ps4_problem1

May 27, 2020

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

0.1.1 PS4, Problem 1: The LASSO

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
from sklearn.linear_model import Lasso
import warnings
warnings.filterwarnings('ignore')

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 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 LassoPreprocessor:
    """
```

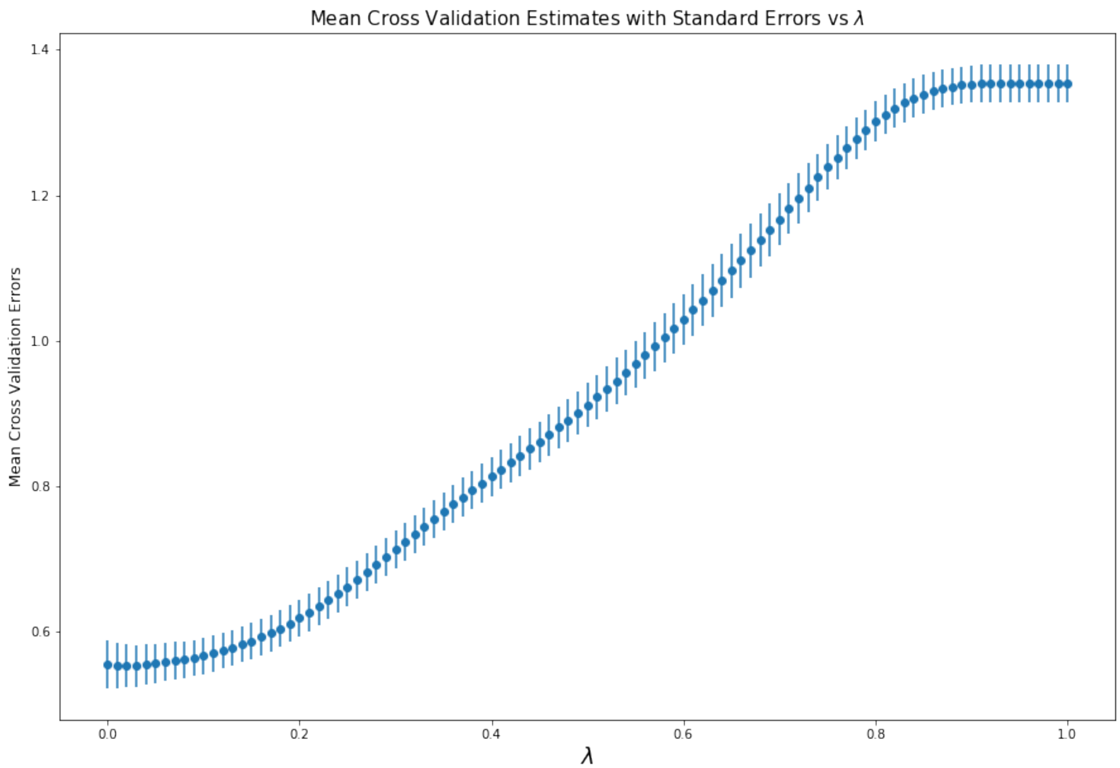
```
    Object that keeps track of the training preprocesing step

    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]
```

[4]:
```
cvs = defaultdict(list)

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

```
    for lamb in np.linspace(0, 1, 101, endpoint=True):
        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 = LassoPreprocessor(train)
            train_processed = preprocessor.preprocess(train)
            test_processed = preprocessor.preprocess(test)

            # calculate lasso estimates and calculate error for fold
            clf = Lasso(alpha=lamb, fit_intercept=False, normalize=False,␣
↪max_iter=10000)
            clf.fit(train_processed[:,:-1], train_processed[:,-1])
            test_preds = predict(clf.coef_, 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))
```

[5]:
```
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.linspace(0, 1, 101, endpoint=True), means, ses,␣
↪linestyle='None', marker='o')
plt.show()
```



[6]:
```
lamb_min_index = np.argmin(means)
lamb_min = lamb_min_index * .01
lamb_min
```

[6]: 0.03

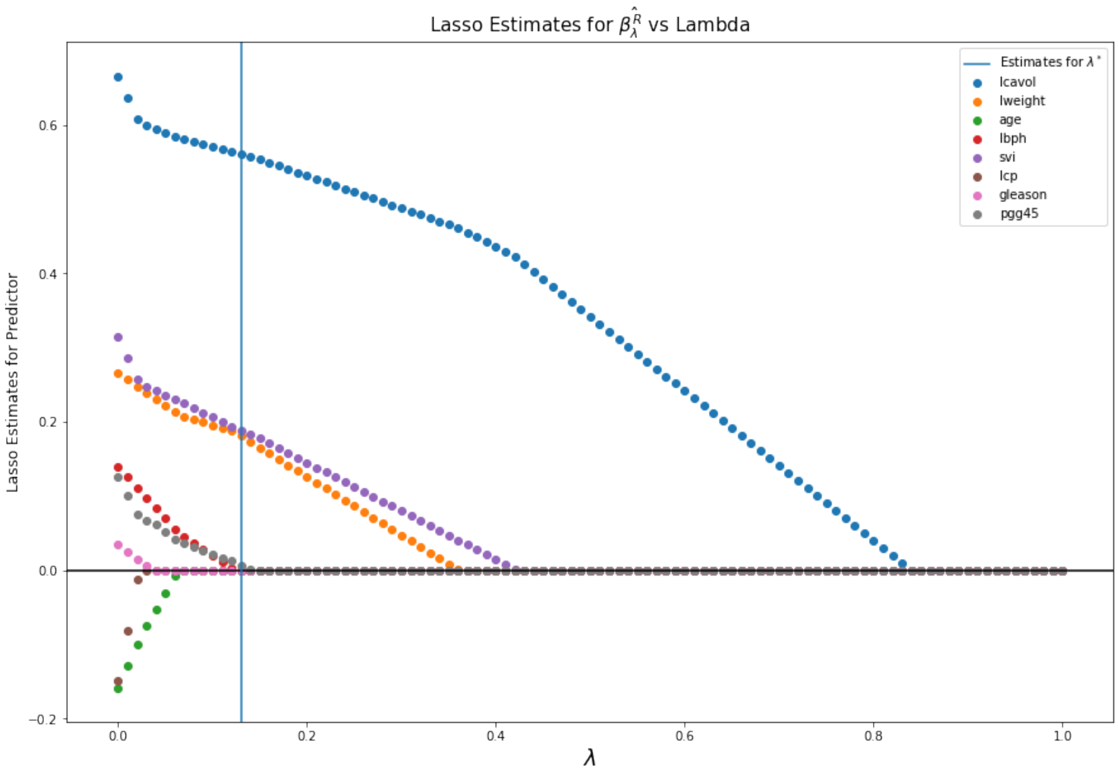
[7]:
```
lamb_best = None

for i in range(len(means)-1, -1, -1):
    if means[i] < means[lamb_min_index] + ses[lamb_min_index]:
        lamb_best_index = i
        lamb_best = i * .01
        break

lamb_best
```

[7]: 0.13

[8]:
```
data = np.genfromtxt('prostate_cancer.csv', delimiter=',', skip_header=1)[:,:-1]
preprocessor = LassoPreprocessor(data)
data = preprocessor.preprocess(data)
```

[9]:
```
betas = []

for lamb in np.linspace(0, 1, 101, endpoint=True):
    clf = Lasso(alpha=lamb, fit_intercept=False, normalize=False,␣
↪max_iter=10000)
    clf.fit(data[:,:-1], data[:,-1])
    betas.append(clf.coef_)
```

