In [1]:
```
import numpy as np
import torch as t
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
%matplotlib inline
from ipywidgets import FloatSlider, interact, interact_manual
```

Part 2: Supervised learning: regression

In supervised learning, we have a bunch of input, x, output, y, pairs as data.

The great thing about supervised learning is that the inputs and outputs could be almost anything,

```
x :: Image,  y :: Int                # Object recognition
x :: Image,  y :: Matrix{Int}        # Image segmentation
x :: Audio,  y :: Str                # Speech recognition
x :; Str,    y :: Audio              # Text-to-speech
```

The goal is to learn something about the mapping from x to y.

One approach is to learn a function that maps from x to a guess about the corresponding y. This guess is called $\hat{y}$,

$$f(x) \rightarrow \hat{y}$$

To learn f, we try to make y from the data as similar as possible to the estimates, $\hat{y}$.

However, to choose this function, we require a measure of "similiarity". One common example is the squared error,

$$\text{SE} = \left(y - \hat{y}\right)^2$$

However, this approach makes learning difficult for discrete objects: what's the error between two outputs "car" and "truck".

Instead, a generic approach is to write a function that takes an x and returns a distribution over y,

$$f(x) \rightarrow \text{P}\left(y|x\right)$$

It is easier to fit such a distribution, because we can always maximise the probability of the y that we saw in the data.

Note: classification and regression are special types of supervised learning. In classification, the output is, a class-label,

```
Y = Int     #classification
```

in regression, the output is one (or many) real values,

```
Y = Float   #regression
```

If the output is more complicated (e.g. a string, image or audio), then its neither regression or classification, its just supervised learning.

Multivariate linear regression

The most useful and instructive supervised learning method is multivariate linear regression.

The input for the λth data point is, $\mathbf{x}$, is a vector and we consider the multi-output case, where the corresponding output, $\mathbf{y}_\lambda$ is also a vector. We assume that $\mathbf{y}_\lambda$ is Gaussian, conditioned on $\mathbf{x}$,

$$\mathrm{P}\left(y_\lambda|\mathbf{x}_\lambda, \mathbf{w}\right) = \mathcal{N}\left(y_\lambda; \mathbf{x}_\lambda \cdot \mathbf{w}, \sigma^2\right) = \frac{1}{\sqrt{2\pi}\sigma}\exp(-\frac{1}{2}(y_\lambda - \mathbf{x}_\lambda \cdot \mathbf{w})^2)$$

It turns out to be easier to write this as a multivariate Gaussian over all outputs, $\mathbf{y}$, jointly,

$$\mathrm{P}\left(\mathbf{y}|\mathbf{X}, \mathbf{w}\right) = \prod_\lambda \mathrm{P}\left(y_\lambda|\mathbf{x}_\lambda, \mathbf{w}\right) = \mathcal{N}\left(\mathbf{y}; \mathbf{X}\mathbf{w}, \sigma^2\mathbf{I}\right)$$

Writing the log-probability out in full,

$$\mathcal{L}\left(\mathbf{w}\right) = \log \mathrm{P}\left(\mathbf{y}|\mathbf{X}, \mathbf{w}\right) = \sum_\lambda \left[-\tfrac{1}{2}\log 2\pi\sigma^2 - \tfrac{1}{2\sigma^2}\left(Y_\lambda - \sum_i X_{\lambda i} w_i\right)^2\right].$$

The maximum likelihood estimate, $\hat{\mathbf{w}}$, is the maximum of the log-likelihood,

$$\hat{\mathbf{w}} = \underset{\mathbf{w}}{\arg\max}\, \mathcal{L}\left(\mathbf{w}\right)$$

To compute the maximum, we take the gradient,

$$\frac{\partial \mathcal{L}\left(\mathbf{w}\right)}{\partial w_\alpha} = -\frac{1}{2\sigma^2}\frac{\partial}{\partial w_\alpha}\sum_\lambda \left(y_\lambda - \sum_i X_{\lambda i} w_i\right)^2.$$

This looks hard. But because we've written it in index notation, we can do everything in terms of standard calculus. We start by applying the chain rule,

$$\frac{\partial \mathcal{L}\left(\mathbf{w}\right)}{\partial w_\alpha} = -\frac{1}{\sigma^2}\sum_\lambda \left(y_\lambda - \sum_i X_{\lambda i} w_i\right)\frac{\partial}{\partial w_\alpha}\left(y_\lambda - \sum_i X_{\lambda i} w_i\right).$$

interlude: the Kronecker delta (which looks alot like an identity matrix),

$$\frac{\partial w_i}{\partial w_\alpha} = \delta_{i\alpha}$$

$$\delta_{i\alpha} = \begin{cases} 1 & \text{if } i = \alpha \\ 0 & \text{if } i \neq \alpha \end{cases}$$

thus,

$$\frac{\partial \mathcal{L}\left(\mathbf{w}\right)}{\partial w_\alpha} = \frac{1}{\sigma^2}\sum_\lambda \left(y_\lambda - \sum_i X_{\lambda i} w_i\right)\sum_i \delta_{i\alpha} X_{\lambda i}.$$

the $\delta_{i\alpha}$ picks out the $i = \alpha$ term in the sum,

$$\frac{\partial \mathcal{L}\left(\mathbf{w}\right)}{\partial w_\alpha} = \frac{1}{\sigma^2}\sum_\lambda \left(y_\lambda - \sum_i X_{\lambda i} w_i\right) X_{\lambda\alpha}.$$

Finally, put everything back in vector/matrix notation,

$$\frac{\partial \mathcal{L}(\mathbf{w})}{\partial \mathbf{w}} = \tfrac{1}{\sigma^2} \mathbf{X}^T (\mathbf{y} - \mathbf{X}\mathbf{w})$$

We could just follow the gradient uphill (and that's what we do in deep learning), but this is super-slow. Instead, there's an answer that is much faster to compute: at the top of the hill, the gradient is zero,

$$\begin{aligned}
\mathbf{0} &= \left.\frac{\partial \mathcal{L}(\mathbf{w})}{\partial \mathbf{w}}\right|_{\mathbf{w}=\hat{\mathbf{w}}}\\
\mathbf{0} &= \tfrac{1}{\sigma^2} \mathbf{X}^T \left(\mathbf{y} - \mathbf{X}\hat{\mathbf{w}}\right)\\
\mathbf{0} &= \mathbf{X}^T \mathbf{y} - \mathbf{X}^T \mathbf{X}\hat{\mathbf{w}}\\
\mathbf{X}^T \mathbf{X}\hat{\mathbf{w}} &= \mathbf{X}^T \mathbf{y}
\end{aligned}$$

Note that we can't just solve for $\hat{\mathbf{w}}$ using the inverse of $\mathbf{X}^T$ because it is almost never square, so the inverse does not exist! As, $\mathbf{X}^T\mathbf{X}$ is square, we can take its inverse (note that $\mathbf{X}^T$ isn't square, so we can't take its inverse,)

$$\hat{\mathbf{w}} = \left(\mathbf{X}^T \mathbf{X}\right)^{-1} \mathbf{X}^T \mathbf{y}$$

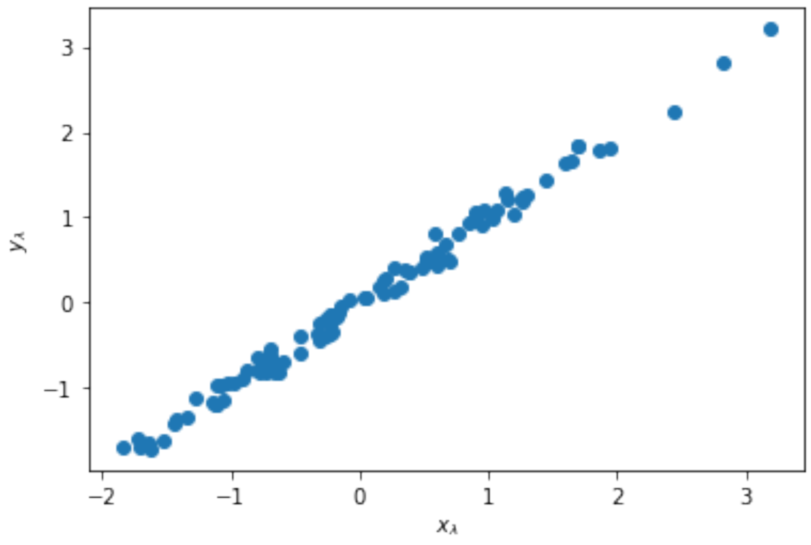
I've done it this way, because the translation into Numpy/PyTorch is most direct, but if you look things up, you will often find the transposes in different places.

Lets write some code! First, we can directly translate the above expression to give a method for fitting $\hat{\mathbf{W}}$,

```
In [2]: def fit_Wh(X, Y):
            return t.inverse(X.T @ X) @ X.T @ Y
```

Now we generate some 1D "fake data"

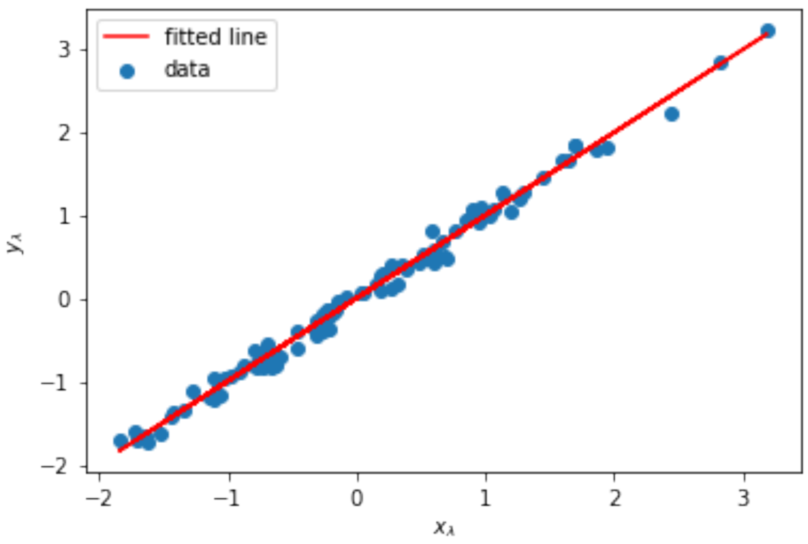
In [3]:
```
N     = 100 # number of datapoints
D     = 1   # dimension of datapoints
sigma = 0.1 # output noise
X     = t.randn(N, D)
Wtrue = t.ones(D, 1)
Y     = X @ Wtrue + sigma*t.randn(N, 1)

fig, ax = plt.subplots()
ax.set_xlabel("$x_\lambda$")
ax.set_ylabel("$y_\lambda$")
ax.scatter(X, Y);
```



In [4]:
```
Wh = fit_Wh(X, Y)
print(f"Wtrue = {Wtrue.T}")
print(f"Wh    = {Wh.T}")
```

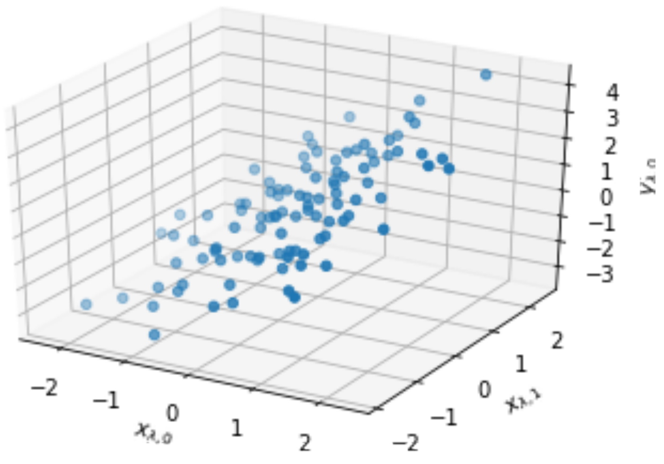
```
Wtrue = tensor([[1.]])
Wh    = tensor([[0.9948]])
```

In [5]:
```
fig, ax = plt.subplots()
ax.set_xlabel("$x_\lambda$")
ax.set_ylabel("$y_\lambda$")
ax.scatter(X, Y, label="data")
ax.plot(X, X@Wh, 'r', label="fitted line")
ax.legend();
```



Now we can also generate some 2D "fake data",

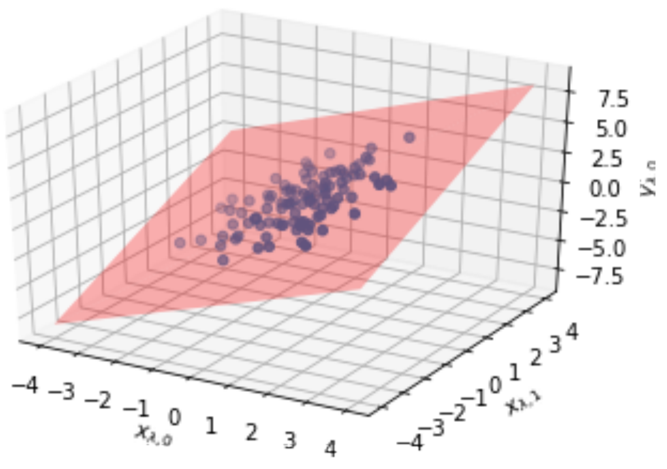
In [6]:
```
N     = 100 # number of datapoints
D     = 2   # dimension of datapoints
sigma = 0.3 # output noise
X     = t.randn(N, D)
Wtrue = t.ones(D, 1)
Y     = X @ Wtrue + sigma*t.randn(N, 1)

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.set_xlabel("$x_{\lambda, 0}$")
ax.set_ylabel("$x_{\lambda, 1}$")
ax.set_zlabel("$y_{\lambda, 0}$")
ax.scatter(xs=X[:, 0], ys=X[:, 1], zs=Y[:, 0]);
```



In [7]:
```
Wh = fit_Wh(X, Y)
print(f"Wtrue = {Wtrue.T}")
print(f"Wh    = {Wh.T}")
```

```
Wtrue = tensor([[1., 1.]])
Wh    = tensor([[1.0201, 1.0456]])
```

In [10]:
```
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.set_xlabel("$x_{\lambda, 0}$")
ax.set_ylabel("$x_{\lambda, 1}$")
ax.set_zlabel("$y_{\lambda, 0}$")
ax.scatter(X[:, 0], X[:, 1], Y[:, 0])

Xp = t.tensor([
    [-4., -4.],
    [-4.,  4.],
    [ 4., -4.],
    [ 4.,  4.]
])

ax.plot_trisurf(
    np.array(Xp[:, 0]),
    np.array(Xp[:, 1]),
    np.array((Xp @ Wh)[:, 0]),
    color='r',
    alpha=0.3
);
```



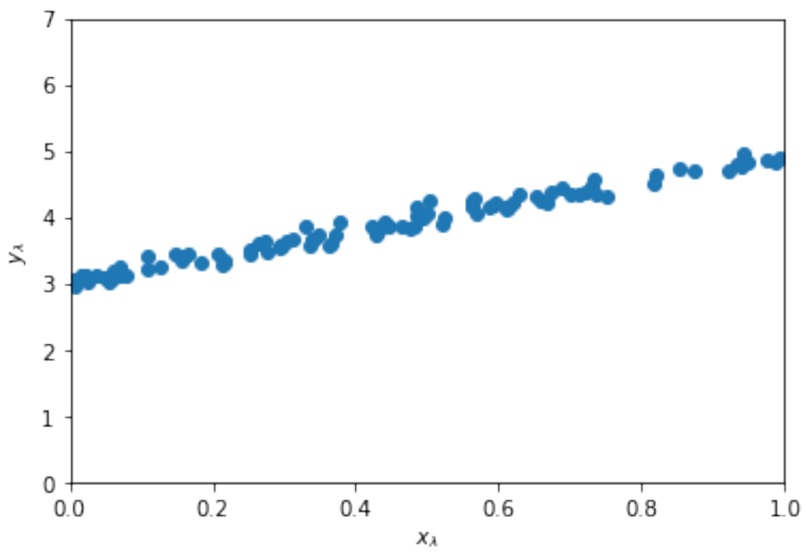
But there are issues with linear regression, as formulated thus far.

In particular, consider 1D data, generated with a bias,

$$\mathrm{P}\left(y_\lambda | \mathbf{x}_\lambda, w, b\right) = \mathcal{N}\left(\mathbf{y}_\lambda; x_\lambda w + b, \sigma^2\right)$$

If we fit our old model, without a bias, it doesn't work,

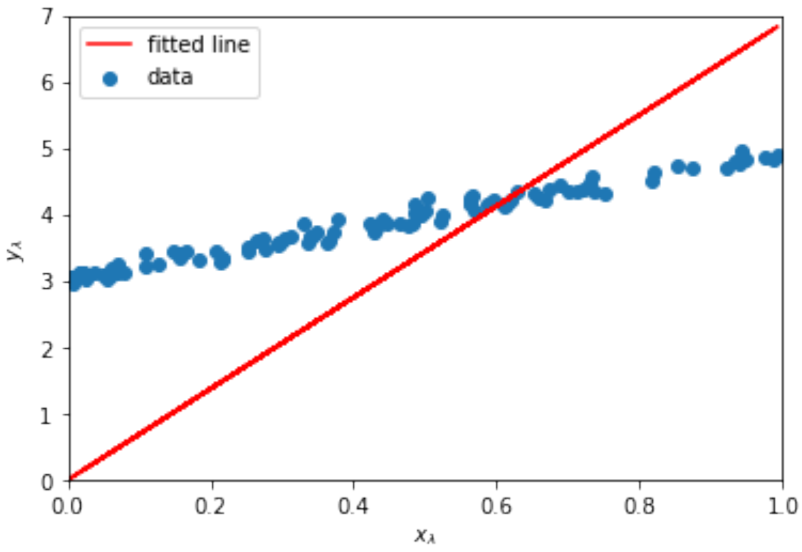
In [11]:
```
N      = 100 # number of datapoints
D      = 1   # dimension of datapoints
sigma = 0.1 # output noise
X      = t.rand(N, D)
Wtrue = 2*t.ones(D, 1)
btrue = 3
Y      = X @ Wtrue + btrue + sigma*t.randn(N, 1)

fig, ax = plt.subplots()
ax.set_xlabel("$x_\lambda$")
ax.set_ylabel("$y_\lambda$")
ax.set_xlim(0, 1)
ax.set_ylim(0, 7)
ax.scatter(X, Y);
```



In [12]:
```
Wh = fit_Wh(X, Y)
print(f"Wtrue = {Wtrue.T}")
print(f"Wh    = {Wh.T}")
```

```
Wtrue = tensor([[2.]])
Wh    = tensor([[6.8757]])
```

In [13]:
```python
fig, ax = plt.subplots()
ax.set_xlabel("$x_\lambda$")
ax.set_ylabel("$y_\lambda$")
ax.set_xlim(0, 1)
ax.set_ylim(0, 7)
ax.scatter(X, Y, label="data")
ax.plot(X, X@Wh, 'r', label="fitted line")
ax.legend();
```



What can we do? Well, we could go through the big derivation above again, incorporating the bias.

But this is super-tedious.

Instead, note that if we expand the 1D feature vector into a 2D feature vector, with the second feature being just biases,

$$\mathrm{P}\left(y_\lambda | \mathbf{x}_\lambda, w, b\right) = \mathcal{N}\left(\mathbf{y}_\lambda; \underbrace{\begin{pmatrix} x_\lambda & 1 \end{pmatrix}}_{\text{expanded feature vector}} \overbrace{\begin{pmatrix} w \\ b \end{pmatrix}}^{\text{expanded weight vector}}, \sigma^2\right).$$

$$\mathrm{P}\left(y_\lambda | \mathbf{x}_\lambda, w, b\right) = \mathcal{N}\left(\mathbf{y}_\lambda; x_\lambda w + b, \sigma^2\right).$$

Now, we can just use our original derivation, and implementation!

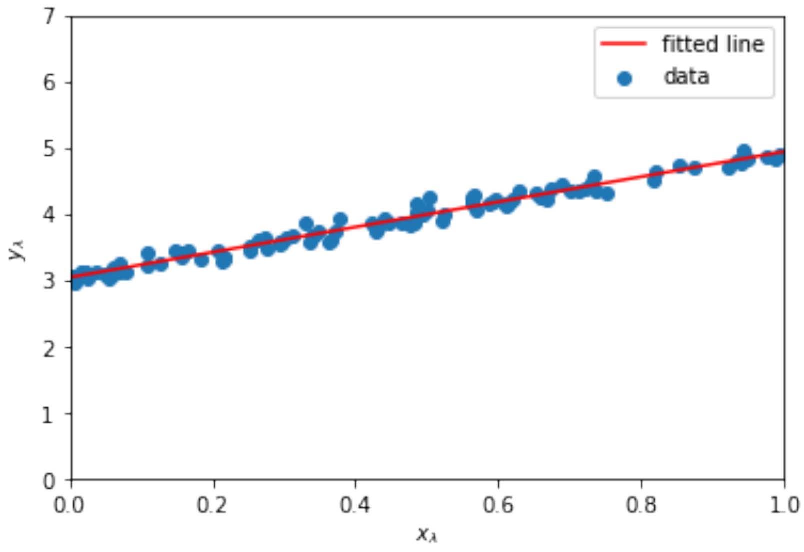
In [14]:
```
def add_bias(X):
    return t.cat([X, t.ones(X.shape[0], 1)], 1)

Xe = add_bias(X)
Xe[:10, :]
```

Out[14]:
```
tensor([[0.0067, 1.0000],
        [0.4455, 1.0000],
        [0.6123, 1.0000],
        [0.6281, 1.0000],
        [0.3726, 1.0000],
        [0.0176, 1.0000],
        [0.3395, 1.0000],
        [0.9377, 1.0000],
        [0.9751, 1.0000],
        [0.0223, 1.0000]])
```

In [15]:
```
Wh = fit_Wh(Xe, Y)
print(f"Wh    = {Wh.T}")
print(f"Wtrue = {Wtrue}")
print(f"btrue = {btrue}")
```

```
Wh    = tensor([[1.8990, 3.0394]])
Wtrue = tensor([[2.]])
btrue = 3
```

In [16]:
```
fig, ax = plt.subplots()
ax.set_xlabel("$x_\lambda$")
ax.set_ylabel("$y_\lambda$")
ax.set_xlim(0, 1)
ax.set_ylim(0, 7)
ax.scatter(X, Y, label="data")

xs = t.tensor([[0.], [1.]])
ax.plot(xs, add_bias(xs)@Wh, 'r', label="fitted line")
ax.legend();
```



It turns out that this idea can be taken *much* further.

In particular, instead of just incorporating a constant feature, we can incorporate arbitrary, nonlinear *functions* of the original input data.

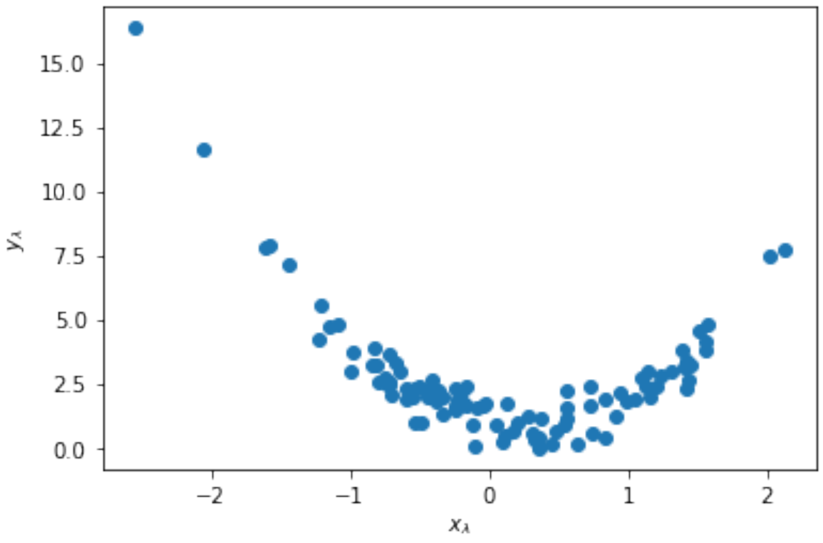
To take an example, consider summing a quadratic, linear and constant function,

$$\mathrm{P}\left(y_\lambda|\mathbf{x}_\lambda, w, b\right) = \mathcal{N}\left(\mathbf{y}_\lambda; \underbrace{\begin{pmatrix} x_\lambda^2 & x_\lambda & 1 \end{pmatrix}}_{\text{expanded feature vector}} \overbrace{\begin{pmatrix} w_1 \\ w_2 \\ w_3 \end{pmatrix}}^{\text{expanded weight vector}}, \sigma^2\right).$$

$$\mathrm{P}\left(y_\lambda|\mathbf{x}_\lambda, w, b\right) = \mathcal{N}\left(\mathbf{y}_\lambda; w_1 x_\lambda^2 + w_2 x_\lambda + w_3, \sigma^2\right).$$

For instance,

In [17]:
```
N     = 100 # number of datapoints
D     = 1   # dimension of datapoints
sigma = 0.5 # output noise
X     = t.randn(N, D)
qtrue = 2  # quadratic term
ltrue = -1 # linear term
btrue = 1  # bias
Y     = qtrue*X**2 + ltrue*X + btrue + sigma*t.randn(N, 1)

fig, ax = plt.subplots()
ax.set_xlabel("$x_\lambda$")
ax.set_ylabel("$y_\lambda$")
ax.scatter(X, Y);
```



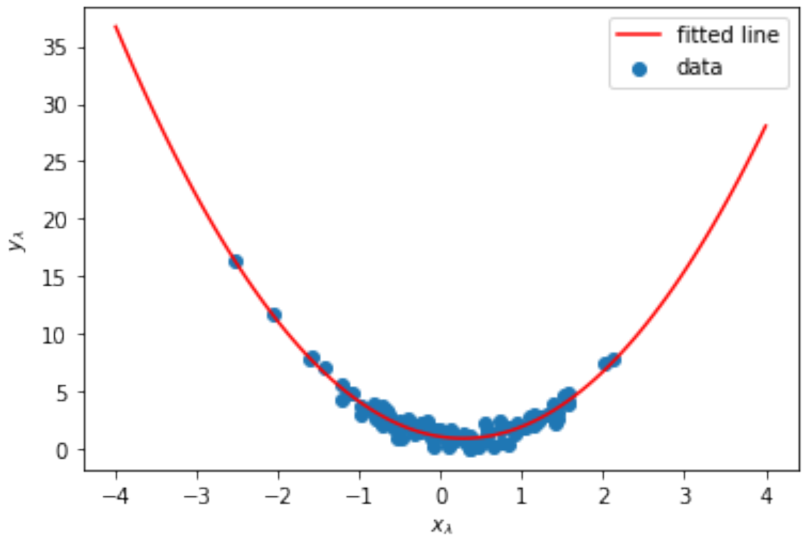
In [18]:
```
def quad(X):
    return t.cat([X**2, X, t.ones(X.shape[0], 1)], 1)

Xe = quad(X)
Xe[:10, :]
```

Out[18]:
```
tensor([[ 0.1000,  0.3162,  1.0000],
        [ 1.5203,  1.2330,  1.0000],
        [ 0.9510, -0.9752,  1.0000],
        [ 0.5260,  0.7252,  1.0000],
        [ 0.0392,  0.1980,  1.0000],
        [ 0.3072,  0.5543,  1.0000],
        [ 1.7088,  1.3072,  1.0000],
        [ 0.1984, -0.4454,  1.0000],
        [ 0.3469, -0.5890,  1.0000],
        [ 0.6601, -0.8125,  1.0000]])
```

In [19]:
```
Wh = fit_Wh(Xe, Y)
print(f"Wh    = {Wh.T}")
print(f"qtrue = {qtrue}")
print(f"ltrue = {ltrue}")
print(f"btrue = {btrue}")
```

```
Wh    = tensor([[ 1.9593, -1.0804,  1.0469]])
qtrue = 2
ltrue = -1
btrue = 1
```

In [20]:
```
fig, ax = plt.subplots()
ax.set_xlabel("$x_\lambda$")
ax.set_ylabel("$y_\lambda$")

ax.scatter(X, Y, label="data")

xs = t.linspace(-4, 4, 100)[:, None]
ax.plot(xs, quad(xs)@Wh, 'r', label="fitted line")
ax.legend();
```



In [ ]:

In [ ]: