```
%matplotlib inline
import numpy as np
import scipy as sp
import scipy.io
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
sns.set(color_codes=True)
from math import exp
```

Problem 1

1.1

```
housing_data = scipy.io.loadmat("data/housing_data.mat")
```

```
raw_x_train = pd.DataFrame(housing_data['Xtrain'])
raw_y_train = pd.DataFrame(housing_data['Ytrain'])
raw_x_validate = pd.DataFrame(housing_data['Xvalidate'])
raw_y_validate = pd.DataFrame(housing_data['Yvalidate'])
print(raw_x_train.shape, raw_y_train.shape)
print( raw_x_validate.shape,raw_y_validate.shape)
```

```
(19440, 8) (19440, 1)
(1200, 8) (1200, 1)
```

```
raw_x_train[9] = 1
raw_x_validate[9] = 1
print(raw_x_train.shape, raw_y_train.shape)
print(raw_x_validate.shape, raw_y_validate.shape)
```

```
(19440, 9) (19440, 1)
(1200, 9) (1200, 1)
```

```
raw_x_train.head()
```

	0	1	2	3	4	5	6	7	9
0	2.9241	32	1083	188	471	178	38.04	-121.11	1
1	4.8993	34	3306	555	1398	585	37.99	-122.55	1
2	5.0528	26	2256	360	937	372	38.13	-121.30	1
3	6.0634	15	1834	330	841	309	33.68	-117.78	1
4	1.7500	27	471	132	315	96	33.81	-118.18	1

```
X = raw_x_train.values
y = raw_y_train.values
x_val = raw_x_validate.values
y_val = raw_y_validate.values
w = np.zeros((1,8))
```

```
w = np.dot(np.dot(np.linalg.inv(np.dot(X.T, X)), X.T), y)
print(w.reshape((9,)))
y_hat = np.dot(x_val,w)
```

```
[  4.05879986e+04   1.19561189e+03  -8.50145688e+00   1.18352188e+02
  -3.77900280e+01   4.30562637e+01  -4.21794075e+04  -4.24573474e+04
  -3.56686439e+06]
```

1.2

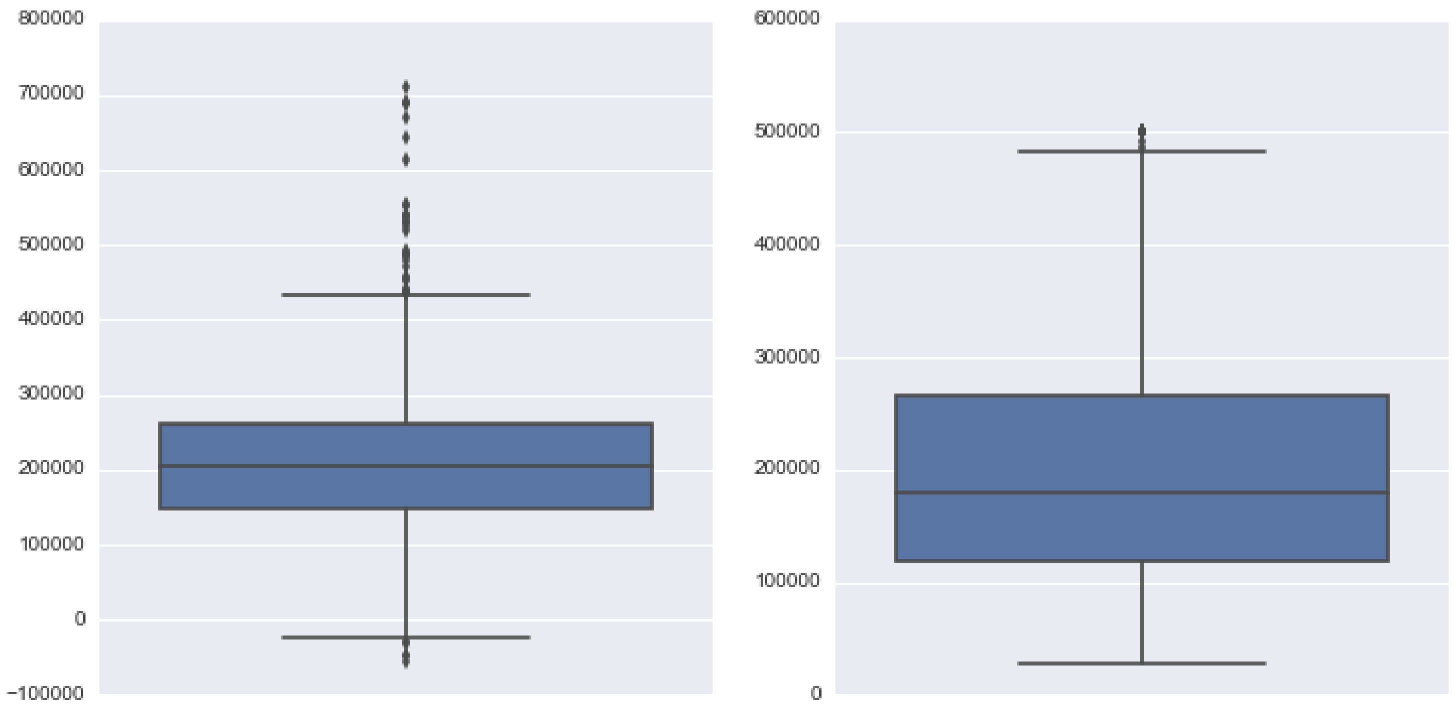
```
rss = np.sum((y_val - y_hat) ** 2)
tss = np.sum((y_val - np.mean(y_val)) ** 2)
r_sqrd = 1 - (rss/tss)
print("RSS: %i" % rss)
print("R^2: %.3f" % r_sqrd)
print("Range of y Hat Values => Min: %i Max: %i" %(min(y_hat), max(y_hat)))
print("Range of True Y Values => Min: %i Max: %i" %(min(y_val), max(y_val)))
```

```
RSS: 5794953797676
R^2: 0.643
Range of y Hat Values => Min: -56562 Max: 710798
Range of True Y Values => Min: 28300 Max: 500001
```

These values do not make sense! How can we have negative household value? Since our model is not robust to outliers, they're likely pulling our entire graph up and messing up the values at the bottom.

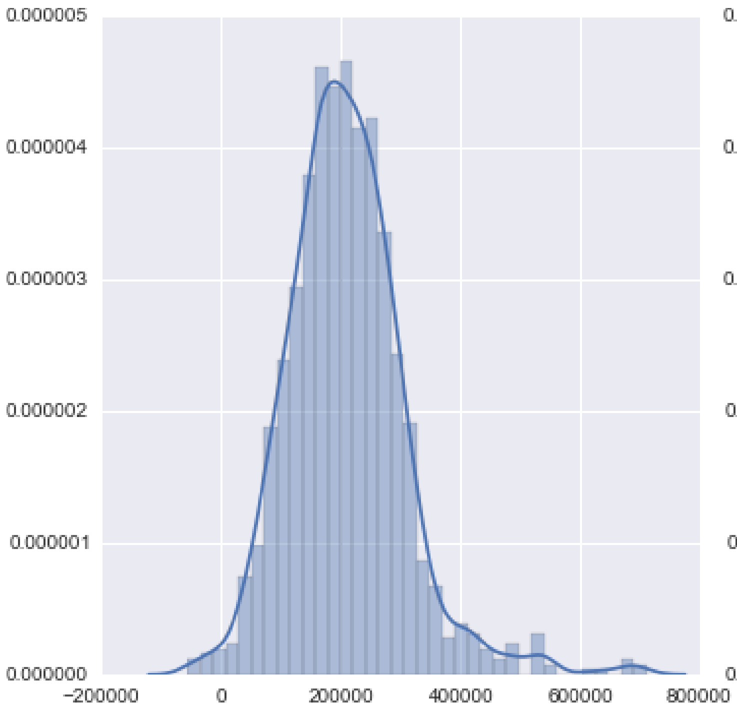
```
f, axes = plt.subplots(1,2, figsize=(12,6), sharex=True)
sns.boxplot(y_hat, orient='v', ax=axes[0])
sns.boxplot(y_val, orient='v', ax=axes[1])
```

