COL 774: Assignment 3
Semester I, 2025-26

[Part A: Decision Trees]. Total Points: 48.
[Part B: Neural Networks]. Total Points: 40.
Due Date (for both parts): Friday Oct 31st, 11:59 pm.

Notes:

- This assignment has two implementation questions of Decision tree and Neural Network.
- Do not submit the datasets.
- Please upload code and a write-up (pdf) file as per the "Submission Guidelines" section below.
- You should use Python as your programming language.
- There is no starter code for the assignment. You are required to implement everything from scratch.
- You will be graded based on what you have submitted as well as your ability to explain your code.
- 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. **(48 points) Decision Trees (and Random Forests):**
   Imagine the tension of a live cricket final, watched by millions around the globe. The world's leading sports broadcaster wants to revolutionize its coverage by introducing a highly accurate, real-time "Win Predictor" graph on screen. To do this, they need an intelligent engine that can learn from thousands of past games.

   They've hired you as a specialist Machine Learning consultant for this project. Your task is to develop the core classification model that will power this new feature. Using a vast dataset of past white-ball matches, you will explore factors like which team batted first, who won the toss, the host, etc, to predict the final result. The accuracy of your model will directly impact the on-screen predictions seen by millions of fans, changing how they experience the game.

   We have provided a processed dataset from StatsGuru, split into three files for this assignment: `train.csv`, `validation.csv`, and `test.csv`. The link for dataset is here

   (a) **(15 points) Decision Tree Construction** Construct a decision tree using the given data to predict whether the result will be a win for the team or not. You will experiment with **mutual information** criteria. At each node, you should select the attribute that results in the maximum purity gain. For the entropy criterion, this means the split with the highest information gain (or mutual information). Remember that some attributes are categorical, while others are continuous. 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 were split based on whether the numerical value of the attribute is greater than the median or not. For example, if you

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 times.

For categorical attributes, you have to perform k way split where k is a number of unique attribute values. For example, if the following instance of training data arrives at a node ($United-States$, $England$, $India$, $Cuba$, $India$) then splitting on this attribute will result in 4 child nodes. While testing if an example coming at this node has a different attribute value (For eg: $Japan$) then consider the node as a leaf node and return the prediction.

Experiment with different maximum depths specifically for $\{5, 10, 15, 20\}$ and report the accuracy on train and test datasets. Feel free to experiment with other depths for performance gain. Plot the train and test set accuracies against the maximum depth in the tree. On the X-axis, you should plot the maximum depth in the tree, and the Y-axis should represent the train and test accuracies. Comment on your observations. Also, report the test accuracies on the result prediction. Comment on your observations.

(b) **(5 points) Decision Tree One Hot Encoding:** Categorical attributes having more than 2 categories can be encoded using one hot encoding. In one hot encoding, each category of attribute is represented by a new attribute which has a 0 or 1 value depending on the value of the original attribute. Replace the categorical attributes having more than 2 categories with respective one-hot encodings of the attributes in your dataset. For example, the attribute *team* has the following categories $\{india, england, australia\}$, then after replacement, the attribute *team* will be replaced with these 3 attributes $team\_india, team\_england$, and $team\_australia$ each taking a value of either 0 or 1. Repeat part (a) for this transformed dataset with maximum depths for $\{15, 25, 35, 45\}$. Compare the results with part (a) and comment on your observations. Feel free to experiment with other depths for performance gain. We will extend this version of the Decision Tree for part (c) below.

(c) **(8 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 the sub-tree below them) by iteratively picking a node to prune so that the resultant tree gives a 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 the sub-tree below it) results in a maximum increase in accuracy over the validation set. This is repeated until any further pruning leads to a decrease in accuracy over the validation set. Post-prune the trees of different maximum depths obtained in part (b) above using the validation set. Again for each tree 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.

(d) **(4 points) Pruning with Gini Impurity**: We have used Information Gain (based on entropy) as our criterion for building and pruning trees. However, another popular and highly effective criterion is **Gini Impurity**. In fact, Gini impurity is the default criterion for many popular libraries like scikit-learn due to its computational advantages. Unlike entropy, Gini impurity does not require a logarithm, which makes it slightly faster to compute. In very large datasets with many features, this speed difference can become significant. While both criteria often produce very similar trees, Gini impurity has a slight tendency to favor splits that isolate the single largest class in a node, whereas entropy tends to produce slightly more balanced trees. Exploring both allows you to see if one has an advantage for our specific dataset. Repeat part (c) by using the criterion as Gini impurity. Share the same plot comparison and findings.

(e) **(8 points) Decision Tree sci-kit learn:** As there are a number of libraries which provide Decision tree implementation, we would like to compare our own implementation against these. Use the sci-kit learn library to build the decision tree on the dataset provided to you. Note the value of the *criterion* parameter. Set it to **entropy** for comparing. Next, (i) Vary max_depth in the range $\{15, 25, 35, 45\}$. Keep all other arguments default. Report the train and test accuracies and plot the accuracies against maximum depth. Use the validation set to determine the best value of the depth parameter for your final model. (ii) Next, choose the default value of the depth parameter (which grows the tree fully), and vary the value of the pruning parameters ccp_alpha, in the range $\{0.0, 0.0001, 0.0003, 0.0005\}$. Use the default value of all other parameters. Again, plot train and test accuracies against the value of the pruning parameter. Choose the value of the ccp_alpha parameter that gives the maximum accuracy on the validation set, and use this as your final model. Now, compare the results obtained using the best model obtained in sub-parts (i) and (ii) above, i.e., with varying max_depth, and ccp_alpha parameters, with parts (b) and (c), and comment on your observations. Note: Sci-kit's internal implementation

uses a one-hot encoding to split multi-valued discrete attributes, so in some sense, the results here are comparable to your implementations in parts (b) and (c), respectively.

(f) **(8 points) Random Forests:** 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 sci-kit learn library of Python to grow a Random Forest. Click here to read the documentation and the details of various parameter options. Set *criterion* parameter of random forest to **entropy**. 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 350 in range of 100). (b) *max_features* (0.1 to 0.9 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. You should perform a grid search over the space of parameters to find the optimal combination (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, val, and test set accuracies, compare with parts (c) and (d) above?

(g) **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. 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?

Submission Guidelines

Directory Structure

All your files must be organized in the following directory structure. The root folder must be named with your entry number (e.g., `2024AIB2287`).

```
2024AIB2287
├── decision_tree.py (Contains your from-scratch DecisionTree class and all helper functions)
├── report.pdf
├── question_a/
│   └── a.py (handle the data loading and execution logic to run experiments for Part A)
├── question_b/
│   └── b.py (handle the data loading and execution logic to run experiments for Part B)
├── question_c/
│   └── c.py (handle the data loading and execution logic to run experiments for Part C)
├── question_d/
│   └── d.py (handle the data loading and execution logic to run experiments for Part D)
├── question_e/
│   └── e.py (handle the data loading and execution logic to run experiments for Part E)
└── question_f/
    └── f.py (handle the data loading and execution logic to run experiments for Part F)
```

Auto-evaluation guidelines

We will run the command below for execution.

```
command:
python {a/b/c/d/e/f}.py <train_data_path> <val_data_path> <test_data_path> <output_folder_path>
```

train_data_path: the absolute path of train data. It will be a CSV file.

val_data_path: the absolute path of validation data. It will be a CSV file.

test_data_path: the absolute path of test data. It will be a CSV file.

output_folder_path: the absolute path of the output file. It will be a CSV file.

it should have one column named **"result"**. Make sure to preserve the sequence of examples.

Share the prediction of the Test data in a CSV file for

(a) Best depth tree obtained by validation accuracy
(b) Best depth tree obtained by validation accuracy
(c) Pruned tree obtained after pruning the max depth 35 tree.
(d) Pruned tree obtained after pruning the max depth 35 tree.
(e) Best tree obtained by validation accuracy after hyperparameter tuning
(f) Best tree obtained by validation accuracy after hyperparameter tuning

2. **(40 points) Neural Networks:**

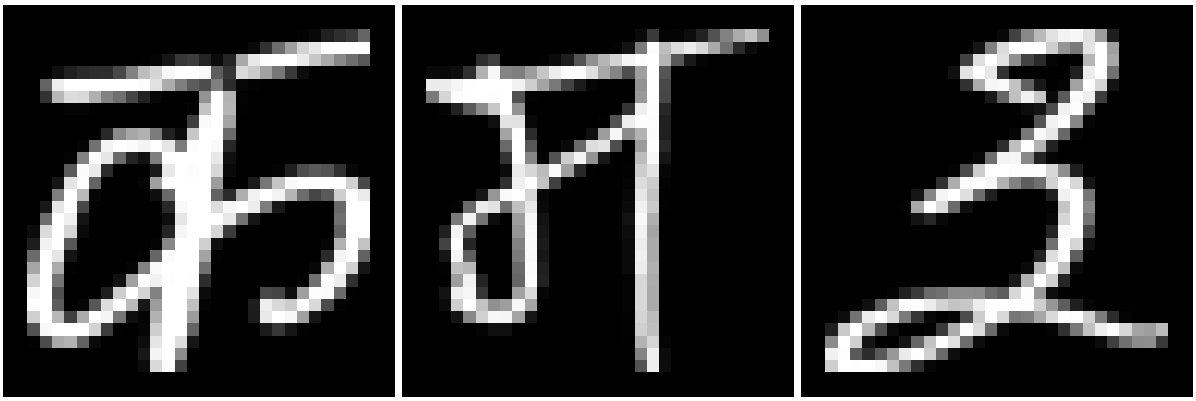

Figure 1: Sample data of three different characters/digits written in Devanagri

In this problem, we will work with the **Devanagari Character** dataset for classifying consonants and digits. The dataset provides a diverse set of handwritten Devanagari characters for training robust machine learning models (see Figure 1 for an illustration.). These images are sized to 32x32 grayscale images for both training and test data. The applications of this dataset include handwriting verification, digitization of archival records, real-time Optical Character Recognition systems, as well as educational and accessibility tools, where it is crucial to digitize handwritten scripts into machine-readable formats for preservation and accessibility.

We will first work with consonants subset of the data building a neural model over them, and then explore transfer learning for building a model for predicting digits in the last part of this question. The consonant subset of the dataset consists of 21,600 training images and 10,800 test images, spanning 36 different character classes. For accessing this dataset, click at this link.

For this task, we will use a Neural network as a prediction model. Instead of using the standard squared error loss over the logistic activation function in the final layer (as taught in class), we will instead use the cross-entropy loss defined over the softmax activation function. Softmax is a generalization of the logistic function for the case of multi-valued variables. Given a set of linear inputs $\{net_k^{(L)}\}_{k=1}^r$, applying softmax over it, results in a probability vector of size r, given as $o_1, o_2 \cdots o_r$, such that $\sum_k o_k = 1$, and where $o_k = \frac{e^{net_k^{(L)}}}{\sum_{k'} e^{net_{k'}^{(L)}}}$ Given two distributions P and Q, defined over a random variable y, the cross-entropy between them is defined via the expression: $\sum_k P(y=k) log Q(y=k)$. Then, in our case, the loss $J(\theta)$, is defined as the cross entropy between the true label distribution given as $\tilde{p}(y=k|x) = \mathbb{1}\{k=\tilde{k}\}$, where $\tilde{k}$, is the true label, and the predicted distribution $\hat{p}(y=k|x) = o_k$ (recall that o_k is simply the probability of outputting label k by the network). Then, the cross entropy loss is defined as:

$$J(\theta) = \sum_{k=1}^{r} -\mathbb{1}\{k=\tilde{k}\} log(o_k) \tag{1}$$

The partial derivative of $J(\theta)$, as defined above is given as:

$$\frac{\partial J(\theta)}{\partial net_k^{(L)}} = \begin{cases} (o_k - 1), & \text{if } k = \tilde{k} \\ o_k & \text{otherwise} \end{cases} \tag{2}$$

(a) **(12 points)** Write a program to implement a generic neural network architecture to learn a model for multi-class classification which outputs probability distribution by using softmax. 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 Cross Entropy Loss over each mini-batch as your loss function (mentioned above in 1). You will use the sigmoid as activation function for the units in the hidden layer (we will experiment with other activation units in one of the parts below) and softmax at output layer to get final predicted probability distribution. 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 0.01. Use a mini-batch size of 32 examples. This will remain constant for the remaining experiments in the parts below. To be specific you will have following arguments in the generic neural network:

- $M = 32$
- $n = 3072 \;\; (32 \times 32 \times 3)$
- Hidden layer sizes to be choosen from options
- $r = 36$

Choose a suitable stopping criterion and report it. Read about precision, recall and F (also known as F1) score here. Report the precision, recall and F1 score for each class at different hidden layer units on test data and train data. You could compute these metrics using scikit-learn utility. Plot the average F1 score vs the number of hidden units. Comment your findings.

(c) **(5 points)** In this part we will experiment with the depth of neural network. Vary the hidden layer sizes as $\{\{512\}, \{512, 256\}, \{512, 256, 128\}, \{512, 256, 128, 64\}\}$[1] Keep learning rate and batch size same as part b. Use same stopping criteria as before and report the precision, recall and F1 score for all variations on test data and train data. Plot the average F1 score vs the network depth. Report 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. 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 in the hidden layers. The activations in the final layer still stay softmax. Note that there is a small issue with ReLU that it is 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). Repeat the part c with ReLU activation. Report your training and test set precision, recall and F1 score. Plot the average F1 socre vs network depth. Also, make a relative comparison of test set accuracies obtained in part c. What do you observe? Which ones performs better?

[1] Models not training properly? Theres's a phenomenon at play here, called vanishing gradient problem.Read more about it here

(e) **(5 points)** Use MLPClassifier from scikit-learn library to implement a neural network with the same architectures as in part e above. Use Stochastic Gradient Descent as the solver. Note that MLPClassifier only allows for Cross Entropy Loss over the final network output. Set the following parameters:

- hidden_layer_sizes: to be vary according to part c.
- activation: relu
- solver: sgd
- alpha: 0
- batch_size: 32
- learning_rate: constant

Keep all other parameter as default. You need to decide the stopping criteria accordingly. Now report the same metrics and plots as of part d. Compare the results with part d, and comment on your observations.

(f) **(7 points)** Transfer learning is a machine learning technique where a model trained on one dataset is adapted to another related dataset. Instead of training a model from scratch, it leverages prior knowledge learned from a large dataset to improve performance on a smaller one. For this part, we will use this subset of the data corresponding to the digits written in Devanagri. Note that in the consonant subset of the dataset provided earlier, classes varied from 1-36 corresponding to consonant characters, and in the current subset of the dataset (i.e., the one provided in this part), classes range from 37-46 corresponding to the 10 digits. We'll explore transfer learning by adapting the model trained on consonant dataset to classify digits by replacing the last layer of the network. We'll first train a network on digits from scratch and compare it against our transfer learning finetuned model, details are given below:

- **Training from Scratch on Digits Data**: Implement a 4-hidden-layer multi-layered-perceptron (MLP) architecture with ReLU activations (as done in part d)) and hidden sizes [512, 256, 128, 64]. Since our goal is to classify digits subset of the data, the last (output) layer will have 10 neurons. Train this for 20 epochs on the Digits subset provided in the data. Plot the train/test F1-scores across epochs.
- **Transfer Learning from Consonants to Digits**: Now, instead of training from scratch, we'll directly adapt the trained Consonant model with [512, 256, 128, 64] hidden layers and ReLU activation units for digit classification by replacing the final (output) layer in the Consonant model with a new output layer having 10 neurons (i.e., for predicting 10 classes). Specifically, we initialize the weights in this model with the Consonant model's weights (trained in part (d) with ReLU activation units) except for the output layer whose weights are initialized as done when training from scratch. Fine-tune [2] the entire model on the Digits subset of the data for 20 epochs, and plot the train/test F1-scores across epochs over the test set.

Compare the performance of the fine-tuned model with the one trained on digits from scratch (by plotting them on the same graph) and report your observations.

Submission Guidelines

Directory Structure

All your files must be organized in the following directory structure. The root folder must be named with your entry number (e.g., `2024AIB2287`).

```
2024AIB2287
├── report.pdf
└── neural_network/
    ├── neural_network.py (Contains the Neural Network class and all its helper functions)
    ├── b.py (implementation for Part B)
    ├── c.py (implementation for Part C)
    ├── d.py (implementation for Part D)
    └── e.py (implementation for Part E)
```

[2] Fine-tuning refers to process of training a network whose weights are initialized using weights trained on another dataset.

```
├── f.py (implementation for Part F)
└── outputs/
```

Auto-evaluation guidelines

The following command will be used for execution:

```
command:
python {b/c/d/e/f}.py <train_data_path> <test_data_path> <output_folder_path>
```

train_data_path : the absolute path of train data folder.

test_data_path : the absolute path of test data folder.

output_folder_path : the absolute path of the output folder. Here, save the test dataset predictions as prediction_{question_number}.csv. Eg. prediction_b.csv. The file should have one column named "prediction". Make sure to preserve the sequence of examples.