Homework 1 - Machine Learning

Catherine Baker

January 2024

Contents

1 Random Forests

See CatherineBakerQ1.py to view my basic implementation of the Random Forests classifier. My code consistently reports about 97% accuracy before any cross validation is performed.

1.1 Cross Validation - m

See CatherineBaker Q1A.py to view my implementation, which builds off of the code from Section 1. I use sklearn.model_selection.cross_val_score() to run cross validation on the Random Forest model.

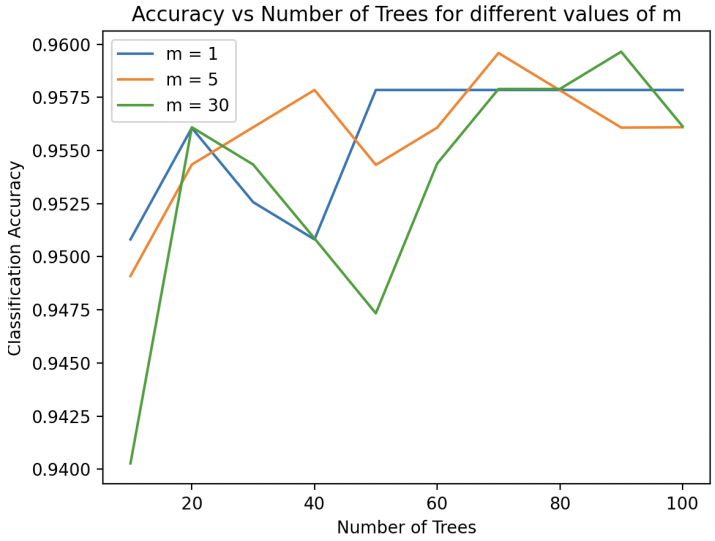

Figure 1: The plot produced by my CatherineBakerQ1A.py code. For our data, $p = 30$ and $\text{int}(\sqrt{p}) = 5$

We can see how variable the classification accuracy is when changing tree number and m in Figure 1. The value of m can affect the diversity and correlation of the trees in our random forest which plays a huge role in accuracy. The number of trees parameter can drastically affect classification accuracy especially with different p values. Graphs like ours show how fine-tuning parameters like m and tree number is so important to classification accuracy. We can see that the graph eventually plateaus on the right showing that after a certain increase in tree number, the value of m has less effect on the classification accuracy.

1.2 Cross Validation - n_{min}

See CatherineBakerQ1B.py to view my implementation, which builds off of the code from Section 1 and Section 1.1. I use sklearn.model_selection.cross_val_score() to run cross validation on the Random Forest model.

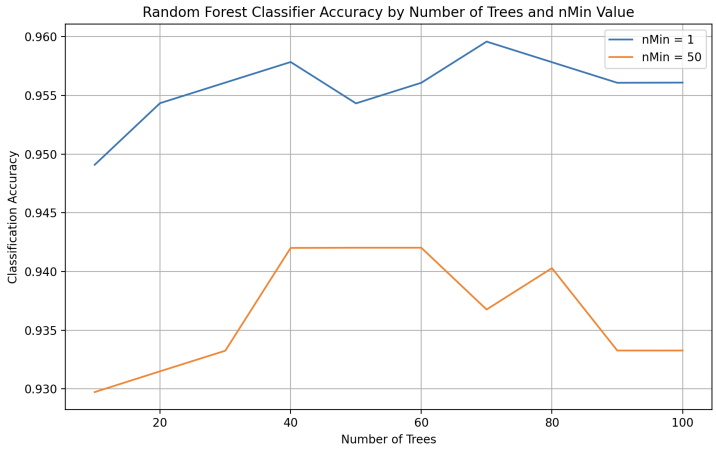

Figure 2: The plot produced by my CatherineBakerQ1B.py code.

This graph shows how changing the depth control parameter can affect the models classification accuracy. Here in Figure 2 we see a very slight difference between the two values of 1 and 50 with an increase in accuracy for $n_{min} = 1$. Increasing this value ensures that the tree does not overfit exactly to the training data. Here it seems that a value of $n_{min} = 50$ was too high to provide any improvement in the training data that would have led to an increase in classification accuracy. This depth control is shown over various tree numbers. The tuning of tree number does seem to affect the classification accuracy compared to depth control.

2 Adaboost M.1.

We plot with 1-node, 3-node, and 5-node trees.

2.1 Cross Validation - tree size

Figure 3 shows the results from our Adaboosting. Overall the node depth of each tree did not have a tremendous effect on classification accuracy when combined with number of weak learners. By tuning both of these parameters we can produce a high classification accuracy with the highest performing category being the 1-node depth trees with over 90 weak learners.

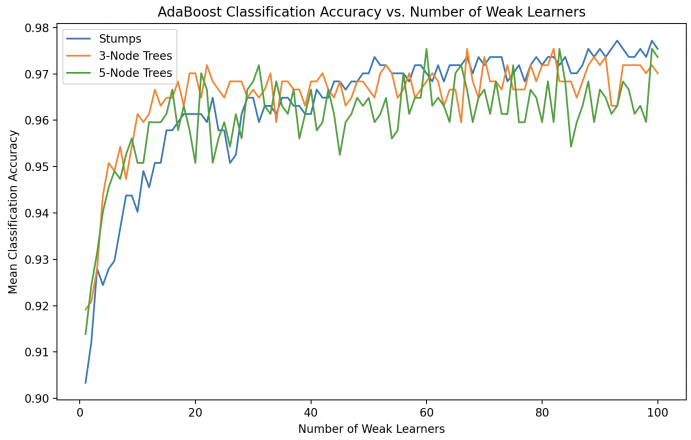

Figure 3: The plot produced by my CatherineBakerQ2A.py code.

2.2 Cross Validation - shrinkage

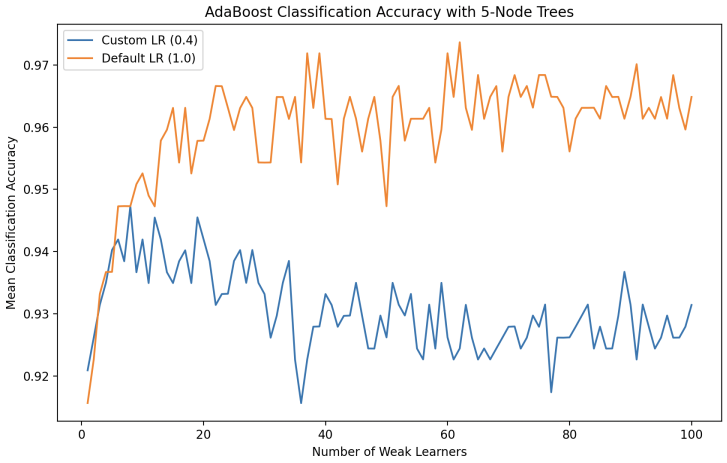

Figure 4: The plot produced by my CatherineBakerQ2B.py code.

Figure 4 shows how adjusting learning rate can affect classification accuracy. Here by shrinking our learning average we lower our classification accuracy. This suggests that in our case the lower learning rate might have caused the model to underfit to our training data (as usually it helps to prevent overfitting). We may see convergence to a higher learning rate with more weak learners. Regardless of our results, it further shows how sensitive the parameters of boosting are, especially learning rate.

3 Neural Networks

3.1 Forward Propagation

If we have input layer x_i and weights w_{ji} for input neuron i and hidden neuron j, then we can write the activation a_j of the hidden layer as the sum of the weights times the input layer plus the bias over all values of i:

$$a_j = \Sigma_{i=1}^{N} w_{ji} x_i + b_i$$

for $1 \leq i \leq N$, which represents our activation function. For a two layer neural network, the first layer of this would be:

$$a_j = \Sigma_{i=1}^{N} w_{ji}^{(1)} x_i + b_i^{(1)}$$

Next, the hidden layer with output z_j will be produced by the logistic sigmoid function:

$$z_j = g(a_j) = tanh(a_j) = tanh(\Sigma_{i=1}^{N} w_{ji}^{(1)} x_i + b_i^{(1)})$$
$$\Rightarrow \frac{e^{\Sigma_{i=1}^{N} w_{ji}^{(1)} x_i + b_i^{(1)}} - e^{-\Sigma_{i=1}^{N} w_{ji}^{(1)} x_i + b_i^{(1)}}}{e^{\Sigma_{i=1}^{N} w_{ji}^{(1)} x_i + b_i^{(1)}} - e^{-\Sigma_{i=1}^{N} w_{ji}^{(1)} x_i + b_i^{(1)}}}$$

which I believe is in its simplest form already. Lastly we want to find our output y_k which is equivalent to our a_k because of the linearity of the activation functions. Because we have a two-layer neural network, we must use the weights and bias term from the second layer and our input is now z_i instead of x_i:

$$y_k = \Sigma_{i=1}^{N} w_{ji}^{(2)} z_i + b_i^{(2)}$$

for $1 \leq i \leq N$.

3.2 Computing δ_k

To find δ_k we would usually find our loss function, take the gradient, and take a step in the opposite direction. In our case, we can take the gradient of the given error function to find this value (with respect to our activation function where $y_k = a_k$):

$$\delta_k = \frac{\partial E_n}{\partial a_k} = \frac{\partial}{\partial a_k}(\frac{1}{2}\Sigma_{k=1}^{K}(y_k - t_k)^2) = \frac{\partial}{\partial y_k}(\frac{1}{2}\Sigma_{k=1}^{K}(y_k - t_k)^2)$$

If we expand the summation it results in a series of $(\frac{1}{2}(y_k - t_k)^2)$'s added together. When taking the derivative of this expanded equation we will only account for one k value at a time. And because these terms are added, all but the term with the k values we focus on will turn to 0. Because of this, taking the derivative of the summation will result in the same value as taking the derivative of $\frac{1}{2}(y_k - t_k)^2$ directly, with respect to y_k:

$$\frac{\partial}{\partial y_k}(\frac{1}{2}\Sigma_{k=1}^{K}(y_k - t_k)^2) = (y_k - t_k) = \delta_k \tag{1}$$

And so we have our δ_k value.

3.3 Computing δ_j

Referring to Slide 37 of Lecture 13 we can see δ_j clearly defined as:

$$\delta_j = g'(IN_j)\Sigma_k w_{jk}\delta_k$$

For our context, $IN_j = a_j$ and because we are working with a 2-layer neural network we can use weights $w_{jk}^{(2)}$:

$$\delta_j = g'(a_j)\Sigma_k w_{jk}^{(2)}\delta_k \tag{2}$$

To find the derivative of $g(a_j)$ we refer to an online resource for the Hyperbolic Identities and another online article on Hyperbolic Derivatives which shows us the following:

$$g'(a_j) = \frac{\partial}{\partial a_j}(tanh(a_j)) = sech^2(a_j) = 1 - tanh^2(a_j) = 1 - z_j^2$$

Plugging this back into Equation 2, we get:

$$\delta_j = (1 - z_j^2)\Sigma_k w_{jk}^{(2)}\delta_k$$

as our equation for δ_j.

3.4 Calculating Derivatives w.r.t. $w_{ji}^{(1)}$ and $w_{kj}^{(2)}$

We have previously (in Section 3.1) defined y_k and a_j as:

$$y_k = \Sigma_{i=1}^{N} w_{ji}^{(2)} z_i + b_i^{(2)} \tag{3}$$

$$a_j = \Sigma_{i=1}^{N} w_{ji}^{(1)} x_i + b_i^{(1)} \tag{4}$$

3.4.1 $w_{kj}^{(2)}$

To find $\frac{\partial E_n}{\partial w_{kj}^{(2)}}$ we must rewrite it as $\frac{\partial E_n}{\partial y_k}\frac{\partial y_k}{\partial w_{kj}^{(2)}}$. We can now use our previously calculated Equation 1 to report $\frac{\partial E_n}{\partial y_k}$ as:

$$\frac{\partial E_n}{\partial y_k} = (y_k - t_k)$$

leaving only $\frac{\partial y_k}{\partial w_{kj}^{(2)}}$. Because it is linear, the derivative is easily calculated as:

$$\frac{\partial y_k}{\partial w_{kj}^{(2)}} = z_i$$

So we can rewrite the derivative with respect to the second layer weights as:

$$\frac{\partial E_n}{\partial w_{kj}^{(2)}} = (y_k - t_k)z_i = \delta_k z_i$$

3.4.2 $w_{ji}^{(1)}$

As above let us rewrite our derivative taking into account that we are now working with the input layer and first layer of the neural network rather than the second layer (using a_j instead of y_k):

$$\frac{\partial E_n}{\partial w_{ji}^{(1)}} = \frac{\partial E_n}{\partial a_j}\frac{\partial a_j}{\partial w_{ji}^{(1)}}$$

Because we aren't able to directly calculate $\frac{\partial E_n}{\partial a_j}$, we can rewrite it as the sum of the derivative in terms of y_k over the values of k:

$$\frac{\partial E_n}{\partial a_j} = \Sigma_{k=1}^{K}(\frac{\partial E_n}{\partial y_k}\frac{\partial y_k}{\partial a_j})$$

which is much easier to find. The value of the first part $\frac{\partial E_n}{\partial y_k} = (y_k - t_k) = \delta_k$. The second part can again be rewritten to be understood more directly:

$$\frac{\partial y_k}{\partial a_j} = \frac{\partial y_k}{\partial z_j}\frac{\partial z_j}{\partial a_j} = w_{kj}^{(2)}g'(a_j)$$

by definition. Plugging this back in we get:

$$\frac{\partial E_n}{\partial a_j} = \Sigma_{k=1}^{K}(\delta_k w_{kj}^{(2)}g'(a_j))$$

We can now pull out the $g'(a_j)$ term as it does not rely on k which allows us to see how this equation is equivalent to δ_j:

$$\frac{\partial E_n}{\partial a_j} = g'(a_j)\Sigma_{k=1}^{K}(\delta_k w_{kj}^{(2)}) = \delta_j$$

Now that we have that simplified, we can look back to the $\frac{\partial a_j}{\partial w_{ji}^{(1)}}$ derivative which is equal to x_i (referring to Equation 4). Putting these two values together we get:

$$\frac{\partial E_n}{\partial w_{ji}^{(1)}} = \frac{\partial E_n}{\partial a_j}\frac{\partial a_j}{\partial w_{ji}^{(1)}} = \delta_j x_i$$

which is my final answer.

3.5 Updating the Weights with Gradient Descent

We will refer to the original weights as w_{kj} and the new, updated weights as w'_{kj}. To update any given weight we subtract the gradient descent multiplied by the learning rate (η) (Slide 39 Lecture 13):

$$w'_{xy} = w_{xy} - \eta\frac{\partial E_n}{\partial w_{xy}}$$

For the second layer weights this would be:

$$w'^{(2)}_{kj} = w^{(2)}_{kj} - \eta\frac{\partial E_n}{\partial w^{(2)}_{kj}} = w^{(2)}_{kj} - \eta(\delta_k z_j)$$

and for the first layer weights this would be:

$$w'^{(1)}_{ji} = w^{(1)}_{ji} - \eta\frac{\partial E_n}{\partial w^{(1)}_{ji}} = w^{(1)}_{ji} - \eta(\delta_j x_i)$$

from Section 3.4.

4 Boosting

Figure 5 shows the plot for the given data points with class 1 as blue circles and class 2 as red squares. To perform Adaboosting as shown in the toy example in class we initially set all weights to $\frac{1}{10}$ (because N = 10 total points).

First we'll start with a half-plane at x = 1 for our first classifier. We will (from now on) refer to blue circle points (class 1) from left to right and top to bottom on the graph as the set of points $B = \{b_1, b_2, b_3, b_4, b_5\}$ and the red square points (class 2) as $R = \{r_1, r_2, r_3, r_4, r_5\}$.

4.1 First Classifier: x = 1

The current set of weights for class 1 is $W_b = \{w_{b1}, w_{b2}, w_{b3}, w_{b4}, w_{b5}\} = \{0.1, 0.1, 0.1, 0.1, 0.1\}$ and for class 2 is $W_r = \{w_{r1}, w_{r2}, w_{r3}, w_{r4}, w_{r5}\} = \{0.1, 0.1, 0.1, 0.1, 0.1\}$. We choose our first weak classifier to be x = 1. This misclassifies our three rightmost class 1 points (blue) giving an err_m value of $\frac{0.3}{1} = 0.3$ (the sum of the weights of misclassified points over the sum of all weights).

Next we calculate $a_m = 0.5 * ln(\frac{1-err_m}{err_m}) = 0.5 * ln(\frac{0.7}{0.3}) \approx 0.42365$. We will update the misclassified points by multiplying them by $exp\{a_m\}$ and the correctly classified points by $exp\{-a_m\}$ citing Lecture 15 Slide 12 and Figure

