Jesse Li
SID: 23822462
HW 4

1a) $$J(w,a) = (Xw + \alpha 1 - y)^T(Xw + \alpha 1 - y) + \lambda w^T w$$

$$= \left[\sum_{i=1}^{n} (X_{i1}w_1 + X_{i2}w_2 + \cdots + X_{id}w_d + \alpha - y_i)^2\right] + \lambda(w_1^2 + w_2^2 + \cdots + w_d^2)$$

Set dJ/dα = 0 and solve for α:

$$\frac{dJ}{d\alpha} = 2\left[\sum_{i=1}^{n} (X_{i1}w_1 + X_{i2}w_2 + \cdots + X_{id}w_d + \alpha - y_i)\right]$$

$$0 = 2\left[\sum_{i=1}^{n} (X_{i1}w_1 + X_{i2}w_2 + \cdots + X_{id}w_d + \alpha - y_i)\right]$$

$$\hat{\alpha} = \frac{1}{n}\left[\sum_{i=1}^{n} (y_i)\right] - \frac{1}{n}\left[\sum_{i=1}^{n} (X_{i1}w_1 + X_{i2}w_2 + \cdots + X_{id}w_d)\right] = \bar{y} - \bar{x}w = \bar{y}$$

Set dJ/dw = 0 and solve for w:

$$\frac{dJ}{dw_i} = \left[\sum_{j=1}^{n} 2X_{ji}(X_{i1}w_1 + X_{i2}w_2 + \cdots + X_{id}w_d + \alpha - y_i)\right] + 2\lambda w_i$$

$$0 = \left[\sum_{j=1}^{n} X_{ji}\left(X_{j1}w_1 + X_{j2}w_2 + \cdots + X_{jd}w_d\right) + X_{ji}\alpha - X_{ji}y_j)\right] + \lambda w_i$$

$$\sum_{j=1}^{n} X_{ji}y_j = n\bar{x}\alpha + \lambda w_i + \sum_{j=1}^{n} X_{ji}\left(X_{j1}w_1 + X_{j2}w_2 + \cdots + X_{jd}w_d\right)$$

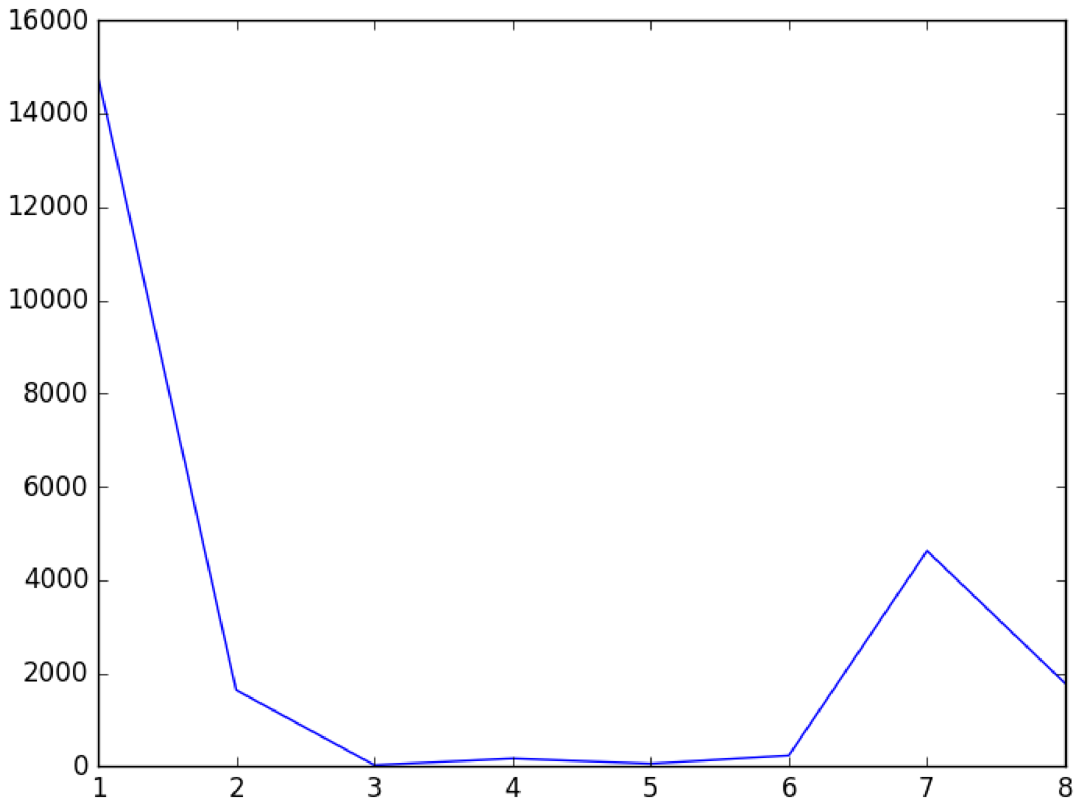
$$X_i^T y = \lambda w_i + X_i^T X_i w_i$$

$$X^T y = \lambda I\hat{w} + X^T X\hat{w} = (\lambda I + X^T X)\hat{w}$$

$$\widehat{w} = (X^T X + \lambda I)^{-1} X^T y$$

b) The RSS on the validation set is 1.39e13. This is about twice what it was in HW3.

c) Regression Coefficient plot:



The regression coefficients have a similar trend to those of HW3 (1, 7, and 8 are the most significant) but differ in that the magnitude of coefficients are smaller since the ridge regression model penalizes large weights.

2)

