February 8, 2018

# 1 2.1

The feature normalization has been implemented in the code:

```
#The new normalization that takes one array as input

def feature_normalization_nugget(x):
    std = np.std(x, axis=0)      #Standard Deviation of the feature values
    mini = x.min(axis=0)         #Minimum value of the column
    ptp = x.ptp(axis=0)          #Difference of min and max of the column


    for i in range(0, x.shape[1]):
        if not std[i] == 0:
            x[:,i] = (x[:,i]-mini[i])/ptp[i]     #Min-Max normalization

    return(x)


def feature_normalization(train, test):

    #Defined a new function that takes one array at a
    time as input called feature_normalization_nugget

    return(feature_normalization_nugget(train), feature_normalization_nugget(test))
```

## 2 2.2.1

Assuming the given in the question,

The “square loss” objective function:

$$J(\theta) = \frac{1}{m} \sum_{i=1}^{m} \left( h_\theta(x_i) - y_i \right)^2 ,$$

can be written as

$$J(\theta) = \frac{1}{m} (X\theta - y)^T (X\theta - y)$$

where $X$ is the "Design Matrix" given in the question, and $y$ is the "response vector."

# 3 2.2.2

Using identities for Matrix differentiation, we can express the gradient in term of Matrix and Vectors.

$$\nabla J(\theta) = \nabla \frac{1}{m}(X\theta - y)^T(X\theta - y)$$

$$\implies \nabla J(\theta) = \frac{1}{m}\nabla(\theta^T X^T X\theta - \theta^T X^T y - y^T X\theta + y^T y)$$

$$\implies \nabla J(\theta) = \frac{1}{m}\nabla tr(\theta^T X^T X\theta - \theta^T X^T y - y^T X\theta + y^T y)$$

$$\implies \nabla J(\theta) = \frac{1}{m}\nabla(tr\theta^T X^T X\theta - 2try^T X\theta + try^T y)$$

$$\implies \nabla J(\theta) = \frac{1}{m}(X^T X\theta + X^T X\theta - 2X^T y)$$

$$\implies \nabla J(\theta) = \frac{2}{m}(X^T X\theta - X^T y)$$

# 4 2.2.3

Approximate expression for change in Objective function

$$J(\theta + \eta h) - J(\theta) = \eta[\nabla J(\theta)]^T h$$

Furthermore, when h $= -\nabla J(\theta)$

$$J(\theta + \eta h) - J(\theta) = -\eta||\nabla J(\theta)||$$

## 5 2.2.4

The expression for updating the $\theta$ in Gradient descent algorithm is:

$$\theta = \theta - \nabla J(\theta)\eta$$

$$\implies \theta = \theta - \eta\frac{2}{m}(X^T X\theta - X^T y)$$

## 6 2.2.5

The code follows:

```
def compute_square_loss(X, y, theta):

    loss = 0

    m = X.shape[0]        #no. of training cases

    normal = (X.dot(theta) - y)
    transpose = ((X.dot(theta) - y).transpose())

    loss = (transpose.dot(normal))/m        #Square loss in Matrix form

    return(loss)
```

## 7 2.2.6

The code follows:

```
def compute_square_loss_gradient(X, y, theta):

    m = X.shape[0]

    Xtranspose = X.transpose()
    Xnorm = Xtranspose.dot(X)

    grad = (2*(Xnorm.dot(theta) - Xtranspose.dot(y)))/m       #Gardient at theta

    return(grad)
```

## 8 2.3

`grad_checker` has been implemented in the code, and the gradient output agrees with output from `compute_square_loss_gradient`.