[12]:
```
labels = ["lcavol", "lweight", "age", "lbph", "svi", "lcp", "gleason", "pgg45"]
plt.rcParams['figure.figsize'] = [15, 10]
for i in range(len(labels)):
    plt.scatter(np.linspace(0, 1, 101, endpoint=True), 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('Lasso Estimates for Predictor', fontsize=12)
plt.title(r'Lasso Estimates for $\hat{\beta_\lambda^R}$ vs Lambda', fontsize=15)
plt.show()
```



[11]:
```
betas[lamb_best_index], preprocessor.get_y_avg()
```

[11]:
```
(array([ 0.56082709,  0.18144093, -0.        ,  0.        ,  0.18858446,
         0.        ,  0.        ,  0.00714978]),
 2.478386878350515)
```

The best final model is $f(X) = \bar{y} + \sum_{i=1}^{p} \beta^{\hat{R}}_{i,\lambda^*} X_i$ where $\beta^{\hat{R}}_{i,\lambda^*}$ = [0.56082709, 0.18144093, 0. , 0. , 0.18858446, 0. , 0. , 0.00714978] and $\bar{y} = 2.478$

2 a. $\mathbb{P}(G=0 \mid X=x) = \mathbb{P}(G=1 \mid X=x)$

$$\cancel{\pi} f_0(x) = \cancel{\pi} f_1(x)$$

$$f_0(x) = f_1(x)$$

$$\sum_{i=1}^{m} \frac{1}{m} \cancel{\left(\frac{1}{(2\pi)^{p/2} |\Sigma|^{1/2}}\right)} e^{-\frac{1}{2}(x-\mu_i)^T \Sigma^{-1} (x-\mu_i)} = \sum_{i=1}^{m} \frac{1}{m} \cancel{\left(\frac{1}{(2\pi)^{p/2} |\Sigma|^{1/2}}\right)} e^{-\frac{1}{2}(x-v_i)^T \Sigma^{-1} (x-v_i)}$$

$$\sum_{i=1}^{m} \exp\left[-\frac{1}{2}(x-\mu_i)^T \Sigma^{-1} (x-\mu_i)\right] = \sum_{i=1}^{m} \exp\left[-\frac{1}{2}(x-v_i)^T \Sigma^{-1} (x-v_i)\right]$$

$$[x_1-\mu_1 \ \dots \ x_p-\mu_p] \overset{p \times p}{\begin{bmatrix} \frac{1}{s} & & 0 \\ & \ddots & \\ 0 & & \frac{1}{s} \end{bmatrix}} \begin{bmatrix} x_1-\mu_1 \\ \vdots \\ x_p-\mu_p \end{bmatrix} = \sum_{j=1}^{p} \frac{1}{s}(x_j-\mu_j)^2$$

by symmetry same thing for vs

$$\sum_{i=1}^{m} \exp\left[\frac{-1}{2s} \sum_{j=1}^{p} (x_j - \underbrace{\mu_{ij}}_{j^{th} \text{ parameter of } i^{th}\ \mu})^2\right] = \sum_{i=1}^{m} \exp\left[\frac{-1}{2s} \sum_{j=1}^{p} (x_j - v_{ij})^2\right]$$

$$0 = \sum_{i=1}^{m} \left(\exp\left[\frac{-1}{2s} \sum_{j=1}^{p} (x_j - v_{ij})^2\right] - \exp\left[\frac{-1}{2s} \sum_{j=1}^{p} (x_j - \mu_{ij})^2\right]\right)$$

ps4_problem2

May 27, 2020

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

0.1.1 PS4, Problem 2b: Decision Boundary of the Bayes Classifier

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 matplotlib.pyplot as plt
```

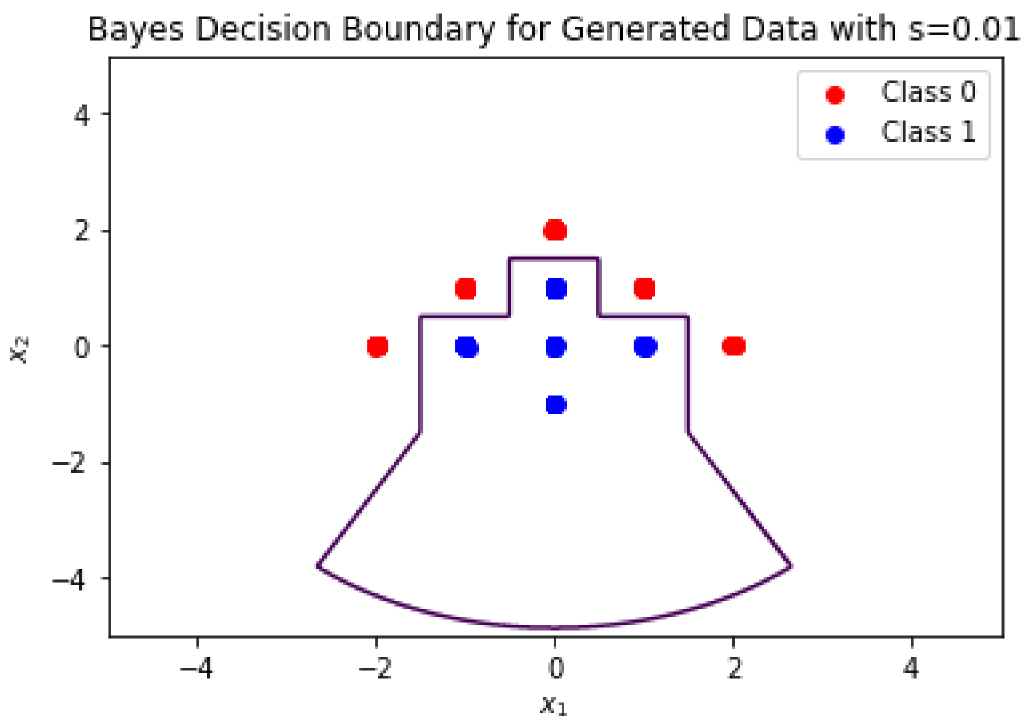
- Part b

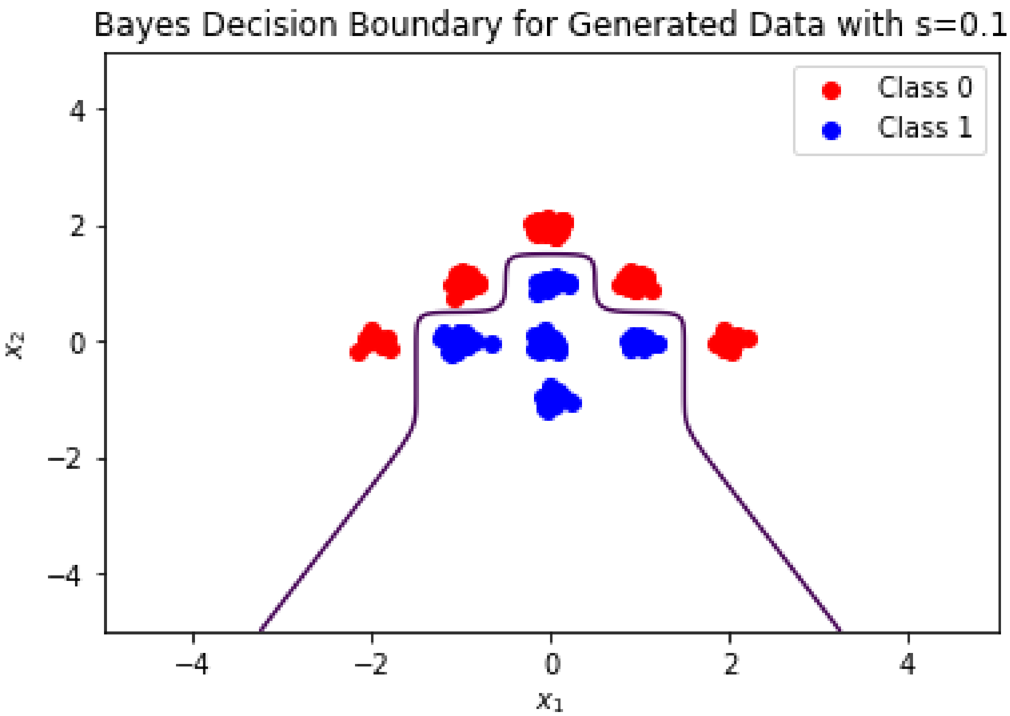
[2]:
```
# define our constants
p = 2
n = 100
us = [
    [-2, 0],
    [-1, 1],
    [0, 2],
    [1, 1],
    [2, 0]
]

