BILKENT UNIVERSITY

CS464 HOMEWORK 2

BERAT BİÇER
21503050

Dec. 4, 2018
ANKARA, TURKEY

Question 1.1

$$min\frac{1}{2}\|y-XB\|^2 \equiv min\, tr(Y'Y)-2tr(Y'XB)+tr(B'X'XB)$$

$$(\partial/\partial B)=0 \Rightarrow \widehat{B}=(X'X)^{-1}X'Y$$

Question 1.2

There are$rank(X'X)=6$ independent features in the model.

Question 1.3

Coefficients: [-7.68973700e-01 -2.66894243e-02 -3.62572818e-02 3.28973372e-05
-3.03106645e-01 5.28691557e-01]
Train Error: 10.20164702644783
Test Error: 35.975047174669974

Question 1.4

If a coefficient is negative, an increase in corresponding feature value will result in a decrease in the prediction. Likewise if a coefficient is positive, an increase in corresponding feature value will result in an increase in the prediction.
As the magnitude of a coefficient increases, the effect of corresponding feature on the prediction will increase, making it a stronger factor in determining the prediction.

Question 1.5

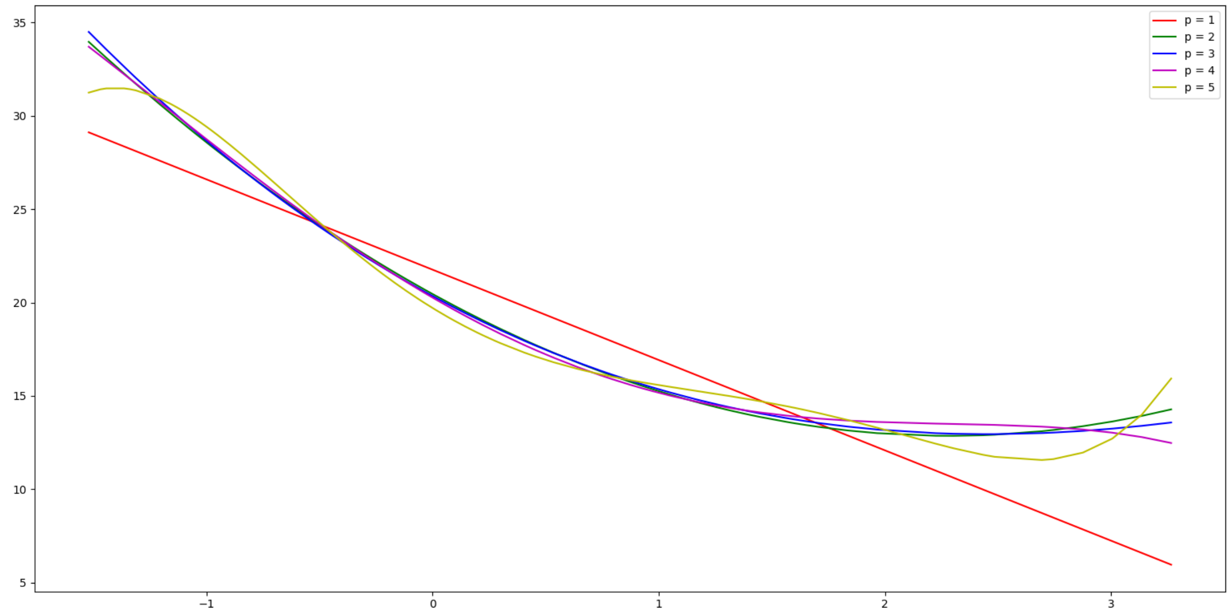
MPG is inversely proportional to horsepower(HP) value in this dataset, which explains the negative coefficient of beta values in Question 1.3.

Question 1.6

Matrix rank increases until p = 2, then stays constant at rank = 3: *rank = p + 1, if p <= 2; else, 3*. This means powers of HP greater than 2 can be written as a linear combination of lesser powers, and they powers cannot be used as features directly. After normalization, we observed that rank of X'X increases as we increase the parameter p, which implies centralization created linearly independent features that can be used to predict class label of train data. Then, rank function becomes: *rank = p, p <= 6*. However if X'X is not invertible, we cannot use the closed solution of beta directly. For example, when p = 0, we have a singular matrix.

Question 1.7

Since p = 0 gives a singular matrix for closed form of beta, we ignore it.



