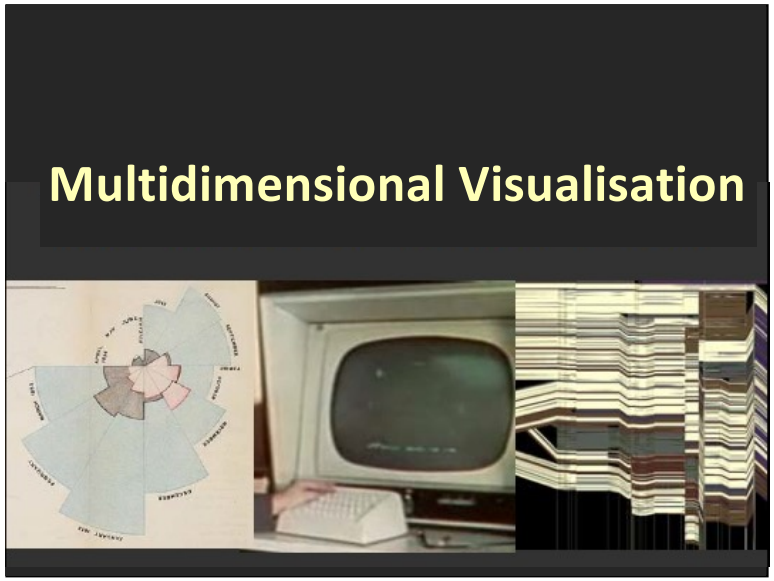
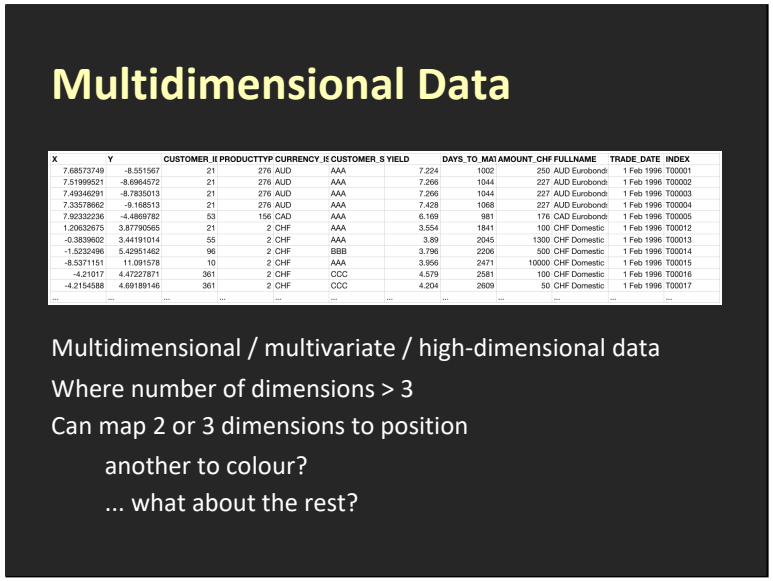


Multidimensional visualisation

I've chosen to have an entire lecture on this topic, as the earlier lectures don't really go far enough (I think) in handling complex data. We will look again at a few familiar simple examples, but also go on to some more advanced methods.

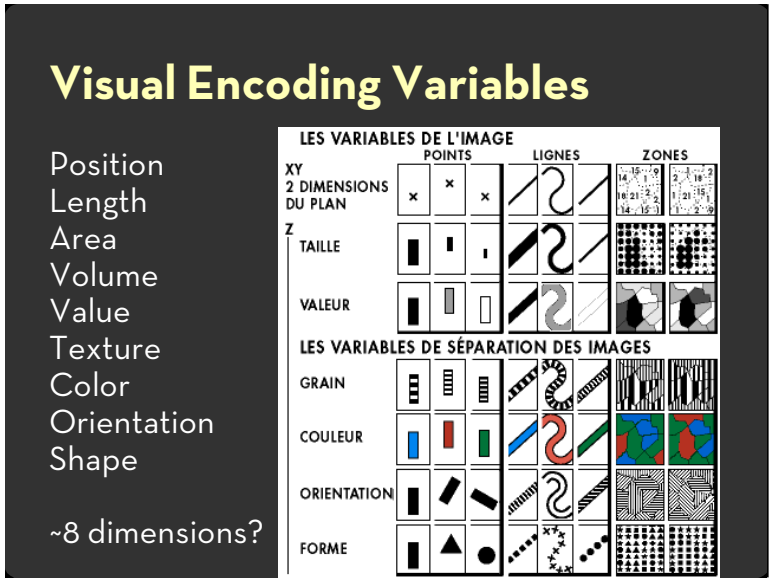


[image shows an example spreadsheet with many columns]

We are dealing with data that involves many columns, if one considers a spreadsheet or table, or attributes. Multivariate is another common name, but here we are calling them dimensions.

Mathematically, we often speak of dimensions on the assumption that these are independent or orthogonal features, so that each one does not depend on the other, and can be analysed separately. For example, if a bank had data on customers that had columns for the bank balance in Pounds Sterling as well as the bank balance in Euros, then – even though currency exchange rates might vary – there would be a strong relationship between those two columns. Those two 'dimensions' are not really independent and orthogonal… so, strictly speaking, we shouldn't call them dimensions… but in what follows we are more relaxed about the terminology. One reason for this is that it might only be through visualization and analysis that one finds whether two attributes are independent or orthogonal.

The issue then is that if we have a number of dimensions much greater than 3, how will we map those dimensions onto graphical features? We can map 2 or 3 dimensions on to position, and perhaps another to colour… but that might leave many many dimensions unmapped. What do we do about them?



[image shows Bertin's chart of visual encodings for data, such as position, length and area]

You might recall this chart of visual encodings from a few lectures back. If we consider the basic graphical variables we can use, there are roughly eight that we can use vaguely independently. So, we could start mapping our high-dimensional data onto these variables, as we talked about before. It might be that the most important variables might get mapped on to position, and then the next one on to the length of the symbols, then the next one on to area, and so on… In this way we might be able to map around 8 dimensions onto graphical features. Would that be a good design strategy?

Example: sales for a coffee shop chain

Sales: quantitative, sequential

Profit: quantitative, diverging

Marketing: quantitative, sequential

Product type: categorical (also called ‘nominal’ or N)

- *Coffee, Espresso, Herbal Tea, Tea*

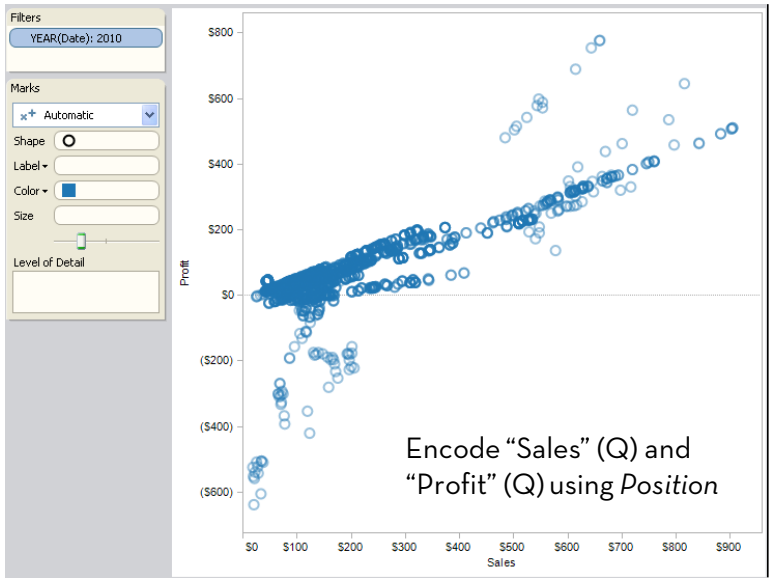
Market: categorical/nominal

- *Central, East, South, West*

Here is an example data set schema, about coffee sales figures for a fictional coffee chain.

We have three Q dimensions: sales, profit and marketing costs. We also have two categorical or nominal dimensions: the product type, naming products such as ‘coffee’ and ‘tea’, and the market, which is the name of the market segment or geographical area that a data point describes.

So, we will consider this a 5-dimensional data…

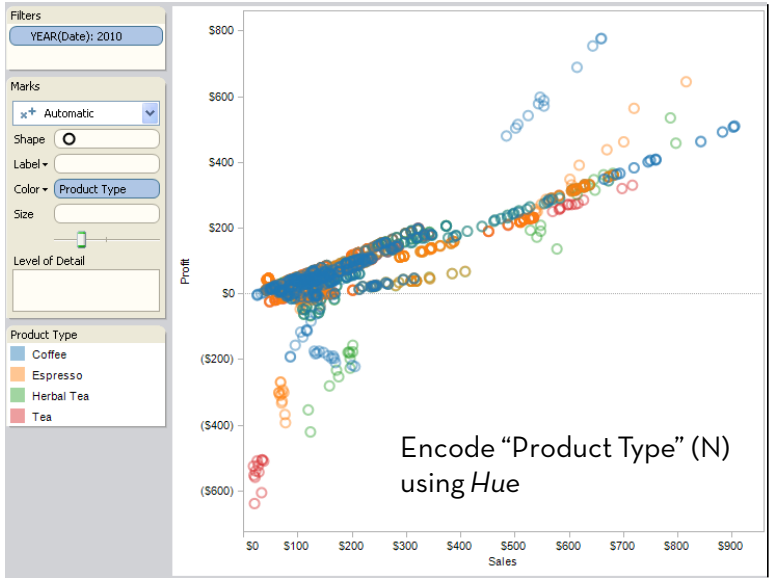


[image shows a scatterplot of sales against profit]

Perhaps we consider Sales and Profit as the most important variables for a particular analysis. We would then encode each of those using Jacques Bertin’s method, using position in x and y. At the moment, we just have a default mark, a blue circle, for every data point, but we can see some structure already. Note the cluster of points with low sales and low profit. There are some items which have strongly negative profit, but low sales… so that might not be too bad. We can also see some items with both high sales and high profit, laid out in sort of linear strands going up and across the top right of the chart.

