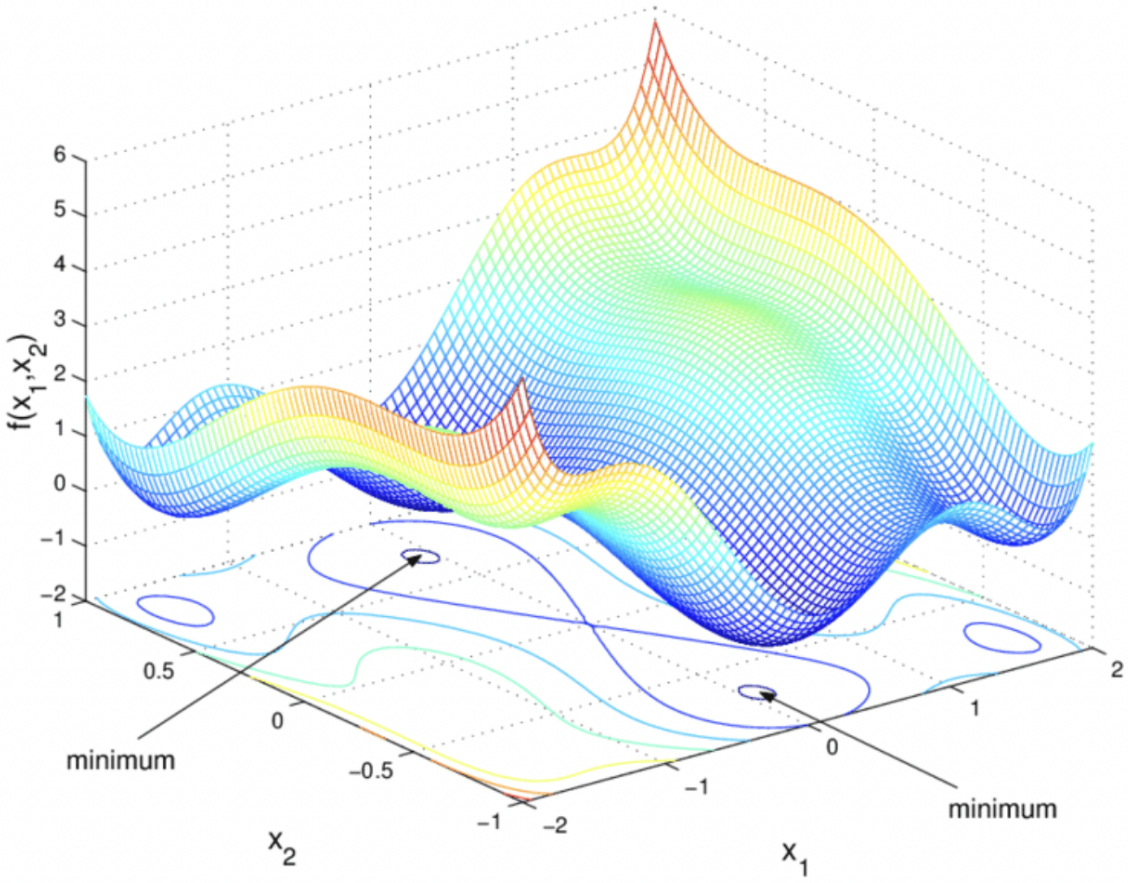


Non-Linear Programming for Variable Selection

By: Amritangshu, Carlee Allen, Harshit Jain, and Sunil Kamkar

Objective:

High-dimensional data processing is a significant challenge for engineers and scholars working in the Machine Learning field. By removing duplicate and redundant data, the variable selection provides a simple yet efficient solution to this problem. Removing extraneous data increases learning accuracy, decreases computation time, and enables a deeper understanding of the learning model or data. The objective of variable selection in Machine Learning is to identify the best set of features that enable one to build useful and constructive models of the subject one is trying to study. In this project, we examine two important variable selection strategies which is Lasso Regression and the Mixed Integer Quadratic Programming (MIQP) to determine if the additional LASSO 'shrinkage' component is indeed more advantageous than determining the 'optimal' set of variables to include in the regression model.

Methodology 1 - MIQP:

To begin with Mixed Integer Quadratic Programming we formulated the equations as follows.