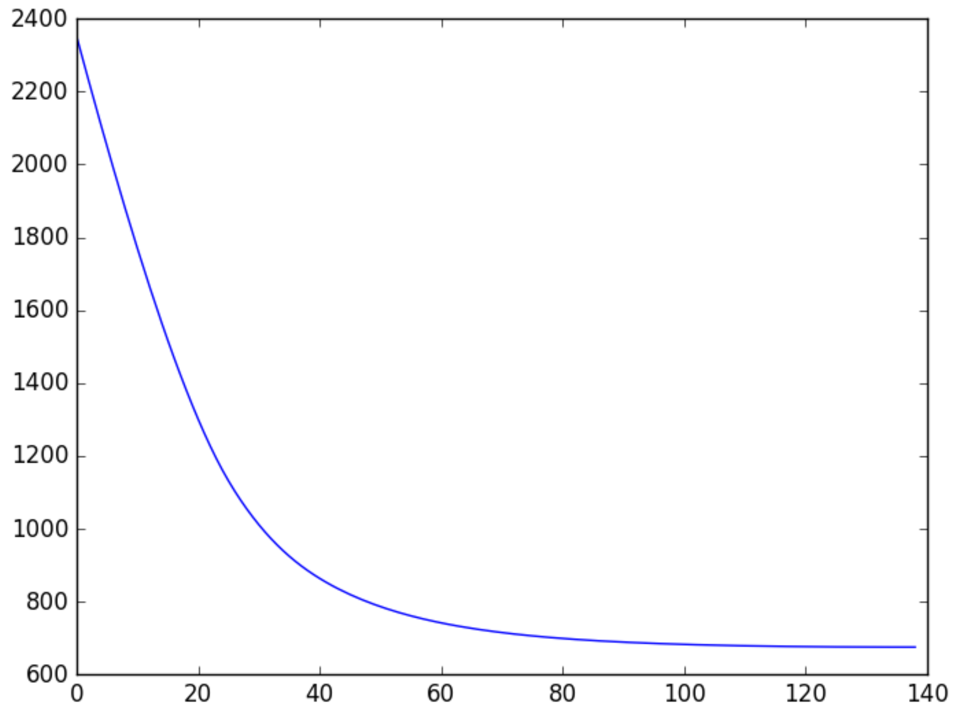
a) R(w0): 1.98837241413
b) mu0: [0.95257413 0.73105858 0.73105858 0.26894142]
c) w1: [-2. 0.94910188 -0.68363271]
d) R(w1): 1.72061709562
e) mu1: [0.89693957 0.54082713 0.56598026 0.15000896]
f) w2: [-1.69083609 1.91981257 -0.83738862]
g) R(w2): 1.85469978479

Code used to generate values:

```
X = np.array([[0,3,1],[1,3,1],[0,1,1],[1,1,1]])
y = np.array([1,1,0,0])
w0 = np.array([-2,1,0])
e = 1
```

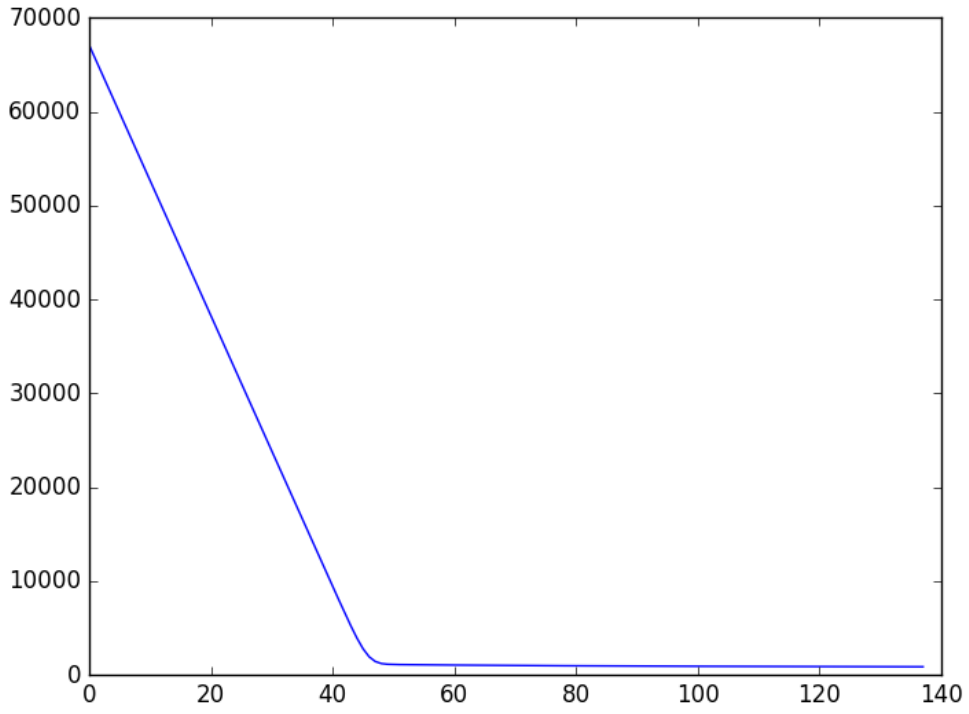
```
def s(gamma):
    return 1/(1+np.exp(-gamma))

def R(w):
    total = 0
    for i in range(X.shape[0]):
        wXi = np.dot(w,X[i])
        total -= (y[i]*math.log(s(wXi))+(1-y[i])*math.log(1 - s(wXi)))
    return total


print('R(w0): ',R(w0))
print('mu0: ', s(np.dot(w0,X.T)))

# iteration 1
total = 0
for i in range(X.shape[0]):
    total += (y[i] - s(np.dot(w0,X[i])))*X[i]
w1 = w0 + e*total

print('w1: ',w1)
print('R(w1): ',R(w1))
print('mu1: ',s(np.dot(w1,X.T)))

# iteration 2
total = 0
for i in range(X.shape[0]):
    total += (y[i] - s(np.dot(w1,X[i])))*X[i]
w2 = w1 + e*total

print('w2: ',w2)
print('R(w2): ',R(w2))
```

3.1) i)



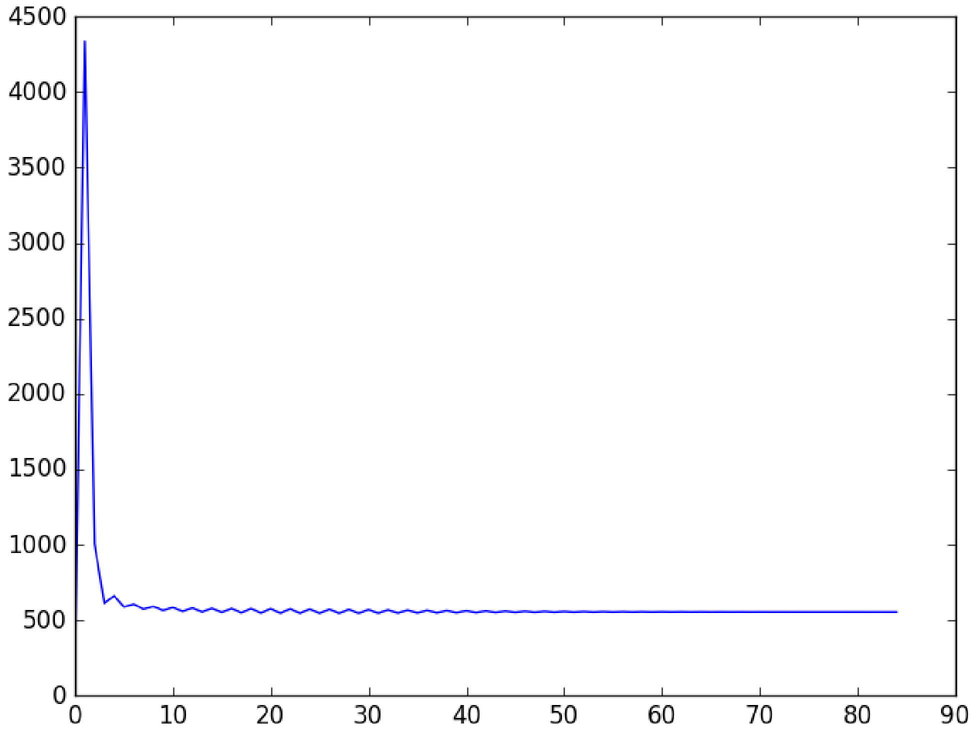
2400
2200
2000
1800
1600
1400
1200
1000
800
600
0
20
40
60
80
100
120
140

