Huawei AI Certification Training

HCIA-AI

Machine Learning Experiment Guide

ISSUE:3.0

HUAWEI TECHNOLOGIES CO., LTD.

Huawei Technologies Co., Ltd.

Address: Huawei Industrial Base Bantian, Longgang Shenzhen 518129 People's Republic of China

Website: http://e.huawei.com

Huawei Certificate System

Huawei Certification follows the "platform + ecosystem" development strategy, which is a new collaborative architecture of ICT infrastructure based on "Cloud-Pipe-Terminal". Huawei has set up a complete certification system consisting of three categories: ICT infrastructure certification, Platform and Service certification and ICT vertical certification, and grants Huawei certification the only all-range technical certification in the industry.

Huawei offers three levels of certification: Huawei Certified ICT Associate (HCIA), Huawei Certified ICT Professional (HCIP), and Huawei Certified ICT Expert (HCIE).

HCIA-AI V3.0 aims to train and certify engineers who are capable of designing and developing AI products and solutions using algorithms such as machine learning and deep learning.

HCIA-AI V3.0 certification demonstrates that: You know the development history of AI, Huawei Ascend AI system and full-stack all-scenario AI strategies, and master traditional machine learning and deep learning algorithms; you can use the TensorFlow and MindSpore development frameworks to build, train, and deploy neural networks; you are competent for sales, marketing, product manager, project management, and technical support positions in the AI field.


Huawei Certification Portfolio
HUAWEI
Huawei Certification
ICT Vertical Certification
Finance
Public Safety
Platform and Service Certification
Big Data
AI
IoT
Intelligent Vision
Enterprise Communication
GaussDB
Kunpeng Application Developer
Cloud Computing
Cloud Service
ICT Infrastructure Certification
Data Center
Storage
Intelligent Computing
Security
Datacom
WLAN
SDN
Transmission
Access
LTE
5G
HCIE
Huawei Certified ICT Expert
HCIP
Huawei Certified ICT Professional
HCIA
Huawei Certified ICT Associate

About This Document

Overview

This document is applicable to the candidates who are preparing for the HCIA-AI exam and the readers who want to understand the AI programming basics. After learning this guide, you will be able to perform basic machine learning programming.

Description

This guide contains one experiment, which is based on how to use sklearn-learn and python packages to predict house prices in Boston using different regression algorithms. It is hoped that trainees or readers can get started with machine learning and have the basic programming capability of machine learning building.

Background Knowledge Required

To fully understand this course, the readers should have basic Python programming capabilities, knowledge of data structures and mechine learning algorithms.

Experiment Environment Overview

Python Development Tool

This experiment environment is developed and compiled based on Python 3.6 and XGBoost will be used.

Contents

About This Document 3
Overview 3
Description 3
Background Knowledge Required 3
Experiment Environment Overview 3
1 Boston House Price Forecast 6
1.1 Introduction 6
1.1.1 About This Experiment 6
1.1.2 Objectives 6
1.1.3 Datasets and Frameworks Used for the Experiment 6
1.2 Experiment Code 7
1.2.1 Introducing Dependencies 7
1.2.2 Loading the Data Set, Viewing Data Attributes, and Visualizing the Data 8
1.2.3 Spliting and Pre-processing the Data Set 9
1.2.4 Using Various Regression Models to Model Data Sets 10
1.2.5 Adjusting Hyperparameters by Grid Search 10
1.3 Summary 12
2 Detail of linear regression 13
2.1 Introduction 13
2.1.1 About This Experiment 13
2.1.2 Objectives 13
2.2 Experiment Code 13
2.2.1 Data preparation 13
2.2.2 Define related functions 14
2.2.3 Start the iteration 15
2.3 Thinking and practice 20
2.3.1 Question 1 20
2.3.2 Question 2 20
3 Decision tree details 21
3.1 Introduction 21
3.1.1 About This Experiment 21
3.1.2 Objectives 21
3.2 Experiment Code 21
3.2.1 Import the modules you need 21

3.2.2 Superparameter definition section 21
3.2.3 Define the functions required to complete the algorithm 22
3.2.4 Execute the code 27

1 Boston House Price Forecast

1.1 Introduction

1.1.1 About This Experiment

The development in this experiment is based on ModelArts. For details about how to set up the environment, see the HCIA-AI V3.0 Experiment Environment Setup Guide. The sample size of the dataset used in this case is small, and the data comes from the open source Boston house price data provided by scikit-learn. The Boston House Price Forecast project is a simple regression model, through which you can learn some basic usage of the machine learning library sklearn and some basic data processing methods.

1.1.2 Objectives

- Upon completion of this task, you will be able to:
  Use the Boston house price data set that is open to the Internet as the model input data.
- Build, train, and evaluate machine learning models.
- Understand the overall process of building a machine learning model.
- Master the application of machine learning model training, grid search, and evaluation indicators.
- Master the application of related APIs.

1.1.3 Datasets and Frameworks Used for the Experiment

This case is based on the Boston dataset, which contains 13 features and 506 data records. Each data record contains detailed information about the house and its surroundings. Specifically, it includes urban crime rate, nitric oxide concentration, average rooms in a house, weighted distance to the downtown area and average house price. The details are as follows:

- CRIM: urban per capita crime rate
- ZN: proportion of residential land exceeds 25,000 square feet
- INDUS: proportion of non-retail commercial land in a town
- CHAS: Charles river empty variable (1 indicates that the boundary is a river; otherwise, the value is 0)
- NOX: Nitric oxide concentration
- RM: average number of rooms in a house

- AGE: proportion of private houses completed before 1940
- DIS: weighted distance to the five central regions of Boston
- RAD: proximity index of a radial highway
- TAX: full value property tax rate of $10,000
- PTRATIO: proportion of teachers and students in urban areas
- target: average price of private houses, unit: $1,000

Framework: Sklearn, which provides Boston house price data, data set segmentation, standardization, and evaluation functions, and integrates various common machine learning algorithms. In addition, XGboost is used, which is an optimized version of GBDT in the integration algorithm.

