In [1]:
```
import torch as t
import matplotlib.pyplot as plt
%matplotlib inline
from ipywidgets import FloatSlider, IntSlider, interact, interact_m
from torch.distributions import Bernoulli
from torch.distributions import MultivariateNormal as MvNormal
```

Part 1: Likelihoods and modelling

First things first:

- I'm experimenting with doing the lectures in Jupyter Notebooks.
- Hopefully, it means we can connect theory and code more closely.
- As much as possible, I'll try to prepare the ground for later deep-leaning and machine learning courses.
- I'm going to use PyTorch, mainly for the distributions library.
- PyTorch's syntax is almost exactly the same as Numpy.
- And so you can't copy code for labs/courseworks (which must be done in Numpy).

What is a model?

Model == a structured approach for making predictions about the future.

For instance, imagine Alice tosses a coin 10 times and gets:

H T H T H T H T H H

Now she tosses the coin again. What happens?

One option is that the coin tosses are indeed fair and random, in which case next time we might get:

T H H H H T H T T T

or the coin might be biased, and we just happened to get an equal number of heads/tails. Next time, we might get:

T H H H H T H H H H

or Alice might have clever trick (https://www.youtube.com/watch?v=A-L7KOjyDrE (https://www.youtube.com/watch?v=A-L7KOjyDrE)), so that next time she can get exactly the same thing:

H T H T H T H T H H

(or whatever else she wants).

Simple example: A biased coin flip

Lets assume that Alice doesn't have a trick, so the coin tosses are independent and random, but that the coin might be biased. In that case, the coin-toss, x can be said to be Bernoulli distributed, with probability p,

$$P(x|p) = \text{Bernoulli}\,(x; p)$$

where,

$$\text{Bernoulli}\,(x; p) = \begin{cases} 1-p & \text{if } x = 0 \\ p & \text{if } x = 1 \end{cases}$$

In [2]:
```
bern = Bernoulli(probs=0.7)
print(bern.log_prob(0.).exp())
print(bern.log_prob(1.).exp())
```

```
tensor(0.3000)
tensor(0.7000)
```

Note: PyTorch only provides the log_prob method, because this is much more numerically stable when doing deep-learning. We therefore have to exponentiate to get back the probability.

Thus, given a probability, p, we can make a predictions about possible values of about Alice's future coin tosses:

In [3]:
```
def sample_bernoulli(p):
    return Bernoulli(probs=p).sample((10,)).to(dtype=t.int)
interact_manual(sample_bernoulli, p=FloatSlider(value=0.5, min=0, m
```

p 0.50

Run Interact

```
tensor([0, 1, 1, 1, 0, 0, 1, 0, 1, 0], dtype=torch.int32)
```

Key question: how do we find p?

Maximum likelihood fitting of a Bernoulli

Consider data:

T T T T H T T T T T H

In [4]:
```
xs = t.tensor([0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 1.])
```

The likelihood, i.e. the probability of all the data, treated as a function of the parameter, p, is

$$\mathrm{P}\left(\mathbf{x}|p\right) = \prod_{\lambda} \mathrm{P}\left(x_{\lambda}|p\right)$$

But these numbers rapidly shrink (and become too small for standard floating points). Instead, we work with the log-likelihood,

$$\mathcal{L}(p) = \sum_{\lambda} \log \mathrm{P}\left(x_{\lambda}|p\right)$$

Importantly, the log-likelihood is considered a function of the parameter(s), p, but not the data, which is considered fixed.

