Multi-Label Classification

Project : classification model (Logistic Regression/OVR, MLP)

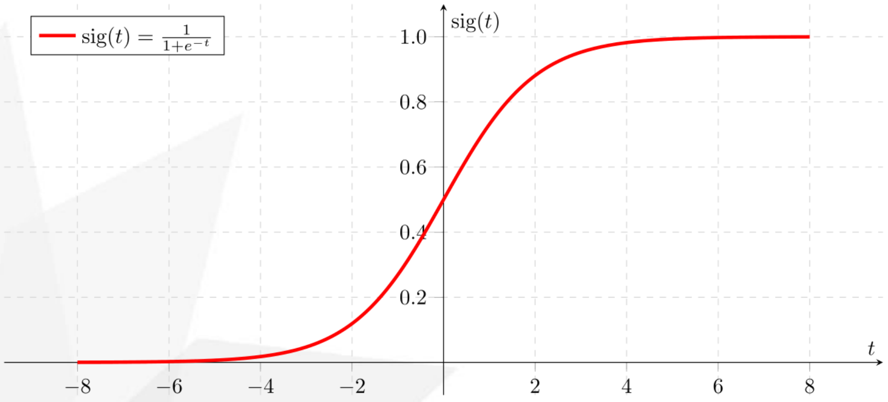
Seunghan Lee (CSE-URP)

20.01.22(Wed)

Goal

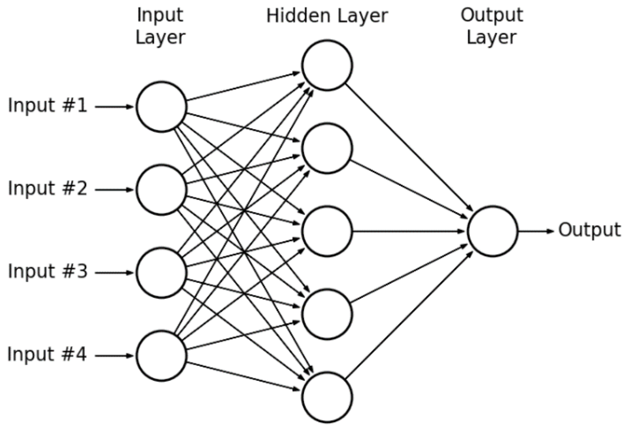
1) Implement classification model using **Logistic Regression / OVR** & **MLP**

2) Evaluate with various metrics (**precision**, **recall**, **accuarcy**, **F1-score**)



Logistic Regression

https://miro.medium.com/max/2400/1*RqXFpiNGwdiKBWyLJc_E7g.png



Multi Layer Perceptron

https://www.researchgate.net/profile/Mohamed_Zahran6/publication/303875065/figure/fig4/AS:371118507610123@1465492955561/A-hypothetical-example-of-Multilayer-Perceptron-Network.png

Contents

1. Introduction

Brief overview of **Logistic Regression** / **OVR**& **MLP**

1. Introduction

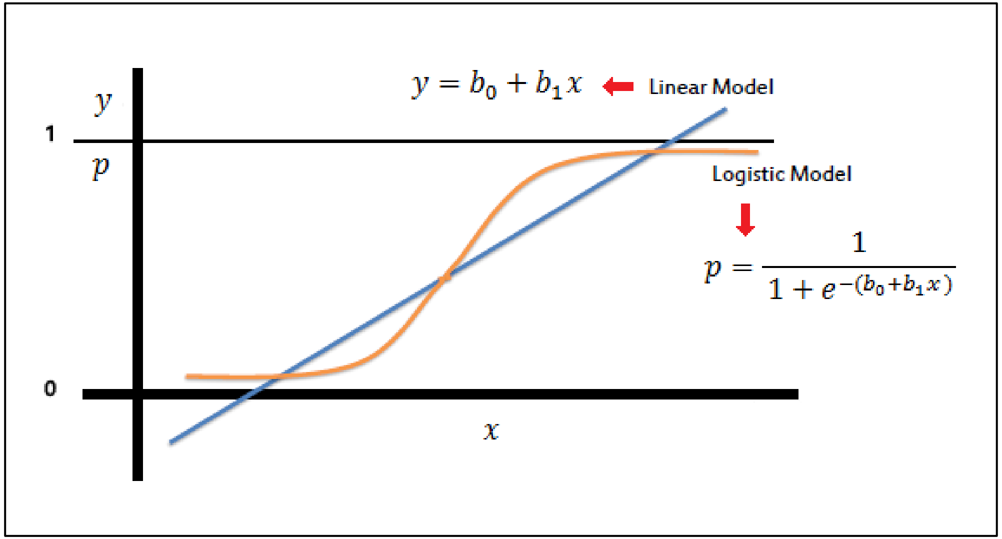
1. Logistic Regression

- Regression (X) **Classification** (O)
- Sigmoid Function (output value : **0 ~ 1**)
- used as a “**binary**” classifier

Find the best b0, b1

that best fits the data!

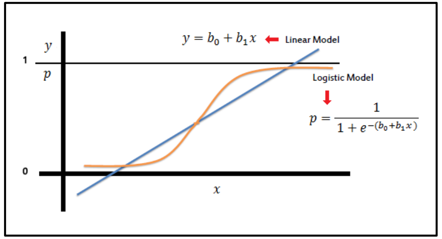
http://juangabrielgomila.com/wp-content/uploads/2015/04/LogReg_1.png

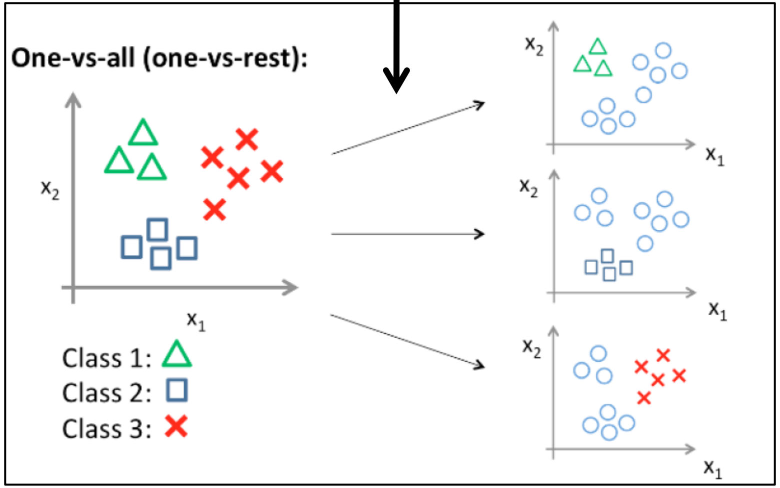


1. Introduction

OVR (One-Versus-Rest)

- used in **MULTI CLASS** Classification
- compare **ONE** group with all the **REST**
  (rest : treat the others as same one group)
- need **“C” Classifiers**
  (C : unique number of classes)





https://houxianxu.github.io/images/logisticRegression/4.png

1. Introduction

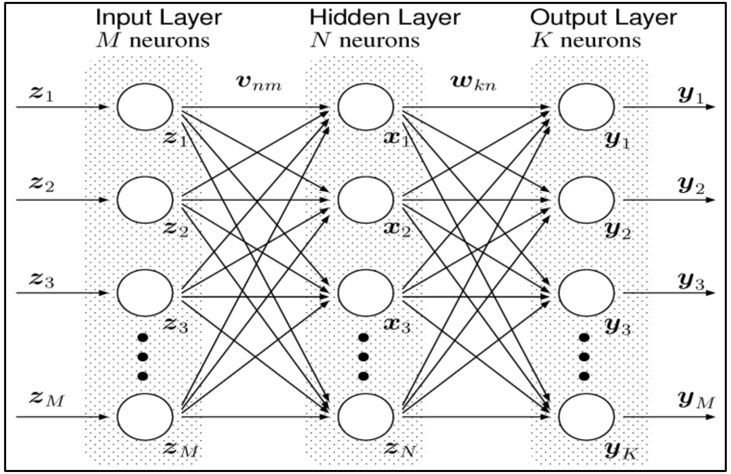
2. Multi Layer Perceptron

- find the weight using **back propagation**
- need **large datasets**
- be aware of **overfitting** (many parameters)

https://www.mdpi.com/information/information-03-00756/article_deploy/html/images/information-03-00756-g002.png

Find the best w, b

that best fits the data!



2. Implementation

1) Karate

(binary classification : Logistic Regression & MLP)

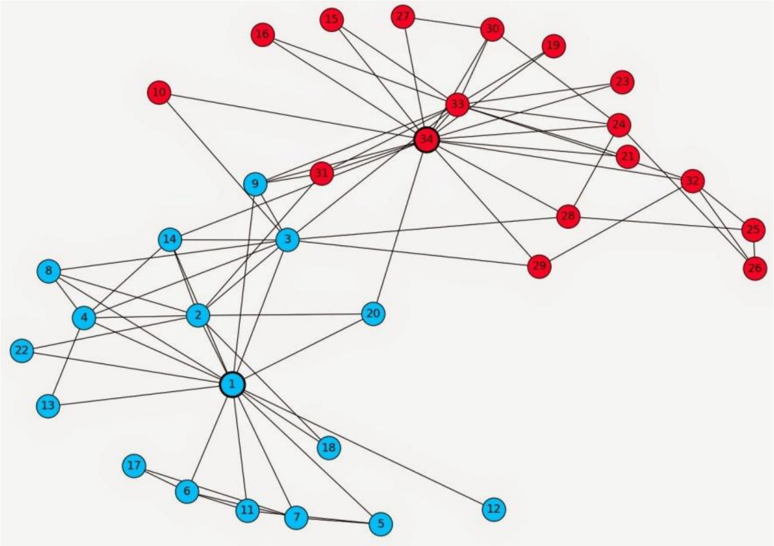
2) Other data

(multi-class (over 2) classification : OVR)

1. Karate Dataset

1. Import Dataset
2. Define Functions
3. Modeling

   (1. Logistic Regression & 2. MLP)
4. Prediction
5. Evaluation



https://bookdown.org/omarlizardo/_main/images/karate.jpg

Implementation 1. Karate

1. Import Dataset

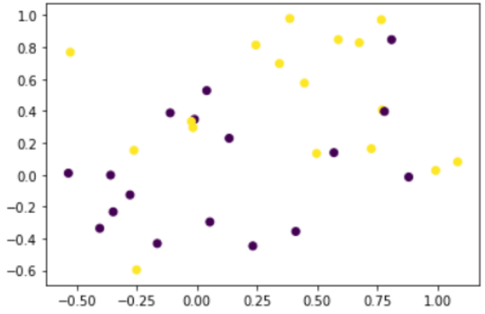
```
ev = pd.read_csv('embedded_vector.csv')
```

```
data = ev[['X','Y','Color']]
data.columns = ['x1','x2','class']
data = data.sample(frac=1) # to shuffle

data.head()
```

	x1	x2	class
20	0.569728	0.138210	0
31	0.232140	-0.446658	0
30	0.040150	0.527334	0
1	-0.251698	-0.596485	1
11	0.587872	0.846916	1

```
plt.scatter(data['x1'], data['x2'], c=data['class'])
plt.show()
```



Balanced dataset

```
In [6]: data['class'].value_counts()

Out[6]: 1    17
        0    17
        Name: class, dtype: int64
```

34 rows, 2 independent variables

```
In [7]: data.shape

Out[7]: (34, 3)
```

Embedded Vector (into 2-dim) of Karate Dataset

Implementation 1. Karate

2. Define Functions

1) **train_test_split** : divide the dataset into two parts (train & test)

2) **mul** : matrix multiplication

3) **sigmoid** : sigmoid activation function

4) **standard_scaler** : scale the columns into Gaussian Distribution

5) **loss_func** : use LogLoss as a loss(cost) function

2. Define Functions

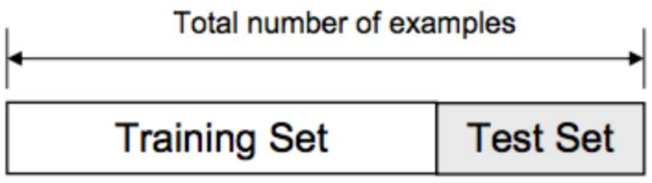
1) train_test_split

```
def train_test_split(data,test_ratio):
    data.iloc[:,[0,1]] = standard_scaler(data.iloc[:,[0,1]])
    test_index = np.random.choice(len(data),int(len(data)*test_ratio),replace=False)
    train = data[~data.index.isin(test_index)]
    test = data[data.index.isin(test_index)]

    train_X = np.array(train)[:,[0,1]]
    train_y = np.array(train)[:,[2]].flatten()
    test_X = np.array(test)[:,[0,1]]
    test_y = np.array(test)[:,[2]].flatten()
    return train_X,train_y, test_X,test_y
```

[process]

1) scale every columns (into Gaussian Distribution)
2) choose random sample x%
3) use x% as test dataset, (100-x)%as train dataset
4) separate X & Y



https://miro.medium.com/max/2272/1*-8_kogvwmL1H6ooN1A1tsQ.png

2. Define Functions

2) mul & 3) sigmoid

```
def mul(W,b,x):
    return np.dot(x,W)+b

def sigmoid(x):
    k = 1 / (1 + np.exp(-x))
    return k[:,0]
```

4) standard_scaler

```
def standard_scaler(x):
    mean = np.mean(x)
    std = np.std(x)
    return (x-mean)/std
```

$$z = \frac{x-\mu}{\sigma}$$

5) loss_fun

```
def loss_func(y_hat,y):
    total_loss = np.mean(y*np.log(y_hat) + (1-y)*np.log(1-y_hat))
    return -total_loss
```

$$H_p(q) = -\frac{1}{N}\sum_{i=1}^{N} y_i \cdot log(p(y_i)) + (1-y_i)\cdot log(1-p(y_i))$$

Binary Cross-Entropy / Log Loss

use log loss (binary cross entropy) as

a loss function

Implementation 1. Karate

3. Modeling

1. Logistic Regression

```
def logreg(x,y,epoch,lr):
    W = np.random.rand(x.shape[1],1)
    b = np.random.rand(1)

    for ep in range(epoch+1):
        Z = mul(W,b,x)
        y_hat = sigmoid(Z)
        loss = loss_func(y_hat,y)
        dw = np.matmul(x.T,y_hat-y)/x.shape[0]
        db = np.sum(y_hat-y)

        W = W-lr*dw.reshape(-1,1)
        b = b-lr*db

        if ep % 10000 == 0:
            print('epoch :',ep,' loss :',loss)

    return W,b
```

2. Multi Layer Perceptron

```
class NN:
    def __init__(self,input_num,output_num,hidden_depth,num_neuron,
                 activation=Sigmoid, activation2=Softmax):
        def init_var(in_,out_):
            weight = np.random.normal(0,0.1,(in_,out_))
            bias = np.zeros((out_,))
            return weight,bias

    ## 1-1. Hidden Layer
        self.sequence = list() # lists to put neurons
        W,b = init_var(input_num,num_neuron)
        self.sequence.append(Neuron(W,b,activation))

        for _ in range(hidden_depth-1):
            W,b = init_var(num_neuron,num_neuron)
            self.sequence.append(Neuron(W,b,activation)) # default : Sigmoid

    ## 1-2. Output Layer
        W,b = init_var(num_neuron,output_num)
        self.sequence.append(Neuron(W,b,activation2)) # default : Softmax

    def __call__(self,x):
        for layer in self.sequence:
            x = layer(x)
        return x

    def calc_grad(self,loss_fun):
        loss_fun.dh = loss_fun.grad()
        self.sequence.append(loss_fun)

        for i in range(len(self.sequence)-1, 0, -1):
            L1 = self.sequence[i]
            L0 = self.sequence[i-1]

            L0.dh = _m(L0.grad(), L1.dh)
            L0.dW = L0.grad_W(L1.dh)
            L0.db = L0.grad_b(L1.dh)

        self.sequence.remove(loss_fun)
```

Implementation 1. Karate

3. Modeling

1. Logistic Regression

```
def logreg(x,y,epoch,lr):
    W = np.random.rand(x.shape[1],1)
    b = np.random.rand(1)

    for ep in range(epoch+1):
        Z = mul(W,b,x)
        y_hat = sigmoid(Z)
        loss = loss_func(y_hat,y)
        dw = np.matmul(x.T,y_hat-y)/x.shape[0]
        db = np.sum(y_hat-y)

        W = W-lr*dw.reshape(-1,1)
        b = b-lr*db

        if ep % 10000 == 0:
            print('epoch :',ep,' loss :',loss)

    return W,b
```

1. Initialize W & b

2. Find the probability

$$h_\theta(X) = \frac{1}{1+e^{-\theta^T X}} = \Pr(Y=1 \mid X;\theta)$$

3. Update weights

$$\begin{aligned}
\frac{\partial}{\partial\theta_j}\ell(\theta) &= \left(y\frac{1}{g(\theta^T x)} - (1-y)\frac{1}{1-g(\theta^T x)}\right)\frac{\partial}{\partial\theta_j}g(\theta^T x) \\
&= \left(y\frac{1}{g(\theta^T x)} - (1-y)\frac{1}{1-g(\theta^T x)}\right)g(\theta^T x)(1-g(\theta^T x)\frac{\partial}{\partial\theta_j}\theta^T x \\
&= \left(y(1-g(\theta^T x)) - (1-y)g(\theta^T x)\right)x_j \\
&= \left(y-h_\theta(x)\right)x_j
\end{aligned}$$

Implementation 1. Karate

3. Modeling

1. Input :

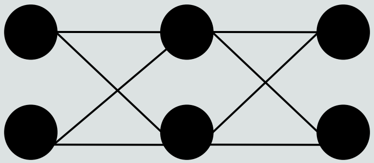
- number of neurons in input layer
- number of neurons in hidden layer
- number of neurons in output layer
- number of hidden layers
- activation function 1 & 2

2. Network Architecture :

```
NeuralNet_10 = NN(2,2,1,2,activation=Sigmoid, activation2=Softmax)
loss_fun = LogLoss()
EPOCH = 16
```

(input)

2D data



(output)

Probability

2. Multi Layer Perceptron

```
class NN:
    def __init__(self,input_num,output_num,hidden_depth,num_neuron,
                 activation=Sigmoid, activation2=Softmax):
        def init_var(in_,out_):
            weight = np.random.normal(0,0.1,(in_,out_))
            bias = np.zeros((out_,))
            return weight,bias

    ## 1-1. Hidden Layer
        self.sequence = list() # lists to put neurons
        W,b = init_var(input_num,num_neuron)
        self.sequence.append(Neuron(W,b,activation))

        for _ in range(hidden_depth-1):
            W,b = init_var(num_neuron,num_neuron)
            self.sequence.append(Neuron(W,b,activation)) # default : Sigmoid

    ## 1-2. Output Layer
        W,b = init_var(num_neuron,output_num)
        self.sequence.append(Neuron(W,b,activation2)) # default : Softmax

    def __call__(self,x):
        for layer in self.sequence:
            x = layer(x)
        return x

    def calc_grad(self,loss_fun):
        loss_fun.dh = loss_fun.grad()
        self.sequence.append(loss_fun)

        for i in range(len(self.sequence)-1, 0, -1):
            L1 = self.sequence[i]
            L0 = self.sequence[i-1]

            L0.dh = _m(L0.grad(), L1.dh)
            L0.dW = L0.grad_W(L1.dh)
            L0.db = L0.grad_b(L1.dh)

        self.sequence.remove(loss_fun)
```

Implementation 1. Karate

4. Prediction

1. Logistic Regression

(1) Split the data

```
train_X_10, train_y_10, test_X_10, test_y_10 = train_test_split(data,0.9)
train_X_30, train_y_30, test_X_30, test_y_30 = train_test_split(data,0.7)
train_X_50, train_y_50, test_X_50, test_y_50 = train_test_split(data,0.5)
train_X_70, train_y_70, test_X_70, test_y_70 = train_test_split(data,0.3)
```

(3) Make a prediction based on cut-off value of 0.5

```
def predict(test_X,W,b):
    preds = []
    for i in sigmoid(np.dot(test_X, W) + b):
        if i>0.5:
            preds.append(1)
        else:
            preds.append(0)
    return np.array(preds)
```

```
y_pred_10 = predict(test_X_10, W_10,b_10)
y_pred_30 = predict(test_X_30, W_30,b_30)
y_pred_50 = predict(test_X_50, W_50,b_50)
y_pred_70 = predict(test_X_70, W_70,b_70)
```

```
y_pred_50
```

```
array([1, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0])
```

(2) Train the model

10%

```
W_10,b_10 = logreg(train_X_10,train_y_10,40000,0.001)
```

```
epoch : 0   loss : 0.6593308365741114
epoch : 10000   loss : 0.16243315108234851
epoch : 20000   loss : 0.08608784835902628
epoch : 30000   loss : 0.05783537826077004
epoch : 40000   loss : 0.04337391029813504
```

30%

```
W_30,b_30 = logreg(train_X_30,train_y_30,40000,0.001)
```

```
epoch : 0   loss : 0.595785050592402
epoch : 10000   loss : 0.5277547221980337
epoch : 20000   loss : 0.5264002682692209
epoch : 30000   loss : 0.5260746776716059
epoch : 40000   loss : 0.5259887552029092
```

50%

```
W_50,b_50 = logreg(train_X_50,train_y_50,40000,0.001)
```

```
epoch : 0   loss : 0.4885542591865314
epoch : 10000   loss : 0.29181192527506544
epoch : 20000   loss : 0.2689832066669257
epoch : 30000   loss : 0.25930977959826224
epoch : 40000   loss : 0.2540498366830823
```

70%

```
W_70,b_70 = logreg(train_X_70,train_y_70,40000,0.001)
```

```
epoch : 0   loss : 0.6410069002890594
epoch : 10000   loss : 0.5650531338346519
epoch : 20000   loss : 0.5616843341978129
epoch : 30000   loss : 0.5614450045147054
epoch : 40000   loss : 0.5614244991790818
```

Implementation 1. Karate

4. Prediction

1. Logistic Regression

(1) Split the data

```
train_X_10, train_y_10, test_X_10, test_y_10 = train_test_split(data,0.9)
train_X_30, train_y_30, test_X_30, test_y_30 = train_test_split(data,0.7)
train_X_50, train_y_50, test_X_50, test_y_50 = train_test_split(data,0.5)
train_X_70, train_y_70, test_X_70, test_y_70 = train_test_split(data,0.3)
```

(2) Train the model

10%

```
W_10,b_10 = logreg(train_X_10,train_y_10,40000,0.001)

epoch : 0   loss : 0.6593308365741114
epoch : 10000   loss : 0.16243315108234851
epoch : 20000   loss : 0.08608784835902628
epoch : 30000   loss : 0.057835378260770044
epoch : 40000   loss : 0.043373910298135046
```

30%

```
W_30,b_30 = logreg(train_X_30,train_y_30,40000,0.001)

epoch : 0   loss : 0.595785050592402
epoch : 10000   loss : 0.5277547221980337
epoch : 20000   loss : 0.5264002682692209
epoch : 30000   loss : 0.5260746776716059
epoch : 40000   loss : 0.5259887552029092
```

50%

```
W_50,b_50 = logreg(train_X_50,train_y_50,40000,0.001)

epoch : 0   loss : 0.48855425918653145
epoch : 10000   loss : 0.29181192527506544
epoch : 20000   loss : 0.2689832066669257
epoch : 30000   loss : 0.25930977959826224
epoch : 40000   loss : 0.2540498366830823
```

70%

```
W_70,b_70 = logreg(train_X_70,train_y_70,40000,0.001)

epoch : 0   loss : 0.6410069002890594
epoch : 10000   loss : 0.5650531338346519
epoch : 20000   loss : 0.5616843341978129
epoch : 30000   loss : 0.5614450045147054
epoch : 40000   loss : 0.5614244991790818
```

(3) Make a prediction based on cut-off value of 0.5

```
def predict(test_X,W,b):
    preds = []
    for i in sigmoid(np.dot(test_X, W) + b):
        if i>0.5:
            preds.append(1)
        else:
            preds.append(0)
    return np.array(preds)
```

```
y_pred_10 = predict(test_X_10, W_10,b_10)
y_pred_30 = predict(test_X_30, W_30,b_30)
y_pred_50 = predict(test_X_50, W_50,b_50)
y_pred_70 = predict(test_X_70, W_70,b_70)
```

```
y_pred_50

array([1, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0])
```

Implementation 1. Karate

4. Prediction

2. Multi Layer Perceptron

(1) Split the data

```
train_X_10, train_y_10, test_X_10, test_y_10 = train_test_split(data,0.9)
train_X_30, train_y_30, test_X_30, test_y_30 = train_test_split(data,0.7)
train_X_50, train_y_50, test_X_50, test_y_50 = train_test_split(data,0.5)
train_X_70, train_y_70, test_X_70, test_y_70 = train_test_split(data,0.3)
```

(Same data preprocessing as Logistic Regression)

(2) Train the model

case 4) train 70%

```
NeuralNet_70 = NN(2,2,1,2,activation=Sigmoid, activation2=Softmax) # inp
loss_fun = LogLoss()
EPOCH = 16

loss_per_epoch_70 = []

for epoch in range(EPOCH):
    for i in range(train_X_70.shape[0]):
        loss = GD(NeuralNet_70,train_X_70[i],train_y_70[i],loss_fun,0.1)
    loss_per_epoch_70.append(loss)
    print('Epoch {} : Loss {}'.format(epoch+1, loss))
```

```
Epoch 1 : Loss 4.898567335894803
Epoch 2 : Loss 4.058881620390265
Epoch 3 : Loss 3.0708254147867633
Epoch 4 : Loss 2.187728069268146
Epoch 5 : Loss 1.5616007602003
Epoch 6 : Loss 1.159184638676766
Epoch 7 : Loss 0.9003590294018504
Epoch 8 : Loss 0.7273030175086146
Epoch 9 : Loss 0.606219333725357
Epoch 10 : Loss 0.5179050523121949
Epoch 11 : Loss 0.4511672046828421
Epoch 12 : Loss 0.399216673960884
Epoch 13 : Loss 0.3577633372026356
Epoch 14 : Loss 0.32399144597221563
Epoch 15 : Loss 0.2959893482892117
Epoch 16 : Loss 0.27241995779724454
```

Implementation 1. Karate

5. Evaluation

```python
def Metrics(pred,actual):
    TP,TN,FP,FN = 0,0,0,0
    for i in range(len(pred)):
        if pred[i]*actual[i]==1:
            TP +=1
        elif pred[i]>actual[i]:
            FP +=1
        elif pred[i]<actual[i]:
            FN +=1
        else:
            TN +=1

    accuracy = (TP+TN) / (TP+TN+FP+FN)
    precision = TP / (TP+FP)
    recall = TP / (TP+FN)
    F1_score = 2*(precision*recall)/(precision+recall)
    return accuracy,precision,recall,F1_score
```

		Actual Values	
		Positive (1)	Negative (0)
Predicted Values	Positive (1)	TP	FP
	Negative (0)	FN	TN

Return 4 metrics

1) Accuracy
2) Precision
3) Recall
4) F1-Score

5. Evaluation

1. Logistic Regression

```
print('Training Dataset 10%')
acc, pre, rec, f1 = Metrics(y_pred_10,test_y_10)
print('accuarcy :', np.round(acc,3))
print('precision :', np.round(pre,3))
print('recall :', np.round(rec,3))
print('f1-score :', np.round(f1,3))
```

```
Training Dataset 10%
accuarcy : 0.667
precision : 0.692
recall : 0.6
f1-score : 0.643
```

10%

```
print('Training Dataset 30%')
acc, pre, rec, f1 = Metrics(y_pred_30,test_y_30)
print('accuarcy :', np.round(acc,3))
print('precision :', np.round(pre,3))
print('recall :', np.round(rec,3))
print('f1-score :', np.round(f1,3))
```

```
Training Dataset 30%
accuarcy : 0.478
precision : 0.421
recall : 0.889
f1-score : 0.571
```

30%

```
print('Training Dataset 50%')
acc, pre, rec, f1 = Metrics(y_pred_50,test_y_50)
print('accuarcy :', np.round(acc,3))
print('precision :', np.round(pre,3))
print('recall :', np.round(rec,3))
print('f1-score :', np.round(f1,3))
```

```
Training Dataset 50%
accuarcy : 0.647
precision : 0.7
recall : 0.7
f1-score : 0.7
```

50%

```
print('Training Dataset 70%')
acc, pre, rec, f1 = Metrics(y_pred_70,test_y_70)
print('accuarcy :', np.round(acc,3))
print('precision :', np.round(pre,3))
print('recall :', np.round(rec,3))
print('f1-score :', np.round(f1,3))
```

```
Training Dataset 70%
accuarcy : 0.8
precision : 0.75
recall : 0.75
f1-score : 0.75
```

70%

Implementation 1. Karate

5. Evaluation

1. Logistic Regression

```
print('Training Dataset 10%')
acc, pre, rec, f1 = Metrics(y_pred_10,test_y_10)
print('accuarcy :', np.round(acc,3))
print('precision :', np.round(pre,3))
print('recall :', np.round(rec,3))
print('f1-score :', np.round(f1,3))
```

```
Training Dataset 10%
accuarcy : 0.667
precision : 0.692
recall : 0.6
f1-score : 0.643
```

10%

```
print('Training Dataset 30%')
acc, pre, rec, f1 = Metrics(y_pred_30,test_y_30)
print('accuarcy :', np.round(acc,3))
print('precision :', np.round(pre,3))
print('recall :', np.round(rec,3))
print('f1-score :', np.round(f1,3))
```

```
Training Dataset 30%
accuarcy : 0.478
precision : 0.421
recall : 0.889
f1-score : 0.571
```

30%

```
print('Training Dataset 50%')
acc, pre, rec, f1 = Metrics(y_pred_50,test_y_50)
print('accuarcy :', np.round(acc,3))
print('precision :', np.round(pre,3))
print('recall :', np.round(rec,3))
print('f1-score :', np.round(f1,3))
```

```
Training Dataset 50%
accuarcy : 0.647
precision : 0.7
recall : 0.7
f1-score : 0.7
```

50%

```
print('Training Dataset 70%')
acc, pre, rec, f1 = Metrics(y_pred_70,test_y_70)
print('accuarcy :', np.round(acc,3))
print('precision :', np.round(pre,3))
print('recall :', np.round(rec,3))
print('f1-score :', np.round(f1,3))
```

```
Training Dataset 70%
accuarcy : 0.8
precision : 0.75
recall : 0.75
f1-score : 0.75
```

70%

5. Evaluation

2. Multi Layer Perceptron

```
In [26]: pred10
Out[26]: array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                0, 0, 0, 0, 0, 0, 0, 0], dtype=int64)

In [27]: pred30
Out[27]: array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                0], dtype=int64)

In [28]: pred50
Out[28]: array([1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1], dtype=int64)

In [29]: pred70
Out[29]: array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0], dtype=int64)
```

TOO SMALL dataset to be

used on Neural Network!

Implication

Too Small Dataset!

Should use most of the data to train the model

(unable to open BlogCatalog)

2. Other Datasets

[Glass Dataset]

- Goal : classify which type the **glass** belongs to!
- Number of classes : 6

	Ri	Na	Mg	Al	Si	K	Ca	Ba	Fe	Type
102	1.51820	12.62	2.76	0.83	73.81	0.35	9.42	0.0	0.20	2
144	1.51660	12.99	3.18	1.23	72.97	0.58	8.81	0.0	0.24	2
94	1.51629	12.71	3.33	1.49	73.28	0.67	8.24	0.0	0.00	2
35	1.51567	13.29	3.45	1.21	72.74	0.56	8.57	0.0	0.00	1
24	1.51720	13.38	3.50	1.15	72.85	0.50	8.43	0.0	0.00	1

[Wine Dataset]

- Goal : classify which type the **wine** belongs to!
- Number of classes : **3**

	0	1	2	3	4	5	6	7	8	9	10	11	12	type
138	13.49	3.59	2.19	19.5	88.0	1.62	0.48	0.58	0.88	5.70	0.81	1.82	580.0	2
32	13.68	1.83	2.36	17.2	104.0	2.42	2.69	0.42	1.97	3.84	1.23	2.87	990.0	0
48	14.10	2.02	2.40	18.8	103.0	2.75	2.92	0.32	2.38	6.20	1.07	2.75	1060.0	0
10	14.10	2.16	2.30	18.0	105.0	2.95	3.32	0.22	2.38	5.75	1.25	3.17	1510.0	0
5	14.20	1.76	2.45	15.2	112.0	3.27	3.39	0.34	1.97	6.75	1.05	2.85	1450.0	0

2. Other Datasets

all the other settings are same as [1.Karate] except…

Binary Classification

Logistic Regression

(1 classifier)

→

Multi-Class Classification

OVR (One-Versus-Rest)

(C classifier)

[Implementation]

1. OVR
2. Confusion Matrix
3. F1-Score

(micro & macro)

Implementation 1. Karate

1. OVR

```python
def OVR(data,test_ratio,epoch,lr):
    train_x,train_y,test_x,test_y = train_test_split(data,test_ratio)
    pred_result = []
    real_result = []
    for index in data['class'].unique():
        train_y2 = (train_y == index).astype(int)
        test_y2 = (test_y == index).astype(int)
        W,b = logreg(train_x,train_y2,epoch,lr)
        y_pred = predict(test_x,W,b)
        pred_result.append(y_pred)
        real_result.append(test_y2)
    pred_OH = (pred_result == np.amax(pred_result,axis=0)).astype('int')
    act_OH = np.concatenate(real_result).ravel().reshape(data.iloc[:,-1].nunique(),-1)
    return pred_OH,act_OH
```

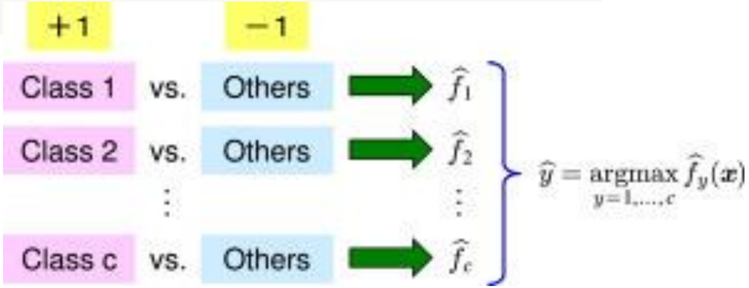
(1) Split the dataset

loop 'C' times (C : number of unique classes)

(2) Return 1 if (class = index)
0 if (class! = index)

(3) Modeling & Training & Predicting

(4) Classify according to =>



Result

```
prediction,actual = OVR(new_data,0.3,100,0.005)
```

```
epoch : 0   loss : 1.012143425232233
epoch : 20   loss : 0.27897828172454686
epoch : 40   loss : 0.26815269797799435
epoch : 60   loss : 0.2674520147626888
epoch : 80   loss : 0.26733675528461476
epoch : 100   loss : 0.2672668927007779
epoch : 0   loss : 0.8958977959857002
epoch : 20   loss : 0.6844183694129793
epoch : 40   loss : 0.6801199682181064
epoch : 60   loss : 0.6762043699273796
epoch : 80   loss : 0.6724611416877259
epoch : 100   loss : 0.668882590984983
epoch : 0   loss : 0.9450494165783728
epoch : 20   loss : 0.8350815753283266
epoch : 40   loss : 0.8185435647036615
epoch : 60   loss : 0.8028498910376841
epoch : 80   loss : 0.7877840275607149
epoch : 100   loss : 0.7733207572259169
epoch : 0   loss : 0.6742882948995139
epoch : 20   loss : 0.36971588004006756
epoch : 40   loss : 0.3554918560868599
```

Classifier 1 : [Class 1] vs [Rest]

Classifier 2 : [Class 2] vs [Rest]

Classifier 3 : [Class 3] vs [Rest]

Classifier 4 : [Class 4] vs [Rest]

2. Confusion Matrix

```python
def confusion_matrix(actual,prediction):
    n = actual.shape[0]
    conf_mat = np.zeros((n,n))
    for i in range(n):
        for j in range(n):
            conf_mat[i][j] += len(np.intersect1d(np.nonzero(actual[i]),np.nonzero(prediction[j])))
    return conf_mat
```

```
In [42]: actual2
Out[42]: array([[1, 1, 0, 0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 1, 1, 1, 0, 0, 1, 0, 0,
                 1, 0, 0, 1, 0, 1, 1, 1, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1,
                 0, 0, 1, 0, 1, 0, 1, 0, 0],
                [0, 0, 0, 1, 1, 1, 0, 1, 1, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 1, 0,
                 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0,
                 0, 1, 0, 0, 0, 0, 0, 1, 0],
                [0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1,
                 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 1, 1, 0, 0, 0, 0, 1, 1, 0,
                 1, 0, 0, 1, 0, 1, 0, 0, 1]])
```

Class 1

Class 2

Class 3

```
In [43]: prediction2
Out[43]: array([[1, 1, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0, 1, 1, 1, 0, 1, 1, 0, 1,
                 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 0, 1, 1, 1,
                 1, 0, 1, 1, 1, 1, 1, 0, 1],
                [0, 0, 0, 1, 1, 1, 0, 1, 1, 1, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 1, 0,
                 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 0, 0,
                 0, 1, 0, 0, 0, 0, 0, 1, 0],
                [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                 0, 0, 0, 0, 0, 0, 0, 0, 0]])
```

Class 1

Class 2

Class 3

Truth

Predicted

	Asphalt	Concrete	Grass	Tree	Building	Total
Asphalt	2385	4	0	1	4	2394
Concrete	0	332	0	0	1	333
Grass	0	1	908	8	0	917
Tree	0	0	0	1084	9	1093
Building	12	0	0	6	2053	2071
Total	2397	337	908	1099	2067	6808

3. F1-Score

Micro 1

pooled	T	F
T	18	9
F	9	45

weight : number of actual class

Macro (average 2 & weighted average 3)

BAC	T	F	CON	T	F	EXP	T	F
T	9	3	T	5	4	T	4	2
F	4	11	F	4	14	F	1	20

3. F1-Score

```python
def f1_scores(con,score):
    # score = 0 : micro / score =1 : macro / score = 2 : weighted macro

    # (1) Micro F1
    if score==0:
        return np.diag(con).sum()/con.sum()
    rec,pre,f1 = [],[],[]

    for i in range(con.shape[0]):
        recall = con[i][i] / con[i].sum()
        precision = con[i][i] / con[:,i].sum()
        f1_score = 2*recall*precision / (recall+precision)
        rec.append(recall)
        pre.append(precision)
        f1.append(f1_score)

    # (2) Macro F1
    if score==1:
        return np.average(f1)

    # (3) Weighted Macro F1
    elif score==2:
        w = [con[x].sum() for x in range(con.shape[0])]
        return np.average(f1,weights=w)
```

Result

Confusion matrix

Glass Data

```
glass_con = confusion_matrix(actual, prediction)
glass_con
```

```
array([[25.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  4.,  0.],
       [ 4.,  0., 21.,  0.,  0.,  0.],
       [ 0.,  0.,  3.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  4.,  0.],
       [ 0.,  0.,  2.,  0.,  1.,  0.]])
```

Wine Data

```
confusion_matrix(actual2, prediction2)
```

```
array([[11.,  0.,  2.],
       [ 2., 15.,  0.],
       [ 6.,  1., 16.]])
```

F1-Score

```
print('Wine Dataset')
print('Micro F1 :',f1_scores(wine_con,0).round(3))
print('Macro F1 (Average) :',f1_scores(wine_con,1).round(3))
print('Macro F1 (Weighted Average) :',f1_scores(wine_con,2).round(3))
```

```
Wine Dataset
Micro F1 : 0.792
Macro F1 (Average) : 0.792
Macro F1 (Weighted Average) : 0.799
```

Thank You!!