#2.

$$\tilde{x} \longrightarrow \begin{array}{c} \begin{array}{ccc} x_{i1} & x_{i2} & \text{bias} \end{array} \\ \begin{pmatrix} 0 & 3 & 1 \\ 1 & 3 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix} \end{array} \qquad \tilde{w} \longrightarrow \begin{pmatrix} w_1 \\ w_2 \\ \alpha \end{pmatrix} \qquad y = \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \end{pmatrix}.$$

(a)

$$w^{(0)} = \begin{pmatrix} -2 \\ 1 \\ 0 \end{pmatrix}. \Rightarrow R\left(w^{(0)}\right) = ?$$

$$R\left(w^{(0)}\right) = -\sum_{i=1}^{4}\left[y_i \ln\left(s\left(w^T X_i\right)\right) + (1-y_i)\ln\left(1 - s\left(w^T X_i\right)\right)\right].$$

$\underline{i=1}$ $\quad s(w^T X_1) = s\left((-2\ 1\ 0)\begin{pmatrix} 0 \\ 3 \\ 1 \end{pmatrix}\right) = s(3)$

$$= \frac{1}{1+e^{-3}} \approx \cancel{0.953}\ 0.953.$$

$\underline{i=2}$ $\quad s(w^T X_2) = s\left((-2\ 1\ 0)\begin{pmatrix} 1 \\ 3 \\ 1 \end{pmatrix}\right) = s(1) = \frac{1}{1+e^{-1}} \approx 0.731.$

$\underline{i=3}$ $\quad s(w^T X_3) = s\left(\begin{pmatrix} -2 \\ 1 \\ 0 \end{pmatrix} \cdot \begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix}\right) = s(1) \approx 0.731.$?

$\underline{i=4}$ $\quad s(w^T X_4) = s\left(\begin{pmatrix} -2 \\ 1 \\ 0 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}\right) = s(-1) \approx 0.269.$

$$\Rightarrow R\left(w^{(0)}\right) \approx -\left(\ln(0.953) + \ln(0.731) + \ln(1-0.731) + \ln(1-0.269)\right)$$

$$= -(-1.99) = 1.99.$$

(b)

$$\mu^{(0)} = \begin{pmatrix} P\left(Y=1 \mid X = X_1\right) \\ \vdots \\ P\left(Y=1 \mid X = X_4\right) \end{pmatrix} = \begin{pmatrix} 1/1 - e^{-\left(w^{(0)T} X_1\right)} \\ \vdots \\ 1/1 - e^{-\left(w^{(0)T} X_4\right)} \end{pmatrix} \approx \begin{pmatrix} 0.953 \\ 0.731 \\ 0.731 \\ 0.269 \end{pmatrix}.$$

#2.

(c) $$w^1 \leftarrow w^0 + 1 \cdot \sum_{i=1}^{4} \left(y_i - s(x_i^T w)\right) x_i$$

$$= \begin{pmatrix} -2 \\ 1 \\ 0 \end{pmatrix} + 1 \cdot \left((1 - (0.953)) x_1 + (1 - 0.73) x_2 + (-0.731) x_3 + (-0.269) x_4 \right)$$

$$= \begin{pmatrix} -2 \\ 1 \\ 0 \end{pmatrix} + 0.047 \begin{pmatrix} 0 \\ 3 \\ 1 \end{pmatrix} + 0.269 \begin{pmatrix} 1 \\ 3 \\ 1 \end{pmatrix} - 0.731 \begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix} - 0.269 \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$$

$$= \begin{pmatrix} -2 \\ 1 \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ 3 \cdot (0.047 + 0.269) - (.731 + .269) \\ 0.047 + \cancel{0.269} - .731 - \cancel{0.269} \end{pmatrix} = \begin{pmatrix} 0 \\ -0.052 \\ -0.684 \end{pmatrix} + \begin{pmatrix} -2 \\ 1 \\ 0 \end{pmatrix}$$

$$= \boxed{\begin{pmatrix} -2 \\ 0.948 \\ -0.684 \end{pmatrix} = w^{(1)}}$$

(d) $R(w^{(1)}) = 1.72$ (see code).

(e) $\mu^{(1)} = \begin{pmatrix} 0.897 \\ 0.5401 \\ 0.566 \\ 0.15 \end{pmatrix}$ "

(f) $w^{(2)} = \begin{pmatrix} -1.69 \\ 1.92 \\ -0.837 \end{pmatrix}$ "

(g) $R(w^{(2)}) = 1.85$ "

① (a) $\boxed{\text{Show } \hat{\alpha} = \bar{y}.}$

$\bar{x} = \bar{0}$, $X = \begin{pmatrix} — x_1^T — \\ \vdots \\ — x_n^T — \end{pmatrix}$, $y = \begin{pmatrix} y_1 \\ \vdots \\ y_n \end{pmatrix}$,
$J(w, \alpha) = \|Xw + \alpha 1 - y\|_2^2 + \lambda \|w\|_2^2$.
$\alpha \in \mathbb{R}$.

Find min:

$$\frac{\partial}{\partial \alpha} J(w,\alpha) = \frac{\partial}{\partial \alpha}\left((Xw + \alpha 1 - y)^T (Xw + \alpha 1 - y) \right) = 0$$

$$= \frac{\partial}{\partial \alpha}\Big(\underline{(Xw)^T Xw} + \underline{(Xw)^T \alpha 1} - \underline{(Xw)^T y} + \underline{(\alpha 1^T)\alpha 1} + \underline{\alpha 1^T Xw} - \alpha 1^T y - \underline{y^T Xw} - y^T \alpha 1 + y^T y \Big)$$

$$= \frac{\partial}{\partial \alpha}\Big(\cancel{\|Xw\|_2^2}^{\to 0} + 2\alpha 1^T Xw - \cancel{2y^T Xw}^{\to 0} + n \cdot \alpha^2 - 2\alpha 1^T y + \cancel{\|y\|_2^2}^{0} \Big)$$

$$= \frac{\partial}{\partial \alpha}\left(2\alpha 1^T Xw + n\alpha^2 - 2\alpha 1^T y \right) = 2\alpha n + 2 1^T Xw - 2 1^T y = 0$$

dim of samples

$$\iff \alpha = \frac{1^T y - 1^T Xw}{n} = \bar{y} - \frac{1}{n}\sum_{i=1}^{n} (x_i^T w) = \bar{y} - \sum_{j=1}^{n'}\sum_{i=1}^{n} x_{ij} w_j \left(\frac{1}{n}\right)$$

$$= \bar{y} - \sum_{j=1}^{n'} \left(\frac{x_{1j} + x_{2j} + \ldots + x_{nj}}{n} \right) w_j = \bar{y} - \sum_{j=1}^{n'} (0 \cdot w) = \bar{y}.$$

since $\bar{x} = \bar{0}$.

Hence, $\hat{\alpha} = \bar{y}$.

$\boxed{\text{Show } \hat{w} = (X^T X + \lambda I)^{-1} X^T y.}$

$$\frac{\partial}{\partial w}(J(w,\alpha)) = \frac{\partial}{\partial w}\left((Xw)^T Xw + 2\alpha 1^T Xw - 2y^T Xw + \lambda w^T w \right) = 2X^T Xw + 2\alpha X^T 1 - 2X^T y + 2\lambda w.$$

Since $\bar{x} = \bar{0}$, $X^T 1 = n\bar{x} = \bar{0} \Rightarrow \frac{\partial}{\partial w} J = 2X^T Xw - 2X^T y + 2\lambda w$. Set $\frac{\partial}{\partial w} = \bar{0}$ to min'ze:

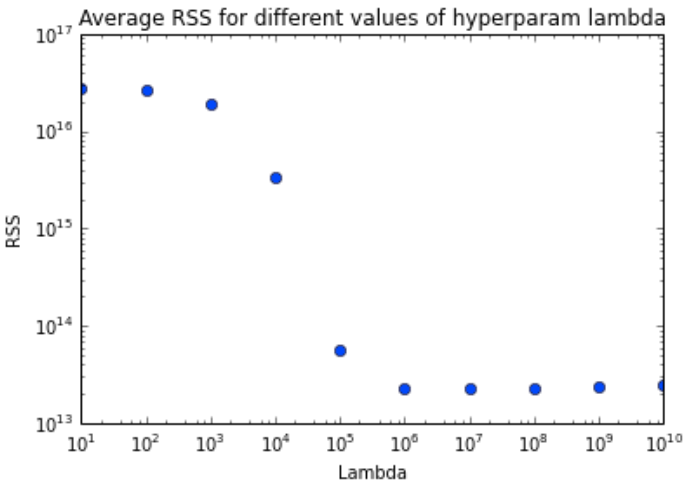
$$\frac{\partial}{\partial w} J = \bar{0} = X^T Xw - X^T y + \lambda w \iff w = (X^T X + \lambda I)^{-1} X^T y \Rightarrow \hat{w} =).$$

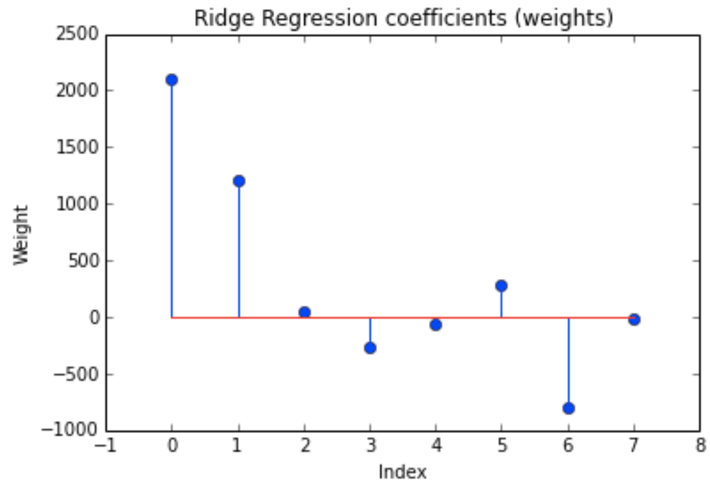
(b) See code.

