Optimization Project 3
Analysis of LASSO vs Direct Variable Selection,

Daniel (Mu-An) Shen, Sahitya Vijayanagar, Brent Hensley, Archit Patel

Problem and Objectives

One of the most important tasks while working on predictive analytics using regression is that of selecting the independent variables that are used to predict the target variables, using a method called 'Variable Selection'. The most intuitive way to do this problem would be to select the most optimal variables through optimization methods that minimize an error metric. However, for decades, this solution has been considered infeasible due to it being computationally expensive and time consuming. This resulted in another method to solve variable selection called 'Regularization' where the coefficients for some features are reduced significantly to diminish their effect on the target variable.

Here, we explore the two ways to solve the problem of variable selection, namely:

1. Direct Variable Selection through Optimization - Mixed Integer Quadratic Programs
2. Regularization - LASSO

Method 1: Mixed Integer Quadratic Programs (MIQP)

We first implemented a direct variable selection method using Mixed Integer Quadratic Programs (MIQP), which sets the coefficient value of the variables and the number of variables being selected by minimizing the sum squared error.

Decision variables: continuous variables x_i, binary variables z_i

- x_i: Continuous variables of the variables' coefficient values
- z_i: Dummy variable controlling if the corresponding variable is selected or not. It forces the corresponding variable coefficient to be zero if z_i is zero.

Objective function: Minimize sum-of-squared error between the true dependent variable and the product of selected coefficients and their respective variables.

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

Constraints:

1. **Number of selected variables equals to k**

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

2. **Controlling if a variable is selected or not using z_i**

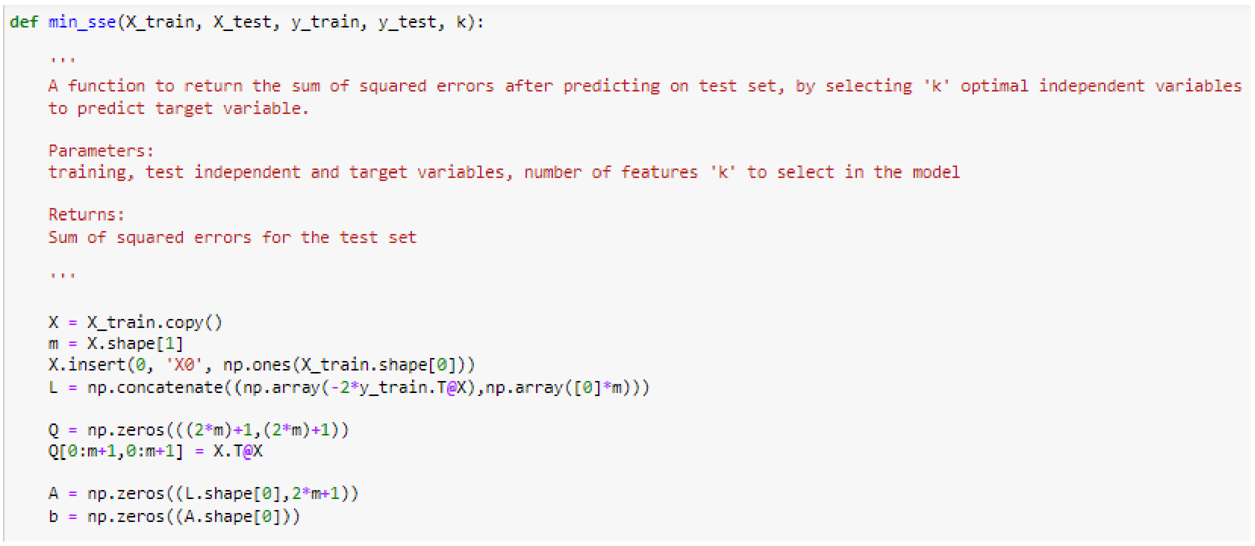
$$-Mz_j \leq \beta_j \leq Mz_j \quad for\ j = 1, 2, 3, \ldots, m$$

   When z_i equals to 0, the corresponding x_i has to be 0. On the other hand, when z_i is one, x_i is a continuous variable representing the variable coefficient. We apply the big M constraint method to this situation and apply a big enough M so that they are all non-binding constraints.

