CW 1: Decision Trees for Letter Classification

MSc Computing: Introduction to Machine Learning
`Imperial College`

February 9, 2025

Introduction

This report details the implementation of a Decision Tree classifier from scratch in Python. The objective was to classify black-and-white pixel images into one of several letters in the English alphabet.

Part 1: Loading and examining the datasets

Question 1.1: A comparison of `train_full.txt` and `train_sub.txt`

The full training dataset, `train_full.txt`, contained 3900 instances, while `train_sub.txt` was a subset with only 600 instances ($\approx 15\%$ of the data). The primary difference between the two was the frequency distribution of the different classes (i.e., characters); `train_full.txt` had an approximately uniform distribution of frequencies, whereas `train_sub.txt` had a non-uniform distribution with a far greater number of instances for character 'C' and far fewer for characters 'G' and 'Q' (see Fig. 1).

Figure 1: A bar chart displaying the frequency distribution of classes in datasets `train_full.txt` and `train_sub.txt`.

These observations were quantified by calculating the standard deviation and range in the proportion of class label instances in each dataset (see Table 1).

Dataset	`train_full.txt`	`train_sub.txt`
Standard deviation	0.01	0.09
Range	0.02	0.28

Table 1: A comparison of standard deviation and range for datasets `train_full.txt` and `train_sub.txt` to analyse their spread.

It was clear from the calculations in Table 1, that the variability of `train_sub.txt` was far higher than that of `train_full.txt`, with a non-uniform distribution of the different class instances. This suggested that using the dataset `train_sub.txt` to train the decision tree would produce a worse classifier than training it on `train_full.txt`, as whilst it may perform excellently on one class, this would come at the expense of greatly reduced accuracy in other classes (see section 2.2 for further analysis).

Question 1.2: A brief discussion of the attributes in `train_full.txt`

The dataset attributes were (discrete) integer values, with the ranges for each in `train_full.txt` being between 0 and 15. All attributes represented pixel-based features extracted from preprocessed letter images.

Question 1.3: A comparison of `train_full.txt` and `train_noisy.txt`

Comparing `train_noisy.txt` and `train_full.txt`, it was observed that 598 of the 3900 total labels differed due to corruption from automatic classification. This represented $\approx 15\%$ of the labels. As a result, the class distribution was shifted, with some classes having greater or fewer instances in `train_noisy.txt` (see Fig. 2).

Figure 2: A bar chart displaying the frequency distribution of classes in `train_full.txt` and the corrupted dataset `train_noisy.txt`.

An analysis shows that the characters 'C' and 'G' have slightly fewer instances in `train_noisy.txt`, whereas the characters 'A', 'E', 'O' and 'Q' have an increased number of instances. However, it is clear by observation that the frequency distribution in both datasets is approximately uniform.

Part 2: Decision Tree Implementation

Task 2.1

2.1.1 Implementation

In the implementation of the decision tree classifier a bottom-up approach was adopted

Since the classifier was designed as a binary tree, a `Node` class was first created, with each instance representing a node in the decision tree. This class contained five attributes: `label` (the class label of a node); `feature` (the feature at which to split the sample); `threshold` (the threshold at which to split the sample); `left` (A pointer to the left sub-node; and `right` (a pointer to the right sub-node). Of these, the value of `label` would be `None` for all non-terminal nodes. In contrast, the values of the other attributes would only be `None` for a terminal leaf node.

Subsequently the `DecisionTreeClassifier` class was developed and Information Gain was chosen as the splitting criterion for selecting an optimal node. To support this, a helper function, `entropy(labels)`, was created to compute the information entropy for a given set of labels. This calculation was straightforward and was performed using the following formula:

$$H(S) = -\sum_{c=1}^{C} p_c \cdot \log_2(p_c) \tag{1}$$

Next, a second helper function, `information_gain(parent_labels, left_labels, right_labels)`, was developed to compute the information gain when splitting a node in the decision tree. This calculation was performed using the following equation:

$$H'(\bar{S}) = \sum_{i \in S} \frac{|S_i|}{N} H(S_i) \tag{2}$$

After implementing the functions to calculate the information gain for a given splitting, the next step was to create a function to determine the optimal split based on attributes and class labels in a given dataset. To prevent overfitting, two key parameters were incorporated: `min_elements_in_subset`, which prevented splits if a node contains fewer than the prescribed minimum instances, and `min_impurity_decrease`, which prevented splits if the optimal information gain for further splitting fell below a specified threshold. While default values were assigned to these parameters at this stage, they were created to allow for easy implementation of hyper-parameter tuning later on (see section 4). There was also a consideration to implement a `max_depth` parameter, but it was deemed more appropriate to incorporate it into a higher-level `build_tree()` function rather than the `best_split` function.

