B455Project1

August 19, 2021

1 B455 Project 1 - Designing a Multi-layer Perceptron

By Owen Gordon The goal of this project was to design a multi layer perceptron to classify types of wine. The attributes for the given inputs were: 1) Alcohol, 2) Malic acid, 3) Ash, 4) Alcalinity of ash, 5) Magnesium, 6) Total phenols, 7) Flavanoids, 8) Nonflavanoid phenols, 9) Proanthocyanins, 10) Color intensity, 11) Hue, 12) OD280/OD315 of diluted wines, and 13) Proline.

To begin the data must be read in, and then the data needs to be separated into the input and the targets. I used a Pandas dataframe to read the data and I used numpy arrays to store the data.

Note: reading the csv assumes the file containing the input data is called wine.csv and is contained in the same directory as this .ipynb file. If this is not the case, the path to the file inside pd.read_csv must be changed.

[1]:
```
import pandas as pd
import numpy as np
from numba import njit

df = pd.read_csv("./wine.csv", header=None)
INPUTS = df[df.columns[1:14]].to_numpy()
TARGETS = df[df.columns[:1]].to_numpy().flatten()
random_state = np.random.get_state()
np.random.shuffle(INPUTS)
np.random.set_state(random_state)
np.random.shuffle(TARGETS)
```

Then, since all the data points in the input data is continuous, the data should be normalized. I used the traditional feature scaling technique using the equation: $z = \frac{x_i - \mu}{\sigma}$ where x_i is unnormalized input value for the i-th element, μ is the mean of this given feature, σ is the standard deviation of this feature, and z is the normalized input value.

This was easily completed since the inputs are all stored as numpy arrays.

[2]:
```
  def normalize_data(inputs):
    return (inputs - inputs.mean(axis=0)) / (inputs.std(axis=0))

  NORMALIZED_INPUTS = normalize_data(INPUTS)
```

Now that the data is prepped, the next step can be to build the classifiers. The first type of classifier that is going to be used is the baseline classifier. This classifier is uninformed and will be making a

random guess at which class the wine belongs to. Given that this is the baseline randome estimate, the MLP should predict with much greater accuracy the classes of wine.

But first, the baseline classifyer:

[3]:
```
def baseline_prediction(input):
  attribute = input[np.random.randint(13)]
  return int(attribute % 3) + 1

def perform_baseline_prediction(inputs, targets):
  predictions = []

  for input in inputs:
    predictions.append(baseline_prediction(input))

  predictions = np.array(predictions)
  return sum(predictions == targets) / len(targets)

baseline_accuracy = perform_baseline_prediction(INPUTS, TARGETS)
print('Baseline Accuracy: ' + str(int(baseline_accuracy * 10000) / 100) + "%")
```

```
Baseline Accuracy: 29.21%
```

The baseline accuracy should be around 33%, or accurate 1/3 of the time. This is becuase there are 3 possible classes, so a random guess has a 1 in 3 chance of being the correct answer.

Next is the Multi Layer Perceptron classifier. This classifer will be an object that contains the weights for the network as well as a specified learning size and the neural network architecture. When a new MLP is initialized, the weight values are randomly generated.

[4]:
```
class MultiLayerPerceptron:
  def __init__(self, neural_net_structrue, step_size=0.1):
    self.step_size = step_size
    self.network_structure = neural_net_structrue
    self.layers = len(self.network_structure)
    self.input_layer = self.network_structure[0]
    self.output_layer = self.network_structure[-1]
    self.network_params = self.initialize()

  def initialize(self):
    params = {}
    for layer in range(1, self.layers):
      params['Weights' + str(layer)] = np.random.randn(
          self.network_structure[layer-1],
          self.network_structure[layer]) * 0.1
      params['Bias' + str(layer)] = np.zeros(self.network_structure[layer])

    return params
```

Just a few more steps before 5-fold Cross validation can be performed. The first is implementing

feedforward algorithm which will be used to calculate the activations of each layer in the network.

