Image recognition with decision trees (C structs and pointers)

Introduction

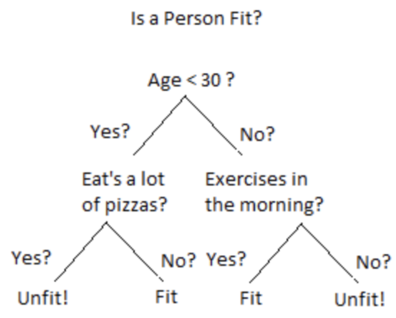
We saw in A1 that handwriting recognition with kNN leads to high accuracy. However, a major downside is that classification (i.e. computing labels of unseen images) can be quite slow, in particular for large images, since it requires a pixel-wise comparison between the input image and all images in the 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 some time to build (i.e. to "train"), but then are much faster at classifying input images.

We will also use a more realistic format for the input data. In A1 the images in the test and training set were provides as individual image files in ASCII format. In the real world, the input for machine learning problems is often formatted in a more compact way. In A2 we provide 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.

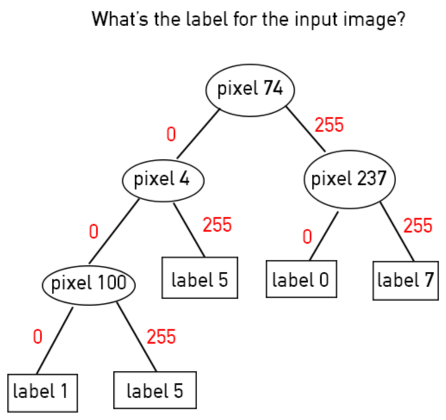
Decision Tree Basics

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".



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 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 #74 is 0 and to the right child if the value of that pixel is 255. An image whose pixel #74 is 0 and pixel #4 is 255 is classified as depicting the digit 5. 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:

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, which outputs a classification, or an internal node, which routes inputs to its children? (This would 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?

To decide whether we want to stop the recursion and for a leaf node, we will 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 have the same label Y, we make this node a leaf node which outputs label Y.

Any leaf node outputs the label that the majority of the training images that are routed to this node belong to (this label is necessarily unique because of the above condition).

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.***

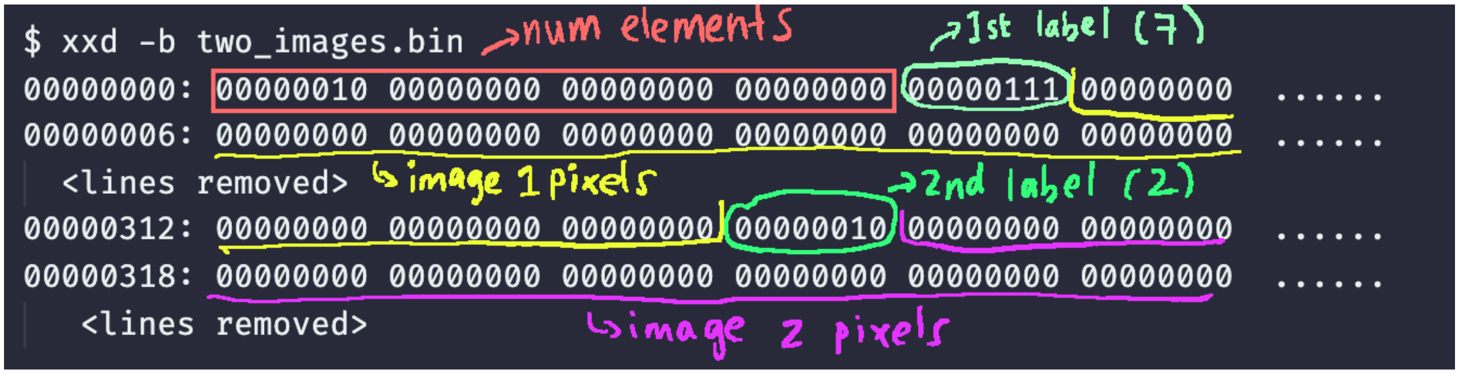
Note that ideally, we want to use a pixel that provides us as much information as possible about the correct classification. For example, you could imagine that the first pixel of an image (top left corner) carries very little information, as it will likely be black for nearly all images, independently 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 homogenous a dataset is.

The input format of the data

Each of the input binary files (representing one dataset) contain (1) The number of elements (i.e. images) in the dataset, and (2) the labels / pixel data for all of the images in the dataset. The bytes of the file are stored in the following way:

- The first 4 bytes in the file represent an integer *N*, which is the number of elements (i.e. images) in the dataset.
- 1 byte for the label for the 1st image, followed by 784 bytes for it's pixels
- 1 byte for the label for the 2nd image, followed by 784 bytes for it's pixels
- ...
- 1 byte for the label of the Nth image, followed by 784 bytes for it's pixels

So, for a dataset with *N* images, the total file size is *4 + (1 + 784)N* bytes. A convenient way to look at the binary data of the files is through a *hex viewer*. The program ***xxd*** is installed on the lab machines for you, and if you run it with a file, it will show you the representation of the data in hex / binary. Let's look at a simple example of a binary file that contains 2 images (found in `datasets/two_images.bin`) in the starter code. The command below passes in some flags to ***xxd*** to show the data directly in binary, by default it shows it in hexadecimal (which is easier to parse if you are familiar with it). The output is only partially shown here due to lack of space:



```
$ xxd -b two_images.bin
00000000: 00000010 00000000 00000000 00000000 00000111 00000000  ......
00000006: 00000000 00000000 00000000 00000000 00000000 00000000  ......
  <lines removed>
00000312: 00000000 00000000 00000000 00000010 00000000 00000000  ......
00000318: 00000000 00000000 00000000 00000000 00000000 00000000  ......
   <lines removed>
```

Some things to note about this output:

- The integer value first 4 bytes are stored in reversed order on the machine (little endian). So `00000010` is actually the *least significant* byte of the integer (and it represents the value 2). This is not an issue for any of the other data since each label and pixels' values take up only 1 byte. This is not something you need to worry about when reading the integer in your code, since the order of the bytes is handled by `fread()`, etc. We are just pointing this out so you are aware of what's going on when you look at the binary file.
- The numbers of the left side are the offset of the first byte in that row relative to the start of the the file (and are in ***hexadecimal***). We know that the 2nd image's label should appear at offset *4+1+784 = 789* in the file (after *N* and the first image's data). Converting 789 to hexadecimal gives us 315, which exactly corresponds to the offset according to the image (312 3).
- We can also verify by looking at the offset that image 1 has exactly 784 pixels (28*28 = 784).
- As mentioned earlier, the images have been processed to that each pixel is either completely black (`00000000`) or completely white (`11111111`).

While you are reading these file, you may find it helpful to try and output a few of the images into the PGM format and view them in an image viewer to make sure they look right and that they correspond to the label you have read.

Note: If you save the output of `xxd` into a text file, you can edit the bits by hand in a text editor and then use `xxd -r` on the edited text file to create a binary file. You can use this to create small files to help you debug. Unfortunately, this only works when you do a hexadecimal output from `xxd`, but can still be very useful. Example below:

```
# Save the output in output.txt (hex output)

$ xxd two_images.bin > output.txt

# Edit the file manually....

# Convert back into a new binary file

$ xxd -r output.txt > new_file.bin
```

You may find this website really helpful in converting between hex / binary / decimal: https://www.rapidtables.com/convert/number/decimal-to-hex.html