Perspective HW3

Yuxin Wu (yuxinwu@uchicago.edu)

February 9th, 2020

In [1]:

```
import numpy as np
from numpy import random
from sklearn.model_selection import train_test_split
import pandas as pd
import matplotlib.pyplot as plt
from sklearn import linear_model
from sklearn.metrics import mean_squared_error
from sklearn.linear_model import LinearRegression
from sklearn.linear_model import Ridge
from sklearn.linear_model import Lasso
from sklearn.linear_model import ElasticNet
from sklearn.linear_model import RidgeCV
from sklearn.linear_model import LassoCV
from sklearn.model_selection import GridSearchCV
from sklearn.linear_model import ElasticNetCV
```

Conceptual exercises

In [2]:

```
random.seed(555)
```

In [3]:

```
x = np.random.randn(1000,20)
```

In [4]:

```
beta = np.random.randint(5) * np.random.randn(20) + np.random.randint(5)
```

In [5]:

```
zero = np.random.randint(1,20, 5)
```

In [6]:

```
for i in zero:
    beta[i] = 0
print(beta)
```

```
[7.06705688 3.32537165 0.         5.06372457 0.7208984  5.53275629
 6.00271004 0.         4.21832637 2.80775456 0.12656218 5.97784656
 3.22190012 0.74818864 5.0528959  0.         4.66012524 4.20046414
 0.         8.03453625]
```

In [7]:

```
beta0 = beta.copy()
```

In [8]:

```
err = np.random.normal(0,1,1000)
```

In [9]:

```
y = np.dot(x, beta) + err
```

In [10]:

```
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.9)
```

In [11]:

```
x_train = pd.DataFrame(x_train)
y_train = pd.DataFrame(y_train)
x_test = pd.DataFrame(x_test)
y_test = pd.DataFrame(y_test)
```

In [12]:

```
def fit_linear_reg(X, Y, X_test, Y_test):
    model_k = linear_model.LinearRegression()
    model_k.fit(X,Y)
    MSE = mean_squared_error(Y,model_k.predict(X))
    MSE_test = mean_squared_error(Y_test,model_k.predict(X_test))
    return model_k, MSE, MSE_test
```

In [13]:

```
combo = []
MSE_list = {}
feature = []
beta_col = []

for m in range(0, 20):
    for i in range(0, 20):

        if i not in combo:
            combo.append(i)
            coef, tmp_result, test_result = fit_linear_reg(x_train[combo], y_train, x_test[combo],
y_test)

            if len(combo) not in MSE_list:
                MSE_list[len(combo)] = [tmp_result, test_result]
                feature.append(combo[:])
                combo.pop()
                beta_col.append(coef.coef_)

            else:
                if tmp_result < MSE_list[len(combo)][0]:
                    MSE_list[len(combo)] = [tmp_result, test_result]
                    beta_col[-1] = coef.coef_
                    feature[-1] = combo[:]
                    combo.pop()
                else:
                    combo.pop()
    combo = feature[-1]
```

In [14]:

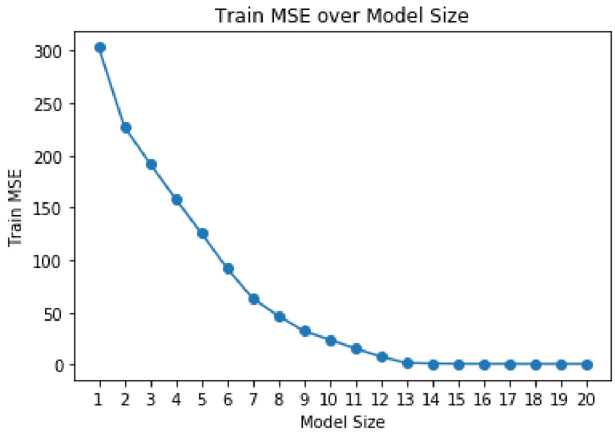
```
pl = []
n = []
count = 1
for i in MSE_list:
    pl.append(MSE_list[i][0])
    n.append(count)
    count += 1
```

In [15]:

```
pl_pd = pd.DataFrame({"num:": n, "train_mse:": pl})
plt.scatter(n, pl)
plt.plot(n, pl)
my_x_ticks = np.arange(1, 21, 1)
plt.xticks(my_x_ticks)
plt.xlabel("Model Size")
plt.ylabel("Train MSE")
```

```
plt.title('Train MSE over Model Size')

print(pd.DataFrame(pl_pd))
print("the smallest train_mse is:", min(pl))
```

```
    num:   train_mse:
0      1   303.118917
1      2   227.014339
2      3   192.133712
3      4   158.317256
4      5   125.877049
5      6    92.227197
6      7    63.622917
7      8    46.770239
8      9    32.305829
9     10    24.080479
10    11    15.754067
11    12     7.976122
12    13     1.864187
13    14     1.394237
14    15     0.957405
15    16     0.925828
16    17     0.899350
17    18     0.880390
18    19     0.872490
19    20     0.871468
the smallest train_mse is: 0.8714682252244788
```



As we can see from the result here, the model with 20 features (all features) take on the minimum train mse value

In [16]:

```
MSE_test = []
for i in MSE_list:
    MSE_test.append(MSE_list[i][1])
```

In [17]:

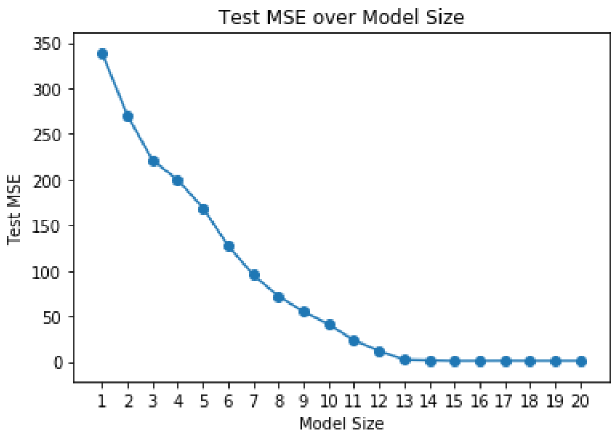
```
MSE_test_pd = pd.DataFrame({"num:": n, "mse_test:": MSE_test})
print(MSE_test_pd)
print("the smallest train_mse is:", min(MSE_test))

plt.plot(n, MSE_test)
plt.scatter(n, MSE_test)
my_x_ticks = np.arange(1, 21, 1)
plt.xticks(my_x_ticks)
plt.xlabel("Model Size")
plt.ylabel("Test MSE")
plt.title('Test MSE over Model Size')
```

```
    num:   mse_test:
0      1   338.434927
1      2   270.414985
2      3   221.195119
3      4   199.772422
```

```
4       5   169.015410
5       6   127.477382
6       7    95.513802
7       8    72.032876
8       9    54.959936
9      10    41.215390
10     11    23.715665
11     12    11.945068
12     13     2.353156
13     14     1.711854
14     15     1.101635
15     16     1.163172
16     17     1.269586
17     18     1.200368
18     19     1.191091
19     20     1.190905
the smallest train_mse is: 1.1016349936363792
```

Out[17]:

```
Text(0.5,1,'Test MSE over Model Size')
```



As we can see the model with the best test mse is the model with 15 features in it.
The test set MSE is minimized for an intermediate model size. This is very reasonable, becasue the best train MSE may not lead to the best test MSE, problem like overfitting may happen with a very good train MSE.

In [18]:

```
print("all betas equals to zero:", zero)
print("all betas in model with 15 features:", feature[14])
print("--------------------------------------------------")
count = 0
for i in feature[14]:
    print("The No."+ str(i) + " beta" )
    print("True beta: ",beta0[i])
    print("Best Model beta: ",beta_col[14][0][count])
    print("****")
    count += 1
```

```
all betas equals to zero: [ 2 18 15 15  7]
all betas in model with 15 features: [0, 19, 6, 14, 3, 11, 5, 16, 17, 12, 8, 1, 9, 13, 4]
---------------------------------------------------
The No.0 beta
True beta:  7.067056878125062
Best Model beta:  6.992450568602655
****
The No.19 beta
True beta:  8.034536246631038
Best Model beta:  8.040492089818569
****
The No.6 beta
True beta:  6.002710037431255
Best Model beta:  6.042077365341427
****
The No.14 beta
True beta:  5.0528959023080535
```

```
Best Model beta:  5.137864865002505
****
The No.3 beta
True beta:  5.063724570209043
Best Model beta:  5.050531610250283
****
The No.11 beta
True beta:  5.97784656041524
Best Model beta:  5.889088850959417
****
The No.5 beta
True beta:  5.532756285228458
Best Model beta:  5.435700174377106
****
The No.16 beta
True beta:  4.660125244255719
Best Model beta:  4.563775015377743
****
The No.17 beta
True beta:  4.200464141028771
Best Model beta:  4.292199497476145
****
The No.12 beta
True beta:  3.2219001190473238
Best Model beta:  3.354967894698495
****
The No.8 beta
True beta:  4.218326367079373
Best Model beta:  4.036561584339044
****
The No.1 beta
True beta:  3.325371654442742
Best Model beta:  3.1493336809286028
****
The No.9 beta
True beta:  2.8077545635178045
Best Model beta:  2.7027970592397073
****
The No.13 beta
True beta:  0.7481886364077006
Best Model beta:  0.7992277866437065
****
The No.4 beta
True beta:  0.7208983975434959
Best Model beta:  0.6326519769584267
****
```

As we can see from here, all the features with a zero beta are excluded from our best model
And as we can see feature No.10 is also left out of the best model. One non-zero beta No.10 equals to 0.12656218, which is very close to zero. Maybe that is exactly the reason why it is also not being included in the best model.
And as we can see, the betas of the best model we generate are very close to the betas we used to generate the true model

In [19]:

```
gen = []
for i in feature:
    l = []
    for m in i:
        l.append(beta0[m])
    gen.append(l)

beta = []
for i in beta_col:
    for m in i:
        beta.append(list(m))
```

In [20]:

```
q = []
for i in range(0, 20):
    c = [(gen[i][m] - beta[i][m])**2 for m in range(len(gen[i]))]
    q.append(c)
```

```
out = []
for m in q:
    o = (sum(m))**(1/2)
    out.append(o)
```

In [21]:

```
Final_pd = pd.DataFrame({"num:": n, "beta residual:": out})
print(Final_pd)
print(min(out))
```

```
    num:  beta residual:
0      1        3.300120
1      2        3.192288
2      3        2.619981
3      4        2.667439
4      5        2.410989
5      6        1.740795
6      7        2.022191
7      8        1.790420
8      9        1.840246
9     10        1.616330
10    11        1.601441
11    12        1.173398
12    13        0.549849
13    14        0.521437
14    15        0.390749
15    16        0.455490
16    17        0.532967
17    18        0.502434
18    19        0.497863
19    20        0.495054
0.3907493330328154
```

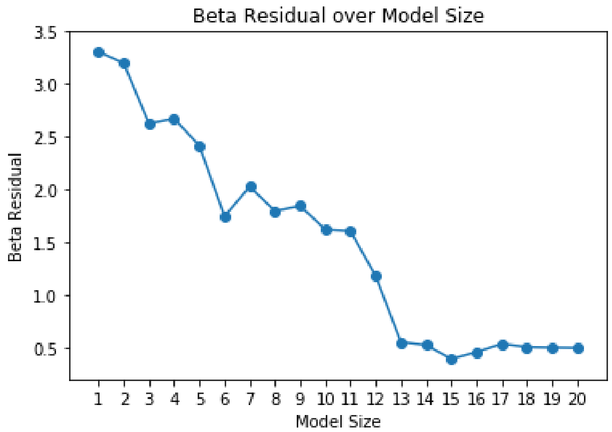
As we can see from the result that the model with the smallest beta residual is also the model with 15 features, which is constant with our previous finding.

