EE 460J Lab 2

Team:
Johnson Zhang - xz5993
David Rollins - Der2366
Peter Wagenaar - pjw845

In [2]:
```
from __future__ import division
import numpy as np
import matplotlib.pyplot as plt
from sklearn import datasets, linear_model
import pandas as pd
from pandas import DataFrame, Series
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.metrics import r2_score
sns.set(style='ticks', palette='Set2')
%matplotlib inline
```

In [3]:
```
# Generate data
np.random.seed(7)

n_samples, n_features = 100, 200
X = np.random.randn(n_samples, n_features)

k = 5
# beta generated with k nonzeros
#coef = 10 * np.random.randn(n_features)
coef = 10 * np.ones(n_features)
inds = np.arange(n_features)
np.random.shuffle(inds)
coef[inds[k:]] = 0  # sparsify coef
y = np.dot(X, coef)

# add noise
y += 0.01 * np.random.normal((n_samples,))

# Split data in train set and test set
n_samples = X.shape[0]
X_train, y_train = X[:25], y[:25]
X_test, y_test = X[25:], y[25:]
```

In [4]:
```
# Problem 2
```

In [5]:
```
#2.1
#Step forward feature selection starts with the evaluation of each individual fe
#and selects that which results in the best performing selected algorithm model

alpha = 0.2
train_samples = 25
test_samples = 75


## Lecture 8 explains idea - we need to project y onto the feature then replace
# use this to find the best error
def evaluate_feature_error(feature_train, feature_test, y_train=y_train, y_test=
    bias = alpha*np.eye(feature_train.shape[1])
    inverse = np.linalg.inv(np.dot(feature_train.T, feature_train)) + bias
    proj_1 = np.dot(feature_train.T, np.reshape(y_train, (train_samples,1)))
    beta_hat = np.dot(inverse, proj_1)
    proj_2 = np.reshape(y_test, (test_samples,1))
    error_vec = proj_2 - np.dot(feature_test, beta_hat)
```

```
    return np.linalg.norm(error_vec, ord=2)**2



feature_selection = []
errors = [None for i in range(0,n_features)]
best_scores = [] # best scores will save all the minimum errors from the feature

train_shape = (train_samples, 1)
test_shape = (test_samples, 1)

for i in range(0, n_features):

    bias_train = np.ones(train_shape)
    bias_test = np.ones(test_shape)

    if len(feature_selection) == 0:
        X_feat_train = bias_train
        X_feat_test = bias_test

    else:

        for old_feat in feature_selection:

            old_feat_train = np.reshape(X_train[:,old_feat], train_shape)
            old_feat_test = np.reshape(X_test[:,old_feat], test_shape)
            X_feat_train = np.hstack((bias_train, old_feat_train))
            X_feat_test = np.hstack((bias_test, old_feat_test))


    for new_feat in range(0, n_features):
        if new_feat not in feature_selection:

            new_feat_train = np.reshape(X_train[:,new_feat], train_shape)
            new_feat_test = np.reshape(X_test[:,new_feat], test_shape)

            feature_train = np.hstack((X_feat_train, new_feat_train))
            feature_test = np.hstack((X_feat_test, new_feat_test))

            errors[new_feat] = evaluate_feature_error(feature_train, feature_tes


    min_error = min([i for i in errors if i is not None])
    selected_feature = errors.index(min_error)
   # print(selected_feature)
    feature_selection.append(selected_feature)
    best_scores.append(min_error)
    errors = [None for i in range(0,n_features)] # reset the errors


print("The order of the features is:")
print(feature_selection)

#print(len(feature_selection) == len(set(feature_selection)))
```

```
The order of the features is:
[138, 79, 108, 116, 166, 54, 163, 89, 194, 35, 113, 131, 130, 74, 160, 154, 95,
118, 90, 103, 167, 32, 180, 26, 36, 2, 68, 162, 135, 97, 91, 52, 13, 187, 44, 19
2, 179, 117, 188, 144, 119, 37, 47, 149, 29, 127, 101, 7, 28, 139, 141, 43, 94,
```

```
146, 189, 92, 198, 129, 31, 159, 60, 99, 122, 51, 12, 110, 62, 45, 140, 64, 121,
61, 88, 115, 169, 14, 148, 3, 65, 145, 5, 120, 58, 55, 105, 171, 137, 0, 151, 18
1, 83, 70, 33, 143, 16, 184, 104, 4, 77, 8, 190, 153, 107, 63, 40, 170, 81, 19,
30, 75, 132, 53, 193, 158, 9, 123, 172, 136, 174, 157, 165, 84, 176, 69, 10, 10
0, 196, 59, 133, 18, 82, 183, 6, 76, 22, 98, 109, 1, 24, 186, 87, 93, 102, 71, 7
3, 199, 48, 85, 39, 86, 38, 17, 66, 57, 126, 155, 164, 168, 80, 177, 25, 152, 2
1, 161, 124, 134, 106, 156, 46, 128, 150, 147, 111, 27, 41, 175, 185, 112, 11, 4
2, 23, 49, 182, 56, 34, 125, 197, 195, 78, 67, 191, 173, 114, 20, 142, 50, 72, 1
78, 15, 96]
```

