Lecture 2 - Visualisations, density estimates and model choice

See https://github.com/jbrinchmann/MLD2024 as usual

Lectures/Lecture 2 has the PDF for today and

Lectures/Lecture 2/Notebooks for Jupyter notebooks used for this lecture

The super-brief review of last lecture

Manage your project using git:

git add git commit -m “I did something” git push

SQL is a language for querying databases - and it looks quite similar to English:

```
SELECT Star, g, r FROM Stars
   WHERE r < 14.5
```

When combining multiple tables we can JOIN them

```
SELECT s.Star, o.Field, o.Date
FROM Stars as s JOIN Observations as o
     ON s.fieldID = o.ID
WHERE s.Star = ‘S2’
```

The super-brief review of last lecture

Parameters are called θ, predicted values $\hat{y}$

We always need a way to say how good a model is - usually composed of the residual of the model minus data:

$$\mathrm{RSS} = \sum_i (y_i - \hat{y}_i)^2$$

Linear models are crucial building-blocks:

$$\hat{y}_i = \theta_0 + \sum_{j=1}^{p} \theta_j x_{ij} \qquad \Leftrightarrow \qquad Y = \mathsf{M}\theta$$

and if you have too few data you might want to regularise

$$\mathrm{RSS} + \lambda \sum_{j=1}^{p} |\theta_j| \qquad \text{(the LASSO)}$$

And now some words about Python & SQL

There are many ways to interact with databases in Python. I will cover two: an in-depth one and a simple one.

Not covered, but maybe of relevance:

MySQL interface for Python: MySQLdb
PostgresSQL with Python: psycopg2
Oracle with Python: cx_Oracle

There are also many places you can learn much more about this topic:

I quite like the compilation of links on Full Stack Python:
https://www.fullstackpython.com/databases.html

Google will give you many other options - it is a very popular topic…

And now some words about Python & SQL

There are many ways to interact with databases in Python. I will cover two: an in-depth one and a simple one.

I will assume that you have a database file already created for the following slides. If you do not, then you can either get ProblemSets/MakeTables/MLD2019.db from the GitHub site, or you can play with e.g. the places.sqlite database in your Firefox profile directory.

Python & sqlite3 the comprehensive way

```
import sqlite3 as lite
```

Load what is necessary

The database must be created first!

```
con = lite.connect(database)
```

Connect to database

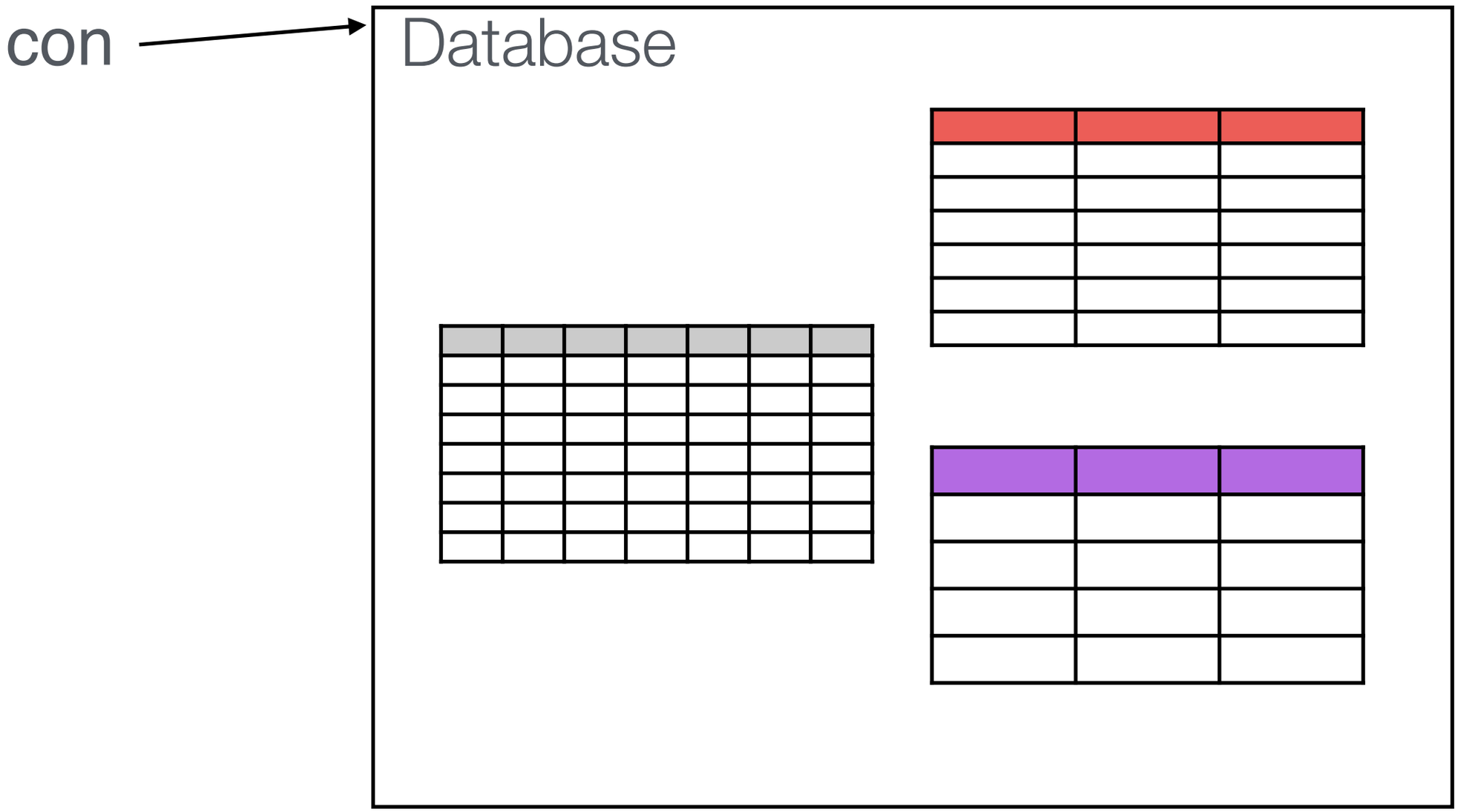
```
with con:

# Get a cursor.
  cur = con.cursor()

  # Execute commands
  cur.execute(command)
```

The Cursor object is one way to pass SQL statements to the database - I like to think about it as the finger reading through a table.

This is frequently seen - but it is not strictly necessary. I would recommend not creating an explicit cursor object like this for simplicity.

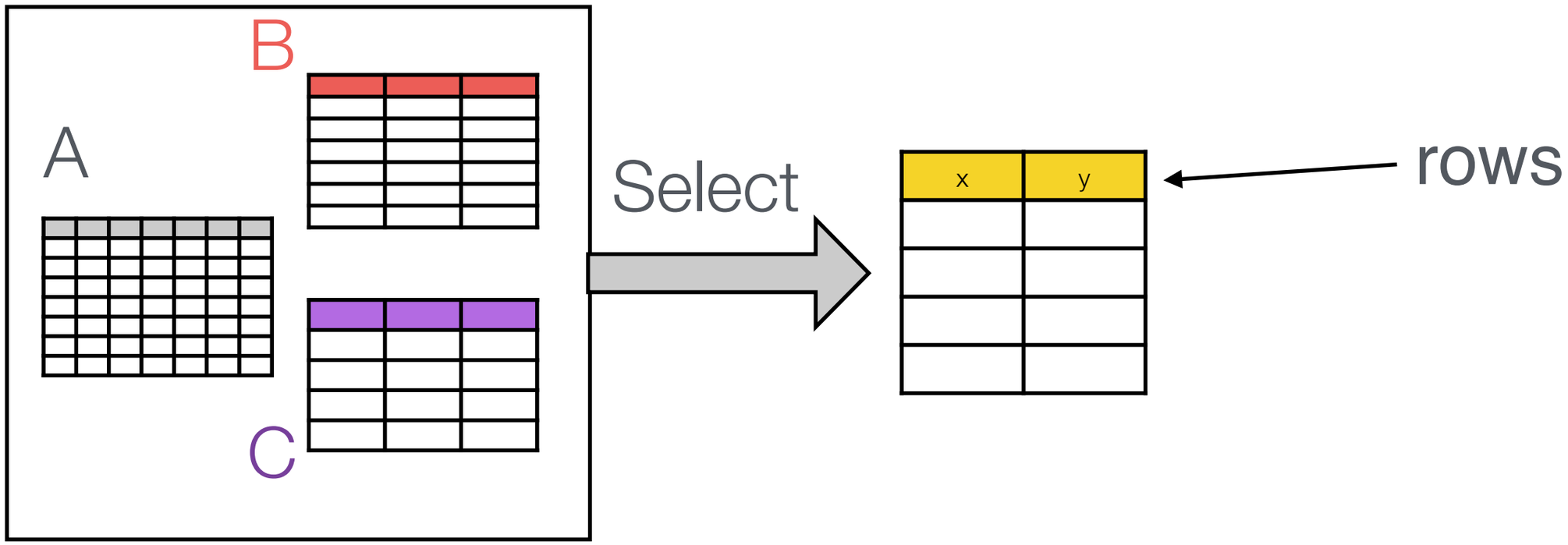
Python & SQL - the Cursor object

con = lite.connect(database)



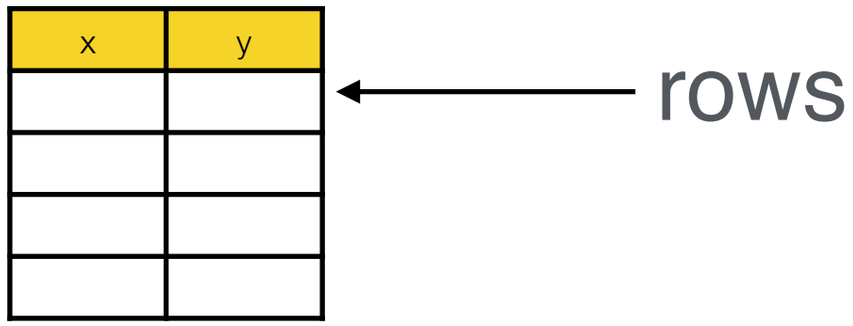
Using python to query the database:

```
In [1]: import sqlite3 as lite;

In [2]: con = lite.connect(“MLD2019.db")

In [4]: query = "select s.star, o.Date from stars as s JOIN observations as o ON
s.fieldID=o.ID"

In [3]: rows = con.execute(query)
```



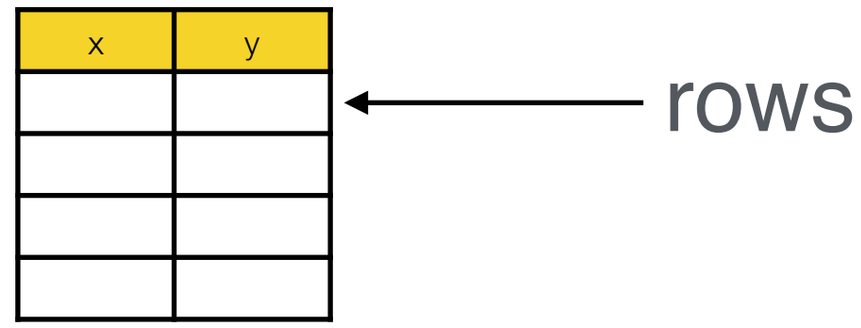
Using python to query the database:

```
In [4]: row = rows.fetchone()

In [5]: print row
Out[57]: ('S1', 92.9885764)
```

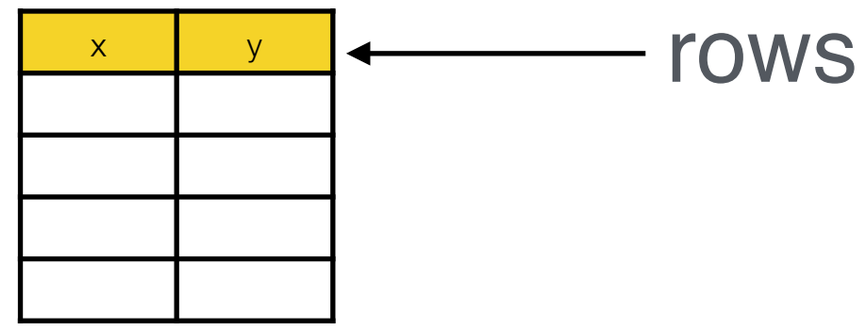


Using python to query the database:

```
In [4]: row = rows.fetchone()

In [5]: print row
Out[5]: ('S1', 92.9885764)
```

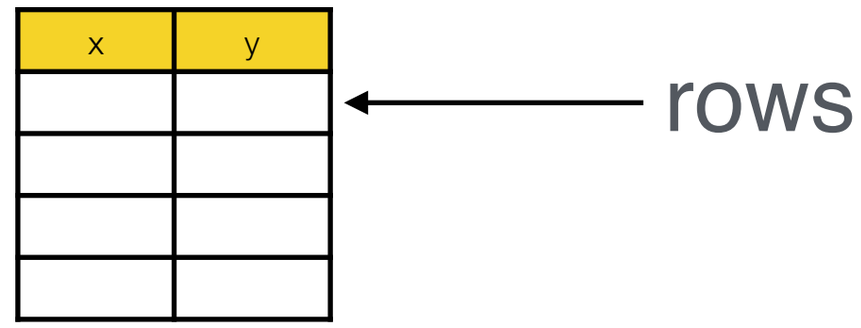



```
In [6]: for row in rows:
   ....:     print(row)
   ....:
('S2', 92.9885764)
('S5', 93.5532134)
('S7', 97.3323764)
```

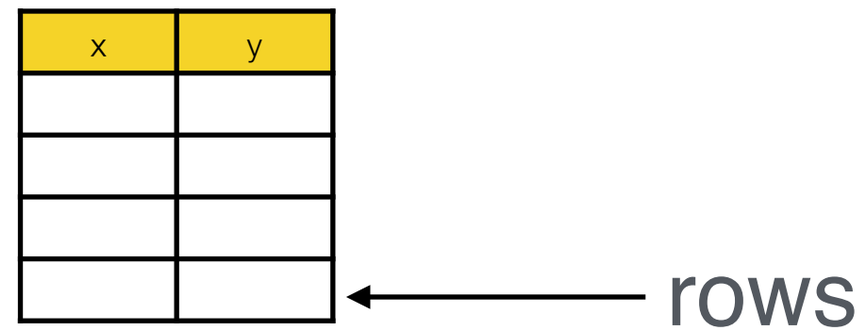


Using python to query the database:

```
In [4]: row = rows.fetchone()

In [5]: print row
Out[5]: ('S1', 92.9885764)
```

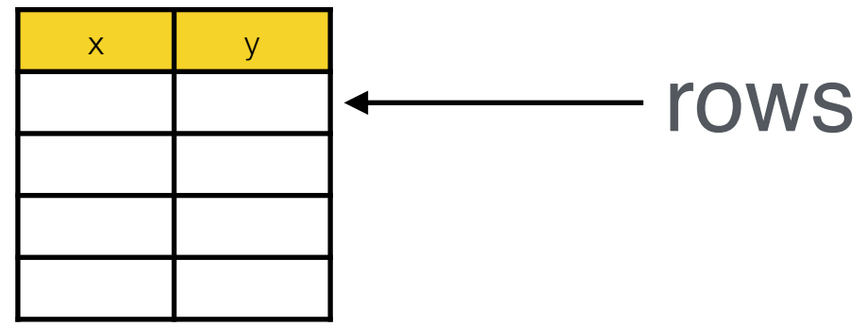



```
In [6]: for row in rows:
   ....:     print(row)
   ....:
('S2', 92.9885764)
('S5', 93.5532134)
('S7', 97.3323764)
```

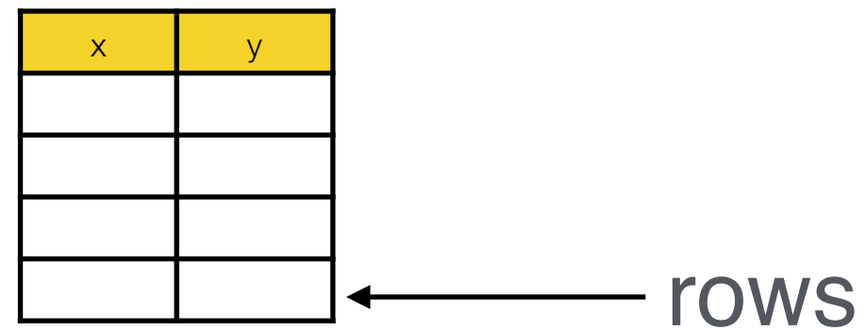


Using python to query the database:

```
In [4]: row = rows.fetchone()

In [5]: print row
Out[5]: ('S1', 92.9885764)
```




```
In [6]: for row in rows:
   ....:     print(row)
   ....:
('S2', 92.9885764)
('S5', 93.5532134)
('S7', 97.3323764)
```



Watch out!

```
In [7]: row = rows.fetchone()
In [8]: row
```

Because we used up all rows in the previous call! If you use cursors this would lead row to contain the last row - a bug waiting to happen.

Using python to query the database - recommended:

```
In [1]: import sqlite3 as lite;

In [2]: con = lite.connect("MLD2019.db")

In [3]: rows = con.execute('SELECT ra, decl FROM Stars')

In [4]: for row in rows:
   ...:     print("Ra={0}  Dec={1}".format(row[0], row[1]))
   ...:
Ra=198.8475  Dec=10.5034722
Ra=198.5654167  Dec=11.0231944
Ra=198.9370833  Dec=9.9168889
Ra=199.2516667  Dec=10.3486944
```

Another possibility (not good for large datasets!!)

```
cur = con.execute('SELECT ra, decl FROM Stars')
rows = cur.fetchall()
```

The simplest(?) way of dealing with tables in Python

pandas:

```
In [35]: import pandas as pd

In [36]: con = lite.connect("MLD2019.db")

In [37]: t = pd.read_sql_query('Select ra, decl from Stars', con)

In [38]: t
Out[38]:
          ra       decl
0  198.847500  10.503472
1  198.565417  11.023194
2  198.937083   9.916889
3  199.251667  10.348694
```

The simplest(?) way of dealing with tables in Python

pandas:

```
In [35]: import pandas as pd

In [36]: con = lite.connect("MLD2019.db")

In [37]: t = pd.read_sql_query('Select ra, decl from Stars', con)

In [38]: t
Out[38]:
          ra       decl
0  198.847500  10.503472
1  198.565417  11.023194
2  198.937083   9.916889
3  199.251667  10.348694
```

and it is easy to make new tables:

```
In [39]: t.to_sql('Stars2', con)
```

That's it.

pandas vs normal sqlite

```
# Next, we create a connection to the database.
con = lite.connect(database)


with con:

        table = "Stars"
    # Create the command to create the table. I use a
    # multiline string to ease readability here.
    command = """CREATE TABLE IF NOT EXISTS {0} (StarID INT,
                                    FieldID INT, Star varchar(10), ra DOUBLE,
                                    decl DOUBLE, g FLOAT, r FLOAT,
                                    UNIQUE(StarID), PRIMARY KEY(StarID),
                                    FOREIGN KEY(FieldID) REFERENCES Observations(ID))""".format(table)

    # Next, actually execute this command.
    con.execute(command)

    # Now that this is working, let us loop over the table entries
    # and insert these into the table.
    for row in cat:
        command = "INSERT INTO Stars VALUES({0},{1},'{2}',{3},{4},{5},{6})".format(row[0], row[1], row[2], row[3], row[4], row[5], row[6])
        print command
        con.execute(command)
```

```
In [39]: t.to_sql('Stars2', con)
```

So why not only pandas?

pandas vs normal sqlite

```
# Next, we create a connection to the database.
con = lite.connect(database)


with con:

        table = "Stars"
    # Create the command to create the table. I use a
    # multiline string to ease readability here.
    command = """CREATE TABLE IF NOT EXISTS {0} (StarID INT,
                                    FieldID INT, Star varchar(10), ra DOUBLE,
                                    decl DOUBLE, g FLOAT, r FLOAT,
                                    UNIQUE(StarID), PRIMARY KEY(StarID),
                                    FOREIGN KEY(FieldID) REFERENCES Observations(ID))""".format(table)

    # Next, actually execute this command.
    con.execute(command)

    # Now that this is working, let us loop over the table entries
    # and insert these into the table.
    for row in cat:
        command = "INSERT INTO Stars VALUES({0},{1},'{2}',{3},{4},{5},{6})".format(row[0], row[1], row[2], row[3], row[4], row[5], row[6])
        print command
        con.execute(command)
```

```
In [39]: t.to_sql('Stars2', con)
```

So why not only pandas?

★ Control
★ Memory usage

Visualising data

Making good scientific plots

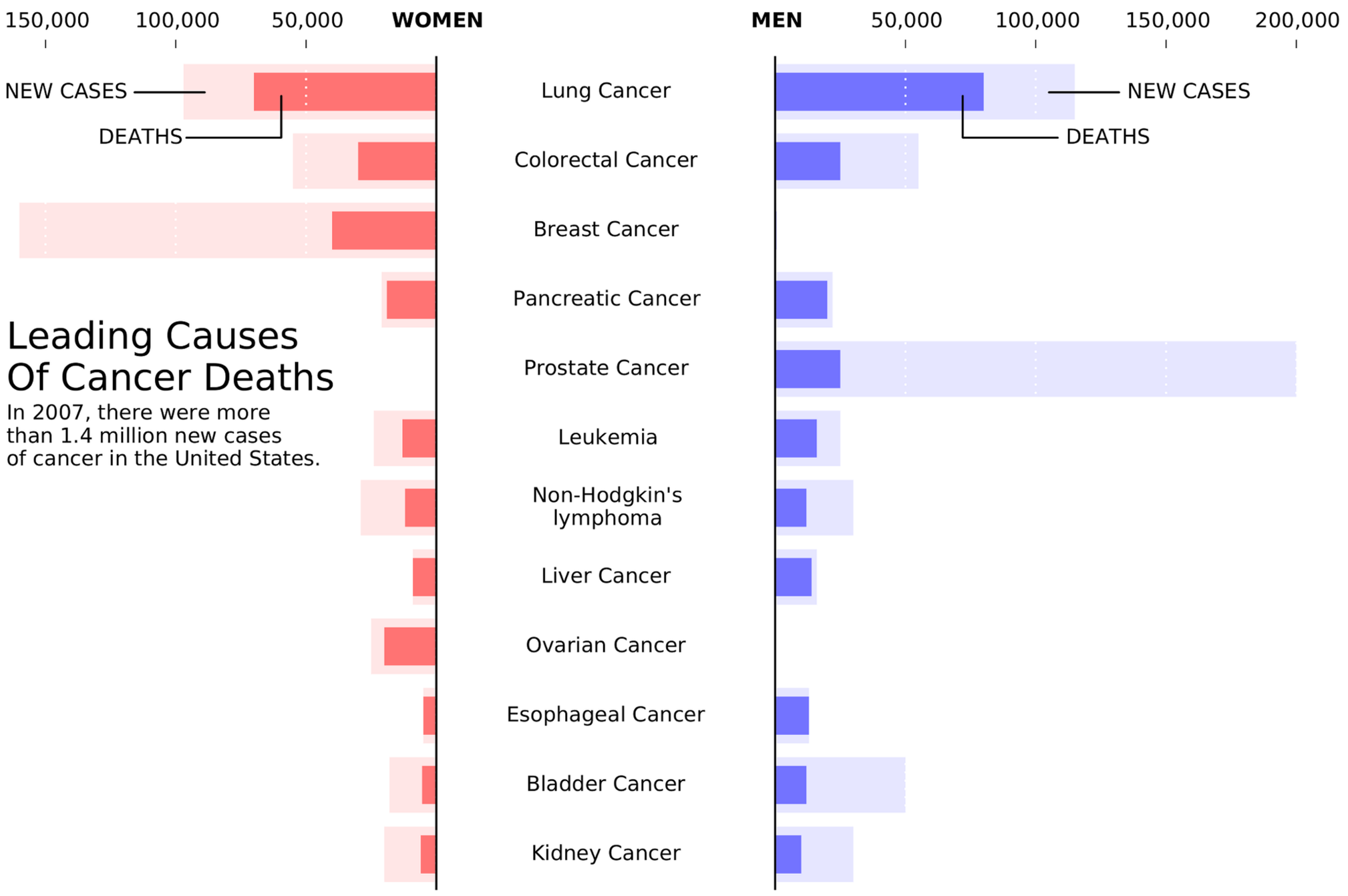
- Know the audience.
- Adapt to the situation.
- Be precise, include all information needed.
- Do not mislead.
- Do not use colour without thinking. Especially bad colour choices.
- Do not overload a figure - or include too little!
- Learn a couple of packages well.

Worth checking out: Rougier NP, Droettboom M, Bourne PE (2014) Ten Simple Rules for Better Figures. PLoS Comput Biol 10(9): e1003833.

http://journals.plos.org/ploscompbiol/article?id=10.1371/journal.pcbi.1003833

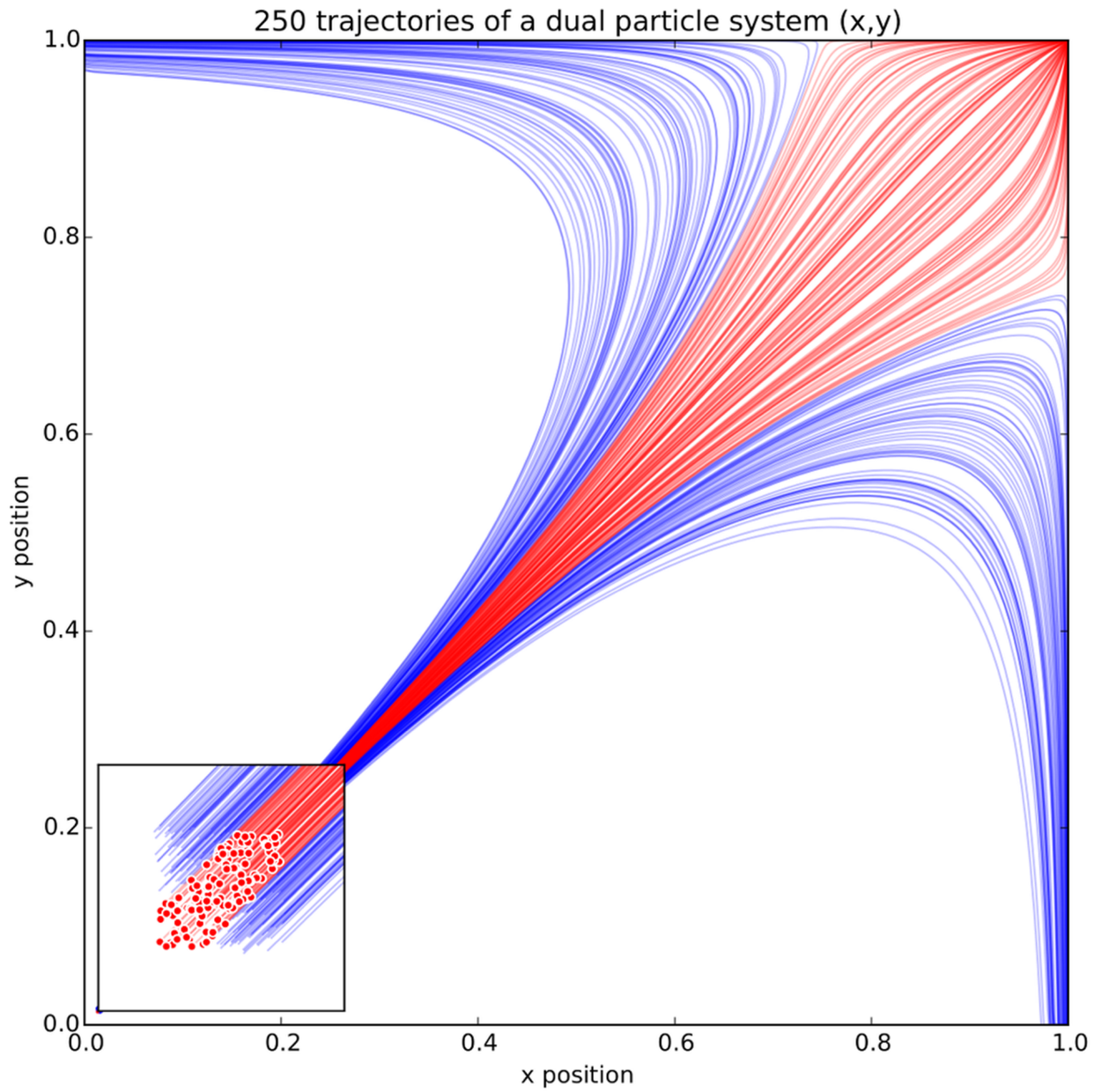
Adapting to the audience

from: Rougier NP, Droettboom M, Bourne PE (2014) - an example of a newspaper article



Adapting to the situation

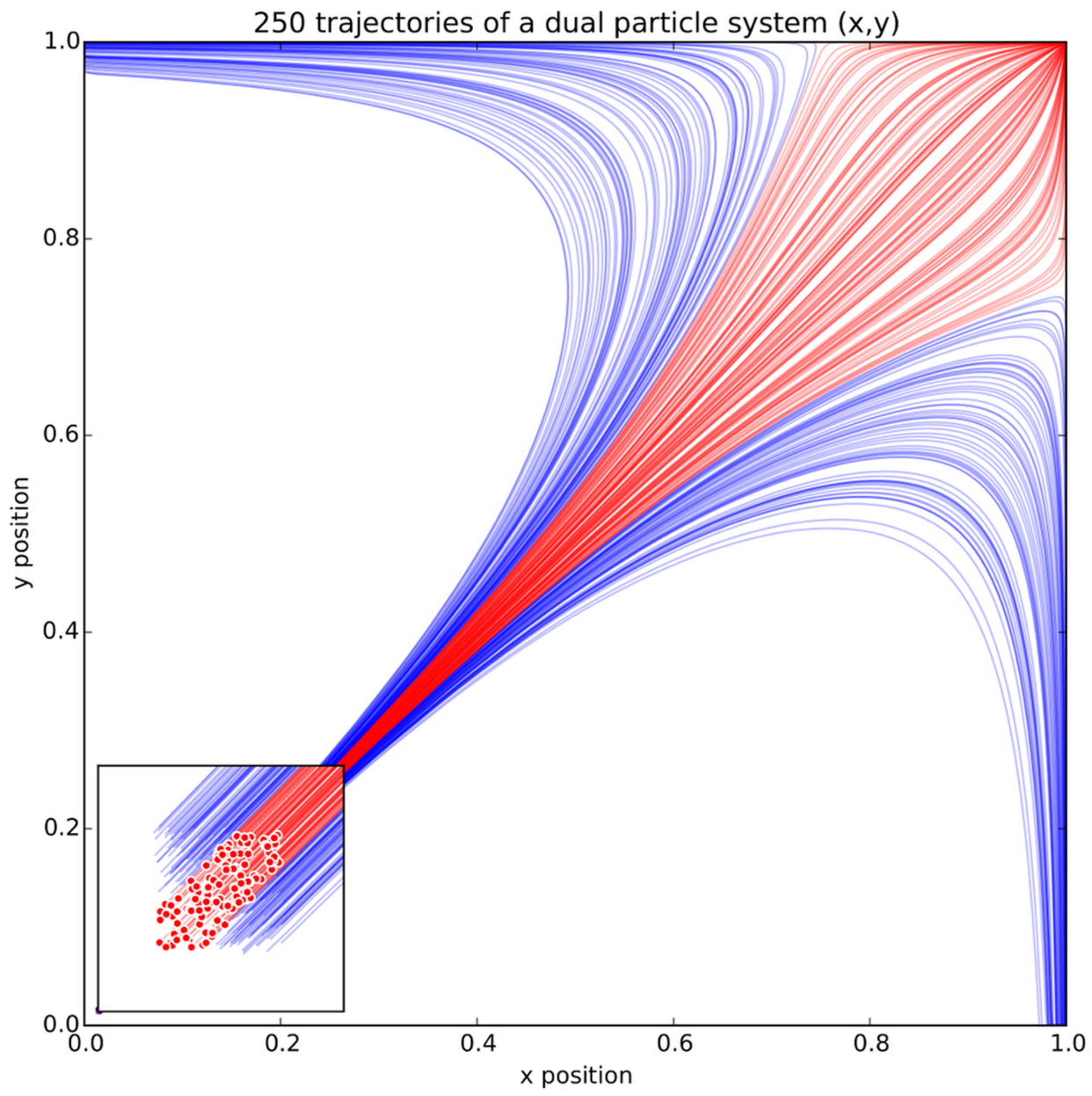
from: Rougier NP, Droettboom M, Bourne PE (2014)



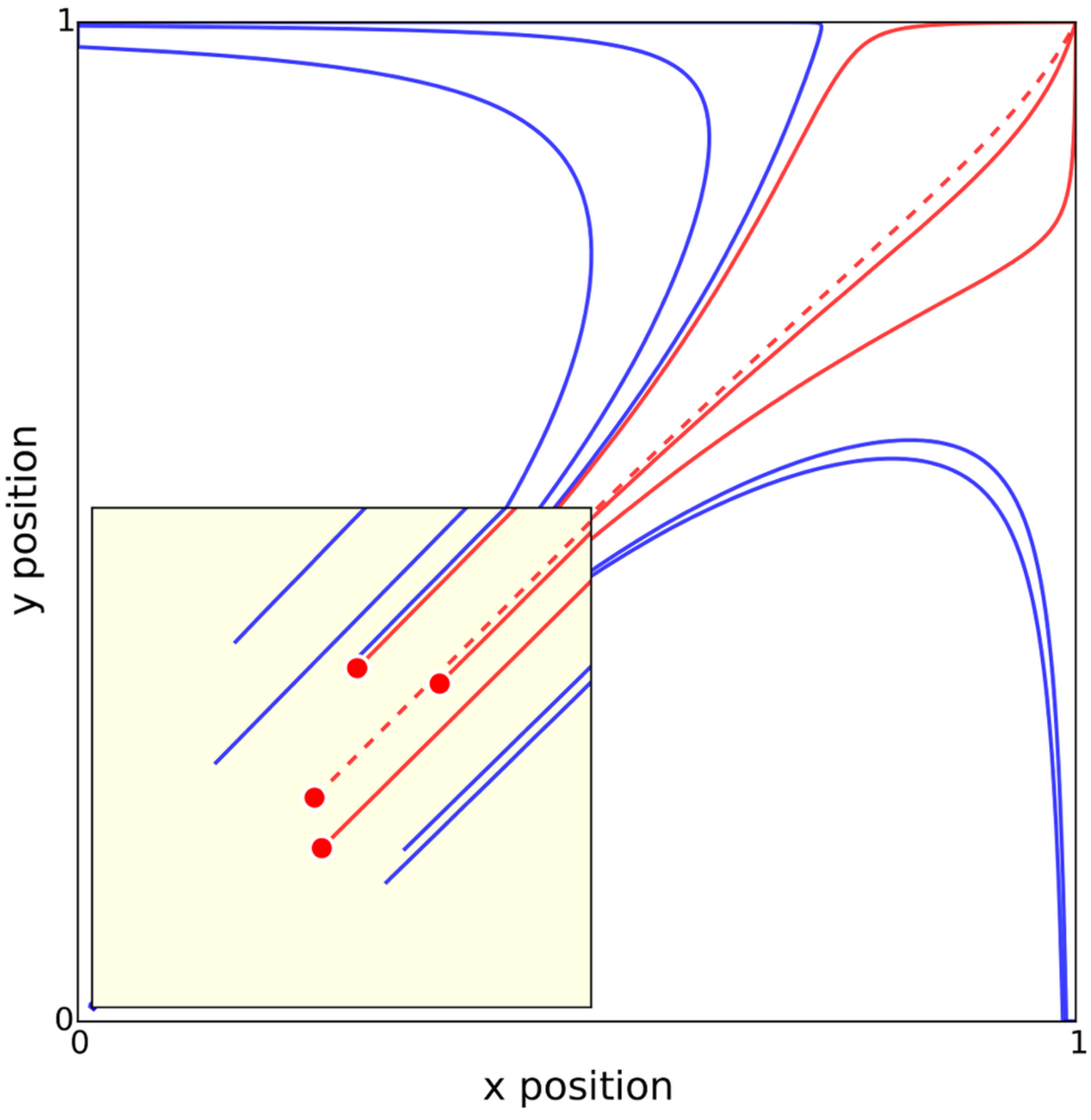
Scientific paper

Adapting to the situation

from: Rougier NP, Droettboom M, Bourne PE (2014)



Scientific paper



Presentation

Why do we create visualisations (plots)?

Why do we create visualisations (plots)?

To help our argumentation.

Why do we create visualisations (plots)?

To help our argumentation.

To summarise a lot of data

Why do we create visualisations (plots)?

To help our argumentation.

To summarise a lot of data

To discover new relationships

Why do we create visualisations (plots)?

To help our argumentation.

To summarise a lot of data

To discover new relationships

To relate different pieces of information

Why do we create visualisations (plots)?

To help our argumentation.

To summarise a lot of data

To discover new relationships

To relate different pieces of information

To develop intuition

A quick interlude - try out:

- Get the file Datafiles/several_datasets.tsv from the course website.
- Read it in using e.g. astropy:

```
from astropy.table import Table
t = Table().read("several_datasets.tsv",
format="ascii.fast_tab")
```

There are 13 datasets there - t['x1'], t['y1'] contains the x & y values for dataset 1 etc.

Calculate summary statistics (mean, standard deviation, correlation coefficient, maybe a linear fit) for each dataset - what do you conclude?

How do you do it?

