

A2 - Decision trees

Due Mar 3 by 5pm **Points** 10

Assignment 2: Image recognition with decision trees (C structs and pointers)

Introduction

We saw in A1 that handwriting recognition with kNN leads to high accuracy. However, a major downside is that classification (i.e. computing labels of unseen images) can be quite slow, in particular for large images, since it requires a pixel-wise comparison between the input image and all images in the training data set. In this assignment we will therefore try a different approach to the same problem. We will use another fundamental technique from machine learning called decision trees. We will see that decision trees take a bit longer to build (i.e. to "train"), but then are much faster at classifying input images. We will also address another reason why our kNN based classifier was quite slow: it spent much of its time reading the input images making up the training and test data sets. One reason was that it required opening such a large number of files. Another reason was that the ASCII format of the files made them larger than necessary. We will fix this by providing input images in binary format instead of ASCII, and by using only two files, one that encodes all the training images and one that encodes all the test images.

Learning Outcomes:

By the end of this assignment you will be able to:

- read data from binary files
- use structs and dynamically allocated memory to create a tree data structure
- describe the key ideas of decision trees

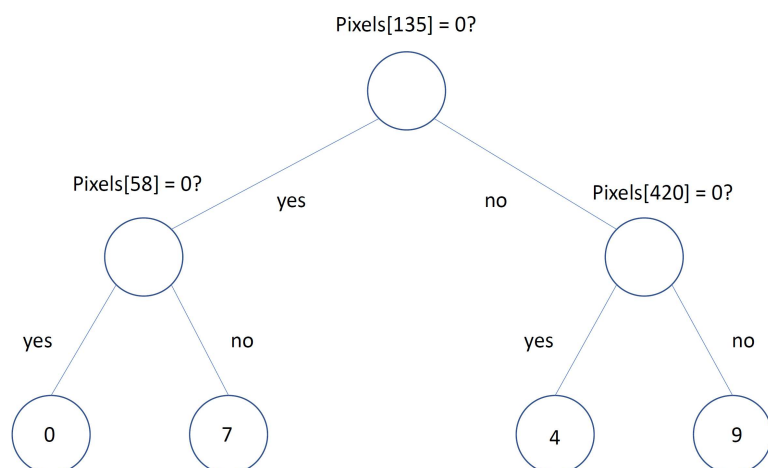
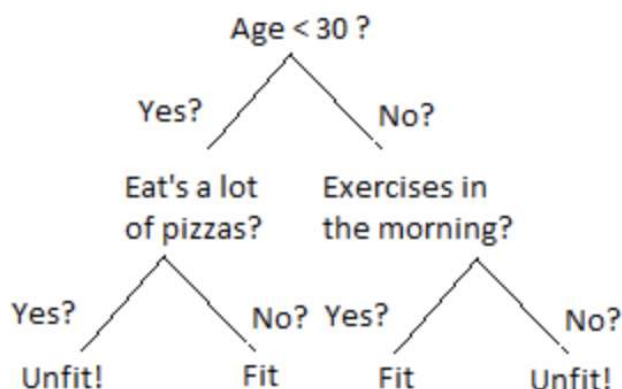
Decision Tree Basics

A decision tree is a tree data structure, that is used to classify input items (images in our case) based on their features (pixel values in our case) into classes (the digits 0-9 in our case). In this assignment we will only consider binary decision trees. Each internal node of the tree represents one feature (pixel) and based on the input item's value of that feature routes the input item either to its right or to its left child. A leaf node represents a classification, i.e. it stands for one particular class/label (one of the 10 digits in our case). To classify an input item using a decision tree we start at the root and follow the corresponding branches, based on the item's features. Once we reach a leaf node we output the class associated with that leaf node.

The binary decision tree below classifies humans into "fit" and "unfit" based on three features: their age, diet and exercise habits (image from xoriant.com). For example, a person that is younger than 30 years

and eats lots of pizzas is classified as "unfit".

Is a Person Fit?



Your goal in this exercise is to build a classification tree that uses the pixels of an image as its features and classifies an image into one of the classes 0 to 9. We will use only images of size 28X28 (=784) pixels, so each image has 784 features. And we have pre-processed the images we give you so that all pixels either have the value 0 (black) or 255 (white). Each node in the tree will make its decision based on one particular pixel.

The image above on the right shows a simplistic example for such a tree. For example, the root node will send an input image to its left child if the value of pixel #135 is 0 and to the right child if the value of that pixel is 1. An image whose pixel #135 is 0 and #pixel 58 is 255 is classified as depicting the digit 7. This particular tree is just a toy example for illustration and will have very poor accuracy.

Now, the question we need to answer is how to build such a tree that has good accuracy. We will get to that next ...

Building a Decision Tree

We will start building the tree starting with the root node, and then recursively build the sub-trees. There are two key design decisions when building a tree:

1. Choosing the stopping criterion, i.e. when does a node become a leaf node?

When we create a new node how do we decide whether that node is a leaf node that outputs a classification, or an internal node that routes inputs to its children? (This will also be the stopping criterion for your recursion.) And if we decide that a node is a leaf node, what classification should that node output?

We will use two different criteria for making a node a leaf node.

1) To keep the size of the tree reasonable we limit its depth to 20, i.e. the number of nodes (including root and leaf node) on any path from the root to a leaf is never larger than 20. That means if a child is at depth 20 it will be turned into a leaf node.

2) The second criteria is an optimization for cases where a node can produce a good classification even it is at a depth less than 20. To make this decision for a node we need to look at the labelled images that make up the training data for guidance. We identify all the images in the training data that would reach this node when following the branches starting from the root. We then look at the labels of those images. Imagine a case where all these images have the same label, e.g. they all represent the digit 2. In this case it seems safe to stop and output the classification 2, as there is no point in further branching. Even if say 99% of these images all have the label 2 we should probably stop and output classification 2.

More generally, we will use the following rule: if more than 95% of the training images that are routed to a given node belong to the same class Y, we make this node a leaf node which outputs class Y.

Any leaf node outputs the class that the majority of the training images that are routed to this node belong to.

2. Choosing which feature a node will branch on:

When we create a new node that is not a leaf node we need to decide which pixel this node will use to route input to its two children. In practice, machine learning researchers have investigated many different methods for choosing branching features. In this assignment, the starter code will provide you with a function that identifies the branching pixel for a node based on a metric called **Gini impurity**.


For the purpose of the assignment you do not need to understand how the function chooses a pixel, but we'll provide a bit of intuition here for those interested. Note that ideally, we want to use a pixel that provides us as much information as possible about the correct classification. For example, you could imagine that the first pixel of an image (top left corner) carries very little information, as it will likely be black for nearly all images, independently of the digit they represent. Branching on that first pixel will send images of all 10 classes to the right child (black). In contrast, we are looking for a branching pixel such that ideally most of the images that have the same value on this pixel (and therefore would be routed to the same child) also belong to the same class. That is the set of images that is routed to the

same child should ideally be *homogeneous* or *pure* in their labels. Gini impurity is one measure of how pure or homogeneous a dataset is. Choosing pixels to split the data that minimize the Gini impurity which is of the form

$$G = p(\text{class}=0 | \text{pixel value}) * (1 - p(\text{class}=0 | \text{pixel value})) + \dots + p(\text{class} = 9 | \text{pixel value}) * (1 - p(\text{class} = 9 | \text{pixel value})).$$

As the variance of a Bernoulli-distributed random variable is $p(1-p)$, we can see the Gini impurity directly measures the total variance over all classes assigned after splitting the data based on the pixel.

Therefore minimizing the Gini impurity makes the resulting data sets more homogeneous in the label which in turn increases the likelihood of correctly classifying the remaining data using the most frequent label.

A [diagram](#)  of a part of the decision tree may help you visualize the algorithm. Note that the indices arrays are not stored in the decision tree node because they are only used to build the tree.

Infinity and NAN

The `gini_impurity` function can return a value that is "not a number" (NAN). This can happen if we try to compute `0/0`, which can occur if a split of the two data sets contains zero members. We want to avoid choosing such splits because no additional information is gained if the model always assigns the images to the left or the right node.

We therefore don't want to use a pixel to split the data on if the `gini_impurity` function returns NAN. Fortunately this is quite easy to handle. First, there is a macro/constant `INFINITY` that you can use as the maximum value that `gini_impurity` might return. Second, NAN is not less than, greater than, or equal to any value, so the same boolean expression you use to find the smallest Gini impurity value will also likely filter out any values that are NAN.

The input format of the data

Your first step in writing this program will be to read in the image data. Since the image data is stored in binary format, you will need to use `fopen` and `fread` to read the data into the correct format. You should only iterate through the file once.

The first 4 bytes of the file contain an `int` that is the number of images in the remainder of the file. Each image is represented by 1 byte (`unsigned char`) containing the label for the image, followed by `NUM_PIXELS` bytes containing the pixel values of the image (as `unsigned chars`).

The full datasets in binary format can be found on teach.cs in `/u/csc209h/winter/pub/datasets/a2_datasets`. They are still pretty big, so you may still want to create a symbolic link to the directory in your account on teach.cs to save space:

```
ln -s /u/csc209h/winter/pub/datasets/a2_datasets datasets
```

We have also created a program called `gen_binary` that can be used to create your own smaller binary datasets using the text files. The source code is found in `/u/csc209h/winter/pub/datasets/a2_datasets/gen_binary.c`. We recommend that you also write a small program to print out the contents of the binary file in text format. This will also help you confirm that your `load_datasets` works correctly.

Some important hints

Here are a few things to remember as you implement and test the functions:

1. Make sure to carefully look over the starter code and any comments we provide in the code to help you understand what you need to implement.
2. You may want to add your own helper functions to `dec_tree.c`. You may not change any of the function signatures or data structures provided.
3. Test thoroughly before submitting a final version. We are using automated testing tools, so your program must compile and run according to the specifications. In particular, your program must produce output in the specified format.
4. Man pages are the best source of information for the system and library calls you need. Start by reading them carefully. Also remember that for the midterm and final exam we will provide you with the man pages for any commands you need to use. So this is good practice for the exams as well.
5. Check for errors on every system or library call that is not a print. Send any potential error messages to `stderr`.
6. Be very careful to initialize pointers to `NULL`.
7. Ensure that all dynamically allocated memory is freed before exiting the program. We will be using `valgrind` to test this.

Submission and Marking

Your program must compile on the teach.cs machines, so please test it there before submission. We will be using `gcc` to compile program with the flags `-Wall` and `-std=gnu99`, as demonstrated in the `Makefile`. Your program should not produce any error or warning messages when compiled. As in assignment 1, programs that do not compile will receive a 0. Programs that produce warning messages will be penalized.

Please submit your files by the deadline on MarkUs. You only need to commit the `.c` files, as the other files should not be modified. Do not commit the datasets or any executable files.