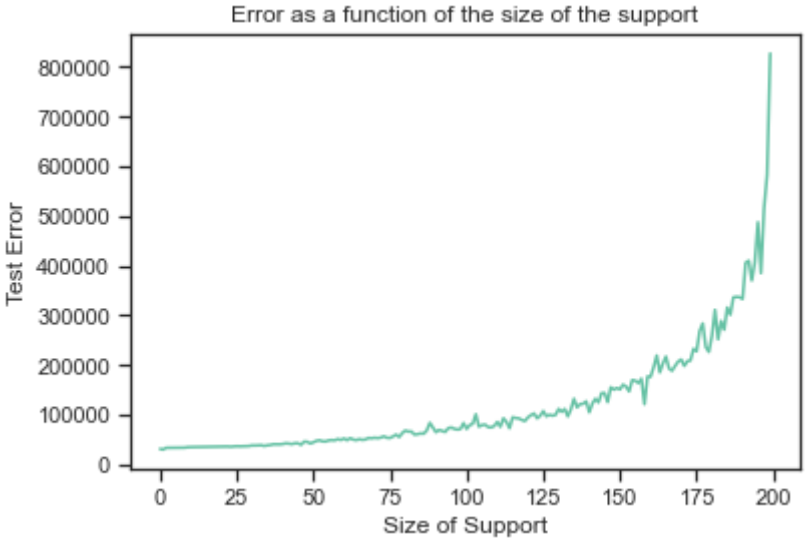
In [6]:

```
#2.2
#Plot test error as a function of the size of the support.  Can you use this to

txt = plt.plot(best_scores)
txt = plt.xlabel("Size of Support")
txt = plt.ylabel("Test Error")
txt = plt.title("Error as a function of the size of the support")

print("We can use this to recover the true support. We need to find the feature
print("Index " + str(best_scores.index(min(best_scores))) +  " error = " + str(m
```

```
We can use this to recover the true support. We need to find the feature with th
e minimal test-error
Index 1 error = 30052.233103646933
```



In [7]:

```
# 2.3
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import classification_report
from sklearn.linear_model import Lasso

alphas = np.logspace(-4, -0.5, 30)
lasso = Lasso(random_state=0, max_iter=10000)

tuned_parameters = [{'alpha': alphas}]
n_folds = 5

clf = GridSearchCV(lasso, tuned_parameters, cv=n_folds, scoring='r2')

clf.fit(X_train, y_train)

print("Best Alpha:", clf.best_params_)
means = clf.cv_results_['mean_test_score']
stds = clf.cv_results_['std_test_score']
```

```
for mean, std, params in zip(means, stds, clf.cv_results_['params']):
        print("%0.3f (+/-%0.03f) for %r"
              % (mean, std * 2, params))
```

```
Best Alpha: {'alpha': 0.0028072162039411755}
-2.903 (+/-5.040) for {'alpha': 0.0001}
-2.903 (+/-5.039) for {'alpha': 0.00013203517797162948}
-2.903 (+/-5.039) for {'alpha': 0.00017433288221999874}
-2.855 (+/-4.990) for {'alpha': 0.00023018073130224678}
-2.563 (+/-4.171) for {'alpha': 0.0003039195382313198}
-2.222 (+/-3.186) for {'alpha': 0.0004012807031942776}
-1.367 (+/-3.581) for {'alpha': 0.0005298316906283707}
-0.753 (+/-2.872) for {'alpha': 0.0006995642156712634}
-0.691 (+/-2.747) for {'alpha': 0.0009236708571873865}
-0.455 (+/-1.938) for {'alpha': 0.0012195704601594415}
-0.138 (+/-0.999) for {'alpha': 0.0016102620275609393}
-0.127 (+/-1.021) for {'alpha': 0.0021261123338996556}
-0.114 (+/-0.894) for {'alpha': 0.0028072162039411755}
-0.168 (+/-0.914) for {'alpha': 0.0037065129109221566}
-0.292 (+/-1.062) for {'alpha': 0.004893900918477494}
-0.286 (+/-1.009) for {'alpha': 0.006461670787466976}
-0.407 (+/-0.931) for {'alpha': 0.008531678524172814}
-0.639 (+/-1.343) for {'alpha': 0.011264816923358867}
-0.644 (+/-1.348) for {'alpha': 0.014873521072935119}
-1.106 (+/-2.020) for {'alpha': 0.0196382800192977}
-1.152 (+/-1.992) for {'alpha': 0.02592943797404667}
-1.162 (+/-1.986) for {'alpha': 0.03423597957607583}
-1.211 (+/-2.002) for {'alpha': 0.04520353656360245}
-1.234 (+/-2.021) for {'alpha': 0.05968456995122311}
-1.248 (+/-2.078) for {'alpha': 0.07880462815669913}
-1.260 (+/-2.134) for {'alpha': 0.10404983103657853}
-1.252 (+/-2.128) for {'alpha': 0.1373823795883264}
-1.240 (+/-2.121) for {'alpha': 0.1813930693911063}
-1.230 (+/-2.114) for {'alpha': 0.2395026619987486}
-1.211 (+/-2.101) for {'alpha': 0.31622776601683794}
```

