Homework 3: Linear Model Selection and Regularization

Chia-Yun Chang

Conceptional Exercises

In [42]:
```
import random
import numpy as np
import pandas as pd
import sklearn.model_selection
from sklearn.model_selection import train_test_split
from tabulate import tabulate
import math
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error as mse
from IPython.core.display import display, HTML
display(HTML("<style>div.output_scroll { height: 38em; }</style>"))
```

1. Generate and split data

In [38]:
```
random.seed(5566)

#ds = pd.DataFrame()
l = []

for i in range(20):

    #generate feature by feature so that each feature can have consistant n
    x = np.random.uniform(random.random(),random.random(),1000)
    X = pd.DataFrame(x)
    l.append(X)

#Note: or, ds = pd.concat(l, axis=1)
X = np.column_stack(l)

#Generate a set of betas
beta = np.random.uniform(-100,100,20)
#Choose random betas to be 0
beta[4] =0
beta[7] =0
beta[14] = 0
beta[15] = 0
beta.shape = (20,)

Y = np.dot(X, beta) + np.random.uniform(-1,1,1000)

X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.1)
Y_train.shape = (900,1)
Y_test.shape = (100,1)

#Check if splited correctly
print (X_train.shape, X_test.shape, Y_train.shape, Y_test.shape)
```

```
(900, 20) (100, 20) (900, 1) (100, 1)
```

2. Find the best subset for each size

Here I use forward selection method for feature selection.

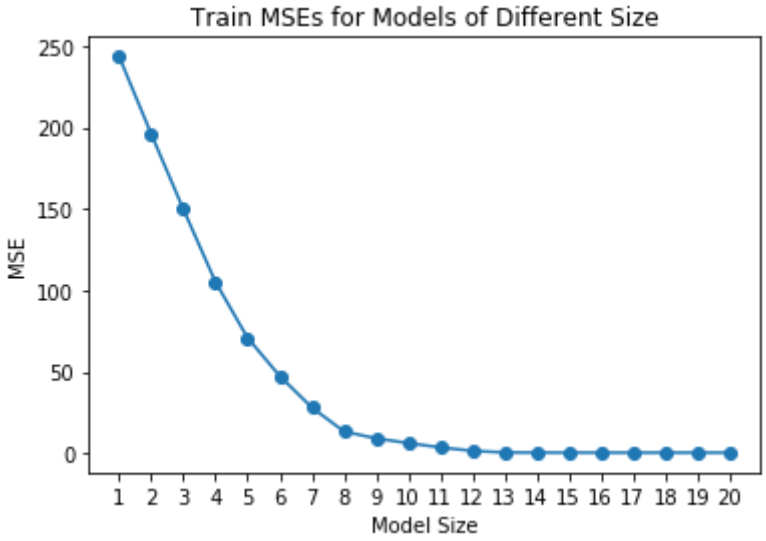
In [16]:
```
def find_best_subset(Y_train, X_train, l_var):
    P = []
    min_mse = 99999999999999
    for i in range(20):
        if i not in l_var:
            m = LinearRegression()
            m.fit(X_train[:,l_var+[i]],Y_train)
            least_mse = mse(Y_train, m.predict(X_train[:,l_var+[i]]))
            if least_mse < min_mse:
                min_mse = least_mse
                p = l_var + [i]
                best_mod = m


    #return(p, min_mse, test_err)
    return(p, min_mse, best_mod)
```

