Chapter 3

Machine Learning and Astroinformatics

Machine learning was once described to me by an anonymous supervisor as "the statistics kept at the back of the textbook". But even accepting its grounding in statistics, is this really an accurate description of the field? I think of machine learning as a data-driven way of formalising predictive problems mathematically, converting between different kinds of statistical problems, and an accompanying set of methods and practices for handling data and uncertainty. The eventual goal is to design some method or algorithm that automatically discovers useful information in (potentially very large) data sets. There are three core components of machine learning: the data, the model, and learning (Deisenroth, Faisal, and Ong, 2020). Before discussing these, we will look at the kinds of problems that machine learning solves.

3.1 Prediction

Machine learning aims to solve *prediction tasks*: problems where we have some data and we seek some kind of output based on that data. Central to prediction tasks are predictors.

3.1.1 Predictors

A *predictor* is some function that produces an output from some given input. A predictor can be represented as a function or as a probabilistic model, depending on the machine learning approach being undertaken. As a function, a predictor maps from some input domain $\mathcal{X}$ into some output domain $\mathcal{Y}$, and is usually written as

$$f : \mathcal{X} \to \mathcal{Y}. \tag{3.1}$$

$\mathcal{X}$ and $\mathcal{Y}$ are commonly (but certainly not always) a real vector space $\mathbb{R}^n$. Because the goal of machine learning involves *finding* a suitable function f for the task at hand, the set of functions is usually constrained. For example, if $\mathcal{X} = \mathbb{R}^n$, we might require that f is a linear function $\mathbb{R}^n \to \mathbb{R}$, easily parametrised by $n + 1$ constants. This constraint is called a *model*.

As a probabilistic model, a predictor is a joint probability distribution between observations and hidden parameters (Deisenroth, Faisal, and Ong, 2020). Using a probabilistic predictor allows us to formally describe and work with uncertainty both in the input space and output space. Such a predictor is usually parametrised by a finite set of parameters, which already includes most common probability distributions.

We will generally assume that our data are generated from some unobserved, true function called the *groundtruth*. This might be a physical process, or a complicated sampling function from some unknown vector space. The assumptions we make on this generative function can greatly change the way we approach machine learning problems.

In some sense, the goal of machine learning is to identify a good predictor from within the space of all possible predictors. Of course, this begs the question: what is a 'good' predictor? We will return to this when we discuss learning, but for now, a good predictor is one that well-approximates the groundtruth.

3.1.2 Classification

Classification is the machine learning task of predicting discrete, unstructured values (Deisenroth, Faisal, and Ong, 2020). These values are called *classes*. Classification is arguably the most important prediction task, as many other problems can be formalised as classification. Astronomy has its fair share of classification tasks, from classical astronomy tasks like galaxy morphology classification (appearing in machine learning literature as e.g. Dieleman, Willett, and Dambre, 2015) to transient detection (e.g. Scalzo et al., 2017); see Section 2.4 for more examples.

A classification problem seeks a predictor where $\mathcal{Y}$ is a finite, discrete set of classes. Classification tasks are usually delineated by the number of classes: much like astronomy's fascination with metals, classification tasks either have two classes, or more than two classes. The former are called *binary classification* tasks and the latter are *multiclass classification* tasks. The reason for this split is that binary classes are dramatically easier to reason about and analyse, and many special cases exist for binary where they do not for multiclass.

The two most common representations of $\mathcal{Y}$ for a binary task are $\mathcal{Y} = \{0, 1\}$ and $\mathcal{Y} = \{-1, 1\}$. In both cases 1 is called the *positive class*; 0 and -1 are both called the *negative class*. Throughout this thesis we will maintain the former convention with 0 and 1 as the negative and positive classes.

Many different tasks can be formalised as classification. An easy way to see why this is the case can be found by taking any prediction problem $\mathcal{X} \to \mathcal{Y}$ and reinterpreting it as the binary classification problem $\mathcal{X} \times \mathcal{Y} \to 0, 1$, i.e. instead of taking an input and predicting an output, take an input and a potential output and determine if they should be related. Of course this is not always the most efficient way to solve a prediction problem (and in the general case described here can be so inefficient as to be unsolvable) but the many known properties of classification make it an appealing framework to cast problems into. In Chapter 4, I will cast the radio astronomy problem of cross-matching galaxies seen in different wavelengths into a binary classification problem, and in Chapter 6 I will classify radio observations as Faraday complex and Faraday simple.

