CS289a - HW3

John Semerdjian

October 6, 2015

Problem 1: Linear Regression

In this problem we will try to predict the median home value in a given Census area by using linear regression. The data is in `housing_data.mat`, and it comes from http://lib.stat.cmu.edu/datasets/ (`houses.zip`). There are only 8 features for each data point; you can read about the features in `housing_data_source.txt`.

1.1 Implement a linear regression model with least squares. Include your code in the submission. You should add a constant term to the training data (e.g. add another dimension to each data point, with the value of 1). This is same as adding the bias term to linear regression (see discussion 4 question 1).

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

housing  = scipy.io.loadmat('data/housing_data.mat')
h_Xtrain = np.asmatrix(housing['Xtrain'])
h_Ytrain = np.asmatrix(housing['Ytrain'])
h_Xtest  = np.asmatrix(housing['Xvalidate'])
h_Ytest  = np.asmatrix(housing['Yvalidate'])

def add_ones(X):
    m = X.shape[0]
    ones = np.ones((m, 1))
    X1 = np.hstack((ones, X))
    return X1

def least_squares(X, y):
    X1 = add_ones(X)
    X1_T = X1.T
    w = np.linalg.inv(X1_T*X1)*X1_T*y
    return w

def rss_calc(y, y_hat):
    rss = np.sum(np.square(y-y_hat))
    return rss

def predict(X, w):
    X1 = add_ones(X)
    y_hat = X1*w
    return y_hat
```

```
w = least_squares(h_Xtrain, h_Ytrain)
```

1.2 Test your trained model on the validation set. What is the residual sum of squares (RSS) on the validation set? What is the range of predicted values? Do they make sense?

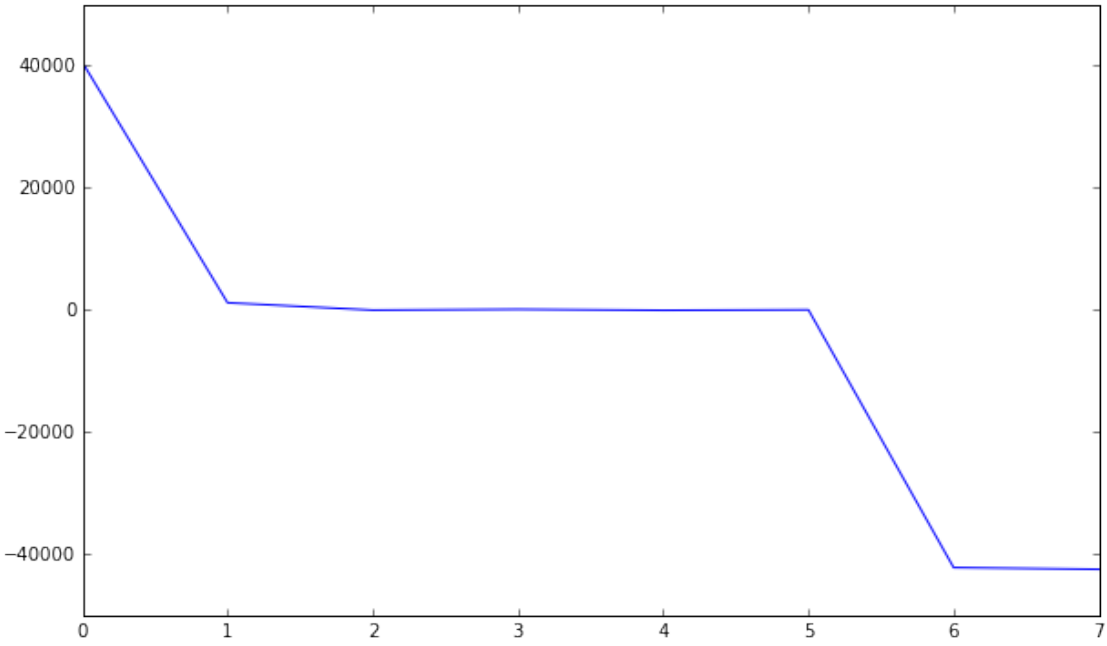
Answer: It seems strange for a house to have a negative value.

```
In [2]: h_Ytest_hat = predict(h_Xtest, w)
        print("RSS: {0:.1f}".format(rss_calc(h_Ytest, h_Ytest_hat)))
        print("Min: {0:.1f}".format(np.min(h_Ytest_hat)))
        print("Max: {0:.1f}".format(np.max(h_Ytest_hat)))
```

```
RSS: 5794953797672.5
Min: -56562.8
Max: 710798.8
```

1.3 Plot the regression coefficient w (plot the value of each coefficient against the index of the coefficient). Be sure to exclude the coefficient corresponding to the constant offset you added earlier.

