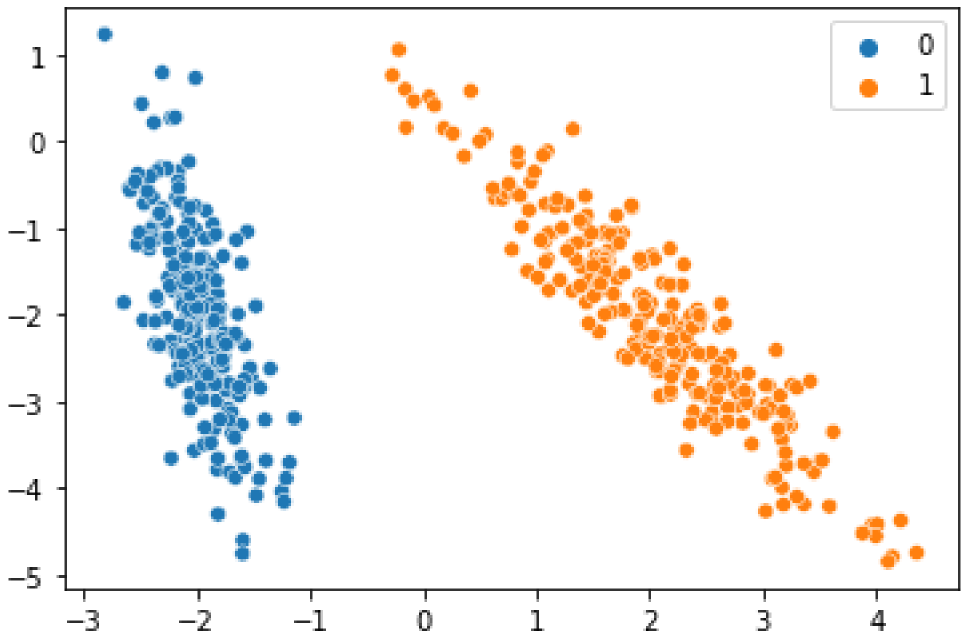
exampl1

September 14, 2023

[125]:
```
from sklearn.datasets import make_classification
import seaborn as sns
import numpy as np
# Generate Clean data
X,y = make_classification(n_samples=500, n_features=2,␣
 ↪n_informative=2,n_redundant=0, n_repeated=0, n_classes=2,␣
 ↪n_clusters_per_class=1,class_sep=2,flip_y=0,weights=[0.5,0.5],␣
 ↪random_state=17)#

sns.scatterplot(x=X[:,0],y=X[:,1],hue=y);
```



[126]:
```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X,
                                                    y,
```

```
                                                        test_size=0.2, # 20% test,␣
↪80% train
                                                        random_state=42) # make the␣
↪random split reproducible

len(X_train), len(X_test), len(y_train), len(y_test)
```

[126]: (400, 100, 400, 100)

[24]:

```
[127]: import torch
X_train = torch.from_numpy(X_train).type(torch.float)
y_train = torch.from_numpy(y_train).type(torch.float)
X_test = torch.from_numpy(X_test).type(torch.float)
y_test =torch.from_numpy(y_test).type(torch.float)
```

```
[128]: #Building a model
import torch
from torch import nn

# Make device agnostic code
device = "cuda" if torch.cuda.is_available() else "cpu"
device
```

[128]: 'cpu'

```
[138]: class myModel(nn.Module):
    def __init__(self):
        super().__init__()
        # 2. Create 2 nn.Linear layers capable of handling X and y input and␣
↪output shapes
        self.layer_1 = nn.Linear(in_features=2, out_features=4) # takes in 2␣
↪features (X), produces 5 features
        self.layer_2 = nn.Linear(in_features=4, out_features=1) # takes in 5␣
↪features, produces 1 feature (y)

    # 3. Define a forward method containing the forward pass computation
    def forward(self, x):
        # Return the output of layer_2, a single feature, the same shape as y
        return self.layer_2(self.layer_1(x)) # computation goes through layer_1␣
↪first then the output of layer_1 goes through layer_2

# 4. Create an instance of the model and send it to target device
model_0 = myModel().to(device)
model_0
```