vs = [
    [0, 1],
    [-1, 0],
    [0, 0],
    [1, 0],
    [0, -1]
]

ss = [.01, .1, 1]
```

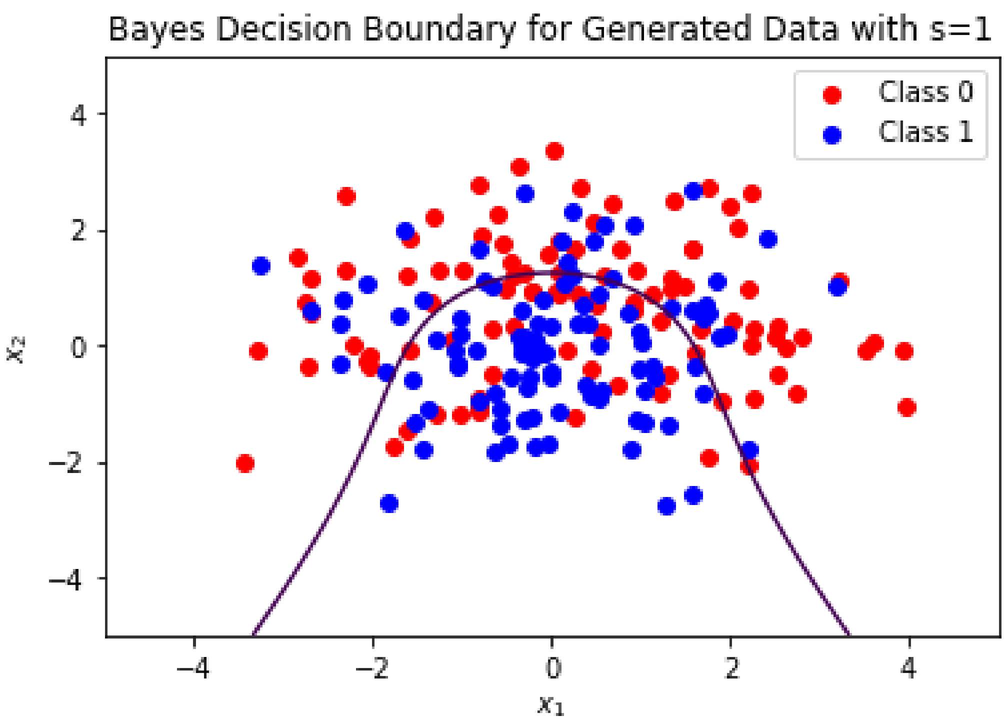
[3]:
```python
def boundary(x, s, us, vs):
    """
    x - candidate point for boundary point
    s - scalar for variance
    us - list of mean vectors for class 0
    vs - list of mean vectors for class 1

    returns the derived quantity from part 2a
    """
    class_1_sum = 0
    class_0_sum = 0

    for i in range(len(us)):
        class_0_sum += np.exp(-1/(2*s) * ((x[0]-us[i][0])**2 +␣
↪(x[1]-us[i][1])**2))
        class_1_sum += np.exp(-1/(2*s) * ((x[0]-vs[i][0])**2 +␣
↪(x[1]-vs[i][1])**2))

    return class_1_sum - class_0_sum
```

[4]:
```python
for s in ss:
    class_0 = []
    class_1 = []

    for i in range(n):
        # for each point we randomly pick a mean vector
        rand_u = us[np.random.choice(len(us))]
        rand_v = vs[np.random.choice(len(vs))]

        # generate random point with randomly seleted mean and variance sI_p
        class_0.append(np.random.normal(rand_u, s))
        class_1.append(np.random.normal(rand_v, s))

    # get the points for the boundary
    class_0 = np.array(class_0)
    class_1 = np.array(class_1)

    x_array = np.arange(-5, 5.01, 0.01)
    y_array = np.arange(-5, 5.01, 0.01)
    X,Y = np.meshgrid(x_array, y_array)
    Z = np.ndarray((1001, 1001))

    for i in range(1001):
        for j in range(1001):
            Z[i][j] = boundary([X[i][j], Y[i][j]], s, us, vs)

    plt.contour(X, Y, Z, [0])
```

```python
    plt.scatter(class_0[:,0], class_0[:,1], c='red', label='Class 0')
    plt.scatter(class_1[:,0], class_1[:,1], c='blue', label='Class 1')
    plt.xlabel('$x_1$')
    plt.ylabel('$x_2$')
    plt.title('Bayes Decision Boundary for Generated Data with s={}'.format(s))
    plt.legend()
    plt.show()
```




Bayes Decision Boundary for Generated Data with s=0.1
Class 0
Class 1
4
2
0
−2
−4
x_2
−4
−2
0
2
4
x_1


Bayes Decision Boundary for Generated Data with s=1
Class 0
Class 1
4
2
0
−2
−4
x_2
−4
−2
0
2
4
x_1

ps4_problem3

May 27, 2020

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

0.1.1 PS4, Problem 3: Logistic Regression Analysis of the Stock Market Data

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 pandas as pd
import scipy.stats

def logit(x, beta):
    """
    x - a random point p dimensional vector
    beta - an estimate for beta

    returns the logistic function for x using beta
    """
    return np.exp(np.matmul(x.transpose(), beta)) / (1 + np.exp(np.matmul(x.
 ↪transpose(), beta)))

def predict(data, beta):
    """
    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
    beta - coefficient estimates for logistic regression

    returns the probabilities of class 1 for each data point
    """
    x = data[:,:-1]
    bias = np.matlib.repmat(1, len(x), 1)
```

```python
    x = np.concatenate((bias, x), axis=1)

    return np.apply_along_axis(logit, 1, x, beta)

def logistic_regression(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 coefficient estimates for logistic regression using IRLS
    """
    x = data[:,:-1]
    y = data[:,-1]

    bias = np.matlib.repmat(1, len(x), 1)
    x = np.concatenate((bias, x), axis=1)

    # encode the data
    y = np.array([0 if item == -1 else item for item in y])
    tol = .000001

    # initialize beta to 0
    beta = np.array([0]*len(x[0]))
    w_k = None
    iters = 1 # just to prevent divide by 0 warning

    while True:
        p_k = np.apply_along_axis(logit, 1, x, beta)
        p_k_minus = 1 - p_k
        w_k = np.diag(np.multiply(p_k, p_k_minus))
        diff = y - p_k

        intermediate = np.matmul(np.matmul(x.transpose(), w_k), x)
        inverse_intermediate = np.linalg.inv(intermediate)
        pseudo_x = np.matmul(inverse_intermediate, x.transpose())
        beta_next = beta + np.matmul(pseudo_x, diff)

        # stopping condition if our two betas do not change enough
        if iters != 1 and np.linalg.norm(beta-beta_next, 2) / np.linalg.
↪norm(beta, 2) < tol:
            break
        else:
            beta = beta_next
            iters += 1

    return beta
```

```python
def sigma_squared(x, beta, j):
    """
    x - a matrix where each row corresponds to the
        p predictors
    beta - the coefficient estimates for logistic regression
    j - index of predictor

    returns the sigma_squared diagnol element
    """
    bias = np.matlib.repmat(1, len(x), 1)
    x = np.concatenate((bias, x), axis=1)

    p_k = np.apply_along_axis(logit, 1, x, beta)
    p_k_minus = 1 - p_k
    w_k = np.diag(np.multiply(p_k, p_k_minus))

    return np.linalg.inv(np.matmul(np.matmul(x.transpose(), w_k), x))[j][j]

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)
```

- Part A

[2]:
```python
train_data = np.genfromtxt('stock_market_train.csv', delimiter=',',␣
↪skip_header=1)
test_data = np.genfromtxt('stock_market_test.csv', delimiter=',', skip_header=1)
```

[3]:
```python
x = train_data[:,:-1]
y = train_data[:,-1]

beta = logistic_regression(train_data)
z_scores = []

# calculate z_scores for each predictor
for i in range(len(beta)):
    b = beta.copy()
    b[i] = 0
```

```python
    sig = np.sqrt(sigma_squared(x, b, i))
    z_scores.append(beta[i] / sig)

z_scores
```

[3]:
```
[-0.4957056982540905,
 -1.2949980620766537,
 -1.3221554529148911,
 -0.1532395033236583,
 0.2928063364320254,
 1.0797555989798535,
 0.7440645920961294]
```

[4]:
```python
p_vals = []

# calculate p_vals using z_scores
for score in z_scores:
    p_vals.append(2*scipy.stats.norm.cdf(-1*np.abs(score)))

p_vals
```

[4]:
```
[0.6201020661861175,
 0.19532089800477326,
 0.18611639147942072,
 0.8782094063929503,
 0.7696701846565389,
 0.2802510280256544,
 0.45683739909704013]
```

[5]:
```python
params = ['1', 'Lag1', 'Lag2', 'Lag3', 'Lag4', 'Lag5', 'Volume']
pd.DataFrame(data={'OLS estimate': beta,
                   'z-score': z_scores,
                   'p val': p_vals},
             index=params)
```

[5]:

	OLS estimate	z-score	p val
1	-0.135275	-0.495706	0.620102
Lag1	-0.073533	-1.294998	0.195321
Lag2	-0.072720	-1.322155	0.186116
Lag3	-0.008628	-0.153240	0.878209
Lag4	0.016658	0.292806	0.769670
Lag5	0.057834	1.079756	0.280251
Volume	0.132874	0.744065	0.456837

[6]:
```python
test_preds = predict(test_data, beta)
test_preds = [1 if item >= .5 else -1 for item in test_preds]
average_err = sum(test_preds != test_data[:,-1]) / len(test_preds)
```

```python
average_err
```

[6]: 0.496

- Part B

[7]:
```python
most_significant_indexes = [0, 1]
train_data = reduce_data(train_data, most_significant_indexes)
new_beta = logistic_regression(train_data)
```

[8]:
```python
test_data = reduce_data(test_data, most_significant_indexes)
test_preds = predict(test_data, new_beta)
test_preds = [1 if item >= .5 else -1 for item in test_preds]
average_err = sum(test_preds != test_data[:,-1]) / len(test_preds)
average_err
```

[8]: 0.472

[9]:
```python
# find test errors for g_0 and g_1
n_10 = 0
n_11 = 0
n_00 = 0
n_01 = 0

for i in range(len(test_preds)):
    if test_data[:,-1][i] == 1:
        if test_preds[i] == 1:
            n_00 += 1
        else:
            n_10 += 1
    else:
        if test_preds[i] == 1:
            n_01 += 1
        else:
            n_11 += 1
```

[10]:
```python
g_0_test_err = n_01 / (n_00 + n_01)
g_1_test_err = n_10 / (n_11 + n_10)
g_0_test_err, g_1_test_err
```

[10]: (0.4603174603174603, 0.5081967213114754)

From these errors we see that the model is wrong 46% of the time when it predicts the market will go up and 51% of the time when it predicts the market will go down. From this, we know the model is as good as guessing thus I would avoid trades using this model.

4. For OLS in this case we have a classification rule of $\hat{\beta}_0 + \hat{\beta}_1 x < C$ meaning our decision boundary is

$$x^* = \frac{c - \hat{\beta}_0}{\hat{\beta}_1}$$

So we will first use the normal equation to calculate $\hat{\beta}$

$$X^T X \beta = X^T y$$

$$\overbrace{\begin{bmatrix} 1 & \cdots & 1 \\ x_1 & \cdots & x_N \end{bmatrix} \begin{bmatrix} 1 & x_1 \\ \vdots & \vdots \\ 1 & x_N \end{bmatrix}}^{X^T X} = \begin{bmatrix} N & \Sigma x_i \\ \Sigma x_i & \Sigma x_i^2 \end{bmatrix} \qquad \overbrace{\begin{bmatrix} 1 & \cdots & 1 \\ x_1 & \cdots & x_N \end{bmatrix} \begin{bmatrix} y_1 \\ \vdots \\ y_N \end{bmatrix}}^{X^T Y} = \begin{bmatrix} \Sigma y_i \\ \Sigma x_i y_i \end{bmatrix}$$

We ultimately want to show that the x^* for OLS is the same as the x^* for LDA so we will define for LDA

$N_1 = \Sigma y_i$

$N_0 = N - N_1$

$\hat{\mu}_0 = \sum_{i:g_0} x_i / N_0 =$

$\hat{\mu}_1 = \sum_{i:g_1} x_i / N_1 = \frac{\Sigma x_i y_i}{N_1}$

thus we can rewrite $X^T X = \begin{bmatrix} N & N_0\hat{\mu}_0 + N_1\hat{\mu}_1 \\ N_0\hat{\mu}_0 + N_1\hat{\mu}_1 & \Sigma x_i^2 \end{bmatrix}$

$X^T y = \begin{bmatrix} N_1 \\ N_1\hat{\mu}_1 \end{bmatrix}$

Now we solve for $\hat{\sigma}^2 = \frac{1}{N-K} \sum_{k=1}^{K} \sum_{i:g_i = g_k} (x_i - \hat{\mu}_k)^2$

$$\hat{\sigma}^2 = \frac{1}{N-2} \left(\sum_{i:g_0} (x_i - \hat{\mu}_0)^2 + \sum_{i:g_1} (x_i - \hat{\mu}_1)^2 \right)$$

$$= \frac{1}{N-2} \left(\sum_{i:g_0} (x_i^2 - 2x_i\hat{\mu}_0 + \hat{\mu}_0^2) + \sum_{i:g_1} (x_i^2 - 2x_i\hat{\mu}_1 + \hat{\mu}_1^2) \right)$$

$$\hat{\sigma}^2 (N-2) = \sum_{i:g_0} x_i^2 - 2\hat{\mu}_0 \sum_{i:g_0} x_i + \overbrace{(N - N_1)}^{N_0}\hat{\mu}_0^2 + \sum_{i:g_1} x_i^2 - 2\hat{\mu}_1 \sum_{i:g_1} x_i + N_1\hat{\mu}_1^2$$

$$\hat{\sigma}^2 (N-2) = \sum_i x_i^2 - 2\hat{\mu}_0^2 N_0 + \overbrace{\boxed{N\hat{\mu}_0^2 - N_1\hat{\mu}_0^2}}^{N_0\hat{\mu}_0^2} - 2\hat{\mu}_1^2 N_1 + N_1\hat{\mu}_1^2$$

$$\hat{\sigma}^2 (N-2) = \sum_i x_i^2 - N_0\hat{\mu}_0^2 - N_1\hat{\mu}_1^2$$

$$\sum_i x_i^2 = \hat{\sigma}^2 (N-2) + N_0\hat{\mu}_0^2 + N_1\hat{\mu}_1^2$$

Returning to solving for $\hat{\beta}$ we have

$$\begin{bmatrix} N & N_0\hat{\mu}_0 + N_1\hat{\mu}_1 \\ N_0\hat{\mu}_0 + N_1\hat{\mu}_1 & \Sigma x_i^2 \end{bmatrix} \begin{bmatrix} \hat{\beta}_0 \\ \hat{\beta}_1 \end{bmatrix} = \begin{bmatrix} N_1 \\ N_1\hat{\mu}_1 \end{bmatrix}$$

$$N\hat{\beta}_0 + (N_0\hat{\mu}_0 + N_1\hat{\mu}_1)\hat{\beta}_1 = N_1$$
$$(N_0\hat{\mu}_0 + N_1\hat{\mu}_1)\hat{\beta}_0 + (\Sigma x_i^2)\hat{\beta}_1 = N_1\hat{\mu}_1$$

