Linear Regression

1. Loss Function and Gradient

In the following questions, we assume w is a vector of weights as the dataset is dealing with binary classification.

Analytical Expression for MSE loss

$$L = \frac{1}{2N}\|X\mathbf{w} + b\mathbf{1} - \mathbf{y}\|_2^2 + \frac{\lambda}{2}\|\mathbf{w}\|_2^2$$

Analytical Expression for the gradient of the MSE loss

Let $\mathbf{z} = X\mathbf{w} + b\mathbf{1} - \mathbf{y}$

The gradient with respect to the weight vector:

$$\implies L = \frac{1}{2N}\mathbf{z}^T\mathbf{z} + \frac{\lambda}{2}\mathbf{w}^T\mathbf{w}$$

$$\implies \frac{\partial L}{\partial \mathbf{w}} = \frac{1}{2N}\frac{\partial \mathbf{z}}{\partial \mathbf{w}}\frac{\partial \mathbf{z}^\mathbf{T}\mathbf{z}}{\partial \mathbf{z}} + \frac{\lambda}{2}\frac{\partial \mathbf{w}^T\mathbf{w}}{\partial \mathbf{w}} = \frac{1}{2N}\left(X^T\right)(2\mathbf{z}) + \frac{\lambda}{2}2\mathbf{w}$$

$$\implies \frac{\partial L}{\partial \mathbf{w}} = \frac{1}{N}X^T\left(X\mathbf{w} + b\mathbf{1} - \mathbf{y}\right) + \lambda\mathbf{w}$$

Similarly the gradient with respect to the bias scalar:

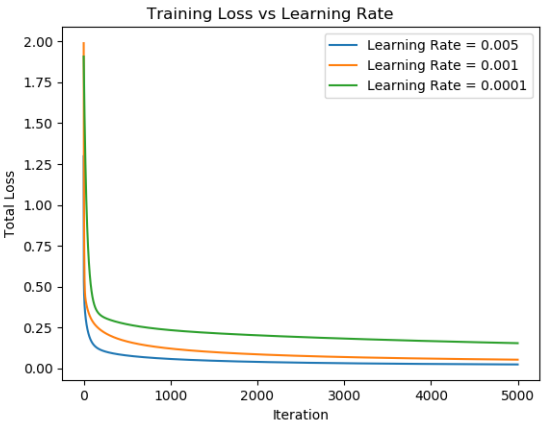
$$\implies \frac{\partial L}{\partial b} = \frac{1}{2N}\frac{\partial \mathbf{z}}{\partial b}\frac{\partial \mathbf{z}^\mathbf{T}\mathbf{z}}{\partial \mathbf{z}} = \frac{1}{2N}\left(\mathbf{1}^T\right)(2\mathbf{z}) = \frac{1}{N}\mathbf{1}^T\left(X\mathbf{w} + b\mathbf{1} - \mathbf{y}\right)$$

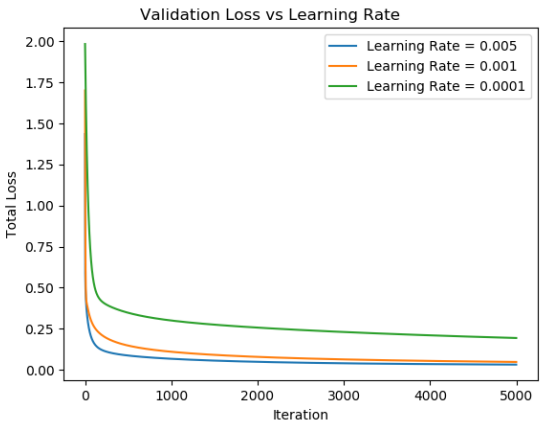
* Code for the MSE and grad_MSE functions can be found in the Appendix - **starter.py** and **GradientDescent_Numpy.py**

2. Gradient Descent Implementation

* Code for the grad_descent function can be found in the Appendix - **GradientDescent_Numpy.py**

3. Tuning the Learning Rate





Plots of the training, validation, and test losses for different learning rates

Learning Rate		Training Data	Validation Data	Test Data
0.005	**Final Loss**	0.02460964	0.0312873	0.02969203
	Accuracy	0.95914285	0.96	0.95862068
0.001	**Final Loss**	0.05380669	0.04740968	0.05117336
	Accuracy	0.91257142	0.97	0.93793103
0.0001	**Final Loss**	0.15476373	0.19326778	0.18222377
	Accuracy	0.77285714	0.71	0.8056

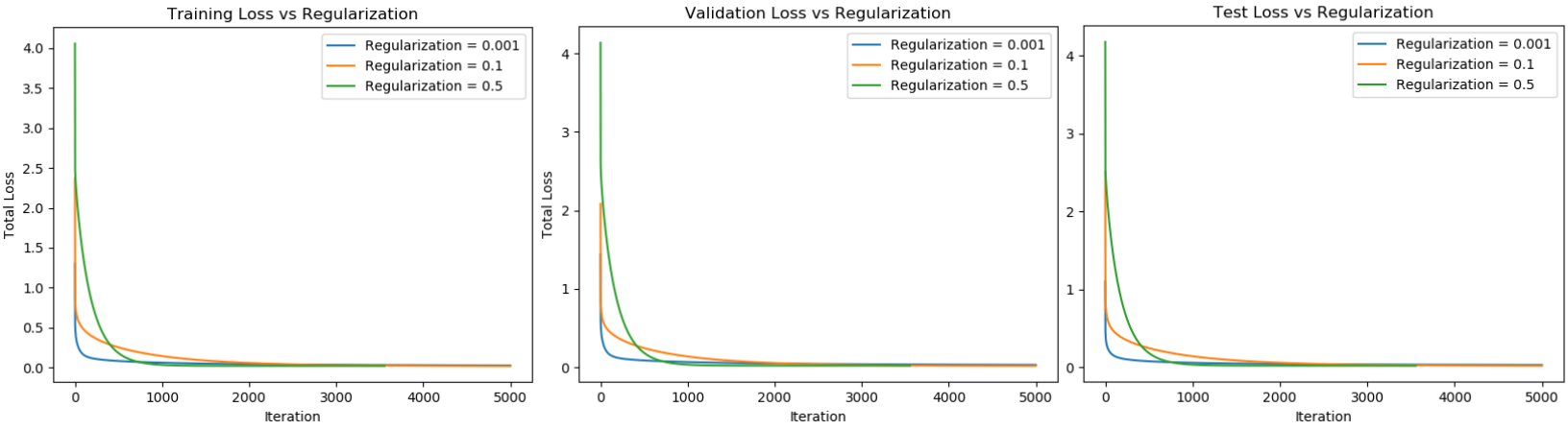
What is the impact of modifying α against the training time?

The smaller the learning rate, the slower the gradient descent algorithm converges (reaches a local minimum). The larger the learning rate, the faster a minimum will be reached. However, the learning step may overshoot the minimum and fail to accurately converge, or may take longer to converge as it jumps between the minimum value.

Final classification accuracy?

Learning rate of 0.005 produced the best final classification accuracy for both the training and test data at around 95.9%. The learning rate of 0.001 coincidentally produced the best validation data classification accuracy. The learning rate of 0.0001 results in the lowest accuracy. This is probably because such a small learning rate makes the gradient descent algorithm converge at a much slower pace.

4. Generalization



Plots of the training, validation, and test losses with a learning rate of 0.005 using different weight decay parameters

Regularization		Training Data	Validation Data	Test Data
0.001	Total Loss	0.02645143	0.03299330	0.03149337
	Accuracy	0.95942857	0.97	0.95862068
0.1	Total Loss	0.01831090	0.01950200	0.02052183
	Accuracy	0.97685714	0.98	0.96551724
0.5	Total Loss	0.02041257	0.02249751	0.02131115
	Accuracy	0.97114285	0.97	0.96551724

Comment on the effect of regularization on performance as well as the rationale behind tuning λ using the validation set.

In general, a lower regularization parameter will tend to 'overfit' to the training data as it does not penalize outlier (larger) weights. This may cause the validation and test losses to be higher than the training loss, and similarly result in lower validation and test accuracies.

In our data collected above we see that for the smallest regularization parameter (0.001), the difference between the test and training losses are ~0.005. The difference in test/training loss for the largest regularization parameter (0.5) is ~0.0009. This shows that the difference in test/training loss for the highest regularization parameter is almost 5 times as small as for the smallest regularization parameter, and this data supports the idea that lower regularization parameters cause overfitting and poor model generalization.

We tune the regularization parameter using the validation set because we want to fit the model on data we have not trained on. If we were to tune the weight decay on the training data we would again be overfitting the non-linear model to the training data and this may reduce the accuracy of the 'out of sample' (test) data.

5. Batch GD vs Normal Equation

When calculating the Normal Equation I have added the bias term as the first weight term (w_0 = b) and set x_0 = 1. The Normal Equation is then given as:

$$\frac{1}{N} X^T \left(X\mathbf{w} - \mathbf{y} \right) + \lambda \mathbf{w} = \mathbf{0}$$
$$\implies \mathbf{w} = \left(X^T X + N\lambda I \right)^{-1} X^T \mathbf{y}$$

* Code for the normal equation can be found in Appendix - **starter.py**

Measured performance of both batch gradient descent and the normal equation using a learning rate of 0.005, with 0 weight decay, and 5000 iterations

Performance Measurement	**Batch Gradient Descent**	**Normal Equation**
Training MSE Loss	0.02460964	0.00935202
Training Accuracy	0.95914285	0.99371428
Computation Time (seconds)	126.99172	0.1515473

Compare in terms of final training MSE, accuracy and computation time between Batch GD and normal equation.

The normal equation produces lower (almost 0) MSE loss compared to the batch gradient descent. This is because the normal equation is a perfect analytical solution to the optimization problem and will always find the minimum loss. However, loss found through batch gradient descent is also very low and for most real world cases is enough.

Similarly the normal equation produces a higher accuracy (almost 100%) compared to the batch gradient descent. This is again because the normal equation is a perfect analytical solution that would find the minimal loss. If this data were linearly separable the training accuracy would be 100%. Regardless, the batch gradient descent boasts an accuracy of 96% which is still very high.

The normal equation is significantly faster to compute for this dataset. However, for larger datasets, the batch gradient descent may be faster to compute as it has a complexity of O(Nd) per iteration vs the O(Nd^2) complexity of the Normal Equation.

Finally the Normal Equation does not prevent overfitting of the training data. This means that while it is possible to reach extremely high training accuracies, the normal equation does not generalize the model onto out-of-sample data sets.

In conclusion it is because of complexity of large datasets and generalization that gradient descent is the more practical optimization technique.

Logistic Regression

1. Loss Function and Gradient

Let $z_n = \mathbf{w}^T\mathbf{x}_n + b = \sum_1^d w_j X_{nj} + b$ where $X = [\mathbf{x}_1 \dots \mathbf{x}_N]^T$

$\mathbf{z} = X\mathbf{w} + b\mathbf{1}$

Analytical Expression for MSE loss

$$L = \frac{1}{N}\sum_1^N \left[-y_n \ln(\sigma(z_n)) - (1-y_n)\ln(1-\sigma(z_n))\right] + \frac{\lambda}{2}\|\mathbf{w}\|_2^2$$

$$L = \frac{1}{N}\sum_1^N \left[(1-y_n)z_n + \ln(1+e^{-z_n})\right] + \frac{\lambda}{2}\|\mathbf{w}\|_2^2$$

$$L = \frac{1}{N}\left[(\mathbf{1}-\mathbf{y})^T\mathbf{z} + \mathbf{1}^T\ln(\mathbf{1}+e^{-\mathbf{z}})\right] + \frac{\lambda}{2}\|\mathbf{w}\|_2^2$$

The gradient with respect to the weight vector:

$$\frac{\partial L_D}{\partial w_i} = \frac{1}{N}\frac{\partial}{\partial w_i}\sum_1^N \left[(1-y_n)z_n + \ln(1+e^{-z_n})\right]$$

$$= \frac{1}{N}\sum_1^N \left[(1-y_n)\frac{\partial z_n}{\partial w_i} + \frac{1}{1+e^{-z_n}}e^{-z_n}\left[-\frac{\partial z_n}{\partial w_i}\right]\right] = \frac{1}{N}\sum_1^N \left[(1-y_n)X_{ni} - \frac{1}{1+e^{z_n}}X_{ni}\right]$$

Since $X^T = [\mathbf{x}_1 \dots \mathbf{x}_N] = [X_{ni}]$ we can remove the summation and vectorize into

$$\frac{\partial L}{\partial \mathbf{w}} = \frac{1}{N}\left[X^T(\mathbf{1}-\mathbf{y}) - X^T\frac{1}{1+e^{\mathbf{z}}}\right] + \lambda\mathbf{w} = \frac{1}{N}X^T\left(\mathbf{1}-\mathbf{y}-\frac{1}{1+e^{\mathbf{z}}}\right) + \lambda\mathbf{w}$$

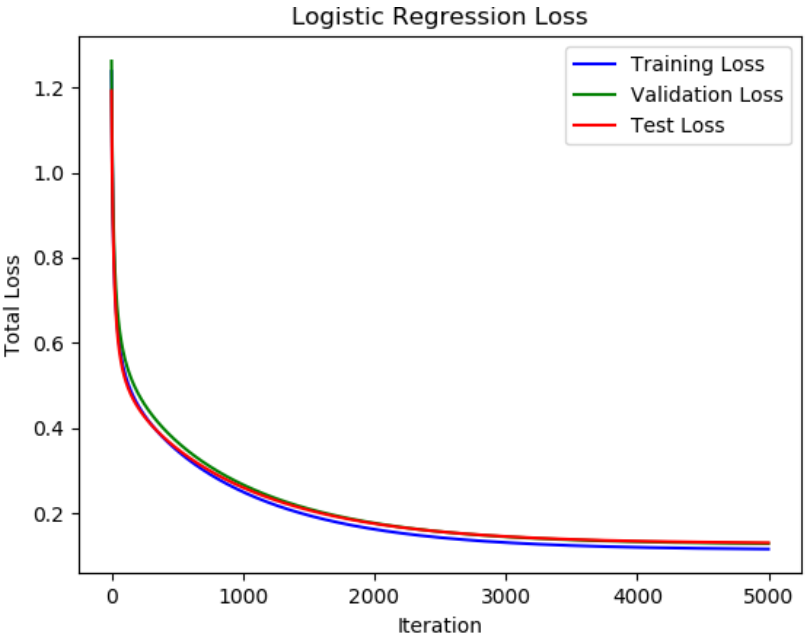
The gradient with respect to the bias scalar:

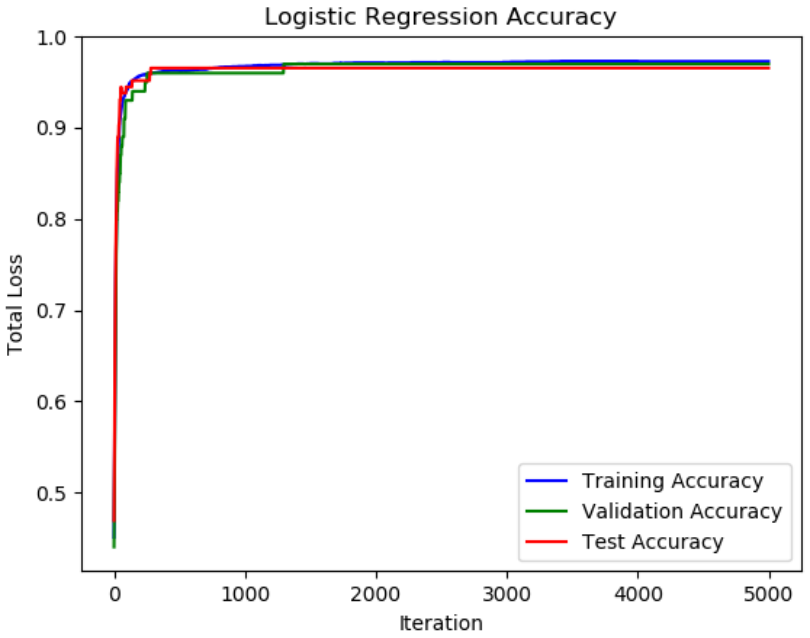
$$\frac{\partial L}{\partial b} = \frac{1}{N}\sum_1^N \left[(1-y_n)\frac{\partial z_n}{\partial b} + \frac{1}{1+e^{-z_n}}e^{-z_n}\left[-\frac{\partial z_n}{\partial b}\right]\right] = \frac{1}{N}\sum_1^N \left[(1-y_n) - \frac{1}{1+e^{z_n}}\right]$$

$$= \frac{1}{N}\mathbf{1}^T\left[\mathbf{1}-\mathbf{y}-\frac{1}{1+e^{\mathbf{z}}}\right]$$

* Code for crossEntropyLoss and gradCE can be found in the Appendix - **starter.py** & **GradientDescent_Numpy.py**

2. Learning





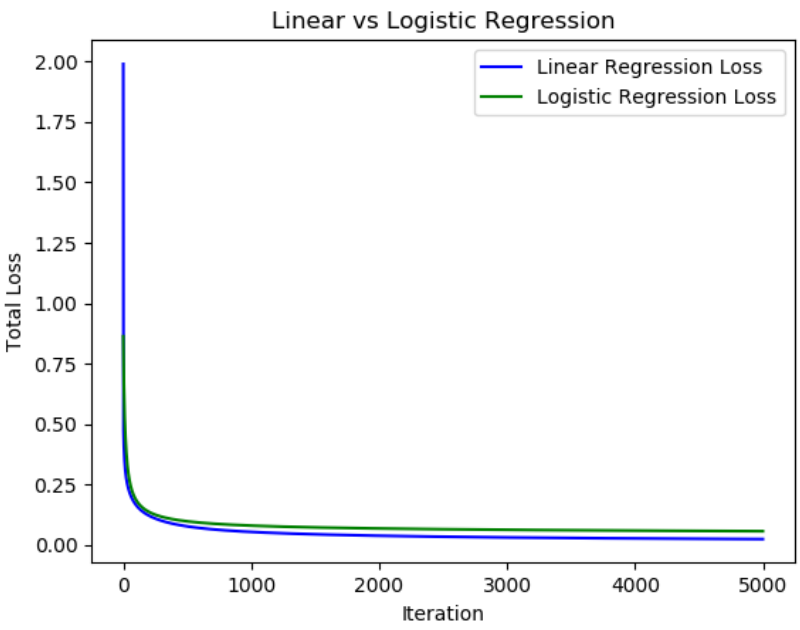
For weight decay = 0.1, learning rate = 0.005, and 5000 iterations, the Loss and Accuracy curves are plotted above.

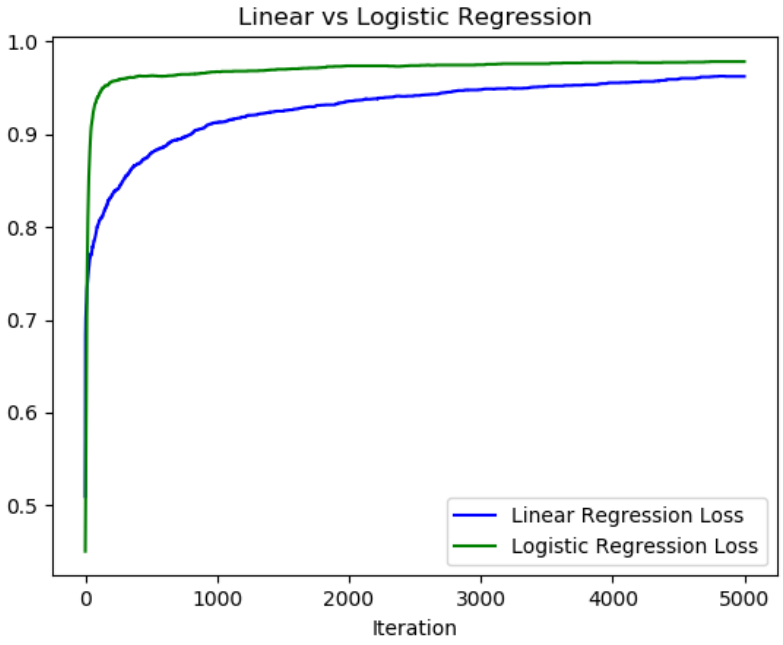
	Training Data	**Validation Data**	**Test Data**
Total Loss	0.11640431	0.12934503	0.13105944
Accuracy	0.97028571	0.97	0.96551724

The classification accuracy for all 3 data sets are very high at around 97%.

* Code for grad_descent can be found in the Appendix - **GradientDescent_Numpy.py**

3. Linear vs Logistic Regression





Linear vs Logistic Regression Loss and Accuracy plotted above with
Weight Decay = 0, Learning Rate = 0.005, 5000 iterations

	Training Loss	**Final Training Accuracy**
Linear Regression	0.02416616	0.96228571
Logistic Regression	0.05690108	0.97828571

Comment on the effect of cross-entropy loss convergence behaviour.

The loss data seems to suggest that Logistic Regression Cross-Entropy converges faster for binary classification learning. This is evident when we look at the slopes of the Linear and Logistic Regression losses; it seems that for a given iteration (>1000) the slope of the Logistic Regression CE loss is flatter and therefore is closer to converging to a minimum (through error tolerance) than Linear Regression MSE is.

In terms of the mathematics, the MSE is the loss function used to fit a Gaussian distributed target, while CE is the loss function used to fit a Bernoulli distributed target. Since we are dealing with a binomial classification problem using the cross-entropy loss function is the better choice, and as shown in the accuracy curves produces the better performance.

Batch GD vs Stochastic GD with ADAM

1. Building the Computational Graph

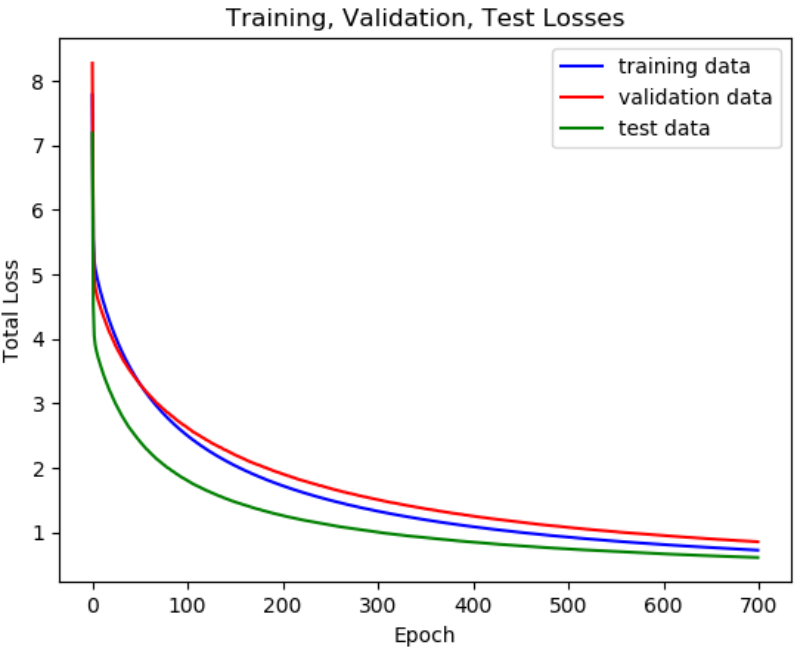
* Instead of creating a tf.truncated_normal tensor and initializing the weight variable with it, I have decided to use the tf.get_variable method which would allow me to both create a weight variable and also initialize it at the same time.
* The code for buildGraph can be found in the Appendix - **StochasticGradientDescent_Tensorflow.py**

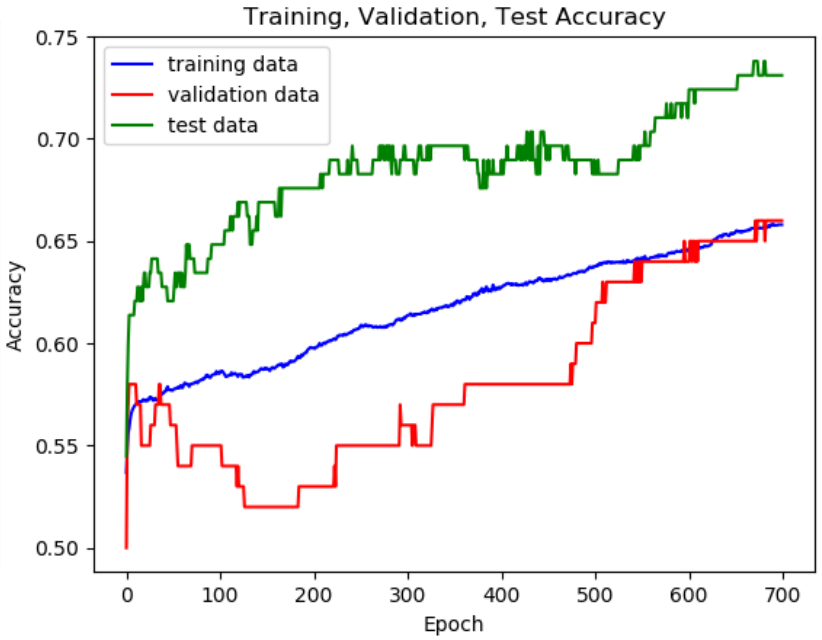
2. Implementing Stochastic Gradient Descent

The number of iterations of SGD is calculated by dividing the total number of data points by the batch size then multiplied by the number of epochs. This would produce the total number of iterations for a given number of epochs.

* The code for stochasticGradientDescent can be found in **StochasticGradientDescent_Tensorflow.py**

For Linear Regression (MSE Loss)

