

# BISHOP QUESTIONS SOLUTIONS

Ex (1.1)

We have been given the following identities:

$$\int_{-\infty}^{\infty} \exp\left\{-\frac{\lambda}{2}x^2\right\} dx = \left(\frac{2\pi}{\lambda}\right)^{1/2} \quad (1)$$

and

$$\prod_{i=1}^d \int_{-\infty}^{\infty} e^{-x_i^2} dx_i = S_d \int_0^{\infty} e^{-r^2} r^{d-1} dr \quad (2)$$

where  $S_d$  is the surface area of the unit sphere in  $d$ -dimensions. We need to prove that:

$$S_d = \frac{2\pi^{d/2}}{\Gamma(d/2)} \quad (3)$$

where  $\Gamma(x)$  is the gamma function defined by:

$$\Gamma(x) = \int_0^{\infty} u^{x-1} e^{-u} du \quad (4)$$

Using (1) and substituting  $\lambda = 2$ , we have:

$$\int_{-\infty}^{\infty} \exp\left\{-\frac{2}{2}x^2\right\} dx = \left(\frac{2\pi}{2}\right)^{1/2}$$

or  $\int_{-\infty}^{\infty} \exp\{-x^2\} dx = \pi^{1/2} \quad (5)$

Substituting in (2),

$$\prod_{i=1}^d \int_{-\infty}^{\infty} e^{-x_i^2} dx_i = \prod_{i=1}^d \pi^{1/2}$$

$$\Rightarrow \pi^{d/2} = S_d \int_0^{\infty} e^{-u} u^{(d-1)/2} du$$

$$\Rightarrow \pi^{d/2} = S_d \int_0^{\infty} e^{-r^2} r^{d-1} dr$$

Now, converting the RHS into an equation in  $u$   
 where  $x^2 = u$  and  $dx = \frac{du}{2\sqrt{u}}$ , we get

$$\begin{aligned} \pi^{d/2} &= S_d \int_0^\infty e^{-u} u^{(d-1)/2} \cdot \frac{du}{2u^{1/2}} \\ \Rightarrow 2\pi^{d/2} &= S_d \int_0^\infty e^{-u} u^{(d/2-1)} du \end{aligned}$$

The term inside integral on RHS reduces to the definition of  $\Gamma(x)$  with  $x = \frac{d}{2}$

Hence, we have

$$\begin{aligned} 2\pi^{d/2} &= S_d \cdot \Gamma(d/2) \\ \Rightarrow S_d &= \boxed{\frac{2\pi^{d/2}}{\Gamma(d/2)}} \quad \text{Hence, proved.} \end{aligned}$$

Next, we have been given that  $\Gamma(1) = 1$  — (A)  
 $\Gamma(3/2) = \frac{\sqrt{\pi}}{2}$  — (B)  
 We need to prove that (3) reduces to well known expressions for  $d=2$  and  $d=3$ .

To start with, we know that:

$$S_2 = 2\pi r \quad [\text{where } r \text{ is the radius}]$$

$$= 2\pi \quad \text{at } r=1 \quad \text{--- (6)}$$

$$\begin{aligned} \text{and } S_3 &= 4\pi r^2 \\ &= 4\pi \quad \text{at } r=1 \quad \text{--- (7)} \end{aligned}$$

Using (3), we have:

$$\begin{aligned} S_2 &= \frac{2\pi^{2/2}}{\Gamma(2/2)} \quad [d=2] \\ &= \frac{2\pi}{\Gamma(1)} = 2\pi \quad (\text{from (A)}) \end{aligned}$$

$$\Rightarrow S_2 = 2\pi^{3/2} \quad - \text{same as (6)}$$

Similarly,  $S_3 = \frac{2\pi^{3/2}}{r(3/2)}$   
 $= \frac{2\pi^{3/2}}{\sqrt{\pi}/2}$  using (B)  
 $\Rightarrow S_3 = 4\pi \quad - \text{same as (7)}$

Hence,  $S_2, S_3$  values reduce to correct expressions.

1.2

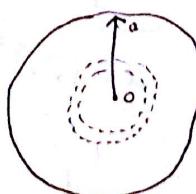
Using (3), we need to show that the volume of a hypersphere of radius 'a' in d-dimensions is given by

$$V_d = \frac{S_d a^d}{d} \quad - (8)$$

We know that the surface area of any d-dimensional sphere with radius 'a' is given as:

$$S_{da} = S_d \cdot a^{d-1}$$

where  $S_d = S_{d1}$ , putting  $a=1$  for a d-dimensional sphere, the volume of all of the sphere may be expressed as the sum of all surface areas of spheres of radii from zero through 'a'



Thus, we can say that the volume of a sphere with d-dimensions and radius 'a' is given as:

$$V_{da} = \int_0^a S_d a^{d-1} da$$

$$= S_d \int_0^a a^{d-1} da \quad [\because S_d \text{ is for a unit sphere and hence, independent of } a]$$

$$= S_d \left[ \frac{a^d}{d} \right]_0^a$$

$$\Rightarrow V_{da} = \frac{S_d \cdot a^d}{d} \quad \text{Hence, proved}$$

Next, we need to show that the ratio of the volume of a hypersphere of radius 'a' to the volume of a hypercube of side '2a' (i.e. The circumscribed hypercube) is given by:

$$\frac{\text{Volume of sphere}}{\text{Volume of cube}} = \frac{\pi^{d/2}}{d \cdot 2^{d-1} \cdot r(d/2)} \quad \text{--- (9)}$$

We know that the volume of a d-dimensional cube with side '2a' is given as:

$$(V_d)_{\text{cube}} = (2a)^d \quad \text{--- (10)}$$

Therefore, we have:

$$\begin{aligned} \frac{V_{da}}{(V_d)_{\text{cube}}} &= \frac{S_d a^d}{d \cdot (2a)^d} \quad (\text{from (8) \& (10)}) \\ &= \frac{S_d}{d \cdot 2^d} \end{aligned}$$

$$= \frac{2\pi^{d/2}}{r(d/2) \cdot d \cdot 2^d} \quad (\text{from } ③)$$

$$\Rightarrow \left[ \frac{\text{Volume of Sphere}}{\text{Volume of Cube}} = \frac{\pi^{d/2}}{d \cdot 2^{d-1} \cdot r(d/2)} \right] \text{ Hence, proved.}$$

for the next part, we need to use Stirling's Approximation:

$$r(x+1) \approx (2\pi)^{1/2} e^{-x} x^{x+1/2} \quad — ⑪$$

which is valid when  $x$  is large.

We need to prove that as  $d \rightarrow \infty$ , the ratio in  
⑨ goes to zero.

Using integration on ④, we get the property

$$r(\alpha) = (\alpha-1) r(\alpha-1)$$

Putting  $\alpha = \frac{d}{2} + 1$ , we have:

$$\begin{aligned} r\left(\frac{d}{2} + 1\right) &= \frac{d}{2} r\left(\frac{d}{2}\right) \\ \Rightarrow r\left(\frac{d}{2}\right) &= \frac{2}{d} r\left(\frac{d}{2} + 1\right) \end{aligned} \quad — ⑫$$

Putting  $x$  as  $\frac{d}{2}$  in ⑪ and substituting in ⑫,

we get:

$$r\left(\frac{d}{2}\right) = \frac{2}{d} \left[ (2\pi)^{1/2} \cdot e^{-d/2} \left(\frac{d}{2}\right)^{d/2 + 1/2} \right]$$

Using the above result in ⑨,

$$\frac{V_{da}}{(V_a)_{\text{cube}}} = \frac{\pi^{d/2}}{d \cdot 2^{d-1} \cdot \frac{2}{d} \cdot (2\pi)^{1/2} \cdot e^{-d/2} \cdot \left(\frac{d}{2}\right)^{\frac{d+1}{2}}}$$

$$= \frac{\pi^{d/2}}{2^{d/2} \cdot (\pi^{1/2}) e^{-d/2} \cdot d^{d/2} \cdot d^{1/2}}$$

$$= \left( \frac{\pi e}{2d} \right)^{d/2} \cdot \frac{1}{(\pi d)^{d/2}}$$

Therefore, as  $d \rightarrow \infty$ ,  $\frac{1}{d} \rightarrow 0$

This gives :

$$\left. \begin{aligned} \frac{V_d a}{(V_a)_{\text{cube}}} &= 0 \times 0 = 0 \end{aligned} \right\} \text{Hence, proved}$$

Next, we need to show that the ratio of the distance from the center of the hypercube to one of the corners, divided by the perpendicular distance to one of the edges is  $\sqrt{d}$  and thus goes to  $\infty$ , as  $d \rightarrow \infty$ .

We know that the length of diagonal of  $d$ -dimensional hypercube with side ' $2a$ ' is given as:

$$l_c = \left( \sum_{i=1}^d (2a)^2 \right)^{1/2} \quad \text{--- (13)}$$

Therefore, length from center to one of the corners is  $\frac{l_c}{2} = \frac{2a\sqrt{d}}{2}$

Also, the perpendicular to one of the edges will have length half of that of side.

$$\therefore l_p = a \quad \text{--- (14)}$$

$$\begin{aligned} \text{Thus, } \frac{l_c/2}{l_p} &= \frac{(2a\sqrt{d})/2}{a} \\ &= \sqrt{d} \end{aligned}$$

Hence, as  $d \rightarrow \infty$ , this ratio goes to  $\infty$ .

1.3

In this question, using (8), we need to prove that the fraction of the volume of the sphere which lies at values of the radius between  $a-\epsilon$  and  $a$ , where  $0 < \epsilon < a$ , is given as:

$$f = 1 - \left(1 - \frac{\epsilon}{a}\right)^d$$

We can write the fraction of volume of the sphere lying between spheres of radii  $a-\epsilon$  and  $a$  as:

$$\frac{V_{da} - V_{d(a-\epsilon)}}{V_{da}} \quad \text{--- (15)}$$

Therefore,

$$\begin{aligned} f &= \frac{S_d \cdot \frac{a^d}{d} - S_d \frac{(a-\epsilon)^d}{d}}{\frac{S_d \cdot a^d}{d}} \\ &= \frac{a^d - (a-\epsilon)^d}{a^d} \\ &= 1 - \left(\frac{a-\epsilon}{a}\right)^d \end{aligned}$$

$$\Rightarrow \boxed{f = 1 - \left(1 - \frac{\epsilon}{a}\right)^d} \quad \text{Hence, proved.}$$

Now, we need to show that for any  $\epsilon$  no matter how small, this fraction tends to 1

as  $d \rightarrow \infty$

We know that

$$0 < \epsilon < a \quad (\text{given})$$

$$\Rightarrow 0 < \frac{\epsilon}{a} < 1$$

$$\Rightarrow 0 < 1 - \frac{\epsilon}{a} < 1$$

Hence, as  $d \rightarrow \infty$ ,  $(1 - \frac{\epsilon}{a})^d \rightarrow 0$

Therefore  $f_{d \rightarrow \infty} = 1 - 0$   
 $= 1$  Hence, proved.

Now, for the next part, we have:

$$\frac{\epsilon}{a} = 0.01$$

$$\Rightarrow 1 - \frac{\epsilon}{a} = 0.99$$

$$\therefore f = 1 - (0.99)^d$$

$$\Rightarrow f_{d=2} = 1 - (0.99)^2 \\ = 0.0199$$

$$\Rightarrow f_{d=10} = 1 - (0.99)^{10} \\ = 0.0956$$

$$\Rightarrow f_{d=1000} = 1 - (0.99)^{1000} \\ = 0.949957$$

Next, we need to evaluate the fraction of the volume of the sphere which lies inside

The radius  $\frac{a}{2}$  for  $d=2, 10, 1000$ .

Using ⑧, we write the ratio  $R$  as

$$R = \frac{S_d \cdot \left(\frac{a}{2}\right)^d}{\frac{S_d \cdot (a)^d}{d}} = \frac{a^d}{2^d \cdot a^d} = \frac{1}{2^d}$$

$$\Rightarrow R_{d=2} = \frac{1}{g^2} = 0.25$$

$$R_{d=10} = \frac{1}{g^{10}} = 0.000976$$

$$R_{d=1000} = \frac{1}{g^{1000}} \approx 0$$

Thus, we infer that almost no point is inside the radius  $\frac{a}{2}$  as  $d$  increases.