Objective

$$\min_{\beta,z} \beta^T(X^T X)\beta + (-2y^T X)\beta$$

Decision Variables

a. β_j : coefficient of variable j $(m+1)$

b. z_j : selector whether variable j should be considered or not in the regression (m) - Binary

Total Decision variables - $2m+1$

Constraints

a. In the regression equation the sum of all variables should be equal to *k*.

$$\sum_{j=1}^{m} z_j \leq k$$

b. Coefficient of each variable j, where j >= 1, β_j should be less than $M\ z_j$

$$\beta_j - Mz_j \leq 0 \ for\ j = 1, 2, \ldots, m$$

c. Coefficient of each variable j, where j >= 1, β_j should be more than $-M\ z_j$

$$\beta_j + Mz_j \geq 0 \ for\ j = 1, 2, \ldots, m$$

Total Constraints - $2m+1$

```
def calculation_of_beta(X, y, k):
    #X: n*p
    #y: n*1
    #k: [5,10,15,20,25,30,35,40,45,50]
    #(m+1) values of β and the m values of z
    #b is the coefficients of the linear regression, z is the binary variable
    # formula for quadratic program
    Q = np.zeros((2*m+1, 2*m+1))
    A = np.zeros((2*m+1, 2*m+1))
    b = np.array([0]*(2*m+1))

    #b vector to be a (2m+1) x 1 vector.
    b[-1] = k

    #A matrix to be a (2m+1) x (2m+1) matrix.
    A[:,0] = 0
    A[0:m, 1:m+1] = np.diag(np.ones(m))
    A[0:m, m+1:2*m+1] = np.diag(np.ones(m))*M
    A[m:2*m, 1:m+1] = np.diag(np.ones(m))
    A[m:2*m, m+1:2*m+1] = np.diag(np.ones(m))*(-M)
    A[2*m:2*m+1, m+1:2*m+1] = [1]*m

    #B_t(X_t*X)B + (-2*Y_t*X)B
    #Q matrix : X_t*X to be a (2m+1) x (2m+1) matrix where the upper left corner of the matrix is equal to XTX, and al
    Xconstant = np.insert(X, 0, [1] * len(X), axis=1)
    Q[0:m+1, 0:m+1] = np.transpose(Xconstant) @ Xconstant

    #c matrix : 2*Y_t*X We also need the linear term of the objective to be a (2m+1) x 1 vector where the first (m+1) c
    constant_arr = np.zeros((len(Xconstant), 2*m+1))
    constant_arr[0:len(Xconstant), 0:m+1] = Xconstant
    c = -2 * np.transpose(y) @ constant_arr

    #(m+1) values of β and the m values of z
    #sense to be a (2m+1) x 1 vector.
    sense = np.array(['>']*m + ['<']*m +['<'])
    #vtype to be a (2m+1) x 1 vector.
    vtype=['C']*m+['C']+['B']*m
    #lower bound lb to be a (2m+1) x 1 vector In order to allow your βs to be negative or positive you must set the lb
    lb = np.array([np.NINF] + [-M]*m + [0]*m)
```

We computed the least square error for the given matrix of coefficients which we got in the previous beta equation and xtrain and ytrain as X,y .

$$\min_{\beta} \sum_{i=1}^{n} (\beta_0 + \beta_1 x_{i1} + \cdots + \beta_m x_{im} - y_i)^2$$

```
In [23]: #function to calculate the least square error given matrix of coeff, X, y
         def least_square_error(coeff, X, y):
             #coeff: m+1*1
             #X: n*p
             #y: n*1
             #formula for least square error
             Xconstant = np.insert(X, 0, [1] * len(X), axis=1)
             yhat = Xconstant @ coeff
             return np.sum((y - yhat)**2)
```

```
In [24]: least_square_error(calculation_of_beta(xtrain, ytrain, 40), xtrain, ytrain)
Out[24]: 524.3706415803392
```

By passing the number of variables to be selected in the model, we get the least square error for those variables. From the above result we can see that if we pass

40 variables, we get the least square error as 524.37. Similarly we can try minimizing the error by passing different numbers of variables.