Alex Walczak | CS 189 | HW 4

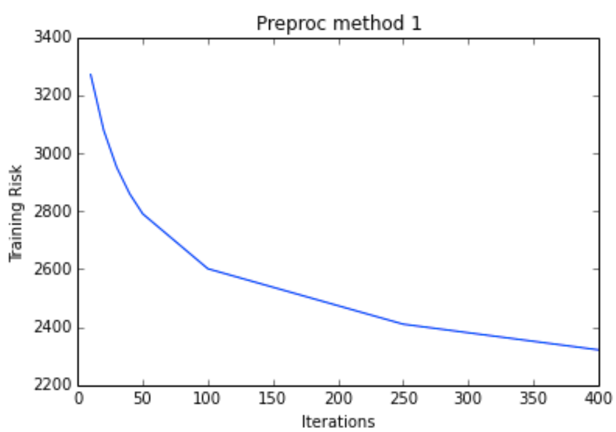
1. (b) RSS = $1.38e13. In HW3, RSS = $1.53e16, which is about 1,000 times larger.

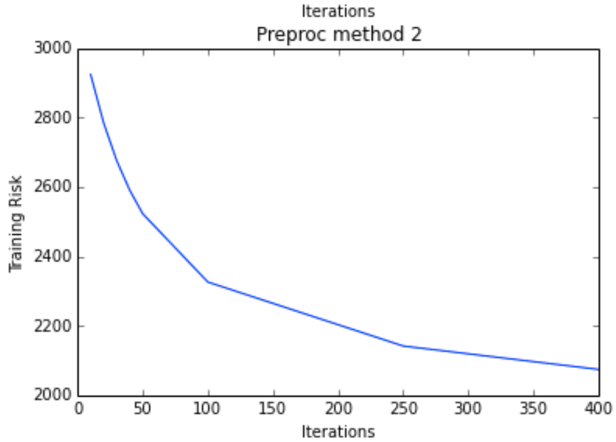
In HW3 the weights had a range from -40,000 to 40,000 -- this is about twenty times as large as with Ridge Regression. Since we are doing Ridge Regression, we penalize larger weight values so we do not allow for the high variance (overfitted) result of Linear Regression.

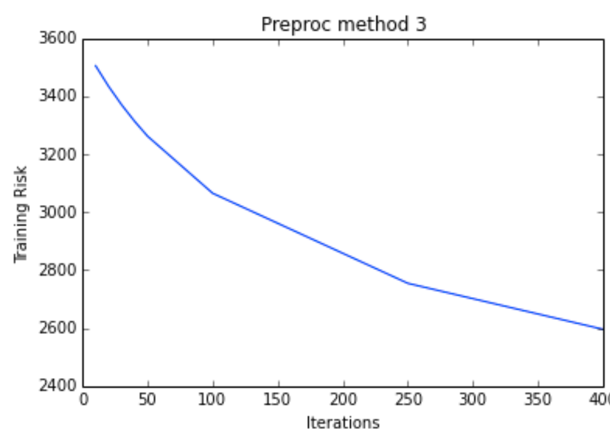




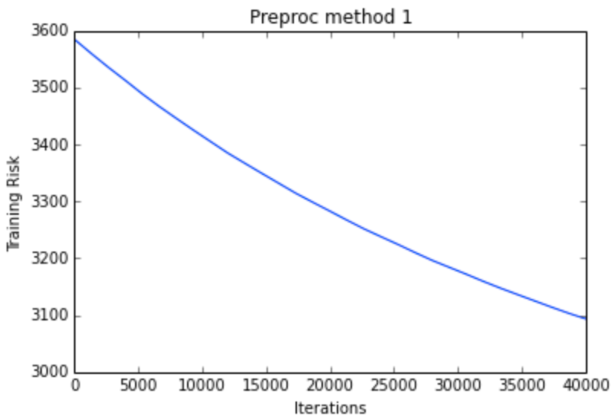
3. (1)

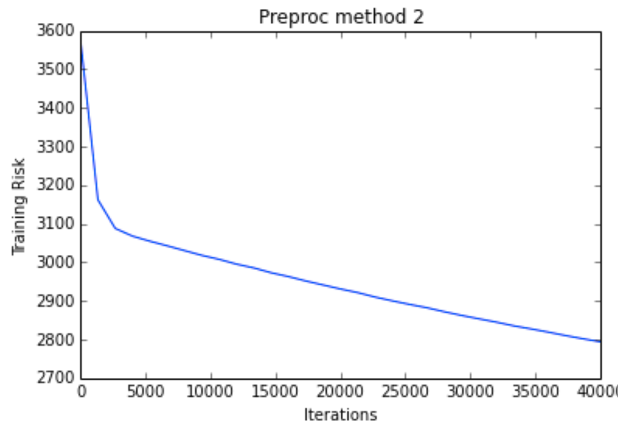


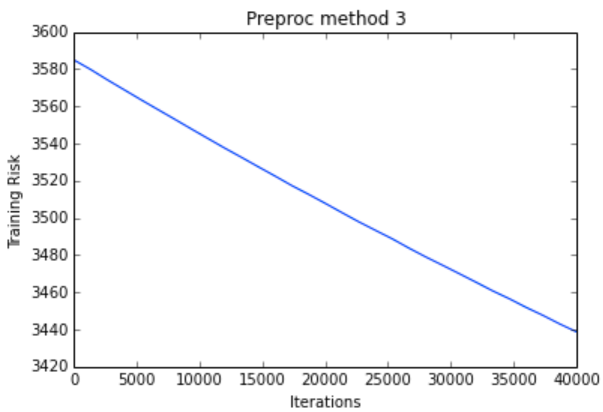




3. (2)

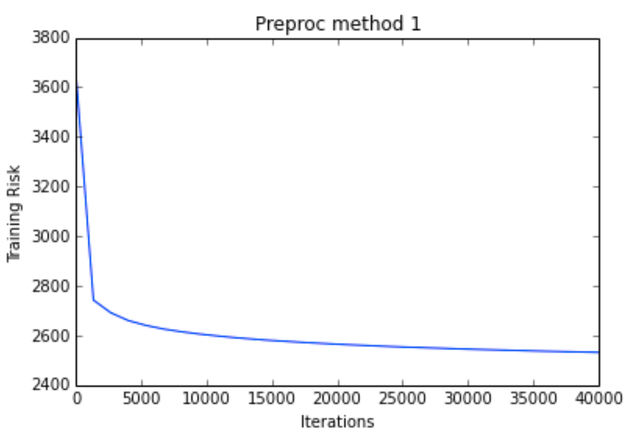


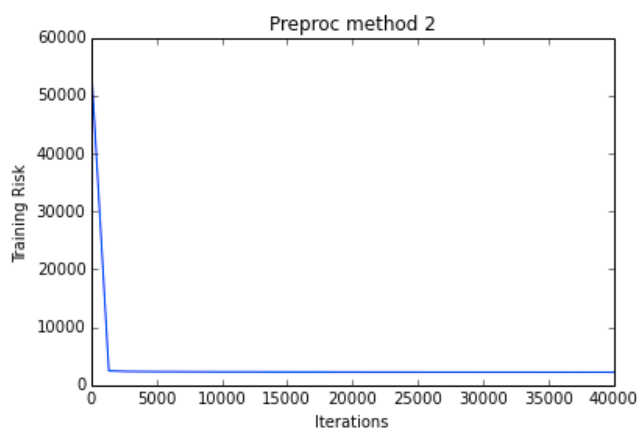


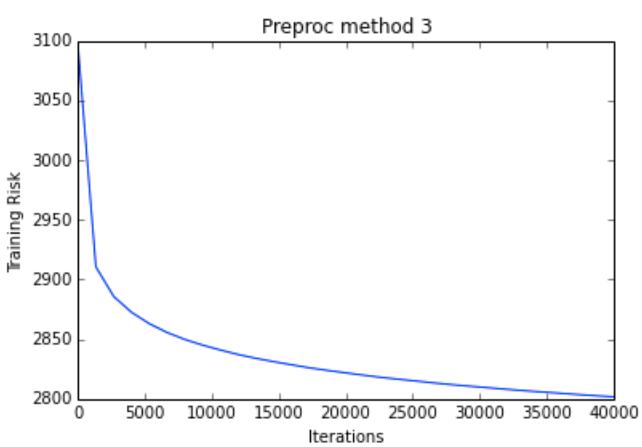


The plots compared to part one have less steep slopes for the same number of iterations; however, a single iteration in batch gradient ascent corresponds to thousands of times more calculations than a stochastic gradient ascent iteration. The amount of time to generate the plots in part two was much shorter than the time required in part one.