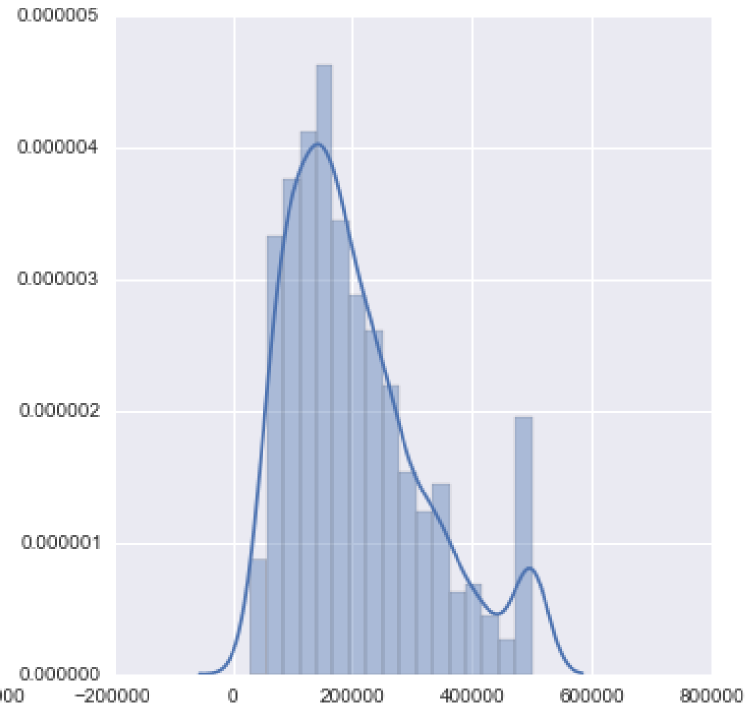
```
<matplotlib.axes._subplots.AxesSubplot at 0x1091c3400>
```



```
f, axes = plt.subplots(1,2, figsize=(12,6), sharex=True)
sns.distplot(y_hat, ax=axes[0])
sns.distplot(y_val, ax=axes[1])
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x109ae47f0>
```

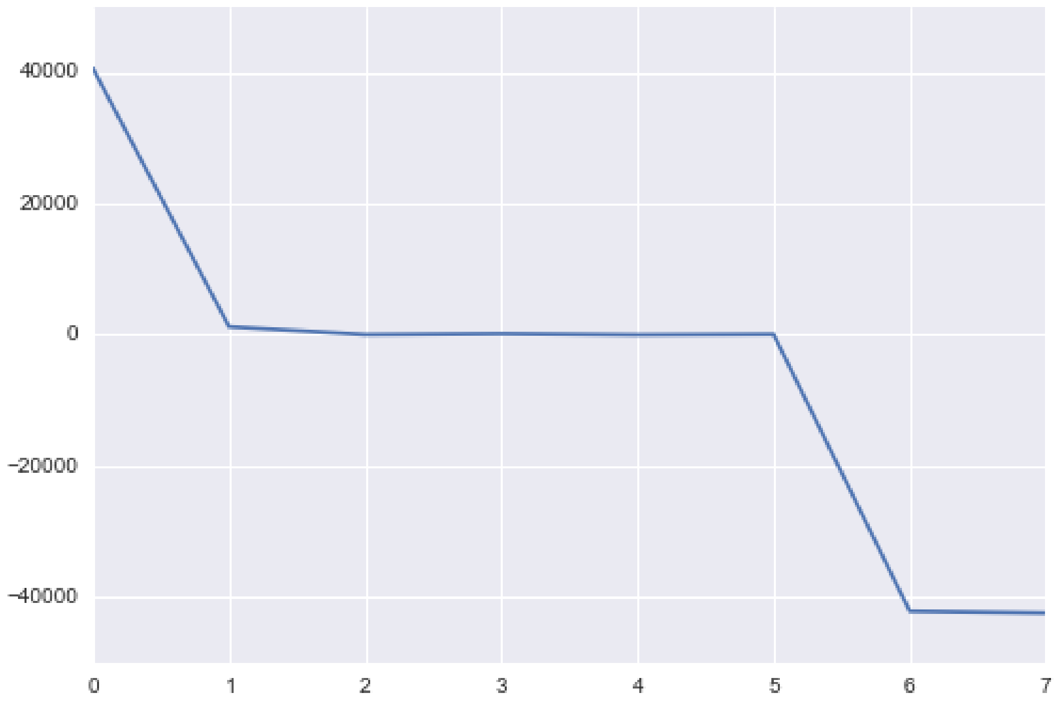




1.3

```
pd.Series(w.reshape((9,))[:8]).plot() # without bias value
```

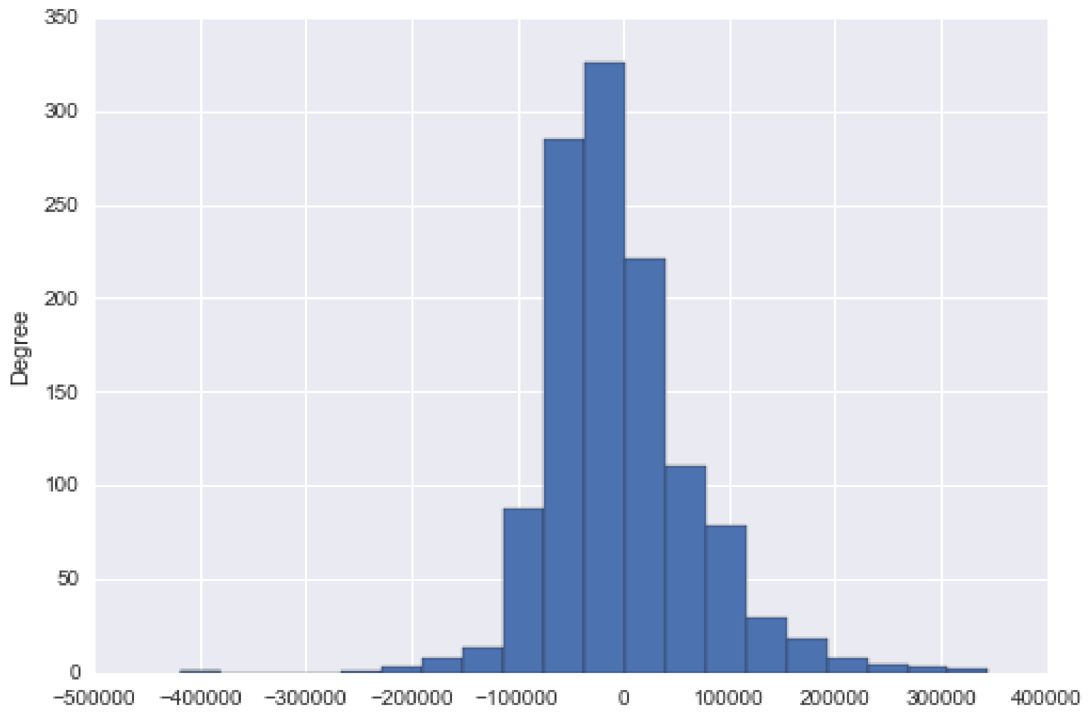
```
<matplotlib.axes._subplots.AxesSubplot at 0x109d6f7b8>
```



1.4

```
pd.Series((y_val - y_hat).reshape((1200,))).plot(kind='hist', bins=20)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x109eba780>
```



This resembles the normal distribution! This is what we want out of our residuals - that our errors are drawn from a normal distribution and there isn't really any discernable patterns beyond the fact that no error is most common and large errors are rare. We can see however that it's a bit skewed.