The code follows:

```
def grad_checker(X, y, theta, epsilon=0.000001, tolerance=1e-4):

    true_gradient = compute_square_loss_gradient(X, y, theta)
    num_features = theta.shape[0]
    approx_grad = np.zeros(num_features) #Initialize the gradient we approximate

    identity = np.identity(num_features)
    identity = epsilon*identity

    for i in range(num_features):
        val = (compute_square_loss(X, y, theta + identity[:,i])
             - compute_square_loss(X, y, theta - identity[:,i]))/2*epsilon
        approx_grad[i] = val

    for i in range(num_features):
        if abs(true_gradient[i] - approx_grad[i]) > tolerance:
            return(False, true_gradient, approx_grad)
            break
        return(True)
```

## 9 2.4.1

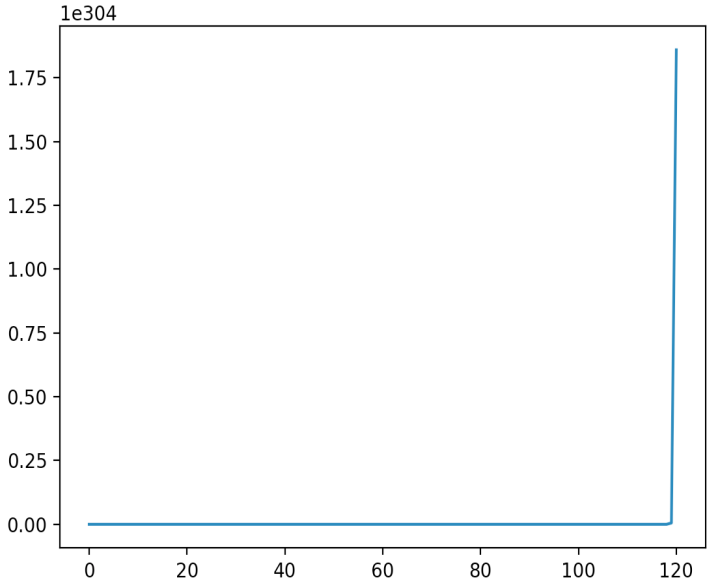
`batch_gradient_descent` has been implemented. The code follows:

```
def batch_grad_descent(X, y, alpha=0.05, num_iter=1000, check_gradient=False):

    for i in range (num_iter):
        loss_hist[i] = compute_square_loss(X,y,theta)
        theta = theta - alpha*compute_square_loss_gradient(X,y,theta)
#Update theta against the gradient
        theta_hist[i,:] = theta          #Theta history updates

    return(theta_hist, loss_hist)
```

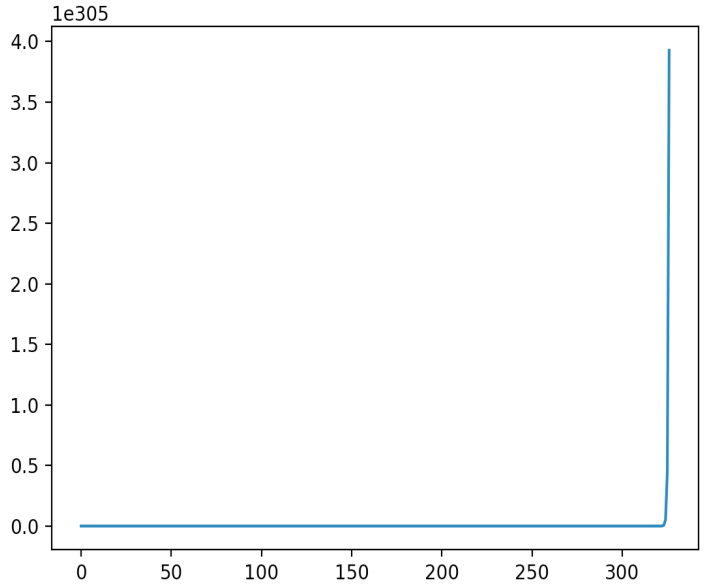
# 10 2.4.2

The loss function blows up for the $step-size = 0.5$, and blows up a little later for a $step-size = 0.1$. In fact, the loss diverges for a step-size of 0.051 while converges for a value of 0.05, as can be seen in the figure. The figure also shows that the loss converges faster for step-size of 0.05 compared to 0.01 which is to be expected. And since, the loss diverges for 0.51, and converges for 0.5, it is (near) the biggest step-size which still ensures convergence, and therefore preferable.

