ECE 421 Assignment 1 Report

Part I

1. Loss Function and Gradient

Here is our code:

```
def MSE(W, b, x, y, reg):
    N = x.shape[0]
    loss = np.linalg.norm(np.dot(x,W)+np.full((N,1),b)-y)**2/2/N+reg/2*np.linalg.norm(W)**2
    return loss

def gradMSE(W, b, x, y, reg):
    N = x.shape[0]
    wGrad = np.dot(np.transpose(x),np.dot(x,W)+b-y)/N+reg*W
    bGrad = np.sum(np.dot(x,W)+b-y)/N
    return wGrad, bGrad
```

The analytic expression we used is:
Loss Function (from the handout):

$$\mathcal{L} = \mathcal{L}_{\mathcal{D}} + \mathcal{L}_{W}$$

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

And the gradient formulas[1]:

$$\frac{\partial}{\partial \mathrm{m}} = \frac{2}{N} \sum_{i=1}^{N} - x_i (y_i - (m x_i + b))$$

$$\frac{\partial}{\partial \mathrm{b}} = \frac{2}{N} \sum_{i=1}^{N} - (y_i - (m x_i + b))$$

[1] Towards Data Science. (2019). *Linear Regression Using Gradient Descent in 10 Lines of Code*. [online] Available at: https://towardsdatascience.com/linear-regression-using-gradient-descent-in-10-lines-of-code-642f995339c0 [Accessed 5 Feb. 2019].

2. Gradient Descent Implementation

The code is very long, this is only the MSE component of the function:

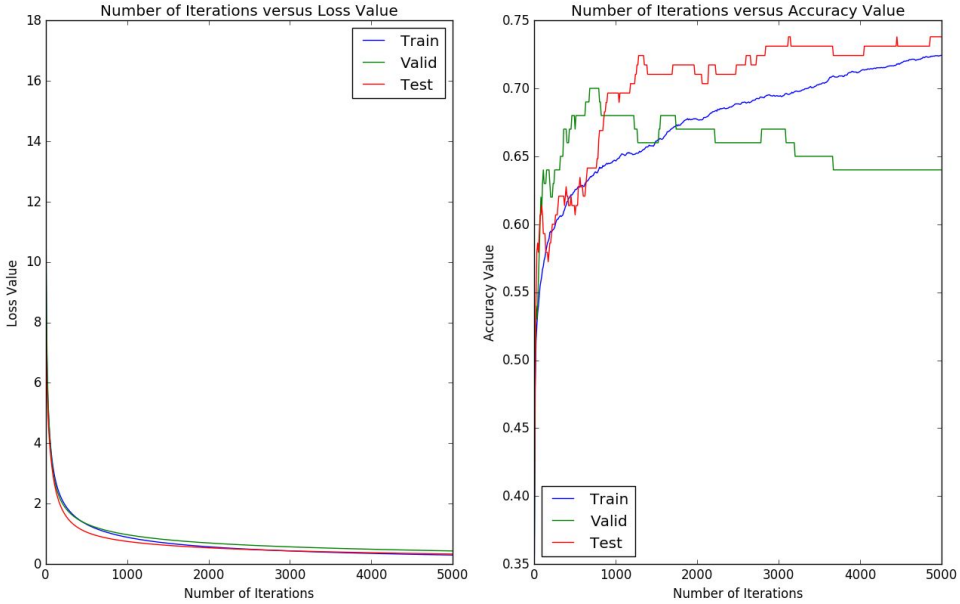
```python
def grad_descent(W, b, trainingData, trainingLabels, validData, validLabels, testData, testLabels, alpha, iterations, reg, EPS, lossType):
    count, count_, trainLoss, trainAccuracy, validLoss, validAccuracy, testLoss, testAccuracy = [], [], [], [], [], [], [], []
    if lossType=="MSE":
        for i in range (iterations):
            w_Grad, b_Grad = gradMSE(W, b, trainingData, trainingLabels, reg)
            if abs(np.linalg.norm(W)-np.linalg.norm(W-alpha*w_Grad)) < EPS:
                break
            W -= alpha*w_Grad
            b -= alpha*b_Grad
            count.append(i)
            trainLoss.append(MSE(W, b, trainingData, trainingLabels, reg))
            validLoss.append(MSE(W, b, validData, validLabels, reg))
            testLoss.append(MSE(W, b, testData, testLabels, reg))
            if i%10 == 0:
                trainAccuracy.append(evaluate_accuracy(W, b, trainingData, trainingLabels))
                validAccuracy.append(evaluate_accuracy(W, b, validData, validLabels))
                testAccuracy.append(evaluate_accuracy(W, b, testData, testLabels))
                count_.append(i)
        fig = plt.figure()
        ax1, ax2 = fig.add_subplot(1,2,1), fig.add_subplot(1,2,2)
        ax1.plot(count,trainLoss,label="Train")
        ax1.plot(count,validLoss,label="Valid")
        ax1.plot(count,testLoss,label="Test")
        ax1.set_xlabel("Number of Iterations")
        ax1.set_ylabel("Loss Value")
        ax1.set_title("Number of Iterations versus Loss Value")
        ax1.legend(loc="best")
        ax2.plot(count_,trainAccuracy,label="Train")
        ax2.plot(count_,validAccuracy,label="Valid")
        ax2.plot(count_,testAccuracy,label="Test")
        ax2.set_xlabel("Number of Iterations")
        ax2.set_ylabel("Accuracy Value")
        ax2.set_title("Number of Iterations versus Accuracy Value")
        ax2.legend(loc="best")
        fig.savefig("RESULTS",dpi=fig.dpi)
        finalTrainAccuracy, finalValidAccuracy, finalTestAccuracy = trainAccuracy[-1], validAccuracy[-1], testAccuracy[-1]
        finalTrainLoss, finalValidLoss, finalTestLoss = trainLoss[-1], validLoss[-1], testLoss[-1]
        return W, b, finalTrainAccuracy, finalValidAccuracy, finalTestAccuracy, finalTrainLoss, finalValidLoss, finalTestLoss
```