We first use elimination to find β_1 and plug in for Σx_i^2

$$\cancel{N(N_0\hat{\mu}_0 + N_1\hat{\mu}_1)\hat{\beta}_0} + (N_0\hat{\mu}_0 + N_1\hat{\mu}_1)^2\hat{\beta}_1 = N_1(N_0\hat{\mu}_0 + N_1\hat{\mu}_1)$$
$$\cancel{-N(N_0\hat{\mu}_0 + N_1\hat{\mu}_1)\hat{\beta}_0} - N(\Sigma x_i^2)\hat{\beta}_1 = -NN_1\hat{\mu}_1$$

$$\left((N_0\hat{\mu}_0 + N_1\hat{\mu}_1)^2 - N(\hat{\sigma}^2(N-2) + N_0\hat{\mu}_0^2 + N_1\hat{\mu}_1^2)\right)\hat{\beta}_1 = N_1N_0\hat{\mu}_0 + N_1^2\hat{\mu}_1 - NN_1\hat{\mu}_1$$

$$\hat{\beta}_1 = \frac{N_1N_0\hat{\mu}_0 + N_1\hat{\mu}_1(N-N_0) - NN_1\hat{\mu}_1}{N_0^2\hat{\mu}_0^2 + 2N_0N_1\hat{\mu}_0\hat{\mu}_1 + N_1^2\hat{\mu}_1^2 - N\hat{\sigma}^2(N-2) - NN_0\mu_0^2 - NN_1\hat{\mu}_1^2}$$

$$\hat{\beta}_1 = \frac{N_1N_0\hat{\mu}_0 - N_1N_0\hat{\mu}_1}{N_0\hat{\mu}_0^2(N-N_1) + 2N_0N_1\hat{\mu}_0\hat{\mu}_1 + N_1\hat{\mu}_1^2(N-N_0) - N\hat{\sigma}^2(N-2) - NN_0\mu_0^2 - NN_1\hat{\mu}_1^2}$$

$$\hat{\beta}_1 = \frac{N_1N_0\hat{\mu}_0 - N_1N_0\hat{\mu}_1}{-N_1N_0\hat{\mu}_0^2 + 2N_0N_1\hat{\mu}_0\hat{\mu}_1 - N_1N_0\hat{\mu}_1^2 - N\hat{\sigma}^2(N-2)}$$

$$\hat{\beta}_1 = \frac{N_1N_0(\hat{\mu}_1 - \hat{\mu}_0)}{N\hat{\sigma}^2(N-2) + (N_1N_0(\hat{\mu}_0^2 - 2\hat{\mu}_0\hat{\mu}_1 + \hat{\mu}_1^2))} = \frac{N_1N_0(\hat{\mu}_1 - \hat{\mu}_0)}{N\hat{\sigma}^2(N-2) + N_1N_0(\hat{\mu}_0 - \hat{\mu}_1)^2}$$

Plugging back in for $\hat{\beta}_0$ we get

$$N\hat{\beta}_0 + \frac{(N_0\hat{\mu}_0 + N_1\hat{\mu}_1)(N_1N_0\hat{\mu}_1 - N_1N_0\hat{\mu}_0)}{N\hat{\sigma}^2(N-2) + (N_1N_0(\hat{\mu}_0 - \hat{\mu}_1)^2)} = N_1$$

$$N\hat{\beta}_0 + \frac{(N_1N_0^2\hat{\mu}_0\hat{\mu}_1 - N_1N_0^2\hat{\mu}_0^2 + N_1^2N_0\hat{\mu}_1^2 - N_1^2N_0\mu_0\mu_1)}{N\hat{\sigma}^2(N-2) + N_1N_0(\hat{\mu}_0 - \hat{\mu}_1)^2} = N_1$$

$$N\hat{\beta}_0 + \frac{N_1 N_0 \hat{\mu}_0 \hat{\mu}_1 (N-N_1) - N_1 N_0 \hat{\mu}_0^2 (N-N_1) + N_1^2 N_0 \hat{\mu}_1^2 - N_1^2 N_0 \hat{\mu}_0 \hat{\mu}_1}{N\hat{\sigma}^2(N-2) + N_1 N_0 (\hat{\mu}_0 - \hat{\mu}_1)^2} = N_1$$

$$N\hat{\beta}_0 + \frac{N N_1 N_0 \hat{\mu}_0 \hat{\mu}_1 - 2N_1^2 N_0 \hat{\mu}_0 \hat{\mu}_1 - N N_1 N_0 \hat{\mu}_0^2 + N_1^2 N_0 \hat{\mu}_0^2 + N_1^2 N_0 \hat{\mu}_1^2}{N\hat{\sigma}^2(N-2) + N_1 N_0 (\hat{\mu}_0 - \hat{\mu}_1)^2} = N_1$$

$$N\hat{\beta}_0 + \frac{N N_1 N_0 \hat{\mu}_0 (\hat{\mu}_1 - \hat{\mu}_0) + N_1^2 N_0 (\hat{\mu}_0 - \mu_1)^2}{N\hat{\sigma}^2(N-2) + N_1 N_0 (\hat{\mu}_0 - \hat{\mu}_1)^2} = N_1$$

$$\cancel{N}\hat{\beta}_0 = \frac{\cancel{N} N_1 \hat{\sigma}^2 (N-2) + \cancel{N_1^2 N_0 (\hat{\mu}_0 - \hat{\mu}_1)^2} - \cancel{N} N_1 N_0 \hat{\mu}_0 (\hat{\mu}_1 - \hat{\mu}_0) \cancel{- N_1^2 N_0 (\hat{\mu}_0 - \hat{\mu}_1)^2}}{N\hat{\sigma}^2(N-2) + N_1 N_0 (\hat{\mu}_0 - \hat{\mu}_1)^2}$$

$$\hat{\beta}_0 = \frac{N_1 \hat{\sigma}^2 (N-2) + N_1 N_0 \hat{\mu}_0 (\hat{\mu}_0 - \hat{\mu}_1)}{N\hat{\sigma}^2(N-2) + N_1 N_0 (\hat{\mu}_0 - \hat{\mu}_1)^2}$$