So, this 2D encoding is pretty common, and pretty useful. We have quite a few dimensions to go, though.

(We won’t do 3D positions for such abstract data, as it’s almost always a bad idea. It makes seeing pattern and structure just too hard)

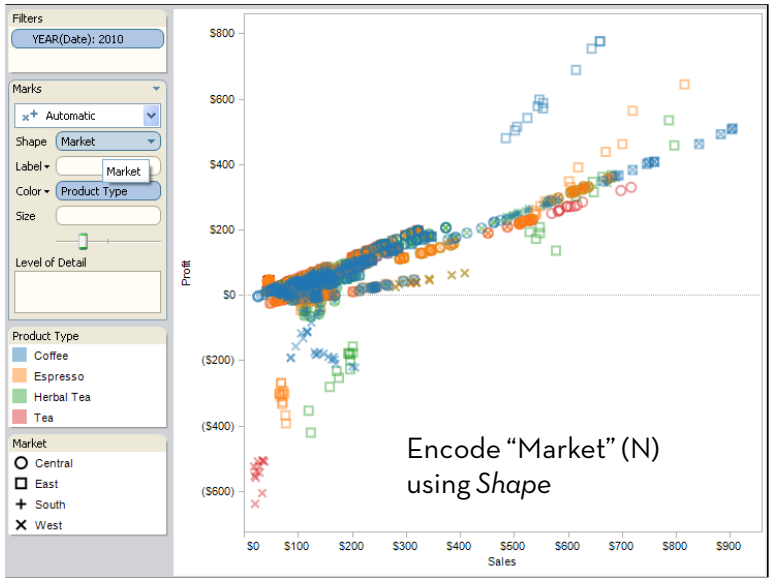


[image shows the same scatterplot as before, but with Product Type encoded using the colour hue]

This image shows one more encoding, with Product Type being mapped on to a small set of easily discriminated colour hues, as you may see from the left column of this tool. There are some problems, for example there is heavy overwriting of marks near the origin, which makes it hard to see details there, but this encoding does let us find some useful specifics about the data, for example that cluster of very unprofitable items seems to be all Teas.

So, this is is a visualization that encodes three dimensions in 2D space, by using x, y and hue.

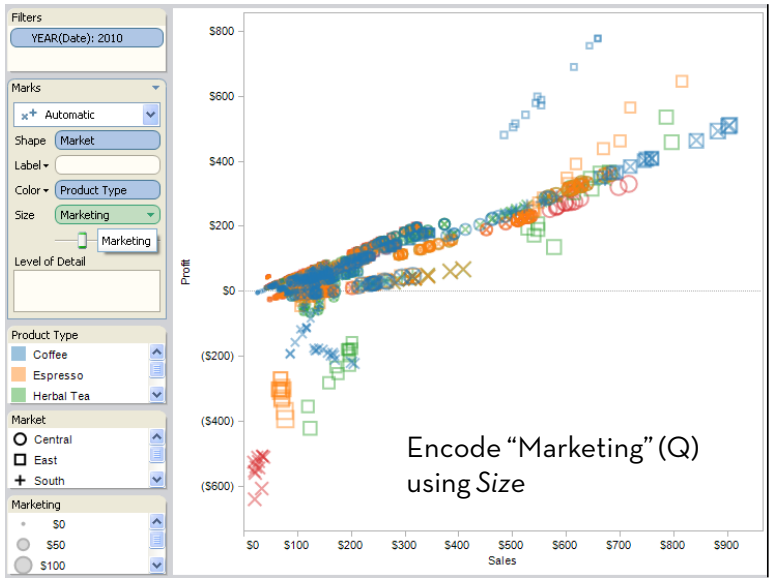
We could probably get some more details here, but recall that we don’t know much about how the sales vary across our sales regions, so the next slide is about that.



[image shows the same scatterplot as before, but with Market encoded using the shape of the mark, e.g. circle for Central and square for East]

The plot is still pretty comprehensible, or at least parts of it are. The highly unprofitable teas are all being sold in the West region, and so that might let the company focus its remedy there.

This is now showing four of our original dimensions, and it’s not too bad… but we are starting to find the details of individual marks overlapping or interfering with each other. The number of dimensions is still quite small, but already our visualization is getting messy, and hard to read.

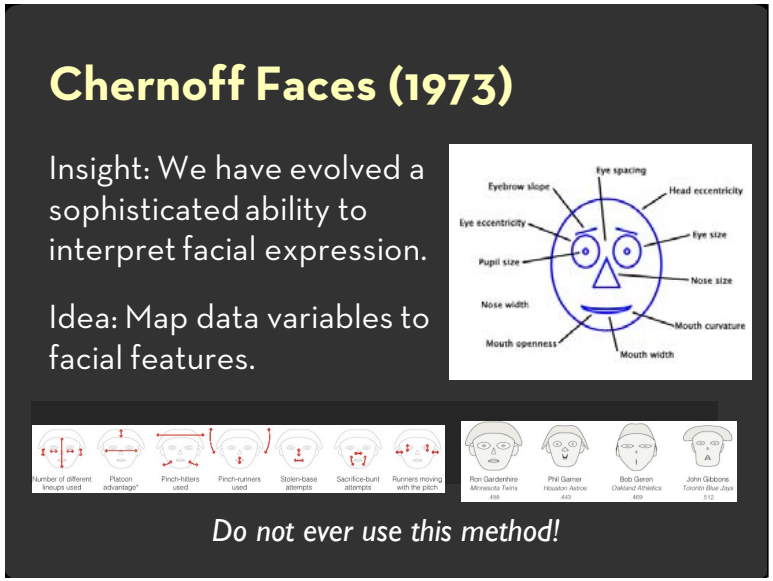


[image shows the same scatterplot as before, but with Marketing cost encoded using the size of the mark, e.g. a large circle for an item with a high marketing cost in Central]

Here we show the fifth dimension in our data by scaling the size of the mark in accordance with a scale you may see in the very bottom left of the chart. I am not sure if it's an exact mathematical relationship, e.g. is the area of a mark for $50 half that of a mark for $100, but the principle is clear.

This is just making the chart more cluttered, though. Bigger marks mean more overlap of marks, and (for example) the dense cluster near the origin, and perhaps also the cluster at the middle of the high sales part of the chart, are harder to read now.

The benefit or payoff of having more dimensions directly encoded is getting smaller. It's becoming harder and harder to read the chart, and get useful information out of it… and we are only handling five dimensions here – which is small compared to many data sets.



[Chernoff faces]

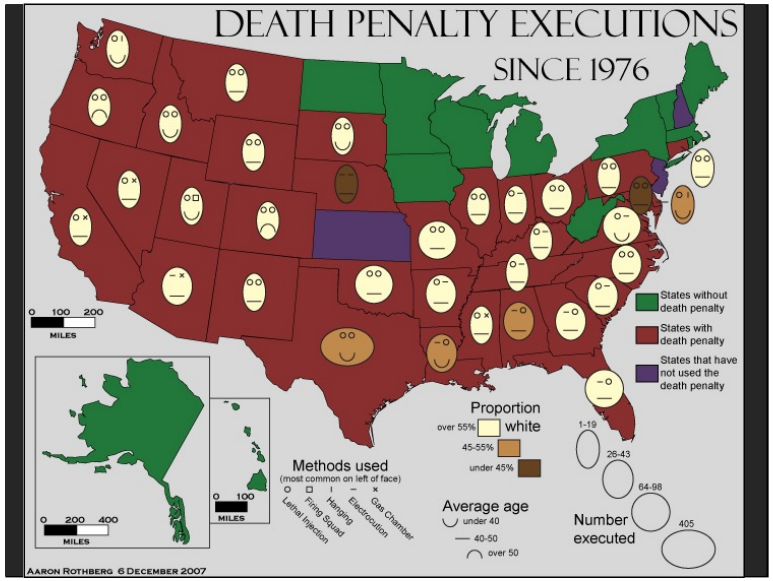
People have been trying to find ways to encode many variables, in a way that we can read.

One bad idea that was tried – and you still see this around some times – is Chernoff Faces. The idea was that, since we have evolved very sophisticated skills in interpreting human faces, a design could use that skill. We could perhaps map each variable onto a facial feature, such as size of the eye, the size of the pupil in the eye, the width of the mouth, the curvature of the mouth and so on. There are, potentially, lots of facial variables to map to. Could we therefore map our data variables to them, in an effective way?

The short answer is 'no'. Don't use them! Chernoff faces are known to be not effective at conveying multidimensional data. Firstly, we perceive faces in ways that mix or blend different attributes. We expect certain coordinations, ratios and patterns among facial attributes, and this biases how we interpret them. We don't interpret such features in anywhere near linear and independent ways. This inaccurate encoding is a weakness for Chernoff faces in general, i.e. for all applications. Don't use them.

