CSC 442 Project 4 Report
Runhua Gao

Githublink : https://github.com/RunhuaGao/AI-Projects/tree/master/Project%204

Part 1: Decision Tree

For this part, I implement the Decision tree algorithm on AIMA Figure 18.5, based on the entropy theory to classify the data.

Let me first introduce the structure of my code for this part.

class decisionTree:

core methods:

learning(examples, attr, parent examples): which is exactly what the algorithm on textbook does. Recursively go through all examples in dataset and establish a decision tree.

calH(probability): pass in a probabilty return it's entropy use formula:

-prob*log2(prob)

calEntropy(dataset): calculate the entropy of a dataset, used in learning method. For each data in dataset, the last element is it's type, which also stands for a probability that each type data appears in this dataset

probability = the number of each type in this dataset/dataset'size

calH & calEntropy constructs the entropy function of my decision tree

splitDataByAttr(dataset,attr): split the dataset by attr(here is the index of attr). For example, if the index here is 1 for Iris dataset, this method will return four data subsets corresponds to the S,M,ML,L

calEntropySplitByAttr(data,attr): calculate the entropy after split the dataset pass in by the attr, used for choose a atrribute as a node of decision tree.

chooseattr(data): choose an attribute from data with most information gain(entropy theory)

plurality(data): choose most common type is data, used in learning method

crossvalidation(k): split examples into k piece and do cross validation experiment, return average error rate

class DecisionTreeNode:

core methos:

addbranch(key,value): add a branch for this node, the key must be a string(for Iris dataset), value could be a exact type or another node

eveluatedata(data): used for the root node, return the result of classification of a data based on this current root node.

testdata(dataset): eveluate each data in dataset and compare the result to the target value, retrun the error rate, used in crossvalidation experiment.

class splitData(example,k): auxiliary class to split the examples into k pieces, used for cross validation.

Result analysis and experiment:

For the Iris dataset, the following picture is the screenshot of the decision my code get from original dataset(does not change data's relative order in dataset):

```
curr attribute is 3
current value is S classified as setosa
--- this branch has printed out, splited by attribute 3
current value is ML
    curr attribute is 2
    current value is ML
        curr attribute is 0
        current value is L classified as versicolor
        --- this branch has printed out, splited by attribute 0
        current value is ML
            curr attribute is 1
            current value is ML classified as versicolor
            --- this branch has printed out, splited by attribute 1
            current value is MS classified as versicolor
            --- this branch has printed out, splited by attribute 1
            current value is S classified as versicolor
            --- this branch has printed out, splited by attribute 1
            whole branches has printed out at attribute 1
        --- this branch has printed out, splited by attribute 0
        current value is MS
            curr attribute is 1
            current value is S classified as versicolor
            --- this branch has printed out, splited by attribute 1
            current value is MS classified as versicolor
            --- this branch has printed out, splited by attribute 1
            current value is ML classified as versicolor
            --- this branch has printed out, splited by attribute 1
            whole branches has printed out at attribute 1
        --- this branch has printed out, splited by attribute 0
        current value is S classified as virginica
        --- this branch has printed out, splited by attribute 0
        whole branches has printed out at attribute 0
    --- this branch has printed out, splited by attribute 2
    current value is MS classified as versicolor
    --- this branch has printed out, splited by attribute 2
    current value is L classified as virginica
    --- this branch has printed out, splited by attribute 2
    whole branches has printed out at attribute 2
--- this branch has printed out, splited by attribute 3
current value is MS classified as versicolor
--- this branch has printed out, splited by attribute 3
current value is L classified as virginica
--- this branch has printed out, splited by attribute 3
whole branches has printed out at attribute 3
```

This screenshot shows the structure of decision tree clearly. For each level, it has different numbers of whitespace. For example, the first level is 3, which has four branches called S,ML,MS,L. From the picture we could easily see that for S,ML,L, these three nodes are leaves, which means the decision could directly decide what the type of a iris flower by its fourth attribute in dataset(here 3 is index in dataset, so 3 means flower's fourth attribute). But if a flower's fourth attribute is ML, it should go further of the tree, which is then decided by it's third(index 2),first(index 0) and second(index 1) attribute in order.

After establishing the decision tree, I firstly test it on the shuffled(use random.shuffle) original Irisdataset. The error rate is as following:
Run code as following: sample is the Iris dataset after parsing

```
dt = DecisionTree(sample)
tree = dt.learning(examples=sample,attrs=dt.initialattrs,parentexamples=sample)
shuffle(sample)
tree.testData(sample)
```

Error rate:

```
0.04666666666666667
```

4% error rate, nice performance.

Also test restaurant dataset, get error rate as following:

```
0.0
```

With code:

```
dt = DecisionTree(restaurant)
tree = dt.learning(examples=restaurant, attrs=dt.initialattrs, parentexamples=restauran
shuffle(restaurant)
tree.testData(restaurant)
```

Here I think the zero error rate could be attributed to the size of restaurant dataset, as it has only 12 items, which makes it difficult to clarify how our algorithm performs.

I also test on the cars dataset from UCI library:

url: https://archive.ics.uci.edu/ml/machine-learning-databases/car/car.data

The error rate by doing test on shuffled original dataset:

```
Error rate is:  0.0
```

With Code:

```
dt = DecisionTree(cars)
tree = dt.learning(examples=cars, attrs=dt.initialattrs, parentexamples=cars)
shuffle(cars)
tree.testData(cars)
```

After doing this simple experiment, I do the cross validation on these two data set, split Iris data to 10 piece and split cars dataset into 12 piece(it has 1728 items, split it into 12 piece, each piece has 1728/12=144 items)

When I firstly do cross validation experiment on Iris, my code crashed(The debugging info is KeyError when I use my decision tree to classify a flower). After I analysis the whole process of this algorithm, I found the question is as following:

When do cross validation, as we split the dataset into pieces, our training data is a subset of original dataset. For example, if we choose first 100 data as our training data. The data in this subset at attribute 1(index 0) has only 3 values: S,ML,MS. But the data in our test dataset at attribute 1 have another value which is L.

In this case, our decision will not work because at attribute 1 has only 3 branches called S,ML,MS. When it wants to classify a data with value L at attribute 1, it does not how to classify this guy. In this case, we need to set a default value about our decision tree. It means when the decision come across a data with a value the tree does not include, directly classify it to the default type of our tree.

However, this may cause another problem: for example, in the last 50 data items in iris dataset , about 10 items has value L at atrribute 1. In this case, our test result(error rate) will go up to around 20%, that is not we want. As result, the solution I come up with is to shuffle the dataset before traning, which could minimize the keyerror in testing because in this case the each type of data is distributed evenly in while dataset. The decision tree are more likely to include all values of each attribute that dataset have.

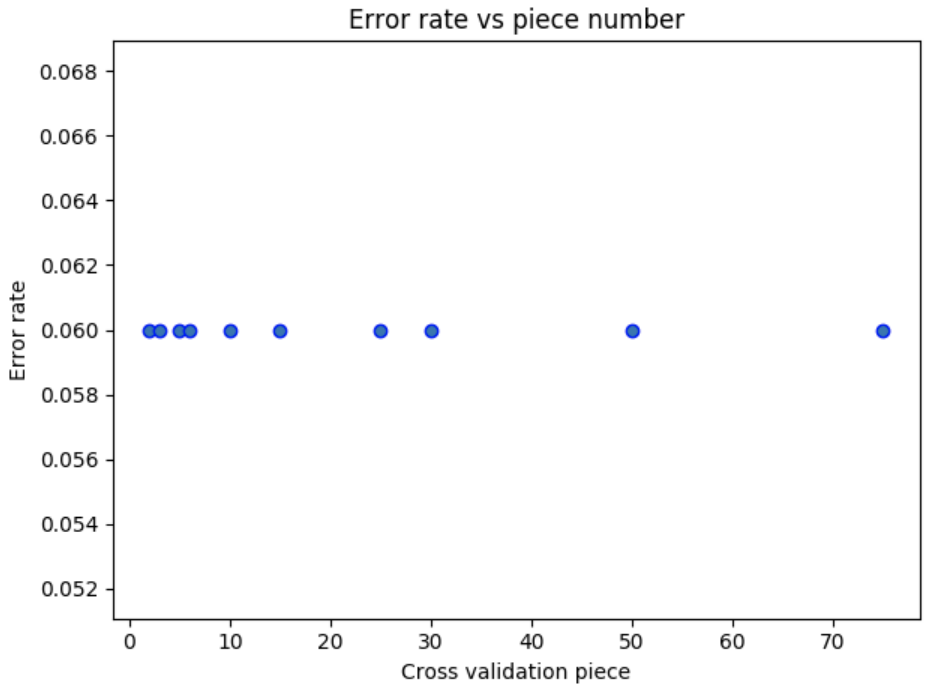
Finally, this situation also reflects a general problem for decision tree classifier. Think about it, even though we build our tree on whole iris dataset, the tree could still not classify a data such as (X,XXL,X,XL). In this situation, it is not proper to classify the data to our default type because the type of data passed in is a totally new type, which is not included in our original dataset. In other words, as long as the data passed in is has a value that our decision tree does not include, it will be difficult to classify this new type data.

After set the default value, the average error rate about Iris dataset is as follows:

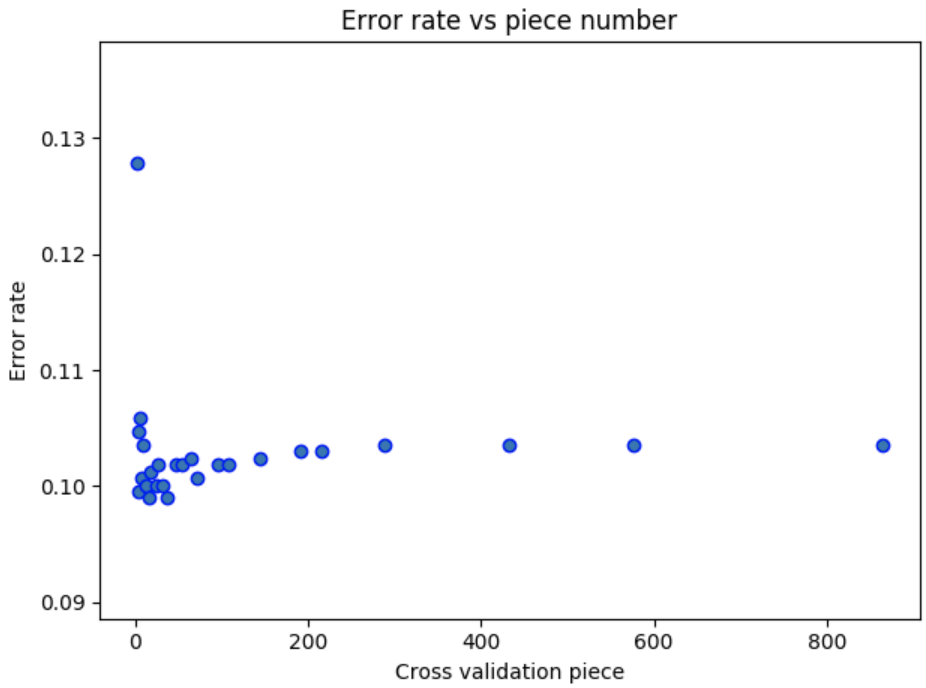
Cross validation code: example is the dataset, num is the piece number you want to assign

```
dt = DecisionTree(example)
dt.crossValidation(num)
```

cross validation error result on iris dataset with different piece number



Cross validation experiment on Cars data set with different piece numbers



Initially it has a relatively big error rate. Then with the piece number increases, the error rate becomes stable around 10%;

After analysis the result of two experiments, I got some conclusions.

Firstly, based on the mechanism of cross validation experiment, the error rate is at a stable value(5% for iris, 10% for cars).

Secondly, shuffling the data & set the default value. It is critical to shuffle the data before training because it could make our decision tree hold more situations and minimize the effect by setting a default value because it makes each type of data distributed evenly.

I have already packaged a function called **docrossvalidation(example,num)**

at the bottom of Decisiontree.py, example is the dataset, num is the number of piece you want to assign when doing crossing validation. (Please make sure the num you pass in meet the requirement: len(size)%num==0)

I have also parsed three dataset called iris, restaurant and cars in DecisionTree.py, you can directly use them.

Part 2: Neural Network

In this part, I establish a Two layers neural network, including one input layer and one output layer, the activation is sigmoid. There is also a bias node in input player.

Preprocess the data:

Here I focus on the Iris data(filename: iris.data.txt) as it has continuous numbers, which suitable for neural network calculation.

As the iris flower dataset has three types totally, and 4 values at each attribute, the network has 5 input node(4 for input data, one for bias) and 3 output nodes.

The output nodes here stands for the probability each for each type that input data is. For example, if the out put value for of three output layer nodes are [0.1,0.1,0.2], then we pick the max one, so the predict result here is type 2.

In this case, we need to transform the label of each data to a vector as following:

type is setosa: vector is [1,0,0]

type is versicolor: vector is [0,1,0]

type is virginica: vector is [0,0,1]

I also make a normalized iris dataset here, use for test later.

Normalization formula: value = value-minvalue/maxvalue-minvalue

Here the minvalue and maxvalue is the maxminum and minimum at each attribute through the whole dataset.

Mathematical Derivation:

In my opinion, the core part of establishing a neural network and do back propagation is to figure out the math theory behind the whole algorithm, so here I want introduce my understand for the BP Neural network algorithm.

Firstly, as my neural network has 5 input nodes, 3 output nodes, it has total (4+1)*3 = 15 weight values.

As to back propagate and update weight value to get minimum loss, we have following mathematical derivation:

Define t as target value, y as output of the output node.

then we define:

loss = (t-y)^2

now we want to find the

deltaweight = dloss/dweight(as required in gradient descent, update weight in opposite direction of gradient)

In this case, take w11(the weight between first node of input layer and first node of output layer) for example.

It is obvious that the w11 has relationship only to L1(the loss of output on first node of output layer): then the above formula becomes:

deltaW11 = dloss1 / dw11

Here we need to use chain rule in derivation, define sum1 as the input of first node of output layer

*deltaW11 = dloos1/dsum1 * dsum1/dw11*

As y1 equals to sigmoid(sum1), the above equation above becomes:

*deltaW11 = dloss1/dy1 * dy1/dsum1 * dsum1/dw11*

It is obvious that dsum1/dw11 = x1 because the sum1 is (w11*x1+w21*x2+w31*x3+w41*x4+b1), and the derivation of a dsigmoid(x)/dx is sigmoid(x)(1-sigmoid(x))

So the result of deltaWeight11 is as following:

*-2(target-y1)*y1*(1-y1)*x1*

Get rid of the constant and minus sign, final result is:

*(target-y1)*y1*(1-y1)*x1*studyrate*

For bias weight(only difference is that it does not depends on input data)

*(target-y1)*y1*(1-y1)*studyrate*

Shown in code: sigmoidd is derivation of sigmoid function

```
def updateweight(self, targetvalue, data):
    actual = self.output(data)
    deltaWeight = (targetvalue - actual) * sigmoidd(actual)
    for i in range(self.size):
        self.weights[0, i] += studyrate * deltaWeight * data[i]
        self.bias += deltaWeight * studyrate
```

Structure of Code:

As this is such an easy neural network, so I only define a class call OutputNode, each node has a weight value vector, and could get output by input an iris flower data.

Here I use stochastic gradient descent algorithm, this is to update each weight value based on each data and its target value, repeat this process in specific times.

The process is shown in Code:

```
def train(times, dataset):
    Nodes = [OutputNode(4), OutputNode(4), OutputNode(4)]
    for _ in range(times):
        shuffle(dataset)
        for d in dataset:
            data, target = d[:-1], d[-1]
            for i in range(len(Nodes)):
                Nodes[i].updateweight(target[i], data)
```

Times is a parameter passed in the train function to decide the number of training times.

Here is a key point, you need to shuffle the whole dataset before each training to make sure each type of data distributed evenly.

The test code is as follows:

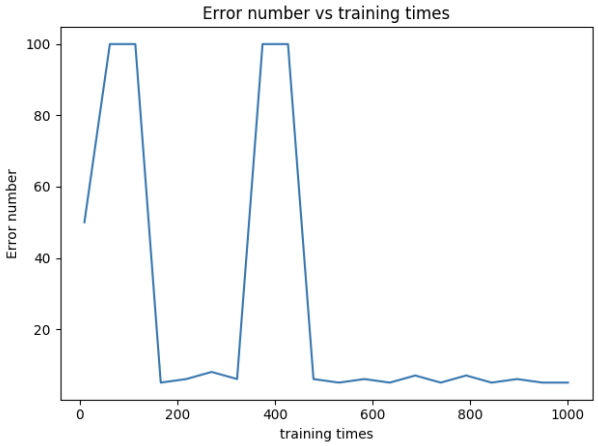
```
    shuffle(dataset)
    count = 0
    for d in dataset:
        data, target = d[:-1], d[-1]
        output = [node.output(data) for node in Nodes]

        pre = np.argmax(output) + 1
        actual = np.argmax(target) + 1
        if pre != actual: count += 1
        # print("Predicted type is: ",pre)
        # print("Actual type is: ",actual)
    print('Error number is: ', count)
    [node.setinitial() for node in Nodes]
    return count
```

Directly compare the output of network and lable, record the wrong number.

Experiment:

The first graph is the error number respect to the training times on original iris dataset

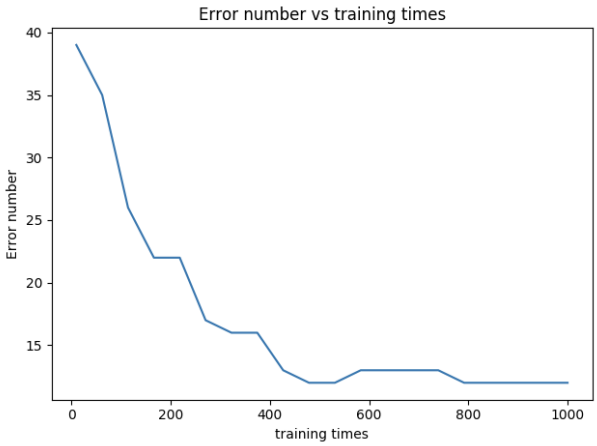
(from 10 to 1000 training times)



Here the step is 10, means the training times is 10,20,30... until 1000.

It obvious at the initial stage, there may be some peaks(error number is 100) . With the increase of training times, the error number becomes stables with a much lower absolute value.

Second Graph is error number respect to the training times on normalized iris dataset,



It is obvious that after implementing normalization on dataset, the error number has a more stable performance, which appears less peak error number at initial stage. The error number decrease stably with increase of training number.

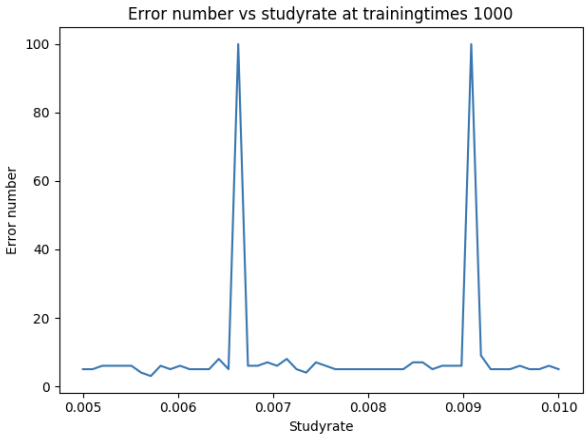
Neural Network Graph: Error rate vs study rate

This Figure is to built based on continuous Iris dataset(original dataset)

The studyrate is in range(0.005,0.10), test 50 points in this range

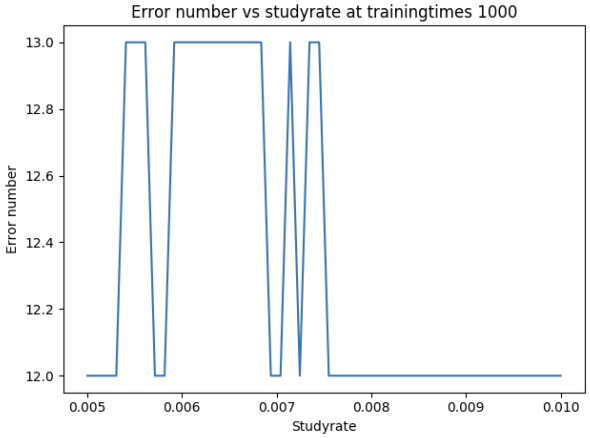
Set default training times is 1000

Figure:



This figure is built based on normalized Iris dataset, (implement normalization of original dataset), has same study rate and default training times as above figure:

Figure:



It can be observed that the normalized dataset has a more stable performance(error number range in 12 and 13), but in original dataset there exists some high error number peaks.

Part 3: Code Document:

This is a document of code in details, if you just want to know how to use the function and class, could jump to readme file

Data Parser.py: Used for parse data from txt files, you do not need to now how it works. I have already packaged the data variable used for both decision tree and neural network in my code.

DecisonTree.py:

Three party library used:

Numpy

Matplotlib

You could use get a decision tree by implement code as following:

dt = DecisionTree(dataset) # establish a decision tree learning instance by dataset

tree = dt.learning() # get the decision tree based on dataset you passed in

tree.testData(dataset): use the tree to do test on dataset and return the error rate(here the dataset must be the original dataset or it's subdataset)

dt.crossvalidation(k)# do cross validation experimental, split dataset into k pieces(tip: k must meet the requirement: len(dataset)/k==0)

you could use the docrossvalidation(dataset,num) directly do the cross validation on dataset you passed in

NeuralNetwork.py: designed for continuousiris&mormalizediris dataset

default studyrate here is 0.01

train(times,dataset,studyrate):

train a neural net work based on dataset passed in

get the network after train k times given a specific study rate

and then do test on original dataset, return the error rate

Experimental.py: Do test on decision tree and neural network

dataset for decision tree: iris, cars

dataset for neural network: continuousIris, normalizedIris

these dataset variables have already been build in this file, you could directly use them

1: plotNeuralNetworkwithTrainingtimes(dataset, mintrainingtimes, maxtrainingtimes):

plot the figure about(error rate with different training times given dataset)

Neural network experiment

Dataset used: continuousIris, normalizedIris

2: plotNeuralNetworkwithStudyrate(dataset, minstudyrate, maxstudyrate, trainingtimes):

plot the figure about(error number vs study rate given training times and dataset)

study rate is in range(minstudyrate,maxstudyrate)

Neural network experiment

Dataset used: continuousIris, normalizedIris

3: plotDecisionTreewithPieceNum(dataset):

plot the figure about(error number vs different cross validation piece number)

Decision Tree experiment

Dataset used: iris, cars