Gaussian Naive Bayes Classifier: Iris Data Set

Nguyen Van Hai
Nguyen Tien Dung
Dao Anh Huy
Luu Thanh Duy

Ton Duc Thang University

Outline

- Overview
  - Iris Data Set
  - Bayes Theorem
  - Normal distribution
- Prepare Data
  - Load Data
  - Split Data
  - Group Data
- Summarize Data
  - Mean
  - Standard Deviation
  - Summary

Outline

- Build Model
  - Overview
  - Prior Probability
  - Likelihood
  - Joint Probability
  - Marginal Probability
  - Posterior Probability
- Test Model
  - Get Maximum A Posterior
  - Predict
  - Accuracy
- Code
- Results
  - Print the results

Overview

The data set has 4 independent variables and 1 dependent variable that have 3 different classes with 150 instances.
- The first 4 columns are the independent variables (features).
- The 5th column is the dependent variable (class).

1. sepal length (cm)
2. sepal width (cm)
3. petal length (cm)
4. petal width (cm)
5. class:
    - Iris Setosa
    - Iris Versicolour
    - Iris Virginica

For example:

sepal_length	sepal_width	petal_length	petal_width	species
5	2	3.5	1	versicolor
6	2.2	4	1	versicolor
6	2.2	5	1.5	virginica
6.2	2.2	4.5	1.5	versicolor
4.5	2.3	1.3	0.3	setosa

Figure: Random 5 Row Sample

Bayes Theorem

Naive Bayes, more technically referred to as the Posterior Probability, updates the prior belief of an event given new information. The result is the probability of the class occuring given the new data.

$$P(class/features) = \frac{P(class) * P(features/class)}{P(features)}$$

- P(class/features) : Posterior Probability
- P(class) : Class Prior Probability
- P(features/class) : Likelihood
- P(features) : Predictor Prior Probability

The probability density of the normal distribution is:

$$f(x|\mu,\, \sigma^2) = \frac{1}{\sqrt{2\,\pi\sigma^2}}\,\mathrm{e}^{-\frac{(x-\mu)^2}{2\,\sigma^2}} \qquad (1)$$

Where

- 'μ' is the mean or expectation of the distribution,
- 'σ' is the standard deviation, and
- 'σ^2' is the variance.

Prepare Data

Read in the raw data and convert each string into an integer.

```
class read_write_data:

    def __init__(self):
        pass

    #read training data
    def read_data(self, filename, header=False):
        training_set = []         #list of data - include targets
        data = []                 #temp list for data

        #read file
        with open(filename) as file:
            lines = csv.reader(file)
            data = list(lines)
        file.close()

        #check, if the file has header, then we just get values of features
        if header:
            data = data[1:]

        #Converting string to float for numeric molecules
        for set in data:
            temp = [float(x) if re.search('\d', x) else x for x in set]
            training_set.append(temp)

        return training_set
```

ĐẠI HỌC TÔN ĐỨC THẮNG

Split the data into a training_set and a testing_set.

The weight will determine how much of the data will be in the training_set.

```
#Randomly selects rows for training according to the weight and uses the rest of the rows for testing
def split_data(self, training_data, weight):
    training_size = int(len(training_data)*weight)    #get size of training data
    training_set = []                                # the training set that we had yet
    for i in range(training_size):
        index = random.randrange(len(training_data))
        training_set.append(training_data[index])
        training_data.pop(index)
    testing_set = training_data

    return [training_set, testing_set]
```

Group the data according to class by mapping each class to individual instances.

```
#Mapping each target to a list of it's features
def group_by_class(self, training_data, target):
    training_set_group = defaultdict(list)
    for index in range(len(training_data)):
        a_training_set = training_data[index]
        if not a_training_set:
            continue
        temp = a_training_set[target]
        training_set_group[temp].append(a_training_set[:-1])
    #print(dict(training_set_group))

    return dict(training_set_group)
```

ĐẠI HỌC TÔN ĐỨC THẮNG

Summarize Data

Calculate the mean.

```
#calculate the mean of training data
def mean(self, values_of_feature):
    return sum(values_of_feature) / float(len(values_of_feature))
```

