Lecture 5

Local regression, ensemble methods and neural networks

Deadline for project

Proposal:

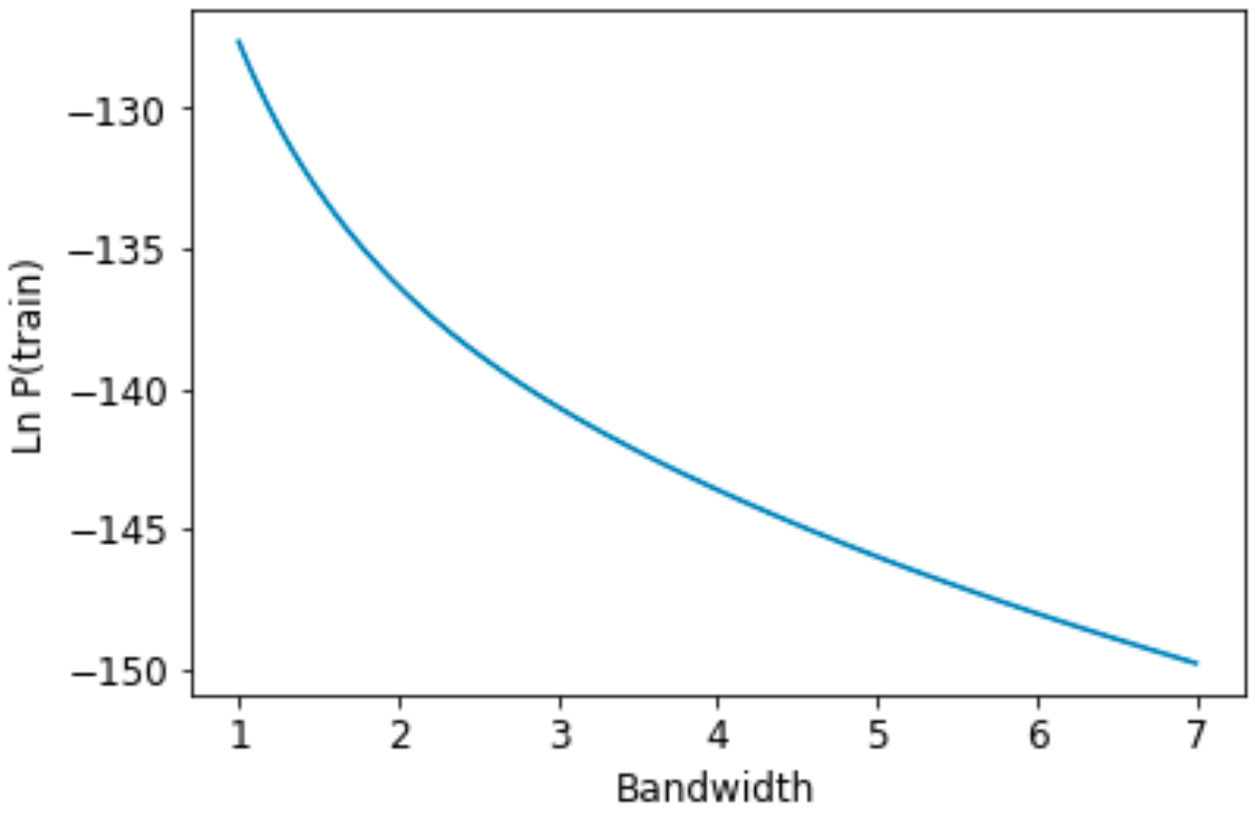
Hand out project this week (hopefully…) to have enough time to finish before Christmas.

Deadline: January 26.

First: Problem set 3

1. Determine an optimal bandwidth for black hole masses

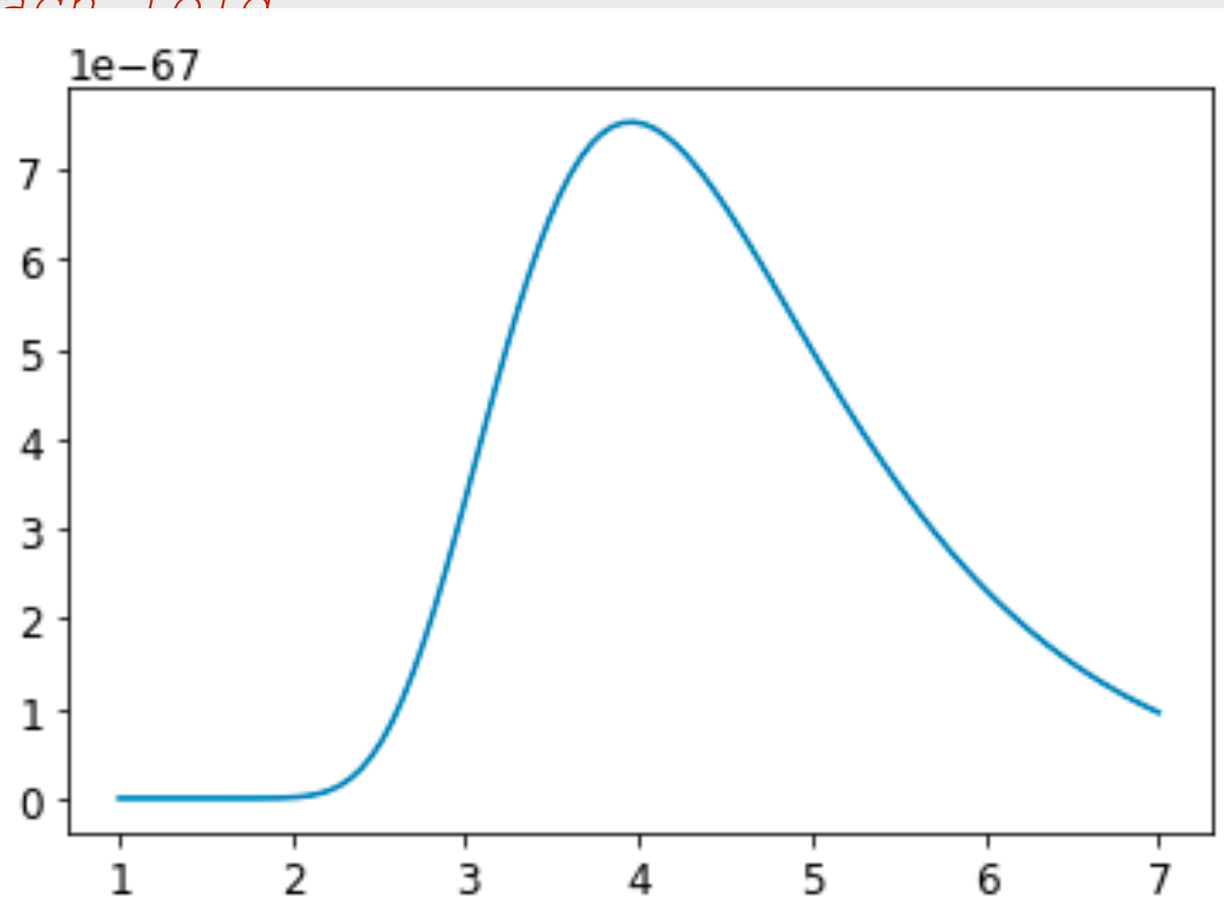
```
def score_one_bandwidth(t, bw, kernel='gaussian'):
    X = t['MBH'][:, np.newaxis]
    kde = KernelDensity(bandwidth=bw, kernel=kernel).fit(X)
    score = kde.score(X)

    return score
```



First: Problem set 3

1. Determine an optimal bandwidth for black hole masses

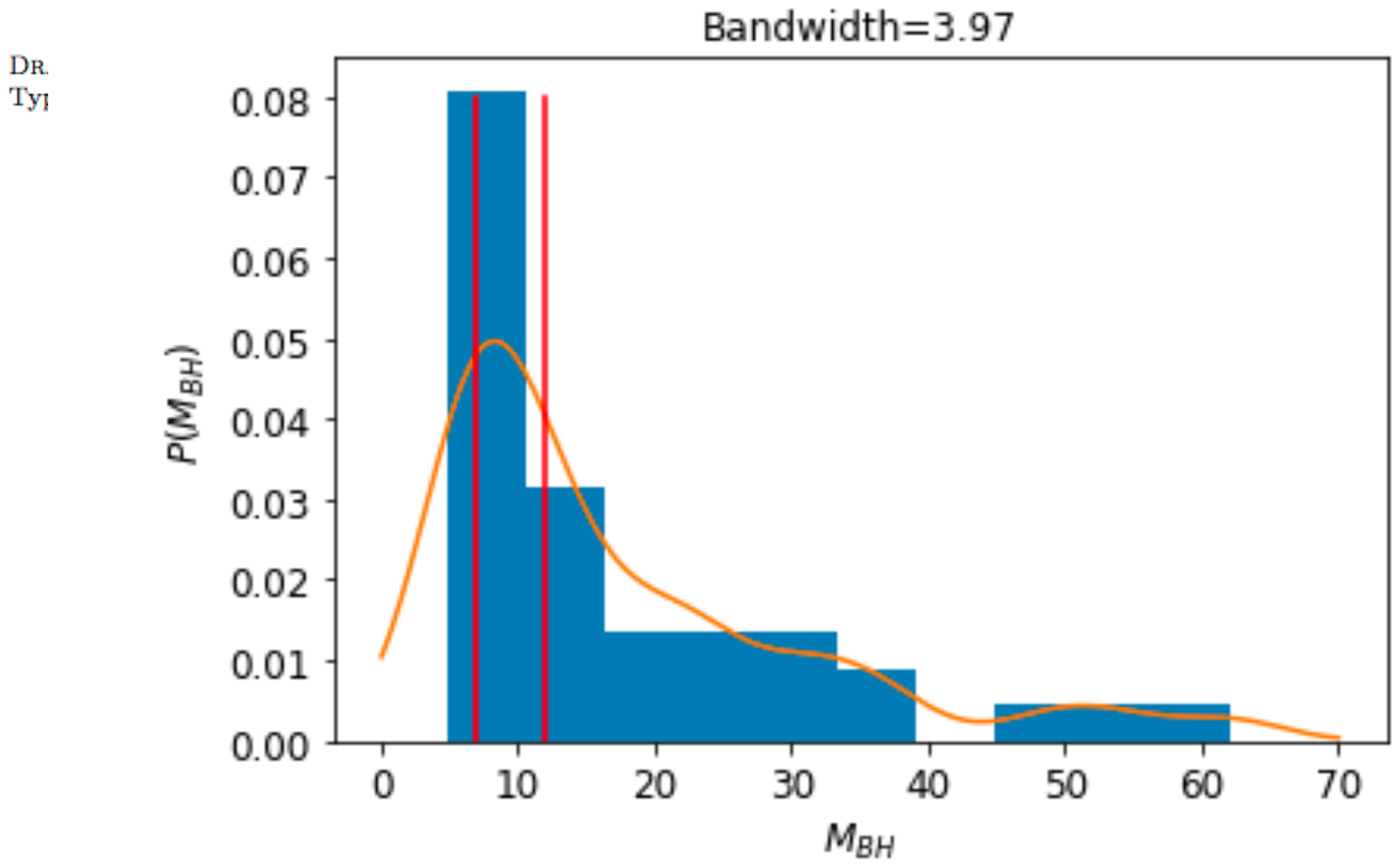
```
for i, bw in enumerate(bws):
    kf = KFold(n_splits=n_splits)

    # Initiate - lnP will contain the log likelihood of the test sets
    lnP = 0.0

    # Loop over each fold
    for train,
        x_train
        x_test =
        kde = K                                                    ).fit(x_train)

        log_pro
```



First: Problem set 3

1. Determine an optimal bandwidth for black hole masses

Today:



Question 2

Use the pulsar mass catalogue to estimate the likelihood of finding a neutron star with $M>1.8\ M_{sun}$

Key point: a KDE can be seen to reconstruct a probability distribution function, $p(x)$, and then we can calculate:

$$\Pr(M > m) = \int_m^\infty p(x)dx$$

Question 2

Simulate the next 5 mergers:

```
m1 = kde.sample(5)
m2 = kde.sample(5)
```

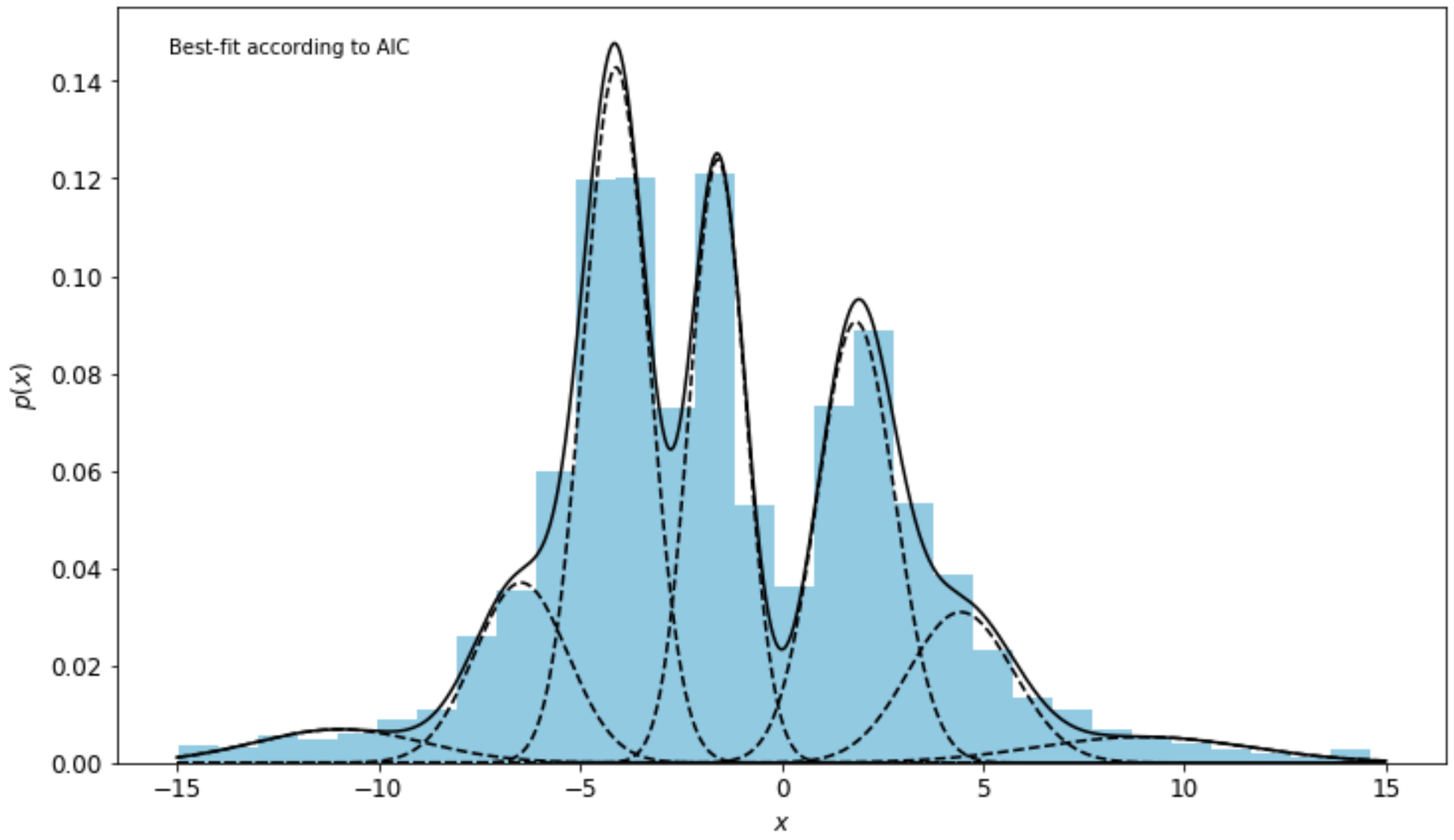
Assumptions made:

- I have assumed that the neutron star masses in the file are representative of the underlying population of neutron stars.
- I have assumed that the binary neutron stars have the same mass distribution as single neutron stars.
- I have assumed uncertainties on the masses are negligible (not a terribly good assumption).

How many peaks?

Main point: Realise that you need a density estimation technique to estimate the distribution and that you need to use some model selection technique to determine the number of components.