For LDA, we know $x^* = \frac{\sigma^2 \log \frac{\pi_m}{\pi_k} + \frac{\mu_k^2 - \mu_m^2}{2}}{\mu_k - \mu_m}$ from Lec 12 pg 68

so setting the boundary of LDA and OLS to each other we can solve for c and show the equivalence of this situation

$$\frac{\hat{\sigma}^2 \log \overset{\frac{N_0}{N_1}}{\frac{\pi_0}{\pi_1}} + \frac{\hat{\mu}_1^2 - \hat{\mu}_0^2}{2}}{\hat{\mu}_1 - \hat{\mu}_0} = \frac{\frac{c(N\hat{\sigma}^2(N-2) + N_1 N_0 (\hat{\mu}_0 - \hat{\mu}_1)^2) - N_1 \hat{\sigma}^2 (N-2) - N_1 N_0 \hat{\mu}_0 (\hat{\mu}_0 - \hat{\mu}_1)}{N\hat{\sigma}^2(N-2) + N_1 N_0 (\hat{\mu}_0 - \hat{\mu}_1)^2}}{\frac{N_1 N_0 (\hat{\mu}_1 - \hat{\mu}_0)}{N\hat{\sigma}^2(N-2) + N_1 N_0 (\hat{\mu}_0 - \hat{\mu}_1)^2}}$$

$$\frac{\hat{\sigma}^2 \log \frac{N_0}{N_1} + \frac{\hat{\mu}_1^2 - \hat{\mu}_0^2}{2}}{\hat{\mu}_1 - \hat{\mu}_0} = \frac{c(N\hat{\sigma}^2(N-2) + N_1 N_0 (\hat{\mu}_0 - \hat{\mu}_1)^2) - N_1 \hat{\sigma}^2 (N-2) - N_1 N_0 \hat{\mu}_0 (\hat{\mu}_0 - \hat{\mu}_1)}{N_1 N_0 (\hat{\mu}_1 - \hat{\mu}_0)}$$

$$\frac{N_1 N_0 \sigma^2 \log \frac{N_0}{N_1} + N_1 N_0 \left(\frac{\hat{\mu}_1^2 - \hat{\mu}_0^2}{2}\right)}{N_1 N_0 (\hat{\mu}_1 - \hat{\mu}_0)} = \quad ||$$

$$C = \frac{N_1 N_0 \hat{\sigma}^2 \log \frac{N_0}{N_1} + N_1 N_0 \left(\frac{\hat{\mu}_1^2 - \hat{\mu}_0^2}{2}\right) + N_1 \hat{\sigma}^2 (N-2) + N_1 N_0 \hat{\mu}_0 (\hat{\mu}_0 - \hat{\mu}_1)}{N \hat{\sigma}^2 (N-2) + N_1 N_0 (\hat{\mu}_0 - \hat{\mu}_1)^2}$$

$$C = \frac{N_1 \hat{\sigma}^2 \left(N_0 \log \frac{N_0}{N_1} + N - 2\right) + N_1 N_0 \left(\frac{\hat{\mu}_1^2 - \hat{\mu}_0^2}{2}\right) + N_1 N_0 \hat{\mu}_0 (\hat{\mu}_0 - \hat{\mu}_1)}{N \hat{\sigma}^2 (N-2) + N_1 N_0 (\hat{\mu}_0 - \hat{\mu}_1)^2}$$

$$C = \frac{2 N_1 \hat{\sigma}^2 \left(N_0 \log \frac{N_0}{N_1} + N - 2\right) + N_1 N_0 (\hat{\mu}_1^2 - \hat{\mu}_0^2) + 2 N_1 N_0 \hat{\mu}_0 (\hat{\mu}_0 - \hat{\mu}_1)}{2 N \hat{\sigma}^2 (N-2) + 2 N_1 N_0 (\hat{\mu}_0 - \hat{\mu}_1)^2}$$

$$C = \frac{2 N_1 \hat{\sigma}^2 \left(N_0 \log \frac{N_0}{N_1} + N - 2\right) + N_1 N_0 \hat{\mu}_1^2 + N_1 N_0 \hat{\mu}_0^2 - 2 N_1 N_0 \mu_1 \mu_0}{2 N \hat{\sigma}^2 (N-2) + 2 N_1 N_0 (\hat{\mu}_0 - \hat{\mu}_1)^2}$$

$$C = \frac{2 N_1 \hat{\sigma}^2 \left(N_0 \log \frac{N_0}{N_1} + N - 2\right) + N_1 N_0 (\hat{\mu}_0 - \hat{\mu}_1)^2}{2 N \hat{\sigma}^2 (N-2) + 2 N_1 N_0 (\hat{\mu}_0 - \hat{\mu}_1)^2}$$

If $N_1 = N_0 = \frac{N}{2}$

$$C = \frac{N \hat{\sigma}^2 (N-2) + \frac{N^2}{4} (\hat{\mu}_0 - \hat{\mu}_1)^2}{2 N \hat{\sigma}^2 (N-2) + \frac{N^2}{2} (\hat{\mu}_0 - \hat{\mu}_1)^2} = \frac{4 N \hat{\sigma}^2 (N-2) + N^2 (\hat{\mu}_0 - \hat{\mu}_1)^2}{8 N \hat{\sigma}^2 (N-2) + 2 N^2 (\hat{\mu}_0 - \hat{\mu}_1)^2}$$

ps4_problem5

May 27, 2020

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

0.1.1 PS4, Problem 5: Linear and Quadratic Discriminant Analysis

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 matplotlib.pyplot as plt

def lda(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 coefficients for the lda functions
    """
    g1 = np.array([observation[:-1] for observation in data if observation[-1]␣
↪== 1])
    g0 = np.array([observation[:-1] for observation in data if observation[-1]␣
↪== 0])

    # calculate pi, mu, and pooled variance for each class
    pi1 = len(g1) / len(data)
    pi0 = len(g0) / len(data)

    mu1 = np.mean(g1, axis=0)
    mu0 = np.mean(g0, axis=0)

    pooled_var = (1 / (len(data) - 2)) * (np.matmul((g1-mu1).transpose(),␣
↪g1-mu1) + np.matmul((g0-mu0).transpose(), g0-mu0))
```

```python
    # get coefficients for each delta function for LDA
    delta_0_term_1 = np.matmul(mu0.transpose(), np.linalg.inv(pooled_var))
    delta_0_term_2 = (1/2) * np.matmul(np.matmul(mu0.transpose(), np.linalg.inv(pooled_var)), mu0)
    delta_0_term_3 = np.log(pi0)

    delta_1_term_1 = np.matmul(mu1.transpose(), np.linalg.inv(pooled_var))
    delta_1_term_2 = (1/2) * np.matmul(np.matmul(mu1.transpose(), np.linalg.inv(pooled_var)), mu1)
    delta_1_term_3 = np.log(pi1)

    return (delta_0_term_1, delta_0_term_2, delta_0_term_3), (delta_1_term_1, delta_1_term_2, delta_1_term_3)

def qda(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 coefficients for the qda functions
    """
    g1 = np.array([observation[:-1] for observation in data if observation[-1] == 1])
    g0 = np.array([observation[:-1] for observation in data if observation[-1] == 0])

    # calculate pi, mu, and variance for each class
    pi1 = len(g1) / len(data)
    pi0 = len(g0) / len(data)

    mu1 = np.mean(g1, axis=0)
    mu0 = np.mean(g0, axis=0)

    var1 = (1 / (len(g1)-1)) * (np.matmul((g1-mu1).transpose(), g1-mu1))
    var0 = (1 / (len(g0)-1)) * (np.matmul((g0-mu0).transpose(), g0-mu0))

    # get coefficients for each delta function for QDA
    delta_0_term_2 = -1/2 * np.log(np.linalg.det(var0))
    delta_0_term_3 = np.log(pi0)

    delta_1_term_2 = -1/2 * np.log(np.linalg.det(var1))
    delta_1_term_3 = np.log(pi1)

    return (mu0, np.linalg.inv(var0), delta_0_term_2+delta_0_term_3), (mu1, np.linalg.inv(var1), delta_1_term_2+delta_1_term_3)
```

