Used to remove the toolbar:

In [1]:

```
%%javascript
$('#menubar').toggle();
```

```
<IPython.core.display.Javascript object>
```

Introduction to Data Science and Systems 2022-2023

Week 7: Probability II

- Probability theory, Bayes' Rule, discrete and continuous random variables

University of Glasgow - material prepared by John H. Williamson (adapted to IDSS by NP/BSJ), *v20222023a*

Summary

By the end of this unit you should know:

- the specific problems of continuous random variables as compared to discrete random variables
- the normal distribution and the central limit theorem
- what multivariate distributions are
- what Monte Carlo approaches are, and how expectation can be approximated
- what inference is
- what population, parameters, statistics and samples are
- how parameters can be estimated from a sample
- basic summary statistics: mean, standard deviation
- how to model simple data using normal distributions
- how estimators and maximum likelihood estimation work
- the maximum likelihood algorithm
- what Bayesian inference involves
- MCMC approaches to sampling posteriors in Bayesian inference
- the difference between the posterior and the predictive posterior

In [1]:

```
import IPython.display
IPython.display.HTML("""
<script>
  function code_toggle() {
    if (code_shown){
      $('div.input').hide('500');
      $('#toggleButton').val('Show Code')
    } else {
      $('div.input').show('500');
      $('#toggleButton').val('Hide Code')
    }
    code_shown = !code_shown
  }

  $( document ).ready(function(){
    code_shown=false;
    $('div.input').hide()
  });
</script>
<form action="javascript:code_toggle()"><input type="submit" id="toggleButton" value
```

Out[1]:

Hide Code

In [2]:

```
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt
from jhwutils.float_inspector import print_shape_html, print_float, print_float_html
from jhwutils.matrices import show_boxed_tensor_latex, print_matrix
import jhwutils.image_audio as ia
import numpy as np
%matplotlib inline
plt.rc('figure', figsize=(8.0, 4.0), dpi=140, frameon=False)
```

Continuous random variables

Problems with continuous variables

Continuous random variables are defined by a PDF (probability *density* function), rather than a PMF (probability *mass* function). A PMF essentially just a vector of values, but a PDF is a function mapping *any* input in its domain to a probability. This seems like a subtle difference (as a PMF has more and more "bins" it gets closer and closer to a PDF, right?), but it has a number of complexities.

- The probability of any specific value is $P(X = x) = 0$ *zero* for every possible x, yet any value in the *support* of the distribution function (everywhere the PDF is non-zero) is possible.
- There is no direct way to sample from the PDF in the same way as we did for the PMF. But there are several tricks for sampling from continuous distributions.

- We cannot estimate the true PDF from simple counts of observations like we did for the empirical distribution. This can never "fill in" the PDF, because it will only apply to a single value with zero "width".
- Bayes' Rule is easy to apply to discrete problems. But how do we do computations with continuous PDFs using Bayes' Rule?
- Simple discrete distributions don't have a concept of dimension. But we can have continuous values in $\mathbb{R}$, or in vector spaces $\mathbb{R}^n$, representing the probability of a random variable taking on a vector value, or indeed distributions over other generalisations (matrices, other fields like complex numbers or quaternions and even more sophisticated structures like Riemannian manifolds).

Probability distribution functions ¶

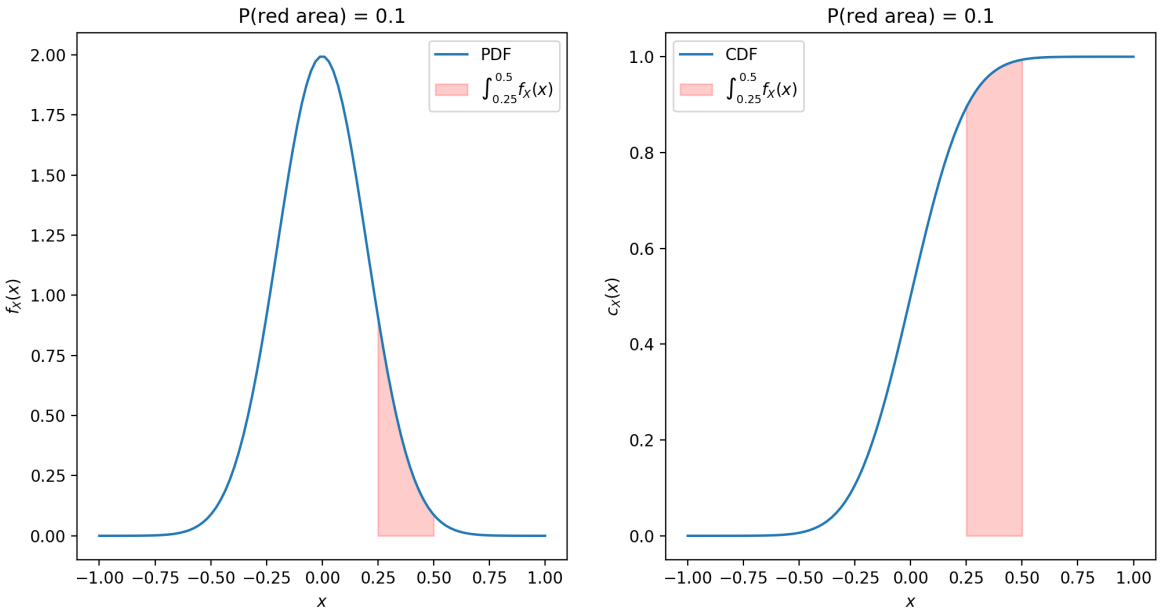
The PDF $f_X(x)$ of a random variable X maps a value x (which might be a real number, or a vector, or any other continuous value) to a single number, the density at the point. It is a function (assuming a distribution over real vectors) $\mathbb{R}^n \to \mathbb{R}^+$, where $\mathbb{R}^+$ is the positive real numbers, and

$$\int_x f_X(x) = 1.$$

- While a PMF can have outcomes with a probability of at most 1, it is *not* the case that the maximum value of a PDF is $f_X(x) \leq 1$ -- *just that the integral of the PDF be 1.*

The value of the PDF at any point is **not** a probability, because the probability of a continuous random variable taking on any specific number must be zero. Instead, we can say that the probability of a continuous random variable X lying in a range (a, b) is:

$$P(X \in (a, b)) = (a < X < b) = \int_a^b f_X(x)$$



Support

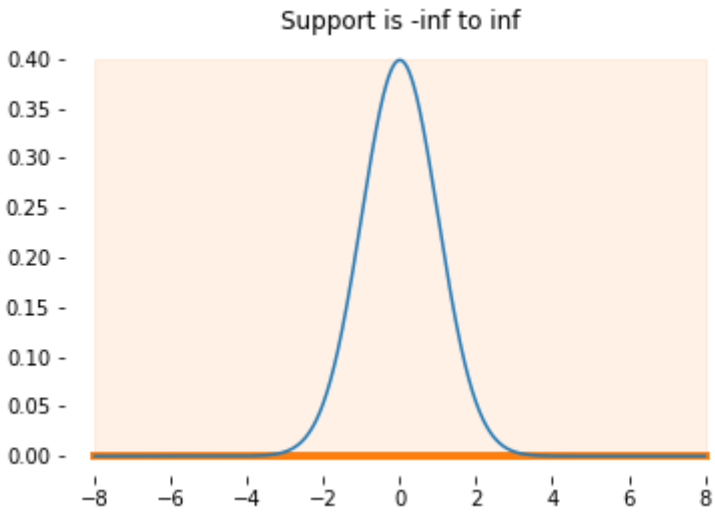
The **support** of a PDF is the domain it maps from where the density is non-zero.

$$\text{supp}(x) = x \text{ such that } f_X(x) > 0$$

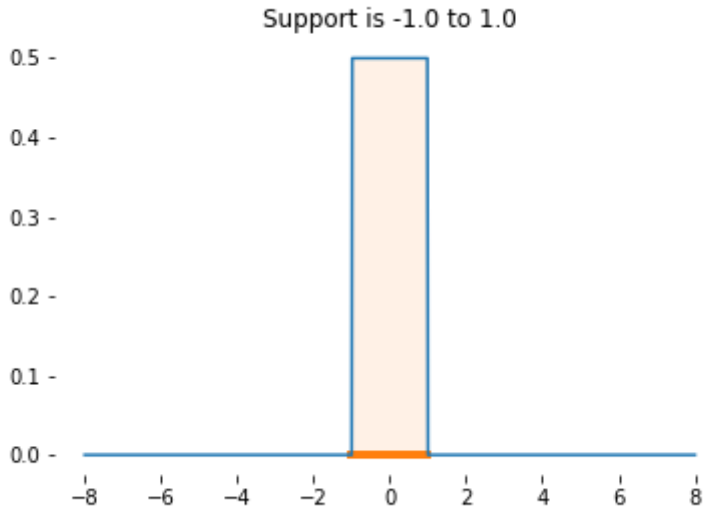
Some PDFs have density over a fixed interval, and have zero density everywhere else. This is true of the uniform distribution, which has a fixed range where the density is constant, and is zero everywhere. This is called **compact support**. We know that sampling from a random variable with this PDF will always give us values in the range of this support. Some PDFs have non-zero density over an infinite domain. This is true of the normal distribution. A sample from a normal distribution could take on *any* real value at all; it's just much more likely to be close to the mean of the distribution than to be far away. This is **infinite support**.

Examples

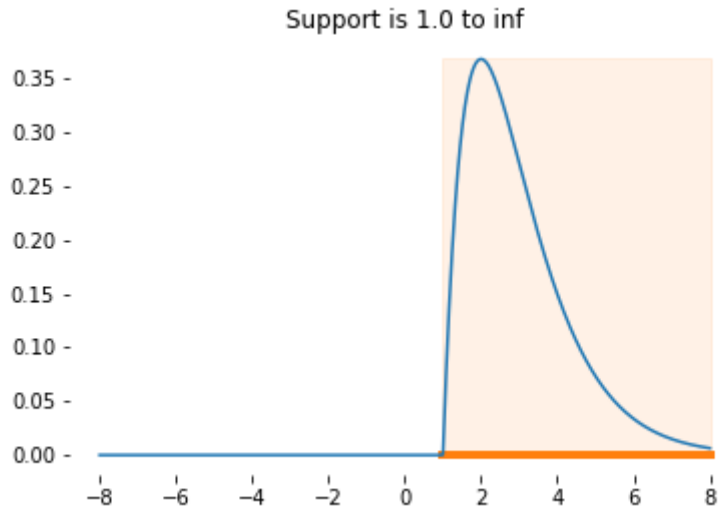
infinite support



compact support



semi-infinite support



Cumulative distribution functions

The **cumulative distribution function** or CDF of a real-valued random variable is

$$F_X(x) = \int_{-\infty}^{x} f_X(x) = P(X \leq x)$$

Unlike the PDF, the CDF always maps x to the codomain [0,1]. For any given value $F_X(x)$ tells us how much probability mass there is that is less than or equal to x. Given a CDF, we can now answer questions, like: what is the probability that random variable X takes on a value between 3.0 and 4.0?

$$P(3 \leq X \leq 4) = F_X(4) - F_X(3)$$

This *is* a probability. Sometimes it is more convenient or efficient to do computations with the CDF than with the PDF.

PDF example: the normal disribution

The most ubiquitous of all continuous PDFs is the **normal** or **Gaussian** distribution. It assigns probabilities to real values $x \in \mathbb{R}$ (in other words, a sample space consisting of all of the real numbers). It has a density given by the PDF:

$$f_X(x) = \frac{1}{Z} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

You do not need to remember this formula, but it is very useful to know that it is essentially just e^{-x^2} with some scaling factors to normalise it -- this is called the **squared exponential function**.

Side note:

$$Z = \sqrt{2\pi\sigma^2}$$

but you do not need to remember this.

We use a shorthand notation to refer to the distribution of continuous random variables,

$$\text{variable} \sim \text{distribution}(\text{parameters}),$$

where $\sim$ is read as "distributed as". For a normal distribution this is:

$$X \sim \mathcal{N}(\mu, \sigma^2),$$

which is read as

> "Random variable X is distributed as [N]ormal with mean μ and variance σ^2"

and means that X has a density function defined by the class of normal density functions with a specific choice of parameters μ and σ^2; $f_X(x) = \frac{1}{Z}\, e^{-\frac{(x-\mu)^2}{2\sigma^2}}$.

There are various other symbols used for other continuous distributions, including $\Gamma(\alpha, \beta), \beta(\alpha, \beta), t(\nu), \chi^2(k), \ldots$, which we will not cover in this course. See the references if you want to learn more.

Location and scale

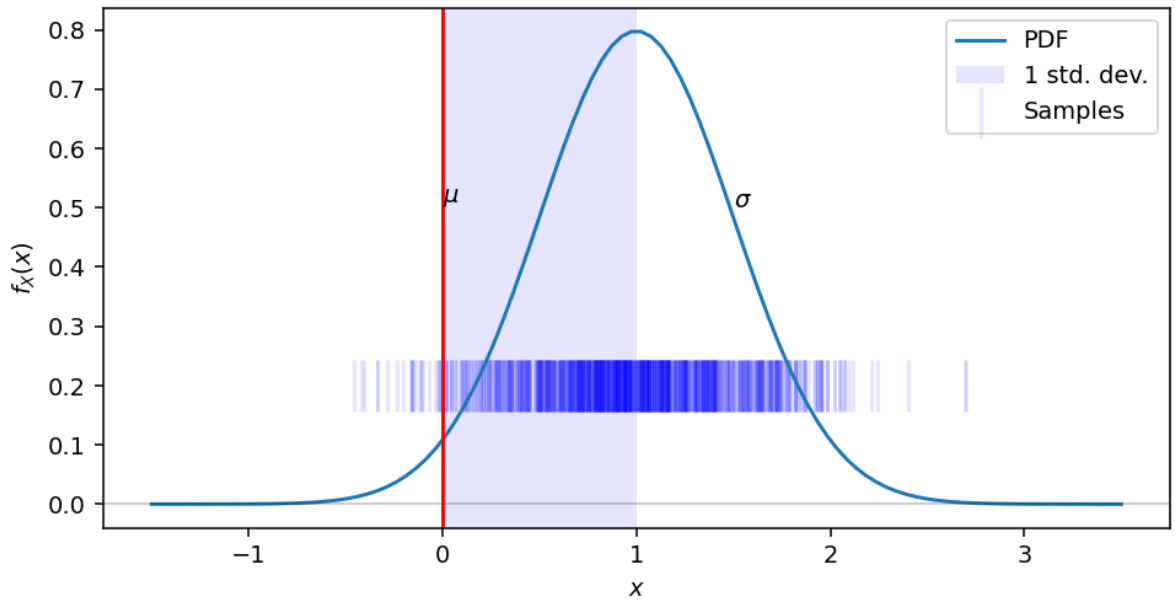
The normal distribution places the point of highest density at to its center μ (the "mean"), with a spread defined by σ^2 (the "variance"). This can be though of the **location** and **scale** of the density function. Most standard continuous random variable PDFs have a location (where density is concentrated) and scale (how spread out the density is).

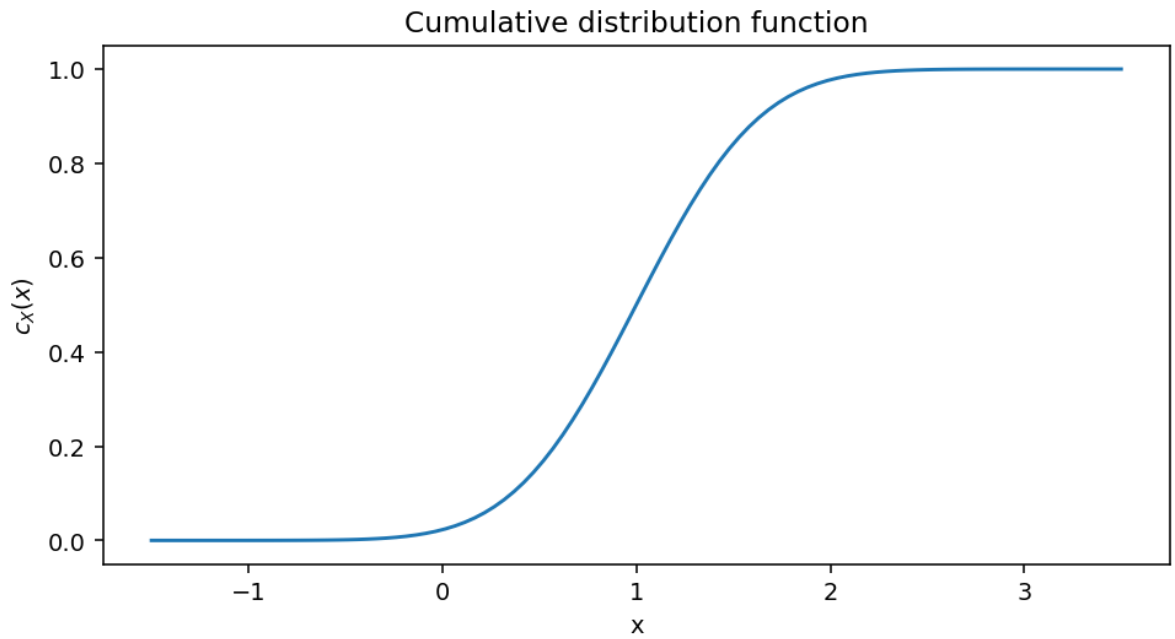
In [4]:

```
import scipy.stats as stats
# Plot the PDF of the normal distribution
def plot_normal(mu, sigma):
    # plot the normal (Gaussian distibution) along with a set of points drawn from t
    x = np.linspace(mu-sigma*5,mu+sigma*5,100)
    y = stats.norm.pdf(x,mu,sigma) # mean 0, std. dev. 1

    fig = plt.figure()
    ax = fig.add_subplot(1,1,1)

    ax.plot(x,y, label="PDF")
    ax.axhline(0, color='k', linewidth=0.2) # axis line

    # mark the mean
    ax.text(0, 0.51, '$\mu$')
    ax.axvline(0, color='r')
    # highlight one std. dev. to the right
    ax.axvspan(0,1, facecolor='b', alpha=0.1, label="1 std. dev.")
    ax.text(mu+sigma, 0.5, '$\sigma$')
    # take 1000 random samples and scatter plot them
    samples = stats.norm.rvs(mu, sigma,1000)
    ax.scatter(samples, np.full(samples.shape, .2), s=448, c='b', alpha=0.1, marker=
    ax.set_xlabel("$x$")
    ax.set_ylabel("$f_X(x)$")
    ax.legend()

    fig = plt.figure()
    ax = fig.add_subplot(1,1,1)
    ax.plot(x, stats.norm.cdf(x,mu,sigma))
    ax.set_title("Cumulative distribution function")
    ax.set_xlabel("x")
    ax.set_ylabel("$c_X(x)$")
```

In [5]:

```
plot_normal(mu=1.0, sigma=0.5)
```





Normal modelling

It seems that this might be a very limiting choice but there are two good reasons for this to work well as a model in many contexts:

1. Normal variables have very nice mathematical properties and are easy to work with analyitically (i.e. without relying on numerical computation).
2. The *central limit theorem* tells us that any sum of random variables (however they are distributed) will tend to a *Levy stable distribution* as the number of variables being summed increases. For most random variables encountered, this means the normal distribution (one specific Levy stable distribution).

Central limit theorem

If we form a sum of many random variables

$$Y = X_1 + X_2 + X_3 + \dots,$$

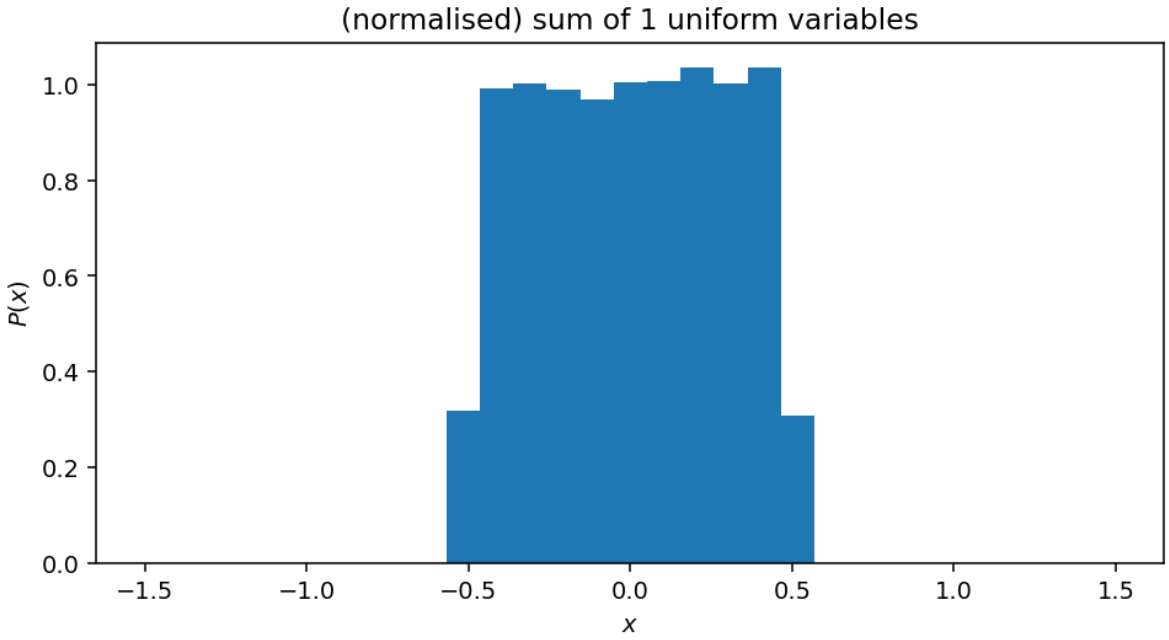
then for almost any PDF that each of $X_1, X_2, \dots$ might have, the PDF of Y will be approximately normal, $Y \sim \mathcal{N}(\mu, \sigma)$. This means that any process that involves a mixture of many random components will tend to be Gaussian under a wide variety of conditions.

