1. Please submit a link to your GitHub repository for your class.

   https://github.com/gshawm/CS450/tree/master/Neural

2. Describe your overall approach to implementing the algorithm in code. How are your classes/data structures organized? How do you keep track of the necessary pieces for back-propagation.

   I designed a NeuralNetwork class that held all my Nodes. The Node is a class as well. Each Node holds inputs and the weights associated with each input. With this, there also existed a static bias input that became an input as well. Each Node also had an error, target (for output nodes), output, and a threshold. However, the threshold was only useful for when implementing a non-Sigmoid activation function. Each node knew how to compute their errors (output and hidden), compute their activation function, and how to update their weights. It was up to the NeuralNetwork to do these in the right order. It should be noted that there existed a static learning rate for all nodes as well.

   The NeuralNetwork, as said, holds all Nodes, number of iterations for training, and keeps track of the accuracies of each iteration. The training method performs a feed forward on the training set. Each time there is a bad prediction. It then calls the back-propagation method, after setting the targets to what they were supposed to be. Using those targets, the back propagate will loop through each layer and compute the error of each node. For the hidden layers, I simply gather the errors and the weights from the Nodes on the next layer. Example: nodes is a list which had a shape of (num_layers, num_nodes_per_layer) or [[Node, Node, Node, Node], [Node, Node, Node]]. So if we are on the i'th layer and the j'th node, I get the errors and weights associated with that node as so:

```
weights = []
errors = []
for k in range(self.nodes[i+1].__len()):
        errors.append(self.nodes[i+1][k].error)
        weights.append(self.nodes[i+1][k].weights[j])
```

   To update the weights. I had perform a similar operation:

```
prev_outputs = []
# This checks for non first layer. The first layer is where the
# inputs are actual inputs.
if i != 0:
        # The first one is the bias and is found in the inputs of the node.
        prev_outputs.append(self.nodes[i][j].inputs[0])
        for k in range(self.nodes[i-1].__len__()):
                prev_outputs.append(self.nodes[i-1][k].output)
# First layer check
else:
        for k in range(self.nodes[i][j].inputs.__len__()):
                prev_outputs.append(self.nodes[i][j].inputs[k])
```

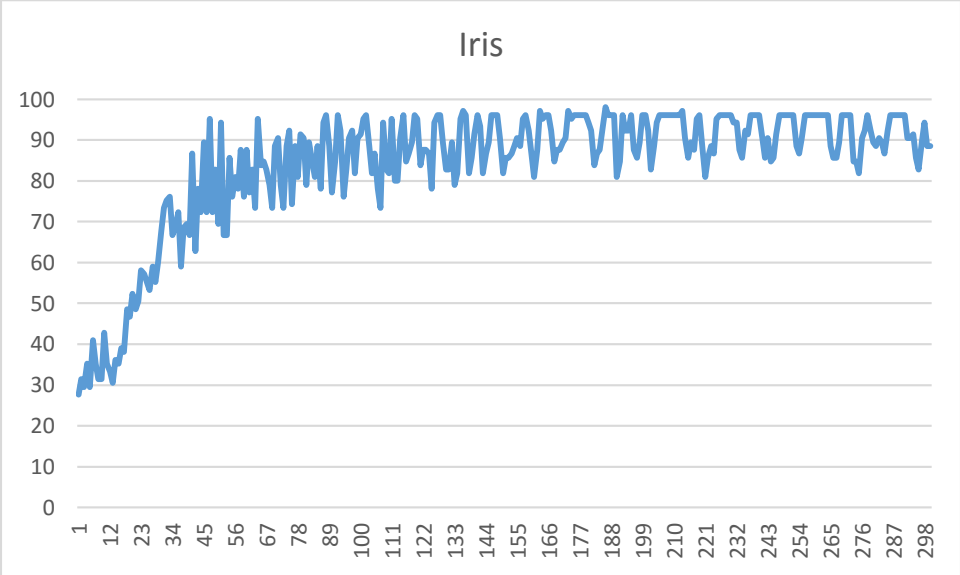
   This process allows me to keep track of the information for the back-propagation but I don't need to change much at all, since it is all stored in the Nodes! It should be

noted that the NeuralNetwork also has methods for creating a neural network and each layer, setting the inputs of each layer (based on the outputs of previous layers/inputs), calculating the totals for each node's activation function and giving a prediction based on the last layer's outputs. It can also display each node and their associated weights.

3. Describe the part of the assignment that gave you the most trouble, and how you overcame it.

   The part that is STILL giving me the most trouble is figuring out the best combination of nodes and layers to use. This is especially true for the diabetes dataset. However, I know that my system is doing the best it can because I'm getting similar results from the existing implementation. As the graph below shows, my system seemed to get around 50-60% during training and gave similar results in the testing phase.

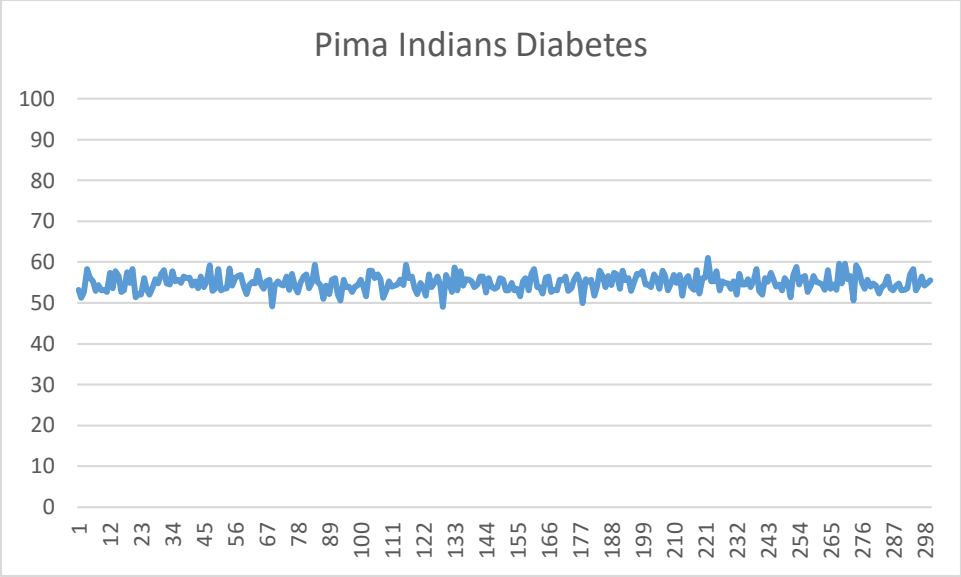
4. Produce at least one graph to show the training progress for the Iris dataset.



5. Compare your results on the Iris dataset to those of an existing implementation.

   After running it through the training phase, my implementation would usually get between 85-100% accuracy. The existing implementation did about the same. I am very satisfied with these results! I found that the deeper the network was, the worse it seemed to do. This wasn't always true, but essentially deepness didn't help my network. In fact just one hidden layer of 4 nodes seemed to do the trick.

6. Produce at least one graph to show the training progress for the Diabetes dataset.



7. Compare your results on the Diabetes dataset to those of an existing implementation.

   Interestingly enough, almost everytime, I got the exact same accuracy as the existing implementation. I found this interesting as my accuracy wasn't very good (about 50-70%). Sometimes mine would do way better and sometimes theirs would do way better, but overall the results were VERY similar if not the same.

8. Describe any efforts you made to go above and beyond.

   I didn't do anything above and beyond.

9. Please state which category you feel best describes your assignment and give a 1-2 sentence justification for your choice: A) Some attempt was made, B) Developing, but significantly deficient, C) Slightly deficient, but still mostly adequate, D) Meets requirements, E) Shows creativity and excels above and beyond requirements.

D) I did everything that was required of me, but I didn't do anything extra. My implementation, I felt was well done as well and got the job done.