Model Optimization Assisting Efficient COVID-19 Vaccine Distribution

Eva Wu & Roy Xiao

In [4]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import statsmodels.api as sm
import statsmodels.formula.api as smf

from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression, Lasso, Ridge
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import cross_val_score, GridSearchCV

import warnings
warnings.filterwarnings("ignore", category=FutureWarning)
```

In [5]:
```
# Helps to display all rows and columns in pandas
pd.set_option('display.max_columns', None)
pd.set_option('display.max_rows', None)
```

In [6]:
```
dfcov = pd.read_csv('Covid002.csv', encoding = 'latin-1')
print(dfcov.shape)
dfcov.head()
```

```
(3107, 93)
```

Out[6]:

	county	state	fips	cases	deaths	population	casespc	deathspc	cty	county_name	cty_pop2000	cz	cz
0	Autauga	Alabama	1001	4434	152	918492	482.74780	16.548864	1001	Autauga	43671	11101	Mont
1	Baldwin	Alabama	1003	10465	278	3102984	337.25601	8.959118	1003	Baldwin	140415	11001	
2	Barbour	Alabama	1005	3157	33	499262	632.33331	6.609756	1005	Barbour	29038	10301	
3	Bibb	Alabama	1007	2291	24	397470	576.39569	6.038192	1007	Bibb	20826	10801	Tus
4	Blount	Alabama	1009	2082	15	997531	208.71532	1.503713	1009	Blount	51024	10700	Birm

1. Filter & Clean

In [7]:
```
# pull out the column of variables' names into a separate list
dfdict = pd.read_csv('vardes.csv')
predlist = dfdict['Variable'].tolist()
```

In [106…
```
# then subset variables in our dataframe based on this list, removing 'casespc'
df = dfcov[np.intersect1d(dfcov.columns, predlist)]
df = df.drop(['casespc'], axis = 1)
df.head()
```

Out[106…

	adjmortmeas_amiall30day	adjmortmeas_chfall30day	bmcruderate	bmi_obese_q1	bmi_obese_q2	bmi_obese_q3	bmi_obes
0	0.146564	0.111778	859.29999	0.375000	0.238095	0.260870	0.13
1	0.145558	0.107229	976.20001	0.298050	0.262467	0.193237	0.13

	adjmortmeas_amiall30day	adjmortmeas_chfall30day	bmcruderate	bmi_obese_q1	bmi_obese_q2	bmi_obese_q3	bmi_obes
2	0.169922	0.107575	1040.90000	0.294118	0.571429	0.545455	0.27
3	0.234408	0.112190	1028.80000	0.466667	0.375000	0.190476	0.10
4	0.177953	0.117951	993.70001	0.347826	0.318182	0.529412	0.23

In [107…

```
# add the county & state for each observation + shifting outcome var 'deathspc' to front
directory = dfcov.iloc[:, 0:2] # extract county & state
df = pd.concat([directory, df], axis = 1)
column_to_move = df.pop('deathspc') # move deathspc to the front
df.insert(2, 'deathspc', column_to_move)
df.head()
```

Out[107…

	county	state	deathspc	adjmortmeas_amiall30day	adjmortmeas_chfall30day	bmcruderate	bmi_obese_q1	bmi_obese_
0	Autauga	Alabama	16.548864	0.146564	0.111778	859.29999	0.375000	0.238
1	Baldwin	Alabama	8.959118	0.145558	0.107229	976.20001	0.298050	0.262
2	Barbour	Alabama	6.609756	0.169922	0.107575	1040.90000	0.294118	0.571
3	Bibb	Alabama	6.038192	0.234408	0.112190	1028.80000	0.466667	0.375
4	Blount	Alabama	1.503713	0.177953	0.117951	993.70001	0.347826	0.318

Reference: extract list from column names, select columns by list, rearrange columns

2. Summary Statistics

In [108…

