Machine Learning

Todd Davies

January 10, 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 2.

```
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.

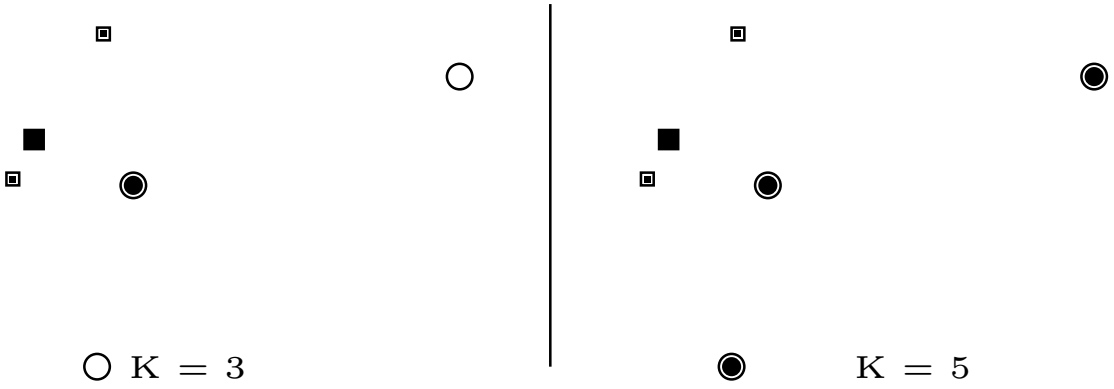

Figure 1: How K affects classification accuracy. The item being classified is the filled square and nearest neighbours are filled, and other elements are left unfilled. When K=3, it would be classified as square, when K=5, it'd be classified as a circle.

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

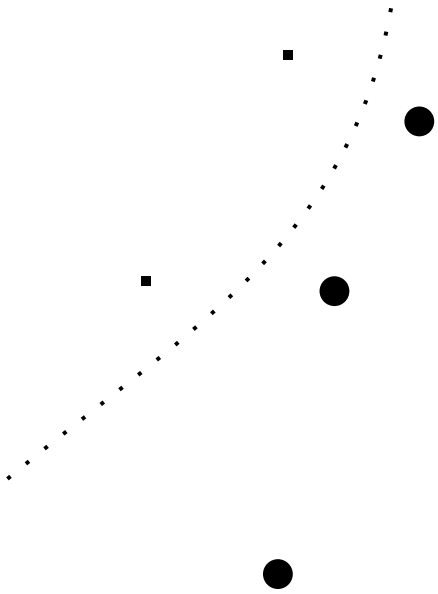

Figure 2: The dotted line represents the decision boundary for a KNN classifier

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.

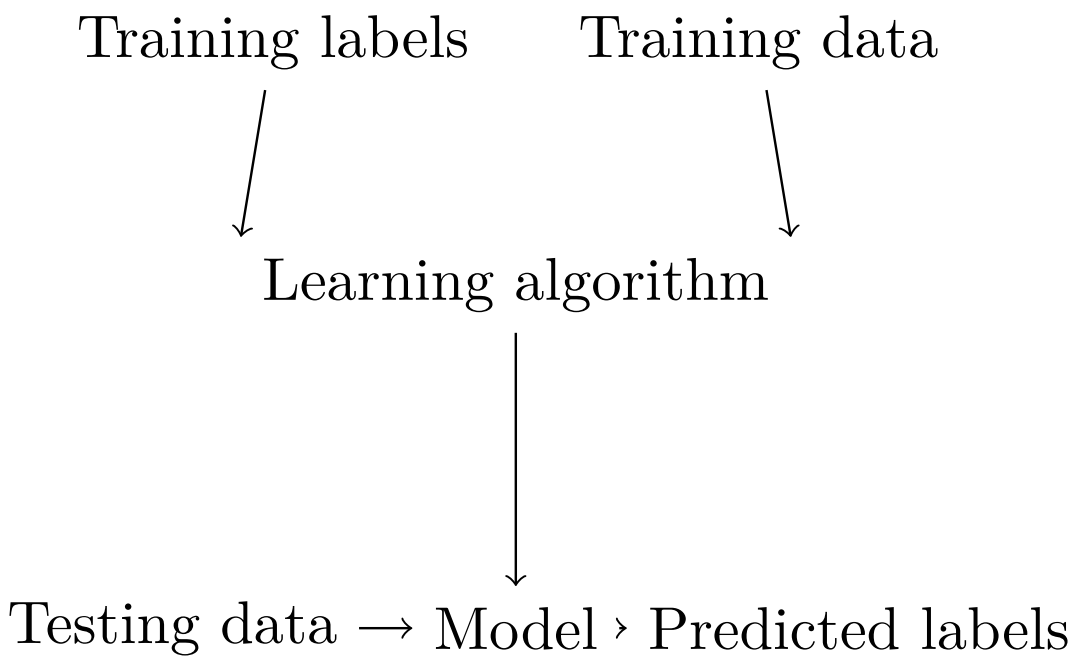

Figure 3: A generic ML algorithm always has the above structure.

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.

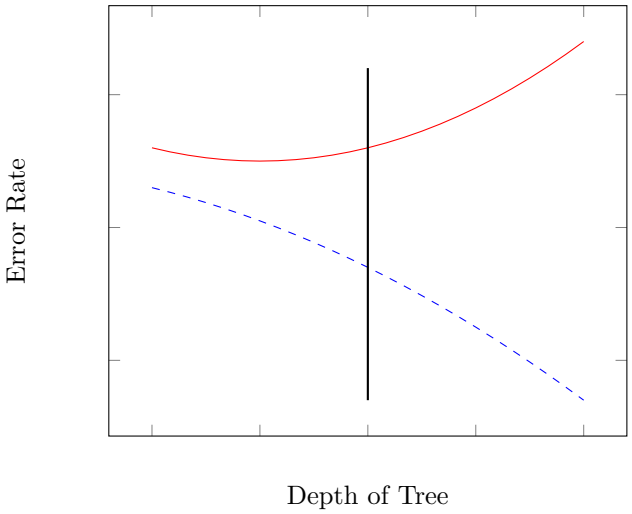

Figure 4: The vertical line shows the cutoff point, where the error rate for testing with the validation data (red, unbroken) begins to rise, while the error rate for the training data (blue, dashed), which we're generating the decision tree from, is still improving. It is here that we should prune the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

An Ensemble System fits multiple models (called base learners) to the training data, and when classifying, is uses the models as a *committee* to vote on the testing data.

In order to create a useful ensemble, the members (different models) can't be identical; they need to disagree. The amount of disagreement is called the diversity. We don't want too much disagreement, since then we won't be able to make a decision.

The way to create diverse members is to use a slightly different training dataset for each model. Cross validation and feature extraction can be good for this.

Not all models are suitable as base learners, if the model only changes by a small amount when the training data is changed, then it won't have much diversity. To ensure the ensemble is diverse, the base learner should produce large changes to it's model for smaller changes in the training data. A good base learner is therefore *unstable*, an example of such an algorithm is the decision tree.

7.1 Bootstrapping

Bootstrapping is a way of generating multiple datasets from an original dataset. You select N training examples from the total N with replacement.

This means that on average, 36.8% of points are unselected.

7.2 Bagging

BAGGING is an acronym for Bootstrap AGGregatING. It's an ensemble method that generates m bootstraps, and trains m models on each. In order to classify new data, a simple majority vote is used.

Bagging is a parallel algorithm; ensemble members work independently to reach a conclusion and then the results are collated.

7.3 Boosting

Boosting is basically bagging, except the bootstrapping part is on roids:

1. Take a bootstrap of the dataset.
2. Train one model on the bootstrap.
3. Take a look at which examples the model gets wrong.
4. Upweight the 'hard' examples and downweight the 'easy' ones.
5. Go back to step 1, but with a weighted bootstrap until you run out of models to train.

This means that each new member of the ensemble focuses on thee ones that the previous member got wrong. It's a serial algorithm, since members are trained one after the other rather than at the same time.

8 Types of ML classifiers

Ke Chen 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.

9 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`

10 Clustering Analysis

A **cluster** is a collection of data objects or points that are similar to one another. Objects in other clusters will be dissimilar to objects in this cluster.