```
In [3]: plt.figure(figsize=(10, 6))
        plt.plot(np.arange(8), w[1:])
        plt.show()
```

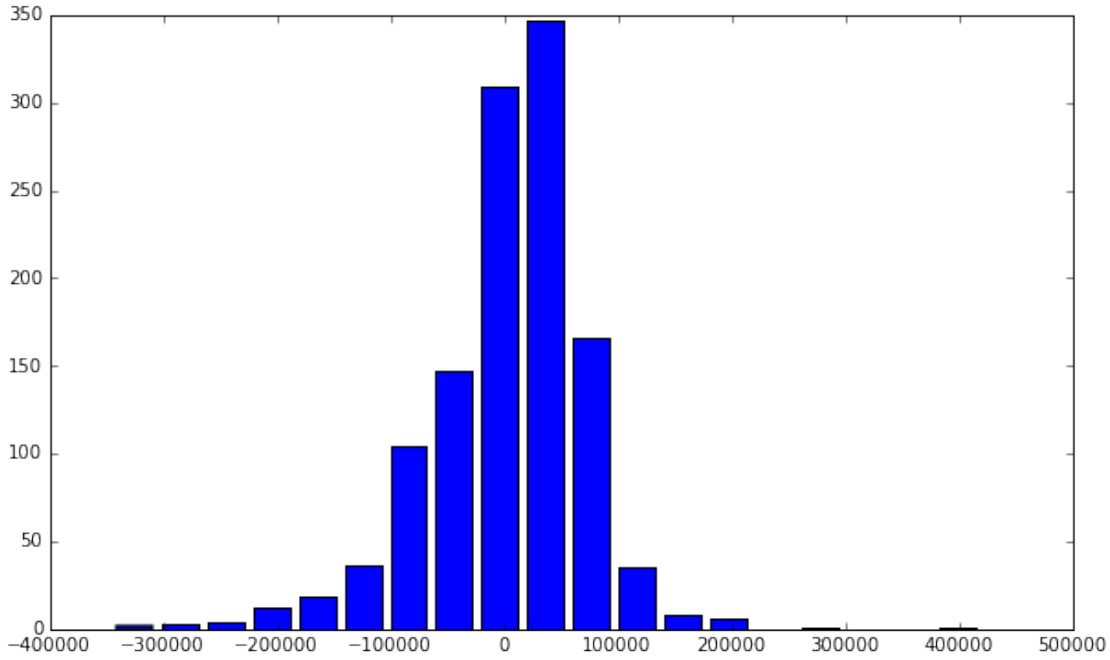


1.4 Plot a histogram of the residuals $(f(x) - y)$. What distribution does this resemble?

NOTE: You may not use any library routine for linear regression or least squares solving. You may use any other linear algebra routines.

Answer: The distribution lookes guassian/normal.

```
In [4]: residuals = h_Ytest_hat - h_Ytest
        bins = np.linspace(np.min(residuals), np.max(residuals), num=20)
        plt.figure(figsize=(10, 6))
        plt.hist(residuals, bins, histtype='bar', rwidth=0.8)
        plt.show()
```



Problem 2: Logistic Regression

Let $\{(x^{(i)}, y^{(i)})^n\}_{i=1}$ be a training set, where $x^i \in \mathbb{R}^d$ and $y^{(i)} \in \{-1, 1\}$. Recall that the loss function for logistic regression is the cross-entropy. Therefore our risk is:

$$R[\mathbf{w}] = \sum_{i=1}^{n} \log(1 + e^{-z^{(i)}})$$

where

$$f(x) = \mathbf{w}^T \mathbf{x}, z^{(i)} = y^{(i)} f(x^{(i)})$$

In this problem you will minimize the cross-entropy risk (also known as the negative log likelihood of $\mathbf{w}$) on a small training set. We have four data points in $\mathbb{R}^2$, two of class 1, and two of class -1. Here is the data (and you may want to draw this on paper to see what it looks like):

$$X = \begin{bmatrix} 0 & 3 \\ 1 & 3 \\ 0 & 1 \\ 1 & 1 \end{bmatrix}, y = \begin{bmatrix} 1 \\ 1 \\ -1 \\ -1 \end{bmatrix}$$

Here, X is the training data matrix; each row $\mathbf{x}^{(i)}$ of X is a data point. Notice that the data cannot be separated by a boundary that goes through the origin. To account for this, you should append 1 to the $\mathbf{x}^{(i)}$ vectors and fit a three-dimensional $\mathbf{w}$ vector that includes an offset term.

2.1 Derive the gradient of the cross-entropy risk with respect to w. Show your work. Your answer should be in matrix-vector expression. Do NOT write your answer in terms of the individual components of the gradient. For notation, you may let the $diag(\mathbf{v})$ denote the square matrix with components of vector v on the diagonal. You may let $Q = diag(\mathbf{y})X$, and you may assume e^M is a matrix, where the i, jth entry of e^M is $e^{M_{ij}}$.

Note: Writing these updates as a matrix operations instead of for-loops makes for cleaner code and allows you to take advantage of highly optimized linear algebra routines.

$$R[\mathbf{w}] = \sum_{i=1}^{n} \log(1 + e^{-z^{(i)}})$$

Using the chain rule:
$$\begin{aligned}\frac{\partial R}{\partial w} &= \frac{\partial log(h)}{\partial h}\frac{1+e^{-z}}{\partial z}\frac{yw^T x^{(i)}}{\partial w} \\ &= (\frac{1}{1+e^{-yw^T x^{(i)}}})(-e^{-yw^T x^{(i)}})(yx^{(i)^T}) \\ &= (\frac{1}{1+e^{-Qw}})(-e^{-Qw})(Q^T) \\ &= -Q^T\frac{e^{-Qw}}{1+e^{-Qw}}\end{aligned}$$

```
In [5]: y = np.array([[ 1],
                      [ 1],
                      [-1],
                      [-1]])

        X = np.matrix([[0, 3, 1],
                       [1, 3, 1],
                       [0, 1, 1],
                       [1, 1, 1]])

        diag_y = np.zeros((y.shape[0], y.shape[0]))
        np.fill_diagonal(diag_y, val=y)

        Q = diag_y*X
```

```
In [6]: def gradient(Q, w):
            z = Q*w
            return -Q.T * (np.exp(-z)/(1 + np.exp(-z)))

        def logistic(z):
            return 1/(1 + np.exp(-z))

        def cross_entropy(Q, w):
            return np.sum(np.log(1+np.exp(-Q*w)))

        def gd(Q, w, n=1, step=1):
            rs = []
            ws = []
            for i in range(n):
                risk = cross_entropy(Q, w)
                w_new = w - step*gradient(Q, w)
                w = w_new
                rs.append(risk)
                ws.append(w)
            return rs, ws
```

2.2 In general, to verify that the function we minimize is convex we will want to show that the Hessian is positive semi-definite. For now, just derive the Hessian of the risk. Show your work; your answer should be a (somewhat complicated) matrix-vector expression.

Take derivative of the gradient: $-Q^T \frac{e^{-Qw}}{1+e^{-Qw}}$

$$H = -Q^T \left[\frac{1}{1+e^{Qw}}\right]^2 \frac{\partial(1+e^{-z})}{\partial z} \frac{\partial Qw}{\partial w_i}$$
$$= -Q^T \left[\frac{1}{1+e^{Qw}}\right]^2 e^{Qw} Q$$

2.3 We will now perform gradient descent for a few iterations. Set the learning rate (η) as 1. We are given that $

$$\mathbf{w}^{(0)} = \begin{bmatrix} -2 & 1 & 0 \end{bmatrix}^T \tag{1}$$

$. $\mathbf{w}^{(0)}$ is the value of $\mathbf{w}$ at the 0^{th} iteration.

2.3a State the value of $\mu^{(0)} = P(Y = 1|X = x)$. This should be an n-dimensional vector.

```
In [7]: w_0 = np.array([[-2],
                        [ 1],
                        [ 0]])
        mu_0 = logistic(X*w_0)
        print(mu_0)
```

```
[[ 0.95257413]
 [ 0.73105858]
 [ 0.73105858]
 [ 0.26894142]]
```

2.3b State the value of $\mathbf{w}^{(1)}$ (the value of w after one iteration).

```
In [8]: w_1 = w_0 - gradient(Q, w_0)
        print(w_1)
```

```
[[-2.        ]
 [ 0.94910188]
 [-0.68363271]]
```

2.3c State the value of $\mu^{(1)}$.

```
In [9]: mu_1 = logistic(X*w_1)
        print(mu_1)
```

```
[[ 0.89693957]
 [ 0.54082713]
 [ 0.56598026]
 [ 0.15000896]]
```

2.3d After performing a second iteration, state the value of $\mathbf{w}^{(2)}$.

```
In [10]: w_2 = w_1 - gradient(Q, w_1)
         print(w_2)
```

```
[[-1.69083609]
 [ 1.91981257]
 [-0.83738862]]
```

Problem 3: Spam classification using Logistic Regression

The spam dataset given to you as part of the homework in `spam.mat` consists of 4601 email messages, from which 57 features have been extracted as follows:

- 48 features giving the proportion (0 to 1) of words in a given message which match a given word on the list. The list contains words such as business, free, george, etc. (The data was collected by George Forman, so his name occurs quite a lot!)
- 6 features giving the proportion (0 - 1) of characters in the email that match a given character on the list. The characters are `;( [ ! $ #`.
- Feature 55: The average length of an uninterrupted sequence of capital letters
- Feature 56: The length of the longest uninterrupted sequence of capital letters
- Feature 57: The sum of the lengths of uninterrupted sequences of capital letters

The dataset consists of a training set size 3450 and a test set of size 1151. One can imagine performing several kinds of preprocessing to this data matrix. Try each of the following separately:

i. Standardize each column so they each have mean 0 and unit variance.

ii. Transform the features using $x_j^{(i)} \leftarrow \log(x_j^{(i)} + 0.1)$.

iii. Binarize the features using $x_j^{(i)} \leftarrow \beth(x_j^{(i)} > 0)$. $\beth$ denotes an indicator variable.

Note: You will need to tune the step size carefully to avoid numerical issues and to avoid a diverging training risk.

```
In [11]: spam = scipy.io.loadmat('data/spam.mat')
         s_Xtrain = np.matrix(spam['Xtrain'])
         s_Ytrain = np.matrix(spam['Ytrain'])

         # preprocess
         s_Xtrain_std = add_ones((s_Xtrain-s_Xtrain.mean(axis=0))/s_Xtrain.std(axis=0))
         s_Xtrain_log = add_ones(np.log(s_Xtrain + 0.1))
         s_Xtrain_bin = add_ones((s_Xtrain > 0).astype(int))
```

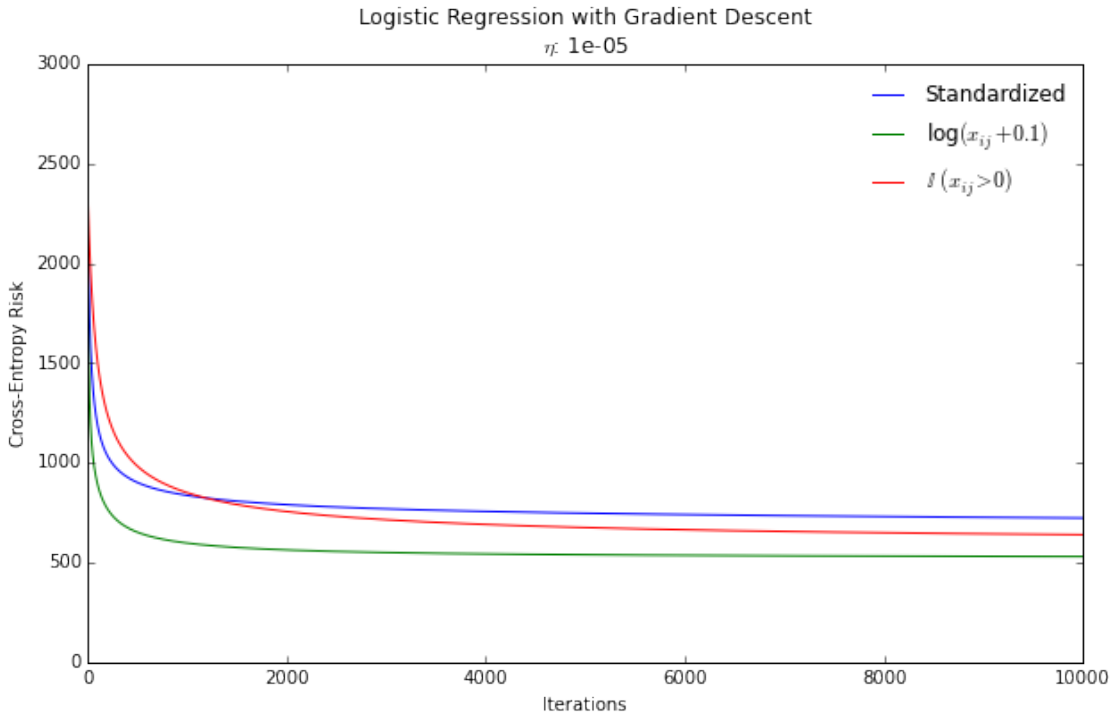
3.1 Implement logistic regression to classify the spam data. Use batch gradient descent. Plot the training risk (the cross-entropy risk of the training set) vs. the number of iterations. You should have one plot for each preprocessing method. Note: One batch gradient descent iteration amounts to scanning through the whole training data and computing the full gradient.

```
In [12]: def logistic(z):
             yhat = 1/(1 + np.exp(-z))
             return yhat

         def cross_entropy(Q, w):
             return np.sum(np.log(1+np.exp(-Q*w)))

         def gradient(Q, w):
             z = Q*w
             return -Q.T*(np.exp(-z)/(1 + np.exp(-z)))

         def batch_gd(X, y, iterations, eta):
             m, n = X.shape
             diag_y = np.zeros((m, m))
             np.fill_diagonal(diag_y, val=y)
             Q = diag_y*X
```

```
        w = np.zeros((n, 1))
        risk_list = []
        for _ in range(iterations):
            w -= eta*gradient(Q, w)
            risk = cross_entropy(Q, w)
            risk_list.append(float(risk))
        return risk_list

    def plot_risk(data, labels, title, ylim=3000):
        plt.figure(figsize=(10, 6))
        for k, v in enumerate(data):
            plt.plot(data[k], label=labels[k])
        plt.legend(frameon=False)
        plt.ylim([0, ylim])
        plt.ylabel("Cross-Entropy Risk")
        plt.xlabel("Iterations")
        plt.title(title)
        plt.show()
```

In [13]:
```
gd_eta = 0.00001

gd_std = batch_gd(s_Xtrain_std, s_Ytrain, 10000, eta=gd_eta)
gd_log = batch_gd(s_Xtrain_log, s_Ytrain, 10000, eta=gd_eta)
gd_bin = batch_gd(s_Xtrain_bin, s_Ytrain, 10000, eta=gd_eta)
```

In [14]:
```
plot_risk(data=[gd_std, gd_log, gd_bin],
          labels=["Standardized", "log$(x_{ij} + 0.1)$", "$\mathbb{I}$ $(x_{ij} > 0)$"],
          title=("Logistic Regression with Gradient Descent\n$\eta$: {}" \
                 .format(gd_eta))
         )
```

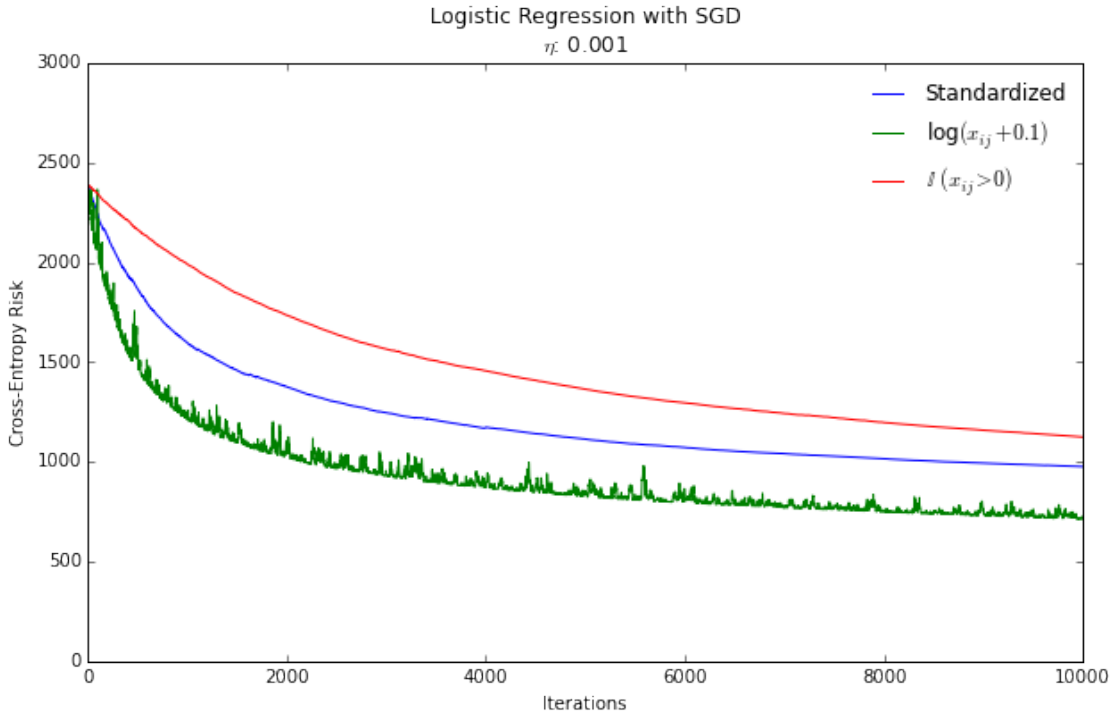


3.2 Derive stochastic gradient descent equations for logistic regression and show your steps. Plot the training risk vs. number of iterations. You should have one plot for each preprocessing method. How are the plots different from (1)? Note: One stochastic gradient descent iteration amounts to computing the gradient using one data point.

Answer: The curves for gradient descent appear to have a steeper fall than stochastic gradient descent. Gradient descent also has a lower Cross Entropy risk for all three feature transformation methods.

The derivation for SGD should be same as above for batch gradient descent. The only difference is that we are updating our weights by computing the gradient for one data point at a time, instead of using the entire dataset for each iteration.

```
In [15]: def stochastic_gradient(Q_i, w):
             z = Q_i*w
             return -Q_i.T * np.exp(-z)/(1 + np.exp(-z))

         def sgd(X, y, iterations, eta):
             m, n = X.shape
             diag_y = np.zeros((m, m))
             np.fill_diagonal(diag_y, val=y)
             Q = diag_y*X
             w = np.zeros((n, 1))
             risk_list = []
             for _ in range(iterations):
                 i = np.random.randint(0, m)
                 w -= eta*gradient(Q[i,:], w)
                 risk = cross_entropy(Q, w)
                 risk_list.append(float(risk))
             return risk_list

In [16]: sgd_eta = 0.001

         sgd_std = sgd(s_Xtrain_std, s_Ytrain, 10000, eta=sgd_eta)
         sgd_log = sgd(s_Xtrain_log, s_Ytrain, 10000, eta=sgd_eta)
         sgd_bin = sgd(s_Xtrain_bin, s_Ytrain, 10000, eta=sgd_eta)

In [17]: plot_risk(data=[sgd_std, sgd_log, sgd_bin],
                   labels=["Standardized", "log$(x_{ij} + 0.1)$", "$\mathbb{I}$ $(x_{ij} > 0)$"],
                   title=("Logistic Regression with SGD\n$\eta$: {}" \
                          .format(sgd_eta))
                  )
```



3.3 Instead of a constant learning rate (η), repeat (2) where the learning rate decreases as $\eta \propto 1/t$ for the t^{th} iteration. Plot the training risk vs number of iterations. Is this strategy better than having a constant $ η? *You should have one plot for each preprocessing method.*

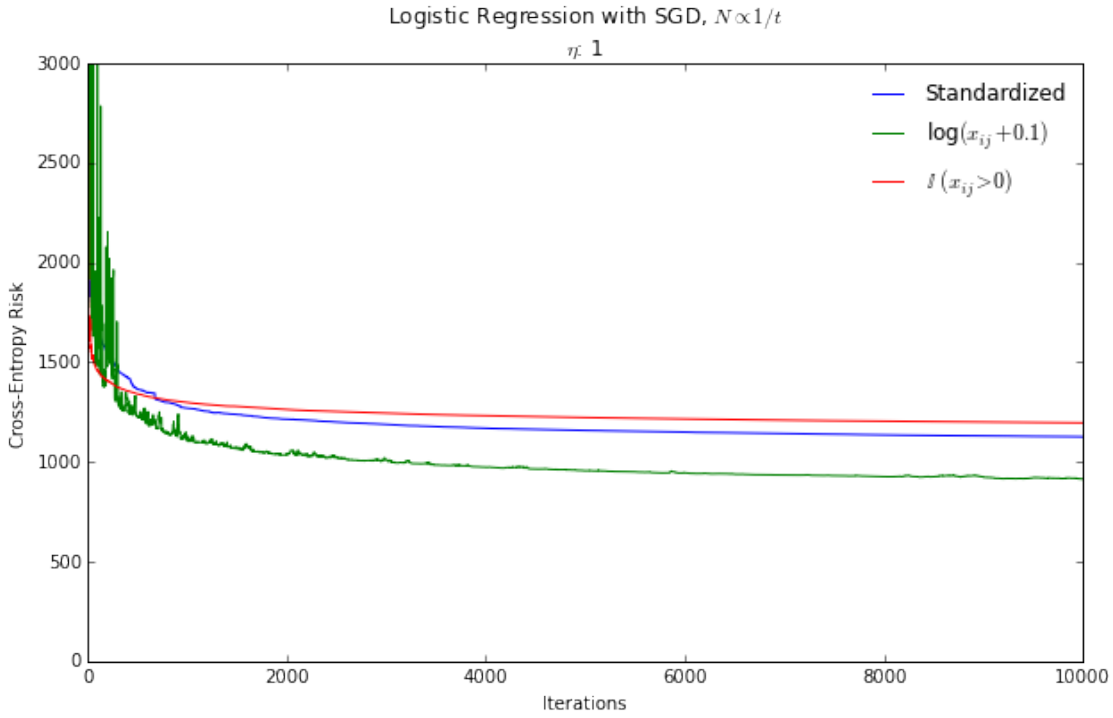
Answer: This increasing learning rate strategy does not appear to be better than normal stochastic descent.

```
In [18]: def sgd_decreasing_rate(X, y, iterations, eta):
             m, n = X.shape
             diag_y = np.zeros((m, m))
             np.fill_diagonal(diag_y, val=y)
             Q = diag_y*X
             w = np.zeros((n, 1))
             risk_list = []
             for index in range(iterations):
                 i = np.random.randint(0, m)
                 w -= 1/(index+1)*eta*gradient(Q[i,:], w)
                 risk = cross_entropy(Q, w)
                 risk_list.append(float(risk))
             return risk_list
```

```
In [19]: sgd_decreasing_rate_eta = 1

         sgd_decreasing_rate_std = sgd_decreasing_rate(s_Xtrain_std, s_Ytrain, 10000, eta=sgd_decreasing
         sgd_decreasing_rate_log = sgd_decreasing_rate(s_Xtrain_log, s_Ytrain, 10000, eta=sgd_decreasing
         sgd_decreasing_rate_bin = sgd_decreasing_rate(s_Xtrain_bin, s_Ytrain, 10000, eta=sgd_decreasing
```

```
In [20]: plot_risk(data=[sgd_decreasing_rate_std, sgd_decreasing_rate_log, sgd_decreasing_rate_bin],
                   labels=["Standardized", "log$(x_{ij} + 0.1)$", "$\mathbb{I}$ $(x_{ij} > 0)$"],
```

```
        title=("Logistic Regression with SGD, $N \propto 1/t$\n$\eta$: {}" \
               .format(sgd_decreasing_rate_eta))
    )
```



3.4a Now let's use kernel logistic regression with a polynomial kernel of degree 2. Our risk is still the same as in problem 2, but our classifier is now:

$$f(\mathbf{x}) = \sum_{i=1}^{n} \alpha_i K(\mathbf{x}^{(i)}, \mathbf{x}) \text{ where } K(\mathbf{x}^{(i)}, \mathbf{x}) = (\mathbf{x}^T \mathbf{x}^{(i)} + 1)^2$$

instead of $f(\mathbf{x}) = \mathbf{w}^T \mathbf{x}$ as it was originally. Show that the stochastic gradient descent update for data point $\mathbf{x}^{(i)}$ is:

$$\alpha_i \leftarrow \alpha_i + \eta S(-z^{(i)}) y^{(i)}$$

$$\text{Where } z^{(i)} = y^{(i)} f(\mathbf{x}^{(i)})$$

$$S(-z^{(i)}) = \frac{1}{1 + exp(z^{(i)})}$$

Note that this measure means $\alpha \in \mathbb{R}^n$. Also derive $\dfrac{\partial L}{\partial \alpha_i}$ directly from the loss function. You should get a different answer than the dual algorithm shown above. Under what condition are the two updates you derived the same?

Answer:

$$\text{Using chain rule: } \frac{\partial L}{\partial \alpha_i} = \frac{\partial L}{\partial z^{(i)}} \frac{\partial z^{(i)}}{\partial f(\mathbf{x}^{(i)})} \frac{\partial f(\mathbf{x}^{(i)})}{\partial \alpha_i}$$

$$\frac{\partial L}{\partial z^{(i)}} = \frac{\partial \log(1 + e^{-z^{(i)}})}{\partial z^{(i)}} = -\frac{1}{1 + exp(z^{(i)})}$$

$$\frac{\partial z^{(i)}}{\partial f(\mathbf{x}^{(i)})} = \frac{\partial y^{(i)} f(x)}{\partial f(x)} = y^{(i)}$$

$$\frac{\partial f(\mathbf{x}^{(i)})}{\partial \alpha_i} = \frac{\partial \alpha_i K(x, x')}{\partial \alpha_i} = K(x, x')$$

$$\frac{\partial L}{\partial \alpha_i} = -\frac{y^{(i)} K(x, x')}{1 + exp(z^{(i)})}$$

This can be re-written as:

$$-S(-z^{(i)}) \cdot y^{(i)} (\mathbf{x}^T \mathbf{x}^{(i)} + 1)^2$$

Which gives us the update rule:

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

3.4b Finally, repeat (2), using the best preprocessing method you found, using kernel logistic ridge regression. Use whichever learning rate scheme you wish. Use $\gamma = 10^{-5}$. You may optionally adjust the value of γ. Generate a plot of training risk vs. number of iterations, and another plot of validation risk vs. number of iterations (use a 2/3, 1/3

split). Use the following update equations (you do not need to derive them):

$$\alpha_i \leftarrow \alpha_i - \gamma \alpha_i + \eta S(-z^{(i)}) y^{(i)}$$

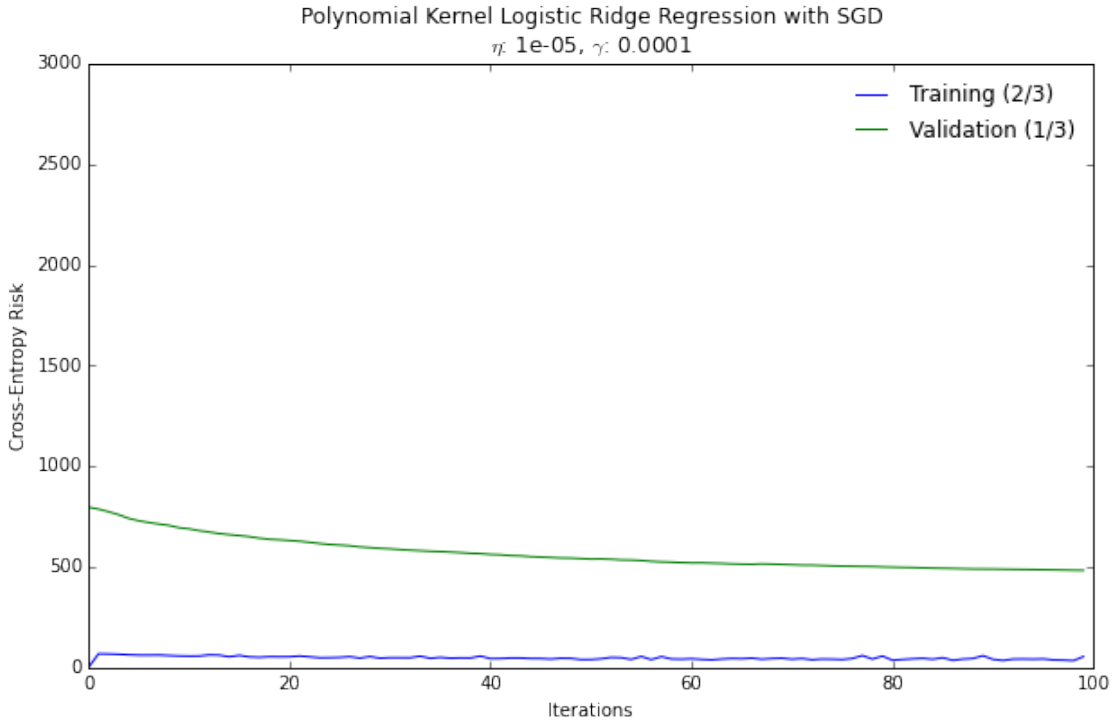
$$\alpha_h \leftarrow \alpha_h - \gamma \alpha_h \text{ for } h \neq i$$

```
In [21]: def fx_poly(x_j, X, alpha):
             k = np.square(X*x_j.T + 1)
             return np.sum(np.multiply(alpha, k))

         def ridge_gradient(z_i, y_i):
             return (1/(1 + np.exp(z_i)))*y_i

         def risk_fun(z_j):
             return np.log(1 + np.exp(z_j))

         def kernel_ridge_sgd(X, y, iterations, eta, gamma, fx):
             # number of training examples
             size = int(X.shape[0] * 2/3)
             data = np.hstack((y, X))
             np.random.shuffle(data)

             X_test,  y_test  = data[size:,1:], data[size:,0]
             X_train, y_train = data[:size,1:], data[:size,0]
             m, n = X_train.shape
             alpha = np.zeros((m,1))

             temp_train_risk = []
             temp_test_risk = []
```

```
    train_risk = []
    test_risk = []
    for index in range(iterations):
        j = np.random.randint(0, m)
        h = np.ones(m, dtype=bool)
        h[[j]] = False

        x_j = X_train[j,:]
        y_j = y_train[j]
        z_j = y_j * fx(x_j, X_train, alpha)

        risk_j = risk_fun(z_j)
        temp_train_risk.append(risk_j)

        alpha[[j]] -= gamma*alpha[[j]] + eta*ridge_gradient(z_j, y_j)
        alpha[[h]] -= gamma*alpha[[h]]

        # sum risks for every batch of 100 values
        if index % 100 == 0:
            train_sum = np.sum(temp_train_risk)
            temp_train_risk = [] # reset the training risk
            train_risk.append(train_sum)

            # get validation results
            for v in range(X_test.shape[0]):
                x_v = X_test[v,:]
                y_v = y_test[v]
                z_v = y_v * fx(x_v, X_train, alpha)
                risk_v = risk_fun(z_v)
                temp_test_risk.append(risk_v)

            test_sum = np.sum(temp_test_risk)
            temp_test_risk = []
            test_risk.append(test_sum)

    return train_risk, test_risk
```

In [22]:
```
ridge_sgd_eta = 10e-6
ridge_sgd_gama = 10e-5

ridge_sgd_std, ridge_sgd_std_test = kernel_ridge_sgd(
    s_Xtrain_std,
    s_Ytrain,
    10000,
    eta=ridge_sgd_eta,
    gamma=ridge_sgd_gama,
    fx=fx_poly
)
```

In [23]:
```
plot_risk(data=[ridge_sgd_std, ridge_sgd_std_test],
          labels=["Training (2/3)", "Validation (1/3)"],
          title=("Polynomial Kernel Logistic Ridge Regression with SGD\n$\eta$: {}, $\gamma$:
                 .format(ridge_sgd_eta, ridge_sgd_gama)),
          ylim=3000)
```

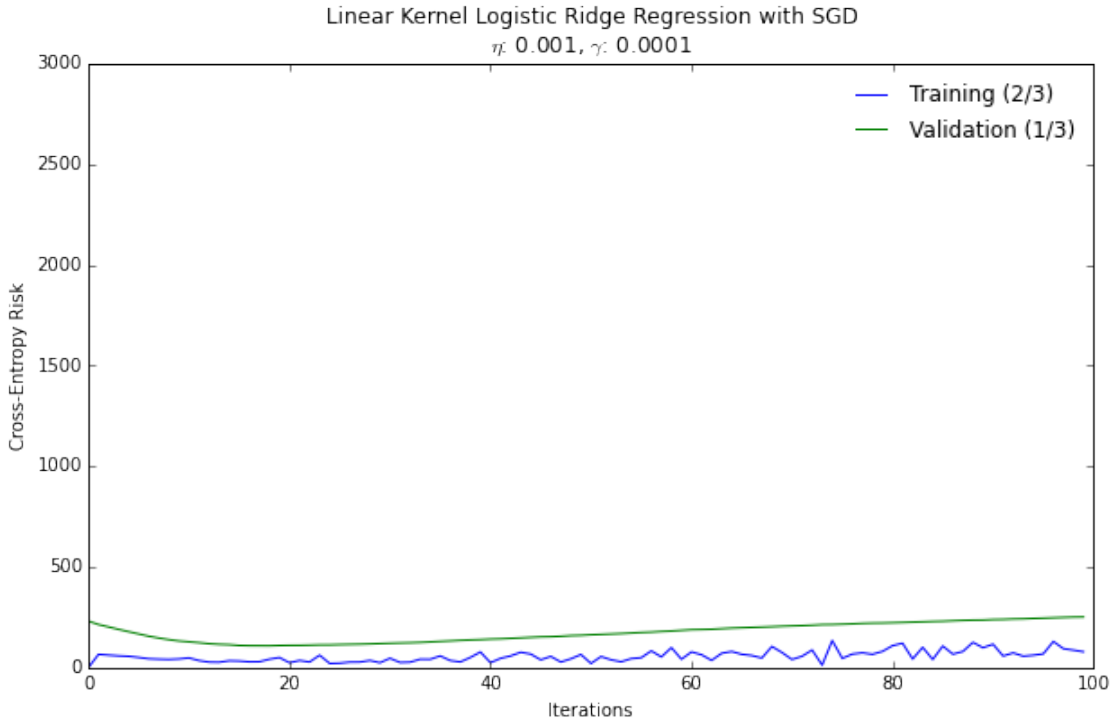


Repeat the same experiment with the linear kernel $K(x^{(i)}, x) = x^T x^{(i)} + 1$. Does the quadratic kernel overfit the data? For each kernel, should you decrease or increase γ to try to improve performance?

Answer: The quadratic kernel did appear to overfit the data, evidenced by the higher validation error in comparison to the training error. Using a larger γ (e.g. 1) gave me a flat risk (or would result in overflow errors) for both validation and training.

```
In [24]: def fx_linear(x_j, X, alpha):
             k = X*x_j.T + 1
             return np.sum(np.multiply(alpha, k))

         ridge_sgd_lin_eta = 10e-4
         ridge_sgd_lin_gamma = 10e-5

         ridge_sgd_std_lin, ridge_sgd_std_lin_test = kernel_ridge_sgd(
             s_Xtrain_std[:1000],
             s_Ytrain[:1000],
             10000,
             eta=ridge_sgd_lin_eta,
             gamma=ridge_sgd_lin_gamma,
             fx=fx_linear
         )
```

```
In [25]: plot_risk(data=[ridge_sgd_std_lin, ridge_sgd_std_lin_test],
                   labels=["Training (2/3)", "Validation (1/3)"],
                   title=("Linear Kernel Logistic Ridge Regression with SGD\n$\eta$: {}, $\gamma$: {}"
                          .format(ridge_sgd_lin_eta, ridge_sgd_lin_gamma)),
                   ylim=3000)
```



Problem 4: Real World Spam Classification

Daniel recently interned as an anti-spam product manager for a large email service provider. His company uses a linear SVM to predict whether an incoming spam message is spam or ham. He notices that the number of spam messages received tends to spike massively upwards a couple minutes before and after midnight. Eager to obtain a return offer, he adds the time-stamp of the received message, stored as number of milliseconds since the previous midnight, to each feature vector for the SVM to train on, in hopes that the ML model will be able to identify the abnormal spike in spam volume at night. To his dismay, after A/B testing with his newly added feature, Daniel discovers that the linear SVM's success rate barely improves. He wants to try to use a kernel to improve his model, but unfortunately he is limited to a quadratic kernel.

Why can't the linear SVM utilize the new feature well, and what can Daniel do to improve his results? This is an actual interview question Daniel received for a machine learning engineering position!

Write a short explanation. This question is open ended and there can be many correct answers. It is not necessary, but feel free to do your own research as long as you cite any sources.

Answer: Since the timestamps are for the *previous* night, we would expect to get very small and very large values for this feature given that spam emails may be sent right before or after midnight. Since this data are non-linear, we would need a kernel that could find an optimal descision boundary. By using cross-validation to determine the best polynomial kernel, Daniel could potentially find the boundary. Another approach would be to create a new feature to determine the number of milliseconds from the *nearest* midnight period instead of from the *previous* midnight.