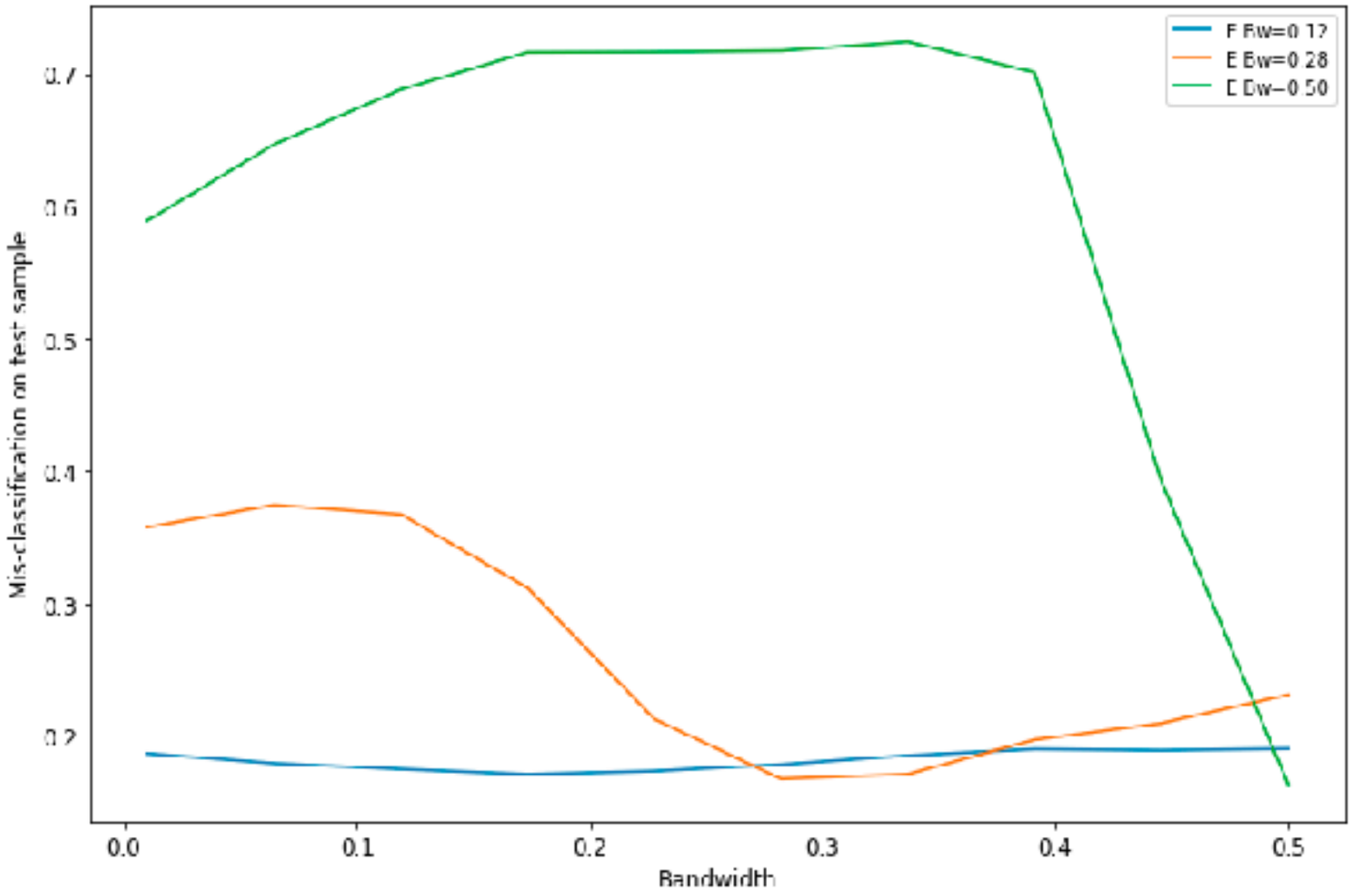
I used Gaussian Mixture Modelling + AIC and BIC



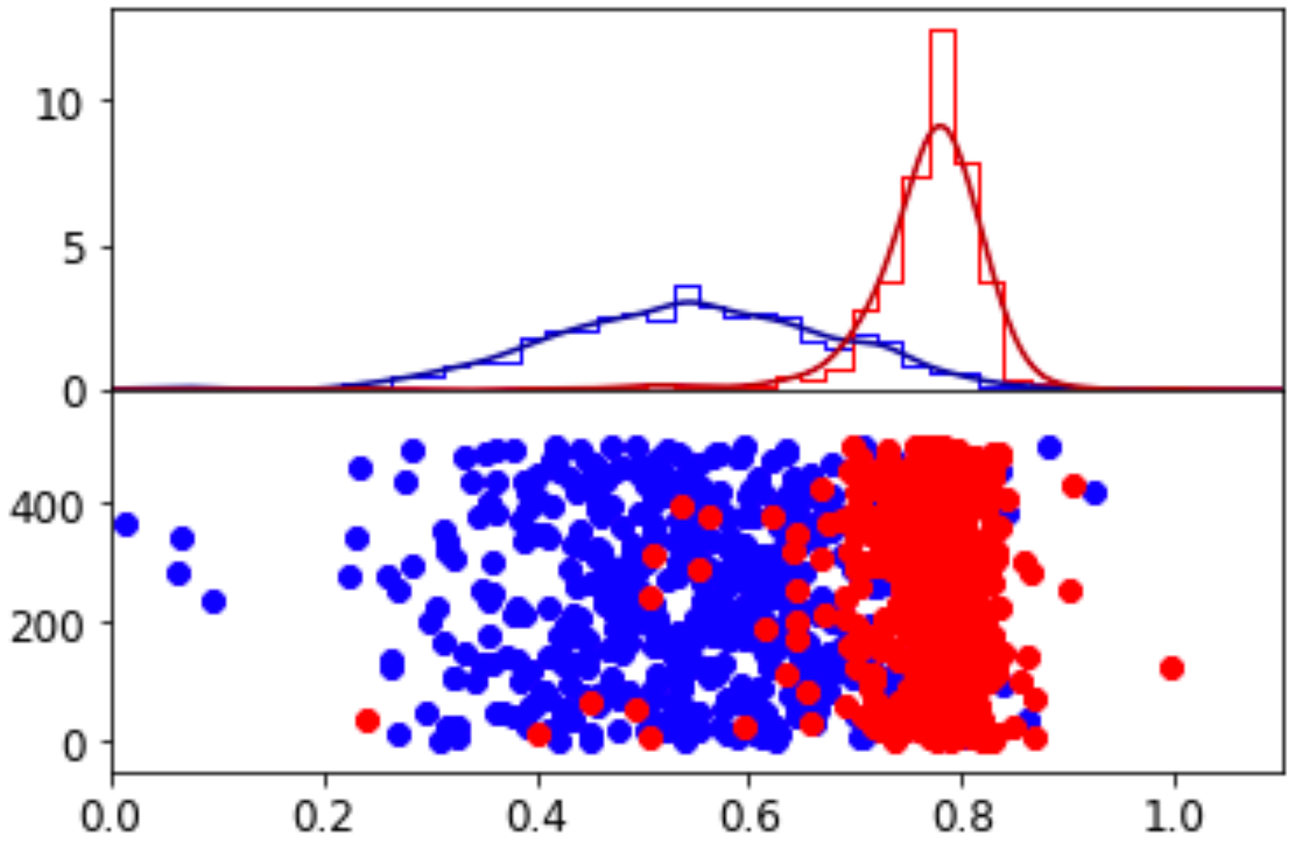
Galaxy classification

Covered in previous lecture, but one point worth noting:

You need to determine the bandwidth of the kernel density estimator. We do this by minimising mis-classification.



Why the lack of sensitivity?



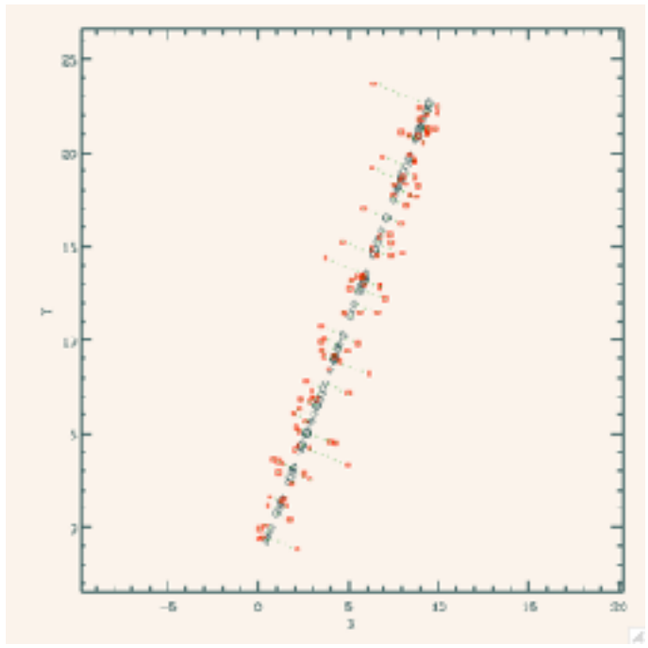
The main problem is the misclassified objects - they impose an upper limit to the accuracy of the classification.

Recap

We looked at Principal Component Analysis (PCA) as a tool to find an optimal coordinate system, to select important features, and as a dimension reduction technique.

The steps were:

1. Calculate the covariance matrix of the data.
2. Calculate the eigenvalues & eigenvectors of the matrix.
3. Based on the eigenvalues, select the important components (or all if you just want a new coordinate system).
4. Place the eigenvectors you want to keep in a feature matrix.
5. Project back using this matrix to get reconstructed data.



Feature extraction & PCA

In general, if you have a lot of data, and a lot of parameters in your model, it might be good to carry out a PCA decomposition first to identify the important features.

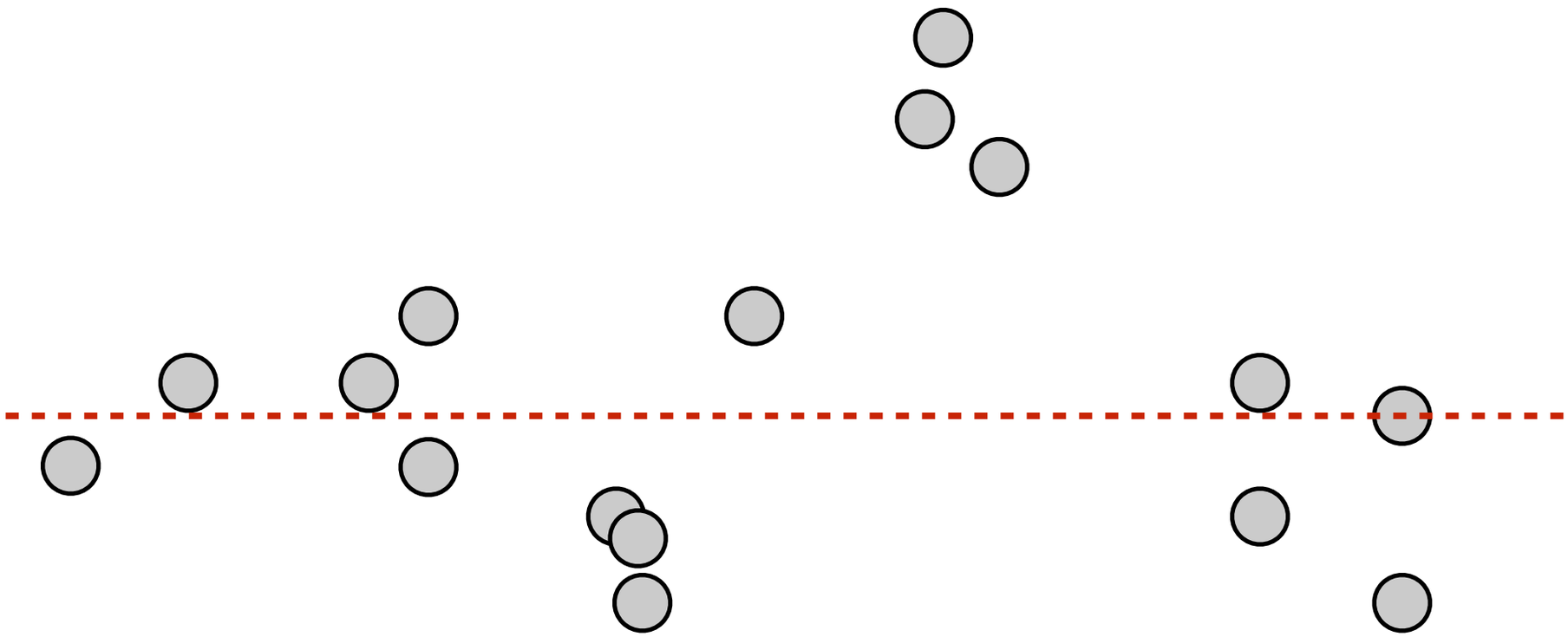
Examples:

- Velocity dispersion estimates for galaxies.
- Estimation of physical parameters of galaxies.
- ....
- Hand-writing recognition.
- Voice-recognition etc etc.

Regression - keeping track of local properties

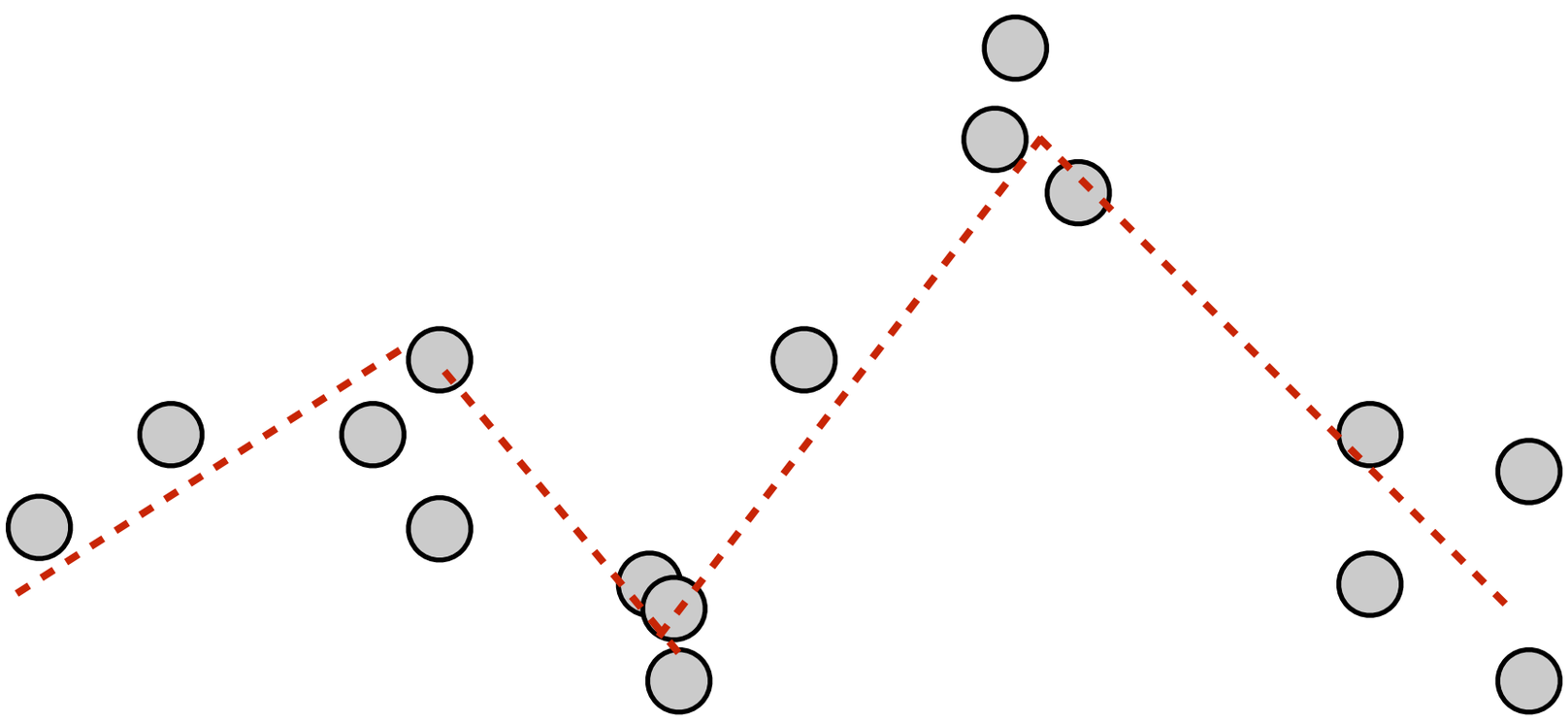
The broad idea

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:

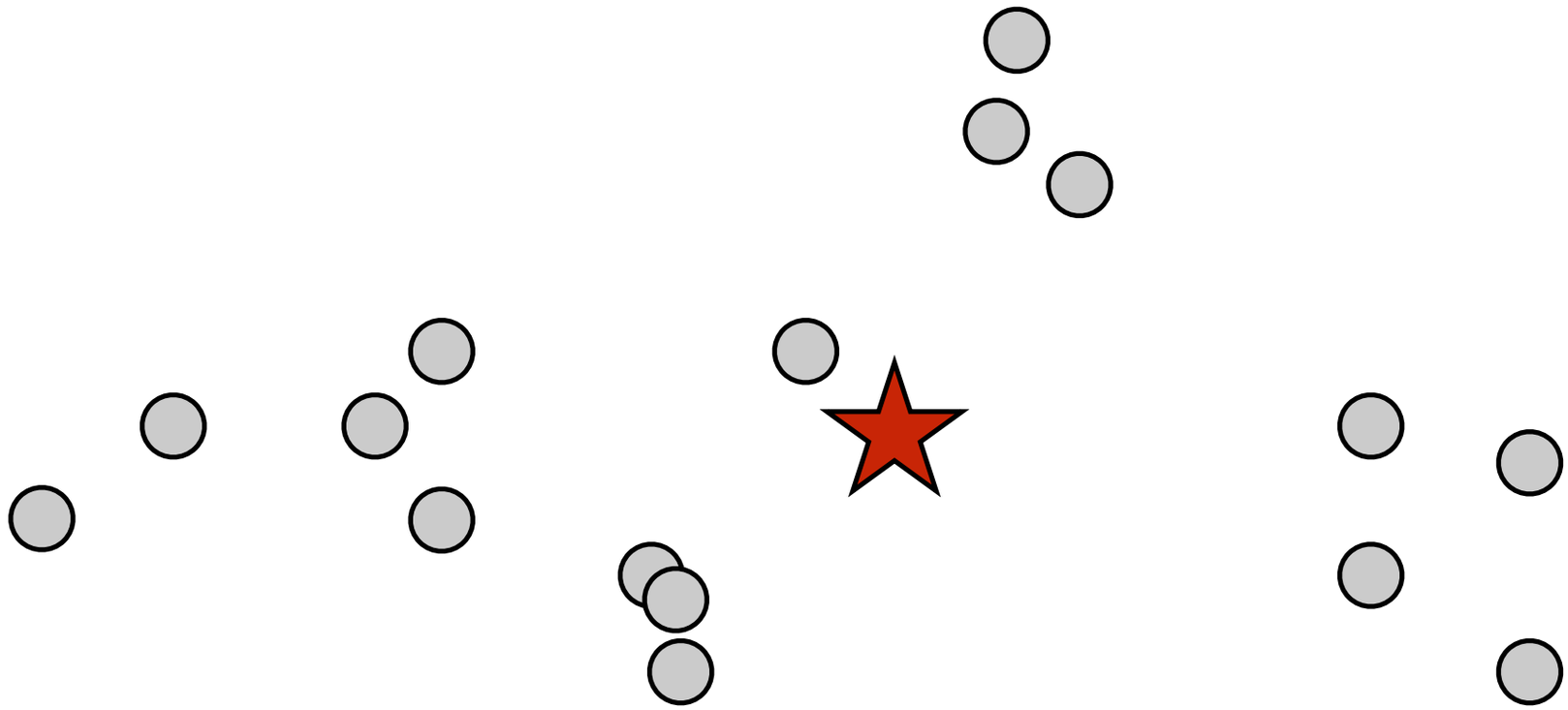


The main techniques

- 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

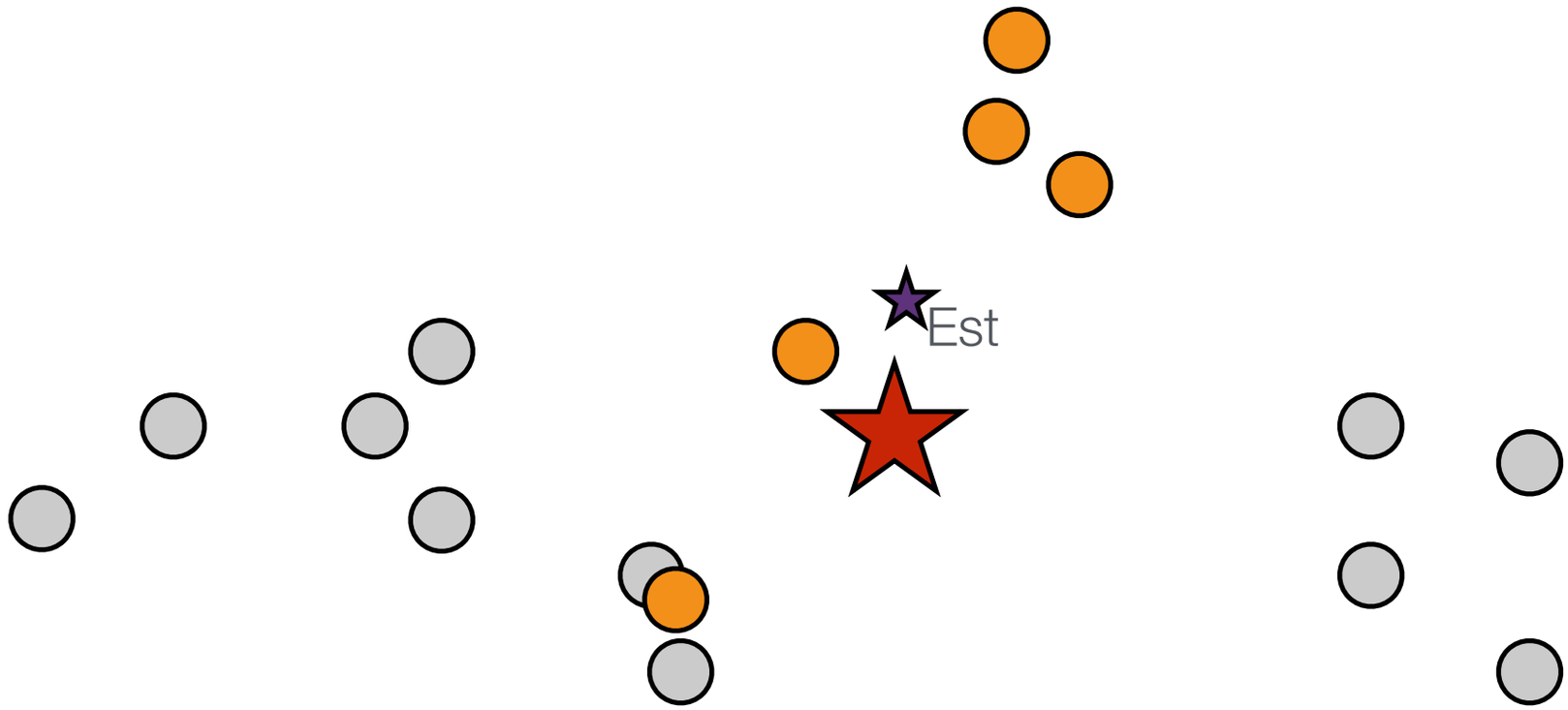
Nearest neighbour regression

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



Nearest neighbour regression

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

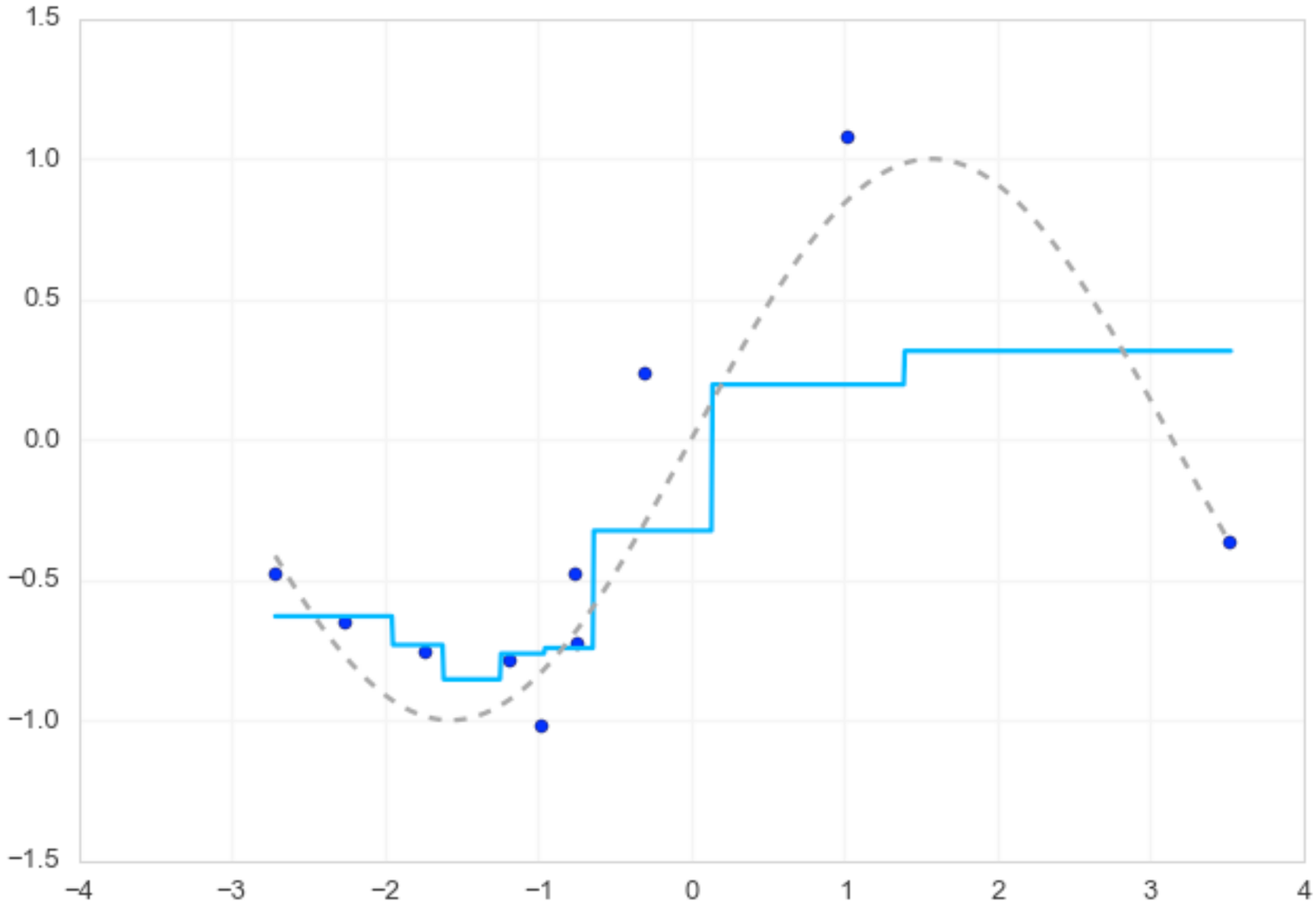


Nearest neighbour regression

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

```
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 $p(x)$

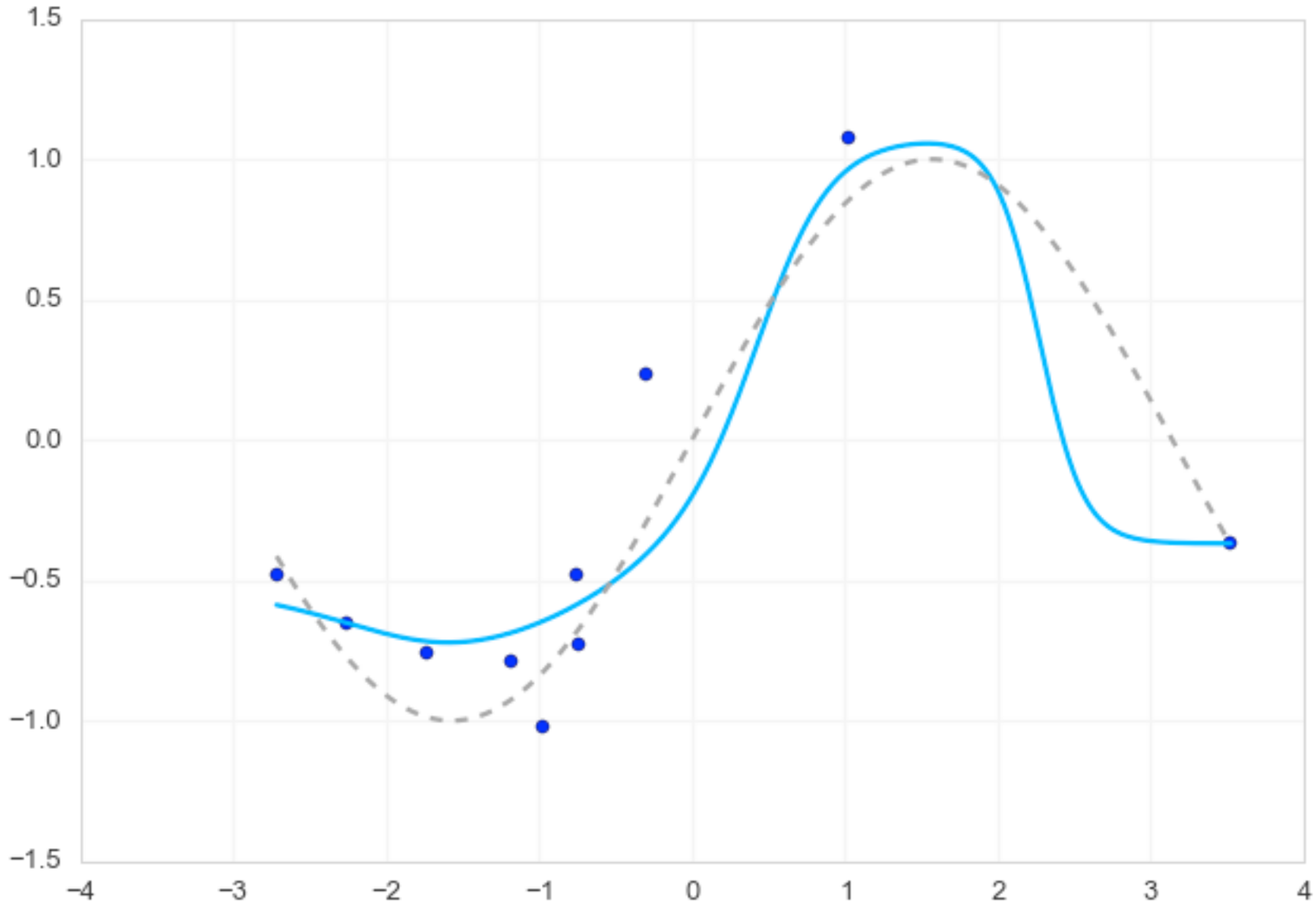
Kernel regression

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:

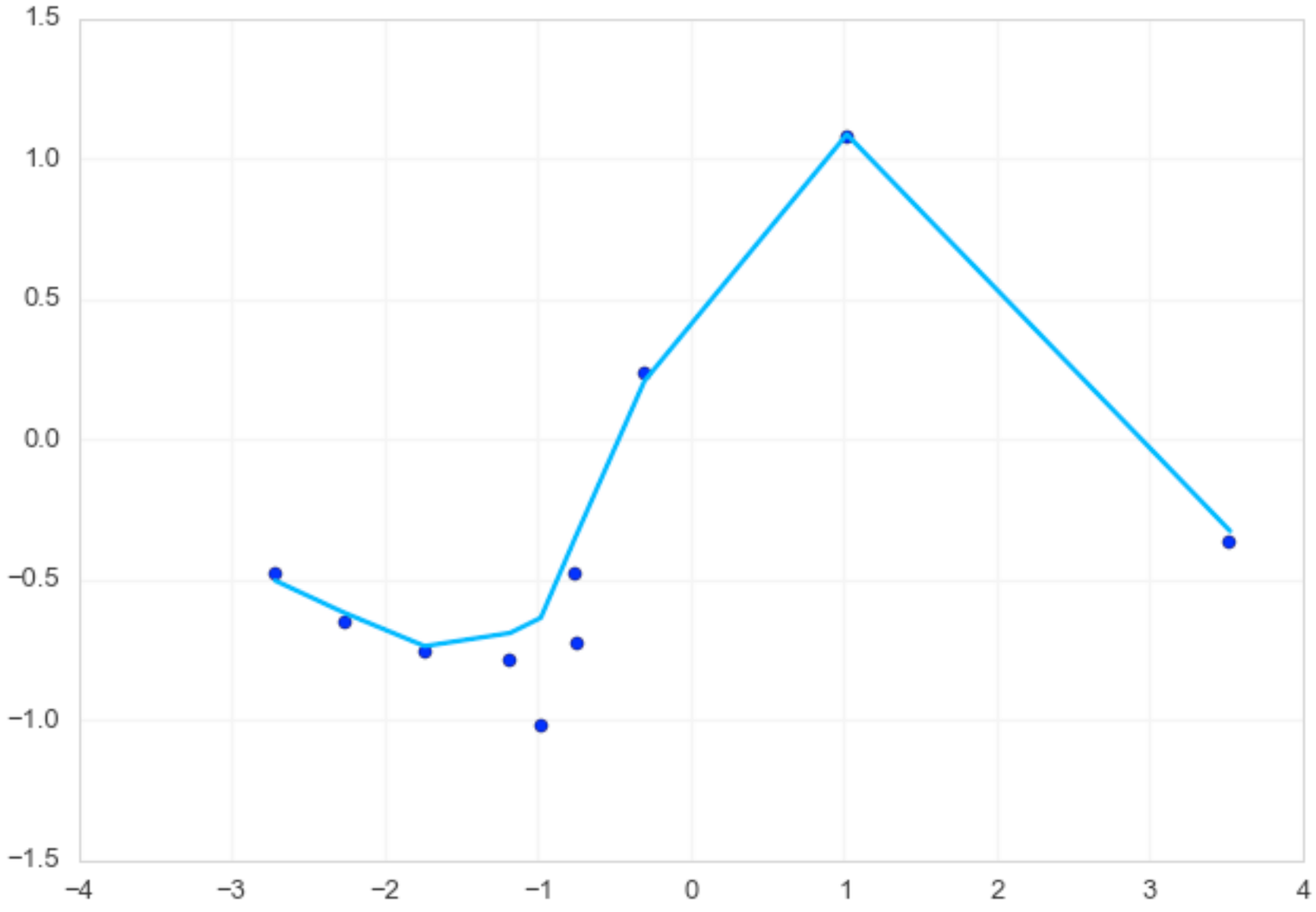
`statsmodels.nonparametric.smoothers_lowess.lowess`
`cylowess`

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

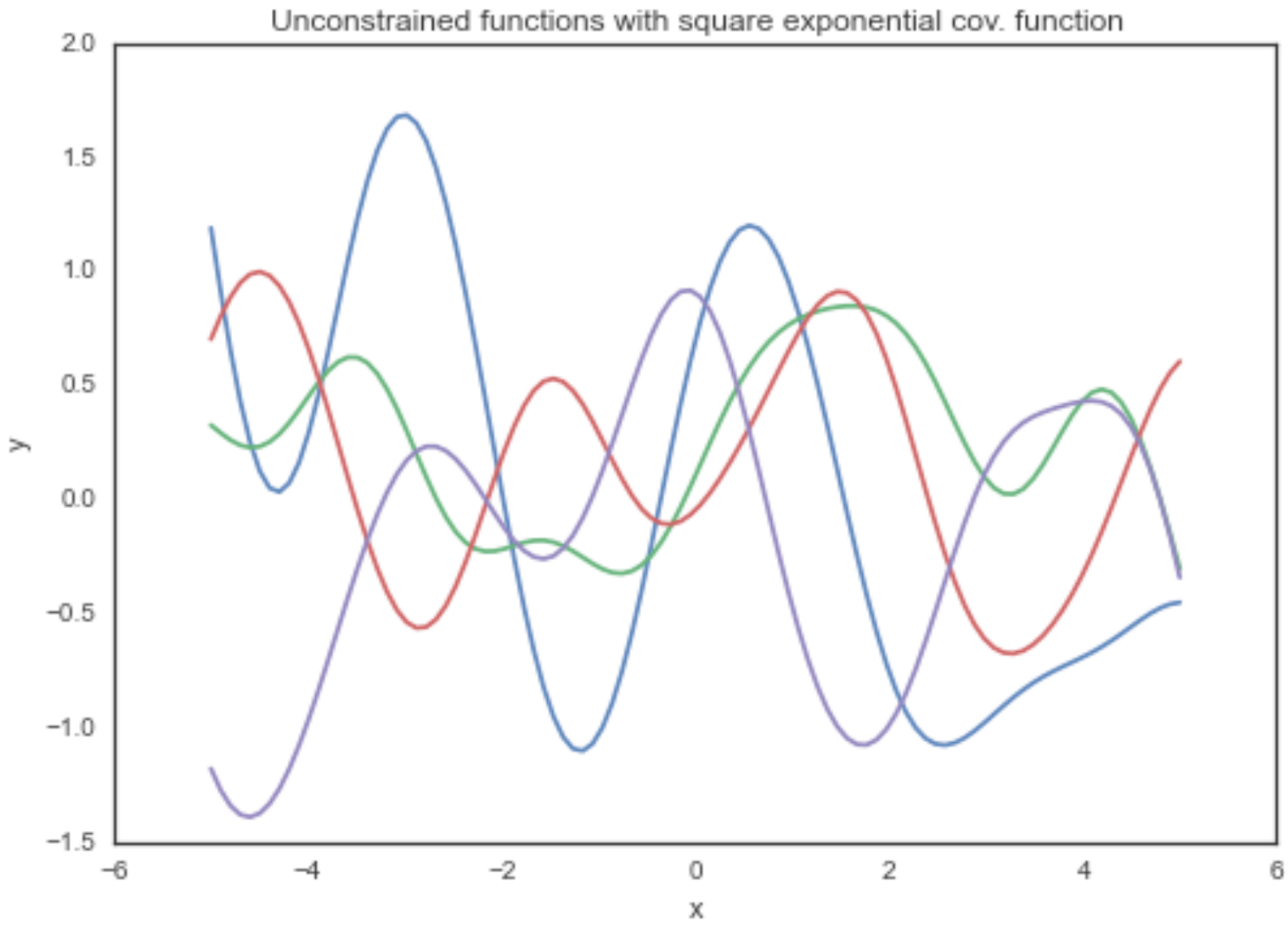