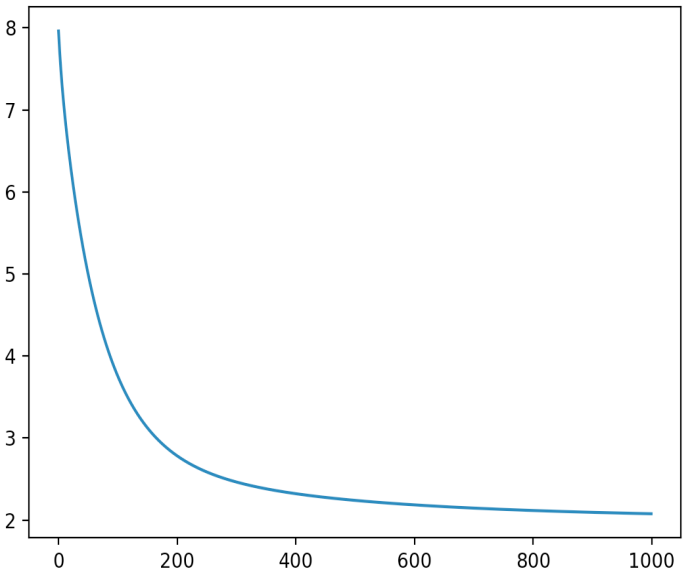
The following are the graphs of loss function against iteration step for various different step-size.



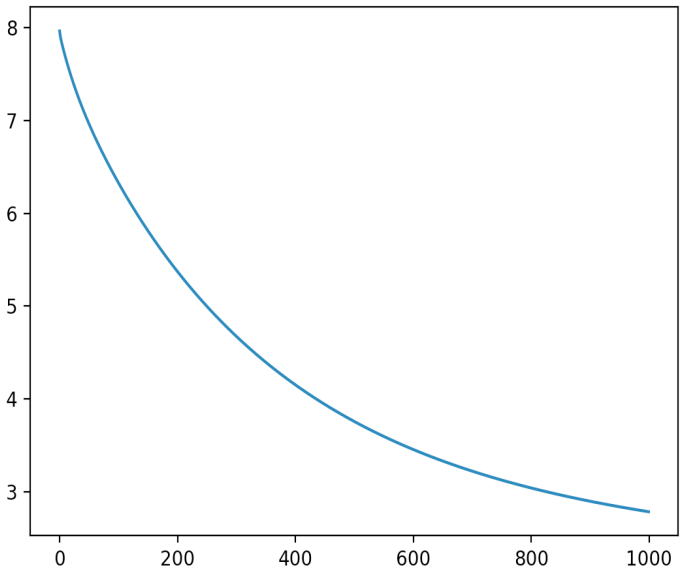

$step - size = 0.5$



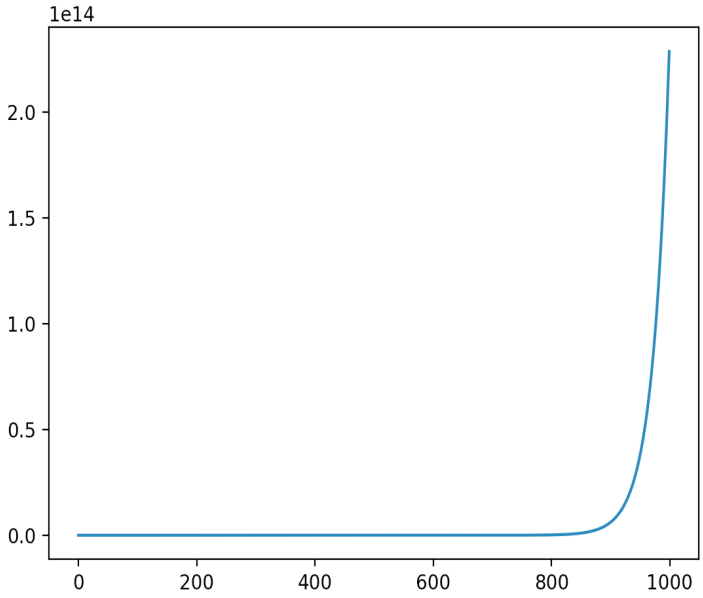

$step - size = 0.1$




$step - size = 0.05$




$step - size = 0.01$




$step - size = 0.51$

## 11 2.5.1

The gradient for ridge regression is calculated as follows:

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m}(h_\theta(x_i) - y_i)^2 + \lambda\theta^T\theta,$$

$$\implies J(\theta) = \frac{1}{m}(X\theta - y)^T(X\theta - y) + \lambda\theta^T\theta$$

Borrowing the calculation from 2.2 and differentiation of $\lambda\theta^T\theta$

$$\implies \nabla J(\theta) = \frac{2}{m}(X^TX\theta - X^Ty) + 2\lambda\theta$$

Furthermore, the expression for updating $\theta$ is:

$$\implies \theta = \theta - \eta[\frac{2}{m}(X^TX\theta - X^Ty) + 2\lambda\theta]$$

## 12 2.5.2

`compute_regularized_square_loss_gradient` has been implemented in the code, whcih follows:

```
def compute_regularized_square_loss_gradient(X, y, theta, lambda_reg):
    m = X.shape[0]


    Xtranspose = X.transpose()
    Xnorm = Xtranspose.dot(X)

    grad = ((2*(Xnorm.dot(theta) - Xtranspose.dot(y)))/m) + 2*lambda_reg*theta

    return(grad)
```

## 13 2.5.3

`regularized_grad_descent` has been implemented in the code, which follows:

```
def regularized_grad_descent(X, y,  alpha=0.03, lambda_reg = 0.02245, num_iter=1000):

    regularization = (theta.transpose()).dot(theta)
    for i in range (num_iter):
        loss_hist[i] = compute_square_loss(X,y,theta) + lambda_reg*regularization
        theta = theta -
                alpha*compute_regularized_square_loss_gradient(X,y,theta, lambda_reg)
        theta_hist[i,:] = theta

    return(theta_hist, loss_hist)
```

## 14 2.5.4

Making B large decreases the effective regularization of the bias term because it decreases the coefficient of the bias term in the linear function. Since intercept needs to be maintained, making B should simultaneously make $\theta_{m+1}$ small. And in the expression for updating $\theta$ we can see that smaller $\theta_{m+1}$ will result in smaller regularization of the bias term.

## 15 2.5.5

For a Mathematical Proof, the regularization term = $\theta_{m+1}$

To show:

$$B \nearrow \implies \theta_{m+1} \searrow$$

We have, for the linear model, 'C' as the constant intercept,

$$B\theta_{m+1} = C$$

$$\implies \theta_{m+1} = \frac{C}{B}$$

$$\implies B \nearrow \implies \theta_{m+1} \searrow$$

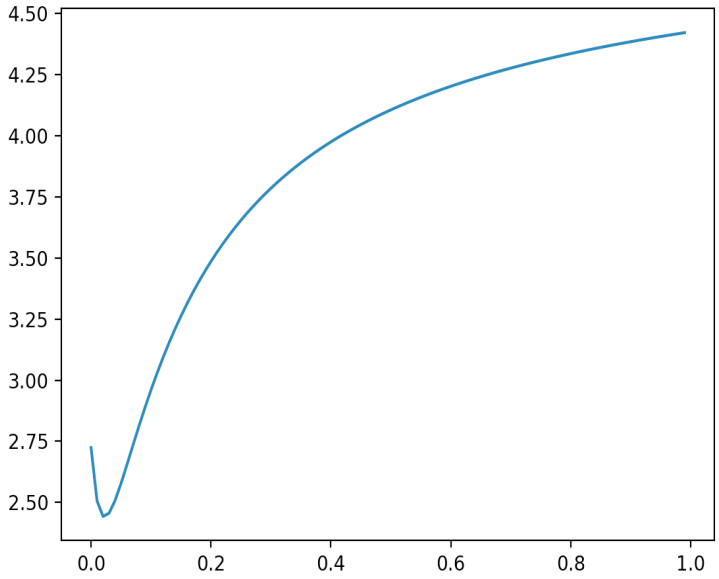
## 16 2.5.6

On trying various values of B to see what performs the best in test data, $B = 3$ seems to result in the lowest value for the loss function. The loss returns 4.411 for $B = 3$ and the value of loss increases on both increasing and decreasing B.

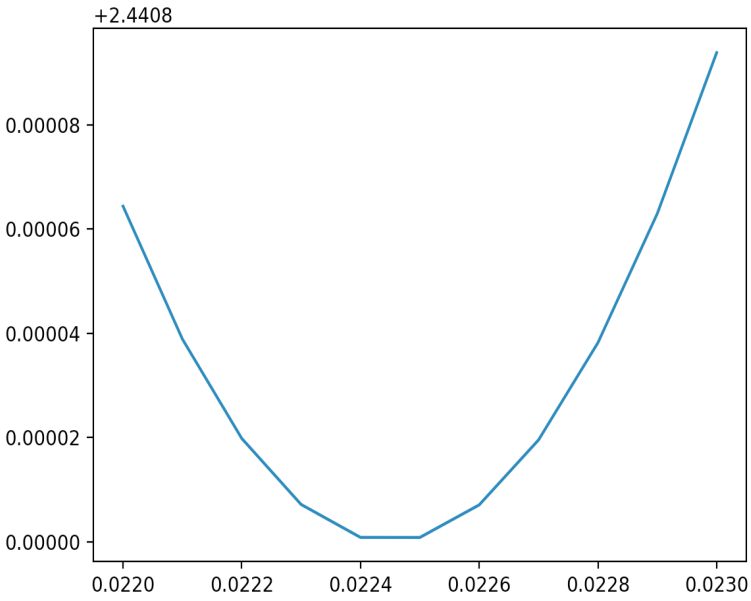
## 17 2.5.7

Upon plotting the test loss and training loss against $\lambda$, we find that the plot for test loss has a minimum that is different from zero, while (and as we expect) the training loss keeps increasing with the increase in $\lambda$.

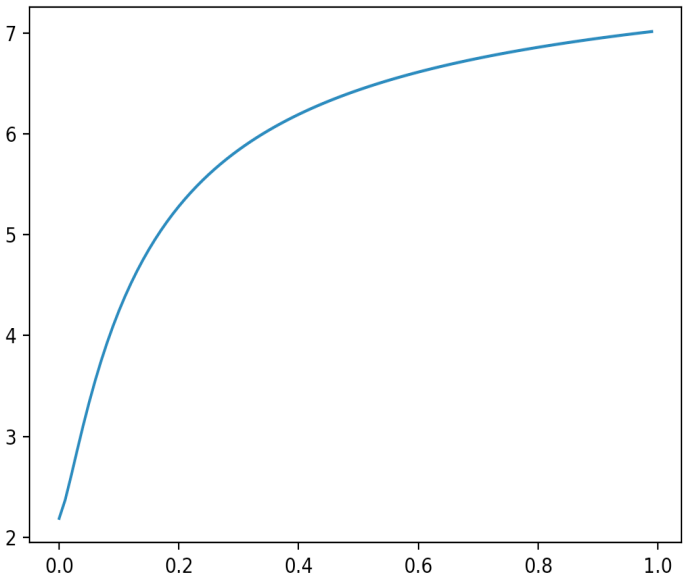
Upon search, the value for $\lambda$ that minimizes the test loss appear to be (as is evident in the plot below) 0.02245




$Test - loss01$




$Test - loss02$




$Train - loss$

## 18 2.5.8

I would select the $\theta$ produced by the ridge regression with $\lambda$ that, under cross-validation, minimizes the test loss.