Another issue is that there are important cultural meanings in faces, which includes significant variations in faces based on ethnic or genetic origins… which are often sensitive and political things. So, often, using faces is not appropriate to the topic being visualized or

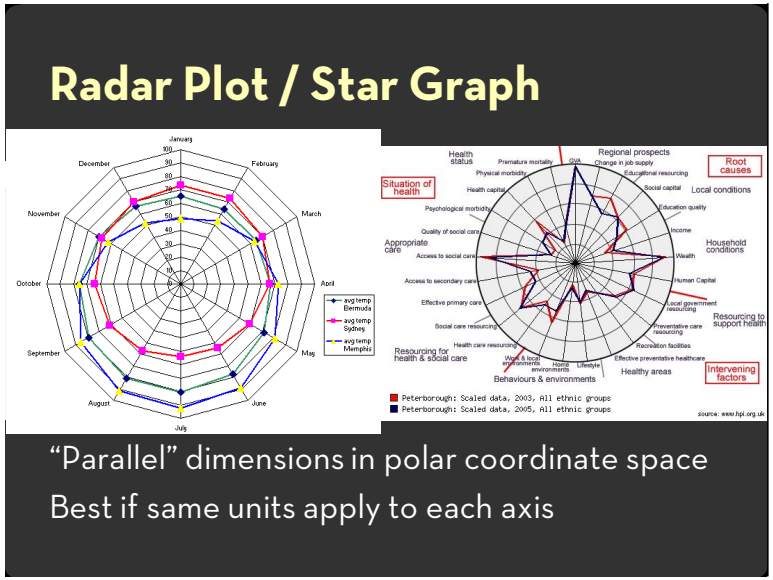
analysed. The next slide has a particularly bad example.



[image shows a chart made in 2007, using Chernoff Faces to show data on death penalty executions in the USA.]

There is a problem in the way that the data here reflects differences in the white and non-white populations, with regard to executions. It's a very sensitive topic, politically, and yet the designer has (for example) used mouth shape as curved up (like a smile) to show an average age under 40, and a frowning curve down for average age over 50. Each eye is encoded differently by the type of method. Small differences in the proportion of white people among the people executed make for strong differences in facial colour in the chart, which can be highly misleading.

The topic of the data clashes so much with the cartoon-like faces, and and so it was politically and culturally unwise to use faces in a chart like this. The designer should have used more abstract data encodings.



[Radar Plot or Star Graph]

This kind of graph is another fairly old way to encode many dimensions. The chart looks a little like a radar display, with lines or axes extending out from the centre, and then – rather like the gridlines in graph paper – circles of increasing size to show scale. The idea is that one treats each of these lines as an independent dimension, and one can make a mark on each such line to show the value of that dimension for a data item. It's then common to draw lines to connect the values for the same item, sometimes forming a sort of ring shape – for example the average temperature in Sydney for each of the 12 months is shown on the left chart in pink. It's easy to follow that line round the chart, and see how the temperature grows and shrinks. One can show a few different data items by using different colours, as you can see here too.

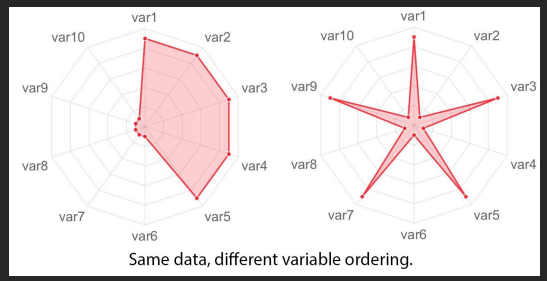
You have to be careful with the scale and units for each axis. This is to do with normalization, in that if one axis has very large values and another only has very small ones, then the details of the second axis will be compressed and hard to read. Also, it's best if the units are actually the same, like a count for every axis, or a price for every axis, as it's really hard to compare values when the units vary on different axes here.

On the right is a chart with even more dimensions – over 30, I think. Note that the chart here is already starting to look cluttered, with only two data items on it. The idea, though, is that one can quickly compare items, by seeing where the lines or shapes for them vary. It is true

that one can make this comparison, but note also that it gets harder with more dimensions… but really the biggest drawback or weakness is that it is hard to see detail among many data items, as they overwrite each other. The ‘D’ for dimensions can be pretty high, but the N for number of items has to be low.

An alternative name for this kind of chart is Star Graph, as the lines for a data item often makes a sort of jagged shape, going in and out as it goes around the circle… slightly like a star symbol.

Radar Plot / Star Graph



Same data, different variable ordering.

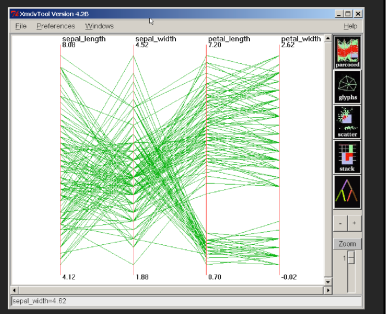
You have to be careful with radar plots, as the order of dimensions affects the interpretation a lot. For example, note how the visualisation the left conveys a different overall message about the magnitude of the 10-d data object than the visualisation on the right. Vary the order to explore the data, if using this method.

[Another radar plot example]

I also want to highlight an issue with the order of the axes or dimensions around the circle. Depending on the order, you might get very different charts, and give a very different impression of the data. This example shows two charts of the same 10D data item. In one, all the high values are nearby, and so we get a solid looking wedge shape in that sector… with a large area… whereas if we rearrange the dimensions, we get the very skinny starfish shape on the right… with very low area. Area is a strongly perceived feature, and you have to be careful with it. It's best to vary the order of dimensions so as to explore the data, so as to avoid this problem.

Parallel Coordinates

- Non-orthogonal display of dimensions
- Inselberg, 1985
- Each object a single polygonal line
  - Intersects each 'axis' at appropriate value
- See patterns, clusters...
- Good for showing correlations between dimensions, if adjacent
  - Might need to re-arrange dimensions to find this, though



[parallel coordinates]

Al Inselberg, of IBM, created a different kind of chart, rather like the radar plot. Here, though, each dimension is laid down as a different line, and the lines are parallel – usually oriented vertically.

Each data item or object then refers to a point on each line, and we can connect them together with a single polygonal line. The line intersects each axis at the value appropriate to the object.

The lines build up, and one can see clusters where many items have similar values for particular dimensions, and also features such as correlations among adjacent axes – where the lines all have the same kind of slope, for example, meaning that there is a simple linear relationship between those two dimensions.

The problem is that this only works for adjacent dimensions, and so one might need to re-arrange dimensions to find such patterns and clusters.

Parallel Coordinates

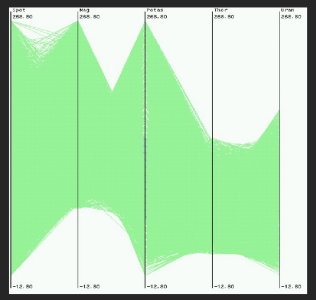
Hard to follow a single object's line left to right

Problem gets worse with bigger data sets

Interactive controls can help

- Mouse-over to highlight a single line

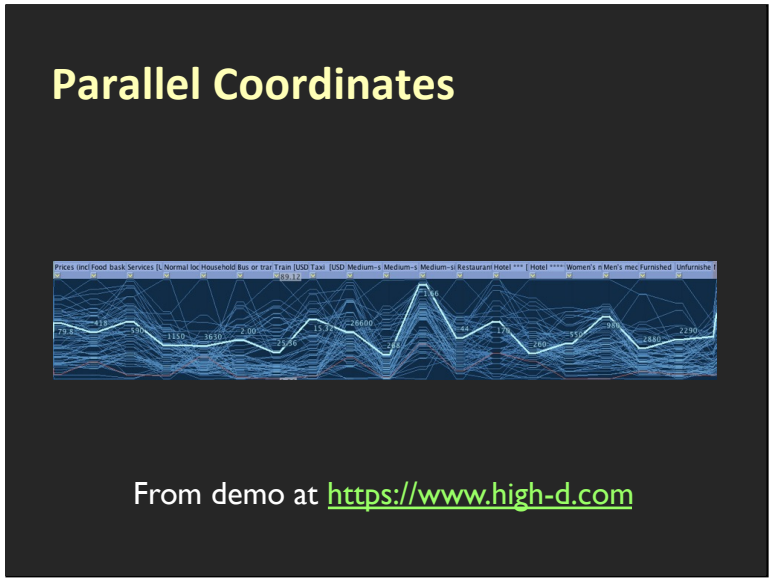
If data set it large and/or complex then you may need to sample or cluster it, and allow the user to drive and see that process



Parallel coordinates, continued

With large data sets, it becomes hard to follow where a single line goes, because of over-writing of lines. One can mouse over a line, to highlight it – as in the next slide -- but this isn't a good strategy if there are many many items.

If N really is high, then you may have to sample or cluster it, and then just show the samples… or the cluster centroids… so that the chart simply has many fewer items in it. Make this something that the user can drive, so it's more comprehensible.



Parallel coordinates, continued

This is an example from a company called Macrofocus, started by two guys who used to work for/with me. You can get some free-for-30 days software from them at https://www.high-d.com. They did a really nice job of colouring, and also of the fine-grained interactions with the components.

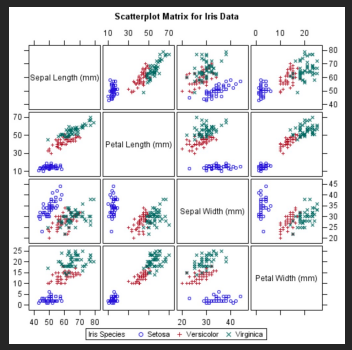
Anyway, the issue here is just to show that a simple mouseover can be used to highlight a line – a data object – and trigger the system to show more detail such as the numerical values for each dimension.

Scatterplot Matrix (SPLOM)

x-y scatterplots of every pair of dimensions

Good for seeing correlations in *pairs* of dimensions

...however they're positioned, unlike radar plots & parallel coords



The scatterplot is a more traditional way of showing multiple dimensions. This stems from statistical graphics, and especially printed forms of statistical graphics.

The example data here is the Iris data, that is so often used in simple analytics examples.

The idea here is to have a scatterplot for each pair of dimensions, so you get a symmetric rectangular matrix of little scatterplots. In this way, it is an example of the 'small multiples' design style. it is good for seeing which pairs of dimensions lead to similar charts, i.e. if or where there are correlations between the pairs.

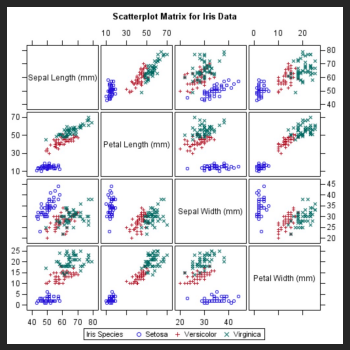
Scatterplot Matrix (SPLOM)

Screen space requirement rises quadratically with dimensionality *d*

($d^2 - d$) plots

Duplication in grid of graphs → just show 'triangle' on one side of diagonal

e.g., the 3, 2 and 1 graphs top right



Note that screen space runs out quickly, as the number of dimensions grows… because the number of charts grows quadratically with the number of dimensions. With large N, each chart is small and there are just so many of them… and so it is hard to really understand patterns that extend across more than a few dimensions.

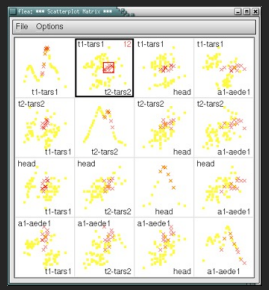
One small trick that does help is to skip the duplicate graphs, either side of the diagonal. Clearly, a chart of A against B is going to be much the same as a chart of B against A, and so sometimes you will see not a rectangle of charts, but a triangle. This still grows quadratically, though, so it only helps a little.

Scatterplot Matrix (SPLOM)

Interaction can make a visualization much more powerful and insightful

Brushing and Linking is the most obvious method here, but there are many more

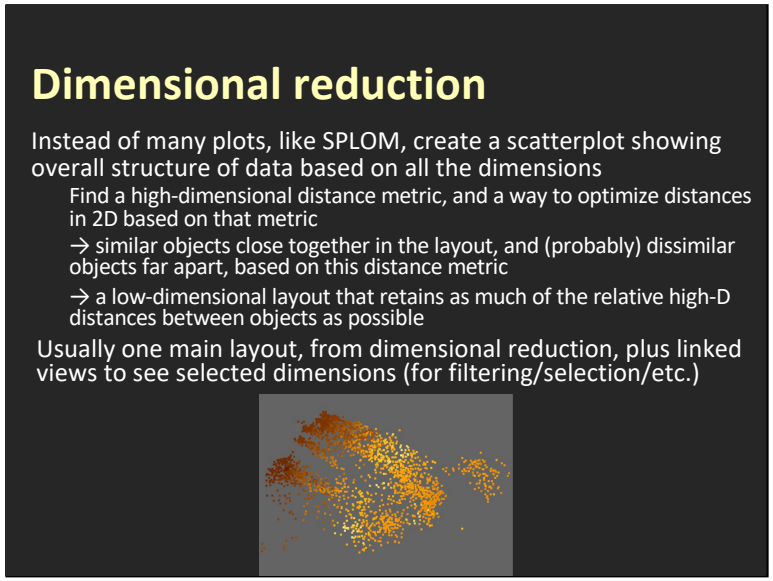
SPLOM *without* such a method is a pretty weak design



Scatterplot matrix continued

Using brushing to highlight a subset of the data in one chart or view, and then linking that selection to the other views — so that they are highlighted there too — is a really powerful technique. It really helps with scatterplot matrices, but (as you know already) is good for linking all kinds of visualisations. In fact, I'd say that scatterplot matrics without a method such as brushing and linking is a pretty bad design, for an interactive system.

One issue here is that most textbooks don't show these interactive methods, as… they are printed on paper! It's hard to explain dynamic interaction methods well in print, so often authors don't cover them in depth.

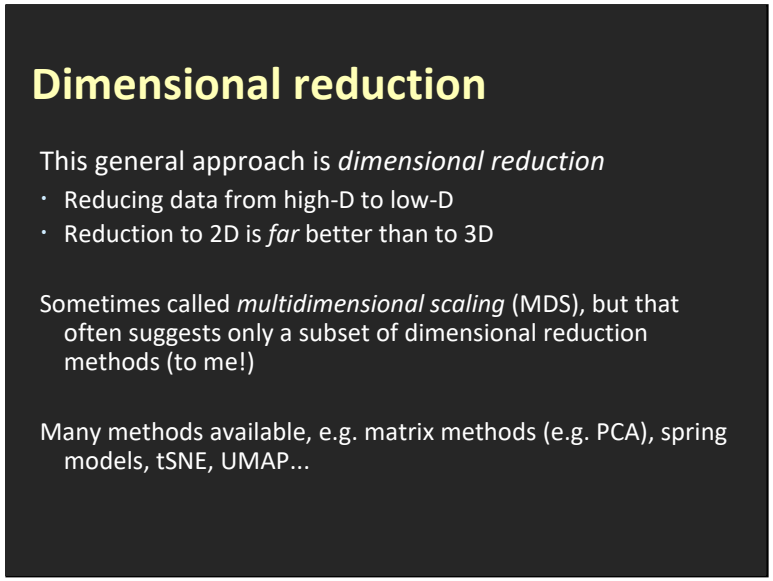


Dimensional reduction

The last few slides showed ways to combine multiple small scatterplots, but it is common to use a single 2D scatterplot to show high-dimensional data. It lets us get away from the problem of techniques (such as SPLOM and Parallel Coordinates) that do not scale to a large number of dimensions. We can then fit a visualization into a reasonable amount of screen space.

There are many ways to do this, and we will look at some of them. Generally, they work on the basis of some metric that defines the distance between your high-dimensional data objects. This is used in some form of optimization process, that will try to lay out objects in 2D, in positions such that the 2D distances approximate these high-dimensional distances. In other words, we can move and position objects that are similar, to be close together in the 2D layout. We can move objects that are dissimilar in high-D space, to be far apart in the 2D space.

The results will be different, depending on the metric you use, and the optimization method you use, but the overall approach is basically the same.

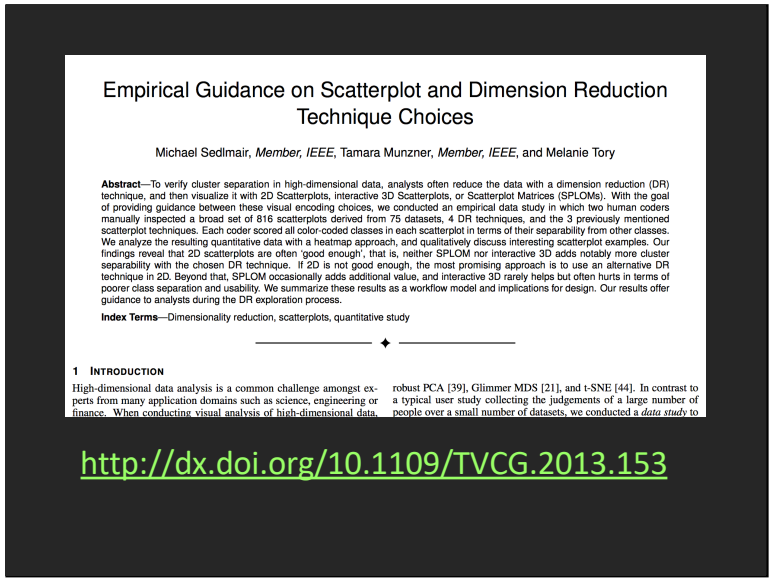


Dimensional reduction

The mathematical principle or pattern here is dimensional reduction, because we are taking some high-dimensional data and reducing it to a low-dimensional representation – to a low-dimensional visualization… usually 2D (because 3D is so hard to use or gain anything useful from).

It is sometimes called multidimensional scaling (MDS), although to me that usually means a subset of these methods, like PCA. (We'll get to PCA soon.)

There is a long history of doing this, originally using matrix methods, which tend to assume linear relationships between dimensions (again, like PCA). These go back far into the last century -- to the year 1901, in fact. However, over the past 20-30 years, there have been many new (and often better) methods. We'll look at a few of them in this lecture.