Calculate the standard deviation.

```
#calculate the standard deviation for a list of data
def standard_deviation(self, values_of_feature):
    avg = self.mean(values_of_feature)
    squared_diff_list = []
    for num in values_of_feature:
        squared_diff = (num - avg) **2
        squared_diff_list.append(squared_diff)
    squared_diff_sum = sum(squared_diff_list)
    var = squared_diff_sum / float(len(values_of_feature) - 1)

    return var ** .5
```

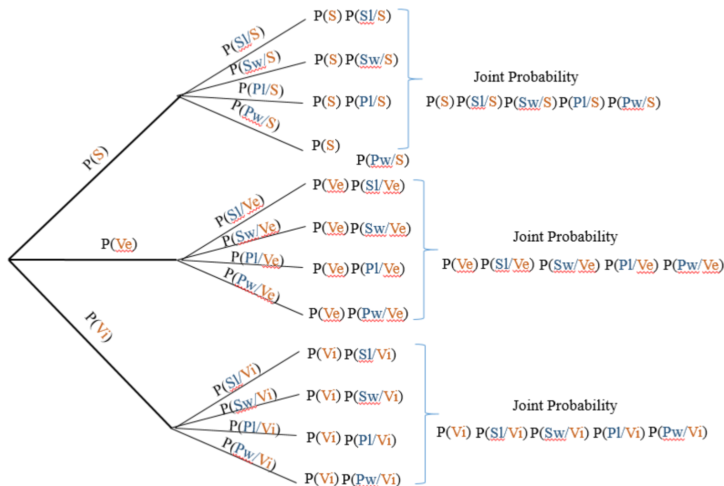
Return the (mean, standard deviation) combination for each feature of the training_set. The mean and the standard deviation will be used when calculating the Normal Probabiltiy values for each feature of the testing_set.

```
#Use zip to line up each feature into a single column across multiple lists - yield the mean and the stdev for each feature.
def summarize(self, training_set):
    for list in zip(*training_set):
        yield {
                'Standard_Deviation': self.standard_deviation(list),
                'Mean': self.mean(list)
                }
```

Build Model

Features	**Class**
Sl : sepal length	S : Iris-setosa
Sw : sepal width	Ve : Iris-versicolor
Pl : petal length	Vi : Iris-virganica
Pw : petal width	

Overview : Bayes Tree Diagram



Prior Probability

Prior Probability is what we know about each class before considering the new data.

It's the probability of each class occurring.

```
#calculate the probability of each target class
def prior_probability(self, group, target, training_data):
    return len(group[target]) / float(len(training_data))
```

Train

This is where we learn from the train set, by calculating the mean and the standard deviation.

Using the grouped classes, calculate the (mean, standard deviation) combination for each feature of each class.

The calculations will later use the (mean, standard deviation) of each feature to calculate class likelihoods.

```
#return each target: the probability of each class and list of {'mean : 0.0', 'standard_deviation : 0.0'}
def train(self, training_data, target):
    group = self.group_by_class(training_data, target)
    self.summarizes = {}
    for target, features in group.items():
        self.summarizes[target] = {
                'prior_probability': self.prior_probability(group, target, training_data),
                'summary': [i for i in self.summarize(features)],
                }

    return self.summarizes
```

Likelihood is calculated by taking the product of all Normal Probabilities.

$$P(\text{features}/\text{class})$$

For each feature given the class we calculate the Normal Probability using the Normal Distribution.

$$P(Sl/S)P(Sw/S)P(Pl/S)P(Pw/S)$$

```
#Gaussian (Normal) Density function
def normal_probability(self, x, mean, stdev):
    variance = stdev ** 2
    exp_squared_diff = (x - mean) ** 2
    exp_power = -exp_squared_diff / (2 * variance)
    exponent = e ** exp_power
    denominator = ((2 * pi) ** .5) * stdev
    normal_prob = exponent / denominator

    return normal_prob
```

