CO395 (Spring 2019/20)

Dr. Josiah Wang

Coursework 1: Decision Trees

Group 55

Fenton, Sebastian
Govani, Naim
Sarif-Kattan, Ethan

Part 1

Task 1.1

The handler is located in src/util/data_read.py, and the data_set class to hold features and ground truths is located in src/util/data_set.py.

We wish to highlight that our data_read function is dynamic (and only requires the path to the target data file as input) as specified. Note also that although the data_read function works with basic python lists, as this is syntactically cleaner for basic read/write operations, these lists are then converted to NumPy arrays for storage as this is more convenient for other tasks.
The second point to note is about our Dataset class. For the purpose of clarity and readability it was decided that the Dataset class would be a NumPy array of DataEntry classes, another self-defined class. This DataEntry class itself contains a NumPy array, which holds the attributes, and a string, which holds the ground truth, both of which correspond to the same single data entry from the training data. By defining custom classes for both the entire training data array and the individual entries we can define member functions in their appropriate class; for example, the overloaded equality operator is defined only in the Data Entry class, thus making future code concise and straightforward to read.

Question 1.1

The attributes provided are integer, save for the ground truth which is categorical. The integer ranges are bounded between 0 and 15 inclusive, and the categorical ground truth is A-Z (26 Upper Case Letters).

1.2

train_full.txt has 3900 data entries, whereas train_sub.txt only has 600. Furthermore, the number of entries per letter in train_full.txt are fairly even (600 per letter), whereas in train_sub.txt they are not as evenly distributed - the letter G has 21 entries, whereas C has 187.

1.3

The proportion of entries that are different can be given by summing the positive entries in the Change column of Figure 1, and dividing that by the total number of entries. This gives us:

$$\% \text{ change} = 77/3900 = 1.97\%$$

The noise has affected class distribution, with increases in the number of A, E, O and Q entries, and decreases in the number of C and G entries, with G being the most substantial change. See Figure 1 for a full summary of the changes.

Letter	Full	Noisy	Change
A	667	681	+14
C	599	571	-28
E	659	678	+19
G	671	622	-49
O	637	666	+29
Q	667	682	+15

Figure 1: Table showing the changes in the distribution of letter entries

Part 2

Task 2.1

Our decision tree is binary, and treats the integer attributes as real-valued as opposed to ordinal. We made this design choice as we felt that a tree which branched 16 ways at each node would be at strong risk of over-classifying, even with pruning.
As our tree is binary, each node in it holds not only the attribute to split on, but also the split point of said attribute. All splits are binary, so each node contains the attribute and the split point, with the assumption that the split will always be a $<$ operation.
We used Information Gain as our statistical method for deciding how to split. We made this decision as we saw almost no difference between methods, even though technically using the Gini Impurity might take slightly less time to train as it does not involve logarithmic calculation.

2.2

Our predict function, although called from DecisionTreeClassifier in classification.py, is mostly written as a method of the BinTree class defined in src/classification/tree.py. We do not believe that there are any special features about its implementation that we ought to highlight.

Task 2.3

We did not make an image-based node visualisation as we felt that this was a lot of work without being able to use libraries such as Igraph or NetworkX. To perform our basic printing operation, we use src/classification/visualise_tree.py as a handler which invokes the __repr__ method in the NodeData class (which itself is located in src/classification/tree.py). See below the visualisation of the tree trained on train_full.txt:

```
[L0] | x_10 < 3 | ChldEntr: 2.15 |
     T: [L1] | x_0 < 6 | ChldEntr: 0.05 |
          T: [L2] | x_15 < 10 | ChldEntr: 0.01 |
               T: [L3] Leaf: A
               F: [L3] | x_2 < 5 | ChldEntr: 0.00 |
                    T: [L4] Leaf: G
                    F: [L4] Leaf: A
          F: [L2] | x_15 < 13 | ChldEntr: 0.60 |
               T: [L3] | x_5 < 12 | ChldEntr: 0.39 |
                    T: [L4] | x_2 < 11 | ChldEntr: 0.00 |
                         T: [L5] Leaf: G
                         F: [L5] Leaf: A
                    F: [L4] Leaf: A
               F: [L3] Leaf: Q
     F: [L1] | x_5 < 7 | ChldEntr: 2.16 |
          T: [L2] | x_14 < 6 | ChldEntr: 1.53 |
               T: [L3] | x_11 < 11 | ChldEntr: 1.26 |
                    T: [L4] | x_9 < 11 | ChldEntr: 1.34 |
                         T: [L5] | x_12 < 3 | ChldEntr: 1.00 |
                              T: [L6] | x_7 < 4 | ChldEntr: 0.21 |
                                   T: [L7] Leaf: E !MDE
                                   F: [L7] | x_8 < 8 | ChldEntr: 0.07 | !MDE
                              F: [L6] | x_7 < 5 | ChldEntr: 1.16 |
                                   T: [L7] | x_12 < 5 | ChldEntr: 1.08 | !MDE
                                   F: [L7] | x_9 < 7 | ChldEntr: 0.64 | !MDE
                         F: [L5] | x_7 < 5 | ChldEntr: 0.36 |
                              T: [L6] | x_14 < 3 | ChldEntr: 0.00 |
                                   T: [L7] Leaf: C !MDE
                                   F: [L7] Leaf: E !MDE
                              F: [L6] | x_15 < 6 | ChldEntr: 0.00 |
                                   T: [L7] Leaf: E !MDE
                                   F: [L7] Leaf: C !MDE
                    F: [L4] | x_12 < 2 | ChldEntr: 0.46 |
                         T: [L5] | x_15 < 11 | ChldEntr: 0.01 |
                              T: [L6] Leaf: C
                              F: [L6] | x_0 < 3 | ChldEntr: 0.00 |
                                   T: [L7] Leaf: E !MDE
                                   F: [L7] Leaf: C !MDE
                         F: [L5] | x_6 < 8 | ChldEntr: 0.66 |
                              T: [L6] | x_15 < 9 | ChldEntr: 0.77 |
                                   T: [L7] | x_10 < 8 | ChldEntr: 0.42 | !MDE
                                   F: [L7] | x_7 < 5 | ChldEntr: 0.45 | !MDE
                              F: [L6] | x_7 < 5 | ChldEntr: 0.00 |
                                   T: [L7] Leaf: A !MDE
                    F: [L7] Leaf: C !MDE
        F: [L3] | x_8 < 7 | ChldEntr: 0.72 |
            T: [L4] | x_10 < 6 | ChldEntr: 1.34 |
                T: [L5] | x_6 < 5 | ChldEntr: 0.23 |
                    T: [L6] Leaf: Q
                    F: [L6] | x_0 < 5 | ChldEntr: 0.00 |
                        T: [L7] Leaf: C !MDE
                        F: [L7] Leaf: O !MDE
                F: [L5] | x_7 < 6 | ChldEntr: 1.02 |
                    T: [L6] | x_10 < 7 | ChldEntr: 0.94 |
                        T: [L7] | x_11 < 10 | ChldEntr: 0.48 | !MDE
                        F: [L7] | x_13 < 10 | ChldEntr: 0.54 | !MDE
                    F: [L6] | x_11 < 15 | ChldEntr: 0.00 |
                        T: [L7] Leaf: G !MDE
                        F: [L7] Leaf: C !MDE
            F: [L4] | x_11 < 9 | ChldEntr: 0.11 |
                T: [L5] | x_0 < 8 | ChldEntr: 0.50 |
                    T: [L6] | x_1 < 11 | ChldEntr: 0.00 |
                        T: [L7] Leaf: O !MDE
                        F: [L7] Leaf: E !MDE
                    F: [L6] Leaf: C
                F: [L5] | x_15 < 11 | ChldEntr: 0.00 |
                    T: [L6] Leaf: E
                    F: [L6] Leaf: G
    F: [L2] | x_14 < 4 | ChldEntr: 2.10 |
        T: [L3] | x_7 < 6 | ChldEntr: 0.95 |
            T: [L4] | x_1 < 5 | ChldEntr: 1.27 |
                T: [L5] | x_8 < 4 | ChldEntr: 0.68 |
                    T: [L6] | x_13 < 9 | ChldEntr: 0.15 |
                        T: [L7] | x_5 < 9 | ChldEntr: 0.00 | !MDE
                        F: [L7] Leaf: Q !MDE
                    F: [L6] | x_0 < 2 | ChldEntr: 0.00 |
                        T: [L7] Leaf: G !MDE
                        F: [L7] Leaf: E !MDE
                F: [L5] | x_11 < 10 | ChldEntr: 0.75 |
                    T: [L6] | x_7 < 5 | ChldEntr: 0.23 |
                        T: [L7] | x_15 < 9 | ChldEntr: 0.00 | !MDE
                        F: [L7] Leaf: O !MDE
                    F: [L6] | x_4 < 4 | ChldEntr: 0.53 |
                        T: [L7] Leaf: A !MDE
                        F: [L7] | x_0 < 5 | ChldEntr: 0.52 | !MDE
            F: [L4] | x_9 < 7 | ChldEntr: 0.37 |
                T: [L5] | x_12 < 3 | ChldEntr: 0.41 |
                    T: [L6] Leaf: Q
                    F: [L6] | x_12 < 4 | ChldEntr: 0.98 |
                        T: [L7] | x_6 < 6 | ChldEntr: 0.35 | !MDE
                        F: [L7] | x_1 < 7 | ChldEntr: 0.00 | !MDE
                F: [L5] | x_13 < 9 | ChldEntr: 0.17 |
                    T: [L6] | x_6 < 6 | ChldEntr: 0.04 |
                        T: [L7] | x_1 < 8 | ChldEntr: 0.00 | !MDE
                        F: [L7] | x_15 < 9 | ChldEntr: 0.01 | !MDE
                    F: [L6] | x_1 < 6 | ChldEntr: 0.72 |
                        T: [L7] | x_5 < 8 | ChldEntr: 0.30 | !MDE
                        F: [L7] Leaf: O !MDE
        F: [L3] | x_7 < 4 | ChldEntr: 2.12 |
            T: [L4] | x_8 < 5 | ChldEntr: 0.73 |
                T: [L5] | x_14 < 6 | ChldEntr: 0.88 |
                    T: [L6] | x_10 < 4 | ChldEntr: 0.50 |
                        T: [L7] | x_1 < 9 | ChldEntr: 1.21 | !MDE
                        F: [L7] | x_6 < 12 | ChldEntr: 0.00 | !MDE
                    F: [L6] | x_2 < 4 | ChldEntr: 0.00 |
                        T: [L7] Leaf: E !MDE
                        F: [L7] Leaf: G !MDE
                F: [L5] | x_6 < 6 | ChldEntr: 0.21 |
                    T: [L6] Leaf: Q
                    F: [L6] | x_15 < 7 | ChldEntr: 0.00 |
                        T: [L7] Leaf: A !MDE
                        F: [L7] Leaf: E !MDE
            F: [L4] | x_12 < 3 | ChldEntr: 2.04 |
                T: [L5] | x_8 < 5 | ChldEntr: 1.16 |
                    T: [L6] | x_3 < 2 | ChldEntr: 0.28 |
                        T: [L7] Leaf: G !MDE
                        F: [L7] | x_15 < 7 | ChldEntr: 0.09 | !MDE
                    F: [L6] | x_7 < 6 | ChldEntr: 0.94 |
                        T: [L7] | x_8 < 6 | ChldEntr: 0.38 | !MDE
                        F: [L7] | x_15 < 10 | ChldEntr: 0.76 | !MDE
                F: [L5] | x_14 < 6 | ChldEntr: 1.98 |
                    T: [L6] | x_12 < 4 | ChldEntr: 1.65 |
                        T: [L7] | x_14 < 5 | ChldEntr: 1.06 | !MDE
                        F: [L7] | x_5 < 9 | ChldEntr: 1.72 | !MDE
                    F: [L6] | x_10 < 8 | ChldEntr: 1.82 |
                        T: [L7] | x_15 < 6 | ChldEntr: 1.79 | !MDE
                        F: [L7] | x_1 < 10 | ChldEntr: 0.56 | !MDE
```

The !MDE flag shows that maximum depth is exceeded. ChldEntr shows the child entropy, and T: or F: show whether the node satisfies the split condition of its parent or not.

Question 2.1

The attributes in the train_full.txt training data are as follow:

x0:	Horizontal position of box
x1:	Vertical position of box
x2:	Width of box
x3:	Height of box
x4:	Total on pixels
x5:	Mean x of on pixels in box
x6:	Mean y of on pixels in box
x7:	Mean x variance
x8:	Mean y variance
x9:	Mean x y correlation
x10:	Mean of $x^2 * y$
x11:	Mean of $x * y^2$
x12:	Mean edge count left to right
x13:	Correlation of **x12** with y
x14:	Mean edge count bottom to top
x15:	Correlation of **x14** with x

The first split is on attribute x10, with split point < 3. While splitting on the Mean of $x^2 * y$ might seem like an odd decision to a human, the algorithm we've implemented iterates through every attribute and all the significant split points for that attribute and calculates the Information Gain for every split point. The split point with the highest Information Gain is then used to split the training data, and the function is then called again on the subsets until every path in the tree leads to a leaf node. Therefore, while the first decision might not make sense from a high-level perspective it is the result of intense statistical analysis of the training data and thus may be the best choice given the constraints we have placed on the tree.
Improvements on the tree are be possible; however, they come in the form of choosing more informative attributes and changing the way in which we calculate splits, both of which are outside the scope of this report.

Part 3

Task 3.6

The function split_k_subsets is in the BinTree class located at src/util/dataset.py. It uses built in numpy functions, and returns an array of Dataset objects. The kfold function is the the root level kfold.py file.

Question 3.1

	Full	Sub	Noisy
Accuracy	0.860	0.750	0.790
Recall	0.859	0.751	0.794
Precision	0.861	0.735	0.792
F1 Score	0.858	0.722	0.788

Figure 2: Table showing the macro-averaged scores for each data set

	A	C	E	G	O	Q
Recall	0.943	0.892	0.781	0.870	0.818	0.85
Precision	0.971	0.892	0.962	0.740	0.794	0.810
F1 Score	0.957	0.892	0.962	0.741	0.794	0.810

Figure 3: Table showing the class scores for the full training set

	A	C	E	G	O	Q
A	33	0	0	1	0	0
C	1	33	2	1	0	0
E	0	0	25	0	1	0
G	1	1	2	20	1	2
O	0	2	0	1	27	4
Q	0	1	3	0	4	34

Figure 4: Table showing the class scores for the full training set

	A	C	E	G	O	Q
Recall	0.900	0.787	0.647	0.750	0.697	0.727
Precision	0.794	1.000	0.846	0.333	0.676	0.762
F1 Score	0.844	0.881	0.733	0.462	0.687	0.762

Figure 5: Table showing the class scores for the sub training set

	A	C	E	G	O	Q
A	27	0	4	0	0	2
C	0	37	0	0	0	0
E	0	3	22	0	0	1
G	1	3	6	9	2	0
O	1	3	0	4	23	5
Q	1	1	2	4	6	32

Figure 6: The confusion matrix for sub training set

	A	C	E	G	O	Q
Recall	0.939	0.969	0.735	0.714	0.750	0.659
Precision	0.912	0.838	0.962	0.556	0.794	0.690
F1 Score	0.925	0.899	0.833	0.625	0.771	0.674

Figure 7: Table showing the class scores for the noisy training set

	A	C	E	G	O	Q
A	31	0	0	0	1	2
C	1	31	1	3	1	0
E	0	1	25	0	0	0
G	0	0	4	15	0	8
O	0	0	1	1	27	5
Q	1	0	3	2	7	29

Figure 8: The confusion matrix for noisy training set

Question 3.2

The tree trained on the Sub data set is less accurate than the one trained on the Full data set. As long as all the training data follows the distribution specified by the true function, which we are attempting to replicate with our tree, more data points allow the tree to draw

more refined statistical conclusions and thus our hypothesis function becomes a better approximation of the true function. This is also why even the tree trained on the Full data set is not 100% accurate, as relatively speaking there can always be more data to sample and even then our observations might even be misguided thus contorting our hypothesis.
The Noisy data set is designed to have a marginally different underlying distribution than the true function and thus causes the decision tree to become even less accurate. The reason for this reduction in accuracy is because our only understanding of the true function comes from the statistical observations that the tree makes on the training data and as this data now represents a different true function, due the differences in distribution, our hypothesis now attempts to approximate this new true function, one which may be similar to the our desired true function but is still a different mapping.
Moving on to the individual classes themselves we can see that 'G' has the worst F1 score over all the sets and further inspection of the confusion matrices shows that an input corresponding to a 'G' can be mistaken for all other classes. This may be due to 'G' having a large overlap between its attribute distribution and other classes attribute distribution, i.e. the attribute values that would ideally detect a 'G' are not distinct enough for the hypothesis to differentiate them from other classes. Another case of this is when the model mistakes an 'O' for a 'Q', this happens frequently and in both directions but is also caused by 'O' and 'Q' having very similar attributes which identify them. On the other hand, class 'A' seems have a high scores in both precision and recall, indicating that the model has identified a set of attribute values that correlate strongly to the label 'A'.

Question 3.3

Having performed 10-fold cross-validation, we arrived at the following accuracy and standard deviation:

$$0.9197435897435898 \pm 0.01019533094978003 \tag{1}$$

The implication of a low standard deviation is that the model will predict more consistently when tested on unseen data.

Question 3.4

The tree with highest accuracy of the kfold subsets gives a 3% improvement in accuracy, from 86% to 89%, when tested against test.txt compared to the tree trained on train_full.txt. This is probably because the train_full.txt dataset wasn't big enough.
Despite the increased accuracy, it is interesting to observe that precision drops significantly from the full tree to our 10-fold tree:
To see the full comparison between the two, we can compare Figure 10 to Figure 3, and Figure 9 to Figure 2.

	10-Fold Best
Accuracy	0.890
Recall	0.893
Precision	0.893
F1 Score	0.890

Figure 9: Table showing the macro-averaged scores for the best-performing tree from the 10-fold cross-validation

	A	C	E	G	O	Q
Recall	0.971	0.891	1.00	0.778	0.882	0.833
Precision	0.917	0.917	0.839	0.955	0.833	0.897
F1 Score	0.943	0.904	0.912	0.857	0.857	0.864

Figure 10: Table showing the class scores when using the best-performing tree from the 10-fold cross-validation

Question 3.5

Figure 11 shows the recall, precision and F1 score when all 10 tress from the k-fold cross-validation are used to determine the prediction. The approach we used was to pass an input into all 10 trees and the select the most common prediction from all of them, thus reducing the bias that is caused by the training data. The F1 Score for each class increased when

	A	C	E	G	O	Q
Recall	0.971	0.919	1.00	0.815	0.912	0.905
Precision	0.917	0.894	0.867	0.956	0.912	0.927
F1 Score	0.970	0.907	0.929	0.880	0.912	0.916

Figure 11: Table showing the class scores when using the all sub-trees from the 10-fold cross-validation

compared to using the best-performing tree, and the overall accuracy of the 'combined tree' system as a predictor has increased to 92% - an additional 3% increase on the accuracy of the best-performing tree. One potential conclusion that we can draw from this result is that improving our approximation of the true function can be achieved by creating multiple approximations based on different data sets and then finding a correlation or consensus between them.

Part 4

Task 4.1

The pruning function was implemented following the simple approach provided in the spec. It performs recursion through the whole tree until it reaches a node where both children are leaf nodes. It then calculates the accuracy of tree before and after removing that node. If the accuracy after removal is less than before, it leaves the node as is; otherwise, it converts the parent node into the majority leaf of the current dataset that is whittled down as we walk down the tree.
Calling the prune function once will try and remove the lowest level of leaves across the entire tree. That is why we call it multiple times in a loop. We can add a check to stop calling it if no leaves were pruned for that call.
The function is implemented in src/classification/tree.py, as a method of BinTree. However, it is invoked from src/classification/prune.py, where we can test the pruning by loading a trained tree from disk and providing validation data.

Question 4.1

As instructed, we applied the pruning function to a tree that had been trained on test_full.txt - the system logs, which we hope are intuitively readable enough to not justify much explanation, are shown below:

```
test accuracy before pruning: 0.86
#prune attempt 0#
    pruned leaves Leaf: O Leaf: C  into  Leaf: O
    acc before:  0.91 acc after:  0.92
#prune attempt 1#
#prune attempt 2#
#prune attempt 3#
#prune attempt 4#
#prune attempt 5#
#prune attempt 6#
#prune attempt 7#
#prune attempt 8#
#prune attempt 9#
test accuracy after pruning: 0.865
```

As we can clearly see from the logs above, accuracy improved by 0.5% after pruning on train_full.txt. It only pruned the lowest level leaves, and only pruned 1 leaf.
We also applied the pruning function to a tree that had been trained on train_noisy.txt. The results were:

```
test accuracy before pruning: 0.79
```

```
#prune attempt 1#
    pruned leaves Leaf: G Leaf: Q  into  Leaf: E
    acc before:  0.84 acc after:  0.85
    pruned leaves Leaf: O Leaf: Q  into  Leaf: E
    acc before:  0.85 acc after:  0.86
    pruned leaves Leaf: O Leaf: E  into  Leaf: G
    acc before:  0.86 acc after:  0.88
    pruned leaves Leaf: E Leaf: C  into  Leaf: E
    acc before:  0.88 acc after:  0.89
    pruned leaves Leaf: E Leaf: G  into  Leaf: G
    acc before:  0.89 acc after:  0.91
    pruned leaves Leaf: G Leaf: C  into  Leaf: G
    acc before:  0.91 acc after:  0.92
    pruned leaves Leaf: A Leaf: Q  into  Leaf: A
    acc before:  0.92 acc after:  0.93
#prune attempt 2#
#prune attempt 3#
#prune attempt 4#
#prune attempt 5#
#prune attempt 6#
#prune attempt 7#
#prune attempt 8#
#prune attempt 9#
#prune attempt 10#
 test accuracy after pruning: 0.795
```

Again, increased accuracy on train_noisy by 0.5%

While pruning did help for both the noisy and full data sets, it only led them to correctly guess a single additional test case than their pre-pruned counterparts. Therefore, for the decision tree we have created pruning improves the accuracy minimally and improves by the same amount for both noisy and normal training data

Question 4.2

The maximum depth remained the same for both datasets, even after pruning. Therefore to comment on the relationship between depth and accuracy would not be grounded in any experimental data. We would hypothesise that the closer the depth is to the number of attributes, at least in a binary tree, then the more accurate the tree would be.