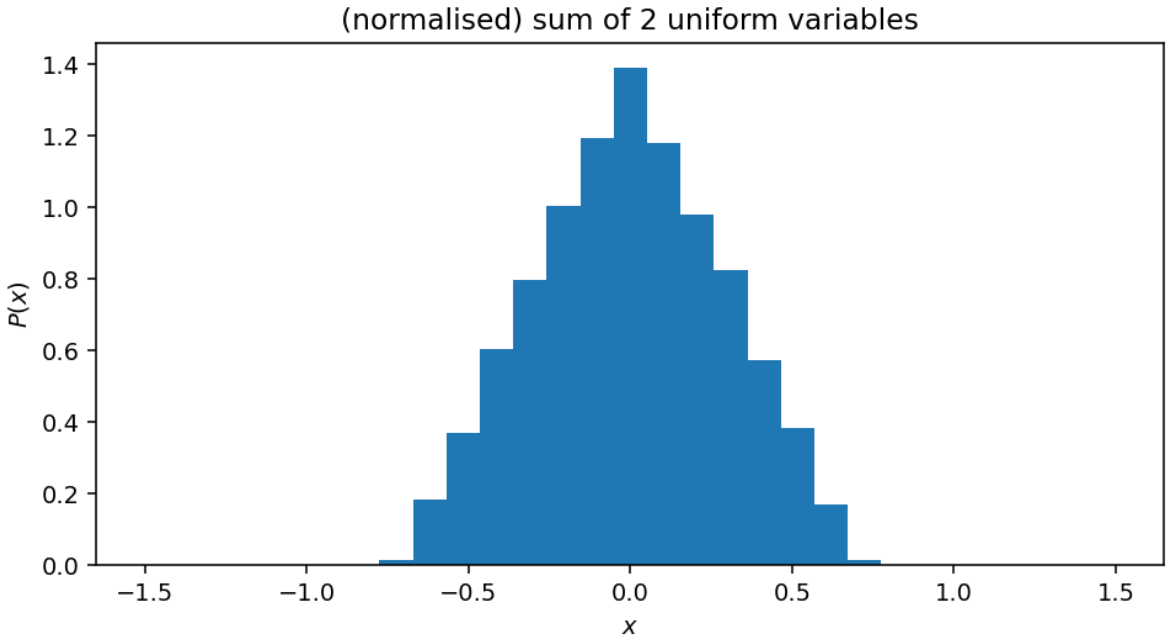
In [6]:

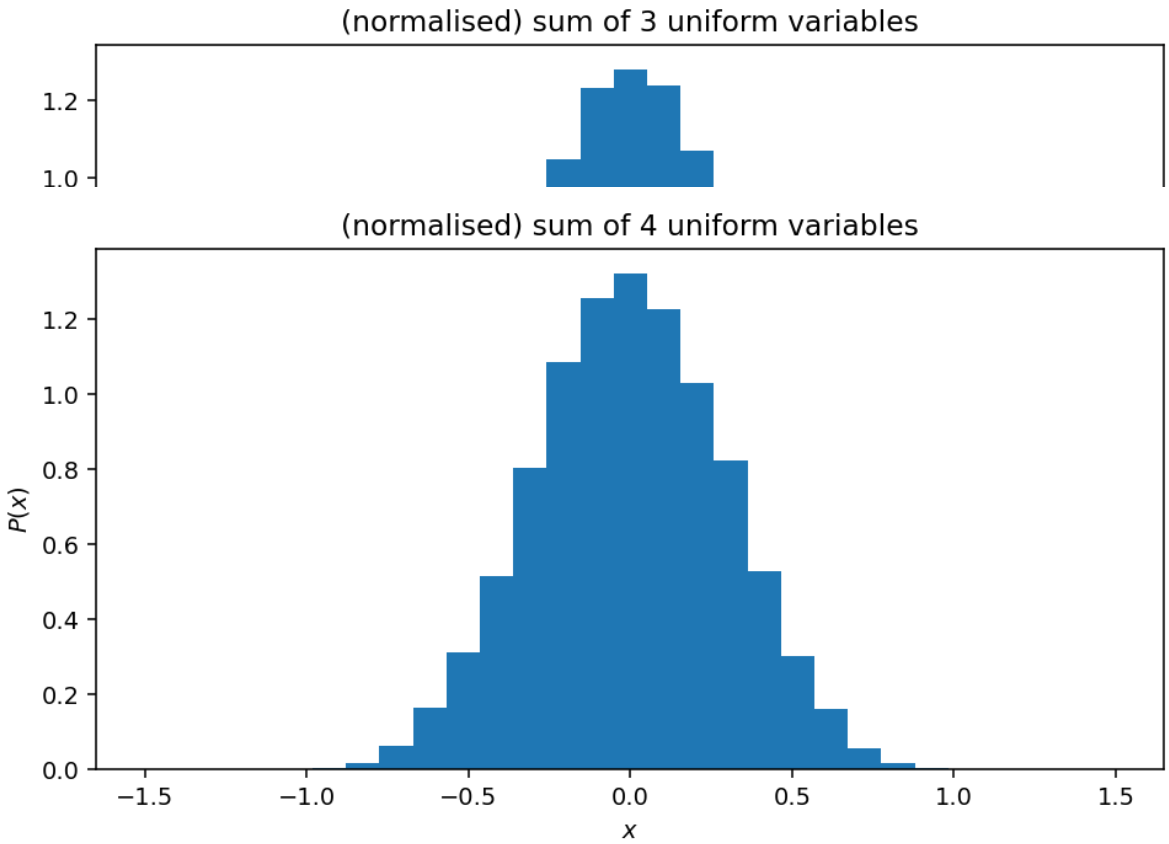
```
def clt():
    # demonstrate the central limit theorem
    for i in [1,2,3,4,8,16,32]:
        x = np.zeros((20000,))
        # add i copies of samples drawn from uniform (flat) distribution together
        for j in range(i):
            x += np.random.uniform(-0.5,0.5, x.shape)
        fig = plt.figure()
        ax = fig.add_subplot(1,1,1)
        ax.hist(x/np.sqrt(i), bins=np.linspace(-1.5,1.5,30), density=True)
        ax.set_title("(normalised) sum of %d uniform variables" % (i))
        ax.set_xlabel("$x$")
        ax.set_ylabel("$P(x)$")
```

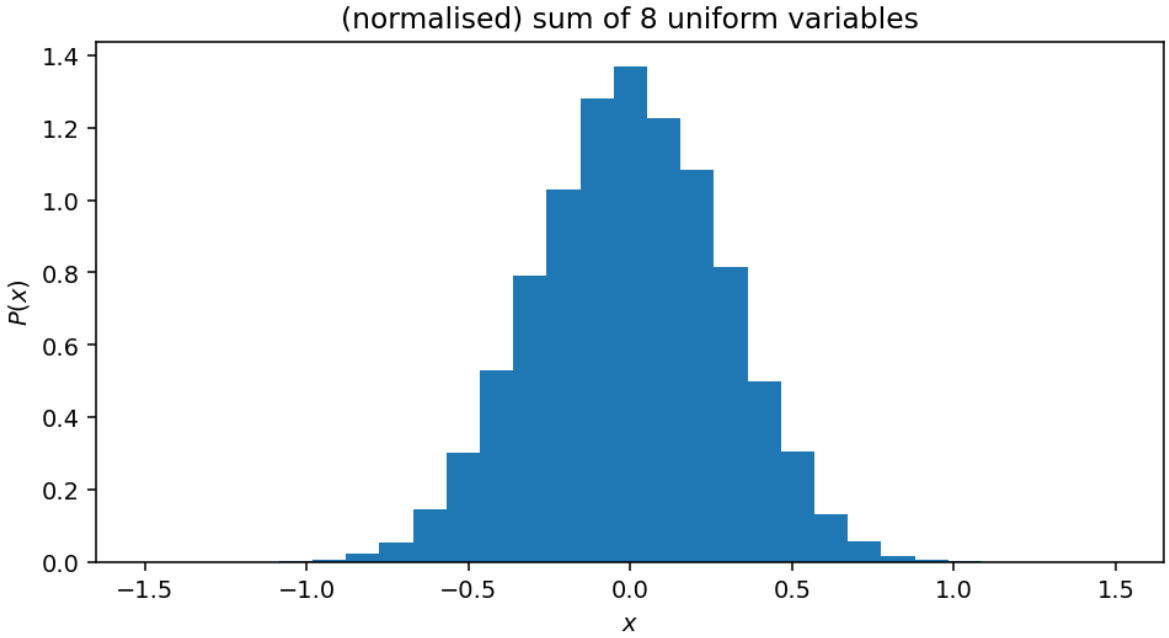
In [7]:

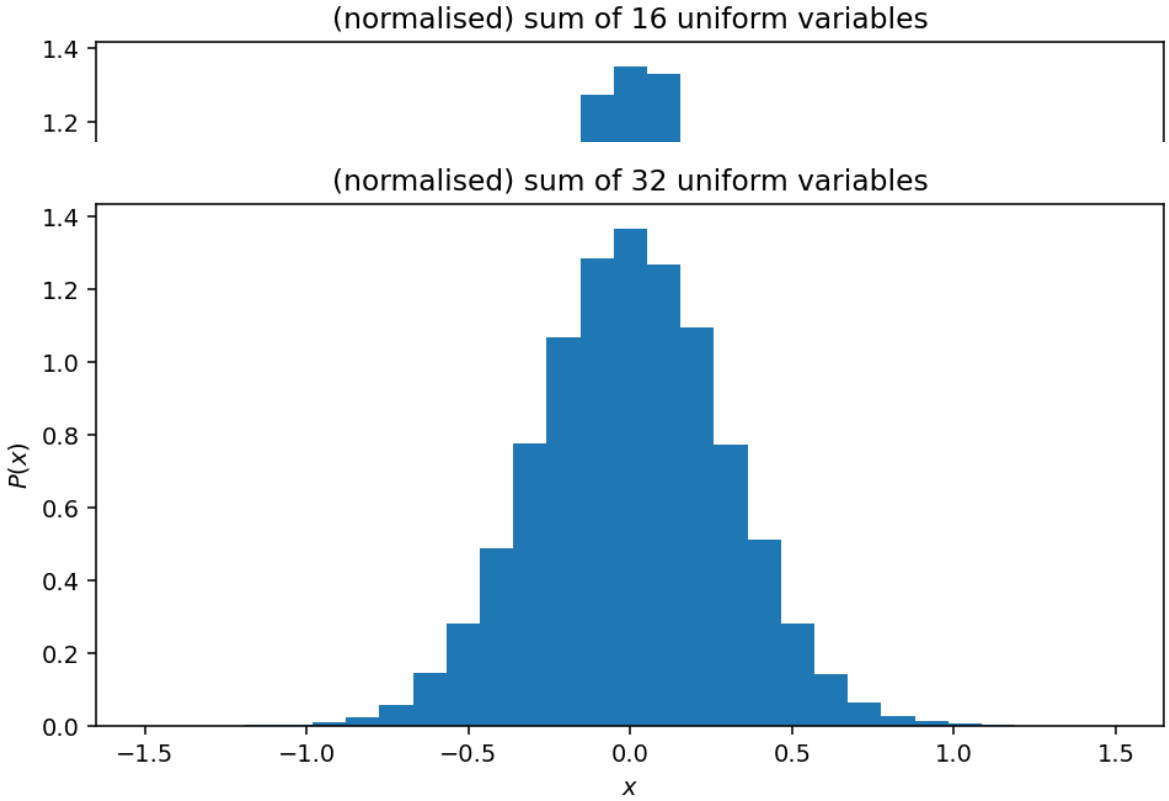
```
clt()
```


(normalised) sum of 1 uniform variables
P(x)
x
0.0
0.2
0.4
0.6
0.8
1.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5


(normalised) sum of 2 uniform variables
P(x)
x
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5


(normalised) sum of 3 uniform variables
1.2
1.0
(normalised) sum of 4 uniform variables
1.2
1.0
0.8
0.6
0.4
0.2
0.0
P(x)
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
x


(normalised) sum of 8 uniform variables
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
P(x)
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
x



Multivariate distributions: distributions over $\mathbb{R}^n$

Continuous distributions generalise discrete variables (probability mass functions) (e.g. over $\mathbb{Z}$) to continuous spaces over $\mathbb{R}$ via probability density functions.

Probability densities can be further generalised to vector spaces, particularly to $\mathbb{R}^n$. This extends PDFs to assign probability across an entire vector space, under the constraint that the (multidimensional integral)

$$\int_{\mathbf{x}\in\mathbb{R}^n} f_X(\mathbf{x}) = 1.$$

This is the same as:

$$\int_{x_0=-\infty}^{x_0=\infty} \int_{x_1=-\infty}^{x_1=\infty} \dots \int_{x_n=-\infty}^{x_n=\infty} f_X([x_0, x_1, \dots, x_n])\, dx_0 dx_1 \dots dx_n = 1.$$

Distributions with PDFs over vector spaces are called **multivariate distributions** (which isn't a very good name; vector distributions might be clearer). In many respects, they work the same as **univariate** continuous distributions. However, they typically require more parameters to specify their form, since they can vary over more dimensions.

Multivariate uniform

The multivariate uniform distribution is particularly simple to understand. It assigns equal density $f_X(x_i) = f_X(x_j)$ to some (axis-aligned) box in a vector space $\mathbb{R}^n$, such that

$$\int_{\mathbf{x}} f_X(\mathbf{x}) = 1, \mathbf{x} \in \text{a box}.$$

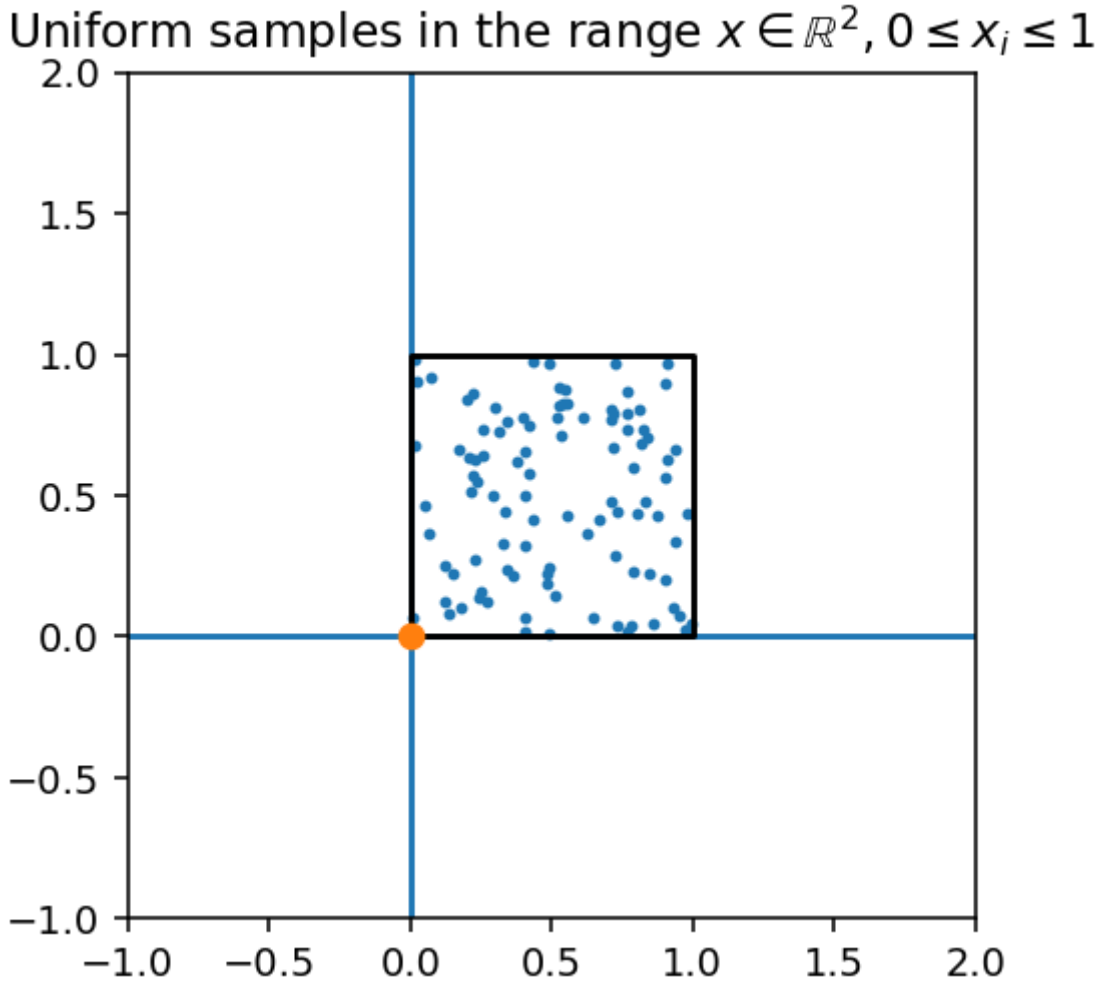
It is trivial to sample from; we just sample *independently* from a one-dimensional uniform distribution in the range [0,1] to get each element of our vector sample. This is a draw from a n-dimensional uniform distribution in the unit box.

In [8]:

```
def uniform_nd(n, A = None, b=None, d=2):
    if A is None:
        A = np.eye(d)
    if b is None:
        b = np.zeros(d)
    return np.random.uniform(0,1, (n,d)) @ A + b, A, b

def draw_pts_2d(pts, A, b):
    # draw the points
    fig = plt.figure()
    ax = fig.add_subplot(1,1,1)
    ax.plot(pts[:,0], pts[:,1], '.', markersize=4)
    # set sensible scales
    ax.set_xlim(-1,2)
    ax.set_ylim(-1,2)
    ax.axvline(0)
    ax.axhline(0)
    ax.set_aspect("equal")
    # draw a bounding box
    bounding = np.array([[0,0],
                         [0,1],
                         [1,1],
                         [1,0],
                         [0,0]]) @ A + b


    ax.plot(bounding[:,0], bounding[:,1], 'k')
    ax.plot(bounding[0,0], bounding[0,1], 'C1o')

pts, A, b = uniform_nd(100, d=2)
draw_pts_2d(pts, A, b)
plt.gca().set_title("Uniform samples in the range $x\in \\mathbb{R}^2, 0 \leq x_i \l
```

Out[8]:

```
Text(0.5, 1.0, 'Uniform samples in the range $x\\in \\mathbb{R}^2, 0
\\leq x_i \\leq 1 $')
```



Transformed uniform distribution

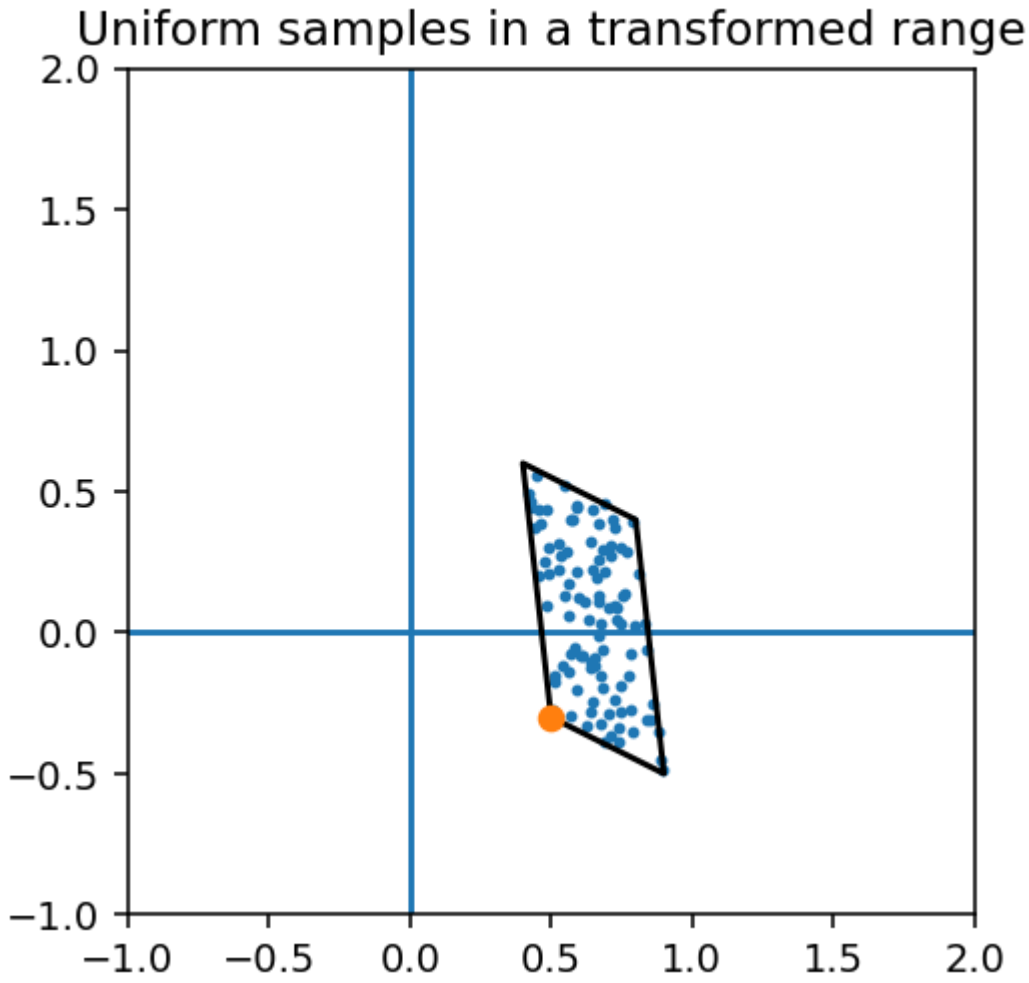
If we want to define a distribution over any box, we can simply transform the vectors with a matrix A and shift by adding an offset vector $\mathbf{b}$

In [9]:

```
# A is "spread"
A = np.array([[0.4, -0.2],
              [-0.1, 0.9]])

# b is "offset" or "centre"
b = np.array([0.5, -0.3])

pts, A, b = uniform_nd(100, A=A, b=b, d=2)
draw_pts_2d(pts, A=A, b=b)
plt.gca().set_title("Uniform samples in a transformed range")
```

Out[9]:

```
Text(0.5, 1.0, 'Uniform samples in a transformed range')
```



Normal distribution

The normal distribution (above) is very widely used as the distribution of continuous random variables. It can be defined for a random variable of *any dimension*; a **multivariate normal** in statistical terminology. In Unit 5, we saw the idea of a **mean vector** μ and a **covariance matrix** Σ which captured the "shape" of a dataset in terms of an ellipse. *These are in fact the parameterisation of the multivariate normal distribution.*

A multivariate normal is fully specified by a mean vector μ and the covariance matrix Σ. If you imagine the normal distribution to be a ball shaped mass in space, the mean *translates* the mass, and covariance applies a transformation matrix (scale, rotate and shear) to the ball.