ii)

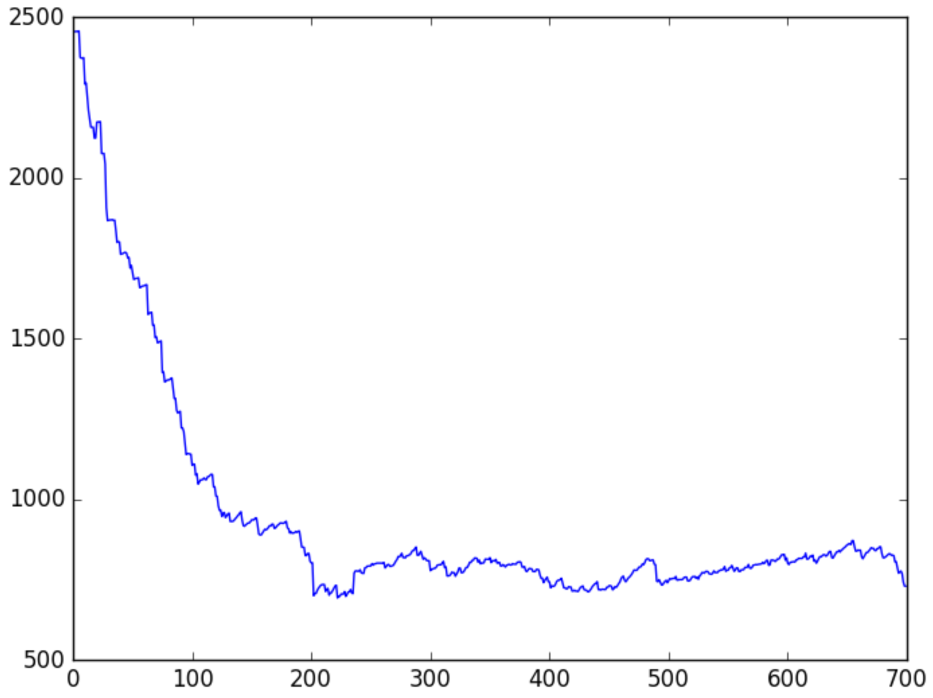


70000
60000
50000
40000
30000
20000
10000
0
0
20
40
60
80
100
120
140

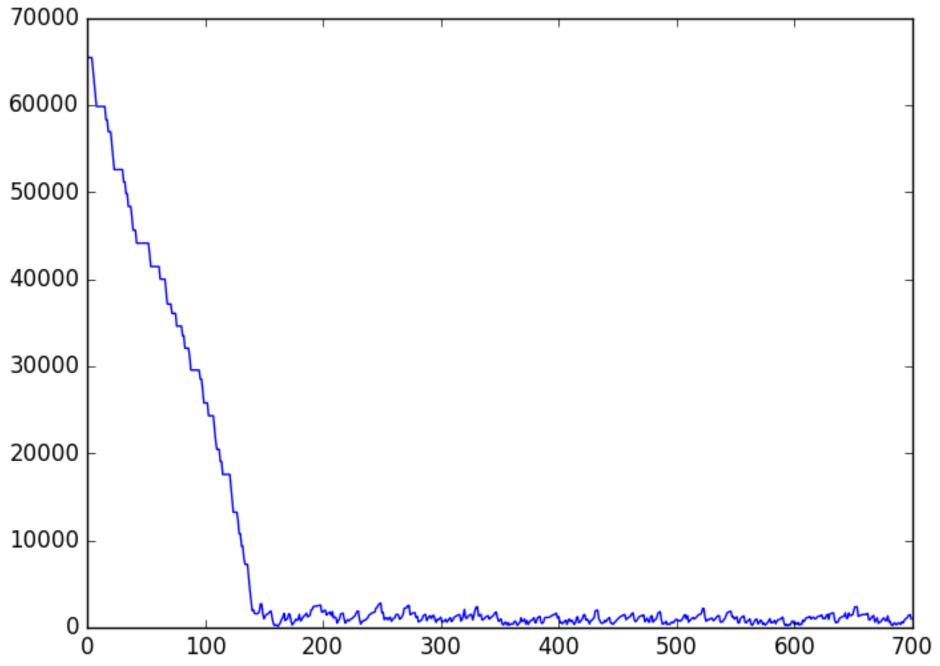
iii)


4500
4000
3500
3000
2500
2000
1500
1000
500
0
0
10
20
30
40
50
60
70
80
90

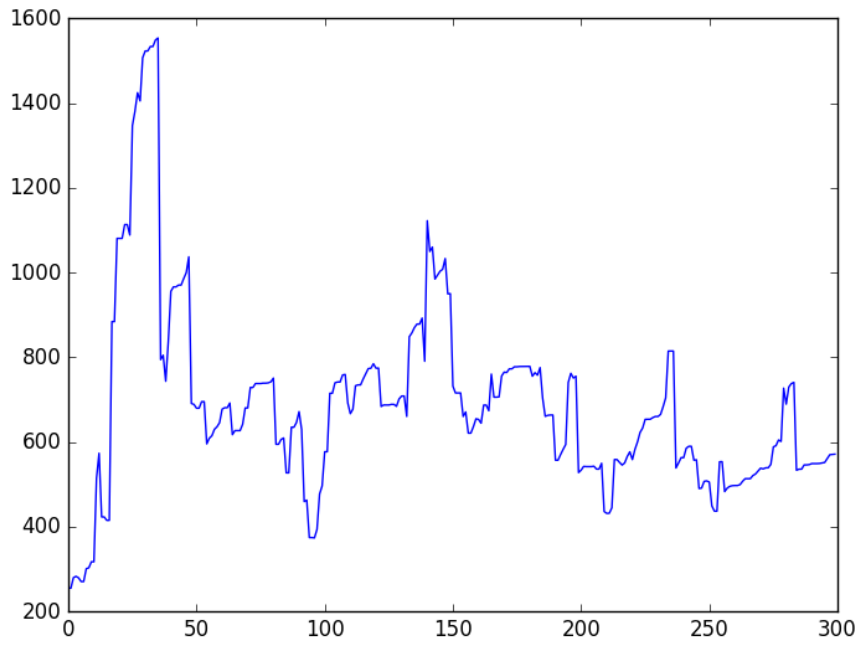
3.2) i)


