Needell_Coen_HW3

February 4, 2020

[76]:
```
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

[77]:
```
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

[78]:
```
from sklearn.model_selection import train_test_split

x_train, x_test, y_train, y_test = train_test_split(df1, y, test_size=100)
```

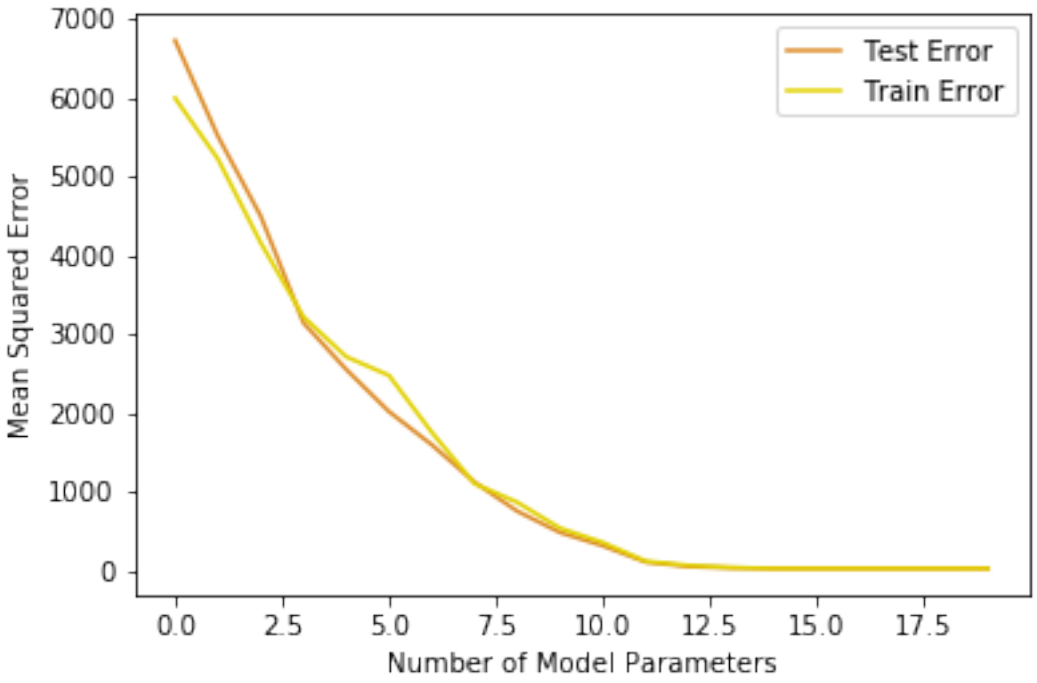
0.2.3 3

[79]:
```
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
```

[82]:
```
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.
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)
```

[83]:
```
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
```



[83]:

	param	test_err	train_err
0	x8	6722.007371	5990.764886
1	x16	5509.094946	5219.571120
2	x11	4495.923015	4156.927315
3	x12	3140.898534	3216.218938
4	x3	2549.805095	2710.059889
5	x2	2014.640384	2473.133942
6	x19	1594.110284	1752.299484
7	x9	1119.566651	1100.439233
8	x15	750.643728	863.342414
9	x10	480.848367	535.761153
10	x20	312.460108	353.891694
11	x5	106.423464	116.470260
12	x17	49.273404	64.045148
13	x7	32.661806	36.762260
14	x1	24.551930	25.388498
15	x4	24.395178	25.377402
16	x18	24.383927	25.366930

```
17   x13    24.383871    25.366832
18    x6    24.522814    25.350192
19   x14    24.987858    25.136589
```

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.

[84]:
```
pd.DataFrame(betas, index=['beta']).transpose()
```

[84]:
```
          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

[85]:
```
best_test = np.argmin(best_selec.test_err)
m = ','.join(best_selec.param[:best_test].sort_values())
m
```

[85]:
```
'x1,x10,x11,x12,x15,x16,x17,x18,x19,x2,x20,x3,x4,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 coefficents 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.