Joint Probability is calculated by taking the product of the Prior Probability and the Likelihood.

$$P(S)P(Sl/S)P(Sw/S)P(Pl/S)P(Pw/S)$$

```
#Take the product of all Normal Probabilities and the Prior Probability
def joint_probability(self, testing_set):
    joint_probs = {}
    for target, features in self.summarizes.items():
        total_features = len(features['summary'])
        likelihood = 1
        for index in range(total_features):
            feature = testing_set[index]
            mean = features['summary'][index]['Mean']
            stdev = features['summary'][index]['Standard_Deviation']
            normal_prob = self.normal_probability(feature, mean, stdev)
            likelihood *= normal_prob
        prior_prob = features['prior_probability']
        joint_probs[target] = prior_prob * likelihood

    return joint_probs
```

Marginal Probability

Calculate the total sum of all joint probabilities.

$$\text{Marginal_probs} = P(S)P(Sl/S)P(Sw/S)P(Pl/S)P(Pw/S) + P(Ve)P(Sl/Ve)P(Sw/Ve)P(Pl/Ve)P(Pw/Ve) + P(Vi)P(Sl/Vi)P(Sw/Vi)P(Pl/Vi)P(Pw/Vi)$$

```
#Marginal Probability Density Function (Predictor Prior Probability)
def marginal_probability(self, joint_probability):
    marginal_prob = sum(joint_probability.values())

    return marginal_prob
```

The Posterior Probability is the probability of a class occuring and is calculated for each class given the new data.

P(class/features)

This where all of the preceding class methods tie together to calculate the Gauss Naive Bayes formula with the goal of selecting MAP.

```
#return a dictionary mapping of class to it's posterior probability
def posterior_probability(self, testing_set):
    posterior_probs = {}
    joint_probability = self.joint_probability(testing_set)
    marginal_prob = self.marginal_probability(joint_probability)
    for target, joint_prob in joint_probability.items():
        posterior_probs[target] = joint_prob / marginal_prob
    #print(posterior_probs)
    return posterior_probs
```

ĐẠI HỌC TÔN ĐỨC THẮNG

Test Model

The get_best_posterior_probability() method will call the posterior_probability() method on a single test_row.

For each test_row we will calculate 3 Posterior Probabilities; one for each class. The goal is to select MAP, the Maximum A Posterior probability.

The get_best_posterior_probability() method will simply choose the Maximum A Posterior probability and return the associated class for the given test_row.

```
#Return the target class with the best posterior probability
def get_best_posterior_probability(self, testing_set):
    posterior_probs = self.posterior_probability(testing_set)
    max_prob = max(posterior_probs, key = posterior_probs.get)

    return max_prob
```

This method will return a prediction for each test_row.

```
#return a list of predicted targets
def get_list_of_predicted_targets(self, testing_set):
    max_probs = []
    for row in testing_set:
        max_prob = self.get_best_posterior_probability(row)
        max_probs.append(max_prob)

    return max_probs
```

Accuracy will test the performance of the model by taking the total number of correct predictions and divide them by the total number of predictions. This is critical in understanding the veracity of the model.

```
#Calculate the average performance of the classifier
def average_performance_of_classifier(self, testing_set, predicted):
    correct = 0
    actual = [item[-1] for item in testing_set]
    for x, y in zip(actual, predicted):
        if x == y:
            correct += 1
    return correct / float(len(testing_set))
```