[138]:
```
myModel(
  (layer_1): Linear(in_features=2, out_features=4, bias=True)
  (layer_2): Linear(in_features=4, out_features=1, bias=True)
)
```

[139]:
```
state_dict = model_0.state_dict()
print(state_dict)
```

```
OrderedDict([('layer_1.weight', tensor([[ 0.5406,  0.5869],
        [-0.1657,  0.6496],
        [-0.1549,  0.1427],
        [-0.3443,  0.4153]])), ('layer_1.bias', tensor([ 0.6233, -0.5188,
0.6146,  0.1323])), ('layer_2.weight', tensor([[ 0.3694,  0.0677,  0.2411,
-0.0706]])), ('layer_2.bias', tensor([0.3854]))])
```

[131]:
```
#Setup loss function and optimizer
# Create a loss function
# loss_fn = nn.BCELoss() # BCELoss = no sigmoid built-in
loss_fn = nn.BCEWithLogitsLoss() # BCEWithLogitsLoss = sigmoid built-in

# Create an optimizer
optimizer = torch.optim.SGD(params=model_0.parameters(),
                            lr=0.1)
```

[ ]:
```

```

[132]:
```
# Calculate accuracy (a classification metric)
def accuracy_fn(y_true, y_pred):
    correct = torch.eq(y_true, y_pred).sum().item() # torch.eq() calculates␣
 ↪where two tensors are equal
    acc = (correct / len(y_pred)) * 100
    return acc
```

[133]:
```
# Train model
# View the frist 5 outputs of the forward pass on the test data
y_logits = model_0(X_test.to(device))[:5]
y_logits
```

[133]:
```
tensor([[ 1.1118],
        [ 1.1551],
        [ 0.2263],
        [-0.1948],
        [-0.0858]], grad_fn=<SliceBackward0>)
```

[134]:
```
# Use sigmoid on model logits
y_pred_probs = torch.sigmoid(y_logits)
y_pred_probs
```

[134]: tensor([[0.7525],
        [0.7605],
        [0.5563],
        [0.4515],
        [0.4786]], grad_fn=<SigmoidBackward0>)

[162]:
```
# Find the predicted labels (round the prediction probabilities)
y_preds = torch.round(y_pred_probs)

# In full
y_pred_labels = torch.round(torch.sigmoid(model_0(X_test.to(device))[:5]))

# Check for equality
print(torch.eq(y_preds.squeeze(), y_pred_labels.squeeze()))

# Get rid of extra dimension
y_preds.squeeze()
```

tensor([True, True, False, False, False])

[162]: tensor([1., 1., 1., 0., 0.], grad_fn=<SqueezeBackward0>)