Empirical Guidance on Scatterplot and Dimension Reduction Technique Choices

Michael Sedlmair, *Member, IEEE*, Tamara Munzner, *Member, IEEE*, and Melanie Tory

Abstract—To verify cluster separation in high-dimensional data, analysts often reduce the data with a dimension reduction (DR) technique, and then visualize it with 2D Scatterplots, interactive 3D Scatterplots, or Scatterplot Matrices (SPLOMs). With the goal of providing guidance between these visual encoding choices, we conducted an empirical data study in which two human coders manually inspected a broad set of 816 scatterplots derived from 75 datasets, 4 DR techniques, and the 3 previously mentioned scatterplot techniques. Each coder scored all color-coded classes in each scatterplot in terms of their separability from other classes. We analyze the resulting quantitative data with a heatmap approach, and qualitatively discuss interesting scatterplot examples. Our findings reveal that 2D scatterplots are often 'good enough', that is, neither SPLOM nor interactive 3D adds notably more cluster separability with the chosen DR technique. If 2D is not good enough, the most promising approach is to use an alternative DR technique in 2D. Beyond that, SPLOM occasionally adds additional value, and interactive 3D rarely helps but often hurts in terms of poorer class separation and usability. We summarize these results as a workflow model and implications for design. Our results offer guidance to analysts during the DR exploration process.

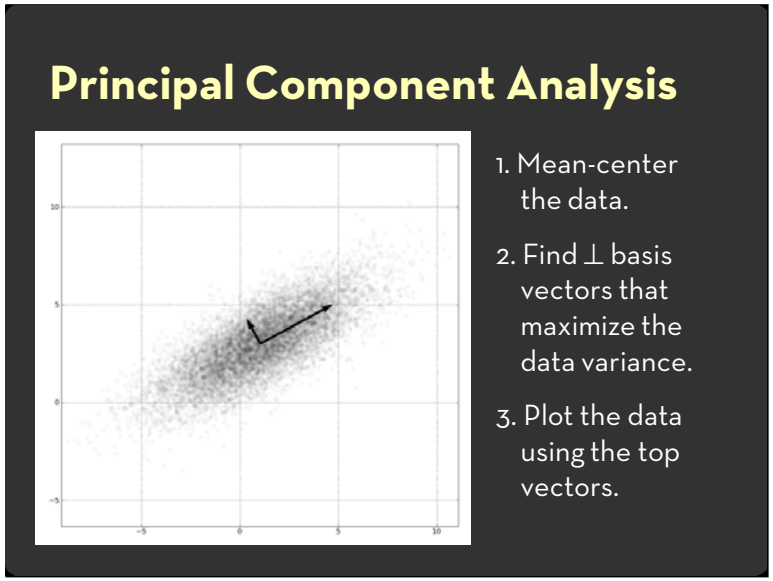
Index Terms—Dimensionality reduction, scatterplots, quantitative study

1 INTRODUCTION

High-dimensional data analysis is a common challenge amongst experts from many application domains such as science, engineering or finance. When conducting visual analysis of high-dimensional data, robust PCA [39], Glimmer MDS [21], and t-SNE [44]. In contrast to a typical user study collecting the judgements of a large number of people over a small number of datasets, we conducted a *data study* to

http://dx.doi.org/10.1109/TVCG.2013.153

In case you are not convinced by me saying how bad 3D is… you might be interested in reading a paper that backs up experimentally what a lot of us within the field have felt for a long time… that 3D plots are generally not useful, and also that SPLOMs are not really so good either (especially with larger numbers of dimensions). Using 2D scatterplots, which may be the result of dimensional reduction algorithms, seem to be the best choice. If you're interested, follow the link above…



Older approaches to dimensional reduction treat it as a matrix problem. They work with the dimensions using matrix mathematics (such as eigenvector analysis and the like). We won't go into the mathematical details of this, but let me just sketch one approach out.

Principal component analysis (PCA) is perhaps the best known example of dimension-centred MDS approach. Developed in 1901 by Karl Pearson, it uses matrix algebra to find an ordered sequence of perpendicular (orthogonal) *basis vectors* that are likely to be good for describing the data set, in that these maximise the variance along each of these vectors in turn. These are also called 'principal components', which gives the name to the overall algorithm. We call it a *linear* dimensional reduction algorithm, as the output dimensions are a linear combination of the input dimensions.

We've a 'toy' example here, in that we are going to analyse 2D data… and of course this technique is for higher dimensional data… but this is just to make it easier to see how PCA works. The first step is just to centre the data set on its mean, or centre of mass. Then, the method looks for the vectors that describe the variation around this mean most. You can see here that the first vector is the one going up and right in this chart. It's the first because it is the direction or axis along which the data is most spread out… In other words, the variation away from the mean or centre of the data is highest. In fact, the variance is shown here (I think), by the length of the arrow going up and right from the centre of the chart.

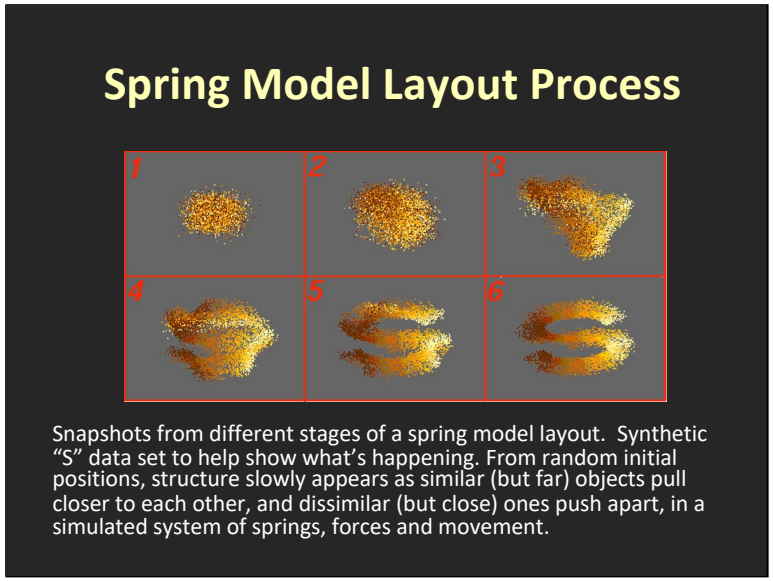
You could stop right then, and use just one basis vector if that's all you want, and effectively

squash or project the points down onto this one axis. You would be reducing the dimensionality to just one dimension. Then, you could make a histogram of the points along that axis. It would show lots of points in the middle, and gradually fewer to the left and right.

However, normally you'd go for more than one basis vector. Each new vector has to be at right angles (orthogonal) to all previous ones, and here it's easy to see that it's the vector that goes up and left, from the centre. The diagram shows the arrow as shorter, as the data set here has less spread or variance in that direction. You could then use the two vectors to graph your data, and with this simple example you'd see that the data is now aligned with the graph axes, with the overall 'blob' of points here aligned with the graph axes.

The example input data set here is just 2D, and so you can only get 2 principal components, but with high-dimensional input data (e.g. 10, 20, 100D, etc.) then you can choose two (or maybe three) principal components to use for your graph or visualisation. Of course, you will lose detail, as you are squashing the data onto your chosen axes, but that is generally unavoidable. Normally, you can't keep *all* the relative distances, separations and patterns in the high dimensional data set, when you do this kind of dimensional reduction. Instead, you try to show some important patterns and relationships.

Briefly, the big assumption of this kind of dimensional reduction technique is that the significant patterns and structures in the data set go along well with the dimensions, e.g. high salary for a footballer is well correlated with high numbers of goals in top football matches, so we can combine those dimensions into one. Often this is not right, though! Patterns and structures in data often don't simply follow big clear patterns along the axes. Unfortunately, you may have to explore the data quite a lot in order to find out whether this is true… although you should be doing that kind of assessment, anyway. If the patterns are more complex, it may be that non-linear algorithms (that we will look at next) are better, such as spring models and UMAP.



Let's now turn to non-linear models. I am going to focus mostly on spring models, as I know them best. These are also often called force-based models (and other names). They (and similar algorithms) are non-linear in that the output dimensions (usually the 2 dimensions used to visualize the data) may not be a linear combination of the input dimensions.

A key feature here is that this is an iterative approach. Each iteration of this kind of algorithm tries to move the objects so that the layout is a little better. The spring model pushes and pulls the objects into better positions, in accordance with the high-dimensional similarity metric that describes the 'ideal' distance between each pair of objects. It is trying to make the low-dimensional distances (in the layout) be the same as the high-dimensional distances.

The six images here are of an example data set sampled from the shape of a letter 'S', and the layout algorithm will (hopefully) reconstruct the letter shape. Image (1) shows the algorithm starting with random positions for the objects. This random start is common, as usually we don't have any information that would let us make a better set of starting positions.

Then, images 2-6 show the layout after more iterations. I suspect that there were quite a few iterations between 1 and 2, and 2 and 3, and so on. It didn't make the layout of 6 in six iterations! It takes longer than that.

Nevertheless, gradually, the overall structure in the data appears in 2D.

Spring models

Simulate a spring between each pair of objects

- Ideal relaxed length of spring proportional to high-D distance between objects

Example: three rows from spreadsheet: call them A, B and C

- Objects A&B quite similar; C quite different to A and to B
- So, AB is a short spring; AC and BC longer springs

X	Y	CUSTOMER_ID	PRODUCTTYPE	CURRENCY_IS	CUSTOMER_SEG	YIELD	DAYS_TO_MATI	AMOUNT_CHF(	FULLNAME
7.68573749	-8.551567	21	276	AUD	AAA	7.224	1002	250	AUD Eurobonds
7.51999521	-8.6964572	21	276	AUD	AAA	7.266	1044	227	AUD Eurobonds
-4.2154588	4.69189146	361	2	CHF	CCC	4.204	2609	50	CHF Domestic E

Spring models

These models simulate springs between objects, to push and pull on them, so as to get them into good positions.

The ideal length of a spring is then proportional to the high-D distance between objects. An example here shows three spreadsheet rows, we will call them objects A, B and C. You may see that A and B are similar, but C if different to A and B. So, we'd model this with AB as a short spring, and AC and BC as longer springs.

Spring models

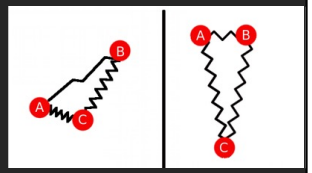
AB is short; AC & BC longer

Start from random positions (left image)

- Some springs too stretched, others too squashed

The spring forces then iteratively push and pull objects until the forces reduce

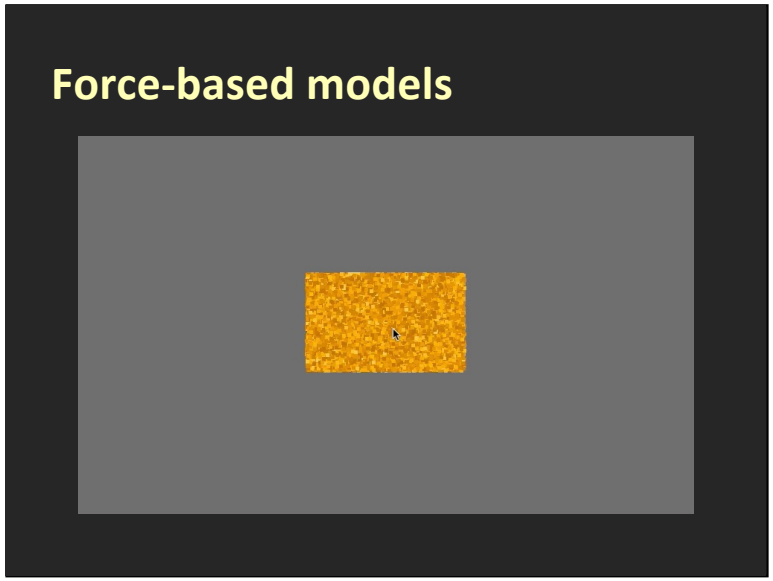
- i.e., the layout process reaches equilibrium (right image)



Springmodels continued

With these three springs, we have a problem. Where to initially place them in the simulation? If we knew how to place them so that they had their ideal distances, then we wouldn't need to do the simulation. Instead, generally, we start from random positions… and then some springs may be too stretched, while others may be too squashed or compressed… when compared to the ideal distances we want to achieve in our final layout, i.e. to represent the high-dimensional similarities.

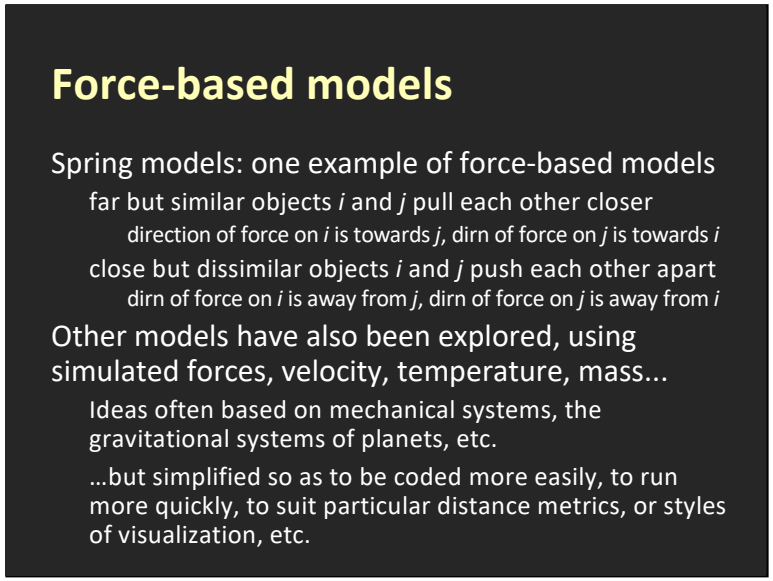
So, the simulation models the forces of springs being too long or too short, and pushes and pulls until the layout reaches equilibrium. The hope is that, then, a compromise has been reached between the inter-object tensions, in a satisfactory layout that shows the relationships between the objects.



Force-based models continued

This is an iterative process, as the video here shows. It can be hard to know how big or small an area to start with, so – as you may see here – the initial layout may mean a lot of potential energy in springs that are too stretched or too compressed. This injects a lot of energy into the system, and so it may seem like it is just exploding at the start… This has to be damped down to keep the system stable… you stop excessive forces and velocities… and this helps the process settle into a good layout. Eventually, though, the simulation will stabilize… and clusters and other patterns to appear.

This example video is bond trade data, from Union Bank of Switzerland. We can have a look at this layout process in an example program, called fsmvis (for 'fast spring model visualization').



Force-based models

I've described spring models, which are one kind of simulated system of objects with forces, position, velocity, forces, mass, and so on...

In general, far but similar objects *i* and *j* pull each other closer,

So, the direction of force on *i* is towards *j*, direction of force on *j* is towards *I*

And… close but dissimilar objects *i* and *j* push each other apart

Direction of force on *i* is away from *j*, direction of force on *j* is away from *i*

The algorithm design

process here very flexible. There are lots of ways to vary this general approach, using different simulations of mechanical spring systems, the gravitational systems of planets, etc. — but simplified, so as to be coded more easily and to run more quickly.

Of course, as a programmer

you can change the rules of the simulation. You can adjust those models, for example to base them on electric charges instead of mechanical springs, to gain slightly different effects, or to just use extra information that you can work out during the process, or to suit particular distance metrics or styles of visualization (for example, to make clusters more apparent, or

to stop very similar objects being right on top of each other.

A simple spring model algorithm

```
Initialise 2D positions randomly
for each iteration // run until 'enough' work has been done
        for each object i {
                totalForce = [0,0];  // a 2D vector used to accumulate forces
                for each other object j
                        totalForce += force(i, j);  // force() method - see next slide

                // ts: a 'time step' constant, that says how strongly to bring
                // in the newly calculated forces.
                velocity(i) += ts * totalForce;   // apply acceleration
                position(i) += ts * velocity(i);
        }
```

Let's look in more detail at a simple spring model algorithm, based on what we've said so far.

It initializes the 2D positions randomly, and then iteratively adjusts the system until either 'enough' iterations have been done. (We will look how to use layout error to judge when 'enough' has been done, in a later slide.)

Each object accumulates forces from all the other objects, and then a proportion of that forces is used to change its velocity… and then a proportion of that velocity is used to change its position. In this way you can manage how quickly the system responds to errors in distances, and how likely it is to become unstable and explode.

You need to dampen it, at least a little, to stop it from exploding… this is what the time step constant is doing here. Often, one has slightly more complex methods to manage and control the system, but this is a simple example.

Note also a simple speedup that could be done here. In the code given here, when we look at object ii (e.g. 7) and compare it to another object jj (e.g. 25), we add on the force on ii based on jj… but when ii increases and eventually becomes (for example) 25, there will be a loop over all other objects, including the (for example) object 7. Then, the equal sized force (with the opposite direction) will be calculated and accumulated. A simple shortcut would be to do the force calculation once instead of twice, when ii < jj, and then add it on to the total force

for ii, and then add the negative of that same force on to jj. Overall, this will reduce the force calculations by roughly half.

I didn't want to put this or other speedups into the slide here, as I wanted to show a really simple version of the algorithm first. There are lots and lots of tricks and speedups like this one can use, as I discuss briefly in slide 34 and 35.

Similarity Metrics

The metric is vital to the *force(i,j)* method

| force(i, j) | $\propto$ *| idealDistance(i,j) - currentLayoutDistance(i,j) |*

But what is the 'ideal' (high-dimensional) distance? It depends!

A simple example: treat spreadsheet as matrix of data M_{ic}

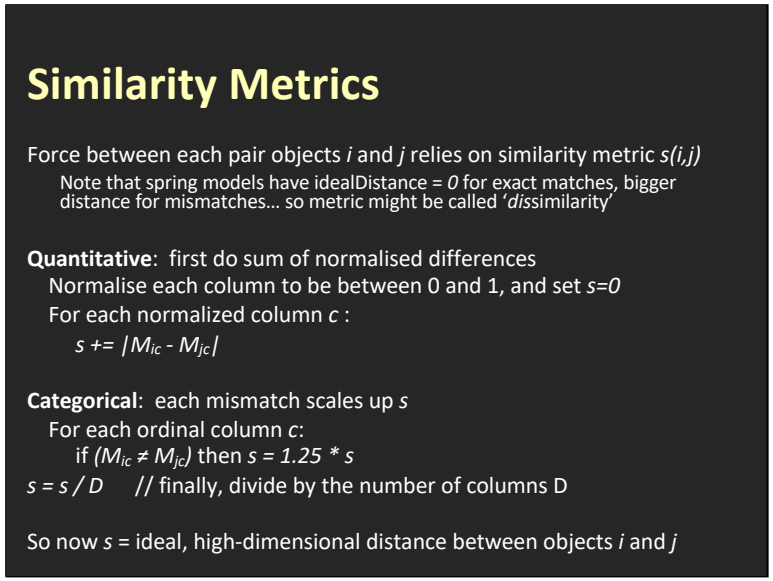
- Each row *i* is an object, e.g. a bond trade, and each column *c* is the values for one data dimension, e.g. *yield*
- Similarity of two objects M_i and M_j uses a combination of similarities for each column
- Handle numerical columns (reals, integers…) and categorical columns (strings, names…) separately, using simpler metrics

X	Y	CUSTOMER_ID	PRODUCTTYPE	CURRENCY_IS	CUSTOMER_SEG	YIELD	DAYS_TO_MATI	AMOUNT_CHF(	FULLNAME
7.68573749	-8.551567	21	276	AUD	AAA	7.224	1002	250	AUD Eurobonds
7.51999521	-8.6964572	21	276	AUD	AAA	7.266	1044	227	AUD Eurobonds
-4.2154588	4.69189146	361	2	CHF	CCC	4.204	2609	50	CHF Domestic E

Similarity metrics are key here, as in most multidimensional visualization methods. How you define similarity or distance can make for very different results.

Here, in force models, forces are calculated on the basis of trying to make all the current layout distances approximate these high-dimensional distances. However, what is this metric? It depends on the semantics of the data, really, and the end user requirements of the system.

An example general-purpose metric for spreadsheets is given here, and the next slide – it was used in the movie and demo. It combines similarities for each dimension or column, treating quantitative values using a simple metric for such data (like difference) and then using a metric for categorical data (ordinals and nominals) such as string matching. Remember, though, lots of metrics are possible. This one is just a simple one, for a demo.



Similarity metrics continued

Here is pseudocode for that metric. We want to have an ideal distance of zero for exact matches, and a bigger number for mismatches… so this might be more accurately called a dissimilarity metric.

For each numeric dimension, we can average the sum of normalized differences. We do this normalization so that columns with large values are scaled down to not wipe out the effect of columns with small values. Look at *Yield* and *DaysToMaturity*, for example. The values, and the differences in values, in those two columns are very different… and they should not be. Once we have these columns normalized, we can add up the differences from them, in our variable *s*.

For each non-quantitative dimension, we increase *s*… perhaps simply by multiplying it by the same factor each time we get a mismatch. Here we use 1.25, for example.

Doing this for all dimensions builds up our metric – an 'ideal' high-dimensional distance between two objects, that we can try to replicate in (for example) a spring model layout. The spring model will move objects, to find 2D distances that approximate these high-dimensional distances.

Layout Error

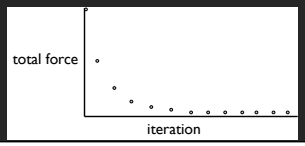
We can measure the current 'error' in the layout: how far from 'perfection' is the current iteration? How many iterations before we stop?

With each iteration, forces act to decrease layout's error

- Spring equilibrium is a compromise: not all relationships *perfectly* represented
- Error might never be zero, though it should get lower and lower
- We can watch for when it is stable (and low), and then stop iteration
- Exact error ('stress') is expensive to calculate
- Cheaper to use force as a proxy: sum over all pairs *i,j* of |*force(i,j)*|

Model may get stuck in a local minimum rather than ideal global minimum

- Can do multiple layouts, and choose the layout with the lowest error
- And/or add stochastic 'jitter' or 'heat' to help shake free of such minima



Layout error

I mentioned before that these models are run until they stabilize or are complete… but how do we know that?

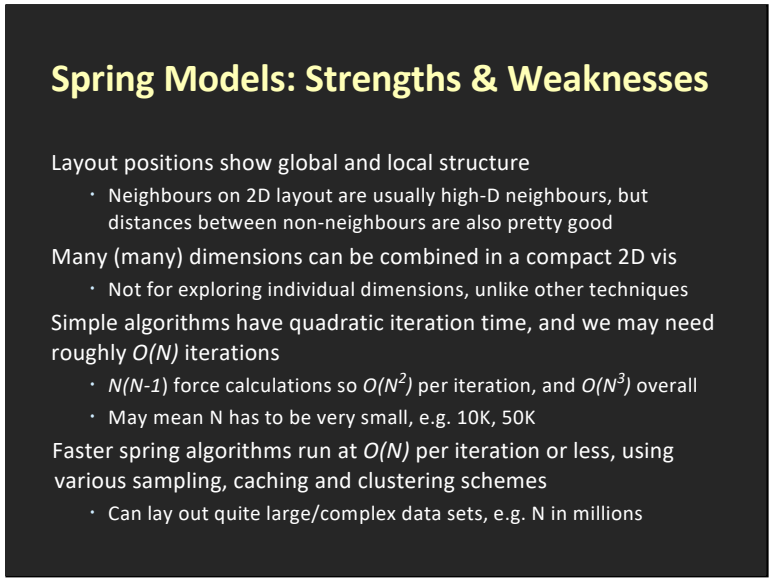
Too often, people just make up a number, like 1000 iterations, and just throw it in. That might be too low, in that the layout hasn't settled yet, or it might be too high, so we are wasting computation on iterations that don't do any more useful work. The best thing would be to adaptively run until the error was low.

We know the ideal distances, and so we can continually calculate the total sum of all the errors… There is a 'proper' way to do this, using the sum of squares of errors (in a metric called stress) but unfortunately it takes O(N^2) time to calculate. When you have big data sets, this is simply too much. (It might take longer than the spring model!).

A cheap way to do this adaptive termination is to use the forces in the spring model. As the total magnitude of forces comes down and flattens out, as in the little diagram at the bottom here, then we assume that the layout is settling into a good shape.

It is pretty likely that we will never get every force(i.j) to be zero, and so the total force to zero. However, usually, the total force will decrease until it stabilises at some low value.

It’s quite possible that this may not be *the* optimum. The layout might get stuck in a configuration that is not optimal. Objects are moving to better positions, but they get pushed back by other objects. This is always a potential problem, but there are some things one can do. First, one can just run the algorithm multiple times, and look for the one with the lowest error. This can be expensive, though. Another approach is again based on the idea of modelling physical forces and objects… to add a stochastic randomness, a bit like the motion of hot gas particles, and that randomness or ‘jitter’ can shake the system out of local minima.



I've used spring models to explain the more general method of dimensional reduction, but they are not the only method. Spring models have their own strengths and weaknesses, compared to other methods.

Firstly, a strength: the layouts show both global structure – big patterns like where clusters and outliers are all make sense – as well as local structure, such as the most similar objects in high-dimensional space being close to each other. Usually, objects that are neighbours in the 2D layout are very similar in high-D. Also, objects that are dissimilar in high-D are roughly the right distance apart. Of course, this all depends on relationships among all the objects.

Another strength compared to SPLOMs and the like… we can take very high dimensional data, and look at it in a manageable way in a compact 2D space, on screen. Often, you'll want to have other visualisations or other tools to look at individual dimensions, though, as we saw in fsmvis.

A weakness of the basic spring model is that it is quadratic time per iteration – because we compare each of N objects with N-1 other objects. It's hard to know how many iterations to use, but it does grow with bigger data sets, so let's say it grows with N… so that makes the algorithm $O(N^3)$ overall – cubic, not quadratic. This means that the data set can't be very large. The demos you might see on the web are usually very small, like 10K or 50K objects.

Lots of people have worked on faster algorithms, as this kind of calculation or optimization is used in many areas of science and engineering. Older methods used hierarchical subdivision of the data set. I'll give an example of that in the next slide. Alistair and I worked on a different approach, based on sampling and caching. We got down from $O(N^3)$ to $O(N^{5/4})$, which helped a lot.

Hierarchical Methods for Force Models

Classic work on this is the *Barnes-Hut approximation*, from 1986

- Divide up the layout space with a binary tree of cells. Each branch combines all the objects lower in the tree. Positioned is centre of mass.
- One can approximate many long-range forces, by one force based on a branch object, to bring force calculation down from $O(N^2)$ to $O(N.logN)$

For iteration, for each object *i*, find the force from all other objects

- For each nearby cell, do exact force calculations for every other object in those cells
- For each distant cell, do one force calculation using branch node's centre of mass

Note that the tree has to be rebuilt, after each iteration… which can be a substantial cost. (This is where sampling-based approaches win out.)

Awesome demo from Jeffrey Heer: https://jheer.github.io/barnes-hut/

Too often, when you look for existing code for spring models, and other force-based models, it uses that very simple method that (in each iteration) compares every object with every other object… and so it quadratic in time. This does not scale well.

Many hierarchical methods have been used, for many years, to improve this, for all kinds of dimensional reduction algorithms. If you can understand the approach I'll describe now – Barnes-Hut, named after two astronomers who did this work around 1986, then you will have a basic understanding of all of them.

The core idea is that if objects are very far away, then maybe we don't need to do exact force calculations on every single one of them. If we divide the layout space hierarchically, in a binary tree of spatial cells, then we can build a model of what objects are in each cell. Each branch in this tree represents the leaf objects below it. It's like a larger object, positioned at the centre of mass of all the leaf node objects. Then, to approximate the force on one object from many objects that are far away in a distant cell, we can use one force calculation on a branch object (instead of doing one force calculation on all of them).

I've sketched out the basic force calculation on one object here. For each nearby cell, we need more exact information, so we do exact force calculations with every object in that cell. However, for a cell that is far away, we don't need such exact information for all the objects in it. Then, we can just use one force calculation, using the branch of that cell.

Note that once we have done all these force calculations, we will move the objects... And this will mean that the tree is likely to have to be rebuilt or adjusted, including branch nodes and centres of mass. The overall the process is still much faster than with the basic quadratic time force calculation, though. Still, it was this cost that led me to work on sampling-based methods, in the late 1990s and early 2000s. Fsmvis is a simple demo of this.

Sampling Methods for Force Models

Early (first?) example is Chalmers' 1996 paper, demo'd in Fsmvis

- Like Barnes-Hut, use fewer force calculations for *dissimilar* objects
- One can use a combination of samples and known high-D neighbours, to bring force calculation down from $O(N^2)$ to $O(N)$
- Relies on each object *i* having two sets: neighbours, and a random sample set
- Initially each neighbour set is empty, but has a constant maximum size (e.g. 5)

For each iteration

- For each object
  - Add up forces from any/all neighbours
  - Get a new random sample (e.g. 10) of all other objects, and add forces from these samples
  - Check each sample, and add it to neighbour set if it's closer than furthest neighbour (or free space)
  - Adjust velocity and position of this object

Barnes-Hut (and related methods) have to do a lot of work managing data structures such as binary trees of spatial cells. I'd used that kind of approach to visualize sets of documents, but found that the cost of updating the tree between each iteration took as much time as the force calculation (or even more).

We can get an even better speedup, though, through sampling and caching. Note that here we are going to try to do less work on *dissimilar* objects, whereas Barnes-Hut tried to do less work on *distant* objects. Distant (in low-D) and dissimilar (in high-D) are not always the same thing, especially at the start of a layout process when objects are really mixed up.

The core idea is that we will keep, for each object, two sets. One will gradually build up the nearest neighbours of the object, although it starts empty. The other is a random sample of other objects, refreshed at each iteration. The maximum size of each set is constant, so at each iteration we will do, for N objects, a constant number of force calculations… so the cost of the force calculation per iteration if O(N).

So, the algorithm goes through this double loop: for each iteration, for each object.. We first look at the neighbours of that object, if we have any, and add up the forces from them. Then we get a new sample set, and also add the forces from these samples. We also check the samples: could any of the samples go into our neighbour set? There might be empty space in it, or it might be that one of the sampled objects is closer in high-dimensional space than one or more neighbours. If so, then we can add the sample into the neighbour set (and maybe

kick out an old neighbour). In this way, over time, the neighbour set will get better and better… eventually getting close to be optimal set of high-D neighbours, in quite a cheap way.

This method works really well. It is much quicker than normal/simple methods, but it creates layouts with errors as good as them (or better). Again, I'll demonstrate with the Fsmvis demo program.

Stochastic Neighbour Embedding

Original SNE from Hinton and Roweis in 2008

- Many variants; t-SNE (van der Maaten & Hinton 2008) now the best known

Like springs, all dimensions are combined in 2D layout

- Unlike springs, similarity metric is *KL-divergence.* Attempts to make the *statistical distributions of neighbours* in low-D and in high-D match
- Focus is on local 'neighbour' structure, but not much concern about high-D non-neighbours
- High penalty for neighbours that are far apart in layout, but low penalty for non-neighbours that are close in layout
- So, global structure is poor (or seen as not important)
- Costly initial phase of calculating neighbours and distributions, then *gradient descent* to move objects into position
- Makes for time being quadratic in N, but also problems of very large matrices

Spring models are not widely used inside the machine learning or data science community. For a while, they tended to use an algorithm called tSNE… which is a variant of an earlier method called SNE. I am not sure why – I think it's partly because of the similarity metric that SNE uses, which are more familiar to ML folk.

For this reason, it puts all its work into getting local structure right… not just which objects are nearby in the layout, but the statistical distribution of those objects.

Its initial phase takes up most of the time, as it works out good parameters to describe the statistical distribution of the other objects around an object… for each object in the data set. Then it uses gradient descent, kind of like spring models do, to move objects into position… but, again, all the work is on the neighbours. Global structure involving relationships among the overall set is not directly modelled.

Stochastic Neighbour Embedding

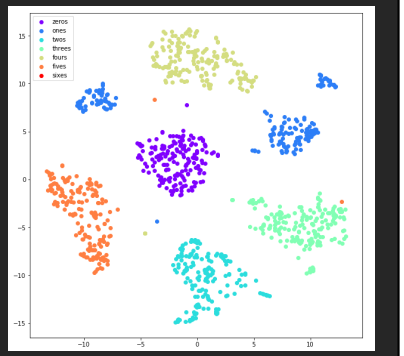
SNE favours clear separate clusters

- People like this, although SNE may make clusters where none exist!
- Very popular in machine learning community, sadly

SNE, tSNE and their variants don't scale well to large data sets

- High memory & computational costs, so N usually tiny, e.g 10k-50k
- Some hierarchical approximations help a lot, though

Excellent interactive overview and critique by Wattenberg and Viegas, at distill.pub



Layout of a subset of MNIST data set of images of written digits (from blog.paperspace.com)

Here's a typical tSNE image, with tight clear clusters. There is a bias in the algorithm to make layouts like that, but people seem to love this (sometimes false) sense of clarity and structure.

Scale is also a big problem. The cost of that first phase is high, and this means that layouts of very large data sets are not attempted with tSNE (although a few variants are now starting to handle bigger numbers)

I recommed you have a look at this great interactive article on tSNE by the leaders of the Google PAIR (People and AI Research) group. It shows tSNE's strengths and weaknesses really well.

UMAP: Uniform Manifold Approximation and Projection (McInnes et al., 2018)

Newer, faster and less biased (than tSNE) algorithm

- Again, out of the ML community. Some very complex maths involved!

Core idea is to find the highD neighbours for each object, but also a model (manifold) that describes connections/distances between objects

- Uses some advanced topological methods to do this
- Also applies very fast algs for finding (approximate) neighbours in high-D spaces (random projection trees, and nearest neighbor descent), and fast variants on stochastic gradient descent.

Python code available on Github here

Best overview is perhaps this video from SciPy conf 2018, on YouTube

UMAP is the new algorithm in town, and it's a great advance upon tSNE in particular. It resolves a lot of the bias in tSNE, and also it scales well

The core idea uses some quite advanced methods in topology and geometry, to set up a model (or manifold) that describes objects and their neighbourhoods. It also uses some other algorithms to do that – random projection trees, from 2008, and nearest neighbor descent, from 2011. The actual layout involves stochastic gradient descent, as well as some clever sampling tricks, to work quickly.

There's a link here to a video of Leland McInnes doing a talk on UMAP on YouTube, and it's worth watching… although the section on topological/geometric methods is tough (for me, at least). His code is also on Github.

UMAP

Balances local & global structure

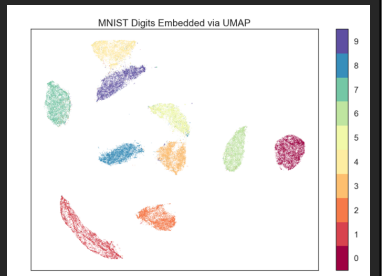
- Keeps similar objects close, but dissimilar ones far
- Rising fast in machine learning community

Ultimately, it's very similar to fast sampling-based spring models

- But UMAP tailors its model to represent inter-object relationships in a more in-depth way

Scales well to large N

- Up to a few million, anyway
- Order of complexity not proven yet, but McInnes guesses O(d*$N^{1.14}$)



Layout of the 70000 MNIST data set of images of written digits (from McInnes' Github page)

UMAP works to get neighbours in highD space (similar objects) to be close together in the layout, but (unlike tSNE) it also works to make objects that are far apart in highD space (dissimilar) be far apart. This means that global structure is better, with the shape of clusters and also the relative position of clusters, outliers, connectors and so on being much better than in tSNE.

Some fast spring models use sampling methods to work kind of like UMAP does, but UMAP has pulled together lots of very fast yet reliable elements, and also its model of local relationships between objects is more detailed and tailored to the data itself.

McInnes hasn't proven the time complexity (last time I looked), but in one Q&A he estimated it as O(d*N^1.14), where d is the dimensionality of the data. It's fast… and it can easily handle data sets like the full 70000 MNIST handwritten digits data set. People are regularly doing layouts of a few million objects, quite accurately.

UMAP is the last dimensional reduction algorithm we'll look at in this lecture.

Visualising Multidimensional Data

Strategies

- First, visualise individual dimensions (maybe of a sample)
- Avoid 'over-encoding' too many dimensions onto one visualisation
- Use space and small multiples (e.g. scatterplot matrix) intelligently
- Dimensional reduction is useful when there are too many dimensions for small multiples
- Use interaction to explore linked *relevant* views

There is rarely a single visualisation that answers all questions. Instead, the ability to create/combine/explore appropriate visualisations quickly is key

So, to conclude…

Here is an overview of what to do when visualizing multidimensional data

Start by examining the data with regard to each individual dimension, if the number of dimensions is small enough. See how it's distributed, or clustered, or not. Look for outliers and errors, and clean it up if need be. If the data set is very large (high N), then just take a random sample so as to get a more manageable size to work with.

As you start to combine dimensions in a visualization, watch out for the problem I tried to show with the coffee shop example. Too many encodings tends to mean a very cluttered and hard-to-read visualization. It is often better then to break up the data into multiple linked views, perhaps using 'small multiple' pattern of similar views, and then use interactions such as brushing and linking, so as to interact with and refine the views.

If the dimensionality is very high, then methods such as scatterplot matrices won't work… so you probably should use dimensional reduction. Again, though, I would suggest that you should use linked views – linking to visualisations of relevant individual dimensions -- so that you can see what is going on in selected parts of your layout, e.g. what clusters mean.

Again, iteration is the trick here. There is rarely if ever one perfect visualization that answers

every possible question, that shows all that there is in the data. The ability to create, move between and combine different visualisations is key to success… it's an iterative and exploratory process.