# Coursework 1: Decision Trees

CO395 (Spring 2019/20)

**Deadline: Monday 10th Feb 2020 (7pm) on CATe**

## 1 Overview

Please read this manual **THOROUGHLY** as it contains crucial information about the coursework, as well as the support provided.

In this coursework, you will implement a **decision tree** from scratch in **Python 3** to classify a set of black-and-white pixel images into one of several letters in the English alphabet (Figure 1). You are expected to deliver a **report** answering any questions specified and discussing your implementation and the results of your experiments. You should also submit the **SHA1 token** for the specific commit on your GitLab repository that you wish to be assessed.

The objectives of this coursework are:

- to give you hands-on experience in implementing a classification pipeline;
- to guide you through the thought process of designing, implementing, and evaluating a classifier;
- to help you appreciate the importance of examining and understanding your data;
- to improve your understanding of decision trees by implementing a decision tree classifier from scratch;
- to gain experience and improve your understanding on how to evaluate classifiers.

We hope you will be able to learn a lot from doing this coursework!

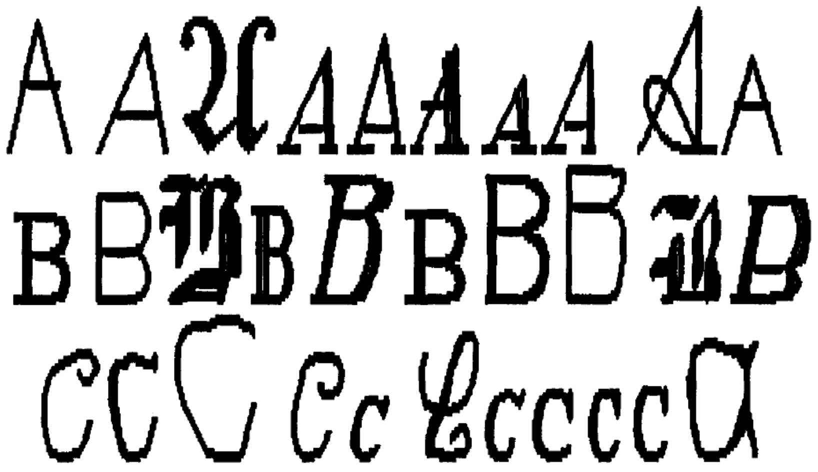


Figure 1: Your task is to implement a decision tree from scratch that can classify a preprocessed blank-and-white pixel image to a letter in the English alphabet. The image is taken from Frey and Slate [1].

## 2 Setup

You should implement all your code in **Python 3**. The link to your group GitLab repository with the skeleton code should be available on `LabTS` (`https://teaching.doc.ic.ac.uk/labts/`). Please push your codes to your repository using the usual `Git` workflow.

The `python3` environment in the Ubuntu workstations in the lab should have everything you need to complete this coursework, so we recommend you work on the Ubuntu workstations in the lab. Alternatively, you can SSH onto one the lab machines from home. See the list on `https://www.doc.ic.ac.uk/csg/facilities/lab/workstations`.

Your code will be assessed and tested on `LabTS`. The `python` environment on the `LabTS` servers should closely mimic the ones on the lab workstations. A stronger reason for you to work on the lab workstations!

We will provide a suite of tests for you to test your code prior to submission using `LabTS`. To test your code, push to your GitLab group repository and access the tests through the `LabTS` portal.

**The tests are only an indication of the status of your code, and may not reflect your final mark in any part of this coursework.** We will assess your final code using a different test suite.

Please note that the test suite is expected to evolve between the release and the submission dates of this coursework.

## 3 Detailed instructions

We have designed the instructions in this section to guide you through the coursework. The coursework comprises **four parts**. The green boxes give you the implementation tasks that you need complete. You should also answer all questions in the red boxes in your report.

**Implementation**

You are expected to **implement** your code in **Python 3**, and to **describe your implementation** in your report.

You are only allowed to use `NumPy` and `Matplotlib` as external libraries for this coursework. Any other libraries, such as `scikit-learn`, are **NOT** allowed.

**Answers to Questions**

Please answer all questions in boxes such as these in your report.

## Part 1: Loading and examining the dataset (10 marks)

The dataset is a small subset of the letter recognition dataset developed by Frey and Slate [1]. We have provided you with *pre-extracted features*; this allows you to focus on the Machine Learning aspects of the coursework and frees you from worrying about processing images. The images themselves are not provided or needed for this coursework.

The `data/` directory contains several datasets. The main datasets are:

- `train_full.txt`: The full training set (3900 rows);
- `train_sub.txt`: A subset of the full training set (600 rows);
- `train_noisy.txt`: A 'noisy' training set (3900 rows);
- `validation.txt`: A validation dataset (100 rows) to optimise any hyperparameters of your classifier or to evaluate your classifier;
- `test.txt`: A test set to be used in Parts 3 and 4. As best practice, please try your best not to look at this dataset until then.

Each dataset file is a text file with $N$ rows and 17 columns separated by commas, where $N$ is a number of samples. The first 16 columns are the *attributes* or *features*; a short description of these attributes are in `data/README.md`. The final column is the *gold standard* or *ground truth* classification label, i.e. the annotated 'answer' given the attributes.

We have also provided simpler versions of the datasets which you may find useful for debugging and development purposes: `toy.txt`, `simple1.txt`, `simple2.txt`. These datasets have fewer attributes, classes, and instances.

**Task 1.1 (4 marks)**

Write a function or class method to read in the datasets. You have freedom to decide how you want to represent your data: for example using a `NumPy` array for attributes and a separate `NumPy` array for classification labels, or to group them as a class attribute in a Python `Dataset` class (as an example).

Your function must be able to handle varying number of attributes and instances, i.e. it should automatically be able to read and handle `toy.txt`, `simple1.txt`, `simple2.txt` as well as the main datasets, without any hard-coding or needing to be explicitly given the number of attributes.

In your report, please give the file name(s) and function/class name(s) of where your implementation of the dataset reader is located. Also highlight any details of your implementation to help us understand your code.

### Understanding the data

One important and useful thing that you should always do is to **examine and understand your data** before you start any Machine Learning experiments.

Look at the datasets `train_full.txt`, `train_sub.txt`, and `train_noisy.txt` (you can examine the raw text files directly or via Python). There are many questions you can ask yourself: How many samples/instances are there? Will that be enough to train a classifier? How many unique class labels (characters to be recognised) are there? What is the distribution across the classes (e.g. 40% `'A'`s, 20% `'C'`s)? Are the samples balanced across all the classes, or are they biased towards one or two classes? What does each of the 16 attributes represent? Understanding these questions can influence how you design your Machine Learning algorithms.

To further understand the datasets, please answer the following questions in your report:

**Question 1.1 (2 marks)**

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`?

**Question 1.2 (2 marks)**

What are the two main differences between `train_full.txt` and `train_sub.txt`?

**Question 1.3 (2 marks)**

`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`.

## Part 2: Implementing decision trees (35 marks)

Your main task in this section is to **implement your own decision tree from scratch** using **Python 3**. As a reminder, you are only allowed to use `NumPy` and `Matplotlib` as external libraries. Any other libraries, such as `scikit-learn`, are **NOT** allowed.

In the lecture, we have provided you with a high-level algorithm for inducing a decision tree:

1. Search for an 'optimal' splitting rule on training data;
2. Split your dataset according to your chosen splitting rule;
3. Repeat (1) and (2) on each of the created subsets.

Step (1) is the key point to your implementation. You have several design decisions to make:

- How many splits should there be at a node?
- What kind of node should be used?
- How should the decision tree select an 'optimal' node?

Think carefully about these issues before you start implementing your decision tree. There is no definite right or wrong answer, although your choice may affect the performance of your decision tree. Below is some guidance:

### How many splits per node?

Your tree can be *binary* (two splits per node) or *multiway* (two or more splits per node). This design choice also partly depends on the node and the type of attributes of your dataset. If you have examined the dataset, you should have seen that the attributes are all integers within a certain range. Thus, you can choose to treat the attributes as real-valued or ordinal. You will most likely choose a *binary* tree if the attributes are real-valued, and either a *binary* or *multiway* tree if they are ordinal.

### What kind of node should be used?

If you decide to treat the attributes as real-valued, then your algorithm should be able to choose the attribute *and* split point that results in the most informative split (e.g. $x_1 < 5$). An efficient method for finding good split points is to first sort the values of the attribute, and then consider only split points that are between two examples in sorted order that have different class labels, while keeping track of the running totals of positive and negative examples on each side of the split point.

If you decide to treat the attributes as ordinal, then your algorithm should be able to discretise the values into several bins (a tree with a 16-way split is unlikely to generalise well to unseen data). You can then assign each value into one of these bins. For example, you might choose to divide $x_1$ into three bins: $x_1 < 5; 5 \leq x_1 < 10; x_1 \geq 10$. Then you assign each $x_1$ in the training set into one of these bins. In this example, your node will have a three-way split for each bin. Choosing the 'correct' bins is, of course, another design decision you will need to consider. You may also encounter problems with discontinuities at the boundary of the bins.

### How should the decision tree select an 'optimal' node?

To select the most informative node for a dataset split, you are free to choose from various statistical tests, such as Information Gain or Gini Impurity Criterion. For our discussion here, we will assume that you chose to use Information Gain as discussed in the lectures, but the same principle applies if you opt for a different statistical test measure.

To evaluate the information gain, suppose that the training dataset $S^{parent}$ has $C$ different class labels. The splitting rule will divide $S^{parent}$ into subsets $\tilde{S}^{children} = \{S_1, S_2, \ldots, S_I\}$. For a binary tree, $I = 2$. The information gain is defined using the general definition of the entropy:

$$Gain(S^{parent}, \tilde{S}^{children}) = H(S^{parent}) - H'(\tilde{S}^{children}) \tag{1}$$

$$H(S) = -\sum_{c=1}^{C} p_c * \log_2(p_c) \tag{2}$$

$$H'(\tilde{S}) = \sum_{i \in \tilde{S}}^{I} \frac{|S_i|}{N} H(S_i) \tag{3}$$

where $p_c$ is the number of samples with label $c$ in a dataset $S$ divided by the total number of samples in $S$; $|S|$ is the number of samples in subset $S$, $I$ the number of splits; $N = \sum_i^I |S_i|$ is the total number of samples across all subsets (equivalent to the number of samples in $S_{parent}$).

Your decision tree will compute $Gain(S^{parent}, \tilde{S}^{children})$ for different sets of $\tilde{S}^{children}$ for different attributes and split points, and select the node which gives the maximum information gain (or equivalently the minimum entropy).

Since a decision tree is essentially a tree data structure, a good way to implement this is via recursion. A possible implementation could be as follows:

**Algorithm 1** Decision Tree induction

```
function INDUCE_DECISION_TREE(dataset)
    if all samples have the same label or dataset cannot be split further then
        return a leaf node with the majority label
    else
        node ← FIND_BEST_NODE(dataset)
        children_datasets ← SPLIT_DATASET(node)
        for child_dataset ∈ children_datasets do
            child_node ← INDUCE_DECISION_TREE(child_dataset)
            node.add_child(child_node)
        end for
        return node
    end if
end function
```

### Training your model

For development purposes, we have provided you with three simplified datasets:

- `toy.txt`: (2 classes, 3 attributes, 10 instances)
- `simple1.txt`: (2 classes, 4 attributes, 1266 instances)
- `simple2.txt`: (4 classes, 7 attributes, 2596 instances).

It is always a good idea to use a simple toy dataset during development to make sure that your code is working as expected, and to debug your code as necessary. Otherwise, you might end up pulling your hair trying to figure out why your code is not behaving as expected. Or you might end up spending a lot of time training on large datasets to later discover some bugs in your code (and have to re-train your model all over again). Boo :-(

In fact, `toy.txt` is small enough that you can even compute the expected decision tree by hand, which you can compare to the output of your implementation.

### Tip: Saving your model

While this is **not** part of the coursework, it is also a good idea to save any models that you painstakingly trained onto the hard drive. You can then load your model later to perform predictions or continue training from a certain 'checkpoint'. This becomes more important when your model takes a long time to train. Depending on how you designed your model, you may choose to use Python's `pickle`, `NumPy`'s `save()`/`load()` or serialise your object(s) in a JSON format. Note that we will **not** be assessing this for the coursework.

## Task 2.1 (15 marks)

Your task is to implement your decision tree algorithm that constructs a decision tree from training data. Complete the `train()` method of the `DecisionTreeClassifier` class in `classification.py`. The method expects two arguments as input:

- an $N \times K$ `NumPy` array representing $N$ training instances of $K$ attributes;
- an $N$-dimensional `NumPy` array containing the class label for each $N$ instance.

The method should return a `DecisionTreeClassifier` instance, which should contain information about the decision tree you have constructed. We expect this instance to be used later for predicting new test instances (by invoking the `predict()` method).

Feel free to include more input arguments as required, as long as you make these input arguments *optional*, i.e. a default value is assigned. During assessment, `LabTS` will attempt to create a new instance of `DecisionTreeClassifier` and invoke `train()` with some arbitrary instances and their class labels.

In your report, describe details of your implementation. Discuss and explain any design decisions you made, as discussed above, and why you made those decisions. Discuss anything else in your decision tree implementation that you think deserves our attention.

## Task 2.2 (5 marks)

This task requires you to complete Task 2.1 first.

Once constructed, your decision tree will also need to be able to predict the class labels of new, unseen samples. Complete the `predict()` method of the `DecisionTreeClassifier` class in `classification.py`. The method expects one argument as input:

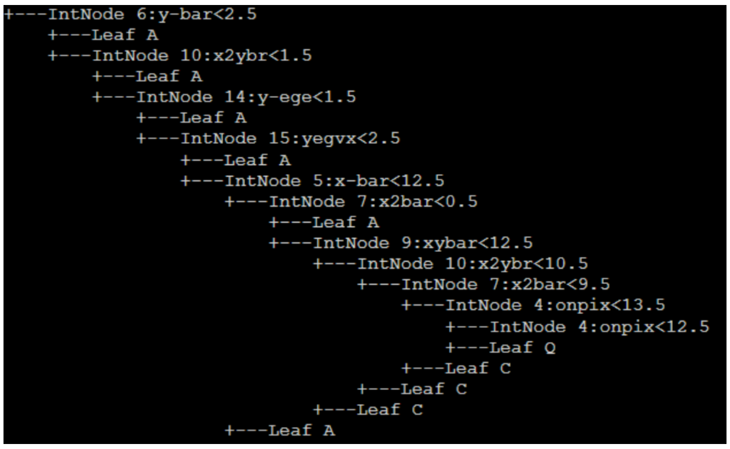
- an $M \times K$ `NumPy` array representing $M$ test instances of $K$ attributes

The method should return an $M$-dimensional `NumPy` array, which should contain the class labels that your decision tree predicted for the given $M$ test instances.

Again, you may include more input arguments to the method as long as you make them optional. During assessment, `LabTS` will invoke `predict()` with some arbitrary instances.

In your report, give a brief description of your implementation. Discuss any design decisions you made, and anything else that you think deserves our attention.

We will also train your decision tree learning implementation on our own training sets to test its robustness **(6 marks)**. For example, we will test whether it can be trained on a different number of attributes or class labels. Therefore, please ensure that your code runs on `LabTS` to avoid losing marks!



```
+---IntNode 6:y-bar<2.5
    +---Leaf A
    +---IntNode 10:x2ybr<1.5
        +---Leaf A
        +---IntNode 14:y-ege<1.5
            +---Leaf A
            +---IntNode 15:yegvx<2.5
                +---Leaf A
                +---IntNode 5:x-bar<12.5
                    +---IntNode 7:x2bar<0.5
                        +---Leaf A
                        +---IntNode 9:xybar<12.5
                            +---IntNode 10:x2ybr<10.5
                                +---IntNode 7:x2bar<9.5
                                    +---IntNode 4:onpix<13.5
                                        +---IntNode 4:onpix<12.5
                                        +---Leaf Q
                                    +---Leaf C
                                +---Leaf C
                            +---Leaf C
                    +---Leaf A
```

Figure 2: Text-based visualisation of a decision tree. Shown here is a tree of max depth 10.

## Task 2.3 (5 marks)

As with examining the dataset, it is also a good idea to examine the decision tree classifier that you have constructed.

Write a function to visualise your decision tree. You can only use `Python`, `NumPy` and `Matplotlib` for this. It is sufficient to implement a text-based visualisation (see Figure 2 for an example). Displaying more information like entropy and/or the class distribution on the tree will earn you more marks.

For a bigger challenge, you can also create an image-based tree visualisation for up to 3 bonus marks (on top of the allocated marks).

In your report, please provide the file names(s) and the functions/classes/methods to visualise the decision tree. Also provide a screenshot of the visualisation of the tree trained on `train_full.txt`. For readability, it is sufficient to show the visualisation up to a tree depth of 10.

## Question 2.1 (4 marks)

Examine the decision tree trained on `train_full.txt` (having a tree visualisation will make this easier). What is first split rule or attribute of the tree (the root node)? Does it make sense for this attribute to be the first split? Explain. Also comment on and discuss about any other split rules/attributes in the tree.

## Part 3: Evaluation (35 marks)

You will now evaluate the predictions of the Decision Tree models that you implemented in Part 2. This section is designed for you to gain some experience in implementing different evaluation metrics and in evaluating the output of Machine Learning models.

You are asked to complete the `Evaluator` class in `eval.py`. `LabTS` will test your implementations by creating an instance of `Evaluator` and invoking the methods with some arbitrary data.

### Task 3.1 (2 marks)

Complete the `confusion_matrix()` method of the `Evaluator` class in `eval.py`. The method expects two mandatory arguments as input:

- an $M$-dimensional `NumPy` array containing the **predicted** class labels for $M$ test instances;
- an $M$-dimensional `NumPy` array containing the **ground truth** class labels for $M$ test instances.

The method also takes an optional argument `class_labels`, which is a $C$-dimensional `NumPy` array containing the sequence of unique class labels (e.g. [‘A’, ‘C’, ‘E’]). If `class_labels` is given at runtime, then the confusion matrix should order their output according to the labels. For example, if `class_labels` is [‘C’, ‘E’, ‘A’], then the first row/column of the matrix should correspond to ‘C’. If `class_labels` is not given, the method will automatically generate a sorted set of class labels from the ground truth and use this ordering for the confusion matrix.

The method should return a $C \times C$ `NumPy` array as the confusion matrix, where $C$ is the number of classes. Please use the convention as in the lecture slides: the rows correspond to the ground truth, the columns correspond to predictions.

`LabTS` will test this method with some arbitrary data. There is no need to discuss this task in your report, unless there is something in your implementation that you wish to highlight or share.

### Task 3.2 (2 marks)

Complete the `accuracy()` method of the `Evaluator` class in `eval.py`. The method expects the confusion matrix from Task 3.1 as an input argument. The method should return the ***accuracy*** score (between 0.0 and 1.0 inclusive).

`LabTS` will test this method with some arbitrary data. There is no need to discuss this task in your report, unless there is something in your implementation that you wish to highlight or share.

### Task 3.3 (2 marks)

Complete the `precision()` method of the `Evaluator` class in `eval.py`. The method expects the confusion matrix from Task 3.1 as an input argument.

The method should return a tuple containing two objects:

- a $C$-dimensional `NumPy` array containing the ***precision*** scores for each class in the confusion matrix;
- a single ***macro-averaged precision*** score across classes.

`LabTS` will test this method with some arbitrary data. There is no need to discuss this task in your report, unless there is something in your implementation that you wish to highlight or share.

## Task 3.4 (2 marks)

Complete the `recall()` method of the `Evaluator` class in `eval.py`. The method expects the confusion matrix from Task 3.1 as an input argument.

The method should return a tuple containing two objects:

- a $C$-dimensional `NumPy` array containing the ***recall*** scores for each class in the confusion matrix;
- a single ***macro-averaged recall*** score across classes.

`LabTS` will test this method with some arbitrary data. There is no need to discuss this task in your report, unless there is something in your implementation that you wish to highlight or share.

## Task 3.5 (2 marks)

Complete the `f1_score()` method of the `Evaluator` class in `eval.py`. The method expects the confusion matrix from Task 3.1 as an input argument.

The method should return a tuple containing two objects:

- a $C$-dimensional `NumPy` array containing the $F_1$ scores for each class in the confusion matrix;
- a single ***macro-averaged*** $F_1$ score across classes[a].

`LabTS` will test this method with some arbitrary data. There is no need to discuss this task in your report, unless there is something in your implementation that you wish to highlight or share.

[a]For this coursework, we define macro-averaged $F_1$ as the mean of $F_1$ scores across the classes. An alternative definition is to compute the macro-averaged $F_1$ as the harmonic mean between the macro-averaged precision and macro-averaged recall. Both definitions are used in the literature. See `https://towardsdatascience.com/a-tale-of-two-macro-f1s-8811ddcf8f04` for a discussion.

We will stress test your implementations in Tasks 3.1–3.5 on `LabTS` with different combinations of inputs.

## Question 3.1 (6 marks)

Train three separate decision trees on these datasets:

- `train_full.txt`
- `train_sub.txt`
- `train_noisy.txt`.

Report the following for all three models evaluated on `test.txt`:

- Confusion matrix
- Accuracy
- Recall, precision, and $F_1$ score per class
- Macro-averaged recall, macro-averaged precision, and macro-averaged $F_1$ scores.

Note that for this question, you are not awarded marks for high scores, but rather for whether the scores you reported match the ones in your implementation.

Note that we will also be testing whether your implementation is generalisable to a hidden, unseen test set (**6 marks**). To avoid losing marks, please make sure that your `DecisionTreeClassifier`'s `train()` and `predict()` methods work correctly to generate predictions.

Question 3.2 (4 marks)

Compare and comment the results of the three models. Which one performs the best? Worst? Give some insights into why. Which classes are accurate? Which classes are often confused? Which classes are they often confused as? Explain.

It is sometimes difficult to conclude definitively that one model is better than another based on just one test set. Your model might just so happen to perform better than another on one particular test set, and the opposite might happen if evaluated on another. Therefore, *k-fold cross-validation* is commonly performed to ensure that the better performance is not just by chance, but is consistent across different data splits. Cross-validation is also useful for selecting any optimal hyperparameters for your model. The next task gives you some experience in performing cross-validation.

Task 3.6 (2 marks)

Implement a function to randomly split the dataset into $k$ subsets. The function should be robust enough to handle any $k$ specified by the user.

Then implement a script/function to enable you to perform $k$-fold cross-validation.

In your report, please give the file name(s) of where your implementation is located and the names of your function(s)/class method(s). Also highlight any details of your implementation to help us understand your code.

Question 3.3 (2 marks)

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, $0.7854 \pm 0.0122$. What does it mean to have a small/large standard deviation in this context?

Question 3.4 (2 marks)

Now take the decision tree model with the highest accuracy from 10 fold cross-validation, and evaluate the model on `test.txt`. Does it perform better or worse than training on the full dataset, according to the different available metrics? Discuss.

Question 3.5 (3 marks)

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.

### Part 4: Pruning (15 marks)

In your decision tree implementation, you have most likely decided to only stop splitting when the dataset cannot be partitioned further. This may result in your decision tree overfitting the training data, especially if the nodes cover only a few training samples. One way to reduce the effects of overfitting is by pruning your decision tree.

In this section, you will implement a pruning function for your decision tree. A simple approach is to prune the tree to reduce the error (or equivalently increase the accuracy) on a validation set (you may use `validation.txt` for this purpose). This approach works as follows: Start from the node(s) at the maximum depth of the tree. For each intermediate node where all its children are leaf nodes, convert this node to a single leaf node (and set the class label by majority vote). Then compute the accuracy of both trees on the validation dataset. If the validation accuracy on this new pruned tree is higher than that of the unpruned tree, then prune the node. Repeat the process until the validation accuracy does not improve. Note that you do not have to use this approach – you are free to make your own design decisions, use a more efficient algorithm, or even design your own algorithm for pruning.

**Task 4.1 (10 marks)**

Implement your decision tree pruning function.

In your report, briefly explain how you implemented your pruning function, and any design decisions you made. Also specify the file name(s) containing your implementation, and the name of the function(s) that perform(s) the pruning.

**Question 4.1 (3 marks)**

Apply the pruning function to the decision trees trained on `train_full.txt` and `train_noisy.txt`. You may use `validation.txt` to compute the validation accuracy in both cases.

Analyse and discuss the results of testing on `test.txt`. Did pruning improve the accuracy (and/or other metrics) in both cases? Did pruning help improve the scores on `train_noisy.txt` more than it helped `train_full.txt`? Discuss.

**Question 4.2 (2 marks)**

Comment on the maximal depth of the tree that you generated for both datasets before and after pruning. What can you tell about the relationship between maximal depth and prediction accuracy?

## 4 Deliverables

**Submission deadline: Monday 10th Feb 2020 (7pm)**

Please submit the following electronically via CATe:

- The `SHA1 token` corresponding to the commit of your group's GitLab repository you wish to be assessed. The repository should contain these files as a minimum:
  - `classification.py`
  - `eval.py`
  - `README.md`: Please include further instructions (if any) that will allow us to understand/manually run your files if needed
  - Any other files required for your code to run

- `report.pdf`: A short report answering all the tasks and questions as requested. There is no limit to the number of pages, but we prefer quality over quantity. To help us speed up marking, please organise your report by Tasks and Questions, as follows:
  - Part 1
    - Task 1.1
    - Question 1.1
    - Question 1.2
    - Question 1.3
  - Part 2
    - Task 2.1
    - Task 2.2
    - Task 2.3
    - Question 2.1
  - Part 3
    - Task 3.1
    - etc.

## 5 Grading scheme

**Max Total = 100 marks**

- Part 1 (10 marks)
  - Task 1.1 (4 marks)
  - Questions 1.1–1.3 (2+2+2=6 marks)
- Part 2 (35 marks)
  - Tasks 2.1–2.3 (15+5+5=25 marks)
  - Question 2.1 (4 marks)
  - Robustness of tree construction on new training data (6 marks)
  - [Optional] Image-based tree visualisation (up to 3 bonus marks)
- Part 3 (35 marks)
  - Tasks 3.1–3.6 (2+2+2+2+2+2=12 marks)
  - Questions 3.1–3.5 (6+4+2+2+3=17 marks)
  - Performance on unseen test data (6 marks)
- Part 4 (15 marks)
  - Task 4.1 (10 marks)
  - Questions 4.1–4.2 (3+2=5 marks)
- Clarity, conciseness and readability of report (5 marks)

## Acknowledgements

This coursework is inspired by the coursework designed by Antoine Cully.

## References

[1] Peter W. Frey and David J. Slate. Letter recognition using Holland-style adaptive classifiers. *Machine Learning*, 6(2):161–182, Mar 1991.