In [18]:
```
trains = []
l_var = []
models = []
for i in range(20):
    model = find_best_subset(Y_train, X_train, l_var)
    l_var = model[0]
    trains.append([l_var, model[1]])
    models.append(model[2])

print(tabulate(trains, headers = ['Predictors','train MSE']) )
```

```
Predictors
train MSE
-------------------------------------------------------------------------  -
----------
[5]
243.57
[5, 13]
195.976
[5, 13, 9]
149.808
[5, 13, 9, 17]
105.177
[5, 13, 9, 17, 3]
70.8182
[5, 13, 9, 17, 3, 12]
47.3804
[5, 13, 9, 17, 3, 12, 1]
28.0309
[5, 13, 9, 17, 3, 12, 1, 16]
13.3426
[5, 13, 9, 17, 3, 12, 1, 16, 2]
9.0769
[5, 13, 9, 17, 3, 12, 1, 16, 2, 0]
6.20394
[5, 13, 9, 17, 3, 12, 1, 16, 2, 0, 10]
3.52664
[5, 13, 9, 17, 3, 12, 1, 16, 2, 0, 10, 6]
1.44588
[5, 13, 9, 17, 3, 12, 1, 16, 2, 0, 10, 6, 19]
0.435834
[5, 13, 9, 17, 3, 12, 1, 16, 2, 0, 10, 6, 19, 18]
0.360868
[5, 13, 9, 17, 3, 12, 1, 16, 2, 0, 10, 6, 19, 18, 8]
0.331235
[5, 13, 9, 17, 3, 12, 1, 16, 2, 0, 10, 6, 19, 18, 8, 11]
0.32668
[5, 13, 9, 17, 3, 12, 1, 16, 2, 0, 10, 6, 19, 18, 8, 11, 7]
0.3256
[5, 13, 9, 17, 3, 12, 1, 16, 2, 0, 10, 6, 19, 18, 8, 11, 7, 4]
0.325339
[5, 13, 9, 17, 3, 12, 1, 16, 2, 0, 10, 6, 19, 18, 8, 11, 7, 4, 14]
0.325104
[5, 13, 9, 17, 3, 12, 1, 16, 2, 0, 10, 6, 19, 18, 8, 11, 7, 4, 14, 15]
0.325098
```

3. Plot the train MSE against model size

In [19]:
```
x_v = list(range(1,21))
y_v = [trains[i][1] for i in range(20)]

plt.plot(x_v,y_v, marker = 'o')
plt.xticks(x_v)
plt.xlabel('Model Size')
plt.ylabel('MSE')
plt.title('Train MSEs for Models of Different Size');
```

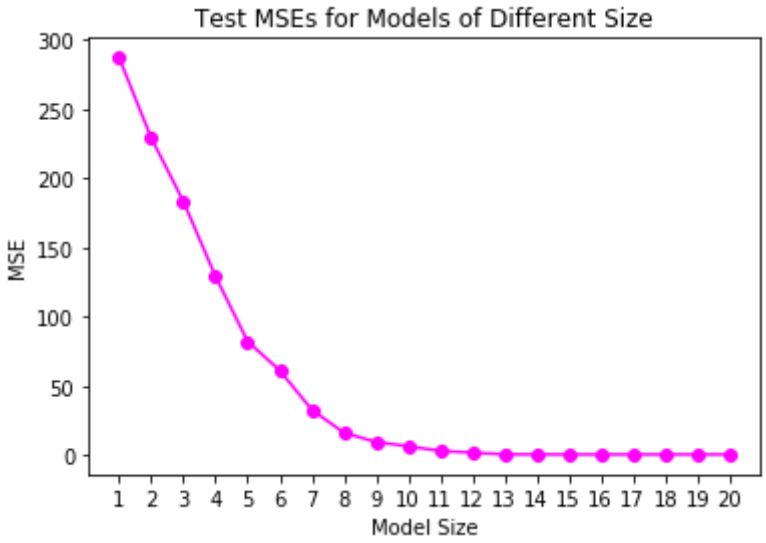


4. Plot the test MSE against model size

In [44]:
```
tests = []
j = 0
for i in range(len(models)):
    test_err = mse(Y_test, models[i].predict(X_test[:, trains[i][0]]))
    tests.append([trains[i][0],test_err])


print(tabulate(tests, headers = ['Predictors','test MSE']) )
```

```
Predictors
test MSE
-------------------------------------------------------------------------  -
---------
[5]
2418.31
[5, 13]
2532.2
[5, 13, 9]
2511.69
[5, 13, 9, 17]
2673.1
[5, 13, 9, 17, 3]
2674.99
[5, 13, 9, 17, 3, 12]
2819.66
[5, 13, 9, 17, 3, 12, 1]
2903.66
[5, 13, 9, 17, 3, 12, 1, 16]
2899.15
[5, 13, 9, 17, 3, 12, 1, 16, 2]
2904.91
[5, 13, 9, 17, 3, 12, 1, 16, 2, 0]
2881.22
[5, 13, 9, 17, 3, 12, 1, 16, 2, 0, 10]
2881.71
[5, 13, 9, 17, 3, 12, 1, 16, 2, 0, 10, 6]
2880.77
[5, 13, 9, 17, 3, 12, 1, 16, 2, 0, 10, 6, 19]
2888.47
[5, 13, 9, 17, 3, 12, 1, 16, 2, 0, 10, 6, 19, 18]
```

