ps3_problem1

May 14, 2020

0.1 IDS/ACM/CS 158: Fundamentals of Statistical Learning

0.1.1 PS3, Problem 1: The Validation Set Approach For Model Selection

Name: Li, Michael

Email address: mlli@caltech.edu

Notes: Please use python 3.6

You are required to properly comment and organize your code.

- Helper functions (add/remove part label according to the specific question requirements)

[1]:
```
import numpy as np
import numpy.random
import matplotlib.pyplot as plt
import numpy.matlib

def find_beta(data):
    """
    data - a matrix where each row corresponds to the
           p predictors in the first p columns and
           the observed output y in the final column

    returns the OLS estimate of the regression parameter
    """
    x = data[:,:-1]
    y = data[:,-1]

    # add bias term to training data
    bias = np.matlib.repmat(1, len(x), 1)
    x = np.concatenate((bias, x), axis=1)

    # calculate beta
    intermediate = np.matmul(x.transpose(), x)
    inverse_intermediate = np.linalg.inv(np.array(intermediate))
    pseudo_x = np.matmul(inverse_intermediate, x.transpose())

    return np.matmul(pseudo_x, y)
```

```
def predict(ols, data):
    """
    ols - ols estimate of the regression parameter
    data - a matrix where each row corresponds to the
           p predictors in the first p columns and
           the observed output y in the final column

    returns the predictions for the observations in data
    """
    x_with_bias_term = np.concatenate((np.matlib.repmat(1, len(data), 1), data[:
 ↪,:-1]), axis=1)
    return np.matmul(x_with_bias_term, ols)

def l2_loss(data, preds):
    """
    data - a matrix where each row corresponds to the
           p predictors in the first p columns and
           the observed output y in the final column
    preds - the predictions for the observations in data

    returns the L2 loss of the values
    """
    return np.mean((data[:,-1] - preds)**2)

def split_train_test(data, train_indices):
    """
    data - a matrix where each row corresponds to the
           p predictors in the first p columns and
           the observed output y in the final column
    train_indices - indices for the training data

    returns the train and test data split using indices
    """
    train_data = data[train_indices]
    val_data = np.delete(data, train_indices, axis=0)

    return train_data, val_data

def validation_err(data, indices):
    """
    data - a matrix where each row corresponds to the
           p predictors in the first p columns and
           the observed output y in the final column
    indices - indices for the training data

    returns the validation error of the data training on the indices
```

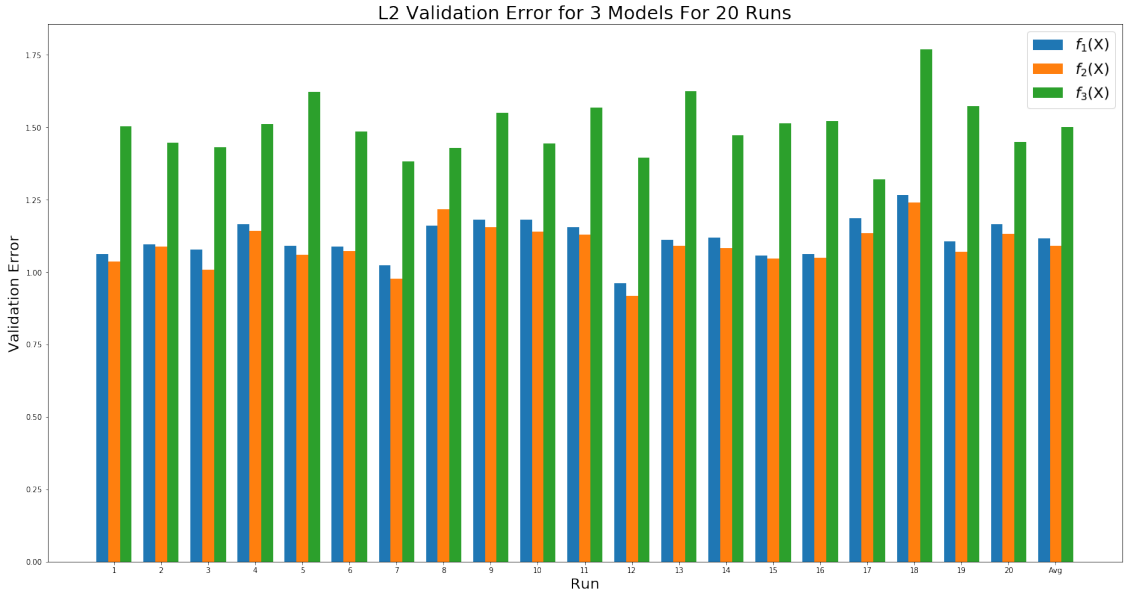
```
    """
    train, val = split_train_test(data, indices)
    reg = find_beta(train)
    preds = predict(reg, val)

    return l2_loss(val, preds)
```

[2]:
```
# reformat data so we have 3 models
f_1_data = np.genfromtxt('dataset5.csv', delimiter=',',skip_header =1)
f_2_data = np.array([[f_1_data[i][0], np.sin(f_1_data[i][1]), f_1_data[i][2]]␣
 ↪for i in range(len(f_1_data))])
f_3_data = np.delete(f_1_data, 1, axis=1)
```

[3]:
```
f_1_errs = []
f_2_errs = []
f_3_errs = []
r = 20

for _ in range(r):
    # randomly choose training indices
    train_indices = numpy.random.choice(len(f_1_data), int(len(f_1_data)/2),␣
 ↪replace=False)

    # find validation errors for each dataset
    f_1_errs.append(validation_err(f_1_data, train_indices))
    f_2_errs.append(validation_err(f_2_data, train_indices))
    f_3_errs.append(validation_err(f_3_data, train_indices))

# add the average error just to see
f_1_errs.append(np.mean(f_1_errs))
f_2_errs.append(np.mean(f_2_errs))
f_3_errs.append(np.mean(f_3_errs))
```