```
df.iloc[:, 2:].describe() # not include county and state
```

Out[108…

	deathspc	adjmortmeas_amiall30day	adjmortmeas_chfall30day	bmcruderate	bmi_obese_q1	bmi_obese_q2	bmi_ob
count	3107.000000	3106.000000	3107.000000	3107.00000	3107.000000	3107.000000	3107.(
mean	23.790131	0.165483	0.108969	1029.15597	0.239166	0.214580	0.
std	67.852145	0.039408	0.023565	248.38181	0.165928	0.153237	0.
min	0.000000	0.000000	0.000000	189.30000	0.000000	0.000000	0.(
25%	0.000000	0.145312	0.096301	864.29999	0.080128	0.000000	0.(
50%	3.802303	0.162727	0.107242	1036.30000	0.272076	0.241590	0.
75%	21.461759	0.183402	0.120155	1194.10000	0.335532	0.304348	0.:
max	2279.610600	0.444663	0.344451	1978.60000	1.000000	1.000000	1.(

3. Drop NA's

In [109…

```
df = df.dropna()
print(df.shape)
```

```
(2915, 63)
```

Note: should replace NA's with mean for more accurate analyses

4. Dummies for States

In [110…

```
df = pd.get_dummies(df, columns = ['state']) # can be run only once
```

In [111…

```
df.describe()
```

Out[111…

	deathspc	adjmortmeas_amiall30day	adjmortmeas_chfall30day	bmcruderate	bmi_obese_q1	bmi_obese_q2	bmi_ob

	deathspc	adjmortmeas_amiall30day	adjmortmeas_chfall30day	bmcruderate	bmi_obese_q1	bmi_obese_q2	bmi_ob
count	2915.000000	2915.000000	2915.000000	2915.000000	2915.000000	2915.000000	2915.(
mean	22.508358	0.166352	0.109098	1029.440137	0.250144	0.224401	0.
std	52.199827	0.033087	0.019315	241.287000	0.161168	0.149001	0.
min	0.000000	0.014564	0.013710	189.300000	0.000000	0.000000	0.(
25%	0.000000	0.146582	0.096893	870.600005	0.172700	0.150758	0.(
50%	4.520105	0.163299	0.107343	1040.000000	0.276798	0.247634	0.
75%	21.833407	0.183091	0.119816	1191.550000	0.340721	0.307692	0.
max	762.398250	0.338776	0.241361	1978.600000	1.000000	1.000000	1.(

5. Split Sample

In [112…
```
df = df.set_index(['county']) # set county as index
```

In [113…
```
X = df.drop(['deathspc'], axis = 1)
y = df[['deathspc']]

X_train, X_val, y_train, y_val = train_test_split(X, y, test_size = 0.2, random_state = 25)
```

6. OLS

a) MSE in training & validation sets

In [114…
```
ols = LinearRegression(fit_intercept = True)
ols.fit(X_train, y_train)

y_pred_training = ols.predict(X_train)
print('MSE of OLS model fitted onto training set: ', mean_squared_error(y_train, y_pred_training))

y_pred_val = ols.predict(X_val)
print('MSE of OLS model fitted onto validation set: ', mean_squared_error(y_val, y_pred_val))
```

```
MSE of OLS model fitted onto training set:  1561.857307690756
MSE of OLS model fitted onto validation set:  1927.2484718467585
```

b) Evidence of overfitting?

Under our original split of the dataset (with random_state = 25), we find that the OLS model trained on our training set has a **higher validation-set mean squared error (MSE) than training-set MSE**. This is usually a sign that our predictive model is overfitting onto the data in the training set. As we include many predictors in our model, the model will fit flexibly on the data we have in the training set and this reduces model bias. However, greater model flexibility simultaneously results in greater training variance, resulting in a bias-variance trade-off that could increase MSE when the model is fitted onto the validation set. In this case, we might infer that the model we specified is overly flexible and thus suffers from overfitting when fitted onto the validation set.

The Validation Set approach suffers from high variability when we use it to estimate the test error of our model, because the validation estimate of the test error changes based on how we split the data.

7. Model Regularisation - Ridge & Lasso

Ridge Regression:

a) Estimate the test error of 100 Ridge Regression models with different tuning parameter values (ranging from 0.01 to 100) using 10-fold Cross Validation:

In [115…