In this case we apply a prior in function space - this prior is specified using a mean & covariance function (since that is all we need for a Gaussian). The most common is:

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

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

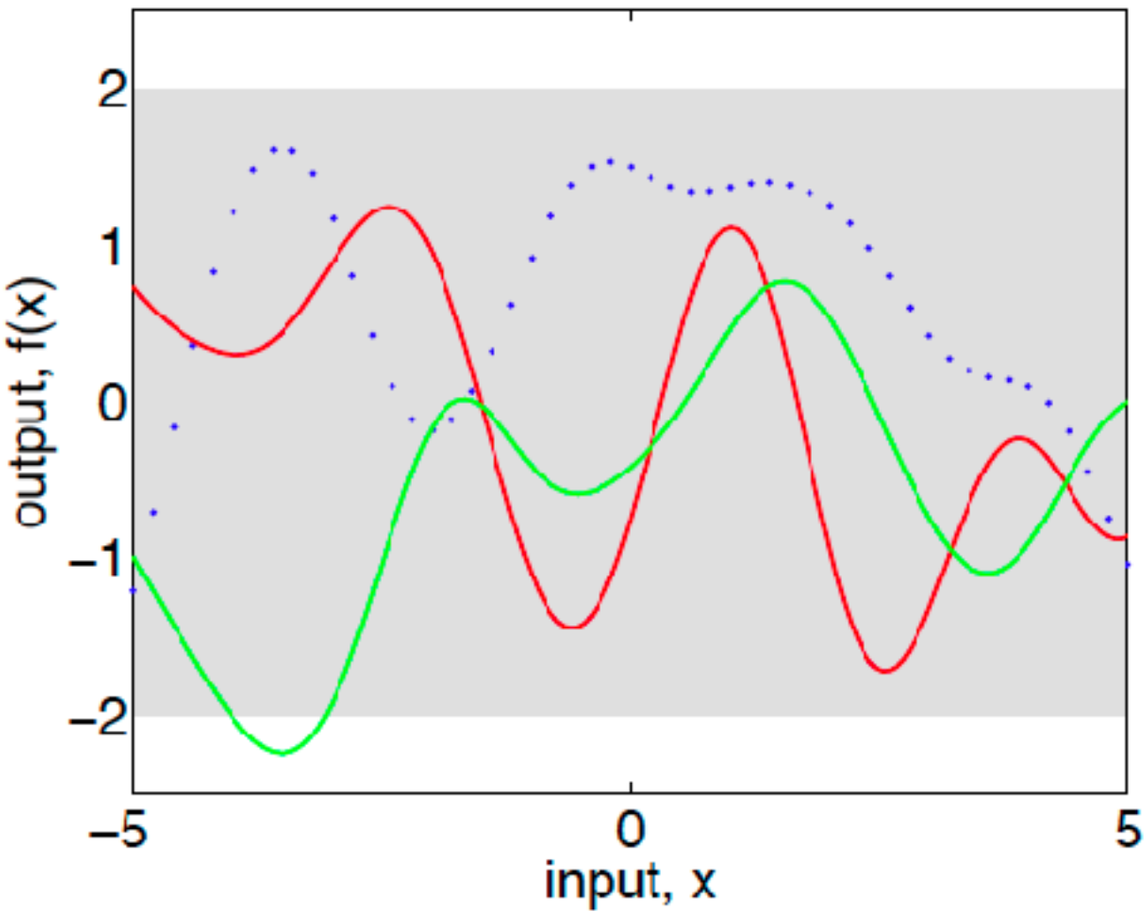


See the IPython notebook on the Github site:

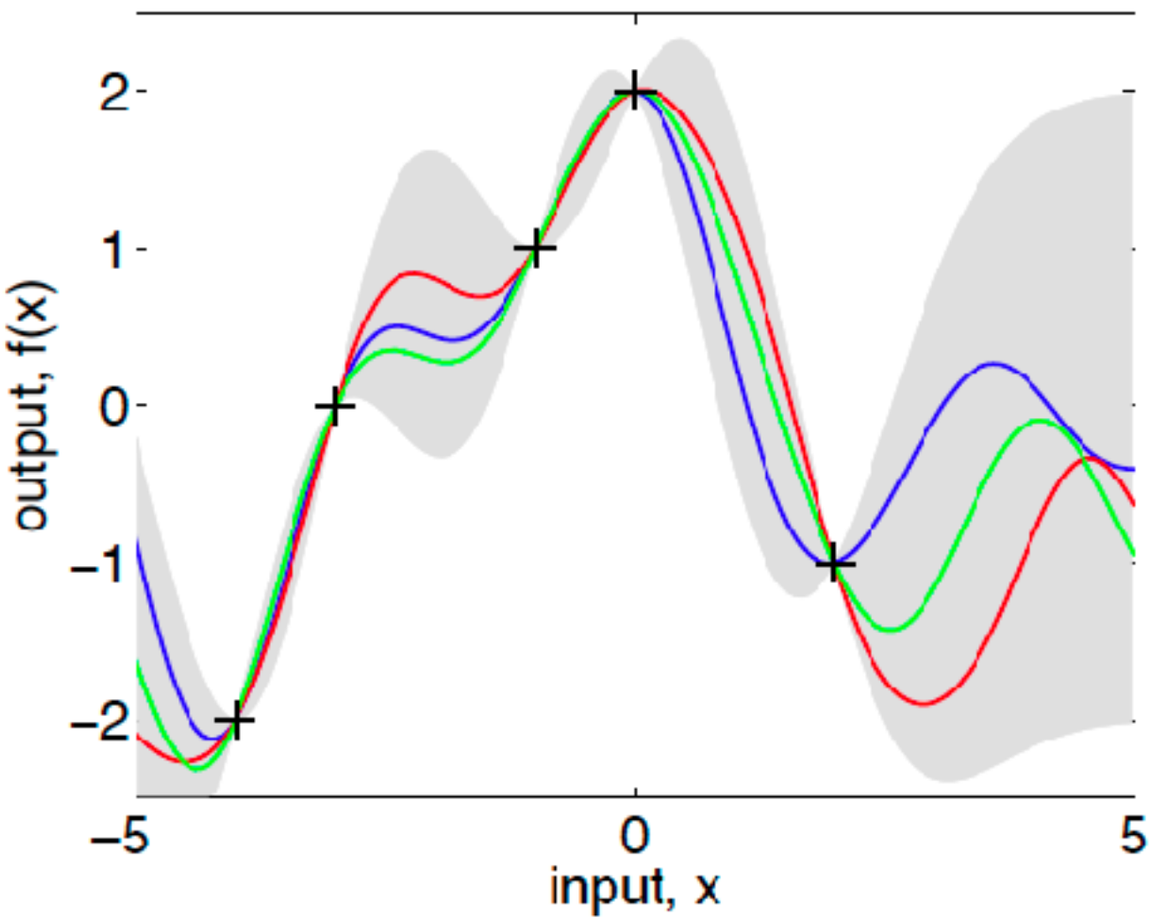
ForLectures/Lecture5/Gaussian process regression.ipynb

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

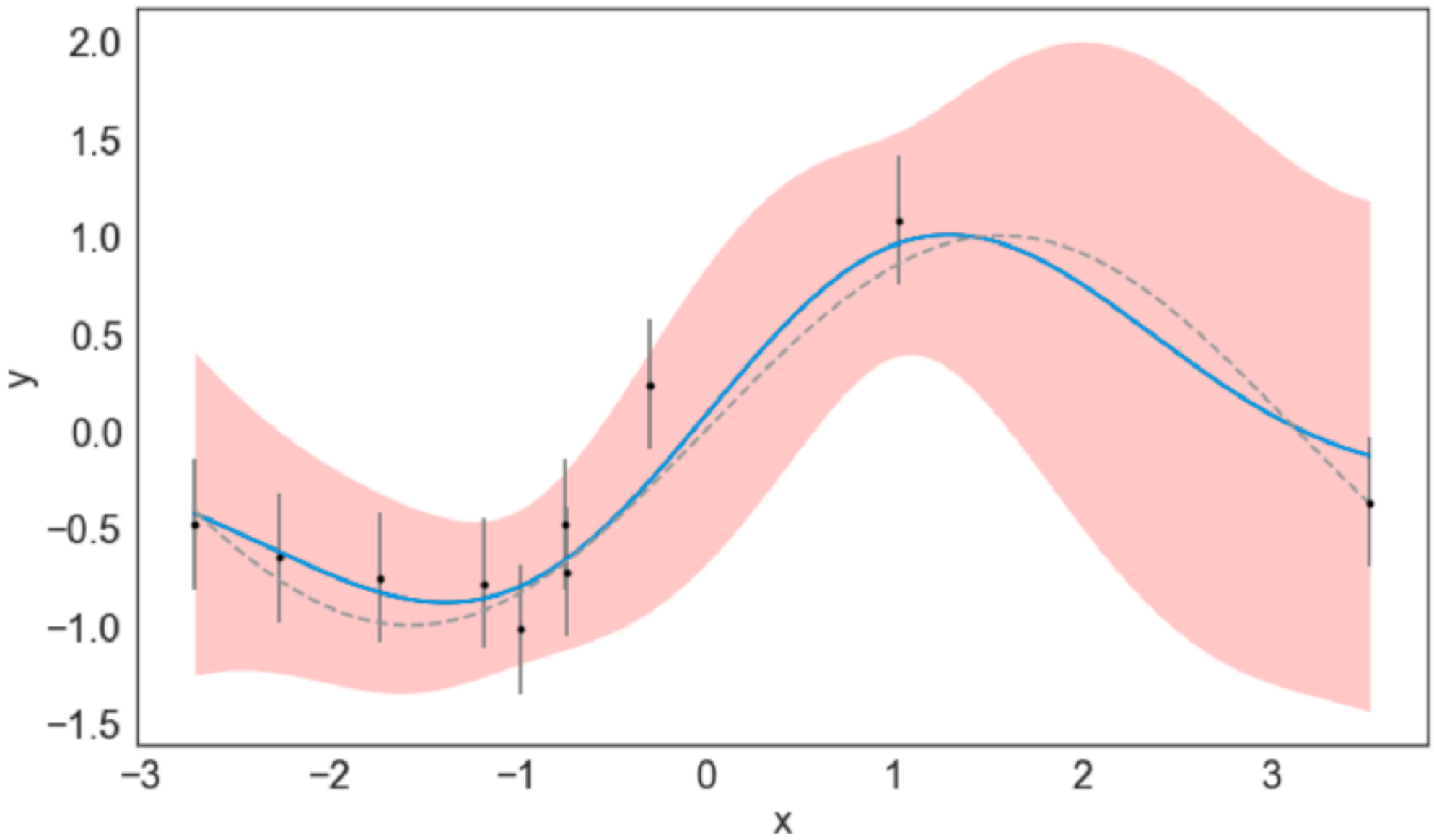
\- Fairly slow

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:

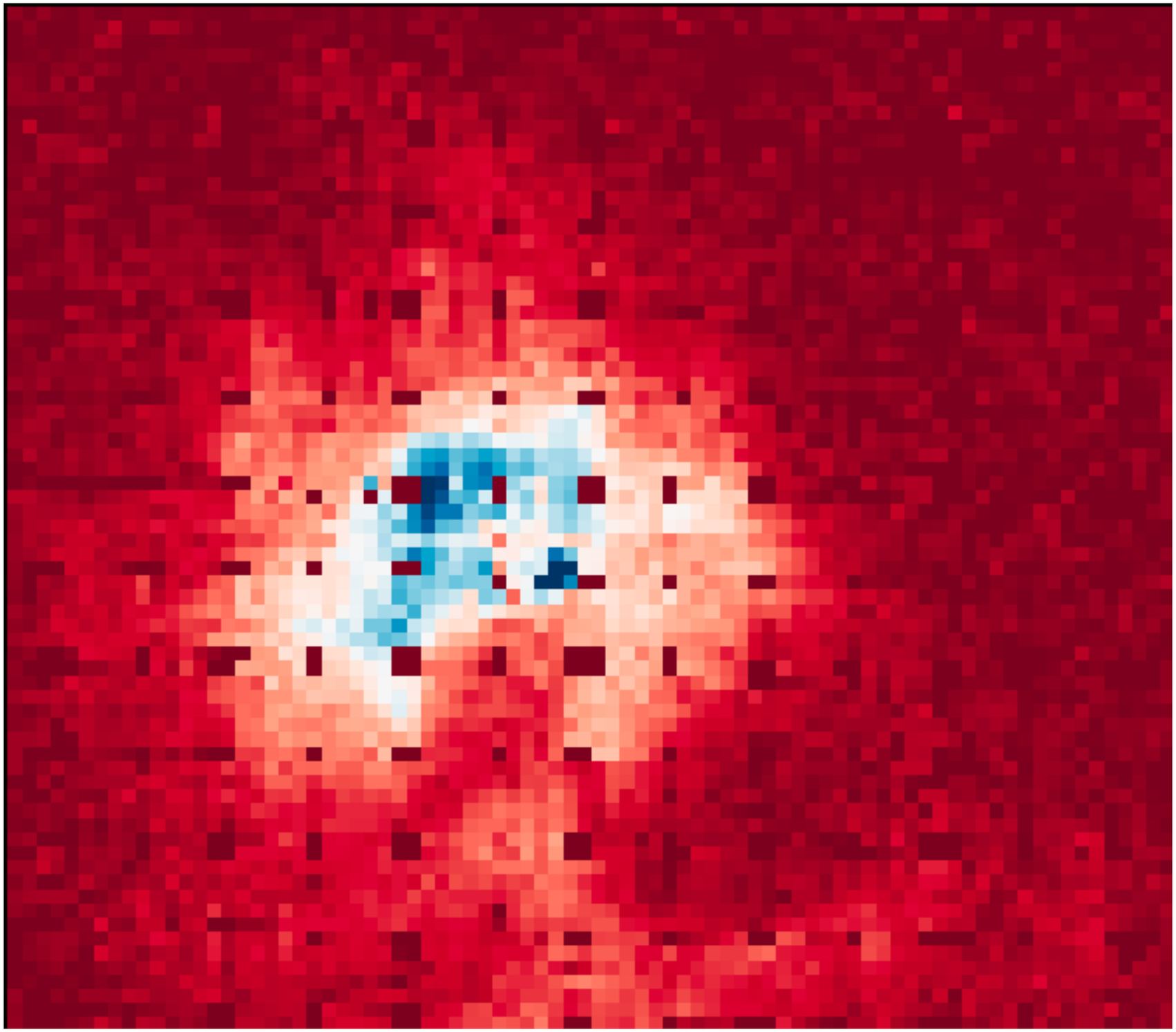
```
g = gp.fit(x[:, np.newaxis], y)
y_pred, sigma = gp.predict(xplot[:, np.newaxis], return_std=True)
plot_a_fit(x, y, dy, xplot, y_pred, sigma, include_true=True)
```



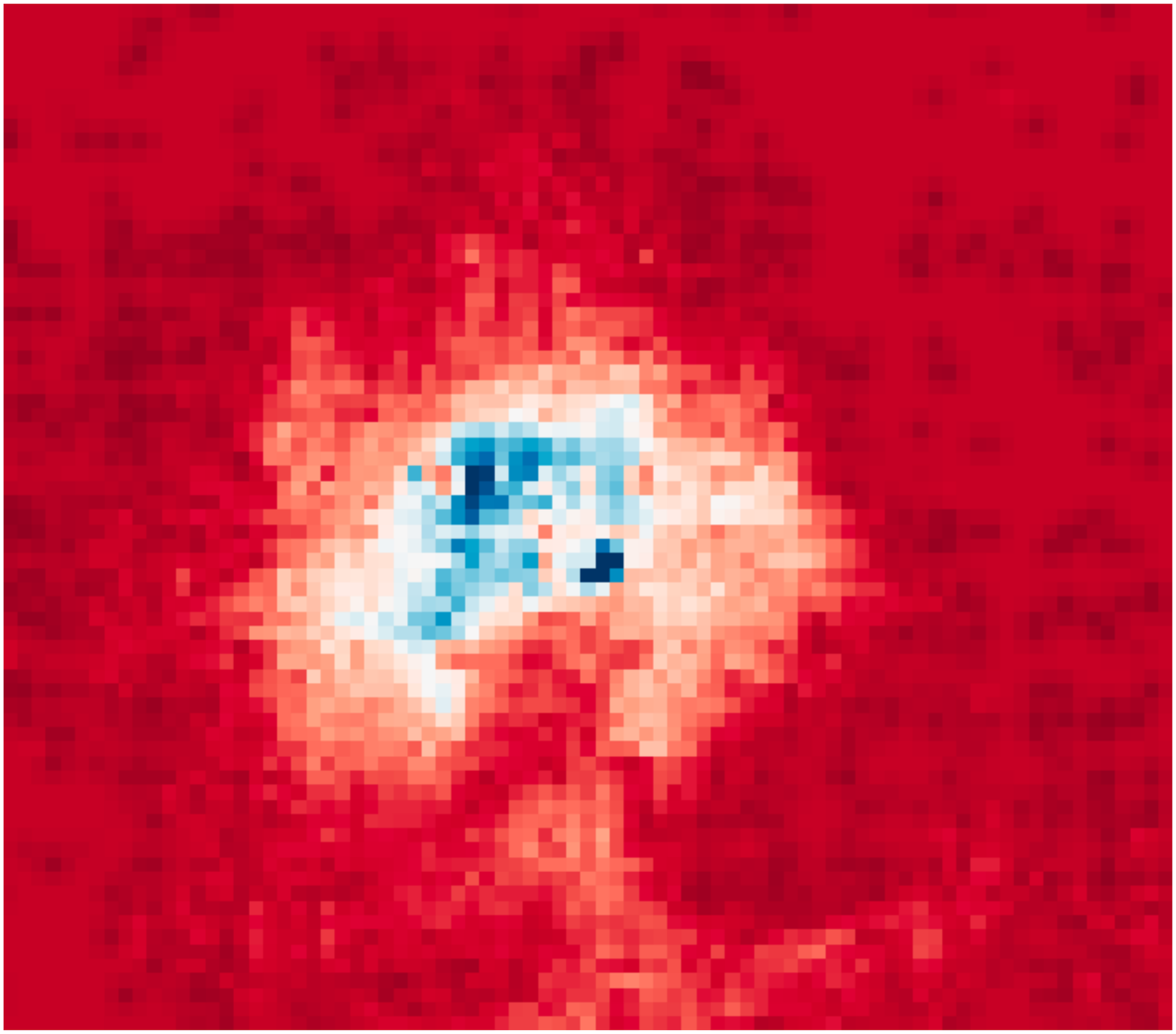
See the IPython notebook on the Github site:

ForLectures/Lecture5/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:

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

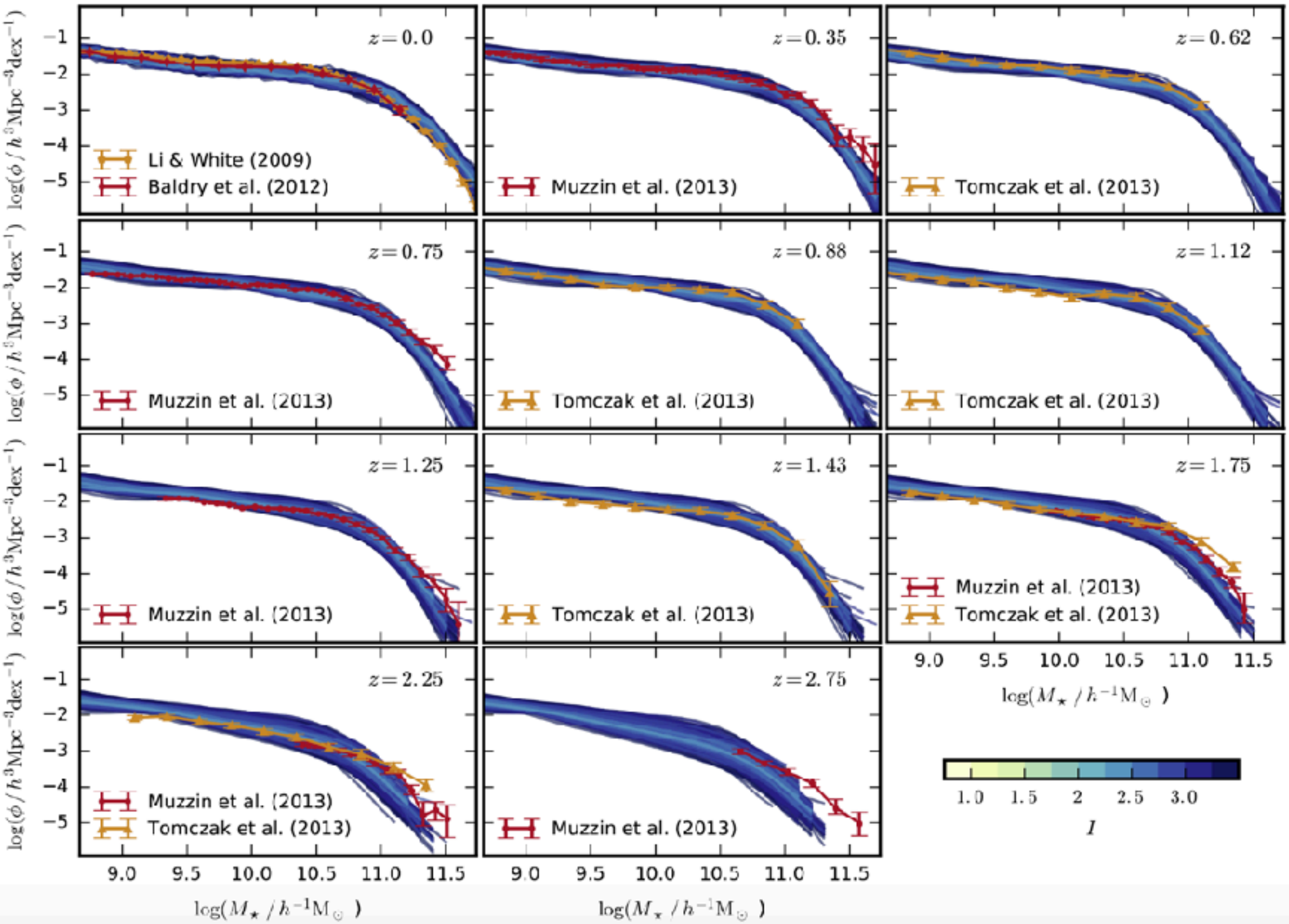
Then write this as a regression problem:

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

Polynomials

Gaussian Process

Fitting data to model - galaxy mass functions



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

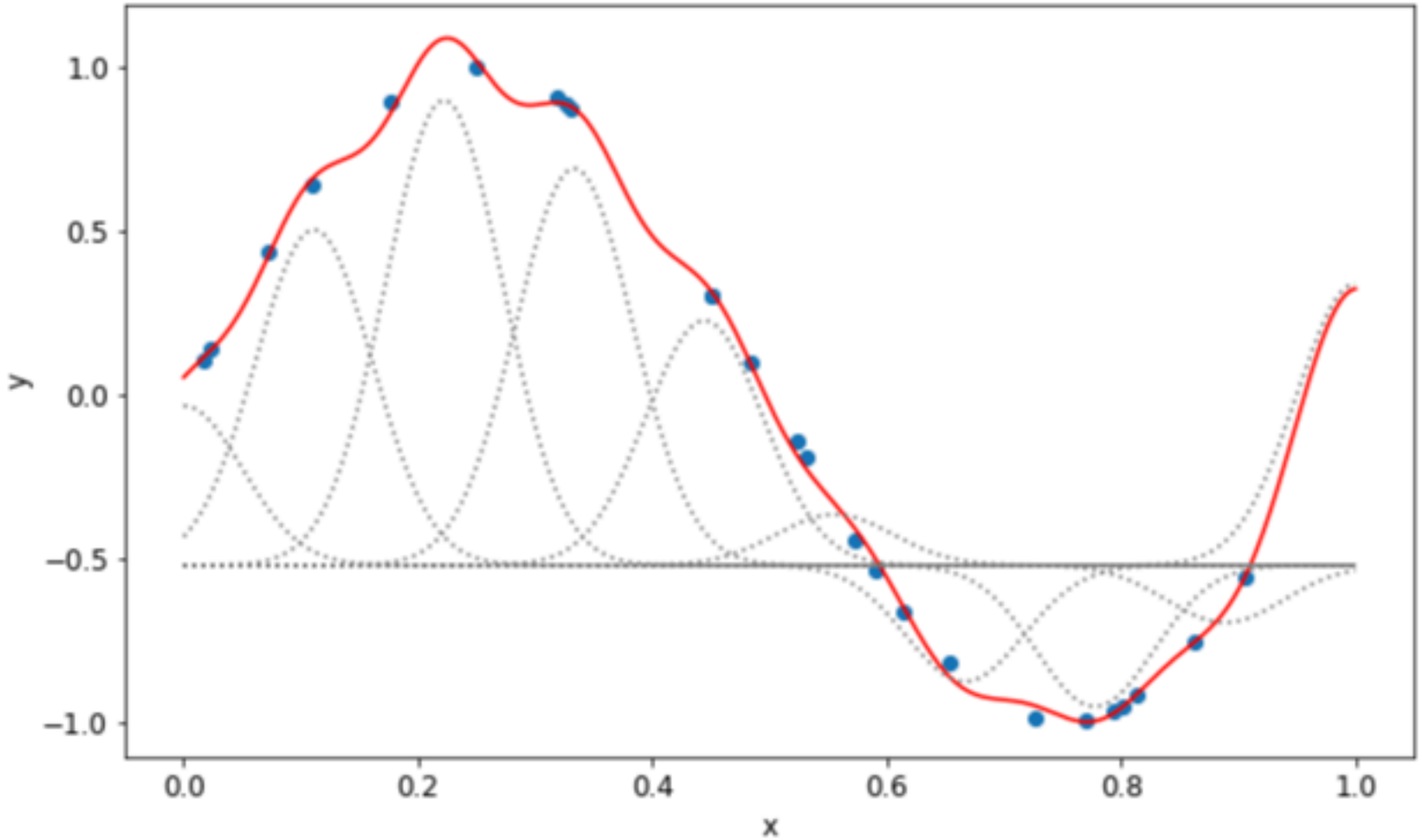
Basis function regression

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.

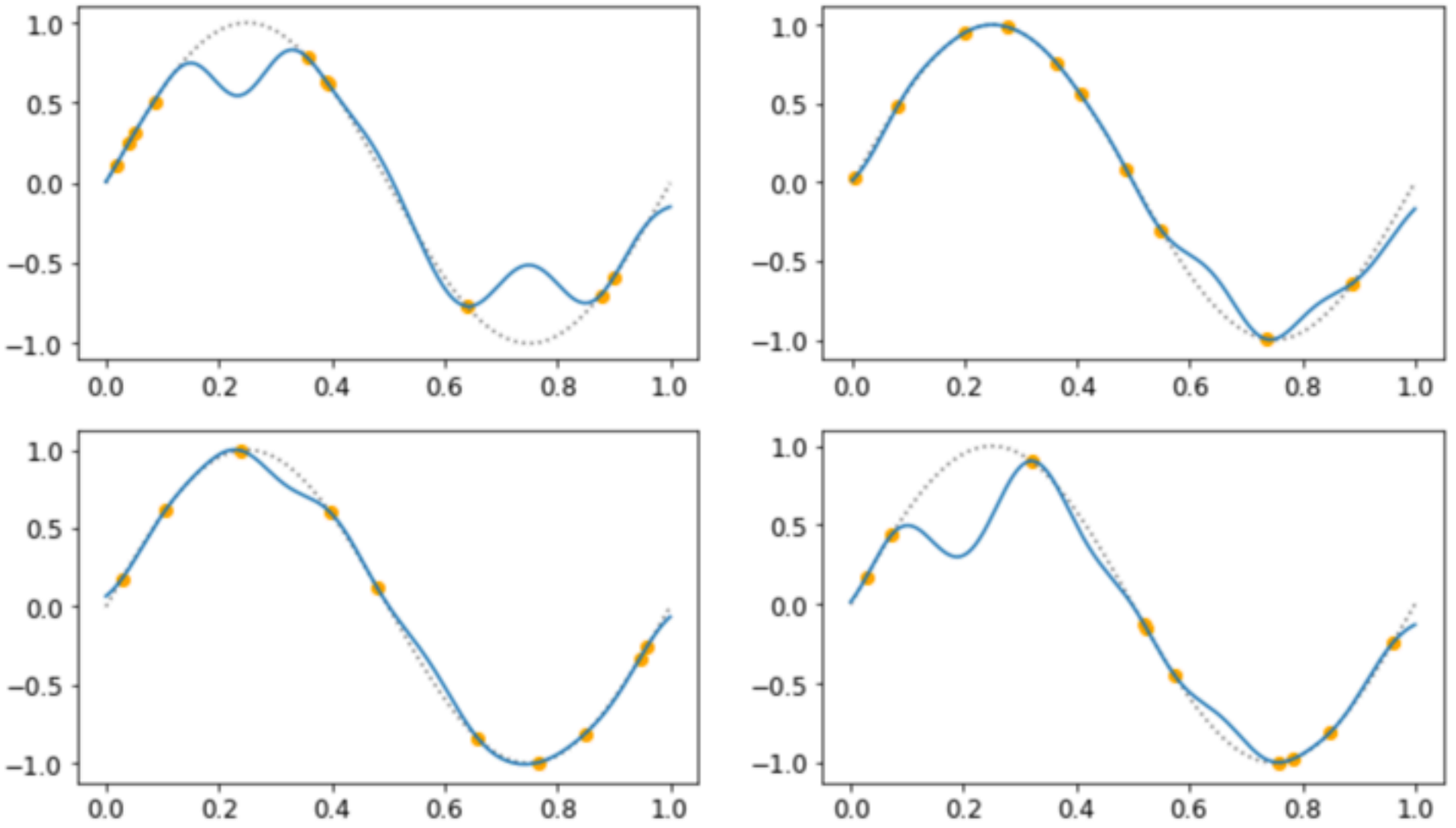
e.g. Gaussian basis:



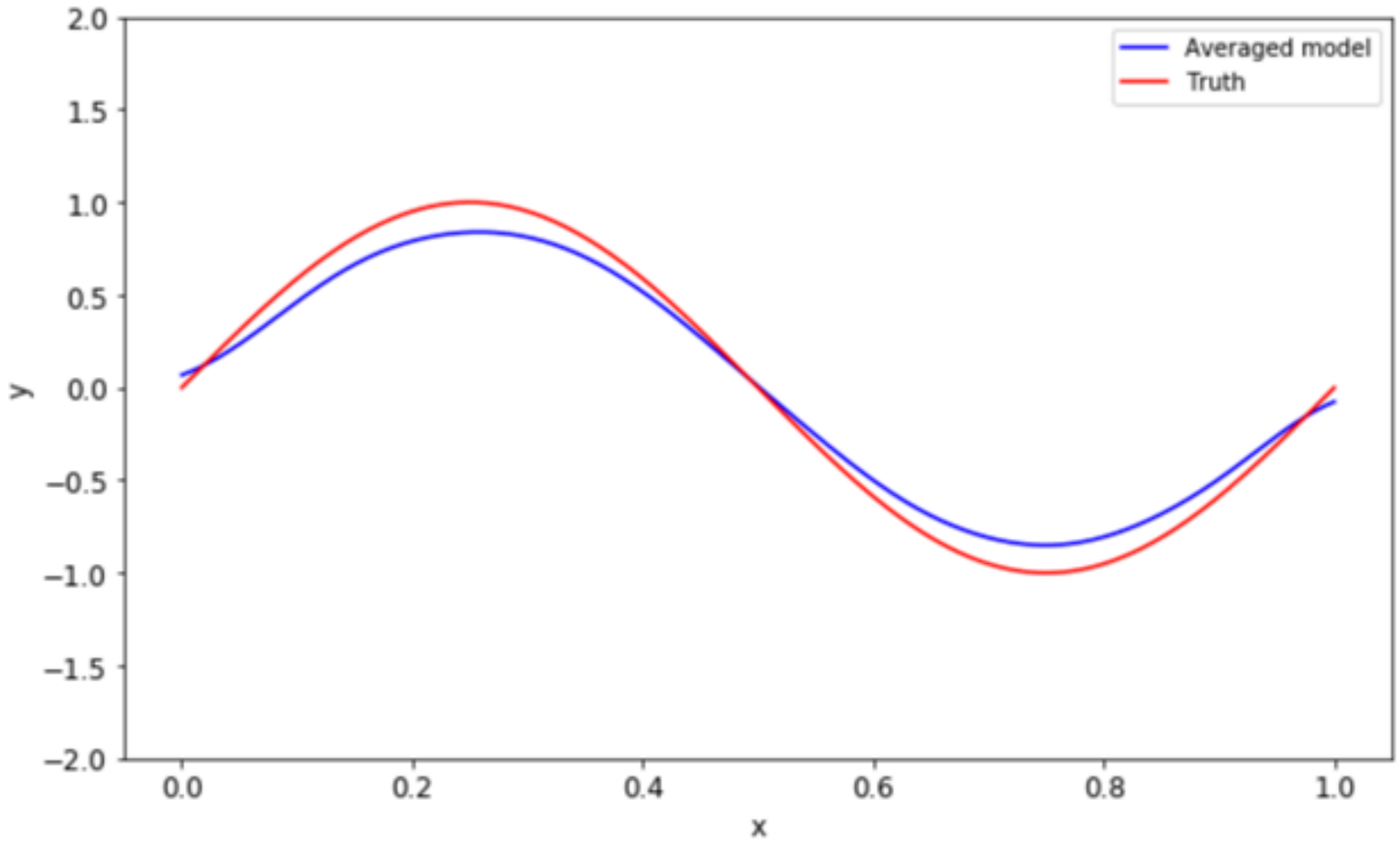
Gaussian-basis function regression

(see notebook under ForLectures/Lecture5 on GitHub)

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 $\frac{\sigma}{\sqrt{N_{\text{models}}}}$

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.

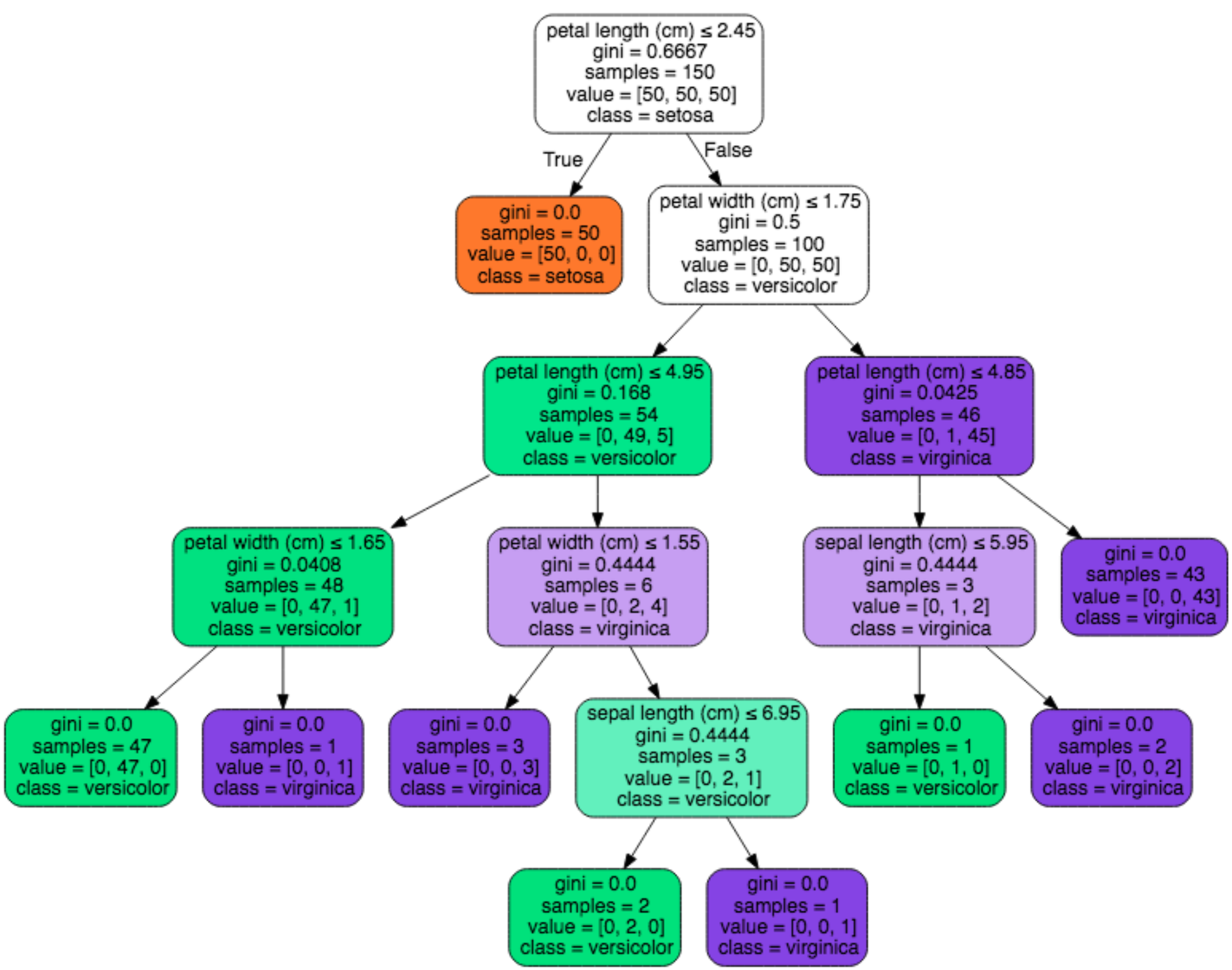
By making the tree deeper you can catch finer details in the data.

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

In sklearn:

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

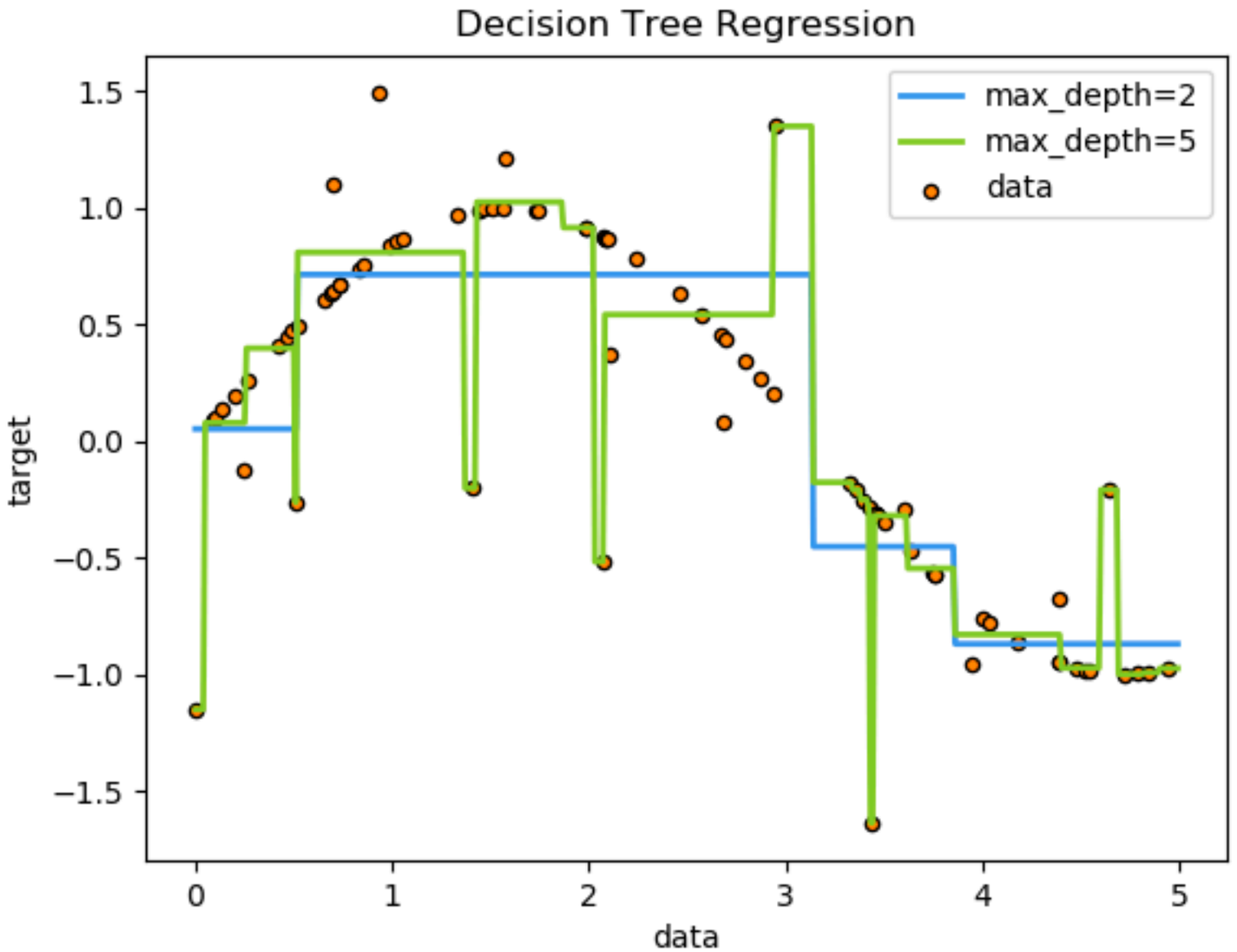
Decision trees - an intermezzo



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

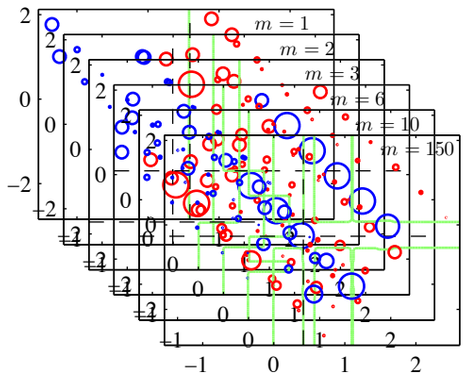
These are techniques that combine many very simple algorithms ("weak learners") to create a powerful final algorithm.

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

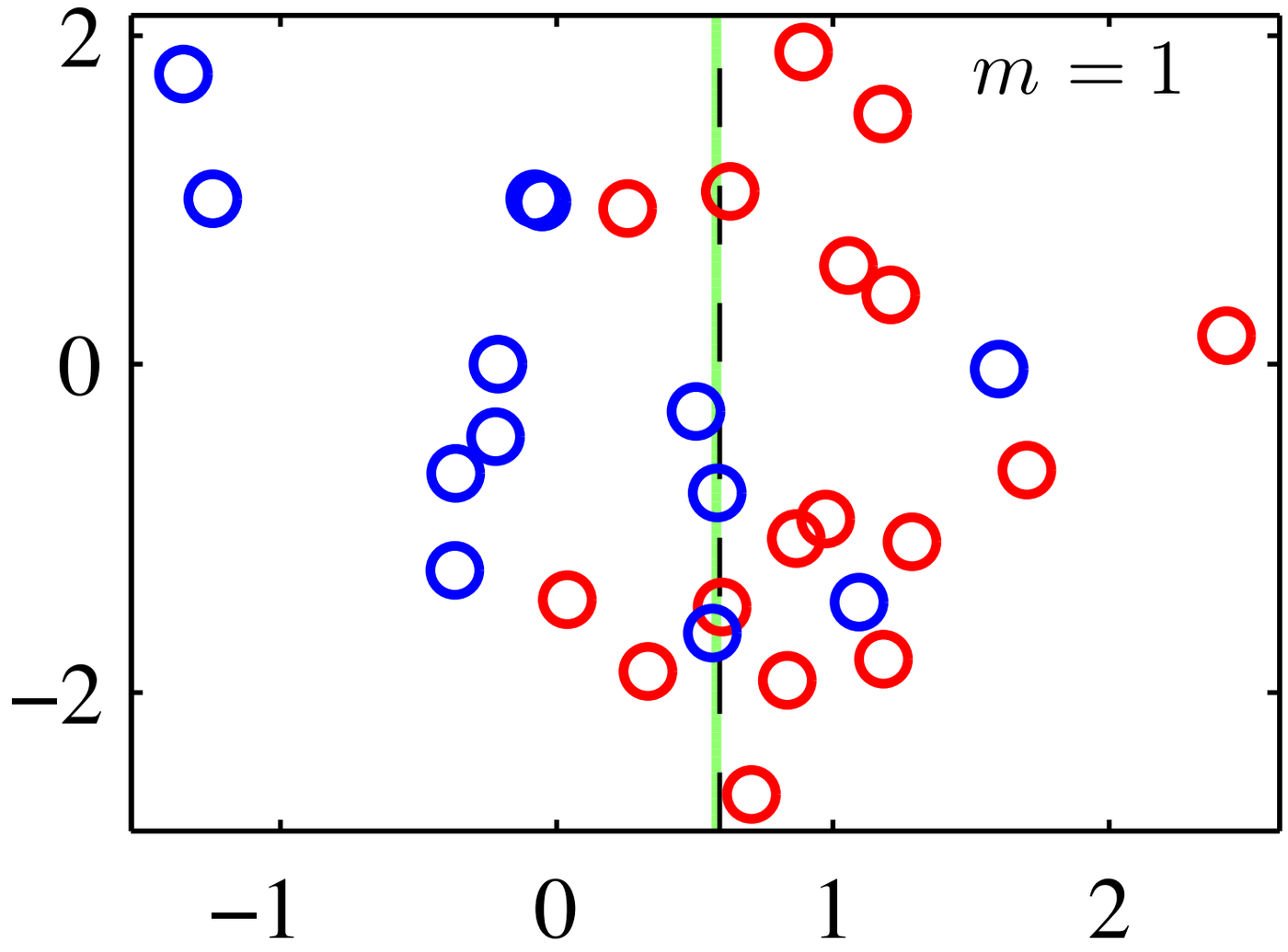
These are techniques that combine many very simple algorithms ("weak learners") to create a powerful final algorithm.



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

Boosting

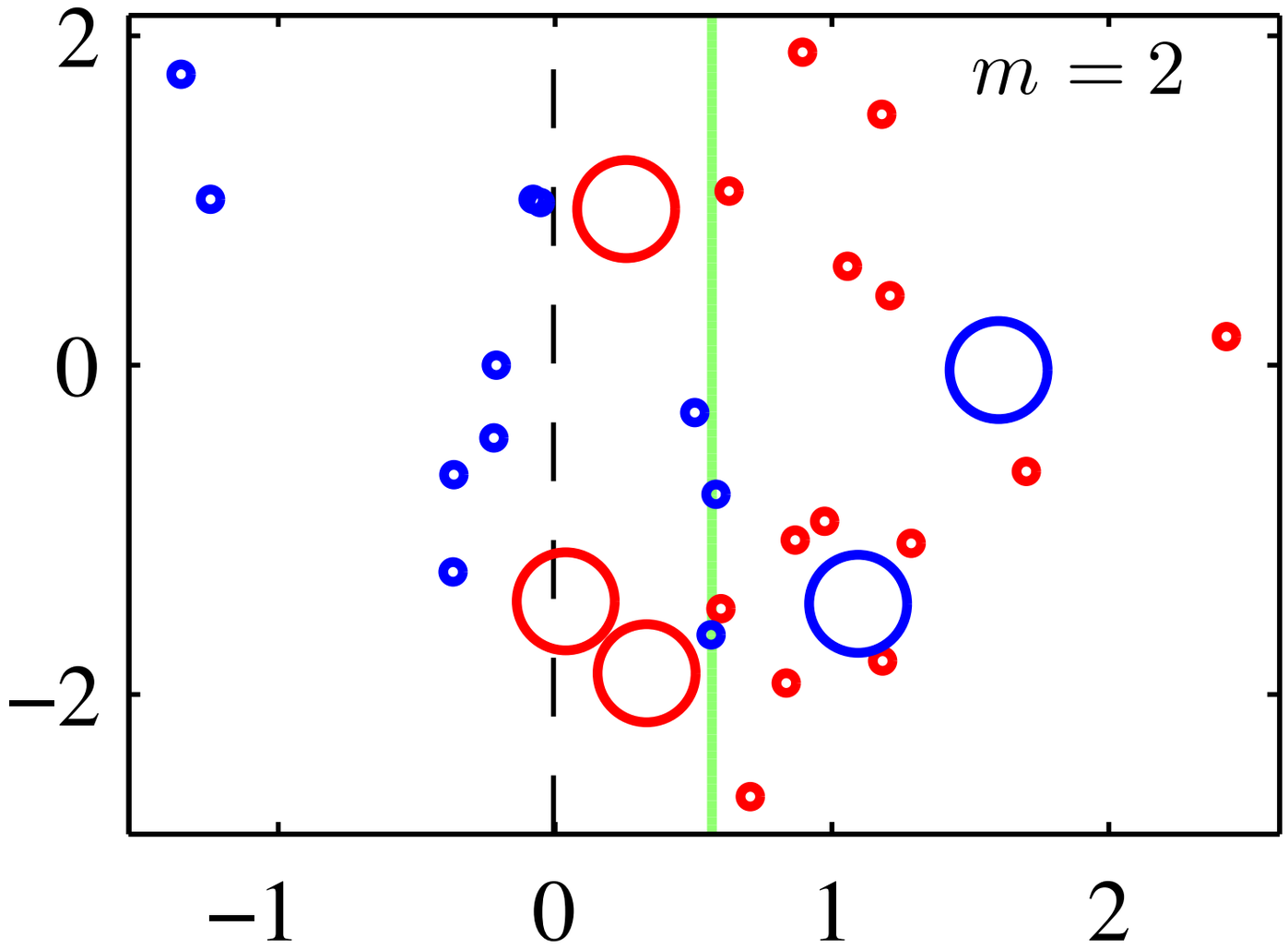
These are techniques that combine many very simple algorithms ("weak learners") to create a powerful final algorithm.



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

Boosting

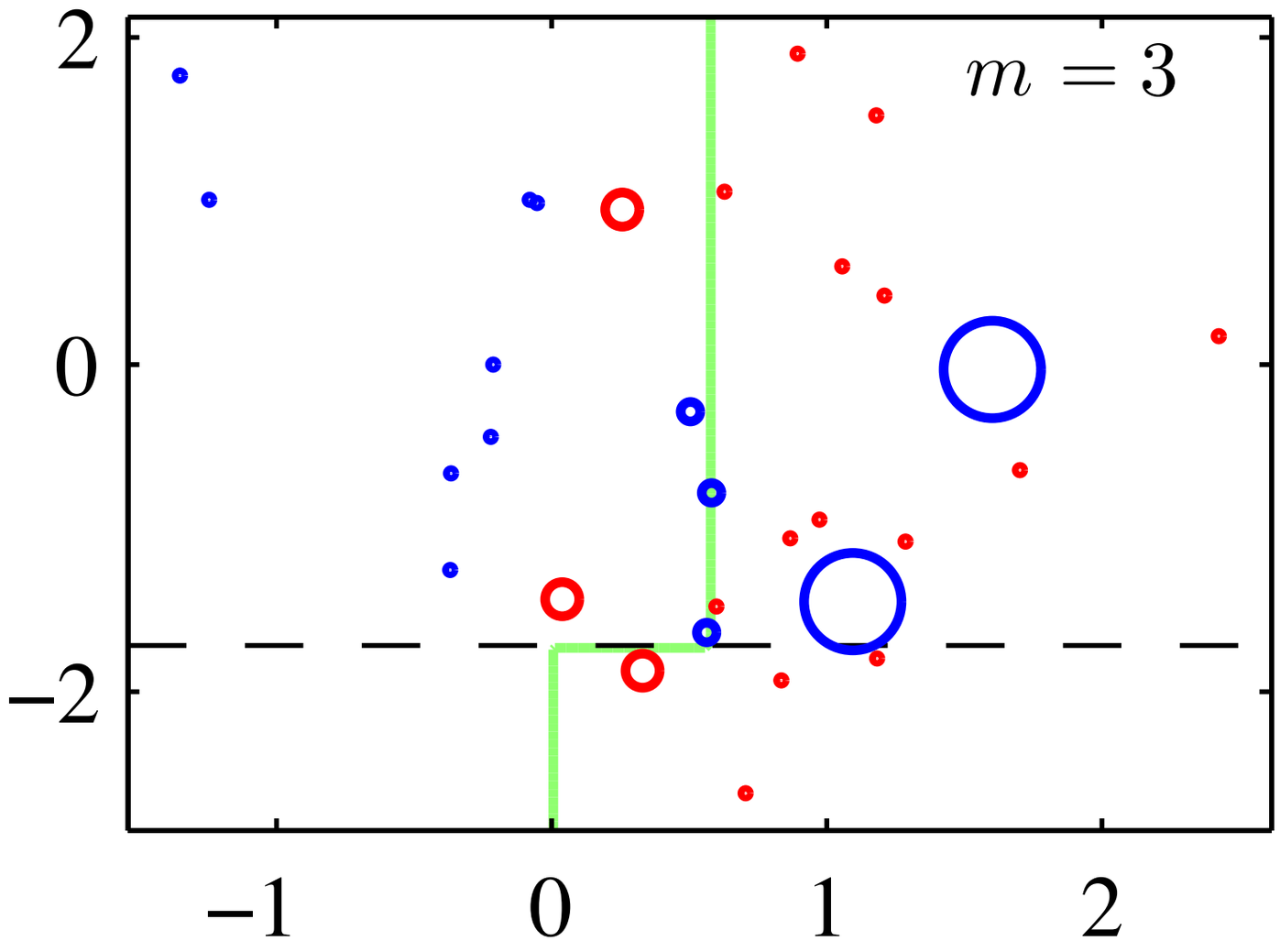
These are techniques that combine many very simple algorithms ("weak learners") to create a powerful final algorithm.



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

Boosting

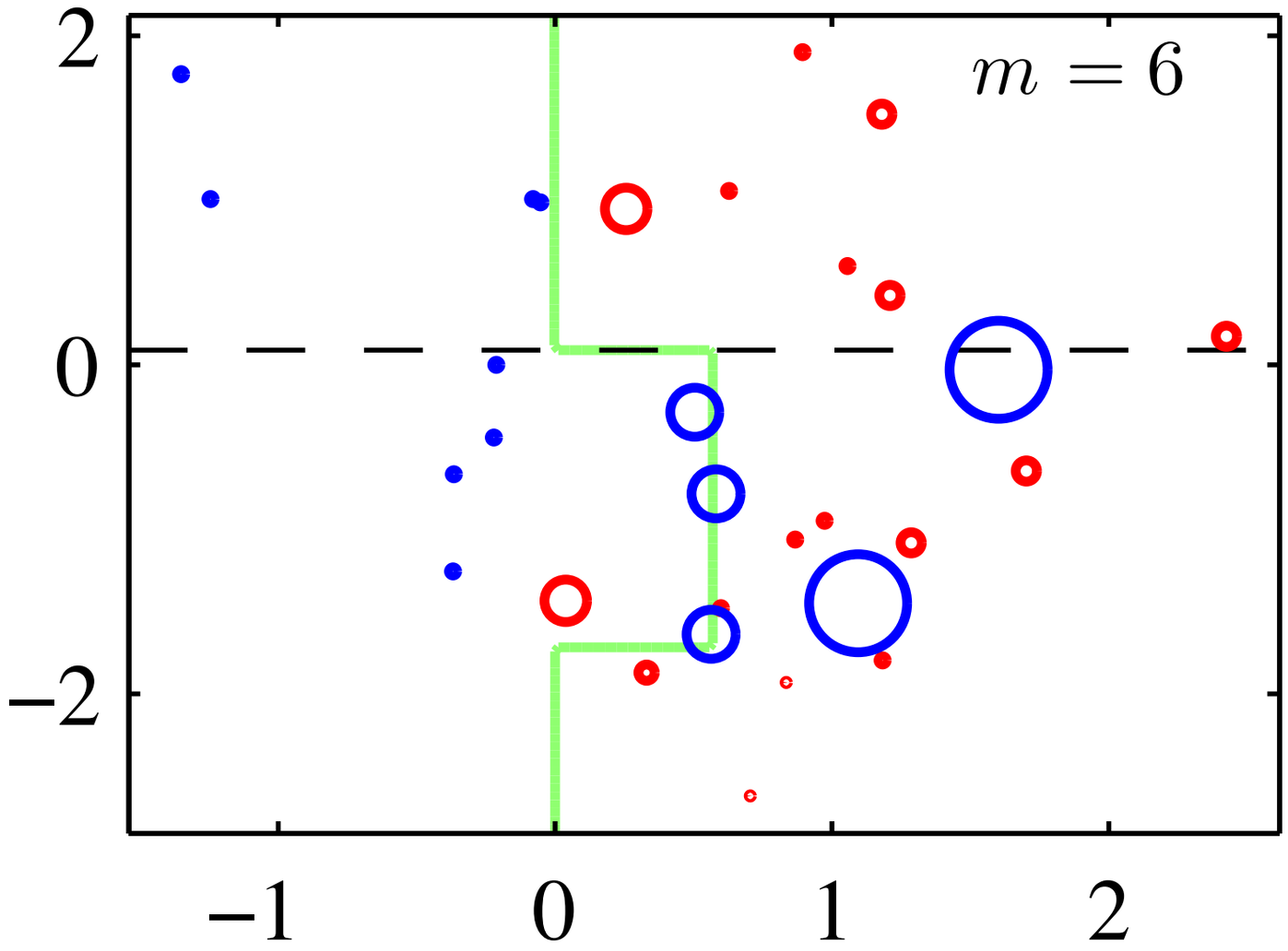
These are techniques that combine many very simple algorithms ("weak learners") to create a powerful final algorithm.



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

Boosting

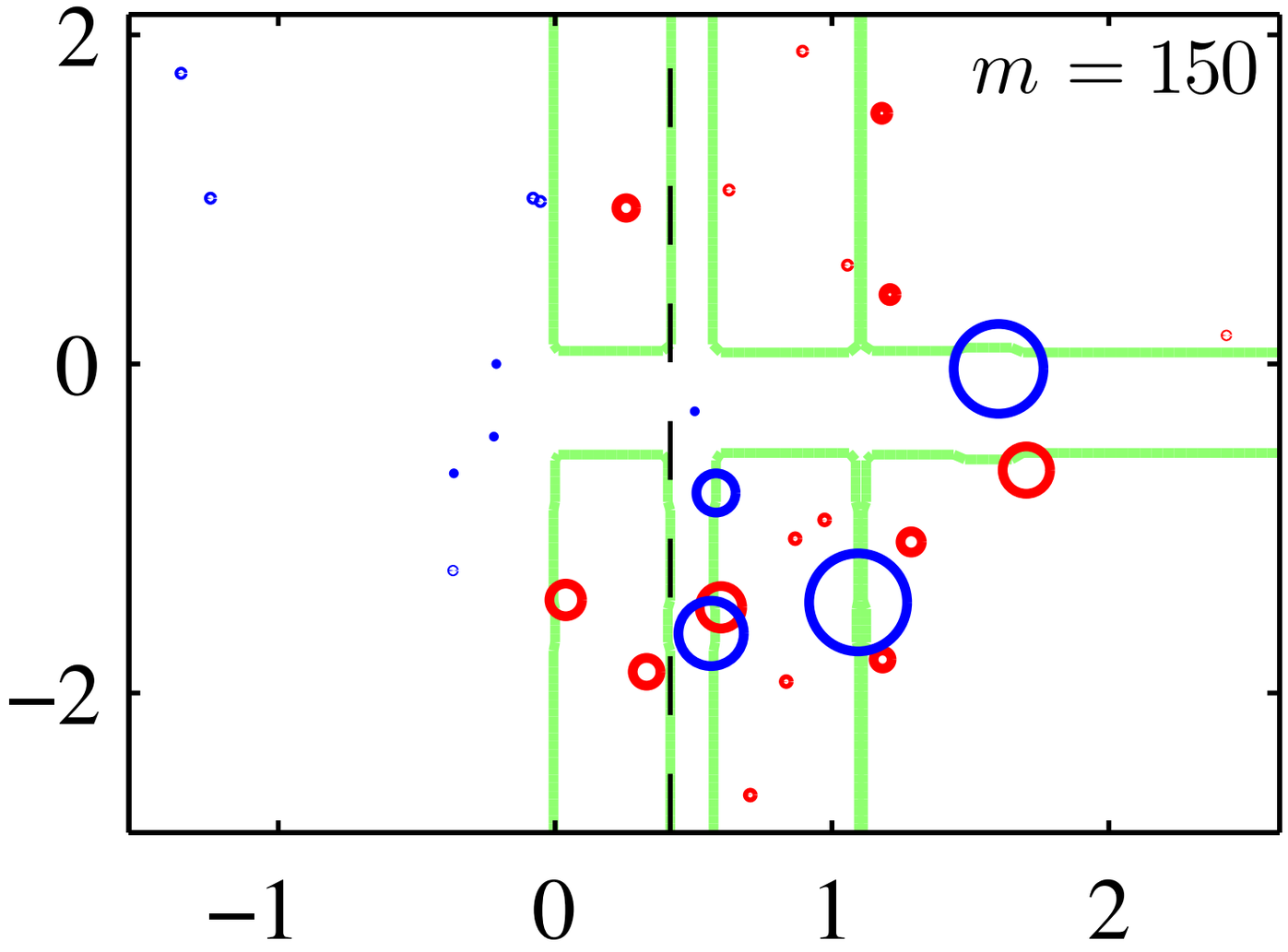
These are techniques that combine many very simple algorithms ("weak learners") to create a powerful final algorithm.



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

Boosting

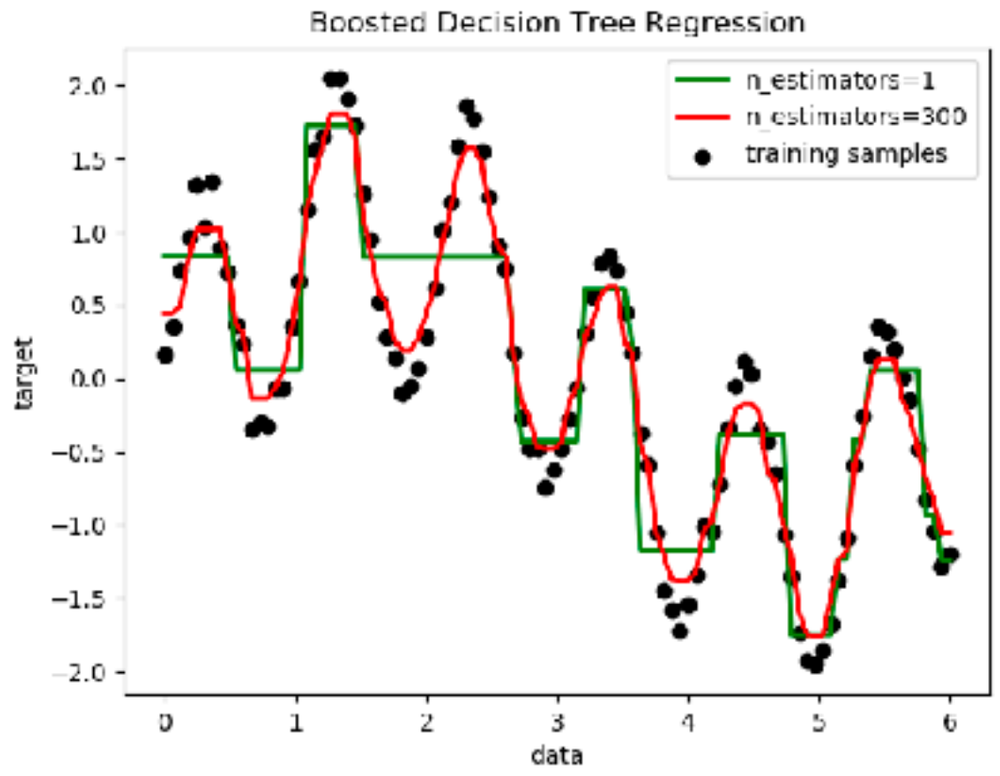
These are techniques that combine many very simple algorithms ("weak learners") to create a powerful final algorithm.



From 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.



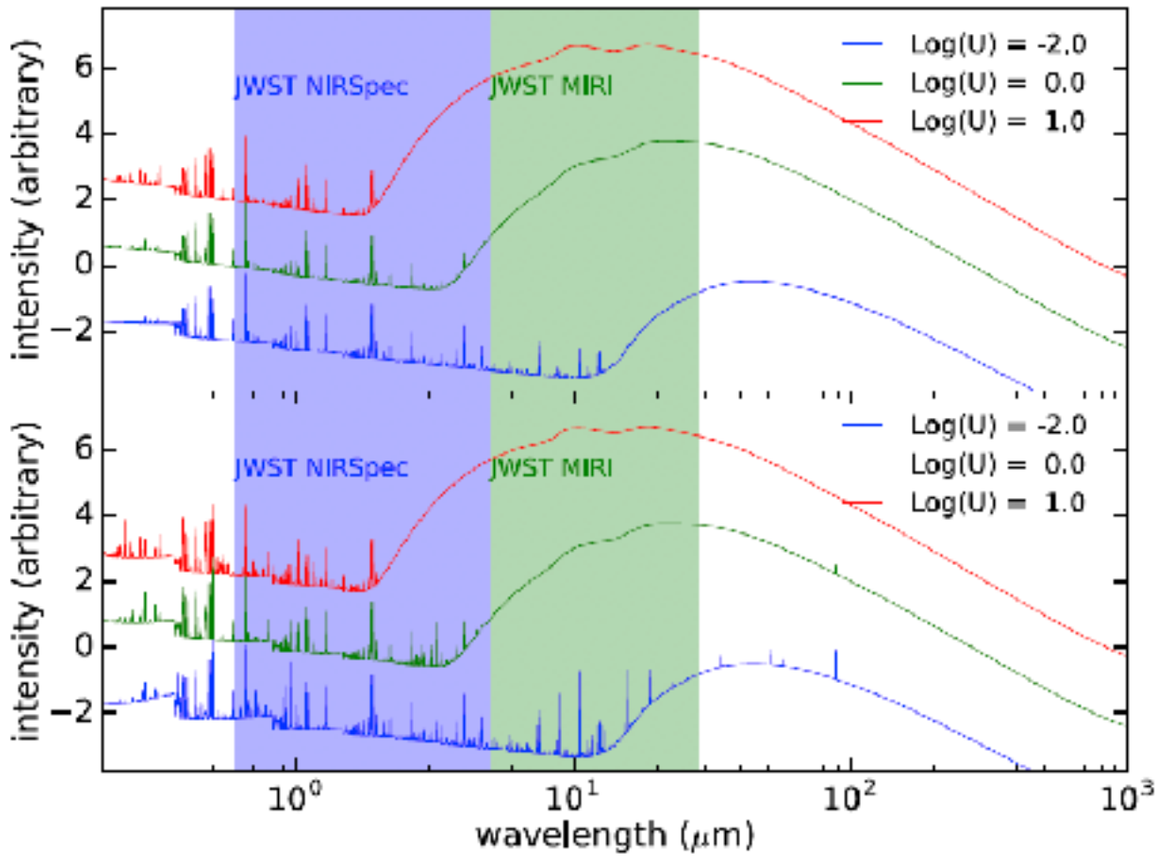

```
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import AdaBoostRegressor
<…>
regr = AdaBoostRegressor(DecisionTreeRegressor(max_depth=4),
                         n_estimators=300)
regr.fit(X, y)
y_2 = regr.predict(X)
```

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
+++

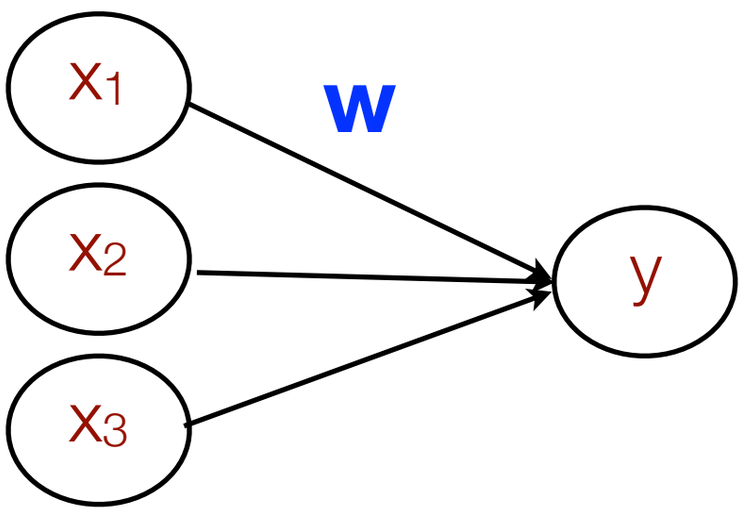
Neural networks

Linear Models - on our way to non-linearity!

We can predict a value of a parameter y using a linear combination of an **x** vector & a set of basis functions ϕ_i:

$$y(\mathbf{x}, \mathbf{w}) = \sum_i w_i \phi_i(\mathbf{x})$$

This then represents a *transformation* of the input data **x**. And we can represent this as:



The simplest Neural Network

Now expand our previous method and create M different linear combinations of **x**, ie. create M different weight vectors $\mathbf{w}_i$:

$$z_j = h\left(\sum_i w_{i,j} x_i\right)$$

The h function is called the **activation function** in neural network contexts.

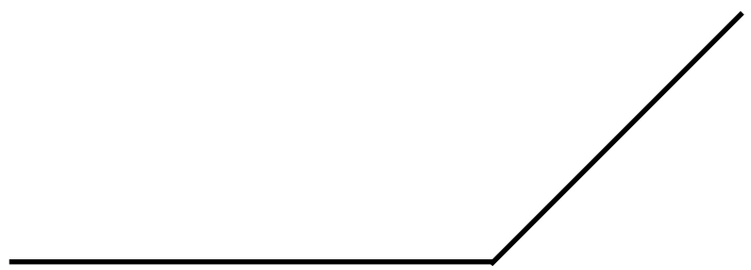
Activation functions

What function can we use?

Simplest: linear, but then the whole system is linear!

So it must be non-linear, but we would like it to behave much like a linear function. The modern choice is the rectified linear unit:

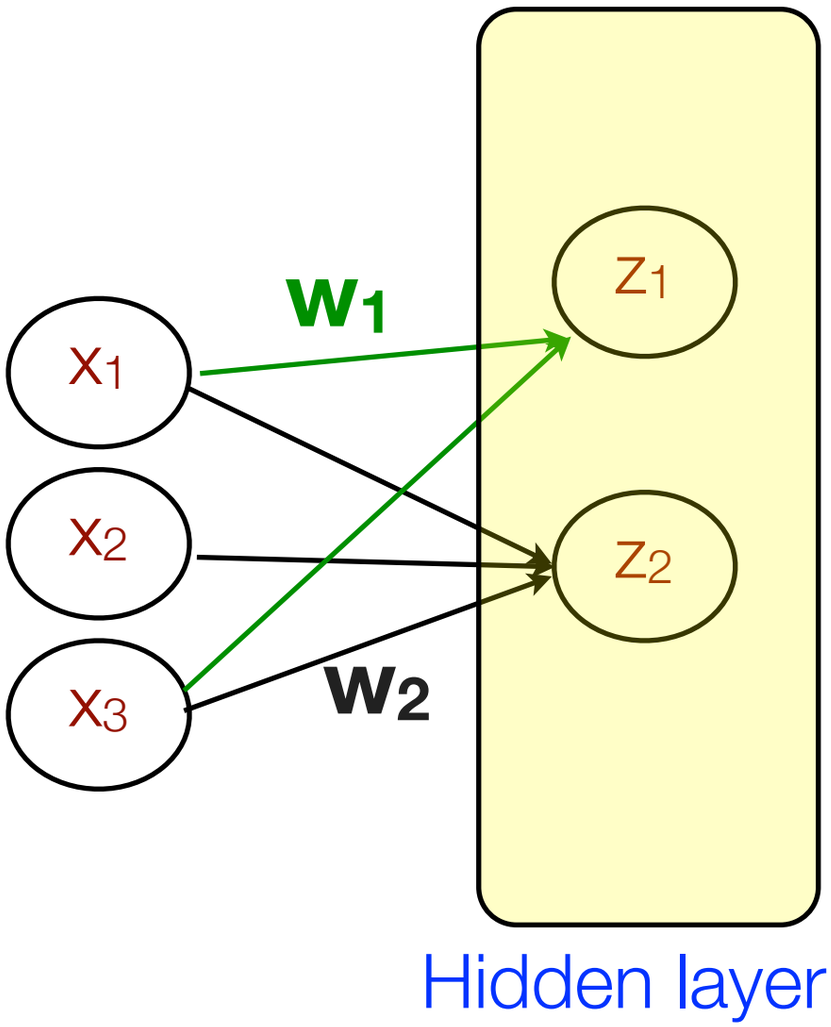
ReLU: $h(z) = \max\{0, z\}$



The simplest Neural Network

Now expand our previous method and create M different linear combinations of **x**, ie. create M different weight vectors $\mathbf{w}_i$:

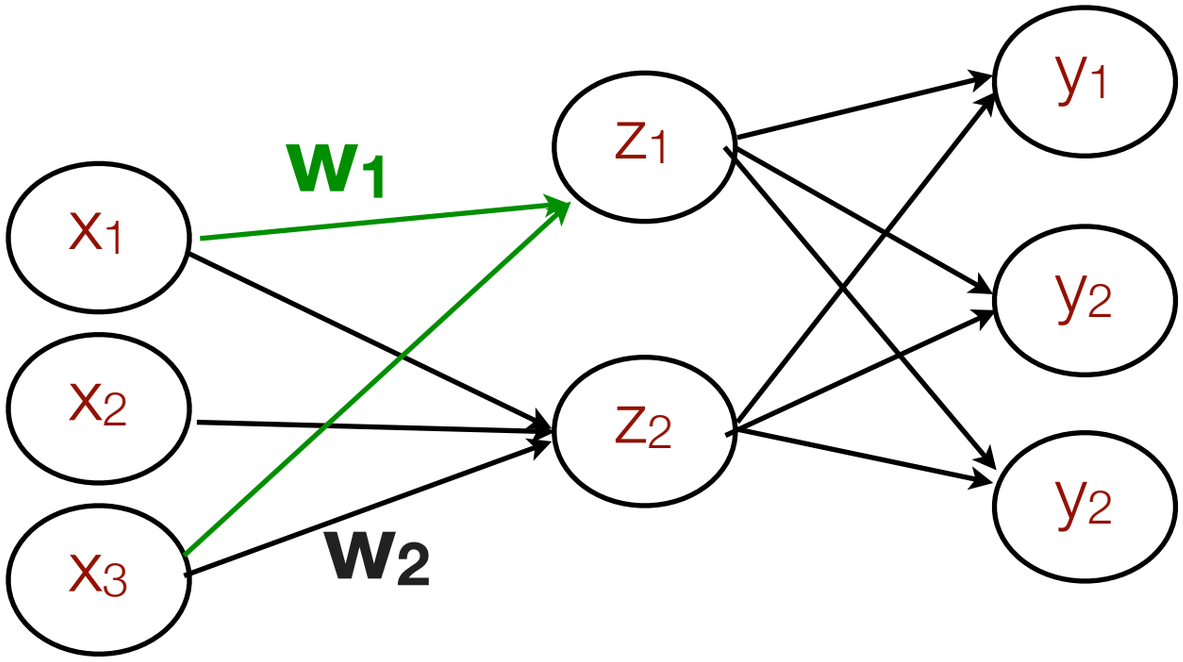
$$z_j = h\left(\sum_i w_{i,j} x_i\right)$$



The simplest Neural Network

Then we want to create one more layer - this is the *output* layer and consists of D outputs.

$$y_k = f\left(\sum_i v_{i,j} z_i\right)$$



Hidden layer

The simplest Neural Network

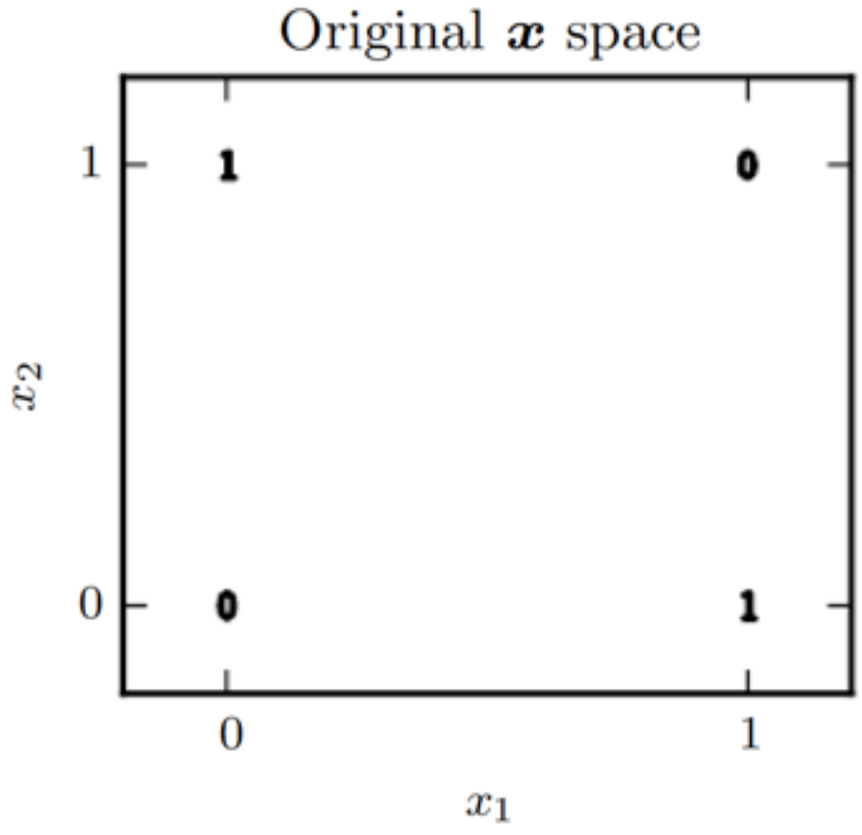
The full equation becomes:

$$y_k(\mathbf{x}, \mathbf{w}, \mathbf{v}) = f\left\{\sum_{j=1}^{M} v_{k,j}\ h\left(\sum_{i=1}^{N} w_{j,i} x_i\right)\right\}$$

For a set of inputs and a choice of weights - this can then be used to calculate y values easily. **The challenge is to determine the right choice of weights.** We will return to this when discussing deep learning.

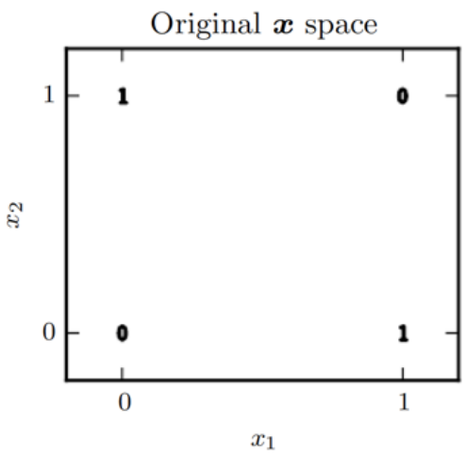
Learning XOR

$$XOR(x, y) = \begin{array}{l} 1 \text{ if either } x = 1 \text{ or } y = 1, \text{ but not both} \\ 0 \text{ otherwise} \end{array}$$



We want to learn a function of x to fit this - obviously a linear function won't work!

Learning XOR



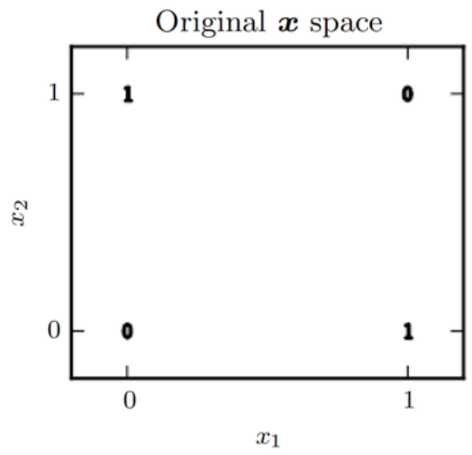
weights: $\mathbf{w}$

first layer: $\mathbf{h} = g\left(\mathbf{W}^T\mathbf{x} + \mathbf{c}\right)$

Using a ReLU:

$$f(\mathbf{x}; \mathbf{W}, \mathbf{c}, \mathbf{w}, b) = \mathbf{w}^T \max\left\{0, \mathbf{W}^T\mathbf{x} + \mathbf{c}\right\} + b$$

Learning XOR



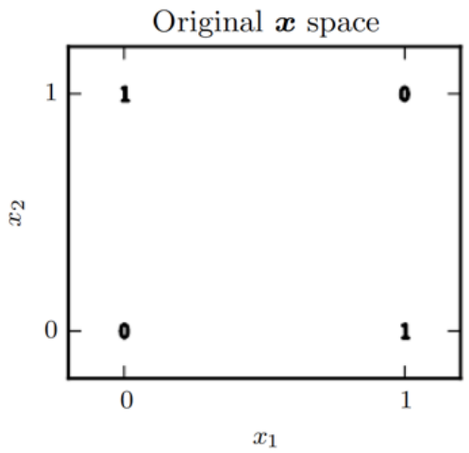
$$f(\mathbf{x}; \mathbf{W}, \mathbf{c}, \mathbf{w}, b) = \mathbf{w}^T \max\left\{0, \mathbf{W}^T\mathbf{x} + \mathbf{c}\right\} + b$$

Let: $\mathbf{W} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$ $\mathbf{c} = \begin{bmatrix} 0 \\ -1 \end{bmatrix}$ $\mathbf{w} = \begin{bmatrix} 1 \\ -2 \end{bmatrix}$

Data: $\mathbf{X} = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix}$ and: $\mathbf{XW} = \begin{bmatrix} 0 & 0 \\ 1 & 1 \\ 1 & 1 \\ 2 & 2 \end{bmatrix}$

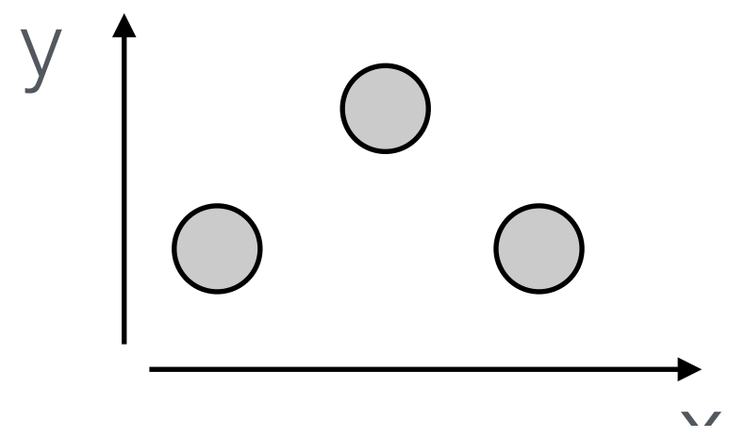
$$\mathbf{y} = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

Learning XOR

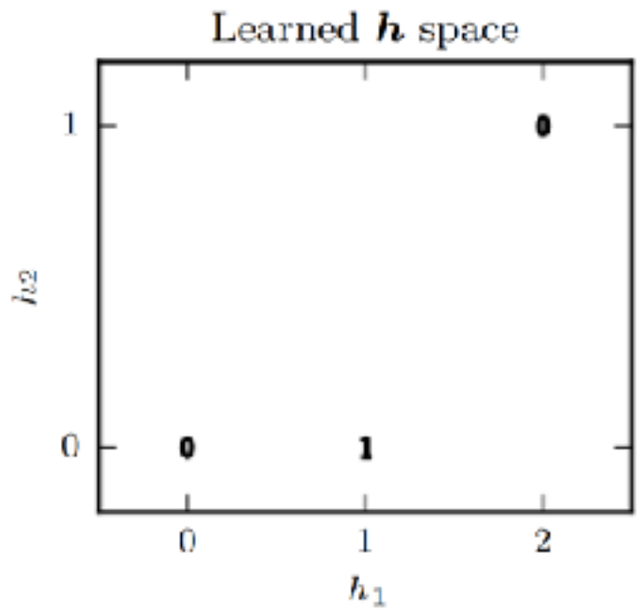


$$f(\mathbf{x}; \mathbf{W}, \mathbf{c}, \mathbf{w}, b) = \mathbf{w}^T \max\left\{0, \mathbf{W}^T \mathbf{x} + \mathbf{c}\right\} + b$$

$$\mathbf{XW} + \mathbf{c} = \begin{bmatrix} 0 & -1 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix}$$



$$\max\{0, \mathbf{XW} + \mathbf{c}\} = \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix}$$



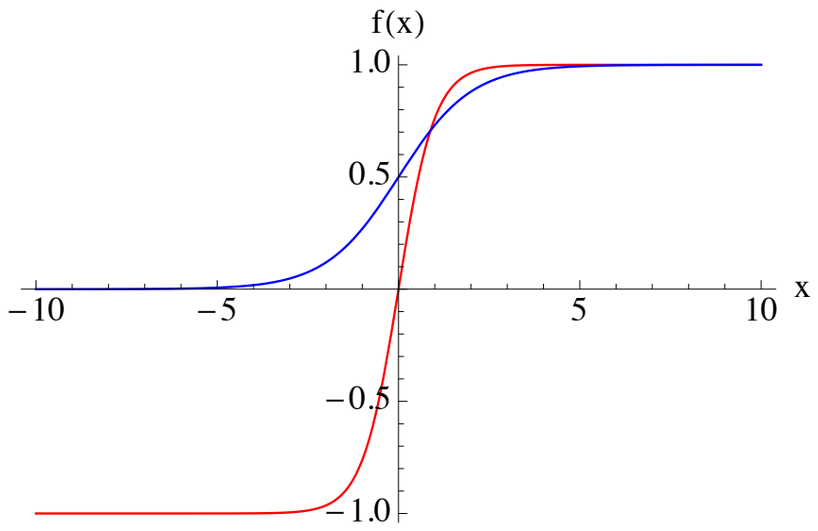
Output units - sigmoids

For classification cases, it is useful to have a function which takes -∞ to +∞ and maps it to -1 to 1 or 0 to 1. These are known as sigmoid functions. They have the advantage that they keep the signals from going off towards infinity and are useful as output units (in the past they were used as activation units too).

There are of course many of these, the most popular in our context are:

$$f(x) = \tanh(x)$$

Logistic: $$f(x) = \frac{1}{1 + e^{-x}}$$



What do you use as input?

You can give a full spectrum if you wish, or an image, but it could become time-consuming - can you do something else?

You can instead provide a (judiciously chosen) set of measurements that summarise a particular image/ spectrum - **features**.

One possibility is to use **PCA** to select features - use these to reduce the dimension of the problem and input the PCA components as input variables.

You can also provide “Observables” - but make sure you don’t provide the same information many times!

Setting weights

1) Select some training data

2) Set some random initial weights

3) Calculate output values, y_i

4) Look at differences y_i and the expected values from the training sample, t_i, using an error function

5) Minimize the difference between y_i and t_i using a minimization technique (gradient descent is typically used with some modifications)

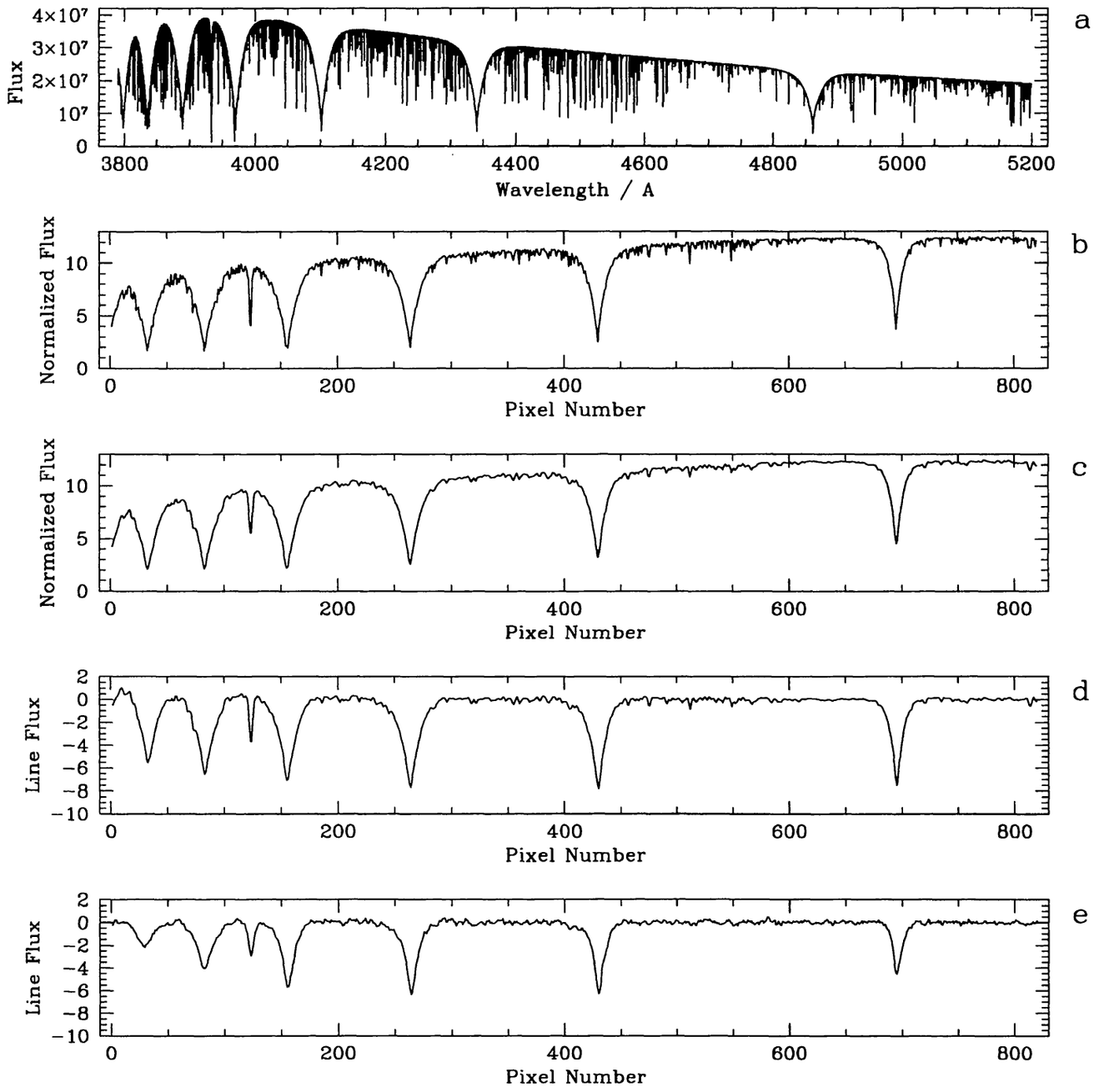
6) Iterate until convergence

7) Test on validation sample - did it work well? How complex/detailed structure can be handled by cross-validation.

Classifying spectra using artificial neural nets

Bailer-Jones et al (1997)

Input:



Classifying spectra using artificial neural nets

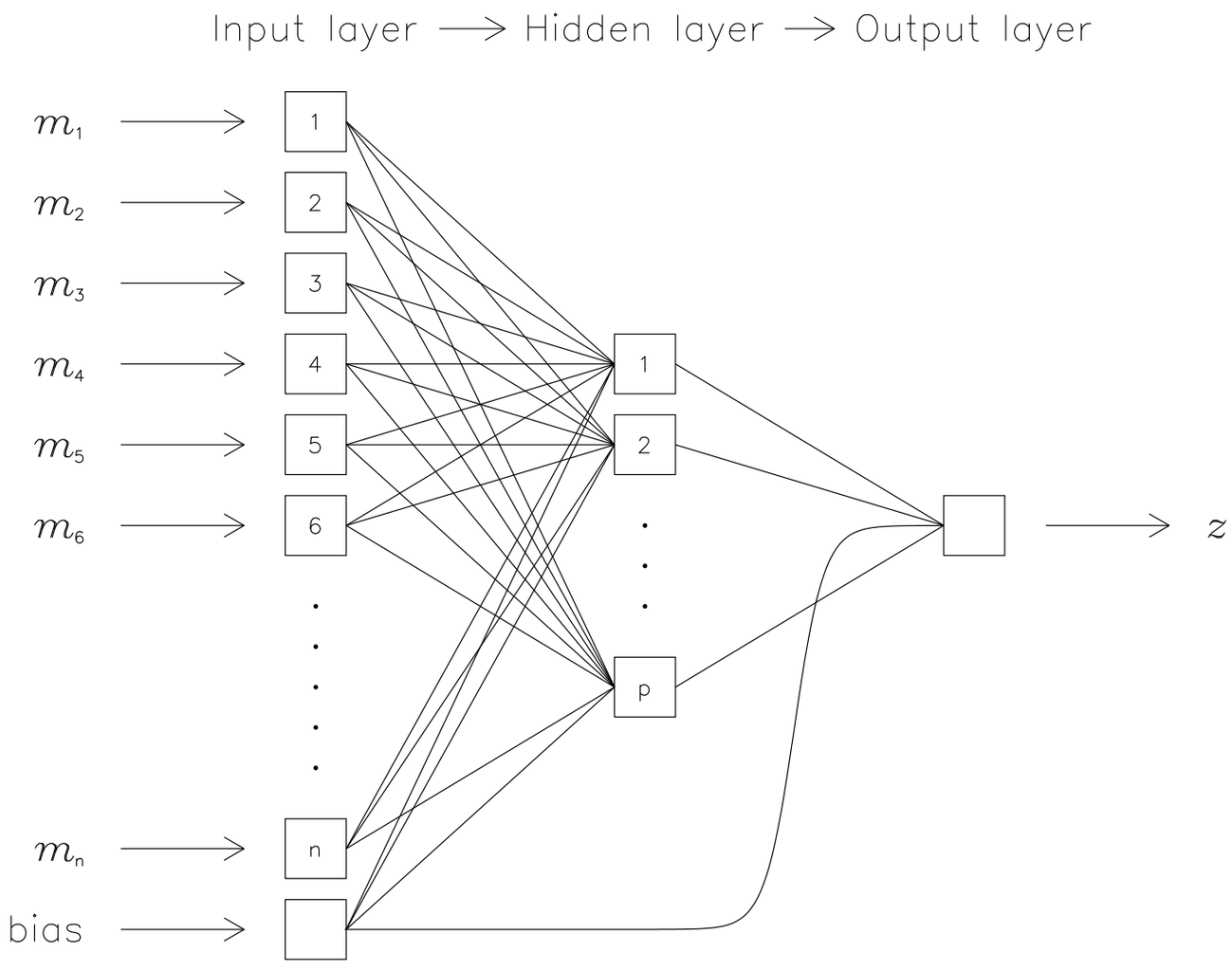
Bailer-Jones et al (1997)

Input: 821 fluxes
Neural net: 821 input units, 2 hidden layers with 5 weights each and 1 output unit

Combined 10 different neural nets (to check results, a kind of simplistic cross-validation).

Found that good estimates of T_{eff} can be found using this simple models but that metallicity introduces some uncertainty.

Redshifts from Neural Networks (ANNz)



ANNz is a freely available code that estimates a galaxy's redshift from input parameters.

Use a group of networks (a committee) and choose the best - needs to train on a comparable sample to what it will be applied to!

http://www.homepages.ucl.ac.uk/~ucapola/annz.html

Is it useful?

Appears to be useful for astronomical problems.
It does have a lot of strong believers and some strong opponents.

Advantages:

Very flexible - can approximate any function using two hidden layers.
Simple in structure and relatively easy to implement.
Widely used so plenty of literature and heuristic information.

Disadvantages:

Can be time-consuming to train and test - cross-validation can be extremely time-consuming.
Very hard to “understand” - do you get any insight from it?