Michael Painter

Spectral Image Analysis for Medical Imaging

Part II Computer Science Tripos

Churchill College, 2016

May 10, 2016

Proforma

Name:	**Michael Painter**
College:	**Churchill College**
Project Title:	**Spectral Image Analysis for Medical Imaging**
Examination:	**Computer Science Part II Project Dissertation, May 2016**
Word Count:	**8432**[1]
Project Originator:	Dr Pietro Lio'
Supervisor:	Dr Pietro Lio' & Dr Gianluca Ascolani

Original aims of the project

TODO

Work completed

TODO

Special difficulties

None.

[1]This word count was computed by `texcount.pl -total diss.tex`

Declaration

I, Michael Painter of Churchill College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed

Date

Contents

List of Figures

List of Tables

Listings

Acknowledgements

I thank my supervisors, director of studies and tutor for their extensive support, especially for putting up with me throughout the year. I would also like to thank Malavika Nair, Ioana Bica and Yanie de Nadaillac for all of their support and helping me keep it together through the tougher parts of the year.

Chapter 1

Introduction

In this chapter we introduce the problem: what sort of data are we given and what would we like to do with it? We begin by introducing new terminology, and specifically defining terms that might otherwise be ambiguous. We briefly discus the motivation behind the implementation of this system, where we then suggest some possible methods that could be used to implement a solution, decide one that we might want to explore and what we might hope to discover by exploring said solution. Finally we will outline the overall pipeline of the system and define an interface for the system.

1.1 Terminology and preliminary definitions

To begin with we will introduce some terminology and preliminary definitions that we will use throughout the dissertation, including the rest of the introduction chapter.

Firstly we will use $\mathbb{N}_k$ where $k \in \mathbb{N}$ to denote the set $\{n \in \mathbb{N} \mid n < k\} = \{0, 1, ..., (k-1)\}$. We will also use $\mathbb{B}$ to denote the set $\{0, 1\} = \mathbb{N}_2$, for a boolean choice.

Typically we use the word *spectral* to refer to use of the Fourier domain, for example *spectral methods* refers to the use of Fourier domain to solve differential equations. For our purposes we will say that the *spectrum* of some signal is the Fourier transform of the time-domain signal, more specifically, we will usually be referring to a quantised spectrum. We will use the word *spectrum* to refer to either a continuous or quantised spectrum, for which the context should make it obvious which one is intended.

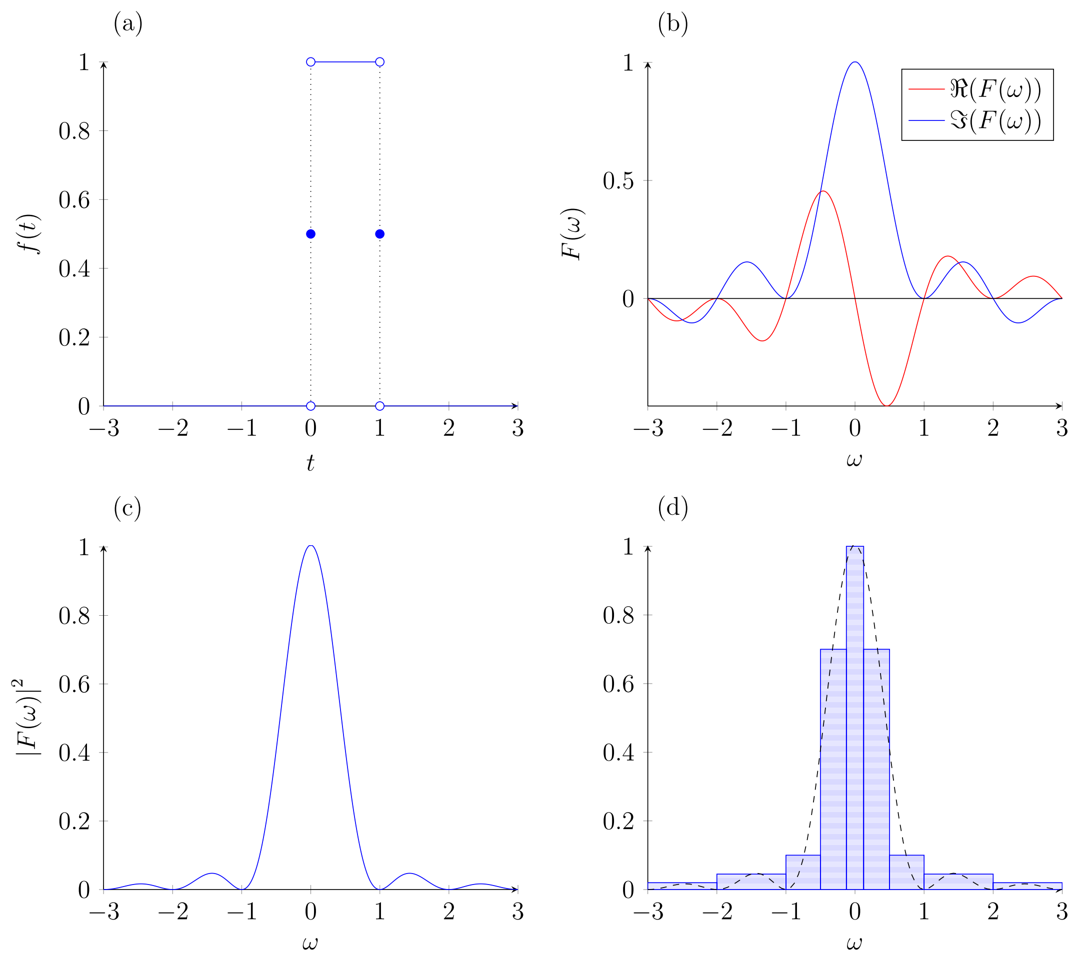

Figure 1.1: Examples of (a) a time-domain signal $f(t)$, (b) it's Fourier transform $F(\omega)$, (c) it's power spectrum $|F(\omega)|^2$ and (d) a quantisation of c.

We will use the term *spectral bin* to refer to a interval of wavelengths. Typically we will be referring to the case where we are dealing with quantised spectra, where the spectral bin refers to the range of wavelengths some bar in the histogram represents. For example consider (c) in figure 1.1.

We will use *spectral image* to refer to an image with an arbitrary number of frequency components. More formally we define an *image* as a function $I : \mathbb{N}_w \times \mathbb{N}_h \to \mathbb{N}$, where w is the width, h is the height. An image is simply a function of two spatial values. For a *spectral image* I' we extend the concept to a function $I' : \mathbb{N}_w \times \mathbb{N}_h \times \mathbb{N}_f \to \mathbb{N}$, where w is the width, h is the height and f is the number of spectral bins. We call this a spectral image, because at each 'pixel' (coordinate), we have a spectrum of values, that is we can define the spectrum s_{xy} of the pixel (x, y) by $s_{xy}(\cdot) = I'(x, y, \cdot)$. An intuitive way to think of a *spectral image* is a cube of values, so a spectral image is commonly referred to as a *data cube* (as in [18] and [3]) and we will use this interchangeably with the term *spectral image*.

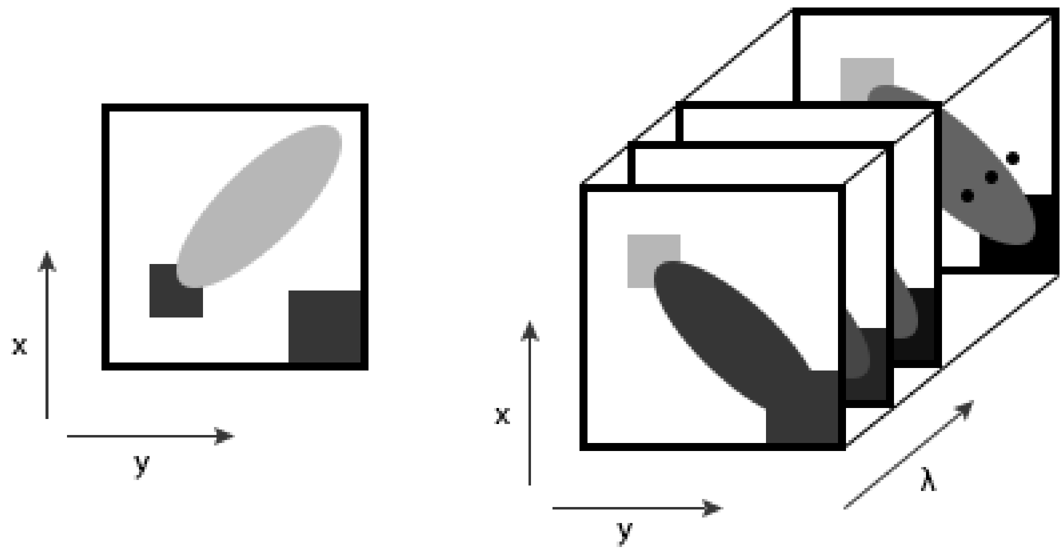

Figure 1.2: Examples of a monochrome image (left) and a spectral image (right).

We specifically note that a common form of spectral image is an "RGB" image, where we have $f = 3$, with a spectral bin for the red, green and blue regions of the visible spectrum.

Finally some literature refers to *multispectral* and *hyperspectral* images (which may be referenced such as in [19] and [3]), which are simply special cases of a spectral image. A multispectral image refers to a small number of spectral bands (such as 5) and a hyperspectral image refers to a large number of spectral bands (such as 50). Intuitively we can think of the difference between multispectral and hyperspectral in terms of their spectra, a multispectral image should be thought of as having quantised spectra (histograms), whereas a hyperspectral image can be thought of as attempting to approximate the continuous spectra.

1.2 What are we trying to do?

Nowadays computer vision techniques are used in abundance to aid medical diagnosis, and are used by doctors to help provide more accurate and earlier diagnoses. This is clearly an area of interest, as such tools may help improve, or even save lives where we wouldn't have been able to do so in the past.

In this project we consider using spectral images (recall that this includes RGB images) for medical purposes, and we wish to segment the image to indicate different regions of interest, typically indicating some form of disease of abnormality. The problem can easily be described mathematically: given an arbitrary spectral image $I : \mathbb{N}_w \times \mathbb{N}_h \times \mathbb{N}_f \to \mathbb{N}$ and a set of possible classes $\mathcal{C}$, we want to output a *pixel labelling* $\mathcal{P} : \mathbb{N}_w \times \mathbb{N}_h \to \mathcal{C}$ that correctly identifies regions of the image according to $\mathcal{C}$. Succinctly put, we want to perform *image segmentation*, on (noisy) medical (spectral) images.

The motivation for this project comes from use of a hyperspectral imager for use with contrast agents [19]. The aim is that the imager can be used to identify the presence of fluorescent contrast agents, some of which have a negative binding response to cancerous cells in the oesophagus, so allow cancerous tissues to be identified.

To summarise, we wish to produce a system capable of producing a pixel labelling from a spectral image. The pixel labelling will depend highly on the data and so we would likely wish to learn from some examples.

1.3 Possible solutions

- TODO: Describe datasets - Siri's, Tengs

As we will be looking to segment images, we will need to identify regions of the image based on the data available. One approach that we might take is to try and identify some properties manually, for example one possible solution in the [[Siri's]]dataset is to explicitly find the spectra of *endmembers* (the fluorescents) in the images and use a *spectral unmixing* algorithm [15] such as constrained least squares to identify the proportions of endmembers present at each given point [19].

An alternative that we will pursue is utilising machine learning methods, where we might hope to learn relationships within the data through supervised learning. Such a system may be more versatile, able to solve many similar problems and not constrained to a specific problem. For example, the spectral unmixing solution above requires the endmembers to be measurable, requiring significant work prior to putting the data through the algorithm. We would hope by using machine learning that we can avoid such additional work, and even solve problems where the 'endmembers' aren't separable, having explicitly measurable spectra.

1.4 Related work

Medical imaging is an active research area and will likely be for a long time. Medical imaging is ubiquitous in hospitals and an essential tool to aid doctors in diagnosing a patient. Along with medical imaging comes medical image analysis, where image processing methods are used to give more information to doctors which they may not be able to see with the naked eye.The task of image segmentation is important in medicine illnesses and viruses are commonly localised, only affecting parts of tissues and identification of which parts of tissues are healthy or not is important to successful treatment.

There is plenty of work and research that has gone into image segmentation, and it is common technique used in medical imaging, implemented using a variety of methods such as classifiers, thresholding and region growing [21]. However non-machine learning methods seem to be out of trend, with it being difficult to find papers written in this decade, whilst these methods are likely still being actively researched, and almost definitely still used in hospitals, they are currently outweighed vastly by machine learning papers.

Lately neural networks has seen a major dominance with regards to machine learning, however within medical imaging other areas of machine learning are still finding success, with whole books still dedicated to the topic [6]. True to the current trend, there is plenty of current work utilising neural networks in medical imaging, for example they have been successfully applied to segment different regions of the brain, such as 'white matter', 'gray matter', and 'cerebrospinal fluid' [28], and similarly for segmentation of brain tumours [29].

However, other machine learning methods are still of interest and random forests are still an active area of research within medical imaging and computer vision [4], especially because of their simplicity and lack of complexity, potentially making them very efficient [7].

SegNet [2] is a deep convolution neural network architecture built on top of Caffe [14] for pixel wise labelling, and provides some inspiration for the project. SegNet uses supervised learning to train the neural network, takes some image as input and outputs a pixel labelling. An extension to SegNet, Bayesian SegNet, includes an additional output image, which is a mapping of uncertainty in each pixel labelling, something that we will want to include in our own system. The Caffe implementation includes support for OpenCL (an open source GPU programming language), that allows for an efficient implementation of SegNet, realised by applying it to run on a real-time video stream, with 360 by 480 resolution. Whilst this implementation isn't necessarily geared towards medical purposes, it could certainly be utilised for a medical application, and is a good example of a system solving the more general problem of image segmentation/pixel labelling.

Chapter 2

Preparation

In this section we will introduce the area of image segmentation and how we may perform segmentation. Then we introduce decision trees and explain the supervised learning method of random forests. Many medical imaging techniques tend to require short exposure times they frequently exhibit visible noise, leading to grainy images. So, we then discuss how we may want to model the noise and how we can deal with this. In the second half of the chapter we outline the design with a requirements analysis, pipeline overview, what tools we will use and how we will evaluate our performance.

From a list of machine learning algorithms: State Vector Machine [16], [17], Hidden Markov Models [26], k Nearest Neighbour [8], Random Forests [6], and Neural Networks [12]. We decide that two algorithms that would be useful in our case are Random Forests and Neural Networks, so we will implement a Random Forests library and use the Encog library for Neural Networks in Java [13].

2.1 An introduction to image segmentation

Image segmentation concerns splitting an image into connected subsets, in some meaningful way. It is typically an ill-posed problem with no 'best' segmentation, however we often have some desired segmentation that we would like our system to perform, which we usually refer to as 'ground truth'. When training a supervised learning system we provide some number of ground truth images.

More formally if we let Ω be the set of pixels in some image, a valid image segmentation is $(S_1, ..., S_n)$ for some $n \in \mathbb{N}$ which satisfy the following:

$$\Omega = \bigcup_{i=1}^{n} S_i \tag{2.1}$$

$$S_i \cap S_j = \emptyset \qquad \text{for all } i \neq j \tag{2.2}$$

and each S_i is connected, which means that there is a path between any two pixels of S_i, where a path may take steps of one pixel up, left, down or right [20].

There are a number of ways that we may try to perform an image segmentation. One such method consists of finding the edges present in an image, which can be found using the zero crossings of some gradient operator such as the Laplacian. Furthermore, we may want to look at edges at different scales, which suggests blurring the image before taking the gradient operator, leading to the Laplacian of Gaussian operator, as discussed by Pal et al [20]. An even simpler method would be to use grey level threshold, where we segment images depending on if they fall above or below some threshold, also discussed by Pal et al [20].

Finally, the method that we will use is to generate some pixel labelling $\mathcal{P} : \Omega \rightarrow \mathcal{C}$ as described in section 1.2. This implicitly defines a segmentation of an image via regions of similarly classed pixels.

2.2 Supervised learning for classification

Before we discuss any machine learning methods we first need to define some terminology that we will be using. In classification we are given a *feature vector* (or *instance*) $\mathbf{x} = (x_1, x_2, ..., x_d) \in \mathbb{R}^d$ and a set of classes $\mathcal{C} = \{C_1, C_2, ..., C_n\}$. We want to *learn* some hypothesis $h : \mathbb{R}^d \rightarrow \mathcal{C}$ which maps instances to their classification. To learn what hypothesis/hypotheses are appropriate we rely on a *training sequence* $\mathbf{s} = ((\mathbf{x}_1, c_1), (\mathbf{x}_2, c_2), ..., (\mathbf{x}_m, c_m))$, hence why the method is *supervised*. We usually assume that the training sequence is noisy, which can be sufficiently represented by noise just in the classes of $\mathbf{s}$, that is $c_1, ..., c_m$ and so we can consider each of the c_i to be a random variable, and hence consider $\mathbf{s}$ to be a random variable. Two sensible approaches that we might take to picking some hypothesis h are picking

$$h_{\mathrm{ML}} = \underset{h \in \mathcal{H}}{\arg\max} \Pr(h|\mathbf{s}) \tag{2.3}$$

or picking

$$h_{\text{MAP}} = \arg\max_{h \in \mathcal{H}} \Pr(\mathbf{s}|h) \Pr(h). \tag{2.4}$$

Where ML stands for *maximum likelihood* and MAP stands for *maximum a-posteriori*. The MAP hypothesis allows the prior $\Pr(h)$ to be used (indicating some prior knowledge about the hypothesis), however if $\Pr(h)$ is uniform, then the ML and MAP hypothesis are easily shown to be equivilent using Bayes' theorem. These two methods of picking a hypothesis are referred to as Bayesian learning methods, as we look to maximise likelihoods and we will see that this is not the only way to pick a suitable hypothesis. [25]

We often also see that some hypothesis will take the form of $h' : \mathbb{R}^d \to (\mathcal{C} \to [0, 1])$, that is it gives a probability distribution over classes. We can then simply set

$$h(\mathbf{x}) = \arg\max_{c \in \mathcal{C}} h'(\mathbf{x})(c) \tag{2.5}$$

which is 'the most likely class'.

2.3 Decision trees and random forests

We will now define what a decision tree is, how we can use it in supervised learning and then extend it onto the concept of random forests. As mentioned in section 2.2 we opt for a hypothesis which outputs a probability distribution over classes as opposed to a single class, which can be used to give a single class as mentioned previously.

2.3.1 Introduction to decision trees

Decision trees consist of a simple binary tree structure. Intuitively we think about asking a question at each node, makeing a decision on which child node to traverse to and eventually reach some conclusion at a leaf node. We can use this model in the situation described in section 2.2, where we are allowed to 'ask a question' about the feature vector at each decision node and leaf nodes consider

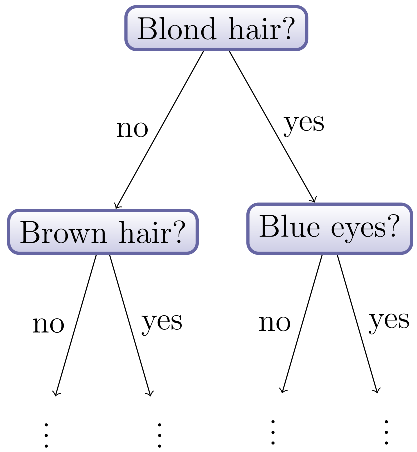

Figure 2.1: Example of (part of) a tree used to classify humans.

More formally we can define a decision tree T for classification, which has a set of states $Q \subseteq \mathbb{N}$, where if $i \in Q$ is a decision node, then it has children $2i, 2i+1 \in Q$.

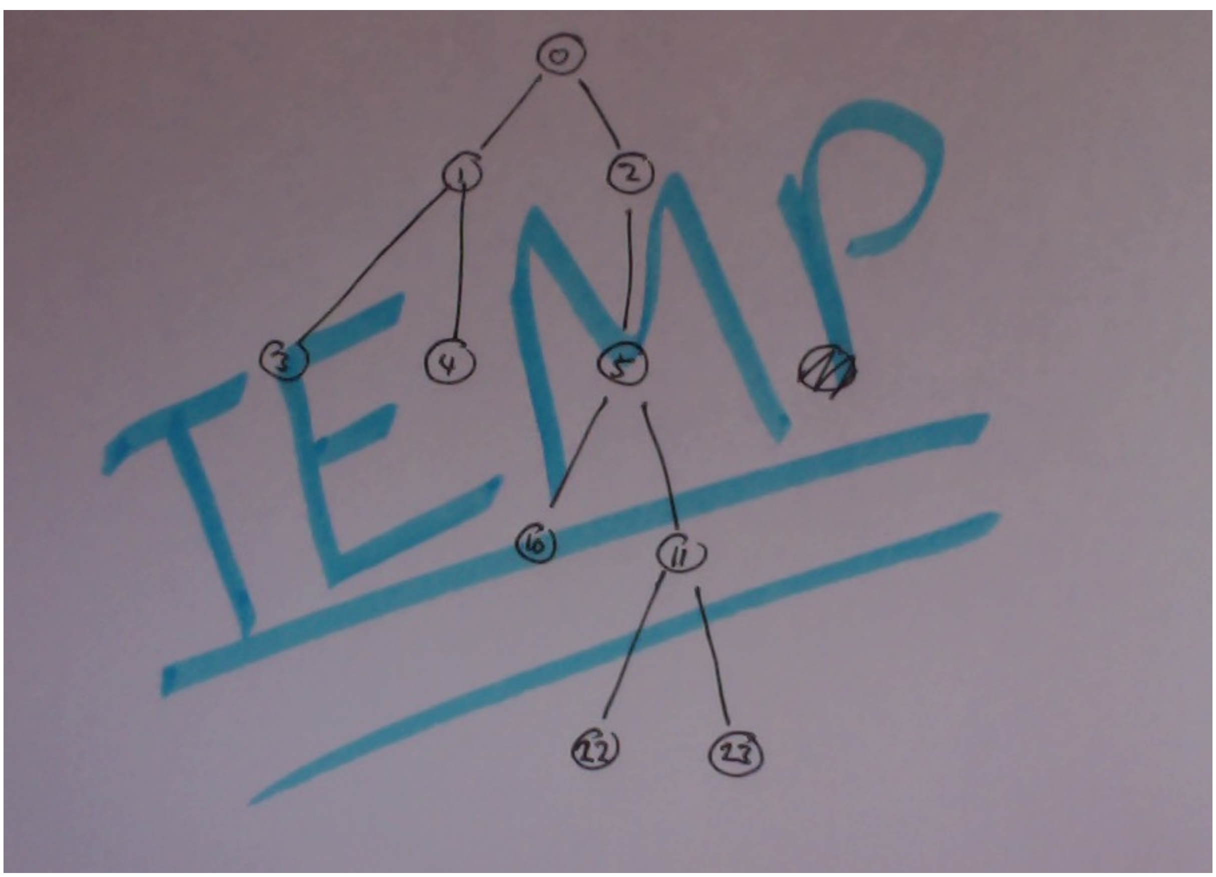

Figure 2.2: Example of node numberings in a decision tree.

For our model we specify some function $f : \mathbb{R}^d \times \mathcal{T} \to \mathbb{B}$, where $\mathcal{T}$ is some parameter space and is used to specify a function from $\mathbb{R}^d$ to $\mathbb{B}$ at each decision node. We call f a *weak learner* or a *split function*. In a decision tree for each state $i \in Q$ we have some associated $\theta_i \in \mathcal{T}$ called a *split parameter*, used to specify $f_i(\mathbf{x}) = f(\mathbf{x}; \theta_i)$. The function[1] $f_i : \mathbb{R}^d \to \mathbb{B}$ is then used to make a decision to traverse to state $2i$ or $2i+1$ next. Finally when we hit a leaf node $j \in Q$ we need to specify some probability distribution $p_j : \mathcal{C} \to [0, 1]$ where we say

$$p_j(c) = \Pr(\mathbf{x} \in c | \mathbf{x} \text{ traversed to } j \text{ in } T). \tag{2.6}$$

As can be seen in figure 2.3 to classify some instance $\mathbf{x}$ using a decision tree T we simply traverse T using f_i at each node $i \in Q$ to make a decision whether to traverse to $2i$ or $2i+1$ next. We usually use the rule if $f_i(\mathbf{x}) = 0$ then traverse to $2i$, otherwise if $f_i(\mathbf{x}) = 1$ then traverse to $2i + 1$. [6]

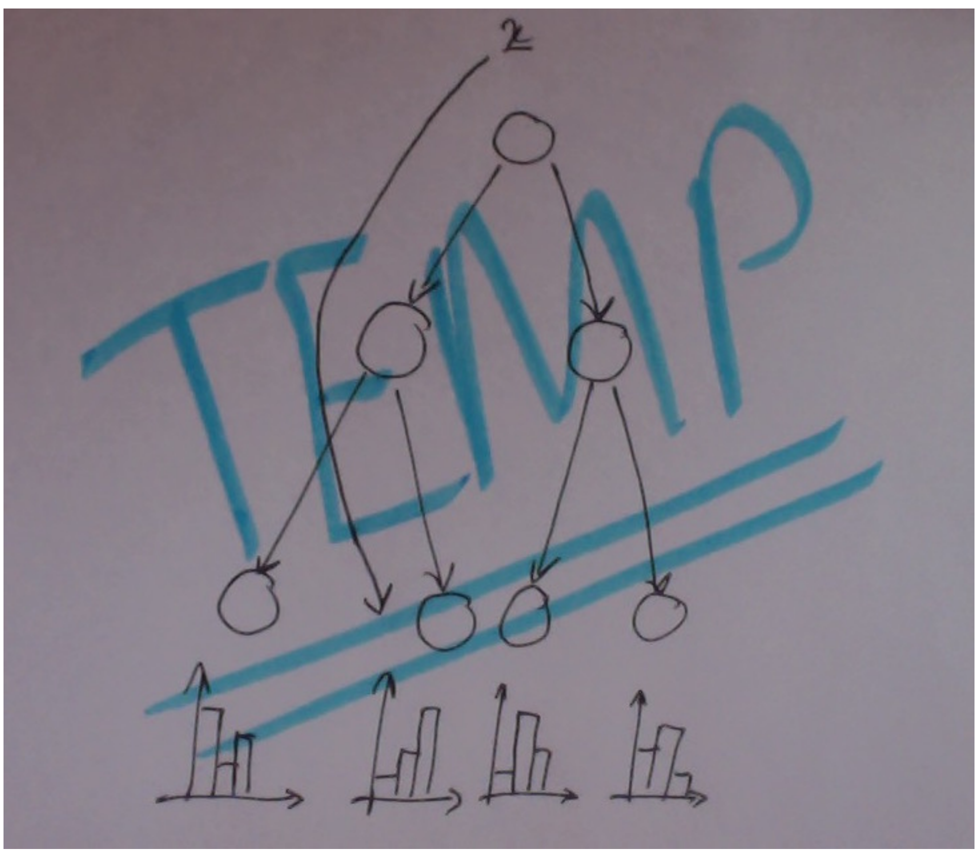

Figure 2.3: Example of how some instance $\mathbf{x}$ would be classified using a decision tree.

[1] $\mathbb{B}$ is defined in section 1.1

2.3.2 Training a decision tree

When we train a decision tree we use a greedy approach, we train at each node optimally. We must use a greedy approach because optimising over all possible trees is a computationally infeasible task (it is an NP-complete problem to decide if a tree is optimal). So consider having a set of states Q and for each $i \in Q$ wanting to learn the split parameter $\theta_i \in \mathcal{T}$. Let $\mathbf{s}_i$ be the training sequence that i is trained on. It is sensible to let $\mathbf{s_0} = \mathbf{s}$.

We first define the *entropy* of a training sequence

$$H(s) = -\sum_{c \in \mathcal{C}} p(c) \log_2(p(c)). \tag{2.7}$$

We assume feature vectors in s to be unique for ease of notation and let

$$p(c) = \frac{|\{\mathbf{v} \mid c' = c \wedge (\mathbf{v}, c') \in S\}|}{|S|}, \tag{2.8}$$

which is the empirical probability of a training sample vector in s being classified as c. Now we define a left and right split of $\mathbf{s}$, using some split parameter $\theta \in \mathcal{T}$ according to our weak learner f as follows

$$L(\mathbf{s}, \theta) = \{(\mathbf{x}, c) \in s | f(\mathbf{x}, \theta) = 0\} \tag{2.9}$$
$$R(\mathbf{s}, \theta) = \{(\mathbf{x}, c) \in s | f(\mathbf{x}, \theta) = 1\}. \tag{2.10}$$

We then consider the *information gain* I for performing a split according to θ of

$$I(\mathbf{s}, \theta) = H(\mathbf{s}) - \frac{1}{|\mathbf{s}|} \left(|L(\mathbf{s}, \theta)| H(L(\mathbf{s}, \theta)) + |R(\mathbf{s}, \theta)| H(R(\mathbf{s}, \theta)) \right). \tag{2.11}$$

We now have a way that we might want to pick θ_i at each node

$$\theta_i = \arg\max_{\theta \in \mathcal{T}} I(\mathbf{s}_i, \theta). \tag{2.12}$$

Once we have found a θ_i we then set $s_{2i} = L(\mathbf{s}_i, \theta_i)$ and $s_{2i+1} = R(\mathbf{s}_i, \theta_i)$, which allows us to begin training at node 0 with $\mathbf{s}_0 = \mathbf{s}$ and then recursively train down the tree. [6]

2.3.3 Moving swiftly on to random forests

Decision trees have a couple major disadvantages, which are solved by extending the concept to random forests. The first problem is that we learn θ_i using

$$\theta_i = \arg\max_{\theta \in \mathcal{T}} I(\mathbf{s}_i, \theta), \tag{2.13}$$

which is a problem because $|\mathcal{T}|$ may be infinite. We solve this problem by randomly sampling $\mathcal{T}$ with ρ values, so let $\mathcal{T}_\rho = \{\theta_i^{(1)}, ..., \theta_i^{(\rho)}\} \subseteq \mathcal{T}$. So we instead set

$$\theta_i = \arg\max_{\theta \in \mathcal{T}_\rho} I(\mathbf{s}_i, \theta), \tag{2.14}$$

when training at node i, and a *fresh* ρ samples are taken from $\mathcal{T}$ per node. This optimisation is called the *random node optimisation*. The second problem with decision trees is that we can easily over fit and also we cannot solve some XOR/parity problems [6]. We can solve this problem by using a *forest* of decision trees. A forest F is simply a set of trees, that is $F = \{T_1, T_2, ..., T_k\}$, where each T_ℓ is trained separately with training sequence $\mathbf{s}$. As we randomly trained each tree on the same data, it is reasonable to say that the probability of any given tree in F being the 'correct' tree is equally likely, hence we set $\Pr(T_\ell) = 1/k$ for each ℓ. Finally, when we wish to classify some instance $\mathbf{x}$ using a random forest we output

$$\begin{aligned}
p(c) &= \Pr(\mathbf{x} \in c) && (2.15)\\
&= \sum_{\ell=1}^{k} \Pr(\mathbf{x} \in c, T_\ell) && (2.16)\\
&= \sum_{\ell=1}^{k} \Pr(\mathbf{x} \in c | T_\ell) \Pr(T_\ell) && (2.17)\\
&= \frac{1}{k} \sum_{\ell=1}^{k} \Pr(\mathbf{x} \in c | T_\ell) && (2.18)
\end{aligned}$$

and each $\Pr(\mathbf{x} \in c | T_\ell)$ is simply the probability given by traversing each tree. [6]

Simply put, to classify using a forest, we just take an average of the outputs from the trees.

2.4 Neural Networks

We now move on to a describe a second method of supervised learning, artificial neural networks. We will begin by defining a single artificial neuron (referred to as just neurons and neural networks henceforth) and we look at how it operates. We then move onto looking at a network of neurons and how we can vary the weights in the network to train it.

TODO

2.5 Imaging and image noise

We now consider image sensor arrays, we want to know at a high level how images are captured and what might cause image noise. We will the define what the quantum noise model is and why it is appropriate in many cases of medical imaging.

2.5.1 Image sensor arrays

All forms of digital imaging involve some form of sensor array. Two common technologies are used in image sensors, Charge-Coupled Devices (CCD) and Complementary Metal-Oxide Semiconductor (CMOS). These devices are laid out in a grid, which are typically combined with (colour) wavelength filters.

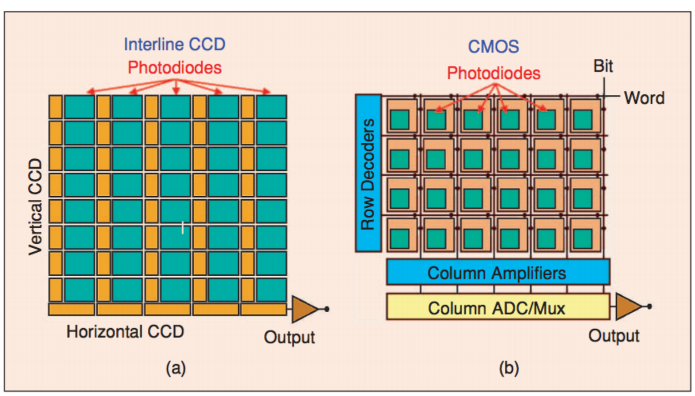

Figure 2.4: CCD and CMOS sensor arrays. Reproduced from Gamel et al [10].

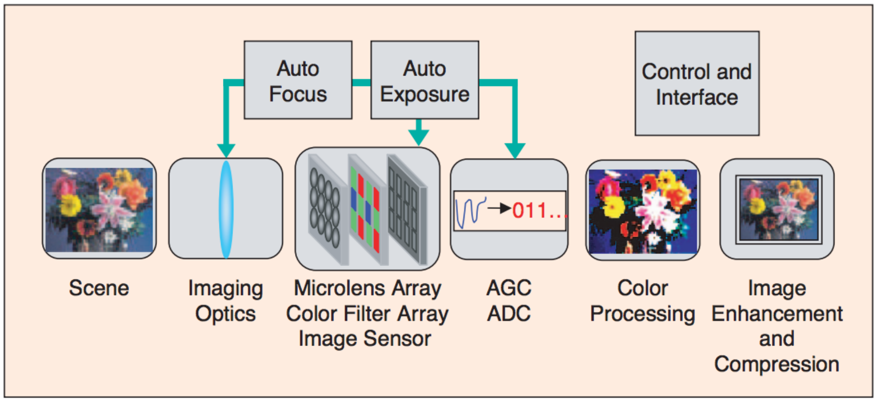

Figure 2.5: Overview of an imaging 'pipeline'. Reproduced from Gamel et al [10].

In a camera we will have a focussing lens, followed by wavelength filters and then our sensors. So it suffices to consider a model where we have parallel streams of photons hitting each sensor.

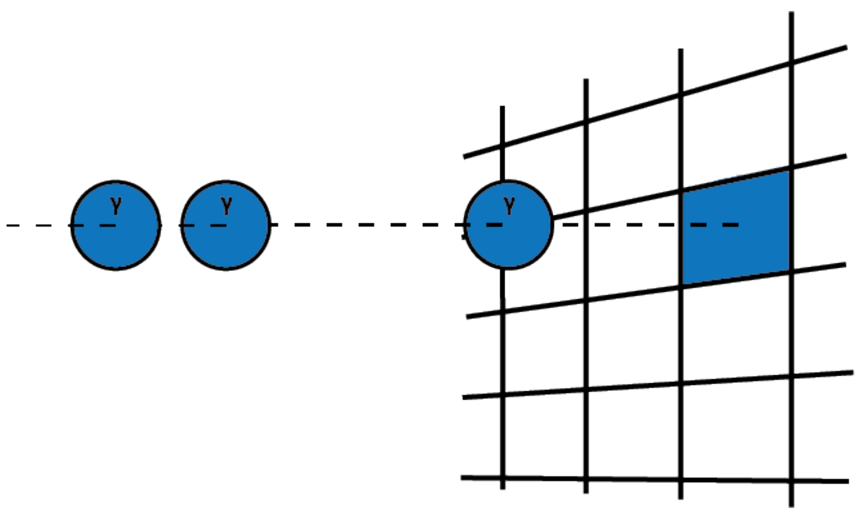

Figure 2.6: Parallel stream of photons incident on one sensor in a sensor array.

For our purpose it is enough to consider that a stream of photons, will be hitting some photodiode (the part of a sensor that induces a current when absorbing a photon), causing charge to collect on some capacitor, which will increase linearly with respect to time at a rate proportional to the rate of photons incident on the photoreceptor (the photocurrent), until it hits some maximum value. [10]

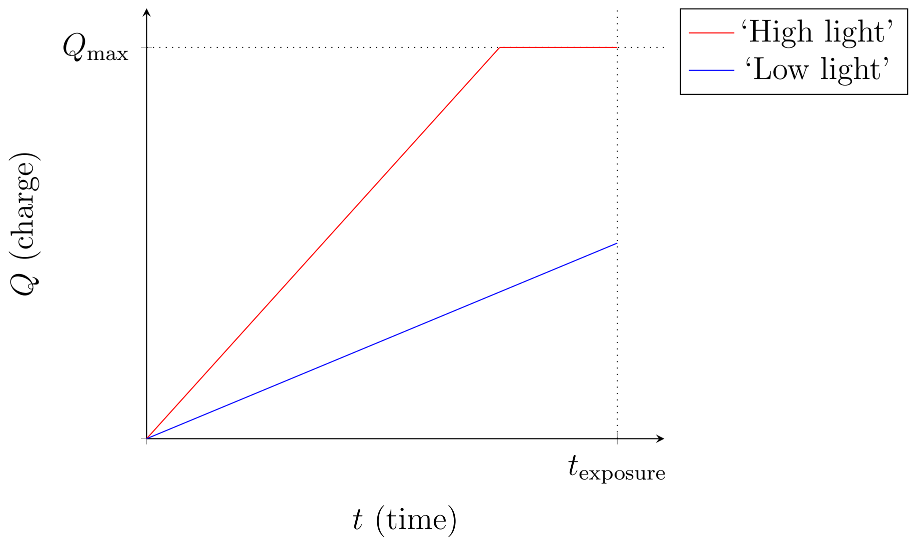

Figure 2.7: The charge held on a capacitor is linear with respect time, until it hits some maximum [10].

2.5.2 Quantum noise

Quantum noise, *Photon noise* or *Poisson noise* can be thought of as a uncertainty associated with the measurement of light, due to the quantised nature of electromagnetic waves and the independence of photon detections [11]. More mathematically suppose we have some noiseless image $I(x, y)$, then we define an image corrupted by photon noise $\tilde{I}$ by

$$\tilde{I}(x, y) \sim \text{Poisson}(I(x, y)). \tag{2.19}$$

So we have

$$\Pr(\tilde{I}(x, y) = k) = \begin{cases} e^{-I(x,y)} \frac{I(x,y)^k}{k!} & 0 \leq k \\ 0 & \text{otherwise} \end{cases} \tag{2.20}$$

$$\mathbb{E}\left[\tilde{I}(x, y)\right] = I(x, y) \tag{2.21}$$

$$\text{Var}\left[\tilde{I}(x, y)\right] = I(x, y). \tag{2.22}$$

We can see that this is appropriate considering figure 2.6 as a Poisson process, where each event is a photon hitting the photoreceptor. Suppose the exposure time to the sensor is t_{exposure}, the average rate of events is ϕ and that $N(t)$ is the number of events that occur in the interval $[0, t]$, then we have $N(t) \sim \text{Poisson}(\phi t)$

[24]. And so we have $N(t_{\text{exposure}}) \sim \text{Poisson}(\phi t_{\text{exposure}})$. From figure 2.7 we can justifiably say that if ϕ is the rate/intensity of photons hitting sensor at index (x, y) in the array, then $I(x, y) \propto N(t_{\text{exposure}})$.

The model is easily and obviously extended to spatial images by including spectral bin indices

$$\tilde{I}(x, y, \lambda) \sim \text{Poisson}(I(x, y, \lambda)). \tag{2.23}$$

Because of the filters (shown in figure 2.5) being spatially separated we can consider each value in the datacube to be an independent random variable. This means that

$$(x, y, \lambda) \neq (x', y', \lambda') \Rightarrow \Pr(\tilde{I}(x, y, \lambda), \tilde{I}(x', y', \lambda')) = \Pr(\tilde{I}(x, y, \lambda)) \Pr(\tilde{I}(x', y', \lambda')). \tag{2.24}$$

However, in images the values are constrained to a finite range, typically we have that for each x, y, λ that $I(x, y, \lambda) \in \mathbb{N}_{256}$, but this however doesn't restrict the value of $\tilde{I}(x, y, \lambda)$ to $\mathbb{N}_{256}$. By considering figure 2.7 we see that the charge becomes *saturated* at the upper limit, that is if we increase the photon intensity above the rate which achieves the maximum value, we still get the maximum value. To deal with this it makes sense to define a *Saturated Poisson* distribution. So we say $X \sim \text{SPoisson}(\mu, n)$ if

$$\Pr(X = k) = \begin{cases} e^{\mu} \frac{\mu^k}{k!} & 0 \leq k < n - 1 \\ \sum\limits_{i=n-1}^{\infty} e^{\mu} \frac{\mu^i}{i!} & k = n - 1 \\ 0 & \text{otherwise.} \end{cases} \tag{2.25}$$

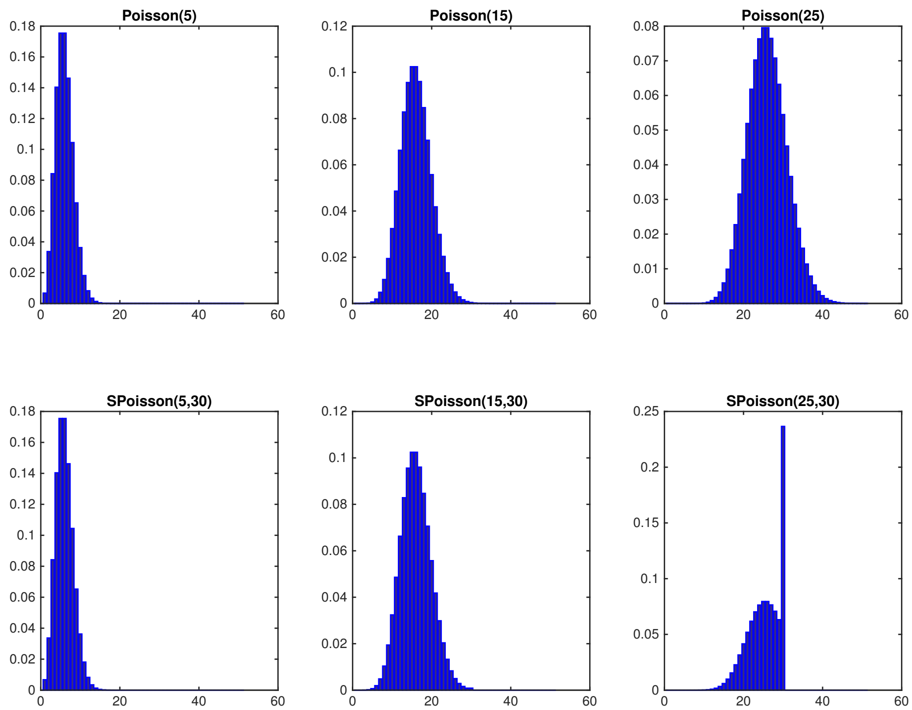

Figure 2.8: Example of Poisson(μ) and SPoisson(μ, 30) distributions, for $\mu = 5, 15, 25$.

2.5.3 Additive White Gaussian Noise

TODO??

2.5.4 Medical imaging noise

We end this section by considering how we should use our noise models ing medical imaging. For many methods of medical imaging we use ionising radiation that would be considered harmful given extended exposure times [22]. Because of this there is an inherent trade off between exposure time (for greater image clarity) and risk of causing damage [27]. Because we may potentially use shorter exposure times, or we may be imaging something in low light we find that often the quantum noise model is more appropriate and also cannot be approximated by Gaussian noise. However, there are many cases such as ultrasound where we may want to consider more sophisticated noise models [5], but for the purpose of this dissertation we will restrict ourselves to the above.

2.6 TVMM Image De-Noising

- A method developed for additive white Gaussian noise [9], which even when produced is outperformed by wavelet methods, outperforms algorithms developed for Poisson noise [23].
- Definition of total variation
- Definition of the optimisation problem and the Q function used
- Some properties of the Q function, that allow us to find the optimal solution to the optimisation problem
- Outline of the procedure
- Requires solution of linear equations. Can be done using conjugate gradients (appendix B).
- Trying to solve an inherently ill-posed problem

2.7 Requirements analysis

With the background reading out of the way, we move onto the design of the system. Firstly we carefully consider the requirements of the system. This is important to begin identifying the work that needs to be undertaken and where the potential risk lies.

Goal Requirement	Priority	Risk	Difficulty
Build a random forests machine learning library	High	Low	Medium
Incorporate a neural network library (Encog)	High	Medium	Medium
Image segmentation via pixel labelling	High	Low	Low
Suitably model and account for image noise	Medium	High	High
Train the pixel labelling system on real data	High	High	Low
Build an aid to help create training sets	Medium	Low	Low
Model uncertainty and output in a certainty map	Medium	Medium	Medium

Table 2.1: High level goals and desired outcomes for the project.

2.8 System Design

Here we outline the overview of the the whole system, where we begin with a work flow diagram comprising of files and system components. We then divide the system into it's separate components and use it to define an interface, along with a brief description of each module, defining inputs and outputs. The formats of the files that the user should input can be found in appendix A.

2.8.1 Pipeline/Overview

We now define here an overview of what we want the system to do (the pipeline of the system) in terms of the files input and output by different modules of the system. In the system overview below, blue boxes are code modules and red boxes represent files.

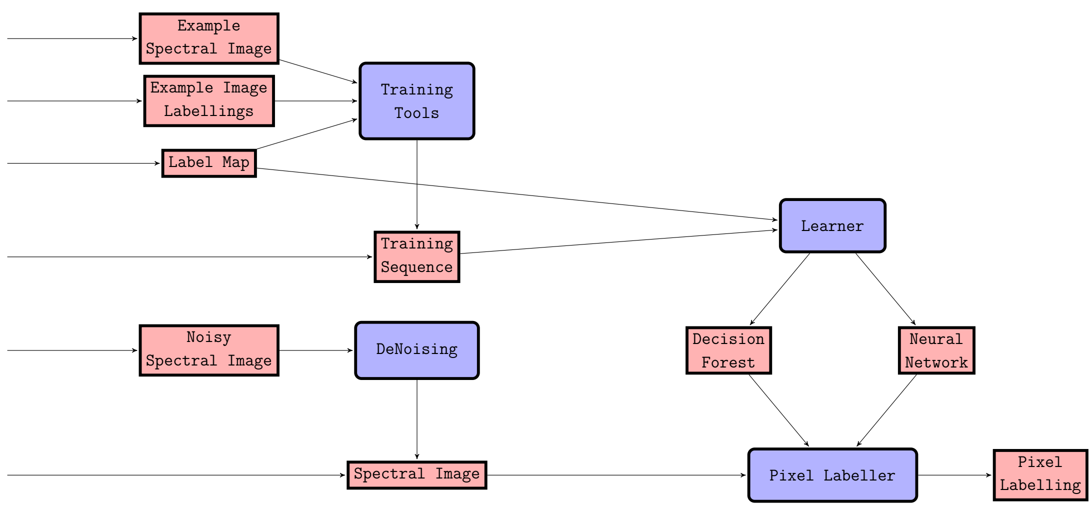

Figure 2.9: Overview of the system and its intended use.

2.8.2 Interface/Usage

We will allow usage for each module separately, however we will also allow for combined steps, that will avoid unnecessary intermediate files to be produced. For our interface we extend our diagrams that consisted of red boxes for files and blue boxes for code modules, to denote files/modules unused in a given function by a faded/duller colour, and we denote an intermediate file that isn't saved during some function by a regularly bright, but dashed boarder. It should be assumed that an intermediate file need not be provided by the user.

2.8.2.1 Training Tools

The training tools function is to take an example class map, pixel labelling and spectral image (see appendix A for how their formats). From this 'ground truth' we will output a training sequence as specified in appendix A. The use of this function is that we can generate vast quantities of training data with very little effort.

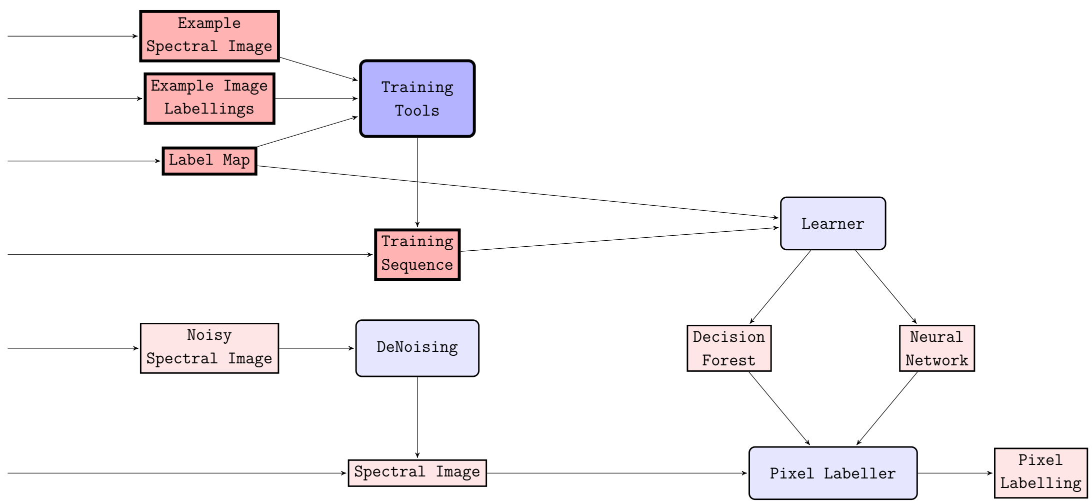

Figure 2.10: Overview of the training tools function.

2.8.2.2 Train

The train function will take a training sequence and then produce either a forest file or a neural network file. These files will be output to be used later by the pixel labeller.

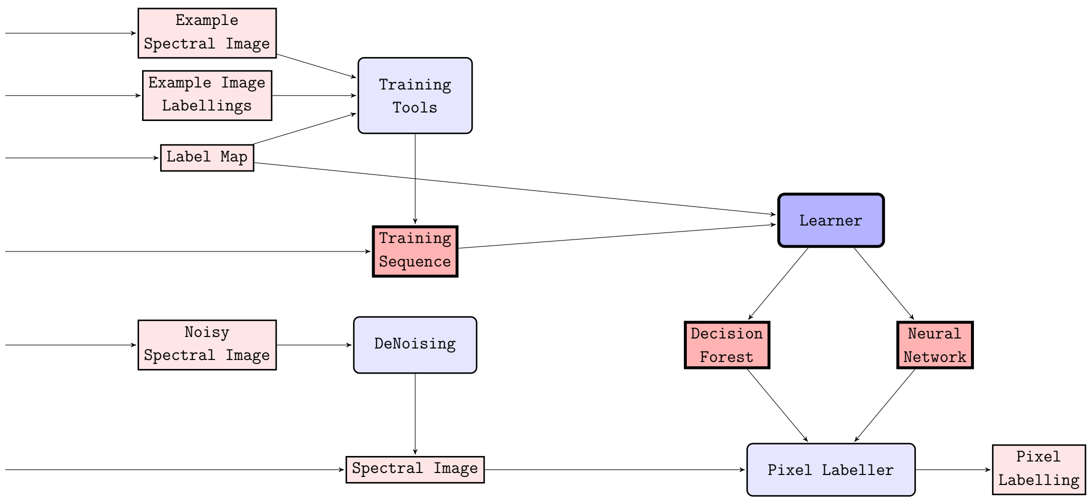

Figure 2.11: Overview of the train function.

2.8.2.3 Learn From Example

The learn from example function is a composite of the training tools and train functions. This will take the same inputs as the training tools function, but instead will immediately use the training sequence produced to train a random forest or neural network, *not* saving the training sequence in the process.

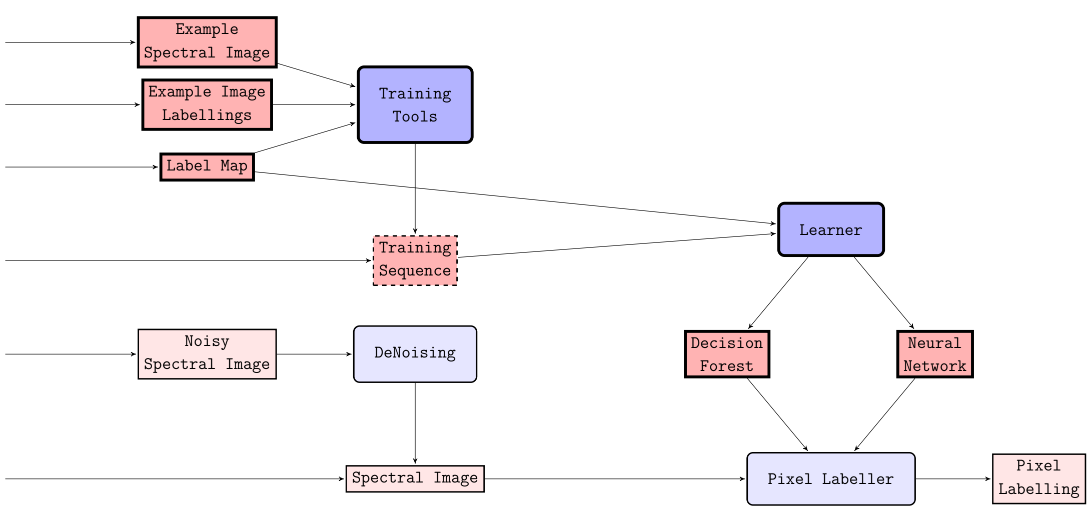

Figure 2.12: Overview of the learn from example function.

2.8.2.4 Pixel Labeller

The pixel labeller could be considered the ‘main’ part of the system. It uses the pixel labeller module to label a spectral image using a trained forest/neural network.

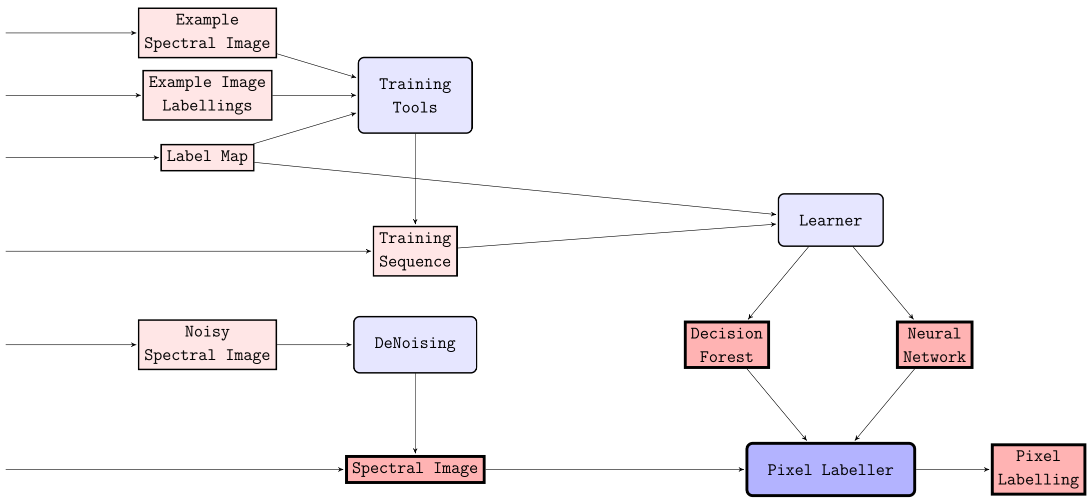

Figure 2.13: Overview of the pixel labeller function.

2.8.2.5 De-Noiser

The de-noiser function simply attempts to reduce the effect of noise on the image. It takes a spectral image as input and outputs the de-noised spectral image.

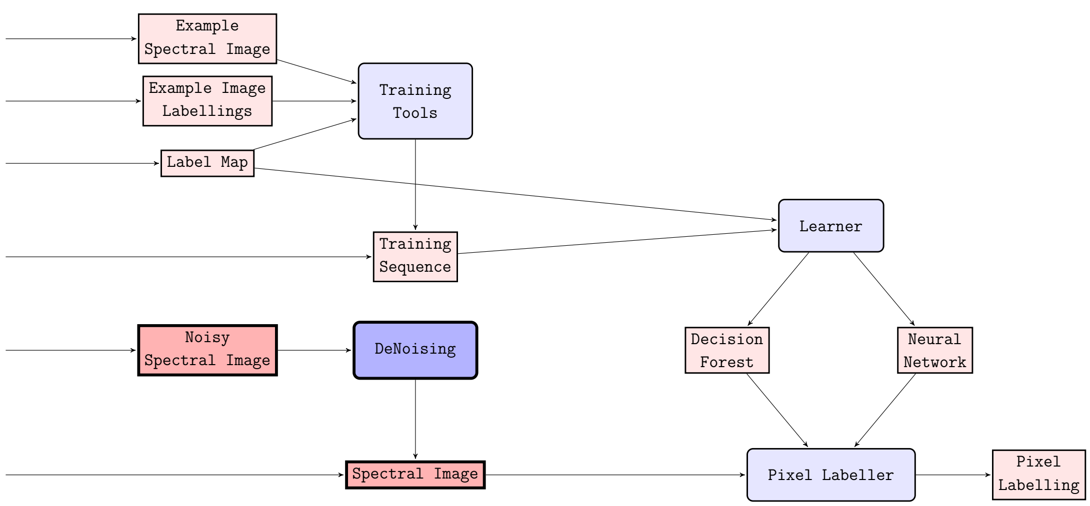

Figure 2.14: Overview of the de-noiser function.

2.8.2.6 Noisy Pixel Labeller

The noisy pixel labeller is a composite of the de-noiser and pixel labeller functions. We take a noisy spectral image as input, along with a trained forest or neural network. De-noising is run on the image before plugging the spectral image into the pixel labeller.

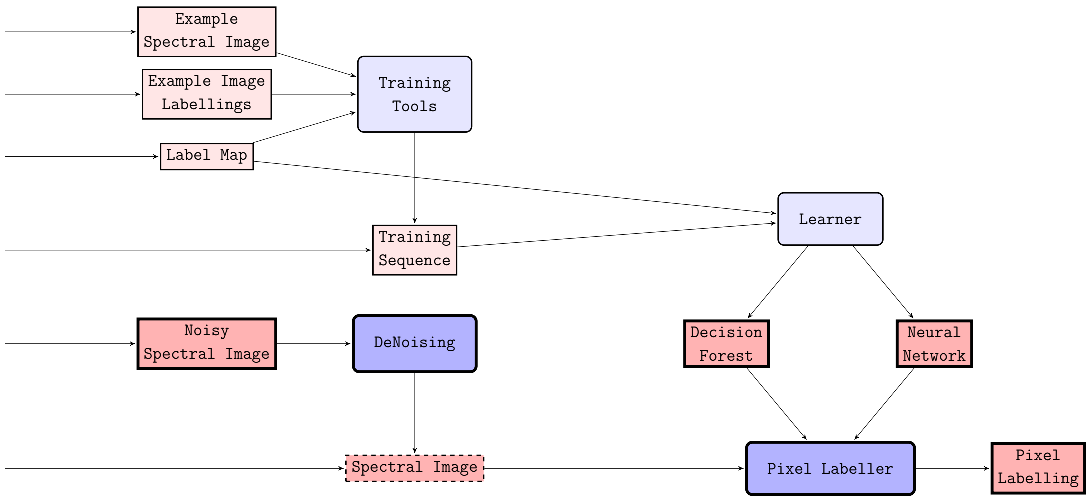

Figure 2.15: Overview of the noisy pixel labeller function.

2.9 Languages and tools

In this section we briefly describe the languages, libraries and tools that we used for the project.

Programming Languages and Libraries

Java: provides an OS independent language and is object oriented to allow for modular design.

JUnit: a Java library used for unit testing, used for white box and black box testing throughout the development of the system. Unit tests are essential to find bugs and prevent their re-introduction.

Encog: an easy to use neural networks library in Java/C# written by Heaton research [13].

Integrated development environment

Eclipse: allows for rapid development in Java and integrates easily with libraries and tools such as JUnit and Git.

EclEmma: an Eclipse plug-in the works with JUnit which highlights lines to indicate code coverage of unit tests.

ObjectAid: another Eclipse plug-in, used to create UML class diagrams.

Statistical analysis and visualisation

Matlab: an easy to use, extensible statistical package used for producing graphs.

Document preparation

LaTeX: used for typesetting this dissertation in a precise manor allowing control over most aspects of document layout and style.

Tikz: a LaTeX library useful for producing diagrams, such as figure 2.9.

Listings: a LaTeX library allowing colour formatted code for a plethora of languages.

Adobe Photoshop: image editing software used to produce some diagrams.

Version Control

Git: allows code to be developed systematically and rollback if necessary, as well as providing the ability to fork my core repository to try different strategies.

Backup

DropBox and Google Drive: repositories were kept in both DropBox and Google Drive folders, which synced every time a file was changed.

Github: provides a remote Git repository to store code, so also helps in provision of version control. The repository was updated every time a stable commit was made.

Time machine: Apple's automatic backup system, allowing for hourly backups of the whole hard drive to be taken.

2.10 Software engineering techniques

2.10.1 Development model

After the design of the system it was fairly obvious that a mixture of iterative and waterfall models. The system would have to be implemented in a modular form (the modules can be seen in figure 2.9). Each module then needed to be implemented and was tested rigorously before moving onto the next, which is important because of dependencies between modules. For each module it was possible to manually produce the input files necessary to perform testing.

Once a working system produce was produced, tested and we have confidence that it is correct, additional features and improvements to the system were added in an iterative manor, similar to a spiral model. Unit tests were used to make sure that any new additions don't break the existing, working parts of the system.

2.10.2 Testing

The development was test driven, and used a mixture of testing techniques to write unit tests. All tests were written as unit tests so that they could be reused later to make sure that the system still works. The following methodologies were employed:

Black box testing: this is when then internal system design not taken into account, used to make sure that the system works as a 'black box', in terms of the expected input/outputs we specified in the design. To assure that internal system design wasn't taken into account these were written *before* any code was written.

White box testing: this is when the internal system design is taken into account, tests are designed so that every line of code is checked using the unit tests. This is difficult to check manually, so Eclipse plug-in EclEmma was used, a code coverage tool that works with JUnit.

Input sanitisation testing: additional unit tests were written to make sure that erroneous inputs were handled correctly.

Incremental integration testing: this refers to using a bottom up approach, testing functionality as it is implemented.

Usability testing: a few users were asked to try use the system given only instructions in a readme file, the feedback from this was vital and shaped the design of the training tools module.

2.10.3 Backup Plan

It is important to make sure that we have an effective backup plan to avoid any software, hardware or user error that may result in a loss of work, something which is important to make sure that we do not loose a years worth of work.

To make sure that no work was lost a number of systems were set in place to not only regularly back up any work but also provide rollback if necessary. Google

Drive and DropBox were employed to keep shadow copies of the project directory, GitHub was used as a remote repository which was pushed to regularly. Finally Apple's time machine software was also used to automatically take backups of the whole local file system every hour.

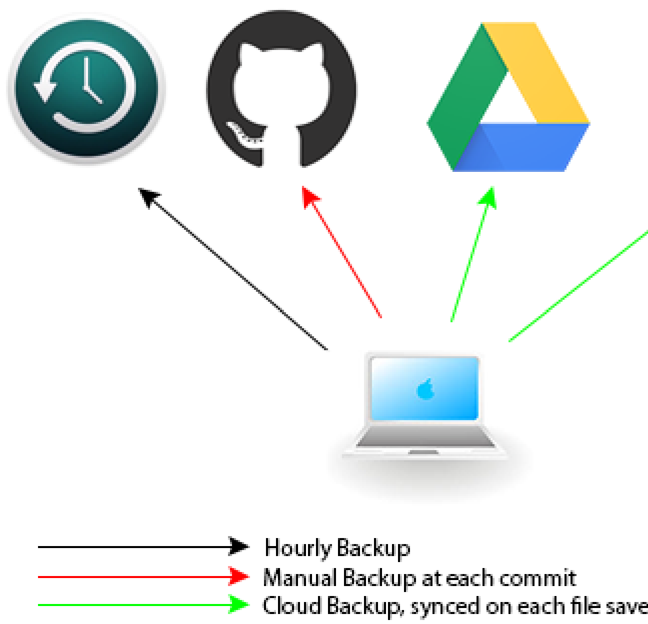

Figure 2.16: Overview of the backup strategy employed[2].

.

[2]Image modified from: apple.com, google.com, dropbox.com, github.com, clipartpanda.com

Chapter 3

Implementation

We begin this chapter by elaborating on the implementation of the Random Forests algorithm and the use of the Encog, a neural network library for Java. We then proceed to use the supervised learning methods to build a procedure for producing a pixel labelling from a spectral image. We ignore until later the problems of how we will produce training data for the supervised learning methods, as these likely need to be extracted from an example labelling, a problem that will be addressed in section 3.4. Finally we also wait until nearer the end of the chapter to deal with image noise in section 3.5.

The project is modularised into a number of packages, each building on top of each other. Dependencies can be visualised in figure 3.1. We will use UML class diagrams[1] to show the overview of classes within each package.

[1]http://www.uml-diagrams.org/class-reference.html is a good reference for UML class diagrams

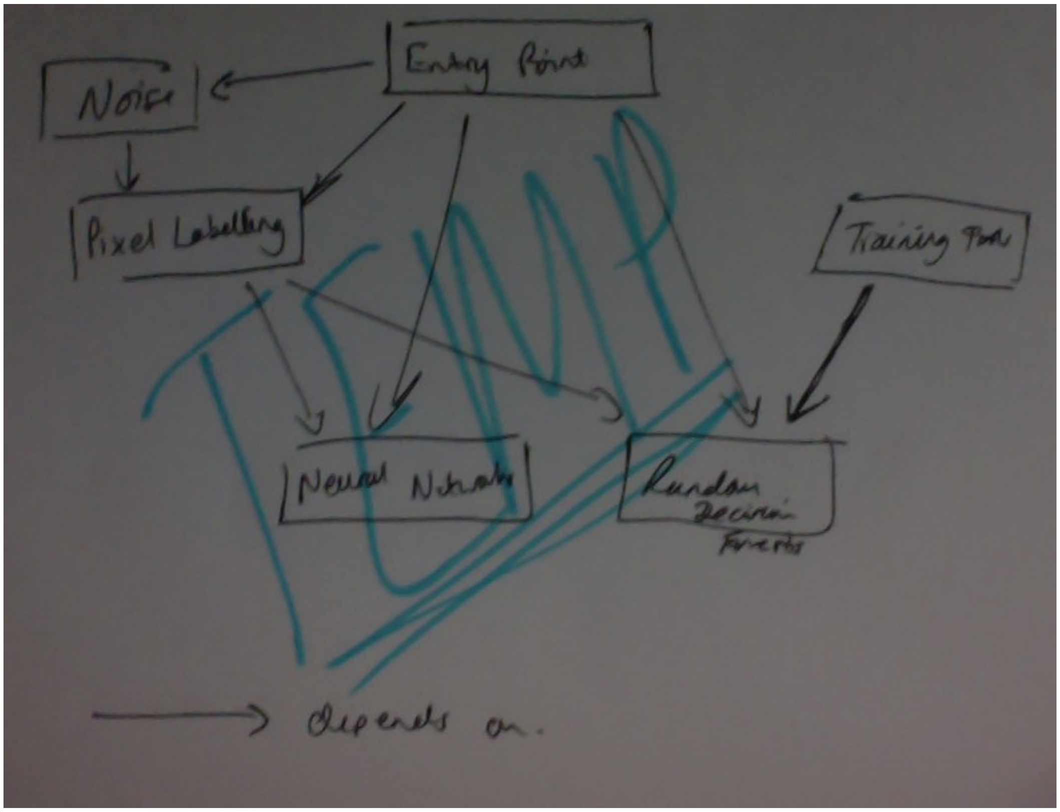

Figure 3.1: The dependencies between different packages/modules in the system. Although TrainingTools package is capable of producing training sequence files both the NeuralNetworks and RandomDecisionForests, it only *depends* on RandomDecisionForests as it uses the class `TrainingSequence`.

3.1 Random Forests Library

We begin by describing our implementation of random forests, looking specifically at some critical design choices that were made. We implement a generic and easily extensible random forests library, which can be specialised for many applications, then write specialising subclasses that utilise the library for our purpose of providing a pixel labelling.

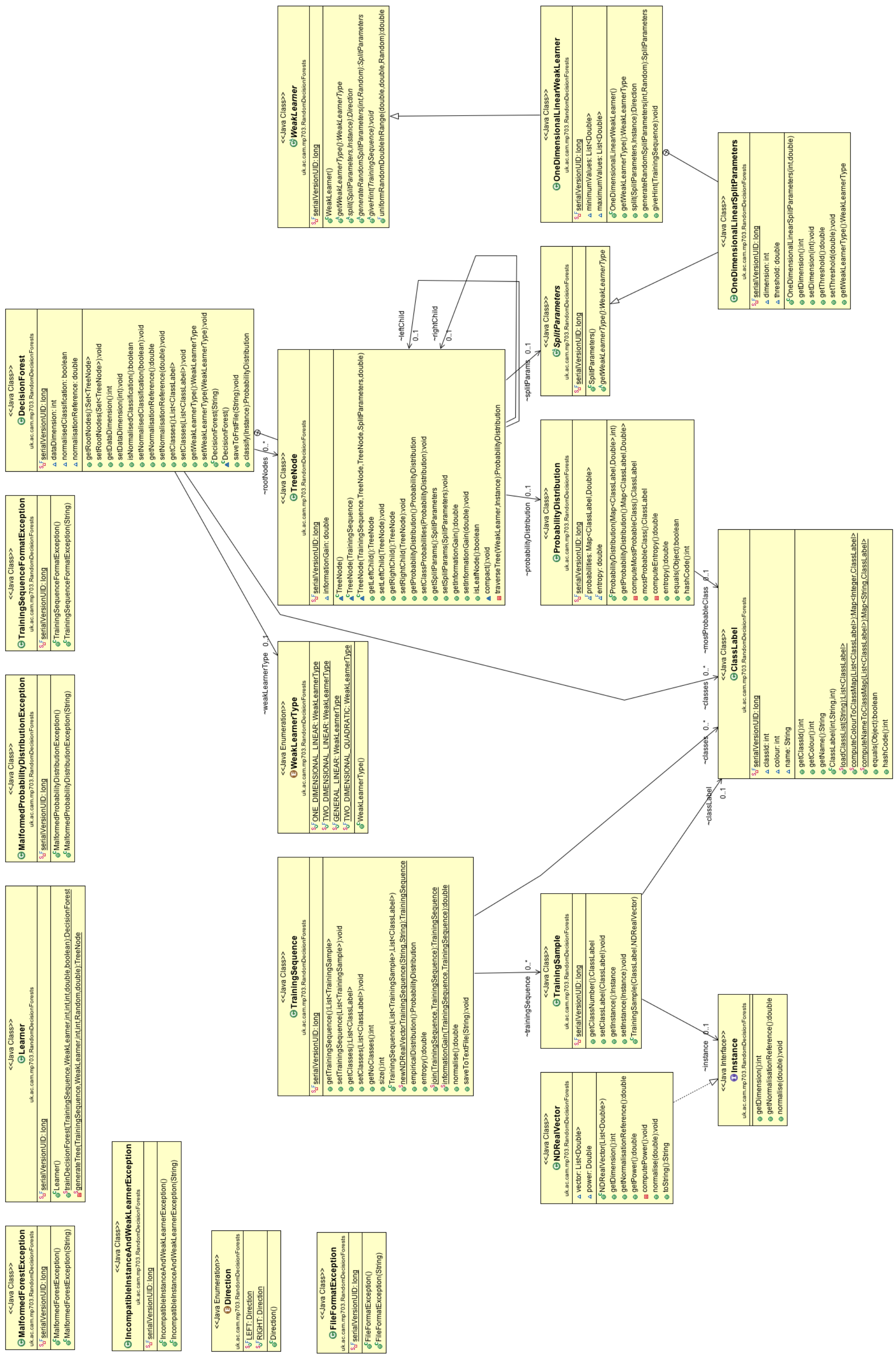

Figure 3.2: An overview of classes in the RandomDecisionForest package.

3.1.1 Data structures

A number of important data structures are used throughout the library, each requiring an efficient implementation and providing necessary abstraction and before we move onto the more complicated issues regarding supervised learning we need to describe the structures used. In this subsection we will describe the main classes that are used in the RandomDecisionForest package, providing a table of functions defined with corresponding English descriptions of what they do and some small code listings where necessary.

3.1.1.1 ClassLabels

Our first class, `ClassLabels`, abstracts classification labels into its own class. As can be seen in figure 3.3 `ClassLabels` groups together a class name and its corresponding colour, both of which should be unique, and we note that `classId` is only used internally. The colour will be used sections 3.3 and 3.4 to indicate its corresponding class in a pixel labelling, where the pixel labelling is either input in a "ground truth" image or output by our system.

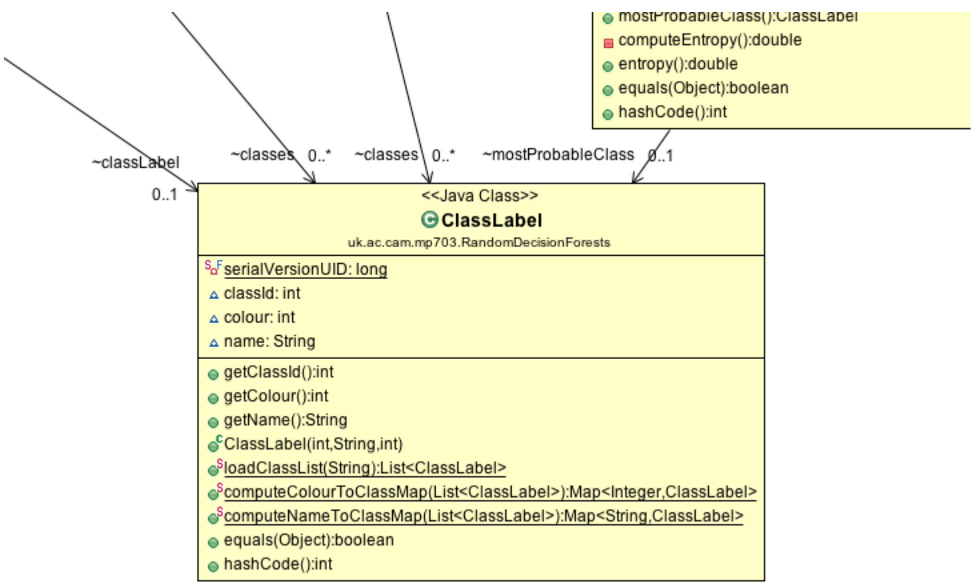

Figure 3.3: `ClassLabel` in figure 3.2 a bit closer.

Function Name	**Function Description**
`loadClassList`	A `static` function that returns a value of type `List<ClassLabel>` given a filename specifying "class colour map" as in appendix A, of classes specified in the file. In this function we make sure that we preserve the uniqueness of class names their associated colours.
`computeColourToClassMap`	Returns a value of type `Map<Integer, ClassLabel>`, a mapping from a class colours to their corresponding `ClassLabel` objects.
`computeNameToClassMap`	Returns a value of type `Map<String, ClassLabel>`, a mapping from a class names to their corresponding `ClassLabel` objects.
`equals`	An override of `Object`'s equality function. This is needed as we will frequently use `ClassLabel` as the key to a map structure, and might not use the same instance as a key. The function checks that the class name's, class colour's and class id's are identical.
`hashCode`	Similarly to `equals` this is overridden for a correct implementation when `ClassLabel` is used as the key in a (hash) map structure.

Table 3.1: Important methods implemented in the `ClassLabel` class.

3.1.1.2 Instances

The `Instance` class is used to specify a feature vector, or some instance of our problem. We define an interface to represent the base structure of an instance.

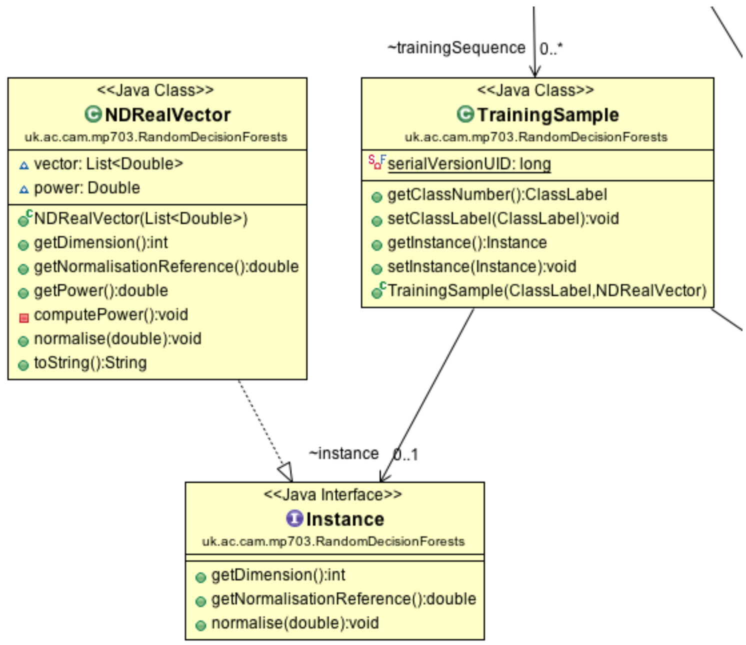

Figure 3.4: `Instance` in figure 3.2 a bit closer.

A feature vector will have some finite number of 'features' that we use, the `getDimension` function should return the number of features (i.e. the dimension) in the instance. The functions `getNormalisationReference` and `normalise` are used for normalisation of instances. During classification we may want to normalise so that small changes in data with low variance have the same weighting as larger changes in data with low variance. This is explored more in section 4.4.6. Intuitively we define some sort of 'power' value for each instance and use normalisation so that all `Instance`s have the same power. It is optional to use normalisation in the learning (and therefore classification) algorithms in sections 3.1.2 and 3.1.3.

We also define `NDRealVector`, a concrete implementation of `Instance` which can also be seen in figure 3.4. We use `NDRealVector` to represent spectra in spectral images, that is, values in $\mathbb{R}^N$. We can simply think of this as a wrapper for a list of values. We also store the power of the spectrum also, and use this as our "normalisation reference" value, and we define the *power* of a spectrum or vector to be sum of squared values in the list.

Function Name	**Function Description**
`getDimension`	This returns n if there are n values in `vector`, our list of doubles.
`getPower`	This returns the power of the spectrum of values. If we consider the list of values to be $\mathbf{v}$ then we return $\|\|\mathbf{v}\|\|^2$.
`getNormalisationReference`	This function returns the power of the spectrum.
`normalise`	Normalisation of vectors can be implemented by scalar division of the vector with the power. I.e. we divide all values in the spectrum by the power.
`toString`	We override `Object`'s `toString` method. This is used later in section 3.4 when we wish to print a training sequence to a text file. It simply returns a comma separated list of values.

Table 3.2: Important methods implemented in the `NDRealVector` class.

3.1.1.3 Probability Distributions

As discussed in section 2.3 probability distributions are associated with each node, and is the effective output from the classification algorithm. So any implementation of Random Forests needs to have some way to represent probability distributions over possible classifications. Although we can represent a probability distribution as `Map<ClassLabel, Double>`, we choose to encapsulate this in it's own class `ProbabilityDistribution` so that we can perform validation on the properties of a probability distribution, that is each probability is a value in $[0, 1]$ and the probabilities sum to a 1 and we also cache some frequently used values such as `entropy`.

The *entropy* of a probability distribution $p : \Omega \to [0, 1]$ is defined to be

$$H(p) = \sum_{x \in \Omega} -p(x) \log_2(p(x)) \tag{3.1}$$

which is consistent with the definition of entropy given in equation 2.7.

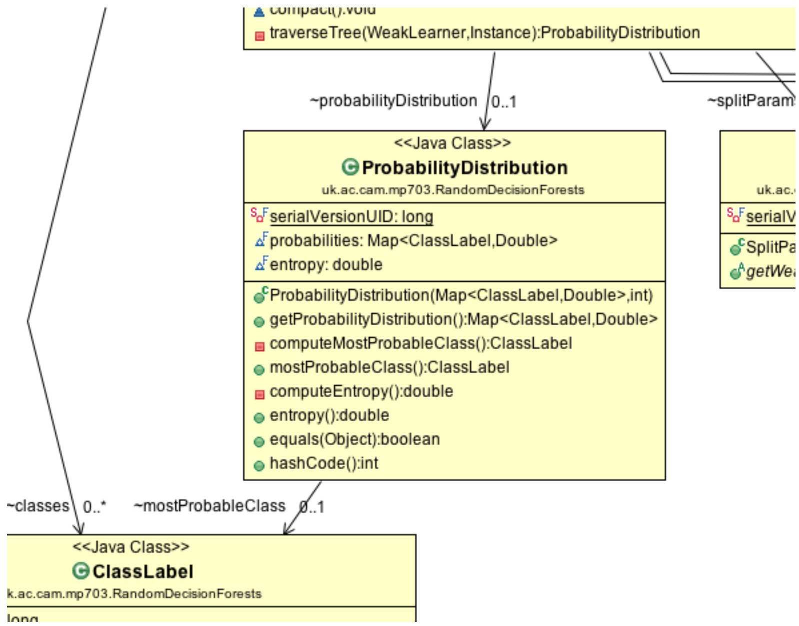

Figure 3.5: `ProbabilityDistribution` in figure 3.2 a bit closer.

We make the class immutable so that we cannot accidentally modify the distribution into something invalid. We make it immutable by making each variable `final` and removing any setter functions. We however also make use of the `unmodifiableMap` function in the `Java.Collections` package, when setting the `probabilities` variable. This gives a reference to a `Map` where no entries can be added, nor removed, which prevents someone from getting a reference to the map `probabilities` and altering it.

Also in the implementation of the class constructor we allow for a small error in the sum, this is because we will often have small errors from rounding in floating point numbers. So the checks that are made in the constructor of `ProaбabilityDistribution` are:

- each value is in the range $[0, 1]$;
- the sum of values is in the range $[1-\epsilon, 1+\epsilon]$, for some $0 < \epsilon \ll 1$.

Another implementation problem is found in the computation of entropy, as $-y\log_2(y)$ isn't defined for the value of $y = 0$. Mathematically we solve this problem by setting

$$-y\log_2(y)|_{y=0} \stackrel{\text{def}}{=} \lim_{y\to 0} -y\log_2(u) \tag{3.2}$$

$$= 0. \tag{3.3}$$

Now when we compute entropy according to equation 3.1 we try to sum with $p(x) = 0$ for some x. In this case we would accidentally set the sum to a `NaN` value

$$\begin{aligned} \texttt{sum} &= \texttt{sum} + 0.0 * \log(0.0) && (3.4)\\ &= \texttt{sum} + 0.0 * -\infty && (3.5)\\ &= \texttt{sum} + \texttt{NaN} && (3.6)\\ &= \texttt{NaN}, && (3.7) \end{aligned}$$

where ∞ is the floating point representation of infinity, and we have

$$\begin{aligned} \log(0.0) &= -\infty, && (3.8)\\ 0.0 * \infty &= \texttt{NaN}. && (3.9) \end{aligned}$$

in accordance with the IEEE 754 standard [1]. We hence need to skip over updating the accumulating `sum` variable when $p(x) = 0$ otherwise we will accidentally set `sum` to a `NaN` value.

```java
public class ProbabilityDistribution implements Serializable {

  ...

  public ProbabilityDistribution(Map<ClassLabel, Double> distribution, int noClasses)
      throws MalformedProbabilityDistributionException {

    ...

    // Check that the sum is 1, allowing for a small floating point error
    double eps = 1e-10;
    if (sum < 1.0-eps || 1.0+eps < sum) {
      throw new MalformedProbabilityDistributionException;
    }

    // Assign the values (so they are immutable)
    this.probabilities = Collections.unmodifiableMap(distribution);
    this.mostProbableClass = computeMostProbableClass();
    this.entropy = computeEntropy();
  }
```

```

    ...
}
```

Listing 3.1: Part of the `ProbabilityDirstribution` constructor, where we set $\epsilon = 2^{-10}$.

Function Name	**Function Description**
`computeMostPorbableClass`	Returns the `ClassLabel` with the highest probability in the distribution. This function is labelled private and is only used in the constructor.
`mostProbableClass`	Getter for the `mostProbableClass` value.
`computeEntropy`	Computes the entropy and returns the value. This function is labelled private and is only used in the constructor.
`entropy`	Getter for the `entropy` value.
`equals`	Override of `Object`'s function `equals`, returns true if the variable `probabilities` are equal in the two objects. We want this when comparing probability distributions rather than referential equality.
`hashCode`	Override of `Object`'s function `equals`, as we have overridden the `equals` function.

Table 3.3: Important methods implemented in the `ProbabilityDistribution` class.

3.1.1.4 Training Sequences

The `TrainingSample` object is simply a pairing between a `ClassLabel` and an `Instance`. There is no other functions in this class other than the getters, setters and constructor.

A `TrainingSequence` consists of a list of `TrainingSamples`, and we keep a list of `ClassLabels` for reference. A number of convenience functions are defined in the class and are listed in table 3.4, which are used to compute useful values, such as entropy and information gain from equations 2.7 and 2.11 respectively.

Figure 3.6: `TrainingSequence` and `TrainingSample` in figure 3.2 a bit closer.

Function Name	**Function Description**
`newNDRealVector-TrainingSequence`	`newNDRealVectorTrainingSequence` is a `static` function taking two file names. One specifying a "class colour map" file and one specifying a training sequence. Both these files should be of the format described in appendix A. This function parses the files and returns a `TrainingSequence` instance generated from it, with instances of the type `NDRealVector` defined in listing ??.
`empericalDistribution`	Returns a `ProbabilityDistribution` that represents the empirical distribution of the training sequence, as defined in equation 2.8.
`entropy`	This returns the entropy of the empirical distribution. It is essentially just a shorthand for calling `empiricalDistribution().entropy()`.
`join`	A `static` function that takes two `TrainingSequences`, and joins them together. This simply appends the lists of `TrainingSamples` from two `TrainingSequences` into a new `TrainingSequence` instance.
`informationGain`	A `static` function that takes two `TrainingSequences` `ts1` and `ts2`. The function returns the information gain for splitting the training sequence `join(ts1,ts2)` into the training sequences `ts1` and `ts2` according to equation 2.11.
`normalise`	This parses all of the `Instances` in the training sequence and computes an average "normalisation reference" (see `getNormalisationReference` in listing ?? and in table 3.2). It then returns a new training sequence where every sample has been normalised to this average value.
`saveToTextFile`	This `static` function writes out a training sequence file with the data from the given `TrainingSequence` instance, in accordance with the file format specified in appendix A.

Table 3.4: Member functions of the `TrainingSequence` class.

3.1.1.5 Split Parameters

We define an abstract class `SplitParameters`, which is used as a common superclass for any implementation of a split parameter, as defined in section ??. We make this an abstract class so that can implement `Serializable` forcing any implementing classes to also implement this interface, which is necessary to be able to save decision forests to a file in section 3.1.1.8. We also specify that each split parameter needs to be able to specify what weak learner it is for, because we only save the split parameters in decision trees and we need some way to determine which `WeakLearner` subclass to use from a `SplitParameter` instance.

We consider also a concrete implementation of `SplitParameters`, specifically `OneDimensionalLinearSplitParameters` which specifies a "dimension" and a threshold as needed for the `OneDimensionalLinearWeakLearner` in section 3.1.1.6.

The `SplitParameter` classes are simple and only include getters, setters and constructors. That is there are no 'interesting' functions in these classes.

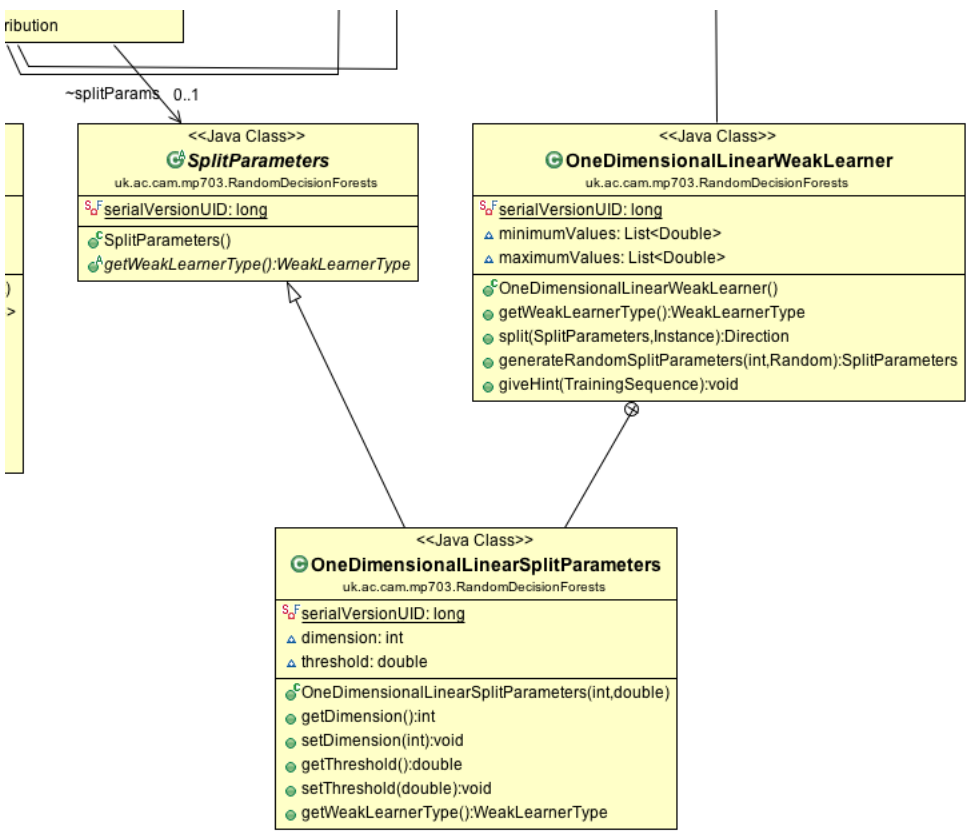

Figure 3.7: `SplitParamter` and `OneDimensionalLinearSplitParameter` in figure 3.2 a bit closer.

3.1.1.6 Weak Learners

We implement `WeakLearner` as an abstract class, because we provide a partial implementation. Any common code between subclasses can be put directly into the `WeakLearner` abstract class, such as helper functions for generation of randomised parameters. The function of each abstract method is explained in table 3.5. The function `uniformRandomDoubleInRange` is a simple convenience method that is used to generate a value uniformly in some range `[lowerBound,highBound]`, and uses Java's implementation of generating a uniform random variable in the range `[0,1]`. To identify the type of each weak learner with some tag we define an `enum` type in listing ??, seen in figure ??.

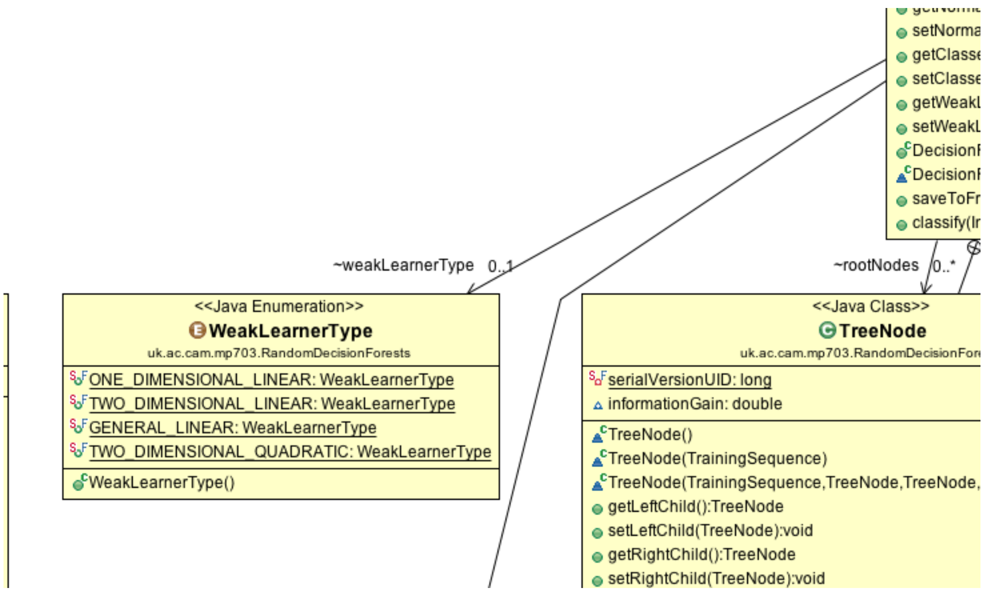

Figure 3.8: The `WeakLearnerType` enum in figure 3.2 a bit closer.

We also consider a concrete implementation for an example, using the `OneDimensionalLinearWeakLearner` class. This is the most simple weak learner function that we can decide and has the simplest parameters, when we are considering `NDRealVector` instances. We note that subclasses of `WeakLearners` are specific to particular subclasses of `Instance`. The `OneDimensionalLinearWeakLearner` simply picks one value in the feature vector/instance, compares it to some threshold and then makes a decision based on this. For example consider (mathematically) an instance $\mathbf{v} \in \mathbb{R}^n$, in this case our split parameters are $\theta = (i, \tau) \in \mathbb{Z}_n \times \mathbb{R}$, and the split function/weak learner is

$$f(\mathbf{v};\theta) = f(\mathbf{v};(i,\tau)) = \begin{cases} 1 & \text{if } v_i \geq \tau \\ 0 & \text{otherwise.} \end{cases} \tag{3.10}$$

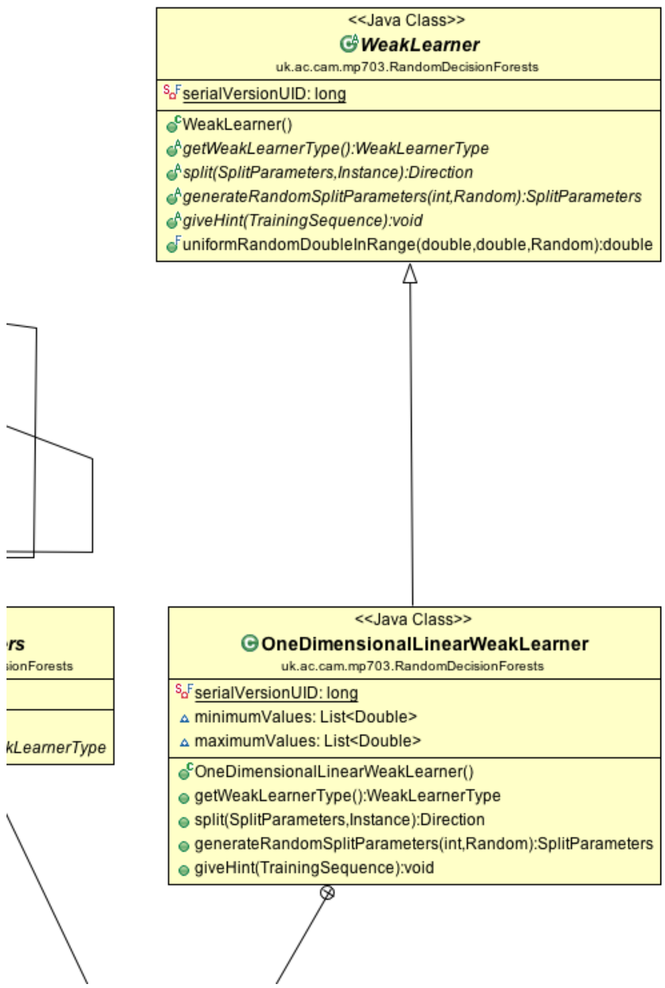

Figure 3.9: `WeakLearner` and `OneDimensionalLinearWeakLearner` in figure 3.2 a bit closer.

These split parameters for the `OneDimensionalLinearWeakLearner` class are represented by the `OneDimensionalLinearSplitParameters` class as described in section 3.1.1.5. This is an example that each subclass of `WeakLearner` should have a corresponding subclass of `SplitParameters` which it uses to represent it's split parameters. And we note that `OneDimensionalLinearSplitParameters` is implemented as a nested class within `OneDimensionalLinearWeakLearner`, to make this relationship obvious.

The `split` function is of particular interest in listing **??** as it implements equation 3.10.

Function Name	**Function Description**
`getWeakLearnerType`	A function that is used to identify what type of weak learner an instance is. This is useful when a subclass is cast to `WeakLearner` and we want to cast it back to the subclass without causing an exception.
`split`	This function takes a split parameter and an instance, it is the concrete implementation of the split function. It returns an enum of type `Direction`, which specifies if we should traverse to the left or right child of a decision tree node.
`giveHint`	In this function we are passed the training sequence prior to training a decision tree. It is used as a chance to look at the data to give a hint to what split parameters we might want to generate, or more specifically not generate.
`generateRandomSplitParameters`	This function provides a routine for generating a random split parameter to be tried in the split function during training a decision tree.

Table 3.5: Member functions of the `OneDimensionalLinearWeakLearner` class.

The `OneDimensionalLinearWeakLearner` makes use of the `giveHint` function by iterating through all `NDRealVector` instances in the training sequence and recording a minimum and maximum value for each dimension in the lists `minimumValues` and `maximumValues`. This therefore allows us to avoid picking particularly bad `OneDimensionalLinearSplitParameters`, as for any of the features/dimensions in the `NDRealVector` that are chosen, if we pick a threshold value outside the range between the minimum and maximum we know that the split function will split the training sequence with zero information gain. Using the notation from equations 2.9 and 2.10 it means that one of $L(\mathbf{s}, \theta)$ or $R(\mathbf{s}, \theta)$ will be empty, leading to an information gain of zero ($I(\mathbf{s}, \theta) = 0$ in equation 2.11).

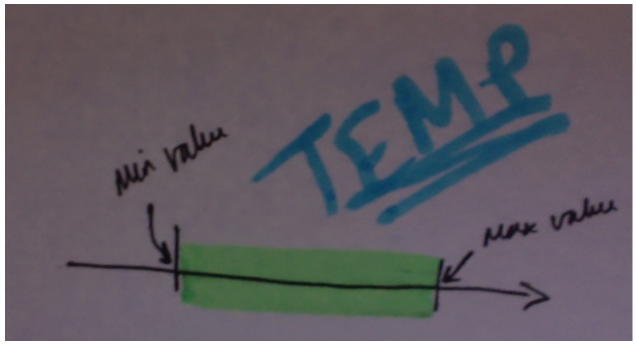

Figure 3.10: Knowing the minimum and maximum values in each dimension allows us to make an informed choice on split parameters to pick, which in this case is the green zone of values.

3.1.1.7 Decision Tree Node

Finally we move onto defining our decision tree, implement the `TreeNode` class nested in `DecisionForest` to show that their behaviour is tightly coupled. Each node has a reference to a left and right child node, which are set to `null` in a leaf node. Every node caches a probability distribution, even if it is a decision node, and the probability distribution that is cached is the empirical distribution of the training sequence that is passed to it. Each decision node has a reference to a `SplitParameters` instance, which it uses with a weak learner to make decisions for tree traversal as in section 2.3.1. Finally we cache the information gain, computed according to the equation 2.11 during the training algorithm.

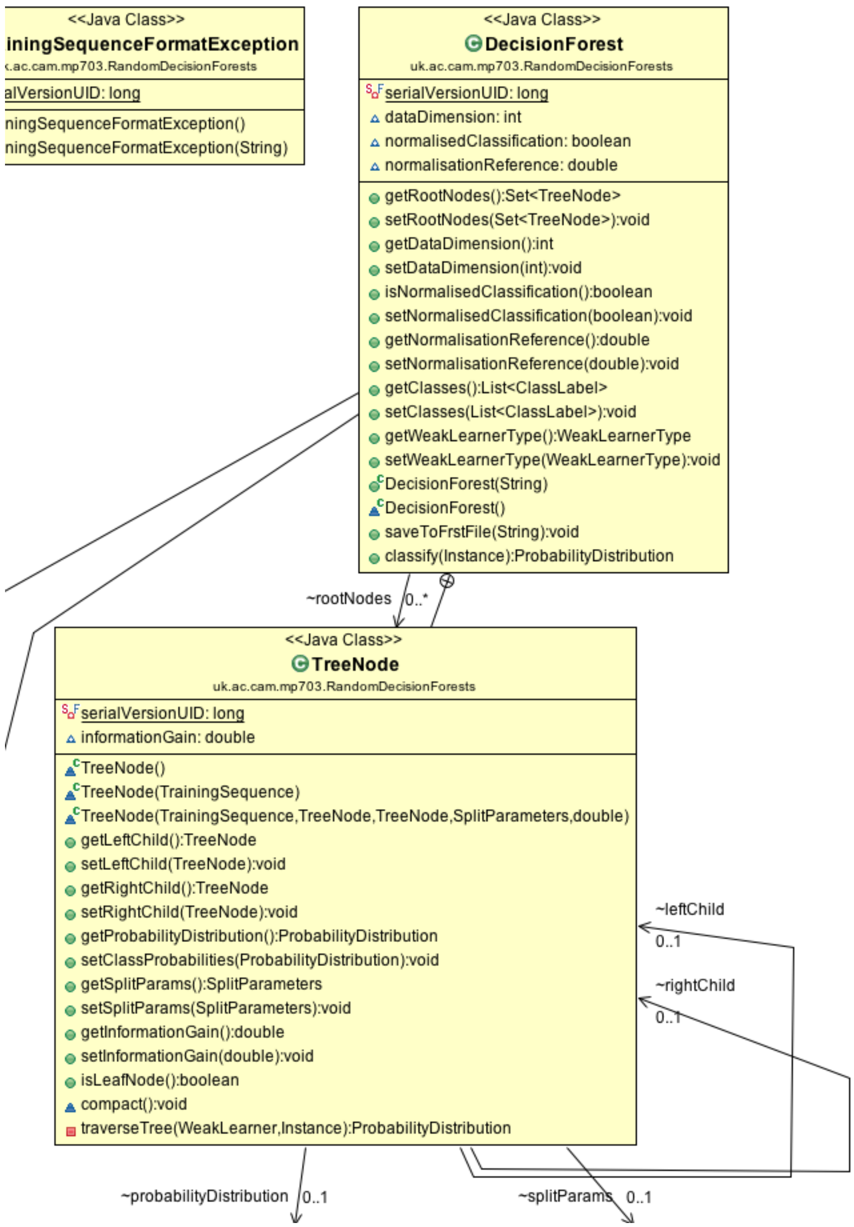

Figure 3.11: `TreeNode` and `DecisionForest` in figure 3.2 a bit closer.

We see in listing 3.2 the function `compact`. This is an optimisation used to eliminate unnecessary nodes from the tree after its construction. The idea is that if we are at some decision node and it has two children are leaf nodes with identical distributions, then we can replace this decision node by a single leaf node with the same distribution. The implementation simply performs `compact` recursively on the left and right children of a decision node first, and `compact` does nothing in the base case (a leaf node), after we try to compact at the given node. Essentially, compaction is performed from the bottom of the tree upwards.

We also note that the `TreeNode` class implements the `Serializable` interface, which is again necessary to be able to use `ObjectOutputStream` in section 3.1.1.8, when saving forests to a file.

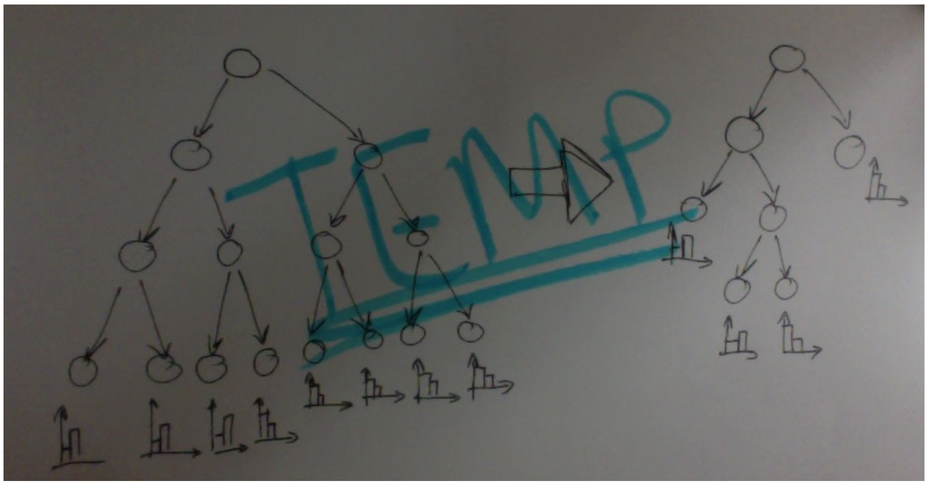

Figure 3.12: An example of how `compact` could be used to reduce the size of a decision tree.

```java
void compact() throws MalformedForestException {
  if (this.isLeafNode()) {
    return;
  }

  this.leftChild.compact();
  this.rightChild.compact();

  if (!this.isLeafNode() &&
      leftChild.isLeafNode() &&
      rightChild.isLeafNode() &&
      probabilityDistribution.equals(leftChild.probabilityDistribution) &&
      probabilityDistribution.equals(rightChild.probabilityDistribution)) {

    // Make this node a leaf node
    leftChild = null;
    rightChild = null;
  }
}
```

Listing 3.2: The `TreeNode` declaration, found as a static class within the `DecisionForest` class.

Function Name	Function Description
`isLeafNode`	Checks if the `leftChild` and `rightChild` have a value of `null`. If they are then the node is a leaf and this functions returns a true value. If only one of the nodes has a value of `null` then the tree is invalid and an exception is raised. Otherwise it returns a false value.
`compact`	As described above, this compacts the tree to an equivalent tree, replacing any unnecessary decision nodes with a leaf node.
`traverseTree`	A method that given an instance of a `WeakLearner` and an `Instance` will traverse the tree and return a probability distribution for that instance. As described in section 2.3.1 and visualised in figure 2.3. The algorithm this method implements is covered in more detail in section **??**.

Table 3.6: Member functions of the `TreeNode` class.

3.1.1.8 Decision Forest

We finally define our `DecisionForest` class. It is basically a set of `TreeNode`s, however we also include additional metadata, which can be seen in figure 3.11. We have made sure that `TreeNode`, `WeakLearner`, `WeakLearnerType`, `SplitParameters` and `ClassLabel` (which are all composite in the forest data structure) implement the interface `Serializable` so that a concise "save to file" method can be implemented. We simply use an instance of `ObjectOutputStream` in the `Java.Io` package to write the object to a binary file in a single line "`out.writeObject(this);`".

`dataDimension` keeps track of the dimension of data this forest classifies, the `weakLearnerType` specifies what weak learner our forest is related to, and implicitly the type of data that it classifies. It keeps a reference to all the possible classification labels in `classes` and it uses `normalisedClassification` and `normalisationReference` to perform normalisation during classification if we specified that normalised values should be used during training.

Function Name	Function Description
`DecisionForest`	We include the constructor here because we provide two. We have a `default` access default constructor, which is used to construct forests in the training algorithm. The other allows the forest to be loaded from a file using an `ObjectInputStream` from the `Java.Io` package.
`saveToFrstFile`	This function saves a forest to a persistent ".frst" file. It's implementation can be found in listing ??.
`classify`	This takes an `Instance` and returns a probability distribution for it, according to the forest. The algorithm implements is discussed in more detail in section 3.1.3.

Table 3.7: Member functions of the `TreeNode` class.

3.1.2 Training

- Walk through algorithm, with a listing(s)
- Discuss information gain cutoff
- Discus tradeoffs between breadth and depth first training
- Abstraction of split parameter generation into the weak learner (etc.) allows for a SINGLE learning routine to be implemented for all Weak-Learners.
- (Still to implement). Parallellisation of training trees. Discuss that much more could be done.
- (Still to implement). Bagging to prevent bias & overfitting

3.1.3 Classification

todo

3.2 Neural Networks

- Introduce encog
- Explain usage of the library and features we used

3.2.1 Training

- Outline how we wrapped up the training of the neural network
- Describe roughly what it does?

3.2.2 Classification

- Outline how we wrapped up the classifier
- Describe roughly what it does?

3.3 Pixel Labelling

- We've given the overview of two supervised machine learning techniques
- Assume for now that we have appropriate training data - dealt with later
- We now need to use that to provide a pixel labelling
- Listing for the datacube, and listing that implements the pixel labelling
- Assuming we have appropriate training data, and non-noisy images we are now done! Unfortunately not the case.

3.4 Training Tools

- Take an image with a manually created ground truth pixel labelling
- Take these and convert into a training sequence, and output to a text file
- The append option -¿ we can pass in a perameter and an existing training sequence and append training sequences

3.5 De-noising

- Explain total variation
- Minimising gradients, can think of this as removing high frequency components
- Listing for overview of

3.6 Application on example data sets

- We've build a system, but we need to actually apply it to some problems!

3.6.1 Siri's data

- TODO: Come up with a better title for this subsection
- Explanation of how produced training data
- Example pixel labellings

3.6.2 Teng's data

- TODO: Come up with a better title for this subsection
- Explanation of how produced training data
- Example pixel labellings

Chapter 4

Evaluation

In this section we will look at the performance of the different components of the system based on a number of metrics.

4.1 Performance measures for classifiers

- Briefly say about the different measures as described at the end of AI II.
- Justify what might be the most informative here.

4.2 Evaluation of Random Forests pixel labelling

- Compute the performance measures for a noiseless toy image, noisy toy image and two data sets provided, and compare

4.3 Evaluation of Neural Networks pixel labelling

- Compute the performance measures for a noiseless toy image, noisy toy image and two data sets provided, and compare
- Compare with random forests - success criterion (table of results and say a bit about which did better)

4.4 Evaluation of the Random Forests library

- Define a standard training set, this can actually be independent of pixel labelling and similar to the Chriminsi forests paper.

4.4.1 Training time

4.4.2 Classification time

4.4.3 The effect of the number of trees

4.4.4 The effect of the depth of trees

4.4.5 The effect of the randomness of trees

4.4.6 The effect of normalisation

4.5 The effect of the de-noising component

- Use a couple images with added noise
- Compare the SNR for the method with respect to different noise models (does it handle any other types of noise?)
- Plot SNR as a function of the parameter lambda in the TVMM method
- Plot SNR as a function of noise power (for different types of nois)

Chapter 5

Conclusion

5.1 Summary

- Overview and summary of work undertaken - re describe system overall
- What was achieved, what was different
- What did the evaluation show?

5.2 Further Work

- Implement a convolutional neural network solution to allow local spatial data to influence the labelling of a pixel. This may allow the model to incorporate image de-noising and could compact two stages of the pipeline into one.
- Implementation of the random forests library to support GPU processing. We could use a language such as OpenCL. Parallelism could be exploited either in the random forest implementation OR/AND the inherent parallelism from performing image processing.

Bibliography

[1] *IEEE standard for binary floating-point arithmetic.* Institute of Electrical and Electronics Engineers, New York, 1985. Note: Standard 754–1985.

[2] Vijay Badrinarayanan, Ankur Handa, and Roberto Cipolla. Segnet: A deep convolutional encoder-decoder architecture for robust semantic pixel-wise labelling. *arXiv preprint arXiv:1505.07293*, 2015.

[3] José M. Bioucas-Dias, Antonio Plaza, Nicolas Dobigeon, Mario Parente, Qian Du, Paul Gader, and Jocelyn Chanussot. Hyperspectal unmixing overview: Geometrical, statistical, and sparse regression-based approaches. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, 5, 2012.

[4] Sailesh Conjeti, Amin Katouzian, Abhijit Guha Roy, Loïc Peter, Debdoot Sheet, Stéphane Carlier, Andrew Laine, and Nassir Navab. Supervised domain adaptation of decision forests: Transfer of models trained in vitro for in vivo intravascular ultrasound tissue characterization. *Medical Image Analysis*, 32:1–17, 2016.

[5] Pierrick Coupé, Pierre Hellier, Charles Kervrann, and Christian Barillot. Nonlocal means-based speckle filtering for ultrasound images. *Image Processing, IEEE Transactions on*, 18(10):2221–2229, 2009.

[6] Antonio Criminisi and Jamie Shotton. *Decision forests for computer vision and medical image analysis.* Springer Science & Business Media, 2013.

[7] Antonio Criminisi, Jamie Shotton, and Ender Konukoglu. Decision forests: A unified framework for classification, regression, density estimation, manifold learning and semi-supervised learning. *Foundations and Trends® in Computer Graphics and Vision*, 7(2–3):81–227, 2012.

[8] Padraig Cunningham and Sarah Jane Delany. k-nearest neighbour classifiers. *Multiple Classifier Systems*, pages 1–17, 2007.

[9] Mario AT Figueiredo, J Bioucas Dias, Joao P Oliveira, and Robert D Nowak. On total variation denoising: A new majorization-minimization algorithm and an experimental comparisonwith wavalet denoising. In *Image Processing, 2006 IEEE International Conference on*, pages 2633–2636. IEEE, 2006.

[10] AE Gamal and H Eltoukhy. Cmos image sensors, an introduction to the technology, design, and performance limits, presenting recent development and future directions. *IEEE circuits and devices mag*, pages 8755–3996, 2005.

[11] Samuel W Hasinoff. Photon, poisson noise. In *Computer Vision*, pages 608–610. Springer, 2014.

[12] Jeff Heaton. *Introduction to neural networks with Java*. Heaton Research, Inc., 2008.

[13] Jeff Heaton. Encog: Library of interchangeable machine learning models for java and c#. *Journal of Machine Learning Research*, 16:1243–1247, 2015.

[14] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[15] Nirmal Keshava. A survey of spectral unmixing algorithms. *Lincoln Laboratory Journal*, 14(1):55–78, 2003.

[16] Dan Klein. Lagrange multipliers without permanent scarring. *University of California at Berkeley, Computer Science Division*, 2004.

[17] Martin Law. A simple introduction to support vector machines. *Lecture for CSE*, 802, 2006.

[18] Qingli Li, Xiaofu He, Yiting Wang, Hongying Liu, Dongrong Xu, and Fangmin Guo. Review of spectral imaging technology in biomedical engineering: achievements and challenges. *Journal of Biomedical Optics*, 18, 2013.

[19] A. Siri Luthman and Sarah E. Bohndiek. Experimental evaluation of a hyperspectral imager for near-infrared fluorescent contrast agent studies. Technical report, University of Cambridge, 2015.

[20] Nikhil R Pal and Sankar K Pal. A review on image segmentation techniques. *Pattern Recognition*, 26(9):1277 – 1294, 1993.

[21] Dzung L Pham, Chenyang Xu, and Jerry L Prince. Current methods in medical image segmentation 1. *Annual review of biomedical engineering*, 2(1):315–337, 2000.

[22] Eugenio Picano. Sustainability of medical imaging: doctors and patients should be more aware of the long term risks of radiological investigations. *British Medical Journal*, 328(7439):578–581, 2004.

[23] Isabel Rodrigues, Joao Sanches, and Jose Bioucas-Dias. Denoising of medical images corrupted by poisson noise. In *Image Processing, 2008. ICIP 2008. 15th IEEE International Conference on*, pages 1756–1759. IEEE, 2008.

[24] Sheldon M Ross. *Probability models for computer science*. Harcourt Academic Press San Diego, 2002.

[25] Stuart Russell, Peter Norvig, and Artificial Intelligence. A modern approach. *Artificial Intelligence. Prentice-Hall, Egnlewood Cliffs*, 25:27, 1995.

[26] Kristie Seymore, Andrew McCallum, and Roni Rosenfeld. Learning hidden markov model structure for information extraction. In *AAAI-99 Workshop on Machine Learning for Information Extraction*, pages 37–42, 1999.

[27] Perry Sprawls. *Physical principles of medical imaging*. Aspen Publishers, 1987.

[28] Guangjun Zhao, Xuchu Wang, Yanmin Niu, Liwen Tan, and Shao-Xiang Zhang. Segmenting brain tissues from chinese visible human dataset by deep-learned features with stacked autoencoder. *BioMed Research International*, 2016, 2016.

[29] Liya Zhao and Kebin Jia. Multiscale cnns for brain tumor segmentation and diagnosis. *Computational and Mathematical Methods in Medicine*, 2016, 2016.

Appendix A

File formats

Here we define the file formats that a user might be expected to input into the system, an explanation of the files that are output by the system and how to interpret them.

A.1 (Example/Noisy) Spectral Image

...

A.2 Example Image Labelling

...

A.3 Label Map

```
typewriter .txt file eg here
```

A.4 Training Sequence

```
typewriter .txt file eg here
```

A.5 Output Files

...

Appendix B

A brief explanation of the method of conjugate gradients

TODO

Appendix C

Project Proposal

Computer Science Project Proposal

Spectral Image Analysis for Medical Imaging

M. Painter, Churchill College

Originator: Dr. Pietro Lio'

Project Supervisor: Dr Pietro Lio', Dr Gianluca Ascolani

Director of Studies: Dr John Fawcett

Project Overseers: Prof John Daugman & Dr David Greaves

Introduction and Description of the Work

The core idea of the project is to use spectral algorithms and machine learning to analyse biomedical images. I will explore the use of multiple learning techniques (namely Random Decision Forests and Neural Networks) and compare them via a number of metrics, described later.

The aim of the project will be to build a classifier that is capable of handling a wide variety of noisy medical images. Different types of medical images will present different challenges such as contrast between tissues and amount and type of noise. It will classify images per pixel into classes (dependent on the image), using the spectral analysis.

During the implementation I will start by implementing something for a toy problem, by which I mean an artificially produced, noiseless image. From this solution I will move onto solving the same problem with the introduction of artificial noise, at which point I will explore the use of de-noising techniques to improve the accuracy of the classifier. Finally after this I will move onto an implementation for real images.

For the real images I will use MRI images from Zhongzhao Teng from the Department of Radiology, Engineering, which include fatty tissues from patients with atherosclerosis (the build up of fatty tissues in arteries). In this case I can train the classifier to recognise regions of calcium, lipids, haemorragic tissue, and mixtures of these.

To demonstrate the versatility of the tool I will also use another set of images obtained in a completely different way, so will present a different set of challenges including the nature of noise in the image. The images are from Siri Luthman of the BSS group in the Department of Physics, and are obtained using a hyper-spectral camera with 72 spectral bins. The images use contrast agents with various spectral profiles, each of which has a negative binding response or neutral binding response for cancerous cells. By 'negative binding response' I mean that the contrast agent binds to only non-cancerous cells, and 'neutral' means that it binds to both cancerous and non-cancerous cells. Here the classes will be cancer cells, non-cancerous cell, and a mixture of both.

Starting Point

I will use Java for the implementation of my algorithms and use libraries provided for Java. I will also use OpenCV's machine learning library as a benchmark to test my implementations against.

Resources Required

I will use my own computer to code on as I am more comfortable with it than the MCS machines.

I will also need example (spectral) image sets to be used in training and testing. I have kindly been provided with some MR images by Zhongzhao Teng of the Department of Ragiology and some hyperspectral images from Siri Luthman of the BSS group in the Department of Physics, as mentioned in the introduction.

Backup Plan

I will be using a git repository for my project, which will be a private repository on GitHub. I will have two branches (at least, more if necessary) one 'master' for completed features and one 'in progress'. The in progress branch will be to push regularly any unfinished/in progress work onto so that my work is always backed up, and I don't loose any work in progress.

The local folder will also be in my Google Drive folder and so will be automatically synced to the Google servers, and I also have a time machine set up, which will take backups of the whole file system every hour.

Work to be done

I can break down the project into the following stages:

1. Implement the infrastructure (data structures etc) and reading in of the raw data.

2. Create a tool used to indicate areas of interest on the teaching data to be used with the training data.

3. Implement the main machine learning algorithm(s), and implement an out the box solution, initially to solve the 'toy problem'.

4. Extend the implementation to work for noisy images and then real images.

Success Criterion for the Main Result

I will use an 'out the box' solution (such as OpenCV's machine learning libraries) as a benchmark to compare my implementation(s) against. I will provide each machine learning algorithm with the same inputs for training and the same inputs for testing and will use the following metrics to compare the performance of the systems and which has 'learnt' better:

- Overall run time (of classifying a single image);
- Accuracy of classifier (percentage images correctly identified).

Possible Extensions

- *Image Segmentation.*

  It would be useful to be able to segment the images given into separate regions of interest. (For example if we want to classify an image with cancer cells present, we indicate the regions containing cancerous cells). This is opposed to just returning the output per pixel, and would provide a more useful output for users.

- *Dimensionality Reduction.*

  If I finish the core of my project then one of the additional tasks that I could look at would be to implement a data reduction (learning) algorithm. In a spectral image with a high number of spectral bands it would be useful to identify which spectral bands are important to the classification and which are not.

  This problem more generally is called "dimensionality reduction" as we are looking to reduce the dimension of data input to the algorithm.

Timetable: Workplan and Milestones to be achieved.

Michaelmas weeks 3–4 (26th Oct to 4th Nov)

In preparation for the main implementation I will read about many machine learning algorithms. For example I will use the book "Decision Forests for Computer Vision and Medical Image Analysis" to learn about Random Forests. I will similarly research about Neural networks, and I will also research some other forms of learning algorithm.

Deliverables:

- A small description/overview of random forests and how they work, and a similar description for any other learning algorithms researched. This should be written up in LaTeX for easy embedding into the preparation chapter of the dissertation.

Milestones:

- Preparation reading completed, so that I am sufficiently able to implement the learning algorithm(s) and handed in description of reading to project supervisor for review - 4th Nov.

Michaelmas weeks 5–6 (5th Nov to 18th Nov)

I will spend this block familiarising myself with any technologies that I will possibly use. This will include the OpenCV library. I will also design a simple UI that will be used for the supervised learning. I will also familiarise myself with OpenCL and GPU programming.

Deliverables:

- A sketch of the UI for the tool which will be used for the supervised learning.

- A small overview of what OpenCV, GPU programming (or other technology(s) I have looked at), and a description of why/where they will be useful. This should again be written up in LaTeX for easy embedding into the preparation chapter of the dissertation.

Milestones:

- Handed in the UI design to supervisor for review - 9th Nov

- Handed in the description of familiarisation of technologies to supervisor for review - 18th Nov.
- Demonstrated any small programs written for familiarisation to supervisor - 18th Nov.

Michaelmas weeks 7–8 (19th Nov to 2nd Dec)

Implementation block 1. Implement the infrastructure that will be used for the project. This will include loading the raw data into the appropriate data structures and do so efficiently as possible.

Deliverables:

- A bullet point list indicating what has been implemented during this block. Written up in LaTeX and to be used as a basis for the implementation portion of the dissertation. To be handed in for review.

Milestones:

- Have the framework that I will use completed, including unit tests, and demonstrate the tests to supervisor to check that this has been done - 2nd Dec.

Christmas vacation weeks 1–2 (3rd Dec - 16th Dec)

I will be on holiday during this period and so I will work on little bits when I can, this may be catching up on any work that I got behind on in Michaelmas term (effectively making this a 'slack' block), and I will begin some work on the next block if up to date.

Christmas vacation weeks 3–4 (17th Dec - 30th Dec)

Implementation block 2. Write the tool that will be used to indicate which images (or image regions) belong to a given class to be used for the supervised learning.

Deliverables:

- A bullet point list indicating what has been implemented during this block. Written up in LaTeX and to be used as a basis for the implementation portion of the dissertation.

Milestones:

- Sent the completed tool to my supervisor as proof of completion. (As it is the vacation I will not be in Cambridge and so will not be able to demonstrate in person). - 30th Dec

Christmas vacation weeks 5–6 (31st Dec - 13th Jan)

Implementation block 3. Implement the machine learning algorithm(s) chosen, and also write the 'out of the box' solution using the OpenCV library. At this stage I will only aim to have

If this block is delivered on time then I am close to having a completed project. I have purposely stacked more of the work in the holiday compared to term time, as I have other commitments in term than out of term, such as lectures and supervisions.

Deliverables:

- A bullet point list indicating what has been implemented during this block. Written up in LaTeX and to be used as a basis for the implementation portion of the dissertation.

Milestones:

- Finished writing the machine learning algorithm, and demonstrated it classifying some 'toy images' to supervisor - 13th Jan. (I will be back in Cambridge by this time).

Lent weeks 1–2 (14th Jan - 27th Jan)

The progress report needs to be given up by noon on the 29th Jan, and so should be written in this block.

This rest of this block will be kept free for 'slack'. This slack should include incorporating any feedback from my supervisor.

If I am up to date and there is no additional feedback that needs to be resolved, then I will proceed to work a block ahead. This will give me an additional block at the end of lent to implement any extensions.

Deliverables:

- Progress report.
- Item written in LaTeX which begins with the supervisors comments/feedback and ends with how that was incorporated into the system.

Milestones:

- Demonstration of any features added made to supervisor - 27th Jan

Lent weeks 3–4 (28th Jan - 10th Feb)

Implementation block 4. In this block I will finish implementing my solution, and extend my simple solution to a more complex one capable of handling 'toy images' with artificial noise, and then real images. This will require implementing de-noising of the images.

Also after the progress report a small presentation needs to be prepared. I will use the beginning of this time block to do this.

Deliverables:

- A bullet point list indicating what has been implemented during this block. Written up in LaTeX and to be used as a basis for the implementation portion of the dissertation.

Milestones:

- Hand in progress report (completed in the previous block) - 29th Jan.
- Perform a small mock presentation/interview with supervisor - 3rd Feb.
- Made progress presentation to overseer group - 5th Feb.
- Finished implementing any de-noising algorithms, and have a working solution for real images, including a demonstration to supervisor - 10th Feb.

Lent weeks 5–6 (11th Feb - 24th Feb)

Evaluation of the system. This will require writing tests that will gather the quantitative data (run time and accuracy of classification) to be used in the comparison required to satisfy the success criteria.

I will generate graphs from data in this phase using MATLAB, so that I can decide if the data is useful or not. This will prevent me from realising that the data isn't good at too late a stage in the project and when there is not enough time to re-evaluate the system. This has the advantage that the graphs will be ready to be exported directly into the dissertation.

Deliverables:

- Table for any qualitative data evaluated and spreadsheet of quantitative data, which must include values for accuracy of the system (in a specific test case) and timings for how long classification takes (on my home machine).
- Generated meaningful graphs using MATLAB with data gathered from evaluation.

Milestones:

- Table/spreadsheet completed and filled out with all data required for a good write up. Spreadsheet and graphs sent to supervisor for reviewing - 24th Feb.

Lent weeks 7–8 (25th Feb - 9th Mar)

I will use my last block of work in lent as another 'slack' block for if I get behind on any work for any reason such as a high load of supervision work through the term. This should also be used to incorporate any final feedback from my supervisor.

If am up to date at this point I will work on implementing some of the extension tasks outlined in this document.

Milestones:

- System complete, success criterion met and demonstrated these to supervisor - 9th Mar.

Easter vacation weeks 1–2 (10th Mar - 23rd Mar)

Set up the dissertation document and write introduction and preparation chapters of the dissertation.

Deliverables:

- Introduction and preparation chapters of dissertation.

Milestones:

- Completed first draft of introduction chapter and given to supervisor - 17th Mar.
- Completed first draft of preparation chapter and given to supervisor - 23rd Mar.

Easter vacation weeks 3–4 (24th Mar - 6th Apr)
Write up the implementation and evaluation chapters of the dissertation.

Deliverables:

- Implementation and evaluation chapters of dissertation.

Milestones:

- Completed first draft of implementation chapter and given to supervisor - 31st Mar.
- Completed first draft of the evaluation chapter and given to supervisor - 6th Apr.

Easter vacation weeks 5–6 (7th Apr - 20th Apr)
Complete dissertation by completing the conclusion section and any appendices. I will also re-iterate through dissertation see if anything can be improved.

Deliverables:

- Draft of complete dissertation.
- 2nd draft of complete dissertation.

Milestones:

- Full first draft of dissertation handed into supervisor for review - 11th Apr.
- Hand in second draft of completed dissertation to supervisor - 18th Apr.

Easter term weeks 1–2 (21st Apr - 5th May)
Final 'slack' block for the dissertation. If the dissertation is completed then I will proof read multiple times. I would like to have the dissertation 'finished' by now to focus on revision for the exam and on learning any of the Easter term courses.

Milestones:

- System complete and success criterion met - 5th May.

Easter term week 3 (6th May - 13th May)
Final proof read and then an early submission so that I can focus on exam revision for the remainder of the term.

Deliverables:

- Dissertation.

Milestones:

- Hand in dissertation on time. (Project finished). - 13th May.

Appendix D

Glossary