part_4_classification

February 21, 2022

[17]:

```
import numpy as np
import torch as t
from torch.distributions import Normal, Categorical, Bernoulli
from torch.distributions import MultivariateNormal as MvNormal
import matplotlib.pyplot as plt
#Enable for prettier plots
#%matplotlib inline
from ipywidgets import FloatSlider, IntSlider, interact, interact_manual
```

Part 4: Classification

Classification is almost exactly the same as regression, except that:

The outputs, y, are discrete class-labels.

Almost all interesting/useful algorithms require iterative solutions

The same considerations are relevant, including,

Overfitting

Regularisation

Cross-validation

Bayes (but this is much harder, as there aren't any exact solutions)

Prerequisites: Bernoulli distribution for two-class classification

Samples from the Bernoulli distribution are either 0 or 1, with probability given by the parameter,

$$\mathrm{P}\left(y|p\right) = \text{Bernoulli}\left(y; p\right) = yp + (1-y)(1-p) = \begin{cases} p & \text{if } y=1 \\ 1-p & \text{if } y=0 \end{cases} \tag{1}$$

[18]:

```
Py = Bernoulli(probs=0.8)
print(Py.sample((10,)))
print(Py.log_prob(t.tensor([0., 1.])).exp())
```

```
tensor([1., 0., 1., 1., 1., 0., 1., 0., 1., 1.])
tensor([0.2000, 0.8000])
```

Logit parameterisation of the Bernoulli distribution and the sigmoid

Working directly with the probabilities turns out to be problematic:

Probabilities live in a strange range, $0 \leq p \leq 1$.

There is a strong risk of numerical underflow, which breaks algorithms such as (stochastic) gradient descent.

Instead, we can also treat the Bernoulli parameter as a logits vector, ℓ, defined such that,

$$p = \text{sigmoid}(\ell) = \sigma(\ell) \tag{2}$$

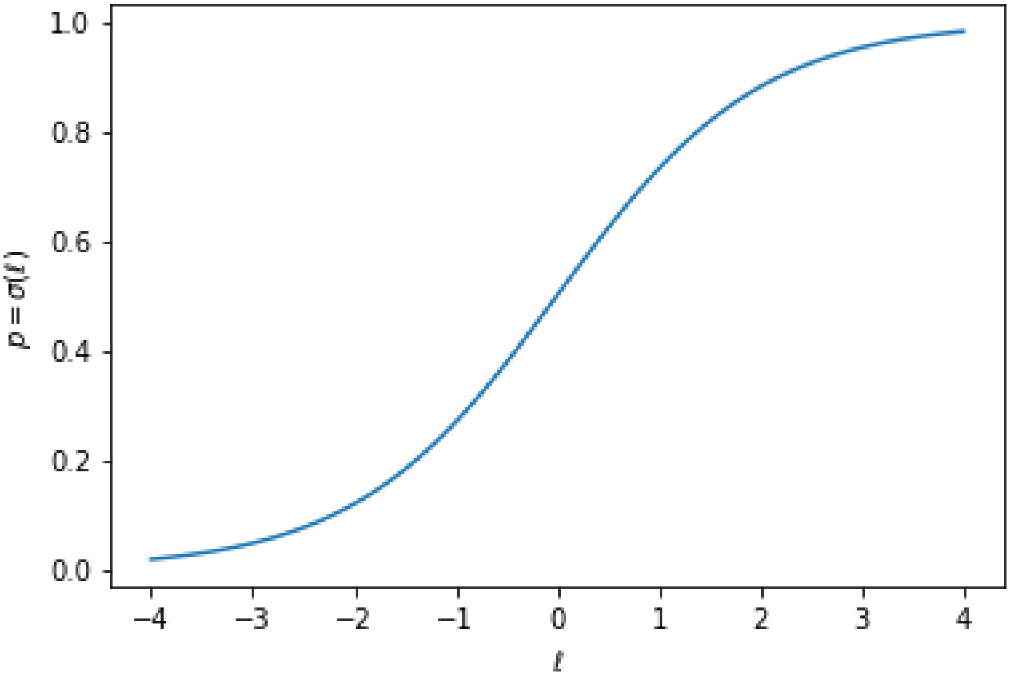
$$p = \frac{1}{1 + e^{-\ell}} \tag{3}$$

Now, no matter what $\boldsymbol{\ell}$ is, the probabilities must lie in the right range, and they are much less likely to underflow.

PyTorch allows you to directly use the logit parameterisation,

[19]:

```
ls = t.linspace(-4, 4, 100)
ps = t.sigmoid(ls)
fig, ax = plt.subplots()
ax.set_xlabel("$\ell$")
ax.set_ylabel("$p = \sigma(\ell)$")
ax.plot(ls, ps);
```



[20]:

```
Py = Bernoulli(logits=10)
print(Py.sample((10,)))
print(Py.log_prob(t.tensor([0., 1.])).exp())
```

```
tensor([1., 1., 1., 1., 1., 1., 1., 1., 1., 1.])
tensor([4.5398e-05, 9.9995e-01])
```

Prerequisites: Categorical distribution for multi-class classification

The Bernoulli distribution takes one parameter and gives the probability of two classes. But what about multiple classes?