3. (3)



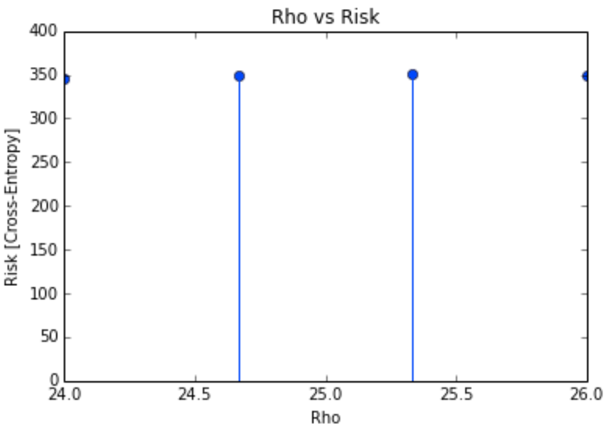




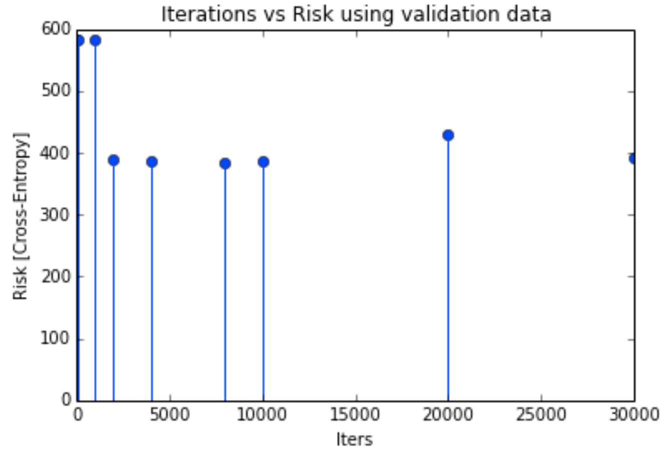
The variable learning rate is a good idea if the number of iterations doesn’t become extremely large. With the variable rate, the convergence to a smaller value in risk happens much more quickly. However, the ability to make significant updates to the weights after 20,000 iterations is lost. So the algorithm can get stuck on a certain weight vector early on when a better one may still exist.

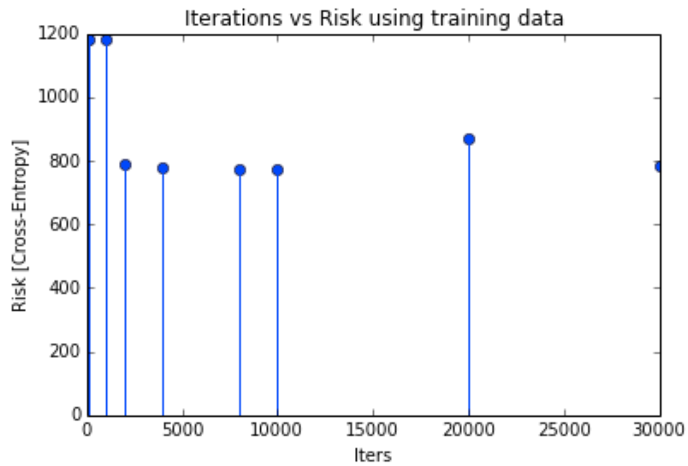
3. (4)

Quadratic Kernel:
After running 10-fold cross-validation, I settled on rho = 21.8. Lambda = 1e-3 worked well, as did the learning rate epsilon = 1. Both methods used the second preprocessing method.

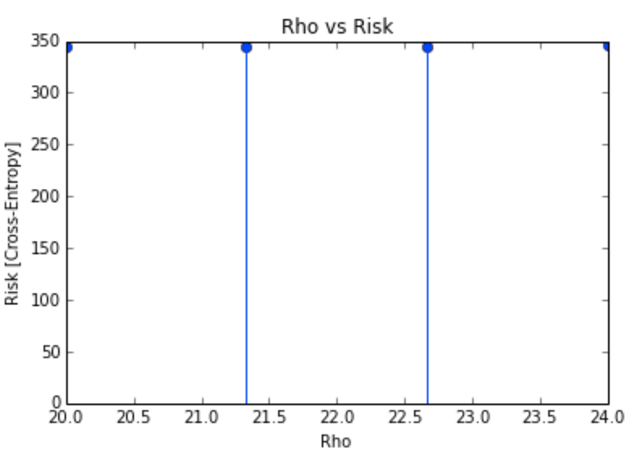


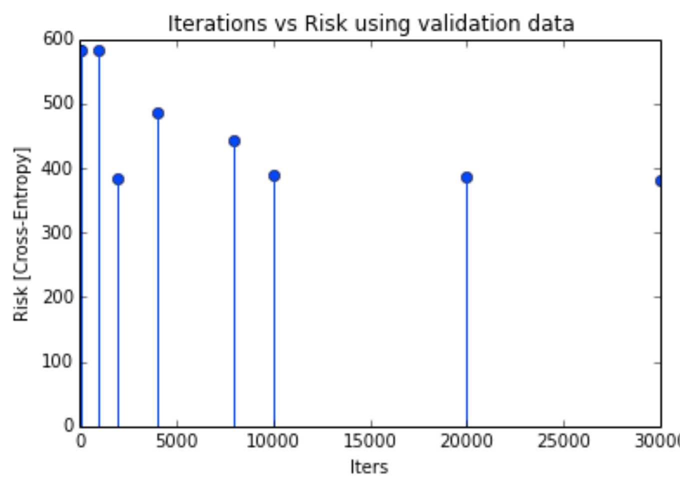
Note: the values of the risk for validation and training data are opposite of what is expected (here, validation < training) because the size of the validation data is half the size of the training data. One could normalize the risk function to be able to compare the two, but you can just multiply one or the other by 2 or 0.5 for the same effect.

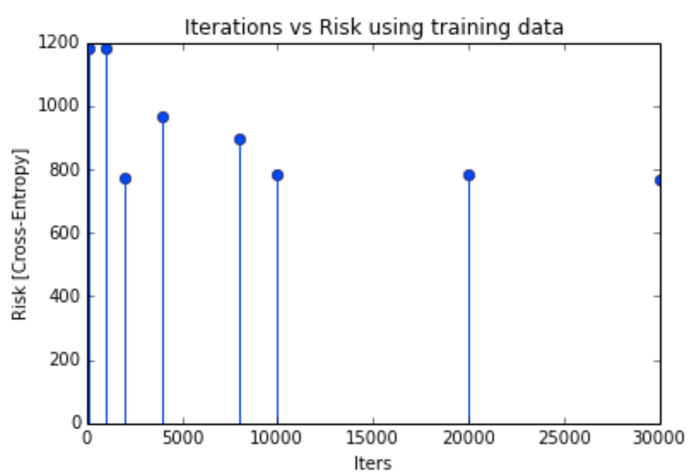




Linear Kernel:
rho = 21.8. Lambda = 1e-3, learning rate epsilon = 1.







3. (5)

Kaggle Score = 0.76905 (highest using linear kernel)

#4.

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

So, $g(z) = \left(s(2z) - \frac{1}{2}\right) + \frac{1}{2} = \left(\frac{1 \cdot e^z}{1+e^{-2z} \cdot e^z} - \frac{1}{2}\right) + \frac{1}{2}$

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

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

(b) $g'(z) = \frac{d}{dz} g(z)$ in terms of tanhz. from (a), (& by Quotient Rule)

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

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

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

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

Since ~~$g(z)$~~ $\tanh z = 2s(2z) - 1$

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

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

then $\boxed{g'(z) = \frac{1}{2}\left(1 - (\tanh z)^2\right)}.$

#4. (c) $J(w) = \sum_{i=1}^{n} y_i \ln(g(X_i^T w)) + (1-y_i)\ln(1-g(X_i^T w))$

Batch gradient ascent rule is: gradient w.r.t. w not a dir derivative!

$w \leftarrow w - \lambda \nabla_w J(w)$. Solve for $\nabla_w J(w)$.

Rewrite $J(w)$ by substituting $f_i(w) = X_i^T w$. Note:

$\nabla f_i(w) = X_i$. So $\nabla_w J(w) = \sum y_i \cdot \frac{1}{g(f_i(w))} \cdot g'(f_i(w)) \cdot \nabla f_i(w) + \frac{(1-y_i)}{(1-g(f_i(w)))} \cdot \nabla(1-g_i(f_i(w)))$

$$= \sum \frac{y_i}{g(f_i(w))} \cdot \left(\frac{1}{2}\left(1-[\tanh(f_i(w))]^2\right)\right) \cdot X_i + \frac{1-y_i}{1-g(f_i(w))} \cdot \left(\frac{-1}{2}\left(1-[\tanh(f_i(w))]^2\right)\right) \cdot X_i$$

$$= \sum \left(\frac{y_i}{g(f_i(w))} - \frac{1-y_i}{1-g(f_i(w))}\right) \cdot \left(\frac{1}{2}\left(1-[\tanh(f_i(w))]^2\right)\right) X_i$$

$$= \sum_{i=1}^{n} \left(\frac{y_i}{g(X_i^T w)} - \frac{1-y_i}{1-g(X_i^T w)}\right) \cdot \left(\frac{1}{2}\left(1-[\tanh(X_i^T w)]^2\right)\right) X_i .$$

5. The linear SVM does not utilize the add feature well because the feature does not provide additional linearly separable data to the SVM. A feature containing milliseconds since the previous midnight will have many spam samples concentrated symmetrically around the origin for that dimension (this is because spam spikes around midnight). Ham will be less dense in this region, and a there will be no clear boundary between the two in this dimension.

Daniel should use the quadratic kernel to increase the size of the feature space. This way, the SVM may be able to find nonlinear boundaries among the samples.

