

1. Linear Regression

1. Loss Function and Gradient [4 pts]:

```
25 def MSE(W, b, x, y, reg):
26     y_hat = np.matmul(x, W) + b
27
28     err = np.square(y_hat - y)
29     loss_mse = np.sum(err)/(2*len(y))
30     loss_reg = (np.linalg.norm(W)**2)*reg/2
31     return (loss_mse + loss_reg)
32
33 def gradMSE(W, b, x, y, reg):
34     # grad w.r.t. weights
35     grad_mse = ((np.matmul(np.matmul(x.T, x), W)).reshape(len(W), 1) \
36                 - np.matmul(x.T, y))/len(y)
37     grad_reg = reg * W
38
39     # grad w.r.t. bias
40     grad_bias = np.sum(np.matmul(x, W) + b - y, keepdims=True) / len(y)
41
42     # return tuple of grad w.r.t. weights and bias
43     return (grad_mse + grad_reg), grad_bias
```

Analytical Expressions:

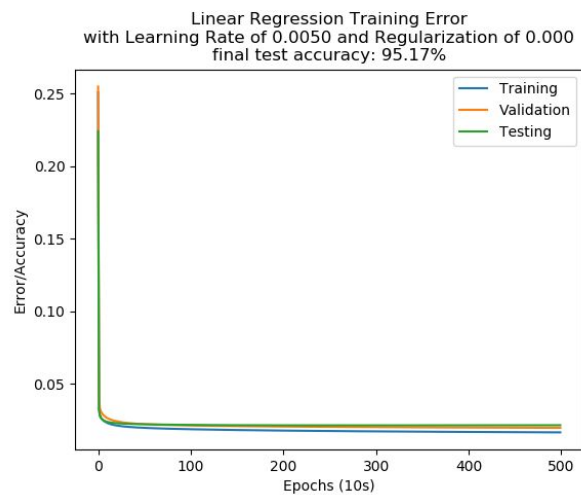
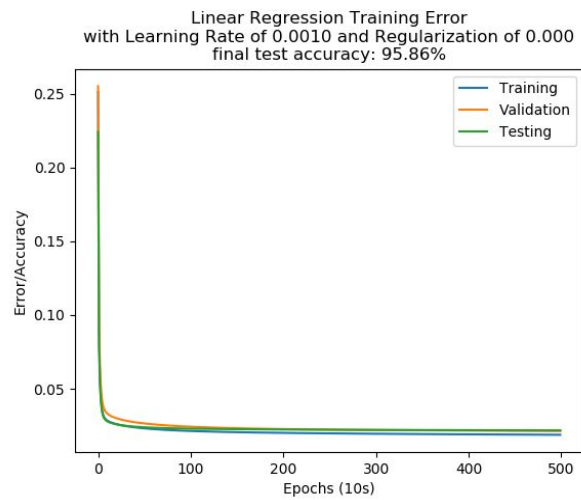
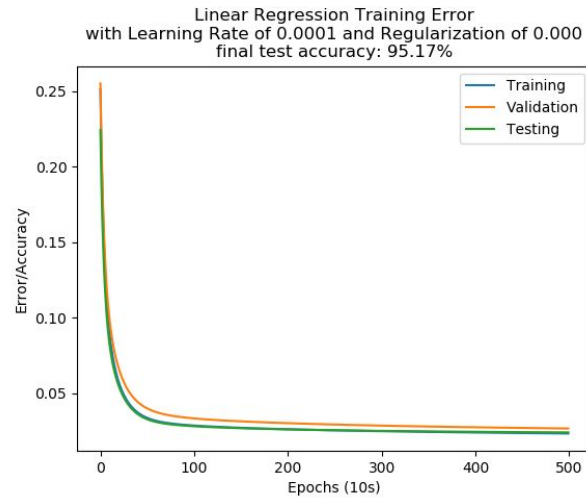
$$MSE = \frac{1}{2N} \sum_{n=1}^N \left\| x^{(n)}W + b - y^{(n)} \right\|_2^2 + \frac{\lambda}{2} \|W\|_2^2$$

$$GRAD(MSE) = \frac{1}{N} (x^T x W - x^T y) + \lambda W$$

2. Gradient Descent Implementation [6 pts]:

```
81 def grad_descent(W, b, trainingData, trainingLabels, alpha, iterations, reg, otherData, EPS=1e-7, loss=MSE, grad=gradMSE):
82     # otherData stores unseen validation dataset
83     plt_train_err = loss(W, b, trainingData, trainingLabels, reg)
84     plt_val_err = loss(W, b, otherData[1], otherData[4], reg)
85
86     W_new = np.copy(W)
87     for i in range(1, iterations):
88         grad_w, grad_b = grad(W, b, trainingData, trainingLabels, reg)
89         W -= alpha * grad_w
90         b -= alpha * grad_b
91
92         # if change in weights is smaller than error tolerance, end descent
93         W_old = np.copy(W_new)
94         W_new = np.copy(W)
95         if(np.sqrt(np.square(np.linalg.norm(W_new-W_old)) + np.square(b)) < EPS):
96             print("hit error tolerance")
97             break
98
99         if i % 10 == 0:
100             plt_train_err = np.append(plt_train_err, loss(W, b, trainingData, trainingLabels, reg))
101             plt_val_err = np.append(plt_val_err, loss(W, b, otherData[1], otherData[4], reg))
102
103     # plot data
104     plot(plt_train_err, plt_val_err, accuracy(W, b, otherData[2], otherData[5]), alpha, reg, True)
105     return W, b
```

