In [368]:
```python
import sklearn
#import sklearn.naive_bayes
#import sklearn.tree
#import sklearn.ensemble
#import sklearn.neural_network
#import sklearn.decomposition
from sklearn import datasets, linear_model
from sklearn.model_selection import train_test_split

import nltk
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.colors
import seaborn
import scipy as sp

import collections
import os
import os.path
import random
import re
import glob
import pandas
import requests
#import json
import math

from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error as MSE

sns.set_style("whitegrid")

%matplotlib inline
```

Training/test error for subset selection

1.

In [369]:
```
np.random.seed(5)
```

In [370]:
```
#Generate 1000x20 matrix of random values (X = np)
X = np.random.randint(10, size=(1000, 20))
X
Xdf = pd.DataFrame(X)
Xdf.columns
```

Out[370]:
```
RangeIndex(start=0, stop=20, step=1)
```

In [372]:
```
beta = [[0.4667656498305942,0.38517076115123844, 0.0, 0.9466665658919328, 0.8246125965327541,
         0.9596005738364424,0.8155693086821791, 0.0, 0.9452070230519676, 0.9331533432762459, 0.0, 0.20882167645312635,
         0.840590313744625, 0.19494971114461077, 0.0, 0.8100856109889598, 0.7997112653935964, 0.0, 0.16143863542515063,
         0.2240441650311026]] #randomly generated 15 beta values (using np.random.random()) and put them in a list,
                             #interspersed 5 "0" values
beta = np.asarray(beta).T
```

In [ ]:
```
#Generating error term values
epsilon = np.random.normal(0, .5, 1000)
epsilon
```

In [374]:
```
print(Xdf.shape, beta.shape, epsilon.shape)
```

```
(1000, 20) (20, 1) (1000,)
```

In [378]:
```
Y = np.matmul(X, beta) + epsilon.reshape((1000, 1))
```

2.

In [379]:
```
X_train, X_test, Y_train, Y_test = train_test_split(Xdf, Y, test_size=9
00)
print(X_train.shape, Y_train.shape)
print(X_test.shape, Y_test.shape)
```

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

In [388]:
```
Xdf = pd.DataFrame(X)
Xdf
```

Out[388]:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
0	3	6	6	0	9	8	4	7	0	0	7	1	5	7	0	1	4	6	2	9
1	9	9	9	1	2	7	0	5	0	0	4	4	9	3	2	4	6	9	3	3
2	2	1	5	7	4	3	1	7	3	1	9	5	7	0	9	6	0	5	2	8
3	6	8	0	5	2	0	7	7	6	0	0	8	5	5	9	6	4	5	2	8
4	8	1	6	3	4	1	8	0	2	2	4	1	6	3	4	3	1	4	2	3
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
995	0	3	5	4	5	1	8	1	4	3	3	7	8	7	1	6	8	1	0	2
996	4	0	3	6	1	5	3	9	9	3	7	5	0	7	4	0	5	4	8	7
997	8	4	1	8	6	3	5	6	9	0	2	8	8	0	9	3	7	2	5	0
998	7	2	3	3	8	7	2	1	6	1	4	3	4	1	7	8	8	3	2	3
999	8	1	8	1	9	1	7	0	7	1	7	5	3	4	0	8	7	1	3	4

1000 rows × 20 columns

In [393]:
```
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error as mse
from itertools import combinations
from functools import partial
```

3.

Creating linear regression model with training data and using that for best subset selection (inputs were X_traindf and Y_traindf).

In [394]:
```
def subset(preds, X_train, X_test, Y_train, Y_test, returnmodel=False):

    tr_preds = X_train[preds] #Conferred with Coen Needell but generate
d separate functions
    te_preds = X_test[preds] #and we each understand how they work
    m = LinearRegression()
    m.fit(preds, Y_train)
    if returnmodel:
        return m
    err_test = mse(Y_test, m.predict(te_preds))
    err_train = mse(Y_train, m.predict(tr_preds))
    return err_train, err_test


def f_step(X_train, X_test, Y_train, Y_test): #functionally best subset
 selection
    n = len(X_train.keys())
    mod = set()
    feats = []
    errors_test = []
    errors_train = []
    mods = []

    for k in range(0, n):
        ag = [st for st in X_train.keys() if st not in mod]
        best_sub = []
        for x in ag:
            best_sub.append(subset(list(mod) + [x], X_train, X_test, Y_
train, Y_test))
        best_sub = np.array(best_sub)
        best = np.argmin(best_sub[:, 1])
        mod.add(ag[best])
```