```python
ridge = Ridge(normalize = True) # normalize is essential in RR

# Defining set of regularization parameters - alpha
# (aka the tuning parameter lambda in the Ridge Regression equation)
# Note that we must use 'alpha' to term the reg param, specifically coded as such in Ridge & Lasso
# Taking 10 to the power of the set of numbers from -2 to 2 (with 100 intervals)
alpha_param = (10**np.linspace(start = -2, stop = 2, num = 100))
```

In [116…

```python
# train Ridge rergession using multiple values of alpha from the list of params defined above
# calculate a vector of mean and standard deviation values for each parameter
# (MSE of RR model with some alpha)

def vector_values(grid_search, trials):
    mean_vec = np.zeros(trials) # an array w/ 'trials' # of 0s
    std_vec = np.zeros(trials)
    i = 0
    final = grid_search.cv_results_

    # Using Grid Search's 'cv_results' attribute to get mean and std for each parameter
    for mean_score, std_score in zip(final["mean_test_score"], final["std_test_score"]):
        mean_vec[i] = -mean_score # negative sign used to get positive MSE
        std_vec[i] = std_score
        i = i + 1

    return mean_vec, std_vec
```

In [117…

```python
# Creating a parameters grid
param_grid = [{'alpha': alpha_param }]

# Running Grid Search over the alpha (regularization) parameter,
# to obtain the estimated test MSE (10-fold CV error) of each RR model w/ different lambda
grid_search_ridge = GridSearchCV(ridge, param_grid, cv = 10, scoring = 'neg_mean_squared_error')
grid_search_ridge.fit(X_train, y_train)
```

Out[117…