2500
2000
1500
1000
500
0
100
200
300
400
500
600
700

ii)

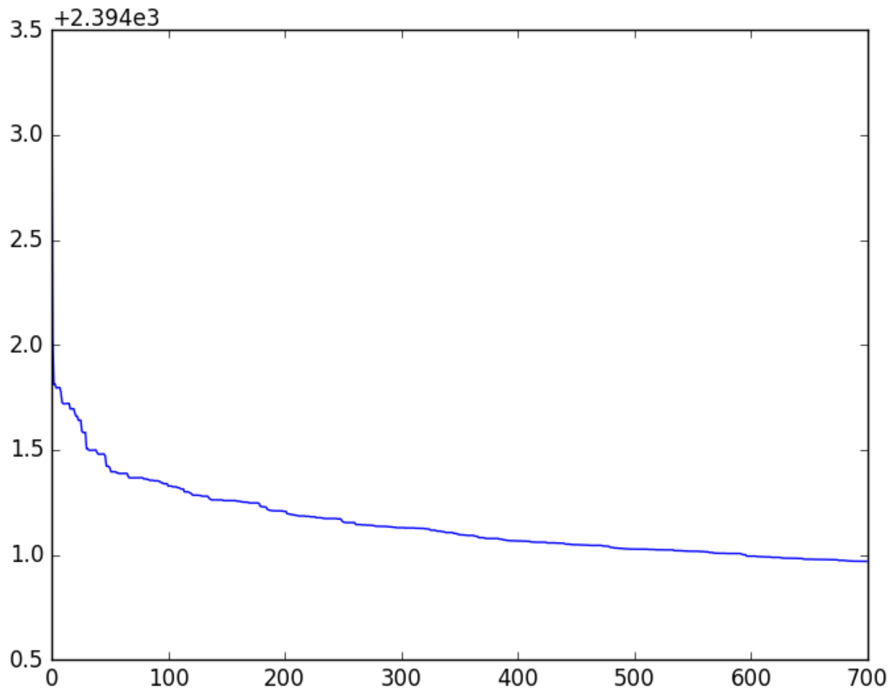


iii)

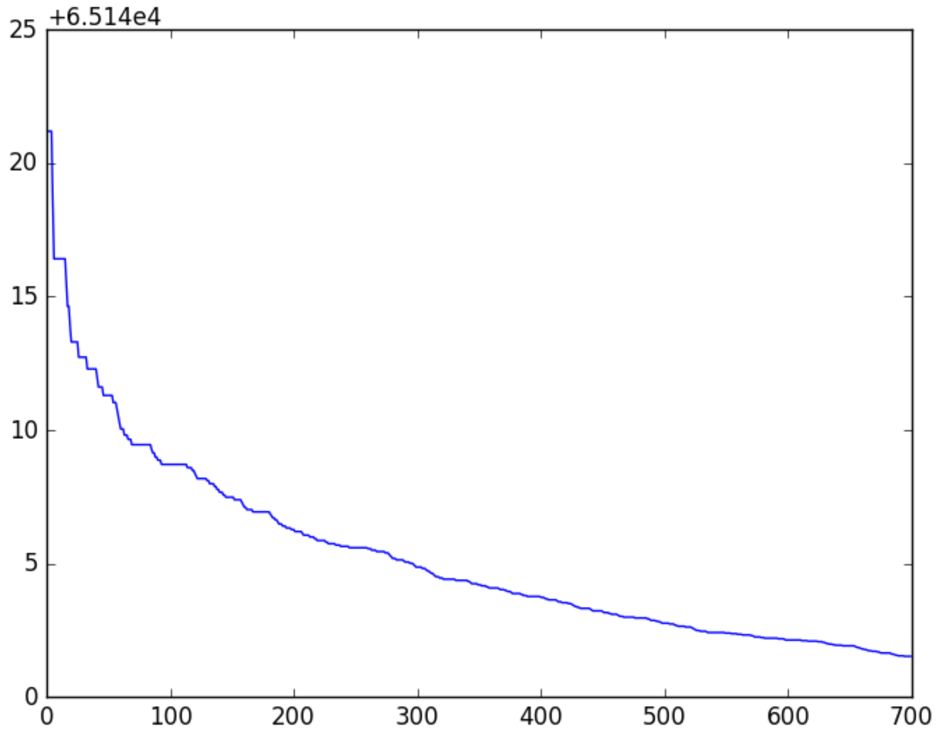


The plots differ from those of part (1) in that they do not strictly decrease, even though there is an overall decreasing trend. The sample chosen for each iteration does not necessarily decrease the risk.

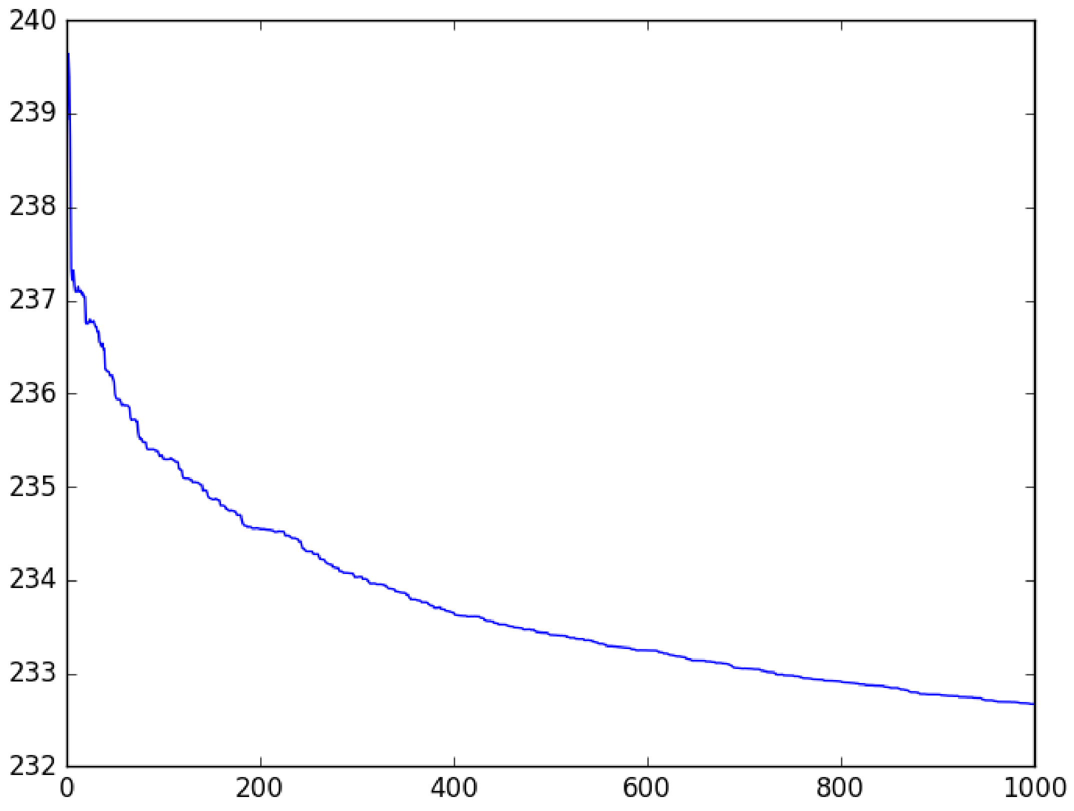
3.3) i)


+2.394e3
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0
100
200
300
400
500
600
700

ii)


+6.514e4
25
20
15
10
5
0
0
100
200
300
400
500
600
700

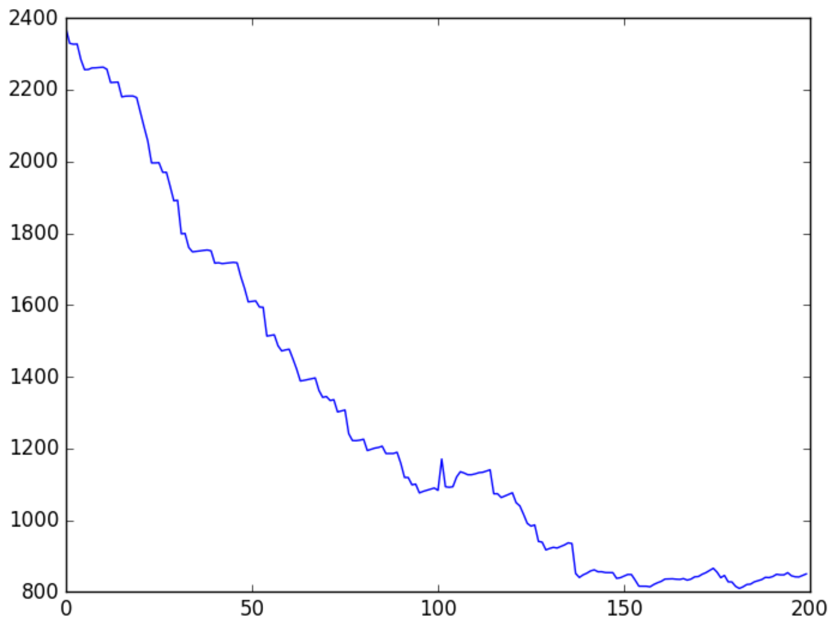
iii)



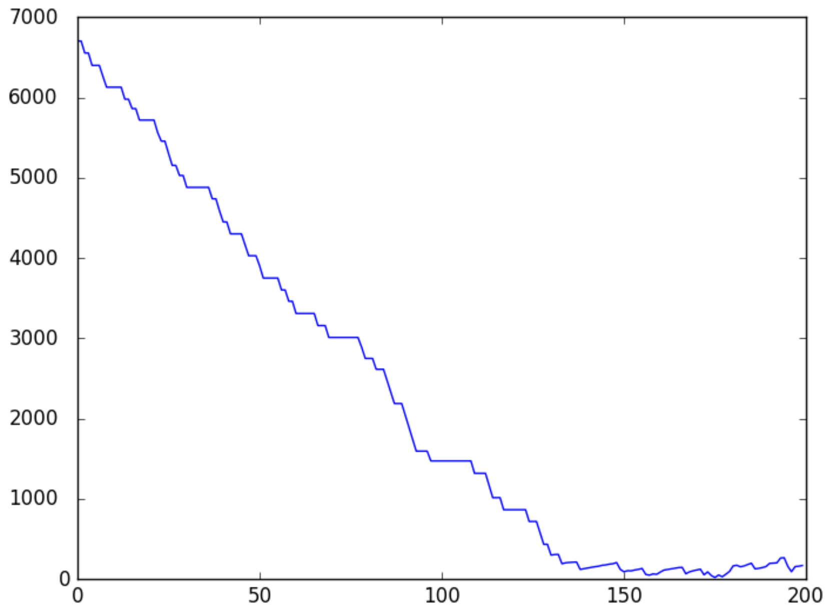
This strategy does not help improve accuracy in this case because it causes the model to converge faster, even when the risk can be further decreased with additional iterations. It does, however, help reduce volatility in the risk updates.

3.4) a) Used preprocessing method 1 since that seemed to give the highest accuracy. Optimal value of ρ was found to be 0.5.

Validation risk vs iterations:



Training risk vs iterations:



b) Training accuracy: 0.701857134
Validation accuracy: 0.712959535

Here we don't see overfitting as the validation error is slightly lower than the training error. It is possible for the quadratic kernel to overfit the data, and if it did overfit the data then we should increase λ to increase performance. This increases the penalty on large weight coefficients, reducing overfitting. For the linear kernel, λ could be decreased for increased performance by preventing underfitting.

3.5) My Kaggle score is 0.79482. I used batch gradient descent with 0 mean and unit variance for preprocessing. My learning rate was 0.0001.