In [3]:
```
# Alex Walczak | CS 189 | Homework 4

# Import functions and libraries
from __future__ import division
import numpy as np, matplotlib.pyplot as plt
from matplotlib.pyplot import *
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
from matplotlib.ticker import LinearLocator, FormatStrFormatter
import scipy.io
from scipy import signal
from mpl_toolkits.axes_grid1 import make_axes_locatable
from pylab import rcParams
rcParams['figure.figsize'] = 7, 7
%matplotlib inline
```

In [501]:
```
### 1 (b)
```

In [88]:
```
from sklearn.cross_validation import KFold
```

In [ ]:
```
# part i
```

In [7]:
```
# Import data
housing_data = scipy.io.loadmat("housing_dataset/housing_data.mat")
htraining_data = housing_data["Xtrain"]
htraining_labels = housing_data["Ytrain"][:,0]
hvalidation_data = housing_data["Xvalidate"]
hvalidation_labels = housing_data["Yvalidate"][:,0]
```

In [44]:
```
def center_data(data):
    return data - np.mean(data, axis = 0)
```

In [53]:
```
def ridge_reg_model(tdata, tlabels, lam):
    tdata = center_data(tdata)
    alpha_hat = np.mean(tlabels)
    w_hat = np.linalg.inv(tdata.T.dot(tdata) + lam*np.eye(htraining
_data.shape[1])).dot((tdata.T.dot(tlabels)))
    return alpha_hat, w_hat
```

In [54]:
```
housing_model = ridge_reg_model(htraining_data, htraining_labels, l
am=12)
```

In [58]:
```
housing_model
```

Out[58]:
```
(206917.56702674896,
 array([  4.05932505e+04,   1.19721890e+03,  -8.51004253e+00,
          1.18216724e+02,  -3.77941709e+01,   4.32822634e+01,
         -4.20870778e+04,  -4.23585160e+04]))
```

In [63]:
```
def ridge_reg_prediction(data, model):
    alpha_hat, w_hat = model
    prediction = data.dot(w_hat) + alpha_hat
    return prediction
```

In [ ]:
```
# part ii (K-fold CV to find appropriate labmda)
```

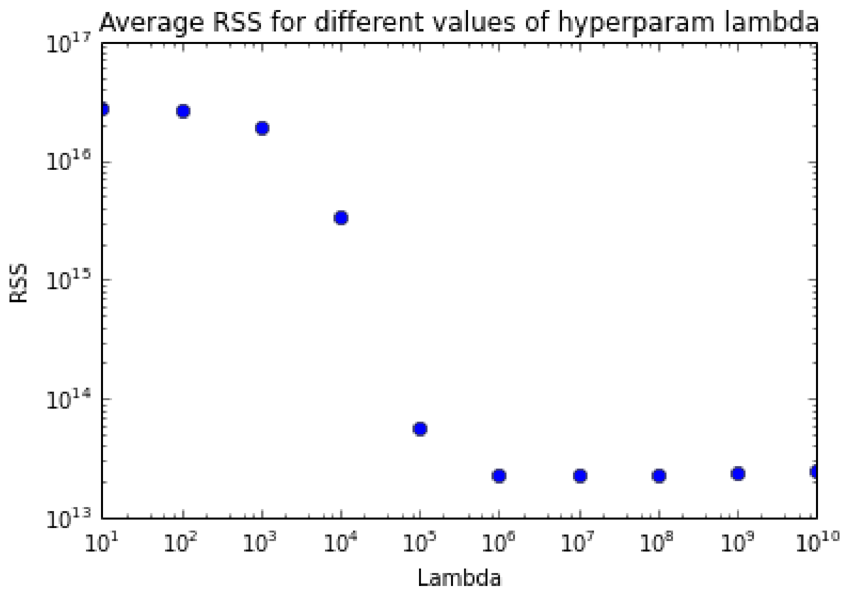
In [170]:
```
kf = KFold(htraining_data.shape[0], n_folds=10, shuffle = True)
```

In [183]:
```
RSS = lambda predn, true: np.linalg.norm(predn - true)**2
```

In [184]:
```
lams = [10,100,1000,1e4,1e5,1e6,1e7,1e8,1e9,1e10]
lam_errors = np.zeros(len(lams))
for i in range(len(lams)):
    error = 0
    for train, test in kf:
        ktrain = htraining_data[train]
        ktrain_label = htraining_labels[train]
        ktest = htraining_data[test]
        ktrue = htraining_labels[test]

        khousing_model = ridge_reg_model(ktrain, ktrain_label, lam=
lams[i])
        kpredn = ridge_reg_prediction(ktest, khousing_model)

        error += RSS(kpredn, ktrue)*0.1
    lam_errors[i] = error
```

In [185]:
```
plt.stem(lams,lam_errors)
plt.yscale('log')
plt.xscale('log')
plt.xlabel('Lambda')
plt.ylabel('RSS')
plt.title('Average RSS for different values of hyperparam lambda')
```

Out[185]: <matplotlib.text.Text at 0x10ea6c5d0>

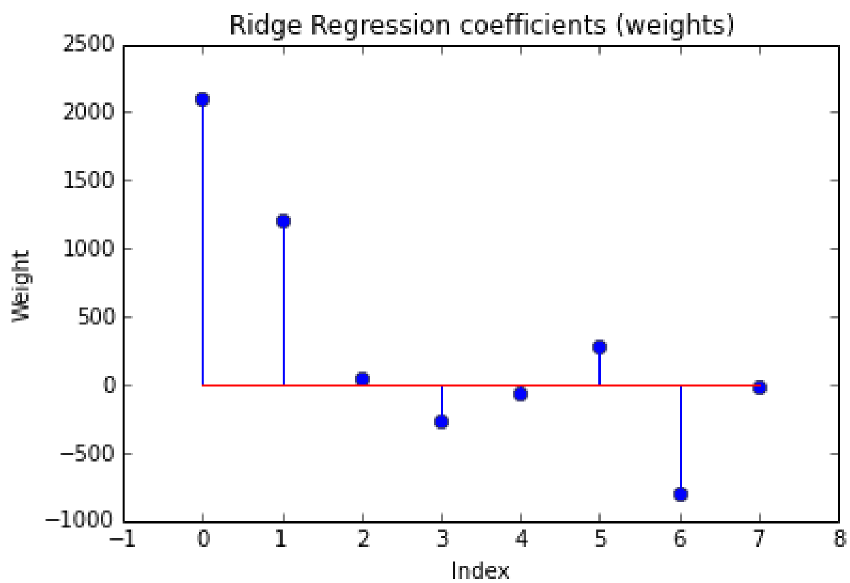


In [187]:
```
lam_hat = 1e6
housing_model = ridge_reg_model(htraining_data, htraining_labels, l
am_hat)
predn = ridge_reg_prediction(hvalidation_data, housing_model)

RSS(predn, hvalidation_labels)
# $13840240904169.389 ~ $1.38e13
# Compare to HW3 $1.53e16

weights = housing_model[1]
```

In [197]:
```
# part iii

# In HW3 the weights had a range from -40,000 to 40,000 -- this is
about twenty times as large as with Ridge Regression.
# Since we are doing Ridge Regression, we penalize larger weight va
lues so we do not allow for the high variance (overfitted)
# result of Linear Regression.
```

In [198]:
```
plt.figure()
plt.stem(weights)
plt.xlim([-1,8])
plt.title("Ridge Regression coefficients (weights)")
plt.xlabel("Index")
plt.ylabel("Weight")
```

Out[198]: <matplotlib.text.Text at 0x10f1641d0>



In [234]:
```
# 2 part ...
```

In [490]:
```
def SS(g):
    return 1/(1+np.exp(-g) + 0.00000000001) # To avoid division by
zero
```

In [491]:
```
def RR(w,y,X):
    R = 0
    for i in range(len(y)):
        R += ((y[i]*np.log(SS(w.T.dot(X[i])))) + (1-y[i])*(np.log(1
-SS(w.T.dot(X[i])))))
    return -R
```

In [258]:
```
def UPDATE_w(w_prev, eps, X, y):
    w_new = np.zeros(3)
    to_add = np.zeros(3)
    for i in range(len(y)):
        to_add += (y[i]-SS(w_prev.T.dot(X[i])))*X[i].T
    w_new = w_prev + eps*(to_add)
    return w_new
```

In [263]:
```
def MU(X,w):
    return np.array([SS(w.T.dot(X[0])), SS(w.T.dot(X[1])), SS(w.T.d
ot(X[2])), SS(w.T.dot(X[3]))])
```

In [273]:
```
w_0 = np.array([-2,1,0])
y = np.array([1,1,0,0])
X = np.array([[0, 3, 1],[1, 3, 1],[0, 1, 1],[1, 1, 1]])
RR(w_0,y,X) # == 1.9883724141284103
w_1 = UPDATE_w(w_0, 1, X, y) # == array([-2.0,  0.94910188, -0.6836
3271])
RR(w_1,y,X) # == 1.7206170956213045
MU(X,w_1) # == array([ 0.89693957,  0.54082713,  0.56598026,  0.150
00896])
w_2 = UPDATE_w(w_1, 1, X, y) # == array([-1.69083609,  1.91981257,
-0.83738862])
RR(w_2,y,X) # == 1.8546997847922486
```

Out[273]: 1.8546997847922486

In [ ]:
```
# 3 part 1.
```

In [1126]:
```
# Import data
spam_data = scipy.io.loadmat("spam_dataset/spam_data.mat")
straining_data = spam_data["training_data"] + 0.0
straining_labels = spam_data["training_labels"][0,:]
stest_data = spam_data["test_data"] + 0.0
```

In [275]:
```
def batch_UPDATE_w(w_prev, eps, X, y):
    w_new = np.zeros(len(w_prev))
    to_add = np.zeros(len(w_prev))
    for i in range(len(y)):
        to_add += (y[i]-SS(w_prev.T.dot(X[i])))*X[i].T
    w_new = w_prev + eps*(to_add)
    return w_new
```

In [535]:
```
def stoch_UPDATE_w(w_prev, eps, X, y, idx):
    idx = idx % 5172
#     w_new = np.zeros(len(w_prev))
    to_add = (y[idx]-SS(w_prev.T.dot(X[idx])))*X[idx].T
    w_new = w_prev + eps*(to_add)
    return w_new
```