In [35]:
```
x_v = list(range(1,21))
y_v = [tests[i][1] for i in range(20)]

plt.plot(x_v,y_v, marker = 'o', color = 'magenta')
plt.xticks(x_v)
plt.xlabel('Model Size')
plt.ylabel('MSE')
plt.title('Test MSEs for Models of Different Size');
```



5. Which size model renders the least MSE?

In my simulation, the best model that renders the least MSE is size 17, with 17 features. This makes sense because out of the 20 features, I made 4 of them 0, which means there are 4 features that does not play a part in Y, i.e. have nothing to do with how Y is the way it is. If we take one step further, we can see that the features are [5, 13, 9, 17, 3, 12, 1, 16, 2, 0, 10, 6, 19, 18, 8, 11, 7], which almost excludes all the 0 beta predictors, [4,7,14,15]. This model however, includes predictor 7. This is becuase there was an error term introduced in the data generation process. The beta fitted for predictor 7 is 1.91, not far from the designated 0.

6. Best model compare to the true generation model.

As shown below, the coefficients are very close.

In [22]:
```
beta
```

Out[22]:
```
array([ -9.4878404 , -24.76726418, -63.2961691 , -93.88724159,
          0.        ,  81.23528306,  -8.32026825,   0.        ,
         21.98178747,  51.82758802,  17.76018042,  95.98491333,
        -24.30412571, -69.07906522,   0.        ,   0.        ,
         53.23924362, -93.4408831 , -30.60099162,  44.70869063])
```

In [25]:
```
# sort orignal beta according to forward selection
lp = [5, 13, 9, 17, 3, 12, 1, 16, 2, 0, 10, 6, 19, 18, 8, 11, 7, 4, 14, 15]
beta_sorted = []
for p in lp:
    beta_sorted.append([p, beta[p]])
```

In [26]:
```
# the original betas
nbeta = []
for b in beta_sorted:
    nbeta.append(b[1])
    print("predictor", b[0] ,'  ', b[1])
```

```
predictor 5    81.23528305764077
predictor 13    -69.07906521558074
predictor 9    51.827588024660685
predictor 17    -93.44088310498402
predictor 3    -93.88724158714584
predictor 12    -24.3041257123149
predictor 1    -24.767264177685377
predictor 16    53.239243623285546
predictor 2    -63.2961691004476
predictor 0    -9.487840400901888
predictor 10    17.760180423530116
predictor 6    -8.320268253780611
predictor 19    44.70869062993813
predictor 18    -30.600991624094775
predictor 8    21.981787469542425
predictor 11    95.9849133283746
predictor 7    0.0
predictor 4    0.0
predictor 14    0.0
predictor 15    0.0
```

In [27]:
```
# the learned model
preds = models[16].coef_[0]
ip = [5, 13, 9, 17, 3, 12, 1, 16, 2, 0, 10, 6, 19, 18, 8, 11, 7]
for p in range(len(preds)):
    print ('predictor', ip[p], '  ',preds[p])
```

```
predictor 5    80.87120116483295
predictor 13    -69.03642416146292
predictor 9    52.02456737406512
predictor 17    -93.66731009725926
predictor 3    -94.08820884560971
predictor 12    -24.26168353618322
predictor 1    -24.67767417235813
predictor 16    53.86005186593691
predictor 2    -62.80027975513988
predictor 0    -9.477589281229669
predictor 10    17.978871691365963
predictor 6    -8.526510662358561
predictor 19    44.740897012063705
predictor 18    -29.532416955371094
predictor 8    21.630649936989364
predictor 11    101.73093377364258
predictor 7    1.917668496408793
```

7. Root of the Squared Sum between generation coeficients and model coeficients

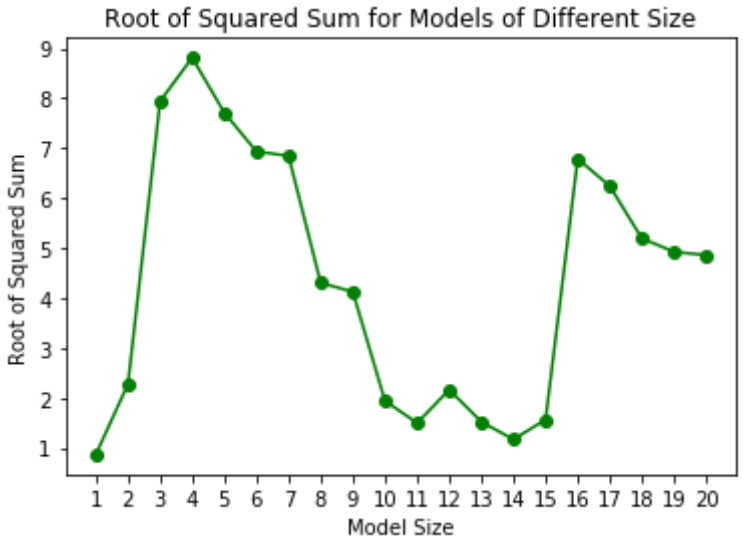
In [28]:
```
c = 2
cy = (nbeta[:3]- models[2].coef_)**2
sum(cy[0])
```

Out[28]:
```
63.053741924540326
```

In [45]:
```
l_s = []
for c in range(20):
    sq = (nbeta[:c+1]- models[c].coef_)**2
    l_s.append([c+1, sum(sq[0])**0.5])

print(tabulate(l_s, headers = ['Model Size','Root of Squared Sum']) )
```

Model Size	Root of Squared Sum
1	0.86494
2	2.27459
3	7.94064
4	8.81241
5	7.71192
6	6.9345
7	6.84589
8	4.30232
9	4.12467
10	1.94084
11	1.49416
12	2.16142
13	1.51936
14	1.16562
15	1.56356
16	6.78717
17	6.24154
18	5.19113
19	4.92427
20	4.85761

In [30]:
```
x7 = list(range(1,21))
y7 = [l_s[i][1] for i in range(20)]

plt.plot(x7,y7, marker = 'o', color = 'green')
plt.xticks(x7)
plt.xlabel('Model Size')
plt.ylabel('Root of Squared Sum')
plt.title('Root of Squared Sum for Models of Different Size');
```

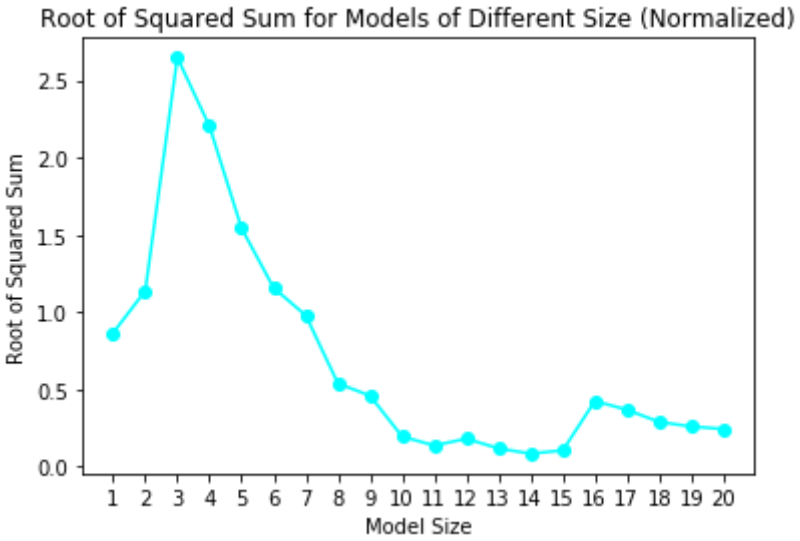


The root of squared sum peaked shoots up radically when irrelevant features are introduced, consistant with when the model starts to overfit. The minimum occured at the sized 11 model.

