Coursework 1: Decision Trees

Introduction to Machine Learning

Deadline: Friday 7th Feb 2025 (7pm GMT)

1 Overview

Please read this manual **THOROUGHLY** as it contains crucial information about the coursework assignment.

In this assignment, 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 **Gitlab hash** 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 decision tree 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 and analyse classifiers.

This coursework is deliberately designed to be somewhat open-ended to give you space to explore and to think about machine learning design decisions. We hope you will be able to learn a lot from doing this coursework!

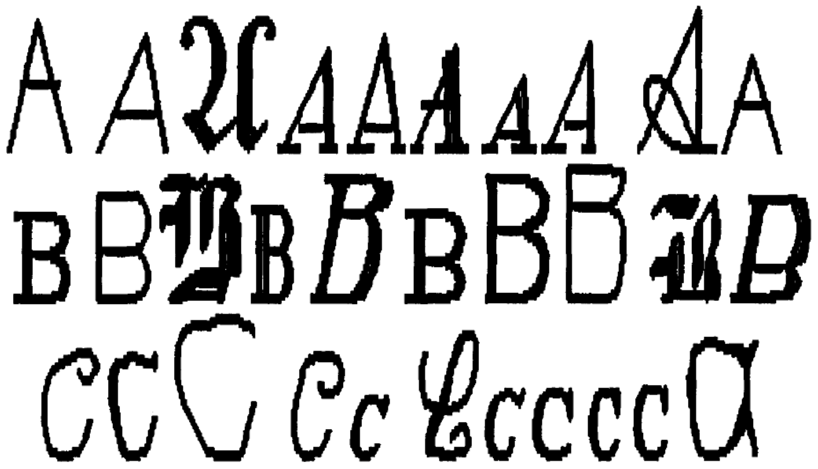


Figure 1: Your task is to implement a decision tree from scratch that can classify a preprocessed black-and-white pixel image to a letter in the English alphabet. The image is taken from Frey and Slate [1].

2 Detailed specifications

We have designed the specifications 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 to complete. You should also answer all questions in the red boxes in your report. Tasks in blue will not be assessed, but are necessary for completing the coursework.

Implementation

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

For Parts 2 and 4, you should implement your decision tree from scratch using only the Python Standard Library (`https://docs.python.org/3.12/library/`), `NumPy`, `SciPy` and `Matplotlib`. You should not use any other libraries (such as `scikit-learn`) to implement your decision trees.

Answers to Questions

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

Unassessed Tasks

These are tasks that you will need to do in order to progress with the coursework, but that will not be assessed (no marks awarded). There is no need to submit the code or write a report for these (unless you want to).

2.1 Setting up your environment

Please see the guidelines provided on Scientia to set up your machines for the coursework.

It is your own responsibility to ensure that your code runs on `LabTS`. **We reserve the right to reduce your marks by 30% for any bits of your code that cannot be run.**

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.

Reading in the datasets

Unassessed Task

Write a function or class method to read in the datasets from the given files. You can just adapt what you have developed in the optional lab tutorials for this.

The function/class method should 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. Your function/class method should return:

1. a `NumPy` array of shape (N, K) representing N training instances of K attributes;
2. a `NumPy` array of shape $(N,)$ containing the class label for each N instance. The class label should be a string representing the character, e.g. `"A"`, `"E"`.

These two `NumPy` arrays will be used as input to your decision tree in Part 2.

There is no need to submit your code for this.

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 check that you have understood the datasets, please answer the following questions in your report:

Question 1.1 (2 marks)

Besides the number of instances, what is another main difference between `train_full.txt` and `train_sub.txt`?

Question 1.2 (4 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 1.3 (4 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 (40 marks)

Your main task in this section is to **implement your own decision tree from scratch** using **Python 3**. As a reminder, you should only implement your decision tree using the Python Standard Library, `NumPy`, `SciPy` and `Matplotlib`. 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 integers. 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 integers, 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

```
 1: function INDUCE_DECISION_TREE(dataset)
 2:     if all samples have the same label or dataset cannot be split further then
 3:         return a leaf node with the majority label
 4:     else
 5:         node ← FIND_BEST_NODE(dataset)
 6:         children_datasets ← SPLIT_DATASET(node)
 7:         for child_dataset ∈ children_datasets do
 8:             child_node ← INDUCE_DECISION_TREE(child_dataset)
 9:             node.add_child(child_node)
10:         end for
11:         return node
12:     end if
13: 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 JSON format. Note that we will **not** be assessing this for the coursework. You should be able to train your decision tree on the full dataset in less than 5 minutes (even less than a minute) if implemented properly using `NumPy`.

Task 2.1 (28 marks)

Your task is to implement your decision tree algorithm that constructs a decision tree from training data. Keep your implementation simple and basic for this task. You will tweak your algorithm later in Part 4, so save it for then!

More specifically, complete the `fit()` method of the `DecisionTreeClassifier` class in `classification.py`. The method expects two arguments as input:

- a `NumPy` array of shape (N, K) representing N training instances of K attributes;
- a `NumPy` array of shape $(N,)$ containing the class label for each N instance. The class label should be a string representing the character, e.g. `"A"`, `"E"`.

The method should then construct a decision tree from the given input. The method does not return anything. The instance will instead be used later to predict 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 `fit()` with our own training sets to test its robustness. For example, we may test whether it can be trained on a different number of attributes or class labels. Therefore, please ensure that your code at least runs on `LabTS` to avoid losing unnecessary marks!

In your report, briefly describe 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.

You will be evaluated based on the quality of your design, justification of your design decisions, your implementation and the robustness of your implementation.

Task 2.2 (12 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:

- a `NumPy` array of shape (M, K) representing M test instances of K attributes.

The method should return a `NumPy` array with shape $(M,)$, with 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. We will also be testing whether your implementation is generalisable to a hidden, unseen test set. To avoid losing marks, please make sure that your `DecisionTreeClassifier`'s `fit()` and `predict()` methods both work correctly on `LabTS`.

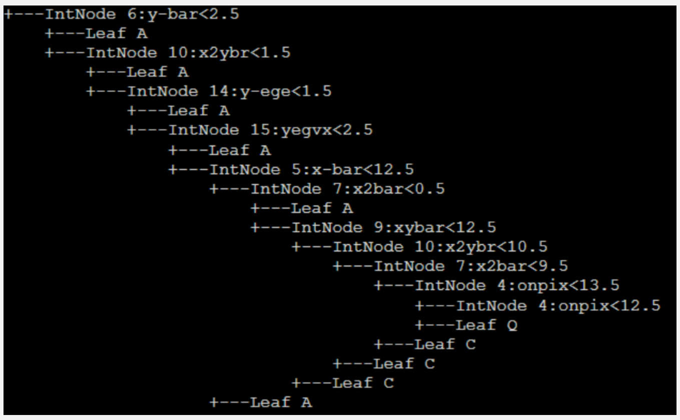
In your report, briefly describe your implementation (a paragraph is sufficient). Highlight anything that you think deserves our attention.

You will be evaluated based on your design and implementation, and on how well it generalises to the hidden test set.

Unassessed task

This part is **NOT** assessed, so please only do this in your spare time as your own challenge.

Decision trees have the advantage of being interpretable. Therefore, it is a good idea to examine the decision tree classifier that you have constructed. Visualising your decision tree is a good way to do this. You can implement a simple text-based visualisation (as in the image below); this can be done by indenting the nodes according to the tree depth. You can also create a image-based visualisation using visualisation libraries (`graphviz`, `plotly`). You can also display more information like entropy and/or the class distribution on the node.



```
+---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
```

You can examine the root node of the tree your algorithm have learnt. Does it make sense for this attribute to be the first split?

Part 3: Evaluation (20 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 evaluating and analysing the output of Machine Learning models.

Question 3.1 (12 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`. You can use the implementations from the optional lab exercises for this purpose.

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

Then compare and comment on the results of the three models.

- Which one performs the best? The worst? Give some insights into why.
- Which classes are accurate? Which classes are often confused? Which classes are they often confused as? Explain.

Note that for this question, you are not awarded marks for high scores, but rather for how you interpret and analyse your results.

Cross-validation

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.

Question 3.2 (4 marks)

Perform 10-fold cross validation on `train_full.txt`. You can use the cross validation code from the lab tutorials.

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?

Question 3.3 (4 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: Improving your decision tree (30 marks)

In this section, your task is to create a better decision tree. You have complete freedom on how you want to improve it. You may build on your decision tree from Part 2, or even consider an alternative design that you think might be more effective, or perform post-pruning, or combine different decision trees. We will consider either quantity (lots of small and simple combined tweaks) or quality (one or two very clever or effective changes). You may also train different trees separately and compare their performance. Impress us with your creativity and ingenuity (but don't go overboard – you do have many other courses and limited time!)

Task 4.1 (22 marks)

Implement your improved or new decision tree.

You are free to choose how to implement this, but you should keep your original `DecisionTreeClassifier` class from Part 2 for assessment. You can just create a subclass of `DecisionTreeClassifier`, or just copy your code and modify. Feel free to create a new Python script/module if needed.

In `improvement.py`, please complete the function named `train_and_predict()` that takes as input `x`, `y`, `x_test`, `x_val`, `y_val`. The function should construct your improved/new tree using `x` and `y`, and predict the labels for `x_test` using this improved/new tree. You may also use `x_val` and `y_val` as your validation dataset if required. You are free to structure anything else in any way you like, as long as you provide us with this function as an interface for `LabTS` to evaluate your improved/new classifier.

In your report, 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. Highlight anything that you did that you think is clever and worth mentioning. Basically, sell your work!

Question 4.1 (8 marks)

Train your new decision tree classifier on `train_full.txt` and `train_noisy.txt`. You may use `validation.txt` if needed.

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.

Again, we are more interested in how you analyse your results here, rather than your performance gain/loss.

3 Deliverables

Please submit the following electronically via Scientia:

- The `Gitlab hash` 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`
  - `improvement.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. As a guide, there should be no more than 10 pages. Note that **this is a maximum, not a target**! We prefer quality over quantity, i.e. the report should be clear, concise and readable, rather than long. To help us speed up marking, please organise your report by Tasks and Questions, as follows:
  - Part 1
    - Question 1.1
    - Question 1.2
    - Question 1.3
  - Part 2
    - Task 2.1
    - Task 2.2
  - Part 3
    - Question 3.1
    - Question 3.2
    - Question 3.3
  - Part 4
    - Task 4.1
    - Question 4.1

4 Grading scheme

Max Total = 100 marks (12% of final module grade)

- Part 1 (10 marks)
- Part 2 (40 marks)
- Part 3 (20 marks)
- Part 4 (30 marks)

References

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