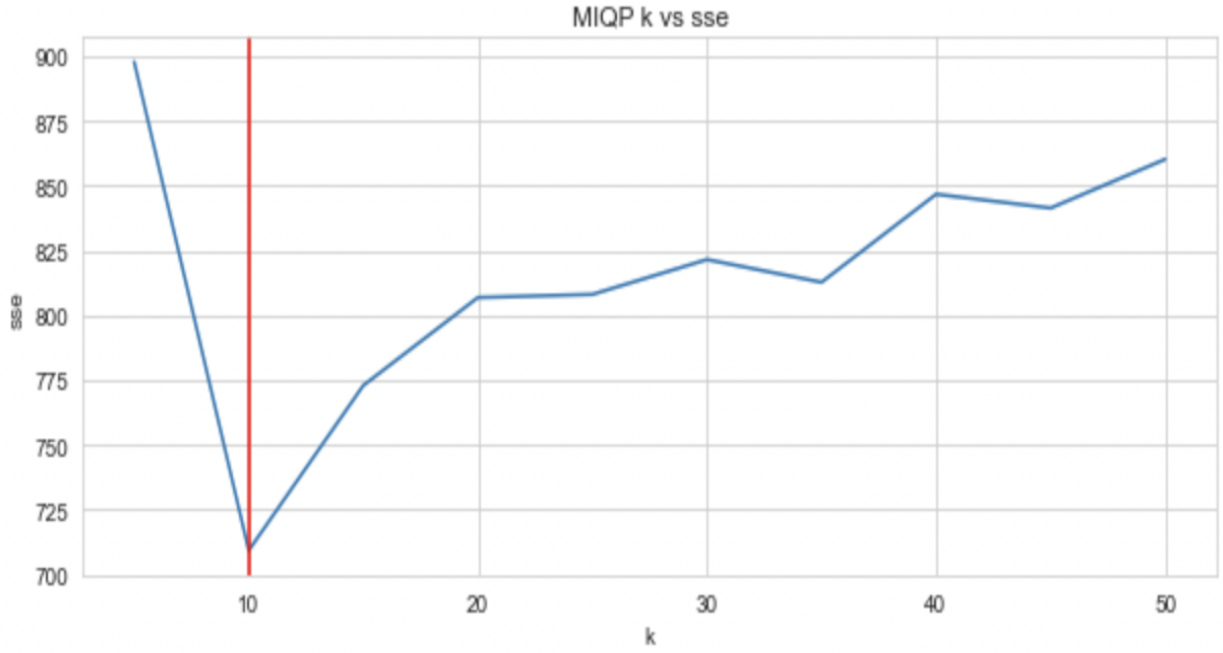
```
def cross_validation(xtrain, ytrain, k):
    #xtrain: n*p
    #ytrain: n*1
    #k: [5,10,15,20,25,30,35,40,45,50]
    #formula for cross validation
    #randomly shuffle your data and split it into 10 folds
    index = np.arange(xtrain.shape[0])
    np.random.shuffle(index)
    xtrain = xtrain[index]
    ytrain = ytrain[index]
    #split data into 10 folds
    xtrain_split = np.array_split(xtrain, n)
    ytrain_split = np.array_split(ytrain, n)
    df = pd.DataFrame(columns = ['coefficients','sse'])
    #calculate the least square error of each fold
    least_square_error_list = []
    for i in range(n):
        xtrain_cv = np.concatenate(xtrain_split[:i] + xtrain_split[i+1:])
        ytrain_cv = np.concatenate(ytrain_split[:i] + ytrain_split[i+1:])
        xtest_cv = xtrain_split[i]
        ytest_cv = ytrain_split[i]
        beta = calculation_of_beta(xtrain_cv, ytrain_cv, k)
        least_square_error_list.append(least_square_error(beta, xtest_cv, ytest_cv))
        df.loc[len(df)] = [beta,least_square_error_list[i]]
    #calculate the average least square error
    sse_sum = np.sum(least_square_error_list)
    average_least_square_error = np.mean(least_square_error_list)
    return df, sse_sum
```

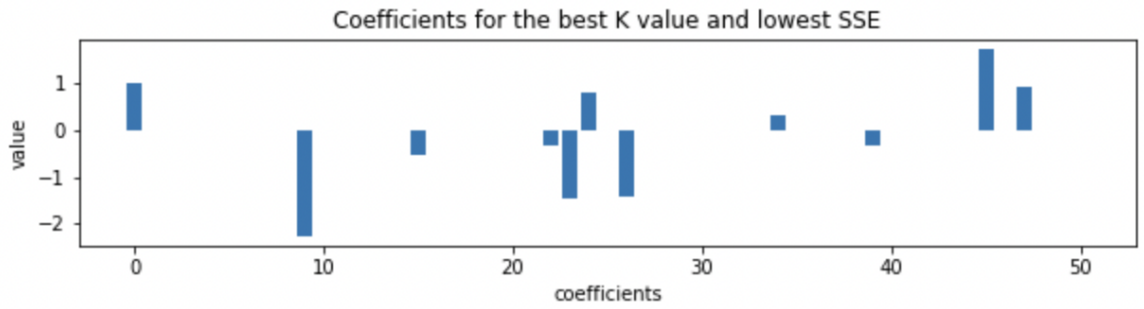
Then we created the list of k's which gives an option to choose the number of variables to be selected in the model and in our data we have close to 51 variables then we performed the 10-fold cross-validation on the training set then we calculated the sum of squared errors from all validation sets for each k. Then we calculated the value of k that corresponds to the minimum error.

```
#list_of_k = [5,10]
list_of_k = [5,10,15,20,25,30,35,40,45,50]
def best_k(xtrain, ytrain, list_of_k):
    #xtrain: n*p
    #ytrain: n*1
    #list_of_k: [5,10,15,20,25,30,35,40,45,50]

    sse_list = []
    mse_list = []
    #df to store k value and best coefficeint
    temp_df = pd.DataFrame(columns = ['k','coefficients','sse','non zero coefficients'])
    for k in list_of_k:
        df, sse = cross_validation(xtrain, ytrain, k)
        sse_list.append(sse)
        mse_list.append(sse/len(xtrain))

        #best set of coefficient in df
        best_coeff = df.loc[df['sse'].idxmin()]['coefficients']
        temp_df.loc[len(temp_df)] = [k, best_coeff, sse, np.count_nonzero(best_coeff)]

    k_select = list_of_k[sse_list.index(min(sse_list))]
    best_sse = min(sse_list)
    return k_select, best_sse,temp_df
```