3. Tuning the Learning Rate

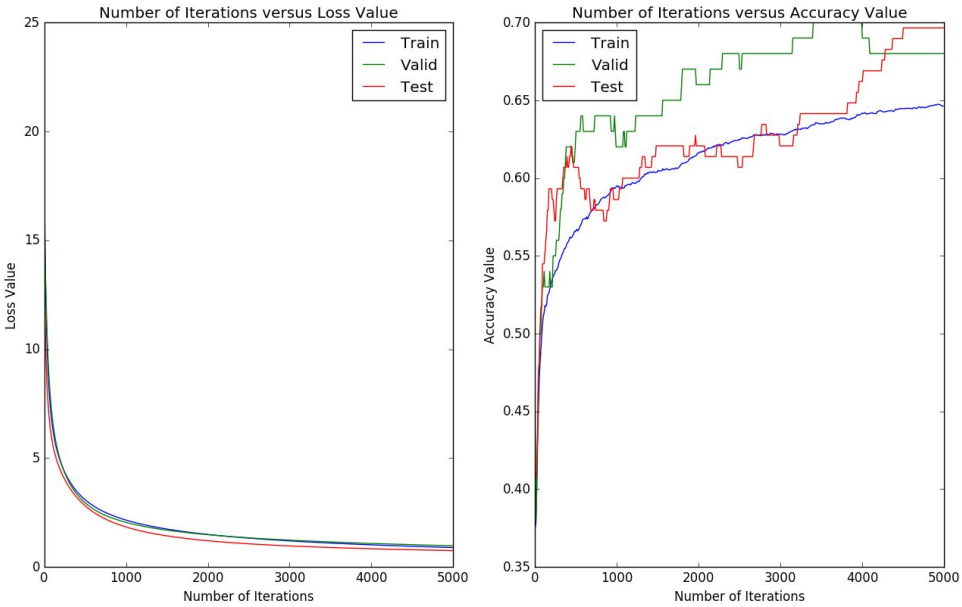
In this section, we initialize the weight values to a random Gaussian distribution with a mean of zero and a standard deviation of 0.5, this is done so that the difference between trials would be more apparent. The bias value is initialized to zero.

```
W, b = np.random.normal(0,0.5,(784,1)), 0
```

Case 1: α=0.005, epochs=5,000, λ=0:



Case 2: α=0.001, epochs=5,000, λ=0:



Case 3: α=0.0001, epochs=5,000, λ=0:

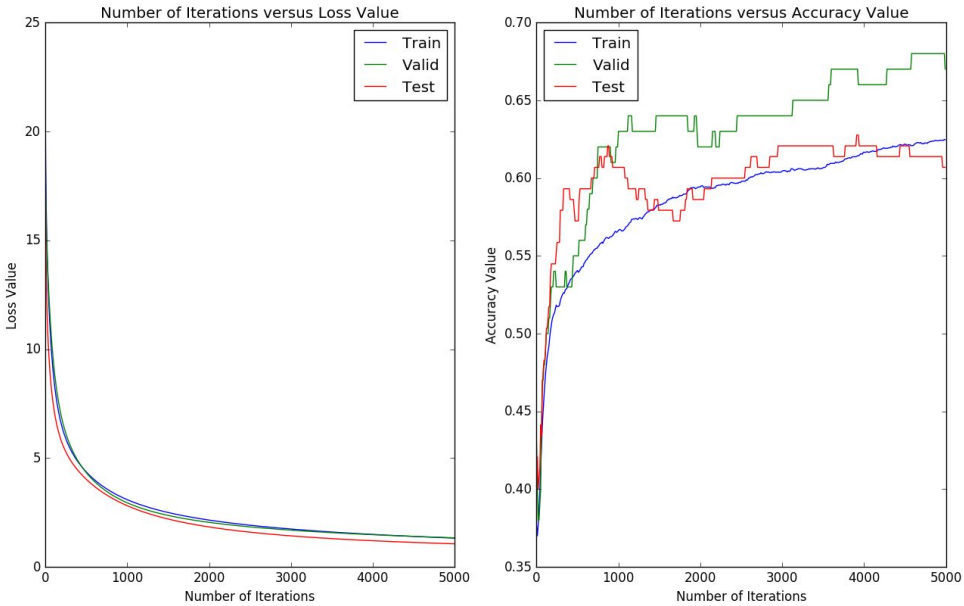


Table of final Loss Values:

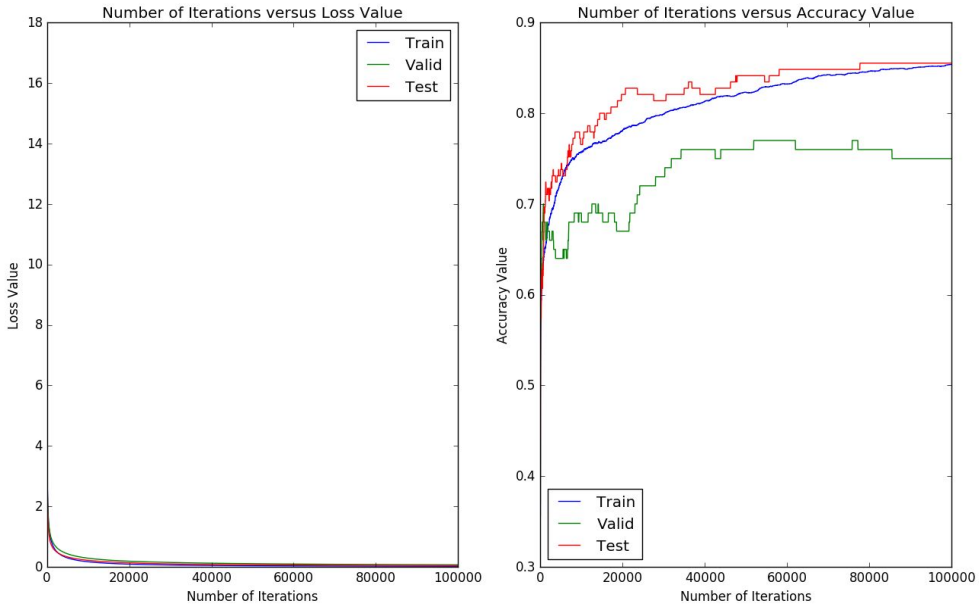
	Train Loss	Valid Loss	Test Loss
Case 1	**0.298**	**0.434**	**0.332**
Case 2	**0.889**	**0.973**	**0.750**
Case 3	**1.323**	**1.337**	**1.062**

Table of final Accuracy Values:

	Train Accuracy	Valid Accuracy	Test Accuracy
Case 1	**72.4%**	**64.0%**	**73.8%**
Case 2	**64.6%**	**68.0%**	**69.7%**
Case 3	**62.5%**	**67.0%**	**60.7%**

As we can see from the results, a higher learning rate will result in higher accuracies and lower losses for the same number of epochs, as the process of gradient descent is faster with a higher learning rate. However, higher learning rate can also lead to overfitting in some cases, which we will discuss later.

Since we have initialized a very flawed weight and bias, we actually need to have more than 5,000 epochs for it to have a decent accuracy. If we were to run case 1 with 100,000 epochs (using the same initialized values):

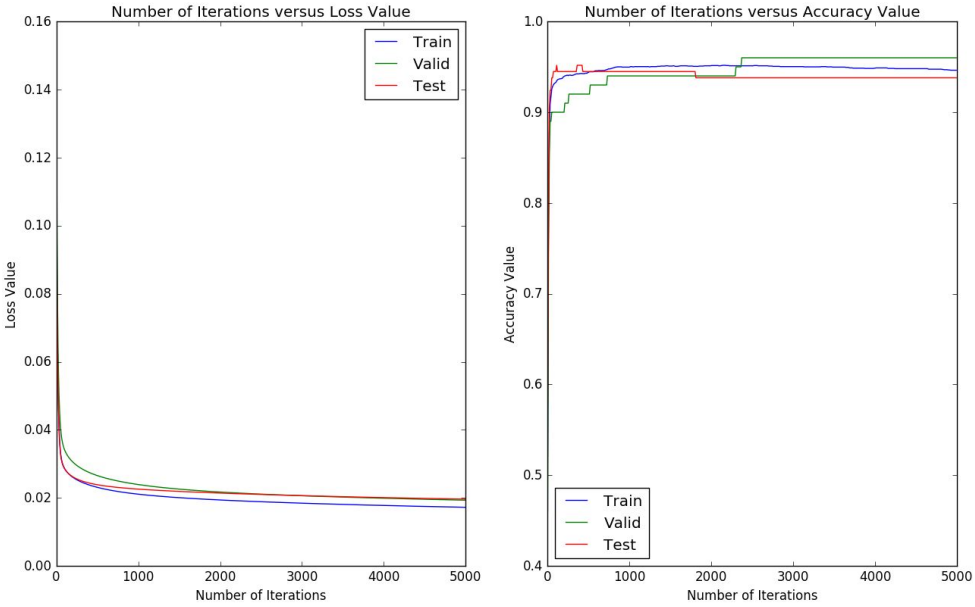


	Train Loss	Valid Loss	Test Loss
100,000 Epochs	**0.0226**	**0.0706**	**0.0470**

	Train Accuracy	Valid Accuracy	Test Accuracy
100,000 Epochs	**85.4%**	**75%**	**85.5%**

