

Needell_Coen_HW3

February 6, 2020

```
[2]: import numpy as np
import pandas as pd
from matplotlib import pyplot as plt
```

0.1 # Conceptual Exercises

0.2 ## Training/test error for subset selection

0.2.1 1

```
[3]: np.random.seed(666)
# sd is 5 because I think it'll make the numbers more interesting.
df1 = {f'x{n}': np.random.normal(0, 5, 1000) for n in range(1, 21)}
df1 = pd.DataFrame(df1)
epsilon = np.random.normal(0, 5, 1000)

bzeros = np.random.randint(1, 20, 5) # decides random betas to be zero
betas = {f'x{n}': (np.random.normal(0, 5)
                    if n not in bzeros else 0) for n in range(1, 21)}

y = np.zeros(1000)
for i in range(1, 21):
    y += df1[f'x{i}'] * betas[f'x{i}']
y += epsilon
```

0.2.2 2

```
[4]: from sklearn.model_selection import train_test_split

x_train, x_test, y_train, y_test = train_test_split(df1, y, test_size=900)
```

0.2.3 3

```
[5]: from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error as mse
from itertools import combinations
from functools import partial
import wes
```

```
[7]: def process_subset(feats, xtr, xte, ytr, yte, returnmodel=False):
    train_feats = xtr[feats]
    test_feats = xte[feats]
    model = LinearRegression()
    model.fit(train_feats, ytr)
    if returnmodel:
        return model
    errte = mse(yte, model.predict(test_feats))
    errtr = mse(ytr, model.predict(train_feats))
    return errtr, errte

# I'm using a forward_stepwise algorithm, assuming that the features
# have some best order which is independent of other features.
# That assumption is baked into linear models though, so it's safe here.
# Also we know the data generating function, and we know that they're
# independant, so we know that both algorithms go to the same result.

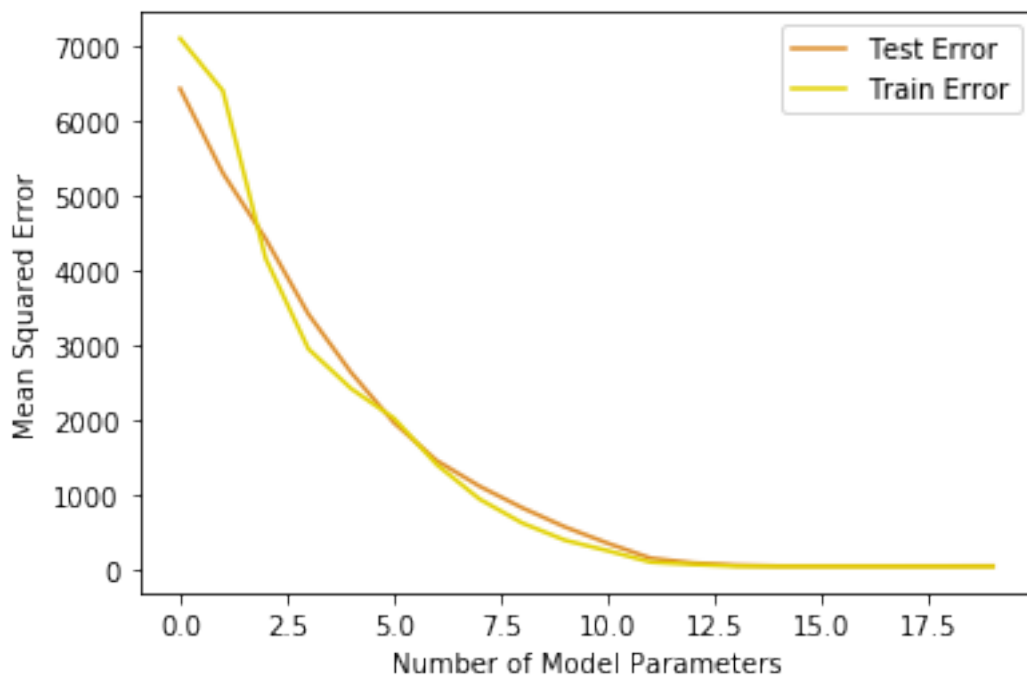
# Also 2 ** 20 = 1,000,000 which is a little out-of-feasibility for python.
def forward_stepwise(xtr, xte, ytr, yte):
    n = len(xtr.keys())
    model = set() # This is a set to which we will add features.
    params = []
    testerrs = []
    trainerrs = []
    models = []
    # This is a list where we'll record
    #the best process_subset outputs

    for k in range(0, n):
        augs = [str(st) for st in xtr.keys() if st not in model]
        results = []
        for x in augs:
            results.append(process_subset(list(model) + [x],
                                         xtr, xte, ytr, yte))
        results = np.array(results)
        best = np.argmin(results[:, 1])
        model.add(augs[best])
        betr, bete = results[best]
        params.append(augs[best])
        testerrs.append(bete)
        trainerrs.append(betr)
    record = pd.DataFrame({"param":params,
                           "test_err":testerrs,
                           "train_err":trainerrs})

    return record

best_selec = forward_stepwise(x_train, x_test, y_train, y_test)
```

```
[8]: wes.set_palette('FantasticFox1')
plt.plot(best_selec.test_err, label='Test Error')
plt.plot(best_selec.train_err, label='Train Error')
plt.xlabel('Number of Model Parameters')
plt.ylabel('Mean Squared Error')
plt.legend()
plt.show()
best_selec
```



```
[8]: param    test_err    train_err
0     x8  6416.784511  7089.241271
1     x16  5289.490369  6390.208384
2     x11  4409.229291  4147.456895
3     x12  3402.061119  2937.561717
4     x19  2620.542325  2405.103896
5      x9  1950.391660  2010.545592
6      x3  1443.183115  1389.383800
7     x10  1100.965204   928.892155
8     x15   811.961551   608.203719
9      x5   555.418121   378.544639
10    x20   337.576521   238.096790
11     x2   138.647058    85.752510
12    x17    74.697425    54.693290
13     x7    45.602417    30.619249
14     x1    34.774484    27.710232
```