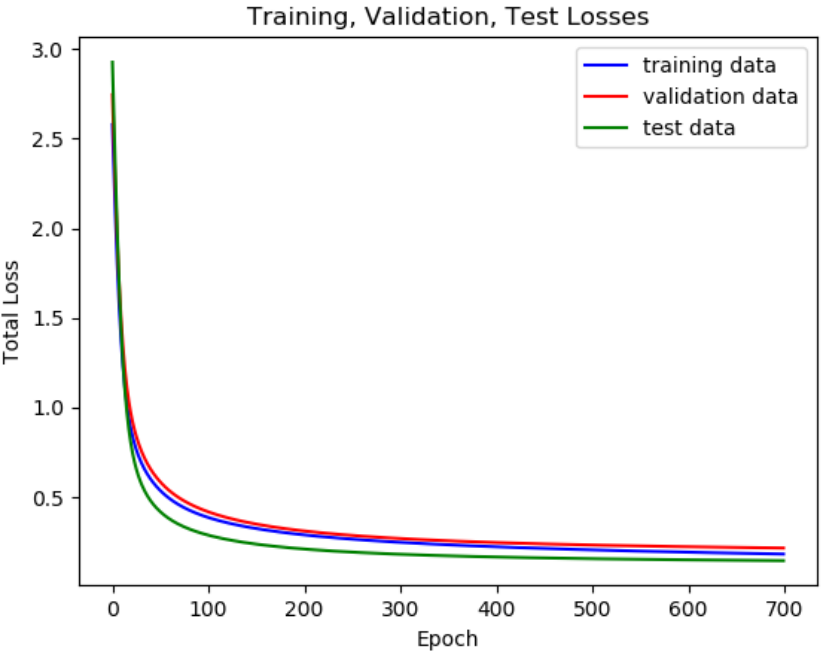


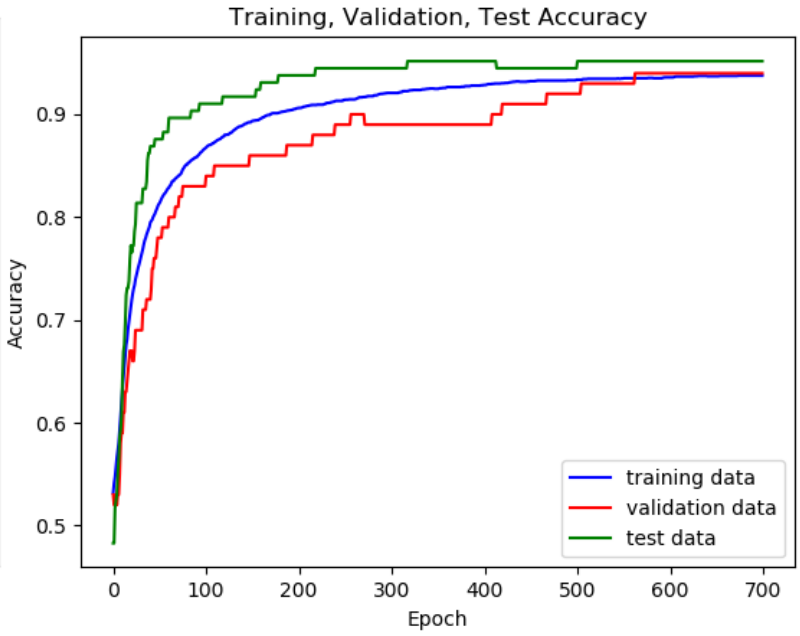


SGD Loss and Accuracy using a mini-batch size of 500 for 700 epochs. Learning Rate = 0.001 and Weight Decay = 0

	Training Data	Validation Data	Test Data
Final Loss	0.72422408	0.85495580	0.61018993
Final Accuracy	0.658	0.66	0.73103448

For Logistic Regression (CE Loss)



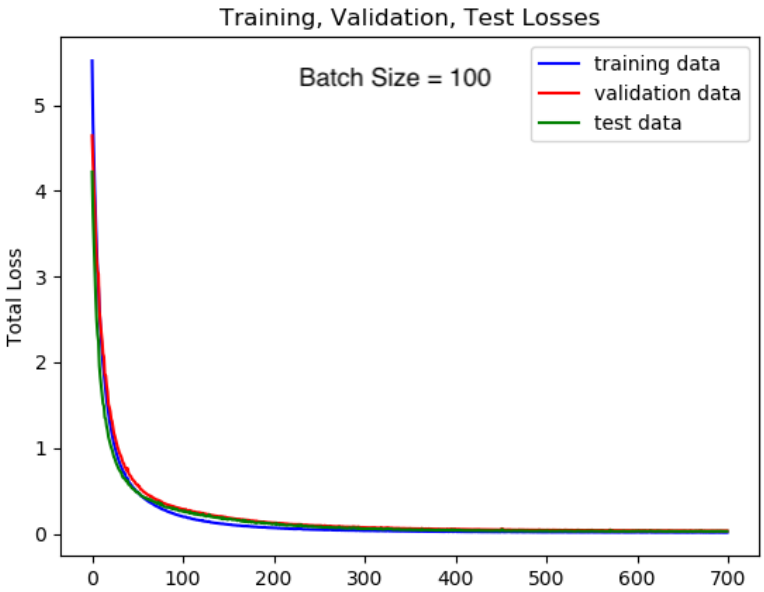


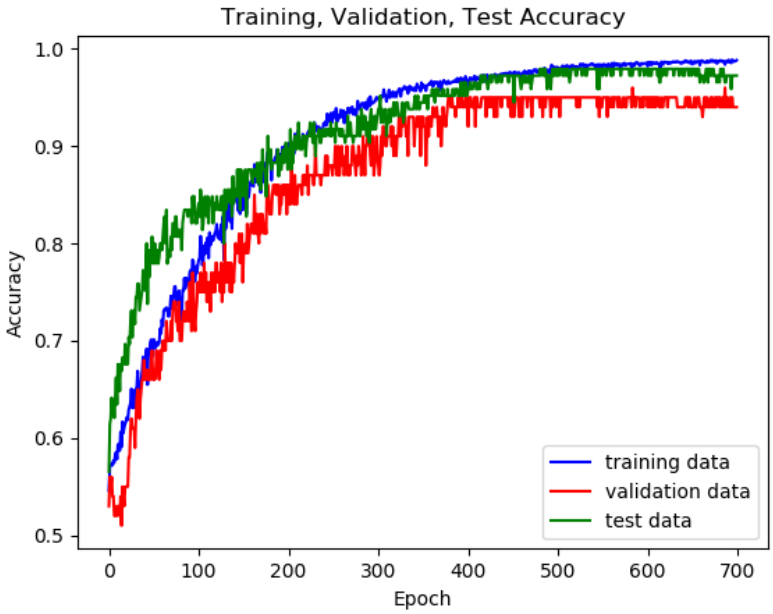
SGD Loss and Accuracy using a mini-batch size of 500 for 700 epochs. Learning Rate = 0.001 and Weight Decay = 0

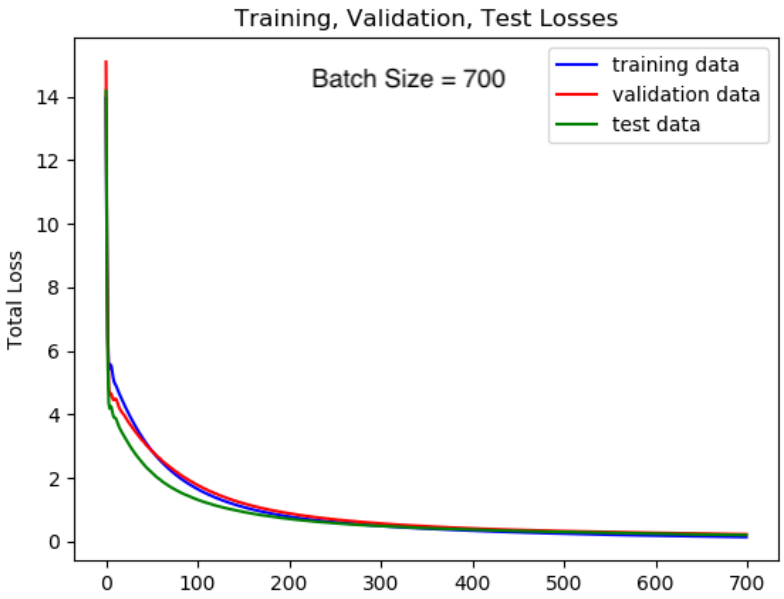
	Training Data	**Validation Data**	**Test Data**
Final Loss	0.18241754	0.21605364	0.14546842
Final Accuracy	0.93771428	0.94	0.95172413

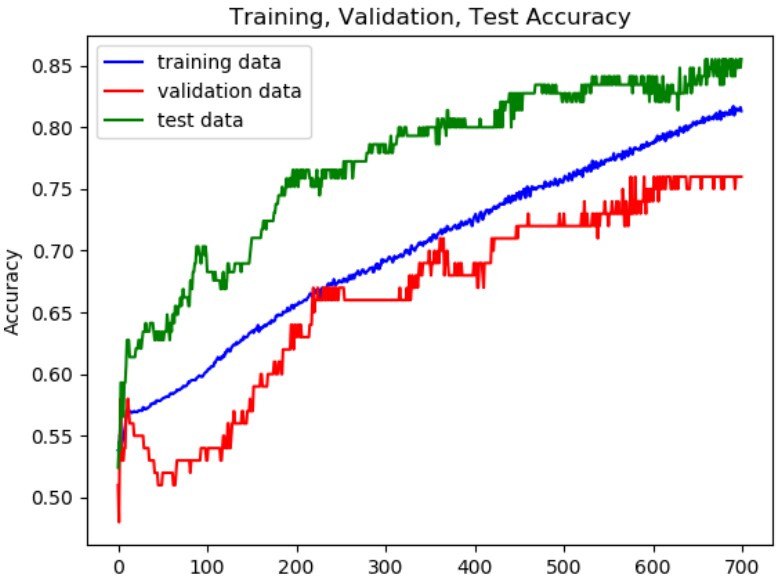
3. Batch Size Investigation with ADAM

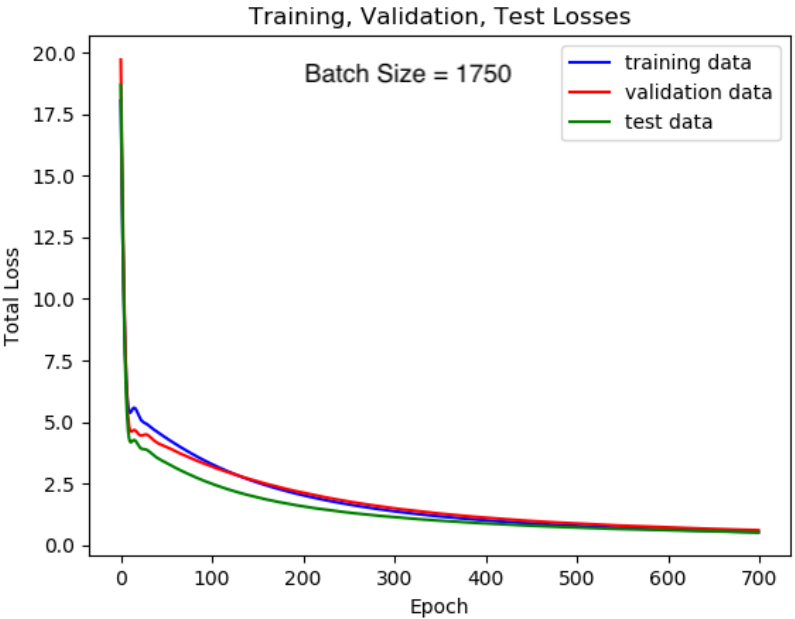
For Linear Regression (MSE Loss)

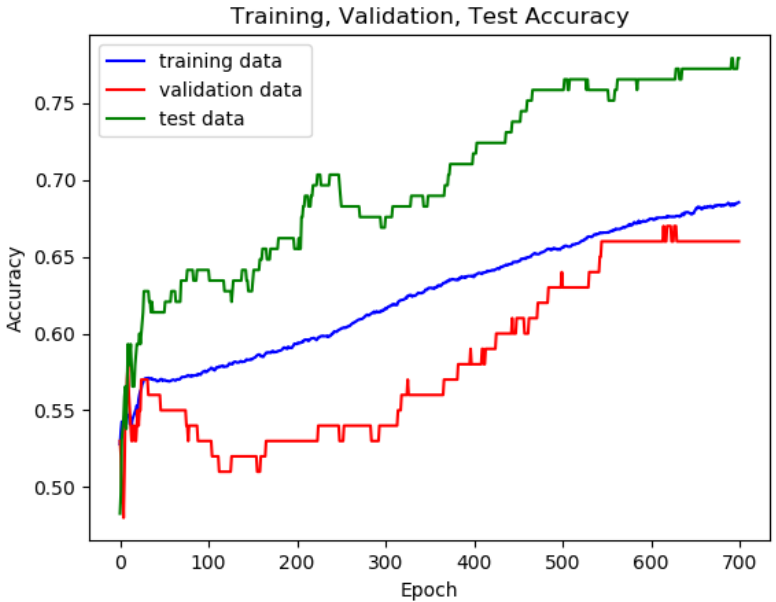








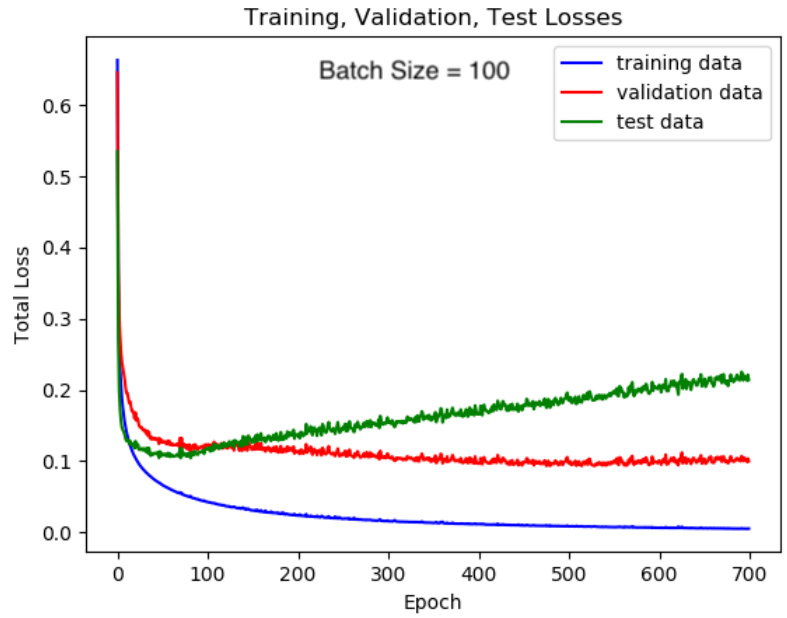


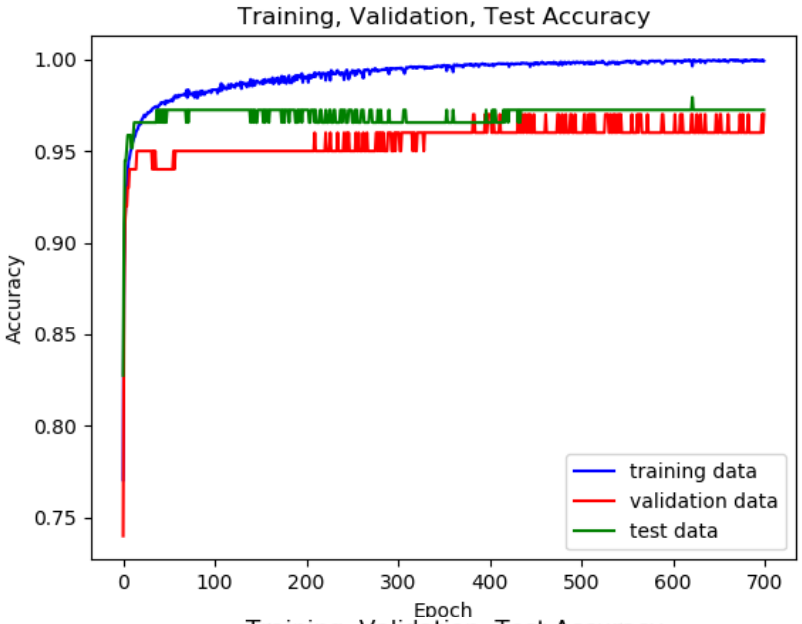


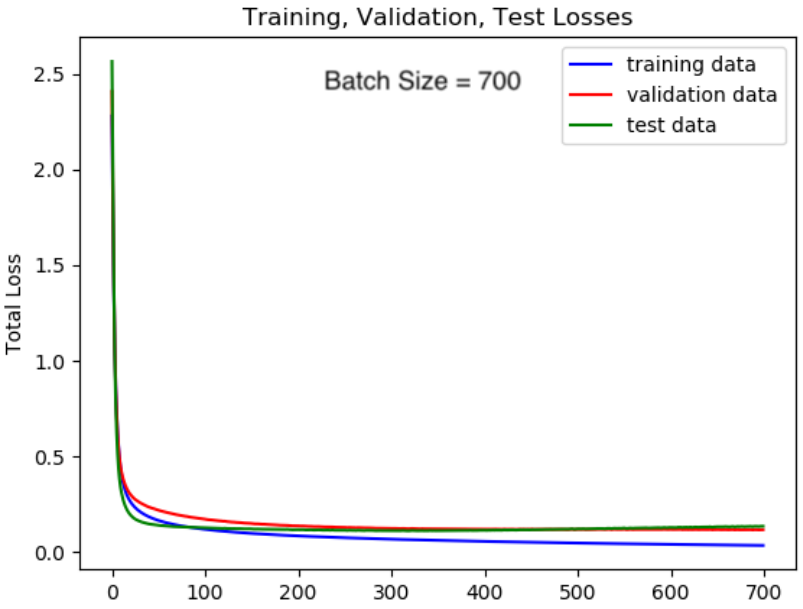
Loss and Accuracy curves using different batch sizes with Learning Rate = 0.001, Weight Decay = 0, 700 epochs

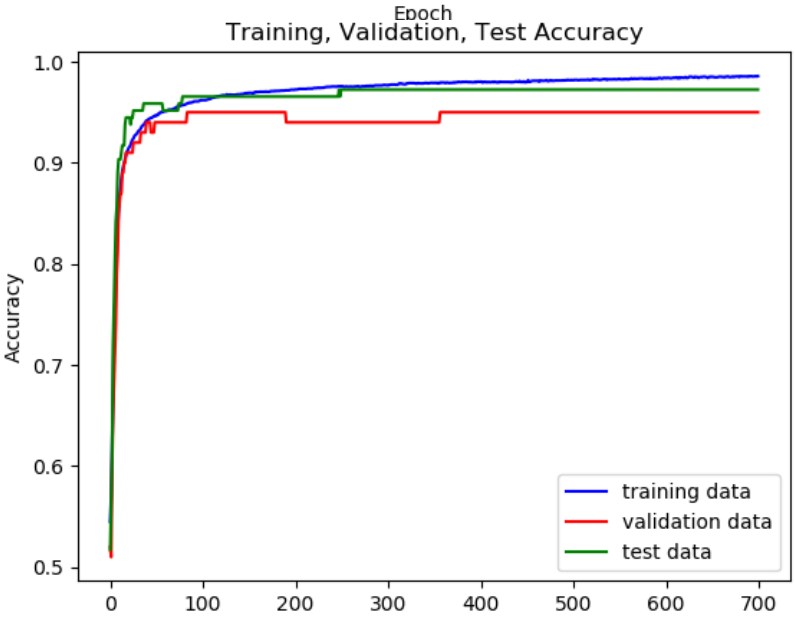
Batch Size		Training Data	Validation Data	Test Data
100	**Total Loss**	0.01439591	0.03407916	0.02853674
	Final Accuracy	0.98828571	0.94	0.97241379
700	**Total Loss**	0.14549444	0.23336813	0.20804074
	Final Accuracy	0.81314285	0.76	0.85517241
1750	**Total Loss**	0.53134798	0.61057714	0.52087586
	Final Accuracy	0.68542857	0.66	0.77931034

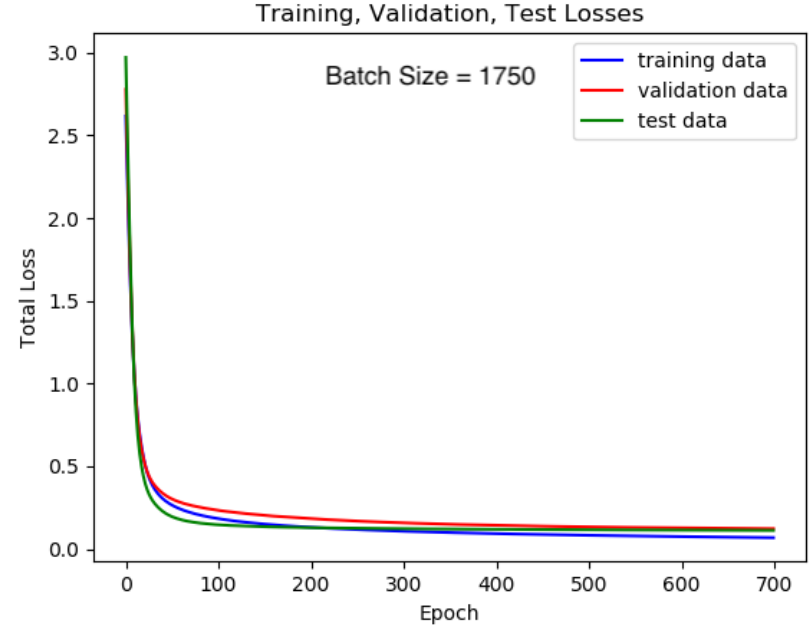
For Logistic Regression (CE Loss)

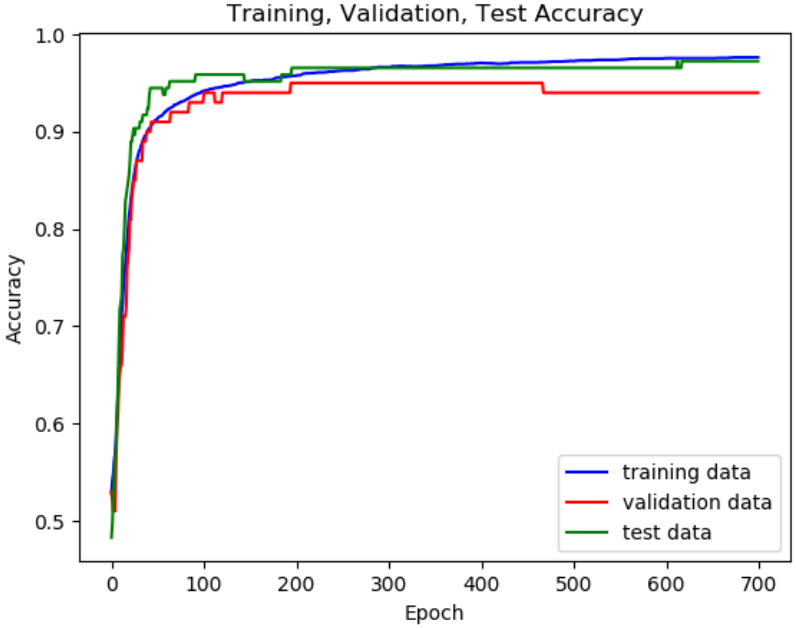












Loss and Accuracy curves using different batch sizes with Learning Rate = 0.001, Weight Decay = 0, 700 epochs

Batch Size		Training Data	Validation Data	Test Data
100	Total Loss	0.00522317	0.09998456	0.21406158
	Final Accuracy	0.99914285	0.97	0.97241379
700	Total Loss	0.03346024	0.11561940	0.13390165
	Final Accuracy	0.98571428	0.95	0.97241379
1750	Total Loss	0.06798275	0.12272204	0.11231069
	Final Accuracy	0.97628571	0.94	0.97241379

What is the impact of batch size on the final classification accuracy for each of the 3 cases?

In the cases tested for both MSE and CE loss, a higher batch size results in lower final accuracy.

What is the rationale for this?

This could be because of two reasons

1) Smaller batch sizes may have a 'regularizing' effect on the model. This is because when we use gradient descent on a sample of the data instead of the whole training set the calculated gradients are not exactly the same as the gradient for the entire data set. This results in a noisy estimate that may help the model avoid overfitting and improve generalization.
2) The same number of epochs for different batch sizes produce different number of total iterations. This means that for a batch size of 100 over 700 epochs, the number of iterations of stochastic gradient descent would be 24500 iterations. Whereas for a batch size of 700 over 700 epochs would require only 3500 iterations. This discrepancy in iterations causes a difference in the number of learning steps taken, which causes the smaller batch size to converge more than the larger batch size.

* Unrelated comment on the loss curves for cross-entropy with batch size 100. The validation and testing loss increases while the training loss decreases. The data is not incorrect, however this is an example of overfitting the training set.

4. ADAM Hyperparameter Investigation

For Linear Regression (MSE Loss)

Measurements using batch size = 500, epochs = 700, learning rate = 0.001, weight decay = 0

Parameter		Training Data	Validation Data	Test Data
B1 = 0.95	**Total Loss**	0.08343537	0.15551991	0.13583801
	Final Accuracy	0.87771428	0.85	0.87586206
B1 = 0.99	**Total Loss**	0.112864789	0.196572773	0.1749424580
	Final Accuracy	0.8474285	0.79	0.868965517
B2 = 0.99	**Total Loss**	0.0325281253	0.069940772	0.0545789062
	Final Accuracy	0.95285714	0.91	0.931034482
B2 = 0.9999	**Total Loss**	0.15318145	0.24136271	0.217540171
	Final Accuracy	0.811714285	0.76	0.8482758620
E = 1e-09	**Total Loss**	0.079987410	0.1505806646	0.129725471
	Final Accuracy	0.878	0.83	0.87586206
E = 1e-04	**Total Loss**	0.0801929517	0.150841063	0.130039070
	Final Accuracy	0.8765714	0.83	0.875862068

For Logistic Regression (CE Loss)

Measurements using batch size = 500, epochs = 700, learning rate = 0.001, weight decay = 0

Parameter		Training Data	Validation Data	Test Data
B1 = 0.95	**Total Loss**	0.0240967792	0.1107732538	0.1414138597
	Final Accuracy	0.99	0.95	0.9655172413
B1 = 0.99	**Total Loss**	0.0252731439	0.113124432	0.138829028
	Final Accuracy	0.98942857	0.95	0.9655172413
B2 = 0.99	**Total Loss**	0.0125086261	0.1011310510	0.162923830
	Final Accuracy	0.99685714	0.96	0.965517241

B2 = 0.9999	**Total Loss**	0.0337364233	0.1143535652	0.1338111551
	Final Accuracy	0.985142857	0.95	0.972413793
E = 1e-09	**Total Loss**	0.023889837	0.1102686326	0.1431339751
	Final Accuracy	0.99028571	0.95	0.965517241
E = 1e-04	**Total Loss**	0.0242886971	0.110876271	0.142228455
	Final Accuracy	0.98971428	0.95	0.965517241

For each, what is the hyperparameter impact on the final training, validation and test accuracy? Why is this happening?

The ADAM update (reference found on wiki)

$$m_w^{(t+1)} \leftarrow \beta_1 m_w^{(t)} + (1-\beta_1)\nabla_w L^{(t)}$$
$$v_w^{(t+1)} \leftarrow \beta_2 v_w^{(t)} + (1-\beta_2)(\nabla_w L^{(t)})^2$$
$$\hat{m}_w = \frac{m_w^{(t+1)}}{1-(\beta_1)^{t+1}}$$
$$\hat{v}_w = \frac{v_w^{(t+1)}}{1-(\beta_2)^{t+1}}$$
$$w^{(t+1)} \leftarrow w^{(t)} - \eta \frac{\hat{m}_w}{\sqrt{\hat{v}_w} + \epsilon}$$

beta1

The beta1 value adjusts the exponential decay rate for the first moment estimates. We see that as beta1 approaches 1, the m_hat term approaches infinity. Since m_hat is proportional to the gradient step update (with momentum) this means that as beta1 approaches 1 the weights will update at a faster rate, and thus converge quickly. However, our data seems to suggest that higher beta1 values result in a slightly higher loss and lower accuracy. This may be because the previous momentum term was very slow (small value) and a higher beta1 value would mean that the new update step is heavily influenced by the previous (slower) momentum term. This would ultimately make convergence slower than if a smaller beta1 value was used.

beta2

The beta2 value adjusts the exponential decay rate for the first moment estimates. We see that as beta2 approaches 1, the v_hat term approaches infinity. Since v_hat is inversely proportional to the gradient step update (with momentum) this means that as beta2 approaches 1 the weights will update at a slower rate, and thus the loss may be higher. Our data supports this idea as a slightly higher beta2 value generally lowered the accuracy and increased the loss.

epsilon

The epsilon value is simply a very small number that prevents division by zero in the ADAM step update. In the case of our data a higher epsilon value results in a slightly lower training accuracy. In general a higher epsilon will cause slower convergence. This is because epsilon inversely controls the step size, and by making e high we make the step size smaller and thus it takes longer to converge.

