Optimization Project 3 - Variable Selection Report

Anushka Iyer - ai6646
Victor Lai - vbl249
Jason Nania - jn28878
Nicole Pham-Nguyen - np9967

Introduction:

Feature selection is an important aspect of regression in predictive analytics. It is used to help us overcome overfitting problems by reducing the number of features and hence the overall computational complexity of the model. Thus, by determining a smaller, more useful set of features that are actually beneficial to the model, we are able to predict our response variable with higher accuracy.

There are many feature selection techniques - Stepwise selection, Dimensionality reduction and Shrinkage. Of these, LASSO has undoubtedly been the method of choice over the last decade. The idea of Lasso regression is to optimize the cost function by reducing the absolute values of the coefficients. It adds an l1 penalty term to the cost function and α is a hyperparameter that tunes the intensity of this penalty term. With the help of the scikit learn package, we are able to perform this computation very quickly.

However, due to the tremendous advancements in optimization software, specifically, the ability to solve mixed integer quadratic programs (MIQP) in the recent past, we aim to explore its computational power in this report by using the Gurobi solver to find the best set of features by optimizing an ordinary least squares problem.

Goal:

Our company normally utilizes LASSO to achieve variable selection. However, our boss has become aware that the process of direct variable selection has become computationally efficient with the introduction of better solvers to the optimization field. In this report, we will perform both LASSO and direct variable selection on a subset of our data in order to determine which method we should prioritize for future projects.

General Approach:

Our training dataset consists of 250 rows and 51 columns (50 x-variables, 1 y-variable). Our testing data consists of the same variables, but only 50 rows. All of the features are continuous and numeric. The general approach for both the MIQP and LASSO methods will be to do a 10-fold cross validation to find the optimal value of k or λ. Then utilizing the optimal values of k or λ from the cross validation, we will fit βs to the entire training set. With our optimized β values we can then make predictions of the y value on the test data. We then are able to calculate

the mean squared error of our predicted y values and the actual y values in the test set. Finally, we could then compare the mean squared error of the MIQP model and the LASSO model to make a comparison of the two models, i.e. their advantages and disadvantages.

MIQP Details:

Below we will discuss the various components of our MIQP model.

Cross Validation

In order to pick k or λ, we manually wrote a 10-fold cross validation on the training set, in order to decide the optimal values of k that would best fit the entire training set. The code for our cross validation is shown below, including the split of the training set and of the manual split of the validation.

```
for k in range(5, 51, 5):
    shuffle = np.random.choice(range(ndata),size=ndata,replace=False)
    sse_test = 0
    sse_train = 0
    for j in range(folds):
        data_size = int(ndata/folds)
        val = total_rows[j*(data_size):(j+1)*data_size]
        train_rows = list(set(total_rows) - set(val))
        validation_set = train_df.iloc[shuffle[val],:]
        validation_x = train_df.iloc[shuffle[val],:].drop(columns = 'y')
        validation_y = train_df.iloc[shuffle[val],:].filter(like = 'y')
        validation_y = validation_y.to_numpy().T[0]
        train_set = train_df.iloc[shuffle[train_rows],:]
        train_x = train_set.drop(columns = 'y')
        train_y = train_set.filter(like = 'y')
```

Objective Function

For this particular problem we will be minimizing the Sum of Squared Errors (SSE).

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

In order to pose this minimization function to gurobi we need to alter the formula using linear algebra. This objective function was obtained using the normal equations of the sum of squared errors; however, this objective function is not the exact same function as the objective function shown above. This function does have the same minimizer as the sum of squared errors, but with a different minimum. Thus, we will compute the βs from the optimization problem with each k-value and manually calculate the sum of squared errors in order to determine the best number of ks to use.

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

Decision Variables

Next, we determined that our optimization problem will have 2m (100) decision variables. 50 continuous β_j variables which will determine the weights of our X variables, 50 binary Z_j variables which will be used to constrain the number of variables picked based on our k value, and one β column to represent the constant.

The code implementation for our decision variables is shown below:

```
#Q matrix
rows = 2*m+1
cols = 2*m+1
q_matrix = np.asarray([[0]*cols]*rows, dtype = "float") #needed to make python interpret as float instead of int
q_matrix[:m+1,:m+1] = x_arr.T @ x_arr

#setting up linear term
linear_term = np.zeros(2*m+1)
linear_term[:m+1] = (-2)* y_arr.T @ x_arr
```

Constraints

1. **Big M constraint:** For this problem we utilized a big M constraint which allows us to set up a constraint that forces the corresponding values of β_j to be zero if Z_j is zero. Thus, we tried different values of big M until we settled on M = 100 since that forces the corresponding β values to be equal to zero and ensures that no value of β is equal to M or -M. We tested a value of M = 100, and that worked well for our problem, and any value greater than 100 for this problem functions the same as M = 100, so we kept M = 100.

$$s.t. -Mz_j \leq \beta_j \leq Mz_j \quad for\ j = 1, 2, 3, \dots, m$$

The code implementation for the Big M constraint is shown below and its usage in the constraint matrix:

```
M = 100

#A matrix
A = np.zeros((2*m+1,2*m+1)) # row 0 is all 1s already
A[0,-m:] = [1]*m # constraint on z
A[:m,1:m+1] = np.eye(m)
A[m:-1,1:m+1] = np.eye(m)
A[:m,m+1:2*m+1] = np.eye(m)*(-M)
A[m:-1,m+1:2*m+1] = np.eye(m)*(M)
A[-1, m+1:] = 1
```

2. **k Constraint:** In order to test the best number of variables to select we needed to test k = [5,10,15,20,25,30,35,40,45,50]. We achieve this constraint by taking the sum of our Z_j and setting the output less than or equal to k. In our code, we set the last row of our constraint matrix to be the k-constraint. As shown above, the coefficients for all of the z-constraints are equal to one, and the sum of those z-constraints must be less than or equal to the value of k.

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

$$z_j \text{ are binary.}$$

```
# setting up b
b = [0]*(2*m+1)
b[-1] = k
b_arr = np.array(b)

#setting sense
sense = ['<']*(m) + ['>']*(m) + ['<']
```

3. **Negativity Constraint:** Since Gurobi assumes that all decision variables are non-negative, and β can be negative, we needed to set the lower bound value of the model to be -M for the first 2-51 decision variables, since the first decision variable was for the constant which can be any number, and the last 51-101 decision variables were the z-variables. This is shown in our code below.

```
mod_x = model.addMVar(2*m+1,vtype=np.array(['C']*(m+1) + ['B']*m),
                      lb=np.array([np.NINF]+[-M]*(m)+ [np.NINF]*m))
```

Calculation of SSE

The calculation of SSE was performed by performing the following matrix multiplication on the vector (Xβ - y).

$$(X\beta - y)^T * (X\beta - y)$$

The code implementation of the calculation is shown below for both the training set and the validation set.

```
#calculate the SSE
    sse_validation += (qval_matrix @ my_coeffs - validation_y).T @ (qval_matrix @ my_coeffs - validation_y)
    sse_train = (x_arr @ my_coeffs - y_mat).T @ (x_arr @ my_coeffs - y_mat)
```

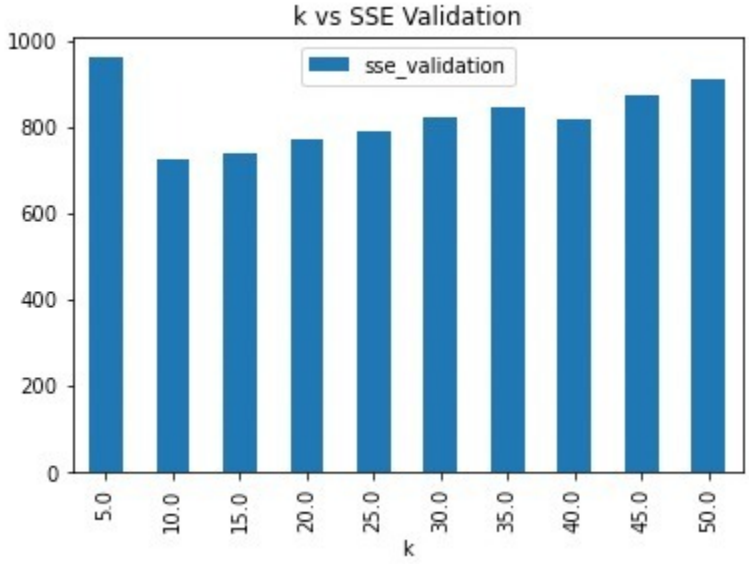
Results

Our cross validation indicated that a k=10 achieves our lowest error on the validation set.

Our β values for k = 10 are shown below.

```
[ 0.96362703  0.          0.          0.          0.          0.
  0.          0.          0.         -2.26096553  0.          0.
  0.          0.          0.         -0.59345415  0.          0.
  0.          0.          0.          0.         -0.29875027 -1.50146728
  0.8829761   0.         -1.40975206  0.          0.          0.
  0.          0.          0.          0.          0.34327789  0.
  0.          0.          0.          0.          0.          0.
  0.          0.          0.          1.77252451  0.          0.88444937
 -0.23396681  0.          0.        ]
```

The comparison of k vs SSE Validation is shown in the bar graph below.



In the graph above, we can see that k = 10 has the lowest SSE and as k increases, so does the validation error, with a slight dip at k = 40. This could suggest that k = 40 includes more relevant variables for the testing data than the lower k values, since the k values and the β values for each variable was computed using the training data. In addition, as the value of k increases, we do see a change in the stock selection. This is due to the idea that as there are more stocks able to be picked, there are more opportunities for a lower minimum than the previous k with that certain selection of stocks. However, in our problem, since k = 10 had the lowest error *and* was a lower k-value than the rest of the k-values tested, we then fit our model with k = 10 on all of the training data and achieved an MSE of 2.34 on the test set.

LASSO Details:

In order to implement LASSO we utilized the LassoCV function from the sklearn package. We did a 10-Fold cross validation on our train data, which resulted in an optimal alpha value of 0.076.

```
the alpha is 0.07638765995113514
the intercept is 1.0015522007348236
number of x variables is: 17 and the coefficients are [-0.         -0.          0.          0.         -0.          0.
 -0.         -0.         -2.16054765  0.         -0.05964031 -0.
 -0.         -0.         -0.41912484 -0.19325408  0.          0.
 -0.          0.          0.         -0.19517759 -1.36388738  0.7425965
 -0.         -1.30481574 -0.          0.          0.05798283  0.
 -0.          0.         -0.09737839  0.28341629  0.          0.
  0.          0.         -0.23157873  0.         -0.          0.
  0.          0.03078191  1.56362172 -0.02160033  0.69992447 -0.09289745
  0.          0.        ]
Test MSE: 2.3496347591605806
```

We then used the βs returned in the model to fit a model to our training data, which we used to predict on our test set. LASSO returned a model that was finalized with 17 X variables and returned an MSE of 2.35.

```
X_train = np.array(train_df.iloc[0:,1:])
X_test = np.array(test.iloc[0:,1:])
y_train = np.array(train_df['y'])
y_test = np.array(test['y'])

lasso = linear_model.LassoCV(cv=10).fit(X_train, y_train) #10 k-fold
y_predict = lasso.predict(X_test)
```

It is noted that there is a slight discrepancy with the coefficient selection when the LASSO model is normalized, with a total number of coefficients of 17 vs. 18. However, since the Test MSE did not improve with normalization, we chose to use the less complex model as the included LASSO model for this report.

Although the LASSO model picks the same coefficients to be non-zero as the MIQP, with an additional 7 variables picked, there is a difference shown in the values of the coefficients selected for LASSO compared to MIQP, due to LASSO utilizing a shrinkage method, where the coefficients face a penalization from lambda, producing smaller values for the coefficients for LASSO.

Advantages and Disadvantages:

- The MIQP process took about 2 hours to complete while the LASSO returned results almost instantly.

- MIQP returned a more accurate model with an MSE of 2.34, but it was only *slightly* better than the result from LASSO, which was 2.35.
- In this instance, LASSO is easier to implement than MIQP due to the pre-existing python packages that can be used to complete the cross-validation and modeling. MIQP implementation in the future would also require a re-evaluation of the k values we wanted to test, as well as a check to make sure our M value is large enough.

Final Recommendation:

The final recommendation would depend on our time constraints as well as whether the small difference in improvement is worth it. If an error of 1 was worth a billion dollars then it is probably worth spending more time for as much error reduction as we can get.

However, we must keep in mind that the difference in error is extremely small. It is very likely that changing the dataset or adding a bit of noise will change the results entirely. It is possible that MIQP did better than LASSO only by random chance, and that LASSO might outperform MIQP on a different dataset. But given our results, it does seem that MIQP does slightly better, and if we have the time to spare then MIQP would be the better choice if minimal error was the end goal. If we needed efficiency for whatever reason then LASSO would be a better choice.

If we wanted to conduct further experimentation then we could also try Ridge and ElasticNet. However Ridge does not zero out as quickly as Lasso, making feature selection a bit more cumbersome, while ElasticNet is a mix of both. Other options include more powerful ensemble models such as RandomForest or Boosting which both include feature importances to help us do feature selection.