Karthik Konath (kk28699), Kyle Polansky (kpp446)

EE 379K-DS Lab 6

Problem 1

In this problem we will use synthetic data sets to explore the bias-variance tradeoff incurred by using regularization.

Generate data of the form: $y = X\beta + \epsilon$, where X is an n×p matrix where n= 51,p= 50, and each X_{ij} ~N(0,1). Also, generate the noise according to ϵ_i ~N(0,1/4). Let β be the all ones vector (for simplicity).

By repeatedly doing this experiment and generating fresh data (fresh X, and y, and hence ϵ – but make that you're not reseting your random seed!) but keeping β fixed, you will estimatemany different solutions,ˆβ. Estimate the mean and variance of ˆβ. Note that ˆβ is a vector, so for this exercise simply estimate the variance of a single component.

In [3]:
```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split, cross_val_score, LeaveOn
eOut
from sklearn.linear_model import LinearRegression, Ridge, RidgeCV, Lasso, Lass
oCV
from sklearn.metrics import mean_squared_error
from sklearn.decomposition import PCA
from sklearn.cross_decomposition import PLSRegression

bhat = []

for i in range(1000):
    x = np.random.randn(51,50)
    e = np.asarray([np.random.normal(0,.25,51)]).transpose()
    b = np.ones(50).transpose()
    y = np.dot(x, b) + e

    fit = LinearRegression().fit(x,y)
    bhat.append(fit.coef_[0,0])

print("Mean: ", np.mean(bhat))
print("Variance: ", np.var(bhat))
```

```
Mean:  -0.3513087670878969
Variance:  11.765113309006235
```

Use ridge regression, i.e., l2 regularization. Vary the regularization coefficient λ= 0.01,0.1,1,10,100 and repeat the above experiment. What do you observe? As you increase λ is the model becoming more simple or more complex? As you increase λ is performance becoming better or worse? Also compute LOOCV for each λ. How does the value of LOOCV, and in particular how it changes as λ varies, compare with what you observe for the explicitly computed variance?

In [191]:
```
alphas = [0.01,0.1,1,10,100]

for alpha in alphas:
    x = np.random.randn(51,50)
    e = np.asarray([np.random.normal(0,.25,51)]).transpose()
    b = np.ones(50).transpose()
    y = np.dot(x, b) + e

    fit = Ridge(alpha).fit(x,y)
    mean = np.mean(fit.coef_[:,0])
    var = np.var(fit.coef_[:,0])

    y_pred = fit.predict(x)

    rmse = np.sqrt(mean_squared_error(y, y_pred))

    loo = LeaveOneOut()
    loo_score = 0
    for train_index, test_index in loo.split(x):
        x_train, x_test = x[train_index], x[test_index]
        y_train, y_test = y[train_index], y[test_index]

        fit = Ridge(alpha=alpha).fit(x_train, y_train)
        y_pred = fit.predict(x_test)

        loo_score += np.sqrt(mean_squared_error(y_test, y_pred))


    print("alpha:", alpha, "mean: ", mean, "variance:", var, "rmse:", rmse, "L
OOCV:", loo_score/loo.get_n_splits(x))
```

```
alpha: 0.01 mean:  0.04210509697234493 variance: 5.852744288755671e-29 rmse:
0.004267142701373676 LOOCV: 0.315668622323263
alpha: 0.1 mean:  -0.017421271808952584 variance: 1.180338265250124e-30 rmse:
0.008554739735711426 LOOCV: 0.18830953475685752
alpha: 1 mean:  -0.13863892935389355 variance: 2.8926712498051423e-32 rmse:
0.05453718819459226 LOOCV: 0.3892386871092322
alpha: 10 mean:  0.02910249644399795 variance: 2.2794891425112652e-33 rmse:
0.11255486349382784 LOOCV: 0.24406365386326734
alpha: 100 mean:  -0.0009539587031071637 variance: 1.8616304033972567e-34 rms
e: 0.20511361219723612 LOOCV: 0.22296113024832562
```

Read about the Bootstrap, and try to use it to compute the variance (as above), but with asingle copy of the data, rather than with many fresh copies of the data.

```
In [106]: import bootstrapped.bootstrap as bs
          import bootstrapped.stats_functions as bs_stats

          x = np.random.randn(51,50)
          e = np.asarray([np.random.normal(0,.25,51)]).transpose()
          b = np.ones(50).transpose()
          y = np.dot(x, b) + e

          samples = y.flatten();

          print("Population Mean:", np.mean(samples))
          print("Population Variance:", np.var(samples))

          bs_mean = bs._bootstrap_sim(samples, bs_stats.mean, None, 10, 100, None)
          bs_std = bs._bootstrap_sim(samples, bs_stats.std, None, 10, 100, None)

          print("Bootstrap Mean:", np.mean(bs_mean))
          print("Bootstrap Variance:", np.mean(np.square(bs_std)))
```

```
Population Mean: -1.0964582685772777
Population Variance: 65.7236986872865
Bootstrap Mean: -1.0993340223690444
Bootstrap Variance: 66.30754449305515
```

Problem 2. Problem 9 from Chapter 6.