4) a)

$$\tanh(z) = 2s(2z) - 1$$

$$s(z) = \frac{1}{(1 + exp(-z))}$$

$$g(z) = \frac{(tanh(z) + 1)}{2} = \frac{(2s(2z) - 1 + 1)}{2} = s(2z) = \frac{1}{1 + \exp(-2z)}$$

Multiply top and bottom by exp(z), :

$$g(z) = \frac{exp(z)}{\exp(z) + \exp(-z)} = \frac{exp(z)}{\exp(z) + \exp(-z)} - \frac{1}{2} + \frac{1}{2}$$

$$= \frac{2exp(z)}{2(\exp(z) + \exp(-z))} - \frac{\exp(z) + \exp(-z)}{2(\exp(z) + \exp(-z))} + \frac{1}{2}$$

$$= \frac{\exp(z) - \exp(-z)}{2(\exp(z) + \exp(-z))} + \frac{1}{2}$$

b)

$$g(z) = \frac{\tanh(z) + 1}{2} = \frac{(2s(2z) - 1 + 1)}{2} = s(2z)$$

d(s(z))/dz = s(z)(1-s(z))

$$g'(z) = \frac{d(s(2z))}{dz} = 2s(2z)\big(1 - s(2z)\big) = 2\left(\frac{\tanh(z)}{2} + \frac{1}{2}\right)\left(1 - \frac{\tanh(z)}{2} - \frac{1}{2}\right)$$

$$= (\tanh(z) + 1)\left(\frac{1}{2} - \frac{\tanh(z)}{2}\right) = \frac{1}{2}\tanh(z) - \frac{1}{2}\tanh(z) - \frac{1}{2}(\tanh(z))^2 + \frac{1}{2}$$

$$= \frac{1}{2} - \frac{1}{2}(\tanh(z))^2$$

c)

$$\nabla_w J(w) = \sum_{i=1}^{n} \frac{y_i X_i(\frac{1}{2}(1-\tanh^2(X_i w)))}{(\tanh(X_i w)+1)/2} + \frac{(1-y_i)X_i(-\frac{1}{2}(1-\tanh^2(X_i w)))}{1-\ (\tanh(X_i w)+1)/2}$$

$$= \sum_{i=1}^{n} \ [y_i(1-\tanh(X_i w)) - (1-y_i)(1+\tanh(X_i w))]X_i$$

$$= \sum_{i=1}^{n} \ [2y_i - 1 - \tanh(X_i w)]X_i$$

$$w \leftarrow w - \lambda \nabla_w J(w) = w + \lambda \sum_{i=1}^{n} \ [2y_i - 1 - \tanh(X_i w)]X_i$$

5) The linear SVM currently does not utilize the new feature well because the spike of spam messages occurs both before and after midnight. The new feature is stored as the number of seconds since the previous midnight, which means the additional spike in spam messages will consist of spam samples that have very high values (right before midnight) or very low values (right after midnight) for the new feature. This makes it impossible to linearly separate the low values from the high values. One way to improve the results would be to use a quadratic kernel and shift the feature vector by 12 hours so that the messages right after midnight have large negative values and the messages just before midnight have large positive values. Squaring this new feature with the quadratic kernel will result in a parabola in which the spam messages near midnight are all at the top, making it possible for the hyperplane to separate the spam messages from the ham messages at the bottom of the parabola.