```python
def find_linear_boundary(delta_0, delta_1):
    """
    delta_0 - the coefficients for delta_0 LDA function
    delta_1 - the coefficients for delta_1 LDA function

    returns the slope and bias for the linear boundary between the two classes
    """
    delta_0_term_1, delta_0_term_2, delta_0_term_3 = delta_0
    delta_1_term_1, delta_1_term_2, delta_1_term_3 = delta_1

    bias = -delta_1_term_2+delta_1_term_3+delta_0_term_2-delta_0_term_3
    mat = delta_0_term_1 - delta_1_term_1

    return -mat[0]/mat[1], bias/mat[1]

def find_quad_boundary(data, delta_0, delta_1):
    """
    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
    delta_0 - the coefficients for delta_0 QDA function
    delta_1 - the coefficients for delta_1 QDA function

    returns the points that make up the boundary for the classes using QDA
    """
    mu0, inverse_var0, bias0 = delta_0
    mu1, inverse_var1, bias1 = delta_1

    # define grid
    x0s = np.arange(min(data[:,0]), max(data[:,0]), .01)
    x1s = np.arange(min(data[:,1]), max(data[:,1]), .01)

    tol = .01
    xs = []
    ys = []

    # check each point in grid and see if delta functions are close enough
    for x0 in x0s:
        for x1 in x1s:
            x = np.array([x0, x1])

            delta_0_term_1 = -1/2 * np.matmul(np.matmul((x-mu0).transpose(),␣
↪inverse_var0), x-mu0)
            delta_1_term_1 = -1/2 * np.matmul(np.matmul((x-mu1).transpose(),␣
↪inverse_var1), x-mu1)
```

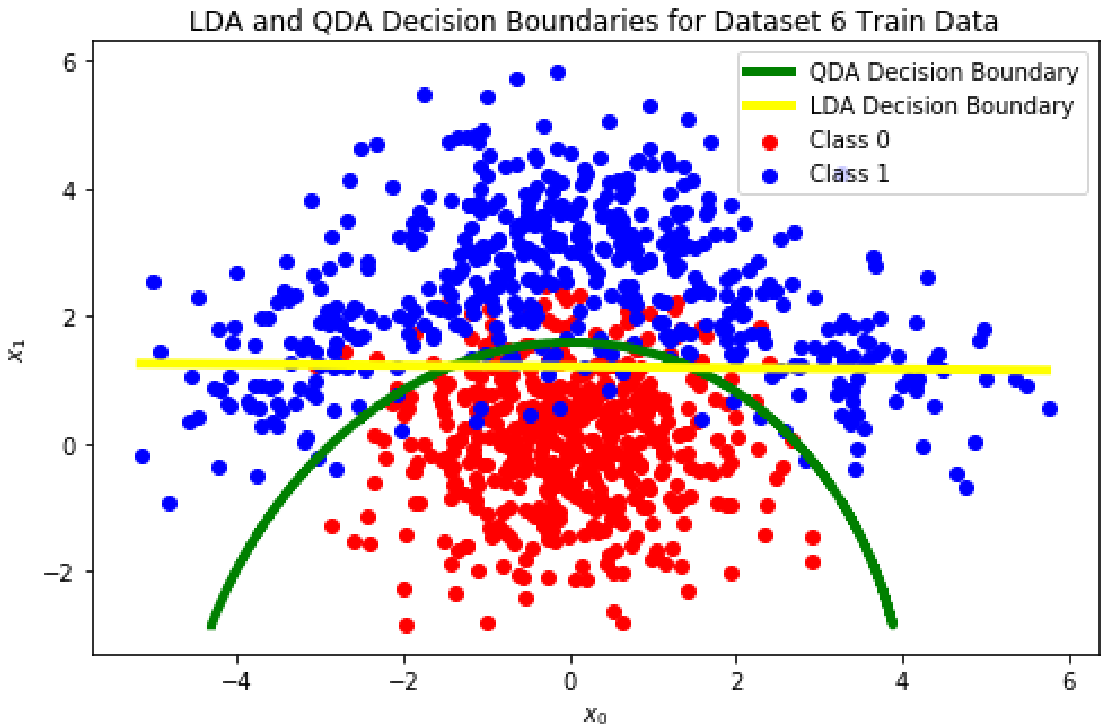
```python
            delta_0 = delta_0_term_1 + bias0
            delta_1 = delta_1_term_1 + bias1

            # if so we add those points to our list of points for the boundary
            if np.abs(delta_0 - delta_1) < tol:
                xs.append(x0)
                ys.append(x1)

    return xs, ys

def lda_predict(delta_0, delta_1, x, alpha=1/2):
    """
    delta_0 - the coefficients for delta_0 LDA function
    delta_1 - the coefficients for delta_1 LDA function
    alpha - classification threshold

    returns the prediction using LDA functions
    """
    delta_0_term_1, delta_0_term_2, delta_0_term_3 = delta_0
    delta_1_term_1, delta_1_term_2, delta_1_term_3 = delta_1

    delta_0 = np.matmul(delta_0_term_1, x) - delta_0_term_2 + delta_0_term_3
    delta_1 = np.matmul(delta_1_term_1, x) - delta_1_term_2 + delta_1_term_3

    if alpha == 0:
        return 0
    elif alpha == 1:
        return 1
    else:
        # classification rule derived in part c
        return 0 if delta_0 > delta_1 + np.log(alpha / (1-alpha)) else 1

def qda_predict(delta_0, delta_1, x, alpha=1/2):
    """
    delta_0 - the coefficients for delta_0 QDA function
    delta_1 - the coefficients for delta_1 QDA function
    alpha - classification threshold

    returns the prediction using QDA functions
    """
    mu0, inverse_var0, bias0 = delta_0
    mu1, inverse_var1, bias1 = delta_1

    delta_0_term_1 = -1/2 * np.matmul(np.matmul((x-mu0).transpose(),
 ↪inverse_var0), x-mu0)
```

```
    delta_1_term_1 = -1/2 * np.matmul(np.matmul((x-mu1).transpose(),␣
 ↪inverse_var1), x-mu1)

    delta_0 = delta_0_term_1 + bias0
    delta_1 = delta_1_term_1 + bias1

    if alpha == 0:
        return 0
    elif alpha == 1:
        return 1
    else:
        # classification rule derived in part c
        return 0 if delta_0 > delta_1 + np.log(alpha / (1-alpha)) else 1

def class_errors(real, preds):
    """
    real - list of real classes for each observation
    preds - list of predicted classes for each observation

    returns test errors on g_0 and g_1
    """
    n_10 = 0
    n_11 = 0
    n_00 = 0
    n_01 = 0

    for i in range(len(preds)):
        if real[i] == 1:
            if preds[i] == 1:
                n_11 += 1
            else:
                n_01 += 1
        else:
            if preds[i] == 1:
                n_10 += 1
            else:
                n_00 += 1

    return n_10 / (n_00 + n_10), n_01 / (n_11 + n_01)
```

- Part A

[2]:
```
train_data = np.genfromtxt('dataset6_train.csv', delimiter=',', skip_header=1)
test_data = np.genfromtxt('dataset6_test.csv', delimiter=',', skip_header=1)

# find the lda coefficients and decision boundary
```

```
lda_delta_0, lda_delta_1 = lda(train_data)
coeffs = find_linear_boundary(lda_delta_0, lda_delta_1)
lda_boundary_x = np.arange(min(train_data[:,0]), max(train_data[:,0]), .1)
lda_boundary_y = [coeffs[0]*i+coeffs[1] for i in lda_boundary_x]

# find the qda coefficients and decision boundary
qda_delta_0, qda_delta_1 = qda(train_data)
qda_boundary_x, qda_boundary_y = find_quad_boundary(train_data, qda_delta_0,␣
↪qda_delta_1)