15	x6	34.737711	27.688195
16	x13	34.740764	27.671565
17	x18	34.782783	27.651672
18	x14	35.063060	27.478137
19	x4	36.026689	27.269170

Unsurprisingly, it starts to even out around 13. If we look at the list of betas it's easy to see why, we know that 5 of the variables are useless, and another two have $|\beta| < 1$. If you look at the parameter that's added in for each step, and compare that to the list of betas, you can see the logic of it. The betas which are set to zero are associated with parameters that are included last.

```
[9]: pd.DataFrame(betas, index=['beta']).transpose()
```

```
[9]:      beta
x1  -0.717342
x2  -2.844386
x3  -4.660543
x4   0.000000
x5  -2.933093
x6  -0.002561
x7   1.038511
x8  -9.030795
x9  -5.204751
x10  3.710081
x11  6.849511
x12  6.229572
x13  0.000000
x14  0.000000
x15  3.321539
x16  5.709028
x17  1.531583
x18  0.000000
x19 -5.046486
x20 -2.791455
```

The high efficiency models (15 and 14) remove x1 and x6 plus the zeroes. Then the zeroes are added back in in order.

0.2.4 4

See plot for # 3. ### 5

```
[10]: best_test = np.argmin(best_selec.test_err)
m = ','.join(best_selec.param[:best_test].sort_values())
m
```

```
[10]: 'x1,x10,x11,x12,x15,x16,x17,x19,x2,x20,x3,x5,x7,x8,x9'
```

So the best model is the model with 16 parameters. Why exactly it's at this number of parameters and not smaller or larger is somewhat interesting. We're measuring performance on the test set, for one, and chances are good that noise effects detected in the additional variables after the 16th are causing some overfitting when those variables are in the linear model. ### 6 We see that the best model contains all of the features except for those which have coefficients closest to zero. Give or take a few. Because of noise, we see one of the coefficient-zero parameters in the mix, as well as a close to zero parameter treated as if it's coefficient is zero.

```
[11]: feats = best_selec.param[:best_test]
m = process_subset(feats, x_train, x_test, y_train, y_test, returnmodel=True)
coef_info = pd.DataFrame({'feats': feats, 'coefs': m.coef_, 'betas':
    ↪ [betas[feat] for feat in feats]})
coef_info
```

```
[11]:
```