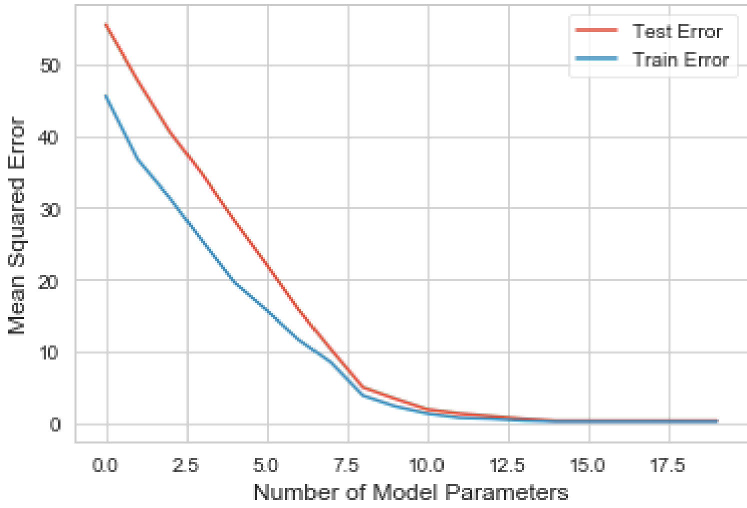
```
        best_train, best_test = best_sub[best]
        feats.append(ag[best])
        errors_test.append(best_test)
        errors_train.append(best_train)
    table = pd.DataFrame({"params":feats,"test_err":errors_test,"train_
err":errors_train})
    return table

best_subset_select = f_step(X_train, X_test, Y_train, Y_test)
```

Based on the plot and table output below:

- The training MSE decreases as the number of predictors in the model increases, with the MSE starting to plateau around the 14th iteration. The model with the lowest training MSE is comprised of all 20 features (MSE = 0.228413). Beta = 0 at indices 2, 7, 10, 14, 17 (these indices correspond with param #/index values since there are 20 betas and 20 params), and these are the features that were added last in the best subset selection.

4. (Plotting training and test MSE, so includes part of 3)

In [395]:
```
plt.plot(best_subset_select.test_err, label='Test Error')
plt.plot(best_subset_select.train_err, label='Train Error')
plt.xlabel('Number of Model Parameters')
plt.ylabel('Mean Squared Error')
plt.legend()
plt.show()
best_subset_select
```



Out[395]:

	param	test_err	train_err
0	8	55.506835	45.601776
1	3	47.734132	36.730073
2	6	40.577620	31.398178
3	5	34.709568	25.426803
4	12	28.264352	19.627414
5	9	22.154479	15.758091
6	4	15.832952	11.655019
7	16	10.305941	8.498480
8	15	4.993678	3.837262
9	0	3.406169	2.337423
10	1	1.923025	1.334687
11	19	1.330933	0.745090
12	11	0.959929	0.607506
13	13	0.584895	0.348869

	param	test_err	train_err
14	18	0.323589	0.230768
15	10	0.323309	0.230732
16	17	0.323273	0.230732
17	7	0.323325	0.230621
18	2	0.324974	0.228499
19	14	0.326186	0.228413

5.

- The test set MSE takes on its lowest value with the model that included parameters 8, 3, 6, 5, 12, 9, 4, 16, 15, 0, 1, 19, 11, 13, 18, 10, and 17 (17 params total, MSE = 0.323273). It did not take on its minimum value for a model containing only an intercept or a model containing all of the features. I explain more implications in my response to question 6.

6.

In [396]:
```
best_model = subset([8, 3, 6, 5, 12, 9, 4, 16, 15, 0, 1, 19, 11, 13, 18
, 10, 17], X_train, X_test, Y_train, Y_test, returnmodel=True)
```

In [397]:
```
best_model
```

Out[397]:
```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None, normaliz
e=False)
```

In [398]:
```
best_model.coef_
```

Out[398]:
```
array([[ 9.51569160e-01,  9.48440877e-01,  8.10492697e-01,
         9.50959615e-01,  8.67817017e-01,  9.28203552e-01,
         8.66074377e-01,  7.79585712e-01,  8.21975008e-01,
```

```
             4.68286975e-01,  3.68079817e-01,  2.80921853e-01,
             1.93712416e-01,  1.91627545e-01,  1.34606021e-01,
             2.48243385e-03, -2.06338218e-04]])
```