## 19 2.6.1

We can simplify the objective function:

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m}\left(h_\theta(x_i) - y_i\right)^2 + \lambda\theta^T\theta$$

$$\Rightarrow J(\theta) = \frac{1}{m}\sum_{i=1}^{m}\left(h_\theta(x_i) - y_i\right)^2 + \frac{1}{m}\sum_{i=1}^{m}\lambda\theta^T\theta$$

$$\Rightarrow J(\theta) = \frac{1}{m}\sum_{i=1}^{m}\left[\left(h_\theta(x_i) - y_i\right)^2 + \lambda\theta^T\theta\right]$$

Comparing to $J(\theta) = \frac{1}{m}\sum_{i=1}^{m} f_i(\theta)$, we can conclude:

$$f_i(\theta) = \left(h_\theta(x_i) - y_i\right)^2 + \lambda\theta^T\theta$$

And in Matrix form,

$$f_i(\theta) = \left(\theta^T x_i - y_i\right)^2 + \lambda\theta^T\theta$$

## 20 2.6.2

Since the underlying distribution is uniform distribution, we have $\forall i \in \{1, \ldots, n\}$

$$Prob(\nabla f_i(\theta)) = \frac{1}{m}$$

So,

$$\mathbb{E}[\nabla f_i(\theta)] = \frac{1}{m} \sum_{i=1}^{m} \nabla f_i(\theta)$$

$$\Rightarrow \mathbb{E}[\nabla f_i(\theta)] = \frac{\nabla}{m} \sum_{i=1}^{m} f_i(\theta)$$

$$\Rightarrow \mathbb{E}[\nabla f_i(\theta)] = \nabla J(\theta)$$

## 21 2.6.3

The update rule is similar to that of ridge regression apart from the fact that we use a single training data instead of a matrix of it.

$$\implies \theta = \theta - \eta[\frac{2}{m}(x_i^T x_i \theta - x_i^T y) + 2\lambda\theta]$$

## 22 2.6.4

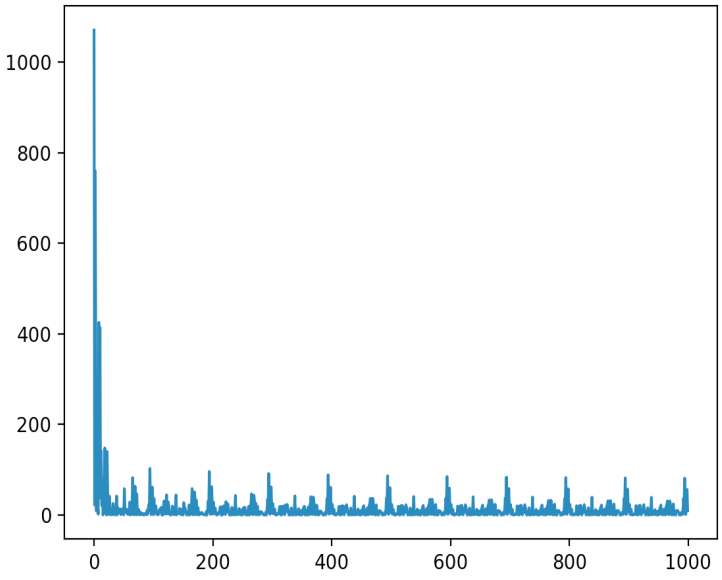
`Stochastic_grad_descent` has been implemented in the code, which follows:

```
def stochastic_grad_descent(X, y, alpha=0.003, lambda_reg=0.02245, num_iter=1000):

    rand = np.arange(num_instances)
    randx = np.random.shuffle(rand)


    for i in range (num_iter):

        inputx = X[rand[i % (num_instances)],:]
        inputx.resize((1,num_features))

        inputy = y[i % (num_instances)]

        regularization = (theta.transpose()).dot(theta)
        loss_hist[i] = compute_square_loss(inputx,inputy,theta) +
                       lambda_reg*regularization


        grad = compute_regularized_square_loss_gradient(inputx,inputy,theta, lambda_reg

        if i>10: alpha = 0.0000001/(1 + 0.0000001*lambda_reg*i)
        theta = theta - alpha*grad[1,:]

        theta_hist[i,:] = theta

    return(theta_hist, loss_hist)
```