3.1.3 Regression

The other main kind of supervised prediction task is *regression*, which is the machine learning task of predicting ordered (and usually continuous) values. In a regression problem, we seek a predictor where $\mathcal{Y}$ is a set of ordered values, usually a subset of $\mathbb{R}^k$ for some positive natural k. Regression is ubiquitous in astronomy, from simple linear relationships like the famous Tully-Fisher relation (Tully and Fisher, 1977) to estimation of redshifts from photometric observations (Baum, 1962, called *photometric redshifts*; first introduced by). We will not directly address any regression problems in this thesis, but we will make use of their results.

3.2 Data and representation

Machine learning is centred on data and the extraction of information from that data. Data can include anything from numeric information, documents, or images, to spectra or galaxies. A collection of data is called a *dataset* and an element of this dataset is (interchangably) called an *example* or *instance*. Generally, data are not easy to work with in their original form and must be converted into a numerical representation before use. As it is relatively easy to work with both numerically and analytically, we usually convert our data into real vectors in $\mathbb{R}^n$. Each axis of this vector space is called a *feature* and the space as a whole is called the *feature space*. Features are non-trivial to choose, and finding good features often requires the expertise of a human who is well-versed in the original dataset (a *domain expert*). The process of finding features is called *feature selection*, or *feature design*.

What makes a feature good? Intuitively, we want to transform our data into a space where it is easy to work with: a space where properties we care about are obvious, or easy to extract. For this reason, features will vary tremendously depending on the problem being faced, and the same data may be represented in many different ways. Much of early machine learning literature focused on good methods for automatically developing features (generally called *feature extraction*), and much early *applied* machine learning focused on identifying these features manually so that good predictors could be easily found. An astronomical example is Proctor (2006), who developed features for representing radio galaxies for the purpose of sorting them. State-of-the-art models like deep neural networks (e.g. Dieleman, Willett, and Dambre, 2015) can be viewed as developing their own task-specific features as part of their training. A good feature space will have a structure that reflects the components of the intrinsic structure of the input data which are useful for the prediction task at hand. Good features may also be useful in other related tasks, such as dataset exploration, dataset visualisation, or other prediction tasks. Chapter 6 largely focuses on finding good features for identifying Faraday complexity in polarised sources.

Another very important piece of the machine learning puzzle are *labels*. Training a predictor with supervised learning requires some known pairs of inputs and outputs, and the known outputs are called labels. Like features, labels also need to be encoded in some way, and this depends on the specific task. Much like features, we want to embed the labels into a space which is easy to work with and has a meaningful structure. For problems where we know the outputs we wish to obtain, this can be a lot simpler than feature selection. For example, a binary classification problem will have only two possible outputs. Common representations for this would be $\{0, 1\}$ as described in Section 3.1.2, but we could also represent the labels as $\{[1, 0]^T, [0, 1]^T\}$, called a *one-hot encoding*. The advantage of the former is its simplicity and ease of integration into binary classification equations, but the advantage of the latter is that it is easily extended into multiclass classification without imposing order on the classes. Despite being simpler to encode, labels can carry a lot more difficulty than features due to their comparative rarity: in essence, features are cheap and labels are expensive. We will discuss labels in more detail in Section 3.5.

3.3 Training

Training a model is the process of using data to find a good predictor that fits the model's constraints. This is generally achieved by minimising a *loss* (also called *error* or *cost*) function over the model. In this section I will introduce loss functions and a common method of optimising parametrised models.

3.3.1 Loss functions

Put simply, a loss function is a function of a predictor and a dataset which describes how good the predictor is at predicting that dataset. Loss functions are high-valued for a predictor that poorly describes the dataset, and are low-valued for a predictor that well-describes the dataset. Usually the loss is minimised at zero, when the predictor perfectly captures the dataset (though whether this is a desired result is another question).

What should the loss function be for a given problem? The answer is not always obvious. Take for example a binary classification problem. The "obvious" loss would be the complement of the accuracy: the rate at which the predictor incorrectly guesses the label. This is easy to compute and we would like our predictor to have a high accuracy. But this is not a good choice: it is tremendously hard to work with as it takes on discrete values, because the accuracy is the number of correct predictions divided by the total number of examples. It is hard to motivate with probabilistic arguments. Finally, it is unclear how the accuracy should work in the case of a probabilistic model.

Instead, the loss function is usually derived by making assumptions on the structure of the data and task. The main assumption we usually make is that data are drawn *independently and identically distributed* (IID), that is, each example is drawn from the same distribution and is not dependent on any other examples. We also assume a structure of the noise in the observed labels: training data are almost never completely accurate, and so there will be intrinsic noise in the distribution of labels about their unobserved "true" value. To demonstrate these assumptions, we will now derive loss functions for regression and binary classification.