```
GridSearchCV(cv=10, estimator=Ridge(normalize=True),
             param_grid=[{'alpha': array([1.00000000e-02, 1.09749877e-02, 1.20450354e-02, 1.32194115e-0
2,
       1.45082878e-02, 1.59228279e-02, 1.74752840e-02, 1.91791026e-02,
       2.10490414e-02, 2.31012970e-02, 2.53536449e-02, 2.78255940e-02,
       3.05385551e-02, 3.35160265e-02, 3.67837977e-02, 4.03701726e-02,
       4.43062146e-02, 4.86260158e-02, 5...
       1.17681195e+01, 1.29154967e+01, 1.41747416e+01, 1.55567614e+01,
       1.70735265e+01, 1.87381742e+01, 2.05651231e+01, 2.25701972e+01,
       2.47707636e+01, 2.71858824e+01, 2.98364724e+01, 3.27454916e+01,
       3.59381366e+01, 3.94420606e+01, 4.32876128e+01, 4.75081016e+01,
       5.21400829e+01, 5.72236766e+01, 6.28029144e+01, 6.89261210e+01,
       7.56463328e+01, 8.30217568e+01, 9.11162756e+01, 1.00000000e+02])}],
             scoring='neg_mean_squared_error')
```

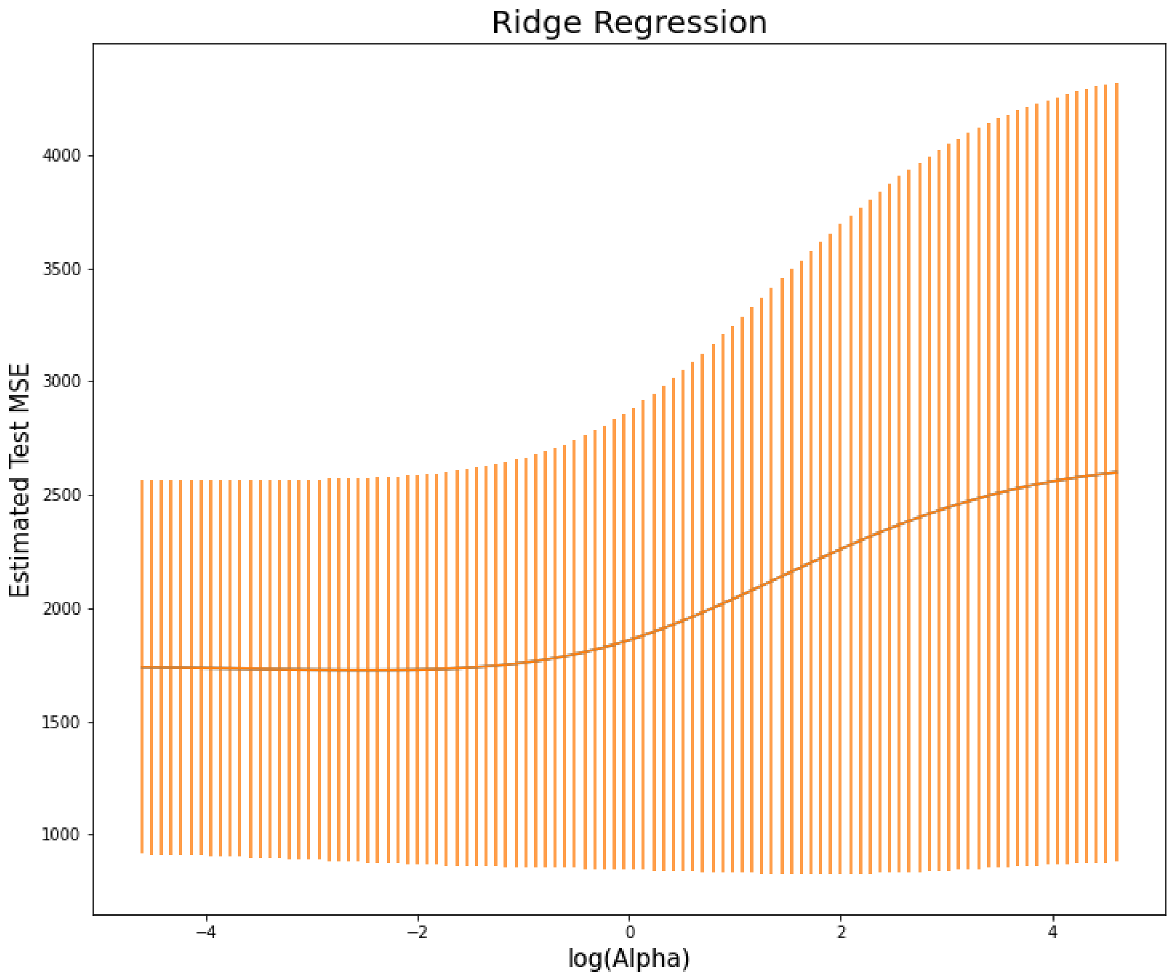
b) Plot 10FCV estimates of test error as a function of lambda value

In [118…

```python
# Applying the vector_values function we created to calculate mean and std dev
# of the estimated test MSE for each RR model
mean_vec, std_vec = vector_values(grid_search_ridge, 100)

plt.figure(figsize = (12, 10))
plt.title('Ridge Regression', fontsize = 20)
plt.plot(np.log(alpha_param), mean_vec) # base e
plt.errorbar(np.log(alpha_param), mean_vec, yerr = std_vec)
plt.ylabel('Estimated Test MSE', fontsize = 15)
plt.xlabel('log(Alpha)', fontsize = 15)
plt.show()

# Plot y versus x as lines and/or markers with attached errorbars.
# x, y define the data locations, xerr, yerr define the errorbar sizes.
# By default, this draws the data markers/lines as well the errorbars.
# Use fmt='none' to draw errorbars without any data markers.
```



c) Choose optimal lambda

In [119…

```
# Find the optimal MSE score --> lowest MSE
print('Minimal 10-fold CV error rate (estimated test error): ', min(mean_vec))