(Predicting the number of applications in College) Note that you will have to read about PCR(Principal Components Regression) and PLS (Partial Least Squares) in the book, since we did notdiscuss these in class.

In this exercise, we will predict the number of application

(a) Split the data set into a training set and a test set.

```
In [86]: df = pd.read_csv("College.csv", header = 0, index_col = 0)
         df.head()

         df.replace({"Yes" : 1, "No" : 0}, inplace = True)
         x = df.drop(["Apps"], axis = 1)
         y = df.loc[:,"Apps"]

         x_train, x_test, y_train, y_test = train_test_split(x, y, test_size = 0.3)
```

(b) Fit a linear model using least squares on the training set, and report the test error obtained.

```
In [87]: fit = LinearRegression().fit(x_train, y_train)
         y_pred = fit.predict(x_test)

         print(np.sqrt(mean_squared_error(y_test, y_pred)))
```

```
1010.6636061663704
```

(c) Fit a ridge regression model on the training set, with λ chosen by cross-validation. Report the test error obtained.

```
In [88]: alphas = np.arange(0.001,10,.01)
         fit = RidgeCV(alphas = alphas).fit(x_train, y_train)
         y_pred = fit.predict(x_test)

         print(np.sqrt(mean_squared_error(y_test, y_pred)))
```

```
1012.8559499061809
```

(d) Fit a lasso model on the training set, with λ chosen by crossvalidation. Report the test error obtained, along with the number of non-zero coefficient estimates.

```
In [89]: fit = LassoCV(alphas = alphas).fit(x_train, y_train)
         y_pred = fit.predict(x_test)

         print(np.sqrt(mean_squared_error(y_test, y_pred)))

         print(sum(fit.coef_ > 0.1) + sum(fit.coef_ < -0.1)) #coefficients less than .1
          considered 0.
```

```
1014.3092155895241
12
```

(e) Fit a PCR model on the training set, with M chosen by crossvalidation. Report the test error obtained, along with the value of M selected by cross-validation.

In [90]:
```
total_variance = 0

for i in range(1,18):
    pca = PCA(n_components = i, svd_solver = "full")
    score = -1 * cross_val_score(pca, x, y, cv = 5).mean()
    pca  = pca.fit(x_train, y_train)
    if total_variance < .9:
        total_variance += pca.explained_variance_ratio_[-1]
        m = i

fit = LinearRegression().fit(x_train.iloc[:,:m+1] , y_train)
x_test_pcr = x_test.iloc[:,:m+1]
y_pred = pcr.predict(x_test_pcr)

print(np.sqrt(mean_squared_error(y_test, y_pred)))
print(m)
```

```
1046.6227708719919
3
```

(f) Fit a PLS model on the training set, with M chosen by crossvalidation. Report the test error obtained, along with the value of M selected by cross-validation.

In [91]:
```
m = 0
best_score = 0

for i in range(1,18):
    score = -1 * cross_val_score(PLSRegression(n_components = i), x, y, cv = 5
).mean()
    if score < best_score:
        best_score = score
        m = i

fit = PLSRegression(n_components = m).fit(x_train, y_train)
y_pred = fit.predict(x_test)

print(np.sqrt(mean_squared_error(y_test, y_pred)))
print(m)
```

```
3028.1645714290194
1
```

(g) Comment on the results obtained. How accurately can we predict the number of college applications received? Is there much difference among the test errors resulting from these five approaches?

All of the models are roughly similar, with the exception of PLS which is significantly worse than the other models. In our case, the linear model has the lest rmse value, very closely followed by ridge and lasso. Using the linear model, college applications received can be estimated with a RMSE of roughly 1000.

Problem 3. Problem 11 from Chapter 6.

(Predicting crime in Boston)

We will now try to predict per capita crime rate in the Boston data set.

(a) Try out some of the regression methods explored in this chapter, such as best subset selection, the lasso, ridge regression, and PCR. Present and discuss results for the approaches that you consider.

In [129]:
```
import pandas as pd
import numpy as np
import itertools
import time
import statsmodels.api as sm
def processSubset(feature_set):
    model = sm.OLS(y.loc[:2*len(X)/3],X.loc[:2*len(X)/3][list(feature_set)])
    regr = model.fit()
    RSS = ((regr.predict(X.loc[2*len(X)/3:][list(feature_set)]) - y.loc[2*len(
X)/3:]) ** 2).sum()
    return {"model":regr, "RSS":RSS}

def getBest(k):
    tic = time.time()
    results = []
    for combo in itertools.combinations(X.columns, k):
        results.append(processSubset(combo)) # Wrap everything up in a nice da
taframe
    models = pd.DataFrame(results)
    # Choose the model with the best residual sum of squares
    best_model = models.loc[models['RSS'].argmin()]
    toc = time.time()
    print("Processed ", models.shape[0], "models on", k, "predictors in", (toc
-tic), "seconds.")
    return best_model

models = pd.DataFrame(columns=["RSS", "model"])
df = pd.read_csv("Boston.csv")
y = df.crim
X = df.drop(['crim'],axis=1)
tic = time.time()
print y.loc[:len(X)/2].shape
for i in range(1,13):
    models.loc[i] = getBest(i)
toc = time.time()
print("Total elapsed time:", (toc-tic), "seconds.")
```

```
(254,)
('Processed ', 13, 'models on', 1, 'predictors in', 0.03861689567565918, 'sec
onds.')
('Processed ', 78, 'models on', 2, 'predictors in', 0.20980405807495117, 'sec
onds.')
('Processed ', 286, 'models on', 3, 'predictors in', 0.8280420303344727, 'sec
onds.')
('Processed ', 715, 'models on', 4, 'predictors in', 1.910106897354126, 'seco
nds.')
('Processed ', 1287, 'models on', 5, 'predictors in', 3.3881428241729736, 'se
conds.')
('Processed ', 1716, 'models on', 6, 'predictors in', 4.733531951904297, 'sec
onds.')
('Processed ', 1716, 'models on', 7, 'predictors in', 4.752110958099365, 'sec
onds.')
('Processed ', 1287, 'models on', 8, 'predictors in', 3.605592966079712, 'sec
onds.')
('Processed ', 715, 'models on', 9, 'predictors in', 2.0500900745391846, 'sec
onds.')
('Processed ', 286, 'models on', 10, 'predictors in', 0.8250260353088379, 'se
conds.')
('Processed ', 78, 'models on', 11, 'predictors in', 0.23470401763916016, 'se
conds.')
('Processed ', 13, 'models on', 12, 'predictors in', 0.04016613960266113, 'se
conds.')
('Total elapsed time:', 22.650835037231445, 'seconds.')
```

```
In [130]: print "AIC: \n", models.apply(lambda row: row[1].aic, axis=1)
          print "BIC: \n", models.apply(lambda row: row[1].bic, axis=1)
          print "R^2: \n", models.apply(lambda row: row[1].rsquared, axis=1)
          print "RSS: \n",models["RSS"]
```

```
AIC:
1     665.984995
2     631.896265
3     633.615819
4     632.771619
5     633.438532
6     555.643732
7     556.293212
8     500.300291
9     473.286494
10    461.542683
11    451.360659
12    440.798575
dtype: float64
BIC:
1     669.808041
2     639.542357
3     645.084957
4     648.063803
5     652.553761
6     578.582007
7     583.054533
8     530.884658
9     507.693907
10    499.773142
11    493.414164
12    486.675126
dtype: float64
R^2:
1     0.296980
2     0.368174
3     0.368698
4     0.373988
5     0.376452
6     0.507573
7     0.509537
8     0.586865
9     0.620849
10    0.635957
11    0.648844
12    0.661656
dtype: float64
RSS:
1     37268.864142
2     33220.789371
3     33366.549965
4     33691.621669
5     34409.081874
6     34398.436365
7     34538.955333
8     34643.154583
9     34952.607655
10    35686.296590
11    36041.248575
12    36432.839014
Name: RSS, dtype: float64
```

Based on R^2 and RSS, you generally get continuously better results as you increase predictor count. However, attempting to minimize AIC and BIC cause us to want to choose a smaller predictor count in order to reduce potential for overfitting among other concerns. In this case, however, AIC and BIC continiously decrease across the 12 features we have, so we feel comfortable using all 12 without too significant of worries about overfitting.