CODE:

```
import scipy.io
import numpy as np
import random
import matplotlib.pyplot as plt
import math
from sklearn import preprocessing
import sklearn
```

```
# Problem 1
'''
housing_data = scipy.io.loadmat("housing_dataset/housing_data.mat")
Xtrain= housing_data['Xtrain']
Xvalidate= housing_data['Xvalidate']
Ytrain= housing_data['Ytrain']
Yvalidate= housing_data['Yvalidate']

# bias = np.ones((Xtrain.shape[0],1))
# Xtrain = np.hstack((Xtrain,bias))


#Xtrain = Xtrain - np.tile(np.mean(Xtrain, axis = 0), (Xtrain.shape[0],1)) # center x
Ytrain = np.ndarray.astype(Ytrain.ravel(), dtype = float)
Yvalidate = np.ndarray.astype(Yvalidate.ravel(), dtype = float)

lambdas = [pow(10,i)*1e-20 for i in range(40)]
# lambdas = [1e9]
best_lambda = 0
best_error = float('inf')


indices = list(range(Xtrain.shape[0]))
random.shuffle(indices)

# find best lambda
errors = []
for l in lambdas:
    chosen_x = []
    chosen_y = []
    for i in range(10):
        subset = indices[i*Xtrain.shape[0]//10:(i+1)*Xtrain.shape[0]//10]
        subsetX = []
        subsetY = []
        for j in subset:
            subsetX.append(Xtrain[j])
            subsetY.append(Ytrain[j])
        chosen_x.append(np.array(subsetX))
        chosen_y.append(np.array(subsetY))
    loss = 0
    for i in range(10):
        x_validate = chosen_x[i]
        y_validate = chosen_y[i]
        x = np.vstack(chosen_x[:i] + chosen_x[i+1:])
        x = x - np.tile(np.mean(x, axis = 0), (x.shape[0],1))
        y = np.hstack(chosen_y[:i] + chosen_y[i+1:])
        w = np.linalg.solve(np.dot(x.T,x)+l*np.identity(x.shape[1]),np.dot(x.T,y))
        # w = np.dot(np.linalg.inv(np.dot(x.T,x) + l*np.identity(x.shape[1])),np.dot(x.T,y))
        a = np.mean(y)
        predictions = np.dot(x_validate,w)+a*np.ones(x_validate.shape[0])
        loss += np.dot((predictions - y_validate).T,(predictions-y_validate)) + l*np.dot(w.T,w)
    errors.append(loss/10)
    print(l, loss/10)
    if loss/10 < best_error:
        best_error = loss/10
        best_lambda = l
plt.semilogx(lambdas,errors)
plt.show()

# part ii and part iii
l = 1e6
Xtrain = Xtrain - np.tile(np.mean(Xtrain, axis = 0), (Xtrain.shape[0],1)) # center x
x = Xtrain
y = Ytrain
w = np.linalg.solve(np.dot(x.T,x)+l*np.identity(x.shape[1]),np.dot(x.T,y))
a = np.mean(y)
predictions = np.dot(Xvalidate,w)+a*np.ones(Xvalidate.shape[0])
RSS = np.dot((predictions - Yvalidate).T,(predictions-Yvalidate))# + l*np.dot(w.T,w)
plt.figure()
plt.plot(range(1,9), abs(w))
plt.show()
'''

# problem 2
'''
X = np.array([[0,3,1],[1,3,1],[0,1,1],[1,1,1]])
y = np.array([1,1,0,0])
w0 = np.array([-2,1,0])
e = 1

def s(gamma):
    return 1/(1+np.exp(-gamma))

def R(w):
    total = 0
    for i in range(X.shape[0]):
        wXi = np.dot(w,X[i])
        total -= (y[i]*math.log(s(wXi))+(1-y[i])*math.log(1 - s(wXi)))
    return total
```

```
print('R(w0): ',R(w0))
print('mu0: ', s(np.dot(w0,X.T)))

# iteration 1
total = 0
for i in range(X.shape[0]):
    total += (y[i] - s(np.dot(w0,X[i])))*X[i]
w1 = w0 + e*total

print('w1: ',w1)
print('R(w1): ',R(w1))
print('mu1: ',s(np.dot(w1,X.T)))

# iteration 2
total = 0
for i in range(X.shape[0]):
    total += (y[i] - s(np.dot(w1,X[i])))*X[i]
w2 = w1 + e*total

print('w2: ',w2)
print('R(w2): ',R(w2))
'''

# problem 3

spam_data = scipy.io.loadmat("spam_dataset/spam_data.mat")
X = spam_data['training_data']
y = spam_data['training_labels'].ravel()
test_data= spam_data['test_data']

indices = list(range(y.shape[0]))
random.shuffle(indices)
validation_x = []
validation_y = []
training_x = []
training_y = []
for i in indices[:y.shape[0]//3]:
    validation_x.append(X[i])
    validation_y.append(y[i])
for i in indices[y.shape[0]//3:]:
    training_x.append(X[i])
    training_y.append(y[i])
X = np.array(training_x)
y = np.array(training_y)
validation_x = np.array(validation_x)
validation_y = np.array(validation_y)
```

```
def s(gamma):
    return 1/(1+np.exp(-gamma))

def R(w,X):
    total = 0
    for i in range(X.shape[0]):
        sigmoid_wXi = s(np.dot(w,X[i]))
        total -= (y[i]*math.log(sigmoid_wXi)+(1-y[i])*math.log(max((1 - sigmoid_wXi),1)))
    return total

# Problem 3.1
'''
def batch(w, X, e = 0.0001, processing = 0, tolerance = 0.01, max_iter = 200):
    if processing == 1:
        X = preprocessing.scale(X) # part (i)
    elif processing == 2:
        X = np.vectorize(lambda Xij: math.log(Xij + 0.1))(X) # part (ii)
    elif processing == 3:
        X = sklearn.preprocessing.binarize(X,0.01) # part (iii)
    risk_list = [R(w,X)]
    total = 0
    for i in range(X.shape[0]):
        total += (y[i] - s(np.dot(w,X[i])))*X[i]
    w = w + e*total
    risk_list.append(R(w,X))
    while abs(risk_list[-1] - risk_list[-2]) > tolerance and len(risk_list) < max_iter:
        total = 0
        for i in range(X.shape[0]):
            total += (y[i] - s(np.dot(w,X[i])))*X[i]
        w = w + e*total
        risk_list.append(R(w,X))
    return w, risk_list

w0 = np.ones(X.shape[1])

w,risks = batch(w0, X, processing = 1)
plt.figure()
plt.plot(range(len(risks)),risks)
accuracy = np.count_nonzero(sklearn.preprocessing.binarize(s(np.dot(validation_x,w)),0.5) ==
validation_y)/validation_y.shape[0]
print(accuracy)

w,risks = batch(w0, X, e = 1e-5, tolerance = 1, processing = 2)
plt.figure()
plt.plot(range(len(risks)),risks)
accuracy = np.count_nonzero(sklearn.preprocessing.binarize(s(np.dot(validation_x,w)),0.5) ==
validation_y)/validation_y.shape[0]
print(accuracy)

w,risks = batch(w0, X, e = 1e-2, processing = 3)
plt.figure()
plt.plot(range(len(risks)),risks)
accuracy = np.count_nonzero(sklearn.preprocessing.binarize(s(np.dot(validation_x,w)),0.5) ==
validation_y)/validation_y.shape[0]
print(accuracy)
'''

#Problem 3.2
'''
def stochastic(w, X, e = 1e-1, processing = 0, max_iter = 1000):
    if processing == 1:
        X = preprocessing.scale(X) # part (i)
    elif processing == 2:
        X = np.vectorize(lambda Xij: math.log(Xij + 0.1))(X) # part (ii)
    elif processing == 3:
        X = sklearn.preprocessing.binarize(X,0.01) # part (iii)
    risk_list = [R(w,X)]
    idx = random.randint(0,(y.shape[0]-1))
    w = w + e*np.dot((y[idx] - s(np.dot(X[idx],w))),X[idx])
