COL 774: Assignment 3
Semester I, 2022-2023

Due Date: Wednesday October 26 (2022), 11:50 pm. Total Points: 47+ 58

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.
- 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.
- 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 Piazza 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, an additional penalty equivalent of the weight of the assignment and possibly much stricter penalties (including a **fail grade** and/or a **DISCO**).

1. **(47 points) Decision Trees, Random Forests and Gradient Boosted Trees:**
   In this problem, we will work with two different datasets.

   Dataset 1:
   Machine learning has been deployed in various domains of computer science as well as other engineering fields. In this problem you will work on predicting the severity (benign or malignant) of a mammographic mass lesion. Mammography is the most effective method for breast cancer screening available today. However, each mammogram interpretation may require significant time from a trained medical personnel, and that is where Machine Learning can be of help. Our objective in this problem is to help physicians in deciding whether to perform a breast biopsy on a suspicious lesion seen in a mammogram or to perform a short-term follow-up examination instead. We will work on a mammography dataset available from the UCI repository, and you can read more about this dataset from this link. We will use a processed version of this dataset for this assignment available here. The dataset contains 5 features: (a) BI-RADS assessment (b) Age (c) Shape (d) Margin (e) Density. The value of the first is based on a human assessment of the mammographic mass for additional analysis, and therefore you **should not** use this in learning your models below. Using the four features: Age, Shape, Margin and Density, you need to grow a decision tree/random forest/gradient boosted trees to predict the severity of a mammographic mass lesion. Note that the target variable as the last (sixth) column in the csv files provided to you. We have split the dataset into train, validation, and test set.

(a) **(2 points) Decision Tree Construction and Visualization** There are several missing values in the dataset splits. One way to handle missing values is to ignore those examples. Drop the samples with any missing attribute and train a decision tree on the training split and report training accuracy, validation accuracy, and test accuracy. Visualize the decision tree. You should use the default parameter values while defining the tree in this part.

(b) **(4 points) Decision Tree Grid Search** Next, perform a grid search over the space of parameters including `max_depth`, `min_samples_split` and `min_samples_leaf`. You can also try to vary any other parameters that you may find relevant. Report training, validation, and test set accuracies for the optimal set of parameters obtained. Visualize the optimal decision tree. Comment on your observations. Specifically, you should compare the tree with that obtained in the part (a) above.

(c) **(3 points) Decision Tree Post Pruning (Cost Complexity Pruning)** In the previous part, we have seen that there are various parameters to prevent Decision tree classifiers from overfitting. Cost complexity pruning provides another option to control the size of a tree, and is a variation of the idea of reduced error pruning discussed in the class. Minimal cost complexity pruning recursively finds the node with the "weakest link". The weakest link is characterized by an effective alpha, and the nodes with the smallest effective alpha are pruned first. In this part we study the effect of `ccp_alpha`(a parameter in the scikit-learn implementation of decision tree) on regularizing the trees based on its accuracy on the validation set. To get an idea of what values of `ccp_alpha` could be appropriate, scikit-learn provides `DecisionTreeClassifier.cost_complexity_pruning_path` that returns the effective alphas and the corresponding total leaf impurities at each step of the pruning process. Use the training split and plot total impurity of leaves vs effective alphas of pruned tree. Plot the number of nodes vs alpha and the depth of tree vs alpha. Plot training accuracy, validation accuracy, and test accuracy vs alpha. What are your observations? Use the validation split to determine the best performing tree and report the train, validation, and test accuracy with respect to the best tree. Visualize the best pruned tree. Comment on your observations. Specifically, you should compare the tree with that obtained in the part (a) and (b) above.

(d) **(4 points) Random Forests** As discussed in class, Random Forests are extensions of 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 : (a) `n_estimators` (b) `max_features` (c) `min_samples_split`. 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.

(e) **(4 points) Missing Data Imputation** Another way to handle missing value in a dataset is called imputation, the process of replacing missing data with substituted values. The simplest way is to replace missing values of missing attributes in an example with some aggregate function of the attribute's value in other examples where it is present. This is done as a pre-processing step before the tree (forest) is grown. Two simplest aggregate functions to try are median and mode. Try these two aggregate functions for imputation (a) - (d). Note that you may need to perfrom the grid search again to obtain the best set of parameters. Compare the results with those obtained by simply ignoring any examples with missing values. What do you see? Comment on your observations.

(f) **(4 points) Gradient Boosted Trees:** XGBoost (Extreme Gradient Boosting) is functional gradient boosting based approach where an ensemble of "weak learners" (decision trees in our case) is used with the goal to construct a model with less bias, and better predictive performance as discussed in the class. You can read about the XGBoost implementation here. Implement an XGBoost classifier and experimenting with different parameter values (in the given range): (a) `n_estimators` (10 to 50 in range of 10) (b) `subsample` (0.1 to 0.6 in range of 0.1) (c) `max_depth`(4 to 10 in range of 1). You should perform a grid search over the space of parameters (read the description at the link provided for performing grid search). Report training, validation and test set accuracies for the optimal set of parameters obtained. Note that XGBoost by itself takes care of missing values, so you do not have to worry about ignoring any examples with missing values, or performing imputation as a pre-processing step.

Dataset 2: In this problem, we will work another dataset related to prediction of rating associated medical drugs, where the attributes are primarily textual. Any dataset with text attributes requires conversion of text to numerical attributes for computations to be performed. One such dataset is provided here. In this dataset there are two text attibutes, namely **conditon** and **review**. The review attribute contains reviews about drugs collected from patients with specified conditions. We are going to use these two text attributes to predict the rating of a drug. In order to make the specified text attributes as numerical attributes, two well known techniques are CountVectorizer and TF-IDFVectorizer. Use (any one of) these techniques to convert text attibutes to numerical attributes. In the provided dataset, the text attributes (particularly reviews) have a lot of stopwords (considered insignificant like 'a', 'an', 'the'). Remove these using information provided here. Also remove unnecessary characters (commas, full stop) and backslash character constants (carriage-return, linefeed). In case a review is missing, consider the review to be an empty string. The date attribute must be split into three numerical attributes (day, month and year).

(a) **(2 points) Decision Tree Construction:** Repeat part(a) above for the Dataset 1. No need to visualise in this case as the number of attributes are really high.

(b) **(4 points) Decision Tree Grid Search:** Repeat part(b) above for the Dataset 1. No need to visualise in this case as the number of attributes are really high.

(c) **(3 points) Decision Tree Post Pruning:** Repeat part(c) above for the Dataset 1. No need to visualise in this case as the number of attributes are really high.

(d) **(4 points) Random Forests:** Repeat part(d) above for the Dataset 1. You should experiment with the following parameter values : (a) `n_estimators` (50 to 450 in range of 50) (b) `max_features`(0.4 to 0.8 in range of 0.1) (c) `min_samples_split (2 to 10 in range of 2)`.

(e) **(4 points) Gradient Boosted Trees (XGBoost):** Repeat part(f) above for the Dataset 1. You should experiment with the following parameter values : (a) `n_estimators` (50 to 450 in range of 50) (b) `subsample` (0.4 to 0.8 in range of 0.1) (c) `max_depth`(40 to 70 in range of 10).

(f) **(4 points) GBM (Gradient Boosted Machines):** LightGBM is a gradient boosting framework that implements another boosting framework, Gradient Boosted Machines. One of the primary advantages of GBMs is its scalability to large datasets. Carefully read the following paper. Try various parameter settings for LightGBM on the above dataset, and see if you can find a setting which improves the performance compared to the earlier trained models. Compare the running times for standard decision tree learner (part (b)), decision tree with pruning (part (c)), random forest (part (c)), XGBoost (part(d)) and LightGBM (part (f)).

(g) **(5 points) Training with Varying amount of data** Randomly sample n number of examples from the training data, with $n \in \{20k, 40k, 60k, 80k, 100k, 120k, 140k, 160k\}$, and repeat the parts (a) - (g) above for each value of n. Note that validation and test sets remain the same. Plot the test set accuracy on y-axis and value of n on x-axis, with one curve for each of the algorithms tried above. Plot a similar curve for the amount of time taken for training on y-axis and n on x-axis. Comment on your observations.

2. **(58 points) Neural Networks :** In this problem, you will work with the Fashion MNIST dataset. The dataset consists of 28×28 gray-scale images belonging to Zalando's 10 article classes. The dataset has been provided here in .csv format. The first 784 columns of the data correspond to the pixel values. The last column is the target label. The data before training/testing needs to be normalised(by division with the highest pixel value). In this problem we will use what is referred to as one-hot encoding of the output labels. Given a total of r classes, the output is represented as a r-sized binary vector (*i.e.*, $y \in \mathbb{R}^{r \times 1}$) 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 one which corresponds to the actual class label and all others will be zero. This is one of the standard ways to represent discrete data in vector form. For the purpose of model development, convert the labels to one-hot encoding. Corresponding to each input, the network should produce a vector o which would have the o_m entry as 1 if the predicted label is m.

(a) **(20 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 number 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) **(6 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 {5, 10, 15, 20, 25}. 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. Additionally, report the confusion matrix for the test set, for each of the above parameter values. 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) **(6 points)** Use an adaptive learning rate inversely proportional to number of epochs i.e. $\eta_e = \frac{\eta_0}{\sqrt{e}}$ where $\eta_0 = 0.1$ is the seed value and e is the current epoch number[1]. 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. Also, report the confusion matrix over the test set in each case. 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) **(6 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 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 part 2c above. Report your training and test set accuracies in each case. Also, the report the confusion matrix over the test set. Make a relative comparison of test set accuracies (and confusion matrix) 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.

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

(e) **(8 points)** Varying the number of hidden layers increases the size of the set of hypothesis functions that neural network can approximate. Experiment by increasing the number of hidden layers in your network from 2 to 5. Keep the number of hidden units per layer to be 50. Plot the training and test accuracy against number of hidden layers to see the increase/decrease in performance. Do this plotting exercise for ReLU as well as sigmoid activation in the hidden layers. Make sure you keep sigmoids in the output layer. Comment on your observations. Experiment with the number of units in each layer and come up with the best architecture for the neural network. Note that the best architecture corresponds to the architecture that maximises the test accuracy.

(f) **(6 points)** In previous parts you have trained the network using MSE objective function. Another popular objective function that is typically used for training the network in Cross-Entropy based loss. Cross-Entropy loss is nothing but the loss corresponding to the negative of the log-likelihood of the data. For two class problems, it is referred to as Binary-Cross Entropy (BCE) loss. Note that since we are modeling our output as one-hot, using a cross-entropy based loss in our case, you will have a BCE based loss term for every possible class label. The corresponding log-likelihood terms capture the log-probability of whether that particular label is 1 or 0. You can read more about BCE here. Calculate the derivative for BCE with respect to output of each neuron in the last layer and mention it in the report. Now, make changes to the network to make it work with Binary Cross Entropy Loss. For the best architecture (i.e., No. of layers) obtained in part 2e) above, mention the training and test accuracy for BCE loss. For the hidden layers, use ReLu as the activation unit.

(g) **(6 points)** Use MLPClassifier from scikit-learn library to implement a neural network with the same architecture as in Part 2f above (use the No. of layers that gave the best performance), and same kind of activation functions; use ReLU for hidden layers and sigmoid for output. Use Stochastic Gradient Descent as the solver, and use BCE as your loss function. Note that MLPClassifier only allows for Cross Entropy Loss over the final network output. Compare the performance with the results of Part 2f. How does training using existing library (and modified loss function) compare with your results in Part 2f above? Comment.