Code

```python
# -*- coding: utf-8 -*-
"""
Created on Sun Apr 28 13:26:12 2019

@author: Hai Nguyen
"""

from collections import defaultdict
from math import pi
from math import e
import random
import csv
import re
import sys
```

```
class read_write_data:

    def __init__(self):
        pass

    #read training data
    def read_data(self, filename, header=False):
        training_set = []         #list of data - include targets
        data = []                 #temp list for data

        #read file
        with open(filename) as file:
            lines = csv.reader(file)
            data = list(lines)
        file.close()

        #check, if the file has header, then we just get values of features
        if header:
            data = data[1:]

        #Converting string to float for numeric molecules
        for set in data:
            temp = [float(x) if re.search('\d', x) else x for x in set]
            training_set.append(temp)

        return training_set
```

```python
#write data into the csv files
def write_data(self, filename, data1, data2, data3, data4):
    with open(filename, 'w') as file:
        writer = csv.writer(file)
        writer.writerow(data1)
        text1 = {'==================================================='}
        writer.writerow(text1)
        for row in data2:
            writer.writerow(list(row))
        writer.writerow(text1)
        text2 = {'The results of the classes corresponding to the data to be checked are: '}
        writer.writerow(text2)
        for row1 in data3:
            writer.writerow(list(row1))
        writer.writerow(text1)
        writer.writerow(data4)

    file.close()
```

```python
class gaussian_naive_bayes:

    #Randomly selects rows for training according to the weight and uses the rest of the rows for testing
    def split_data(self, training_data, weight):
        training_size = int(len(training_data)*weight)    #get size of training data
        training_set = []                                 # the training set that we had yet
        for i in range(training_size):
            index = random.randrange(len(training_data))
            training_set.append(training_data[index])
            training_data.pop(index)
        testing_set = training_data

        return [training_set, testing_set]

    #Mapping each target to a list of it's features
    def group_by_class(self, training_data, target):
        training_set_group = defaultdict(list)
        for index in range(len(training_data)):
            a_training_set = training_data[index]
            if not a_training_set:
                continue
            temp = a_training_set[target]
            training_set_group[temp].append(a_training_set[:-1])
        #print(dict(training_set_group))

        return dict(training_set_group)
```

Code


```python
#calculate the mean of training data
def mean(self, values_of_feature):
    return sum(values_of_feature) / float(len(values_of_feature))

#calculate the standard deviation for a list of data
def standard_deviation(self, values_of_feature):
    avg = self.mean(values_of_feature)
    squared_diff_list = []
    for num in values_of_feature:
        squared_diff = (num - avg) **2
        squared_diff_list.append(squared_diff)
    squared_diff_sum = sum(squared_diff_list)
    var = squared_diff_sum / float(len(values_of_feature) - 1)

    return var ** .5

#Use zip to line up each feature into a single column across multiple lists - yield the mean and the stdev for each feature.
def summarize(self, training_set):
    for list in zip(*training_set):
        yield {
            'Standard_Deviation': self.standard_deviation(list),
            'Mean': self.mean(list)
            }
```

```python
#calculate the probability of each target class
def prior_probability(self, group, target, training_data):
    return len(group[target]) / float(len(training_data))

#return each target: the probability of each class and list of {'mean : 0.0', 'standard_deviation : 0.0'}
def train(self, training_data, target):
    group = self.group_by_class(training_data, target)
    self.summarizes = {}
    for target, features in group.items():
        self.summarizes[target] = {
                'prior_probability': self.prior_probability(group, target, training_data),
                'summary': [i for i in self.summarize(features)],
                }

    return self.summarizes

#Gaussian (Normal) Density function
def normal_probability(self, x, mean, stdev):
    variance = stdev ** 2
    exp_squared_diff = (x - mean) ** 2
    exp_power = -exp_squared_diff / (2 * variance)
    exponent = e ** exp_power
    denominator = ((2 * pi) ** .5) * stdev
    normal_prob = exponent / denominator

    return normal_prob
```

Code


```
#Marginal Probability Density Function (Predictor Prior Probability)
def marginal_probability(self, joint_probability):
    marginal_prob = sum(joint_probability.values())

    return marginal_prob

#Take the product of all Normal Probabilities and the Prior Probability
def joint_probability(self, testing_set):
    joint_probs = {}
    for target, features in self.summarizes.items():
        total_features = len(features['summary'])
        likelihood = 1
        for index in range(total_features):
            feature = testing_set[index]
            mean = features['summary'][index]['Mean']
            stdev = features['summary'][index]['Standard_Deviation']
            normal_prob = self.normal_probability(feature, mean, stdev)
            likelihood *= normal_prob
        prior_prob = features['prior_probability']
        joint_probs[target] = prior_prob * likelihood

    return joint_probs
```