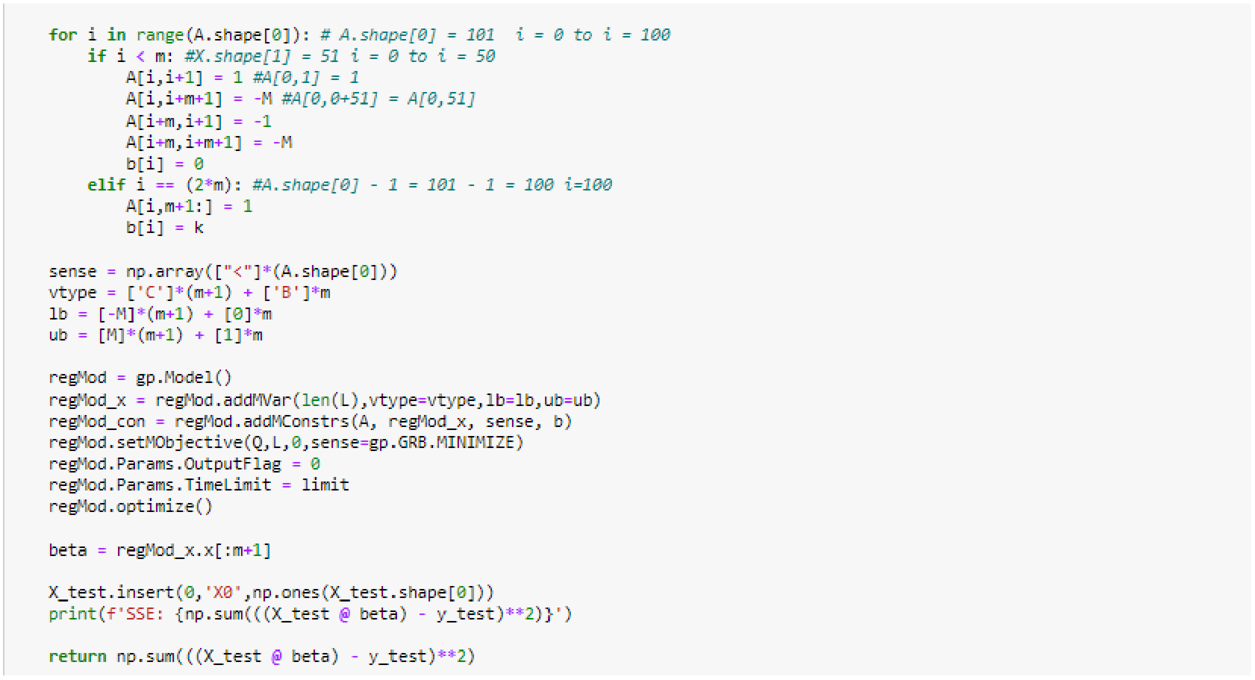
In order to formulate the problem, we create a matrix Q that represents the quadratic part of the problem, and an array L, representing the linear part of the problem. As shown in the equation below, the first term represents the Q matrix, and the second one represents the linear array L.

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

The code snippets representing the above is as follows:



```
def min_sse(X_train, X_test, y_train, y_test, k):
    '''
    A function to return the sum of squared errors after predicting on test set, by selecting 'k' optimal independent variables
    to predict target variable.

    Parameters:
    training, test independent and target variables, number of features 'k' to select in the model

    Returns:
    Sum of squared errors for the test set

    '''

    X = X_train.copy()
    m = X.shape[1]
    X.insert(0, 'X0', np.ones(X_train.shape[0]))
    L = np.concatenate((np.array(-2*y_train.T@X),np.array([0]*m)))

    Q = np.zeros(((2*m)+1,(2*m)+1))
    Q[0:m+1,0:m+1] = X.T@X

    A = np.zeros((L.shape[0],2*m+1))
    b = np.zeros((A.shape[0]))
```



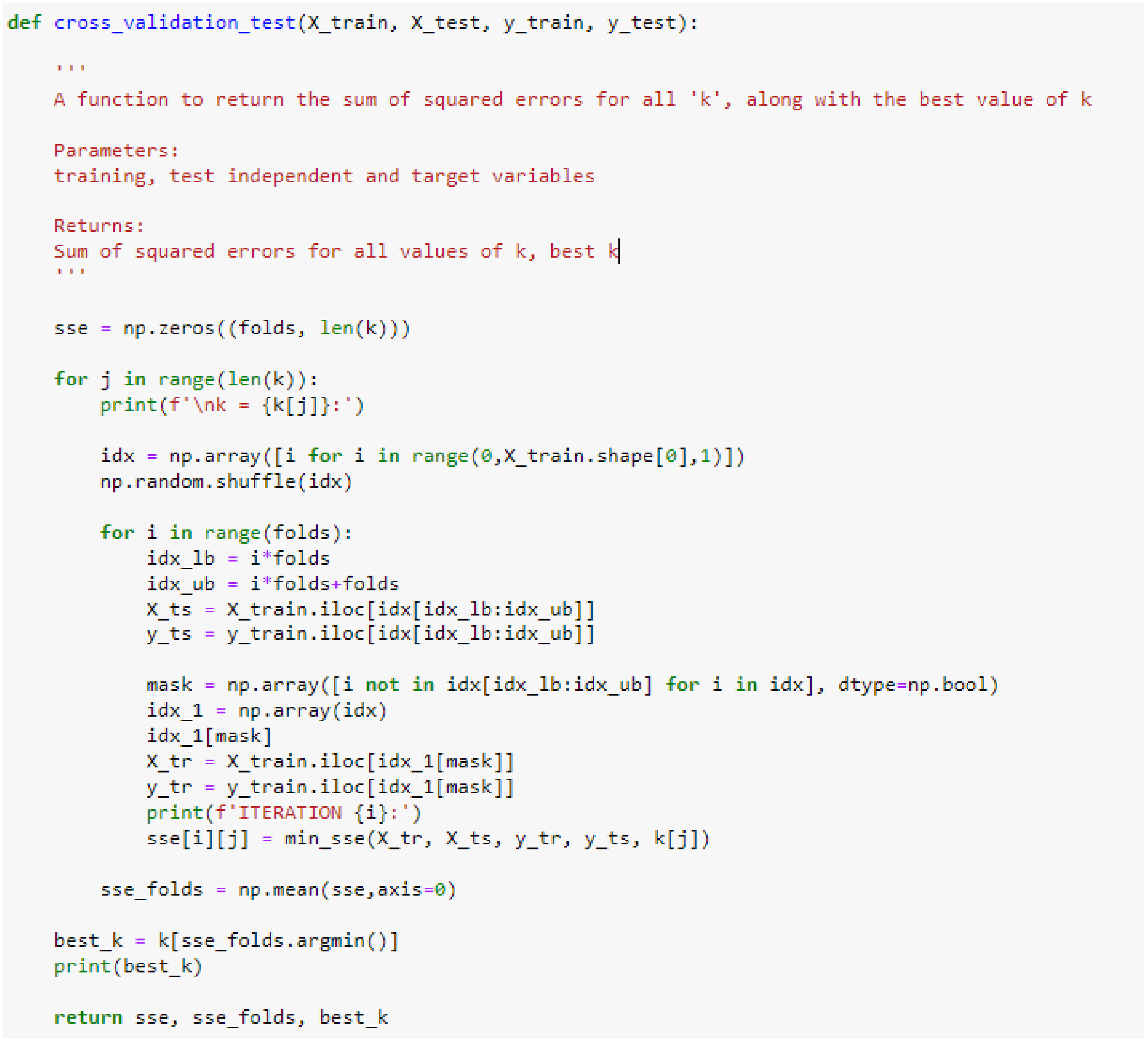
```
    for i in range(A.shape[0]): # A.shape[0] = 101  i = 0 to i = 100
        if i < m: #X.shape[1] = 51 i = 0 to i = 50
            A[i,i+1] = 1 #A[0,1] = 1
            A[i,i+m+1] = -M #A[0,0+51] = A[0,51]
            A[i+m,i+1] = -1
            A[i+m,i+m+1] = -M
            b[i] = 0
        elif i == (2*m): #A.shape[0] - 1 = 101 - 1 = 100 i=100
            A[i,m+1:] = 1
            b[i] = k

    sense = np.array(["<"]*(A.shape[0]))
    vtype = ['C']*(m+1) + ['B']*m
    lb = [-M]*(m+1) + [0]*m
    ub = [M]*(m+1) + [1]*m

    regMod = gp.Model()
    regMod_x = regMod.addMVar(len(L),vtype=vtype,lb=lb,ub=ub)
    regMod_con = regMod.addMConstrs(A, regMod_x, sense, b)
    regMod.setMObjective(Q,L,0,sense=gp.GRB.MINIMIZE)
    regMod.Params.OutputFlag = 0
    regMod.Params.TimeLimit = limit
    regMod.optimize()

    beta = regMod_x.x[:m+1]

    X_test.insert(0,'X0',np.ones(X_test.shape[0]))
    print(f'SSE: {np.sum(((X_test @ beta) - y_test)**2)}')

    return np.sum(((X_test @ beta) - y_test)**2)
```

Parameter selection for ‘k’
For the purpose of this problem, we look at reducing the sum-of-squared errors. The min_sse() uses optimization to identify the beta values using which we can test the model on the test dataset and report the test error.

One of the important parameters that needs to be passed to the min_sse() is the value of ‘k’ that dictates the number of optimal features that needs to be selected. In order to identify the best value of k, we perform cross validation, the steps of which are as follows:

1. Create a list of different possible ‘k’ values. k = [i for i in range(5,55,5)]
2. For each value of k in the list, repeat
    a. Shuffle and split the training dataset into a number of folds (10, in this case)
    b. Run different iterations ensuring each of the above splits are considered as the test set once, and all other splits as th training set
    c. Call the min_sse() function in each iteration and record the sum-of-squared errors
    d. Average the sse values of the 10 folds to give an average sum-of-squared error for each k
3. Select the best k based on the minimum value of the SSE identified in step 2d.

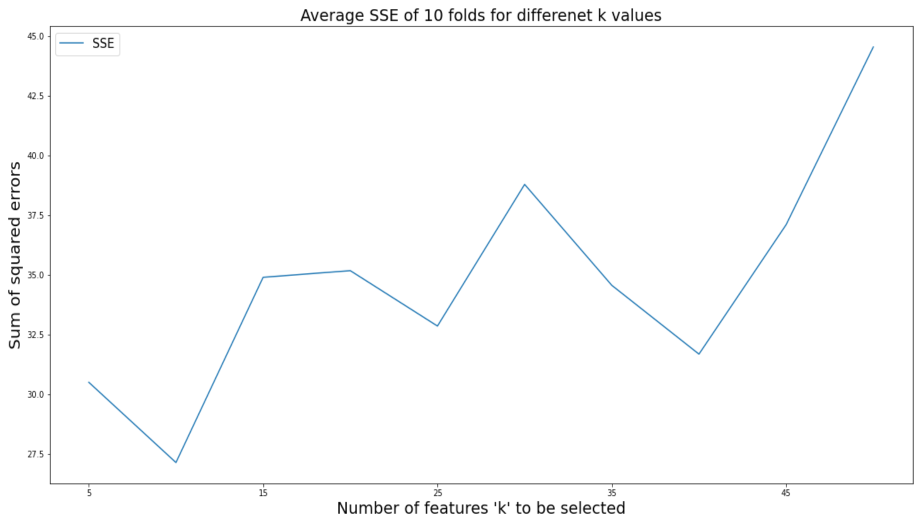
The code for the same is as follows:



```
def cross_validation_test(X_train, X_test, y_train, y_test):

    '''
    A function to return the sum of squared errors for all 'k', along with the best value of k

    Parameters:
    training, test independent and target variables

    Returns:
    Sum of squared errors for all values of k, best k
    '''

    sse = np.zeros((folds, len(k)))

    for j in range(len(k)):
        print(f'\nk = {k[j]}:')

        idx = np.array([i for i in range(0,X_train.shape[0],1)])
        np.random.shuffle(idx)

        for i in range(folds):
            idx_lb = i*folds
            idx_ub = i*folds+folds
            X_ts = X_train.iloc[idx[idx_lb:idx_ub]]
            y_ts = y_train.iloc[idx[idx_lb:idx_ub]]

            mask = np.array([i not in idx[idx_lb:idx_ub] for i in idx], dtype=np.bool)
            idx_1 = np.array(idx)
            idx_1[mask]
            X_tr = X_train.iloc[idx_1[mask]]
            y_tr = y_train.iloc[idx_1[mask]]
            print(f'ITERATION {i}:')
            sse[i][j] = min_sse(X_tr, X_ts, y_tr, y_ts, k[j])

        sse_folds = np.mean(sse,axis=0)

    best_k = k[sse_folds.argmin()]
    print(best_k)

    return sse, sse_folds, best_k
```

Through the program above alongside cross validation, we concluded that when K equals to 10 the program generates the smallest sum-of-squared error of 27.15 as shown in the table and plot below.

	SSE
5	30.513200
10	27.156444
15	34.904315
20	35.181969
25	32.865494
30	38.790432
35	34.568680
40	31.691171
45	37.104564
50	44.537307



As a result, we used the model with K=10 to predict our test data. The predicted values are shown below.

	y_true	y_predicted
0	7.107949	6.179859
1	5.796272	5.095243
2	1.598651	3.285595
3	2.532953	3.758485
4	0.590685	-0.332975

Using the MIQP method with the optimal k=10, we are able to achieve a sum-of-squared error of **116.83** between the test y values and predicted y values.

The 10 features selected using the MIQP method: ['X9' 'X15' 'X16' 'X23' 'X24' 'X26' 'X34' 'X45' 'X47' 'X48']

index	0.0	9.0	15.0	16.0	23.0	24.0	26.0	34.0	45.0	47.0	48.0
coefficient	0.972524	-2.308207	-0.518326	-0.204162	-1.559143	0.866973	-1.311919	0.408165	1.781475	0.887383	-0.282292

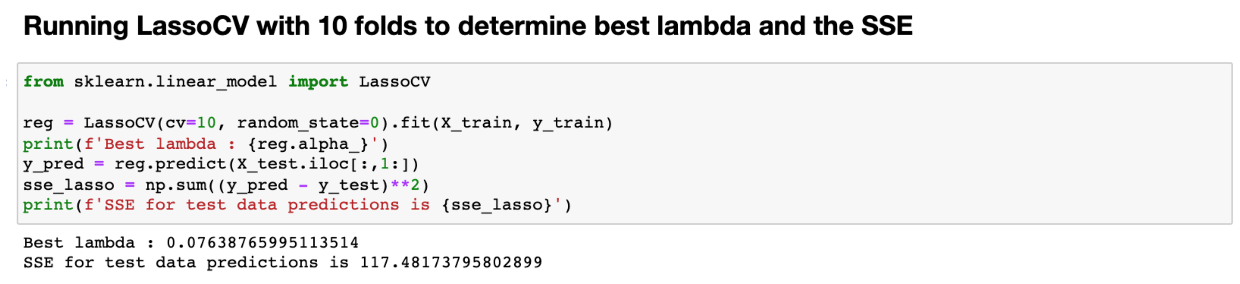
Method 2: LASSO

Objective Function:

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

We then implemented an indirect variable selection using LASSO, which minimizes beta by finding optimal values for lambda.

Using sklearn, we implemented LassoCV using 10 cross validation folds, and estimated the lambda value to be **0.0764**. Testing the fitted values on the test set, we get the sum-of-squared errors as **117.48.**

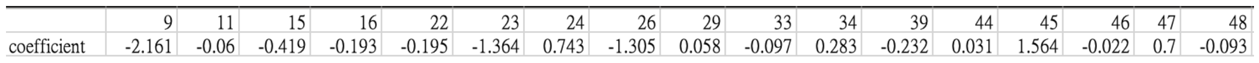


Running LassoCV with 10 folds to determine best lambda and the SSE

```python
from sklearn.linear_model import LassoCV

reg = LassoCV(cv=10, random_state=0).fit(X_train, y_train)
print(f'Best lambda : {reg.alpha_}')
y_pred = reg.predict(X_test.iloc[:,1:])
sse_lasso = np.sum((y_pred - y_test)**2)
print(f'SSE for test data predictions is {sse_lasso}')
```

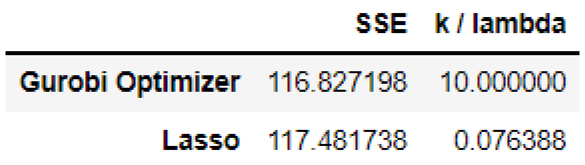
```
Best lambda : 0.07638765995113514
SSE for test data predictions is 117.48173795802899
```

The features selected using the LassoCV are: ['X9', 'X11', 'X15', 'X16', 'X22', 'X23', 'X24', 'X26', 'X29', 'X33', 'X34', 'X39', 'X44', 'X45', 'X46', 'X47', 'X48']



	9	11	15	16	22	23	24	26	29	33	34	39	44	45	46	47	48
coefficient	-2.161	-0.06	-0.419	-0.193	-0.195	-1.364	0.743	-1.305	0.058	-0.097	0.283	-0.232	0.031	1.564	-0.022	0.7	-0.093

The advantage of LASSO is that it is a quality optimization method with minimal computational time required. While its output may not always be the best, the benefit of the computational ease makes it a worthwhile option for a myriad of situations.



	SSE	k / lambda
Gurobi Optimizer	116.827198	10.000000
Lasso	117.481738	0.076388

Conclusion and Recommendations

Based on the results of the two methods, for their respective optimal models, we can see that MIQP method has a slightly lower SSE of 116.83 compared to Lasso's 117.48. MIQP is the more accurate method for variable selection. However, MIQP is computationally inefficient compared to Lasso since it takes significantly longer to run.

In conclusion, if our firm has the tools and equipment with high computational abilities, we recommend using the MIQP method for variable selection. The advent of better solvers allows the firm to use MIQP, which generates a more accurate outcome, in a more efficient manner. However, if our firm lacks such a computationally efficient solver, using Lasso for variable selection is still an effective method that generates a low sum squared error.