In [902]:

```
def preproc1(data):
    # Standardize each column to have mean 0 and unit variance.
    ctrd_data = data - np.mean(data, axis=0)
    stdzd_data = ctrd_data/np.std(data, axis=0)
    return stdzd_data

def preproc2(data):
    # Transform the features using Xij ← log(Xij + 0.1).
    transformed_data = np.log(data + 0.1)
    return transformed_data

def preproc3(data):
    # Binarize the features using Xij ← I(Xij > 0).
    data[data > 0] = 1
    data[data != 1] = 0
    return data

# Build and combine the preprocessed datasets.

sdata1 = preproc1(np.copy(straining_data))
sdata2 = preproc2(np.copy(straining_data))
sdata3 = preproc3(np.copy(straining_data))

preprocessed = [sdata1, sdata2, sdata3]
```

In [507]:

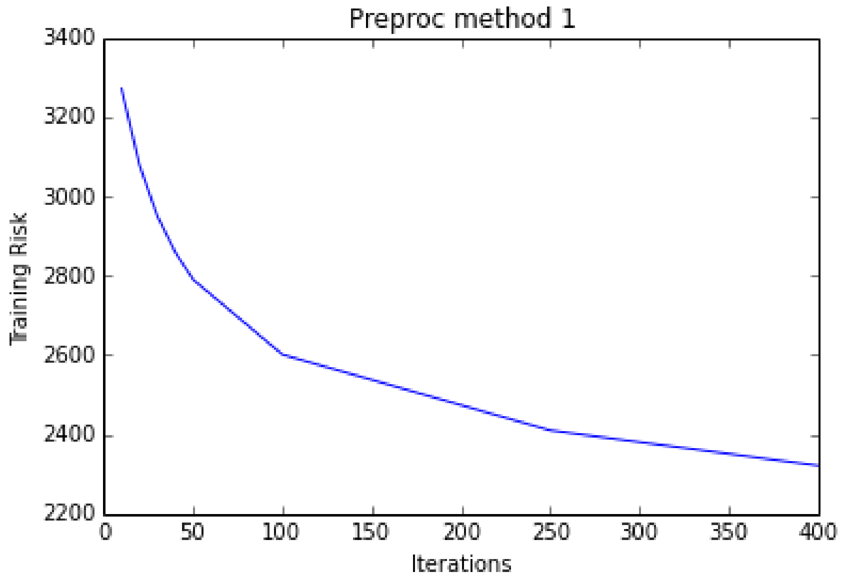
```
iterations = np.array([10,20,30,40,50,100,250,400])
eps = 0.00001
```

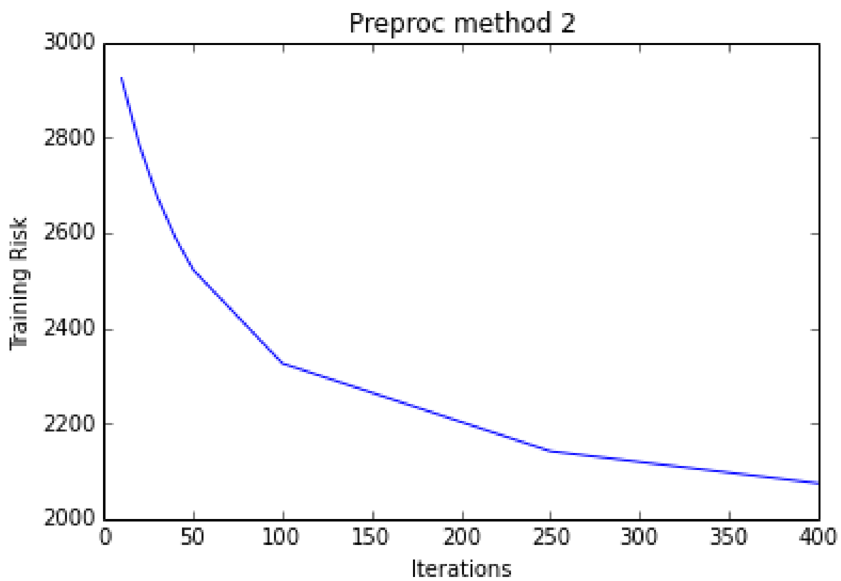
In [508]:

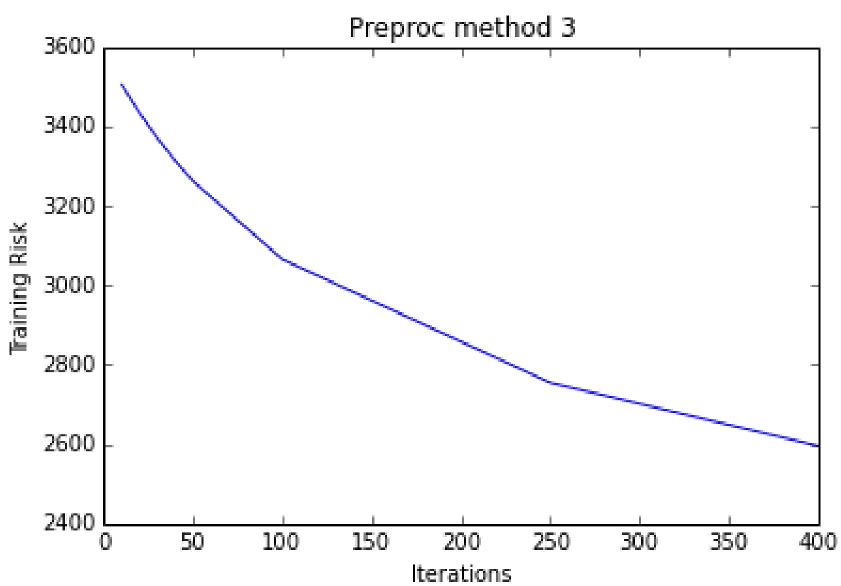
```
# 1. Batch Gradient Descent
```

In [509]:

```
for p in range(len(preprocessed)):
    data = preprocessed[p]
    risks = np.zeros(len(iterations))
    for it in range(len(iterations)):
        batch_w = np.zeros(32)
        for i in range(iterations[it]):
            batch_w = batch_UPDATE_w(batch_w, eps, data, straining_
labels)
        risks[it] = RR(batch_w, straining_labels, data)
    plt.figure()
    plt.plot(iterations, risks)
    plt.title('Preproc method ' + str(p+1))
    plt.ylabel('Training Risk')
    plt.xlabel('Iterations')
```


Preproc method 1
Training Risk
Iterations
3400
3200
3000
2800
2600
2400
2200
0
50
100
150
200
250
300
350
400


Preproc method 2
Training Risk
Iterations
3000
2800
2600
2400
2200
2000
0
50
100
150
200
250
300
350
400

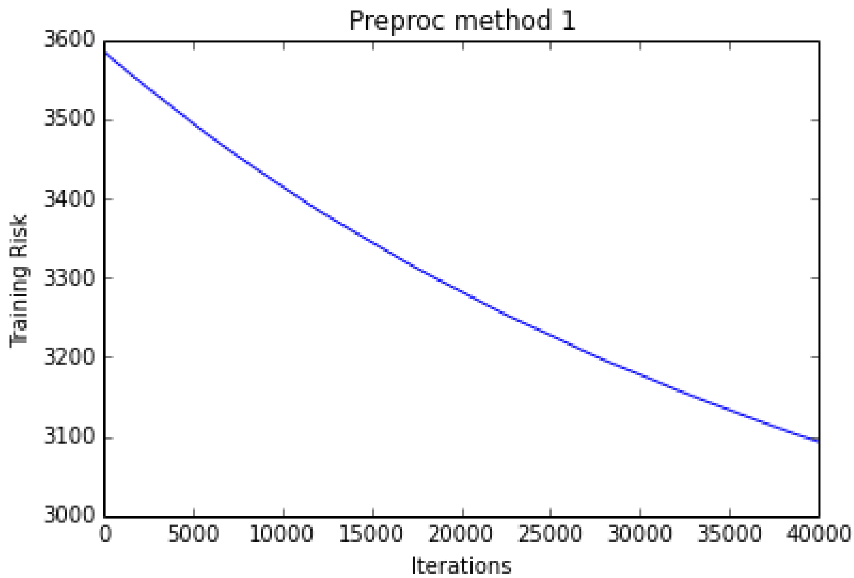

Preproc method 3
Training Risk
Iterations
3600
3400
3200
3000
2800
2600
2400
0
50
100
150
200
250
300
350
400

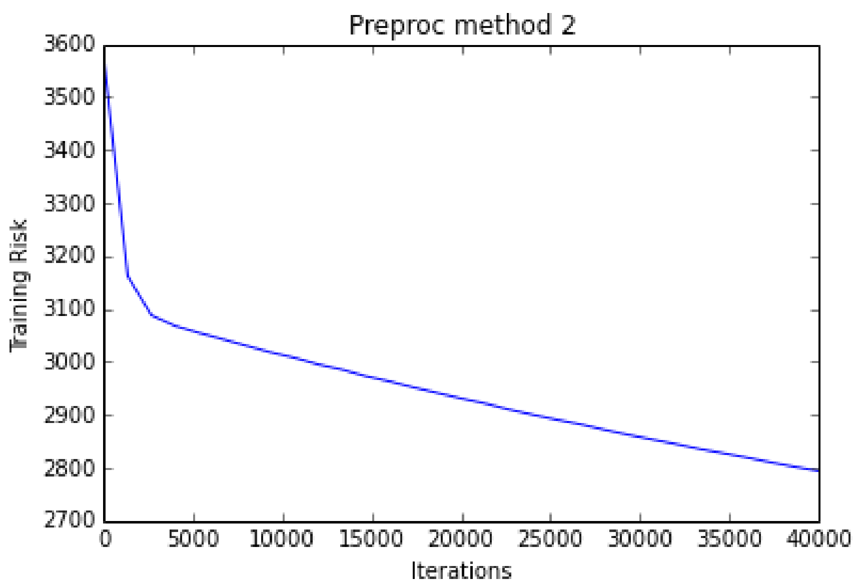
In [356]:
```
# 2. Stochastic Gradient Descent
```