Samples from the Categorical distribution are integers $0 \leq y < K$, with probability given explicity by a length K vector of, $\mathbf{p}$,

$$\mathrm{P}(y|\mathbf{p}) = \text{Categorical}\,(y; \mathbf{p}) = p_y \tag{4}$$

[21]:

```
#A uniform Categorical distribution,
Py = Categorical(probs=t.ones(10)/10)
print(Py.sample((20,)))
```

```
tensor([7, 4, 5, 1, 6, 0, 4, 4, 7, 3, 3, 7, 0, 8, 0, 8, 3, 6, 0, 0])
```

[22]:

```
ys = t.arange(10)
print(ys)
Py.log_prob(ys).exp()
```

```
tensor([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

[22]:

```
tensor([0.1000, 0.1000, 0.1000, 0.1000, 0.1000, 0.1000, 0.1000, 0.1000, 0.1000,
        0.1000])
```

[23]:

```
#A non-uniform Categorical distribution,
p = t.tensor([0.5, 0.1, 0.1, 0.1, 0.1, 0.1])
Py = Categorical(probs=p)
y = Py.sample((20,))
y
```

[23]:

```
tensor([2, 0, 0, 4, 0, 0, 0, 3, 1, 1, 0, 3, 2, 2, 0, 1, 0, 0, 3, 2])
```

[24]:

```
Py.log_prob(t.arange(6)).exp()
```

[24]:

```
tensor([0.5000, 0.1000, 0.1000, 0.1000, 0.1000, 0.1000])
```

Logit parameterisation of the Categorical distribution and the softmax

Working directly with the probabilities turns out to be problematic:

Probabilities live in a strange range, $0 \leq p_i \leq 1$.

Probabilities must sum to 1.

There is a strong risk of numerical underflow, which breaks algorithms such as (stochastic) gradient descent.

Instead, we can also treat the Categorical parameter as a logits vector, $\boldsymbol{\ell}$, defined such that,

$$
\begin{align}
\mathbf{p} &= \operatorname{softmax}(\boldsymbol{\ell}) \tag{5}\\
p_i &= \frac{e^{\ell_i}}{\sum_{j=0}^{K-1} e^{\ell_j}} \tag{6}
\end{align}
$$

Now, no matter what $\boldsymbol{\ell}$ is, the probabilities must lie in the right range, they must normalize, and they are much less likely to underflow.

PyTorch allows you to directly use the logit parameterisation,

[25]:
```
# Uniform Categorical
Py = Categorical(logits = -10*t.ones(10))
print(Py.sample((20,)))
print(Py.log_prob(t.arange(10)).exp())
```

```
tensor([0, 9, 7, 6, 5, 7, 3, 5, 9, 6, 3, 8, 7, 3, 8, 6, 9, 2, 0, 4])
tensor([0.1000, 0.1000, 0.1000, 0.1000, 0.1000, 0.1000, 0.1000, 0.1000, 0.1000,
        0.1000])
```

[26]:
```
# Non-uniform Categorical
l = t.tensor([1., 0., -1., -2., -10.])
Py = Categorical(logits = l)
print(Py.sample((20,)))
print(Py.log_prob(t.arange(5)).exp())
```

```
tensor([0, 0, 0, 2, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 3, 0, 0, 1])
tensor([6.4391e-01, 2.3688e-01, 8.7143e-02, 3.2058e-02, 1.0754e-05])
```

Formalising maximum-likelihood supervised learning

For linear regression, the goal was to predict the distribution over a floating-point y_λ based on a given value for $\mathbf{x}_\lambda$,

$$
\mathrm{P}(y_\lambda|\mathbf{x}_\lambda) = \mathcal{N}\left(y_\lambda; \mathbf{x}_\lambda \cdot \mathbf{w}, \sigma^2\right) \tag{7}
$$

In classification, the goal is similar: we again want to predict a distribution over y_λ, based on an input, $\mathbf{x}_\lambda$. The only difference is that here, y_λ is an integer from 0 to $K-1$. Thus, when we predict a probability over y_λ it either needs to be Bernoulli (for two classes), or categorical (if there can be more classes,

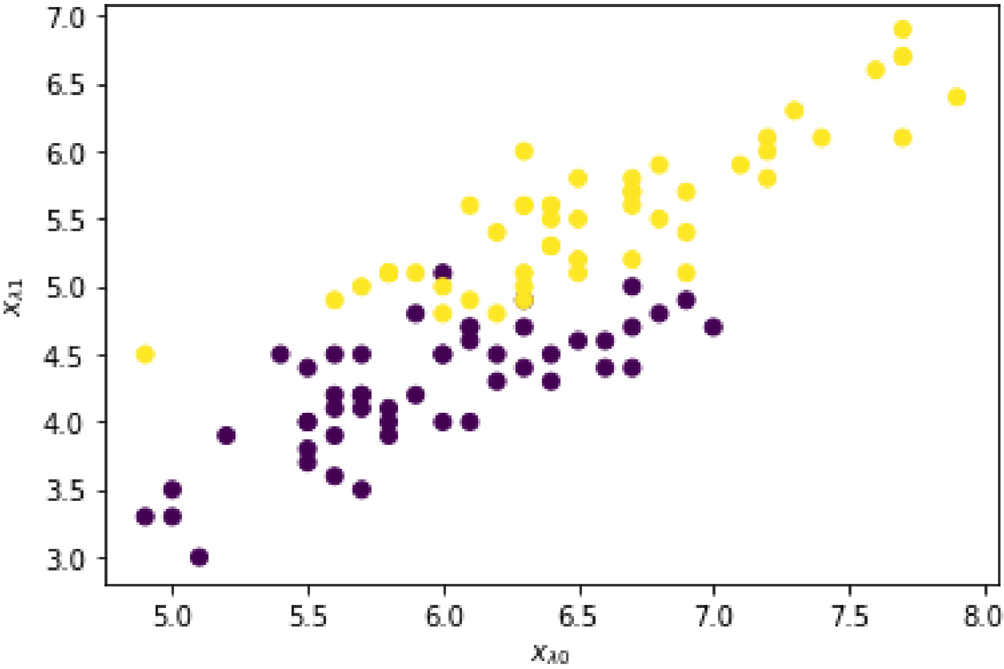
$$\mathrm{P}\left(y_\lambda|\mathbf{x}_\lambda\right) = \text{Bernoulli}\left(y_\lambda; p = \sigma\left(\mathbf{x}_\lambda \cdot \mathbf{w}\right)\right) \tag{8}$$
$$\mathrm{P}\left(y_\lambda|\mathbf{x}_\lambda\right) = \text{Categorical}\left(y_\lambda; \mathbf{p} = \text{softmax}\left(\mathbf{x}_\lambda^T \mathbf{W}\right)\right) \tag{9}$$

or

$$\mathrm{P}\left(\mathbf{y}|\mathbf{X}\right) = \prod_\lambda \mathrm{P}\left(y_\lambda|\mathbf{x}_\lambda\right) = \text{Bernoulli}\left(\mathbf{y}; \sigma\left(\mathbf{X}\mathbf{w}\right)\right) \tag{10}$$
$$\mathrm{P}\left(\mathbf{y}|\mathbf{X}\right) = \prod_\lambda \mathrm{P}\left(y_\lambda|\mathbf{x}_\lambda\right) = \text{Categorical}\left(\mathbf{y}; \text{softmax}\left(\mathbf{X}\mathbf{W}\right)\right) \tag{11}$$

As an example, consider the “Iris” dataset, which is about classifying flowers based on features such as Petal length.

[27]:
```python
from sklearn import datasets
iris = datasets.load_iris()
_X = t.tensor(iris['data'][50:, [0, 2]]).float()
_Y = (t.tensor(iris['target'][50:])-1).float()