5. Mean Squared Error Loss vs Cross Entropy Loss

* Parts 3.1.2 to 3.1.4 have already been repeated for cross entropy loss and the results are shown their corresponding questions.

How do the two models compare against each other in terms model performance (i.e. final classification accuracy)?

Part 3.1.2 - SGD
- minibatch size = 500
- epochs = 700
- weight decay = 0
- learning rate = 0.001

Comparing the loss and accuracy curves for both linear regression using MSE and logistic regression using CE it is apparent that minimizing the binary Cross-Entropy loss yields far better accuracy as well as much lower loss for all training, validation, and test data. CE is about 25% more accurate than MSE for this classification dataset.

Part 3.1.3 - Batch Size Investigation
- minibatch size = {100, 700, 1750}
- epochs = 700
- weight decay = 0
- learning rate = 0.001

Comparing the loss and accuracy using MSE and CE at different batch sizes a similar trend appears. At small batch sizes CE is the slightly better model as it yields lower loss and higher accuracy. However, as the batch sizes increases to 1750 this performance difference grows more significant. For CE at batch size=1750 the accuracy still maintains mid-90%, however for MSE at batch size=1750 the accuracy plummets to ~65%.

Part 3.1.4 - ADAM Hyperparameter Investigation
- minibatch size = 500
- epochs = 700
- weight decay = 0
- learning rate = 0.001

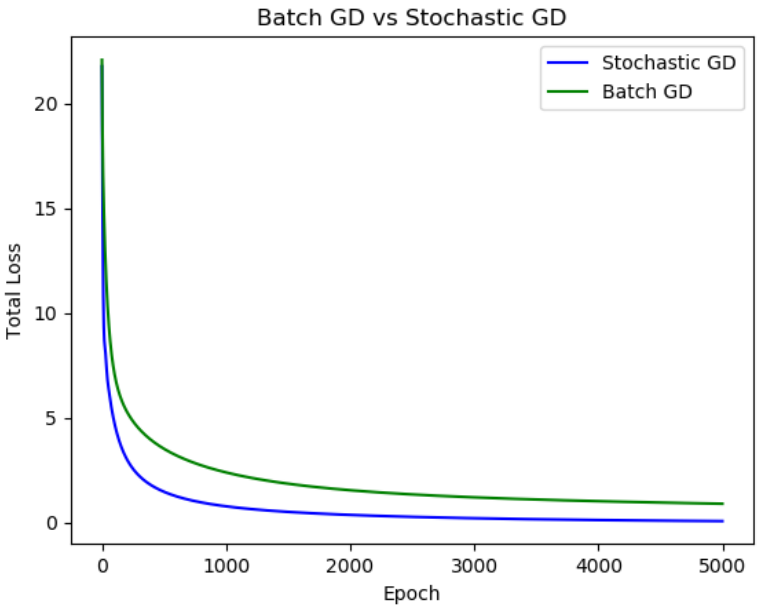
Comparing the loss and accuracy using MSE and CE for different ADAM hyperparameters suggests that that CE is the far more accurate loss model for this binary classification problem. The accuracies using cross-entropy are all ~95% while the accuracies using mean squared error all ~85%.

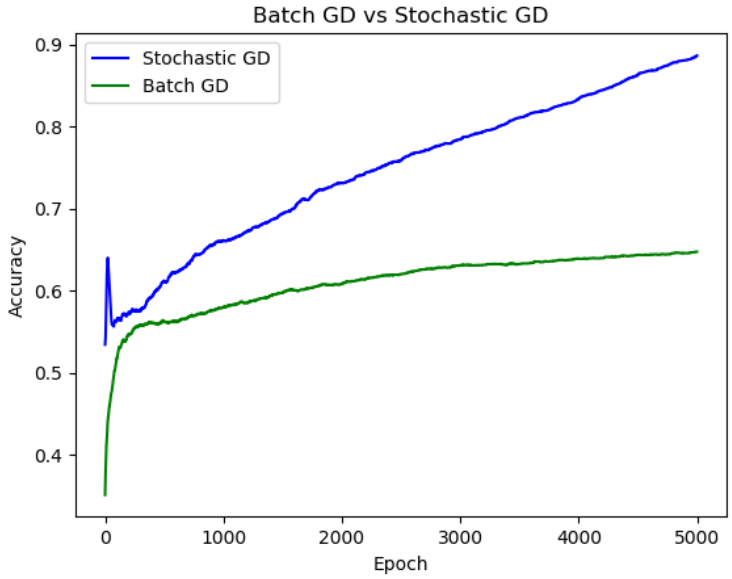
Conclusions
The accuracies of both mean squared error and cross-entropy models strongly suggest that the logistic regression cross-entropy loss is the better model to use for this binary classification problem.

6. Batch GD vs Stochastic GD with ADAM

In order to more accurately compare BGD vs SGD & ADAM we have tested both optimization methods using the following parameters

- Initialize weights using a random normal distribution with stddev=0.5
- Initialize bias using with constant 0
- Learning Rate = 0.001
- Weight Decay = 0
- Loss Type = MSE
- Batch size = Entire training data = 3500 (so that epochs == iterations)
- Number of epochs = 5000





Comment on the overall performance of the SGD algorithm with Adam vs. the batch gradient descent algorithm you implemented earlier.

The stochastic gradient descent algorithm using ADAM is significantly better in terms of loss, accuracy, and computation time (~120s vs ~40s).

Comment on the plots of the losses and accuracies of the SGD vs. batch gradient descent implementation. What do you notice about the curves? Why is this happening?

The loss curve for SGD with ADAM has a steeper descent than the loss curve for BGD. This is because the ADAM optimizer uses moments to provide an adaptive step size. When the loss is very large the step size is also large to allow faster convergence.

Similarly the loss curve for SGD with ADAM becomes flatter than the loss curve for BGD at low loss. This is also because of the ADAM optimizer's adaptive step size. At low loss the step size becomes very small to allow more precision in determining the minimum.

The accuracy curve for SGD with ADAM is much steeper than the accuracy curve for BGD. This further suggests that SGD with ADAM provides faster convergence than regular BGD.

Conclusion

Both the loss and accuracy curves strongly suggest that Stochastic Gradient Descent with ADAM is far superior to regular Batch Gradient Descent

Appendix - Code

---

starter.py

```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
from timeit import default_timer as timer

def loadData():
    with np.load('notMNIST.npz') as data :
        # Data (data['images']) is image intensity matrix 0-255
        # Target (data['labels']) is a number from 0 to 9
        Data, Target = data ['images'], data['labels']
        posClass = 2
        negClass = 9

        # get index array of all correctly labelled data
        dataIndx = (Target==posClass) + (Target==negClass)

        # normalize data intensity 0-1
        Data = Data[dataIndx]/255.

        # Target set to a column vector of only correctly labelled data
        Target = Target[dataIndx].reshape(-1, 1)
        # Classification of target data
        Target[Target==posClass] = 1
        Target[Target==negClass] = 0

        # seed the random number generator for repeatable results
        np.random.seed(421)

        # Shuffle the order of the data and target
        randIndx = np.arange(len(Data))
        np.random.shuffle(randIndx)
        Data, Target = Data[randIndx], Target[randIndx]

        # Set the training data, validation data, and test data
        trainData, trainTarget = Data[:3500], Target[:3500]
        validData, validTarget = Data[3500:3600], Target[3500:3600]
        testData, testTarget = Data[3600:], Target[3600:]
    return trainData, validData, testData, trainTarget, validTarget, testTarget

def parseData(trainData, trainTarget):
    num_data = trainData.shape[0]
    X = trainData.reshape(num_data, -1)
    y = trainTarget
    return X, y

# SET TRAINING, VALIDATION, TEST DATA TO GLOBAL VARS
trainingData, validData, testData, trainingLabels, validTarget, testTarget = loadData()
X_valid, y_valid = parseData(validData, validTarget)
X_test, y_test = parseData(testData, testTarget)
```

```
# ---------------------------------------------------------------------------
# LOSS AND ACCURACY FUNCTIONS

def MSE(w, b, X, y, reg):
    N = np.size(X, 0)   # number of rows in X
    L_D = (1/(2*N)) * (np.linalg.norm(X @ w + b - y)**2)
    L_W = (reg/2) * (np.linalg.norm(w)**2)
    return L_D + L_W

def crossEntropyLoss(w, b, X, y, reg):
    N = np.size(X, 0)   # number of datapoints
    z = X @ w + b
    ones = np.ones(N)
    CE = (1/N) * ((1-y).T @ z + ones.T @ np.log(1 + np.exp(-z)))[0,0]
    WD = (reg/2) * (np.linalg.norm(w)**2)
    return CE + WD

def classificationAccuracy(X, y, w, b):
    correct = 0.0
    for n in range(X.shape[0]):
        y_hat = X[n,:] @ w + b
        y_label = y[n,:]

        # Classify the linear regression output
        if y_hat >= 0.5:
            y_hat = 1
        else:
            y_hat = 0

        # Verify classification accuracy
        if y_hat == y_label:
            correct = correct + 1.0

    # Return classification accuracy
    return correct/X.shape[0]

def normalEquation(X, y, reg):
    # add [1 X] for bias term
    X = np.insert(X, 0, 1, axis=1)
    N = np.size(X, 0)   # number of rows in X

    pseudo_inv = np.linalg.inv(X.T @ X + N*reg*np.identity(X.shape[1])) @ X.T
    w_aug = pseudo_inv @ y
    b = w_aug[0,0]              # bias is constant for entire row 0
    w = w_aug[1:,:]
    return w, b

def measurePerformance(w, b, X, y, reg, lossType):
    # Measure Training, Validation, and Test Performance
    if lossType=="MSE":
        train_loss = MSE(w, b, X, y, reg)
        valid_loss = MSE(w, b, X_valid, y_valid, reg)
        test_loss = MSE(w, b, X_test, y_test, reg)
    elif lossType=="CE":
        train_loss = crossEntropyLoss(w, b, X, y, reg)
```

```
        valid_loss = crossEntropyLoss(w, b, X_valid, y_valid, reg)
        test_loss = crossEntropyLoss(w, b, X_test, y_test, reg)
    train_accuracy = classificationAccuracy(X, y, w, b)
    valid_accuracy = classificationAccuracy(X_valid, y_valid, w, b)
    test_accuracy = classificationAccuracy(X_test, y_test, w, b)

    return train_loss, valid_loss, test_loss, train_accuracy, valid_accuracy, test_accuracy
```