[4]:
```
fig = plt.figure(figsize=(20, 10))
ax = fig.add_axes([0,0,1,1])
barwidth = .25

x = np.arange(len(f_1_errs))

ax.bar(x + 0.00, f_1_errs, width=barwidth, label='$f_1$(X)')
ax.bar(x + 0.25, f_2_errs, width=barwidth, label='$f_2$(X)')
ax.bar(x + 0.50, f_3_errs, width=barwidth, label='$f_3$(X)')

ticks = [i+1 for i in range(len(f_1_errs)-1)]
ticks.append('Avg')

plt.xticks([r+barwidth for r in range(len(f_1_errs))], ticks)
```

```
plt.ylabel('Validation Error', fontsize=20)
plt.xlabel('Run', fontsize=20)
plt.legend(prop={'size': 20})
plt.title('L2 Validation Error for 3 Models For 20 Runs', fontsize=25)
plt.show()
```



It looks like from this graph that $f_2(\mathrm{X})$ is the best model. It has the lowest average validation error. Additionally, on each run it almost always has the lowest validation error.

ps3_problem2

May 14, 2020

0.1 IDS/ACM/CS 158: Fundamentals of Statistical Learning

0.1.1 PS3, Problem 2: Leave-One-Out Cross Validation For Model Selection

Name: Li, Michael

Email address: mlli@caltech.edu

Notes: Please use python 3.6

You are required to properly comment and organize your code.

- Helper functions (add/remove part label according to the specific question requirements)

[1]:
```
import numpy as np
import numpy.random
import matplotlib.pyplot as plt
import numpy.matlib

def find_beta(data):
    """
    data - a matrix where each row corresponds to the
           p predictors in the first p columns and
           the observed output y in the final column

    returns the OLS estimate of the regression parameter
    """
    x = data[:,:-1]
    y = data[:,-1]

    # add bias term to training data
    bias = np.matlib.repmat(1, len(x), 1)
    x = np.concatenate((bias, x), axis=1)

    # calculate beta
    intermediate = np.matmul(x.transpose(), x)
    inverse_intermediate = np.linalg.inv(np.array(intermediate))
    pseudo_x = np.matmul(inverse_intermediate, x.transpose())

    return np.matmul(pseudo_x, y), np.matmul(x, pseudo_x)
```

```
def predict(ols, data):
    """
    ols - ols estimate of the regression parameter
    data - a matrix where each row corresponds to the
           p predictors in the first p columns and
           the observed output y in the final column

    returns the predictions for the observations in data
    """

    x_with_bias_term = np.insert(data[:-1], 0, 1)
    return np.matmul(x_with_bias_term, ols)

def leave_one_out_cv(data):
    """
    data - a matrix where each row corresponds to the
           p predictors in the first p columns and
           the observed output y in the final column

    returns the leave one out cross validation of the data
    """
    ols, hat = find_beta(data)
    return np.mean([((data[i][-1] - predict(ols, data[i])) / (1-hat[i][i]))**2␣
 ↪for i in range(len(data))])
```

[2]:
```
# reformat data so we have 3 models
f_1_data = np.genfromtxt('dataset5.csv', delimiter=',',skip_header =1)
f_2_data = np.array([[f_1_data[i][0], np.sin(f_1_data[i][1]), f_1_data[i][2]]␣
 ↪for i in range(len(f_1_data))])
f_3_data = np.delete(f_1_data, 1, axis=1)
```

[3]:
```
# calculate the leave one out cross validation for each dataset
f_1_err = leave_one_out_cv(f_1_data)
f_2_err = leave_one_out_cv(f_2_data)
f_3_err = leave_one_out_cv(f_3_data)
```

[4]:
```
print("The Leave One Out Cross Validation for Model 1 is {}".format(f_1_err))
print("The Leave One Out Cross Validation for Model 2 is {}".format(f_2_err))
print("The Leave One Out Cross Validation for Model 3 is {}".format(f_3_err))
```

```
The Leave One Out Cross Validation for Model 1 is 1.1074945247730847
The Leave One Out Cross Validation for Model 2 is 1.0802973038999084
The Leave One Out Cross Validation for Model 3 is 1.500086491089816
```

From the leave one out cross validations, it looks like model 2 has the lowest estimated test error. Thus, I would definitively select model $f_2(\mathrm{X})$ as the best model using this metric since each model was trained and tested on the same data.

ps3_problem3

May 14, 2020

0.1 IDS/ACM/CS 158: Fundamentals of Statistical Learning

0.1.1 PS3, Problem 3: Best Subset Selection

Name: Li, Michael

Email address: mlli@caltech.edu

Notes: Please use python 3.6

You are required to properly comment and organize your code.

- Helper functions (add/remove part label according to the specific question requirements)

