Deniz ALKISLAR / 21502930

1.1

X = feature set where each row starts with 1
y = corresponding MPG values for the feature set
β = coefficient matrix

$J_n = (½) (y - Xβ)^T(y – Xβ)$
$= (½) [y^Ty – y^TXβ – β^TX^Ty + β^TX^TXβ]$
$= (½) [y^Ty – 2β^TX^Ty + β^T(X^TX)β]$

By taking derivative of this equation with respect to β, we get:

$d(J_n)/dβ = 0 – (X^Ty) + (X^TXβ)$

By setting the result to zero, we get:

$X^TXβ = X^Ty$
$\mathbf{β = (X^TX)^{-1}X^Ty}$

1.2

Rank gives us the number of linearly independent columns in a matrix. If the rank of a matrix is less than the number of its columns, by applying elementary row operations we can create rows with all zeros. This all-zero rows lead matrix's determinant to be zero. A matrix with determinant value of zero becomes non-invertible.

So the solution in **1.1** is not applicable if some rows of (X^TX) are linearly dependent because we can not take inverse of this multiplication.

The rank of matrix carbig is 7 and it is invertible because it has 7 columns. So the solution in **1.1** is applicable.

1.3

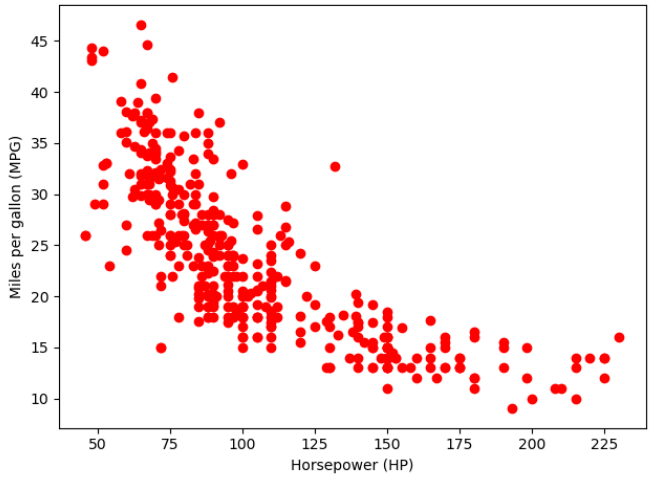
β =
7.12340269e+00
-4.01804611e-01
2.62555848e-03
-7.81396659e-03
-5.47747274e-03
-4.17356632e-02
4.58838891e-01

MSE for training set = 7.35933501
MSE for test set = 35.89671278

1.4

If the regression coefficients are negative this means for every unit increase in X, we expect a β unit decrease in Y, holding all other variables constant.

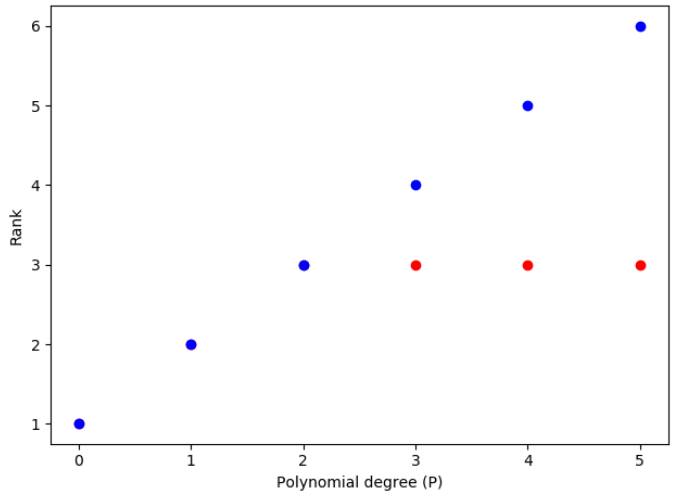
1.5



Graph 1. Miles per gallon vs horsepower graph

As horsepower increases, miles per gallon decreases exponentially.