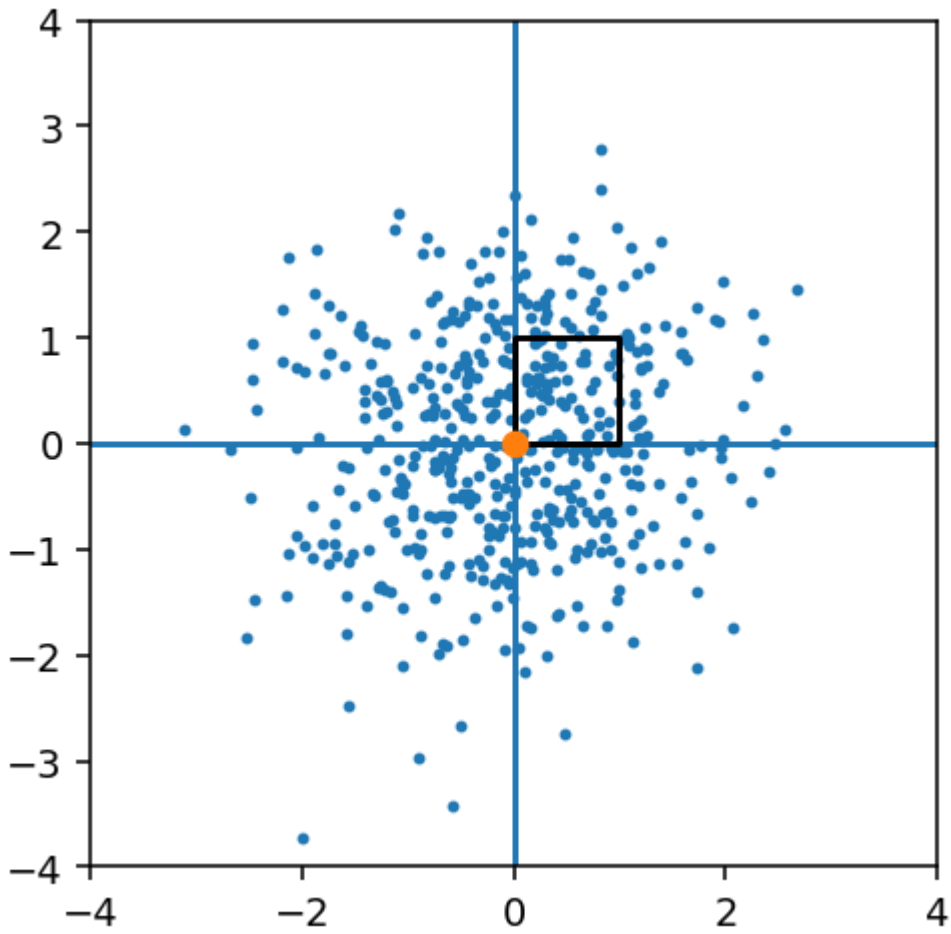
Just like the uniform distribution, we can think of drawing samples from a "unit ball" with an independent normal distribution in each dimension. These samples are transformed linearly by the covariance matrix Σ and the mean vector μ, just like A and b above (though Σ is actually $A^{-\frac{1}{2}}$ for technical reasons)

In [10]:

```
def normal_nd(n, A = None, b=None, d=2):
    if A is None:
        A = np.eye(d)
    if b is None:
        b = np.zeros(d)
    return np.random.normal(0,1, (n,d)) @ A + b, A, b

pts, A, b = normal_nd(500, A=None, b=None, d=2)
draw_pts_2d(pts, A=A, b=b)
plt.gca().set_xlim(-4,4)
plt.gca().set_ylim(-4,4)
```

Out[10]:

$(-4.0,\ 4.0)$

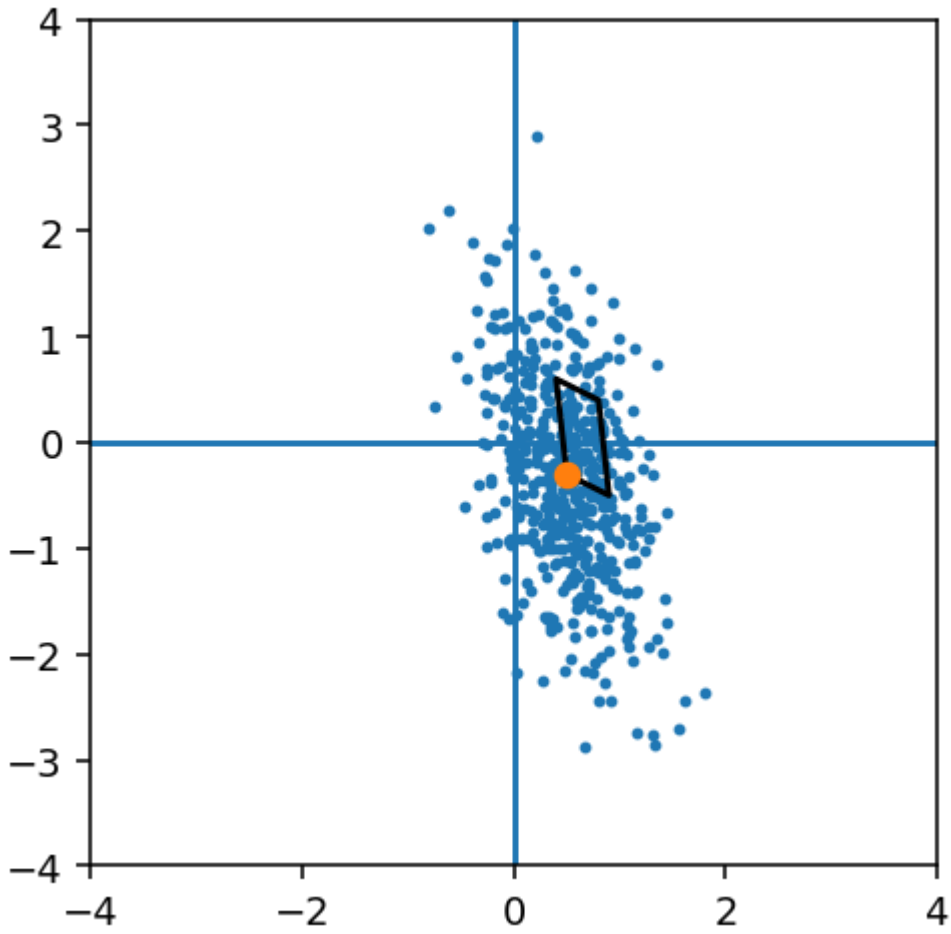


In [11]:

```
# A is "spread"
A = np.array([[0.4, -0.2], [-0.1, 0.9]])
# b is "offset" or "centre"
b = np.array([0.5, -0.3])
pts, A, b = normal_nd(500, A=A, b=b, d=2)
draw_pts_2d(pts, A=A, b=b)
plt.gca().set_xlim(-4,4)
plt.gca().set_ylim(-4,4)
```

Out[11]:

$(-4.0,\ 4.0)$

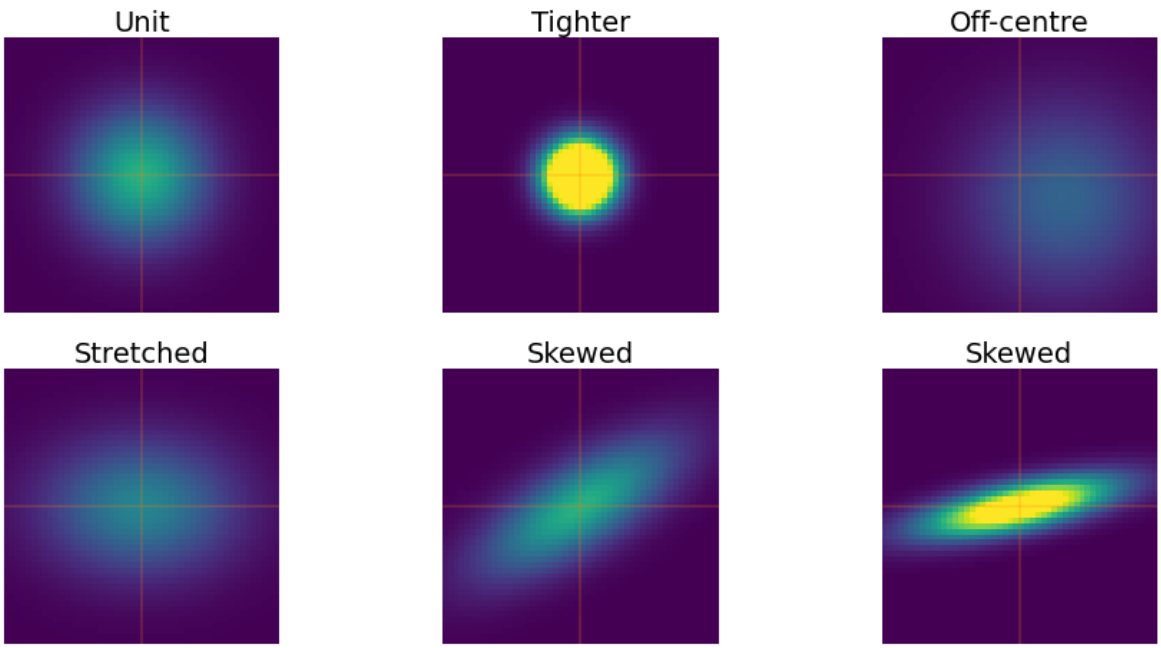


Joint and marginal PDFs

We can look at the PDF of a multivariate normals for different covariances and mean vector (centres and spreads).

In [12]:

```
import scipy.stats
def demo_normal(ax, mean, cov, title):
    x,y = np.meshgrid(np.linspace(-3,3,50), np.linspace(-3,3,50))
    pos = np.empty(x.shape + (2,))
    pos[:,:,0] = x
    pos[:,:,1] = y
    joint_pdf = scipy.stats.multivariate_normal.pdf(pos, mean, cov)
    ax.pcolor(x,y,joint_pdf, cmap='viridis', vmin=0, vmax=0.25)

    ax.axhline(0, color='C1', linewidth=0.2)
    ax.axvline(0, color='C1', linewidth=0.2)
    ax.text(0, 3.2, title, ha='center')
    ax.axis("off")
    ax.axis("image")


fig = plt.figure()
ax = fig.add_subplot(2,3,1)
demo_normal(ax, [0,0], [[1,0],[0,1]], "Unit")
ax = fig.add_subplot(2,3,2)
demo_normal(ax, [0,0], [[0.25,0],[0,0.25]], "Tighter")
ax = fig.add_subplot(2,3,3)
demo_normal(ax, [1,-0.5], [[2,0],[0,2]], "Off-centre")
ax = fig.add_subplot(2,3,4)
demo_normal(ax, [0,0], [[2,0],[0,1]], "Stretched")
ax = fig.add_subplot(2,3,5)
demo_normal(ax, [0,0], [[2,0.1],[1,1]], "Skewed")
ax = fig.add_subplot(2,3,6)
demo_normal(ax, [0,0], [[2,-0.9],[0.4,0.2]], 'Skewed')
```



Joint and marginal distributions

We can now talk about the **joint probability density function** (density over all dimensions) and the **marginal probability density function** (density over some sub-selection of dimensions).

For example, consider $X \sim N(\mu, \Sigma)$, $X \in \mathbb{R}^2$, a two dimensional ("bivariate") normal distribution. We can look at some examples of the PDF, showing:

- Joint $P(\mathbf{X})$
- Marginal $P(X_1)$ and $P(X_2)$
- Conditionals $P(X_1|X_2)$ and $P(X_2|X_1)$

In [13]:

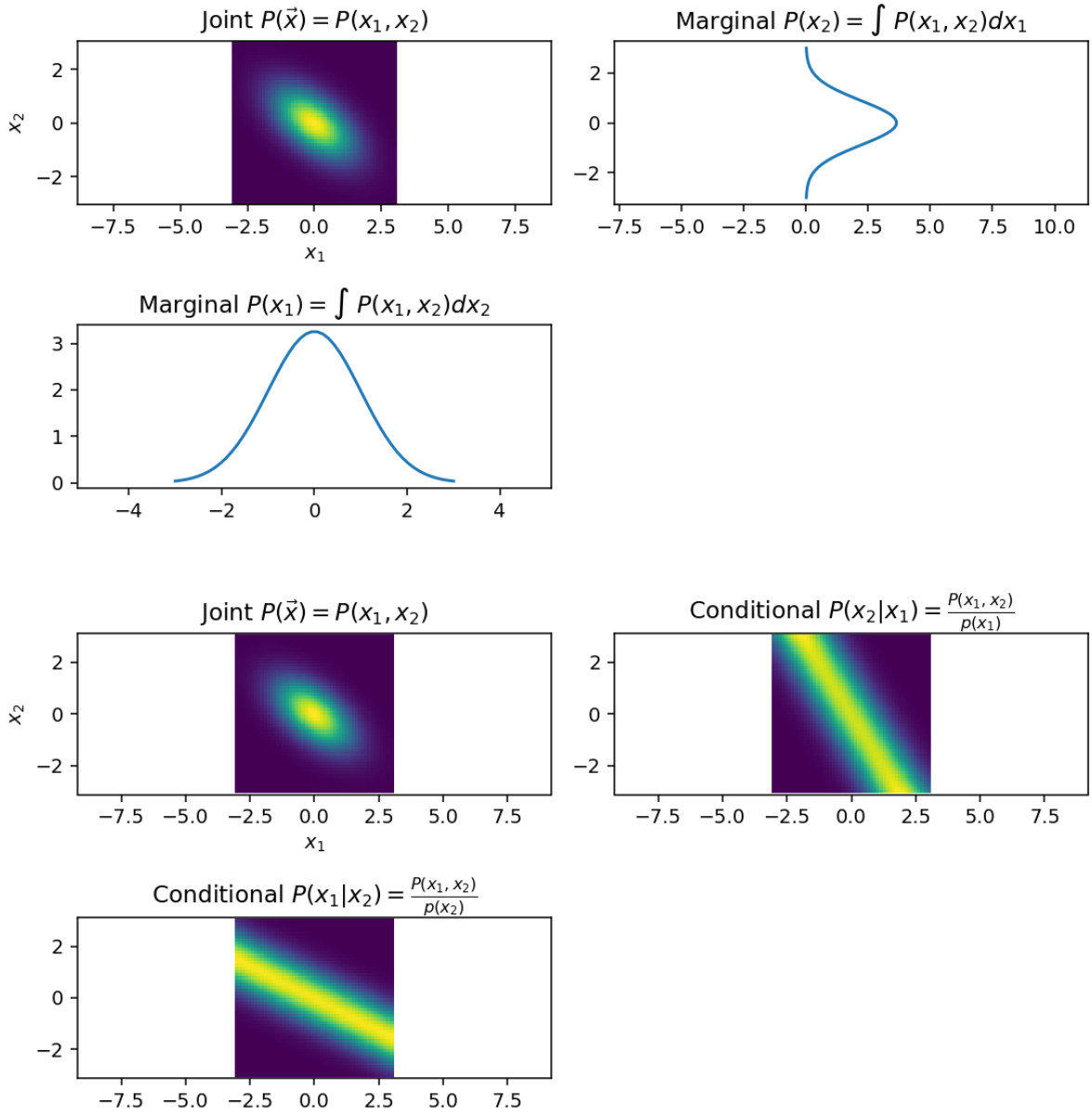
```
import scipy.stats
def joint_marginal(cov):
    # create an independent 2D normal distribution
    x,y = np.meshgrid(np.linspace(-3,3,50), np.linspace(-3,3,50))
    pos = np.empty(x.shape + (2,))
    pos[:,:,0] = x
    pos[:,:,1] = y
    joint_pdf = scipy.stats.multivariate_normal.pdf(pos, [0,0], cov)
    fig = plt.figure()
    # plot the joint
    ax = fig.add_subplot(2,2,1)
    ax.axis('equal')
    ax.set_xlabel("$x_1$")
    ax.set_ylabel("$x_2$")
    plt.title("Joint $P(\\vec{x})=P(x_1,x_2)$")
    ax.pcolor(x,y,joint_pdf, cmap='viridis')
    # plot the marginals
    ax = fig.add_subplot(2,2,3)
    ax.axis('equal')
    plt.title("Marginal $P(x_1) = \int\  P(x_1,x_2) dx_2$")
    ax.plot(x[0,:], np.sum(joint_pdf, axis=0))
    ax = fig.add_subplot(2,2,2)
    ax.axis('equal')
    plt.title("Marginal $P(x_2) = \int\  P(x_1,x_2) dx_1$")
    ax.plot(np.sum(joint_pdf, axis=1), x[0,:])
    plt.tight_layout()

    # plot p(x|y) and p(y|x)
    fig = plt.figure()
    # plot the joint
    ax = fig.add_subplot(2,2,1)
    ax.axis('equal')
    ax.set_xlabel("$x_1$")
    ax.set_ylabel("$x_2$")
    plt.title("Joint $P(\\vec{x})=P(x_1,x_2)$")
    ax.pcolor(x,y,joint_pdf, cmap='viridis')
    ax = fig.add_subplot(2,2,3)
    ax.axis('equal')
    plt.title("Conditional $P(x_1|x_2) = \\frac{P(x_1,x_2)}{p(x_2)}$")
    marginal = np.tile(np.sum(joint_pdf, axis=0), (joint_pdf.shape[0],1))
    ax.pcolor(x,y,joint_pdf/marginal, cmap='viridis')
    plt.tight_layout()
    ax = fig.add_subplot(2,2,2)
    ax.axis('equal')
    plt.title("Conditional $P(x_2|x_1) = \\frac{P(x_1,x_2)}{p(x_1)}$")
    marginal = np.tile(np.sum(joint_pdf, axis=1), (joint_pdf.shape[0],1))
    ax.pcolor(x,y,joint_pdf/marginal.T, cmap='viridis')
    plt.tight_layout()
```

In [14]:

```
joint_marginal([[1, -0.3], [-0.5, 0.8]])
print_matrix("\mu", np.array([0.0, 0.0]))
print_matrix("\Sigma", np.array([[1, 0], [0.5, 1]]))
plt.tight_layout()
```

$$\mu = \begin{bmatrix} 0 & 0 \end{bmatrix}$$

$$\Sigma = \begin{bmatrix} 1.0 & 0 \\ 0.5 & 1.0 \end{bmatrix}$$



Ellipses

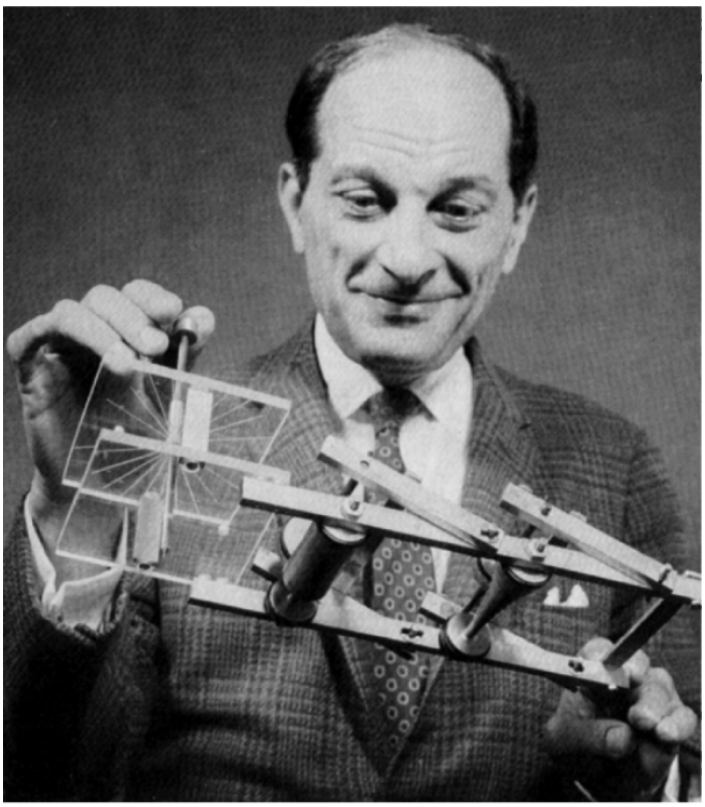
When we spoke informally about the covariance matrix "covering" the data with an ellipsoidal shape, we more precisely meant that *if* we represented the data as being generated with a normal distribution, and chose the mean vector and covariance matrix that best approximated that data, then the contours of the density of the PDF corresponding to equally spaced isocontours would be ellipsoidal.