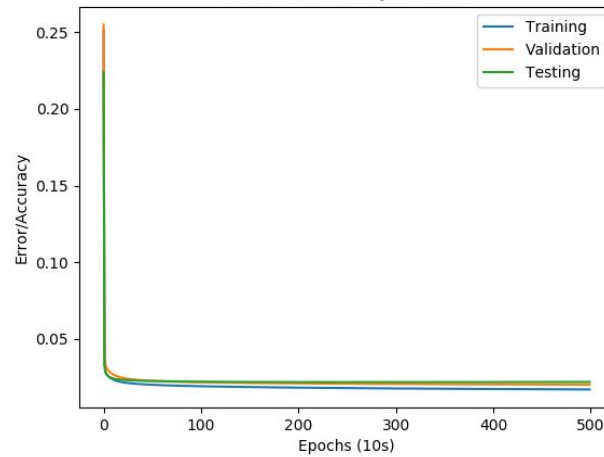
3. Tuning the Learning Rate[3 pts]:



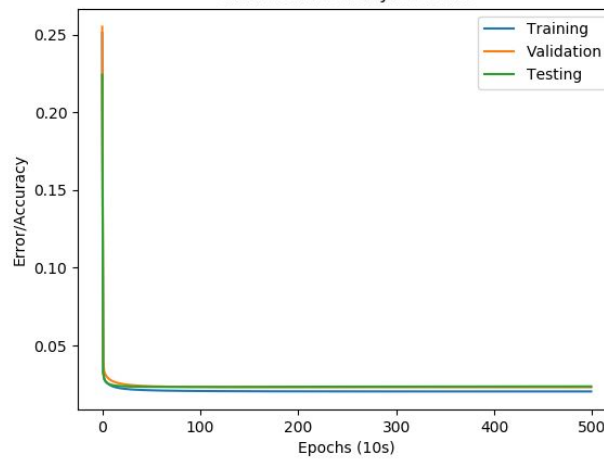
Higher learning rates result in faster convergence to an optimal weight vector. Final test accuracy is 95.17% , 95.86% , and 95.17% for learning rates of 0.005, 0.001, and 0.0001 respectively.

4. Generalization [3 pts]:

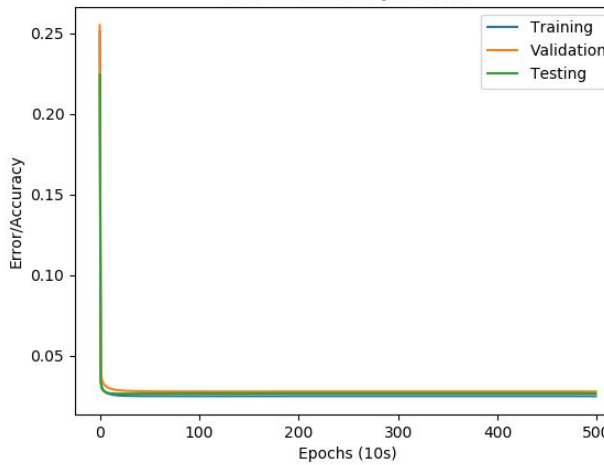
Linear Regression Training Error
with Learning Rate of 0.0050 and Regularization of 0.001
final test accuracy: 95.17%



Linear Regression Training Error
with Learning Rate of 0.0050 and Regularization of 0.100
final test accuracy: 95.17%



Linear Regression Training Error
with Learning Rate of 0.0050 and Regularization of 0.500
final test accuracy: 95.17%



The rationale of tuning regularization with validation is to increase performance on out of sample data by reducing overfitting. In our tests, as lambda increases, the final training, validation, and test error get slightly closer together implying better model generalization. With these values the effects were very minimal and the final test accuracy stays the same. When tested over much higher regularization values the test accuracy drops off significantly which leads us to believe that tuning with regularization would require a larger range of values than those demonstrated above.

5. Comparing Batch GD with normal equation [2 pts]:

```
72 def linear_normal(x, y, reg):
73     # returns optimal weights and bias for binary classification linear regression
74
75     x = np.insert(x, [0], 1, axis=1)
76     W = np.matmul(np.matmul(np.linalg.inv(np.matmul(x.T, x)), x.T), y)
77     return W[1:], W[0]
```

- Test accuracy with the normal equation was 93.80% in 0.48 seconds
- Test accuracy with Gradient Descent (5000 epochs, learning rate 0.001) was 95.86% in 258.95 seconds
- With 500 epochs, learning rate 0.001, no regularization the training rate was still in 95.86% in 26.13 seconds.
- With 100 epochs of the same parameters, 95.17% accuracy was achieved in only 5.6 seconds.

It appears that Gradient Descent achieves a higher test accuracy, though takes more time to train. With this particular dataset not many epochs are needed to surpass the accuracy of the linear normal equation.

2. Logistic Regression

1. Loss Function and Gradient [4 pts]:

```
49 def sigmoid(W, b, x):
50     #return sigmoid activation
51     z = np.matmul(x, W) + b
52     a = 1/(1 + np.exp(-z))
53     return a
54
55 def crossEntropyLoss(W, b, x, y, reg):
56     y_hat = sigmoid(W, b, x)
57
58     log_loss = -(y * np.log(y_hat)) - (1 - y) * np.log(1 - y_hat)
59     loss = np.sum(log_loss) / len(y) + (np.linalg.norm(W)**2)*reg/2
60     loss = np.squeeze(loss)
61     assert(isinstance(loss, float))
62     return loss
63
64 def gradCE(W, b, x, y, reg):
65     y_hat = sigmoid(W, b, x)
66
67     grad_w = np.dot(x.T, y_hat - y) / len(y) + (reg * W)
68     grad_b = np.sum(y_hat - y, keepdims=True) / len(y) + (reg * b)
69
70     # return tuple of grad w.r.t. weights and bias
71     return grad_w, grad_b
```

Analytic expressions:

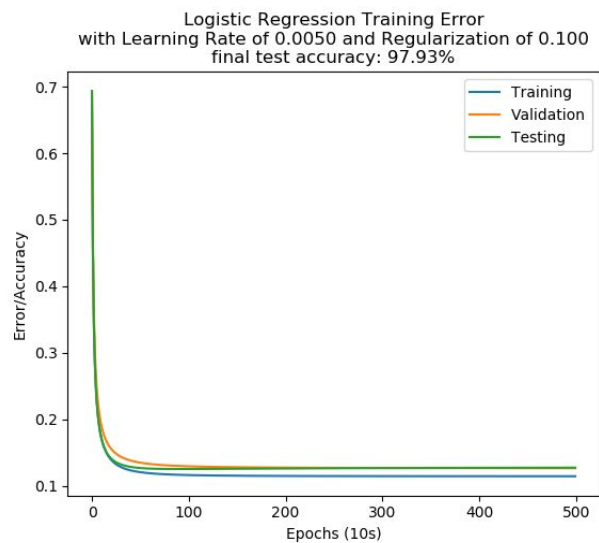
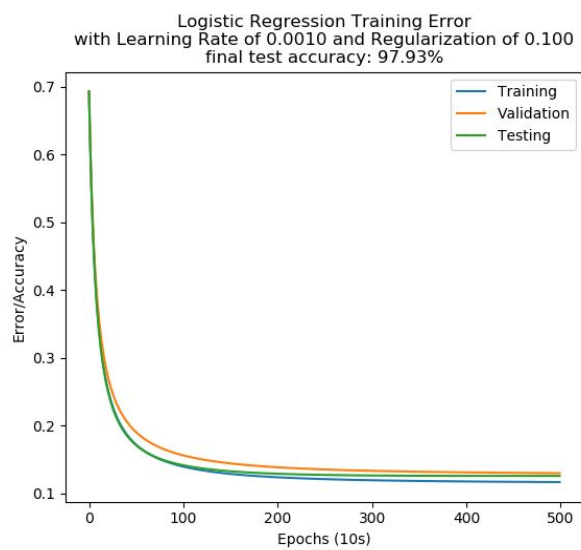
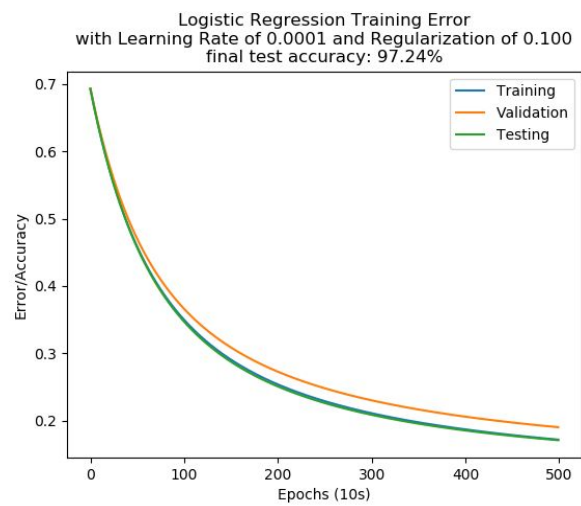
$$CE = \frac{1}{N} \sum_{n=1}^N \left[-y^{(n)} \log \hat{y}(x^{(n)}) - (1 - y^{(n)}) \log(1 - \hat{y}(x^{(n)})) \right] + \frac{\lambda}{2} \|W\|_2^2$$