fig, ax = plt.subplots()
ax.set_xlabel("$x_{\lambda 0}$")
ax.set_ylabel("$x_{\lambda 1}$")
ax.scatter(_X[:, 0], _X[:, 1], c=_Y);
```



Split the data into train and test (we shuffle the data first, because the classes are ordered)

[28]:

```
t.manual_seed(0)
perm = t.randperm(_X.shape[0])
X = _X[perm, :]
X = t.cat([X, t.ones(X.shape[0], 1)], -1)
Y = _Y[perm][:, None]

X_train = X[:70, :]
Y_train = Y[:70, :]
X_test  = X[70:, :]
Y_test  = Y[70:, :]
```

There are some more quicker iterative algorithms. But they don't add much understanding. So instead, we use PyTorch-magic to do gradient-descent.

[29]:

```
W = t.randn((3,1), requires_grad=True)/100

for i in range(50000):
    L = Bernoulli(logits=X_train@W).log_prob(Y_train).sum()

    dW = t.autograd.grad(outputs=L, inputs=(W,))[0]
    if 0==i % 1000:
        print(L.item())
    W.data += 0.001*dW
```

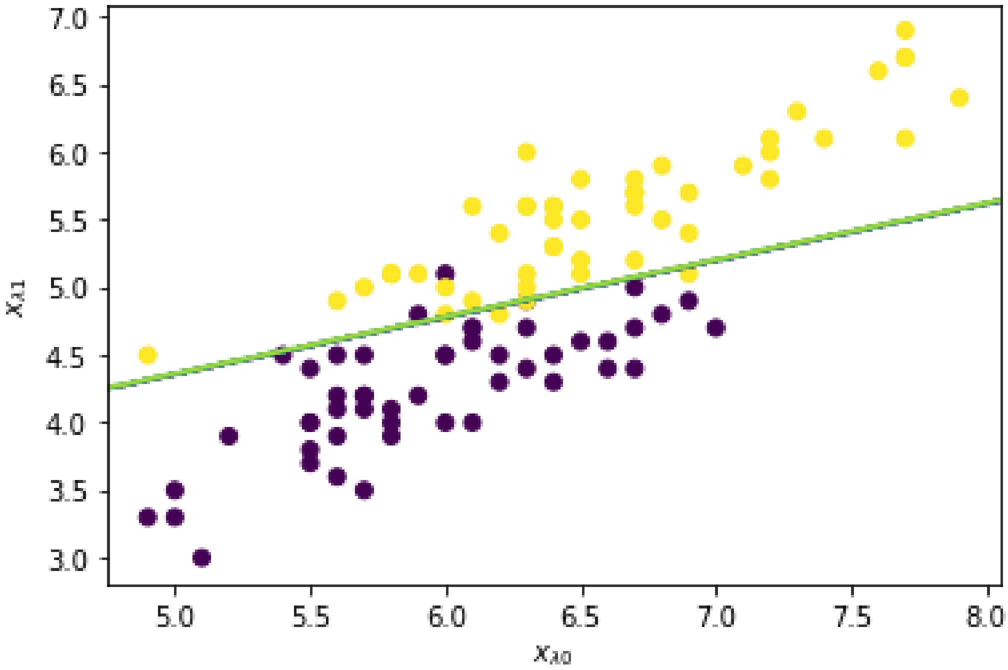
```
-48.97450637817383
-20.44990348815918
-17.061649322509766
-15.644775390625
-14.787422180175781
-14.168647766113281
-13.676431655883789
-13.261651039123535
-12.899397850036621
-12.57563304901123
-12.281757354736328
-12.01214599609375
-11.762897491455078
-11.53117561340332
-11.314810752868652
-11.112079620361328
-10.921627044677734
-10.742270469665527
-10.57301139831543
-10.413019180297852
-10.26151180267334
-10.117837905883789
```