Looking at odd data

x	y	x	y	x	y	x	y
10.0	8.04	10.0	9.14	10.0	7.46	8.0	6.58
8.0	6.95	8.0	8.14	8.0	6.77	8.0	5.76
13.0	7.58	13.0	8.74	13.0	12.74	8.0	7.71
9.0	8.81	9.0	8.77	9.0	7.11	8.0	8.84
11.0	8.33	11.0	9.26	11.0	7.81	8.0	8.47
14.0	9.96	14.0	8.10	14.0	8.84	8.0	7.04
6.0	7.24	6.0	6.13	6.0	6.08	8.0	5.25
4.0	4.26	4.0	3.10	4.0	5.39	19.0	12.50
12.0	10.84	12.0	9.13	12.0	8.15	8.0	5.56
7.0	4.82	7.0	7.26	7.0	6.42	8.0	7.91
5.0	5.68	5.0	4.74	5.0	5.73	8.0	6.89

An example of *odd* data

x	y	x	y	x	y	x	y
10.0	8.04	10.0	9.14	10.0	7.46	8.0	6.58
8.0	6.95	8.0	8.14	8.0	6.77	8.0	5.76
13.0	7.58	13.0	8.74	13.0	12.74	8.0	7.71
9.0	8.81	9.0	8.77	9.0	7.11	8.0	8.84
11.0	8.33	11.0	9.26	11.0	7.81	8.0	8.47
14.0	9.96	14.0	8.10	14.0	8.84	8.0	7.04
6.0	7.24	6.0	6.13	6.0	6.08	8.0	5.25
4.0	4.26	4.0	3.10	4.0	5.39	19.0	12.50
12.0	10.84	12.0	9.13	12.0	8.15	8.0	5.56
7.0	4.82	7.0	7.26	7.0	6.42	8.0	7.91
5.0	5.68	5.0	4.74	5.0	5.73	8.0	6.89

An example of *odd* data

x	y	x	y	x	y	x	y
10.0	8.04	10.0	9.14	10.0	7.46	8.0	6.58
8.0	6.95	8.0	8.14	8.0	6.77	8.0	5.76
13.0	7.58	13.0	8.74	13.0	12.74	8.0	7.71
9.0	8.81	9.0	8.77	9.0	7.11	8.0	8.84
11.0	8.33	11.0	9.26	11.0	7.81	8.0	8.47
14.0	9.96	14.0	8.10	14.0	8.84	8.0	7.04
6.0	7.24	6.0	6.13	6.0	6.08	8.0	5.25
4.0	4.26	4.0	3.10	4.0	5.39	19.0	12.50
12.0	10.84	12.0	9.13	12.0	8.15	8.0	5.56
7.0	4.82	7.0	7.26	7.0	6.42	8.0	7.91
5.0	5.68	5.0	4.74	5.0	5.73	8.0	6.89

mean(x) = **9**, mean(y) = **7.5**

Var(x) = **11**, Var(y) = **4.12**

The correlation coefficient: **0.816**

The best-fit line: **y = 3 +0.5 x**

An example of *odd* data

x	y	x	y	x	y	x	y
10.0	8.04	10.0	9.14	10.0	7.46	8.0	6.58
8.0	6.95	8.0	8.14	8.0	6.77	8.0	5.76
13.0	7.58	13.0	8.74	13.0	12.74	8.0	7.71
9.0	8.81	9.0	8.77	9.0	7.11	8.0	8.84
11.0	8.33	11.0	9.26	11.0	7.81	8.0	8.47
14.0	9.96	14.0	8.10	14.0	8.84	8.0	7.04
6.0	7.24	6.0	6.13	6.0	6.08	8.0	5.25
4.0	4.26	4.0	3.10	4.0	5.39	19.0	12.50
12.0	10.84	12.0	9.13	12.0	8.15	8.0	5.56
7.0	4.82	7.0	7.26	7.0	6.42	8.0	7.91
5.0	5.68	5.0	4.74	5.0	5.73	8.0	6.89

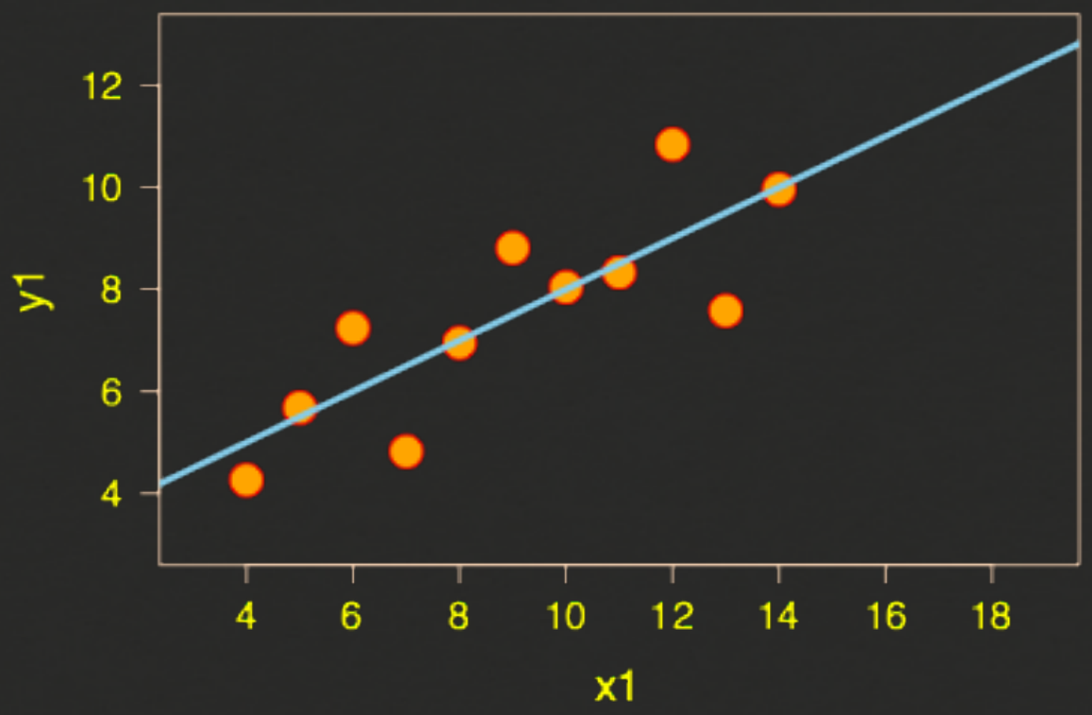
mean(x) = **9**, mean(y) = **7.5**

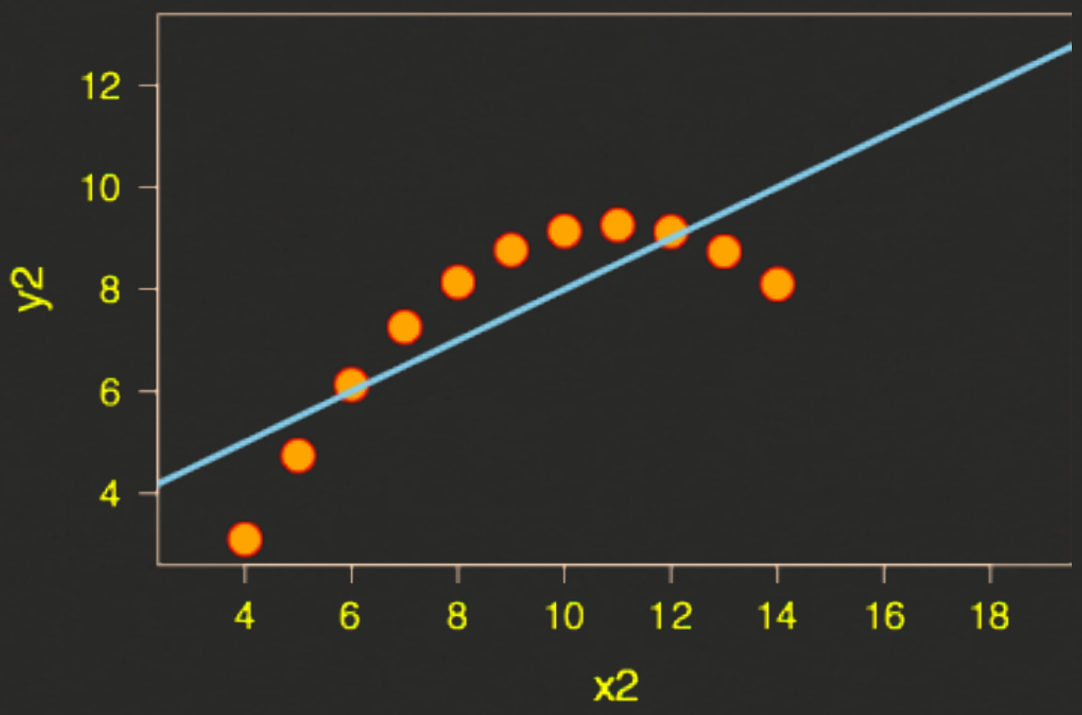
Var(x) = **11**, Var(y) = **4.12**

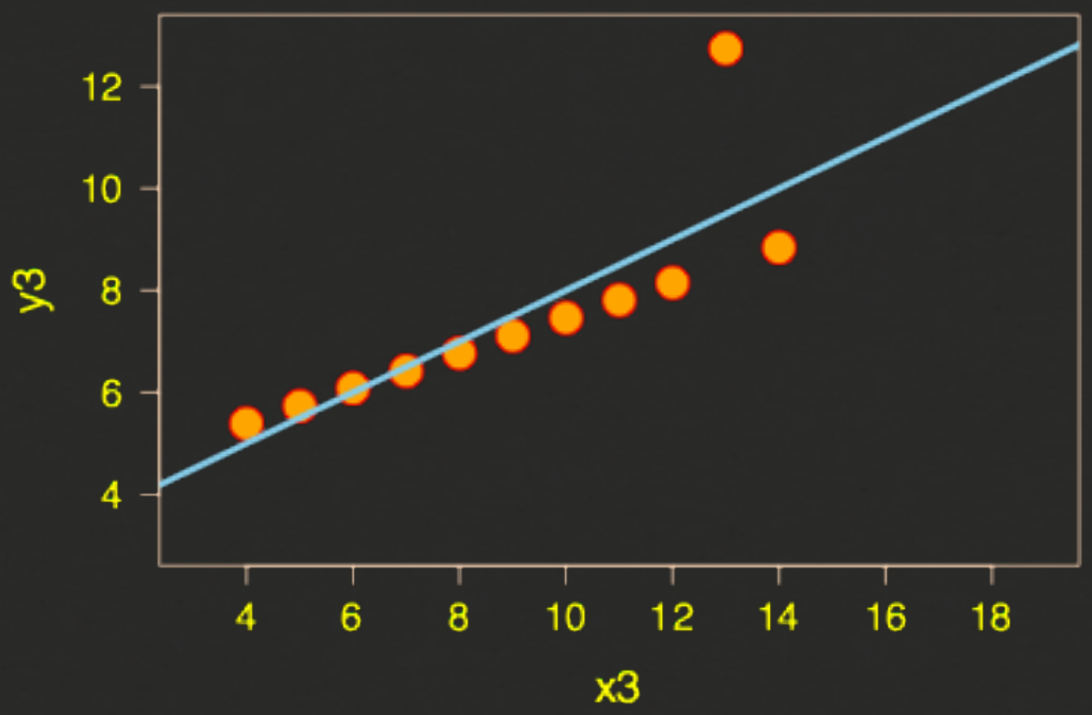
The correlation coefficient: **0.816**

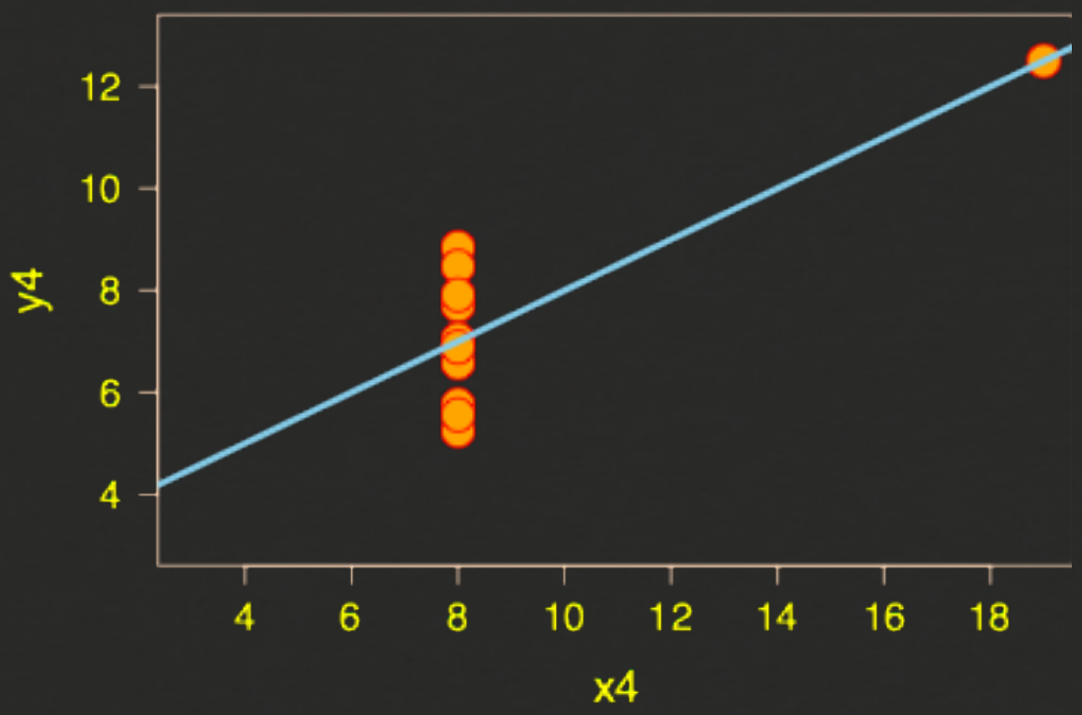
The best-fit line: **y = 3 +0.5 x**

So they seem to be very similar!









The bottom line: Always plot your data! (but you knew that already I hope!)
But how would you deal with this in N dimensions?

Graphical displays should:

- Show the data.
- Induce the viewer to think about the substance.
- Avoid distorting what the data have to say.
- Present many numbers in a small space.
- Make large data sets coherent.
- Encourage the eye to compare different pieces of data.
- Reveal data at several layers of detail.
- Be relevant.
- **Show units, show scales!**

Based on: Tufte: "The Visual Display of Quantitative Information"

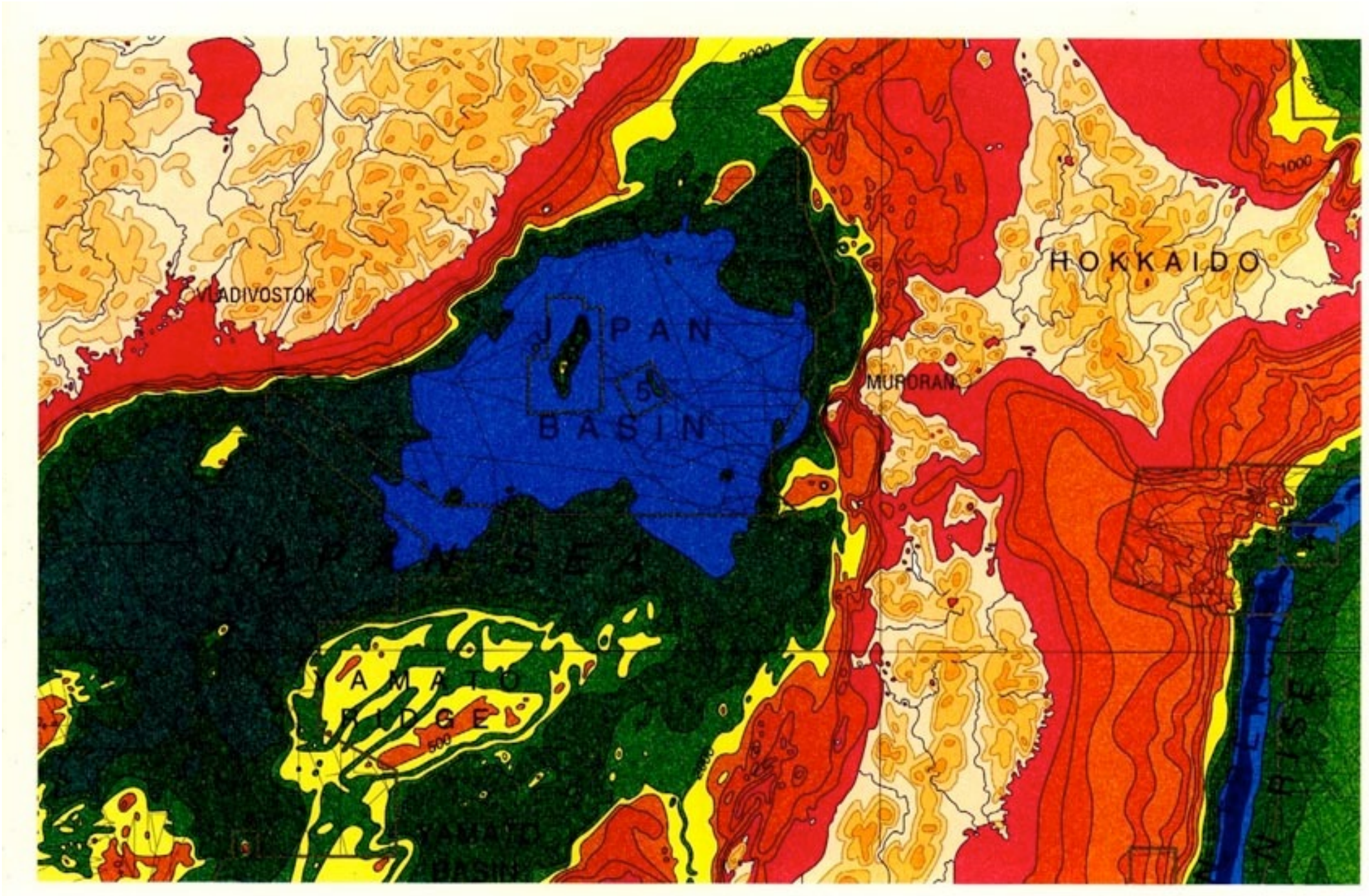
Colour choice

The “Smallest effective difference”

From Edward Tufte (1997), “Visual explanations”

Colour choice

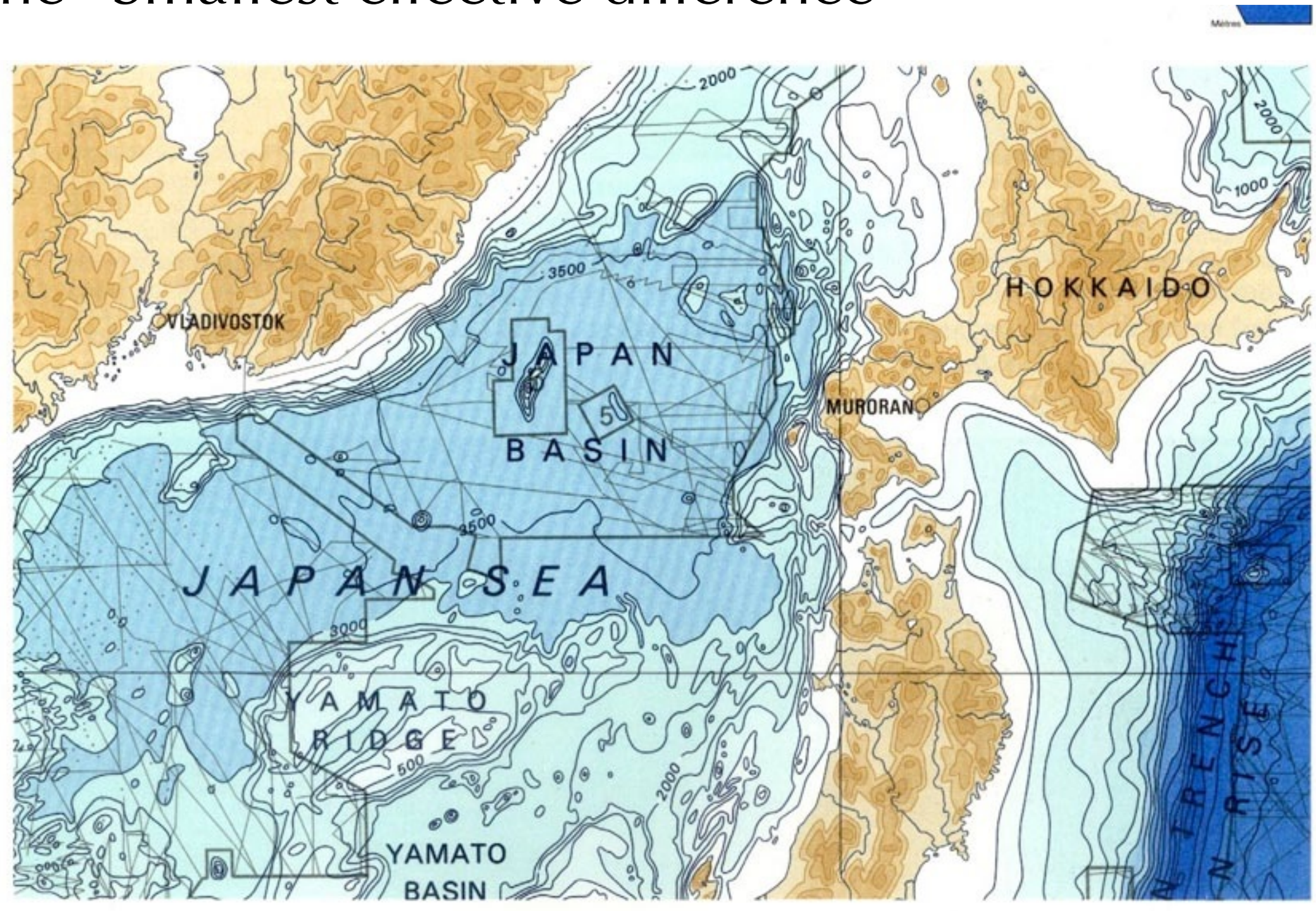
The “Smallest effective difference”



From Edward Tufte (1997), “Visual explanations”

Colour choice

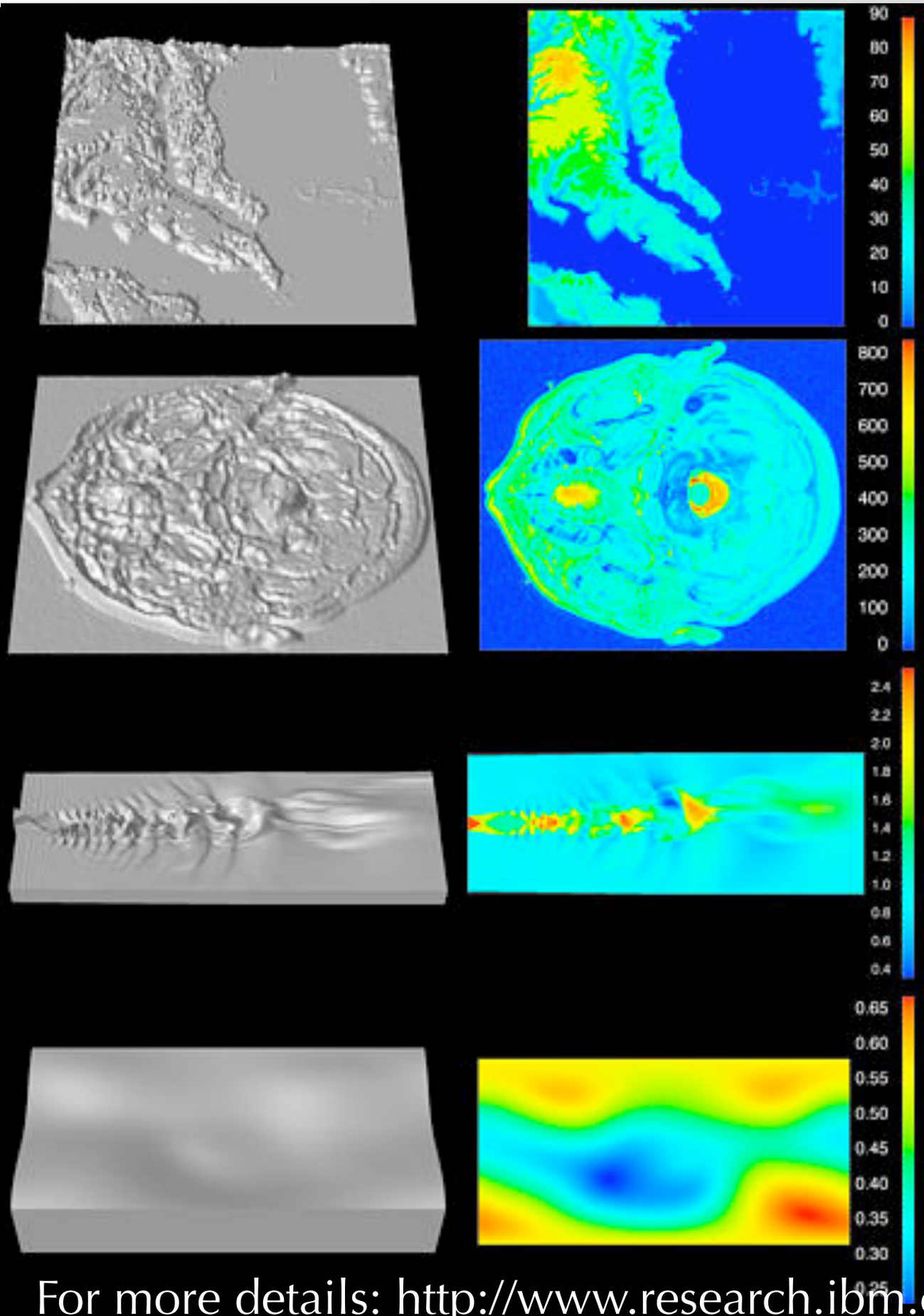
The “Smallest effective difference”



From Edward Tufte (1997), “Visual explanations”

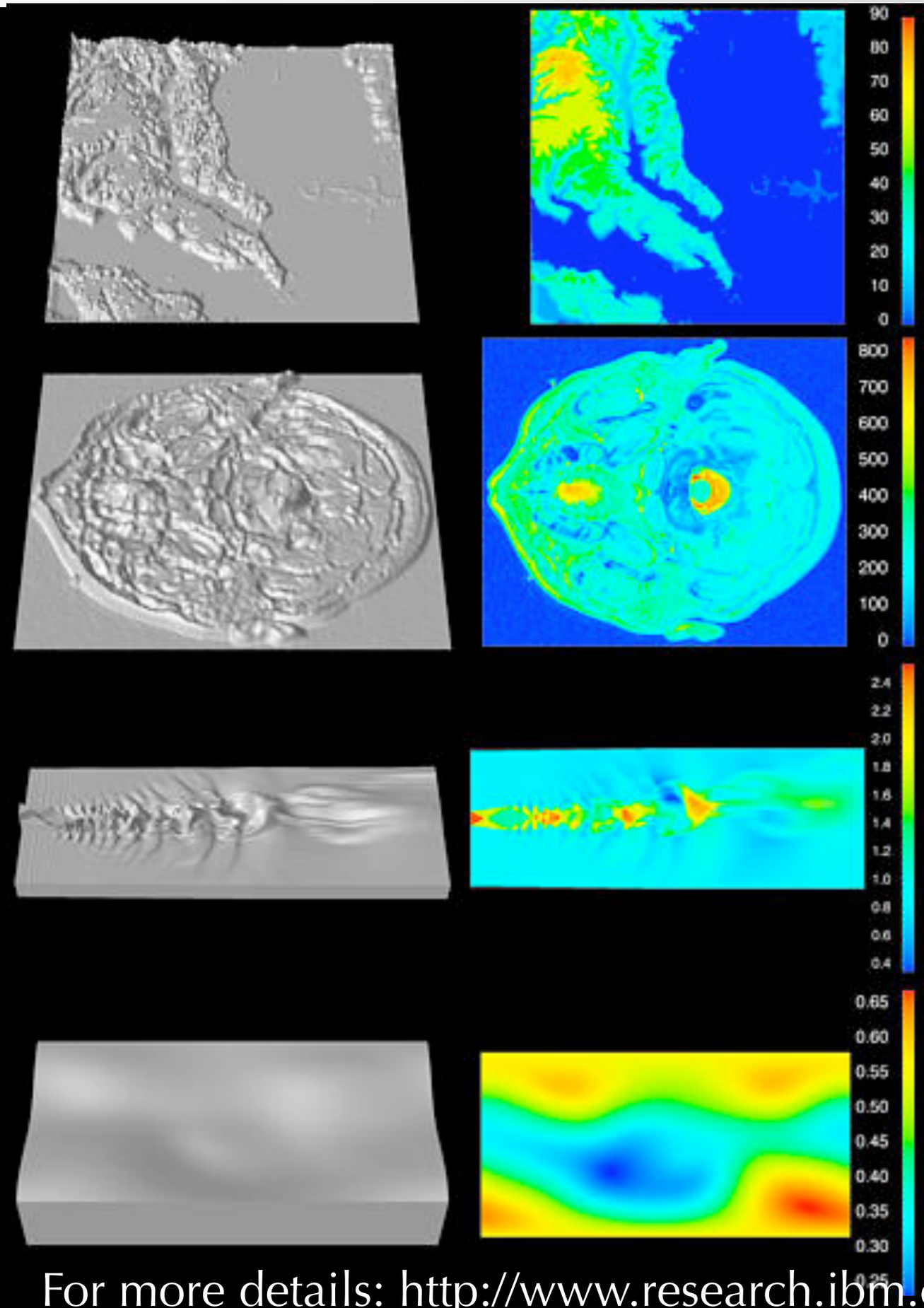
The issue of colour & 2D plots

There is a tendency for astronomers to choose from a small set of colour schemes for their images and plots.
Sometimes this leads you to create artificial trends where there are none.



Bad Habits

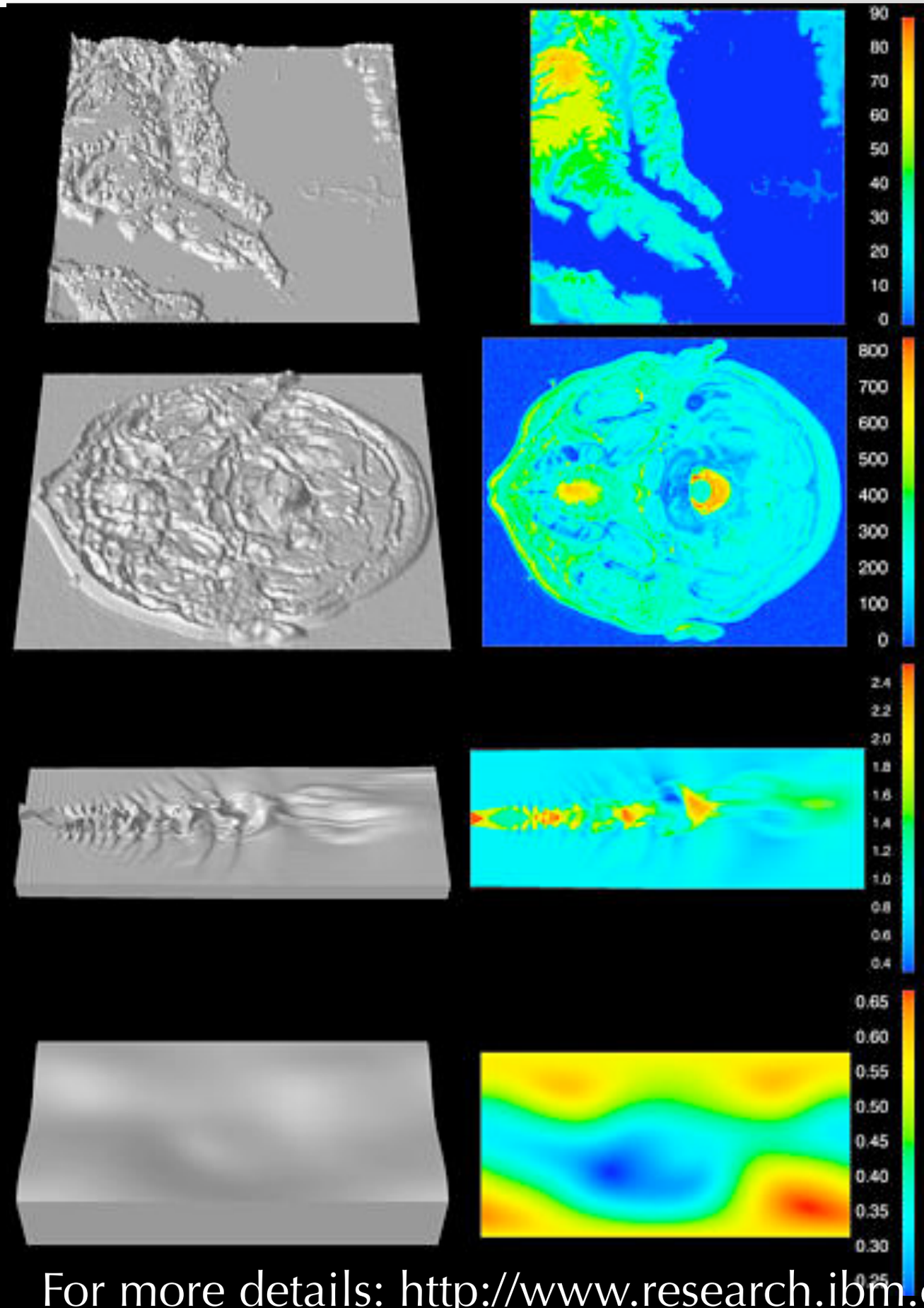
For more details: http://www.research.ibm.com/people/l/lloydt/color/color.HTM



Chesapeake Bay - note the artificial structure at higher altitude when it is actually quite gradual

Bad Habits

For more details: http://www.research.ibm.com/people/l/lloydt/color/color.HTM

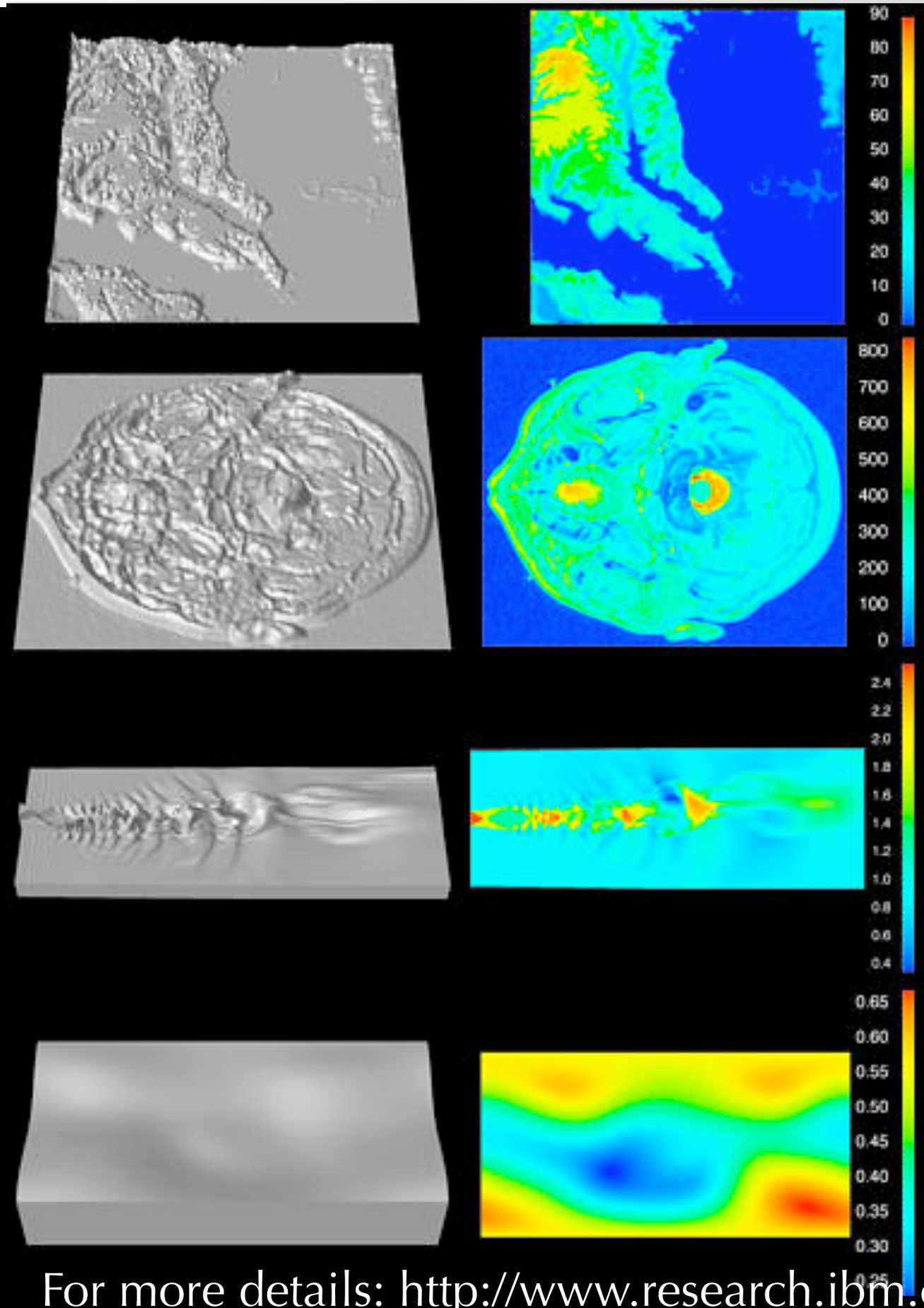


Chesapeake Bay - note the artificial structure at higher altitude when it is actually quite gradual

An slice of an MRI scan of a human brain. Washing out of detail and artificial structure.

Bad Habits

For more details: http://www.research.ibm.com/people/l/lloydt/color/color.HTM



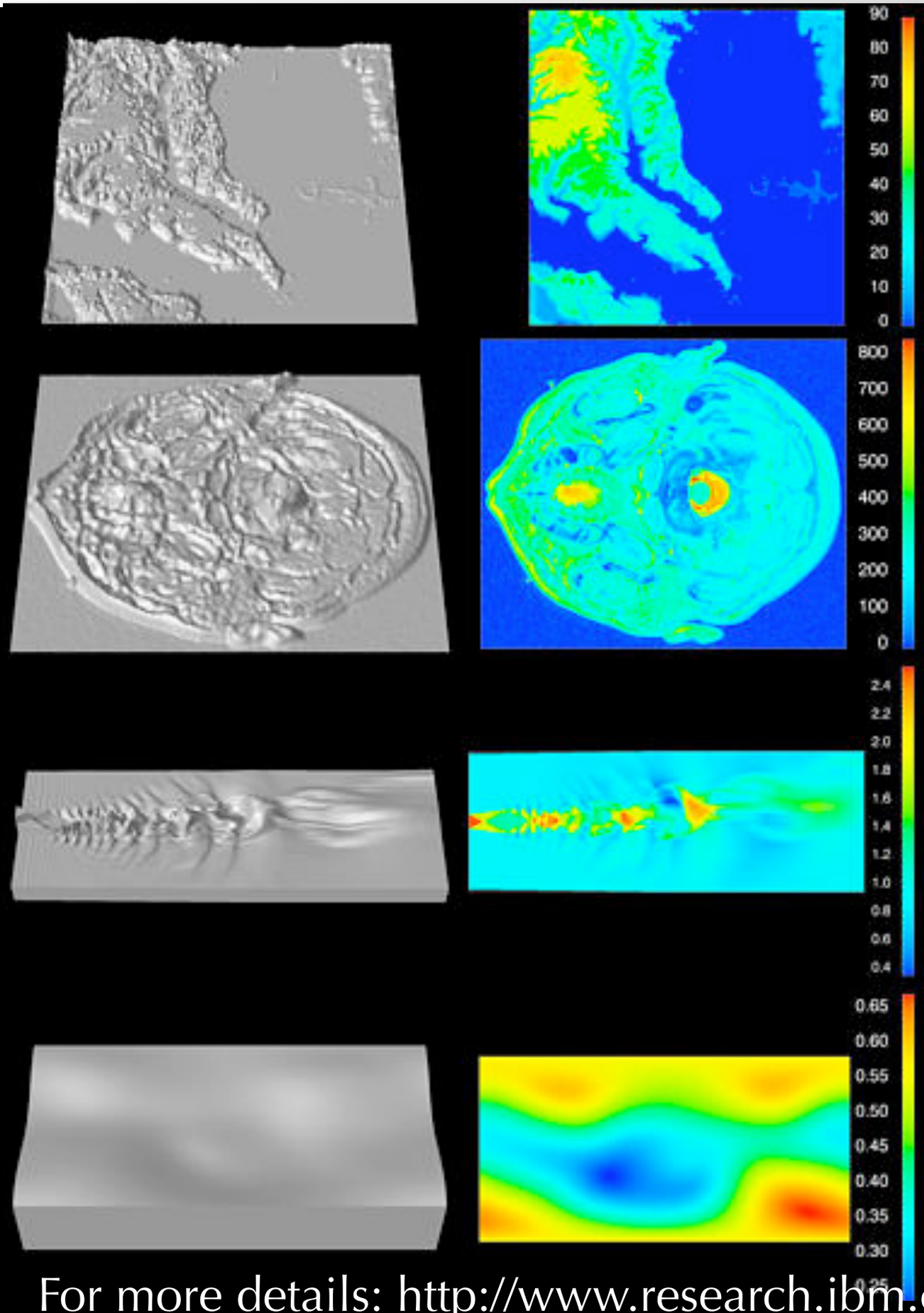
Chesapeake Bay - note the artificial structure at higher altitude when it is actually quite gradual

An slice of an MRI scan of a human brain. Washing out of detail and artificial structure.

Bad Habits

Turbulent flow from a jet engine.

For more details: http://www.research.ibm.com/people/l/lloydt/color/color.HTM



Chesapeake Bay - note the artificial structure at higher altitude when it is actually quite gradual

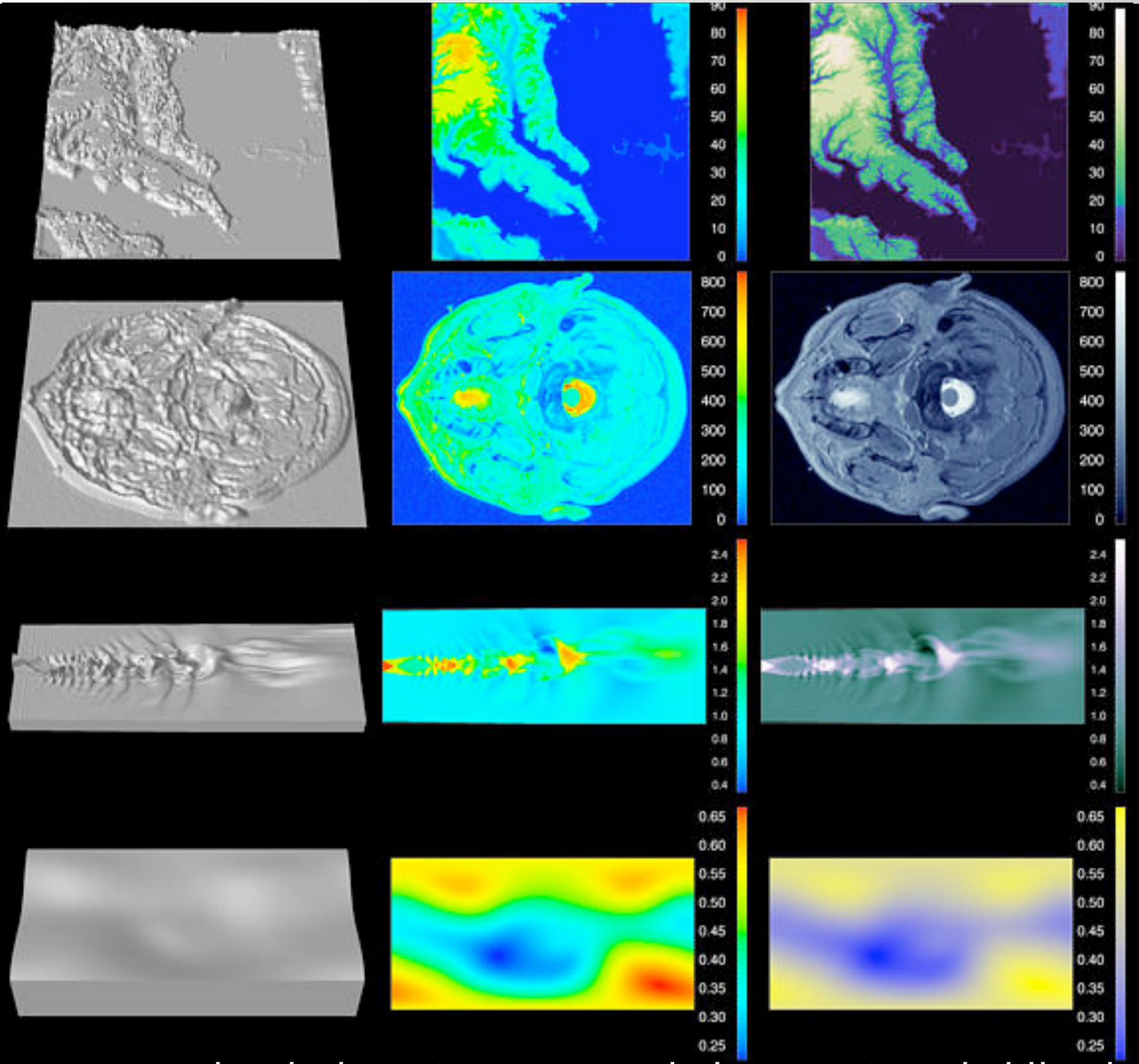
An slice of an MRI scan of a human brain. Washing out of detail and artificial structure.

Bad Habits

Turbulent flow from a jet engine.

Earth's magnetic field in a Cartesian projection - note the smooth structure.