[1]:
```
import numpy as np
import numpy.matlib
import scipy.stats
import itertools
import matplotlib.pyplot as plt

def standardize_col(column):
    """
    column - an np array of values from a population

    returns the standardized column with mean 0 and std = 1
    """
    mean = np.mean(column)
    std = np.std(column, ddof=1)

    return (column - mean) / std

def predict(ols, data):
    """
    ols - ols estimate of the regression parameter
    data - a matrix where each row corresponds to the
           p predictors in the first p columns and
           the observed output y in the final column

    returns the predictions for the observations in data
    """
```

```
    x_with_bias_term = np.concatenate((np.matlib.repmat(1, len(data), 1), data[:
↪,:-1]), axis=1)
    return np.matmul(x_with_bias_term, ols)

def reduce_data(data, indices):
    """
    data - a matrix where each row corresponds to the
           p predictors in the first p columns and
           the observed output y in the final column
    indices - which indices to use from the data

    returns the reduced dataset containing only the predictors in indices
    """
    return np.append(data[:,indices], data[:,-1][...,None], 1)

def find_beta(data):
    """
    data - a matrix where each row corresponds to the
           p predictors in the first p columns and
           the observed output y in the final column

    returns the OLS estimate of the regression parameter
    """
    x = data[:,:-1]
    y = data[:,-1]

    # add bias term to training data
    bias = np.matlib.repmat(1, len(x), 1)
    x = np.concatenate((bias, x), axis=1)

    # calculate beta
    intermediate = np.matmul(x.transpose(), x)
    inverse_intermediate = np.linalg.inv(np.array(intermediate))
    pseudo_x = np.matmul(inverse_intermediate, x.transpose())

    return np.matmul(pseudo_x, y)

def rss(data, preds):
    """
    data - a matrix where each row corresponds to the
           p predictors in the first p columns and
           the observed output y in the final column
    preds - the predictions for the observations in data

    returns the residual sum of squares for the values
    """
    return np.sum((data[:,-1] - preds)**2)
```

```
def l2_loss(data, preds):
    """
    data - a matrix where each row corresponds to the
           p predictors in the first p columns and
           the observed output y in the final column
    preds - the predictions for the observations in data

    returns the L2 loss of the values
    """
    return np.mean((data[:,-1] - preds)**2)
```

[2]:
```
data = np.genfromtxt('prostate_cancer.csv', delimiter=',', skip_header=1)

standardized_data = data.copy()

for i in range(len(data[0])-2):
    standardized_data[:,i] = standardize_col(data[:,i])

# split the data into train and test
train_data = np.array([observation[:-1] for observation in standardized_data if␣
↪observation[-1] == 1])
test_data = np.array([observation[:-1] for observation in standardized_data if␣
↪observation[-1] == 0])
```

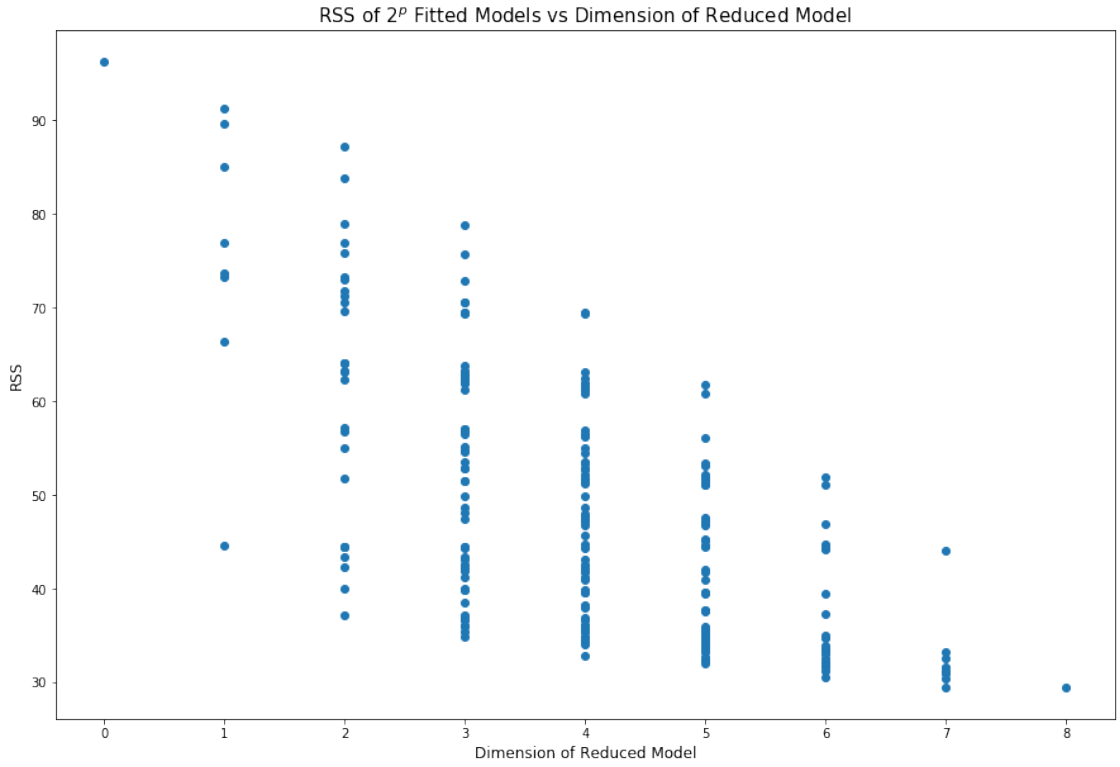
- Part A

[3]:
```
models = {
    0:[],
    1:[],
    2:[],
    3:[],
    4:[],
    5:[],
    6:[],
    7:[],
    8:[]
}
```

[4]:
```
best_models = []

for p_reduced in range(9):
    # keep track of the best model for each p_reduced (rss, indices, beta)
    p_reduced_best = (10**100, None)

    for indices in itertools.combinations(range(len(train_data[0][:-1])),␣
↪p_reduced):
```

```
        # get new dataset and calculate OLS using only indices
        reduced_data = reduce_data(train_data, indices)
        beta = find_beta(reduced_data)

        # find residuals
        preds = predict(beta, reduced_data)
        residuals = rss(reduced_data, preds)
        models[p_reduced].append(residuals)

        # update if model has lower residuals
        if residuals < p_reduced_best[0]:
            p_reduced_best = (residuals, indices, beta)

    # keep track of best model for each p_reduced
    best_models.append((p_reduced_best[1], p_reduced_best[2]))
```

[5]:
```
subset_sizes = []
residuals = []

for key in models:
    for _ in range(len(models[key])):
        subset_sizes.append(key)

for key in models:
    for r in models[key]:
        residuals.append(r)

plt.rcParams['figure.figsize'] = [15, 10]
plt.scatter(subset_sizes, residuals)
plt.xlabel('Dimension of Reduced Model', fontsize=12)
plt.ylabel('RSS', fontsize=12)
plt.title('RSS of $2^p$ Fitted Models vs Dimension of Reduced Model',␣
 ↪fontsize=15)
plt.show()
```



[6]:
```
np.array(best_models)[:,0]
```

[6]:
```
array([(), (0,), (0, 1), (0, 1, 4), (0, 1, 3, 4), (0, 1, 3, 4, 7),
       (0, 1, 3, 4, 5, 7), (0, 1, 2, 3, 4, 5, 7),
       (0, 1, 2, 3, 4, 5, 6, 7)], dtype=object)
```

Best model for $\tilde{p}$=0 includes no inputs

Best model for $\tilde{p}$=1 includes lcavol

Best model for $\tilde{p}$=2 includes lcavol and lweight

Best model for $\tilde{p}$=3 includes lcavol, lweight, and svi

Best model for $\tilde{p}$=4 includes lcavol, lweight, lbph, and svi

Best model for $\tilde{p}$=5 includes lcavol, lweight, lbph, svi, and pgg45

Best model for $\tilde{p}$=6 includes lcavol, lweight, lbph, svi, lcp, and pgg45

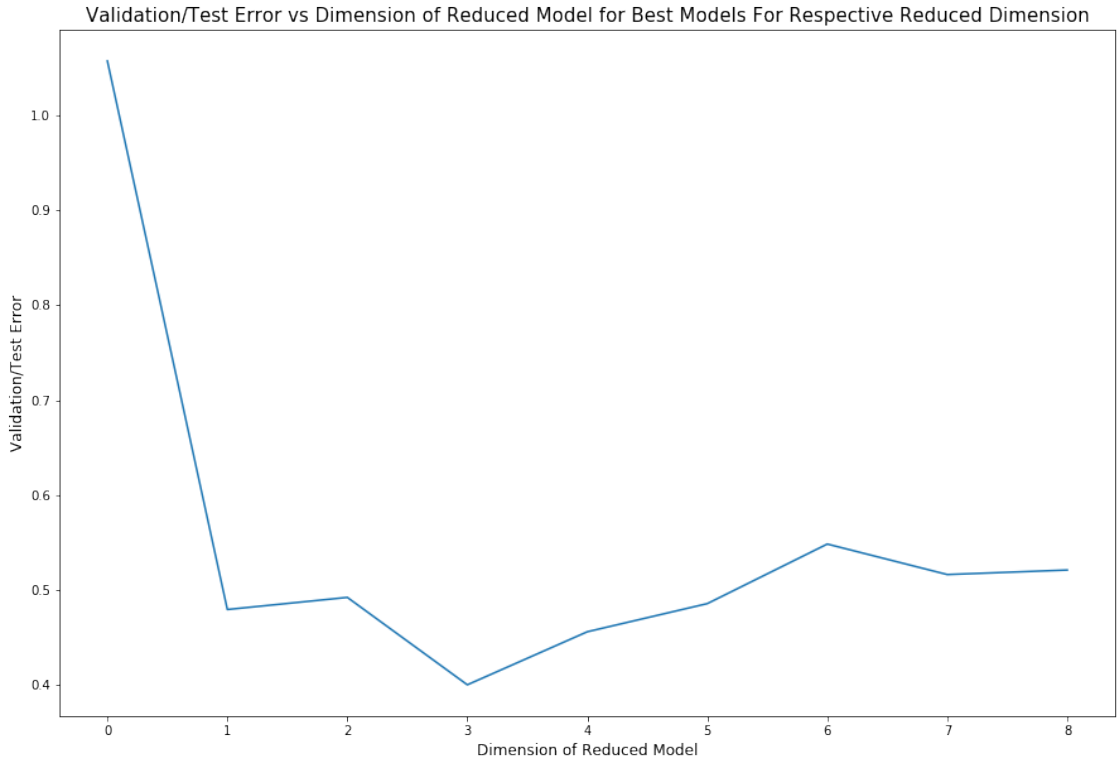
Best model for $\tilde{p}$=7 includes lcavol, lweight, age, lbph, svi, lcp, and pgg45

Best model for $\tilde{p}$=8 includes lcavol, lweight, age, lbph, svi, lcp, gleason, pgg45

[7]:
```
test_errs = []

for indices, beta in best_models:
```

```
    # calculate test error using best reduced models
    reduced_test = reduce_data(test_data, indices)
    preds = predict(beta, reduced_test)
    test_errs.append(l2_loss(reduced_test, preds))
```

[8]:
```
plt.plot(np.arange(len(train_data[0])), test_errs)
plt.xlabel('Dimension of Reduced Model', fontsize=12)
plt.ylabel('Validation/Test Error', fontsize=12)
plt.title('Validation/Test Error vs Dimension of Reduced Model for Best Models␣
 ↪For Respective Reduced Dimension', fontsize=15)
plt.show()
```



[9]:
```
best_models[3]
```

[9]:
```
((0, 1, 4), array([2.46944993, 0.61286858, 0.31565144, 0.22268916]))
```

The best final model is $f(X) = \sum_{i=0}^{p} \hat{\beta} X_i$ where $\hat{\beta}$ = [2.46944993, 0.61286858, 0.31565144, 0, 0, 0.22268916, 0, 0, 0]. Specifically, lcavol, lweight, and svi are inputs.