3.3.1.1 Loss function for regression

To derive a loss function for regression, let us assume that our labels are a random variable y modelled by a predictor $y = f(x)$. Further, let us assume that a predicted y is normally distributed about its true value, i.e.

$$y \sim \mathcal{N}(y \mid \mu, \sigma^2) \tag{3.2}$$

for the true mean μ and standard deviation σ. Under this assumption the probability that y is equal to a target t given an example x is

$$p(y = t \mid x) = \mathcal{N}(t \mid f(x), \sigma^2). \tag{3.3}$$

What would the probability be of observing a set of targets $T = \{t_1, \dots, t_n\}$ given corresponding examples $X = \{x_1, \dots, x_n\}$? Letting $Y = \{y_1, \dots, y_n\}$ be random variables like y, the joint probability distribution $p(Y = T \mid X)$ is

$$p(Y = T \mid X) = \prod_{i=1}^{n} p(y_i = t_i \mid x_i) \tag{3.4}$$

by using our independence assumption. We can then substitute the normal distribution:

$$p(Y = T \mid X) = \prod_{i=1}^{n} \mathcal{N}(t_i \mid f(x_i), \sigma^2). \tag{3.5}$$

$p(Y \mid X)$ is called the *likelihood*. We would like to maximise this likelihood over f, which is called a *maximum likelihood* approach to finding a predictor. It is, however, not very easy to work with in this current form. Maximising the likelihood is equivalent to minimising its negative logarithm, so we write:

$$\mathcal{L}(f; T, X) = -\sum_{i=1}^{n} \log \mathcal{N}(t_i \mid f(x_i), \sigma^2) \tag{3.6}$$

where $\mathcal{L}$ is the *negative log-likelihood*, a loss function. We can simplify this dramatically by cancelling the logarithm and the exponential within the normal distribution:

$$\mathcal{L}(f; T, X) = \sum_{i=1}^{n} \frac{(t_i - f(x_i))^2}{2\sigma^2} \tag{3.7}$$

and by noting that arbitrary scaling of $\mathcal{L}$ does not change the minimising f we can scale $\mathcal{L}$ by σ^2 and arrive at the *sum-of-squares error*, also known as the *least-squares* *error*, the most common loss function for regression:

$$\mathcal{L}(f; T, X) = \frac{1}{2}\sum_{i=1}^{n}(t_i - f(x_i))^2. \tag{3.8}$$

The factor of half helps keep the derivative tidy:

$$\frac{\mathrm{d}\mathcal{L}}{\mathrm{d}\theta}(f; T, X) = \sum_{i=1}^{n}(t_i - f(x_i))\frac{\mathrm{d}f}{\mathrm{d}\theta}(x_i). \tag{3.9}$$

3.3.1.2 Loss function for binary classification

As for regression, we first assume a form for the noise. Assume that our labels are a random variable $y \in \{0, 1\}$ and that the prediction y is drawn from a Bernoulli distribution based on a predictor $f(x)$:

$$p(y = t \mid x) = \mathcal{B}(t; f(x)). \tag{3.10}$$

The Bernoulli distribution is parametrised by one parameter, usually called $p \in [0, 1]$, and in this case set to $f(x)$. It is:

$$\mathcal{B}(a; p) = p^a(1 - p)^{1-a}. \tag{3.11}$$

It can be thought of as a biased coin toss with a probability p of tossing heads. To gain some intuition into how this expression works, imagine setting a to 0 and then to 1. Continuing to derive the loss function, we once again determine the likelihood making the IID assumption:

$$p(Y = T \mid X) = \prod_{i=1}^{n} p(y_i = t_i \mid x_i) = \prod_{i=1}^{n} f(x_i)^{t_i}(1 - f(x_i))^{1-t_i}. \tag{3.12}$$

Then we find the negative log-likelihood and hence the *binary cross-entropy loss* for binary classification:

$$\mathcal{L}(f; T, X) = -\sum_{i=1}^{n}\log\left(f(x_i)^{t_i}(1 - f(x_i))^{1-t_i}\right) \tag{3.13}$$

$$= -\sum_{i=1}^{n} t_i \log f(x_i) + (1 - t_i)\log(1 - f(x_i)). \tag{3.14}$$

3.3.2 Gradient descent

Given a loss function and a parametrised model, how can we find parameters for the model that minimise the loss function? There are many optimisation strategies but if both the loss function and model are differentiable with respect to the parameters then we can employ a particularly efficient approach: *gradient descent*. Assume we have a model $f(x; \vec{w})$ parametrised by some vector $\vec{w}$ and a loss function $\mathcal{L}(\vec{w}; T, X)$.

Then the value of $\vec{w}$ after the $k + 1$th update of gradient descent is

$$\vec{w}^{(k+1)} = \vec{w}^{(k)} - \lambda \nabla_{\vec{w}} \mathcal{L}(\vec{w}^{(t)}; T, X) \tag{3.15}$$

where $\lambda > 0$ is a small scalar called the *learning rate*. With an appropriately small choice of λ $\vec{w}$ will converge to a local minimum of $\mathcal{L}$. Many variations on this concept exist which attempt to avoid local minima, such as introducing a 'momentum' term that accumulates as multiple iterations move $\vec{w}$ in the same direction.

3.4 Models

This section describes some common models for classification. There are a plethora of different classification models and variations on these models, but I will present here only those relevant to this thesis: logistic regression, decision tree ensembles, and neural networks. These are, not coincidentally, also the most common models in astroinformatics. Logistic regression provides reliable and interpretable results. Decision tree ensembles are a fantastic off-the-shelf choice which work on a large variety of datasets. Neural networks have proved extremely effective for a wide variety of tasks, especially in computer vision.

3.4.1 Logistic regression

Logistic regression is a linear, binary, probabilistic classifier. Linear classifiers can only separate classes using a hyperplane in the feature space, with objects on one side of the plane being assigned to one class and objects on the other side being assigned to the other. A binary classifier works on binary classification tasks. Probabilistic classifiers, as discussed in Section 3.1.1, have outputs interpretable as class probabilities.

Logistic regression in a d-dimensional feature space is parametrised by a *weights vector* $w \in \mathbb{R}^d$. Given a set of features $x \in \mathbb{R}^d$, logistic regression is:

$$f(x; w) = \sigma(w^T x) \tag{3.16}$$

where σ is the *logistic function* or *sigmoid*:

$$\sigma(a) = \frac{1}{1 + e^{-a}}. \tag{3.17}$$

The output of logistic regression applied to an example x is the probability that x is in the positive class. σ, and thus logistic regression, has a domain of $(-\infty, \infty)$ and a range of $(0, 1)$. This enforces the output to be like a probability. $w^T x$ defines a d-dimensional hyperplane, called the *separating hyperplane* or *decision surface*.

Logistic regression is differentiable, which allows us to optimise its parameters w using gradient descent. Interpreting the classifier is possible through examining the weights vector, with a larger absolute value of a weight corresponding to a 'more important' feature.

A limitation of logistic regression is its sensitivity to scale. Features need to be of approximately the same order of magnitude and should have a standard deviation of approximately 1. An implicit assumption is that each features has a mean of 0 across the dataset. This can be enforced by normalising and scaling: subtract the mean of the dataset and divide by the new standard deviation.

3.4.2 Decision tree ensembles

A *decision tree* is a non-linear classifier. It repeatedly splits a dataset based on binary comparisons until every subset contains only one class (or mostly one class, with the amount of purity left as a hyperparameter). Each split only uses one feature for the comparison, making decision trees relatively easy to visualise and interpret. However, because of this, each split is axis-parallel, which can be a limitation for some datasets. They are not sensitive to scale and do not require a zero mean, making them easy to apply without preprocessing a dataset.

Key limitations of a decision tree are:

- They can only output a prediction, not a confidence of this prediction or a score of how likely an instance is to be found within each class.
- Small changes to the dataset or training method can result in large changes to the tree.
- They have high variance.
- With many low-information features, decision trees have quite poor performance (Breiman, 2001).

A *decision tree ensemble* aims to reduce some of these limitations by training multiple, slightly different, independently-trained decision trees. Depending on the implementation each constituent decision tree may only have access to some of the features or some of the data. To predict, each tree produces a prediction and 'votes' for this prediction; the votes can be combined to produce the overall prediction (e.g. with majority voting). A simple example of such an ensemble is decision tree bagging (Breiman, 1996), which trains each tree with a random subset of the training data and takes a plurality vote. Decision tree ensembles decrease variance, increase the usability of low-information features, and increase stability of the trained model.

The most well-known description of decision tree ensembles is the *random forest* (Breiman, 2001), which has found common use in astronomy partly to its readily available `Python` implementation in `scikit-learn` (Pedregosa et al., 2011b). Splits are decided from a subset of features and training samples are randomly drawn with replacement from the total training set. One downside of random forests is the large number of hyperparameters that need to be set, and these vary a lot depending on the problem being addressed.

3.4.3 Convolutional neural networks

talk about conv nets

3.5 Labels

As described in Section 3.2, labels are the known outputs of supervised prediction tasks. They are used for two main, distinct purposes: training and validation. Labels for training are used to evaluate loss and determine how to update the model. Labels for validation are used to evaluate and characterise the model's behaviour.

3.5.1 Where do labels come from?

I mentioned previously that labels were 'expensive'. This is to be interpreted as expensive in either or both time and money: labelling can be a slow, manual process, and labelling can be costly. Labelling is usually completed by hand, manually examining instances and determining what class they belong to (for classification) or what target they ought to be associated with (for regression).