Introduction to Machine Learning
Winter 2020
Lab 1: Linear and Logistic Regression

Team member	Contribution
Yunyi Zhu	1 (1.1 -1.5), 3.1.1 - 3.1.3
Zibo Wen	2 (2.1.1 - 2.1.3), 3.1.4 - 3.1.6

0. Introduction

The purpose of this lab is to investigate the classification performance of linear and logistic regression. We first implemented a linear regression classifier and a logistic regression classifier in NumPy, and then applied a batch Gradient Descent optimization algorithm to both of the learning models. The learning results from both models are investigated, evaluated and compared against ADAM using Stochastic Gradient Descent which is implemented by Tensorflow. The learning objective is to recognize the letter "C" (the positive class) and letter "J" (the negative class) from a dataset containing images of these two letters.

1. Linear Regression

1.1 Loss Function and Gradient

$$L = \frac{1}{N} \sum_{n=1}^{N} (W^T x^{(n)} + b - y^{(n)})^2 + \frac{\lambda}{2} (W^T W)$$

From this formula, we introduced the implementation of MSE

```
def MSE(W, b, x, y, reg):
      # Your implementation here
      length = len(x[0])*len(x[0][0]) #784
      x_matrix = np.reshape(x,(len(x),length))
      mse = np.sum(np.square(np.matmul(x_matrix,W)+b-y))/len(x) +
                (reg/2)*np.matmul(np.transpose(W),W)
      return mse[0][0];
```

In order to get the gradient, we take derivative on MSE.
The gradient of the MSE with respect to W is:

$$\frac{\partial L}{\partial W} == \frac{2}{N} x^T (xW + B - y) + \lambda W$$

and the gradient with respect to b is:

$$\frac{\partial L}{\partial b} = \frac{2}{N} (xW + B - y)$$

where $x$is a (N by d) data matrix,B is a (d by 1) array filled with b

From these two expressions, we implemented gradMSE:

```
def gradMSE(W, b, x, y, reg):
    # Your implementation here
    length = len(x[0])*len(x[0][0]) #784
    x_matrix = np.reshape(x,(3500,length))
    error = np.matmul(x_matrix,W) + b - y
    grad_w = np.matmul(np.transpose(x_matrix), error) * 2/len(x) + reg * W
    grad_b = np.sum(error)*2/len(x)
    return grad_w, grad_b;
```

1.2 Gradient Descent Implementation

The update in gradient descent is

$$W_{t+1} = W_t - \alpha g_t$$

where g is the gradient.

This applies to both weight and bias, coming up with the small snippet of grad_descent:

```
for i in range(epochs):
        grad_w, grad_b = gradMSE(W, b, x, y, reg);
        oldW = W
        W = W - alpha * grad_w
        b = b - alpha * grad_b
        if np.linalg.norm(oldW-W) < error_tol :
            break;
```

After adding additional arguments and code relating to plots, the grad_descent function looks like this:

```
#helper function, calculates the (number of correct estimate)/(number of true labels)
def accuracy(weight,bias,data,trueLabel):
      length = len(data[0])*len(data[0][0]) #784
      result = np.multiply(np.matmul(np.reshape(data,
(len(data),length)),weight)+bias-0.5, trueLabel-0.5)
      numCorrect = np.sum(result>0)
      return numCorrect/len(trueLabel);

def grad_descent(W, b, x, y, alpha, epochs, reg, error_tol, validData, testData,
validTarget, testTarget):
    # Your implementation here
    trainingError = []
    validationError = []
    testError = []
    trainingAccuracy = []
    validationAccuracy = []
    testAccuracy = []
    ploting = true
    for i in range(epochs):
        if ploting:
            trainingError.append(MSE(W, b, x, y, reg))
            validationError.append(MSE(W, b, validData, validTarget, reg))
            testError.append(MSE(W, b, testData, testTarget, reg))
            trainingAccuracy.append(accuracy(W, b, x, y))
            validationAccuracy.append(accuracy(W, b, validData, validTarget))
            testAccuracy.append(accuracy(W, b, testData, testTarget))

        grad_w, grad_b = gradMSE(W, b, x, y, reg);
        oldW = W
        W = W - alpha * grad_w
        b = b - alpha * grad_b
        if np.linalg.norm(oldW-W) < error_tol :
            break;

    plt.figure(1, figsize=(10, 10))
    trainingErrorLine, = plt.plot(trainingError, label='trainingError')
    validationErrorLine, = plt.plot(validationError, label='validationError')
    testErrorLine, = plt.plot(testError, label='testError')
    plt.ylabel('Error')
    plt.xlabel('Epoch')
    plt.title('Errors in Epoches')
    plt.legend([trainingErrorLine, validationErrorLine, testErrorLine],
['trainingError', 'validationError', 'testError'])
    plt.figure(2, figsize=(10, 10))
    trainingAccuracyLine, = plt.plot(trainingAccuracy, label='trainingAccuracy')
    validationAccuracyLine, = plt.plot(validationAccuracy, label='validationAccuracy')
    testAccuracyLine, = plt.plot(testAccuracy, label='testAccuracy')
    plt.ylabel('Accuracy')
    plt.xlabel('Epoch')
    plt.title('Accuracy in Epoches')
    plt.legend([trainingAccuracyLine, validationAccuracyLine, testAccuracyLine],
['trainingAccuracy', 'validationAccuracy', 'testAccuracy'])
    plt.show()
    return W,b
```

1.3 Tuning the Learning Rate

We trained the model three times with the learning rate = 0.005, 0.001 and 0.0001, respectively.

```
trainData, validData, testData, trainTarget, validTarget, testTarget = loadData();
W = np.empty((784,1));
W.fill(0.1);
b = 0.5;
reg = 0;
alpha = 0.005; #alpha = 0.001; alpha = 0.0001;
W, b = grad_descent(W, b, trainData, trainTarget, alpha, 5000, 0, 1E-7, validData,
testData, validTarget, testTarget);
```

From the training results below we can see that the model

Learning Rate	α=0.005	α=0.001	α=0.0001
bias	0.3649281434189931	0.24914456634713084	0.303054959771414
Training Accuracy	0.9842857142857143	0.9782857142857143	0.9025714285714286
Validation Accuracy	0.98	0.98	0.84
Testing Accuracy	0.9724137931034482	0.9793103448275862	0.9103448275862069
Training Error	0.025298483260130757	0.20114733150335246	0.6380647452803603
Validation Error	0.029954095588835467	0.22013125766991326	0.859909468517175
Testing Error	0.035899577318181886	0.20451971189136953	0.726420912101011