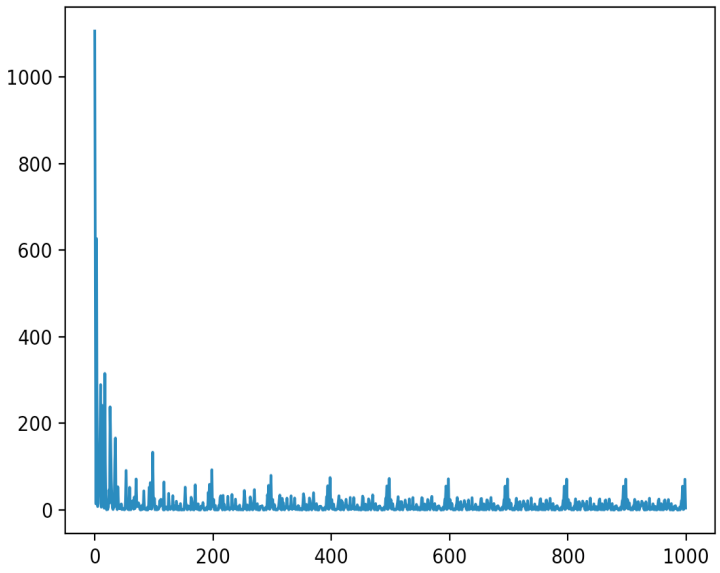
## 23 2.6.5

The following are the graphs of loss function against number of steps. The loss of step-size $\frac{1}{t}$ seems to regularly be lower than that for constant step-size. It might not be immediately clear from the graph, but $\frac{1}{t}$ performs better than regular step-size (compare 4.4 loss value of decreasing step-size $\frac{1}{t}$ against 9.5 of constant step-size). Also, $\frac{1}{t}$ step-size appears to have more variance towards the beginning compared to constant step-size.

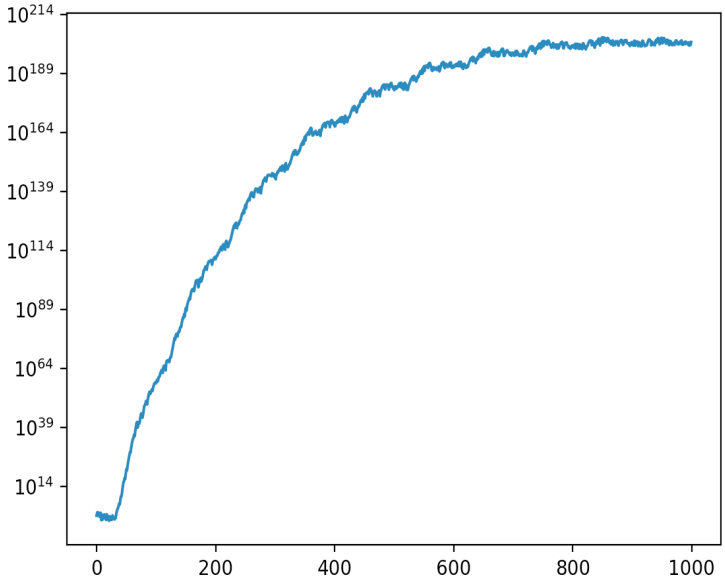
Meanwhile, for the same number of iterations, $\frac{1}{\sqrt{t}}$ step-size appears to completely diverge and has thus been plotted in log scale for the loss value below.




$step - size = 0.003$




$step - size = 1/t$




$step - size = 1/\sqrt{t}$

## 24 2.6.6

The step-size $\eta_t = \frac{\eta_0}{1+\eta_0 \lambda t}$ results in a graph that looks very similar to the graph of constant step-size. But, for $\eta_0 = 10^{-7}$ the loss seems to converge to a value ranging from 0.15 to 1, which is the lowest loss we have seen for any algorithm so far.

## 25 3.1.1

Assuming the setting in the question,

$$a^* = argmin_a \, \mathbb{E}(a - y)^2$$
$$\Rightarrow a^* = argmin_a \, \mathbb{E}(a^2 + y^2 - 2ay)$$
$$\Rightarrow a^* = argmin_a (a^2 + \mathbb{E}\, y^2 - 2a\, \mathbb{E}\, y)$$

Differentiating w.r.t 'a' and equating to zero to find the minimum, which exists because we have a convex curve.

$$\frac{d}{da}(a^2 + \mathbb{E}\, y^2 - 2a\, \mathbb{E}\, y) = 0$$
$$\Rightarrow (2 - 2\, \mathbb{E}\, y) = 0$$
$$\Rightarrow a^* = \mathbb{E}\, y$$

Now, to find Bayes Risk,

$$R(a^*) = \mathbb{E}(a^* - y)^2$$
$$\Rightarrow R(a^*) = \mathbb{E}(\mathbb{E}\, y - y)^2$$
$$\Rightarrow R(a^*) = \mathbb{E}((\mathbb{E}\, y)^2 + y^2 - 2y\, \mathbb{E}\, y)$$
$$\Rightarrow R(a^*) = (\mathbb{E}\, y)^2 + \mathbb{E}\, y^2 - 2(\mathbb{E}\, y)^2$$
$$\Rightarrow R(a^*) = \mathbb{E}\, y^2 - (\mathbb{E}\, y)^2$$
$$\Rightarrow R(a^*) = Var(y)$$

## 26 3.1.2a

To find $f^*(x) = argmin_a \left[(a-y)^2 \mid x\right]$, we can use the result from the previous question and find the solution right away as: $f^*(x) = \mathbb{E}[y \mid x]$

## 27 3.1.2b

We know,

$$\mathbb{E}\left[(f^*(x) - y)^2 \mid x\right] \leq \mathbb{E}\left[(f(x) - y)^2 \mid x\right]$$

We take expectation on both side, which still preserves the inequality,

$$\mathbb{E}[\mathbb{E}\left[(f^*(x) - y)^2 \mid x\right]] \leq \mathbb{E}[\mathbb{E}\left[(f(x) - y)^2 \mid x\right]]$$

By law of iterated expectation,

$$\Rightarrow \mathbb{E}\left[(f^*(x) - y)^2\right] \leq \mathbb{E}\left[(f(x) - y)^2\right]$$

This is true because we found $f^*(x)$ such that it minimizes risk among all possible functions from our hypothesis space, so it would follow that expectation of the loss, which is risk, of this function is smaller than that of all other functions.

## 28 3.2

Given $\ell(\hat{y}, y) = |y - \hat{y}|$, suppose

$$y^* = argmin_a \, \mathbb{E} \, |y - a|$$

$$\Rightarrow y^* = argmin_a [P(y > a) \, \mathbb{E}(y - a) + P(y < a)(- \mathbb{E}(y - a))]$$

$$\Rightarrow y^* = argmin_a \, \mathbb{E}(y - a)[P(y > a) - P(y < a)]$$

$$\Rightarrow y^* = argmin_a (\mathbb{E} \, y - a)[P(y > a) - P(y < a)]$$

Differentiating w.r.t 'a' to find the minimum:

$$\frac{d}{da}(\mathbb{E} \, y - a)[P(y > a) - P(y < a)] = 0$$

$$\Rightarrow -[P(y > a) - P(y < a)] = 0$$

$$\Rightarrow P(y > a) = P(y < a)$$

As sum of probability is 1:

$$\Rightarrow P(y > a) = P(y < a) = \frac{1}{2}$$

So $y^*$ is the median of $y$.
Following that, if $f^*(x) = argmin_a \, [|a - y| \mid x]$
Then by our conclusion, $f^*(x) = Meadian(y|x)$