Pytorch: Classification

- Predict Labels
- Categories for these labels ⇒ Classes

Encodings

- input and output need to be numeric
- categorical data ⇒ Numeric format (encoding)

Two methods

1. Label Encoding
2. One hot encoding

1. Label Encoding

- when categories have any meaningful order
- eg: letter grade

```
df['letter_grade'] = df['letter_grade'].map({'A':1,'B':2,'C':3,
```

or

```
df['Letter_Grade'] = df['Letter_Grade'].replace(
    {'A':4,
    'B':3,
    'C':2,
```

```
    'D':1,
    'F':0})
```

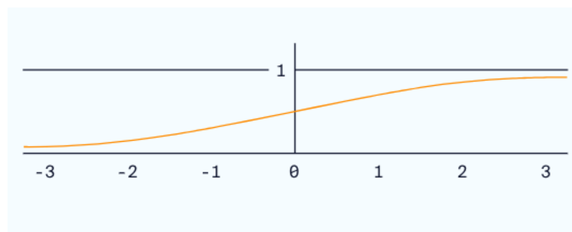
2. One Hot Encoding

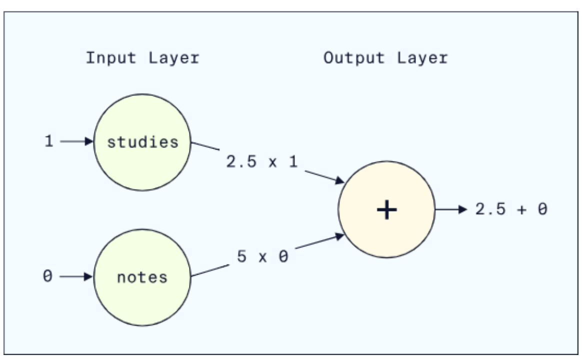
- Binary column for each entry

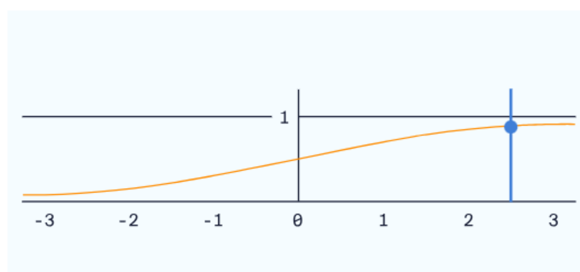
```
df = pd.get_dummies(df, columns =['High_school_type'],dtype=int
```

Sigmoids and Thresholds

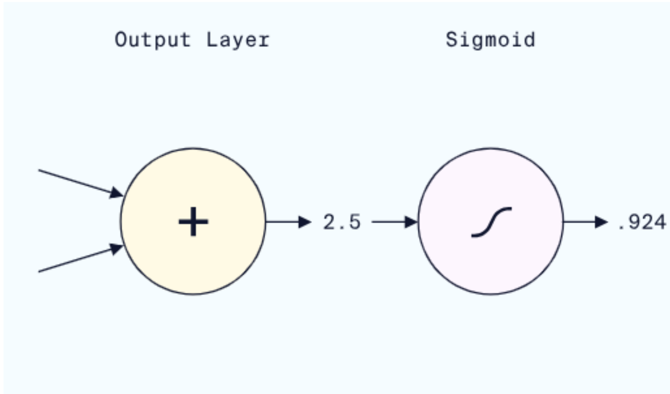
$$\text{sigmoid}(x) = \frac{1}{1 + e^{-x}}$$







0.924



o/p is 2.5, we need 1 or 0

convert 2.5 ⇒ probability (0→1) [cant be 0 or 1, just btw them]

```
# import Python's math module to access the exponential function
import math

# define sigmoid
def sigmoid(x):
    return 1 / (1 + math.exp(-x))
```

Threshold

- usually 0.5 (can change)
- above it 1 else 0
-

```
classification = int(probability>threshold)
#probability>threshold: Bool
```

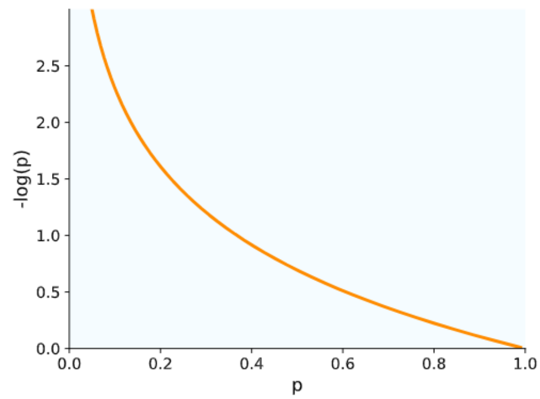
Binary Cross Entropy Loss (BCELoss)

- for Binary Classification Problems

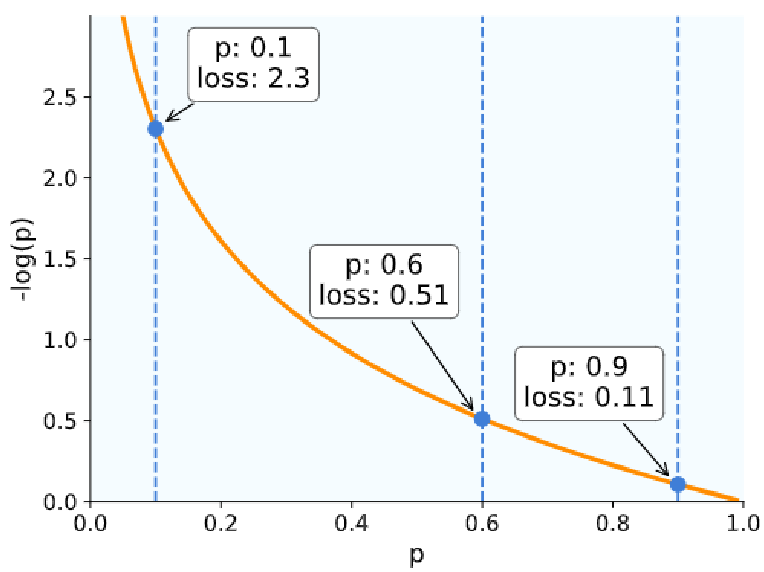
Case 1: true classification `1`

- probability p

$$BCELoss(p) = -log(p)$$



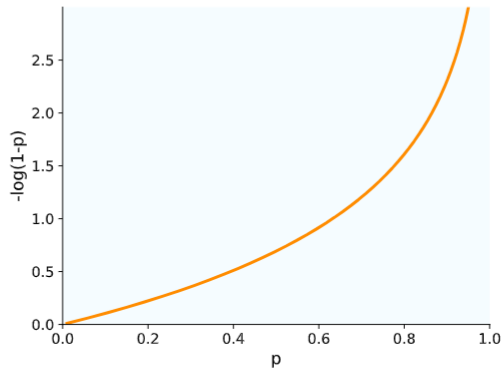
More penalty for lower p



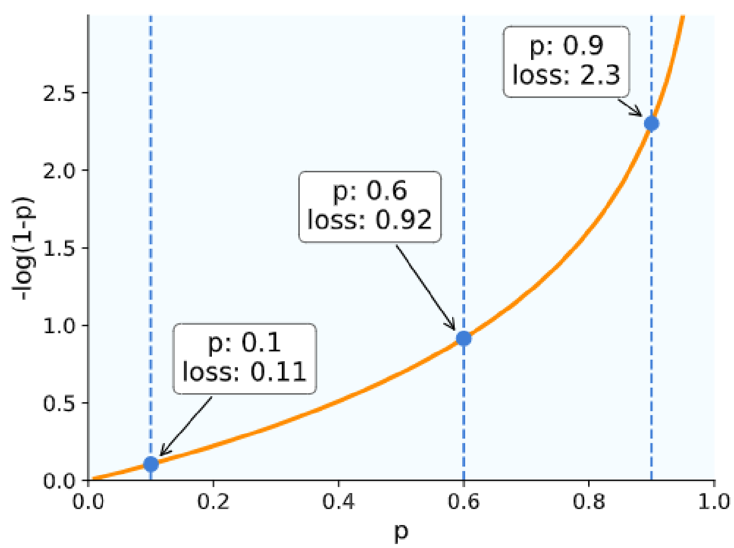
💡 As our predicted probability gets further away from `1`, and thus more and more wrong, the loss increases (indicating larger and larger error).

Case 2: true classification `0`

$$BCELoss(p) = -log(1-p)$$



Probabilities close to `0` now result in very low loss, since `0` is now the true classification



Combining them:

1. Split the data with true classification 1 and true classification 0

2. true classification 1: -log(p)
3. true classification 0: -lob(1-p)
4. collect all loss and find average loss

```
def BCELoss(p,y):
    if y == 1: #if the true classification is 1
        return -np.log(p)
    else: # if the true classification is 0
        return -np.log(1-p)
p = .5
```

```
import torch
from torch import nn

# create an instance of BCELoss
loss = nn.BCELoss()

# create a tensor with an output probability
p = torch.tensor([0.7], dtype = torch.float)
# create a tensor with the actual classification
y = torch.tensor([0], dtype=torch.float)

# define loss_value
loss_value = loss(p,y)
```

Training

Steps of training a NN:

1. Split Train and Test
2. Initialize Loss
3. Initialize Optimizer (+lr)

4. Number of epochs
5. run loop on epochs

```
#for my practice

from sklearn.model_selection import train_test_split
import torch
from torch import nn as nn
from torch import optim as optim

X_train, X_test,y_train,y_test = train_test_split(X,y,test_size
#note X and y are converted to tensors

loss = nn.BCELoss()
optimizer = optim.Adam(model.parameters(), lr = 0.01)

epochs = 100
for i in range(1,epochs+1):
    preds = model(X_train)
    bceloss = loss(preds,y_train)
    bceloss.backward()
    optimizer.step()
    optimzer.zero_grad()
```

```
optimizer = optim.SGD(
    model.parameters(),
    lr=0.001)q
```

SGD: Stochastic Gradient Descent

- instead of using the entire training set, we take randomly few points

Accuracy:

- Loss tells about the “correctness” of probabilty

- Accuracy will the the “correctness” of 1 and 0
-

$$\text{Accuracy} = \frac{\#\text{ of correct predictions}}{\#\text{ of predictions}}$$

```
from sklearn.metrics import accuracy_score

# use a threshold of .5 to predict classifications
predicted_labels = (predictions >= 0.5).int()

# calculate accuracy
accuracy = accuracy_score(y_train, predicted_labels)
```

Full Example:

```
loss = nn.BCELoss()
optimizer = optim.SGD(model.parameters(), lr=0.001)

epochs = 500
for i in range(1, epochs+1):
    predictions = model(X_train)
    lossBCE = loss(predictions, y_train)
    lossBCE.backward()
    optimizer.step()
    optimizer.zero_grad()

    if i%100 ==0:
        preds_01 = int(predictions>=0.5)
        accuracy = accuracy_score(y_train, preds_01)
```

```
        print(f"epoch {i}, BCELoss = {lossBCE}, accuracy={accura
```

Evaluation

1. Accuracy

	Predict	Actual
FP	1	0
FN	0	1
TP	1	1
TN	0	0

The **accuracy**, in this context, is

$$\frac{\text{Correct predictions}}{\text{All predictions}} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}}$$

2. Precision

- Emphasizes False Positive
- eg: I want to give more attention to failed students. But my model marked a failed student (actual =0) as 1(predicted =1) false positive
-

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

Precision being 50% means that half of our model's positive predictions were false.

💡 higher false positives ⇒ Lower precision

3. Recall

- Emphasis on false negatives
- $$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

recall 100% ⇒ all positives were actually positives (?)

> 💡 Higher false negative ⇒ lower Recall

4. F1 score

- harmonic mean of precision and recall
- Balances the concern for False Positives and False Negatives
- $$\text{F1} = \frac{2 * \text{Precision} * \text{Recall}}{\text{Precision} + \text{Recall}}$$

> 💡 F1 is often a good first evaluation metric to try.

```
from sklearn.metrics import accuracy_score, precision_score, rec

accuracy = accuracy_score(y_test, predicted_labels)
```

```
precision = precision_score(y_test, predicted_labels)
recall = recall_score(y_test, predicted_labels)
f1 = f1_score(y_test, predicted_labels)
```

Multiclass Models

- in Binary class we too probabilities between 0 and 1
- In multiclass it is difficult to manage multiple classes to approximate to 0 and 1
- what we do is :
    - hostel : yes 1; no 0
    - home : yes 1; no 0
    - PG : yes 1; no 0

  (similar to one hot encoding)

  wont work when there is overlap between labels (home and hostel) **labels are independent**

Softmax

- We can't use sigmoid on the last layer
    - Sigmoid operates on each node
    - As an example, here's what sigmoid output could look like:
        - **dorm output**: .9
        - **family output**: .8
        - **independent output**: .4

      Interpreted as probabilities, this output means that our model predicts a 90% probability the student lives in the dorm and an 80% probability that the student lives at home with their family. doesnt make sense

      -
- `sofmax`

-
- **dorm output**: .4
- **family output**: .36
- **independent off-campus output**: .24

All three probabilities now sum to `1`, or `100%`, which is much easier to interpret. We can now say that our model predicts a 40% probability of living in the dorms, 36% probability of living with family, and 24% probability of living off-campus.

- `softmax` is in `pytorch` but can be built in , in some cases
  - `nn.Softmax`

WHAT SOFTMAX DOES?

normalization factor:

$$e^{0.9} + e^{0.8} + e^{0.4}$$

- take each o/p
- take exp
- add them

next:

$$\frac{e^{0.9}}{e^{0.9} + e^{0.9} + e^{0.9}}$$

```
import numpy as np

softmax_9 = np.exp(.9) / (np.exp(.9) + np.exp(.8) + np.exp(.4))

## YOUR SOLUTION HERE ##

softmax_8 = np.exp(0.8)/(np.exp(0.9)+np.exp(0.8)+np.exp(0.4))
softmax_4 = np.exp(0.4)/(np.exp(0.9)+np.exp(0.8)+np.exp(0.4))
# show output
print(np.round(softmax_9,2))
print(np.round(softmax_8,2))
print(np.round(softmax_4,2))
```

Argmax

eg:

- 0: always takes notes
- 1: almost always takes notes
- 2: sometimes takes notes
- 3: never takes notes

output of model:

```
[[-0.1480,  0.0144, -0.0396,  0.0027],
[-0.1065, -0.0680,  0.0191,  0.0787]]
#tensor
```

apply softmax:

```
[[0.2246, 0.2641, 0.2502, 0.2611],
[0.2285, 0.2375, 0.2591, 0.2750]]
#probabilities
```

- row1: max prob = 0.2641 ⇒ 1:almost always takes notes
- row2 : max prob=0.2750 ⇒ 3: never takes notes

apply argmax

```
torch.argmax(softmax_output,dim=1) #dim=1 tells find max in ROW
```

⇒ We dont necessarily need softmax, unless:

- we want to see "how confident" the final predictions is)

⇒ pytorch automatically does it for us

```
raw_output = torch.tensor([[0.1320, 0.0160, 0.9614, 0.9919],
        [0.7180, 0.7303, 0.6234, 0.1197],
        [0.8757, 0.2045, 0.1977, 0.3845],
        [0.8934, 0.5677, 0.1377, 0.6420],
        [0.4017, 0.8363, 0.1119, 0.6557]], dtype = torch.float)


argmax_output = torch.argmax(raw_output, dim=1)

argmax_output
```

Multiclass: Train and evaluate

Eg:

- **Below Average:** grades 0 and 1
- **Average:** grades 2 and 3
- **Above Average:** grades 4 and 5

⇒ Pytorch indexing starts from 0 so.

- **Below Average** will be `0`
- **Average** will be `1`
- **Above Average** will be `2`

⇒ pytorch labels need to be `dtype = torch.long`

```
y = torch.tensor(df['Performance_outcomes'].values, dtype = torc
```

steps:

1. **Forward pass:** training data through model
2. **Loss**: between training o/p (predictions) and labels
3. **Backward Pass**: gradients of loss function
4. **Optimizer** : adjust weights and biases
5. **Iterate**: reset gradients and repeat

Cross Entropy Loss Function

```
loss = nn.CrossEntropyLoss()
```

1. applies softmax (sortof) to get prob distrib
2. takes -ve log of the prob

💡 In sequential model, last should be nn.linear(<>, categories) and no activation fn
⇒ algorithm applies a version of softmax to the output of the network when performing optimization.

```
from sklearn.metrics import accuracy_score

# set a random seed - do not modify
torch.manual_seed(42)

# define a model
torch.manual_seed(42)
model = nn.Sequential(
    nn.Linear(55, 240),
    nn.ReLU(),
    nn.Linear(240, 110),
    nn.ReLU(),
    nn.Linear(110, 3)
)

## YOUR SOLUTION HERE ##
loss = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.01)

# Train the neural network
num_epochs = 1000
for epoch in range(num_epochs):
    predictions = model(X_train)
    CELoss = loss(predictions,y_train)
    CELoss.backward()
    optimizer.step()
    optimizer.zero_grad()
```

```
    ## DO NOT MODIFY ##
    # keep track of the loss and accuracy during training
    if (epoch + 1) % 100 == 0:
        predicted_labels = torch.argmax(predictions, dim=1)
        accuracy = accuracy_score(y_train, predicted_labels)
        print(f'Epoch [{epoch+1}/{num_epochs}], CELoss: {CELoss
```

```
from sklearn.metrics import accuracy_score, classification_repor

model.eval()
with torch.no_grad():
    ## YOUR SOLUTION HERE ##
    predictions = model(X_test)
    predicted_labels = torch.argmax(predictions,dim=1)
    accuracy = accuracy_score(predicted_labels, y_test)
    report = classification_report(y_test, predicted_labels)

# show output - do not modify
print(f'Accuracy: {accuracy.item():.4f}')
print(report)
```

- the macro average gives equal weight to each class (arithmetic mean)

- the weighted average weights classes with a larger # of observations (their support) higher taking into account class imbalances

avoid overfitting:

- change the training features
- change the number of nodes in the hidden layers
- increase/decrease the number of training epochs

- test different activation functions
- test different optimizers and learning rates

> 💡 Binary cross entropy loss expects a 2d tensor (y)
> Cross entropy loss expects a 1d tensor (y) and (long)