Results:
Coefficients of Beta for p = 1:
[21.75876359 -4.83692608]
Train Error for p = 1: 14.341899019749064
Test Error for p = 1: 73.83848307979181
Coefficients of Beta for p = 2:
[20.45431495 -6.65954213 1.46039737]
Train Error for p = 2: 10.839424149052432
Test Error for p = 2: 60.53249695797431
Coefficients of Beta for p = 3:
[20.34507117 -6.56475888 1.66498393 -0.08860305]
Train Error for p = 3: 10.816044462865847
Test Error for p = 3: 60.31291330318858
Coefficients of Beta for p = 4:
[20.26224255 -7.02669019 1.80373431 0.23874763 -0.10890227]
Train Error for p = 4: 10.771087052589538
Test Error for p = 4: 59.991301591670435
Coefficients of Beta for p = 5:
[19.70530656 -7.42906053 3.86834442 0.26457814 -1.07547209 0.24861662]
Train Error for p = 5: 10.383020895493294
Test Error for p = 5: 58.329788653047466

Question 1.8

Coefficients of Beta for p = 1:
[22.0255366 -4.69684017 0.71822189]
Train Error for p = 1: 14.066764652607663
Test Error for p = 1: 59.22618224681992
Coefficients of Beta for p = 2:
[20.21911804 -6.62573072 1.53009732 2.44046804 1.54598368]
Train Error for p = 2: 9.45156444953559
Test Error for p = 2: 22.293142924636175
Coefficients of Beta for p = 3:
[20.08490786 -6.5159226 1.76776157 -0.10331268 2.44994773 1.56194401]
Train Error for p = 3: 9.419849048918984
Test Error for p = 3: 21.94819248761814

We observe that since underlying distribution is not linear, using polynomials of a feature set gives better results than simple linear regression. Moreover, using many features (ignoring curse of dimensionality) results in lower test error for almost all cases. It is also clear that after p = 2, for 1.7, using higher powers do not result in significant performance improvement which means p = 2 is the most cost-effective predictor for Question 1.7 Similarly, we see that p = 2 is the most cost-effective predictor for Question 1.8 as well. Finally, observe that polynomials of multiple features is the best predictor we trained, which is suitable with the assumption that more features usually means better outcomes.

Question 2.1

Hyper parameters: Iteration count -> 2500, Learning Rate -> 0.001
Confusion Matrix -> {'tp': 20, 'tn': 20, 'fp': 0, 'fn': 0}

Question 2.2

Current status of Question 2.2 includes forward selection and backward elimination implementations, however, because of the long execution time we are unable to acquire the final output (backward elimination only completed ~70 eliminations in approximately 4 hours, and forward selection completed ~1000 features). However, the program we provide correctly selects features except the first feature, which is a by-product of initialization step of the algorithms. Since we do not have the final output, we are unable to provide the confusion matrix, however, assuming execution is completed, the program will indeed print the confusion matrices for each feature set.

As a final note to their executions, we observed that intermediate outputs of two algorithms are different. This is so since backward elimination seeks features that, when removed from feature set, will decrease classification accuracy whereas forward selection seeks features that, when inserted into a subset of the feature set, will improve the classification accuracy. From here, we see the distinction between the two as selecting non-optimal features that increase intermediate prediction accuracy vs. removing non-optimal features that do not reduce intermediate prediction accuracy. Hence, the execution of the algorithms above and their outputs are naturally different. To address this issue, we might run forward selection algorithm multiple times with random shuffling of the feature set in order to eliminate the effect of locations of features on the output.

Question 3.1

First option is line $x1 + x2 = 3$, with margin $w = 1 / \sqrt{2}$. In this case, optimization function is $||w||^2 + C * (\text{number of errors}) = ||w||^2$, since no error exists.

Another option is $x1 + x2 = 5$, with margin $= \sqrt{2}$. In this case, optimization function is $||w||^2 + C * 1$, since data point (2,2) violates the margin.

Last option is $x1 + x2 = 7$, with margin $= 2\sqrt{2}$. In this case, optimization function is $||w||^2 + C * 2$, since orange points violate the margin.

So, for any C value greater than 0, optimizer will select the hard margin SVM.

Question 3.2

If C is set to infinity, all constraints are enforced and the margin becomes a hard one, since cost of a margin violation becomes infinity.

Question 3.3

C -> 10.0
Confusion Matrix -> {'tp': 42, 'tn': 141, 'fp': 0, 'fn': 0}
Accuracy: 1.0

We used k-fold validation with iteration count 5, for statistically significant results , and small number of iterations.

Question 3.4

Linear SVM with C = 10.0
Confusion Matrix -> {'tp': 42, 'tn': 141, 'fp': 0, 'fn': 0}
Accuracy: 1.0

SVM with RBF kernel, gamma = 0.0625
Confusion Matrix -> {'tp': 42, 'tn': 138, 'fp': 3, 'fn': 0}
Accuracy: 0.9836065573770492

We used k-fold validation with iteration count 5, for statistically significant results , and small number of iterations.