In [14]:

```
# 2.4

best_alphas = []
for n_fold in range(3,12):
    print("Number of folds: ", n_fold)
    clf = GridSearchCV(lasso, tuned_parameters, cv=n_fold, scoring='r2')
    clf.fit(X_train, y_train)
    print("Best Alpha:", clf.best_params_)
    best_alphas.append(clf.best_params_['alpha'])

print()
print("As the number of folds increases, the hyper parameter increases till it p
```

```
Number of folds:  3
Best Alpha: {'alpha': 0.0021261123338996556}
Number of folds:  4
Best Alpha: {'alpha': 0.0021261123338996556}
Number of folds:  5
Best Alpha: {'alpha': 0.0028072162039411755}
Number of folds:  6
Best Alpha: {'alpha': 0.0021261123338996556}
Number of folds:  7
Best Alpha: {'alpha': 0.008531678524172814}
Number of folds:  8
Best Alpha: {'alpha': 0.31622776601683794}
```

```
Number of folds:  9
Best Alpha: {'alpha': 0.0009236708571873865}
Number of folds:  10
Best Alpha: {'alpha': 0.0009236708571873865}
Number of folds:  11
Best Alpha: {'alpha': 0.0009236708571873865}

As the number of folds increases, the hyper parameter increases till it peaked a
t 8 folds, then it went started to decrease.
```

2.5

LassoCV will choose a set of alphas and perform comparison searches over then, using cross-validation to generate lasso models and internally score the results. The previous step used GridsearchCV + Lasso model, which is essentially performing the same functions as LassoCV. The results from LassoCV and GridsearchCV + Lasso should agree with each other if the parameters were configured the same way.

In [ ]:

In [8]:
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import datasets, linear_model
from sklearn.preprocessing import MinMaxScaler
from sklearn.svm import SVR
from sklearn.model_selection import KFold
import pandas as pd
from pandas import DataFrame, Series
import seaborn as sns
sns.set(style='ticks', palette='Set2')
%matplotlib inline
```

In [10]:
```
#read_file = pd.read_csv ('housing.data.txt', header = None, delim_whitespace= T
#read_file.to_csv ('housing_data.csv', index=None)
```

In [11]:
```
housing = pd.read_csv('housing_data.csv')
```

In [12]:
```
X = housing.iloc[:, [0, 12]]
y = housing.iloc[:, 13]
```

In [13]:
```
scaler = MinMaxScaler(feature_range=(0, 1))
X = scaler.fit_transform(X)
```

In [14]:
```
bias = []
variance_total = []
```

In [15]:
```
mean_score = []
variance = []
best_svr = SVR(kernel='rbf')
for i in np.arange(1,100):
    scores = []
    cv = KFold(5, shuffle= True)
    for train_index, test_index in cv.split(X):
        X_train, X_test, y_train, y_test = X[train_index], X[test_index], y[trai
        best_svr.fit(X_train, y_train)
        scores.append(best_svr.score(X_test, y_test))
    mean_score.append(np.mean(scores))
    variance.append(np.std(scores))
bias.append(np.mean(mean_score))
variance_total.append(np.std(variance))
print("50 Folds: Mean - " + str(np.mean(mean_score)) + " | Variance - " + str(np
```