- Part B

[10]:
```
def kfolds(data):
    """
```

```
    data - data to split into 5 folds

    returns 5 different folds of data
    """
    np.random.shuffle(data)
    return [data[:19], data[19:38], data[38:57], data[57:77], data[77:]]

def split_folds(folds, index):
    """
    folds - list of K folds of data
    index - which of the folds to use for test data

    returns train and test of the data
    """
    test = folds[index]
    train_temp = np.delete(folds, index, axis=0)
    train = []

    for fold in train_temp:
        for row in fold:
            train.append(row)

    return np.array(train), test

def mean_and_se(data):
    """
    data - a column of data

    returns the mean of the data and standard error
    """
    mean = np.mean(data)
    se = np.sqrt(np.mean((data-mean)**2))

    return mean, se
```

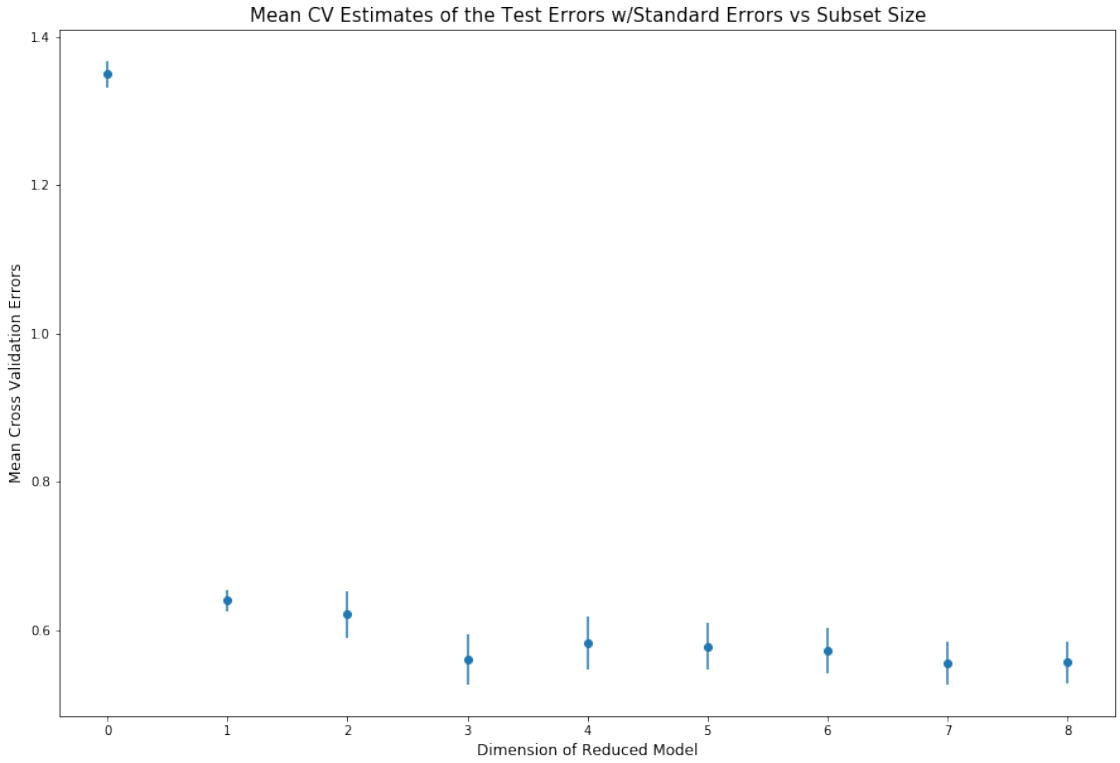
[11]:
```
r_cvs = {
    0:[],
    1:[],
    2:[],
    3:[],
    4:[],
    5:[],
    6:[],
    7:[],
    8:[]
}
```

[12]:
```
for _ in range(100):
    folds = kfolds(standardized_data[:,:-1])
    cvs = {
            0:[],
            1:[],
            2:[],
            3:[],
            4:[],
            5:[],
            6:[],
            7:[],
            8:[]
    }

    for k in range(len(folds)):
        # organize our train and test from the fold split
        train, test = split_folds(folds, k)

        for p_reduced in range((len(train[0]))):
            # keep track of the best model for each p_reduced (rss, indices,␣
↪beta)
            p_reduced_best = (10**100, None)

            for indices in itertools.combinations(range(len(train[0][:-1])),␣
↪p_reduced):
                # reduce the data and calculate the residuals
                reduced_data = reduce_data(train, indices)
                beta = find_beta(reduced_data)
                preds = predict(beta, reduced_data)
                residuals = rss(reduced_data, preds)

                # update the best model if residuals are smaller
                if residuals < p_reduced_best[0]:
                    p_reduced_best = (residuals, indices, beta)

            # retrain best model and calculate the test error
            reduced_train = reduce_data(train, p_reduced_best[1])
            reduced_test = reduce_data(test, p_reduced_best[1])
            preds = predict(p_reduced_best[2], reduced_test)
            cvs[p_reduced].append(l2_loss(reduced_test, preds))

    # keep track of cv err for each run
    for key in cvs:
        r_cvs[key].append(np.mean(cvs[key]))
```

[13]:
```
means = []
ses = []
```

```
for key in r_cvs:
    mean, se = mean_and_se(r_cvs[key])
    means.append(mean)
    ses.append(se)
```

[14]:
```
plt.errorbar(np.arange(9), means, ses, linestyle='None', marker='o')
plt.xlabel('Dimension of Reduced Model', fontsize=12)
plt.ylabel('Mean Cross Validation Errors', fontsize=12)
plt.title('Mean CV Estimates of the Test Errors w/Standard Errors vs Subset␣
 ↪Size', fontsize=15)
plt.show()
```



[15]:
```
p_reduced_min = np.argmin(means)
p_reduced_min
```

[15]: 7

[16]:
```
p_reduced_best = None

for i in range(len(means)):
    if means[i] < means[p_reduced_min] + ses[p_reduced_min]:
```

```
        p_reduced_best = i
        break
```

[17]:
```
p_reduced_best
```

[17]:
```
3
```

$\tilde{p}_{min} = 7$ $\tilde{p}^* = 3$

[18]:
```
best_model = (10**100, None)

for indices in itertools.combinations(range(len(standardized_data[0][:-2])),␣
↪p_reduced_best):
    # find all models with only p* predictors and find best one with lowest rss
    reduced_data = reduce_data(standardized_data[:,:-1], indices)
    beta = find_beta(reduced_data)
    preds = predict(beta, reduced_data)
    residuals = rss(reduced_data, preds)

    if residuals < best_model[0]:
        best_model = (residuals, indices, beta)
```

[19]:
```
best_model[1:]
```

[19]:
```
((0, 1, 4), array([2.47838688, 0.61978211, 0.28350966, 0.27558254]))
```

The best final model is $f(X) = \sum_{i=0}^{p} \hat{\beta} X_i$ where $\hat{\beta} =$ [2.47838688, 0.61978211, 0.28350966, 0, 0, 0.27558254, 0, 0, 0]. Specifically, lcavol, lweight, and svi are inputs.

ps3_problem4

May 14, 2020

0.1 IDS/ACM/CS 158: Fundamentals of Statistical Learning

0.1.1 PS3, Problem 4: Ridge Regression

Name: Li, Michael

Email address: mlli@caltech.edu

Notes: Please use python 3.6

You are required to properly comment and organize your code.

- Helper functions (add/remove part label according to the specific question requirements)

[1]:
```
import numpy as np
import numpy.matlib
import scipy.stats
import itertools
import matplotlib.pyplot as plt
from collections import defaultdict

def predict(ols, data, y_avg):
    """
    ols - ols estimate of the regression parameter
    data - a matrix where each row corresponds to the
           p predictors in the first p columns and
           the observed output y in the final column
    y_avg - average y prediction from the training data
            to add back when we predict

    returns the predictions for the observations in data
    """
    return np.matmul(data[:,:-1], ols) + y_avg

def ridge_regression(data, lamb):
    """
    data - a matrix where each row corresponds to the
           p predictors in the first p columns and
           the observed output y in the final column
    lamb - lambda to fit the ridge estimates with
```

```
    returns the ridge estimate of the regression parameter
    """
    x = data[:,:-1]
    y = data[:,-1]

    intermediate = np.matmul(x.transpose(), x)
    regularization = lamb * np.identity(len(x[0]))
    inverse_intermediate = np.linalg.inv(np.array(intermediate) +␣
↪regularization)
    pseudo_x = np.matmul(inverse_intermediate, x.transpose())

    return np.matmul(pseudo_x, y)

def l2_loss(data, preds):
    """
    data - a matrix where each row corresponds to the
           p predictors in the first p columns and
           the observed output y in the final column
    preds - the predictions for the observations in data

    returns the L2 loss of the values
    """
    return np.mean((data[:,-1] - preds)**2)

def split_folds(folds, index):
    """
    folds - list of K folds of data
    index - which of the folds to use for test data

    returns train and test of the data
    """
    test = folds[index]
    train_temp = np.delete(folds, index, axis=0)
    train = []

    for fold in train_temp:
        for row in fold:
            train.append(row)

    return np.array(train), test

def kfolds(data):
    """
    data - data to split into 5 folds

    returns 5 different folds of data
```

```
    """
    np.random.shuffle(data)
    return [data[:19], data[19:38], data[38:57], data[57:77], data[77:]]

def mean_and_se(data):
    """
    data - a column of data

    returns the mean of the data and standard error
    """
    mean = np.mean(data)
    se = np.sqrt(np.mean((data-mean)**2))

    return mean, se
```

[2]:
```
class RidgePreprocessor:
    """
    Object that keeps track of the training preprocesing step

    Initialize with data and object keeps track of
    mean of each column, standard deviations of each column, and
    the average y of the data
    """
    def __init__(self, data):
        self.means = np.mean(data[:,:-1], axis=0)
        self.stds = np.std(data[:,:-1], axis=0, ddof=1)
        self.y_avg= np.mean(data[:,-1])

    def _standardize_col(self, column, mean, std):
        """
        column - an np array of values from a population

        returns the standardized column with mean 0 and std = 1
        """
        return (column - mean) / std

    def preprocess(self, data):
        """
        given a dataset, standardize it using the saved means, stds, and y_avg
        """
        standardized_data = data.copy()

        for i in range(len(data[0])-1):
            standardized_data[:,i] = self._standardize_col(data[:,i], self.
 ↪means[i], self.stds[i])

        standardized_data[:,-1] -= self.y_avg
```