The goal is to find $\hat{p}$ which maximizes $\mathcal{L}(p)$,

$$\hat{p} = \arg\max_{p} \mathcal{L}(p)$$

Lets find $\hat{p}$! We can code-up the log-likelihood using,

In [5]:
```
def log_likelihood(p):
    return Bernoulli(probs=p).log_prob(xs).sum(-1, keepdim=True)

interact(log_likelihood, p=FloatSlider(min=0.01, max=0.99, step=0.0
```

p 0.01

```
tensor([-9.3108])
```

Lets plot the log-likelihood,

In [6]:
```
ps = t.linspace(0.01, 0.99, 99)[:, None]
print(ps.shape)
ps[:6, :]
```

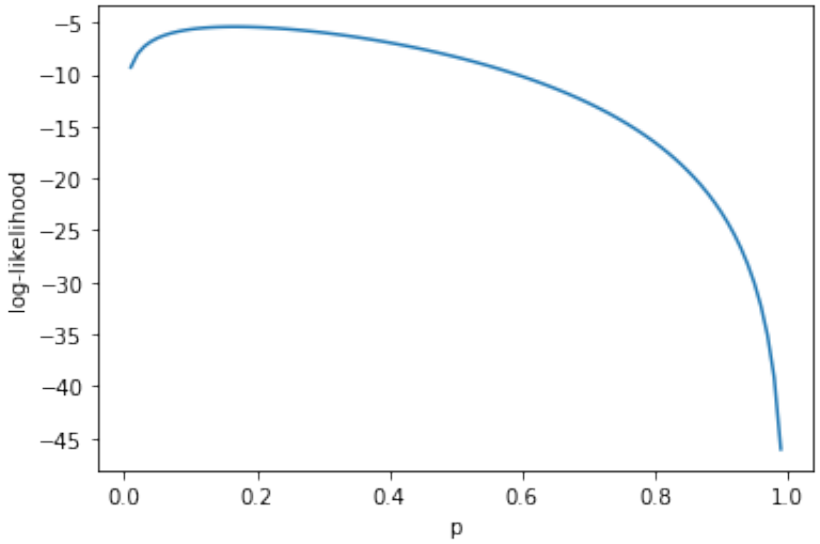
```
torch.Size([99, 1])
```

Out[6]:
```
tensor([[0.0100],
        [0.0200],
        [0.0300],
        [0.0400],
        [0.0500],
        [0.0600]])
```

In [7]:
```
lls = log_likelihood(ps)
lls.shape
```

Out[7]:
```
torch.Size([99, 1])
```

In [8]:
```
fig, ax = plt.subplots()
ax.plot(ps, lls)
ax.set_xlabel("p")
ax.set_ylabel("log-likelihood");
```



In simple cases like this, we could just use the above plot to find the value of p with the maximal log-likelihood.

But in more complicated cases with many more parameters, this is no longer possible.

Instead, the first approach is to try to find an analytic expression for the p with the maximal log-probability.

Remember,

$$\mathcal{L}(p) = \sum_\lambda \log \mathrm{P}\left(x_\lambda | p\right)$$

$$\mathcal{L}(p) = \left(\sum_\lambda x_\lambda\right) \log p + \left(N - \sum_\lambda x_\lambda\right) \log(1-p)$$

And solve for where the gradient is zero,

$$0 = \left.\frac{\partial \mathcal{L}(p)}{\partial p}\right|_{p=\hat{p}}$$

$$0 = \frac{\sum_\lambda x_\lambda}{\hat{p}} - \frac{N - \sum_\lambda x_\lambda}{1-\hat{p}}$$

$$\frac{\sum_\lambda x_\lambda}{\hat{p}} = \frac{N - \sum_\lambda x_\lambda}{1-\hat{p}}$$

$$(1-\hat{p}) \sum_\lambda x_\lambda = \hat{p}\left(N - \sum_\lambda x_\lambda\right)$$

$$\sum_\lambda x_\lambda = \hat{p} N$$

$$\hat{p} = \tfrac{1}{N} \sum_i x_i$$

The maximum-likelihood probability is the empirical mean of the data points. This is sensible, but not at all obvious: we needed to go through the derivation!

Using this derivation, we can write a function that automatically fits a Bernoulli distribution,

In [9]:
```
def fit_bernoulli(xs):
    p = xs.sum() / xs.shape[-1]
    return Bernoulli(probs=p)

fitted_bernoulli = fit_bernoulli(xs)
fitted_bernoulli
```

Out[9]:
```
Bernoulli(probs: 0.1666666716337204)
```

such that samples from the fitted Bernoulli look like the data,

In [10]:
```
xs
```

Out[10]:
```
tensor([0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 1.])
```

In [11]:
```
fitted_bernoulli.sample((12,))
```

Out[11]:
```
tensor([1., 0., 0., 0., 0., 0., 0., 1., 1., 0., 0., 0.])
```

Bayesian fitting of a Bernoulli

Maximum likelihood works well when we have a reasonable number of datapoints.

