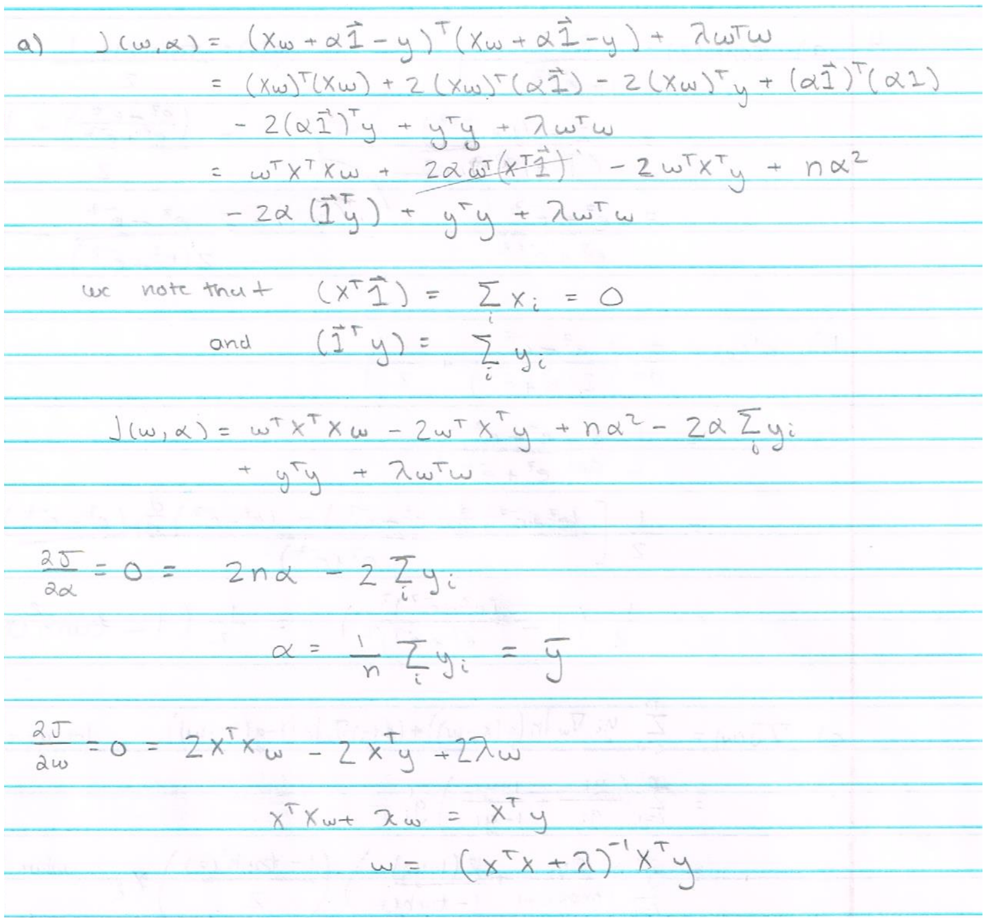
CS 189: Homework 4

Michael Stephen Chen
Kaggle Acct: michaelstchen
SID: 23567341

April 2, 2016

Problem 1

a) Below is my write up for this part



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

$= (Xw)^T(Xw) + 2(Xw)^T(\alpha\vec{1}) - 2(Xw)^T y + (\alpha\vec{1})^T(\alpha 1)$

$- 2(\alpha\vec{1})^T y + y^T y + \lambda w^T w$

$= w^T X^T X w + \cancel{2\alpha w^T (X^T\vec{1})} - 2 w^T X^T y + n\alpha^2$

$- 2\alpha(\vec{1}^T y) + y^T y + \lambda w^T w$

we note that $(X^T\vec{1}) = \sum_i x_i = 0$

and $(\vec{1}^T y) = \sum_i y_i$

$J(w,\alpha) = w^T X^T X w - 2 w^T X^T y + n\alpha^2 - 2\alpha \sum_i y_i$

$+ y^T y + \lambda w^T w$

$\frac{\partial J}{\partial \alpha} = 0 = 2n\alpha - 2\sum_i y_i$

$\alpha = \frac{1}{n}\sum_i y_i = \bar{y}$

$\frac{\partial J}{\partial w} = 0 = 2X^T X w - 2X^T y + 2\lambda w$

$X^T X w + \lambda w = X^T y$

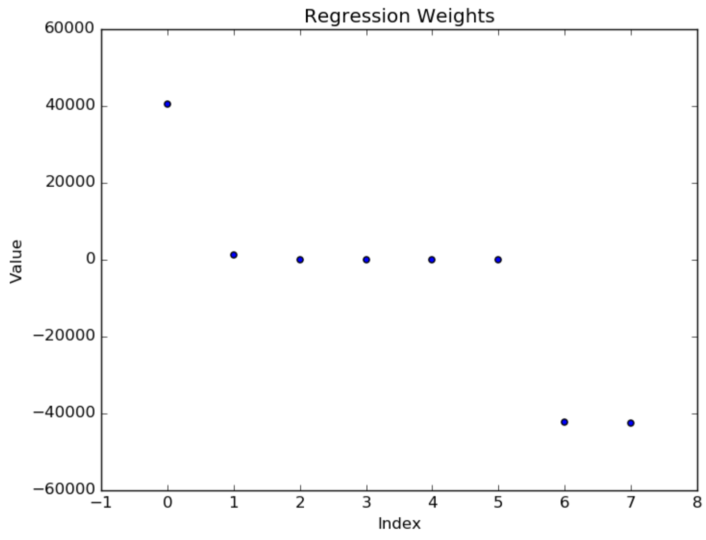
$w = (X^T X + \lambda)^{-1} X^T y$

b) i. See *prob1.py* in the Appendix

ii. After 10-fold cross-validation, I found the optimal $\lambda = 0$. The RSS for the validation set was **5782240895023.2158** (or 5.7822e12), which is on the same order of magnitude as the RSS for hw3 which was **5794953797667.3555**.

This is expected because when $\lambda = 0$, the ridge regresssion essentially becomes a normal linear regression because there is no penalty against large weights. The slight discrepancy between the two RSS values comes from the fact that instead of appending a column of ones to the data and regressing over that, we just add an average to account for translation from the origin.

iii. Below is a plot of the regression coefficients for our ridge regression:



Again the regression coefficients are essentially the same as the ones from our linear regression in hw3. As aforementioned, this is because we found the optimal lambda to be $\lambda = 0$, which essentially reduces our ridge regression into a regular linear regression.