$$\nabla (CE)_W = \frac{1}{N} x^T (\hat{y} - y) + \lambda W$$

$$\nabla (CE)_b = \frac{1}{N} (\hat{y} - y) + \lambda b$$

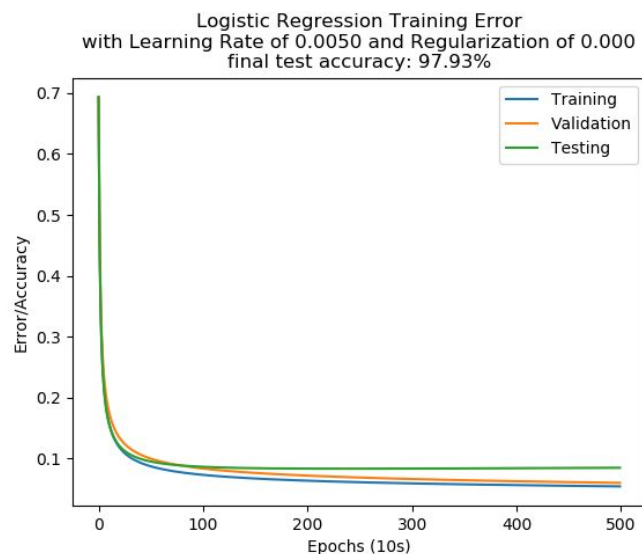
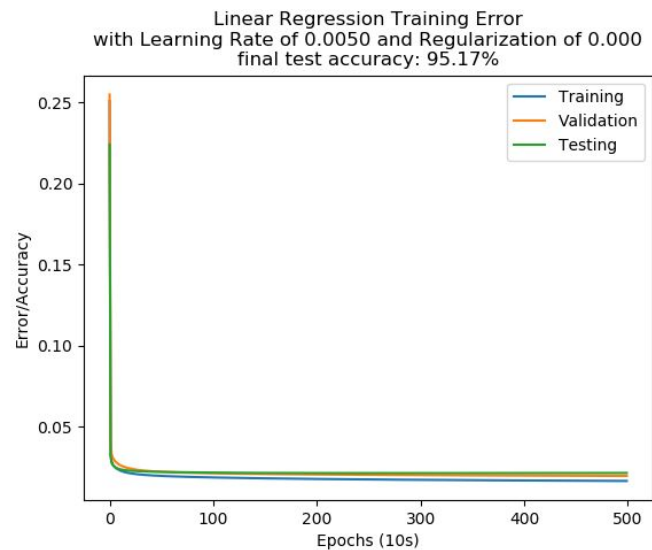
$$\text{Where } \hat{y} = \frac{1}{1 + e^{-xW + b}}$$

2. Learning [4 pts]: Same code snippet as in question 1.2



Using a cross-entropy loss function gradient descent converged to a relatively accurate weight vector but the effects on convergence time were more pronounced than in our linear regression model. The fastest learning rate did not negatively affect accuracy and achieved convergence within fewer epochs.

3. Comparison to Linear Regression [2 pts]:



Though an MSE loss function converged in fewer epochs than a CE loss function, the Cross-Entropy loss function converged to a higher accuracy and it's computation time was much faster.

3. Batch Gradient Descent vs. SGD and Adam

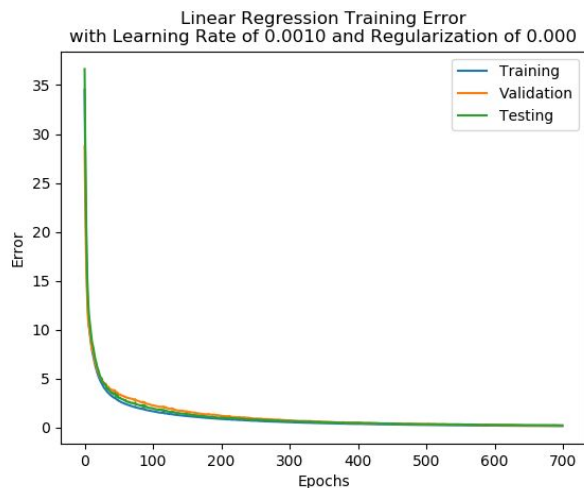
1. Building the Computational Graph [5 pts]:

```
158 def buildGraph(beta1=None, beta2=0.9999, epsilon=None, lossType='MSE', alpha=1e-3):
159     g = tf.Graph()
160     tf.set_random_seed(421)
161
162     # Init weight and bias tensors
163     # weight tensors use tf.truncated_normal w/ std. dev. = 0.5
164     weights = tf.get_variable("weights", [784, 1], initializer=tf.truncated_normal_initializer(stddev=0.5))
165     bias = tf.get_variable("bias", [1], initializer=tf.constant_initializer(0.0))
166
167     # use tf.placeholder to create tensors for data, labels, and reg
168     data = tf.placeholder(tf.float32, name="data")
169     labels = tf.placeholder(tf.uint8, name="labels")
170     reg = tf.placeholder(tf.float32, name="reg")
171
172     predictedLabels = tf.math.sigmoid(tf.matmul(data, weights) + bias)
173     loss = tf.losses.sigmoid_cross_entropy(labels, predictedLabels)
174     optimizer = tf.train.AdamOptimizer(learning_rate=alpha).minimize(loss)
175     accuracy = tf.metrics.accuracy(labels, predictedLabels)
176
177     return weights, bias, data, labels, loss, reg, optimizer, accuracy
```

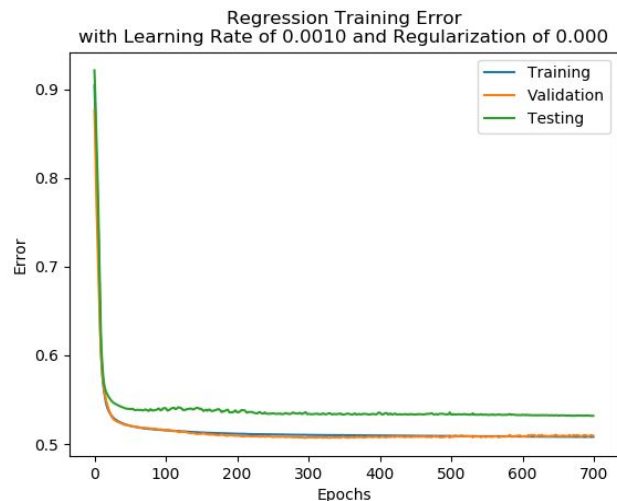
Note that above is code for cross entropy loss.

2. Implementing Stochastic Gradient Descent [5 pts.] I

MSE loss

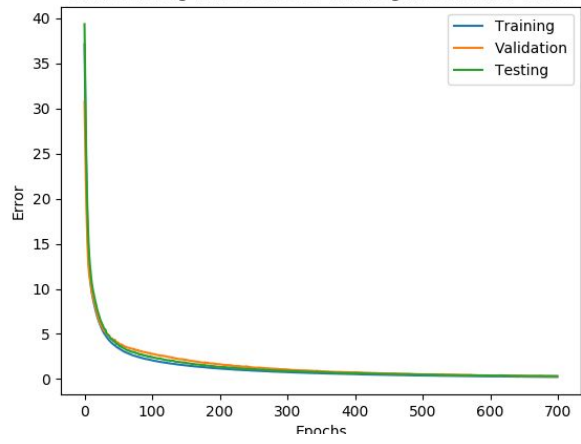
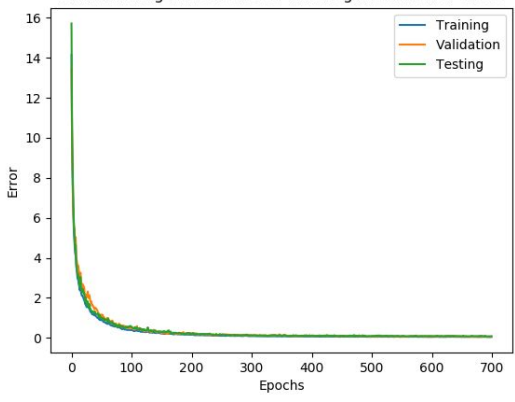


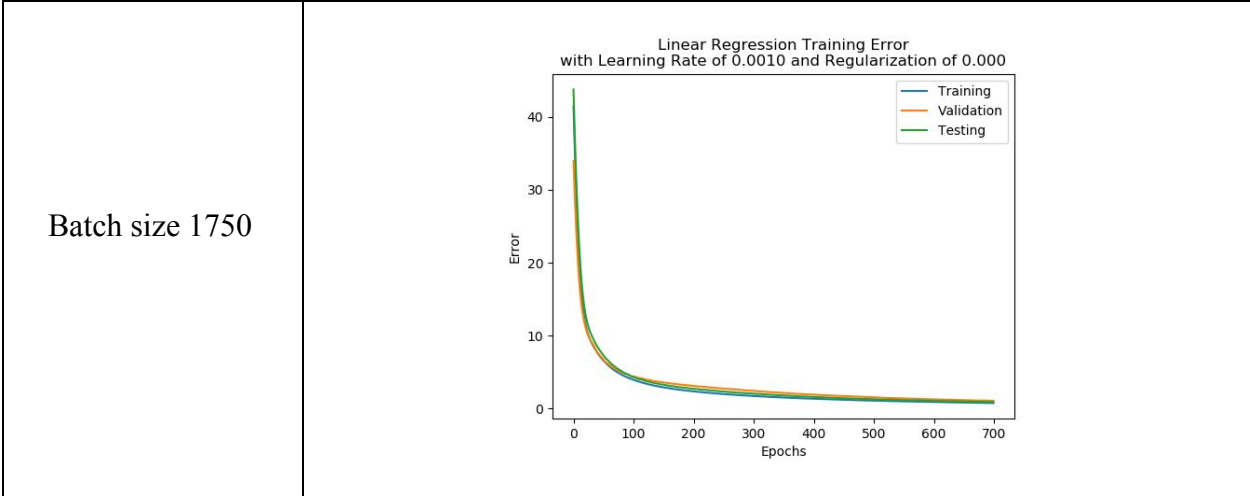
Cross-Entropy loss



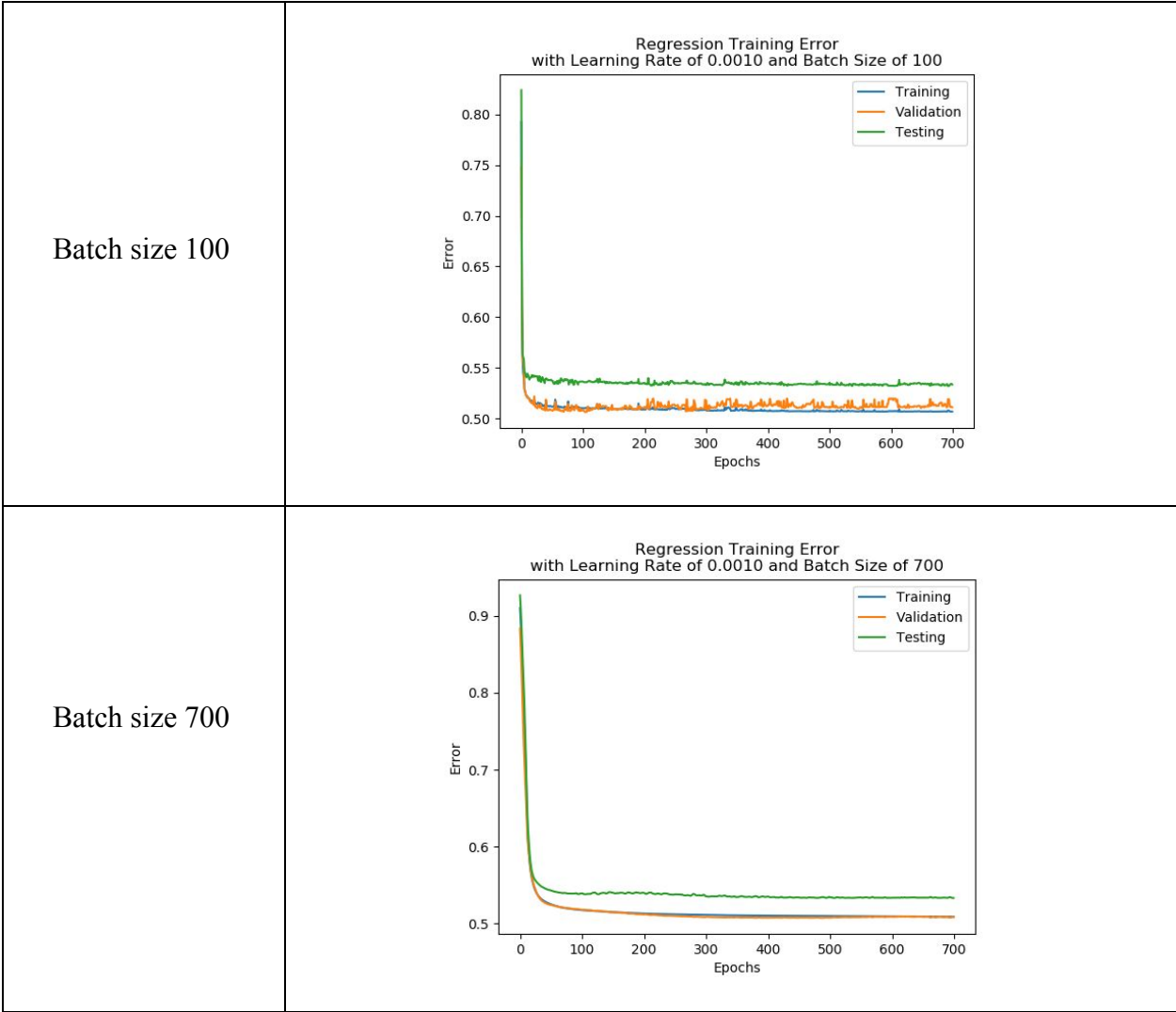
3. Batch Size Investigation [2 pts.] / 5. Cross Entropy Loss Investigation [6 pts.]

MSE loss

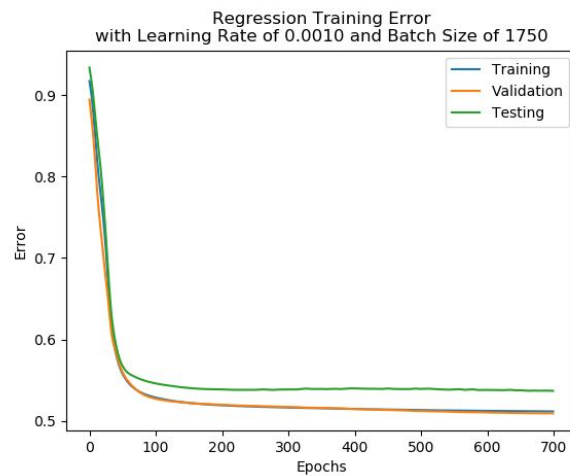
Batch size 100	<p>Linear Regression Training Error with Learning Rate of 0.0010 and Regularization of 0.000</p>  <p>Training Validation Testing</p>
Batch size 700	<p>Linear Regression Training Error with Learning Rate of 0.0010 and Regularization of 0.000</p>  <p>Training Validation Testing</p>



Cross-Entropy loss



Batch size 1750

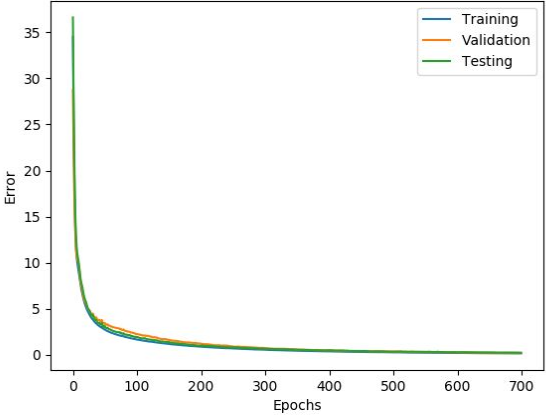
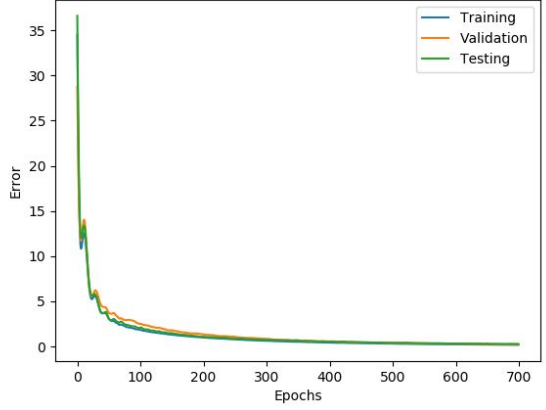
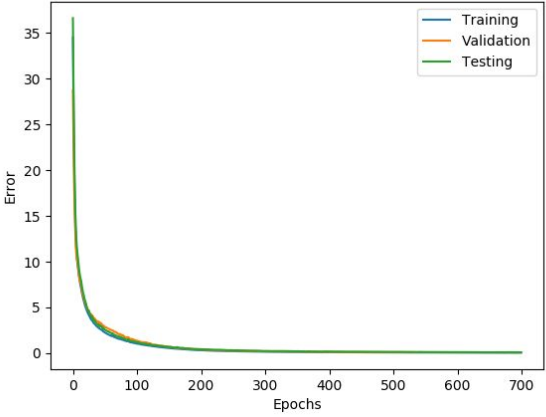


Final classification accuracy is nearly identical for all batch sizes after many epochs. Larger batch sizes have an extremely small advantage. This is because the larger batches end with batch gradients that are closer to resembling the average (expected) gradient of the entire dataset, with smaller batches oscillation occurs due to the random sampling of input data creating gradients further from the expected direction.

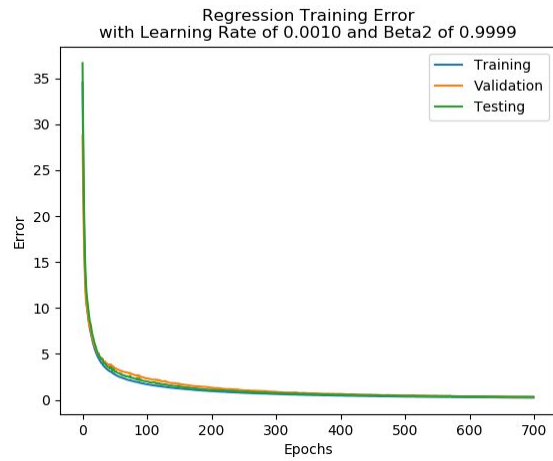
Smaller batches still converge to an appropriate test accuracy and can potentially reduce overfitting, while also increasing computation speed.

4. Hyperparameter Investigation [4 pts.] / 5. Cross Entropy Loss Investigation [6 pts.]

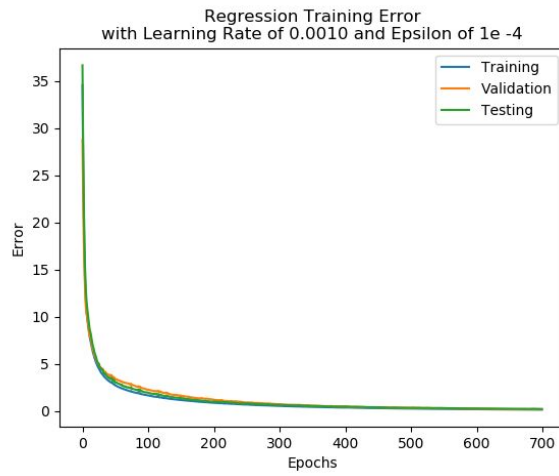
MSE Loss

<p>Beta1 = 0.95 Final accuracy is good, descent is smooth</p>	<p>Regression Training Error with Learning Rate of 0.001 and Beta1 of 0.95</p> 
<p>Beta1 = 0.99 Final accuracy is good, descent is noisy because momentum weight is increased so earlier incorrect stochastic gradients are weighted too heavily</p>	<p>Regression Training Error with Learning Rate of 0.0010 and Beta1 of 0.99</p> 
<p>Beta2 = 0.99 Final accuracy is good, Converges quickly</p>	<p>Regression Training Error with Learning Rate of 0.0010 and Beta2 of 0.99</p> 

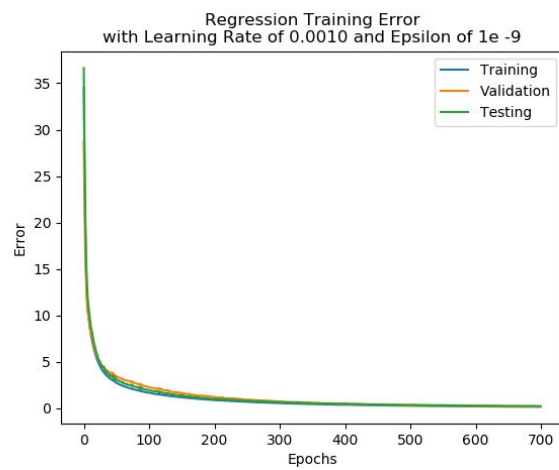
Beta2 = 0.999
Final accuracy is good,
Convergence slightly
slowed down.