The following formula was used for the feedforward algorithm: $h_\zeta = \sum_{i=1}^{N} x_i w_i + b$ N is the number of nodes in the layerx_i is the ith input valuew_i is the ith weight value b is the bias for this layerh_ζ is the activation value for this layer. Then the hidden layer activation function is given h_ζ to calculate the activation value for this layer. The sigmoid activation fuction is the function used. This formula is: $a_\zeta = g(h_\zeta) = \frac{1}{1+e^{h_\zeta}}$ For the output layer, a softmax activation function was used because there are 3 classes for the output. The softmax formula for the activation of the kth node, h_k, is: $y_k = g(h_k) = \frac{e^{h_k}}{\sum_{j=1}^{N} e^{h_j}}$ Where $\sum_{j=1}^{N} e^{h_j}$ is the sum of activations h_j for the N output nodes.

[5]:
```
def feed_forward(MLP, inputs):
  activations = [inputs]
  for layer in range(1, MLP.layers):
    activation = np.dot(activations[layer - 1], MLP.network_params['Weights' +␣
↪str(layer)]) + MLP.network_params['Bias' + str(layer)]

    if layer < MLP.layers:
      activation = 1 / (1 + np.exp(-activation))

    if layer == MLP.layers:
      activation = np.exp(activation) / np.sum(np.exp(activation))

    activations.append(activation)
  return activations
```

The next step is implementing the back propogation algorithm which will update the layer weights.

Starting with the output layer, the error of the output layer is calculated using: $\delta_o(k) = (y_k - t_k)y_k(1 - y_k)$ Where k is the kth output node y_k is the activation value of the kth output node t_k is the target value of the kth output node

The error of the hidden layers is calculated slightly differently using: $\delta_h(\zeta) = a_\zeta(1 - a_\zeta)\sum_{k=1}^{N} w_\zeta \delta_o(k)$ Where ζ is the ζth node in the hidden layer a_ζ is the activation of the ζth node in the hidden layer w_ζ is the weight of the ζth node in the hidden layer N is the number of nodes in the output layer (or the layer 1 layer forward) $\delta_o(k)$ is the error of the kth node in the layer one layer forward

The weights and biases (bias is w_0) in each layer are then updated according to: $w_k = w_k - \eta\delta_o(k)h_\zeta$ Where w_k is kth weight of the current layer η is the network learning rate δ_o is the error of the layer one layer forward layer h_ζ is the activation of the previous layer

$\delta_o(k)h_\zeta$ is called the *layer gradient* in my program.

[6]:
```
def back_propogation(MLP, targets, activations):
  current_layer = MLP.layers - 1
  output_layer_delta = activations[current_layer] - targets
  layer_gradient = np.dot(np.transpose([activations[current_layer - 1]]),␣
↪output_layer_delta)
  bias_gradient = output_layer_delta.mean(axis=0)
```

```
 MLP.network_params['Weights' + str(current_layer)] = MLP.
↪network_params['Weights' + str(current_layer)] - (MLP.step_size *␣
↪layer_gradient)
 MLP.network_params['Bias' + str(current_layer)] = MLP.network_params['Bias' +␣
↪str(current_layer)] - (MLP.step_size * bias_gradient)
 layer_delta = output_layer_delta

 current_layer -= 1
 while current_layer > 0:
   previous_delta = layer_delta
   layer_delta = np.dot(previous_delta, np.transpose(MLP.
↪network_params['Weights' + str(current_layer + 1)])) *␣
↪activations[current_layer] * (1 - activations[current_layer])
   layer_gradient = np.dot(np.transpose([activations[current_layer - 1]]),␣
↪layer_delta)
   bias_gradient = layer_gradient.mean(axis=0)
   MLP.network_params['Weights' + str(current_layer)] = MLP.
↪network_params['Weights' + str(current_layer)] - (MLP.step_size *␣
↪layer_gradient)
   MLP.network_params['Bias' + str(current_layer)] = MLP.network_params['Bias'␣
↪+ str(current_layer)] - (MLP.step_size * bias_gradient)
   current_layer -= 1
```

The last step before the 5-fold algorithm is implementing the training and testing algorithms. The training algorithm sends the training data through the feedforward algorithm, then compares the feedforward activations to the targets, and updates the weights of the network accordingly using the backpropogation algorithm. The number of iterations of this back and forth is the *epochs* the network experiences. A higher epoch value means a longer time for a network to learn, but a higher chance for overfitting. The testing algorithm is very similar to the training algorithm, except instead of sending the feedforward activations through the backpropogation algorithm, the activation values are compared to the targets, and the number of correct activations is the accuracy for this network.

[7]:
```
def train(MLP, training_inputs, training_targets, epochs):
  inputs = training_inputs
  targets = np.zeros((len(training_targets), 3))
  for i in range(len(training_targets)):
    t = training_targets[i]
    targets[i][t-1] = 1

  for i in range(epochs):
    np.random.rand()
    random_state = np.random.get_state()
    np.random.shuffle(inputs)
    np.random.set_state(random_state)
    np.random.shuffle(targets)
```

```
    for i in range(len(inputs)):
      activations = feed_forward(MLP, inputs[i])
      back_propogation(MLP, [targets[i]], activations)

def test(MLP, testing_inputs, testing_targets):

  activations = feed_forward(MLP, testing_inputs)
  predictions = activations[-1]

  p = []
  for i in range(len(predictions)):
    p.append(predictions[i].argmax() + 1)

  predictions = np.array(p)
  return sum(predictions == testing_targets) / len(testing_targets)
```

The data is now ready to go into a 5-fold cross-validation method and be prepped to begin the MLP training. The inputs and targets will both be passed into the 5-fold method, and the first step from there will be to split the data into 5 partitions. Then the data will enter a loop and iterate over the 5 classes, giving each class one chance to act as the testing sample as the others act as the training samples. The accuracy of this testing sample will then be recorded, and the mean of these accuracies is the network accuracy.

[8]:
```
def five_fold_cross_val(neural_net_structrue, inputs, targets, epochs=200):
  def five_fold(inputs, targets):
    D_inputs = []
    D_targets = []
    D_dict = {'d1': [], 'd2': [], 'd3': [], 'd4': [], 'd5': []}
    D_target_dict = {'d1': [], 'd2': [], 'd3': [], 'd4': [], 'd5': []}

    for x in range(len(inputs)):
      D_dict['d' + str((x % 5) + 1)].append(inputs[x])
      D_target_dict['d' + str((x % 5) + 1)].append(targets[x])
    for k in D_dict.keys():
      D_inputs.append(np.array(D_dict[k]))
      D_targets.append(np.array(D_target_dict[k]))
    return D_inputs, D_targets

  D, t = five_fold(inputs, targets)

  accuracies = []
  for i in range(len(D)):
    testing_inputs = D[i]
    training_inputs = []
    testing_targets = t[i]
    training_targets = []
    for j in range(len(D)):
```

```
      if i != j:
        training_inputs.extend(D[j])
        training_targets.extend(t[j])
    training_inputs = np.array(training_inputs)
    training_targets = np.array(training_targets)

    mlp = MultiLayerPerceptron(neural_net_structrue)
    train(mlp, training_inputs, training_targets, epochs)
    accuracy = test(mlp, testing_inputs, testing_targets)
    accuracies.append(accuracy)

  return np.mean(accuracies)
```

The only other necessary step is to create the network architecture for the neural network. This means specifiying the number of nodes in each layer. This information is contained in a list where the first element of the list is the number of nodes in the input layer, the last is the number of nodes in the output layer, and any other elements are the number of nodes in each hidden layer.

The inputs, targets, and network architecture are sent into the 5-fold cross validation method, and the network is trained and tested.

[9]:
```
basic_nn_structure = [13, 6, 3]

mlp_accuracy = five_fold_cross_val(basic_nn_structure, NORMALIZED_INPUTS,␣
 ↪TARGETS)

print('MLP Accuracy: ' + str(mlp_accuracy * 100) + "%")
```

```
MLP Accuracy: 98.88888888888889%
```

Due to random initialization and random shuffling of the inputs, the final accuracy of the network varies slightly, but using a basic neural network, the accuracy is generally around 97% - 98%. Even 97% accuracy beats the baseline classifyer, so the network appears to be learning, and it also appears that the data is seperable.

There are a few things that can be modified to find the optimal network architecture. First, the number of hidden layers can be adjusted. Second, the number of nodes in the hidden layers can be adjusted.

By experimenting with these two settings, an optimal network can hopefully be achieved.

There are many possible factors when choosing network paramters. One set up that is common is to choose a number of nodes between the number of output nodes and input nodes. Another common idea is to choose a number of nodes far greater than the number of input nodes. I am going to try a variety of layer and node structures to see which architecture performs the best.

Due to computational limits, network architectures with many layers, or layers with large numbers of neurons take too long to analyze. I am going to only look at smaller networks, with maybe a few exceptions.

[10]:
```
in_between_structure_1 = [13, 10, 3]
in_between_structure_2 = [13, 7, 3]
in_between_structure_3 = [13, 4, 3]

large_hidden_layer = [13, 50, 3]
extra_large_hidden_layer = [13, 500, 3]
extremely_large_hidden_layer = [13, 1000, 3]

many_smaller_hidden_layers = [13, 4, 4, 4, 4, 4, 3]
many_larger_hidden_layers = [13, 7, 7, 7, 7, 7, 3]

large_to_small_hidden = [13, 50, 25, 7, 3]
small_to_large_hidden = [13, 7, 25, 50, 3]

print("Beginning tests on network structures with number of hidden nodes in␣
 ↪between number of input nodes and output nodes")
in_between_structure_1_accruacy = five_fold_cross_val(in_between_structure_1,␣
 ↪NORMALIZED_INPUTS, TARGETS)
in_between_structure_2_accruacy = five_fold_cross_val(in_between_structure_2,␣
 ↪NORMALIZED_INPUTS, TARGETS)
in_between_structure_3_accruacy = five_fold_cross_val(in_between_structure_3,␣
 ↪NORMALIZED_INPUTS, TARGETS)
print("Network testing complete\n\nMoving onto testing networks with a single␣
 ↪large hidden layer")

large_hidden_layer_accuracy = five_fold_cross_val(large_hidden_layer,␣
 ↪NORMALIZED_INPUTS, TARGETS)
extra_large_hidden_layer_accuracy =␣
 ↪five_fold_cross_val(extra_large_hidden_layer, NORMALIZED_INPUTS, TARGETS)
extremely_large_hidden_layer_accuracy =␣
 ↪five_fold_cross_val(extremely_large_hidden_layer, NORMALIZED_INPUTS, TARGETS)
print("Network testing complete\n\nMoving onto testing networks with many␣
 ↪hidden layers")

many_smaller_hidden_layers_accuracy =␣
 ↪five_fold_cross_val(many_smaller_hidden_layers, NORMALIZED_INPUTS, TARGETS)
many_larger_hidden_layers_accuracy =␣
 ↪five_fold_cross_val(many_larger_hidden_layers, NORMALIZED_INPUTS, TARGETS)
print("Network testing complete\n\nMoving onto testing networks with both large␣
 ↪hidden layers, and many hidden layers")

large_to_small_hidden_accuracy = five_fold_cross_val(large_to_small_hidden,␣
 ↪NORMALIZED_INPUTS, TARGETS)
small_to_large_hidden_accuracy = five_fold_cross_val(small_to_large_hidden,␣
 ↪NORMALIZED_INPUTS, TARGETS)
print("Network testing complete\n\nResults:")
```

```
print(f"In between results:\n\tnodes = 10: {in_between_structure_1_accruacy *␣
↪100}%\n\tnodes = 7: {in_between_structure_2_accruacy * 100}%\n\tnodes = 4:␣
↪{in_between_structure_3_accruacy * 100}%")
print(f"Large hidden number of nodes in hidden layer results:\n\tnodes = 50:␣
↪{large_hidden_layer_accuracy * 100}%\n\tnodes = 500:␣
↪{extra_large_hidden_layer_accuracy * 100}%\n\tnodes = 1000:␣
↪{extremely_large_hidden_layer_accuracy * 100}%")
print(f"Many hidden layers results:\n\tlayers = 5, nodes = 4:␣
↪{many_smaller_hidden_layers_accuracy * 100}%\n\tlayers = 7, nodes = 7:␣
↪{many_larger_hidden_layers_accuracy * 100}%")
print(f"Multiple hidden layers and many nodes results:\n\tlayer_1 = 50 nodes,␣
↪layer_2 = 25 nodes, layer_3 = 7 nodes: {large_to_small_hidden_accuracy *␣
↪100}%\n\tlayer_1 = 7 nodes, layer_2 = 25 nodes, layer_3 = 50 nodes:␣
↪{small_to_large_hidden_accuracy * 100}%")
```

```
Beginning tests on network structures with number of hidden nodes in between
number of input nodes and output nodes
Network testing complete

Moving onto testing networks with a single large hidden layer
Network testing complete

Moving onto testing networks with many hidden layers
Network testing complete

Moving onto testing networks with both large hidden layers, and many hidden
layers
Network testing complete

Results:
In between results:
        nodes = 10: 98.31746031746033%
        nodes = 7: 97.76190476190474%
        nodes = 4: 98.33333333333331%
Large hidden number of nodes in hidden layer results:
        nodes = 50: 98.31746031746033%
        nodes = 500: 98.31746031746033%
        nodes = 1000: 98.31746031746033%
Many hidden layers results:
        layers = 5, nodes = 4: 29.841269841269842%
        layers = 7, nodes = 7: 34.85714285714286%
Multiple hidden layers and many nodes results:
        layer_1 = 50 nodes, layer_2 = 25 nodes, layer_3 = 7 nodes:
39.333333333333336%
        layer_1 = 7 nodes, layer_2 = 25 nodes, layer_3 = 50 nodes:
34.285714285714285%
```

Given these results, the networks with the highest accuracy were networks with many nodes in a single hidden layer. One final test is going to be network and algorithm effieciency. I am going to compare the speed of the 50 node architecture to the speed of the 1000 node architecture.

[11]:
```
import time

start = time.time()
five_fold_cross_val(large_hidden_layer, NORMALIZED_INPUTS, TARGETS)
end = time.time()
smaller_effieciency = end - start

start = time.time()
five_fold_cross_val(extremely_large_hidden_layer, NORMALIZED_INPUTS, TARGETS)
end = time.time()
larger_efficiency = end - start

print(f"The network with 50 hidden nodes took {smaller_effieciency} seconds to␣
 ↪train")
print(f"The network with 1000 hidden nodes took {larger_efficiency} seconds to␣
 ↪train")
```

```
The network with 50 hidden nodes took 21.94187021255493 seconds to train
The network with 1000 hidden nodes took 38.62543773651123 seconds to train
```

2 Conclusion

In conclusion, the multi layer perceptron was able to significantly improve the accuracy of predictions on this data. The baseline predictor had an average accuracy around 30%, and the basic multi layer perceptron predictor had an accuracy above 98%. After testing and optimization of the network architecture, it became apparent that a single hidden layer with many nodes was the network architecture that gave the greatest accuracy. Then after testing the training efficiency, it took about 3 times longer to train the network with 1000 nodes than the network with 50 nodes. Therefore the architecture with one hidden layer and 50 nodes in the layer is the best predictor I was able to find.

[12]:
```
final_accuracy = five_fold_cross_val(large_hidden_layer, NORMALIZED_INPUTS,␣
 ↪TARGETS)
print(f"Final accuracy using a network architecture with 50 nodes in the hidden␣
 ↪layer is {final_accuracy * 100}%")
```

```
Final accuracy using a network architecture with 50 nodes in the hidden layer is
98.31746031746033%
```