DECISION TREES AND NEURAL NETWORKS

COL 774 MACHINE LEARNING: ASSIGNMENT 3

1 DECISION TREES (AND RANDOM FORESTS):

Part (a) - Decision Tree Construction

For this part, we simply construct a decision tree. For continuous attributes, we have a binary split based on median, whereas for categorical attributes, we have k splits, where k is the number of categories corresponding to that attribute.

For the construction of my decision tree:

Node class:

```
class Node:
    def __init__(self):
        self.numTrueSamples = 0
        self.numFalseSamples = 0
        self.isleaf = False

        self.splitAttribute = None
        self.splitValue = None # only useful in case of numerical attributes

        self.children = []
        self.depth = 0
```

DecisionTree class:

```
class DecisionTree:
    def __init__(self, max_depth, attributes, isNumAttributes, categoricalMappings):
        self.root = Node() # Root node
        self.root.depth = 0

        # Catalogue
        self.attributes = attributes
        self.isNumAttributes = isNumAttributes
        self.categoricalMappings = categoricalMappings

        self.max_depth = max_depth
```

We have used information gain to determine the best split at a node while training. This involves calculation of entropy at a node.

Entropy

Entropy is a measure of **impurity** or **uncertainty** at a node. For a classification problem, the entropy H at a node with a class distribution $p_1, p_2, \ldots, p_k$ is calculated as:

$$H = -\sum_{i=1}^{k} p_i \log_2(p_i)$$

- If the node is **pure** (all samples belong to one class), entropy is 0.
- If the node has an **even class distribution**, entropy is highest.

Information Gain

Information gain measures the **reduction in entropy** after a dataset is split on a particular feature:

$$\text{Information Gain} = H(\text{parent}) - \sum_{j} \frac{n_j}{n} H(\text{child}_j)$$

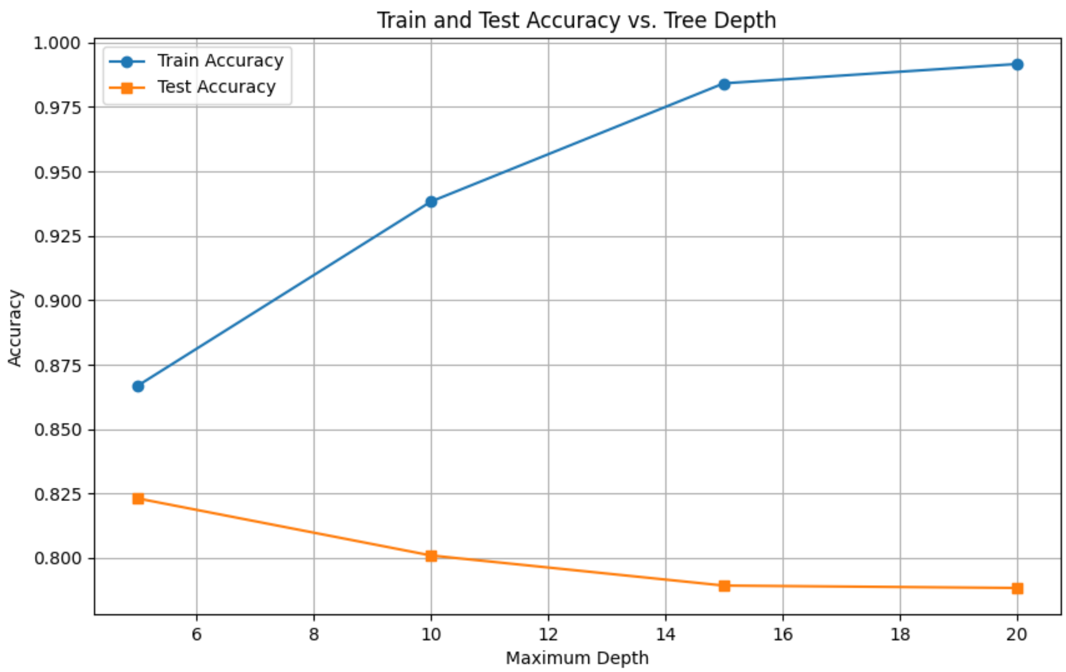
Where:

- $H(\text{parent})$ is the entropy before the split,
- $H(\text{child}_j)$ is the entropy of each child node,
- $\frac{n_j}{n}$ is the proportion of samples going into each child.

The split that **maximizes information gain** is selected as the best split at that node.

There is an edge case during prediction i.e. if the attribute in the test data has a category not seen during training, then we go with the majority class at the node where the decision path terminated (regardless of whether it is a leaf)

The plot of train and test accuracies is as follows:



From the plot, we can clearly observe that the train accuracy increases as the tree depth increases. However, we can see that the test accuracy decreases with increase in depth. This is because deeper trees tend to memorize the training data, capturing noise and leading to overfitting. As a result, they generalize poorly to unseen test data, reducing test accuracy.

The train accuracy remains the same after depth=20 at 99.16%. This is not 100%, since the splits can be in a way such that no more splits can increase the accuracy. (This is very much possible for the data given to us – capital.gain, capital.loss have 0 as their value for most of the records).

Here are the results of the decision tree:

Max Depth: 5 | Train Accuracy: 0.8669 | Test Accuracy: 0.8231
Max Depth: 10 | Train Accuracy: 0.9382 | Test Accuracy: 0.8010
Max Depth: 15 | Train Accuracy: 0.9841 | Test Accuracy: 0.7893
Max Depth: 20 | Train Accuracy: 0.9916 | Test Accuracy: 0.7884

Part (b) - Decision Tree Construction: One Hot Encoding

In this part, for each categorical attribute, we one-hot-encode it i.e. each attribute is now accounted for as several attributes of the form attribute_category, and the data values now become 0/1 only.

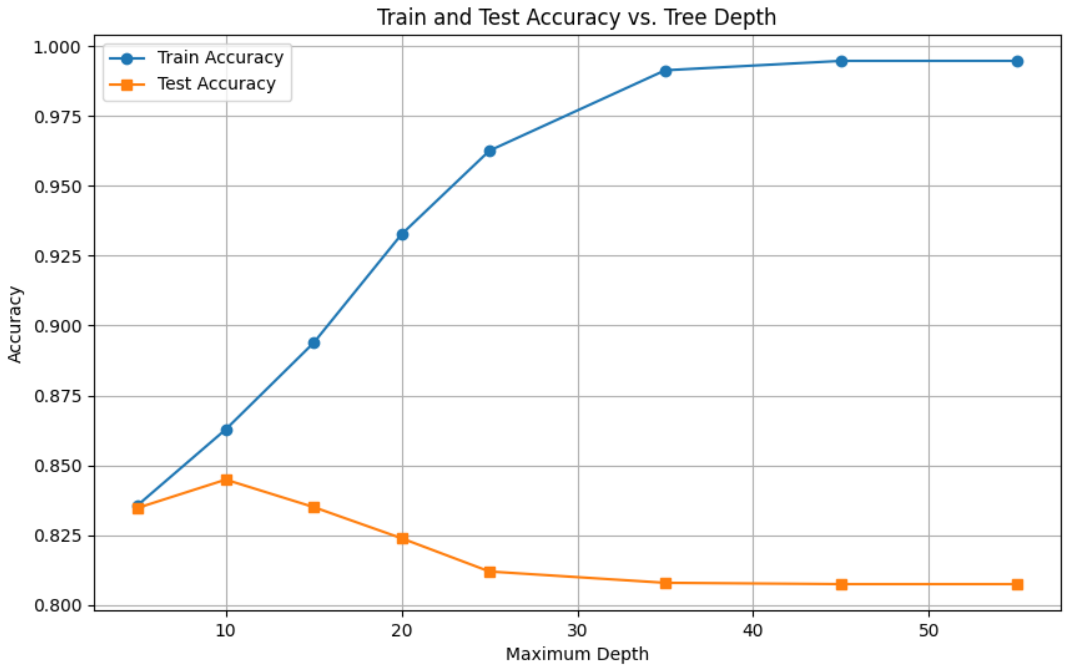
Since we must deal with this version in the assignment later, and the children list was somewhat inconvenient implementation-wise, we have new NodeOHE and DecisionTreeOHE classes to handle this part and the following ones.

```python
class NodeOHE:
    def __init__(self):
        self.numTrueSamples = 0
        self.numFalseSamples = 0
        self.isleaf = False

        self.splitAttribute = None
        self.splitValue = None # only useful in case of numerical attributes

        self.leftChild = None
        self.rightChild = None
        self.depth = 0

        # New fields for pruning
        self.parent = None           # Pointer to parent node
        self.numValidSamples = 0     # Validation samples reaching this node
        self.numValidTrue = 0        # Validation samples with y=1
        self.numValidFalse = 0       # Validation samples with y=0
        self.numValidCorrect = 0     # Validation samples correctly classified

class DecisionTreeOHE:
    def __init__(self, max_depth, attributes, isNumAttributes):
        self.root = NodeOHE() # Root node
        self.root.depth = 0

        # Catalogue
        self.attributes = attributes
        self.isNumAttributes = isNumAttributes

        self.max_depth = max_depth

        # Prune Tracking
        self.num_nodes_list = []
        self.train_acc_list = []
        self.valid_acc_list = []
        self.test_acc_list = []
```

The only difference between these classes and the previous one is that node class now has only two children always, and this sort-of gives a binary tree feeling, which is easier implementation-wise.

There are several fields which will be later useful during pruning.