```
-9.981404304504395
-9.851656913757324
-9.728117942810059
-9.610359191894531
-9.49797534942627
-9.39061450958252
-9.287946701049805
-9.189658164978027
-9.095460891723633
-9.005101203918457
-8.918354988098145
-8.835001945495605
-8.754850387573242
-8.677701950073242
-8.603403091430664
-8.53178882598877
-8.46267032623291
-8.395984649658203
-8.331560134887695
-8.269282341003418
-8.20904541015625
-8.150765419006348
-8.094314575195312
-8.039603233337402
-7.986579895019531
-7.935169219970703
-7.885283470153809
-7.836832046508789
```

Now, we can plot the classification boundary,

[30]:

```
fig, ax = plt.subplots()
ax.set_xlabel("$x_{\lambda 0}$")
ax.set_ylabel("$x_{\lambda 1}$")
ax.scatter(_X[:, 0], _X[:, 1], c=_Y);

x0range = t.linspace(*ax.get_xlim(), 500)
x1range = t.linspace(*ax.get_ylim(), 500)

x0s, x1s = t.meshgrid(x0range, x1range)
xs = t.stack([x0s, x1s, t.ones(500, 500)], -1)

ps = Bernoulli(logits=xs@W.detach()).probs[:, :, 0]
ax.contour(x0s, x1s, 0.5<ps);
```



Now, we can compute the classification-error,

[31]:

```
def class_error(X, Y):
    Py = Bernoulli(logits=X@W)
    pred = 0.5 < Py.probs
    N_correct = (pred == Y).sum()
    print(f"{N_correct}/{X.shape[0]} = {100.*N_correct/X.shape[0]}%")

print("Training correct")
class_error(X_train, Y_train)

print("Test correct")
class_error(X_test, Y_test)
```

```
Training correct
67/70 = 95.71428680419922%
Test correct
28/30 = 93.33333587646484%
```

Note that the classifier performs really well on training data, but less well on test data.

This should almost always be true to some extent, and the small difference here shouldn't worry us.

Overfitting in classification

[32]:

```
def cheb(xs, c):
    # c is int
    coefs = c*[0] + [1]
    return np.polynomial.chebyshev.chebval(xs, coefs)
def chebX(X, order):
    assert (-1 <= X).all() and (X <= 1).all()

    xs = []
    for c in range(order):
        xs.append(cheb(X, c))
    return t.cat(xs, 1)
t.manual_seed(0)
N = 100
X = 2*t.rand(N, 1)-1
W_true = t.tensor([[4.]])
Y = Bernoulli(logits=X@W_true).sample()

def plot(order):
    Xe= chebX(X, order)
    W = t.zeros((order, 1), requires_grad=True)

    for i in range(15000):
        L = Bernoulli(logits=Xe@W).log_prob(Y).sum()
        if 0==i % 2000:
            print(L.item())
        dW = t.autograd.grad(outputs=L, inputs=(W,))[0]
        W.data += 0.001*dW

    fig, ax = plt.subplots()
    ax.set_xlabel("$x$")
    ax.set_ylabel("probability / $y$")
    ax.scatter(X, Y)
    xs = t.linspace(-1, 1, 100)[:, None]
    ps_fitted = Bernoulli(logits=chebX(xs, order)@W).probs.detach()
    ps_true = Bernoulli(logits=xs@W_true).probs.detach()
    ax.plot(xs, ps_fitted, label="fitted probability")
    ax.plot(xs, ps_true, label="true probability")
    ax.legend()

interact_manual(plot, order=IntSlider(min=2, max=30));
```

```
interactive(children=(IntSlider(value=2, description='order', max=30, min=2),␣
 ↪Button(description='Run Interact…
```

You can use the same techniques to control overfitting:

Regularisation

Cross-validation (either on the number of classification errors for test points, or on the test-log-likelihood

Bayesian inference (though its much harder here because the posterior over the weights doesn't have an analytic form)

Simpler, heuristic methods for classification

We've seen a bunch of complex methods, and implementing them is often quite involved. Is there anything simpler?

The answer is yes!

K-nearest neighbour

One approach is to look at the nearby datapoints. If the nearby points come from one class, then the chances are that our datapoint come from the same class.

[35]:

```
from sklearn import datasets
iris = datasets.load_iris()
X = t.tensor(iris['data'][50:, [0, 2]]).float()
Y = (t.tensor(iris['target'][50:])-1)[:, None]

def plot(K):
    fig, ax = plt.subplots()
    ax.set_xlabel("$x_{\lambda 0}$")
    ax.set_ylabel("$x_{\lambda 1}$")
    ax.scatter(X[:, 0], X[:, 1], c=Y[:, 0]);

    x0range = t.linspace(*ax.get_xlim(), 500)
    x1range = t.linspace(*ax.get_ylim(), 500)

    x0s, x1s = t.meshgrid(x0range, x1range)
    xs = t.stack([x0s, x1s], -1)

    X_exp = X[:, None, None, :]

    dist2 = ((X_exp - xs)**2).sum(-1)
    elems = (-dist2).topk(K, dim=0).indices
    pred = (0.5<Y[elems, 0].float().mean(0)).float()
    ax.contour(x0s, x1s, pred)

interact_manual(plot, K=IntSlider(min=1, max=11, step=2));
```

```
interactive(children=(IntSlider(value=1, description='K', max=11, min=1,␣
 ↪step=2), Button(description='Run Inte…
```

Cross-validation

Choose K using cross-validation. Note that leave-one-out cross validation is very suitable here, as we make a prediction for each point, based on its neighbours, in parallel.

Weighted-nearest neighbour

This approach is to take a weighted average of nearby datapoints,

$$p_y(\mathbf{x}) = \frac{\sum_\lambda k(\mathbf{x}, \mathbf{x}_\lambda)\delta_{y,y_\lambda}}{\sum_\lambda k(\mathbf{x}, \mathbf{x}_\lambda)} \tag{12}$$

where we might use a squared-exponential kernel/weights (but many other choices are available),

$$k(\mathbf{x}_\lambda, \mathbf{x}_{\lambda'}) = e^{-(\mathbf{x}_\lambda - \mathbf{x}_{\lambda'})^2/(2b)} \tag{13}$$

where b is a bandwidth/lengthscale parameter.

[36]:

```python
def plot(bandwidth):
    fig, ax = plt.subplots()
    ax.set_xlabel("$x_{\lambda 0}$")
    ax.set_ylabel("$x_{\lambda 1}$")
    ax.scatter(X[:, 0], X[:, 1], c=Y[:, 0]);

    x0range = t.linspace(*ax.get_xlim(), 500)
    x1range = t.linspace(*ax.get_ylim(), 500)

    x0s, x1s = t.meshgrid(x0range, x1range)
    xs = t.stack([x0s, x1s], -1)

    X_exp = X[:, None, None, :]

    dist2 = ((X_exp - xs)**2).sum(-1)
    ws = t.exp(-dist2/(2*bandwidth**2))
    print(ws.shape)
    pred = (ws*Y[:, :, None]).sum(0) / ws.sum(0)
    ax.contour(x0s, x1s, 0.5<pred);

interact_manual(plot, bandwidth=FloatSlider(min=0.15, max=1., step=0.01));
```

```
interactive(children=(FloatSlider(value=0.15, description='bandwidth', max=1.0,␣
↪min=0.15, step=0.01), Button(d…
```

Cross-validation

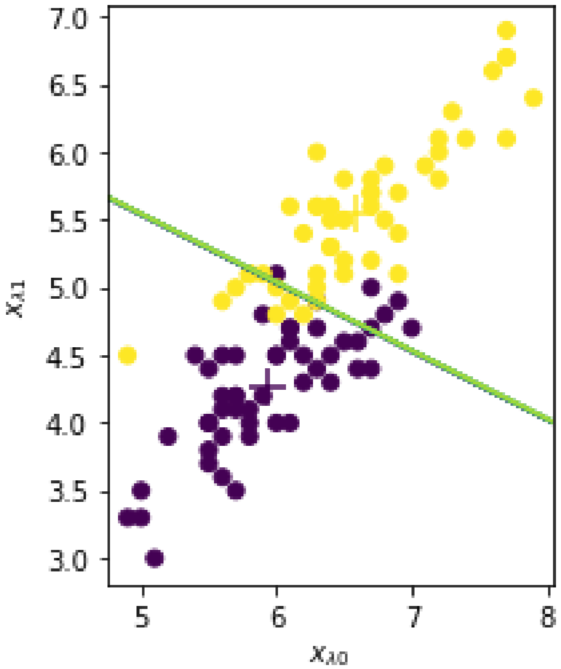
Choose bandwidth using cross-validation. Note that leave-one-out cross validation is very suitable here, as we make a prediction for each point, based on its neighbours, in parallel.

Nearest centroids

Saw this in the lab. Here we compute the center of the inputs for each class, and to classify a new input point, we ask which cluster-center it is closest to.

[37]:

```python
fig, ax = plt.subplots()
ax.set_xlabel("$x_{\lambda 0}$")
ax.set_ylabel("$x_{\lambda 1}$")
ax.scatter(X[:, 0], X[:, 1], c=Y[:, 0]);

mu0 = (X*(1-Y)).sum(0) / (1-Y).sum()
mu1 = (X*Y).sum(0) / Y.sum()

ax.scatter(t.tensor([mu0[0], mu1[0]]), t.tensor([mu0[1], mu1[1]]), c=t.tensor([0., 1.]), s=200, marker="+")
ax.set_aspect('equal', 'box')

x0range = t.linspace(*ax.get_xlim(), 500)
x1range = t.linspace(*ax.get_ylim(), 500)

x0s, x1s = t.meshgrid(x0range, x1range)
xs = t.stack([x0s, x1s], -1)

d0 = ((xs - mu0)**2).sum(-1)
d1 = ((xs - mu1)**2).sum(-1)
pred = d1 < d0

ax.contour(x0s, x1s, pred);
```



Using Bayes theorem to do classification

Nearest centroid works quite poorly, because it only takes into account the mean of the classes, not their shape.

To take into account the shape of these distributions, it turns out that we can use Bayes theorem to compute a probability distribution over the class-label (in contrast to previously, where we used Bayes theorem to compute a distribution over the parameters for linear regression).

In particular, above we directly learned weights that map directly from the data point to a distribution over class labels. Here, we separately learn a distribution over data-points for a single class, $\mathrm{P}(\mathbf{x}|y)$. And we could in principle use this distribution to generate $\mathbf{x}$ that look like the input points from that class.

We can also use Bayes theorem to give us the probability of a class, conditioned on a data-point,

$$\mathrm{P}(y|\mathbf{x}) = \frac{\mathrm{P}(y)\,\mathrm{P}(\mathbf{x}|y)}{\mathrm{P}(\mathbf{x})} \propto \mathrm{P}(y)\,\mathrm{P}(\mathbf{x}|y)\,. \tag{14}$$

The simplest example is to take,

$$\mathrm{P}(\mathbf{x}|y) = \mathcal{N}\left(\mathbf{x}; \boldsymbol{\mu}_y, \sigma^2\mathbf{I}\right). \tag{15}$$

This ends up being equivalent to nearest-centroid!

In particular, consider a uniform prior,

$$\mathrm{P}(y) = 1/K \tag{16}$$

so the posterior becomes proportional to the likelihood,

$$\mathrm{P}(y|\mathbf{x}) \propto \mathrm{P}(\mathbf{x}|y) = \mathcal{N}\left(\mathbf{x}; \boldsymbol{\mu}_y, \sigma^2\mathbf{I}\right). \tag{17}$$

And the prediction is the class with the highest probability density.

Converting to the log-domain (remembering that the log-transform is monotonically increasing, so it doesn't change the ordering),

$$\begin{aligned}
\log \mathrm{P}(y|\mathbf{x}) &= \log \mathcal{N}\left(\mathbf{x}; \boldsymbol{\mu}_y, \sigma^2\mathbf{I}\right) + \text{const} && (18)\\
&= -\tfrac{1}{2\sigma^2}\left(\mathbf{x} - \boldsymbol{\mu}_y\right)^2 + \text{const} && (19)\\
& && (20)
\end{aligned}$$

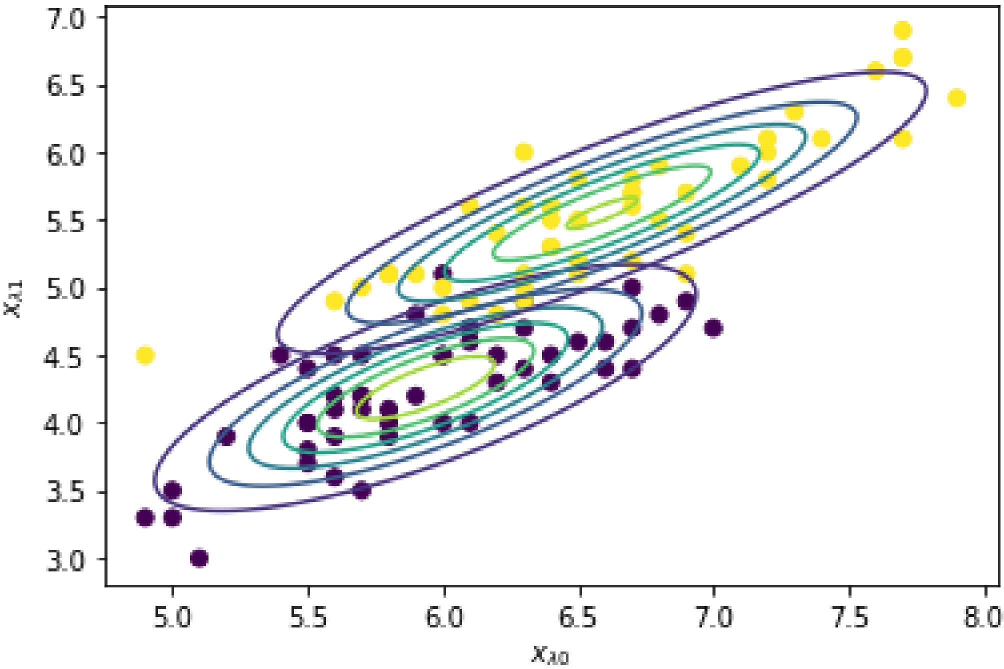
So the highest-probability class is the one with the smallest distance between the input, $\mathbf{x}_\lambda$, and the mean for that class, $\boldsymbol{\mu}_y$.

Critically, once we have this probabilistic representation, we get a recipe for doing a better job, by taking into account the shape of the input distributions. In particular, we can fit multivariate normals to the input points,

$$\mathrm{P}(\mathbf{x}|y) = \mathcal{N}\left(\mathbf{x}; \boldsymbol{\mu}_y, \ {}_y\right) . \tag{21}$$

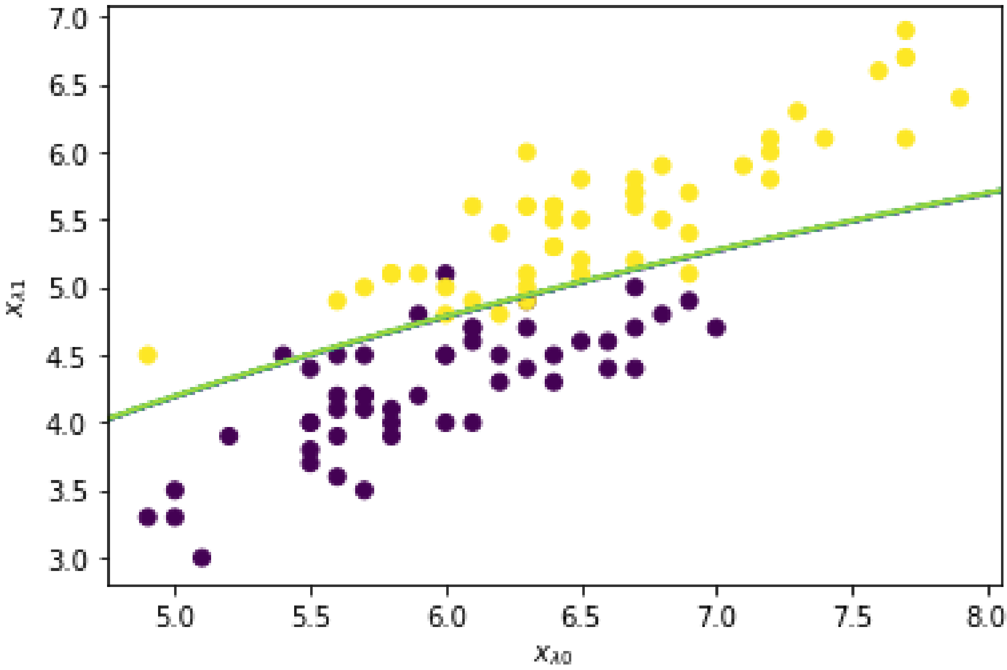
[38]:

```
fig, ax = plt.subplots()
ax.set_xlabel("$x_{\lambda 0}$")
ax.set_ylabel("$x_{\lambda 1}$")
ax.scatter(X[:, 0], X[:, 1], c=Y[:, 0]);

X0 = X[Y[:, 0]==0, :]
X1 = X[Y[:, 0]==1, :]

mu0 = X0.mean(0)
mu1 = X1.mean(0)

C0 = (X0-mu0).t() @ (X0-mu0) / X0.shape[0]
C1 = (X1-mu1).t() @ (X1-mu1) / X1.shape[0]

N0 = MvNormal(mu0, C0)
N1 = MvNormal(mu1, C1)


x0range = t.linspace(*ax.get_xlim(), 500)
x1range = t.linspace(*ax.get_ylim(), 500)
x0s, x1s = t.meshgrid(x0range, x1range)
xs = t.stack([x0s, x1s], -1)

ax.contour(x0s, x1s, N0.log_prob(xs).exp())
ax.contour(x0s, x1s, N1.log_prob(xs).exp());
```



[39]:

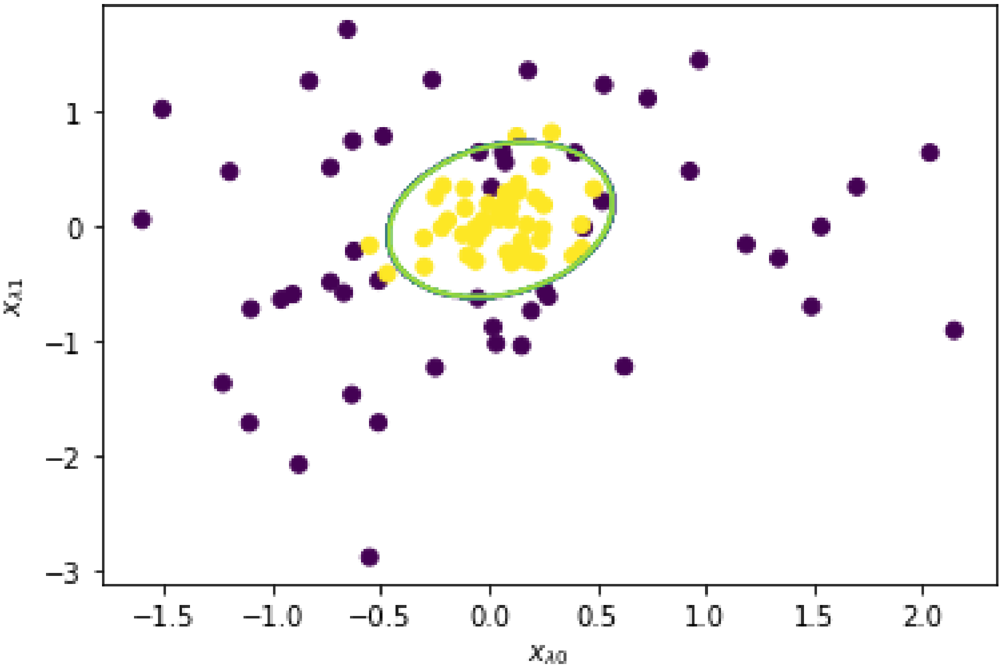
```
fig, ax = plt.subplots()
ax.set_xlabel("$x_{\lambda 0}$")
ax.set_ylabel("$x_{\lambda 1}$")
ax.scatter(X[:, 0], X[:, 1], c=Y[:, 0]);

x0range = t.linspace(*ax.get_xlim(), 500)
x1range = t.linspace(*ax.get_ylim(), 500)
x0s, x1s = t.meshgrid(x0range, x1range)
xs = t.stack([x0s, x1s], -1)

ax.contour(x0s, x1s, 0<(N1.log_prob(xs) - N0.log_prob(xs)));
```



[40]:

```
X0 = t.randn(50, 2)
X1 = t.randn(50, 2) / 5

Y0 = t.zeros(50, 1)
Y1 = t.ones(50, 1)

X = t.cat([X0, X1], 0)
Y = t.cat([Y0, Y1], 0)


fig, ax = plt.subplots()
ax.set_xlabel("$x_{\lambda 0}$")
ax.set_ylabel("$x_{\lambda 1}$")
ax.scatter(X[:, 0], X[:, 1], c=Y[:, 0]);

X0 = X[Y[:, 0]==0, :]
X1 = X[Y[:, 0]==1, :]

mu0 = X0.mean(0)
mu1 = X1.mean(0)

C0 = (X0-mu0).t() @ (X0-mu0) / X0.shape[0]
C1 = (X1-mu1).t() @ (X1-mu1) / X1.shape[0]
```

```
N0 = MvNormal(mu0, C0)
N1 = MvNormal(mu1, C1)


x0range = t.linspace(*ax.get_xlim(), 500)
x1range = t.linspace(*ax.get_ylim(), 500)
x0s, x1s = t.meshgrid(x0range, x1range)
xs = t.stack([x0s, x1s], -1)

ax.contour(x0s, x1s, 0<(N1.log_prob(xs) - N0.log_prob(xs)));
```



Naive Bayes

What about when the data is variable-length?

For instance, consider classifying an email as spam/not-spam using just the number of each word (e.g. with “viagra” indicating spam and “Excel” indicating not-spam).

We take each word, $x_{\lambda,i}$, in an email, x_λ to be given by a probability distribution over words,

$$P(x_{\lambda,i} = \text{viagra}|y_\lambda = \text{spam}) = 0.6 \tag{22}$$
$$P(x_{\lambda,i} = \text{excel}|y_\lambda = \text{spam}) = 0.1 \tag{23}$$
$$P(x_{\lambda,i} = \text{the}|y_\lambda = \text{spam}) = 0.3 \tag{24}$$

and

$$\begin{align}
P(x_{\lambda,i} = \text{viagra}|y_\lambda = \text{not spam}) &= 0.05 \tag{25}\\
P(x_{\lambda,i} = \text{excel}|y_\lambda = \text{not spam}) &= 0.65 \tag{26}\\
P(x_{\lambda,i} = \text{the}|y_\lambda = \text{not spam}) &= 0.3 \tag{27}
\end{align}$$

The "naive" part is that we take each word to be IID, so the probability of an email, x_λ, is a product over the probability of each word,

$$P(\mathbf{x}_\lambda|y_\lambda) = \prod_i P(x_{\lambda,i}|y_\lambda) \tag{28}$$

To compute the probability that an email is spam given the words in it, we need Bayes theorem,

$$\mathrm{P}(y_\lambda|\mathbf{x}_\lambda) \propto \mathrm{P}(y_\lambda)\,\mathrm{P}(\mathbf{x}_\lambda|y_\lambda) \tag{29}$$

Where $\mathrm{P}(y_\lambda)$ is the prior, describing whether spam emails are more or likely than real emails,

$$\begin{align}
P(y_\lambda = \text{not spam}) &= 0.2 \tag{30}\\
P(y_\lambda = \text{spam}) &= 0.8 \tag{31}
\end{align}$$

The probability of the email, given it is spam is:

$$\begin{align}
P(\mathbf{x}_\lambda = \text{viagra, the viagra}|y_\lambda = \text{spam}) =& P(x_{\lambda,1} = \text{viagra}|y_\lambda = \text{spam}) \tag{32}\\
& P(x_{\lambda,2} = \text{the}|y_\lambda = \text{spam}) \tag{33}\\
& P(x_{\lambda,3} = \text{viagra}|y_\lambda = \text{spam}) \tag{34}\\
& \tag{35}
\end{align}$$

[41]:

```
like_spam = 0.6*0.3*0.6
like_spam
```

[41]:

```
0.108
```

The probability of the email, give it is not spam is:

$$\begin{align}
P(\mathbf{x}_\lambda = \text{viagra, the viagra}|y_\lambda = \text{not spam}) =& P(x_{\lambda,1} = \text{viagra}|y_\lambda = \text{not spam}) \tag{36}\\
& P(x_{\lambda,2} = \text{the}|y_\lambda = \text{not spam}) \tag{37}\\
& P(x_{\lambda,3} = \text{viagra}|y_\lambda = \text{not spam}) \tag{38}\\
& \tag{39}
\end{align}$$

[42]:

```
like_notspam = 0.05*0.3*0.05
like_notspam
```

[42]:

```
0.00075
```

We can immediately see that the email is most likely spam.

However, before we can fully reach that conclusion, we need to incorporate the prior and renormalise.

$$\mathrm{P}(y_\lambda|\mathbf{x}_\lambda) \propto \mathrm{P}(y_\lambda)\,\mathrm{P}(\mathbf{x}_\lambda|y_\lambda) \tag{40}$$

[43]:

```
propto_post_spam    = 0.8*like_spam
propto_post_notspam = 0.2*like_notspam

(propto_post_spam, propto_post_notspam)
```

[43]:

```
(0.0864, 0.00015000000000000001)
```

[26]:

```
post_spam    = propto_post_spam    / (propto_post_spam + propto_post_notspam)
post_notspam = propto_post_notspam / (propto_post_spam + propto_post_notspam)

(post_spam, post_notspam)
```

[26]:

```
(0.9982668977469671, 0.0017331022530329290)
```