Unsupervised learning algorithms try and find hidden structure in unlabelled data. Unlike supervised learning algorithms, there is no way to evaluate potential solutions, since the data is unlabelled.

Cluster analysis involves finding similarities between data and grouping similar data points into clusters. It's a type of *unsupervised learning*, and is used both as a stand alone tool to gain an insight into data, but also as a pre-processing step for other algorithms.

As a simple example, you could split a list of animals (*dog, cat, shark, minnow, frog, worm*) into ones that have legs (*dog, cat, frog*) and ones that don't (*shark, minnow, worm*). This is a trivial example, but the technique is used in many fields:

- **Banking/Internet Security** (Fraud detection)
- **Biology** (Classification of species)
- **Climate Change** (Better understanding the climate, finding atmospheric patterns)
- **Finance** Uncover correlation in underlying shares
- **Image Compression** (group similar pixels)
- Many more...

10.1 Representing data

10.1.1 Data matrix

The easiest way to represent data is using a matrix, where the rows represent each data point, and the columns represent each feature. It's good at representing data with n datapoints (rows) and p dimensions (columns). Also known as *object by feature*. Data matrices can be said to have two 'modes' since their rows and columns can be interchanged.

10.1.2 Distance matrix

This is a triangular matrix that describes the distance between points.

	A	B	C	D
A	0	4	6	3
B	4	0	2	7
C	6	2	0	5
D	3	7	5	0

You can see that the matrix is symmetrical, so we generally only show the bottom left side (to save confusion and space). If you shade the cells of the matrix in according to their value, you can get a heat map of the data, which can reveal hidden patterns.

You may not have considered that there may be different ways of computing the distance between coordinates, but we will cover three ways in this course. In the following list, n is the number of data points, and p is the number of features:

Euclidean Distance
$d(a,b) = \sqrt{|a_1 - b_1|^2 + \cdots + |a_n - b_n|^2}$

Manhattan Distance
$d(a,b) = |a_1 - b_1| + \cdots + |a_n - b_n|$
This one would find the distance you'd have to go if you could only go in straight lines, and turn in 90° increments (so it's the distance you'd have to walk in Manhattan).

Minkowski Distance
$d(a,b) = \sqrt[p]{|a_1 - b_1|^p + \cdots + |a_n - b_n|^p}, p > 0$

10.1.3 Cosine Similarity

Cosine similarity is a measure of similarity between two vectors in n dimensional space. It measure the cosine of the angle between the vectors; $cos(0°) = 1$, while any angle other than 0° is less than 1. It is therefore a measure of the *orientation, irrespective of the magnitude* of the two vectors. Diametrically opposed vectors will have a similarity of -1.

In order to calculate the cosine similarity, you use this formula:

$$\cos(x, y) = \frac{x \cdot y}{\|x\|\|y\|} = \frac{\sum_{i=1}^{n} x_i y_i}{\sqrt{\sum_{i=1}^{n} (x_i)^2} \times \sqrt{\sum_{i=1}^{n} (y_i)^2}}$$

In order to find the **distance** between two vectors using cosine similarity, you do:

$$d(x, y) = 1 - cos(x, y)$$

One example use of the cosine similarity, is in text mining, where if you have documents represented as a vectors, where each word is given a dimension that represents the number of times it appears in the document, cosine similarity can find how similar to each other, documents are.

Here's an example:

$$\begin{aligned}
x &= (1, 2, 3, 4, 5) \\
y &= (0, 3, 4, 7, 9) \\
x \cdot y &= (1 * 0) + (2 * 3) + (3 * 4) + (4 * 7) + (5 * 9) \\
&= 0 + 6 + 12 + 28 + 45 \\
&= 91 \\
\|x\| &= \sqrt{1^2 + 2^2 + 3^2 + 4^2 + 5^2} \\
&= \sqrt{1 + 4 + 9 + 16 + 25} \\
&\approx 7.416 \\
\|y\| &= \sqrt{0^2 + 3^2 + 4^2 + 7^2 + 9^2} \\
&= \sqrt{9 + 16 + 49 + 81} \\
&\approx 12.450 \\
cos(x, y) &= \frac{91}{7.416 \cdot 12.450} = 0.986 \\
d(x, y) &= 1 - 0.986 = 0.14
\end{aligned}$$

10.1.4 Measuring the distance of binary features

You can measure the distance between binary features by converting them all to 1 and 0 (if you've not already), and draw a *contingency table* for them:

		y	
		1	0
x	1	a	b
	0	c	d

Where:

- a is the number of features that are 1 for x and y
- b is the number of features that are 0 for x and 1 for y
- c is the number of features that are 1 for x and 0 for y
- d is the number of features that are 0 for x and y

To find the actual distance for a symmetric binary attribute (where both classes are equal (e.g. male and female)):

$$d(x, y) = \frac{b + c}{a + b + c + d}$$

If the attributes are asymmetric (one is more rare than the other, say if the test for a disease is positive (important) or negative (less important)), set the rarest one to 1 and do:

$$d(x, y) = \frac{b + c}{a + b + c}$$

For example:

Name	Dairy	Pollen	Gluten	Fish	Mushrooms	Peanuts
Bob	1	0	0	0	1	0
Bill	0	1	0	0	1	1

To find the distance (these would be asymmetric binary attributes), you do:

$$d(bob, bill) = \frac{1 + 2}{1 + 2 + 1} = \frac{3}{4} = 0.75$$

10.2 Distance for nominal features

If the values for dimensions are discrete, we can convert them into more binary dimensions. For example, if we had one colour dimension, that had $\{red, blue, green, black\}$ as it's values, we could convert that into four binary dimensions (and then condense is back into one dimension (e.g. 1000, or 0100), and do binary distance measuring on it.

A simpler (but worse) way of finding the distance between two nominal entities is to do:

$$d(x, y) = \frac{\text{number of mis-matching features}}{\text{total number of features}}$$

10.3 Clustering approaches

These are the main clustering approaches:

Partitioning approach
: You can partition the data with a view to evaluating all of the partitions by some criteria (e.g. minimising the sum of distances). K-means and k-medoid does this.

Hierarchical approach
: Aim to classify the data into hierarchies (e.g. draw increasingly small circles around datapoints). Agglomerative, Diana, Agnes, BIRCH are all examples of this.

Density approach
: Create a function that will calculate the density or connectiveness of the data at a point, and use that to form clusters.

Spectral approach
: Convert the data into a weighted graph, and cut the graph into sub-graphs that correspond to clusters.

Ensemble approach
Combine multiple clustering techniques to generate clusters.

10.3.1 K-means clustering

As previously mentioned, k-means is a partitioning clustering approach. It iteratively partitions the training data to produce several non-empty clusters. The idea is to produce optimal partitions in which the sum of the squared distance to the 'representative object' is minimised.

Since partitioning the dataset into K optimal partitions is NP-Hard, the problem must be solved using heuristic algorithm. The end goal is to minimise the value of:

$$\sum_{k=1}^{K} \sum_{x \in S_k} d^2(x, m_k)$$

Where K is the number of partitions to find, S_k is the kth partition (a set) of the data, and m_k is the 'Representative object' (or the mean of the points) in the set S_i.

The algorithm works like so:

1. Pick K random seed points (from the data).
2. Assign each data object to the cluster with the nearest seed point (measured with whatever distance metric you're using e.g. Manhattan).
3. Compute the mean points of the clusters (aka centroids).
4. Go back to Step 1), but the seed points are now the means calculated in Step 3). Do this until the assigned sets don't change between iterations.

When K-means has been applied to a dataset, the K partitions are mutually exclusive, and therefore partition the dataset. A *Voronoi Diagram* is created.

K-means is fairly easy to compute using the heuristic algorithm. The runtime is $O(tKn)$, where t is the number of iterations, K is the number of clusters, and n is the number of objects. K and t are usually much smaller than n.

There are a number of problems with K-means:

K-Medoids can mitigate bullet 3, and K-modes can mitigate bullet 6.

- The initial seed points can cause 'local optimum' clusters, which may not be representative of the whole data.
- The number of clusters K needs to be specified in advance, but may be unknown.
- Noisy data and outliers mess up the algorithm.
- If a cluster has a non-convex (i.e. concave) shape, then it won't be detected.
- Unable to classify categorical data (unless you modify the algorithm).
- It's hard to evaluate the performance of the algorithm.

10.3.2 Hierarchical Clustering

Hierarchical clustering partitions the data sequentially, construction partitions layer by layer by grouping objects together into a tree. A distance matrix is used to determine clusters.

There are two sequential strategies used to construct a tree of clusters:

The Agglomerative approach is a bottom up strategy, whereas the diversive approach is a top down strategy.

Agglomerative
Each data object is in it's own cluster initially, before they are merged into progressively larger clusters.

Divisive
All objects start in a single cluster, which is then divided into smaller and smaller clusters.

You can measure the distance between clusters in three ways:

Single Link
Use the *smallest* distance between an element in one cluster and an element in another.

Complete Link
Use the *largest* distance between an element in one cluster and an element in another.

Average
Find the average distance between the points in the two clusters and use that.

For example, given a set of named one dimensional locations $\{a = 1, b = 3, c = 7, d = 8, e = 9\}$, and the clusters $C_1 = \{a, b\}, C_2 = \{c, d, e\}$, find the distances between them:

$$\begin{aligned}
\text{Single Link} &= min(d(a,c), d(a,d), d(a,e), d(b,c), d(b,d), d(b,e)) \\
&= min(6, 7, 8, 4, 5, 6) = 4 \\
\text{Complete Link} &= max(d(a,c), d(a,d), d(a,e), d(b,c), d(b,d), d(b,e)) \\
&= max(6, 7, 8, 4, 5, 6) = 8 \\
\text{Average Link} &= \frac{(d(a,c) + d(a,d) + d(a,e) + d(b,c) + d(b,d) + d(b,e))}{6} \\
&= \frac{6 + 7 + 8 + 4 + 5 + 6}{6} = 6
\end{aligned}$$

One issue with hierarchical approaches is what value to choose for K (when we've found enough clusters). One heuristic is to stop processing when the distance between clusters reaches a certain threshold.

10.3.3 Hierarchical - Agglomerative

The agglomerative algorithm is as follows:

- Convert the object attributes into a distance matrix, and make each object a cluster (of size 1).
- Merge the two closest clusters together
- Update the distance matrix to take into account the new cluster
- Repeat from step two until we've got the desired number of clusters.

Since we have fewer clusters as the algorithm runs for longer, updating the distance matrix will take less and less time (if you can't work out why, check out slide 10 of Ke's second lecture.

You can show the creation of clusters with a dendrogram, as in Figure 5. The x axis in a dendrogram is the datapoints, and the y axis is the distance. The lines merge when the agglomerative algorithm merges two clusters, and the number under the horizontal merge line indicates the ith merge. The data that would produce such an algorithm is in Table 1.

	a	b	c	d	e
a	0	3	6	7	9
b	3	0	3	4	6
c	6	3	0	1	3
d	7	4	1	0	2
e	9	6	3	2	0

Table 1: A distance matrix for a hierarchical clustering algorithm.

The agglomerative approach has some weaknesses; it's very sensitive to noise and outlying datapoints, and it is less (usually) efficient than k-means clustering, with a runtime of $O(n^2)$.

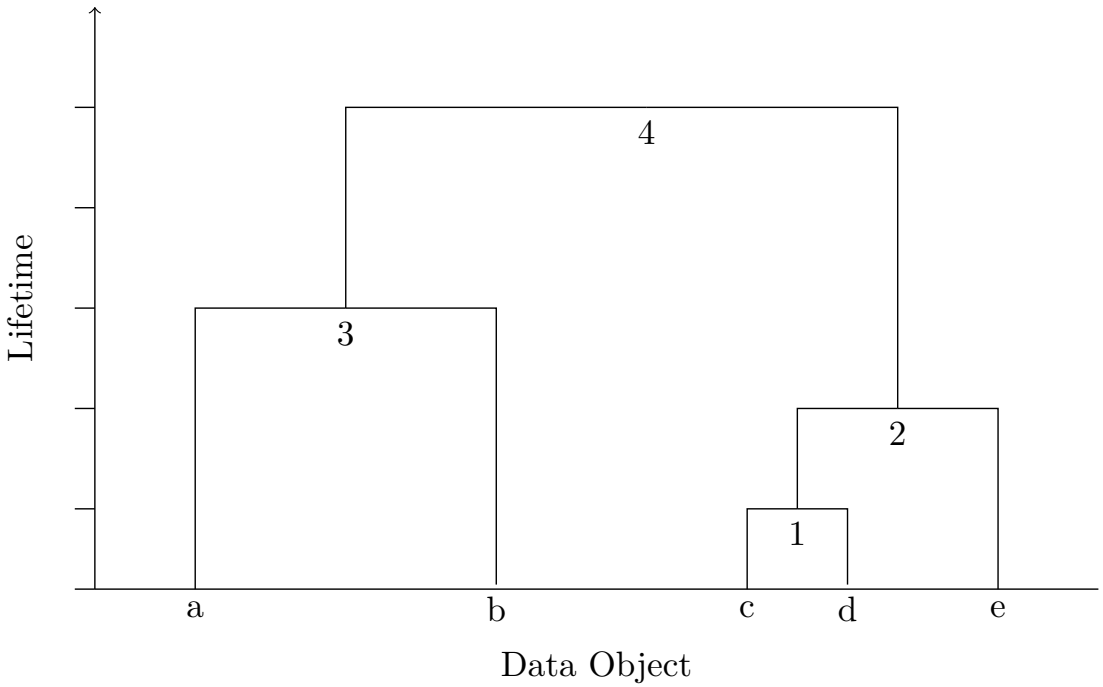

Figure 5: A dendrogram produced from running the agglomerative algorithm on the data in Table 1.

10.4 Cluster validation

Cluster validation is the art of evaluating the results of clustering algorithms in a quantitative and objective manner.

10.4.1 Internal Indexes

Internal indexes are used to validate the value of K for our clustering algorithms when the 'ground truth' is unavailable. Because of this, common sense and prior knowledge is used to validate the clusters.

Ground Truth is the term used to describe the training data that we can use for our algorithm

We can use variance to evaluate the clusters, checking to see that the intra- cluster variance is minimised, and the inter-cluster variance is maximised.

Intra-cluster = within a cluster. Inter-cluster = over multiple clusters. In this context, a low intra-cluster, and high inter-cluster variance will make for more disjoint, widely spaced, yet tightly packed clusters.

This analysis generates an 'F-ratio index' that we can use to determine how well our clustering algorithms have done. In order to calculate the F-ratio index, we the divide the intra-cluster variance by the inter-cluster variance:

$$F(m) = \frac{mSSW(m)}{SSB(m)} = \frac{m \sum_{i=1}^{m} \sum_{j=1}^{n_i} d^2(x_{ij}, c_i)}{\sum_{i=1}^{m} n_i d^2(c_i, c)}$$

While this algorithm looks like something that walked out of Euler's notebook, it's actually not that hard:

Variable	Meaning
m	The number of clusters generated by the algorithm.
n_i	The number of datapoints in the ith cluster.
c_i	The centroid for the ith cluster.
x_{ij}	The jth datapoint in cluster c_i.
c	The mean centroid for the whole dataset.
$d(x, y)$	The distance function we're using.

The way to use the variance analysis, is to find the number of clusters with the smallest f-ratio index. To do this, run the clustering algorithm ranging from K (the number of clusters) set as 2, up to n (an arbitrary limit). Then find the F-ratio index for each K value, and use the value with the lowest F-ratio index.

10.4.2 External Indexes

When the ground truth is available, but our clustering algorithm doesn't make use of the information (probably because it's unsupervised), we can evaluate the effectiveness of the clustering algorithm using external indexes. Unlike internal indexes, which find a good value for K, external indexes are used to find a performance evaluation for our clustering efforts.

One issue of external indexes, is that if we have a different number of classes to the number of clusters, then we'll probably get a lower score for our clusters.

We can use the Rand Index do this, however, I don't understand enough of the course notes to write a well explained description here.