# Optimal alpha --> one that minimizes MSE
print('Optimal tuning parameter value: ',
      alpha_param[np.where(mean_vec == min(mean_vec))][0])
```

```
Minimal 10-fold CV error rate (estimated test error):  1725.783200975835
Optimal tuning parameter value:  0.08497534359086446
```

d) Re-estimate using optimal lambda

In [120…

```
ridge_optimal = Ridge(alpha = alpha_param[np.where(mean_vec == min(mean_vec))][0],
                      fit_intercept = True, normalize = True)
ridge_optimal.fit(X_train, y_train)
ridge_optimal.coef_
```

Out[120…

```
array([[-5.52354362e+00,  7.26861602e+00,  2.62173247e-03,
        -3.04356240e+00,  4.20809248e+00, -1.59563057e+01,
         2.15227316e+00, -6.34129038e-01,  1.26844000e+00,
         1.40665081e+01,  3.50779400e+01, -1.32973055e+02,
         1.08527605e+00,  3.37697609e+01,  1.88687564e+01,
         5.76526142e-01, -5.61030407e-02, -3.89741402e+01,
         1.60680618e+01, -8.03701310e-01,  1.00090006e+00,
         3.71427283e+00,  8.62611112e+00,  1.89752783e-01,
        -5.30680721e-01,  1.27104575e-02, -1.45339124e+00,
         1.52811314e+01, -4.60849519e+00, -6.50694873e+00,
        -7.11976793e+01, -1.44208068e+01, -1.02675444e+01,
         6.41061566e-04, -1.70387149e+00,  2.66239208e+00,
        -9.92555781e-03,  1.75880319e+00,  5.35484303e-05,
        -1.17885548e+02, -1.91552648e+02,  7.41837676e-01,
         3.80606207e-01, -5.20449990e+00,  3.10601340e+00,
         6.89749505e-03,  1.16503612e-03, -7.85960359e-01,
```

```
-5.74449463e-04,  1.15287491e-01, -3.47726764e+00,
 9.54050176e-02,  3.77617428e-04, -1.80050150e-01,
 8.34669387e-01, -3.55785644e-01, -3.15282354e+01,
-1.59833372e+02, -4.89047207e-01,  3.19763259e-03,
-1.66196870e+01, -2.53181638e+01, -1.94287394e+01,
-3.61925400e+01,  1.32420958e+01,  8.06623418e+01,
 3.73280929e+01, -1.56360488e+01,  1.85371851e+01,
 8.26819702e+00,  1.77676659e+00,  2.76076115e+01,
 4.42672283e+00, -3.21382736e+00, -7.74626234e-01,
 4.86443480e+01, -2.21866884e-01, -8.00437217e+00,
 7.95197672e+01,  1.99849237e+01,  5.24158049e-01,
 2.60687591e+00, -4.34329068e+00,  1.34023112e+01,
 4.58765549e+00, -1.14335514e+01, -5.07532961e+00,
-1.93950737e+01,  3.27915864e+01, -2.05292457e+01,
 7.60433056e+00,  7.53185557e+00, -6.68824240e-01,
 2.62902340e+00,  7.80507909e+00, -1.00401712e+01,
-2.49782806e+01,  7.33442805e+00, -1.36975294e+01,
-1.37288091e+01, -8.35166854e+00, -6.54677720e+00,
-2.19883391e+01,  1.05686222e+01, -4.50470183e+00,
-4.38173855e+00, -5.42707117e+00]])
```

LASSO method:

a) Estimate the test error of 100 Lasso models with different tuning parameter values (ranging from 0.01 to 100) using 10-fold Cross Validation:

In [121…

```
lasso = Lasso(normalize = True)

# Defining set of regularization parameters - alpha
# (aka the tuning parameter lambda in the Lasso equation)
# Note that we must use 'alpha' to term the reg param, specifically coded as such in Ridge & Lasso
# Taking 10 to the power of the set of numbers from -3 to 1 (with 100 intervals)
alpha_param = (10**np.linspace(start = -3, stop = 1, num = 100))
# shift range in order to find optimal value
```

In [122…