```
50 Folds: Mean - 0.5756046264442204 | Variance - 0.01621214575019245
```

In [16]:
```
mean_score = []
variance = []
best_svr = SVR(kernel='rbf')
for i in np.arange(1,100):
    scores = []
    cv = KFold(10, shuffle= True)
```

```
    for train_index, test_index in cv.split(X):
        X_train, X_test, y_train, y_test = X[train_index], X[test_index], y[trai
        best_svr.fit(X_train, y_train)
        scores.append(best_svr.score(X_test, y_test))
    mean_score.append(np.mean(scores))
    variance.append(np.std(scores))
bias.append(np.mean(mean_score))
variance_total.append(np.std(variance))
print("50 Folds: Mean - " + str(np.mean(mean_score)) + " | Variance - " + str(np
```

```
50 Folds: Mean - 0.5833956249966143 | Variance - 0.017251725110677237
```

In [17]:

```
mean_score = []
variance = []
best_svr = SVR(kernel='rbf')
for i in np.arange(1,100):
    scores = []
    cv = KFold(20, shuffle= True)
    for train_index, test_index in cv.split(X):
        X_train, X_test, y_train, y_test = X[train_index], X[test_index], y[trai
        best_svr.fit(X_train, y_train)
        scores.append(best_svr.score(X_test, y_test))
    mean_score.append(np.mean(scores))
    variance.append(np.std(scores))
bias.append(np.mean(mean_score))
variance_total.append(np.std(variance))
print("50 Folds: Mean - " + str(np.mean(mean_score)) + " | Variance - " + str(np
```

```
50 Folds: Mean - 0.5824509527907801 | Variance - 0.017676315213079788
```

In [18]:

```
mean_score = []
variance = []
best_svr = SVR(kernel='rbf')
for i in np.arange(1,100):
    scores = []
    cv = KFold(50, shuffle= True)
    for train_index, test_index in cv.split(X):
        X_train, X_test, y_train, y_test = X[train_index], X[test_index], y[trai
        best_svr.fit(X_train, y_train)
        scores.append(best_svr.score(X_test, y_test))
    mean_score.append(np.mean(scores))
    variance.append(np.std(scores))
bias.append(np.mean(mean_score))
variance_total.append(np.std(variance))
print("50 Folds: Mean - " + str(np.mean(mean_score)) + " | Variance - " + str(np
```

```
50 Folds: Mean - 0.5438518638280582 | Variance - 0.07102601150329856
```

In [19]:

```
mean_score = []
variance = []
best_svr = SVR(kernel='rbf')
for i in np.arange(1,100):
    scores = []
    cv = KFold(50, shuffle= True)
    for train_index, test_index in cv.split(X):
        X_train, X_test, y_train, y_test = X[train_index], X[test_index], y[trai
        best_svr.fit(X_train, y_train)
        scores.append(best_svr.score(X_test, y_test))
    mean_score.append(np.mean(scores))
```

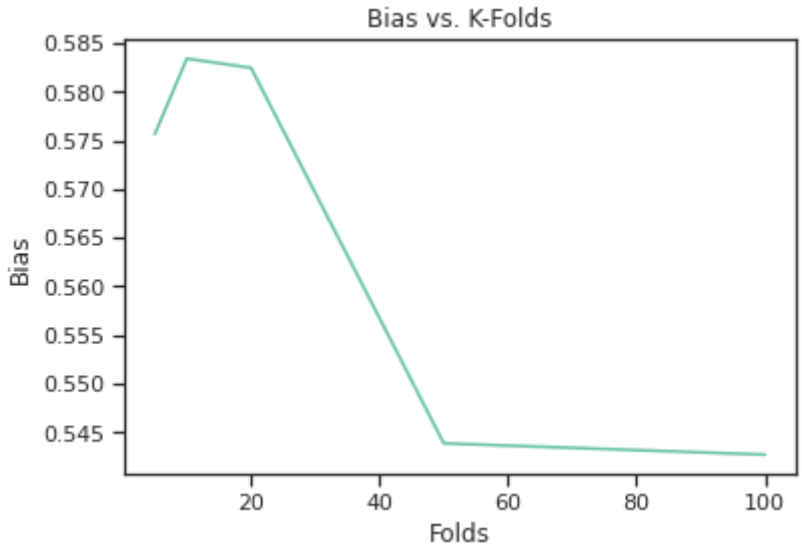
```
    variance.append(np.std(scores))
bias.append(np.mean(mean_score))
variance_total.append(np.std(variance))
print("50 Folds: Mean - " + str(np.mean(mean_score)) + " | Variance - " + str(np
```