# get the points of the training data
train_1s = np.array([data[:-1] for data in train_data if data[-1] == 1])
train_0s = np.array([data[:-1] for data in train_data if data[-1] == 0])

plt.rcParams['figure.figsize'] = [8, 5]
plt.title('LDA and QDA Decision Boundaries for Dataset 6 Train Data')
plt.xlabel('$x_0$')
plt.ylabel('$x_1$')
plt.scatter(train_0s[:,0], train_0s[:,1], c='red', label='Class 0')
plt.scatter(train_1s[:,0], train_1s[:,1], c='blue', label='Class 1')
plt.plot(qda_boundary_x, qda_boundary_y, linewidth='4', c='green', label='QDA␣
↪Decision Boundary')
plt.plot(lda_boundary_x, lda_boundary_y, linewidth='4', c='yellow', label='LDA␣
↪Decision Boundary')
plt.legend()
plt.show()
```



- Part B

[3]:
```
# make predictions with the two models and check errors
lda_test_preds = [lda_predict(lda_delta_0, lda_delta_1, x) for x in test_data[:
↪,:-1]]
qda_test_preds = [qda_predict(qda_delta_0, qda_delta_1, x) for x in test_data[:
↪,:-1]]

lda_average_err = sum(lda_test_preds != test_data[:,-1]) / len(lda_test_preds)
qda_average_err = sum(qda_test_preds != test_data[:,-1]) / len(qda_test_preds)
lda_average_err, qda_average_err
```

[3]:
```
(0.127, 0.059)
```

We see here from the errors that Quadratic Discriminant Analysis is more eficient on this particular dataset.

- Part C

$$P(G = G_0|X = x) > \alpha \implies P(G = G_0|X = x) = \alpha \text{ and } P(G = G_0|X = x) = 1 - \alpha$$

$$P(G = G_0|X = x) = \frac{\alpha}{1-\alpha} P(G = G_1|X = x)$$

$$\log(P(G = G_0|X = x)) = \log(\frac{\alpha}{1-\alpha} P(G = G_1|X = x))$$

$$\log(\pi_0 f_0(x)) - \log(\sum_{s=1}^{k} \pi_s f_s(x)) = \log(\frac{\alpha}{1-\alpha}) + \log(\pi_1 f_1(x)) - \log(\sum_{s=1}^{k} \pi_s f_s(x))$$

$$\log(\pi_0) + \log(\frac{1}{(2\pi)^{p/2}|\Sigma|^{\frac{1}{2}}}) + \log(\exp[\frac{-1}{2}(x-\mu_0)^T \Sigma^{-1}(x-\mu_0)]) = \log(\frac{\alpha}{1-\alpha}) + \log(\pi_1) + \log(\frac{1}{(2\pi)^{p/2}|\Sigma|^{\frac{1}{2}}}) + \log(\exp[\frac{-1}{2}$$

$$\log(\pi_0) + \log(\exp[\frac{-1}{2}(x-\mu_0)^T \Sigma^{-1}(x-\mu_0)]) = \log(\frac{\alpha}{1-\alpha}) + \log(\pi_1) + \log(\exp[\frac{-1}{2}(x-\mu_1)^T \Sigma^{-1}(x-\mu_1)])$$

From lecture 12 page 69, we know that we are able to derive $\delta_0(x)$ and $\delta_1(x)$ from here which gives us our definition of each delta so we can substitute that in here. By symmetry, we can say the same rule applies for QDA. Thus, our new decision boundary is found by this

$$\delta_0(x) = \delta_1(x) + \log(\frac{\alpha}{1-\alpha})$$

So this gives us our new decision boundary and our new classification rule becomes

$$\delta_0(x) > \delta_1(x) + \log(\frac{\alpha}{1-\alpha})$$

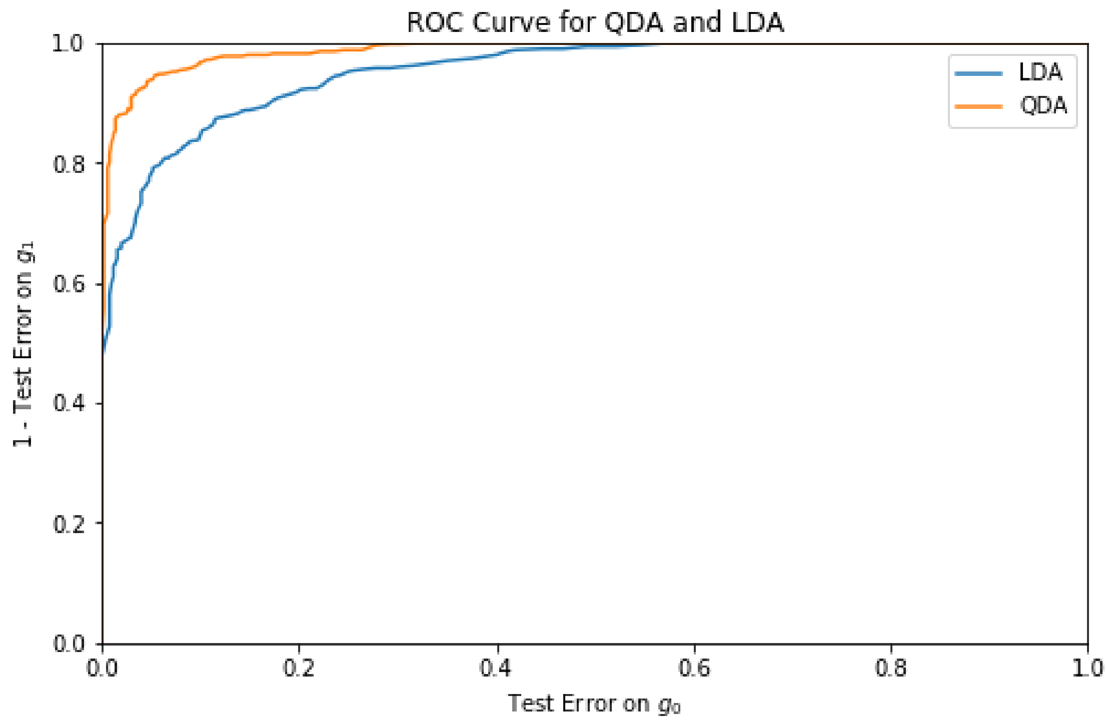
if this is true then x is class G_0 otherwise it is G_1

[4]:
```
alphas = np.arange(0, 1.01, .01)
lda_err_0s = []
lda_one_minus_err_1s = []

qda_err_0s = []
qda_one_minus_err_1s = []

# for all the classification thresholds
for a in alphas:
    # make our predictions and find the class errors and add to our list of␣
 ↪points to plot
    lda_test_preds = [lda_predict(lda_delta_0, lda_delta_1, x, a) for x in␣
 ↪test_data[:,:-1]]
    qda_test_preds = [qda_predict(qda_delta_0, qda_delta_1, x, a) for x in␣
 ↪test_data[:,:-1]]

    lda_errs = class_errors(test_data[:,-1], lda_test_preds)
    qda_errs = class_errors(test_data[:,-1], qda_test_preds)

    lda_err_0s.append(lda_errs[0])
    lda_one_minus_err_1s.append(1-lda_errs[1])
```

```
    qda_err_0s.append(qda_errs[0])
    qda_one_minus_err_1s.append(1-qda_errs[1])
```

[5]:

```
plt.plot(lda_err_0s, lda_one_minus_err_1s, label='LDA')
plt.plot(qda_err_0s, qda_one_minus_err_1s, label='QDA')
plt.xlim(0, 1)
plt.ylim(0, 1)
plt.title('ROC Curve for QDA and LDA')
plt.xlabel('Test Error on $g_0$')
plt.ylabel('1 - Test Error on $g_1$')
plt.legend()
plt.show()
```



QDA is still more efficient based on the ROC curves.