In [134]:
```
print models.loc[12,"model"].params
print models.loc[12,"model"].rsquared
```

```
zn        -0.001278
indus      0.029043
chas       0.044097
rm         0.166329
age        0.002430
dis       -0.005937
rad        0.023421
tax        0.001935
ptratio   -0.047123
black     -0.003340
lstat      0.030302
medv       0.002556
dtype: float64
0.661655563609
```

In [135]:
```
from sklearn.linear_model import RidgeCV
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
X_std = scaler.fit_transform(X)
ridge_cv = RidgeCV(alphas=[0.001, 0.01, 0.1, 1.0, 5, 7, 8.5, 8.7, 8.9, 8.995,
9, 9.25, 9.3,  9.5, 9.7, 10.0, 100, 1000])
model_cv = ridge_cv.fit(X_std, y)
print model_cv.coef_
print ridge_cv.score(X_std,y)
```

```
[ 0.92827304 -0.53382935 -0.1856728  -0.97065932  0.28088454  0.03738141
 -1.84895149  4.49652911 -0.06958163 -0.4725579  -0.72316063  0.96750015
 -1.60331888]
0.453056847934
```

In [136]:
```
from sklearn.linear_model import LassoCV
lasso_cv = LassoCV(alphas=[0.001, 0.01, 0.1, 1.0, 5, 7, 8.5, 8.7, 8.9, 8.995,
9, 9.25, 9.3,  9.5, 9.7, 10.0, 100, 1000])
model_lcv = lasso_cv.fit(X_std,y)
print model_lcv.coef_
print lasso_cv.score(X_std,y)
```

```
[ 1.04074183 -0.43916991 -0.18914335 -1.18580599  0.29948192  0.03806506
 -2.06908831  5.10216051 -0.62159656 -0.58270034 -0.68739465  0.90092437
 -1.81984219]
0.454009513354
```

```
In [137]: from sklearn import model_selection
          from sklearn.decomposition import PCA
          from sklearn.linear_model import LinearRegression
          pca = PCA()
          regr = LinearRegression()
          X_r = pca.fit_transform(X_std)
          kf = model_selection.KFold(n_splits=10, shuffle=True, random_state=42)
          r2d2 = []
          score = model_selection.cross_val_score(regr, np.ones((len(X_r),1)), y.ravel
          (), cv=kf, scoring='r2').mean()
          r2d2.append(score)

          for i in np.arange(1, 20):
              score = model_selection.cross_val_score(regr, X_r[:,:i], y.ravel(), cv=k
          f, scoring='r2').mean()
              r2d2.append(score)
          print r2d2
          print max(r2d2)
```

```
[-0.015713985042872958, 0.36848743113535093, 0.36828233432338431, 0.457585429
65026994, 0.46370396184248852, 0.46209774108169527, 0.46521536003832181, 0.46
153466954177702, 0.49293600915250868, 0.49550475919938713, 0.5002682495113940
4, 0.5000246793017552, 0.50057953045400638, 0.50749067103523182, 0.5074906710
3523182, 0.50749067103523182, 0.50749067103523182, 0.50749067103523182, 0.507
49067103523182, 0.50749067103523182]
0.507490671035
```

It seems that finding the best subset or using PCR produces the best results in terms of achieving a good R^2 value without overfitting. Ridge and Lasso seem to fare pretty abysmally here.

(b) Propose a model (or set of models) that seem to perform well on this data set, and justify your answer. Make sure that you are evaluating model performance using validation set error, crossvalidation, or some other reasonable alternative, as opposed to using training error.

Using the model generated by the best subset seems to work best. Taking the validation set R^2 value of .66 in comparison to the other tested models, I feel fairly confident in saying that this is the best model of the 4 for our purposes by a good margin.

zn -0.001278 indus 0.029043 chas 0.044097 rm 0.166329 age 0.002430 dis -0.005937 rad 0.023421 tax 0.001935 ptratio -0.047123 black -0.003340 lstat 0.030302 medv 0.002556

(c) Does your chosen model involve all of the features in the data set? Why or why not?

Yes, it does. Due to the monotonically decreasing values of AIC and BIC, I feel confident that the risk of overfitting is low. The R^2 value seems to increase by a couple of percentage point for every predictor included as well, so the model definitely seems to benefit from the added features. As a result, I chose to use all the features given to create my model.

Problem 4

This is a written problem, supporting Problem 9 above. Note that a lot of this has been solved in class, but it is good for you to try to do it again without referencing the class notes.

Consider the Least Squares optimization problem, given data X and y ...

Note that x_i represents the ith row of X and hence is a row-vector. Hence $x_{i\beta}$ represents the dot product between the p-length vectors x_i and β. Derive a closed form solution (as wedid in class) for ˆβ_{LS}, by expanding out, taking the derivative and setting it equal to zero. It might be easiest to work in vector notation rather than deal with the individual x_i's.

Now consider the Ridge Regression problem ... Use the same approach as above to again derive a closed form expression for the solution, ˆβ_R.

title