Source Code

Question 1

```
# Cylinders, Displacement, Horsepower, Weight, Acceleration, Model Year and
MPG

import numpy as np
from numpy.linalg import inv, matrix_rank
import matplotlib.pyplot as plt

data = np.loadtxt("../data/carbig.csv", delimiter=',')
print("Question 1.2: ")
(x, y) = data.shape
train_sample_count = 300
train_data = data[:train_sample_count, : y - 1]
train_labels = data[:train_sample_count, y - 1]
test_data = data[train_sample_count : x + 1, : y - 1]
test_labels = data[train_sample_count: x + 1, y - 1]
# print(matrix_rank(np.transpose(train_data).dot(train_data)))
beta                                                                     =
inv(np.transpose(train_data).dot(train_data)).dot(np.transpose(train_data)).do
t(train_labels)
train_predictions = train_data.dot(beta)
train_error = (np.square(train_labels - train_predictions)).mean(axis = None)
test_predictions = test_data.dot(beta)
test_error = (np.square(test_labels - test_predictions)).mean(axis = None)
print("Coefficients of Beta: ")
print(beta)
print("Train Error: " + str(train_error))
print("Test Error: " + str(test_error))

print("******************************")
print("Question 1.5: ")
(x, y) = data.shape
plt.plot(data[:, 2], data[:, y - 1], ".")
plt.xlabel("Horsepower")
plt.ylabel("MPG")
plt.show()

print("******************************")
print("Question 1.6: ")
features = np.column_stack((np.ones(x), data[:, 2], data[:, 2] ** 2, data[:,
2] ** 3, data[:, 2] ** 4, data[:, 2] ** 5))
f1 = features[:, 0]
f2 = features[:, 0:2]
f3 = features[:, 0:3]
```

```
f4 = features[:, 0:4]
f5 = features[:, 0:5]
f6 = features[:, 0:6]
xTx1 = np.transpose(f1).dot(f1)
xTx2 = np.transpose(f2).dot(f2)
xTx3 = np.transpose(f3).dot(f3)
xTx4 = np.transpose(f4).dot(f4)
xTx5 = np.transpose(f5).dot(f5)
xTx6 = np.transpose(f6).dot(f6)
rank1 = matrix_rank(xTx1)
rank2 = matrix_rank(xTx2)
rank3 = matrix_rank(xTx3)
rank4 = matrix_rank(xTx4)
rank5 = matrix_rank(xTx5)
rank6 = matrix_rank(xTx6)
print("Ranks before normalization: ")
print("Rank for p = 0: " + str(rank1))
print("Rank for p = 1: " + str(rank2))
print("Rank for p = 2: " + str(rank3))
print("Rank for p = 3: " + str(rank4))
print("Rank for p = 4: " + str(rank5))
print("Rank for p = 5: " + str(rank6))

mean = features.mean(0)
stddev = features.std(0)
for i in range(x):
      for j in range(1, y - 1):
            features[i][j] = (lambda x, y, z, i, j: float(x - y[j]) / z[j])
(features[i][j], mean, stddev, i, j)

f1 = features[:, 0]
f2 = features[:, 0:2]
f3 = features[:, 0:3]
f4 = features[:, 0:4]
f5 = features[:, 0:5]
f6 = features[:, 0:6]
xTx1 = np.transpose(f1).dot(f1)
xTx2 = np.transpose(f2).dot(f2)
xTx3 = np.transpose(f3).dot(f3)
xTx4 = np.transpose(f4).dot(f4)
xTx5 = np.transpose(f5).dot(f5)
xTx6 = np.transpose(f6).dot(f6)
rank1 = matrix_rank(xTx1)
rank2 = matrix_rank(xTx2)
rank3 = matrix_rank(xTx3)
rank4 = matrix_rank(xTx4)
rank5 = matrix_rank(xTx5)
rank6 = matrix_rank(xTx6)
print("Ranks before normalization: ")
```

```
print("Rank for p = 0: " + str(rank1))
print("Rank for p = 1: " + str(rank2))
print("Rank for p = 2: " + str(rank3))
print("Rank for p = 3: " + str(rank4))
print("Rank for p = 4: " + str(rank5))
print("Rank for p = 5: " + str(rank6))

print("******************************")
print("Question 1.7: ")
hp = data[:, 2]
mean_hp = hp.mean(0)
stddev_hp = hp.std(0)
for i in range(x):
    hp[i] = (lambda x: (x - mean_hp) / stddev_hp)(hp[i])

features_17_1 = np.column_stack((np.ones(x), hp))
features_17_2 = np.column_stack((np.ones(x), hp, hp ** 2))
features_17_3 = np.column_stack((np.ones(x), hp, hp ** 2, hp ** 3))
features_17_4 = np.column_stack((np.ones(x), hp, hp ** 2, hp ** 3, hp ** 4))
features_17_5 = np.column_stack((np.ones(x), hp, hp ** 2, hp ** 3, hp ** 4, hp
** 5))

# p = 1
train_sample_count = 300