Code


```
#return a dictionary mapping of class to it's posterior probability
def posterior_probability(self, testing_set):
    posterior_probs = {}
    joint_probability = self.joint_probability(testing_set)
    marginal_prob = self.marginal_probability(joint_probability)
    for target, joint_prob in joint_probability.items():
        posterior_probs[target] = joint_prob / marginal_prob
    #print(posterior_probs)
    return posterior_probs

#Return the target class with the best posterior probability
def get_best_posterior_probability(self, testing_set):
    posterior_probs = self.posterior_probability(testing_set)
    max_prob = max(posterior_probs, key = posterior_probs.get)

    return max_prob
```

```
#return a list of predicted targets
def get_list_of_predicted_targets(self, testing_set):
    max_probs = []
    for row in testing_set:
        max_prob = self.get_best_posterior_probability(row)
        max_probs.append(max_prob)

    return max_probs

#Calculate the average performance of the classifier
def average_performance_of_classifier(self, testing_set, predicted):
    correct = 0
    actual = [item[-1] for item in testing_set]
    for x, y in zip(actual, predicted):
        if x == y:
            correct += 1
    return correct / float(len(testing_set))
```

Code


```
def main(args):
    class_rwd = read_write_data()
    class_gnb = gaussian_naive_bayes()
    training_data = class_rwd.read_data("iris.csv", header = True)

    [training_set, testing_set] = class_gnb.split_data(training_data, 0.67)
    class_gnb.train(training_set, -1)

    predicted = class_gnb.get_list_of_predicted_targets(testing_set)
    accuracy = class_gnb.average_performance_of_classifier(testing_set, predicted)

    #print and write results into results.csv file
    print("Using %s rows for training and %s rows for testing!" % (len(training_set), len(testing_set)))
    data1 = {"Using %s rows for training and %s rows for testing!" % (len(training_set), len(testing_set))}
    print("=================================================")
    data2 = []
    for row in testing_set:
        posterior_probs = class_gnb.posterior_probability(row)
        data2.append(posterior_probs.items())
        print(posterior_probs)
    print("=================================================")
    print("The results of the classes corresponding to the data to be checked are: ")
    temp_testing_set = []
    for row in testing_set:
        temp_testing_set.append(row[:-1])
    data3 = []
    for i, j in zip(temp_testing_set, predicted):
        print("Testing set: %s " % i, " --> It can be: %s species." % j)
        temp_text = {"Testing set: %s  --> It can be: %s species." % (i, j)}
        data3.append(temp_text)
    print("Accuracy: %.3f" % accuracy)
    data4 = {"Accuracy: %.3f" % accuracy}

    class_rwd.write_data('results.csv', data1, data2, data3, data4)

if __name__ == '__main__':
    main(sys.argv)
```