In [399]:

```
best_model_coeff = ["8", 9.51569160e-01,  "3", 9.48440877e-01,  "6", 8.
10492697e-01, "5", 9.50959615e-01,
                    "12", 8.67817017e-01,  "9", 9.28203552e-01, "4", 8.
66074377e-01,  "16", 7.79585712e-01,
                    "15", 8.21975008e-01, "0", 4.68286975e-01,  "1", 3.
68079817e-01,  "19", 2.80921853e-01,
                     "11", 1.93712416e-01,  "13", 1.91627545e-01,  "18"
, 1.34606021e-01, "10", 2.48243385e-03,
                    "17", -2.06338218e-04] #labeling indices
best_model_coeff
```

Out[399]:

```
['8',
 0.95156916,
 '3',
 0.948440877,
 '6',
 0.810492697,
 '5',
 0.950959615,
 '12',
 0.867817017,
 '9',
 0.928203552,
 '4',
 0.866074377,
 '16',
 0.779585712,
 '15',
 0.821975008,
 '0',
 0.468286975,
 '1',
 0.368079817,
 '19',
 0.280921853,
 '11',
```

```
 0.193712416,
 '13',
 0.191627545,
 '18',
 0.134606021,
 '10',
 0.00248243385,
 '17',
 -0.000206338218]
```

In [400]:

```python
best_original_betas = ["8", beta[8][0], "3", beta[3][0], "6", beta[6][0
], "5", beta[5][0],
                        "12", beta[12][0], "9", beta[9][0], "4", beta[4]
[0], "16", beta[16][0],
                        "15", beta[15][0], "0", beta[0][0], "1", beta[1]
[0], "19", beta[19][0],
                        "11", beta[11][0], "13", beta[13][0], "18", beta
[18][0], "10", beta[10][0],
                       "17", beta[17][0]] #labeling indices



best_original_betas
```

Out[400]:

```
['8',
 0.9452070230519676,
 '3',
 0.9466665658919328,
 '6',
 0.8155693086821791,
 '5',
 0.9596005738364424,
 '12',
 0.840590313744625,
 '9',
 0.9331533432762459,
 '4',
 0.8246125965327541,
 '16',
```

```
0.7997112653935964,
'15',
0.8100856109889598,
'0',
0.4667656498305942,
'1',
0.38517076115123844,
'19',
0.2240441650311026,
'11',
0.20882167645312635,
'13',
0.19494971114461077,
'18',
0.16143863542515063,
'10',
0.0,
'17',
0.0]
```

- I ran a linear regression on the subset of parameters with the lowest test MSE and put the coefficients it generated into a list. Then, I compared them with the corresponding original betas. The coefficients are overall similar, which we would expect, since the better the model, the better it can capture the underlying beta- so it should have the similar predicted beta close to the original beta.

- The best-performing model/best subset includes 17 parameters (Y = X8(0.95) + X3(0.948)....+ episilon. Apart fom param 17, none of the 17 parameters had original betas that equalled 0, which makes sense because that means those parameters had little weight, which means that they wouldn't have contributed much to the model/it made sense to weed them out.

7.

In [407]:
```
feat_full = [8, 3, 6, 5, 12, 9, 4, 16, 15, 0, 1, 19, 11, 13, 18, 10, 17
```

```
, 7, 2, 14]
```

