– Part 1  
**Question 1.1**

Besides the number of instances, what is another main difference between train\_full.txt and train\_sub.txt?

The main difference is that train\_sub.txt is rather unbalanced dataset in comparison to train\_full.txt. From figure 1, it can be observed that all the labels are well represented within the range from 599 to 671 observations. This makes the dataset relatively balanced. On the other hand, when examining figure 2, we can see that the representation of each label is not uniform, ranging from 21 up to 187 observations. This makes this dataset relatively unbalanced.

There is no large variation between the minimum and maximum values of each column between the datasets. The same holds true for means and standard deviation, which means it should not influence the result too much.

Figure 1: Train\_full.txt:

Text

Description automatically generated

Figure 2: Train\_sub.txt

Text

Description automatically generated

**Question 1.2**

What kind of attributes are provided in the dataset (Binary? Categorical/Discrete? Integers? Real numbers?) What are the ranges for each attribute in train\_full.txt?

All the attributes are integers. The class variable contains categorical/discrete strings. However, the way we built our decision tree is by treating the integers as real values.

The ranges for each attribute, labelled from left to right using integers, e.g., 0, 1, 2 etc., can be found in table 1.

Table 1: ranges of each attribute for train\_full.txt

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Attribute index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |  |  |  |
| Min value | 0 | 0 | 1 | 0 | 0 | 2 | 0 | 0 | 0 | 3 | 0 | 4 | 0 | 1 | 0 | 1 |
| Max value | 10 | 15 | 11 | 12 | 14 | 14 | 14 | 10 | 12 | 13 | 11 | 15 | 12 | 15 | 11 | 14 |

**Question 1.3**

Train\_noisy.txt is actually a corrupted version of train\_full.txt, where we have replaced the ground truth labels with the output of a simple automatic classifier. What proportion of labels in train\_noisy.txt is different than from those in train\_full.txt? (Note that the observations in both datasets are the same, although the ordering is different). Has the class distribution been affected? Specify which classes have a substantially larger or smaller number of examples in train\_noisy.txt compared to train\_full.txt.

The distribution has not been significantly affected. Both C and G labels showed the highest difference of 0.008 and 0.013 respectively.

Figure 3:

**Text, table

Description automatically generated with medium confidence**

Figure 4:

Chart, bar chart

Description automatically generated

# – Part 2

**Task 2.1**

We divided this problem into smaller tasks which we have solved by implementing the corresponding helper functions: *calculate\_entropy, make\_opposite\_filter, calculate\_best\_info\_gain* and *split\_by\_best\_rule*. Those functions were then used in a recursive function: *induce\_tree*. The following paragraphs shed some light on the design decisions and why they were made.

**calculate\_entropy**

A helper procedure whose purpose is to calculate the entropy for a given slice of the dataset. It has time complexity, where N is the number of observations in the given slice of the dataset. It is used in *calculate\_best\_info\_gain* function.

**make\_opposite\_filter**

A helper producer that returns the inverted NumPy array of False and True values. It has time complexity, where N is the number of observations in the given array. It is also used in *calculate\_best\_info\_gain* function.

**calculate\_best\_info\_gain**

This is a procedure which iteratively keeps splitting the dataset by both features (column indices) and values (integers in each column). The splitting is binary and for each split, it calculates the information gain. Return is the feature index and the integer value, whose combination gave us the highest information gained. In other words, this function iteratively calculates the best splitting decision based on the highest information gain.

The design decision was to use two loops. The First outer loop iterates over all features, the inner loop iterates over all unique values of a given feature column. The whole procedure has an upper bound of where N is the number of observations. In a case where the number of features is larger or equal to the number of observations, the upper bound would be , however, for our datasets where the highest number of features was 16, linear time complexity can be assumed.

We were careful to use NumPy methods to manipulate the dataset to be as efficient as possible. For instance, numpy.unique method was used to not only obtain the unique integer values of a given column, but also sort them in a convenient way which was used for the inner loop. This design decision does not waste a single iteration; the inner loop does not iterate over anything it does not need to, skipping integers that are not present and not iterating over a distinct integer more than once. Furthermore, the numpy.unique method is more efficient than simply sorting the array or looking for minimum and maximum values and using the range object.

This function is used in recursive function, *induce\_tree*.

**split\_by\_best\_rule**

This procedure takes as input the dataset, and the feature index and the integer value whose combination maximises the information gain. It returns two datasets which are split according to feature index and the integer value*.*

It has time complexity, where N is the number of observations in the input dataset. NumPy arrays and operations were used to split the dataset efficiently.

**induce\_tree**

So far, all we have done was for one binary split. This procedure recursively repeats the operations above. The time complexity of the recursion itself is since we have binary splits, which reduces the database with each call. Each recursive call calls procedures, discussed above, whose total time complexity is , unless the number of features exceeds the number of observations (does not occur in the datasets provided). The overall time complexity of *induce\_tree* is then .

**Task 2.2**

We are storing the model in a class called Node, through which we do the predictions. The Node stores the Child Nodes (left and right), feature index and split value by which the splitting is done. Furthermore, it contains classification attribute which is set to a class label if it is leaf. Finally, we store all y-values that are carried through in the data variable, so a majority can be computed for pruning.

The test data is passed through the Node to arrive at a prediction, using the predict method.

# – Part 3

**Question 3.1**

Confusion Matrix for train\_full.txt:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Actual/Predict** | **A** | **C** | **E** | **G** | **O** | **Q** |
| **A** | **33** | **0** | **0** | **0** | **1** | **0** |
| **C** | **0** | **34** | **2** | **1** | **0** | **0** |
| **E** | **0** | **0** | **25** | **0** | **1** | **0** |
| **G** | **0** | **2** | **0** | **23** | **0** | **2** |
| **O** | **0** | **1** | **0** | **0** | **30** | **3** |
| **Q** | **0** | **0** | **2** | **0** | **3** | **37** |

Confusion Matrix for train\_sub.txt

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Actual/Predict** | **A** | **C** | **E** | **G** | **O** | **Q** |
| **A** | **29** | **1** | **2** | **0** | **0** | **2** |
| **C** | **0** | **36** | **0** | **1** | **0** | **0** |
| **E** | **0** | **2** | **23** | **1** | **0** | **0** |
| **G** | **1** | **6** | **7** | **7** | **4** | **2** |
| **O** | **1** | **1** | **1** | **3** | **24** | **4** |
| **Q** | **1** | **2** | **0** | **1** | **5** | **33** |

Confusion Matrix for train\_noisy.txt:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Actual/Predict** | **A** | **C** | **E** | **G** | **O** | **Q** |
| **A** | **30** | **0** | **0** | **0** | **1** | **3** |
| **C** | **1** | **31** | **0** | **3** | **2** | **0** |
| **E** | **0** | **2** | **24** | **0** | **0** | **0** |
| **G** | **0** | **1** | **1** | **16** | **1** | **8** |
| **O** | **0** | **1** | **0** | **1** | **28** | **4** |
| **Q** | **2** | **0** | **3** | **3** | **5** | **29** |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **All** | **Accuracy** | **Macro-averaged recall** | **Macro-averaged precision** | **Macro-averaged F1** |
| **train\_full.txt** | 0.910 | 0.911 | 0.913 | 0.911 |
| **train\_sub.txt** | 0.760 | 0.744 | 0.737 | 0.728 |
| **train\_noisy.txt** | 0.790 | 0.792 | 0.794 | 0.791 |

|  |  |  |  |
| --- | --- | --- | --- |
| **Class A** | **Precision** | **Recall** | **F1 Score** |
| **train\_full.txt** | 1.00 | 0.971 | 0.985 |
| **train\_sub.txt** | 0.906 | 0.853 | 0.869 |
| **train\_noisy.txt** | 0.909 | 0.882 | 0.900 |

|  |  |  |  |
| --- | --- | --- | --- |
| **Class C** | **Precision** | **Recall** | **F1 Score** |
| **train\_full.txt** | 0.920 | 0.920 | 0.920 |
| **train\_sub.txt** | 0.750 | 0.873 | 0.847 |
| **train\_noisy.txt** | 0.886 | 0.838 | 0.861 |

|  |  |  |  |
| --- | --- | --- | --- |
| **Class E** | **Precision** | **Recall** | **F1 Score** |
| **train\_full.txt** | 0.862 | 0.962 | 0.909 |
| **train\_sub.txt** | 0.697 | 0.885 | 0.780 |
| **train\_noisy.txt** | 0.857 | 0.923 | 0.889 |

|  |  |  |  |
| --- | --- | --- | --- |
| **Class G** | **Precision** | **Recall** | **F1 Score** |
| **train\_full.txt** | 0.958 | 0.852 | 0.902 |
| **train\_sub.txt** | 0.538 | 0.259 | 0.350 |
| **train\_noisy.txt** | 0.700 | 0.593 | 0.640 |

|  |  |  |  |
| --- | --- | --- | --- |
| **Class O** | **Precision** | **Recall** | **F1 Score** |
| **train\_full.txt** | 0.857 | 0.882 | 0.870 |
| **train\_sub.txt** | 0.727 | 0.706 | 0.716 |
| **train\_noisy.txt** | 0.757 | 0.824 | 0.789 |

|  |  |  |  |
| --- | --- | --- | --- |
| **Class Q** | **Precision** | **Recall** | **F1 Score** |
| **train\_full.txt** | 0.881 | 0.881 | 0.881 |
| **train\_sub.txt** | 0.805 | 0.686 | 0.795 |
| **train\_noisy.txt** | 0.659 | 0.690 | 0.674 |

Which one performs the best? The worst? Give some insights into why.

The model performed overall much better on the train\_full dataset. The model performed the worst on train\_sub dataset. The main reason is that train\_sub is a very unbalanced dataset in comparison to the other two.

Which classes are accurate?

The most accurate is Class A, followed by Class C.

Which classes are often confused? Which classes are they often confused as?

For train\_full most classes were correctly predicted. There are only a few false positives and false negatives for Q and O. For train\_noisy classes Q and O have the most false positives and false negatives.

Train\_sub has the most mispredictions. Class Q performed particularly bad, which coincides with low number of observations. Q only has 21 observations in comparison to the other classes which have around 150 to 200. From this we can generalise, the lower the ratio of the total number of observations for a given class, the lower the accuracy.

**Question 3.2**

Perform 10-fold cross validation on train\_full.txt.

Report the average accuracy across the 10 folds. Also report the standard deviation of the accuracies. For example, 0.7854 ± 0.0122.

What does it mean to have a small/large standard deviation in this context?

Having a small standard deviation translates into having smaller confidence interval. On the other hand, having a large standard deviation translates into having a larger confidence interval. Note that having more observations also contributes to obtaining smaller confidence interval, this can be observed from the calculation of standard error:

**Question 3.3**

Try combining the predictions on test.txt for all 10 decision trees. The decision tree could vote on the class label, that is you can select the mode (most frequent) of the predicted class labels across the 10 trees’ predictions. Does it perform better than training a single decision tree on the full train full.txt? Discuss.

Yes. The accuracy without cross validation was 0.91, whereas the majority label prediction from the cross validation results in a slightly higher accuracy of 0.94.

TEXT … DISCUSS

– Part 4  
**Task 4.1**

Briefly describe your proposed improvements or design, and your motivations/justifications for introducing these. Discuss and explain any design decisions you made and why you made those decisions.

Once upon a time, we had a very simple Decision Tree, then it occurred to us that maybe we could improve it by implementing Max Depth. Sadly, the dragon didn’t come out of the cave and the result did not improve at all, so we aimlessly wandered into the Random Forest and decided to Prune all the trees there.

We have tried the following three improvements:

Max Depth

Max depth puts a limit on the number of node levels of the tree. Through hyper-parameter tunning we cross validated the data to find the optimal number for the depth. However, this did not yield improved result. The best depth was 15 levels with an average accuracy of 0.913. When we tried this on the test dataset, we get an accuracy of 0.87.

Random Forest

The random forest is an algorithm consisting of many decisions trees. Our implementation uses feature randomness when building each individual tree to try to create an uncorrelated forest of trees whose prediction is more accurate than that of any individual tree.

Pruning

Pruning goes through each internal node that is connected to leaf nodes, replacing it with a leaf node based on the majority label. It replaces it if and only if the resulting accuracy is higher than without pruning.

**Question 4.1**

Analyse and discuss the results of testing on test.txt. How does your new model(s) perform compared to the model in Part 2? Discuss.

The final accuracy resulting from both Random Forest and Pruning implementation is 0.955 – 0.975. The average accuracy from the original model was 0.910.