Solutions 1

Jumping Rivers

Building a first model

The **jrpytorch** package has some concentric circle data which can be loaded with

```
import jrpytorch

X, y = jrpytorch.datasets.load_circles()
```

- Create an exploratory visualisation of the data

```
import matplotlib.pyplot as plt
plt.figure()
plt.scatter(X[:,0],X[:,1],c=y)
plt.show()
```

- Create a class for logistic regression for this problem

```
import torch
import torch.nn as nn

class log_reg(nn.Module):
  def __init__(self):
    super(log_reg,self).__init__()
    self.lin = nn.Linear(in_features=2, out_features=1)

  def forward(self,x):
    return torch.sigmoid(self.lin(x))
```

- Run a simple training routine to fit this model.

```
x_tensor = torch.tensor(X).float()
y_tensor = torch.tensor(y.reshape(-1,1)).float()
model = log_reg()
loss_fn = nn.BCELoss()
optimiser = torch.optim.SGD(model.parameters(), lr = 0.1)

for epoch in range(2000):
  optimiser.zero_grad()
  output = model(x_tensor)
  loss = loss_fn(output,y_tensor)
  loss.backward()
  optimiser.step()
```

- How many predictions did you get correct?

```
sum((model(x_tensor).detach().numpy()[:,0] > 0.5) == y)
```

```
## 102
```

- Assuming that your model object is called `model`, you can visualise the predicted probability space with the following code.

```
import numpy as np
x1 = np.linspace(-1.5,1.5,100)
grid = np.array([(x,y) for x in x1 for y in x1])
output = model(torch.from_numpy(grid).float()).detach().numpy()
plt.figure()
plt.pcolormesh(x1, x1,output.reshape(100,100))
plt.scatter(X[:,0],X[:,1],c=y, edgecolor = 'black')
plt.xlim([-1.5,1.5])
plt.ylim([-1.5,1.5])
plt.show()
```

- It is very hard to classify this dataset with logistic regression using only the input variables provided. Can you improve performance by introducing some additional features.

```
X_new = np.hstack([X,X*X])

class log_reg(nn.Module):
  def __init__(self):
    super(log_reg,self).__init__()
    self.lin = nn.Linear(in_features=4, out_features=1)

  def forward(self,x):
    return torch.sigmoid(self.lin(x))

x_tensor = torch.tensor(X_new).float()
y_tensor = torch.tensor(y.reshape(-1,1)).float()
model = log_reg()
loss_fn = nn.BCELoss()
optimiser = torch.optim.SGD(model.parameters(), lr = 0.1)

for epoch in range(2000):
  optimiser.zero_grad()
  output = model(x_tensor)
  loss = loss_fn(output,y_tensor)
  loss.backward()
  optimiser.step()
```

```
x1 = np.linspace(-1.5,1.5,100)
grid = np.array([(x,y) for x in x1 for y in x1])
grid = np.hstack([grid,grid*grid])
output = model(torch.from_numpy(grid).float()).detach().numpy()
plt.figure()
plt.pcolormesh(x1, x1,output.reshape(100,100))
plt.scatter(X[:,0],X[:,1],c=y, edgecolor = 'black')
plt.xlim([-1.5,1.5])
plt.ylim([-1.5,1.5])
plt.show()

sum((model(x_tensor).detach().numpy()[:,0] > 0.5) == y)
```

Optional

- Create a flexible logistic regression class that could be integrated with a `sklearn.Pipeline` for predicting a binary output

```
class log_reg(nn.Module):
  def __init__(self, nin, bias = True):
    super(log_reg,self, nin).__init__()
    self.lin = nn.Linear(in_features=nin, out_features=1, bias=bias)

  def forward(self,x):
    return torch.sigmoid(self.lin(x))

from sklearn.base import BaseEstimator, ClassifierMixin
import inspect
from sklearn.metrics import accuracy_score

class torch_logistic(BaseEstimator, ClassifierMixin):
  def __init__(self, input_dim = 2,  bias = True, num_epochs = 1000, learning_rate = 0.1):
    self._model = None
    args, _, _, values = inspect.getargvalues(inspect.currentframe())
    values.pop('self')
    for arg, val in values.items():
      setattr(self,arg,val)

  def _build_model(self):
    self._model = log_reg(self.input_dim, self.bias)

  def _reshape(self,y):
    if len(y.shape) == 1:
      return y.shape(-1,1)
```

```
  return y

def _train_model(self,X,y):
  torch_x = torch.tensor(X).float()
  torch_y = torch.tensor(self._reshape(y)).float()
  loss_fn = nn.BCELoss()
  optimiser = torch.optim.SGD(self._model.parameters(), lr = self.learning_rate)

  for epoch in range(self.num_epochs):
    optimiser.zero_grad()
    output = self._model(torch_x)
    loss = loss_fn(train_output,torcy_y)
    loss.backward()
    optimiser.step()

def fit(self,X,y):
  self._build_model()
  self._train_model(X,y)
  return self

def predict(self,X,y = None):
  torch_x = torch.tensor(X).float()
  return self._model(torch_x).detach().numpy().ravel()

def score(self,X,y,sample_weight = None):
  y_pred = self.predict(X)
  return accuracy_score(y, (y_pred > 0.5).astype(int))
```