In [22]:

```
my_x_ticks = np.arange(1, 21, 1)
plt.xticks(my_x_ticks)
plt.plot(n, out)
plt.scatter(n, out)

plt.xlabel("Model Size")
plt.ylabel("Beta Residual")
plt.title('Beta Residual over Model Size')
```

Out[22]:

```
Text(0.5,1,'Beta Residual over Model Size')
```



As we can see from the graph above, the general trend is similar to the Test MSE graph I plotted above.
And the model that generates the lowest beta residual value is the model that contains 15 features, which is also the same model that

And the model that generates the lowest beta residual value is the model that contains 15 features, which is also the same model that generates the lowest test MSE.
But one thing to notice is that low beta residual does not necessarily mean a low test mse. For example, when the the model with size 6 generates a lower beta residual than the model with size 7, but its test MSE is higher.

Application exercises

In [23]:

```
gss_train = pd.read_csv("gss_train.csv")
gss_test = pd.read_csv("gss_test.csv")
```

In [24]:

```
y_train = gss_train["egalit_scale"]
y_test = gss_test["egalit_scale"]
x_train = gss_train.drop(["egalit_scale"], axis = 1)
x_test = gss_test.drop(["egalit_scale"], axis = 1)
```

In [25]:

```
lr = LinearRegression().fit(x_train, y_train)
y_pred_train = lr.predict(x_train)
y_pred_test = lr.predict(x_test)
lr_MSE_train = mean_squared_error(y_pred_train, y_train)
lr_MSE_test = mean_squared_error(y_pred_test, y_test)
print("Least Squares Linear Train MSE: ", lr_MSE_train)
print("Leasst Squares Linear Test MSE: ", lr_MSE_test)
```

```
Least Squares Linear Train MSE:  55.12263854924573
Leasst Squares Linear Test MSE:  63.213629623014995
```

In [26]:

```
Rid = RidgeCV(cv = 10).fit(x_train, y_train)
Rid.fit(x_train, y_train)
y_pred_test = Rid.predict(x_test)
Rid_MSE_test = mean_squared_error(y_pred_test, y_test)
print("Ridge Test MSE: ", Rid_MSE_test)
print("Number of non-zero coefficients:", (Rid.coef_ != 0).sum())
```

```
Ridge Test MSE:  62.49920243957809
Number of non-zero coefficients: 77
```

In [27]:

```
las = LassoCV(cv = 10).fit(x_train, y_train)
las.fit(x_train, y_train)
y_pred_test = las.predict(x_test)
Lasso_MSE_test = mean_squared_error(y_pred_test, y_test)
print("Lasso Test MSE: ", Lasso_MSE_test)
print("Number of non-zero coefficients:", (las.coef_ != 0).sum())
```

```
Lasso Test MSE:  62.7780157899344
Number of non-zero coefficients: 24
```

In [28]:

```
en = ElasticNetCV(alphas = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1], cv = 10).fit(x_train, y
_train)
en.fit(x_train, y_train)
mse = mean_squared_error(en.predict(x_train), y_train)
test_mse = mean_squared_error(en.predict(x_test), y_test)
print("Best Model Lambda: ",en.alpha_)
print("Best Model alphas: ",en.l1_ratio_)
print("Best Model Train MSE:", mse)
print("Best Model Test MSE:", test_mse)
```

```
print("Best Model Test MSE:", test_mse)
print("Number of non-zero coefficients:", (en.coef_ != 0).sum())
```

```
Best Model Lambda:  0.1
Best Model alphas:  0.5
Best Model Train MSE: 57.071744805780774
Best Model Test MSE: 62.5070860872212
Number of non-zero coefficients: 40
```

As we can see from the results above, all models we use generate a test MSE around 62 to 63. Least Square Linear perform worst while Ridge gives the best result. Generally speaking, these models perform similarly and poorly in terms of predicting an individual's egalitarianism. Maybe these models cant help solving this kind of problem very well. Trying other models may help.