train_data = features_17_1[:train_sample_count, : y - 1]
train_labels = data[:train_sample_count, y - 1]
test_data = features_17_1[train_sample_count : x + 1, : y - 1]
test_labels = data[train_sample_count: x + 1, y - 1]
beta_1                                                                        =
inv(np.transpose(train_data).dot(train_data)).dot(np.transpose(train_data)).do
t(train_labels)
train_predictions = train_data.dot(beta_1)
train_error = (np.square(train_labels - train_predictions)).mean(axis = None)
test_predictions = test_data.dot(beta_1)
test_error = (np.square(test_labels - test_predictions)).mean(axis = None)
print("Coefficients of Beta for p = 1: ")
print(beta_1)
print("Train Error for p = 1: " + str(train_error))
print("Test Error for p = 1: " + str(test_error))

# p = 2
train_sample_count = 300
train_data = features_17_2[:train_sample_count, : y - 1]
train_labels = data[:train_sample_count, y - 1]
test_data = features_17_2[train_sample_count : x + 1, : y - 1]
test_labels = data[train_sample_count: x + 1, y - 1]
beta_2                                                                        =
inv(np.transpose(train_data).dot(train_data)).dot(np.transpose(train_data)).do
t(train_labels)
```

```
train_predictions = train_data.dot(beta_2)
train_error = (np.square(train_labels - train_predictions)).mean(axis = None)
test_predictions = test_data.dot(beta_2)
test_error = (np.square(test_labels - test_predictions)).mean(axis = None)
print("Coefficients of Beta for p = 2: ")
print(beta_2)
print("Train Error for p = 2: " + str(train_error))
print("Test Error for p = 2: " + str(test_error))

# p = 3
train_sample_count = 300
train_data = features_17_3[:train_sample_count, : y - 1]
train_labels = data[:train_sample_count, y - 1]
test_data = features_17_3[train_sample_count : x + 1, : y - 1]
test_labels = data[train_sample_count: x + 1, y - 1]
beta_3                                                                       =
inv(np.transpose(train_data).dot(train_data)).dot(np.transpose(train_data)).do
t(train_labels)
train_predictions = train_data.dot(beta_3)
train_error = (np.square(train_labels - train_predictions)).mean(axis = None)
test_predictions = test_data.dot(beta_3)
test_error = (np.square(test_labels - test_predictions)).mean(axis = None)
print("Coefficients of Beta for p = 3: ")
print(beta_3)
print("Train Error for p = 3: " + str(train_error))
print("Test Error for p = 3: " + str(test_error))

# p = 4
train_sample_count = 300
train_data = features_17_4[:train_sample_count, : y - 1]
train_labels = data[:train_sample_count, y - 1]
test_data = features_17_4[train_sample_count : x + 1, : y - 1]
test_labels = data[train_sample_count: x + 1, y - 1]
beta_4                                                                       =
inv(np.transpose(train_data).dot(train_data)).dot(np.transpose(train_data)).do
t(train_labels)
train_predictions = train_data.dot(beta_4)
train_error = (np.square(train_labels - train_predictions)).mean(axis = None)
test_predictions = test_data.dot(beta_4)
test_error = (np.square(test_labels - test_predictions)).mean(axis = None)
print("Coefficients of Beta for p = 4: ")
print(beta_4)
print("Train Error for p = 4: " + str(train_error))
print("Test Error for p = 4: " + str(test_error))

# p = 5
train_sample_count = 300
train_data = features_17_5[:train_sample_count, : y - 1]
train_labels = data[:train_sample_count, y - 1]
```

```
test_data = features_17_5[train_sample_count : x + 1, : y - 1]
test_labels = data[train_sample_count: x + 1, y - 1]
beta_5                                                                    =
inv(np.transpose(train_data).dot(train_data)).dot(np.transpose(train_data)).do
t(train_labels)
train_predictions = train_data.dot(beta_5)
train_error = (np.square(train_labels - train_predictions)).mean(axis = None)
