CSCI 740 **Spring 2021**

Assignment 3

Due Date: 11:30 pm, May 12, 2021

100 points (+40 Extra Credit points)

Read and Follow Assignment Instructions Carefully

1. This is an individual assignment. All work submitted must be your own.
2. First read the entire assignment description to get the big picture; make notes down the control flow, expected functionality of the various methods and why you are being asked to implement specific items.
3. Read and understand Jupyter Workbooks 12 and 13.
4. You are **allowed** to use any Scikit-Learn **models or functions.**
5. Download the wine quality dataset from:

   http://archive.ics.uci.edu/ml/datasets/Wine+Quality

   You will be using the white wine dataset: "winequality-white.csv"

6. Deliverables:
   a. The code and answers should be written in a Jupyter notebook named `<lastname>_<firstname>_assignment3.ipynb.
   b. Compress **and submit as `<lastname>_<firstname>_assignment3.zip`**
7. Make sure you copy each question with the question number as a Markdown Cell in Jupyter and have the code response right below it (as shown in the workbooks assigned in class). Points will be deducted if it is difficult to locate the question and response.
8. Make sure you comment your code. Points will be deducted if code logic is not apparent.
9. The written sections will be graded on correctness and preciseness while programming code will be graded on structure, implementation and correctness.

Q1: Multi-Layer Perceptron for Classification Dataset: You will use the UCI Optical Recognition of Handwritten Digits Dataset:

https://archive.ics.uci.edu/ml/datasets/Optical+Recognition+of+Handwritten+Digits

You should get this data set using Scikit-Learn (by using *sklearn.datasets.load_digits*):
https://scikit-learn.org/stable/modules/generated/sklearn.datasets.load_digits.html

General requirements:

Multi-Layer Perceptron classifier with a **single hidden layer** for performing both **binary and multi-class classification**. The model implements the *backpropagation* algorithm. To optimize the process of updating the weight matrices, it uses the Stochastic Gradient Descent (SGD) algorithm.

You can use **`sklearn.neural_network`.MLPClassifier as long as all the functionality required in the problem description exists.**

For 40 extra credit points, implement from scratch MLPClassifier as in previous assignments.

The MLPClassifier model takes the training data/feature matrix (X) and the training labels (Y) which is a 1D NumPy array with labels corresponding to each row of the feature matrix X.

If early stopping is set, then the model should split the training data randomly into a training subset and a validation subset.

The model should be able to decide whether it's a binary or multi-class classification problem by looking at Y.

- If there are only two unique class labels in Y, then it's a binary classification problem.
- If there are more than two unique class labels in Y, then it's a multi-class classification problem. For a multi-class classification problem, the model should transform Y into a matrix containing a one-hot vector for each instance. The number of rows is equal to the number of rows in Y. The number of columns is equal to the number of unique class indices/labels in Y (i.e., the number of classes). You may use the one-hot-vector function from the previous assignment.

  Output Layer Neuron & Activation function:

- In binary classification a single neuron should be used in the output layer; and a logistic sigmoid function should be used to compute the class probability. The threshold to determine the class label is 0.5.
- For multi-class classification the number of neurons in the output layer should be equal to the number of classes; and a softmax function should be used for computing the class probabilities.

The model uses backpropagation algorithm to learn weights of two matrices:

- Input-to-hidden layer weight matrix
- Hidden layer-to-output layer weight matrix

You should use the Stochastic Gradient Descent (SGD) algorithm to optimize the process of updating the two weight matrices. Note that the weight matrices should be initialized with small random real numbers (not zeros).

The model should ***add a bias neuron*** to both input layer and hidden layer.

In SGD, during each epoch the model should iterate equivalent to the number of training data points. For example, if there are 'm' number of training instances, then during each epoch the model should iterate 'm' times. In each iteration (within an epoch) it should select one training data point **randomly** for forward propagation and backward propagation to update the weight matrices. During each iteration the model should compute the training loss (cross-entropy or regularized cross-entropy) and stores it into an array. Then, at the completion of an epoch, it computes the average training loss for that epoch. It stores the average training loss in another array, which will be used later to plot the training loss vs epochs graph. The total number of epochs is set by the model hyperparameter (max_iter).

If **early stopping** is set, then at the end of each epoch (after completing 'm' iterations), the model should compute the validation loss and validation score by using the **entire validation subset**. These values should be stored in two arrays, which will be used later to plot the following two graphs: validation loss vs epochs, and validation score vs epochs.

Type of Loss Function:
The loss function for classification should be cross-entropy. Note that the cross-entropy loss function for binary classification and multi-class classification is different. Depending on the type of classification (binary or multi-class), your model should be able to use the appropriate cross-entropy function for computing the loss.

Implement the following functions for the MLPClassifier model class. Implement additional functions as needed.

1. Implement the following function that creates a weight matrix and initializes it with small random real numbers. **[4 pts]**
initializeTheta(in, out):
Arguments:
in : int number of output neurons/features.
out : int number of output neurons/features.
Returns:
Theta : ndarray The weight matrix initialized by small random numbers.

2. Implement the logistic sigmoid activation function. **[2 pts]**

logistic(z)
Arguments:
z : ndarray
Returns:
An ndarray containing the logistic sigmoid values of the input.

3. Implement the ReLU (rectified linear unit) activation function**. [3 pts]**

relu(z)

Arguments:
z : ndarray
Returns:
An ndarray containing the relu output values of the input.

4. Implement the tanh (hyperbolic tangent) activation function. **[3 pts]**
tanh(z)
Arguments:
z : ndarray
Returns:
An ndarray containing the tanh output values of the input.

5. Implement a **MLPClassifier** model class. It has a single hidden layer. It should have the following three methods. The model uses the *backpropagation* algorithm for learning the weights of the features/neurons. Note the that "fit" method should implement the **Stochastic Gradient Descent** algorithm for optimizing the weight update process. **[40 pts]**

a)

fit(self, X, Y, hidden_layer_neurons=2, activation= 'logistic', regularizer=None, alpha=0.0001, learning_rate='constant', learning_rate_init=0.001, max_iter=1000, tol = 0.0001, verbose=False, early_stopping=False, validation_fraction=0.1, n_iter_no_change=10,**kwargs):

Arguments:
X : ndarray A numpy array with rows representing data samples and columns representing features.

Y : ndarray A 1D numpy array with labels corresponding to each row of the feature matrix X.

hidden_layer_neurons : int It provides the number of neurons in the hidden layer (*excluding the bias unit*).

activation : string (default 'logistic') Activation function for the hidden layer.
- 'logistic', the logistic sigmoid function, returns f(x) = 1 / (1 + exp(-x)).
-'tanh', the hyperbolic tan function, returns f(x) = tanh(x).
-'relu', the rectified linear unit function, returns f(x) = max(0, x)

regularizer : string The string value could be one of the following: l2, None.
If it's set to None, the cost function without the regularization term will be used for computing the gradient and updating the weight vector. However, if it's set to l2, the appropriate regularized cost function needs to be used for computing the gradient and updating the weight vector.

Note: you may define a helper function for computing the regularized cost using "l2" regularizer.

alpha : float It provides the regularization coefficient. It is used only when the "regularizer" is set to l2.

learning_rate : string (default 'constant') Learning rate schedule for weight updates.
-'constant': a constant learning rate given by 'learning_rate_init'.
-'adaptive': gradually decreases the learning rate based on a schedule or some heuristic.
Write a function that would be used if learning_rate is set to 'adaptive'. [**Extra Credit 10 pts**]

Note on defining the "adaptive" learning_rate function:

There are different heuristics to write an adapting learning rate function. You are encouraged to explore existing functions in the literature, or try a new one. Here are some suggestions for you.

- Decrease learning rate gradually during each epoch:
  Learning rate = $\frac{a}{epoch+b}$ where a and b are two constants that you need to determine empirically. Choose constant *a* and *b* such that initially the learning rate is large enough.
- Divide the initial learning rate by some constant: This will require validation data (you need to implement 'early stopping'). The function keeps the learning rate constant to 'learning_rate_init' as long as validation loss keeps decreasing. Each time two consecutive epochs, within 'n_iter_no_change' number of epochs, fail to decrease validation loss by at least tol, or fail to increase validation score by at least tol, the current learning rate is divided by a constant (e.g., 2).

learning_rate_init : double The initial learning rate used. It controls the step-size in updating the weights.

max_iter : int Maximum number of iterations. The solver iterates until convergence (determined by 'tol') or this number of iterations. For Stochastic Gradient Descent, note that this determines the number of epochs (how many times each data point will be used), not the number of gradient steps.

tol : float Tolerance for the optimization. When the loss or score is not improving by at least tol for n_iter_no_change consecutive iterations, unless learning_rate is set to 'adaptive', convergence is considered to be reached and training stops.

verbose : bool Whether to print the following progress messages at each epoch to stdout.
-Epoch number, Training loss, validation loss, validation score, and step size (eta).
-Note that the validation loss and validation score is reported if early_stopping is set to True.

early_stopping : bool Whether to use early stopping to terminate training when **validation score** is not improving. If set to true, it will automatically set aside a percentage of training data (set by validation_fraction) as validation and terminate training when validation score is not improving by at least *tol* for *n_iter_no_change* consecutive epochs. If the program terminates early, then it should display the following message: *Early Stopping because the validation score change between two consecutive epochs is less than (value of "tol") over the last (value of "n_iter_no_change") iterations.*

validation_fraction : float The proportion of training data to set aside as validation set for

early stopping. Must be between 0 and 1. Only used if early_stopping is True.

n_iter_no_change : int Maximum number of epochs to not meet tol improvement.
Note: the "fit" method should use two weight matrices "Theta1" and "Theta2" that contain the parameters for the model (weights and bias terms) for the input layer and the hidden layer, respectively.
The "Theta1" should be a matrix with dimension: *no. of features/neurons (including bias) in the input layer* x *no. of features/neurons (excluding bias) in the hidden layer*
The "Theta2" should be a matrix with dimension: *no. of features/neurons (including bias) in the hidden layer* x *no. of features/neurons in the output layer*

-For a Binary Classification problem, number of neurons in the output layers should be one.

-For a Multi-class Classification problem, the number of output neurons is the number of classes in the dataset.

Finally, it should update the model parameters "Theta1" and "Theta2" to be used in "predict" method as follows.
self.Theta1 = Theta1
self.Theta2 = Theta2

b)

predict(self, X)

Arguments:
X : ndarray

A numpy array containing samples to be used for prediction. Its rows represent data samples and columns represent features.

Returns:
1D array of predicted class labels for each row in X.

Note: the "predict" method uses the **self.Theta1** and **self.Theta2** to make predictions.

c) __init__(self)

It's a standard python initialization function so we can instantiate the class. Just "pass" this.

Multi-Class Classification using MLPClassifier

6. Read the handwritten digits dataset using the *sklearn.datasets.load_digits* function for performing **multi-class classification**.
7. Standardize the features. **[1 pts]**
8. Partition the data into train and test set. **[2 pts]**

9. **You don't need to report hyperparameter tuning.** Note that unlike previous assignments, hyperparameter tuning is time-consuming for the MLP model. You may want to perform an educated tuning of the hyperparameters.

You need to report the optimal values of the hyperparameters that you used for training. For hyperparameter tuning, the following parameters should have following fixed setting. **[16 pts]**

regularizer="l2"
verbose=True
early_stopping=True
validation_fraction=0.1

Find the optimal values for the following hyperparameters.

hidden_layer_neurons
activation
alpha
learning_rate
learning_rate_init
max_iter
tol n_iter_no_change

10. Your jupyter notebook should display the following items. You will **not get any credit** if your jupyter notebook doesn't have these items displayed during your submission. Additionally, submit a PDF file containing the following items. **[2 + 6 + 10 + 10 = 28 pts]**

a) Epoch number, Training loss, validation loss, validation score, and step size (eta). This should be displayed as a single row, as follows. There should be max_iter number of rows, one for each epoch.

Example:

```
Epoch   1 of 2000: Training Loss = 3.39896 | Validation Loss = 3.41845 | Validation score = 0.000530 | Eta = 0.13405
Epoch   2 of 2000: Training Loss = 3.38801 | Validation Loss = 3.37391 | Validation score = 0.000530 | Eta = 0.13396
Epoch   3 of 2000: Training Loss = 3.32568 | Validation Loss = 3.33608 | Validation score = 0.000530 | Eta = 0.13387
Epoch   4 of 2000: Training Loss = 3.33486 | Validation Loss = 3.29821 | Validation score = 0.000530 | Eta = 0.13378
```

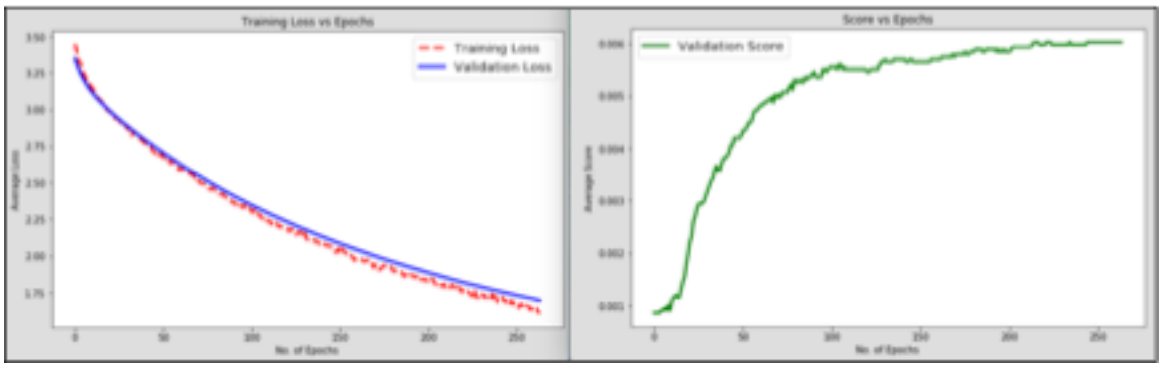
If the program terminates early, then it should display the following message:

Early stopping because the validation score change between two consecutive epochs is less than (value of "tol") over the last (value of "n_iter_no_change") epochs.

b) Two graphs:

- First graph plots both training loss and validation loss against epochs.
- Second graph plots validation score vs epochs.

Example:



c)For training data: accuracy, no. of correct predictions, confusion matrix, precision, recall, f1 score for each class.
d) For test data: accuracy, no. of correct predictions, confusion matrix, precision, recall, f1 score for each class.

Example:

```
Test Accuracy:  0.9

Test - No. of correct predictions: 324/360

Test - Confusion Matrix:
 [[37  0  0  0  1  0  0  0  0  0]
 [ 0 29  1  1  0  0  1  0  1  3]
 [ 0  0 29  2  0  0  0  0  0  0]
 [ 0  0  0 28  0  0  0  0  0  0]
 [ 0  0  0  0 34  0  0  0  0  0]
 [ 0  0  0  0  0 37  1  1  0  3]
 [ 0  1  0  0  0  0 40  0  0  0]
 [ 0  0  0  0  2  0  0 36  1  0]
 [ 0  1  0  1  0  1  0  1 27  3]
 [ 1  0  0  2  0  2  0  4  1 27]]

Test - Classification Report:
              precision    recall  f1-score   support

           0       0.97      0.97      0.97        38
           1       0.94      0.81      0.87        36
           2       0.97      0.94      0.95        31
           3       0.82      1.00      0.90        28
           4       0.92      1.00      0.96        34
           5       0.93      0.88      0.90        42
           6       0.95      0.98      0.96        41
           7       0.86      0.92      0.89        39
           8       0.90      0.79      0.84        34
           9       0.75      0.73      0.74        37

    accuracy                           0.90       360
   macro avg       0.90      0.90      0.90       360
weighted avg       0.90      0.90      0.90       360
```

Note that if your test accuracy is less than 90% you will lose 10% of the total obtained points. If your test accuracy is less than 80% you will lose 30% of the total obtained points.