[163]:
```
import torch
torch.manual_seed(42)
epochs = 5

# Put data to target device

# Build training and evaluation loop
for epoch in range(epochs):
    ### Training
    model_0.train()

    # 1. Forward pass (model outputs raw logits)
    y_logits = model_0(X_train).squeeze() # squeeze to remove extra `1`␣
↪dimensions, this won't work unless model and data are on same device
    y_pred = torch.round(torch.sigmoid(y_logits)) # turn logits -> pred probs␣
↪-> pred labls

    # 2. Calculate loss/accuracy
    # loss = loss_fn(torch.sigmoid(y_logits), # Using nn.BCELoss you need torch.
↪sigmoid()
    #                y_train)
    loss = loss_fn(y_logits, # Using nn.BCEWithLogitsLoss works with raw logits
                   y_train)
    #print(y_logits)
    acc = accuracy_fn(y_true=y_train,
                      y_pred=y_pred)
```

```
    # 3. Optimizer zero grad
    optimizer.zero_grad()

    # 4. Loss backwards
    loss.backward()

    # 5. Optimizer step
    optimizer.step()
    ### Testing
    model_0.eval()
    with torch.inference_mode():
        # 1. Forward pass
        test_logits = model_0(X_test).squeeze()
        test_pred = torch.round(torch.sigmoid(test_logits))
        # 2. Caculate loss/accuracy
        test_loss = loss_fn(test_logits,
                            y_test)
        test_acc = accuracy_fn(y_true=y_test,
                               y_pred=test_pred)


    # Print out what's happening every epochs
    print(f"Epoch: {epoch} | Loss: {loss:.5f}, Accuracy: {acc:.2f}% | Test loss:
 ↪ {test_loss:.5f}, Test acc: {test_acc:.2f}%")
    state_dict = model_0.state_dict()
    print(state_dict)
```

```
Epoch: 0 | Loss: 0.54902, Accuracy: 86.00% | Test loss: 0.53172, Test acc:
91.00%
OrderedDict([('layer_1.weight', tensor([[ 0.5406,  0.5869],
        [-0.1657,  0.6496],
        [-0.1549,  0.1427],
        [-0.3443,  0.4153]])), ('layer_1.bias', tensor([ 0.6233, -0.5188,
0.6146,  0.1323])), ('layer_2.weight', tensor([[ 0.3694,  0.0677,  0.2411,
-0.0706]])), ('layer_2.bias', tensor([0.3854]))])
Epoch: 1 | Loss: 0.54902, Accuracy: 86.00% | Test loss: 0.53172, Test acc:
91.00%
OrderedDict([('layer_1.weight', tensor([[ 0.5406,  0.5869],
        [-0.1657,  0.6496],
        [-0.1549,  0.1427],
        [-0.3443,  0.4153]])), ('layer_1.bias', tensor([ 0.6233, -0.5188,
0.6146,  0.1323])), ('layer_2.weight', tensor([[ 0.3694,  0.0677,  0.2411,
-0.0706]])), ('layer_2.bias', tensor([0.3854]))])
Epoch: 2 | Loss: 0.54902, Accuracy: 86.00% | Test loss: 0.53172, Test acc:
91.00%
OrderedDict([('layer_1.weight', tensor([[ 0.5406,  0.5869],
        [-0.1657,  0.6496],
```

```
        [-0.1549,  0.1427],
        [-0.3443,  0.4153]])), ('layer_1.bias', tensor([ 0.6233, -0.5188,
0.6146,  0.1323])), ('layer_2.weight', tensor([[ 0.3694,  0.0677,  0.2411,
-0.0706]])), ('layer_2.bias', tensor([0.3854]))])
Epoch: 3 | Loss: 0.54902, Accuracy: 86.00% | Test loss: 0.53172, Test acc:
91.00%
OrderedDict([('layer_1.weight', tensor([[ 0.5406,  0.5869],
        [-0.1657,  0.6496],
        [-0.1549,  0.1427],
        [-0.3443,  0.4153]])), ('layer_1.bias', tensor([ 0.6233, -0.5188,
0.6146,  0.1323])), ('layer_2.weight', tensor([[ 0.3694,  0.0677,  0.2411,
-0.0706]])), ('layer_2.bias', tensor([0.3854]))])
Epoch: 4 | Loss: 0.54902, Accuracy: 86.00% | Test loss: 0.53172, Test acc:
91.00%
OrderedDict([('layer_1.weight', tensor([[ 0.5406,  0.5869],
        [-0.1657,  0.6496],
        [-0.1549,  0.1427],
        [-0.3443,  0.4153]])), ('layer_1.bias', tensor([ 0.6233, -0.5188,
0.6146,  0.1323])), ('layer_2.weight', tensor([[ 0.3694,  0.0677,  0.2411,
-0.0706]])), ('layer_2.bias', tensor([0.3854]))])
```

[101]:
```
loss = nn.CrossEntropyLoss()
output = loss(input, target)
output.backward()
```

[73]:
```
print(input,'\n',target,'\n',output)
```

```
tensor([[ 0.3367,  0.1288,  0.2345,  0.2303, -1.1229],
        [-0.1863,  2.2082, -0.6380,  0.4617,  0.2674],
        [ 0.5349,  0.8094,  1.1103, -1.6898, -0.9890]], requires_grad=True)
 tensor([0, 4, 3])
 tensor(2.4607, grad_fn=<NllLossBackward0>)
```