Results

Print the results


```
Using 100 rows for training and 50 rows for testing!
==================================================
{'versicolor': 7.608205827154341e-17, 'virginica': 1.0395298548389571e-23, 'setosa': 1.0}
{'versicolor': 5.813854020488955e-16, 'virginica': 2.3583286384836388e-23, 'setosa': 0.9999999999999994}
{'versicolor': 7.423337288313684e-13, 'virginica': 2.7477506762852405e-19, 'setosa': 0.9999999999992577}
{'versicolor': 3.3794345809847667e-16, 'virginica': 2.420676049323138e-23, 'setosa': 0.9999999999999997}
{'versicolor': 3.3151200867354663e-16, 'virginica': 3.753264227848916e-23, 'setosa': 0.9999999999999997}
{'versicolor': 1.775879216258757e-16, 'virginica': 1.555099459524475e-23, 'setosa': 0.999999999999998}
{'versicolor': 3.26295469554402e-15, 'virginica': 6.0012252507676586e-21, 'setosa': 0.999999999999967}
{'versicolor': 9.115116335118537e-16, 'virginica': 8.813361948786339e-23, 'setosa': 0.999999999999991}
{'versicolor': 7.156883679525364e-14, 'virginica': 1.2510888454320313e-20, 'setosa': 0.9999999999999284}
{'versicolor': 7.918320638454333e-13, 'virginica': 1.4165922926199588e-19, 'setosa': 0.9999999999992082}
{'versicolor': 1.4813911795244634e-14, 'virginica': 7.514993590174295e-22, 'setosa': 0.9999999999999852}
{'versicolor': 4.1976674557697647e-16, 'virginica': 6.609354883237713e-23, 'setosa': 0.9999999999999996}
{'versicolor': 4.957861262757422e-18, 'virginica': 6.347951833686384e-24, 'setosa': 1.0}
{'versicolor': 1.775879216258757e-16, 'virginica': 1.555099459524475e-23, 'setosa': 0.999999999999998}
{'versicolor': 2.1577256612096433e-16, 'virginica': 3.514483114647478e-23, 'setosa': 0.9999999999999999}
{'versicolor': 4.695925507783814e-11, 'virginica': 1.434443466151134e-17, 'setosa': 0.999999999530408}
{'versicolor': 0.7743863639953067, 'virginica': 0.22561363600469336, 'setosa': 6.911771838487283e-116}
{'versicolor': 0.9371195399479558, 'virginica': 0.06288046005204413, 'setosa': 4.6135124122256164e-113}
{'versicolor': 0.9997855926652767, 'virginica': 0.00021440733472326753, 'setosa': 8.31336791234416e-73}
{'versicolor': 0.9934447043986492, 'virginica': 0.006555295601350893, 'setosa': 5.95257874893996e-91}
{'versicolor': 0.9999811650402203, 'virginica': 1.8834959779673082e-05, 'setosa': 8.695211336758893e-64}
{'versicolor': 0.9996032638525717, 'virginica': 0.00039673614742820736, 'setosa': 1.321735949345131e-75}
{'versicolor': 0.9060798304237504, 'virginica': 0.09392016957624955, 'setosa': 5.7347639432049923e-129}
{'versicolor': 0.9973165882077674, 'virginica': 0.0026834117922327006, 'setosa': 5.091677488269248e-90}
{'versicolor': 0.97899137254678, 'virginica': 0.021008627453220068, 'setosa': 7.214480490375632e-106}
{'versicolor': 0.5768315876079528, 'virginica': 0.4231684123920472, 'setosa': 2.460253126397837e-143}
{'versicolor': 0.8303390231255526, 'virginica': 0.1696609768744473, 'setosa': 6.392842963440205e-109}
{'versicolor': 0.999550273454734, 'virginica': 0.00044972654526601844, 'setosa': 5.490331575180847e-78}
{'versicolor': 0.9999992797911051, 'virginica': 7.20208894828783e-07, 'setosa': 1.034985445005222e-37}
{'versicolor': 0.9993970620016934, 'virginica': 0.0006029379983065072, 'setosa': 5.6366938595457174e-83}
{'versicolor': 0.9996654308785016, 'virginica': 0.00033456912149837056, 'setosa': 1.0950094740117818e-78}
{'versicolor': 3.5666158911858714e-10, 'virginica': 0.9999999996433384, 'setosa': 3.7465806387645317e-259}
{'versicolor': 5.602413415023557e-07, 'virginica': 0.9999994397586586, 'setosa': 1.774162111443364e-229}
{'versicolor': 7.10462011277672e-07, 'virginica': 0.9999992895379888, 'setosa': 3.9047316482684435e-225}
{'versicolor': 1.4927161626855389e-09, 'virginica': 0.9999999985072838, 'setosa': 7.879500735973066e-288}
{'versicolor': 4.044025937814406e-06, 'virginica': 0.9999959559740622, 'setosa': 5.159408936507201e-244}
{'versicolor': 0.015538006749004192, 'virginica': 0.9844619932509958, 'setosa': 1.6720935022000668e-157}
{'versicolor': 0.13653121887172756, 'virginica': 0.8634687811282724, 'setosa': 1.4284268271313648e-142}
{'versicolor': 1.8001223879050985e-05, 'virginica': 0.999981998776121, 'setosa': 6.935530804023583e-220}
{'versicolor': 0.20032634684720618, 'virginica': 0.7996736531527938, 'setosa': 2.619772759025068e-136}
{'versicolor': 2.4886361868981746e-05, 'virginica': 0.999975113638131, 'setosa': 1.6258626092691418e-204}
{'versicolor': 2.8111727814595917e-06, 'virginica': 0.9999971888272184, 'setosa': 8.060100242535088e-235}
{'versicolor': 1.4461673161838145e-09, 'virginica': 0.9999999985538327, 'setosa': 1.623242918455343e-266}
{'versicolor': 4.016633341539335e-06, 'virginica': 0.9999959833666584, 'setosa': 7.11305832594
{'versicolor': 0.6802713547474679, 'virginica': 0.3197286452525322, 'setosa': 5.40163262775952
```