1.4 In this question, we are given a probability density function  $f(x)$  in  $d$ -dimensions, which is a function of radius  $r = \|x\|$  and which has a Gaussian form

$$f(x) = \frac{1}{(2\pi\sigma^2)^{d/2}} \exp\left(-\frac{\|x\|^2}{2\sigma^2}\right) \quad (16)$$

We need to prove that by changing variables from Cartesian to polar coordinates, the probability mass inside a thin shell of radius  $r$  and thickness  $\epsilon$  is given as  $p(r)\epsilon$  where

$$p(r) = \frac{S_d r^{d-1}}{(2\pi\sigma^2)^{d/2}} \exp\left(-\frac{r^2}{2\sigma^2}\right) \quad (17)$$

Then, we need to show that the function  $p(r)$  has a single maximum which, for large values of  $d$ , is located at  $\hat{r} \approx \sqrt{d}\sigma$ . Finally, by considering  $p(\hat{r} + \epsilon)$  where  $\epsilon \ll \hat{r}$  we need to show that for large  $d$

$$p(\hat{r} + \epsilon) = p(\hat{r}) \exp\left(-\frac{\epsilon^2}{\sigma^2}\right) \quad (18)$$

We have,  $r = \|x\| = \left( \sum_{i=1}^d x_i^2 \right)^{1/2}$

Therefore,

$$p(x) = \frac{1}{(2\pi\sigma^2)^{d/2}} \exp\left(-\frac{\|x\|^2}{2\sigma^2}\right)$$

$$= \frac{1}{(2\pi\sigma^2)^{d/2}} \exp\left(-\frac{\sum_{i=1}^d x_i^2}{2\sigma^2}\right)$$

Now, probability mass inside a shell of radius  $r$   
and thickness  $\epsilon$  is  $= \int_{\|x\|=r}^{\|x\|=r+\epsilon} p(\vec{x}) d\vec{x}$

$$= \iiint_{\|x\|=r}^{\|x\|=r+\epsilon} p(x_1, x_2, \dots, x_d) dx_1 dx_2 \dots dx_d$$

$$= \frac{1}{(2\pi\sigma^2)^{d/2}} \iiint_{\|x\|=r}^{\|x\|=r+\epsilon} e^{-x_1^2/2\sigma^2} \cdot e^{-x_2^2/2\sigma^2} \cdots e^{-x_d^2/2\sigma^2} dx_1 \dots dx_d$$

$$= \frac{1}{(2\pi\sigma^2)^{d/2}} \prod_{i=1}^d \int_{\|x\|=r}^{\|x\|=r+\epsilon} (2\sigma^2) e^{-x_i^2/2\sigma^2} \cdot \frac{dx_i}{2\sigma^2}$$

$$= \frac{(2\sigma^2)^d}{(2\pi\sigma^2)^{d/2}} \cdot S_d \cdot \int_r^{r+\epsilon} e^{-r^2/2\sigma^2} \left(\frac{r}{2\sigma^2}\right)^{d-1} \cdot \frac{dr}{2\sigma^2}$$

(From ② in 1.1)

$$= \frac{1}{(2\pi\sigma^2)^{d/2}} \cdot S_d \int_r^{r+\epsilon} e^{-r^2/2\sigma^2} \cdot r^{d-1} dr$$

Since  $dr \approx \epsilon$ , we can ignore the integral

$$\approx \frac{1}{(2\pi\sigma^2)^{d/2}} \cdot r^{d-1} \cdot e^{-r^2/2\sigma^2} \cdot S_d dr$$

$$= \frac{S_d r^{d-1} e^{-r^2/2\sigma^2}}{(2\pi\sigma^2)^{d/2}} \cdot \epsilon = p(r) \epsilon$$

Hence, proved.

For the second part, we differentiate (17) w.r.t.  $r$   
partially

$$\begin{aligned}\frac{\delta p(r)}{\delta r} &= \left[ \frac{(d-1)r^{d-2}}{(2\pi\sigma^2)^{d/2}} S_d - \frac{2r \cdot S_d \cdot r^{d-1}}{2\sigma^2 (2\pi\sigma^2)^{d/2}} \right] e^{-r^2/2\sigma^2} \\ &= 0 \quad (\text{for optimum value}) \\ \Rightarrow \left[ (d-1)r^{d-2} - \frac{r^d}{\sigma^2} \right] \cdot \frac{S_d}{(2\pi\sigma^2)^{d/2}} e^{-r^2/2\sigma^2} &= 0\end{aligned}$$

The second term cannot be zero.

$$\Rightarrow (d-1)r^{d-2} - \frac{r^d}{\sigma^2} = 0$$

$$\Rightarrow r^{d-2} ((d-1)\sigma^2 - r^2) = 0$$

$$\Rightarrow r^2 = \sigma^2 (d-1)$$

$$\Rightarrow r \approx \sqrt{d} \cdot \sigma \quad \text{for large } d \quad \text{--- (19)}$$

for  $r < \sqrt{d} \cdot \sigma$ , we have

$$\text{sign} \left( \frac{\delta p(r)}{\delta r} \right) = \frac{|d\sigma^2 - r^2|}{d\sigma^2 - r^2} \rightarrow +ve \quad \text{--- (20)}$$

for  $r > \sqrt{d} \cdot \sigma$ , we have

$$\text{sign} \left( \frac{\delta p(r)}{\delta r} \right) = -ve \quad \text{--- (21)}$$

from (19), (20) and (21), we can say that  $\sqrt{d} \cdot \sigma$  is a maxima and is unique.

Hence, proved.



Now, for the final part, we have

$$p(\hat{r} + \epsilon) = \frac{S_d (\hat{r} + \epsilon)^{d-1}}{(2\pi\sigma^2)^{d/2}} \cdot \exp \left\{ -\frac{(\hat{r} + \epsilon)^2}{2\sigma^2} \right\}$$

Therefore,

$$\begin{aligned} \frac{p(\hat{r} + \epsilon)}{p(\hat{r})} &= \frac{(\hat{r} + \epsilon)^{d-1}}{(\hat{r})^{d-1}} \cdot \frac{\exp \left\{ -\frac{(\hat{r} + \epsilon)^2}{2\sigma^2} \right\}}{\exp \left\{ -\frac{\hat{r}^2}{2\sigma^2} \right\}} \\ &= \left[ 1 + \frac{\epsilon}{\hat{r}} \right]^{d-1} \exp \left\{ \frac{\hat{r}^2}{2\sigma^2} - \frac{(\hat{r} + \epsilon)^2}{2\sigma^2} \right\} \\ &= \exp \left\{ \ln \left( 1 + \frac{\epsilon}{\hat{r}} \right)^{d-1} \right\} \exp \left\{ \frac{1}{2\sigma^2} (\hat{r}^2 - (\hat{r}^2 + \epsilon^2 + 2\hat{r}\epsilon)) \right\} \\ &= \exp \left\{ (d-1) \ln \left( 1 + \frac{\epsilon}{\hat{r}} \right) - \frac{\epsilon^2}{2\sigma^2} - \frac{\epsilon \hat{r}}{\sigma^2} \right\} \end{aligned} \quad (22)$$

Now, as  $d \rightarrow \infty$ , the higher order terms of (22)'s Taylor Series go to zero. This is because

$\hat{r} \propto \sqrt{d}$ .

Taking first two terms from Taylor series, we get:

$$\frac{p(\hat{r} + \epsilon)}{p(\hat{r})} = \exp \left\{ (d-1) \left[ \frac{\epsilon}{\hat{r}} - \frac{\epsilon^2}{2\hat{r}^2} \right] - \frac{\epsilon^2}{2\sigma^2} - \frac{\epsilon \hat{r}}{\sigma^2} \right\}$$

Substituting  $\hat{r}$  with  $\sigma\sqrt{d-1}$ ,

$$\begin{aligned} \frac{p(\hat{r} + \epsilon)}{p(\hat{r})} &= \exp \left\{ \frac{\epsilon\sqrt{d-1}}{\sigma} - \frac{\epsilon^2}{2\sigma^2} - \frac{\epsilon^2}{2\sigma^2} - \frac{\epsilon\sqrt{d-1}}{\sigma} \right\} \\ &= \exp \left\{ -\frac{\epsilon^2}{\sigma^2} \right\}. \end{aligned}$$

Hence, proved.

---

# Analysis on Logistic and Softmax Regression Using MNIST Dataset

---

**Chetan Gandontra**  
UC San Diego  
9500 Gilman Drive  
cgandontr@ucsd.edu

**Rishabh Misra**  
UC San Diego  
9500 Gilman Drive  
r1misra@ucsd.edu

## Abstract

1 This report discusses the first programming assignment of course CSE 253: Neural  
2 Networks and Pattern Recognition, its solutions and the inferences. MNIST dataset  
3 was used and the hand-written digits in it were classified using Logistic and Softmax  
4 Regressions. Under Logistic Regression, two-way classification was performed  
5 on specific digits (2's and 3's, 2's and 8's). An accuracy of more than 97% was  
6 achieved on both of these subsets of data using Logistic Regression. For Softmax  
7 Regression, we performed a ten-way classification (for all digits from 0 to 9) and  
8 achieved an accuracy of 87.65% on the test set.

## 9 1 Derivation of Gradient for Logistic Regression

### 10 1.1 Introduction

The problem statement here is to find the gradient of the cost function. The error function for the logistic regression follows from the negative log likelihood, which can be written as:

$$E(w) = - \sum_{n=1}^N \{t^n \ln y^n + (1 - t^n) \ln(1 - y^n)\}$$

### 11 1.2 Methodology

In this section, we will derive the gradient of cost function, which will be used in the later parts of this report. To find the optimal weight parameters, we need to take the partial derivative of the error function with respect to  $w_j$  as follows:

$$\frac{\partial E(w)}{\partial w_j} = - \sum_{n=1}^N \left[ t^n \frac{\partial \ln y^n}{\partial w_j} + (1 - t^n) \frac{\partial \ln(1 - y^n)}{\partial w_j} \right]$$

$$\frac{\partial E(w)}{\partial w_j} = - \sum_{n=1}^N \left[ \frac{t^n}{y^n} \frac{\partial y^n}{\partial w_j} + \frac{(1 - t^n)}{1 - y^n} \frac{\partial(1 - y^n)}{\partial w_j} \right]$$

Since  $y^n = \sigma(\mathbf{w} \cdot \mathbf{x}^n)$ , the derivative can be written as:

$$\frac{\partial E(w)}{\partial w_j} = - \sum_{n=1}^N \left[ \frac{t^n}{\sigma(\mathbf{w} \cdot \mathbf{x}^n)} \frac{\partial \sigma(\mathbf{w} \cdot \mathbf{x}^n)}{\partial w_j} + \frac{(1 - t^n)}{1 - \sigma(\mathbf{w} \cdot \mathbf{x}^n)} \frac{\partial(1 - \sigma(\mathbf{w} \cdot \mathbf{x}^n))}{\partial w_j} \right]$$

<sup>12</sup> Using the following properties of sigmoid function

$$\sigma(-\mathbf{x}) = 1 - \sigma(\mathbf{x}) \quad (1)$$

$$\frac{\partial \sigma(-\mathbf{x})}{\partial x} = \sigma(\mathbf{x})\sigma(-\mathbf{x}) \quad (2)$$

the derivative can be written as:

$$\frac{\partial E(w)}{\partial w_j} = - \sum_{n=1}^N \left[ \frac{t^n}{\sigma(\mathbf{w} \cdot \mathbf{x}^n)} \sigma(\mathbf{w} \cdot \mathbf{x}^n) \sigma(-\mathbf{w} \cdot \mathbf{x}^n) x_j^n - \frac{(1-t^n)}{\sigma(-\mathbf{w} \cdot \mathbf{x}^n)} \sigma(-\mathbf{w} \cdot \mathbf{x}^n) \sigma(\mathbf{w} \cdot \mathbf{x}^n) x_j^n \right]$$

$$\frac{\partial E(w)}{\partial w_j} = - \sum_{n=1}^N x_j^n [t^n \sigma(-\mathbf{w} \cdot \mathbf{x}^n) - (1-t^n) \sigma(\mathbf{w} \cdot \mathbf{x}^n)]$$

Using (1) we get,

$$\frac{\partial E(w)}{\partial w_j} = - \sum_{n=1}^N x_j^n [t^n (1 - \sigma(\mathbf{w} \cdot \mathbf{x}^n)) - (1-t^n) \sigma(\mathbf{w} \cdot \mathbf{x}^n)]$$

Solving the above equation, we get:

$$\frac{\partial E(w)}{\partial w_j} = - \sum_{n=1}^N x_j^n [t^n - \sigma(\mathbf{w} \cdot \mathbf{x}^n)]$$

or

$$-\frac{\partial E(w)}{\partial w_j} = \sum_{n=1}^N (t^n - y^n) x_j^n$$

### <sup>13</sup> 1.3 Results

Hence, from the above derivation, it follows that for  $n^{th}$  sample, the gradient of error can be written as:

$$-\frac{\partial E^n(w)}{\partial w_j} = (t^n - y^n) x_j^n$$

### <sup>14</sup> 1.4 Discussion

<sup>15</sup> The expression takes the difference between true label and predicted label and weigh it by the input  
<sup>16</sup> data value. It makes sense because if there is a stark difference between the true and the predicted  
<sup>17</sup> labels, the gradient value would be large. Thus, the corresponding component of the weight vector  
<sup>18</sup> would be adjusted quickly in the direction of gradient to reduce the loss.

19 **2 Derivation of Gradient for Softmax Regression**

20 **2.1 Introduction**

In this section, the focus is to find the gradient of the loss function of Softmax Regression -  $E(w)$ . The error function for the softmax regression follows from the negative log likelihood, which can be written as:

$$E(w) = - \sum_{n=1}^N \sum_{k'=1}^C t_{k'}^n \ln y_{k'}^n$$

21 **2.2 Methodology**

To find the optimal weight parameters for each class, we need to take the partial derivative of the error function with respect to  $w_{jk}$  as follows:

$$\begin{aligned} \frac{\partial E(w)}{\partial w_{jk}} &= - \sum_{n=1}^N \sum_{k'=1}^C \left[ t_{k'}^n \frac{\partial \ln y_{k'}^n}{\partial w_{jk}} \right] \\ - \frac{\partial E(w)}{\partial w_{jk}} &= \sum_{n=1}^N \left[ \frac{t_k^n}{y_k^n} \frac{\partial y_k^n}{\partial w_{jk}} \right] + \sum_{k' \neq k} \left[ \frac{t_{k'}^n}{y_{k'}^n} \frac{\partial y_{k'}^n}{\partial w_{jk}} \right] \end{aligned} \quad (3)$$

Since,

$$y_k^n = \frac{e^{\mathbf{w}_k \cdot \mathbf{x}^n}}{\sum_{l=1}^C e^{\mathbf{w}_l \cdot \mathbf{x}^n}}$$

22 the derivatives  $\frac{\partial y_k^n}{\partial w_{jk}}$  and  $\frac{\partial y_{k'}^n}{\partial w_{jk}}$  can be written as:

$$\begin{aligned} \frac{\partial y_k^n}{\partial w_{jk}} &= \frac{e^{\mathbf{w}_k \cdot \mathbf{x}^n}}{\sum_{l=1}^C e^{\mathbf{w}_l \cdot \mathbf{x}^n}} x_j^n - e^{\mathbf{w}_k \cdot \mathbf{x}^n} \left[ \frac{1}{(\sum_{l=1}^C e^{\mathbf{w}_l \cdot \mathbf{x}^n})^2} \right] e^{\mathbf{w}_k \cdot \mathbf{x}^n} x_j^n \\ \frac{\partial y_k^n}{\partial w_{jk}} &= (y_k^n - (y_k^n)^2) x_j^n \end{aligned} \quad (4)$$

$$\begin{aligned} \frac{\partial y_{k'}^n}{\partial w_{jk}} &= -e^{\mathbf{w}_{k'} \cdot \mathbf{x}^n} \left[ \frac{1}{(\sum_{l=1}^C e^{\mathbf{w}_l \cdot \mathbf{x}^n})^2} \right] e^{\mathbf{w}_{k'} \cdot \mathbf{x}^n} x_j^n \\ \frac{\partial y_{k'}^n}{\partial w_{jk}} &= - (y_{k'}^n)^2 x_j^n \end{aligned} \quad (5)$$

23 Substituting (4) and (5) in (3), we get

$$\begin{aligned} - \frac{\partial E(w)}{\partial w_{jk}} &= \sum_{n=1}^N \left[ \frac{t_k^n}{y_k^n} (y_k^n - (y_k^n)^2) x_j^n \right] - \sum_{k' \neq k} \left[ \frac{t_{k'}^n}{y_{k'}^n} (y_{k'}^n)^2 x_j^n \right] \\ - \frac{\partial E(w)}{\partial w_{jk}} &= \sum_{n=1}^N [t_k^n (1 - y_k^n) x_j^n] - \sum_{k' \neq k} [t_{k'}^n y_{k'}^n x_j^n] \end{aligned} \quad (6)$$

24 Now, for any sample, only one of the C labels in  $t^n$  would be 1, and all the others would be 0. This is  
25 because the label one would be the label set, and each training example can correspond to only one  
26 label. Thus, for any sample  $a$  where  $t_k^n$  is 1, the derivative would be:

$$-\frac{\partial E^a(w)}{\partial w_{jk}} = [(1 - y_k^a) x_j^a]$$

<sup>27</sup> or it could be written as:

$$-\frac{\partial E^a(w)}{\partial w_{jk}} = [(t_k^n - y_k^a) x_j^a] \quad (7)$$

For any sample  $b$  where one of  $t_{k'}^n$  is 1 (where  $k' \neq k$ ), the derivative would be:

$$-\frac{\partial E^b(w)}{\partial w_{jk}} = [(-y_k^b) x_j^b]$$

<sup>28</sup> or it could be written as:

$$-\frac{\partial E^b(w)}{\partial w_{jk}} = [(t_k^n - y_k^b) x_j^b] \quad (8)$$

<sup>29</sup> Using the results of (7) and (8), (6) could be written as:

$$-\frac{\partial E(w)}{\partial w_{jk}} = \sum_{n=1}^N (t_k^n - y_k^n) x_j^n$$

### <sup>30</sup> 2.3 Results

<sup>31</sup> Thus, using our findings above, we can say that for  $n^{th}$  sample, the derivative can be written as:

$$-\frac{\partial E^n(w)}{\partial w_{jk}} = (t_k^n - y_k^n) x_j^n \quad (9)$$

### <sup>32</sup> 2.4 Discussion

<sup>33</sup> Interestingly, the expression of gradient looks similar to that of logistic regression. In this case, the  
<sup>34</sup> derivative takes the difference between true label and predicted label for the  $k^{th}$  class and weigh it  
<sup>35</sup> by the input data value. Again, if the difference is big between the true and the predicted labels, the  
<sup>36</sup> gradient value would be large. Thus, the corresponding component of the weight vector would be  
<sup>37</sup> adjusted quickly in the direction of gradient to reduce the loss.

38 **3 Read in Data**

39 **3.1 Introduction**

40 As mentioned in the abstract, we deal with the "MNIST" dataset in this programming assignment.  
41 The first and foremost task before operating on the data was to load it.

42 **3.2 Methodology**

43 The MNIST data was downloaded from the website at <http://yann.lecun.com/exdb/mnist/>  
44 (the same link as given in PA1). To read the data, GitHub library at  
45 <https://github.com/akosiorek/CSE/blob/master/MLCV/> was used, which returns the training  
46 and testing data in matrix form, and labels as a vectors. Operations were then performed on this data  
47 to add a column of ones (bias term) at the beginning and to extract digit specific data (2-3's and  
48 2-8's). Also, the data was restricted to first 20k entries, 10% of which was allocated to a hold-out set.  
49 The size of test data was kept as two thousand. This was done by picking the first 2k entries from the  
50 test data returned by the library.

51 **3.3 Results**

52 Using some existing libraries on Github, we were able to extract the data into variables in Python.  
53 This data, however, consisted of the full 60k training data points and 10k testing data points. We  
54 extracted the first 20k training data points and the first 2k testing data points. 10% of the training data  
55 was designated as a hold-out set.

56 **3.4 Discussion**

57 Computation on large data sets can often be time consuming. Due to this reason, we extracted the  
58 full data and restricted the size of training, validation and test sets. This allowed faster computations  
59 throughout the programming assignment. A hold-out set acts as a dummy test set, which we use so as  
60 to improve performance of our model by testing it on the hold-out set. Good accuracy on hold-out set  
61 leads to a good accuracy on test set in general.

62 **4 Logistic Regression via gradient descent**

63 **4.1 Introduction**

64 In this part, we are required to use logistic regression and classify a given hand-written digit as either  
65 2 or 3. Since logistic regression uses binary output, we say that the target is 1 if the input is from the  
66 "2" category and 0 if it is from the other category. We are required to produce the following:

- 67 1. Plot of loss function ( $E$ ) over the training set, test set and the hold-out set
- 68 2. Plot of percent correct classification over training for the training set, the hold-out set, and  
69 the test set
- 70 3. The above two plots for digits 2 and 8
- 71 4. Display weights as images for both the classifiers (2 vs. 3 and 2 vs. 8). Plot the difference  
72 between weights as well.

73 **4.2 Methodology**

To plot the first graph, loss function is put against the Y-axis, and the iteration number along the X-axis. The loss function used was:

$$E(w) = - \sum_{n=1}^N \{t^n \ln y^n + (1 - t^n) \ln(1 - y^n)\}$$

74 This was done for all the three sets - training, test and validation. Hence, for each set, the value of  $N$   
75 and the corresponding data/labels change. This procedure was repeated for the 2's and 8's data set.

76 In the next graph, we plotted "percent correct" against the iteration number. The value of percent  
77 correct can be inferred by going over all the examples as test set in a way, and seeing what our model  
78 predicts on it. For every correct classification, we add one to the number of data points classified  
79 correctly. Then, at the end, we find the corresponding percentage. This is repeated at each iteration  
80 for all the three sets - training, test and validation. Again, this procedure was repeated for the 2's and  
81 8's data set.

82 To display the weight vectors for both the cases and their difference, the bias term in them was  
83 dropped. This reduced the weight vector to a 784 dimensional vector, which was re-shaped into a 28  
84 x 28 matrix and then plotted using Python.

### 85 4.3 Results

86 Various results are plotted below:

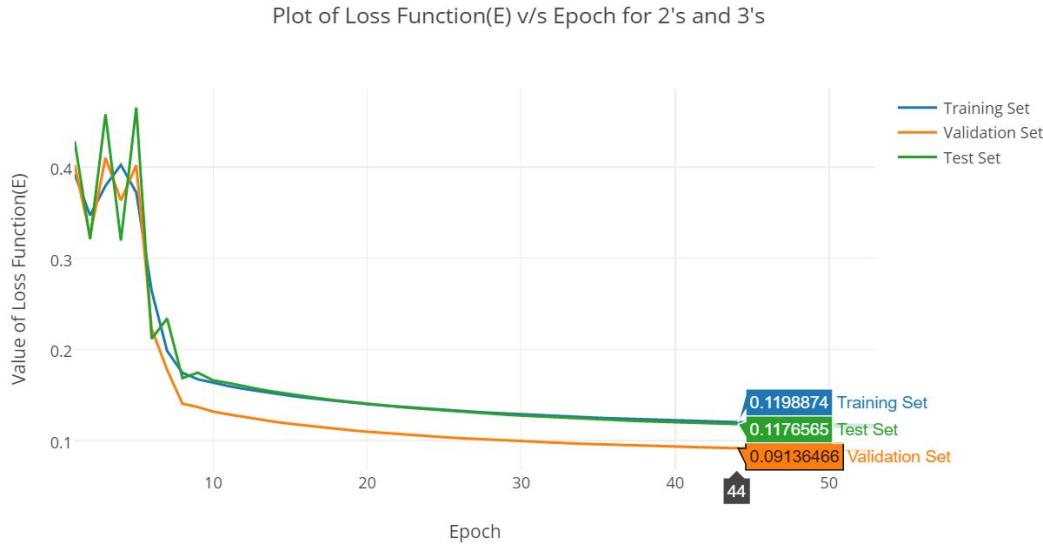


Figure 1: Loss Function (E) vs Epoch for 2's and 3's



Figure 2: Loss Function calculated every 1/20 of Epoch vs Epoch for 2's and 3's

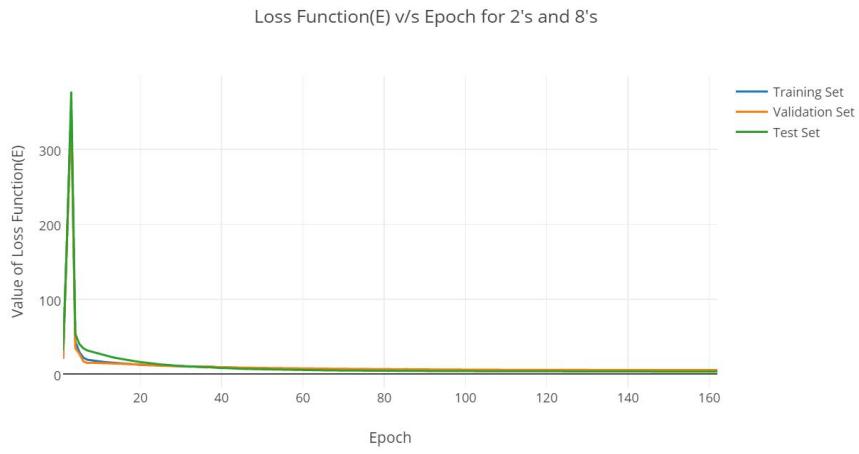


Figure 3: Loss Function (E) vs Epoch for 2's and 8's

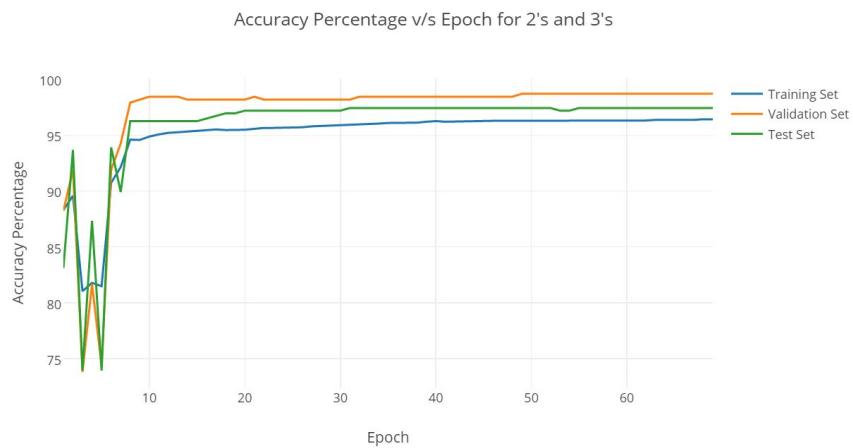


Figure 4: Percent correct classification vs Epoch for 2's and 3's

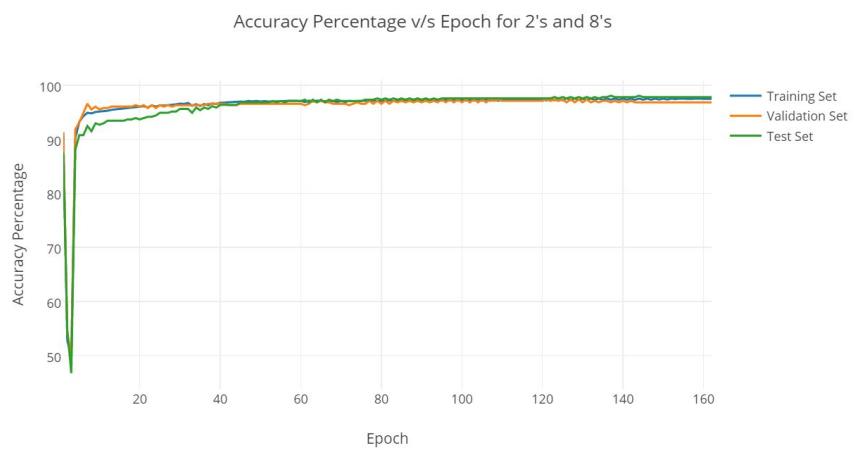


Figure 5: Percent correct classification vs Epoch for 2's and 8's

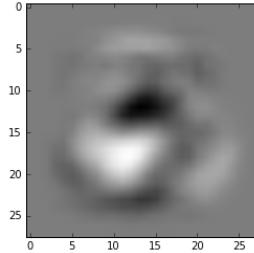


Figure 6: Weights as an image for 2's and 3's

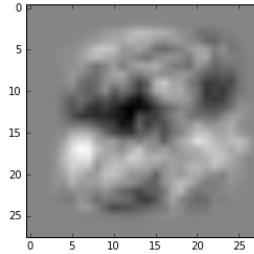


Figure 7: Weights as an image for 2's and 8's

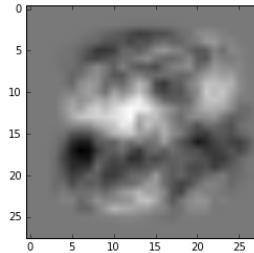


Figure 8: Difference of weights as an image for 2's and 3's and 2's and 8's

#### 87 4.4 Discussion

88 Generally, we assume that the data for training and testing are generated from same underlying  
 89 distribution. Thus, they have similar underlying properties. However, since test data is not available to  
 90 us in the real world, we extract a portion of training data to test our model and call it a validation set.  
 91 And since the validation set and the test set are generated from the same underlying distribution, the  
 92 performance of our model should not vary much on validation and test set in ideal case (it would very  
 93 a little bit in our case as validation set is small as compared to test set). The aforementioned behavior  
 94 is reaffirmed by our experiments. From figure 1 and figure 2, we see that curves for validation error  
 95 and test error are similar and validation error hovers around the test error. Same is the case for  
 96 classification of 2's and 8's in figure 3. Thus, we can safely say that validation set closely captures  
 97 how our model would perform on test data. One interesting thing to note is that in the case of batching  
 98 (contained 1/20 of all points), the curve of error function is smooth . This is because the gradient  
 99 is calculated only on few points at a time, which makes the gradient increase gradually towards the  
 100 optimum.

101 We have chosen the value of the hyperparameter T as 2000 empirically. Also, for the early stopping  
 102 of our algorithm, we have checked whether the error is non-decreasing on 15 iterations. The reason  
 103 for choosing a bit large value is that the error on the validation set used to change in the steps of 5-10  
 104 iterations. Thus, just to insure that the error does not decrease after early stopping, we took added  
 105 some buffer iterations.

106 In figure 4 and figure 5, we see that the accuracy percentage increases over time. This is expected, as  
 107 with more number of iterations, we minimize loss and fit the data. This means that we will be able

108 to classify more number of points correctly leading to higher accuracy. Also, note that the graph of  
109 2's and 8's is more overlapping. This can be because the validation/test data is very similar to the  
110 training data.

111 Next, we plot the weight vectors for three cases - 2's and 3's, 2's and 8's, and the difference of these  
112 two in figure 6, 7 and8 respectively. For both 2's and 3's case and the 2's and 8's case, the weight  
113 vector seems like images of 2 and 3/8 have been superimposed. This seems intuitively correct as  
114 the weight vector needs to predict either of these. For the difference case, the image looks like a  
115 mirror-image of 3. This is because the 2's component is common in both the cases and must get  
116 cancelled out. Thereafter, we are left with 8's and 3's, in which superimposition cancels out too.  
117 Hence, what we are left with is the portion of 8 that was not covered by 3 and thus looks like a mirror  
118 image of 3.

119 **5 Regularization**

120 **5.1 Introduction**

121 Regularization is a commonly used technique to improve the model generalization. We write the  
122 regularized loss function  $J(w)$  as:

$$J(w) = E(w) + \lambda C(w)$$

123 where  $C(w)$  is the complexity penalty and  $\lambda$  is the strength of regularization. For  $\lambda = 0$ ,  $J$  reduces to  
124  $E$ . Considering  $L_2$  norm as the complexity penalty, we have:

$$J(w) = E(w) + \lambda ||w||^2$$

125 For  $L_1$  norm, we have:

$$J(w) = E(w) + \lambda |w|$$

126 In the first part, we are expected to derive the update term for both  $L_1$  and  $L_2$  penalties.

127 Next, we are expected to plot the percent correct v/s iterations graph for different  $\lambda$  values. This is  
128 followed by plotting length of weight vector v/s iterations for different  $\lambda$  values. Then, we plot the  
129 final test error with each of the  $\lambda$ . Finally, we are expected to plot the weights as images.

130 **5.2 Methodology**

131 To derive the update term, we take derivative of this function with respect to  $w$ , the weight vector.  
132 Hence, we have:

$$\frac{\partial J(w)}{\partial w} = \frac{\partial E(w)}{\partial w} + \lambda \frac{\partial C(w)}{\partial w}$$

133 We have already calculate the first part of the equation -  $\frac{\partial E(w)}{\partial w}$  in Question 1. Hence, according to  
134 the question, solving for  $\frac{\partial C(w)}{\partial w}$ , we have:

$$\frac{\partial C(w)}{\partial w} = \frac{||w||^2}{\partial w} = 2w$$

Therefore, we have:

$$\frac{\partial J(w)}{\partial w} = \frac{\partial E(w)}{\partial w} + 2\lambda w$$

Similarly, for  $L_1$  norm as the complexity penalty, we have

$$J(w) = E(w) + \lambda |w|$$

Therefore,

$$\frac{\partial J(w)}{\partial w} = \frac{\partial E(w)}{\partial w} + \frac{\partial \lambda |w|}{\partial w}$$

$$\frac{\partial J(w)}{\partial w} = \frac{\partial E(w)}{\partial w} + \lambda \frac{\partial |w|}{\partial w}$$

135 Now, the entry at index  $j$  of partial derivative of  $|w|$  can be written as:

$$\frac{\partial |w|}{\partial w_j} = \begin{cases} 1, & \text{if } w_j \geq 0 \\ -1, & \text{otherwise} \end{cases}$$

136 Thus, the value of  $\frac{\partial|w|}{\partial w}$  is A vector of all one's or minus one's depending upon the sign of entries in  
 137 ( $w$ ) vector and has the same number of elements as in  $w$ .

138 To plot the graphs, a similar approach as in Section 4 was undertaken. The only difference was that  
 139 earlier we did it for different data sets - training, test and validation. In these graphs, we always take  
 140 the training set and calculate percent error and length of weight vector at that iteration. This is done  
 141 multiple times by changing  $\lambda$  values. Then, we plot the final test error for each  $\lambda$  value, keeping the  
 142 learning rate fixed. This plot is made as a bar graph, with one bar for each  $\lambda$ . Finally, we plot the  
 143 weights as images like we did in Section 4. To display the weight vectors, the bias term was dropped.  
 144 This reduced the weight vector to a 784 dimensional vector, which was re-shaped into a 28 x 28  
 145 matrix and then plotted using Python.

146 **5.3 Results**

The partial derivative of loss function with  $L_1$  norm regularization is:

$$\frac{\partial J(w)}{\partial w} = \frac{\partial E(w)}{\partial w} + \lambda \frac{\partial|w|}{\partial w}$$

147 where the partial derivative of  $|w|$  can be written as:

$$\frac{\partial|w|}{\partial w_j} = \begin{cases} 1, & \text{if } w_j \geq 0 \\ -1, & \text{otherwise} \end{cases}$$

148 and the partial derivative in case of  $L_2$  norm regularization is:

$$\frac{\partial J(w)}{\partial w} = \frac{\partial E(w)}{\partial w} + 2\lambda w$$

149 The "percent correct" value was plotted over the number of training iterations for the training set, for  
 150 different lambda values keeping the other hyper-parameters same.

151 Parameters used: **Penalty** =  $L_2$  norm, **Learning rate**  $\eta = 0.0001$

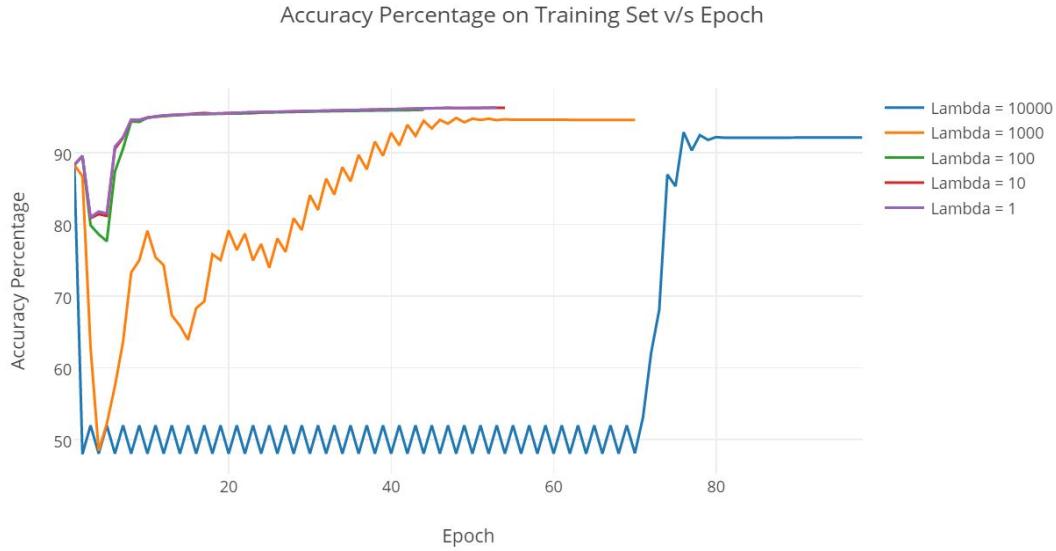


Figure 9: Accuracy Percentage v/s Epoch for  $L_2$  norm

152 Now for penalty as  $L_1$  norm and Learning rate  $\eta = 0.0001$ , we have:

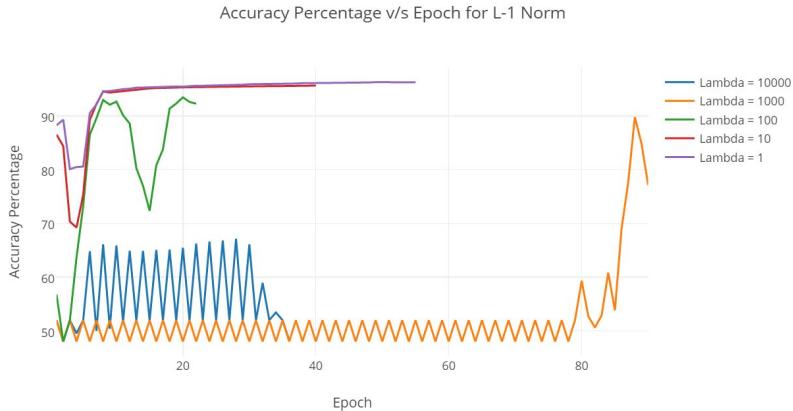


Figure 10: Accuracy Percentage v/s Epoch for  $L_1$  norm

- 153 The length of weight vector against training iterations produced a graph as follows for the  $L_2$  norm case:

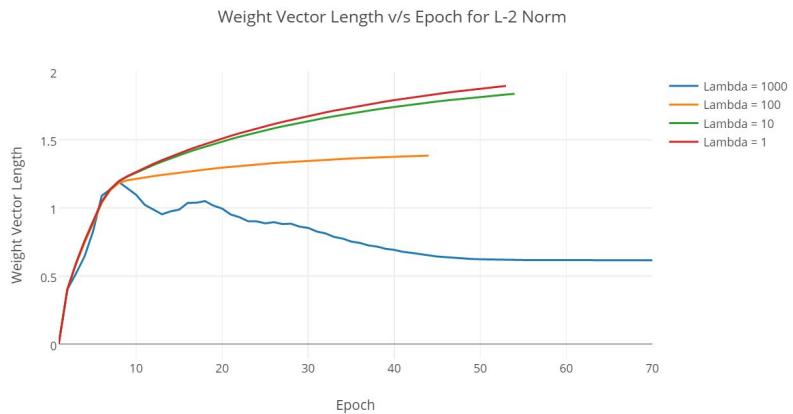


Figure 11: Weight Vector Length v/s Epoch with  $L_2$  norm

154

For  $L_1$  norm, it becomes:

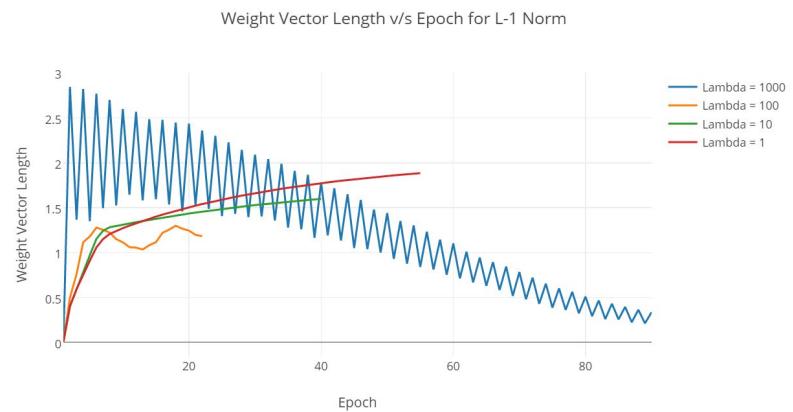


Figure 12: Weight Vector Length v/s Epoch with  $L_1$  norm

155

- 156 Note that the learning rate  $\eta$  used was 0.0001 in both the cases.
- 157 The plot of final test error for various  $\lambda$  values with  $L_2$  norm penalty is as follows:

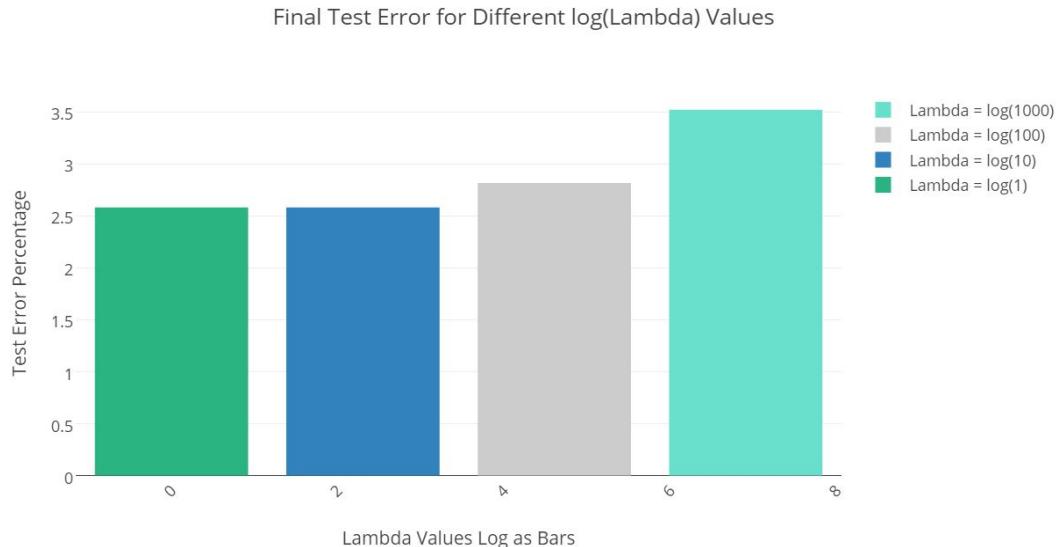


Figure 13: Final Test Error v/s Log( $\lambda$ )  $L_2$  norm

- 158 The plot of final test error for various  $\lambda$  values with  $L_1$  norm penalty is as follows:

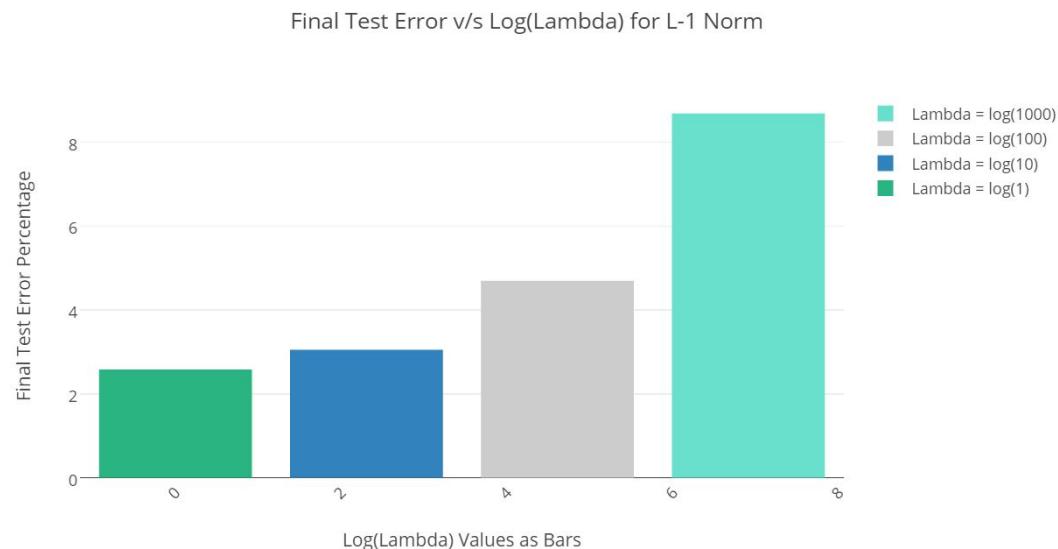


Figure 14: Final Test Error v/s Log( $\lambda$ )  $L_1$  norm

- 159 Note that the learning rate  $\eta$  used was 0.0001 in both the cases.
- 160 For **L1** norm: using learning rate  $\eta$  as 0.0001 in both the cases and different  $\lambda$  values, the final weight vectors were plotted as images. Here are the findings:

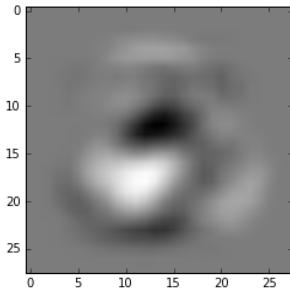


Figure 15: Weight Vector Image for 2's and 3's with  $L_1$  norm and  $\lambda = 1$

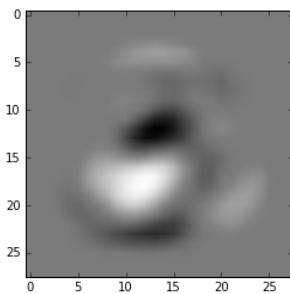


Figure 16: Weight Vector Image for 2's and 3's with  $L_1$  norm and  $\lambda = 10$

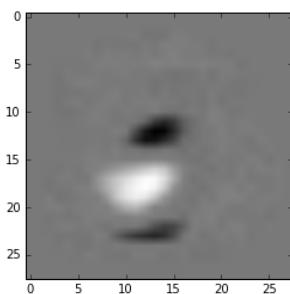


Figure 17: Weight Vector Image for 2's and 3's with  $L_1$  norm and  $\lambda = 100$

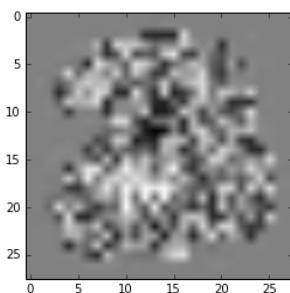


Figure 18: Weight Vector Image for 2's and 3's with  $L_1$  norm and  $\lambda = 1000$

<sup>162</sup> For **L1** norm: Using learning rate  $\eta$  as 0.0001 in both the cases and optimal  $\lambda$  value of 0.0001, the  
<sup>163</sup> image looks like following:

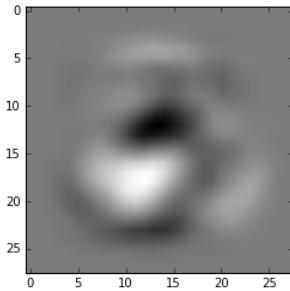


Figure 19: Weight Vector Image for 2's and 3's with  $L_1$  norm and  $\lambda = 0.0001$

164 For **L2** norm: using learning rate  $\eta$  as 0.0001 in both the cases and different  $\lambda$  values, the final weight  
165 vectors were plotted as images. Here are the findings:

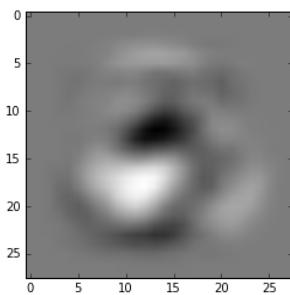


Figure 20: Weight Vector Image for 2's and 3's with  $L_2$  norm and  $\lambda = 1$

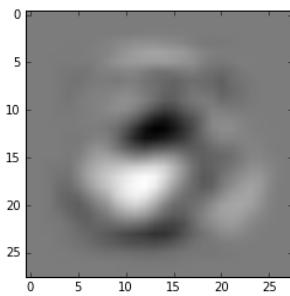


Figure 21: Weight Vector Image for 2's and 3's with  $L_2$  norm and  $\lambda = 10$

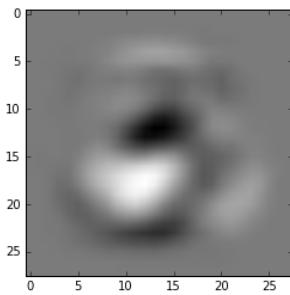


Figure 22: Weight Vector Image for 2's and 3's with  $L_2$  norm and  $\lambda = 100$

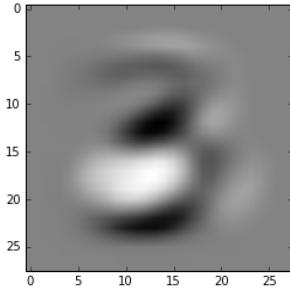


Figure 23: Weight Vector Image for 2's and 3's with  $L_2$  norm and  $\lambda = 1000$

166 For **L2** norm: Using learning rate  $\eta$  as 0.0001 in both the cases and optimal  $\lambda$  value of 0.0001, the  
167 image looks like following:

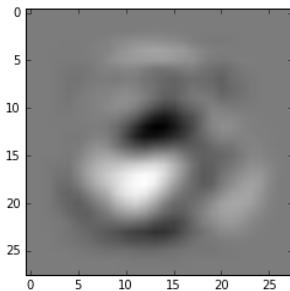


Figure 24: Weight Vector Image for 2's and 3's with  $L_2$  norm and  $\lambda = 0.0001$

168 **5.4 Discussion**

169 Note that the  $L_1$  norm is not differentiable at 0. However, all that matters is that we can compute  
170 a subgradient / subderivative. Since it's differentiable everywhere else, we can just fill in any  
171 "reasonable" value (such as -1 or 1; we have chosen -1) for the gradient at 0. So, for the cases where  
172 the weights are negative, we would have positive regularization term that would drive them towards  
173 zero and for the cases where the weights are positive, the regularization term would be negative, thus  
174 decreasing the weights. Hence,  $L_1$  tries to limit the coefficient which restricts overfitting.

175 Graph 9 shows how the accuracy on training data varies according to the iteration number (epoch) for  
176 several  $\lambda$  values.  $\lambda$  is added to avoid over-fitting. Thus, higher the  $\lambda$  value, lower will be the accuracy  
177 on training set as more emphasis is given to the complexity function rather than the original loss  
178 function  $E(w)$ .

179 Graph 10 shows how the accuracy on training data varies according to the iteration number (epoch)  
180 for several  $\lambda$  values. The same generalization as above holds here as well. Note that the graph line  
181 for  $\lambda = 10000$  overlaps with that of  $\lambda = 1000$  and is not visible clearly.

182 Next, we plot the length of weight vectors against training iterations for different  $\lambda$  values. In both the  
183 cases, as  $\lambda$  increases, the corresponding weight vector length value decreases. Both these values are  
184 inversely related. This is due because as  $\lambda$  increases, the overall regularized loss function value tends  
185 to increase. However, since our goal is to minimize loss, the weight vector balances this increase in  $\lambda$   
186 by decreasing itself. A similar argument is valid for decreasing  $\lambda$  values as well.

187 Then we have the plots of final test error for different  $\log(\lambda)$  values. In both the above graphs, as  
188  $\lambda$  (or  $\log(\lambda)$ ) increases, the final test error increases. This is because for large  $\lambda$  values, the model  
189 complexity is low. Beyond a certain level, the complexity may become so low that it no longer fits the  
190 data well leading to mis-classification on a large number of points. Increasing  $\lambda$  value only decreases  
191 the complexity further, leading to even further decline in test accuracy. Similarly, if the  $\lambda$  value is too  
192 low, the model may become highly complex, so much so that over-fitting happens. Test error in this  
193 case will again be large, as the model won't generalize well for points in test set.

<sup>194</sup> Finally, we have the weight vector images. Since, regularization limits the weights from being too  
<sup>195</sup> high, the images of weights are little bit softer in nature.

196 **6 Softmax Regression Via Gradient Descent**

197 **6.1 Introduction**

198 The task at hand is to perform Softmax Regression on the MNIST data set and come up with the best  
199 parameters that may perform well on the test data, without actually looking at test data. Then, we  
200 need to plot the loss function values over number of training iterations for training, hold-out and test  
201 data sets. Finally, we are expected to plot the percent correct values over training iterations the three  
202 data set parts.

203 **6.2 Methodology**

204 Softmax Regression was performed on the first 20,000 training data points and was used to do a  
205 10-way classification of the hand-written digits using a hold-out set and regularization parameter  
206  $\lambda$ . The size of hold-out set was again set to 10% of the size of training data. To figure out the best  
207 hyper-parameter values, a hold-out set was used. The parameters performing the best on this hold-out  
208 set were chosen to be the final parameters. The loss function( $E$ ) was plotted over the number of  
209 training iterations for the training, hold-out and test sets.

210 Parameters used: Penalty =  $L_2$  norm; Regularization parameter  $\lambda = 0.0001$ ; Learning rate  $\eta = 0.0001$

211 Next, the "percent correct" (or accuracy percentage) was plotted over the number of training iterations  
212 for the training, hold-out and test sets. Same parameters as above were used.

213 **6.3 Results**

214 The percentage error values recorded on the hold-out set for different values of hyper-parameters are  
215 as follows:

Table 1: Error on Hold-out Set for Different Hyper-parameters

$\eta$	$\lambda$	Norm	Error %
0.0001	0.01	$L_1/L_2$	8.1
0.0001	0.1	$L_1/L_2$	8.1
0.0001	0.0	$L_1/L_2$	8.1
0.001	0.0	$L_1/L_2$	8.15
0.01	0.0	$L_1/L_2$	8.2
0.1	0.0	$L_1/L_2$	88.2

216 All the graphs produced are plotted and reported herein.

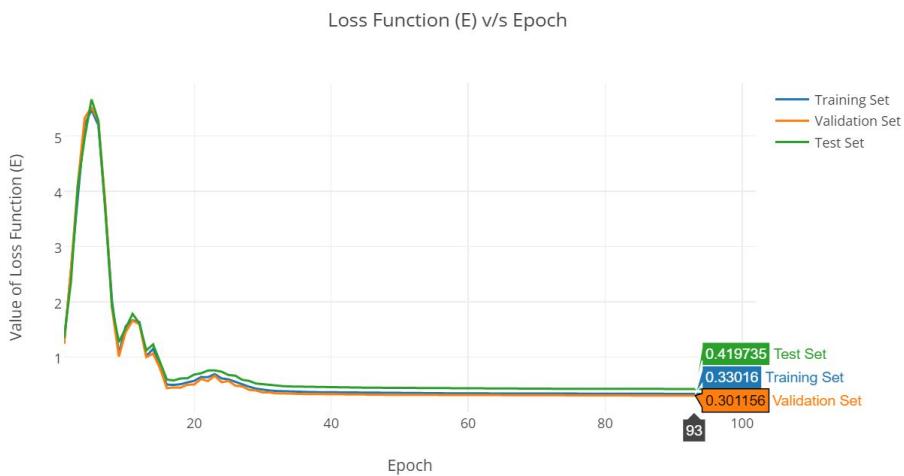


Figure 25: Loss Function(E) v/s Epoch

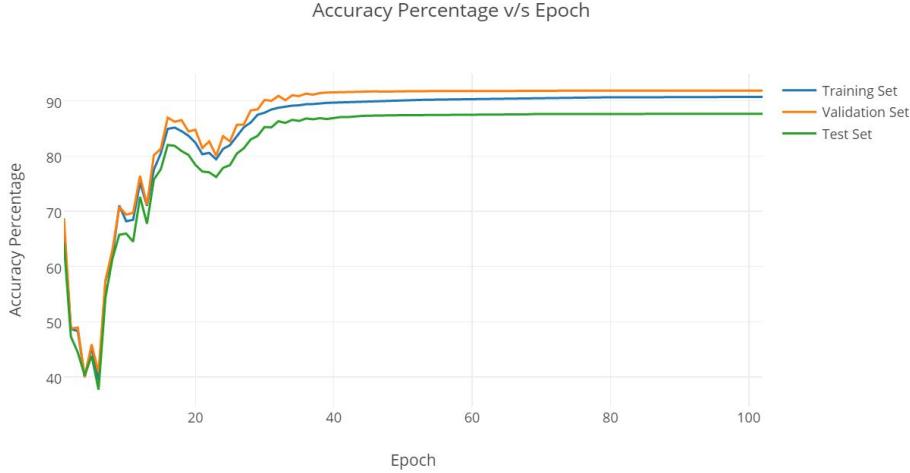


Figure 26: Accuracy Percentage v/s Epoch

#### 217 6.4 Discussion

218 As evident from the given table, the optimum value of error on hold-out (validation) set was obtained  
 219 at more than one pair of values of  $\eta$  and  $\lambda$ . We decided to go with the highest  $\eta$  and the largest  
 220 non-zero  $\lambda$  that gave us the optimum error value on hold-out set. This is because we wanted  $\lambda$  as  
 221 large as possible to help generalization and avoid over-fitting, and at the same time being a good  
 222 representative of the data. Also, higher learning rate was preferred for faster convergence. Since the  
 223 penalties at both  $L_2$  and  $L_1$  norm did not seem to affect the error values, we decided to go forward  
 224 with  $L_2$  norm, as it gives a better measure of loss and is convex everywhere.

225 Thus, the hyper-parameters chosen were": Penalty =  $L_2$  norm Regularization parameter  $\lambda = 0.1$   
 226 Learning rate  $\eta = 0.0001$

227 Validation Error = 8.1

228 Test Error obtained = 12.35

229 Note that early stopping was used to make sure that we do not over-fit the data.

230 Graph 25 shows how the training, validation and test errors vary according to the iteration number  
 231 (epoch) for the same values of hyper-parameters. As evident from the graph, the loss function  
 232 decreases and stabilizes over time, which corresponds to convergence. As an indicative measure, the  
 233 values of test, validation and training errors have been displayed for a particular value (93) of epoch.

234 Graph 26 shows how the training, validation and test percent correct values vary according to the  
 235 iteration number (epoch) for the same values of hyper-parameters. As evident from the graph, the  
 236 accuracy saturates over time and does not improve. The weights learned give the best performance  
 237 on validation set, followed by training and test set. It is interesting to note that all the values are fairly  
 238 close to each other and similar in shape, which means that the training set is a good representative of  
 239 the hold-out and test sets.

240 **7 Results and Learnings**

241 The best accuracy achieved using Logistic Regression was more than 97% for both the subsets of  
242 data - having digits 2 and 3, and having digits 2 and 8. The best learning rate was  $\eta = 0.0001$  and the  
243 best regularization parameter  $\lambda$  was 0.0001 for 2's and 3's, and 0.1 for 2's and 8's. Using Softmax  
244 Regression, our accuracy was around 87.65% on test data. The  $\lambda$  used in this case was 0.001 while  
245 the learning rate was kept to be 0.0001. Annealing of learning rate per iteration helped us avoid  
246 over-fitting of the data, even when number of iterations was huge. This annealing parameter was  
247 set to a convenient value (2000) to ensure a gradual yet sufficient decrease in learning rate. The  
248 early stopping margin for number of iterations was set to 15 for the both the cases to ensure that the  
249 algorithm was not stopping at some sub-optimal value, which was the case when this value was small.

250 Having taken courses like 250A and 250B, we knew the working and mechanism of Logistic and  
251 Softmax Regression, but we had never had a chance to perform their in-depth analysis by ourselves.  
252 Deriving the expressions and implementing these two algorithms along with regularization gave us a  
253 new insight and deep understanding of the working of these methods. Questions involving plotting  
254 of loss functions, weight vectors as images and weight vector lengths showed us how these values  
255 vary with different regularization parameters  $\lambda$  and the iteration number. We had never looked at  
256 the MNIST digit classification problem from this perspective and now have a clearer idea as to how  
257 the various hyper-parameters are related to each other. The impact of regularization on the model  
258 and weights is now in front of us, while previously we only looked at it theoretically. The concepts  
259 of annealing and early stopping were entirely new to us, as these concepts were never visible when  
260 using *scikit* for performing these computations.

261 **8 Individual Contributions**

262 Being roommates, it was extremely convenient and simple for both the authors to coordinate and  
263 work in sync, while ensuring equal distribution of work and time spent on the assignment. Whenever  
264 one of the authors got stuck at some point, the other was there to help him out and unblock instantly.  
265 The process was initially started with both of us sitting down and solving on a white board the various  
266 derivations involved. The work thereafter was taken up as under:

267 Chetan Gandotra implemented the part which involved reading of MNIST data and implemented  
268 Softmax Regression. Then, debugging and graph plotting of Logistic Regression was taken up by  
269 him.

270 Rishabh Misra took up the implementation of Logistic Regression after extracting digit specific data.  
271 Thereafter, debugging and graph plotting of Softmax Regression was taken up by him.

272 The implementation and graphs for regularization question (Q5) were divided equally, with 5 (b) and  
273 5 (c) being taken up by Chetan, and the others by Rishabh. For parameter tuning (values of  $\lambda$ ,  $\eta$ , T,  
274 early stopping iteration number etc.), we made an Excel sheet with possible list of values and divided  
275 them equally amongst us. We then ran the code for these values for parameters on our respective  
276 systems. When it came to writing the report, we took alternate question parts, with Rishabh taking  
277 odd questions and Chetan taking up even questions.

278 **References**

279 [1] <https://github.com/akosiorek/CSE/tree/master/MLCV>

280 **Appendix**

```
281 LoadMNIST.py
282 import os, struct
283 from array import array as pyarray
284 from numpy import append, array, int8, uint8, zeros
285
286 def load_mnist(dataset="training", digits=None, path=None, asbytes=False, selection=None, retu
287 """
288     Loads MNIST files into a 3D numpy array.
289
290     You have to download the data separately from [MNIST]_. It is recommended
291     to set the environment variable ``MNIST`` to point to the folder where you
292     put the data, so that you don't have to select path. On a Linux+bash setup,
293     this is done by adding the following to your ``.bashrc``::
294
295         export MNIST=/path/to/mnist
296
297     Parameters
298     -----
299     dataset : str
300         Either "training" or "testing", depending on which dataset you want to
301         load.
302     digits : list
303         Integer list of digits to load. The entire database is loaded if set to
304         ``None``. Default is ``None``.
305     path : str
306         Path to your MNIST datafiles. The default is ``None``, which will try
307         to take the path from your environment variable ``MNIST``. The data can
308         be downloaded from http://yann.lecun.com/exdb/mnist/.
309     asbytes : bool
310         If True, returns data as ``numpy.uint8`` in [0, 255] as opposed to
311         ``numpy.float64`` in [0.0, 1.0].
312     selection : slice
313         Using a 'slice' object, specify what subset of the dataset to load. An
314         example is ``slice(0, 20, 2)`` , which would load every other digit
315         until—but not including—the twentieth.
316     return_labels : bool
317         Specify whether or not labels should be returned. This is also a speed
318         performance if digits are not specified, since then the labels file
319         does not need to be read at all.
320     return_indices : bool
321         Specify whether or not to return the MNIST indices that were fetched.
322         This is valuable only if digits is specified, because in that case it
323         can be valuable to know how far
324         in the database it reached.
325
326     Returns
327     -----
328     images : ndarray
329         Image data of shape ``(N, rows, cols)``, where ``N`` is the number of images. If neither
330     labels : ndarray
331         Array of size ``N`` describing the labels. Returned only if ``return_labels`` is ``True``.
332     indices : ndarray
333         The indices in the database that were returned.
334
335     Examples
336     -----
337     Assuming that you have downloaded the MNIST database and set the
```

```

338     environment variable ``$MNIST`` point to the folder, this will load all
339     images and labels from the training set:
340
341     >>> images, labels = ag.io.load_mnist('training') # doctest: +SKIP
342
343     Load 100 sevens from the testing set:
344
345     >>> sevens = ag.io.load_mnist('testing', digits=[7], selection=slice(0, 100), return_labels=True)
346
347     """
348
349     # The files are assumed to have these names and should be found in 'path'
350     files = {
351         'training': ('train-images.idx3-ubyte', 'train-labels.idx1-ubyte'),
352         'testing': ('t10k-images.idx3-ubyte', 't10k-labels.idx1-ubyte'),
353     }
354
355     if path is None:
356         try:
357             path = 'C:\\\\Users\\\\Chetan\\\\Documents\\\\Python Scripts\\\\Way1'
358             #path = os.environ['MNIST']
359         except KeyError:
360             raise ValueError("Unspecified path requires environment variable $MNIST to be set")
361
362     try:
363         images_fname = os.path.join(path, files[dataset][0])
364         labels_fname = os.path.join(path, files[dataset][1])
365     except KeyError:
366         raise ValueError("Data set must be 'testing' or 'training'")
367
368     # We can skip the labels file only if digits aren't specified and labels aren't asked for
369     if return_labels or digits is not None:
370         flbl = open(labels_fname, 'rb')
371         magic_nr, size = struct.unpack(">II", flbl.read(8))
372         labels_raw = pyarray("b", flbl.read())
373         flbl.close()
374
375         fimg = open(images_fname, 'rb')
376         magic_nr, size, rows, cols = struct.unpack(">IIII", fimg.read(16))
377         images_raw = pyarray("B", fimg.read())
378         fimg.close()
379
380     if digits:
381         indices = [k for k in range(size) if labels_raw[k] in digits]
382     else:
383         indices = range(size)
384
385     if selection:
386         indices = indices[selection]
387     N = len(indices)
388
389     images = zeros((N, rows, cols), dtype=uint8)
390
391     if return_labels:
392         labels = zeros((N), dtype=int8)
393     for i, index in enumerate(indices):
394         images[i] = array(images_raw[ indices[i]*rows*cols : (indices[i]+1)*rows*cols ]).reshape((rows, cols))
395         if return_labels:
396             labels[i] = labels_raw[indices[i]]
397
398     if not asbytes:
399         images = images.astype(float)/255.0
400
401     ret = (images, )
402     if return_labels:

```

```

403         ret += (labels,)
404     if return_indices:
405         ret += (indices,)
406     if len(ret) == 1:
407         return ret[0] # Don't return a tuple of one
408     else:
409         return ret
410
411 Logistic_Regression_via_Gradient_Descent.py
412
413 """
414 CSE 253: Neural Networks and Pattern Recognition
415 Logistic Regression With and Without Gradient Descent
416 This file contains code for questions 4 and 5, including all graph plots
417 """
418 import numpy
419 import math
420 import matplotlib.pyplot as plt
421 import plotly.plotly as py1
422 import plotly.graph_objs as go
423
424 from LoadMNIST import load_mnist
425 #-----Utility functions-
426 def get_data(N, N_test):
427     #load MNIST data using libraries available
428     training_data, training_labels = load_mnist('training')
429     test_data, test_labels = load_mnist('testing')
430
431     training_data = flatArray(N, 784, training_data) #training_data is N x 784 matrix
432     training_labels = training_labels [:N]
433     test_data = flatArray(N_test, 784, test_data)
434     test_labels = test_labels [:N_test]
435
436     # adding column of 1s for bias
437     training_data = addOnesColAtStart(training_data)
438     test_data = addOnesColAtStart(test_data)
439
440     # Last 10% of training data size will be considered as the validation set
441     N_validation = int (N / 10)
442     validation_data = training_data[N-N_validation:N]
443     validation_labels = training_labels[N-N_validation:N]
444     N=N-N_validation
445     #update training data to remove validation data
446     training_data = training_data [:N]
447     training_labels = training_labels [:N]
448
449     return training_data, training_labels, test_data, test_labels, validation_data, validation_labels
450
451 def flatArray(rows, cols, twoDArr):
452     flattened_arr = numpy.zeros(shape=(rows, cols))
453     for row in range(0, rows):
454         i=0
455         for element in twoDArr[row]:
456             for ell in element:
457                 flattened_arr[row][i] = ell
458                 i = i+1
459     return flattened_arr
460
461 def addOnesColAtStart(matrix):
462     Ones = numpy.ones(len(matrix))
463     newMatrix = numpy.c_[Ones, matrix]
464     return newMatrix
465
466 # custom sigmoid function; if -x is too large, return value 0
467 def sigmoid(x):

```

```

468     if(-x < 709):
469         return 1 / (1 + math.exp(-x))
470     else:
471         return 1 / (1 + math.exp(708))
472
473 def extract_digit_specific_data(digits , data , label):
474     pruned_data = numpy.zeros(shape = (1, len(data[0])))
475     pruned_labels = []
476     cnt = 0
477     for i in range(0, len(label)):
478         if label[i] in digits:
479             if (cnt == 0):
480                 for j in range(len(data[0])):
481                     pruned_data[0][j] = data[i][j]
482             else:
483                 pruned_data = addRowToMatrix(pruned_data , data[i])
484             pruned_labels.append(digits.get(label[i]))
485             cnt = cnt + 1
486     return pruned_data , pruned_labels
487
488 def addRowToMatrix(matrix , row):
489     newMatrix = numpy.zeros(shape = ((len(matrix) + 1), len(matrix[0])))
490     for i in range(len(matrix)):
491         newMatrix[i] = matrix[i]
492     newMatrix[i+1] = row
493     return newMatrix
494
495 def calculate_log_liikelihood(data , label , weights , t):
496     log_likelihood = 0.0
497     for j in range(0,t):
498         #print(numpy.log(sigmoid(numpy.dot(weights , data[j]))))
499         log_likelihood += (label[j]*numpy.log(sigmoid(numpy.dot(weights , data[j])))) + ((1-label[j])*numpy.log(1-sigmoid(numpy.dot(weights , data[j]))))
500
501     return -1*log_likelihood/t
502
503 # 5.b, 5.c – Plot of Percent correct on training data v/s iterations ,
504 #length of weight vector v/s lambda
505 def plotlyGraphsRegularization(error_plot_array , lamda_vals , graph_name , min_index = -1):
506     pyl.sign_in('chetang' , 'vil7vTAuCSWt2lEZvaH9')
507
508     trace = []
509
510     for i in range(len(lamda_vals)):
511         y1 = error_plot_array[i]
512         #y1 = y1[:min_index[i]]
513         x1 = [j+1 for j in range(len(y1))]
514
515         trace1 = go.Scatter(
516             x=x1 ,
517             y=y1 ,
518             name = 'lambda = ' + (str)(lamda_vals[i]) , # Style name/legend entry with html tag
519             connectgaps=True
520         )
521
522         trace.append(trace1)
523     data = trace
524
525     fig = dict(data=data)
526     pyl.iplot(fig , filename=graph_name)
527
528 def plotlyGraphs(error_plot_array , labels , name):
529     pyl.sign_in('cgandotr' , '3c9fh04498')
530     trace = []
531
532     for i in range(len(labels)):

```

```

533     y1 = error_plot_array[i]
534     y1 = [k for k in y1]
535     x1 = [(j+1) for j in range(len(y1))]
536
537     trace1 = go.Scatter(
538         x=x1,
539         y=y1,
540         name = str(labels[i]), # Style name/legend entry with html tags
541         connectgaps=True
542     )
543     trace.append(trace1)
544 data = trace
545 fig = dict(data=data)
546 pyl.iplot(fig, filename=name)
547
548 def early_stopping(early_stopping_horizon, accuracy, i):
549     if i > early_stopping_horizon + 1:
550         counter = 0;
551         for p in range(0,early_stopping_horizon):
552             if accuracy[i-p] <= accuracy[i-p-1]:
553                 counter+=1
554             else:
555                 break
556
557         if(counter == early_stopping_horizon):
558             return True
559         else:
560             return False
561
562 def calculate_error(weights, data, label):
563     error = 0.0;
564     for j in range(0,len(data)):
565         prediction = sigmoid(numpy.dot(weights, data[j]))
566         if(prediction >0.5 and label[j]!=1):
567             error += 1;
568         elif (prediction <=0.5 and label[j]!=0):
569             error += 1;
570     return error;
571
572 def dropFirstColumn(weights):
573     return numpy.array(weights)[0][1:]
574
575 def fit(training_data, training_label, test_data, test_label, validation_data,
576         validation_label, digits, learning_rate=0.0001, iteration=200, batch_size=0, T=5000, l
577
578     t = len(training_data)
579     accuracy_plot_array = []
580     log_likelihood_array = []
581     weight_vector_length_array = []
582     accuracy_plot_training_array = []
583     weights_for_all_lamda = []
584     test_error_array = []
585     org_learning_rate = learning_rate
586
587     for lamda in lamda_vals:
588         weights = numpy.matrix(numpy.zeros(len(training_data[0])))
589         weights_array = []
590         weight_vector_length = []
591
592         accuracy_plot_training = []
593         accuracy_plot_validation = []
594         accuracy_plot_testing = []
595
596         log_likelihood_training = []
597         log_likelihood_validation = []

```

```

598 log_likelihood_testing = []
599 min_error_index = 0
600 learning_rate = org_learning_rate
601
602 for i in range(0, iteration):
603     # initialise Gradient
604     gradient = numpy.matrix(numpy.zeros(len(training_data[0])))
605     cnt = 0
606
607     norm_term = []
608     if (norm == 2):
609         norm_term = l2_norm(lamda, weights)
610     else:
611         norm_term = l1_norm(lamda, weights)
612
613     # calculate gradient over all the samples
614     for j in range(0,t):
615         # update gradient
616         gradient += ((training_label[j]) - sigmoid(numpy.dot(weights, training_data[j])))
617         cnt += 1
618         if (batch_size != 0 and cnt == (int)(t/batch_size)):
619             cnt = 0
620             weight_vector_length.append(numpy.linalg.norm(weights))
621             weights = weights + learning_rate * (gradient - norm_term)
622             # re-initialise Gradient
623             gradient = numpy.matrix(numpy.zeros(len(training_data[0])))
624             # calculating log likelihood of training, validation and test dataset
625             log_likelihood_training.append(calculate_log_liikelihood(training_data, train_label))
626             log_likelihood_validation.append(calculate_log_liikelihood(validation_data, validation_label))
627             log_likelihood_testing.append(calculate_log_liikelihood(test_data, test_label))
628
629     if (batch_size == 0):
630         weight_vector_length.append(numpy.linalg.norm(weights))
631         # update weights vector according to the update rule of Gradient descent method
632         weights = weights + learning_rate * (gradient - norm_term)
633
634         # calculating log likelihood of training, validation and test dataset
635         log_likelihood_training.append(calculate_log_liikelihood(training_data, training_label))
636         log_likelihood_validation.append(calculate_log_liikelihood(validation_data, validation_label))
637         log_likelihood_testing.append(calculate_log_liikelihood(test_data, test_label, weights))
638
639     # annealing of learning rate
640     learning_rate = learning_rate/(1+i/T)
641
642     # calculating error percentage on train, test and validation data
643     accuracy_plot_training.append((len(training_data) - calculate_error(weights, training_data)) / len(training_data))
644     accuracy_plot_validation.append((len(validation_data) - calculate_error(weights, validation_data)) / len(validation_data))
645     accuracy_plot_testing.append((len(test_data) - calculate_error(weights, test_data)) / len(test_data))
646
647     weights_array.append(weights)
648
649     # check for early stopping
650     early_stopping_horizon = 15
651     min_error_index = i
652     if(early_stopping(early_stopping_horizon, accuracy_plot_validation, i) and i > early_stopping_horizon):
653         min_error_index = i-early_stopping_horizon;
654         weights = weights_array[min_error_index]
655         break
656
657     weight_vector_length_array.append(weight_vector_length)
658     weights_for_all_lamda.append(weights)
659
660     log_likelihood_array.append(log_likelihood_training);
661     log_likelihood_array.append(log_likelihood_validation);
662     log_likelihood_array.append(log_likelihood_testing);

```

```

663     accuracy_plot_array.append(accuracy_plot_training)
664     accuracy_plot_array.append(accuracy_plot_validation)
665     accuracy_plot_array.append(accuracy_plot_testing)
666     accuracy_plot_training_array.append(accuracy_plot_training)
667
668     test_error = (calculate_error(weights, test_data, test_label)*100)/len(test_data)
669     validation_error = (calculate_error(weights, validation_data, validation_label)*100)/len(validation_data)
670     print('Error on validation dataset : ' + str(validation_error) + '%');
671     print('Error on test dataset : ' + str(test_error) + '%');
672     test_error_array.append(test_error)
673
674 #For single lambda value
675 if (len(lamda_vals) == 1):
676     plotlyGraphs(log_likelihood_array , ['Training Set','Validation Set','Test Set'], 'Log Likelihood')
677     plotlyGraphs(accuracy_plot_array , ['Training Set','Validation Set','Test Set'], 'Percentage Error')
678 else:
679     #For multiple lambda values
680     #plotlyGraphsRegularization(accuracy_plot_training_array , lamda_vals , "Accuracy vs Epoch")
681     #plotlyGraphsRegularization(weight_vector_length_array , lamda_vals , "Weight Vector Length")
682     #plotlyErrorVsLamda(test_error_array , lamda_vals)
683
684 # printing error on training and testing dataset
685 return weights_for_all_lamda
686
687 def plot_weights(weights):
688     lamda_vals = [1000, 100, 10, 1, 0.1, 0.001, 0.0001]
689     i = 0
690     for w in weights:
691         # Plot weights as image after removing bias terms. The rest of columns are pixels
692         print ('lambda = ' + str(lamda_vals[i]))
693         i += 1
694         pixels1 = dropFirstColumn(w)
695         pixels = numpy.reshape(pixels1 , (28, 28))
696         plt.imshow(pixels , cmap='gray')
697         plt.show()
698
699 def l2_norm(lamda , weights):
700     return 2*lamda*weights;
701
702 def l1_norm(lamda , weights):
703     w = numpy.ones(len(weights))
704     for i in range(len(weights)):
705         if (weights[i] < 0):
706             w[i] = -1
707     return lamda*w;
708
709 # 5.d - Final test error v/s Lambda values graph
710 # Generates bar graphs
711 def plotlyErrorVsLamda(test_error_array , lamda_vals):
712     lamda_vals1 = [math.log(lamda) for lamda in lamda_vals]
713     pyl.sign_in('chetang' , 'vil7vTAuCSWt2lEZvaH9')
714     trace = []
715     colors = ['rgb(104,224,204)', 'rgb(204,204,204)', 'rgb(49,130,189)', 'rgb(41,180,129)']
716     for i in range(0, len(test_error_array)):
717         y1 = test_error_array[i]
718         x1 = lamda_vals1[i]
719
720         trace1 = go.Bar(
721             x=x1 ,
722             y=y1 ,
723             name='Lambda = log(' + (str)(lamda_vals[i]) + ')',
724             marker=dict(
725                 color=colors[i]
726             )
727         )

```

```

728         trace.append(trace1)
729
730     layout = go.Layout(
731         xaxis=dict(tickangle=-45),
732         barmode='group',
733     )
734     fig = go.Figure(data=trace, layout=layout)
735     py1.iplot(fig, filename='Final Error vs Lambda')
736
737 #-----Main function-----
738
739 if __name__ == "__main__":
740     numpy.random.seed(0)
741
742     N = 20000
743     N_test = 2000
744     iteration = 200
745     batch_size = 0
746     T = 2000
747
748     #lamda_vals = [0]
749     lamda_vals = [1000, 100, 10, 1, 0.1, 0.001, 0.0001] # regularization weightage parameter.
750     norm = 2 # 2 for l-2 norm, 1 for l-1 norm
751
752     full_training_data, full_training_label, full_test_data, full_test_label, full_validation_
753
754     # Parameters for training data on 2's and 3's
755     digits = {2:1, 3:0}
756     training_data, training_label = extract_digit_specific_data(digits, full_training_data, fu
757     validation_data, validation_label = extract_digit_specific_data(digits, full_validation_da
758     test_data, test_label = extract_digit_specific_data(digits, full_test_data, full_test_labe
759     learning_rate = 0.0001
760
761     weights23 = fit(training_data, training_label, test_data, test_label,
762                     validation_data, validation_label, learning_rate, iteration,
763                     batch_size, digits, T, lamda_vals, norm)
764     plot_weights(weights23)
765
766     # Parameters for training data on 2's and 8's
767     digits = {2:1, 8:0}
768     training_data, training_label = extract_digit_specific_data(digits, full_training_data, fu
769     validation_data, validation_label = extract_digit_specific_data(digits, full_validation_da
770     test_data, test_label = extract_digit_specific_data(digits, full_test_data, full_test_labe
771     learning_rate = 0.1
772     lamda_vals = [0] #Regularization not asked for with 2/8 case
773
774     weights28 = fit(training_data, training_label, test_data, test_label,
775                     validation_data, validation_label, digits, learning_rate, iteration,
776                     batch_size, T, lamda_vals, norm)
777     plot_weights(weights28)
778
779     weights = weights28[0] - weights23[0]
780
781     plot_weights(weights)
782
783
784 Softmax_Regression.py
785
786 """
787 Softmax Regression on MNIST Data set to perform 10-way classification
788 """
789
790 import numpy
791 import math
792 import plotly.plotly as py1

```

```

793 import plotly.graph_objs as go
794
795 from LoadMNIST import load_mnist
796
797 #-----Utility functions-
798 def get_data(N, N_test):
799     #load MNIST data using libraries available
800     training_data, training_labels = load_mnist('training')
801     test_data, test_labels = load_mnist('testing')
802
803     training_data = flatArray(N, 784, training_data) #training_data is N x 784 matrix
804     training_labels = training_labels [:N]
805     test_data = flatArray(N_test, 784, test_data)
806     test_labels = test_labels [:N_test]
807
808     # adding column of 1s for bias
809     training_data = addOnesColAtStart(training_data)
810     test_data = addOnesColAtStart(test_data)
811
812     # Last 10% of training data size will be considered as the validation set
813     N_validation = int (N / 10)
814     validation_data = training_data[N-N_validation:N]
815     validation_labels = training_labels[N-N_validation:N]
816     N=N-N_validation
817     #update training data to remove validation data
818     training_data = training_data [:N]
819     training_labels = training_labels [:N]
820
821     return training_data, training_labels, test_data, test_labels, validation_data, validation_labels
822
823 def flatArray(rows, cols, twoDArr):
824     flattened_arr = numpy.zeros(shape=(rows, cols))
825     for row in range(0, rows):
826         i=0
827         for element in twoDArr[row]:
828             for ell in element:
829                 flattened_arr[row][i] = ell
830                 i = i+1
831     return flattened_arr
832
833 def addOnesColAtStart(matrix):
834     Ones = numpy.ones(len(matrix))
835     newMatrix = numpy.c_[Ones, matrix]
836     return newMatrix
837
838 # custom sigmoid function; if -x is too large, return value 0
839 def exp(x):
840     if(x < 709):
841         return math.exp(x)
842     else:
843         return math.exp(709)
844
845 def addRowToMatrix(matrix, row):
846     newMatrix = numpy.zeros(shape = ((len(matrix) + 1), len(matrix[0])))
847     for i in range(len(matrix)):
848         newMatrix[i] = matrix[i]
849     newMatrix[i+1] = row
850     return newMatrix
851
852 def l2_norm(lamda, weights):
853     return 2*lamda*weights;
854
855 def l1_norm(lamda, weights):
856     w = numpy.ones((len(numpy.array(weights)), len(numpy.array(weights)[0])))
857     for i in range(len(w)):

```

```

858         for j in range(len(w[0])):
859             if (weights[i,j] < 0):
860                 w[i][j] = -1
861     return lamda*numpy.matrix(w);
862
863 # custom sigmoid function; if -x is too large , return value 0
864 def sigmoid(x):
865     if(-x < 709):
866         return 1 / (1 + math.exp(-x))
867     else:
868         return 1 / (1 + math.exp(708))
869
870 def calculate_log_liikelihood(data , label , weights , t):
871     log_likelihood = 0.0
872     for j in range(0,t):
873         log_likelihood += (label[j]*numpy.log(sigmoid(numpy.dot(weights , data[j])))) + ((1-label[j])*numpy.log(1-sigmoid(numpy.dot(weights , data[j]))))
874
875     return -1*log_likelihood/t
876
877 def plotlyGraphs(error_plot_array , labels , name):
878     pyl.sign_in('cgandotr' , '3c9fh04498')
879     trace = []
880     for i in range(len(labels)):
881         y1 = error_plot_array[i]
882         y1 = [k for k in y1]
883         #y1 = y1[:min_index[i]]
884         x1 = [(j+1) for j in range(len(y1))]
885
886         trace1 = go.Scatter(
887             x=x1 ,
888             y=y1 ,
889             name = str(labels[i]) , # Style name/legend entry with html tags
890             connectgaps=True
891         )
892
893         trace.append(trace1)
894     data = trace
895     fig = dict(data=data)
896     pyl.iplot(fig , filename=name)
897
898 def early_stopping(early_stopping_horizon , accuracy , i):
899     if i > early_stopping_horizon:
900         counter = 0;
901         for p in range(0,early_stopping_horizon):
902             if accuracy[i-p] <= accuracy[i-p-1]:
903                 counter+=1
904             else:
905                 break
906
907         if(counter == early_stopping_horizon):
908             return True
909         else:
910             return False
911
912 def calculate_error(weights , data , label , k = 10):
913     error = 0.0;
914     for j in range(0,len(data)):
915         softmax_denom = 0.0
916         softmax_num = numpy.zeros(k);
917         for x in range(0,k):
918             softmax_num[x] = exp(numpy.dot(numpy.transpose(weights[:,x]), data[j]))
919             softmax_denom += softmax_num[x]
920
921         prediction = numpy.argmax(softmax_num/softmax_denom)
922         if(prediction != label[j]):
```

```

923         error += 1;
924     return error;
925
926 def softmax_loss(weights, labels, data, c):
927     loss = 0.0
928     for i in range(len(data)):
929         softmax_denom = 0.0
930         softmax_num = numpy.zeros(c);
931         for x in range(0,c):
932             softmax_num[x] = exp(numpy.dot(numpy.transpose(weights[:,x]), data[i]))
933             softmax_denom += softmax_num[x]
934         softmax = softmax_num[labels[i]]/softmax_denom
935         if (softmax > 0):
936             log_softmax = math.log(softmax)
937         else:
938             log_softmax = 0
939         loss += (log_softmax)
940     return -1*loss/(len(data))
941
942 def getSoftmax(k, weights, training_data, j):
943     softmax_denom = 0.0
944     softmax_num = numpy.zeros(k);
945     for x in range(0,k):
946         softmax_num[x] = exp(numpy.dot(numpy.transpose(weights[:,x]), training_data[j]))
947         softmax_denom += softmax_num[x]
948     return softmax_num/softmax_denom
949
950 def fit(training_data, training_label, test_data, test_label, validation_data,
951         validation_label, iteration = 1000, T=2000, lamda=0.001,
952         learning_rate = 0.0001, norm = 2):
953     k = len(numpy.unique(training_label))
954     t = len(training_data)
955     weights = numpy.matrix(numpy.zeros((len(training_data[0]), k)))
956     weights_array = []
957
958     early_stopping_horizon = 15
959     error_plot = numpy.zeros(iteration)
960     min_error_index = 0
961     loss_array = []
962     loss_training = []
963     loss_validation = []
964     loss_testing = []
965
966     accuracy_plot_array = []
967     accuracy_plot_training = []
968     accuracy_plot_validation = []
969     accuracy_plot_testing = []
970     test_error = 0.0
971
972     for i in range(0, iteration):
973         # initialise Gradient
974         gradient = numpy.matrix(numpy.zeros((len(training_data[0]), k)))
975
976         # calculate gradient over all the samples
977         for j in range(0,t):
978             modified_label = numpy.zeros(k);
979             modified_label[training_label[j]] = 1
980             softmax = getSoftmax(k, weights, training_data, j)
981             gradient += numpy.transpose(numpy.transpose(numpy.matrix(modified_label - softmax))
982
983             norm_term = 0.0
984             if (norm == 2):
985                 norm_term = l2_norm(lamda, weights)
986             else:
987                 norm_term = l1_norm(lamda, weights)

```

```

988 # update weights vector according to the update rule of Gradient descent method
989 weights = weights + learning_rate * (gradient - norm_term)
990
991 loss_training.append(softmax_loss(weights, training_label, training_data, k))
992 loss_validation.append(softmax_loss(weights, validation_label, validation_data, k))
993 loss_testing.append(softmax_loss(weights, test_label, test_data, k))
994
995 # calculating error percentage on train , test and validation data
996 training_error = calculate_error(weights, training_data, training_label)
997 validation_error = calculate_error(weights, validation_data, validation_label)
998 test_error = calculate_error(weights, test_data, test_label)
999
1000 accuracy_plot_training.append((len(training_data) - training_error)*100/len(training_data))
1001 accuracy_plot_validation.append((len(validation_data) - validation_error)*100/len(validation_data))
1002 accuracy_plot_testing.append((len(test_data) - test_error)*100/len(test_data))
1003
1004 learning_rate = learning_rate/(1+i/T)
1005
1006 error_plot[i] = validation_error*100/len(validation_data);
1007 weights_array.append(weights)
1008
1009 # check for early stopping
1010 if (early_stopping(early_stopping_horizon, accuracy_plot_validation, i)):
1011     min_error_index = i-early_stopping_horizon;
1012     weights = weights_array[min_error_index]
1013     break
1014 min_error_index = i
1015
1016 loss_array.append(loss_training)
1017 loss_array.append(loss_validation)
1018 loss_array.append(loss_testing)
1019
1020 accuracy_plot_array.append(accuracy_plot_training)
1021 accuracy_plot_array.append(accuracy_plot_validation)
1022 accuracy_plot_array.append(accuracy_plot_testing)
1023
1024 # printing error on training and testing dataset
1025 print('Error on validation dataset : ' + str(error_plot[min_error_index]) + '%');
1026 print('Error on test dataset : ' + str(test_error*100/len(test_data)) + '%');
1027
1028 return weights, loss_array, accuracy_plot_array
1029
1030 #-----Main function-----
1031
1032 if __name__ == "__main__":
1033     numpy.random.seed(0)
1034     learning_rate = 0.0001
1035     N = 20000
1036     N_test = 2000
1037     lamda = 0.001      # regularization weightage parameter
1038     T = 2000
1039     iteration = 1000
1040     training_data, training_label, test_data, test_label, validation_data, validation_label =
1041
1042     weights, loss_array, accuracy_plot_array = fit(training_data, training_label,
1043                                         test_data, test_label, validation_data,
1044                                         validation_label)
1045
1046     plotlyGraphs(loss_array, ['Training Set','Validation Set','Test Set'], "Loss Function and
1047     plotlyGraphs(accuracy_plot_array, ['Training Set','Validation Set','Test Set'], "Accuracy

```