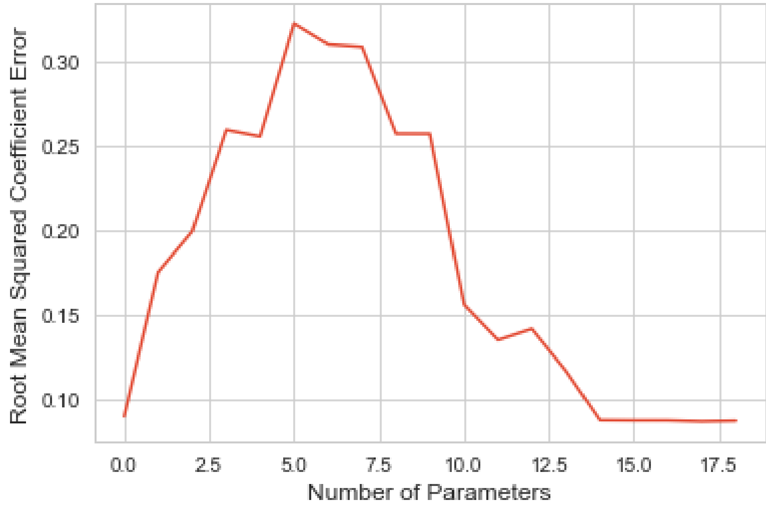
In [434]:
```
rss_ls = []
for i in range(19):
    model_i = subset(feat_full[:i+1], X_train, X_test, Y_train, Y_test,
 returnmodel=True)
    b_hats = model_i.coef_
    #print(b_hats)
    diff_beta = b_hats[0] - [b[0] for b in beta[feat_full[:i+1]]]
    diff_sq = [diff**2 for diff in diff_beta]
    sum_sq = np.array(diff_sq).sum()
    rss = sum_sq**0.5
    rss_ls.append(rss)
```

In [435]:
```
rss_ls
```

Out[435]:
```
[0.08995076382829204,
 0.1750566914946543,
 0.19902854813313006,
 0.2589832975679624,
 0.25512608298073747,
 0.3218020316447201,
 0.30958503689450634,
 0.3081201159902431,
 0.2567611184149912,
 0.2566971129449609,
 0.1559254600178762,
 0.13504812303136607,
 0.1416307192838785,
 0.11682804987922703,
 0.08772639138101153,
 0.08759349126471999,
 0.08758047045140707,
 0.08702699348445667,
 0.0873807396069241]
```

In [436]:
```
plt.plot(rss_ls)
plt.ylabel('Root Mean Squared Coefficient Error')
```

```
plt.xlabel('Number of Parameters')
plt.show()
```



- We can see that root mean squared coefficient error increases as the number of parameters increases, and then decreases after about 7 parameters and plateaus around 15 parameters. This graph gives us a distribution plot of errors; in the MSE plot we can see that as more features are added, MSE declines.
- This makes sense because when starting best subset selection, the algorithm didn't have enough information about the other parameters, and once it ran through almost all the parameters the RMSE evened out.

Application exercises

In [438]:
```
gss_tr = pd.read_csv('/Users/Sruti/Downloads/gss_train.csv')
gss_te = pd.read_csv('/Users/Sruti/Downloads/gss_test.csv')
```

In [507]:
```
gss_tr.head() #78 predictors, 77 once we drop egalitarianism since tha
t's what we're predicting
```

Out[507]:

	age	attend	authoritarianism	black	born	childs	colath	colrac	colcom	colmil	...	zodiac_(
0	21	0	4	0	0	0	1	1	0	1	...	
1	42	0	4	0	0	2	0	1	1	0	...	
2	70	1	1	1	0	3	0	1	1	0	...	
3	35	3	2	0	0	2	0	1	0	1	...	
4	24	3	6	0	1	3	1	1	0	0	...	

5 rows × 78 columns

In [441]:
```
gss_tr.egalit_scale
```

Out[441]:
```
0       22
1       14
2       20
3       34
4       35
        ..
1476    18
1477    29
1478    13
1479    22
1480    25
Name: egalit_scale, Length: 1481, dtype: int64
```

In [442]:
```
gss_te.egalit_scale
```

Out[442]:
```
0      13
1       1
2      28
3       7
4      20
       ..
488    29
489     1
```

```
490    26
491     4
492    14
Name: egalit_scale, Length: 493, dtype: int64
```

1.

In [450]:
```
x_train = gss_tr.drop(['egalit_scale'], axis=1)
x_test = gss_te.drop(['egalit_scale'], axis=1)
y_train = gss_tr['egalit_scale']
y_test = gss_te['egalit_scale']
ls_linear_model = LinearRegression().fit(x_train, y_train)
MSE = mean_squared_error(ls_linear_model.predict(x_test), y_test)
print(MSE)
```

```
63.213629623014995
```

Test MSE from least squares linear model (above)

2.

In [496]:
```
from sklearn.datasets import load_boston
from sklearn.linear_model import RidgeCV
alphas = np.arange(0.1,10,0.1)
regressor = RidgeCV(alphas = alphas) #alpha = lambda, default = 10 fold
s
regressor.fit(x_train, y_train)
MSE = mean_squared_error(regressor.predict(x_test) , y_test) #now we're
 predicting y_test using the best alpha
print(MSE) #test mse based on the best trained mse
```

```
62.50220161742881
```

Test MSE from Ridge Regression (above)

3.