Following this, the `build_tree()` method, called by `DecisionTreeClassifier.fit()` on the training data to build the decision tree, was designed. As mentioned above, `max_depth`, a parameter which limited the depth of the decision tree, was introduced to prevent overfitting, effectively pre-pruning it. A high-level overview of our algorithm is outlined below:

Algorithm 1 Build Decision Tree

```
function BUILD_TREE(attributes, labels, depth=0)
    if max depth is reached or all samples have the same label or dataset has reached a preset minimum size then
        return a leaf node with the majority label
    end if
    (feature, threshold) ← BEST_SPLIT(attributes, labels)
    if no valid split was found then
        return a leaf node with the majority label
    end if
    (left_labels, right_labels) ← SPLIT_NODE(feature, threshold)
    left_subtree ← BUILD_TREE(left_attributes, left_labels, depth + 1)
    right_subtree ← BUILD_TREE(right_attributes, right_labels, depth + 1)
    node ← NODE(feature, threshold, left_subtree, right_subtree)
    return node
end function
```

Finally, the `fit()` method was implemented to validate the input data format, construct the decision tree by calling `build_tree()` on `root` (an attribute of `DecisionTreeClassifier`), and set a flag indicating that the classifier had been trained.

2.1.2 Design Decisions

The implementation of the decision tree classifier involved several key design choices to ensure efficiency, interpretability, and generalisation.

Since the data consisted of discrete, integer-valued attribute values (0 to 15), it could be treated as either real-valued or ordinal integers, and there was an option of implementing either a binary or a multiway tree. Binary splitting was opted for since it is more computationally efficient. This is because it reduces the search space for optimal partitions, making training and prediction faster. Binary splits were well-suited for the data, allowing for threshold-based decisions. Further, binary splitting has a lower risk of overfitting compared to multiway splitting, because it allows for more granular, step-by-step decisions rather than introducing complex partitioning for each step.

The `build_tree()` method builds the decision tree recursively, splitting at each decision node based on an optimal feature and threshold. Recursion allows the model to adapt dynamically to patterns in the data, forming a hierarchical structure of decision nodes until stopping criteria are met. This approach maintained computational efficiency, whilst ensuring a systematic division of the dataset.

As outlined in section 2.1.1, multiple stopping criteria were incorporated to prevent overfitting to the training data and improve generalisation of the model. `max_depth` limited the depth of the tree during training, acting as a form of pre-pruning. `min_elements_in_subset` ensured that splits were only performed if a node contained at least a predefined number of instances, preventing overly fine-grained splits. `min_impurity_decrease` was a threshold-based stopping condition that prevented further splitting of a branch if the reduction in impurity (measured by information gain) fell below a predefined value. Additionally, if all instances in a node belonged to the same class, then the node was designated as a leaf. These criteria balanced model flexibility with robustness, mitigating the risk of overfitting while preserving classification accuracy.

To enhance performance and scalability, various technical optimisations were incorporated.

NumPy arrays were utilised for efficient numerical computations and matrix operations, minimising computation time. Functions such as `entropy()` and `information_gain()` were also implemented using NumPy operations to avoid costly Python loops. Additionally, the tree was stored using linked node references rather than an explicit matrix representation, optimising space complexity.

Finally, to facilitate model performance evaluation and debugging, several attributes were implemented to track key tree characteristics. `depth`, `num_nodes` and `num_leaves` dynamically tracked the tree's depth, the number of nodes and the number of leaf nodes respectively during tree construction by the `build_tree()` function. These attributes are indicators of model complexity and potential overfitting, and allow for optimisations such as post-pruning.

