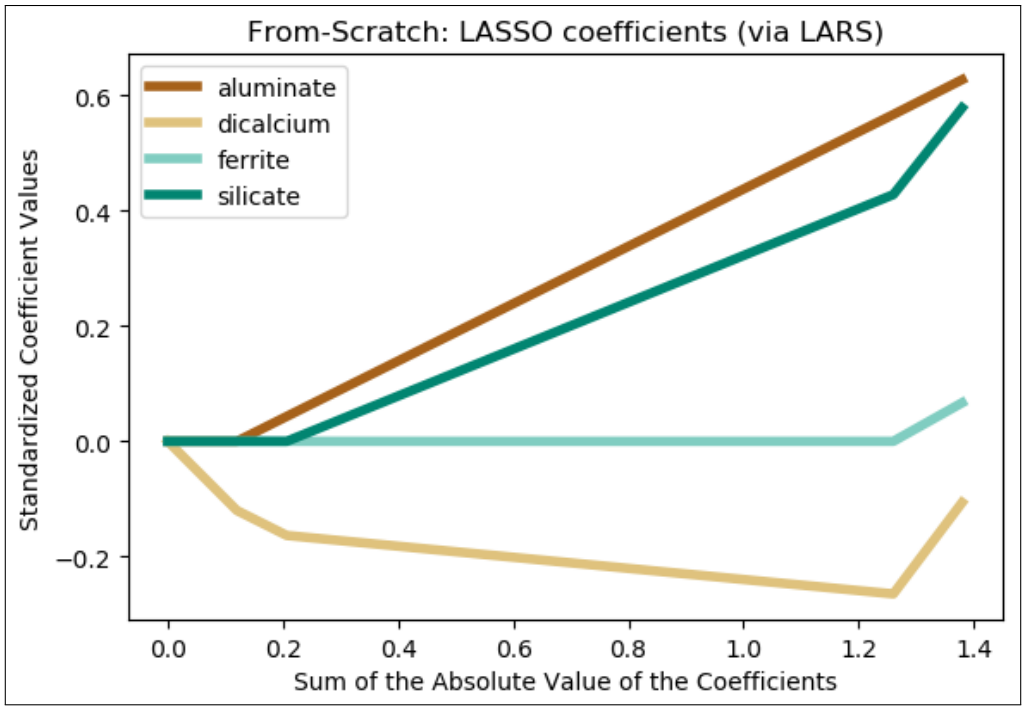


Figure 5: Hald cement coefficient progression plot for SAS regression in Python (from scratch implementation). (This is also visually encapsulates the results of my work on this project!)

4.2 Comparisons of Final Models

While the coefficient progression plots paint somewhat disparate results from each of the four implementations, the results detailed in Table 1 show high similarity. Especially within the pre-built packages in R, SAS, and Python, the results are nearly identical. We end up with a model similar to the following:

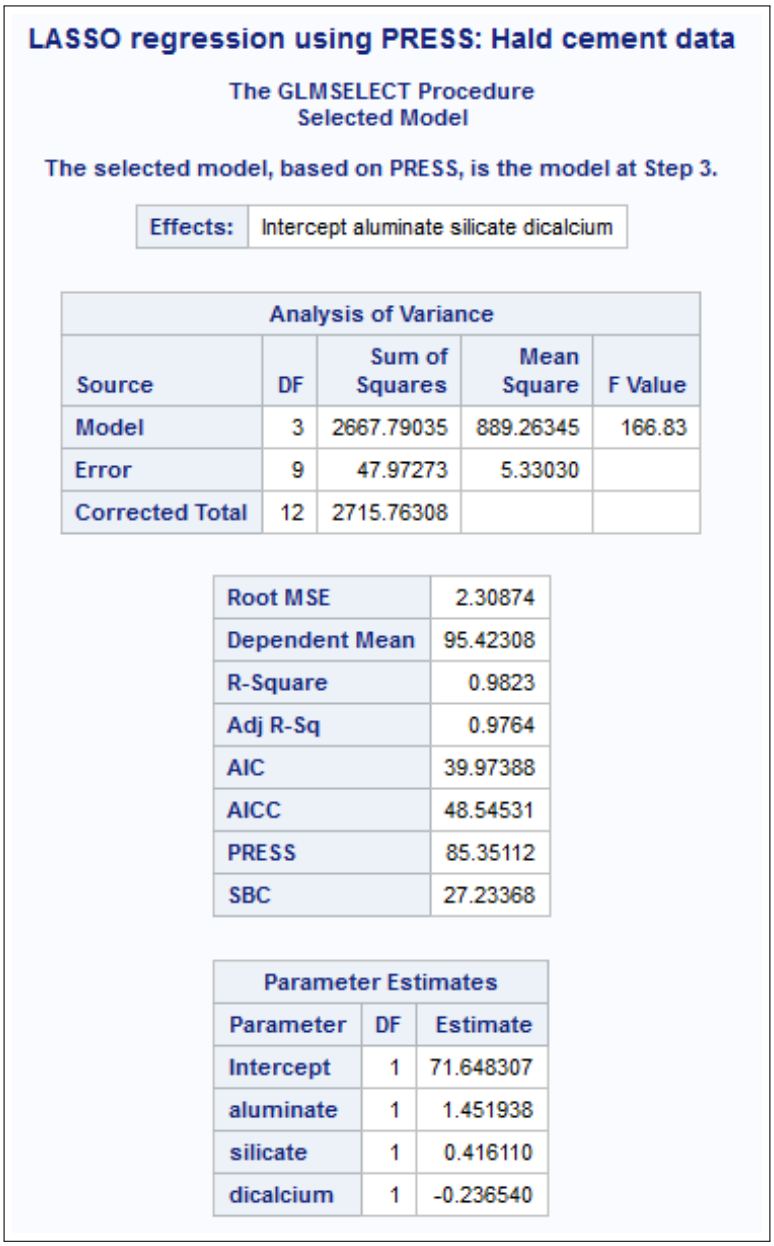
$$\left[\hat{heat} = 71.9 + (1.4)(aluminate) + (0.41)(silicate) - (0.24)(dicalcium)\right] \text{ (ferrite variable omitted)}$$

After the inconsistency in coefficient plots, it is refreshing to see such concordance between the pre-built LAR-Lasso packages. Although my from-scratch implementation doesn't yield quite the same model (especially due to the "no intercept issue" discussed in Subsection 3.2), it was encouraging to see that the silicate and dicalcium coefficients were close to what they ought to be. Overall, the pre-built packages achieved nearly identical final coefficients and MSE values, with SAS having the slight edge in performance with an MSE value of 3.691.

One other thing which surprised me: the MSE value given in the SAS output is incorrect. In Figure 6 we can see that the reported MSE of the best model from SAS is about 5.33. This didn't make any sense, as the coefficient values were nearly identical to those of the R and pre-built Python packages. To be sure, I took the final SAS coefficients and computed the MSE of this model by hand and got an MSE of 3.691, which ended up being slightly better than both R and Python, and was much better than the original SAS MSE of 5.33! I could not figure out why this was, as the output indicates that the final model is what we'd expect it to be: an intercept term, and coefficients for each variable except for ferrite. My best guess is that it has something to do with how cross-validated error was calculated.

Hald cement data: final models and performance						
Software	**intercept**	**aluminate**	**ferrite**	**silicate**	**dicalcium**	**MSE**
R	71.971	1.432	0.000	0.411	-0.234	3.723
SAS	71.648	1.452	0.000	0.416	-0.237	3.691
Python	72.220	1.424	0.000	0.407	-0.235	3.753
From-scratch	n/a	0.567	0.000	0.428	-0.264	n/a

Table 1: Final model comparisons for R, SAS, Python (pre-built), and Python (from-scratch) LAR-Lasso implementations.