```
# Creating a parameters grid
param_grid = [{'alpha': alpha_param }]

# Running Grid Search over the alpha (regularization) parameter
grid_search_lasso = GridSearchCV(lasso, param_grid, cv = 10, scoring = 'neg_mean_squared_error')
grid_search_lasso.fit(X_train, y_train)
```

Out[122…

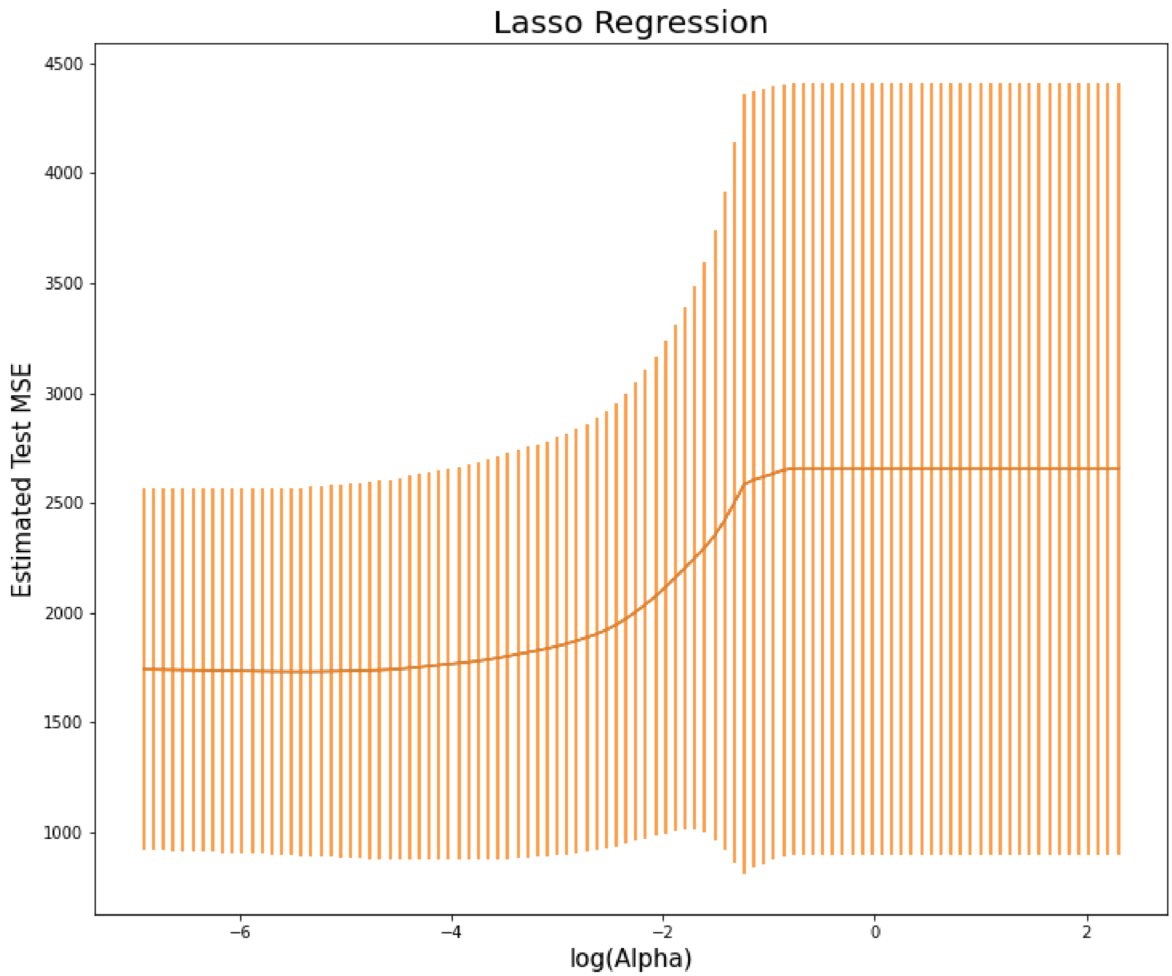
```
GridSearchCV(cv=10, estimator=Lasso(normalize=True),
             param_grid=[{'alpha': array([1.00000000e-03, 1.09749877e-03, 1.20450354e-03, 1.32194115e-0
3,
       1.45082878e-03, 1.59228279e-03, 1.74752840e-03, 1.91791026e-03,
       2.10490414e-03, 2.31012970e-03, 2.53536449e-03, 2.78255940e-03,
       3.05385551e-03, 3.35160265e-03, 3.67837977e-03, 4.03701726e-03,
       4.43062146e-03, 4.86260158e-03, 5...
       1.17681195e+00, 1.29154967e+00, 1.41747416e+00, 1.55567614e+00,
       1.70735265e+00, 1.87381742e+00, 2.05651231e+00, 2.25701972e+00,
       2.47707636e+00, 2.71858824e+00, 2.98364724e+00, 3.27454916e+00,
       3.59381366e+00, 3.94420606e+00, 4.32876128e+00, 4.75081016e+00,
       5.21400829e+00, 5.72236766e+00, 6.28029144e+00, 6.89261210e+00,
       7.56463328e+00, 8.30217568e+00, 9.11162756e+00, 1.00000000e+01])}],
             scoring='neg_mean_squared_error')
```

b) Plot 10FCV estimates of test error as a function of lambda value

In [123…