GradientDescent_Numpy.py

```
from starter import *

def gradMSE(w, b, X, y, reg):
    N = np.size(X, 0)   # number of rows in X
    ones = np.ones(N)
    gradMSE_W = (1/N) * X.T @ (X @ w + b - y) + reg*w
    gradMSE_B = (1/N) * ones.T @ (X @ w + b - y)
    return gradMSE_W, gradMSE_B


def gradCE(w, b, X, y, reg):
    N = np.size(X, 0)   # number of data points in X
    z = X @ w + b
    ones = np.ones(N)
    neg_sigmoid = 1./(1. + np.exp(z))

    dLoss_dz = (1/N) * ((1-y) - neg_sigmoid)
    dLoss_dw = X.T @ dLoss_dz + reg*w
    dLoss_db = ones.T @ dLoss_dz
    return dLoss_dw, dLoss_db

def grad_descent(X, y, alpha, iterations, reg, lossType, EPS=1e-7):
    # INITIALIZE WEIGHTS & BIAS
    w = np.random.normal(0, 0.1, (X.shape[1], y.shape[1]))
    b = np.random.uniform(-1, 1)
    train_loss, valid_loss, test_loss = [], [], []
    train_accuracy, valid_accuracy, test_accuracy = [], [], []

    for t in range(iterations):
        # CALCULATE LOSS
        if lossType=="MSE":
            E_in = MSE(w, b, X, y, reg)
        elif lossType=="CE":
            E_in = crossEntropyLoss(w, b, X, y, reg)

        # Measure Training, Validation, and Test Performance
        _train_loss, _valid_loss, _test_loss, \
        _train_accuracy, _valid_accuracy, _test_accuracy \
            = measurePerformance(w, b, X, y, reg, lossType)
        train_loss.append(_train_loss); valid_loss.append(_valid_loss); test_loss.append(_test_loss)
        train_accuracy.append(_train_accuracy); valid_accuracy.append(_valid_accuracy);
test_accuracy.append(_test_accuracy)
```

```
    # UPDATE STEP
    if lossType=="MSE":
        gradMSE_W, gradMSE_B = gradMSE(w, b, X, y, reg)
    elif lossType=="CE":
        gradMSE_W, gradMSE_B = gradCE(w, b, X, y, reg)
    w = w - alpha * gradMSE_W                    # Update Weights
    b = b - alpha * gradMSE_B                    # Update Biases

    # STOPPING CONDITION
    if lossType=="MSE":
        E_in_new = MSE(w, b, X, y, reg)
    elif lossType=="CE":
        E_in_new = crossEntropyLoss(w, b, X, y, reg)
    if np.abs(E_in_new - E_in) < EPS:
        break

return w, b, train_loss, valid_loss, test_loss, train_accuracy, valid_accuracy, test_accuracy
```

StochasticGradientDescent_Tensorflow.py

```
from starter import *

def buildGraph(X_data, y_data, learning_rate, lossType, ADAM, beta1=0.9, beta2=0.999, epsilon=1e-08):

    # INITIALIZE ---------------------------------------------------------------
    tf.set_random_seed(421)
    n_features = X_data.shape[1]

    # Define placeholders for input
    X = tf.placeholder(tf.float32, shape=(None, n_features), name="X")
    y = tf.placeholder(tf.float32, shape=(None, 1), name="y")
    reg = tf.placeholder(tf.float32, name="lambda")

    # Define weights and biases
    w = tf.get_variable("w", (n_features, 1),
                        initializer=tf.truncated_normal_initializer(mean=0, stddev=0.5))
    b = tf.get_variable("b", (1, ),
                        initializer=tf.constant_initializer(0.0))

    # DEFINE LOSS FUNCTIONS ----------------------------------------------------
    if lossType=="MSE":
        y_pred = tf.add(tf.matmul(X, w), b, name="y_pred")
        mse = 0.5 * tf.reduce_mean(tf.square(y_pred - y))   # Mean Squared Error
        wd = 0.5 * reg * tf.reduce_sum(tf.square(w))        # Weight Decay
        loss = tf.add(mse, wd, name="loss")                 # Total Loss

    elif lossType=="CE":
        y_pred_linear = tf.add(tf.matmul(X, w), b)
        y_pred = tf.nn.sigmoid(y_pred_linear, name="y_pred")
        ce = tf.reduce_mean(
            tf.nn.sigmoid_cross_entropy_with_logits(labels=y, logits=y_pred_linear))
        wd = 0.5 * reg * tf.reduce_sum(tf.square(w))
        loss = tf.add(ce, wd, name="loss")
```

```
    # INITIALIZE GRADIENT DESCENT OPTIMIZER ----------------------------------
    if ADAM==False:
        optimizer = tf.train.GradientDescentOptimizer(learning_rate).minimize(loss)
    else:
        optimizer = tf.train.AdamOptimizer(learning_rate, beta1, beta2, epsilon).minimize(loss)

    return X, w, b, y_pred, y, loss, optimizer, reg

def stochasticGradientDescent(batch_size, n_epochs, learning_rate, _lambda, lossType, ADAM, beta1=0.9,
beta2=0.999, epsilon=1e-08):
    X_data, y_data = parseData(trainingData, trainingLabels)
    n_datapoints = X_data.shape[0]
    iter_per_epoch = int(np.ceil(n_datapoints/batch_size))
    iterations = iter_per_epoch * n_epochs

    X, w, b, y_pred, y, loss, optimizer, reg = buildGraph(X_data, y_data, learning_rate, lossType, ADAM, beta1, beta2,
epsilon)
    global_init = tf.global_variables_initializer()
    train_loss, valid_loss, test_loss = [], [], []
    train_accuracy, valid_accuracy, test_accuracy = [], [], []

    with tf.Session() as sess:
        sess.run(global_init)
        for iter in range(iterations):
            # New Epoch Shuffle Data
            if (iter+1) % iter_per_epoch == 0:
                shuffle_index = np.random.choice(n_datapoints, n_datapoints, replace=False)
                X_data, y_data = X_data[shuffle_index], y_data[shuffle_index]

            # Select Mini-Batch
            X_batch, y_batch = X_data[:batch_size], y_data[:batch_size]

            # Gradient Descent Step on Mini-Batch
            _, _w, _b, _loss = sess.run([optimizer, w, b, loss],
                            feed_dict={X: X_batch, y: y_batch, reg:_lambda})

            # Shift data by batch size so next sample is new
            X_data = np.roll(X_data, batch_size, axis=0)
            y_data = np.roll(y_data, batch_size, axis=0)

            # After New Epoch Calculate the Loss and Accuracy
            if (iter+1) % iter_per_epoch == 0:
                print("EPOCH {}".format(int((iter+1)/iter_per_epoch)))

                # Measure Training, Validation, and Test Performance on ALL DATA (not just mini-batch)
                _train_loss, _valid_loss, _test_loss, \
                _train_accuracy, _valid_accuracy, _test_accuracy \
                    = measurePerformance(_w, _b, X_data, y_data, _lambda, lossType)
                train_loss.append(_train_loss); train_accuracy.append(_train_accuracy)
                valid_loss.append(_valid_loss); valid_accuracy.append(_valid_accuracy)
                test_loss.append(_test_loss); test_accuracy.append(_test_accuracy)

        w_optimal, b_optimal = sess.run([w, b])

    return w_optimal, b_optimal, train_loss, train_accuracy, valid_loss, valid_accuracy, test_loss, test_accuracy
```