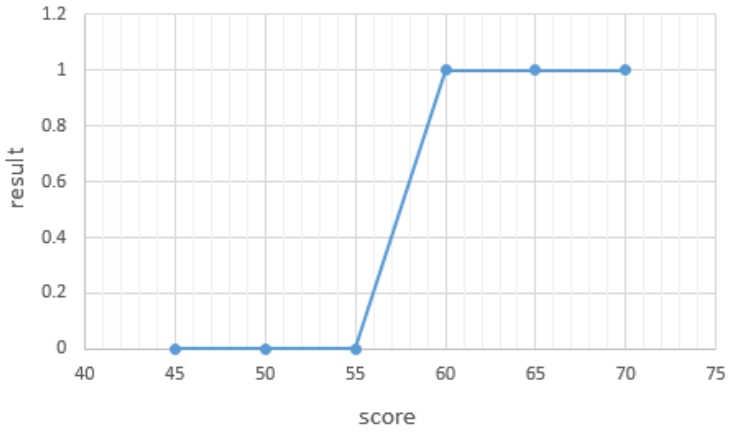
Logistic Regression

상태	Basic ML
담당자	

Logistic Regression

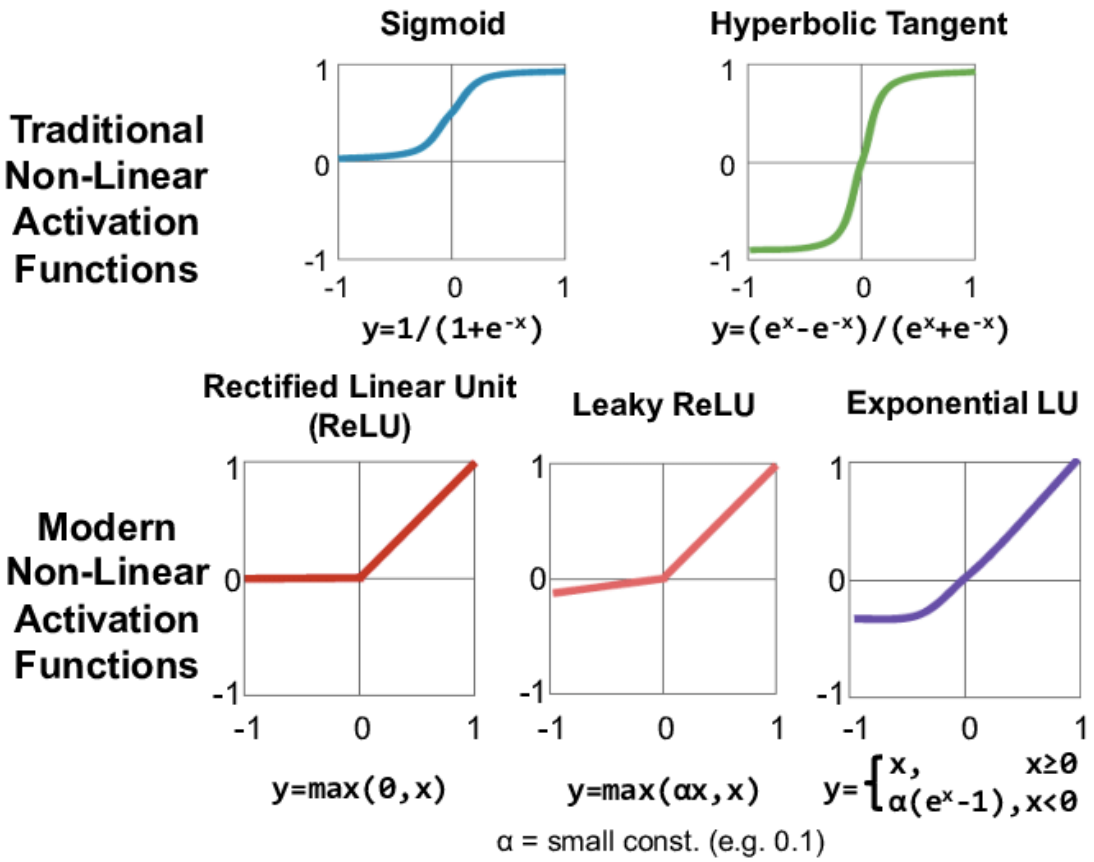
> Binary classification을 위한 모델로, 여러 설명 변수들로 사건이 일어날 확률을 예측하고
> → threshold을 조절하여 positive class와 negative class를 어떻게 나눌지 설정



input X에 대한 output이 binary한 경우, 그래프를 그려 보면 알파벳 S와 비슷한 개형이 나온다. 이는 단순 선형 회귀 $H(x) = Wx + b$로 잘 트래킹하기 어렵다. (따라서 좋은 분류기가 될 수 없다)

그러므로 단순 선형 회귀를 S자 곡선으로 잘 피팅시켜 줄 수 있는 어떤 함수 f를 통과시켜 hypothesis function을 $H(X) = f(Wx + b)$으로 수정한다. 이때의 함수 f가 시그모이드 함수이다. (이렇게 마지막 단계에서 사용되어 최종적인 출력을 가능하게 하는 함수를 '활성화 함수'라 한다.)

▼ 활성화 함수



선형 시스템을 아무리 깊게 쌓더라도 비선형 문제를 해결할 수 없다. 따라서 활성화 함수로는 반드시 비선형 함수가 사용(sigmoid, relu, softmax, ...)된다.

Logistic Regression을 사용한 Binary Classification

Logistic regression을 사용해 binary classification 문제를 해결하는 방식은 다음과 같다.

1. 매 에포크(의 미니배치)마다
2. $X \in \mathbb{R}^{m \times d}$인 input에 weight $W \in \mathbb{R}^{d \times 1}$을 곱한 다음 (선형 회귀식)
3. $f = \frac{1}{1+e^{-w^T x}} = \frac{1}{1+e^{-XW}}$을 통과시켜 0과 1 사이의 확률값을 리턴 (시그모이드 함수)
4. threshold 0.5를 기준으로 $m \times 1$의 형태로 결과를 예측, target 변수인 y와 비교
5. Gradient Descent로 Weight을 업데이트

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
```

```
# For reproducibility
torch.manual_seed(1)

x_data = [[1, 2], [2, 3], [3, 1], [4, 3], [5, 3], [6, 2]]
y_data = [[0], [0], [0], [1], [1], [1]]

x_train = torch.FloatTensor(x_data)
y_train = torch.FloatTensor(y_data)
```

```
class BinaryClassifier(nn.Module):
    def __init__(self):
        super().__init__()
        self.linear = nn.Linear(2, 1)
        self.sigmoid = nn.Sigmoid()

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

6 x 2 크기의 텐서 `x`가 `nn.Linear()`를 지나며(가설식 $H(X) = Wx + b$을 구현하기 위해) 2 x 1 을 통과해 6 x 1 크기의 `y`와 같은 크기가 나오게 된다. 이는 다시 우리가 원하는 작업인 `nn.Sigmoid()`을 지나게 된다.

```
optimizer = optim.SGD(model.parameters(), lr=1)
nb_epochs = 100

for epoch in range(nb_epochs + 1):# 오잉 이러면 epoch + 1 번 학습하게 되는 거 아닌가? range(1, nb_epochs + 1)이 맞지 않나..
    # Hypothesis 구현
    hypothesis = model(x_train) # 선형 회귀식이 시그모이드 함수에 대입

    # Cost 계산
    cost = F.binary_cross_entropy(hypothesis, y_train)

    # Optimizer로 hypothesis 개선
    optimizer.zero_grad()
    cost.backward() # 미분을 통한 gradient descent
    optimizer.step()

    # 10번마다 로그 출력
    if epoch % 10 == 0:
        prediction = hypothesis >= torch.FloatTensor([0.5]) # 1이라고 예측하는 것이 True/False
        correct_prediction = prediction.float() == y_train # 1이 맞은 예측
        accuracy = correct_prediction.sum().item() / len(correct_prediction)
        print('Epoch : {:4d}/{} Cost : {:.6f} Accuracy : {:2.2f}'.format(
            epoch, nb_epochs, cost.item(), accuracy*100))
```

🤔 `correct_prediction = prediction.float() == y_train`

- `prediction`은 boolean 처럼 보인데 어떻게 `.float()`에도 에러가 나지 않을까?
- `float(prediction)`으로 바꾸면 `ValueError: only one element tensors can be converted to Python scalars` 에러가 발생한다. `prediction`에서 기대되는 값이 tensor라는 힌트를 얻을 수 있었다.
- `type(prediction)`을 통해 타입을 확인하면 `torch.Tensor`이고, prediction을 `print`해 보면 이미 0과 1로 구성된 텐서(구체적으로 ByteTensor)이다.
- 따라서 `prediction.float`으로 접근 가능하며 `y_train`과 비교했을 때에도 그 결과가 boolean 값이 아니라 0과 1의 텐서로 나온다.

▼ 왜 상속받은 자식 클래스에서 `super().__init__`을 사용하는가?

`nn.Module`을 상속받은 클래스 `BinaryClassifier` 역시 정의할 때 `__init__(self)`을 사용해 필요한 초기 세팅을 불러온다.

자식 클래스가 `nn.Module`을 상속받은 이유는 부모 클래스에서 사용된 메서드들(과 변수들)을 모두 가져와 사용하기 위해서이다.

그런데 부모 클래스인 `nn.Module` 역시 `__init__(self)`로 필요한 초기 세팅이 있을 것이다. 이를 무시하고 자식 클래스가 부모 클래스를 상속받으면 부모 클래스의 `__init__()`은 자식 클래스의 `__init__()`에 의해 오버라이딩 된다.

따라서, 자식 클래스의 `__init__()` 메서드 과정에서 부모 클래스의 `__init__()` 메서드의 변수들(초기 세팅들)을 가져오기 위해서는 자식 클래스의 `__init__()` 메서드 안에 `super().__init__()`을 입력하면 된다.

만약 부모 클래스의 `__init__()` 메서드에 argument들이 있다면 자식 클래스의 `__init__()` 메서드와 `super().__init__()` 메서드에는 `**kwargs`를 인자로 전달해 주어야 한다.