We have included depths 5,10,15 and 20 for comparison



Here are the results of the decision tree:

Max Depth: 5 | Train Accuracy: 0.8358 | Test Accuracy: 0.8348
Max Depth: 10 | Train Accuracy: 0.8629 | Test Accuracy: 0.8449
Max Depth: 15 | Train Accuracy: 0.8938 | Test Accuracy: 0.8351
Max Depth: 20 | Train Accuracy: 0.9327 | Test Accuracy: 0.8239
Max Depth: 25 | Train Accuracy: 0.9626 | Test Accuracy: 0.8120
Max Depth: 35 | Train Accuracy: 0.9913 | Test Accuracy: 0.8080
Max Depth: 45 | Train Accuracy: 0.9947 | Test Accuracy: 0.8075
Max Depth: 55 | Train Accuracy: 0.9947 | Test Accuracy: 0.8075

The test accuracy is better in case of one-hot-encoded attributes in comparison to (a) for corresponding tree depths. One-hot encoding creates cleaner decision boundaries by transforming categorical variables into binary features, hence, we can see a higher test accuracy. The train accuracy has also improved slightly, which occurs since k-splits do not give as much information gain as 2-splits (a basic entropy result).

Part (c) - Decision Tree Construction: Pruning

One of the ways to prevent overfitting in decision trees (due to over exceeding number of nodes) is to prune the decision tree. In our case, we'll be performing post-pruning i.e. prune the internal nodes of the decision tree based on the validation set accuracy.

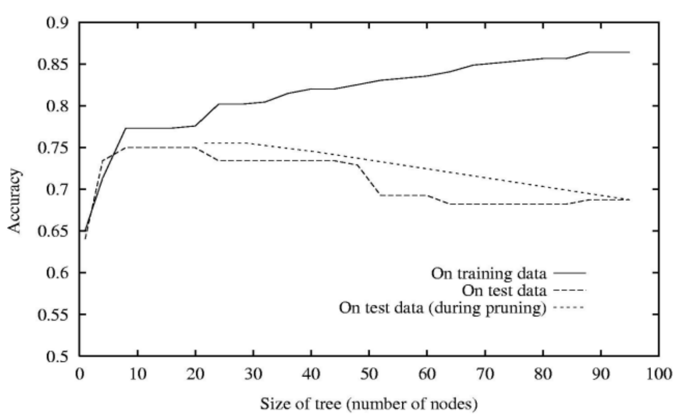
The algorithm, broadly, is as follows:

Do until further pruning is harmful:

1. Evaluate impact on *validation* set of pruning each possible node (plus those below it)
2. Greedily remove the one that most improves *validation* set accuracy

To optimize, notice that the examples affected go to the nodes lying in the decision path from the root of the decision tree to the pruned node. As a result, re-computation is only necessary for those nodes. Hence, the computation time decreases quite a lot, since now only a part of the tree and training samples are affected by pruning in each step. Time complexity is approximately O(M*(N+D)), M = #validationSamples, N = #nodes, D = depth of tree

Effect of Reduced-Error Pruning



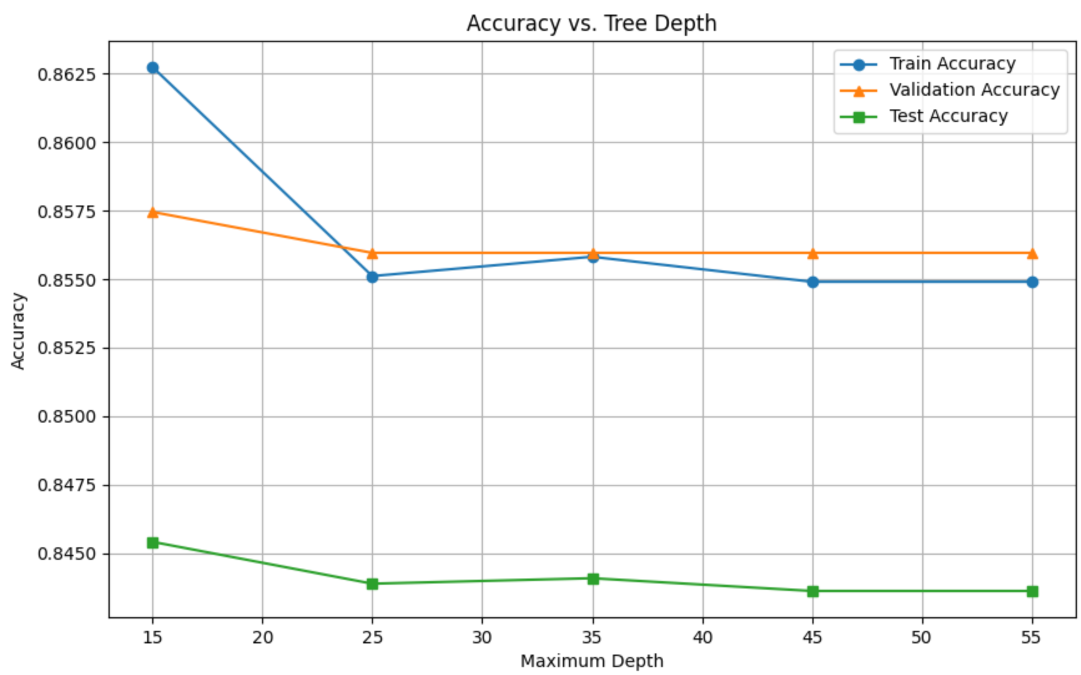
In our analysis, we are adding another maxDepth value 15

The results I have got are the following:

Max Depth: 15 | Train Accuracy: 0.8627 | Valid Accuracy: 0.8575 | Test Accuracy: 0.8454
Max Depth: 25 | Train Accuracy: 0.8551 | Valid Accuracy: 0.8560 | Test Accuracy: 0.8439

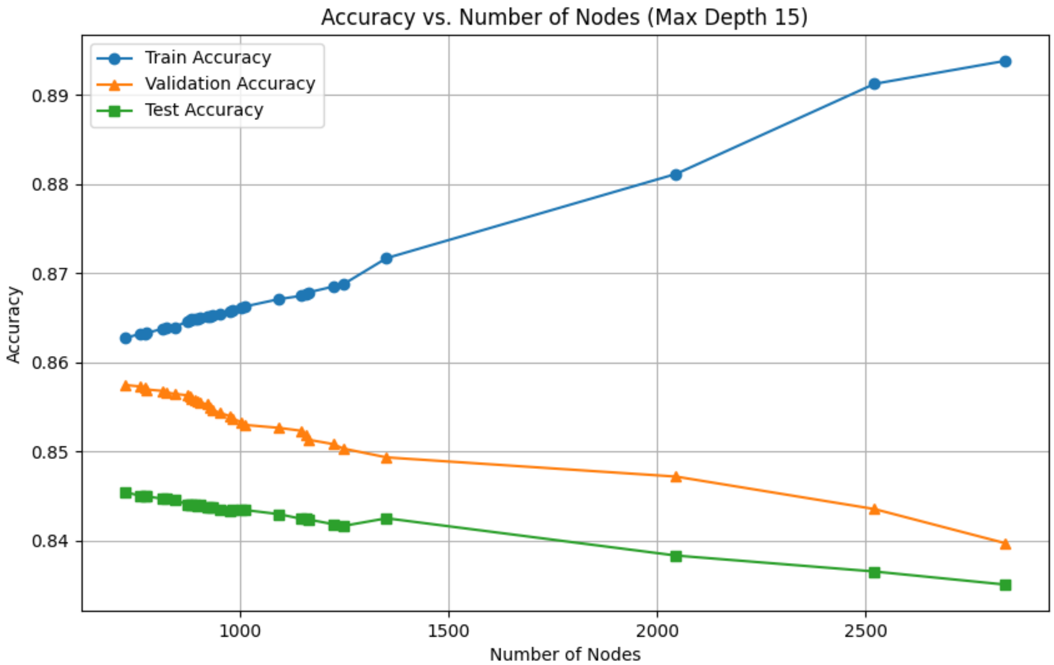
Max Depth: 35 | Train Accuracy: 0.8558 | Valid Accuracy: 0.8560 | Test Accuracy: 0.8441
Max Depth: 45 | Train Accuracy: 0.8549 | Valid Accuracy: 0.8560 | Test Accuracy: 0.8436
Max Depth: 55 | Train Accuracy: 0.8549 | Valid Accuracy: 0.8560 | Test Accuracy: 0.8436

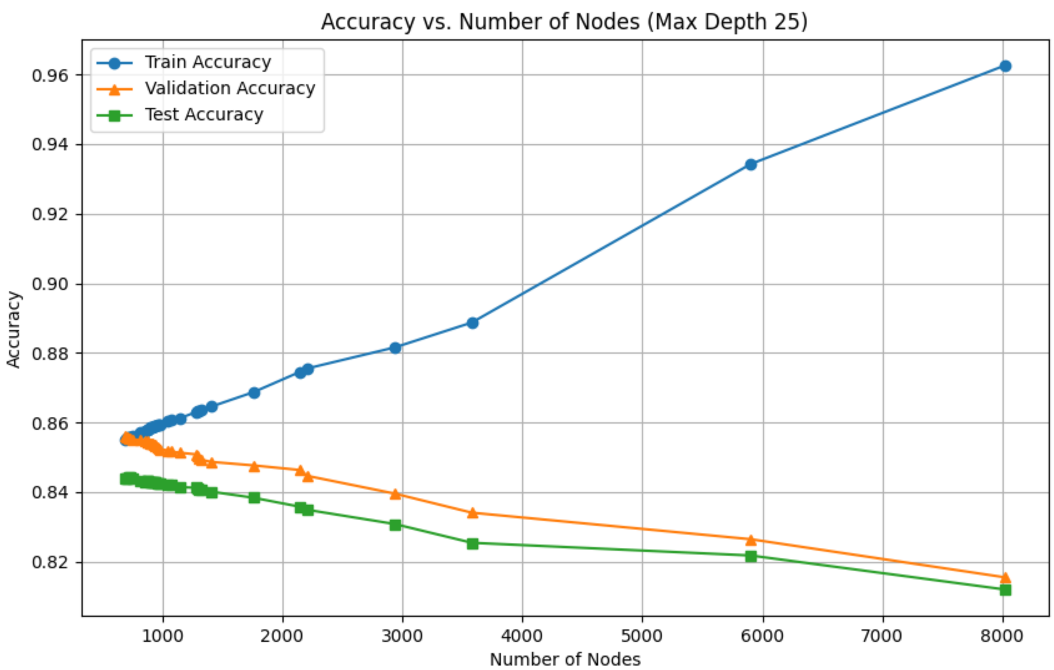
We can clearly observe that the trees do not overfit that much anymore (you can see the training accuracy going down and test accuracy going up for each tree depth as compared to part (b)).