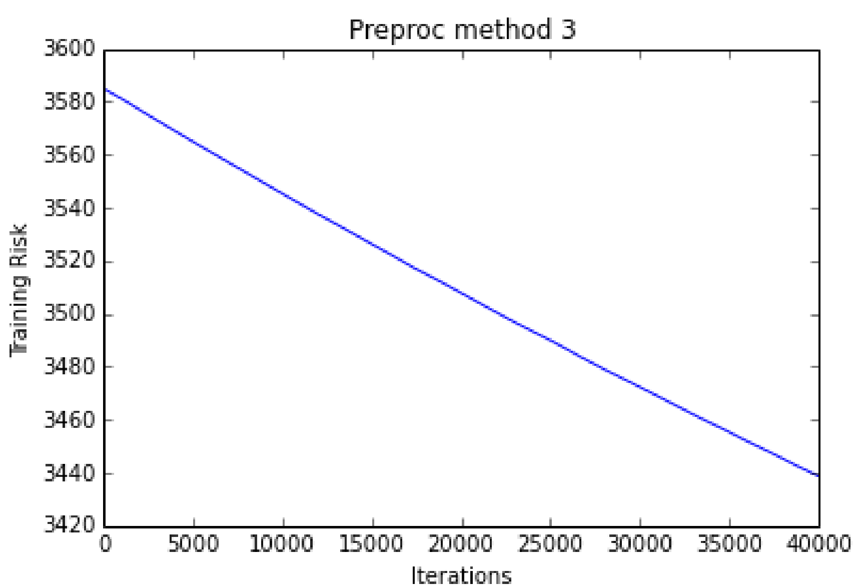
In [494]:
```
iterations = (np.linspace(10,40000,31)).astype('int')
eps = 0.001
```

In [502]:

```
for p in range(len(preprocessed)):
    data = preprocessed[p]
    risks = np.zeros(len(iterations))
    for it in range(len(iterations)):
        stoch_w = np.zeros(32)
        for i in range(iterations[it]):
            stoch_w = stoch_UPDATE_w(stoch_w, eps, data, straining_
labels, i)
        risks[it] = RR(stoch_w, straining_labels, data)
    plt.figure()
    plt.plot(iterations, risks)
    plt.title('Preproc method ' + str(p+1))
    plt.ylabel('Training Risk')
    plt.xlabel('Iterations')
```


Preproc method 1
Training Risk
3600
3500
3400
3300
3200
3100
3000
0
5000
10000
15000
20000
25000
30000
35000
40000
Iterations

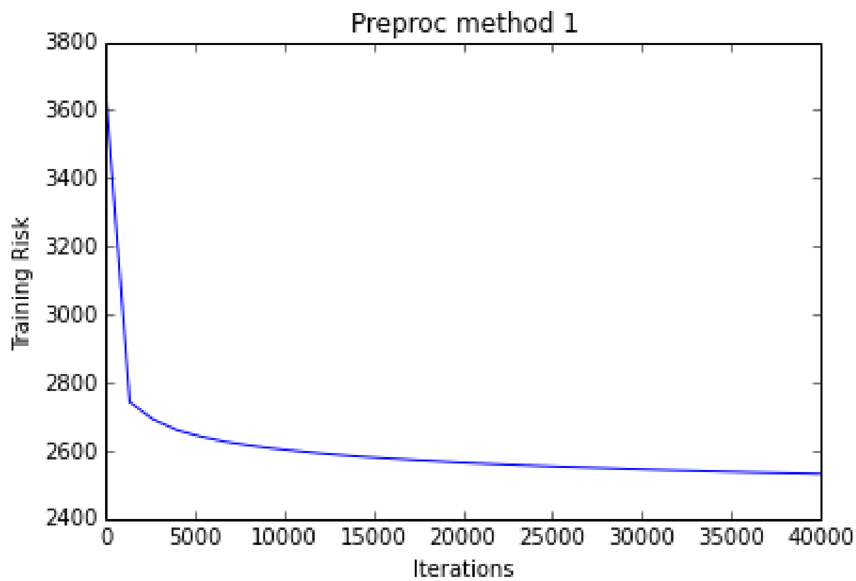

Preproc method 2
Training Risk
3600
3500
3400
3300
3200
3100
3000
2900
2800
2700
0
5000
10000
15000
20000
25000
30000
35000
40000
Iterations

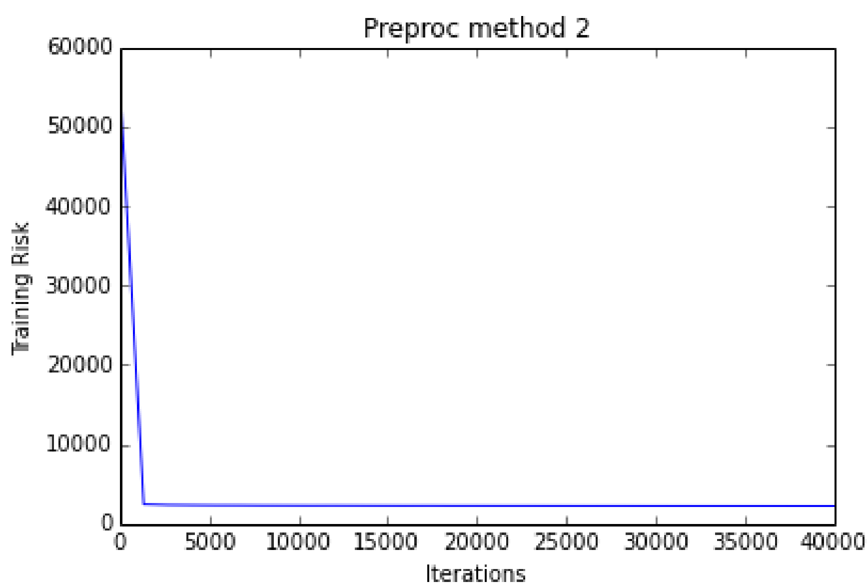

Preproc method 3
Training Risk
3600
3580
3560
3540
3520
3500
3480
3460
3440
3420
0
5000
10000
15000
20000
25000
30000
35000
40000
Iterations

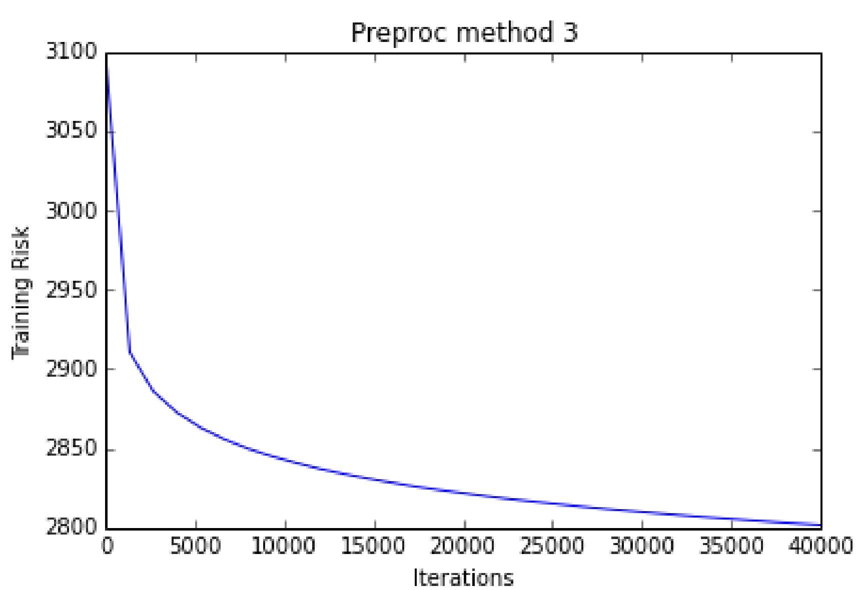
In [496]:
```
# 3. Variable Learning Rate + Stochastic Gradient Descent
```

In [ ]:
```
iterations = (np.linspace(10,40000,31)).astype('int')
```

In [500]:
```
for p in range(len(preprocessed)):
    data = preprocessed[p]
    risks = np.zeros(len(iterations))
    for it in range(len(iterations)):
        stoch_w = np.zeros(32)
        for i in range(iterations[it]):
            eps = 1/(i+1)
            stoch_w = stoch_UPDATE_w(stoch_w, eps, data, straining_
labels, i)
        risks[it] = RR(stoch_w, straining_labels, data)
    plt.figure()
    plt.plot(iterations, risks)
    plt.title('Preproc method ' + str(p+1))
    plt.ylabel('Training Risk')
    plt.xlabel('Iterations')
```


Preproc method 1
Training Risk
Iterations
3800
3600
3400
3200
3000
2800
2600
2400
0
5000
10000
15000
20000
25000
30000
35000
40000


Preproc method 2
Training Risk
Iterations
60000
50000
40000
30000
20000
10000
0
0
5000
10000
15000
20000
25000
30000
35000
40000


Preproc method 3
Training Risk
Iterations
3100
3050
3000
2950
2900
2850
2800
0
5000
10000
15000
20000
25000
30000
35000
40000

In [558]:
```
# Quadratic Kernel + Kernel Logistic Ridge Regression
```

In [988]:
```
kf2 = KFold(straining_data.shape[0], n_folds=10, shuffle = True)
kf3 = KFold(straining_data.shape[0], n_folds=3, shuffle = True)
```

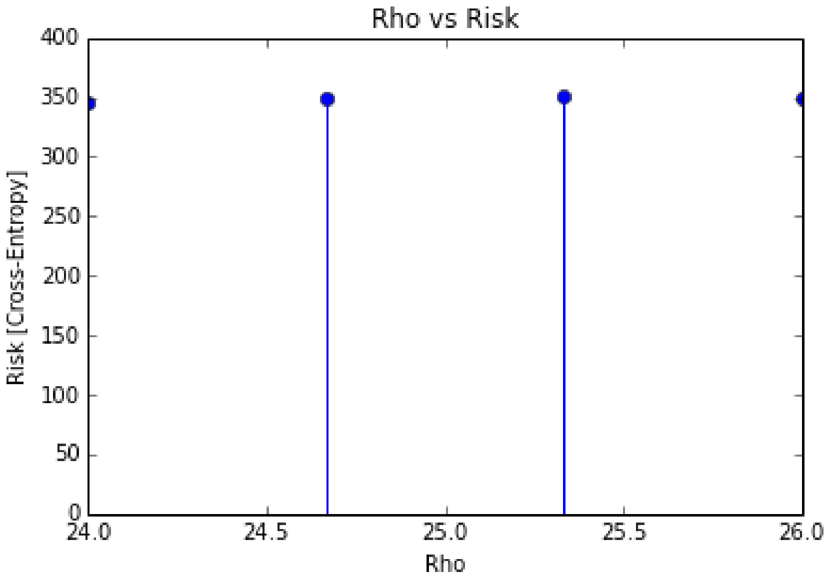
In [929]:
```
def build_kernel_matrix(data1, data2, rho, deg):
    return (data1.dot(data2.T)+rho)**deg

def kernel_update(a, Ka_i, y, i, eps, lam):
    i = i % len(y)
    a = a - eps*lam*a
    a[i] = a[i] + eps*(y[i] - SS(Ka_i))
    return a

def kernel_risk(Ka, y):
    return -np.sum((y*np.log(SS(Ka))) + (1-y)*(np.log(1-SS(Ka))))

def ker_predn(Ka):
    predn_vector = SS(Ka)
    predn_vector[predn_vector >= 0.5] = 1
    predn_vector[predn_vector != 1] = 0
    return predn_vector
```

In [1022]:
```
rand_inds = np.random.permutation(len(training_data))
```

In [1255]:

```
rhos = np.linspace(24,26,4)
eps = 1 # CAN BE VARIABLE.
lam = 1e-3
deg = 2
rho_errors = np.zeros(len(rhos))

training_data = np.copy(preprocessed[1])

for i in range(len(rhos)):
    error = 0
    for train, test in kf2:
        ktrain = training_data[rand_inds[train]]
        ktrain_label = straining_labels[rand_inds[train]]
        ktest = training_data[rand_inds[test]]
        ktrue = straining_labels[rand_inds[test]]

        a = np.ones(len(ktrain))

        Mdeg2 = build_kernel_matrix(ktrain, ktrain, rhos[i], deg)

        for j in range(3000): #iterations of gradient ascent.
            a = kernel_update(a, Mdeg2[j%len(Mdeg2)].dot(a), ktrai
n_label, j, eps, lam)

        K_predn = build_kernel_matrix(ktrain, ktest, rhos[i], deg)
        predn = ker_predn(K_predn.T.dot(a))

        error += kernel_risk(predn, ktrue)*(1/10)
    rho_errors[i] = error

plt.title('Rho vs Risk')
plt.xlabel('Rho')
plt.ylabel('Risk [Cross-Entropy]')
plt.stem(rhos,rho_errors)
```

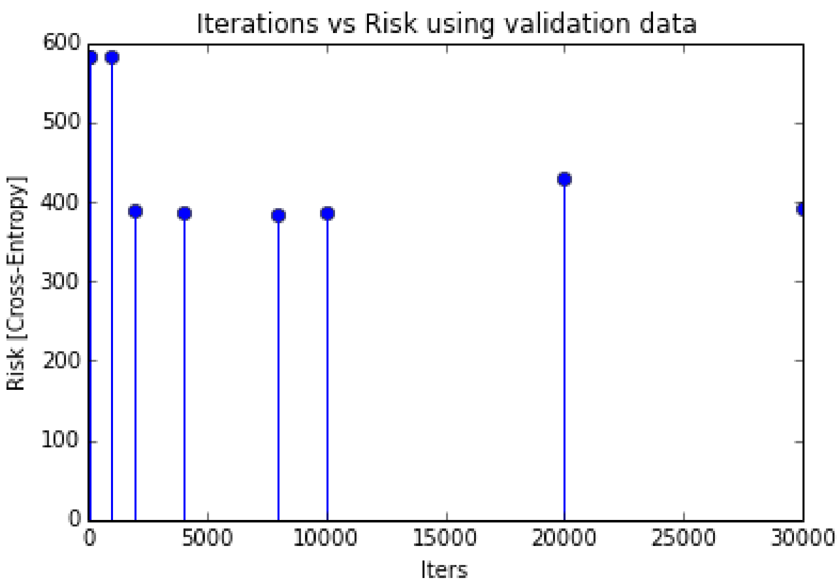
Out[1255]: <Container object of 3 artists>



In [1256]:

```
### plot risk vs. iterations for single rho value.

rho = 21.8
eps = 1
lam = 1e-3
deg = 2
iterations = [10, 100, 1000, 2000, 4000, 8000, 10000, 20000, 3000
0]
errors = np.zeros(len(iterations))
training_data = np.copy(preprocessed[1])

for train, test in kf3:

    ktrain = training_data[rand_inds[train]]
    ktrain_label = straining_labels[rand_inds[train]]
    ktest = training_data[rand_inds[test]]
    ktrue = straining_labels[rand_inds[test]]

    Mdeg2 = build_kernel_matrix(ktrain, ktrain, rho, deg)
    K_predn = build_kernel_matrix(ktrain, ktest, rho, deg)

    for i in range(len(iterations)):
        it = iterations[i]
        a = np.ones(len(ktrain))

        for j in range(it): #iterations of gradient ascent.
            a = kernel_update(a, Mdeg2[j%len(Mdeg2)].dot(a), ktrai
n_label, j, eps, lam)

        Ka_predn = K_predn.T.dot(a)
        predn = ker_predn(Ka_predn)
        errors[i] += kernel_risk(predn, ktrue)*(1/3)
    break

plt.title('Iterations vs Risk using validation data')
plt.xlabel('Iters')
plt.ylabel('Risk [Cross-Entropy]')
plt.stem(iterations, errors)
```

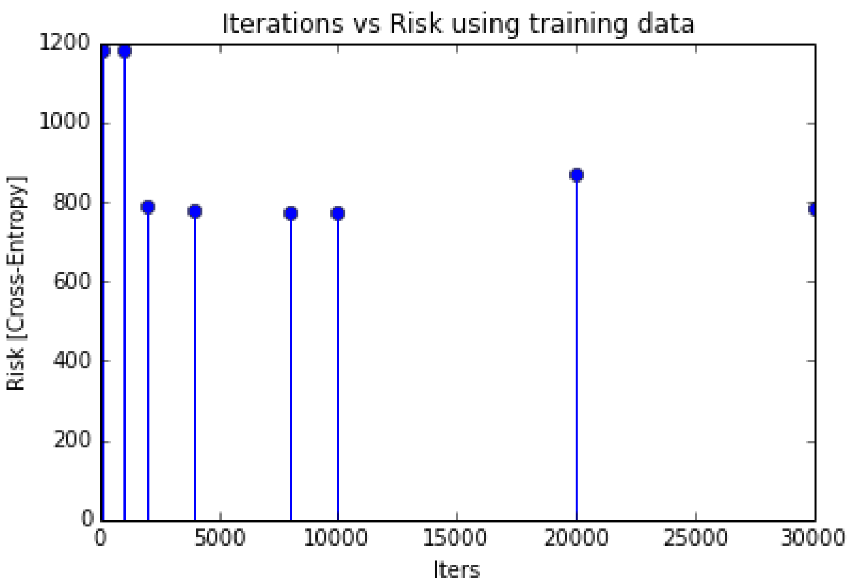
Out[1256]: <Container object of 3 artists>



In [1261]:

```
### plot risk vs. iterations for single rho value.

rho = 21.8
eps = 1
lam = 1e-3
deg = 2
iterations = [10, 100, 1000, 2000, 4000, 8000, 10000, 20000, 3000
0]
errors = np.zeros(len(iterations))
training_data = np.copy(preprocessed[1])

for train, test in kf3:

    ktrain = training_data[rand_inds[train]]
    ktrain_label = straining_labels[rand_inds[train]]
    ktest = training_data[rand_inds[test]]
    ktrue = straining_labels[rand_inds[test]]

    Mdeg2 = build_kernel_matrix(ktrain, ktrain, rho, deg)
    K_predn = build_kernel_matrix(ktrain, ktrain, rho, deg)

    for i in range(len(iterations)):
        it = iterations[i]
        a = np.ones(len(ktrain))

        for j in range(it): #iterations of gradient ascent.
            a = kernel_update(a, Mdeg2[j%len(Mdeg2)].dot(a), ktrai
n_label, j, eps, lam)

        Ka_predn = K_predn.T.dot(a)
        predn = ker_predn(Ka_predn)
        errors[i] += kernel_risk(predn, ktrain_label)*(1/3)
    break

plt.title('Iterations vs Risk using training data')
plt.xlabel('Iters')
plt.ylabel('Risk [Cross-Entropy]')
plt.stem(iterations, errors)
```