But what about when we have very little data (e.g. we might have many biased coins, and be able to toss each one a few times).

In particular, lets say we have a population of coins, whose probabilities are drawn from a uniform distribution,

$$\mathrm{P}\left(p\right) = \mathrm{Uniform}(p; 0, 1)$$

we take out one coin, toss it twice, and get two zeros,

In [12]:
```
xs = t.tensor([0., 0.])
```

What can we say about p?

Maximum likelihood would tell us that $\hat{p} = 0$,

In [13]:
```
fit_bernoulli(xs)
```

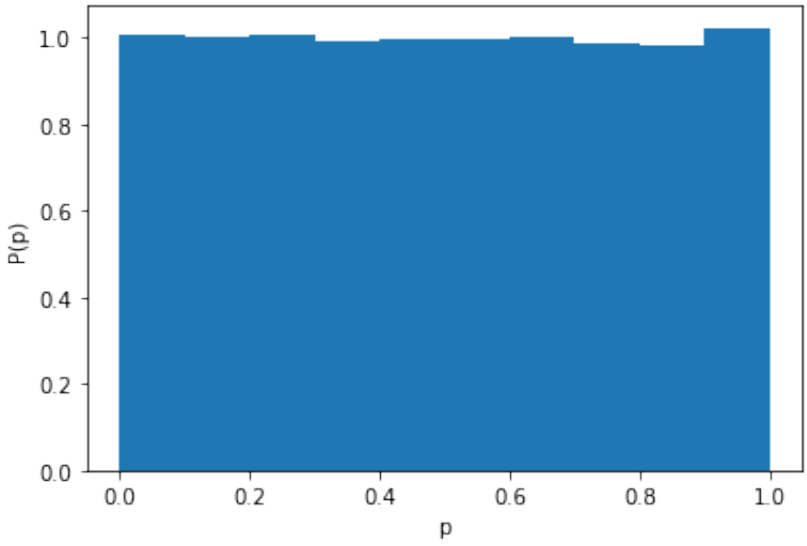
Out[13]:
```
Bernoulli(probs: 0.0)
```

Which is a bit strange, because p could have been quite large, but coincidentally, our two coin-tosses happened to be zeros.

To understand this in a bit more depth, we can run a simulation.

We start by drawing a large number of p's, from the uniform prior,

In [14]:
```
N = 10**5
ps = t.rand(N)
fig, ax = plt.subplots()
ax.set_xlabel("p")
ax.set_ylabel("P(p)")
ax.hist(ps, density=True);
```



Note: the values on the x-axis are probability *densities*, not probabilities.

The probability of being in any given bin is:

$$\int_{p_0}^{p_0+\delta} dp\ \mathrm{P}\left(p\right) \approx \delta \mathrm{P}\left(p\right)$$

i.e. the bin-width times the probability density.

Here, we have 10 bins, with probability density 1, and bin width $\delta = 1/20$.

The probability of being in any 1 bin is therefore $\mathrm{P}\left(p\right) \times \delta = 1 \times 1/10 = 1/10$, and adding up the 10 bins gives us 1.