    risk_list.append(R(w,X))
    while len(risk_list) < max_iter:
        idx = random.randint(0,y.shape[0]-1)
        w = w + e*np.dot((y[idx] - s(np.dot(X[idx],w))),X[idx])
        risk_list.append(R(w,X))
    return w, risk_list

w0 = np.ones(X.shape[1])
w,risks = stochastic(w0, X, processing = 1, max_iter = 700)
plt.figure()
plt.plot(range(len(risks)),risks)
accuracy = np.count_nonzero(sklearn.preprocessing.binarize(s(np.dot(validation_x,w)),0.5) ==
validation_y)/validation_y.shape[0]
print(accuracy)

w,risks = stochastic(w0, X, e = 1e-2, processing = 2, max_iter = 700)
plt.figure()
plt.plot(range(len(risks)),risks)
accuracy = np.count_nonzero(sklearn.preprocessing.binarize(s(np.dot(validation_x,w)),0.5) ==
validation_y)/validation_y.shape[0]
print(accuracy)
```

```
w,risks = stochastic(w0, X, e = 1, processing = 3, max_iter = 300)
plt.figure()
plt.plot(range(len(risks)),risks)
accuracy = np.count_nonzero(sklearn.preprocessing.binarize(s(np.dot(validation_x,w)),0.5) ==
validation_y)/validation_y.shape[0]
print(accuracy)
'''

# Problem 3.3
'''
def decreasing_e(w, X, e = 1e-1, processing = 0, max_iter = 1000):
    if processing == 1:
        X = preprocessing.scale(X) # part (i)
    elif processing == 2:
        X = np.vectorize(lambda Xij: math.log(Xij + 0.1))(X) # part (ii)
    elif processing == 3:
        X = sklearn.preprocessing.binarize(X,0.01) # part (iii)
    risk_list = [R(w,X)]
    idx = random.randint(0,(y.shape[0]-1))
    w = w + e*np.dot((y[idx] - s(np.dot(X[idx],w))),X[idx])
    risk_list.append(R(w,X))
    while len(risk_list) < max_iter:
        idx = random.randint(0,y.shape[0]-1)
        w = w + e*np.dot((y[idx] - s(np.dot(X[idx],w))),X[idx])/len(risk_list)
        risk_list.append(R(w,X))
    return w, risk_list


w,risks = decreasing_e(w0, X, e = 1e-3, processing = 1, max_iter = 700)
plt.figure()
plt.plot(range(len(risks)),risks)
accuracy = np.count_nonzero(sklearn.preprocessing.binarize(s(np.dot(validation_x,w)),0.5) ==
validation_y)/validation_y.shape[0]
print(accuracy)

w,risks = decreasing_e(w0, X, e = 1e-4, processing = 2, max_iter = 700)
plt.figure()
plt.plot(range(len(risks)),risks)
accuracy = np.count_nonzero(sklearn.preprocessing.binarize(s(np.dot(validation_x,w)),0.5) ==
validation_y)/validation_y.shape[0]
print(accuracy)

w,risks = decreasing_e(w0, X, e = 0.1, processing = 3)
plt.figure()
plt.plot(range(len(risks)),risks)
```

```
accuracy = np.count_nonzero(sklearn.preprocessing.binarize(s(np.dot(validation_x,w)),0.5) ==
validation_y)/validation_y.shape[0]
print(accuracy)
'''

# Problem 3.4 (a)
'''
def k(x,z,rho):
    return np.dot((np.dot(x.T,z)+rho).T, np.dot(x.T,z)+rho)

def kernal_R(a,x):
    gamma = np.dot(np.dot(a,x),x.T)
    r_i = y * np.log(s(gamma)) + (1-y) * np.log(1 - s(gamma))
    return -np.sum(r_i)

def stochastic_kernel(a, X, l=1e-3, e=0.1, rho = 0.5, tol = 1e-8, max_iter=200):
    K=np.zeros((X.shape[0],X.shape[0]))
    z1 = X[random.randint(0,y.shape[0]-1)]
    for i in range(K.shape[0]):
        for j in range(K.shape[0]):
            K[i][j] = k(X[i],z1,rho)
    r_list = [kernal_R(a, X)]
    delta = float('inf')
    while(len(r_list) <max_iter):
        idx = random.randint(0,y.shape[0]-1)
        a = a-e*l*a[idx]+e*(y[idx]*s(np.dot(K,a)[idx]))
        r = kernal_R(a, X)
        delta = r_list[-1] - r
        r_list.append(r)
    return a, r_list, K

a = np.zeros(y.shape[0])
X = preprocessing.scale(X)
a_final,r_list,K = stochastic_kernel(a,X)
accuracy = np.count_nonzero(sklearn.preprocessing.binarize(s(np.dot(K,a_final)),0.5) ==
y)/y.shape[0]

# Find best rho
# for rho in [i/10 for i in range(1,10)]:
#     a_final,r_list,K = stochastic_kernel(a,X, rho = rho)
#     accuracy = np.count_nonzero(sklearn.preprocessing.binarize(s(np.dot(K,a_final)),0.5) ==
y)/y.shape[0]
'''

# Problem 3.4 (b)
'''
```

```
def k(x,z,rho):
    return np.dot(x.T,z)+rho

def kernal_R(a,x):
    gamma = np.dot(np.dot(a,x),x.T)
    r_i = y * np.log(s(gamma)) + (1-y) * np.log(1 - s(gamma))
    return -np.sum(r_i)

def stochastic_kernel(a, X, l=1e-3, e=0.1, rho = 0.5, tol = 1e-8, max_iter=200):
    K=np.zeros((X.shape[0],X.shape[0]))
    z1 = X[random.randint(0,y.shape[0]-1)]
    for i in range(K.shape[0]):
        for j in range(K.shape[0]):
            K[i][j] = k(X[i],z1,rho)
    r_list = [kernal_R(a, X)]
    delta = float('inf')
    while(len(r_list) <max_iter):
        idx = random.randint(0,y.shape[0]-1)
        a = a-e*l*a[idx]+e*(y[idx]*s(np.dot(K,a)[idx]))
        r = kernal_R(a, X)
        delta = r_list[-1] - r
        r_list.append(r)
    return a, r_list, K

a = np.zeros(y.shape[0])
X = preprocessing.scale(X)
a_final,r_list,K = stochastic_kernel(a,X)
accuracy = np.count_nonzero(sklearn.preprocessing.binarize(s(np.dot(K,a_final)),0.5) ==
y)/y.shape[0]

# Find best rho
# for rho in [i/10 for i in range(1,10)]:
#     a_final,r_list,K = stochastic_kernel(a,X, rho = rho)
#     accuracy = np.count_nonzero(sklearn.preprocessing.binarize(s(np.dot(K,a_final)),0.5) ==
y)/y.shape[0]
'''

# Problem 3.5
'''
def batch(w, X, e = 0.0001, processing = 0, tolerance = 0.01, max_iter = 200):
    if processing == 1:
        X = preprocessing.scale(X) # part (i)
    elif processing == 2:
        X = np.vectorize(lambda Xij: math.log(Xij + 0.1))(X) # part (ii)
    elif processing == 3:
        X = sklearn.preprocessing.binarize(X,0.01) # part (iii)
    risk_list = [R(w,X)]
    total = 0
    for i in range(X.shape[0]):
        total += (y[i] - s(np.dot(w,X[i])))*X[i]
    w = w + e*total
    risk_list.append(R(w,X))
    while abs(risk_list[-1] - risk_list[-2]) > tolerance and len(risk_list) < max_iter:
        total = 0
        for i in range(X.shape[0]):
            total += (y[i] - s(np.dot(w,X[i])))*X[i]
        w = w + e*total
        risk_list.append(R(w,X))
    return w, risk_list

w0 = np.ones(X.shape[1])

w,risks = batch(w0, X, processing = 1)
plt.figure()
plt.plot(range(len(risks)),risks)
predictions = sklearn.preprocessing.binarize(s(np.dot(test_data,w)),0.5)

numbers = np.arange(predictions.ravel().shape[0]) + 1
test_predict = np.vstack((numbers,predictions))
np.savetxt("spam.csv", test_predict.transpose(), delimiter=",",fmt = '%u')
'''
```