1.2 Experiment Code

1.2.1 Introducing Dependencies

Code:

```
#Prevent unnecessary warnings.
import warnings
warnings.filterwarnings("ignore")

#Introduce the basic package of data science.
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt
import pandas as pd
import scipy.stats as st
import seaborn as sns
##Set attributes to prevent garbled characters in Chinese.
mpl.rcParams['font.sans-serif'] = [u'SimHei']
mpl.rcParams['axes.unicode_minus'] = False

#Introduce machine learning, preprocessing, model selection, and evaluation indicators.
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import r2_score

#Import the Boston dataset used this time.
from sklearn.datasets import load_boston

#Introduce algorithms.
from sklearn.linear_model import RidgeCV, LassoCV, LinearRegression, ElasticNet
#Compared with SVC, it is the regression form of SVM.
from sklearn.svm import SVR
#Integrate algorithms.
from sklearn.ensemble import RandomForestRegressor, GradientBoostingRegressor
from xgboost import XGBRegressor
```

1.2.2 Loading the Data Set, Viewing Data Attributes, and Visualizing the Data

Step 1 Load the Boston house price data set and display related attributes.

Code:

```
#Load the Boston house price data set.
boston = load_boston()

#x features, and y labels.
x = boston.data
y = boston.target

#Display related attributes.
print('Feature column name')
print(boston.feature_names)
print("Sample data volume: %d, number of features: %d"% x.shape)
print("Target sample data volume: %d"% y.shape[0])
```

Output:

```
Feature column name
['CRIM' 'ZN' 'INDUS' 'CHAS' 'NOX' 'RM' 'AGE' 'DIS' 'RAD' 'TAX' 'PTRATIO'
'B' 'LSTAT']
Sample data volume: 506; number of features: 13
Target sample data volume: 506
```

Step 2 Convert to the dataframe format.

Code:

```
x = pd.DataFrame(boston.data, columns=boston.feature_names)
x.head()
```

Output:

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT
0	0.00632	18.0	2.31	0.0	0.538	6.575	65.2	4.0900	1.0	296.0	15.3	396.90	4.98
1	0.02731	0.0	7.07	0.0	0.469	6.421	78.9	4.9671	2.0	242.0	17.8	396.90	9.14
2	0.02729	0.0	7.07	0.0	0.469	7.185	61.1	4.9671	2.0	242.0	17.8	392.83	4.03
3	0.03237	0.0	2.18	0.0	0.458	6.998	45.8	6.0622	3.0	222.0	18.7	394.63	2.94
4	0.06905	0.0	2.18	0.0	0.458	7.147	54.2	6.0622	3.0	222.0	18.7	396.90	5.33

Figure 1-1 First 5 Data Information

Step 3 Visualize label distribution.

Code:

```
sns.distplot(tuple(y), kde=False, fit=st.norm)
```

Output:

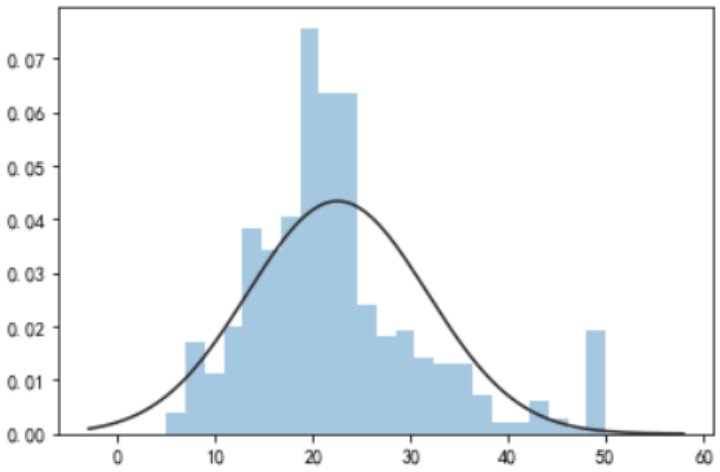

Figure 1-2 Target data distribution

1.2.3 Spliting and Pre-processing the Data Set

Code:

```
#Segment the data.
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2, random_state=28)
#Standardize the data set.
ss = StandardScaler()
x_train = ss.fit_transform(x_train)
x_test = ss.transform(x_test)
x_train[0:100]
```

Output:

```
array([[-0.35451414, -0.49503678, -0.15692398, ..., -0.01188637,
         0.42050162, -0.29153411],
       [-0.38886418, -0.49503678, -0.02431196, ..., 0.35398749,
         0.37314392, -0.97290358],
       [ 0.50315442, -0.49503678, 1.03804143, ..., 0.81132983,
         0.4391143 , 1.18523567],
       ...,
       [-0.34444751, -0.49503678, -0.15692398, ..., -0.01188637,
         0.4391143 , -1.11086682],
       [-0.39513036, 2.80452783, -0.87827504, ..., 0.35398749,
         0.4391143 , -1.28120919],
       [-0.38081287, 0.41234349, -0.74566303, ..., 0.30825326,
         0.19472652, -0.40978832]])
```

1.2.4 Using Various Regression Models to Model Data Sets

Code:

```
#Set the model name.
names = ['LinerRegression',
        'Ridge',
        'Lasso',
        'Random Forrest',
        'GBDT',
        'Support Vector Regression',
        'ElasticNet',
        'XgBoost']


#Define the model.
# cv is the cross-validation idea here.
models = [LinearRegression(),
          RidgeCV(alphas=(0.001,0.1,1),cv=3),
          LassoCV(alphas=(0.001,0.1,1),cv=5),
          RandomForestRegressor(n_estimators=10),
          GradientBoostingRegressor(n_estimators=30),
          SVR(),
          ElasticNet(alpha=0.001,max_iter=10000),
          XGBRegressor()]
# Output the R2 scores of all regression models.

#Define the R2 scoring function.
def R2(model,x_train, x_test, y_train, y_test):

          model_fitted = model.fit(x_train,y_train)
          y_pred = model_fitted.predict(x_test)
          score = r2_score(y_test, y_pred)
          return score

#Traverse all models to score.
for name,model in zip(names,models):
          score = R2(model,x_train, x_test, y_train, y_test)
          print("{}: {:.6f}".format(name,score.mean()))
```

Output:

```
LinerRegression: 0.564144
Ridge: 0.563700
Lasso: 0.564078
Random Forrest: 0.646657
GBDT: 0.725883
Support Vector Regression: 0.517310
ElasticNet: 0.564021
XgBoost: 0.765266
```

1.2.5 Adjusting Hyperparameters by Grid Search

Step 1 Build a model.

Code:

```
'''
  'kernel': kernel function
   'C': SVR regularization factor
   'gamma': 'rbf', 'poly' and 'sigmoid' kernel function coefficient, which affects the model performance
'''
parameters = {
   'kernel': ['linear', 'rbf'],
    'C': [0.1, 0.5,0.9,1,5],
    'gamma': [0.001,0.01,0.1,1]
}

#Use grid search and perform cross validation.
model = GridSearchCV(SVR(), param_grid=parameters, cv=3)
model.fit(x_train, y_train)
```

Output:

```
GridSearchCV(cv=3, error_score='raise',
       estimator=SVR(C=1.0, cache_size=200, coef0=0.0, degree=3, epsilon=0.1, gamma='auto',
kernel='rbf', max_iter=-1, shrinking=True, tol=0.001, verbose=False),
       fit_params={}, iid=True, n_jobs=1,
       param_grid={'kernel': ['linear', 'rbf'], 'C': [0.1, 0.5, 0.9, 1, 5], 'gamma': [0.001, 0.01, 0.1, 1]},
       pre_dispatch='2*n_jobs', refit=True, return_train_score=True,
       scoring=None, verbose=0)
```

Step 2 Obtain optimal parameters.

Code:

```
print("Optimal parameter list:", model.best_params_)
print("Optimal model:", model.best_estimator_)
print("Optimal R2 value:", model.best_score_)
```

Output:

```
Optimal parameter list: {'C': 5, 'gamma': 0.1, 'kernel': 'rbf'}
Optimal model: SVR(C=5, cache_size=200, coef0=0.0, degree=3, epsilon=0.1, gamma=0.1,
kernel='rbf', max_iter=-1, shrinking=True, tol=0.001, verbose=False)
Optimal R2 value: 0.797481706635164
```

Step 3 Perform visualization.

Code:

```
##Perform visualization.
ln_x_test = range(len(x_test))
y_predict = model.predict(x_test)

#Set the canvas.
plt.figure(figsize=(16,8), facecolor='w')
#Draw with a red solid line.
plt.plot (ln_x_test, y_test, 'r-', lw=2, label=u'Value')
#Draw with a green solid line.
```

```
plt.plot (ln_x_test, y_predict, 'g-', lw = 3, label=u'Estimated value of the SVR algorithm, $R^2$=%.3f' %
(model.best_score_))

#Display in a diagram.
plt.legend(loc ='upper left')
plt.grid(True)
plt.title(u"Boston Housing Price Forecast (SVM)")
plt.xlim(0, 101)
plt.show()
```

Output:

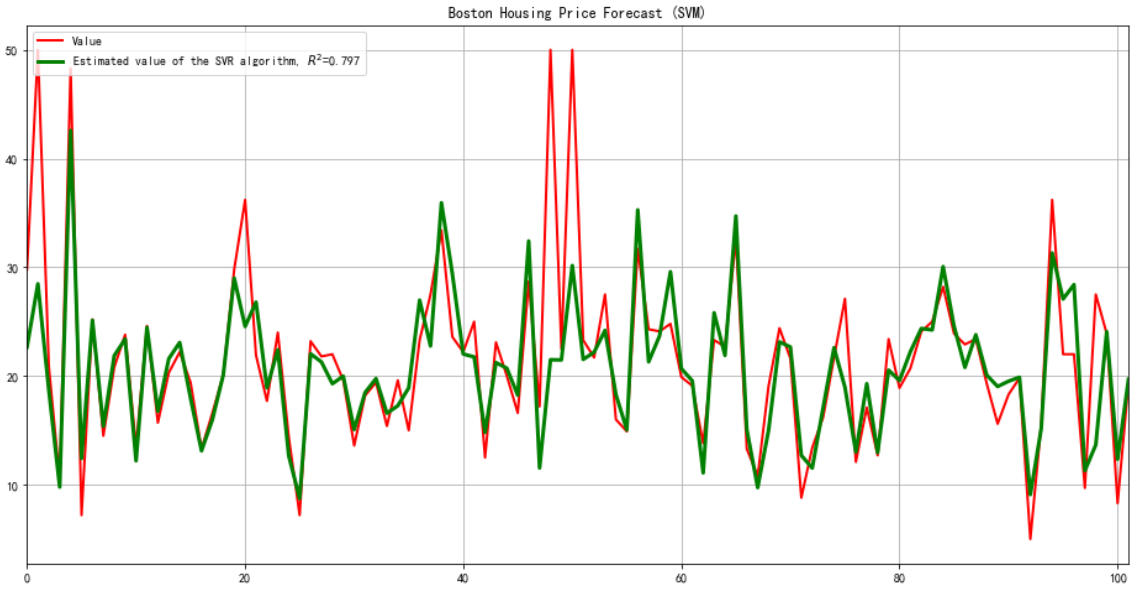

Figure 1-3 Visualization result

1.3 Summary

This chapter describes how to build a Boston house price regression model based on sklearn, including importing, segmenting, and standardizing data, defining models, and setting hyperparameters, and provides trainees with a basic concept of machine learning model building.

2 Detail of linear regression

2.1 Introduction

2.1.1 About This Experiment

This experiment mainly uses basic Python code and the simplest data to reproduce how a linear regression algorithm iterates and fits the existing data distribution step by step.

The experiment mainly used Numpy module and Matplotlib module.Numpy for calculation, Matplotlib for drawing.

2.1.2 Objectives

The main purpose of this experiment is as follows.

- Familiar with basic Python statements
- Master the implementation steps of linear regression

2.2 Experiment Code

2.2.1 Data preparation

10 data were randomly set, and the data were in a linear relationship.

The data is converted to array format so that it can be computed directly when multiplication and addition are used.

Code:

```
#Import the required modules, numpy for calculation, and Matplotlib for drawing
import numpy as np
import matplotlib.pyplot as plt
#This code is for jupyter Notebook only
%matplotlib inline

# define data, and change list to array
x = [3,21,22,34,54,34,55,67,89,99]
x = np.array(x)
y = [1,10,14,34,44,36,22,67,79,90]
y = np.array(y)

#Show the effect of a scatter plot
plt.scatter(x,y)
```

Output:

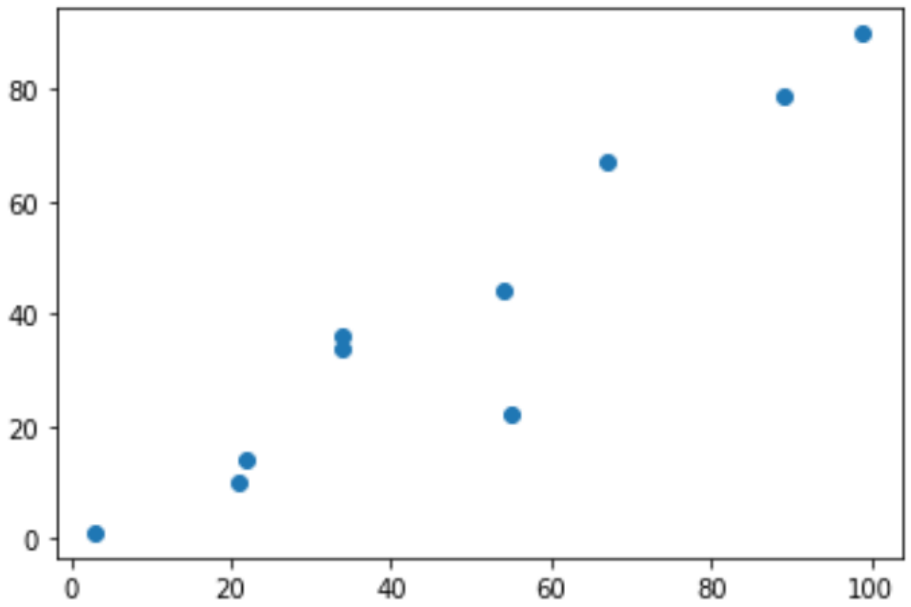

Figure 2-1 Scatter Plot

2.2.2 Define related functions

Model function: Defines a linear regression model wx+b.

Loss function: loss function of Mean square error.

Optimization function: gradient descent method to find partial derivatives of w and b.

Code:

```
#The basic linear regression model is wx+ b, and since this is a two-dimensional space, the model is ax+ b

def model(a, b, x):
    return a*x + b

#The most commonly used loss function of linear regression model is the loss function of mean variance difference
def loss_function(a, b, x, y):
    num = len(x)
    prediction=model(a,b,x)
    return (0.5/num) * (np.square(prediction-y)).sum()

#The optimization function mainly USES partial derivatives to update two parameters a and b
def optimize(a,b,x,y):
    num = len(x)
    prediction = model(a,b,x)
    #Update the values of A and B by finding the partial derivatives of the loss function on a and b
    da = (1.0/num) * ((prediction -y)*x).sum()
    db = (1.0/num) * ((prediction -y).sum())
```

```
    a = a - Lr*da
    b = b - Lr*db
    return a, b

#iterated function, return a and b
def iterate(a,b,x,y,times):
    for i in range(times):
        a,b = optimize(a,b,x,y)
    return a,b
```

2.2.3 Start the iteration

Step 1 Initialization and Iterative optimization model

Code:

```
#Initialize parameters and display
a = np.random.rand(1)
print(a)
b = np.random.rand(1)
print(b)
Lr = 1e-4

#For the first iteration, the parameter values, losses, and visualization after the iteration are displayed
a,b = iterate(a,b,x,y,1)
prediction=model(a,b,x)
loss = loss_function(a, b, x, y)
print(a,b,loss)
plt.scatter(x,y)
plt.plot(x,prediction)
```

Output:

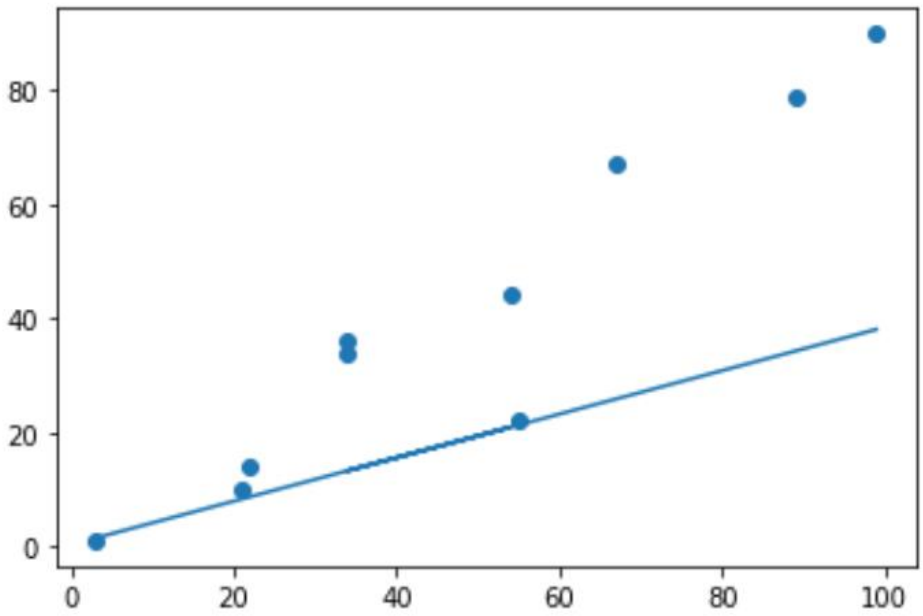

Figure 2-2 Iterate 1 time

Step 2 In the second iteration, the parameter values, loss values and visualization effects after the iteration are displayed

Code:

```
a,b = iterate(a,b,x,y,2)
prediction=model(a,b,x)
loss = loss_function(a, b, x, y)
print(a,b,loss)
plt.scatter(x,y)
plt.plot(x,prediction)
```

Output:

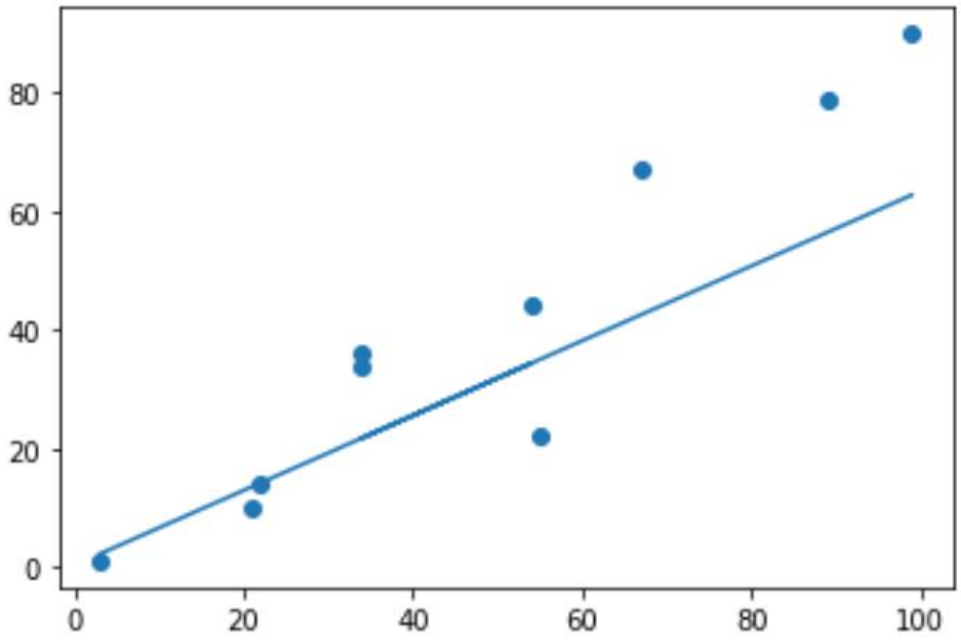

Figure 2-3 Iterate 2 times

Step 3 The third iteration shows the parameter values, loss values and visualization after iteration

Code:

```
a,b = iterate(a,b,x,y,3)
prediction=model(a,b,x)
loss = loss_function(a, b, x, y)
print(a,b,loss)
plt.scatter(x,y)
plt.plot(x,prediction)
```

Output:

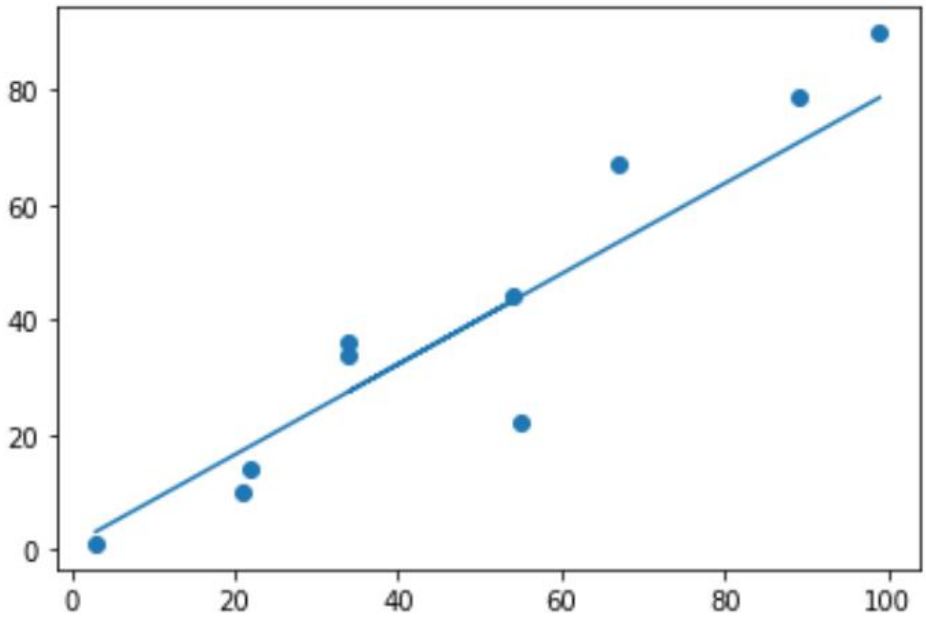

Figure 2-4 Iterate 3 times

Step 4 In the fourth iteration, parameter values, loss values and visualization effects are displayed

Code:

```
a,b = iterate(a,b,x,y,4)
prediction=model(a,b,x)
loss = loss_function(a, b, x, y)
print(a,b,loss)
plt.scatter(x,y)
plt.plot(x,prediction)
```

Output:

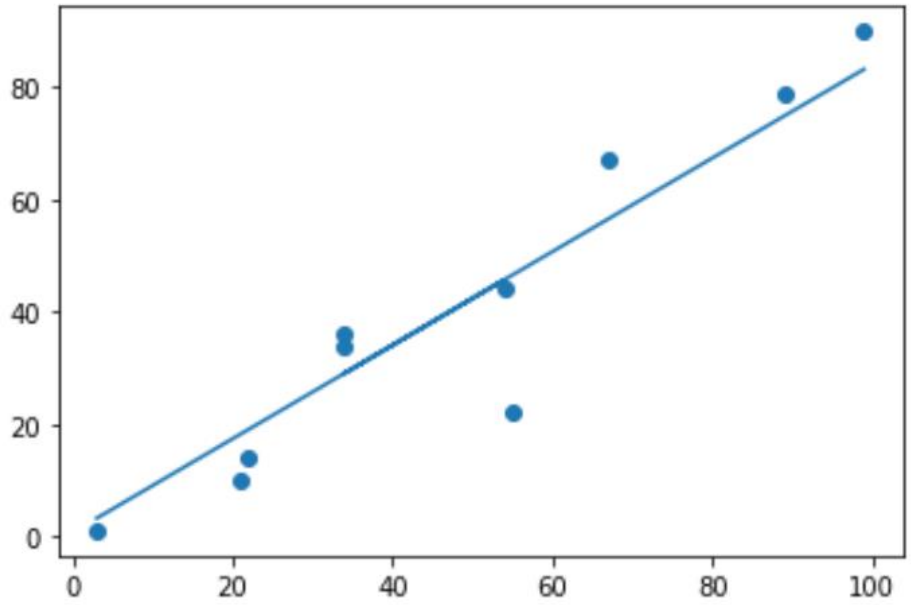

Figure 2-5 Iterate 4 times

Step 5 The fifth iteration shows the parameter value, loss value and visualization effect after iteration

Code:

```
a,b = iterate(a,b,x,y,5)
prediction=model(a,b,x)
loss = loss_function(a, b, x, y)
print(a,b,loss)
plt.scatter(x,y)
plt.plot(x,prediction)
```

Output:

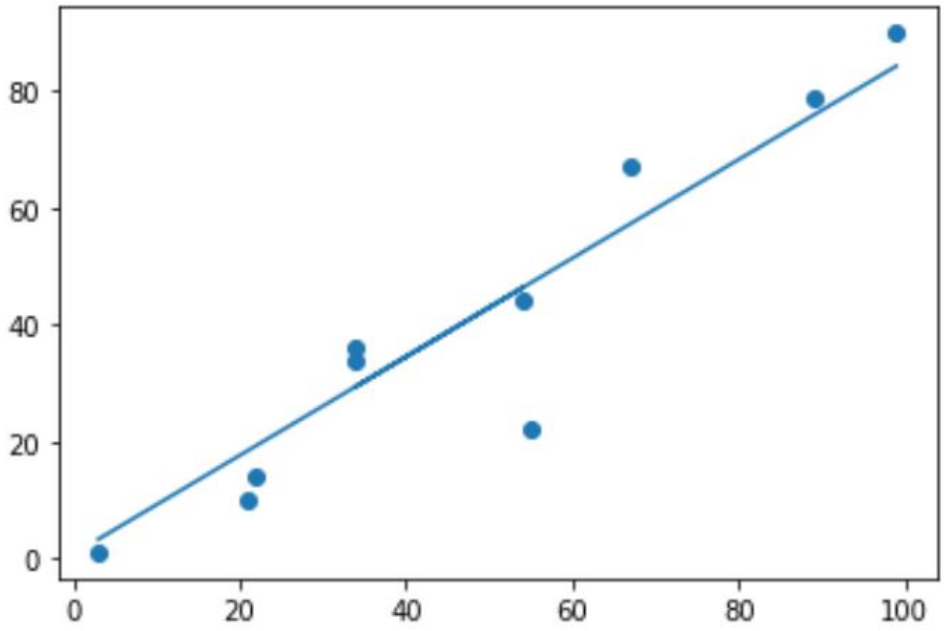

Figure 2-6 Iterate 5 times

Step 6 The 10000th iteration, showing the parameter values, losses and visualization after iteration

Code:

```
a,b = iterate(a,b,x,y,10000)
prediction=model(a,b,x)
loss = loss_function(a, b, x, y)
print(a,b,loss)
plt.scatter(x,y)
plt.plot(x,prediction)
```

Output:

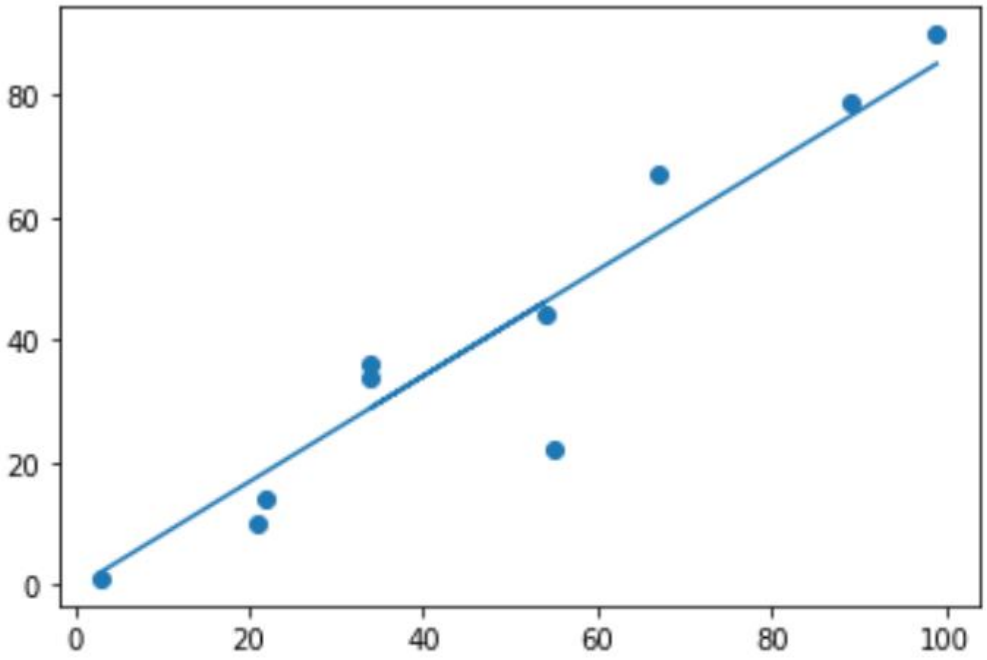

Figure 2-7 Iterate 10000 times

2.3 Thinking and practice

2.3.1 Question 1

Try to modify the original data yourself, Think about it: Does the loss value have to go to zero?

2.3.2 Question 2

Modify the values of Lr, Think: What is the role of the Lr parameter?

3 Decision tree details

3.1 Introduction

3.1.1 About This Experiment

This experiment focuses on the decision tree algorithm through the basic Python code.

It mainly uses Numpy module, Pandas module and Math module. We will implement the CART tree（Classification and Regressiontree models) in this experiment.

You have to download the dataset before this experiment through this link:

https://data-certification.obs.cn-east-2.myhuaweicloud.com/ENG/HCIA-AI/V3.0/ML-Dataset.rar

3.1.2 Objectives

The purpose of this experiment is as follows:

- Familiar with basic Python syntax
- Master the principle of Classification tree and implement with Python code
- Master the principle of Regression tree and implement with Python code

3.2 Experiment Code

3.2.1 Import the modules you need

Pandas is a tabular data processing module.

Math is mainly used for mathematical calculations.

Numpy is the basic computing module.

Code:

```
import pandas as pd
import math
import numpy as np
```

3.2.2 Superparameter definition section

Here you can choose to use Classification tree or Regression tree. Specifies the address of the dataset. Get feature name. Determine whether the algorithm matches the data set

Code:

```
algorithm = "Regression"    #    Algorithm: Classification, Regression
algorithm = "Classification"    #    Algorithm: Classification, Regression

# Dataset1: Text features and text labels
#df = pd.read_csv("D:/Code/Decision Treee/candidate/decision-trees-for-ml-master/decision-trees-for-ml-master/dataset/golf.txt")

# Dataset2: Mix features and Numeric labels, here you have to change the path to yours.
df = pd.read_csv("ML-Dataset/golf4.txt")

# This dictionary is used to store feature types of continuous numeric features and discrete literal features for subsequent judgment
dataset_features = dict()

num_of_columns = df.shape[1]-1
#The data type of each column of the data is saved for displaying the data name
for i in range(0, num_of_columns):
    #Gets the column name and holds the characteristics of a column of data by column
    column_name = df.columns[i]
    #Save the type of the data
    dataset_features[column_name] = df[column_name].dtypes
# The size of the indent when display
root = 1

# If the algorithm selects a regression tree but the label is not a continuous value, an error is reported
if algorithm == 'Regression':
    if df['Decision'].dtypes == 'object':
        raise ValueError('dataset wrong')
# If the tag value is continuous, the regression tree must be used
if df['Decision'].dtypes != 'object':
    algorithm = 'Regression'
    global_stdev = df['Decision'].std(ddof=0)
```

3.2.3 Define the functions required to complete the algorithm

Step 1 ProcessContinuousFeatures: Used to convert a continuous digital feature into a category feature.

Code:

```
# This function is used to handle numeric characteristics
def processContinuousFeatures(cdf, column_name, entropy):
    # Numerical features are arranged in order
    unique_values = sorted(cdf[column_name].unique())

    subset_ginis = [];
    subset_red_stdevs = []

    for i in range(0, len(unique_values) - 1):
        threshold = unique_values[i]
        # Find the segmentation result if the first number is used as the threshold
        subset1 = cdf[cdf[column_name] <= threshold]
```

```
            subset2 = cdf[cdf[column_name] > threshold]
            # Calculate the proportion occupied by dividing the two parts
            subset1_rows = subset1.shape[0];
            subset2_rows = subset2.shape[0]
            total_instances = cdf.shape[0]
            # In the text feature part, entropy is calculated by using the cycle,
            # and in the numeric part, entropy is calculated by using the two groups after segmentation,
            # and the degree of entropy reduction is obtained
            if algorithm == 'Classification':
                decision_for_subset1 = subset1['Decision'].value_counts().tolist()
                decision_for_subset2 = subset2['Decision'].value_counts().tolist()

                gini_subset1 = 1;
                gini_subset2 = 1

                for j in range(0, len(decision_for_subset1)):
                    gini_subset1 = gini_subset1 - math.pow((decision_for_subset1[j] / subset1_rows), 2)

                for j in range(0, len(decision_for_subset2)):
                    gini_subset2 = gini_subset2 - math.pow((decision_for_subset2[j] / subset2_rows), 2)

                gini = (subset1_rows / total_instances) * gini_subset1 + (subset2_rows / total_instances)
* gini_subset2

                subset_ginis.append(gini)

            # Take standard deviation as the judgment basis, calculate the decrease value of standard
deviation at this time
            elif algorithm == 'Regression':
                superset_stdev = cdf['Decision'].std(ddof=0)
                subset1_stdev = subset1['Decision'].std(ddof=0)
                subset2_stdev = subset2['Decision'].std(ddof=0)

                threshold_weighted_stdev = (subset1_rows / total_instances) * subset1_stdev + (
                        subset2_rows / total_instances) * subset2_stdev
                threshold_reducted_stdev = superset_stdev - threshold_weighted_stdev
                subset_red_stdevs.append(threshold_reducted_stdev)

        #Find the index of the split value
        if algorithm == "Classification":
            winner_one = subset_ginis.index(min(subset_ginis))
        elif algorithm == "Regression":
            winner_one = subset_red_stdevs.index(max(subset_red_stdevs))
        # Find the corresponding value according to the index
        winner_threshold = unique_values[winner_one]

        # Converts the original data column to an edited string column.
        # Characters smaller than the threshold are modified with the <= threshold value
        cdf[column_name] = np.where(cdf[column_name] <= winner_threshold, "<=" +
str(winner_threshold),">" + str(winner_threshold))

        return cdf
```

Step 2 CalculateEntropy: Used to calculate Gini or variances, they are the criteria for classifying.

Code:

```
# This function calculates the entropy of the column, and the input data must contain the Decision column
def calculateEntropy(df):
    # The regression tree entropy is 0
    if algorithm == 'Regression':
        return 0

    rows = df.shape[0]
    # Use Value_counts to get all values stored as dictionaries, keys: finds keys, and Tolist: change to lists.
    # This line of code finds the tag value.
    decisions = df['Decision'].value_counts().keys().tolist()

    entropy = 0
    # Here the loop traverses all the tags
    for i in range(0, len(decisions)):
        # Record the number of times the tag value appears
        num_of_decisions = df['Decision'].value_counts().tolist()[i]
        # probability of occurrence
        class_probability = num_of_decisions / rows
        # Calculate the entropy and sum it up
        entropy = entropy - class_probability * math.log(class_probability, 2)

    return entropy
```

Step 3 FindDecision: Find which feature in the current data to classify.

Code:

```
# The main purpose of this function is to traverse the entire column of the table,
# find which column is the best split column, and return the name of the column
def findDecision(ddf):
    # If it's a regression tree, then you take the standard deviation of the true value
    if algorithm == 'Regression':
        stdev = ddf['Decision'].std(ddof=0)
    # Get the entropy of the decision column
    entropy = calculateEntropy(ddf)

    columns = ddf.shape[1];
    rows = ddf.shape[0]
    # Used to store Gini and standard deviation values
    ginis = [];
    reducted_stdevs = []
    # Traverse all columns and calculate the relevant indexes of all columns according to algorithm selection
    for i in range(0, columns - 1):
        column_name = ddf.columns[i]
        column_type = ddf[column_name].dtypes

        # Determine if the column feature is a number, and if so, process the data using the following function,
        # which modifies the data to a string type category on return.
```

```
        # The idea is to directly use character characteristics, continuous digital characteristics into
discrete character characteristics
        if column_type != 'object':
            ddf = processContinuousFeatures(ddf, column_name, entropy)
        # The statistical data in this column can be obtained, and the continuous data can be
directly classified after processing,
        # and the categories are less than the threshold and greater than the threshold
        classes = ddf[column_name].value_counts()
        gini = 0;
        weighted_stdev = 0
        # Start the loop with the type of data in the column
        for j in range(0, len(classes)):
            current_class = classes.keys().tolist()[j]
            # The final classification result corresponding to the data is selected
            # by deleting the value of the df column equal to the current data
            subdataset = ddf[ddf[column_name] == current_class]

            subset_instances = subdataset.shape[0]
            # The entropy of information is calculated here
            if algorithm == 'Classification':   # GINI index
                decision_list = subdataset['Decision'].value_counts().tolist()

                subgini = 1

                for k in range(0, len(decision_list)):
                    subgini = subgini - math.pow((decision_list[k] / subset_instances), 2)

                gini = gini + (subset_instances / rows) * subgini
            # The regression tree is judged by the standard deviation,
            # and the standard deviation of the subclasses in this column is calculated here
            elif algorithm == 'Regression':
                subset_stdev = subdataset['Decision'].std(ddof=0)
                weighted_stdev = weighted_stdev + (subset_instances / rows) * subset_stdev

        # Used to store the final value of this column
        if algorithm == "Classification":
            ginis.append(gini)
        # Store the decrease in standard deviation for all columns
        elif algorithm == 'Regression':
            reducted_stdev = stdev - weighted_stdev
            reducted_stdevs.append(reducted_stdev)

    # Determine which column is the first branch
    # by selecting the index of the largest value from the list of evaluation indicators
    if algorithm == "Classification":
        winner_index = ginis.index(min(ginis))
    elif algorithm == "Regression":
        winner_index = reducted_stdevs.index(max(reducted_stdevs))
    winner_name = ddf.columns[winner_index]

    return winner_name
```

Step 4 FormatRule: Standardize the final output format.

Code:

```
# ROOT is a number used to generate ' 'to adjust the display format of the decision making process
def formatRule(root):
    resp = ''

    for i in range(0, root):
        resp = resp + '    '

    return resp
```

Step 5 BuildDecisionTree: Main function.

Code:

```
# With this function, you build the decision tree model,
# entering data in dataframe format, the root value, and the file address
# If the value in the column is literal, it branches directly by literal category
def buildDecisionTree(df, root):
    # Identify the different charForResp
    charForResp = '"'
    if algorithm == 'Regression':
        charForResp = ''

    tmp_root = root * 1

    df_copy = df.copy()
    # Output the winning column of the decision tree, enter a list,
    # and output the column name of the decision column in the list
    winner_name = findDecision(df)

    # Determines whether the winning column is a number or a character
    numericColumn = False
    if dataset_features[winner_name] != 'object':
        numericColumn = True

    # To ensure the integrity of the original data and prevent the data from changing,
    # mainly to ensure that the data of other columns besides the winning column does not change,
    # so as to continue the branch in the next step.
    columns = df.shape[1]
    for i in range(0, columns - 1):
        column_name = df.columns[i]
        if df[column_name].dtype != 'object' and column_name != winner_name:
            df[column_name] = df_copy[column_name]
    # Find the element in the branching column
    classes = df[winner_name].value_counts().keys().tolist()
    # Traversing all classes in the branch column has two functions:
    # 1. Display which class is currently traversed to; 2. Determine whether the current class is
already leaf node
    for i in range(0, len(classes)):
        # Find the Subdataset as in FindDecision, but discard this column of the current branch
        current_class = classes[i]
        subdataset = df[df[winner_name] == current_class]
        # At the same time, the data of the first branch column is discarded and the remaining data
is processed
        subdataset = subdataset.drop(columns=[winner_name])
```

```
        # Edit the display situation. If it is a numeric feature, the character conversion has been
completed when searching for branches.
        #If it is not a character feature, it is displayed with column names
        if numericColumn == True:
            compareTo = current_class    # current class might be <=x or >x in this case
        else:
            compareTo = " == '" + str(current_class) + "'"

        terminateBuilding = False

        # ----------------------------------------------

        # This determines whether it is already the last leaf node
        if len(subdataset['Decision'].value_counts().tolist()) == 1:
            final_decision = subdataset['Decision'].value_counts().keys().tolist()[
                0]    # all items are equal in this case
            terminateBuilding = True
        # At this time, only the Decision column is left, that is, all the segmentation features have
been used
        elif subdataset.shape[1] == 1:
            # get the most frequent one
            final_decision = subdataset['Decision'].value_counts().idxmax()
            terminateBuilding = True
        # The regression tree is judged as leaf node if the number of elements is less than 5
        #elif algorithm == 'Regression' and subdataset.shape[0] < 5:    # pruning condition
        # Another criterion is to take the standard deviation as the criterion and the sample mean in
the node as the value of the node
        elif algorithm == 'Regression' and subdataset['Decision'].std(ddof=0)/global_stdev < 0.4:
            # get average
            final_decision = subdataset['Decision'].mean()
            terminateBuilding = True
        # ----------------------------------------------
        # Here we begin to output the branching results of the decision tree.。

        print(formatRule(root), "if ", winner_name, compareTo, ":")

        # ----------------------------------------------
        # check decision is made
        if terminateBuilding == True:
            print(formatRule(root + 1), "return ", charForResp + str(final_decision) + charForResp)
        else:    # decision is not made, continue to create branch and leafs
            # The size of the indent at display represented by root
            root = root + 1
            # Call this function again for the next loop
            buildDecisionTree(subdataset, root)

        root = tmp_root * 1
```

3.2.4 Execute the code

Code:

```
# call the function
buildDecisionTree(df, root)
```

Output:

```
if Outlook == 'Sunny' :
    if Temp. <=83 :
        if Wind == 'Strong' :
            if Humidity <=95 :
                return 30
        if Wind == 'Weak' :
            return 36.5
    if Temp. >83 :
        return 25
if Outlook == 'Rain' :
    if Wind == 'Weak' :
        return 47.666666666666664
    if Wind == 'Strong' :
        return 26.5
if Outlook == 'Overcast' :
    return 46.25
```

Figure 3-1 Regression tree result

```
if Outlook == 'Sunny' :
    if Humidity == 'High' :
        return 'No'
    if Humidity == 'Normal' :
        return 'Yes'
if Outlook == 'Rain' :
    if Wind == 'Weak' :
        return 'Yes'
    if Wind == 'Strong' :
        return 'No'
if Outlook == 'Overcast' :
    return 'Yes'
```

Figure 3-2 CART tree result