test_predictions = test_data.dot(beta_5)
test_error = (np.square(test_labels - test_predictions)).mean(axis = None)
print("Coefficients of Beta for p = 5: ")
print(beta_5)
print("Train Error for p = 5: " + str(train_error))
print("Test Error for p = 5: " + str(test_error))

shp = np.sort(hp)
plt.plot(shp, (lambda x: beta_1[0] + beta_1[1] * x)(shp), "r", label = "p =
1")
plt.plot(shp, (lambda x: beta_2[0] + beta_2[1] * x + beta_2[2] * (x ** 2))
(shp), "g", label = "p = 2")
plt.plot(shp, (lambda x: beta_3[0] + beta_3[1] * x + beta_3[2] * (x ** 2) +
beta_3[3] * (x ** 3))(shp), "b", label = "p = 3")
plt.plot(shp, (lambda x: beta_4[0] + beta_4[1] * x + beta_4[2] * (x ** 2) +
beta_4[3] * (x ** 3) + beta_4[4] * (x ** 4))(shp), "m", label = "p = 4")
plt.plot(shp, (lambda x: beta_5[0] + beta_5[1] * x + beta_5[2] * (x ** 2) +
beta_5[3] * (x ** 3) + beta_5[4] * (x ** 4) + beta_5[5] * (x ** 5))(shp), "y",
label = "p = 5")
plt.legend()
plt.show()

print("*****************************")
print("Question 1.8: ")
my = data[:, 5]
mean_my = my.mean(0)
stddev_my = my.std(0)
for i in range(x):
    my[i] = (lambda x: (x - mean_my) / stddev_my)(my[i])

features_18_1 = np.column_stack((np.ones(x), hp, my))
features_18_2 = np.column_stack((np.ones(x), hp, hp ** 2, my, my ** 2))
features_18_3 = np.column_stack((np.ones(x), hp, hp ** 2, hp ** 3, my, my **
2, my ** 3))

# p = 1
train_sample_count = 300
train_data = features_18_1[:train_sample_count, : y - 1]
train_labels = data[:train_sample_count, y - 1]
test_data = features_18_1[train_sample_count : x + 1, : y - 1]
test_labels = data[train_sample_count: x + 1, y - 1]
beta                                                                      =
```

```
inv(np.transpose(train_data).dot(train_data)).dot(np.transpose(train_data)).do
t(train_labels)
train_predictions = train_data.dot(beta)
train_error = (np.square(train_labels - train_predictions)).mean(axis = None)
test_predictions = test_data.dot(beta)
test_error = (np.square(test_labels - test_predictions)).mean(axis = None)
print("Coefficients of Beta for p = 1: ")
print(beta)
print("Train Error for p = 1: " + str(train_error))
print("Test Error for p = 1: " + str(test_error))

# p = 2
train_sample_count = 300
train_data = features_18_2[:train_sample_count, : y - 1]
train_labels = data[:train_sample_count, y - 1]
test_data = features_18_2[train_sample_count : x + 1, : y - 1]
test_labels = data[train_sample_count: x + 1, y - 1]
beta                                                                         =
inv(np.transpose(train_data).dot(train_data)).dot(np.transpose(train_data)).do
t(train_labels)
train_predictions = train_data.dot(beta)
train_error = (np.square(train_labels - train_predictions)).mean(axis = None)
test_predictions = test_data.dot(beta)
test_error = (np.square(test_labels - test_predictions)).mean(axis = None)
print("Coefficients of Beta for p = 2: ")
print(beta)
print("Train Error for p = 2: " + str(train_error))
print("Test Error for p = 2: " + str(test_error))

# p = 3
train_sample_count = 300
train_data = features_18_3[:train_sample_count, : y - 1]
train_labels = data[:train_sample_count, y - 1]
test_data = features_18_3[train_sample_count : x + 1, : y - 1]
test_labels = data[train_sample_count: x + 1, y - 1]
beta                                                                         =
inv(np.transpose(train_data).dot(train_data)).dot(np.transpose(train_data)).do
t(train_labels)
train_predictions = train_data.dot(beta)
train_error = (np.square(train_labels - train_predictions)).mean(axis = None)
test_predictions = test_data.dot(beta)
test_error = (np.square(test_labels - test_predictions)).mean(axis = None)
print("Coefficients of Beta for p = 3: ")
print(beta)
print("Train Error for p = 3: " + str(train_error))
print("Test Error for p = 3: " + str(test_error))
```

