Chemical compound classifier

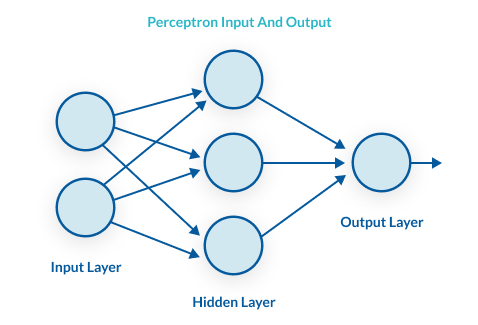
**aditya joshi**

The given dataset contains details about organic chemical compounds including their chemical features, isomeric conformation, names and the classes in which they are classified. The compounds are classified as either ‘Musk’ or ‘Non-Musk’ compounds. Your task is to build a classification model on the given data using any Deep Learning approach that you deem appropriate viz. Multi-Layer Perceptron, CNN, RNN, etc. or you could also use transfer learning approaches through selection of appropriate pre-trained model. The data has to be split in a 80:20 ratio for training and validation datasets. You can perform whatever preprocessing and post-processing operations on the data that may help you improve the performance of your model. You are required to report the performance measures of the model viz. Accuracy( Training and Validation) and Loss(Training and Validation) graphs, F1 score, precision, recall, etc. along with a well detailed report of what models, pre-processing, post-processing approaches you have used and why you chose to use these approaches

This code helps you classify compounds either 'Musk' or 'Non-Musk' using Multi-Layer perceptron.

**What is Multi-Layer perceptron?**

A multilayer perceptron (MLP) is a feedforward artificial neural network that generates a set of outputs from a set of inputs. An MLP is characterized by several layers of input nodes connected as a directed graph between the input and output layers. MLP uses back propagation for training the network. MLP is a deep learning method.



If a multilayer perceptron has a linear activation function in all neurons, that is, a linear function that maps the weighted inputs to the output of each neuron, then linear algebra shows that any number of layers can be reduced to a two-layer input-output model. In MLPs some neurons use a nonlinear activation function that was developed to model the frequency of action potentials, or firing, of biological neurons

The two historically common activation functions are both sigmoids, and are described by



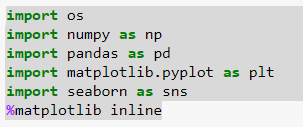
### **Layers**

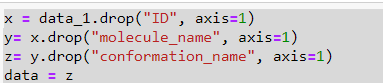
The MLP consists of three or more layers (an input and an output layer with one or more hidden layers) of nonlinearly-activating nodes. Since MLPs are fully connected, each node in one layer connects with a certain weight to every node in the following layer.

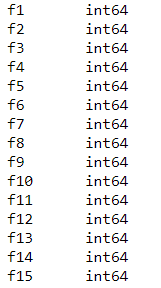
**Why is Multi-Layer perceptron?**

MLPs are suitable for classification prediction problems where inputs are assigned a class or label.They are also suitable for regression prediction problems where a real-valued quantity is predicted given a set of inputs. Data is often provided in a tabular format, such as you would see in a CSV file or a spreadsheet.

**Pre-processing Technique**

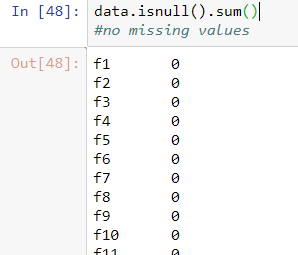
**LIBRARIES IMPORTED**

**DROPPING UNECCESARY COLUMNS**

**DATA TYPES**

**MISSING VALUES**What are missing values in DataSet?

The Missing Value Analysis procedure performs three primary functions: Describes the pattern of missing data. ... Fills in (imputes) missing values with estimated values using regression or EM methods; however, multiple imputation is generally considered to provide more accurate results.



**BUILDING OUR NN MODEL**We need to do the below steps to build our nn model.

* Define Network structure ( # of input units, # of hidden units, etc).
* Initialise the model’s parameters
* Perform the below steps in loop until we get minimum cost/optimal parameters.

1. Implement forward propagation
2. Compute loss
3. Implement backward propagation to get the gradients
4. Update parameters

Then merge all the above steps into one function we call “nn\_model()”.

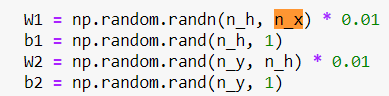
Once we built “nn\_model()” and learnt the right parameters, we can make predictions on new data.

1. **Define Network Structure**: As mentioned earlier, for input layer number of nodes will be 2, and for hidden layer i set it to 4. By choosing more nodes in this layer, we can make model learn complex functions. But it comes at a cost of heavy computation to make predictions and learn the network parameters. More number of hidden layers and nodes could also lead to over-fitting of our data.

**n\_x = X.shape[0] # size of input layer`  
n\_h = 4  
n\_y = Y.shape[0] # size of output layer**

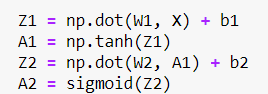
2. **Initialize the model’s parameters**: W1 (weight  
matrix for hidden layer) and W2(wight matrix for  
output layer) parameters are initialized randomly using the numpy random function. Multiplied by 0.01 as we do not want the initial weights to be large, because it will lead to slower learning. b1 and b2 are initialized to zeros.

W1 — weight matrix of shape (n\_h, n\_x) for hidden layer  
b1 — bias vector of shape (n\_h, 1)  
W2 — weight matrix of shape (n\_y, n\_h) for output layer  
b2 — bias vector of shape (n\_y, 1)

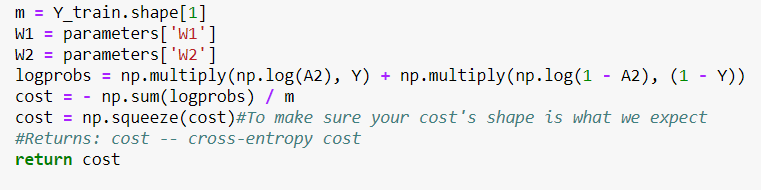


3. **Forward Propagation**: During forward propagation the  
input feature matrix is fed to the every neuron in the hidden layer. Which will be multiplied by the respective initial set of weights(W1) and bias(b1) will be added to form Z1 matrix (linear transformations of the given inputs). Then we apply the non-linearity to Z1 by feeding it through an activation function (to apply non-linearity). We chose ‘tanh’ as our activation function as it fits to many scenarios. The output of this activation function/hidden layer will be A1 (which is a matrix of size (4,1) contains the activations from the 4 neurons i.e a1, a2, a3 a4).

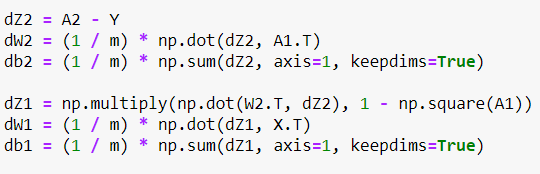
For the next layer which is the final output layer in our case, we multiply the inputs from the previous layer (A1) with the initial weights of output layer(W2), add bias(b2) to form Z2. Then apply the sigmoid activation function on Z2 to produce out final output A2 (which is our predictions). We used sigmoid for our final layer as we want our output to be between 0 and 1. Based on the probability threshold we can decide weather the output is red or blue. This is how nn makes predictions during forward propagation, which is just a sequence of matrix multiplications and application of activation function(s).



1. **Compute Loss**: Now that we have our predictions, next step would be to check how much our predictions differ from the actual values, i.e loss/error. Here we do not use mean square error (MSE) to compute our loss as our prediction function is non-linear(sigmoid). Squaring the prediction will results in non-convex function with many local minimums. In such case gradient descent many not find the optimal global minimum. Hence we use the binary Cross\_Entropy loss (log-likelihood method for error estimate), this cost function is convex in nature, so reaching the global minimum point (minimum loss point) will be easier.



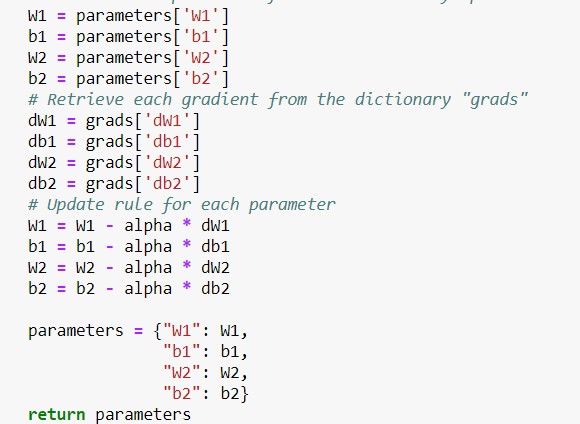
1. **BackPropagation/Gradient Descent (GD)**: Back propagation is used to calculate the gradients(slope/derivatives) of the loss function with respect to the model parameters(w1,b1,w2,b2). To minimize our cost we use the GD algorithm, which uses the computed gradients to update the parameters so that the our cost keeps reducing over iterations, i.e it help move towards global minimum.



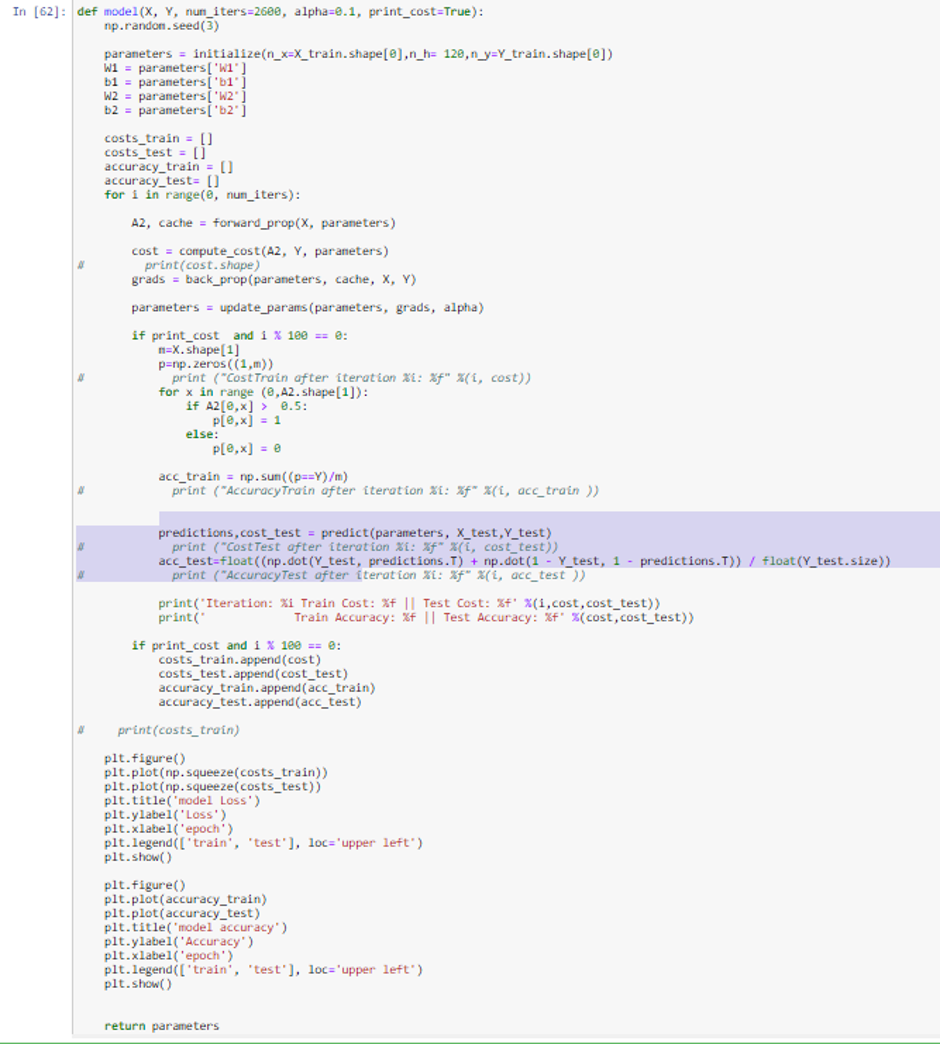
7. **Update the parameters**: Once we have computed our gradients, we multiply them with a factor called learning-rate (converging rate) and subtract from the initial parameters to get the updated parameters(weights and biases). Learning rate should be minimal so that we will not miss the global minimum point.

Multiply the gradients by learning rate

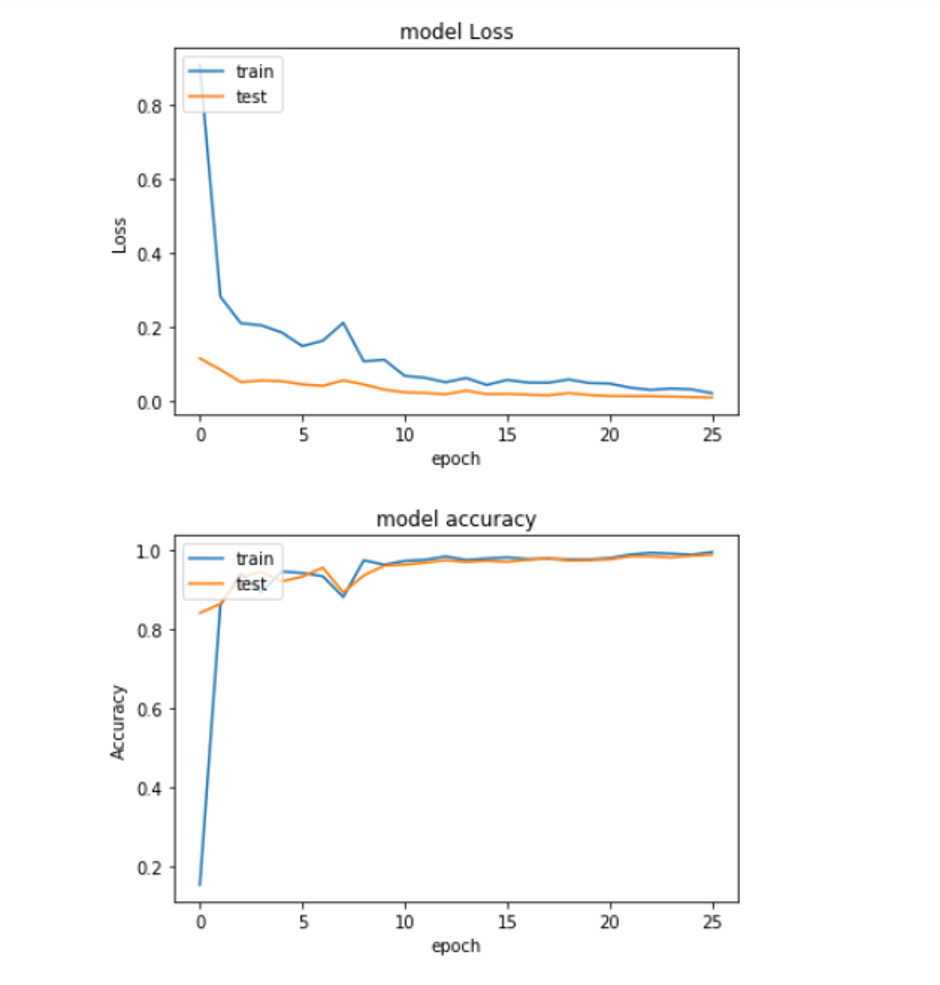
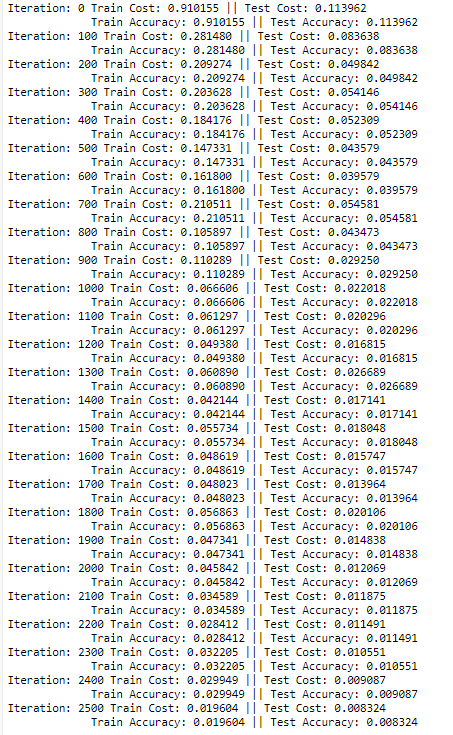
Subtract from weights

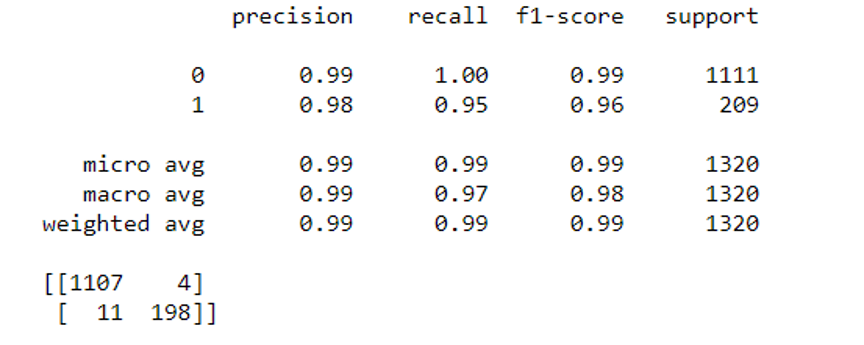


Now we have performed one round of forward propagation and backward propagation for all the training examples, i.e we completed 1 epoch. We need to repeat these steps over multiple epochs till our cost is minimum (model reaches global minimum point) or the learning stops (no updates to the parameters).



Now lets train our final model by running the function nn\_model over 2500 epochs and see the results.



As we can see from the results, our nn model has learnt the patterns well. Which is able to learn non-linear decision boundaries which separates the classes. And our cost started from 0.11 and reached to 0.083 after 2500 epochs. Final accuracy came to 98% which is Quite acceptable.



THANK YOU