Lecture 3 - Model choice and inference

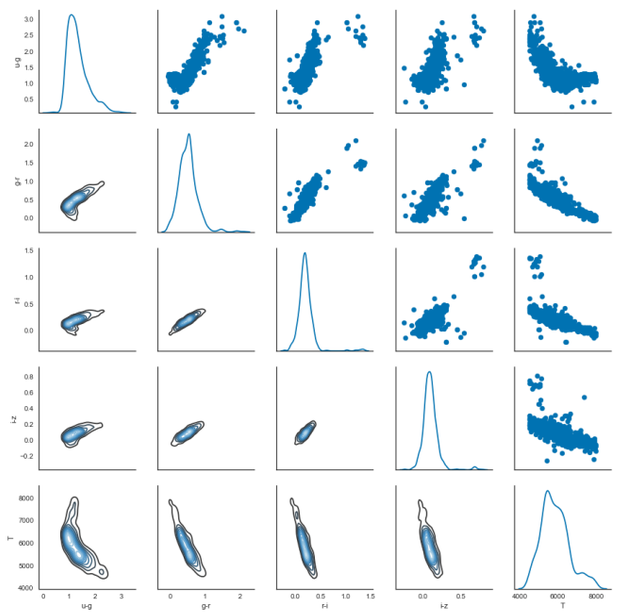
See https://github.com/jbrinchmann/MLD2019 as usual

Lectures/Lecture 3 has the PDF for today and

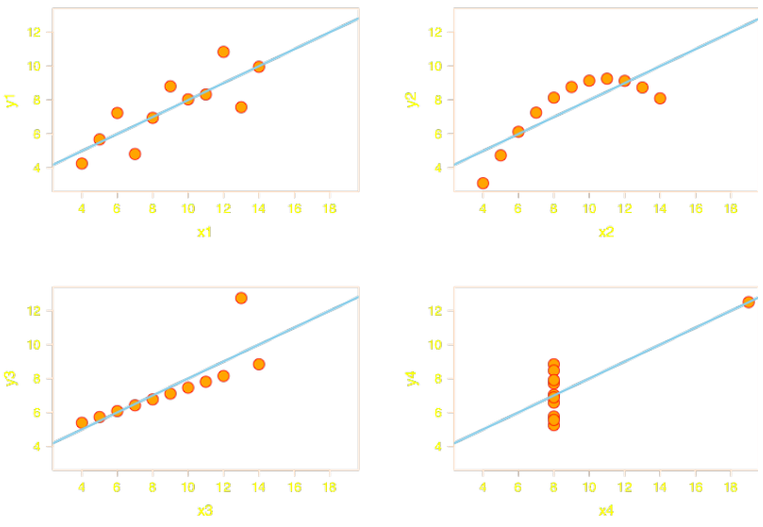
A review of last lecture

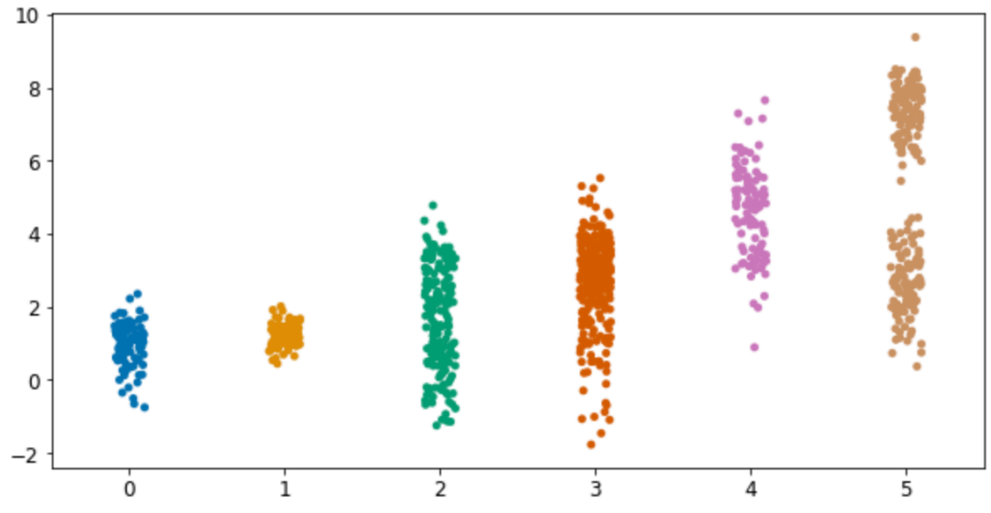
```
import sqlite3 as lite

con = lite.connect('DB.db')
con.execute('SELECT * FROM Stars')
```

Accessing databases with Python



Methods for visualisation

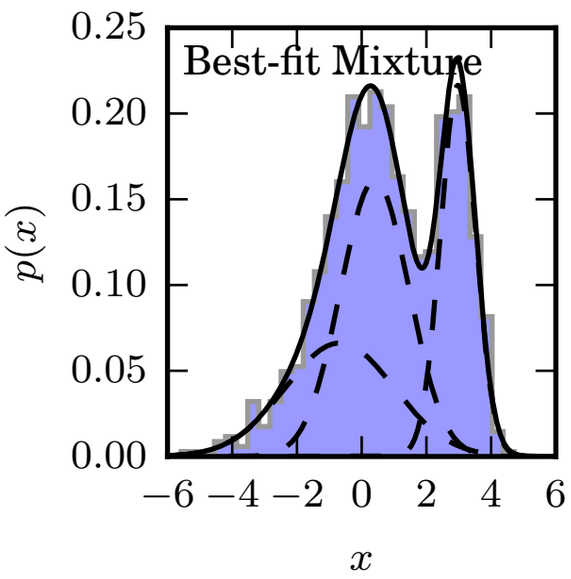




Density estimation

Kernel methods:

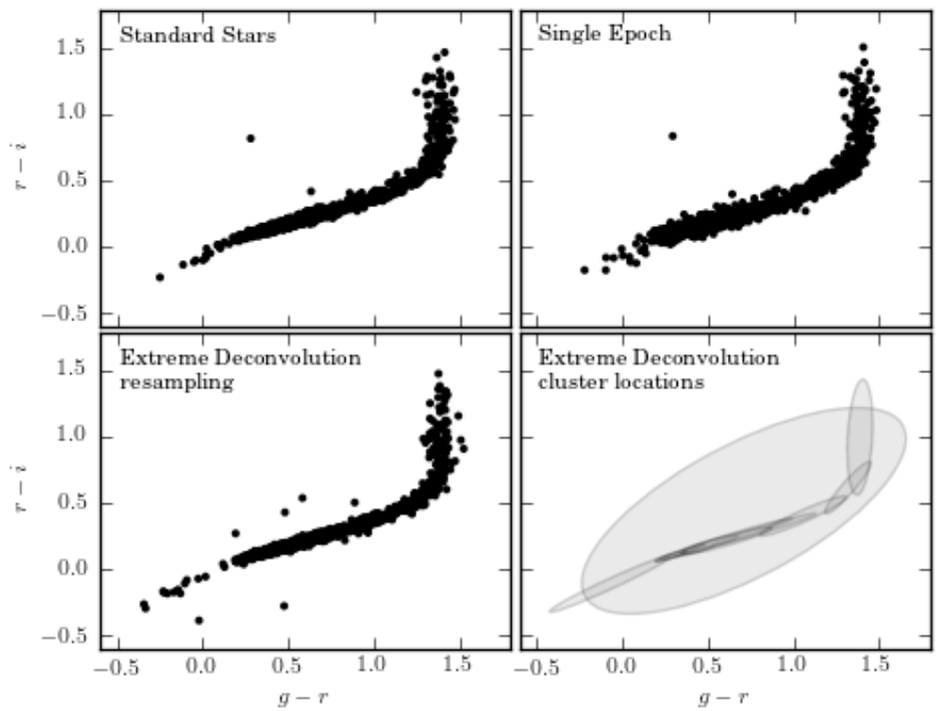
$$p(\mathbf{x}) = \frac{1}{N}\sum_{i=1}^{N}\frac{1}{V_i}k\left(\frac{\mathbf{x}-\mathbf{x}_i}{h_i}\right)$$



Gaussian mixture models

$$p(x_i|\boldsymbol{\theta}) = \sum_{j=1}^{M}\alpha_j N(x_i;\mu_j,\sigma_j)$$

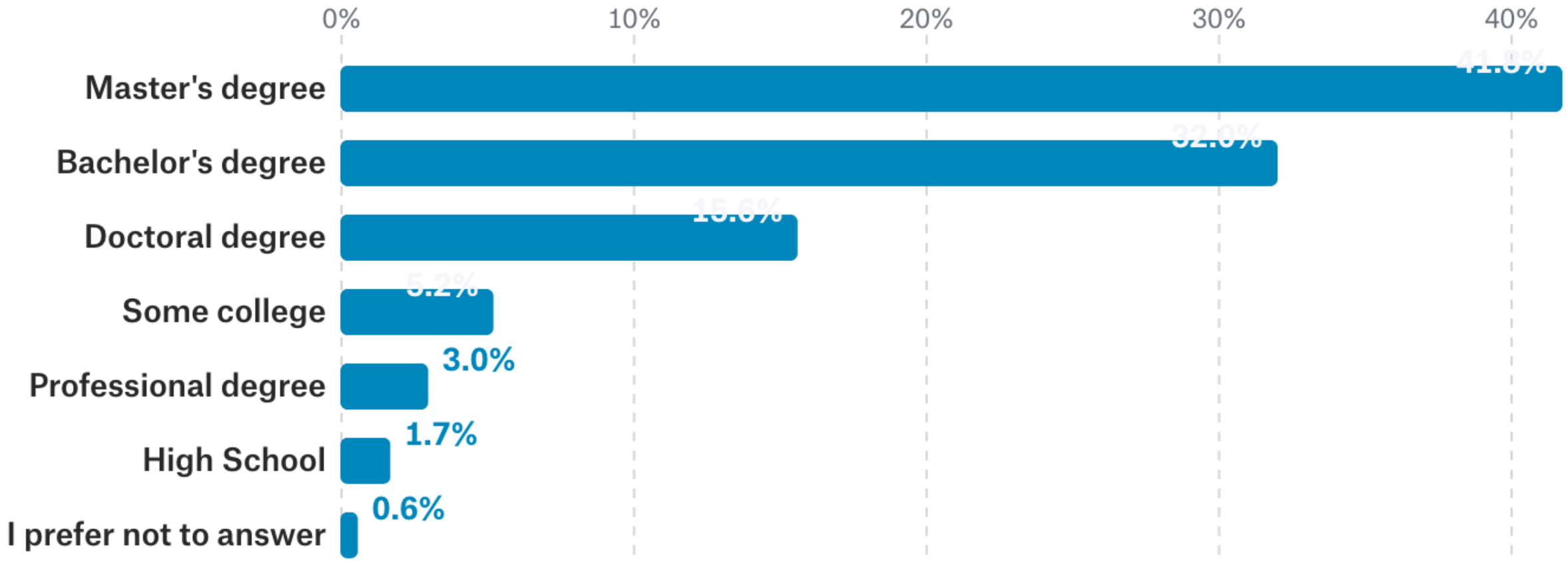
Extreme deconvolution:



Some fun survey results

From Kaggle's user survey:

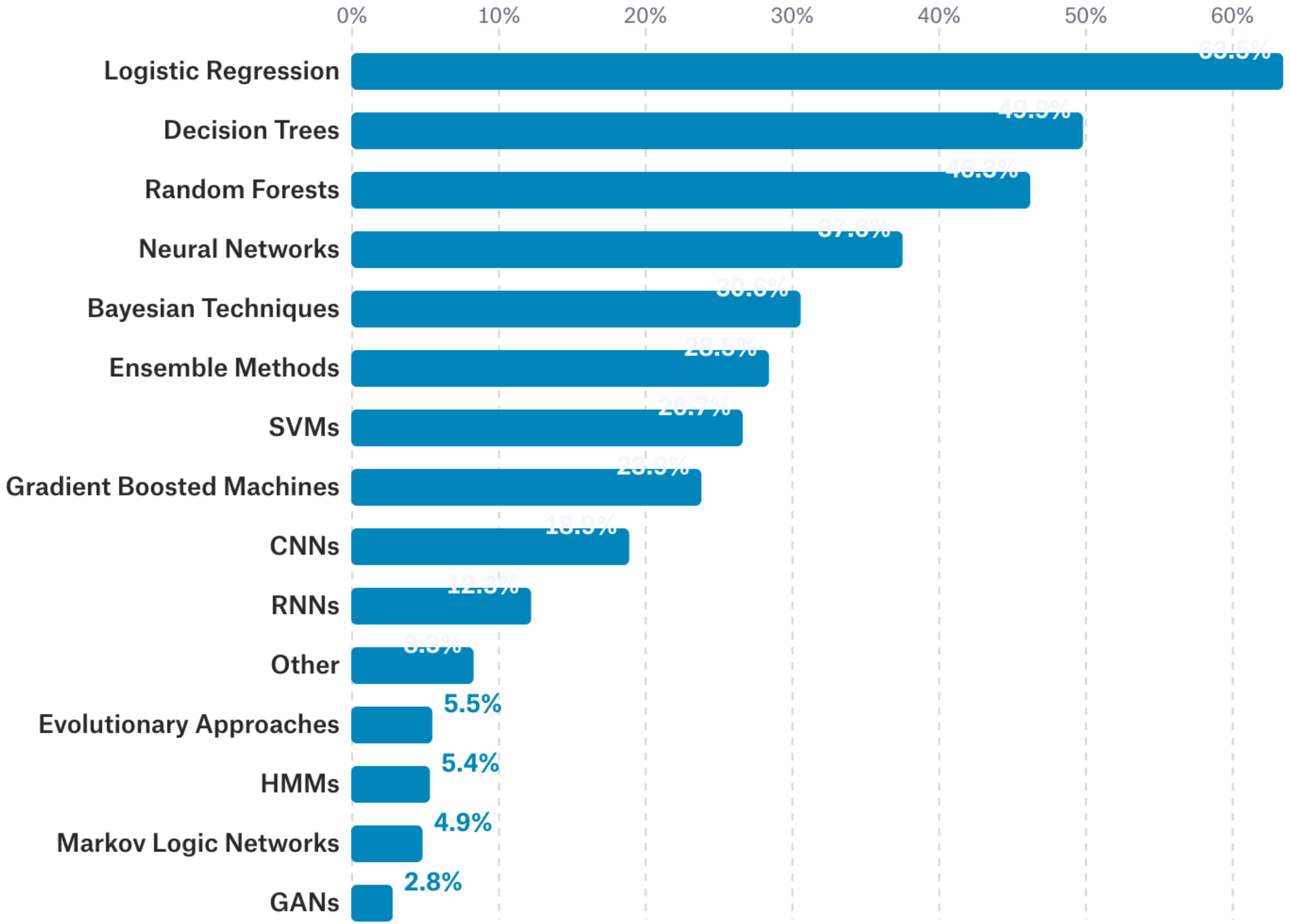
Participant's highest education level:



15,015 responses

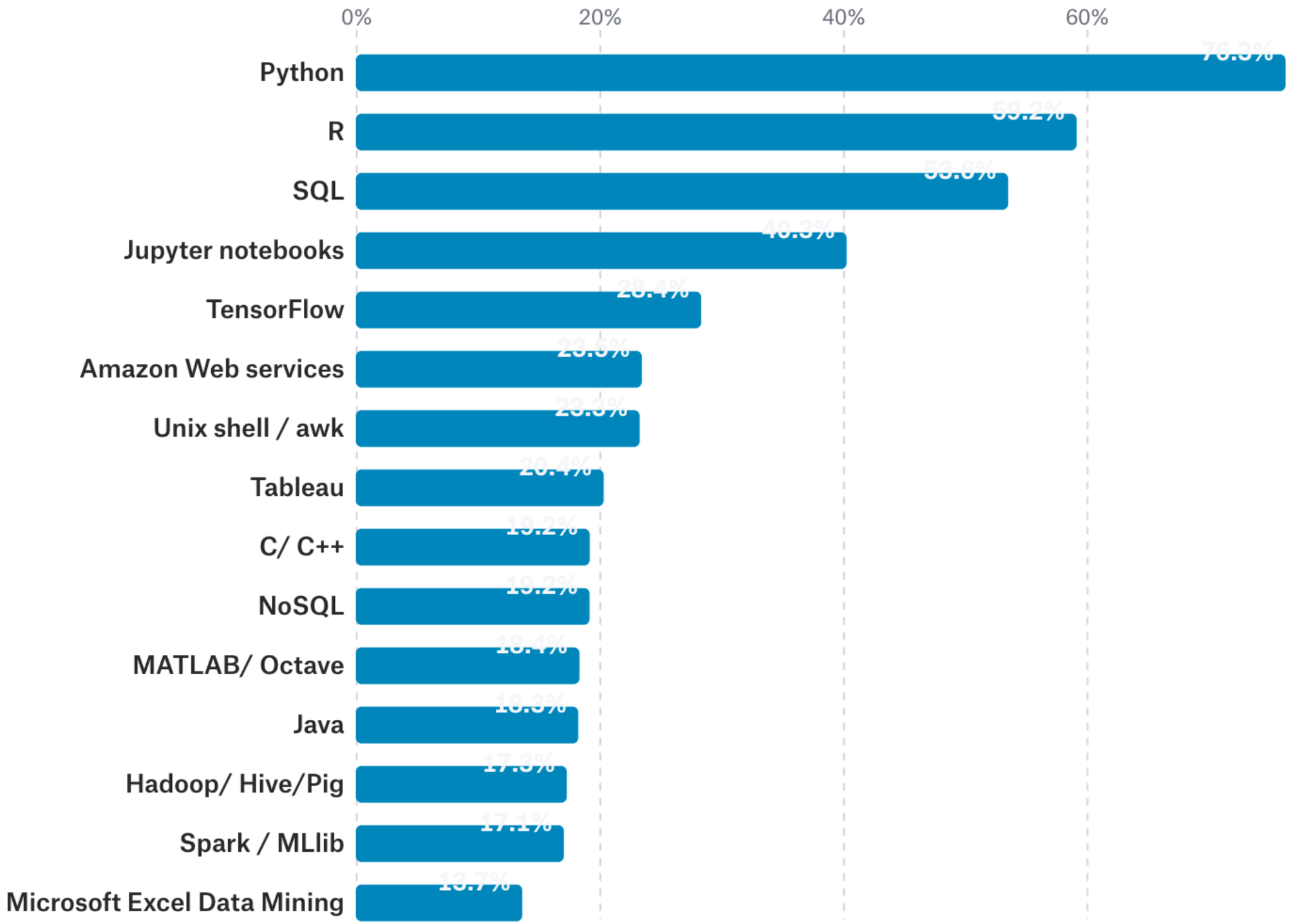
View code in Kaggle Kernels

Techniques used at work (lots we have not discussed!):



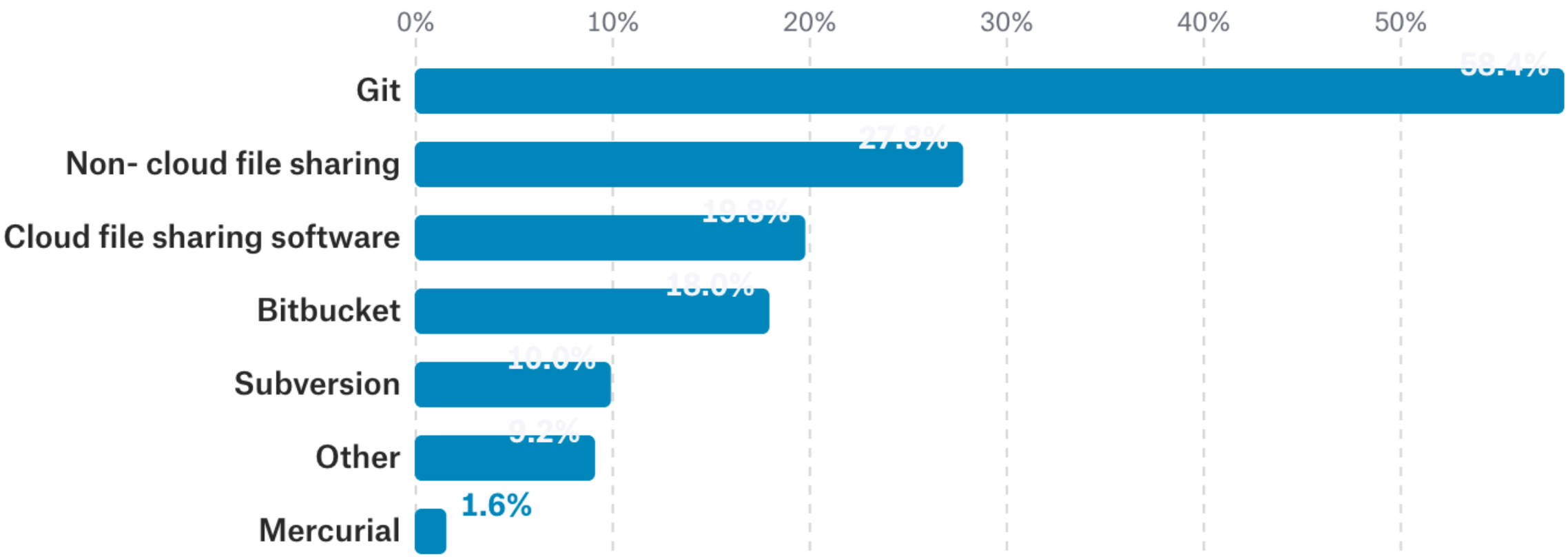
7,301 responses

Tools used at work:



7,955 responses

Code sharing used at work:



6,203 responses

See more at:
https://www.kaggle.com/code/mhajabri/what-do-kagglers-say-about-data-science

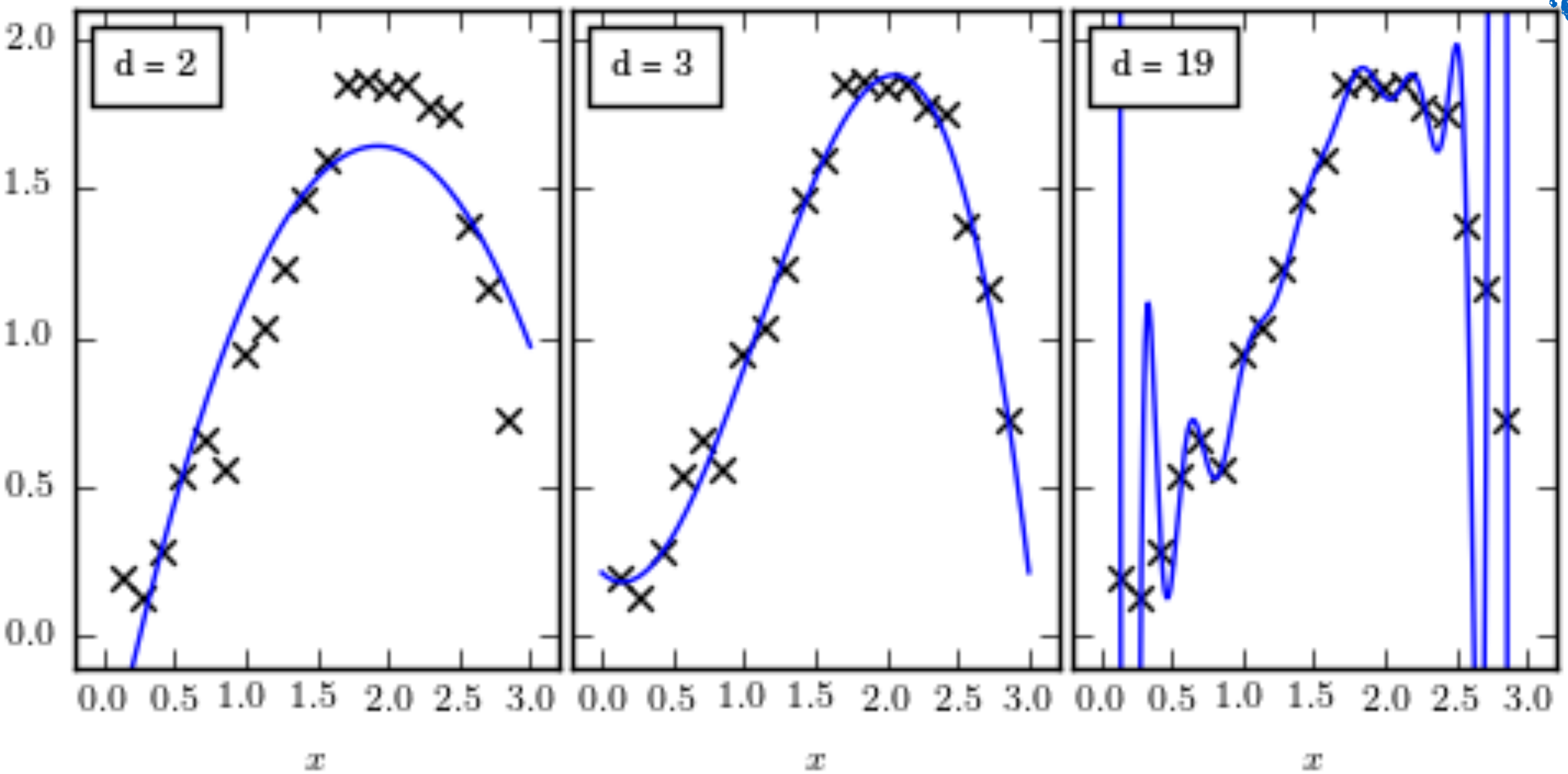
When is good, good enough?

Model choice/complexity decision

From astroML.org

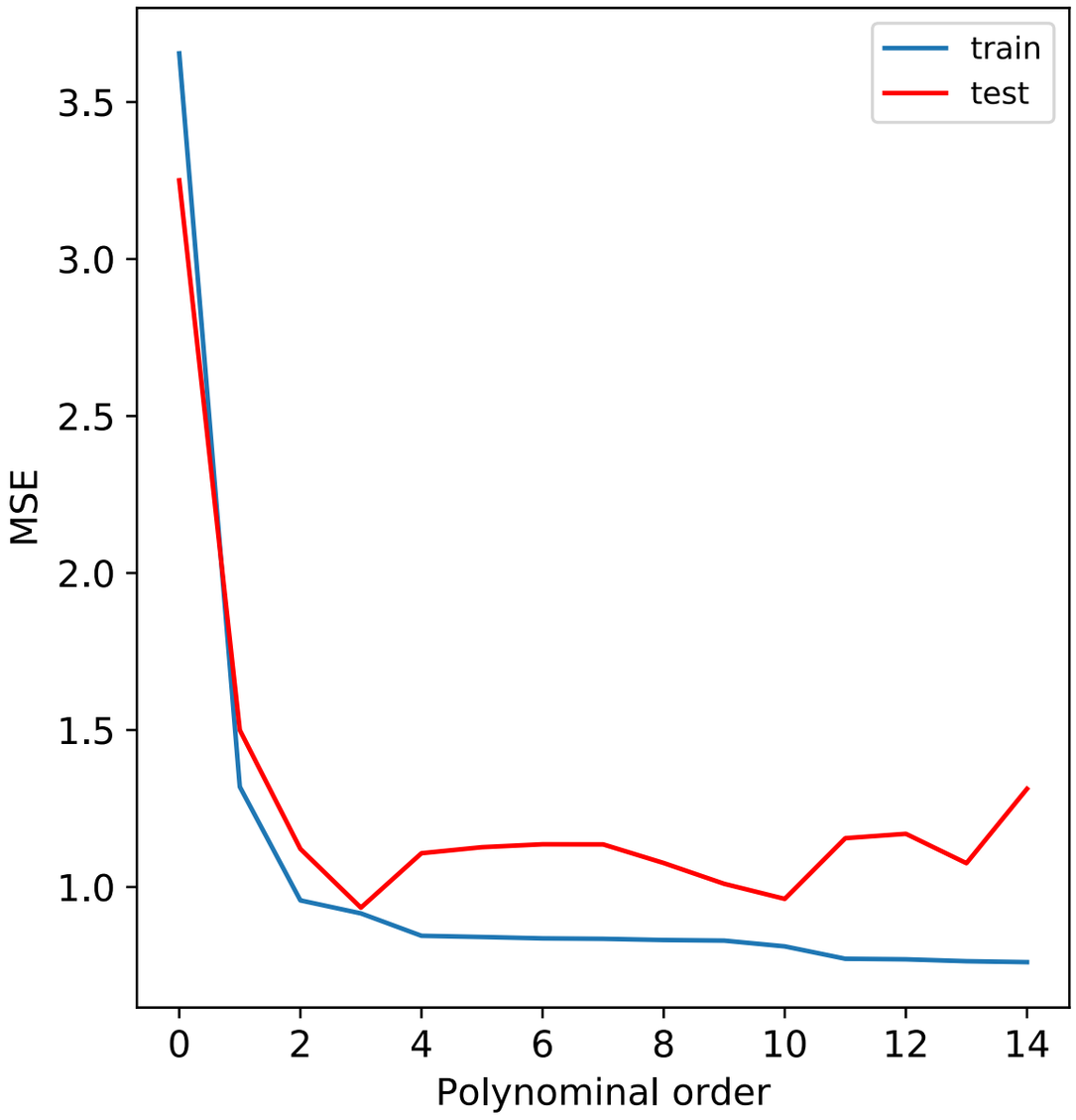
Reminder

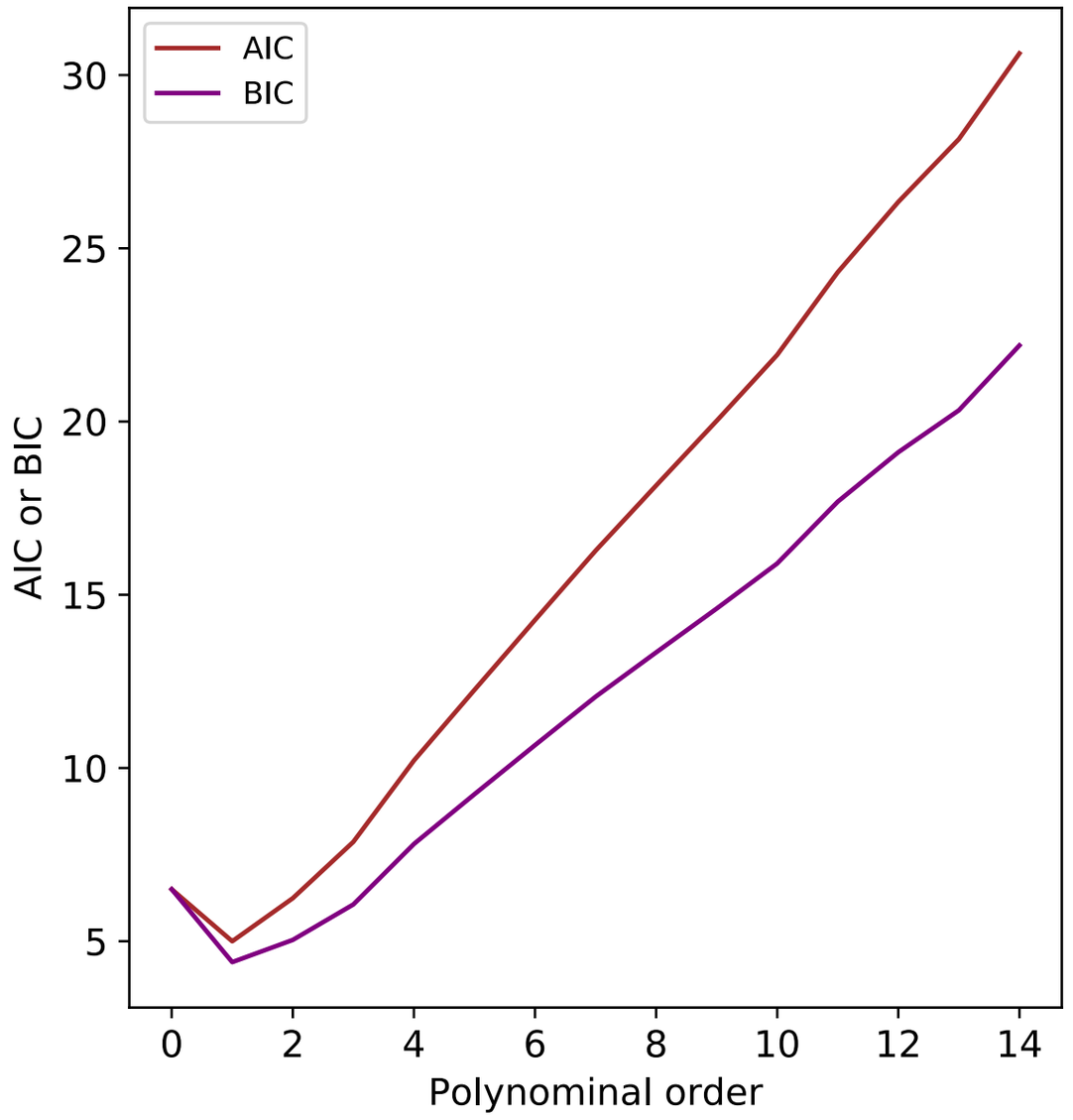
Increasing the flexibility:



Our eye would probably say a 3rd degree polynomial is ok.