Problem 2

2.1 and 2.2

```
from IPython.display import Image
Image(filename='2.1.png')
```

$R[w] = \sum^{n} \log(1 + e^{-y^i w^T x^i})$

$h = 1 + e^{-y^i w^T x_i}$

$z = -y^i w^T x_i$

$$\frac{\partial R}{\partial w} = \sum_{i=1}^{n} \frac{\partial R}{\partial h} \cdot \frac{\partial h}{\partial z} \cdot \frac{\partial z}{\partial w_i}$$

$$= \sum \frac{1}{1+e^{-y^i w^T x_i}} \cdot -e^{-z} \cdot y^i w^T x_i$$

$$= \sum \frac{1}{1+e^{-y_i w^T x_i}} \cdot -e^{-y_i w^T x_i} \cdot y^i x_i^T$$

now we clean & put it into matrix notation.

$$\underbrace{\underset{n \times 1}{\frac{1}{1+e^{-diag(y)\cdot X \cdot w}}} \quad \underset{n \times 1}{-e^{-diag(y)\cdot X \cdot w}}}_{\text{we can compose those two}} \qquad \underbrace{\underset{n \times d}{diag(y)\cdot X}}_{\text{and we move this one to the front}}$$

so we can matrix multiply.

$$-Q^T \frac{e^{-Qw}}{1+e^{-Qw}} \cdot \frac{e^{Qw}}{e^{Qw}}$$

$$= -Q^T \cdot \frac{1}{e^{Qw}}$$

2.2

```
Image(filename='2.2.png')
```

$$\frac{\partial}{\partial w_i} -Q^T \underbrace{\frac{e^{-Qw}}{1+e^{-Qw}} \cdot \frac{e^{Qw}}{e^{Qw}}}_{\text{simplifies to}}$$

$$\frac{\partial}{\partial w_i} -Q^T \cdot \frac{1}{1+e^{Qw}}$$

now we chain rule our result

$z = Qw$
$g = 1 + e^z$
$h = \frac{1}{g}$

$$\frac{\partial l}{\partial w_i} = -Q^T \frac{\partial l}{\partial h} \frac{\partial h}{\partial g} \cdot \frac{\partial z}{\partial w_i} \cdot$$

$$= -Q^T \frac{1}{g^2} \cdot e^z \cdot Q$$

$$\boxed{= -Q^T \frac{1}{(1+e^{Qw})^2} \cdot e^{Qw} \cdot Q}$$

2.3

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

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

def descend(X, y, w, alpha):
    Q = np.dot(np.diag(y), X)
    z = np.dot(Q, w)
    top = np.exp(-z)
    bottom = 1 + np.exp(-z)
    combo = (np.dot(-Q.T, (top/bottom)))
    return w - alpha * combo
```

```
w_0 = np.array([-2,1,0])
w_0_pred = log_func(np.dot(X, w_0))
print(w_0)
print(w_0_pred)
print("---- end w_0 ----")
w_1 = descend(X, y, w_0, 1)
w_1_pred = log_func(np.dot(X, w_1))
print(w_1)
print(w_1_pred)
print("---- end w_1 ----")
w_2 = descend(X, y, w_1, 1)
w_2_pred = log_func(np.dot(X, w_2))
print(w_2)
print(w_2_pred)
print("---- end w_2 ----")
```

```
[-2  1  0]
[ 0.95257413  0.73105858  0.73105858  0.26894142]
---- end w_0 ----
[-2.          0.94910188 -0.68363271]
[ 0.89693957  0.54082713  0.56598026  0.15000896]
---- end w_1 ----
[-1.69083609  1.91981257 -0.83738862]
[ 0.99276849  0.96199213  0.74695242  0.35242149]
---- end w_2 ----
```

Problem 3

Transformations

```python
def standardize_cols(X):
    zscore = lambda x: (x.mean() - x)/x.std()
    return pd.DataFrame(X).apply(zscore).values

def binarize_cols(X):
    temp = X.copy()
    temp[temp > 0] = 1
    temp[temp <= 0] = 0
    return temp
```

```python
spam_data = scipy.io.loadmat("data/spam.mat")
Xraw = pd.DataFrame(spam_data['Xtrain'])
y = spam_data['Ytrain'].reshape((3450,))
Xtestraw = pd.DataFrame(spam_data['Xtest'])
print(Xraw.shape, y.shape, Xtestraw.shape)

stan_X = pd.DataFrame(standardize_cols(Xraw.values))
log_X = pd.DataFrame(np.log(Xraw.values + 0.1))
bin_X = pd.DataFrame(binarize_cols(Xraw.values))
stan_X[57] = 1
log_X[57] = 1
bin_X[57] = 1
Xraw[57] = 1
X = Xraw.values
stan_X = stan_X.values
log_X = log_X.values
bin_X = bin_X.values
print(X.shape, stan_X.shape, log_X.shape, bin_X.shape)
```

```
(3450, 57) (3450,) (1151, 57)
(3450, 58) (3450, 58) (3450, 58) (3450, 58)
```

3.1

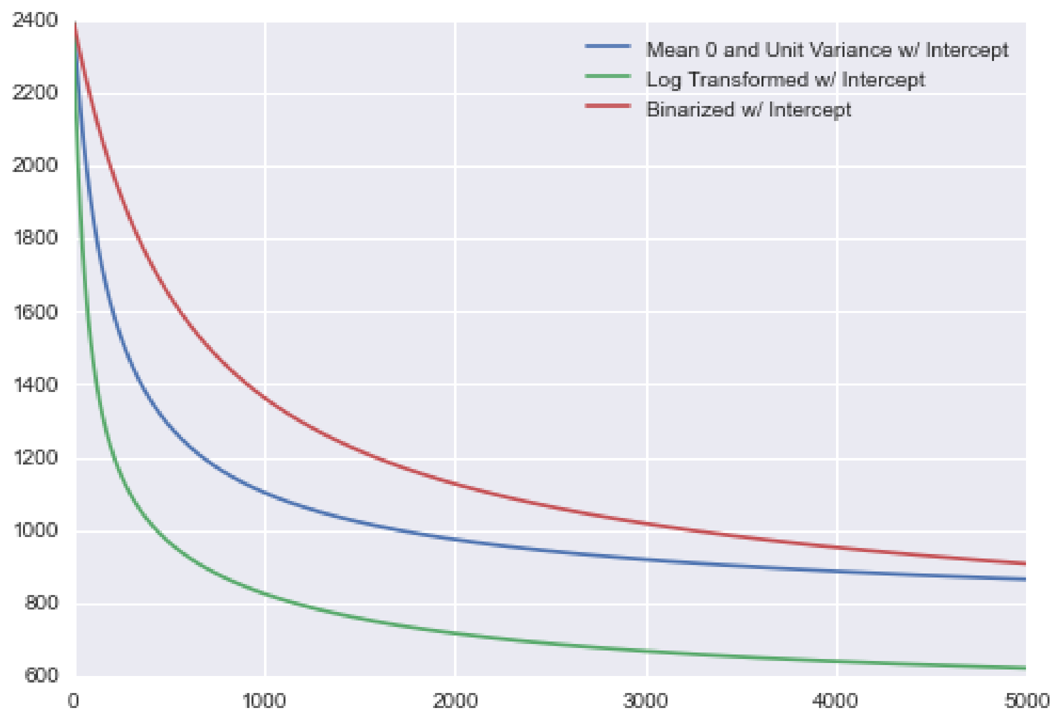
```python
def batch_gradient_descent(Q, w, alpha):
    z = np.dot(Q, w)
    top = np.exp(-z)
    bottom = 1 + np.exp(-z)
    combo = (np.dot(-Q.T, (top/bottom)))
    risk = np.sum(np.log(1 + np.exp(-z)))
    return w - alpha * combo / len(X), risk

def batch_wrapper(X, y, alpha, num_iters):
    w = np.zeros(len(X[0, :]))
    Q = np.dot(np.diag(y), X)
    risks = []
    for count in range(num_iters):
        new_w, risk = batch_gradient_descent(Q, w, alpha)
        risks.append(risk)
        w = new_w
    return w, pd.DataFrame(risks)
```

```python
alpha = 0.005
num_iters = 5000
```

```python
w2, risks2 = batch_wrapper(stan_X, y, alpha, num_iters)
w3, risks3 = batch_wrapper(log_X, y, alpha, num_iters)
w4, risks4 = batch_wrapper(bin_X, y, alpha, num_iters)
```

```python
all_risks = pd.concat([risks2, risks3, risks4], axis=1)
all_risks.columns = ['Mean 0 and Unit Variance w/ Intercept', 'Log Transformed w/ Intercept'
, 'Binarized w/ Intercept']
all_risks.plot()
# # if you want each plot just call this
# risks2.plot()
# risks3.plot()
# risks4.plot()
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x10a20a3c8>
```



3.2

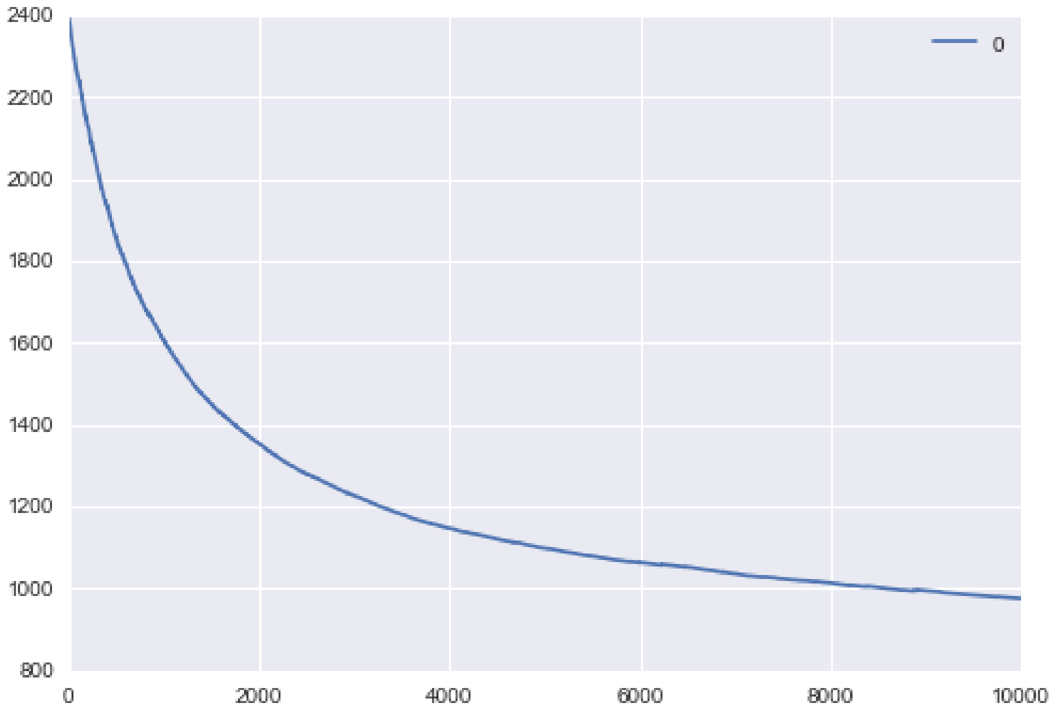
This is the same update as before except we only use one point to derive the gradient, not the entire dataset as we were doing before. You can see this in the code below where now *Q* is lowercase, signifying that we're doing it based on one row.

```
from random import shuffle

def shuffle_data(X, y):
    shuff = list(range(len(X)))
    shuffle(shuff)
    X = X[shuff]
    y = y[shuff]
    Q = np.dot(np.diag(y), X)
    return Q, X, y

def stochastic_gradient_descent(Q, w, alpha):
    z = np.dot(Q, w)
    top = np.exp(-z)
    bottom = 1 + top
    combo = (np.dot(-Q.T, (top/bottom)))
    return w - alpha * combo

def stochastic_wrapper(X, y, num_iters, alpha):
    w = np.zeros(len(X[0,:]))
    risks = []
    Q, X, y = shuffle_data(X, y)
    nullr = False
    if not alpha:
        nullr = True
    for count in range(num_iters):
        if nullr:
            alpha = 1 / (count + 1)
        choice = np.random.randint(0,len(X))
        new_w = stochastic_gradient_descent(Q[choice], w, alpha)
        w = new_w
        risks.append(np.sum(np.log(1 + np.exp(-Q.dot(w)))))
    return w, pd.DataFrame(risks)
```