Monte Carlo

How do we *draw* samples from a continuous distribution? How can we simulate the outcomes of a random variable X? This is a vital tool in computational statistics. One of the reasons computers are useful for statistical analysis is that they can generate (pseudo)-random numbers very quickly.

von Neumann and Ulam

During the *Manhattan project* that developed the atomic bomb during the Second World War, there were many difficult probabilistic equations to work out. Although *analytical techniques* for solving certain kinds of problems existed, they were only effective some narrow types of problem and were tricky to apply to the problems that the Manhattan project had to solve.



Stanslaw Ulam

John von Neumann

John von Neumann and Stanislaw Ulam developed the **Monte Carlo** method to approximate the answer to probabilistic problems, named after the casinos of Monte Carlo. This involved setting up a *simulation* with stochastic (random) components. By running the simulation many times with different random behaviour, the population of *possible* behaviours could be approximated.

For example, computing the expectation of a function of a random variable can often be hard for continuous random variables. The integral for:

$$\mathbb{E}[g(X)] = \int_x f_X(x) g(x)\, dx$$

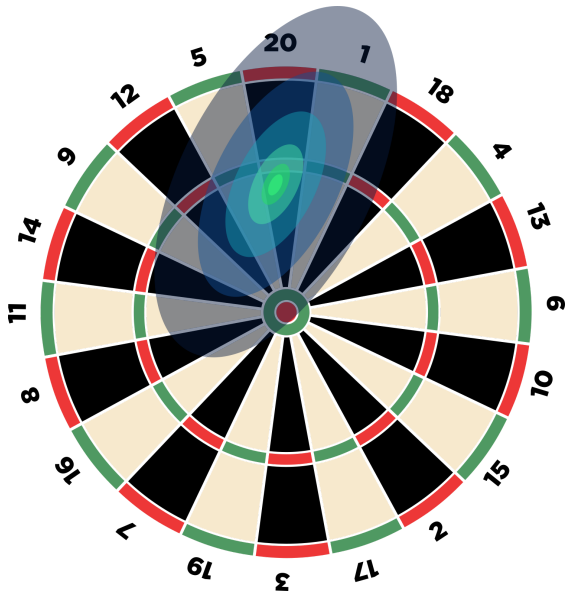
may be intractable. However it is often very easy to compute $g(x)$ for any possible x. If we can somehow sample from the distribution $P(X = x)$, then we can approximate this very easily:

$$\mathbb{E}[g(X)] = \int_x f_X(x) g(x)\, dx \approx \frac{1}{N} \sum_{i=1}^{N} g(x_i)\,,$$

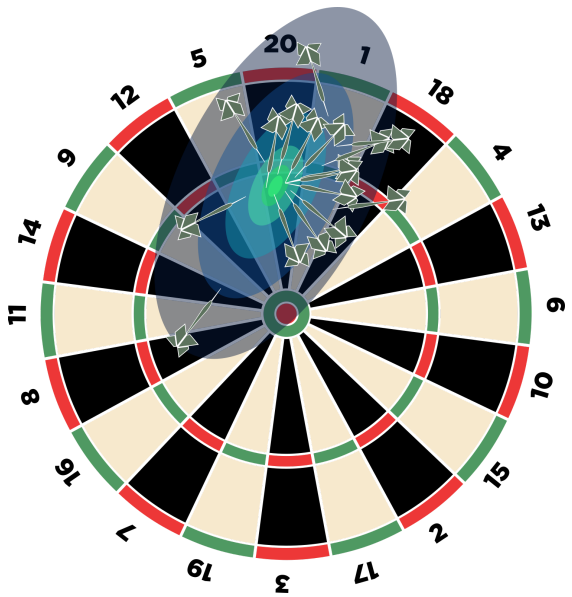
where x_i are random samples from $P(X = x)$, defined by the PDF $f_X(x)$. This gets better as N gets larger.

Throwing darts

For example, imagine trying to work out the expectation of dart throw. A dart board has sections giving different scores. We might model the position of the dart as a normal distribution over the dart space. This models the human variability in throwing. The expected score of a throw requires evaluating the integral of the normal PDF multiplied by the score at each point -- which isn't feasible to compute directly.



But we can sample from a multivariate normal distribution easily; we saw this in the last unit; just sample from d independent standard normals, and transform with a linear transform (matrix). So instead of trying to solve a very hard integral, we can simulate lots of dart throws, which follow the pattern of the normal distribution, and take the average score that they get. If we simulate a lot of darts, the average will be close to the true value of the integral.



Bullseye example

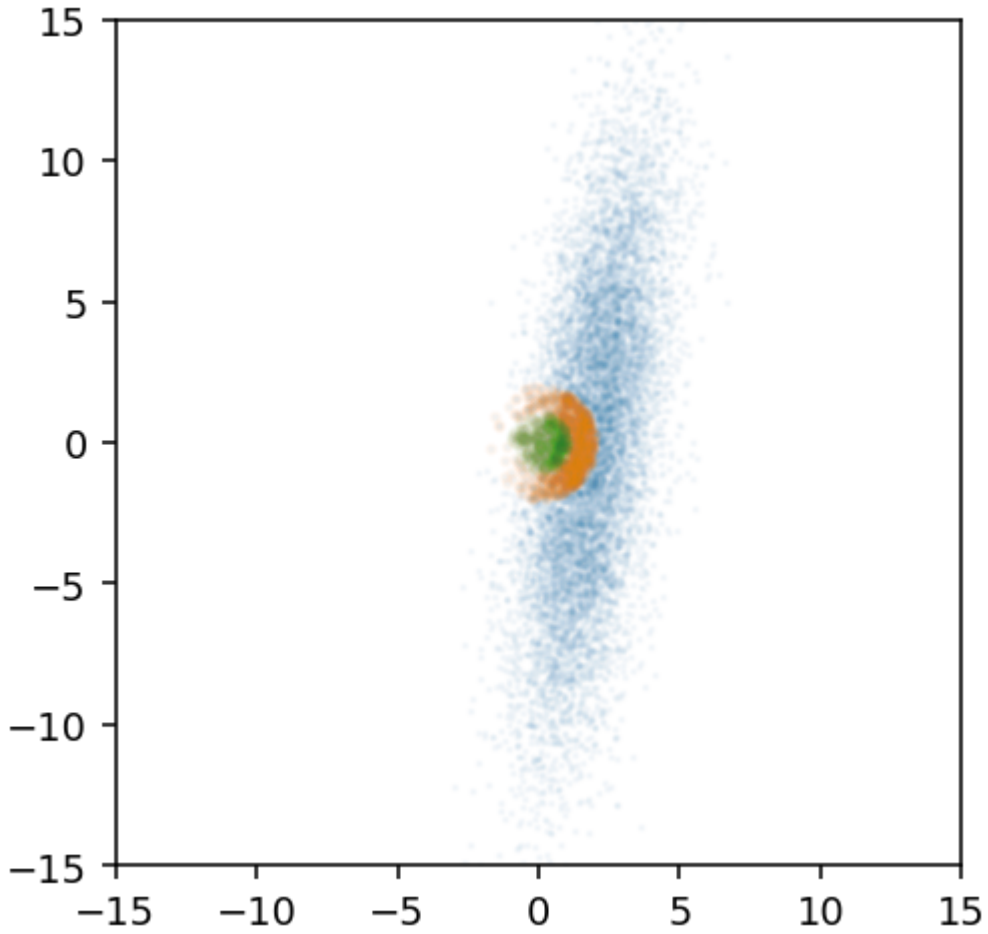
For example, we might want to define a circular score region, which gives us 25 points if we land in it, 50 points if we lie in a smaller coencentric circle, and 0 otherwise; this is our function $g(X)$. We might model the throw of the dart with a multivariate normal. How do we compute the expected score $E[g(X)]$?

In [15]:

```
# centre of throw distribution
mu = [1.9, 0.3]

# spread
sigma = [[4, 27.5],
         [-0.2, 7]]

N = 10000

# draw N samples
throws = scipy.stats.multivariate_normal(mean=mu, cov=sigma).rvs(N)

# check radius from a centre point
outer = np.linalg.norm(throws - np.array([0,0]), axis=1) < 2
inner = np.linalg.norm(throws - np.array([0,0]), axis=1) < 1

# plot the results
fig, ax = plt.subplots()
ax.scatter(throws[:,0], throws[:,1], s=1, alpha=np.sqrt(10.0/N))
ax.scatter(throws[outer,0], throws[outer,1], s=5, c='C1', alpha=np.sqrt(10.0/N))
ax.scatter(throws[inner,0], throws[inner,1], s=8, c='C2', alpha=np.sqrt(10.0/N))
ax.set_aspect("equal")
ax.set_xlim(-15, 15)
ax.set_ylim(-15, 15);

# compute approximate expectation
print("E[g(X)] ~=", np.mean(outer*25 + inner*50))
```

```
E[g(X)] ~= 4.3475
```

```
/home/nicolas/anaconda3/lib/python3.9/site-packages/scipy/stats/_multi
variate.py:653: RuntimeWarning: covariance is not positive-semidefinit
e.
  out = random_state.multivariate_normal(mean, cov, size)
```

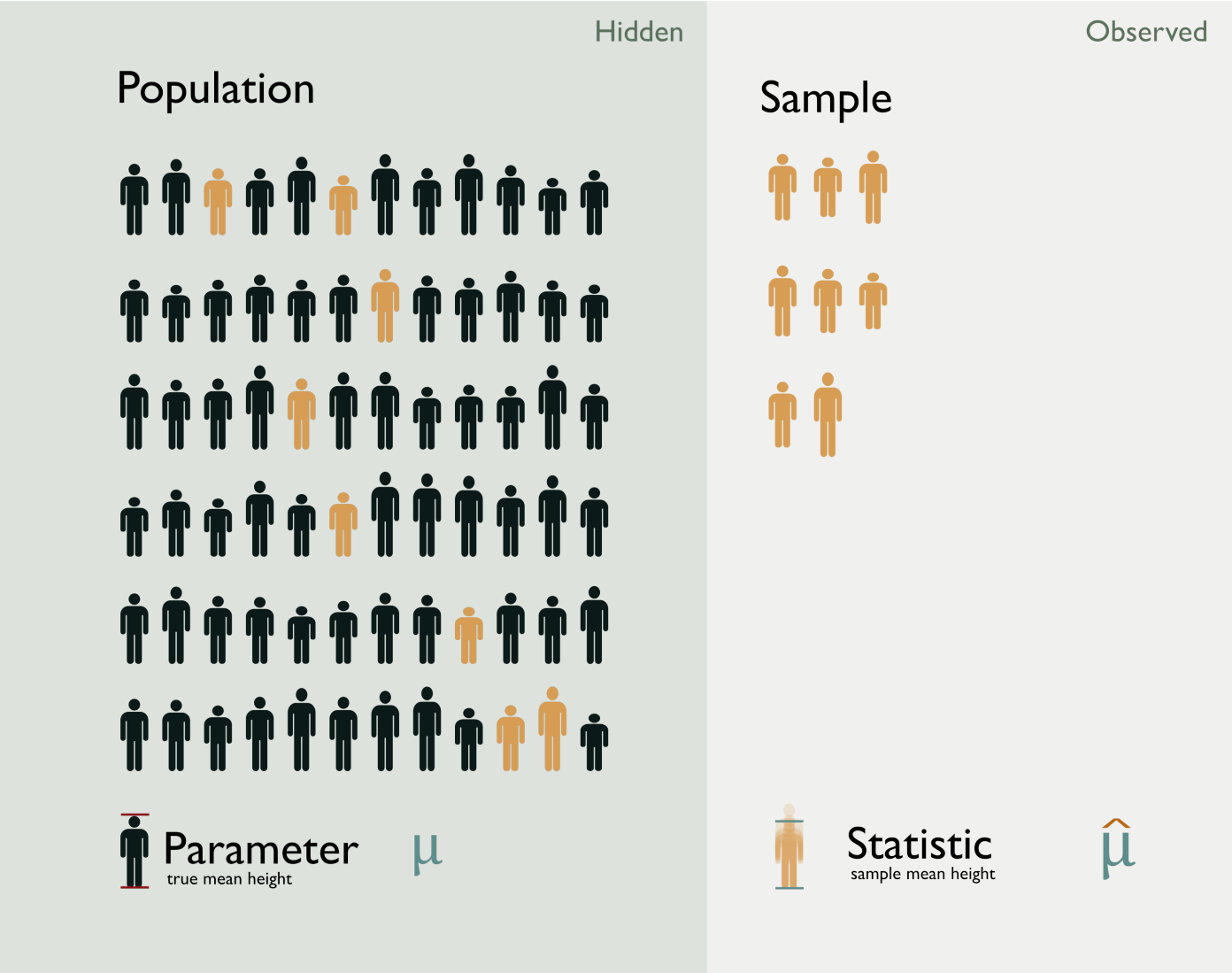


While this might seem like a unsubstantiated "hack", random sampling approaches in fact have strong statistical guarantees and are exceptionally powerful methods for performing **inference.**

Inference

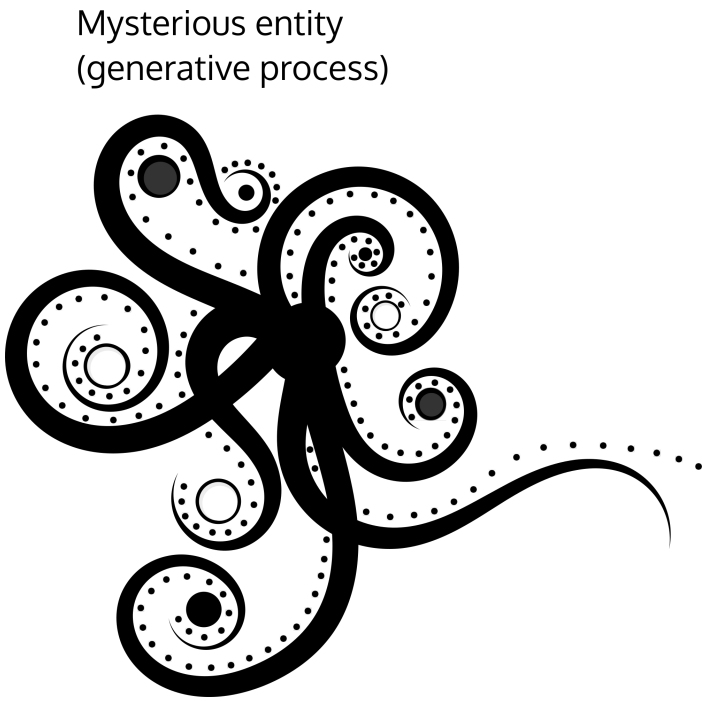
Population and samples, statistics and parameters

Inferential statistics is concerned with estimating the properties of an unobserved "global" **population** of values from a limited set of observed **samples**. This assumes that there is some underlying distribution from which samples are being drawn. This is a hidden process (the "mysterious entity"), which we only partially observe through the samples we see.



- **Population** is the unknown set of outcomes (which might be infinite)
  - **Example** the weight of all beetles
  - **Parameter** describes this **whole population**, e.g. the mean weight of all beetles
- **Sample** is some subset of the population that has been observed.
  - **Example** 20 beetles whose weight has been measured
  - **Statistic** is a function of the sample data, e.g. the arithmetic mean of those 20 samples

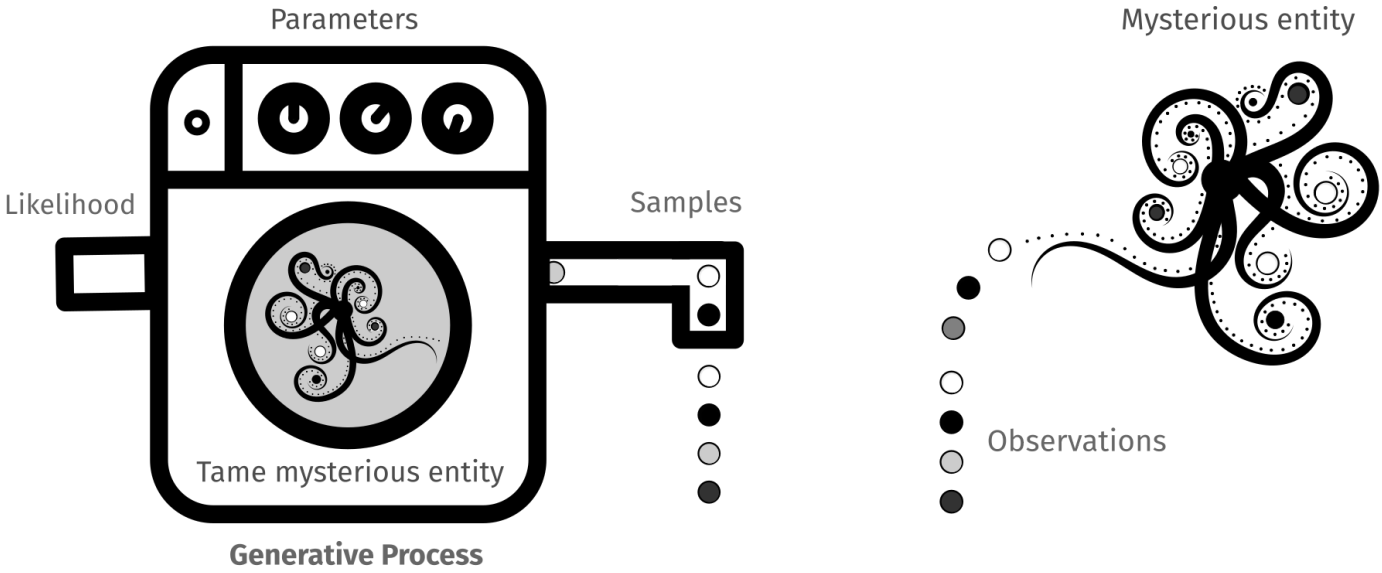
The parameters of the population distribution govern the generation of the samples that are observed. The problem of statistics is how to **infer** parameters given samples.



Our old friend; the mysterious entity

Our model of the world is that there is some unknown entity which **generates** data that we observe, according to some definite but unknown rules. These rules are codified by a **distribution** (a "type" of rule) and **parameters** (the specifics of rules applied). We assume the model has some *randomness* or *stochastic* elements, either because it truly does, or because this makes it simpler to represent the model.

Inference is the process of determining these rules (i.e. the parameters) by looking at the aftermath of the actions of the mysterious entity. These are the **samples** or **observations** that we have. From these we can work out what must have been going on in the mysterious entities world. Or at least approximate it as well as we can. We usually assume that we know or have chosen a specific **distribution** which we expect to be governing the process, and focus on identifying the parameters involved.

