Introduction to Machine Learning
— Coursework 1 Report —

Davide Locatelli, Benedetta Delfino,
Michael George, Ben Butterworth
Course: CO395, Imperial College London

February 10, 2020

Loading and Examining the Dataset

Task 1.1

- File name: `dataset.py`
- Class name: `Dataset`
- Implementation: the `dataset.py` script defines the Dataset class, which wraps attributes and labels stored as numpy arrays. A Dataset object can be initialised in two ways:
  - If the arrays are already in memory they can be passed to the constructor.
  - If the arrays need to be loaded from a file, this can be done by passing the file name to the class method `load_from_file()`. This method uses numpy's `loadtxt()` function and convenient array slicing to separate labels from attributes.

Example usage can be found at the end of the `dataset.py` file.

Question 1.1

- The attributes are positive real integers, and, more specifically, **ordinal values**. In fact, they have a natural ordering. For example, attribute 0 is "horizontal position of box" and attribute 1 is "vertical position of box".
- Overall, attribute values range from 0 to 15, but many attributes have smaller ranges as shown in Figure 1.

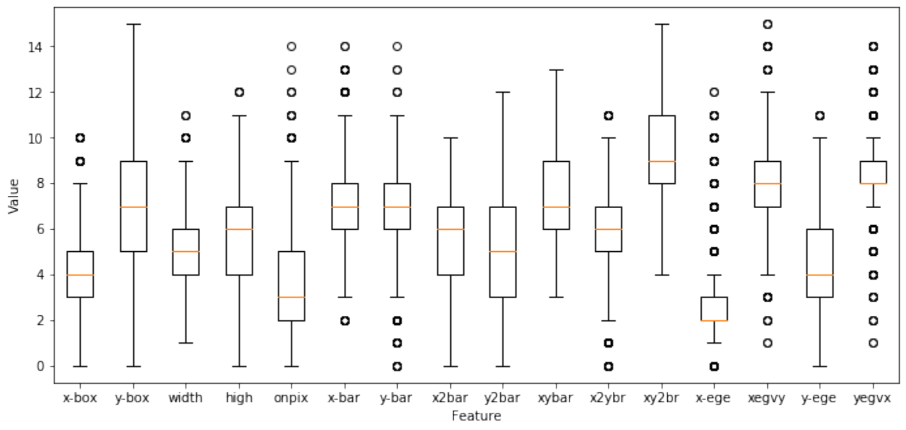


Figure 1: Value ranges for each attribute in `train_full.txt`

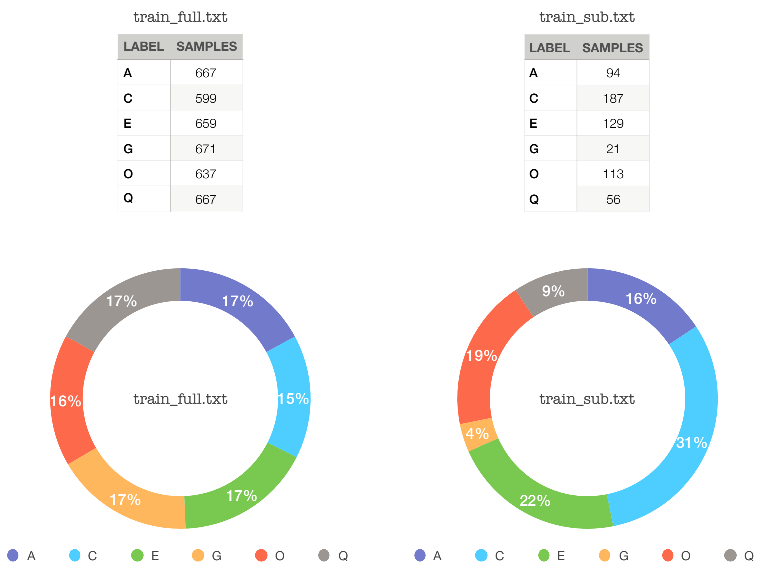
Question 1.2

- `train_sub.txt` has 600 samples, while `train_full.txt` has 3900 samples.
- Labels in `train_full.txt` are equally represented, but `train_sub.txt` labels 'G' and 'Q' are underrepresented in the data set.

The distribution of labels in the two files is represented in Figure 2.

