Self-Organizing Maps

In this lecture. . .

- Assigning samples to groups
- Evolving the group vectors
- Visualizing in two dimensions

Introduction

A self-organizing map is an unsupervised-learning technique.

We start with vectors of features for all our sample data points. These are then grouped together according to how similar their vectors are.

The data points are then mapped to a two-dimensional grid so we can visualize which data points have similar characteristics.

What Is It Used For?

Self-organizing Maps are used for

- Grouping together data points according to similarities in their numerical features and visualizing the results
- Example: Group people according to the contents of their supermarket baskets
- Example: Group countries together according to demographics and economic statistics

The end result of a self-organizing map (SOM) is a typically two-dimensional picture, consisting of a usually square array of cells. Each cell represents a vector and individual items will be placed into one of the cells.

Thus each cell forms one group and the closer that two cells are to one another the more they are related.

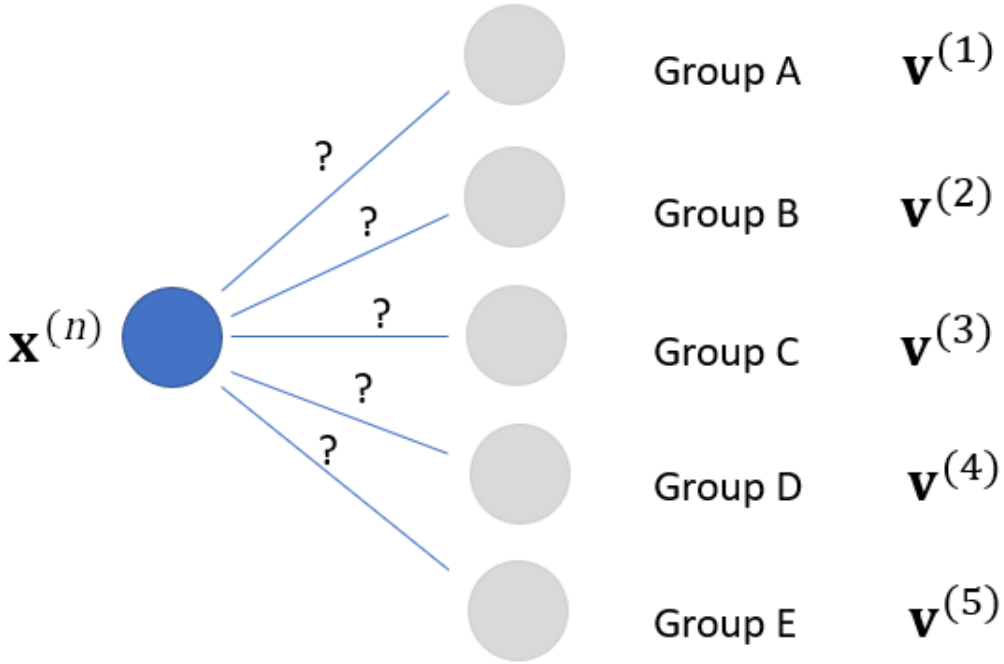
The Method

Each item that we want to analyze and group is represented by a vector, $\mathbf{x}^{(n)}$, dimension M, of numerical data. There will be N such items, so $n = 1$ to N.

As is often the case we shall be measuring and comparing distances. So we need to make sure that the magnitude of distances are similar for all elements. Just as we've done before we should translate and scale, to either make mean and standard deviations the same, or the minimum and maximum, for all features.

Each group/cell/node is represented by a vector $\mathbf{v}^{(k)}$, also of dimension M. And we'll have K such groups. For visualization purposes in two dimensions it is usual to make K a square number, 25, 64, 100.... These $\mathbf{v}$ are often called 'weights.'

But we are getting ahead of ourselves. Before we go on to talk about two-dimensional arrays and visualization let's talk about how we determine in which group we put item n. We have the vector $\mathbf{x}^{(n)}$ and we want to decide whether it goes into Group A, B, C, D, . . . , this is represented below.



Remember that all of $\mathbf{x}^{(n)}$ and the $\mathbf{v}$s are vectors with M entries. All we do is to measure the distance between item n and each of the K cell vectors $\mathbf{v}$:

$$|\mathbf{x}^{(n)} - \mathbf{v}^{(k)}| = \sqrt{\sum_{m=1}^{M} \left(x_m^{(n)} - v_m^{(k)} \right)^2}.$$

The k for which the distance is shortest

$$\underset{k}{\text{argmin}} \, |\mathbf{x}^{(n)} - \mathbf{v}^{(k)}|,$$

is called the **Best Matching Unit** (BMU) for that data point n.

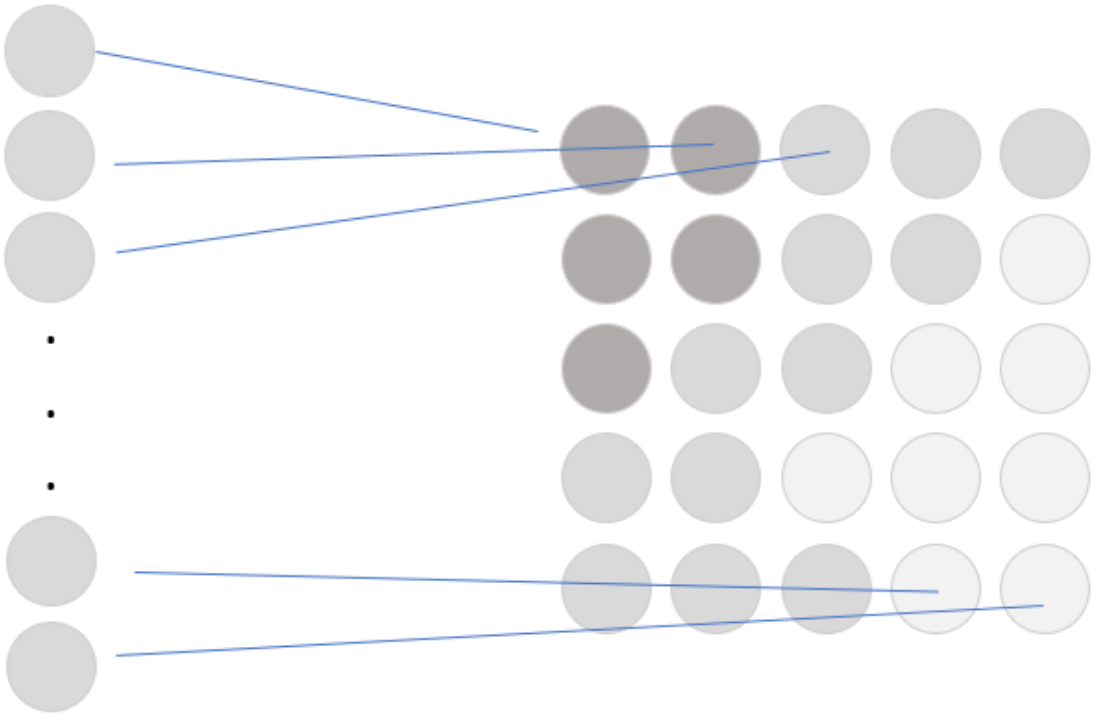
But how do we know what the $\mathbf{v}$s are?

Finding the BMU is straightforward. But that's assuming we know what the $\mathbf{v}$s are.

We could specify them manually: Classify dogs according to different characteristics.

But, no, instead we are going to train the system to find them.

We begin by redistributing our $\mathbf{v}$ vectors in a square array, as shown below. However unless there is some relationship between cells then this redistribution is without any meaning. More anon.



We are now ready to bring everything together. . . via learning.

The Learning Algorithm

Step 0: Pick the initial weights

We need to start with some initial $\mathbf{v}$s, which during the learning process will change. As always there are several ways to choose from, perhaps pick K of the N items, or randomly generate vectors with the right order of magnitudes for the entries.

Step 1: Pick one of the N data vectors, the $\mathbf{x}$s, at random

Suppose it is $n = n_t$.

Find its BMU. We are going to be picking from the $\mathbf{x}$s at random as we iterate during the learning process.

Keep track of the number of iterations using the index t.

So t starts at 1, then next time around becomes 2, and so on.

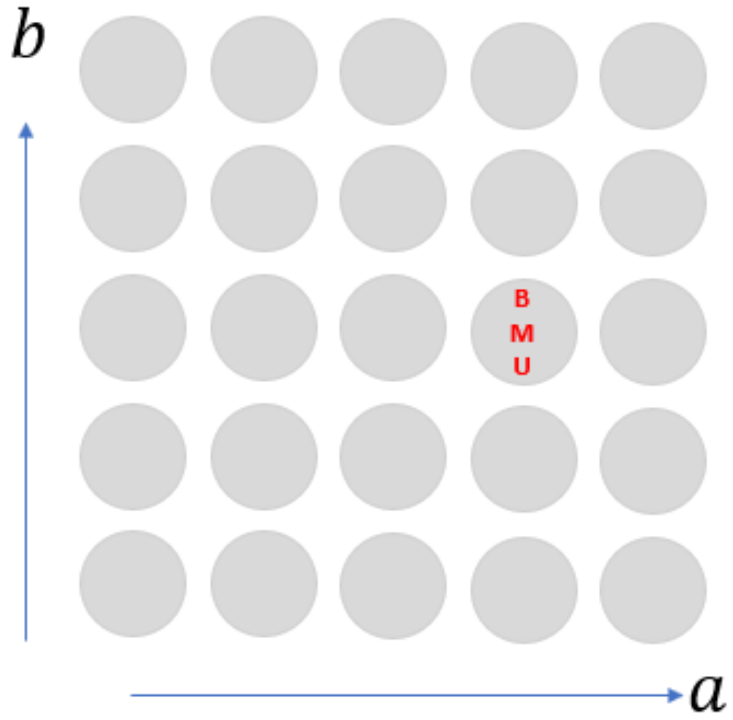
Step 2: Adjust the weights

Change all of the weights $\mathbf{v}^{(k)}$ for $k = 1$ to K slightly to move in the direction of $\mathbf{x}^{(n_t)}$.

The closer a cell is to the BMU the more the weight is adjusted.

This is where it gets interesting. Let's see the mathematical details.

What do we mean by 'closer'? This is where the topography of the two-dimensional array comes in, and the distribution of cells becomes meaningful. Look at the figure which shows the current BMU and surrounding cells.



Measure the distance between the BMU Cell and a weight, according to the obvious

$$\text{Distance} = D = \sqrt{(a - a_{\text{BMU}})^2 + (b - b_{\text{BMU}})^2}. \qquad (1)$$

This distance is going to determine by how much we move each $\mathbf{v}^{(k)}$ in the direction of our randomly chosen item $\mathbf{x}^{(n_t)}$. The as and bs are integers representing the position of each node in the array.

Note that this is not the same distance we measured when we were comparing vectors. It is the distance between the cells, as drawn on the page and nothing to do with the vectors that these cells represent.

And here is the updating rule:

$$\mathbf{v}^{(k)} \leftarrow \mathbf{v}^{(k)} + \beta\left(\mathbf{x}^{(n_t)} - \mathbf{v}^{(k)}\right) \quad \text{for all} \quad 1 \leq k \leq K.$$

Here β is the learning rate.

It is a function of the distance, D, in (1) and the number of iterations so far, t.

For example, we might choose the function $\beta(D,t)$ to be

$$\beta(D,t) = \eta(t) \exp\left(-\frac{D^2}{2\sigma(t)^2}\right)$$

where

$$\eta(t) = \eta_0 \exp\left(-\frac{t}{\tau_\eta}\right)$$

and

$$\sigma(t) = \sigma_0 \exp\left(-\frac{t}{\tau_\sigma}\right).$$

This can be interpreted as:

- The adjustment to each weight decreases with 'time' (i.e. iteration)
- The adjustment to each weight decreases with distance from that iteration's BMU
- The effective range over which each weight is adjusted decreases with iteration

If entries in the weights are mean zero, range of one, then a suitable η_0 might be around 0.2. And σ_0 might be around $\frac{1}{2}\sqrt{K}$. There are typically two phases during the learning process. The first is a 'topological ordering' where adjacent cells change to have similar weights and then convergence.

Step 2: Iterate Go back to Step 1 above and repeat until convergence.

Example: Grouping shares

Suppose you want to classify constituents of the S&P500 index. You could try:

1. Grouping according to Sector, Market Capitalization, Earnings, Gender of the CEO,. . .

2. Measure expected returns, volatilities and correlations

3. Principal Components Analysis of returns

But that's what everyone does. The second of these is the popular (at least in the text books) Modern Portfolio Theory (MPT).

What I am going to do is a slightly unusual example of self-organized maps. Usually one would have significantly different features for each stock. Perhaps we would have the Sector, Market Capitalization, Earnings, Gender of the CEO, etc. mentioned above.

But I want to tie this in more closely to the MPT of classical quantitative finance. For that reason I am going to use annual stock price returns as a feature.

So for each stock I will have a vector of five 'features,' the last five annual returns.

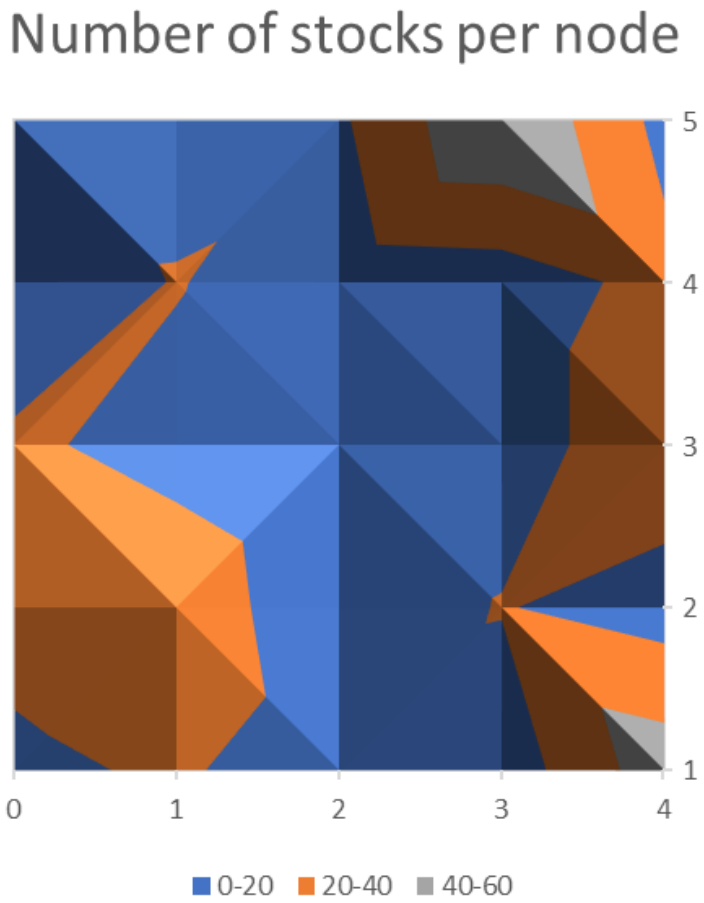
I will use annual returns for each of 476 constituents of the S&P index. ("Why not 500?" you ask? "Because some stocks left the index and others joined during this period," I reply.)

These are the $\mathbf{x}^{(n)}$s. So the 'features' of the stock are returns and the m^{th} entry in each $\mathbf{x}^{(n)}$ is the return over the m^{th} year.

And I will have $K = 25$.

A note on scaling: Because I've chosen features which are very similar, just returns over different years, scaling is not as crucial as it would be with features that are significantly different. So in what follows I have left all data unscaled.

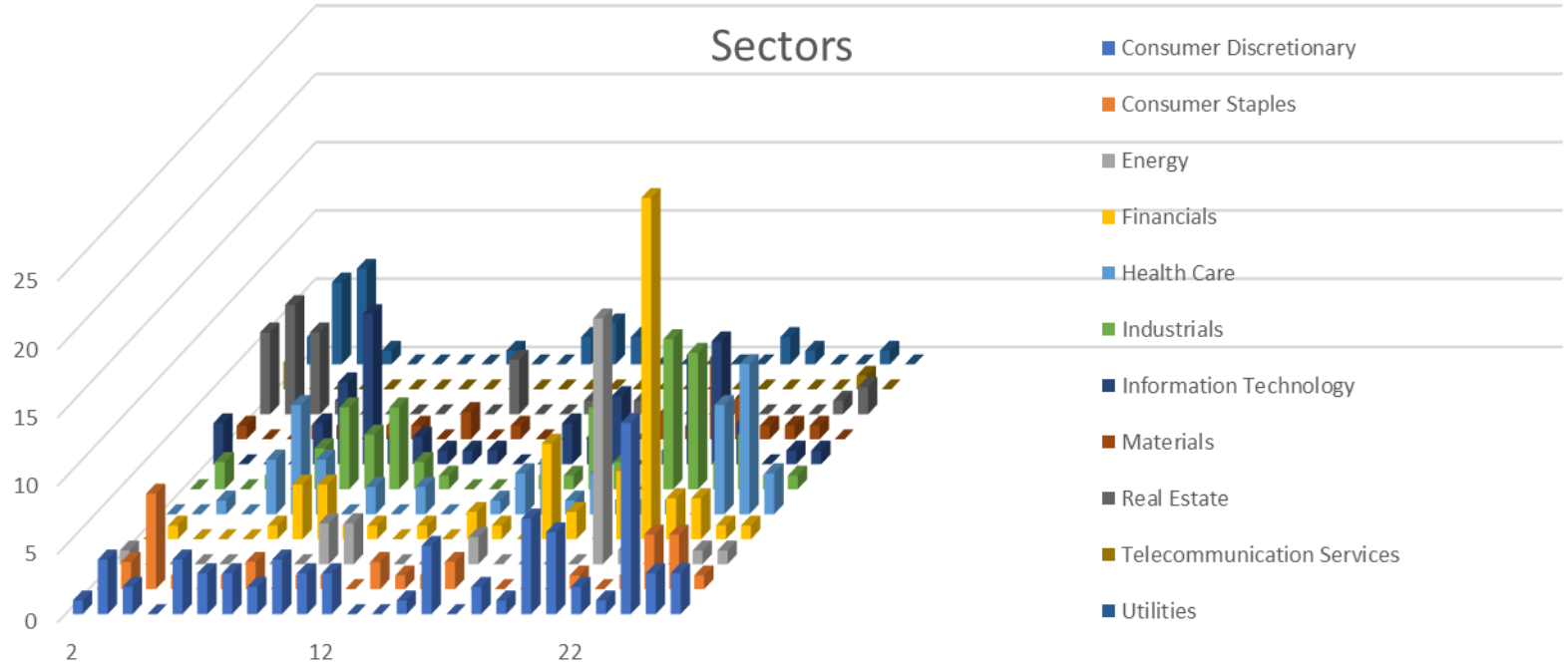
Below we see a contour plot how many of the 476 stocks appear in each node. Some nodes are more popular than others. In those nodes there are a lot of stocks that move closely in sync.



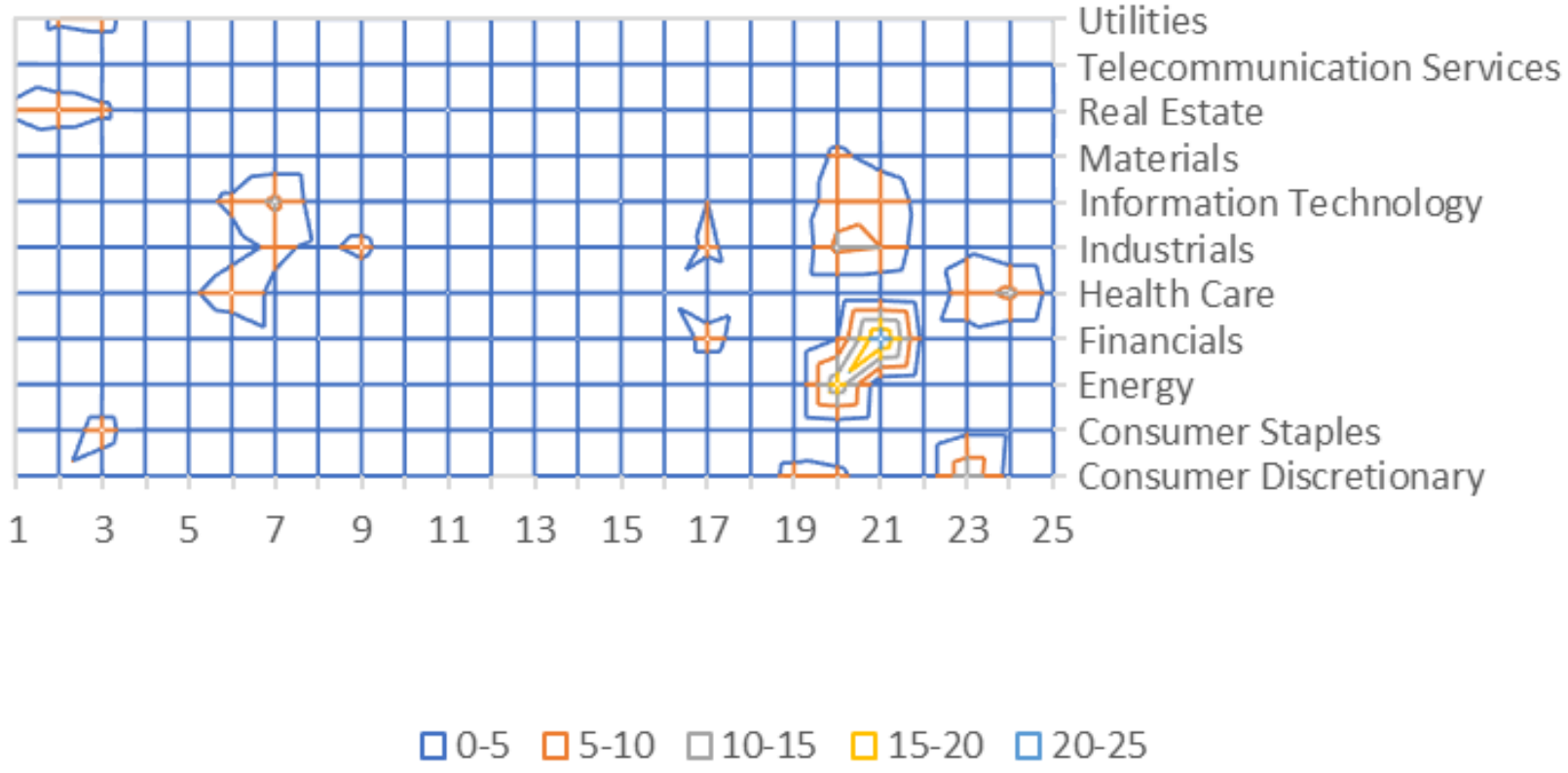
The most popular node, the one with the most stocks, is number 20. Its constituents are:

ADM	BAX	DVN	GRMN	KMX	NI	PWR	WDC
ADS	BEN	EBAY	HAL	KSU	NOV	PXD	WMB
AKAM	BWA	EOG	HES	LNC	NRG	QCOM	WYNN
AMG	CF	ETN	HOG	LUK	NSC	RRC	XEC
APA	CHK	FCX	HP	MOS	NTAP	STX	
APC	CMI	FLS	HST	MRO	OKE	UAL	
ARNC	COP	FMC	IP	MU	PKG	UNP	
AXP	DISH	FTI	KMI	NBL	PNR	URI	

I want to finish this example by looking at what this solution has to say about financial sectors. In the figure I show the number of stocks from each sector in each cell.



And this is repeated as a contour map:



There are clearly some sectors that are very focused: Financials are concentrated around Node 21; Real Estate around Nodes 1–3. Energy around Node 20.

There are some sectors that are very spread out, such as Consumer Discretionary.

Note that I never told the algorithm anything about sectors. But it does look as if SOM has found something rather sector-like, but only for some sectors.

Example: Voting in the House of Commons

At time of writing the UK is going through some political turmoil.

So I couldn't resist doing something with voting in Parliament. I downloaded voting data from `https://www.publicwhip.org.uk`.

I took the last 18 months of voting records of Members of Parliament, who voted, in which direction and who abstained or wasn't present.

There are 650 members of Parliament, the MPs. However, the data gets a bit confusing. Some MPs don't appear in the data because Sinn Fein MPs never attend the House of Commons, they don't fancy swearing allegiance to the Queen apparently. (Is Northern Ireland still part of the UK?) And quite a few MPs have been kicked out of the Labour party, and others have become independent. Every time something like that happens the MP gets a new ID Number which complicates things a bit. I haven't dealt with the redesignation of such MPs properly, but it would be interesting to look at their voting before and after changing allegiance.

TBH it was all happening faster than I could type these notes!

Each entry in the vector represents an individual's position on a specific vote.

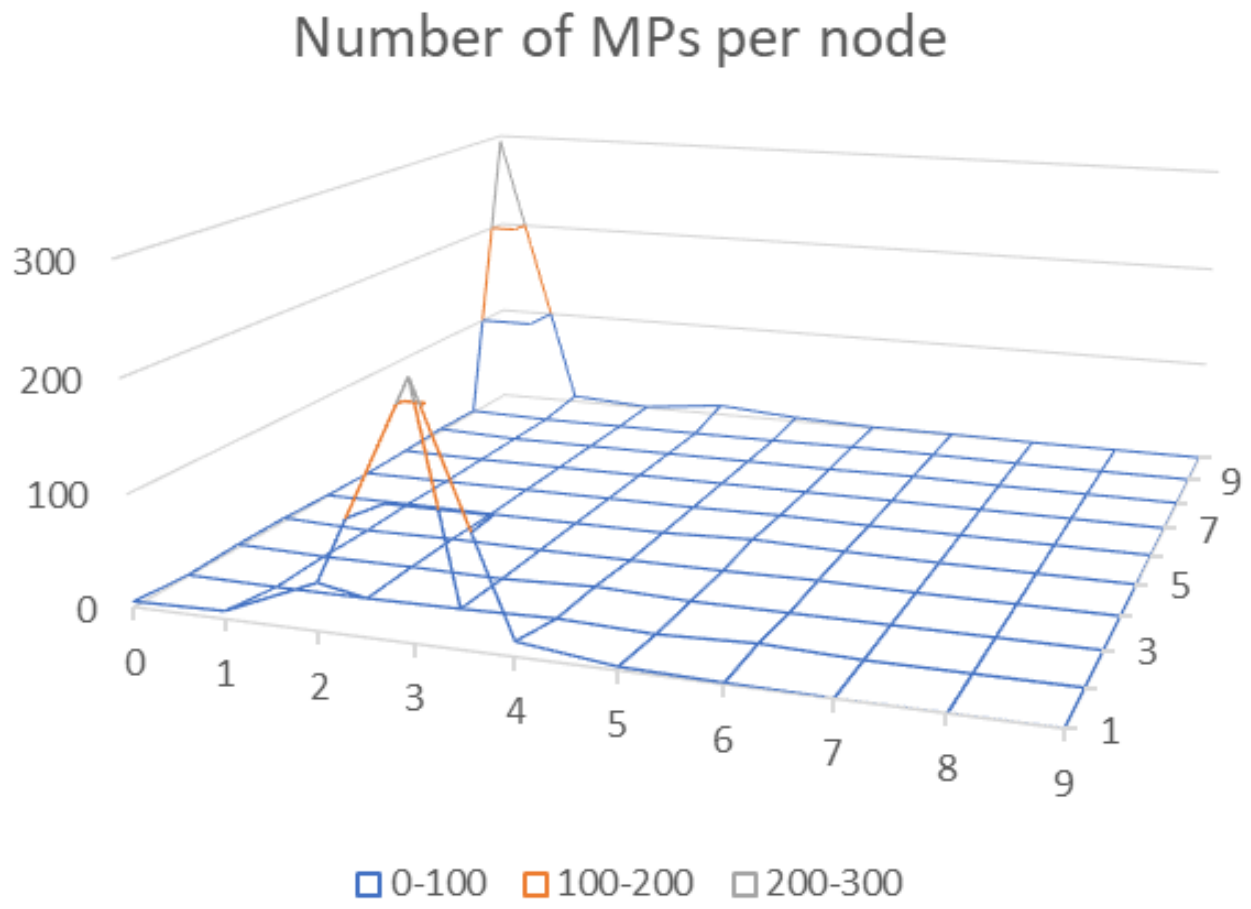
I have used 1 to represent a vote in favour, -1 for a vote against, and zero for abstention or not being present.

Now obviously turning such data into numbers has to be meaningful for this method to work.

Fortunately the ordering 'no,' 'abstention,' 'yes' is quite well represented by -1, 0, 1.

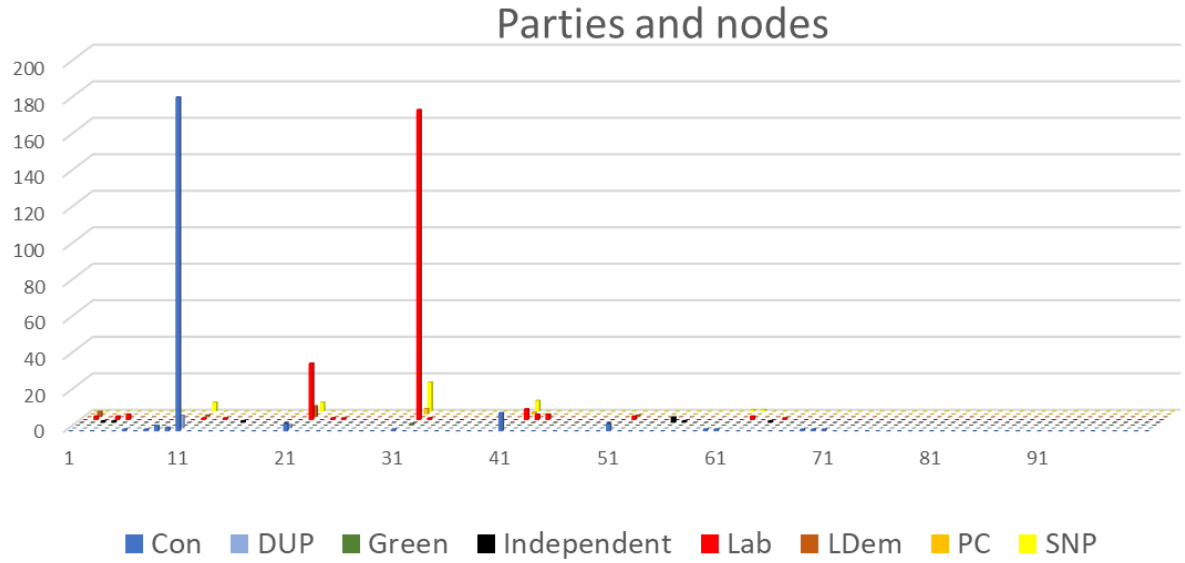
If we had used data for sock colour instead then replacing red, blue, yellow, striped, spotted,. . . with numbers would have been silly.

The results are shown below. The first figure shows that there are two main groupings, surprise surprise.



The next figure shows the breakdown of the nodes by party. Conservative and Labour are mainly in completely different areas of the grid. However Labour are a little bit more spread out.

Interestingly, there are members of the main parties scattered about, not being where you'd expect them to be. It would be interesting to look at those MPs who don't seem to be affiliated with the correct party and why. Note that nodes 11, 21, 31, etc. are all close together in the grid.



Obviously I have used this data for illustrative purposes. If this were being done for real I would take a lot more care, such as testing the results, and also probably break down the data according to things like the subject of the vote.

Summary

Please take away the following important ideas

- SOM groups together data points according to similarities in their feature vectors
- The weighting vectors are found by iteration
- The results are very visual, appearing on a grid