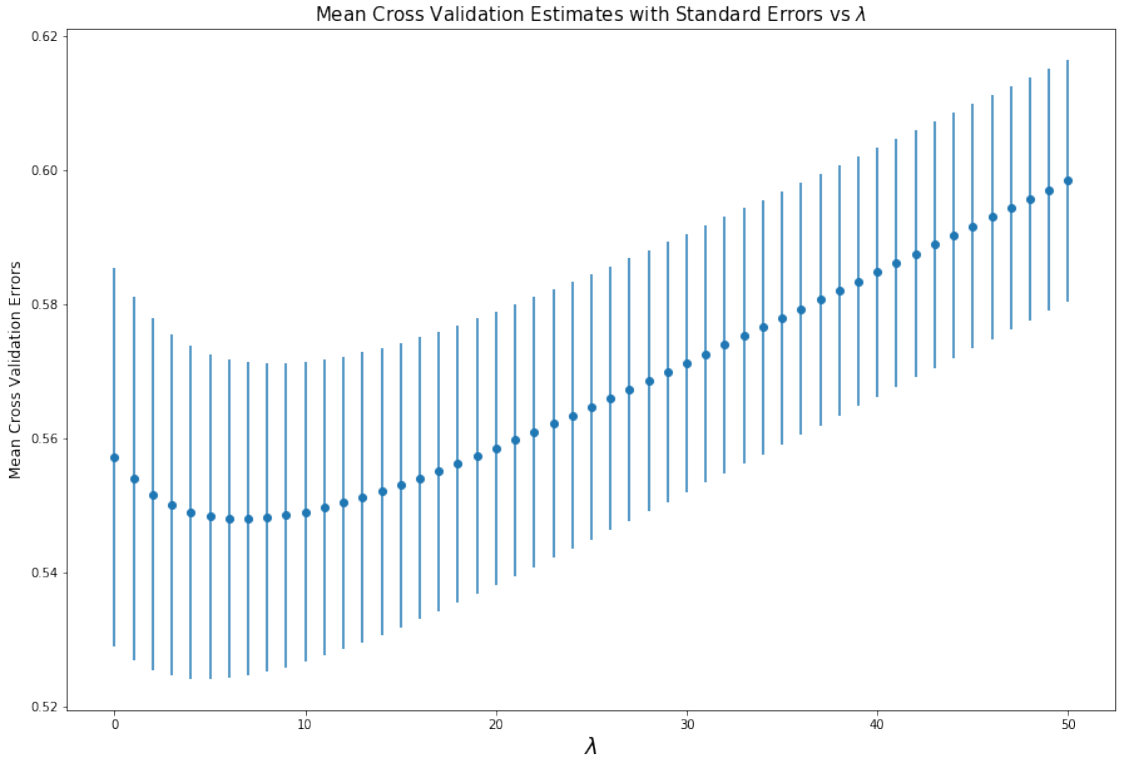
```
        return standardized_data

    def get_y_avg(self):
        """
        Getter method to get the y_avg value
        """
        return self.y_avg
```

[3]:
```
data = np.genfromtxt('prostate_cancer.csv', delimiter=',', skip_header=1)[:,:-1]
```

[24]:
```
cvs = defaultdict(list)

for _ in range(100):
    # split up our data into folds
    folds = kfolds(data)

    for lamb in range(51):
        cv_err = []

        for k in range(len(folds)):
            # organize our train and test and preprocess everything according␣
 ↪to train
            train, test = split_folds(folds, k)
            preprocessor = RidgePreprocessor(train)
            train_processed = preprocessor.preprocess(train)
            test_processed = preprocessor.preprocess(test)

            # calculate ridge estimates and calculate error for fold
            ols_ridge = ridge_regression(train_processed, lamb)
            test_preds = predict(ols_ridge, test_processed, preprocessor.
 ↪get_y_avg())
            cv_err.append(l2_loss(test, test_preds))

        # keep track of average cross validation error for 5 folds
        cvs[lamb].append(np.mean(cv_err))
```

[25]:
```
means = []
ses = []

for key in cvs:
    mean, se = mean_and_se(cvs[key])
    means.append(mean)
    ses.append(se)

plt.rcParams['figure.figsize'] = [15, 10]
```

```
plt.xlabel('$\lambda$', fontsize=18)
plt.ylabel('Mean Cross Validation Errors', fontsize=12)
plt.title('Mean Cross Validation Estimates with Standard Errors vs $\lambda$',␣
 ↪fontsize=15)
plt.errorbar(np.arange(51), means, ses, linestyle='None', marker='o')
plt.show()
```



[26]:
```
lamb_min = np.argmin(means)
lamb_min
```

[26]: 7

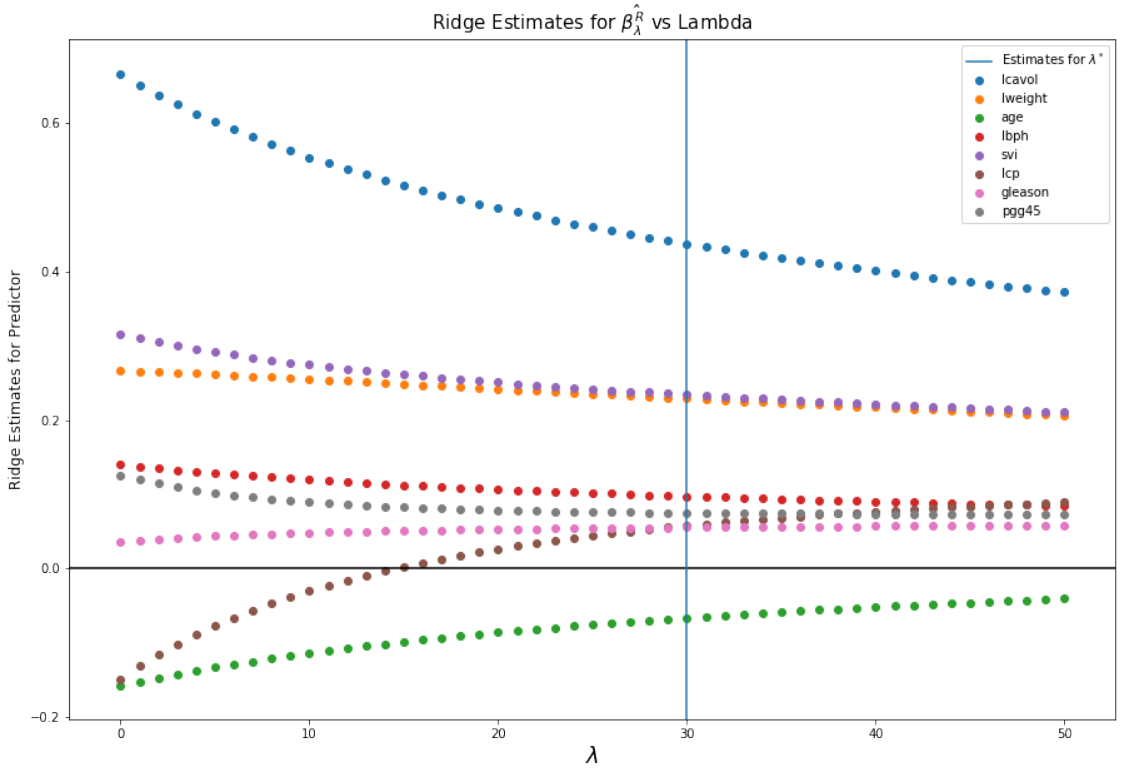
[27]:
```
lamb_best = None

for i in range(len(means)-1, -1, -1):
    if means[i] < means[lamb_min] + ses[lamb_min]:
        lamb_best = i
        break

lamb_best
```