I think this graph would be more meaningful if the squared roots are divided by the size of the model to normalize becuase essentially, more predictors means more variances to be added according to the numerical function given. The normalized graph is demonstrated below.

If we want to see a trend closer to the MSE, we would have to treat the learned model not as selected variable models, but full models (with unused predictor beta set to 0)to compare to the original generation model.

In [33]:
```
l_s = []
for c in range(20):
    sq = (nbeta[:c+1]- models[c].coef_)**2
    l_s.append([c+1, sum(sq[0])**0.5/(c+1)])

x7 = list(range(1,21))
y7 = [l_s[i][1] for i in range(20)]

plt.plot(x7,y7, marker = 'o', color = 'cyan')
plt.xticks(x7)
plt.xlabel('Model Size')
plt.ylabel('Root of Squared Sum')
plt.title('Root of Squared Sum for Models of Different Size (Normalized)');
```



Application Exercises

In [1]:
```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from tabulate import tabulate
import math
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.linear_model import LinearRegression
from sklearn.linear_model import RidgeCV
from sklearn.linear_model import LassoCV
from sklearn.linear_model import ElasticNetCV
from sklearn.metrics import mean_squared_error as mse
```

0. Import data

In [2]:
```
train = pd.read_csv('gss_train.csv')
test = pd.read_csv('gss_test.csv')

x_train = train.drop('egalit_scale', axis =1)
x_test = test.drop('egalit_scale', axis =1)
y_train = train.egalit_scale
y_test = test.egalit_scale
```

1. Least Square Linear

In [3]:
```
lr = LinearRegression().fit(x_train, y_train)
test_mse = mse(lr.predict(x_test),y_test)
print('Linear regression test error:', test_mse)
```

```
Linear regression test error: 63.213629623014995
```

2. Ridge Regression

In [4]:
```
r = RidgeCV(cv = 10, alphas = [1e-3, 1e-2, 1e-1, 1, 10, 100, 1000]).fit(x_t
test_mse = mse(r.predict(x_test),y_test)
print('Ridge regression test error:', test_mse)
```

```
Ridge regression test error: 62.256410822269125
```

3. Lasso Regression

In [5]:
```
l = LassoCV(cv = 10, alphas = [1e-3, 1e-2, 1e-1, 1, 10, 100, 1000]).fit(x_t
test_mse = mse(l.predict(x_test),y_test)
print('Lasso regression test error:', test_mse)
```

```
Lasso regression test error: 62.7784155547739
```

In [6]:
```
s= 0
for c in l.coef_:
    if c == 0 or c == -0:
        s+=1
print ('Number of non-zero coefficient:', s)
```

```
Number of non-zero coefficient: 53
```

4. Elastic Net

In [9]:
```
alph = [x*0.1 for x in range(1, 11)]
e = ElasticNetCV(cv = 10, l1_ratio = alph).fit(x_train, y_train)
test_mse = mse(e.predict(x_test),y_test)
print('Elastic Net test error:', test_mse)
```

```
Elastic Net test error: 62.7780157899344
```

In [10]:
```
s= 0
for c in e.coef_:
    if c == 0 or c == -0:
        s+=1
print ('Number of non-zero coefficient:', s)
```

```
Number of non-zero coefficient: 53
```

5. Comment

The simple linear regression has the highest test error. This makes sense because linear regression did not perform any tuning. The model could be too sensitive/ reacting too much to parameters that have nothing to do with predicting egalitarian. Ridge performs slightly better than Elastic Net, and Elastic Net than Lass. Ridge does not tune coefficient to 0, whereas both Lasso and Elastic net has 53 coefficient equals to 0. Since Elastic Net takes into account both the penalty terms of Ridge and Lasso, it makes sense that its the 2nd best among the three. This could hint that the dataset we are working with does not contain really unnecessary parameters, and since the parameters could at some point be correlated, Lasso performs slightly worse than Elastic Net. This being said, the 3 models (excluding linear regression) has very similiar performance.