# Machine Learning

Todd Davies

January 3, 2015

## Introduction

Machine learning is concerned with creating mathematical "data structures" that allow a computer to exhibit behaviour that would normally require a human. Typical applications might be spam filtering, speech recognition, medical diagnosis, or weather prediction. The data structures we use (known as "models") come in various forms, e.g. trees, graphs, algebraic equations, probability distributions. The emphasis is on constructing these models automatically from data—for example making a weather predictor from a datafile of historical weather patterns. This course will introduce you to the concepts behind various Machine Learning techniques, including how they work, and use existing software packages to illustrate how they are used on data. The course has a fairly mathematical content although it is intended to be self-contained.

## Aims

To introduce methods for extracting rules or learning from data, and provide the necessary mathematical background to enable students to understand how the methods work and how to get the best performance from them. This course covers basics of both supervised and unsupervised learning paradigms and is pitched towards any student with a mathematical or scientific background who is interested in adaptive techniques for learning from data as well as data analysis and modelling.

## Additional reading

| | | |
|---|---|---|
| Introduction to machine learning (2nd edition) | Alpaydin, Ethem | 2010 |

# Contents

# 1 Machine Learning

Machine Learning is the creation of self-configuring data structures that allow a computer to do things that would be classed as 'intelligent' if a human did it.

Machine learning has been around in it's infancy since the 40's, where reasoning and logic were first studied by Claude Shannon and Kurt Godel. Steady progress was made with lots of funding through until the 70's, where people then realised Artificial Intelligence was very hard, causing funding to dry up, and the term 'AI Winter' to be coined.

In the 80's people started to look at biologically inspired algorithms such as neural networks and genetic algorithms, and there is more investment. This leads to the field of AI diverging into many other fields such as Computer Vision, NLP, Machine Learning etc. In the 00's, ML begins to overlap with statistics, and the first *useful* applications emerge.

# 2 Nearest Neighbour Classifier

The Nearest Neighbour (NN) classifier is a simple implementation of a machine learning algorithm. We can give it a set of training data with each point labelled by a class, and then give it another datapoint, and that datapoint will be classified according to the data.

The premise is that you plot all the points of the training data on a graph, and when you want to classify another datapoint, you simply plot it on the existing graph, and classify it by the classes of it's nearest neighbour(s).

One nice aspect of the NN classifier is that you can use it with as many features as you like with little extra effort. A standard implementation of the NN classifier might plot graphs in two dimensions, and thus will only be able to classify data with two features. However, if you want to include more (or less) features, you need to adjust the number of dimensions on your graph to match the number of features you're including. Then when you find the nearest neighbours on the graph, you find them in $n$-dimensional space rather than 2-dimensional space, where $n$ is the number of dimensions on the graph.

## 2.1 Finding the distance between two $n$-dimensional points

In order to calculate which points in the training dataset are the nearest neighbours to the new data point, we can calculate the *Euclidean Distance* between the two points. In $n$ dimensions, this how:

$$d(p,q) = \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2 + \cdots + (p_{n-1} - q_{n-1})^2 + (p_n - q_n)^2}$$

This is actually a very simple computation. If you're given two arrays (containing the coordinates of your points) of equal lengths, then you can do something similar to Listing 1.

```
1 def euclideanDistance(c1: List[Double], c2: List[Double]): Double = {
2   val delta: Set[Double] = for {(a,b) <- c1 zip c2} yield math.pow(a-b, 2)
3   Double = math.sqrt(delta.sum)
4 }
```

Listing 1: Scala Euclidean Distance

The reason why using KNN on n-dimensions is that you can classify different types of data. A 2d KNN classifier could classify on two variables, such as weight and height, while a 256d classifier could classify a 16x16 monochrome image where each pixel is represented by a dimension.