For more details: http://www.research.ibm.com/people/l/lloydt/color/color.HTM

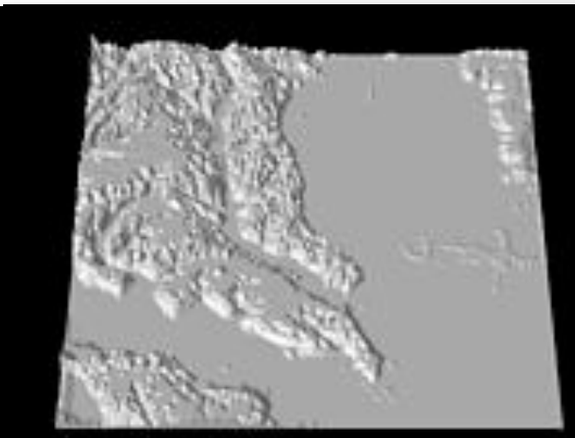


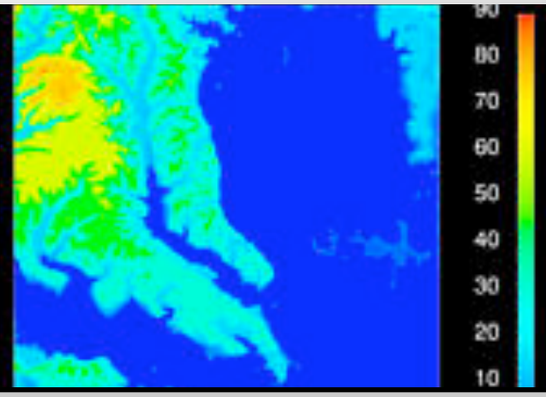
A natural zero

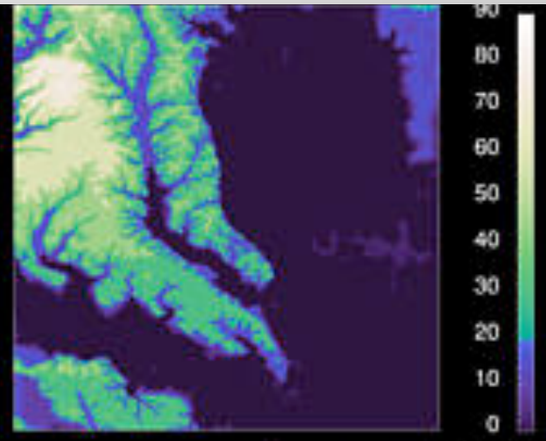
High frequency information - best with variation of luminance

Low frequency information - colour variation (saturation is good)

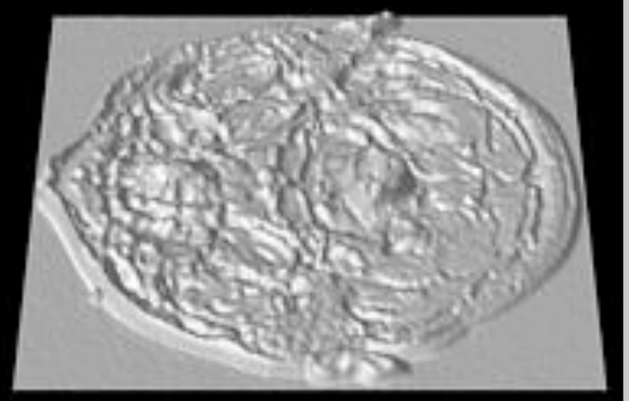
For more details: http://www.research.ibm.com/people/l/lloydt/color/color.HTM





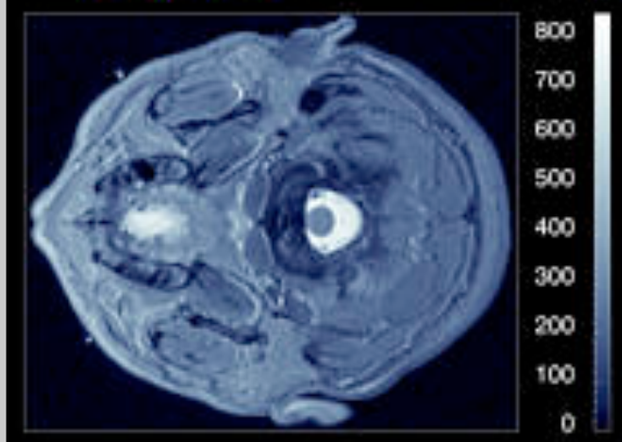


A natural zero

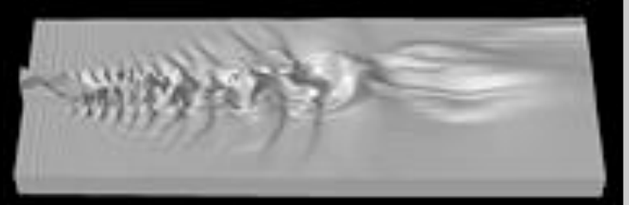


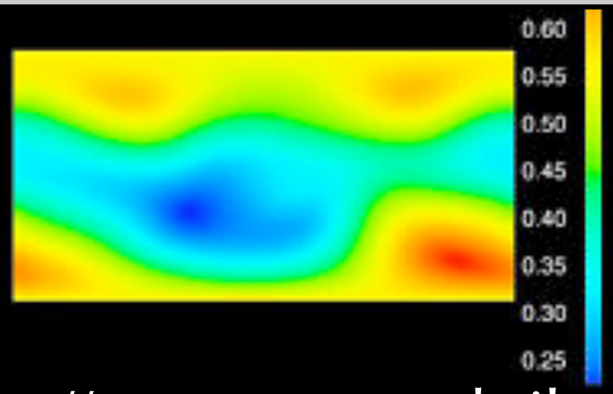
One key lesson:

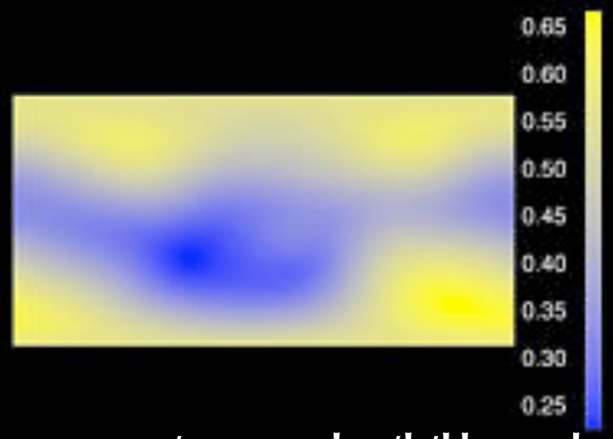
Try different colour maps and do not use a default one mindlessly.



High frequency information - best with variation of luminance



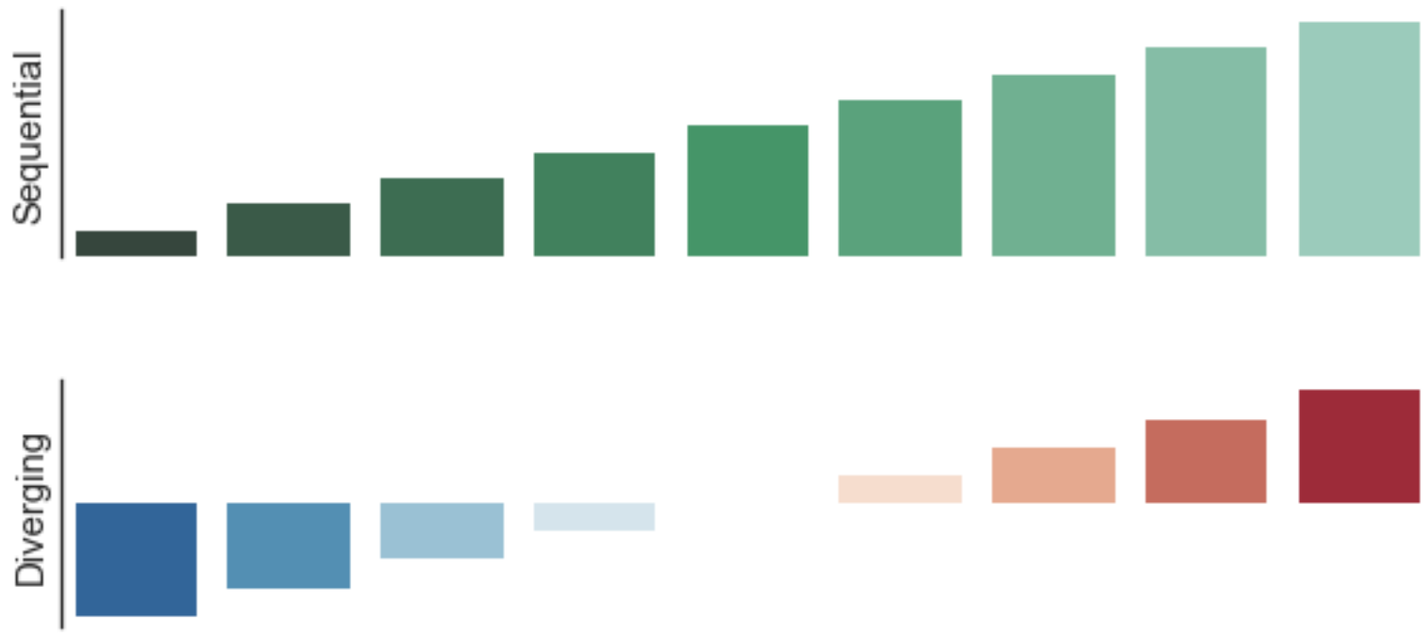


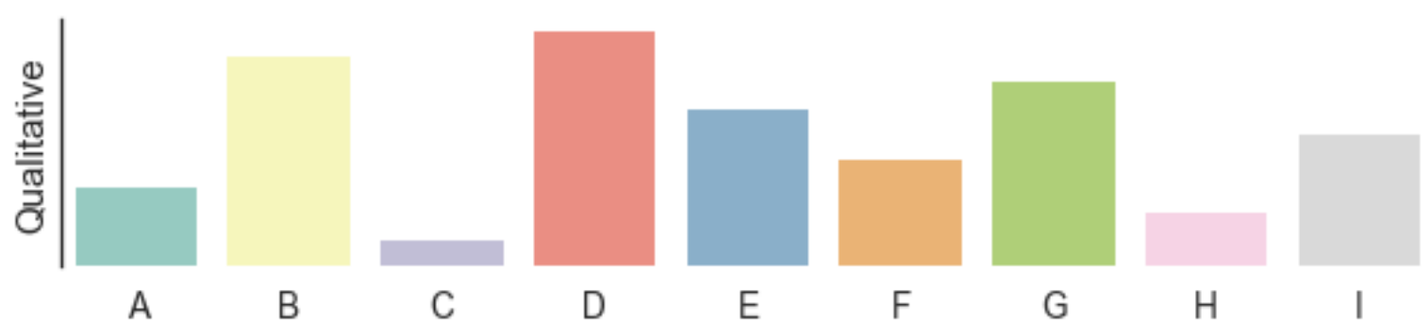


Low frequency information - colour variation (saturation is good)

For more details: http://www.research.ibm.com/people/l/lloydt/color/color.HTM

Colour should provide information





From the seaborn gallery [but meant to illustrate colour choices]

Colour choice - colour vision deficiency

This affects ~10% of the population so worth keeping in mind when you make plots for sharing with others.

Simplest: Do not use red & green to create contrast.

Better: use safe colour palettes (viridis/cividis in matplotlib):

In Seaborn:

```
import seaborn as sns
sns.set_palette('colorblind')
```

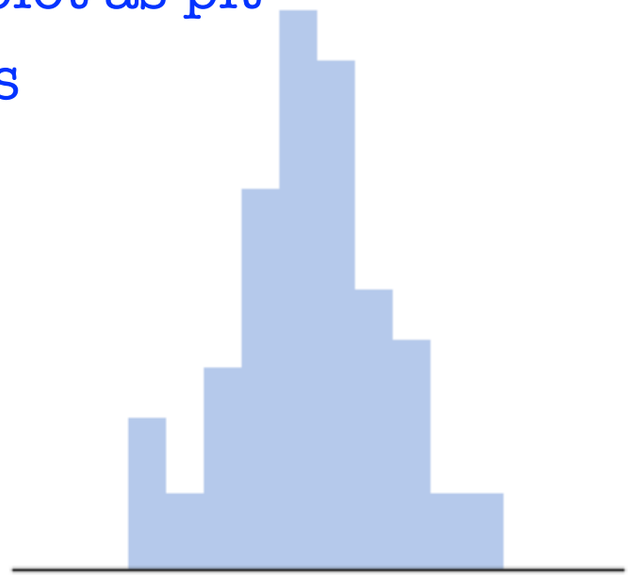
Paul Tol has a very useful page:

https://personal.sron.nl/~pault/

Checking: Photoshop/Illustrator have options, in Python, use Jörg Dietrich's Daltonize package: https://github.com/joergdietrich/daltonize

Various ways to show 1D distributions

```
import matplotlib.pyplot as plt
import seaborn as sns
```



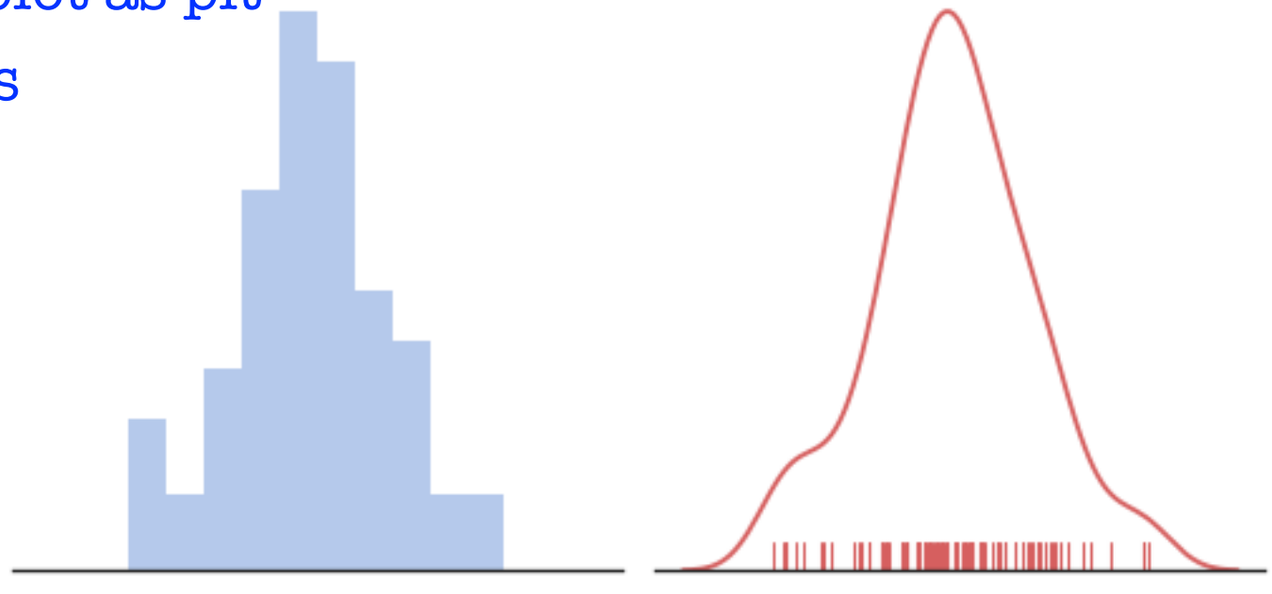
```
sns.distplot(x, kde=False)
or
plt.hist(x)
```

Histograms are convenient but bin the data and are not differentiable - we'll look at kernel density plots later. Rug plots complement histograms and kernel density plots by showing the data.

http://seaborn.pydata.org/examples/distplot_options.html

Various ways to show 1D distributions

```
import matplotlib.pyplot as plt
import seaborn as sns
```



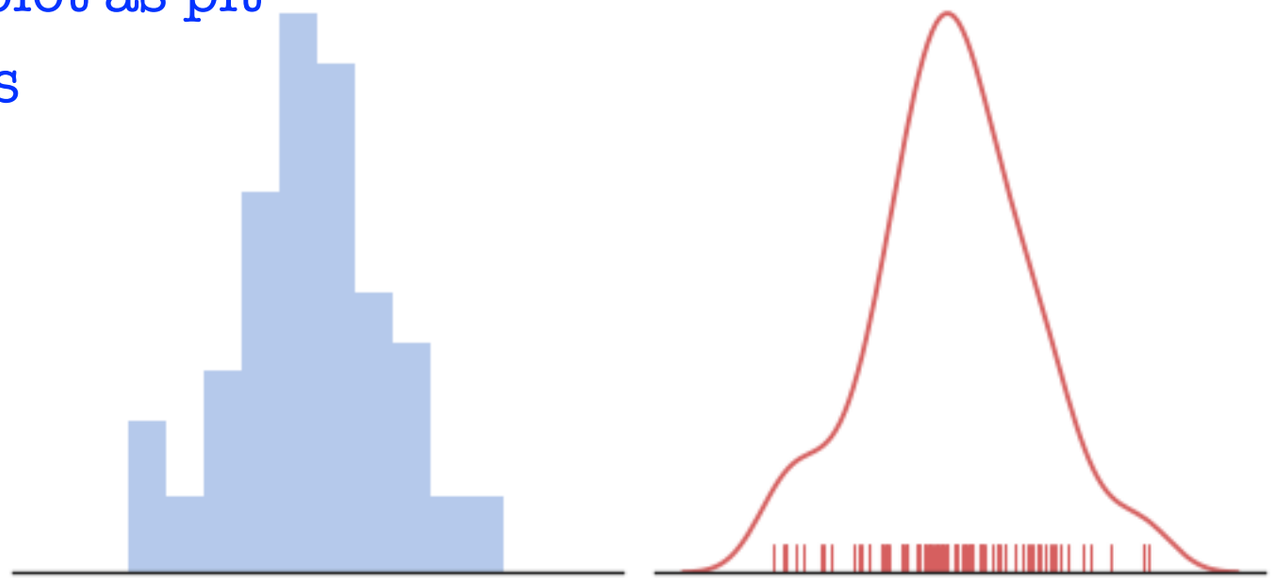
```
sns.distplot(x, kde=True,
rug=True)
```

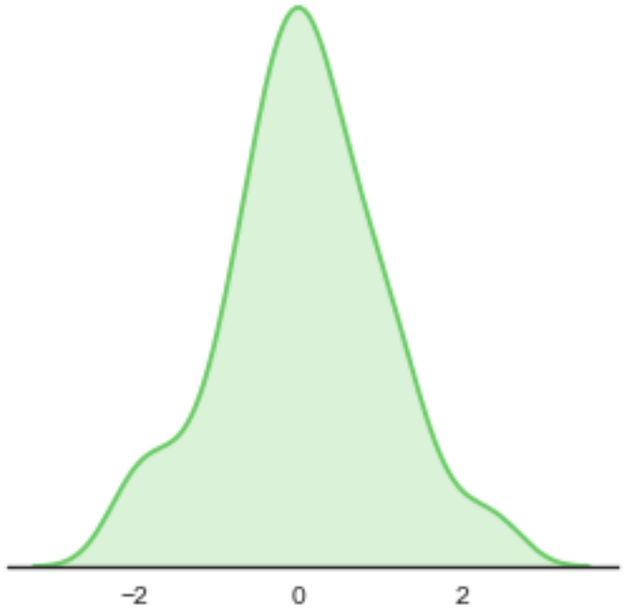
Histograms are convenient but bin the data and are not differentiable - we'll look at kernel density plots later. Rug plots complement histograms and kernel density plots by showing the data.

http://seaborn.pydata.org/examples/distplot_options.html

Various ways to show 1D distributions

```
import matplotlib.pyplot as plt
import seaborn as sns
```





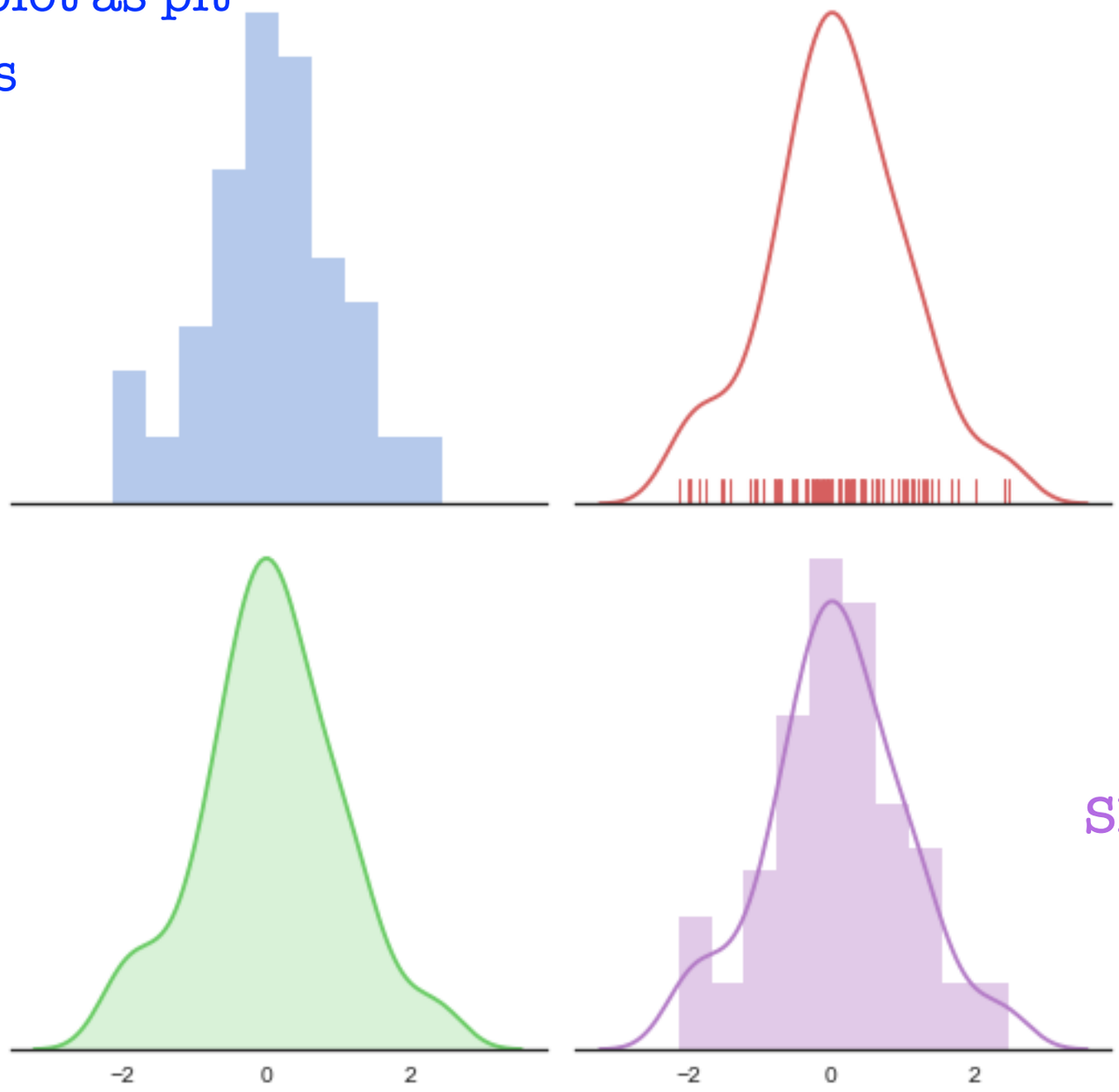

```
sns.distplot(x, hist=False)
```

Histograms are convenient but bin the data and are not differentiable - we'll look at kernel density plots later. Rug plots complement histograms and kernel density plots by showing the data.

http://seaborn.pydata.org/examples/distplot_options.html

Various ways to show 1D distributions

```
import matplotlib.pyplot as plt
import seaborn as sns
```



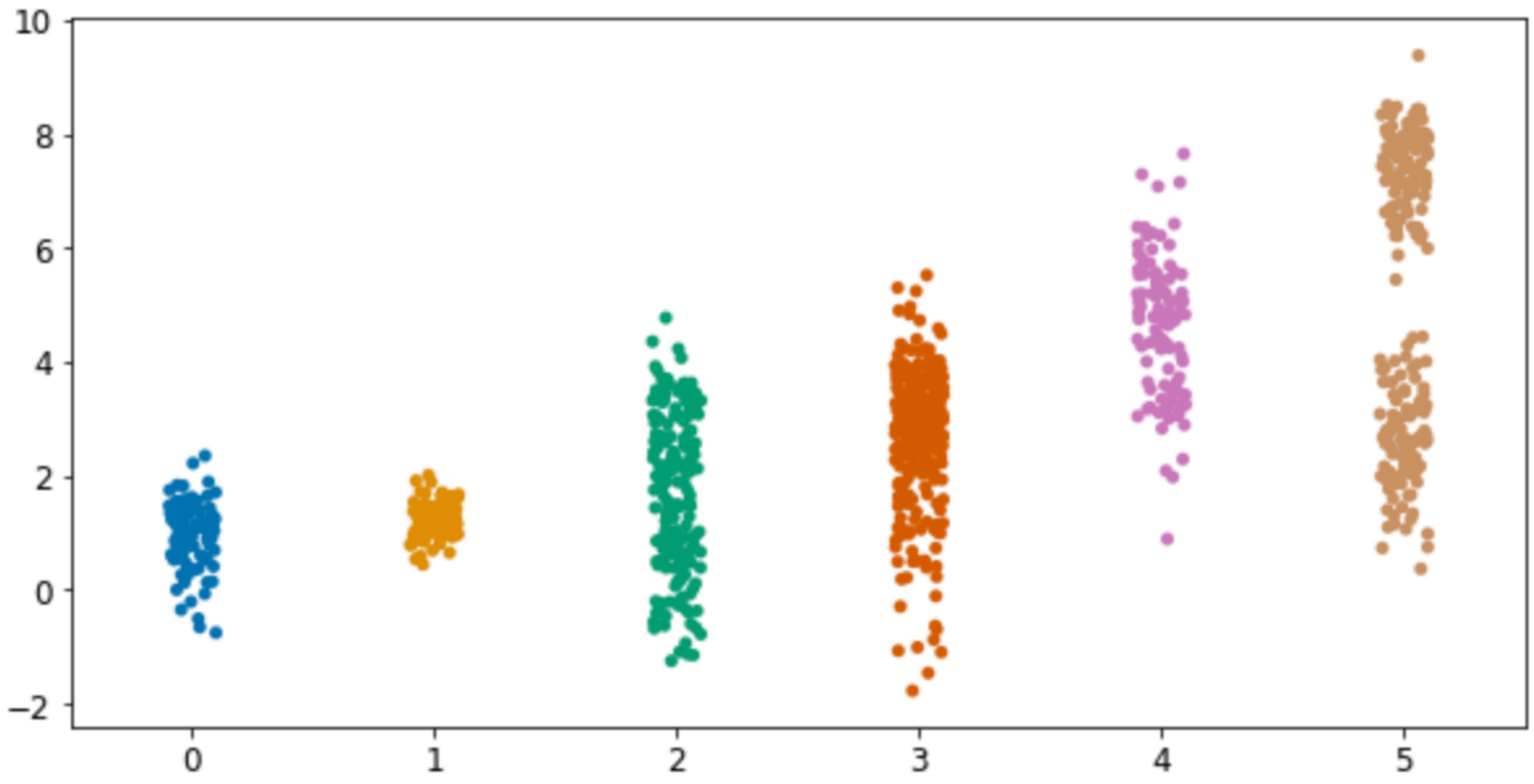

```
sns.distplot(x)
```

Histograms are convenient but bin the data and are not differentiable - we'll look at kernel density plots later. Rug plots complement histograms and kernel density plots by showing the data.

http://seaborn.pydata.org/examples/distplot_options.html

Comparing multiple 1D distributions

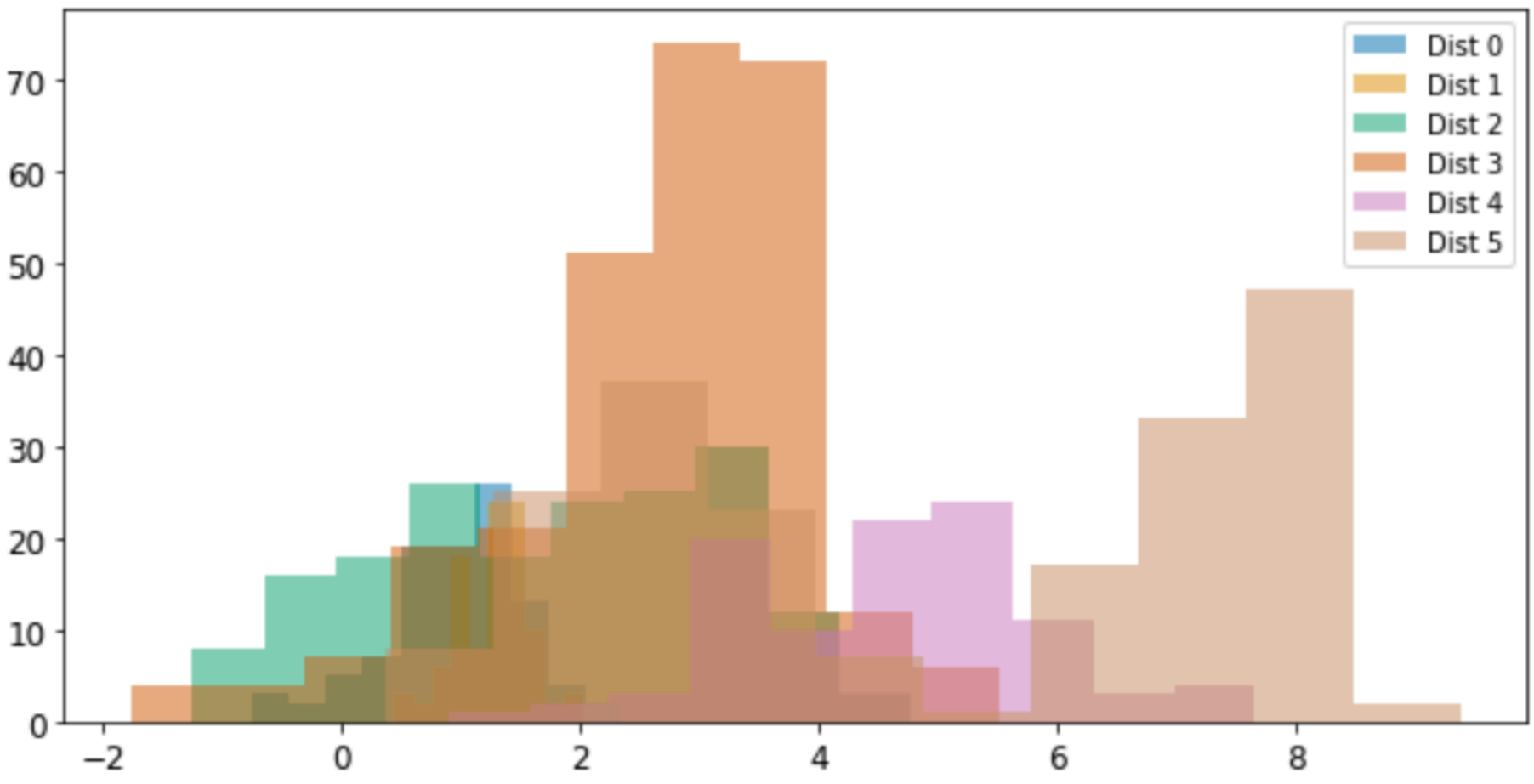
Scenario: 6 classes with measurements -
how do we compare them?



See the Distribution Illustration notebook in the Lecture directory

Comparing multiple 1D distributions

Scenario: 6 classes with measurements -
how do we compare them?

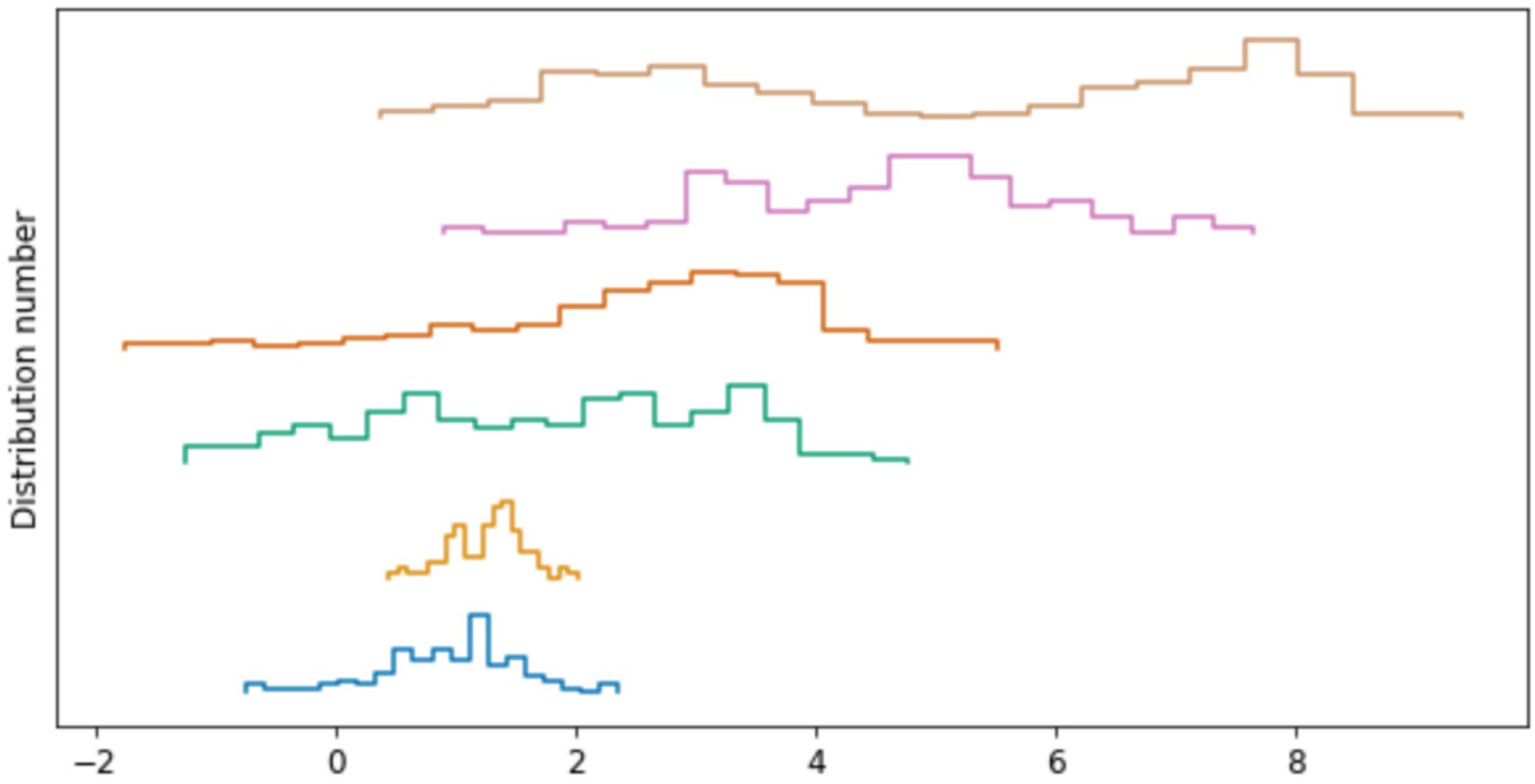


over-plotting histograms does not work well

See the Distribution Illustration notebook in the Lecture directory

Comparing multiple 1D distributions

Scenario: 6 classes with measurements -
how do we compare them?

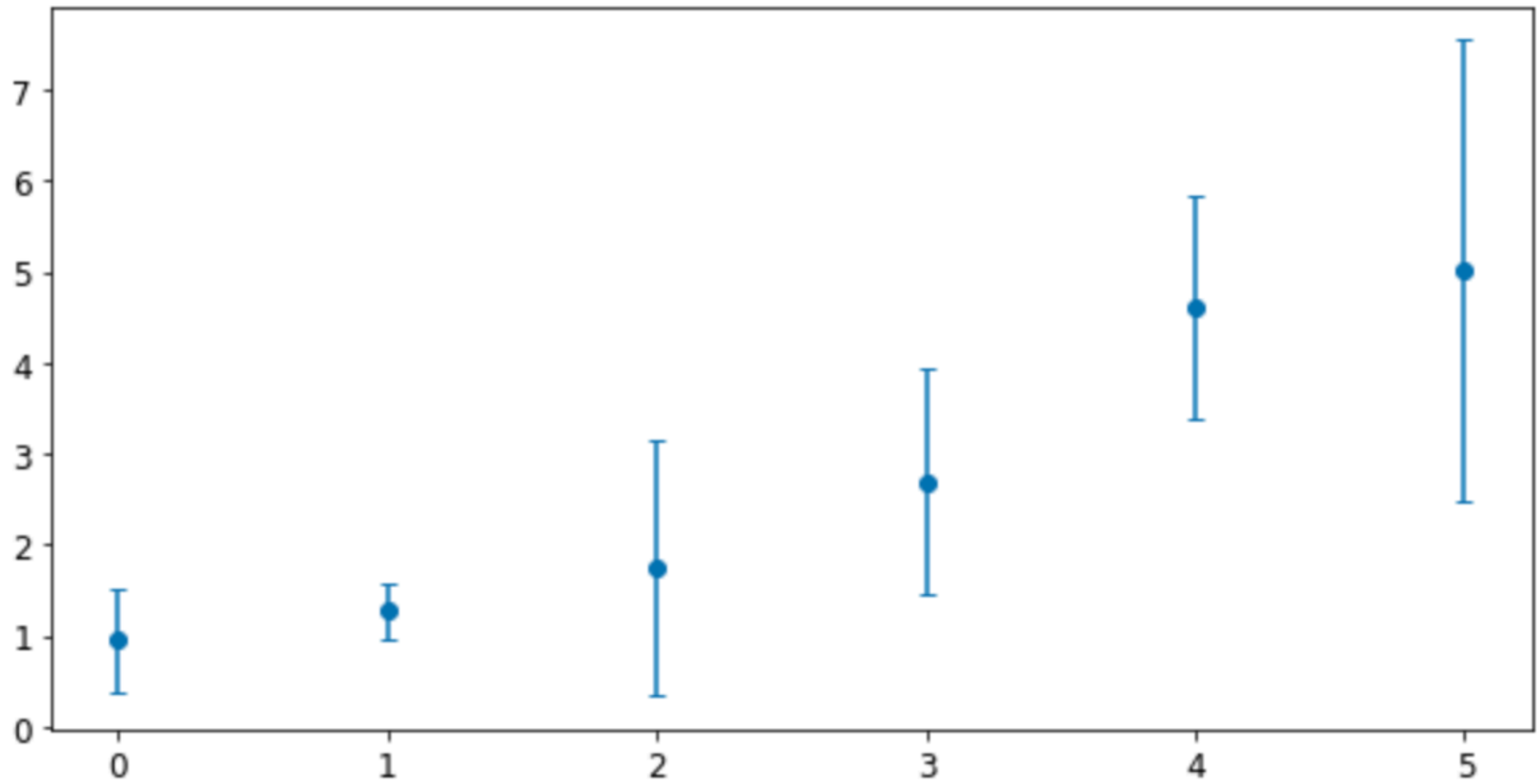


offsetting histograms is a possibility

See the Distribution Illustration notebook in the Lecture directory

Comparing multiple 1D distributions

Scenario: 6 classes with measurements -
how do we compare them?

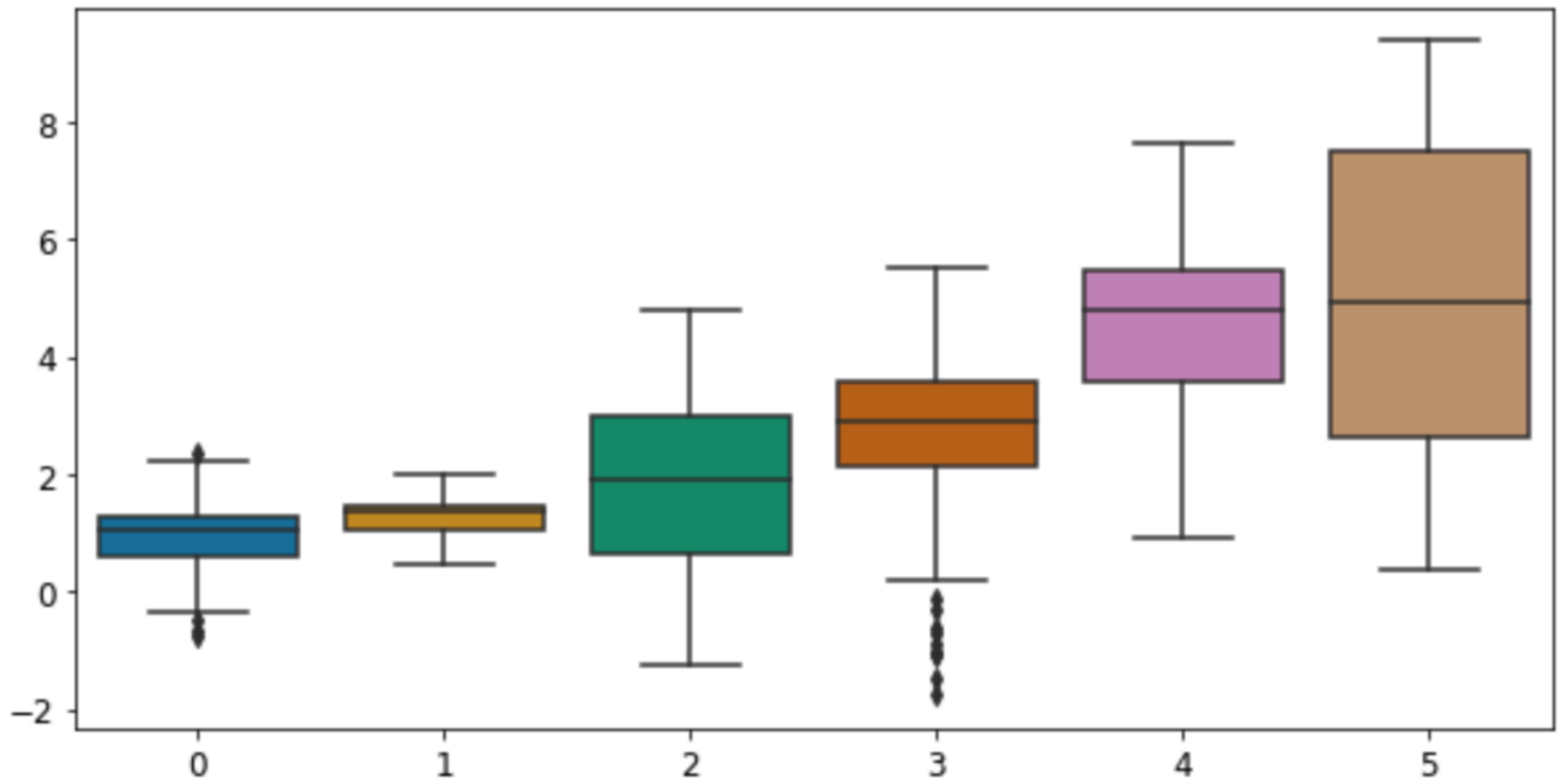


errorbars give some illustration of spread but lacks detail

See the Distribution Illustration notebook in the Lecture directory

Comparing multiple 1D distributions

Scenario: 6 classes with measurements -
how do we compare them?

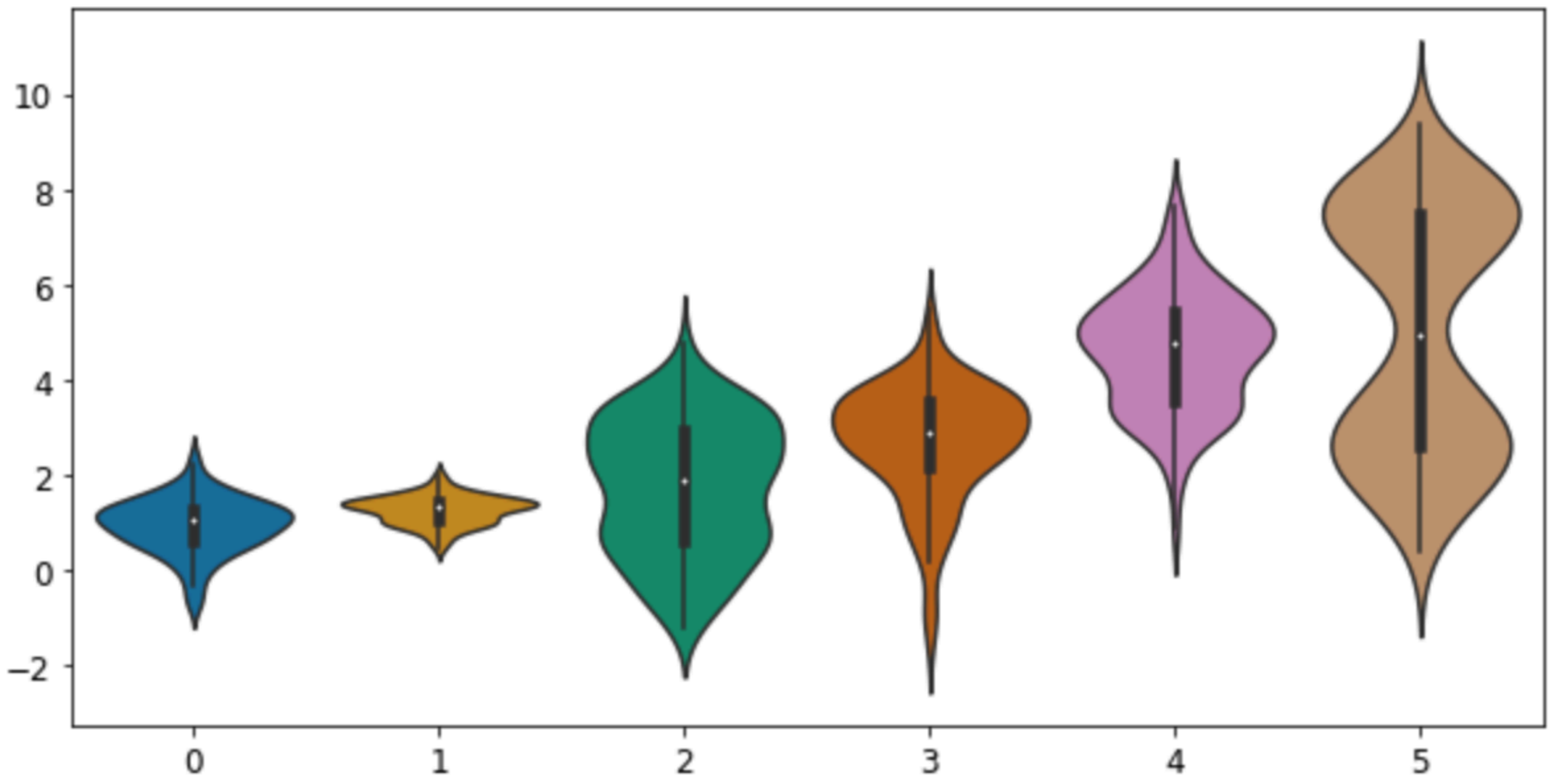


box plots are better - but only for uni-modal data

See the Distribution Illustration notebook in the Lecture directory

Comparing multiple 1D distributions

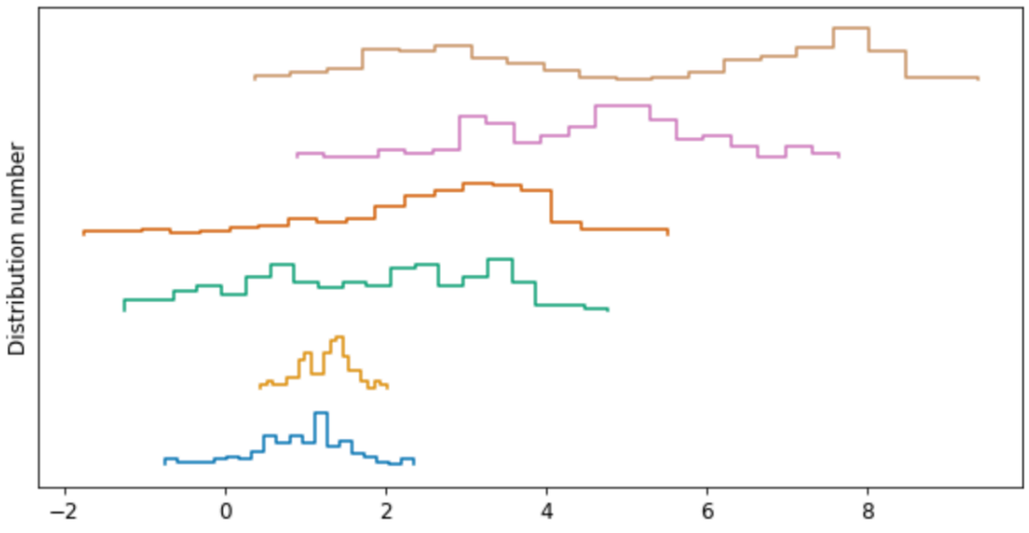
Scenario: 6 classes with measurements -
how do we compare them?

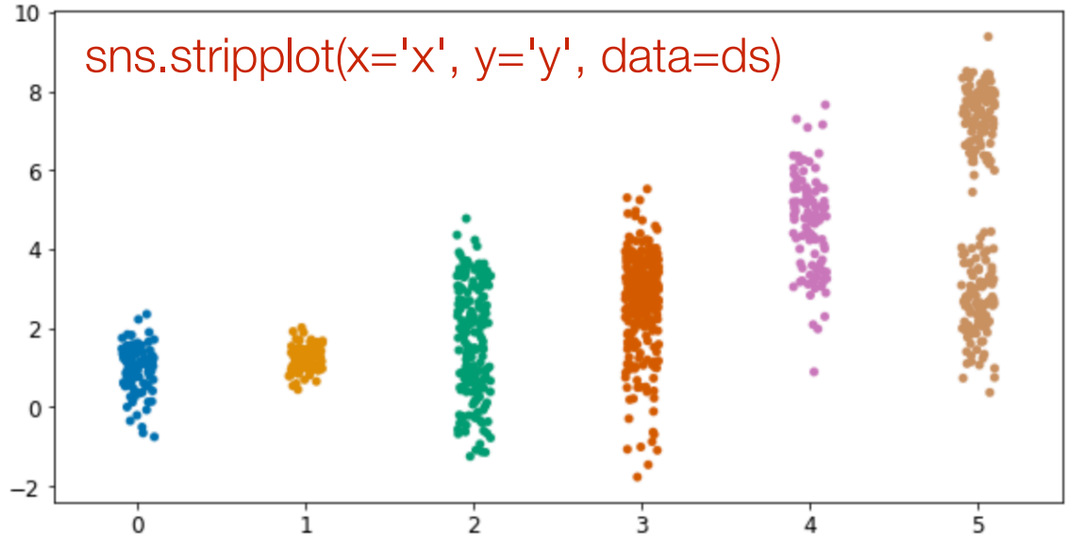


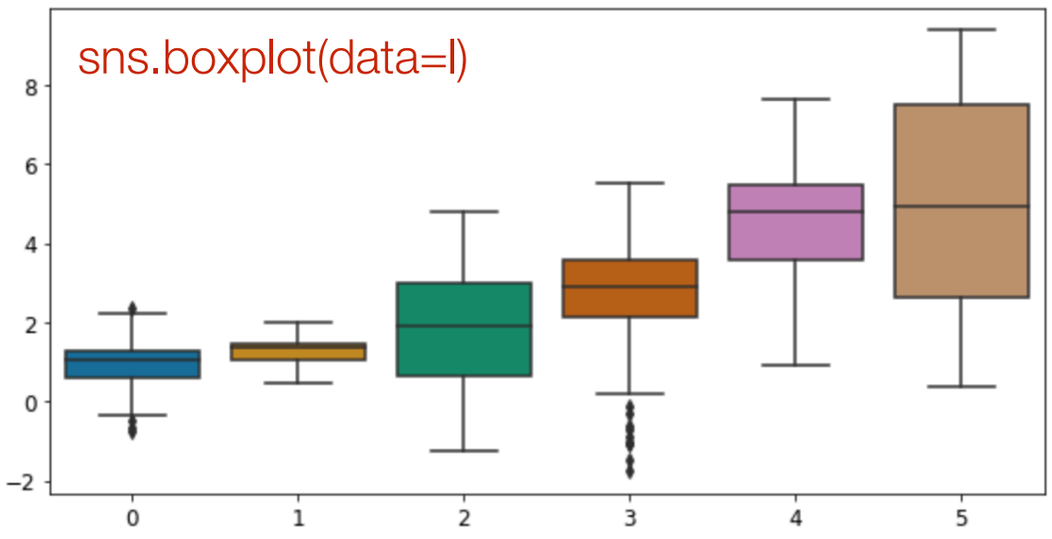
violin plots can be useful for this

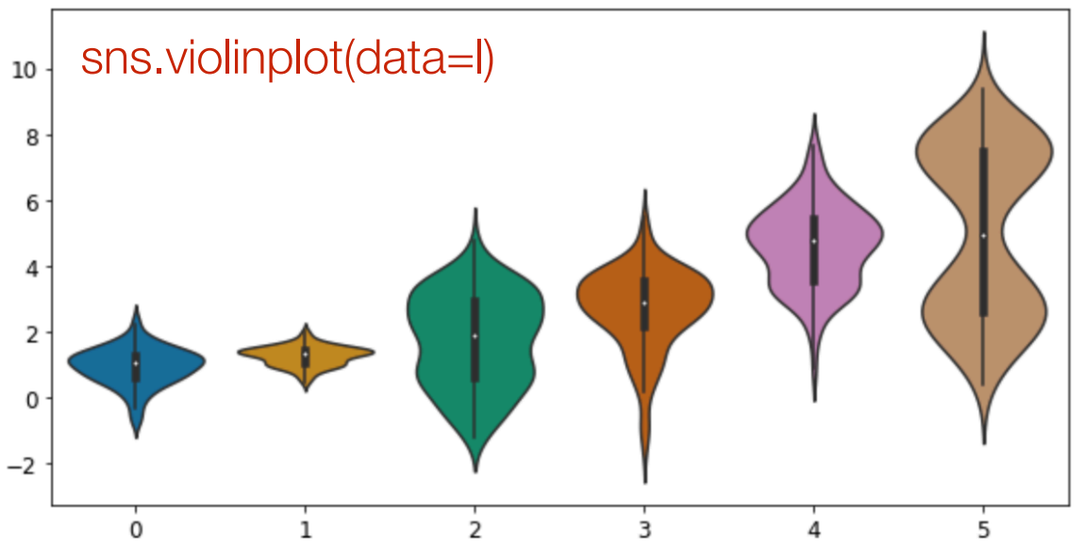
See the Distribution Illustration notebook in the Lecture directory

Comparing multiple 1D distributions



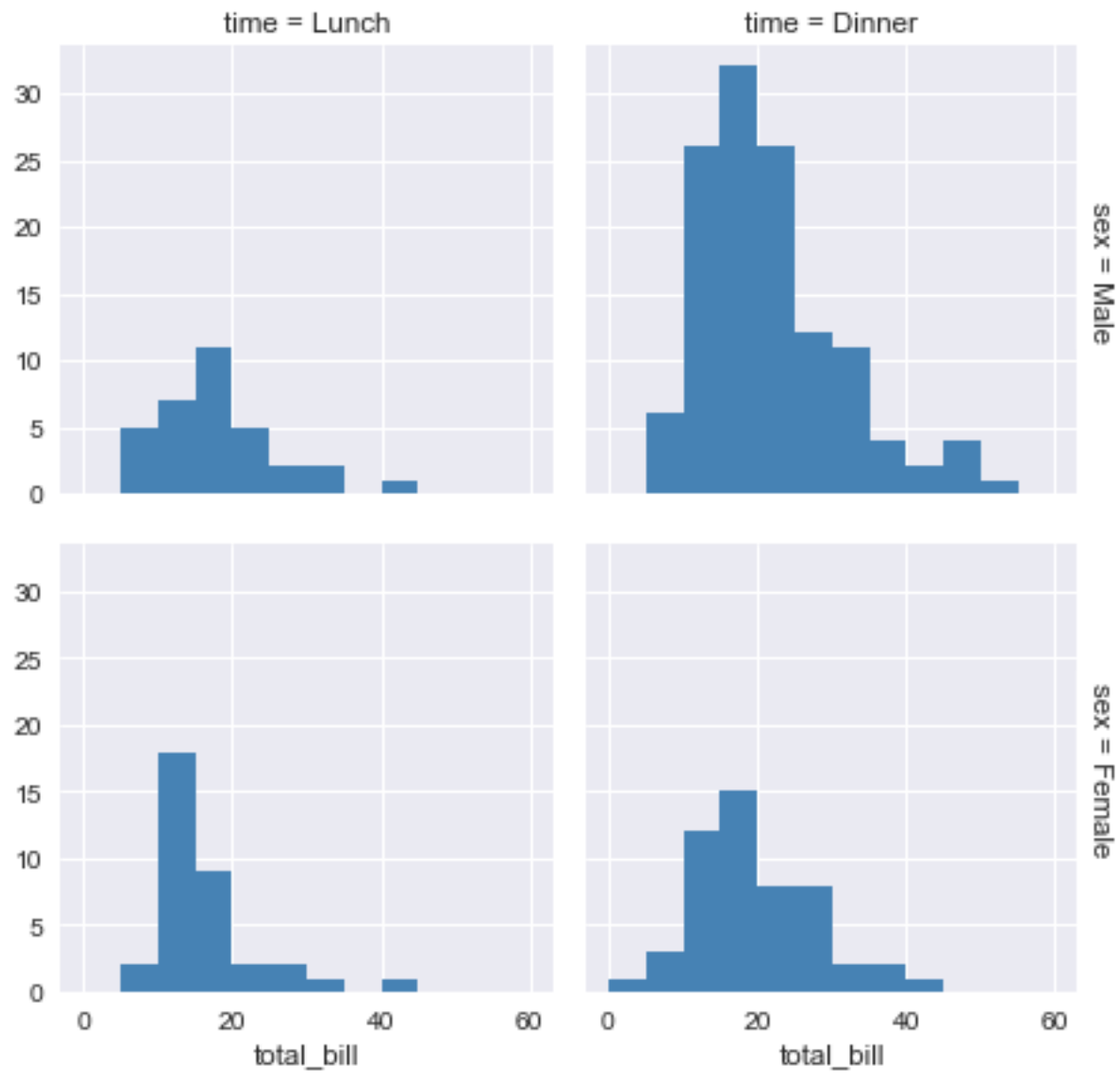






See the Distribution Illustration notebook in the Lecture directory

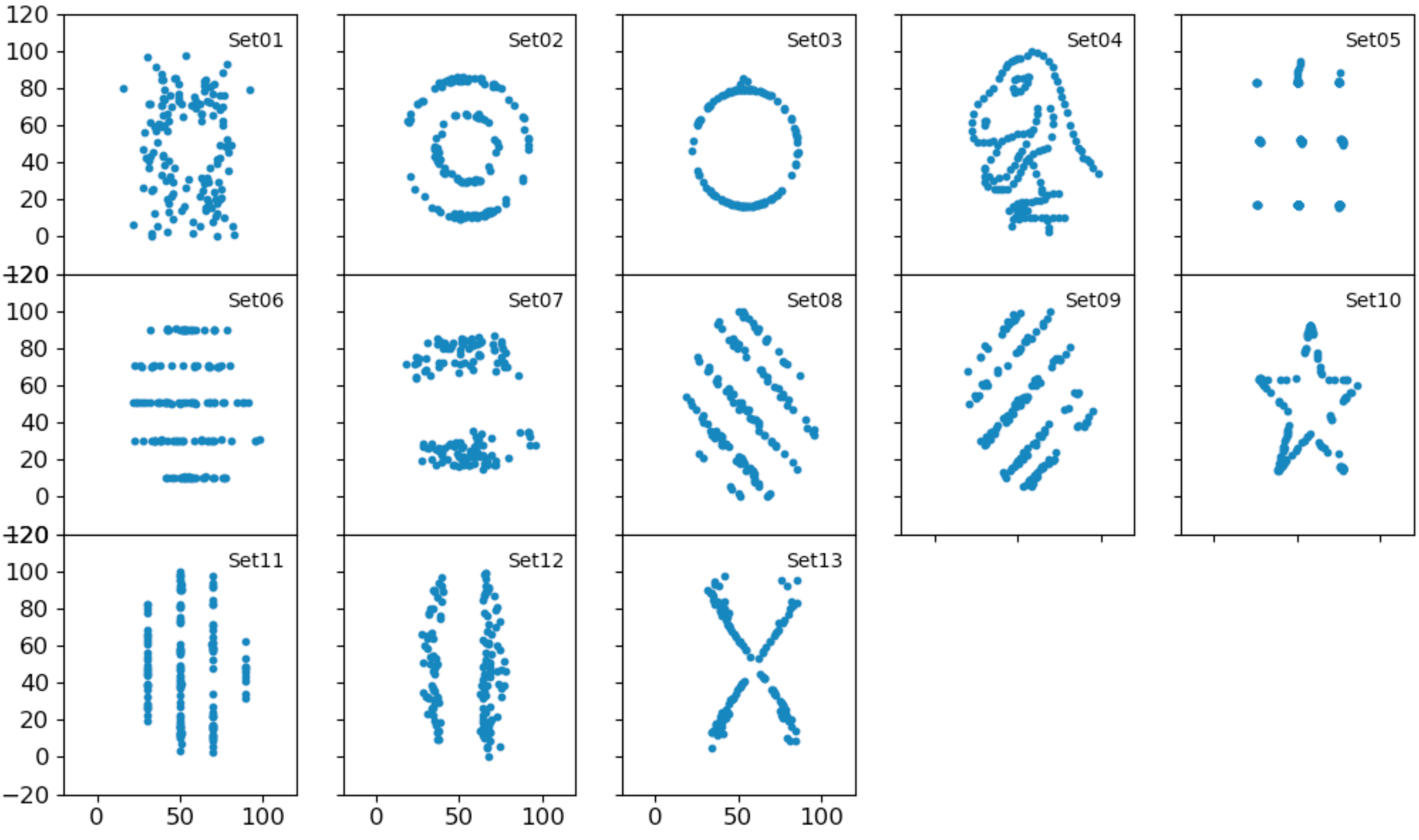
Various ways to show 1D distributions



Conditional plots (called FacetGrids in seaborn - in R the lattice package provides these) are very powerful tools for exploring complex multi-dimensional data. While shown for histograms here they can be used for all kinds of plots.

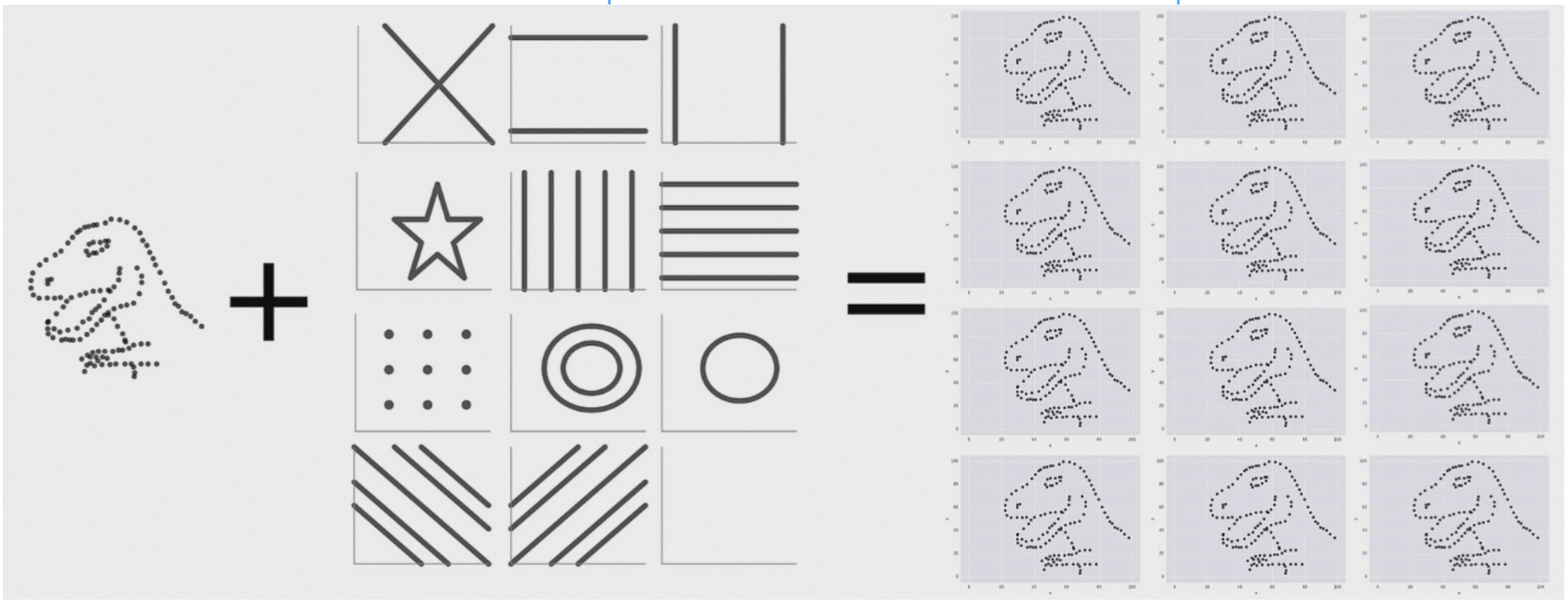
http://seaborn.pydata.org/examples/faceted_histogram.html

Visualisation - problem



But how did they do that?

From: https://www.autodeskresearch.com/publications/samestats

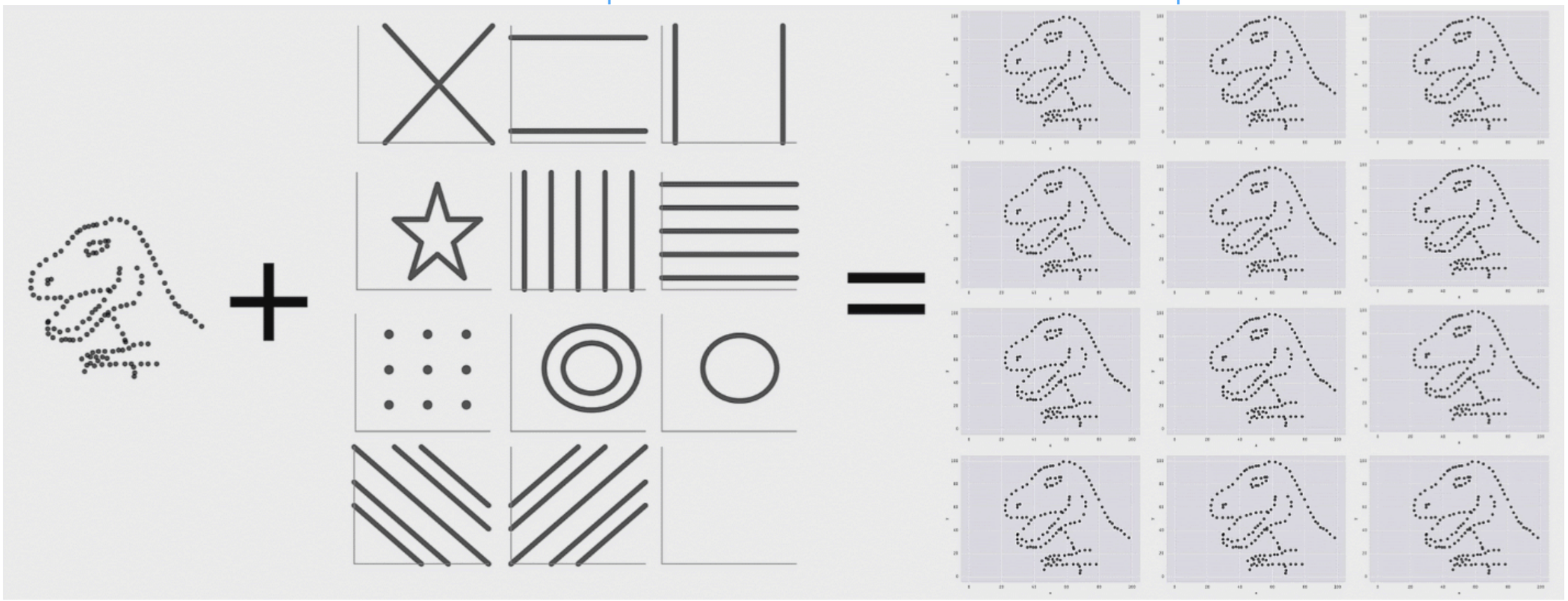


Start with a particular distribution, then shift points slightly making sure the summary statistics is the same and that you head towards a particular target statistic.

https://www.autodeskresearch.com/publications/samestats

But how did they do that?

From: https://www.autodeskresearch.com/publications/samestats

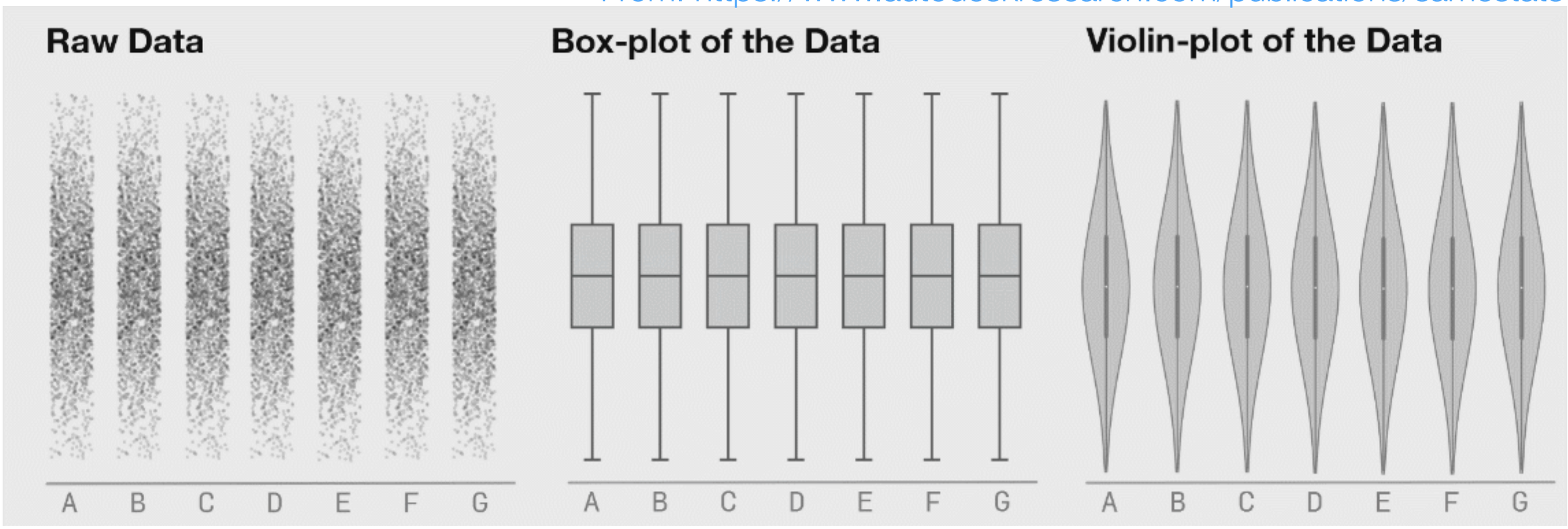


Start with a particular distribution, then shift points slightly making sure the summary statistics is the same and that you head towards a particular target statistic.

https://www.autodeskresearch.com/publications/samestats

Why boxplots can be untrustworthy

From: https://www.autodeskresearch.com/publications/samestats

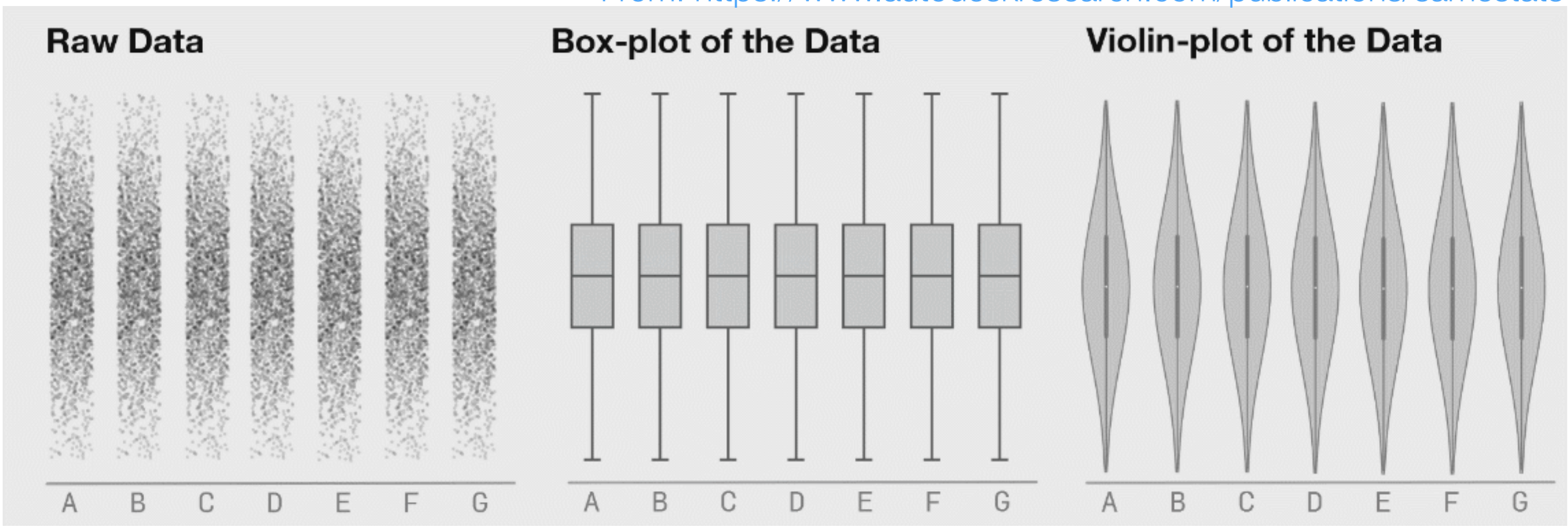


Again - same technique. Read more at

https://www.autodeskresearch.com/publications/samestats

Why boxplots can be untrustworthy

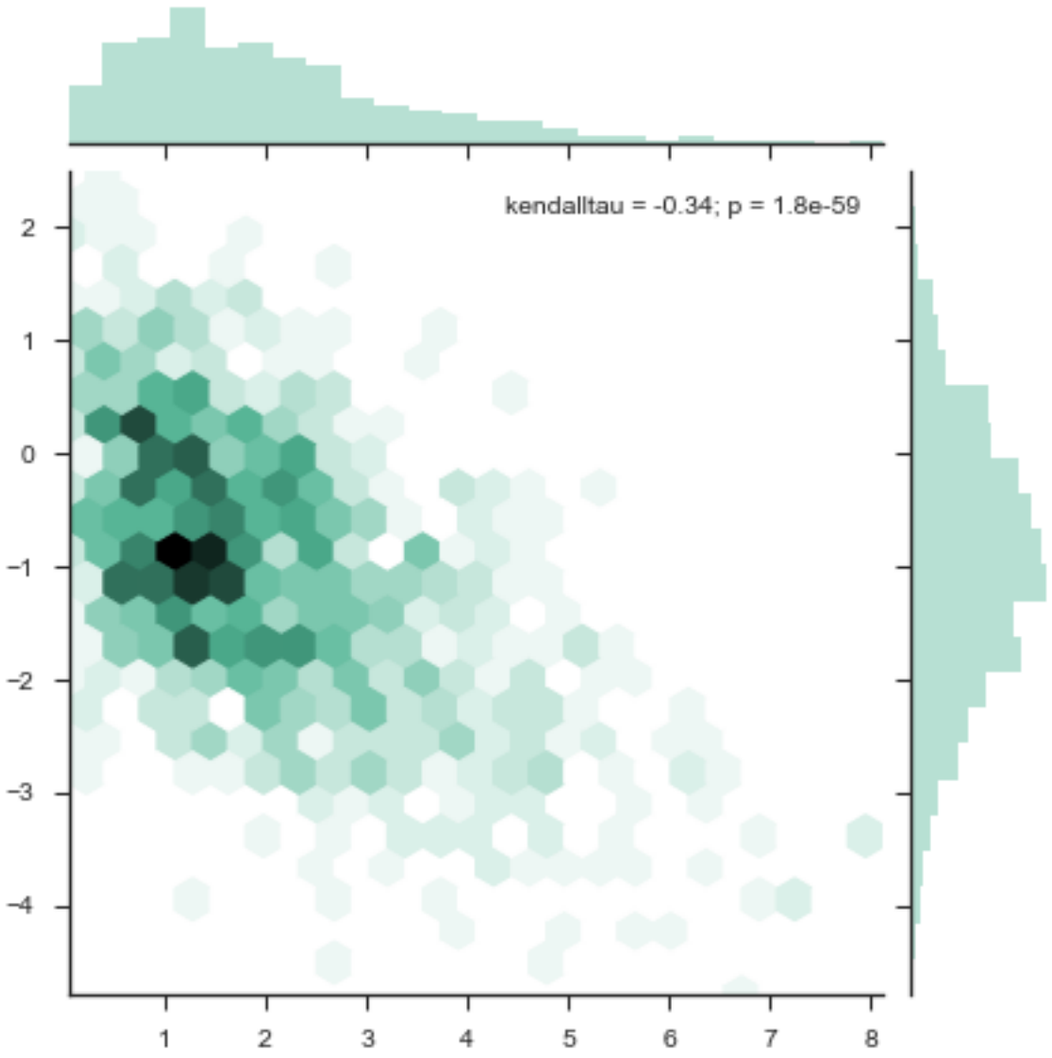
From: https://www.autodeskresearch.com/publications/samestats

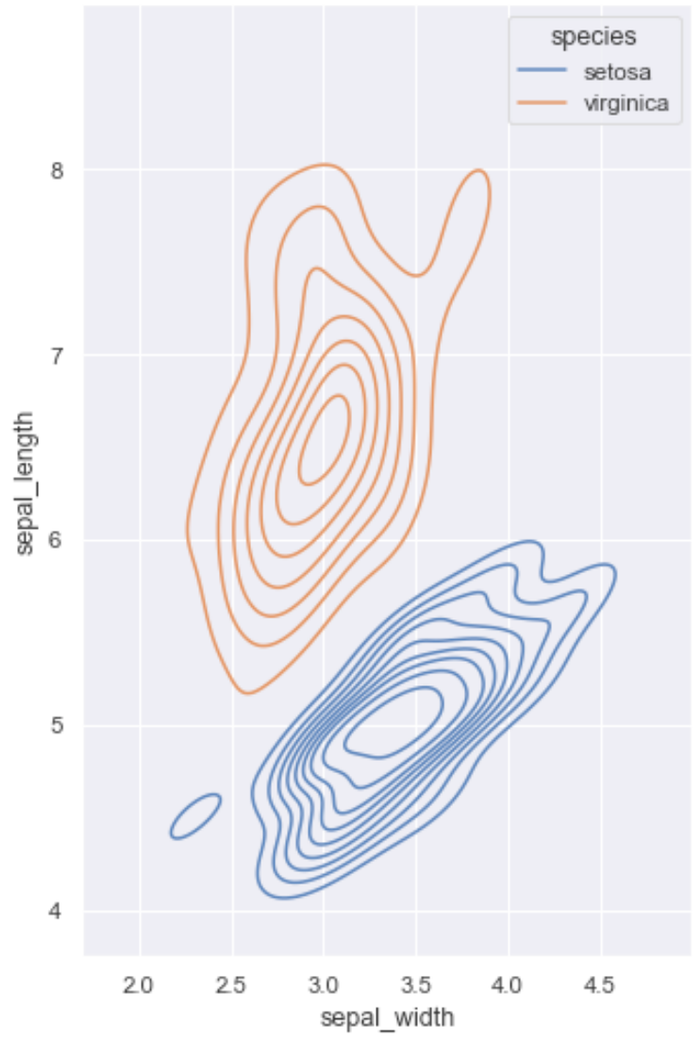


Again - same technique. Read more at

https://www.autodeskresearch.com/publications/samestats

Various ways to show 2D distributions

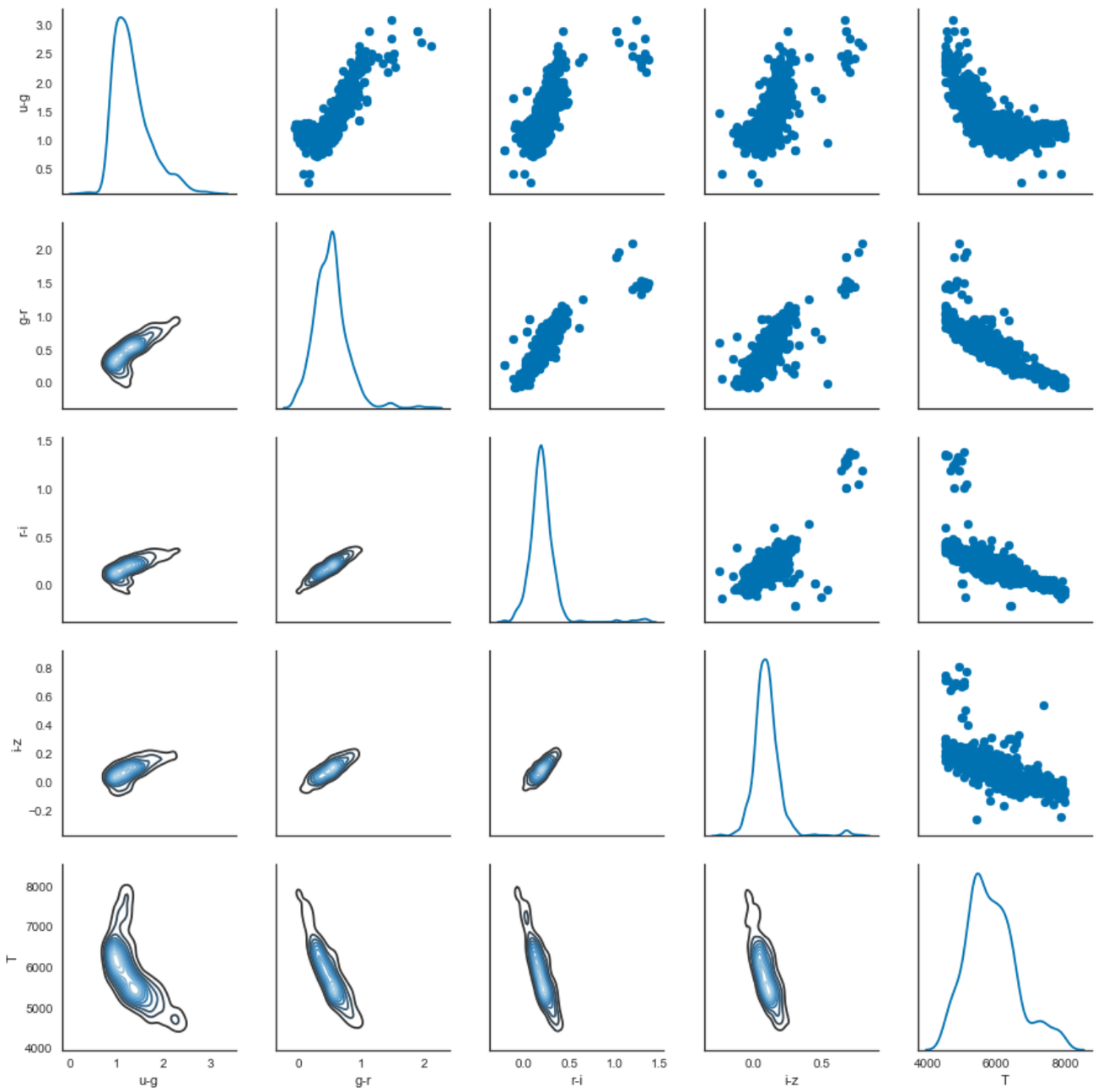




Hex-bin plots (left), or kernel density maps (right) are powerful ways to view 2D data structures and should be in your toolbox.

http://seaborn.pydata.org/examples/hexbin_marginals.html and https://seaborn.pydata.org/examples/multiple_bivariate_kde.html

Multi-dimensional data

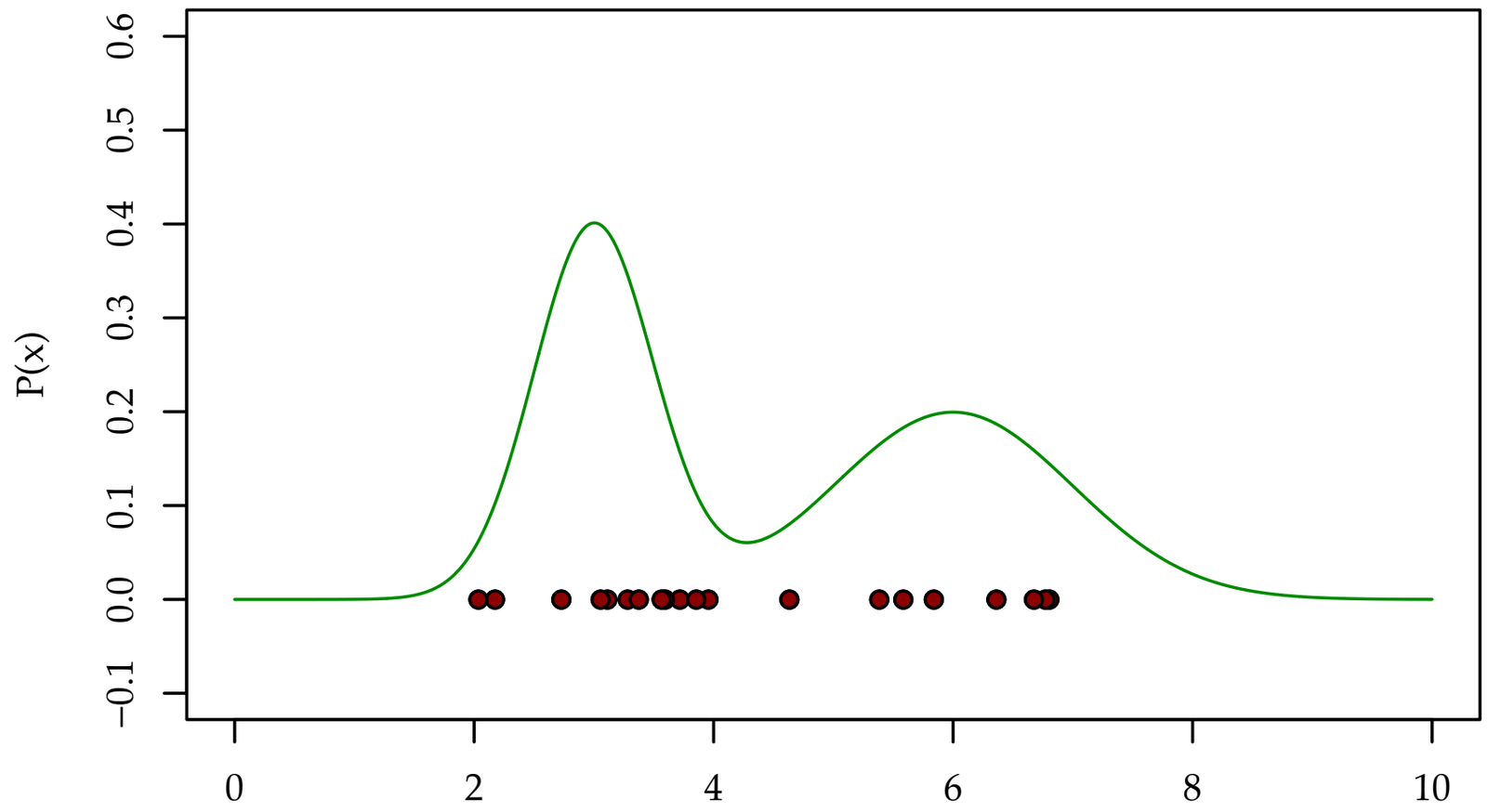


Scatter-plot matrices or pairplots are very powerful ways to explore the inter-relationship of many variables quickly. But always keep in mind that the true correlation might be hidden in these projections.

https://seaborn.pydata.org/generated/seaborn.pairplot.html

Density estimation

How do you recover the “true” density distribution?



Two main approaches:

Non-parametric - where no assumptions about the functional form is made.

Parametric - where we do assume a particular functional form (two Gaussians for instance), and fit for the parameters of the function

Density estimation - formally

Formally a density estimation problem is to find/ learn a function

$$p_{\text{model}}(\mathbf{x}) : \mathbb{R}^n \rightarrow \mathbb{R}$$

In most cases it is natural to interpret $p_{model}(\boldsymbol{x})$ as a probability density function.

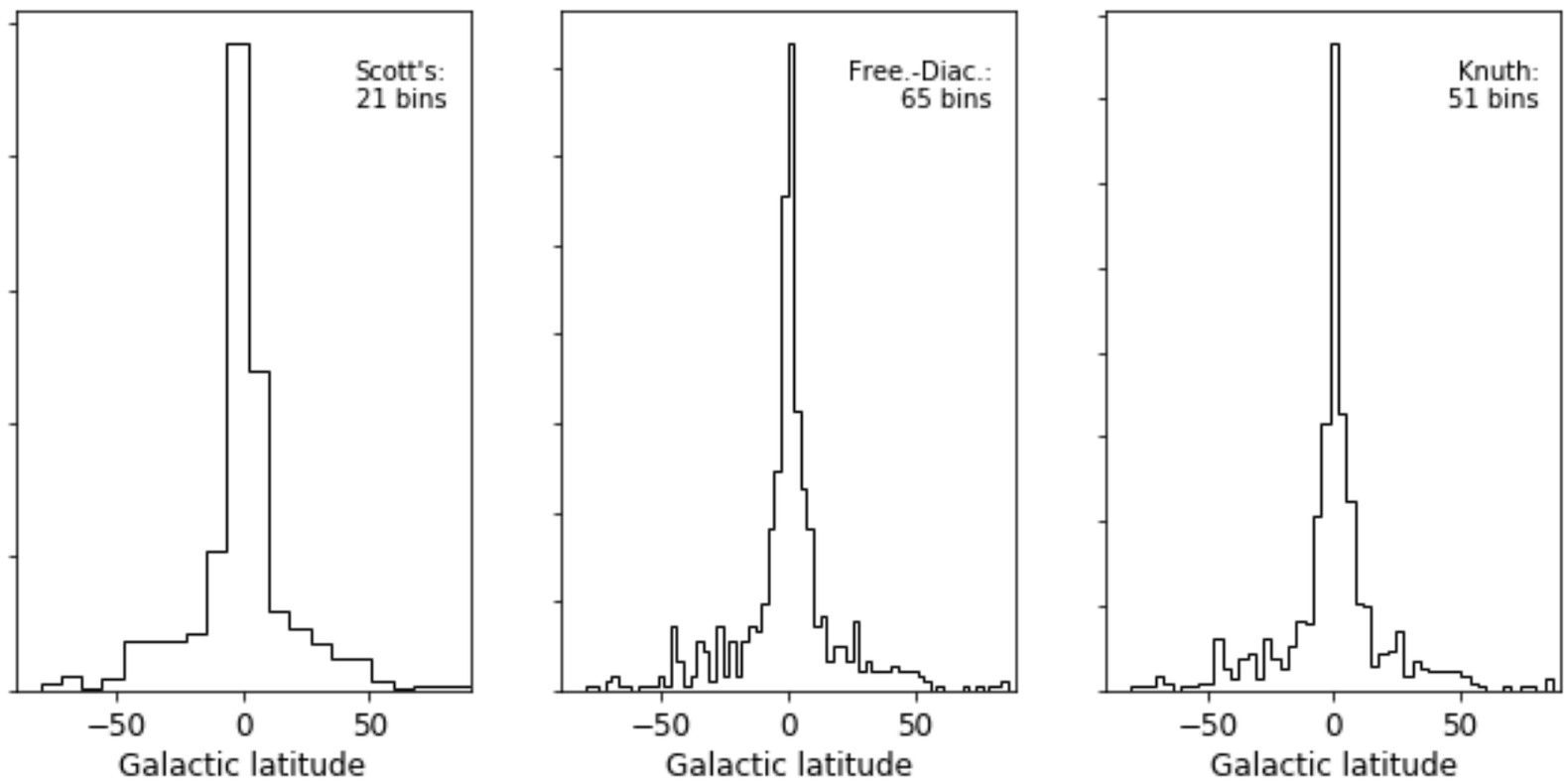
As soon as you have this function you could use it to predict missing data, take the derivative of it, create new data, or use it for visualisation - among other things.

Simple Non-parametric estimators

Histograms

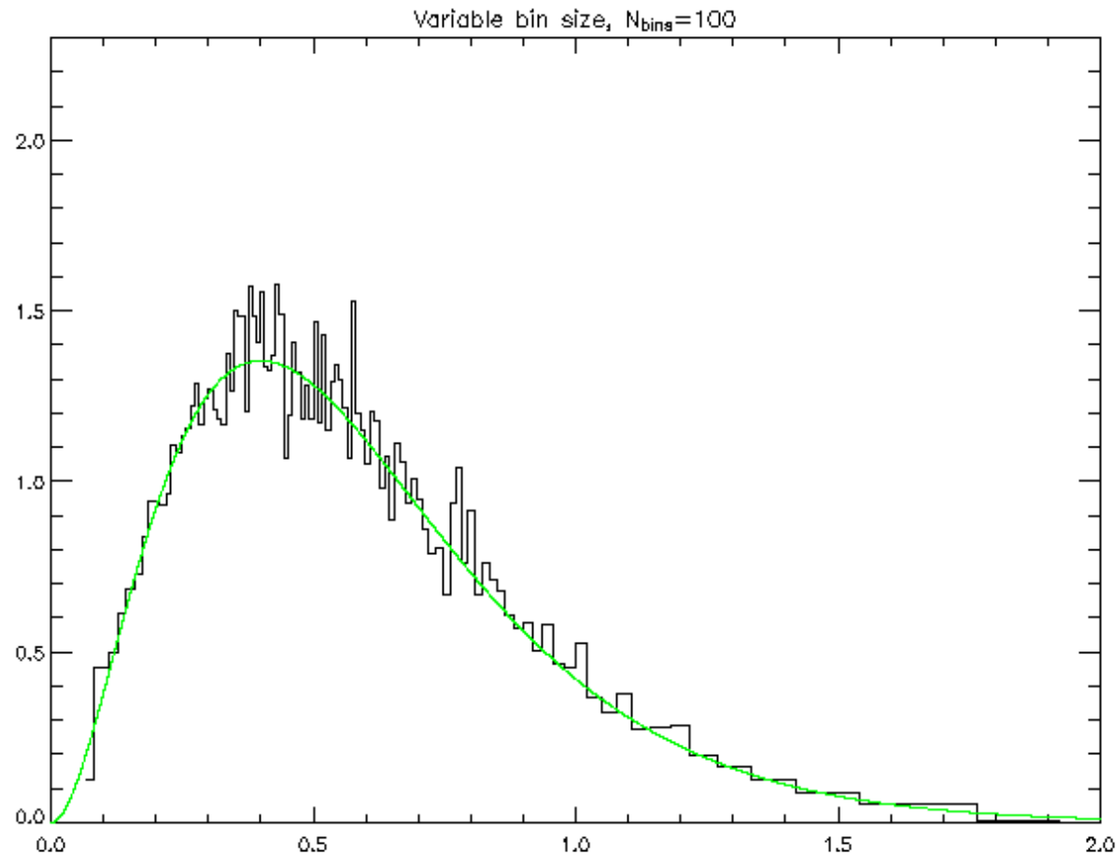
Count objects in bins. Assume the bin size of bin i is Δ_i and that there are n_i objects in this bin. The probability of the value x_i corresponding to that bin is then:

$$p_i(x_i) = \frac{n_i}{N\Delta_i}$$



Histogram Problems

We can adapt the bin sizes to the data, but even so there are two major problems with the histogram:



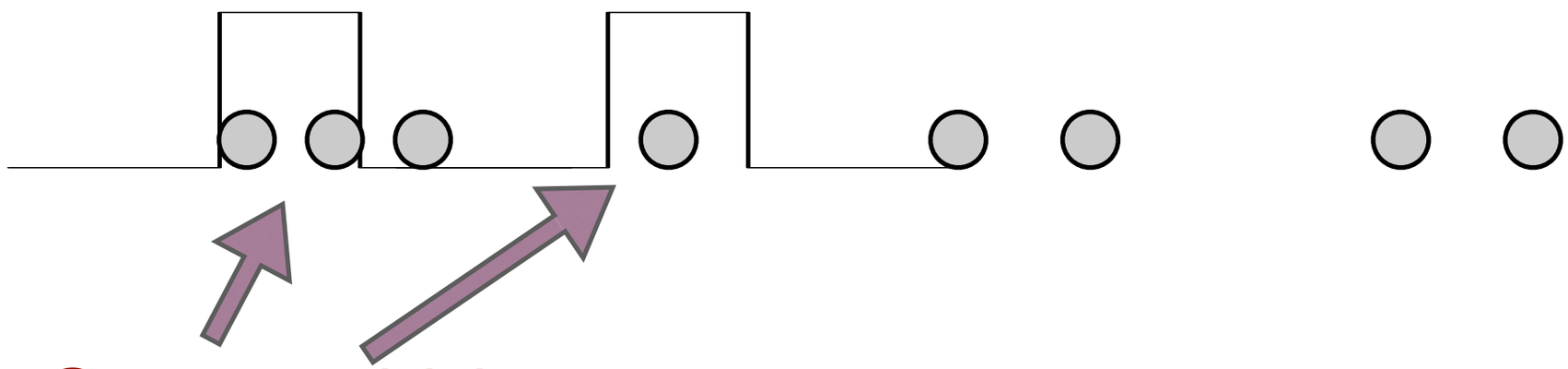
a) It is discontinuous

b) It does not scale well to higher dimensions!

But it is still very useful! [some authors disregard it unnecessarily]
To do better we need to refine our density estimator.

Histograms & kernels

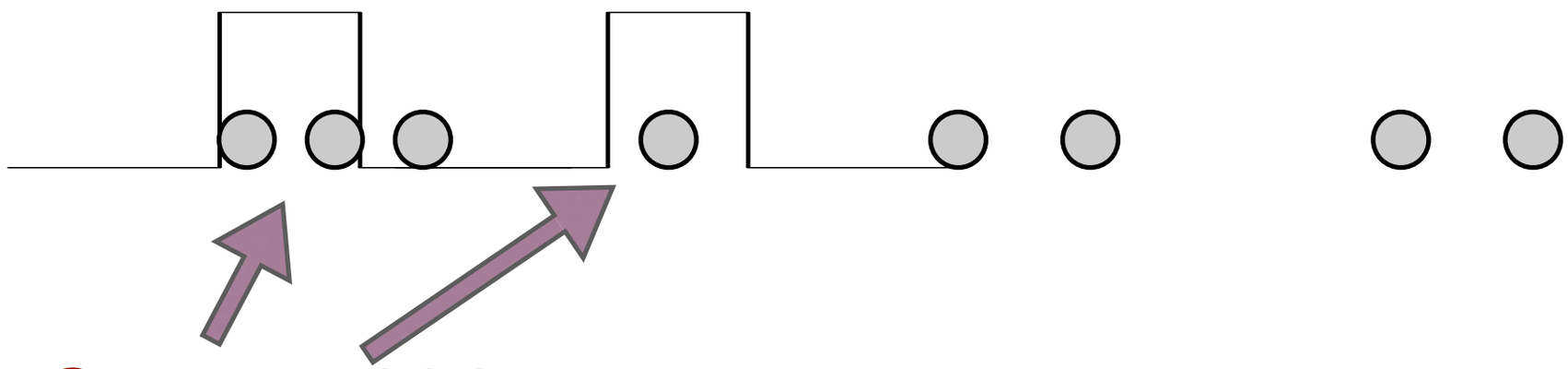
We want to add together effects from points close to **x**.



Count within

Histograms & kernels

We want to add together effects from points close to **x**.

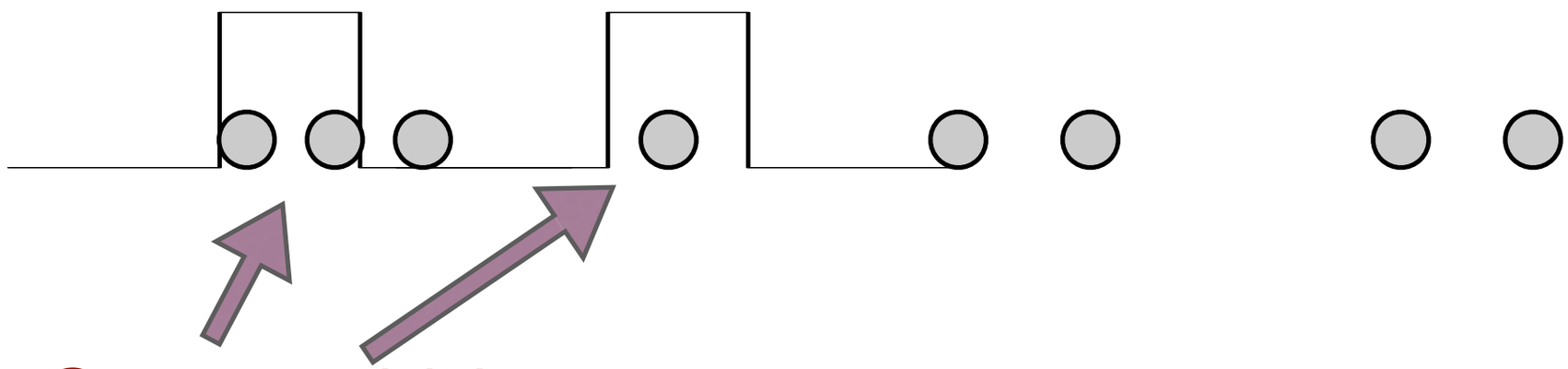


Count within

Define a k(x) so that: $k(\mathbf{u}) = 1 \text{ iff } \forall |u_i| < 1/2$

Histograms & kernels

We want to add together effects from points close to **x**.



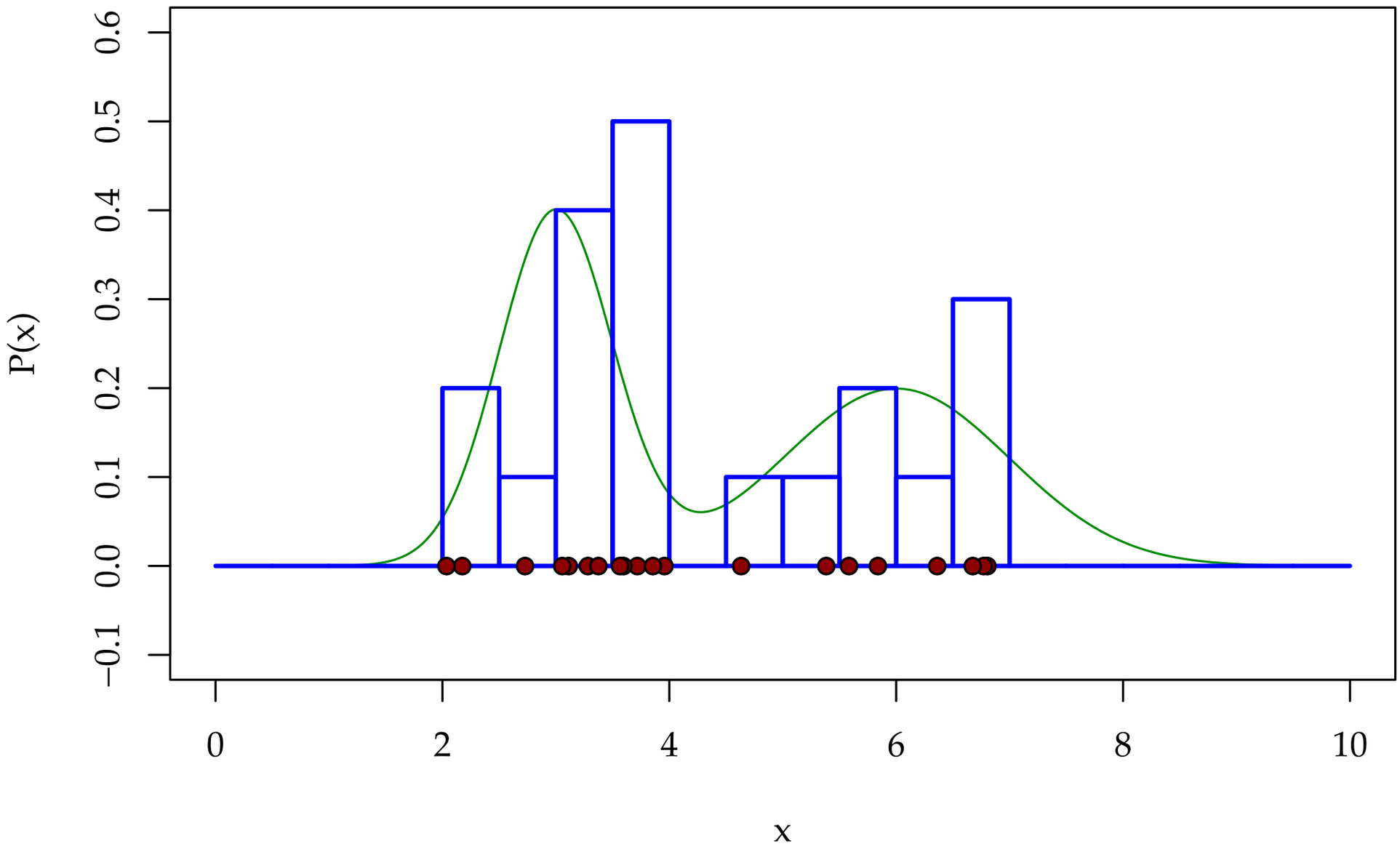
Count within

Define a k(x) so that: $k(\mathbf{u}) = 1 \text{ iff } \forall |u_i| < 1/2$

Then the histogram (number of counts) can be defined as:

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

Histograms



What we are doing here is adding one box for each data point. This is our ‘kernel’.

Histograms & kernels

The histogram (number of counts) can be defined as:

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

Recasting this as a probability distribution we get:

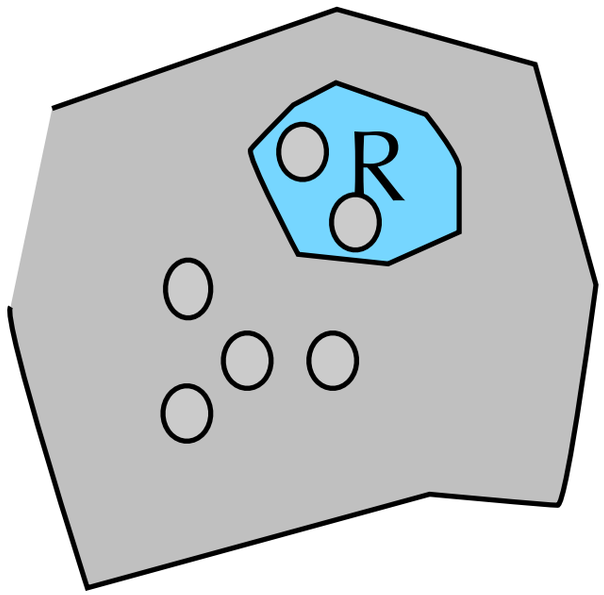
$$p_i(x_i) = \frac{1}{N}\sum_{j=1}^{N}\frac{1}{h}K\left(\frac{x_i - x_j}{h}\right) \qquad K(x) = \begin{cases} 1, & |x| \leq \frac{1}{2} \\ 0, & |x| > \frac{1}{2} \end{cases}$$

this is a good starting point for density estimators

Simple Non-parametric estimators

Kernel estimators

Like in the histogram method we want to use objects near to a point to help us estimate the density there. If we look at a small region:



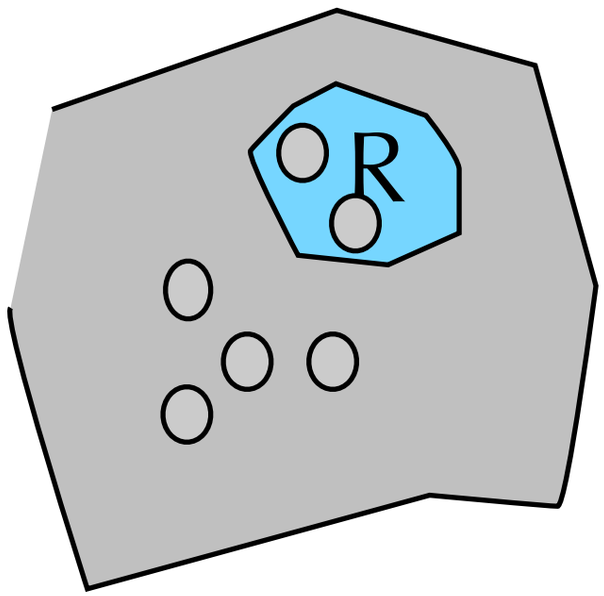
N observations in total
K inside R
V is volume of R

$$p(\mathbf{x}) = \frac{K}{NV}$$

Simple Non-parametric estimators

Kernel estimators

Like in the histogram method we want to use objects near to a point to help us estimate the density there. If we look at a small region:



N observations in total
K inside R
V is volume of R

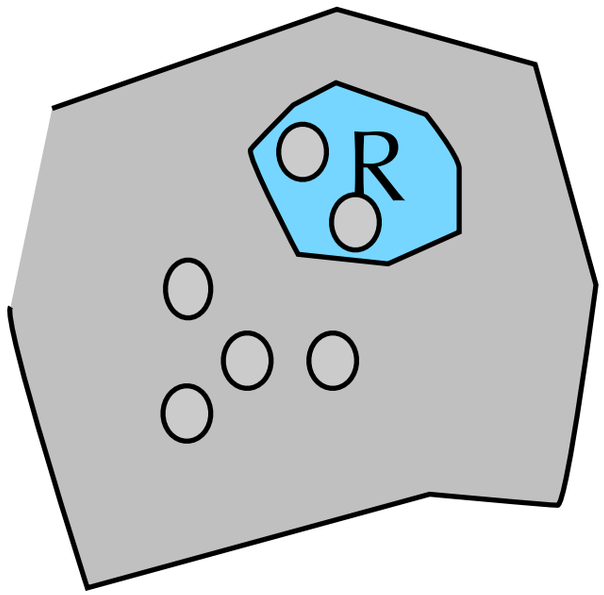
$$p(\mathbf{x}) = \frac{K}{NV}$$

So we can either fix K or fix V

Simple Non-parametric estimators

Kernel estimators

Like in the histogram method we want to use objects near to a point to help us estimate the density there. If we look at a small region:



N observations in total
K inside R
V is volume of R

$$p(\mathbf{x}) = \frac{K}{NV}$$

So we can either fix K or fix V

Kernel estimators fix V and the **k-nearest neighbours estimator** fixes K.

These are one type of instance-based learning methods

Instance-based learning

Basically: methods that use all training data to evaluate new data.

E.g. compare a linear method:

$$\hat{y}(x_i) = \theta_0 + \sum_{j=1}^{N} \theta_j \phi_j(x_i)$$

with using the nearest neighbours:

$$\hat{y}(x_i) = \frac{1}{k} \sum_{j:x_j \in N_k(x_i)} y_j$$

Instance-based learning

These are very useful techniques and are often overlooked.

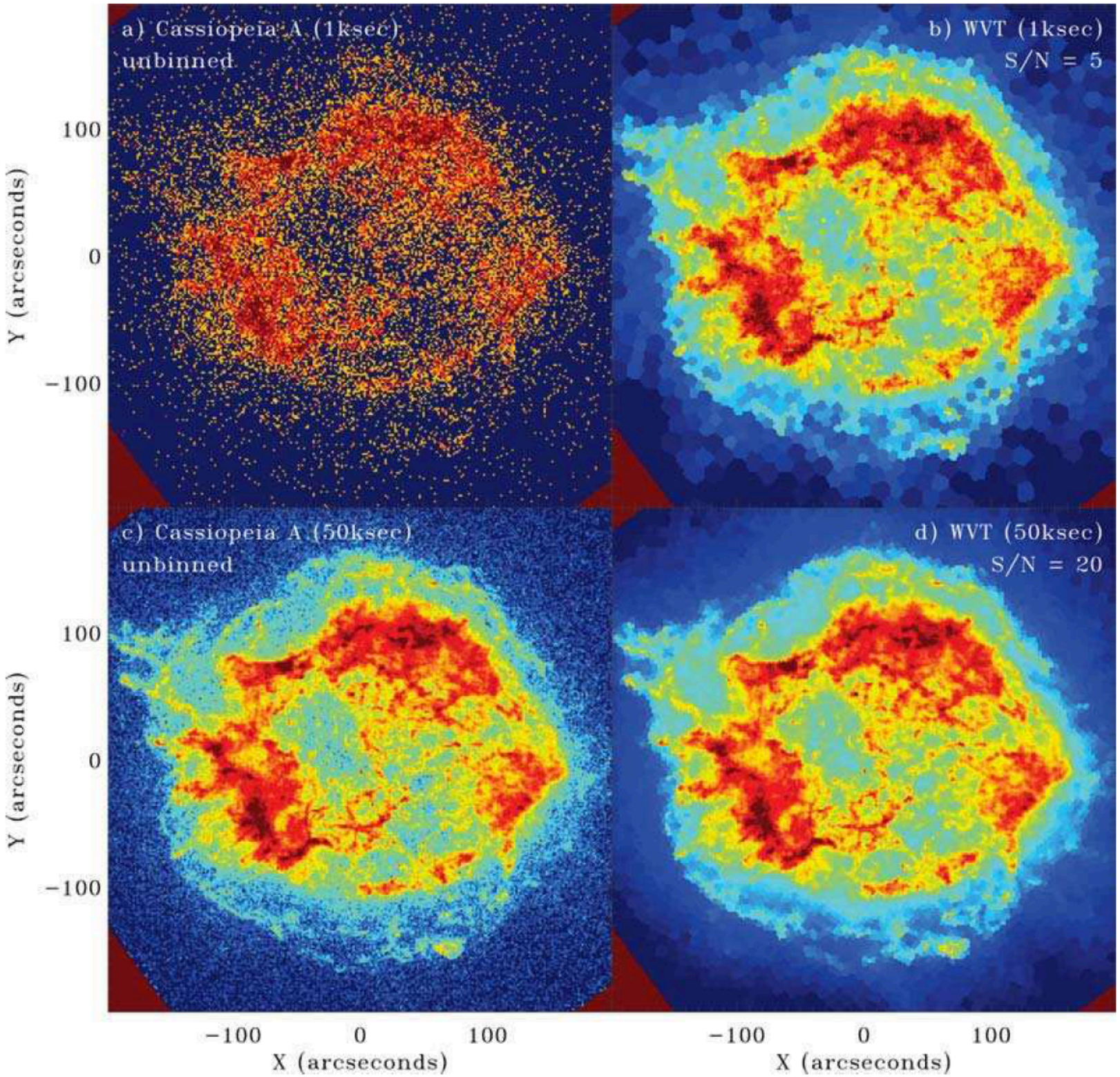
Pros:

- **Flexible** - can approximate very complex functions fairly easily
- **Can adapt to new data** - and throw away old
- **Simple**
- **Scales to large data**

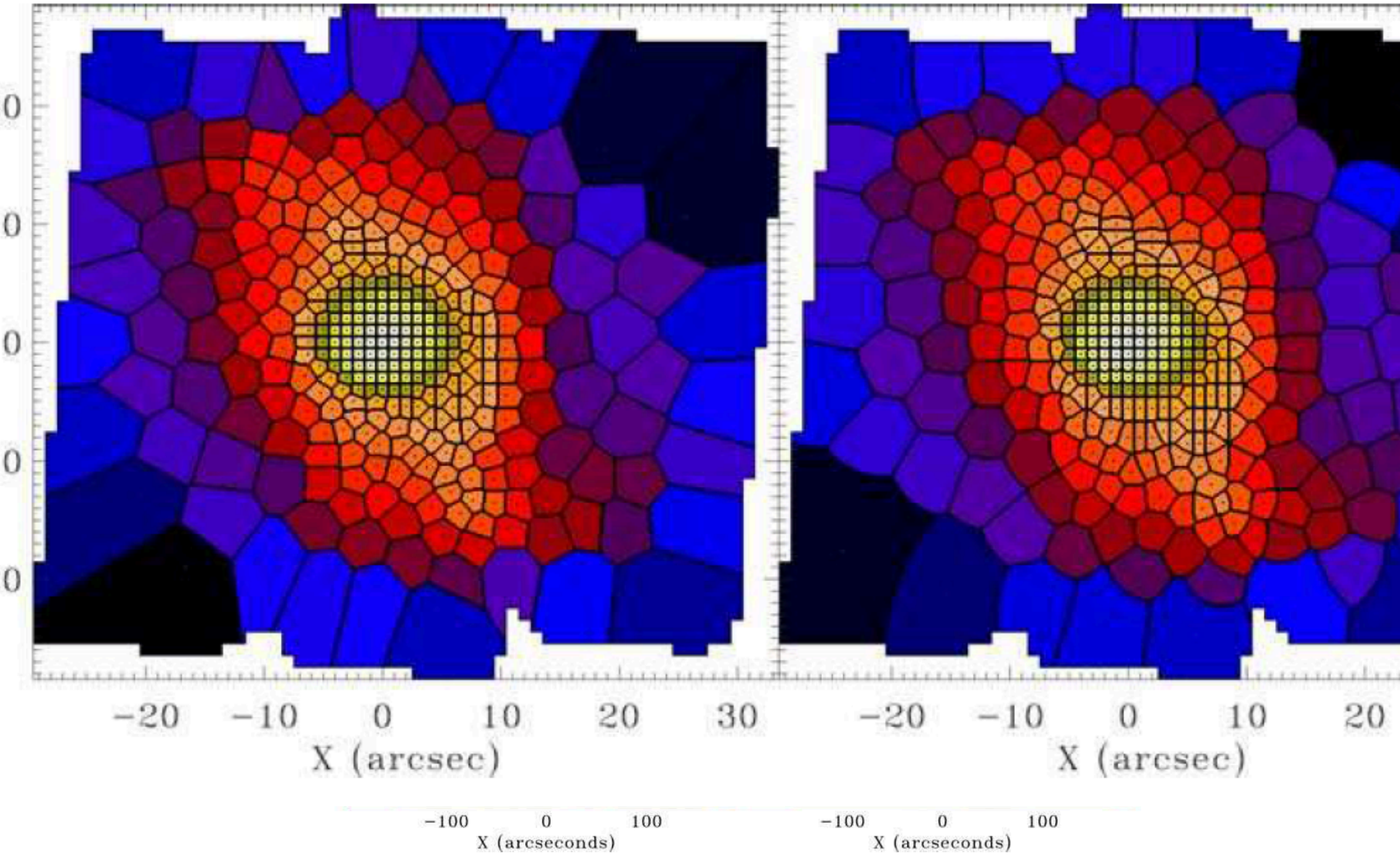
Cons:

- **Memory expensive** - all data needs to be in memory
- **Time expensive** - naïve implementations need comparison to all data when a new example is found
- **Opaque** - it is difficult to gain understanding from these

Diehl & Statler (2006)



Diehl & Statler (2006)



Kernel methods

We want to add together effects from points close to **x.**

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

This is the basic equation for kernel estimation

Different kernels lead to somewhat different final distributions. The Epanchenikov kernel is the one that theoretically gives the minimum variance.

Kernel methods

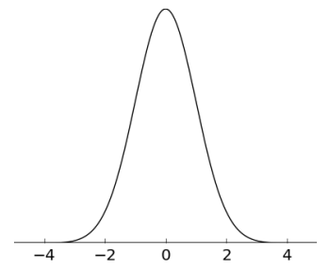
We want to add together effects from points close to **x.**

Square kernel:

$$k(\mathbf{u}) = 1 \text{ iff } \forall |u_i| < 1/2$$

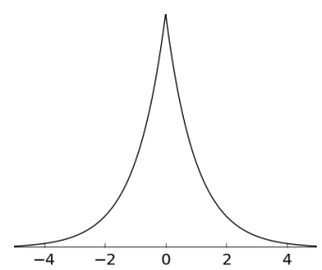
Gaussian kernel:

$$k(\mathbf{u}) = \frac{1}{(2\pi)^{1/D}} e^{-\|\mathbf{u}\|^2/2}$$



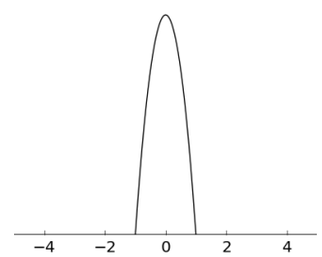
Exponential kernel:

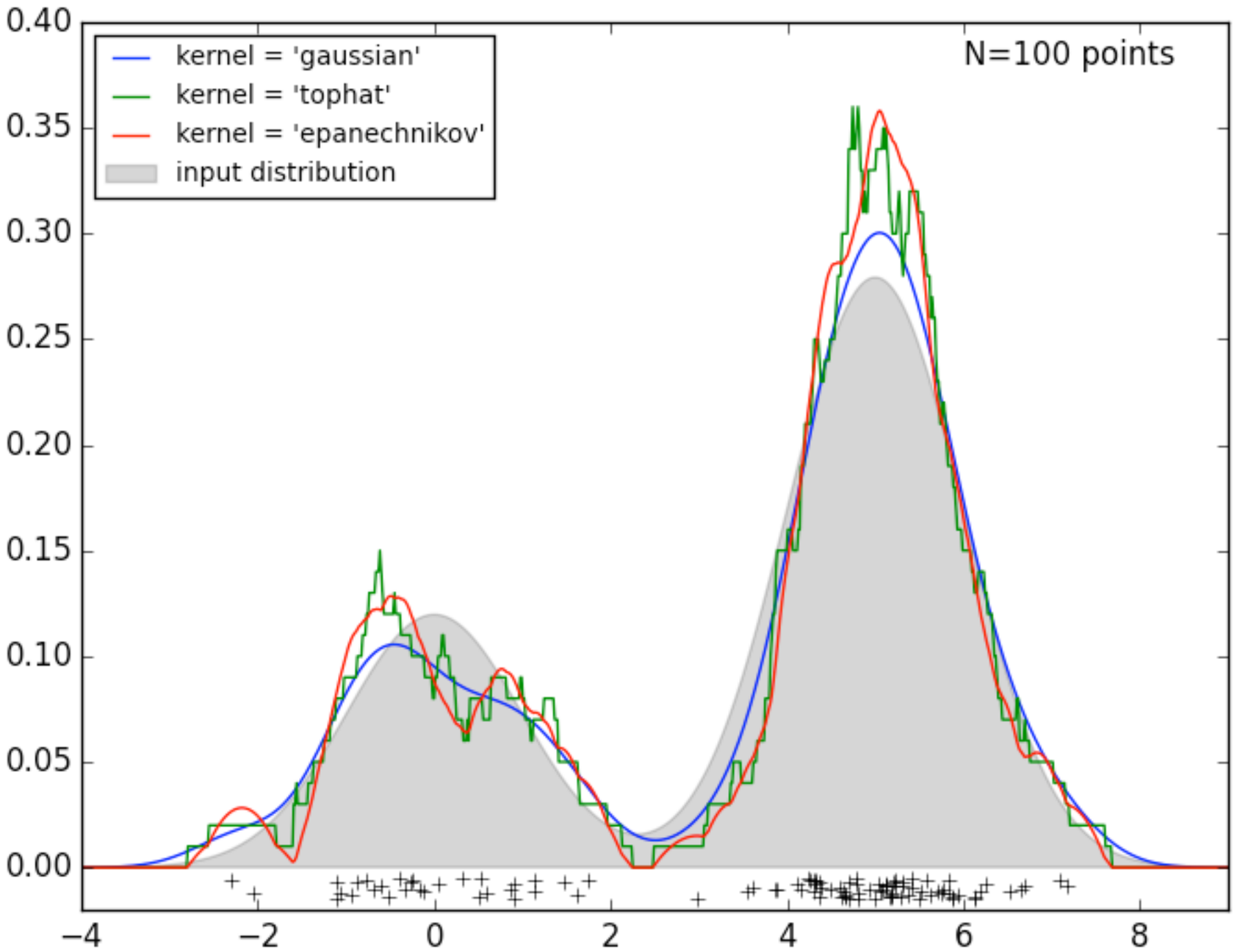
$$k(\boldsymbol{u}) = \frac{1}{D! V_D(1)} e^{-|u|}$$



Epanechnikov kernel:

$$k(\boldsymbol{u}) = \begin{cases} \frac{3}{4}\left(1 - u^2\right), & \text{if } |u| < 1 \\ 0 & \text{otherwise} \end{cases}$$





Taken from: http://scikit-learn.org/stable/auto_examples/neighbors/plot_kde_1d.html

Kernel methods

We want to add together effects from points close to **x.**

$$p(\mathbf{x}) = \frac{1}{N}\sum_{i=1}^{N}\frac{1}{V}K\left(\frac{\mathbf{x}-\mathbf{x}_i}{h}\right)$$

This is the basic equation for kernel density estimation (KDE)

Note that this **has** a parameter: the bandwidth - changing this changes the results.

Kernel methods

We want to add together effects from points close to **x.**

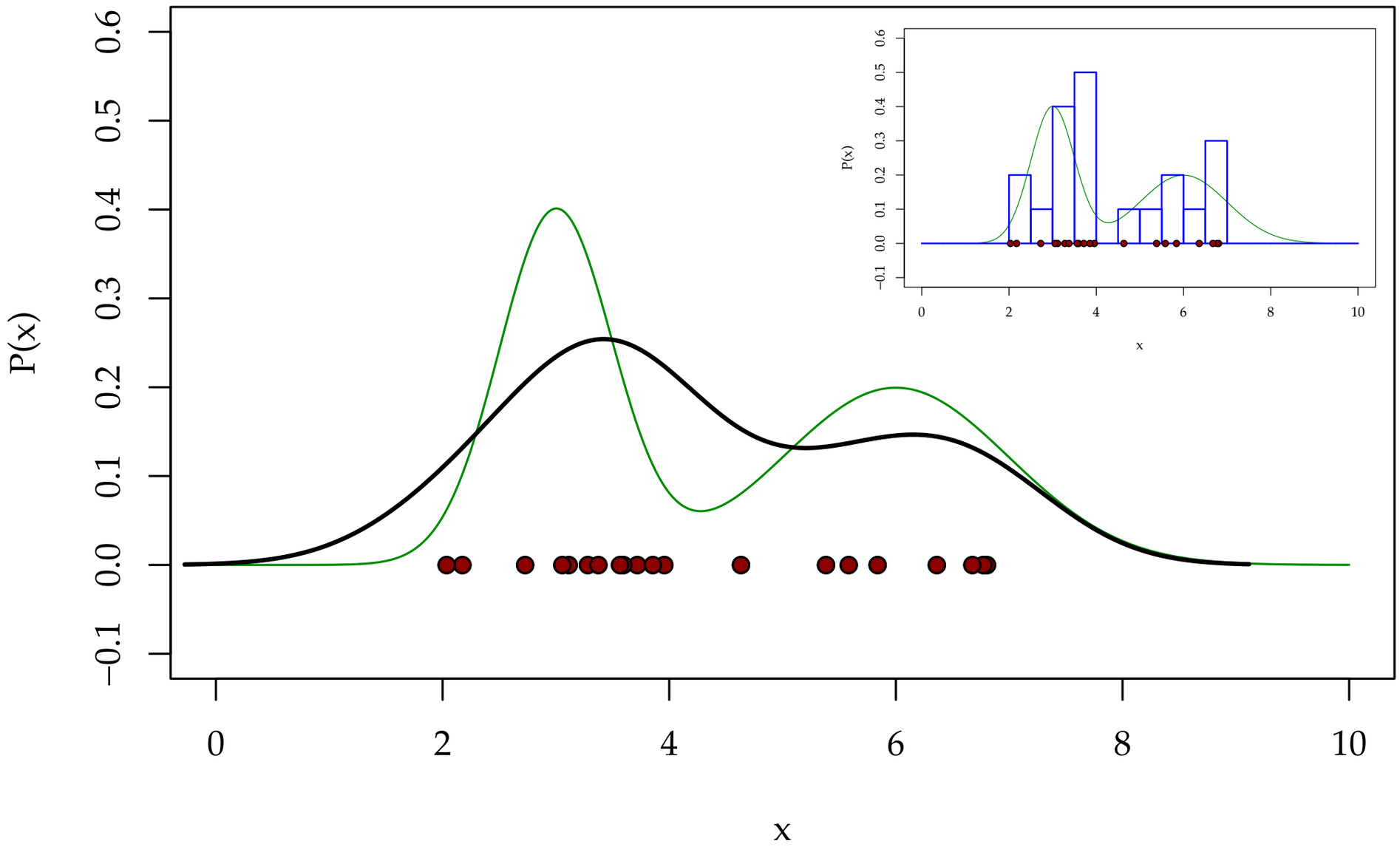
$$p(\mathbf{x}) = \frac{1}{N}\sum_{i=1}^{N}\frac{1}{V}K\left(\frac{\mathbf{x}-\mathbf{x}_i}{h}\right)$$

Bandwidth

This is the basic equation for kernel density estimation (KDE)

Note that this **has** a parameter: the bandwidth - changing this changes the results.

Count the number weighted by a kernel



Practical use:

```
from sklearn.neighbors import KernelDensity

kde = KernelDensity(<bandwidth<, kernel='gaussian').fit(X)
Xgrid = <same shape as X: N_samples x N_features>

log_dens = kde.score_samples(Xgrid)
```

Try it out

Load the Datafiles/catalog-of-pulsars.vot file from the course website:

from astropy.table import Table
Table().read('catalog-of-pulsars.vot')

Create a kernel density estimate for the galactic latitude (GLAT) - can you find any structure here?

How do you do it?

Kernels to expansions - radial basis functions

Inspired by this we can expand the function p(x) using the kernel functions

$$p(\mathbf{x}) = \frac{1}{N}\sum_{i=1}^{N}\frac{1}{V}K\left(\frac{\mathbf{x}-\mathbf{x}_i}{h}\right)$$

This gives radial basis function expansion. Very useful for interpolation.

$$f(\mathbf{x}) = \sum_{i=1}^{N}\beta_i K\left(\frac{\mathbf{x}-\boldsymbol{\xi}_i}{\lambda_i}\right)$$

[see e.g. chap. 6.7 in the Elements of Statistical Learning]

By making this a bit more flexible we get a very useful density estimation technique.

When is good, good enough?

Model choice/complexity decision

Key ingredients of machine learning

Reminder

Accuracy/error-rate

How well is our algorithm doing? A common measure - the mean squared error:

$$\mathrm{MSE} = \frac{1}{N}\sum_{i=1}^{N}\left(y - y_{\mathrm{pred}}\right)^2$$

Key ingredients of machine learning

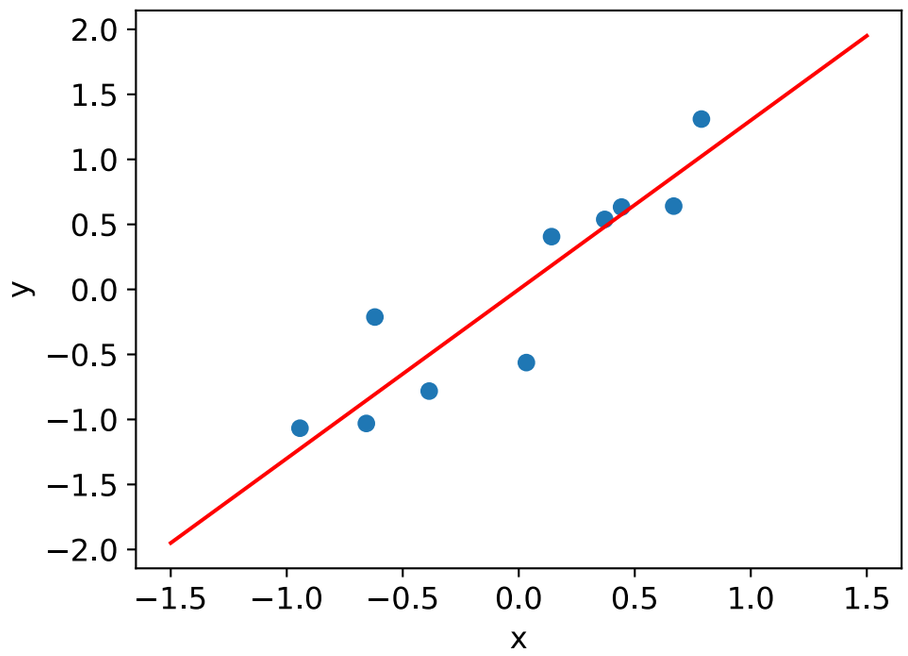
Reminder

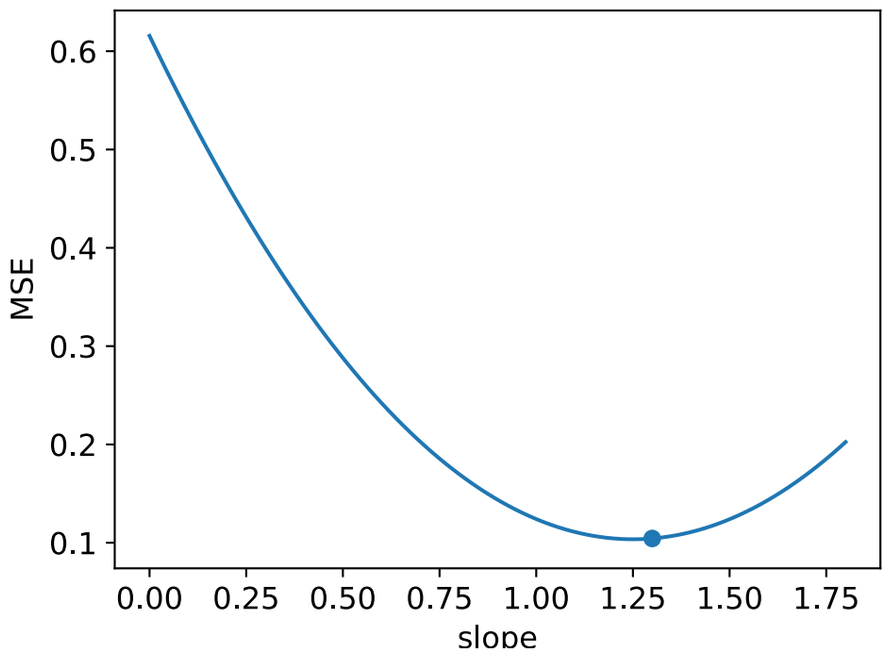
Accuracy/error-rate

How well is our algorithm doing? A common measure - the mean squared error:

$$\mathrm{MSE} = \frac{1}{N}\sum_{i=1}^{N}\left(y - y_{\mathrm{pred}}\right)^2$$

Try this on linear regression $y = \mathrm{slope} \times x$



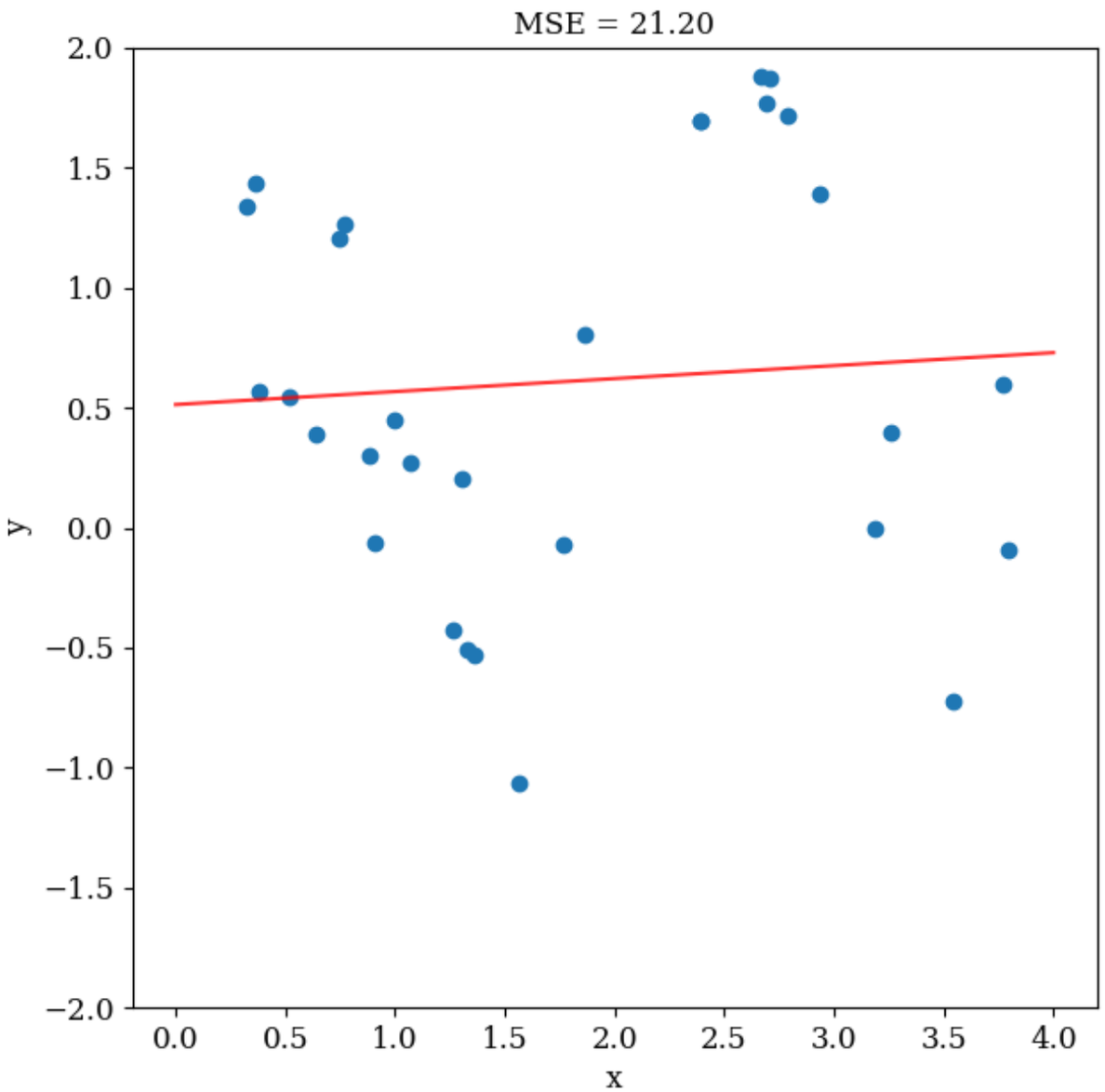


A simple example - fitting a polynomial

Reminder

Overfitting, underfitting

We want our method to **generalise** well



High bias!

A simple example - fitting a polynomial

Reminder

How do you do it?

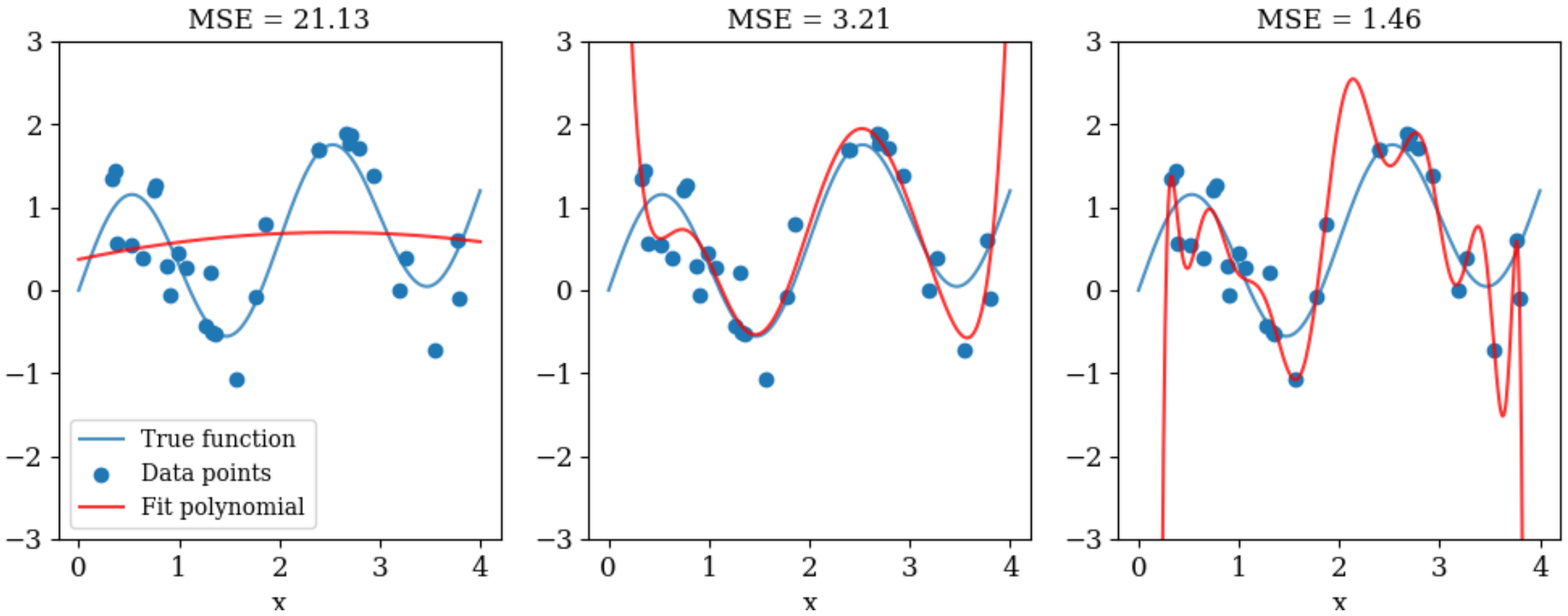
```
import numpy as np
from numpy.polynomial import Polynomial

<… Get data x & y …>

p = Polynomial.fit(x, y, degree)
y_pred = p(x)
```

And we can also easily calculate the MSE:

```
mse = np.sum((y_pred-y)**2)
```

Increasing the flexibility:

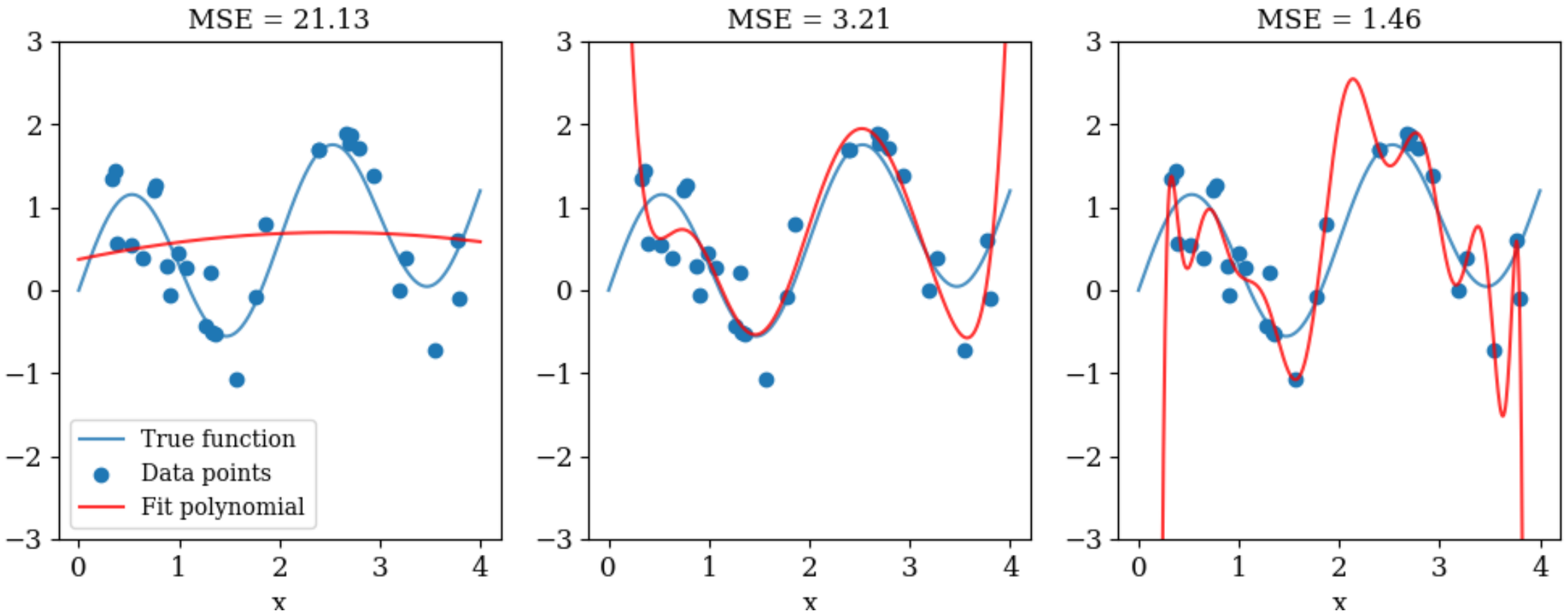
Reminder



$$\mathrm{MSE} = \frac{1}{N}\sum_{i=1}^{N}(y - y_{\mathrm{pred}})^2$$

Increasing the flexibility:

Reminder



Underfitting

Overfitting

$$\mathrm{MSE} = \frac{1}{N}\sum_{i=1}^{N}(y - y_{\mathrm{pred}})^2$$

Key ingredients of machine learning

Reminder

Parameters, hyper-parameters

Consider a polynomial:

$$y = \sum_{i=0}^{M} a_i x^i$$

Here a_i are the parameters of the model and what our machine learning algorithm will give us.

But M - the maximum polynomial power is *also* a parameter. It is a different kind of parameter and we call this a **hyperparameter**.

How do you decide

Reminder

We will return to this but we should think of dividing our data in three:

The training set

This is what we use to find the best-fitting parameters and we minimise the **training error** (e.g. MSE) on this.

The validation set

We use this to determine the optimal settings of the hyper-parameters.

The test set

We evaluate the generalisation error of our algorithm on this sample. These data are *not* used for the fitting. We refer to the error achieved on this set as the **test error**.

Taking this for our polynomial example

1. For each polynomial order M, fit a polynomial to the test data.
2. Use this fitted polynomial on the validation set and calculate an MSE for this.
3. Repeat for the next M
4. Choose the M that gives the smallest MSE on the validation set.
5. Calculate the MSE on the test set to get a quantitative estimate of how well the calculation did go.

How do you do it?

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

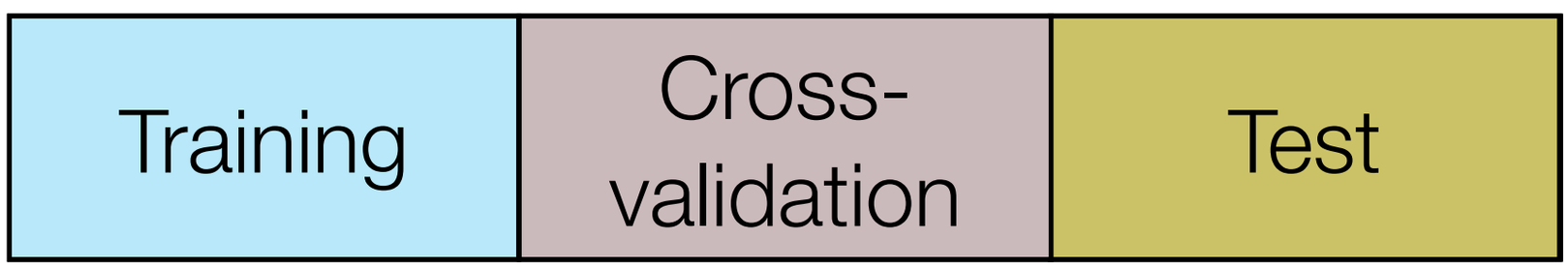
This is fine if we have 100,000s of data (usually), but can leave too few points to train on.

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.



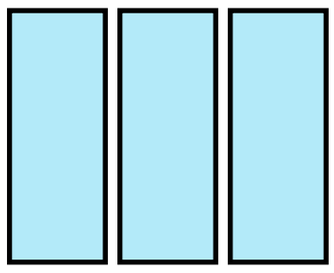
One possibility

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

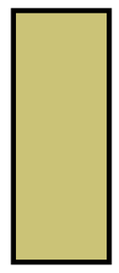
k-fold CV



Training

.....

k-copies



Test

Do this splitting k times.

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

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.

$$\ln L_{CV} = \frac{1}{k}\sum_{i=1}^{k} \ln p_{x_i}(\boldsymbol{z_i})$$

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.cross_validation package:

```
from sklearn.cross_validation import KFold

# Create folds for N data points in n_folds:
kf = KFold(N, n_folds=n_folds)

for train, test in kf:
                x_train = x[train, :]
                x_test = x[test, :]

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

Short-cutting CV - BIC & AIC

A full CV evaluation can be very costly time-wise - in that case you might want a cheaper way to assess quality of fit.

Enter the information criteria:

The Bayesian and the Akaike, BIC & AIC

$$\mathrm{BIC} = -2 \ln L_{\max} + k \ln N$$

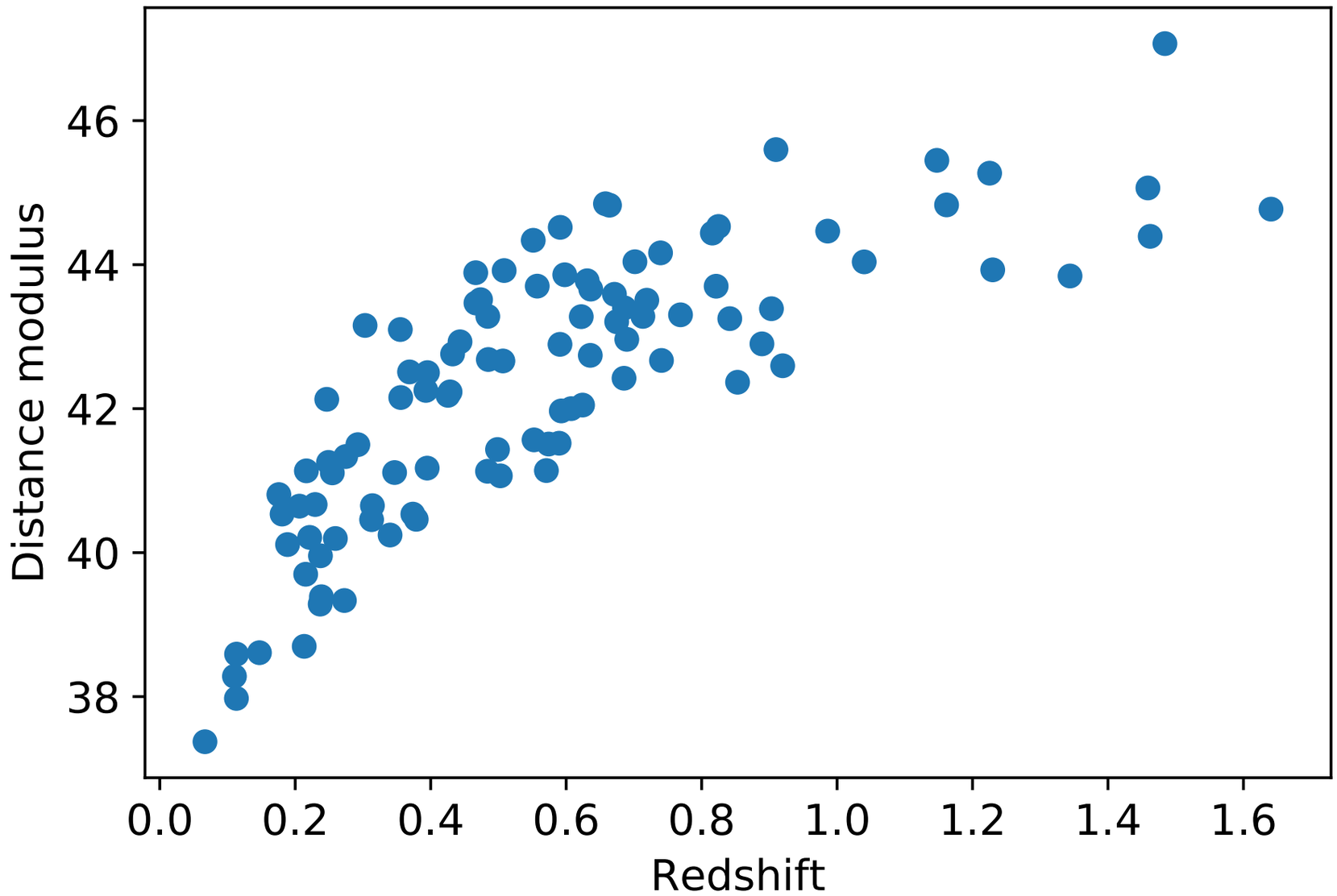
$$\mathrm{AIC} = -2 \ln L_{\max} + 2k + \frac{2k(k+1)}{N-k-1}$$

k: number of parameters

N: number of data points

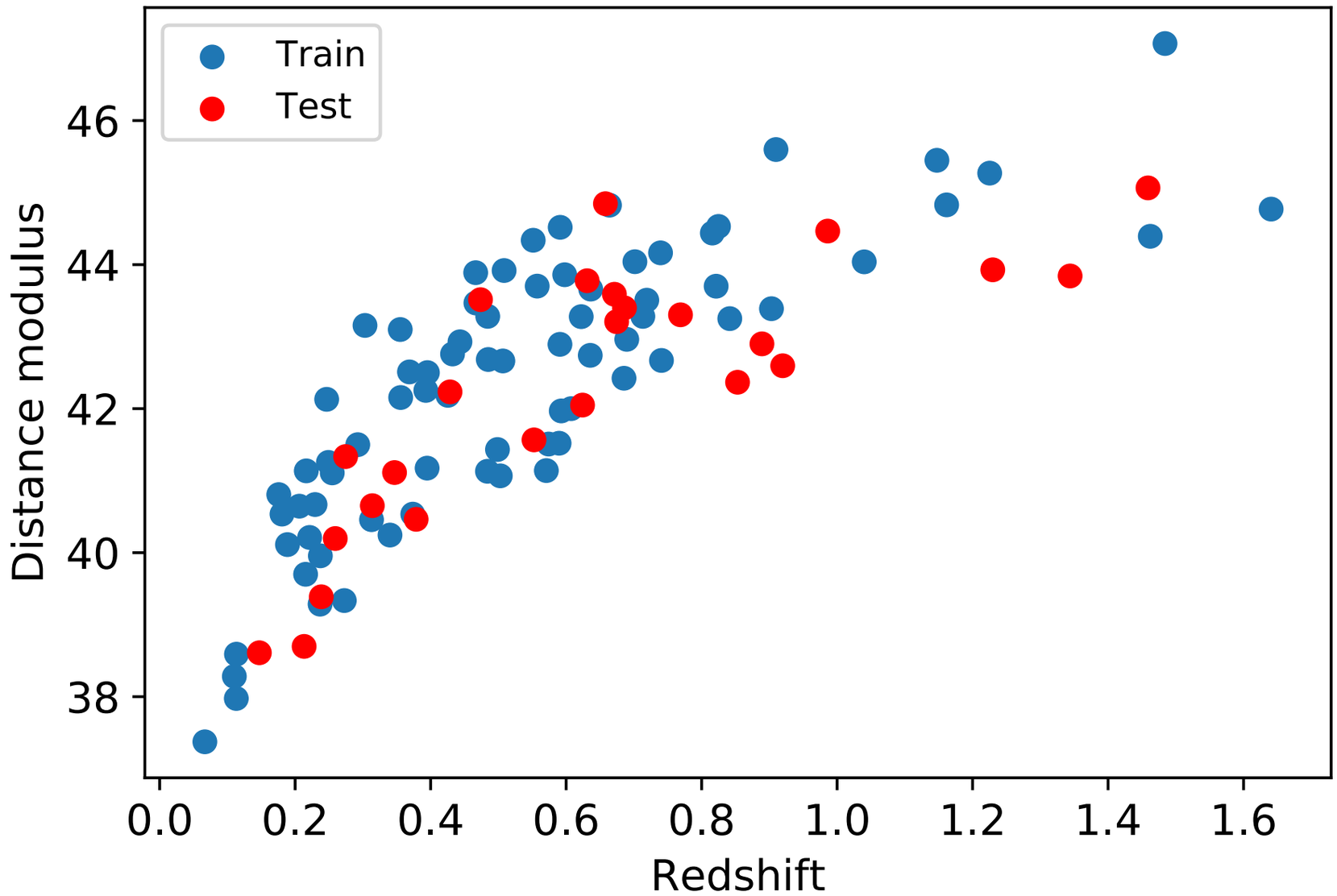
$L_{\max}$: max likelihood

A simple example




```
from sklearn.model_selection import train_test_split

indices = np.arange(n_sample, dtype=np.int)
i_train, i_test = train_test_split(indices)
```

A simple example




```
plt.scatter(z_sample[i_train], mu_sample[i_train],
            label='Train')
plt.scatter(z_sample[i_test], mu_sample[i_test],
            color='red', label='Test')
```

A simple example

A simple example

Fit the training sample:

```
orders = np.arange(n_orders)
for i, order in enumerate(orders):
    # Fit the training sample using polyfit
    p = np.polyfit(z_sample[i_train], mu_sample[i_train], order)
```

A simple example

Fit the training sample:

```
orders = np.arange(n_orders)
for i, order in enumerate(orders):
    # Fit the training sample using polyfit
    p = np.polyfit(z_sample[i_train], mu_sample[i_train], order)
```

Calculate quality of fit on training sample:

```
  mu_fit_train = np.polyval(p, z_sample[i_train])
  mu_train = mu_sample[i_train]
  MSE_train[i] = np.sum((mu_train-mu_fit_train)**2)/n_train
```

A simple example

Fit the training sample:

```
orders = np.arange(n_orders)
for i, order in enumerate(orders):
    # Fit the training sample using polyfit
    p = np.polyfit(z_sample[i_train], mu_sample[i_train], order)
```

Calculate quality of fit on training sample:

```
mu_fit_train = np.polyval(p, z_sample[i_train])
mu_train = mu_sample[i_train]
MSE_train[i] = np.sum((mu_train-mu_fit_train)**2)/n_train
```

Calculate quality of fit on test sample:

```
mu_fit_test = np.polyval(p, z_sample[i_test])
mu_test = mu_sample[i_test]
MSE_train[i] = np.sum((mu_test-mu_fit_test)**2)/n_test
```

A simple example

Fit the training sample:

```
orders = np.arange(n_orders)
for i, order in enumerate(orders):
    # Fit the training sample using polyfit
    p = np.polyfit(z_sample[i_train], mu_sample[i_train], order)
```

Calculate quality of fit on training sample:

```
mu_fit_train = np.polyval(p, z_sample[i_train])
mu_train = mu_sample[i_train]
MSE_train[i] = np.sum((mu_train-mu_fit_train)**2)/n_train
```

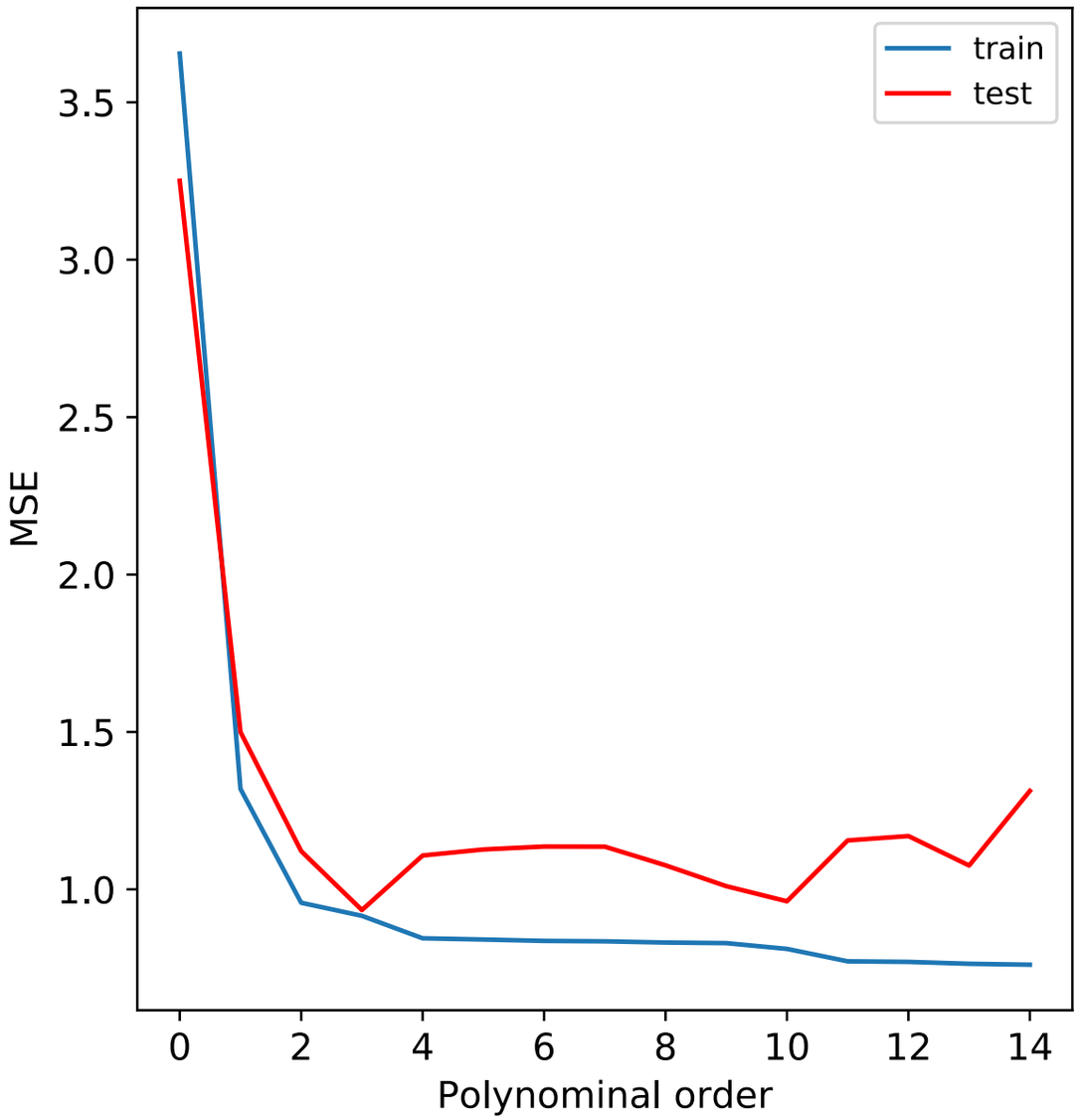
Calculate quality of fit on test sample:

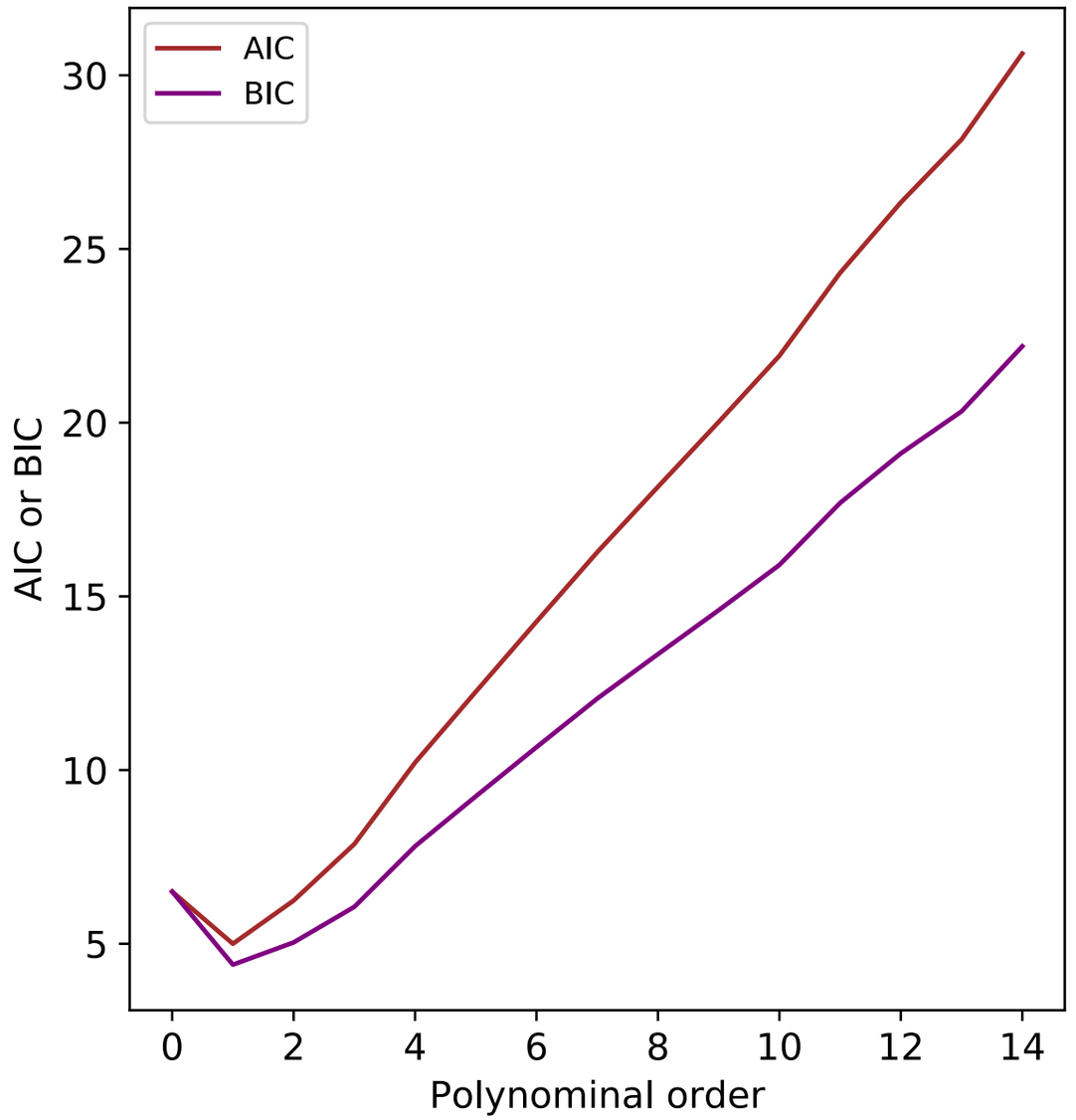
```
mu_fit_test = np.polyval(p, z_sample[i_test])
mu_test = mu_sample[i_test]
MSE_train[i] = np.sum((mu_test-mu_fit_test)**2)/n_test
```

and for completeness, calculate BIC & AIC:

```
BIC[i] = 2*MSE_test[i] + order*np.log10(len(i_test))
AIC[i] = 2*order + 2*MSE_test[i]
```

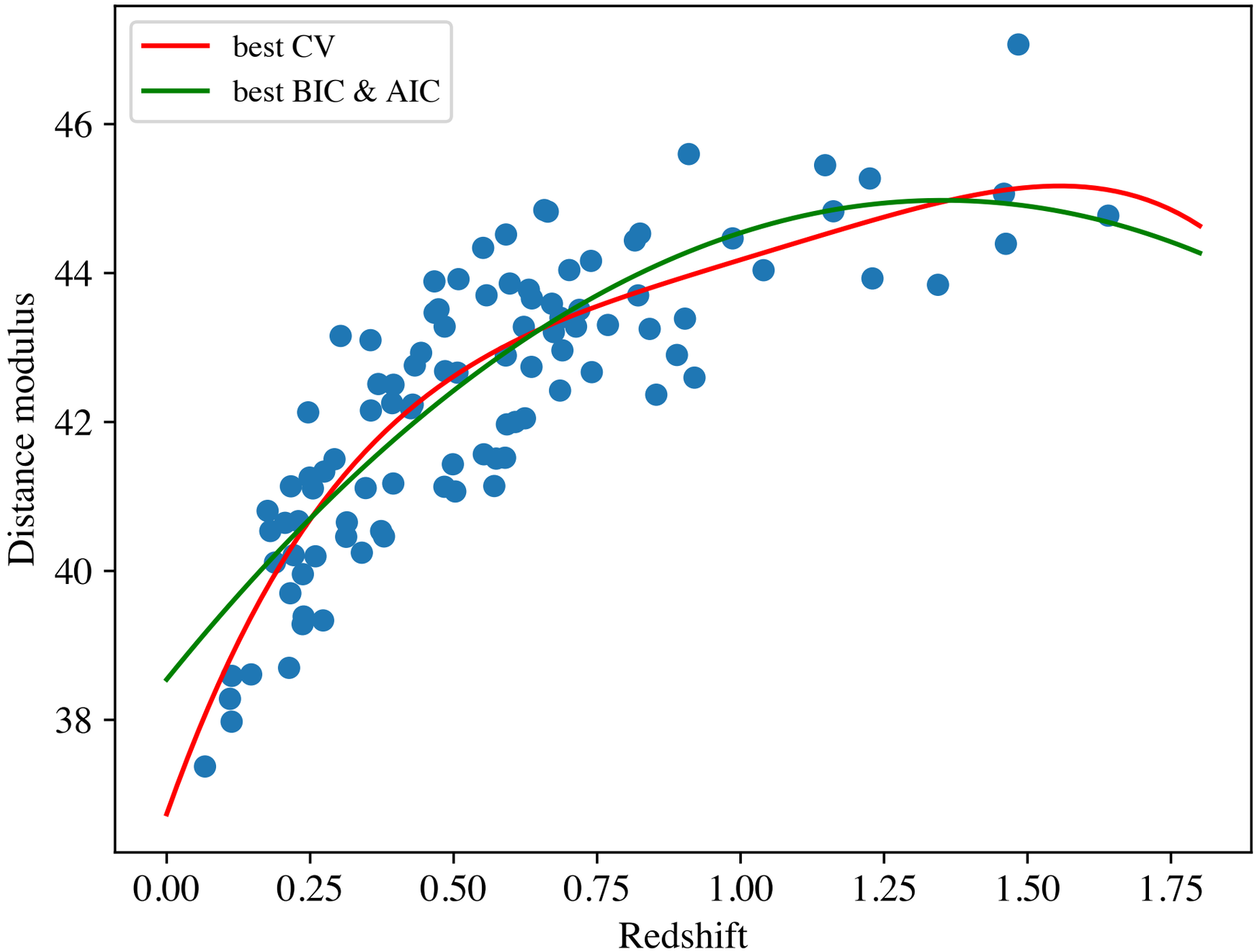
A simple example - result





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.

Testing it out

Find the Density estimation-MLD2024.ipynb file in the Lecture 2/Notebooks directory.

This shows you how to do cross-validation for a set of stars from Gaia. Now try to do this for the pulsar file you loaded earlier and find the best kernel density estimate for the Galactic latitude. What bandwidth works best?

How do you do it?

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 mean 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-

Expected mean square error at $x=x_0$:

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

$$y = f(x) + \epsilon$$

$$\epsilon \sim N(0, \sigma_\epsilon)$$

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-

Expected mean square error at $x=x_0$:

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

$$y = f(x) + \epsilon$$

$$\epsilon \sim N(0, \sigma_\epsilon)$$

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

Small versus big bins - the bias variance trade-

Expected mean square error at $x=x_0$:

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

$$y = f(x) + \epsilon$$

$$\epsilon \sim N(0, \sigma_\epsilon)$$

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-

Expected mean square error at $x=x_0$:

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

$$y = f(x) + \epsilon$$

$$\epsilon \sim N(0, \sigma_\epsilon)$$

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

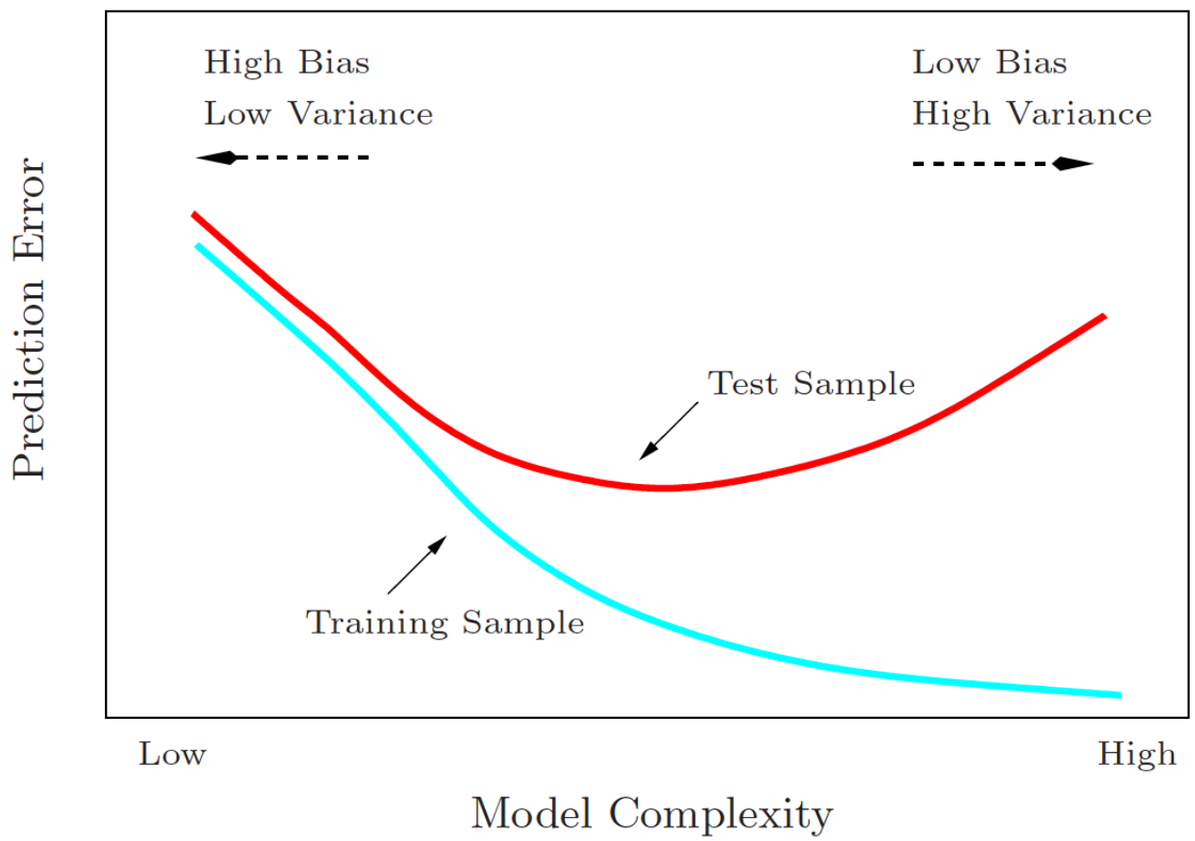
Bias^2

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

Gaussian Mixture Modeling

Gaussian mixture modelling

Let us assume that our unknown distribution can be written as a sum of Gaussians:

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

We need to estimate the parameters and can write the log likelihood of all data as

$$\ln L = \sum_{i=1}^{N} \ln \left[\sum_{j=1}^{M} \alpha_j N(x_i; \mu_j, \sigma_j) \right]$$

Gaussian mixture modelling

Let us assume that our unknown distribution can be written as a sum of Gaussians:

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

We need to estimate the parameters and can write the log likelihood of all data as

$$\ln L = \sum_{i=1}^{N} \ln \left[\sum_{j=1}^{M} \alpha_j N(x_i; \mu_j, \sigma_j) \right]$$

And this now needs to be maximised to determine the *3M-1* parameters [why -1?]

Gaussian mixture modelling

In this case, the parameters are found using the Expectation-Maximisation algorithm:

1. Calculate the log likelihood of the data [E]

$$\ln L = \sum_{i=1}^{N} \ln \left[\sum_{j=1}^{M} \alpha_j \, N(\mu_j, \sigma_j) \right]$$

2. Find the parameters that maximise the likelihood

and repeat

Gaussian mixture modelling

In this case, the parameters are found using the Expectation-Maximisation algorithm:

1. Calculate the log likelihood of the data [E]

$$\ln L = \sum_{i=1}^{N} \ln \left[\sum_{j=1}^{M} \alpha_j N(\mu_j, \sigma_j) \right]$$

2. Find the parameters that maximise the likelihood

and repeat

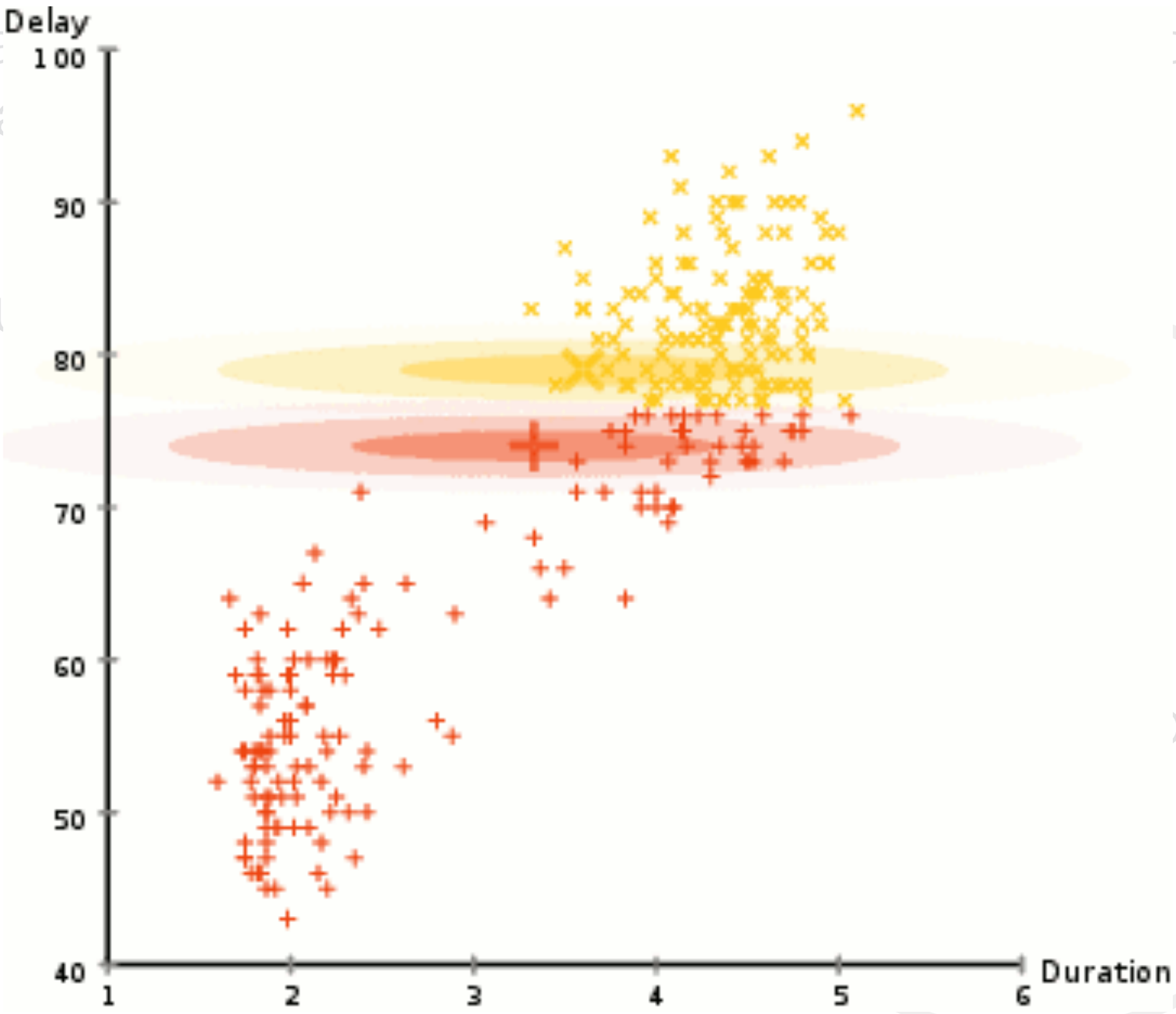
Gaussian mixture modelling

In this ca ctation-
Maximisa

1. Calcu

2. Find od

and repeat



https://en.wikipedia.org/wiki/Expectation%E2%80%93maximization_algorithm#/media/File:EM_Clustering_of_Old_Faithful_data.gif

Gaussian mixture modelling - in practice

(in older versions GMM)

```
from sklearn.mixture import GaussianMixture
```

(you can make mixtures of other things than Gaussians but they are not implemented in sklearn.mixture)

```
model = GaussianMixture(2)
```

Create a model with a given number of components - here two - this number can be known *a priori*, or will have to be determined as the kernel bandwidth.

```
res = model.fit(X)
```

fit the data - just as with regression or kernel density estimation

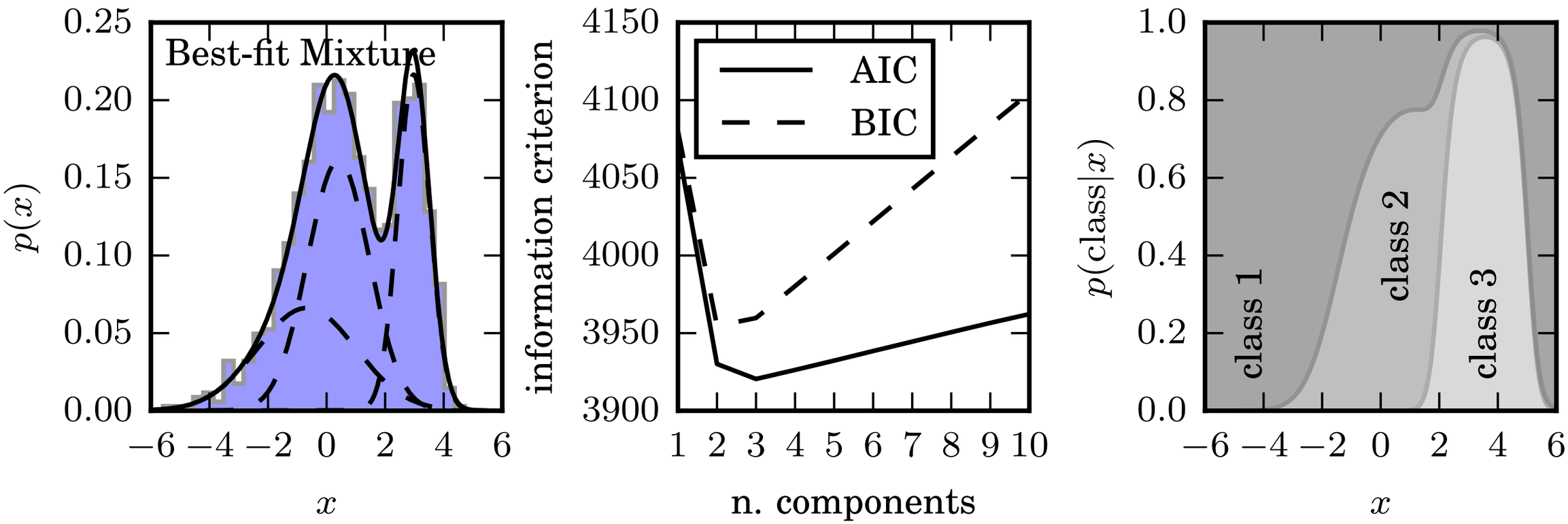
```
res.bic(X)
```

```
print model.means_
```

You can get the BIC or AIC easily - these are qualities of fit which we'll get back to soon.

Gaussian mixture modelling

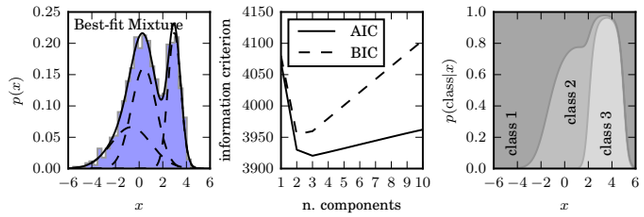
Using the BIC or AIC to find the best number of Gaussians



Based on Figure 4.2 from Statistics, Data Mining, and Machine Learning in Astronomy - see http://www.astroml.org/book_figures/chapter4/fig_GMM_1D.html

Gaussian mixture modelling

Use of AIC/BIC in practice:



X = X[:, np.newaxis]

```
from sklearn.mixture import GaussianMixture

# fit models with 1-10 components
N = range(1, 11)
models = [None for i in N]

for i in range(len(N)):
    models[i] = GaussianMixture(N[i]).fit(X)

# compute the AIC and the BIC
AIC = [m.aic(X) for m in models]
BIC = [m.bic(X) for m in models]
```

Based on Figure 4.2 from Statistics, Data Mining, and Machine Learning in Astronomy - see http://www.astroml.org/book_figures/chapter4/fig_GMM_1D.html

Gaussian mixture modelling

Getting fit results back:

```
logprob, p_component = M_best.score_samples(u)
```

logprob: The log p(x) for the overall model. So how likely data **u** is under this model.

p_component: The likelihood of the data **u** for each individual component.

If u has 10 elements and the GMM has 3 components, p_component will be 10x3.

Gaussian mixture modelling

When do we use GMM?

- When you think your distributions are reasonably described by Gaussians.

- As a way to describe distributions.

- When you want a simple approach that works often.

- For classification/assignment.

Gaussian mixture modelling - degeneracies

Consider the model

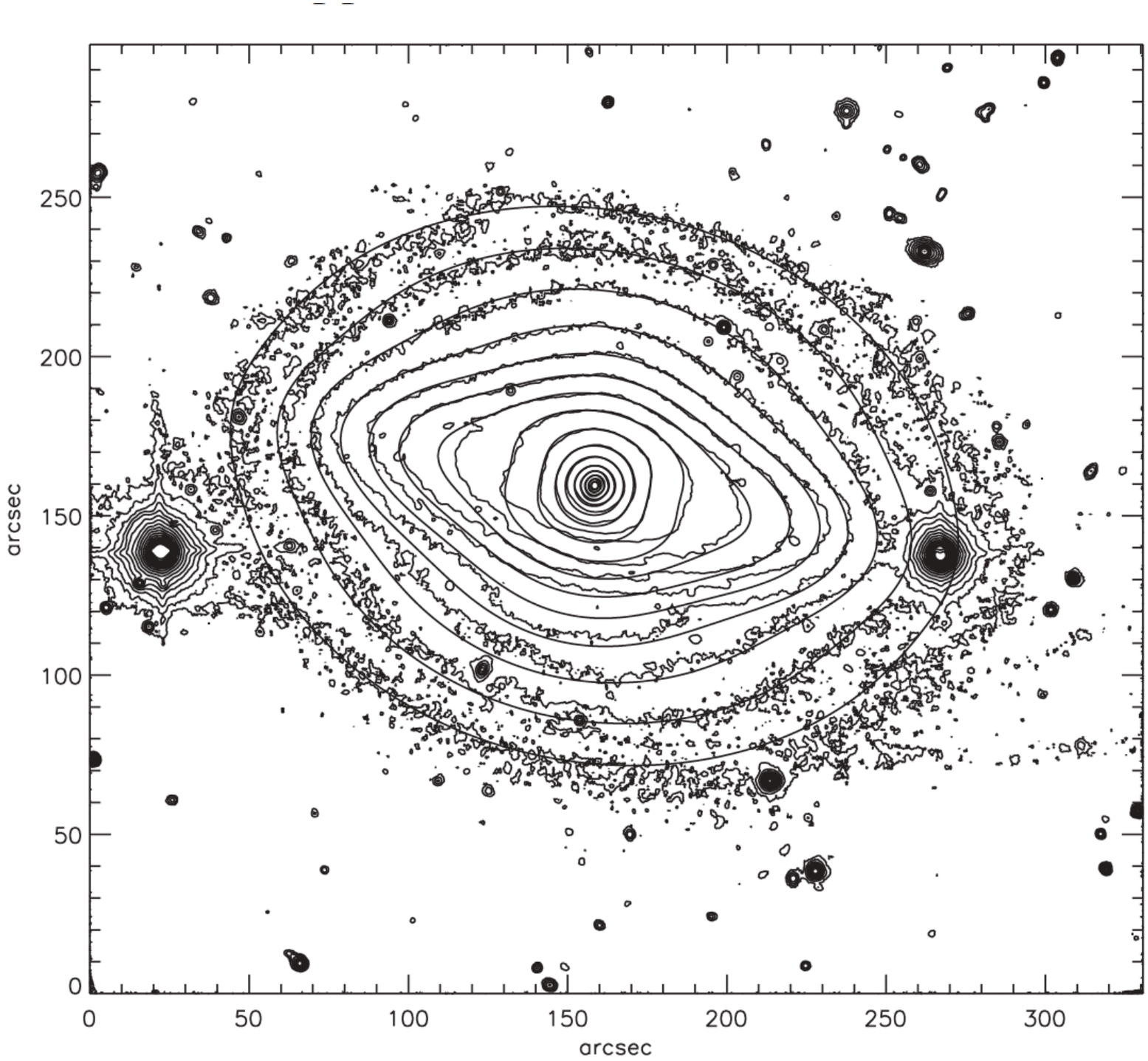
$$\ln L = \sum_{i=1}^{N} \ln \left[\sum_{j=1}^{M} \alpha_j \, N(x_i; \mu_j, \sigma_j) \right]$$

Assume now that $\mu_k = x_i$ for some i, k. In that case we have a term

$$p_k = \frac{1}{\sqrt{2\pi}\sigma_k}$$

which diverges for $\sigma_k \to 0$ - in which case that component takes all power.

Multi-gaussian expansion of galaxy images:



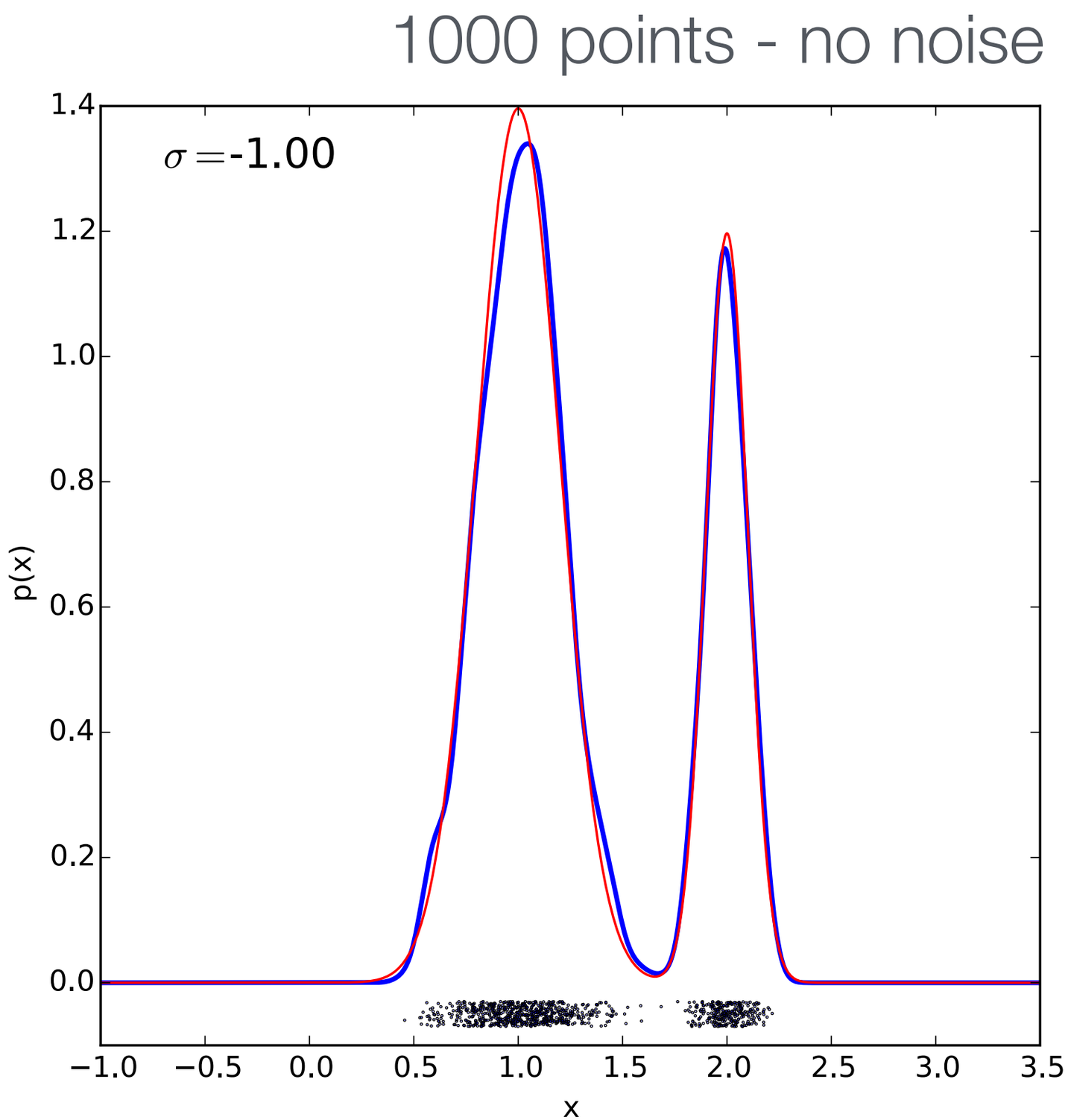
Cappellari (2002)

The effect of noise

Uncertainties in the data

True

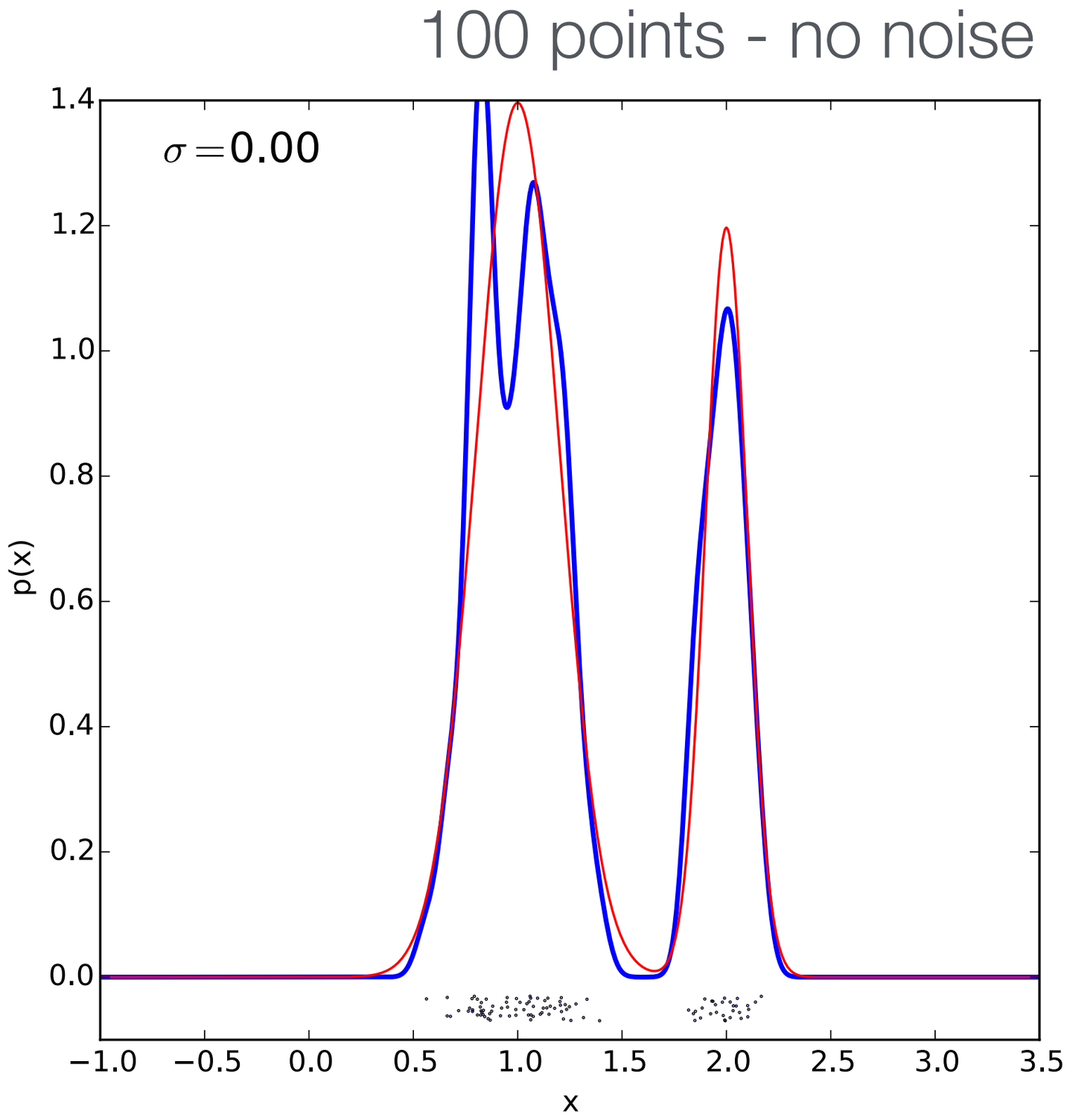
KDE



Uncertainties in the data

True

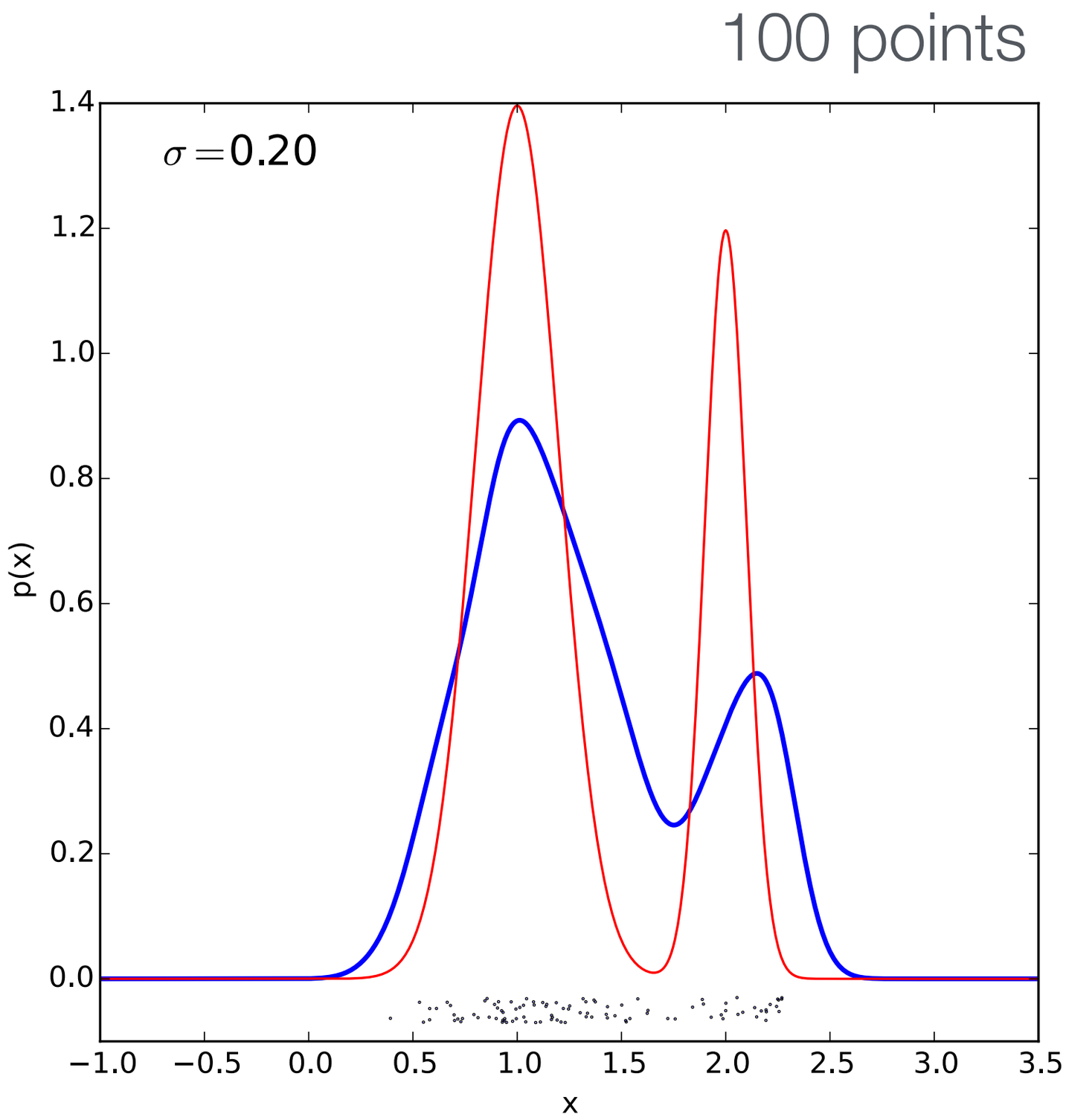
KDE



Uncertainties in the data

True

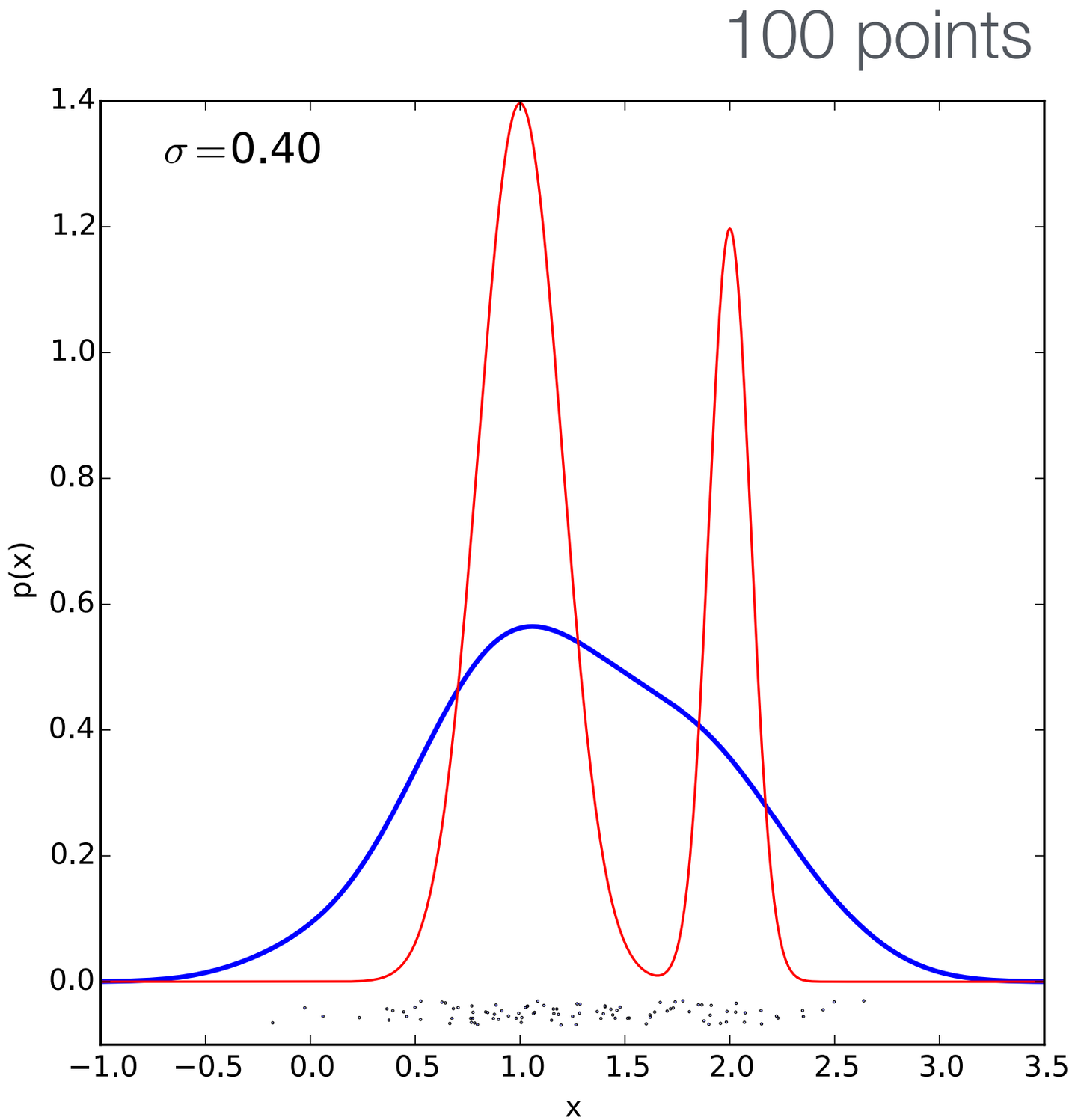
KDE



Uncertainties in the data

True

KDE



Uncertainties in the data - KDE

Your observations are (presumably) a random realisation of the underlying true value - this noise acts to broaden the recovered distribution. For equal uncertainties, σ, we have:

$$h_{\rm obs} = \left(h_{\rm true} \star g_\sigma\right)(x) = \int_{-\infty}^{\infty} h(u) g_\sigma(x-u)\, du$$

So can we recover a better estimate of the “true” distribution?

Uncertainties in the data - KDE

Your observations are (presumably) a random realisation of the underlying true value - this noise acts to broaden the recovered distribution. For equal uncertainties, σ, we have:

$$h_{\rm obs} = \left(h_{\rm true} \star g_\sigma\right)(x) = \int_{-\infty}^{\infty} h(u) g_\sigma(x-u)\, du$$

So can we recover a better estimate of the “true” distribution?

Yes: Deconvoluting KDE

(Delaigle & Meister (2008, Bernoulli, 14, 2, 562) - no Python implementation, but Matlab & R available at http://www.ms.unimelb.edu.au/~aurored/links.html#Code

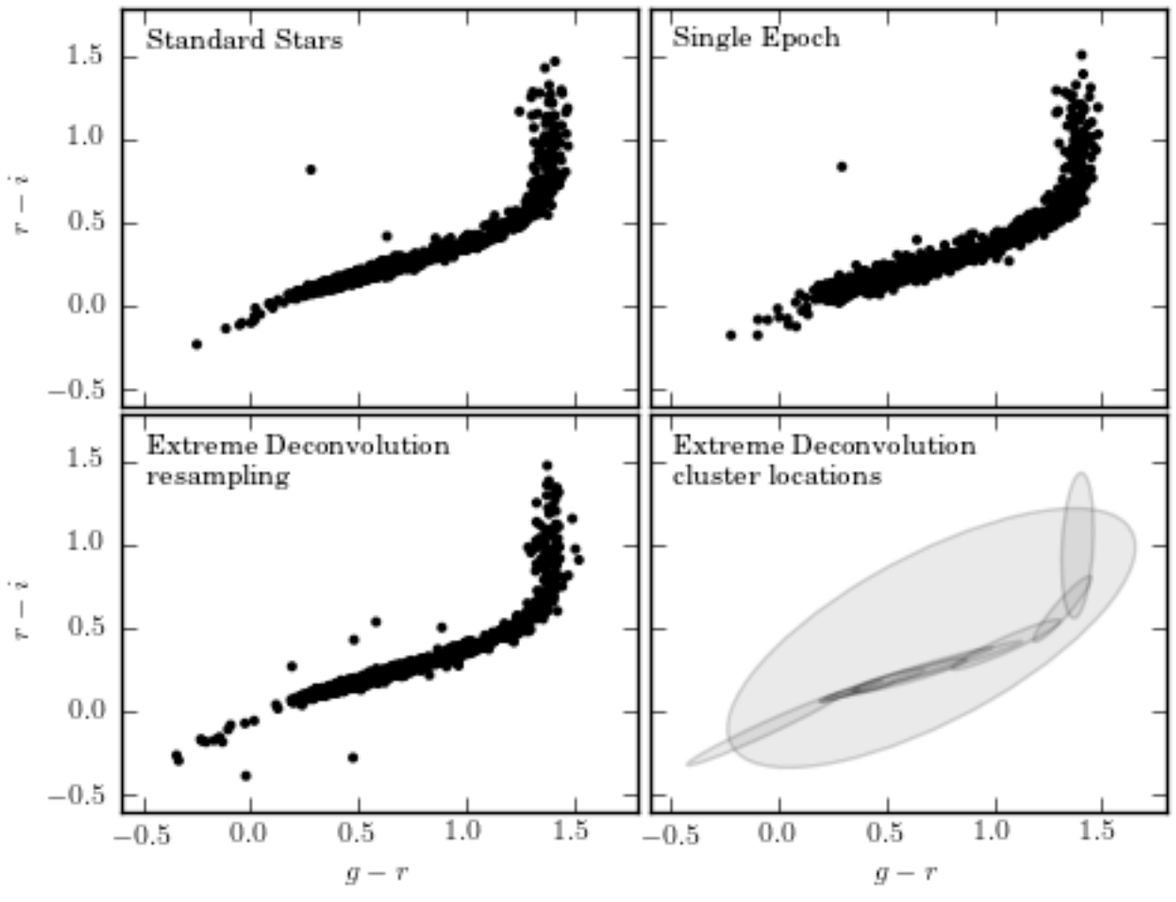
$$h_{\rm est} = \mathcal{F}^{-1}\left[\frac{\mathcal{F}(h_{\rm obs})}{\mathcal{F}(g)}\right]$$

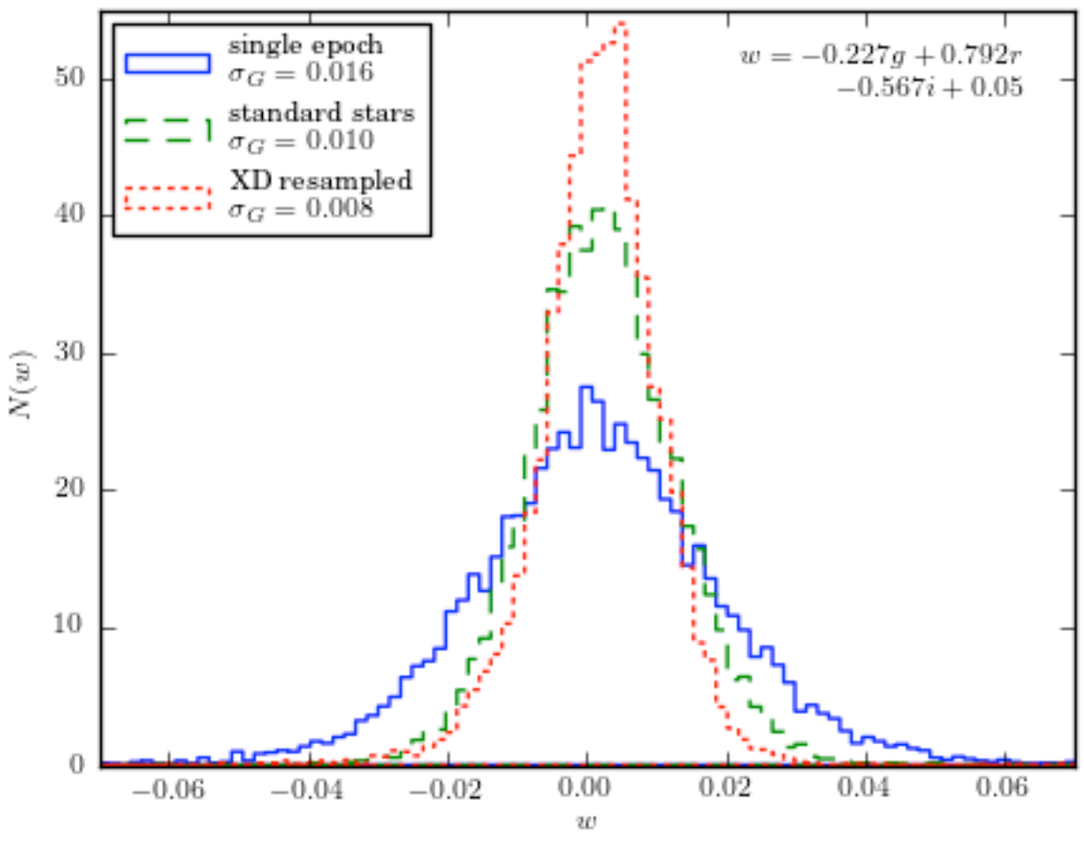
$\mathcal{F}$ being the Fourier transform

Uncertainties in the data - GMM

Extreme-deconvolution (Bovy, Hogg & Roweis 2011)

$$p(\boldsymbol{x}) + \sum_j \alpha_j N\left(\boldsymbol{x}|\mu_j, \boldsymbol{\Sigma}_j\right) \quad + \quad \boldsymbol{x}_i = \boldsymbol{R}_i \boldsymbol{v}_i + \epsilon_i$$





Uncertainties in the data - GMM

Trying it out (for after the lecture!):

```
from astroML.density_estimation import XDGMM

@pickle_results("XD_lgm-d4000.pkl")
def compute_XD_results(X, Xerr, n_components=10,
n_iter=500):
    clf = XDGMM(n_components, n_iter=n_iter)
    clf.fit(X, Xerr)
    return clf
```

X: N_{obs} x $N_{features}$

Xerr: N_{obs} x $N_{features}$ x $N_{features}$

Data - e.g. x & y

Covariances

Uncertainties in the data - GMM

Working with the result:

```
sample = clf.sample(1000)
```

Draw from the fitted distribution

```
clf.mu
```

Position of each component

```
clf.V
```

Covariance matrix of each component

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.

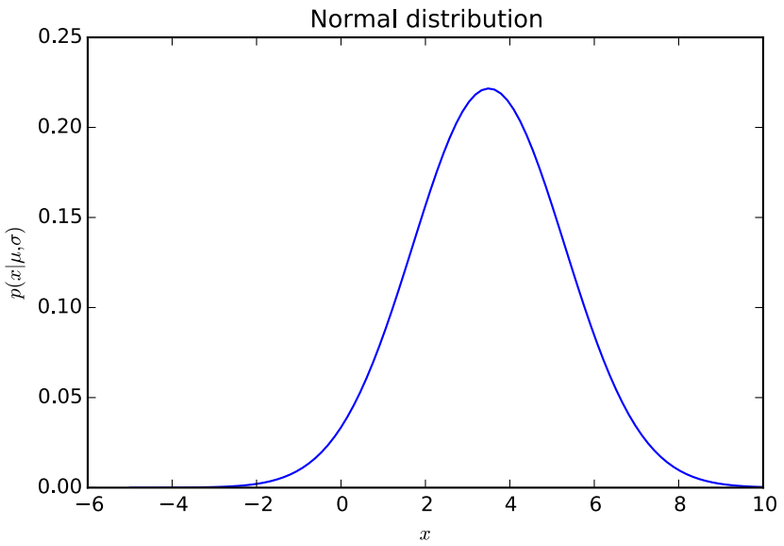
Simulation

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

Density estimation

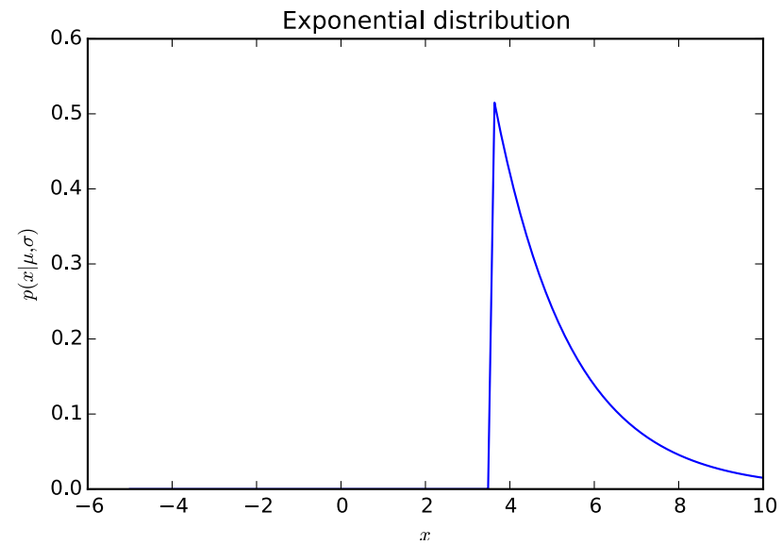
See GitHub for a IPython notebook on distributions. (Lecture 2/Notebooks)

Summarising distributions/data



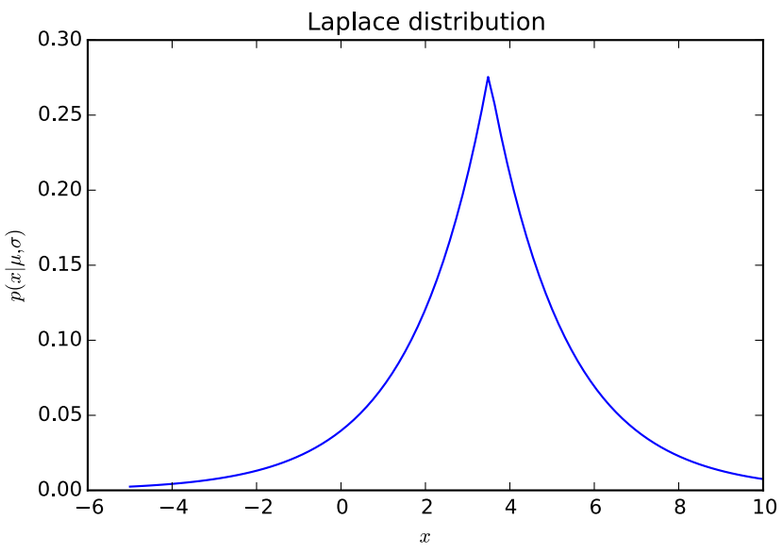
$$p(x|\mu,\sigma) = \frac{1}{\sqrt{2\pi}\sigma} e^{-(x-\mu)^2/2\sigma^2}$$

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



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

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



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

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

The Normal distribution

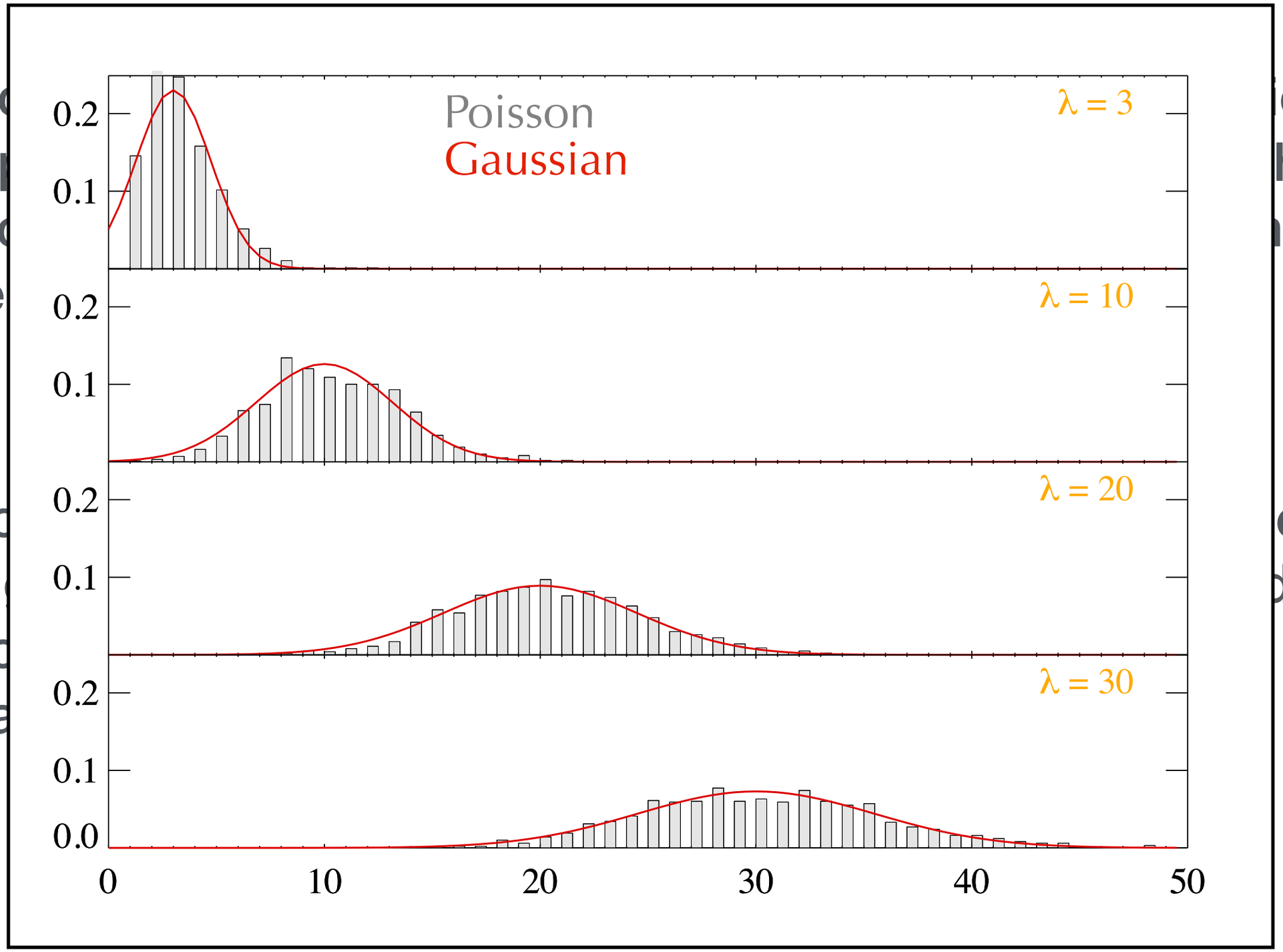
A particularly important distribution is the normal distribution. It is important in data mining because it is easy to deal with analytically and because the mean of large numbers of random numbers is distributed as a normal distribution.

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 partic…on.
It is imp…h
analytic…dom
numbe…

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



The Central Limit Theorem

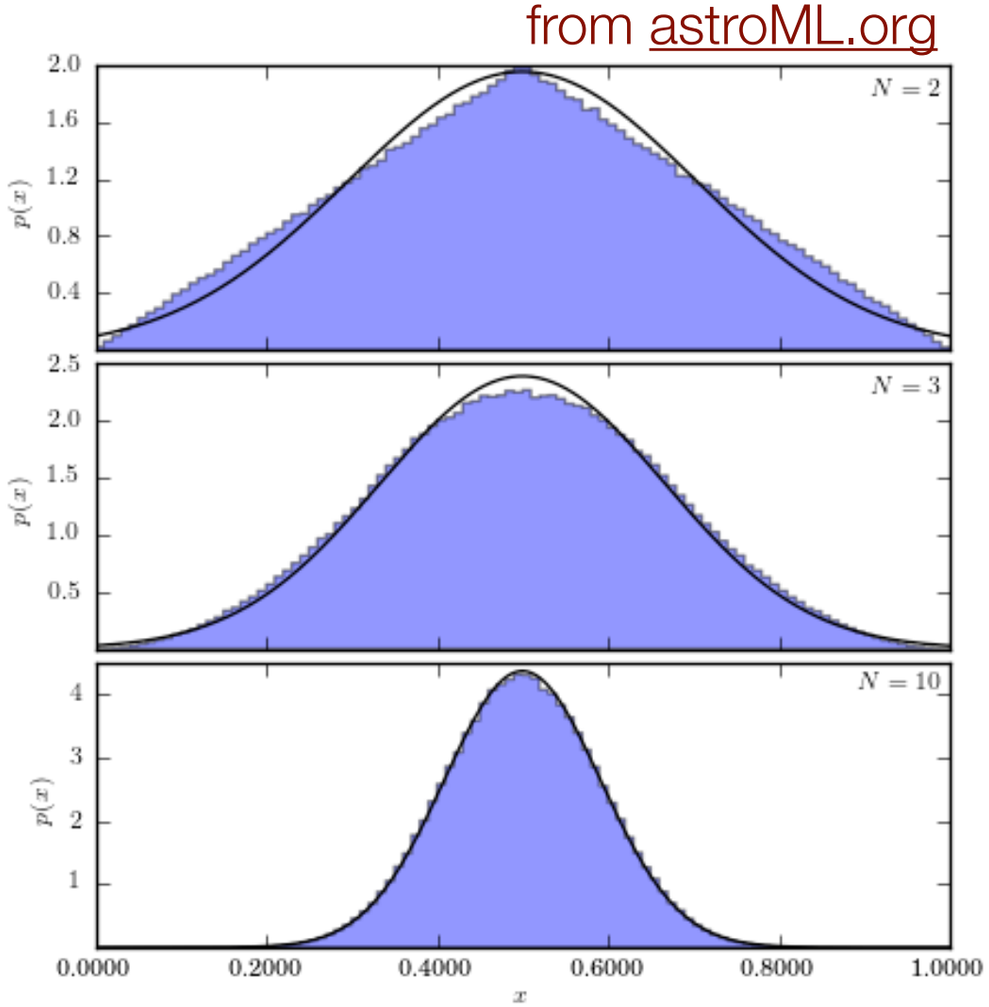
Arbitrary distribution $h(x)$ with mean, μ, and standard deviation σ

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, but we will typically want to store the results of calculations to disk.

Pickling

```
def pickle_to_file(data, fname):
    """A simple function to pickle
some data to a file"""
    try:
        fh = open(fname, 'wb')
        pickle.dump(data, fh)
        fh.close()
    except:
        print("Pickling failed!",
sys.exc_info()[0])
```

```
def pickle_from_file(fname):
    """The corresponding simple
routine to load from a file"""
    try:
        fh = open(fname, 'rb')
        data = pickle.load(fh)
        fh.close()
    except:
        print("Loading pickled data
failed!", sys.exc_info()[0])
        data = None
```

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 "l ({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 "l ({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 $M(a, b, \ldots)$

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 $M(\boldsymbol{\theta})$

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:

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

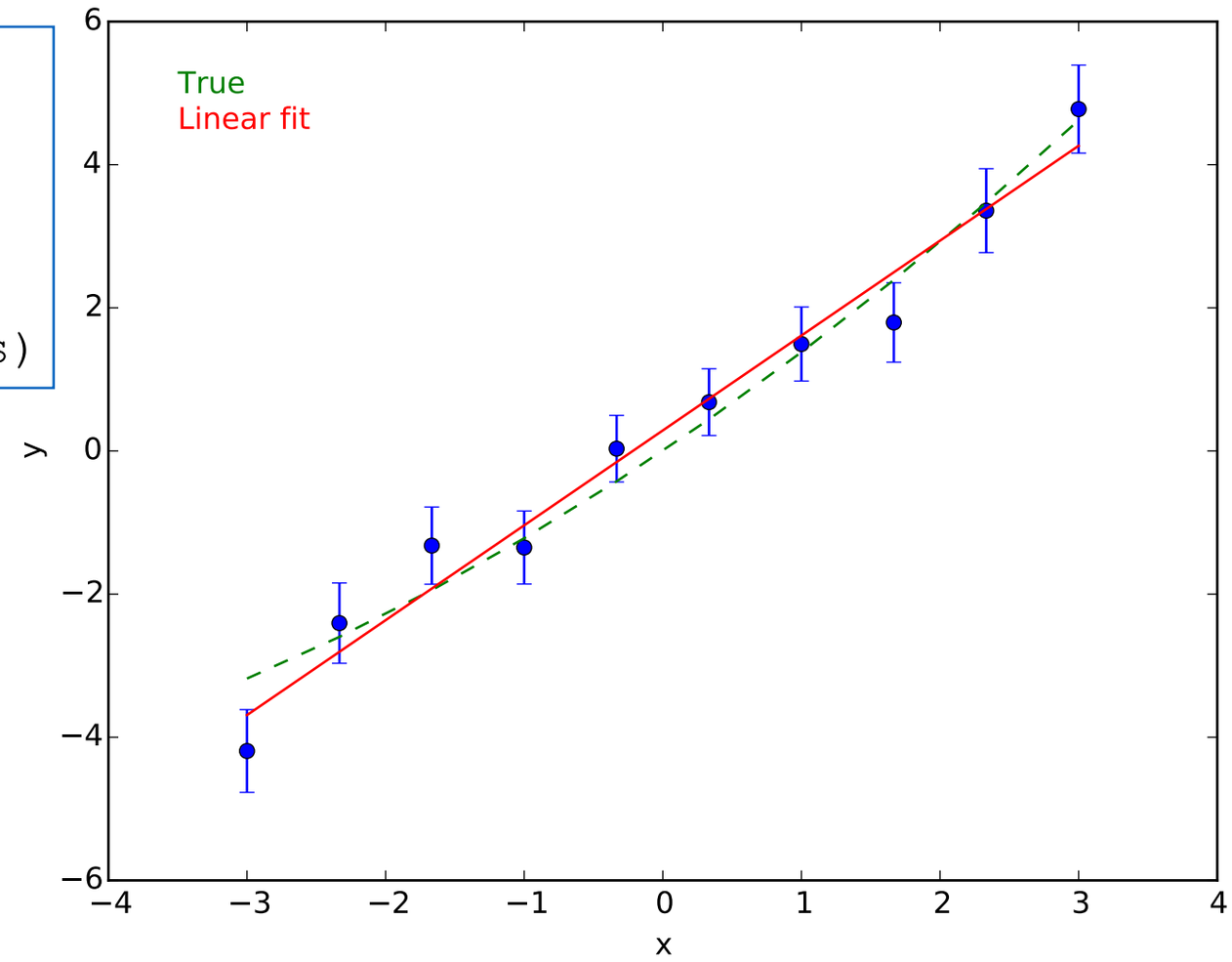
Full Likelihood $L(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{\sigma} | \alpha, \beta) = \prod_i P(x_i, y_i, \sigma_i | \alpha, \beta)$

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 neglnL(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(neglnL, [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 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
```

```
# 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.

It is installed on STRW computers - on your laptop 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
```

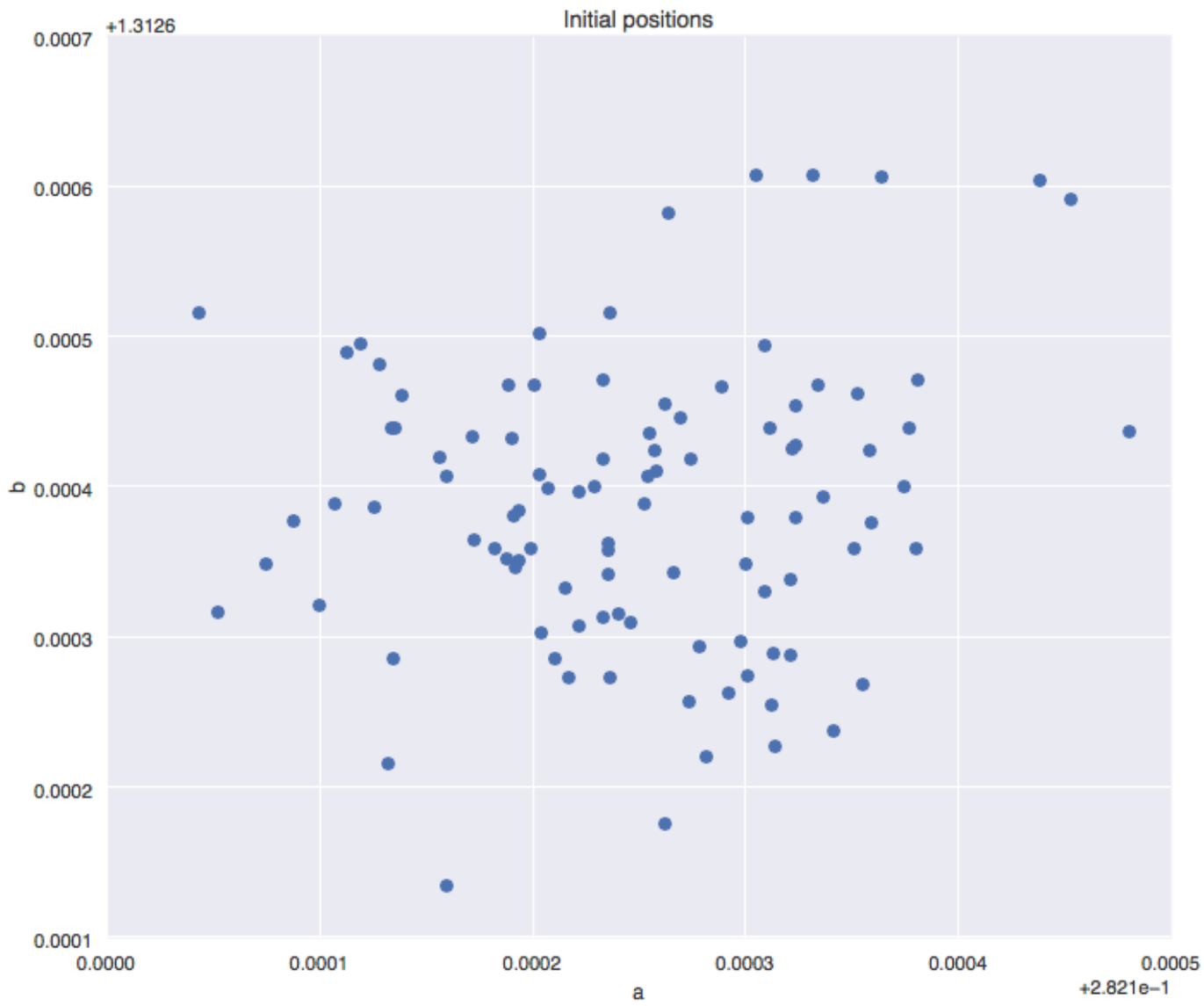
We have two parameters - so the dimensionality is 2

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.



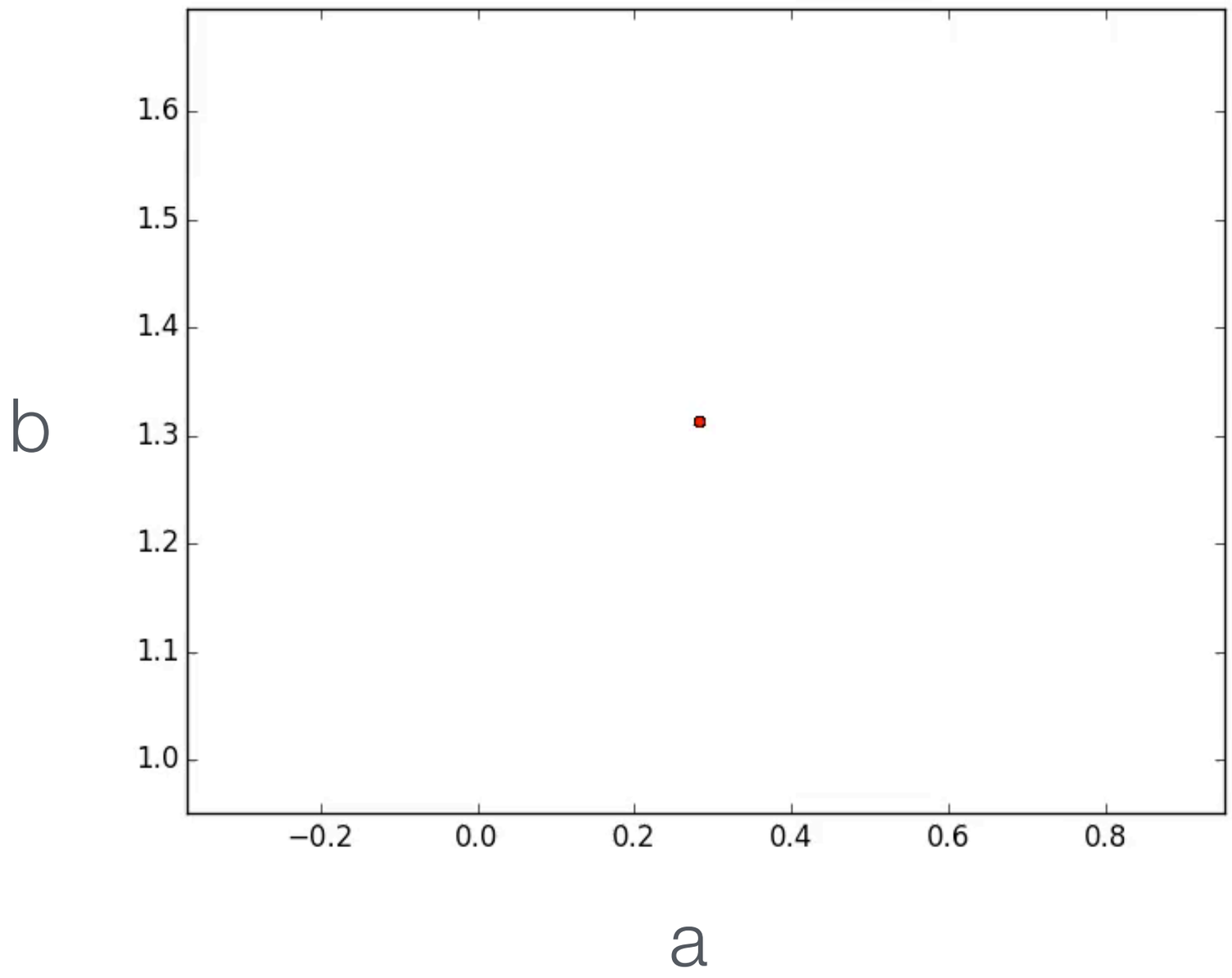
A practical example - line fitting

```
# 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.

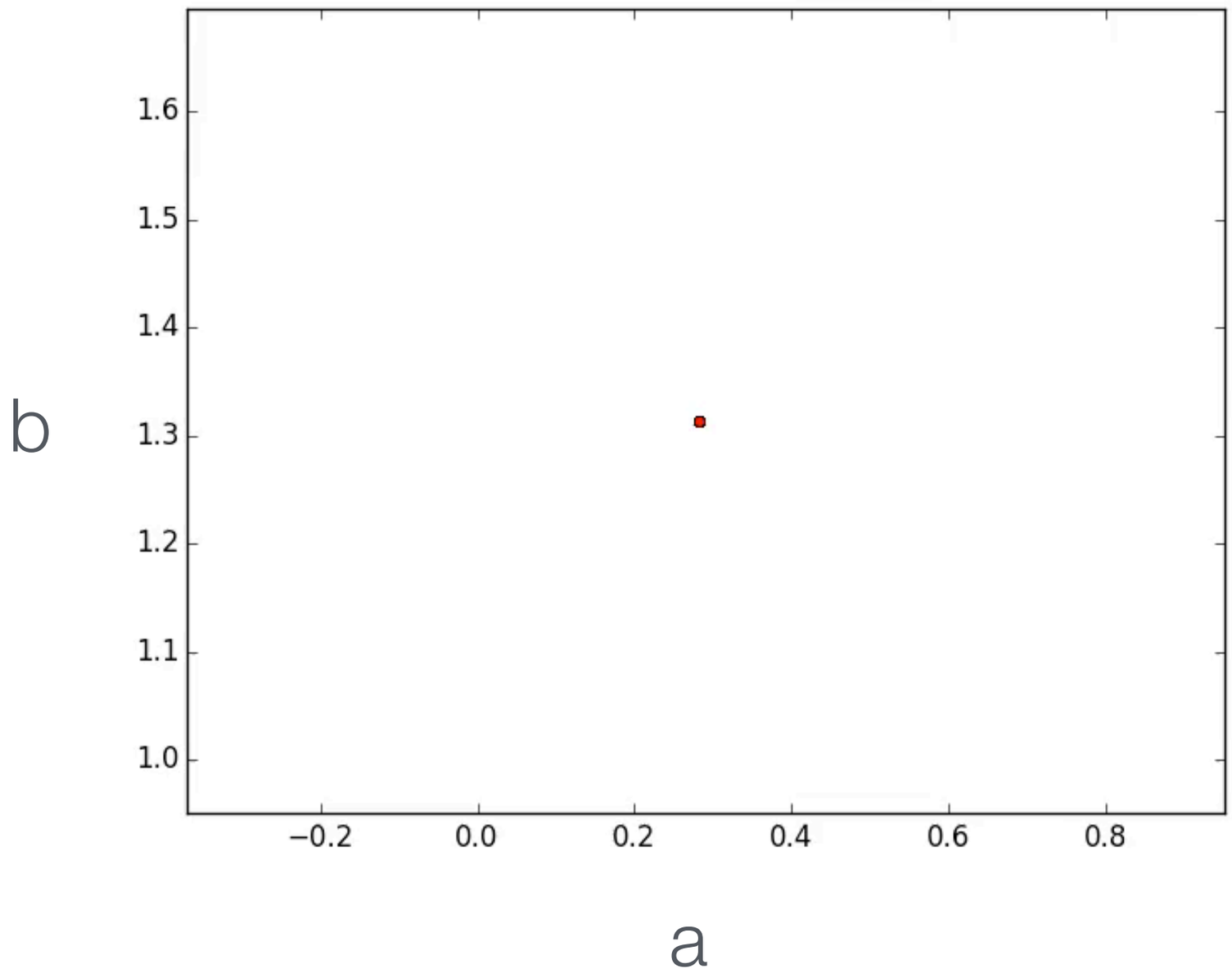
A practical example - line fitting

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



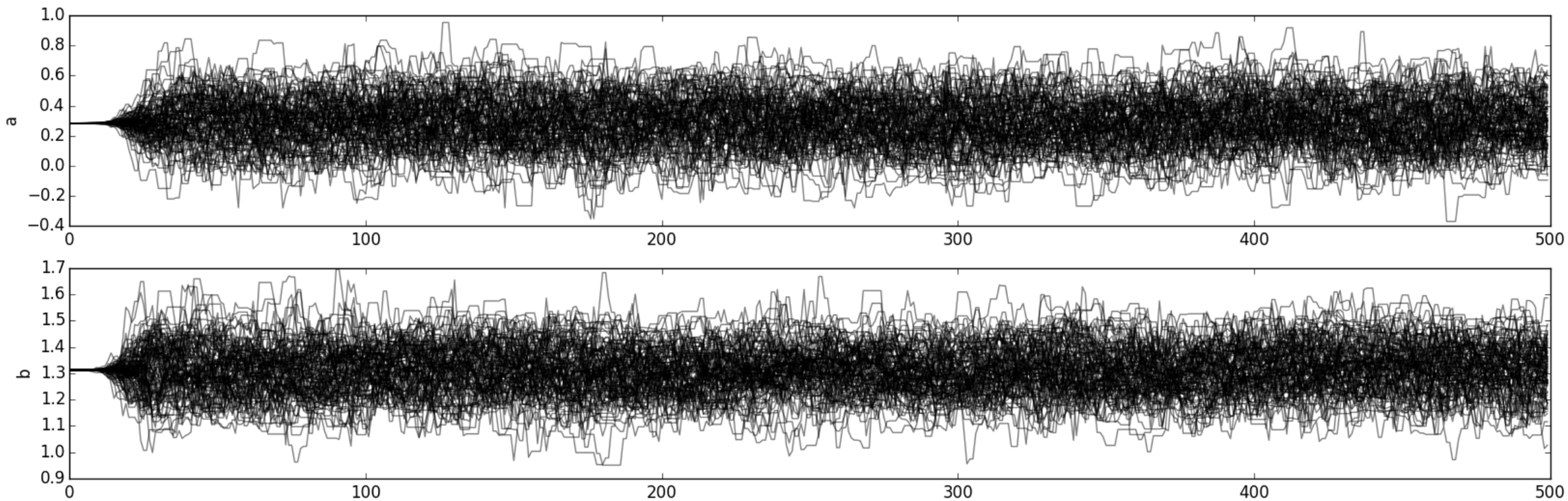
A practical example - line fitting

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



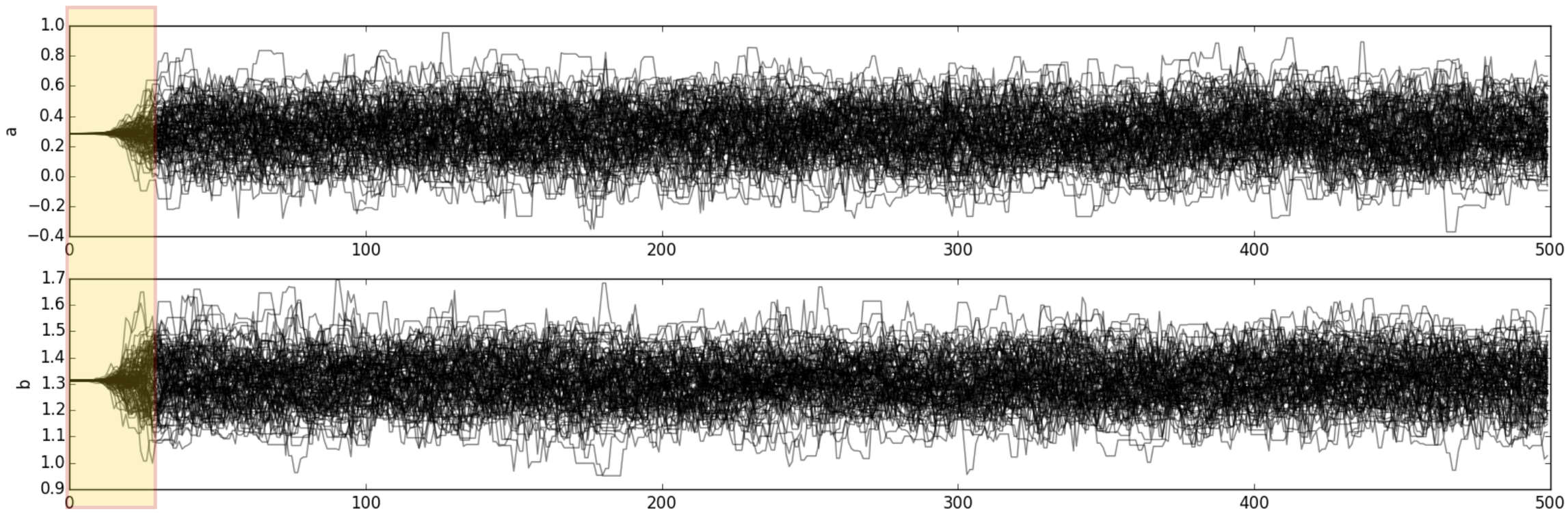
Look at chain behaviour

```
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)
```



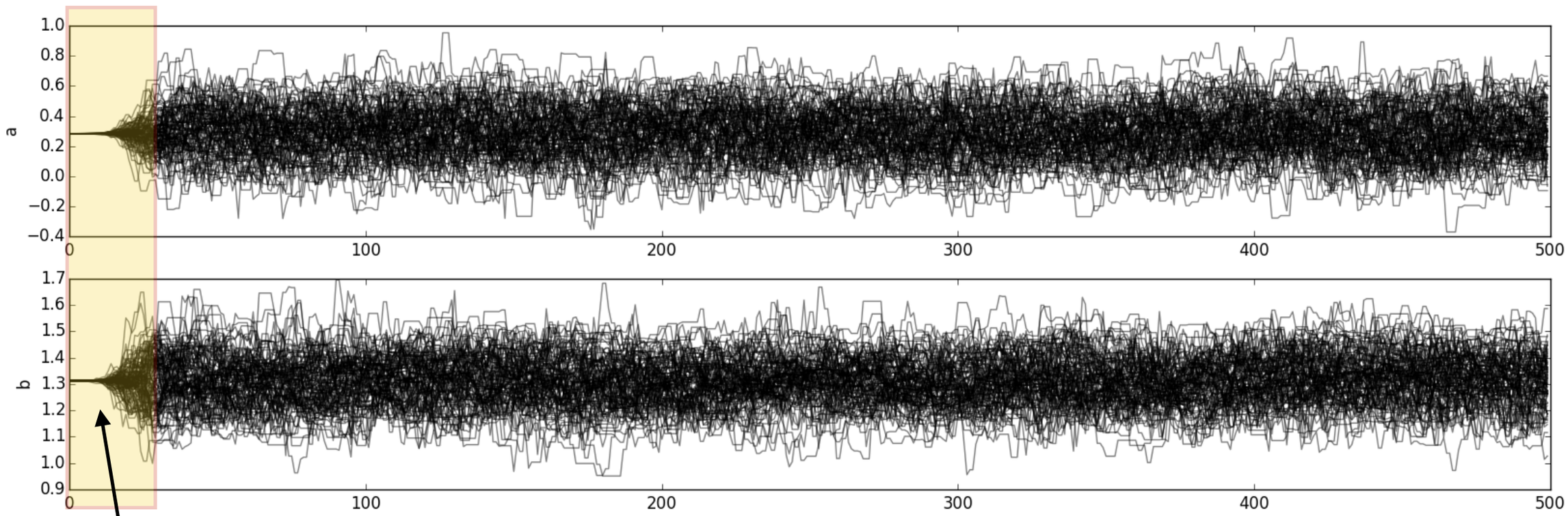
Look at chain behaviour

```
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

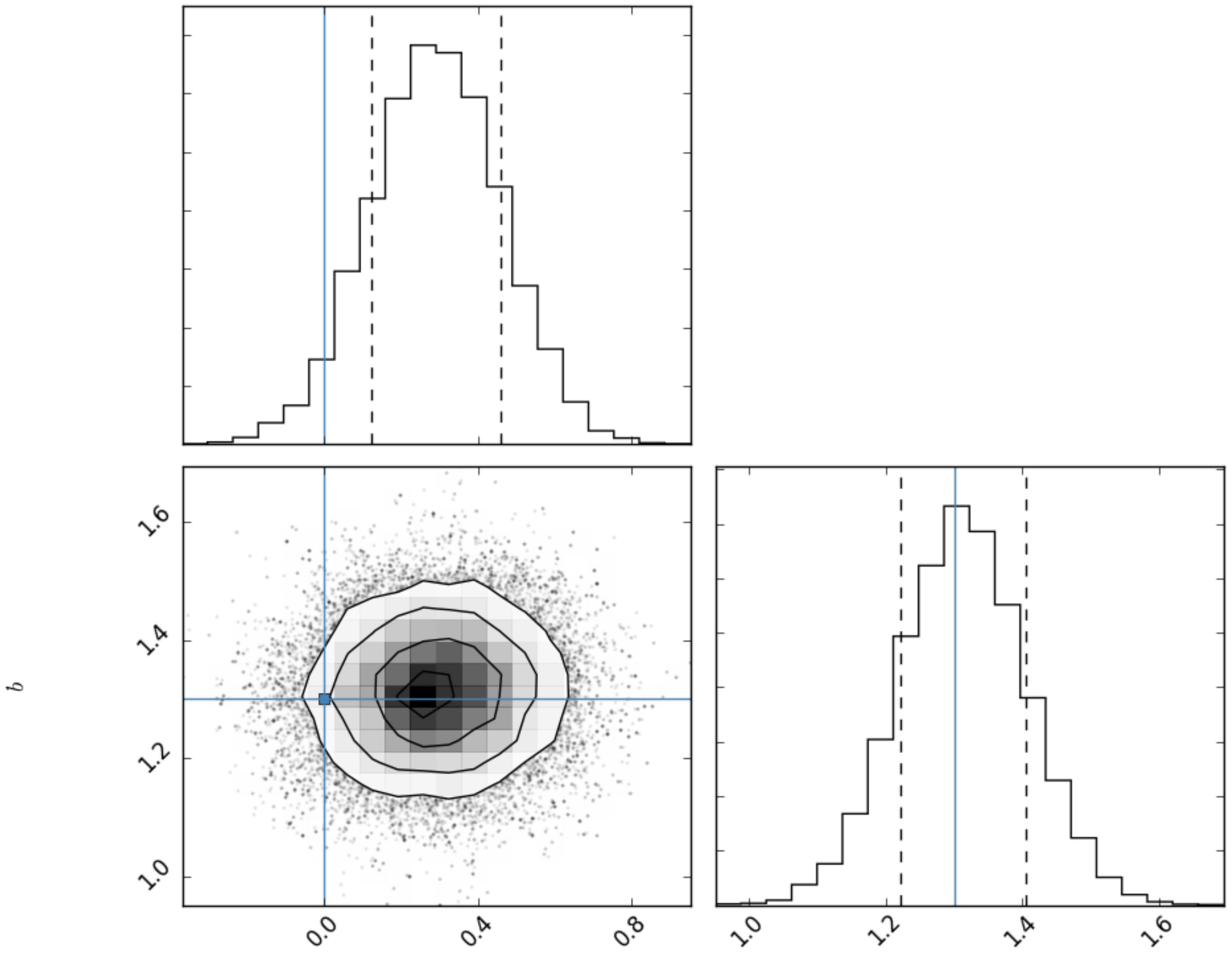
```
samples = sampler.chain[:, 50:, :].reshape((-1, 2))
```

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 practical example - showing the result

alkers



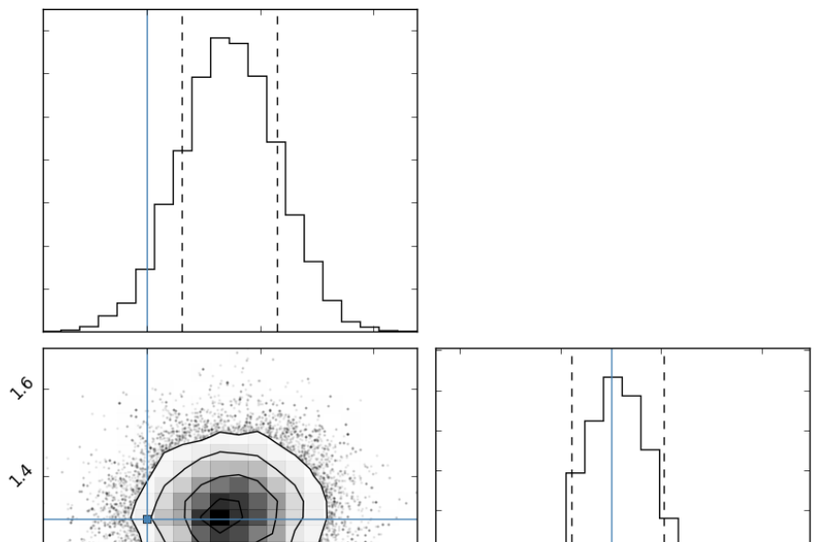
A practical example - showing the result

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()
```



Bayesian model selection

This is arguably the most rigorous but has its own problems.

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

What we want here is $p(M \,|\, D, I)$ so that we can compare against a different model. Thus we need to integrate out θ

This can be very challenging to impossible in multi-D situations.