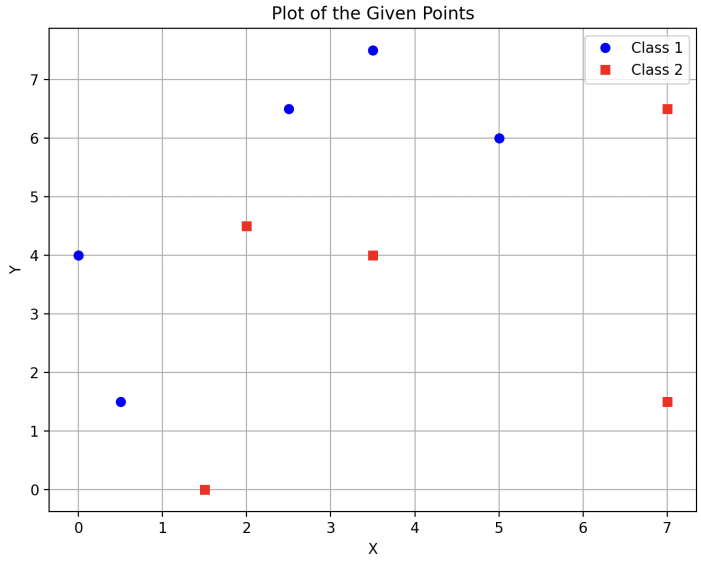

Figure 5: The plot produced by my CatherineBakerQ4.py code.

1 from Evaluating enterprise risk of default using boosting procedures for our formulas. This changes our class 1 and 2 weights to the following:

$$W_b = \{0.065465, 0.065465, 0.15275, 0.15275, 0.15275\}$$
$$W_r = \{0.065465, 0.065465, 0.065465, 0.065465, 0.065465\}$$

4.2 Second Classifier: y = 5.25

We choose our next classifier to account for the increased weights of our top 3 class 1 points. This classifier misclassified our b_1 and b_2 points as well as our r_4 (on top) point. Calculating the sum of misclassified weights, we get $0.065465 + 0.065465 + 0.065465 = 0.196395$. Then $a_m = 0.5 * ln(\frac{0.803605}{0.196395}) \approx 0.70449$. Updating our final weights as before we get:

$$W_b = \{0.132424, 0.132424, 0.075514, 0.075514, 0.075514\}$$
$$W_r = \{0.032363, 0.032363, 0.032363, 0.132424, 0.032363\}$$

We can see how our previously misclassified weights w_{b3}, w_{b4}, w_{b5} have all shifted in their importance with this new classifier. Even with our two classifiers we still have some misclassified points so we will do another iteration.

4.3 Third Classifier: x = 6

We will consider a final classifier x = 6 to further separate class 1 and 2 points. This classifier treats all blue to its left as correctly classified and all red to its right as the same. The three points r_1, r_2, r_3 are considered misclassified. This gives the following values:

$$err_m = 0.032363 + 0.032363 + 0.032363 = 0.97089$$

$$a_m = 0.5 * ln(\frac{0.02911}{0.97089}) \approx -1.75357$$

$$W_b = \{0.764772, 0.764772, 0.436107, 0.436107, 0.436107\}$$

$$W_r = \{0.0056038, 0.0056038, 0.0056038, 0.764772, 0.186902\}$$

Because this combination of weak classifiers correctly classifies all our points we will stop adding classifiers. We will illustrate this boundary by updating our CatherineBakerQ4.py code in Figure 6. Our final weights are shown above.

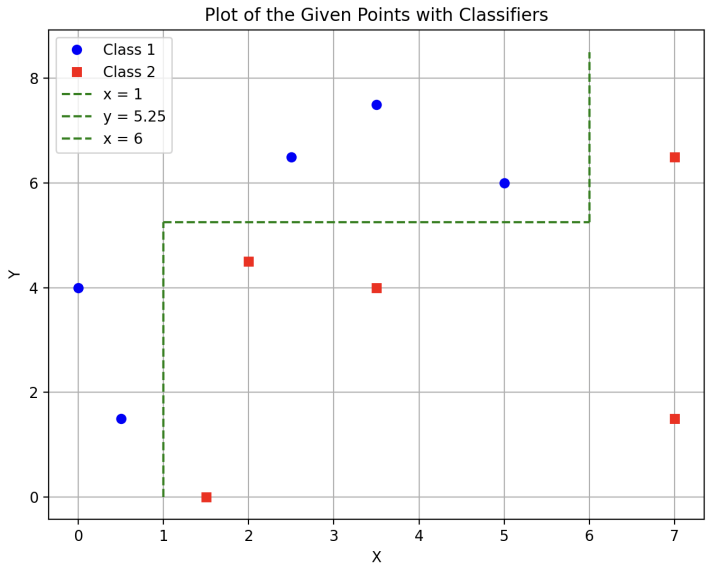

Figure 6: The plot produced by my updated CatherineBakerQ4.py code. This code adds in plotted green line segments on the decision boundary of the weak classfiers chosen.