Problem 2

For the following problems, please see *prob2.py* for the code I used to obtain my values

a) $R(\boldsymbol{w}^{(0)}) = 1.9883724141284103$

b) $\boldsymbol{\mu}^{(0)} = [0.95257413, 0.73105858, 0.73105858, 0.26894142]^T$

c) $\boldsymbol{w}^{(1)} = [-2, 0.94910188, -0.68363271]^T$

d) $R(\boldsymbol{w}^{(1)}) = 1.7206170956213047$

e) $\boldsymbol{\mu}^{(1)} = [0.89693957, 0.54082713, 0.56598026, 0.15000896]^T$

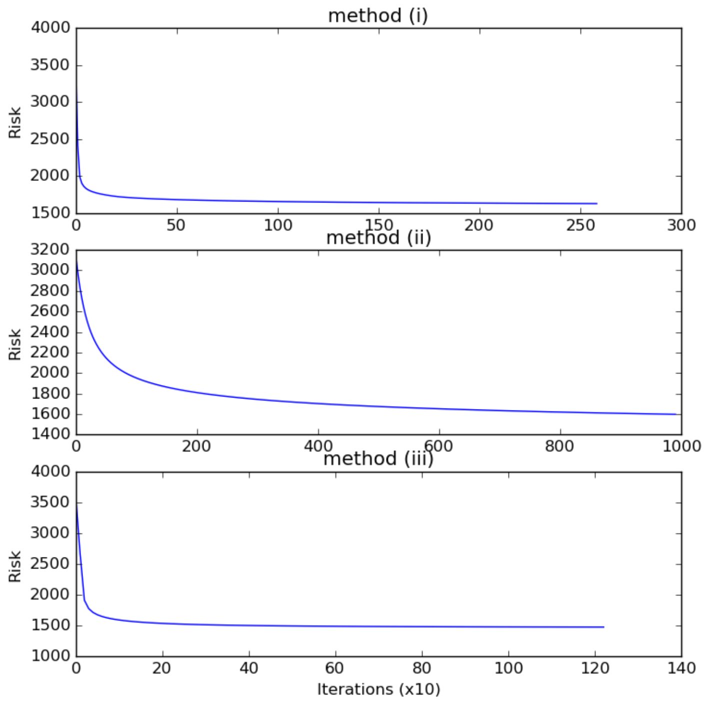
f) $\boldsymbol{w}^{(2)} = [-1.69083609, 1.91981257, -0.83738862]^T$

g) $R(\boldsymbol{w}^{(2)}) = 1.8546997847922495$

Problem 3

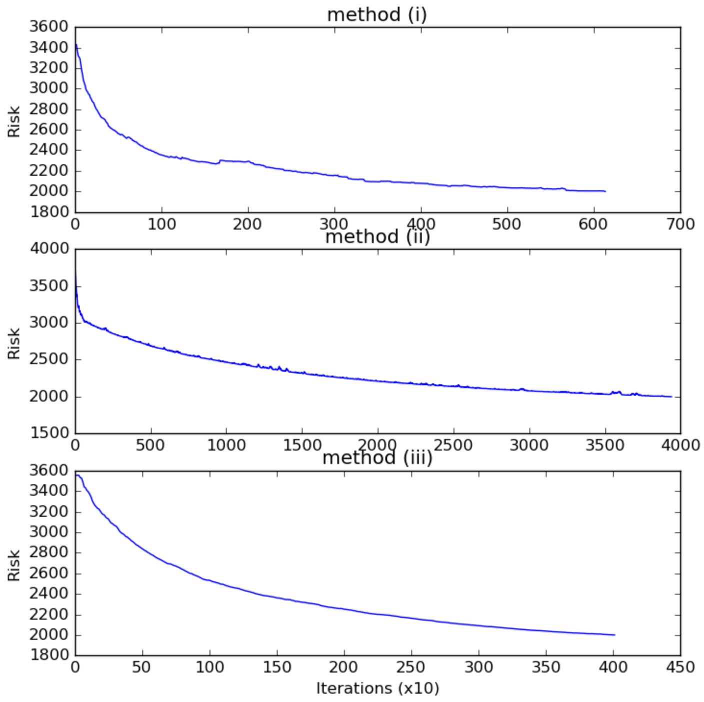
For the scripts, please check the Appendix. All scripts are named after the part they were used for.

1. Batch gradient descent with a stopping criteria of 0.1 resulted in risks of 1743.60, 1954.24, and 1592.64 for methods (i), (ii), and (iii), respectively. The step sizes were 1e-3, 1e-6, and 1e-3 for methods (i), (ii), and (iii). The initial weights were all initialized at 0.01 for all three methods. Below are the plots depicting the risks as a function of update iterations:



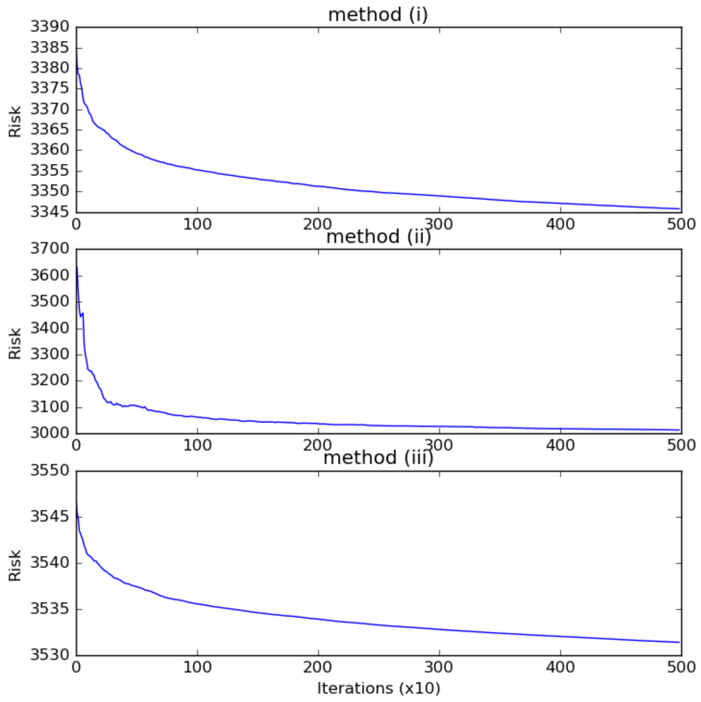
From the plots above, we see that both methods (i) and (iii) converge fairly quickly at around 2500 and 1200 iterations, respectively. (iii) converges slightly faster than (i), and at a slightly lower risk value as well. At first I found it odd that the binary preprocessing method gave a lower risk value than the scaling method. I would have thought that the binary method would underfit the data given that it clamps values, and would consequently result in a higher training risk. Then I came to the realization that although this might be true, the scaling method was probably "underfitting", maybe its more appropriate to use "generalizing", to a larger degree than the binary method.

2. Stochastic gradient descent results are presented below. Unlike the previous part, this time our stopping criteria was when the risk < 2000 and our initial weight vector consisted of all 0's except for the first element which was initialized to 1. The step sizes were 1e-2, 1e-4, and 1e-2 for methods (i), (ii), and (iii).



Like in part (1), we see that methods (i) and (iii) reach the stopping criteria relatively at around 6000 and 4000 iterations, respectively. It takes more iterations for stochastic gradient descent to converge, which makes sense because only one sample is effectively examined at a time whereas in batch gradient descent all of the samples are considered each iteration. This is also why we observe more fluctuatuions than with the batch process (risk occaisonally increases).

3. We used an initial learning rate of 0.01 for each of the three preprocessing methods.



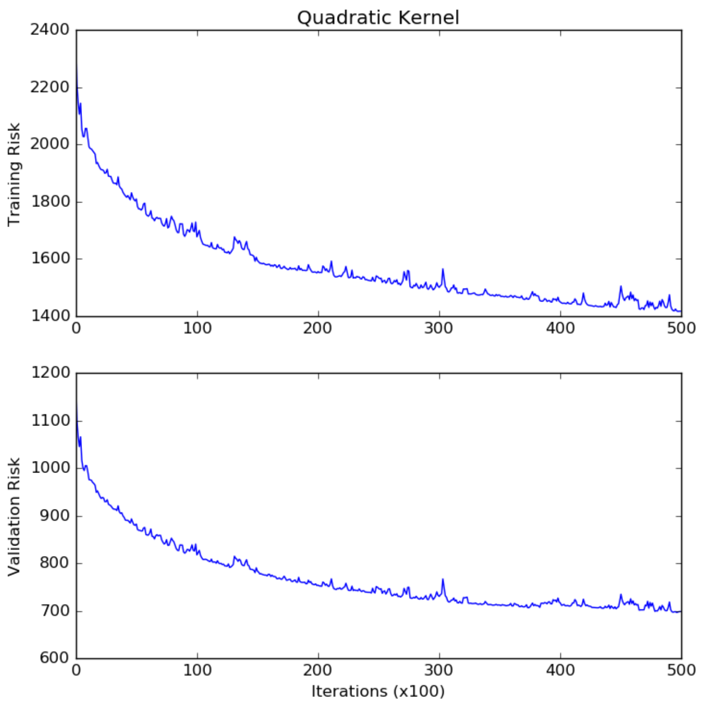
Initially when the learning rate is still relatively large we see fluctuations in the risk just like we did in part (2). However as the number of iterations increases, the risk plot becomes smoother than the correponding plot for part (2) because the learning rate decreases inversely proportional to the number of iterations (so updates are smaller in magnitude). Because of the decreasing learning rate, the risks at 5000 iterations using this method are considerably higher than those for the constant learning rate.

4. For the kernel implentations I decided to use preprocessing method (i) as opposed to (iii). Even though (iii) slightly outperforms (i) with respect to the training risk in part (2), (i) produced lower validation risks when I compared the two methods for this part. Maybe its because I was unable to tune the parameters to best suit (iii) and/or (iii) doesn't generalize as well as (i).

   a) For the quadratic kernel, I used $\lambda = 0.001$ and $\epsilon = 1e-5$. The initial weights were all set to 0. Tuning the ρ parameter via 6-fold cross validation, we found that the optimal value was $rho = 1$:

ρ	$risk$
0.0001	337.83
0.01	338.38
0.1	326.34
1	318.12
10	318.31
100	348.03
1000	681.77

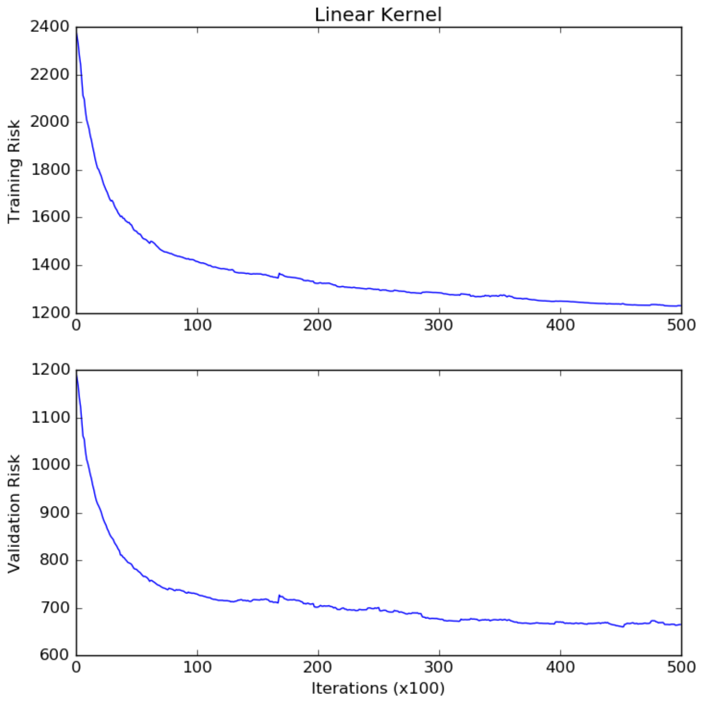
   The two plots below plot the training (top) and validation risks (bottom) over the iterations. The training risk settled at 1415.67 and the validation risk settled at 698.56.



b) For the linear kernel, I used $\lambda = 0.001$ and $\epsilon = 0.01$. The initial weights were all set to 0. Tuning the ρ parameter via 6-fold cross validation, we found that the optimal value was $rho = 100$:

ρ	$risk$
0.0001	524.80
0.01	523.60
0.1	524.20
1	514.88
10	407.23
100	335.81
1000	715.65

The two plots below plot the training (top) and validation risks (bottom) over the iterations. The training risk settled at 1229.46 and the validation risk settled at 664.97.

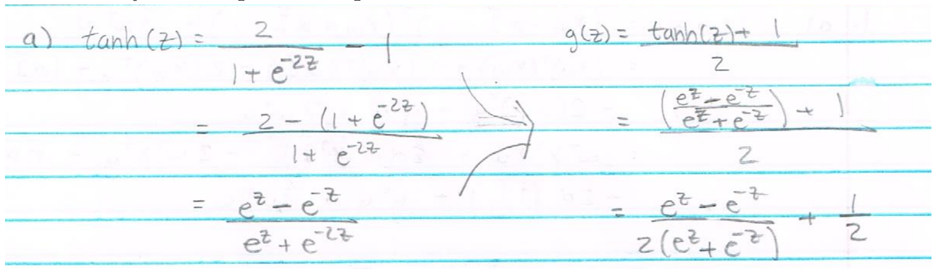


We can usually tell if we are overfitting by examining both the training and validation results. The training risk should always trend downwards asymptotically. If the validation risk trends down but then starts trending upwards again, then that is a sign that we are overfitting the training data and thus not generalizing well to the validation data. Even at 50000 iterations we see no obvious sings of overfitting for either the quadratic or linear kernel logistic regressions. Perhaps this is because our λ is sufficiently large enough to prevent overfitting by penalizing large weights in $\vec{a}$. We would have expected that if either of the methods would overfit, it would be the quadratic kernel because of its higher dimensionality. If that were the case we would want to increase λ to prevent $\vec{a}$ from getting too large and overfitting. Conversely we would expect that the linear kernel would be more susceptible to underfitting so we would want to decrease λ.

5. I used the linear kernel logistic regression, where the data has been standardized with mean 0 and variance 1, for my Kaggle predictions. I used a custom featurizer, which is just the one given for previous homeworks with additional words added (e.g. viagra, re :, etc.). My Kaggle score was **0.85065**

Problem 4

a) Below is my write up for this part



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

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

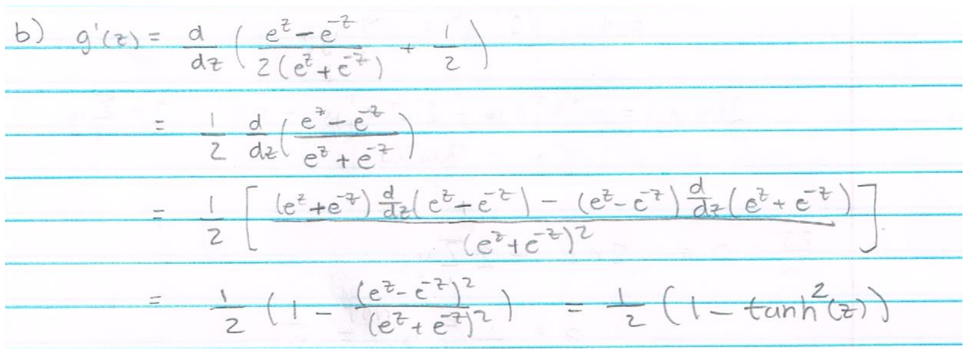
$= \frac{e^{z}-e^{-z}}{e^{z}+e^{-2z}}$

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

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

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

b) Below is my write up for this part



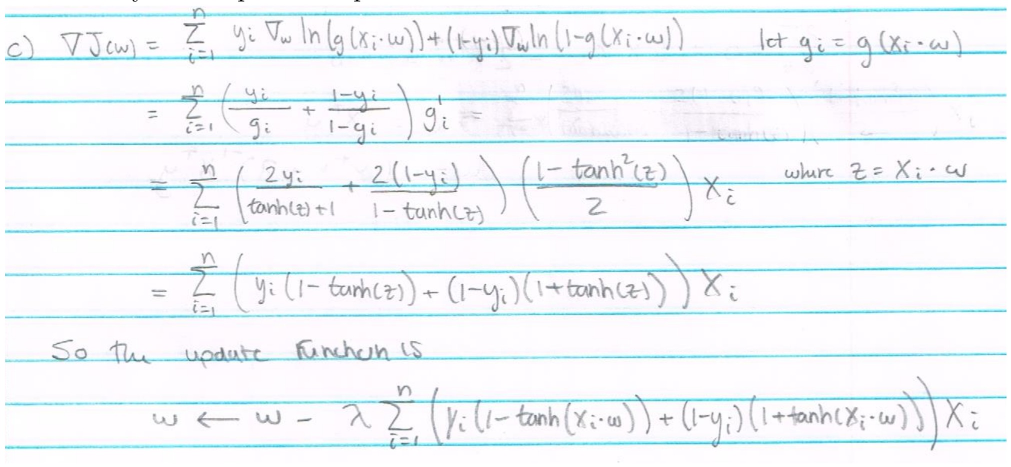
b) $g'(z) = \frac{d}{dz}\left(\frac{e^z - e^{-z}}{2(e^z+e^{-z})} + \frac{1}{2}\right)$

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

$= \frac{1}{2}\left[\frac{(e^z+e^{-z})\frac{d}{dz}(e^z-e^{-z}) - (e^z-e^{-z})\frac{d}{dz}(e^z+e^{-z})}{(e^z+e^{-z})^2}\right]$

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

c) Below is my write up for this part



c) $\nabla J(w) = \sum_{i=1}^{n} y_i \nabla_w \ln(g(x_i\cdot w)) + (1-y_i)\nabla_w \ln(1-g(x_i\cdot w))$ let $g_i = g(x_i \cdot w)$

$= \sum_{i=1}^{n}\left(\frac{y_i}{g_i} + \frac{1-y_i}{1-g_i}\right) g_i' =$

$= \sum_{i=1}^{n}\left(\frac{2y_i}{\tanh(z)+1} + \frac{2(1-y_i)}{1-\tanh(z)}\right)\left(\frac{1-\tanh^2(z)}{2}\right) X_i$ where $z = X_i \cdot w$

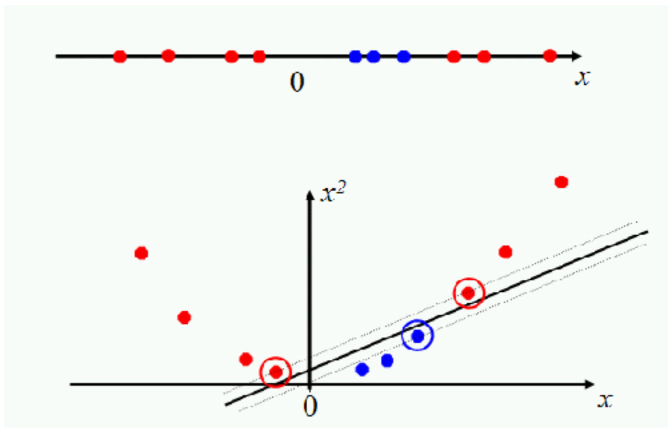
$= \sum_{i=1}^{n}\Big(y_i(1-\tanh(z)) + (1-y_i)(1+\tanh(z))\Big) X_i$

So the update function is

$w \leftarrow w - \lambda \sum_{i=1}^{n}\Big(y_i(1-\tanh(x_i\cdot w)) + (1-y_i)(1+\tanh(x_i\cdot w))\Big) X_i$

Problem 5

Daniel timestamps all messages based on the time since the previous midnight. He also notes that the number of spam messages tends to increase around midnight. If this is indeed the case, then the spam data points will be concentrated about the two ends of the timestamp feature space; the beginning and ending of the feature space represent the the times just after and before midnight, respectively. Of course such a data set is not linearly separable and so linear SVM cannot adequately classify based on this feature. If instead were were to lift the feature into a higher space, by using a quadratic kernel then we would be able to linearly separate the data based on this feature as seen below (red represents spam and blue represents ham):



Source: http://nlp.stanford.edu/IR-book/html/htmledition/nonlinear-svms-1.html

So to improve results when adding this feature and using a linear SVM, Daniel should use a quadratic kernel.

Appendix: prob1.py

```
import numpy as np
import math
import csv
import matplotlib.pyplot as plt
from scipy.io import loadmat

def benchmark(pred_labels, true_labels):
    errors = pred_labels != true_labels
    err_rate = sum(errors) / float(len(true_labels))
    indices = errors.nonzero()
    return err_rate, indices

housing_data = loadmat(file_name="housing_data.mat", mat_dtype=True)

Xtrain = housing_data["Xtrain"]
Ytrain = housing_data["Ytrain"]
Xvalid = housing_data["Xvalidate"]
Yvalid = housing_data["Yvalidate"]

Ytrain = Ytrain[:, 0]
Yvalid = Yvalid[:, 0]

# Centering data
Xtrain = Xtrain - np.mean(Xtrain, axis=0)
Xvalid = Xvalid - np.mean(Xvalid, axis=0)

num_train = Xtrain.shape[0]
num_valid = Xvalid.shape[0]

param_list = [0.0, 1e-10, 1e-5, 1e-3, 1e-2, 1, 1e2, 1e3, 1e5, 1e10, 1e15, 1e20]
#J_list = [0] * len(param_list)
R_list = [0] * len(param_list)
for i in range(0, len(param_list)):
    for k in range(0, num_train, num_train/10):
        param = param_list[i]
        X = np.vstack((Xtrain[:k], Xtrain[k+num_train/10:]))
        Y = np.append(Ytrain[:k], Ytrain[k+num_train/10:])
        Xval = Xtrain[k:k+num_train/10]
        Yval = Ytrain[k:k+num_train/10]

        Xt_X = np.dot(X.T, X)
        lambda_I = param * np.identity(len(Xt_X))
        Xt_y = np.dot(X.T, Y)
        w = np.dot(np.linalg.inv(Xt_X + lambda_I), Xt_y)

        alpha = np.mean(Y)
        Ypred = np.dot(Xval, w) + alpha*np.ones(len(Yval))
        res = Ypred - Yval
```

```
            #J_list[i] = J_list[i] + np.dot(res, res)+param*np.dot(w, w)
            R_list[i] = R_list[i] + np.dot(res, res)

        #J_list[i] = J_list[i] / 10
        R_list[i] = R_list[i] / 10

param = param_list[R_list.index(min(R_list))]
Xt = np.transpose(Xtrain)
Xt_X = np.dot(Xt, Xtrain)
lambda_I = param * np.identity(len(Xt_X))
Xt_y = np.dot(Xt, Ytrain)
w = np.dot(np.linalg.inv(Xt_X + lambda_I), Xt_y)

alpha = np.mean(Ytrain)
Ypred = np.dot(Xvalid, w) + alpha*np.ones(len(Yvalid))
res_valid = Yvalid - Ypred
RSS = np.dot(res_valid, res_valid)

plt.scatter(range(0, len(w)), w)
plt.title('Regression Weights')
plt.xlabel('Index')
plt.ylabel('Value')
plt.show()
```

Appendix: prob2.py

```
import numpy as np
from math import log, exp

X = np.array([[0, 3, 1],
              [1, 3, 1],
              [0, 1, 1],
              [1, 1, 1]], dtype=np.float)

y = np.array([1, 1, 0, 0], dtype=np.float)
w0 = np.array([-2, 1, 0], dtype=np.float)
eps = 1

def sig(x):
    return 1 / (1 + np.exp(-x))

def mu(X, w):
    return sig(np.dot(X, w))

def risk(X, y, w):
    emp_risk = 0.0
    for i in range(0, len(X)):
        a = y[i]*log(sig(np.dot(w, X[i])))
        b = (1 - y[i])*log(1 - sig(np.dot(w, X[i])))
        emp_risk -= a + b
    return emp_risk

def grad_asc_update(X, y, w):
    upd = np.zeros(3)
    for i in range(0, np.size(X, 0)):
        upd += (y[i] - mu(X[i, :], w)) * X[i, :]
    return w + (eps * upd)


R0 = risk(X, y, w0)
mu0 = mu(X, w0)

w1 = grad_asc_update(X, y, w0)
R1 = risk(X, y, w1)
mu1 = mu(X, w1)

w2 = grad_asc_update(X, y, w1)
R2 = risk(X, y, w2)
```

Appendix: prob3 1.py

```
import numpy as np
import math
import csv
import matplotlib.pyplot as plt
from scipy.io import loadmat
from sklearn.preprocessing import scale, binarize

housing_data = loadmat(file_name="spam_data.mat", mat_dtype=True)
train_data = np.array(housing_data['training_data'])
train_labels = np.transpose(np.array(housing_data['training_labels']))

train_data_i = scale(train_data)
train_data_ii = np.log(train_data + 0.1)
train_data_iii = binarize(train_data)

def sig(x):
    return 1 / (1 + np.exp(-x))

def risk(X, y, w):
    sig_Xw = sig(np.dot(X, w))
    a = np.multiply(y, np.log(sig_Xw + 1e-100))
    b = np.multiply(1-y, np.log(1-sig_Xw + 1e-100))
    emp_risk = np.sum(a + b)
    return -emp_risk

def grad_desc_batch(X, y, w, eps):
    upd = y - sig(np.dot(X, w))
    upd = (X.T * upd).T
    upd = np.sum(upd, axis=0)
    return w + (eps * upd)

def log_regr_batch(X, y, w0, learn, err):
    risk_list = []
    risk_list.append(risk(X, y, w0))
    w = grad_desc_batch(X, y, w0, learn)
    risk_list.append(risk(X, y, w))
    i = 2
    while (abs(risk_list[-1]-risk_list[-2]) > err):
        w = grad_desc_batch(X, y, w, learn)
        if (i % 10 == 0):
            risk_list.append(risk(X, y, w))
        i = i + 1
    return risk_list


w0 = np.ones(len(train_data[0])) / 100

# For preprocessing (i)
risk_i = log_regr_batch(train_data_i,train_labels[:,0],w0,1e-3,0.1)

# For preprocessing (ii)
risk_ii = log_regr_batch(train_data_ii,train_labels[:,0],w0,1e-6,0.1)

# For preprocessing (iii)
risk_iii = log_regr_batch(train_data_iii,train_labels[:,0],w0,1e-3,0.1)

plt.subplot(3, 1, 1)
plt.plot(range(0,len(risk_i)), risk_i)
plt.title('method (i)')
plt.ylabel('Risk')

plt.subplot(3, 1, 2)
plt.plot(range(0,len(risk_ii)), risk_ii)
plt.title('method (ii)')
plt.ylabel('Risk')

plt.subplot(3, 1, 3)
plt.plot(range(0,len(risk_iii)), risk_iii)
plt.title('method (iii)')
plt.ylabel('Risk')
plt.xlabel('Iterations (x10)')

plt.show()
```

Appendix: prob3 2.py

```
import numpy as np
import math
import csv
import matplotlib.pyplot as plt
from scipy.io import loadmat
from sklearn.preprocessing import scale, binarize

spam_data = loadmat(file_name="spam_data.mat", mat_dtype=True)
train_data = np.array(spam_data['training_data'])
train_labels = np.transpose(np.array(spam_data['training_labels']))

train_data_i = scale(train_data)
train_data_ii = np.log(train_data + 0.1)
train_data_iii = binarize(train_data)
```

```
def sig(x):
    return 1 / (1 + np.exp(-x))

def risk(X, y, w):
    sig_Xw = sig(np.dot(X, w))
    a = np.multiply(y, np.log(sig_Xw + 1e-100))
    b = np.multiply(1-y, np.log(1-sig_Xw + 1e-100))
    emp_risk = np.sum(a + b)
    return -emp_risk

def grad_desc_stoch(X, y, w, eps):
    upd = (y - sig(np.dot(X, w))) * X
    return w + (eps * upd)

def log_regr_stoch(X, y, w0, learn):
    risk_list = []
    risk_list.append(risk(X, y, w0))
    w = grad_desc_stoch(X[0], y[0], w0, learn)
    risk_list.append(risk(X, y, w))
    i = 0
    while (risk_list[-1] > 2000):
        j = np.random.randint(0, len(X)-1)
        w = grad_desc_stoch(X[j], y[j], w, learn)
        if (i % 10 == 0):
            risk_list.append(risk(X, y, w))
        i = i + 1
    return risk_list


w0 = np.zeros(len(train_data[0]))
w0[0] = 1.0
risk_i = log_regr_stoch(train_data_i, train_labels[:,0], w0, 1e-2)
risk_ii = log_regr_stoch(train_data_ii, train_labels[:,0], w0, 1e-4)
risk_iii = log_regr_stoch(train_data_iii, train_labels[:,0], w0, 1e-2)

plt.subplot(3, 1, 1)
plt.plot(range(0,len(risk_i)), risk_i)
plt.title('method_(i)')
plt.ylabel('Risk')

plt.subplot(3, 1, 2)
plt.plot(range(0,len(risk_ii)), risk_ii)
plt.title('method_(ii)')
plt.ylabel('Risk')

plt.subplot(3, 1, 3)
plt.plot(range(0,len(risk_iii)), risk_iii)
plt.title('method_(iii)')
plt.xlabel('Iterations_(x10)')
plt.ylabel('Risk')

plt.show()
```

Appendix: prob3 3.py

```
import numpy as np
import math
import csv
import matplotlib.pyplot as plt
from scipy.io import loadmat
from sklearn.preprocessing import scale, binarize

spam_data = loadmat(file_name="spam_data.mat", mat_dtype=True)
train_data = np.array(spam_data['training_data'])
train_labels = np.transpose(np.array(spam_data['training_labels']))

train_data_i = scale(train_data)
train_data_ii = np.log(train_data + 0.1)
train_data_iii = binarize(train_data)

def sig(x):
    return 1 / (1 + np.exp(-x))

def risk(X, y, w):
    sig_Xw = sig(np.dot(X, w))
    a = np.multiply(y, np.log(sig_Xw + 1e-100))
    b = np.multiply(1-y, np.log(1-sig_Xw + 1e-100))
    emp_risk = np.sum(a + b)
    return -emp_risk

def grad_desc_stoch(X, y, w, eps):
    upd = (y - sig(np.dot(X, w))) * X
    return w + (eps * upd)

def log_regr_stoch(X, y, w, learn, lim):
    risk_list = []
    for t in range(1, lim):
        j = np.random.randint(0, len(X)-1)
        w = grad_desc_stoch(X[j], y[j], w, learn/t)
        if (t % 10 == 0):
            risk_list.append(risk(X, y, w))
    return risk_list

lim = 5000
w0 = np.zeros(len(train_data[0]))
w0[0] = 1.0
risk_i = log_regr_stoch(train_data_i,train_labels[:,0],w0,0.01,lim)
risk_ii = log_regr_stoch(train_data_ii,train_labels[:,0],w0,0.01,lim)
risk_iii = log_regr_stoch(train_data_iii,train_labels[:,0],w0,0.01,lim)

plt.subplot(3, 1, 1)
plt.plot(range(0,len(risk_i)), risk_i)
plt.title('method_(i)')
plt.ylabel('Risk')

plt.subplot(3, 1, 2)
plt.plot(range(0,len(risk_ii)), risk_ii)
plt.title('method (ii)')
plt.ylabel('Risk')

plt.subplot(3, 1, 3)
plt.plot(range(0,len(risk_iii)), risk_iii)
plt.title('method (iii)')
plt.xlabel('Iterations (x10)')
plt.ylabel('Risk')

plt.show()
```

Appendix: prob3 4.py

```
import numpy as np
import math
import csv
import matplotlib.pyplot as plt
from scipy.io import loadmat
from sklearn.preprocessing import scale, binarize

def kernel_mat(X1, X2, deg, rho):
    K = np.dot(X1, X2.T) + rho
    return K**deg

def sig(x):
    return 1 / (1 + np.exp(-x))

def risk(K, a, y):
    sig_Ka = sig(np.dot(K, a))
    a = np.multiply(y, np.log(sig_Ka + 1e-100))
    b = np.multiply(1-y, np.log(1 - sig_Ka + 1e-100))
    return -np.sum(a + b)

def grad_desc_stoch(i, y_i, Ka_i, a, eps, lam):
    a_upd = a - (eps * lam) * a
    a_upd[i] = a_upd[i] + eps * (y_i - sig(Ka_i))
    return a_upd

def log_regr_kernel(X, y, K, a, eps, lam, lim):
    risks = []
    for i in range(0, lim):
        j = np.random.randint(0, len(X))
        Ka_j = np.dot(K[j], a)
        a = grad_desc_stoch(j, y[j], Ka_j, a, eps, lam)
        if (i % 100 == 0):
            risks.append(risk(K, a, y))
    return a, risks

def log_regr_kernel2(Xt, yt, Xv, yv, Kt, Kv, a, eps, lam, lim):
    trisks = []
    vrisks = []
    for i in range(0, lim):
        j = np.random.randint(0, len(Xt))
        Kta_j = np.dot(Kt[j], a)
        a = grad_desc_stoch(j, yt[j], Kta_j, a, eps, lam)
        if (i % 100 == 0):
            trisks.append(risk(Kt, a, yt))
            vrisks.append(risk(Kv, a, yv))
    return a, trisks, vrisks

''' LOADING AND SHUFFLING DATA '''
spam_data = loadmat(file_name="spam_data48.mat", mat_dtype=True)
train_data = np.array(spam_data['training_data'])
train_labels = np.transpose(np.array(spam_data['training_labels']))
train_labels = train_labels[:, 0]

random_state = np.random.get_state()
np.random.shuffle(train_data)
np.random.set_state(random_state)
np.random.shuffle(train_labels)

train_data = scale(train_data)
num_train = len(train_data)


''' 6-FOLD CROSS VALIDATION '''
lim = 10000
lam = 0.001
eps = 0.01
# eps = 1e-5
rho_list = [1000, 100, 10, 1, 0.1, 0.01, 0.001]
rho_risks = [0] * len(rho_list)
for i in range(0, len(rho_list)):
    print('Iteration: ' + str(i))
    rho = rho_list[i]
    for k in range(0, num_train, num_train/6):
        td = np.vstack((train_data[:k], train_data[k+num_train/6:]))
        tl = np.append(train_labels[:k],train_labels[k+num_train/6:])
        vd = train_data[k:k+num_train/6]
        vl = train_labels[k:k+num_train/6]

        K = kernel_mat(td, td, 1, rho)
        #K = kernel_mat(td, td, 2, rho)
        a = np.zeros(len(td))
        a, risks = log_regr_kernel(td, tl, K, a, eps, lam, lim)

        K_val = kernel_mat(vd, td, 1, rho)
        #K_val = kernel_mat(vd, td, 2, rho)
        rho_risks[i] = rho_risks[i] + risk(K_val, a, vl)

    rho_risks[i] = rho_risks[i] / 6


rho = rho_list[rho_risks.index(min(rho_risks))]


''' PLOTTING TRAINING AND VALIDATION RISKS '''
k = len(train_data) * 2 / 3
```

```
valid_data = train_data[k:]
valid_labels = train_labels[k:]
train_data = train_data[0:k]
train_labels = train_labels[0:k]

lim = 20000
lam = 0.001
a0 = np.zeros(len(train_data))

#eps = 0.01
#rho = 1
#K_train = kernel_mat(train_data, train_data, 1, rho)
#K_valid = kernel_mat(valid_data, train_data, 1, rho)

eps = 1e-5
rho = 100
K_train = kernel_mat(train_data, train_data, 2, rho)
K_valid = kernel_mat(valid_data, train_data, 2, rho)

a, trisks, vrisks = log_regr_kernel2(train_data, train_labels, valid_data, valid_labels, K


plt.subplot(2, 1, 1)
plt.plot(trisks)
#plt.title('Linear Kernel')
plt.title('Quadratic Kernel')
plt.ylabel('Training Risk')

plt.subplot(2, 1, 2)
plt.plot(vrisks)
plt.xlabel('Iterations (x100)')
plt.ylabel('Validation Risk')

plt.show()
```

Appendix: prob3 5.py

```
import numpy as np
import math
import csv
import matplotlib.pyplot as plt
from scipy.io import loadmat
from sklearn.preprocessing import scale, binarize

def kernel_mat(X1, X2, deg, rho):
    K = np.dot(X1, X2.T) + rho
    return K**deg

def sig(x):
    return 1 / (1 + np.exp(-x))

def risk(K, a, y):
    sig_Ka = sig(np.dot(K, a))
    a = np.multiply(y, np.log(sig_Ka + 1e-100))
    b = np.multiply(1-y, np.log(1 - sig_Ka + 1e-100))
    return -np.sum(a + b)

def grad_desc_stoch(i, y_i, Ka_i, a, eps, lam):
    a_upd = a - (eps * lam) * a
    a_upd[i] = a_upd[i] + eps * (y_i - sig(Ka_i))
    return a_upd

def log_regr_kernel(X, y, K, a, eps, lam, lim):
    risks = []
    for i in range(0, lim):
        j = np.random.randint(0, len(X))
        Ka_j = np.dot(K[j], a)
        a = grad_desc_stoch(j, y[j], Ka_j, a, eps, lam)
        if (i % 100 == 0):
            risks.append(risk(K, a, y))
    return a, risks


''' LOADING DATA '''
spam_data = loadmat(file_name="spam_data.mat", mat_dtype=True)
train_data = np.array(spam_data['training_data'])
train_labels = np.transpose(np.array(spam_data['training_labels']))
train_labels = train_labels[:, 0]

train_data = scale(train_data)
train_data = train_data[:3000]
train_labels = train_labels[:3000]
num_train = len(train_data)

''' LINEAR KERNEL LOGISTIC REGRESSION'''
lim = 15000
lam = 0.001
a0 = np.zeros(len(train_data))

eps = 1e-5
rho = 100
K_train = kernel_mat(train_data, train_data, 2, rho)

a, risks = log_regr_kernel(train_data, train_labels, K_train, a0, eps, lam, lim)

''' PREDICTING '''
test_data = np.array(spam_data['test_data'])
test_data = scale(test_data)
K_test = kernel_mat(test_data, train_data, 1, rho)
pred_labels = np.rint(sig(np.dot(K_test, a)))

with open('test_labels.csv', 'wb') as f:
    writer = csv.writer(f)
    writer.writerow(['Id'] + ['Category'])
    for i in range(0, len(pred_labels)):
        writer.writerow([i+1] + [int(pred_labels[i])])
```

Appendix: featurize.py

```
'''
**************** PLEASE READ ***************

Script that reads in spam and ham messages and converts each training example
into a feature vector

Code intended for UC Berkeley course CS 189/289A: Machine Learning

Requirements:
-scipy ('pip install scipy')

To add your own features, create a function that takes in the raw text and
word frequency dictionary and outputs a int or float. Then add your feature
in the function 'def generate_feature_vector'

The output of your file will be a .mat file. The data will be accessible using
the following keys:
    -'training_data'
    -'training_labels'
    -'test_data'

Please direct any bugs to kevintee@berkeley.edu
'''

from collections import defaultdict
import glob
import re
import scipy.io

NUM_TRAINING_EXAMPLES = 5172
NUM_TEST_EXAMPLES = 5857

BASE_DIR = './'
SPAM_DIR = 'spam/'
HAM_DIR = 'ham/'
TEST_DIR = 'test/'

# ************* Features *************

# Features that look for certain words
def freq_pain_feature(text, freq):
    return float(freq['pain'])

def freq_private_feature(text, freq):
    return float(freq['private'])

def freq_bank_feature(text, freq):
    return float(freq['bank'])

def freq_money_feature(text, freq):
    return float(freq['money'])
```

```
def freq_drug_feature(text, freq):
    return float(freq['drug'])

def freq_spam_feature(text, freq):
    return float(freq['spam'])

def freq_prescription_feature(text, freq):
    return float(freq['prescription'])

def freq_creative_feature(text, freq):
    return float(freq['creative'])

def freq_height_feature(text, freq):
    return float(freq['height'])

def freq_featured_feature(text, freq):
    return float(freq['featured'])

def freq_differ_feature(text, freq):
    return float(freq['differ'])

def freq_width_feature(text, freq):
    return float(freq['width'])

def freq_other_feature(text, freq):
    return float(freq['other'])

def freq_energy_feature(text, freq):
    return float(freq['energy'])

def freq_business_feature(text, freq):
    return float(freq['business'])

def freq_message_feature(text, freq):
    return float(freq['message'])

def freq_volumes_feature(text, freq):
    return float(freq['volumes'])

def freq_revision_feature(text, freq):
    return float(freq['revision'])

def freq_path_feature(text, freq):
    return float(freq['path'])

def freq_meter_feature(text, freq):
    return float(freq['meter'])

def freq_memo_feature(text, freq):
    return float(freq['memo'])

def freq_planning_feature(text, freq):
    return float(freq['planning'])
```

```
def freq_pleased_feature(text, freq):
    return float(freq['pleased'])

def freq_record_feature(text, freq):
    return float(freq['record'])

def freq_out_feature(text, freq):
    return float(freq['out'])

# Features that look for certain characters
def freq_semicolon_feature(text, freq):
    return text.count(';')

def freq_dollar_feature(text, freq):
    return text.count('$')

def freq_sharp_feature(text, freq):
    return text.count('#')

def freq_exclamation_feature(text, freq):
    return text.count('!')

def freq_para_feature(text, freq):
    return text.count('(')

def freq_bracket_feature(text, freq):
    return text.count('[')

def freq_and_feature(text, freq):
    return text.count('&')

# -------- Add your own feature methods ----------

def freq_buy_feature(text, freq):
    return text.count('buy')

def freq_cash_feature(text, freq):
    return text.count('cash')

def freq_freetrial_feature(text, freq):
    return text.count('free trial')

def freq_microsoft_feature(text, freq):
    return text.count('microsoft')

def freq_medication_feature(text, freq):
    return text.count('medication')

def freq_viagra_feature(text, freq):
    return text.count('viagra')

def freq_cialis_feature(text, freq):
    return text.count('cialis')
```

```
def freq_sex_feature(text, freq):
    return text.count('sex')

def freq_sexual_feature(text, freq):
    return text.count('sexual')

def freq_penis_feature(text, freq):
    return text.count('penis')

def freq_havebeenselected_feature(text, freq):
    return text.count('have been selected')

def freq_percent_feature(text, freq):
    return text.count('%')

def freq_www_feature(text, freq):
    return text.count('www')

def freq_100_feature(text, freq):
    return text.count('100')

def freq_re_feature(text, freq):
    return text.count('re :')

def freq_cc_feature(text, freq):
    return text.count('cc :')

# Generates a feature vector
def generate_feature_vector(text, freq):
    feature = []
    feature.append(freq_pain_feature(text, freq))
    feature.append(freq_private_feature(text, freq))
    feature.append(freq_bank_feature(text, freq))
    feature.append(freq_money_feature(text, freq))
    feature.append(freq_drug_feature(text, freq))
    feature.append(freq_spam_feature(text, freq))
    feature.append(freq_prescription_feature(text, freq))
    feature.append(freq_creative_feature(text, freq))
    feature.append(freq_height_feature(text, freq))
    feature.append(freq_featured_feature(text, freq))
    feature.append(freq_differ_feature(text, freq))
    feature.append(freq_width_feature(text, freq))
    feature.append(freq_other_feature(text, freq))
    feature.append(freq_energy_feature(text, freq))
    feature.append(freq_business_feature(text, freq))
    feature.append(freq_message_feature(text, freq))
    feature.append(freq_volumes_feature(text, freq))
    feature.append(freq_revision_feature(text, freq))
    feature.append(freq_path_feature(text, freq))
    feature.append(freq_meter_feature(text, freq))
    feature.append(freq_memo_feature(text, freq))
    feature.append(freq_planning_feature(text, freq))
    feature.append(freq_pleased_feature(text, freq))
    feature.append(freq_record_feature(text, freq))
    feature.append(freq_out_feature(text, freq))
    feature.append(freq_semicolon_feature(text, freq))
    feature.append(freq_dollar_feature(text, freq))
    feature.append(freq_sharp_feature(text, freq))
    feature.append(freq_exclamation_feature(text, freq))
    feature.append(freq_para_feature(text, freq))
    feature.append(freq_bracket_feature(text, freq))
    feature.append(freq_and_feature(text, freq))

    # --------- Add your own features here ---------
    # Make sure type is int or float

    feature.append(freq_buy_feature(text, freq))
    feature.append(freq_cash_feature(text, freq))
    feature.append(freq_freetrial_feature(text, freq))
    feature.append(freq_microsoft_feature(text, freq))
    feature.append(freq_medication_feature(text, freq))
    feature.append(freq_viagra_feature(text, freq))
    feature.append(freq_cialis_feature(text, freq))
    feature.append(freq_sex_feature(text, freq))
    feature.append(freq_sexual_feature(text, freq))
    feature.append(freq_penis_feature(text, freq))
    feature.append(freq_havebeenselected_feature(text, freq))
    feature.append(freq_percent_feature(text, freq))
    feature.append(freq_www_feature(text, freq))
    feature.append(freq_100_feature(text, freq))
    feature.append(freq_re_feature(text, freq))
    feature.append(freq_cc_feature(text, freq))

    return feature

# This method generates a design matrix with a list of filenames
# Each file is a single training example
def generate_design_matrix(filenames):
    design_matrix = []
    for filename in filenames:
        with open(filename) as f:
            text = f.read() # Read in text from file
            text = text.replace('\r\n', ' ') # Remove newline character
            words = re.findall(r'\w+', text)
            word_freq = defaultdict(int) # Frequency of all words
            for word in words:
                word_freq[word] += 1

            # Create a feature vector
            feature_vector = generate_feature_vector(text, word_freq)
            design_matrix.append(feature_vector)
    return design_matrix

# ************** Script starts here **************
# DO NOT MODIFY ANYTHING BELOW

spam_filenames = glob.glob(BASE_DIR + SPAM_DIR + '*.txt')
spam_design_matrix = generate_design_matrix(spam_filenames)
```

```
ham_filenames = glob.glob(BASE_DIR + HAM_DIR + '*.txt')
ham_design_matrix = generate_design_matrix(ham_filenames)
# Important: the test_filenames must be in numerical order as that is the
# order we will be evaluating your classifier
test_filenames = [BASE_DIR + TEST_DIR + str(x) + '.txt' for x in range(1, NUM_TEST_EXAMPLE
test_design_matrix = generate_design_matrix(test_filenames)

X = spam_design_matrix + ham_design_matrix
Y = [1]*len(spam_design_matrix) + [0]*len(ham_design_matrix)

file_dict = {}
file_dict['training_data'] = X
file_dict['training_labels'] = Y
file_dict['test_data'] = test_design_matrix
scipy.io.savemat('spam_data.mat', file_dict)
```