**Final Project**

Github: <https://github.com/yesgomez/finalproject>

Part 1. 8x3x8 encoder (see: <https://github.com/yesgomez/finalproject/8x3x8_encoder.py>)

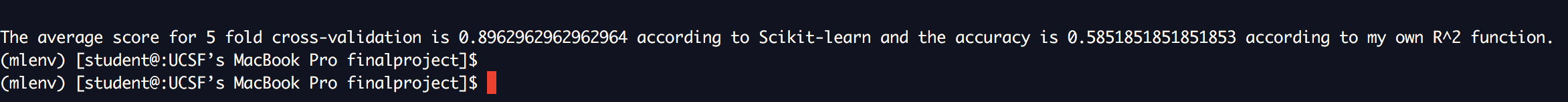
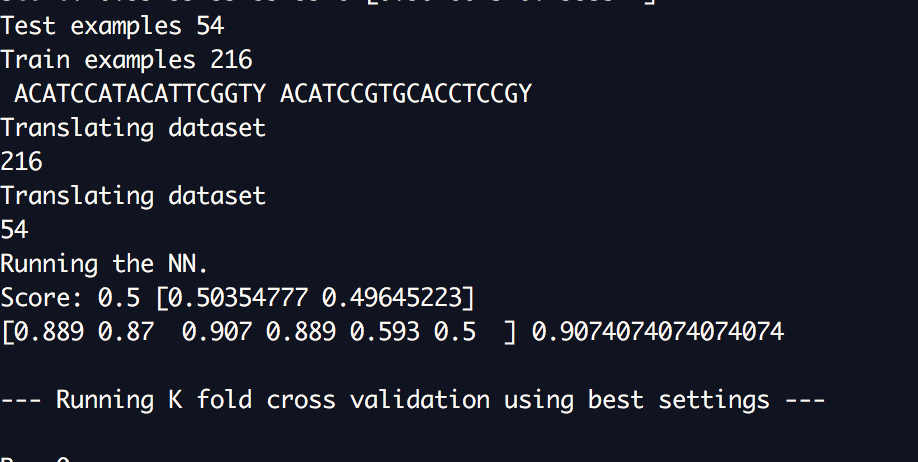
Part 2. Learning procedure for an ANN.