```
learning_rate = 0.001
num_iters = 10000
```

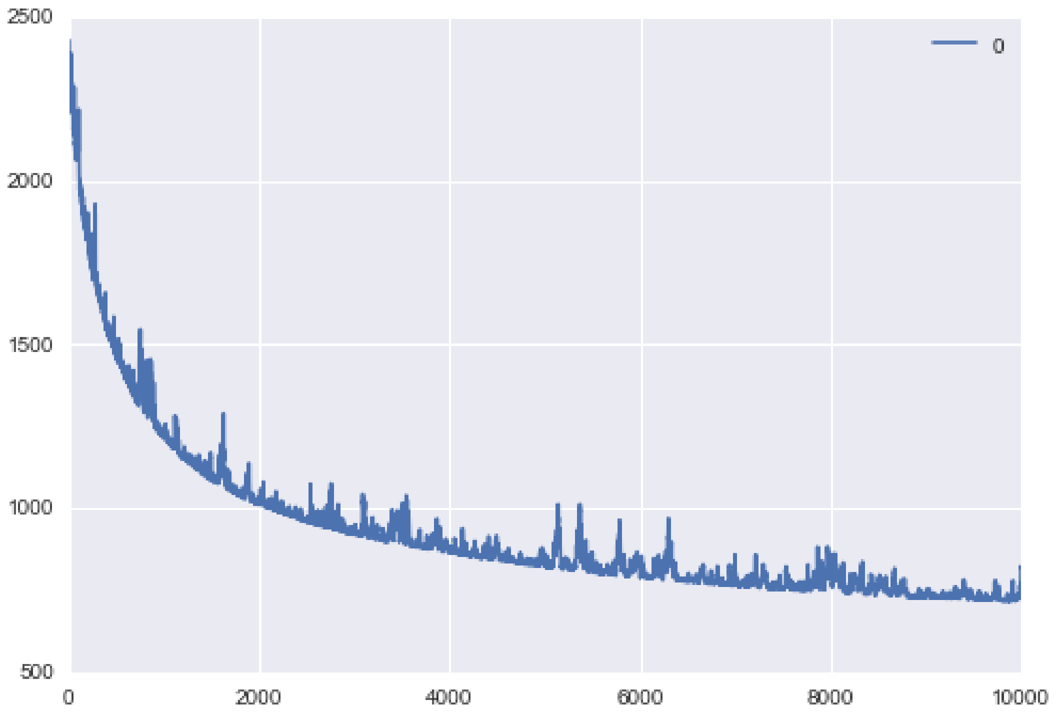
```
w2, risks2 = stochastic_wrapper(stan_X, y, num_iters, learning_rate)
risks2.plot()
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x10a20a7f0>
```



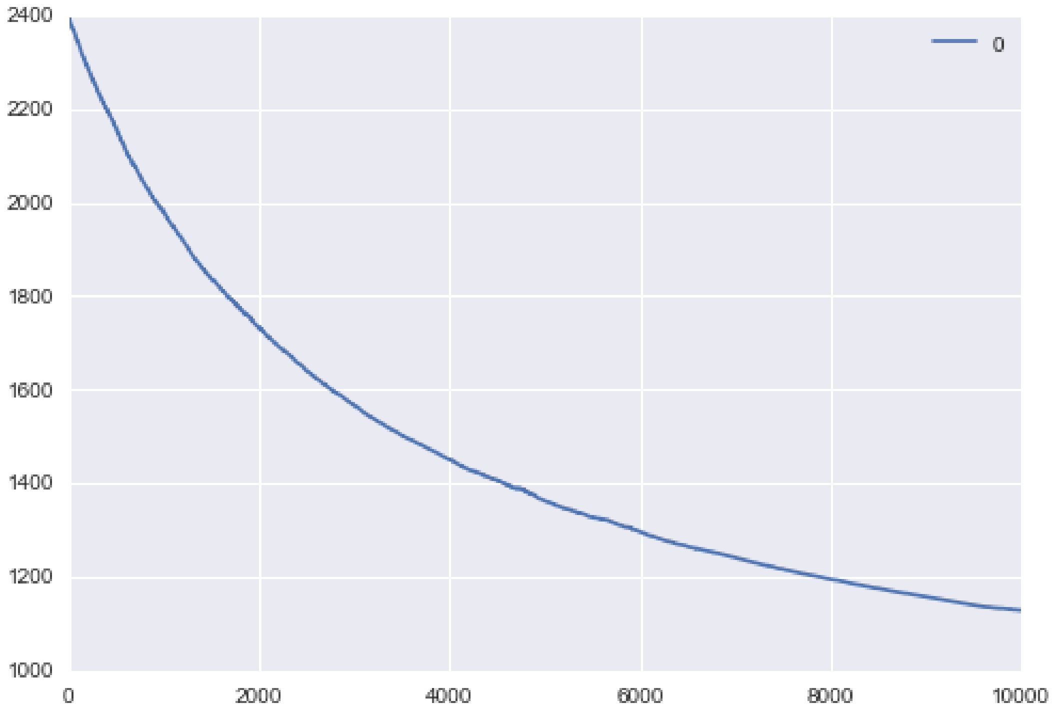
```
w3, risks3 = stochastic_wrapper(log_X, y, num_iters, learning_rate)
risks3.plot()
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x10a54dc50>
```



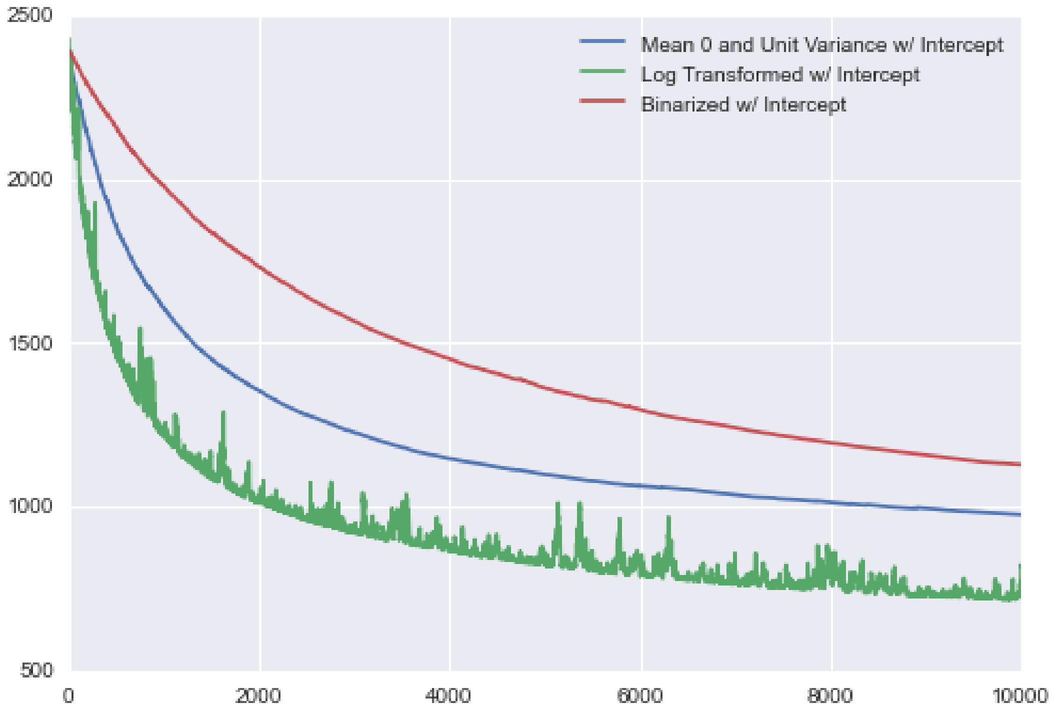
```
w4, risks4 = stochastic_wrapper(bin_X, y, num_iters, learning_rate)
risks4.plot()
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x111bd1358>
```



```
all_risks = pd.concat([risks2,risks3,risks4], axis=1)
all_risks.columns = ['Mean 0 and Unit Variance w/ Intercept', 'Log Transformed w/ Intercept'
, 'Binarized w/ Intercept']
all_risks.plot()
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x111e0da90>
```



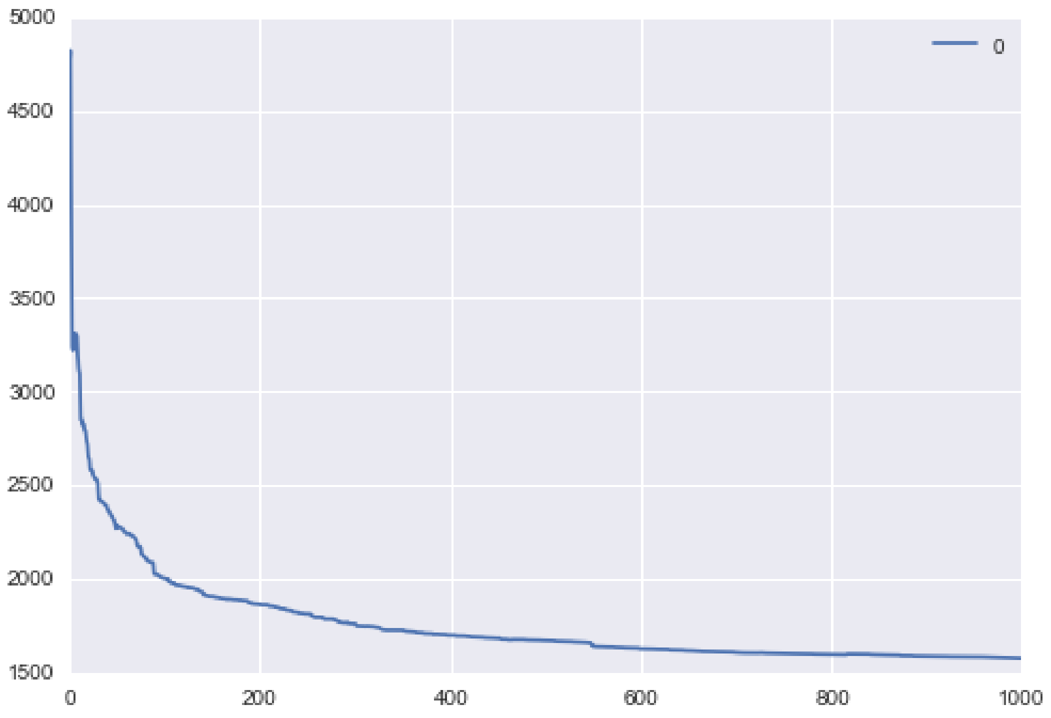
The plots are different from before because they're a lot choppier. This makes sense as we're not always choosing the optimal gradient to go down.

3.3

```
learning_rate = None
num_iters = 1000
```

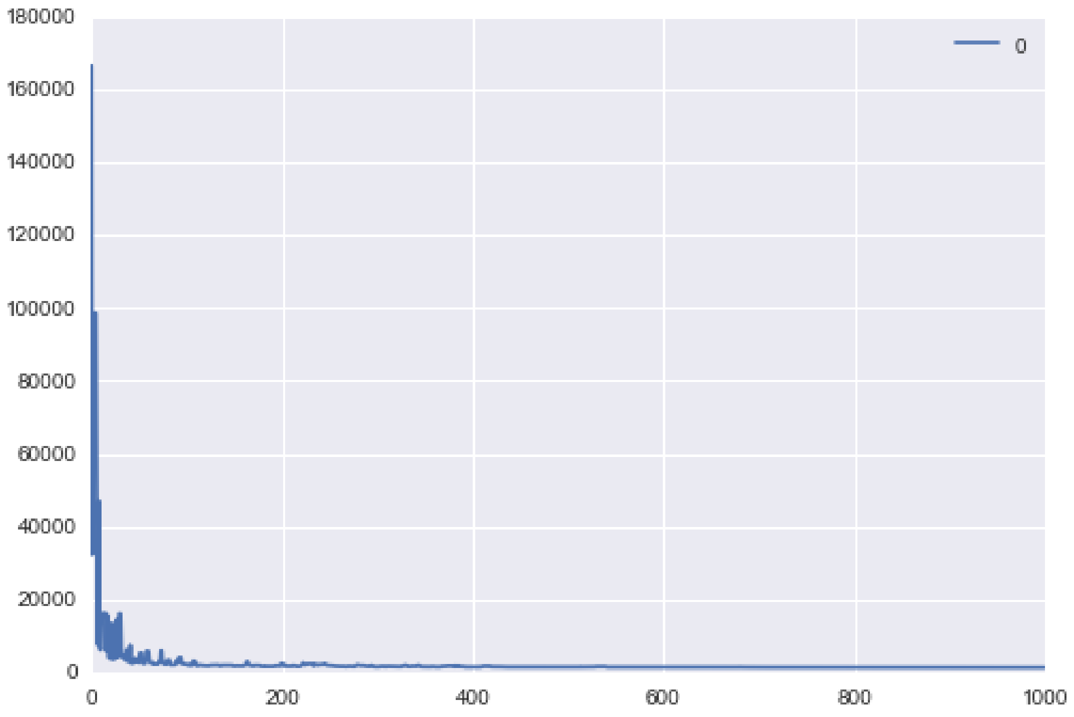
```
w2, risks2 = stochastic_wrapper(stan_X, y, num_iters, learning_rate)
risks2.plot()
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x10a7c7080>
```



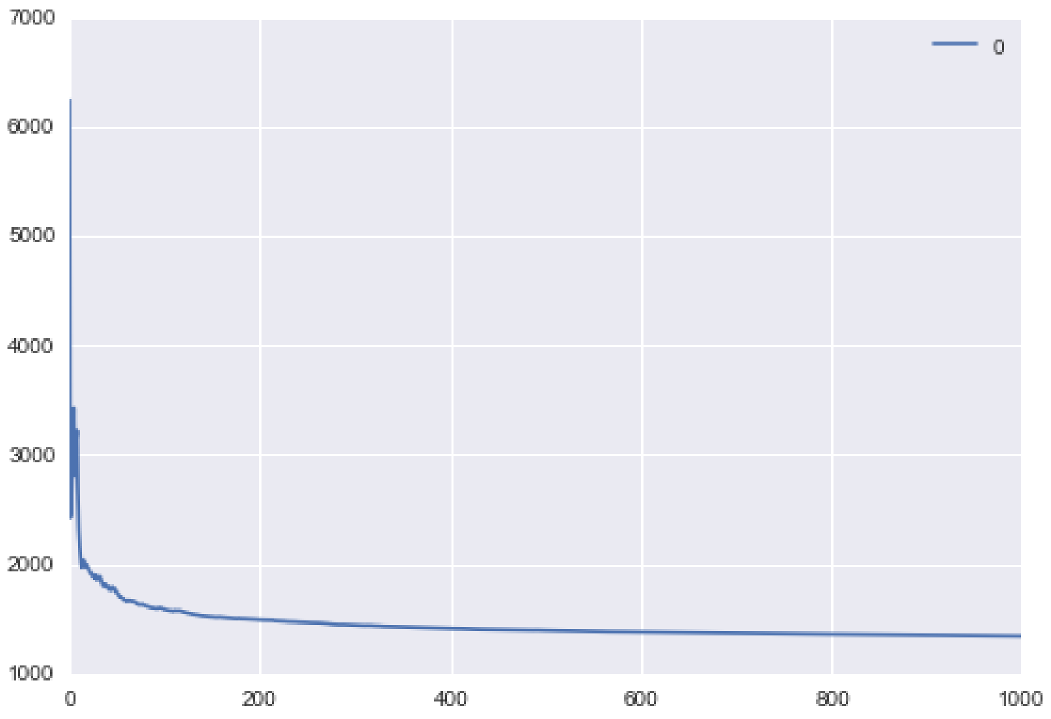
```
w3, risks3 = stochastic_wrapper(log_X, y, num_iters, learning_rate)
risks3.plot()
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x11210ad68>
```



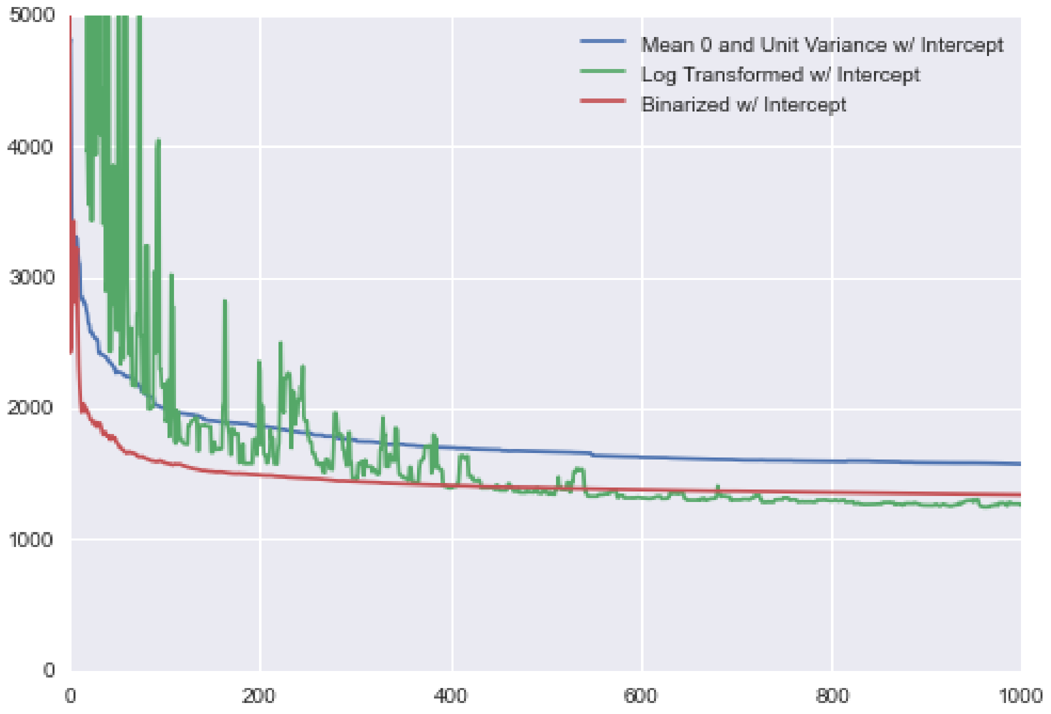
```
w4, risks4 = stochastic_wrapper(bin_X, y, num_iters, learning_rate)
risks4.plot()
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x111fcba58>
```



```
all_risks = pd.concat([risks2,risks3,risks4], axis=1)
all_risks.columns = ['Mean 0 and Unit Variance w/ Intercept', 'Log Transformed w/ Intercept'
, 'Binarized w/ Intercept']
all_risks.plot(ylim=(0, 5000))
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x111f296a0>
```



The automatic iteration is much less stable, it also seems to get stuck in a local minimum.

3.4.a

```
Image(filename='3.4.a.png')
```

$$\frac{\partial L}{\partial \alpha_i} = \frac{\partial \log(1+e^{-y^i f(x^i)})}{\partial \alpha_i}$$

$f(x) = \sum_{i=1}^{n} \alpha_i k(x^i, x)$

$k(x^i, x) = (x^T x^i + 1)^2$

$$= \partial \log(1+e^{-y^i f(x^i)})$$

Chain rule... w/ info from last problem

$$= \frac{1}{1+e^{-y^i f(x^i)}} \cdot e^{-y^i f(x^i)} \cdot \frac{\partial -y f(x^i)}{\partial \alpha_1}$$

$\Downarrow$ simplifies to

$$= \frac{1}{1+e^{-y^i f(x)}} \cdot \frac{\partial -y^i f(x^i)}{\partial \alpha_i}$$

$$\frac{\partial L}{\partial \alpha_i} = (-S(-z)) \cdot y^i (x^T x^i + 1)^2$$

leading to the update

$$\alpha_i \leftarrow \alpha_i + \eta S(-z) \cdot y^i (x^T x^i + 1)^2$$

The two updates would be the same when $x^T x^i = 0$, when the examples are orthogonal. (x $ x_i)

3.4.b

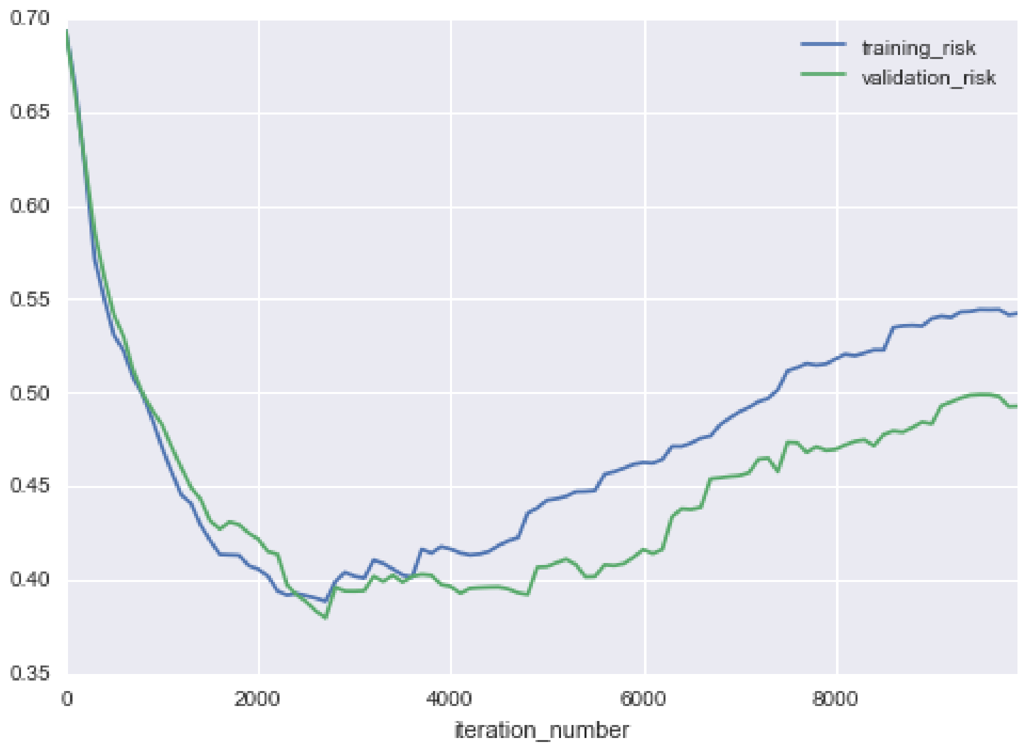
```
def qkrsgd_test(vX, vy, tX, alpha):
    val_risk = []
    for count, (xi, yi) in enumerate(zip(vX, vy)):
        fx = np.dot(alpha, np.power((np.dot(tX, xi) + 1), 2))
        z = yi * fx
        loss = np.log(1 + np.exp(z * -1))
        val_risk.append(loss)
    return np.sum(val_risk) / len(vX)

def qkrsgd_wrapper(X, y, gamma, num_iters, lr=None):
    split = int(len(X) * 2 / 3)
```

```python
    tX, ty = X[:split], y[:split]
    vX, vy = X[split:], y[split:]
    print("Creating training and test sets of %i and %i" % (len(tX), len(vX)))
    alpha = np.zeros(split)
    print("Creating alphas of length: ", len(alpha))
    np.random.seed(200)
    iters = np.random.choice(split, num_iters)
    print("Creating iteration choices of length ", len(iters))
    nullLR = False
    if not lr:
        nullLR = True
        print("auto learning rate")
    else:
        print("Learning rate of ", lr)
    risks = []
    iteration_numbers = []
    val_risks = []
    epoch_risk = []

    kernel_gram = np.power((tX.dot(tX.T) + 1), 2)
    kgwy = kernel_gram.dot(np.diag(ty))

    for count, choice in enumerate(iters):
        if nullLR:
            lr = 0.0001 / (count/5 + 1)
        ai = alpha[choice]
        yi = ty[choice]
        update_prep = alpha.dot(kgwy)
        zi = update_prep[choice]
#         if zi > 200:
#             print("WARNING: LARGE Z VALUE => NOW CLAMPING")
#             zi = 200
        s = lambda z: 1 / (1 + np.exp(-1 * zi))
        update = ai - (gamma * ai) + (lr * s(zi) * ty[choice])
        alpha[choice] = update
        alpha[choice+1:] -= (alpha[choice+1:] * gamma)
        alpha[:choice] -= (alpha[:choice] * gamma)


        if count % 100 == 0:
            risk = np.sum(np.log(1 + np.exp(update_prep * -1))) / len(tX)
            iteration_numbers.append(count)
            risks.append(risk)
            epoch_risk = [] # resetting the epoch risk
            val_risks.append(qkrsgd_test(vX, vy, tX,  alpha))
    return alpha, pd.DataFrame({
            "iteration_number": iteration_numbers,
            "training_risk":risks,
            "validation_risk": val_risks
        })
```

```
gamma, iters, lr = 10e-5, 10000, 0.00005
```

```
w, risks = qkrsgd_wrapper(stan_X, y, gamma, iters, lr)
risks.plot(x='iteration_number')
```

```
Creating training and test sets of 2300 and 1150
Creating alphas of length:  2300
Creating iteration choices of length  10000
Learning rate of  5e-05






<matplotlib.axes._subplots.AxesSubplot at 0x109cb6710>
```

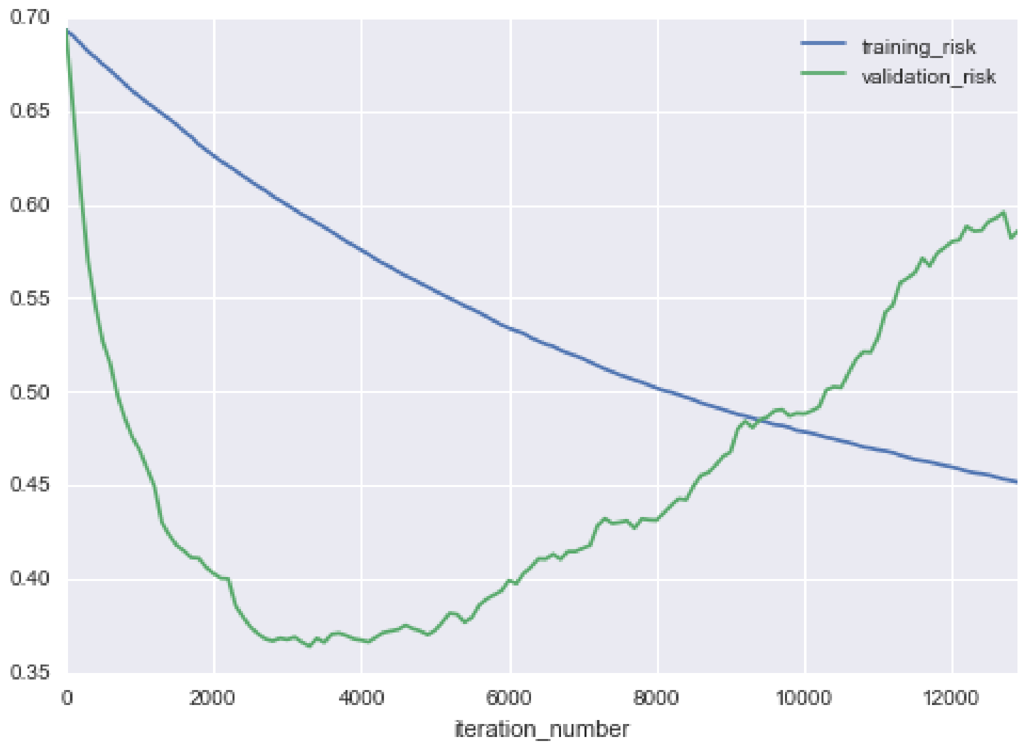


3.4.c

```
def lkrsgd_test(vX, vy, tX, alpha):
    val_risk = []
    for count, (xi, yi) in enumerate(zip(vX, vy)):
        fx = np.dot(alpha, (np.dot(xi, tX.T) + 1))
        z = yi * fx
        loss = np.log(1 + np.exp(z * -1))
        val_risk.append(loss)
    return np.sum(val_risk) / len(vX)

def lkrsgd_wrapper(X, y, gamma, num_iters, lr=None):
    split = int(len(X) * 2 / 3)
    tX, ty = X[:split], y[:split]
    vX, vy = X[split:], y[split:]
    print("Creating training and test sets of %i and %i" % (len(tX), len(vX)))
    alpha = np.zeros(split)
    print("Creating alphas of length: ", len(alpha))
    np.random.seed(200)
    iters = np.random.choice(split, num_iters)
    print("Creating iteration choices of length ", len(iters))
    nullLR = False
    if not lr:
        nullLR = True
        print("auto learning rate")
    else:
        print("Learning rate of ", lr)
    risks = []
    iteration_numbers = []
    val_risks = []
    epoch_risk = []

    kernel_gram = (tX.dot(tX.T) + 1)
    kgwy = kernel_gram.dot(np.diag(ty))

    for count, choice in enumerate(iters):
        if nullLR:
            lr = 0.0001 / (count/5 + 1)
        ai = alpha[choice]
        yi = ty[choice]
        update_prep = alpha.dot(kgwy)
        zi = update_prep[choice]
        if zi > 200:
            print("WARNING: LARGE Z VALUE => NOW CLAMPING")
            zi = 200
        s = lambda z: 1 / (1 + np.exp(-zi))
        update = ai - (gamma * ai) + (lr * s(zi) * ty[choice])
        alpha[choice] = update
        alpha[choice+1:] -= (alpha[choice+1:] * gamma)
        alpha[:choice] -= (alpha[:choice] * gamma)


        if count % 100 == 0:
            risk = np.sum(np.log(1 + np.exp(update_prep * -1))) / len(tX)
            iteration_numbers.append(count)
            risks.append(risk)
            epoch_risk = [] # resetting the epoch risk
            val_risks.append(qkrsgd_test(vX, vy, tX, alpha))
    return alpha, pd.DataFrame({
            "iteration_number": iteration_numbers,
            "training_risk":risks,
            "validation_risk": val_risks
        })
```

```
gamma, iters, lr = 10e-5, 13000, 0.000075
w2, risks2 = lkrsgd_wrapper(stan_X, y, gamma, iters, lr)
risks2.plot(x='iteration_number')
```

```
Creating training and test sets of 2300 and 1150
Creating alphas of length:  2300
Creating iteration choices of length  13000
Learning rate of  7.5e-05






<matplotlib.axes._subplots.AxesSubplot at 0x11b23d358>
```



We can see that the linear kernel, has a much lesser capacity for information that the quadratic kernel. Therefore it descends more slowly (requiring more iterations) and also has a smaller validation risk. However we can see towards the end that the linear kernel will overfit as well. In fact the more data we look at, the worse it seems to perform.

Both of them see to be overfitting the data a lot, I don't think that our sample size justifies the use of this kernel.

If we wanted to try and reduce overfitting, increasing γ would be a way that we could do that.

Problem 4

One of the issues could be that it's going to be hard for the kernel function to take advantage of this new information. One reason for the observed problem is that 00:00 and 23:59 are nearly the same time but are represented completely differently. One would be a large number while the other one would be a smaller one. Time is not continuous in the way that he is using it. Therefore he should try another representation - perhaps treating it as a categorical variable for hour and minute. That might allow for cleaner use of interactions that the kernel would generate.