Task 2.2

The `predict()` method traverses the constructed tree to assign labels to new instances. The implemented design supported dynamic input dimensions in the training stage (see section 2.1) to ensure flexibility, and allowed for optimisations such as hyperparameter tuning and pruning to further improve the model. Since decision trees require prior training, the `predict()` method first asserts that the classifier has been trained before making predictions in order to prevent errors. It then iterates through each new instance and traverses the decision tree's structure recursively until it reaches a terminal node. This is done by calling the recursive function `predict_sample()`. If the node is a leaf, the corresponding class label is returned; otherwise, the decision rule is applied to determine the next branch to follow. The design ensures that the implementation aligns with LabTS evaluation requirements while maintaining efficiency and interpretability, ensuring robust generalisation to unseen data.

Part 3: Evaluation

Question 3.1

To evaluate the impact of training data size and quality on model performance, three separate decision trees were trained on datasets `train_full.txt`, `train_sub.txt`, and `train_noisy.txt`. Following this, the model was evaluated using the `test.txt` dataset. The metric used for evaluation were: confusion matrices, accuracy, recall, precision, F_1 scores, and the macro-averaged values of the aforementioned three metrics.

Accuracy Comparison

Dataset	`train_full.txt`	`train_sub.txt`	`train_noisy.txt`
Accuracy	0.865	0.730	0.795

Table 2: Accuracy of models trained using the datasets `train_full.txt`, `train_sub.txt`, and `train_noisy.txt`.

As expected, the full dataset achieved the highest accuracy, confirming that model performance is improved with a larger training dataset and a uniform class distribution. Notably, the noisy dataset yielded higher accuracy than the subset despite the presence of mislabelled data. This suggested that a larger dataset, even with some noise, can be more advantageous for model performance

than training on a smaller uncorrupted dataset, though this performance may not be consistent across different classes (see below). Furthermore, this indicates that the decision tree algorithm is relatively robust to noise, as it generalises well from additional training data, even when some contain incorrect labels. However, it is necessary to state that there will clearly be a point at which corruption in the labelling of a dataset offsets the benefits of an increased dataset size.

Confusion Matrix Comparison

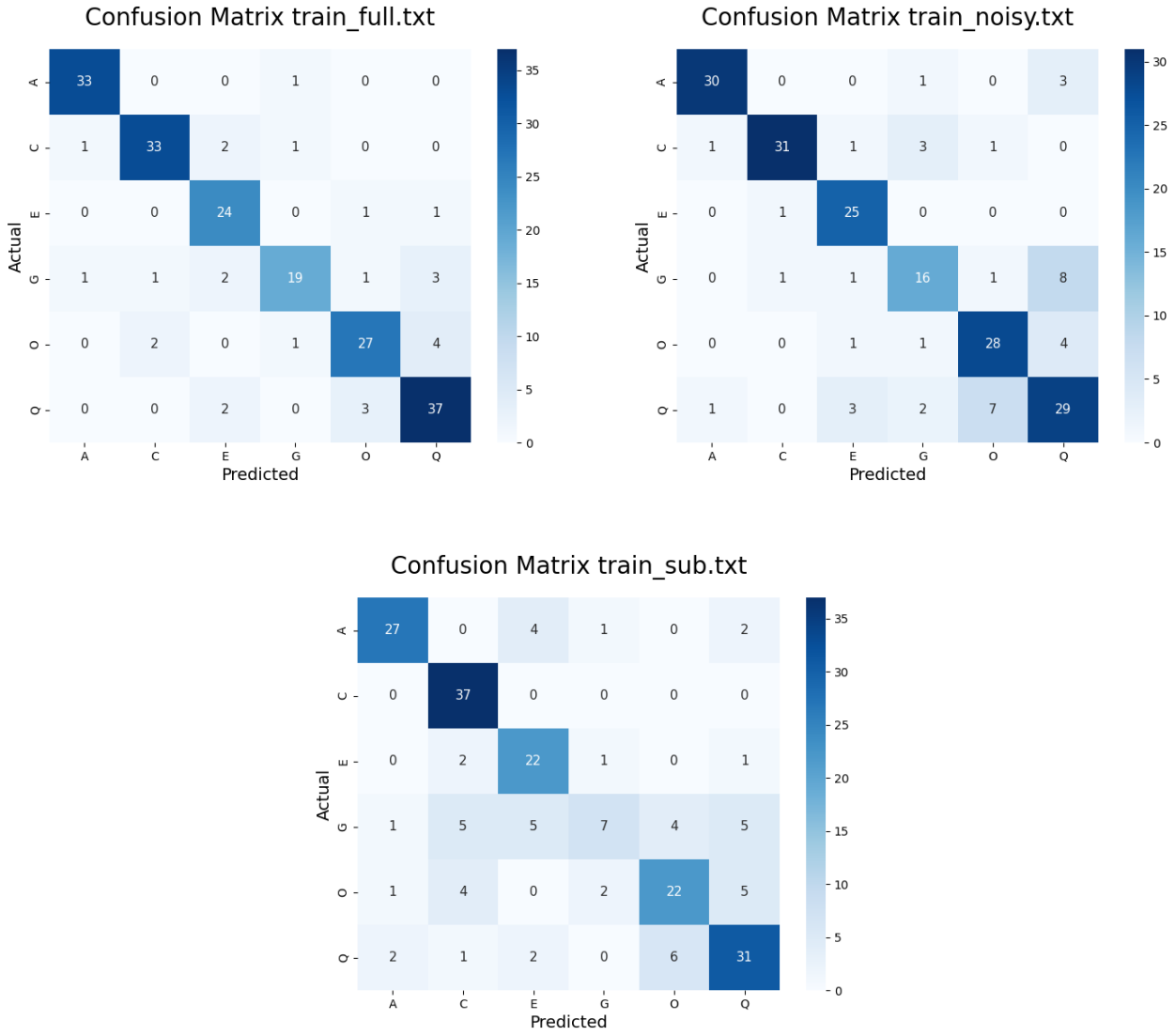

Figure 3: Confusion matrix heat maps for the `train_full.txt` (top left), `train_noisy.txt` (top right) and `train_sub.txt` (bottom) datasets.

The confusion matrix for the full dataset displays high values along the main-diagonal for 'A', 'C' and 'Q', indicating that these classes were largely correctly classified (see Fig. 3). Conversely, 'G' has a value a lower relative value, as shown in the heat map, indicating less accurate classification of this class. Off-diagonal values are mostly 0 or very low, indicating relatively few misclassifications, though there is some notable confusion between certain classes, particularly 'O'-'Q'. This suggests that the model struggles to differentiate between these two categories, possibly due to similarities in their feature distributions.

For the subset dataset, classes 'C' and 'Q' have the highest relative main-diagonal values, indicating that these labels are generally correctly classified, while G again has the lowest relative value, indicating poor classification. The off-diagonal values are generally low, with the exception significant confusion between 'Q' and every other class except 'A' (with confusion values of 4 and 5).

In comparison to the other datasets, the noisy dataset has very high main-diagonal values, though notably, 'G' is often incorrectly classified as 'Q'. Since most off-diagonal values are low, this misclassification pattern suggests that the noisy dataset struggles more with distinguishing between certain letter pairs compared to the full dataset. Further, it is now clear from analysis of all three confusion matrices, that in every dataset, the class 'G' is general is very poorly classified (and that this is exacerbated in the noisy dataset due to lack of 'G' labels for training).

Recall, precision, and F1 score per class

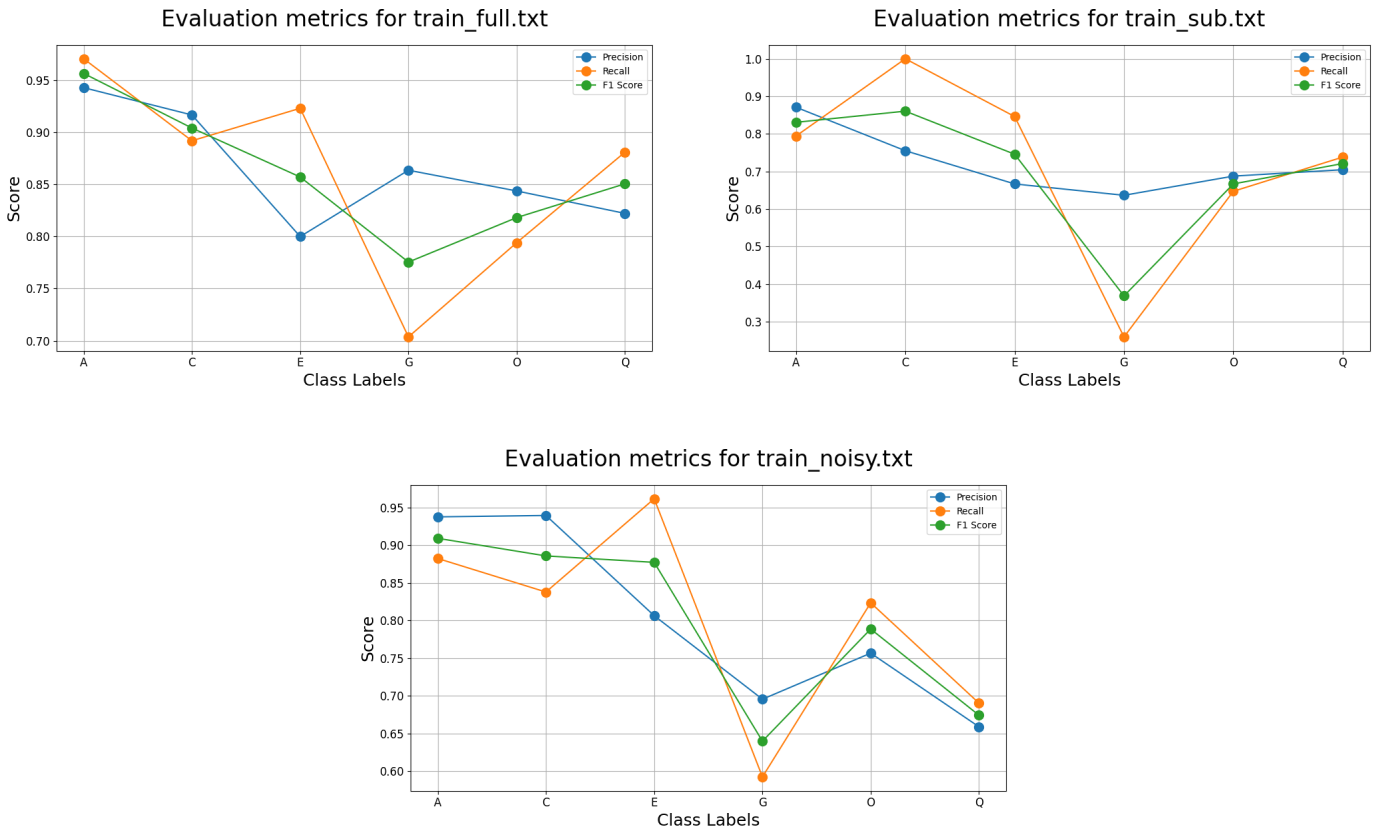

Figure 4: Graphs showing precision, recall and F1 score per class for `train_full.txt` (top left), `train_sub.txt` (top right) and `train_noisy.txt` (bottom) datasets

As seen in Fig. 4, `train_full.txt` shows the most consistent performance across the classes, achieving high macro-averaged precision, recall and F1 scores of 0.86 for all three metrics. It performed particularly well on classes `'A'` and `'C'` with F1 scores greater than 0.90, indicating that these classes are being accurately identified whilst minimising false positives and false negatives. Meanwhile, class `'G'` shows relatively weaker performance with the lowest F1 score of 0.78 and precision of 0.70.

There are significant variations in metrics across classes for `train_sub.txt`. Class `'C'` has the highest precision of 1 but with a lower recall of 0.75. This means that although most `'C'` predictions are correct, around a quarter of `'C'` instances are misclassified as other classes. This

could potentially be due to the model learning very distinct features of `'C'`, as it hasn't been exposed to enough data to learn from. Conversely, `'G'` is the worst performing class with a precision of 0.25, indicating the subset model struggles with distinguishing class `'G'` from other letters. Additionally, classes `'A'` and `'E'` generally have balanced performance metrics ranging between 0.75 and 0.85, while the metrics for `'O'` and `'Q'` converge around 0.70. The varying performance across classes highlights how limited training data affects the model's ability to learn robust letter representations.

Despite the added noise, the model maintains relatively good performance for most classes, suggesting robustness to noise. Class `'G'` shows the poorest performance with precision and recall values of 0.59 and just under 0.70. The consistent poor performance of this class amongst the datasets shows inherent difficulty for the model to distinguish this letter. Furthermore, class `'E'` has the highest precision value of 0.97 but a significantly lower recall of 0.81, indicating that the model is not correctly classifying a significant portion of correct `'E'` instances.

Overall, the full dataset shows the most consistent performance across classes, whilst the noisy dataset shows most pronounced variations between classes. Amongst all the datasets, G is the worst performing class, indicating that this class may be inherently harder to distinguish.

Macro-averaged Metrics

Dataset	`train_full.txt`	`train_sub.txt`	`train_noisy.txt`
Macro-averaged Precision	0.86	0.72	0.80
Macro-averaged Recall	0.86	0.71	0.80
Macro-averaged F1	0.86	0.70	0.80

Table 3: Macro-averaged metrics for datasets `train_full.txt`, `train_sub.txt`, and `train_noisy.txt`.

As mentioned above, `train_full.txt` performs best overall with a macro-averaged precision, recall and F1 values of 0.86. This indicates consistent performance across all classes and the close values between precision and recall suggests the model makes balanced predictions without favouring either false positives or negatives.

`train_sub.txt` shows the poorest performance, with a macro-averaged precision of 0.72, recall of 0.71 and F1 value of 0.70. The lower macro-averaged metric scores reflect the model's inconsistent performance across classes, it particularly struggles with class 'G'. This is likely due to the reduced training data which makes it harder for the model to learn class representations in a robust manner.

`train_noisy.txt` gives moderate macro-averaged metrics, with a precision, recall and F1 values of 0.80. The noisy dataset outperforms the subset model, indicating that more training data is better than less data without the noise.

Question 3.2

The dataset was split into 10 equal folds to evaluate the performance of the Decision Tree Classifier using 10-fold cross-validation. This ensured that the model's performance was assessed across multiple train/test splits and was not dependent on a single dataset division. In each iteration, the model was trained on 9 folds and tested on the remaining fold. This process was repeated 10 times ensuring each data point was used for testing exactly once. The average accuracy obtained across the 10 folds was 92.31% $\pm$ 0.01%, where the standard deviation (given as the error) reflects the consistency of the model's performance across the folds.

The high average accuracy indicates that the decision tree performs well on this dataset and should generalise well to unseen data, provided the dataset is representative of the broader problem domain. The low standard deviation suggests that the performance is consistent across different training and testing splits. In contrast, a high standard deviation would imply variability in accuracy across folds, which could be a sign of sensitivity to specific train/test splits, potentially due to overfitting, data imbalance, or insufficient data.

Question 3.3

To evaluate performance on the unseen test set (`test.txt`), predictions from the 10 decision trees, trained during the 10-fold cross-validation process, were combined using majority voting. This involved the mode predicted class label being selected as the final output for each instance in the test set.

This ensemble approach achieved an accuracy of 90.5% on the test set, outperforming the 86.5% accuracy obtained with a single decision tree trained on the entire `train_full.txt` dataset (see Section 3.1) and analysis of the confusion matrix confirmed fewer misclassifications. The improvement demonstrates the effectiveness of ensemble methods in reducing variance and smoothing out errors made by individual models, highlighting its reliability compared to a single decision tree.

Part 4: Enhancements

Task 4.1

To improve the decision tree model, two main approaches were initially considered: refining the existing decision tree or constructing an entirely new one with an alternative structure, such as multiway splits instead of binary splits. However, given that the current decision tree had already achieved a reasonable performance with 86.5% accuracy with randomly selected parameters, and 90.5% accuracy with majority voting, it was decided the focus should be on optimising the existing model. Several strategies were identified to enhance the classifier's performance, including hyperparameter tuning, alternative splitting criteria (e.g. Gini Index), and post-pruning techniques. Hyperparameter tuning was selected as the primary focus, as it provided a structured and systematic approach to refining the decision tree's performance. The parameters optimised were `max_depth`, `min_sample_split` and `min_impurity_decrease` (see Section 2 for further discussion).

The `train_and_predict` function encapsulated the training and evaluation process by integrating grid search and cross-validation. A 10-fold cross-validation approach was applied to the full dataset, ensuring a comprehensive evaluation by training and validating on multiple subsets. The `grid_search` function systematically iterated through a Cartesian product of all hyperparameter combinations for preset ranges, before selecting the best-performing mode based on the highest average accuracy across the 10 folds. Two models were constructed using these optimal parameters. The first was a single decision tree, trained on the entire dataset (either train_full.txt or train_noisy.txt) with the best hyperparameters, before making predictions on the `text.txt` dataset. The second model utilised the ensemble of 10 decision trees trained using the best-performing parameters to all make predictions on the `text.txt` dataset, before aggregating the predictions using majority voting. Finally, both models' performances were evaluated by comparing its predictions to the ground truth labels in `y_test`, including accuracy and other performance metrics, which were displayed to assess whether the hyperparameter tuning and majority voting approach enhance the model's reliability compared to previous methods.

Question 4.1

The best performing parameters were identified as a `max_depth` of `None`, a `min_sample_split` of 6, and a `min_impurity_decrease` of 0.055. These had an average accuracy of 92.5% across the 10 folds, highlighting its consistent performance across different data splits.

The single decision tree achieved an accuracy of 89% for the full dataset and 82% for the noisy dataset, both representing a 2.5% increase on their Part 2 accuracy. The higher minimum sample split ensured that each split had a sufficient number of samples, preventing overly specific splits that possibly lead to overfitting previously. Similarly, the minimum impurity decrease acted as a safeguard against unnecessary splits, ensuring that a split only occurred when it significantly reduced impurity, which helped in maintaining effective decision boundaries across different dataset variations. The increase to 82% on the noisy dataset further indicates that the tree was more resistant to noise, likely because the adjusted impurity threshold prevented overfitting to noisy patterns. In contrast, in Part 2, the model may have been too flexible, allowing it to create overly complex branches that captured noisy data, leading to weaker generalisation.

The majority voting model achieved an accuracy of 91.5% on the test dataset when trained using `train_full.txt`. This improvement demonstrates the effectiveness of ensemble methods in reducing the variability of individual model predictions and achieving better generalisation. While the improvement is partly due to stronger individual models, another key factor is the diversity among trees. The ensemble benefits from combining predictions from slightly different models, reducing variance and mitigating overfitting to specific data patterns. Thus, even if an individual tree misclassifies an instance other models in the ensemble can correct it, leading to a more robust overall performance.

Interestingly the ensemble model did not outperform the single tree when trained on the noisy dataset, maintaining the same 82% accuracy. This suggests that noise was present across all training samples, meaning individual decision trees still learned similar noisy patterns despite majority voting sharing the same biases. As a result majority voting could not correct errors as effectively as it did for the clean dataset.

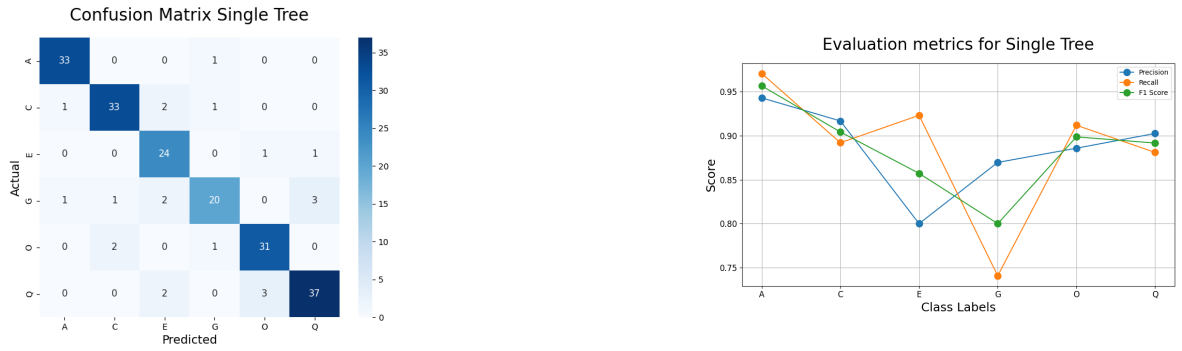

Figure 5: Confusion matrix (left) and evaluation metrics (right) for the single decision tree trained on the full dataset after hyperparameter tuning, illustrating improved classification performance.

References

[1] J.R. Quinlan, "Induction of Decision Trees", Machine Learning, 1986.

[2] L. Breiman et al., "Classification and Regression Trees", 1984.