In [506]:
```
from sklearn.linear_model import Lasso, ElasticNetCV
alphas = np.arange(0.1,10,0.1)
regressor = LassoCV(alphas = alphas) #alpha = lambda, default = 10 fold
s
regressor.fit(x_train, y_train)
MSE = mean_squared_error(regressor.predict(x_test) , y_test) #now we're
 predicting y_test using the best alpha
print(MSE)
len(regressor.coef_) - sum(np.isclose(regressor.coef_, 0))
```

```
/Users/Sruti/opt/anaconda3/lib/python3.7/site-packages/sklearn/model_se
lection/_split.py:1978: FutureWarning: The default value of cv will cha
nge from 3 to 5 in version 0.22. Specify it explicitly to silence this
warning.
  warnings.warn(CV_WARNING, FutureWarning)
```

```
62.77841555477389
```

Out[506]:
```
24
```

Test MSE from Lasso Regression (above) and number of non-zero coefficient estimates (24/77 predictors meaningfully contribute to the model)

4.

In [505]:
```
alphas = np.arange(0.0,1,0.1)
regressor = ElasticNetCV(alphas = alphas) # 0-1 in increments of 0.1, f
old default = 10
regressor.fit(x_train, y_train)
MSE = mean_squared_error(regressor.predict(x_test) , y_test) #predictin
g y_test using the best alpha/lambda
print(MSE)
len(regressor.coef_) - sum(np.isclose(regressor.coef_, 0))
```

```
/Users/Sruti/opt/anaconda3/lib/python3.7/site-packages/sklearn/model_se
lection/_split.py:1978: FutureWarning: The default value of cv will cha
```

```
nge from 3 to 5 in version 0.22. Specify it explicitly to silence this
warning.
  warnings.warn(CV_WARNING, FutureWarning)
```

```
62.5070860872212
```

```
/Users/Sruti/opt/anaconda3/lib/python3.7/site-packages/sklearn/linear_m
odel/coordinate_descent.py:471: UserWarning: Coordinate descent with al
pha=0 may lead to unexpected results and is discouraged.
  tol, rng, random, positive)
/Users/Sruti/opt/anaconda3/lib/python3.7/site-packages/sklearn/linear_m
odel/coordinate_descent.py:471: ConvergenceWarning: Objective did not c
onverge. You might want to increase the number of iterations. Duality g
ap: 26852.762875207496, tolerance: 9.302133130699088
  tol, rng, random, positive)
/Users/Sruti/opt/anaconda3/lib/python3.7/site-packages/sklearn/linear_m
odel/coordinate_descent.py:471: UserWarning: Coordinate descent with al
pha=0 may lead to unexpected results and is discouraged.
  tol, rng, random, positive)
/Users/Sruti/opt/anaconda3/lib/python3.7/site-packages/sklearn/linear_m
odel/coordinate_descent.py:471: ConvergenceWarning: Objective did not c
onverge. You might want to increase the number of iterations. Duality g
ap: 24581.63105170257, tolerance: 8.938171428571428
  tol, rng, random, positive)
/Users/Sruti/opt/anaconda3/lib/python3.7/site-packages/sklearn/linear_m
odel/coordinate_descent.py:471: UserWarning: Coordinate descent with al
pha=0 may lead to unexpected results and is discouraged.
  tol, rng, random, positive)
/Users/Sruti/opt/anaconda3/lib/python3.7/site-packages/sklearn/linear_m
odel/coordinate_descent.py:471: ConvergenceWarning: Objective did not c
onverge. You might want to increase the number of iterations. Duality g
ap: 27776.117727362333, tolerance: 9.14361568825911
  tol, rng, random, positive)
```

Out[505]:

```
40
```

Test MSE from Elastic Net Regression (above) and number of non-zero coefficient estimates (40/77 predictors meaningfully contribute to the model)

5.

All four models have similar test error - the linear model had the highest (about 63, which makes sense given no tuning), while Ridge Regression had the lowest (about 62.5). Given that all four of these are regression-methods and the MSE's only differ by <1, these regressions are all performing about the same (i.e., predicting an individual's egalitarianism with roughly the same degree of accuracy - but test error of 62-63 is not great). A more complex (e.g., non-linear) model might do a better job.

```
In [ ]:
```