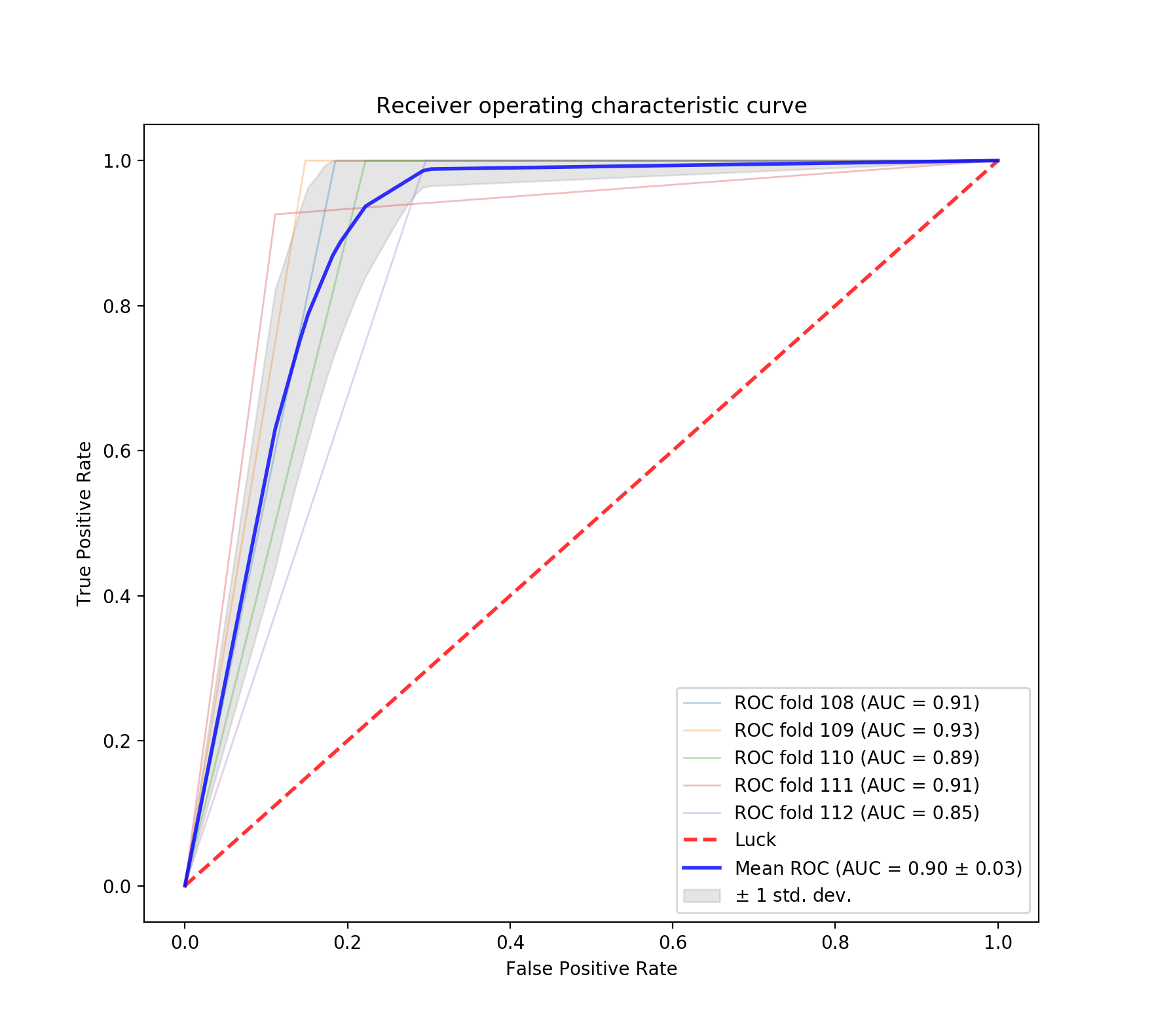
* I tried to encode each nucleotide as binary ([A, T, C, G] = ['01000001', '01010100', '01000011', '01000111']), but ran into np.add and subtract dtype errors
  + Instead I encoded each letter as an integer ([A, T, C, G] = [2, 3, -2, -3]) and Yes/No was encoded as 1/-1.
* Each DNA sequence (17 bp) is an entry, fed in as an array of shape (17,1).
* Input layers, nodes = 1, 17
  + (Since each nt is an independent variable, it is a feature.)
* Hidden layers, nodes = 1, 3
  + (I have ~220 training examples\*. To follow the rule of 10X more examples than weights I start with ~2 hidden nodes. To keep the number of hidden nodes < # of input nodes and > # output nodes, I should have between 2-17. These are just rules of thumb, however.)
* Output layers, nodes = 1, 2
  + The probability of being a site and of not being a site. (This could also be represented as a single node where 0<x<0.5 is not a site and 0.5<x<1 is a site.)

Part 3. Training regime.

* Format the negative data in the same manner as the positive data (text file with 17 nt per line). Remove all negative examples that match the positive data.
* For both sets
  + translate nt to numeric array
  + import to NN as a matrix of (17, len(file))
* Use an 80/10/10 scheme to split the data (80% as training data, 10% as testing data, and 10% as validation data)
  + Since there are 137 true positives, I can use ~109 as positive training examples. \*To not overweight the negative data, I would use the same number of negative training examples, thus giving me a pool of 218 training examples.
  + The stop criterion should be when the change in the gradient is 0 over >1 iterations because that means the weights have collectively reached a (global) minima. I simply used 2000 iterations.

Part 4. Cross validation experiments.

* I wrote a script that will run K-fold cross validation for a given dataset and a NN.
  + It uses average R² score (“number that indicates the proportion of the variance in the dependent variable that is predictable from the independent variable(s)”) to determine the system’s performance. Should be 0 < n < 1
  + I also compared it to the output of scikit-learn’s Score function
* I ended up using a classifier with a that had 16 nodes in a single hidden layer, logistic activation function, and ran for 2000 iterations. Alpha was set to 0.001. Number of hidden nodes and alpha were varied concurrently
  + Increasing alpha too much decreased the (R) scores because it made it more difficult to converge
  + Increasing the number of hidden nodes seemed to increase the scores, possibly because it could converge faster. Since this is not a high dimensional classification problem, more nodes are less likely to overfit
  + This is a sample output of the alpha testing, where the 6 item array shows how the evaluation scores changed for increasing values of alpha.
  + Above it, each individual sample prints its score and one example of the raw classification probabilities (where tuple(left) = probability of being ‘N(ot a site)’, tuple(right) = probability of being ‘a site’, and the sum(tuple)=1)
* I also calculated ROC curves for each fold of the cross-validation (below). They have a large area underneath and are very consistent

Part 5. See <https://github.com/yesgomez/finalproject/predictions.txt> for details.