LASSO regression using PRESS: Hald cement data

The GLMSELECT Procedure
Selected Model

The selected model, based on PRESS, is the model at Step 3.

Effects:	Intercept aluminate silicate dicalcium

Analysis of Variance				
Source	DF	Sum of Squares	Mean Square	F Value
Model	3	2667.79035	889.26345	166.83
Error	9	47.97273	5.33030	
Corrected Total	12	2715.76308		

Root MSE	2.30874
Dependent Mean	95.42308
R-Square	0.9823
Adj R-Sq	0.9764
AIC	39.97388
AICC	48.54531
PRESS	85.35112
SBC	27.23368

Parameter Estimates		
Parameter	DF	Estimate
Intercept	1	71.648307
aluminate	1	1.451938
silicate	1	0.416110
dicalcium	1	-0.236540

Figure 6: SAS `proc glmselect` output for Hald cement data, illustrating incorrect MSE value.

5 Conclusion

I learned a number of very valuable lessons from this project. A surprising takeaway was that the implementations and results for a certain model can vary widely across software packages, even for something as well-known as Lasso regression. I also had to swallow the fact that Python isn't always the best for every modeling task, even though it is my preferred language for statistical and machine learning modeling. This was a good reminder that a strong preference or proficiency with a certain language isn't necessarily a sufficient reason to exclusively use that language.

I also gained a lot of depth in my technical knowledge of Lasso, the LAR algorithm, and how they fit together. I had to learn the hard way that reading the LAR paper thoroughly and ***then*** moving onto coding would have been a much smarter route to take rather than jumping feet-first into coding with only a general understanding of the algorithm. However, by making some naive mistakes early on I think my learning on this topic will stick much deeper. By first trying the "uniform adjustment approach" (see Subsection 3.1) to coefficient updating I realized that I was missing something crucial. This prompted me to go back and carefully read the LAR paper, at which point I was more easily able to pick out that the *equiangular vector* is the secret sauce that makes the LAR algorithm so powerful. Additionally, this route of learning helped solidify my understanding of optimization methods that rely on many iterations (e.g. gradient descent, coordinate descent) versus methods that take few iterations but instead rely on clever computation and matrix inversion (e.g. Newton's method, LAR algorithm). Being able to connect what I'd learned in the optimization class I took in the fall to this problem was very satisfying!

I really enjoyed working on this project, and felt that I gained **a lot** from it. Becoming better at programming, understanding mathematics and optimization, as well as sharpening my academic paper reading skills and patience levels were all big benefits. Thank you for recommending this project Dr. Cutler. I loved this class; it was the perfect way to tie together my statistics education at Utah State!

References

[1] B. Efron, T. Hastie, I. Johnstone, and R. Tibshirani, "Least angle regression," *The Annals of statistics*, vol. 32, no. 2, pp. 407–499, 2004.

[2] J. Friedman, T. Hastie, and R. Tibshirani, *The elements of statistical learning*. Springer series in statistics New York, 2001, vol. 1, no. 10.

[3] H. Perkins, "Lars," https://github.com/hughperkins/selfstudy-LARS/blob/master/test_lars.ipynb, 2016.

[4] T. Hastie and J. Qian, "Glmnet vignette," *Retrieve from http://www. web. stanford. edu/˜ hastie/Papers/Glmnet_ Vignette. pdf. Accessed September*, vol. 20, p. 2016, 2014.

[5] R. A. Cohen, "Introducing the glmselect procedure for model selection," in *Proceedings of the Thirty-First Annual SAS Users Group International Conference*. Citeseer, 2006, pp. 4770–4792.

[6] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg *et al.*, "Scikit-learn: Machine learning in python," *the Journal of machine Learning research*, vol. 12, pp. 2825–2830, 2011.