Question 2

```
import numpy as np
```

```
import matplotlib.pyplot as plt
import sys

def predict(w0, weights, sample):
    tmp = 1 / (1 + np.exp(w0 + np.sum(np.multiply(weights, sample))))
    return 0 if tmp > 0.5 else 1

def gradientAscent(lr, iteration_count, feature_count, sample_count,
train_data, train_labels):
    w0 = 0
    weights = np.zeros((feature_count))
    for it in range(iteration_count):
             y_minus_prediction = np.asarray([train_labels[i] - predict(w0,
weights, train_data[i]) for i in range(sample_count)])
        w0 = w0 + lr * np.sum(y_minus_prediction)
        tmp = train_data * y_minus_prediction[:, np.newaxis]
        tmp = tmp.sum(axis = 0)
        weights = weights + lr * tmp
    return (w0, weights)

def forwardSelection(kf_train_data, kf_train_labels, ic, lr):
    indices = np.asarray([0])
    previous_score = float(sys.maxsize)
    (t1, t2) = kf_train_data[0].shape
    for j in range(1, t2): # for each feature
        print("Feature at index " + str(j) + "...")
        new_indices = np.append(indices, j)
        mse = np.zeros(len(kf_train_data))

        for l in range(len(kf_train_data)): # k-fold
            new_data = kf_train_data[l][:, new_indices]
            new_test_data = kf_test_data[l][:, new_indices]
            (x_train, y_train) = new_data.shape
            (w0, weights) = gradientAscent(lr, ic, y_train, x_train, new_data,
kf_train_labels[l])
            predictions = np.vectorize(lambda x: 0 if x > 0.5 else 1)(1 / (1 +
np.exp(np.multiply(new_test_data, weights).sum(axis = 1) + w0)))
            mse[l] = np.sum(np.power(kf_test_labels[l] - predictions, 2))

        avg_mse = np.sum(mse) * float(1 / len(mse))
        print("avg_mse: " + str(avg_mse))
        print("previous_score: " + str(previous_score))
        if avg_mse < previous_score:
            print("Feature at index " + str(j) + " is accepted.")
            indices = new_indices
            previous_score = avg_mse
        else:
            print("Feature at index " + str(j) + " is rejected.")
    return indices
```

```
def backwardElimination(kf_train_data, kf_train_labels, ic, lr):
    indices = np.arange(len(kf_train_data[0][0]))
    previous_score = -sys.maxsize
    (t1, t2) = kf_train_data[0].shape
    for i in range(t2):
        print("Feature at index " + str(i) + "...")
        new_indices = np.delete(indices, i)
        mse = np.zeros(len(kf_train_data))

        for l in range(len(kf_train_data)): # k-fold
            new_data = kf_train_data[l][:, new_indices]
            new_test_data = kf_test_data[l][:, new_indices]
            (x_train, y_train) = new_data.shape
            (w0, weights) = gradientAscent(lr, ic, y_train, x_train, new_data,
kf_train_labels[l])
            predictions = np.vectorize(lambda x: 0 if x > 0.5 else 1)(1 / (1 +
np.exp(np.multiply(new_test_data, weights).sum(axis = 1) + w0)))
            mse[l] = np.sum(np.power(kf_test_labels[l] - predictions, 2))

        avg_mse = np.sum(mse) * float(1 / len(mse))
        print("avg_mse: " + str(avg_mse))
        print("previous_score: " + str(previous_score))
        if avg_mse > previous_score:
            print("Feature at index " + str(i) + " is accepted.")
        else:
            print("Feature at index " + str(i) + " is rejected.")
            indices = new_indices
            previous_score = avg_mse
    return indices

data = np.loadtxt("../data/ovariancancer.csv", dtype = "float", delimiter=',')
labels = np.loadtxt("../data/ovariancancer_labels.csv", dtype = "float",
delimiter=',')
test_data = np.row_stack((data[:20,:], data[121:141,:]))
test_labels = np.concatenate((labels[:20], labels[121:141]))
train_data = np.row_stack((data[20:121,:], data[141:,:]))
train_labels = np.concatenate((labels[20:121], labels[141:]))
iteration_count = np.array([500, 1000, 1500, 2000, 2500, 3000, 3500, 4000,
4500, 5000])
learning_rate = np.array([0.001, 0.002, 0.005, 0.01, 0.015, 0.02, 0.025,
0.03])
kf_test_data = np.asarray((train_data[:35], train_data[35:70],
train_data[70:105], train_data[105:140], train_data[140:]))
kf_train_data = np.asarray((train_data[35:], np.concatenate((train_data[:35],
train_data[70:])), np.concatenate((train_data[:70], train_data[105:])),
np.concatenate((train_data[:105], train_data[140:])), train_data[:140]))
kf_test_labels = np.asarray((train_labels[:35], train_labels[35:70],
```

```
train_labels[70:105], train_labels[105:140], train_labels[140:]))
kf_train_labels = np.asarray((train_labels[35:],
np.concatenate((train_labels[:35], train_labels[70:])),
np.concatenate((train_labels[:70], train_labels[105:])),
np.concatenate((train_labels[:105], train_labels[140:])), train_labels[:140]))
kf_error_ic = np.zeros(len(iteration_count))
kf_error_lr = np.zeros(len(learning_rate))
index = 0

for ic in iteration_count:
    print("Iteration Count = " + str(ic))
    mse = np.zeros(len(kf_train_data))
    for i in range(len(kf_train_data)):