Out[1261]: <Container object of 3 artists>

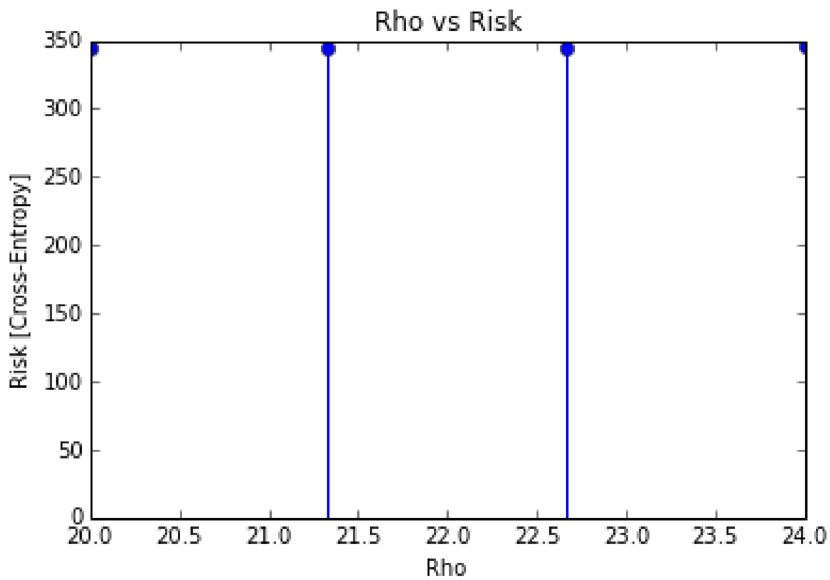


In [1129]:
```
# Linear Kernel
```

In [1250]:

```
rhos = np.linspace(20,24,4)
# rhos = [21.8]
eps = 1
lam = 1e-3
deg = 2.5
rho_errors = np.zeros(len(rhos))

training_data = np.copy(preprocessed[2])

for i in range(len(rhos)):
    error = 0
    for train, test in kf2:
        ktrain = training_data[rand_inds[train]]
        ktrain_label = straining_labels[rand_inds[train]]
        ktest = training_data[rand_inds[test]]
        ktrue = straining_labels[rand_inds[test]]

        a = np.ones(len(ktrain))

        Mdeg2 = build_kernel_matrix(ktrain, ktrain, rhos[i], deg)

        for j in range(3000): #iterations of gradient ascent.
            a = kernel_update(a, Mdeg2[j%len(Mdeg2)].dot(a), ktrai
n_label, j, eps, lam)

        K_predn = build_kernel_matrix(ktrain, ktest, rhos[i], deg)
        predn = ker_predn(K_predn.T.dot(a))

        error += kernel_risk(predn, ktrue)*(1/10)
    rho_errors[i] = error

plt.title('Rho vs Risk')
plt.xlabel('Rho')
plt.ylabel('Risk [Cross-Entropy]')
plt.stem(rhos,rho_errors)
```

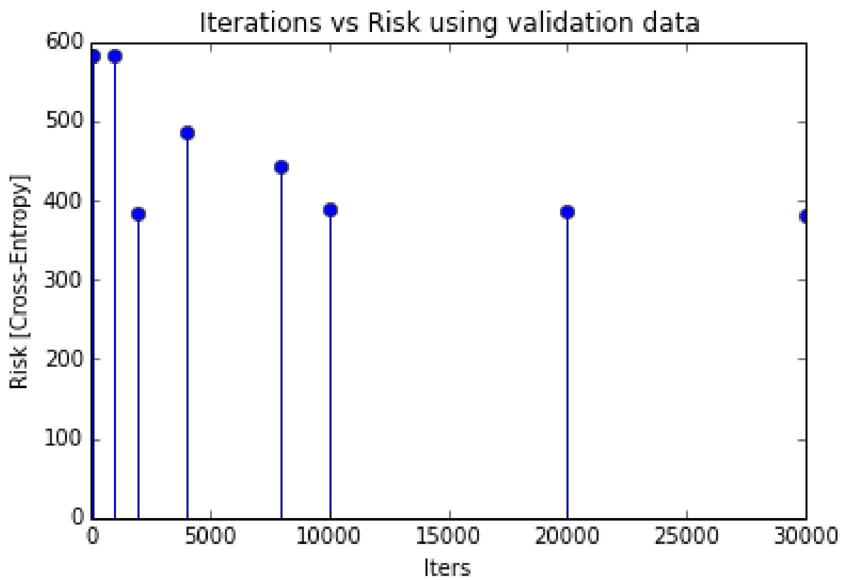
Out[1250]: <Container object of 3 artists>



In [1257]:

```
### plot risk vs. iterations for single rho value.
rho = 21.8
eps = 1
lam = 1e-3
deg = 1
iterations = [10, 100, 1000, 2000, 4000, 8000, 10000, 20000, 3000
0]
errors = np.zeros(len(iterations))
training_data = np.copy(preprocessed[2])

for train, test in kf3:

    ktrain = training_data[rand_inds[train]]
    ktrain_label = straining_labels[rand_inds[train]]
    ktest = training_data[rand_inds[test]]
    ktrue = straining_labels[rand_inds[test]]

    Mdeg2 = build_kernel_matrix(ktrain, ktrain, rho, deg)
    K_predn = build_kernel_matrix(ktrain, ktest, rho, deg)

    for i in range(len(iterations)):
        it = iterations[i]
        a = np.ones(len(ktrain))

        for j in range(it): #iterations of gradient ascent.
            a = kernel_update(a, Mdeg2[j%len(Mdeg2)].dot(a), ktrai
n_label, j, eps, lam)

        Ka_predn = K_predn.T.dot(a)
        predn = ker_predn(Ka_predn)
        errors[i] += kernel_risk(predn, ktrue)*(1/3)
    break

plt.title('Iterations vs Risk using validation data')
plt.xlabel('Iters')
plt.ylabel('Risk [Cross-Entropy]')
plt.stem(iterations, errors)
```

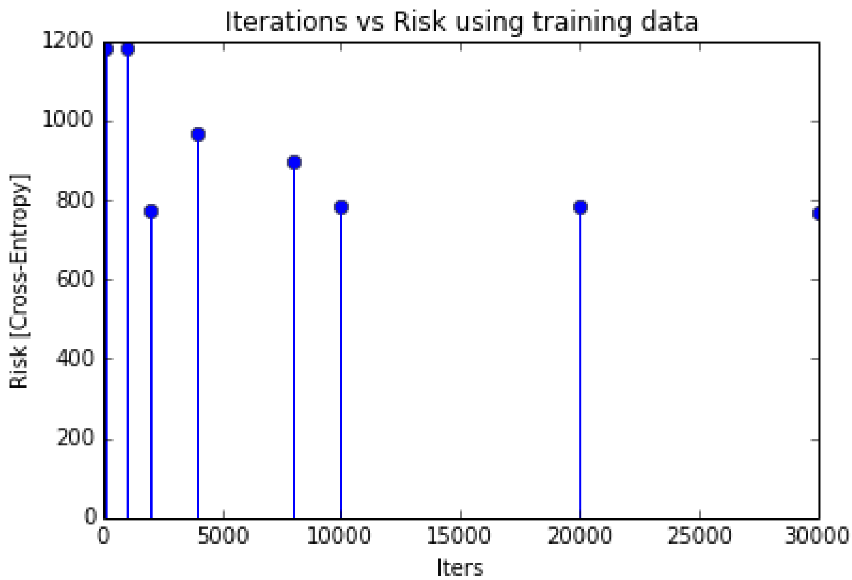
```
Out[1257]: <Container object of 3 artists>
```



In [1258]:

```
### plot risk vs. iterations for single rho value.
rho = 21.8
eps = 1
lam = 1e-3
deg = 1
iterations = [10, 100, 1000, 2000, 4000, 8000, 10000, 20000, 3000
0]
errors = np.zeros(len(iterations))
training_data = np.copy(preprocessed[2])

for train, test in kf3:

    ktrain = training_data[rand_inds[train]]
    ktrain_label = straining_labels[rand_inds[train]]
    ktest = training_data[rand_inds[test]]
    ktrue = straining_labels[rand_inds[test]]

    Mdeg2 = build_kernel_matrix(ktrain, ktrain, rho, deg)
    K_predn = build_kernel_matrix(ktrain, ktrain, rho, deg)

    for i in range(len(iterations)):
        it = iterations[i]
        a = np.ones(len(ktrain))

        for j in range(it): #iterations of gradient ascent.
            a = kernel_update(a, Mdeg2[j%len(Mdeg2)].dot(a), ktrai
n_label, j, eps, lam)

        Ka_predn = K_predn.T.dot(a)
        predn = ker_predn(Ka_predn)
        errors[i] += kernel_risk(predn, ktrain_label)*(1/3)
    break

plt.title('Iterations vs Risk using training data')
plt.xlabel('Iters')
plt.ylabel('Risk [Cross-Entropy]')
plt.stem(iterations, errors)
```

Out[1258]: <Container object of 3 artists>



In [1246]:
```
# Generate Kaggle Labels (score with linear = 0.76905)
```

In [1244]:
```
_stest_data = preproc2(stest_data)
Kaggle_predn = build_kernel_matrix(ktrain, _stest_data, rho=6, deg
=2.5)
spam_labels = ker_predn(Kaggle_predn.T.dot(a))
```

In [1152]:
```
from __future__ import print_function
```

In [1154]:
```
# Save labels
f1 = open('kaggle_spam.csv', 'w+')
print('Id,Category', file = f1)
for i in range(len(spam_labels)):
    print(str(i+1)+","+str(int(spam_labels[i])), file = f1)
```