[27]: 30

[41]:
```
data = np.genfromtxt('prostate_cancer.csv', delimiter=',', skip_header=1)[:,:-1]
preprocessor = RidgePreprocessor(data)
data = preprocessor.preprocess(data)
```

[42]:
```
betas = []

for lamb in range(51):
    betas.append(ridge_regression(data, lamb))
```

[43]:
```
labels = ["lcavol", "lweight", "age", "lbph", "svi", "lcp", "gleason", "pgg45"]
plt.rcParams['figure.figsize'] = [15, 10]
for i in range(len(labels)):
    plt.scatter(np.arange(51), np.array(betas)[:,i], label=labels[i])

plt.axhline(y=0, color='k')
plt.axvline(lamb_best, label='Estimates for $\lambda^*$')
plt.legend()
plt.xlabel('$\lambda$', fontsize=18)
plt.ylabel('Ridge Estimates for Predictor', fontsize=12)
plt.title(r'Ridge Estimates for $\hat{\beta_\lambda^R}$ vs Lambda', fontsize=15)
plt.show()
```



[44]:
```
betas[lamb_best], preprocessor.get_y_avg()
```

[44]:
```
(array([ 0.43733146,  0.2288155 , -0.0662534 ,  0.09746244,  0.23424639,
         0.05798012,  0.0554223 ,  0.07499681]),
 2.478386878350515)
```

The best final model is $f(X) = \bar{y} + \sum_{i=1}^{p} \beta^{\hat{R}}_{i,\lambda^*} X_i$ where $\beta^{\hat{R}}_{i,\lambda^*} = [\,0.43733146, 0.2288155\ , -0.0662534\ , 0.09746244, 0.23424639, 0.05798012, 0.0554223\ , 0.07499681]$ and $\bar{y} = 2.478$

5. a. $\tilde{X} = \begin{bmatrix} X \\ \sqrt{\lambda} I_p \end{bmatrix} \qquad \tilde{y} = \begin{bmatrix} y \\ 0_{p\times 1} \end{bmatrix}$

$$\hat{\tilde{\beta}}_{OLS} = (\tilde{X}^T \tilde{X})^{-1} \tilde{X}^T \tilde{y}$$

$$\tilde{X}^T \tilde{X} = \begin{bmatrix} X^T & \sqrt{\lambda} I_p \end{bmatrix} \begin{bmatrix} X \\ \sqrt{\lambda} I_p \end{bmatrix} = X^T X + \lambda I_p$$

$$\tilde{X}^T \tilde{y} = \begin{bmatrix} X^T & \sqrt{\lambda} I_p \end{bmatrix} \begin{bmatrix} y \\ 0_{p\times 1} \end{bmatrix} = X^T y + 0 = X^T y$$

$$\boxed{\hat{\tilde{\beta}}_{OLS} = (X^T X + \lambda I_p)^{-1} X^T y = \hat{\beta}^R}$$

b. $\hat{\beta}^R = \tilde{Q}\left(\tilde{\Sigma}^T\tilde{\Sigma} + \lambda I_p\right)^{-1}\tilde{\Sigma}^T\tilde{P}^T y$

$\|\hat{\beta}^R\|_2 = \hat{\beta}^{R^T}\hat{\beta}^R$ ($\tilde{\Sigma}$ is just a diagonal matrix so transpose is just itself)

$= y^T\tilde{P}\tilde{\Sigma}\left(\tilde{\Sigma}^T\tilde{\Sigma}+\lambda I_p\right)^{-T}\tilde{Q}^T\tilde{Q}\left(\tilde{\Sigma}^T\tilde{\Sigma}+\lambda I_p\right)^{-1}\tilde{\Sigma}^T\tilde{P}^T y$ ($\tilde{Q}^T\tilde{Q} = I_p$)

$= y^T\tilde{P}\underbrace{\tilde{\Sigma}\left(\tilde{\Sigma}^T\tilde{\Sigma}+\lambda I_p\right)^{-2}\tilde{\Sigma}^T}_{r\times r \text{ diagonal matrix}}\tilde{P}^T y$

$(\Sigma^2+\lambda I_r)^{-2}$

$\begin{bmatrix}\Sigma & 0\\ 0 & 0\end{bmatrix}_{N-r}^{r} \quad \begin{bmatrix}\quad & 0\\ 0 & \frac{1}{\lambda^2}I_{p-r}\end{bmatrix}_{N-r}^{r} \quad \begin{bmatrix}\Sigma & 0\\ 0 & 0\end{bmatrix}_{N-r}^{r} = \Sigma\left(\Sigma^2+\lambda I_p\right)^{-2}\Sigma$

(columns: r, $p-r$)

diagonal matrix with (j,j) element $\frac{\sigma_j^2}{(\sigma_j^2+\lambda)^2}$

$$\|\hat{\beta}^R\|_2 = \sum_{j=1}^{r}\frac{\left(\tilde{P}^T y\right)_j^2\,\sigma_j^2}{(\sigma_j^2+\lambda)^2}$$

each term is proportional to $\frac{\sigma_j^2}{(\sigma_j^2+\lambda)^2}$ thus as λ increases we know each term is decreasing since a larger λ means a larger denominator so similarly as $\lambda\to\infty$ then each term goes to 0 thus $\|\hat{\beta}^R\|_2 \to 0$