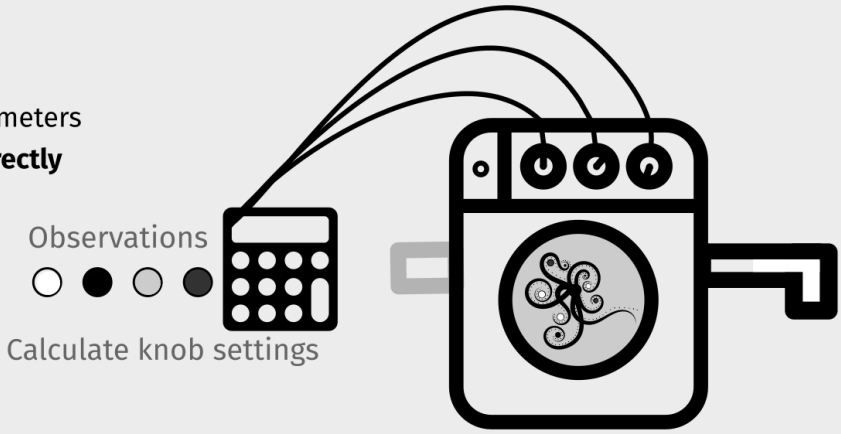


Estimator

Look at observations
Algorithmically adjust parameters
Adjust process to match **directly**

Direct estimation

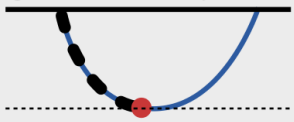
$\hat{\mu} \Rightarrow \mu$

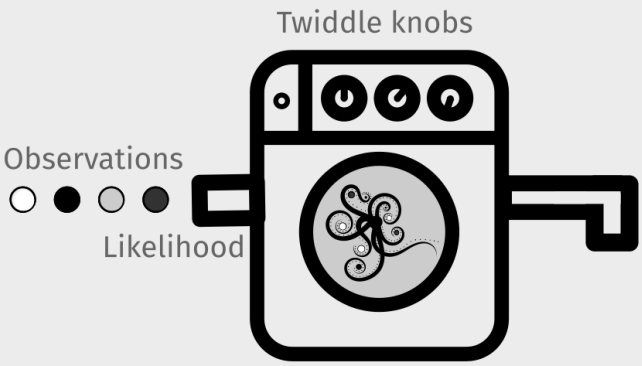


Maximum Likelihood

Compute likelihood of each observation
Tweak knobs to maximise likelihood
Optimise liklihood (usually negative log likelihood is easier)

Log-likelihood optimisation

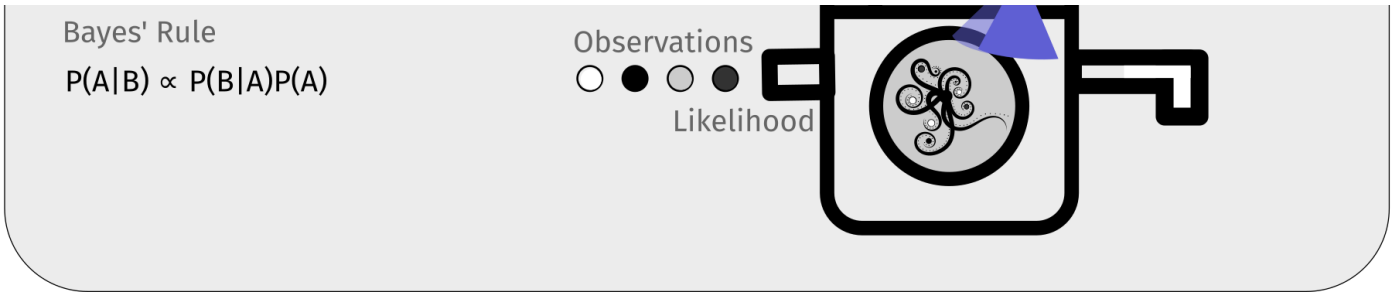




Bayesian Inference

Form guesses over possible knob states
Update guesses based on likelihood
Result in **distributions** over knob settings

Update knob **beliefs**
Prior -> Posterior



Two worldviews

- **Bayesian inference** means that we consider *parameters* to be random variables that we want to refine a distribution over, and that data are fixed (known, observed data). We talk about belief in particular parameter settings.
- **Frequentist inference** means that we consider *parameters* to be fixed, but data to be random. We talk about how we approach an accurate estimate of the true parameters as our sample size increases.

Three approaches

We will see three different approaches to doing inference:

- **Direct estimation** of parameters, where we define *functions of observations* that will estimate the values of parameters of distributions *directly*. This requires we assume the form of the distribution governing the mysterious entity. It is very efficient, but only works for very particular kinds of model. We need to have *estimator functions* for each specific distribution we want to estimate, which map observations into parameter estimates.
- **Maximum likelihood estimation** of parameters, where we use **optimisation** to find parameter settings that make the the observations appear as likely as possible. We can see this as tweaking the parameters of some predefined model until they "best align" with the observations we have. This requires an iterative optimisation process, but it works for any model where the distribution has a known likelihood function (that is we can compute how likely observations were to have been generated by that model).
- **Bayesian, probabilistic** approaches explicitly encode belief about the behaviour of the mysterious entity using probability distributions. In Bayesian models, we assume a distribution over the parameters themselves, and consider the *parameters to be random variables*. We have an initial hypotheses for these parameters ("prior") and we use observations to update this belief to hone our estimate of the parameters to a tighter (hopefully) distribution ("posterior"). Unlike the other methods, we do not estimate a single "parameter setting", but instead we always have a distribution over possible parameters which changes as data is observed. This is much more robust and arguably more coherent way to do inference, but it is harder to represent and harder to compute. We require both **priors** over parameters, and a **likelihood function** that will tell us how likely data is to have been generated under a particular parameter setting.

Note: there are more general forms of Bayesian inference, like Approximate Bayesian Computation (ABC) which do not even require likelihood functions, just the ability to sample from distributions. We will not discuss these.

An inference scenario: The app rating problem

Original [Image](https://flickr.com/photos/smemon/5324223435 "apps") by [Sean MacEntee](https://flickr.com/people/smemon) shared [CC BY](https://creativecommons.org/licenses/by/2.0/)

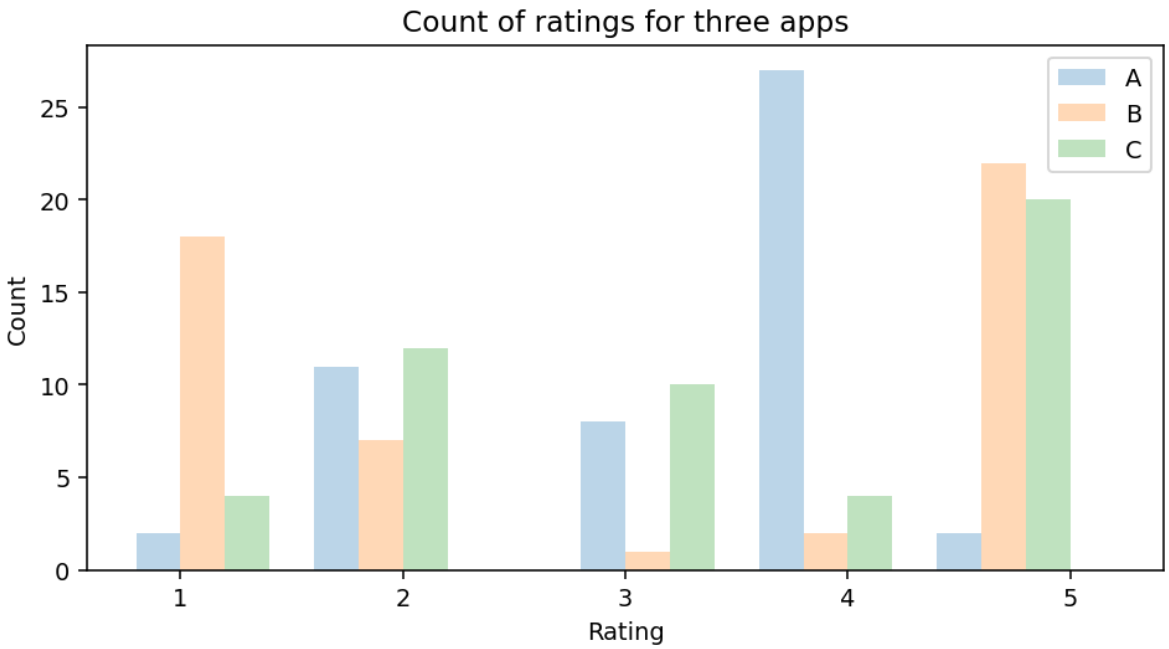
You've written an app, and it'll make you rich. If you can make a version that people really like, that is. So maybe you've released a few different beta version to try out some options. Each user has rated the app with 1-5 stars. You need to work out which version is better.

In [16]:

```
def sample_pmf(ps, n):
    return np.digitize(np.random.uniform(0,1,n), np.cumsum(ps))

app_A = sample_pmf([0.05, 0.2, 0.2, 0.5, 0.05], 50) + 1
app_B = sample_pmf([0.4, 0.125, 0.025, 0.05, 0.4], 50) + 1
app_C = sample_pmf([0.1, 0.2, 0.2, 0.20, 0.30], 50) + 1
apps = [app_A, app_B, app_C]

fig, ax = plt.subplots()
ax.hist(app_A, alpha=0.3, align='left', label="A", width=0.2)
ax.hist(app_B, alpha=0.3, label="B", width=0.2)
ax.hist(app_C, alpha=0.3, align='right', label="C", width=0.2)
ax.legend()
ax.set_xlabel("Rating")
ax.set_ylabel("Count")
ax.set_title("Count of ratings for three apps")
```

Out[16]:

```
Text(0.5, 1.0, 'Count of ratings for three apps')
```



The problem is that you only have a sample of responses. Not every user rated the app, and actually you don't even care about the users who have *already* bought the app and rated it -- you want to know how *prospective* customers will view it, and by definition you cannot have sampled from this population of users.

Imagine we are trying to infer the distribution of app scores *assuming that they were generated by a normal distribution*. That is, we imagine there is some function like this:

In [17]:

```
def app_samples(mu, sigma):
    return np.random.normal(mu, sigma)
```

This isn't a very good approximation to the real samples; for example, it can generate negative ratings, or ratings with fractional values. It also assumes that all of the ratings that we see are independent of each other

(**independence assumption**) and that they are all drawn from the *same* underlying distribution (**idential distribution assumption**).

But it is simple to work with, and the problem we have to solve is: given a collection of return values from this function (samples), what values did `mu` and `sigma` have?

> This is a problem of inference.

Estimators

Unlike discrete distributions, where the PMF can be estimated directly from observations using the empirical distribution (as we did for Romeo and Juliet), there is no analogous direct procedure for continuous distributions.

For many continuous distributions statisticians have developed **estimators**; functions that can be applied to sets of observations to estimate the most likely parameters of a probability density function defining a distribution that might have generated them.

The **form** of the distribution must be decided in advance (for example, the assumption that the data has been generated by an approximately normal distribution); this is usually called the **model**. The specific parameters can then be calculated under the assumption of this model.

Direct estimation

One way of doing inference is to, if we assume a particular *form* of the distribution (e.g. assume it is normal), use **estimators** of **parameters** (such as the mean and variance) of this population distribution. These **estimators** are computed via **statistics** which are summarising functions we can apply to data. *These estimators need to specially derived for each specific kind of problem.*

For example, the arithmetic mean, and the standard deviation of a set of observed samples are **statistics** which are **estimators** of the parameters of μ and σ normal distribution. If we have observations (believed to have been) drawn from a normal distribution, we can estimate the parameters μ and σ of that distribution just by computing the mean and standard deviation.

In other words, we might want to know the "true" average app rating -- the **population mean** of the distribution which is "generating" app ratings. We have an assumption that a random process is creating these ratings, whose operational characteristics (parameters) we can learn from samples. But we can only observe a limited sample of ratings by measuring some specific subset of ratings from users who actually rated the app and computing statistics of the results -- the **sample mean**.

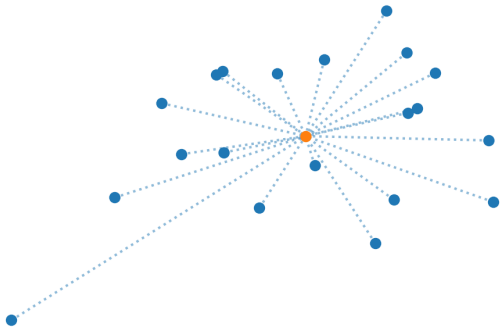


Image: direct estimation uses a statistic, which is a function of data, to directly estimate parameters; for example, the arithmetic mean of a point cloud estimates the mean vector of the population.

Standard estimators

Mean

The **arithmetic mean** is sum of sample values $x_1, x_2, \ldots, x_n$ divided by the number of values:

$$\hat{\mu} = \frac{1}{N} \sum_{i=1}^{N} x_i$$

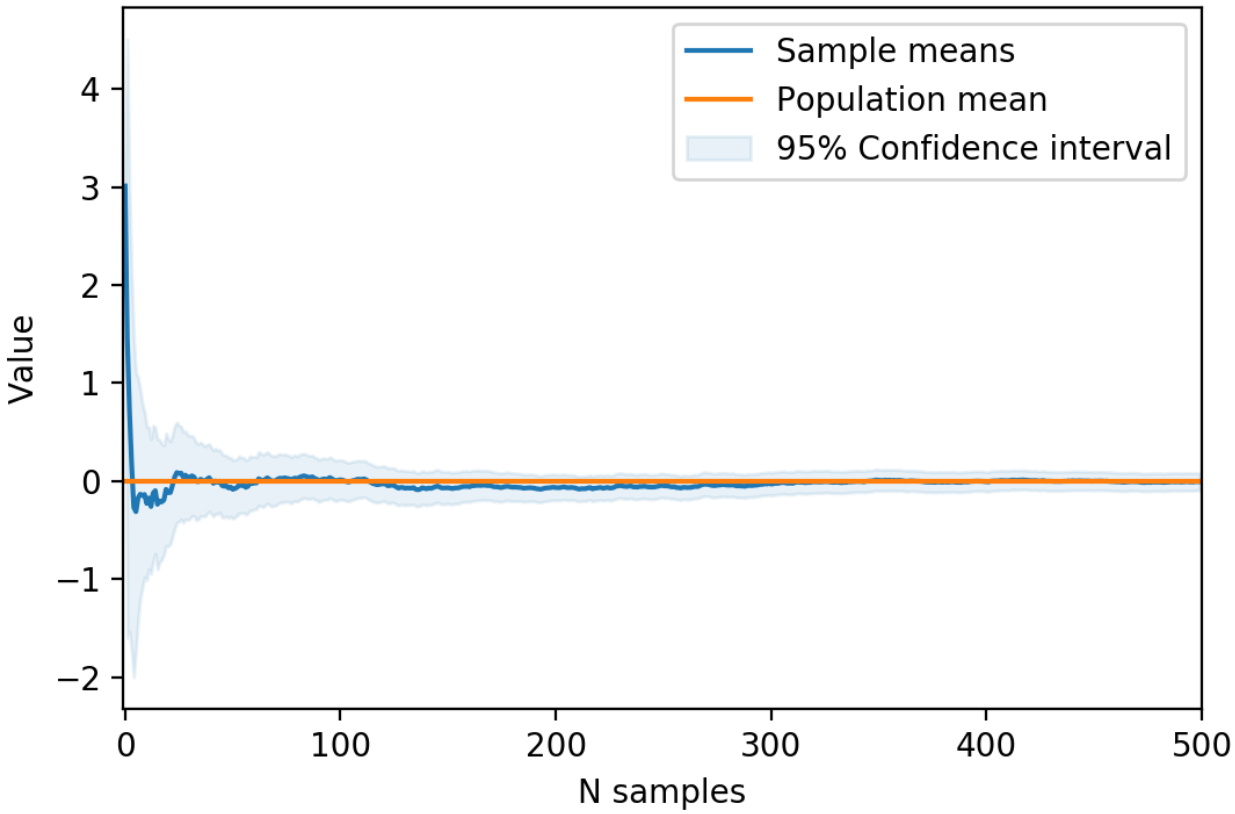
Sample mean

The population mean is $\mu = \mathbb{E}[X]$ for a random variable X. It turns out the *arithmetic mean of the observed samples* or **sample mean**, which we write with a *little hat* $\hat{\mu}$ is a good (footnote: good is what statisticians would call "unbiased") estimator of the true population mean μ. As the number of samples increases, our estimate $\hat{\mu}$ of the population mean μ gets better and better.

It's important to separate the idea of

- the population mean μ, which (usually!) exists but is not knowable directly. It is the expectation of the random variable $\mathbb{E}[X]$.
- the sample mean $\hat{\mu}$ which is just the arithmetic average of samples we have seen (e.g. computed via `np.mean(x, axis=0)`)

The sample mean is a **statistic** (a function of observations) which is an **estimator** of the population mean (which could be a **parameter** of a distribution). Specific bounds can be put on this estimate; the standard error gives a measure of how close we expect that the arithmetic mean of samples is to the population mean, although the interpretation is not straightforward.



The mean measures the **central tendency** of a collection of values. The **mean vector** generalises this to higher dimensions.

Variance and standard deviation

The sample variance is the squared difference of each value of a sequence from the mean of that sequence:

$$\hat{\sigma^2} = \frac{1}{N}\sum_{i=1}^{N}\left(x_i - \hat{\mu}\right)^2.$$

It is an estimator of the population variance, $\mathbb{E}[(X - \mathbb{E}[X])^2]$

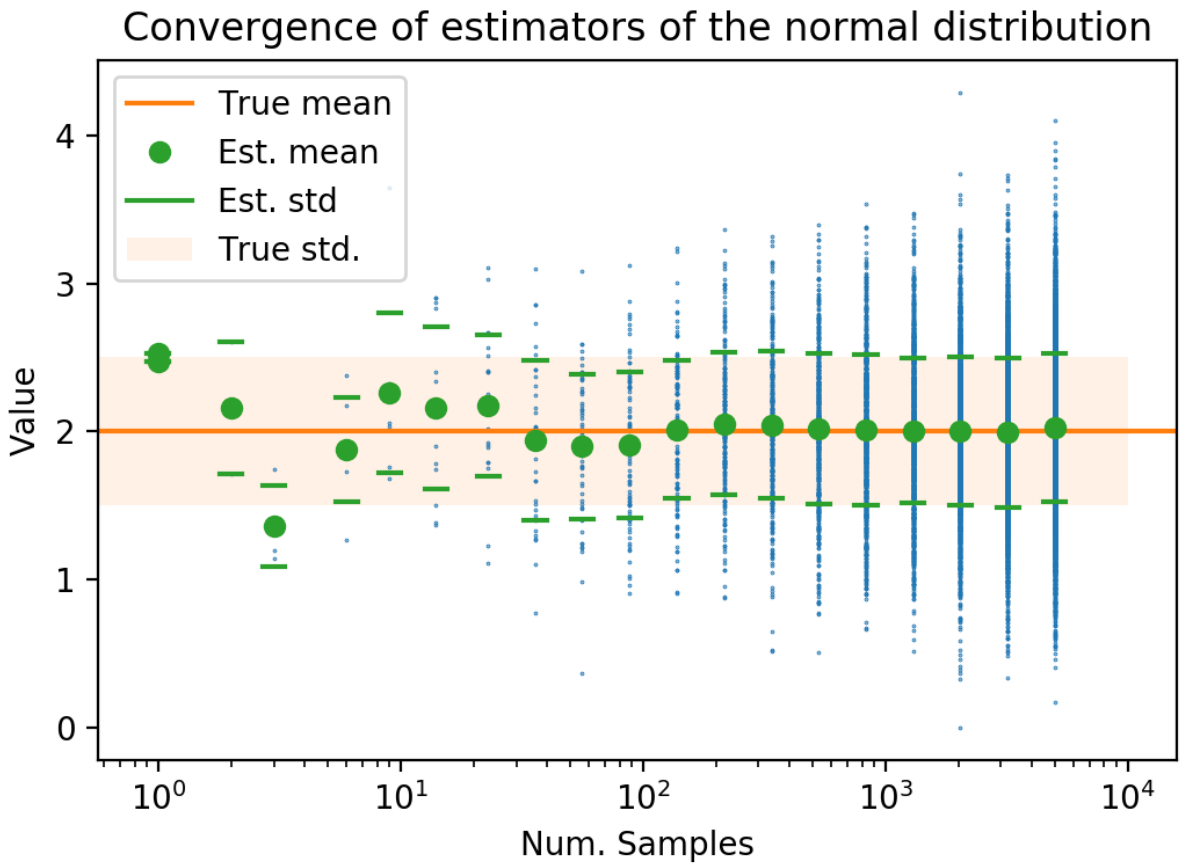
The sample standard deviation is just the square root of this value.

$$\hat{\sigma} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}\left(x_i - \hat{\mu}\right)^2}$$

The variance and the standard deviation measure the **spread** of a collection of values. The **covariance matrix** Σ generalises this idea to higher dimensions.

Relation to normal distribution

If we *assume* that our data is generated by a normal distribution, then the statistics **mean** $\hat{\mu}$ and **variance** $\hat{\sigma^2}$ estimate the parameters μ, σ of that normal distribution, $\mathcal{N}(\mu, \sigma)$. Even if the underlying process isn't exactly normal, it may well be close to being normal because of the Central Limit Theorem. And even if that doesn't apply, the mean and the variance are still useful *descriptive statistics*.



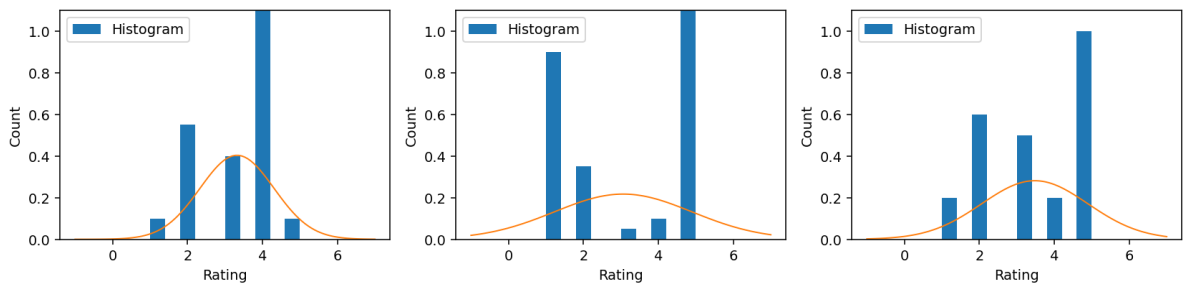
Fitting

What does it mean to estimate the parameters of a normal distribution that might be creating app ratings? We are **fitting** a distribution, governed by a PDF, to a set of observations. In our discrete examples, we could fit a distribution simply by computing the empirical distribution (assuming we had enough samples). But estimating a PDF requires some structure, a space of functions with some parameterisation.