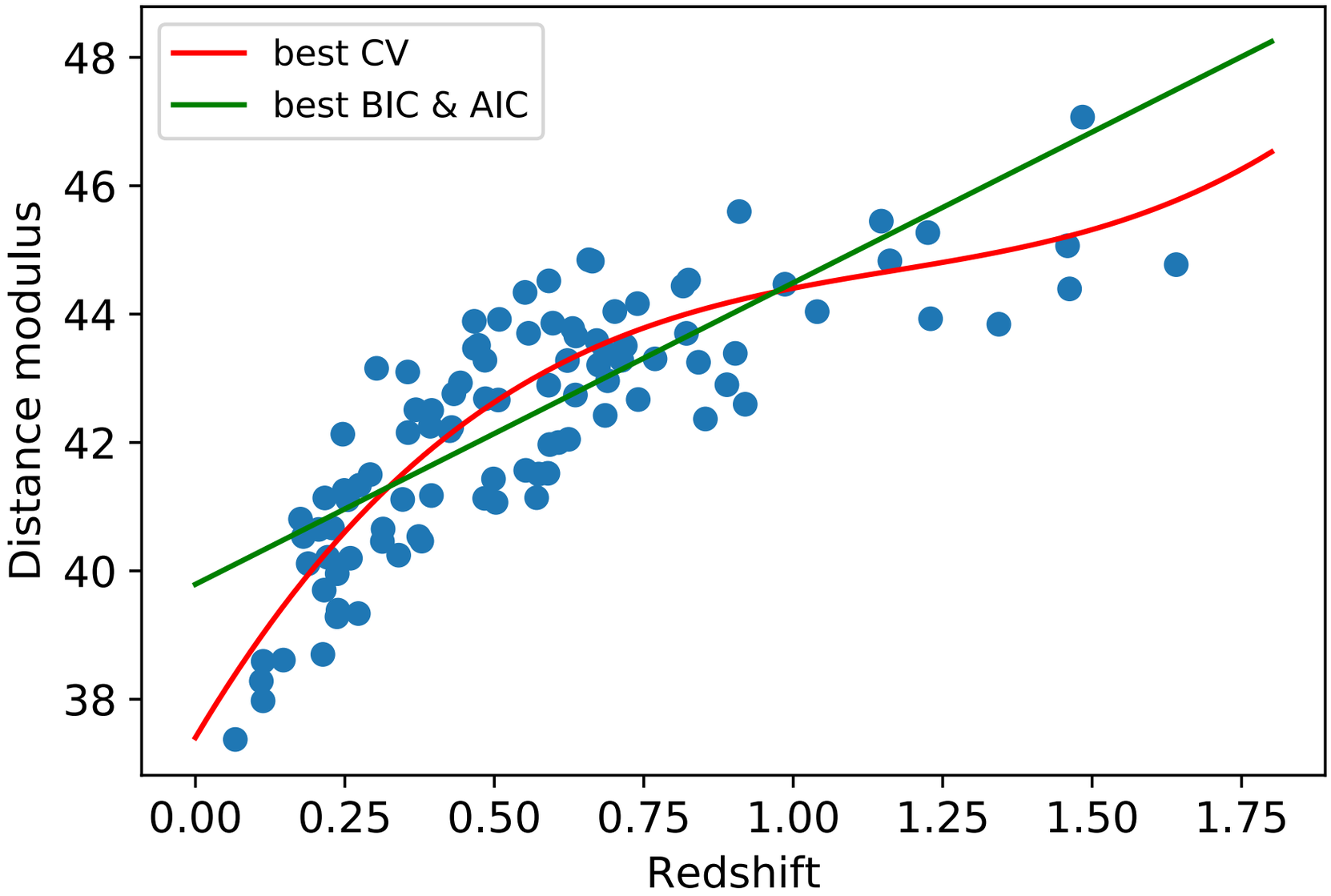
A simple example - result





So CV gives 3 as the best order, and AIC & BIC gives 1…

A simple example - result



Illustrates the challenge of “best fit” - it is not always obvious. The BIC and AIC often want simple models, while the data might support more complexity.

Why does it matter - the bias-variance trade-off

Complexity versus simplicity - the bias variance trade-off

Assume that the true relation between x & y is

$$y = f(x) + \epsilon \qquad \epsilon \sim N(0, \sigma_\epsilon)$$

And write the predicted value of y at x_0 $\hat{y}$

Let us ask what the expected square error in our prediction at $x{=}x_0$ is:

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

Small versus big bins - the bias variance trade-off

Expected mean square error at $x=x_0$:

$$E\left[(Y-\hat{y})^2\right][x=x_0] \qquad Y(x)=f(x)+\epsilon \qquad \epsilon \sim N(0,\sigma_\epsilon)$$

We can expand this and add and subtract $(\mathrm{E}[\hat{y}])^2$

to get:

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

Small versus big bins - the bias variance trade-off

Expected mean square error at $x=x_0$:

$$E\left[(Y-\hat{y})^2\right][x=x_0] \qquad Y(x)=f(x)+\epsilon \qquad \epsilon \sim N(0,\sigma_\epsilon)$$

We can expand this and add and subtract $(\mathrm{E}[\hat{y}])^2$ to get:

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

Bias²

Small versus big bins - the bias variance trade-off

Expected mean square error at $x=x_0$:

$$E\left[(Y-\hat{y})^2\right][x=x_0] \qquad Y(x)=f(x)+\epsilon \qquad \epsilon \sim N(0,\sigma_\epsilon)$$

We can expand this and add and subtract $(\mathrm{E}[\hat{y}])^2$ to get:

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

Bias^2 Variance

Small versus big bins - the bias variance trade-off

Expected mean square error at $x=x_0$:

$$E\left[(Y-\hat{y})^2\right][x=x_0] \qquad Y(x)=f(x)+\epsilon \qquad \epsilon \sim N(0,\sigma_\epsilon)$$

We can expand this and add and subtract $(\mathrm{E}[\hat{y}])^2$ to get:

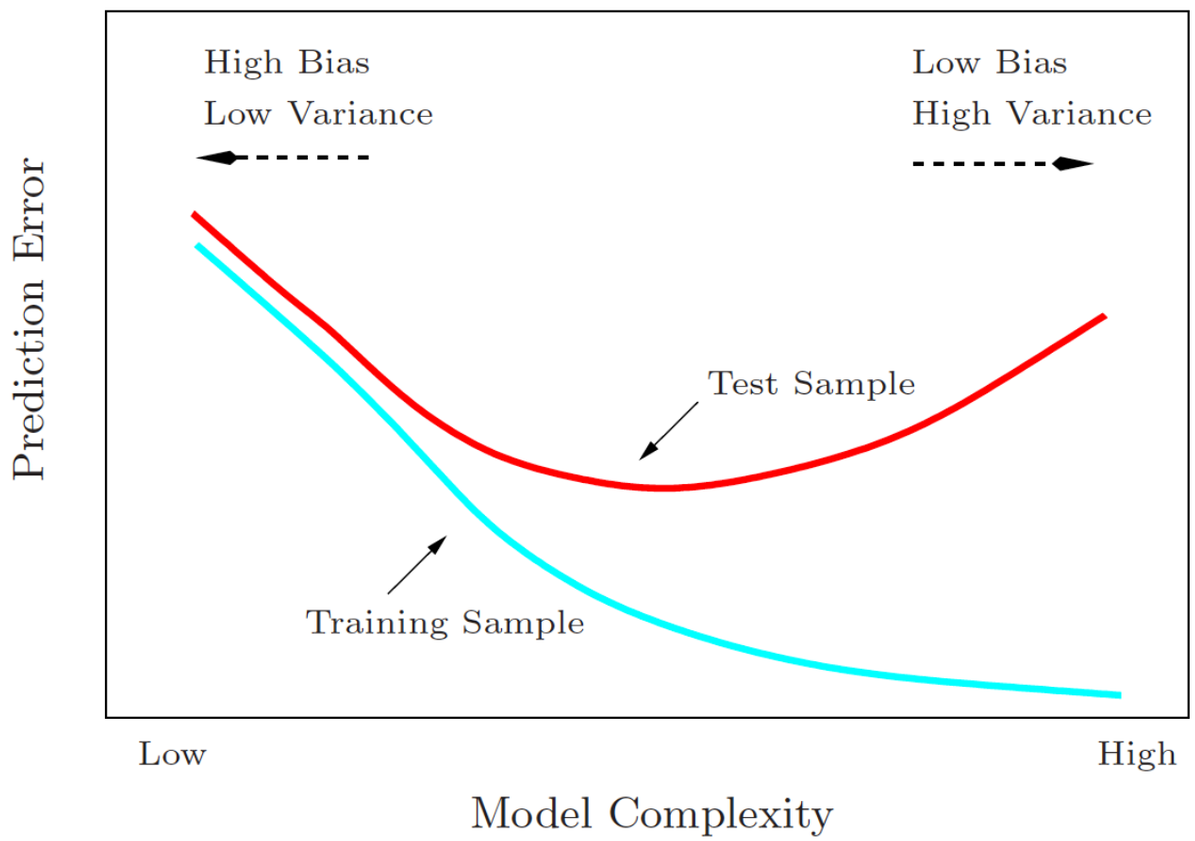
$$\mathrm{Err}(x_0) = \sigma_\epsilon^2 + (E[\hat{y}]-f(x_0))^2 + \mathrm{E}\left[(\hat{y}-\mathrm{E}[\hat{y}])^2\right]$$

Irreducible Error — Bias² — Variance

The bias variance trade-off

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

This decomposition into two positive terms is a general result and usually we have to choose our method to have low bias *or* low variance but not both.



From “The Elements of Statistical Learning - Data Mining, Inference, and Prediction”, Hastie, Tibshirani & Friedmann, second edition, Figure 2.11

Evaluating the generalisation error

What we want:

How well does a fitted function or trained Machine Learning method work on new data? **Generalisation error**

We can’t quite have that - but this is where we use the test set/validation set.

One possibility

This is fine if we have 100,000s of data (usually), but can leave too few points to train on.

Evaluating the generalisation error

What we want:

How well does a fitted function or trained Machine Learning method work on new data? **Generalisation error**

We can’t quite have that - but this is where we use the test set/validation set.

One possibility

This is fine if we have 100,000s of data (usually), but can leave too few points to train on.

Evaluating the generalisation error

What we want:

How well does a fitted function or trained Machine Learning method work on new data? **Generalisation error**

We can’t quite have that - but this is where we use the test set/validation set.

One possibility

This is fine if we have 100,000s of data (usually), but can leave too few points to train on.

Evaluating the generalisation error

What we want:

How well does a fitted function or trained Machine Learning method work on new data? **Generalisation error**

We can’t quite have that - but this is where we use the test set/validation set.

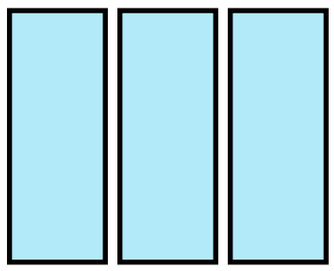
One possibility

This is fine if we have 100,000s of data (usually), but can leave too few points to train on.

Cross-validation to the rescue

One possibility

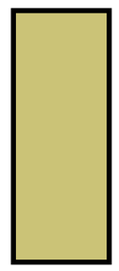
k-fold CV



Training

.....

k-copies

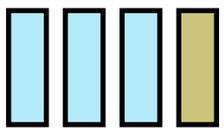


Test

Do this splitting k times.

It is generally recommended to use 5- or 10- fold cross-validation.

The details of cross-validation



Divide the set of N data into K (almost) equal subsets (folds), F_i.

Let κ_i be the set of that are set aside in the fit in iteration i

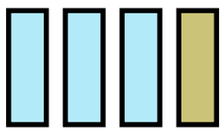
The error in estimation for that iteration is then (for RSS):

$$\mathrm{err}_k = \sum_{i \in F_k} \left[y_i - \hat{y}\left(-\kappa_i\right) \right]^2$$

where -κ(i) signifies that the estimate is calculated with the objects in κ(i) excluded.

See “The Elements of Statistical Learning - Data Mining, Inference, and Prediction”, Hastie, Tibshirani & Friedmann, second edition, chapter 7

The details of cross-validation



Divide the set of N data into K (almost) equal subsets (folds), F_i.

Let κ_i be the set of that are set aside in the fit in iteration i

The error in estimation for that iteration is then (for RSS):

$$\mathrm{err}_k = \sum_{i \in F_k} \left[y_i - \hat{y}\left(-\kappa_i\right) \right]^2$$

where -κ(i) signifies that the estimate is calculated with the objects in κ(i) excluded.

The CV estimate of the generalisation error is the average of this over the K folds:

$$\frac{1}{N} \sum_{k=1}^{K} \mathrm{err}_k$$

See “The Elements of Statistical Learning - Data Mining, Inference, and Prediction”, Hastie, Tibshirani & Friedmann, second edition, chapter 7

Example: determining the optimal bandwidth for KDE

KDE allows us to construct a distribution function. $p_x(\boldsymbol{x})$

This should tell me how likely a value z is, so I can evaluate $p_x(z)$ for z in my **test set**. This gives the likelihood of that test set.

Do this for all k folds.

$$\ln L_{\mathrm{CV}} = \frac{1}{N}\sum_{i=1}^{K} \ln p_{x_i}(z_i)$$

Then do this for all band-widths and select the best (highest L_{CV} in this case)

Example: determining the optimal bandwidth for KDE

KDE allows us to construct a distribution function. $p_x(\boldsymbol{x})$

This should tell me how likely a value z is, so I can evaluate $p_x(z)$ for z in my **test set**. This gives the likelihood of that test set.

Loss function = $\mathrm{err_k}$

Do this for all k folds.

$$\ln L_{\mathrm{CV}} = \frac{1}{N} \sum_{i=1}^{K} \ln p_{x_i}(z_i)$$

Then do this for all band-widths and select the best (highest L_{CV} in this case)

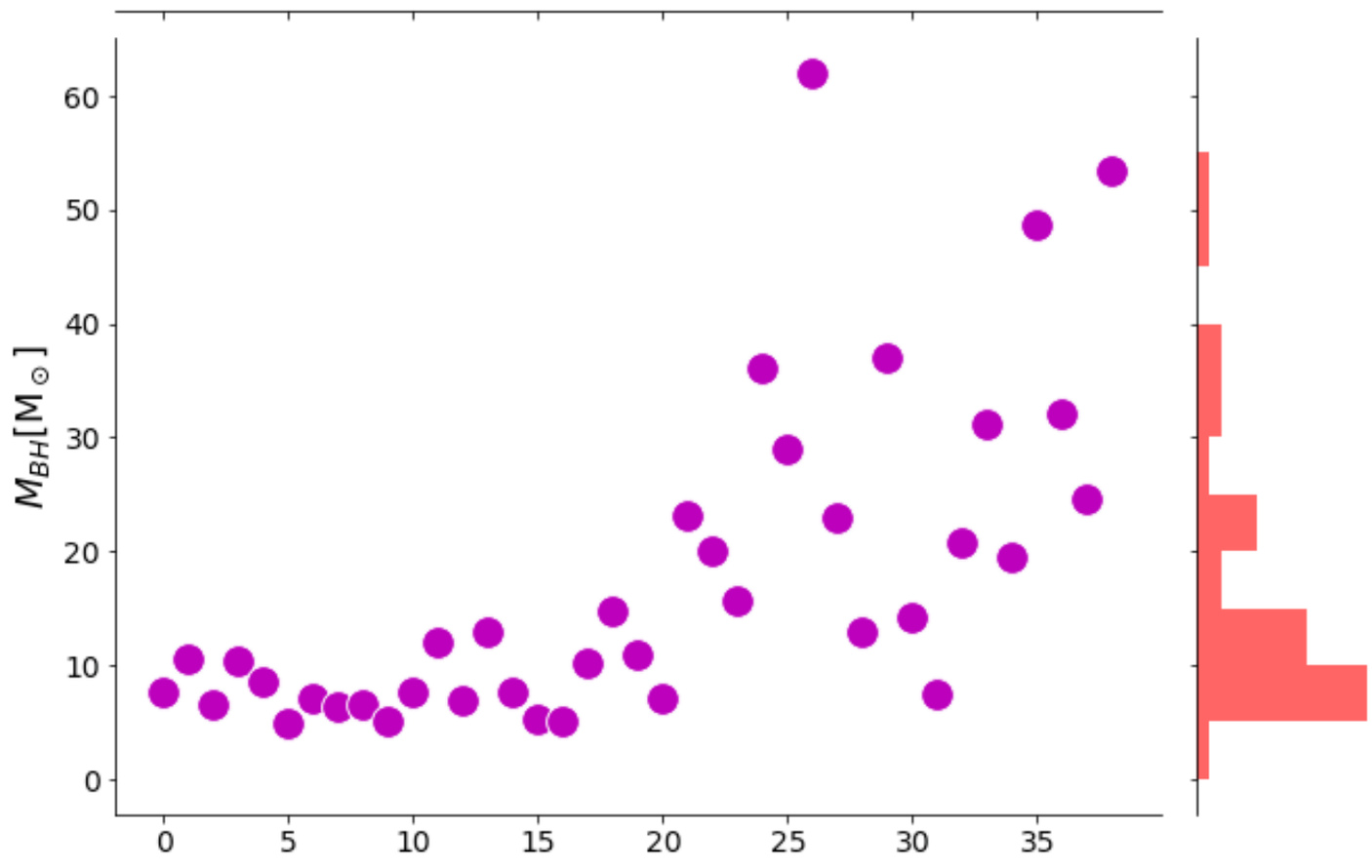
Practical steps

You can do CV completely manually - but I recommend using the sklearn.model_selection package:

```
from sklearn.model_selection import KFold

kf = KFold(n_splits=n_splits)

for train, test in kf.split(X):
        x_train = x[train, :]
        x_test = x[test, :]

<do calculations & calculate score using x_test>
```

An example - estimating the distribution of

This is from problem set



We would like to reconstruct the distribution of black hole masses given these samples.

An example - estimating the distribution of black hole masses

First step, calculate a KDE with some bandwidth, bw, and find its score (quality):

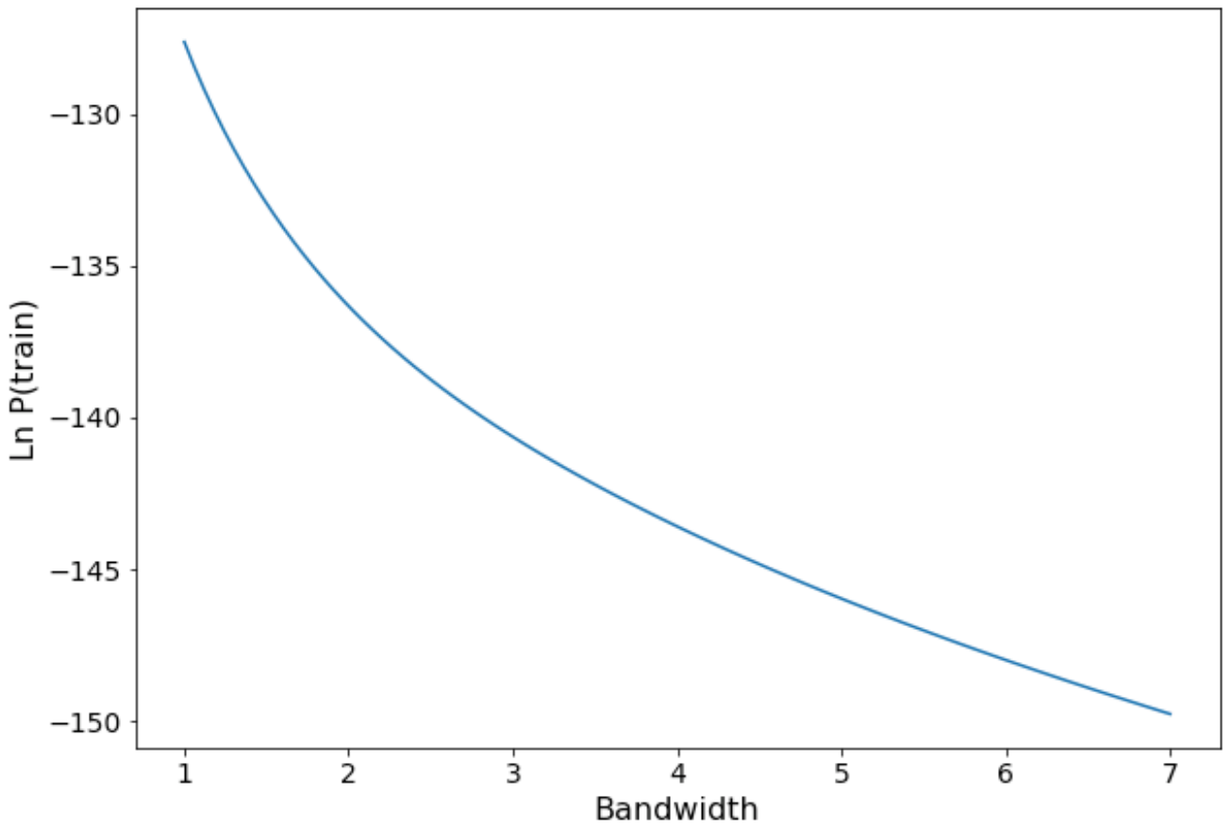
```
X = MBH[:, np.newaxis]
kde = KernelDensity(bandwidth=bw, kernel=kernel).fit(X)
score = kde.score(X)
```

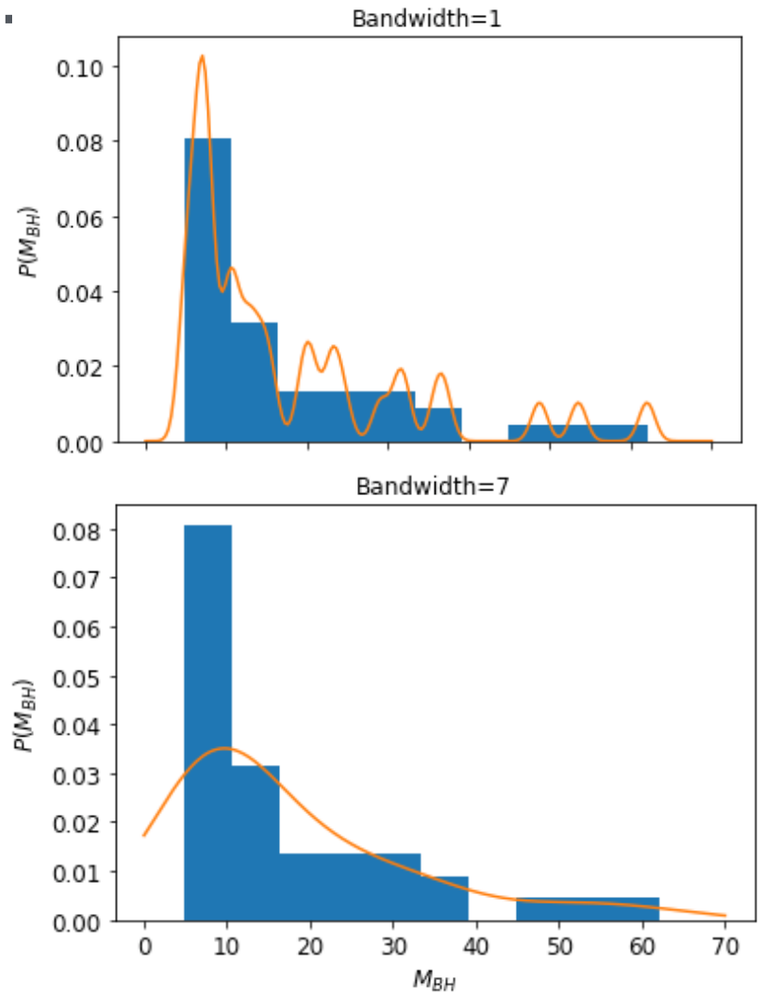
then, put this in a function (score_one_bandwidth) and do many:

```
bws = np.linspace(1, 7.0, 100)
scores = np.zeros_like(bws)
for i, bw in enumerate(bws):
    scores[i] = score_one_bandwidth(MBH, bw, kernel=kernel)
```

An example - estimating the distribution of black hole masses

Evaluated on the training sample we get:





which of course does not tell us much about what would be the best curve.

An example - estimating the distribution of black hole masses

```
# I will do N-fold CV here. This divides X into N_folds
kf = KFold(n_splits=n_splits)
lnP = 0.0

for train, test in kf.split(X):
    x_train = X[train, :]
    x_test = X[test, :]

    kde = KernelDensity(kernel=kernel, bandwidth=bw).fit(x_train)
    log_prob = kde.score(x_test)
    lnP += log_prob
```

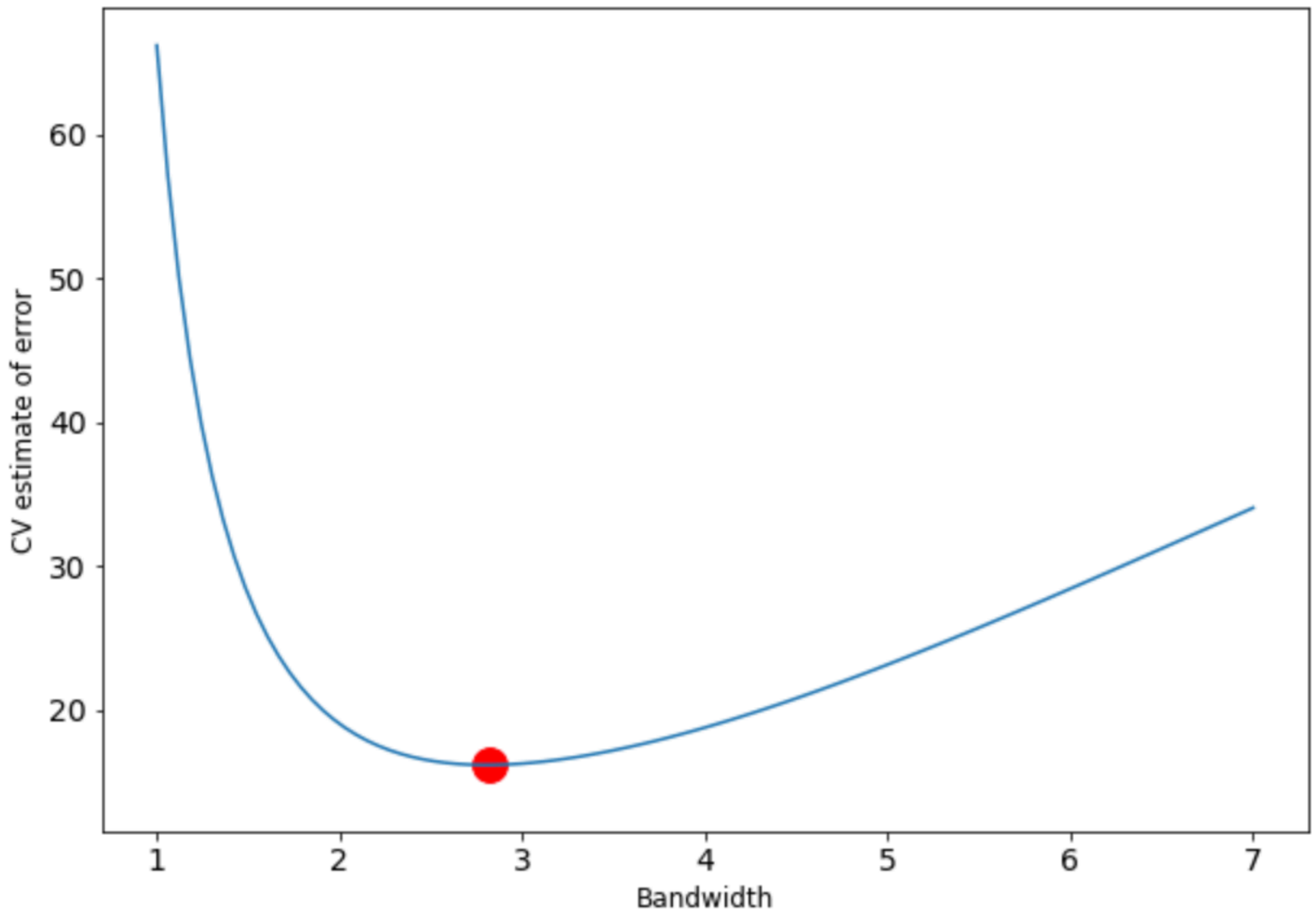
An example - estimating the distribution of black hole masses

```
# I will do N-fold CV here. This divides X into N_folds
kf = KFold(n_splits=n_splits)
lnP = 0.0

for train, test in kf.split(X):
    x_train = X[train, :]
    x_test = X[test, :]
    kde = KernelDensity(kernel=kernel, bandwidth=bw).fit(x_train)
    log_prob = kde.score(x_test)
    lnP += log_prob
```

$$\ln L_{\mathrm{CV}} = \frac{1}{N} \sum_{i=1}^{K} \ln p_{x_i}(z_i)$$

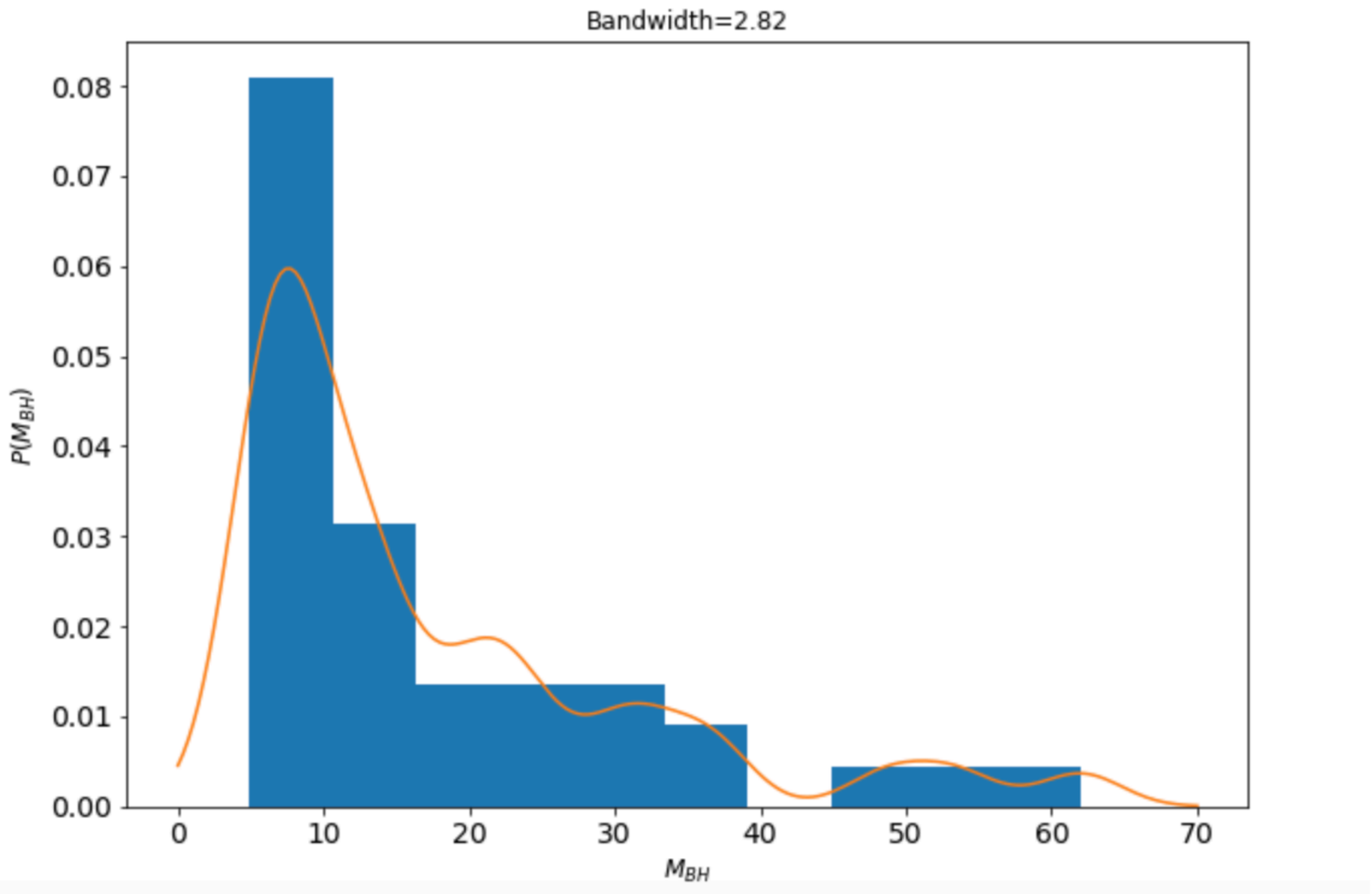
An example - estimating the distribution of black hole masses

The best bandwidth is then the minimum of -ln L_{CV}.



