Neural_Network

May 25, 2022

1 Yaseen Haffejee

2 1827555

3 Importing Relevant Libraries

[1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
sns.set()
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix,accuracy_score
from prettytable import PrettyTable
```

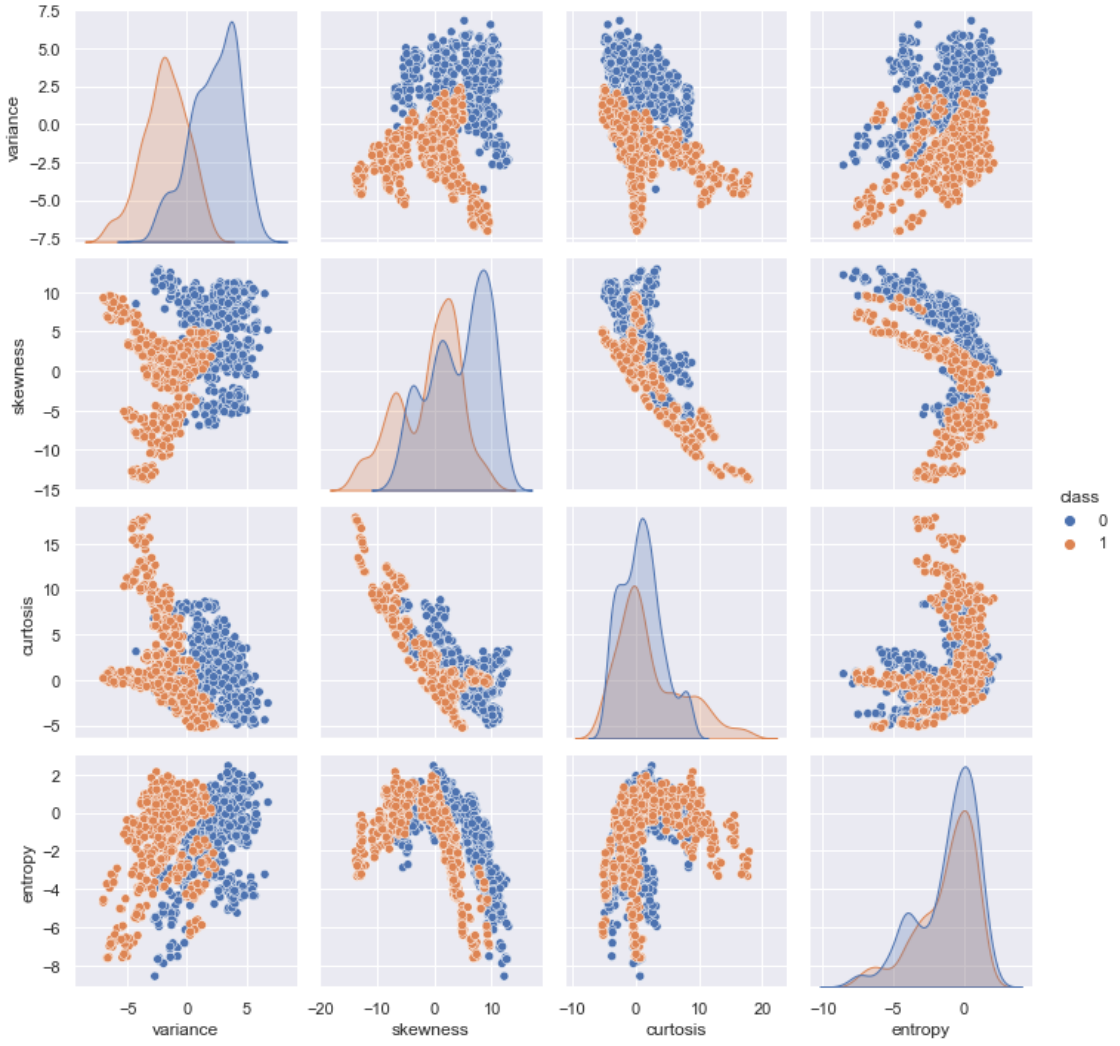
4 Data Preprocessing and Visualisation

4.0.1 Load the data

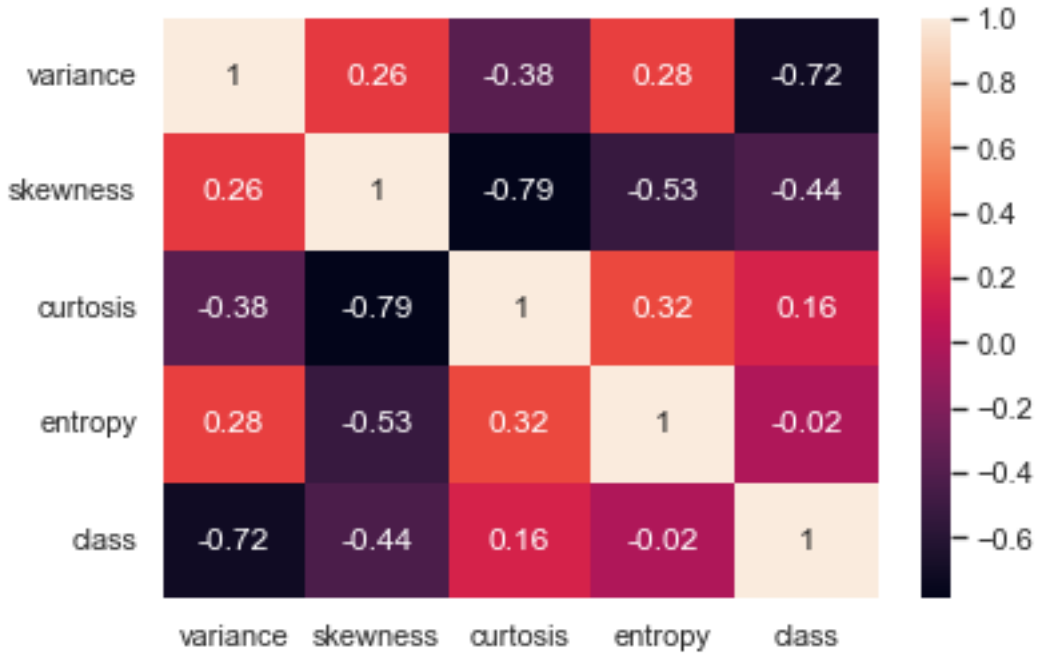
[2]:
```
data = pd.read_csv("BankNote_Authentication.csv")
```

Plot the variables against each other and colour code the points according to the class they belong to . - Class 0 represents normal - Class 1 represents fraudulent

[3]:
```
sns.pairplot(data,hue="class")
plt.show()
```



[114]:
```
sns.heatmap(np.round(data.corr(),2),annot=True,fmt="g")
plt.show()
```



- We can see that the data is clearly Non-Linear and will thus require non-linear activation functions.
- We also see there is no multicollinearity consequently we should not experience overfitting in general.

5 Class for dealing with the data

5.0.1 Understanding the dataset

- We are trying to predict whether or not a bank-note is fraudulent based on certain features.
- The features are extracted from images which are 400x400 pixels, and contain a resolution of 660 dots per inch(dpi).
- Wavelet Transform tool were used to extract features from images.
- The features we use to predict are :
  1. Variance: 1 represents a fraudulent note and 0 a legitimate note.
  2. Skewness.
  3. Curtosis.
  4. Entropy.
- We have a total of 1372 data points.

- In the class we simply call the Load_and_Split_Data method
- The method takes in parameters such as the actual dataset, the validation_and_test_size which is a float between 0 - 1 representing the percentage of the data that will be used for validation and testing, and the scale is a boolean representing whether or not to standardize the data.
- We return the Training, Validation and Testing data

[4]:
```
class Dataset:
    ## Method to split the data with test size using the built-in sklearn␣
↪method for splitting
    def __SplitData(self, x,y,testSize):
        trainx,testx, trainy,testy =␣
↪train_test_split(x,y,test_size=testSize,random_state=42)

        return trainx,trainy,testx,testy
    ## General method to standardize  features
    def __Standardize(self,x):
        rows,cols = np.shape(x)

        for i in range(cols):
            feature = x[:,i]
            mean = np.mean(feature)
            standard_deviation = np.std(feature)
            feature -= mean
            feature /= standard_deviation
            x[:,i] = feature

        return x
    ## We call this method which will call standardize on the features
    def __ScaleData(self,x):
        scaledx  = self.__Standardize(x)
        return scaledx
    ## This method is the interface for interacting with the class
    ## Data is the entire dataset
    ## validation_and_test_size: the size we want our validation and test data␣
↪to be. a decimal between 0 and 1
    ## We make validation set 70% of the stipulated size and test 30%
    ## scale : a boolean value telling us whether or not we want to scale the␣
↪features
    ## Returns: Training, Validation and Testing features and targets
    def Split_Data(self, data,validation_and_test_size,scale):
        ## Making sure we drop any missing values if they are present
        data = data.dropna()
        arrData = data.to_numpy()
        np.random.shuffle(arrData)

        rows,cols = np.shape(arrData)
        x = arrData[:,:cols-1]
        y = arrData[:, -1]
        y = np.reshape(y,(np.shape(y)[0],1))
        if(scale):
            ## We only scale the features and not the targets
            x = self.__ScaleData(x)
        test_size = np.round(validation_and_test_size*0.3,2)
```

```
        ## We first split the dataset into two sets, The training set and the␣
↪set of validation + testing
        trainx,trainy,tempx,tempy = self.
↪__SplitData(x,y,validation_and_test_size)
        ## We now split the validation and testing into separate datasets.
        validationx,validationy,testx,testy = self.
↪__SplitData(tempx,tempy,test_size)
        return trainx,validationx,testx, trainy,validationy,testy
```

6 1. Basic implementation of neural network trained using back-propagation

7 Neural Network Implementation Class

[5]:
```
class NeuralNetwork:
    ## This method is used to initalise the weight matrices for the architecture
    def␣
↪__initialiseWeights(self,NumberOfInputFeatures,NumberOfHiddenLayers,NumberOfNeuronsPerHidde
↪

        if(NumberOfHiddenLayers > len(NumberOfNeuronsPerHiddenLayer)):
            raise Exception("Please ensure that the number of neurons for every␣
↪hidden layer is specified!")
        elif(NumberOfHiddenLayers < len(NumberOfNeuronsPerHiddenLayer)):
            raise Exception("Please ensure that the number of hidden layers is␣
↪the same as the size of the number of neurons per hidden layer!")

        ## The +1 indicates the addition of the bias
        inputLayerWeights = np.random.
↪rand(NumberOfNeuronsPerHiddenLayer[0],NumberOfInputFeatures+1)
        weights  = [inputLayerWeights]

        for i in range(NumberOfHiddenLayers):
            w = None
            ## We have reached the layer which connects to the output layer
            if(NumberOfHiddenLayers -1 == i):
                w = np.random.
↪rand(NumberOfNeuronsInOutputLayer,NumberOfNeuronsPerHiddenLayer[i]+1)

            else:
                w = np.random.
↪rand(NumberOfNeuronsPerHiddenLayer[i+1],NumberOfNeuronsPerHiddenLayer[i]+1)

            weights.append(w)
```

```
        return weights
    ## Input:
    ## lastActivation: a matrix containing the values of the output of the␣
↪network. This is indicative of a probability that the point belongs to a␣
↪certain class.
    ## We convert these probabilities to a class value to determine how well␣
↪the network is learning
    ## returns: a list containing which class the network predicts
    def __ConvertProbabilityToClass(self,lastActivation):
        indices = np.where(lastActivation<0.5)
        lastActivation[indices] = 0
        indices2 = np.where(lastActivation>=0.5)
        lastActivation[indices2] = 1
        return lastActivation

    ## Input:
    ## Z which is the z value at a layer. Z1 = theta1 x X, Z2 = theta2*a1 etc
    ## function represents the activation function for the layer. We use this␣
↪to calculate the activation by applying the function to every entry in Z.
    ## We also calculate the derivative of z, which is used in the Error␣
↪calculation per layer in backprop.
    def __g(self,Z,function):
        a = None
        derivative_of_z = None
        if (function == "sigmoid"):
            a = (1/(1+np.exp(-Z)))
            derivative_of_z = a * (1-a)

        elif (function == "tanh"):
            a =  np.tanh(Z)
            derivative_of_z = 1 - (a**2)

        elif (function == "leaky relu"):
            a = np.copy(Z)
            a = np.where(a<0,0.001*a,a)
            derivative_of_z = np.copy(a)
            derivative_of_z[derivative_of_z<=0] = 0.001
            derivative_of_z[derivative_of_z>0] = 1

        elif (function == "relu"):
            a =  np.maximum(0,Z)
            derivative_of_z = np.copy(a)
            derivative_of_z[derivative_of_z<=0] = 0
            derivative_of_z[derivative_of_z>0] = 1

        elif (function == "softmax"):
            expo = np.exp(Z)
```

```
            expo_sum = np.sum(np.exp(Z))
            a = expo/expo_sum
            s = Z.reshape(-1,1)
            derivative_of_z = np.diagflat(s) - np.dot(s, s.T)

        return a,derivative_of_z

    ## Inputs:
    ## x: The training x values which are the input into the network
    ## weights: a list containing the weights that are used to propagate through the network
    ## activations: a list of activation functions that are used at each layer
    ## returns: avals which is a list containing the activations of each layer.
    ## derivatives, which is a list containing the derivatives of each z value which we use in our error calculation
    def __ForwardPropagation(self,x,weights,activations):
        ## The first a value is always the input features themselves
        avals = [x]
        derivatives = [0]
        n = len(weights)
        for i in range(n):
            z  = None
             ## Adding the bias node to the data
            aprev = np.insert(avals[i],0,1,axis=1)
            z = np.dot(weights[i],aprev.T)
            avals[i] = aprev

            a,derivative_of_z = self.__g(z,activations[i])
            avals.append(a.T)
            derivatives.append(derivative_of_z)

        return avals,derivatives

    ## Inputs:
    ## a: the activations of each layer. a is a list and each index represents the activations at that layer starting at a1
    ## y: The original set of training y values which we use to calculate the error in the final layer
    ## Weights: a list containing the weight matrices for the layers
    ## derivatives: the set of derivatives calculated at the layer based on the activation function. We calculate the derivatives finished when we are calculating the activations in order to save computation cycles
    ## returns: a list containing the errors in reversed order. Eg, [a4,a3,a2]
    def __Errors(self,a,y,weights,derivatives):
        ## n will tell us how many layers we have and thus we compute n-1 errors since there's no error term in the input layer
```

```
        n = len(a)
        ## first element is simply the last a value of forward propagation␣
↪minus the actual y values
        errors = [a[n-1]-y]
        loopLength = n -2
        for j in range(loopLength):
            ## Calculating the erros per layer excluding the input layer
            e = (((weights[loopLength - j].T)[1:,:])@(errors[j].T))*␣
↪derivatives[loopLength - j]
            errors.append(e.T)

        return errors

    ## Input:
    ## errors: a list containing the errors of each layer
    ## activations: a list containing the activations of each layer
    ## returns: a list containing the gradients of each layer
    def __Gradients(self,errors,activations):
        gradients = []
        reversed_errors = list(reversed(errors))
        n  = len(activations)
        for i in range(n-1):
            g = (reversed_errors[i].T)@(activations[i])
            gradients.append(g)
        return gradients

    ## Input:
    ## x,y: rpresent the training data
    ## weights: a list containing the weight matrices for each layer
    ## activations: a list containing the activations of each layer
    ## number_of_datapoints: an integer which we use to normalize the gradients
    ## regularization: a value which dictates the amount of regularization. If␣
↪it is 0, no regularization is applied
    ## gradients: a list containing the gradients of each layer from the␣
↪previous iteration , which get updated.
    ## returns: gradients which is the list containing the updated gradients
    def␣
↪__BackPropagation(self,x,y,weights,activations,number_of_datapoints,regularization,gradient
↪
        avalues,derivatives = self.__ForwardPropagation(x,weights,activations)

        errors = self.__Errors(avalues,y,weights,derivatives)

        Updatedgradients = self.__Gradients(errors,avalues)
        for k in range(len(gradients)):
                gradients[k]+=Updatedgradients[k]
```

```
        if(regularization != 0):
            n = len(gradients)
            for i in range(n):
                gradients[i] = (1/number_of_datapoints)*gradients[i] +␣
↪regularization*weights[i]
        else:
            n = len(gradients)
            for i in range(n):
                gradients[i] = (1/number_of_datapoints)*gradients[i]

        return gradients
    ## Input
    ## weightsNew: the updated weights after an iteration
    ## WeightsOld : the weights before they were updated
    ## returns True: If the norm of the difference between the weights is < 0.
↪00005 for at least l-2 weight matrices where l is the number of layers
    ## returns False: If the norm of the difference between the weights is NOT␣
↪< 0.00005 for more than 2 weight matrices.
    def __CheckWeightConvergence(self,weightsNew,WeightsOld):
        n = len(weightsNew)
        controls = []
        for k in range(n):
            if(np.linalg.norm(weightsNew[k]-WeightsOld[k])<0.00005):
                controls.append(True)
            else:
                controls.append(False)

        numFalse = np.where(np.asarray(controls)==False)
        length = len(numFalse[0])
        if(length>1):
            return False
        else:
            return True

    ## Function that is used to ensure we do not encounter division by 0 errors
    def __eta(self,x):
        ETA = 0.0000000001
        return np.maximum(x, ETA)

    ## The function which calculates the loss /error given a set of weights
    def __Cost(self,y,predicted,regularization,weights):
        n = len(y)
        yhat_inv = np.subtract(1.0,predicted)
        y_inv = np.subtract(1.0,y)
        yhat = self.__eta(predicted) ## clips value to avoid NaNs in log
        yhat_inv = self.__eta(yhat_inv)
        weightsSummation =0;
```

```
        for w in weights:
            weightsSummation+= np.sum(w**2)
        loss = -1/n * (np.sum(np.multiply(np.log(yhat), y) + np.
↪multiply((y_inv), np.log(yhat_inv))))+ (regularization/
↪(2*n))*weightsSummation
        return loss

    ## Input:
    ## x,y: rpresent the training data
    ## weights: a list containing the weight matrices for each layer
    ## number_of_datapoints: an integer which we use to normalize the gradients
    ## regularization: a value which dictates the amount of regularization. If␣
↪it is 1, no regularization is applied
    ## learningRate: the learning rate to be used
    ## Epochs:an integer denoting the maximum number of iterations for training
    ## ActivationFunctions: a list containing the activation functions of each␣
↪layer
    ## returns: the updated weights after completion of training and the number␣
↪of iterations taken to train/ converge
    def␣
↪__GradientDescent(self,x,y,weights,number_of_datapoints,learningRate,regularization,Epochs,
↪
        gradients = [None]*len(weights)
        loss = []
        for i in range(len(gradients)):
            gradients[i] = np.zeros(np.shape(weights[i]))

        iterations=0
        control = True
        while iterations<Epochs and control:
            Old_Weights = weights.copy()
            D = self.
↪__BackPropagation(x,y,weights,ActivationFunctions,number_of_datapoints,regularization,gradi


            for j in range(len(D)):
                weights[j] = weights[j]- learningRate*D[j]

            if( self.__CheckWeightConvergence(weights,Old_Weights)):
                control = False
            iterations+=1
            a,v = self.__ForwardPropagation(x,weights,ActivationFunctions)
            predictedValues = a[len(a)-1]
            lossValue = self.__Cost(y,predictedValues,regularization,weights)
            loss.append(lossValue)
        return weights,iterations,loss
```

```
    ## This method is used to essentially propagate validation or testing data␣
↪through the network and look at the results
    ## Input:
    ## x,y: rpresent the testing data
    ## outputs: the accuracy of the network and the confusion matrix
    def predict(self,testx,testy):
        act,der = self.__ForwardPropagation(testx,self.weights,self.
↪activationFunctions)
        act[len(act)-1] = self.__ConvertProbabilityToClass(act[len(act)-1])
        predictedValues = act[len(act)-1]
        conf = confusion_matrix(testy,predictedValues)
        acc = accuracy_score(testy,predictedValues)
        print(f'The accuracy is: {acc*100}')
        print("Confusion Matrix: ")
        sns.heatmap(conf,annot=True,fmt='g')
        plt.show()
        return acc

    ## This method is used to plot the error over the training period
    ## Inputs:
    ## error: The error which is caluclated in the gradient descent method and␣
↪returned
    def __plotError(self,error):
        x = np.arange(1,(len(error)+1))
        sns.lineplot(x = x ,y = error)
        plt.title("Error vs Epochs")
        plt.xlabel("Epochs")
        plt.ylabel("Cost")
        plt.show()

    ## This is the method we use to interact with the neural network and train␣
↪it
    ## TrainingX and TrainingY represent the training data
    ## NumberOfHiddenLayers is an integer which represents the number of hidden␣
↪layers in the architecture
    ## NumberOfNeuronsPerHiddenLayer is a list whose size must equal␣
↪NumberOfHiddenLayers. Each entry in the list corresponds to the number of␣
↪neurons
    ## for that hidden layer
    ## NumberOfNeuronsInOutputLayer is an integer indicating the number of␣
↪neurons in the output layer
    ## ActivationFunctions is a list of the activation functions to be used in␣
↪the layers. They may vary for each layer. Must include activations for all␣
↪layers!
```

```
    ## the total number of activations specified should be equal to the number␣
↪of hidden layers+1.
    ## Epochs denotes the maximum number of epoch to allow for training.
    def␣
↪fit(self,TrainingX,TrainingY,NumberOfHiddenLayers,NumberOfNeuronsPerHiddenLayer,NumberOfNeu
↪
        Number_of_datapoints, Number_of_features = np.shape(TrainingX)

        Weight_Parameters = self.
↪__initialiseWeights(Number_of_features,NumberOfHiddenLayers,NumberOfNeuronsPerHiddenLayer,N
        assert(len(ActivationFunctions) ==␣
↪NumberOfHiddenLayers+NumberOfNeuronsInOutputLayer), "Please ensure that the␣
↪number of activation functions specified is the same as the layers!"
        Weights,epochs_to_converge,error = self.
↪__GradientDescent(TrainingX,TrainingY,Weight_Parameters,Number_of_datapoints,learningRate,r
        self.__plotError(error)
        self.weights = Weights
        self.activationFunctions = ActivationFunctions
        return Weights,epochs_to_converge,error
```

8 Standardization and splitting of the data

We standardize the data since this will help the convergence of Gradient Descent. We also make the training size 70% of the dataset. The validation and testing size is further split by splitting the 30% remaining data into 70% for the validation and 30% for the testing data.

[6]:
```
trainx,validationx,testx, trainy,validationy,testy = Dataset().
↪Split_Data(data,0.3,True)
```

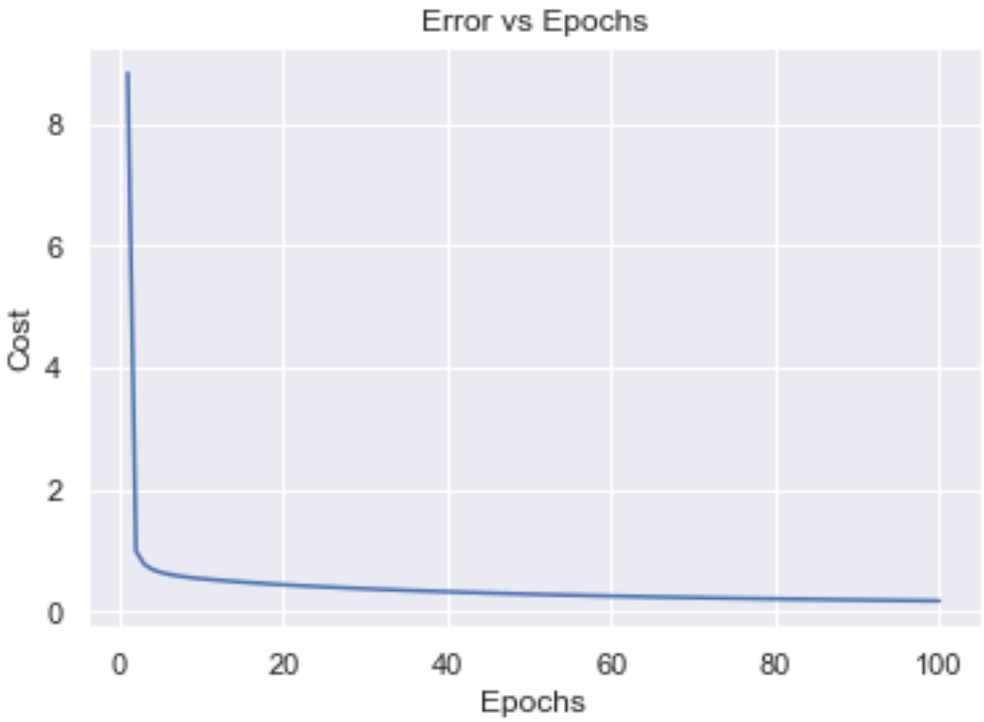
9 2. Investigating the effects of utilising varying activation functions

For this section, the architecture will remain constant. We will be using a neural network with 2 hidden layers, 2 neurons per hidden layer and 1 neuron in the final layer. We will use a learning rate of 0.1, with no regularisation and a maximum epochs of 100.

9.1 2.1 Using ReLu as the only activation function

[7]:
```
NN_Relu = NeuralNetwork()
print("The training error graph: ")
Relu_Returns =  NN_Relu.fit(trainx,trainy,2,[2,2],1,["relu","relu","relu"],0.
↪1,0,100)
```

```
The training error graph:
```



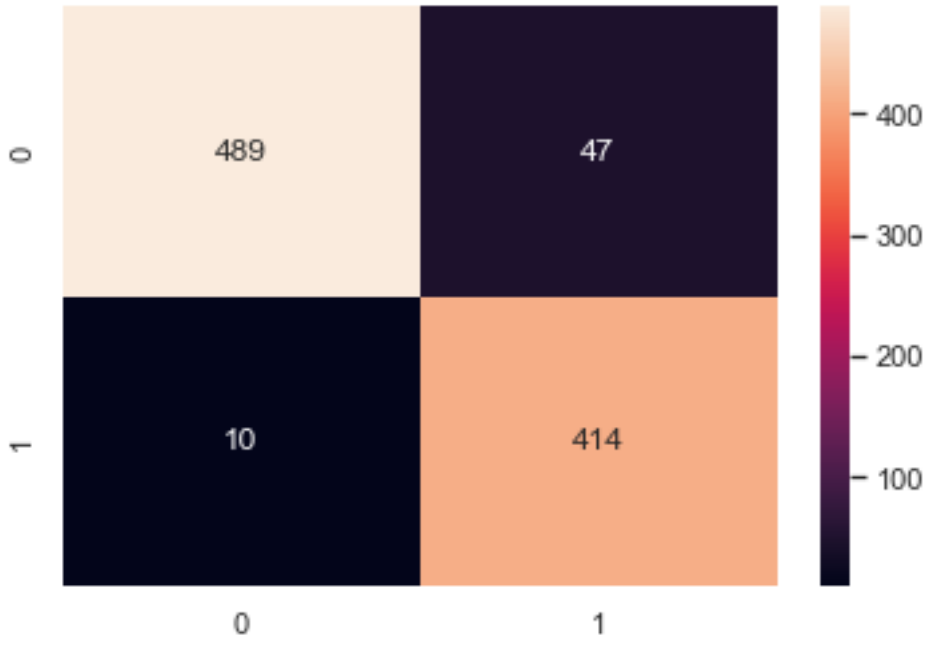
[8]:
```
print(f'The number of epochs taken to converge is: {Relu_Returns[1]}')
```

```
The number of epochs taken to converge is: 100
```

9.1.1 2.1.1 Training Accuracy

[9]:
```
Relu_Training_Acc = NN_Relu.predict(trainx,trainy)
```

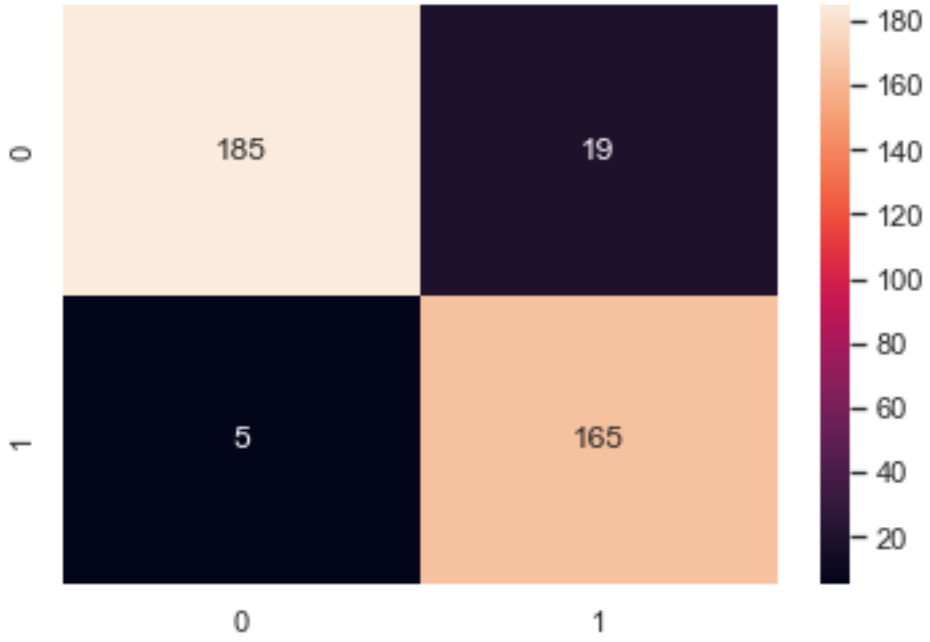
```
The accuracy is: 94.0625
Confusion Matrix:
```



9.1.2 2.1.2 Validation Accuracy

[10]:
```
Relu_Validation_Acc = NN_Relu.predict(validationx,validationy)
```

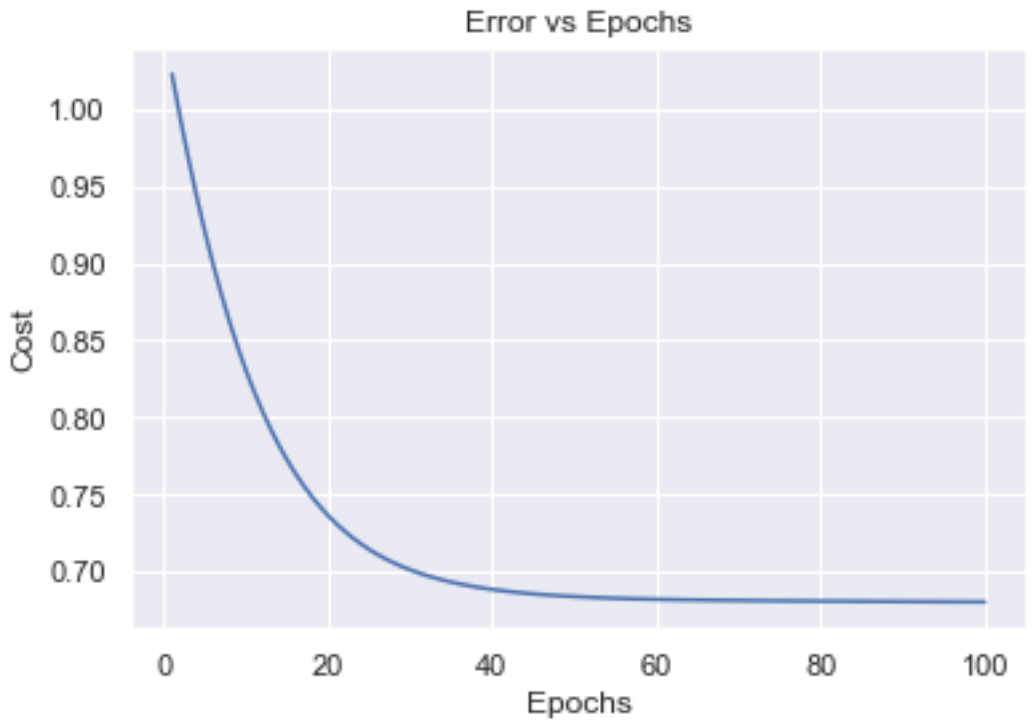
```
The accuracy is: 93.58288770053476
Confusion Matrix:
```



9.2 2.2 Using Sigmoid as the only activation function

[11]:
```
NN_Sigmoid = NeuralNetwork()
print("The training error graph: ")
Sigmoid_Returns =  NN_Sigmoid.
 ↪fit(trainx,trainy,2,[2,2],1,["sigmoid","sigmoid","sigmoid"],0.1,0,100)
```

```
The training error graph:
```



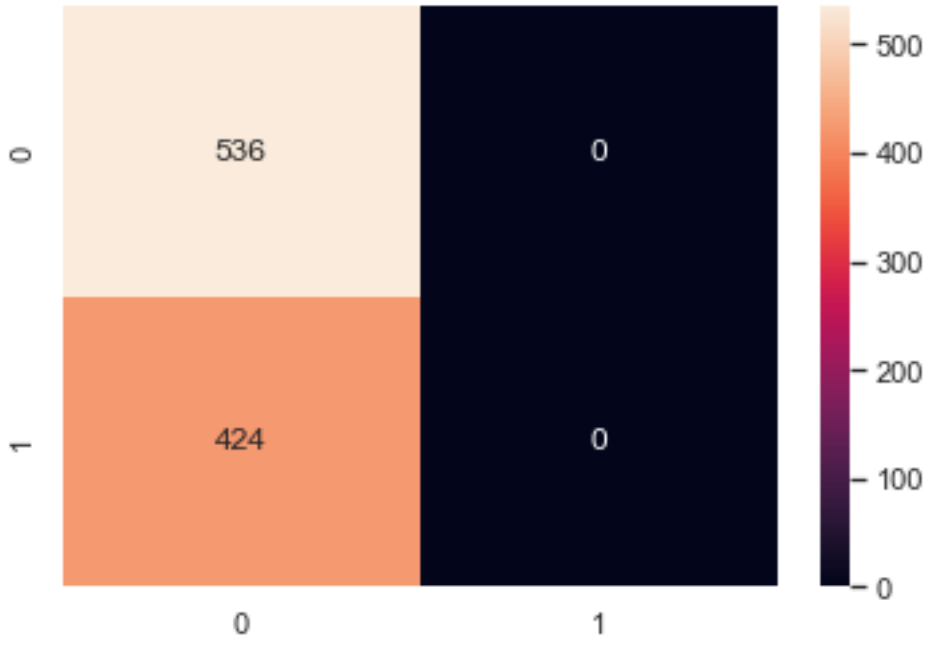
[12]:
```
print(f'The number of epochs taken to converge is: {Sigmoid_Returns[1]}')
```

```
The number of epochs taken to converge is: 100
```

9.2.1 2.2.1 Training Accuracy

[13]:
```
Sigmoid_Training_Acc = NN_Sigmoid.predict(trainx,trainy)
```

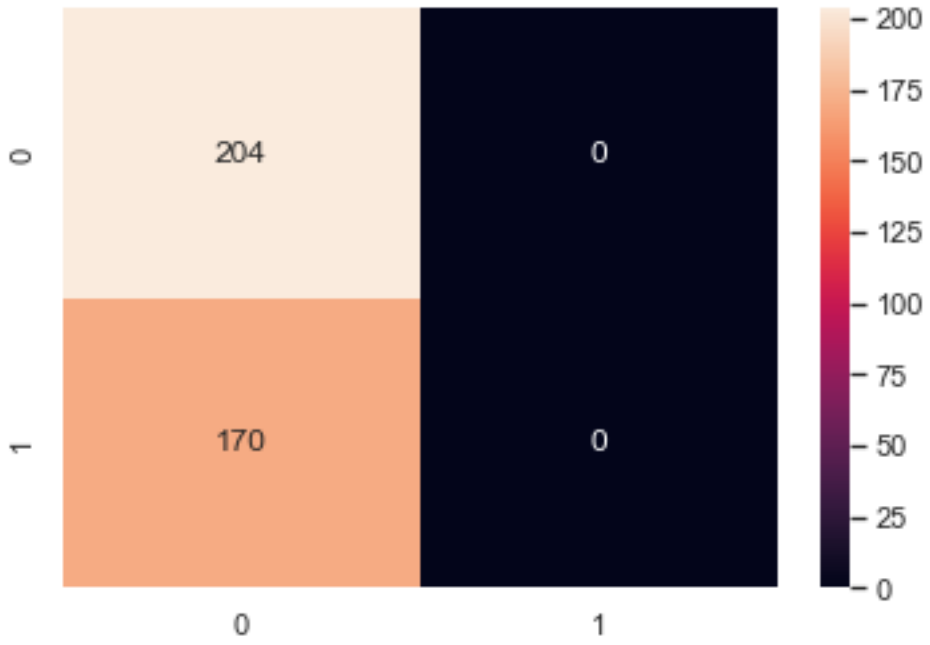
```
The accuracy is: 55.833333333333336
Confusion Matrix:
```



9.2.2 2.2.2 Validation Accuracy

[14]:
```
Sigmoid_Validation_Acc = NN_Sigmoid.predict(validationx,validationy)
```

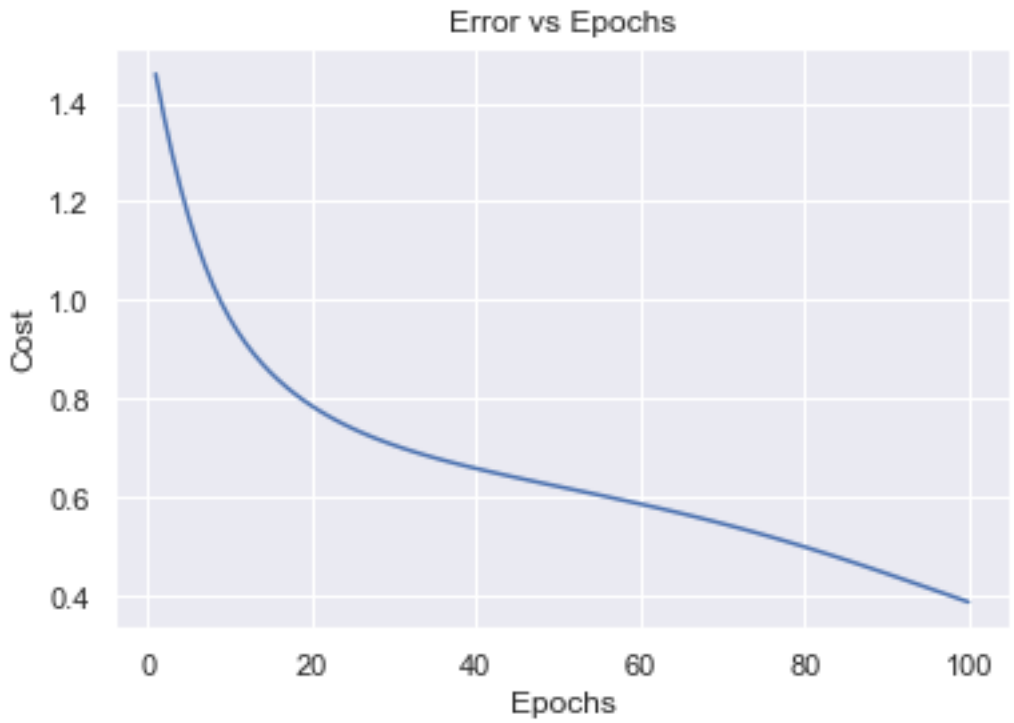
```
The accuracy is: 54.54545454545454
Confusion Matrix:
```



9.3 2.3 Using Tanh as the only activation function

[15]:
```
NN_Tanh = NeuralNetwork()
print("The training error graph: ")
Tanh_Returns =  NN_Tanh.fit(trainx,trainy,2,[2,2],1,["tanh","tanh","tanh"],0.
↪1,0,100)
```

```
The training error graph:
```



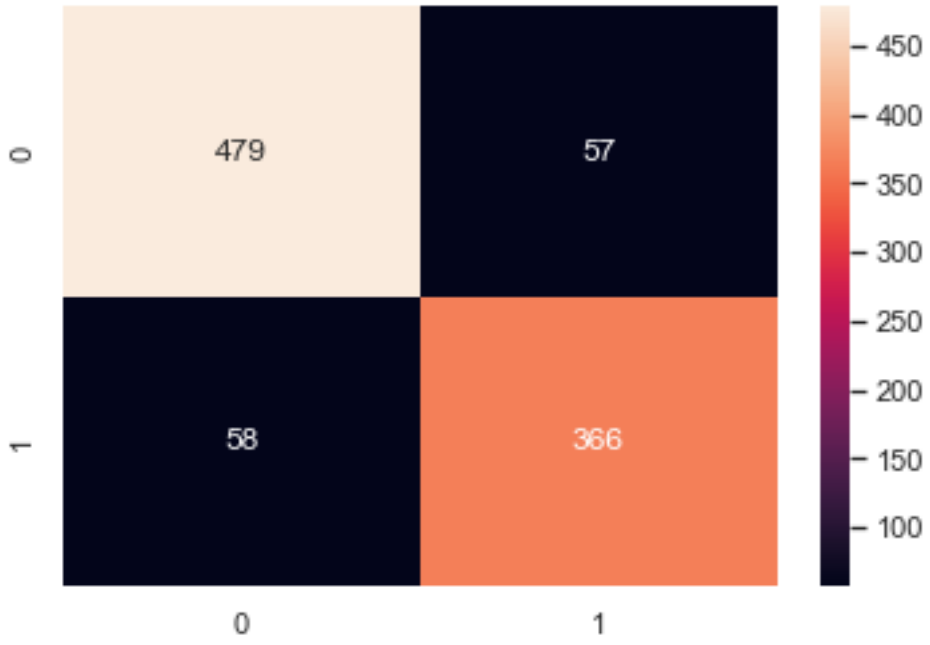
[16]:
```
print(f'The number of epochs taken to converge is: {Tanh_Returns[1]}')
```

```
The number of epochs taken to converge is: 100
```

9.3.1 2.3.1 Training Accuracy

[17]:
```
Tanh_Training_Accuracy = NN_Tanh.predict(trainx,trainy)
```

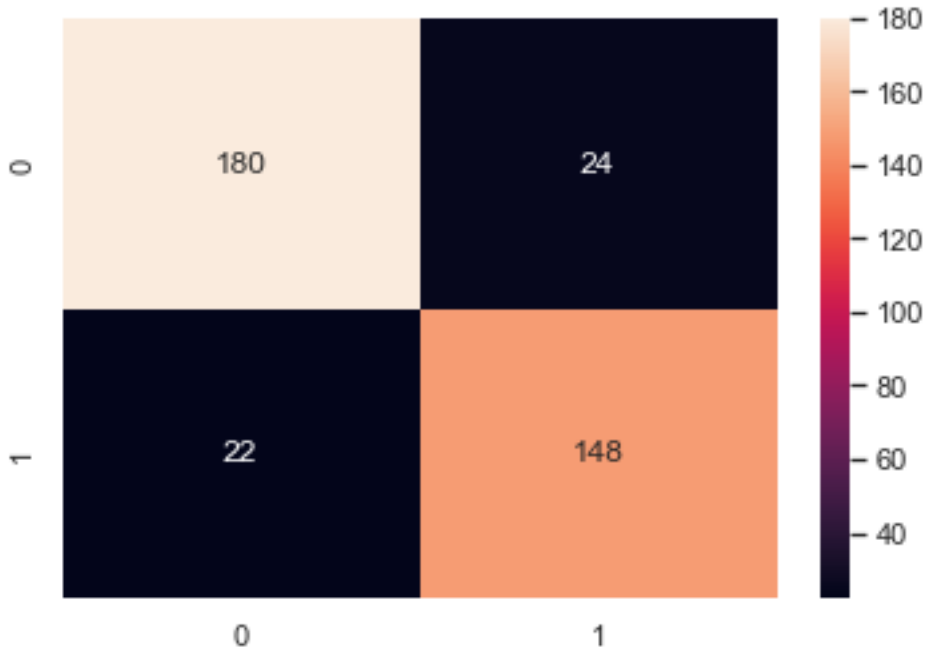
```
The accuracy is: 88.02083333333334
Confusion Matrix:
```



9.3.2 2.3.2 Validation Accuracy

[18]:
```
Tanh_Validation_Accuracy = NN_Tanh.predict(validationx,validationy)
```

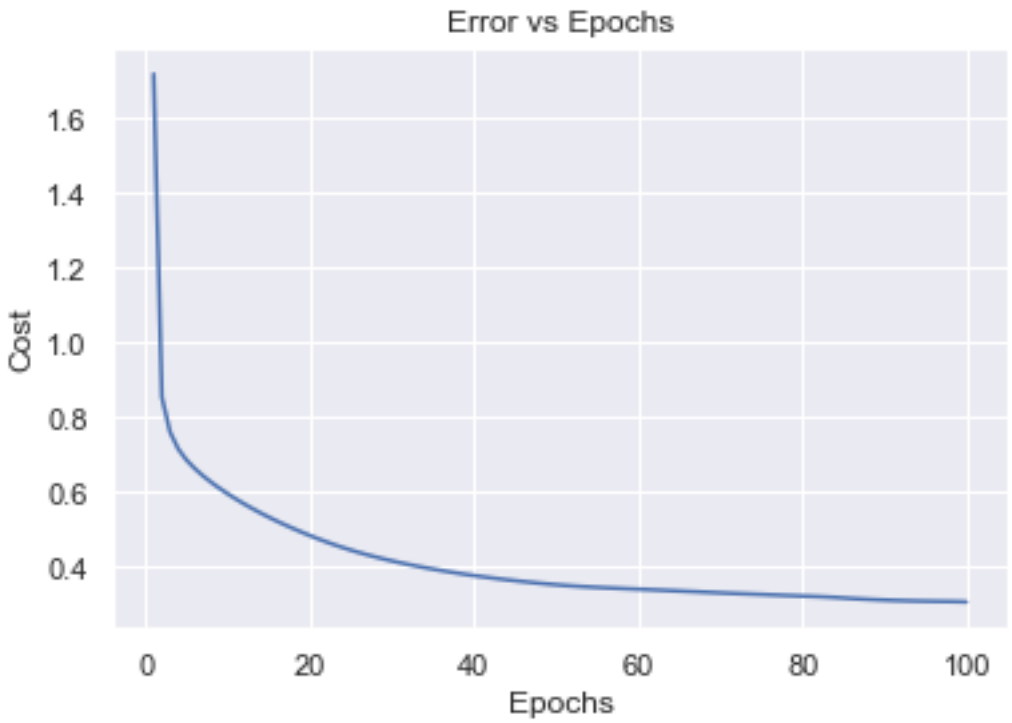
```
The accuracy is: 87.70053475935828
Confusion Matrix:
```



9.4 2.4 Using Leaky ReLu as the only activation function

[19]:
```
NN_LeakyRelu = NeuralNetwork()
print("The training error graph: ")
LeakyRelu_Returns = NN_LeakyRelu.fit(trainx,trainy,2,[2,2],1,["leaky␣
↪relu","leaky relu","leaky relu"],0.1,0,100)
```

```
The training error graph:
```



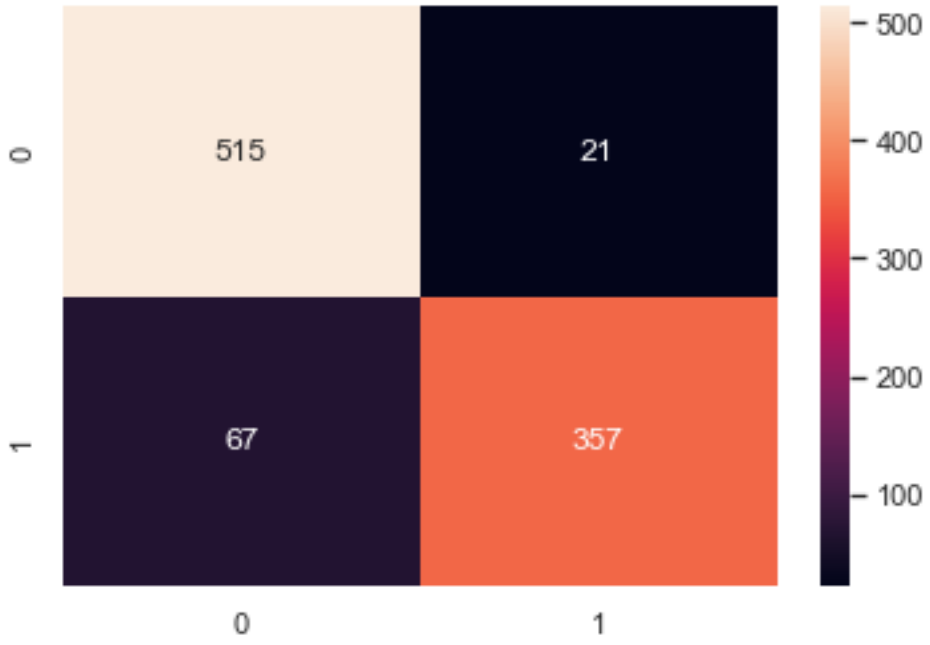
[20]:
```
print(f'The number of epochs taken to converge is: {LeakyRelu_Returns[1]}')
```

```
The number of epochs taken to converge is: 100
```

9.4.1 2.4.1 Training Accuracy

[21]:
```
LeakyRelu_Training_Acc = NN_LeakyRelu.predict(trainx,trainy)
```

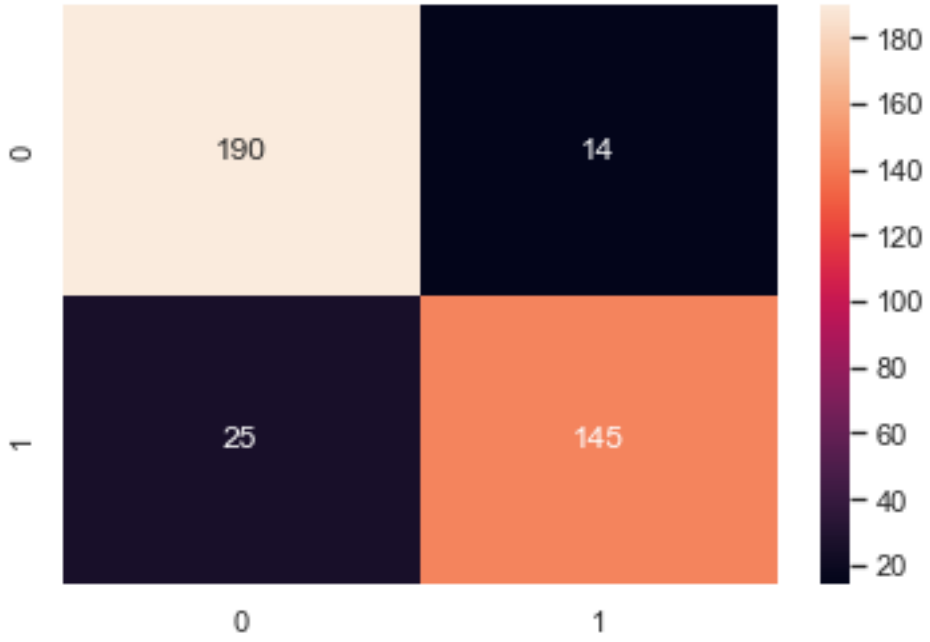
```
The accuracy is: 90.83333333333333
Confusion Matrix:
```



9.4.2 2.4.2 Validation Accuracy

[22]:
```
LeakyRelu_Validation_Acc = NN_LeakyRelu.predict(validationx,validationy)
```

```
The accuracy is: 89.57219251336899
Confusion Matrix:
```



9.5 2.5 Tabularized Results

[23]:
```
Activation_Table = PrettyTable(["Activation Function","Epochs to␣
↪train","Training Accuracy","Validation Accuracy"])

Activation_Table.add_row(["ReLu",Relu_Returns[1],np.
↪round(Relu_Training_Acc*100,2),np.round(Relu_Validation_Acc*100,2)])
Activation_Table.add_row(["Sigmoid",Sigmoid_Returns[1],np.
↪round(Sigmoid_Training_Acc*100,2),np.round(Sigmoid_Validation_Acc*100,2)])
Activation_Table.add_row(["Tanh",Tanh_Returns[1],np.
↪round(Tanh_Training_Accuracy*100,2),np.
↪round(Tanh_Validation_Accuracy*100,2)])
Activation_Table.add_row(["Leaky ReLu",LeakyRelu_Returns[1],np.
↪round(LeakyRelu_Training_Acc*100,2),np.
↪round(LeakyRelu_Validation_Acc*100,2)])
```

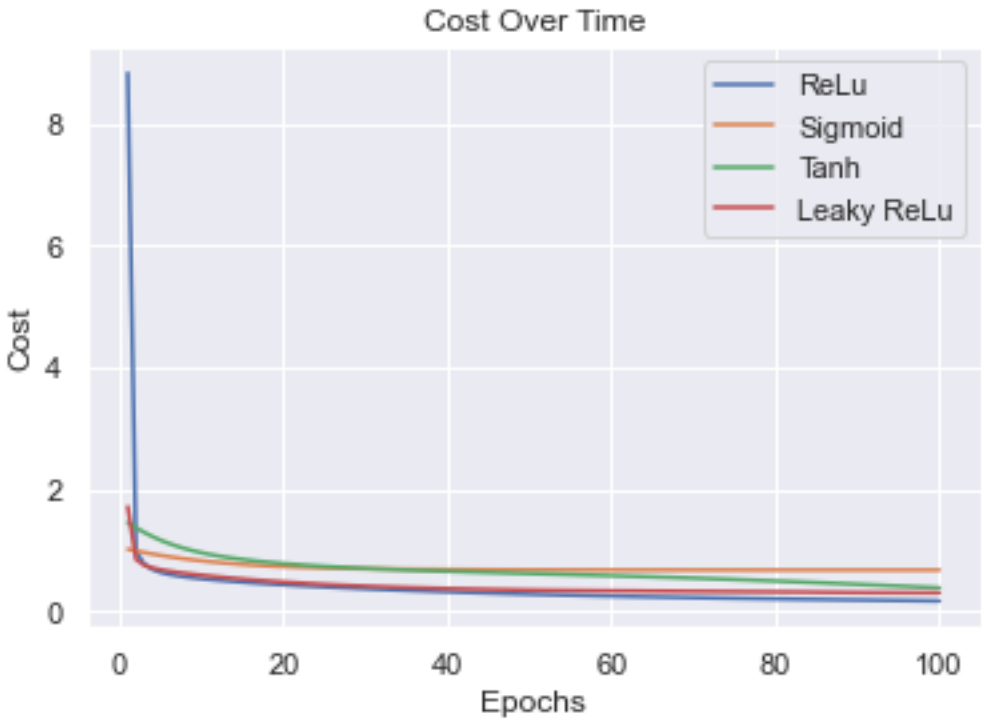
[24]:
```
print(Activation_Table)
```

```
+---------------------+-----------------+-------------------+-------------------
--+
| Activation Function | Epochs to train | Training Accuracy | Validation
Accuracy |
+---------------------+-----------------+-------------------+-------------------
--+
|         ReLu        |       100       |       94.06       |        93.58
|
|       Sigmoid       |       100       |       55.83       |        54.55
|
|         Tanh        |       100       |       88.02       |         87.7
|
|      Leaky ReLu     |       100       |       90.83       |        89.57
|
+---------------------+-----------------+-------------------+-------------------
--+
```

9.6 2.6 Discussion of Results

[108]:
```
errors =␣
↪[Relu_Returns[2],Sigmoid_Returns[2],Tanh_Returns[2],LeakyRelu_Returns[2]]
labels = ["ReLu","Sigmoid","Tanh","Leaky ReLu"]
for i in range(len(errors)):
    x = np.arange(1,(len(errors[i])+1))
    plt.plot(x,errors[i],label = labels[i])
    plt.xlabel("Epochs")
```

```
    plt.ylabel("Cost")
    plt.title("Cost Over Time")

plt.legend()
plt.show()
```



- The first thing we notice is that all the networks used the maximum epochs to train.
- If we look at the validation accuracies, we see that the leaky relu and ReLu perform the best. This could be attributed to the fact that the Leaky ReLu does not suffer from the vanishing gradients problem and the ReLu does not suffer as bad as Tanh and the sigmoid. Consequently they are able to learn better and outperform the other models that do have vanishing gradients.
- We also see that the sigmoid performs the worst. This is because the other functions have steeper derivatives and thus provide more value during backpropagation and enable better learning.
- If we look at the graphs, we see that ReLu and Leaky ReLu have te fastest initial error decrease whilst the sigmoid has the lowest. As a result, by the time the max epochs are reached, the sigmoid weights are not sufficiently adjusted.
- From these results, we can choose the Leaky Relu as the optimal activation function since we are certain it does not suffer from vanishing gradients.

10 3. Exploring the effect of network size on generalizability

In order to study the effect of the networks architecture, we will have to keep the activation function constant. We will use a Leaky ReLuin every layer. We will use a learning rate of 0.1, with no regularisation and a maximum epochs of 100.

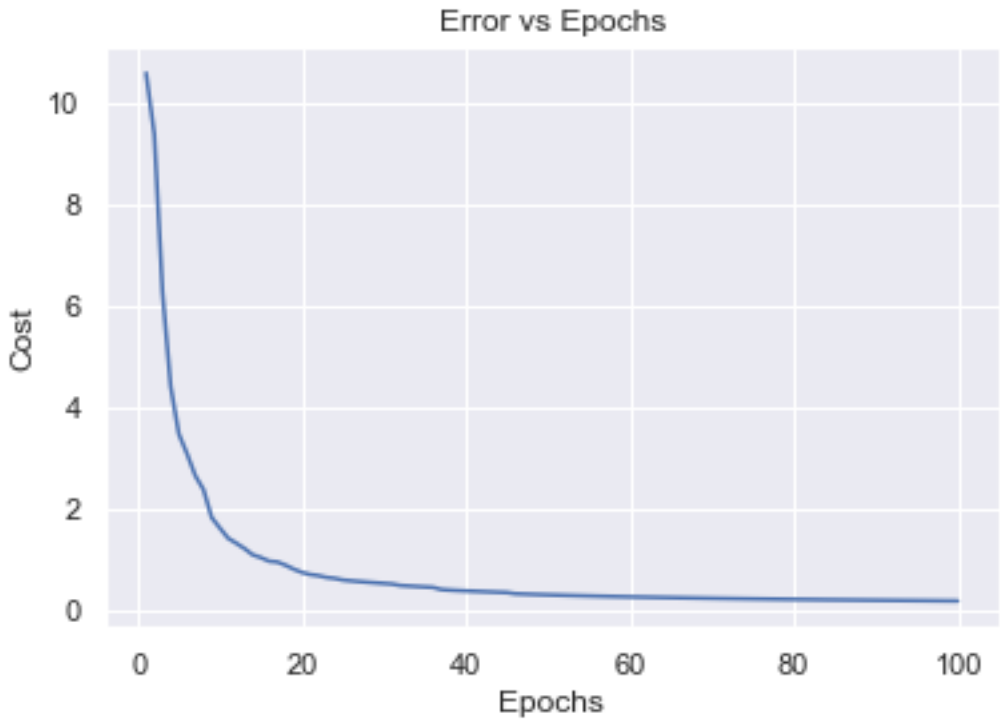
10.1 3.1 Investigating the effect of hidden layers

We will keep the number of neurons in a hidden layer constant. We will use 4 neurons per hidden layer

10.1.1 3.1.1 Using 1 Hidden Layer

[25]:
```
NN_One_Hidden_Layer = NeuralNetwork()
print("The training error graph: ")
NN_One_Hidden_Layer_Returns = NN_One_Hidden_Layer.
 ↪fit(trainx,trainy,1,[4],1,["leaky relu","leaky relu"],0.1,0,100)
```

```
The training error graph:
```



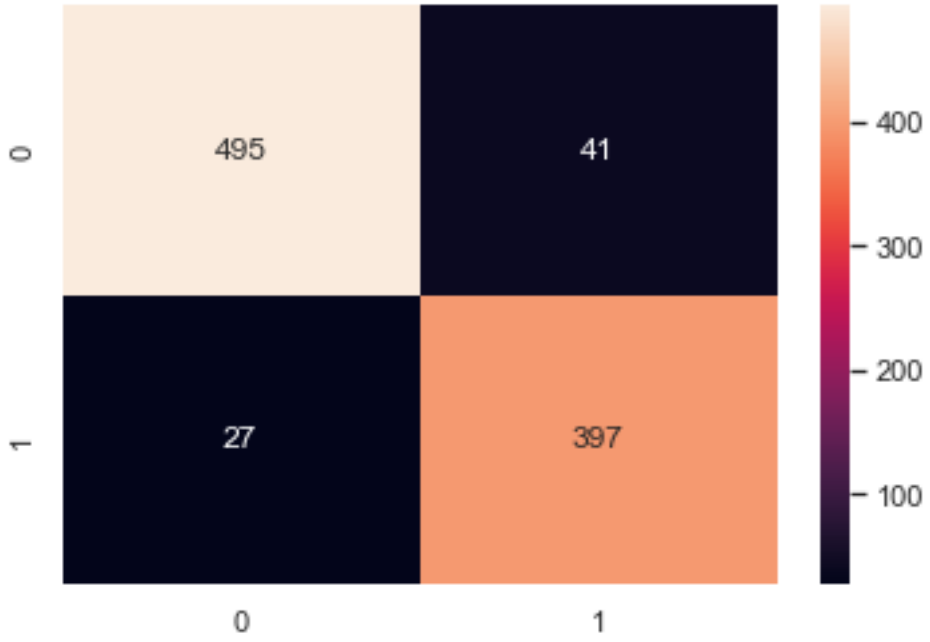
[26]:
```
print(f"The number of epochs taken to converge is:␣
 ↪{NN_One_Hidden_Layer_Returns[1]}")
```

```
The number of epochs taken to converge is: 100
```

10.1.2 3.1.1.1 Training Accuracy

[27]:
```
One_Hidden_Layer_Training_Acc = NN_One_Hidden_Layer.predict(trainx,trainy)
```

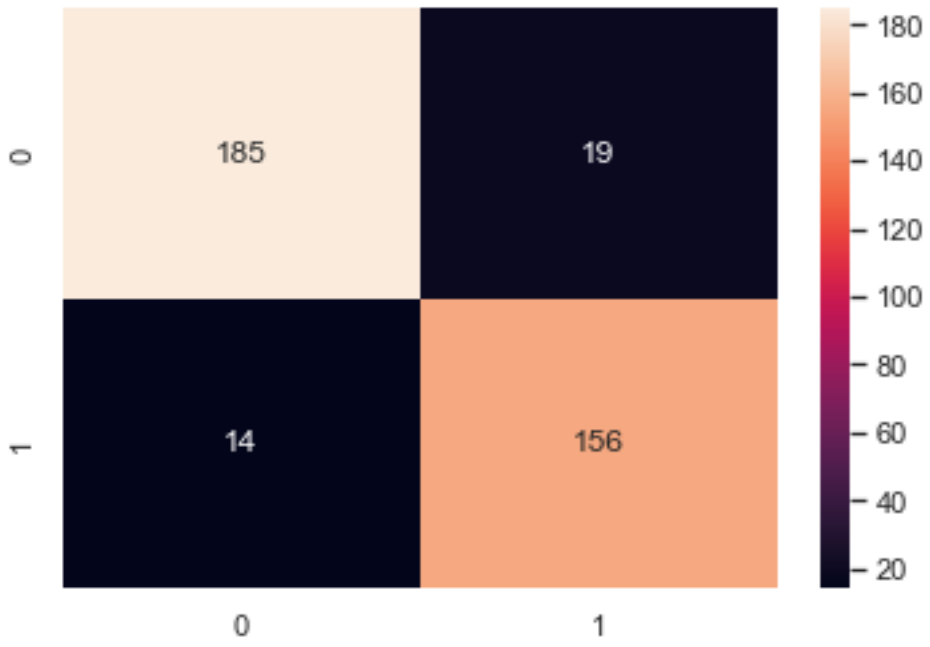
```
The accuracy is: 92.91666666666667
Confusion Matrix:
```



10.1.3 3.1.1.2 Validation Accuracy

[28]:
```
One_Hidden_Layer_Validation_Acc = NN_One_Hidden_Layer.
↪predict(validationx,validationy)
```

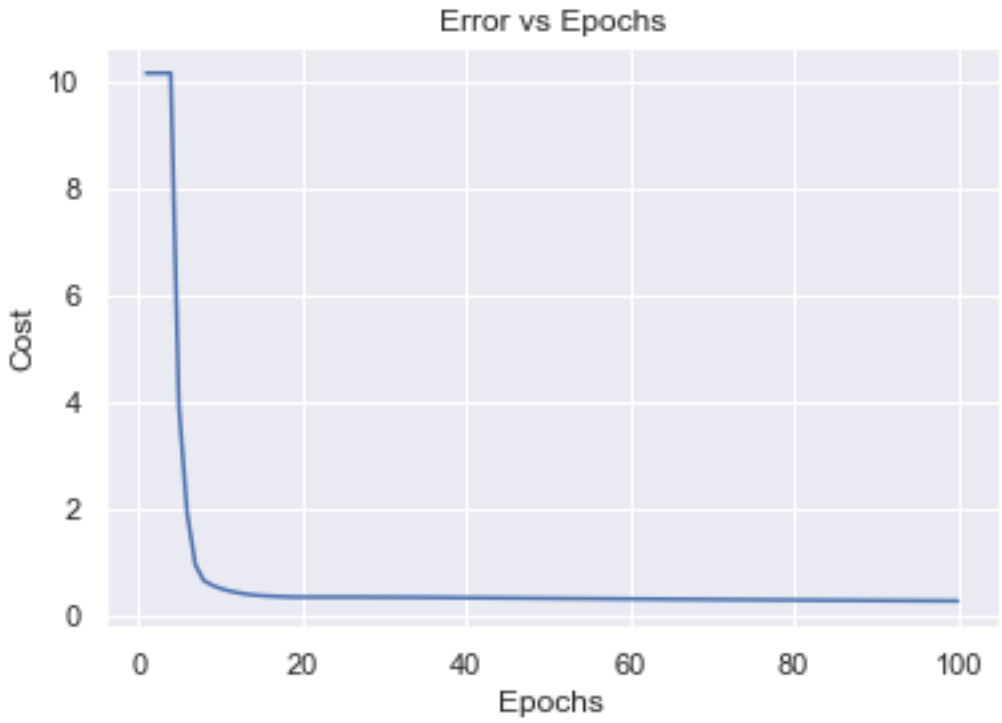
```
The accuracy is: 91.17647058823529
Confusion Matrix:
```



10.1.4 3.1.2 Using 2 Hidden Layers

[29]:
```
NN_Two_Hidden_Layers = NeuralNetwork()
print("The training error graph: ")
NN_Two_Hidden_Layers_Returns = NN_Two_Hidden_Layers.
 ↪fit(trainx,trainy,2,[4,4],1,["leaky relu","leaky relu","leaky relu"],0.
 ↪1,0,100)
```

The training error graph:



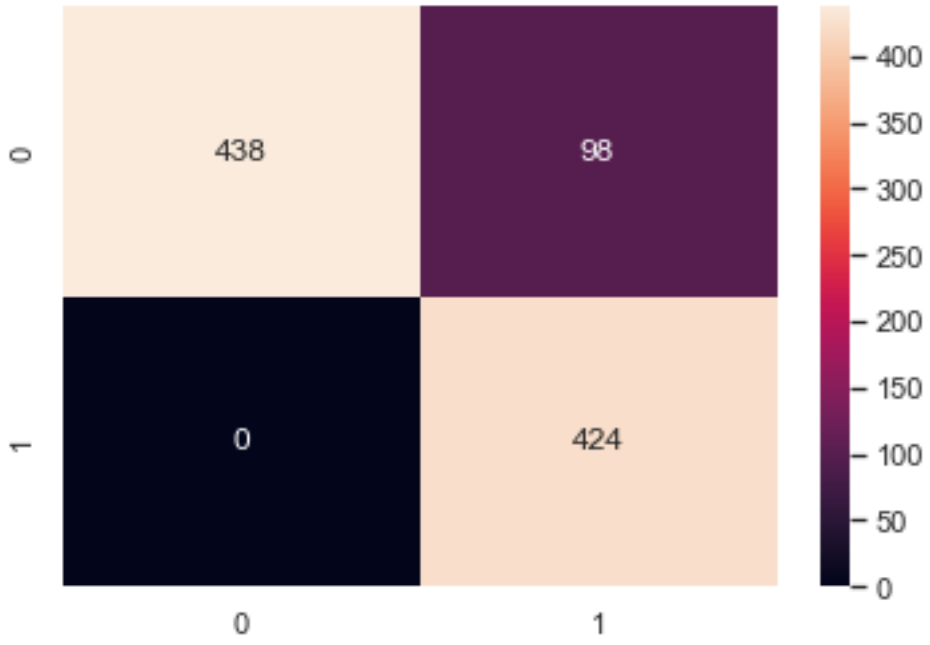
[30]:
```
print(f"The number of epochs taken to converge is:␣
↪{NN_Two_Hidden_Layers_Returns[1]}")
```

```
The number of epochs taken to converge is: 100
```

10.1.5 3.1.2.1 Training Accuracy

[31]:
```
Two_Hidden_Layers_Training_Acc = NN_Two_Hidden_Layers.predict(trainx,trainy)
```

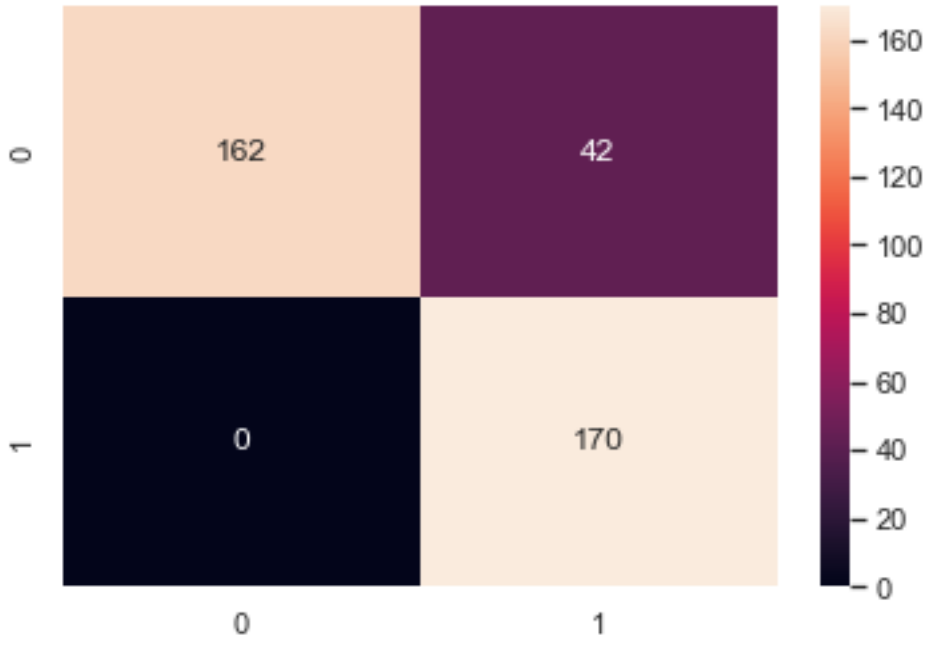
```
The accuracy is: 89.79166666666667
Confusion Matrix:
```



10.1.6 3.1.2.2 Validation Accuracy

[32]:
```
Two_Hidden_Layers_Validation_Acc = NN_Two_Hidden_Layers.
↪predict(validationx,validationy)
```

```
The accuracy is: 88.77005347593582
Confusion Matrix:
```



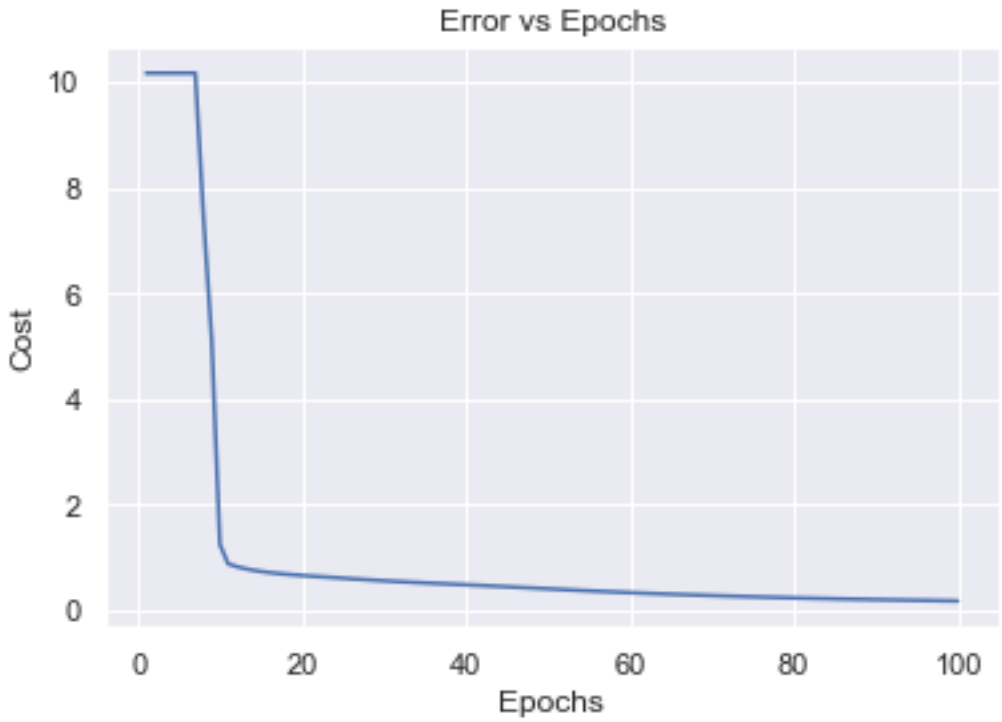
10.2 3.2 Investigating the effects of the number of Neurons in a hidden layer

We will keep the number of layers constant. We will use 2 hidden layers. We will use a sigmoid as the activation function. We will use a learning rate of 0.1, with no regularisation and a maximum epochs of 100.

10.2.1 3.2.1 Using 5 Neurons Per Hidden Layer

[33]:
```
NN_Five_Neurons = NeuralNetwork()
print("The training error graph: ")
NN_Five_Neurons_Returns = NN_Five_Neurons.fit(trainx,trainy,2,[5,5],1,["leaky␣
↪relu","leaky relu","leaky relu"],0.1,0,100)
```

```
The training error graph:
```



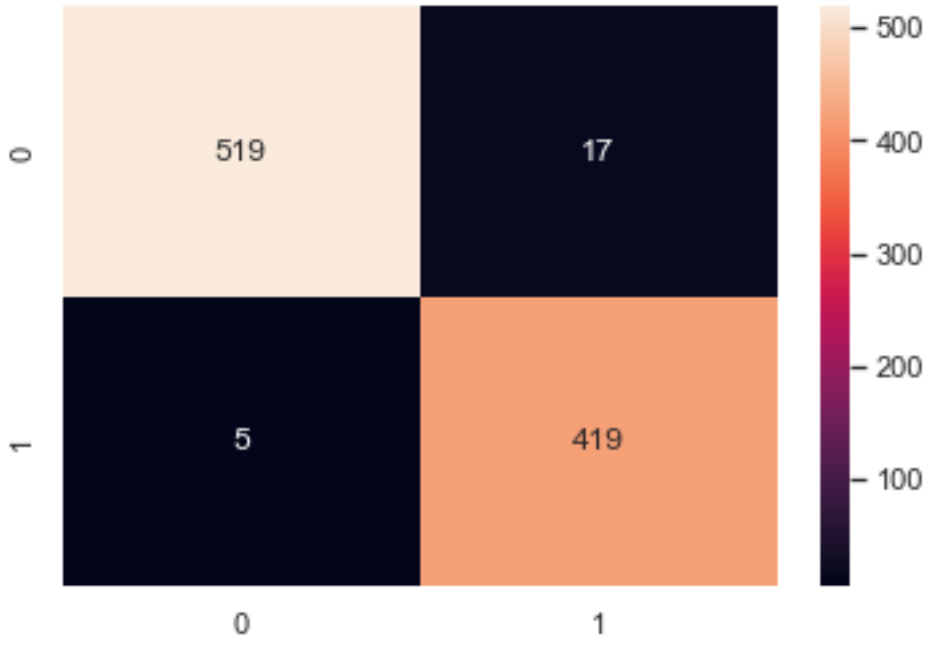
[34]:
```
print(f'The number of epochs taken to converge is:␣
↪{NN_Five_Neurons_Returns[1]}')
```

```
The number of epochs taken to converge is: 100
```

10.2.2 3.2.1.1Training Accuracy

[35]:
```
NN_Five_Neurons_Training_Acc = NN_Five_Neurons.predict(trainx,trainy)
```

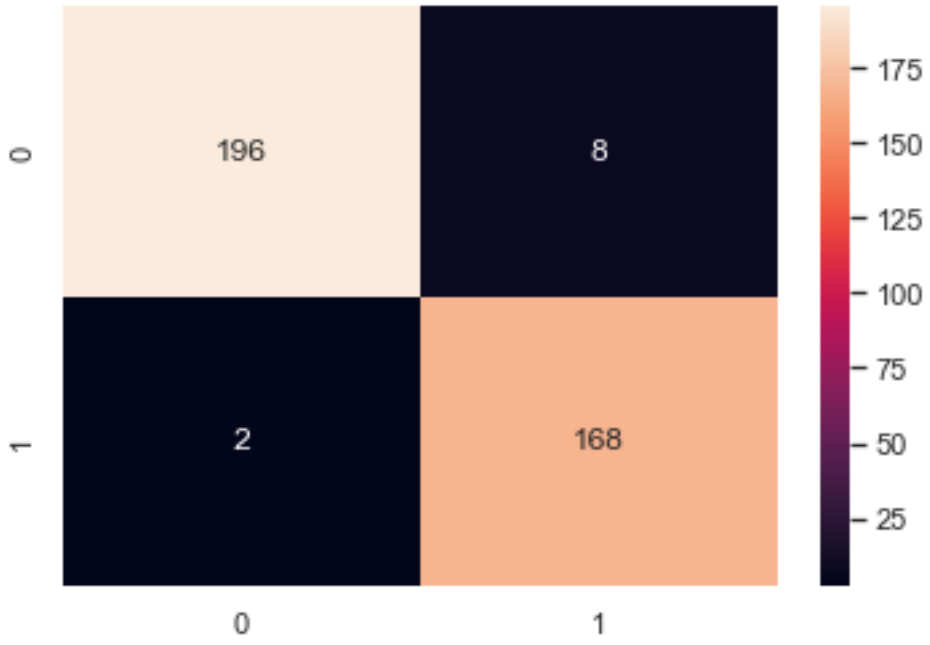
```
The accuracy is: 97.70833333333333
Confusion Matrix:
```



10.2.3 3.2.1.2 Validation Accuracy

[36]:
```
NN_Five_Neurons_Validation_Acc = NN_Five_Neurons.
↪predict(validationx,validationy)
```

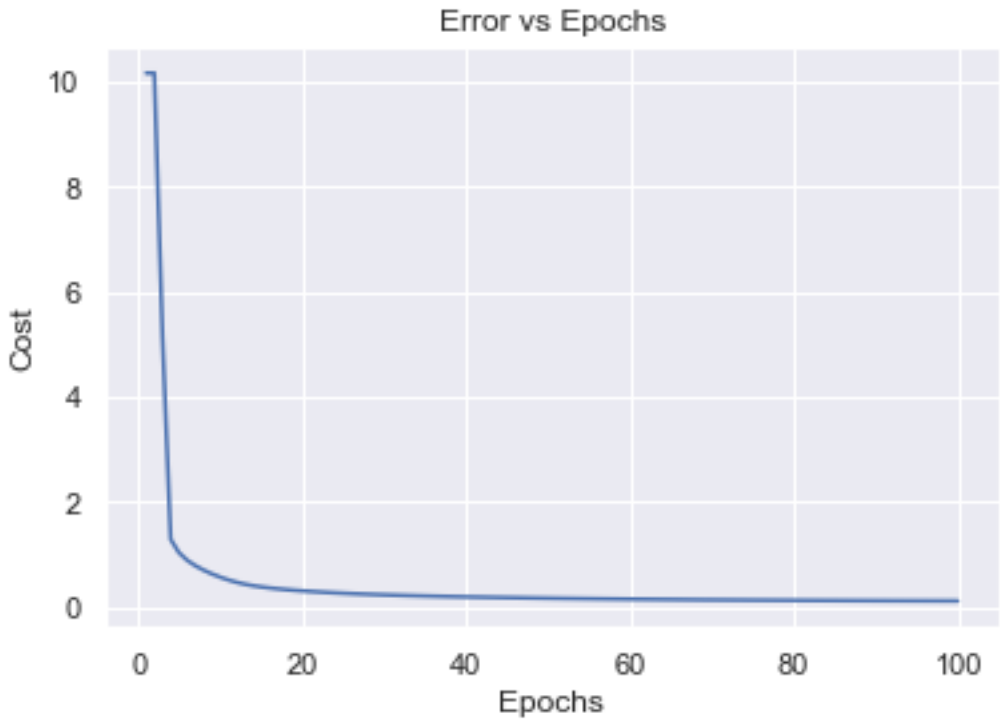
```
The accuracy is: 97.32620320855615
Confusion Matrix:
```



10.2.4 3.2.2 Using 6 Neurons Per Hidden Layer

[37]:
```
NN_Six_Neurons = NeuralNetwork()
print("The training error graph: ")
NN_Six_Neurons_Returns = NN_Six_Neurons.fit(trainx,trainy,2,[5,5],1,["leaky␣
↪relu","leaky relu","leaky relu"],0.1,0,100)
```

```
The training error graph:
```



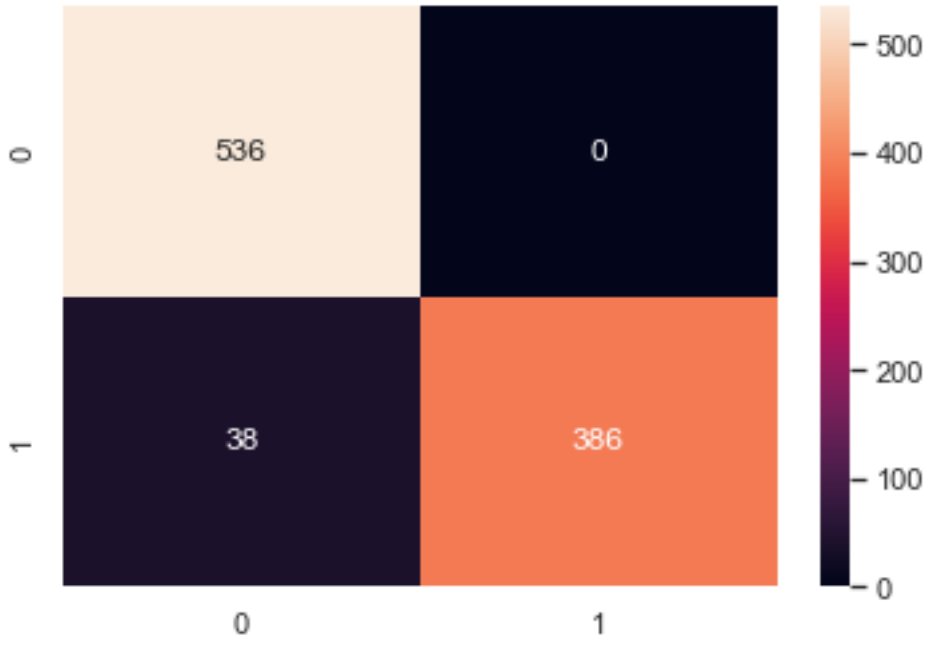
[38]:
```
print(f'The number of epochs taken to converge is: {NN_Six_Neurons_Returns[1]}')
```

```
The number of epochs taken to converge is: 100
```

10.2.5 3.2.2.1 Training Accuracy

[39]:
```
NN_Six_Neurons_Training_Acc = NN_Six_Neurons.predict(trainx,trainy)
```

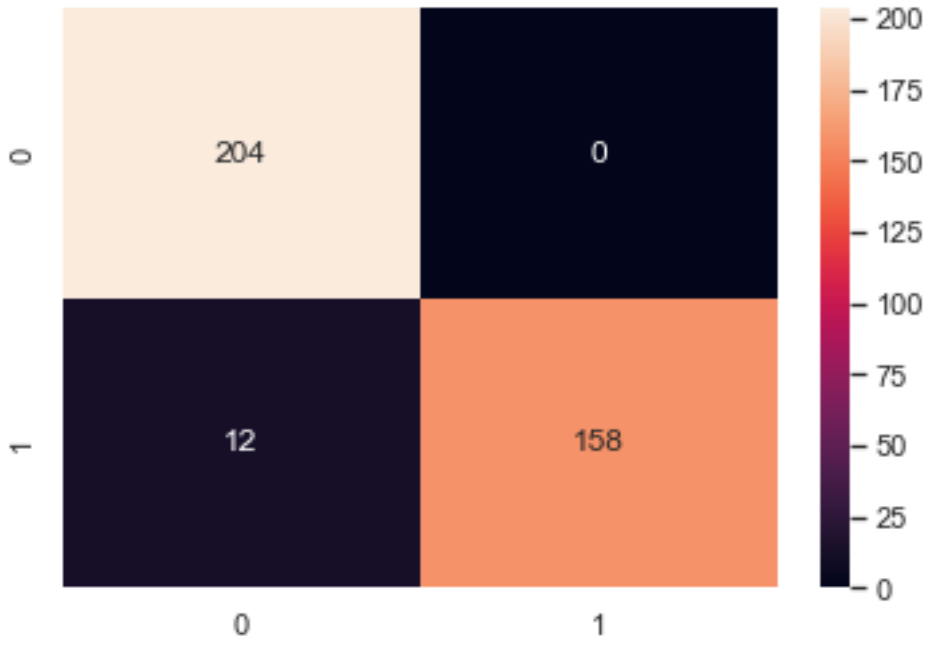
```
The accuracy is: 96.04166666666667
Confusion Matrix:
```



10.2.6 3.2.2.2 Validation Accuracy

[40]:
```
NN_Six_Neurons_Validation_Acc = NN_Six_Neurons.predict(validationx,validationy)
```

```
The accuracy is: 96.79144385026738
Confusion Matrix:
```



10.3 3.3 Tabularized Results

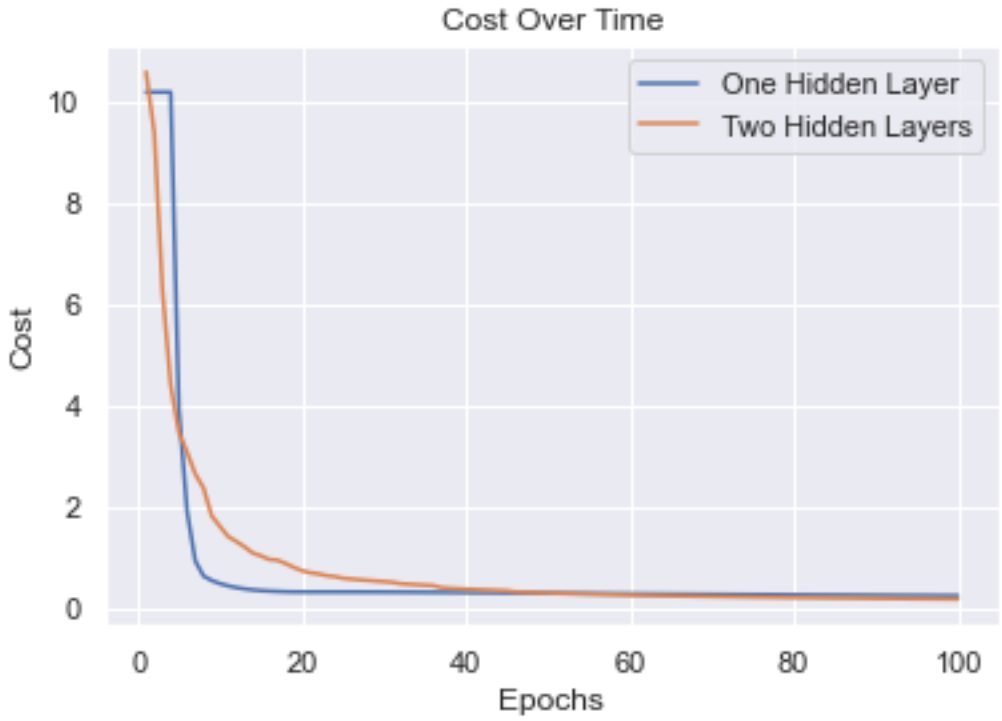
[41]:
```
results_Table = PrettyTable(["Number of Hidden Layers",
                             "Number of Neurons in Hidden Layer",
                             "Epochs to Train",
                             "Training Accuracy",
                             "Validation Accuracy"])

results_Table.add_row([1,4,NN_One_Hidden_Layer_Returns[1],np.
 ↪round(One_Hidden_Layer_Training_Acc*100,2),np.
 ↪round(One_Hidden_Layer_Validation_Acc*100,2)])
results_Table.add_row([2,4,NN_Two_Hidden_Layers_Returns[1],np.
 ↪round(Two_Hidden_Layers_Training_Acc*100,2),np.
 ↪round(Two_Hidden_Layers_Validation_Acc*100,2)])
results_Table.add_row([2,5,NN_Five_Neurons_Returns[1],np.
 ↪round(NN_Five_Neurons_Training_Acc*100,2),np.
 ↪round(NN_Five_Neurons_Validation_Acc*100,2)])
results_Table.add_row([2,6,NN_Six_Neurons_Returns[1],np.
 ↪round(NN_Six_Neurons_Training_Acc*100,2),np.
 ↪round(NN_Six_Neurons_Validation_Acc*100,2)])
```

[42]:
```
print(results_Table)
```

```
+-------------------------+-----------------------------------+-----------------
+------------------+--------------------+
| Number of Hidden Layers | Number of Neurons in Hidden Layer | Epochs to Train
| Training Accuracy | Validation Accuracy |
+-------------------------+-----------------------------------+-----------------
+------------------+--------------------+
|            1            |                 4                 |       100
|       92.92       |        91.18       |
|            2            |                 4                 |       100
|       89.79       |        88.77       |
|            2            |                 5                 |       100
|       97.71       |        97.33       |
|            2            |                 6                 |       100
|       96.04       |        96.79       |
+-------------------------+-----------------------------------+-----------------
+------------------+--------------------+
```

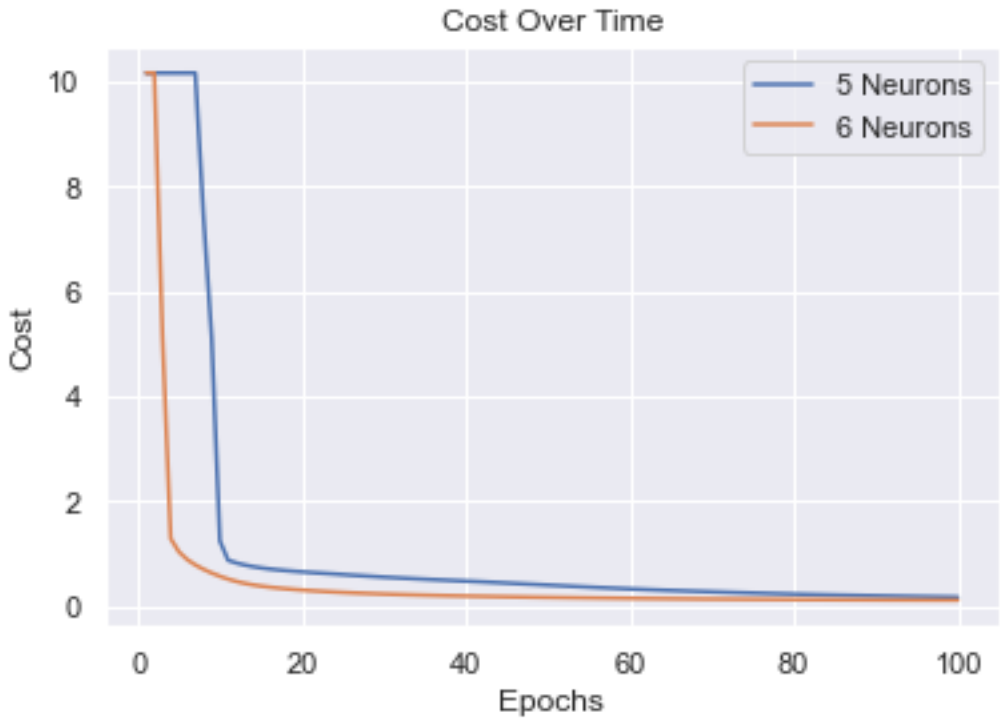
10.4 3.4 Discussion of Results

[111]:
```
errors = [NN_Two_Hidden_Layers_Returns[2],NN_One_Hidden_Layer_Returns[2]]
labels = ["One Hidden Layer","Two Hidden Layers"]
for i in range(len(errors)):
    x = np.arange(1,(len(errors[i])+1))
    plt.plot(x,errors[i],label = labels[i])
    plt.xlabel("Epochs")
    plt.ylabel("Cost")
    plt.title("Cost Over Time")

plt.legend()
plt.show()
```



[112]:
```
errors = [NN_Five_Neurons_Returns[2],NN_Six_Neurons_Returns[2]]
labels = ["5 Neurons","6 Neurons"]
for i in range(len(errors)):
    x = np.arange(1,(len(errors[i])+1))
    plt.plot(x,errors[i],label = labels[i])
    plt.xlabel("Epochs")
    plt.ylabel("Cost")
    plt.title("Cost Over Time")

plt.legend()
```

```
plt.show()
```



- We first notice that having an additional layer increases the training epochs. This is because the architecture is more complex with more weights and thus we need these additional weights to converge as well.
- This is also seen in the above two plots whereby the more complex networks take longer to converge. Looking at the validation accuracy:
- We notice that the network with 2 hidden layers and 6 neurons per hidden layer performs worst than the network with 2 hidden layers but 5 neurons per hidden layer. This indicates that the additional neuron may be too complex for the data and consequently negatively affects the performance.
- Therefore we can see that simply adding neurons is not always going to improve the performance and there is a point where the architecture is optimal and adding additional complexities degragades the performance.
- From these results we can see a two hidden layers with five neurons is optimal.

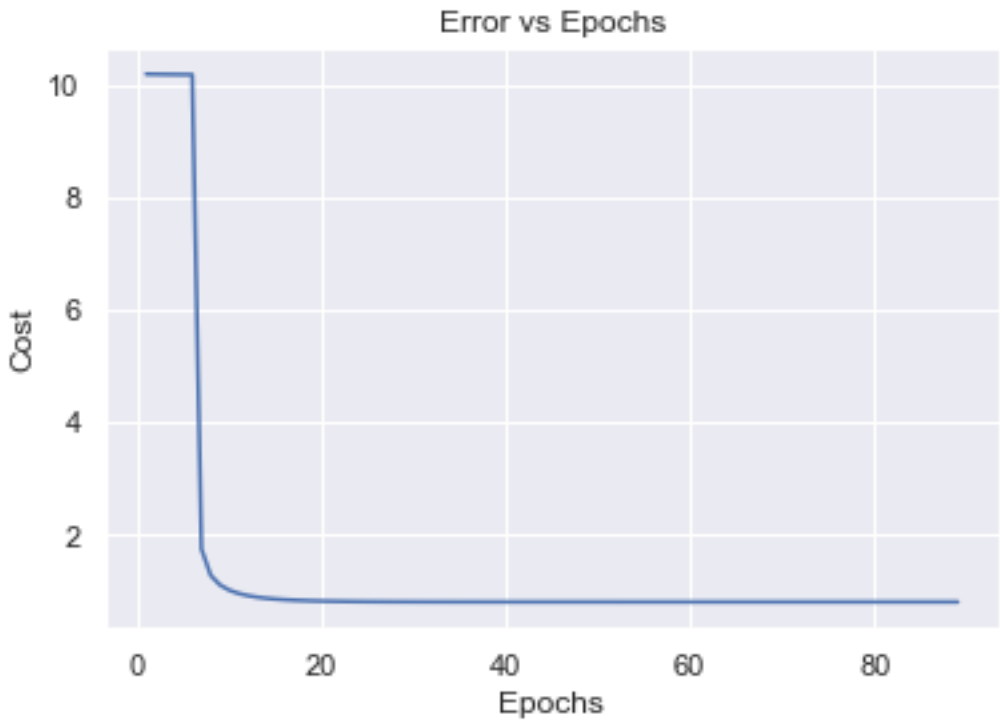
11 4. Investigating The Effect Of Regularization

Since the leaky ReLu achieved the best results we will use it as the activation function. We will also use 2 hidden layers with 5 neurons

11.1 4.1 Setting Regularization = 1

[43]:
```
Reg_NN_C1 = NeuralNetwork()
print("The training error graph: ")
Reg_NN_C1_Results = Reg_NN_C1.fit(trainx,trainy,2,[5,5],1,["leaky relu","leaky␣
↪relu","leaky relu"],0.1,1,100)
```

```
The training error graph:
```



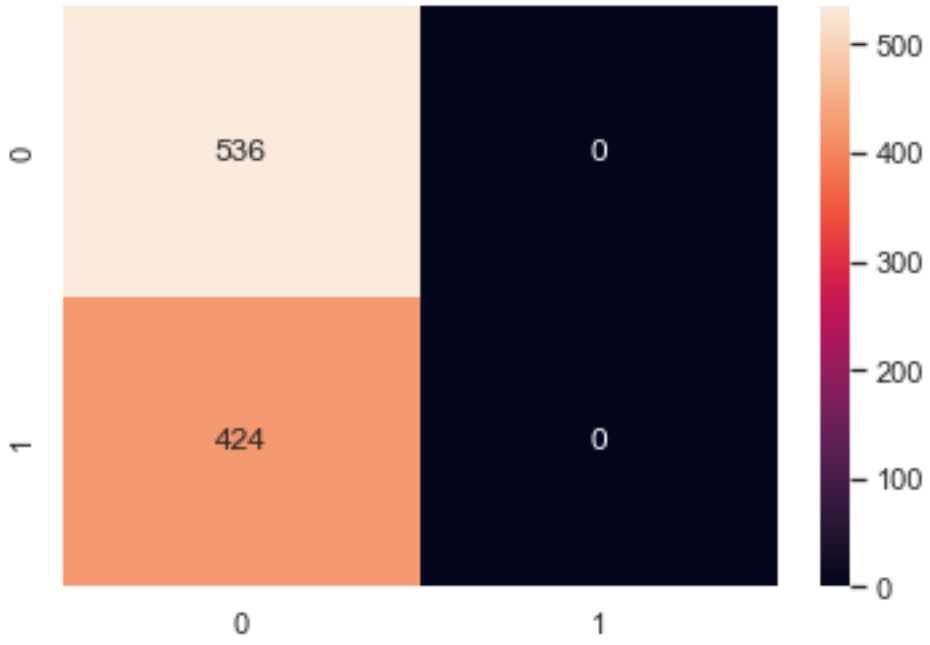
[44]:
```
print(f"The number of epochs taken to converge is: {Reg_NN_C1_Results[1]}")
```

```
The number of epochs taken to converge is: 89
```

11.1.1 4.1.1 Training Accuracy

[45]:
```
Reg_NN_C1_Training_Acc = Reg_NN_C1.predict(trainx,trainy)
```

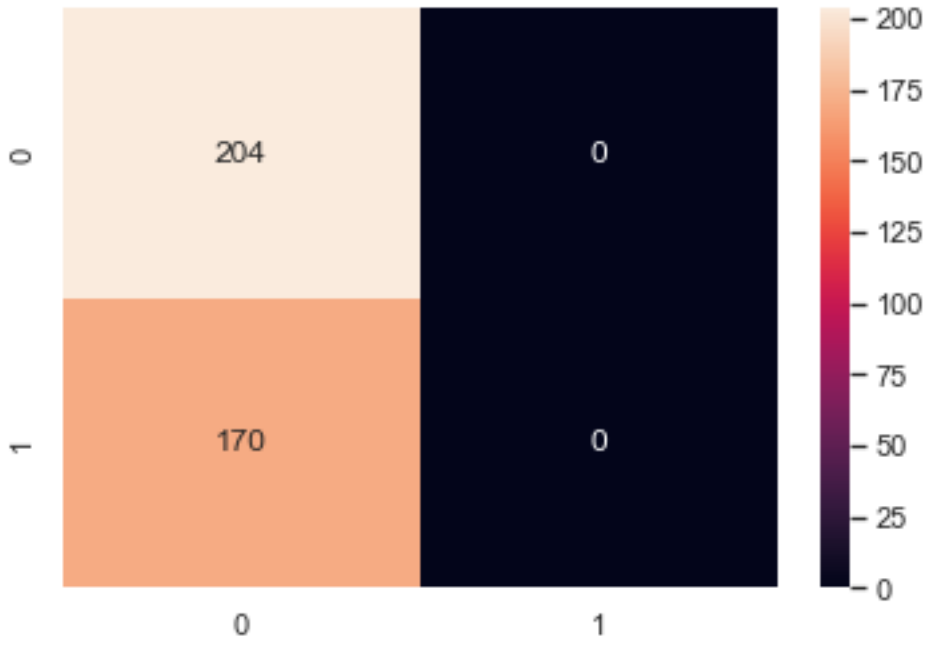
```
The accuracy is: 55.833333333333336
Confusion Matrix:
```



11.1.2 4.1.2 Validation Accuracy

[46]:
```
Reg_NN_C1_Validation_Acc = Reg_NN_C1.predict(validationx,validationy)
```

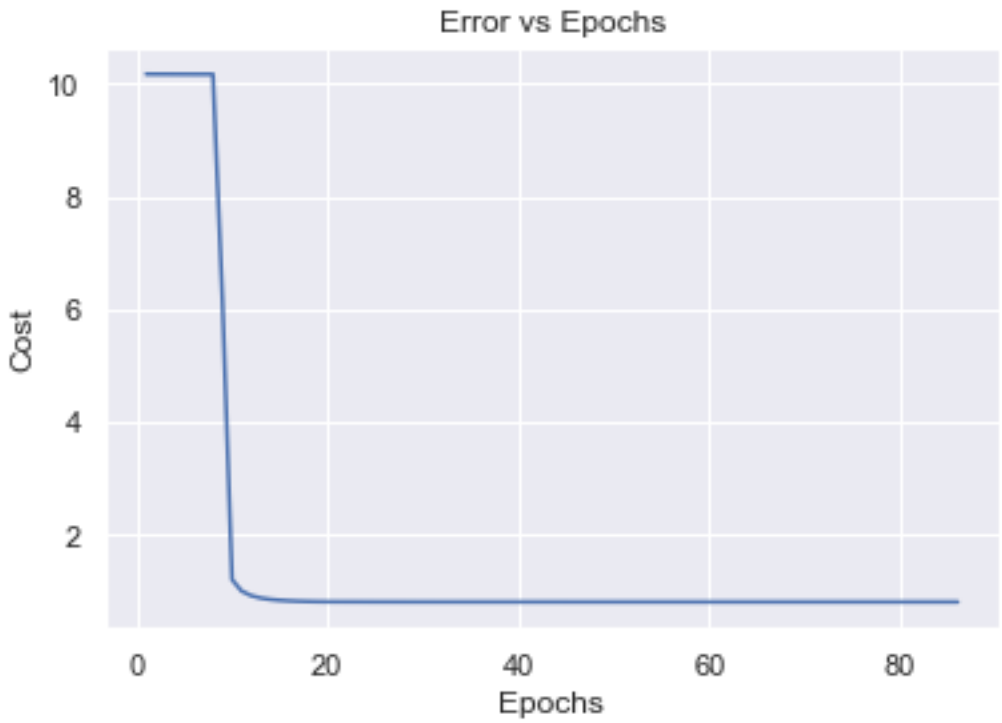
```
The accuracy is: 54.54545454545454
Confusion Matrix:
```



11.2 4.2 Setting Regularization = 2

[47]:
```
Reg_NN_C2 = NeuralNetwork()
print("The training error graph: ")
Reg_NN_C2_Results = Reg_NN_C2.fit(trainx,trainy,2,[5,5],1,["leaky relu","leaky␣
↪relu","leaky relu"],0.1,1,100)
```

```
The training error graph:
```



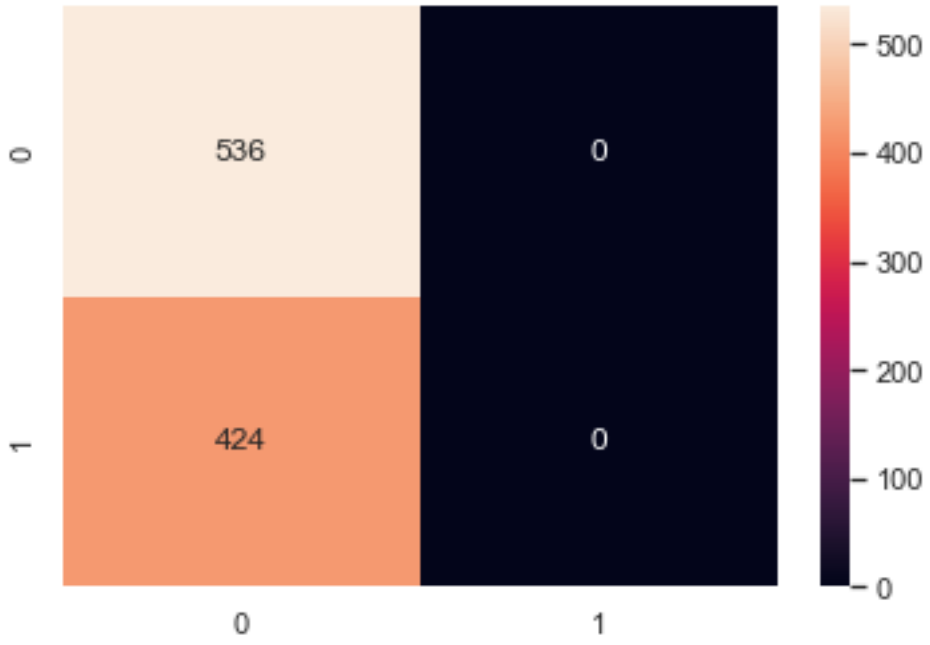
[48]:
```
print(f"The number of epochs taken to converge is: {Reg_NN_C2_Results[1]}")
```

```
The number of epochs taken to converge is: 86
```

11.2.1 4.2.1 Training Accuracy

[49]:
```
Reg_NN_C2_Training_Acc = Reg_NN_C2.predict(trainx,trainy)
```

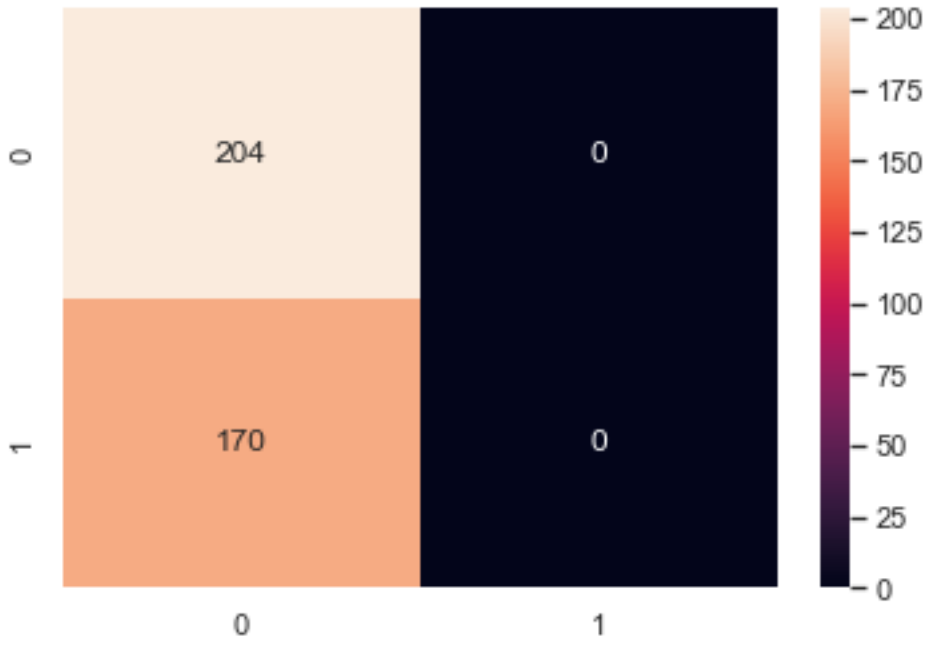
```
The accuracy is: 55.833333333333336
Confusion Matrix:
```



11.2.2 4.2.2 Validation Accuracy

[50]:
```
Reg_NN_C2_Validation_Acc = Reg_NN_C2.predict(validationx,validationy)
```

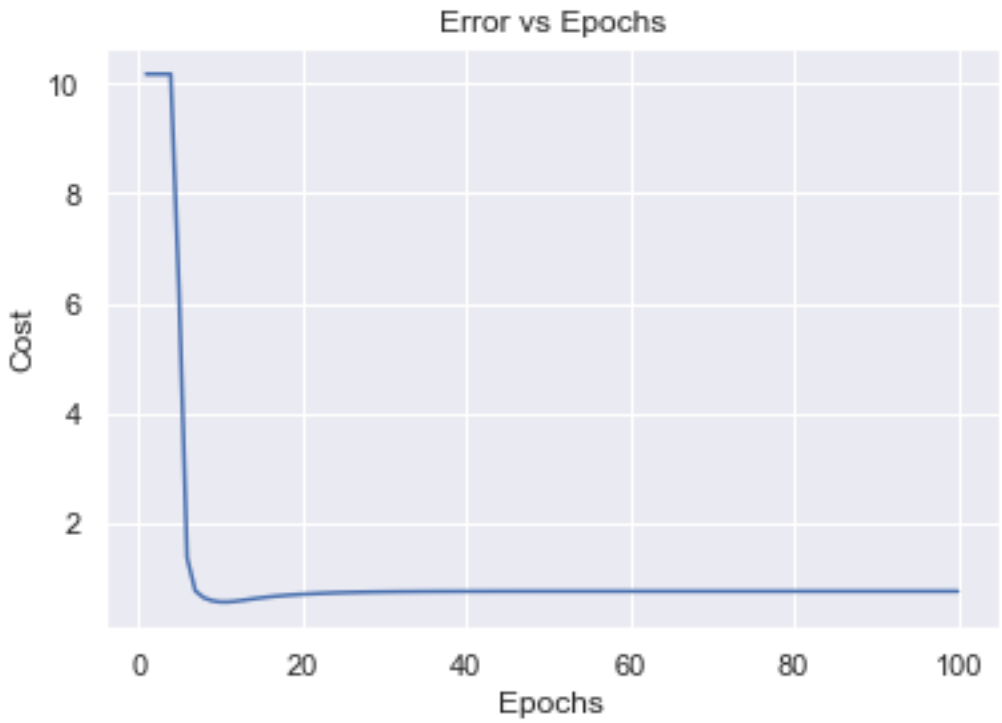
```
The accuracy is: 54.54545454545454
Confusion Matrix:
```



11.3 4.3 Setting Regularization = 0.75

[51]:
```
Reg_NN_C3 = NeuralNetwork()
print("The training error graph: ")
Reg_NN_C3_Results = Reg_NN_C3.fit(trainx,trainy,2,[5,5],1,["leaky relu","leaky␣
↪relu","leaky relu"],0.1,0.75,100)
```

```
The training error graph:
```



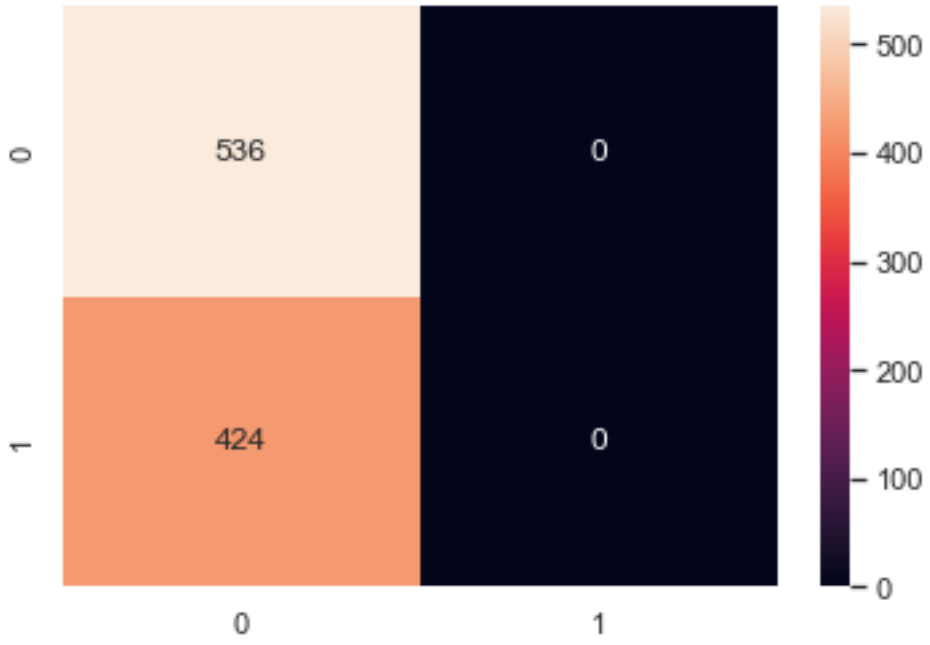
[52]:
```
print(f"The number of epochs taken to converge is: {Reg_NN_C3_Results[1]}")
```

```
The number of epochs taken to converge is: 100
```

11.3.1 4.3.1 Training Accuracy

[53]:
```
Reg_NN_C3_Training_Acc = Reg_NN_C3.predict(trainx,trainy)
```

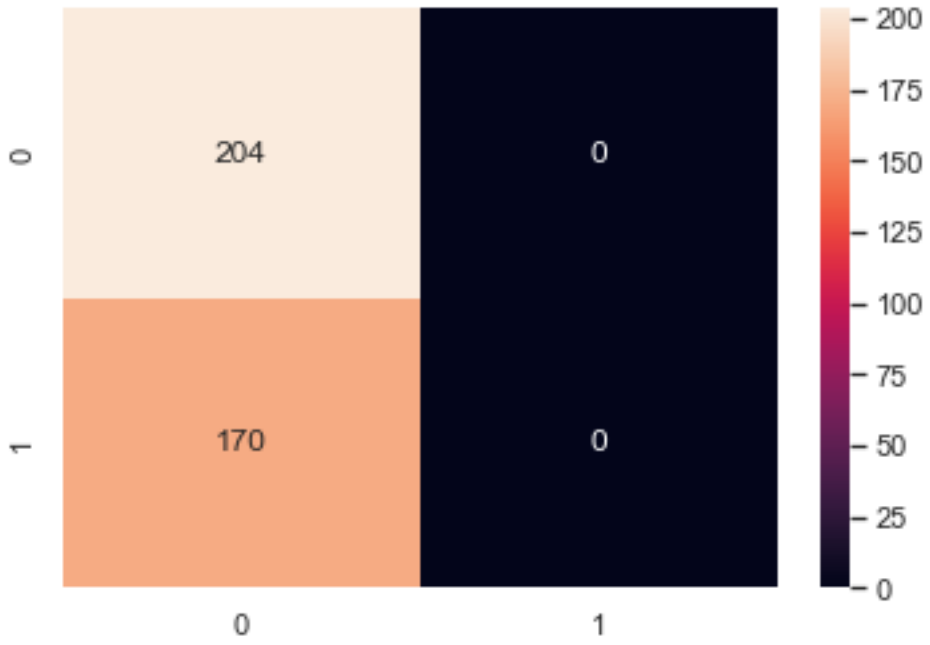
```
The accuracy is: 55.833333333333336
Confusion Matrix:
```



11.3.2 4.3.2 Validation Accuracy

[54]:
```
Reg_NN_C3_Validation_Acc = Reg_NN_C3.predict(validationx,validationy)
```

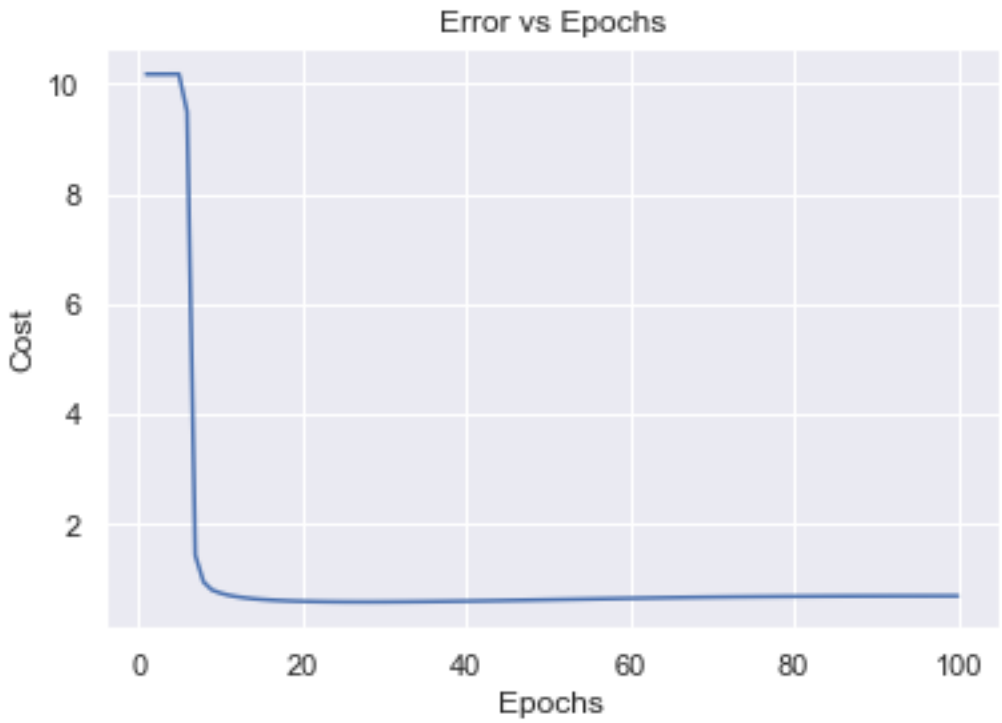
```
The accuracy is: 54.54545454545454
Confusion Matrix:
```



11.4 4.4 Setting Regularization = 0.25

[55]:
```
Reg_NN_C4 = NeuralNetwork()
print("The training error graph: ")
Reg_NN_C4_Results = Reg_NN_C4.fit(trainx,trainy,2,[5,5],1,["leaky relu","leaky␣
↪relu","leaky relu"],0.1,0.25,100)
```

```
The training error graph:
```



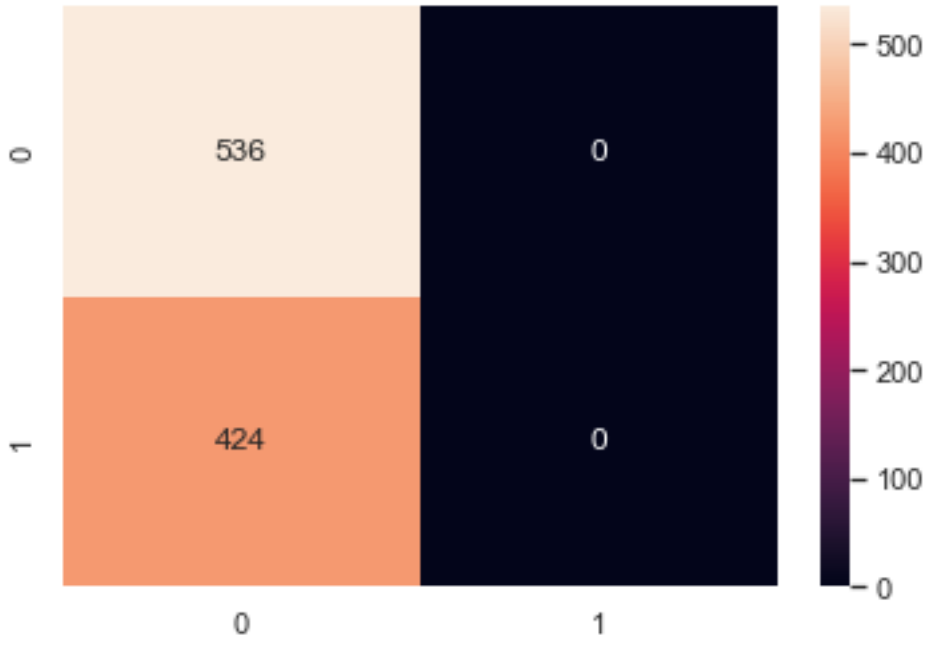
[56]:
```
print(f"The number of epochs taken to converge is: {Reg_NN_C4_Results[1]}")
```

```
The number of epochs taken to converge is: 100
```

11.4.1 4.4.1 Training Accuracy

[57]:
```
Reg_NN_C4_Training_Acc = Reg_NN_C4.predict(trainx,trainy)
```

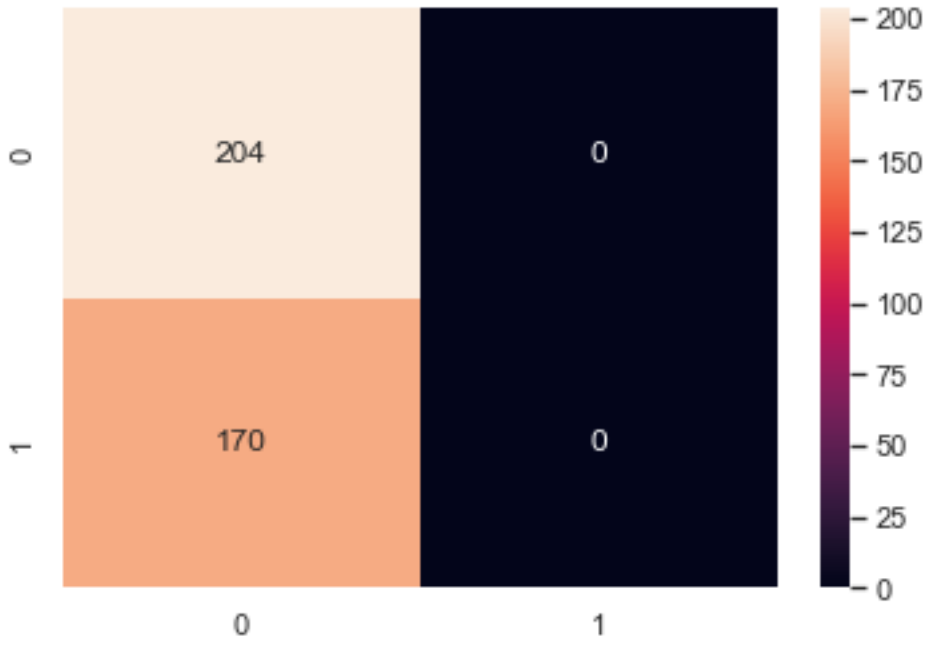
```
The accuracy is: 55.833333333333336
Confusion Matrix:
```



11.4.2 4.4.2 Validation Accuracy

[58]:
```
Reg_NN_C4_Validation_Acc = Reg_NN_C4.predict(validationx,validationy)
```

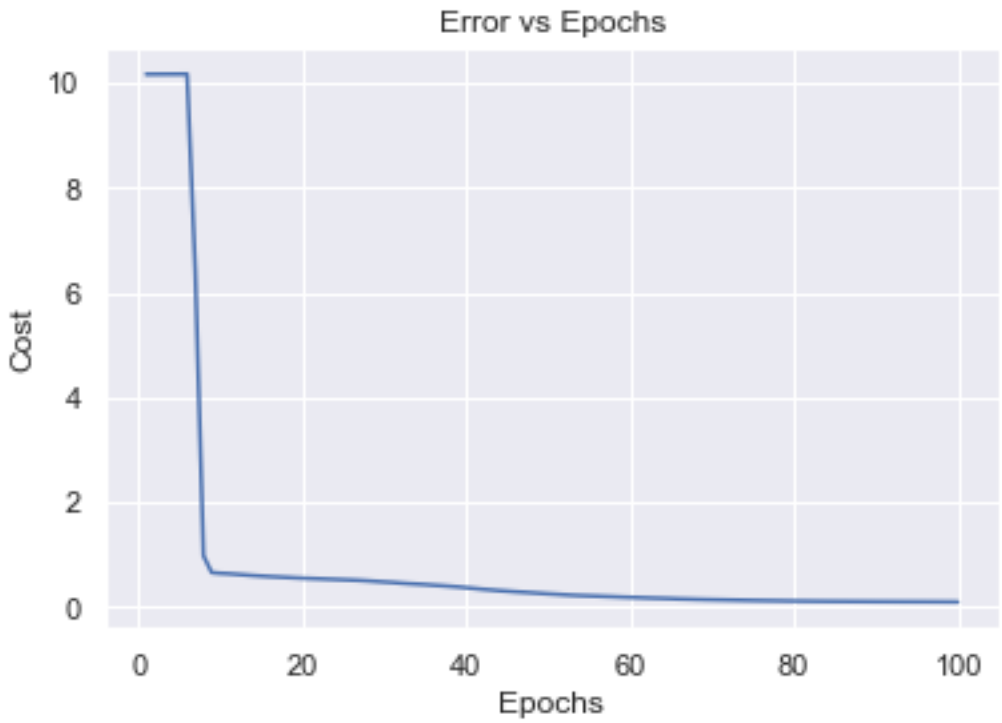
```
The accuracy is: 54.54545454545454
Confusion Matrix:
```



11.5 4.5 Setting Regularization = 0

[59]:
```
no_reg = NeuralNetwork()
print("The error graph:")
no_reg_results = no_reg.fit(trainx,trainy,2,[5,5],1,["leaky relu","leaky␣
↪relu","leaky relu"],0.1,0,100)
```

```
The error graph:
```



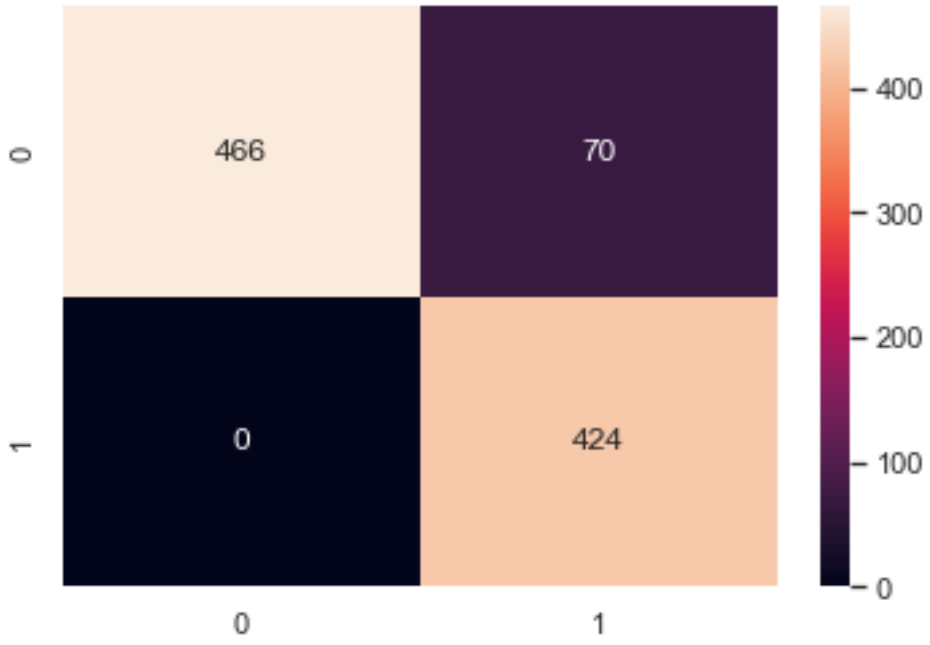
[60]:
```
print(f"The number of epochs taken to converge is: {no_reg_results[1]}")
```

```
The number of epochs taken to converge is: 100
```

11.5.1 4.5.1 Training Accuracy

[61]:
```
no_reg_train_acc = no_reg.predict(trainx,trainy)
```

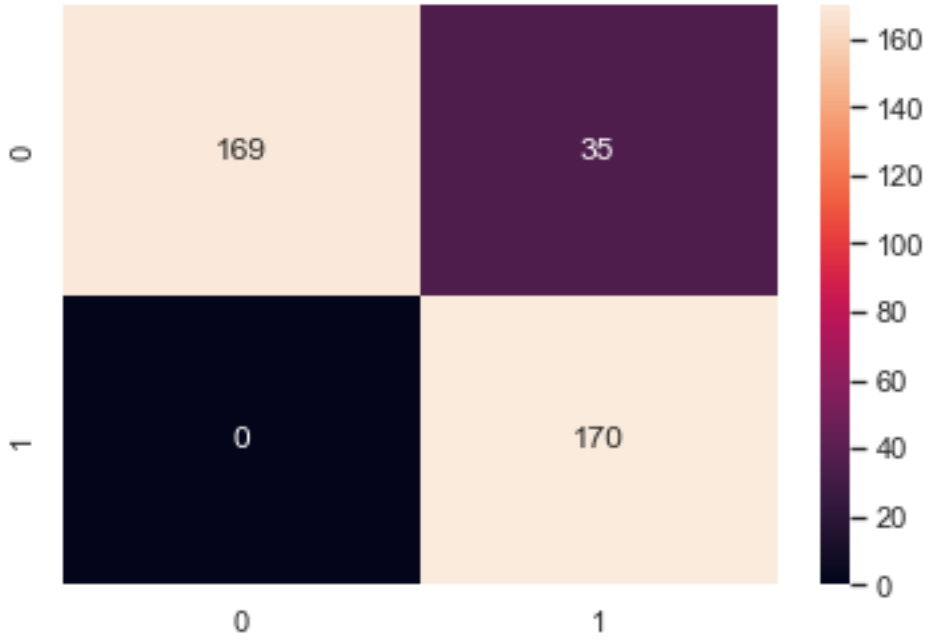
```
The accuracy is: 92.70833333333334
Confusion Matrix:
```



11.5.2 4.5.2 Validation Accuracy

[62]:
```
no_reg_valid_acc = no_reg.predict(validationx,validationy)
```

```
The accuracy is: 90.64171122994652
Confusion Matrix:
```



11.6 4.6 Tabularized Regularization Results

[63]:
```
reg_table = PrettyTable(["Regularization","Epochs Taken to Train","Training␣
↪Accuracy","Validation Accuracy"])

reg_table.add_row([2,Reg_NN_C2_Results[1],np.
↪round(Reg_NN_C2_Training_Acc*100,2),np.
↪round(Reg_NN_C2_Validation_Acc*100,2)])
reg_table.add_row([1,Reg_NN_C1_Results[1],np.
↪round(Reg_NN_C1_Training_Acc*100,2),np.
↪round(Reg_NN_C1_Validation_Acc*100,2)])
reg_table.add_row([0.75,Reg_NN_C3_Results[1],np.
↪round(Reg_NN_C3_Training_Acc*100,2),np.
↪round(Reg_NN_C3_Validation_Acc*100,2)])
reg_table.add_row([0.25,Reg_NN_C4_Results[1],np.
↪round(Reg_NN_C4_Training_Acc*100,2),np.
↪round(Reg_NN_C4_Validation_Acc*100,2)])
reg_table.add_row([0,no_reg_results[1],np.round(no_reg_train_acc*100,2),np.
↪round(no_reg_valid_acc*100,2)])
```

[64]:
```
print(reg_table)
```

```
+----------------+----------------------+------------------+-----------------
---+
| Regularization | Epochs Taken to Train | Training Accuracy | Validation
Accuracy |
+----------------+----------------------+------------------+-----------------
---+
|       2        |          86          |       55.83      |        54.55
|
|       1        |          89          |       55.83      |        54.55
|
|      0.75      |          100         |       55.83      |        54.55
|
|      0.25      |          100         |       55.83      |        54.55
|
|       0        |          100         |       92.71      |        90.64
|
+----------------+----------------------+------------------+-----------------
---+
```

11.7 4.7 Discussion of Regularization Results

- We can clearly see from the above table that as we increase the regularization the number of epochs taken to converge decreases but the validation accuracy is the same except when we

have no regularization.

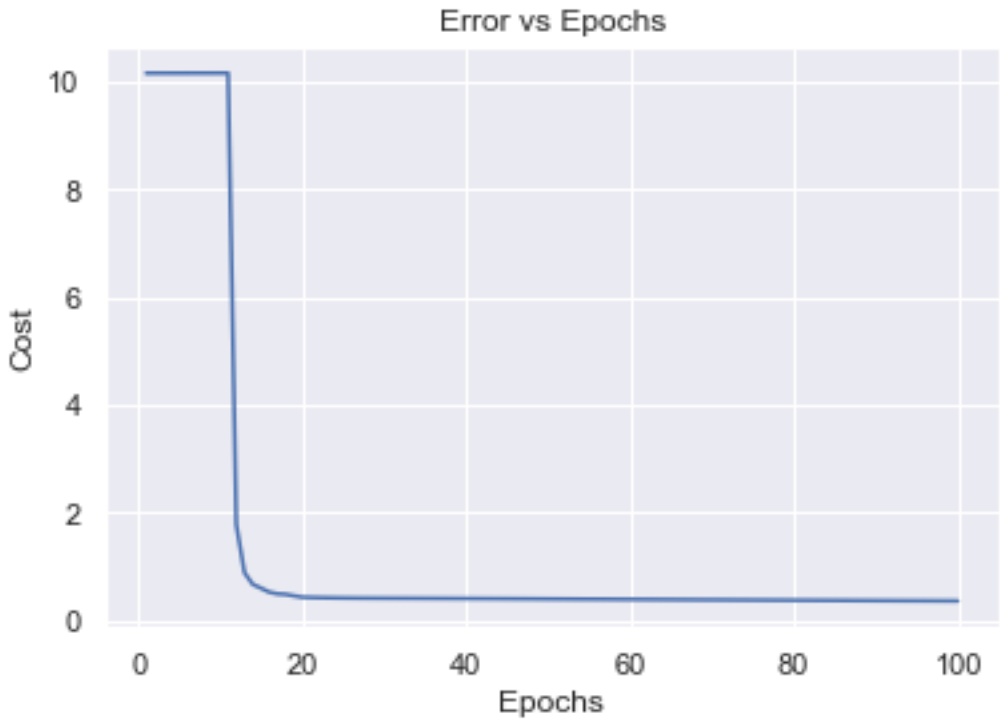
- Firstly the reason the epochs to train decreases as we increase regularization is because we penalise the weights more and conseuqently our consecutive weight updates are small. Since our convergence loop stops execution if the norm difference between consecutive weight updates is smaller than 0.00005, as a result when we implement larger regularization this criterion is met faster and the training stops.
- If we look at the training and validation accuracies, we see that they are the same for all regularization values that are not 0. If we look at all the error graphs, we see that in the inital epochs there is a drastic drop in the cost function, this is because the random weights undergo a massive initial adjustment. However, after the initial adjustment, the updates are extremely small and almost negligible in affecting the predictive power, hence we end up with models that have a high bias and underfit the data.
- If we also look at the confusion matrix, we see that due to the penalization of the weights, we are only able to learn to predict a single class and this leads to the poor performance.

12 5. Investigating The Effect Of The Learning Rate

12.1 5.1 Setting Learning Rate = 0.2

[65]:
```
NN_LR1 = NeuralNetwork()
print("The training error graph: ")
NN_LR1_Results = NN_LR1.fit(trainx,trainy,2,[5,5],1,["leaky relu","leaky␣
↪relu","leaky relu"],0.1,0,100)
```

```
The training error graph:
```



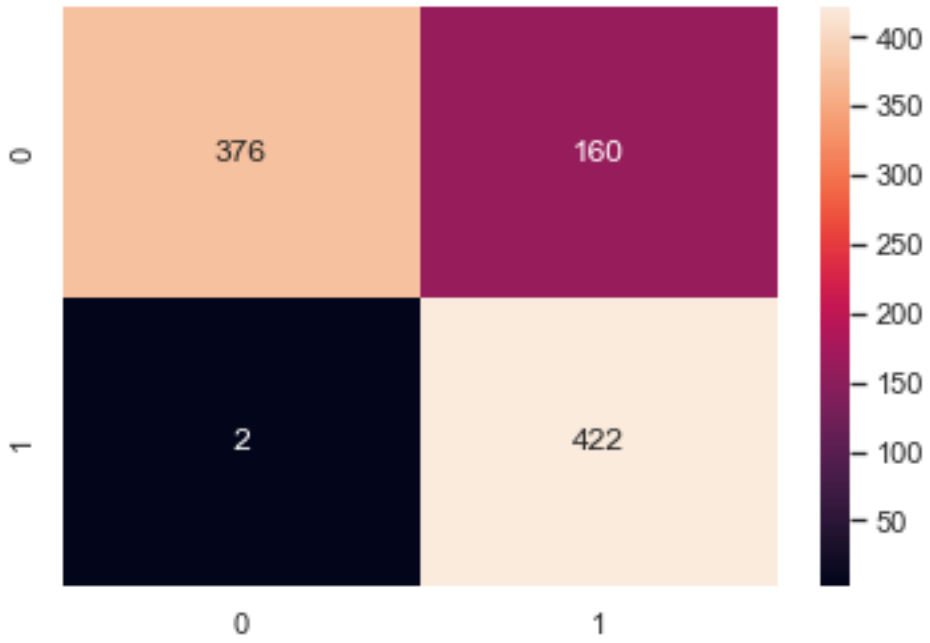
[66]:
```
print(f"The number of epochs taken to converge is: {NN_LR1_Results[1]}")
```

```
The number of epochs taken to converge is: 100
```

12.1.1 5.1.1 Training Accuracy

[67]:
```
NN_LR1_Train_Acc = NN_LR1.predict(trainx,trainy)
```

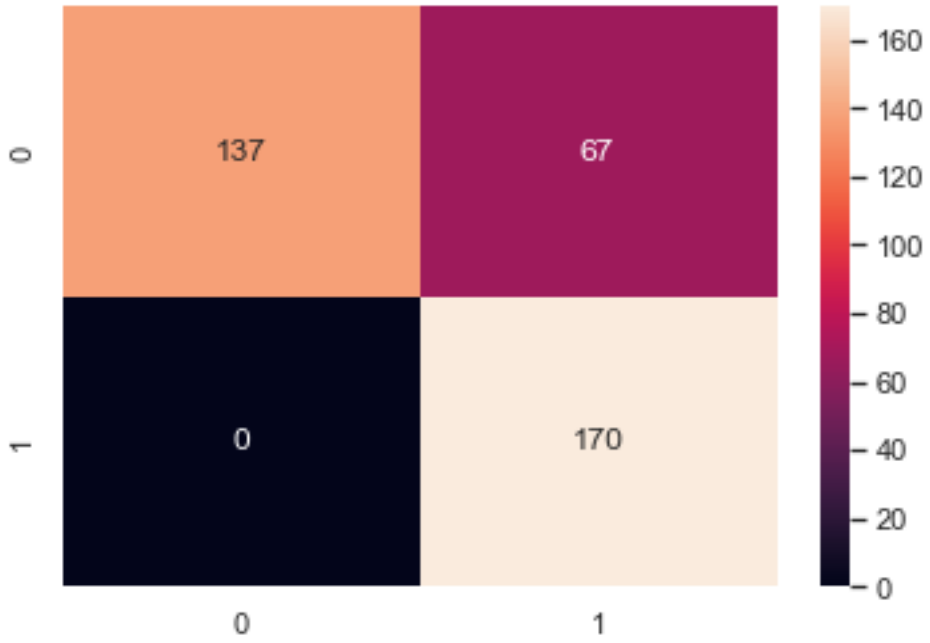
```
The accuracy is: 83.125
Confusion Matrix:
```



12.1.2 5.1.2 Validation Accuracy

[68]:
```
NN_LR1_Valid_Acc = NN_LR1.predict(validationx,validationy)
```

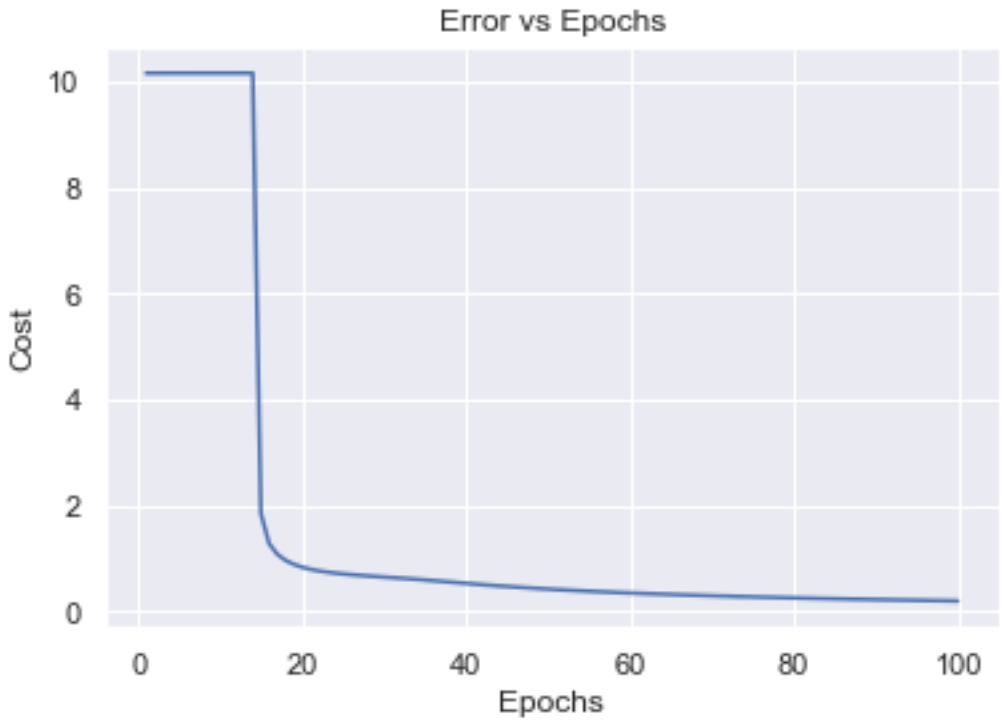
```
The accuracy is: 82.0855614973262
Confusion Matrix:
```



12.2 5.2 Setting Learning Rate = 0.1

[69]:
```
NN_LR2 = NeuralNetwork()
print("The training error graph: ")
NN_LR2_Results = NN_LR2.fit(trainx,trainy,2,[5,5],1,["leaky relu","leaky␣
↪relu","leaky relu"],0.1,0,100)
```

```
The training error graph:
```



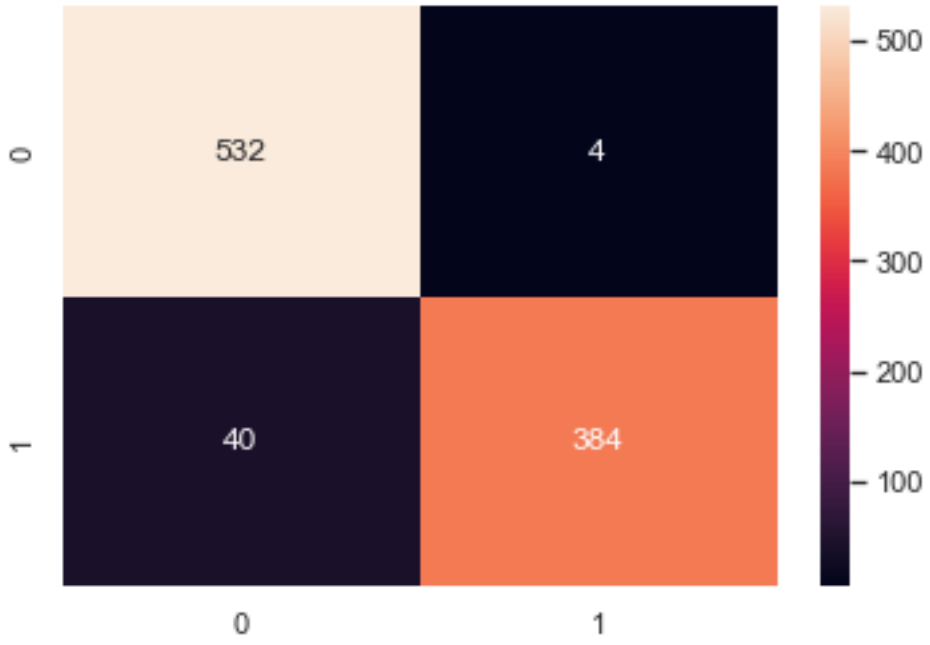
[70]:
```
print(f"The number of epochs taken to converge is: {NN_LR2_Results[1]}")
```

```
The number of epochs taken to converge is: 100
```

12.2.1 5.2.1 Training Accuracy

[71]:
```
NN_LR2_Train_Acc = NN_LR2.predict(trainx,trainy)
```

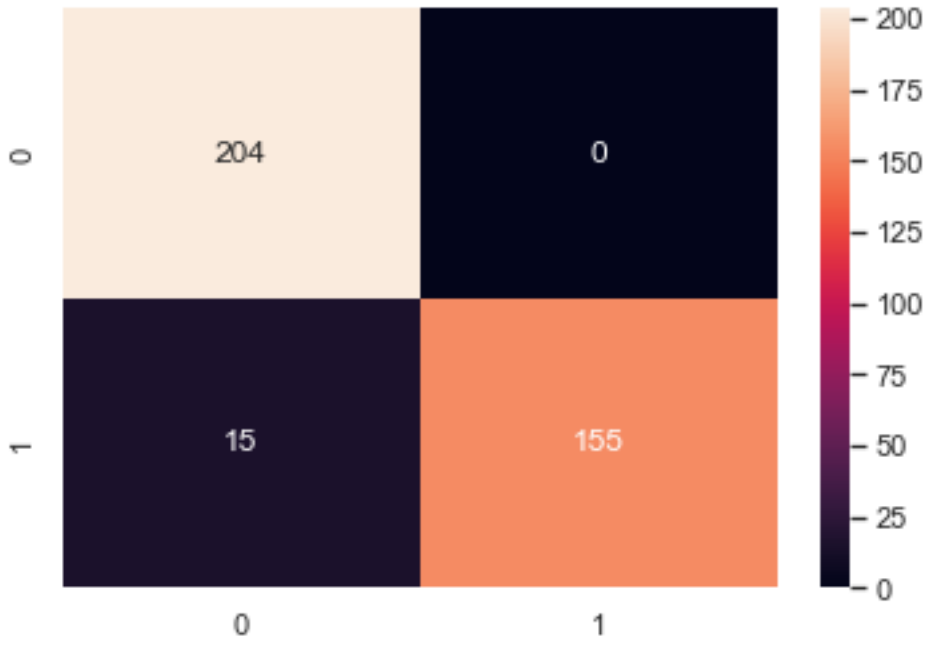
```
The accuracy is: 95.41666666666667
Confusion Matrix:
```



12.2.2 5.2.2 Validation Accuracy

[72]:
```
NN_LR2_Valid_Acc = NN_LR2.predict(validationx,validationy)
```

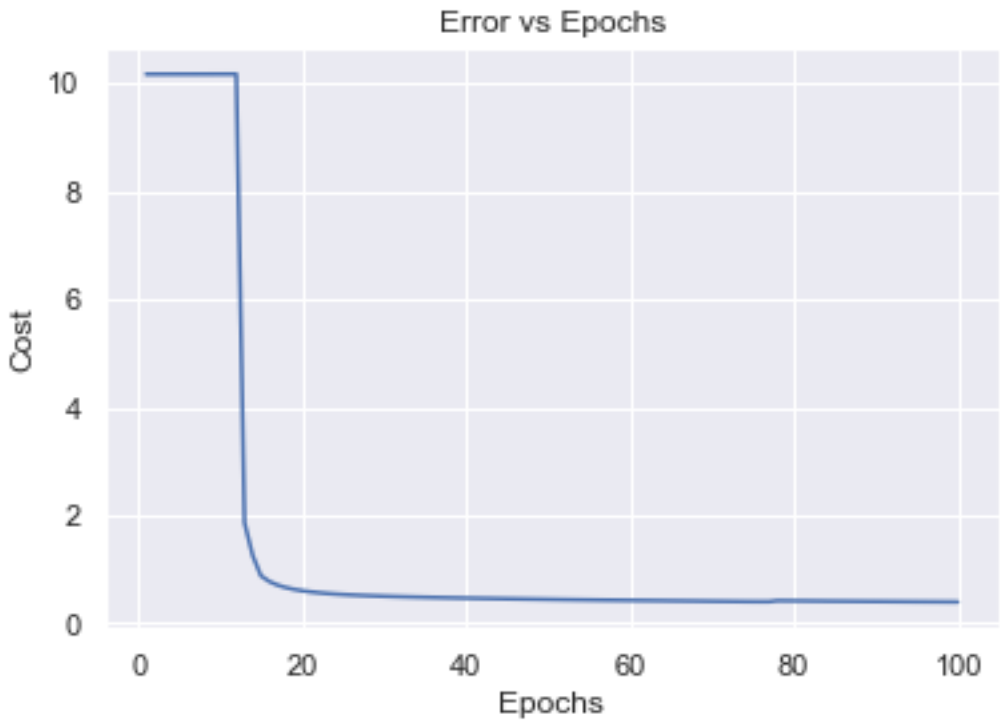
```
The accuracy is: 95.98930481283422
Confusion Matrix:
```



12.3 5.3 Setting Learning Rate = 0.01

[73]:
```
NN_LR3 = NeuralNetwork()
print("The training error graph: ")
NN_LR3_Results = NN_LR3.fit(trainx,trainy,2,[5,5],1,["leaky relu","leaky␣
↪relu","leaky relu"],0.1,0,100)
```

```
The training error graph:
```



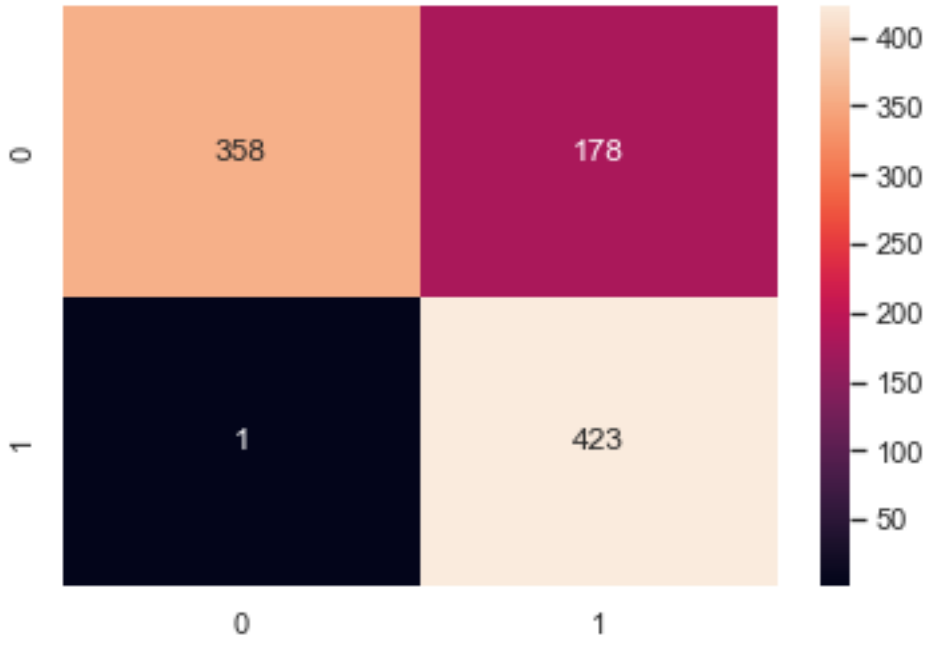
[74]:
```
print(f"The number of epochs taken to converge is: {NN_LR3_Results[1]}")
```

```
The number of epochs taken to converge is: 100
```

12.3.1 5.3.1 Training Accuracy

[75]:
```
NN_LR3_Train_Acc = NN_LR3.predict(trainx,trainy)
```

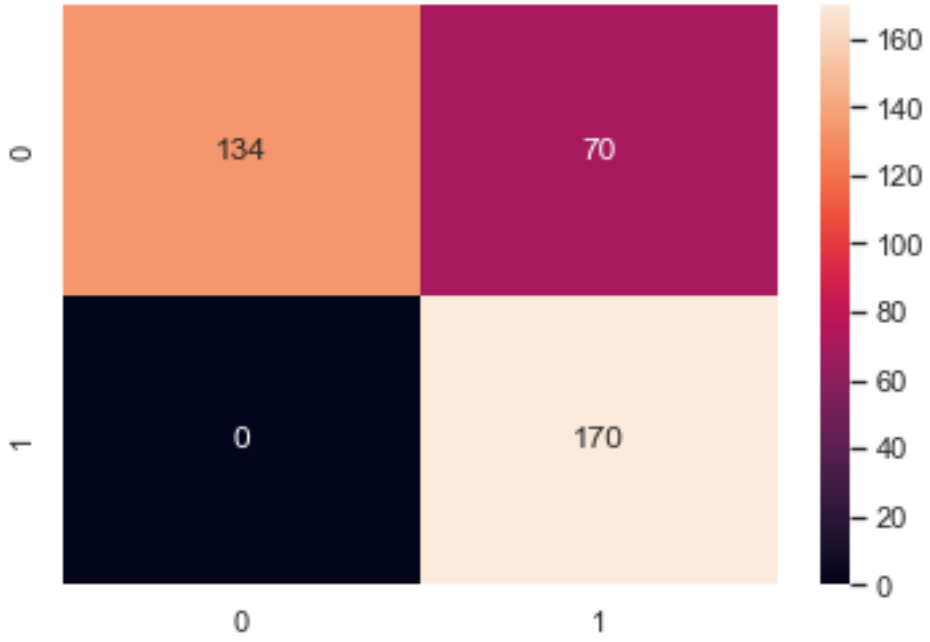
```
The accuracy is: 81.35416666666667
Confusion Matrix:
```



12.3.2 5.3.2 Validation Accuracy

[76]:
```
NN_LR3_Valid_Acc = NN_LR3.predict(validationx,validationy)
```

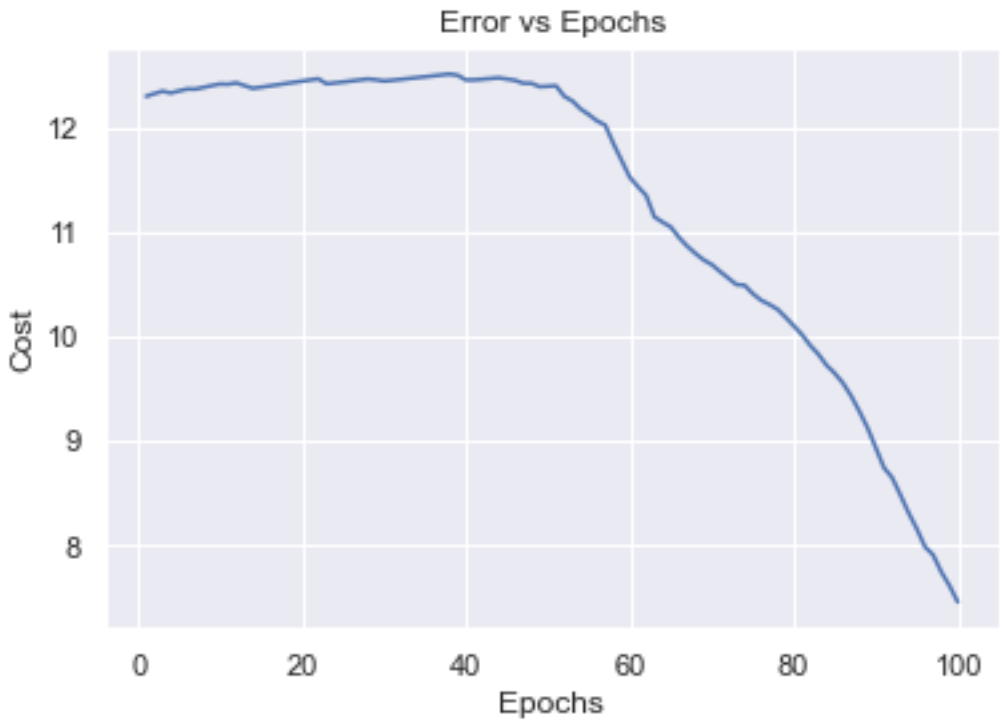
```
The accuracy is: 81.28342245989305
Confusion Matrix:
```



12.4 5.4 Setting Learning Rate = 0.001

[77]:
```
NN_LR4 = NeuralNetwork()
print("The training error graph: ")
NN_LR4_Results = NN_LR4.fit(trainx,trainy,2,[5,5],1,["leaky relu","leaky␣
↪relu","leaky relu"],0.001,0,100)
```

```
The training error graph:
```



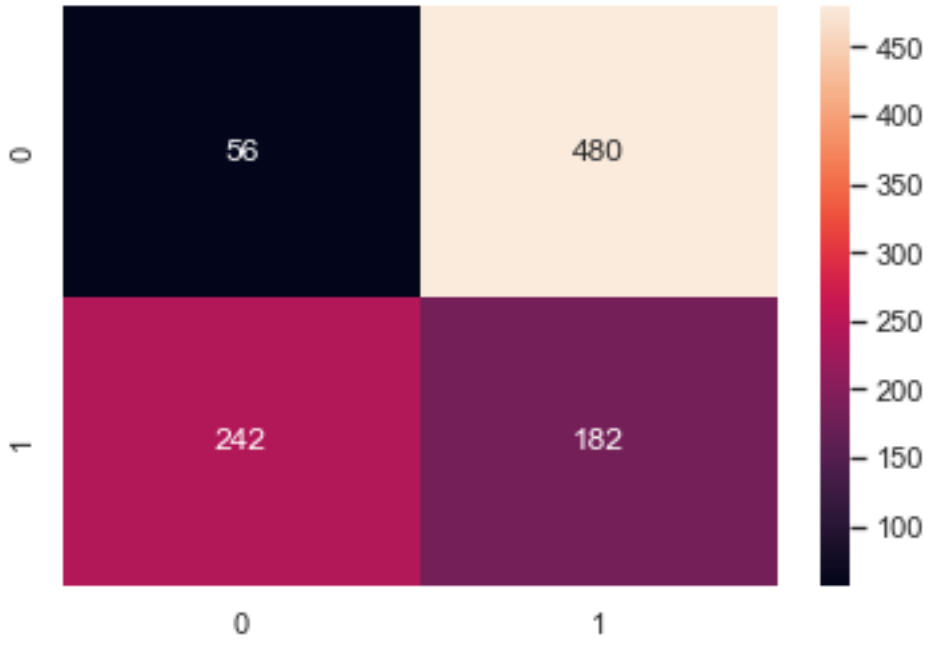
[78]:
```
print(f"The number of epochs taken to converge is: {NN_LR4_Results[1]}")
```

```
The number of epochs taken to converge is: 100
```

12.4.1 5.4.1 Training Accuracy

[79]:
```
NN_LR4_Train_Acc = NN_LR4.predict(trainx,trainy)
```

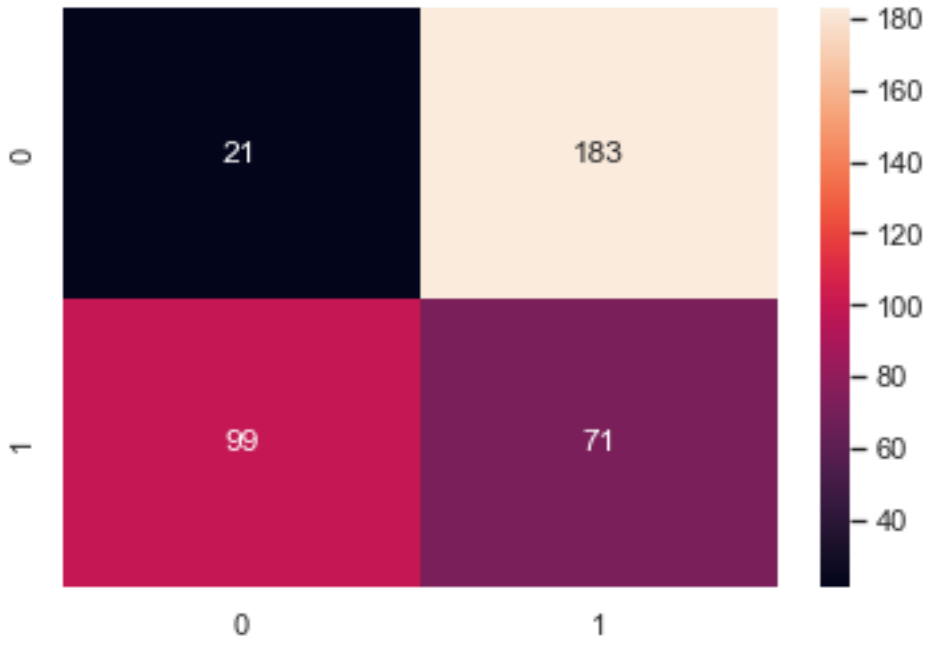
```
The accuracy is: 24.791666666666668
Confusion Matrix:
```



12.4.2 5.4.2 Validation Accuracy

[80]:
```
NN_LR4_Valid_Acc = NN_LR4.predict(validationx,validationy)
```

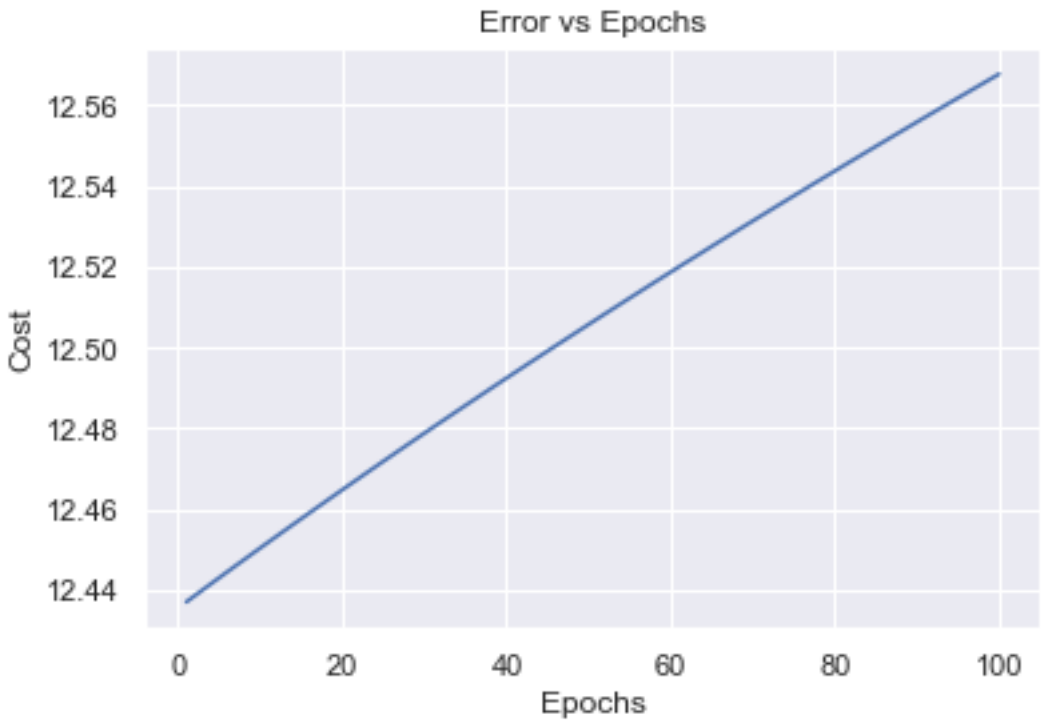
```
The accuracy is: 24.598930481283425
Confusion Matrix:
```



12.5 5.5 Setting Learning Rate = 0.0001

[81]:
```
NN_LR5 = NeuralNetwork()
print("The training error graph: ")
NN_LR5_Results = NN_LR5.fit(trainx,trainy,2,[5,5],1,["leaky relu","leaky␣
↪relu","leaky relu"],0.0001,0,100)
```

```
The training error graph:
```



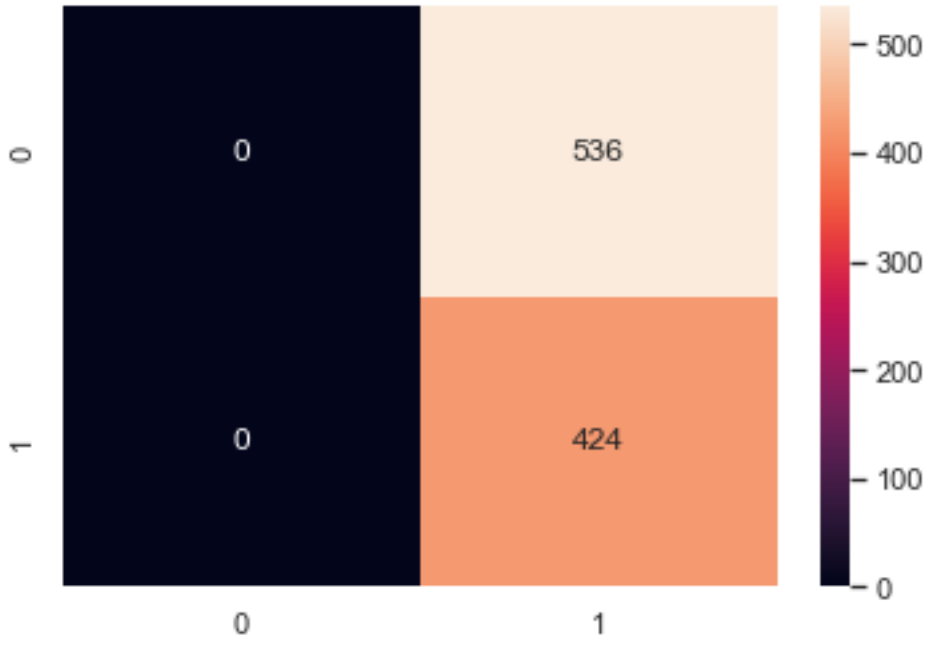
[82]:
```
print(f"The number of epochs taken to converge is: {NN_LR5_Results[1]}")
```

```
The number of epochs taken to converge is: 100
```

12.5.1 5.5.1 Training Accuracy

[83]:
```
NN_LR5_Train_Acc = NN_LR5.predict(trainx,trainy)
```

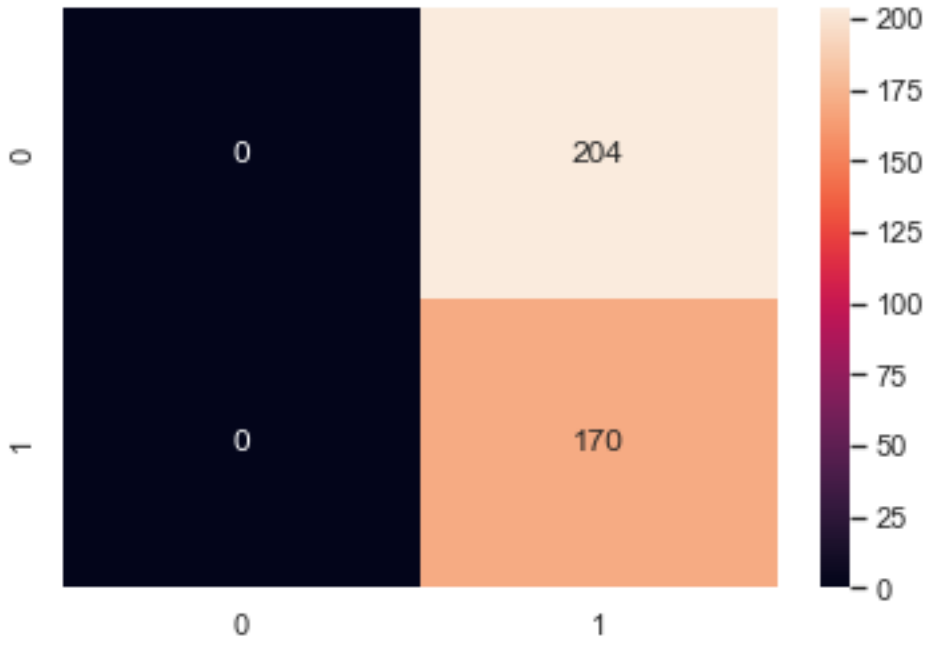
```
The accuracy is: 44.166666666666664
Confusion Matrix:
```



12.5.2 5.5.2 Validation Accuracy

[84]:
```
NN_LR5_Valid_Acc = NN_LR5.predict(validationx,validationy)
```

```
The accuracy is: 45.45454545454545
Confusion Matrix:
```



12.6 5.6 Tabularized Results

[85]:
```
lr_table = PrettyTable(["Learning Rate","Epochs Taken to Train","Training␣
↪Accuracy","Validation Accuracy"])

lr_table.add_row([0.2,NN_LR1_Results[1],np.round(NN_LR1_Train_Acc*100,2),np.
↪round(NN_LR1_Valid_Acc*100,2)])
lr_table.add_row([0.1,NN_LR2_Results[1],np.round(NN_LR2_Train_Acc*100,2),np.
↪round(NN_LR2_Valid_Acc*100,2)])
lr_table.add_row([0.01,NN_LR3_Results[1],np.round(NN_LR3_Train_Acc*100,2),np.
↪round(NN_LR3_Valid_Acc*100,2)])
lr_table.add_row([0.001,NN_LR4_Results[1],np.round(NN_LR4_Train_Acc*100,2),np.
↪round(NN_LR4_Valid_Acc*100,2)])
lr_table.add_row([0.0001,NN_LR5_Results[1],np.round(NN_LR5_Train_Acc*100,2),np.
↪round(NN_LR5_Valid_Acc*100,2)])
```

[86]:
```
print(lr_table)
```

```
+---------------+----------------------+------------------+------------------
--+
| Learning Rate | Epochs Taken to Train | Training Accuracy | Validation
Accuracy |
+---------------+----------------------+------------------+------------------
--+
|      0.2      |          100         |       83.12      |        82.09
|
|      0.1      |          100         |       95.42      |        95.99
|
|      0.01     |          100         |       81.35      |        81.28
|
|     0.001     |          100         |       24.79      |         24.6
|
|     0.0001    |          100         |       44.17      |        45.45
|
+---------------+----------------------+------------------+------------------
--+
```

12.7 5.7 Discussion Of Learning Rate Results

- We can clearly see that the optimal value for the learning rate is 0.1.
- If we look at learning rates larger, the model overshoots and diverges. This explains why smaller learning rates perform better. Also the reasoning behind these larger learning rates still giving a relatively good accuracy is because we reach the maximum number of epochs prior to the weights being adjusted so much that they end up providing completely incorrect predictions.

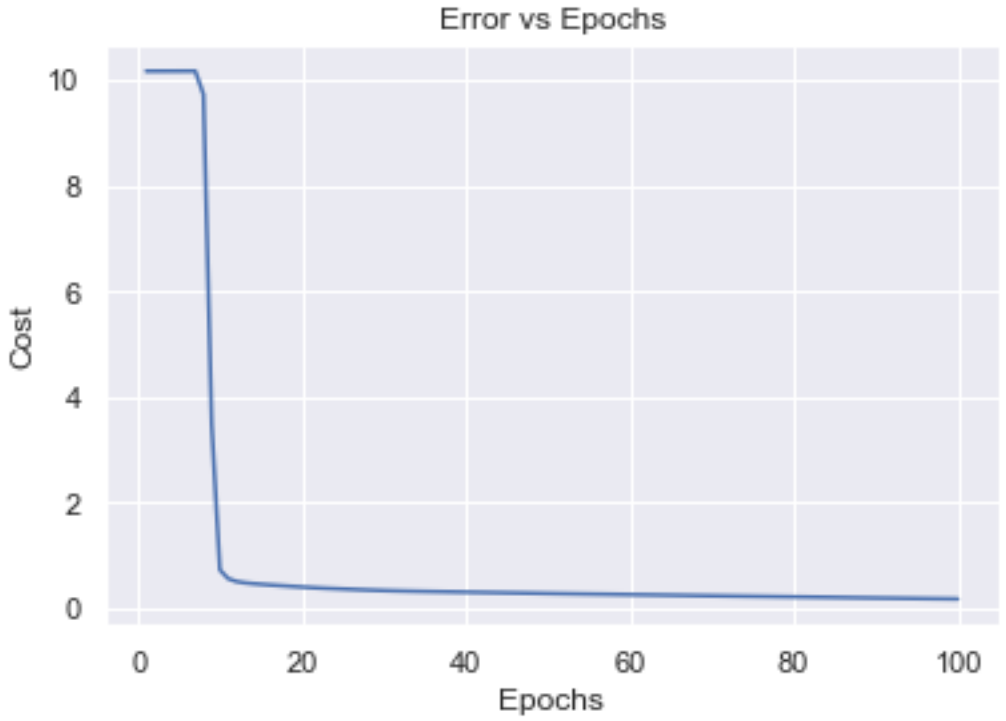
- We also see that as we decrease the learning rate, the accuracy decreases. This is simply because we are taking smaller steps in the direction of the minima, and consequently we reach the maximum epochs before we converge. Thus we end up underfitting and with a high bias model.

13 5. Network Built With Optimal Parameters

- Using the optimal activation function Leaky ReLu.
- Using the optimal architecture of two hidden layers with 5 nodes each.
- Using the optimal learning rate of 0.1.
- Not using regularization since this provided optimal results.

[102]:
```
Final_NN = NeuralNetwork()
print("The training error graph: ")
Final_NN_Results = Final_NN.fit(trainx,trainy,2,[5,5],1,["leaky relu","leaky␣
↪relu","leaky relu"],0.1,0,100)
```

```
The training error graph:
```



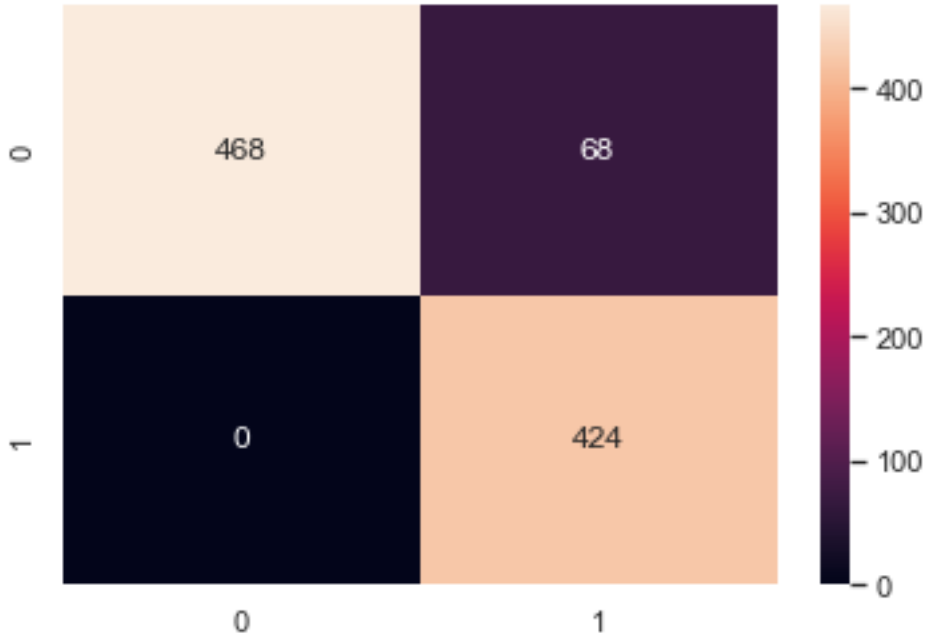
[103]:
```
print(f"The number of epochs taken to converge is: {Final_NN_Results[1]}")
```

```
The number of epochs taken to converge is: 100
```

13.0.1 5.1 Training Accuracy

[104]:
```
Final_NN_Training_Acc = Final_NN.predict(trainx,trainy)
```

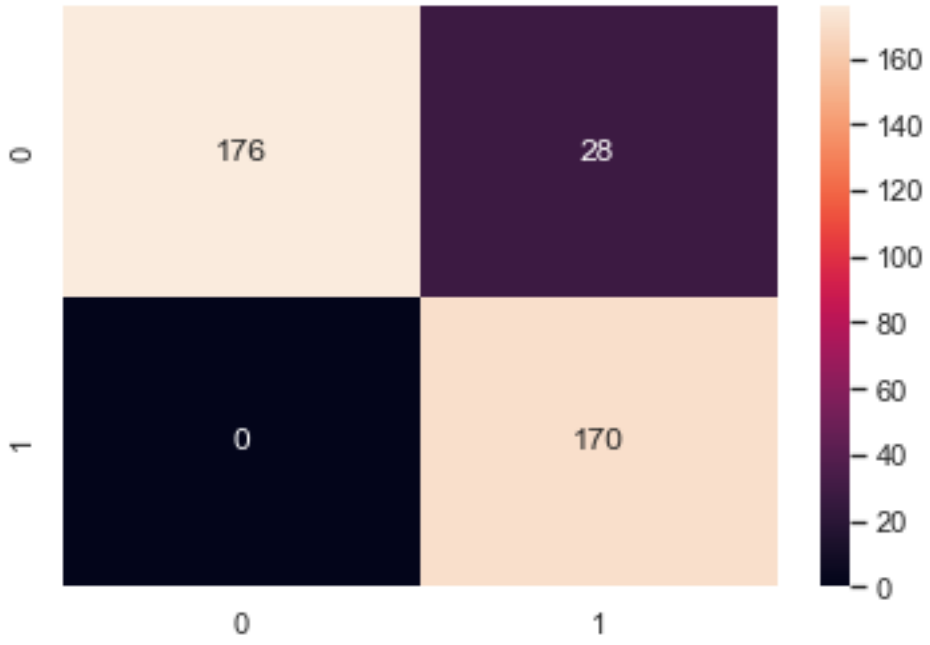
```
The accuracy is: 92.91666666666667
Confusion Matrix:
```



13.0.2 5.2 Validation Accuracy

[105]:
```
Final_NN_Validation_Acc = Final_NN.predict(validationx,validationy)
```

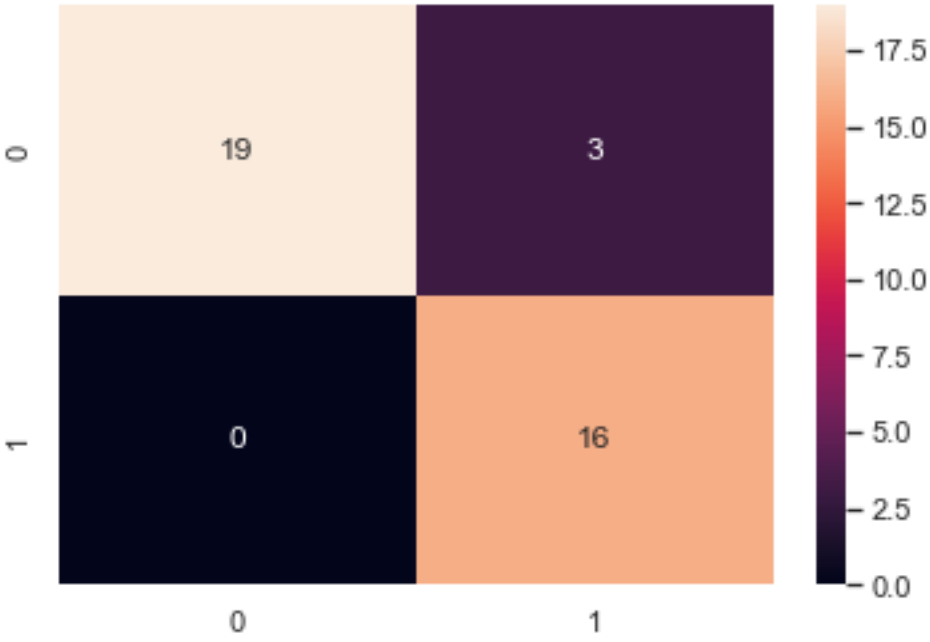
```
The accuracy is: 92.51336898395722
Confusion Matrix:
```



13.0.3 5.3 Testing Accuracy

[106]:
```
Final_NN_Testing_Acc = Final_NN.predict(testx,testy)
```

```
The accuracy is: 92.10526315789474
Confusion Matrix:
```



We can see that the model achieves an 92% test accuracy which is a good score and implies the model has not overfit since it is able to generalize on the test dataset. Consequently we can say that we have a low bias and low variance model.