```
50 Folds: Mean - 0.5426904073913306 | Variance - 0.2241660726495882
```

In [20]:
```
k = [5, 10, 20, 50, 100]
```

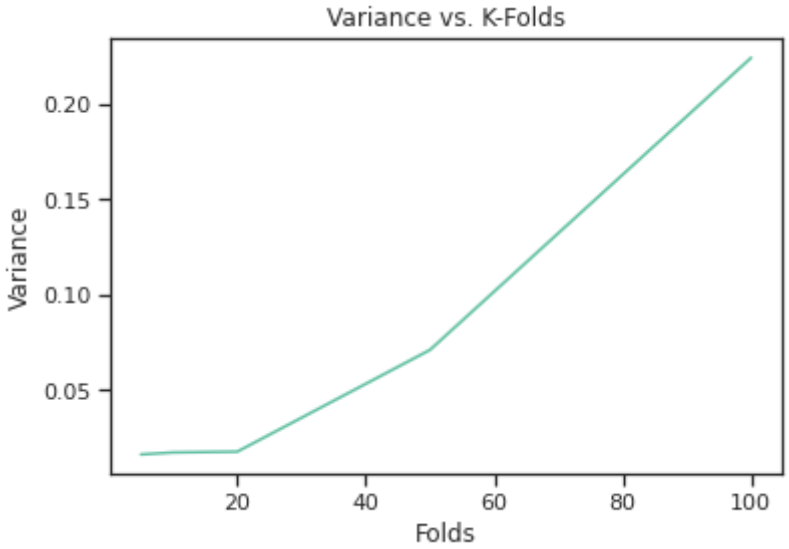
In [21]:
```
plt.plot(k, bias)
plt.title('Bias vs. K-Folds')
plt.xlabel('Folds')
plt.ylabel('Bias')
```

Out[21]:
```
Text(0, 0.5, 'Bias')
```



In [22]:
```
plt.plot(k, variance_total)
plt.title('Variance vs. K-Folds')
plt.xlabel('Folds')
plt.ylabel('Variance')
```

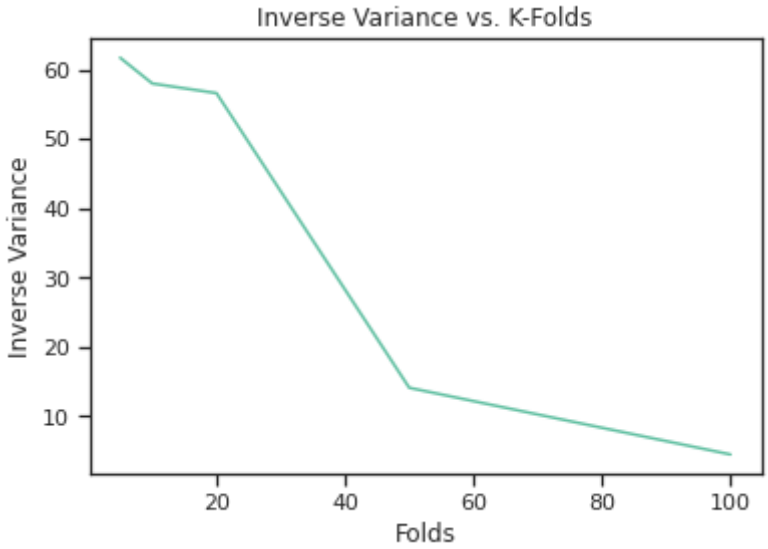
Out[22]:
```
Text(0, 0.5, 'Variance')
```



In [26]:
```
v = np.array(variance_total)
```

In [27]:
```
plt.plot(k, 1/v)
plt.title('Inverse Variance vs. K-Folds')
plt.xlabel('Folds')
plt.ylabel('Inverse Variance')
```

Out[27]: Text(0, 0.5, 'Inverse Variance')



In [ ]:
```
# Looking at this simulation, it seems as though the claim that bias increases a
# However, in this graph, at a small K, our bias inccreases as we get larger rat
# This is also sometimes true with our Variance. During some iterations, we can
```