```
# Applying the vector_values function we created to calculate mean and
# std dev of the estimated test MSE for each Lasso model
mean_vec, std_vec = vector_values(grid_search_lasso, 100)

plt.figure(figsize = (12,10))
plt.title('Lasso Regression', fontsize = 20)
plt.plot(np.log(alpha_param), mean_vec)
plt.errorbar(np.log(alpha_param), mean_vec, yerr = std_vec)
plt.ylabel("Estimated Test MSE", fontsize = 15)
plt.xlabel("log(Alpha)", fontsize = 15)
plt.show()
```



c) Choose optimal lambda

In [124…

```
# Find the optimal MSE score --> lowest MSE
print('Minimal 10-fold CV error rate (estimated test error): ', min(mean_vec))

# Optimal alpha --> one that minimizes MSE
print('Optimal tuning parameter value: ',
      alpha_param[np.where(mean_vec == min(mean_vec))][0])
```

```
Minimal 10-fold CV error rate (estimated test error):  1729.3554197235583
Optimal tuning parameter value:  0.004430621457583882
```

d) Re-estimate using optimal lambda

In [125…

```
lasso_optimal = Lasso(alpha = alpha_param[np.where(mean_vec == min(mean_vec))][0],
                      normalize = True)
lasso_optimal.fit(X_train, y_train)
lasso_optimal.coef_
```

Out[125…

```
array([-0.00000000e+00,  2.07121656e+01,  1.57363718e-04, -1.10776948e+00,
        0.00000000e+00, -1.59535871e+01,  0.00000000e+00,  0.00000000e+00,
        9.81229515e-01,  0.00000000e+00,  1.61812565e+01, -1.21744198e+02,
        1.13329555e+00,  3.79219144e+01, -0.00000000e+00,  7.59247931e-01,
       -0.00000000e+00, -3.96932413e+01,  1.35815726e+01, -0.00000000e+00,
        0.00000000e+00,  1.80921365e+00,  6.68904182e+00,  1.14367494e-01,
       -5.69209377e-01,  0.00000000e+00, -0.00000000e+00,  1.94738643e+01,
       -4.23371311e+00, -7.14882754e+00, -8.22227305e+01, -8.90075800e+00,
       -1.28564019e+01,  9.31514029e-04, -0.00000000e+00,  2.41018966e+00,
       -0.00000000e+00,  2.04368967e+00,  6.90649328e-06, -9.01791008e+01,
       -2.38116872e+02,  3.71947961e-01,  0.00000000e+00, -1.41162840e+00,
       -0.00000000e+00,  8.10725651e-03,  0.00000000e+00, -8.53818080e-01,
       -4.54981259e-04,  9.27204448e-02, -3.84252475e+00,  1.55624135e-01,
        0.00000000e+00, -1.42611075e-01,  7.13457749e-01, -0.00000000e+00,
       -5.14566698e+00, -1.77008498e+02, -4.12298100e-01, -0.00000000e+00,
       -1.68041292e+01, -2.07300055e+01, -2.02327295e+01, -3.40476492e+01,
```