## 2.2 Computing the nearest neighbour

Computing the nearest neighbour to a point is simply as easy as finding the point in the training data that has the smallest euclidean distance to it, as shown in Listing 3.

```
def nearestNeighbour(input: Point, existing: Set[Point]): Point = {
  existing.min((p: Point) => p.euclideanDistance(input))
}
```

Listing 2: Scala Nearest Neighbour

Assume that there is a custom implementation of Point that exposes the method to find the euclidean distance between it and another point, similar to that in Listing 1

## 2.3 Multiple nearest neighbours (K-NN)

It's possible that more accurate classifications of points could be obtained by taking into account multiple close neighbours rather than one nearest one. In order to do this, you need to find the number of occurrences of a specific class in the top $K$ nearest neighbours:

```
def kNearestNeighbour(k: Int, input: Point, existing: Set[Point]): Class = {
  val topK: List[Class] = existing.toList.sorted(
      (p: Point) => p.euclideanDistance(input)
    ).take(k).map(_.class)
  topK.groupBy(identity).maxBy(_._2.size)._1
}
```

Listing 3: Scala Nearest Neighbour

One must be careful when choosing a value of $K$ for the classifier. As $K$ increases proportional to the size of the dataset, then the number of incorrect classifications will increase.

Figure 1 shows two identical datasets that a KNN classification is being applied to. When $K = 3$, the correct classification of a square is made, and when $K = 5$, the wrong classification is made, simply because there are more circles than squares in the dataset.

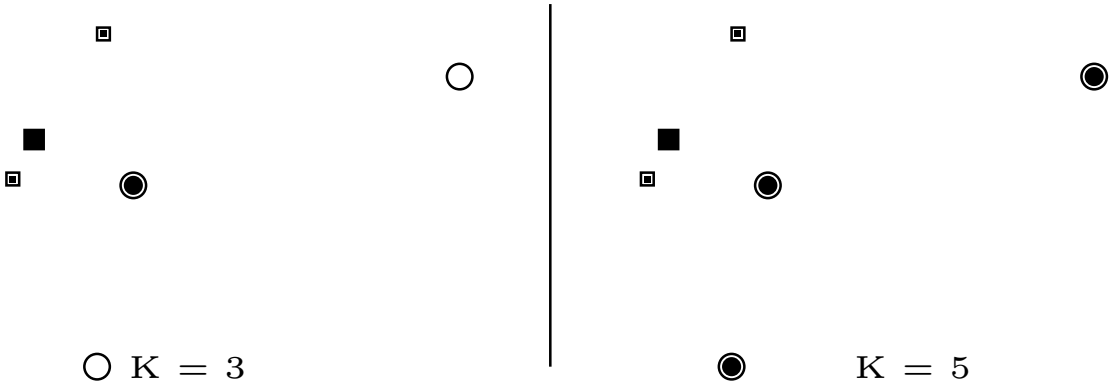


Figure 1: How K affects classification accuracy. The item being classified is the filled square and nearest neighbours are filled, and other elements are left unfilled. When K=3, it would be classified as square, when K=5, it'd be classified as a circle.

With a nearest neighbour classifier, there is always one or more *boundaries* that defines which class a new item will be placed in, as shown in Figure 2. Be aware that the **Decision Boundary** isn't always contiguous or a straight line.

## 2.4 Overfitting

Overfitting is a problem with all ML algorithms, and occurs when an algorithm is trained on a small proportion of the dataset, that isn't Representative of the whole data. It can also be when

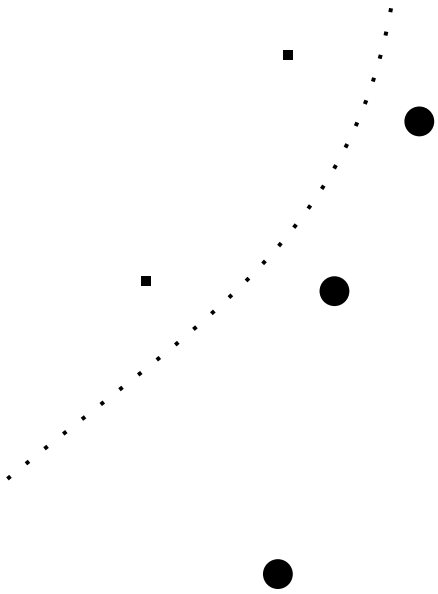

Figure 2: The dotted line represents the decision boundary for a KNN classifier

the algorithm is trained in such a way that it describes the noise or random error in the training set.

Avoiding overfitting is hard, but generally it involves choosing the right algorithm, the right parameters and the right training data.

# 3 Linear Classifier

The nearest neighbour algorithm requires a minimal amount of logic in the training stage, since it's really only the value of $K$ that needs to be considered. In fact, for a simple knn classifier, you don't even need to train it, since you could just pick an arbitrary value for $K$, and then all the work of the algorithm would be done at the 'Model' stage? However, most other ML algorithms require some kind of learning phrase before they can become useful. The simplest of these is probably the linear classifier.

A simple linear classifier is one that looks at only one parameter, and can classify into two classes. Essentially it says:

```
if(parameter > threshold) class1 else class2
```

Listing 4: A simple linear classifier

This classifier would work well for some limited use cases; maybe if we were classifying boxers into different weight categories. However, we need to decide on a threshold value in order for the classifier to work. In the case of boxing this is easy, since there are known weight categories, however, in some other cases, we might have to discover the threshold value from the data.

If there are two classes in the data which are linearly separable (i.e. there is one or more threshold(s) that will accurately tell them apart), then an algorithm to tell them apart is easy to write so long as you know the correct threshold value. In order to find the right threshold value, you need a learning algorithm such as this:

```
var errors: Int = 0
var threshold: Int = 0;
do {
  // classify will use a linear classifier to classify the data with a specific
  // threshold, and return the number of errors.
  errors = classify(data, correctLabels, threshold++)
```

```
} while(errors > 0);
```

Listing 5: Linear classifier learning algorithm

In all learning algorithms, an error function is needed to evaluate whether the changes to the parameters for the classifier on each iteration have had a positive or negative effect on the accuracy of classification. The error function in this case is `while(errors > 0)`, since we are assuming that the data is linearly separable, we can just keep incrementing the threshold value until we get a one that produces no errors.

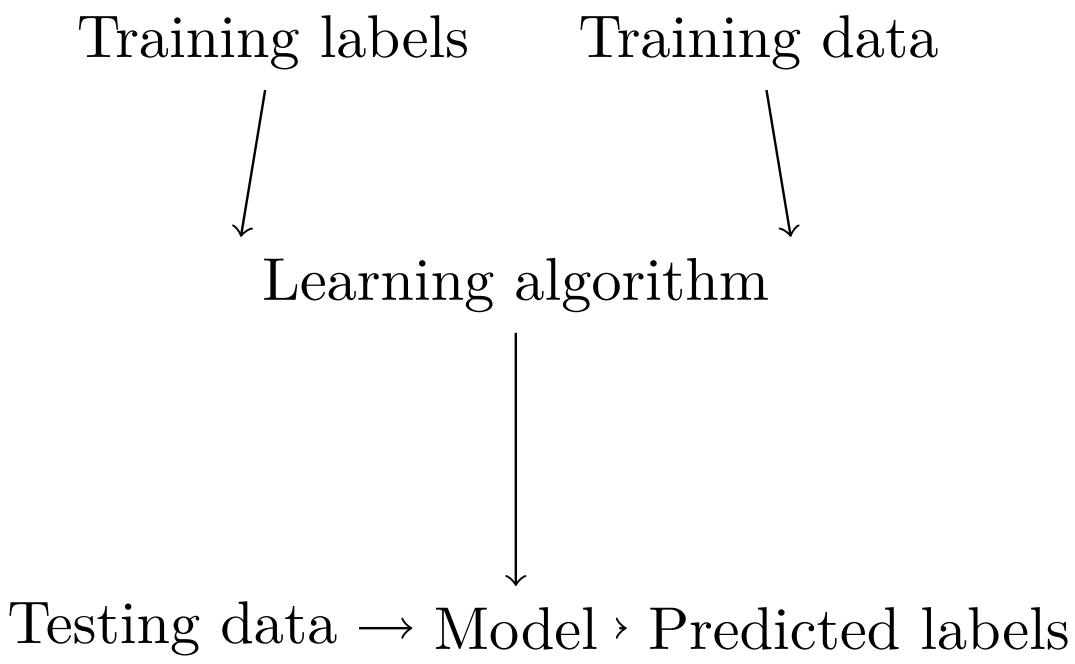


Figure 3: A generic ML algorithm always has the above structure.

## 4 Perceptron

A peceptron is an ML algorithm that mimics a single neuron in a brain. Despite emulating only one neuron, a peceptron can accurately classify a wide variety of data.

The algorithm has $d$ inputs $(i_0, \ldots, i_d)$, and $d$ weights $(w_0, \ldots, w_d)$, where each input has a specific weight associated with it. It also has another parameter $t$, which is the classification threshold. The algorithm can be expressed easily in mathematical notation:

$$classify(I) = \begin{cases} 1 & \text{if } t \leq \sum_{j=0}^{d} i_j w_j \\ 0 & \text{otherwise} \end{cases}$$

For the less mathematically inclined, here's the MATLAB code that does the same thing:

```
function output = perceptron(threshold, inputs, weights)

% Compute the activiation level
activation_level = sum(inputs.*weights);

if activation_level > threshold
  output = 1;
else
```

```
9     output = 0;
```

Listing 6: A perceptron implementation in MATLAB

Therefore:

```
$ inputs = [0.1, 0.4, 0.2]
$ weights = [0.9, 0.1, 0.7]
$ threshold = 0.14
$ perceptron(threshold, inputs, weights)
  > 1
$ threshold = 0.32
$ inputs = [0.1, 0.6, 0.1]
$ perceptron(threshold, inputs, weights)
  > 0
```

Training a perceptron is a matter of adjusting the weights and the threshold to get the best classification accuracy on the training data. We need a rule that we can apply over and over again to the parameters to refine them towards better values:

$$weight = weight \times learning\_rate \times (actual - output) \times input$$

Henceforth, we can create a learning algorithm for the perceptron. Lines 5 and 6 of Listing **??** are the implementation of the perceptron learning rule, everything else is the logic to loop over all the weights a certain number of times.

```
1  for(int i = 0; i < iterations; i++) {
2    for(int example = 0; example < trainingSet.length; example++) {
3      int output = perceptron(threshold, trainingSet[example], weights);
4      for(int weight = 0; weight < weights.length; weight++) {
5        int scaleFactor = learningRate * (trainingLabels[example] - output);
6        weights[weight] = weights[weight] * trainingSet[example][weight] *
             learningRate;
7      }
8    }
9  }
```

Listing 7: A perceptron learning algorithm in Java

Note how computationally intensive the training algorithm is. The time complexity is $O(i \cdot t \cdot w)$ where $i$ is the number of training iterations to do, $t$ is the number of training examples, and $w$ is the number of weights/inputs/dimensions to classify with.

Even though training is expensive, classification is relatively cheap, with a linear runtime.

There is a theorem, called the **Perceptron Convergence Theorem**, that states "if the data is linearly separable, then application of the perceptron learning rule will find a decision boundary within a finite number of iterations".

## 4.1 Multilayer Perceptrons (MLP)

A a multilayer perceptron is basically a graph structure, where every node is a perceptron, and edges are connections from the output of one perceptron to the input of another. In order to make this work, a different type of perceptron is often used. Instead of having a threshold value that is compared to the sum of the weighted inputs, the sum is run through a sigmoid function before being output.

With a network of perceptrons, the decision boundary can now be curved and doesn't have to be linear (as it is with only one perceptron). This means more complex problems can be attempted.

In order to train a neural network, a technique called *backpropogation* is used. However, this is beyond the scope of this course.

# 5 Decision trees

A decision tree is a tree of questions, where the answers to each question will either lead to another question, or a classification/answer. They are good at handling categorical data, and worse at handling continuous data, since they require a specific answer to progress to the next level of the tree.

## 5.1 Building a decision tree

An algorithm to build a decision tree is relatively easy to come up with. Listing 8 shows an example algorithm (adapted from the course notes).

```
1  Tree learnTree(data) {
2    if(isAllSameLabel(data) != null) {
3      return new Leaf(isAllSameLabel(data));
4    } else {
5      Tree out = new Tree();
6      Feature importantFeature = extractImportantFeature(data);
7      for(Value v : importantFeature.values) {
8        out.addBranch(v, learnTree(importantFeature.rowsWithValue(data, v)));
9      }
10     return out;
11   }
12 }
```

Listing 8: An algorithm (the ID3 algorithm) to produce a decision tree in Java

This is kind of understandable, but there are some quirks. How do we extract an important feature?

### 5.1.1 Entropy

Entropy is the amount of information contained in a variable, given the symbol $H$.

$$H(x) = -\sum_i p(x_i) log_2 p(x_i)$$

We measure entropy in bits, since we're using log of base 2.

When we're choosing an important feature, we want to reduce the entropy in the system, so that we gain the maximum amount of information. The best feature to choose is the one where the $H(T) - H(T|F)$ is largest, as defined by:

$F$ is a feature, such as windy, or bottle size...

$$\begin{aligned} H(T) &= \text{The entropy before the split} \\ H(T|F_1) &= \text{The entropy of the data on the first branch} \\ H(T|F_n) &= \text{The entropy of the data on the n'th branch} \\ H(T|F) &= \text{The weighted average of the entropy on all the branches} \end{aligned}$$

If we had the following table:

| **Colour** | **Bottle Size** | **Class** |
|---|---|---|
| Red | Big | Wine |
| Red | Big | Beer |
| Yellow | Small | Cider |
| White | Big | Wine |
| Yellow | Small | Beer |

We can compute $H(T)$:

$$\begin{aligned} H(T) &= -\sum_i p(x_i) log_2 p(x_i) \\ &= -(\frac{1}{5} log_2 \frac{1}{5} + \frac{2}{5} log_2 \frac{2}{5} + \frac{2}{5} log_2 \frac{2}{5}) \\ &= 1.52193 \end{aligned}$$

If we choose the size to be our next question:

$$\begin{aligned} H(T|S = Small) &= -\left(\frac{1}{2} log_2 \frac{1}{2} + \frac{1}{2} log_2 \frac{1}{2}\right) \\ &= 1 \\ H(T|S = Big) &= -\left(\frac{1}{3} log_2 \frac{1}{3} + \frac{2}{3} log_2 \frac{2}{3}\right) \\ &= 0.91830 \\ H(T|S) &= \frac{2}{5} H(T|S = Small) + \frac{3}{5} H(T|S = Big) \\ &= 0.95098 \end{aligned}$$

If we chose colour:

$$\begin{aligned} H(T|C = Red) &= -\left(\frac{1}{2} log_2 \frac{1}{2} + \frac{1}{2} log_2 \frac{1}{2}\right) \\ &= 1 \\ H(T|C = Yellow) &= -\left(\frac{1}{2} log_2 \frac{1}{2} + \frac{1}{2} log_2 \frac{1}{2}\right) \\ &= 1 \\ H(T|C = White) &= -\left(\frac{1}{1} log_2 \frac{1}{1}\right) \\ &= 0 \\ H(T|C) &= \frac{2}{5} H(T|C = Red) + \frac{2}{5} H(T|C = Yellow) + \frac{1}{5} H(T|C = White) \\ &= 0.8 \end{aligned}$$

What should we choose then? Well:

$$\begin{aligned} H(T) - H(T|C) &= 1.52193 - 0.8 && = 0.72193 \\ H(T) - H(T|S) &= 1.52193 - 0.95098 && = 0.57095 \end{aligned}$$

Therefore we should choose colour to be our first question, since it has the maximum value of $H(T) - H(T|F)$

### 5.2 Overfitting decision trees

It is important to ensure that decision trees are not overfitted. The most extreme case of overfitting, is when you have $n$ rules, and $n$ pieces of data in your dataset (i.e. you have a rule for every piece of data).

In order to stop overfitting, we can either stop generating the tree after a certain depth (which also keeps it small and efficient), or we can prune the tree after we've built it to make it smaller.

Figure 4 shows when it's best to start pruning the decision tree.

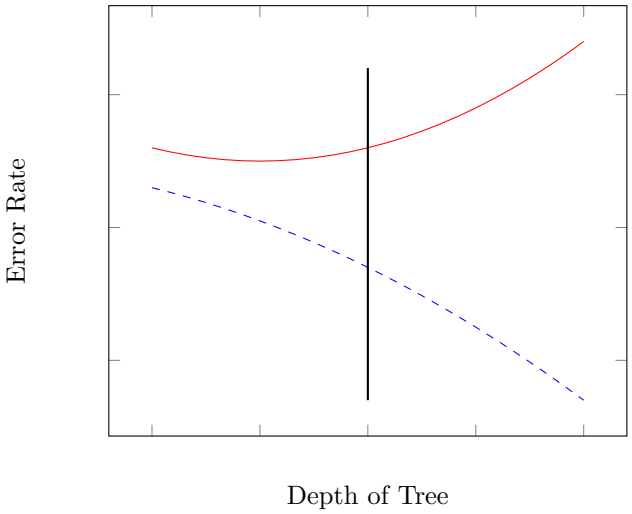


Figure 4: The vertical line shows the cutoff point, where the error rate for testing with the validation data (red, unbroken) begins to rise, while the error rate for the training data (blue, dashed), which we're generating the decision tree from, is still improving. It is here that we should prune the tree.

## 6 Learning experiments in Machine Learning

The way we train our ML algorithms, is to get all the data that we have access to and split it in half. Then, we can train on the first half and test our parameters on the second half. Often we don't split the data into equal halves, but make the training set smaller than the testing set.

When we test our algorithm on the datasets, we usually get a height 'testing error' than 'training error'. This is because we have trained our ML algorithm on the training data, so it will usually be better at classifying it.

### 6.1 Cross validation

To make better use of our data, we could split our **training data** into $n$ chunks, train on $n-1$ chunks and test on the one that has been left out, before rotating the chunks so that a different one is for testing. Then, see which parameters were best in our training rounds, and use them on the training data.

The aim is to make sure that all the data has an equal chance of appearing in the training or the testing set.

### 6.2 Dealing with misclassifications

If we had one class that was very rare, and another that was almost ubiquitous, then it would be hard to train our algorithm, since we would always get low error rates by always classifying as the most common class.

The solution to this is to measure the accuracy on each class separately, and try to get a distributed, but low error rate over all the classes.

We can create a confusion matrix to analyse our errors:

| | | Prediction | |
|---|---|---|---|
| | | Class 1 | Class 2 |
| Actual value | Class 1 | Correct1 | False-Positive1 |
| | Class 2 | False-Positive2 | Correct2 |

If Class 2 was the rare one, then we could calculate our *sensitivity* (the chances of correctly classifying it when given it) by doing:

$$\frac{Correct2}{Correct2 + False - Positive2}$$

We could calculate our *specificity*, which is the chance of correctly classifying a member of Class 2 (the common one) by doing:

$$\frac{Correct1}{Correct1 + False - Positive1}$$

Confusion Matrices are also useful if falsely classifying one class is worse than falsely classifying another, since you can work out the cost of false classifications:

$$cost = (\#False - Positive - 1 \times cost1) + (\#False - Positive - 2 \times cost2)$$

This is called ROC (Receiver Operator Characteristics). It was developed in world war two (hence the weird name) to help analyse radar classifications of bombers.

## 7 Ke Chen's types of classifiers

Ke is very keen to emphasise the following:

**Discriminative** classifiers both model a classification rule directly (such as a decision tree), and model the probability of class memberships based on input data.

**Generative** classifiers make a probabilistic model of data within each class.

**Probabilistic** classifiers use probabilities to classify data.

Generative classifiers are always probabilistic classifiers (at least it seems that way in his notes!)

If that's not very clear, discriminative classifiers give a probability for *each* class based on input data, while generative classifiers are trained on a specific class, and give a probability for that class.

## 8 Naive Bayes Classifier

Rather than me warbling on about this; have a gander at what Sebastian Raschka has to say instead. He has prettier diagrams, and a better understanding than I do:

- `http://sebastianraschka.com/Articles/2014_naive_bayes_1.html`

- http://sebastianraschka.com/PDFs/articles/naive_bayes_1.pdf