A Appendix

SAS Code

```
/*********************************************
*********************************************
STAT 6100, Final Project, spring 2020
Hald cement analysis
Jared Hansen
*********************************************
*********************************************/

options nodate pageno=1; run;

/*********************************************
READING IN THE HALD CEMENT DATA
*********************************************/
data cement;
input aluminate silicate ferrite dicalcium heat;
label aluminate = "Percentage (by weight) of tricalcium aluminate";
label silicate = "Percentage (by weight) of tricalcium silicate";
label ferrite = "Percentage (by weight) of tetracalcium alumino";
label dicalcium = "Percentage (by weight) of dicalcium silicate";
label heat = "Heat evolve during hardening in calories per gram (RESPONSE)";
datalines;
 7  26   6  60   78.5
 1  29  15  52   74.3
11  56   8  20  104.3
11  31   8  47   87.6
 7  52   6  33   95.9
11  55   9  22  109.2
 3  71  17   6  102.7
 1  31  22  44   72.5
 2  54  18  22   93.1
21  47   4  26  115.9
 1  40  23  34   83.8
11  66   9  12  113.3
10  68   8  12  109.4
;
run;

*Apply LASSO regression to the data using LOOCV for selection;
title1 "LASSO regression using PRESS: Hald cement data";
proc glmselect data=cement plots=coefficients;
model heat = aluminate silicate ferrite dicalcium /
selection=LASSO(choose=PRESS steps=50 LSCOEFFS);
run;
```

R Code

```
#+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
# File name      : R_cementLASSO.R
# Author         : Jared Hansen
# R version      : 3.6.3
#
# DESCRIPTION: fitting LASSO model to the Hald cement data, generating coef
#              progression plot, and MSE of best model (chosen by mean Cv err).
#+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

# IMPORT STATEMENTS
#+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
library(glmnet)
library(ggplot2)
library(useful)
library(coefplot)
library(DT)

# Read in the data
#+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
path <- "C:/__JARED/__USU_Sp2020/stat6100_AdvRegression/finalProject/data/haldCement.csv"
cement <- read.table(path, header=TRUE, sep=",")
# Split into x_mat (predictor values) and y_vec (response values).
x_mat = as.matrix(cement[c('aluminate', 'ferrite', 'silicate', 'dicalcium')])
y_vec = as.matrix(cement['heat'])

# Fit LASSO using cv.glmnet. By specifying nfolds=nrow(x_mat)-1 we are
# effectively doing PRESS (leave-one-out CV) for our selection method.
#+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
cvfit = cv.glmnet(x_mat, y_vec, nfolds=(nrow(x_mat)-1), )
plot(cvfit, xvar="lambda", label=TRUE)
coefpath(cvfit, "Hald Cement")

# Determine the MSE of the best selected model (model with lowest mean CV error)
coef(cvfit, s = "lambda.min")
preds = predict(cvfit, newx=x_mat, s="lambda.min")
mse = mean((preds - y_vec)^2)
mse
```

Python Code: Using Pre-made Packages

```
'''
File name      : PythonPackages_cementLASSO.py
Author         : Jared Hansen
Python version : 3.7.3

DESCRIPTION:
Using pre-made Python packages to fit a LASSO model, create a coefficient
progression plot, and determine the true MSE of the best model (where best is
the model with the lowest MSE).
'''


#++++ IMPORT STATEMENTS ++++++++++++++++++++++++++++++++++++++++++++++++++++++
#++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.linear_model import LassoLarsCV
from matplotlib.lines import Line2D


#++++ PROCEDURAL CODE ++++++++++++++++++++++++++++++++++++++++++++++++++++++++
#++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
# Read in the data
data_path = 'C:/__JARED/__USU_Sp2020/stat6100_AdvRegression/finalProject/data'
cement = pd.read_csv(data_path+'/haldCement.csv')

# Split the data into a matrix of predictors and a vector of response values.
x_mat = cement[['aluminate', 'silicate', 'ferrite', 'dicalcium']]
y_vec = cement['heat']

# Create the LASSO model using LOOCV (PRESS).
model = LassoLarsCV(fit_intercept = True, normalize=True, cv=12,
                    precompute=False).fit(x_mat, y_vec)
# Take a look at the model coefficients for the best model.
print(dict(zip(x_mat.columns, model.coef_)))
model.intercept_
#best_betas = [72.2200604, 1.42376893, 0.40740646, 0., -0.2346226 ]

# Create the coefficient progression plot.
custom_lines = [Line2D([0], [0], color='red', lw=4),
                Line2D([0], [0], color='blue', lw=4),
                Line2D([0], [0], color='orange', lw=4),
                Line2D([0], [0], color='green', lw=4)]
m_log_alphas = -np.log10(model.alphas_)
ax = plt.gca()
```

```
plt.plot(m_log_alphas, model.coef_path_.T)
plt.axvline(-np.log10(model.alpha_), linestyle='--', color='k', label='alpha CV')
plt.ylabel('Regression Coefficients')
ax.legend(custom_lines, ['dicalcium', 'aluminate', 'silicate', 'ferrite'])
plt.xlabel('-log(alpha)')
plt.title('Hald cement: Python LassoLarsCV Coefficient Progression')
plt.show()
```

Python Code: From Scratch

```
'''
File name      : fromScratchLASSO_final.py
Author         : Jared Hansen
Python version : 3.7.3

DESCRIPTION:
Implementing LAR-selected LASSO from scratch on the Hald cement data.
Estimates a model at each step and generates a coefficient progression plot.
'''



#++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
#++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
#++++ IMPORT STATEMENTS ++++++++++++++++++++++++++++++++++++++++++++++++++++++
#++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
#++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
#import copy
from sklearn.preprocessing import StandardScaler
import numpy.linalg as npla
scaler = StandardScaler()
from matplotlib.lines import Line2D



#++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
#++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
#++++ DATA PREPARATION ++++++++++++++++++++++++++++++++++++++++++++++++++++++++
#++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
#++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
# Read in the data
data_path = 'C:/__JARED/__USU_Sp2020/stat6100_AdvRegression/finalProject/data'
cement = pd.read_csv(data_path+'/haldCement.csv')
#df = cement.copy(deep=True)

# IT MUST BE that each column (each individual predictor AND the response) is
# standardized to have mean 0 and unit length.
cement_scaled = scaler.fit_transform(cement)
cement = pd.DataFrame(cement_scaled, columns = cement.columns)

# Divide standardized values by L2 norm to get unit length of each column.
for col in cement:
    cement[col] = cement[col] / npla.norm(cement[col])
```

```
# Check to make sure that it worked.
print('Check to make sure each column is unit length')
for col in cement:
    vec = cement[col]
    length = np.sqrt((vec*vec).sum())
    print(col, ":", length)
print('Check to make sure each column sums to zero')
for col in cement:
    vec = cement[col]
    print(col, ":", vec.sum())

# Separate the response and predictors into a vector and matrix.
X_vars = ['aluminate', 'dicalcium', 'ferrite', 'silicate']
cement = cement[X_vars + ['heat']]
y = np.array(cement['heat'])
X = np.array(cement[X_vars])

# Define the number of predictors and the number of observations
m = len(X[0])
n = len(X)
y = y.reshape((n,1))

# Create a vector to store the initial predictions (which are all 0 to start).
mu_hats = np.zeros((n,1))



#++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
# At this point, we've added zero variables to the model. Now to add number 1.
#++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
# Create a list to store the MSE values for each successive model.
mse_vals = []
mse_vals.append(((y - mu_hats)*(y- mu_hats)).sum())
# Initialized a vector to store the current correlations of each predictor with
# the residuals (given in alphabetical order of predictor name.)
c_hats = np.dot(X.T, (y - mu_hats))
# Create a float to store the current largest absolute correlation.
bigC_hat = np.max(np.absolute(c_hats))
# Make a vector to track the signs associated with each predictor that is in
# the active set (the sign of that predictor's correlation with the residuals
# at the current step).
signs = np.zeros(m)
# Find the index in c_hats of the maximum absolute correlation.
j_hat = np.argmax(np.absolute(c_hats))
signs[j_hat] = np.sign(c_hats[j_hat])
# Create the active set list, active. Sort it whenever we append to it.
active = []
```

```
active.append(j_hat)
active.sort()
# Calculate the X_a matrix (for all active columns in a, multiply by the sign
# of that column's correlation with the current residuals).
X_a = np.dot(X[ : , active], np.diag(signs[active]))
# Calculate the G_a matrix and the A_a matrix.
G_a = np.dot(X_a.T, X_a)
ones_a = np.ones((len(active),1))
A_a = 1 / (np.sqrt(np.dot(np.dot(ones_a.T, npla.inv(G_a)), ones_a)))[0][0]
# Calculate the w_a and u_a vectors.
w_a = A_a * np.dot(npla.inv(G_a), ones_a)
u_a = np.dot(X_a, w_a)
# Comput the inner product vector a_vec.
a_vec = np.dot(X.T, u_a)
# Calculate the gamma value used for the update step.
gammas = []
# Determine the complement of active.
active_compl = [i for i in range(m) if i not in active]
for i in active_compl:
    gamma1 = (bigC_hat - c_hats[i][0])/(A_a - a_vec[i][0])
    gamma2 = (bigC_hat + c_hats[i][0])/(A_a + a_vec[i][0])
    if(gamma1 > 0.): gammas.append(gamma1)
    if(gamma2 > 0.): gammas.append(gamma2)
    print('var:', i)
    print('g1: ', gamma1)
    print('g2: ', gamma2)
gamma = min(gammas)
# Update our response predictions.
mu_hats += (gamma * u_a)
# Now that we have update predictions, let's create a data structure to track
# our coefficients. (We have to work backward from the predictions to get the
# coefficients).
models = np.zeros((1,m))
# Calculate the betas for the model associated with the new mu_hats.
new_betas = np.dot(npla.pinv(X), mu_hats).reshape(1,m)
models = np.concatenate((models, new_betas))



#++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
# At this point, we've added one variable to the model. Now to add number 2.
#++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
# Calculate and append the MSE of the current model.
mse_vals.append(((y - mu_hats)*(y- mu_hats)).sum())
# Recompute the current correlations of each predictor with
# the residuals (given in alphabetical order of predictor name.)
c_hats = np.dot(X.T, (y - mu_hats))
# Round these values so you get accurate results
```

```
c_hats = np.around(a = c_hats, decimals = 4)
# Create a float to store the current largest absolute correlation.
bigC_hat = np.amax(c_hats)
# Find the index in c_hats of the maximum absolute correlation which isn't
# already in the active set.
js = np.argwhere(np.absolute(c_hats).flatten() == bigC_hat).flatten().tolist()
# Determine the new j_hat index for this step
j_hat = [i for i in js if i not in active][0]
signs[j_hat] = np.sign(c_hats[j_hat])[0]
# Append the new index to the active set, sort the active set.
active.append(j_hat)
active.sort()
# Calculate the X_a matrix (for all active columns in a, multiply by the sign
# of that column's correlation with the current residuals).
X_a = np.dot(X[ : , active], np.diag(signs[active]))
# Calculate the G_a matrix and the A_a matrix.
G_a = np.dot(X_a.T, X_a)
ones_a = np.ones((len(active),1))
A_a = 1 / (np.sqrt(np.dot(np.dot(ones_a.T, npla.inv(G_a)), ones_a)))[0][0]
# Calculate the w_a and u_a vectors.
w_a = A_a * np.dot(npla.inv(G_a), ones_a)
u_a = np.dot(X_a, w_a)
# Comput the inner product vector a_vec.
a_vec = np.dot(X.T, u_a)
# Calculate the gamma value used for the update step.
gammas = []
# Determine the complement of active.
active_compl = [i for i in range(m) if i not in active]
for i in active_compl:
    gamma1 = (bigC_hat - c_hats[i][0])/(A_a - a_vec[i][0])
    gamma2 = (bigC_hat + c_hats[i][0])/(A_a + a_vec[i][0])
    if(gamma1 > 0.): gammas.append(gamma1)
    if(gamma2 > 0.): gammas.append(gamma2)
    print('var:', i)
    print('g1: ', gamma1)
    print('g2: ', gamma2)
gamma = min(gammas)
# Update our response predictions.
mu_hats += (gamma * u_a)
# Calculate the betas for the model associated with the new mu_hats.
new_betas = np.dot(npla.pinv(X), mu_hats).reshape(1,m)
models = np.concatenate((models, new_betas))



#++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
# At this point, we've added one variable to the model. Now to add number 3.
#++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
# Calculate and append the MSE of the current model.
mse_vals.append(((y - mu_hats)*(y- mu_hats)).sum())
# Recompute the current correlations of each predictor with
# the residuals (given in alphabetical order of predictor name.)
c_hats = np.dot(X.T, (y - mu_hats))
# Round these values so you get accurate results
c_hats = np.around(a = c_hats, decimals = 3)
# Create a float to store the current largest absolute correlation.
bigC_hat = np.amax(c_hats)
# Find the index in c_hats of the maximum absolute correlation which isn't
# already in the active set.
js = np.argwhere(np.absolute(c_hats).flatten() == bigC_hat).flatten().tolist()
# Determine the new j_hat index for this step
j_hat = [i for i in js if i not in active][0]
signs[j_hat] = np.sign(c_hats[j_hat])[0]
# Append the new index to the active set, sort the active set.
active.append(j_hat)
active.sort()
# Calculate the X_a matrix (for all active columns in a, multiply by the sign
# of that column's correlation with the current residuals).
X_a = np.dot(X[ : , active], np.diag(signs[active]))
# Calculate the G_a matrix and the A_a matrix.
G_a = np.dot(X_a.T, X_a)
ones_a = np.ones((len(active),1))
A_a = 1 / (np.sqrt(np.dot(np.dot(ones_a.T, npla.inv(G_a)), ones_a)))[0][0]
# Calculate the w_a and u_a vectors.
w_a = A_a * np.dot(npla.inv(G_a), ones_a)
u_a = np.dot(X_a, w_a)
# Comput the inner product vector a_vec.
a_vec = np.dot(X.T, u_a)
# Calculate the gamma value used for the update step.
gammas = []
# Determine the complement of active.
active_compl = [i for i in range(m) if i not in active]
for i in active_compl:
    gamma1 = (bigC_hat - c_hats[i][0])/(A_a - a_vec[i][0])
    gamma2 = (bigC_hat + c_hats[i][0])/(A_a + a_vec[i][0])
    if(gamma1 > 0.): gammas.append(gamma1)
    if(gamma2 > 0.): gammas.append(gamma2)
    print('var:', i)
    print('g1: ', gamma1)
    print('g2: ', gamma2)
gamma = min(gammas)
# Update our response predictions.
mu_hats += (gamma * u_a)
# Calculate the betas for the model associated with the new mu_hats.
new_betas = np.dot(npla.pinv(X), mu_hats).reshape(1,m)
models = np.concatenate((models, new_betas))
```

```
#++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
# At this point, we've added one variable to the model. Now to add number 4.
#++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
# Calculate and append the MSE of the current model.
mse_vals.append(((y - mu_hats)*(y- mu_hats)).sum())
# Recompute the current correlations of each predictor with
# the residuals (given in alphabetical order of predictor name.)
c_hats = np.dot(X.T, (y - mu_hats))
# Round these values so you get accurate results
c_hats = np.around(a = c_hats, decimals = 7)
# Create a float to store the current largest absolute correlation.
bigC_hat = np.amax(c_hats)
# Find the index in c_hats of the maximum absolute correlation which isn't
# already in the active set.
js = np.argwhere(np.absolute(c_hats).flatten() == bigC_hat).flatten().tolist()
# Determine the new j_hat index for this step
#j_hat = [i for i in js if i not in active][0]
# For whatever reason, the proper variable doesn't come out to have the highest
# correlation, and thus doesn't get added to the model. So I had to fix it by
# hand. Honestly not sure as to why this is. My guess is that it's a numerical
# precision issue.
j_hat = 2
signs[j_hat] = np.sign(c_hats[j_hat])[0]
# Append the new index to the active set, sort the active set.
active.append(j_hat)
active.sort()
# Calculate the X_a matrix (for all active columns in a, multiply by the sign
# of that column's correlation with the current residuals).
X_a = np.dot(X[ : , active], np.diag(signs[active]))
# Calculate the G_a matrix and the A_a matrix.
G_a = np.dot(X_a.T, X_a)
ones_a = np.ones((len(active),1))
A_a = 1 / (np.sqrt(np.dot(np.dot(ones_a.T, npla.inv(G_a)), ones_a)))[0][0]
# Calculate the w_a and u_a vectors.
w_a = A_a * np.dot(npla.inv(G_a), ones_a)
u_a = np.dot(X_a, w_a)
# Comput the inner product vector a_vec.
a_vec = np.dot(X.T, u_a)
# The LAR paper is actually unclear on what to do in this situation, meaning
# how we should calculate gamma once we've added the final variable to the
# active set. When we reach the last variable, the complement of the active set
# is the empty set, and we therefore cannot compute a gamma. Therefore, we take
# the typical gamma calculation formula, and no longer can draw c_hats
# (correlations) or a_vec values, and are just left with the value of the
# largest absolute correlation divided by the A_a constant.
```

```
gamma = bigC_hat / A_a
# Update our response predictions.
mu_hats += (gamma * u_a)
# Calculate the betas for the model associated with the new mu_hats.
new_betas = np.dot(npla.pinv(X), mu_hats).reshape(1,m)
models = np.concatenate((models, new_betas))
# Calculate and append the MSE of the current model.
mse_vals.append(((y - mu_hats)*(y- mu_hats)).sum())



#++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
#++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
#++++ CREATING THE COEFFICIENT PROGRESSION PLOT ++++++++++++++++++++++++++++++++
#++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
#++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

# Create a custom legend for the plot.
custom_lines = [Line2D([0], [0], color='#a6611a', lw=4),
                Line2D([0], [0], color='#dfc27d', lw=4),
                Line2D([0], [0], color='#80cdc1', lw=4),
                Line2D([0], [0], color='#018571', lw=4)]
# Calculate the sum of the absolute value of the coefficients at each step in
# the algorithm. The values of this vector will serve as our x-axis in the plot
sum_coef = np.sum(np.abs(models), 1)
# Generate the plot.
ax = plt.gca()
plt.plot(sum_coef, models[:,0], color='#a6611a', lw=4)
plt.plot(sum_coef, models[:,1], color='#dfc27d', lw=4)
plt.plot(sum_coef, models[:,2], color='#80cdc1', lw=4)
plt.plot(sum_coef, models[:,3], color='#018571', lw=4)
plt.title('From-Scratch: LASSO coefficients (via LARS)')
plt.ylabel('Standardized Coefficient Values')
ax.legend(custom_lines, ['aluminate',  'dicalcium', 'ferrite', 'silicate'])
plt.xlabel('Sum of the Absolute Value of the Coefficients')
plt.show()
```