Then, for each of these p's, we toss a couple of coins,

In [15]:
```
xs = Bernoulli(ps).sample((2,))
xs
xs.shape
```

Out[15]:
```
torch.Size([2, 100000])
```

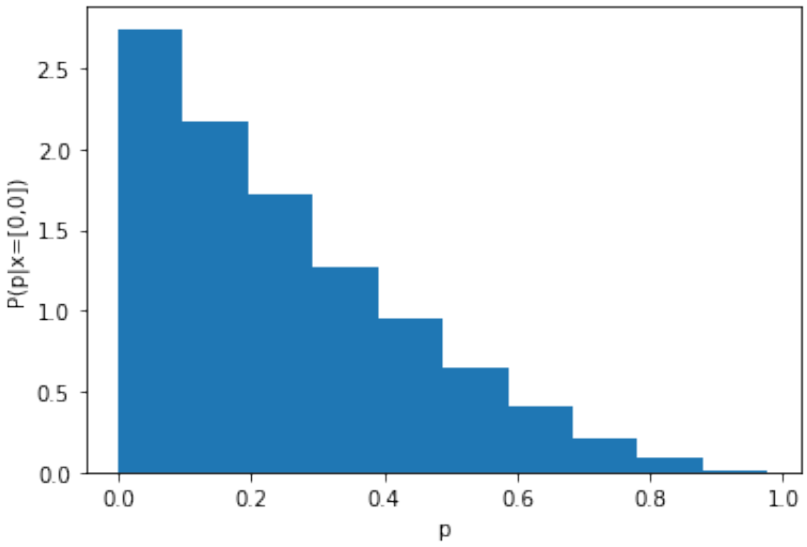
Then, we filter out only those values of p which actually gave us two zeros,

In [16]:
```
all_zeros = (xs==0.).all(0)
print(all_zeros)
print(all_zeros.sum())
ps_all_zeros = ps[all_zeros]
print(ps_all_zeros.shape)
```

```
tensor([False,  True, False,  ...,  True, False,  True])
tensor(33303)
torch.Size([33303])
```

Now, we can plot those p's

In [17]:
```
fig, ax = plt.subplots()
ax.set_xlabel("p")
ax.set_ylabel("P(p|x=[0,0])")
ax.hist(ps_all_zeros, density=True);
```



This makes sense: we expect the probability to be small, because when we flipped the coin, we observed two zeros.

But the probability could still be large, with the two zeros we observed being coincidences.

It turns out this idea (sampling until we get something very close to the data, then looking at the corresponding latents, here the probability, p), is a real algorithm, called "approximate Bayesian computation" (ABC).

But its a terrible idea: don't do it in practice unless you really have to. For any non-trivial dataset, you will be waiting a very, very long time before random sampling produces something "sufficient close".

Instead, can we do exact Bayesian inference to directly compute the distribution $\mathrm{P}\left(p|\mathbf{x}\right)$.

The answer is yes!

In particular, the law of joint probability tells us that we can write the joint, $\mathrm{P}\left(\mathbf{x}, p\right)$ in two equivalent forms,

$$\mathrm{P}\left(\mathbf{x}, p\right) = \mathrm{P}\left(\mathbf{x}|p\right)\mathrm{P}\left(p\right) = \mathrm{P}\left(p|\mathbf{x}\right)\mathrm{P}\left(\mathbf{x}\right)$$

The first form $\mathrm{P}\left(\mathbf{x}|p\right)\mathrm{P}\left(p\right)$, is the standard one, and we can readily compute it given the expressions above. The second form, $\mathrm{P}\left(p|\mathbf{x}\right)\mathrm{P}\left(\mathbf{x}\right)$ is a bit more unusual: it isn't immediately obvious how we can compute these terms.

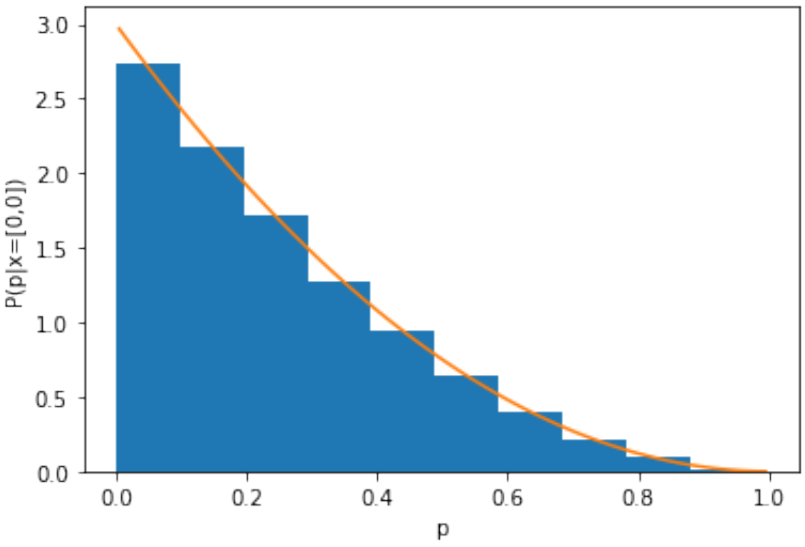
Nonetheless, we can rearrange to compute the term we're interested in,

