# hw4

March 12, 2015

# 1 Homework 4

- Name: Austin Chen
- SID: 23826762
- Repro: Open up hw4.ipynb in IPython Notebook.

#### 1.0.1 (1)

$$l(\beta) = \lambda||\beta||_2^2 - \sum_{i=1}^{n}[y_i log\mu_i + (1-y_i)log(1-\mu_i)]$$

$$\nabla_\beta l(\beta) = \frac{d}{d\beta}[\lambda\beta^T\beta - \sum_{i=1}^{n}[y_i log\mu_i + (1-y_i)log(1-\mu_i)]]$$

$$\nabla_\beta l(\beta) = 2\lambda\beta - \sum_{i=1}^{n}\frac{d}{d\beta}[y_i log\mu_i + (1-y_i)log(1-\mu_i)]$$

$$\nabla_\beta l(\beta) = 2\lambda\beta - \sum_{i=1}^{n}x_i[y_i - \mu_i]$$

Lemma:

$$\frac{d}{d\beta}[y_i log\mu_i + (1-y_i)log(1-\mu_i)]$$

$$= y_i\frac{d}{d\beta}log\mu_i + (1-y_i)\frac{d}{d\beta}log(1-\mu_i)$$

$$= y_i(1/\mu_i)\frac{d}{d\beta}\mu_i + (1-y_i)/(1-\mu_i)\frac{d}{d\beta}(1-\mu_i)$$

Let $k = e^{-\beta^T x_i}$.
Then, $\mu_i = 1/(1+k)$, and $1-\mu_i = k/(1+k)$.
Also, $\frac{d}{d\beta}k = -x_i k$.

$$= y_i(1+k)\frac{d}{d\beta}1/(1+k) + (1-y_i)(1+k)/k * \frac{d}{d\beta}k/(1+k)$$

$$= y_i(1+k)(x_i k)/(1+k)^2 + (1-y_i)(1+k)/k * [(1+k)(-x_i k) - (k)(-x_i k)/(1+k)^2]$$

$$= y_i(x_i k)/(1+k) + (1-y_i)1/k * (-x_i k)/(1+k)]$$

$$= y_i(x_i k)\mu_i + (1-y_i) * (-x_i)\mu_i]$$

$$= \mu_i x_i[y_i k - (1-y_i)]$$

$$= \mu_i x_i[y_i(k+1) - 1]$$

$$= x_i[y_i - \mu_i]$$

### 1.0.2 (2)

$$2\lambda I + \sum_{i=1}^{n}(u_i)(1-u_i)x_i x_i^T$$

### 1.0.3 (3)

$$B^{(t+1)} = B^{(t)} - H_\beta^{-1} * \nabla_\beta l(\beta)$$

### 1.0.4 (4)

In [1]:
```
import numpy as np
import scipy.io
%matplotlib inline
import matplotlib.pyplot as plt
```

In [82]:
```
def mu(B, X):
    mu = np.empty((len(X), 1))
    for i in range(len(X)):
        mu[i] = mui(B, X, i)
    return mu

def mui(B, X, i):
    mu = 1 / (1 + np.exp(-B.T.dot(X[i])))
    mu = max(mu, .000000000001)
    mu = min(mu, .999999999999)
    return mu

def diag(mu):
    # Return the diagonal matrix, ith entry = u_i(1-u_i)
    return np.diagflat(mu * (1 - mu))

def gradient(B, X, y, l):
    u = mu(B, X)
    return 2 * l * B - X.T.dot(y - u)

def hessian(B, X, y, l):
    u = mu(B, X)
    return 2 * l * np.eye(len(X.T)) + X.T.dot(diag(u)).dot(X)

def loss(B, X, y, l):
    u = mu(B, X)
    return (l * B.T.dot(B) - y.T.dot(np.log(u)) - (1 - y).T.dot(np.log(1 - u)))[0][0]
```

In [102]:
```
X = np.array([
[0, 3, 1],
[1, 3, 1],
[0, 1, 1],
[1, 1, 1]])

B = np.array([[-2, 1, 0]]).T
y = np.array([[1, 1, 0, 0]]).T
l = 0.07

def step(B, X, y, l):
```

```
        return B - np.linalg.inv(hessian(B, X, y, l)).dot(gradient(B, X, y, l))

    print("u0:", mu(B, X))
    B = step(B, X, y, l)
    print("B1:", B)

    print("u1:", mu(B, X))
    B = step(B, X, y, l)
    print("B2:", B)
```

```
u0: [[ 0.95257413]
 [ 0.73105858]
 [ 0.73105858]
 [ 0.26894142]]
B1: [[-0.38676399]
 [ 1.40431761]
 [-2.28417115]]
u1: [[ 0.87311451]
 [ 0.82375785]
 [ 0.29320813]
 [ 0.21983683]]
B2: [[-0.51222668]
 [ 1.45272677]
 [-2.16271799]]
```

## 1.1 Problem 2

### 1.1.1 (1)

```
In [2]: # Load the data
        data = np.loadtxt(open('data/YearPredictionMSD.txt','rb'), delimiter=',')
```

```
In [3]: # Separate the training data
        X = data[:463715, 1:]
        y = data[:463715, :1].astype(int)

        # Introduce extra row for constant term
        X = np.hstack((X, np.ones((len(X), 1))))
```

```
In [4]: # Solve for B
        B = np.linalg.solve(np.dot(X.T, X), np.dot(X.T, y))
```

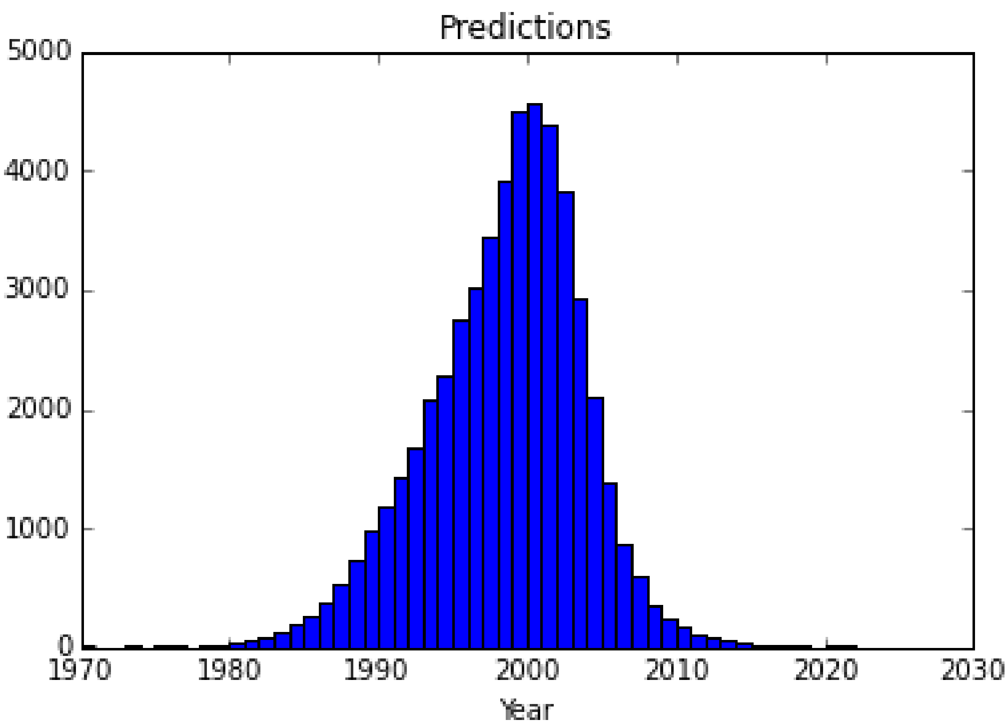
### 1.1.2 (2)