	k	coefficients	sse	non zero coefficients
0	5	[1.072142212813712, 0.0, 0.0, 0.0, 0.0, 0.0, 0...	897.966422	6
1	10	[1.0281635192521825, 0.0, 0.0, 0.0, 0.0, 0.0, ...	709.164379	11
2	15	[0.9325400368355301, 0.0, 0.0, 0.2491478572782...	772.905255	16
3	20	[1.0053626377878528, 0.0, 0.0, 0.0, 0.0, -0.14...	806.882860	21
4	25	[0.9948449073334427, 0.0, 0.0, 0.0, 0.19572441...	808.094927	26
5	30	[0.998205934225931, 0.0, 0.0, 0.45682080614049...	821.558505	31
6	35	[1.0121696641951963, -0.19144951691991263, 0.0...	812.789028	36
7	40	[0.9742567623643477, 0.0, 0.1679322305367185, ...	846.792993	41
8	45	[0.9949815335909946, 0.0, 0.1200286270122203, ...	841.454740	46
9	50	[0.9853528782584199, 0.11951818444684648, 0.19...	860.312597	51

From the above table, we can see that K=10 has the lowest sum of squared error and non-zero coefficients are 11 including beta value. The same is also plotted below and we can visually see that k=10 generates the lowest SSE.





We created a visualization above of all the non-zero coefficients in MIQP selection using the coefficient values for the model with the lowest error for MIQP. The chart illustrates that direct variable selection utilizing mixed integer quadratic programming eliminates more variables by reducing their coefficients to 0.

We identified the value of k that had the least cross-validation error, and using that value of k, we fitted the MIQP model to the full training set. Next, we used the betas we discovered in this MIQP to predict the y values in the test set, and on our test set, we got a prediction error of 116.82

```
#fit model on entire train data with best k
best_coefficient=calculation_of_beta(xtrain, ytrain, k_select)

#use the best coefficient to predict yhat on test data
Xconstant = np.insert(xtest, 0, [1] * len(xtest), axis=1)
yhat = Xconstant @ best_coefficient

#print yhat
print(yhat)

#calculate sse on test data
sse_test_MIQP = np.sum((ytest - yhat)**2)
print('The sse on test data is', sse_test_MIQP)

#calculate r2 on test data
r2_test_MIQP = 1 - sse_test_MIQP/np.sum((ytest - np.mean(ytest))**2)
print('The r2 on test data is', r2_test_MIQP)
```

```
[ 6.17985878  5.09524299  3.28559532  3.75848539 -0.33297526 -5.14273683
 -3.14454357 -1.23806288  1.38511093 -0.44173854 -1.69500225  2.73035027
  0.74744903 -0.97192232 -0.68681528  8.04522381 -7.94698471  3.89063974
 -4.58142919 -3.21992082 -2.16211454  3.21686318 -3.19810533  0.19740731
 -2.35988844 -0.41999885 -1.9125216  -3.32418587 -3.14170972 -3.55379324
 -1.80842543 -0.37134301  1.8670808   5.04927886 -1.80005614  3.09427675
  4.38154309  2.6988627   1.6132886   5.97584637 -1.1973583   5.2232542
 -5.84899891 -1.14461528  4.51802998  4.18774866  4.12046008  0.61483809
  1.95723246 -1.54904383]
The sse on test data is 116.8271982276262
The r2 on test data is 0.8586682910020946
```

Methodology 2 - LASSO:

We did a 10-fold cross-validation on the training set to pick λ. Once you find the best value of λ, fit a LASSO model to the entire training set using that value of λ. With the βs you find in that LASSO model make a prediction of the y values in the test set.

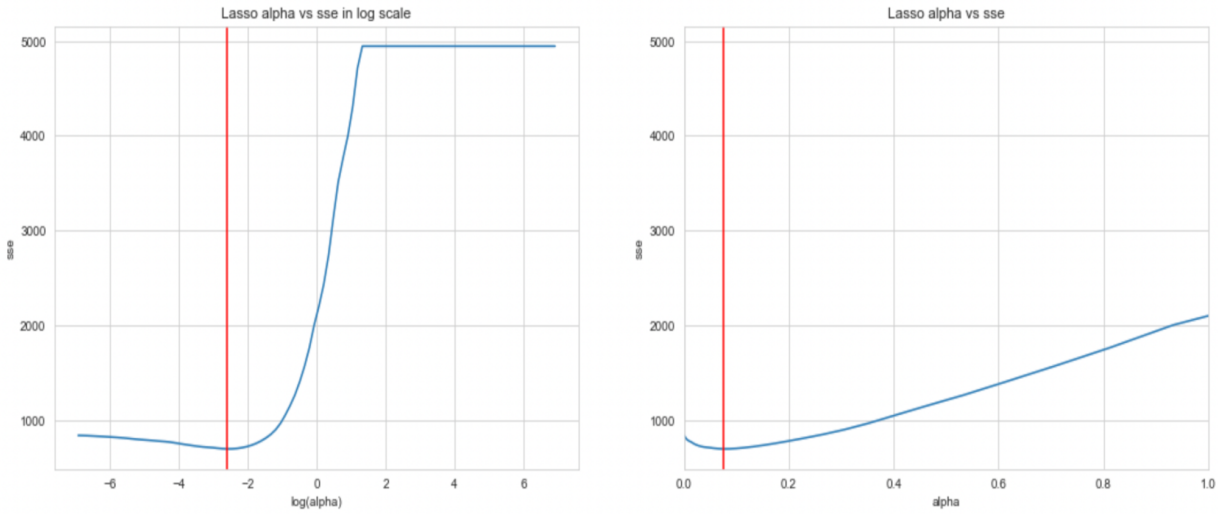
```
def cv_sse_score_lasso(X,y,alpha):
    #X: n*p
    #y: n*1
    #alpha: [0.0001,0.001,0.01,0.1,1,10,100,1000]
    #formula for cross validation
    sse = 0
    sse_sum=[]
    sse_list=[]
    kf = KFold(n_splits=n)
    for train_index, test_index in kf.split(X):
        X_train, X_test = X[train_index], X[test_index]
        y_train, y_test = y[train_index], y[test_index]
        lasso = Lasso(alpha=alpha)
        lasso.fit(X_train, y_train)
        yhat = lasso.predict(X_test)
        sse = np.sum((y_test - yhat)**2)
        sse_list.append(sse)
    sse_sum = np.sum(sse_list)
    average_least_square_error = np.mean(sse_list)
    return sse_sum
```

For example the function gives a sum of SSE value of 778 when we input an alpha of 0.01

```
cv_sse_score_lasso(xtrain, ytrain, 0.01)
✓ 0.8s
```

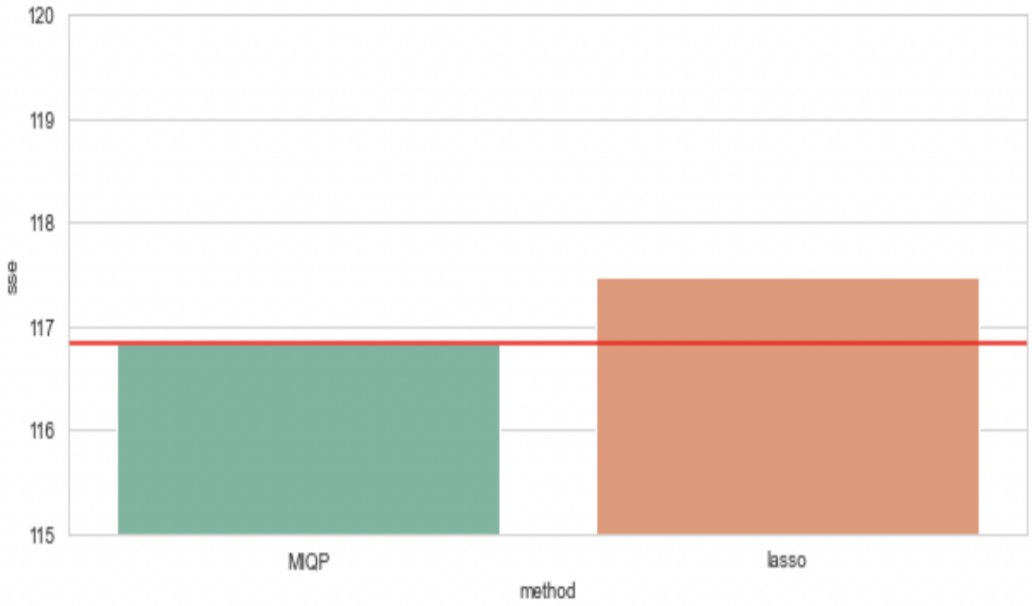
```
778.4000133076972
```

We looked at various values of alpha and the corresponding sum of SSE we get using cross-validation. We were able to conclude that we get the best SSE values at an alpha of 0.07564



Comparing Performance of LASSO Model & MIQP:

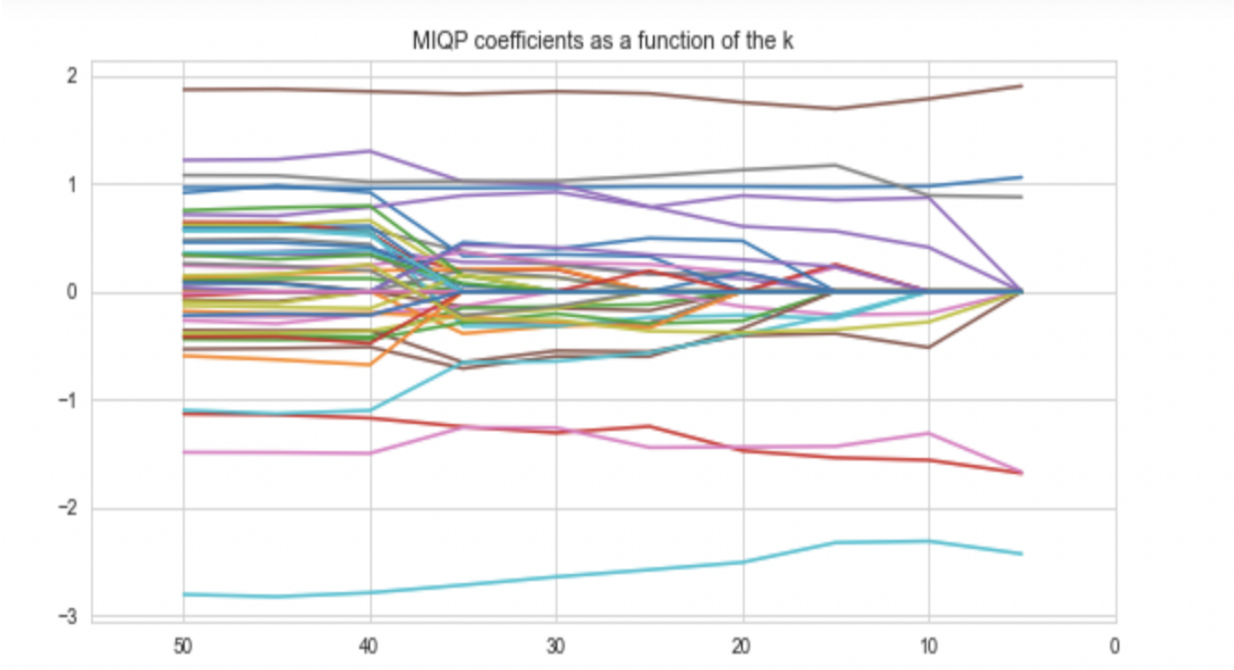
Considering at a high level, we may compare how well the MIQP and LASSO models performed when their sse, r2, and number of non zero coefficients were evaluated. The MIQP model, which has a sse of 116.83, a r2 of.859, and 10 non-zero variable coefficients, surpasses the LASSO model in all respects, despite the fact that the models first appear to perform quite similarly. In contrast, the LASSO model had 16 non-zero variable coefficients after computations, a sse of 117.47, and a r2 of.858. Overall, as we can see from the graph below, the MIQP model performed significantly better than the LASSO model since it had a smaller error, a greater predictive value, and better variable selection.

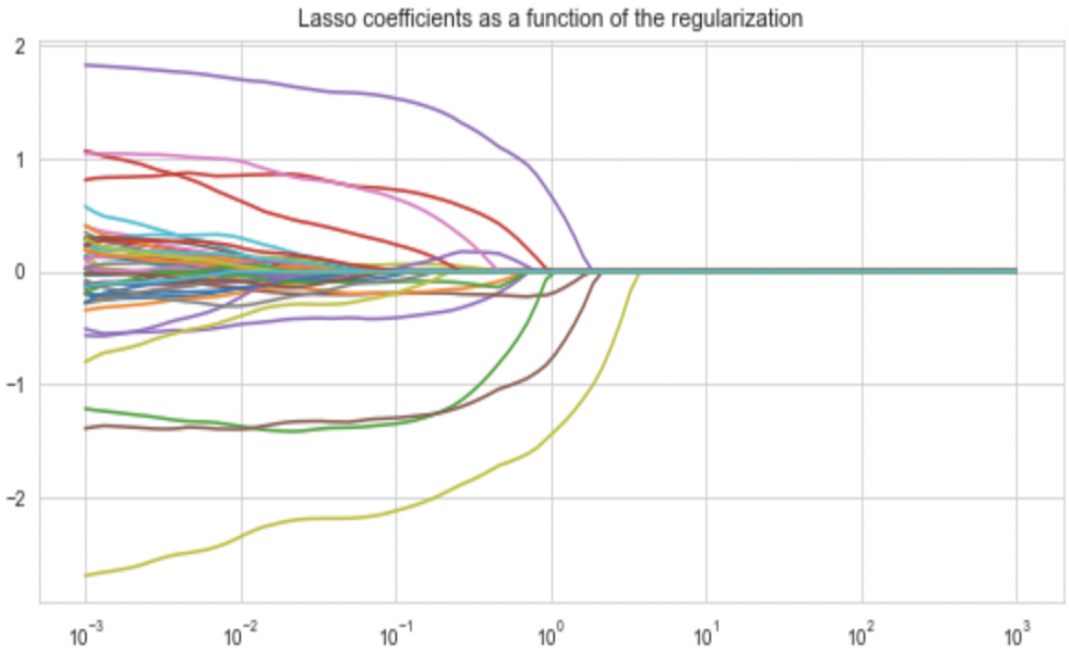


	method	sse	r2	number of non zero coefficients
0	MIQP	116.827198	0.858668	11
1	lasso	117.468790	0.857892	17

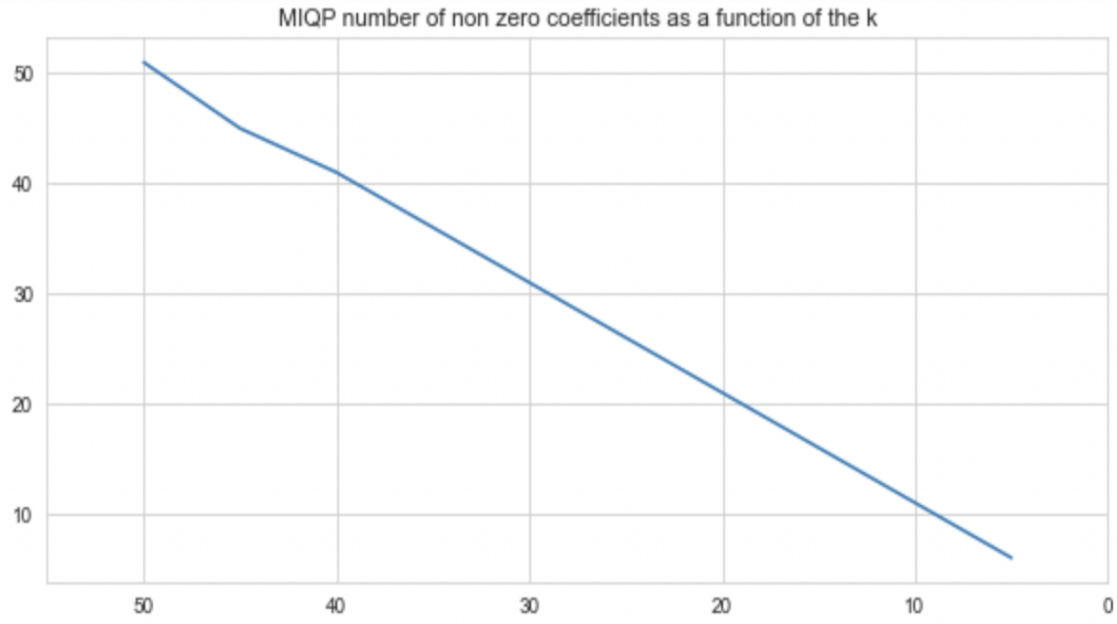
We decided to run further sub-plots in order to deeply investigate the coefficient estimates of both models as functions of their k values/regularizations. Through the sub-plots, we were able to evaluate the performance of each model in order to reconfirm our analysis above in that the two models perform relatively similarly, with MIQP's model performance being ultimately higher.

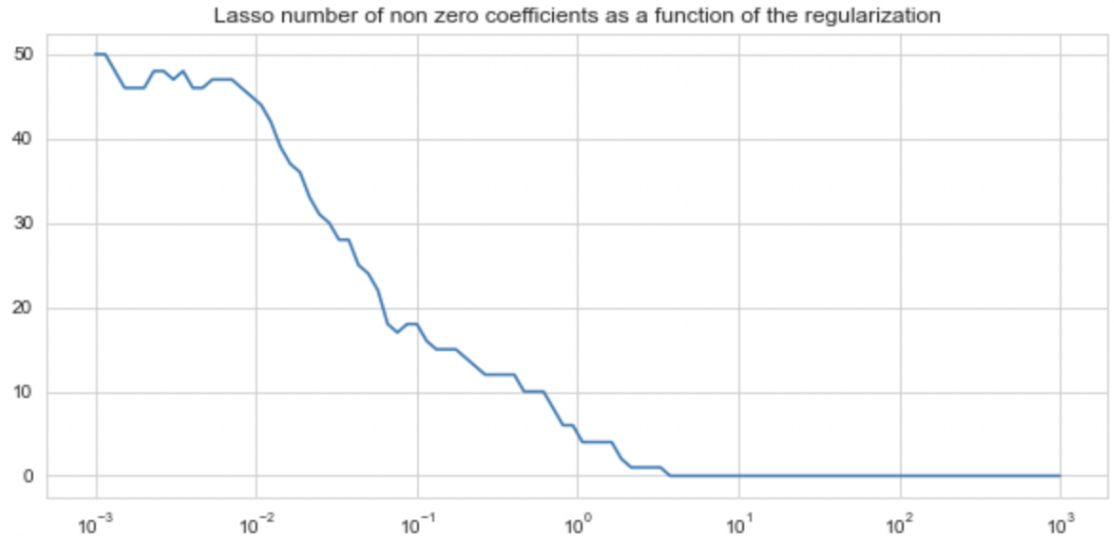
As seen below, the number of coefficients in MIQP is equal to the k value, however, in LASSO, all coefficients decrease to zero at an alpha that is almost equal to 5. Each colored line in both graphs corresponds to the value that each coefficient in our model's coefficients took. As we can see, more and more of our coefficients approach 0 as the value of k or alpha rises. However, the number of coefficients in MIQP that become zero is a direct function of our k value. For instance, in the graph below, if k is 10, all but 10 coefficients become zero.