$$\mathrm{P}\left(p|\mathbf{x}\right) = \frac{\mathrm{P}\left(\mathbf{x}|p\right)\mathrm{P}\left(p\right)}{\mathrm{P}\left(\mathbf{x}\right)} \propto \mathrm{P}\left(\mathbf{x}|p\right)\mathrm{P}\left(p\right)$$
$$\log \mathrm{P}\left(p|\mathbf{x}\right) = \mathcal{L}\left(p\right) + \log \mathrm{P}\left(p\right) + \mathrm{const}$$

The proportionality arises because we take data, $\mathbf{x}$, to be fixed, so we only care about parameter, p dependence (as long as we make sure the distribution normalizes!

Lets do it!

In [18]:
```
xs = t.Tensor([0., 0.])

def log_prior(p):
    return 0.

def log_likelihood(p):
    return Bernoulli(p).log_prob(xs).sum(-1, keepdim=True)

ps = t.linspace(0.005, 0.995, 100).view(-1, 1)
dp = ps[1] - ps[0]

unnorm_log_posterior = log_prior(ps) + log_likelihood(ps)
unnorm_posterior = unnorm_log_posterior.exp()
norm_posterior = unnorm_posterior / (dp * unnorm_posterior.sum())

fig, ax = plt.subplots()
ax.set_xlabel("p")
ax.set_ylabel("P(p|x=[0,0])")
ax.hist(ps_all_zeros, density=True);
ax.plot(ps, norm_posterior);
```



It is possible to analytically compute the distribution over p for a coin-toss. But this doesn't give us much additional insight. Instead, we'll do this below for a Gaussian distribution.

More interesting examples

In the previous example, a datapoint was either 0 or 1,

```
X = {0, 1}
```

but x could take on almost any other type. Common examples include:

```
X = Vector{Float}   # A vector
X = Str             # A string
X = Image           # An image
```

Complex, state-of-the-art models for images (GANs) and text (GTP-2) do exactly what we described above. Then, they take a large dataset of images/text do a very large amount of processing, to fit a distribution to that data. Once the distribution has been fitted, they can draw samples from that distribution, that should look like the data.

Multivariate Normal (Gaussian)

The multivariate Normal is the most important distribution over vectors.

$$\begin{aligned} \mathrm{P}\left(\mathbf{x}|\boldsymbol{\mu}, \boldsymbol{\Sigma}\right) &= \mathcal{N}\left(\mathbf{x}; \boldsymbol{\mu}, \boldsymbol{\Sigma}\right) \\ \log \mathcal{N}\left(\mathbf{x}; \boldsymbol{\mu}, \boldsymbol{\Sigma}\right) &= -\tfrac{1}{2}\log|2\pi\boldsymbol{\Sigma}| - \tfrac{1}{2}(\mathbf{x}-\boldsymbol{\mu})\,\boldsymbol{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu})^{T} \end{aligned}$$

The expectation of the distribution is given by $\boldsymbol{\mu}$ and the covariance is given by $\boldsymbol{\Sigma}$,

$$\begin{aligned} \mathbb{E}\left[\mathbf{x}\right] &= \boldsymbol{\mu} \\ \mathrm{Cov}\left[\mathbf{x}\right] &= \boldsymbol{\Sigma} \end{aligned}$$

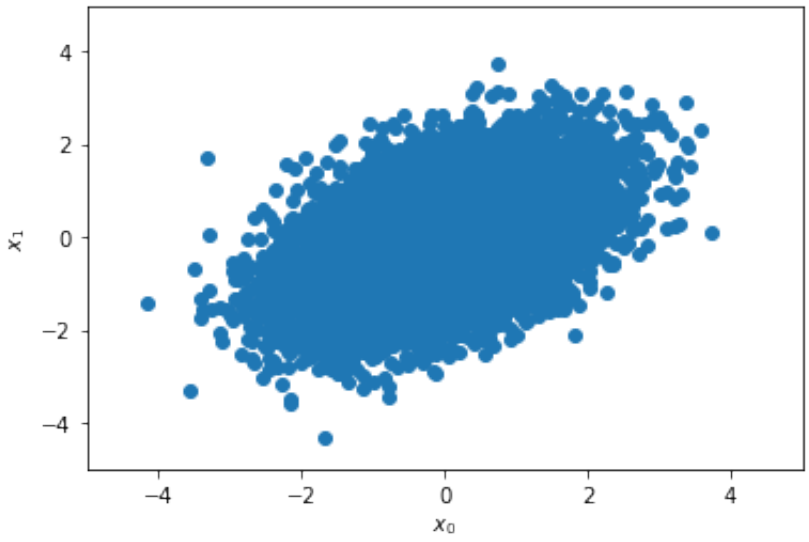
To get some intuition for what the covariance means, we can plot samples from a multivariate normal with the same variance for x_0 and x_1, but different covariances,

In [19]:
```
def plot(c):
    fig, ax = plt.subplots()
    ax.set_xlabel("$x_0$")
    ax.set_ylabel("$x_1$")
    ax.set_xlim(-5, 5)
    ax.set_ylim(-5, 5)

    mu = t.zeros(2)
    Sigma = c*t.ones(2, 2) + (1-c)*t.eye(2)
    print("Sigma =")
    print(Sigma)
    dist = MvNormal(mu, Sigma)
    xs = dist.sample((10000,))
    ax.scatter(xs[:, 0], xs[:, 1])

interact_manual(plot, c=FloatSlider(value=0, min=-0.99, max=0.99, s
```

c 0.47

Run Interact

```
Sigma =
tensor([[1.0000, 0.4700],
        [0.4700, 1.0000]])
```



To begin, consider positive $\mathrm{Cov}\left[x_0, x_1\right] = \mathbb{E}\left[x_0 x_1\right]$. To get positive covariances, x_0 and x_1 will tend to have the same sign (either both positive or both negative), so the overall product, $x_0 x_1$ is usually positive.

Now, consider negative $\mathrm{Cov}\left[x_0, x_1\right] = \mathbb{E}\left[x_0 x_1\right]$. To get negative covariances, x_0 and x_1 will tend to have the different signs (one positive and the other negative), so the overall product, $x_0 x_1$ is usually negative.

Now, consider $0 = \mathrm{Cov}\left[x_0, x_1\right] = \mathbb{E}\left[x_0 x_1\right]$. In this case, x_0 and x_1 are unrelated.

As the covariance approaches the variance, the distribution gets narrower.

Until eventually, the only way of achieving,

$$1 = \mathrm{Var}\left[x_0\right] = \mathrm{Var}\left[x_1\right] \approx \mathrm{Cov}\left[x_0, x_1\right] = \mathbb{E}\left[x_0 x_1\right]$$

is by setting,

$$x_0 = x_1$$

Maximum likelihood fitting

Maximum likelihood parameters, $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$, for a multivariate normal is given by the empirical mean, $\hat{\boldsymbol{\mu}}$ and covariance $\hat{\boldsymbol{\Sigma}}$,

$$\hat{\boldsymbol{\mu}}, \hat{\boldsymbol{\Sigma}} = \underset{\boldsymbol{\mu}, \boldsymbol{\Sigma}}{\arg\max} \left[\sum_{\lambda} \log \mathcal{N}\left(\mathbf{x}_{\lambda} | \boldsymbol{\mu}, \boldsymbol{\Sigma}\right) \right]$$

This seems sensible, but requires some fairly hairy matrix differentials to prove.

In [ ]:

In [ ]: