writer: Guofeng Cui

---

1. Directory & File

1. *.py* file under *code* directory are python codes.
   1). *DecisionTree .py*: Implement Decision Tree, including train and test.
   2). *NeuralNetwork .py*: Implement Backpropagation Neural Network.
   3). *Execution .py*: User can define evaluation here.
2. Data file
   1). Example data files with *.csv* type are put in *./code/data*.
   2). *DecisionTree .py* and *NeuralNetwork .py* do not specify directory of data file.

2. How to Run Decision Tree

2.1. Supported Input Data

1). Only support classification of **discrete data**,
2). Data and label can be any standard type, including **number, string**.

2.2. Entropy Function Describe

1). infoGain(): calculation of **information gain**.
2). giniIndex(): calculation of **gini index**.
3). User can define their own entropy function, but three parameter needed to be delivered.
   a. **data index**: ID of attributes belong to current node, with form of $[id1, id2, \ldots]$.
   b. **data**: data list, type of $[list1, list2]$. Where $list1$ indicates list of entire attribute name (not id, not for current node), $list2$ indicates list of data, with the form of $[data\_list1, data\_list2, \ldots]$
   c. **data_label**: label of each data, with form of $[label1, label2, \ldots]$

2.3. Train funtion: train(data, data_label, fun)

1). Build Up Decision Tree
2). **data**: data with the same form of **data** in entropy function.
3). **data_label**: label of data, with the same form of **data_label** in entropy function.
4). **fun**: pointer of entropy function.

2.4. Test function: test(data)

1). Prediction class of current data, user should call *train* function to build up tree first.
2). **data**: with the same form of **data** above.

3). return type: return label of data, with the same form of **data_label** above.

2.5. Example to train decision tree and test.

1). **simpleRunDI** function in *Execution. py*
2). Code:

```
def simpleRunDT(self, train_ratio = 0.75, function = 1):
    # split train data and test data
    train_data, test_data, train_data_label, test_data_label = self.rando
mSplit(train_ratio, self.data, self.data_label)

    # train
    dt = tree.DT()
    if function == 0:
        dt.train(data = train_data, data_label = train_data_label, fun =
 dt.infoGain)
    else:
        dt.train(data = train_data, data_label = train_data_label, fun =
 dt.giniIndex)

    # test
    predictions = dt.test(test_data)
    accuracy = sum([1 for label, pred in zip(test_data_label, prediction
s) if label == pred]) / len(predictions)

    print("accuracy is: %f" %(accuracy))
    return accuracy
```

3. How to Run Neural Network

3.1. Support Input Data

1). Only support numerical data and data label.

3.2. Activition Function

1). Activition.sigmoid(data, deri = False):
sigmoid function (deri = False),

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

as well as **derivative sigmoid** (deri = True),

$$\frac{\delta sig(x)}{\delta x} = sig(x)(1 - sig(x))$$

2). Activition.tanh(data, deri = False):
tanh function (deri = False),

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

as well as **derivative tanh** (deri = True),

$$\frac{\delta tanh(x)}{\delta x} = 1 - (tanh(x))^2$$

3). Activition.ReLu(data, deri = False):
ReLu function (deri = False),

$$ReLu(x) = max(0, x)$$

derivative ReLu (deri = True),

$$\frac{\delta ReLu(x)}{\delta x} = \begin{cases} 0 & x <= \\ 1 & x > 0 \end{cases}$$

4). User can define activation function by themself. Two parameters should be included.
   a. **data**: a single number
   b. **deri**: bool type, to define whether to return derivative result.

3.3. Train with One-Epoch train

1). If user needs to define special funtion for each epoch, use **doTrain**, otherwise, see section 3.4.
2). **doTrain(data_list, data_label, pace, init_w = False)**: train for one epoch
data_list: data list, with the form of $[list1, list2, \dots]$
data_label: label of data, with the form of $[label1, label2, \dots]$, each label should be one-hot value, such as $[0, 1, 0]$
pace: update rate of $w = w + \frac{\delta f(x)}{x} \times pace$
init_w: whether to initialize value of weight
3). Guide to use: Define training data and label, define epoches, batch and so on, then for each epoch call *dotrain*. But first, user needs to define layers. Example showed as below:

```
nn.deliverLayer(len(train_data[0]))
nn.activeLayer(8, act.sigmoid)
nn.activeLayer(len(clas_list), act.sigmoid)
nn.softmaxLayer(len(clas_list))
```

Which means, the first layer is input layer, the second and third one is hiddren layer with sigmoid function, the last layer is a softmax.

4). Using such function to train gives user more freedom to modify, for example in *train* function in *Execution .py*

```
def train(self, data, data_label, test_data, test_data_label, clas_list,
 epoches, batches, pace, nn):
    # start train
    cur_pos = 0
    data = copy.copy(data)
    data_label = copy.copy(data_label)
    for epoch in range(epoches):
        # get current batch
        if cur_pos + batches < len(data):
            train_data = copy.copy(data[cur_pos : cur_pos + batches])
            train_data_label = copy.copy(data_label[cur_pos : cur_pos + b
atches])
            cur_pos += batches
        else:
            train_data = copy.copy(data[cur_pos :])
            train_data_label = copy.copy(data_label[cur_pos :])

            # next round
            data, data_label = nn.ranShuf(data, data_label)
            cur_pos = batches - (len(data) - cur_pos)
            train_data += copy.copy(data[: cur_pos])
            train_data_label += copy.copy(data_label[: cur_pos])

        loss = nn.doTrain(train_data, train_data_label, pace)

        if(epoch % 100 == 0):
            # test
            predictions = nn.predict(test_data[1])
            accuracy = sum([1 for label, pred in zip(test_data_label, pre
dictions) if clas_list[label.index(1)] == clas_list[pred.index(1)]]) / le
n(predictions)
            print("epoch: %d, loss: %s, accuracy: %f" %(epoch, sum([abs(l
s/batches) for ls in loss]), accuracy))
```

3.4. Train with defined train function

1). If user only want to train by standard type, just call **train**.
2). **train(self, data, data_label, epoches, batches, pace, init_w = False)**
epoches: number of round of training
batches: number of samples used to train in a round.
3). How to run: both forms of *data* and *data_label* are similar to that of $doTrain$. Just define and call the function.
4). Example to run:

```
nn.deliverLayer(len(train_data[0]))
nn.activeLayer(8, act.sigmoid)
nn.activeLayer(len(clas_list), act.sigmoid)
nn.softmaxLayer(len(clas_list))
```

```
self.train(train_data, train_data_label, test_data, test_data_label, clas
_list, epoches, batch, pace, nn)
# test
predictions = nn.predict(test_data)
accuracy = sum([1 for label, pred in zip(test_data_label, predictions) if
 clas_list[label.index(1)] == clas_list[pred.index(1)]]) / len(prediction
s)
```

3.5. Test

1). **predict(test_data)**: prediction result of test data, return one-hot value
2). **test_data**: same form of data in train.

4. Example

4.1 Execution .py

1). **randomSplit**: split data into train and test
2). **loadFile**: load data from file
3). **simpleRunDT**: example of decision tree implementation
4). **simpleRunNN**: example of neural network implementation, which call **train** to actual run.