        print("k-Fold iteration " + str(i) + "...")
        (x_train, y_train) = kf_train_data[i].shape
        (x_test, y_test) = kf_test_data[i].shape
        (w0, weights) = gradientAscent(learning_rate[0], ic, y_train, x_train,
kf_train_data[i], kf_train_labels[i])
               predictions = 1 / (1 + np.exp(np.multiply(kf_test_data[i],
weights).sum(axis = 1) + w0))

        for k in range(len(predictions)):
            predictions[k] = 0 if predictions[k] > 0.5 else 1
            # if predictions[k] != kf_train_labels[i][k]:
            mse[i] = mse[i] + (kf_test_labels[i][k] - predictions[k]) ** 2
    kf_error_ic[index] = np.sum(mse) * 0.2
    index = index + 1

for lr in learning_rate:
    print("Learning Rate = " + str(lr))
    mse = np.zeros(len(kf_train_data))
    for i in range(len(kf_train_data)):
        print("k-Fold iteration " + str(i) + "...")
        (x_train, y_train) = kf_train_data[i].shape
        (x_test, y_test) = kf_test_data[i].shape
            (w0, weights) = gradientAscent(lr, iteration_count[0], y_train,
x_train, kf_train_data[i], kf_train_labels[i])
               predictions = 1 / (1 + np.exp(np.multiply(kf_test_data[i],
weights).sum(axis = 1) + w0))

        for k in range(len(predictions)):
            predictions[k] = 0 if predictions[k] > 0.5 else 1
            # if predictions[k] != kf_train_labels[i][k]:
            mse[i] = mse[i] + (kf_test_labels[i][k] - predictions[k]) ** 2
    kf_error_lr[index] = np.sum(mse) * 0.2
    index = index + 1

# k-Fold Results
kfold_ic = iteration_count[np.argmin(kf_error_ic)]
```

```
kfold_lr = learning_rate[np.argmin(kf_error_lr)]
(kfold_sample_count, kfold_feature_count) = test_data.shape
(kfold_w0, kfold_weights) = gradientAscent(0.001, 2500, kfold_feature_count,
kfold_sample_count, test_data, test_labels)
kfold_predictions = 1 / (1 + np.exp(np.multiply(test_data,
kfold_weights).sum(axis = 1) + kfold_w0))
confusion_matrix = {"tp": 0, "tn": 0, "fp": 0, "fn": 0}

for i in range(len(kfold_predictions)):
    kfold_predictions[i] = 0 if kfold_predictions[i] > 0.5 else 1
    if test_labels[i] == 1 and kfold_predictions[i] == 1: # tp
        confusion_matrix["tp"] = confusion_matrix["tp"] + 1
    elif test_labels[i] == 0 and kfold_predictions[i] == 0: # tn
        confusion_matrix["tn"] = confusion_matrix["tn"] + 1
    elif test_labels[i] == 0 and kfold_predictions[i] == 1: # fp
        confusion_matrix["fp"] = confusion_matrix["fp"] + 1
    elif test_labels[i] == 1 and kfold_predictions[i] == 0: # fn
        confusion_matrix["fn"] = confusion_matrix["fn"] + 1

print("Iteration Count: " + str(kfold_ic) + ", Learning Rate: " +
str(learning_rate))
print("Confusion Matrix: ")
print(confusion_matrix)

# Feature Selection
indices_forward = forwardSelection(kf_train_data, kf_train_labels, kfold_ic,
kfold_lr)
print("Forward Selection...")
new_test_data = test_data[:, indices_forward]
(test_sample, test_feature) = new_test_data.shape
(w0, weights) = gradientAscent(kfold_lr, kfold_ic, test_feature, test_sample,
new_test_data, test_labels)
predictions = 1 / (1 + np.exp(np.multiply(new_test_data, weights).sum(axis =
1) + w0))
confusion_matrix = {"tp": 0, "tn": 0, "fp": 0, "fn": 0}

for i in range(len(predictions)):
    predictions[i] = 0 if predictions[i] > 0.5 else 1
    if test_labels[i] == 1 and predictions[i] == 1: # tp
        confusion_matrix["tp"] = confusion_matrix["tp"] + 1
    elif test_labels[i] == 0 and predictions[i] == 0: # tn
        confusion_matrix["tn"] = confusion_matrix["tn"] + 1
    elif test_labels[i] == 0 and predictions[i] == 1: # fp
        confusion_matrix["fp"] = confusion_matrix["fp"] + 1
    elif test_labels[i] == 1 and predictions[i] == 0: # fn
        confusion_matrix["fn"] = confusion_matrix["fn"] + 1
print("Confusion Matrix: ")
print(confusion_matrix)
```

```
# Backward Elimination
indices_backward = backwardElimination(kf_train_data, kf_train_labels, kfold_ic, kfold_lr)
print("Backward Elimination...")
new_test_data = test_data[:, indices_backward]
(test_sample, test_feature) = new_test_data.shape
(w0, weights) = gradientAscent(kfold_lr, kfold_ic, test_feature, test_sample, new_test_data, test_labels)
predictions = 1 / (1 + np.exp(np.multiply(new_test_data, weights).sum(axis = 1) + w0))
confusion_matrix = {"tp": 0, "tn": 0, "fp": 0, "fn": 0}

for i in range(len(predictions)):
    predictions[i] = 0 if predictions[i] > 0.5 else 1
    if test_labels[i] == 1 and predictions[i] == 1: # tp
        confusion_matrix["tp"] = confusion_matrix["tp"] + 1
    elif test_labels[i] == 0 and predictions[i] == 0: # tn
        confusion_matrix["tn"] = confusion_matrix["tn"] + 1
    elif test_labels[i] == 0 and predictions[i] == 1: # fp
        confusion_matrix["fp"] = confusion_matrix["fp"] + 1
    elif test_labels[i] == 1 and predictions[i] == 0: # fn
        confusion_matrix["fn"] = confusion_matrix["fn"] + 1
print("Confusion Matrix: ")
print(confusion_matrix)
```

Question 3

```
from sklearn import svm
import numpy as np

data = np.asarray(np.loadtxt("../data/UCI_Breast_Cancer.csv", dtype = "int32", delimiter=','))[:, 1:]
labels = data[:, -1]
data = data[:, :-1]

# 3.3
train_data = data[:500,:]
train_labels = labels[:500]
test_data = data[500:,:]
test_labels = labels[500:]
kf_test_data = np.asarray((train_data[:100], train_data[100:200], train_data[200:300], train_data[300:400], train_data[400:]))
kf_train_data = np.asarray((train_data[100:], np.concatenate((train_data[:100], train_data[200:])), np.concatenate((train_data[:200], train_data[300:])), np.concatenate((train_data[:300], train_data[400:])), train_data[:400]))
kf_test_labels = np.asarray((train_labels[:100], train_labels[100:200], train_labels[200:300], train_labels[300:400], train_labels[400:]))
kf_train_labels = np.asarray((train_labels[100:], np.concatenate((train_labels[:100], train_labels[200:])),
```

```
np.concatenate((train_labels[:200],                      train_labels[300:])),
np.concatenate((train_labels[:300], train_labels[400:])), train_labels[:400]))
C = np.array((10**-3, 10**-2, 10**-1, 1, 10**1, 10**2, 10**3))
errors = np.zeros(len(C))
index = 0
for c in C:
    mse = np.zeros(5)
    for i in range(5): # kfold
           model = svm.LinearSVC(C=c, dual=False, fit_intercept=True, loss =
"squared_hinge", max_iter=1000, penalty = "l2", random_state=0, tol=1e-05)
        model.fit(kf_train_data[i], kf_train_labels[i])
        predictions = model.predict(kf_test_data[i])
        mse[i] = np.sum(np.power(kf_test_labels[i] - predictions, 2))
    errors[index] = float(np.sum(mse) / 5)
    index = index + 1

c = C[np.argmin(errors)]
print("Linear SVM with C = " + str(c))

model   =   svm.LinearSVC(C=c,   dual=False,   fit_intercept=True,   loss   =
"squared_hinge", max_iter=1000, penalty = "l2", random_state=0, tol=1e-05)
model.fit(train_data, train_labels)
score = model.score(test_data, test_labels)
predictions = model.predict(test_data)
confusion_matrix = {"tp": 0, "tn": 0, "fp": 0, "fn": 0}

for i in range(len(test_labels)):
    if test_labels[i] == 4 and predictions[i] == 4: # tp
        confusion_matrix["tp"] = confusion_matrix["tp"] + 1
    elif test_labels[i] == 2 and predictions[i] == 2: # tn
        confusion_matrix["tn"] = confusion_matrix["tn"] + 1
    elif test_labels[i] == 2 and predictions[i] == 4: # fp
        confusion_matrix["fp"] = confusion_matrix["fp"] + 1
    elif test_labels[i] == 4 and predictions[i] == 2: # fn
        confusion_matrix["fn"] = confusion_matrix["fn"] + 1
print("Confusion Matrix -> ")
print(confusion_matrix)
print("Accuracy: " + str(score))

# 3.4
train_data = data[:500,:]
train_labels = labels[:500]
test_data = data[500:,:]
test_labels = labels[500:]
kf_test_data     =     np.asarray((train_data[:100],     train_data[100:200],
train_data[200:300], train_data[300:400], train_data[400:]))
kf_train_data                 =                 np.asarray((train_data[100:],
np.concatenate((train_data[:100],                        train_data[200:])),
np.concatenate((train_data[:200],                        train_data[300:])),
```

```
np.concatenate((train_data[:300], train_data[400:])), train_data[:400]))
kf_test_labels   =   np.asarray((train_labels[:100],   train_labels[100:200],
train_labels[200:300], train_labels[300:400], train_labels[400:]))
kf_train_labels               =               np.asarray((train_labels[100:],
np.concatenate((train_labels[:100],                      train_labels[200:])),
np.concatenate((train_labels[:200],                      train_labels[300:])),
np.concatenate((train_labels[:300], train_labels[400:])), train_labels[:400]))
gammA = np.array((2**-4, 2**-3, 2**-2, 2**-1, 1, 2**1, 2**2, 2**3, 2**4))
errors = np.zeros(len(gammA))
index = 0
for gamma in gammA:
    mse = np.zeros(5)
    for i in range(5): # kfold
        model = svm.SVC(kernel = "rbf", gamma = gamma)
        model.fit(kf_train_data[i], kf_train_labels[i])
        predictions = model.predict(kf_test_data[i])
        mse[i] = np.sum(np.power(kf_test_labels[i] - predictions, 2))
    errors[index] = float(np.sum(mse) / 5)
    index = index + 1

gamma = gammA[np.argmin(errors)]
print("SVM with RBF kernel, gamma = " + str(gamma))

model = svm.SVC(kernel = "rbf", gamma = gamma)
model.fit(train_data, train_labels)
score = model.score(test_data, test_labels)
predictions = model.predict(test_data)
confusion_matrix = {"tp": 0, "tn": 0, "fp": 0, "fn": 0}

for i in range(len(test_labels)):
    if test_labels[i] == 4 and predictions[i] == 4: # tp
        confusion_matrix["tp"] = confusion_matrix["tp"] + 1
    elif test_labels[i] == 2 and predictions[i] == 2: # tn
        confusion_matrix["tn"] = confusion_matrix["tn"] + 1
    elif test_labels[i] == 2 and predictions[i] == 4: # fp
        confusion_matrix["fp"] = confusion_matrix["fp"] + 1
    elif test_labels[i] == 4 and predictions[i] == 2: # fn
        confusion_matrix["fn"] = confusion_matrix["fn"] + 1
print("Confusion Matrix -> ")
print(confusion_matrix)
print("Accuracy: " + str(score))
```