We can visualise this:

In [18]:

```
def plot_fits(apps, mus, sigmas, fit_color='C1'):
    fig = plt.figure(figsize=(12,3))

    for i,(app,mu, sigma) in enumerate(zip(apps, mus, sigmas)):
        ax = fig.add_subplot(1,3,i+1)

        ax.hist(app, density=True, label="Histogram")
        xs = np.linspace(-1.0,7.0,100)

        # compute the PDF function for this setting of mu and sigma
        for m, s in zip(mu, sigma):
            ax.plot(xs, scipy.stats.norm(m, s).pdf(xs),
                            alpha=1.0/np.sqrt(len(mu)), color=fit_color, lw=1)

        # set labeling and axis
        ax.set_ylim(0,1.1)
        ax.set_xlabel("Rating")
        ax.set_ylabel("Count")
        ax.legend()

        plt.tight_layout()

apps = [app_A, app_B, app_C]

# use the sample mean, sample standard deviation
plot_fits(apps, [[np.mean(app)] for app in apps], [[np.std(app)] for app in apps])
```

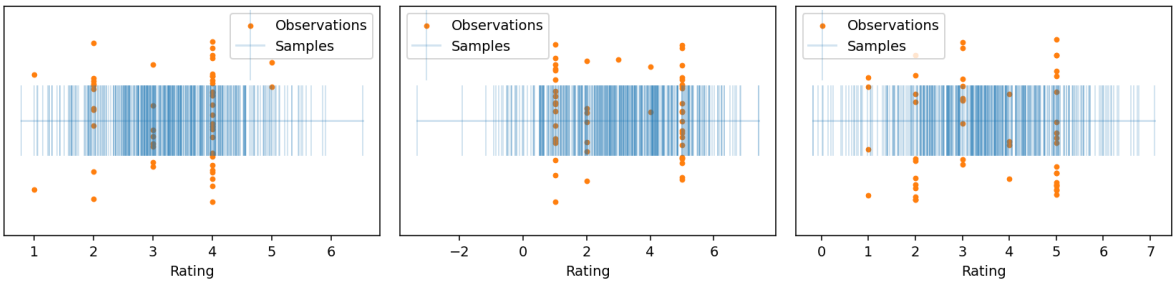


Sampling from the model

We can draw samples from our fitted distribution, and compare them to our results. They won't be a very good representation, because the data we have is clearly not normal. But they show what our tame mysterious entity is producing, and let us assess our **modelling assumptions** -- that the app ratings were characterised by just a mean and standard deviation.

In [19]:

```
def plot_samples(apps, mus, sigmas, n_samples):
    fig = plt.figure(figsize=(12,3))

    for i,(app,mu, sigma) in enumerate(zip(apps, mus, sigmas)):
        ax = fig.add_subplot(1,3,i+1)


        ax.scatter(app, np.random.uniform(0,1,app.shape),
                   marker='.', color='C1', label="Observations")


        # compute the PDF function for this setting of mu and sigma
        for m, s in zip(mu, sigma):
            samples = np.random.normal(mu, sigma, n_samples)
            ax.plot(samples, samples*0+0.5, marker='|', markersize=50, label="Sample

        # set labeling and axis
        ax.set_ylim(-0.2,1.2)
        ax.set_yticks([])
        ax.set_xlabel("Rating")

        ax.legend()

        plt.tight_layout()


plot_samples(apps, [[np.mean(app)] for app in apps], [[np.std(app)] for app in apps]
```



Maximum likelihood estimation: estimation by optimisation

What if we don't have estimators, ready built to estimate the parameters that we want? How can we do inference? How can we fit distribution parameters to observations?

In many cases, we can compute the **likelihood** of a an observation being generated by a specific underlying random distribution. This is the **likelihood** that we saw earlier. For a PDF, the likelihood of a value x is just the value of the PDF at x: $f_X(x)$. The likelihood is a function of the data, under the assumption of some particular parameters.

The likelihood of many *independent* observations is the product of the individual likelihoods, and the log-likelihood is the sum of the individual log-likelihoods.

$$\log \mathcal{L}(x_1, \ldots, x_n) = \sum_i \log f_X(x_i)$$

Imagine we have a distribution which we *don't* know any **estimators** for the parameters. How could we estimate what they might be, given some data? We could write all of our parameters as vector θ; for example a normal distribution would have $\theta = [\mu, \sigma]$.

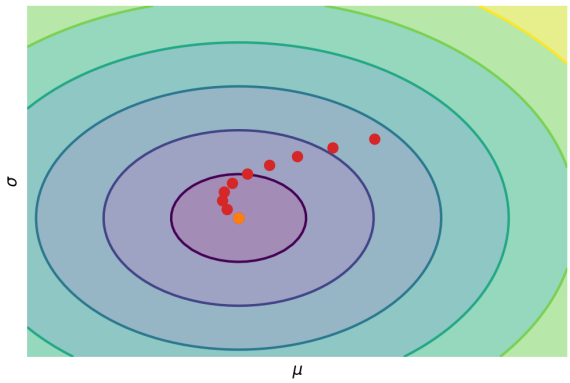


Image: Maximum likelihood estimation uses optimisation to maximise the likelihood function of the data and find the optimal value of the parameters given the data. It does not require a special estimator function; just a likelihood.

Optimisation solves all problems

Even though we don't have a fixed, closed form function to estimate the parameters, with a likelihood function we can apply optimisation to work out a parameter setting under which the data we *actually* observed was most likely. This corresponds to twiddling the knobs on our "mysterious entity" machine, until we find one that outputs the largest likelihood values when we feed in samples to it.

If the likelihood depends on some parameters of a distribution θ, then we write:

$$\mathcal{L}(\theta|x)$$

Then, we could define an **objective function**; to maximise the log-likelihood, or equivalently to minimise the negative log-likelihood.

$$\theta^* = \arg\min_\theta L(\theta)$$
$$L(\theta) = -\log \mathcal{L}(\theta|x_1, \dots, x_n) = -\sum_i \log f_X(x_i; \theta),$$

assuming our $f_X(x_i)$ can be written as $f(x; \theta)$ to represent the PDF of f with some specific choice of parameters given by θ.

Maximum likelihood estimation

This is *very* similar to the approximation objective function we saw before $||f(\mathbf{x}; \theta) - y||$, but we have $y = 0$ and we only have a scalar output from f so the norm is unnecessary. We already know how to solve this kind of problem; just optimise. This is called **maximum likelihood estimation** and is a general technique for determining parameters of a distribution which we don't know given some observations. It will find the **best** setting of parameters that would explain how the observations came to be.

If we're lucky, this will be differentiable and we can use gradient descent (or stochastic gradient descent -- note that the objective function is a sum of simple sub-objective functions). If we're not, we can fall back to less efficient optimisers. We don't need special estimators in this case, as long as we can evaluate the PDF $f(x; \theta)$

for any setting of parameters θ. *This works for a much wider class of probability distributions*.

Fitting a normal with MLE

We can for example look at the problem of estimating the mean and variance of a normal distribution from a set of (assumed to be independent) samples *without* using estimators; for example our app ratings. To do this, we need to be able to compute the likelihood for any given sample, and take the product (or rather sum of log likelihoods) for all of those samples.

This gives us our objective function. If we flip the sign, so that we minimise the negative log-likelihood, we will then search for the parameter vector that makes the data most likely.

For a univariate normal distribution, the parameters are just μ and σ, so $\theta = [\mu, \sigma]$.

In this case, of course, we *do* have estimators; but the procedure works just as well when we only have a likelihood function.

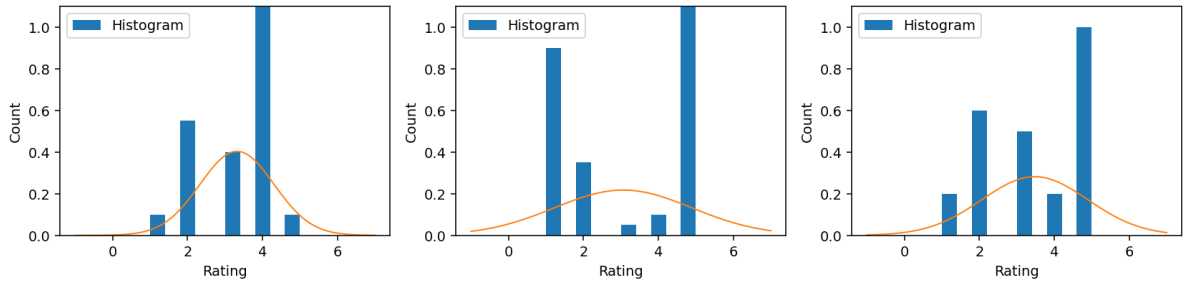
In [20]:

```
import scipy.stats
import scipy.optimize

# our data on the app ratings
apps = [app_A, app_B, app_C]

def loglik_norm(data, theta):
    # compute log-likelihood of observations
    # how likely is this data *under the assumption*
    # that these parameters are set
    return -np.sum(scipy.stats.norm(theta[0],
                                    np.abs(theta[1])).logpdf(data))

thetas = []
for app,name in zip(apps, "ABC"):
    print("MLE estimate for app {app}".format(app=name))


    # compute the log-likelihood for this specific dataset
    loglik = lambda theta, data=app: loglik_norm(data, theta)
    # MLE optimise
    theta = scipy.optimize.minimize(loglik,
                                    [1.0, 1.0],
                                    tol=1e-5).x
    thetas.append(theta)
    print("Sample statistics for app {app}".format(app=name))
    print(theta)
    # compare with using the direct estimates of these parameters
    print(np.mean(app), np.std(app))
    print()
```

```
MLE estimate for app A
Sample statistics for app A
[3.32000016 0.98873658]
3.32 0.9887365675446621

MLE estimate for app B
Sample statistics for app B
[3.05999985 1.83749831]
3.06 1.8374982993189408

MLE estimate for app C
Sample statistics for app C
[3.47999994 1.41760354]
3.48 1.417603611733548
```

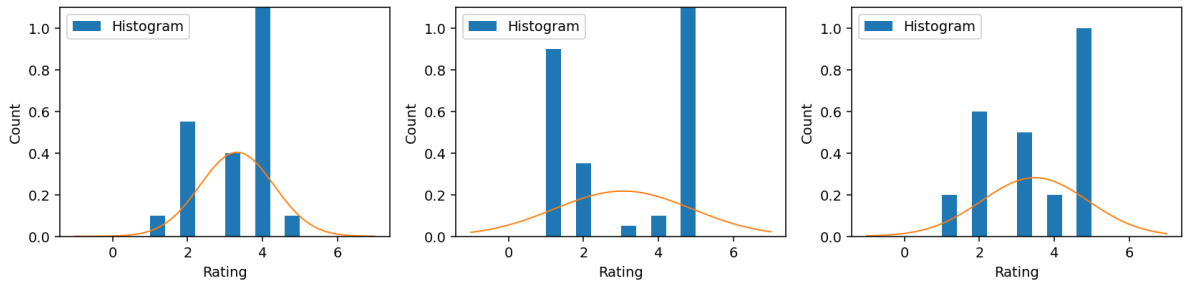
In [21]:

```
# use the sample mean, sample standard deviation
plot_fits(apps, [[np.mean(app)] for app in apps], [[np.std(app)] for app in apps])
```



In [22]:

```
# Use the MLE fit, which should look exactly the same (within some tiny tolerance)
plot_fits(apps, [[theta[0]] for theta in thetas], [[theta[1]] for theta in thetas])
```



A mixture model

But what if our model was more complicated that just a normal distribution? We could imagine that we model in some other way, perhaps that might be able to capture the fact that app B seems to have two "humps" on either side. One very simple model is a **mixture of Gaussians**, where we just say that we expect the PDF of the distribution we are trying to fit is a weighted combination (convex sum) of N different normal distributions ("components") $\mathcal{N}_i(\mu_i, \sigma_i)$, each with its *own* μ_i, σ_i, and with a weighting factor λ_i that says how important this "component" is, where $\sum_i \lambda_i = 1$. This lets us represent "humpy distributions.

This model lets us imagine that ratings might belong to one "cluster" or another. The placement and size of each cluster is given by the μ_i and σ_i for that component and λ_i gives an idea of how likely data is to fall into that cluster.

We can easily plot the PDF of this function; it's just:

$$f_X(x) = \sum_i \lambda_i n_X(x; \mu_i, \sigma_i),$$

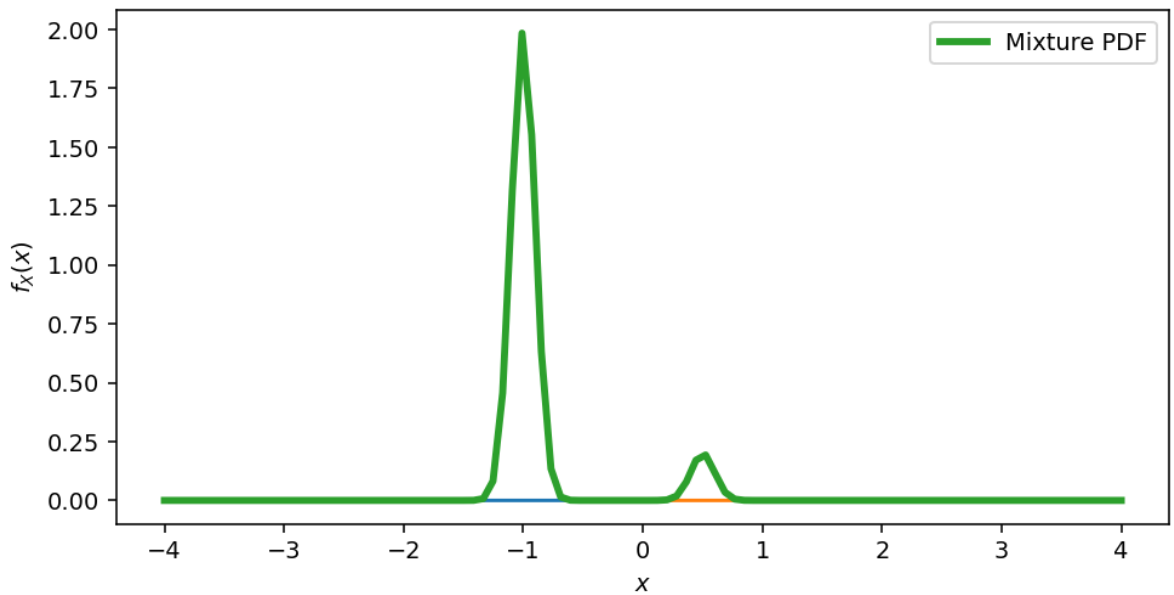
where $n_X(x; \mu, \sigma) = \frac{1}{Z} e^{\frac{(x-mu_i)^2}{2\sigma^2}}$ is the standard normal PDF function.

In [23]:

```
def pdf_mixture(mus, sigmas, lambdas):
    fig = plt.figure()
    ax = fig.add_subplot(1,1,1)
    xs = np.linspace(-4,4,100)
    pdf = np.zeros_like(xs)

    # compute the PDF of each component, weighted by
    # the corresponding lambda
    for mu, sigma, l in zip(mus, sigmas, lambdas):
        # sum of pdfs
        sub_pdf = scipy.stats.norm(mu, sigma).pdf(xs) * l
        pdf += sub_pdf
        ax.plot(xs, sub_pdf)

    # plot the sum PDF
    ax.plot(xs, pdf, lw=3, label="Mixture PDF")
    ax.legend()
    ax.set_xlabel("$x$")
    ax.set_ylabel("$f_X(x)$")
```

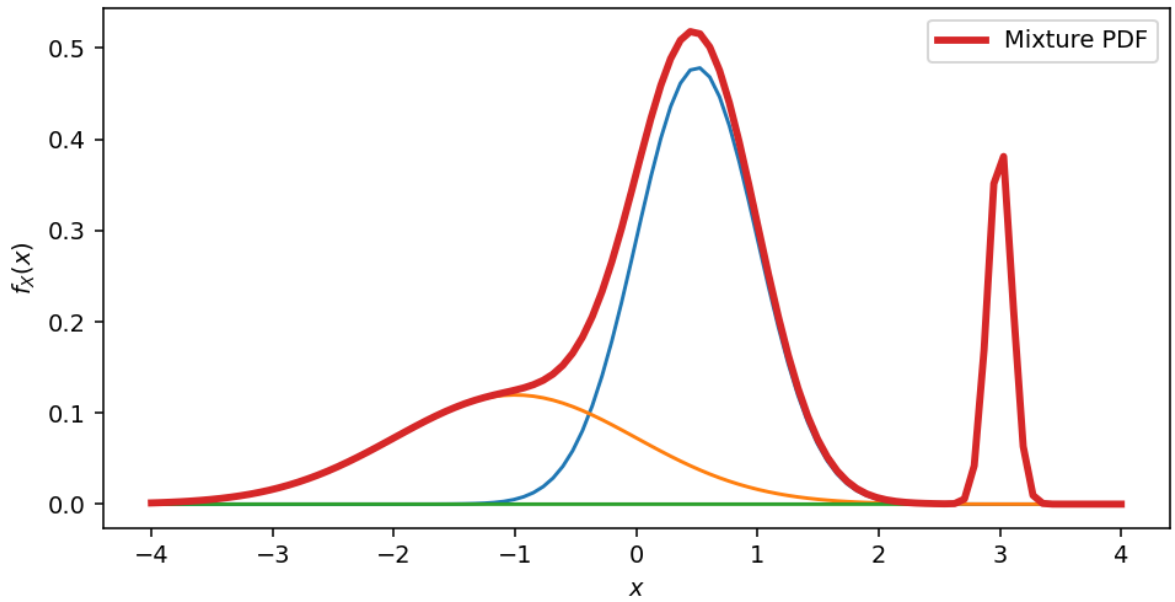
In [24]:

```
# N = 2
pdf_mixture([0.5, -1], [0.1, 0.1], [0.05, 0.5])
```



In [25]:

```
# N = 3
pdf_mixture([0.5, -1, 3],
            [0.5, 1.0, 0.1],
            [0.6, 0.3, 0.1])
```



Fitting mixtures

This is a much more plausible model of our app ratings, and might be a much better model. But how do we fit it? Even if we fix N in advance, we definitely don't have any direct estimators that can estimate the mean and standard deviation (and weighting) of a sum of normal PDFs. This simply isn't something we know how to do.

But the (log) likelihood is trivial to write in code. For each observation x, we just compute the sum of the weighted PDFs for each component, and the result is likelihood for that observation. This is a function of the data $\mathcal{L}(\theta|x)$, and our parameter vector is $\theta = [\mu_1, \sigma_1, \lambda_1, \mu_2, \sigma_2, \lambda_2, \ldots]$.

In [26]:

```
def pdf_mixture(xs, mus, sigmas, lambdas):
    return sum([scipy.stats.norm(mu, sigma).
                pdf(xs) * l
               for mu,sigma,l in
                zip(mus, sigmas, lambdas)])

# log likelihood of the mixture PDF
def llik_pdf_mixture(xs, mus, sigmas, lambdas):
    return np.sum(np.log(pdf_mixture(xs, mus, sigmas, lambdas)))
```

This means we can also fit it with maximum likelihood. We do not require an explicit estimator for the parameters; we can just optimise. In this case, we fix N to 2, and then find the parameters that best fit the data. This is a point in the vector space $\mathbb{R}^6$; we need three parameters $\mu_i, \sigma_i, \lambda_i$ for each component i.