Epsilon = 1e-4
Converges well,
Effect is negligible

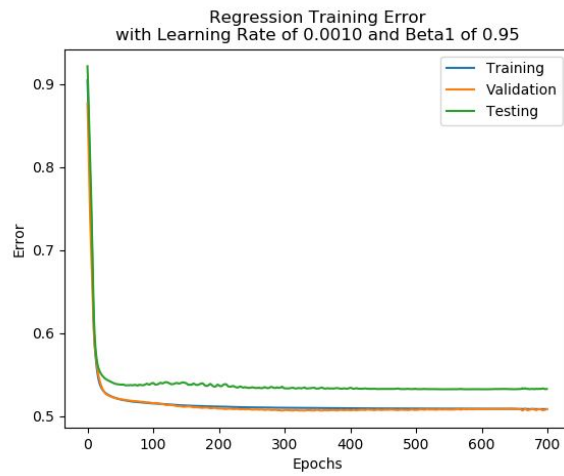


Epsilon = 1e-9
Converges well,
Effect is negligible

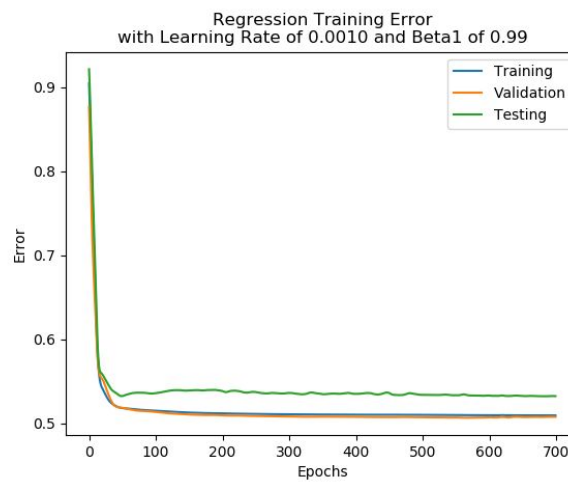


Cross-Entropy Loss

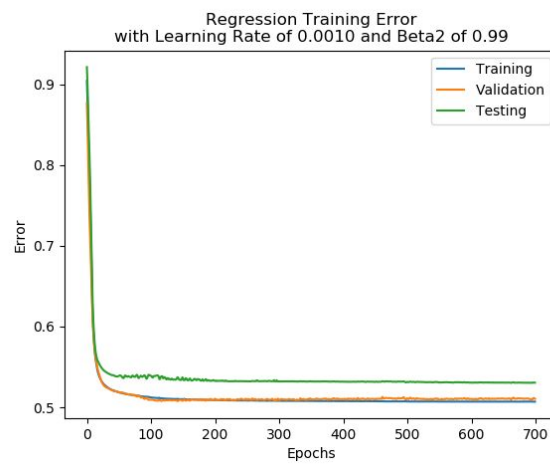
Beta1 = 0.95



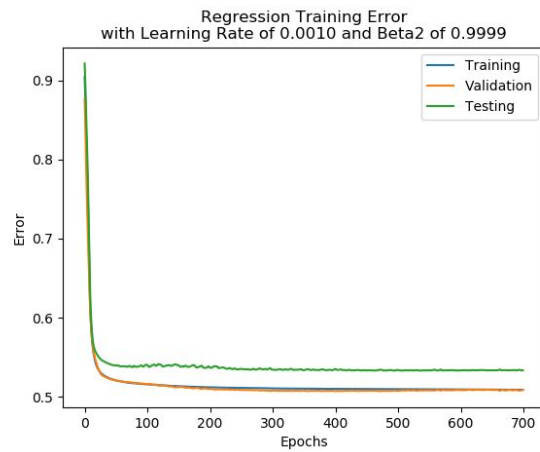
Beta1 = 0.99



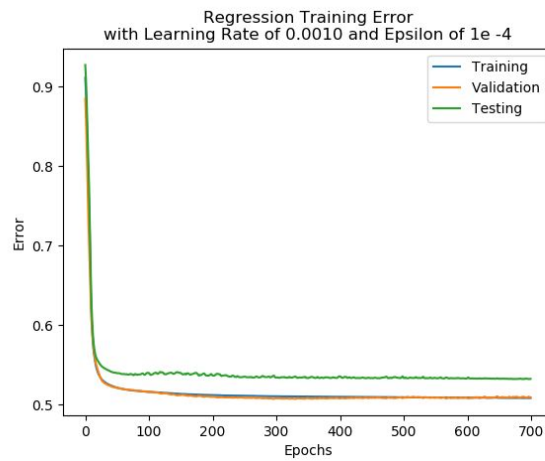
Beta2 = 0.99



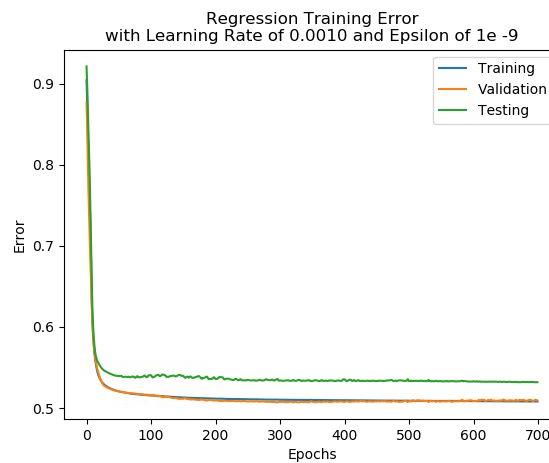
Beta2 = 0.999



Epsilon = 1e-4



Epsilon = 1e-9



5. See sections 3.1.2 and 3.1.4 for graphs.

Final classification error is slightly better when optimizing Cross-Entropy loss. In addition to improved accuracy, computation time is also improved on our computer.

6. Comparison against Batch GD [3 pts.]

The SGD algorithm with ADAM converged to a similar accuracy as the batch gradient descent, however it ran significantly faster (likely in part due to the use of tensorflow library).

The final training accuracies were slightly higher for batch gradient descent than SGD since SGD creates oscillation near the minimum because each mini-batch gradient is a stochastic approximation of the entire dataset gradient. This impact is lessened on the validation and test data because an extremely accurate training set performance may be indicative of overfitting.

The training curves for SGD are much noisier than the batch gradient descent curves; this is because SGD finds the gradient of a fraction of the batch and then tests on the entire batch. The mini-batch gradient may differ from the dataset gradient causing training error to increase on some iterations if the weights change in the wrong direction. In the long term, however, the mini-batch gradient has an expected value equal to the dataset gradient and still converges to a minimum.