```
 1.33505495e+01,  8.45495261e+01,  3.32260695e+01, -1.60934332e+01,
 1.75433474e+01,  6.98720538e+00,  9.63524018e-01,  2.97102288e+01,
 2.58218517e+00, -2.75980899e+00,  6.29052842e-01,  4.90309775e+01,
-0.00000000e+00, -6.66714579e+00,  8.68632956e+01,  1.90832714e+01,
-0.00000000e+00,  0.00000000e+00, -2.16322070e+00,  1.30050963e+01,
 1.52022203e+00, -6.25280041e+00, -2.92191089e+00, -1.79593665e+01,
 3.30020478e+01, -2.14711076e+01,  3.95369832e+00,  6.97931980e+00,
 0.00000000e+00,  1.69100643e+00,  8.35996449e+00, -7.33765862e+00,
-2.70745331e+01,  3.45005884e+00, -1.26270047e+01, -1.56614253e+01,
-4.32426372e+00, -6.35483589e+00, -2.44806231e+01,  1.15725346e+01,
-1.33845272e+00, -3.65468958e+00, -4.75318025e+00])
```

8. Evaluation

In [126…

```
y_pred_training = ols.predict(X_train)
print('MSE of OLS model fitted onto training set: ',
      mean_squared_error(y_train, y_pred_training))

y_pred_val = ols.predict(X_val)
print('MSE of OLS model fitted onto validation set: ',
      mean_squared_error(y_val, y_pred_val))

y_pred_training_ridge = ridge_optimal.predict(X_train)
print('MSE of RR model fitted onto training set: ',
      mean_squared_error(y_train, y_pred_training_ridge))

y_pred_val_ridge = ridge_optimal.predict(X_val)
print('MSE of RR model fitted onto validation set: ',
      mean_squared_error(y_val, y_pred_val_ridge))

y_pred_training_lasso = lasso_optimal.predict(X_train)
print('MSE of LASSO model fitted onto training set: ',
      mean_squared_error(y_train, y_pred_training_lasso))

y_pred_val_lasso = lasso_optimal.predict(X_val)
print('MSE of LASSO model fitted onto validation set: ',
      mean_squared_error(y_val, y_pred_val_lasso))
```

```
MSE of OLS model fitted onto training set:  1561.857307690756
MSE of OLS model fitted onto validation set:  1927.2484718467585
MSE of RR model fitted onto training set:  1584.5061729130412
MSE of RR model fitted onto validation set:  1919.5396540261922
MSE of LASSO model fitted onto training set:  1579.746859264915
MSE of LASSO model fitted onto validation set:  1880.376121119161
```

The validation-set error (in this case, the MSE of the model fitted onto data in the validation set) gives us an estimate of the true test error of the model, which is a measure of the predictive accuracy of a model.

From our results above, we see that the Lasso model (optimally tuned using 10-fold Cross Validation) suffers the lowest validation-set error (1880.38). The Ridge Regression model (also optimally tuned using 10-fold Cross Validation) suffers a higher validation-set error (1919.54) than the Lasso model but lower than the OLS model. The OLS model has the highest validation-set error (1927.25). This implies that the Ridge Regression and the Lasso method of model regularization both improved the predictive accuracy of our model, but **the LASSO model performed the best**, and should thus be recommended to the CDC for predicting Covid-19 deaths per capita at the county-level.