1.6



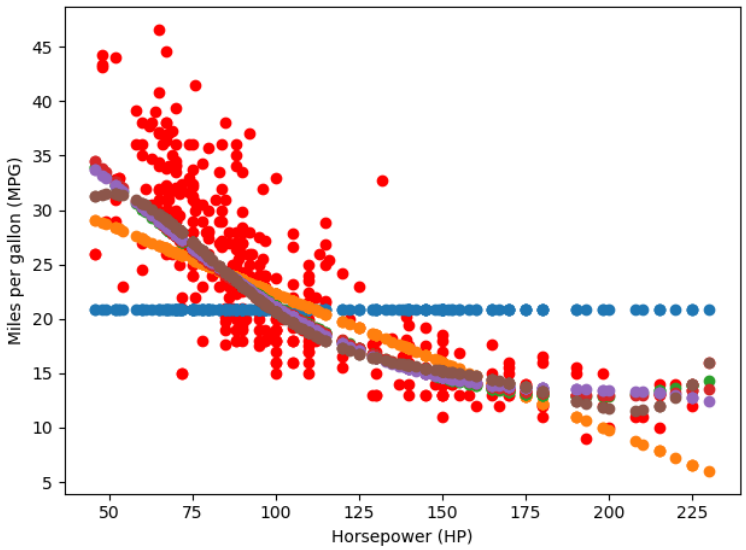
Graph 2. Polynomial degree vs rank graph , red = non-centralized, blue = centralized

Centralizing the feature set increases the rank values. As polynomial degree reaches to 5, rank becomes 6 and feature set becomes linearly independent.

1.7

MSE	P = 0	P = 1	P = 2	P = 3	P = 4	P = 5
Train data	39.94431956	14.34189902	10.83942415	10.81604446	10.77108705	10.3830209
Test data	157.5984514	73.83848308	60.53249696	60.3129133	59.99130159	58.32978865

Table 1. Mean square errors of train and test data of horsepower feature with polynomial regression coefficients of {0, 1, 2, 3, 4, 5}



Graph 3. Regression lines and the train data

1.8

MSE	P = 1	P = 2	P = 3
Train data	14.06676465	9.45156445	9.36677828
Test data	59.22618225	22.29314292	28.63125096

Table 2. Mean square errors of train and test data of horsepower and model year features with polynomial regression coefficients of {1, 2, 3}

The mean square error decreases when we add model year feature.

2.1

Iteration count = 4500, learning rate = 0.02

Prediction / Actual	Positive	Negative
Positive	19	0
Negative	1	20

Table 3. Confusion matrix for the hyper-parameters

2.2

Prediction / Actual	Positive	Negative
Positive	19	0
Negative	1	20

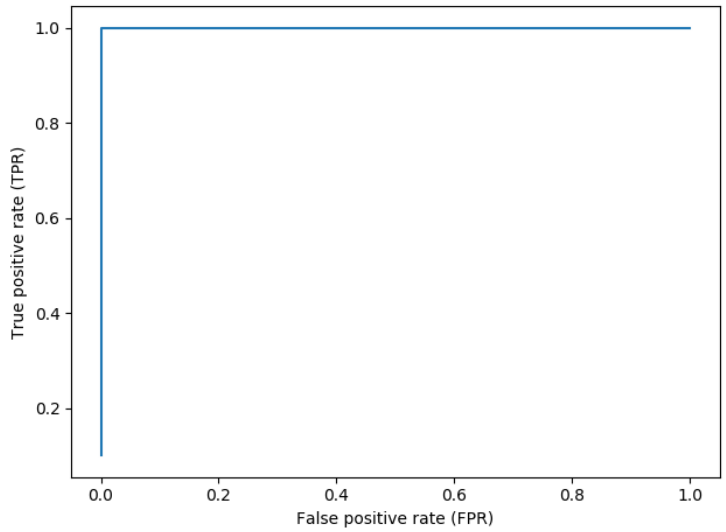
Table 4. Confusion matrix for the backward elimination technique

Prediction / Actual	Positive	Negative
Positive	17	0
Negative	3	20

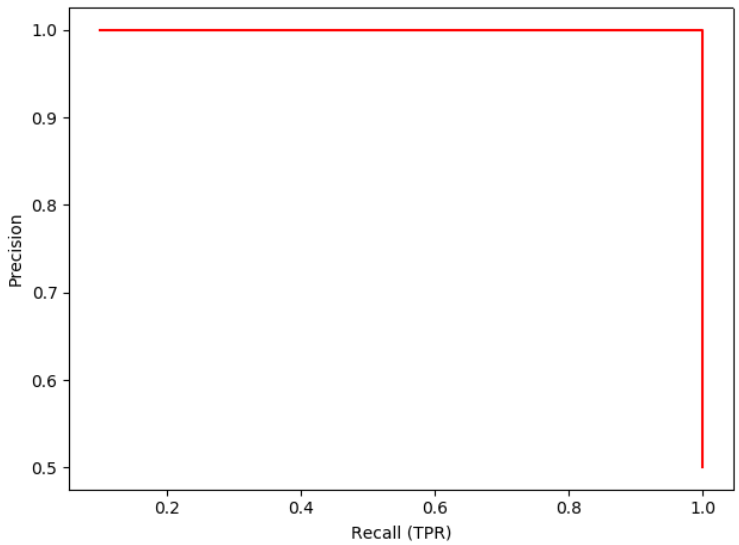
Table 5. Confusion matrix for the forward elimination technique

The two models are not the same. By applying forward elimination technique, our model can reach to a high accuracy with the contribution of few features. However, in the backward elimination technique, without eliminating any features, our model already has a high accuracy. So eliminating features does not make a significant change in the accuracy. The forward elimination is more successful than the backward elimination in this data set because we can get rid of most of the features which do not affect the mean accuracy. By this way, we can reduce our computation times and also obtain a high accuracy.

2.3

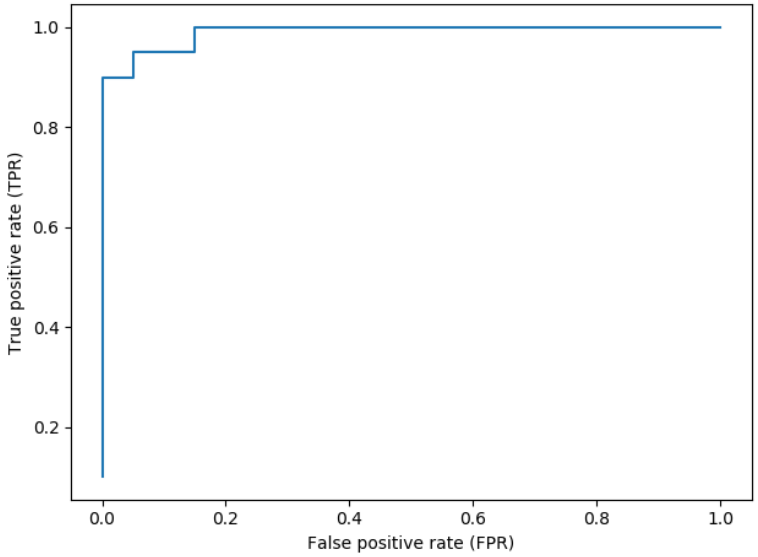


Graph 4. ROC curve for the backward elimination technique

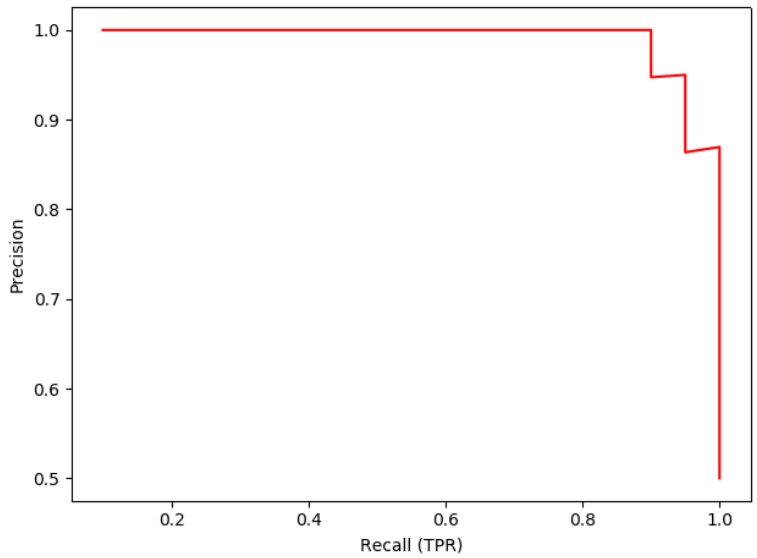


Graph 5. Precision-recall curve for the backward elimination technique

We can conclude that by using backward elimination, our model becomes a perfect model. Because in the ROC curve, the graph touches to a good corner which is the top left corner. Similarly, precision-recall curve touches to a good corner which is the top right corner.



Graph 6. ROC curve for the forward elimination technique



Graph 7. Precision-recall curve for the forward elimination technique

We can conclude that by using forward elimination, our model becomes a good model. Because in the ROC curve, the graph is close to the top left corner. Similarly, precision-recall curve is close to the top right corner.

Deniz ALKISLAR / 21502930

3.1

We have two possible margins. First margin is equal to √2 where $w_1 = (1, 1)$ and $||w_1|| = \sqrt{2}$.

By using these two equations, we can find α_+ and α_-

$$\frac{\partial L_p}{\partial \mathbf{w}} = 0 \Rightarrow \mathbf{w} = \sum_{i=1}^{N} \alpha_i y_i \mathbf{x}_i$$

$$\frac{\partial L_p}{\partial b} = 0 \Rightarrow \sum_{i=1}^{N} \alpha_i y_i = 0$$

$\alpha_-(-1)(1, 1) + \alpha_+(1)(2, 2) + \alpha_+(1)(3, 3) = w_1 = (1, 1)$
Eq. 3.1.1. $-\alpha_- + 5\alpha_+ = 1$

$\alpha_-(-1) + (2)\alpha_+ = 0$
Eq. 3.1.2. $-\alpha_- + 2\alpha_+ = 0$

By using equations 3.1.1 and 3.1.2, we can find $\boldsymbol{\alpha_+ = (1/3)}$ and $\boldsymbol{\alpha_- = (2/3)}$

The other possible margin is equal to 2√2 where $w_2 = (0.5, 0.5)$ and $||w_2|| = \sqrt{2}/2$.

$\alpha_-(-1)(1, 1) + \alpha_+(1)(2, 2) + \alpha_+(1)(3, 3) = w_2 = (0.5, 0.5)$
Eq. 3.1.3. $-\alpha_- + 5\alpha_+ = 0.5$

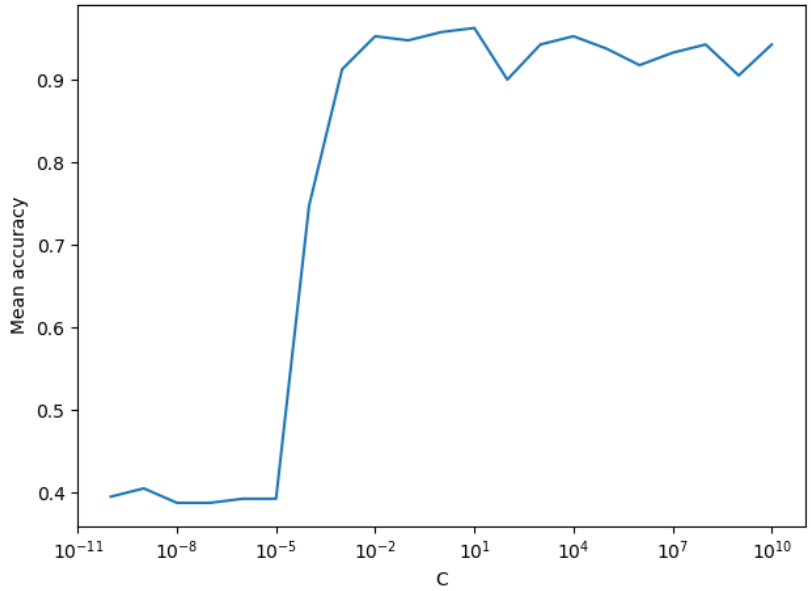
$\alpha_-(-1) + (2)\alpha_+ = 0$
Eq. 3.1.4. $-\alpha_- + 2\alpha_+ = 0$

By using equations 3.1.3 and 3.1.4, we can find $\boldsymbol{\alpha_+ = (1/6)}$ and $\boldsymbol{\alpha_- = (1/3)}$

3.2

If the value of C is set to infinity, it enforces all the constraints and margin becomes a hard margin.

3.3



Graph 8. Mean accuracy vs C

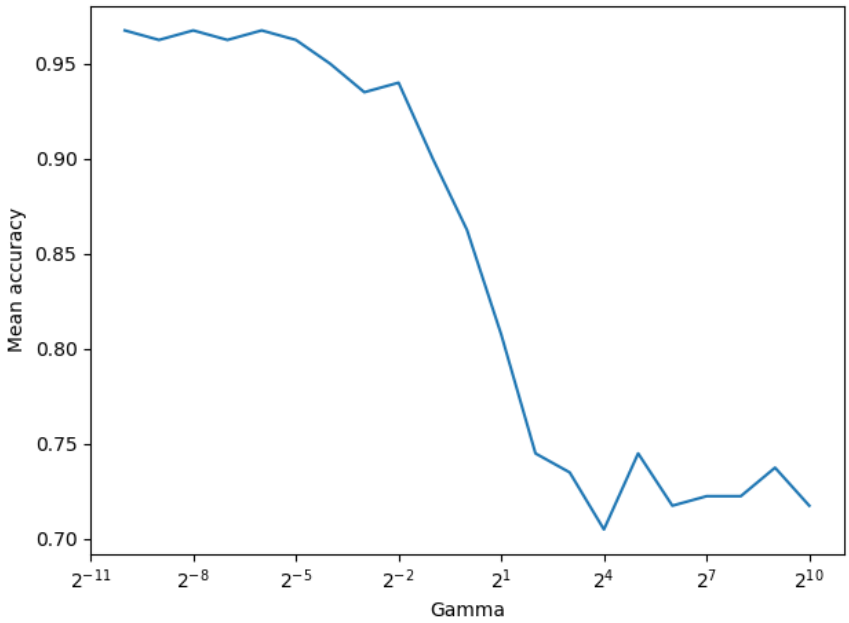
Optimum c = 1

Prediction / Actual	**Positive**	**Negative**
Positive	141	0
Negative	0	42

Table 6. Confusion matrix for the optimum c

I chose k as 5 because I wanted to separate my data as 20% validation and 80% training.

3.4



Graph 9. Mean accuracy vs gamma

Optimum gamma = 0.0009765625

Prediction / Actual	**Positive**	**Negative**
Positive	141	0
Negative	0	42

Table 7. Confusion matrix for the optimum gamma

```
from numpy import genfromtxt, matmul, hstack, ones, mean, std, power, squeeze, asarray
from numpy.linalg import matrix_rank, inv
import matplotlib.pyplot as plt

def find_beta(X, Y):
    beta = inv(matmul(X.T, X))
    beta = matmul(beta, X.T)
    beta = matmul(beta, Y)
    return beta

def find_mse(X, Y, beta):
    mse = Y - matmul(X, beta)
    mse = matmul(mse.T, mse) / len(X)
    return mse

def centralize_matrix(M):
    means = mean(M, axis=0)
    stds = std(M, axis=0)
    for i in range(len(M)):
        for j in range(1, M.shape[1]):
            M.itemset((i, j), (M.item(i, j) - means[j]) / stds[j])

# create matrices and find rank
car_big_csv = genfromtxt('carbig.csv', delimiter=',')
car_big = hstack((ones((len(car_big_csv), 1)), car_big_csv))
car_big_t = car_big[:, :-1].T
mult = matmul(car_big_t, car_big[:, :-1])
print(matrix_rank(mult))

# separate X and y matrices as train and test
car_big_train = car_big[:300][:, :-1]
car_big_test = car_big[300:][:, :-1]
y_train = car_big[:300][:, [-1]]
y_test = car_big[300:][:, [-1]]

# find beta coefficients
beta = find_beta(car_big_train, y_train)
print(beta)

# calculate mean square error for train and test data
mse = find_mse(car_big_train, y_train, beta)
print("Trn mse:", mse)
mse = find_mse(car_big_test, y_test, beta)
print("Tst mse:", mse)

# plot MPG vs. HP
P = 5
MPG = car_big[:, 7]
HP = car_big[:, 3]
plt.plot(HP, MPG, 'ro')
plt.xlabel('Horsepower (HP)')
plt.ylabel('Miles per gallon (MPG)')
plt.show()

# create polynomial HP matrix
HP = car_big[:, [0, 3]]
```

```
for i in range(2, P + 1):
    a = power(HP[:, [1]], i)
    HP = hstack((HP, a))

# plot for non-centralized rank values
a, rank = ([], [])
for i in range(P + 1):
    a.append(i)
    X = HP[:, a]
    rank.append(matrix_rank(matmul(X.T, X)))
plt.plot(range(0, P + 1), rank, 'ro', label='Non-centralized')
plt.xlabel('Polynomial degree (P)')
plt.ylabel('Rank')

# centralize the HP matrix
centralize_matrix(HP)

# plot for centralized rank values
a, rank = ([], [])
for i in range(P + 1):
    a.append(i)
    X = HP[:, a]
    rank.append(matrix_rank(matmul(X.T, X)))
plt.plot(range(0, P + 1), rank, 'bs', label='Centralized')
plt.show()

# find mse for centralized HP for each p value
HP_train = HP[:300]
HP_test = HP[300:]
a = []
print("HP ONLY")
plt.plot(car_big[:, 3], car_big[:, 7], 'ro')
for i in range(P + 1):
    a.append(i)
    beta = find_beta(HP_train[:, a], y_train)
    beta_plot = squeeze(asarray(matmul(HP_train[:, a], beta)))

    plt.plot(car_big[:300][:, 3], beta_plot, 'o')
    plt.xlabel('Horsepower (HP)')
    plt.ylabel('Miles per gallon (MPG)')

    mse = find_mse(HP_train[:, a], y_train, beta)
    print("P(trn) =", i, mse)
    mse = find_mse(HP_test[:, a], y_test, beta)
    print("P(tst) =", i, mse)
plt.show()

# create polynomial MY matrix and centralize it
P = 3
MY = car_big[:, [6]]
for i in range(2, P + 1):
    a = power(MY[:, [0]], i)
    MY = hstack((MY, a))
centralize_matrix(MY)

# create matrix with HP and MY
```

```
HP_MY = hstack((HP[:, [x for x in range(P + 1)]], MY))

# find mse errors for HP_MY matrix
HP_MY_train = HP_MY[:300]
HP_MY_test = HP_MY[300:]
a = [0]
print("HP + MODEL YEAR")
for i in range(1, P + 1):
    a.append(i)
    a.append(i + P)
    beta = find_beta(HP_MY_train[:, a], y_train)
    mse = find_mse(HP_MY_train[:, a], y_train, beta)
    print("P(trn) =", i, mse)
    mse = find_mse(HP_MY_test[:, a], y_test, beta)
    print("P(tst) =", i, mse)
from numpy import genfromtxt, dot, hstack, ones, matrix, zeros, concatenate, exp, ndarray,
lexsort
from numpy.random import shuffle
import matplotlib.pyplot as plt

IT_COUNT = [500, 1000, 1500, 2000, 2500, 3000, 3500, 4000, 4500, 5000]
L_RATE = [0.001, 0.002, 0.005, 0.01, 0.015, 0.02, 0.025, 0.03]

# hypothesis function
def hypo(X, beta):
    z = dot(X, beta)
    return 1 / (1 + exp(-z))

# finding gradients
def find_gradients(X, Y, beta):
    n = len(X)
    gradients = dot(X.T, (hypo(X, beta) - Y))
    return -gradients / n

# gradient ascent
def gradient_ascent(X, Y, it_count, l_rate):
    beta = zeros((X.shape[1], 1))
    for i in range(it_count):
        beta += l_rate * find_gradients(X, Y, beta)
    return beta

# function for finding TP, FP, TN, FN
def confusion_matrix(X, Y, beta, show=False):
    predict = exp(dot(X, beta)) > 1
    comparison = hstack((predict, Y))

    TP, FP, TN, FN = (0, 0, 0, 0)
    for i in range(len(comparison)):
        if comparison.item(i, 0) + comparison.item(i, 1) == 2:
            TP += 1
        elif comparison.item(i, 0) + comparison.item(i, 1) == 0:
            TN += 1
        elif comparison.item(i, 0) == 1 and comparison.item(i, 1) == 0:
            FP += 1
        elif comparison.item(i, 0) == 0 and comparison.item(i, 1) == 1:
            FN += 1
```

```
    if show:
        print("  \t", "(+)", "(-)")
        print("(+)\t", TP, " ", FP)
        print("(-)\t", FN, " ", TN)

    return TP, FP, FN, TN

# function for evaluating the performance
def accuracy_rate(X, Y, beta):
    predict = exp(dot(X, beta)) > 1
    comparison = hstack((predict, Y))
    wrong_predicts = 0
    for i in range(len(predict)):
        if comparison.item(i, 0) != comparison.item(i, 1):
            wrong_predicts += 1
    return 1 - (wrong_predicts / len(Y))

# k-fold function
def k_fold(X_Y, it_count, l_rate):
    k = 5
    fold = int(len(X_Y) / k)
    accuracy = []
    shuffle(X_Y)
    x, y = (X_Y[:, :-1], X_Y[:, [-1]])
    start, end = (0, fold)

    # continue until end of the last fold reaches end of the data
    while end < len(X_Y):
        # separate train and validation data
        x_validation = x[start:end]
        x_train = concatenate((x[:start], x[end:]), axis=0)
        y_validation = y[start:end]
        y_train = concatenate((y[:start], y[end:]), axis=0)

        # train the model and use validation data to find accuracies
        new_beta = gradient_ascent(x_train, y_train, it_count, l_rate)
        accuracy.append(accuracy_rate(x_validation, y_validation, new_beta))

        # move on to next fold
        end += fold
        start += fold
    return sum(accuracy) / len(accuracy)

def find_optimum_param(X_Y):
    accuracies = {}
    for i in range(len(IT_COUNT)):
        for j in range(len(L_RATE)):
            accuracies[(IT_COUNT[i], L_RATE[j])] = k_fold(X_Y, IT_COUNT[i], L_RATE[j])
            print((IT_COUNT[i], L_RATE[j]), accuracies[(IT_COUNT[i], L_RATE[j])])
    return accuracies

def forward_elimination(X_Y, it_count, l_rate):
    selected = [0]
    current_accuracy = 0
    best_set, remaining = ([], [])
```

```
    for i in range(1, X_Y.shape[1] - 1):
        remaining.append(i)

    while True:
        prev_accuracy = current_accuracy
        best_set.clear()
        for i in range(len(remaining)):
            if remaining[i] not in selected:
                current_features = selected + [remaining[i]]
                local_max = k_fold(X_Y[:, current_features + [X_Y.shape[1] - 1]], it_count, l_rate)
                if local_max > current_accuracy:
                    best_set = current_features
                    current_accuracy = local_max
                    print("Local max:", current_features, local_max)

        remaining = [x for x in remaining if x not in best_set]
        selected += [x for x in best_set if x not in selected]
        print("Selected:", selected)
        print("Max accuracy:", current_accuracy)
        if current_accuracy <= prev_accuracy:
            break
    return selected

def backward_elimination(X_Y, it_count, l_rate):
    current_accuracy = 0
    best_set, selected = ([], [])
    for i in range(1, X_Y.shape[1] - 1):
        selected.append(i)

    while True:
        prev_accuracy = current_accuracy
        best_set.clear()
        for i in range(len(selected)):
            print("Feature:", selected[i])
            tested_set = [x for x in selected if x not in [selected[i]]]
            local_max = k_fold(X_Y[:, [0] + tested_set + [X_Y.shape[1] - 1]], it_count, l_rate)
            if local_max > current_accuracy:
                best_set = tested_set
                current_accuracy = local_max
                print("Removed:", selected[i], "Max:", local_max)

        selected = best_set
        print("Current accuracy:", current_accuracy)
        if current_accuracy <= prev_accuracy:
            break
    return selected

# read feature set
ovariancancer = genfromtxt('ovariancancer.csv', delimiter=',')
ovariancancer = hstack((ones((len(ovariancancer), 1)), ovariancancer))
x_train = concatenate((ovariancancer[20:121], ovariancancer[141:]))
x_test = concatenate((ovariancancer[:20], ovariancancer[121:141]))

# read labels
ovariancancer_labels = genfromtxt('ovariancancer_labels.csv')
ovariancancer_labels = (matrix(ovariancancer_labels)).T
```

```
y_train = concatenate((ovariancancer_labels[20:121], ovariancancer_labels[141:]))
y_test = concatenate((ovariancancer_labels[:20], ovariancancer_labels[121:141]))
X_Y = hstack((x_train, y_train))

# find optimum parameters
accuracies = find_optimum_param(X_Y)
max = 0
opt = 0
for key, value in accuracies.items():
    if value > max:
        max = value
        opt = key
print(opt)

# forward elimination and testing
forward_selected = forward_elimination(X_Y, opt[0], opt[1])
beta = gradient_ascent(x_train[:, forward_selected], y_train, opt[0], opt[1])