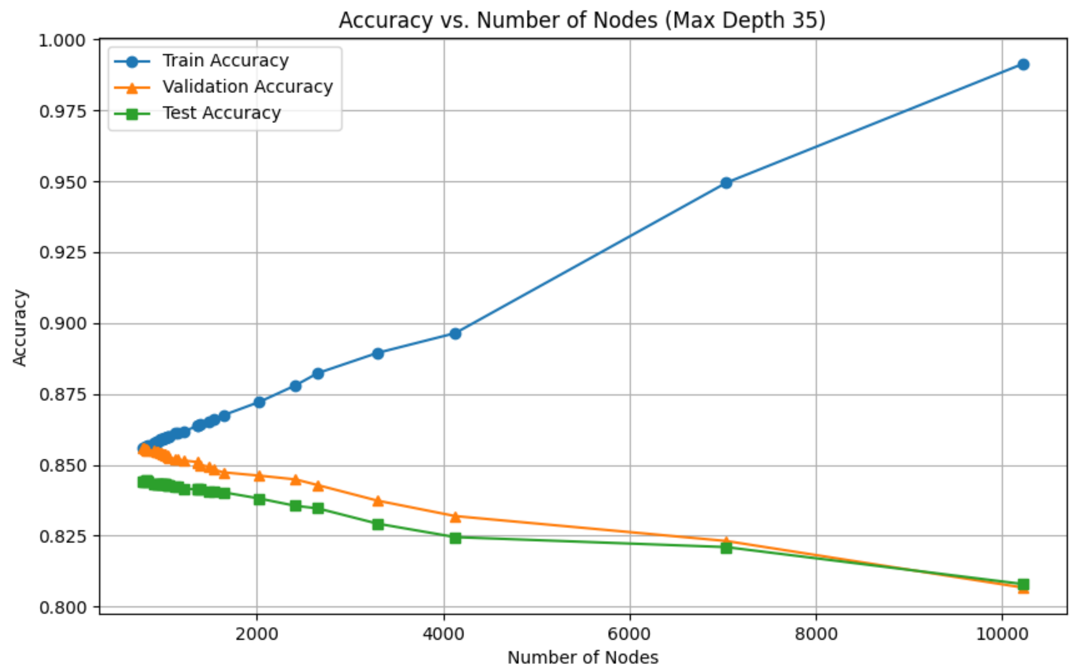


You can also observe from part (b) by the difference in accuracies that pruning is effective when the tree is grown fully. Pruning a tree with a smaller maxDepth doesn't really help that much.

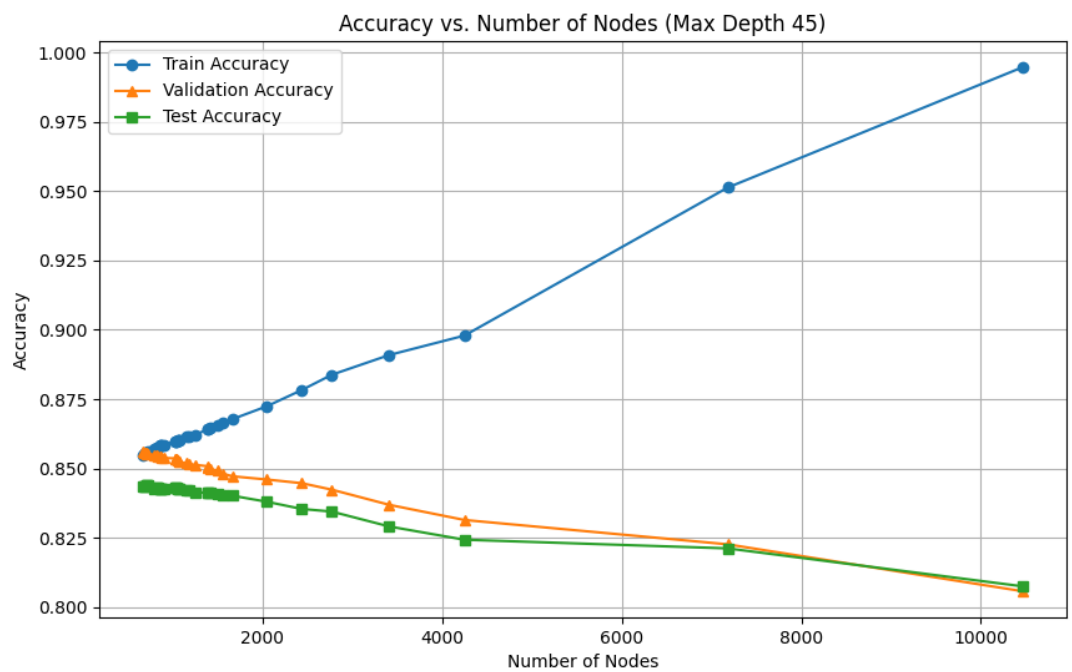
The plot Accuracy vs. Tree Depth gives a graphical summary of the results obtained at each maximum depth. Let's now look into the results for each depth a bit more closely:


Accuracy vs. Number of Nodes (Max Depth 15)
Train Accuracy
Validation Accuracy
Test Accuracy
Accuracy
Number of Nodes
0.89
0.88
0.87
0.86
0.85
0.84
1000
1500
2000
2500

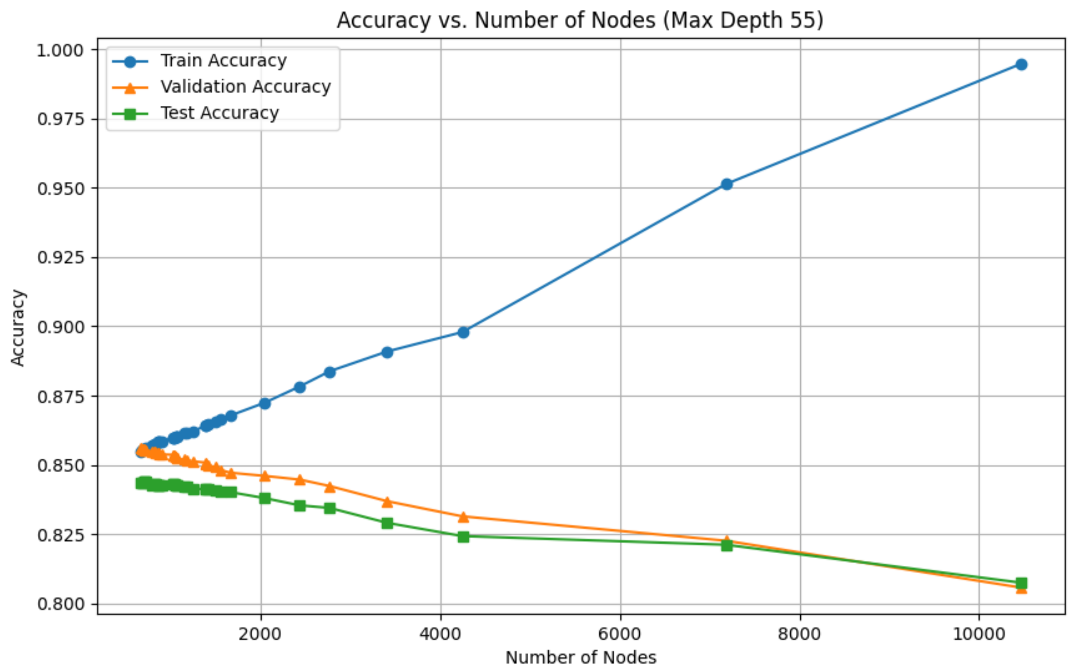

Accuracy vs. Number of Nodes (Max Depth 25)
Train Accuracy
Validation Accuracy
Test Accuracy
Accuracy
Number of Nodes
0.96
0.94
0.92
0.90
0.88
0.86
0.84
0.82
1000
2000
3000
4000
5000
6000
7000
8000



Accuracy vs. Number of Nodes (Max Depth 35)
Train Accuracy
Validation Accuracy
Test Accuracy
Accuracy
Number of Nodes



Accuracy vs. Number of Nodes (Max Depth 45)
Train Accuracy
Validation Accuracy
Test Accuracy
Accuracy
Number of Nodes



Decision tree pruning at varying depths (15, 25, 35, 45, 55) balances model complexity and generalization. At depth 15, pruning retains simpler trees, reducing overfitting but potentially underfitting complex data. Depths of 25 and 35 allow more nuanced splits, improving accuracy while still controlling variance. At 45 and 55, deeper trees capture intricate patterns but risk overfitting without aggressive pruning, requiring techniques like cost-complexity pruning to optimize performance in decision tree.

Part (d) - Decision Tree scikit-learn

In this section, we evaluate the performance of a decision tree classifier implemented using scikit-learn's **DecisionTreeClassifier** on a structured dataset. The main objective is to compare different configurations of the model by tuning two important hyperparameters: **max_depth** and **ccp_alpha**. These experiments aim to identify the best-performing model in terms of generalization accuracy while avoiding overfitting.

Scikit-learn's internal implementation uses one-hot encoding for multi-valued discrete attributes. Therefore, results obtained here are inherently comparable to those from custom implementations that explicitly split on each attribute value.

We proceed by first analyzing the effect of varying the maximum depth of the decision tree and then explore the impact of post-pruning using the cost-complexity pruning parameter (ccp_alpha). The performance is evaluated using accuracy on the training, validation, and test sets.

Tuning the Maximum Depth:

max_depth=25 | Train Accuracy: 0.9576 | Valid Accuracy: 0.8243 | Test Accuracy: 0.8210
max_depth=35 |Train Accuracy: 0.9896 | Valid Accuracy: 0.8117 | Test Accuracy: 0.8108
max_depth=45 | Train Accuracy: 0.9990 | Valid Accuracy: 0.8122 | Test Accuracy: 0.8085
max_depth=55 |Train Accuracy: 1.0000 | Valid Accuracy: 0.8125 | Test Accuracy: 0.8107

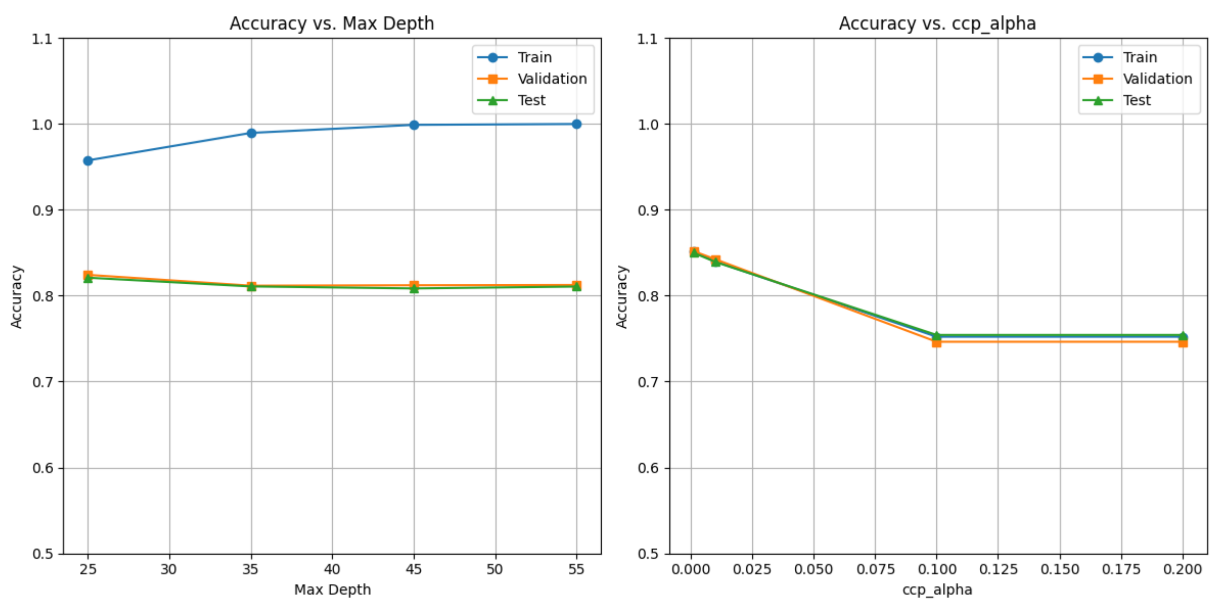
As observed, increasing the depth of the tree leads to an almost perfect fit on the training data. This is expected because deeper trees have more capacity to memorize the training examples by capturing even very specific patterns. However, this increased expressiveness comes at the cost of generalization. The validation and test accuracies slightly drop as the depth increases beyond 25.

The best validation accuracy (0.8243) and test accuracy (0.8210) are achieved when max_depth=25. This suggests that a depth of 25 strikes a balance between underfitting and overfitting: it provides enough complexity to model the decision boundaries while still maintaining good generalization.

Tuning the Pruning Parameter (ccp_alpha):

While limiting tree depth helps control overfitting, **cost-complexity pruning** offers a more refined approach by removing subtrees that contribute little to impurity reduction. The ccp_alpha parameter controls pruning strength—higher values lead to more aggressive simplification. As ccp_alpha increases, the model becomes simpler, but excessive pruning (e.g., 0.1 or 0.2) can cause underfitting. The best performance is achieved at ccp_alpha=0.001, with a validation accuracy of 0.8520 and test accuracy of 0.8501. Despite a lower training accuracy (0.8514), the model generalizes better, showing that pruning effectively mitigates overfitting.

ccp_alpha=0.001, Train Accuracy: 0.8514, Valid Accuracy: 0.8520, Test Accuracy: 0.8501
ccp_alpha=0.01, Train Accuracy: 0.8397, Valid Accuracy: 0.8424, Test Accuracy: 0.8392
ccp_alpha=0.1, Train Accuracy: 0.7522, Valid Accuracy: 0.7464, Test Accuracy: 0.7543
ccp_alpha=0.2, Train Accuracy: 0.7522, Valid Accuracy: 0.7464, Test Accuracy: 0.7543



Final Result:

Maximum Depth Tuning:
Best max_depth value: 25
Best validation accuracy: 0.8243
Corresponding test accuracy: 0.8210

Tuning the Pruning Parameter (ccp_alpha):
Best ccp_alpha value: 0.001
Best validation accuracy: 0.8520
Corresponding test accuracy: 0.8501

Comparison of best models:
Best max_depth model: Validation Accuracy = 0.8243, Test Accuracy = 0.8210

Best ccp_alpha model: Validation Accuracy = 0.8520, Test Accuracy = 0.8501
Final model uses ccp_alpha=0.001

Final model results:

Training accuracy: 0.8514
Validation accuracy: 0.8520
Test accuracy: 0.8501

Cost-complexity pruning improves generalization by removing branches that capture noise rather than meaningful patterns. The ccp_alpha parameter controls this: higher values prune more aggressively. At low values (e.g., 0.001), pruning removes unnecessary complexity while preserving useful structure, reducing overfitting. As ccp_alpha increases too much, the model starts underfitting by discarding important splits, lowering accuracy. The best performance at ccp_alpha=0.001 shows that careful pruning helps the model focus on signal over noise.

Part (e) - Random forests scikit-learn

In the sci-kit learn's Grid Classifier class, there was no smart way to remove cross-validation. Note that cross-validation is irrelevant here, since OOB accuracy already does that sort of work (around 1/3 of samples not even picked into the bag). But since there was no smart way to remove it, I settled with a cv value of 2.

Here is the output of the Grid Search (just starting few lines):

```
Fitting 2 folds for each of 120 candidates, totalling 240 fits
[CV 1/2] END max_features=0.1, min_samples_split=4, n_estimators=50;, score=0.848 total time=   2.8s
[CV 1/2] END max_features=0.1, min_samples_split=2, n_estimators=50;, score=0.840 total time=   2.9s
[CV 2/2] END max_features=0.1, min_samples_split=2, n_estimators=50;, score=0.848 total time=   3.0s
[CV 2/2] END max_features=0.1, min_samples_split=4, n_estimators=50;, score=0.852 total time=   3.0s
[CV 1/2] END max_features=0.1, min_samples_split=2, n_estimators=150;, score=0.845 total time=   3.6s
[CV 2/2] END max_features=0.1, min_samples_split=4, n_estimators=150;, score=0.856 total time=   3.7s
[CV 1/2] END max_features=0.1, min_samples_split=6, n_estimators=50;, score=0.848 total time=   0.6s
[CV 2/2] END max_features=0.1, min_samples_split=2, n_estimators=150;, score=0.850 total time=   4.6s
[CV 2/2] END max_features=0.1, min_samples_split=6, n_estimators=50;, score=0.853 total time=   1.6s
[CV 2/2] END max_features=0.1, min_samples_split=2, n_estimators=250;, score=0.853 total time=   5.1s
[CV 1/2] END max_features=0.1, min_samples_split=6, n_estimators=150;, score=0.851 total time=   2.1s
[CV 1/2] END max_features=0.1, min_samples_split=4, n_estimators=250;, score=0.849 total time=   5.4s
...
```

Final Result:

Best Parameters: {'max_features': 0.5, 'min_samples_split': 8, 'n_estimators': 150}
Training Accuracy: 0.9703
OOB Accuracy: 0.8576
Validation Accuracy: 0.8571
Test Accuracy: 0.8541

The grid search evaluated 120 Random Forest models using 2-fold cross-validation, totaling 240 fits. The best-performing models consistently used max_features=0.5, min_samples_split=8 or 10, and n_estimators=250 or 350, achieving cross-validation scores up to **0.863**. Increasing n_estimators generally improved performance but also increased training time. Using too many features (max_features=0.9) tended to slightly reduce accuracy, indicating some overfitting. Overall, moderate feature subsets and deeper trees (lower min_samples_split) balanced bias and variance well.

Comparison with (b) and (c):

Part (e) using Random Forest outperforms Parts (b) and (c) in generalization. While training accuracy (97.03%) is slightly lower than Part (b), the test accuracy (85.41%) and OOB accuracy (85.76%) are more stable and higher, indicating better generalization. Random Forest reduces overfitting by averaging across multiple trees, whereas a single decision tree in Part (b) overfits. Part (c) uses pruning to prevent overfitting but sacrifices some accuracy. Overall, Random Forest in Part (e) balances training and test accuracy more effectively than both non-pruned and pruned decision trees.

When comparing part (d) (tuning max_depth and ccp_alpha) with part (e) (Random Forest), we observe:

Overfitting and Generalization: Random Forest in part (e) provides better generalization with a test accuracy of 85.41%, while part (d) shows consistent performance across train, validation, and test sets, indicating good fine-tuning.
Bias-Variance Trade-off: Part (d) reduces overfitting with ~85% accuracy across datasets, while Random Forest in part (e) reduces variance through its ensemble approach, achieving slightly better test accuracy.
Complexity: Random Forest, with multiple trees and feature selection, is more

robust and less prone to overfitting, whereas part (d) benefits from pruning to control complexity.

In conclusion, part (d) offers a well-tuned decision tree, while part (e) leverages the ensemble power of Random Forest, leading to more stable, slightly superior performance

2 NEURAL NETWORKS:

Part (a) - Neural Network Construction

For the implementation of neural networks, my NeuralNetwork class defines a simple feedforward neural network for multi-class classification, with implementations of forward propagation, backpropagation, and training using mini-batch stochastic gradient descent (SGD).

Here are the salient features:

Weights Initialization: Each weight matrix is initialized using a **Xavier initialization** (also called Glorot initialization), which helps in avoiding vanishing or exploding gradients:

$$W_i \sim \mathcal{N}(0, \frac{2}{n_i})$$

where n_i is the number of nodes in the previous layer.

Bias Initialization: Biases are initialized to zero:

$$b_i = 0$$

Loss Function (`computeCrossEntropyLoss` method):

The network uses **cross-entropy loss** for multi-class classification:

$$\mathcal{L} = -\frac{1}{m} \sum_{i=1}^{m} \sum_{c=1}^{C} y_{i,c} \log(\hat{y}_{i,c})$$

- Where:
  - $y_{i,c}$ is the true label of class c for the i-th sample (one-hot encoded).
  - $\hat{y}_{i,c}$ is the predicted probability for class c for the i-th sample.
  - m is the number of samples.
  - C is the number of classes.

Activation Functions:

1. **Sigmoid Activation**: This function is used in the hidden layers:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

   - This function squashes input values into the range [0, 1].

2. **Sigmoid Derivative**: The derivative of the sigmoid function is used during backpropagation:

$$\sigma'(x) = \sigma(x)(1 - \sigma(x))$$

   - This derivative is needed to compute the gradient of the loss function with respect to the weights and biases.

3. **Softmax Activation**: This function is used in the output layer for multi-class classification:

$$\text{softmax}(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

   - This function converts the output values into a probability distribution across multiple classes, where each output is between 0 and 1, and the sum of all output values equals 1.

Parameter Update (`updateParameters` method):

The weights and biases are updated using gradient descent:

$$W^{[l]} := W^{[l]} - \eta \frac{\partial \mathcal{L}}{\partial W^{[l]}}$$

$$b^{[l]} := b^{[l]} - \eta \frac{\partial \mathcal{L}}{\partial b^{[l]}}$$

Where η is the learning rate.

Adaptive Learning Rate (`train_with_adaptive_lr` method):

The learning rate decays over time using the formula:

$$\eta_{\text{new}} = \frac{\eta_0}{\sqrt{\text{epoch} + 1}}$$

Where η_0 is the initial learning rate and epoch is the current epoch number.

Backpropagation (`backwardPropagation` method):

- **Output Layer Gradient**: The gradient for the output layer is computed as the difference between predicted and true values:

$$\delta^{[L]} = \hat{y} - y$$

Where L is the output layer, $\hat{y}$ is the predicted output, and y is the true output.

- **Hidden Layer Gradients**: For each hidden layer, the gradient is computed by backpropagating the error from the layer after it:

$$\delta^{[l]} = \left(\delta^{[l+1]} W^{[l+1]T}\right) \cdot \sigma'(z^{[l]})$$

Where:

- $\delta^{[l]}$ is the error term for layer l.
- $W^{[l+1]T}$ is the transposed weight matrix from the layer after l.
- $\sigma'(z^{[l]})$ is the derivative of the activation function applied to the output of layer l.

- **Weight and Bias Gradients**: The gradients for weights and biases are computed as follows:

$$\frac{\partial \mathcal{L}}{\partial W^{[l]}} = \frac{1}{m} \cdot \text{activation}^{[l-1]T} \cdot \delta^{[l]}$$

$$\frac{\partial \mathcal{L}}{\partial b^{[l]}} = \frac{1}{m} \sum_{i=1}^{m} \delta^{[l]}$$

- Where:
  - $W^{[l]}$ are the weights at layer l.
  - $b^{[l]}$ are the biases at layer l.
  - $\delta^{[l]}$ is the error at layer l.

I have also implemented early stopping based on validation loss to prevent overfitting. If the validation loss is around the same (i.e. difference of less than 1e-4) for some n=10 iterations, then you stop the training and do not run any further epochs

Part (b) - Experiments with a single hidden layer

Stopping criterion:

The stopping criteria in this case is that num_epochs < 100 as well as the early stopping criteria described in (a) in case it overfits. As such, no overfitting was observed (num_epochs is very low, but accuracy is good)

Recall, Precision and F1 Score:

In the previous report, I explained what these three terms mean and their different averaging methods. Here, we have an imbalanced dataset, and we are going for macro-averaging methods in this case.

Summary of Results:

Hidden Units	Train F1	Test F1	Train Accuracy	Test Accuracy
1	0.0228	0.0214	13.19%	13.10%
5	0.2731	0.2467	57.72%	54.13%
10	0.4892	0.4418	72.95%	68.06%
50	0.9112	0.7642	93.33%	84.74%
100	0.9287	0.7877	94.09%	85.34%

The results indicate that as the number of hidden units increases, both training and test accuracy improve, which is expected as a larger number of neurons allows the network to capture more complex patterns. However, there is also a noticeable gap between train and test performance, suggesting potential overfitting, especially with 50 and 100 hidden units. The model's performance stabilizes after a certain point, as seen with the 100 hidden units.

Recall, Precision and F1 Score per class:

Hidden Units = 1

Test Metrics:

```
Test Metrics:
Accuracy: 13.10%
Average Precision: 0.0159
Average Recall: 0.0550
Average F1 Score: 0.0214

Per-Class Metrics:
Class | Precision | Recall | F1 Score
 0    | 0.0000    | 0.0000 | 0.0000
 1    | 0.1393    | 0.9431 | 0.2428
 2    | 0.0597    | 0.1373 | 0.0832
 3    | 0.0000    | 0.0000 | 0.0000
 4    | 0.0000    | 0.0000 | 0.0000
 5    | 0.0613    | 0.1349 | 0.0843
 6    | 0.0000    | 0.0000 | 0.0000
 7    | 0.1050    | 0.1400 | 0.1200
 8    | 0.0000    | 0.0000 | 0.0000
 9    | 0.0000    | 0.0000 | 0.0000
10    | 0.0471    | 0.0242 | 0.0320
11    | 0.0000    | 0.0000 | 0.0000
12    | 0.0730    | 0.0188 | 0.0300
13    | 0.1971    | 0.9653 | 0.3274
14    | 0.0000    | 0.0000 | 0.0000
15    | 0.0000    | 0.0000 | 0.0000
16    | 0.0000    | 0.0000 | 0.0000
17    | 0.0000    | 0.0000 | 0.0000
18    | 0.0000    | 0.0000 | 0.0000
19    | 0.0000    | 0.0000 | 0.0000
20    | 0.0000    | 0.0000 | 0.0000
21    | 0.0000    | 0.0000 | 0.0000
22    | 0.0000    | 0.0000 | 0.0000
23    | 0.0000    | 0.0000 | 0.0000
24    | 0.0000    | 0.0000 | 0.0000
25    | 0.0000    | 0.0000 | 0.0000
26    | 0.0000    | 0.0000 | 0.0000
27    | 0.0000    | 0.0000 | 0.0000
28    | 0.0000    | 0.0000 | 0.0000
29    | 0.0000    | 0.0000 | 0.0000
30    | 0.0000    | 0.0000 | 0.0000
31    | 0.0000    | 0.0000 | 0.0000
32    | 0.0000    | 0.0000 | 0.0000
33    | 0.0000    | 0.0000 | 0.0000
34    | 0.0000    | 0.0000 | 0.0000
35    | 0.0000    | 0.0000 | 0.0000
36    | 0.0000    | 0.0000 | 0.0000
37    | 0.0000    | 0.0000 | 0.0000
38    | 0.0000    | 0.0000 | 0.0000
39    | 0.0000    | 0.0000 | 0.0000
40    | 0.0000    | 0.0000 | 0.0000
41    | 0.0000    | 0.0000 | 0.0000
42    | 0.0000    | 0.0000 | 0.0000
```

