Lecture 4 - Classification, dimensional reduction

See https://github.com/jbrinchmann/MLD2023 as usual

Lectures/Lecture 4 has the PDF for today and

Model choice & optimisation

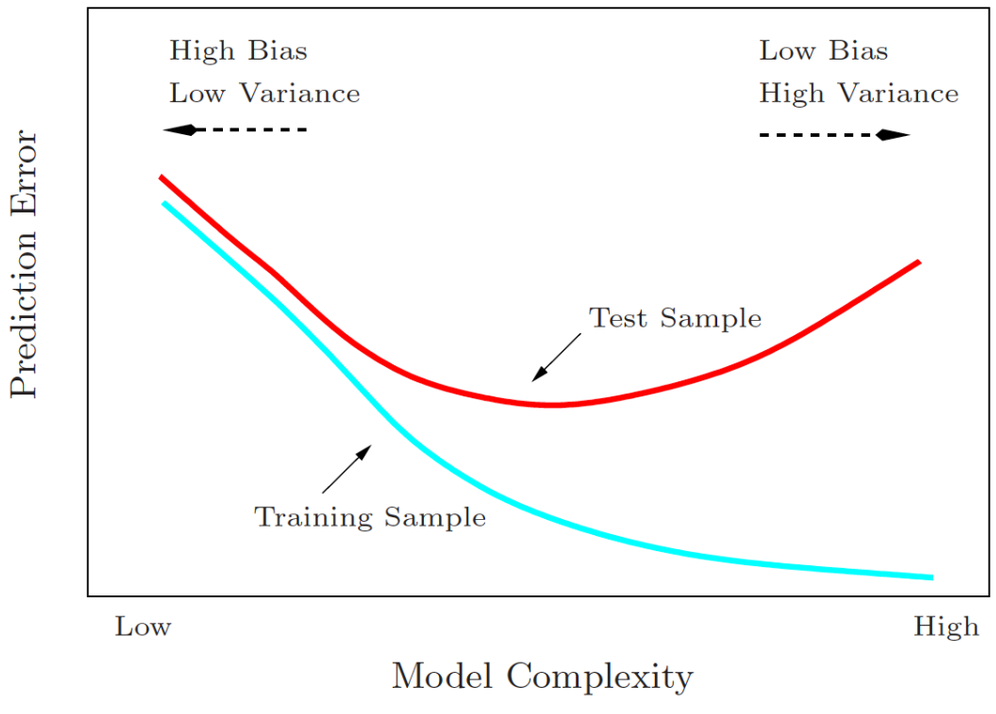
Overfitting - When the model is too flexible for the data and hence does not generalise well.

Underfitting - When the model is not flexible enough to fit the data.

The bias-variance trade-off

$$\mathrm{Err}(x_0) = (E[\hat{y}] - f(x_0))^2 + \mathrm{E}\left[(\hat{y} - \mathrm{E}[\hat{y}])^2\right]$$

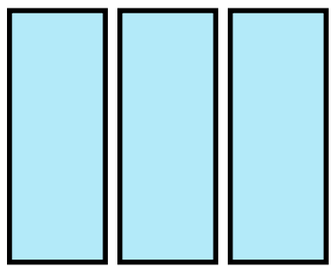
Bias^2 Variance



Cross-validation

One possibility

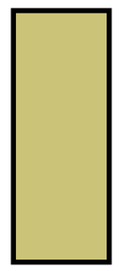
k-fold CV



Training

.....

k-copies

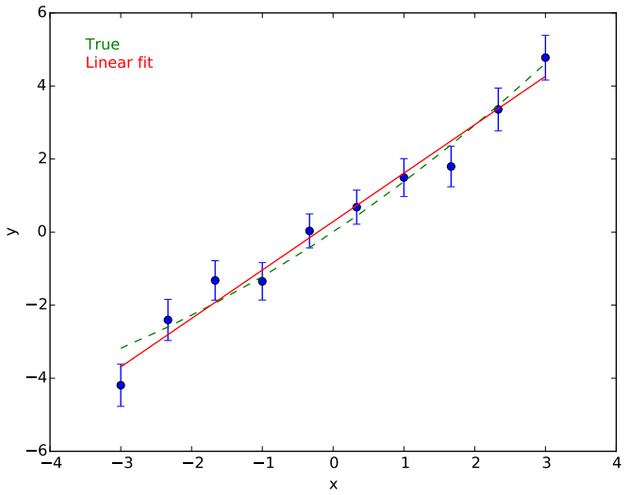


Test

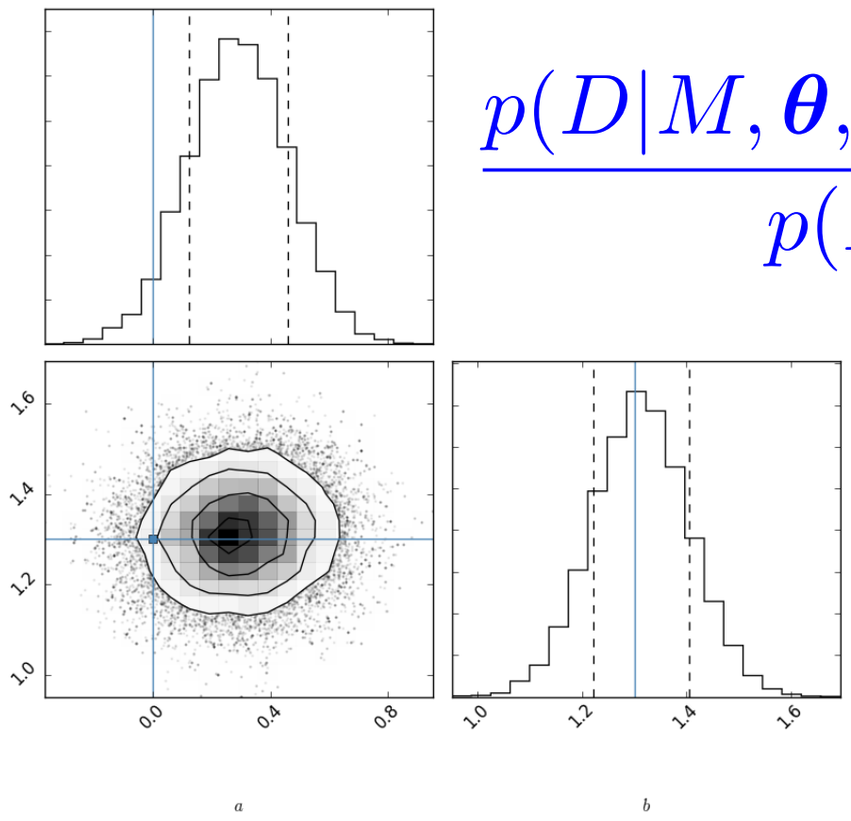
Do this splitting k times.

Last lecture:

$$\ln L = -\frac{1}{2}\ln 2\pi - \frac{1}{2}\sum_i \frac{[y_i - (\alpha + \beta x_i)]^2}{\sigma_i^2}$$

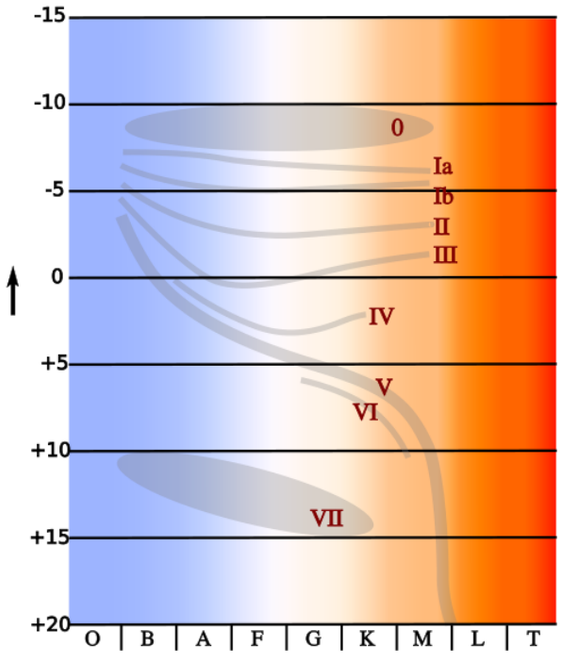


Classical inference

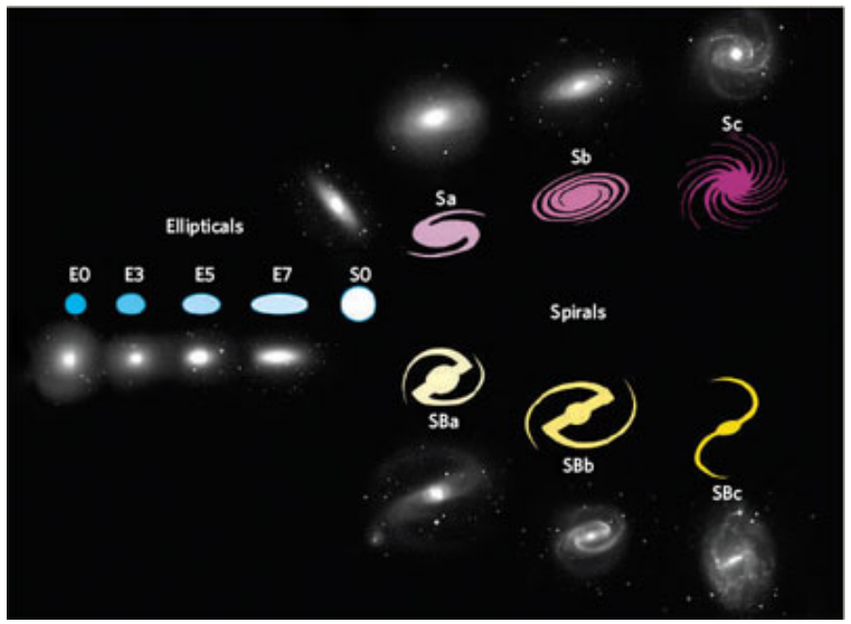


$$\frac{p(D|M,\boldsymbol{\theta},I)p(M,\boldsymbol{\theta}|I)}{p(D|I)}$$

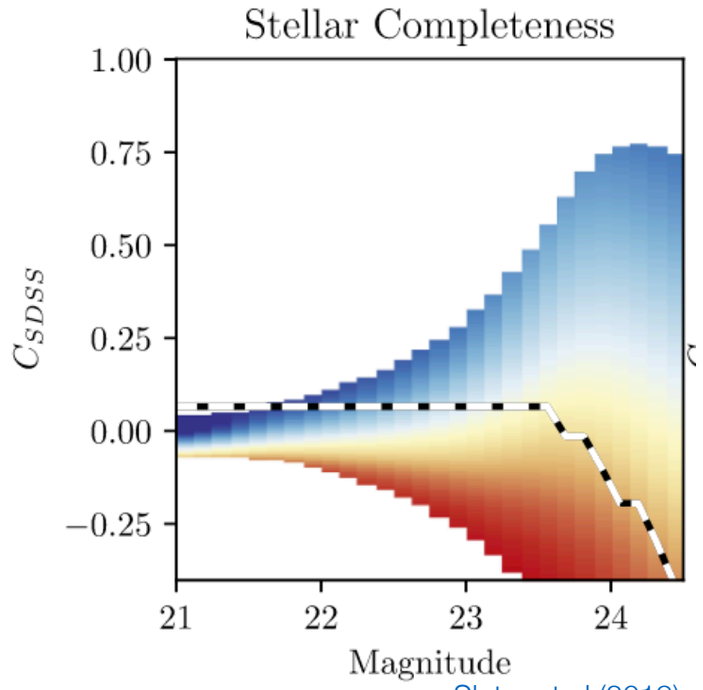
Bayesian inference



Wikipedia

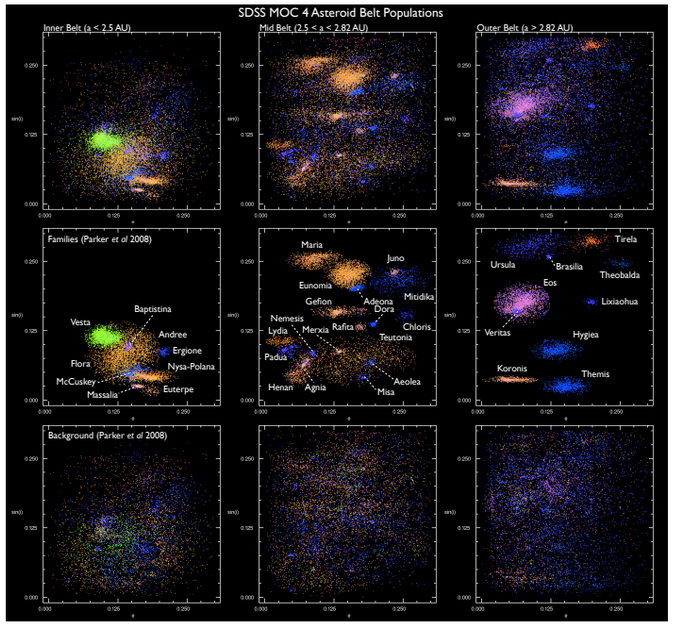


Kennicutt (2006), Nature

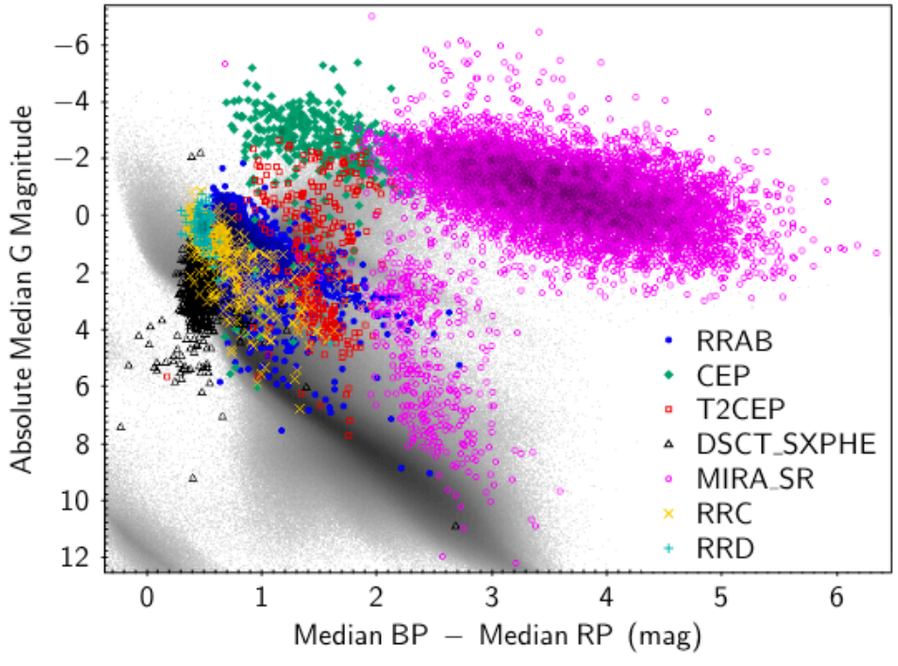


Slater et al (2019)

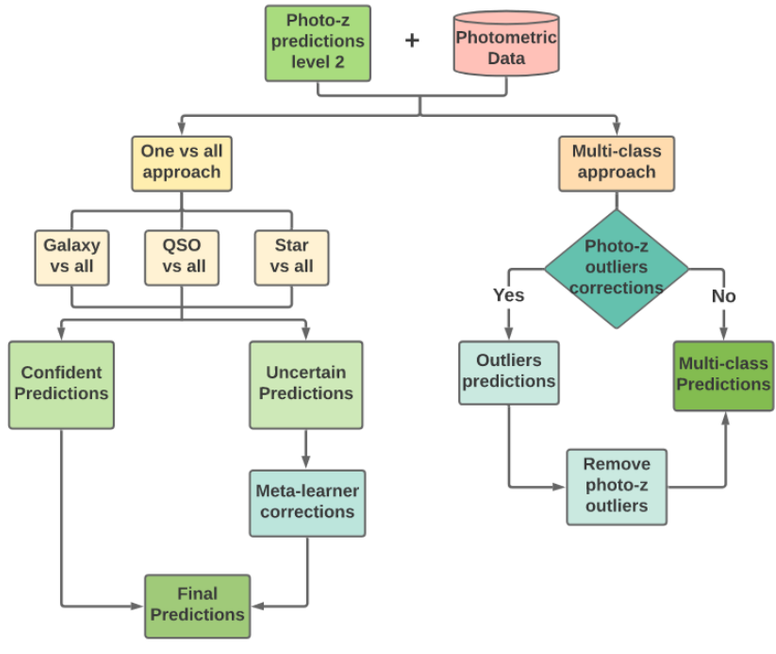
Classification



Parker et al (2008, Icarus 198,138-155)

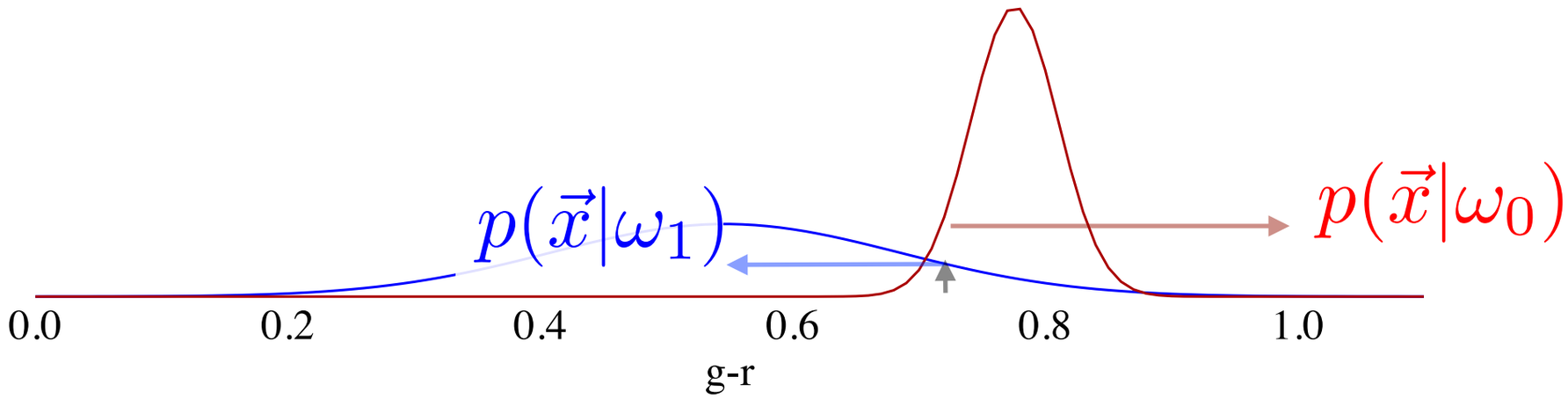


Rimoldini et al (2019)

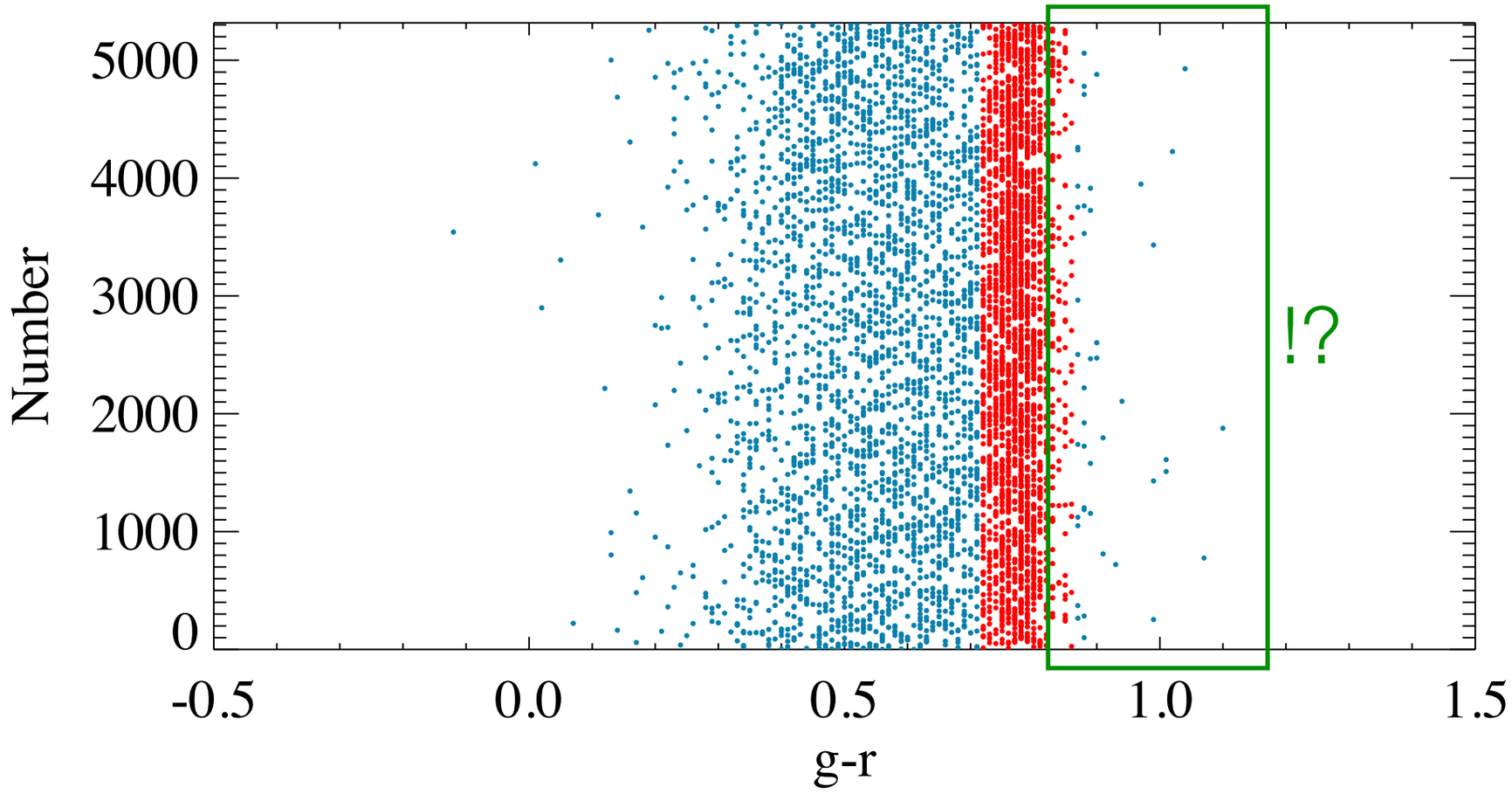


Cunha & Humphreys (2022)

Bayesian classification:



So in this case we assign a class Elliptical for the red and we can also give the likelihood relative to the spiral class. We do this for all objects:



High dimensionality - Naïve Bayes & Bayesian networks

That appears to be a simple technique, but estimating $p(\vec{x}|\omega_i)$ is challenging in high dimensions - if you need N objects for a 1D PDF, you need N^d data points for d-dimensional PDF.

A simple way to reduce the requirements is to assume that all variables are independent, so that you have

$$p(\vec{x}|\omega_j) = \prod_i p(x_i|\omega_j)$$

Now you can just repeat what I showed for each ‘feature’ and multiply the results together - this is known as **Naïve Bayes**.

A bit more sophisticated is to partially factorize $p(\vec{x}|\omega_i)$ This leads to **Bayesian networks**. But the process is very similar to what we have seen.

Other classification/grouping methods

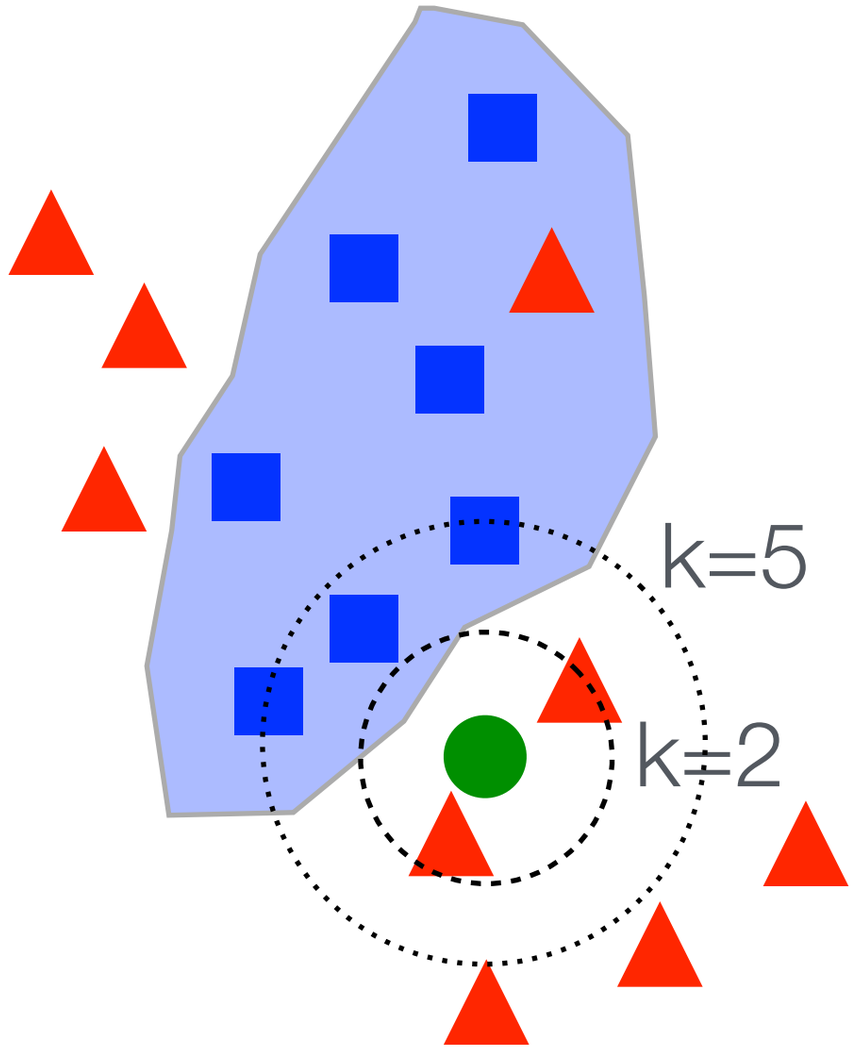
In many situations a full application of Bayesian statistics is difficult to carry out, particularly in high dimension. There are therefore a range of alternative methods that are less “optimal” but can provide very flexible and useful techniques.

It is worth keeping in mind however, that the Bayesian technique not only is optimal (if you know the PDFs...), but it also assigns a probability for belonging to a particular class.

k-Nearest neighbours

Training: For each object, find the nearest k objects - assign the class of the majority of the neighbours.

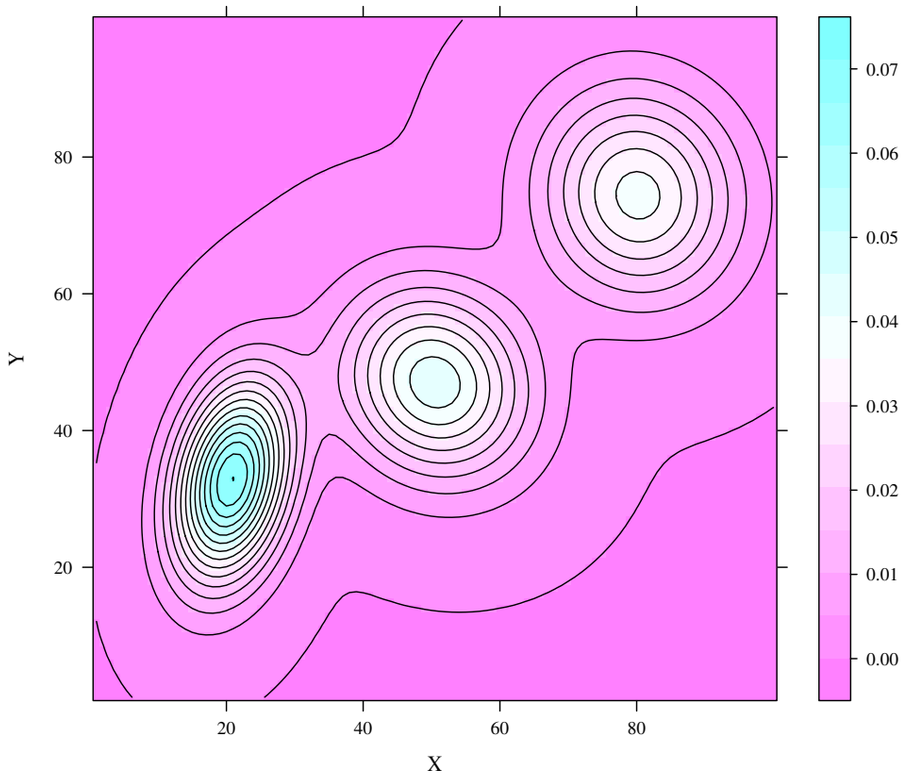
Application: Find the nearest k objects and assign the majority class.



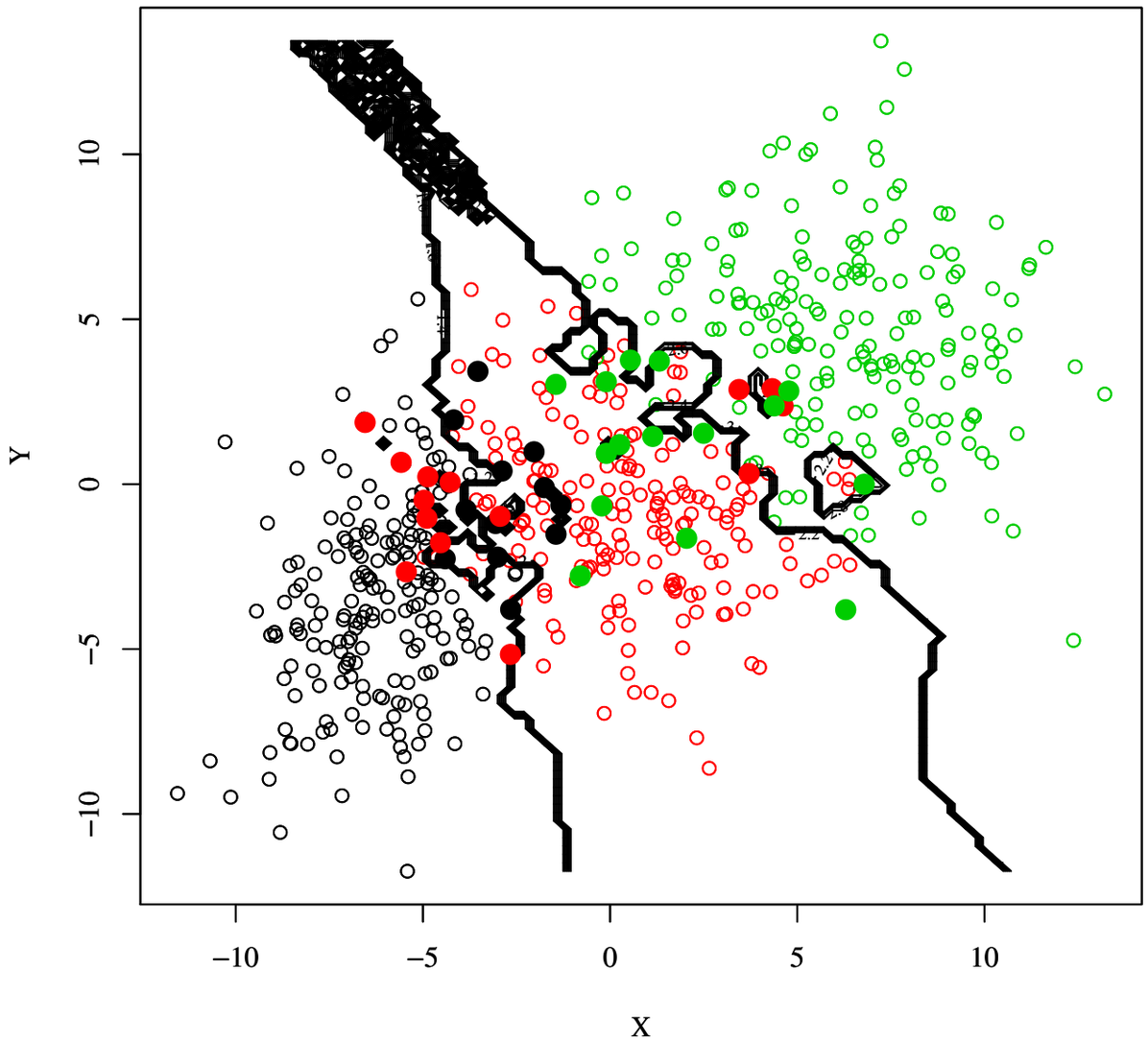
In python: `from sklearn.neighbors import KNeighborsClassifier`

k-Nearest neighbours

Underlying distributions



The result of applying knn



knn can lead to rather complex decision surfaces and is therefore not always optimal for **understanding** what is going on.

Example: k-nearest neighbours regression

```
def knn_regress(x, y, xout, k=3):

    # Our estimates
    yhat = np.zeros(len(xout))
    for i, xo_i in enumerate(xout):
        i_close = find_k_closest(xo_i, x, k=k)

        yhat[i] = np.mean(y[i_close])

    return yhat
```

The real work is done in the `find_k_closest` code.

Example: k-nearest neighbours regression

```
def find_k_closest(X_i, X, k=3):

    # Get the pairwise distances
    dd = (X_i-X)**2

    # Sort the distances.
    i_sort = np.argsort(dd.squeeze())

    # Find the k closest
    k_closest = i_sort[0:k]

    return k_closest
```

SLOW!

Example: k-nearest neighbours regression

How long does it take to run?

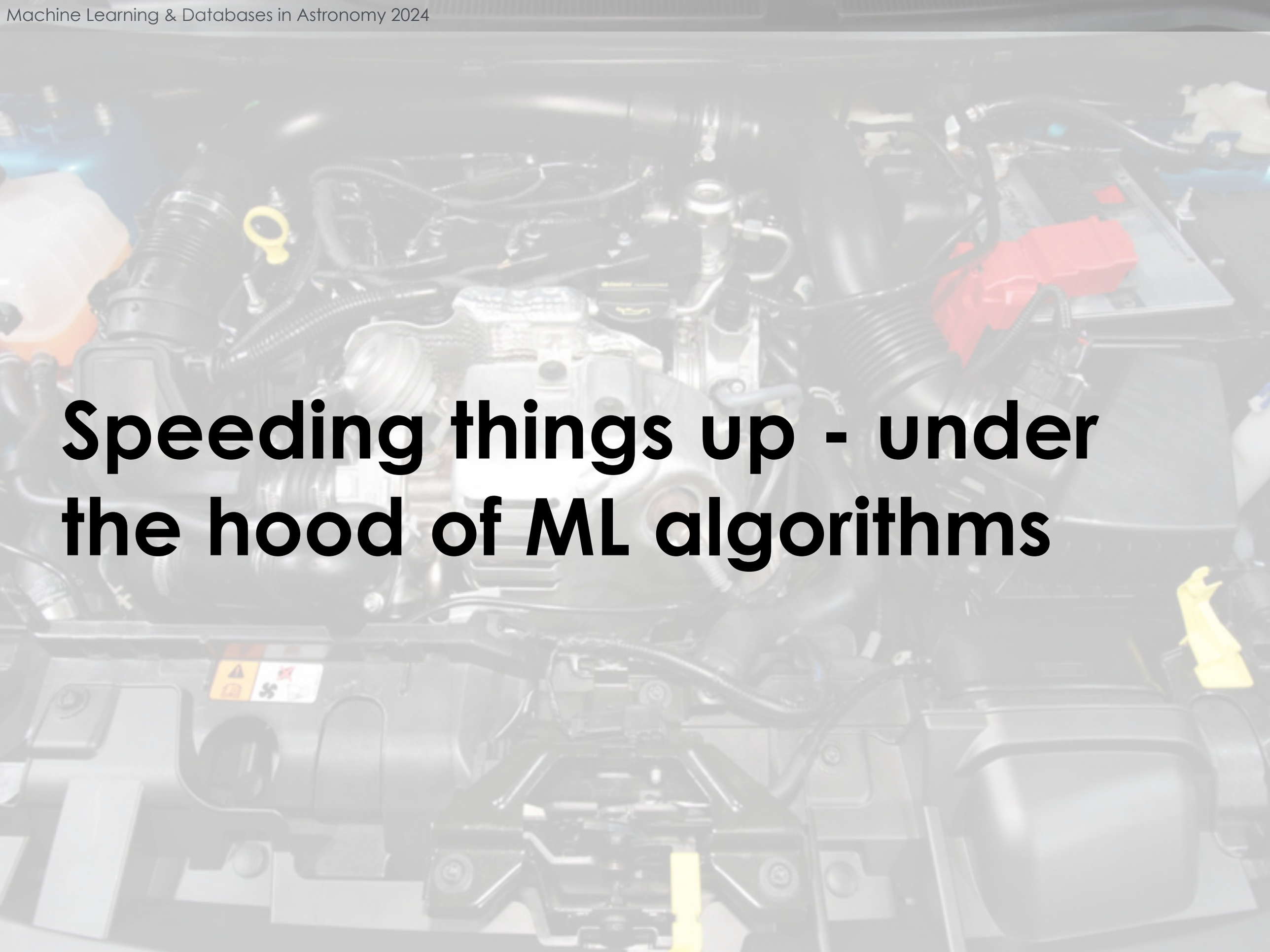
N^2

k-nearest neighbours regression

How long does it take to run?

$$N^2$$

So going from 10 samples to 100, means an increase in running time of a factor of 100.



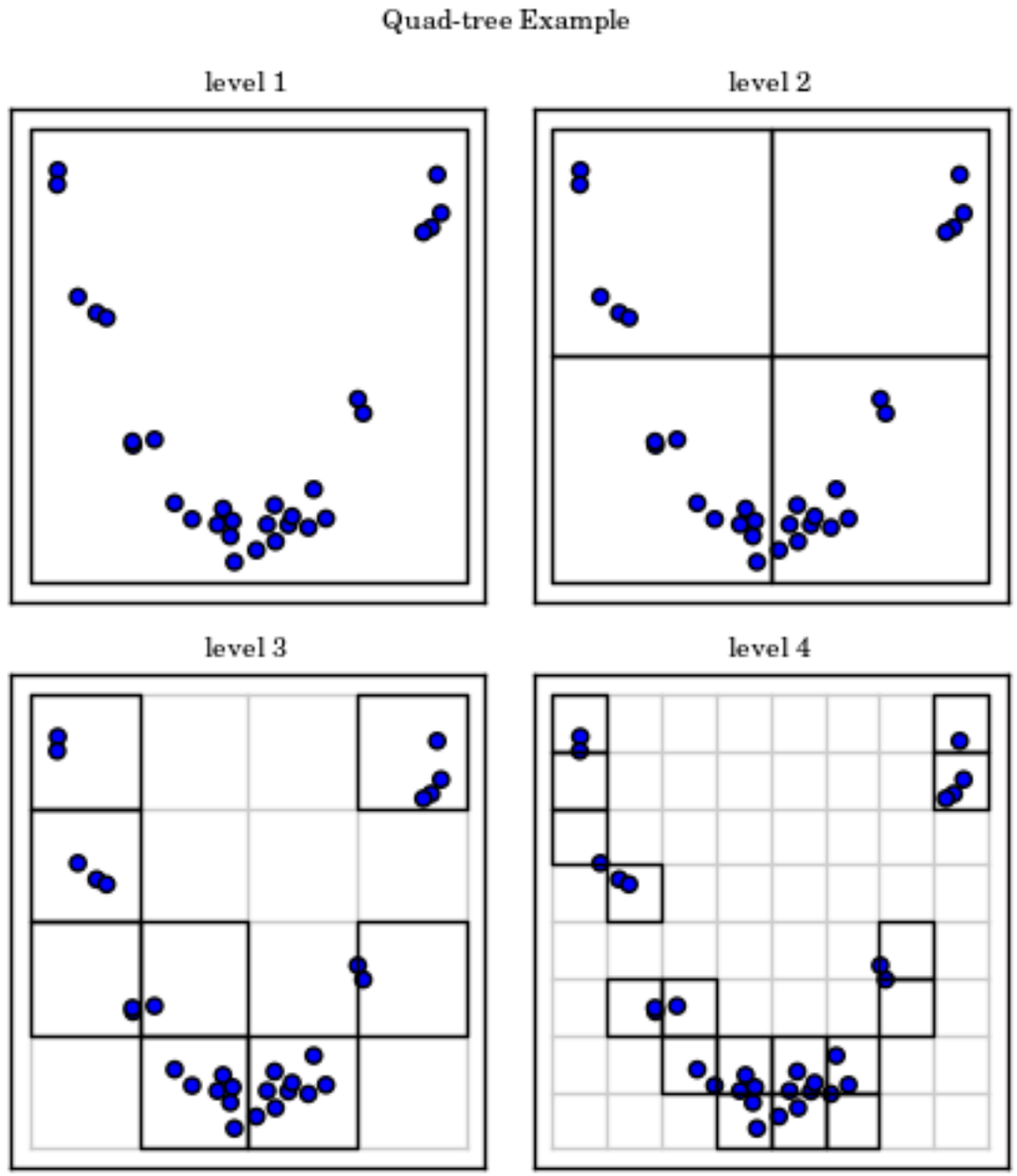
Speeding things up - under the hood of ML algorithms

Finding neighbours - tactics

Split & conquer - organise your data in a clever way:

The quad-tree is a quick and easy way to divide data. Creation of the tree takes $O(N \log N)$ to create and $O(\log N)$ to search

In 3D called an oct-tree: make 8 children per node etc.



From Ivezic et al, “Statistics, Data Mining, and Machine Learning in Astronomy" (2013)

Finding neighbours - tactics

From quad-tree to kd-tree:

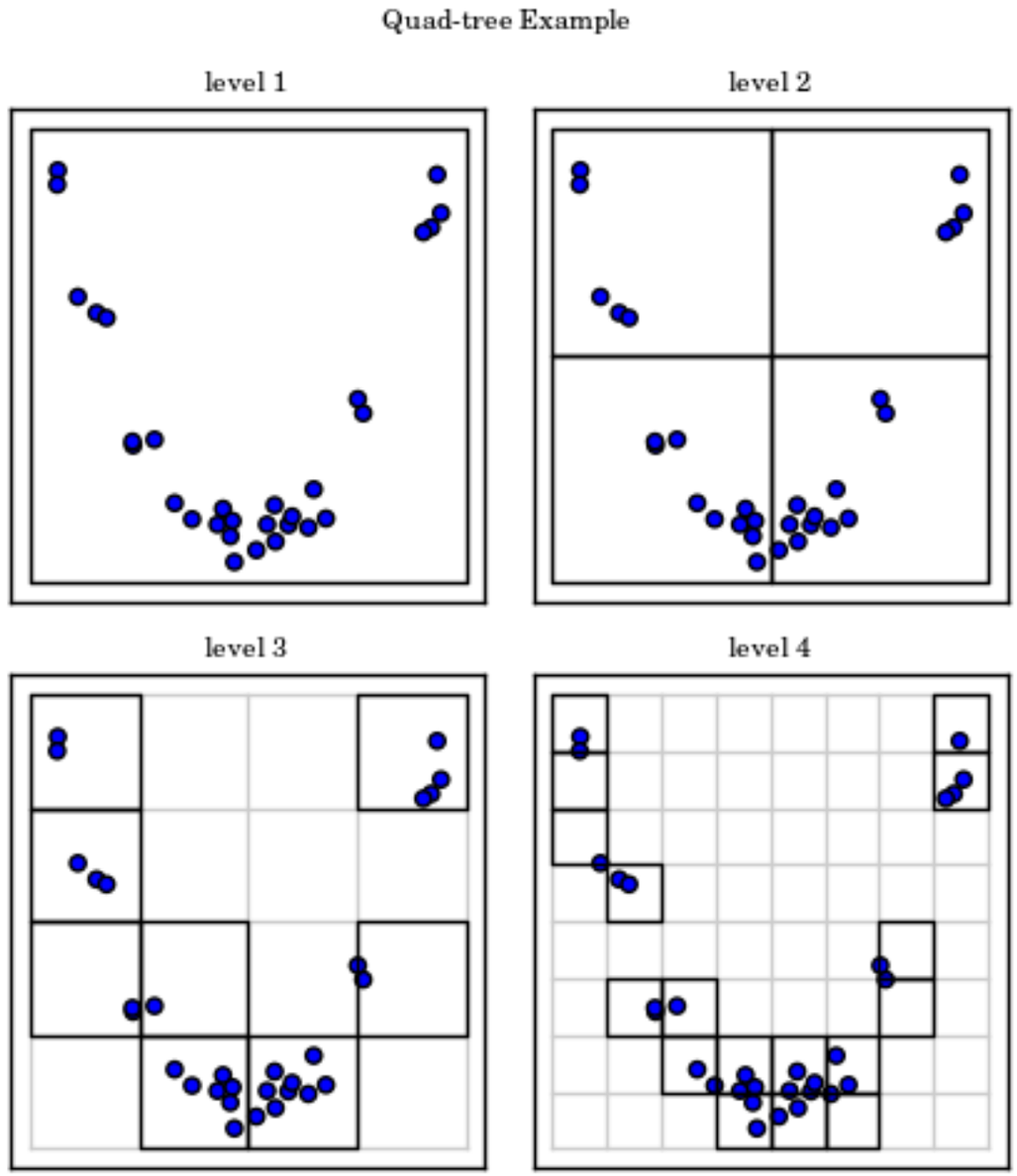
Extending to D dimensions is appealing but can be costly:

$$N_{\text{children}} = 2^D$$

So for D=100 & 1 byte per node, we need:

$$N_{\text{children}} = 2^{100} \approx 1.2 \times 10^{15}\,\text{Pb}$$

to store one level…



From Ivezic et al, “Statistics, Data Mining, and Machine Learning in Astronomy" (2013)

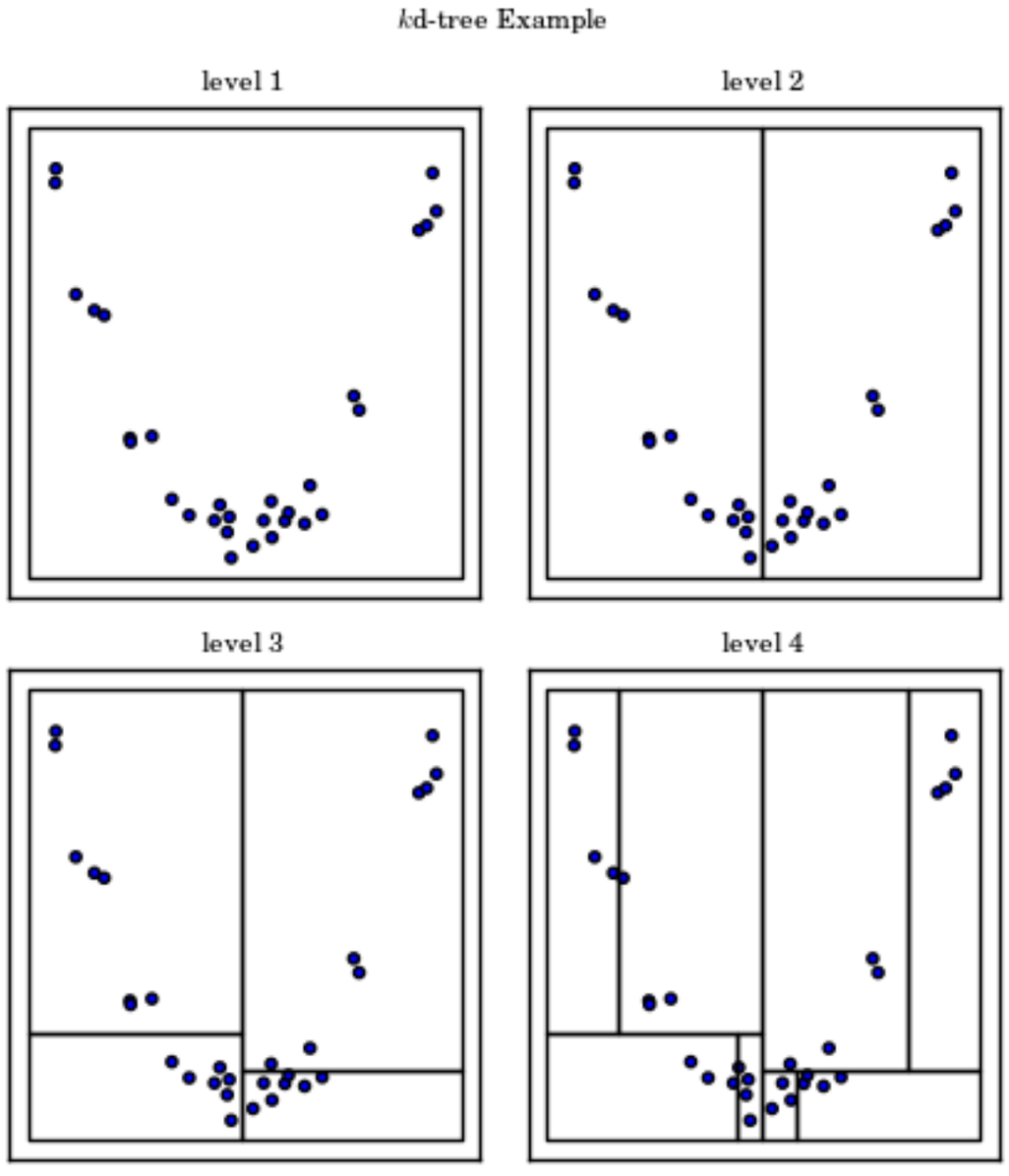
Finding neighbours - tactics

From quad-tree to kd-tree:

Extending the D dimensions is appealing but can be costly:

$$N_{\text{children}} = 2^D$$

Instead: divide each dimension in 2 at a time - this gives **kd-trees.** And they work well, at least part of the time

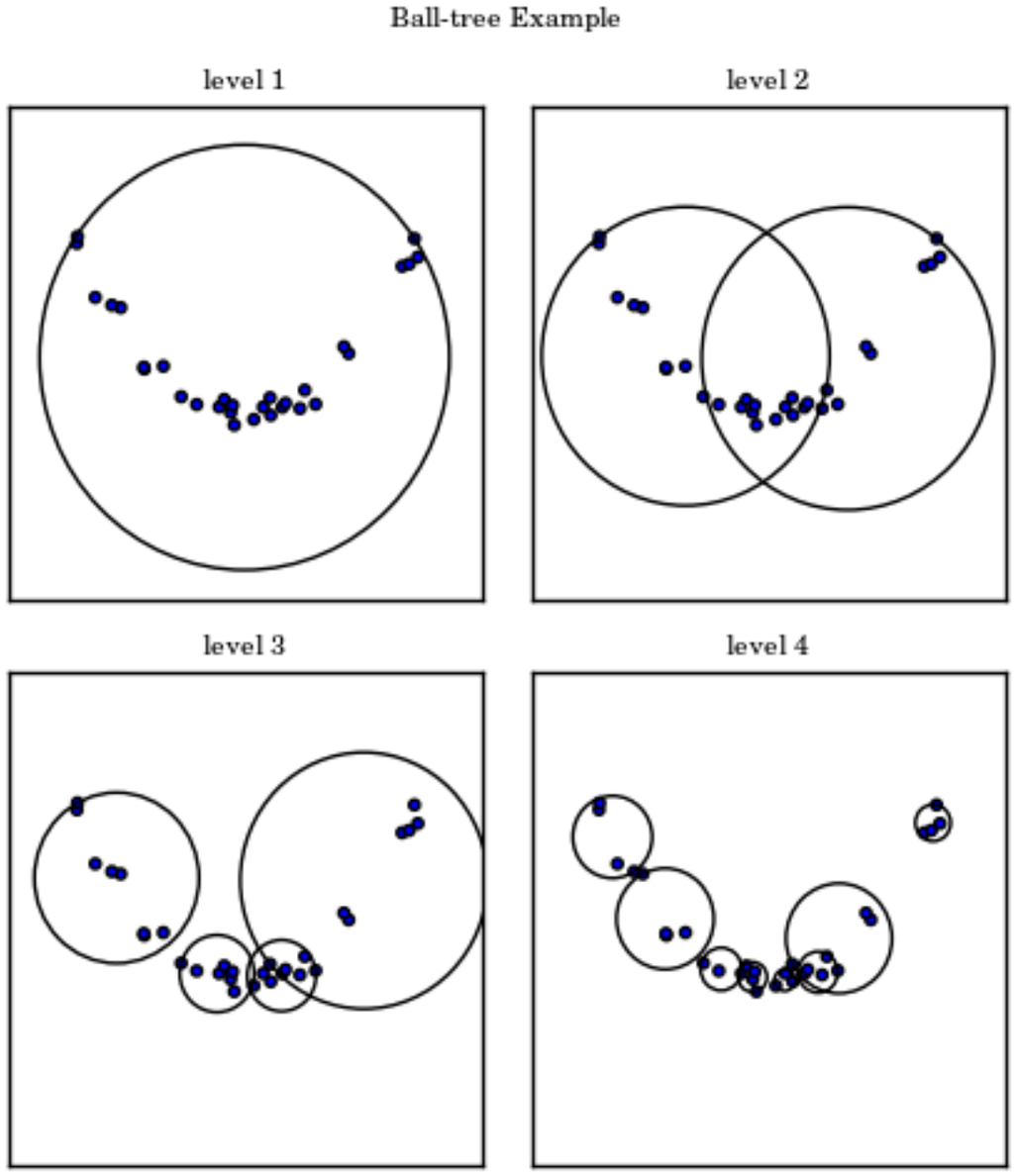


From Ivezic et al, “Statistics, Data Mining, and Machine Learning in Astronomy" (2013)

Finding neighbours - tactics

Ball-trees

kd-trees also work badly for high dimensions - ball-trees are an alternative to this



From Ivezic et al, “Statistics, Data Mining, and Machine Learning in Astronomy" (2013)

Finding neighbours - tactics

Knn regression with Ball-trees:

```
def knn_regress_ball(x, y, xout=None, k=3):
    # Make the ball tree
    X = x[:, np.newaxis]
    bt = BallTree(X)

    # Query & prediction
    dist, inds = bt.query(xout[:, np.newaxis], k=k)
    yhat = np.mean(y[inds], axis=1)

    return yhat
```

From Ivezic et al, “Statistics, Data Mining, and Machine Learning in Astronomy" (2013)

Dimensional reduction

Fitting models & dimensionality

Assume you wish to fit a model to a high order polynomial:

$$y = w_0 + \sum_i w_i x_i + \sum_i \sum_j w_{ij} x_i x_j + \sum_i \sum_j \sum_k w_{ijk} x_i x_j x_k$$

The number of terms grows as d^M where d is the number of input variables and M is the polynomial order. So for higher order functions you need to constrain a large number of terms and need huge training datasets.

There is a theorem (Barron 1993) saying that for polynomials the error in an approximation goes as $O(1/M^{2/D})$ - D is the dimensionality. While for non-linear functions it goes as O(1/M). For large D you therefore need many more term for polynomial fits than for non-linear fits.

An aside: Higher dimensions - weirdities

You might think that the optimal way to sample spaces is in regular bins - and that the cube & the sphere cover similar volumes.

$$V_{\mathrm{sphere}}(r) = \frac{\pi^{\frac{D}{2}}}{\Gamma(\frac{D}{2}+1)} r^D$$

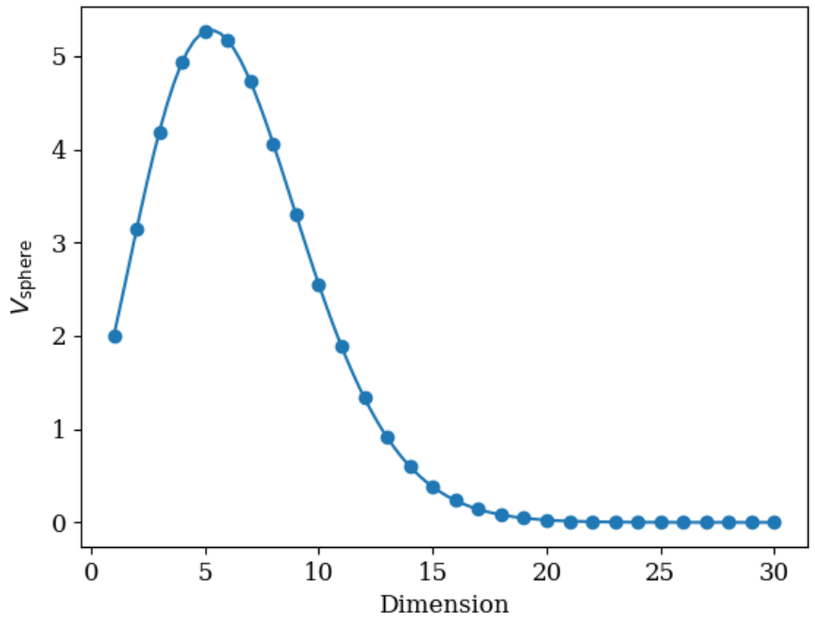
in D dimensions, and since:

$$\frac{V_{\mathrm{sphere}}(1) - V_{\mathrm{sphere}}(1-\epsilon)}{V_{\mathrm{sphere}}(1)} = 1 - (1-\epsilon)^D$$

most of this volume is in a thin shell for high D. And indeed the volume of a sphere goes to zero when D goes to infinity!

Higher dimensions - weirdities

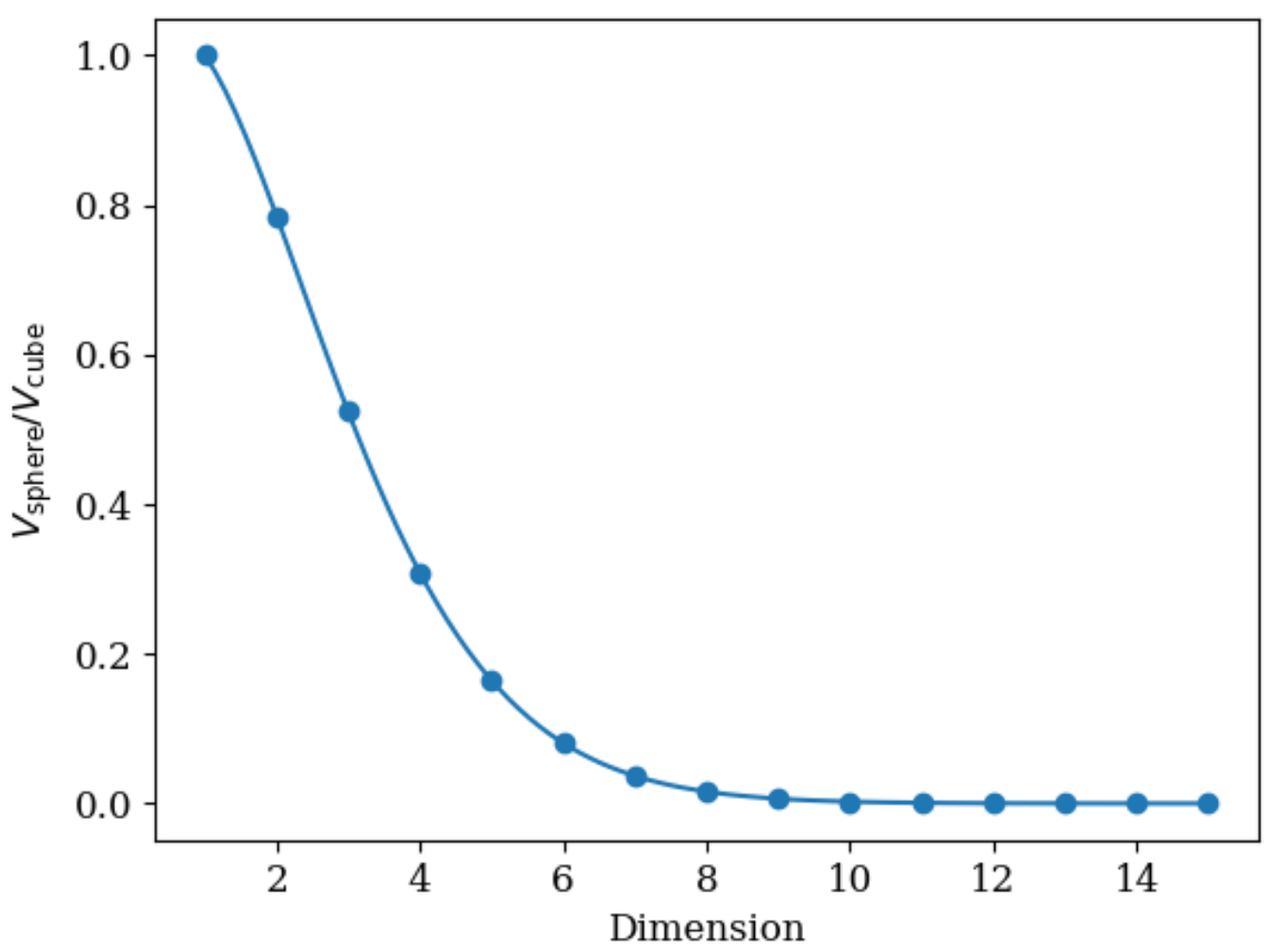
$$V_{\mathrm{sphere}}(r) = \frac{\pi^{\frac{D}{2}}}{\Gamma(\frac{D}{2}+1)} r^D$$



We can also compare this to the volume of the cube, and we find:

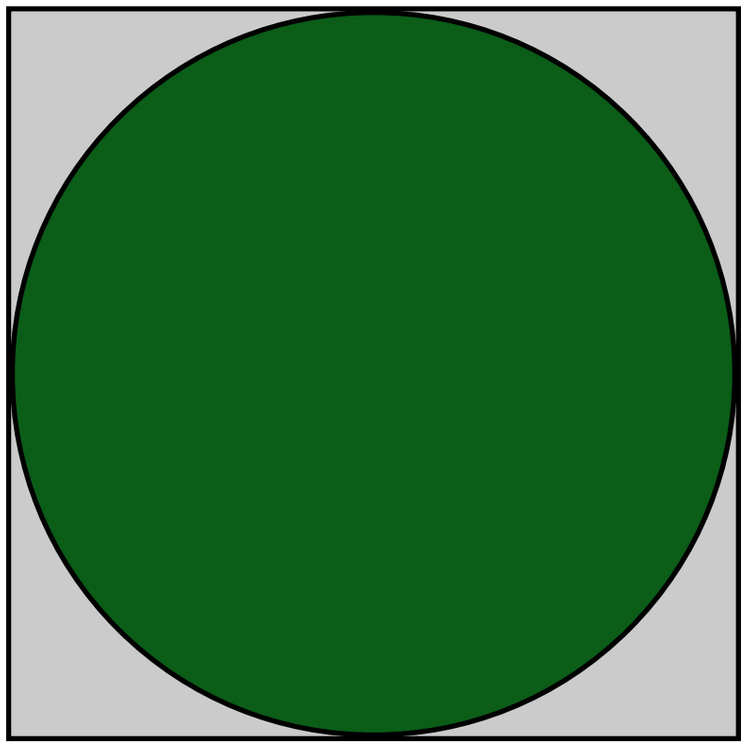
$$\frac{V_{\mathrm{sphere}}}{V_{\mathrm{cube}}} = \frac{\pi^{\frac{D}{2}}}{\Gamma(\frac{D}{2}+1)}$$

Very few points lie within a unit radius from the origin!



The curse of dimensions

So does this matter? It matters for instance in Monte Carlo techniques

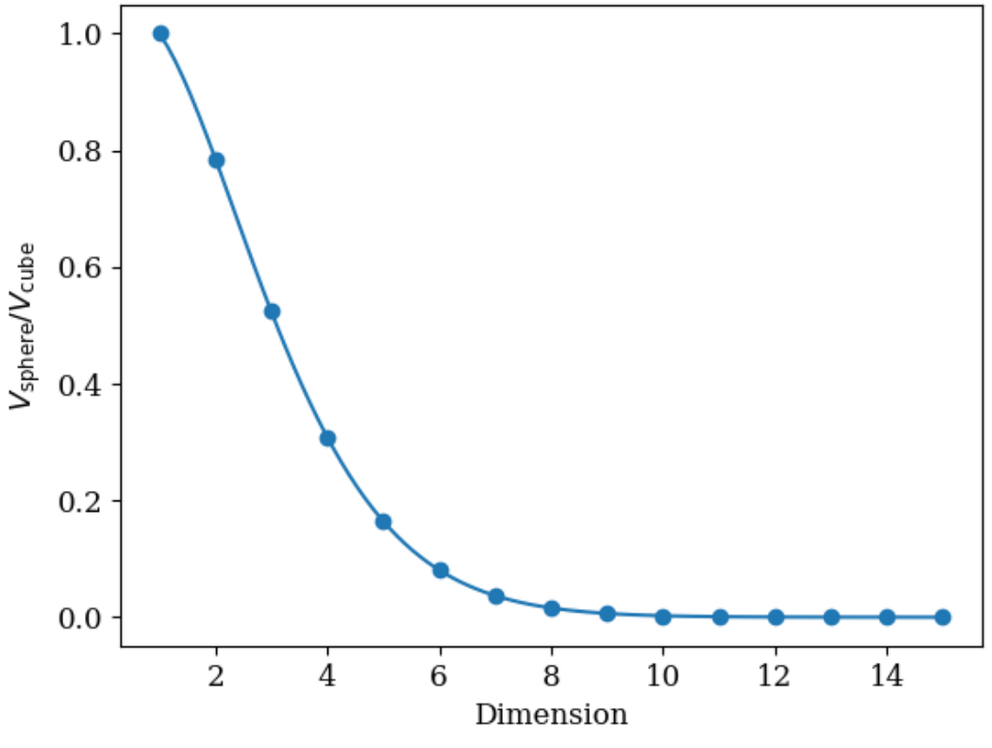


If you want to calculate

$$\int\int f(x, y)\, dx dy$$

over the green disk - you might opt for a Monte Carlo integration technique where you draw random numbers within the gray square and reject those that lie outside the green disk.

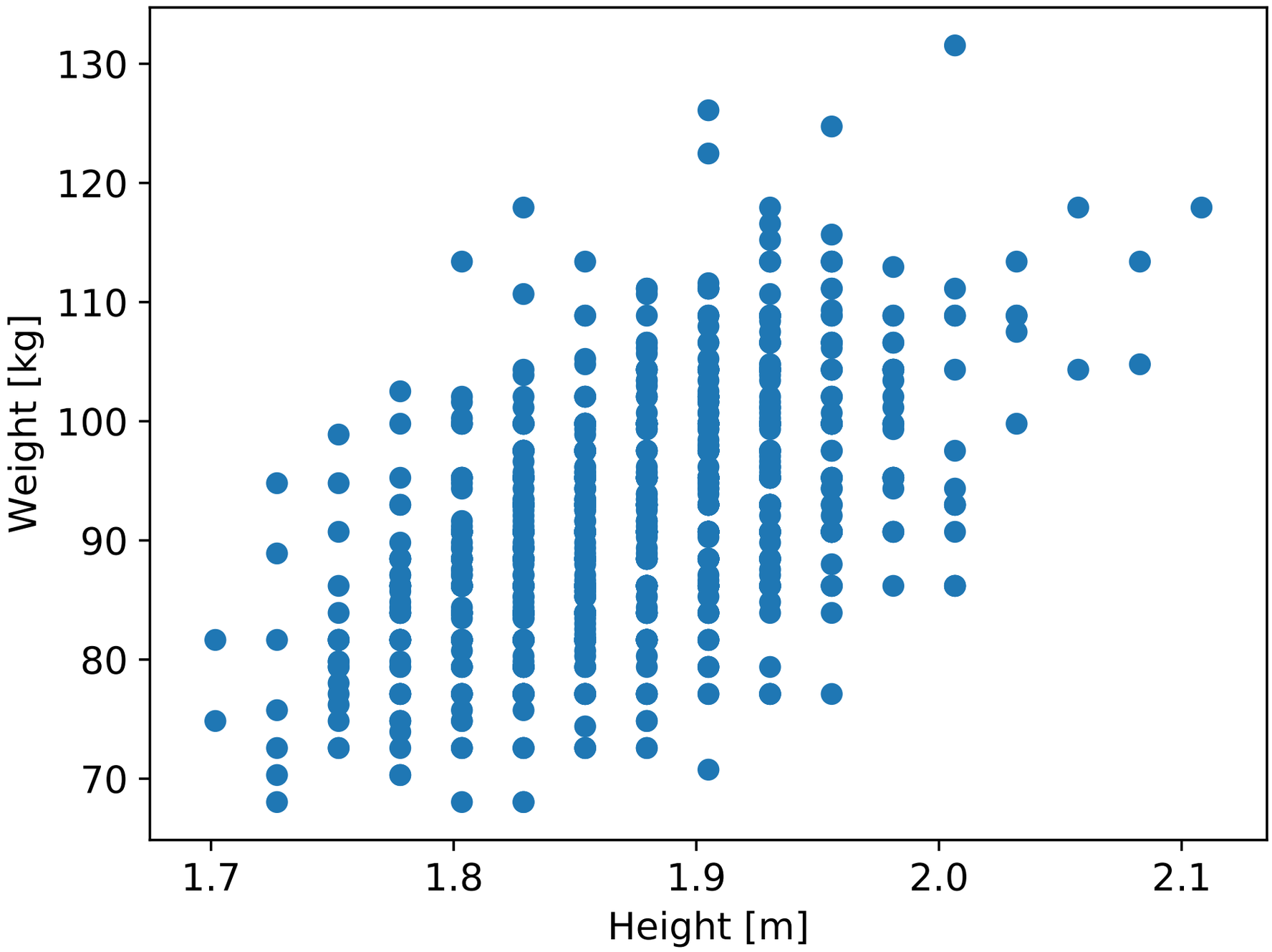
But in high-D, the chances of a point falling in the green sphere is very low!

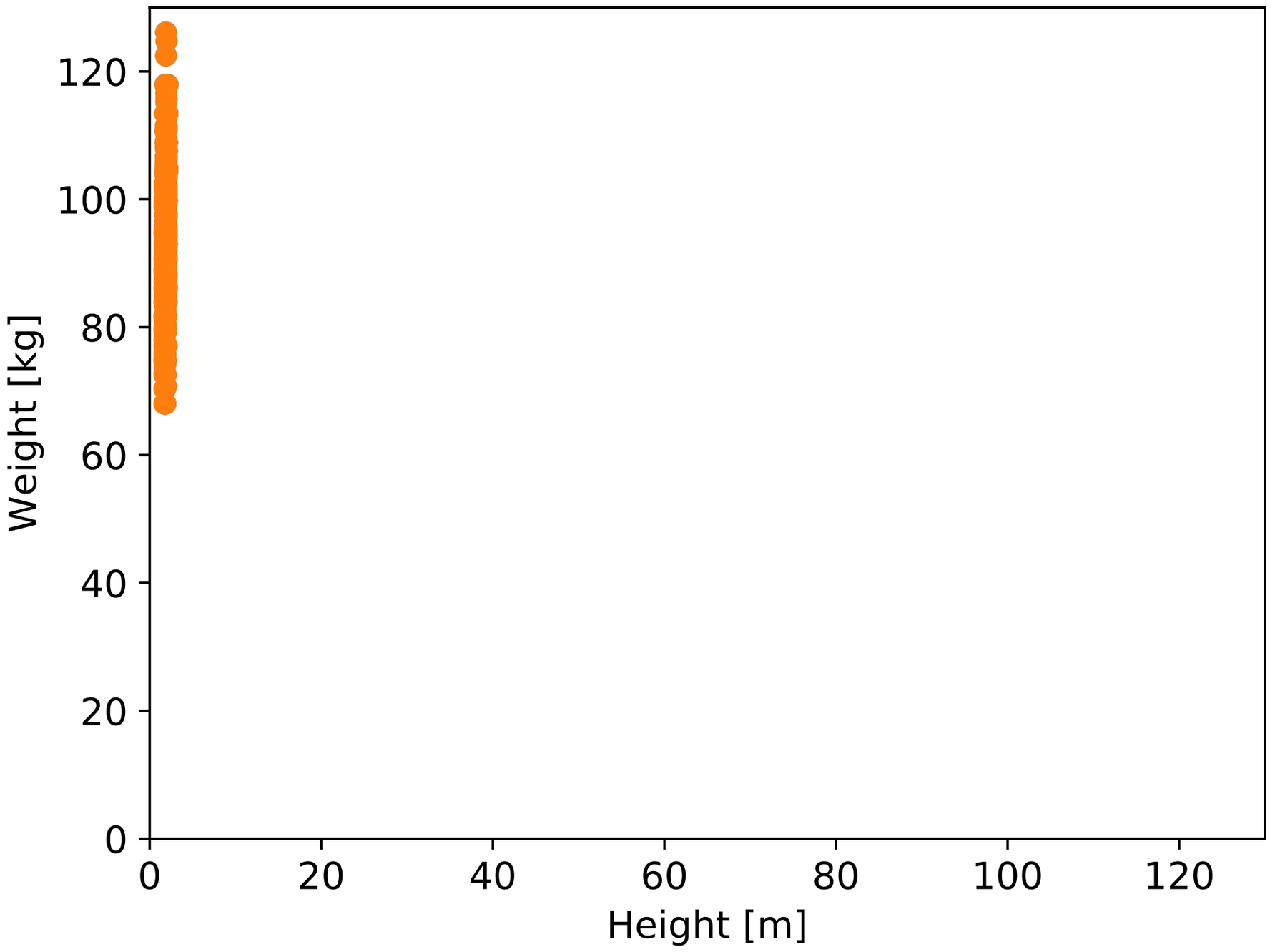


But in reality, life is better

Real physical objects do not fill “space” arbitrarily - thus we can look for ways to **reduce** the effective **dimension** of the problem.

But first - you might need to pre-process your data:


130
120
110
100
90
80
70
Weight [kg]
1.7
1.8
1.9
2.0
2.1
Height [m]


Weight [kg]
0
20
40
60
80
100
120
Height [m]
0
20
40
60
80
100
120

Rescaling data

In many algorithms it is highly desirable that the ranges probed in each dimension are comparable. To achieve that we **scale** our data.

Scale range to -1 to 1, or close to it:

$$x_i \to \frac{x_i - x_{\min}}{x_{\max} - x_{\min}}$$

min-max scaling

$$x_i \to \frac{x_i - \bar{x}}{\sigma_x}$$

standardizing

Often called whitening

Approaches to standardizing

$$x_i \rightarrow \frac{x_i - \bar{x}}{\sigma_x}$$

1. Use sklearn convenience function:

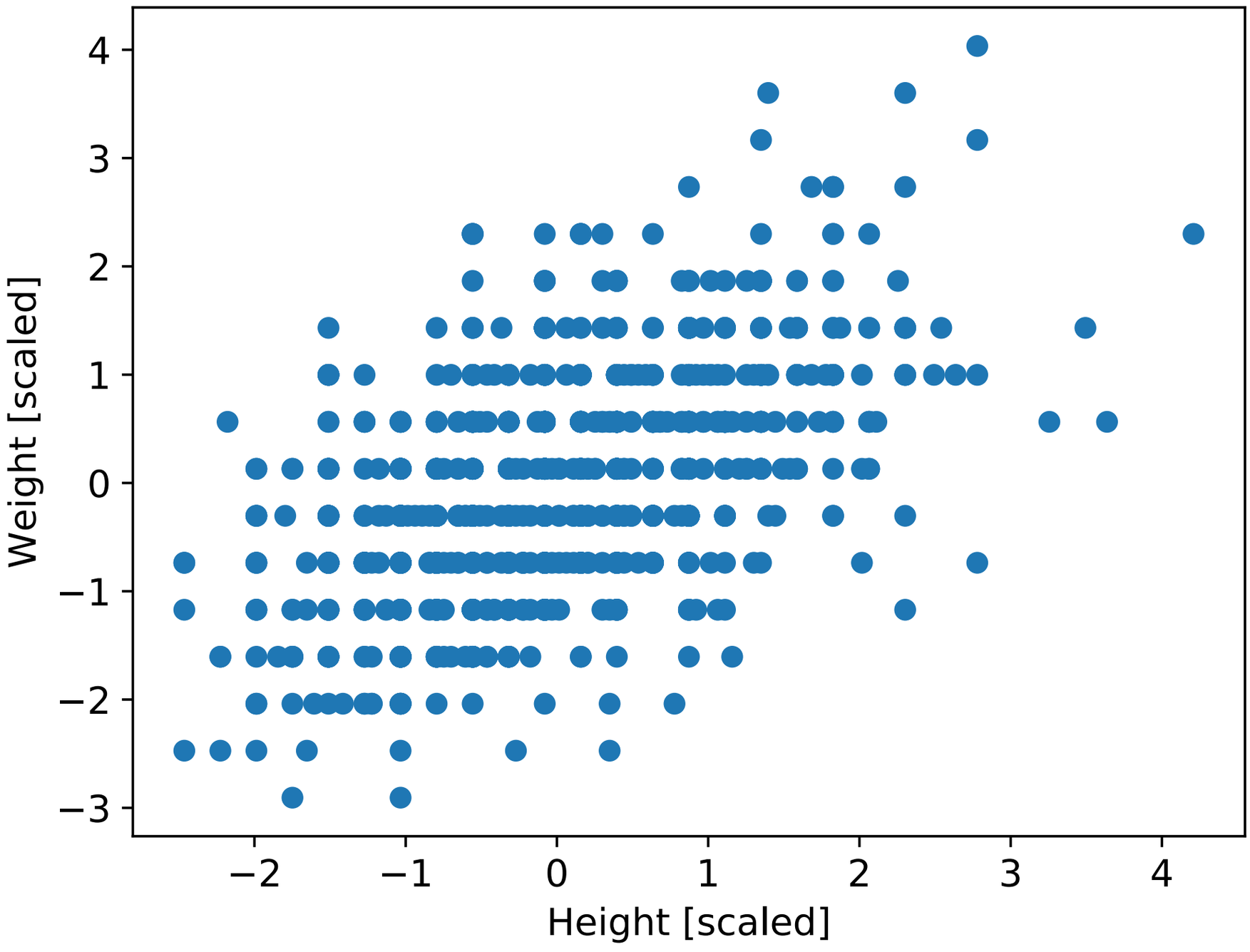
```
from sklearn.preprocessing import StandardScaler
X_std = StandardScaler().fit_transform(X)
```

2. Calculate μ & σ yourself & scale the data manually

1. has the advantage that it can be chained into **pipelines** in sklearn and keeps track of things for you. But if your data are badly affected by outliers it will not detect that for you.

2. has the advantage that you know what you are doing, and if you want to use a robust estimator for the mean or variance you can. The downside is that you need to keep track of the values and remember to apply them!

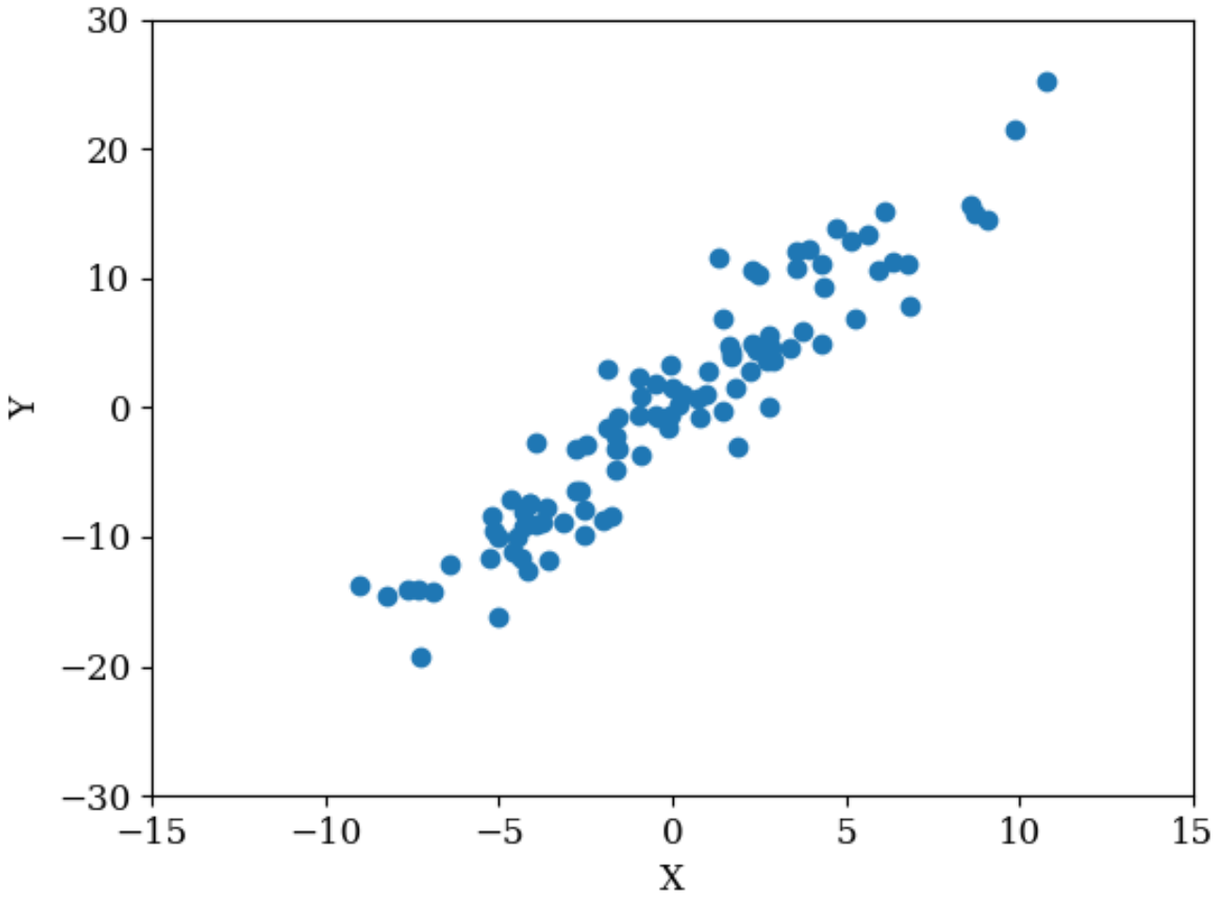
Standardized data



Principal Component Analysis (PCA)

To see the code for the simple example:
`Lectures/Lecture 4/Notebooks/Simple PCA example.ipynb`

An introductory example



Start with a dataset with x & y values.

Step 1: Subtract the means.

We now want to find the direction along which the data vary most.

What direction?

$$\mathrm{Cov}(i,j) = \frac{1}{N}\sum_{n=1}^{N}\left(v_{i,n} - \langle v \rangle\right)\left(v_{j,n} - \langle v_j \rangle\right)$$

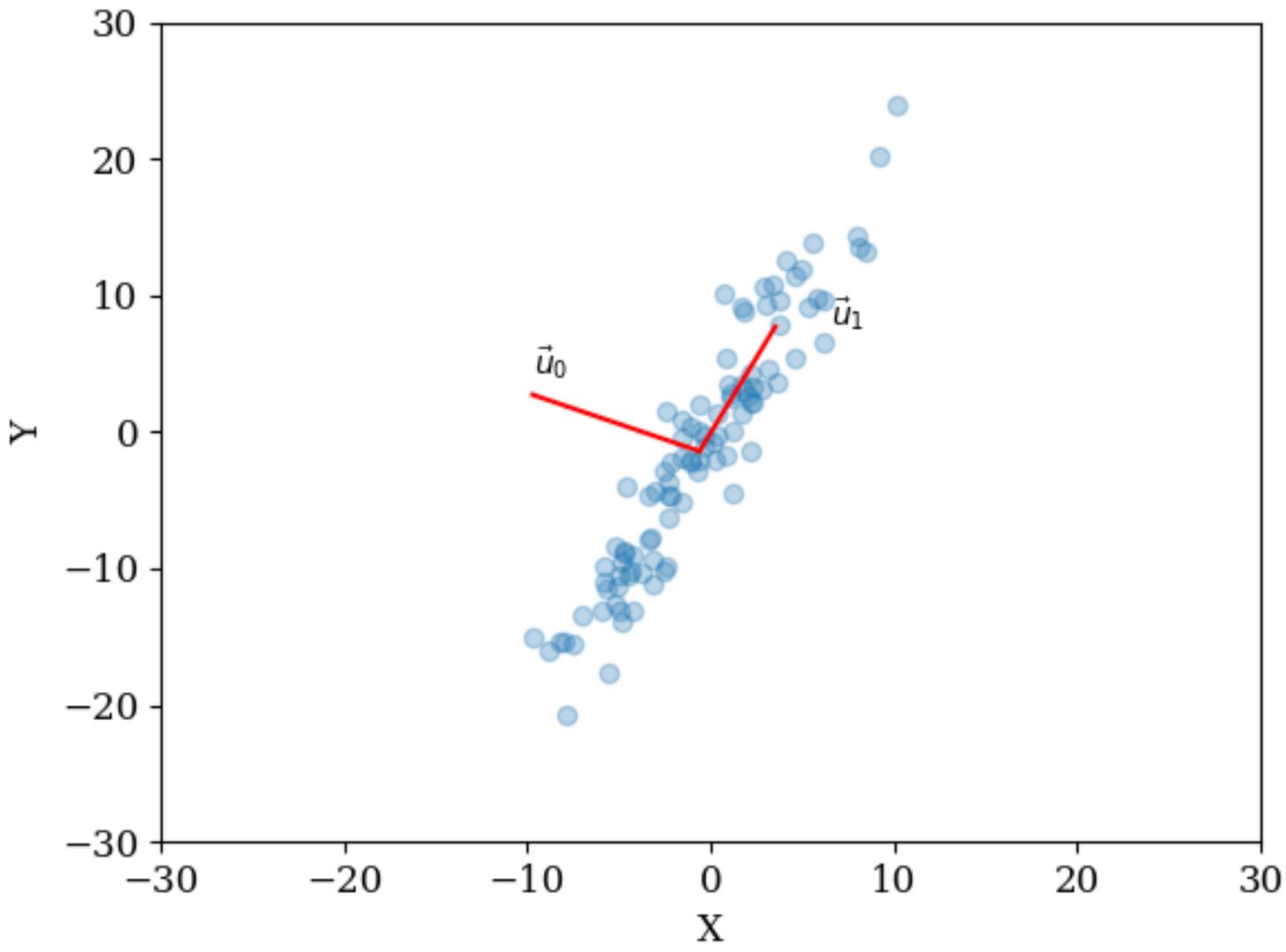
18.55	37.26
37.26	83.77

Then: We want to rotate this so that it becomes diagonal.

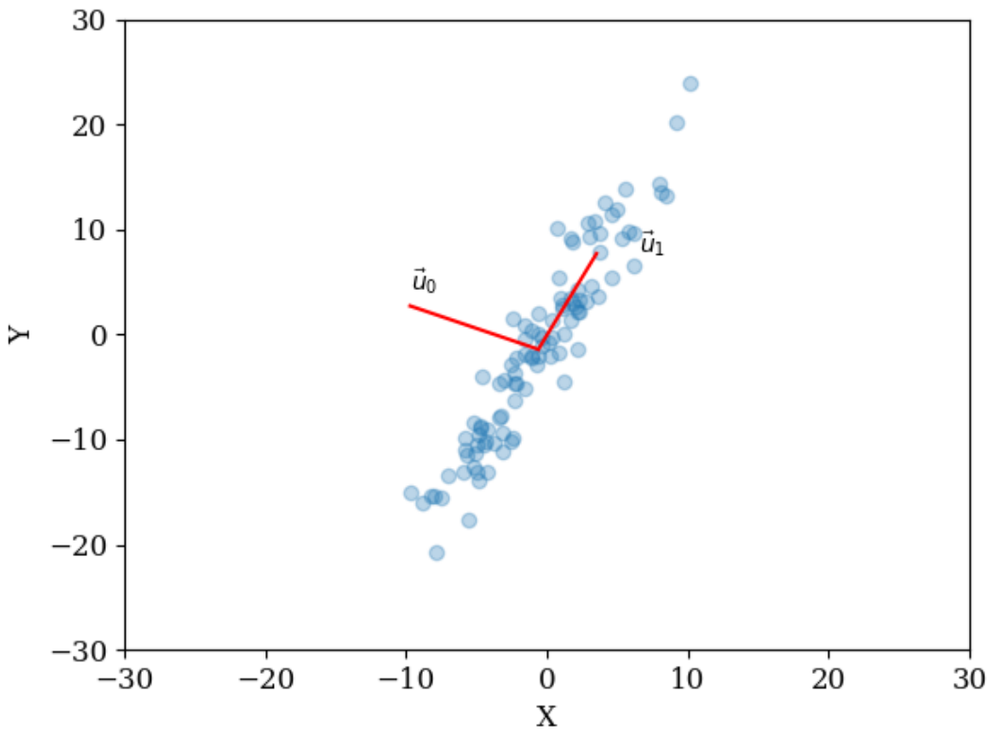
Eigenvectors & Eigenvalues

```
numpy: np.cov(x, y)
```

The two eigenvectors



The two eigenvectors



A very standard calculation and all main programming languages support this easily.

Python:

```
import numpy as np
C = np.cov(x, y)
e_vals, e_vecs = np.linalg.eigh(C)
```

Julia:

```
using LinearAlgebra, Statistics
C = cov(hcat(x, y))
e_vals, e_vecs = eigen( C)
```

R:

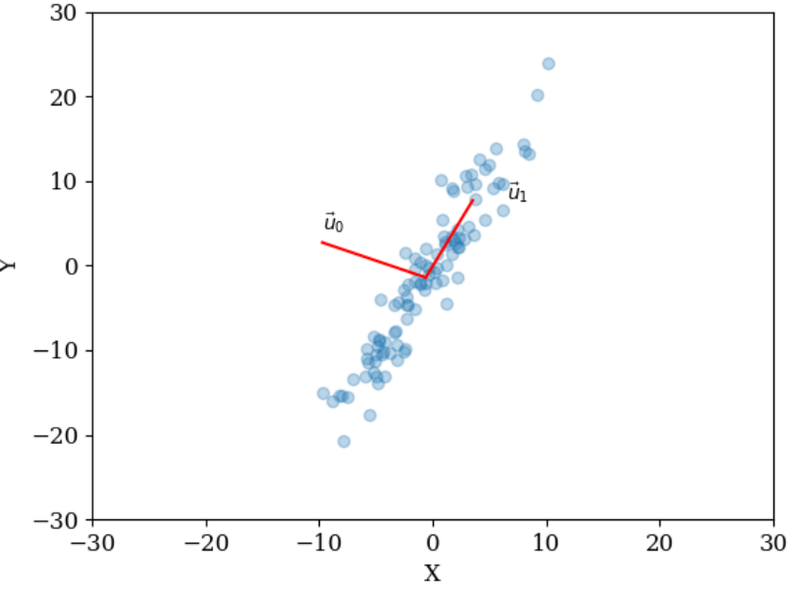
```
C = cov(cbind(x, y))
e = eigen( C)
```

IDL:

```
M = transpose([[x], [y]])
C = correlate(M, /covar)
evals = eigenql(C, eigenvectors=evecs)
```

How do you do it?

Interpreting the results



The eigenvector corresponding to the largest eigenvalue defines the direction of most variation in the data ($\mathbf{u}_1$).

So we could say: I do not care about the little scatter around this line, let me reduce the data to just 1D.

How do you do that?

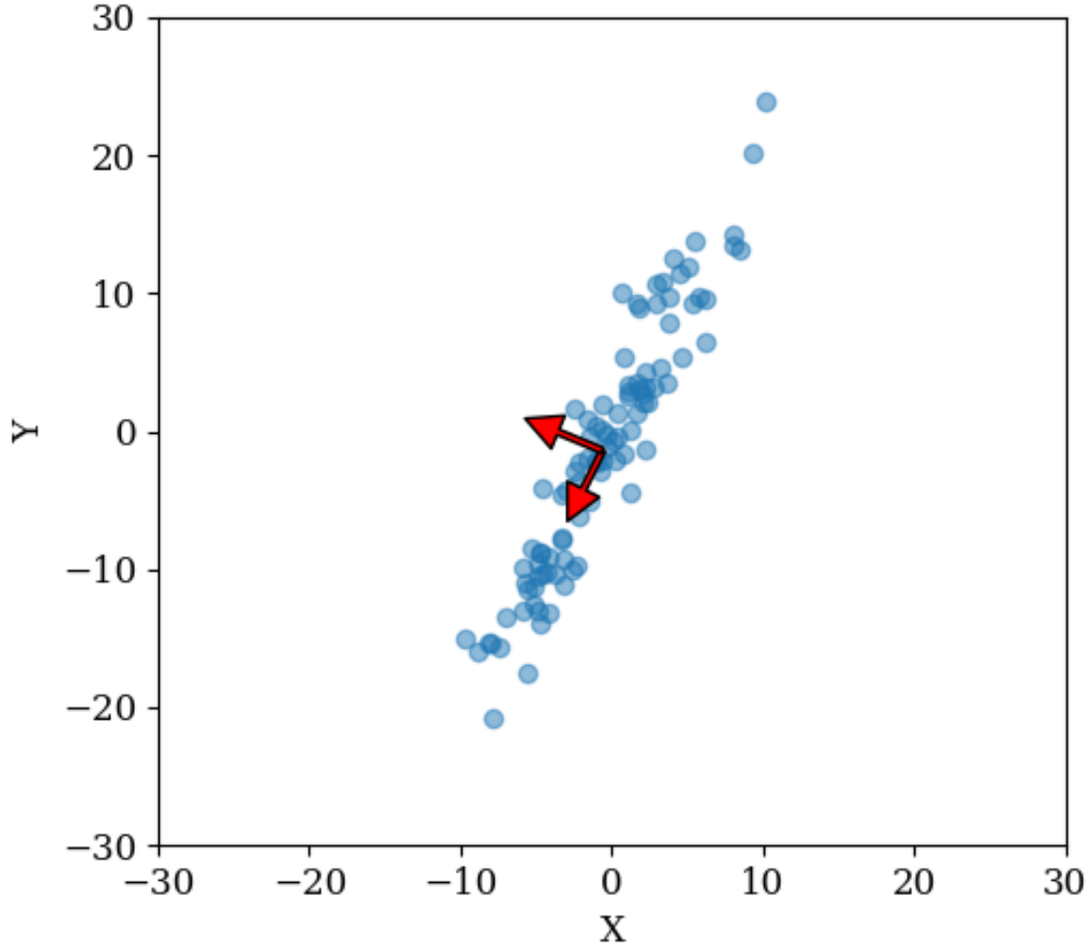
Projecting the solution

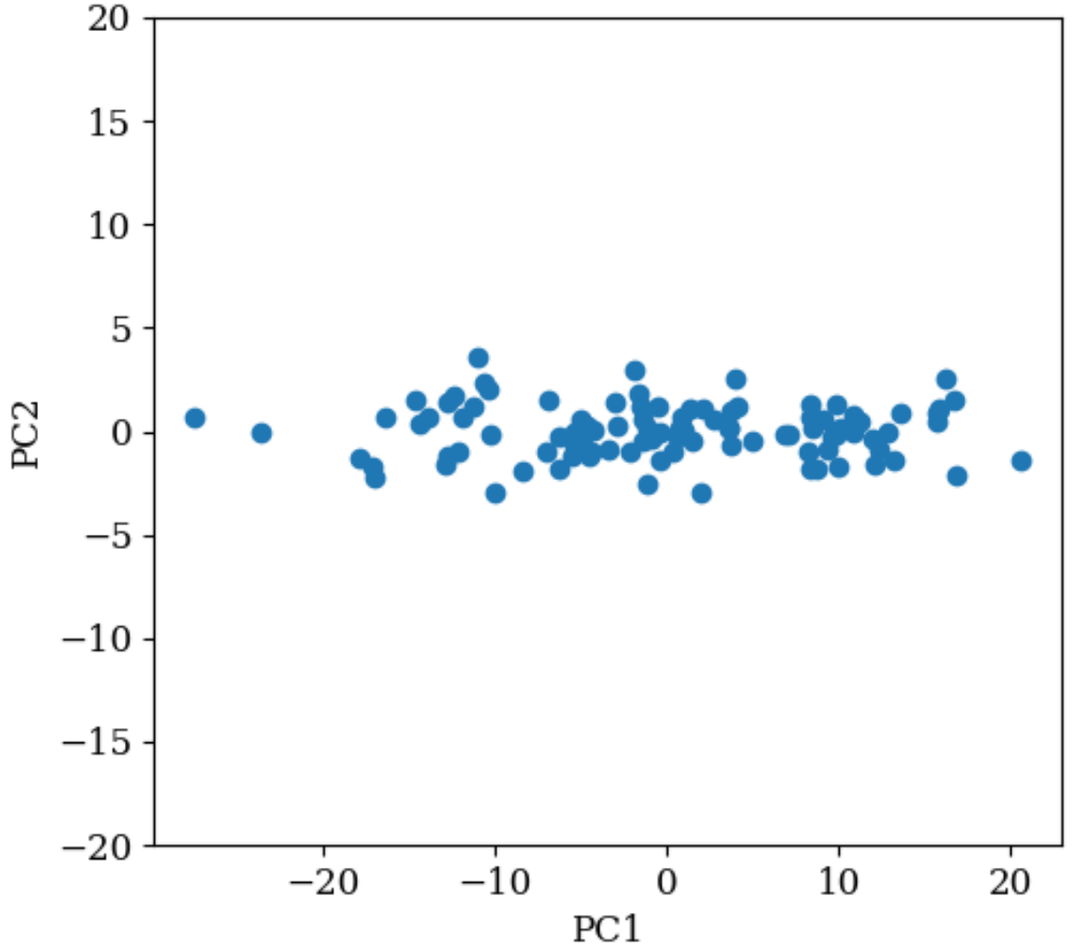
Note: The direction of the axes is arbitrary!

We want to project:

$$(x, y) \rightarrow (u, v) \text{ using the eigenvectors.}$$

$(u, v) = \mathsf{E}(x, y)^T$ where **E** is the matrix of eigenvectors





Dimension reduction

Until now we have considered a situation where we have 2 variables, x & y, and we recover two variables PC1 & PC2. We can also reduce the dimensionality of our problem by reconstructing the original data using only the first eigenvector.

First: Take the new variable u = PC1:

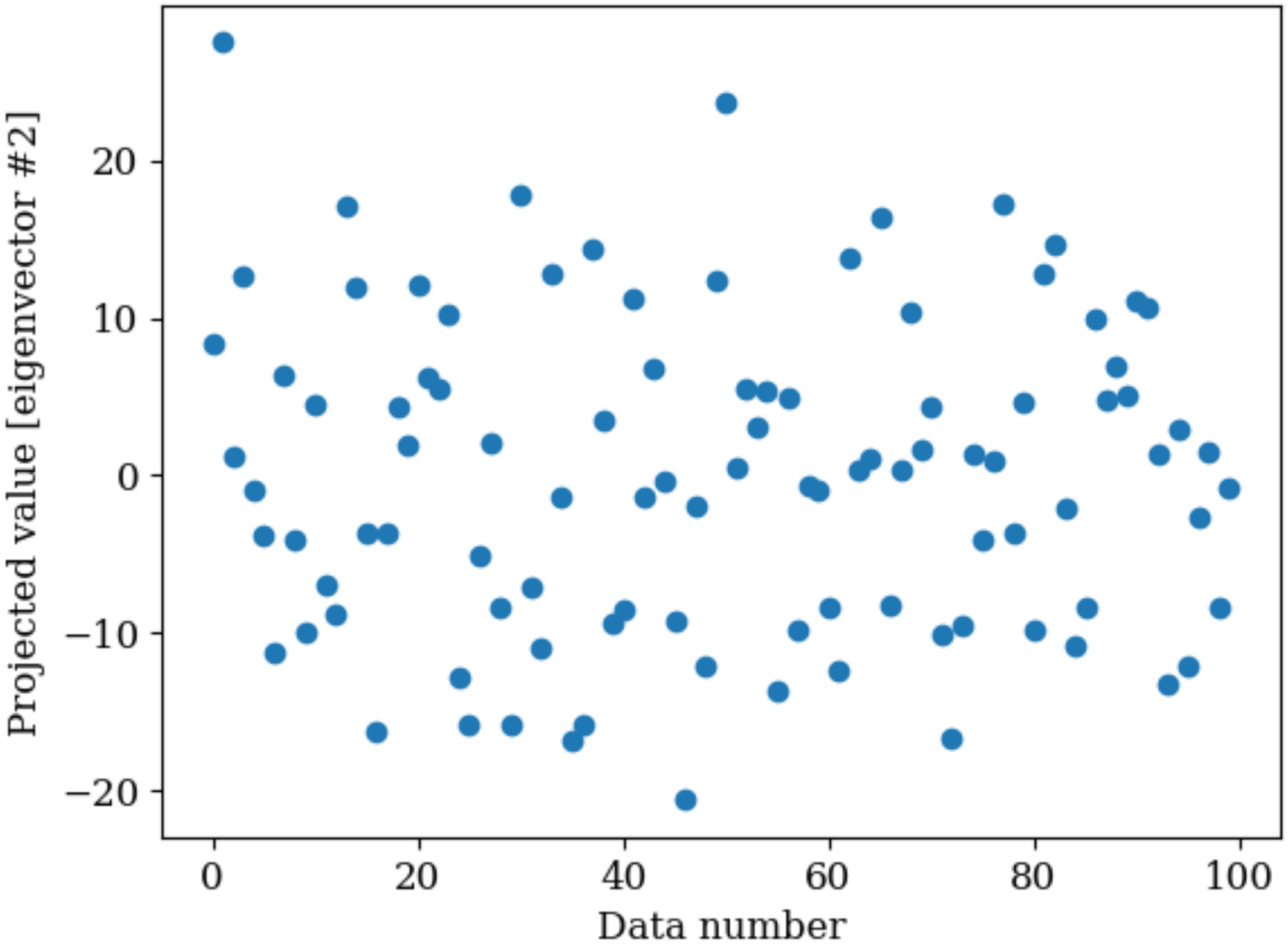
$$u_i = (x_i, y_i)\mathbf{x}\begin{pmatrix} e_{1,x} \\ e_{1,y} \end{pmatrix}$$

Then project back to the original space:

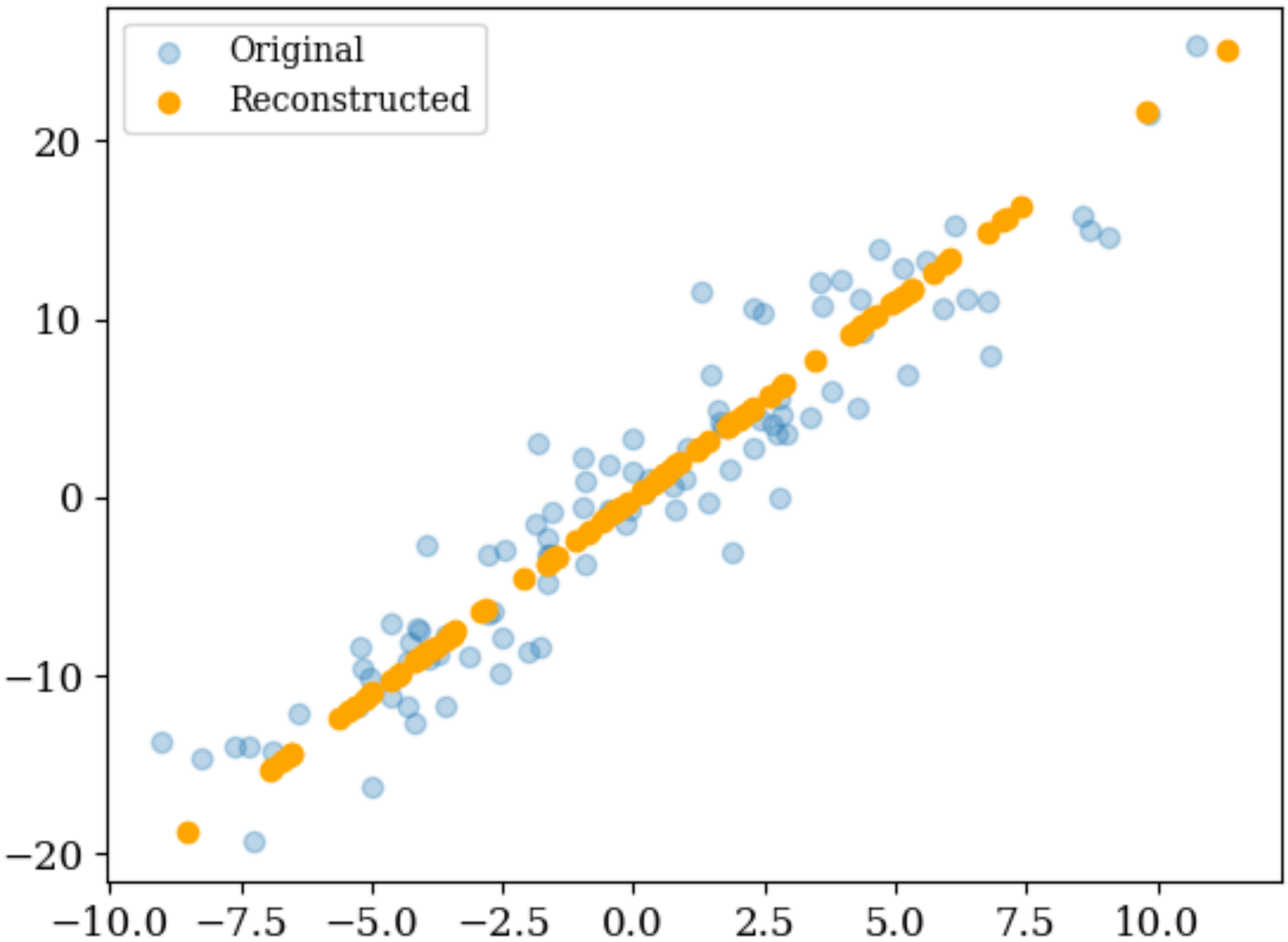
$$(\tilde{x}_i, \tilde{y}_i) = u_i(e_{1,x}, e_{1,y})$$

Dimension reduction

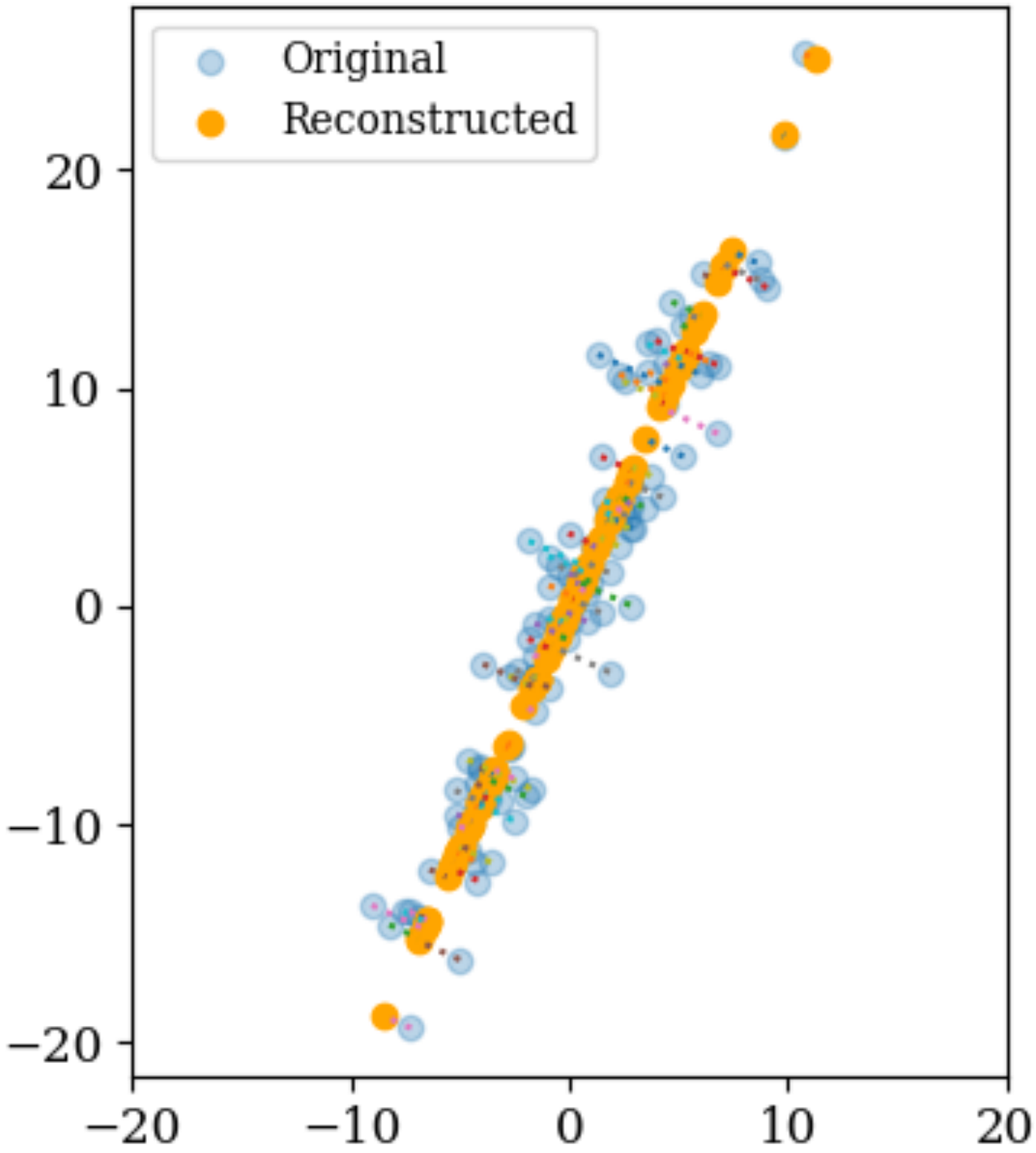
The result of projecting onto the first eigenvector



Dimension reduction



Dimension reduction - orthogonal



This shows fairly clearly that the projection is orthogonal.

While this is not a regression technique, it finds a best line treating x and y equally.

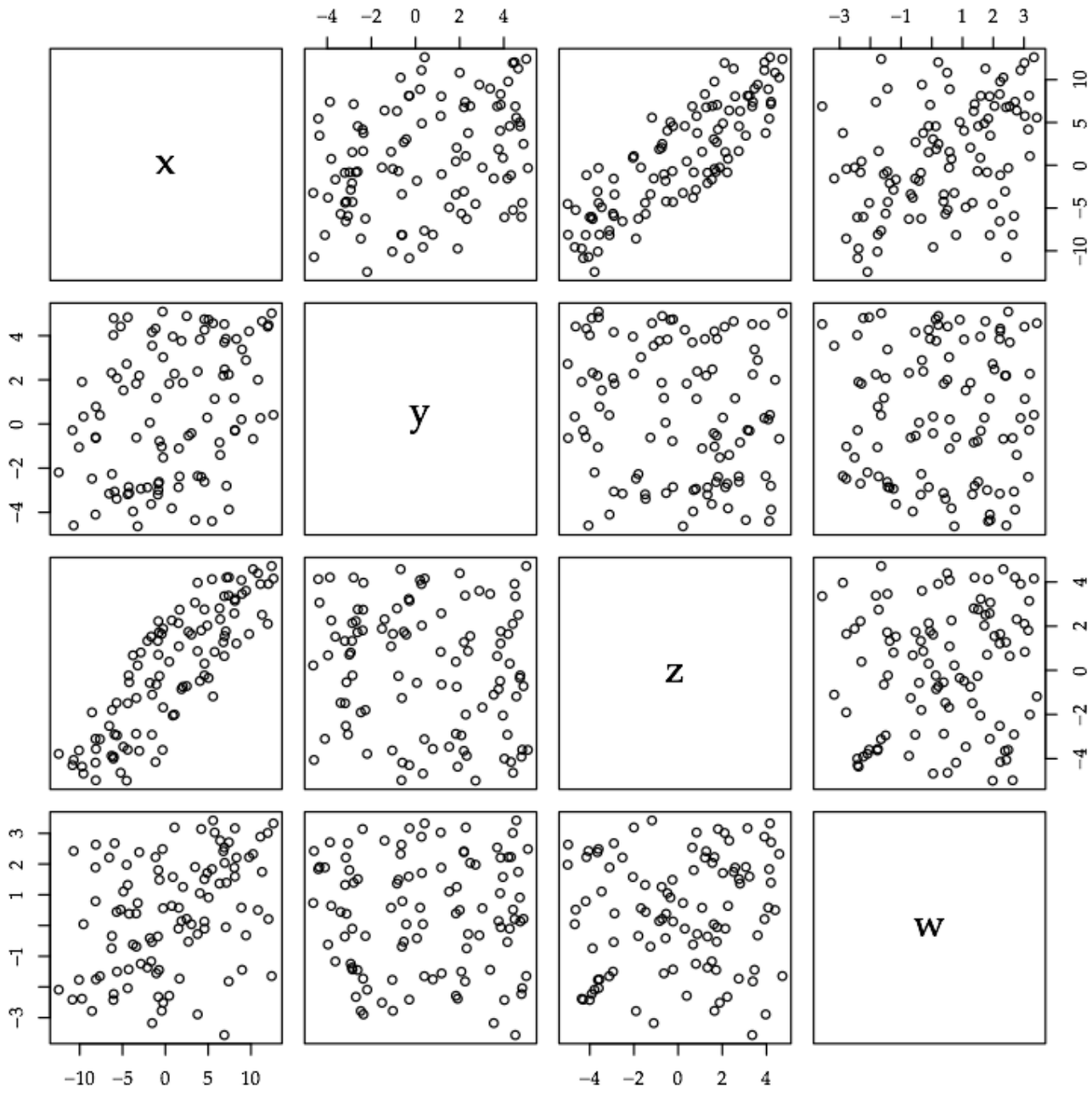
Looking at variances

So PCA gives us a way to find directions where the variance is large. It is therefore possible to ask how much of the variance is **explained** by particular eigenvectors.

The variance is given by the diagonal elements in the diagonalised covariance matrix (the sum of variances is independent of coordinate system).

Looking at variances

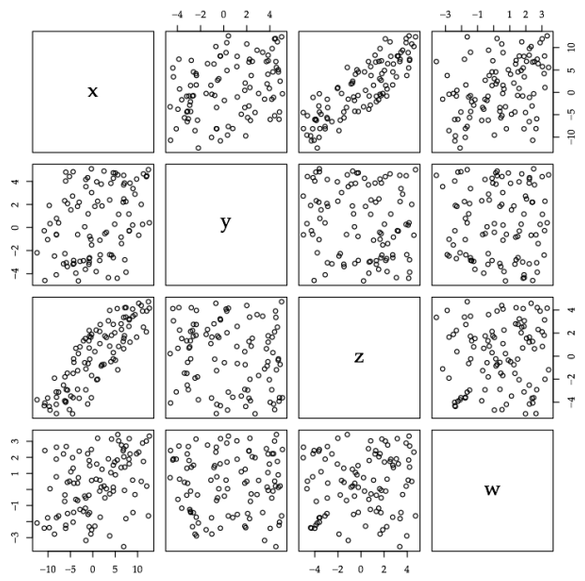
How much of the variance is **explained** by particular eigenvectors.



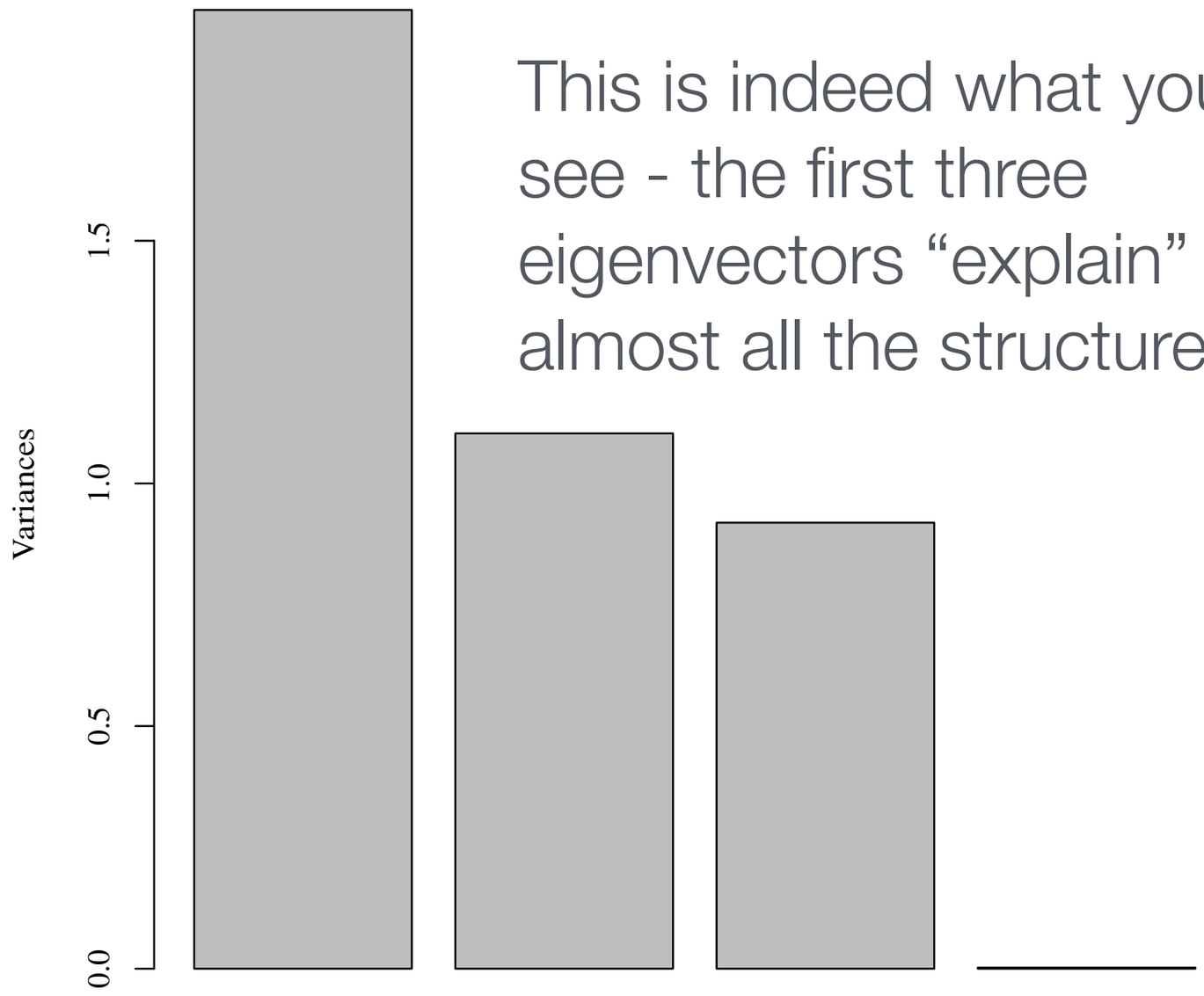
4D dataset but we see clearly a correlation between x & z so they probably add much the same information.

Looking at variances

How much of the variance is **explained** by particular eigenvectors.



4D dataset but we see clearly a correlation between x & z so they probably do not add a lot of information.



This is indeed what you see - the first three eigenvectors “explain” almost all the structure.

More Generally

We have a set of measurements: $\{x_n\}_{i=1}^{N}$

And let us now transform these measurements:

$$\mathbf{y}_n = \mathbf{M}\mathbf{x}_n$$

We would like these **y** to have nice properties. One reasonable requirement would be that they are (linearly) independent. We can formulate this in terms of their covariance matrix:

$$C_y = \frac{1}{N}\sum_i \left(y_i - \langle y \rangle\right)\left(y_i - \langle y \rangle\right)^T$$

Requiring the **y** values to be independent is equivalent to ensuring the covariance matrix being diagonal.

What does that mean?

Diagonalizing the covariance matrix

$$\mathrm{C}_{\mathbf{y}} = \frac{1}{N}\sum_i (\mathbf{y_i} - \langle \mathbf{y} \rangle)(\mathbf{y_i} - \langle \mathbf{y} \rangle)^T$$

$$\mathrm{C}_{\mathbf{y}} = \frac{1}{N}\sum_i M(\mathbf{x_i} - \langle \mathbf{x} \rangle)(M(\mathbf{x_i} - \langle \mathbf{x} \rangle))^T$$

$$\mathrm{C}_{\mathbf{y}} = \frac{1}{N}\sum_i M\boxed{(\mathbf{x_i} - \langle \mathbf{x} \rangle)(\mathbf{x_i} - \langle \mathbf{x} \rangle)^T} M^T$$

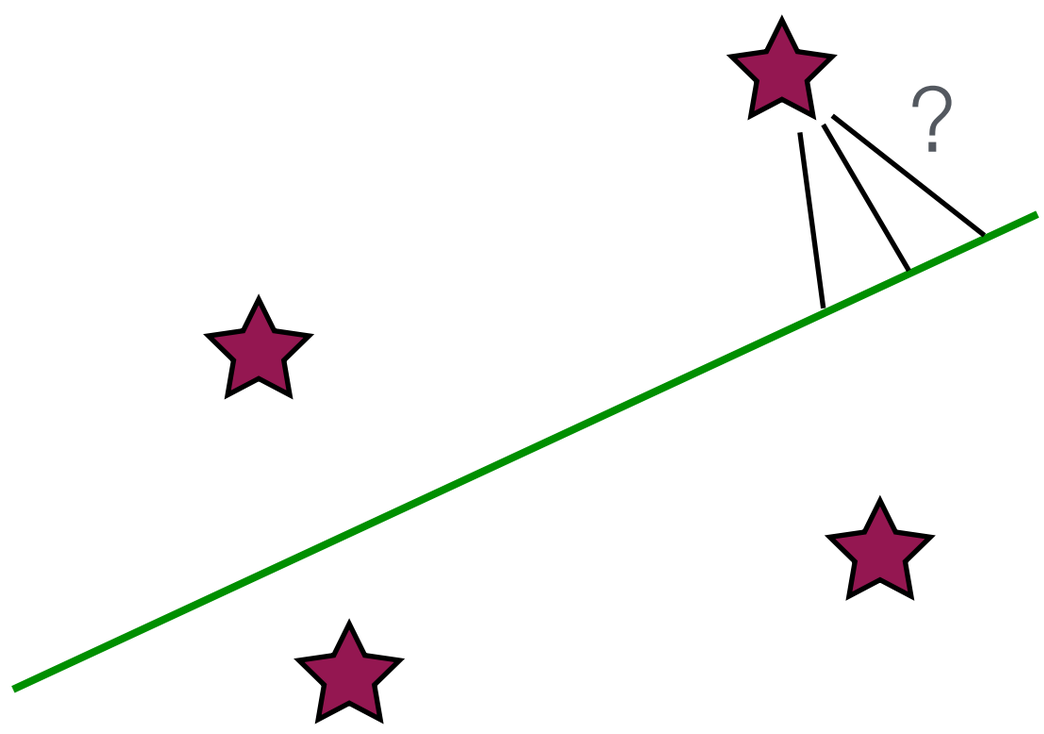
$$C_{\mathbf{y}} = M C_{\mathbf{x}} M^T$$

So by finding the matrix that diagonalizes the covariance matrix of the data, we find a new set of uncorrelated variables.

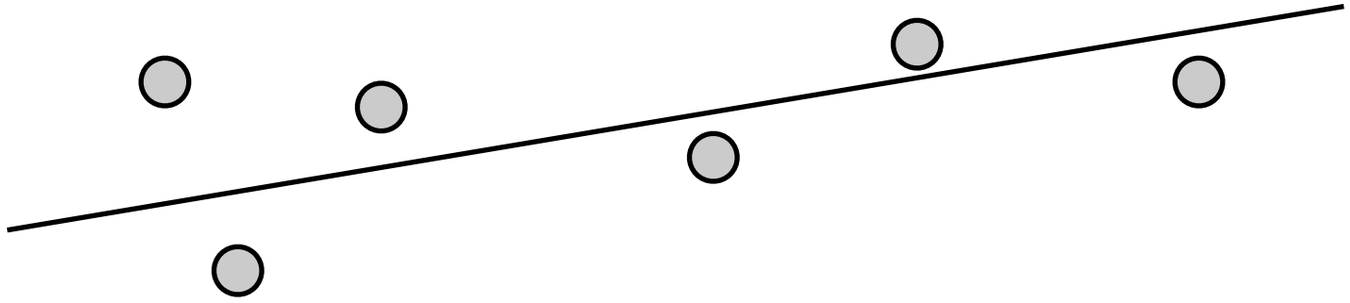
A different view

The data, $\mathbf{x}_n$, is D-dimensional, where D can be a large number (several thousand easily for images).

Let us now ask - what is the “best” way to project it down to a lower-dimensional space with M dimensions?



What is the “best” way to project D-dimensional data to a lower-dimensional space with M dimensions?

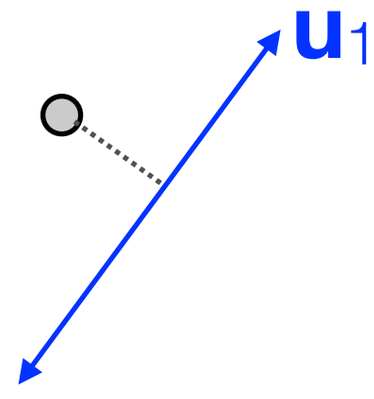


Going from 2D to 1D it would be along this line because it “explains” the data in the simplest way. Reformulating it we say that we want the direction where the data has maximal **variance**.

So from D dimensions to 1 dimension:

We project $\mathbf{x}_n$ onto a vector $\mathbf{u}_1$:

$$\mathbf{y}_n = \mathbf{u}_1^T \mathbf{x}_n$$
$$\mathbf{u}_1^T \mathbf{u}_1 = 1$$



So we want to maximise the variance of $\mathbf{y}$:

$$\mathrm{Var}[\mathbf{y}] = \frac{1}{N}\sum_{n=1}^{N}\left(\mathbf{y}_n - \langle \mathbf{y} \rangle\right)^2$$

Which can be rewritten:

$$(\mathsf{A}-\mathsf{B})^2 = (\mathsf{A}-\mathsf{B})(\mathsf{A}-\mathsf{B})^T$$
$$(\mathsf{A}\mathbf{x})^T = \mathbf{x}^T\mathsf{A}^T$$

$$\mathrm{Var}[\mathbf{y}] = \mathbf{u}_1^T \mathsf{S} \mathbf{u}_1 = \mathbf{u}_1^T \mathsf{C}_{\mathbf{x}} \mathbf{u}_1$$

With

$$\mathsf{S} = \mathsf{C}_{\mathbf{x}} = \left(\mathbf{x} - \langle \mathbf{x} \rangle\right)\left(\mathbf{x} - \langle \mathbf{x} \rangle\right)^T$$

$$\mathrm{Var}[\mathbf{y}] = \mathbf{u}_1^T \mathsf{S} \mathbf{u}_1 = \mathbf{u}_1^T \mathsf{C}_\mathbf{x} \mathbf{u}_1$$

So to maximise this with the constraint $\mathbf{u}_1^T \mathbf{u}_1 = 1$:

$$V = \mathbf{u}_1^T \mathsf{C}_\mathbf{x} \mathbf{u}_1 + \lambda_1 \left(1 - \mathbf{u}_1^T \mathbf{u}_1\right)$$

Set $\partial V / \partial \mathbf{u}_1 = 0$

$$\mathsf{C}_\mathbf{x} \mathbf{u}_1 = \lambda_1 \mathbf{u}_1 \Rightarrow \mathbf{u}_1^T \mathsf{C}_\mathbf{x} \mathbf{u}_1 = \lambda_1$$

Hence the variance is maximised in 1D if $\mathbf{u}_1$ is set to the largest eigenvector of the covariance matrix.

Projecting

So, now we have all the eigenvectors & eigenvalues, how do we calculate the new coordinates?

a) Create a “feature vector”/transform matrix by placing the eigenvectors in a matrix where each row is an eigenvector. $\mathbf{M}$

b) Transform the data to the new coordinates using this matrix

$$\mathbf{u} = \mathbf{M}\mathbf{x}$$

c) Transform back using the transpose of **M**:

$$\mathbf{r} = \mathbf{u}\mathbf{M}^T$$ (+ offsets & scalings applied to **r**)

Step-by-step

- If necessary, transform data to have zero mean and usually (but not always) unit variance.
- Calculate the covariance matrix of the data, $\mathbf{C_x}$.
- Find the eigenvalues & eigenvectors of the covariance matrix.
- Decide which eigenvalues are worth keeping and put these into a feature matrix, **M** - if you just want a rotation of the coordinate system, keep all!
- Use **M** to calculate the new coordinates.
- If appropriate, use **M** to go back to the original coordinate system, undoing the scaling and shift if applied.

Python: sklearn.decomposition.PCA

R: prcomp
Julia: MultivariateStats.PCA
IDL: pcomp

`Simple example: Lectures/Lecture 4/Notebooks/Simple PCA example.ipynb`
`More advanced: Lectures/Lecture 4/Notebooks/PCA of Pickles.ipynb`

Some general issues

- The sign (direction) of an eigenvector is arbitrary.
- A covariance matrix depends on the scaling of the variables. This can be really important!
- New variables should be defined using normalised eigenvectors (not always done by computer packages, e.g. IDL)
- PCA is not a magic bullet. You must think carefully when using it!

SCALING

If we look at one element in the covariance matrix:

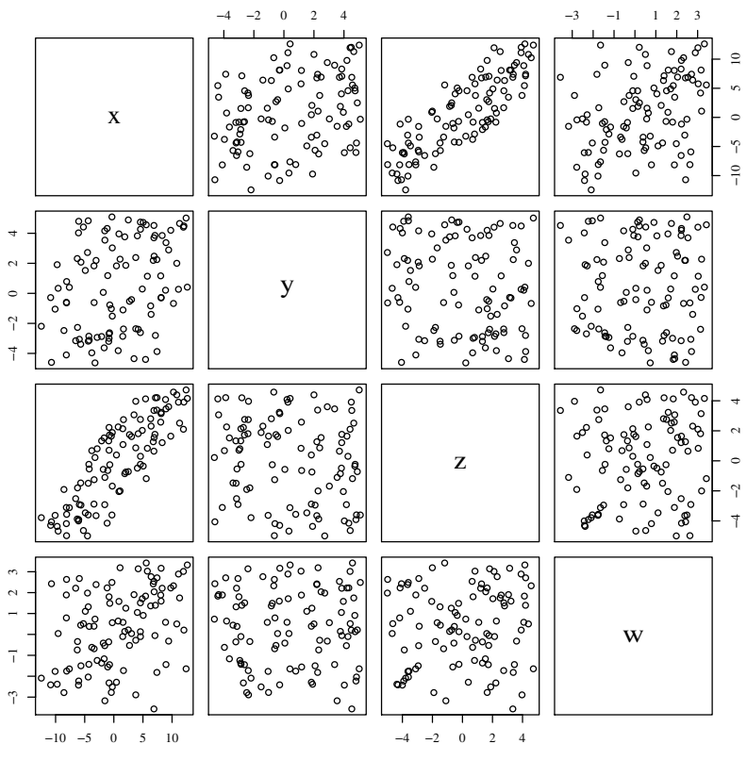
$$\mathrm{Cov}(i,j) = \frac{1}{N}\sum_{n=1}^{N}\left(v_{i,n} - \langle v_i \rangle\right)\left(v_{j,n} - \langle v_j \rangle\right)$$

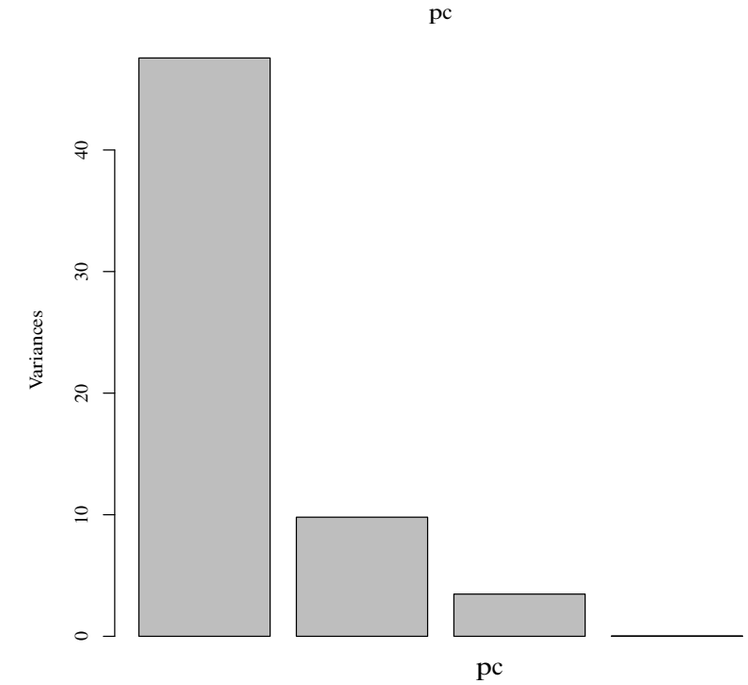
Clearly if we scale all y by the same factor we can make Cov(i, j) arbitrarily large or small. This means that the *eigenvectors* also can be arbitrarily changed!

Note also that the mean is subtracted off - it is usually a good idea to do that ourselves. Sometimes people recommend also scaling by their empirical variance (this can be be bad if you are careless). This is standardizing we saw earlier. For PCA this might be a bad idea but it depends on the question.

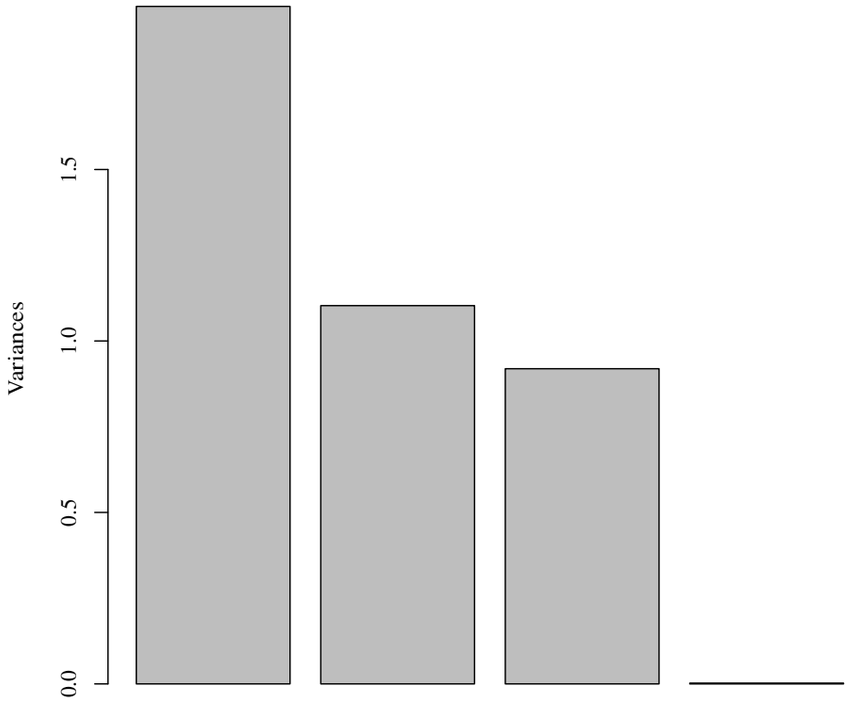
SCALING

But sometimes scaling is crucial to take out trivial features in the data!





No scaling



With scaling

SCALING

The physical meaning of this:

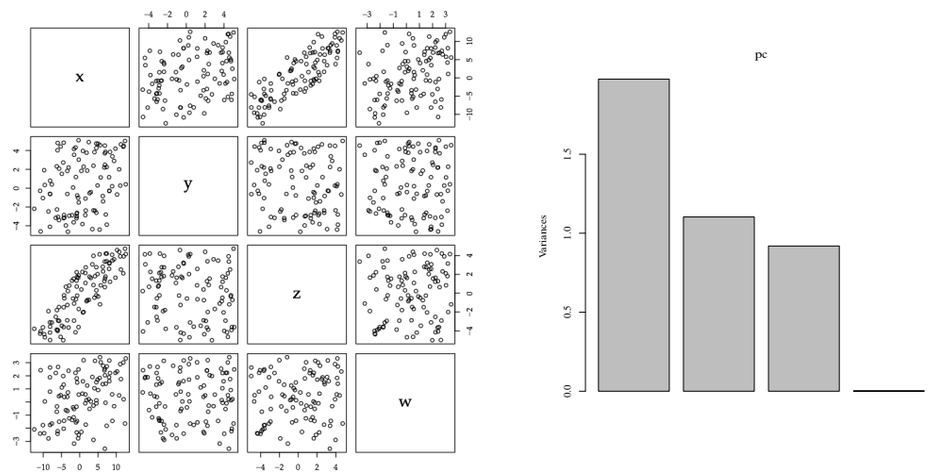
We might have a system where we know that x & y are correlated, say y increases a bit when x increases. This produces a non-zero covariance between the two variables. However we might know that this is an unimportant physical correlation (y is mostly governed by something else).

What would normalising by the spread in x & y mean?

Often a good alternative: Work with relative quantities/intrinsic quantities rather than absolute/extrinsic quantities.

Uses of PCA

I. Find the effective dimensionality of your data:



2. Identify useful features of your data (images, spectra, etc.) - both for feature extraction and science.

3. Compress your high-dimensional dataset to fewer variables to make fits to models easier. [pre-processing]

4. Find linear relations in your data while treating all variables on the same footing (no need to assume an independent variable)

Trying it out in Python

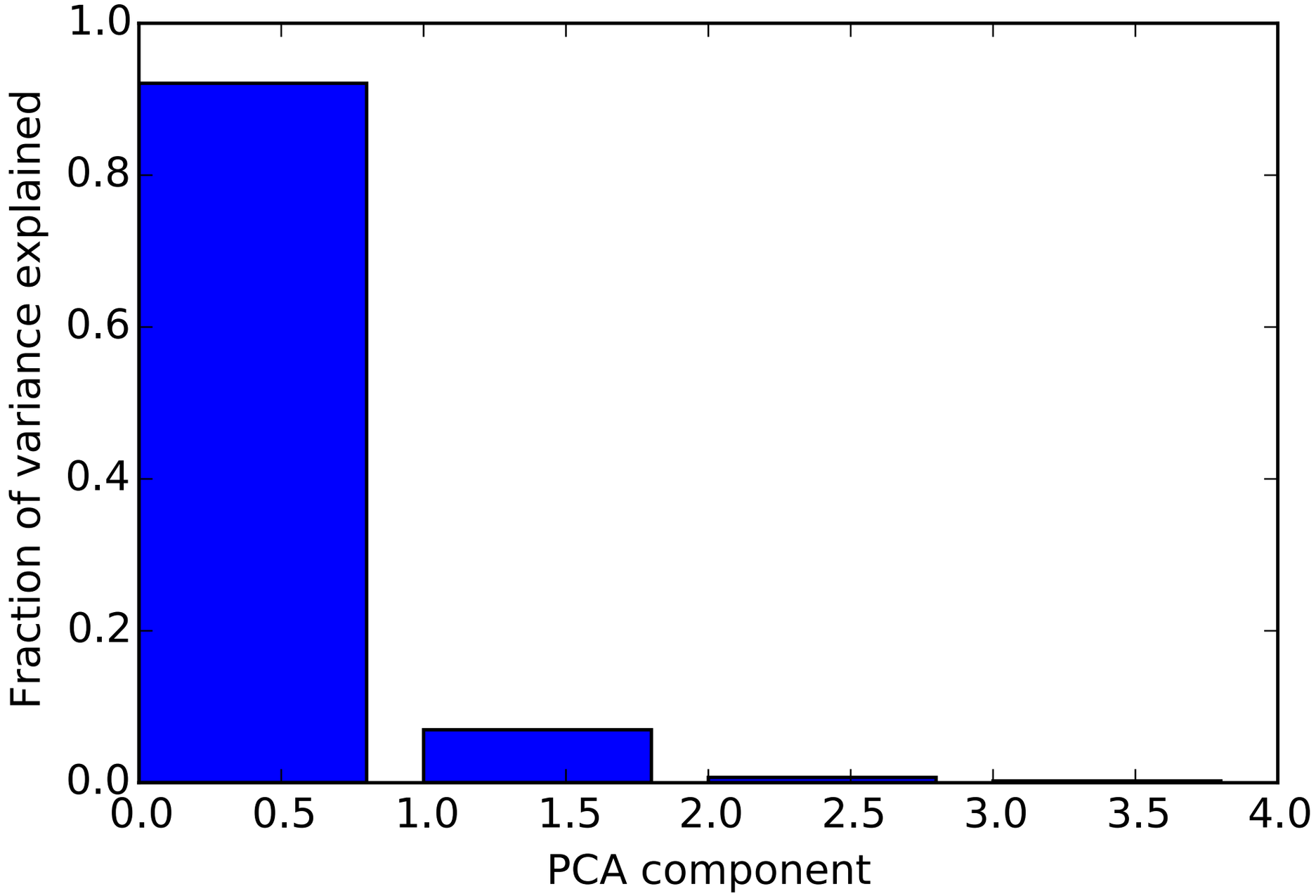
```
from sklearn.decomposition import PCA
M, T = pickle_from_file('T-vs-colour-regression.pkl')
pca = PCA(whiten=True, n_components=4)
pca.fit(M)

pca.components_
```

```
plt.bar(np.arange(len(pca.explained_variance_)),
pca.explained_variance_ratio_)
```

From the PCA components I get that

$$1.84\,(u-g) + 1.1\,(g-r) + 0.39\,(r-i) + 0.28\,(i-z)$$

explains most of the variance.

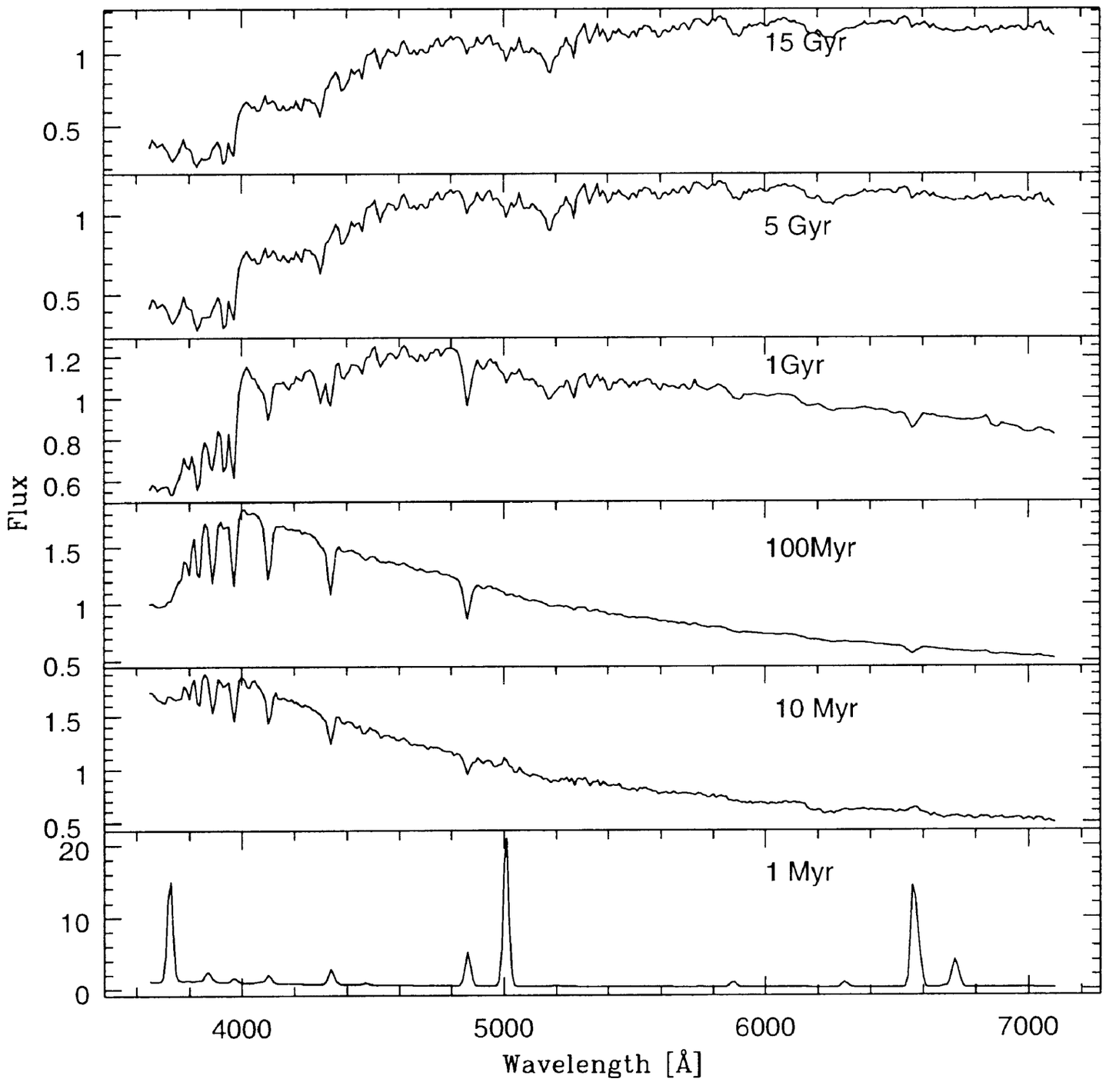



```
plt.bar(i_comp, pca.explained_variance_ratio_)
```

Some further applications of PCA

Classifying galaxy spectra using PCA

Ronen et al (1999)



Observed galaxy spectra show a wide range of structure.

They are composed of stars of different ages etc - can we disentangle them using PCA?

Classifying spectra using PCA

A spectrum is a function of wavelengths: $f(\lambda_i)$

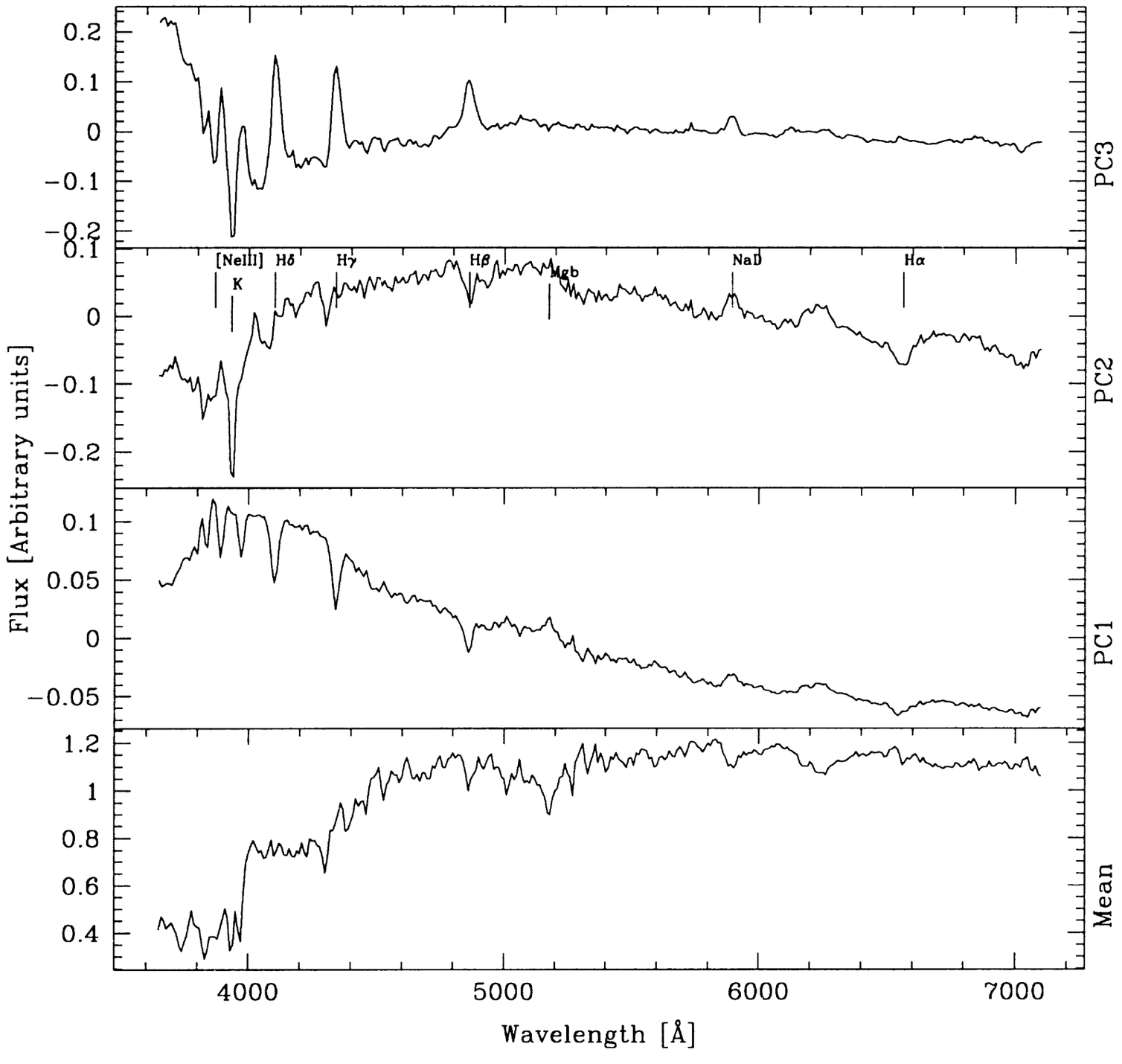
But you could just as well think of this as a collection of numbers: $\vec{x} \quad \text{where} \quad x_i = f(\lambda_i)$

Then your spectrum with N wavelength points become a vector in N dimensional space and you can use PCA if you have a set of spectra!

1. Calculate the covariance matrix.
2. Calculate eigenvalues & eigenvectors.
3. Determine the important eigenvectors.

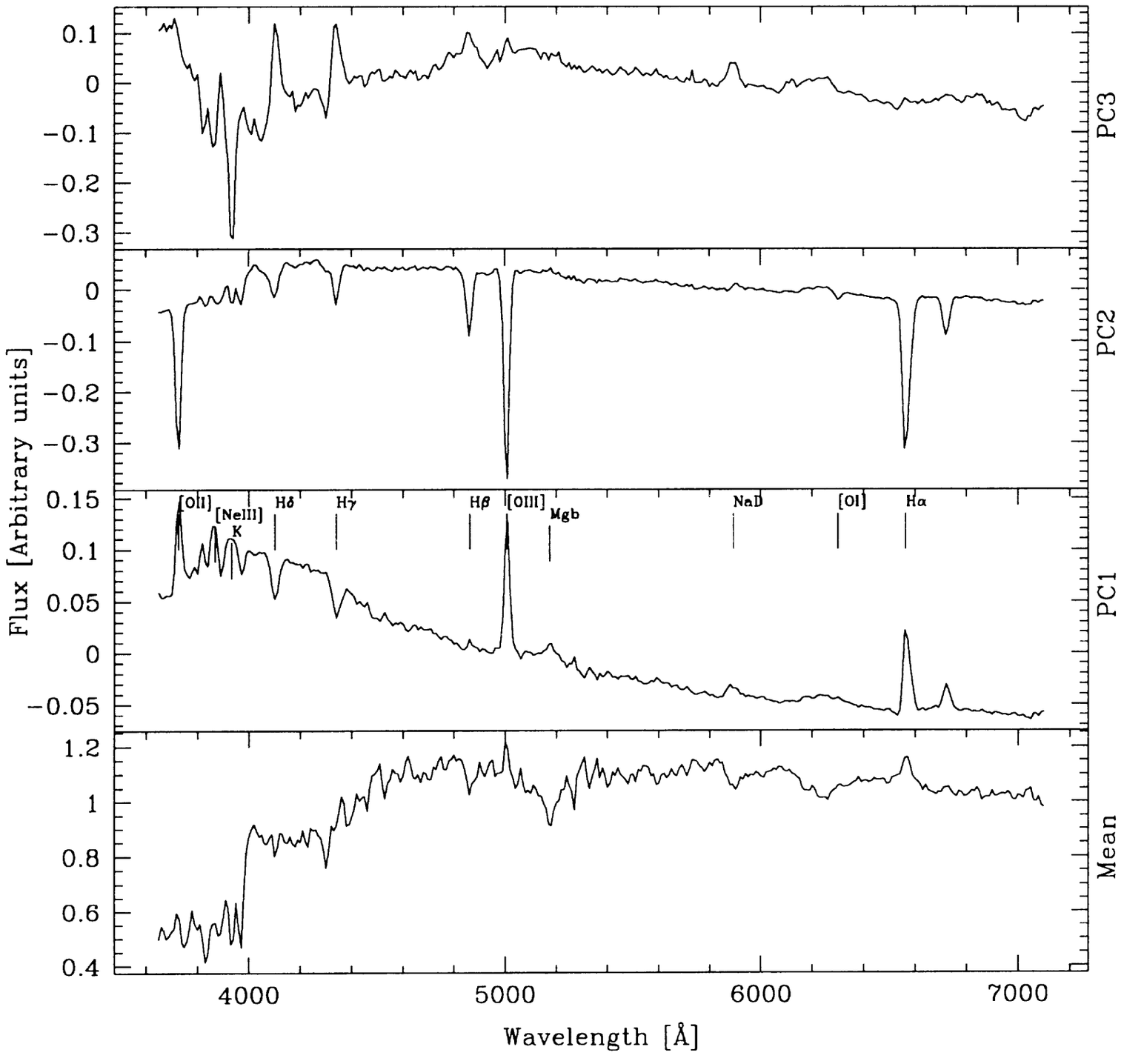
Classifying galaxy spectra using PCA

Ronen et al (1999)



Classifying galaxy spectra using PCA

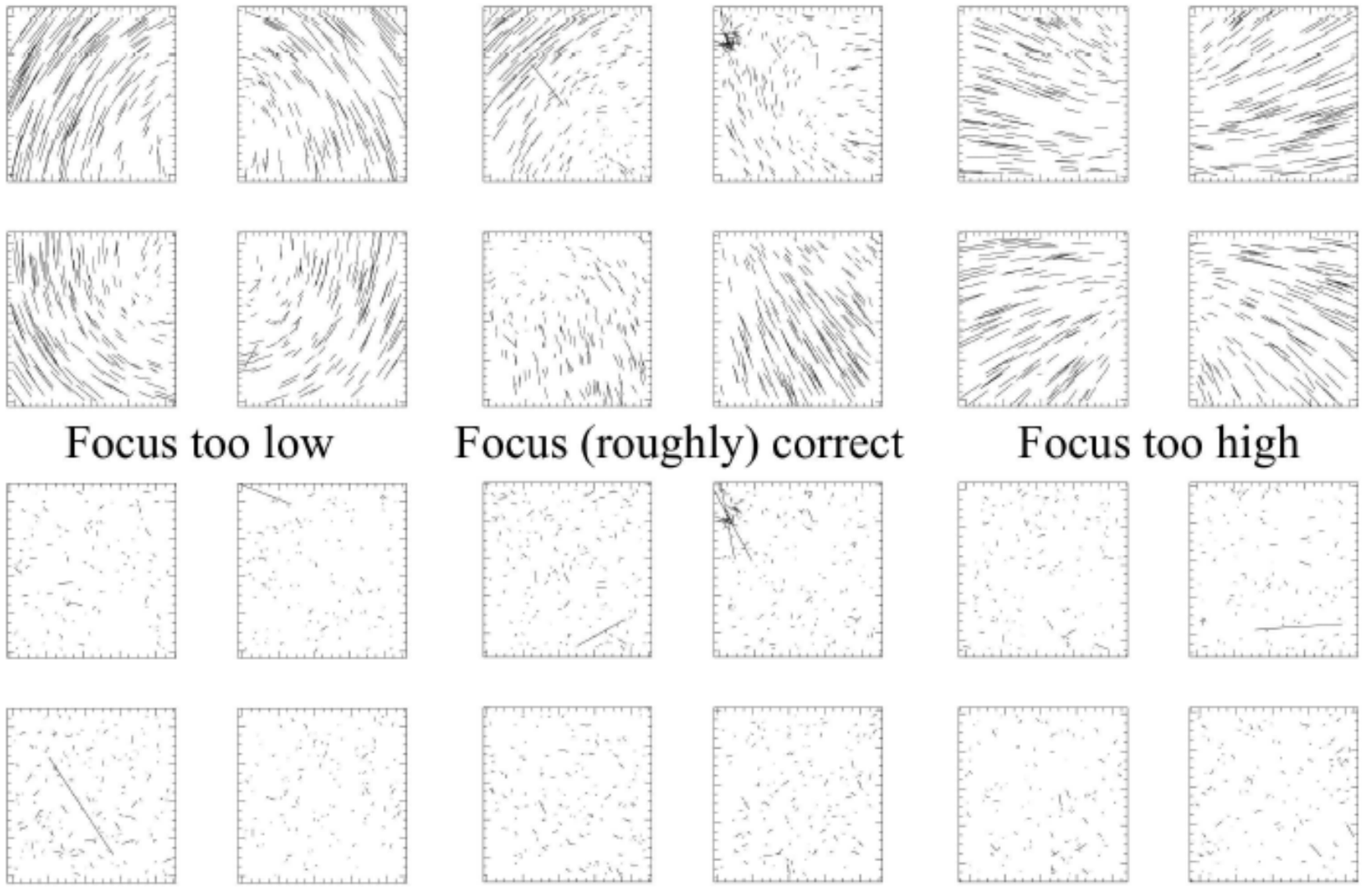
Ronen et al (1999)



A very young system

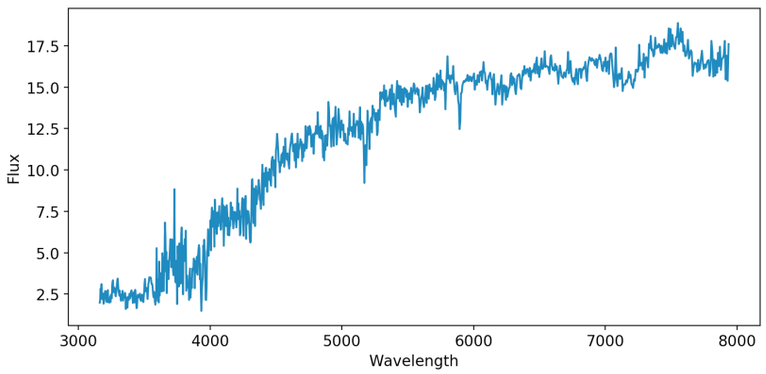
PCA for correcting point-spread functions

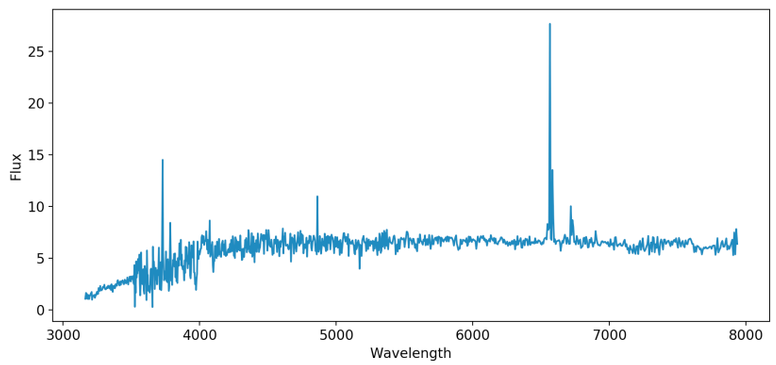
Weak lensing techniques for probing the structure of the Universe are very sensitive to the PSF. The true variation of the PSF can be very complex - PCA decomposition allows you to focus on the important features



Jarvis & Jain (2004), see also Schrabback et al (2010)

Handling of high-D data





Say you want to create a grid of these spectra for fitting - how would you do that?

One possibility: PCA

$$f(\lambda_i) = \sum_i \omega_i(\theta)\Phi_i(\lambda)$$

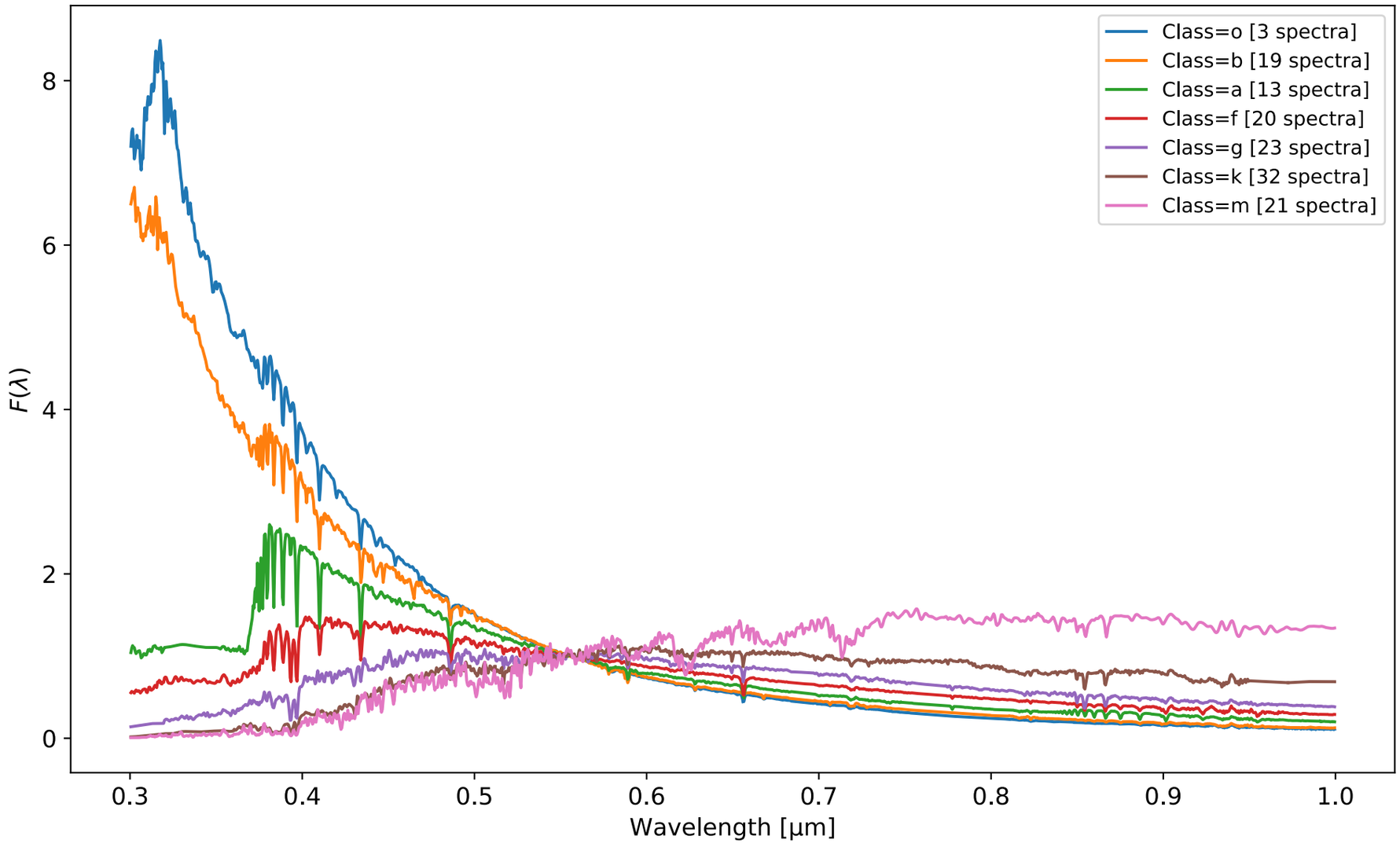
Eigenvectors

PCA weights

One can then truncate this expansion and have a few (3-10) eigen components which can be put on a grid.

PCA of stellar spectra

Starting point: Pickle's library of stars @ 5Å resolution



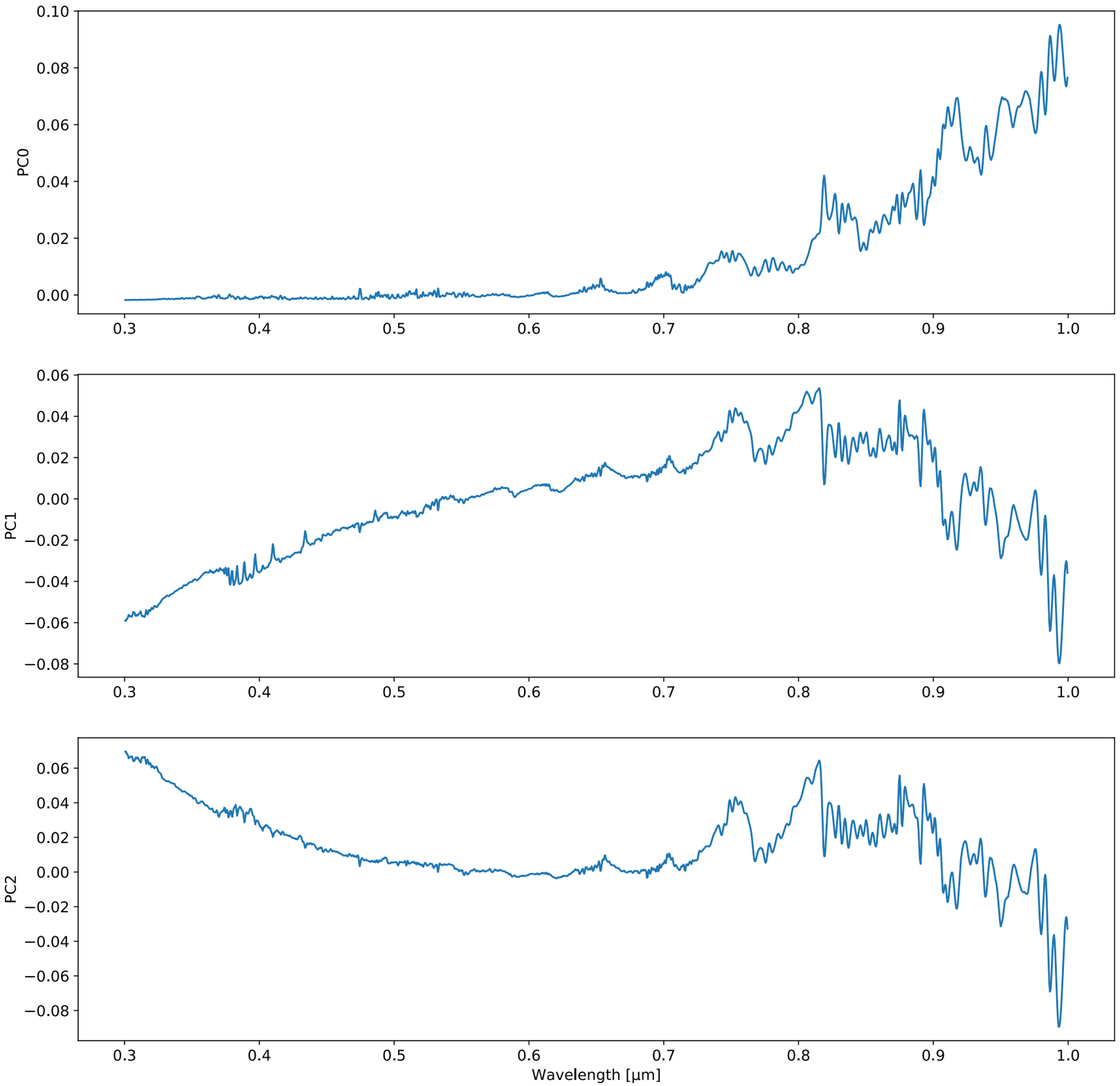
PCA of stellar spectra

```
X = fluxes of all stars stacked (131x4771)
```

So this has 131 examples in 4771-dimensional space.

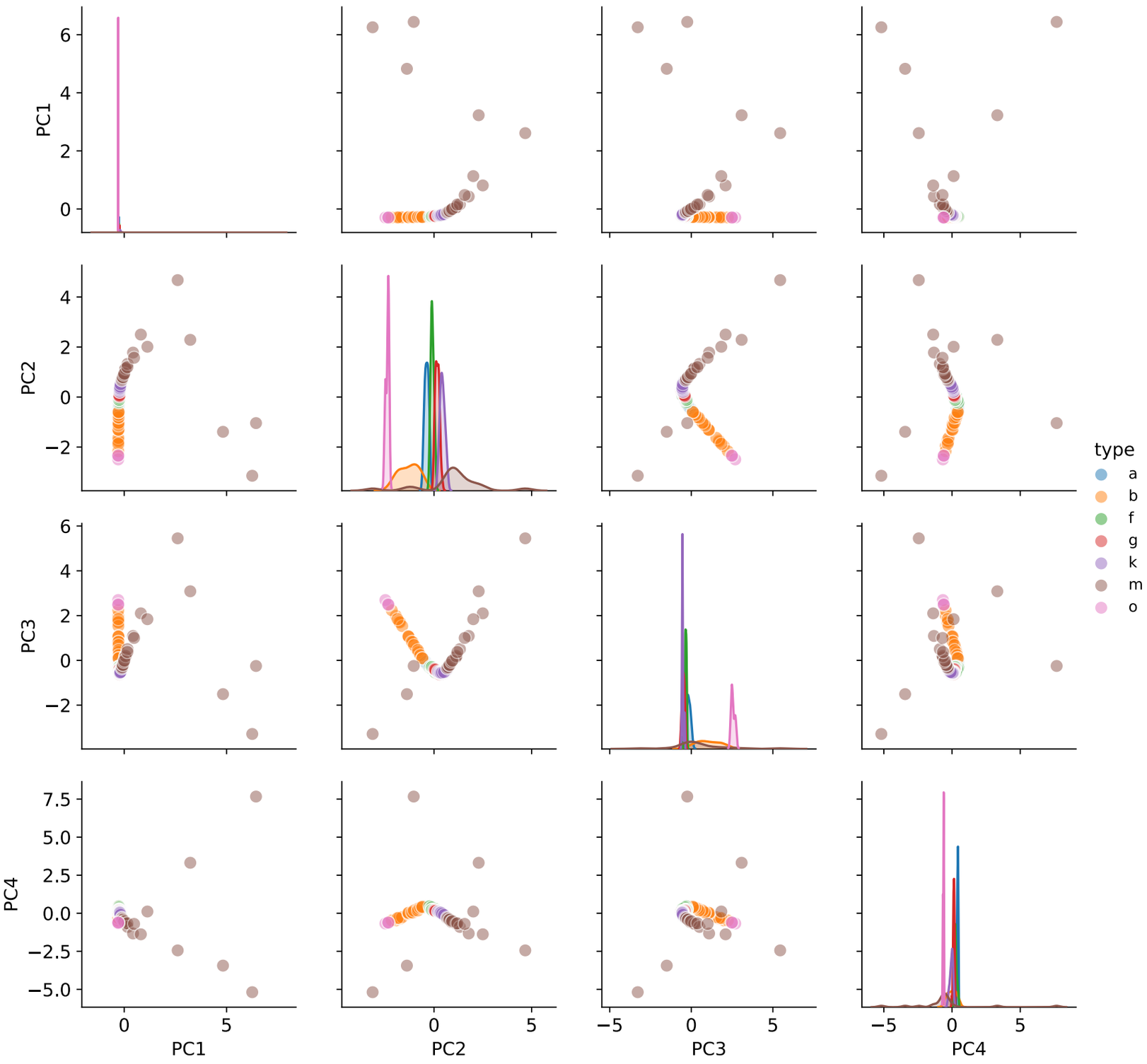
```
from sklearn.decomposition import PCA
pca = PCA(n_components=n_components, whiten=True)
pca.fit(X)
```

PCA of stellar spectra



The mean spectrum

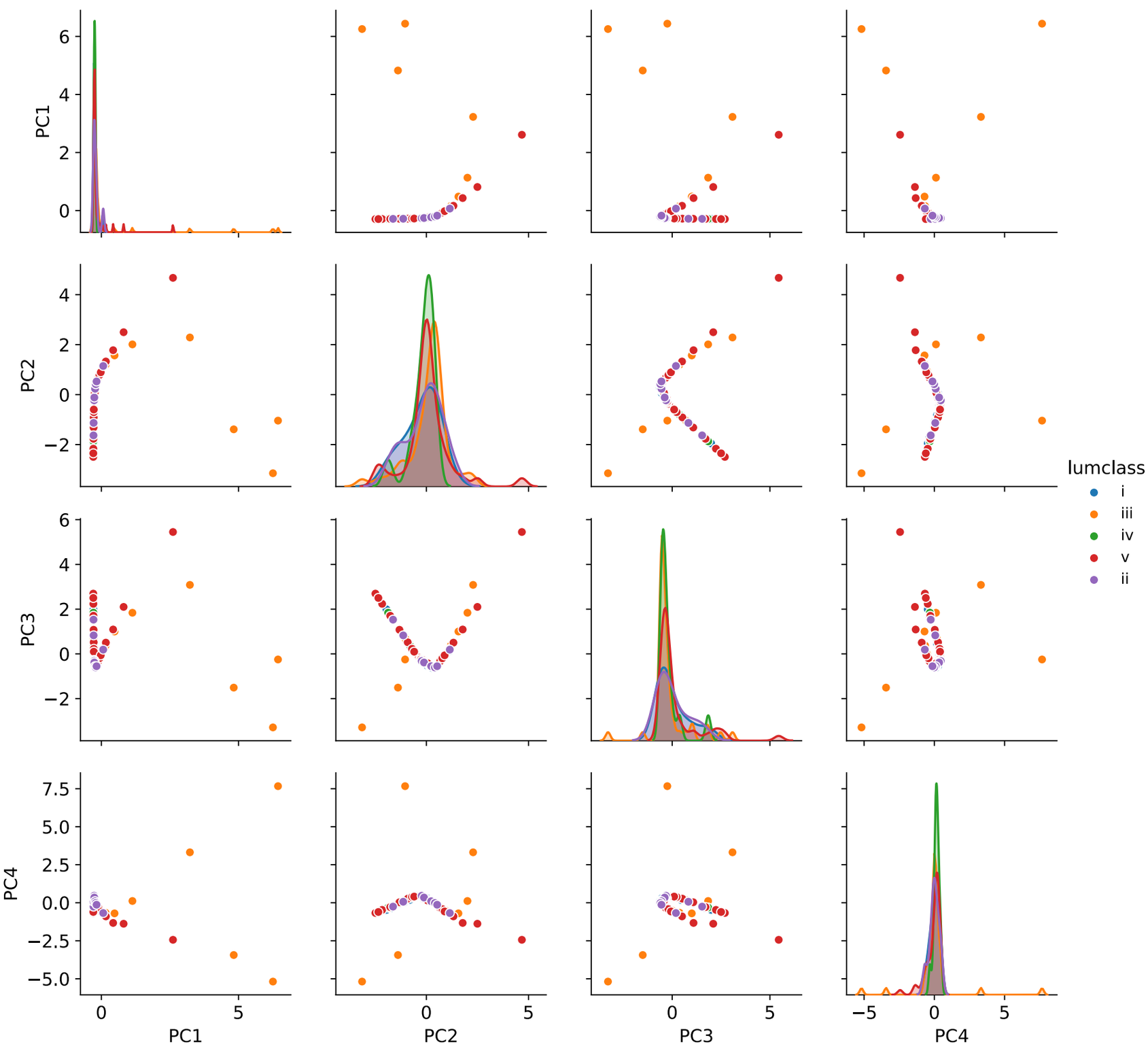
PCA of stellar spectra - the content of the PCs



Stellar types

seems to work fairly well

PCA of stellar spectra - the content of the PCs

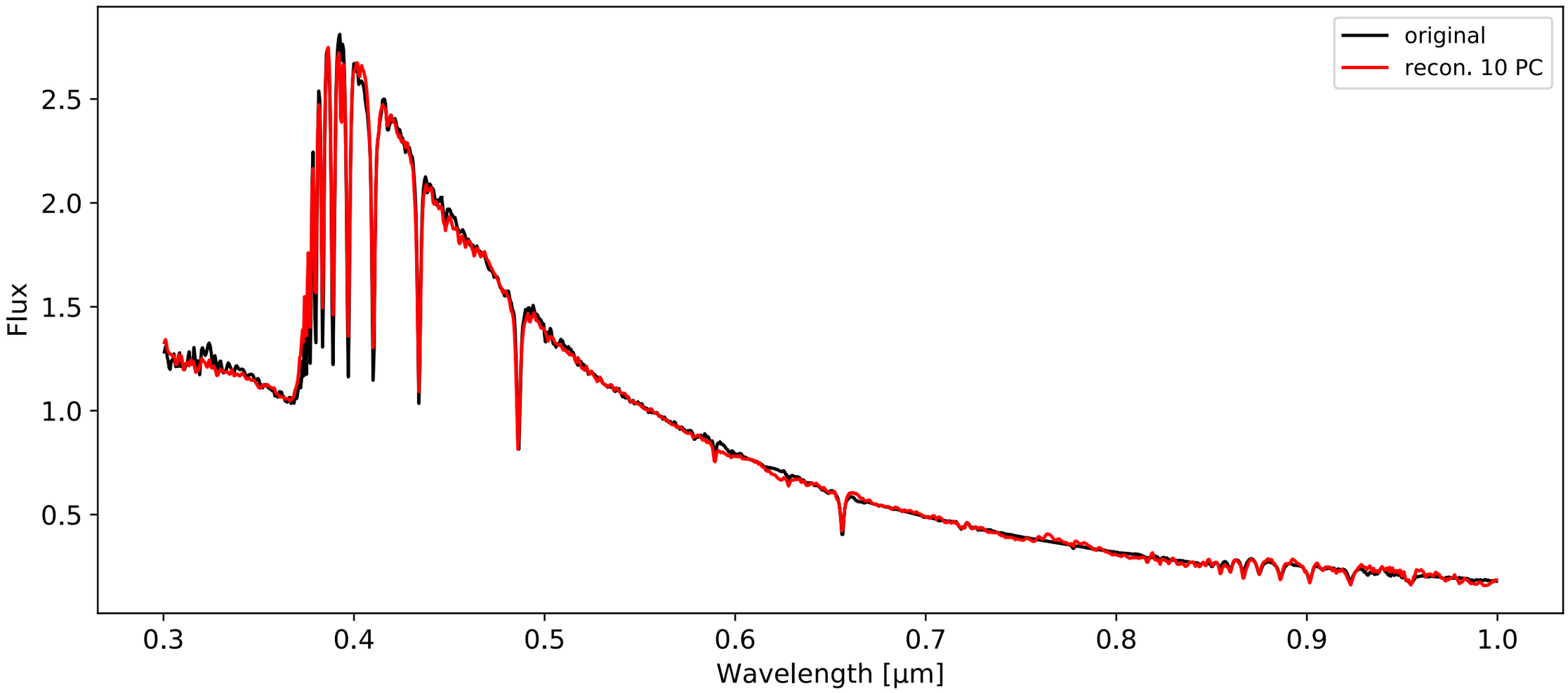


Luminosity class

Does not work at all

PCA of stellar spectra - reconstruction

```
pcs = pca.transform(X)
sp_depr = pca.inverse_transform(pcs)
```



Physical decomposition - positivity

One clear disadvantage of PCA when trying to interpret its results is that the eigencomponents can be negative (bad for spectra!)

An alternative approach is Non-negative Matrix Factorisation (NMF) which can enforce non-negative components.

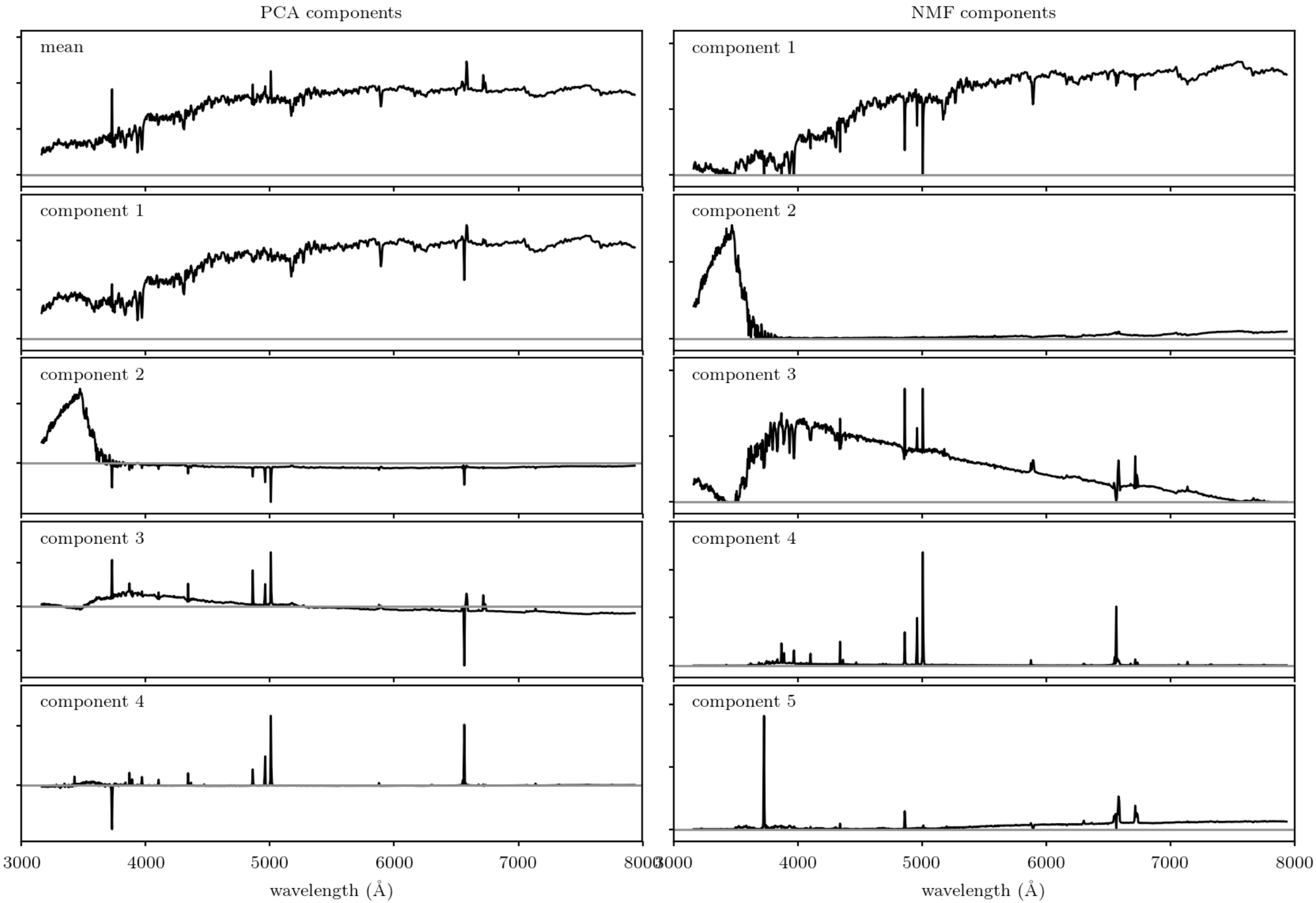
This is used in astronomy - particularly in fitting galaxy spectra (Blanton & Roweis 2007), but has also seen use in exo-planet direct imaging (Gomez Gonzales et al 2017), sky subtraction of spectra (Zhang et al 2016), X-ray bursts (Degenaar et al 2016), and PAH spectra (Rosenberg et al 2011) to mention some.

One down-side is that you need to decide the number of components first and the data must be >0 (not surprisingly), so noisy data is not suitable.

Physical decomposition - positivity

One
is tha

An al
(NMF



NMF in Python

The use of NMF is nearly identical to that of PCA:

```
from sklearn.decomposition import NMF

nmf = NMF(n_components)
nmf.fit(spectra)
nmf_comp = nmf.components_
```

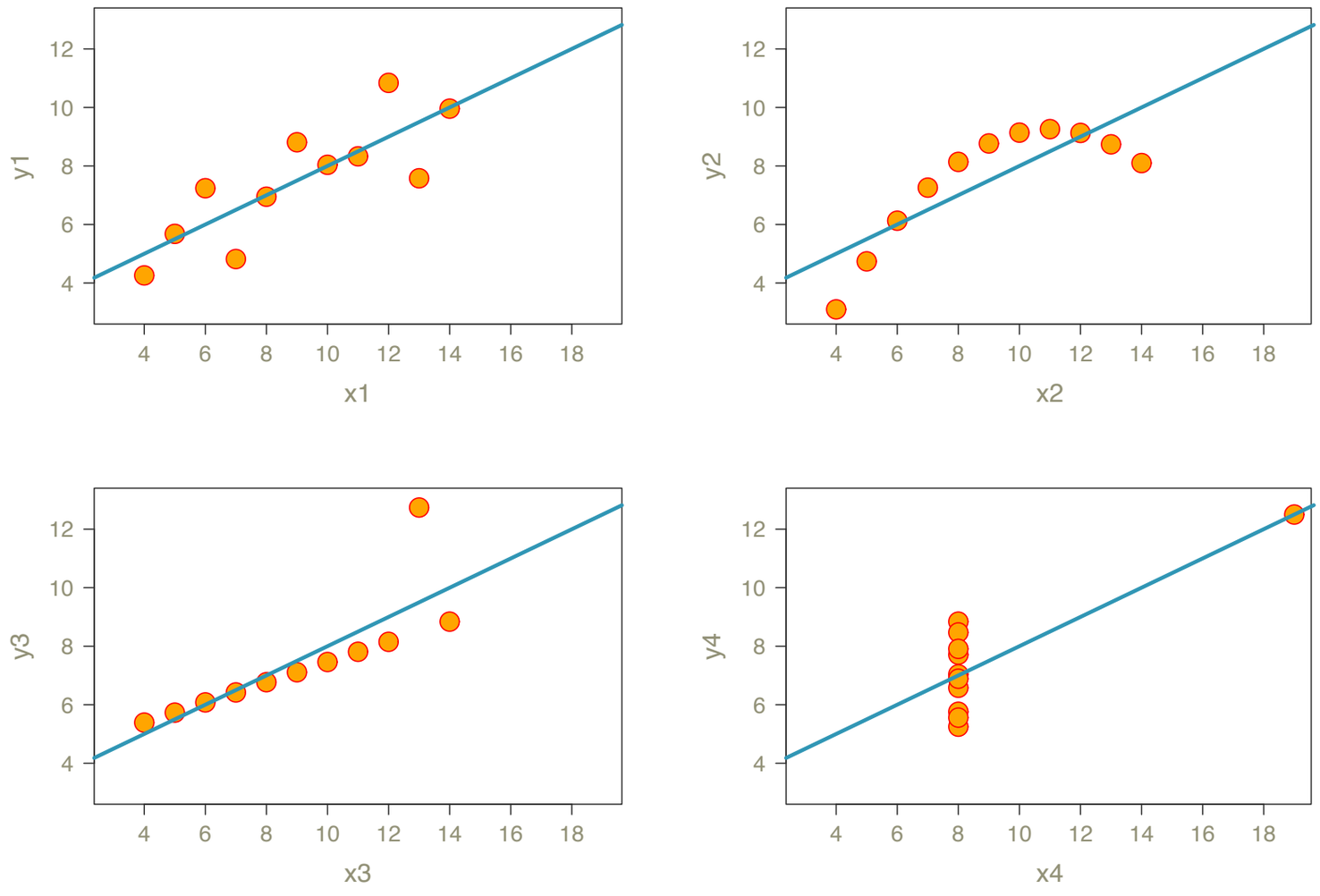
Manifold learning

Linear vs non-linear models

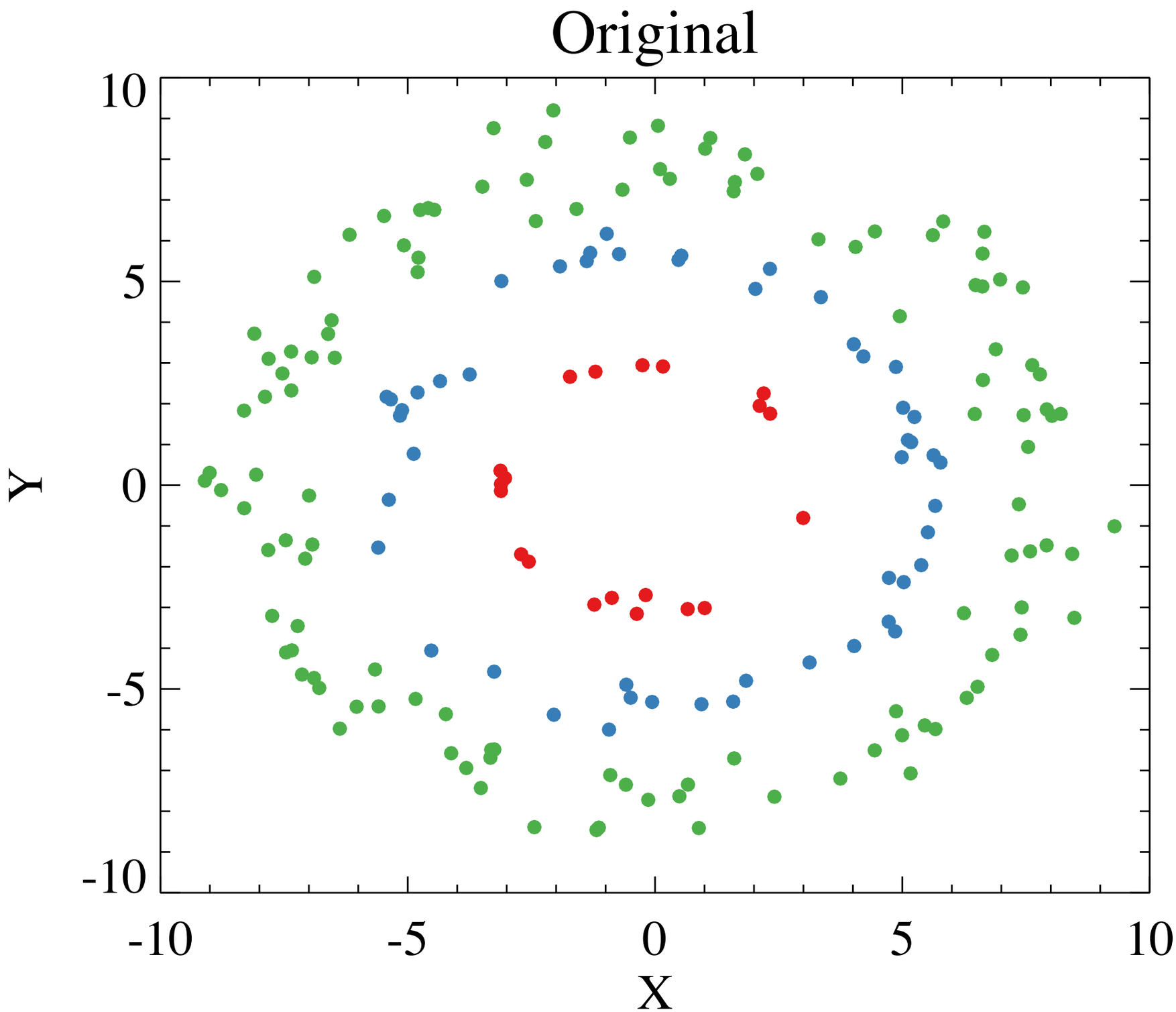
PCA is a linear method - it can find combinations of data vectors that make the resulting principal components linearly independent. Does that mean that PCs have to be independent?

No! (despite what people often write in their papers)

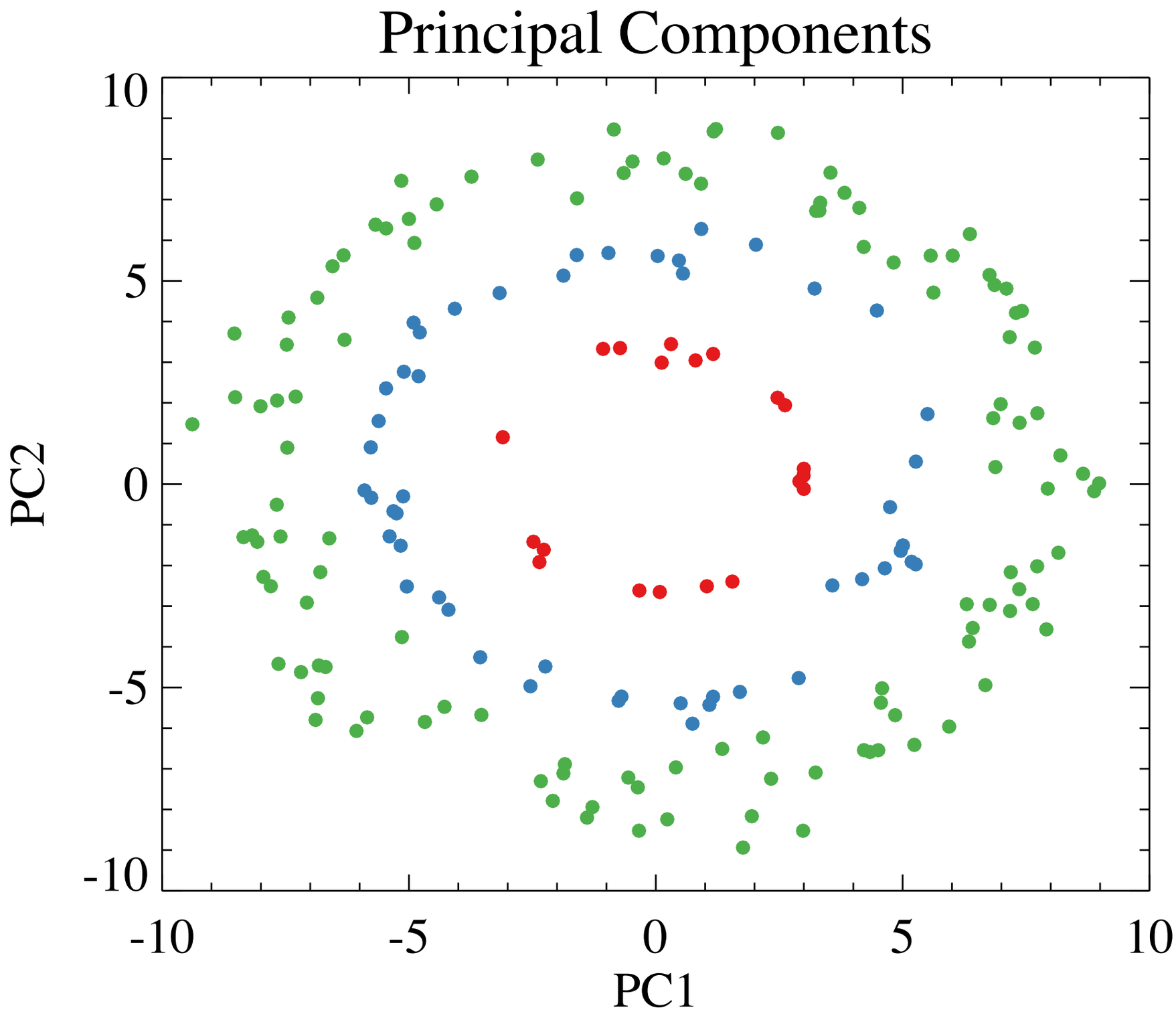
The **correlation coefficient** measures how closely a dataset can be fit by a **straight line** and the *correlation matrix* generalises this to higher dimensions.



Linear vs non-linear



Linear vs non-linear

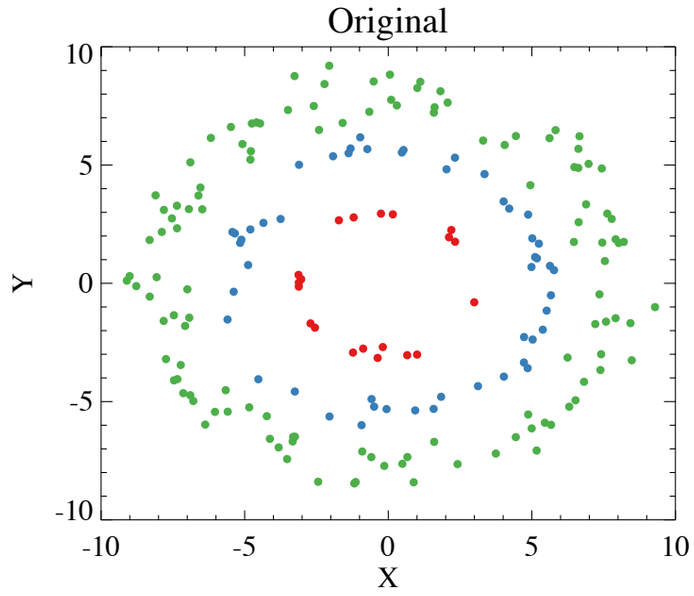


Linear vs non-linear

In this case we know what we should do: We should convert to polar coordinates:

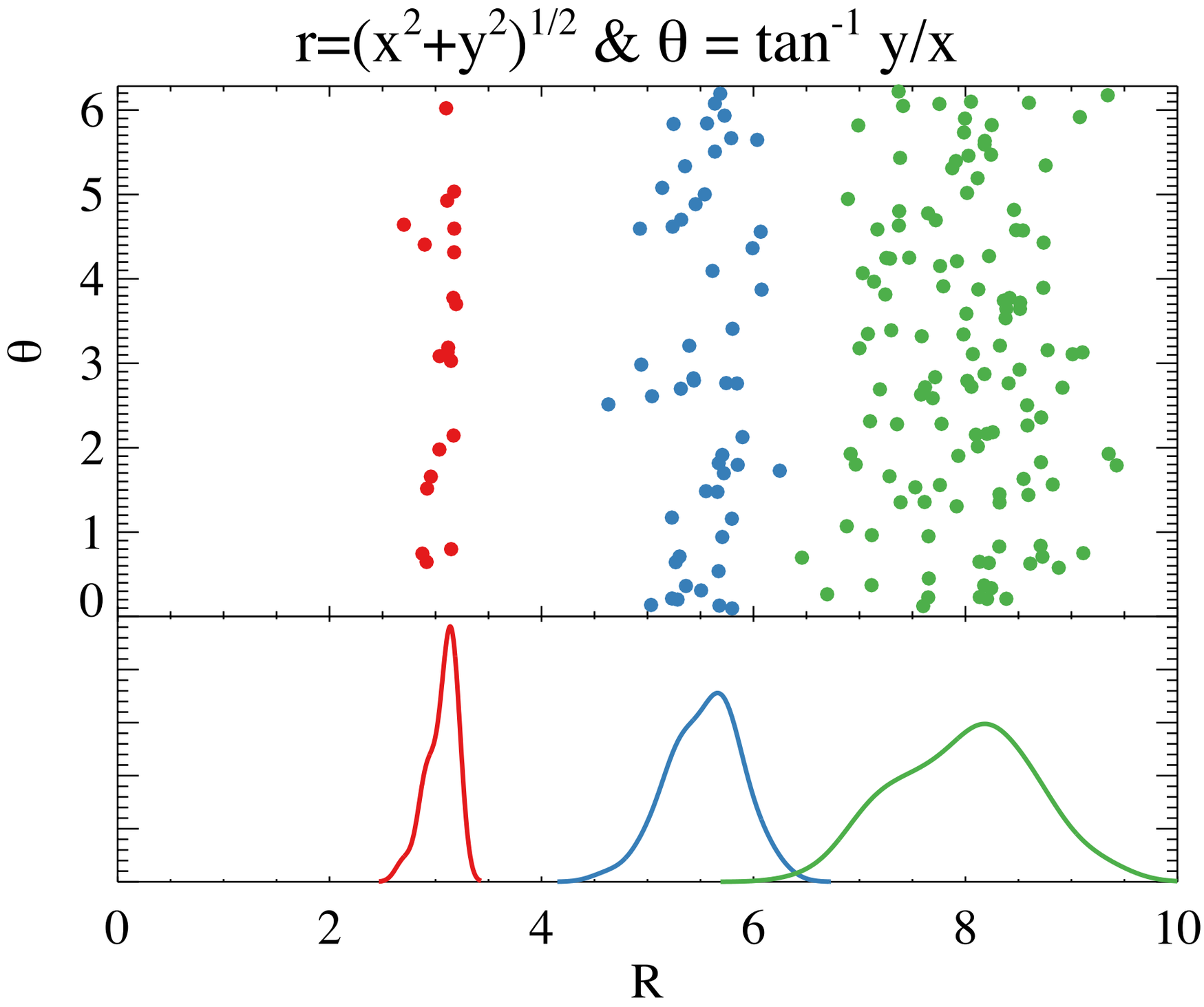
$$r = \sqrt{x^2 + y^2}$$

$$\theta = \tan^{-1} \frac{y}{x}$$

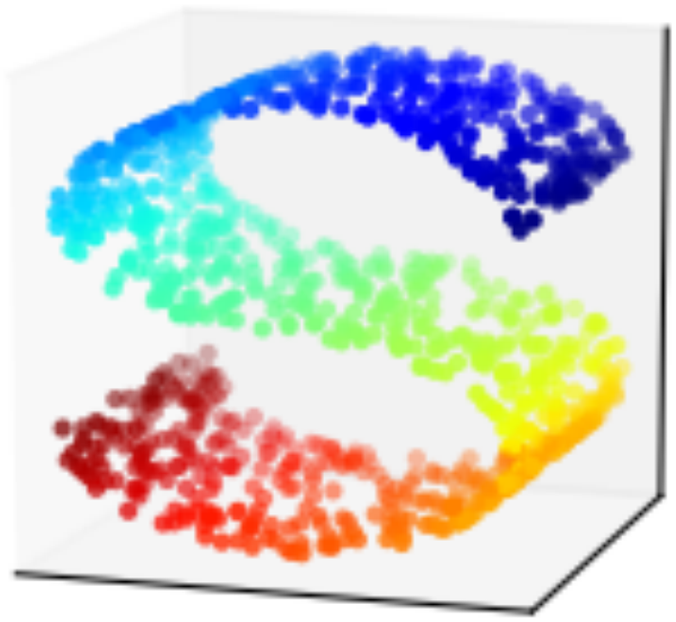


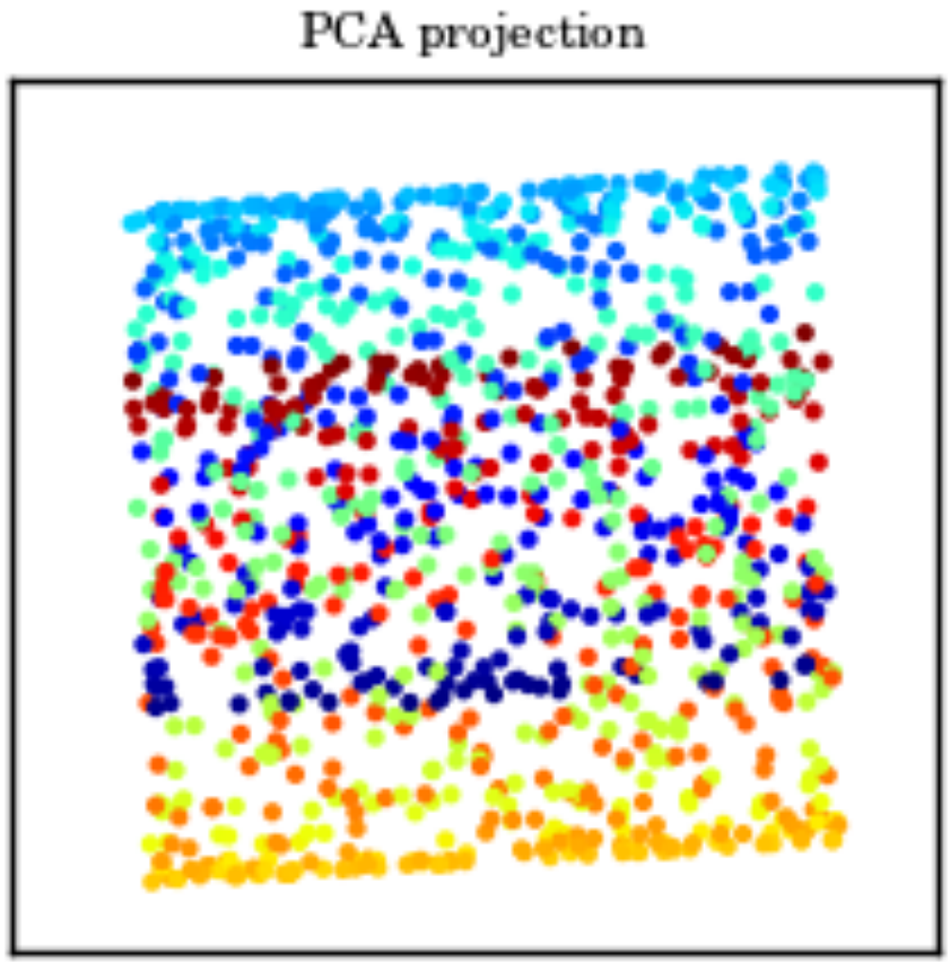
But this is a **non-linear** coordinate transformation so cannot be found by doing **PCA** which only can provide you with **linear transformations** of the input coordinates.

Linear vs non-linear



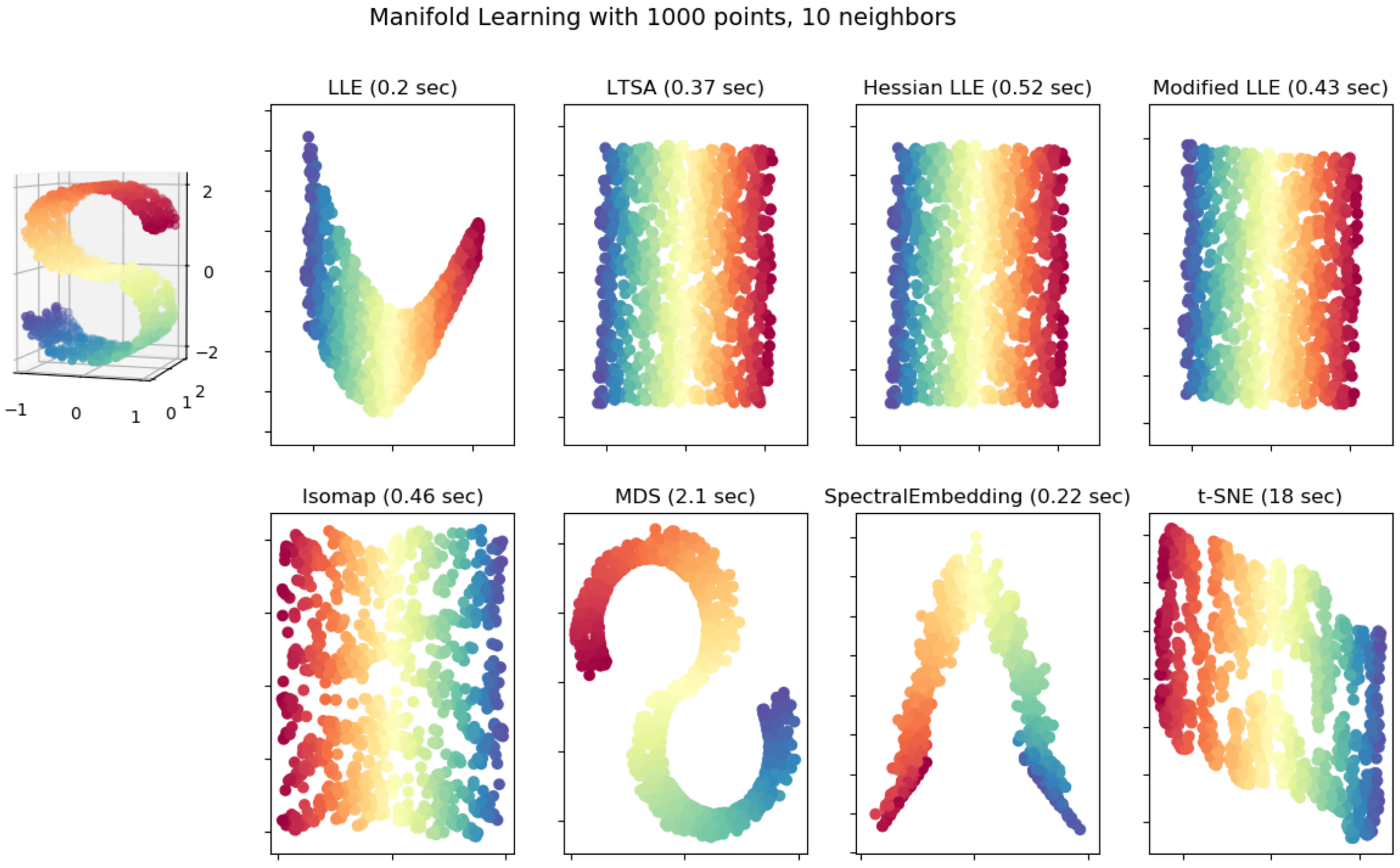
Linear versus non-linear





Enter manifold learning (or nonlinear dimensionality reduction)…

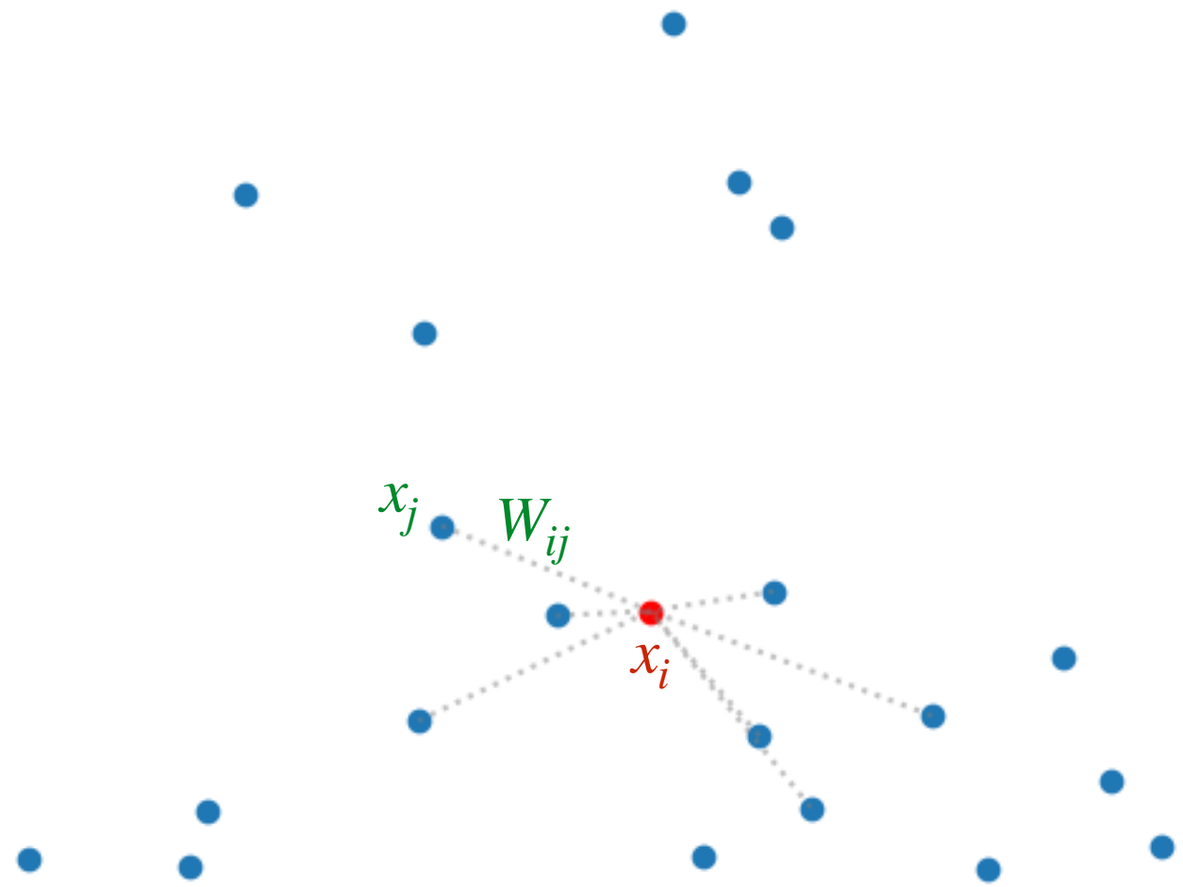
There is a zoo of these - here are a few:



See http://scikit-learn.org/stable/auto_examples/manifold/plot_compare_methods.html

General features

As a rule, the various manifold learning methods use local structure to find good projections.



Using global information to predict

General features

As a rule, the various manifold learning methods use local structure to find good projections.

Example: Locally Linear Embedding (LLE)

Here you try to fit small planes to the neighbourhood of each point and then match these together to project to lower dimensions.

So basically we want to minimise the reconstruction error:

$$\mathrm{Err} = |X - W X|^2$$

Locally Linear Embedding

So basically we want to minimise the reconstruction error of X based on its neighbours:

$$\mathrm{Err} = |X - W X|^2$$

Writing this out we have

$$\mathrm{Err}_x = \sum_{i=1}^{N} \left| x_i - \sum_{j=1}^{N} W_{ij} x_j \right|^2$$

In LLE the trick is to set $W_{ii}=0$ and W_{ij} to zero for all but the k nearest neighbours. Then we minimise the error function to find W_{ij}

Locally Linear Embedding

That sorts things out in the high dimensional space, but now we want to project onto a lower dimensional space. To do this we minimise:

$$\mathrm{Err}_y = \sum_{i=1}^{N} \left| y_i - \sum_{j=1}^{N} W_{ij} y_j \right|^2$$

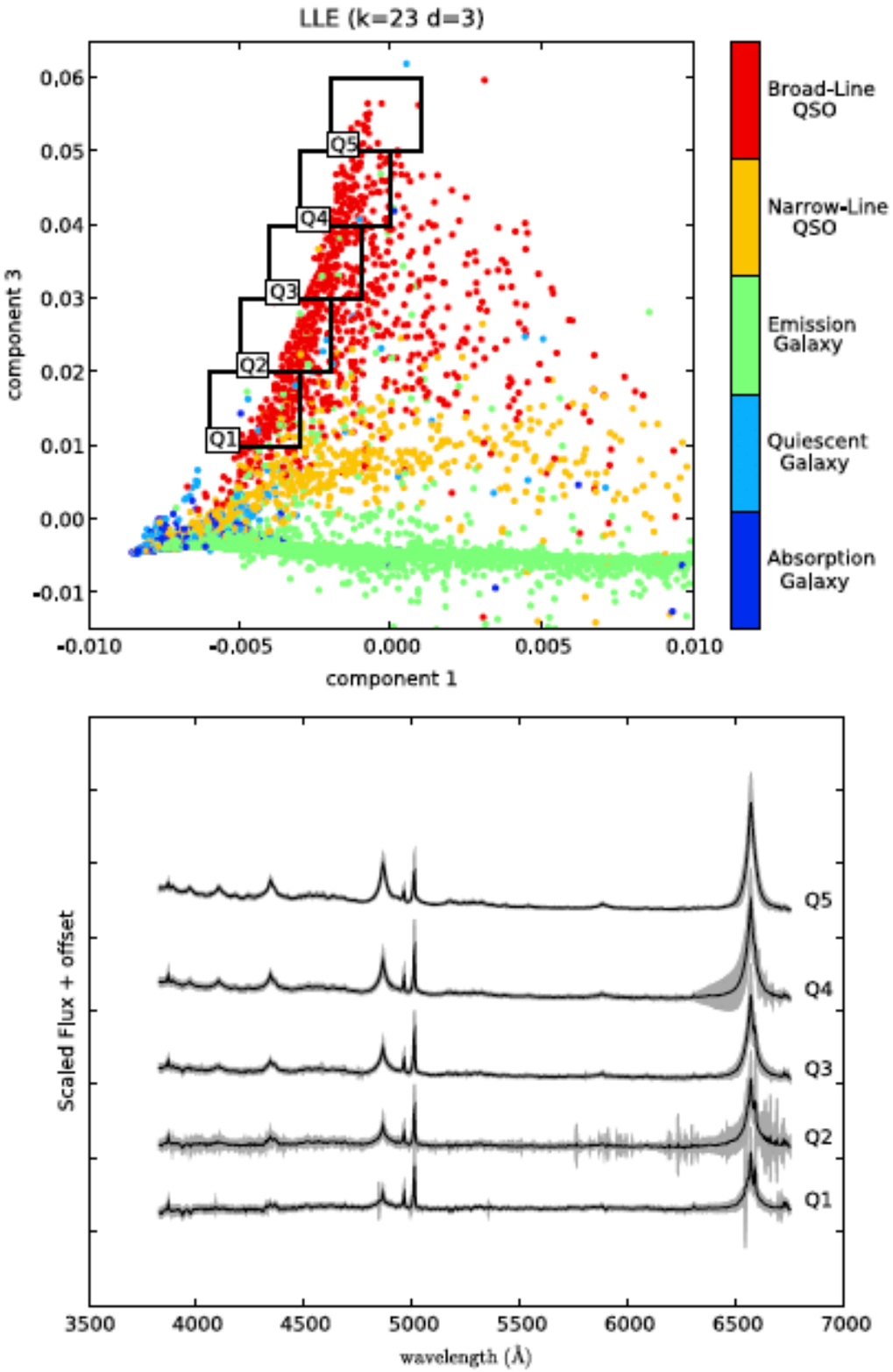
Here W_{ij} is fixed and the positions, y_i, are modified. This is an eigenvector problem and you get some eigenvectors as results.

```
from sklearn.manifold import LocallyLinearEmbedding
lle = LocallyLinearEmbedding(k, n)
lle.fit(X)
proj = lle.transform(X)
```

Locally Linear Embedding in astronomy

Vanderplas & Connolly (2009):

LLEs can be used to classify galaxy spectra from the Sloan Digital Sky Survey. The data is here entire spectra and you look at some components of this.



t-Stochastic Neighbour Embedding (t-SNE)

This is an example of a method that is really aimed to project to 2-3 dimensions.

1. Measure the similarities (“closeness”) of samples in the original high-D space

$$p_{j|i} = \frac{\exp\left(-\left|x_i - x_j\right|^2 / 2\sigma_i^2\right)}{\sum_{k \neq i} \exp\left(-\left|x_i - x_k\right|^2 / 2\sigma_i^2\right)}$$

2. Create an error function based on the separation in the low-dimensional space and optimise this:

$$q_{ij} = \frac{f\left(\left|x_i - x_j\right|\right)}{\sum_{k \neq i} f\left(\left|x_i - x_k\right|\right)}$$

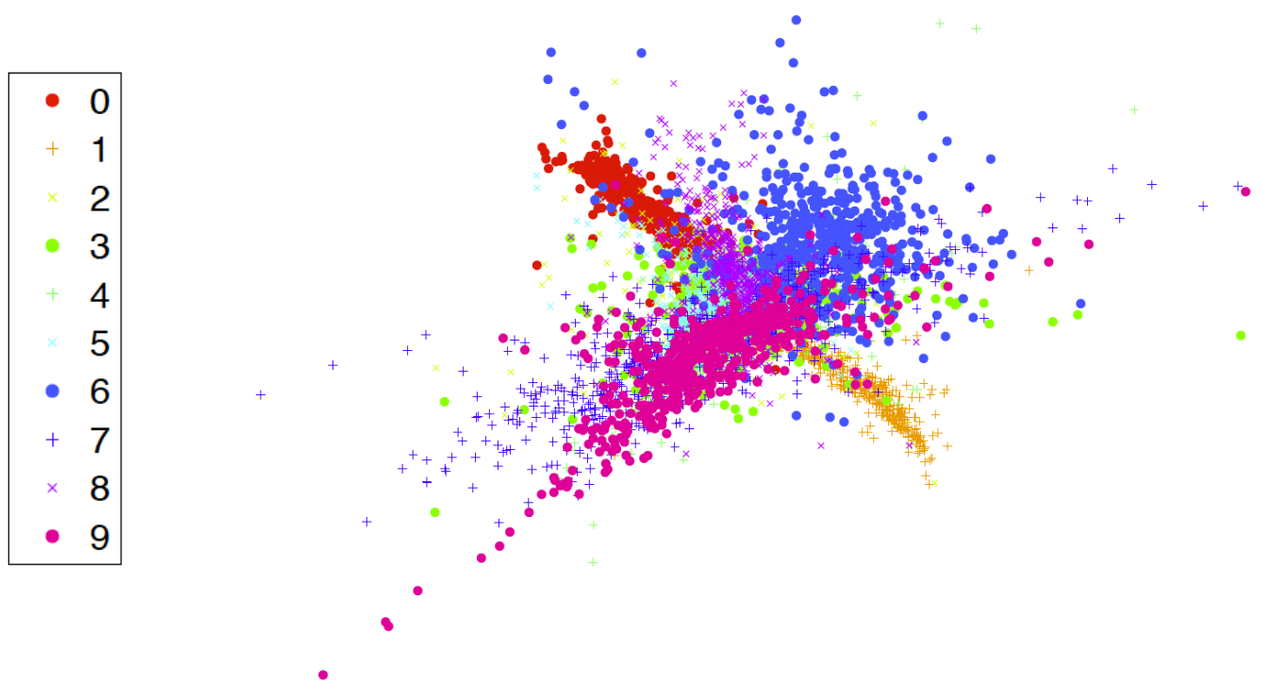
$$f(x) = \frac{1}{1 + x^2}$$

A Cauchy distribution

t-Stochastic Neighbour Embedding (t-SNE)

The MNIST hand-written digits dataset is a classic dataset for testing.

LLE:

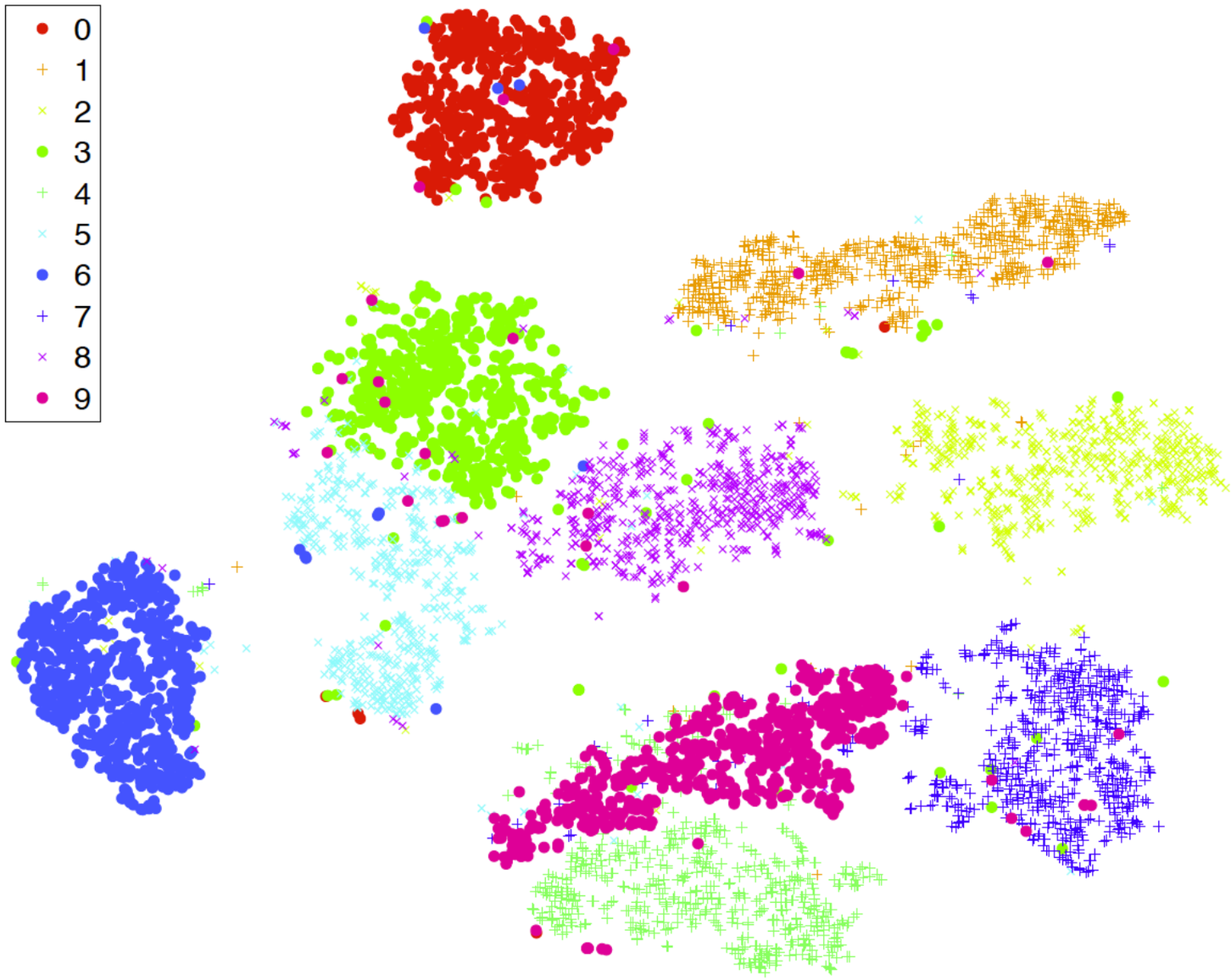


From van Maaten & Hinton (2008), Journal of Machine Learning Research 9 (2008) 2579-2605

t-Stochastic Neighbour Embedding (t-SNE)

The MNIST hand-written digits dataset is a classic dataset for testing.

t-SNE:



From van Maaten & Hinton (2008), Journal of Machine Learning Research 9 (2008) 2579-2605

t-Stochastic Neighbour Embedding (t-SNE)

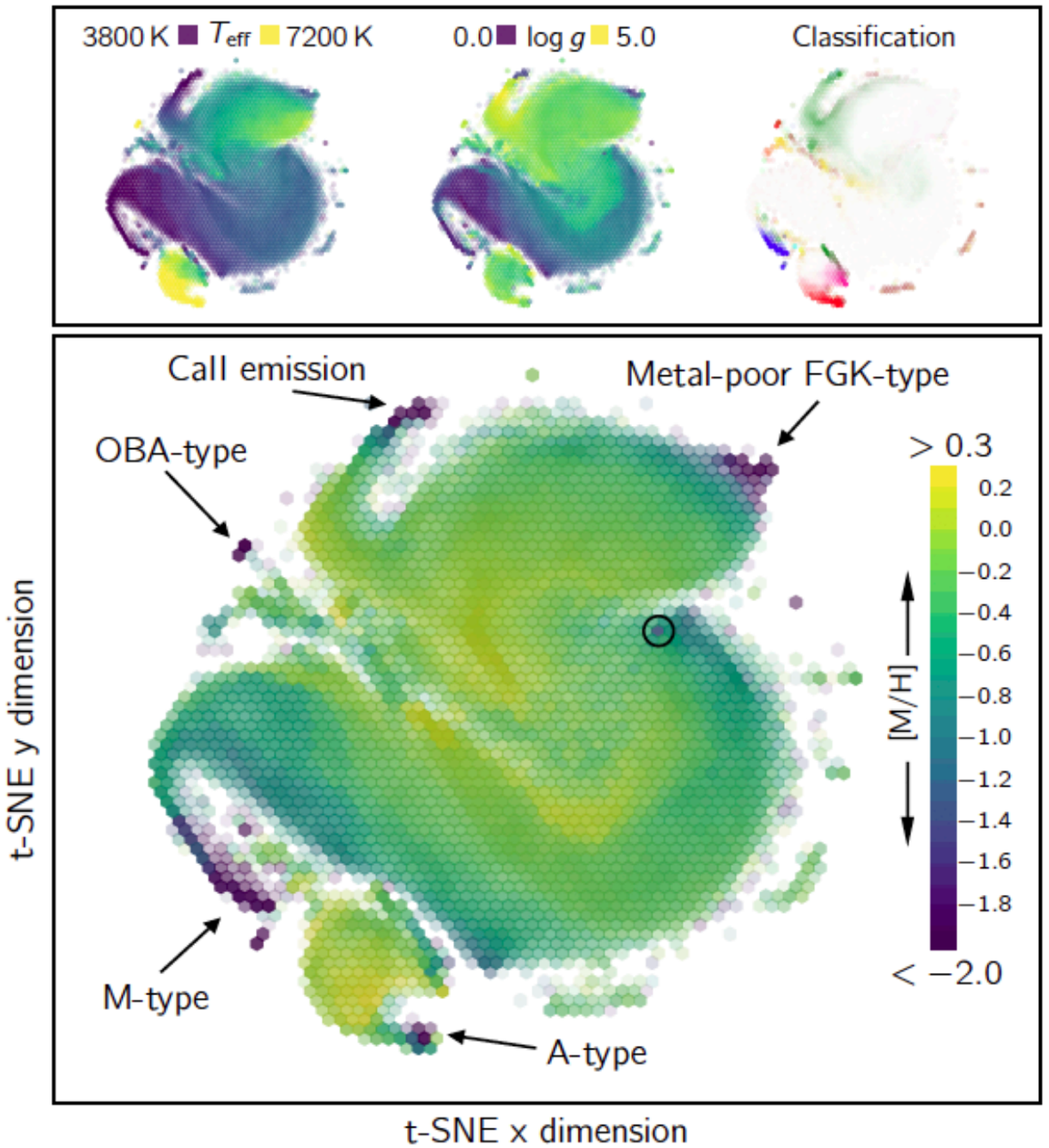
9

see: https://www.oreilly.com/learning/an-illustrated-introduction-to-the-t-sne-algorithm

t-SNE and others and astronomy

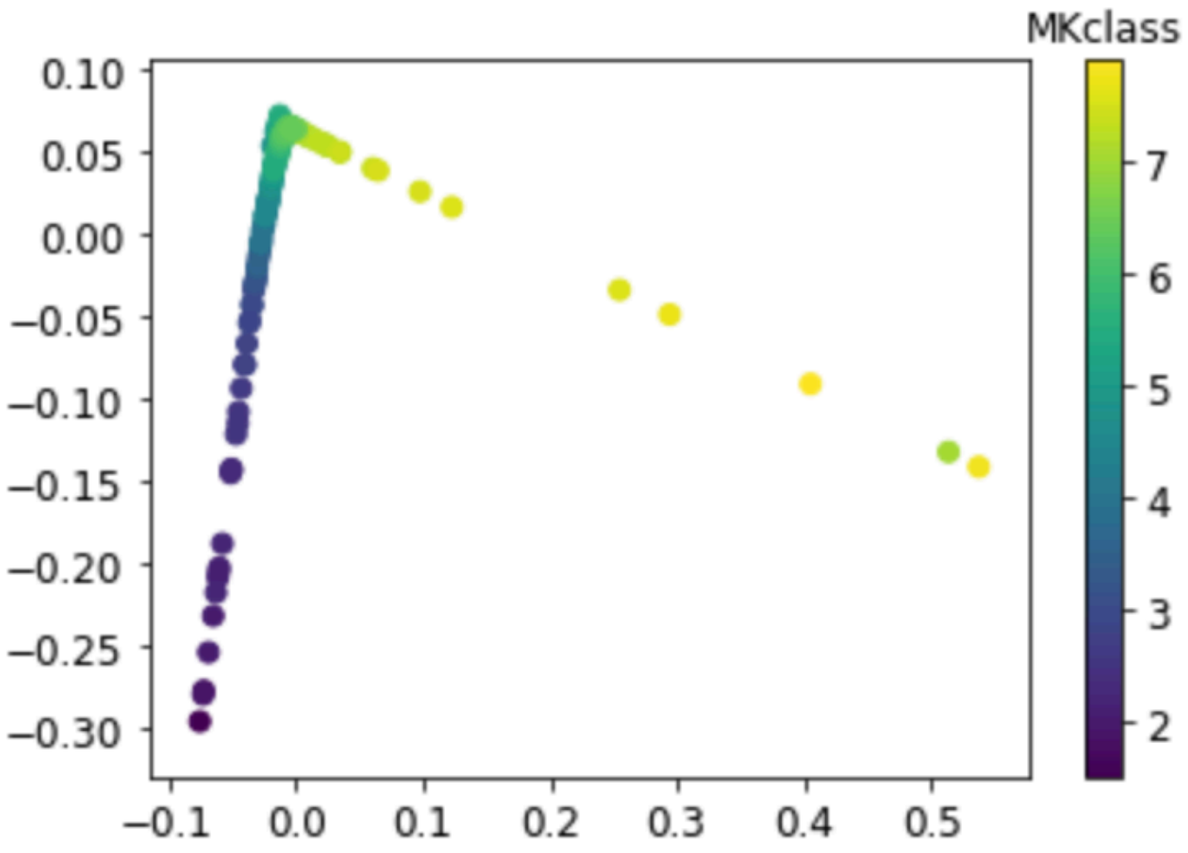
t-SNE is fairly slow but with an update using the Barnes-Hut N-body algorithm it is now useable for large-ish datasets.

Matijevic et al (2017):
Use t-SNE to analyse spectra of stars from the RAVE survey. This is used as a way to identify candidate metal-poor stars.



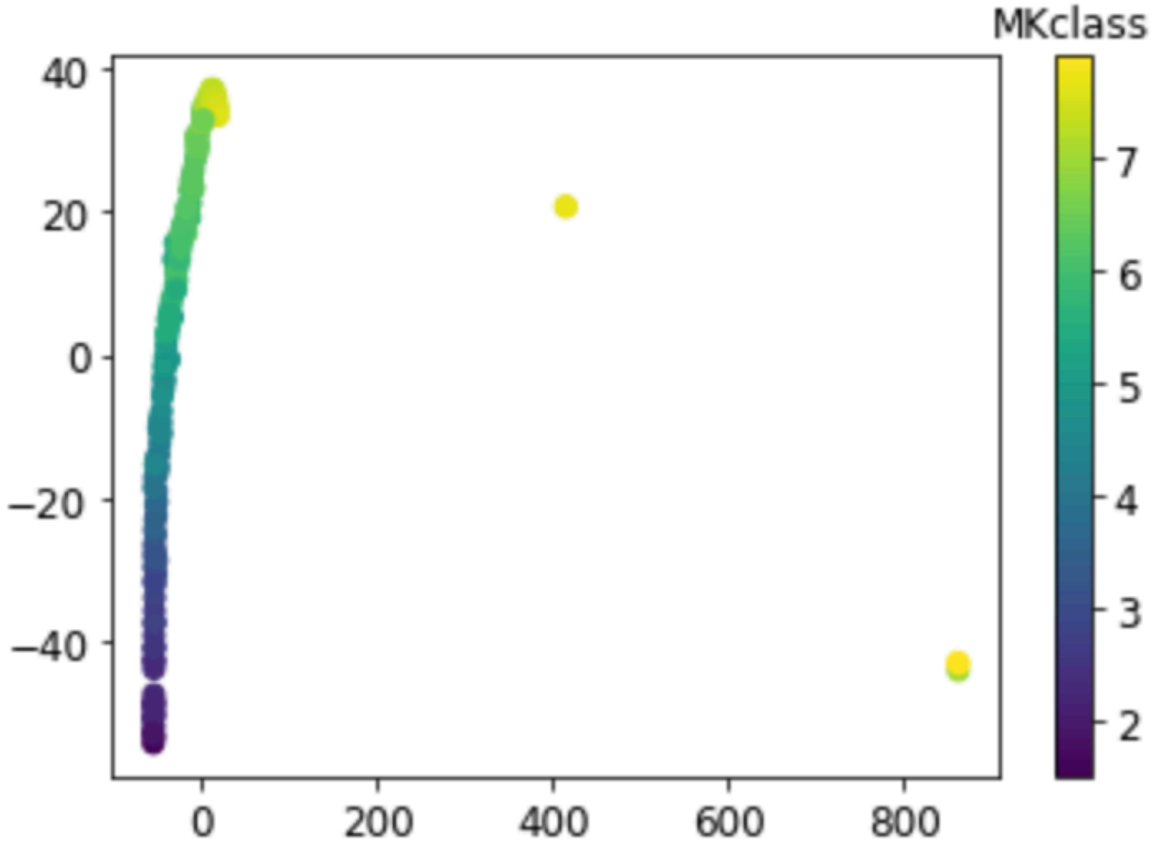
t-SNE & LLE on the stellar spectra

LLE




```
lle = LocallyLinearEmbedding(10,
n_components=2)
proj = lle.fit_transform(X)
```

t-SNE




```
sne = TSNE(n_components=2, init='pca',
random_state=0, perplexity=10)
proj = sne.fit_transform(X)
```

The pros and cons of manifold learners

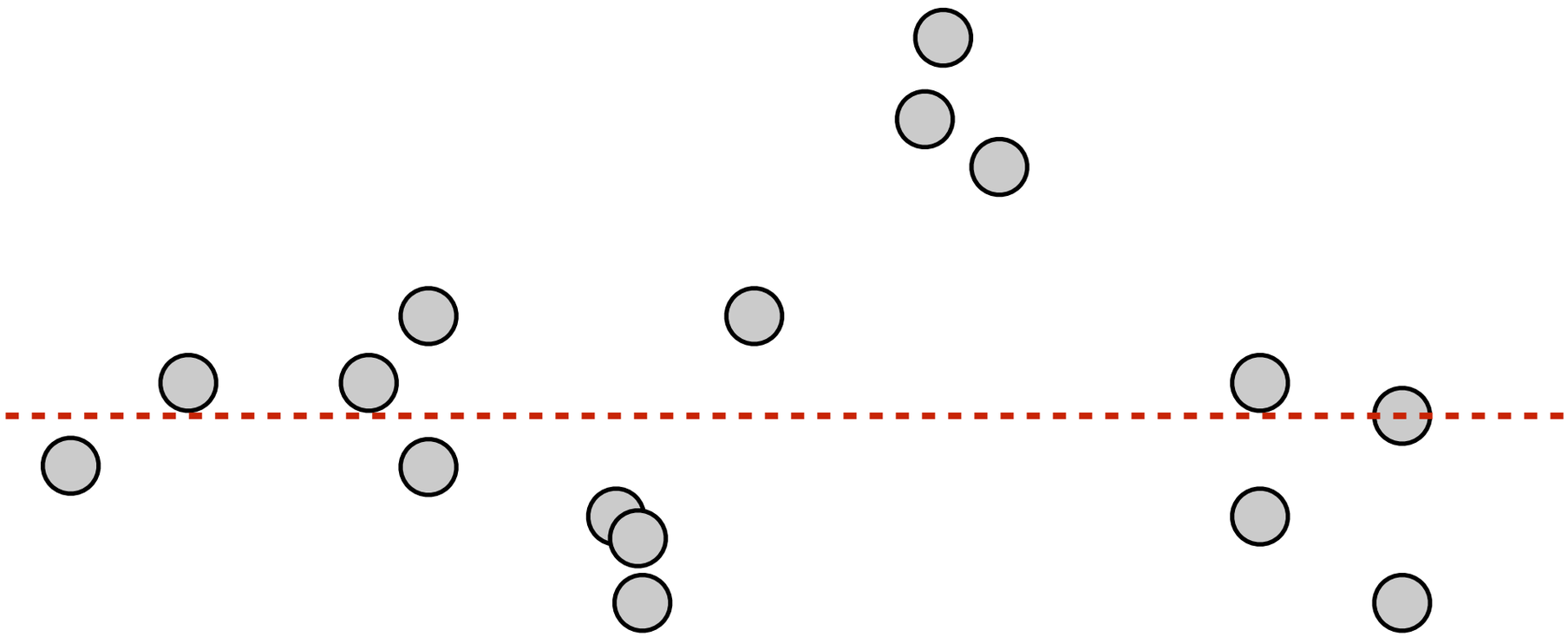
- Non-linear dimension reduction can extract structure out of data with non-linearities.
- Can lead to substantial dimensional reduction.
- They deal badly with noisy/gappy data and are (usually) sensitive to outliers.
- All have tuning parameters that you need to decide on.
- To project to lower dimensions you need the full data.
- Memory/CPU requirements can be substantial.

Bottom line: Try PCA first, or possibly NMF.

Regression - keeping track of local properties

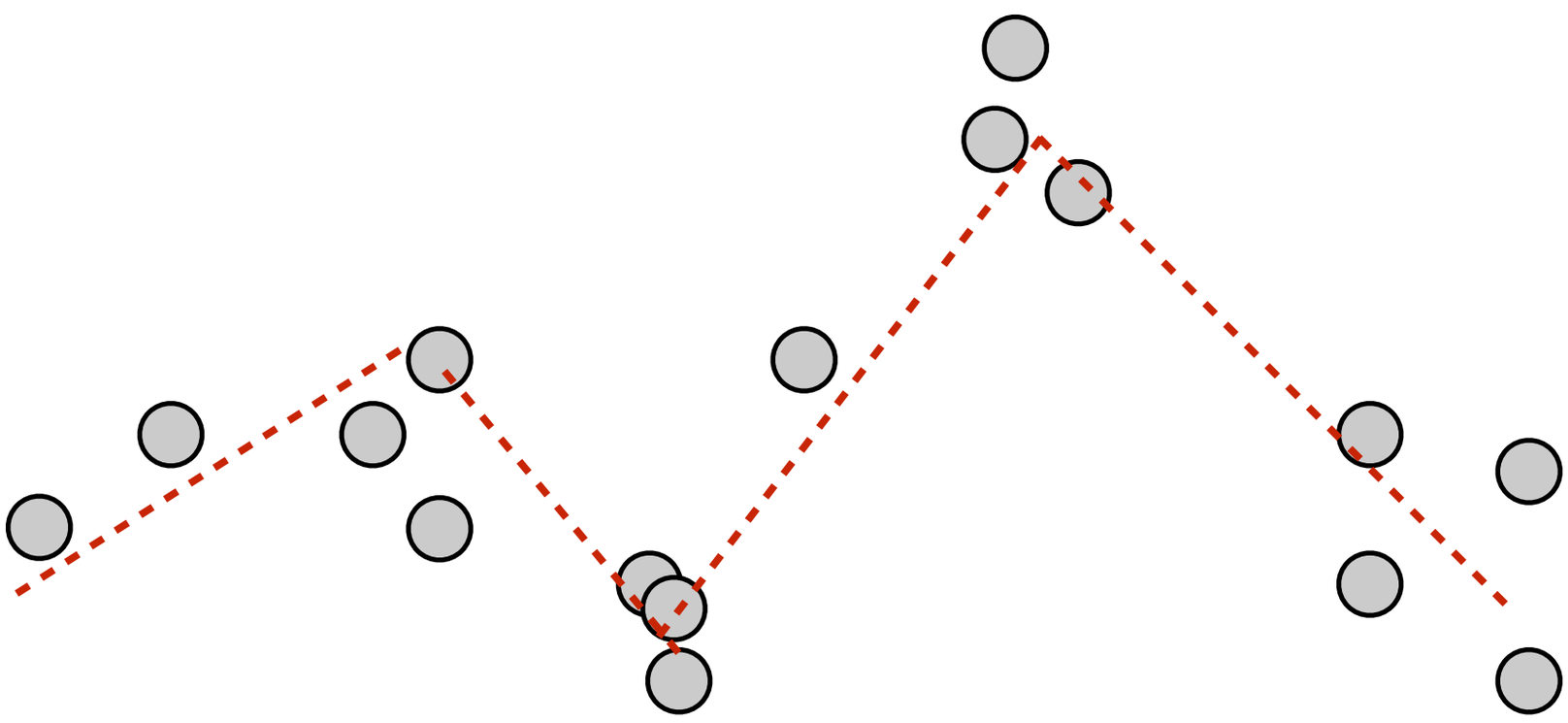
The broad idea

Faced with data where we do not have a clear idea how they are related a simple regression might be unsuitable:



The broad idea

Keeping track of local trends might be a better idea:

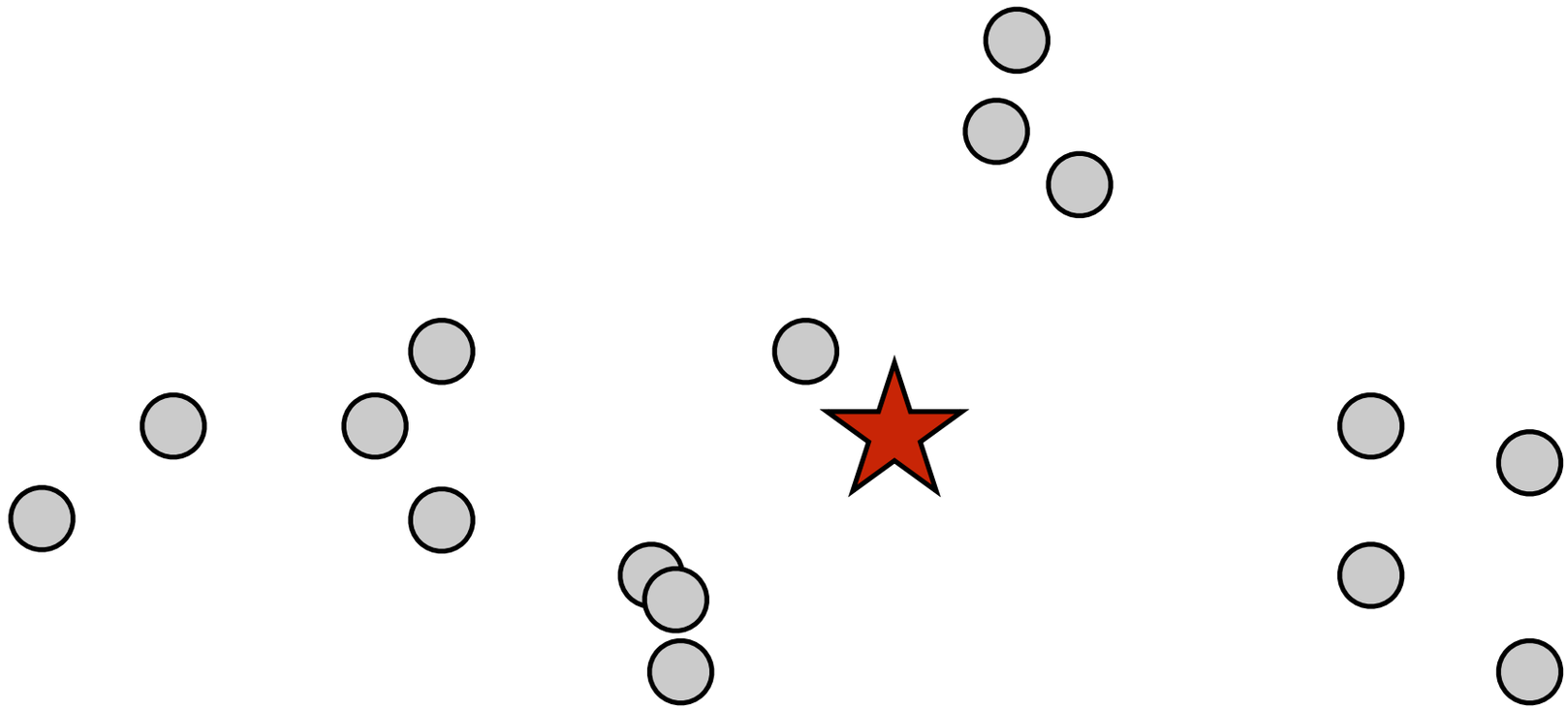


The main techniques

- Nearest neighbour regression
  - Take the mean of the k nearest points
- Kernel regression
  - Calculate the weighted mean of training points
- Locally linear regression
  - Calculate a weighted linear regression at each point
- Gaussian process regression
  - Drop fixed functions and try to fit in the space of “all” functions

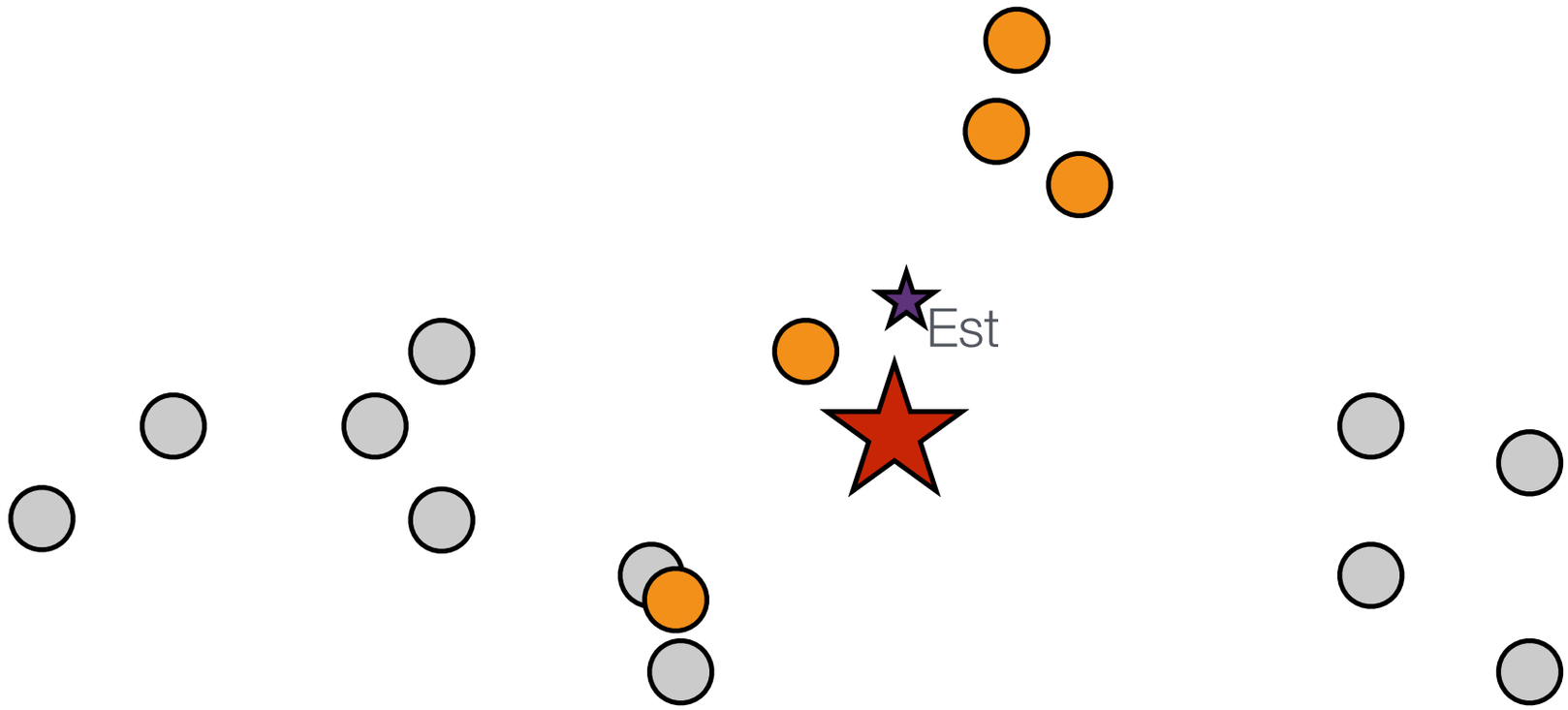
Nearest neighbour regression

$$\hat{y}(x) = \frac{1}{k} \sum_{x_j \in \mathrm{Neighbours(x;\ k)}} y_j$$



Nearest neighbour regression

$$\hat{y}(x) = \frac{1}{k} \sum_{x_j \in \mathrm{Neighbours(x;\ k)}} y_j$$

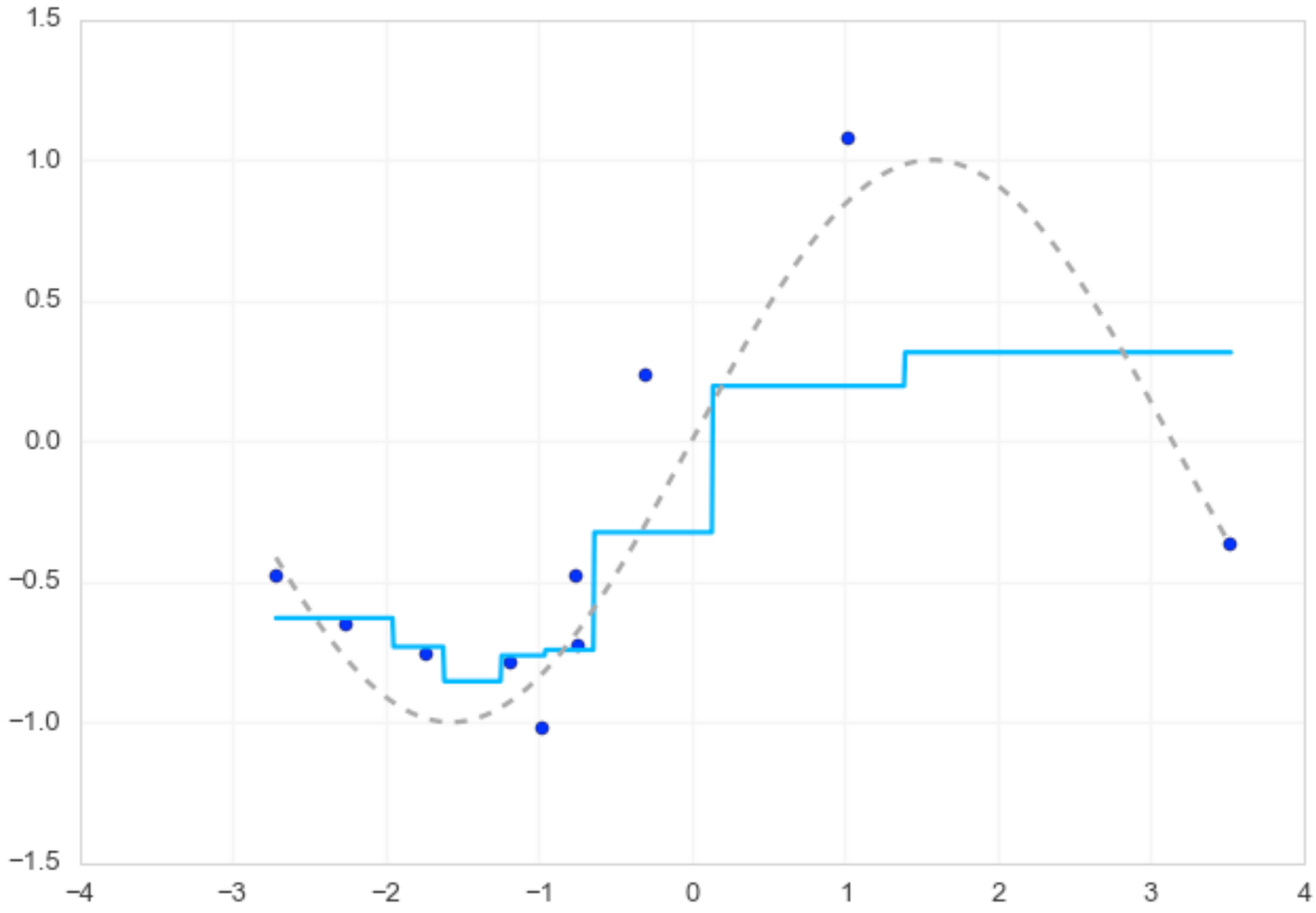


Nearest neighbour regression

$$\hat{y}(x) = \frac{1}{k} \sum_{x_j \in \mathrm{Neighbours(x;\ k)}} y_j$$

```
from sklearn import neighbors

k = 3
knn = neighbors.KNeighborsRegressor(k)
y_est = knn.fit(X, y).predict(Xplot)
```

Nearest neighbour regression




```
from sklearn import neighbors

k = 3
knn = neighbors.KNeighborsRegressor(k)
y_est = knn.fit(X, y).predict(Xplot)
```

Kernel regression

In knn regression we give equal weight to each point. If instead we give a variable weight we get kernel regression

$$\hat{y}(x) = \sum_{i=1}^{N} K_h(x, x_i)\, y_i$$

It is actually not necessary that the x_i are at the same place as y_i, but I will assume that they are. (if they are not you have to be careful with the normalisation of the basis functions)

h is a complexity parameter so needs to be determined by AIC/BIC or cross-validation for instance.

Kernel regression

The most common formulation of kernel regression renormalises the kernel functions to give the Nadaraya-Watson method:

$$\hat{y}(x) = \frac{\sum_{i=1}^{N} K_h(x, x_i)\, y_i}{\sum_{i=1}^{N} K_h(x, x_i)}$$

This comes from

$$\hat{y}(x) = E[Y|X = x] = \int y p(y|x) dy = \int y \frac{p(x,y)}{p(x)} dx$$

and inserting kernel density estimates for $p(x,y)$ and $p(x)$

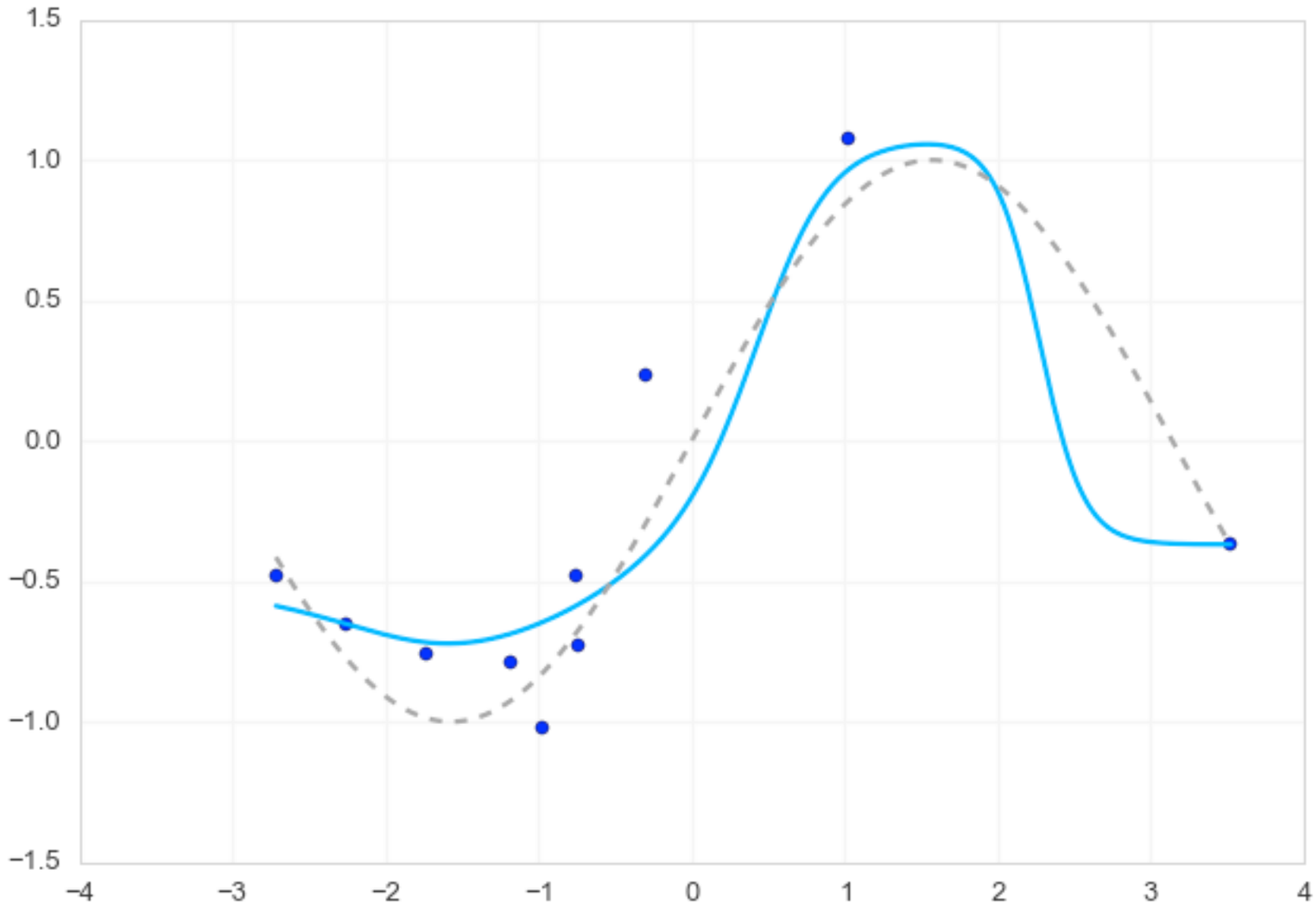
Kernel regression

The most common formulation of kernel regression renormalises the kernel functions to give the Nadaraya-Watson method:

$$\hat{y}(x) = \frac{\sum_{i=1}^{N} K_h(x, x_i)\, y_i}{\sum_{i=1}^{N} K_h(x, x_i)}$$

```
from astroML.linear_model import NadarayaWatson

model = NadarayaWatson('gaussian', np.asarray(0.6))
model.fit(X, y)
y_est = model.predict(Xplot)
```

Kernel regression




```
from astroML.linear_model import NadarayaWatson

model = NadarayaWatson('gaussian', np.asarray(0.6))
model.fit(X, y)
y_est = model.predict(Xplot)
```

Locally linear regression

In knn and kernel regression we effectively work with the zeroth level Taylor expansion - the constant term. The next step is to fit a weighted linear regression:

$$\theta_0(x), \theta_1(x) = \underset{\theta_0,\theta_1}{\operatorname{argmin}} \sum_{i=1}^{N} \left(y_i - \theta_0 - \theta_1(x - x_i)\right)^2 K_h(x, x_i)$$

This turns out to be very useful in many situations and is often used as a powerful smoother under the name loess/lowess and a powerful package `locfit` is available in R (see rpy2)

h is a complexity parameter so needs to be determined by AIC/BIC or cross-validation for instance.

Locally linear regression

The weight/kernel is usually take to be the tri-cubic function:

$$w_i = (1 - |t|^3)^3 \; I(|t| \leq 1)$$

with $t = (x - x_i)/h$

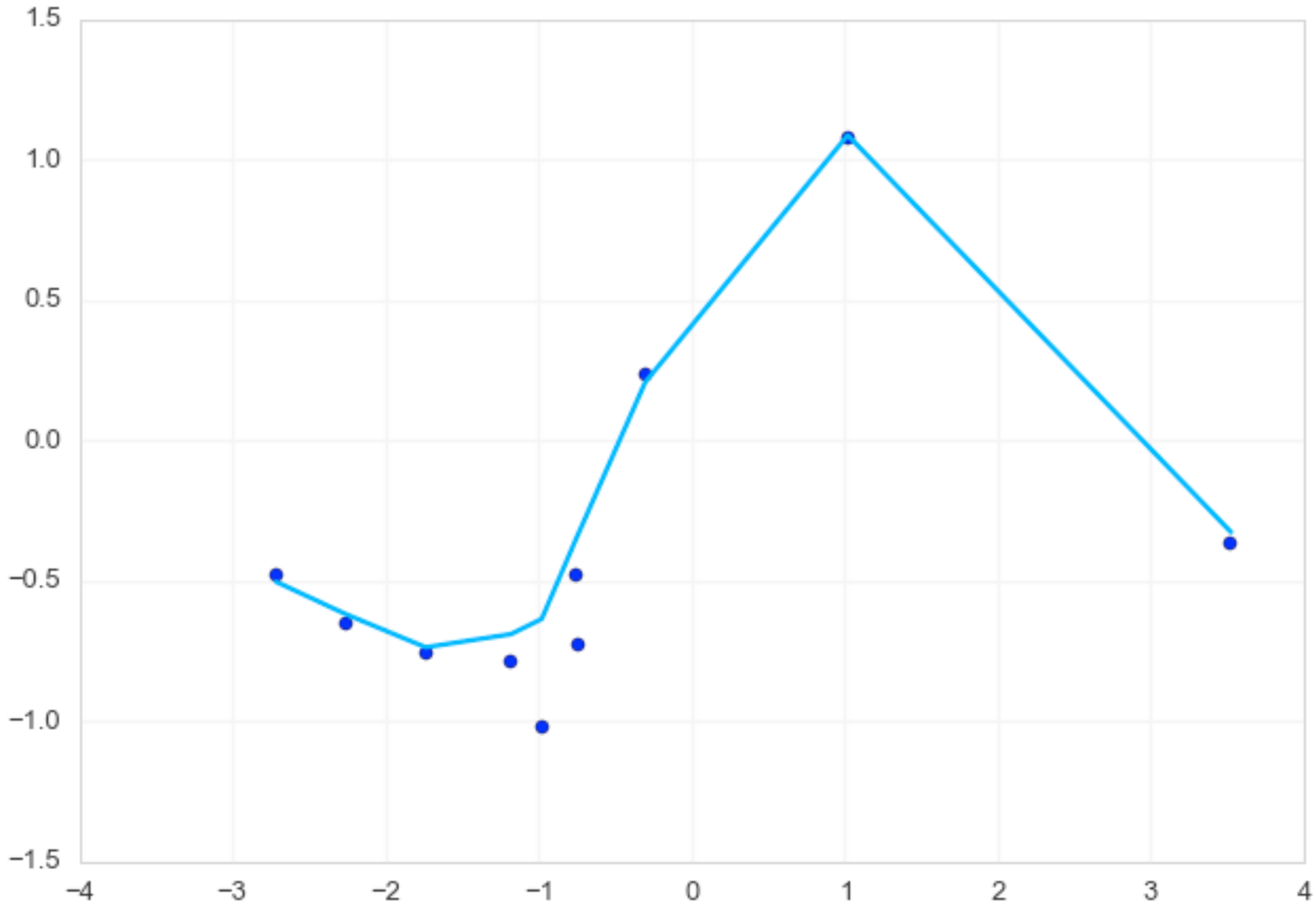
Python packages:
`statsmodels.nonparametric.smoothers_lowess.lowess`
`cylowess`

This is an area where R is better, but cylowess is decent.

Locally linear regression

```
import cylowess
c_lowess = cylowess.lowess

res_c = c_lowess(y,x)
plt.plot(res_c[:, 0], res_c[:, 1])
```

Note the order!!



Gaussian process regression

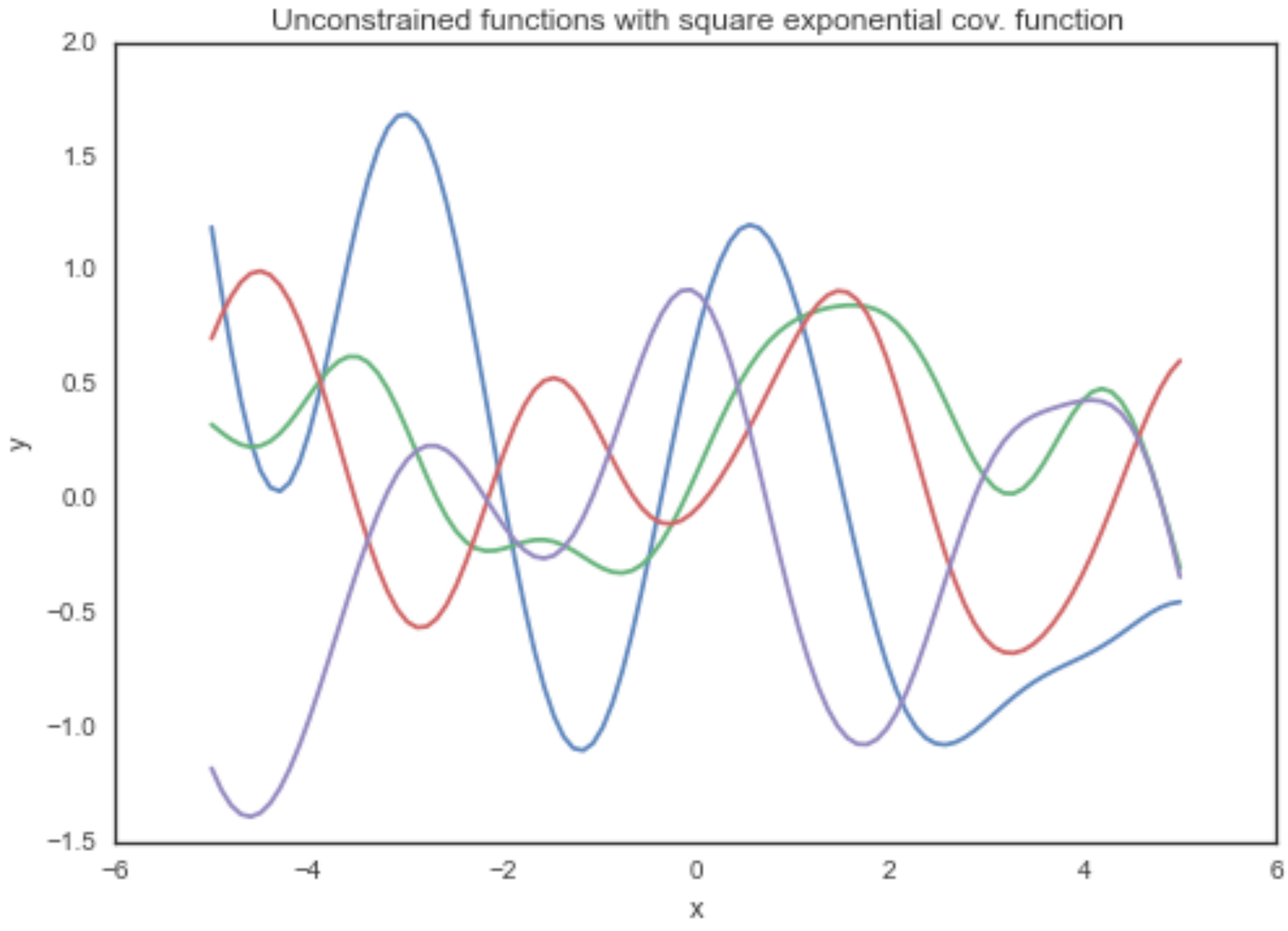
In this case we apply a prior in function space - this prior is specified using a mean & covariance function (since that is all we need for a Gaussian). The most common is:

$$\mathrm{Cov}\,(x, x') = K(x, x') = \exp\left(-\frac{|x - x'|}{2h}\right)$$

If we set the mean to zero, we can then draw random functions because at each x we know what the covariance matrix should be and that is all we need.

Gaussian process regression

Random functions - h=1

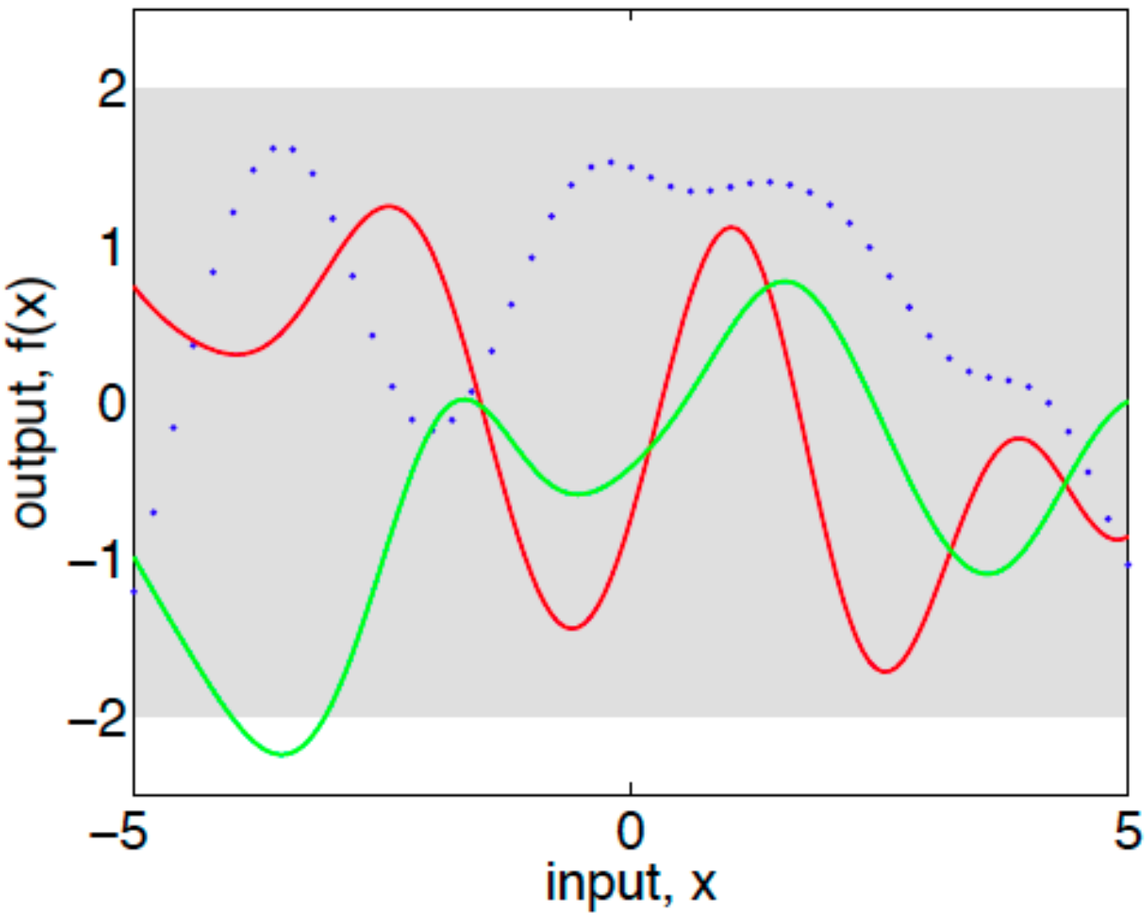


See the Jupyter notebook on the Github site:

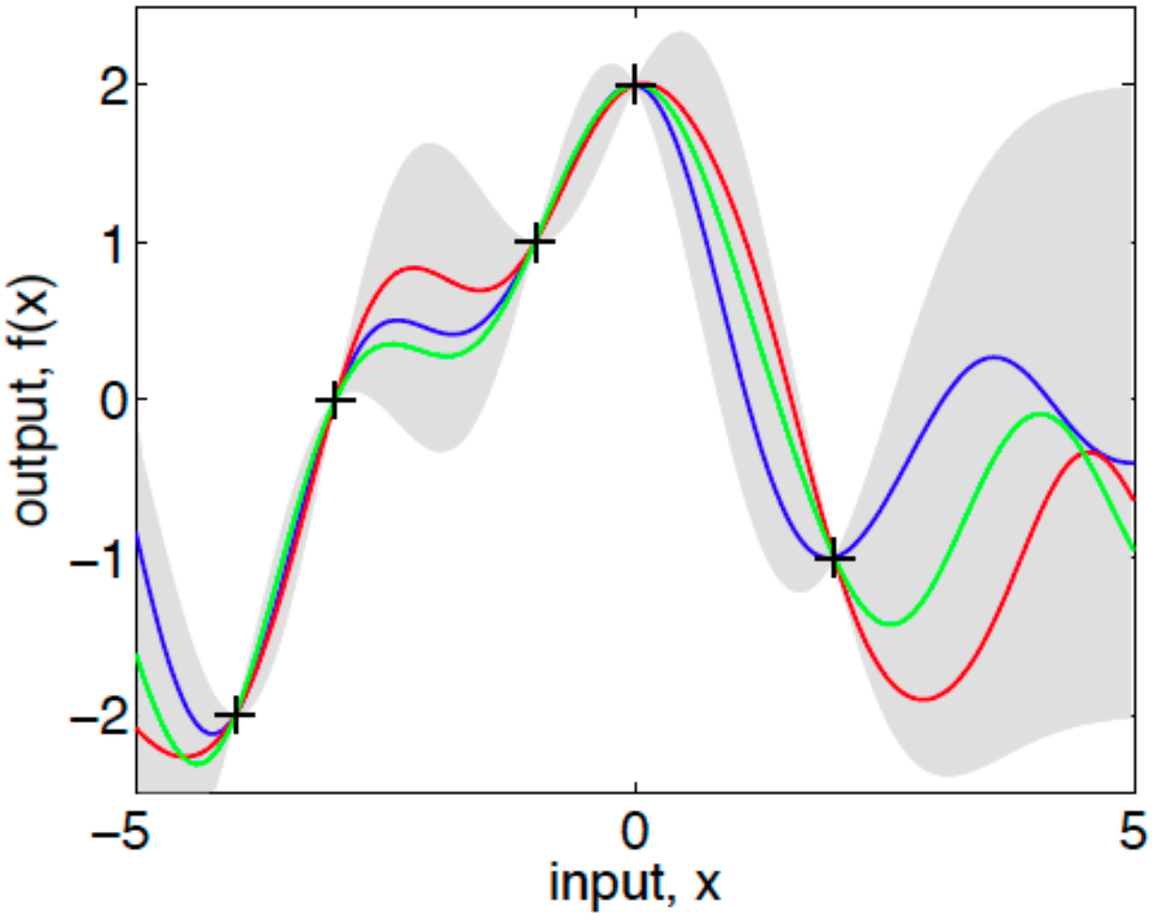
Lectures/Lecture 4/Notebooks/Gaussian process regression.ipynb

Gaussian process regression

We apply constraints by multiplying the prior with the likelihood:



(a), prior



(b), posterior

Taken from Rasmussen & Williams, “Gaussian processes for Machine Learning”, 2006, Figure 2.2
http://www.gaussianprocess.org/

Gaussian Process Regression - features

\+ Very flexible

\+ Provides covariance estimates on predictions

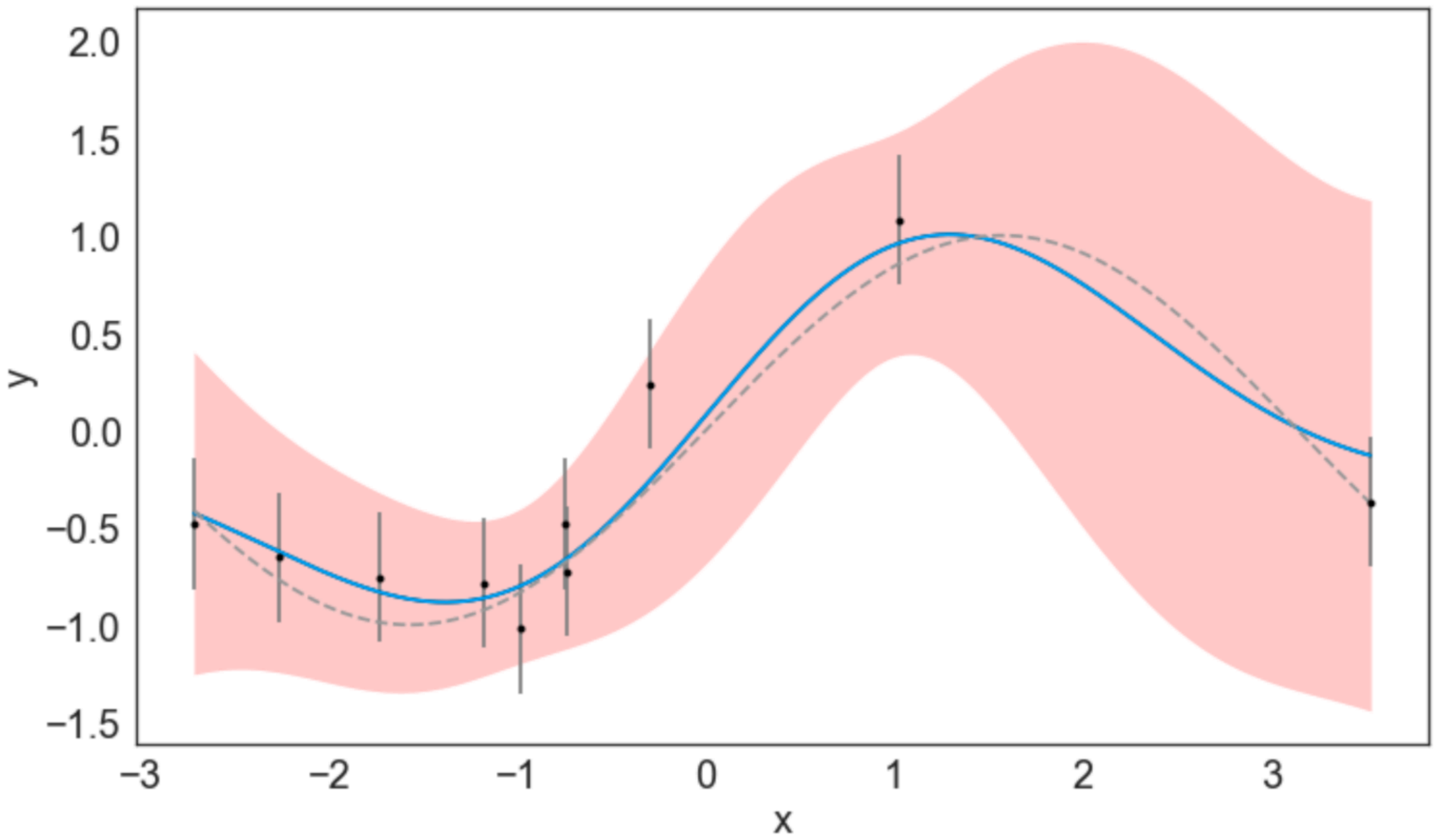
\- Fairly slow

Python usage:

```
from sklearn.gaussian_process import GaussianProcessRegressor
from sklearn.gaussian_process.kernels import RBF, WhiteKernel

gp = GaussianProcessRegressor(kernel=RBF(0.1),
                              alpha=(dy/y)**2)
```

Example use:

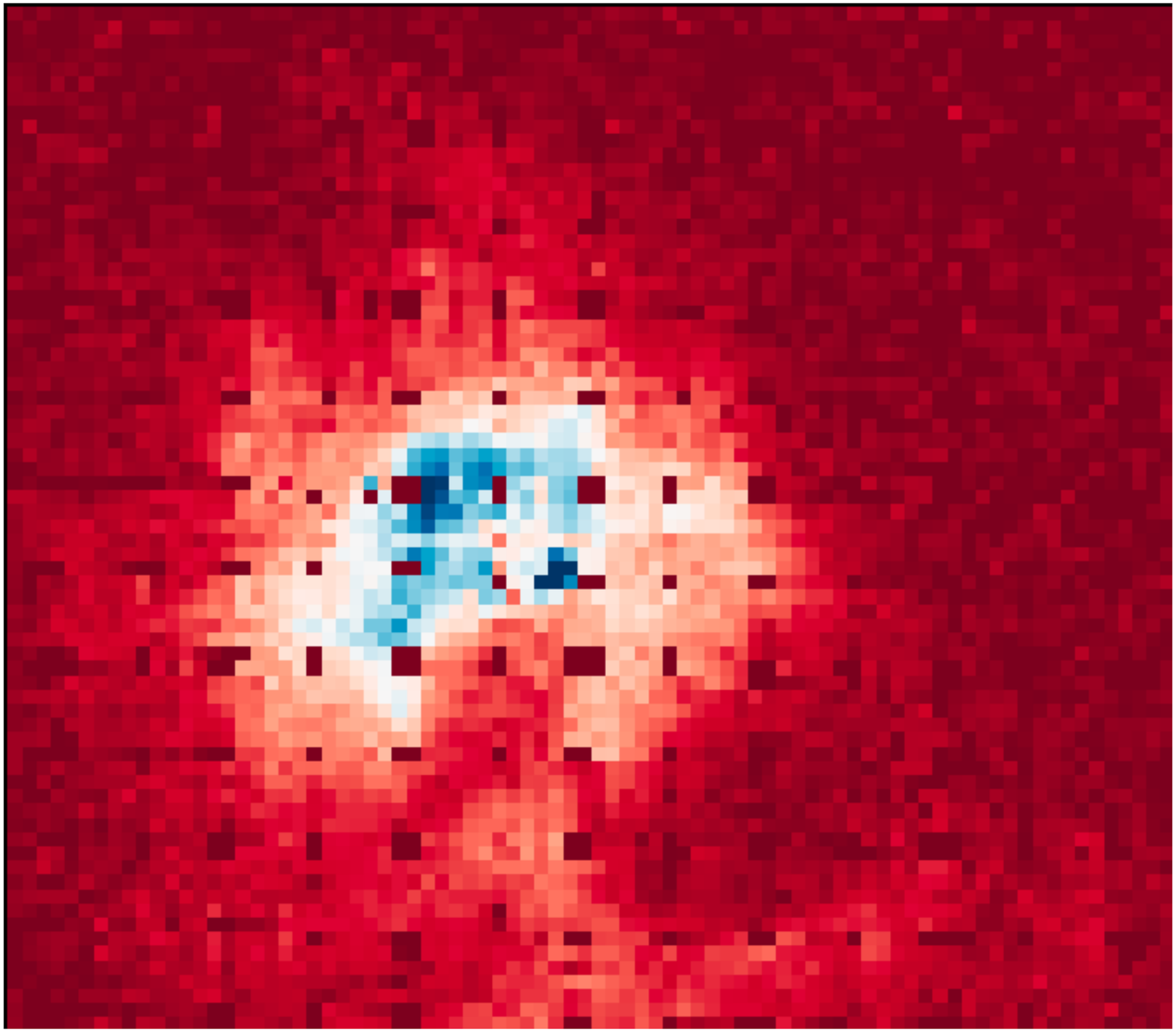
```
g = gp.fit(x[:, np.newaxis], y)
y_pred, sigma = gp.predict(xplot[:, np.newaxis], return_std=True)
plot_a_fit(x, y, dy, xplot, y_pred, sigma, include_true=True)
```



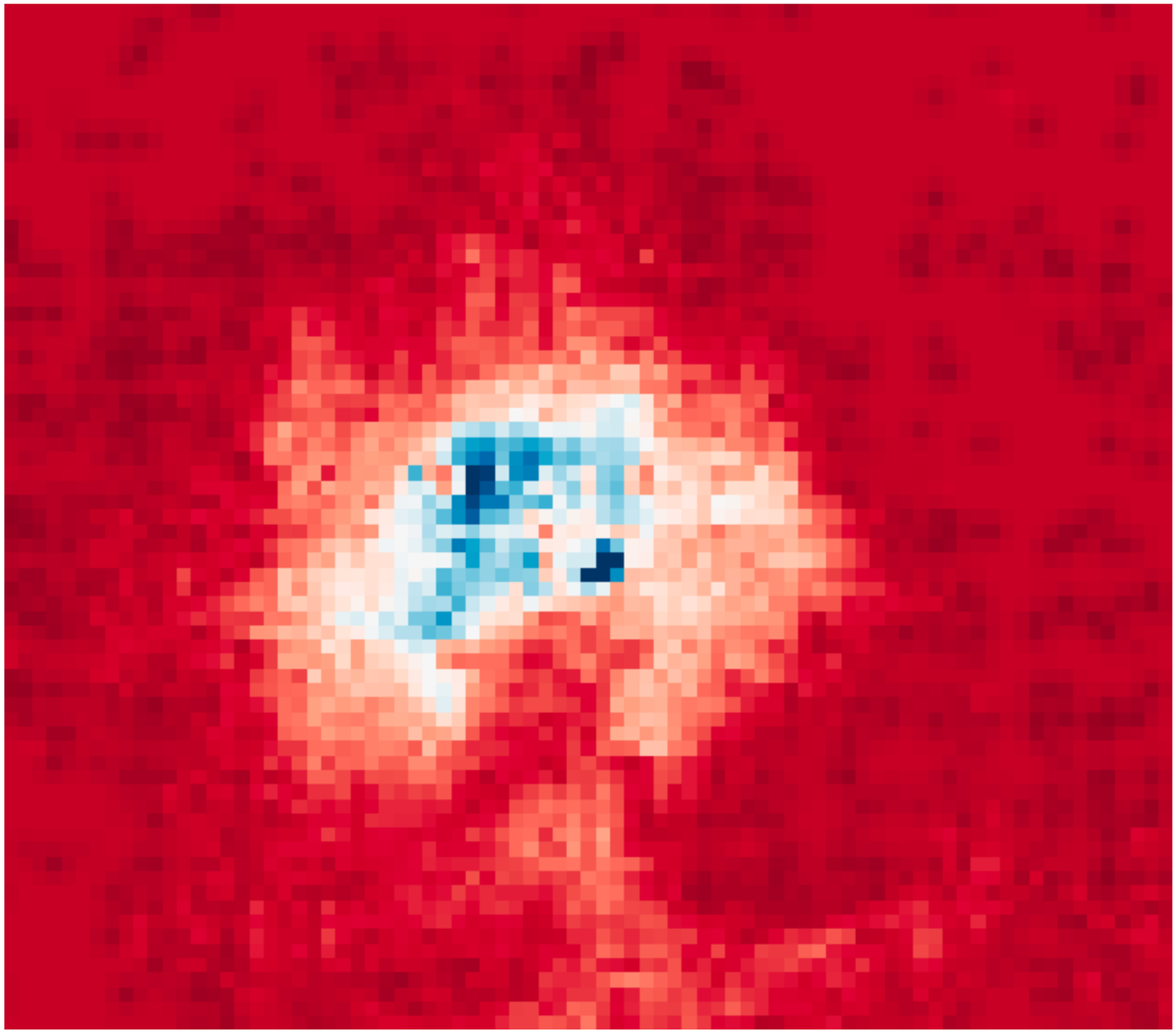
See the Jupyter notebook on the Github site:

Lectures/Lecture 4/Notebooks/Gaussian process regression.ipynb

Example for an image



Example for an image



Usefulness in astronomy

- Light-curve modelling: stars (Brewer & Stello 2009), AGNs (Kelly et al 2014), X-ray binaries (Uttley et al 2005).
- Gaussian random field models for cosmic microwave background and large scale structure (e.g. Bond & Efstathiou 1987) are also possible to cast as a Gaussian Process.
- Quasar time-delay modelling (Hojjati et al 2013).
- Spectroscopic calibration (Czekala et al 2017).
- Widely used also as part of a larger model.

Usefulness in astronomy

- Widely used - also as part of a larger model.

Example: Model emulation to create galaxy formation models -
Rodrigues, Vernon & Bower (2016, MNRAS, **466**:2, 2418)

Semi-analytic models of galaxy formation. Many parameters, expensive model to calculate.

The 20 parameters considered in Rodrigues et al:

Process modelled	Section	Parameter name [units]	Range		GP14	Scaling
Star formation (quiescent)	§2.2.1	ν_{sf} [Gyr^{-1}]	0.025	1.0	0.5	lin
		P_{sf}/k_B [cm^{-3}K]	1×10^4	5×10^4	1.7×10^4	log
		β_{sf}	0.65	1.10	0.8	lin
Star formation (bursts)	§2.2.2	f_{dyn}	1.0	100.0	10	log
		$\tau_{min,burst}$ [Gyr]	10^{-3}	1	0.05	log
SNe feedback	§2.2.3	α_{hot}	1.0	3.7	3.2	lin
		$\beta_{0,burst}$	0.5	40.0	11.16	lin
		$\beta_{0,disc}$	0.5	40.0	11.16	lin
		α_{reheat}	0.15	1.5	1.26027	lin
AGN feedback	§2.2.4	α_{cool}	0.1	2.0	0.6	log
		ε_{edd}	0.004	0.1	0.03979	log
		f_{smbh}	0.001	0.01	0.005	lin
Galaxy mergers		f_{burst}	0.01	0.5	0.1	log
		f_{ellip}	0.01	0.5	0.3	log
Disk stability	§2.2.5	f_{stab}	0.61	1.1	0.8	lin
Reionization		V_{cut} [km s^{-1}]	20	60	30	lin
		z_{cut}	5	15	10	lin
Metal enrichment		p_{yield}	0.02	0.05	0.021	lin
Ram pressure stripping		ε_{strip}	0.01	0.99	n/a	lin
		α_{rp}	1.0	3.0	n/a	lin

Rodrigues, Vernon & Bower (2016, MNRAS, **466**:2, 2418)

Fitting data to model

The model is prohibitively expensive to run millions/billions of times, so how do you fit this in a Bayesian framework?

Write the vector of model outputs as:

$$\mathbf{M}\left(\nu_{0,\mathrm{sf}}, P_{\mathrm{sf}}/k_{\mathrm{B}}, \ldots, \alpha_{\mathrm{rp}}\right) = \mathbf{M}(\theta)$$

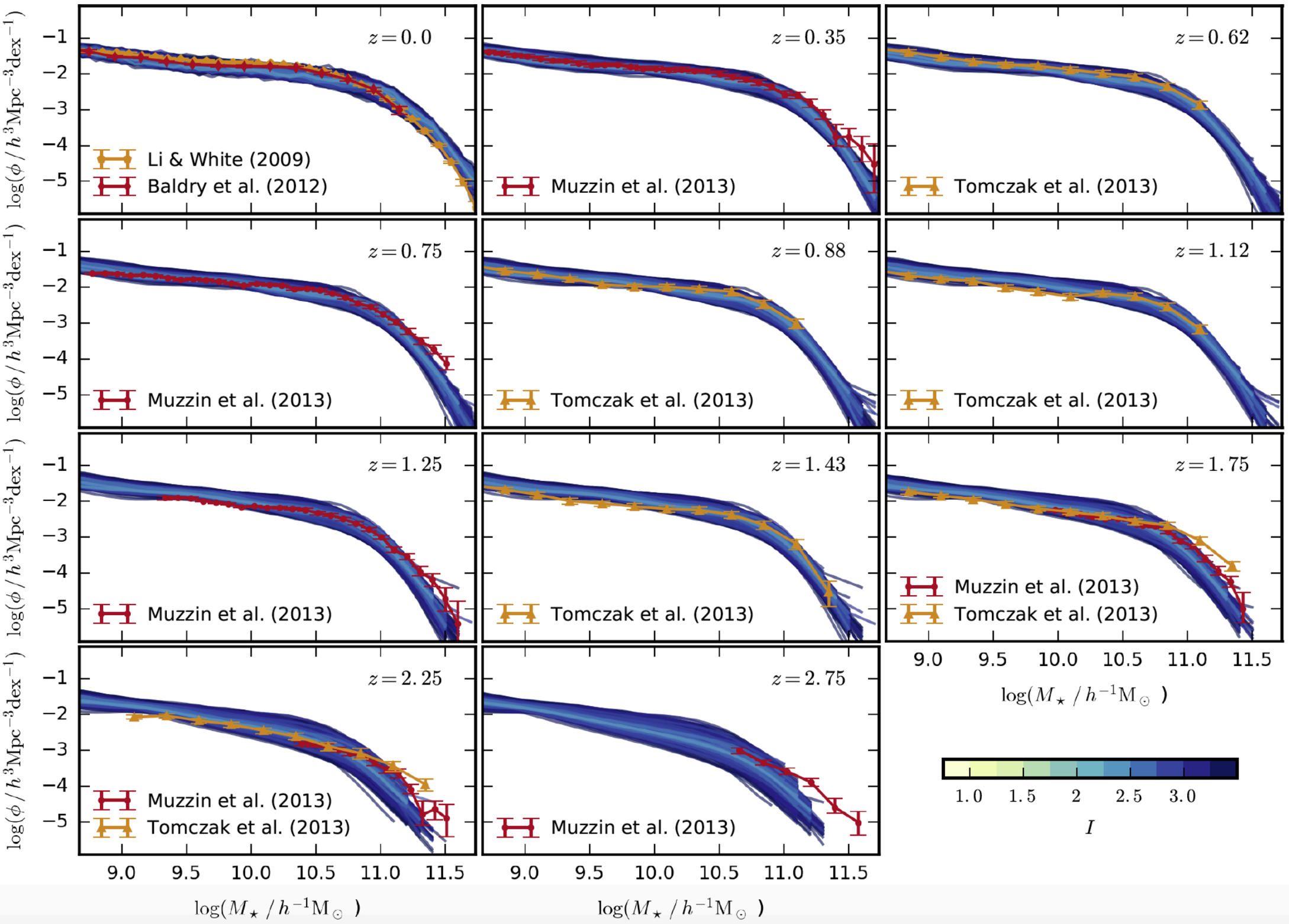
Then write this as a regression problem:

$$M_i(\theta) = \sum_j \beta_{ij} g_{ij}(\theta) + u_i(\theta) + \nu_i(\theta)$$

Polynomials

Gaussian Process

Fitting data to model - galaxy mass functions



Rodrigues, Vernon & Bower (2016, MNRAS, **466**:2, 2418)

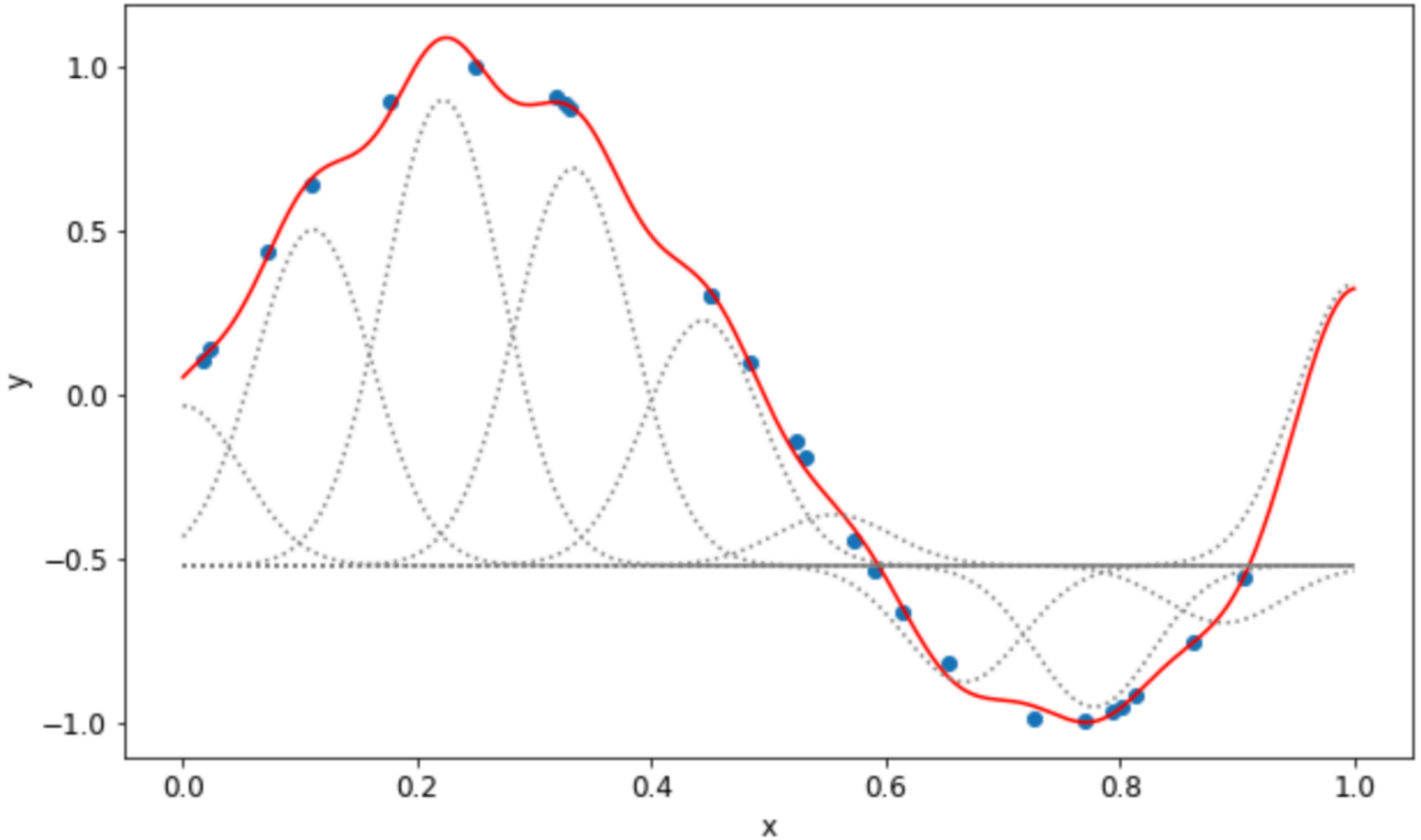
Basis function regression

Finally, let us return to an earlier topic - basis function regression.

$$f(x) = \sum_i w_i \phi_i(x)$$

This is linear regression, but transforming the x values.

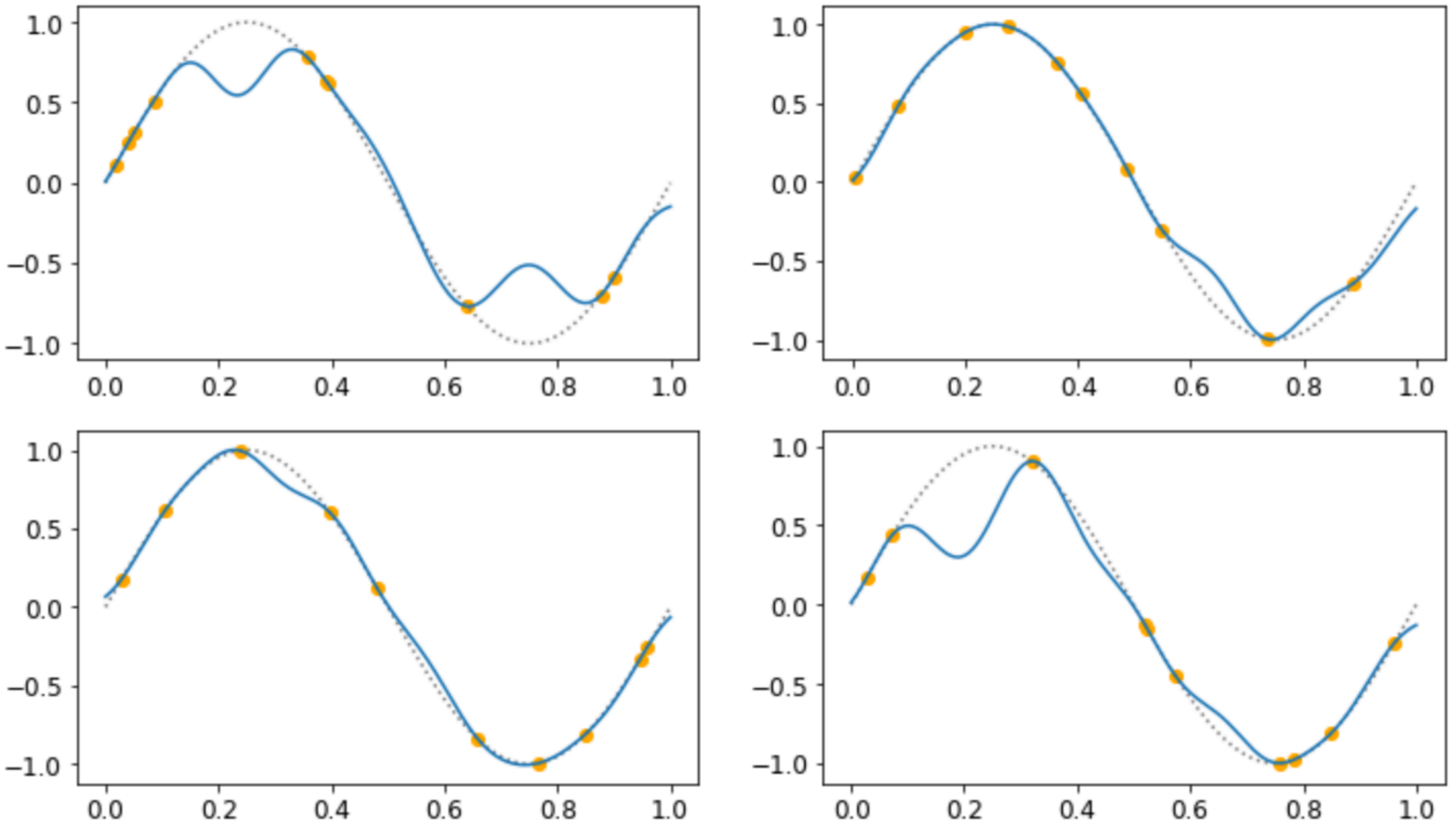
e.g. Gaussian basis:



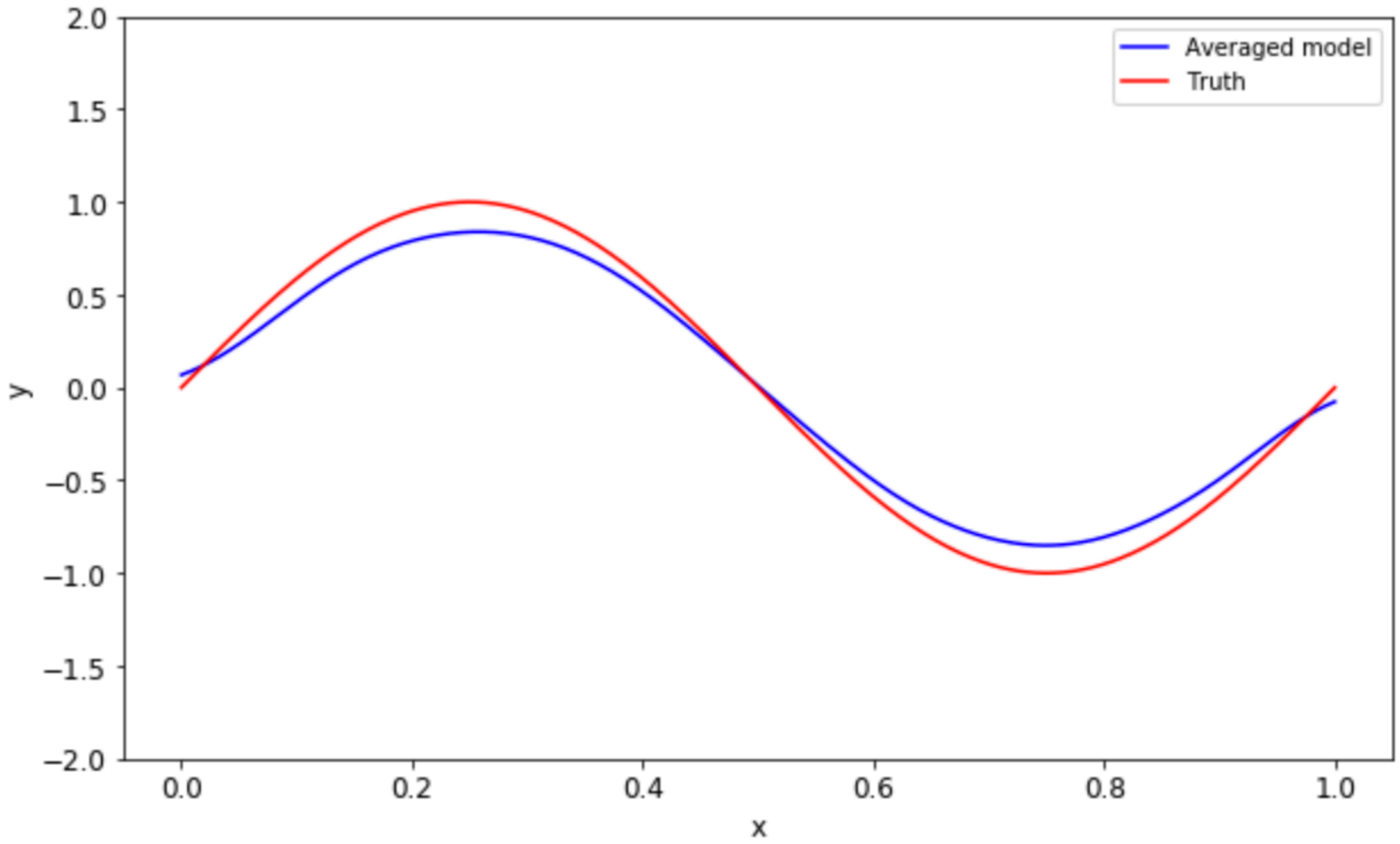
Gaussian-basis function regression

(see notebook under Lectures/Lecture 4/Notebooks/)

Create 100 datasets with 10 points each, fit these with 10 Gaussians with fixed width. Thus easily affected by overfitting, but low bias.



Combining it all



So averaging the results of the fits gives a better final result, less sensitive to overfitting.

Can this be generalised? We usually do not have N indepedent samples…

Ensemble methods

Bagging

- Draw bootstrap realisations of your data.
- Fit these data.
- Average the result.

This is also known as Bootstrap aggregating.

It typically can help reduce overfitting problems.

In the case of perfectly uncorrelated errors in prediction, the error goes down as $\frac{\sigma}{\sqrt{N_{\mathrm{models}}}}$

If the errors are completely correlated, the error stays the same - ie. bagging has no effect.

Decision trees - an intermezzo

These are tree structures where at each level you split the data according to some criterion.

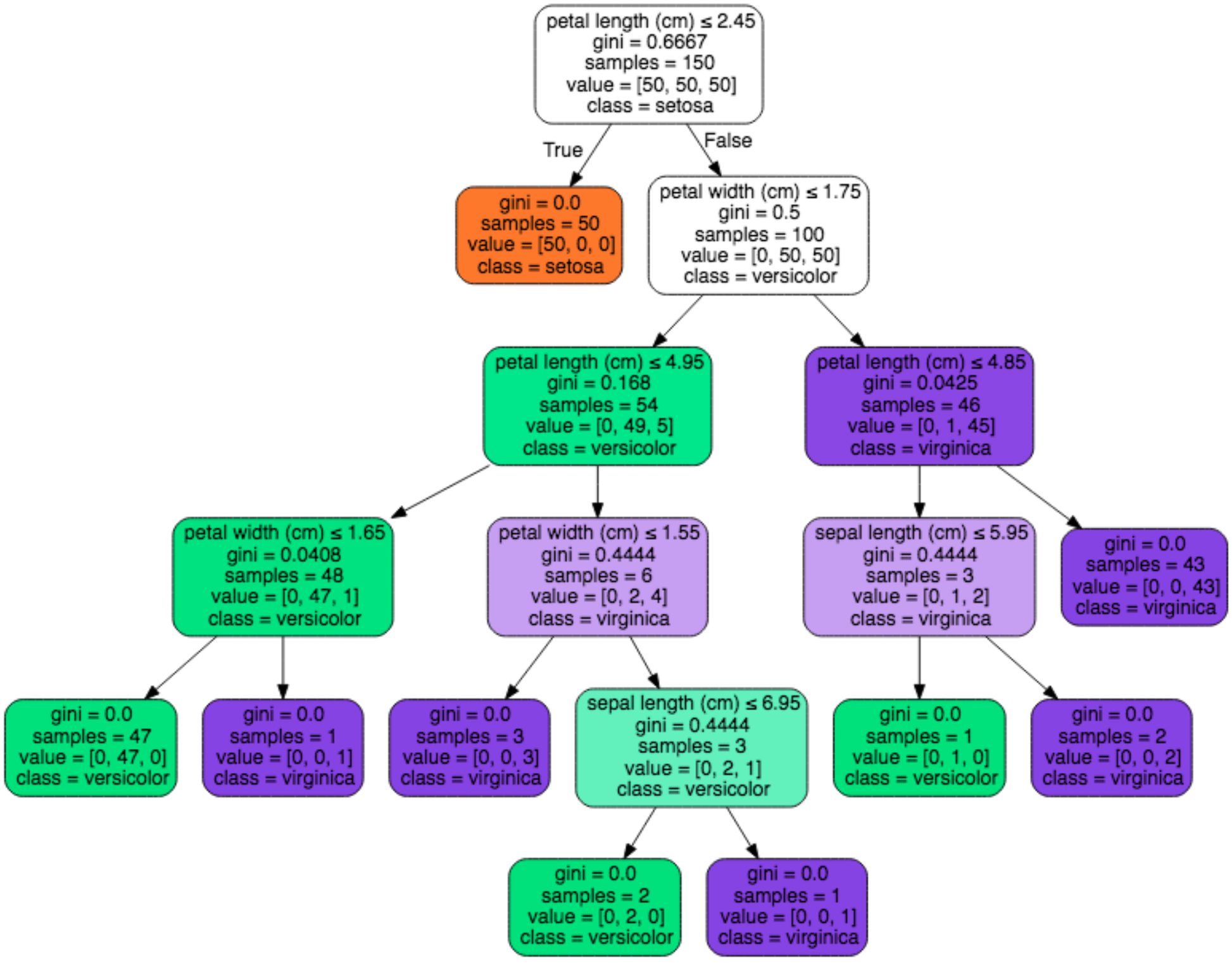
By making the tree deeper you can catch finer details in the data.

These are powerful techniques and can be inspected afterwards (good!) but do not usually have very high accuracy (not so good), and they have high variance (not ideal).

In sklearn:

```
from sklearn import tree
<get your data>
clf = tree.DecisionTreeClassifier()
clf = clf.fit(X, Y)
```

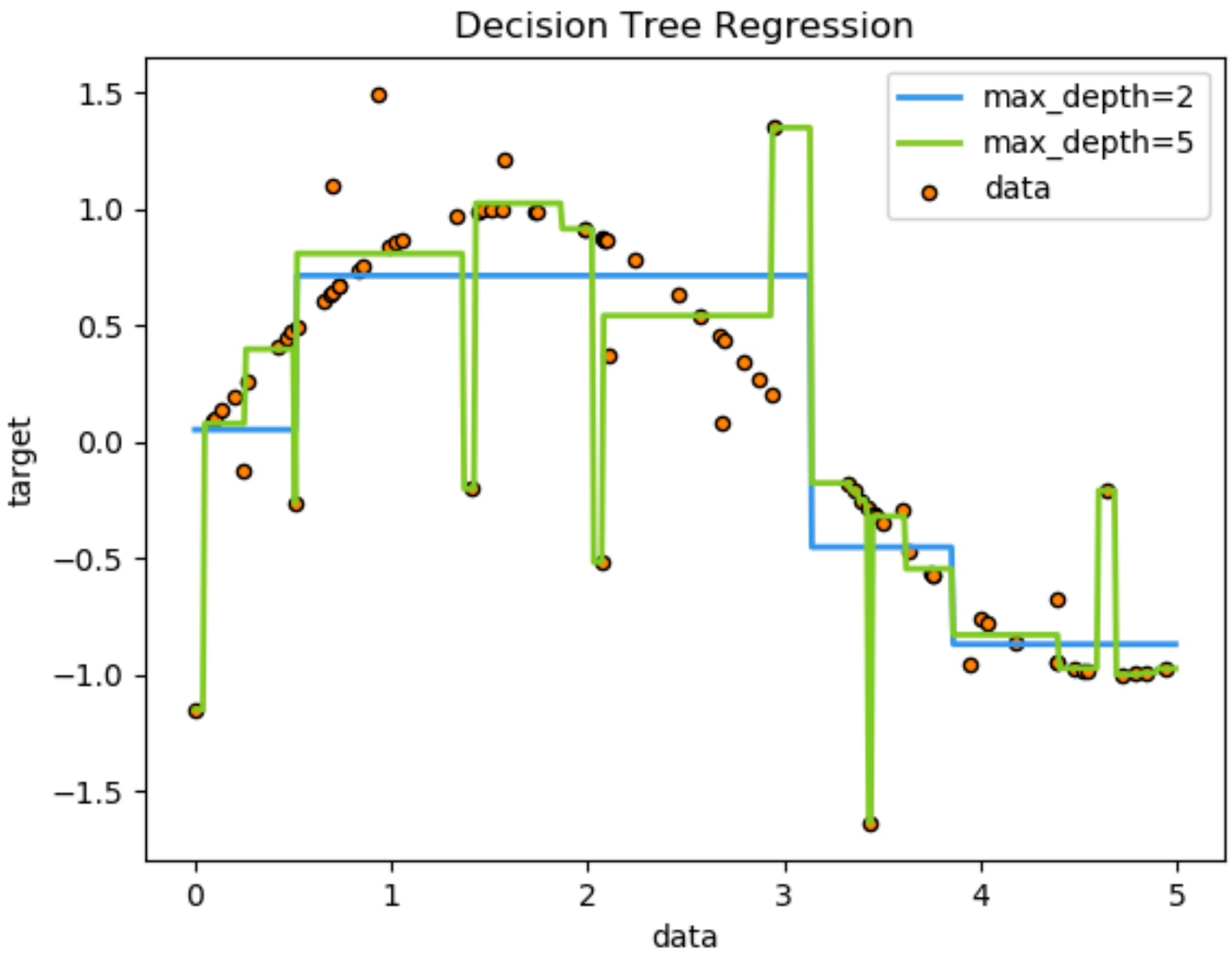
Decision trees - an intermezzo



http://scikit-learn.org/stable/modules/tree.html

Decision trees - an intermezzo

It can also be used for regression:



http://scikit-learn.org/stable/modules/tree.html

Random Forests

The bagging method can be applied to decision trees too. However if you combine bagging with randomly choosing a subset of features at each level, you get **random forests** and these seem to perform particularly well.

Basic algorithm (https://www.stat.berkeley.edu/~breiman/RandomForests/cc_home.htm

```
from sklearn.ensemble import RandomForestClassifier
```

For each tree:

1. Sample N cases at random - but with replacement, from the original data
2. At each node, m variables are selected at random out of the M and the best split on these m is used to split the node. The value of m is held constant during the forest growing.
3. Each tree is grown to the largest extent possible. There is no pruning.

The random subsets is to avoid correlation (recall the bagging error function)

Boosting

These are techniques that combine many very simple algorithms ("weak learners") to create a powerful final algorithm.

A weak learner typically is chosen to have a low bias, so as a consequence has high variance. The aim of boosting is to combine these to get a low bias, low variance estimator.

Boosting

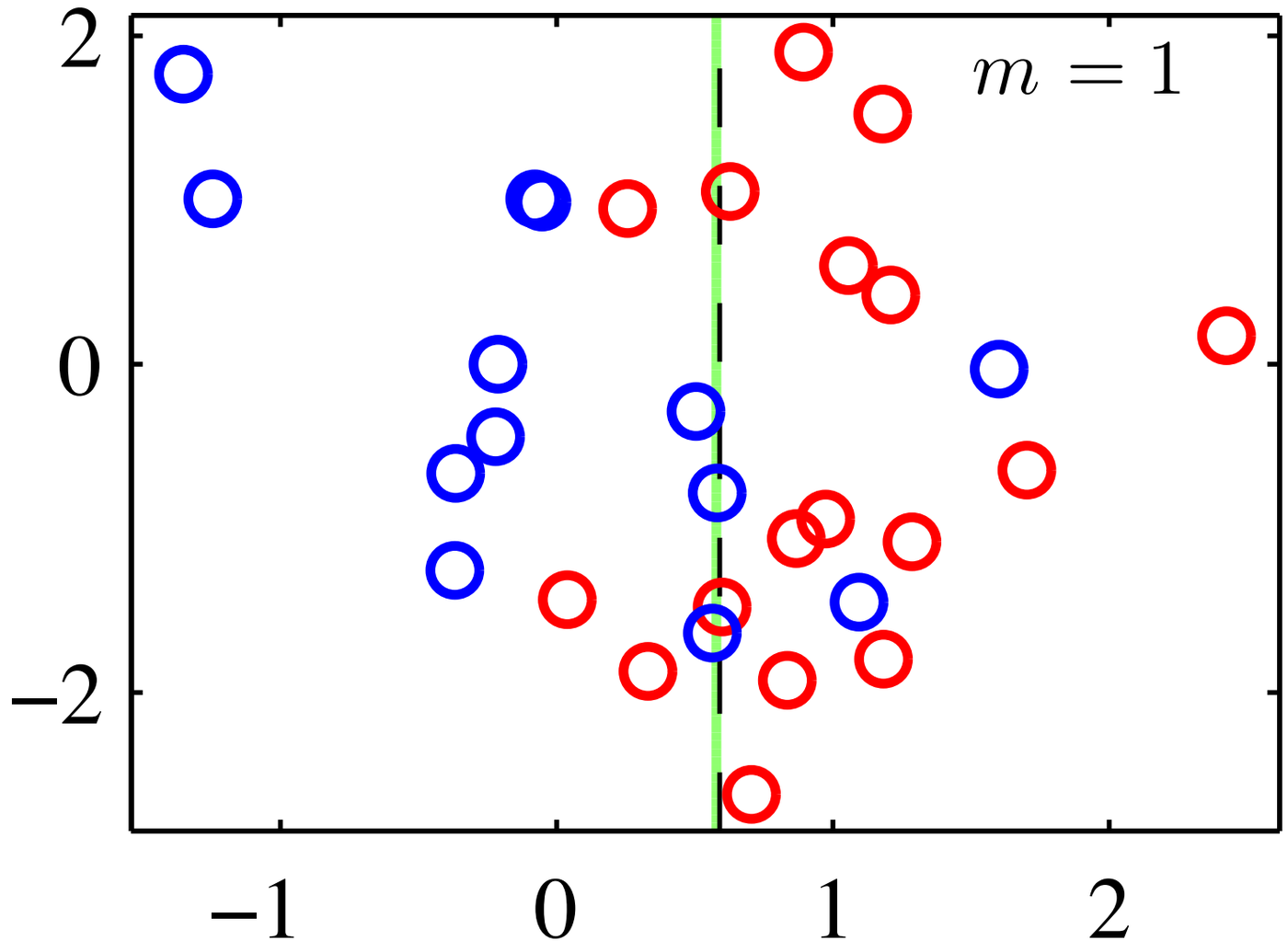
These are techniques that combine many very simple algorithms ("weak learners") to create a powerful final algorithm.

The approach (AdaBoost):

1. Fit a learner (algorithm)
   Find how well this works and give high weight to those examples it did **not** fit.
   Repeat.
2. Average the results using weights estimated during the fitting procedure.

Boosting

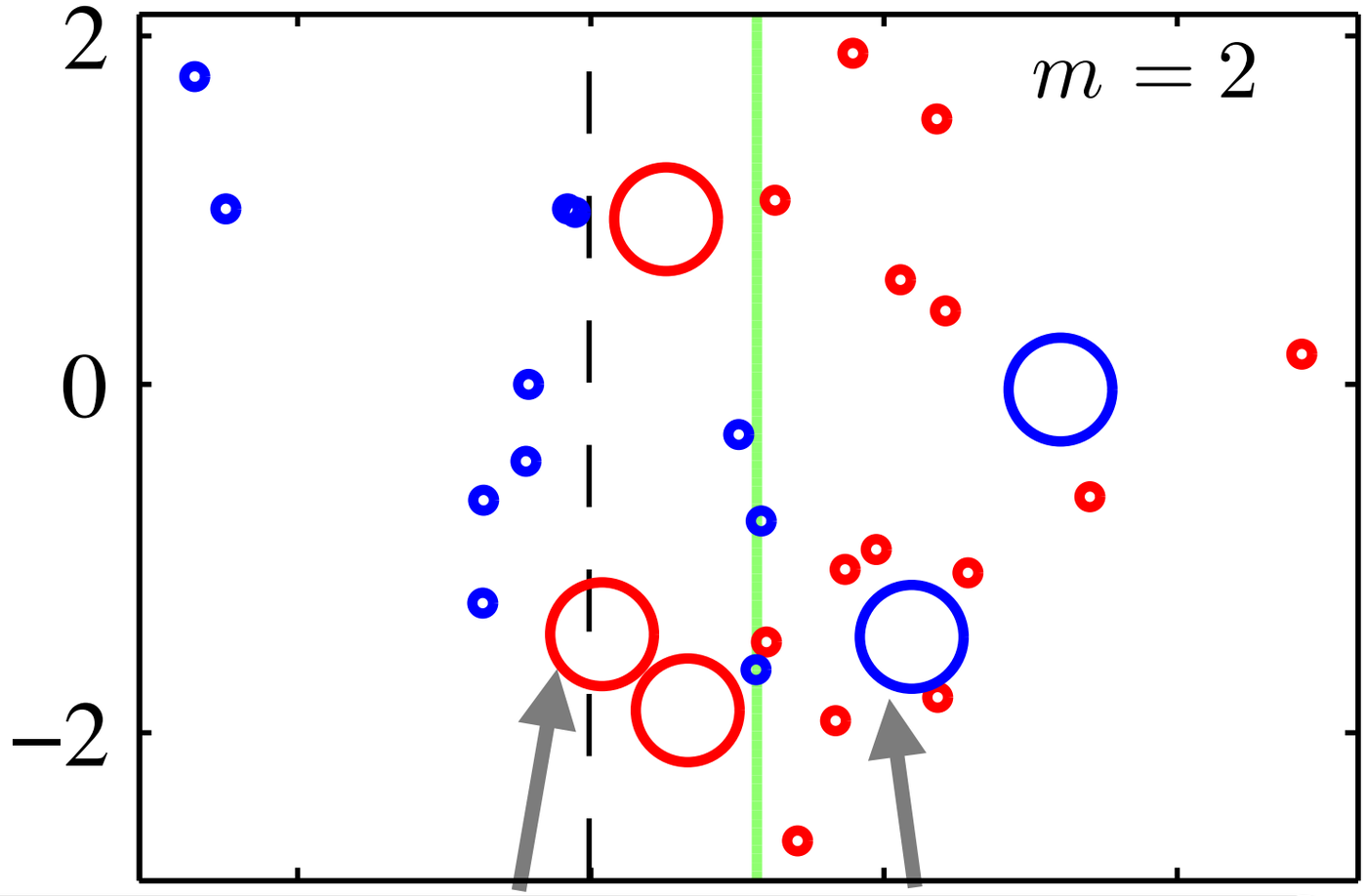
These are techniques that combine many very simple algorithms ("weak learners") to create a powerful final algorithm.



From Bishop "Pattern Recognition and Machine Learning", Figure 14.2

Boosting

These are techniques that combine many very simple algorithms ("weak learners") to create a powerful final algorithm.

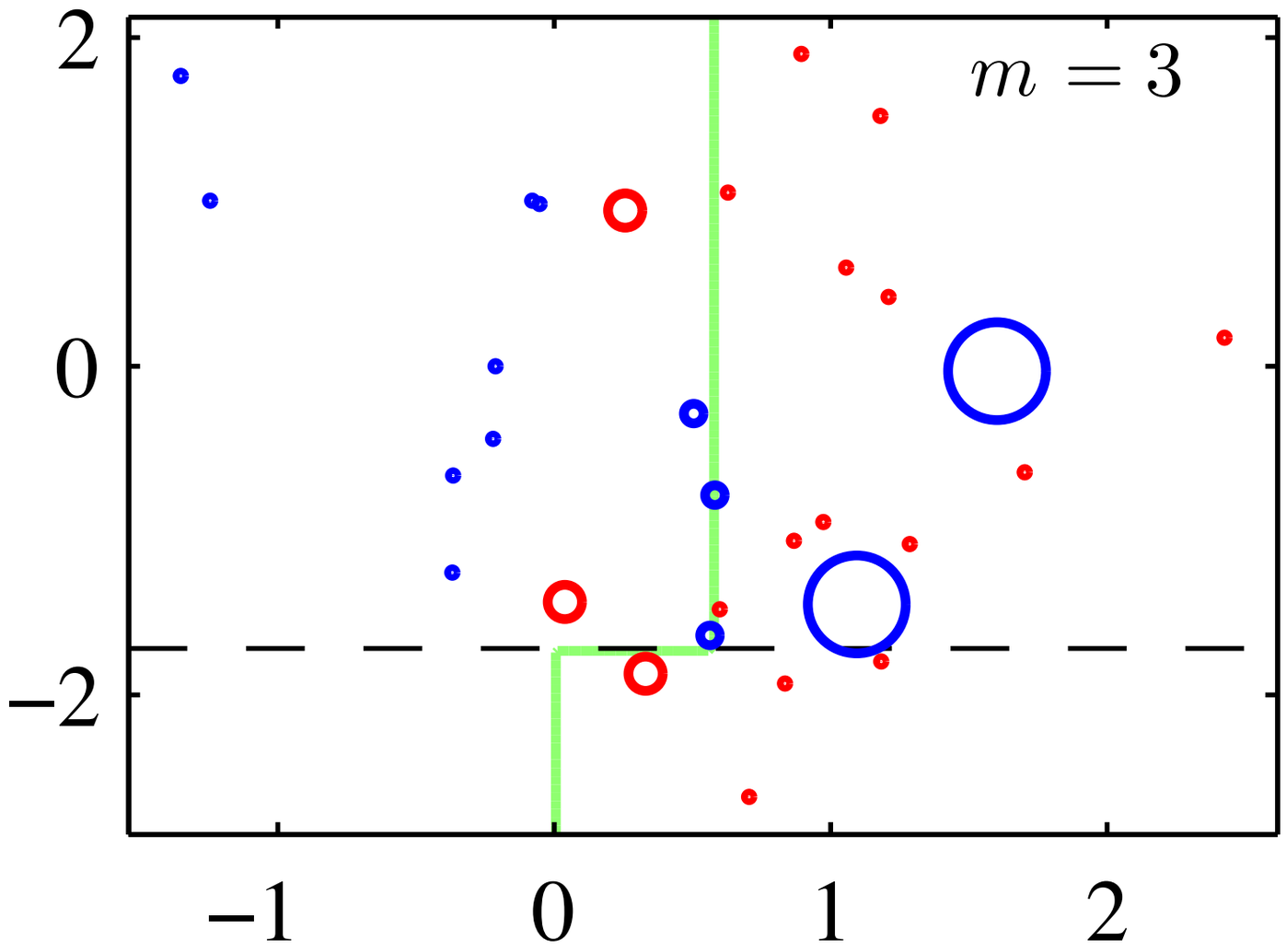


Higher weight to misclassified examples

From Bishop "Pattern Recognition and Machine Learning", Figure 14.2

Boosting

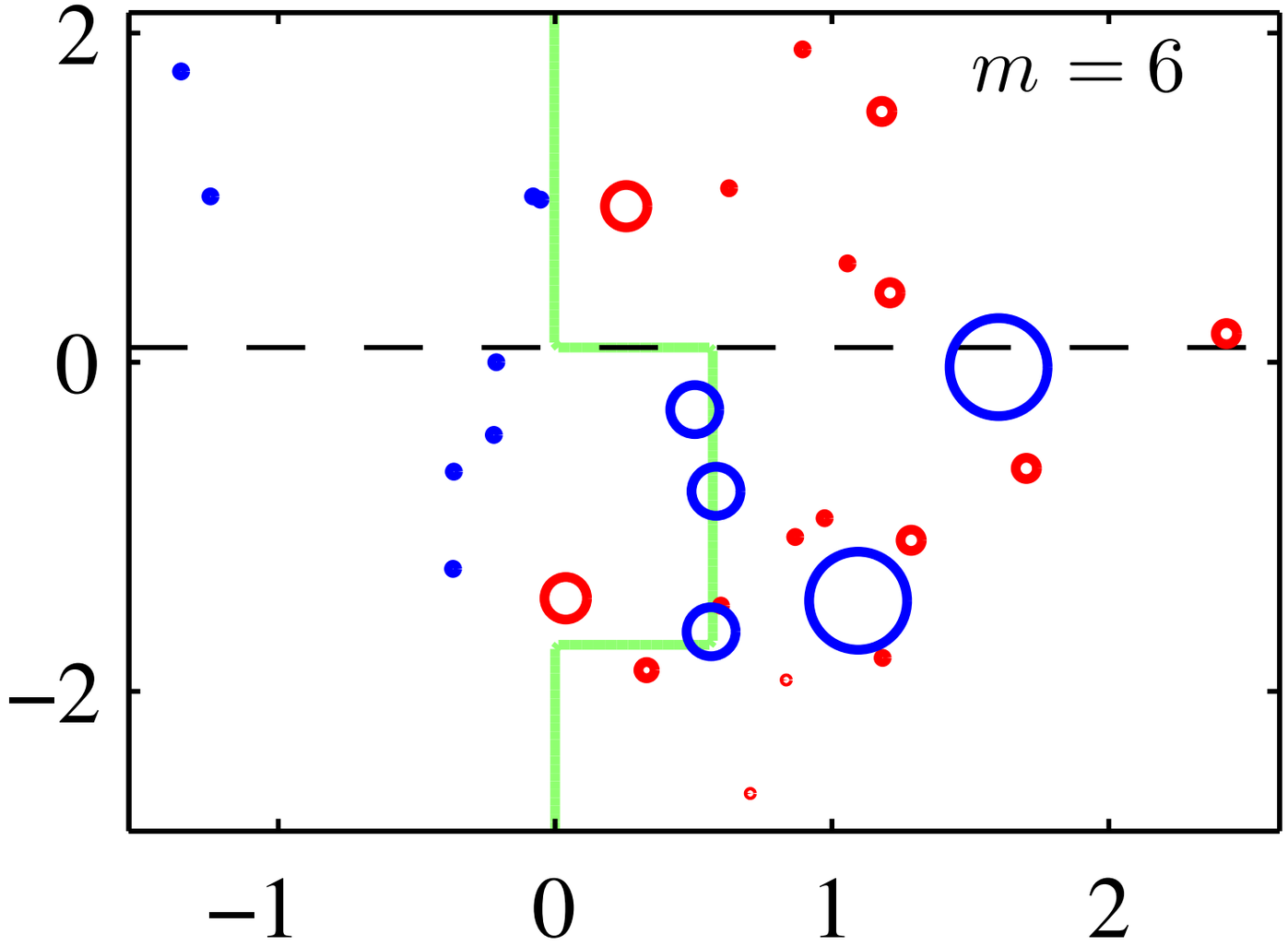
These are techniques that combine many very simple algorithms ("weak learners") to create a powerful final algorithm.



From Bishop "Pattern Recognition and Machine Learning", Figure 14.2

Boosting

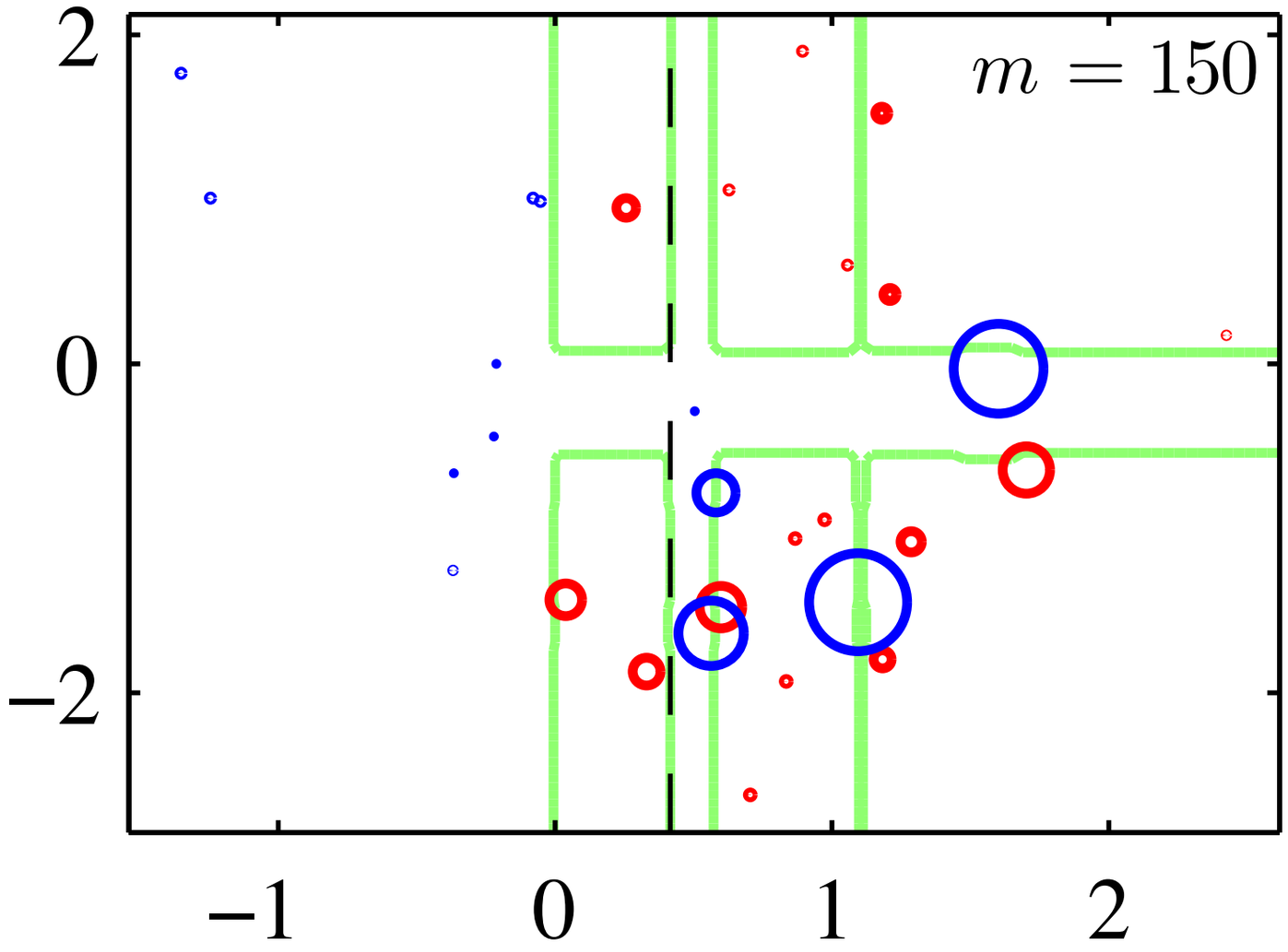
These are techniques that combine many very simple algorithms ("weak learners") to create a powerful final algorithm.



From Bishop "Pattern Recognition and Machine Learning", Figure 14.2

Boosting

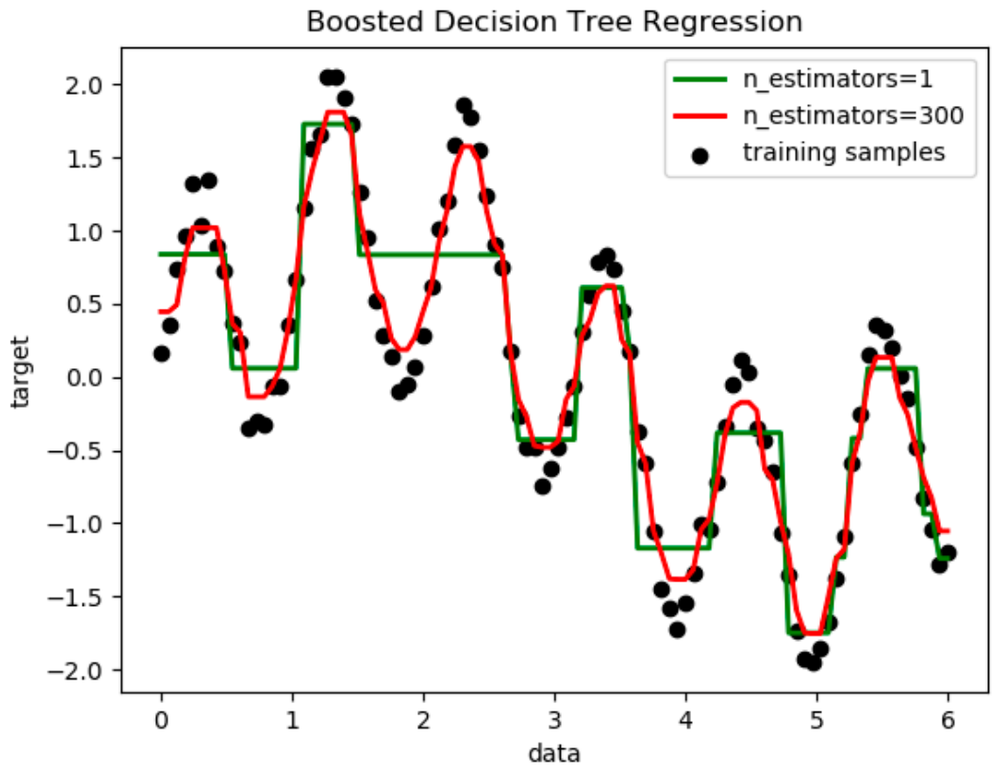
These are techniques that combine many very simple algorithms ("weak learners") to create a powerful final algorithm.



From Bishop "Pattern Recognition and Machine Learning", Figure 14.2

Boosting

These are techniques that combine many very simple algorithms ("weak learners") to create a powerful final algorithm.



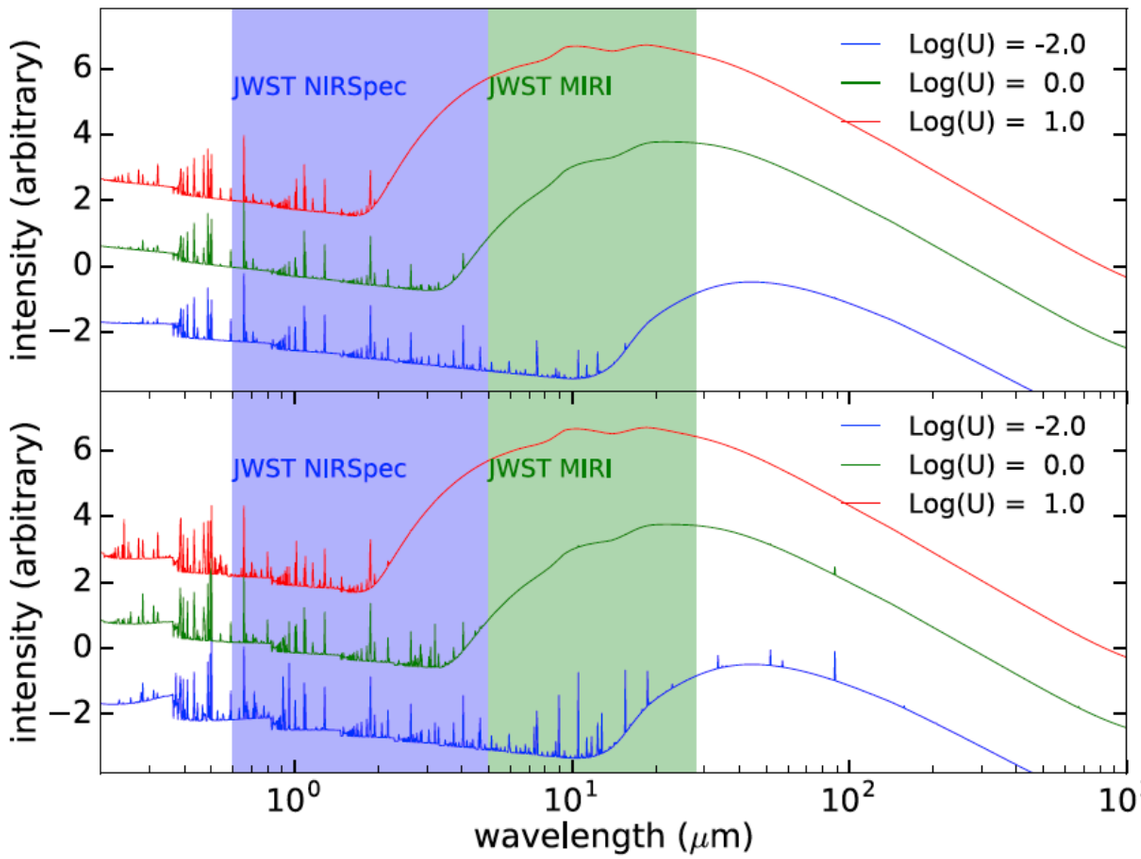

```
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import AdaBoostRegressor
<…>
regr = AdaBoostRegressor(DecisionTreeRegressor(max_depth=4),
                         n_estimators=300)
regr.fit(X, y)
y_2 = regr.predict(X)
```

http://scikit-learn.org/stable/auto_examples/ensemble/plot_adaboost_regression.htm

Use in astronomy

AdaBoost: Ucci et al (2017): Fitting emission line spectra (code GAME)

Input library to compare against:



Use in astronomy

AdaBoost:
Ucci et al (2017): Fitting emission line spectra.
Xin et al (2017): Finding impact craters on Mars.
Zitlau et al (2016): Photometric redshifts for SDSS.
++

Random Forests:
Kuntzer & Courbin (2017): Detecting binary stars.
García-Varela et al (2017): Finding variable stars.
Jouvel et al (2017): Photometric redshifts of galaxies.
Bastien et al (2017): Classification of radio galaxies
Torres et al (2019): White dwarfs in the MW from GAIA
Ulmer-Moll (2019): Exoplanet mass-radius relation
+++