Despite having an improved accuracy, it is still fairly inaccurate, which means it requires further training, however if we initialize the all the weights to a value of 0.001, which is closer to the "ideal" weight, the accuracy becomes much higher (90%+). The parameters used are α=0.001, epochs=5,000, λ=0, with 5,000 Epochs:

```
W, b = np.full((784,1),0.001), 0
```

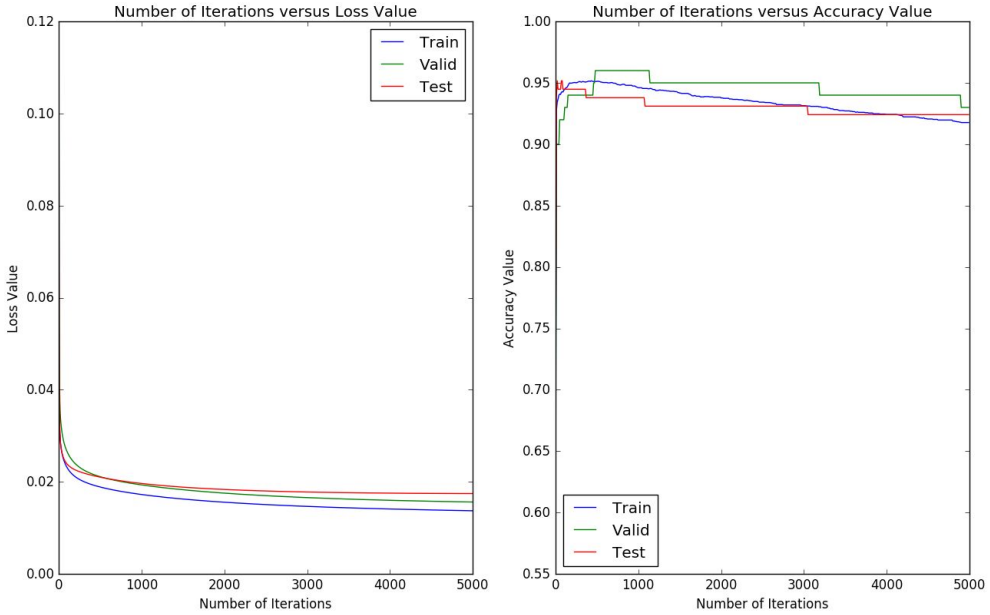


As we can see, the peak accuracy is quite high but the accuracy actually goes down slowly after a certain number of iterations, which is possibly due to overfitting, and we will discuss the effect of regularization parameter in the next section.

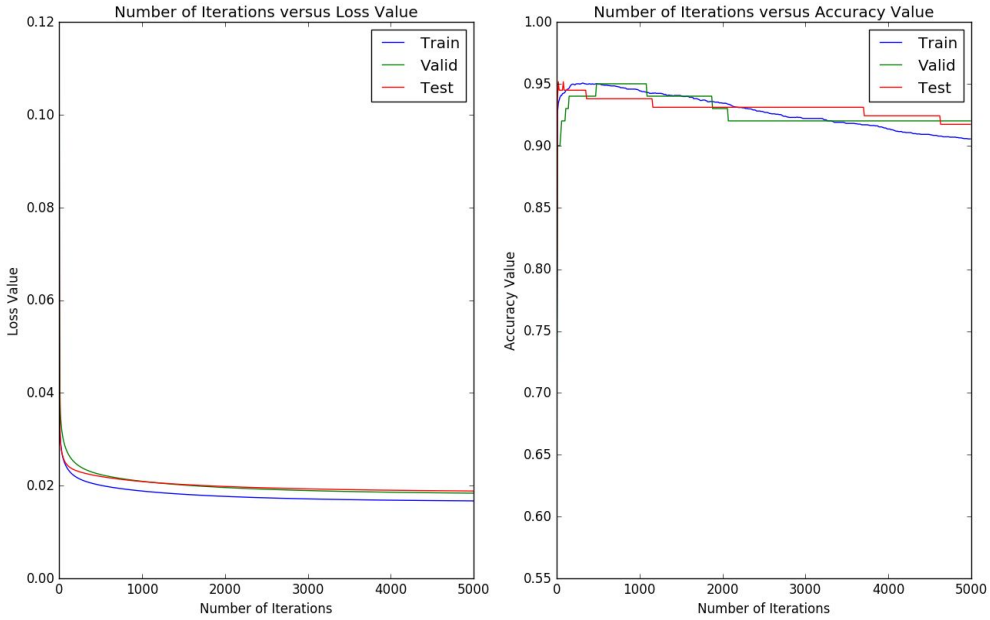
4. Generalization

In this section we will use initialized weights of 0.001 and a bias of zero.