Hidden Units = 5

Test Metrics:

```
Test Metrics:
Accuracy: 54.13%
Average Precision: 0.2632
Average Recall: 0.2841
Average F1 Score: 0.2467

Per-Class Metrics:
Class | Precision | Recall | F1 Score
 0    | 0.0000    | 0.0000 | 0.0000
 1    | 0.6105    | 0.8556 | 0.7126
 2    | 0.4604    | 0.8680 | 0.6017
 3    | 0.3852    | 0.2089 | 0.2709
 4    | 0.7323    | 0.6258 | 0.6748
 5    | 0.3413    | 0.3381 | 0.3397
 6    | 0.8333    | 0.0333 | 0.0641
 7    | 0.4179    | 0.6956 | 0.5221
 8    | 0.2329    | 0.0756 | 0.1141
 9    | 0.7441    | 0.7937 | 0.7681
10    | 0.6305    | 0.8970 | 0.7405
11    | 0.4865    | 0.8595 | 0.6213
12    | 0.8443    | 0.8406 | 0.8424
13    | 0.8252    | 0.9111 | 0.8660
14    | 0.0000    | 0.0000 | 0.0000
15    | 0.0870    | 0.0095 | 0.0172
16    | 0.0000    | 0.0000 | 0.0000
17    | 0.8386    | 0.9528 | 0.8921
18    | 0.2057    | 0.6282 | 0.3099
19    | 0.0000    | 0.0000 | 0.0000
20    | 0.0000    | 0.0000 | 0.0000
21    | 0.0000    | 0.0000 | 0.0000
22    | 0.0000    | 0.0000 | 0.0000
23    | 0.0000    | 0.0000 | 0.0000
24    | 0.0000    | 0.0000 | 0.0000
25    | 0.5455    | 0.7500 | 0.6316
26    | 0.0000    | 0.0000 | 0.0000
27    | 0.0000    | 0.0000 | 0.0000
28    | 0.0000    | 0.0000 | 0.0000
29    | 0.0000    | 0.0000 | 0.0000
30    | 0.0000    | 0.0000 | 0.0000
31    | 0.6667    | 0.0815 | 0.1452
32    | 0.0000    | 0.0000 | 0.0000
33    | 0.4458    | 0.1762 | 0.2526
34    | 0.0000    | 0.0000 | 0.0000
35    | 0.4205    | 0.6513 | 0.5111
36    | 0.0000    | 0.0000 | 0.0000
37    | 0.0000    | 0.0000 | 0.0000
38    | 0.5627    | 0.9623 | 0.7102
39    | 0.0000    | 0.0000 | 0.0000
40    | 0.0000    | 0.0000 | 0.0000
41    | 0.0000    | 0.0000 | 0.0000
42    | 0.0000    | 0.0000 | 0.0000
```

Hidden Units = 10

Test Metrics:

```
Test Metrics:
Accuracy: 68.06%
Average Precision: 0.4967
Average Recall: 0.4545
Average F1 Score: 0.4418

Per-Class Metrics:
Class | Precision | Recall | F1 Score
 0    | 0.0000    | 0.0000 | 0.0000
 1    | 0.6597    | 0.6542 | 0.6569
 2    | 0.5000    | 0.7253 | 0.5919
 3    | 0.6678    | 0.8400 | 0.7441
 4    | 0.6424    | 0.8030 | 0.7138
 5    | 0.5186    | 0.4206 | 0.4645
 6    | 0.3824    | 0.6067 | 0.4691
 7    | 0.7518    | 0.6933 | 0.7214
 8    | 0.5769    | 0.6756 | 0.6223
 9    | 0.8151    | 0.7896 | 0.8021
10    | 0.7259    | 0.8667 | 0.7901
11    | 0.6135    | 0.8881 | 0.7257
12    | 0.8863    | 0.9377 | 0.9113
13    | 0.9168    | 0.9792 | 0.9469
14    | 0.8984    | 0.8185 | 0.8566
15    | 0.7054    | 0.4333 | 0.5369
16    | 0.8889    | 0.0533 | 0.1006
17    | 0.7604    | 0.9611 | 0.8491
18    | 0.6017    | 0.7282 | 0.6589
19    | 0.0000    | 0.0000 | 0.0000
20    | 0.0000    | 0.0000 | 0.0000
21    | 0.7619    | 0.1778 | 0.2883
22    | 0.0000    | 0.0000 | 0.0000
23    | 0.3971    | 0.1800 | 0.2477
24    | 0.0000    | 0.0000 | 0.0000
25    | 0.5658    | 0.8417 | 0.6767
26    | 0.3448    | 0.1111 | 0.1681
27    | 0.0000    | 0.0000 | 0.0000
28    | 0.6939    | 0.2267 | 0.3417
29    | 0.0000    | 0.0000 | 0.0000
30    | 0.0000    | 0.0000 | 0.0000
31    | 0.3996    | 0.7370 | 0.5182
32    | 0.0000    | 0.0000 | 0.0000
33    | 0.8053    | 0.8667 | 0.8349
34    | 0.8583    | 0.8583 | 0.8583
35    | 0.9276    | 0.9205 | 0.9241
36    | 0.0000    | 0.0000 | 0.0000
37    | 0.0000    | 0.0000 | 0.0000
38    | 0.7853    | 0.9594 | 0.8637
39    | 0.7347    | 0.4000 | 0.5180
40    | 0.9062    | 0.3222 | 0.4754
41    | 0.6667    | 0.0667 | 0.1212
42    | 0.0000    | 0.0000 | 0.0000
```

Hidden Units = 50

Test Metrics:

```
Test Metrics:
Accuracy: 84.74%
Average Precision: 0.8439
Average Recall: 0.7440
Average F1 Score: 0.7642

Per-Class Metrics:
Class | Precision | Recall | F1 Score
 0    | 0.9524    | 0.3333 | 0.4938
 1    | 0.7847    | 0.9111 | 0.8432
 2    | 0.8028    | 0.9120 | 0.8539
 3    | 0.7490    | 0.8756 | 0.8074
 4    | 0.8669    | 0.8091 | 0.8370
 5    | 0.7862    | 0.8111 | 0.7984
 6    | 0.6753    | 0.6933 | 0.6842
 7    | 0.8561    | 0.7667 | 0.8089
 8    | 0.8194    | 0.8067 | 0.8130
 9    | 0.9219    | 0.8604 | 0.8901
10    | 0.8360    | 0.9424 | 0.8860
11    | 0.8542    | 0.8929 | 0.8731
12    | 0.9615    | 0.9406 | 0.9509
13    | 0.9578    | 0.9778 | 0.9677
14    | 0.9841    | 0.9185 | 0.9502
15    | 0.8805    | 0.9476 | 0.9128
16    | 0.9840    | 0.8200 | 0.8945
17    | 0.9489    | 0.9806 | 0.9645
18    | 0.7015    | 0.7231 | 0.7121
19    | 0.4000    | 0.0333 | 0.0615
20    | 0.6222    | 0.6222 | 0.6222
21    | 0.9429    | 0.3667 | 0.5280
22    | 0.8254    | 0.8667 | 0.8455
23    | 0.6716    | 0.6000 | 0.6338
24    | 1.0000    | 0.1222 | 0.2178
25    | 0.8189    | 0.8854 | 0.8509
26    | 0.6422    | 0.7278 | 0.6823
27    | 0.8333    | 0.2500 | 0.3846
28    | 0.8952    | 0.7400 | 0.8102
29    | 0.9250    | 0.8222 | 0.8706
30    | 0.6414    | 0.6200 | 0.6305
31    | 0.6080    | 0.7926 | 0.6881
32    | 0.6364    | 0.1167 | 0.1972
33    | 0.9283    | 0.9857 | 0.9561
34    | 0.9573    | 0.9333 | 0.9451
35    | 0.9759    | 0.9333 | 0.9541
36    | 0.9820    | 0.9083 | 0.9437
37    | 0.9787    | 0.7667 | 0.8598
38    | 0.9409    | 0.9696 | 0.9550
39    | 1.0000    | 0.6667 | 0.8000
40    | 0.8165    | 0.9889 | 0.8945
41    | 0.9512    | 0.6500 | 0.7723
42    | 0.9692    | 0.7000 | 0.8129
```

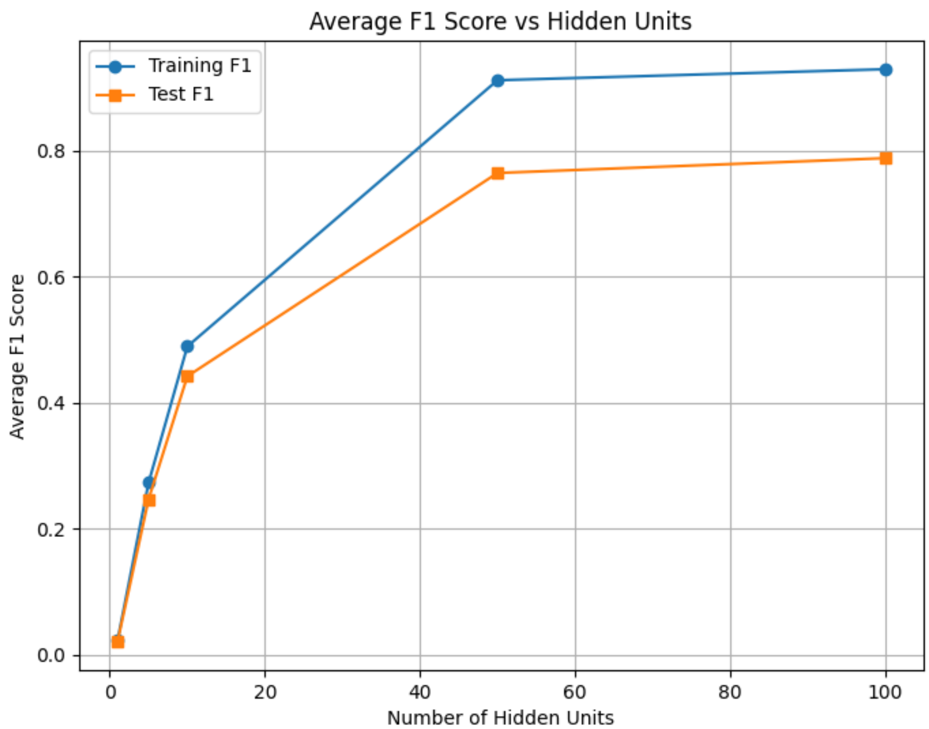
Hidden Units = 100

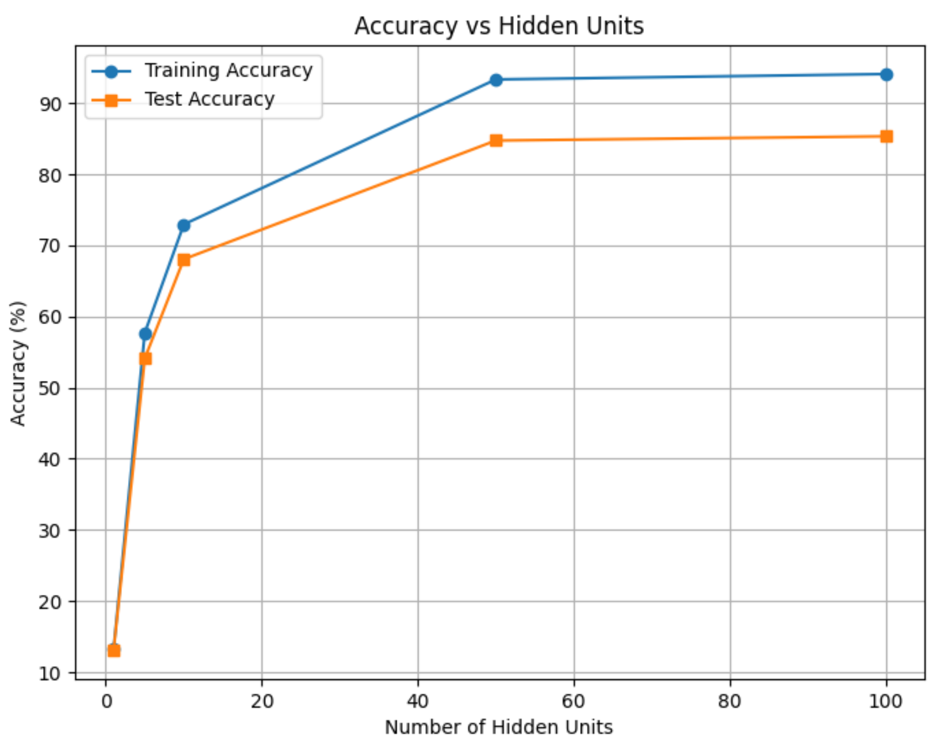
Test Metrics:

Test Metrics:
Accuracy: 85.34%
Average Precision: 0.8580
Average Recall: 0.7647
Average F1 Score: 0.7877

Per-Class Metrics:

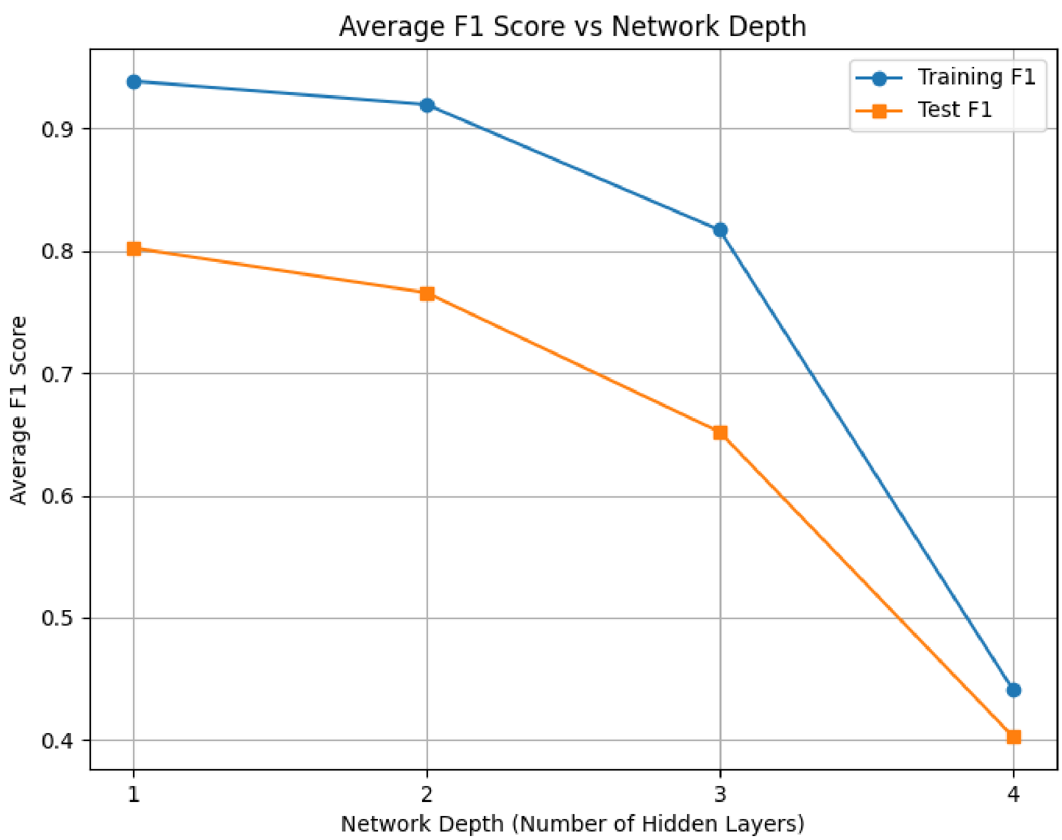
Class	Precision	Recall	F1 Score
0	1.0000	0.3000	0.4615
1	0.7564	0.9097	0.8260
2	0.8009	0.9067	0.8505
3	0.8143	0.8578	0.8355
4	0.8554	0.8333	0.8442
5	0.7462	0.8540	0.7964
6	0.6074	0.6600	0.6326
7	0.8633	0.7578	0.8071
8	0.8264	0.7933	0.8095
9	0.9420	0.8458	0.8913
10	0.8517	0.9485	0.8975
11	0.8642	0.8786	0.8713
12	0.9602	0.9449	0.9525
13	0.9629	0.9722	0.9675
14	0.9795	0.8852	0.9300
15	0.8767	0.9476	0.9108
16	0.9769	0.8467	0.9071
17	0.9353	0.9639	0.9494
18	0.7870	0.6821	0.7308
19	0.5714	0.2000	0.2963
20	0.5462	0.7222	0.6220
21	0.9375	0.3333	0.4918
22	0.9533	0.8500	0.8987
23	0.7881	0.6200	0.6940
24	0.9600	0.2667	0.4174
25	0.8359	0.8917	0.8629
26	0.6419	0.8167	0.7188
27	0.8889	0.2667	0.4103
28	0.9483	0.7333	0.8271
29	0.9333	0.9333	0.9333
30	0.7559	0.6400	0.6931
31	0.6775	0.8481	0.7533
32	0.8158	0.5167	0.6327
33	0.9224	0.9619	0.9417
34	0.9914	0.9583	0.9746
35	0.9639	0.9590	0.9614
36	0.9224	0.8917	0.9068
37	0.9412	0.8000	0.8649
38	0.9612	0.9696	0.9654
39	0.8955	0.6667	0.7643
40	0.7350	0.9556	0.8309
41	0.9556	0.7167	0.8190
42	0.9455	0.5778	0.7172


Average F1 Score vs Hidden Units
Training F1
Test F1
Average F1 Score
0.0
0.2
0.4
0.6
0.8
0
20
40
60
80
100
Number of Hidden Units


Accuracy vs Hidden Units
Training Accuracy
Test Accuracy
Accuracy (%)
10
20
30
40
50
60
70
80
90
0
20
40
60
80
100
Number of Hidden Units

Part (c) - Experiments with multiple hidden layer(s)

Summary of Results:



Metrics for hidden units: 512
Average Train F1: 0.9397
Average Test F1: 0.8000
Best performing class: 38 (F1=0.9683)
Worst performing class: 24 (F1=0.4247)

Metrics for hidden units: 512_256
Average Train F1: 0.9120
Average Test F1: 0.7412
Best performing class: 17 (F1=0.9696)
Worst performing class: 24 (F1=0.2914)

Metrics for hidden units: 512_256_128
Average Train F1: 0.7194
Average Test F1: 0.5762
Best performing class: 13 (F1=0.9631)
Worst performing class: 0 (F1=0.0000)

Metrics for hidden units: 512_256_128_64
Average Train F1: 0.1987
Average Test F1: 0.1725
Best performing class: 13 (F1=0.7549)
Worst performing class: 0 (F1=0.0000)

Explanation:

As the number of hidden units and layers increases, the training F1 score decreases significantly, while the test F1 score follows a similar downward trend, indicating overfitting. The model performs well with 512 hidden units but struggles to generalize as more layers and nodes are added, especially with the performance of certain classes (e.g., class 0 with F1=0.0000). The drop in F1 scores with larger architectures suggests that increasing complexity beyond a certain point may harm generalization.

Part (d) - Experiments with multiple hidden layer(s) using adaptive learning rate

Summary of Results:

Summary of Results (Adaptive Learning Rate):

Hidden Layers	Depth	Train F1	Test F1	Train Accuracy	Test Accuracy
[512]	1	0.6337	0.5374	76.04%	67.44%
[512, 256]	2	0.2928	0.2524	52.52%	47.66%
[512, 256, 128]	3	0.0694	0.0580	21.31%	19.31%
[512, 256, 128, 64]	4	0.0154	0.0181	9.56%	10.82%

Explanation:

With an adaptive learning rate, the performance initially improves for a single hidden layer (512 units), but as more layers are added, both training and test F1 scores decline sharply. The decrease in performance suggests that the learning rate decay, while preventing overfitting, may not have been ideal for deeper architectures, causing the model to struggle with convergence as complexity increases.

Test Metrics:
Accuracy: 67.44%
Average Precision: 0.6016
Average Recall: 0.5226
Average F1 Score: 0.5374

Per-Class Metrics:

Class	Precision	Recall	F1 Score
0	0.0000	0.0000	0.0000
1	0.7213	0.8375	0.7751
2	0.6241	0.6573	0.6403
3	0.4510	0.3578	0.3990
4	0.6267	0.6106	0.6186
5	0.3473	0.5270	0.4187
6	0.5584	0.5733	0.5658
7	0.6822	0.5200	0.5902
8	0.2411	0.3311	0.2790
9	0.9048	0.6729	0.7718
10	0.6565	0.9152	0.7646
11	0.7851	0.8524	0.8174
12	0.8570	0.9464	0.8994
13	0.9383	0.9722	0.9550
14	0.9430	0.7963	0.8635
15	0.9217	0.7286	0.8138
16	0.9184	0.6000	0.7258
17	0.9130	0.9333	0.9231
18	0.6082	0.5692	0.5881
19	0.0000	0.0000	0.0000
20	0.0000	0.0000	0.0000
21	0.8333	0.0556	0.1042
22	0.8000	0.6333	0.7070
23	0.2952	0.2067	0.2431
24	0.0000	0.0000	0.0000
25	0.5825	0.8750	0.6994
26	0.6087	0.4667	0.5283
27	0.0000	0.0000	0.0000
28	0.5385	0.5600	0.5490
29	0.0000	0.0000	0.0000
30	0.0000	0.0000	0.0000
31	0.6102	0.5741	0.5916
32	0.0000	0.0000	0.0000
33	0.9431	0.9476	0.9454
34	0.9904	0.8583	0.9196
35	0.9114	0.7385	0.8159
36	0.9722	0.5833	0.7292
37	1.0000	0.3833	0.5542
38	0.6824	0.9841	0.8059
39	0.9836	0.6667	0.7947
40	0.9540	0.9222	0.9379
41	0.4630	0.4167	0.4386
42	1.0000	0.2000	0.3333

Test Metrics:
Accuracy: 47.66%
Average Precision: 0.2937
Average Recall: 0.2738
Average F1 Score: 0.2524

Per-Class Metrics:

Class	Precision	Recall	F1 Score
0	0.0000	0.0000	0.0000
1	0.6379	0.6264	0.6321
2	0.3515	0.5600	0.4319
3	0.1575	0.1467	0.1519
4	0.2891	0.2409	0.2628
5	0.1438	0.1381	0.1409
6	0.0000	0.0000	0.0000
7	0.6798	0.3822	0.4893
8	0.1030	0.0756	0.0872
9	0.5049	0.5417	0.5226
10	0.3624	0.7439	0.4873
11	0.6499	0.8310	0.7294
12	0.5593	0.9087	0.6924
13	0.8830	0.9750	0.9267
14	0.9045	0.7370	0.8122
15	0.0000	0.0000	0.0000
16	0.0000	0.0000	0.0000
17	0.8556	0.9056	0.8799
18	0.3295	0.4436	0.3781
19	0.0000	0.0000	0.0000
20	0.0000	0.0000	0.0000
21	0.0000	0.0000	0.0000
22	0.0000	0.0000	0.0000
23	0.0000	0.0000	0.0000
24	0.0000	0.0000	0.0000
25	0.3722	0.7188	0.4904
26	0.8333	0.0278	0.0538
27	0.0000	0.0000	0.0000
28	0.0000	0.0000	0.0000
29	0.0000	0.0000	0.0000
30	0.0000	0.0000	0.0000
31	0.4365	0.4074	0.4215
32	0.0000	0.0000	0.0000
33	0.5849	0.5905	0.5877
34	1.0000	0.1167	0.2090
35	0.6607	0.5692	0.6116
36	0.0000	0.0000	0.0000
37	0.0000	0.0000	0.0000
38	0.4971	0.9768	0.6588
39	0.0000	0.0000	0.0000
40	0.8333	0.1111	0.1961
41	0.0000	0.0000	0.0000
42	0.0000	0.0000	0.0000

Test Metrics:
Accuracy: 19.31%
Average Precision: 0.0576
Average Recall: 0.0866
Average F1 Score: 0.0580

Per-Class Metrics:

Class	Precision	Recall	F1 Score
0	0.0000	0.0000	0.0000
1	0.2100	0.1875	0.1981
2	0.1370	0.4253	0.2072
3	0.1111	0.0133	0.0238
4	0.0533	0.0621	0.0574
5	0.1304	0.0190	0.0332
6	0.0000	0.0000	0.0000
7	0.0000	0.0000	0.0000
8	0.0000	0.0000	0.0000
9	0.0000	0.0000	0.0000
10	0.1266	0.6303	0.2108
11	0.3209	0.3476	0.3337
12	0.1643	0.2478	0.1976
13	0.4505	0.8222	0.5821
14	0.0000	0.0000	0.0000
15	0.0000	0.0000	0.0000
16	0.0000	0.0000	0.0000
17	0.0000	0.0000	0.0000
18	0.0113	0.0051	0.0071
19	0.0000	0.0000	0.0000
20	0.0000	0.0000	0.0000
21	0.0000	0.0000	0.0000
22	0.0000	0.0000	0.0000
23	0.0000	0.0000	0.0000
24	0.0000	0.0000	0.0000
25	0.3143	0.0458	0.0800
26	0.0000	0.0000	0.0000
27	0.0000	0.0000	0.0000
28	0.0000	0.0000	0.0000
29	0.0000	0.0000	0.0000
30	0.0000	0.0000	0.0000
31	0.0000	0.0000	0.0000
32	0.0000	0.0000	0.0000
33	0.0000	0.0000	0.0000
34	0.0000	0.0000	0.0000
35	0.1968	0.1872	0.1919
36	0.0000	0.0000	0.0000
37	0.0000	0.0000	0.0000
38	0.2504	0.7304	0.3729
39	0.0000	0.0000	0.0000
40	0.0000	0.0000	0.0000
41	0.0000	0.0000	0.0000
42	0.0000	0.0000	0.0000

```
Test Metrics:
Accuracy: 10.82%
Average Precision: 0.0168
Average Recall: 0.0450
Average F1 Score: 0.0181

Per-Class Metrics:
Class | Precision | Recall | F1 Score
 0    | 0.0000    | 0.0000 | 0.0000
 1    | 0.0831    | 0.6403 | 0.1471
 2    | 0.1082    | 0.4693 | 0.1759
 3    | 0.0000    | 0.0000 | 0.0000
 4    | 0.0000    | 0.0000 | 0.0000
 5    | 0.0000    | 0.0000 | 0.0000
 6    | 0.0000    | 0.0000 | 0.0000
 7    | 0.0000    | 0.0000 | 0.0000
 8    | 0.0000    | 0.0000 | 0.0000
 9    | 0.0000    | 0.0000 | 0.0000
10    | 0.1226    | 0.5788 | 0.2023
11    | 0.0000    | 0.0000 | 0.0000
12    | 0.0000    | 0.0000 | 0.0000
13    | 0.1667    | 0.0042 | 0.0081
14    | 0.0000    | 0.0000 | 0.0000
15    | 0.0000    | 0.0000 | 0.0000
16    | 0.0000    | 0.0000 | 0.0000
17    | 0.0000    | 0.0000 | 0.0000
18    | 0.0000    | 0.0000 | 0.0000
19    | 0.0000    | 0.0000 | 0.0000
20    | 0.0000    | 0.0000 | 0.0000
21    | 0.0000    | 0.0000 | 0.0000
22    | 0.0000    | 0.0000 | 0.0000
23    | 0.0000    | 0.0000 | 0.0000
24    | 0.0000    | 0.0000 | 0.0000
25    | 0.0000    | 0.0000 | 0.0000
26    | 0.0000    | 0.0000 | 0.0000
27    | 0.0000    | 0.0000 | 0.0000
28    | 0.0000    | 0.0000 | 0.0000
29    | 0.0000    | 0.0000 | 0.0000
30    | 0.0000    | 0.0000 | 0.0000
31    | 0.0000    | 0.0000 | 0.0000
32    | 0.0000    | 0.0000 | 0.0000
33    | 0.0000    | 0.0000 | 0.0000
34    | 0.0000    | 0.0000 | 0.0000
35    | 0.0000    | 0.0000 | 0.0000
36    | 0.0000    | 0.0000 | 0.0000
37    | 0.0000    | 0.0000 | 0.0000
38    | 0.2421    | 0.2435 | 0.2428
39    | 0.0000    | 0.0000 | 0.0000
40    | 0.0000    | 0.0000 | 0.0000
41    | 0.0000    | 0.0000 | 0.0000
42    | 0.0000    | 0.0000 | 0.0000
```