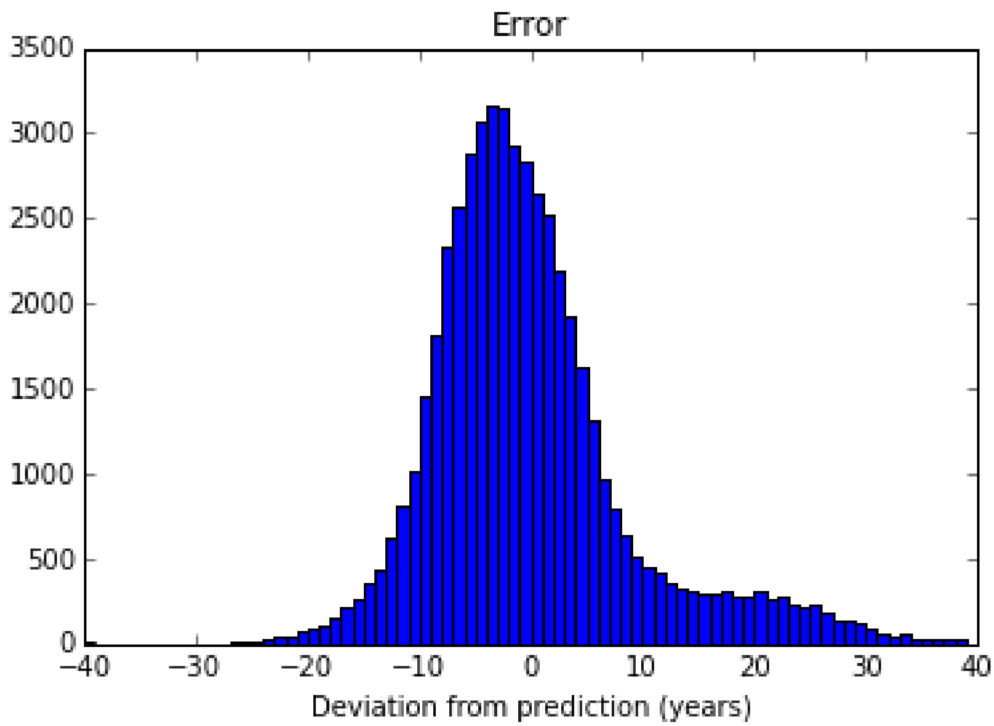
On the test set, I get an RSS of around 4,670,000, and predictions from 1980 to 2015 with a few outliers (shown below). This mostly makes sense, as the data provided spans 1922 to 2011 with a peak in the 2000s.

```
In [21]: # Separate the test data
         Xt = data[463715:, 1:]
         yt = data[463715:, :1].astype(int)

         # Account for constant term
         Xt = np.hstack((Xt, np.ones((len(Xt), 1))))

         # Calculate residual sum of squares
         RSS = np.linalg.norm(np.dot(Xt, B) - yt) ** 2
```

```python
print("RSS is", RSS)

# Some interesting visualizations
predictions = np.dot(Xt, B)
plt.hist(predictions, bins=np.arange(1970, 2025, 1))
plt.title("Predictions")
plt.xlabel("Year")
plt.show()

plt.hist(predictions - yt, bins= np.arange(-40, 40, 1))
plt.title("Error")
plt.xlabel("Deviation from prediction (years)")
plt.show()
```

```
RSS is 4669580.17951
```

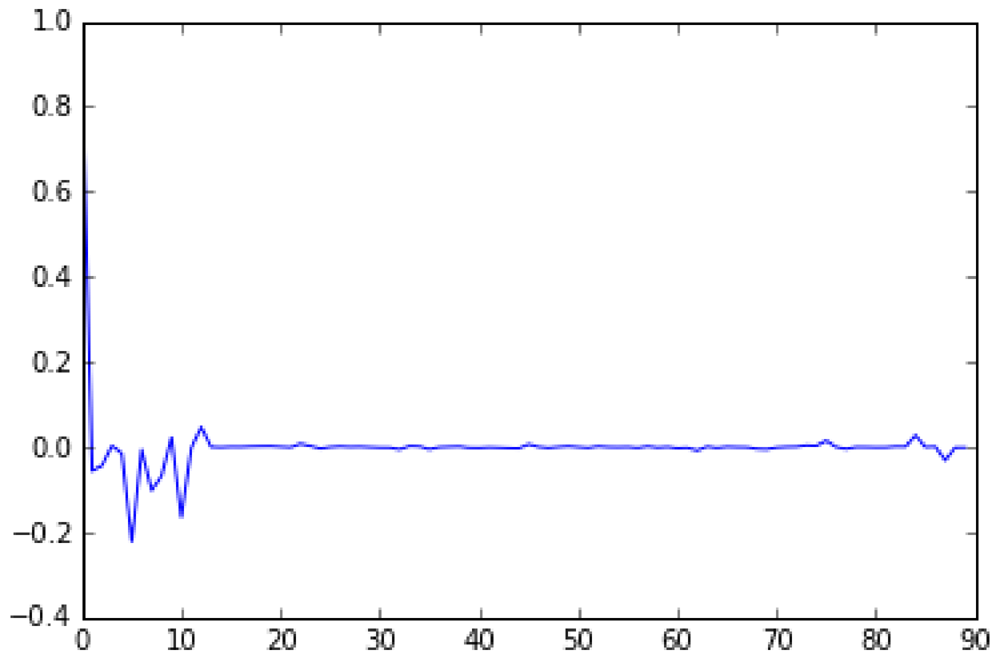





#### 1.1.3 (3)

```
In [120]: # Plot B, excluding B0
          plt.plot(B[:-1])

          print("B0 is", B[-1])
```

```
B0 is [ 1951.1221816]
```




#### 1.1.4 (4)

Linear regression is a reasonable model for this problem, since the value we're trying to calculate (years) is a continuous metric. As evidenced by the histogram of deviations, the prediction error is mostly within +/- 10 years.

### 1.2 Problem 3

In [114]:
```
# Load the data
mat = scipy.io.loadmat('data/spam.mat')
xtrain, ytrain, xtest = mat['Xtrain'], mat['ytrain'], mat['Xtest']
```

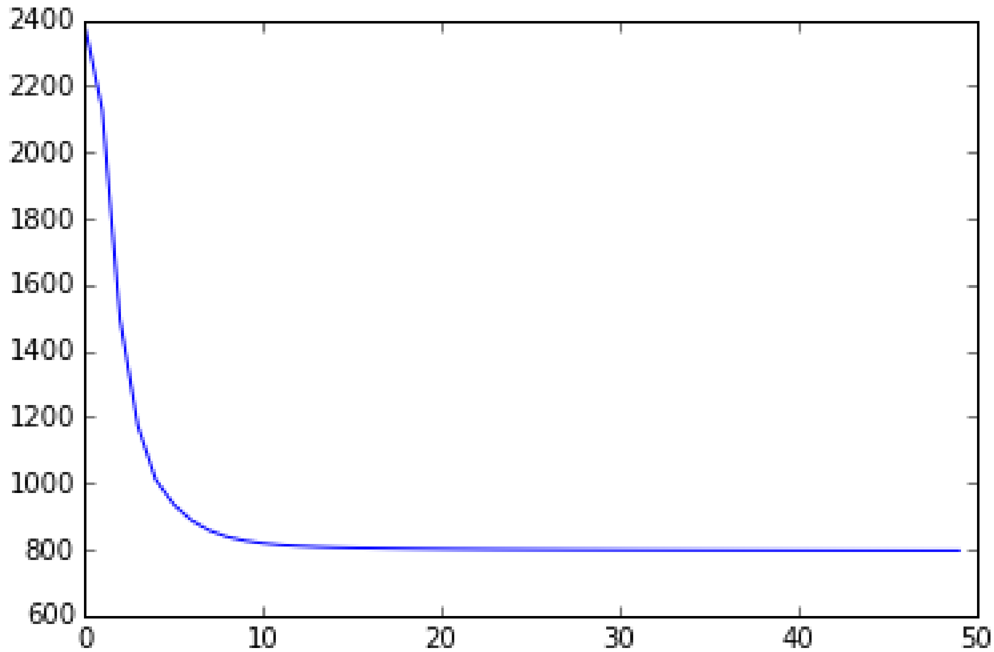
In [115]:
```
from sklearn import preprocessing

# i) Standardize the columns to 0 mean, unit variance
# X1 = preprocessing.scale(xtrain)
X1 = (xtrain - np.mean(xtrain, axis=0)) / np.std(xtrain, axis=0)

# ii) Transform the features with log
X2 = np.log(xtrain + 0.1);

# iii) Binarize the features
# X3 = preprocessing.binarize(xtrain)
X3 = np.array([[1 if x > 0 else 0 for x in col] for col in xtrain]).astype(float)
```

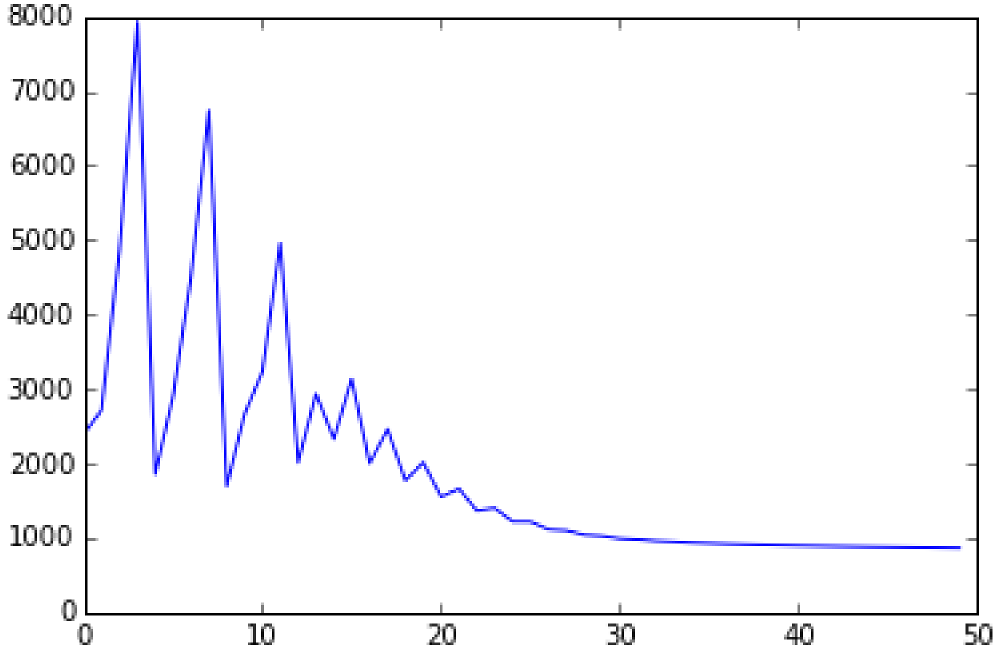
#### 1.2.1 (1)

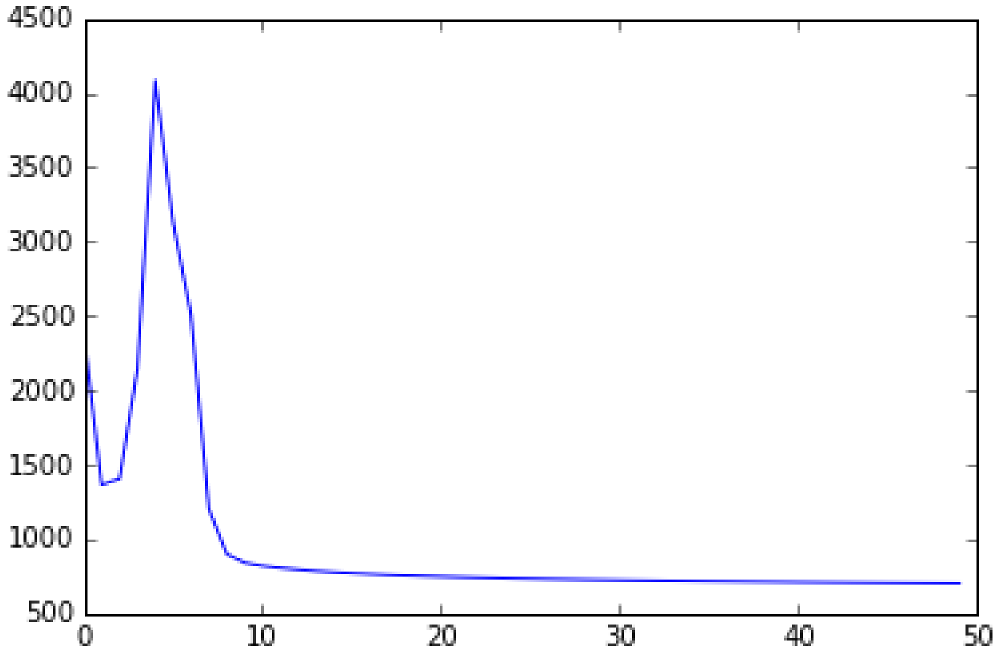
$$B^{(t+1)} = B^{(t)} + \eta \nabla_B l(B^{(t)}$$

Note: I picked very aggressive step sizes to reduce the number of iterations to converge, resulting in some variation in the loss before said convergence.

In [116]:
```
def iterated_batch(X, y, l, step, iterations):
    B = np.zeros((len(X.T), 1)) # B0
    losses = []
    for _ in range(iterations):
        losses.append(loss(B, X, ytrain, l))
#         print(losses[-1])
        B = B - step * gradient(B, X, y, l)
    plt.plot(losses)
    plt.show()

iterated_batch(X1, ytrain, 1, 0.003, 50)
iterated_batch(X2, ytrain, 1, 0.00002, 50)
iterated_batch(X3, ytrain, 1, 0.001, 50)
```






#### 1.2.2 (2)

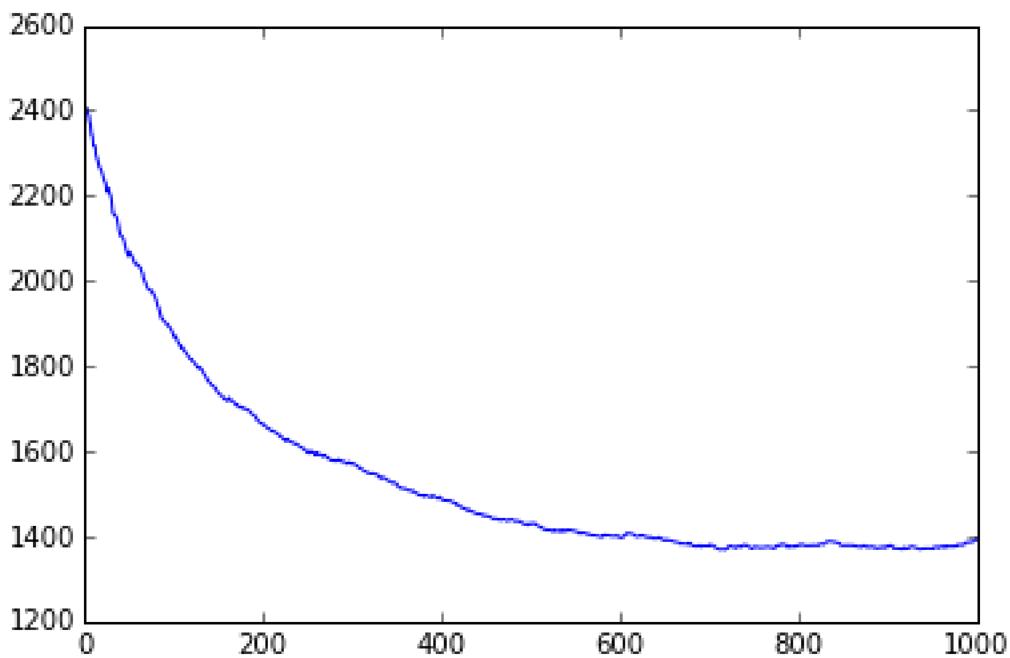
$$B^{(t+1)} = B^{(t)} + \eta(y_{it} - \mu_{it}(B^{(t)}))$$

These curves have a lot more variance than the curves in part 1, which is to be expected since they're randomized approximations designed to run faster.

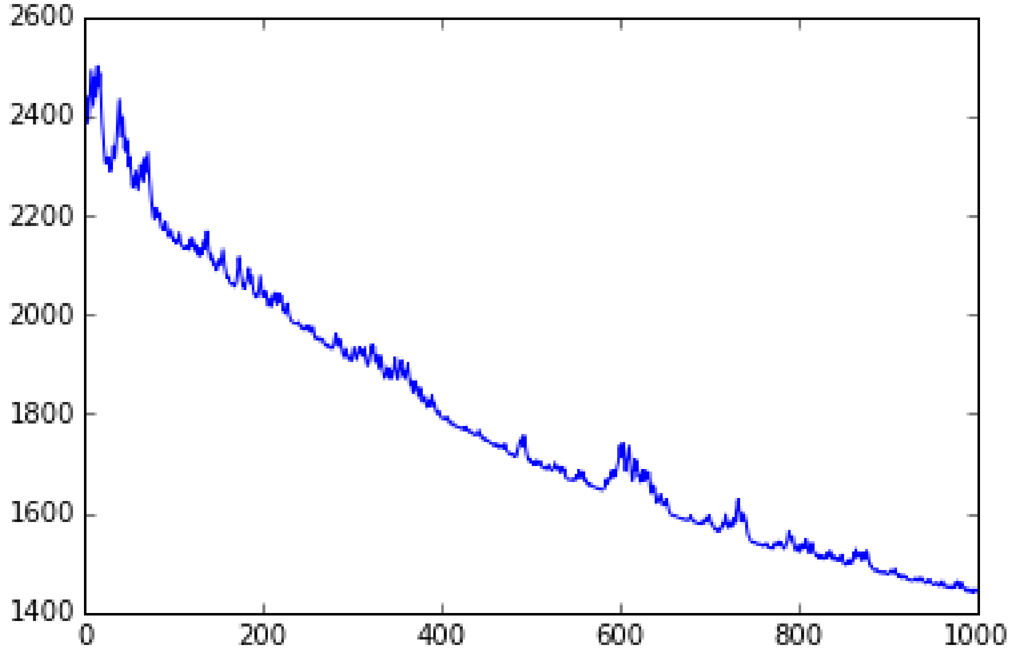
Again, I'm using aggressive step sizes.

```
In [147]: def iterated_stochastic(X, y, l, step, iterations):
              B = np.zeros((len(X.T), 1)) # B0
              losses = []
              for _ in range(iterations):
                  losses.append(loss(B, X, ytrain, l))
          #         print(losses[-1])
                  i = np.random.randint(len(B))
                  gi = step * (2 * l * B[i] - (y[i] - mui(B, X, i))) * X[i]
                  gi = np.reshape(gi, (len(B), -1))
                  B = B - gi
              plt.plot(losses)
              plt.show()
```

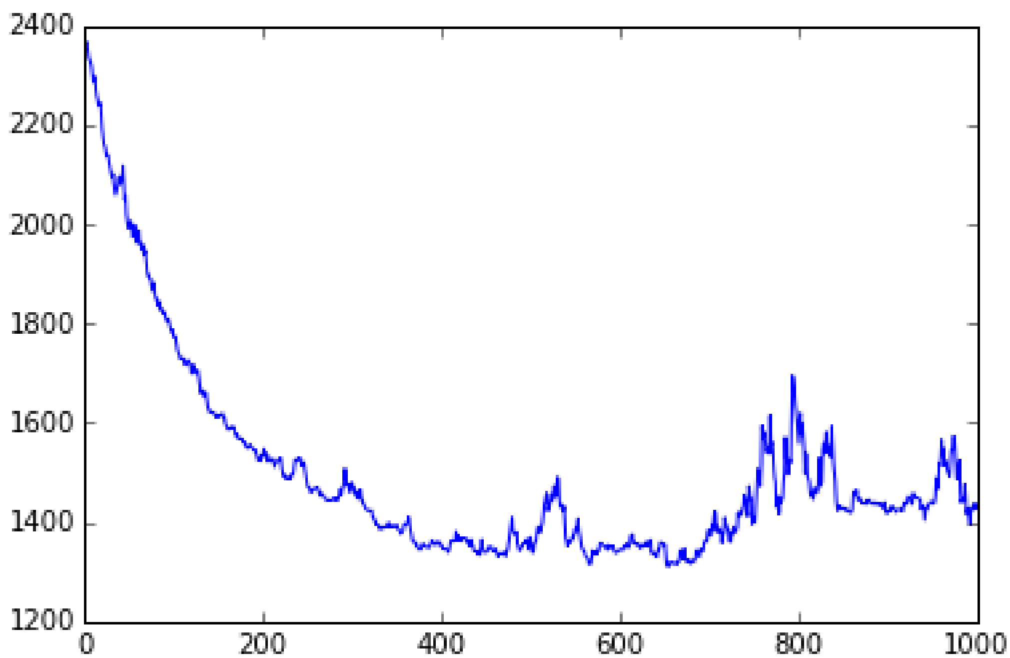
```
In [105]: iterated_stochastic(X1, ytrain, 1, 0.005, 1000)
```




```
In [67]: iterated_stochastic(X2, ytrain, 1, 0.0005, 1000)
```




In [145]: iterated_stochastic(X3, ytrain, 1, 0.02, 1000)



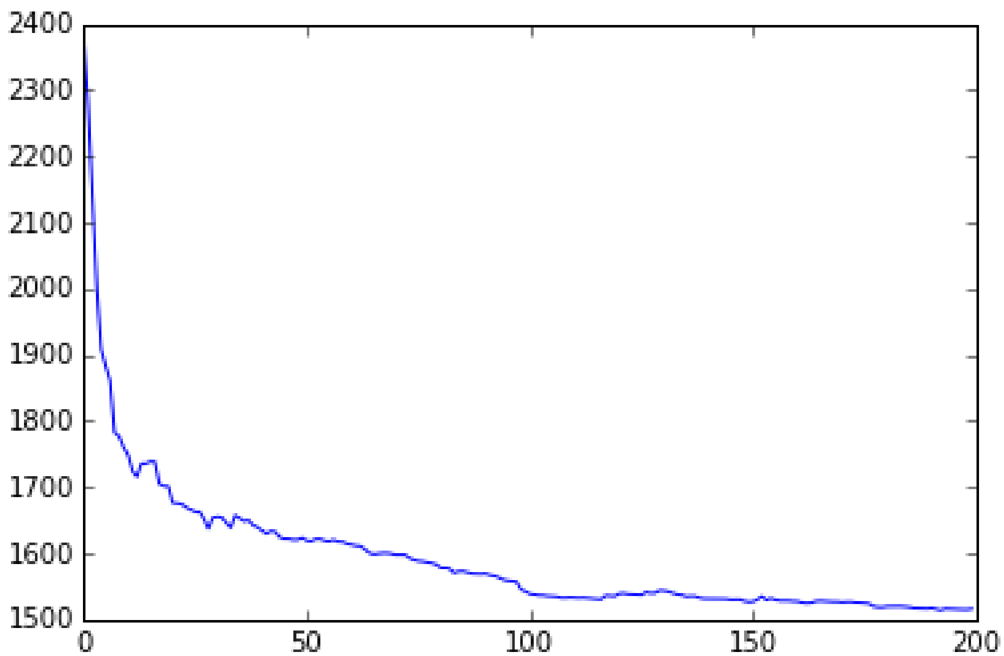
### 1.2.3 (3)