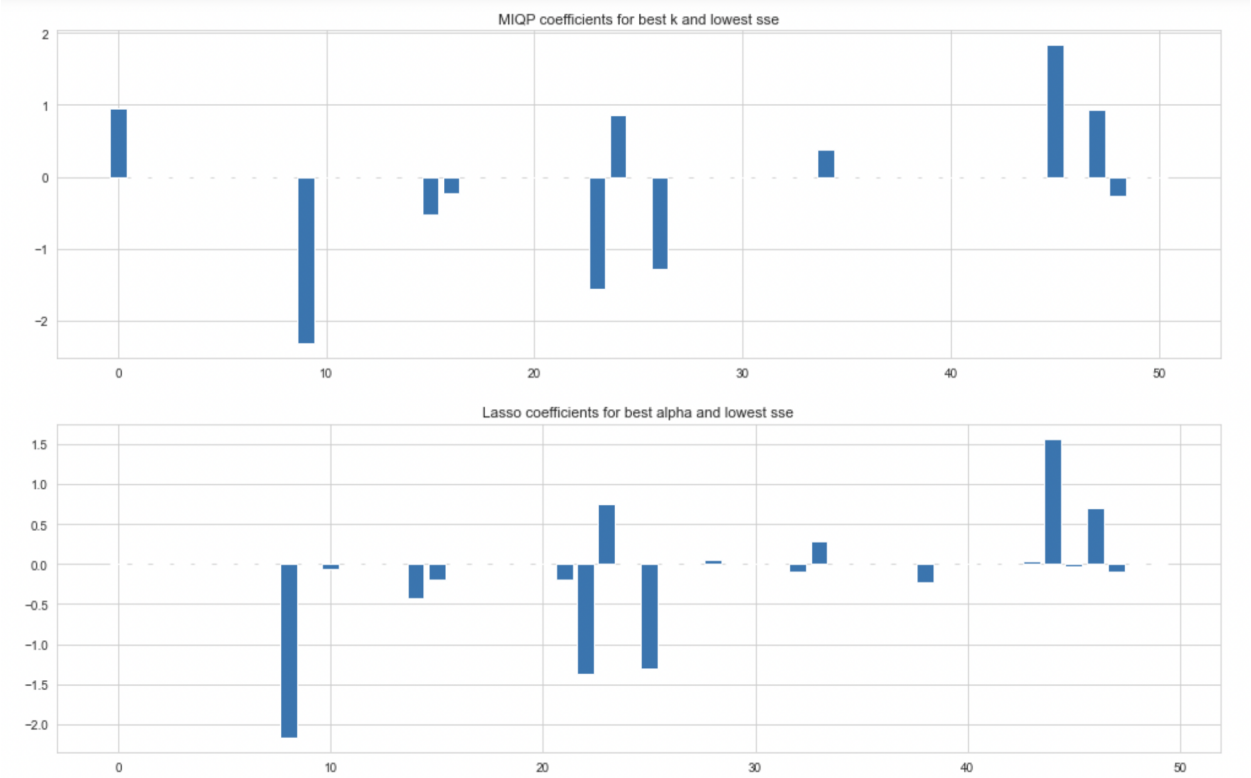


Following up from our earlier observations, In the graph below, we can see that the non-zero LASSO coefficient diminishes with the increase in regularization, and the MIQP ends up having the same non-zero coefficient equal to the value K.





Looking at the coefficients for the best models(lowest SSE) in LASSO and MIQP, In the below graph, we can see that the coefficients of the variables have a similar value in LASSO, as well as with MIQP. The values of the coefficients are nearly identical, which is a sign that both predict similar effects of the underlying features on the dependent variable.



Conclusion

We compared the output from MIQP and LASSO at a very granular level and have collated our observations in the below table for a quick comparison:

MIQP	LASSO
Non-Zero Coefficients in best case: 10	Non-Zero Coefficients in best case: 16
More control while handling variables	Less control while handling variable
Variable selection is not Automatic	Variable selection is Automatic
Takes more time to compute	Takes less time to compute
Variable selection is Explainable	Variable selection is not easily Explainable
Provides better result and model will be more rigid	Provides result which will be slightly low compared to MIQP

In Conclusion, we can say that MIQP performs slightly better than LASSO and as it does not introduce bias to the weight estimates, it makes a good alternative to LASSO. However, the computational time of direct variable selection is still higher than Lasso even with the advent of better solvers.

Recommendation:

The recommendation would be that if we have enough time to process data and have sufficient computing power then only we should move from LASSO to MIQP, which would give us better results.