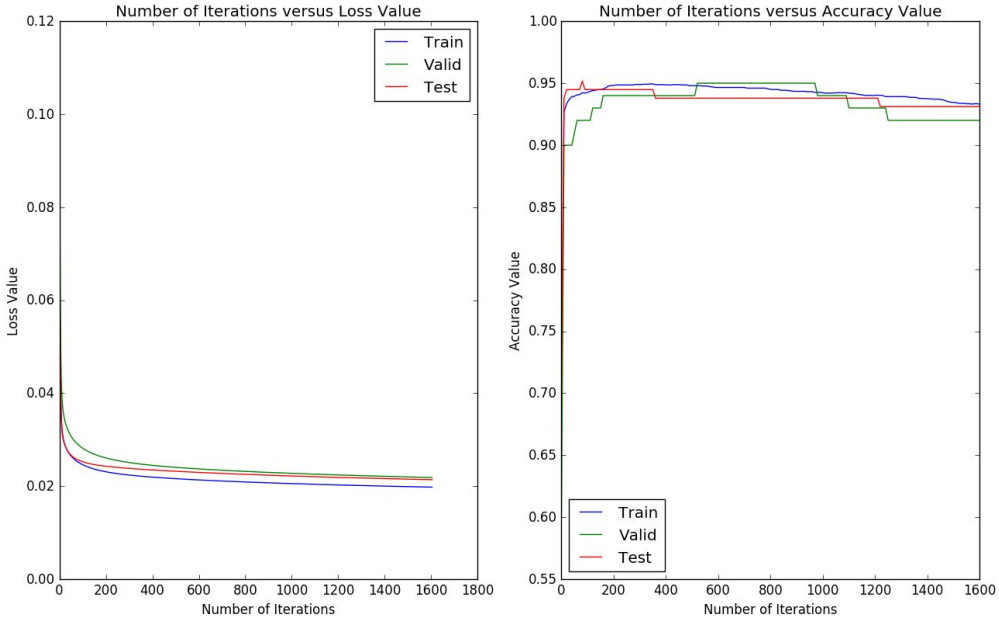
Case 1: α=0.005, epochs=5,000, λ=0.001:



Case 2: α=0.005, epochs=5,000, λ=0.1:



*****An additional graph with λ=0.25 is included for better visualization of changes:**



Case 3: α=0.005, epochs=5,000, λ=0.5:

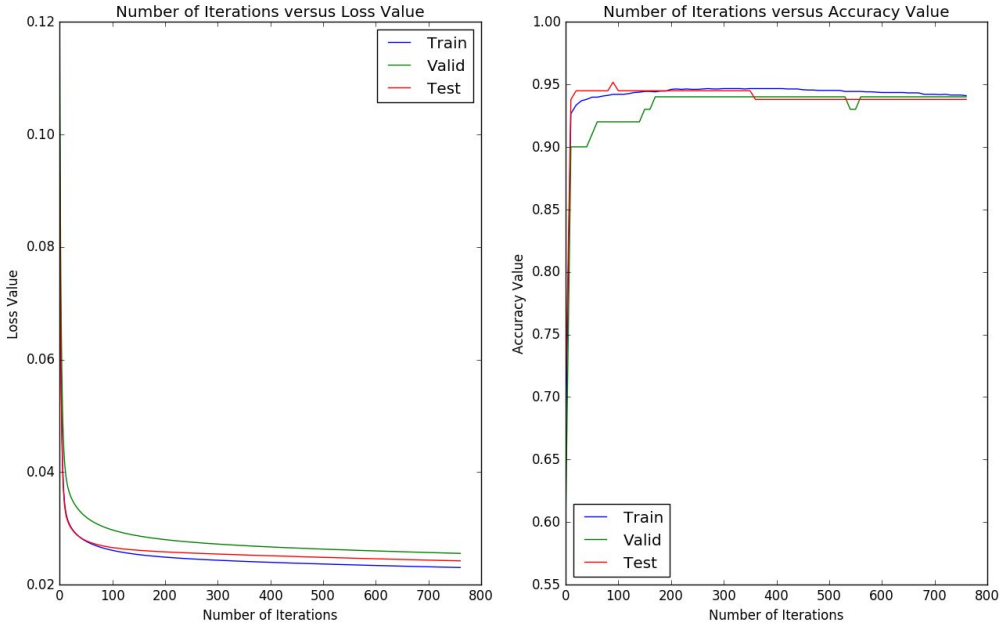


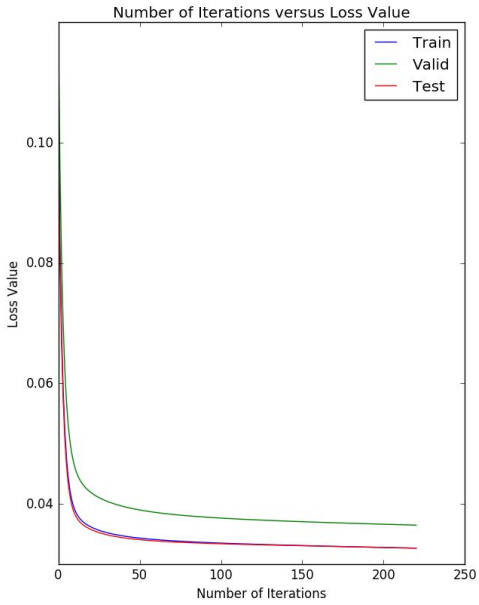
Table of final Loss Values:

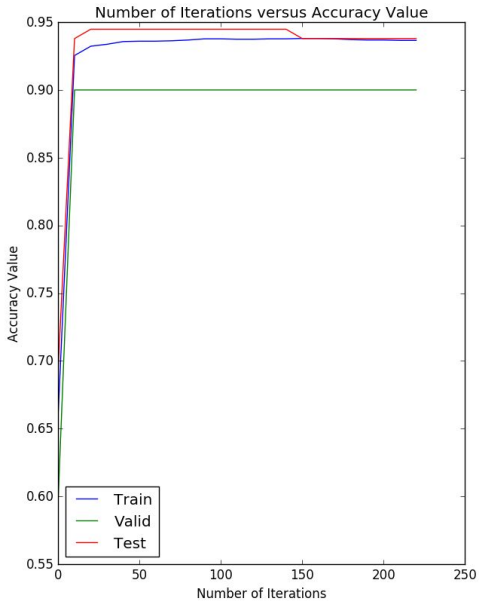
	Train Loss	Valid Loss	Test Loss
Case 1	**0.0137**	**0.0156**	**0.0174**
Case 2	**0.0167**	**0.0184**	**0.0189**
Case 3	**0.0230**	**0.0256**	**0.0242**

Table of final Accuracy Values:

	Train Accuracy	Valid Accuracy	Test Accuracy
Case 1	**91.8%**	**94.0%**	**92.4%**
Case 2	**90.5%**	**92.0%**	**91.7%**
Case 3	**94.1%**	**94.0%**	**93.8%**

Looking at the results, we can conclude that a suitable regularization parameter can effectively control overfitting, however the optimal regularization parameter has to be determined by analyzing and performing multiple trials with different values. When looking at the validation accuracy, if the decrease from peak accuracy to final accuracy is too large, that means we need to increase the reg value, since the effect of overfitting is becoming apparent, as we increase reg, the validation accuracy graph can become more flat, and the point at which peak accuracy occur will start shifting right as the reg increases. However, a larger reg does not necessarily lead to higher final accuracy, since it will also cause the algorithm to reach the error tolerance quicker and therefore terminate, so making sure it terminates when the peak accuracy occurs is the key to obtaining the highest final accuracy. The reg value also affects the computation time of the algorithm. The reg parameter needs to be fine tuned by performing multiple trials. Below is an additional example in which the reg value is too large. (reg value is 2.0)


Number of Iterations versus Loss Value
Train
Valid
Test
Loss Value
0.10
0.08
0.06
0.04
0
50
100
150
200
250
Number of Iterations


Number of Iterations versus Accuracy Value
Accuracy Value
0.95
0.90
0.85
0.80
0.75
0.70
0.65
0.60
0.55
Train
Valid
Test
0
50
100
150
200
250
Number of Iterations

5. Comparing Batch GD with normal equation

In this section we will use initialized weights of 0.001 and a bias of zero. The parameters are α=0.001, epoch=5,000, λ=0.

The normal equation is given by[2]:

$$\hat{\beta} = (X^T X)^{-1} X^T y$$

Table of final Loss Values:

	Train Loss	Valid Loss	Test Loss
Batch GD	0.0172	0.0193	0.0196
Normal Equation	0.0116	0.0301	0.0334

Table of final Accuracy Values:

	Train Accuracy	Valid Accuracy	Test Accuracy
Batch GD	94.6%	96.0%	93.8%
Normal Equation	98.4%	94.0%	92.4%

In terms of performance for the 5,000 epochs case (number of epochs only affects batch GD computation time), the normal equation takes around **4.5 seconds** to compute the results while batch GD takes slightly over 9 times the time at **41.5 seconds**. This is including the reshaping of raw data and any setups before the core computation component. The most computation heavy component for the normal equation is computing the inverse matrix, while in the batch GD case it is simply the high volume of relatively simple matrix multiplications that the computer has to perform. In conclusion, the results from the normal equation seem to have a higher accuracy and lower loss for a larger set of test data, whereas the batch GD approach would result in a more consistent accuracy and loss over datasets of different sizes, but the performance of the two approaches is quite similar for this specific case.

However, in the majority of the situations we have tested, the normal equation will always yield a very high accuracy (at around 95%) regardless of the values of the initialized weights and bias, which is not the case for batch GD since different initial values will require different amount of computation (epochs) to achieve the same accuracy, which is demonstrated in section 3.

[2] The Clever Machine. (2019). *Derivation: Ordinary Least Squares Solution and Normal Equations*. [online] Available at: https://theclevermachine.wordpress.com/2012/09/01/derivation-of-ols-normal-equations/ [Accessed 5 Feb. 2019].

Part II

1. Loss Function and Gradient

Here is our code:

```
def crossEntropyLoss(W, b, x, y, reg):
    N = x.shape[0]
    modelOutput = np.reciprocal(1+np.power(np.exp(1),(-1)*(np.dot(x,W)+b)))
    loss = (np.dot(np.transpose(y),np.log2(modelOutput))+np.dot(1-np.transpose(y),np.log2(1-modelOutput)))/(-N)+reg/2*np.linalg.norm(W)**2
    return loss.flatten()

def gradCE(W, b, x, y, reg):
    N = x.shape[0]
    modelOutput = np.reciprocal(1+np.power(np.exp(1),(-1)*(np.dot(x,W)+b)))
    wGrad = np.dot(np.transpose(x),modelOutput-y)/N+reg*W
    bGrad = np.sum(modelOutput-y)/N
    return wGrad, bGrad
```

The analytic expression we used is:
Loss Function (from the handout):

$$\mathcal{L} = \mathcal{L}_{\mathcal{D}} + \mathcal{L}_W$$

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

And the gradient formulas[3]:

$$\frac{\partial C}{\partial w_j} = \frac{1}{n} \sum_x x_j (\sigma(z) - y).$$

$$\frac{\partial C}{\partial b} = \frac{1}{n} \sum_x (\sigma(z) - y).$$

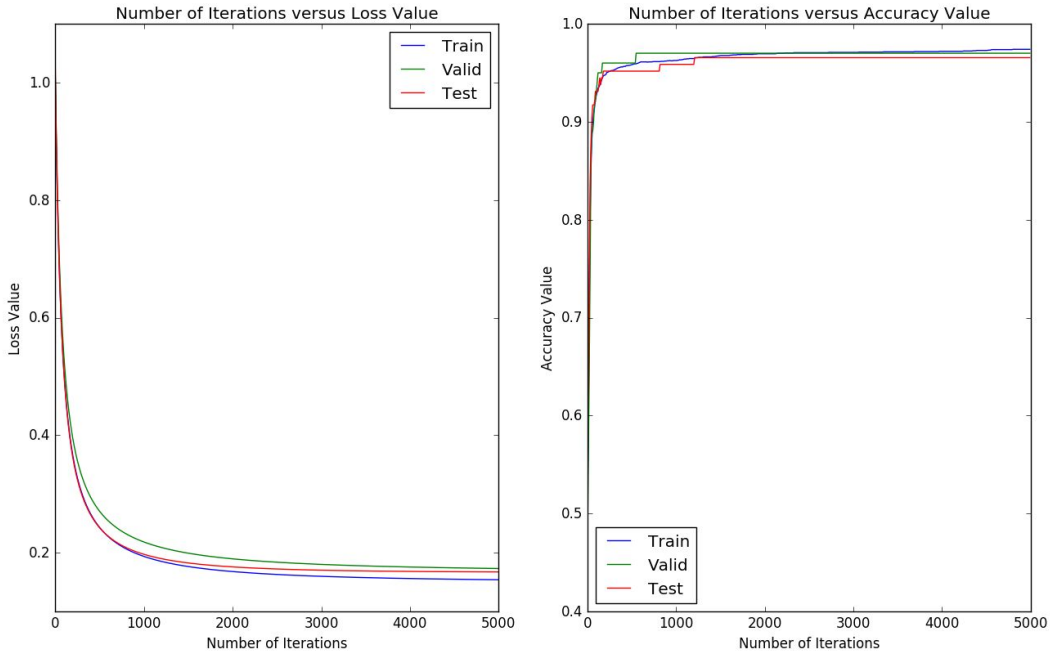
[3] Nielsen, M. (2019). *Neural Networks and Deep Learning*. [online] Neuralnetworksanddeeplearning.com. Available at: http://neuralnetworksanddeeplearning.com/chap3.html [Accessed 5 Feb. 2019].

2. Gradient Descent Implementation

In this section, we initialize the weight values to a random Gaussian distribution with a mean of 0.0 and a standard deviation of 0.5, the parameters are: α=0.005, epochs=5,000, λ=0.1.

The code is very long, this is only the CE component of the function, note that an assertion error will be displayed if the user inputs anything other than MSE or CE for the loss type:

```
elif lossType=="CE":
    for i in range (iterations):
        w_Grad, b_Grad = gradCE(W, b, trainingData, trainingLabels, reg)
        if abs(np.linalg.norm(W)-np.linalg.norm(W-alpha*w_Grad)) < EPS:
            break
        W -= alpha*w_Grad
        b -= alpha*b_Grad
        count.append(i)
        trainLoss.append(crossEntropyLoss(W, b, trainingData, trainingLabels, reg))
        validLoss.append(crossEntropyLoss(W, b, validData, validLabels, reg))
        testLoss.append(crossEntropyLoss(W, b, testData, testLabels, reg))
        if i%10 == 0:
            trainAccuracy.append(evaluate_accuracy(W, b, trainingData, trainingLabels))
            validAccuracy.append(evaluate_accuracy(W, b, validData, validLabels))
            testAccuracy.append(evaluate_accuracy(W, b, testData, testLabels))
            count_.append(i)
    fig = plt.figure()
    ax1, ax2 = fig.add_subplot(1,2,1), fig.add_subplot(1,2,2)
    ax1.plot(count,trainLoss,label="Train")
    ax1.plot(count,validLoss,label="Valid")
    ax1.plot(count,testLoss,label="Test")
    ax1.set_xlabel("Number of Iterations")
    ax1.set_ylabel("Loss Value")
    ax1.set_title("Number of Iterations versus Loss Value")
    ax1.legend(loc="best")
    ax2.plot(count_,trainAccuracy,label="Train")
    ax2.plot(count_,validAccuracy,label="Valid")
    ax2.plot(count_,testAccuracy,label="Test")
    ax2.set_xlabel("Number of Iterations")
    ax2.set_ylabel("Accuracy Value")
    ax2.set_title("Number of Iterations versus Accuracy Value")
    ax2.legend(loc="best")
    fig.savefig("RESULTS",dpi=fig.dpi)
    finalTrainAccuracy, finalValidAccuracy, finalTestAccuracy = trainAccuracy[-1], validAccuracy[-1], testAccuracy[-1]
    finalTrainLoss, finalValidLoss, finalTestLoss = trainLoss[-1], validLoss[-1], testLoss[-1]
    return W, b, finalTrainAccuracy, finalValidAccuracy, finalTestAccuracy, finalTrainLoss, finalValidLoss, finalTestLoss
else:
    assert False, "Error: Loss Type Mismatch"
```

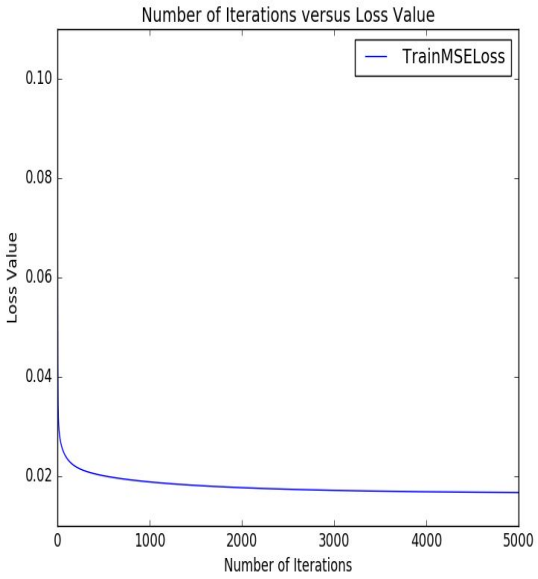


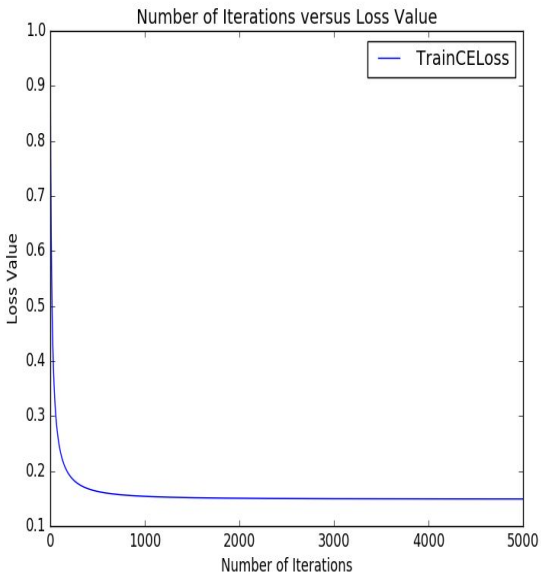
Logistic regression seems to have a better accuracy compared to linear regression for the same number of epochs, the final accuracy is very high at around 97% for all three sets of data.

3. Comparison to Linear Regression

Using the same set of initialization and parameter from the previous section.

A comparison of training loss for linear (left) and logistic (right) regression:





The CE loss of logistic regression model converges very quickly, after around 1,000 epochs, the change in CE loss becomes almost negligible as we can see from the graph.