北京航空航天大學
BEIHANG UNIVERSITY

Pattern Recognition and Machine Learning Experiment Report

院（系）名称 自动化科学与电气工程学院

专业名称 模式识别与智能系统

学生学号 14031259

学生姓名 孔昭宁

2017 年 5 月 15 日

3 Decision Tree Learning for Classification

3.1 Introduction

Decision tree induction is one of the simplest and yet most successful learning algorithms. A decision tree (DT) consists of internal and external nodes and the interconnections between nodes are called branches of the tree. An internal node is a decision-making unit to decide which child nodes to visit next depending on different possible values of associated variables. In contrast, an external node also known as a leaf node, is the terminated node of a branch. It has no child nodes and is associated with a class label that describes the given data. A decision tree is a set of rules in a tree structure, each branch of which can be interpreted as a decision rule associated with nodes visited along this branch.

3.2 Principle and Theory

Decision trees classify instances by sorting them down the tree from root to leaf nodes. This tree-structured classifier partitions the input space of the data set recursively into mutually exclusive spaces. Following this structure, each training data is identified as belonging to a certain subspace, which is assigned a label, a value, or an action to characterize its data points. The decision tree mechanism has good transparency in that we can follow a tree structure easily in order to explain how a decision is made. Thus interpretability is enhanced when we clarify the conditional rules characterizing the tree.

Entropy of a random variable is the average amount of information generated by observing its value. Consider the random experiment of tossing a coin with probability of heads equal to 0.9, so that $p(head) = 0.9$ and $p(tail) = 0.1$. This provides more information than the case where $p(head) = 0.5$ and $p(tail) = 0.5$.

Entropy is used to evaluate randomness in physics, where a large entropy value indicates that the process is very random. The decision tree is guided heuristically according to the information content of each attribute. Entropy is used to evaluate the information of each attribute, as a means of classification. Suppose we have m classes, for a particular attribute, we denoted it by p_i by the proportion of data which belongs to class C_i where $i = 1,2, \dots, m$.

The entropy of this attribute is then:

$$Entropy = \sum_{i=1}^{m} -p_i \cdot log_2 p_i$$

We can also say that entropy is a measurement of the impurity in a collection of training examples: larger the entropy, the more impure the data is. Based on entropy, Information Gain (IG) is used to measure the effectiveness of an attribute as a means of discriminating between classes.

$$IG(S,A) = Entropy(S) - \sum_{v \in Values(A)} \frac{|S_V|}{|S|} Entropy(S_v)$$

where all examples S is divided into several groups (i.e. S_v for $v \in Values(A)$) according to the value of A. It is simply the expected reduction of entropy caused by partitioning the examples according to this attribute.

3.3 Objective

The goals of the experiment are as follows:

(1) To understand why we use entropy-based measure for constructing a decision tree.

(2) To understand how Information Gain is used to select attributes in the process of building a decision tree.

(3) To understand the equivalence of a decision tree to a set of rules.

(4) To understand why we need to prune the tree sometimes and how can we prune? Based on what measure we prune a decision tree.

(5) To understand the concept of Soft Decision Trees and why they are important extensions to classical decision trees.

3.4 Contents and Procedures

I conducted this experiment in Python 3.5.1, and visualize the decision tree with the graph drawing software Graphviz 2.3.6, with the help of the python interface package Graphviz 0.7.

Stage 1

(1) According to the above principles and theories in section 3.2, implement the code to calculate the information entropy of each attribute.

The information entropy is derived from the following formula

$$Entropy = \sum_{i=1}^{m} -p_i \cdot log_2 p_i$$

where m is the number of classes, while p_i denotes the likelihood that a sample falls in the class C_i.

(2) Select the most informative attribute from a given classification problem.

The information gain is derived from the following formula

$$IG(S, A) = Entropy(S) - \sum_{v \in Values(A)} \frac{|S_V|}{|S|} Entropy(S_v)$$

where $Entropy(S_v)$ denotes the information entropy of samples belonging to class v, while $\frac{|S_V|}{|S|}$ serves as a regularization term. The information gain measures the effectiveness of an attribute with which we classify the samples into different classes.

To select the most informative attribute, I calculated information gain for all attributes given the dataset. The attribute which provides the maximum information gain is then selected to classify the data into two separate classes.

(3) Find an appropriate data structure to represent a decision tree. Building the tree from the root to leaves based on the principles discussed in section 3.2 by using Information Gain guided heuristics.

The decision tree I implemented is implemented by the following data structures. The branch node (internal node) is represented by a 'BranchNode' class, each containing the attribute and value used to separate incoming data into two classes. In addition, each instance of 'BranchNode' contains references to both its left/right children.

```
class BranchNode():
        def __init__(self, attribute, threshold, left, right):
            self.attribute = attribute
            self.threshold = threshold
            self.left = left
            self.right = right
```

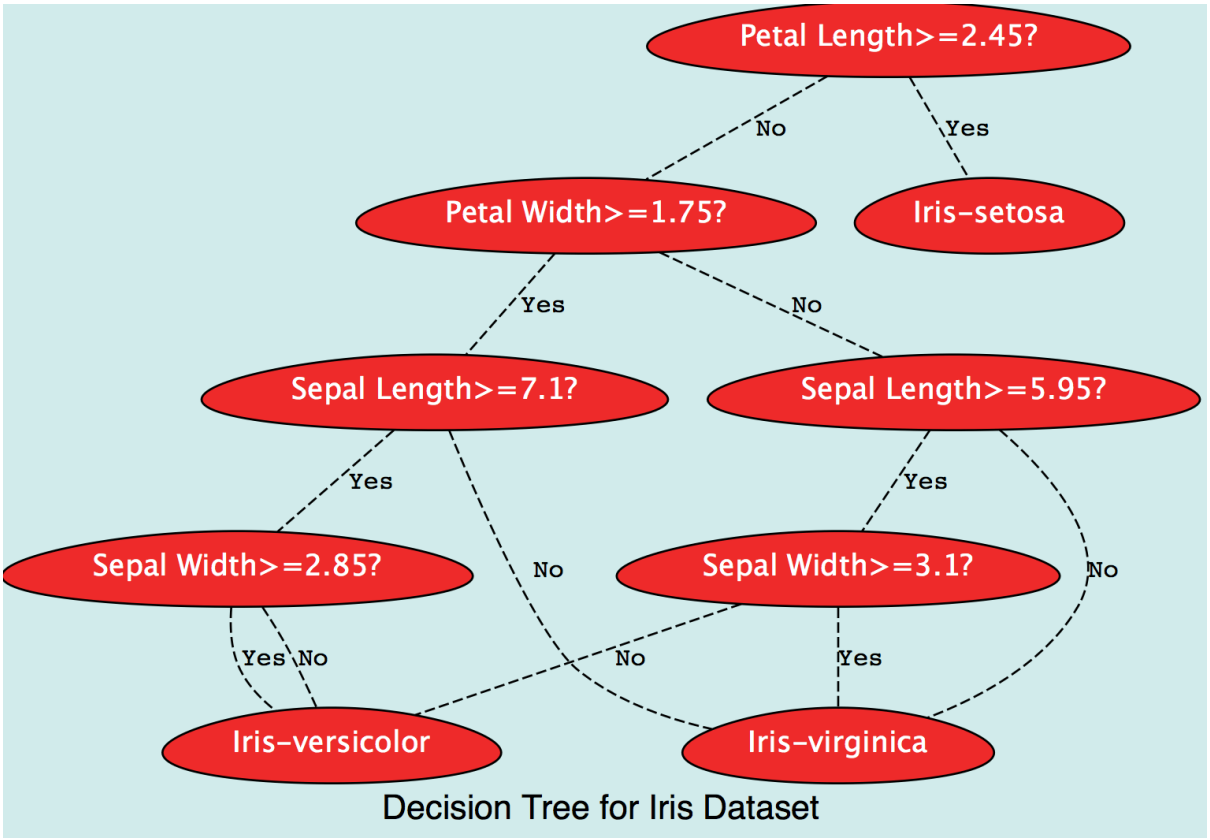
The leaf nodes are represented as integers, which are the indices representing each class.

The decision tree is then encapsulated into the class 'DecisionTree', which consists of a method 'createTree()' which recursively calls itself to construct the decision tree from root to leaf. This function is automatically called by the constructor of the class, when a dataset is used to initialize an 'DecisionTree' instance. In addition, the 'DecisionTree' class provided an API named 'visualize()', which visualizes the decision tree in svg form and saves it to disk.

After the encapsulations described above, the decision tree can be easily created with the following code:

```
dataset = DecisionTree.load_data()
tree = DecisionTree(dataset)
tree.visualize()
```

The code above trains a decision tree, and visualizes it as follows. The phenomenon that both the two cases whether sepal width ≥ 2.9 are classified as Iris-versicolor is due to pruning: The decision fails to correctly classify 100% samples, after all attributes have been used once. However, Iris-versicolor accounts for the majority of the samples arriving at this node, thus they are classified as Iris-versicolor.



Stage 2

(1) Now consider the case of with continuous attributes or mixed attributes (with both continuous and discrete attributes), how can we deal with the decision trees? Can you propose some approaches to discretization?

Given dataset D and a continuous attribute a, my approach to discretize continuous attributes is as follows:

1) Sort the dataset in ascending order of this attribute, which produces $\{a^1, a^2, \dots, a^n\}$. With a partitioning threshold t, the dataset can be separated into D_t^- and D_t^+, where D_t^- contains samples whose $a < t$, whereas D_t^+ contains samples whose $a \geq t$.

2) Obviously, for adjacent attribute values a^i and a^{i+1}, any value of t between $(a^i, a^{i+1}]$ results in identical partitions. Therefore, for attribute a, there exists $n-1$ values with which we can partition the dataset:

   $T_a = \left\{\frac{a^i+a^{i+1}}{2} | 1 \leq i \leq n-1\right\}$.

3) For each of these values, I calculated their information gain (IG), and selected the value where the information gain is the greatest:

$$Gain(D,a) = \max_{t \in T_a} Gain(D,a,t) = \max_{t \in T_a} Ent(D) - \sum_{\lambda \in \{-,+\}} \frac{|D_t^\lambda|}{|D|} Ent(|D_t^\lambda|)$$

(2) Is there a tradeoff between the size of the tree and model accuracy? Is there an optimal tree in both compactness and performance?

There has been a tradeoff between tree size and model accuracy. With a tree too large, it tends to learn noises of the training set, which causes overfit. In such circumstances, the performance on the test set may not be desirable, although the accuracy on the training set can be extremely high.

On the contrary, a small tree is not able to capture the important structural information within the dataset, which performs poorly on both training and test datasets.

Pruning is a typical way to attain balance between compactness and performance. Pruning can occur in a top-down or bottom-up fashion, based on several different criteria, such as reduced error and cost complexity.

(3) For one data element, the classical decision tree gives a hard boundary to decide which branch to follow, can you propose a "soft approach" to increase the robustness of the decision tree?

In canonical hard decision trees, one of the children is chosen at each decision node. Therefore, a single path from the root to one of the leaves is traversed.

In soft decision trees, however, all children are selected but all with a certain probability. That is, we follow all the paths to all the leaves and all the leaves contribute to the final decision but with different probabilities.

Soft decision trees have several advantages. First, they provide a soft response whereas hard trees have discontinuous response at leaf boundaries. This enables soft decision trees to have smoother fits, and hence lower bias around the split boundaries. Second, a linear gating

function enables soft trees to make oblique splits in contrast to the axis-orthogonal splits made by hard trees.

(4) Compare to the Naïve Bayes, what are the advantages and disadvantages of the decision tree learning?

Advantages:

1) Very flexible, intuitive, easy to interpret, easy to debug.
2) Requires small amount of data to train model
3) Good results obtained in most of the cases
4) Insensitive to missing values

Disadvantages:

1) Discretization reduces accuracy
2) Easy to overfit, susceptible to external disturbances
3) Difficult for regression problems
4) Unable to handle correlated attributes

3.5 Experience

Conducting this experiment have strengthened my understanding of decision trees. Having coded a complete decision tree in detail, I had a better grasp about how to select the most discriminative attributes, how to select the appropriate data structure for the decision tree, how to discretize a continuous dataset, etc.

In addition, by comparing decision trees with other classifiers, I have learned the pros and cons of decision trees, and some improvements such as pruning and soft-boundaries.

3.6 References

[1] Soft Decision Trees. *O. Irsoy, O. T. Yildiz, E. Alpaydin* ICPR 21, Tsukuba, Japan.

3.7 Codes

```
# -*- coding:utf-8 -*-
import os
import numpy as np
import graphviz as gv

# Construct global name references
name2type = {'Iris-setosa' : 0, 'Iris-versicolor' : 1, 'Iris-virginica': 2}
type2name = {0 : 'Iris-setosa', 1 : 'Iris-versicolor', 2 : 'Iris-virginica'}
attribute2name = {0 : 'Sepal Length', 1 : 'Sepal Width', 2 : 'Petal Length', 3 : 'Petal
```

```
Width'}

class DecisionTree():
    # Branch nodes are classes and leaf nodes are integers
    class BranchNode():
        def __init__(self, attribute, threshold, left, right):
            self.attribute = attribute
            self.threshold = threshold
            self.left = left
            self.right = right

    # Construct decision tree with dataset including n-1 attributes and 1 flag
    # Build decision tree recursively
    def __init__(self, dataset):
        attributes = set([i for i in range(len(dataset[0]) - 1)])
        self.root = self.createTree(dataset, attributes)

    # Load data into a matrix with flag in numbers
    @staticmethod
    def load_data():
        fileName = 'bezdekIris.txt'
        filePath = os.path.join('data', fileName)
        data = []
        with open(filePath, 'r') as f:
            for line in f.readlines():
                line = line.split(sep = ',')
                tmp = [float(i) for i in line[:-1]]
                tmp.append(name2type[line[-1][:-1]])
                data.append(tmp)
        return data

    # Calculate entropy of certain dataset
    @staticmethod
    def calculateEntropy(dataset):
        flag2num = dict()
        total = len(dataset)
        entropy = 0
        for data in dataset:
            flag = data[-1]
            if flag in flag2num.keys():
                flag2num[flag] += 1
            else:
                flag2num[flag] = 1
        for flag in flag2num:
            p = flag2num[flag] / total
            entropy -= p * np.log2(p)
        return entropy

    # Calculate entropy gain W.R.T. a certain attribute and threshold
    @staticmethod
    def calculateEntropyGain(dataset, attribute, threshold):
        # Sort data according to attribute
        dataset.sort(key = lambda x : x[attribute])
        entropy = DecisionTree.calculateEntropy(dataset)
        # Find the threshold position
        thres_pos = 0
        while (dataset[thres_pos][attribute] < threshold):
            thres_pos += 1
        entropy -= DecisionTree.calculateEntropy(dataset[:thres_pos]) * thres_pos /
len(dataset)
        entropy -= DecisionTree.calculateEntropy(dataset[thres_pos:]) * (len(dataset) -
thres_pos) / len(dataset)
        return entropy

    # Calculate optimal threshold for a particular attribute
```

```
    @staticmethod
    def calculateThreshold(dataset, attribute):
        # Sort data according to attribute
        dataset.sort(key = lambda x : x[attribute])
        threshold_list = [data[attribute] for data in dataset]
        entropy_gain_max = 0
        for threshold in threshold_list:
            entropy_gain_current = DecisionTree.calculateEntropyGain(dataset, attribute,
threshold)
            if entropy_gain_current > entropy_gain_max:
                entropy_gain_max = entropy_gain_current
                final_threshold = threshold
        return final_threshold

    # Select attribute with maximum entropy gain and its threshold
    @staticmethod
    def selectAttribute(dataset, attributes):
        maxEntropyGain = 0
        threshold_selected = 0
        for attribute in attributes:
            threshold = DecisionTree.calculateThreshold(dataset, attribute)
            entropyGain = DecisionTree.calculateEntropyGain(dataset, attribute, threshold)
            if entropyGain > maxEntropyGain:
                maxEntropyGain = entropyGain
                attribute_selected = attribute
                threshold_selected = threshold
        return (attribute_selected, threshold_selected)

    # Check majority flag in dataset and return
    @staticmethod
    def majorityFlag(dataset):
        flag2num = dict()
        for data in dataset:
            if data[-1] in flag2num.keys():
                flag2num[data[-1]] += 1
            else:
                flag2num[data[-1]] = 1
        maxCnt = 0
        for flag in flag2num.keys():
            if flag2num[flag] > maxCnt:
                maxCnt = flag2num[flag]
                maxFlag = flag
        return maxFlag

    # Create decision tree recursively
    @staticmethod
    def createTree(dataset, attributes):
        flagList = [data[-1] for data in dataset]
        # If all samples belong to same flag, return this flag
        if flagList.count(flagList[0]) == len(flagList):
            return flagList[0]
        # No more attributes, return majority flag
        if len(attributes) == 0:
            return DecisionTree.majorityFlag(dataset)
        # Selected attribute with maximum entropy gain
        (attributeSelected, threshold) = DecisionTree.selectAttribute(dataset, attributes)
        # Generate two branches with new dataset and attributes
        dataset_l = [data for data in dataset if data[attributeSelected] < threshold]
        dataset_h = [data for data in dataset if data[attributeSelected] >= threshold]
        # Remove current attribute
        attributes.remove(attributeSelected)
        if len(dataset_l) > 0:
            # Create sub-tree recursively
            left = DecisionTree.createTree(dataset_l, attributes)
        else:
```

```
            # Assign child node with majority flag of parent node
            left = DecisionTree.majorityFlag(dataset)
        if len(dataset_h) > 0:
            # Create sub-tree recursively
            right = DecisionTree.createTree(dataset_h, attributes)
        else:
            # Assign child node with majority flag of parent node
            right = DecisionTree.majorityFlag(dataset)
        node = DecisionTree.BranchNode(attributeSelected, threshold, left, right)
        attributes.add(attributeSelected)
        return node

    # Visualize the decision tree
    def visualize(self):
        # Depth first search decision tree, add node/edge to set
        def dfs(root, graph, nodeSet, edgeSetL, edgeSetR):
            assert isinstance(root, DecisionTree.BranchNode)
            title = attribute2name[root.attribute] + '>=' + str(root.threshold) + '?'
            graph.node(title)
            # Plot the left tree
            if isinstance(root.left, DecisionTree.BranchNode):
                dfs(root.left, graph, nodeSet, edgeSetL, edgeSetR)
                title_left = attribute2name[root.left.attribute] + '>=' +
str(root.left.threshold) + '?'
                edgeSetL.add((title, title_left))
            else:
                nodeSet.add(type2name[root.left])
                edgeSetL.add((title, type2name[root.left]))
            # Plot the right tree
            if isinstance(root.right, DecisionTree.BranchNode):
                dfs(root.right, graph, nodeSet, edgeSetL, edgeSetR)
                title_right = attribute2name[root.right.attribute] + '>=' +
str(root.right.threshold) + '?'
                edgeSetR.add((title, title_right))
            else:
                nodeSet.add(type2name[root.right])
                edgeSetR.add((title, type2name[root.right]))
        # Update styles to graph
        def apply_styles(graph, styles):
            graph.graph_attr.update(('graph' in styles and styles['graph']) or {})
            graph.node_attr.update(('nodes' in styles and styles['nodes']) or {})
            graph.edge_attr.update(('edges' in styles and styles['edges']) or {})
            return graph

        self.graph = gv.Graph(format = 'svg')
        nodeSet = set()
        edgeSetL = set()
        edgeSetR = set()
        dfs(self.root, self.graph, nodeSet, edgeSetL, edgeSetR)
        # Plot all elements in set
        for node in nodeSet:
            self.graph.node(node)
        for node in edgeSetL:
            self.graph.edge(*node, **{'label' : 'Yes'})
        for node in edgeSetR:
            self.graph.edge(*node, **{'label' : 'No'})
        # Choose graph styles
        styles = {
            'graph': {
                'label': 'Decision Tree for Iris Dataset',
                'fontname' : 'Helvetica',
                'fontsize': '16',
                'fontcolor': 'black',
                'bgcolor': '#D1EEEE',
            },
```

```
            'nodes': {
                'fontname': 'Lucida Grande',
                'shape': 'egg',
                'fontcolor': 'white',
                'color': 'black',
                'style': 'filled',
                'fillcolor': '#EE2C2C',
            },
            'edges': {
                'style': 'dashed',
                'color': 'black',
                'arrowhead': 'open',
                'fontname': 'Courier',
                'fontsize': '12',
                'fontcolor': 'black',
            }
        }
        self.graph = apply_styles(self.graph, styles)
        filename = self.graph.render(filename = os.path.join('img', 'graph'))

if __name__ == '__main__':
    dataset = DecisionTree.load_data()
    tree = DecisionTree(dataset)
    tree.visualize()
```