Table 1.3.A - Training results with different learning rates

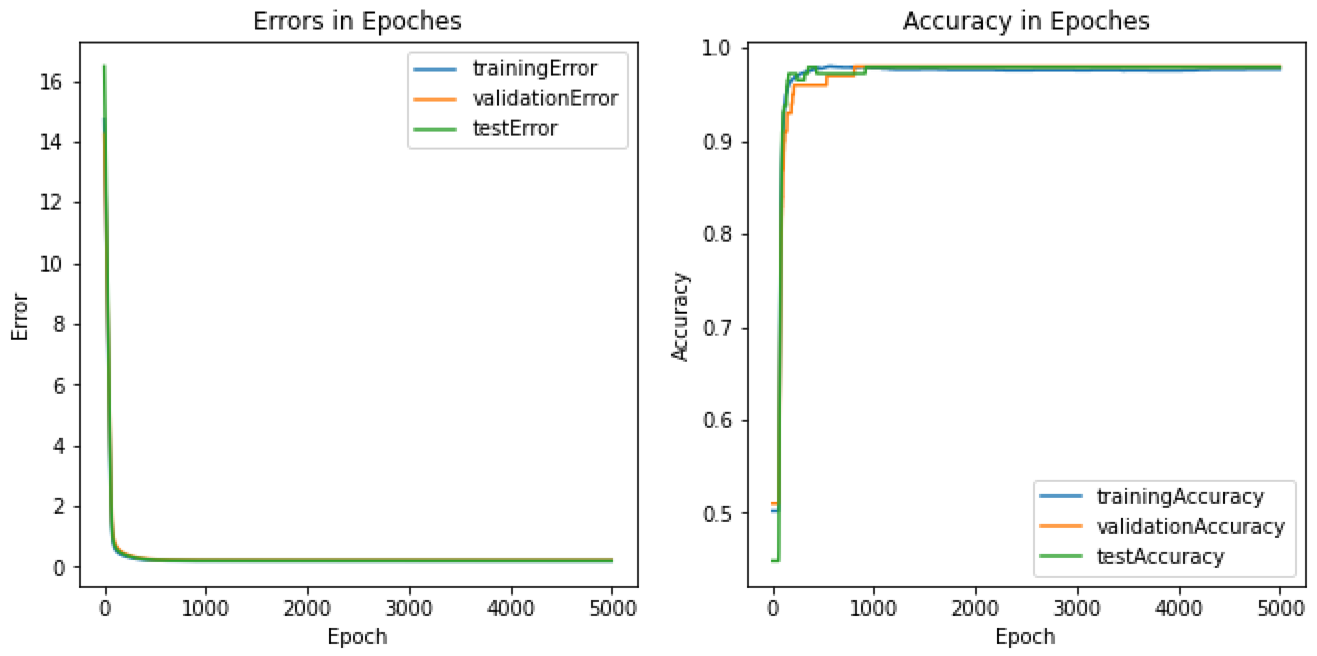


Figure 1.3.A - Training results with learning rate of 0.005

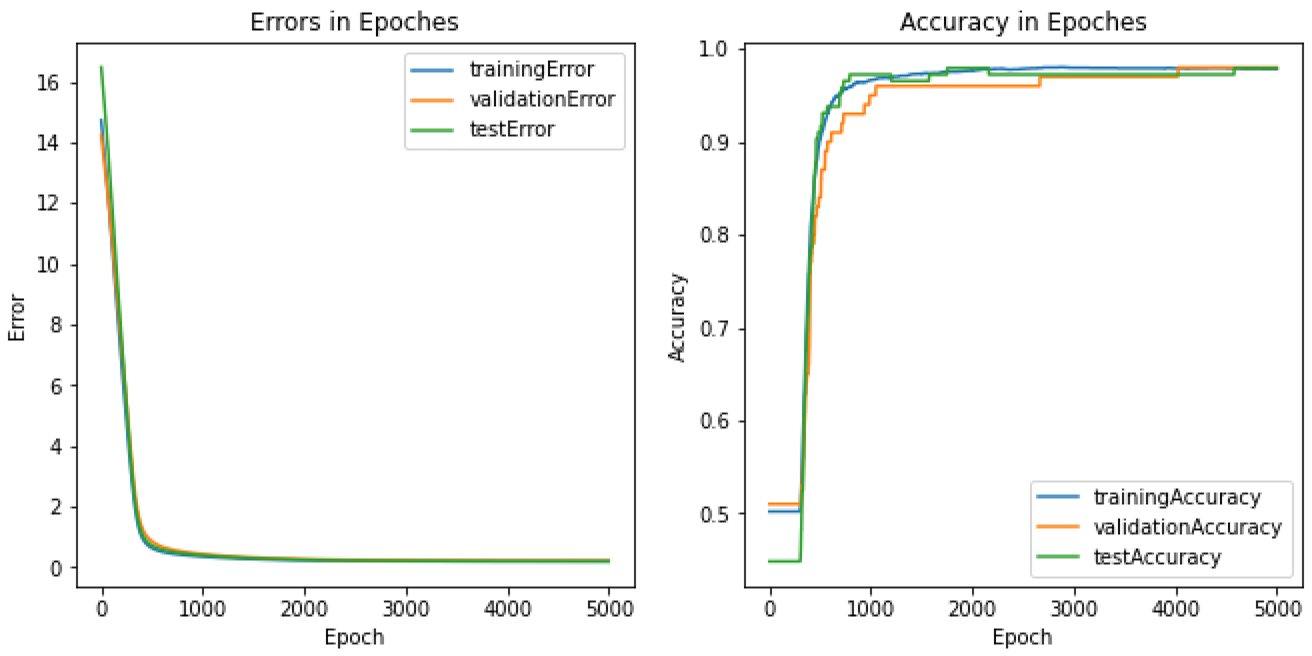


Figure 1.3.B - Training results with learning rate of 0.005

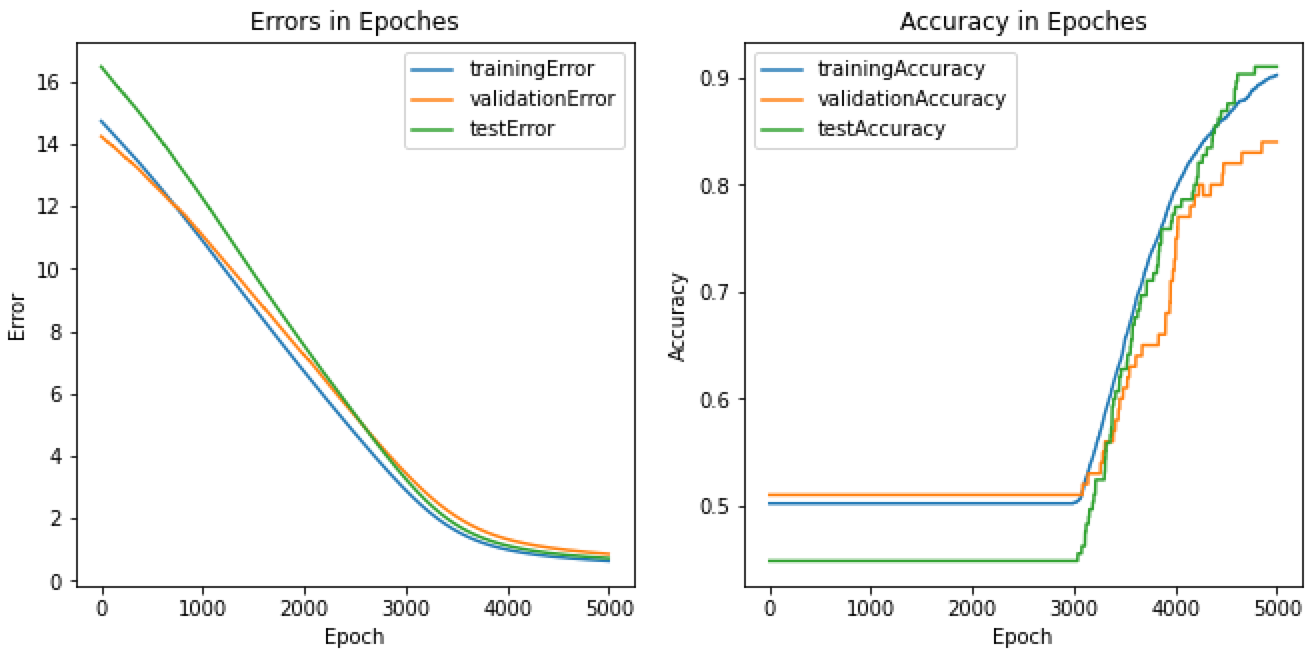


Figure 1.3.C - Training results with learning rate of 0.005

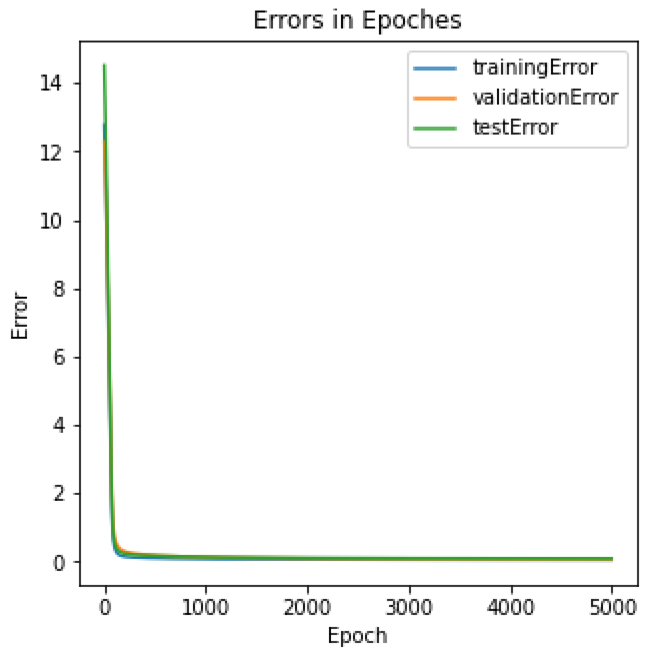
1.4 Generalization

Now we are investigating the impact of change in the regularization parameter. We trained the classifiers using λ=0.001, 0.1 and 0.5. The following results are generated:

```
trainData, validData, testData, trainTarget, validTarget, testTarget = loadData();
W = np.empty((784,1));
W.fill(0.1);
b = 0.5;
alpha = 0.005;
reg = 0.001
W, b = grad_descent(W, b, trainData, trainTarget, alpha, 5000, reg, 1E-7, validData,
testData, validTarget, testTarget);
```

Learning Rate	λ=0.001	λ=0.1	λ=0.5
bias	0.23552705620566744	0.16542970147555427	0.06866002330064858
Training Accuracy	0.9831428571428571	0.9814285714285714	0.9771428571428571
Validation Accuracy	0.97	0.98	0.98
Testing Accuracy	0.9793103448275862	0.9793103448275862	0.9793103448275862
Training Error	0.057094223859926754	0.11546489790645209	0.19907578992426883
Validation Error	0.06977541763627232	0.12738346635332867	0.21658284644420328
Testing Error	0.09453362368056131	0.12727926117345167	0.20172234064439654

Table 1.4.A - Training results with different regularization values



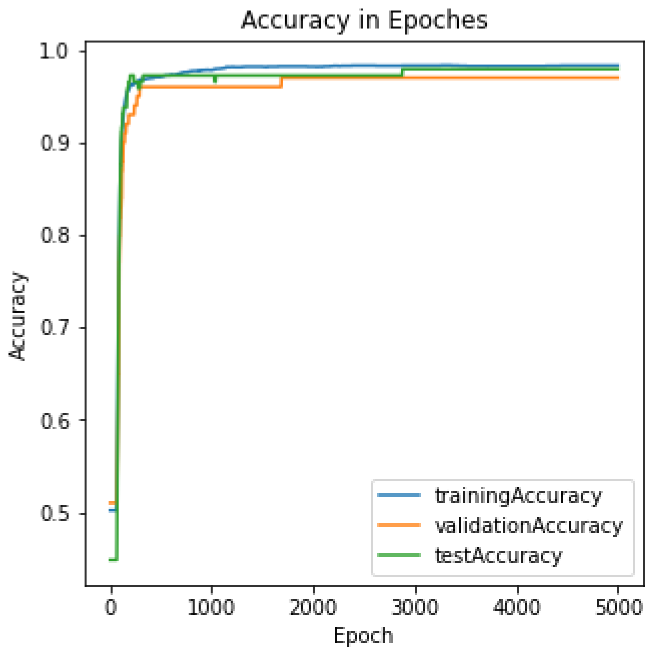


Figure 1.3.A - Training results with regularization of 0.001

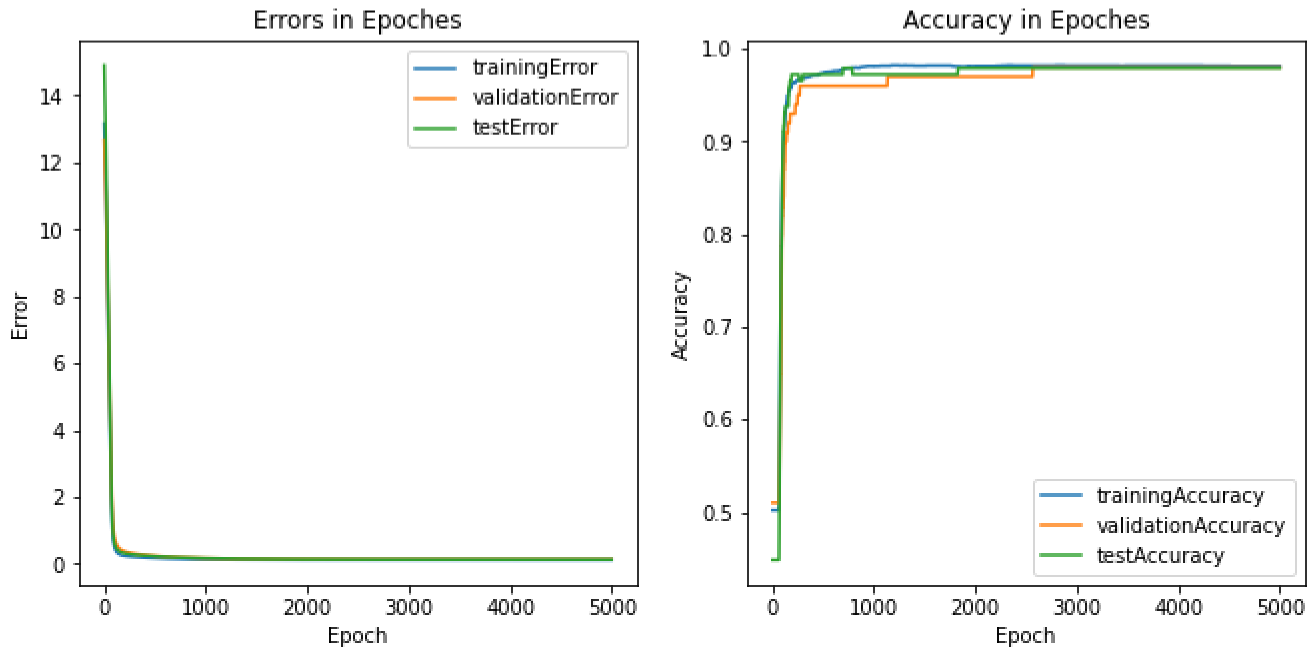


Figure 1.3.B - Training results with regularization of 0.1

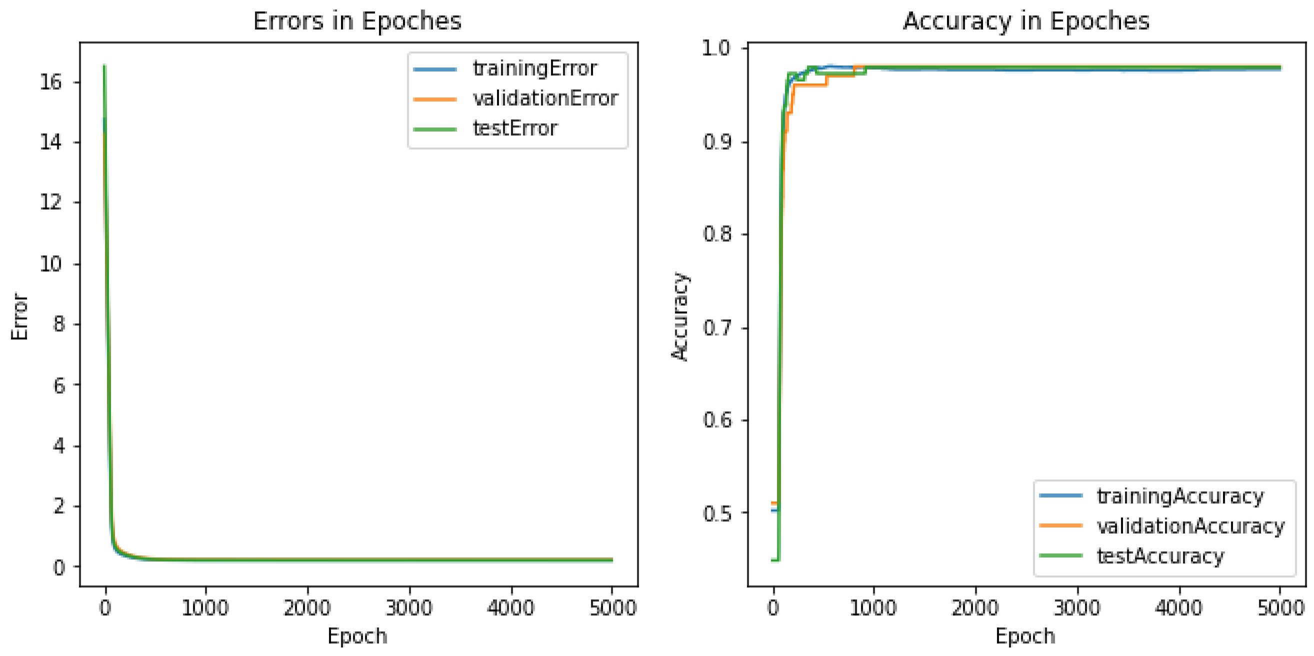


Figure 1.3.C - Training results with regularization of 0.5

1.5 Comparing Batch GD with normal equation

As taught in class, the normal equation for linear regression is as follows.

$$W = (x^T x + \lambda I)^{-1} x^T y$$

Together with this formula, we assume that the bias is 0.5 so y is either -0.5 or 0.5 instead of 0 or 1. With this assumption we came up with this implementation.

```
def normalEquation(data, reg, trueLabel):
    length = len(data[0])*len(data[0][0]) #784
    data_matrix = np.reshape(data,(3500,length))
    pseudoInvert = np.linalg.inv(np.matmul(np.transpose(data_matrix),data_matrix) +
                                 reg * np.identity(length) );
    weight = np.matmul(np.matmul(pseudoInvert, np.transpose(data_matrix)),trueLabel-0.5)
    return weight;
```

It is calculated with the following code snippet, assuming reg = 0 as it is not mentioned, and also assumes bias is 0.5 so y is from -0.5 to 0.5.

```
import time
trainData, validData, testData, trainTarget, validTarget, testTarget = loadData();
print(time.time());
W = normalEquation(trainData, 0, trainTarget)
print('trainDataAccuracy',accuracy(W,0.5,trainData,trainTarget))
print('validDataAccuracy',accuracy(W,0.5,validData,validTarget))
print('testDataAccuracy',accuracy(W,0.5,testData,testTarget))
print('trainDataError',MSE(W, 0.5, trainData, trainTarget, 0))
print('validDataError',MSE(W, 0.5, validData, validTarget, 0))
print('testDataError',MSE(W, 0.5, testData, testTarget, 0))
print(time.time());
```

Generating result of:

Training Accuracy	0.9948571428571429
Validation Accuracy	0.97
Testing Accuracy	0.9586206896551724
Training Error	0.019489841503138778
Validation Error	0.04765398976384138
Testing Error	0.05826213629284311
Computation time (second)	0.1785

Table 1.5.A - Training Results with Normal Equation

For the time with batch GD (1.3.1), calculation of error and accuracy during each epoch is turned off by turning if condition from 1 to 0.

Training Accuracy	0.9842857142857143
Validation Accuracy	0.98
Testing Accuracy	0.9724137931034482
Training Error	0.025298483260130757
Validation Error	0.029954095588835467
Testing Error	0.035899577318181886
Computation time (second)	15.9924

Figure 1.5.B - Training Results with Batch Gradient Descent

By comparing these aspects, it is clear that the training data accuracy when using the normal equation is phenomenal at 0.995. Whereas the validation data accuracy and testing data accuracy are lower than the case with batch gradient descent. This can be explained by the concept learned in class where exact solutions might not be desirable because it is overfitting in terms of the noise in the training data.

In terms of computation time, even though batch GD is having complexity of O(Nd) per iteration, while normal equation is expected to have O(Nd^2) complexity. In this case d=784, N=3500, iteration=5000, so Nd*iteration>Nd^2. This gives us the reason why the computation time is slow with GD. And the fact that np.linalg.inv is from the library but the implementation of GD is implemented manually without much optimization also contributes to the longer computation time.

2 Logistic Regression

2.1 Binary cross-entropy loss

2.1.1 Loss Function and Gradient

According to the handout, the cross-entropy loss is defined as the following:

$$L = L_D + L_W$$

$$= \sum_{n=1}^{N} \frac{1}{N}(-y^{(n)} log\, \hat{y}(x^{(n)}) - (1 - y^{(n)}) log(1 - \hat{y}(x^{(n)})) + \frac{\lambda}{2}||W||_2^2$$

where

$$\hat{y}(x) = \sigma(W^T x + b) = \frac{1}{1+e^{-(W^T x+b)}}.$$

Based on the equations above, the cross entropy loss calculation is implemented. The prediction is represented as 'sigma' and the total cross entropy loss is the sum of 'loss_D' and 'loss_W'.

Also, we found that after a number of iterations, the value of the sigmoid function can get very close to 0 or 1 and get evaluated by NumPy as 0 or 1, which causes error in the later log operations. Thus, we introduced a very small constant to prevent these errors.

```
def crossEntropyLoss(W, b, x, y, reg):
    N = len(y)
    x = x.reshape((x.shape[0], x.shape[1] * x.shape[2]))
    sigma = 1 / (1 + np.exp(- (np.dot(x, W) + b)))
    for i in range(len(sigma)):
        if sigma[i] == 0:
            sigma[i] = EPSILON_0
        elif sigma[i] == 1:
            sigma[i] = 1-EPSILON_0

    for i in range(len(sigma)):
        if sigma[i] == 0:
            print('error')
        elif sigma[i] == 1:
            print('error')
    loss_D = - (y * (np.log(sigma)) + (1-y) * np.log(1 - sigma))
    loss_D = 1/N * loss_D.sum()
    loss_W = (reg / 2) * (np.linalg.norm(W) ** 2)
    return loss_D + loss_W
```

The gradient of the cross entropy loss is found by taking the partial derivatives.

$$set\ z = e^{-(W^T x^{(n)} + b)}$$

$$\frac{\partial}{\partial W} L$$

$$= \sum_{n=1}^{N} \frac{1}{N} \frac{\partial}{\partial W} (- y^{(n)} log\, \hat{y}(\frac{1}{1+z}) - (1 - y^{(n)}) log(\frac{z}{1+z})) + \frac{\partial}{\partial W} (\frac{\lambda}{2} ||W||_2^2)$$

$$= \sum_{n=1}^{N} \frac{1}{N} \frac{\partial}{\partial W} (y^{(n)} log(1 + z) - (1 - y^{(n)}) log(z)$$

$$+ (1 - y^{(n)}) log(1 + z)) + \frac{\partial}{\partial W} (\frac{\lambda}{2} ||W||_2^2)$$

$$= \sum_{n=1}^{N} \frac{1}{N} \frac{\partial}{\partial W} (log(1 + z) - (1 - y^{(n)})(- (W^T x^{(n)} + b)))$$

$$+ \frac{\partial}{\partial W} (\frac{\lambda}{2} ||W||_2^2)$$

$$= \sum_{n=1}^{N} \frac{1}{N} (\frac{-x^{(n)} e^{-z}}{1+e^{-z}} + (1 - y^{(n)}) x^{(n)}) + \lambda W$$

$$= \sum_{n=1}^{N} \frac{1}{N} (x^{(n)} (\frac{1}{1+e^{-z}} - y^{(n)})) + \lambda W$$

$$= \frac{1}{N} (x(\hat{y} - y)) + \lambda W$$

$$\Downarrow$$

$$\frac{\partial}{\partial W}L = \frac{1}{N}(x(\hat{y} - y)) + \lambda W$$

Similarly, the partial derivative with respect to b is calculated:

$$\frac{\partial}{\partial B}L$$

$$= \sum_{n=1}^{N} \frac{1}{N}\frac{\partial}{\partial B}(- y^{(n)} log\, \hat{y}(\frac{1}{1+z}) - (1 - y^{(n)})log(\frac{z}{1+z})) + \frac{\partial}{\partial B}(\frac{\lambda}{2}||W||_2^2)$$

$$= \sum_{n=1}^{N} \frac{1}{N}\frac{\partial}{\partial B}(y^{(n)}log(1 + z) - (1 - y^{(n)})log(z)$$

$$+ (1 - y^{(n)})log(1 + z))$$

$$= \sum_{n=1}^{N} \frac{1}{N}\frac{\partial}{\partial B}(log(1 + z) - (1 - y^{(n)})(- (W^T x^{(n)} + b)))$$

$$= \sum_{n=1}^{N} \frac{1}{N}(\frac{-e^{-z}}{1+e^{-z}} + (1 - y^{(n)}))$$

$$= \sum_{n=1}^{N} \frac{1}{N}(\frac{1}{1+e^{-z}} - y^{(n)})$$

$$= \sum_{n=1}^{N} \frac{1}{N}(\hat{y}^{(n)} - y^{(n)})$$

$$\Downarrow$$

$$\frac{\partial}{\partial B}L = \sum_{n=1}^{N} \frac{1}{N}(\hat{y}^{(n)} - y^{(n)})$$

The gradCE function is implemented based on the analytical expressions in the way below:

```python
def gradCE(W, b, x, y, reg):
    N = len(y)
    x = x.reshape((x.shape[0], x.shape[1] * x.shape[2]))
    sigma = 1 / (1 + np.exp(- x @ W - b))
    grad_B = 1/N * (sigma - y).sum()
    grad_W = 1/N * (x.T @ (sigma - y)) + reg * W
    return grad_W, grad_B
```

2.1.2 Learning

By setting reg = 0.1, alpha = 0.005 and 5000 epochs, we get the following results.

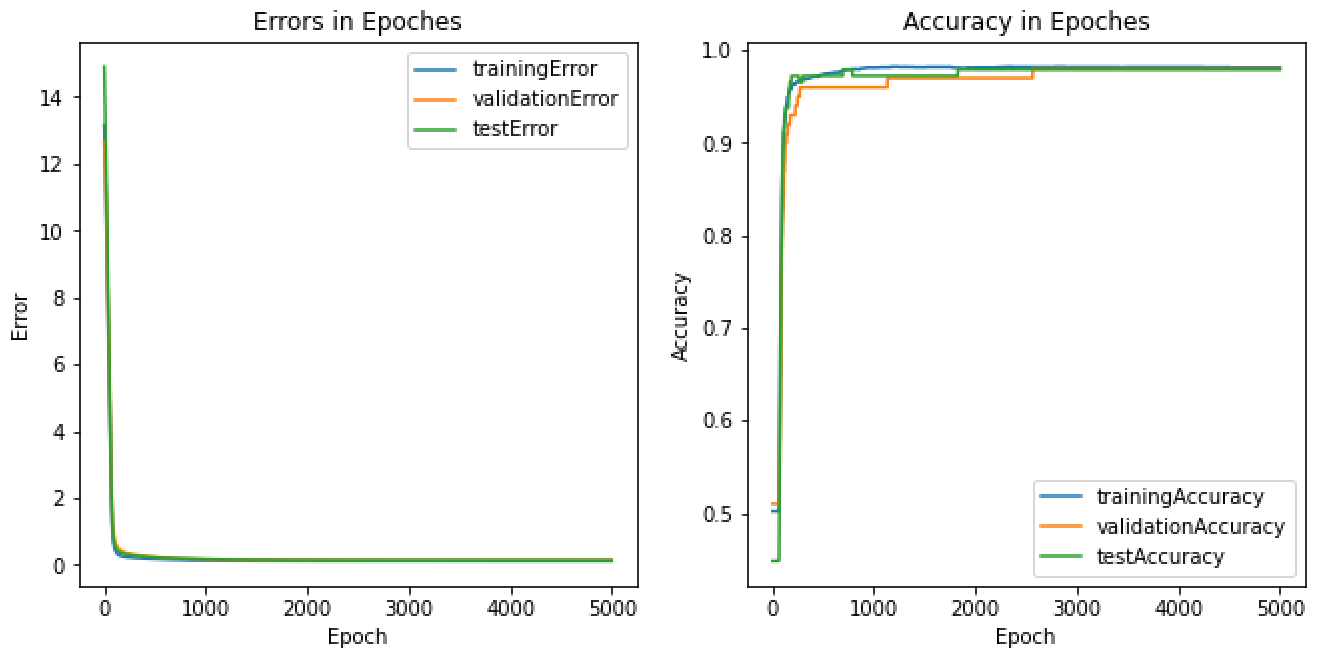


Figure 2.1.2.A - Logistic Regression Model Training Results

bias	0.23552705620566744
Training Accuracy	0.9831428571428571
Validation Accuracy	0.97
Testing Accuracy	0.9793103448275862
Training Error	0.057094223859926754
Validation Error	0.06977541763627232
Testing Error	0.09453362368056131

Table 2.1.2.A - Linear Regression Model Training Results

2.1.3 Comparison to Linear Regression

With zero weight decay, learning rate of 0.005 and 5000 epochs, it can be seen from the graphs that the cross-entropy loss for logistic regression method converges a bit slowly compared to the MSE loss for linear regression.

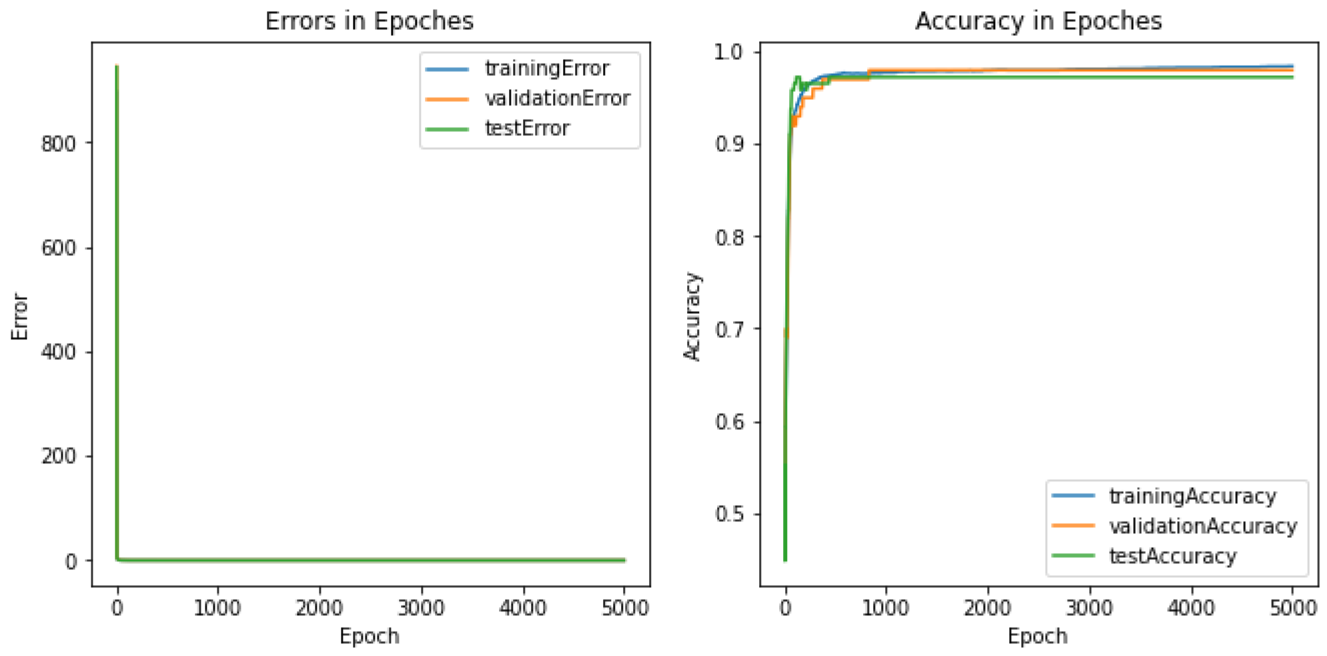


Figure 2.1.3.A - Linear Regression Model Training Result

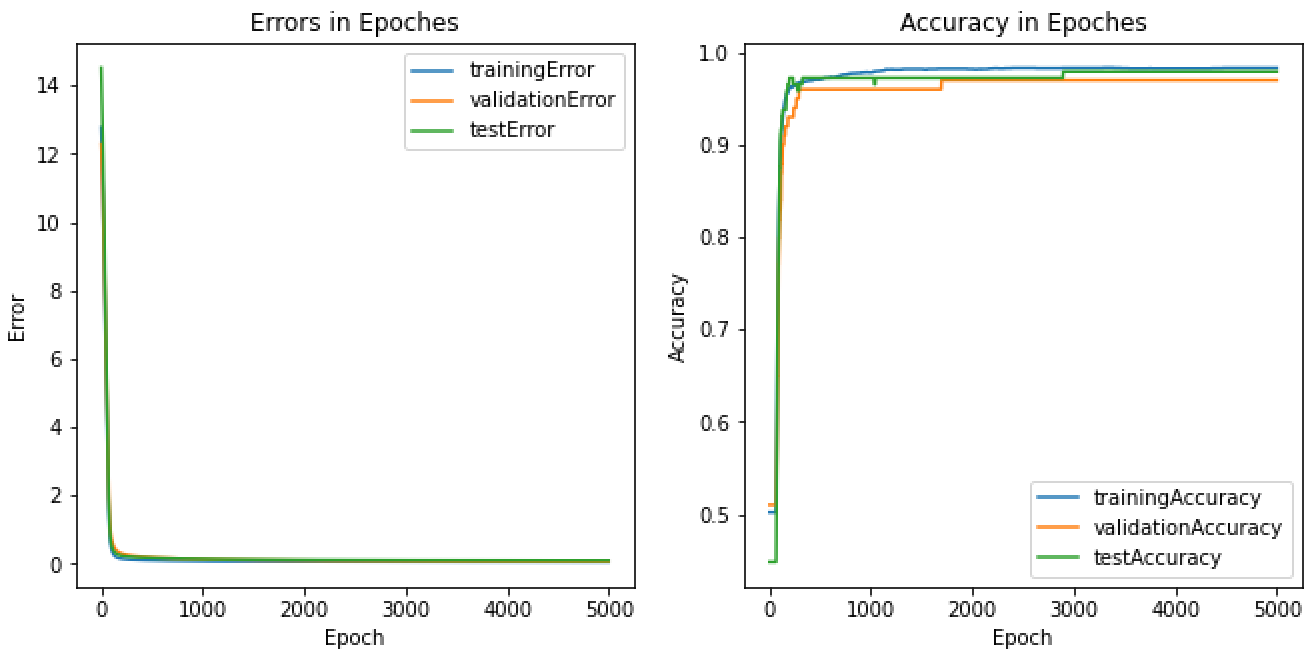


Figure 2.1.3.B - Logistic Regression Model Training Result

Loss Function	MSE	CE
bias	0.3649281434189932	0.23638640277692763
Training Accuracy	0.9842857142857143	0.9831428571428571
Validation Accuracy	0.98	0.97
Testing Accuracy	0.9724137931034482	0.9793103448275862
Training Error	0.5258724105476759	0.05580386511016057
Validation Error	0.5272327079592162	0.06872604563814537
Testing Error	0.5537272220543418	0.09410253581752777

Table 2.1.3.A - Training Results from Both Linear Regression and Logistic Regression Models

3 Batch Gradient Descent vs. SGD and Adam

3.1 SGD

3.1.1 Building the Computational Graph

The implementation of the buildGraph follows the instructions in the handout. The optimizer used is AdamOptimizer. One deviation is the use of batch size which is not stated in the instructions, but will get used in the later parts for mini-batch gradient descent.

```
def buildGraph(loss="MSE"):
    #Initialize weight and bias tensors
    batchSize = 500
    weight = tf.Variable(tf.truncated_normal(shape=(784, 1), stddev=0.5,
dtype=tf.float32))
    bias = tf.Variable(0.5)

    data = tf.placeholder(tf.float32, shape=(batchSize, 784), name="data")
    labels = tf.placeholder(tf.float32, shape=(batchSize, 1), name="labels")
    reg = tf.placeholder(tf.float32)

    alpha = tf.constant(0.001)

    theLoss = 0;
    predictions = 0;

    tf.set_random_seed(421)

    if loss == "MSE":
        # Your implementation
        predictions = tf.matmul(data, weight)+bias
        theLoss = tf.losses.mean_squared_error(labels=labels, predictions=predictions) +
                                                     reg/2*tf.reduce_sum(tf.math.square(weight))

    elif loss == "CE":
        #Your implementation here
        predictions = tf.sigmoid(tf.matmul(data, weight) + tf.convert_to_tensor(bias))
        theLoss = tf.losses.sigmoid_cross_entropy(labels, predictions) +
                                                   reg/2*tf.reduce_sum(tf.math.square(weight))

    optimizer = tf.train.AdamOptimizer(alpha).minimize(theLoss)
    return weight,bias,reg,predictions,data,labels,theLoss,optimizer
```

3.1.2 Implementing Stochastic Gradient Descent

The following code snippet utilizes the graph with batch size of 500.

```
trainData, validData, testData, trainTarget, validTarget, testTarget = loadData();
W,b,reg,predictions,data,labels,loss,optimizer = buildGraph("MSE");

init_op = tf.global_variables_initializer()

batchSize = 500
batchAmount = int(len(trainData)/batchSize)
epoch = 700
length = len(trainData[0])*len(trainData[0][0]) #784
trainDataReshaped = np.reshape(trainData,(3500,length))

trainingError = []
validationError = []
testError = []
trainingAccuracy = []
validationAccuracy = []
testAccuracy = []
reglambda = 0

with tf.Session() as sess:
    sess.run(init_op)
    for i in range(epoch):
        for j in range(batchAmount):
            dataBatch = trainDataReshaped[j*batchSize:(j+1)*batchSize]
            labelBatch = trainTarget[j*batchSize:(j+1)*batchSize]
            feed_dict={labels: labelBatch,data: dataBatch,reg: reglambda}
            sess.run(optimizer, feed_dict=feed_dict);

        trainingError.append(MSE(W.eval(), b.eval(), trainData, trainTarget, reglambda))
        validationError.append(MSE(W.eval(), b.eval(), validData, validTarget,
                                            reglambda))
        testError.append(MSE(W.eval(), b.eval(), testData, testTarget, reglambda))
        trainingAccuracy.append(accuracy(W.eval(), b.eval(), trainData, trainTarget))
        validationAccuracy.append(accuracy(W.eval(), b.eval(), validData, validTarget))
        testAccuracy.append(accuracy(W.eval(), b.eval(), testData, testTarget))
```

As a result, the following plots and outputs are generated.

bias	0.40881944
Training Accuracy	0.9291428571428572
Validation Accuracy	0.93
Testing Accuracy	0.8896551724137931
Training Error	0.08831005
Validation Error	0.11563801
Testing Error	0.15910906

Table 3.1.2.A - Training Results from Linear Regression Model with SGD

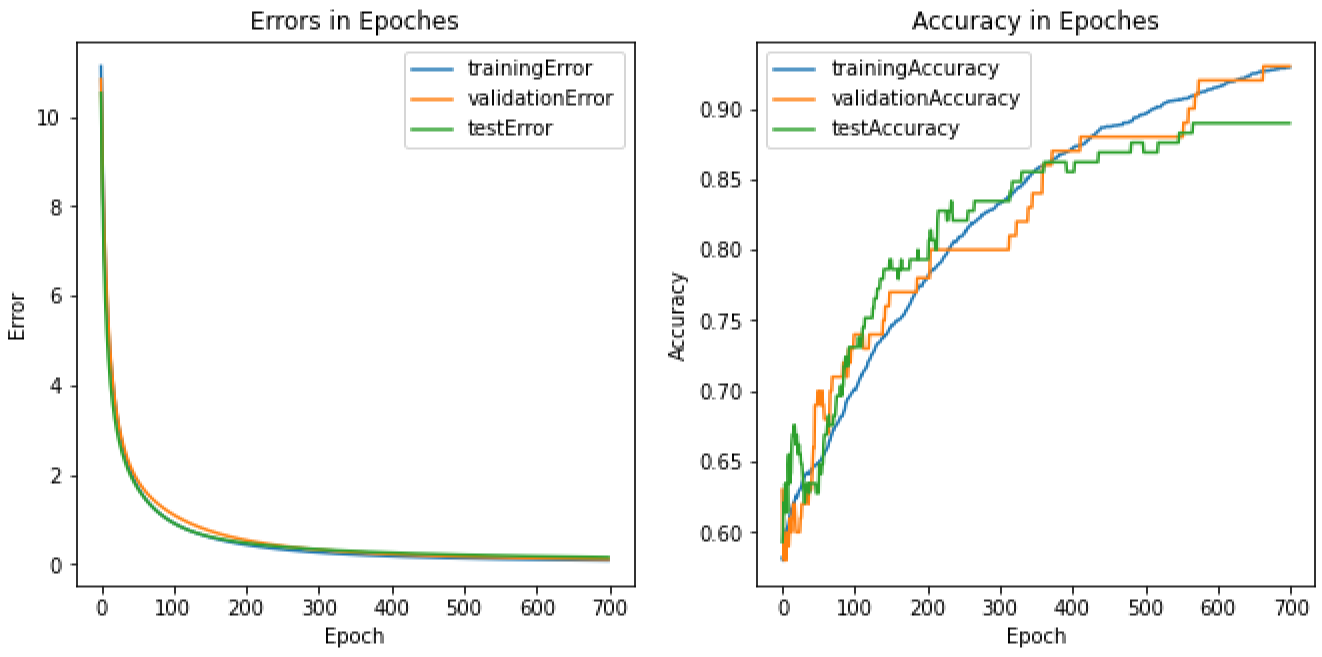


Figure 3.1.2.A - Training Results from Linear Regression Model with SGD

3.1.3 Batch Size Investigation

By observing the testing results listed below, the final accuracy is the best with small batch size 100, and the case worsens when batch size gets larger; the worst case is with the large batch size 1750. The reason behind this is that with smaller batch sizes, more updates are done to the weights and bias. That is, with batch size 100, 24500 updates are done, while with the batch size of 1750 only 1400 updates are done. Even though the update is according to a subset of all points, more updates generally make more progress.

Batch Size	100	700	1750
bias	0.3690248	0.40754357	0.4186941
Training Accuracy	0.9842857142857143	0.8037142857142857	0.7537142857142857
Validation Accuracy	0.95	0.79	0.71
Testing Accuracy	0.9724137931034482	0.7448275862068966	0.696551724137931
Training Error	0.030033266	0.33667582	0.5772425
Validation Error	0.05926081	0.42381102	0.93712115
Testing Error	0.04965886	0.4061949	0.8028544

Table 3.1.3.A - Training Results from Linear Regression Model with SGD using Different Batch Size

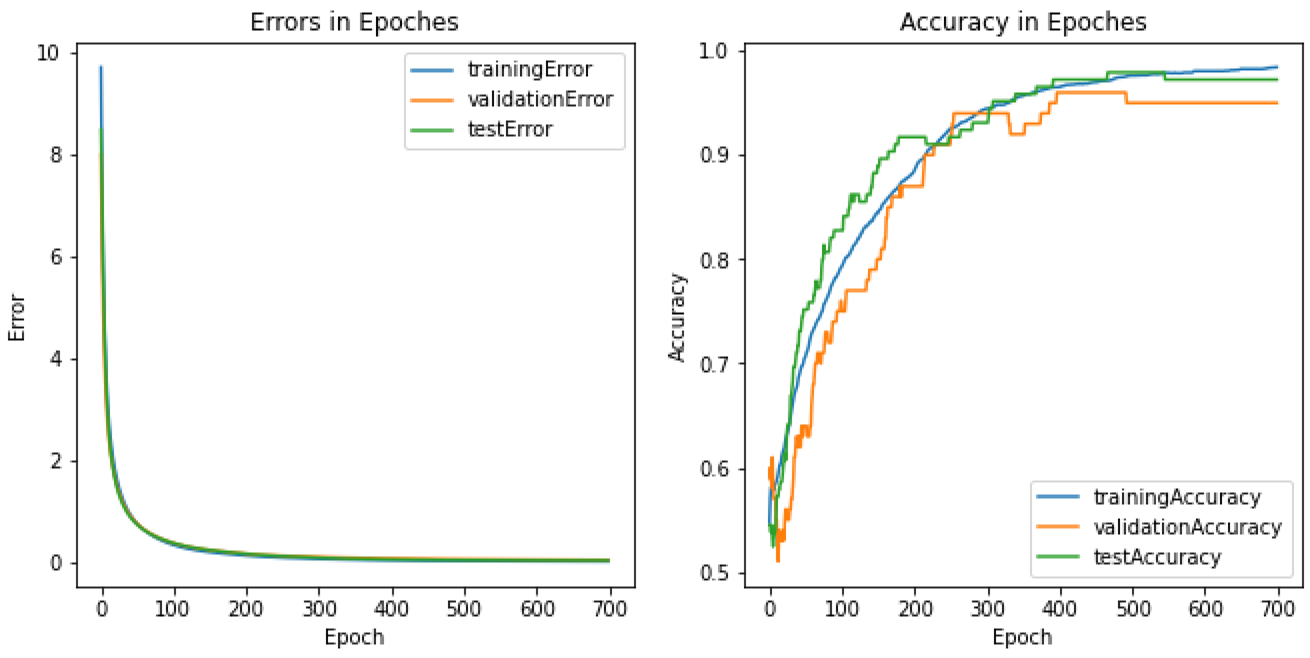


Figure 3.1.2.A - Training Results with Batch Size of 100 (MSE + SGD)

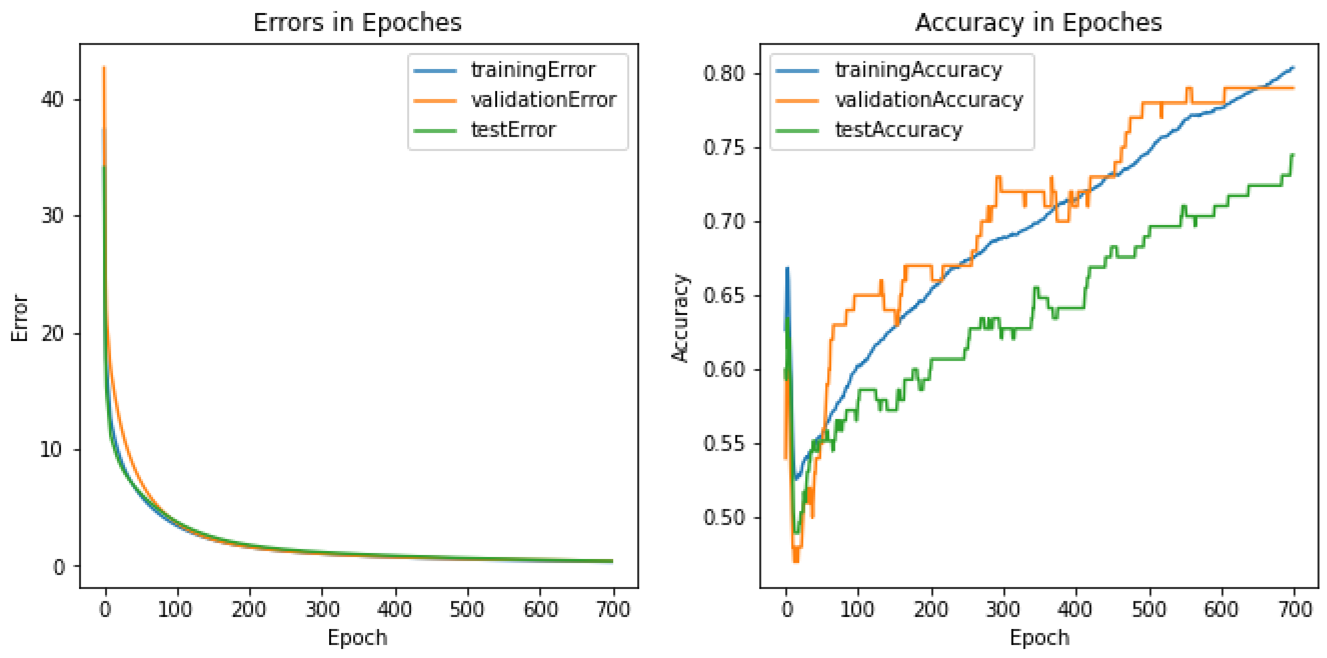


Figure 3.1.2.B - Training Results with Batch Size of 700 (MSE + SGD)

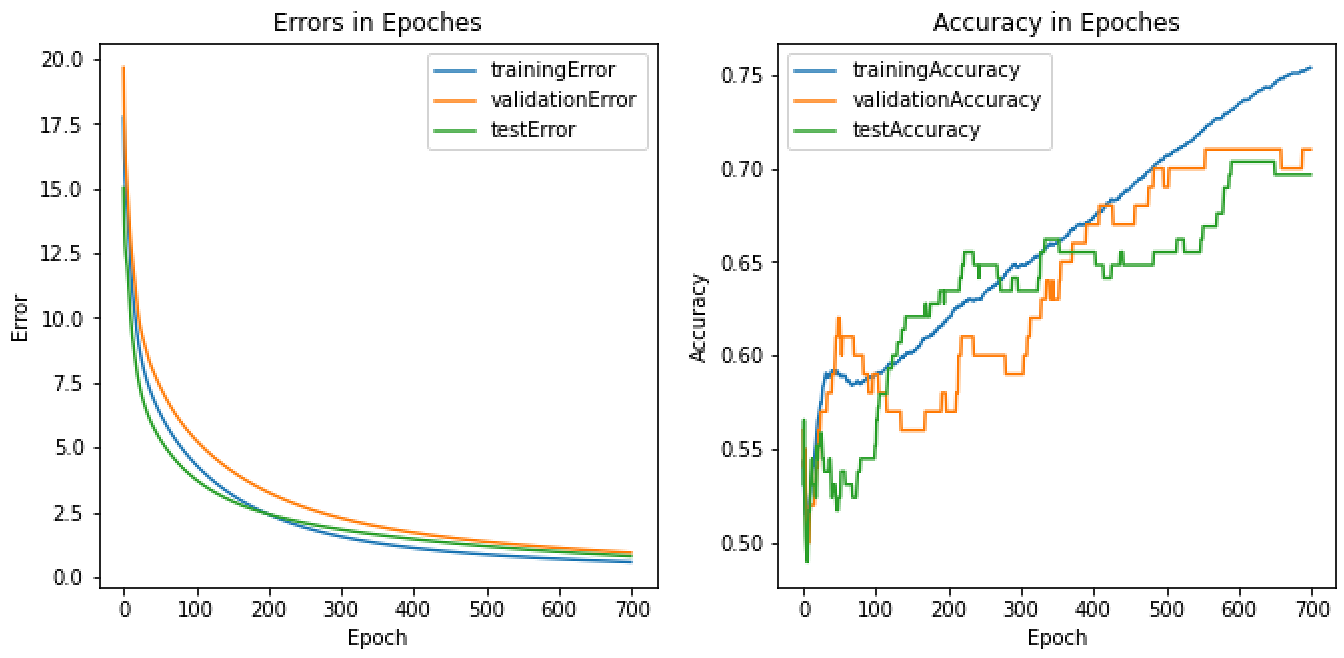


Figure 3.1.2.A - Training Results with Batch Size of 1750 (MSE + SGD)

3.1.4 Hyperparameter Investigation

3.1.4.1 Change in β1

The testing result suggests that the smaller β1, 0.95 produces the better training result. This might be because with a smaller β1, the momentum decays quicker so that by the end of the training, the trained model will not be influenced by the momentum too much.

β1	0.95	0.99
bias	0.32937667	0.37132633
Training Accuracy	0.9205714285714286	0.9037142857142857
Validation Accuracy	0.87	0.89
Testing Accuracy	0.9103448275862069	0.8275862068965517
Training Error	0.11236409	0.1439735
Validation Error	0.2147988	0.23316114
Testing Error	0.1602432	0.2138181

Table 3.1.4.A - Training Results with different β1 values (MSE + SGD)

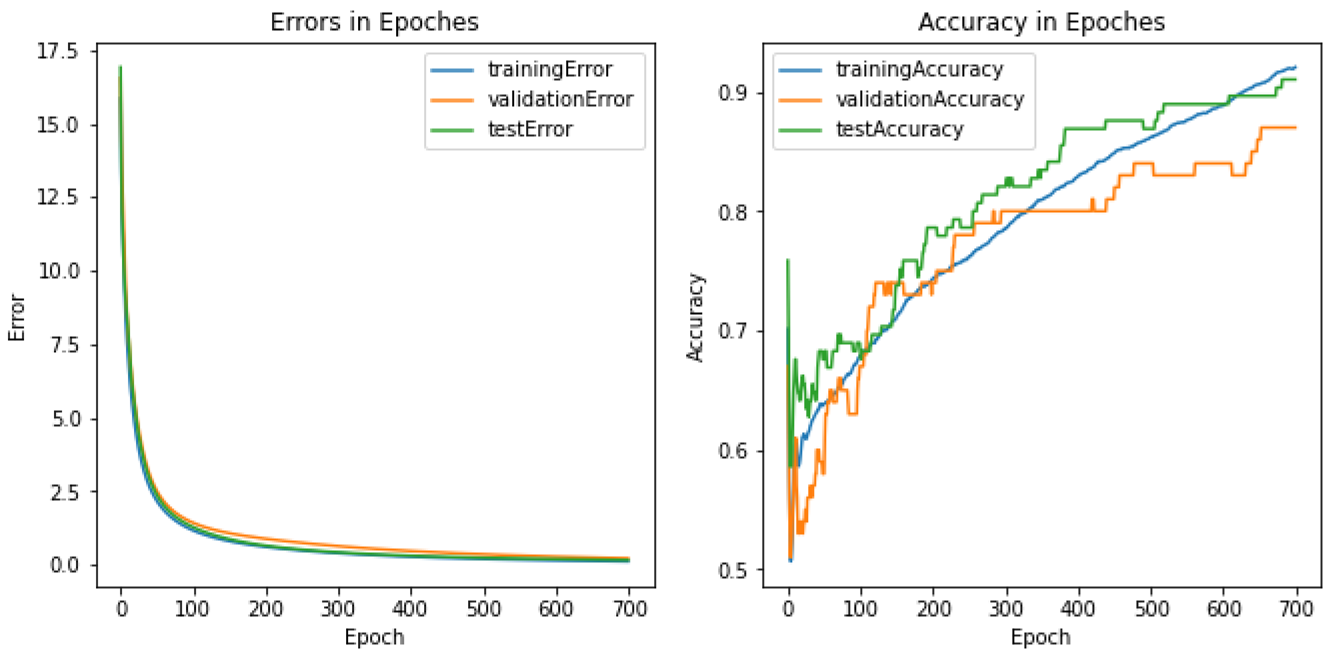


Figure 3.1.4.A - Training Results with β1=0.95 (MSE + SGD)

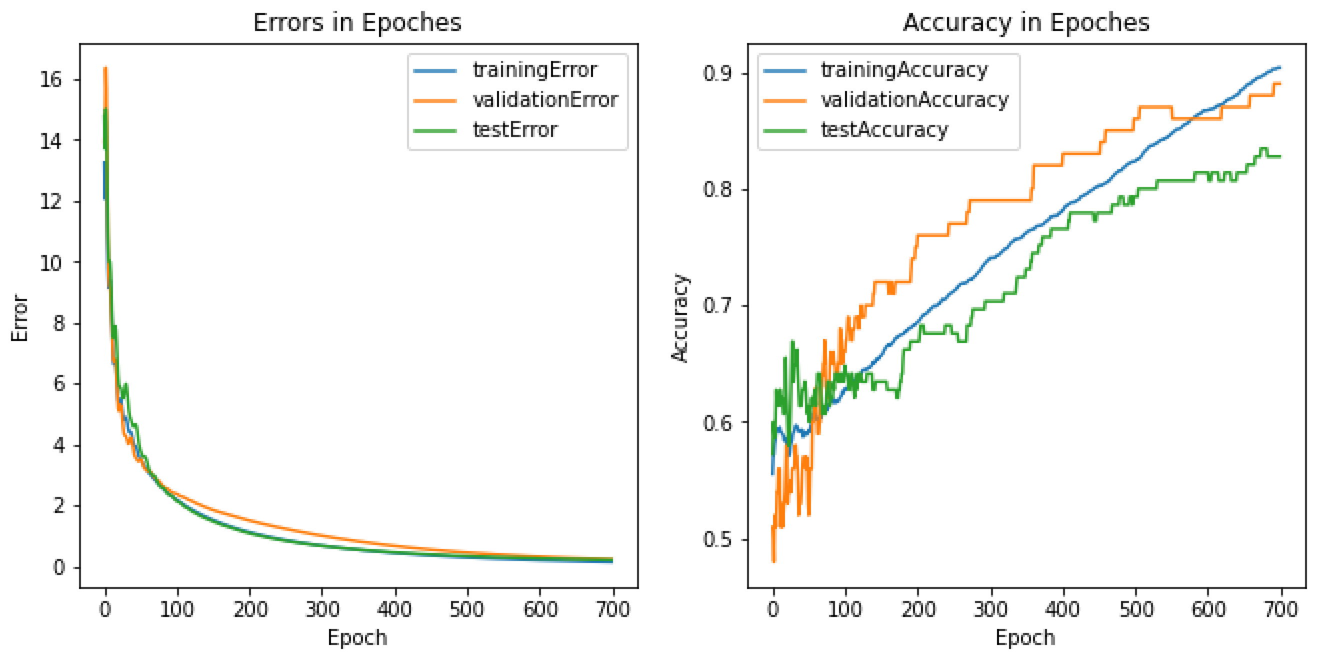


Figure 3.1.4.B - Training Results with β1=0.99 (MSE + SGD)

3.1.4.2 Change in β2

Similar to β1, the smaller β2, 0.99, gives more accurate results. The decaying momentum can provide better convergence by the end of the training.

β2	0.99	0.9999
bias	0.3904941	0.4589015
Training Accuracy	0.9717142857142858	0.7445714285714286
Validation Accuracy	0.95	0.73
Testing Accuracy	0.9379310344827586	0.6620689655172414
Training Error	0.044751093	0.5980295
Validation Error	0.0809719	1.0074929
Testing Error	0.08170669	0.74695885

Table 3.1.4.B - Training Results with different β2 values (MSE + SGD)

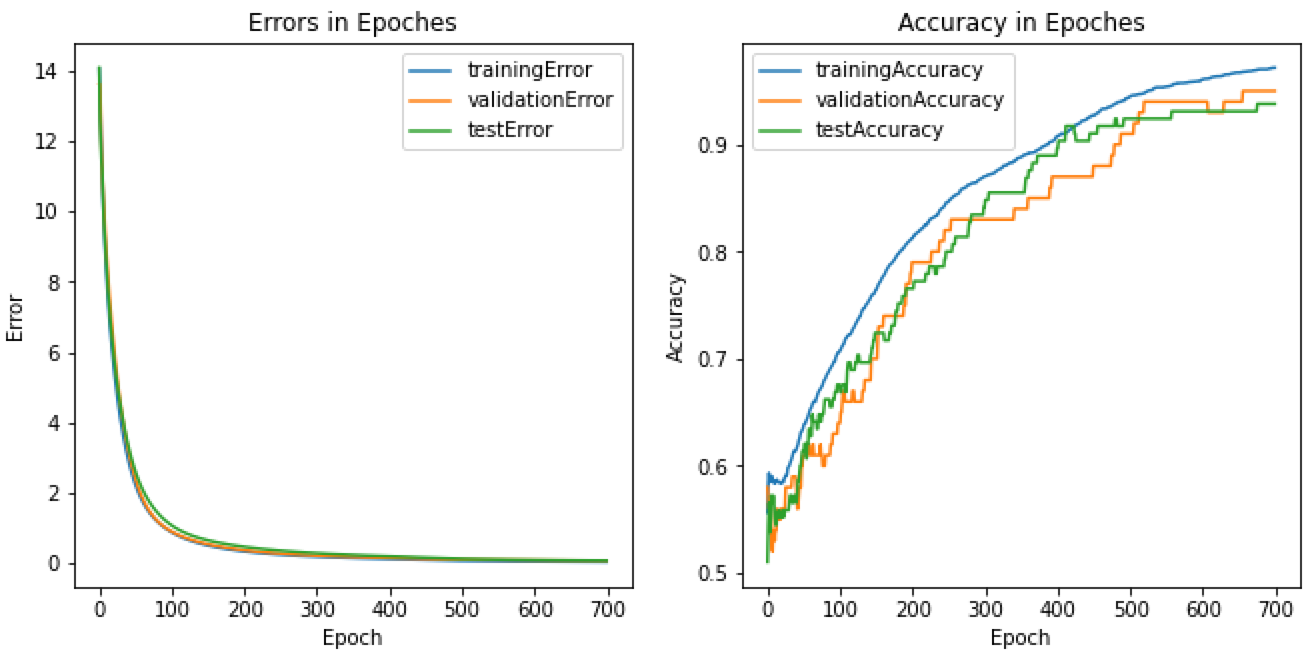


Figure 3.1.4.C - Training Results with β2=0.99 (MSE + SGD)

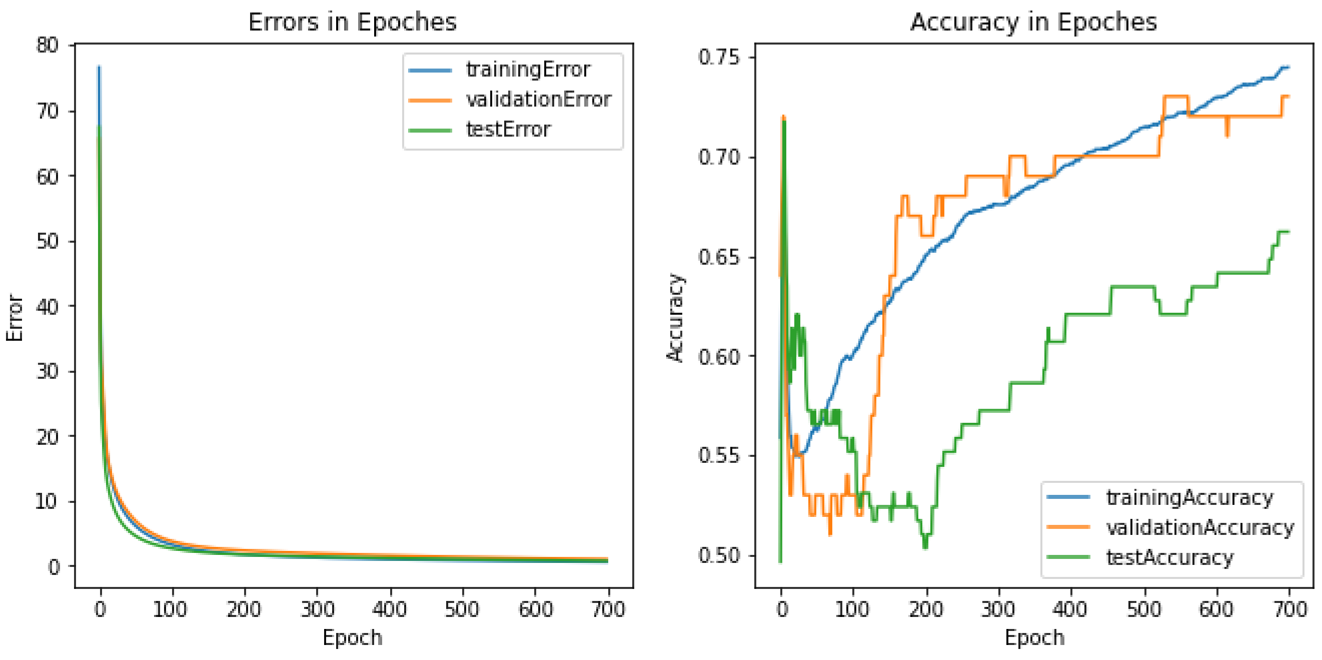


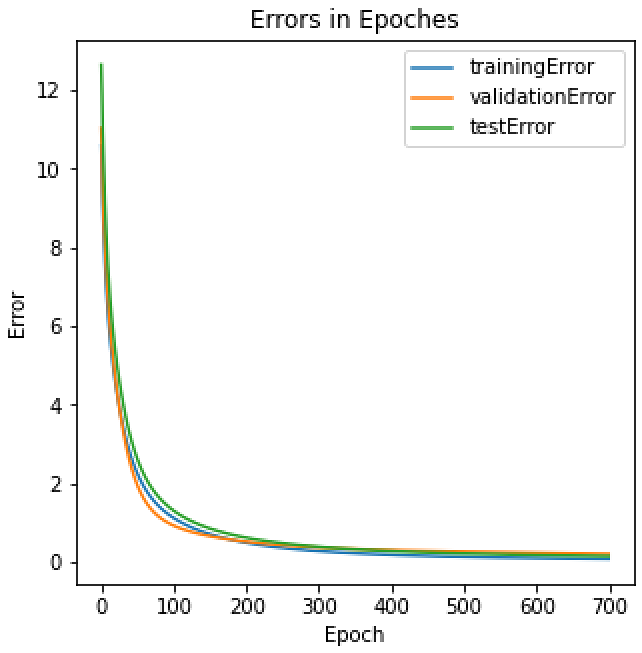
Figure 3.1.4.D - Training Results with β2=0.9999 (MSE + SGD)

3.1.4.3 Change in ϵ

It can be seen that the smaller epsilon, 1e-09, gives the slightly better testing accuracy. This is because epsilon is supposed to be a number that is very close to 0 to prevent any division by zero in the calculation. If the value for epsilon is too large, the greater error is introduced into the calculation.

ϵ	1e-09	1e-04
bias	0.3574337	0.3180791
Training Accuracy	0.9354285714285714	0.9257142857142857
Validation Accuracy	0.87	0.86
Testing Accuracy	0.896551724137931	0.8827586206896552
Training Error	0.08196478	0.10187195
Validation Error	0.20766734	0.1643379
Testing Error	0.15221685	0.18677466

Table 3.1.4.B - Training Results with different ϵ values (MSE + SGD)



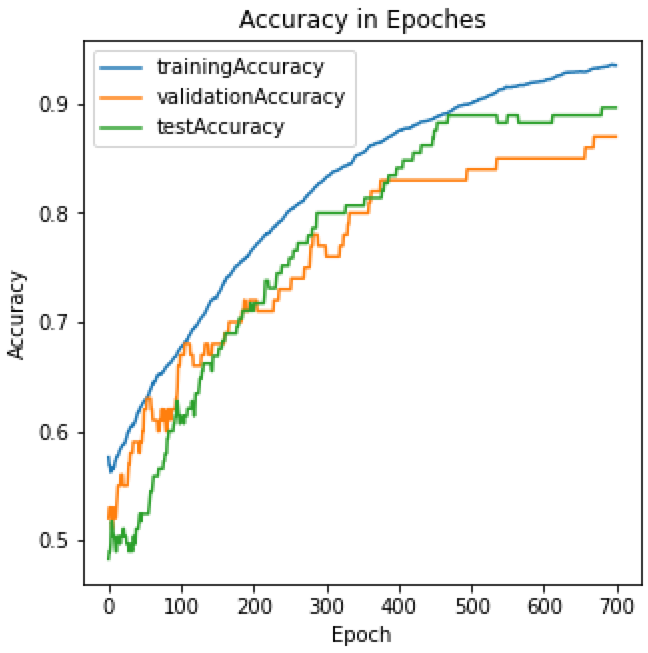
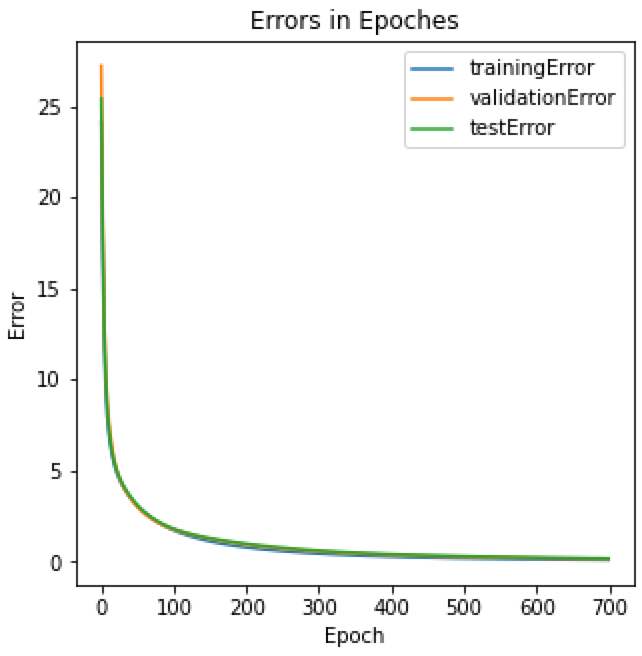
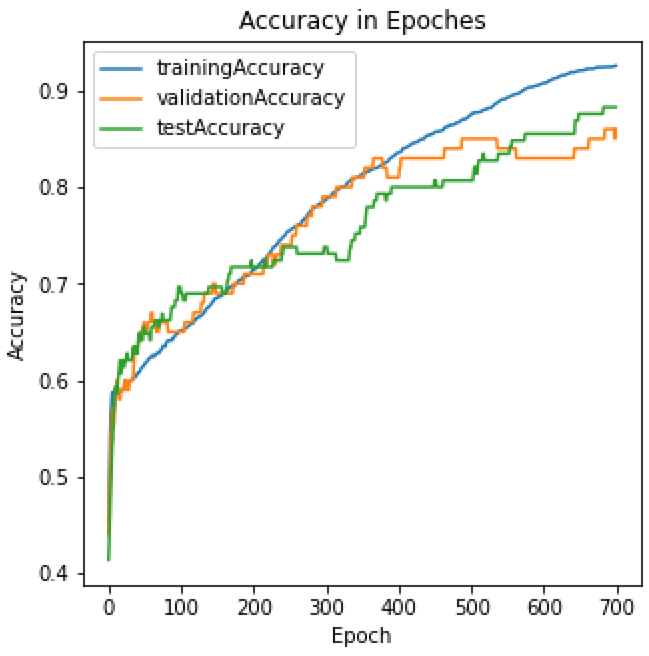


Table 3.1.4.B - Training Results with different ϵ values (MSE + SGD)





(Note: The differences in the accuracies between each comparison in their groups are fairly small and the influence of the randomness from SGD cannot be fully eliminated.)

3.1.5 Cross Entropy Loss Investigation

In general, in terms of working with SGD and Adam optimization, the Logistic Regression by minimizing the binary cross entropy loss behaves much better than Linear Regression by minimizing the MSE loss.
The final trainDataAccuracy, validDataAccuracy and testDataAccuracy are all around 0.98 for CE loss method, while the number is only around 80 for the MSE method.

Implementing Stochastic Gradient Descent

The implementation of SGD (with a minibatch size of 500 optimizing over 700 epochs, minimizing the CE) gives the following results:

bias	0.079911456
Training Accuracy	0.9882857142857143
Validation Accuracy	0.98
Testing Accuracy	0.9862068965517241
Training Error	0.13769974039136557
Validation Error	0.04559302530806555
Testing Error	0.1672928455371929

Table 3.1.5.A - Training Results with Batch Size of 500 (CE + SGD)

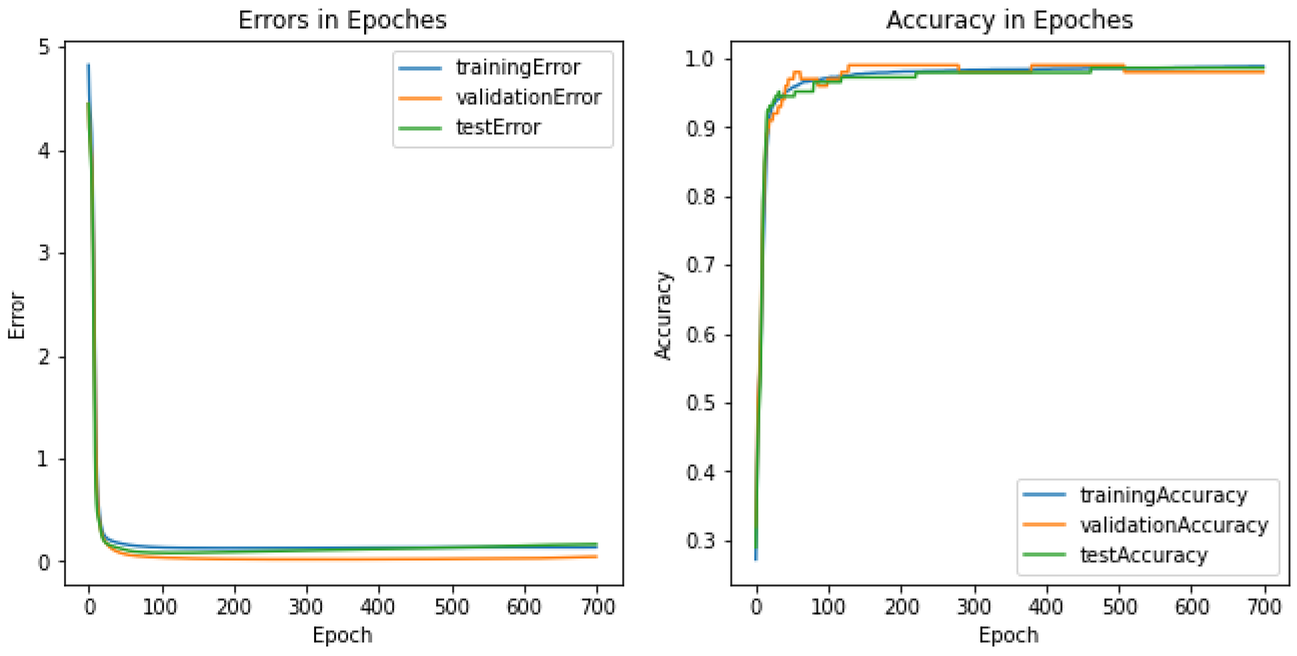


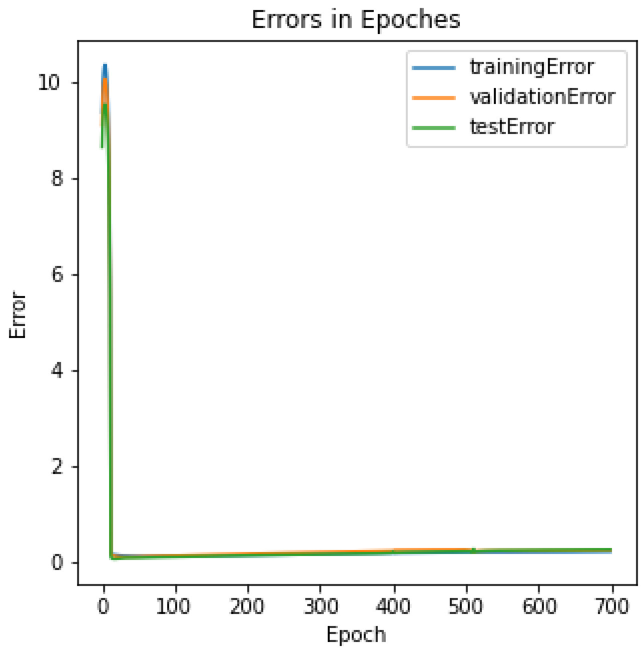
Figure 3.1.5.A - Training Results with Batch Size of 500 (CE + SGD)

Batch Size Investigation

Here are the running results with batch size of 100, 700 and 1750 respectively. As expected, the smallest batch size grants the best training result, and the converge speed decreases as the batch size grows, which complies with the conclusion drawn from the batch size investigation performed in section 3.1.3.

Batch Size	100	700	1750
bias	0.079911456	0.09720248	bias 0.20641649
Training Accuracy	0.9882857142857143	0.9897142857142858	0.9862857142857143
Validation Accuracy	0.98	0.97	0.96
Testing Accuracy	0.9862068965517241	0.9793103448275862	0.9655172413793104
Training Error	0.13769974039136557	0.12693422402224114	0.1101924999550106
Validation Error	0.045593025308065555	0.2199372990980805	0.19779261707467574
Testing Error	0.1672928455371929	0.2265682386441882	0.25323158346269203

Table 3.1.5.B - Training Results with Different Batch Size (CE + SGD)



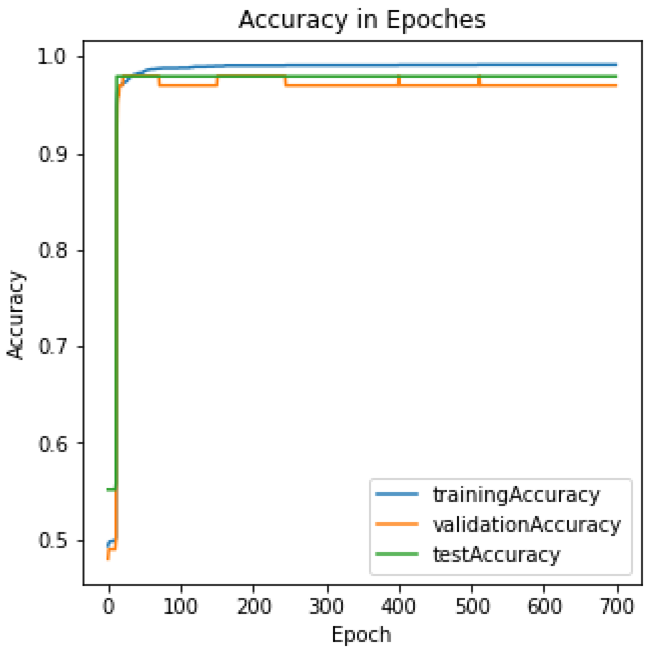


Figure 3.1.5.B - Training Results with Batch Size of 100 (CE + SGD)

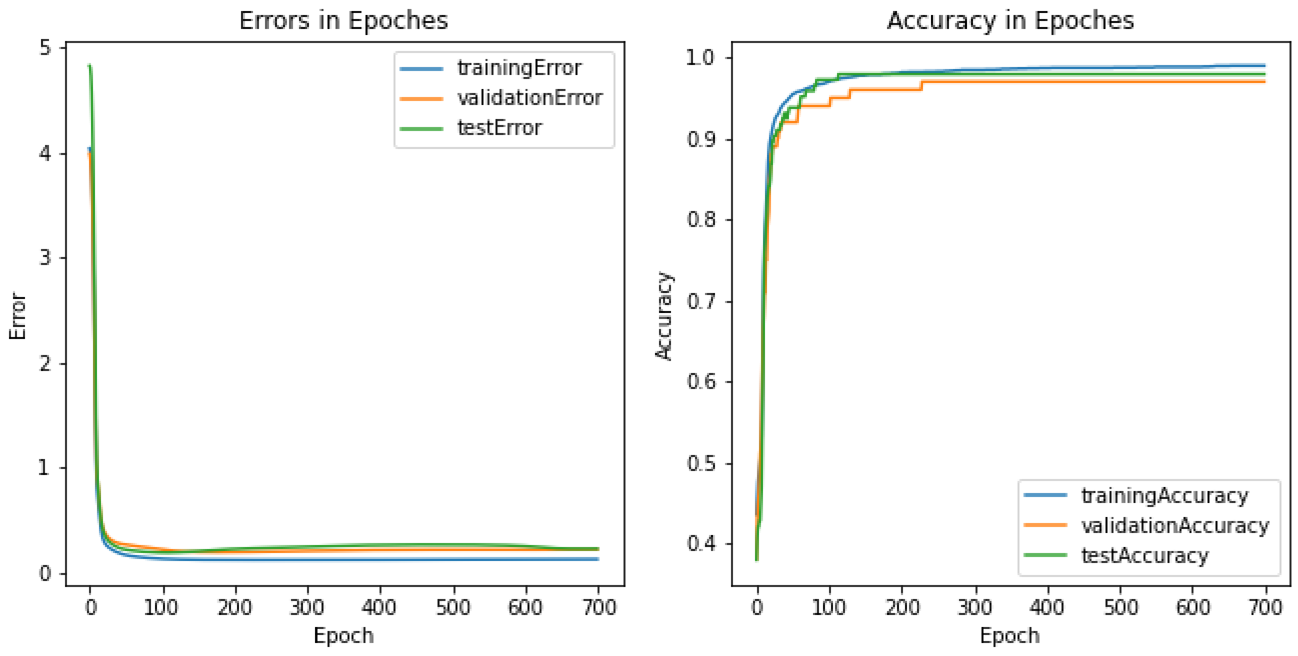


Figure 3.1.5.C - Training Results with Batch Size of 500 (CE + SGD)

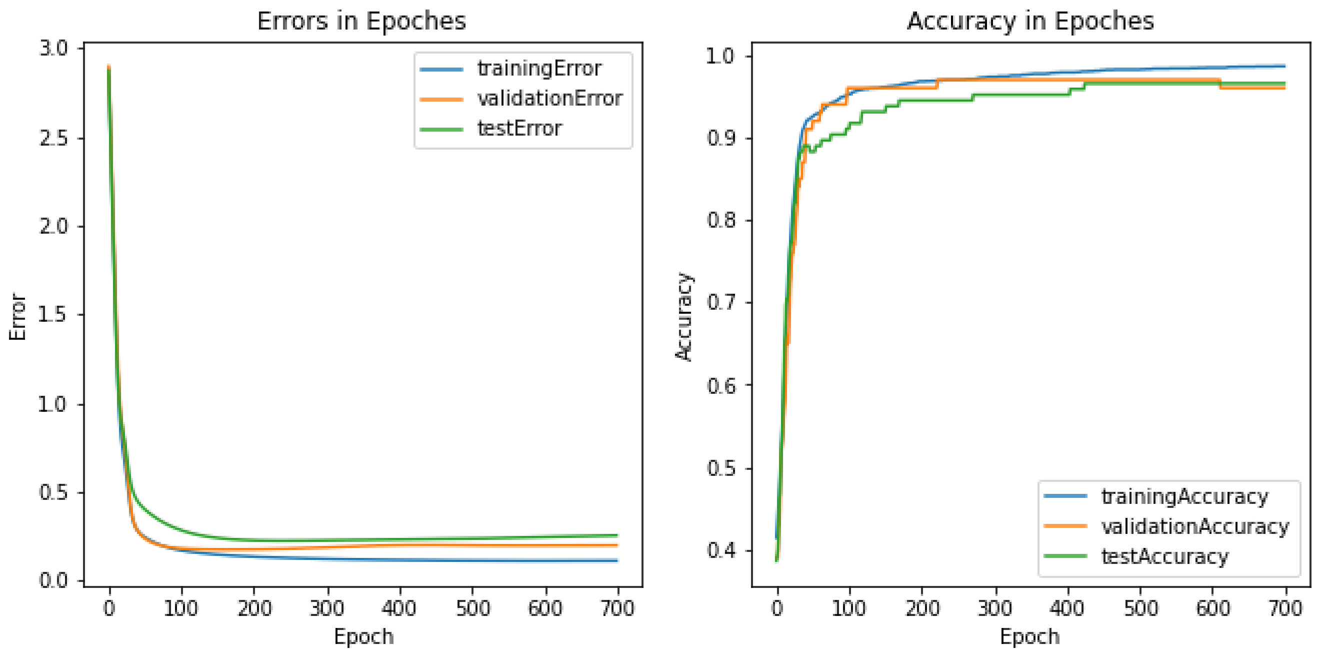


Figure 3.1.5.D - Training Results with Batch Size of 1750 (CE + SGD)

Hyperparameter Investigation

In the case of logistic regression, the differences in training results caused by different hyperparameter values are very small with 700 epoches. All of the models trained give a fairly good validation accuracy around 97% or 98%. The reason behind this situation might be that linear regression models fit this task well so the minor differences in hyperparameter values cannot make huge impacts on the final training results by the end of epoch 700.

β1	0.95	0.99
bias	-0.13203566	0.25422055
Training Accuracy	0.9902857142857143	0.9891428571428571
Validation Accuracy	0.98	0.98
Testing Accuracy	0.9655172413793104	0.9862068965517241
Training Error	0.12421522255700605	0.14275016855822953
Validation Error	0.0736128352378245	0.1131507764597377
Testing Error	0.32462369433256405	0.213273276612504

Table 3.1.5.C - Training Results with different β1 values (CE + SGD)

β2	0.99	0.9999
bias	-0.18558736	-0.041061457
Training Accuracy	0.992	0.9888571428571429
Validation Accuracy	0.98	0.97
Testing Accuracy	0.9793103448275862	0.9793103448275862
Training Error	0.2018285612890073	0.12222597200759895
Validation Error	0.2629107781522179	0.22840552661825517
Testing Error	0.4816635037514372	0.3226154984712807

Table 3.1.5.D - Training Results with different β2 values (CE + SGD)

ε	1e-09	1e-04
bias	0.04464168	-0.08105063
Training Accuracy	0.9897142857142858	0.9891428571428571
Validation Accuracy	0.97	0.98
Testing Accuracy	0.9793103448275862	0.9793103448275862
Training Error	0.1290858757961887	0.12293708093642514
Validation Error	0.045790462951286165	0.10237634610447531
Testing Error	0.13475772376163736	0.16165798467828582

Table 3.1.5.E - Training Results with different β1 values (CE + SGD)

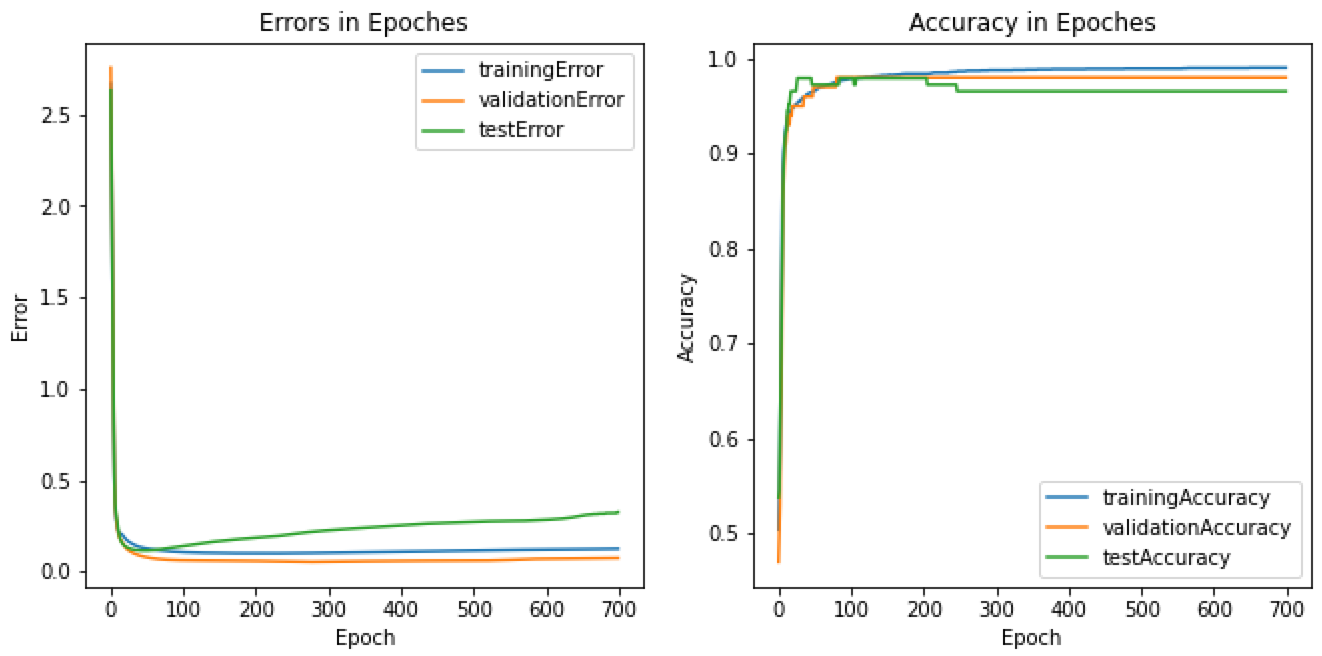


Figure 3.1.5.E - Training Results with β1=0.95 (CE + SGD)

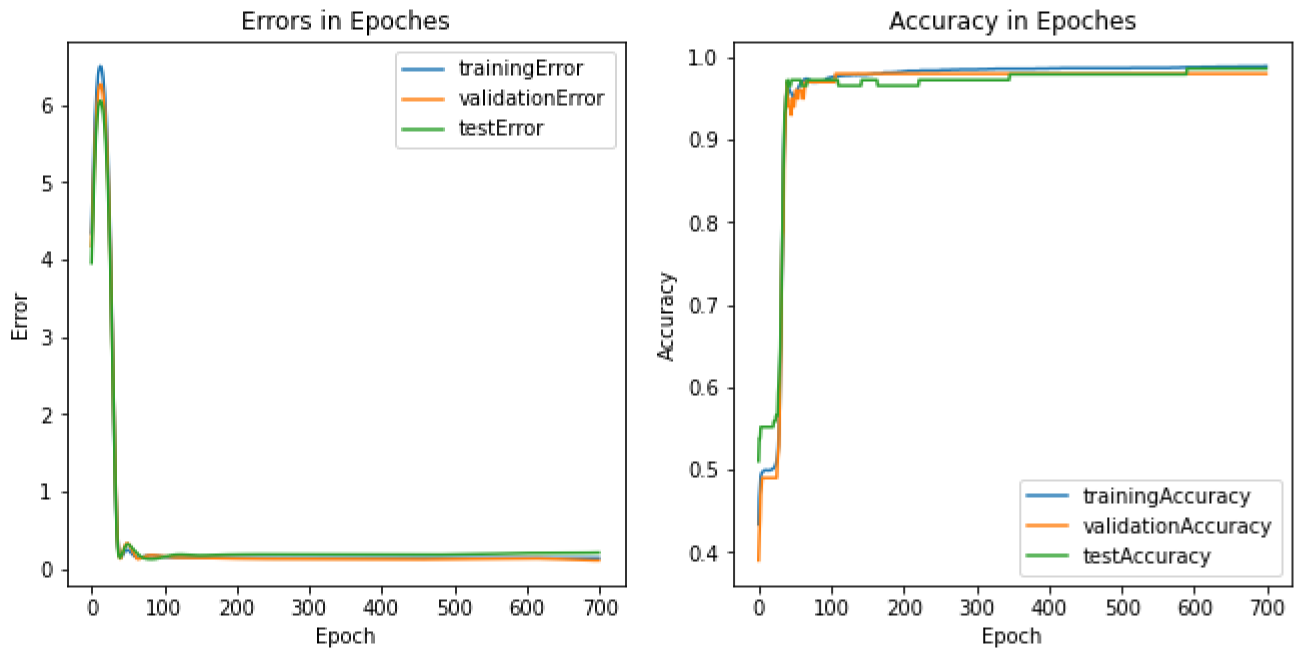


Figure 3.1.5.F - Training Results with β1=0.99(CE + SGD)

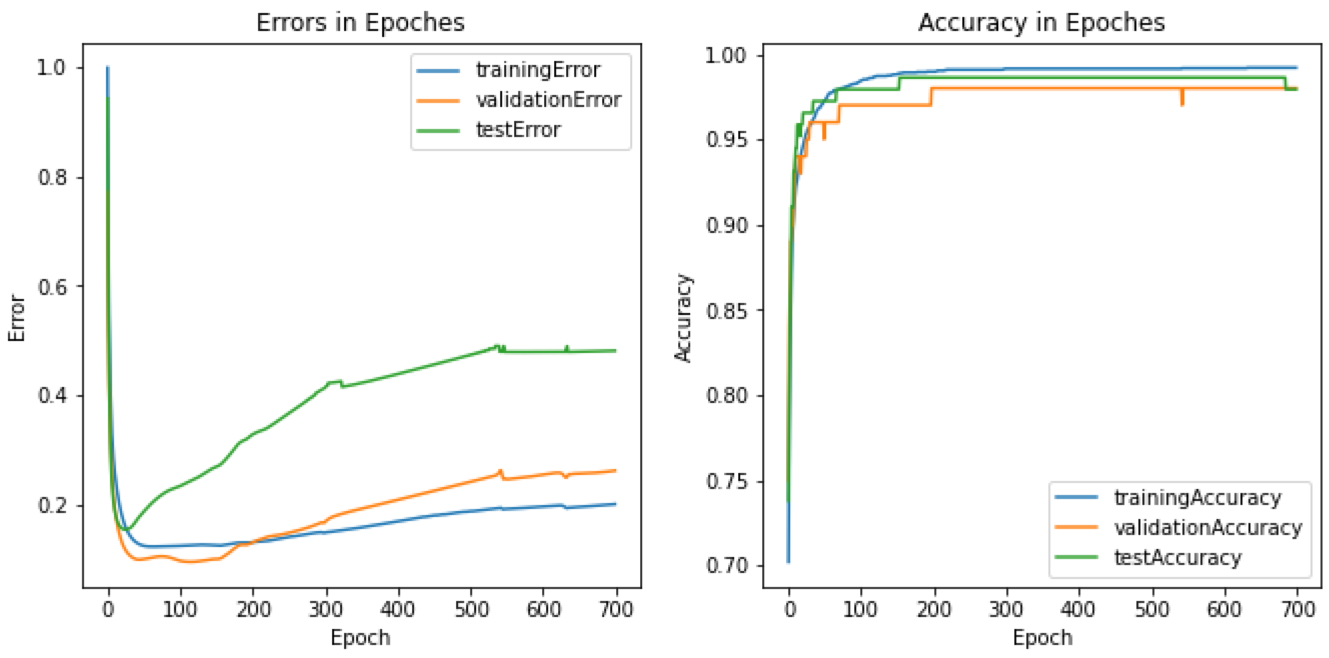


Figure 3.1.5.G - Training Results with β2=0.99(CE + SGD)

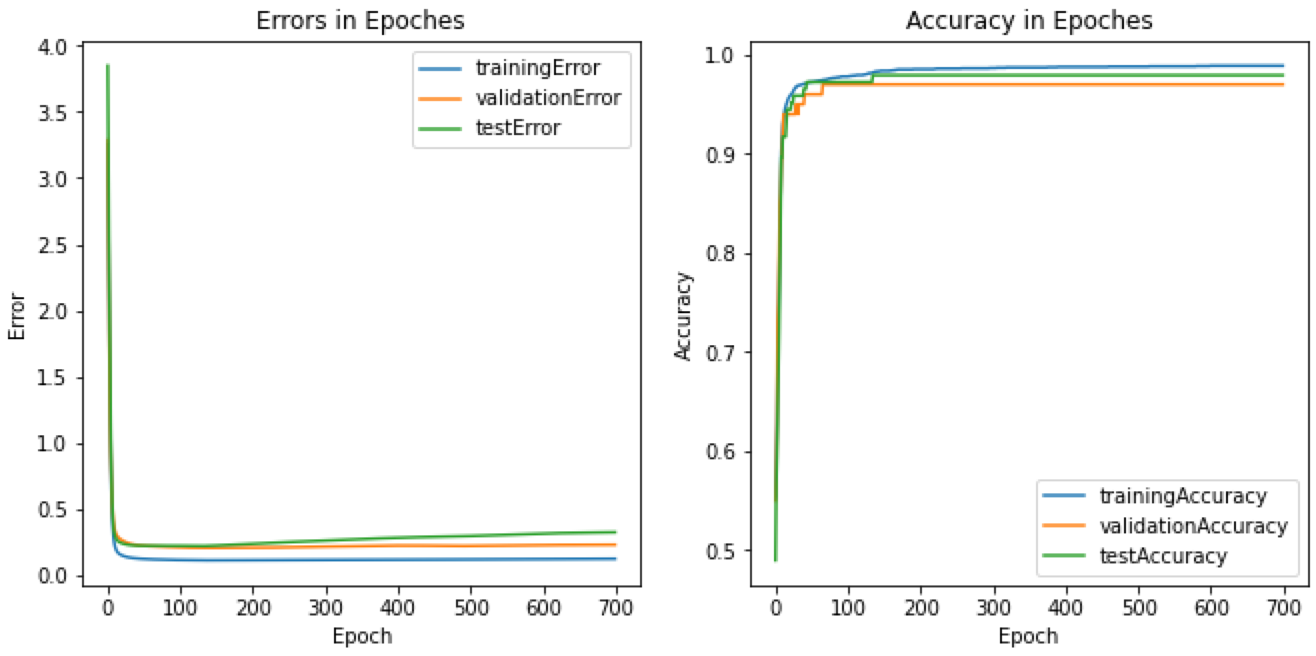


Figure 3.1.5.H - Training Results with β2=0.9999 (CE + SGD)

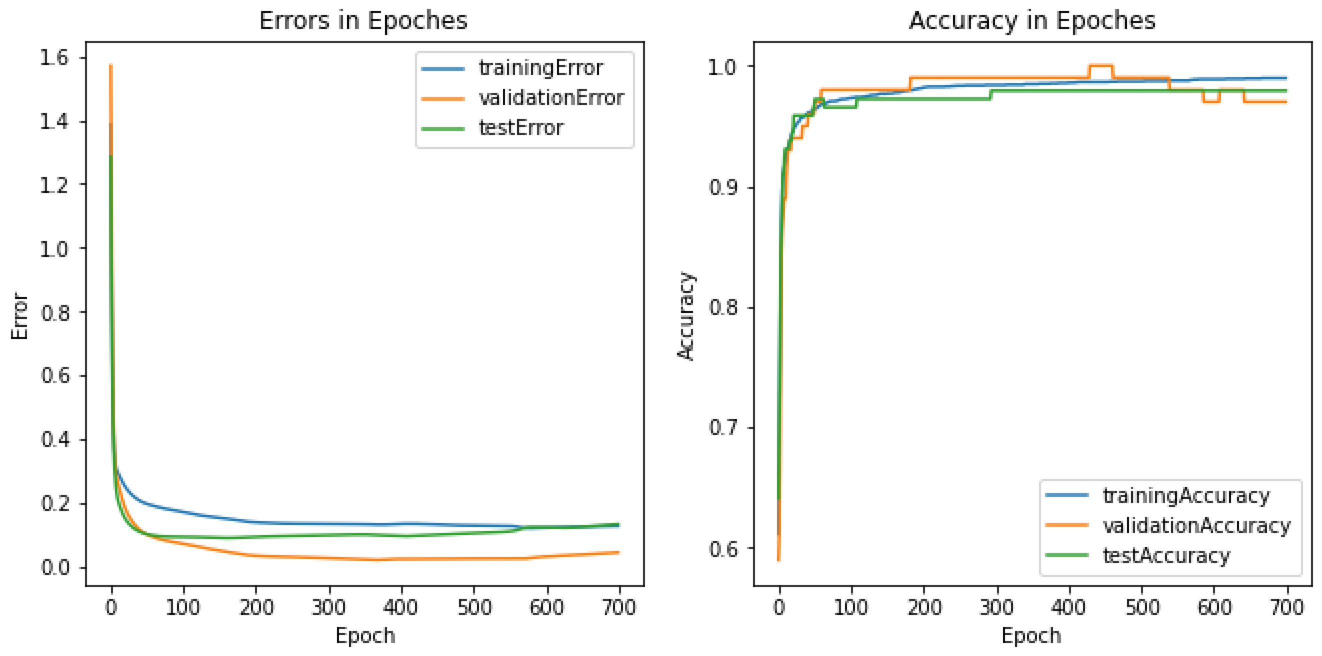


Figure 3.1.5.I - Training Results with ε=1e-09 (CE + SGD)

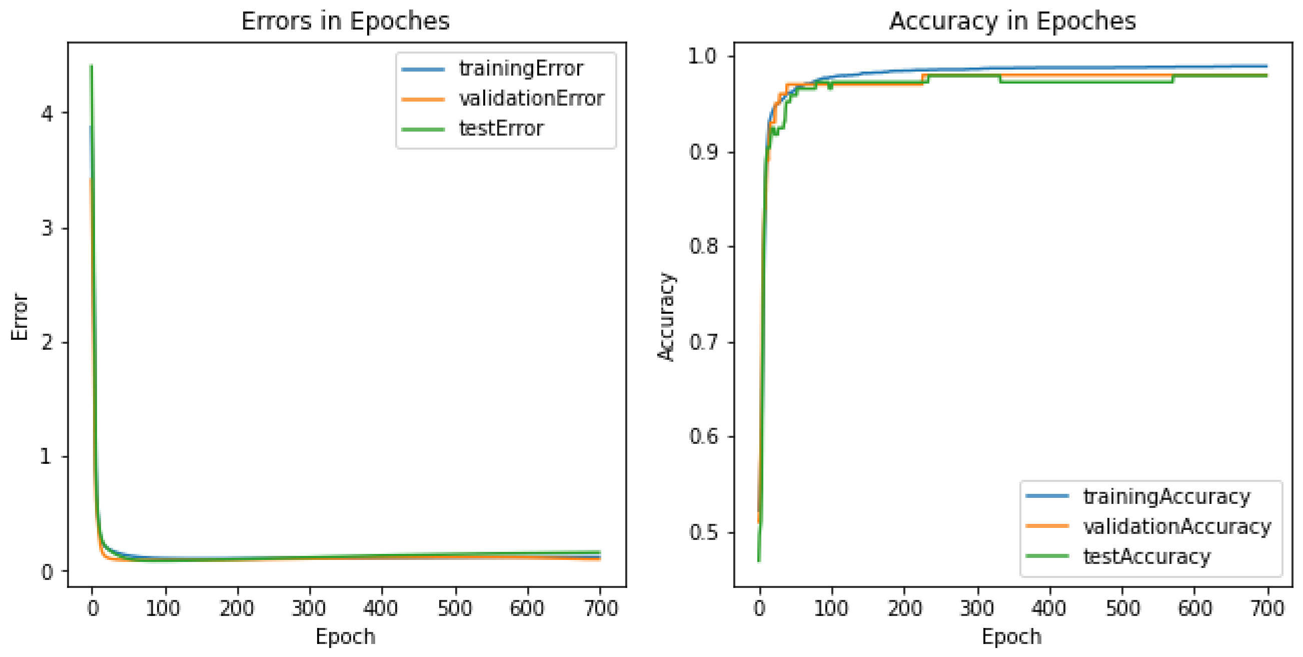


Figure 3.1.5.J - Training Results with ε=1e-04 (CE + SGD)

3.1.6 Comparison against Batch GD

For linear regression by minimizing MSE loss, the performance in terms of final accuracy of the SGD algorithm is slightly worse than the batch gradient descent algorithm (dropping from around 98% to around 92%). For the logistic regression by minimizing CE loss, the accuracy of the SGD algorithm is just slightly dropped and is almost as good as that from the batch gradient descent algorithm (both around 98%).

With respect to the curves, the SGD algorithm converges faster compared to the batch gradient descent algorithm, because by taking the 'momentum' into consideration, the algorithm can find the global minimum faster. SGD with momentum can reduce the chance to get stuck at saddle points and over shallow local miniums, and it leads to faster convergence even for convex functions.