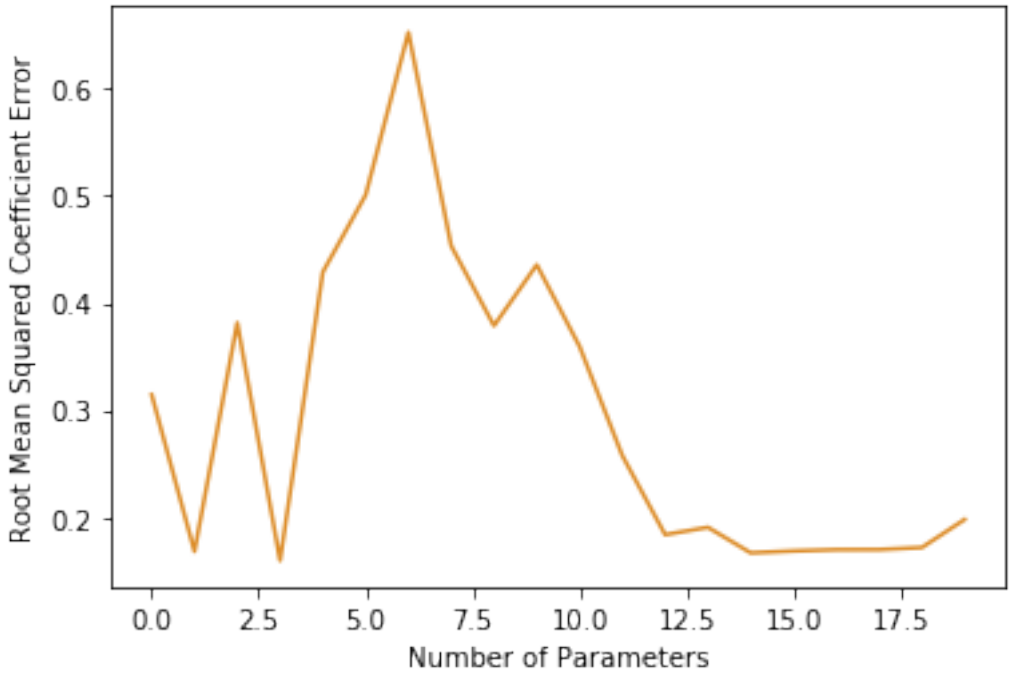
[86]:
```
feats = best_selec.param[:best_test]
m = process_subset(feats, x_train, x_test, y_train, y_test, returnmodel=True)
coef_info = pd.DataFrame({'feats': feats, 'coefs': m.coef_, 'betas':␣
↪[betas[feat] for feat in feats]})
coef_info
```

[86]:

	feats	coefs	betas
0	x8	-9.057275	-9.030795
1	x16	5.687508	5.709028
2	x11	6.885942	6.849511
3	x12	6.236172	6.229572
4	x3	-4.667103	-4.660543
5	x2	-2.880743	-2.844386
6	x19	-4.991266	-5.046486
7	x9	-5.173359	-5.204751
8	x15	3.425403	3.321539
9	x10	3.645617	3.710081
10	x20	-2.811505	-2.791455
11	x5	-2.981893	-2.933093
12	x17	1.540596	1.531583
13	x7	1.011963	1.038511
14	x1	-0.679074	-0.717342
15	x4	-0.020901	0.000000
16	x18	-0.020657	0.000000

0.2.5 7

[87]:
```
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.

[88]:
```
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

[89]:
```
def report_mse(model):
    model.fit(x_tr, y_tr)
    err = mse(model.predict(x_te), y_te)
    return err, model
```

[90]:
```
print(report_mse(LinearRegression())[0])
```

```
63.213629623014995
```

0.3.2 2

[91]:
```
print(report_mse(RidgeCV(alphas=np.logspace(-5, 5)))[0])
```

```
62.202521583380204
```

A little better...

[92]:
```
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 coefficent␣
 ↪estimates')
```

```
mse is 62.90446289883912
There are 54 non-zero coefficent estimates
```

woof

[93]:
```
mes, mod = report_mse(ElasticNetCV(alphas=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 62.507086087221204
There are 37 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.

[ ]:
```

```