COL 774: Assignment 3 (Part A and B)
Sem II, 2019-20

[Part A] Due Date: Mar 24, 11:50 pm [Deadline is over now for part A!]. Total Points: 30

[Part B] Due Date: Apr 24, 11:50 pm. Total Points: 30 (Part B)

Notes:

- This assignment has two implementation questions.
- You should submit all your code (including any pre-processing scripts written by you) and any graphs that you might plot.
- Do not submit the datasets. Do not submit any code that we have provided to you for processing.
- Include **a write-up (pdf) file**, one (consolidated) for each part, which includes a brief description for each question explaining what you did. Include any observations and/or plots required by the question in this single write-up file (one for each of the parts A and B).
- You should use Python as your programming language. For Decision tree question (only), we may allow using C/C++/Java - but you need to confirm with us first.
- Your code should have appropriate documentation for readability.
- You will be graded based on what you have submitted as well as your ability to explain your code.
- Refer to the course website for assignment submission instructions.
- This assignment is supposed to be done individually. You should carry out all the implementation by yourself.
- We plan to run Moss on the submissions. We will also include submissions from previous years since some of the questions may be repeated. Any cheating will result in a zero on the assignment, a penalty of -10 points and possibly much stricter penalties (including a **fail grade** and/or a **DISCO**).

1. **(30 points) Decision Trees (and Random Forests):**
   Machine learning has been deployed in various domains of computer science as well as other engineering fields. In this problem you will work on detecting files infected with virus on your system. You will work with VirusShare dataset available for download from the UCI repository. Read about the dataset in detail from the link given above. You have been provided with a pre-defined set of test, train and validation of dataset to work with (available for download from the course website). In this dataset, for any given example, a large number attribute values are missing (they can be thought of having as a 'default' value,*i.e.* 0). Correspondingly, you have also been provided sparse files and you can use **pyxclib** for reading and writing the sparse files. You have to implement the decision tree algorithm for predicting the virus infected files based on a variety of features. You will also experiment with Random Forests in the last part of this problem.

   (a) **(10 points) Decision Tree Construction** Construct a decision tree using the given data to predict which files are infected. You should use mutual information as the criteria for selecting the attribute to split on. At each node, you should select the attribute which results in maximum decrease in the entropy of the class variable (i.e. has the highest mutual information with respect to the class variable). This problem has all its attributes as integer (continuous) valued. For handling continuous

attributes, you should use the following procedure. At any given internal node of the tree, a numerical attribute is considered for a two way split by calculating the median attribute value from the data instances coming to that node, and then computing the information gain if the data was split based on whether the numerical value of the attribute is greater than the median or not. For example, if you have have 10 instances coming to a node, with values of an attribute being (0,0,0,1,1,2,2,2,3,4) in the 10 instances, then we will split on value 1 of the attribute (median). Note that in this setting, a numerical attribute can be considered for splitting multiple number of times. At any step, choose the attribute which results in highest mutual information by splitting on its median value as described above. Note that a large number of attribute values are missing for any given instance, and for this problem, it is safe to treat them as having a default value of '0' [1]. Plot the train, validation and test set accuracies against the number of nodes in the tree as you grow the tree. On X-axis you should plot the number of nodes in the tree and Y-axis should represent the accuracy. Comment on your observations.

(b) **(6 points) Decision Tree Post Pruning** One of the ways to reduce overfitting in decision trees is to grow the tree fully and then use post-pruning based on a validation set. In post-pruning, we greedily prune the nodes of the tree (and sub-tree below them) by iteratively picking a node to prune so that resultant tree gives maximum increase in accuracy on the validation set. In other words, among all the nodes in the tree, we prune the node such that pruning it(and sub-tree below it) results in maximum increase in accuracy over the validation set. This is repeated until any further pruning leads to decrease in accuracy over the validation set. Read the following notes on pruning decision trees to avoid overfitting (also available from the course website). Post prune the tree obtained in step (a) above using the validation set. Again plot the training, validation and test set accuracies against the number of nodes in the tree as you successively prune the tree. Comment on your findings.

(c) **(10 points) Random Forests:** As discussed in class, Random Forests are extensions are decision trees, where we grow multiple decision trees in parallel on bootstrapped samples constructed from the original training data. A number of libraries are available for learning Random Forests over a given training data. In this particular question you will use the scikit-learn library of Python to grow a Random Forest. Click here to read the documentation and the details of various parameter options. Try growing different forests by playing around with various parameter values. Especially, you should experiment with the following parameter values (in the given range): (a) *n_estimators* (50 to 450 in range of 100). (b) *max_features* (0.1 to 1.0 in range of 0.2) (c) *min_samples_split* (2 to 10 in range of 2). You are free to try out non-default settings of other parameters too. Use the out-of-bag accuracy (as explained in the class) to tune to the optimal values for these parameters. You should perform a grid search over the space of parameters (read the description at the link provided for performing grid search). Report training, out-of-bag, validation and test set accuracies for the optimal set of parameters obtained. How do your numbers, i.e., train, validation and test set accuracies compare with those you obtained in part (b) above (obtained after pruning)?

(d) **(4 points) Random Forests - Parameter Sensitivity Analysis:** Once you obtain the optimal set of parameters for Random Forests (part (c) above), vary one of the parameters (in a range) while fixing others to their optimum. Plot the validation and test Repeat this for each of the parameters considered above. What do you observe? How sensitive is the model to the value of each parameter? Comment.

(e) **Extra Fun: No Credits!**: Read about the XG-boost algorithm which is an extension of decision trees to Gradient Boosted Trees. You can read about gradient boosted trees here (link 1) and here (link 2). Try out using XG-boost on the above dataset. Try out different parameter settings, and find the one which does best on the validation set. Report the corresponding test accuracies. How do these compare with those reported for Random Forests?

[1] think about why this makes sense. Are there any other ways that you can deal with these missing attribute values? Feel free to try alternate methods

2. **Neural Networks (30 points):**

In this problem, you will work with the English Alphabets Dataset available for download from the course website. This is an image dataset consisting of greyscale pixel values of 28×28 size images of 26 Alphabets. Therefore this is a multiclass dataset containing 26 classes. The Dataset contains a train and a test set consisting of 13000 and 6500 examples, respectively. The last entry in each row denotes the class label. Note that the dataset is balanced; i.e. the number of examples is the same for all classes in train and test set. In this problem, we will use what is referred to a one-hot encoding of the output labels. Given a total of r classes, the output is represented an r sized binary vector (i.e., $y \in \mathcal{R}^{r \times 1}$), such that each component represents a Boolean variable, i.e., $y_l \in \{0, 1\}, \forall l, 1 \leq l \leq r$). In other words, each y vector will have exactly one entry as being 1 which corresponds to the actual class label and all others will be 0. This is one of the standard ways to represent discrete data in vector form [2]. Corresponding to each output label y_l, our network will produce (independently) an output label o_l where $o_l \in [0, 1]$.

(a) **(10 points)** Write a program to implement a generic neural network architecture to learn a model for multi-class classification using one-hot encoding as described above. You will implement the backpropagation algorithm (from first principles) to train your network. You should use mini-batch Stochastic Gradient Descent (mini-batch SGD) algorithm to train your network. Use the Mean Squared Error(MSE) over each mini-batch as your loss function. Given a total of m examples, and M samples in each batch, the loss corresponding to batch # b can be described as:

$$J^b(\theta) = \frac{1}{2M} \sum_{i=(b-1)M}^{bM} \sum_{l=1}^{r} (y_l^{(i)} - o_l^{(i)})^2 \tag{1}$$

Here each $y^{(i)}$ is represented using one-hot encoding as described above. You will use the sigmoid as activation function for the units in **output** layer as well as in the hidden layer (we will experiment with other activation units in one of the parts below). Your implementation(including back-propagation) MUST be from first principles and not using any pre-existing library in Python for the same. It should be generic enough to create an architecture based on the following input parameters:

- Mini-Batch Size (M)
- Number of features/attributes (n)
- Hidden layer architecture: List of numbers denoting the number of perceptrons in the corresponding hidden layer. Eg. a list [100 50] specifies two hidden layers; first one with 100 units and second one with 50 units.
- Number of target classes (r)

Assume a fully connected architecture i.e., each unit in a hidden layer is connected to every unit in the next layer.

(b) **(5 points)** Use the above implementation to experiment with a neural network having a **single** hidden layer. Vary the number of hidden layer units from the set {1, 5, 10, 50, 100}. Set the learning rate to 0.1. Use a mini-batch size of 100 examples. This will remain constant for the remaining experiments in the parts below. Choose a suitable stopping criterion and report it. Report and plot the accuracy on the training and the test sets, time taken to train the network. Plot the metric on the Y axis against the number of hidden layer units on the X axis. What do you observe? How do the above metrics change with the number of hidden layer units? NOTE: For accuracy computation, the inferred class label is simply the label having the highest probability as output by the network.

(c) **(5 points)** Use an adaptive learning rate inversely proportional to number of epochs i.e. $\eta_e = \frac{\eta_0}{\sqrt{e}}$ where $\eta_0 = 0.5$ is the seed value and e is the current epoch number[3]. See if you need to change your stopping criteria. Report your stopping criterion. As before, plot the train/test set accuracy, as well as training time, for each of the number of hidden layers as used in 2b above using this new adaptive learning rate. How do your results compare with those obtained in the part above? Does the adaptive learning rate make training any faster? Comment on your observations.

(d) **(5 points)** Several activation units other than sigmoid have been proposed in the literature such as tanh, and RelU to introduce non linearity into the network. ReLU is defined using the function: g(z) = max(0, z). In this part, we will replace the sigmoid activation units by the ReLU for all the units

[2] Feel free to read about the one-hot encoding of discrete variables online

[3] One epoch corresponds to one complete pass through the data

in the hidden layers of the network (the activation for units in the output layer will still be sigmoid to make sure the output is in the range (0,1)). You can read about relative advantage of using the ReLU over several other activation units on this blog.

Change your code to work with the ReLU activation unit. Note that there is a small issue with ReLU that it non-differentiable at $z = 0$. This can be resolved by making the use of sub-gradients - intuitively, sub-gradient allows us to make use of any value between the left and right derivative at the point of non-differentiability to perform gradient descent see this (Wikipedia page for more details). Implement a network with 2 hidden layers with 100 units each. Experiment with both ReLU and sigmoid activation units as described above. Use the adaptive learning rate as described in part2c above. Report your training and test set accuracies in each case. Also, make a relative comparison of test set accuracies obtained using the two kinds of units. What do you observe? Which ones performs better? Also, how do these results compare with results in part 2b using a single hidden layer with sigmoid. Comment on your observations.

(e) **(5 points)** Use MLPClassifier from scikit-learn library to implement a neural network with the same architecture as in Part 2d above, and same kind of activation functions (ReLU for hidden layers, sigmoid for output). Use Stochastic Gradient Descent as the solver. Note that MLPClassifier only allows for Cross Entropy Loss over the final network output. Use the binary cross entropy loss for the multi-class classification problem. Here, we have a binary prediction for each possible output label, and apply a cross entropy loss over each such prediction. Note that in this formulation, there are r such outputs of the network, one for each label, and cross entropy loss is applied for each of them (and then added to compute the final loss). Read about the binary cross entropy loss here[4] (you should refer to the two class entropy loss formulation only, since we have transformed our multi-class problem into r two class (problems), one for each label). Compare the performance with the results of Part 2d. How does training using existing library (and modified loss function) compare with your results in Part 2d above.

(f) **(Extra Fun, No Credits!)** Modify your loss function in Part 2d above (your own implementation) to be binary cross entropy loss as in Part 2e above. How do your results change? What is the impact of using a different loss function in your implementation? Also compare with results obtained in Part 2e. After this change, the only difference remaining is between your own implementation and the library implementation of the learning algorithm (since the network architecture as well as the loss functions are now the same). Does prediction accuracy (train/test) of your implementation compare with the library implementation? Why or why not? What could the possible reasons for differing accuracy numbers?

[4] you can also refer to the lecture notes from April 10 - section on GANs to know more about cross entropy loss