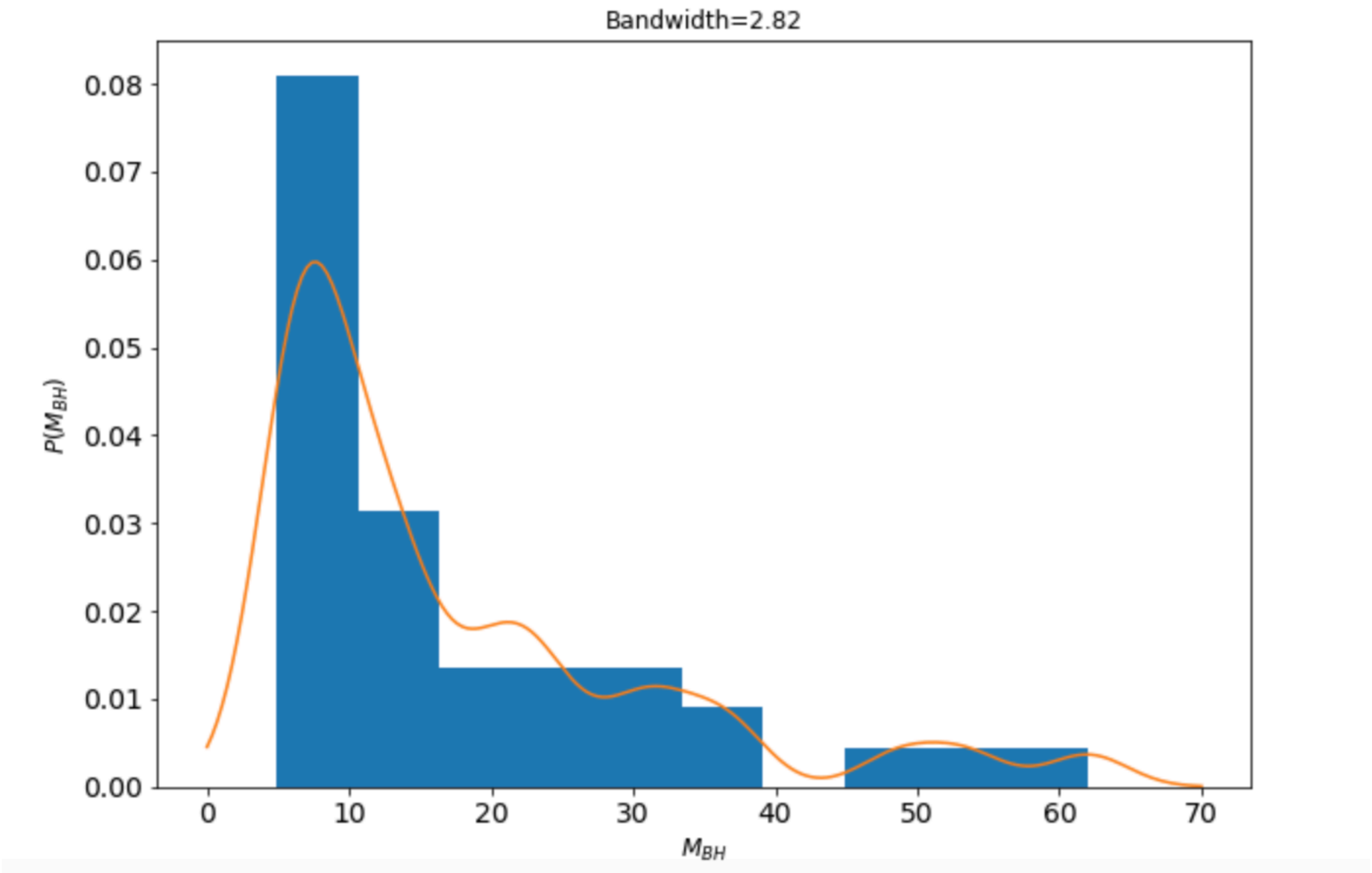
An example - estimating the distribution of black hole masses

The best bandwidth is then the minimum of -ln L_{CV}.



An example - estimating the distribution of black hole masses

The best bandwidth is then the minimum of -ln L_{CV}.



Problems?

Inference

Degree of belief

f(x) - the degree of belief about the value of a quantity

Example:

We observe X photons from a source, as long as the number of photons is large we might say that we observed a flux of $X \pm \sqrt{X}$

What we mean then is that our "degree of belief" about the value of X is given by (in this case):

$$f(x) \propto e^{-(x-\hat{\mu})^2/2\hat{\sigma}^2} = e^{-(x-X)^2/2X}$$

This can be formalised (Jaynes 1998) using probabilities

Statistical context

Assume: $\mathbf{x} \sim p\left(\mathbf{x}; \boldsymbol{\theta}\right)$

If you know $p(\boldsymbol{x};\ )$ you might want to **estimate** its parameters $\boldsymbol{\Theta}$.

Inference/ estimation

If you know $p(\boldsymbol{x};\ \boldsymbol{\Theta})$ you might want to draw random variables from it.

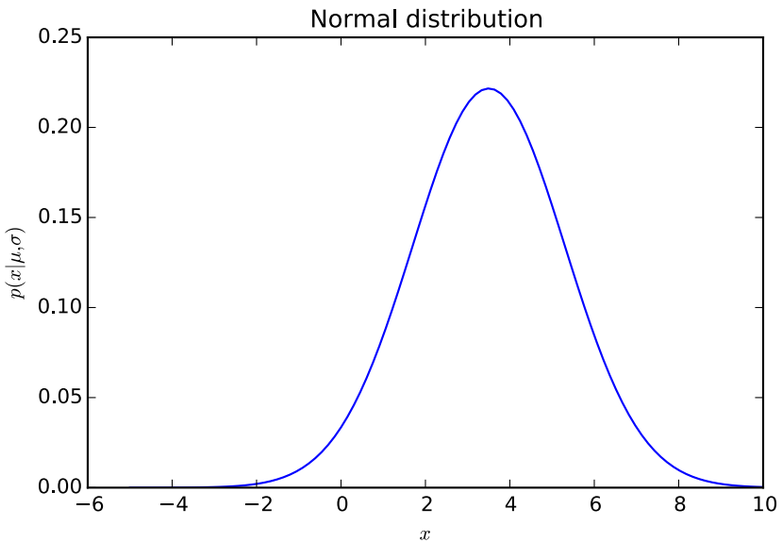
Simulation

If you know $\mathbf{x}$ but not $p(\boldsymbol{x})$, you might want to estimate $p(\boldsymbol{x})$.

Density estimation

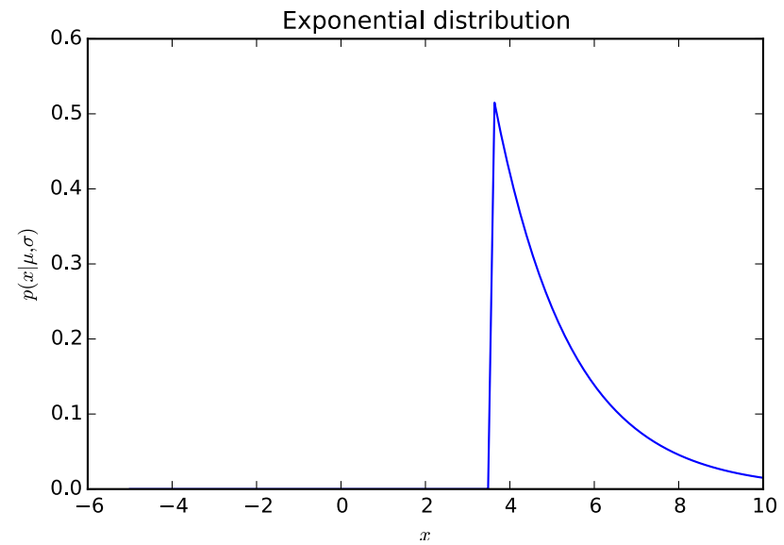
See GitHub for a IPython notebook on distributions.

Summarising distributions/data



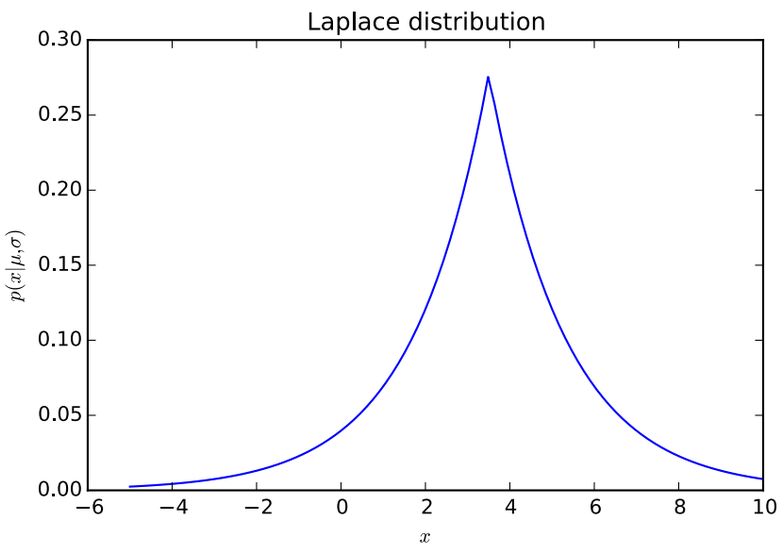
$$p(x|\mu,\sigma) = \frac{1}{\sqrt{2\pi}\sigma} e^{-(x-\mu)^2/2\sigma^2}$$

$$E[x] = \mu \qquad V = E[(x-\mu)^2] = \sigma^2$$



$$p(x|\mu,\sigma) = \frac{1}{2\sigma} e^{-|x-\mu|/\sigma} H(x-\mu)$$

$$E[x] = \mu + \sigma \qquad V = \sigma^2$$



$$p(x|\mu,\sigma) = \frac{1}{2\sigma} e^{-|x-\mu|/\sigma}$$

$$E[x] = \mu \qquad V[x] = 2\sigma^2$$

The Normal distribution

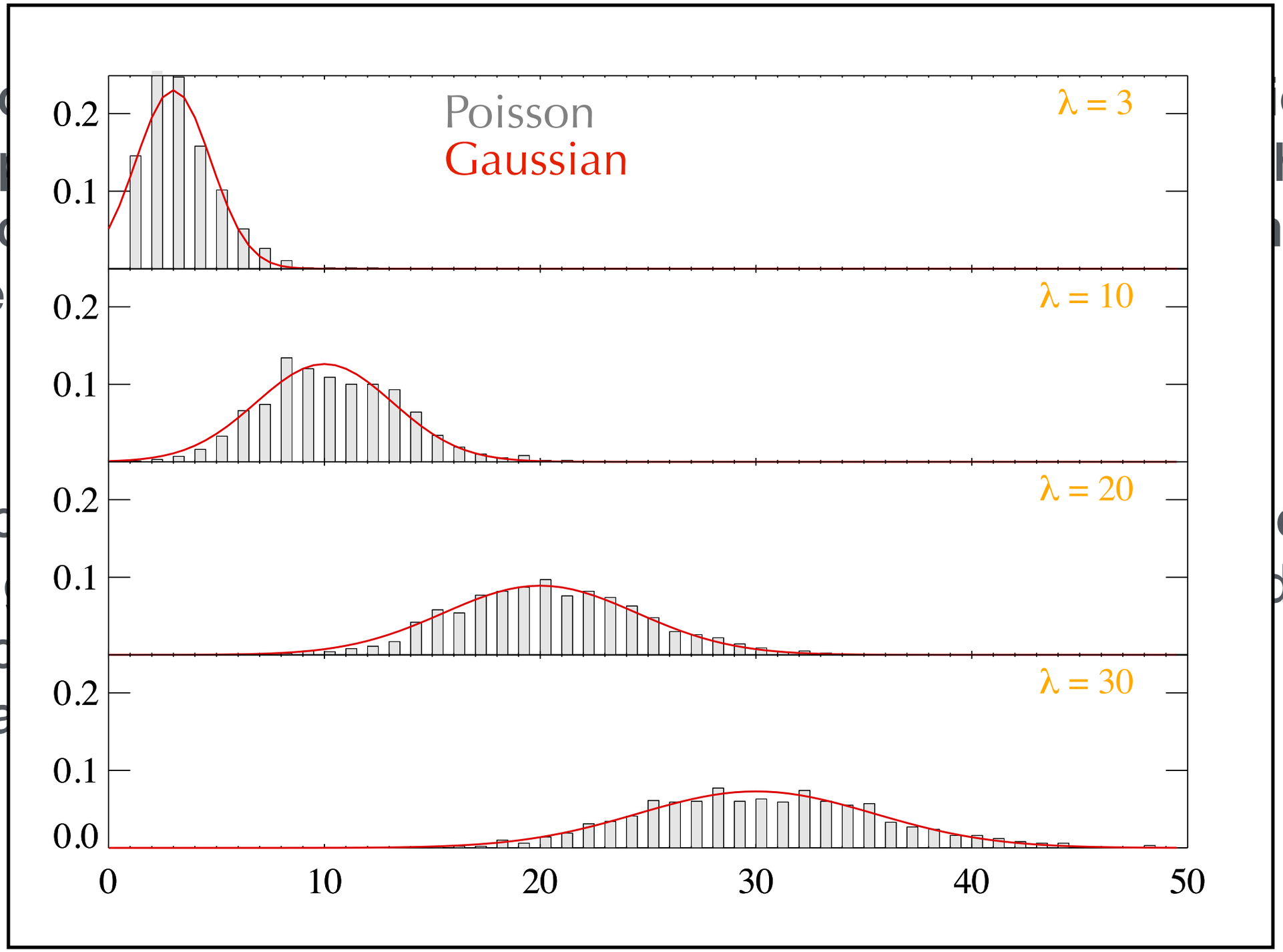
A particularly important distribution is the normal distribution. It is important in data mining because it is easy to deal with analytically and because the mean of large numbers of random numbers is distributed as a normal distribution.

In astronomy, the key reason is that if you observe a source emitting N photons you will detect n, where n is distributed as a Poisson distribution. However as soon as n > 10 (or so), that is essentially a Gaussian distribution.

The Normal distribution

A parti on.
It is imp h
analyti dom
numbe

In astro e
emittin d
as a Po
so), tha



The Central Limit Theorem

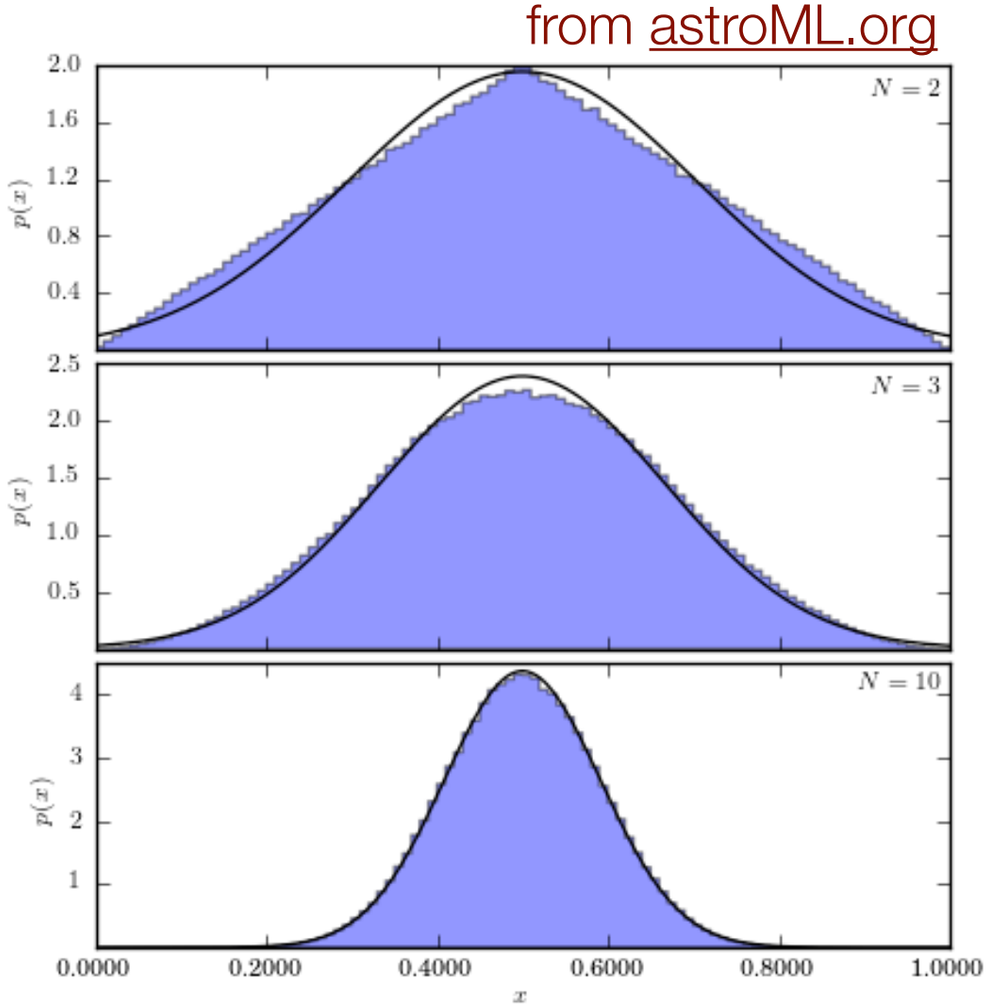
Arbitrary distribution $h(x)$ with mean, μ, and standard deviation σ

Draw M values x from this distribution - the mean of x will follow

$$\bar{x} \sim N(\mu, \sigma/\sqrt{M})$$

Illustration for the mean of N = 2, 3 & 10 values drawn from a uniform distribution.

from astroML.org



The normal distribution - multi-dimensional

$$P(\boldsymbol{x}) = \frac{1}{(2\pi)^{1/D}} \frac{1}{|\Sigma|^{1/2}} e^{-\frac{1}{2}(x-\mu)^T \Sigma^{-1} (x-\mu)}$$

Where Σ is the covariance matrix:

$$\Sigma = E\left[(x-\mu)(x-\mu)^T\right]$$

This is a useful distribution in many cases and underlies a lot of the techniques we will be using.

Drawing from a distribution

This is a common task. Most computer languages provide you with a (more or less useful) random number generator:

R:

```
x = runif(N, [min=0, max=1])
```

```
x = runif(1000, min=-1, max=1)
```

IDL:

```
x = randomu(seed, N)
```

```
x = 2*(randomu(ss, 1000)-0.5)
```

Python (after doing: import numpy as np)

```
x = np.random.uniform(a, b, N)
```

```
x = np.random.uniform(-1, 1, 1000)
```

More complex distributions:

$$r = \mathrm{CDF}^{-1}(x)$$

where x is uniformly distributed

In many-D this is more complex and other algorithms might be used.

A brief intermezzo - storing and caching

Some of the calculations you will do can be time-consuming. When nothing has changed there is no reason to redo the calculation - but of course you need to store the result.

The general programming technique is called memoizing, in case you cared. We will typically want to store the results of calculations to disk.

Pickling

```
try:
    import cPickle
except:
    import _pickle as cPickle

def pickle_to_file(data, fname):
    fh = open(fname, 'w')
    cPickle.dump(data, fh)
    fh.close()

def pickle_from_file(fname):
    fh = open(fname, 'r')
    data = cPickle.load(fh)
    fh.close()
    return data
```

A standard way to store data in Python - can store complex data (but in general use other formats for this).

Decorating

This is a standard Pythonic way to modify functions.

```
def get_text(name):
   return "I ({0}) am a decorator".format(name)

def p_decorate(func):
   def wrapper(name):
      return "*** {0} ***".format(func(name))
   return wrapper

my_get_text = p_decorate(get_text)
```

Decorating

This is a standard Pythonic way to modify functions.

```
def get_text(name):
   return "I ({0}) am a decorator".format(name)

def p_decorate(func):
   def wrapper(name):
       return "*** {0} ***".format(func(name))
   return wrapper

my_get_text = p_decorate(get_text)
```

Usage:

```
In [2]: my_get_text('Jarle')
Out[2]: '*** I (Jarle) am a decorator ***'
```

Decorating

This is a standard Pythonic way to modify functions.

```
def get_text(name):
   return "I ({0}) am a decorator".format(name)

def p_decorate(func):
   def wrapper(name):
       return "*** {0} ***".format(func(name))
   return wrapper

my_get_text = p_decorate(get_text)
```

“Nicer” way (more Pythonic):

```
@p_decorate
def get_text_d(name):
   return "I ({0}) am a decorator".format(name)
```

Why should you care?

astroML has a very convenient way to store results of heavy calculations:

```
from astroML.decorators import pickle_results

@pickle_results("GMM_three_Gaussians.pkl")
def compute_GMM(data, N, n_iter=1000):
    # Setup the output array
    models = [None for n in N]
    for i in range(len(N)):
        models[i] = GMM(n_components=N[i], n_iter=n_iter,
                        covariance_type=covariance_type)
        models[i].fit(data)

    return models
```

When the function is called with the same arguments - the results are read from the file & not redone!.

Classical inference

The broad idea

We want to compare data, $\{y_i\}$, to a model M(a,b,…)

To do so we need:

The broad idea

We want to compare data, $\{y_i\}$, to a model M(a,b,…)

To do so we need:

A way to compare our data to the model (cost functions)

e.g.: $|f(x) - f_{\mathrm{true}(x)}|$ $\int \left(f(x) - f_{\mathrm{true}(x)}\right)^2 dx$

The broad idea

We want to compare data, $\{y_i\}$, to a model M(a,b,…)

To do so we need:

A way to compare our data to the model (cost functions)

e.g.: $|f(x) - f_{\mathrm{true}(x)}|$ $\int \left(f(x) - f_{\mathrm{true}(x)}\right)^2 dx$

A way to rank/compare different models

Likelihood ratios, information criteria (AIC) etc.

The broad idea

We want to compare data, $\{y_i\}$, to a model M(a,b,…)

To do so we need:

A way to compare our data to the model (cost functions)

e.g.: $|f(x) - f_{\mathrm{true}(x)}|$ $\int \left(f(x) - f_{\mathrm{true}(x)}\right)^2 dx$

A way to rank/compare different models

Likelihood ratios, information criteria (AIC) etc.

A way to assess whether the fit we obtained is “good”

e.g. χ^2 tests

The broad idea

We want to compare data, $\{y_i\}$, to a model M(a,b,…)

To do so we need:

A way to compare our data to the model (cost functions)

e.g.: $|f(x) - f_{\mathrm{true}(x)}|$ $\int \left(f(x) - f_{\mathrm{true}(x)}\right)^2 dx$

A way to rank/compare different models

Likelihood ratios, information criteria (AIC) etc.

A way to assess whether the fit we obtained is “good”

e.g. χ^2 tests

This is arguably the approach most widely used in Machine Learning/Data Mining.

Maximum Likelihood

How were the data generated?

1. Define a likelihood of the data given a model $p(D|M)$

I will sometimes write the parameters of the model $\boldsymbol{\theta}$ and the model $M(\boldsymbol{\theta})$

Maximum Likelihood

How were the data generated?

1. Define a likelihood of the data given a model $p(D|M)$

I will (sometimes) write the parameters of the model $\boldsymbol{\theta}$ and the model $M(\boldsymbol{\theta})$

2. Find $\boldsymbol{\theta} = \boldsymbol{\theta}^0$ that maximise $p(D|M)$ **point estimates**

For this we use a minimization routine (e.g. scipy.optimize)

Maximum Likelihood

How were the data generated?

1. Define a likelihood of the data given a model $p(D|M)$

   I will sometimes write the parameters of the model $\boldsymbol{\theta}$ and the model $M(\boldsymbol{\theta})$

2. Find $\boldsymbol{\theta} = \boldsymbol{\theta}^0$ that maximise $p(D|M)$ **point estimates**

   For this we use a minimization routine (e.g. scipy.optimize)

3. Find confidence levels for $\boldsymbol{\theta}^0$

$$\sigma^2_{j,k} = - \left. \frac{d^2 \ln L}{d\theta_j d\theta_k} \right|_{\theta=\theta_0}$$

ML - Simple example - Gaussian errors

$\boldsymbol{\theta} = (\alpha, \beta)$ $\epsilon_i \sim N(0, \sigma_i^2)$

Model y as $y_i = \alpha + \beta x_i + \epsilon_i$

Likelihood of data:

$$P(x_i, y_i, \sigma_i | \alpha, \beta) = \frac{1}{\sqrt{2\pi}\sigma_i} e^{-[y_i - (\alpha + \beta x_i)]^2 / 2\sigma_i^2}$$

ML - Simple example - Gaussian errors

$\boldsymbol{\theta} = (\alpha, \beta)$ $\epsilon_i \sim N(0, \sigma_i^2)$

Model y as $y_i = \alpha + \beta x_i + \epsilon_i$

Likelihood of data:

$$P(x_i, y_i, \sigma_i | \alpha, \beta) = \frac{1}{\sqrt{2\pi}\sigma_i} e^{-[y_i - (\alpha + \beta x_i)]^2 / 2\sigma_i^2}$$

Full Likelihood $L(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{\sigma} | \alpha, \beta) = \prod_i P(x_i, y_i, \sigma_i | \alpha, \beta)$

ML - Simple example - Gaussian errors

$\boldsymbol{\theta} = (\alpha, \beta)$ $\epsilon_i \sim N(0, \sigma_i^2)$

Model y as $y_i = \alpha + \beta x_i + \epsilon_i$

Likelihood of data:

$$P(x_i, y_i, \sigma_i | \alpha, \beta) = \frac{1}{\sqrt{2\pi}\sigma_i} e^{-[y_i - (\alpha + \beta x_i)]^2 / 2\sigma_i^2}$$

Full Likelihood $L(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{\sigma} | \alpha, \beta) = \prod_i P(x_i, y_i, \sigma_i | \alpha, \beta)$

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

ML - Simple example - Gaussian errors

$\boldsymbol{\theta} = (\alpha, \beta)$ $\epsilon_i \sim N(0, \sigma_i^2)$

Model y as $y_i = \alpha + \beta x_i + \epsilon_i$

Likelihood of data:

$$P(x_i, y_i, \sigma_i | \alpha, \beta) = \frac{1}{\sqrt{2\pi}\sigma_i} e^{-[y_i - (\alpha + \beta x_i)]^2 / 2\sigma_i^2}$$

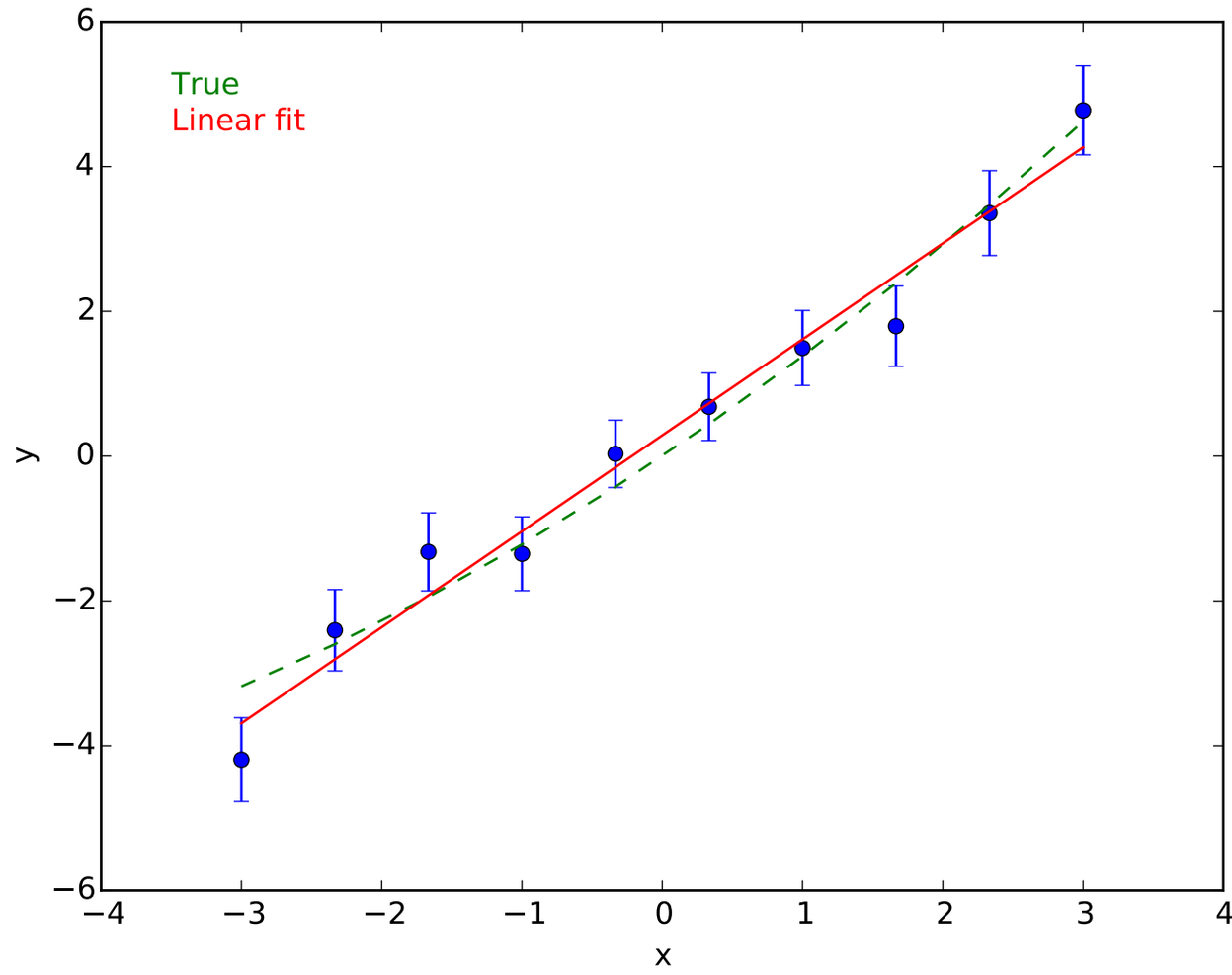
Full Likelihood $L(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{\sigma} | \alpha, \beta) = \prod_i P(x_i, y_i, \sigma_i | \alpha, \beta)$

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

Minimisation of this is the same as least squares minimisation.

ML - Simple example - Gaussian errors, standard function

```
from scipy.optimize import curve_fit

def func_line(x, a, b):
    return a + b*x

pars, cov = curve_fit(func_line, x, y_obs)
```



An alternative approach with more flexibility in the regression function:

```
from astroML.linear_model import LinearRegression
m = LinearRegression()
m.fit(x[:, None], y_obs, sigma)
a, b = m.coef_
```

ML - Simple example - explicit likelihood

```
def negInL(theta, x, y, yerr):
    a, b = theta
    model = b * x + a
    inv_sigma2 = 1.0/(yerr**2)

    return 0.5*(np.sum((y-model)**2*inv_sigma2))
```

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

ML - Simple example - explicit likelihood

```
def negInL(theta, x, y, yerr):
    a, b = theta
    model = b * x + a
    inv_sigma2 = 1.0/(yerr**2)

    return 0.5*(np.sum((y-model)**2*inv_sigma2))
```

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

```
import scipy.optimize as op
result = op.minimize(negInL, [1.0, 0.0], args=(x, y_obs, sigma))
a_ml, b_ml = result["x"]
```

ML - Simple example - explicit likelihood

```
def negInL(theta, x, y, yerr):
    a, b = theta
    model = b * x + a
    inv_sigma2 = 1.0/(yerr**2)

    return 0.5*(np.sum((y-model)**2*inv_sigma2))
```

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

```
import scipy.optimize as op
result = op.minimize(negInL, [1.0, 0.0], args=(x, y_obs, sigma))
a_ml, b_ml = result["x"]
```

More work in this case - but a much more general approach.

Bayesian inference

Bayesian vs Frequentist

You may have heard about this controversy - I'm not going to spend much time on that here.

In practice the difference can perhaps boil down to differences in uncertainty statements:

Bayesian: Given our data we have a 95% probability that the true value of **θ** lies within the credible region.

Frequentist: If we repeat the experiment man times, in 95% of the cases the confidence interval will contain the true value of **θ**.

Based on Vanderplas (2014, https://arxiv.org/abs/1411.5018) see that for a readable overview

Bayesian statistics

This allows us to calculate the likelihood distribution of parameters

$$p(\mathrm{Model}|\mathrm{Data}) = p(M|D) = \frac{p(D|M)p(M)}{p(D)}$$

The key part here is that we need to specify what **prior** information we have on the model parameters.

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

The overall plan of attack

- Define the likelihood - the best is a generative one that mimic how you think the data were created. $p(D|\boldsymbol{\theta}, M, I)$
- Decide on the prior - ie. what range of model parameters are likely. $p(\boldsymbol{\theta}, M|I)$
- Use Bayes' theorem to calculate the posterior likelihood distribution. $p(M|D, I)$
- To calculate the posterior distribution we often use Markov Chain Monte Carlo calculations. MCMC

The result of the calculation can be summarised - the maximum of $p(M|D,I)$ gives the maximum a posteriori (**MAP**) estimate - means, medians are also reasonable options.

A practical example - line fitting - model

Generative model: $y_i = \alpha + \beta x_i + \epsilon_i$ $\epsilon_i \sim N(0, \sigma_i^2)$

A practical example - line fitting - model

Generative model: $y_i = \alpha + \beta x_i + \epsilon_i \qquad \epsilon_i \sim N(0, \sigma_i^2)$

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

(note that since σ_i are known, the first two terms are constants and can be ignored for maximisation

```
def lnL(theta, x, y, yerr):
    a, b = theta
    model = b * x + a
    inv_sigma2 = 1.0/(yerr**2)

    return -0.5*(np.sum((y-model)**2*inv_sigma2))
```

A practical example - line fitting - prior

Prior:

$$p(a, b, M|I) = \begin{cases} \text{const}, & \text{if } a \in [-5, 5] \text{ and } b \in [-10, 10]; \\ 0 & \text{otherwise} \end{cases}$$

```
def lnprior(theta):
    a, b = theta
    if -5.0 < a < 5.0 and -10.0 < b < 10.0:
        return 0.0
    return -np.inf
```

(the -np.inf is because p=0 means ln p = -∞)

A practical example - line fitting - posterior

Putting it together: $p(D|a, b, M, I)p(a, b, M|I)$

```
def lnprob(theta, x, y, yerr):
    """
    The likelihood to include in the MCMC.
    """
    lp = lnprior(theta)
    if not np.isfinite(lp):
        return -np.inf
    return lp + lnL(theta, x, y, yerr)
```

Note that I ignore the normalisation $p(D|I)$

A practical example - line fitting

```
import emcee
```

A practical example - line fitting

```
import emcee
```

```
# Use ML to get a starting point.
# result = run_ml()
p_init = np.array([ 0.28233725,  1.31299656])
```

(result[“x”])

A practical example - line fitting

```
import emcee
```

```
# Use ML to get a starting point.
# result = run_ml()
p_init = np.array([ 0.28233725,  1.31299656])
```

(result[“x”])

```
# Set up the properties of the problem.
ndim, nwalkers = 2, 100
```

A practical example - line fitting

```
import emcee
```

```
# Use ML to get a starting point.
# result = run_ml()
p_init = np.array([ 0.28233725,  1.31299656])
```

(result[“x”])

```
# Set up the properties of the problem.
ndim, nwalkers = 2, 100
```

```
# Setup a number of initial positions.
pos = [p_init + 1e-4*np.random.randn(ndim) for i in range(nwalkers)]
```

A practical example - line fitting

```
import emcee
```

```
# Use ML to get a starting point.
# result = run_ml()
p_init = np.array([ 0.28233725,  1.31299656])
```

(result[“x”])

```
# Set up the properties of the problem.
ndim, nwalkers = 2, 100
```

```
# Setup a number of initial positions.
pos = [p_init + 1e-4*np.random.randn(ndim) for i in range(nwalkers)]
```

```
# Create the sampler.
sampler = emcee.EnsembleSampler(nwalkers, ndim, lnprob,
                                args=(x, y_obs, sigma))
```

A practical example - line fitting

```
import emcee
```

```
# Use ML to get a starting point.
# result = run_ml()
p_init = np.array([ 0.28233725,  1.31299656])
```

(result[“x”])

```
# Set up the properties of the problem.
ndim, nwalkers = 2, 100
```

```
# Setup a number of initial positions.
pos = [p_init + 1e-4*np.random.randn(ndim) for i in range(nwalkers)]
```

```
# Create the sampler.
sampler = emcee.EnsembleSampler(nwalkers, ndim, lnprob,
                                args=(x, y_obs, sigma))
```

```
# Run the process.
sampler.run_mcmc(pos, 500)
```

A practical example - importing the package

```
import emcee
```

We will use the MCMC Hammer library (emcee) - http://dan.iel.fm/emcee/current/

it is a pure Python implementation (as pyMC), flexible and reasonably fast - for challenging problems other packages (MultiNest, BUGS, JAGS, STAN) might be better choices but it is well worth starting with this as the learning curve is less steep.

You can install it with:

```
pip install emcee
pip install corner
```

A practical example - starting position

```
# Use ML to get a starting point.
# result = run_ml()
p_init = np.array([ 0.28233725,  1.31299656])
```

(result[“x”])

The MCMC process will randomly (but cleverly) step around in your parameter space. To do this well you need a good starting position. A maximum-likelihood solution gives a good starting point.

I used scipy.optimize.minimize here. I set the result to a variable p_init.

A practical example - line fitting

```
# Set up the properties of the problem.
ndim, nwalkers = 2, 100
```

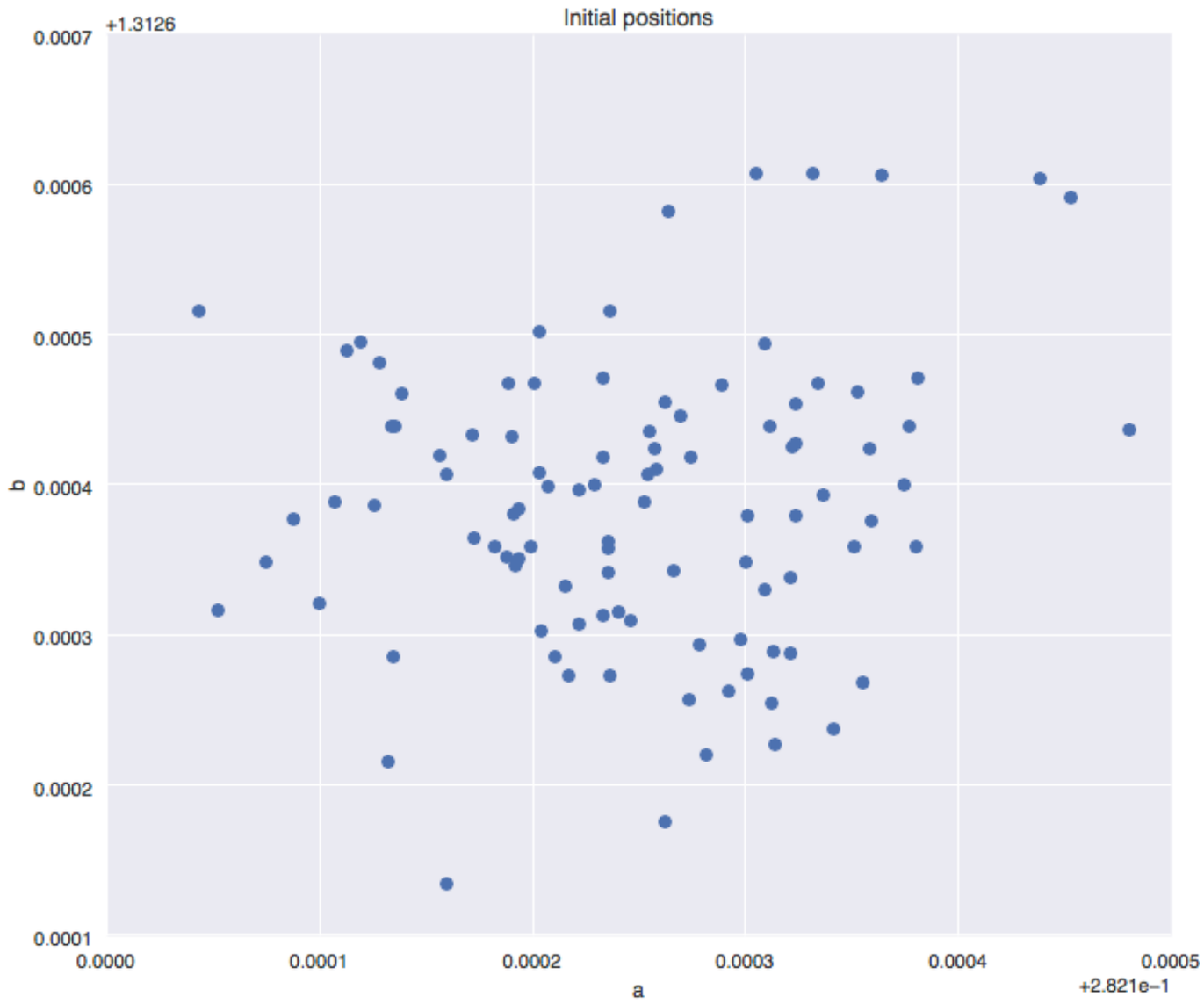
We have two parameters - so the dimensionality is 2

And we also need “walkers” - view these as random processes that explore your parameter space along different paths. The emcee documentation recommends using as many as you can get away with.

A practical example - line fitting

```
# Setup a number of initial positions.
pos = [p_init + 1e-4*np.random.randn(ndim) for i in range(nwalkers)]
```

This creates a set of different starting positions - one for each walker.



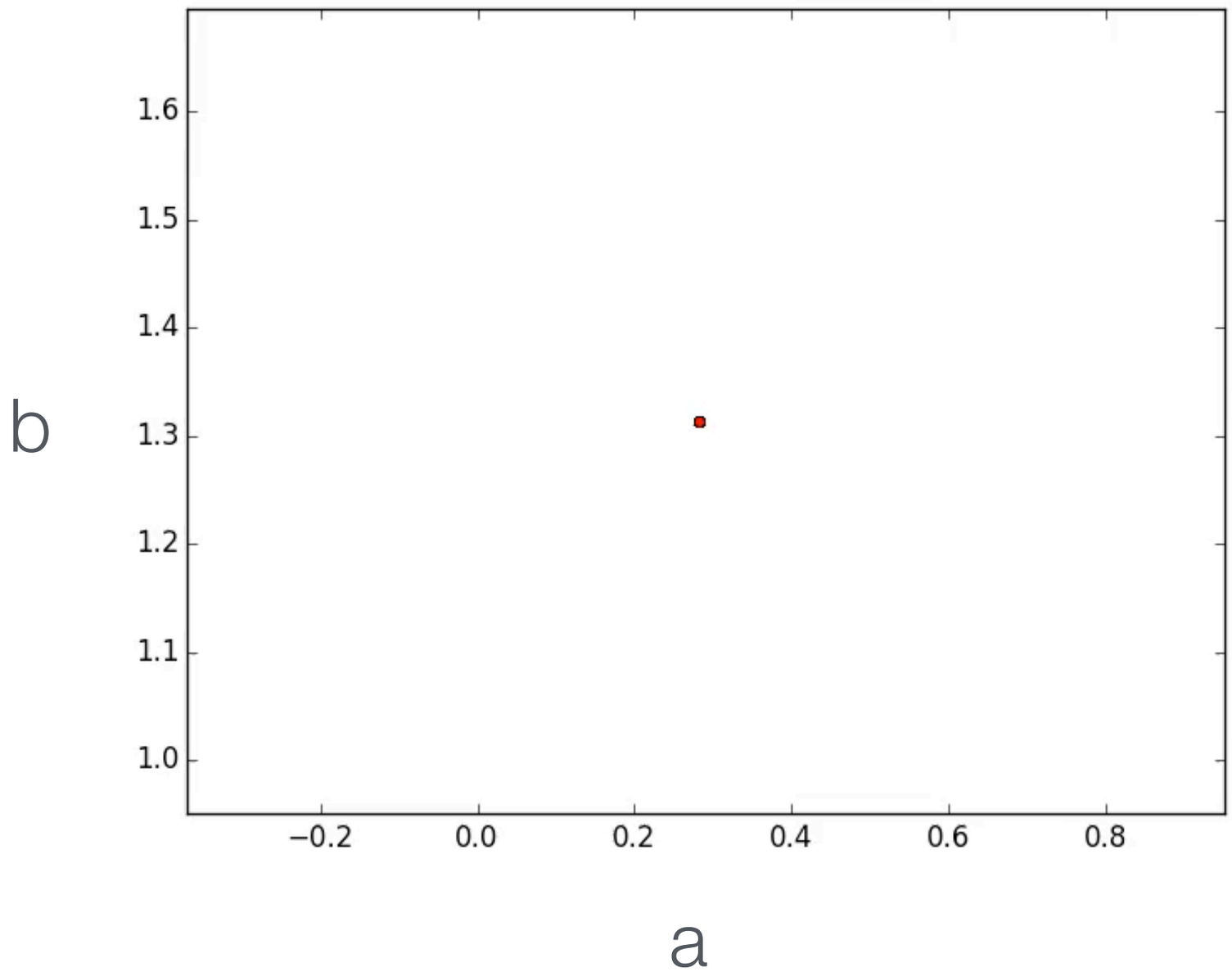
A practical example - line fitting

```
# Create the sampler.
sampler = emcee.EnsembleSampler(nwalkers, ndim, lnprob,
                                args=(x, y_obs, sigma))
```

This sets up the MCMC process - we give the function lnprob and the data as a tuple.

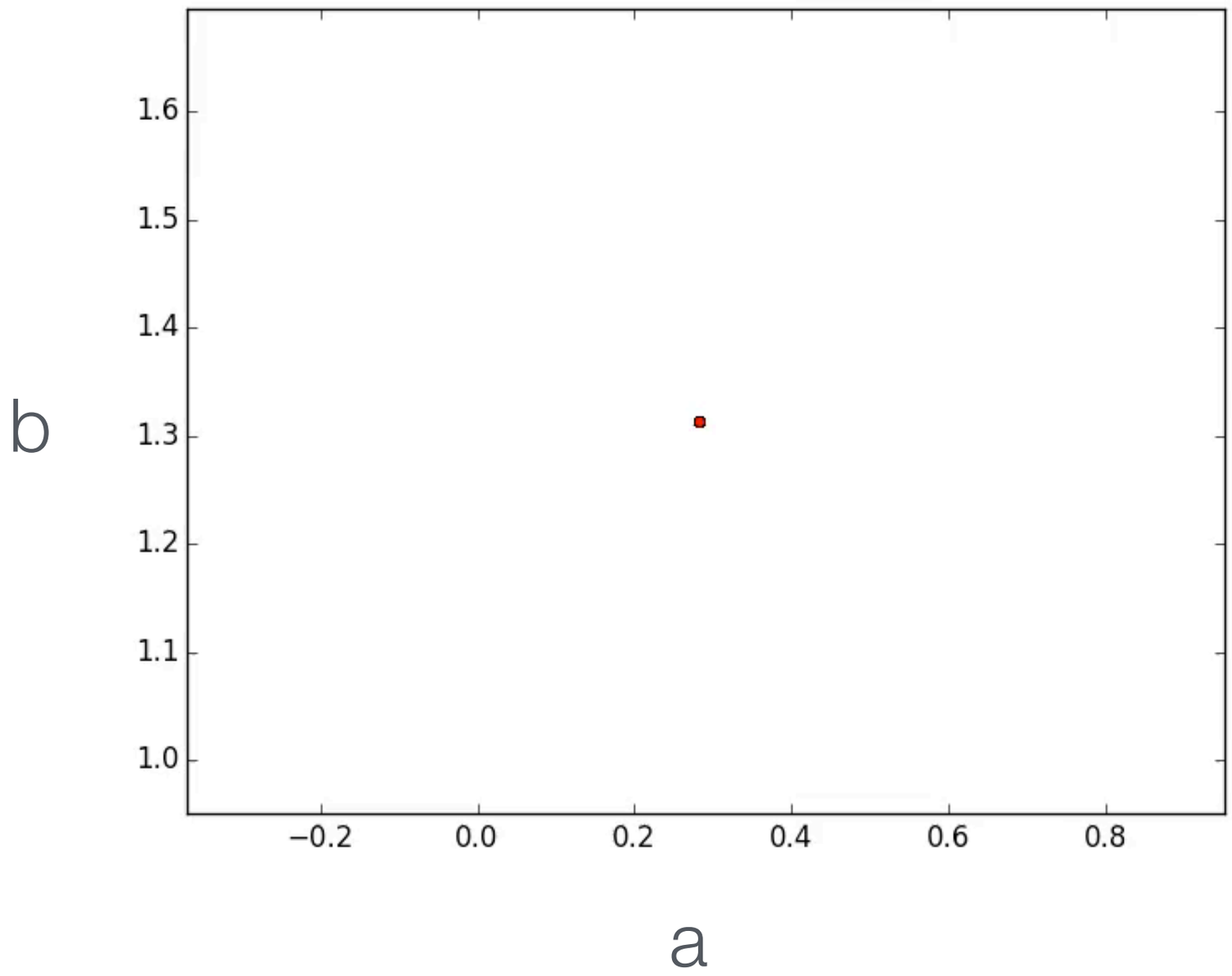
A practical example - line fitting

```
# Run the process.
sampler.run_mcmc(pos, 500)
```



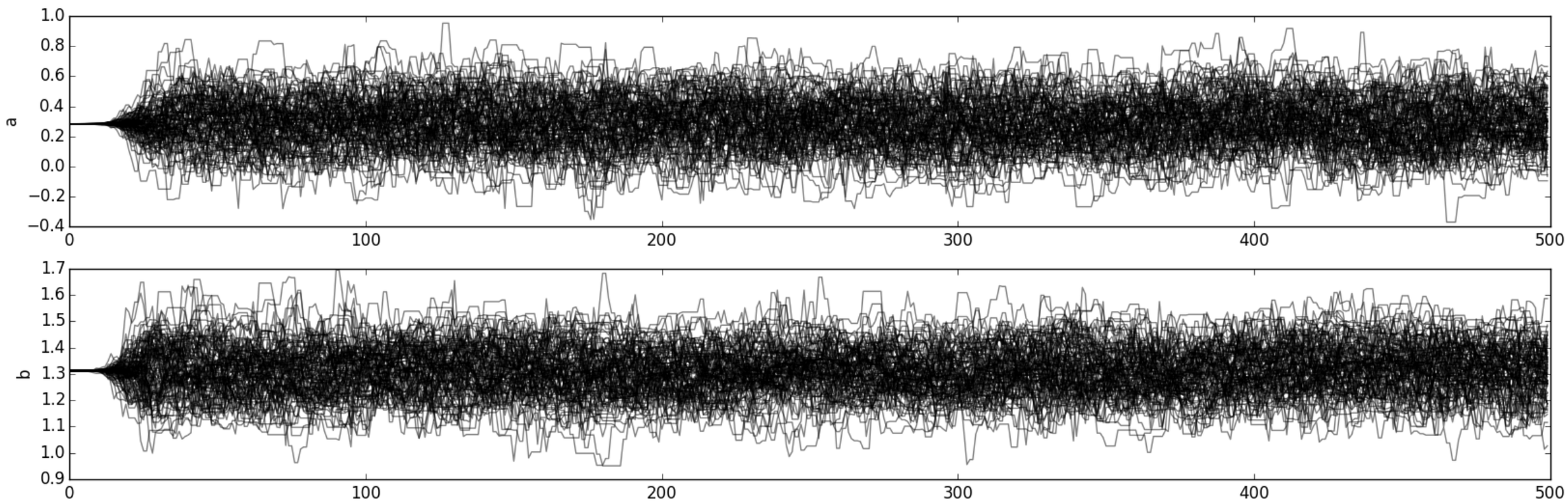
A practical example - line fitting

```
# Run the process.
sampler.run_mcmc(pos, 500)
```



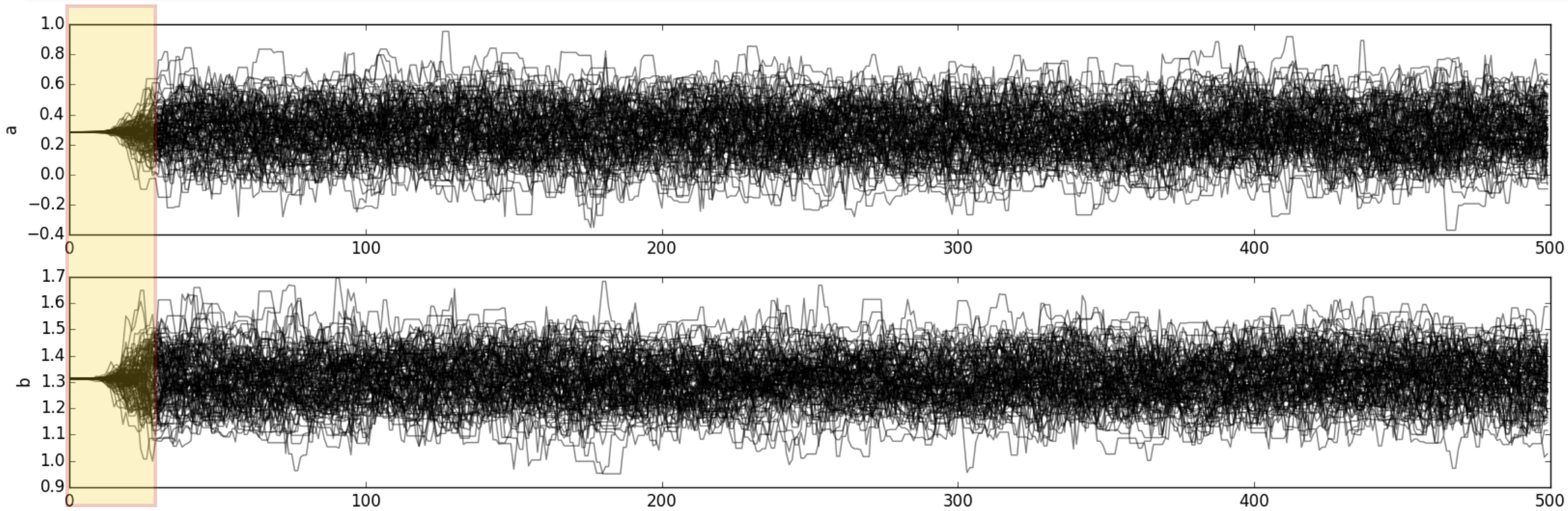
Look at chain behaviour

```
labels = [‘a’, ‘b’]
chain = sampler.chain
for i_dim in range(2):
    plt.subplot(2,1,i_dim+1)
    plt.ylabel(labels[i_dim])

    for i in range(100):
        plt.plot(chain[i,:,i_dim],color='black', alpha=0.5)
```



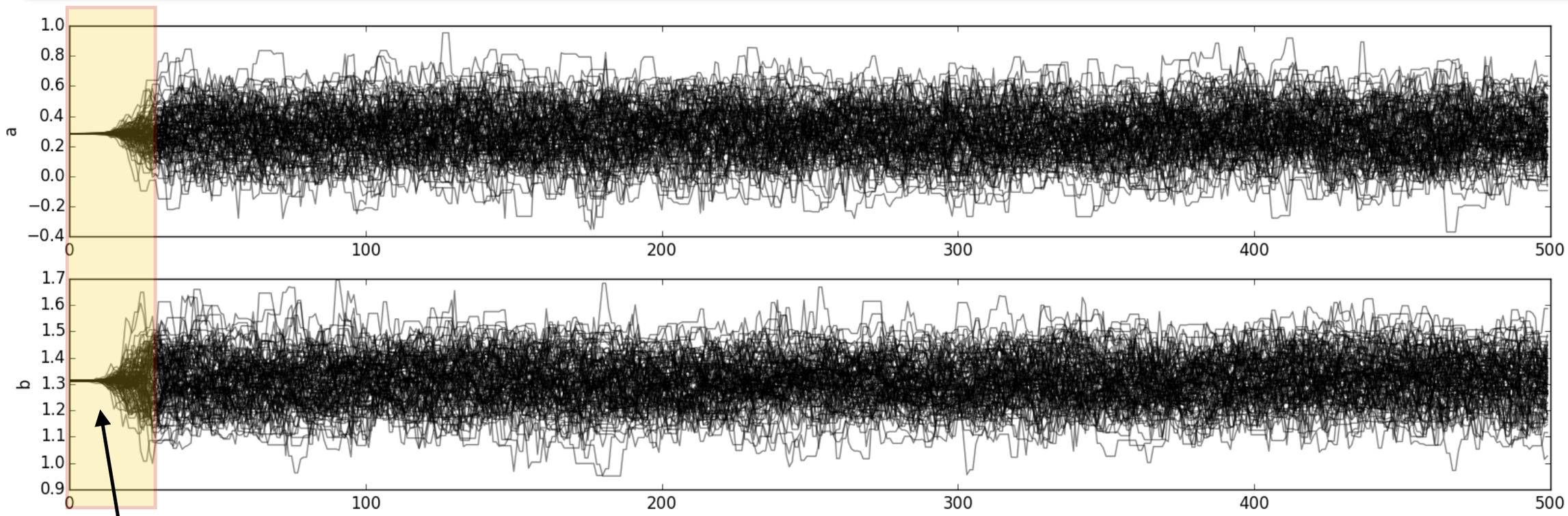
Look at chain behaviour

```
labels = [‘a’, ‘b’]
chain = sampler.chain
for i_dim in range(2):
    plt.subplot(2,1,i_dim+1)
    plt.ylabel(labels[i_dim])

    for i in range(100):
        plt.plot(chain[i,:,i_dim],color='black', alpha=0.5)
```



Look at chain behaviour

```
labels = [‘a’, ‘b’]
chain = sampler.chain
for i_dim in range(2):
    plt.subplot(2,1,i_dim+1)
    plt.ylabel(labels[i_dim])

    for i in range(100):
        plt.plot(chain[i,:,i_dim],color='black', alpha=0.5)
```



Burn-in - we need to wait until this has happened before we start to use the samples.

A practical example - showing the result

Let us extract the samples from 50 onwards and collapse the different walkers

```
samples = sampler.chain[:, 50:, :].reshape((-1, 2))
```

We then show the result using the corner package:

```
import corner

fig = corner.corner(samples, labels=["$a$", “$b$"],
                    truths=[0.0, 1.3], quantiles=[0.16, 0.84])
fig.show()
```

A

L

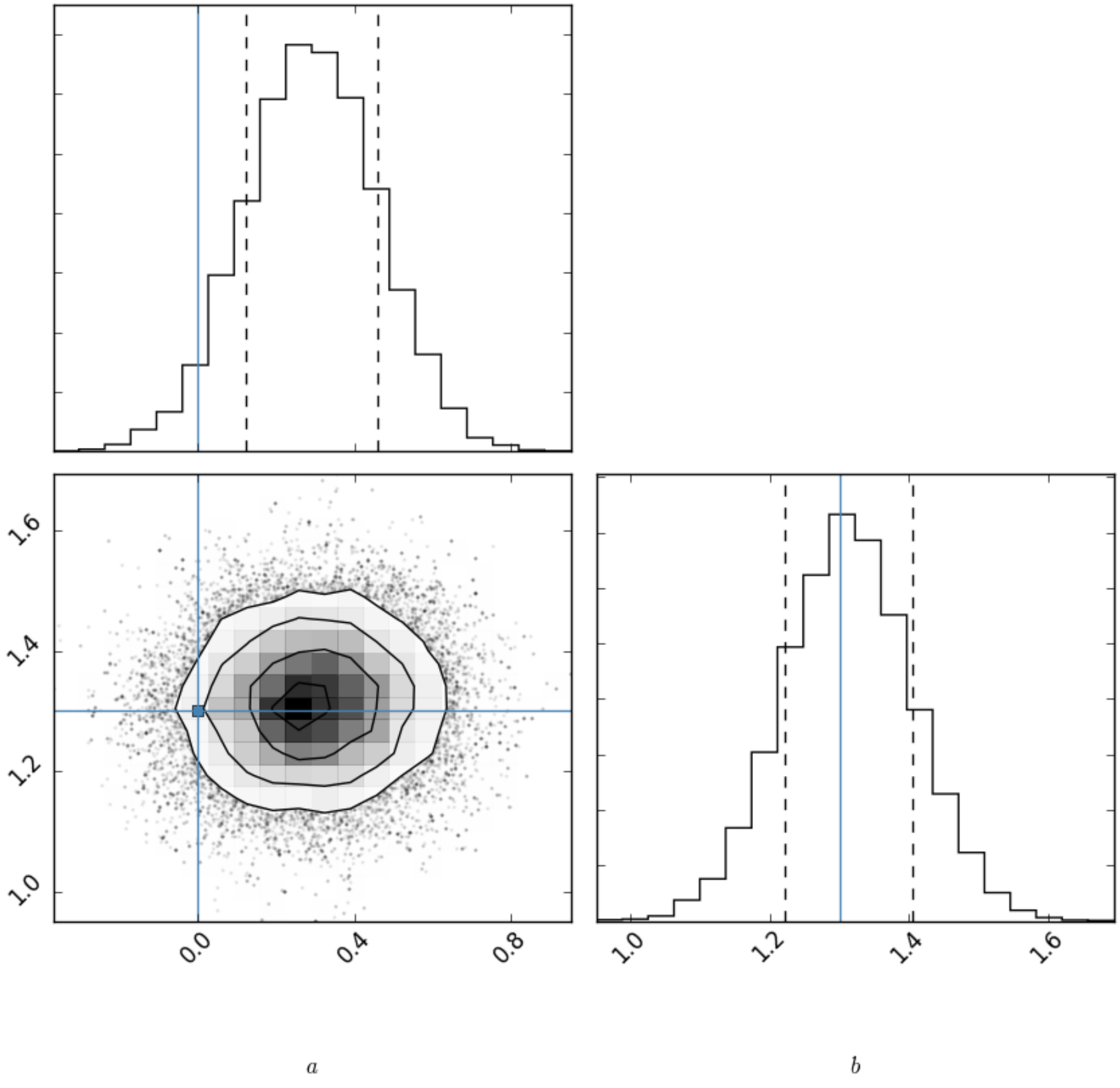
s

V

ir

fi

fi



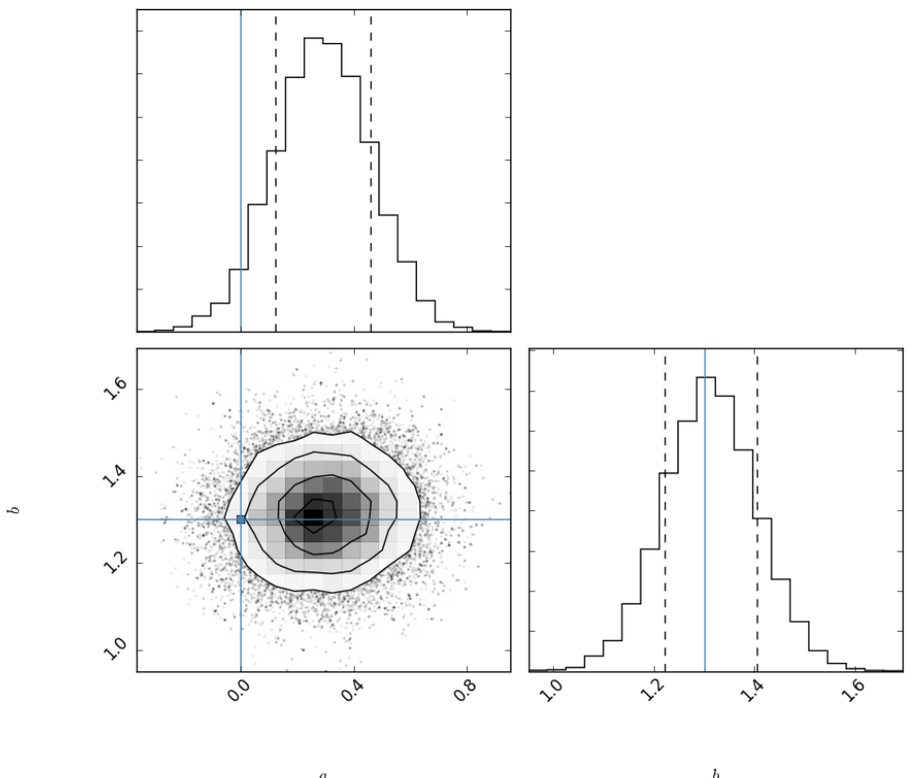
A practical example - showing the result

Let us extract the samples from 50 onwards and collapse the different walkers

```
samples = sampler.chain[:, 50:, :].reshape((-1, 2))
```

We then show the result using the corner package:

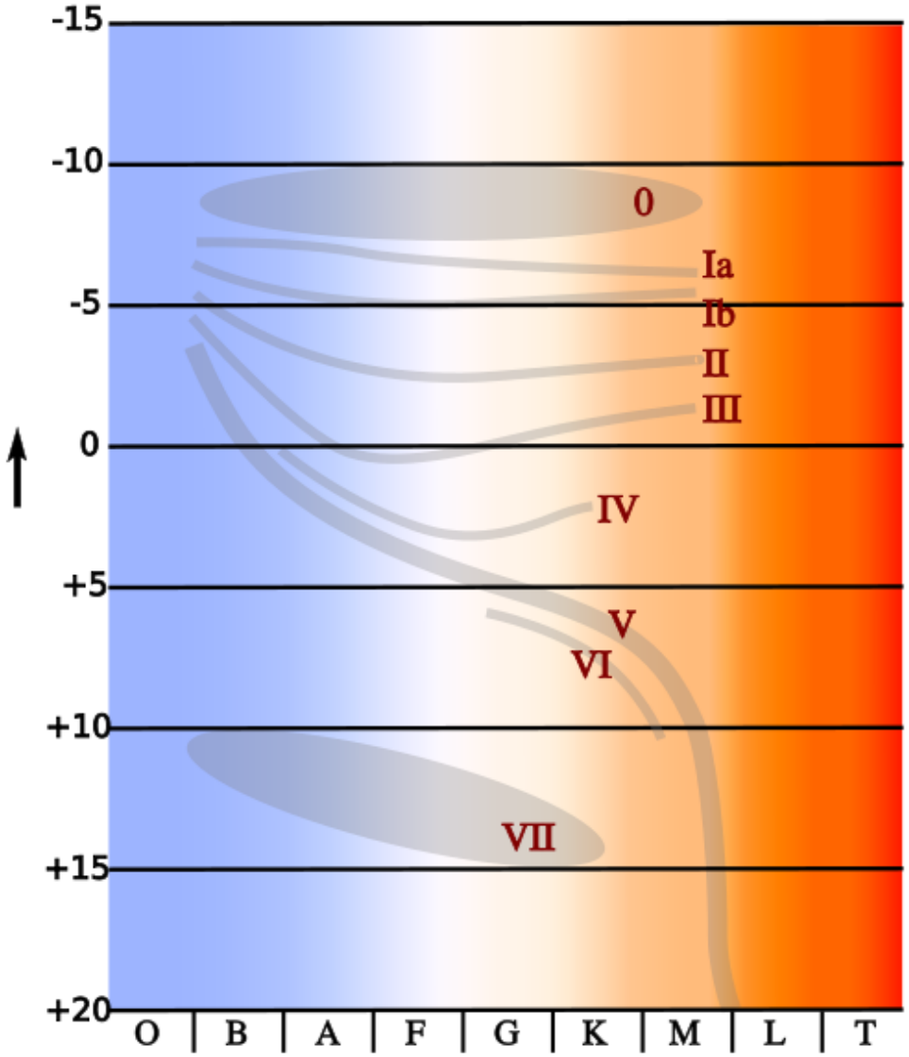
```
import corner

fig = corner.corner(samples, labels=["$a$", “$b$"],
                    truths=[0.0, 1.3], quantiles=[0.16, 0.84])
fig.show()
```



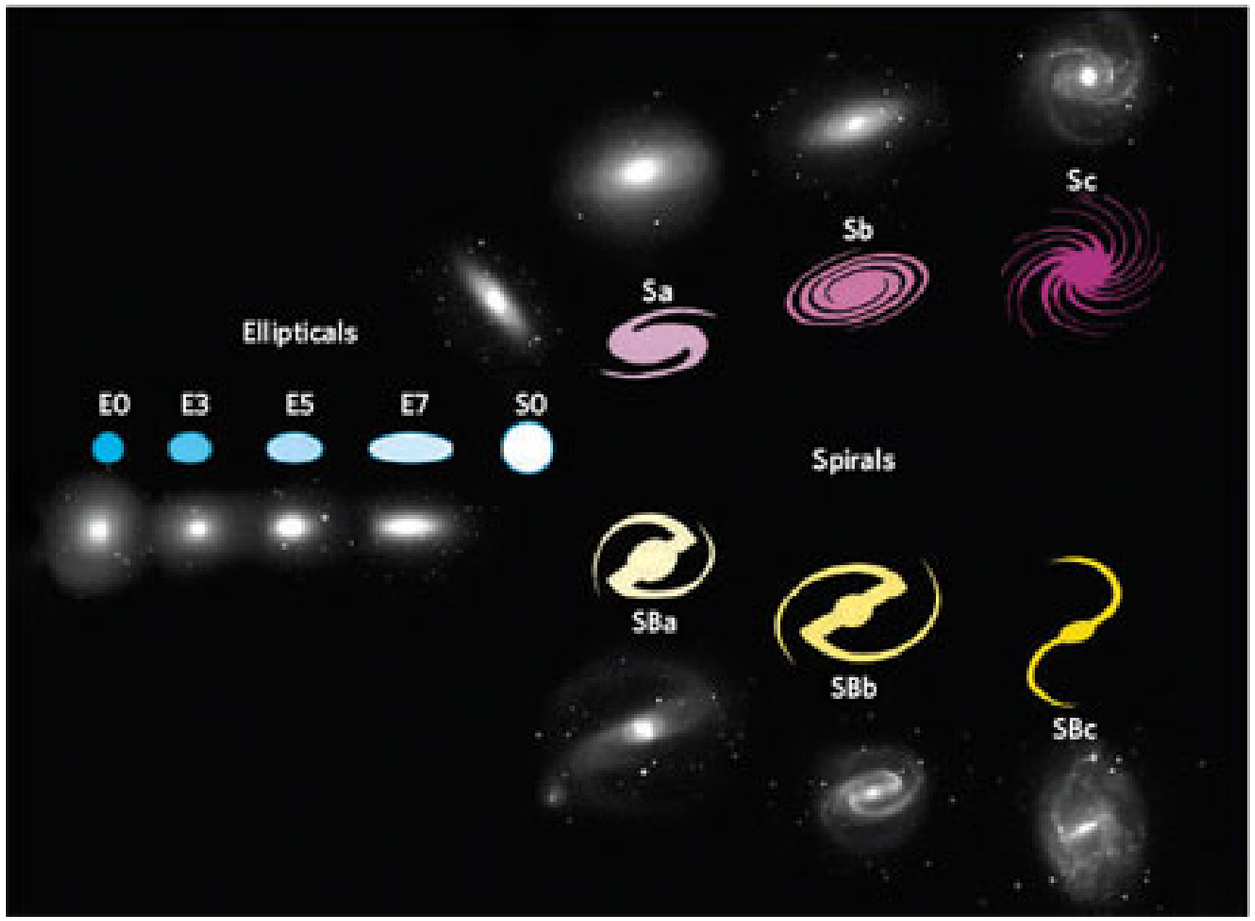
Classification

Classifying data

Astronomers very often classify stuff - this is one classical data-mining task that is very common in astronomy.



Wikipedia



Kennicutt (2006), Nature

The meanina of classification

I. Combination into known classes:

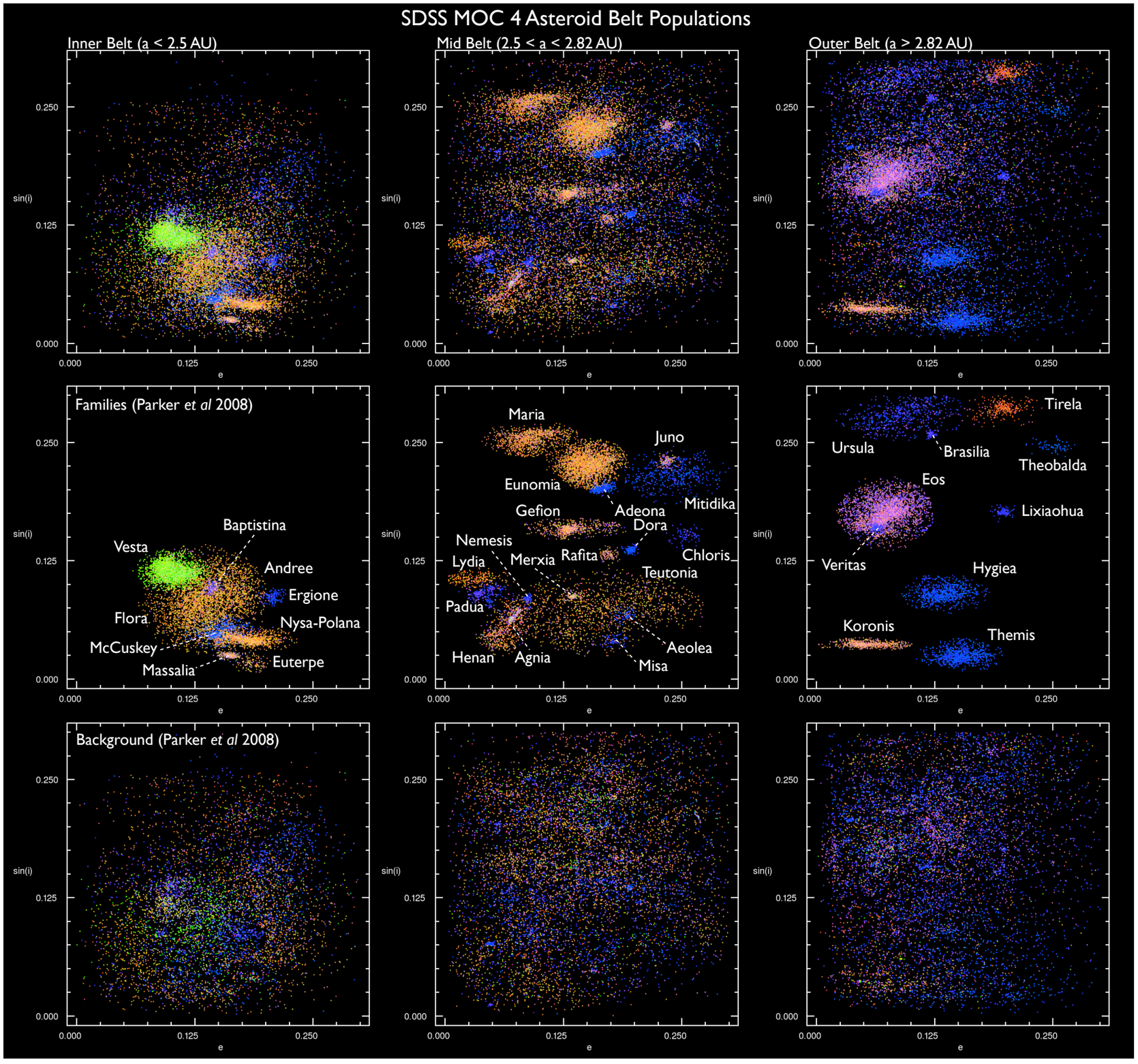
E.g GAIA:

Have: Photometric observations of N objects and low-resolution spectroscopic observations of M objects.

Want: Classification of objects into stars, galaxies, quasars etc. For the stars we also want T_{eff}, log g, [Fe/H] etc. [this is actually at least two different problems!]

So how do you do that?

II. Group objects to explore their nature



Finding and classifying asteroids

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

Supervised & unsupervised classification

Supervised classification:

Given: Objects with “features” $\{x_i\}$, and known class ω_j

Needed: For an object with features $\{y_i\}$, assign a class ω

It is supervised, because we know of the existence of the classes before and we have some objects that have been reliably classified in advance.

Classification of stellar spectra and galaxy images can fall into this category.

Supervised & unsupervised classification

Unsupervised classification:

Given: Objects with “features” $\{x_i\}$

Needed: Assign these to N (which can be known or given) classes. Then use this to assign classes to new data.

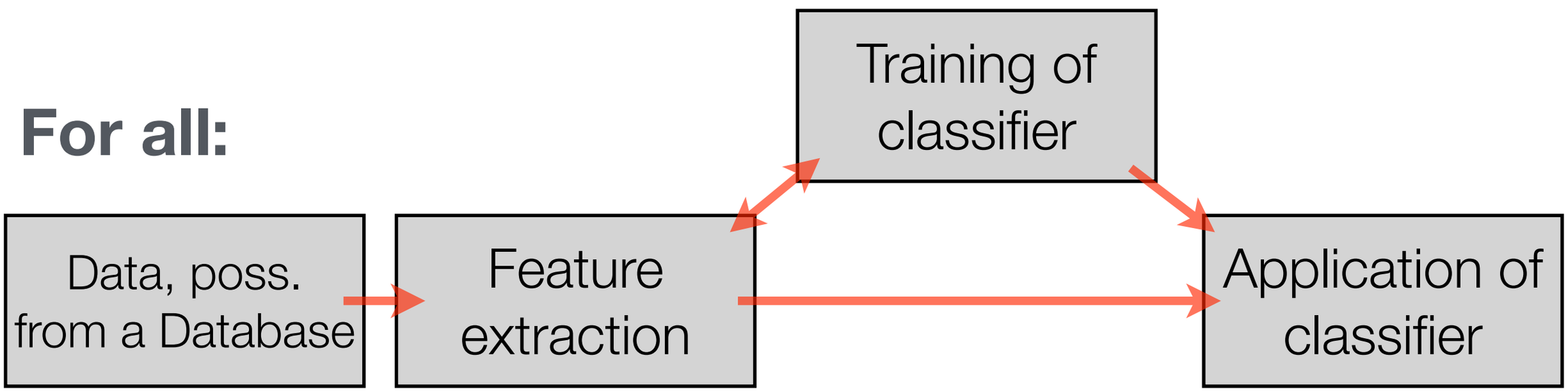
In this case we know nothing, or very little about the classification of the data. Thus this is often exploratory and the results can be a lot harder to interpret.

Finding asteroid families fall in this category.

Techniques for classification

- Bayesian decision criteria
- Nearest neighbour methods & friends.
- Linear models.
- Artificial Neural Networks.
- And many more (cladistics, hierarchical trees, **support vector machines** etc etc.)

For all:



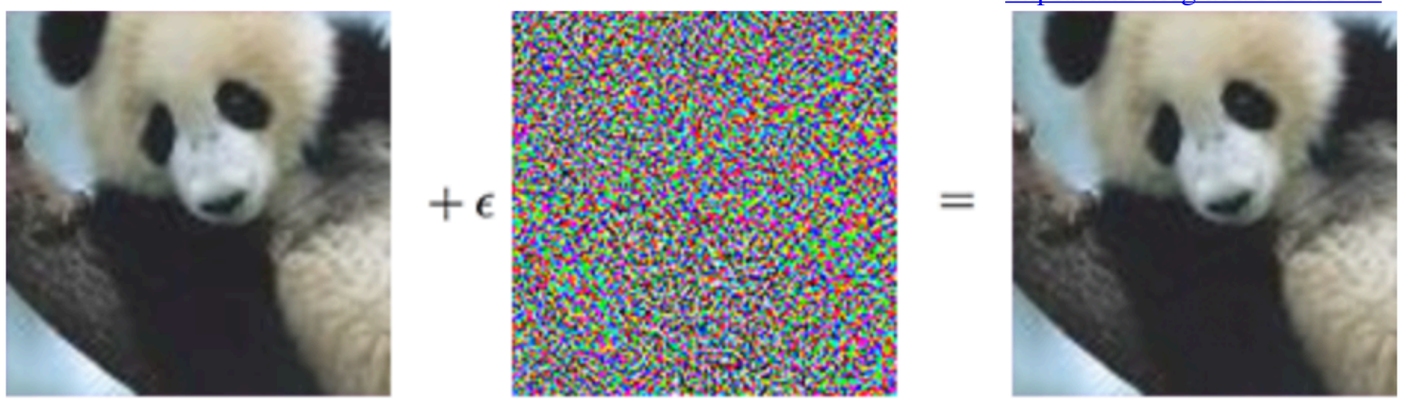
What is feature extraction?

- A crucial part of data classification/data mining - it is how you go about choosing the input data to the training algorithm.
- Depends strongly on the scientific question under consideration.
- For spectra it could be emission lines and absorption lines.
- For an image magnitudes, colours, light concentration, light profiles etc.
- Often it can be advisable to use something like principal component analysis (later) to guide your choice.
- It could also be the full image/full spectrum - this is often the way taken by deep learning algorithms (but with some modifications).

Classification - and training

Machine learning typically relies on you having good training data covering the possible results, but even so it sometimes fails:

https://arxiv.org/abs/1412.6572



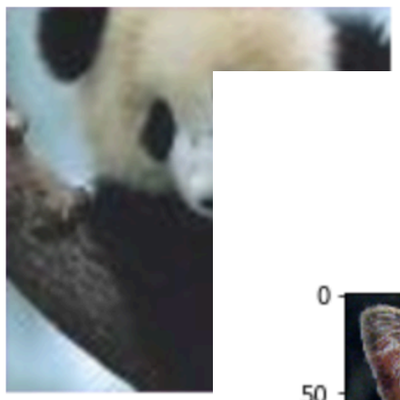
“panda”
57.7% confidence

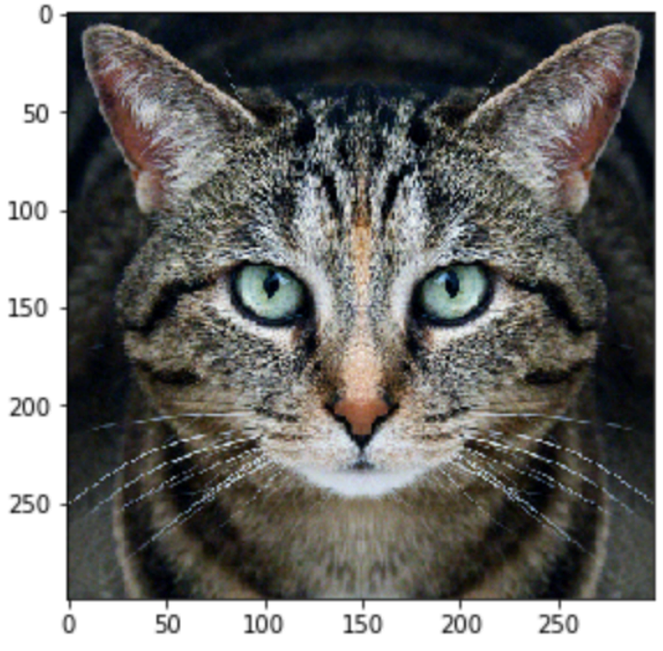
“gibbon”
99.3% confidence

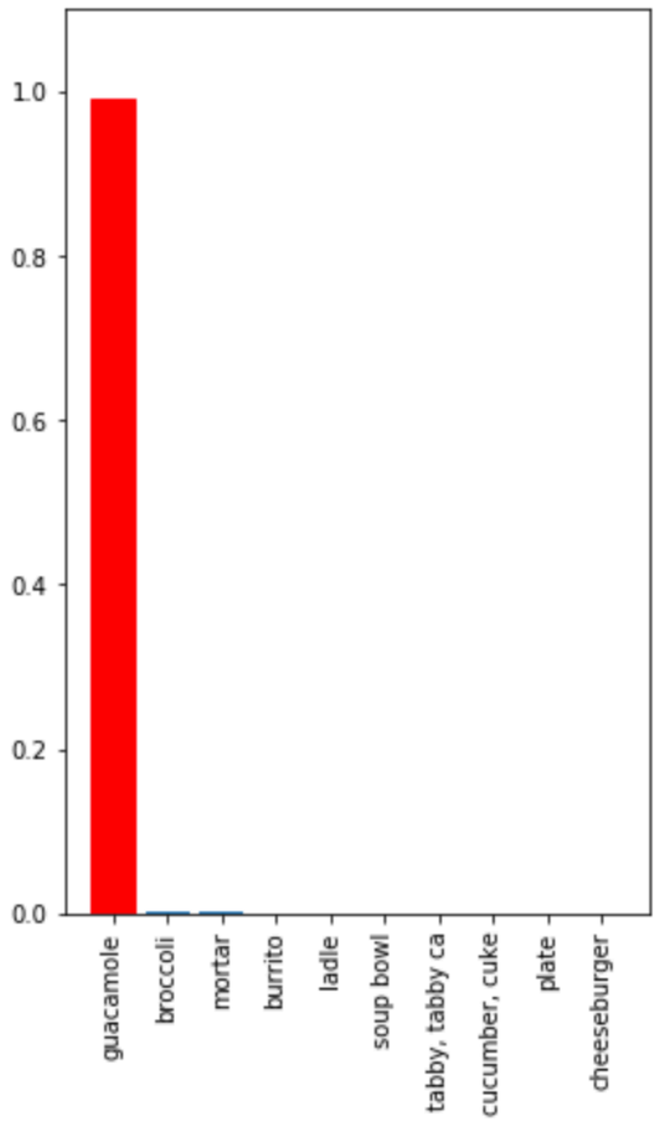
Classification - and training

Machine learning typically relies on you having good training data covering the possible results, but even so it sometimes fails:

https://arxiv.org/abs/1412.6572







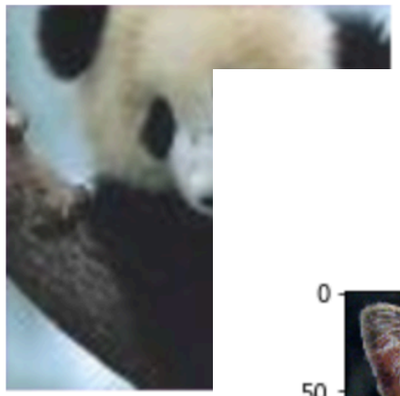
"pan

57.7% cor

Classification - and training

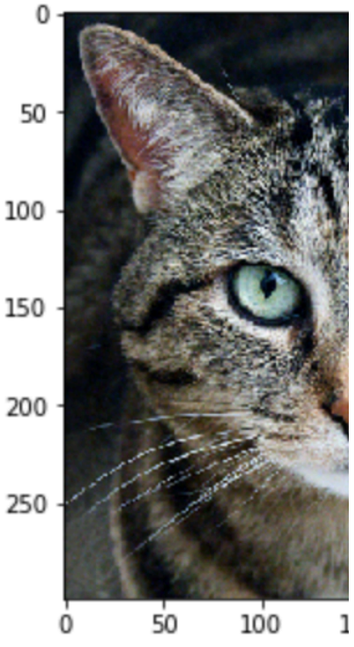
Machine learning typically relies on you having good training data covering the possible results, but even so it sometimes fails:

https://arxiv.org/abs/1412.6572

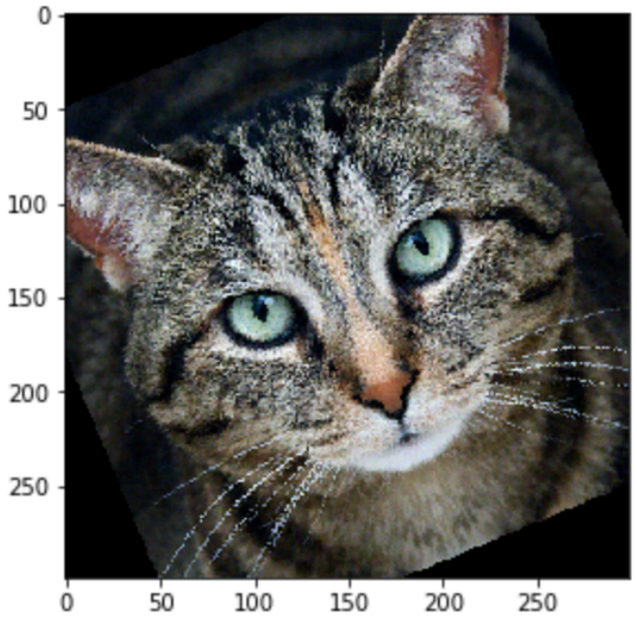


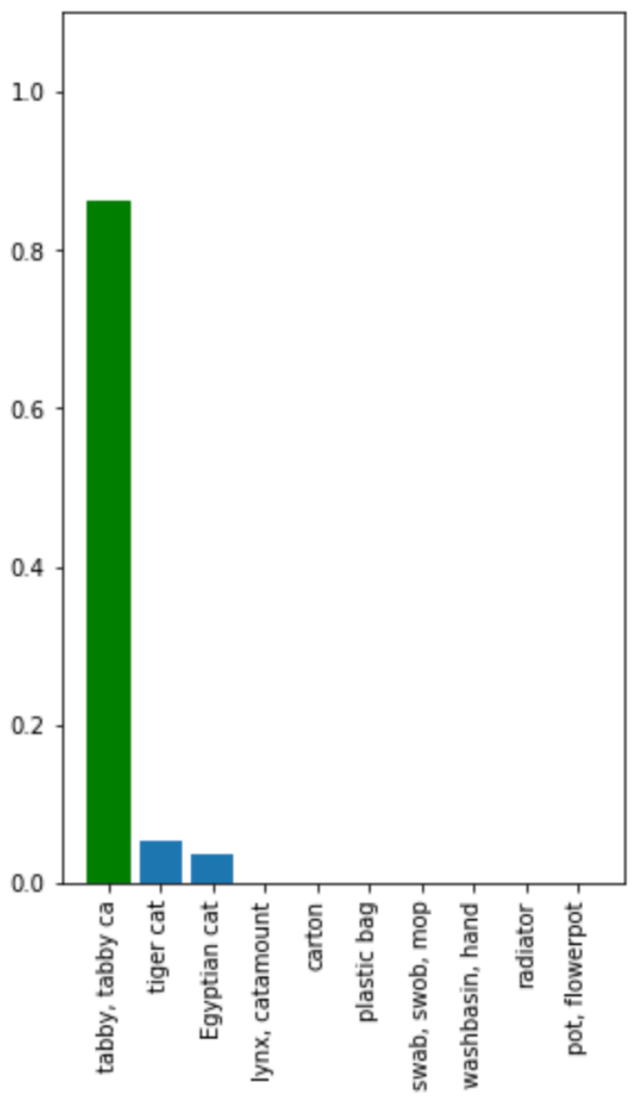
“pan

57.7% cor



However:





An example: Bayesian classification of galaxies

Starting point: Morphological classifications by Nair & Abraham (2010), downloaded from Vizir & matched to SDSS absolute magnitudes I calculated separately.

The task: Design a classifier that given a galaxy colour will return the morphological class.

Notation:

ω_0	Elliptical
ω_1	Spiral
$\vec{x}$	Observables/Features - here: g-r colour

Bayes' theorem & optimal class assignment

Bayes' theorem for our case: $$p(\omega_i|\vec{x}) = \frac{p(\vec{x}|\omega_i)p(\omega_i)}{p(\vec{x})}$$

$p(\omega_i|\vec{x})$ is the **posterior** probability

$p(\vec{x}|\omega_i)$ is the **likelihood**

$p(\omega_i)$ is the **prior** probability of a class

$$p(\vec{x}) = \sum_i p(\vec{x}|\omega_i)p(\omega_i)$$

ensures the probabilities are normalised but as it is independent of the classes we can ignore it here.

Bayes' theorem & optimal class assignment

Bayes' theorem for our case: $$p(\omega_i|\vec{x}) = \frac{p(\vec{x}|\omega_i)p(\omega_i)}{p(\vec{x})}$$

Classification is then very straightforward:

$$p(\omega_0|\vec{x}) > p(\omega_1|\vec{x}) \Rightarrow \text{Class } 0 \text{ (elliptical)}$$
$$p(\omega_0|\vec{x}) < p(\omega_1|\vec{x}) \Rightarrow \text{Class } 1 \text{ (spiral)}$$

But this supposes we know $p(\vec{x}|\omega_i)$ and $p(\omega_i)$

Bayes' theorem & optimal class assignment

Bayes' theorem for our case: $$p(\omega_i|\vec{x}) = \frac{p(\vec{x}|\omega_i)p(\omega_i)}{p(\vec{x})}$$

Classification is then very straightforward:

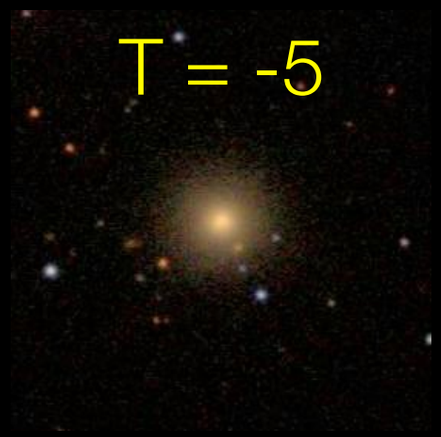
$$p(\omega_0|\vec{x}) > p(\omega_1|\vec{x}) \Rightarrow \text{Class } 0 \text{ (elliptical)}$$
$$p(\omega_0|\vec{x}) < p(\omega_1|\vec{x}) \Rightarrow \text{Class } 1 \text{ (spiral)}$$

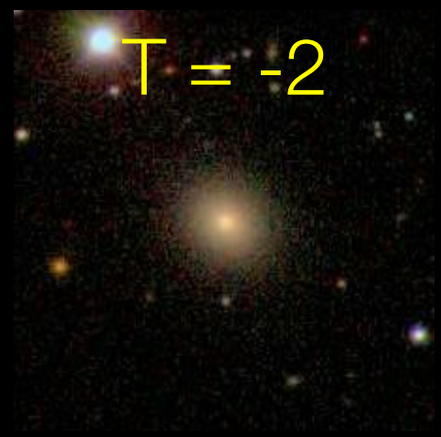
But this supposes we know $p(\vec{x}|\omega_i)$ and $p(\omega_i)$

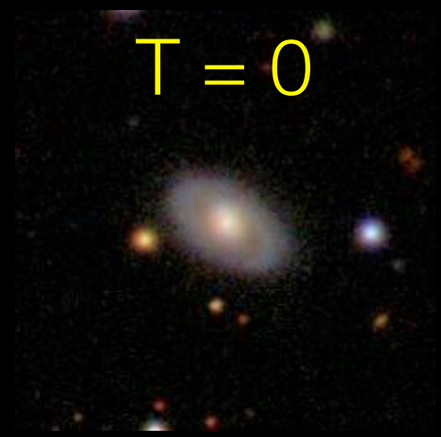
If we assume all classes are equally probable, we just need $p(\vec{x}|\omega_i)$ and for this we can use the tools from earlier:

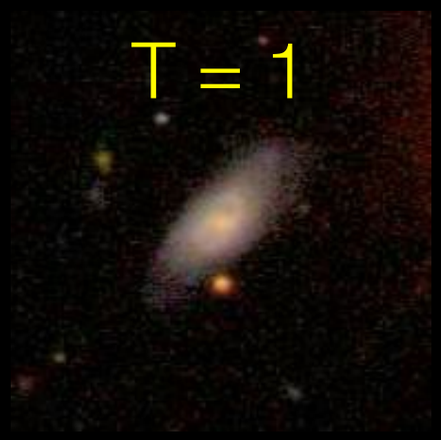
Histograms, kernel estimates or estimating the mean and standard deviation of a Gaussian.

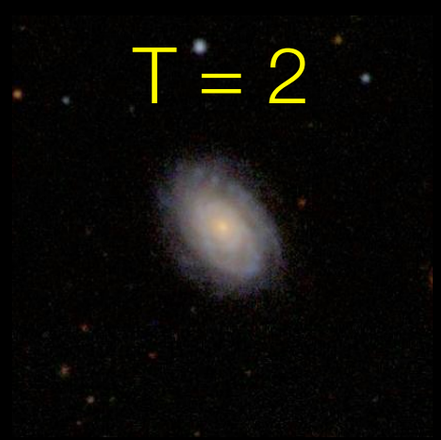
Interlude - Galaxy T-types

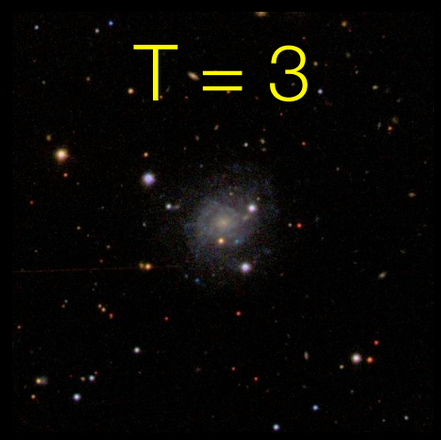


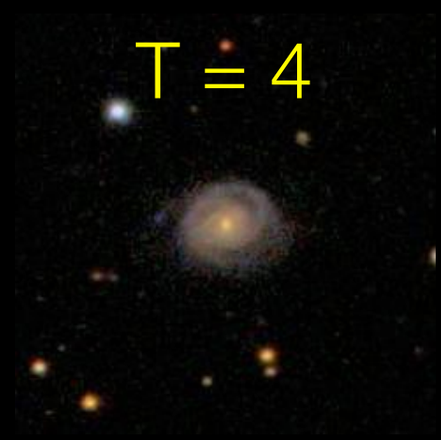


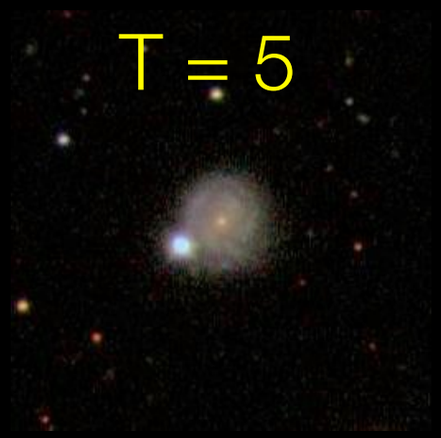


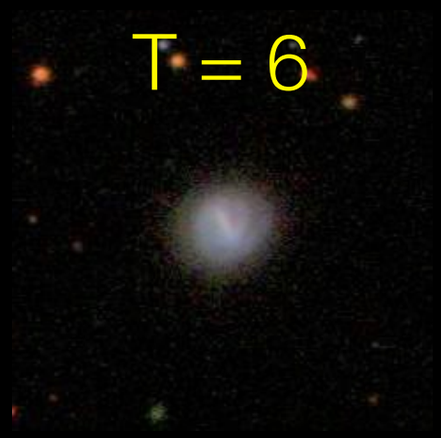


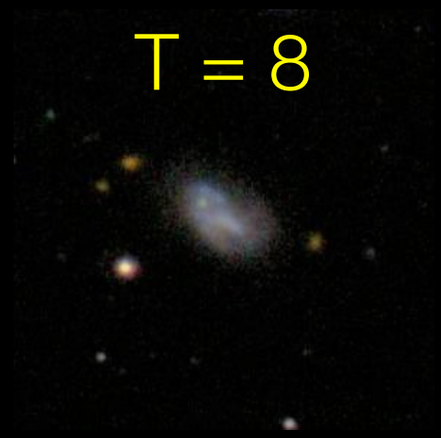




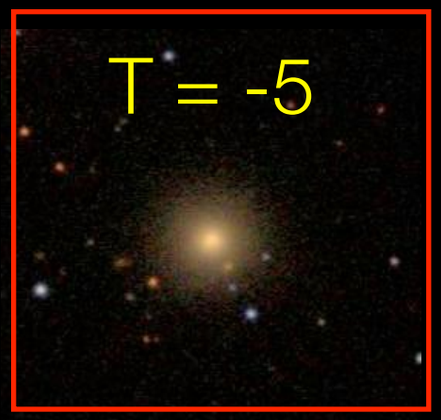


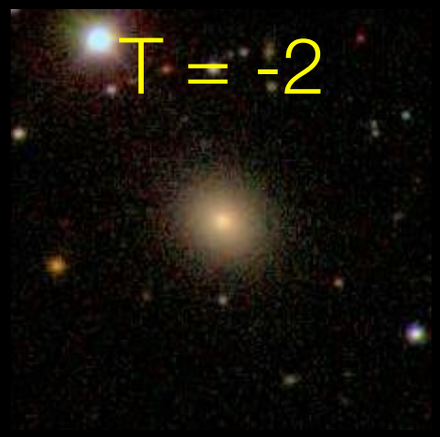


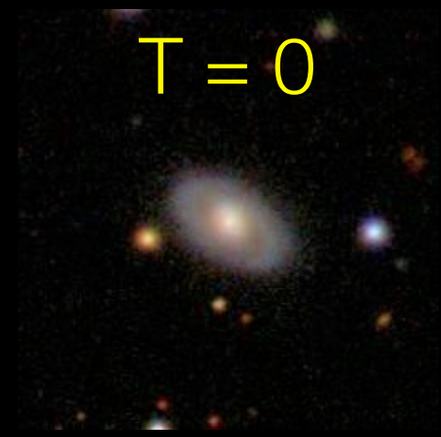


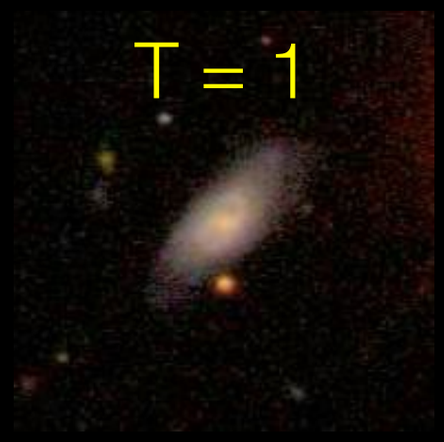


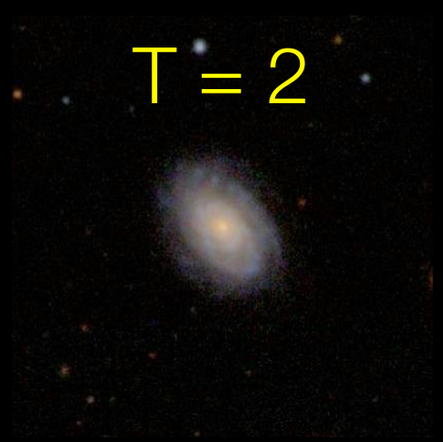
Interlude - Galaxy T-types

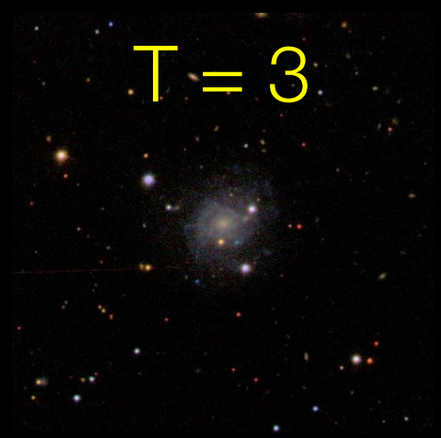


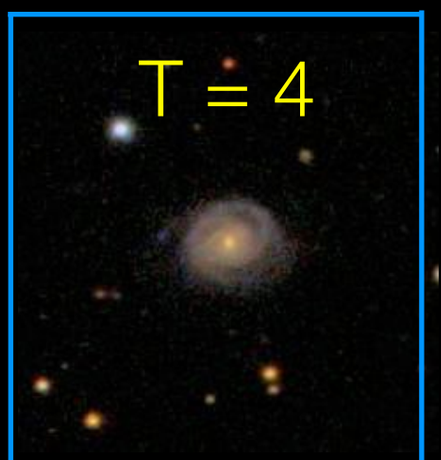


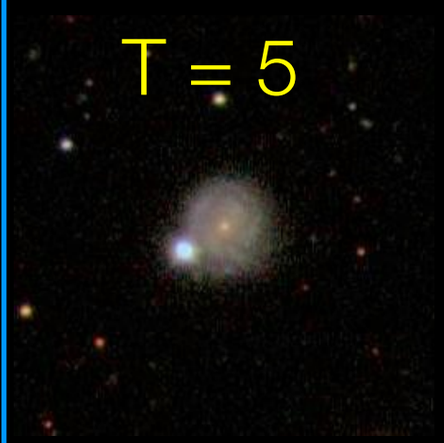


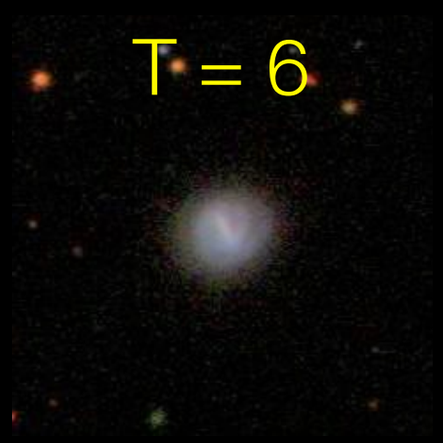


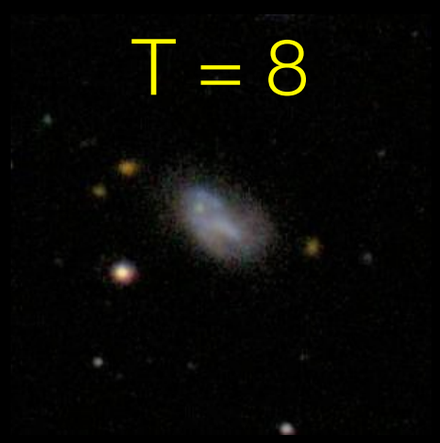


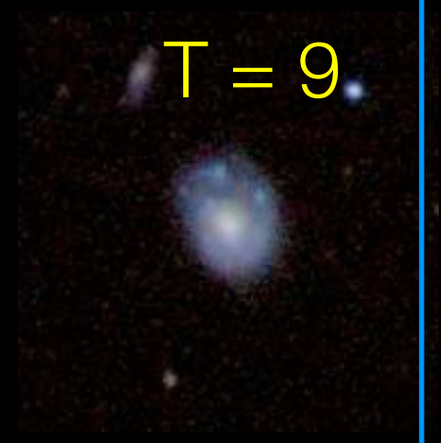


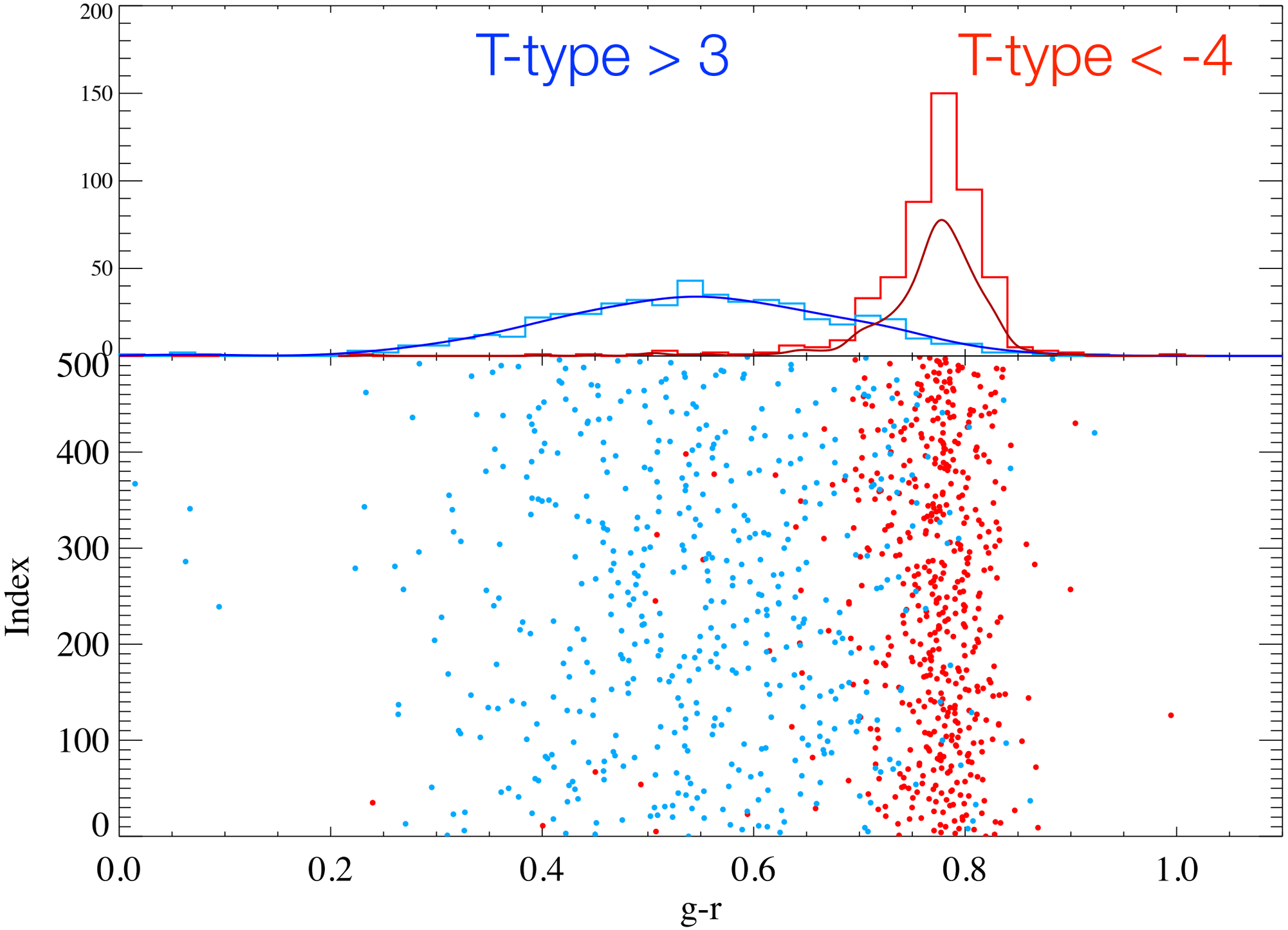


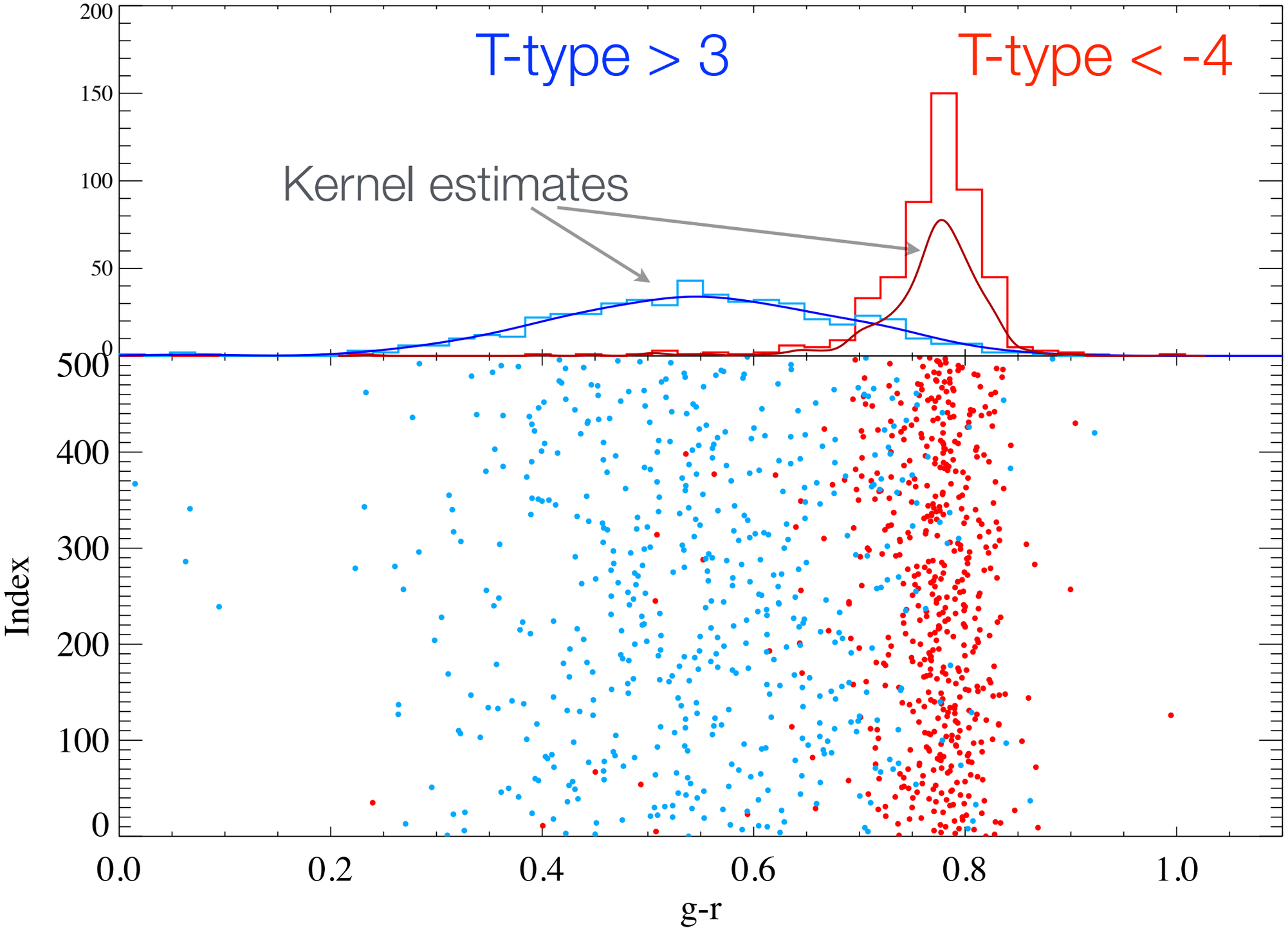


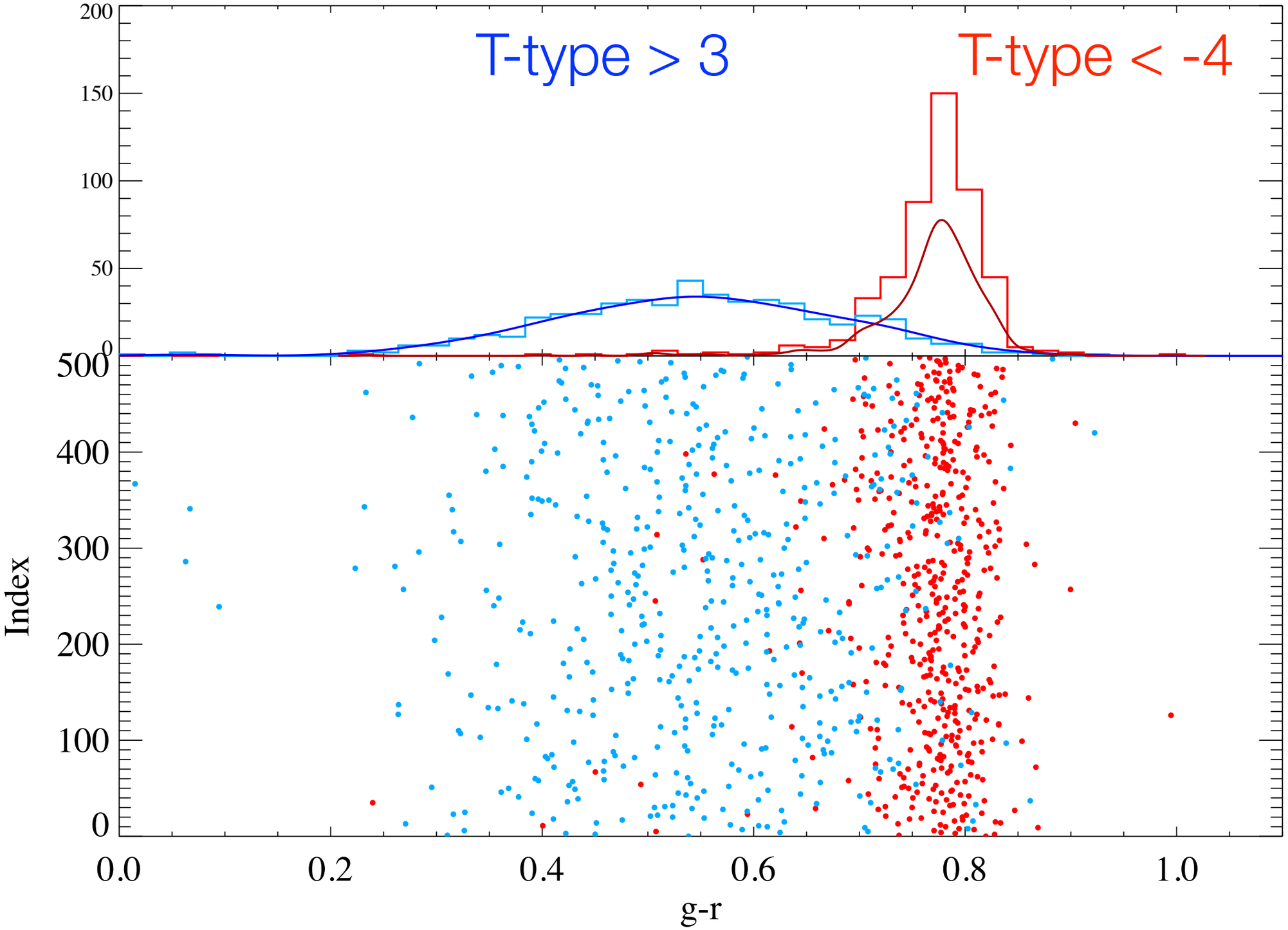


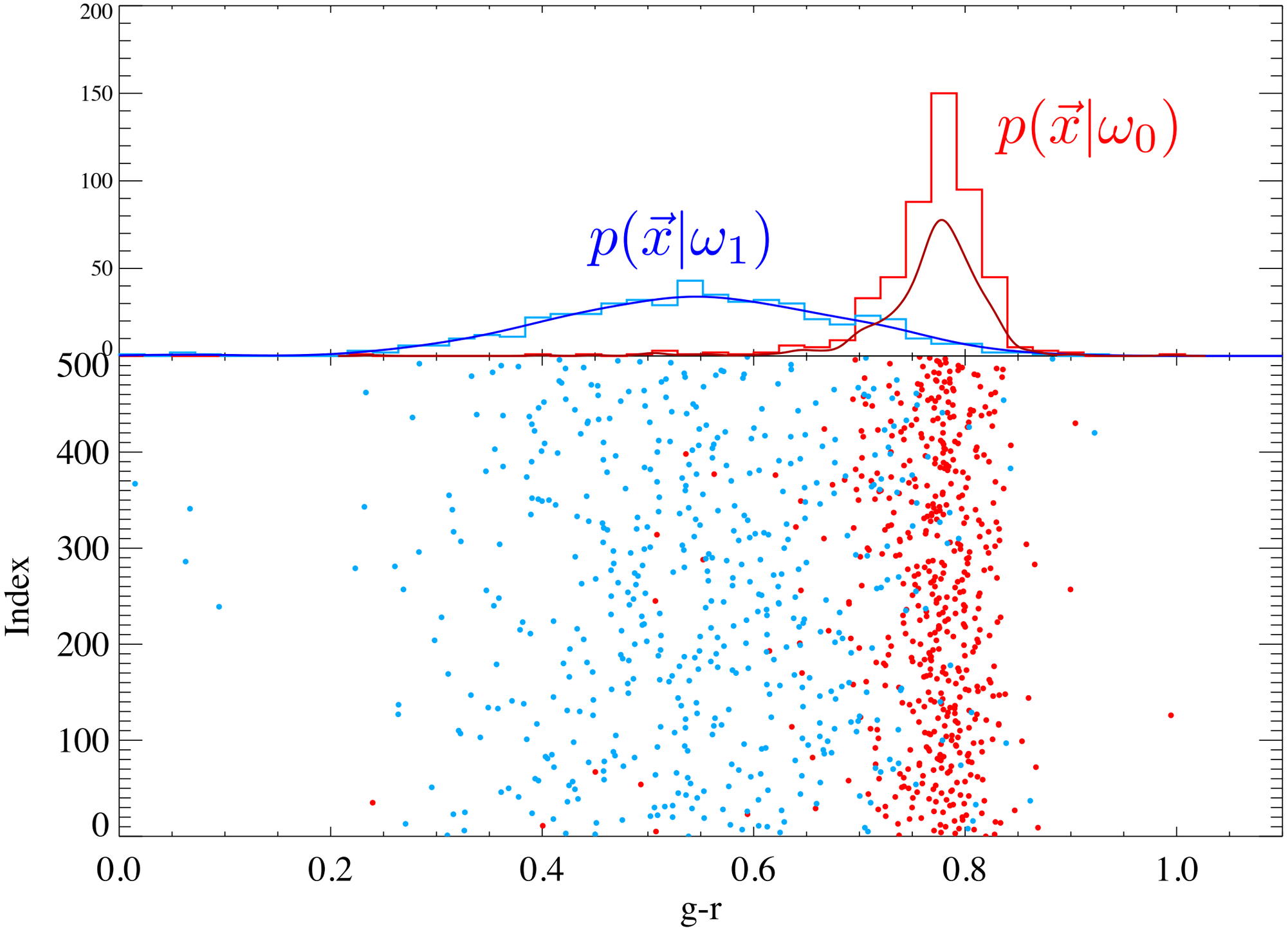




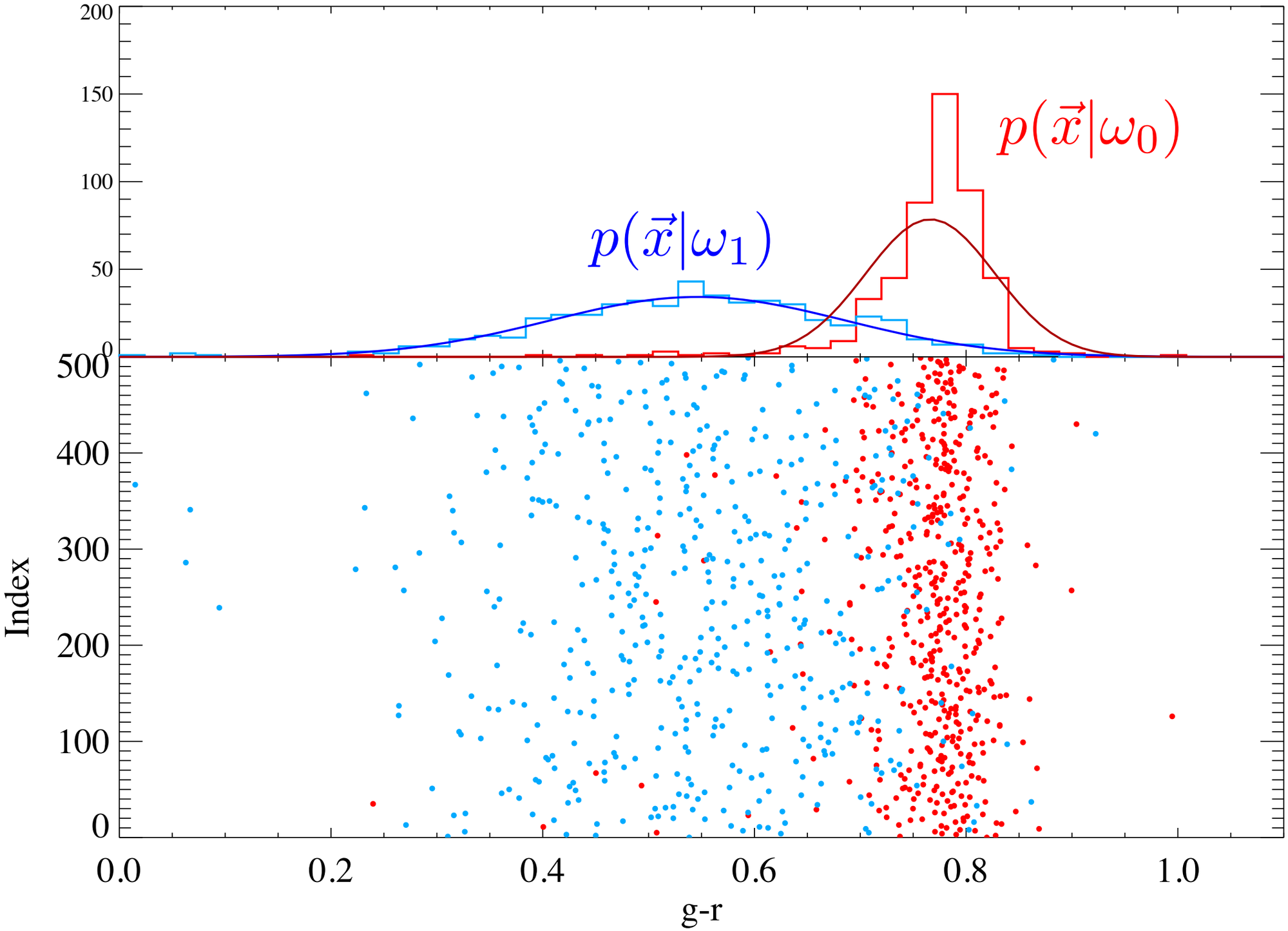

T-type > 3
T-type < -4
200
150
100
50
0
500
400
300
200
100
0
Index
0.0
0.2
0.4
0.6
0.8
1.0
g-r


T-type > 3
T-type < -4
Kernel estimates
200
150
100
50
0
500
400
300
200
100
0
Index
0.0
0.2
0.4
0.6
0.8
1.0
g-r

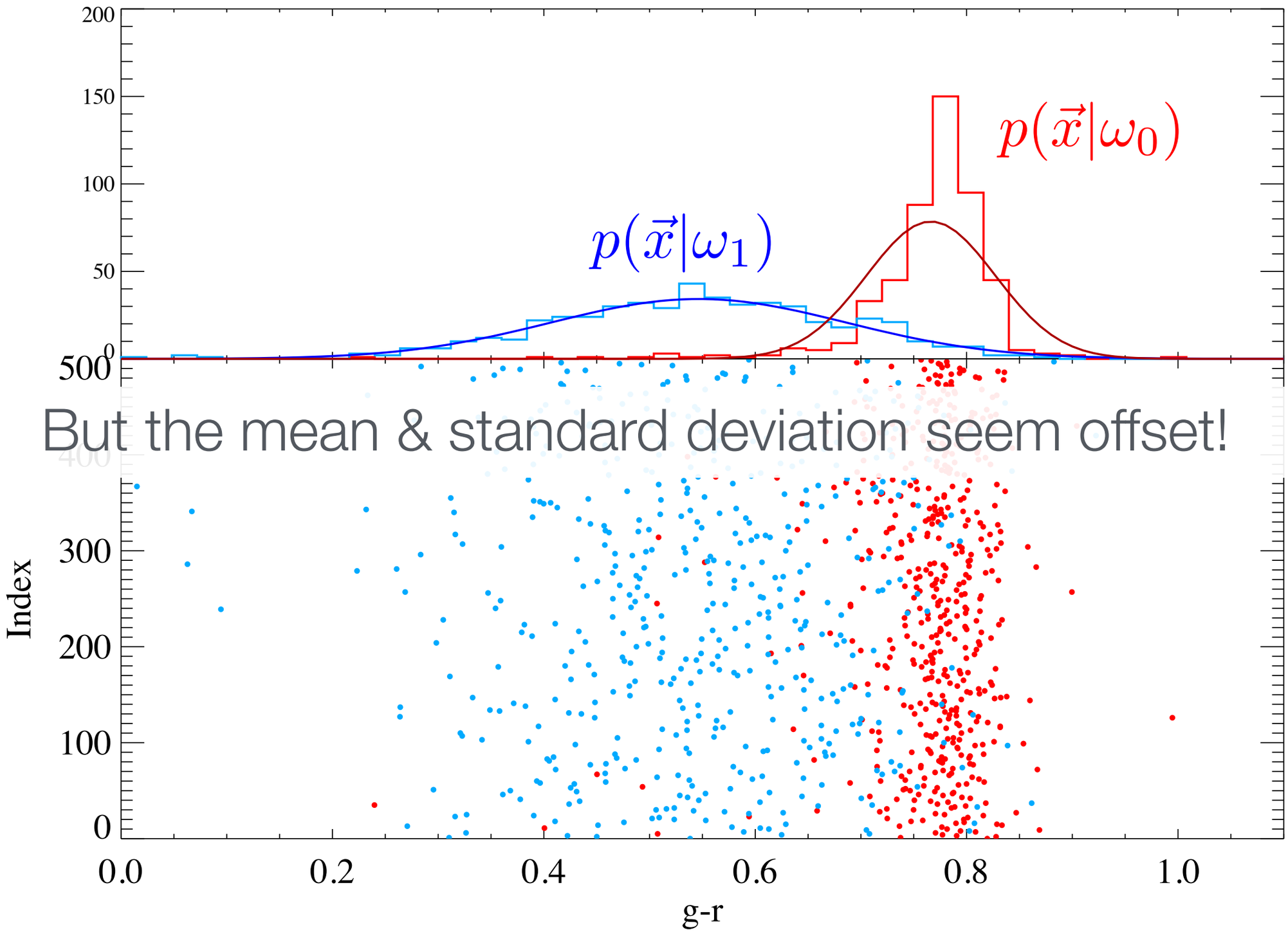

T-type > 3
T-type < -4
200
150
100
50
0
500
400
300
200
100
0
Index
0.0
0.2
0.4
0.6
0.8
1.0
g-r


$p(\vec{x}|\omega_1)$
$p(\vec{x}|\omega_0)$
Index
g-r
0.0
0.2
0.4
0.6
0.8
1.0
0
100
200
300
400
500
50
100
150
200

A Gaussian approximation is easier to work with:

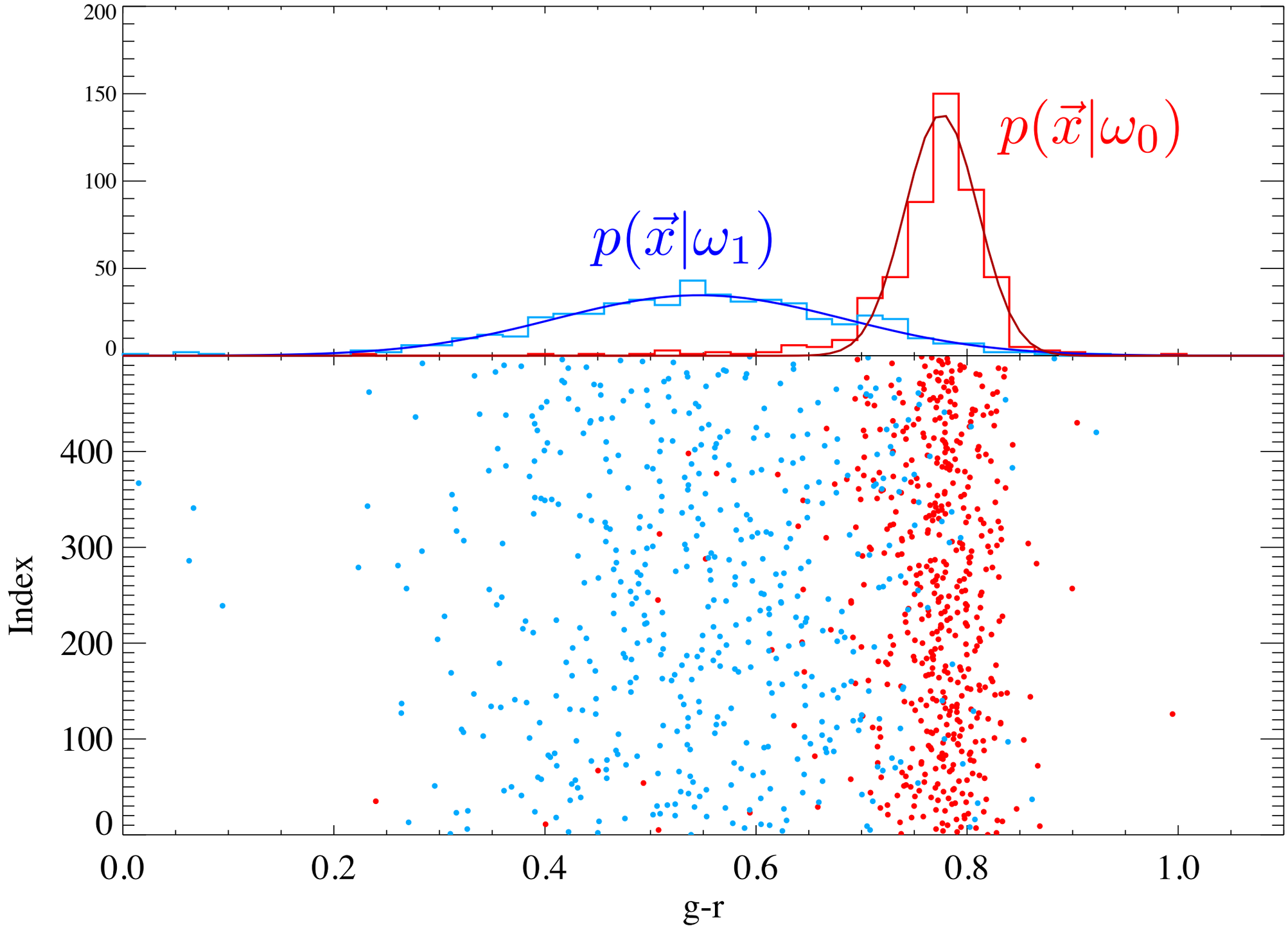


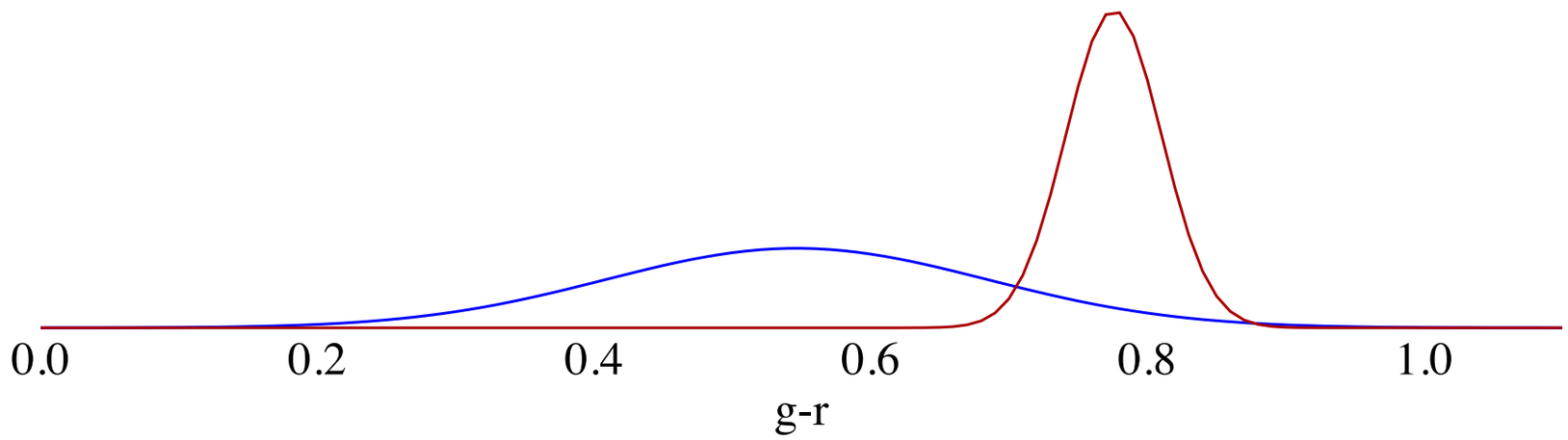
A Gaussian approximation is easier to work with:

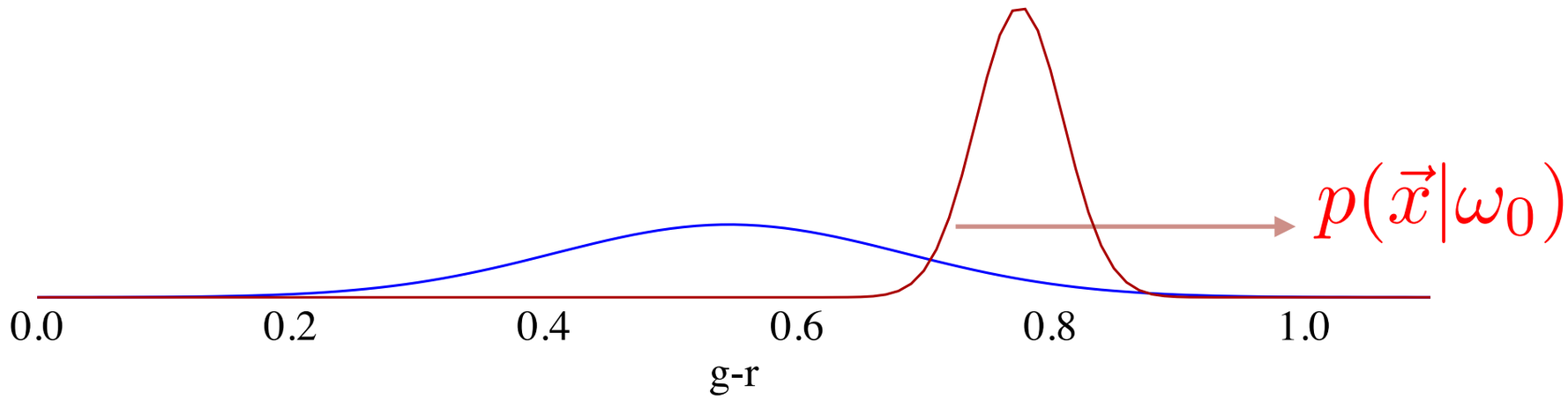


But the mean & standard deviation seem offset!

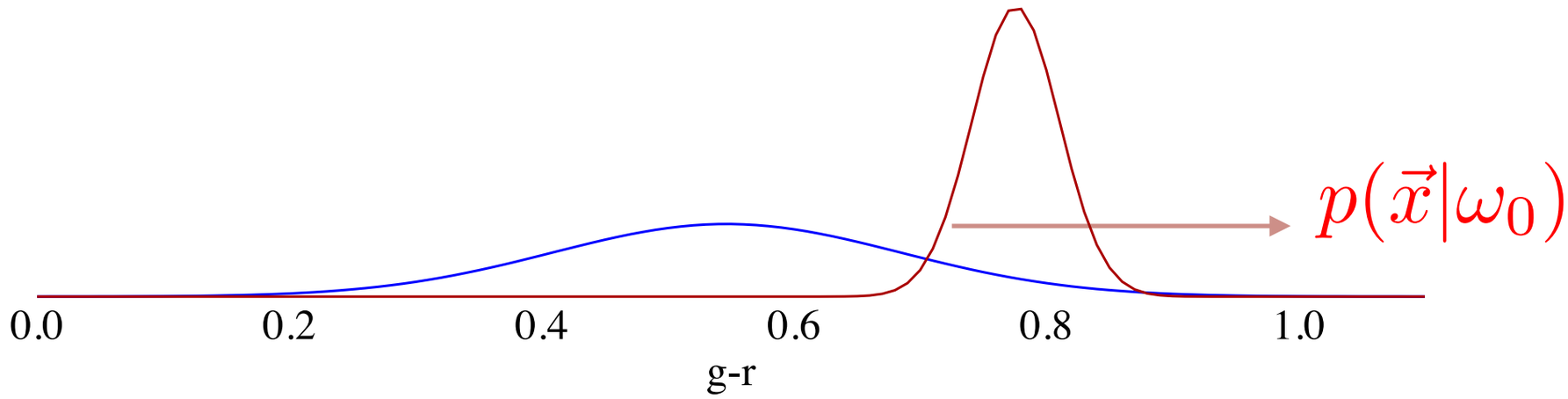
Maybe robust estimators are better (median, MAD):



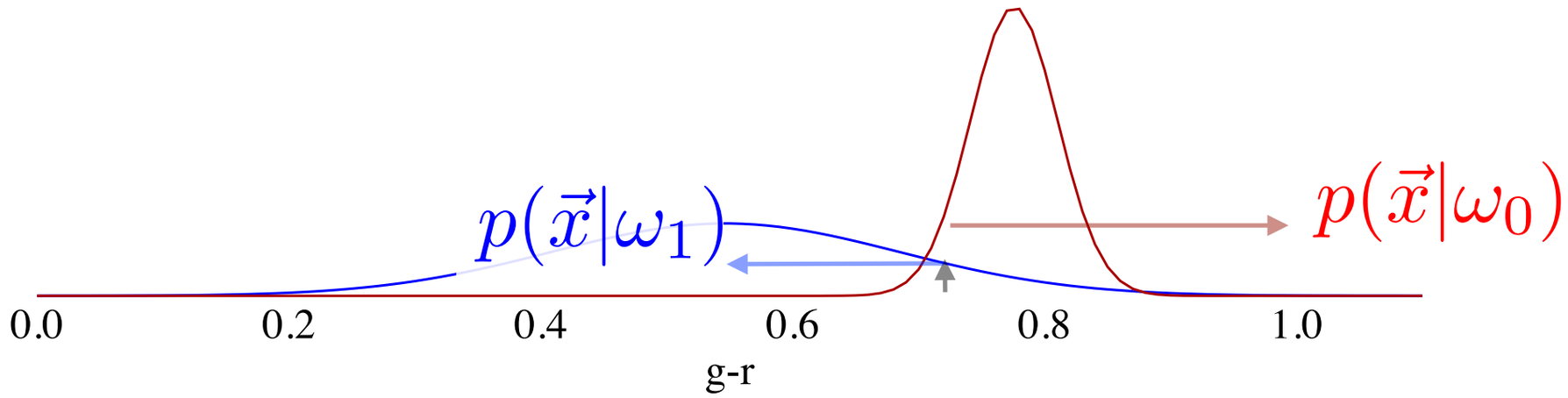

0.0
0.2
0.4
0.6
0.8
1.0
g-r


$p(\vec{x}|\omega_0)$
0.0
0.2
0.4
0.6
0.8
1.0
g-r

Bayesian classification:

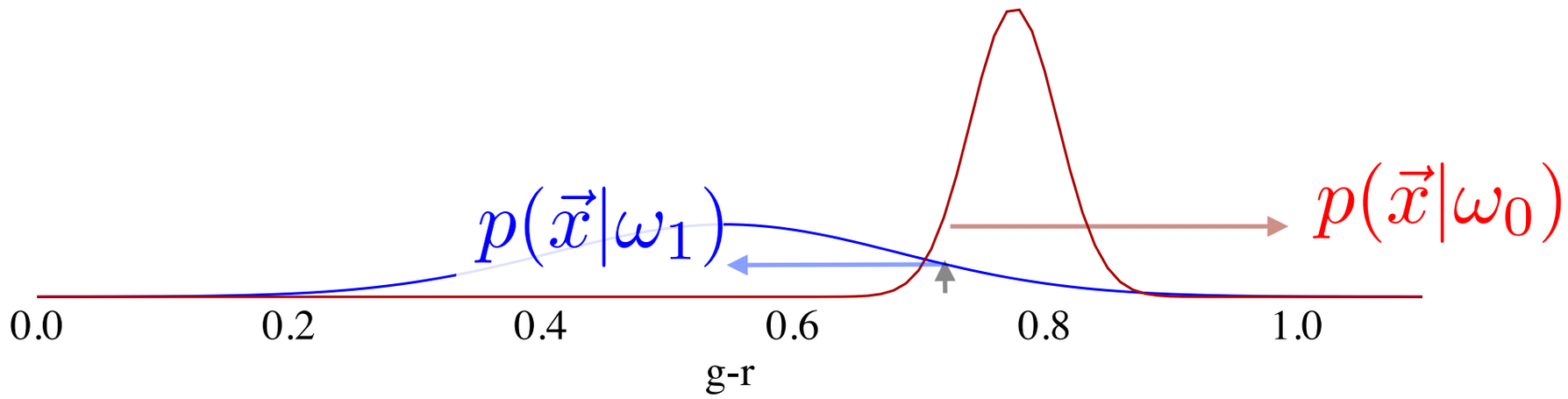


Bayesian classification:

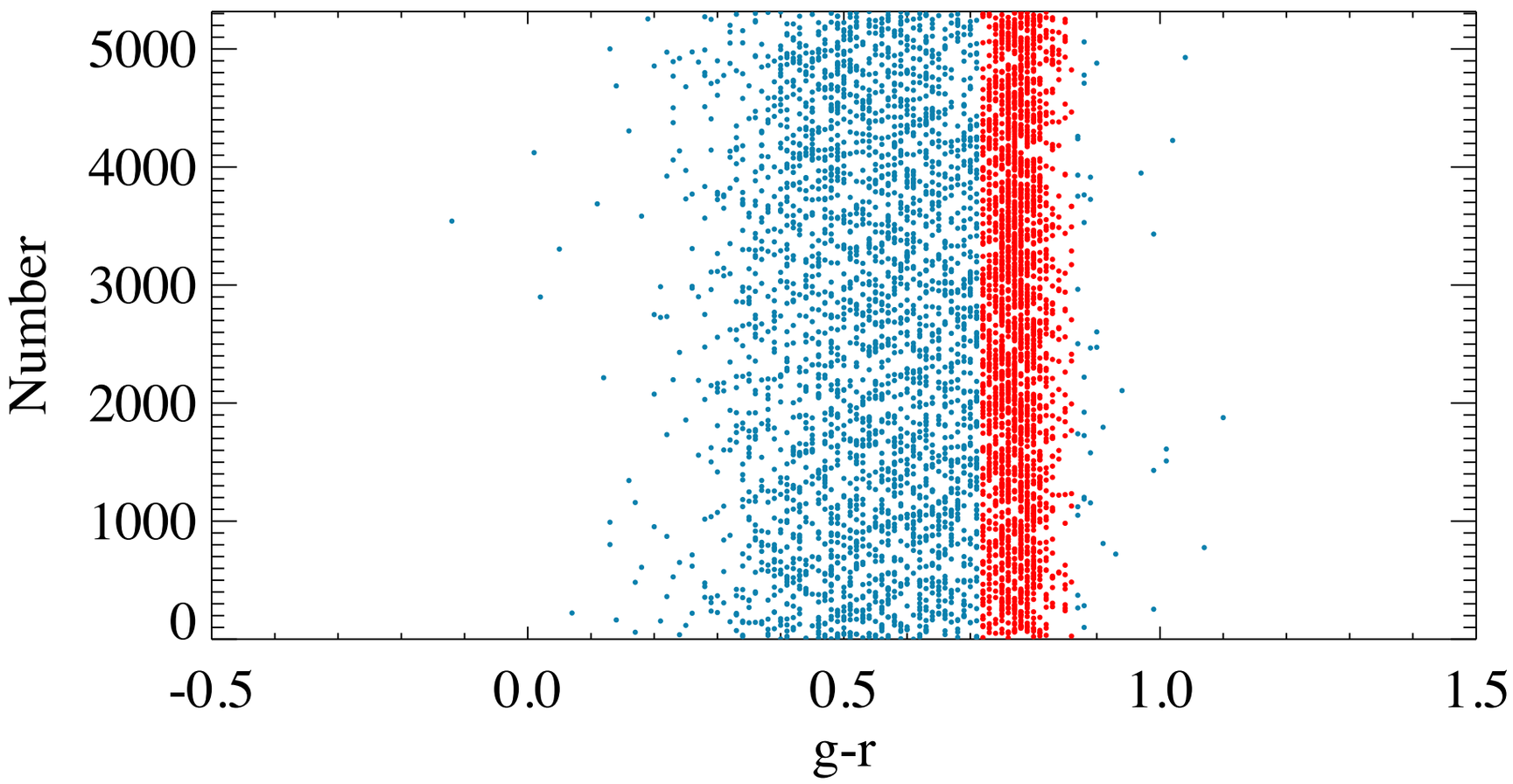


So in this case we assign a class Elliptical and we can also give the likelihood relative to the spiral class. We do this for all objects:

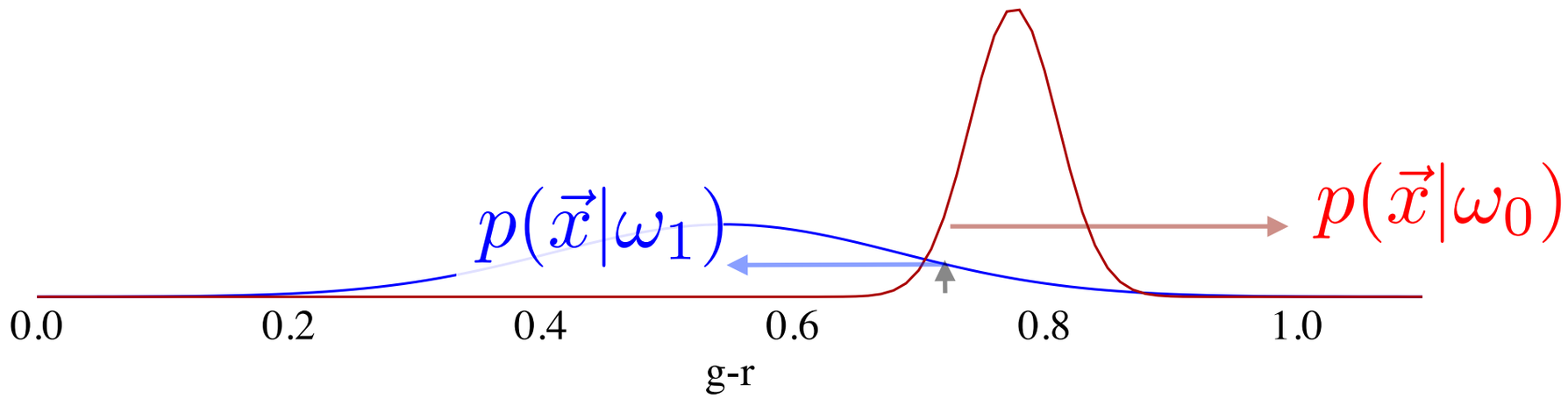
Bayesian classification:



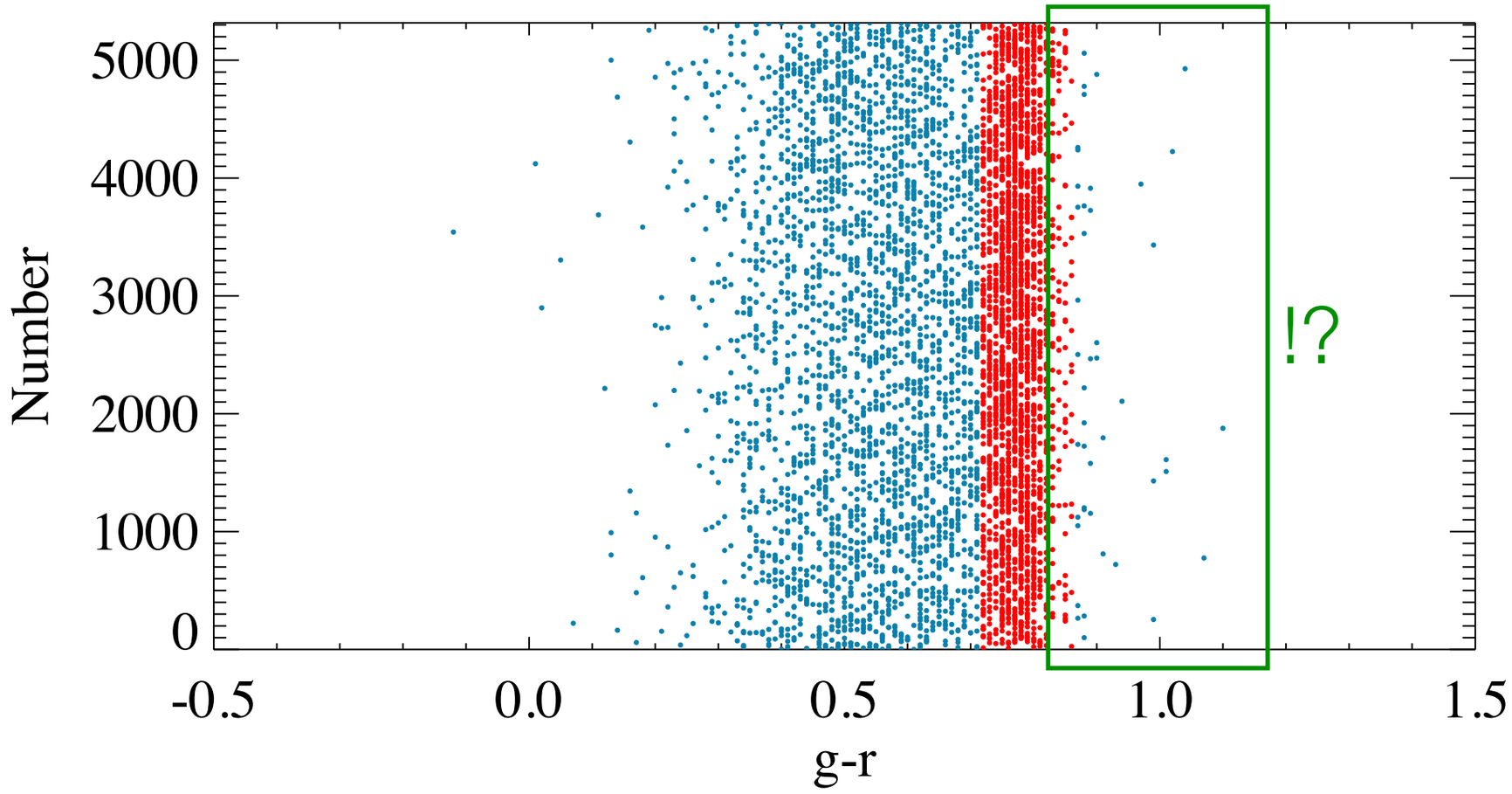
So in this case we assign a class Elliptical and we can also give the likelihood relative to the spiral class. We do this for all objects:



Bayesian classification:



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



Bayesian classification - decision surfaces

We can write our class separation criterion as:

$$p(\omega_0|\vec{x}) = p(\omega_1|\vec{x})$$

or

$$\ln p(\vec{x}|\omega_0)p(\omega_0) = \ln p(\vec{x}|\omega_1)p(\omega_1)$$

This will define decision regions, sometimes a surface

Taking logarithms and assuming a Gaussian PDF we have:

$$\left(\frac{x-\mu_0}{2\sigma_0}\right)^2 - \left(\frac{x-\mu_1}{2\sigma_1}\right)^2 = \ln\frac{\sigma_1}{\sigma_0}$$

$$p(\vec{x}|\omega_0) = \frac{1}{\sqrt{2\pi}\sigma_0} e^{-(x-\mu_0)^2/2\sigma_0^2}$$

$$p(\vec{x}|\omega_1) = \frac{1}{\sqrt{2\pi}\sigma_1} e^{-(x-\mu_1)^2/2\sigma_1^2}$$

So generally you get two solutions!

Validation

Getting a solution is a start - now we need to test (validate) the solution.

For this we have another sample which has known class - we apply our classifier and compare the predicted class with the known class.

$$\frac{\#\text{ cases where predicted class != true class}}{\#\text{ of cases}} = \text{Misclassification rate}$$

In our case: 10% (also if we use kernel estimation)

But if we remove our gap in T-types - the misclassification rate goes up to 25%...

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.

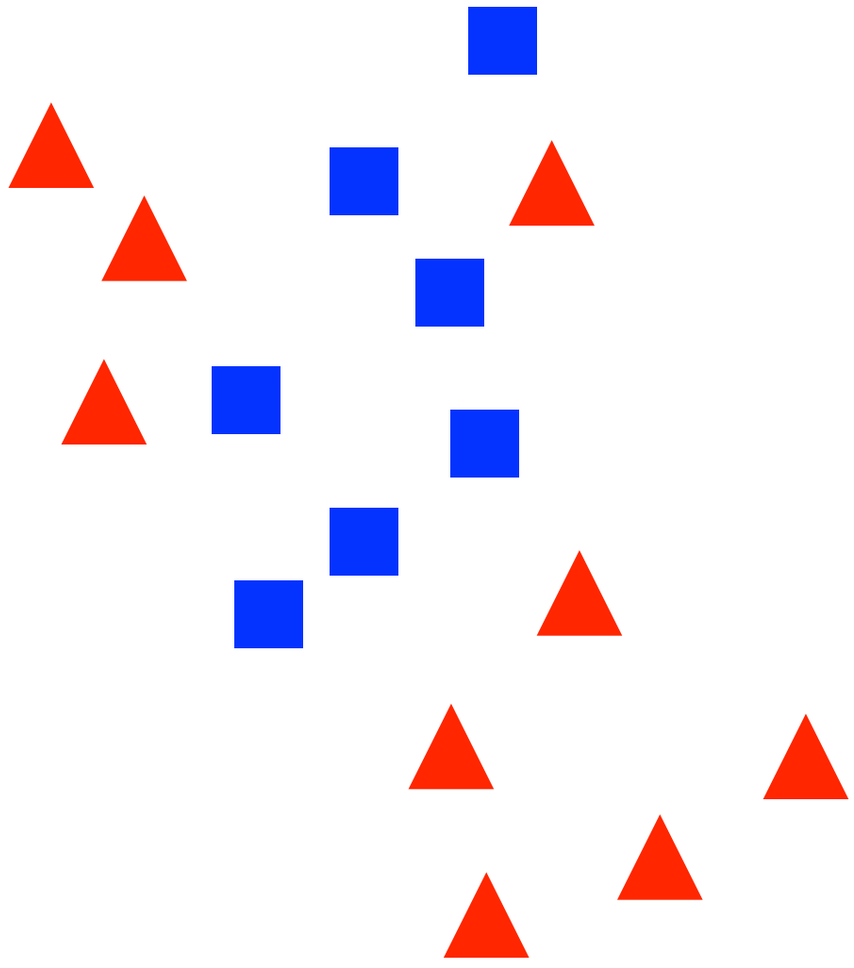
Other classification/grouping methods

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

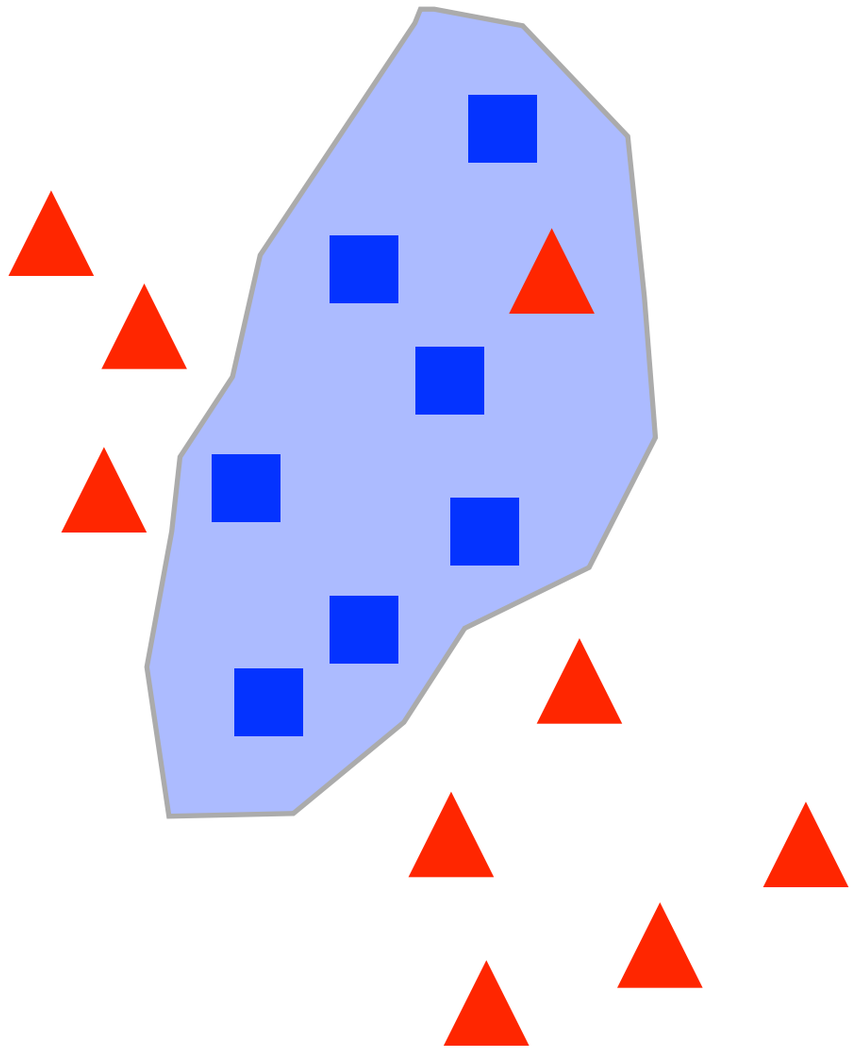
Training: For each object, find the nearest k objects - assign the class of the majority of the neighbours.



In python: from sklearn.neighbors import KNeighborsClassifier

k-Nearest neighbours

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

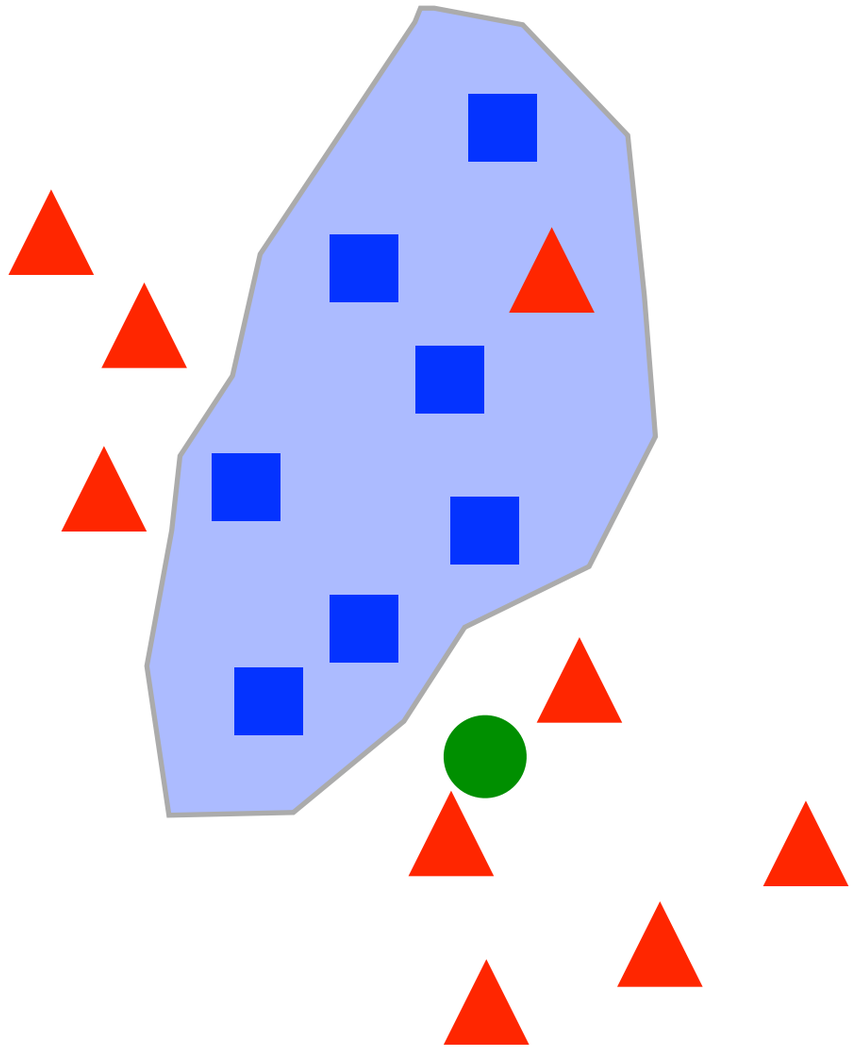


In python: from sklearn.neighbors import KNeighborsClassifier

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.

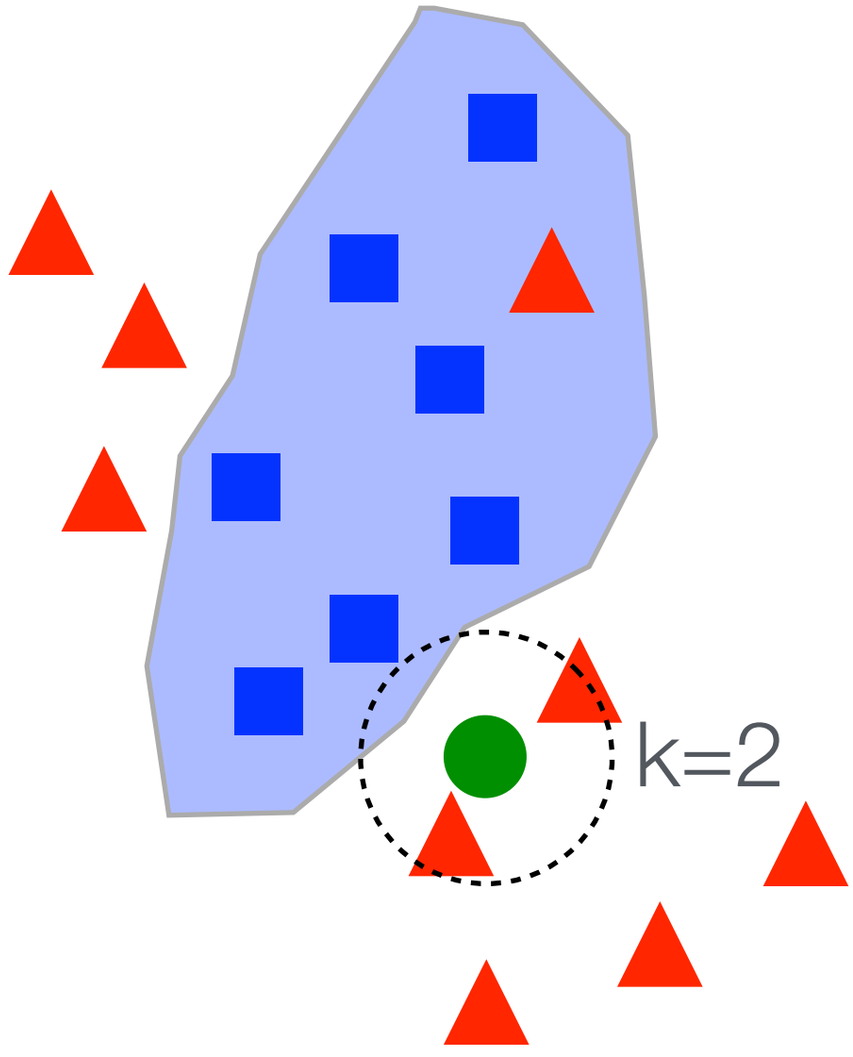


In python: from sklearn.neighbors import KNeighborsClassifier

k-Nearest neighbours

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.

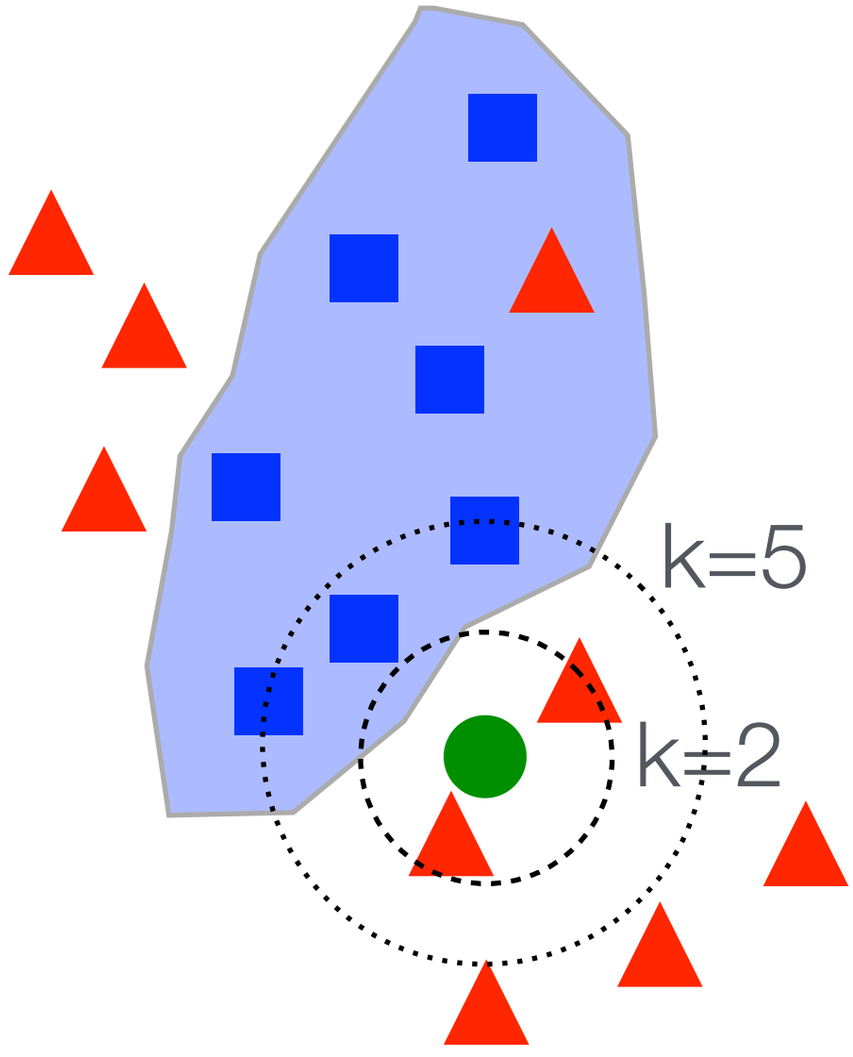


In python: from sklearn.neighbors import KNeighborsClassifier

k-Nearest neighbours

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

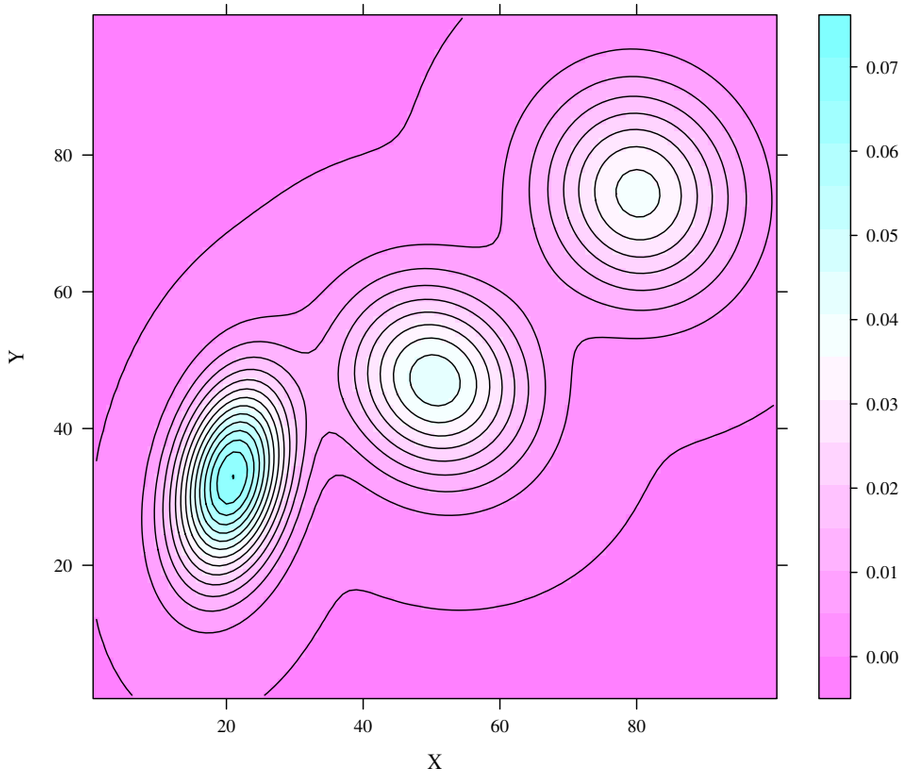
Application: Find the nearest k objects and assign the majority class.



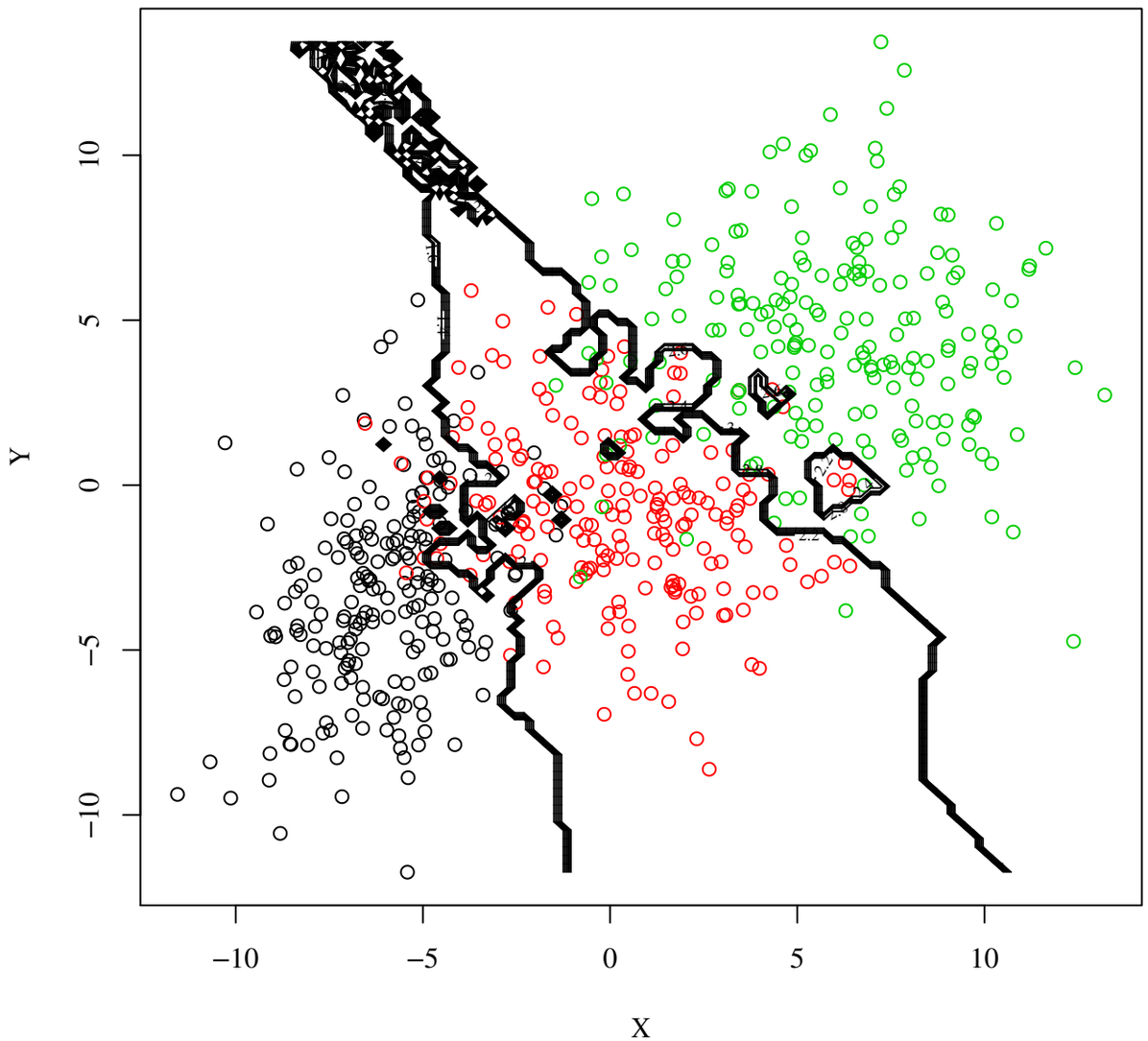
In python: from sklearn.neighbors import KNeighborsClassifier

k-Nearest neighbours

Underlying distributions



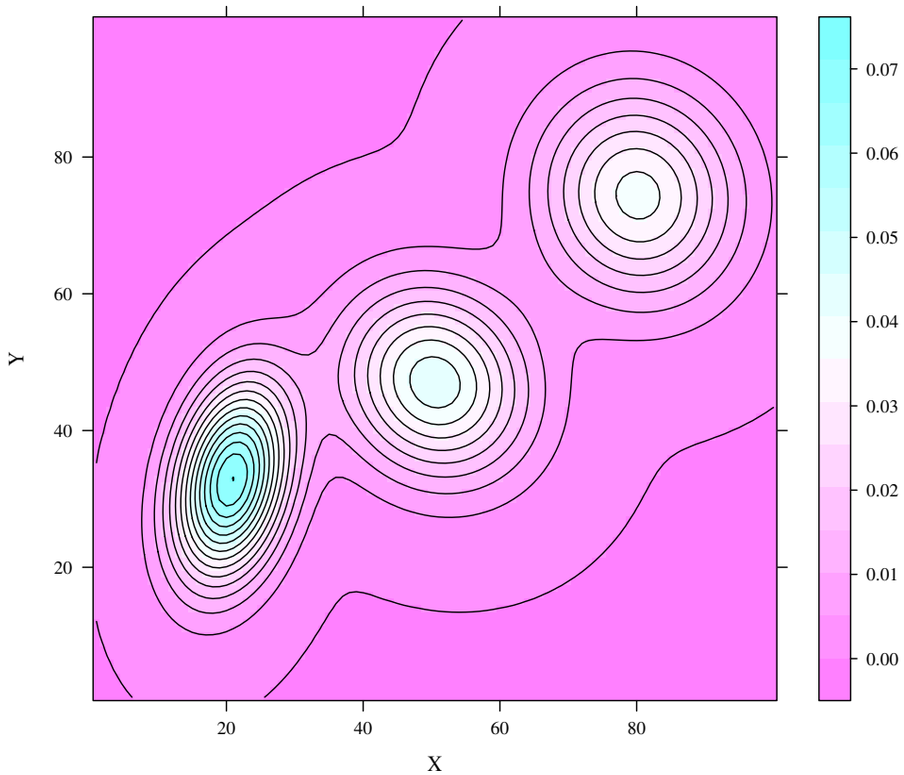
The result of applying knn



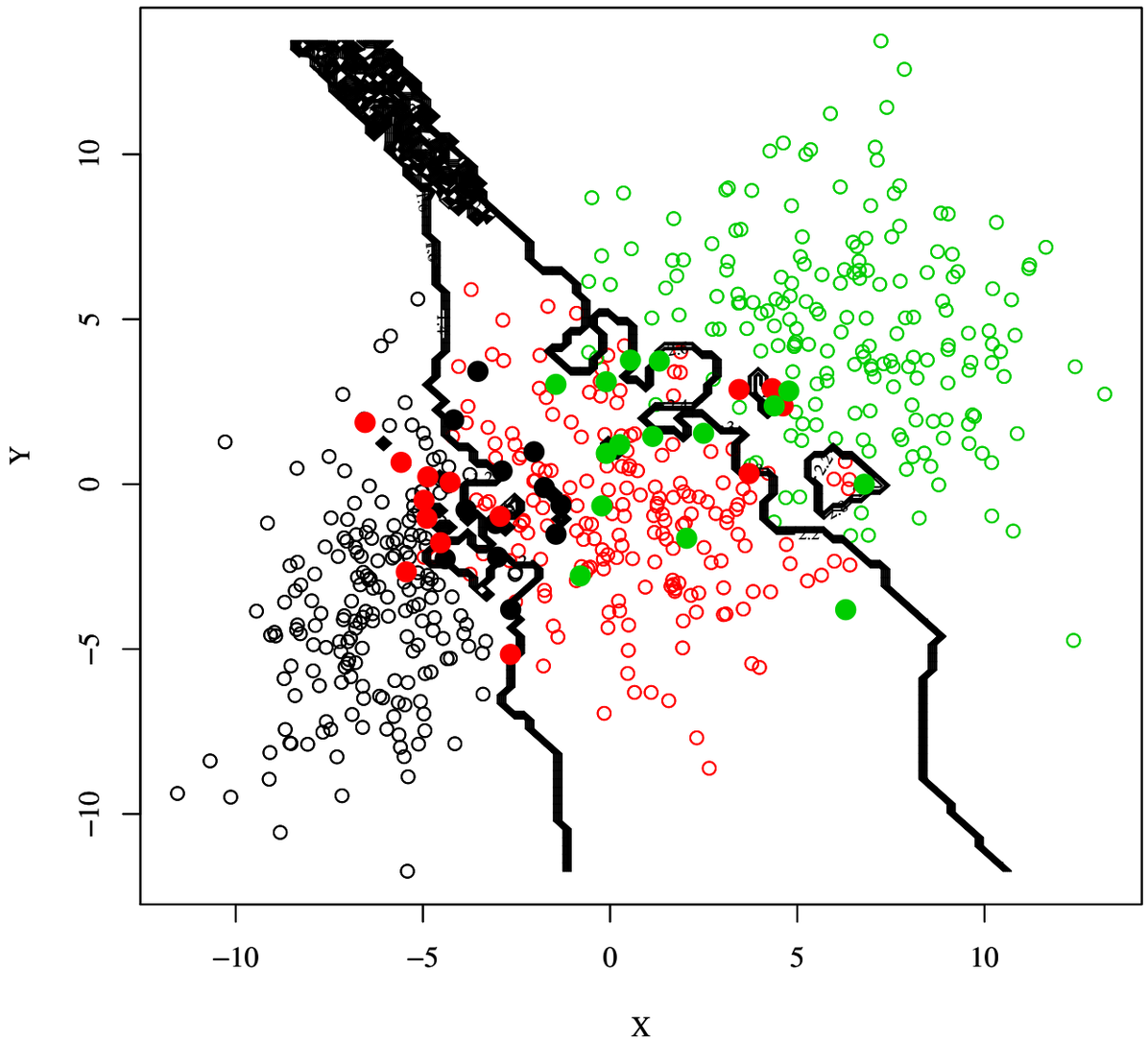
knn can lead to rather complex decision surfaces and is therefore not always optimal for **understanding** what is going on.

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.

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.

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!

k-nearest neighbours regression

How long does it take to run?

k-nearest neighbours regression

How long does it take to run?

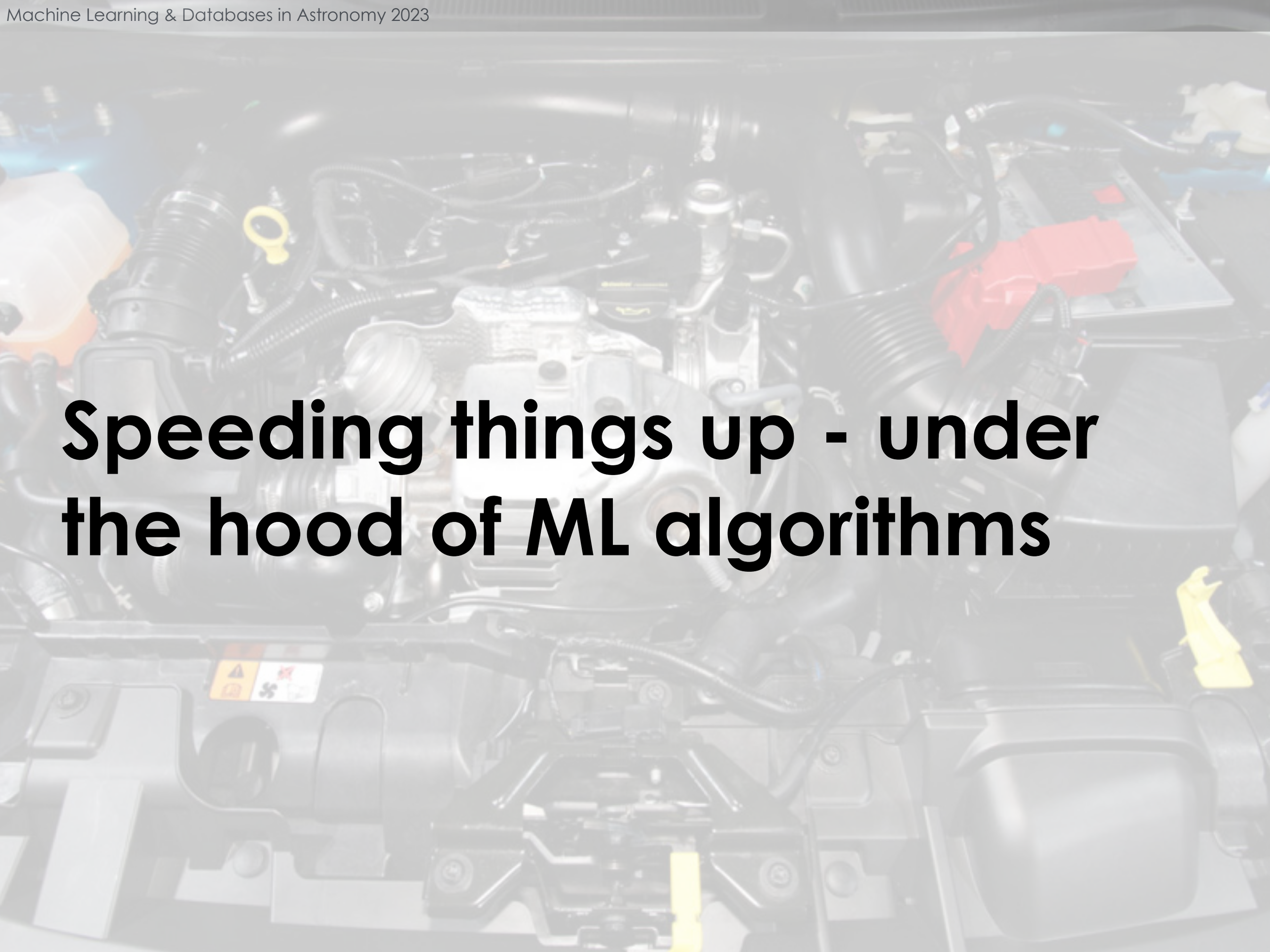
N^2

Problem 5 - 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.



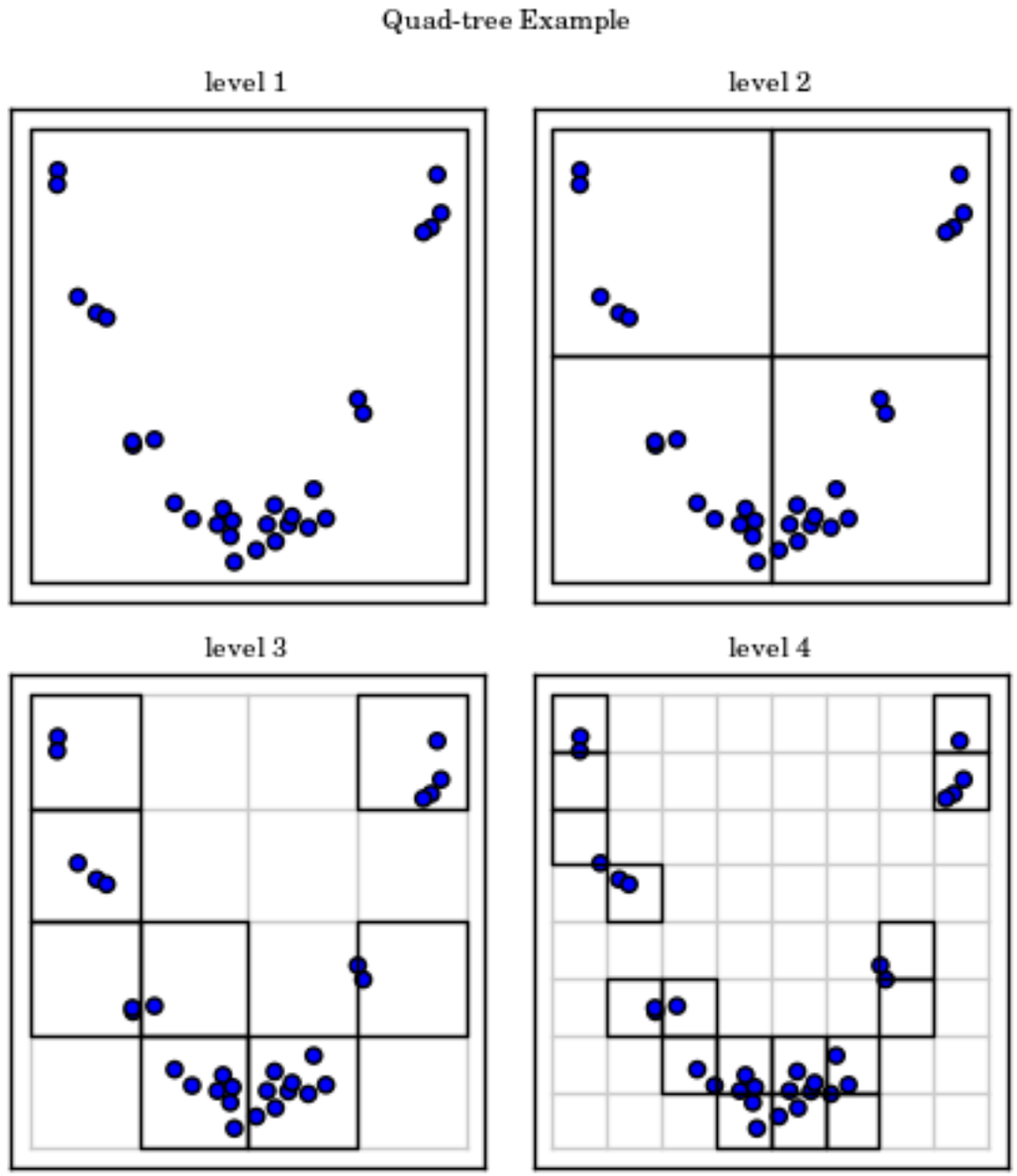
Speeding things up - under the hood of ML algorithms

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:

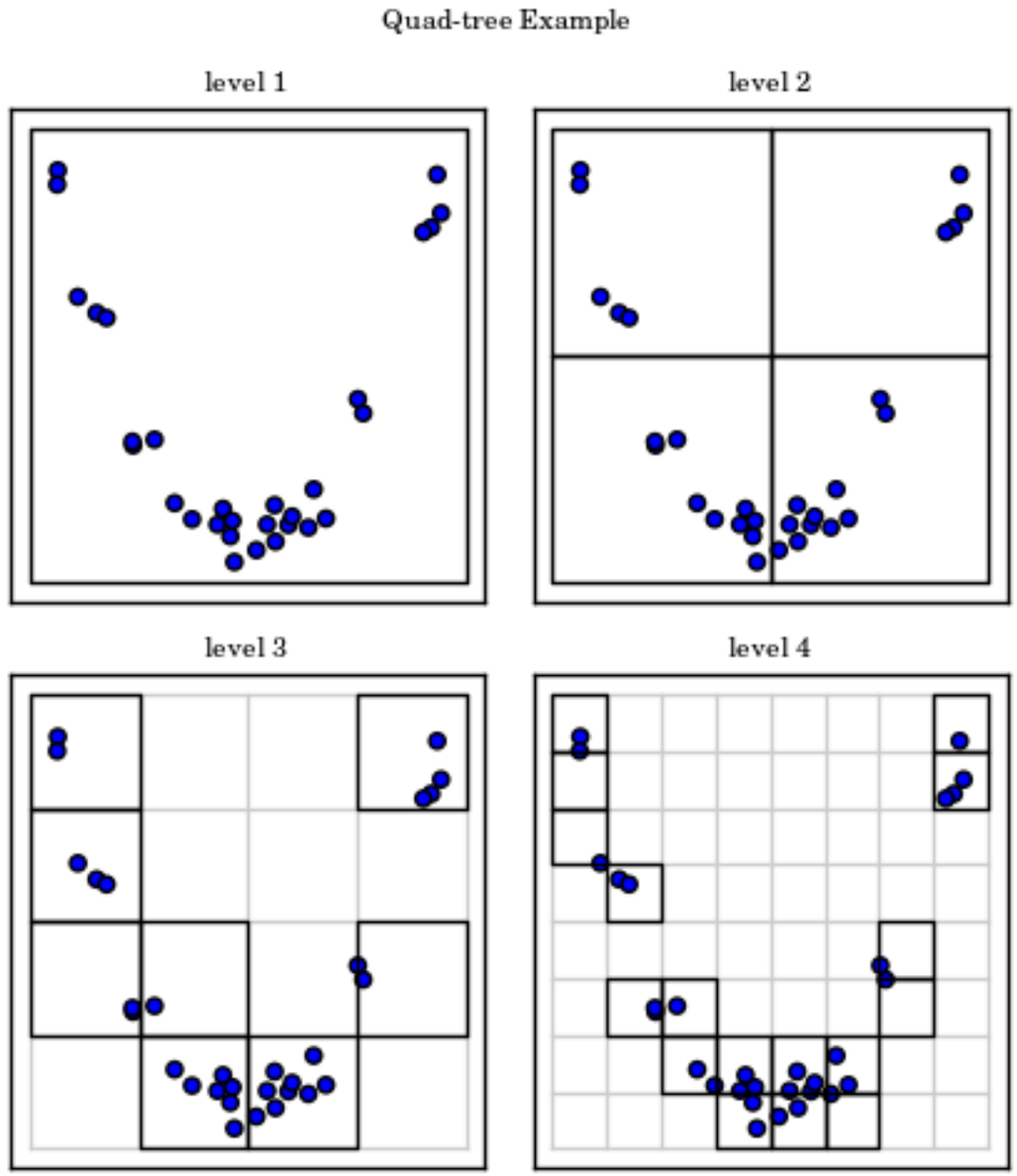
Extending the D dimensions is appealing but can be costly:

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

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)

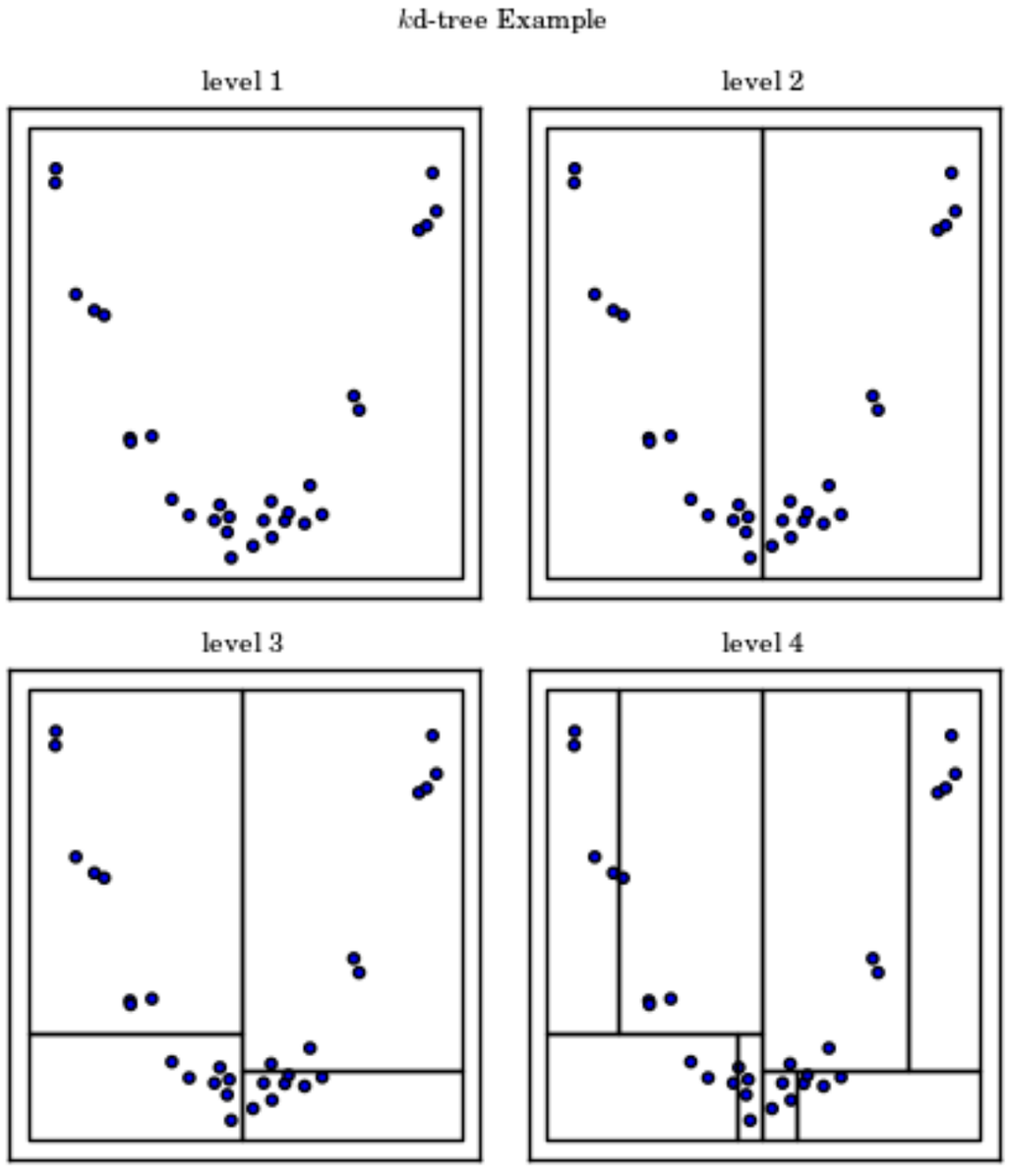
Finding neighbours - tactics

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

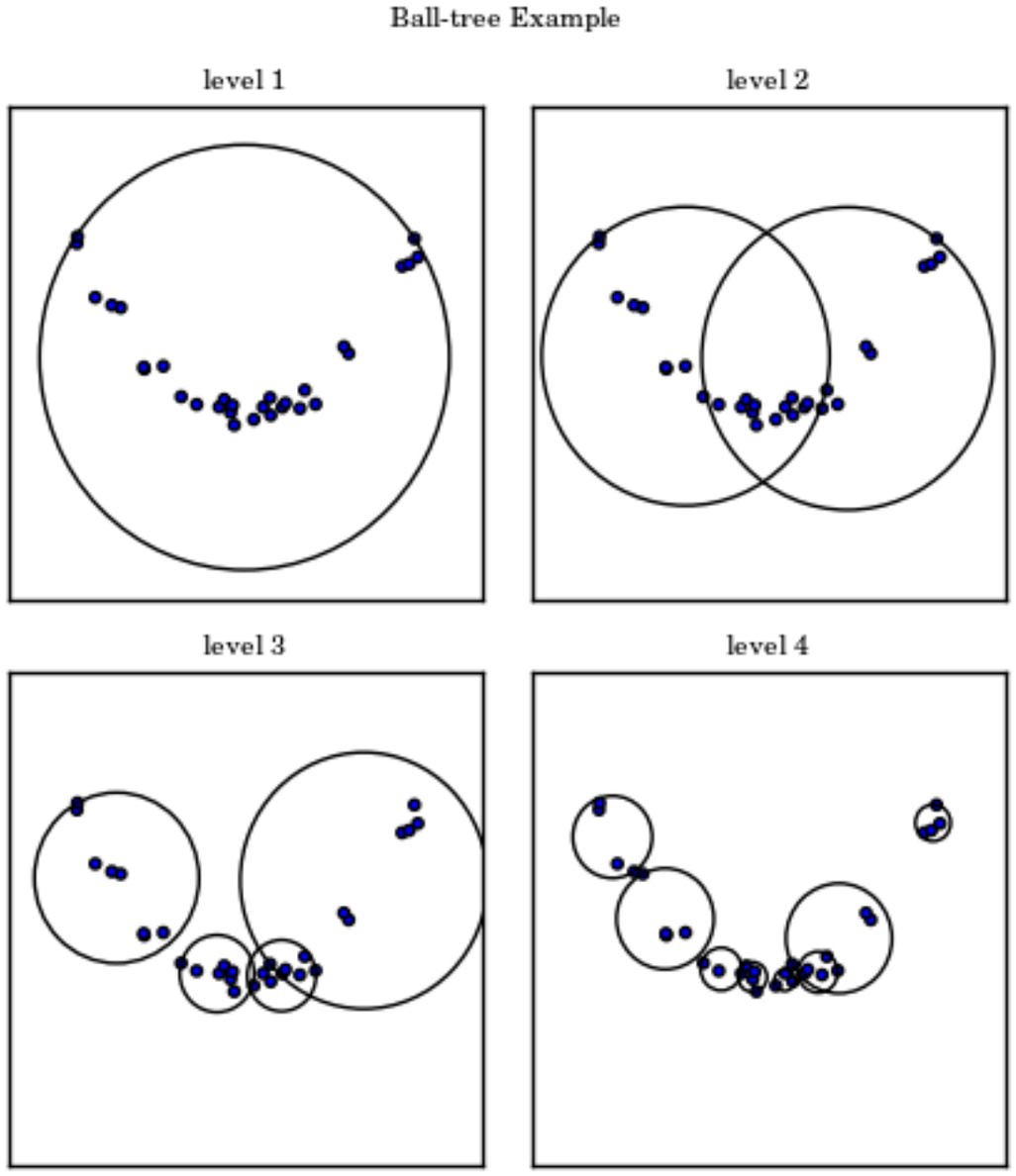


From Ivezic et al, “Statistics, Data Mining, and Machine Learning in Astronomy" (2013)

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)