Question 1.3

- The proportion of labels in `train_noisy.txt` different than from those in `train_full.txt` is 598/3900.
- The class distribution has not been affected in `train_noisy.txt`: the representation for each class in `train_noisy.txt` is similar to the corresponding class in `train_full.txt` as shown in Figure 3.
- There is a considerable difference in the number of mistakes for each label in `train_noisy.txt`, e.g. there were 28 mislabelled As but 198 mislabelled Gs. The proportion of misclassified samples for each label is shown in Figure 4.



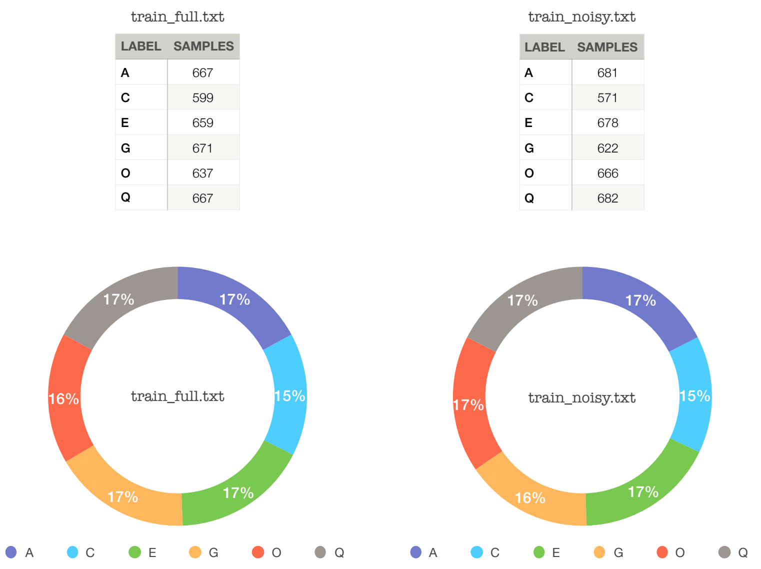
train_full.txt

LABEL	SAMPLES
A	667
C	599
E	659
G	671
O	637
Q	667

train_sub.txt

LABEL	SAMPLES
A	94
C	187
E	129
G	21
O	113
Q	56

Figure 2: Distribution of labels in `train_full.txt` and `train_sub.txt`



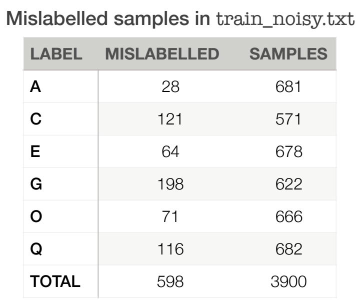
train_full.txt

LABEL	SAMPLES
A	667
C	599
E	659
G	671
O	637
Q	667

train_noisy.txt

LABEL	SAMPLES
A	681
C	571
E	678
G	622
O	666
Q	682

Figure 3: Distribution of labels in `train_full.txt` and `train_noisy.txt`



Mislabelled samples in train_noisy.txt

LABEL	MISLABELLED	SAMPLES
A	28	681
C	121	571
E	64	678
G	198	622
O	71	666
Q	116	682
TOTAL	598	3900

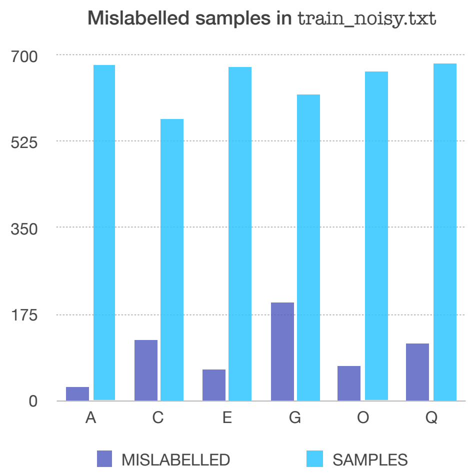


Figure 4: Misclassified samples in `train_noisy.txt`

Implementing Decision Trees

We have decided to implement the `DecisionTreeClassifier` as a binary tree because the attributes are real-valued integers. It seemed desirable for each split to create two children based on whether the attribute is greater/equal or less than the chosen threshold.

The end result is a `DecisionTreeClassifier` made of `Node` objects. A `Node` can either be a leaf or not. If it is a leaf, it stores a label. Otherwise, it stores a left and a right `Node`, the most frequent label of the dataset at that point, and a `Rule`. A `Rule` object describes how a `Node` splits the dataset. It holds both the attribute and the threshold value that the node uses to perform the split.

Generally, our `DecisionTreeClassifier` is generated by comparing the entropy of the parent dataset with the children datasets. All splits are considered but only the one with the highest information gain is actually executed. If two splits would lead to the same information gain, then our algorithm will take the last possible split with that information gain.

Task 2.1

- File name: `classification.py`
- Method names: `train()`, `create_tree()`
- Implementation: the `train()` function calls `create_tree()`. Both functions take two parameters: the array of attributes and the array of labels. `create_tree()` returns the root of the tree, which is stored in `self.root`. Finally, the whole `DecisionTreeClassifier` is returned by `train()`. `create_tree()` is a recursive function:
  - First it checks if the training labels only have one letter left, if all attributes are the same or if the highest information gain has value 0. If any one of these conditions hold, then we have reached the end of the subtree and so the leaf will acquire an attribute called label, which is the most frequent label of that dataset.
  - If none of the base cases above are satisfied, it proceeds to the recurrence. It uses the information gain to find the best split rule and create two children, splitting the attributes and labels according to the chosen rule. Lastly, it calls itself recursively on each child and links it to the parent via the attributes `left` and `right`. The function returns a `Node` object.

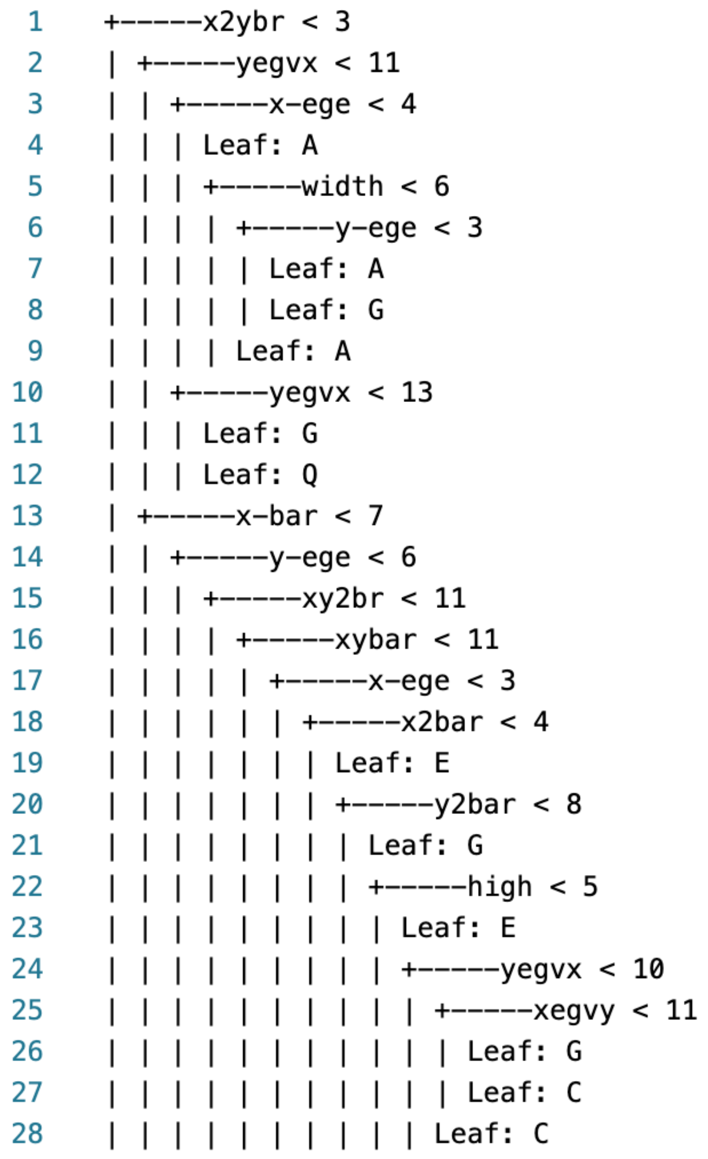
Task 2.2

- File name: `classification.py`

- Method names: `predict()`, `find_label()`
- Implementation: the `predict()` function takes the numpy array of test attributes as a parameter. It iterates through the rows of attributes, and stores the predicted label for that row. To do the latter operation, the function calls `find_label()`, passing to it the attributes array, the row being considered, and the root node of the tree. `find_label()` is a recursive function:
  - First it checks whether the node passed to it is a leaf. If it finds a leaf, it returns the label of that leaf.
  - Otherwise, the search for the label moves either to the right or the left of the tree, depending on the threshold value stored in the rule of the node.

Task 2.3

- File name: `classification.py`
- Method name: `print_tree()`, `print_subtree()`
- Example visualization:



```
+-----x2ybr < 3
| +-----yegvx < 11
| | +-----x-ege < 4
| | | Leaf: A
| | | +-----width < 6
| | | | +-----y-ege < 3
| | | | | Leaf: A
| | | | | Leaf: G
| | | | Leaf: A
| | +-----yegvx < 13
| | | Leaf: G
| | | Leaf: Q
| +-----x-bar < 7
| | +-----y-ege < 6
| | | +-----xy2br < 11
| | | | +-----xybar < 11
| | | | | +-----x-ege < 3
| | | | | | +-----x2bar < 4
| | | | | | | Leaf: E
| | | | | | | +-----y2bar < 8
| | | | | | | | Leaf: G
| | | | | | | | +-----high < 5
| | | | | | | | | Leaf: E
| | | | | | | | | +-----yegvx < 10
| | | | | | | | | | +-----xegvy < 11
| | | | | | | | | | | Leaf: G
| | | | | | | | | | | Leaf: C
| | | | | | | | | | Leaf: C
```

Figure 5: A section of the `DecisionTreeClassifier` trained on `train_full.txt` up to depth 10

Question 2.1

The tree should be read as follows:

- The string '+- - - - -' represents a node that contains a rule
- The number of '|'s represents the depth of a node
- The leaves for each depth are printed in order: first the left leaf (e.g. line 7) and then the right leaf (e.g. line 8)

The decision tree trained on `train_full.txt` has as its first split rule $x2ybr < 3$ (where $x2ybr$ is one of the given attributes). The left child has the following labels: 524 As, 3 Gs, 3 Qs. This reveals that by splitting on $x2ybr < 3$ we get a large majority of As. As such, the information gain is high: we are very likely to correctly identify an A in this child. This is understandable, as A is the most visually distinct in comparison to other letters. In fact, we get a leaf 'A' after only two further splits $yegvx < 11$ and $x-ege < 4$. The tree then separates the As that have not been captured with the previous splits from the other labels. Hence we see two more 'A' leaves on lines 7 and 9. It is also interesting to see how the tree identifies early on a split that divides some Gs from some Qs, two letters visually quite similar: $yegvx < 13$.

Evaluation

Question 3.1

Analysis of train_full.txt

Confusion Matrix:

$$\begin{bmatrix} 33 & 0 & 0 & 0 & 1 & 0 \\ 0 & 34 & 2 & 1 & 0 & 0 \\ 0 & 0 & 25 & 0 & 1 & 0 \\ 0 & 2 & 0 & 23 & 0 & 2 \\ 0 & 1 & 0 & 0 & 30 & 3 \\ 0 & 0 & 2 & 0 & 3 & 37 \end{bmatrix}$$

Accuracy = 0.91

Class	Precision	Recall	F1 Scores
A	1.0000	0.9706	0.9851
C	0.9189	0.9189	0.9189
E	0.8621	0.9615	0.9091
G	0.9583	0.8519	0.9020
O	0.8571	0.8824	0.8696
Q	0.8810	0.8810	0.8810
Macro-averaged	0.9129	0.911	0.9109

Analysis of train_sub.txt

Confusion Matrix:

$$\begin{bmatrix} 29 & 1 & 2 & 0 & 0 & 2 \\ 0 & 36 & 0 & 1 & 0 & 0 \\ 0 & 2 & 23 & 1 & 0 & 0 \\ 1 & 6 & 7 & 7 & 4 & 2 \\ 1 & 1 & 1 & 3 & 24 & 4 \\ 1 & 2 & 0 & 1 & 5 & 33 \end{bmatrix}$$

Accuracy: 0.76

Class	Precision	Recall	F1 Scores
A	0.9062	0.8529	0.8788
C	0.7500	0.9730	0.8471
E	0.6970	0.8846	0.7797
G	0.5385	0.2593	0.3500
O	0.7273	0.7059	0.7164
Q	0.8049	0.7857	0.7952
Macro-averaged	0.7373	0.7436	0.7279

Analysis of train_noisy.txt

Confusion Matrix:

$$\begin{bmatrix} 30 & 0 & 0 & 0 & 1 & 3 \\ 1 & 31 & 0 & 3 & 2 & 0 \\ 0 & 2 & 24 & 0 & 0 & 0 \\ 0 & 1 & 1 & 16 & 1 & 8 \\ 0 & 1 & 0 & 1 & 28 & 4 \\ 2 & 0 & 3 & 3 & 5 & 29 \end{bmatrix}$$

Accuracy: 0.79

Class	Precision	Recall	F1 Scores
A	0.9091	0.8824	0.8955
C	0.8857	0.8378	0.8611
E	0.8571	0.9231	0.8889
G	0.6957	0.5926	0.6400
O	0.7568	0.8235	0.7887
Q	0.6591	0.6905	0.6744
Macro-averaged	0.7939	0.7916	0.7914

Question 3.2

Which model performs best? Worst? Why?

- `train_full.txt` performs the best because it has the most number of samples to train on, as well as containing an even distribution of each label.
- `train_sub.txt` performs the worst because it does not hold an even distribution of labels from the data set: more specifically, 'G' and 'Q' are underrepresented. This is because an unrepresentative sample of `train_full.txt` was taken to form a smaller set of 600 samples, and minority labels accuracy are significantly affected by majority classes. Interestingly, although label 'O' was well represented in the data set, it had low precision and recall, caused by similarity in features with the underrepresented samples. More generally, having a smaller sample set will also decrease accuracy.
- `train_noisy.txt` has a similar number of samples to `train_full.txt`, but the trained model was significantly less accurate than the one from `train_full.txt`. The noise in the data set appears to have affected 'C', 'O', and 'G' most.

Which classes are accurate? Why?

- The distinct appearance of samples from a given label allows a class to be more accurately classified by the model. For example, class 'A' consistently has the best evaluation metrics, and has a visually distinct shape than 'O', 'E', 'C', 'G', and 'Q'.
- 'C' and 'E' have high performance metrics, but are lower than 'A'. This is because 'C' and 'E' have a few similar visual features. This means the tree has a higher change of a false negative or false positive between 'C' and 'E', when compared to 'A'.

Which classes are often confused, and as what? Why?

- 'G' and 'O' are most often confused with other classes. This does not happen in `train_full.txt`, but happens in `train_noisy`, and significantly in `train_sub`.
- 'O' is most commonly confused with 'Q', and sometimes 'G'.
- 'C' is most commonly confused with 'E', but 'C' generally has a high recall.
- Samples of 'G' are confused with many other labels in `train_sub.txt`, as the tree used only 21 samples for training.
- Whilst precision and recall for 'G', 'O' and 'Q' are relatively low in all data sets, they get worse for `train_noisy.txt`. This means there are many false negatives and false positives. The features of 'G', 'O' and 'Q' are likely to overlap due to their visual similarity.

Task 3.6

- File name: `cross_validation.py`
- Function names: `split()`, `cross_validation()`
- Implementation:

– `split()` takes as parameters the number of subsets into which the dataset needs to be split (k) and the filename from where to load the dataset. It loads the contents of the file through numpy's `loadtxt()`. Then, the rows of the dataset are shuffled using numpy's `random.shuffle()`. Lastly, the numpy array is split into k subsets using numpy's `array_split()`. All k subsets are stored in a numpy array.

– `cross_validation()` calls `split()` to split the dataset into k subsets. Then, it iterates over each subset using an iterator *i*. It keeps the subset at position *i* as the training subset and concatenates all other subsets to make the training dataset. Lastly, it divides the datasets into attributes and labels so these can be passed to the `DecisionTreeClassifier` methods. At each iteration, the accuracy of the predictions is calculated and stored. Finally, the accuracy values stored are used to calculate the global error estimate and extract the best accuracy.

Question 3.3

For each fold, the (rounded) accuracies are:

1. 0.9179
2. 0.9487
3. 0.9231
4. 0.9359
5. 0.9256
6. 0.9385
7. 0.9333
8. 0.9205
9. 0.9359
10. 0.9333

The average accuracy across the 10 folds is: 0.9313 with a standard deviation of 0.0089. So, we have that the average accuracy is 0.9313 ± 0.0089.
The low standard deviation indicates that there is a small degree of variability in the accuracies of the ten models trained. This means the individual subsets consist of a sufficient number and distribution of samples, resulting in each instance of 10-fold cross validation. For example, if some subsets contain more samples of one label, this will show as reduced accuracy in those instances, increasing the standard deviation.

Question 3.4

The model with the highest accuracy of 0.9487 was evaluated using `test.txt` with the following results:

Accuracy: 0.895

Class	Precision	Recall	F1 Scores
A	0.9697	0.9412	0.9552
C	0.8947	0.9189	0.9067
E	0.9259	0.9615	0.9434
G	0.9231	0.8889	0.9057
O	0.8182	0.7941	0.8060
Q	0.8605	0.8810	0.8706
Macro-averaged	0.8987	0.8976	0.8979

Overall, the cross-validation model performs marginally worse than training on the full dataset, for all metrics. The difference is small, for example, macro-averaged precision here is about 0.90, while on `train_full.txt` it was 0.91.

390 samples (N/k where N is total samples, k = 10) were held for testing when training the cross-validation model. This means the model employed less data during training, reducing the accuracy slightly for cross validation models. The advantage of cross validation is to reduce over fitting and selection bias, but in this case, these were not a problem.

Question 3.5

- File name: `question3-5.py`
- Function name: `combine_cross_validation()`
- Implementation: `combine_cross_validation()` works like `cross_validation()` from `cross_validation().py`, with the following differences:
  - It stores the prediction on `test.txt` of each subset of the tree
  - It creates a prediction for all labels by identifying the most common prediction for each label in `test.txt` among the k subsets
  - It returns the accuracy of this prediction
- Result: $accuracy = 0.94$

Compared to the single tree trained on `train_full.txt`, whose accuracy is 0.91, there is an improvement of 0.3. Arguably, the improvement in accuracy is due to the fact that for each label we have have selected the most popular prediction among the k models. This is likely to work by reducing the bias in our singly trained tree. In our cross-validated models, if one of our model were to incorrectly classify a particular label based on some faulty assumption, it would be out-voted by the others.

Pruning

Task 4.1

- File name: `pruning.py`
- Function names: `pruning()`, `prune()`
- Implementation: `pruning()` takes a tree and the validation dataset as parameters. This function only fulfils two tasks:
  - It checks that the tree has actually been trained.
  - It calls `prune()` that which performs the pruning operations.

  `prune()` takes two parameters: a node and the validation dataset.
  - First, the function checks whether we have reached a node that has two leaves as children (and so could be pruned). If it has not reached such a node, it progresses down the tree, looking for such a node (left branch first).
  - If the function finds a node that has two leaves as children, it calculates the accuracy of the tree as is, and stores it in accuracy_before. Then, it stores the labels of the leaves in temp_label_left and temp_label_right. The decision to store the labels comes from the necessity to restore the original shape of the tree if pruning results in lowered accuracy.
  - Then, pruning actually occurs: the two leaves are deleted and the majority label is stored in the parent node. Finally, the accuracy after pruning is calculated and stored in accuracy_after. If accuracy_after is higher than accuracy_before, the pruned tree is favoured.
  - Otherwise, the original tree is restored.
  - This is repeated recursively on each node of the tree.

Question 4.1

- `train_full.txt` evaluation metrics (rounded)
  - before pruning: accuracy = 0.91, precision = 0.91, recall = 0.91, F1 = 0.91
  - after pruning: accuracy = 0.81, precision = 0.81, recall = 0.81, F1 = 0.81
- `train_noisy.txt` evaluation metrics (rounded)
  - before pruning: accuracy = 0.79, precision = 0.79, recall = 0.79, F1 = 0.79
  - after pruning: accuracy = 0.78, precision = 0.79, recall = 0.78, F1 = 0.78

With regards to `train_full.txt`, pruning decreased all evaluation metrics from 0.91 to 0.81. We are pruning the tree based on an increase of accuracy on the validation set. This does not necessarily entail that we increase accuracy on all datasets (for example, it decreases accuracy in the given test set). Generally, pruning reduces overfitting on the training set. The final performance of the decision tree varies amongst different test sets.

With regards to `train_noisy.txt` pruning did not change the evaluation metrics (except for accuracy- marginally). Arguably, pruning has created a more generalised tree that is able to accommodate for the noise of the noisy training set.

Question 4.2

- Maximal depth before pruning of `train_full.txt`: 21
- Maximal depth after pruning of `train_full.txt`: 19
- Maximal depth before pruning of `train_noisy.txt`: 20
- Maximal depth after pruning of `train_noisy.txt`: 12

The maximal depth of the tree of the noisy set has almost halved. Yet, the accuracy has remained relatively unchanged. Conversely, the maximum depth of the tree of the full set has only decreased by two. Yet, the accuracy has decreased (0.91 to 0.81, see previous question). The pruning function checks the accuracy of the tree before and after pruning with respect to the validation function. Since the noisy dataset contains noise, it is likely that the accuracy before pruning is lower than the accuracy after pruning. As such, more branches will be pruned. In turn, this removes some noise and so increases the accuracy of the model overall. Since the full dataset did not contain noise, it is possible that the accuracy after pruning was lower than before and so more branches were kept intact.