Yes, this strategy is better, since it allows for very large step sizes, and thus improvements to B in the beginning, then slowly decreases the step sizes as convergence occurs. The net result is that significantly fewer iterations are required for a good result.

```
In [127]: def iterated_stochastic2(X, y, l, step, iterations):
              B = np.zeros((len(X.T), 1)) # B0
              losses = []
              for t in range(iterations):
                  losses.append(loss(B, X, ytrain, l))
          #         print(losses[-1])
                  i = np.random.randint(len(B))
                  gi = step / (t + 1) * (2 * l * B[i] - (y[i] - mui(B, X, i))) * X[i]
                  gi = np.reshape(gi, (len(B), -1))
                  B = B - gi
              plt.plot(losses)
              plt.show()
```

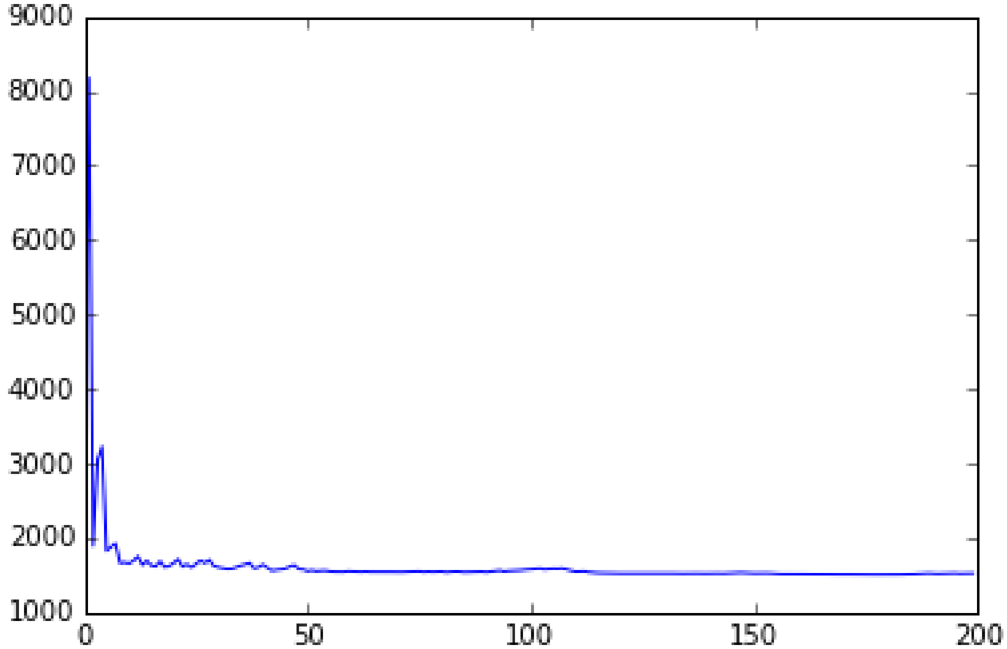
```
In [134]: iterated_stochastic2(X1, ytrain, 1, 0.55, 200)
```

```
2391.35777293
2285.33686019
```




```
In [153]: iterated_stochastic2(X2, ytrain, 1, 0.05, 200)
```

```
2391.35777293
8178.9984967
```




```
In [142]: iterated_stochastic2(X3, ytrain, 1, 2, 200)
```

```
2391.35777293
7382.66225213
```