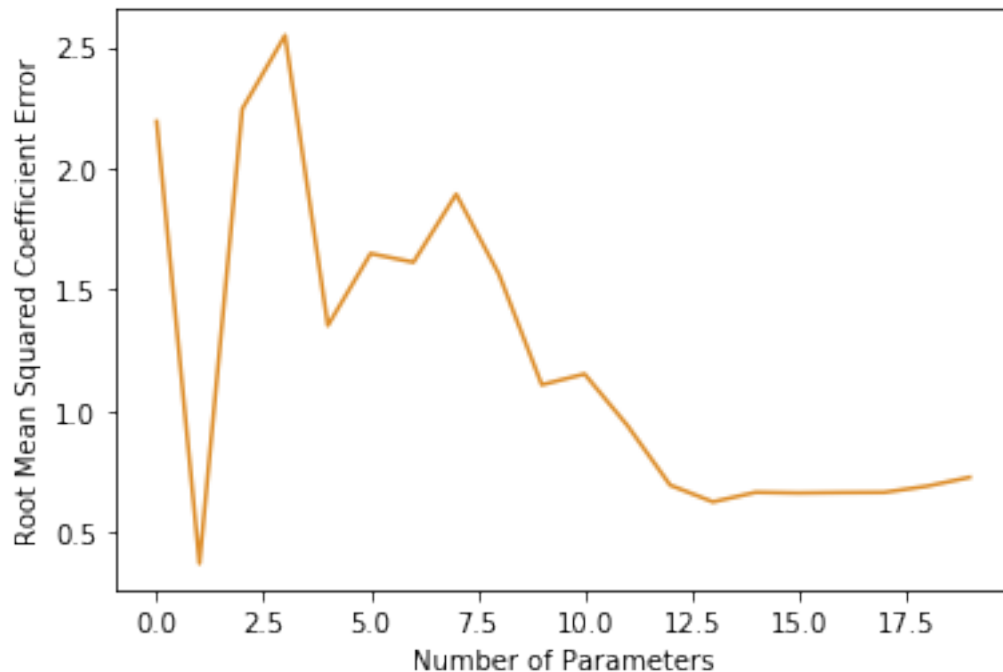
	feats	coefs	betas
0	x8	-9.423881	-9.030795
1	x16	5.525469	5.709028
2	x11	6.976860	6.849511
3	x12	6.230386	6.229572
4	x19	-4.830579	-5.046486
5	x9	-5.160344	-5.204751
6	x3	-4.709259	-4.660543
7	x10	3.847959	3.710081
8	x15	3.447054	3.321539
9	x5	-3.039797	-2.933093
10	x20	-2.759123	-2.791455
11	x2	-2.932852	-2.844386
12	x17	1.387939	1.531583
13	x7	0.886101	1.038511
14	x1	-0.424171	-0.717342

0.2.5 7

```
[12]: rmcoefaccs = []
for r in range(1, 21):
    feats = best_selec.param[:r]
    m = process_subset(feats, x_train, x_test, y_train, y_test,
    ↪ returnmodel=True)
    bhat = m.coef_
    b = np.array([betas[f] for f in feats])
    inner = (b - bhat) ** 2
    outer = np.sqrt(np.sum(inner))
    rmcoefaccs.append(outer)

plt.plot(rmcoefaccs)
plt.ylabel('Root Mean Squared Coefficient Error')
plt.xlabel('Number of Parameters')
```

```
plt.show()
```



The first thing we notice is this huge hump between 5 parameters and 10. This is due to the fact that the linear regression algorithm is using the available parameters to spuriously explain other parameters. We can compare this to the MSE plot where the error pretty much declines steadily as we add in more parameters, in this case we see a sort of distribution of error, before we get into the more accurate models about $p = 15$.

0.3 # Application Exercises

Please note that `sklearn` calls the λ parameter α . I will use this convention.

```
[15]: from sklearn.linear_model import RidgeCV, LassoCV, ElasticNetCV
gss_tr = pd.read_csv('data/gss_train.csv')
gss_te = pd.read_csv('data/gss_test.csv')
x_tr = gss_tr.drop('egalit_scale', axis=1)
y_tr = gss_tr.egalit_scale
x_te = gss_te.drop('egalit_scale', axis=1)
y_te = gss_te.egalit_scale
```

0.3.1 1

```
[16]: def report_mse(model):  
      model.fit(x_tr, y_tr)  
      err = mse(model.predict(x_te), y_te)  
      return err, model
```

```
[17]: print(report_mse(LinearRegression())[0])
```

63.213629623014995

0.3.2 2

```
[18]: print(report_mse(RidgeCV(alphas=np.logspace(-5, 5)))[0])
```

62.202521583380204

A little better...

```
[19]: mes, mod = report_mse(LassoCV(alphas=np.logspace(-5, 5)))  
      print(f'mse is {mes}')  
      print(f'There are {sum(np.isclose(mod.coef_, 0))} non-zero coefficient_  
            ↳estimates')
```

mse is 62.90446289883912

There are 54 non-zero coefficient estimates

woof

```
[21]: mes, mod = report_mse(ElasticNetCV(l1_ratio=np.linspace(.1,1,11)))  
      print(f'mse is {mes}')  
      print(f'There are {sum(np.isclose(mod.coef_, 0))} non-zero coefficient_  
            ↳estimates')
```

mse is 62.81645879274796

There are 54 non-zero coefficient estimates

Gotta say, we're not exactly killing it on the accuracy front. The feature ranges from 1 to 35, and the best of these methods, RidgeCV, was off by 7.8 points on average. Which I guess isn't so bad, but a more complicated method would, I'm sure, do a lot better. One of the interesting things from this exercise is that all of these MSEs are within 1 point of each other, which tells us that the regression method isn't having too much of an effect on the results. This implies that what we really need is a non-linear model, as we've probably hit the upper bound on what is possible with linear regressions.

I do also find it interesting that the Elastic Net considers more coefficients than the Lasso.