In [27]:

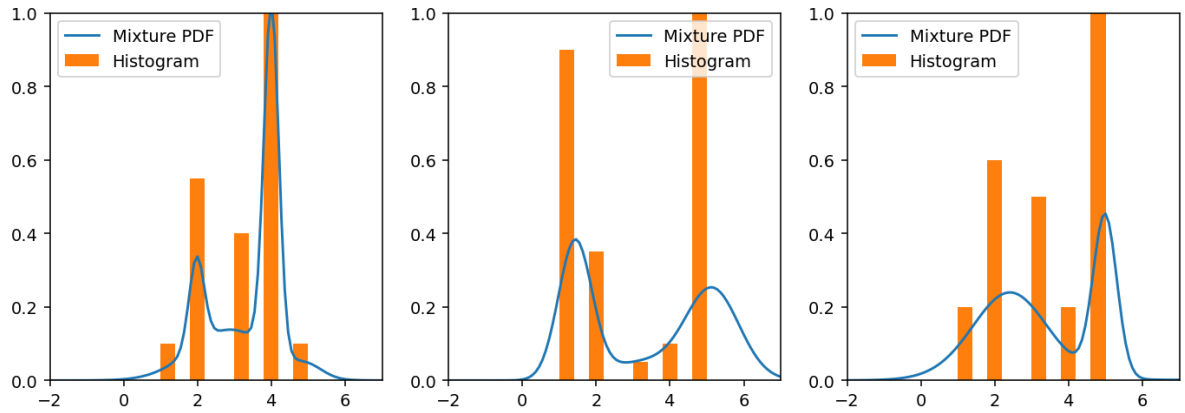
```
# our data on the app ratings
apps = [app_A, app_B, app_C]

def theta_to_params(theta):
    # remap our vector into three parameters
    # forcing lambdas to add to one
    # and sigma to be positive, and not
    # ridiculously small
    mus, sigmas, lambdas = theta.reshape(-1,3).T
    # force lambda to normalise to 1 (and be positive)
    lambdas = np.abs(lambdas)
    lambdas = lambdas / np.sum(lambdas)
    # force sigma not to collapse to 0
    # or be negative
    return mus, np.exp(sigmas)+0.2, lambdas

def loglik_mixture(data, theta):
    # compute log-likelihood of observations
    # how likely is this data *under the assumption*
    # that these parameters are set
    mus, sigmas, lambdas = theta_to_params(theta)
    r = -llik_pdf_mixture(data, mus, sigmas, lambdas)

    return r
```

In [28]:

```
# we can even do this with differnt numbers of components
# without changing the rest of the code
init_theta = [1.0, 1, 0.25,
              5.0, 1, 0.25,
              2.5, 2, 0.25,
              8.5, 2, 0.25],
```

In [29]:

```
###############
fig = plt.figure(figsize=(12,4))

# iniital guess for parameters
#            mu log(sigma) lambda
#init_theta =[1.0,   1,      0.5,
#            5.0,    1,      0.5]

for i,(app,name) in enumerate(zip(apps, "ABC")):


    loglik = lambda x, y=app: loglik_mixture(y, x)

    # assume we have 2 components, so we must specify
    # an initial guess for mu, sigma and lambda
    # for each of these; this is six numbers
    # MLE optimise
    np.random.seed(2018)
    theta = scipy.optimize.minimize(loglik, init_theta,
                                    tol=1e-5,
                                   method='TNC').x


    mus, sigmas, lambdas = theta_to_params(theta)

    # plot the results
    ax = fig.add_subplot(1,3,i+1)
    xs = np.linspace(-2, 7, 100)
    # compute the PDF curve
    pdf = pdf_mixture(xs, mus, sigmas, lambdas)
    ax.plot(xs, pdf, label="Mixture PDF")
    ax.hist(app,  density=True, label="Histogram")
    ax.set_ylim(0,1)
    ax.set_xlim(-2,7)
    ax.legend()
```



Bayesian Inference

Bayesian inference involves thinking about the problem quite differently. Bayesians represent the *parameters* of the distribution they are estimating as random variables *themselves*, with distributions of their own.

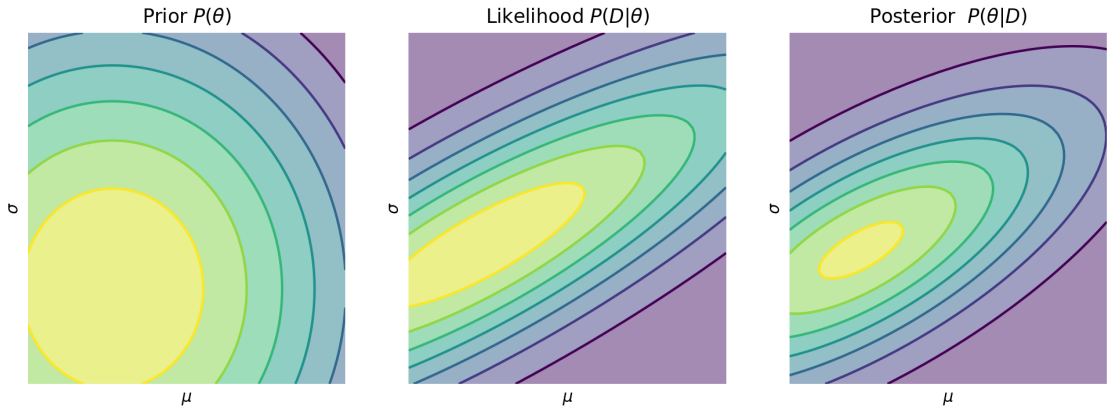
Prior distributions are defined over these parameters (e.g. we might believe that the mean app rating could be any value 1.0-5.0 with equal probability) and evidence arriving as updates is combined using Bayes' Rule to refine our belief about the distribution of the parameters. We again consider our distribution to be characterised by some parameter vector θ and we want to refine a distribution over possible θs.

We don't think about estimators, or their sampling distributions, and it doesn't make sense to talk about finding the best parameter setting; we can only have *beliefs* in parameter settings which must be represented probabilistically. We do not seek to find the most likely parameter setting (as in direct estimation or MLE), but to infer a distribution over possible parameter settings *compatible with the data*.

We talk about inferring a **posterior** distribution over the parameters, given some **prior** belief and some **evidence**. We assume that we have a **likelihood function** $P(D|\theta)$, and a prior over parameters $P(\theta)$ and we can then use Bayes Rule in the form:

$$P(\theta|D) = \frac{P(D|\theta)P(\theta)}{P(D)}$$

which gives us a new distribution over θ given some observations. Bayes' rule applies just as well to continuous distributions as to discrete ones, but computations in "closed form" (i.e. algebraically) are much harder.



This *can* be done in closed form to find $P(\theta|D)$ in certain cases, but the algebra is often complex and the model choices are limited; we will not discuss how to do this. When it is possible, it is, however, much more computationally efficient. Instead we will approach this from a computational perspective and find a way to draw *samples* from the posterior distribution $P(\theta|D)$.

Example

Remember, we are assuming that app ratings are generated by this function:

In [30]:

```
def app_samples(mu, sigma):
    return np.random.normal(mu, sigma)
```

We want to *infer a distribution* over μ and σ (**NOT** a distribution over the observations!). That is, we will treat the parameters themselves as random variables, with their own distributions, and use Bayesian reasoning (i.e. applying Bayes Rule) to infer a posterior distribution over the parameters given some prior, and some evidence

observed.

Parameters and samples

We have a collection of observations $D = x_1, x_2, \ldots$, which represent actual app ratings. We're not sure how much those ratings really tell us about the unseen population of potential users. We represent the distribution parameters as $\theta = [\mu, \sigma]$, and can talk about $P(\theta)$, a distribution over the parameter vectors.

Priors

Assume we have some prior belief θ, $P(\theta)$: for example this might be a very simple assumption that our prior is that μ and σ are uniformly distributed

- $\mu \sim U(1, 5)$
- $\sigma \sim U(0, 10)$

This gives us a form for $P(\theta)$. We could write a way of sampling from this distribution in code, and a way of evaluating the prior probability of any given parameter setting. As we will see later, we will need both of these functions to do Bayesian inference.

In [31]:

```
def sample_prior():
    return [np.random.uniform(1,5),
            np.random.uniform(0,3)]

def log_prior(theta):
    mu, sigma = theta
    # p(x) = 1/x if x ~ U(0,x), and 0 otherwise
    return (scipy.stats.uniform(1.0, 4.0).logpdf(mu) +
            scipy.stats.uniform(0.0, 3.0).logpdf(sigma))
```

In [32]:

```
log_prior([1.5, 0.3])
```

Out[32]:

$$-2.484906649788$$

Likelihood

We need to be able to define a **likelihood function**. This is a function of data given some parameter setting, and in this case it is the same as the likelihood function used for MLE: the likelihood of one sample is just the normal PDF evaluated at that point, and the likelihood of all samples is the product of these likelihoods.

In [33]:

```
def log_likelihood(samples, theta):
    # for a *known* mu, theta
    # compute the log PDF at each sample, and sum them
    return np.sum(scipy.stats.norm(theta[0],
                                   theta[1]).logpdf(samples))
```

Note carefully: in many cases we can only evaluate this likelihood function directly for a *specific* setting of θ; but we have a **distribution** over θ to deal with when doing Bayesian inference.

Inference

How can we compute the posterior distribution $P(\theta|D)$? We won't discuss how to find this in closed form (as a function) -- this is sometimes possible, but mathematically involved because we need to deal with distributions over θ -- but rather how to draw samples from this posterior, given a prior and a likelihood and some observations.

There is a huge literature on how to solve this problem, which has a few nasty parts:

- $P(D|\theta)$ needs to be computed for a **distribution** over θ, not just some numbers. It's no good to just compute the probability for one specific θ; we have to work with distribution functions.
- $P(D) = \int_\theta P(D|\theta)P(\theta)$ which is likely intractable.

Making it tractable

There are lots of ways this can be simplified to make it possible to solve. We are going to use two:

Samples will do

We often can't compute $P(\theta|D)$ because we don't know how to do operations on products of functions. But it's often trivial for *specific, concrete* values of θ. For example, for a given fixed θ we can compute both the likelihood and the prior of that specific example.

This leads us to the idea of **drawing samples** from the posterior distribution $P(\theta|D)$, instead of trying to compute the distribution exactly.

Relative probability only

We can make a simplifying assumption: we only care about the *relative* probability of different parameter settings with the data that we actually have, D. That is we have

$$P(\theta|D) \propto P(D|\theta)P(\theta)$$

and ignore the fact that this is the posterior scaled by some unknown constant $Z = \frac{1}{P(D)}$. This only makes sense because we are only considering one model with one set of data in this example.

Markov Chain Monte Carlo

We can implement a procedure to sample from the (relative) posterior distribution via a very simple modification of the *simulated annealing* algorithm.

This defines a random walk through the space of parameter settings, proposing small random tweaks to the parameter settings, and accepting "jumps" if they make the estimate more likely, or with a probability proportional to the change in $P(D|\theta)P(\theta)$ if not. The advantage of this approach is that we can work with *definite samples* from θ and we don't have to do any tricky integrals. This approach is called **Markov Chain Monte Carlo**

All we require is a way of evaluating $P(\theta)$ (prior) and $P(D|\theta)$ (likelihood) for any specific θ.

MCMC in practice: sampling issues

We will use Markov Chain Monte Carlo to solve the Bayesian inference problem. The **great thing** about MCMC approaches is that you can basically write down your model and then run inference directly. There is no need to derive complex approximations, or to restrict ourselves to limited models for which we can compute answers analytically. Essentially, no maths by hand; everything is done algorithmically.

MCMC allows us to draw samples from any distribution $P(X = x)$ *that we can't sample from directly*. In particular, we will be able to sample from the posterior distribution over parameters.

The **bad thing** about MCMC approaches is that, even though it will do the "right thing" *asymptotically*, the choice of sampling strategy has a very large influence for the kind of sample runs that are practical to execute. Bayesian inference should depend only on the priors and the evidence observed; but MCMC approaches also depend on the sampling strategy used to approximate the posterior.

What distribution are we sampling from?

In the case of Bayesian inference $P(\theta|D) = \frac{P(D|\theta)P(\theta)}{P(D)} = \frac{P(D|\theta)P(\theta)}{\int_\theta P(D|\theta)P(\theta)}$.

- $P(\theta|D)$ is the posterior, the distribution over the parameters θ given the data (observations) D, using:
- the likelihood $P(D|\theta)$,
- prior $P(\theta)$ and
- evidence $P(D)$.

In other words, what is the distribution over the parameters given the observations and the prior? If we assume, as above, that we don't care about $P(D)$, because we are only comparing different possible values of θ then we can draw samples from a distribution proportional to $P(D|\theta)P(\theta)$.

Metropolis-Hastings

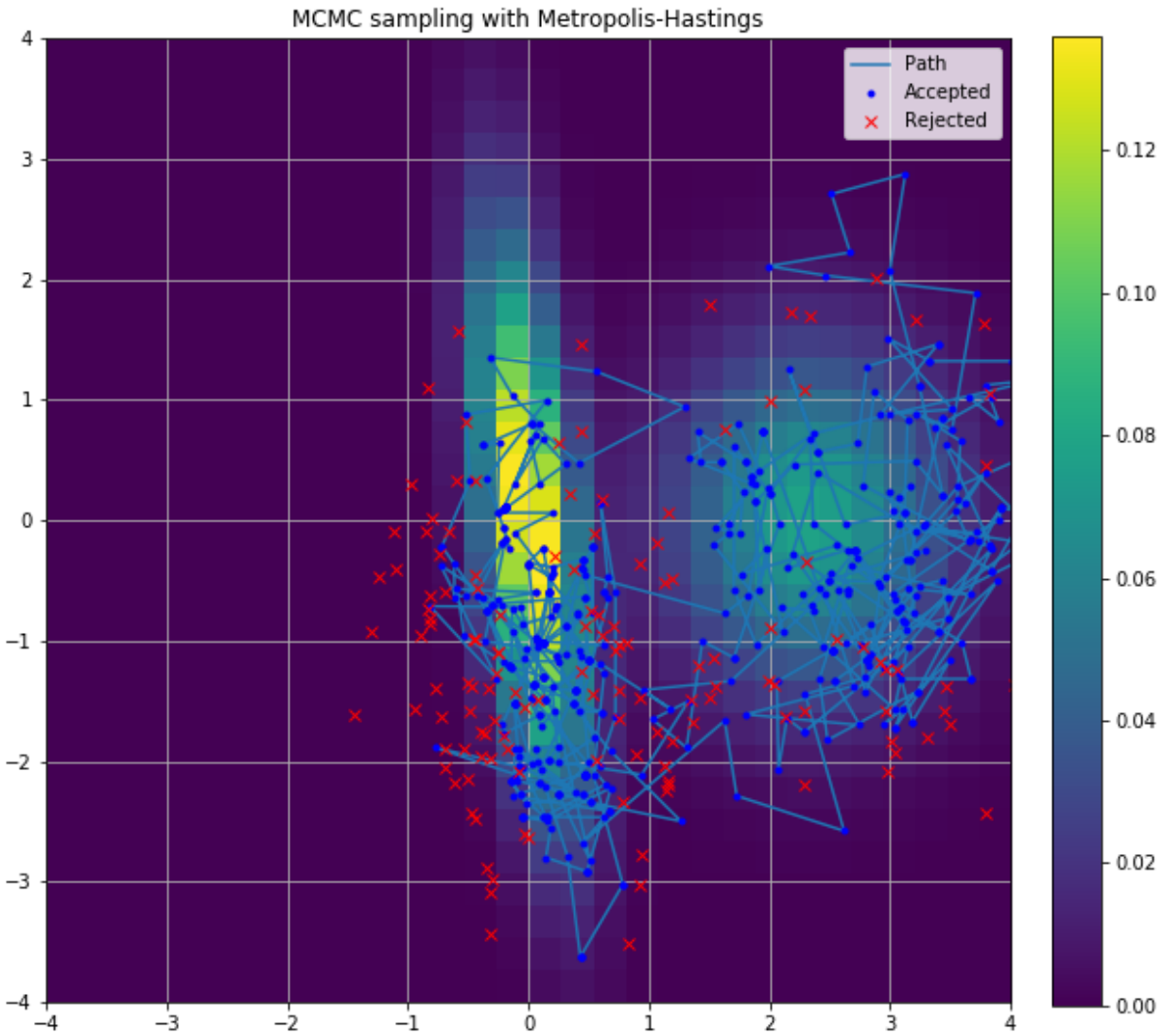
Metropolis-Hastings (or just plain Metropolis) is a wonderfully elegant and relatively effective way of doing this MCMC algorithm, and is able to work in high-dimensional spaces (i.e. when θ is complicated).

Metropolis sampling uses a simple auxiliary distribution called the **proposal distribution** $Q(\theta'|\theta)$ to help draw samples from an intractable posterior distribution $P(\theta|D)$. This is analogous to what we called the **neighbourhood function** in the optimisation section.

Metropolis-Hastings uses this to **wander around** in the distribution space, accepting jumps to new positions using $Q(\theta'|\theta)$ to randomly sample the space of $P(\theta|D)$. This random walk (a **Markov chain**, because we make a random jump conditioned only on where we currently are) is a the "Markov Chain" bit of "Markov Chain Monte Carlo".

This is just like the simulated annealing algorithm, except now there is a function $f_X(\theta)$ which makes some steps more likely than others instead of a likelihood function. We just take our current position θ, and propose a new position θ', that is a random sample drawn from $Q(\theta'|\theta)$. Often this is something very simple like a normal distribution with mean x and some preset σ: $Q(\theta'|\theta) = \mathcal{N}(\theta, \sigma')$

We can show a simple demo of this, drawing samples from a tricky 2D probability distribution.



Trace

The history of accepted samples of an MCMC process is called the **trace**. We can estimate model parameters by looking at the histogram of the **trace**, for example. The trace is the sequence of samples $[x^{(1)}, x^{(2)}, x^{(3)}, \ldots x^{(n)}]$, (approximately) drawn from the **posterior** distribution $P(\theta|D)$ via MCMC.

Applying MCMC

Let's apply this to our app rating problem. We want to estimate the distribution over the parameters μ and σ, given some observations (evidence/data) $x_1, x_2, x_3, \ldots$ drawn from a distribution we assume is $\mathcal{N}(\mu, \sigma)$.

We already have a prior function `prior_pdf` and a likelihood function `likelihood` . So we can just do this, using the same procedure as above, but this time starting the random process in various random initial conditions drawn from the prior to make sure we don't get stuck in some bad part of the space.

In [34]:

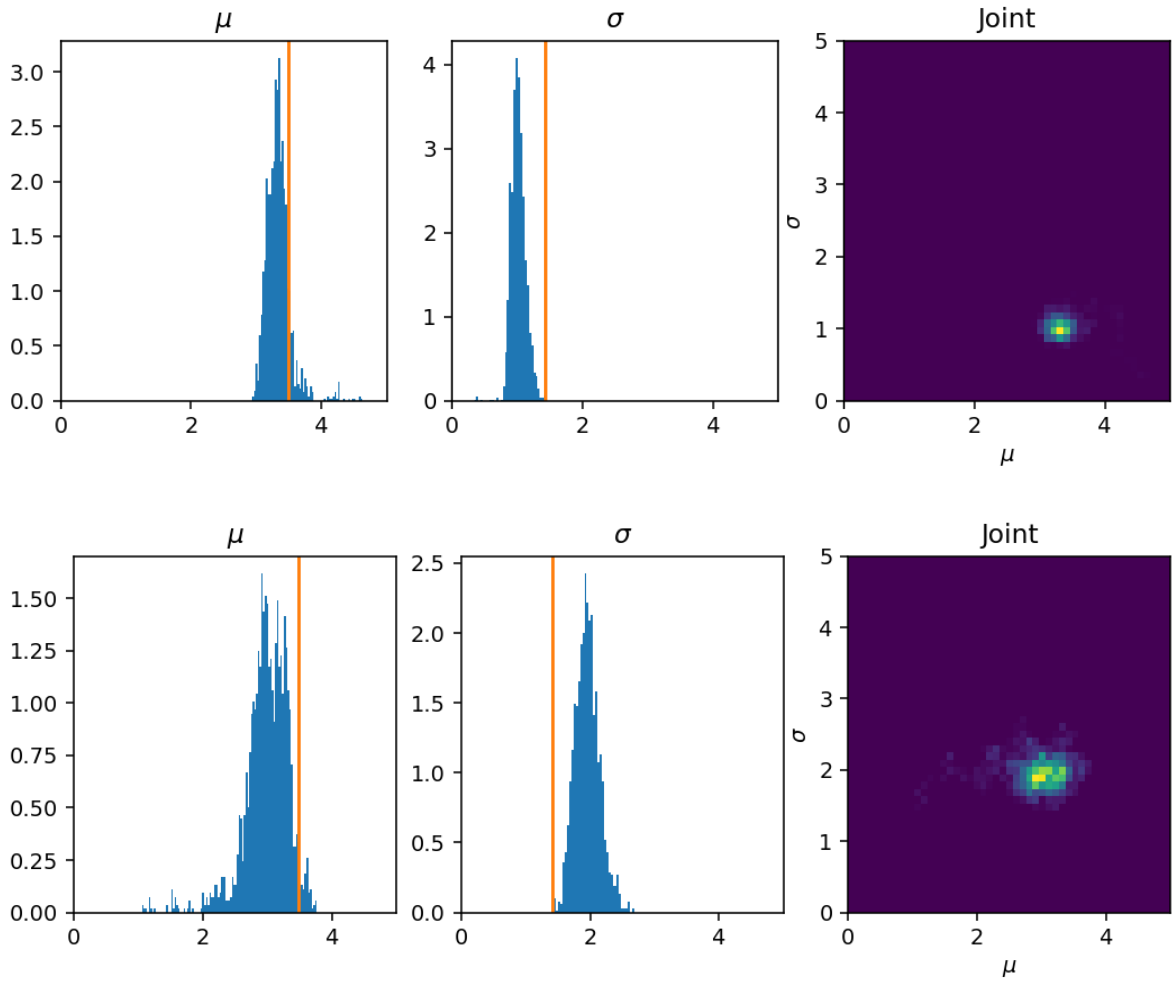
```
from metropolis import log_metropolis

traces = []
for app in apps:
    init_guess = sample_prior()
    log_fx = lambda theta, data=app:log_prior(theta) + log_likelihood(data, theta)
    q = lambda x: x + np.random.normal(0, 0.08, size=2) # a small random step
    # apply our algorithm
    accepted = log_metropolis(log_fx, q, init_guess, 2000)
    traces.append(accepted)
```

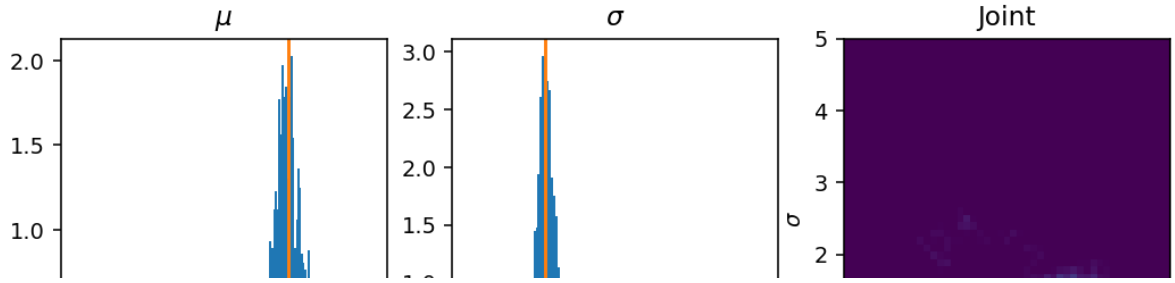
```
Acceptance ratio: 0.697
Acceptance ratio: 0.810
Acceptance ratio: 0.778
```

We can visualise the trace of this MCMC inference process; for example via histograms.

In [35]:

```
for trace in traces:
    # plot the results
    fig = plt.figure(figsize=(9,3))
    ax = fig.add_subplot(1,3,1)
    ax.hist(trace[:,0], density=True, bins=np.linspace(1.0, 5.0, 150), label="Poster
    ax.axvline(np.mean(app), color='C1', label="Direct estimate")
    ax.set_xlim(0,5)
    ax.set_title("$\mu$")

    ax = fig.add_subplot(1,3,2)
    ax.axvline(np.std(app), color='C1', label="Direct estimate")
    ax.hist(trace[:,1], density=True, bins=np.linspace(0.0, 5.0, 150), label="Poster
    ax.set_xlim(0,5)
    ax.set_title("$\sigma$")

    ax = fig.add_subplot(1,3,3)

    ax.hist2d(trace[:,0], trace[:,1], bins=np.linspace(0.0, 5.0, 50))
    ax.set_xlabel("$\mu$")
    ax.set_ylabel("$\sigma$")
    ax.set_title("Joint")
```





Predictive posterior: sampling from the model

What we have plotted is samples from the **posterior distribution of the model parameters**; i.e. the values we expect the model parameters to take on given the data we observed and our prior.

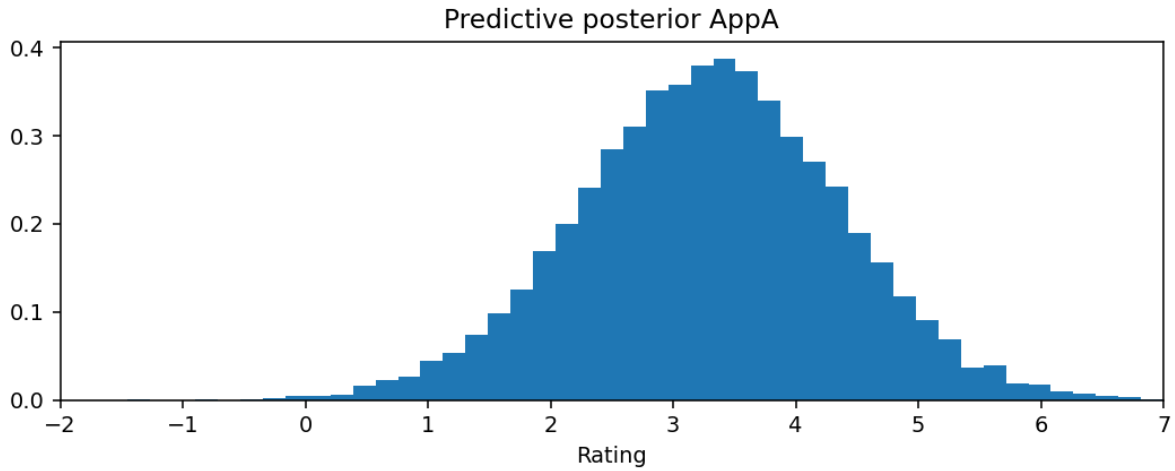
The **predictive posterior** is the *distribution over observations* we would expect to see; predictions of future samples. This means drawing samples from the model, while integrating over parameters from the posterior. By sampling from the predictive posterior, we are generating new synthetic data that should have the same statistical properties as the data (if our model is good).

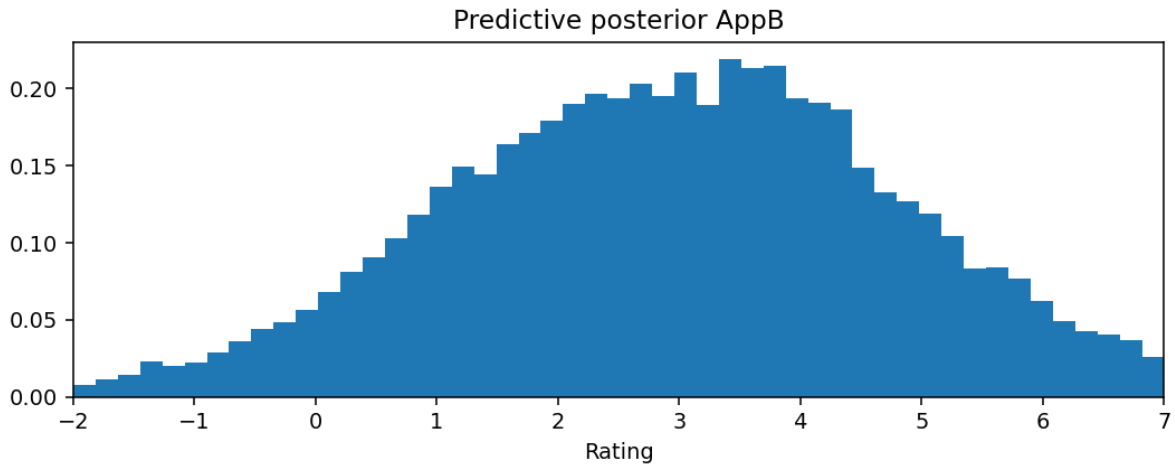
We can do this with a two step, nested process:

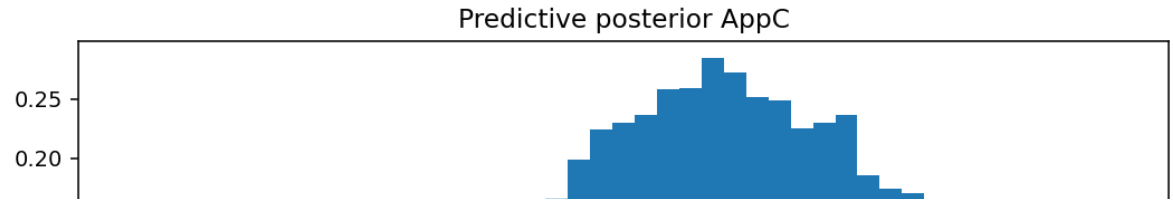
- for n repetitions
- draw samples from our posterior distribution over parameters to give us a concrete distribution
    - for m repetitions
        - draw samples from this concrete distribution

In [36]:

```
import random
n = 20
m = 500

for trace, label in zip(traces, "ABC"):
    fig = plt.figure(figsize=(9,3))
    ax = fig.add_subplot(1,1,1)

    predictives = []
    for i in range(n):
        # choose a random parameter setting
        mu, theta = random.choice(trace)
        # sample from this distribution m times
        predictive = np.random.normal(mu, theta, m)
        predictives.append(predictive)

    ax.hist(np.concatenate(predictives), density=True, bins=np.linspace(-2, 7, 50))
    ax.set_xlim(-2, 7)
    ax.set_title("Predictive posterior App"+label)
    ax.set_xlabel("Rating")
```







Inference: a review

Linear regression

Linear regression is the fitting of a line to observed data. It assumes the mysterious entity generates data where one of the observed variables is scaled and shifted version of another observed variable, corrupted by some noise; a linear relationship. It is a very helpful lens through which different approaches to data modelling can be seen; it is pretty much the simplest useful model of data with relationships, and the techniques we use easily generalise from linear models to more powerful representations.

The problems is to estimate what that scaling and shifting is. In a simple 2D case, this is the gradient m and offset c in the equation $y = mx + c$. It can be directly generalised to higher dimensions to find A and $\mathbf{b}$ in

$$\mathbf{y} = A\mathbf{x} + \mathbf{b},$$

but we'll use the simple "high school" $y = mx + c$ case for simplicity.

We assume that we will fit a line to *noisy* data. That is the process that we assume that is generating the data is

$$y = mx + c + \epsilon,$$

where ϵ is some noise term. We have to make assumptions about the distribution of ϵ in order to make inferences about the parameters.

One simple assumption is that

$$\epsilon \sim \mathcal{N}(0, \sigma^2),$$

i.e. that we have normally distributed variations in our measurements. So our full equation is:

$$y = mx + c + \mathcal{N}(0, \sigma^2),$$

or equivalently, putting the $mx + c$ as the mean of the normal distribution:

$$y \sim \mathcal{N}(mx + c, \sigma^2)$$

Note that we assume that y is a random variable, x is known, and that m, c, σ are parameters that we wish to infer from a collection of observations:

$$[(x_1, y_1),$$
$$(x_2, y_2),$$
$$\dots,$$
$$(x_n, y_n)]$$

In code, we could write down what we *assume* is generating data, our tame mysterious entity:

In [37]:

```
def model(x, theta):
    m, c, sigma = theta
    y =  np.random.normal(x * m + c, sigma)
    return y
```

Our problem is: given just the inputs x and return values y, what are the values of the *other* argument θ.

Linear regression via direct optimisation

We saw how this problem could be solved as a **function approximation** problem using optimisation. We can write an objective function:

$$L(\theta) = \|f(x;\theta) - y\|,$$

where $\theta = [m, c]$ and

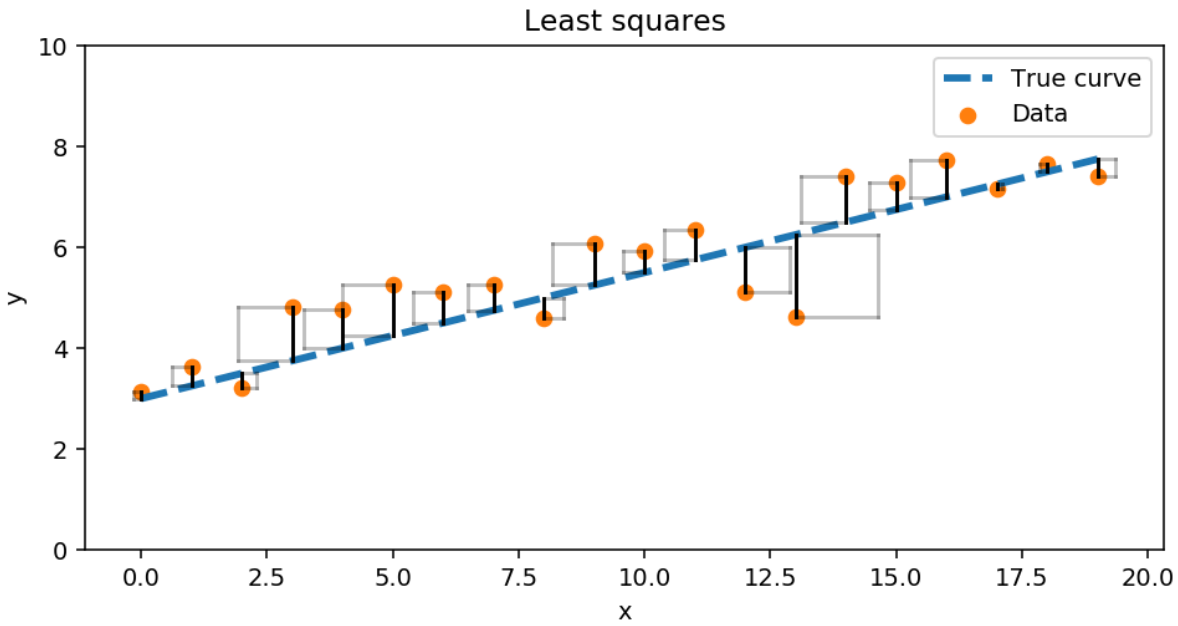
$$f(x;\theta) = \theta_0 x + \theta_1$$

.

If we choose the squared Euclidean norm, then we have, for the simple $y = mx + c$ case :

$$L(\theta) = \|f(x;\theta) - y\|$$
$$L(\theta) = \|\theta_0 x + \theta_1 - y\|_2^2 = (\theta_0 x + \theta_1 - y)^2,$$

which we can easily minimise, e.g. by gradient descent, since computing $\nabla L(\theta)$ turns out to be easy. This is **ordinary linear least-squares**.



Linear least squares tried to make the size of the squares nestled between the line and data points as small as possible

In fact, we can find a closed form solution to this problem, without doing any iterative optimisation. This is because we have an **estimator** that gives us an estimate of the parameters of the line fit directly from observations. We can derive this, for example, by setting $\nabla L(\theta) = 0$ and solving directly (high-school optimisation).

Linear regression via maximum likelihood estimation

We could also consider this to be a problem of inference. We could explicitly assume that we are observing samples from a distribution whose parameters we wish to estimate. This is a **maximum likelihood approach**. This requires that we can write down the problem in terms of the distribution of random variables.

If we assume that "errors" are normally distributed values which are corrupting a perfect $y = mx + c$ relationship, we might have a model $Y \sim \mathcal{N}(mx + c, \sigma^2)$; Y has mean $mx + c$ and some standard deviation σ.

We can write this as a maximum likelihood problem (MLE), where we maximise $\mathcal{L}(\theta|x_1, y_1, x_2, y_2, \ldots, x_n, y_n)$. To avoid underflow, we work with the log of the likelihood and minimise the negative log-likelihood. The log-likelihood of independent samples x_i is given by:

$$\log \mathcal{L}(\theta|x_1, y_1, \ldots, x_n, y_n) = \log \prod_i f_Y(x_i, y_i) = \sum_i \log f_Y(x_i, y_i),$$

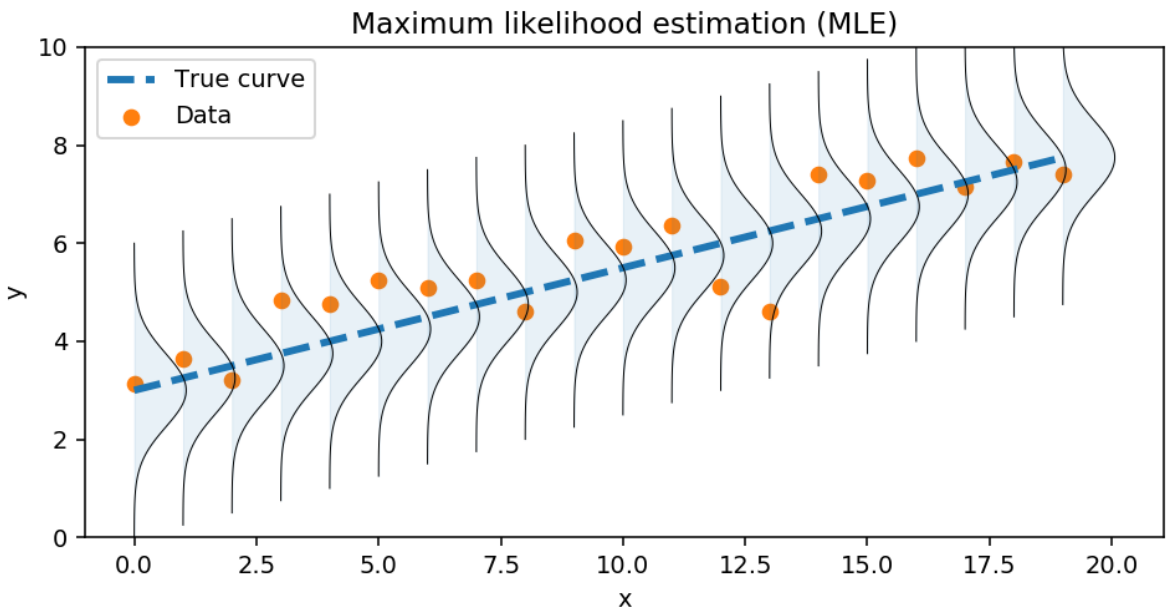
$$f_Y(x_i, y_i) = \frac{1}{Z} e^{-\frac{(y_i-\mu)^2}{2\sigma^2}}, \quad \mu = mx_i + c$$

We can then minimise the negative log-likelihood to find the "most likely" setting for $\theta = [m, c, \sigma]$, which (if we feel like writing out long equations in LaTeX), we could write as an objective function:

$$L(\theta) = -\sum_i \log \left[\frac{1}{Z} e^{-\frac{(y_i-\theta_0 x_i+\theta_1)^2}{2\theta_2^2}} \right],$$

In the case where we have normally distributed noise for linear regression, this turns out to be *exactly* equivalent to the direct optimisation with linear least-squares, although we will also find the standard deviation of the error σ in addition to m and c. This is **maximum likelihood linear regression**.

[Note: you definitely do not need to remember these equations or be able to derive them. You should understand the logic behind them, however].



Maximum likelihood estimation tried to find parameters of a line that made the observations likely

Bayesian linear regression

What if we wanted to know how sure our estimates of m and c (and σ) were? MLE will tell us the *most likely setting*, but it won't tell us the possible settings that are compatible with the data.

The Bayesian approach is to let the parameters themselves by random variables. We don't want to optimise. We don't want to find the most likely parameters. We instead want to derive a belief about the parameters as a probability distribution. This is what Bayesians do; they represent belief with probability.

So we can write $\theta = [m, c, \sigma]$ as a random variable, and try and infer the distribution over it. We can do this using Bayes' rule. Writing in the form (D=data, H=hypothesis; hypothesised parameter settings):

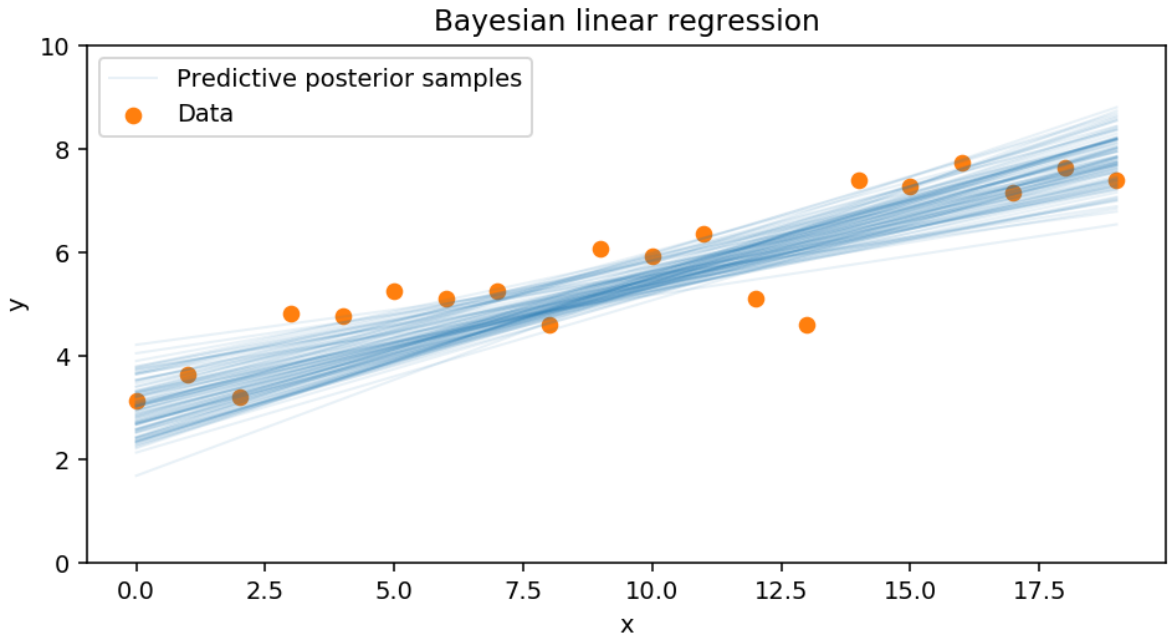
$$P(H|D) = \frac{P(D|H)P(H)}{P(D)}$$

Assuming our hypotheses H are parameterised by θ, then we want to know $P(\theta|D) = \frac{P(D|\theta)P(\theta)}{P(D)}$, where D stands for the data $[(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)]$. In linear regression θ can be seen as the hypothesis that the data was generated by a line with parameters specified by θ.

We need:

- a **prior** over the parameters $P(\theta)$. An initial belief about the possible gradient m, offset c and noise level σ, in the linear regression case.
- a way of calculating the **likelihood** $P(D|\theta)$.
- a way of combining these using Bayes Rule. In general this is impossible to compute exactly (in particular the $P(D)$ term is often intractable), but we could sample from it using **Markov Chain Monte Carlo**, for example.

This will give us samples from the posterior distribution of $P(\theta|D)$, so we can see how sure we should be about our beliefs about the parameters of the mysterious entity.



Bayesian regression tries to update a distribution over line parameters given evidence

Summary of terms

- **Parameters** variables that affect a random process
- **Prior** belief about parameters before observing
- **Observations** samples from a process we have seen
- **Posterior** belief about parameters after seeing observations
- **Predictive posterior** belief about observations we would see, given the posterior we have inferred.

Resources

- **Bayesian methods for Hackers** https://github.com/CamDavidsonPilon/Probabilistic-Programming-and-Bayesian-Methods-for-Hackers (https://github.com/CamDavidsonPilon/Probabilistic-Programming-and-Bayesian-Methods-for-Hackers) (a full "book" on Bayesian methods and inference)
- **MCMC for dummies** (http://twiecki.github.io/blog/2015/11/10/mcmc-sampling/)
- **Bayesian Linear Regression** (https://www.chrisstucchio.com/blog/2017/bayesian_linear_regression.html)