confusion_matrix(x_test[:, forward_selected], y_test, beta, show=True)

# backward elimination and testing
backward_selected = forward_elimination(X_Y, opt[0], opt[1])
beta = gradient_ascent(x_train[:, backward_selected], y_train, opt[0], opt[1])
confusion_matrix(x_test[:, backward_selected], y_test, beta, show=True)

# curve drawing process
X_Y_test = hstack((hypo(x_test[:, forward_selected], beta), y_test))
temp = X_Y_test.view(ndarray)
X_Y_test = temp[lexsort((temp[:, 0],))]

knob = 0
roc_x, roc_y, pr_x, pr_y = ([], [], [], [])
for m in range(len(X_Y_test)):
    predict = []
    for j in range(len(X_Y_test)):
        if X_Y_test.item(j, 0) >= knob:
            predict.append(1)
        else:
            predict.append(0)
    predict = matrix(predict).T
    comparison = hstack((predict, X_Y_test[:, [-1]]))
    TP, FP, TN, FN = (0, 0, 0, 0)
    for i in range(len(comparison)):
        if comparison.item(i, 0) + comparison.item(i, 1) == 2:
            TP += 1
        elif comparison.item(i, 0) + comparison.item(i, 1) == 0:
            TN += 1
        elif comparison.item(i, 0) == 1 and comparison.item(i, 1) == 0:
            FP += 1
        elif comparison.item(i, 0) == 0 and comparison.item(i, 1) == 1:
            FN += 1
    TPR = TP / (TP + FN)
    FPR = FP / (FP + TN)
    TNR = TN / (TN + FP)
    PPV = TP / (TP + FP)
    roc_x.append(FPR)
    roc_y.append(TPR)
```

```
    pr_y.append(PPV)
    knob = X_Y_test.item(m, 0)

plt.plot(roc_x, roc_y)
plt.xlabel('False positive rate (FPR)')
plt.ylabel('True positive rate (TPR)')
plt.show()

plt.plot(roc_y, pr_y, color='red')
plt.xlabel('Recall (TPR)')
plt.ylabel('Precision')
plt.show()import numpy as np
from sklearn.svm import LinearSVC, SVC
from sklearn.metrics import accuracy_score, confusion_matrix
import matplotlib.pyplot as plt
from math import pow

# k-fold function
def k_fold(X_Y, linear, C=1.0, gamma='auto'):
    k = 5
    fold = int(len(X_Y) / k)
    accuracy = []
    np.random.shuffle(X_Y)
    x, y = (X_Y[:, :-1], X_Y[:, [-1]])
    start, end = (0, fold)

    # continue until end of the last fold reaches end of the data
    while end < len(X_Y):
        # separate train and validation data
        x_validation = x[start:end]
        x_train = np.concatenate((x[:start], x[end:]), axis=0)
        y_validation = y[start:end]
        y_train = np.concatenate((y[:start], y[end:]), axis=0)

        # train the model and use validation data to find accuracies
        if linear:
            svm = LinearSVC(C=C)
            svm.fit(x_train, np.squeeze(np.asarray(y_train)))
            predict = svm.predict(x_validation)
            accuracy.append(accuracy_score(np.squeeze(np.asarray(y_validation)), predict))
        else:
            svm = SVC(gamma=gamma)
            svm.fit(x_train, np.squeeze(np.asarray(y_train)))
            predict = svm.predict(x_validation)
            accuracy.append(accuracy_score(np.squeeze(np.asarray(y_validation)), predict))

        # move on to next fold
        end += fold
        start += fold
    return sum(accuracy) / len(accuracy)

def find_optimum_param(X_Y):
    # find best C
    limit = 10
    max, optimum_c, optimum_g = 0, 0, 0
```

```
    C, accuracies = [pow(10, c) for c in range(-limit, limit + 1)], []
    for c in C:
        accuracy = k_fold(X_Y, True, C=c)
        accuracies.append(accuracy)
        if accuracy > max:
            max = accuracy
            optimum_c = c

    plt.plot(C, accuracies)
    plt.xlabel('C')
    plt.ylabel('Mean accuracy')
    plt.xscale('log', basex=10)
    plt.show()

    max = 0
    accuracies.clear()
    G = [pow(2, g) for g in range(-limit, limit + 1)]
    for g in G:
        accuracy = k_fold(X_Y, False, gamma=g)
        accuracies.append(accuracy)
        if accuracy > max:
            max = accuracy
            optimum_g = g

    plt.plot(G, accuracies)
    plt.xlabel('Gamma')
    plt.ylabel('Mean accuracy')
    plt.xscale('log', basex=2)
    plt.show()

    return optimum_c, optimum_g

# read features and labels
x_y = np.genfromtxt('UCI_Breast_Cancer.csv', delimiter=',')
x_train = x_y[:500][:, [x for x in range(1, 10)]]
x_test = x_y[500:][:, [x for x in range(1, 10)]]
y_train = x_y[:500][:, [-1]]
y_test = x_y[500:][:, [-1]]

X_Y = np.hstack((x_train, y_train))
params = find_optimum_param(X_Y)

# Run the model with optimum c
svm = LinearSVC(C=params[0])
svm.fit(x_train, np.squeeze(np.asarray(y_train)))
predict = svm.predict(x_test)
print(confusion_matrix(np.squeeze(np.asarray(y_test)), predict))
print(params)

# Run the model with optimum gamma
svm = SVC(gamma=params[1])
svm.fit(x_train, np.squeeze(np.asarray(y_train)))
predict = svm.predict(x_test)
print(confusion_matrix(np.squeeze(np.asarray(y_test)), predict))
```