ĐẠI HỌC TÔN ĐỨC THẮNG

Print the results


```
Testing set: [4.9, 3.1, 1.5, 0.1]    --> It can be: setosa species.
Testing set: [5.7, 4.4, 1.5, 0.4]    --> It can be: setosa species.
Testing set: [5.1, 3.5, 1.4, 0.3]    --> It can be: setosa species.
Testing set: [5.4, 3.4, 1.7, 0.2]    --> It can be: setosa species.
Testing set: [4.8, 3.4, 1.9, 0.2]    --> It can be: setosa species.
Testing set: [5.0, 3.0, 1.6, 0.2]    --> It can be: setosa species.
Testing set: [5.2, 3.5, 1.5, 0.2]    --> It can be: setosa species.
Testing set: [5.5, 4.2, 1.4, 0.2]    --> It can be: setosa species.
Testing set: [4.9, 3.1, 1.5, 0.1]    --> It can be: setosa species.
Testing set: [5.5, 3.5, 1.3, 0.2]    --> It can be: setosa species.
Testing set: [5.1, 3.8, 1.9, 0.4]    --> It can be: setosa species.
Testing set: [7.0, 3.2, 4.7, 1.4]    --> It can be: versicolor species.
Testing set: [6.5, 2.8, 4.6, 1.5]    --> It can be: versicolor species.
Testing set: [5.2, 2.7, 3.9, 1.4]    --> It can be: versicolor species.
Testing set: [5.9, 3.0, 4.2, 1.5]    --> It can be: versicolor species.
Testing set: [5.6, 2.5, 3.9, 1.1]    --> It can be: versicolor species.
Testing set: [6.1, 2.8, 4.0, 1.3]    --> It can be: versicolor species.
Testing set: [6.3, 2.5, 4.9, 1.5]    --> It can be: versicolor species.
Testing set: [6.4, 2.9, 4.3, 1.3]    --> It can be: versicolor species.
Testing set: [6.0, 2.9, 4.5, 1.5]    --> It can be: versicolor species.
Testing set: [6.0, 2.7, 5.1, 1.6]    --> It can be: versicolor species.
Testing set: [6.0, 3.4, 4.5, 1.6]    --> It can be: versicolor species.
Testing set: [5.6, 3.0, 4.1, 1.3]    --> It can be: versicolor species.
Testing set: [5.0, 2.3, 3.3, 1.0]    --> It can be: versicolor species.
Testing set: [5.7, 2.9, 4.2, 1.3]    --> It can be: versicolor species.
Testing set: [5.7, 2.8, 4.1, 1.3]    --> It can be: versicolor species.
Testing set: [6.3, 3.3, 6.0, 2.5]    --> It can be: virginica species.
Testing set: [7.1, 3.0, 5.9, 2.1]    --> It can be: virginica species.
Testing set: [6.5, 3.0, 5.8, 2.2]    --> It can be: virginica species.
Testing set: [7.6, 3.0, 6.6, 2.1]    --> It can be: virginica species.
Testing set: [7.3, 2.9, 6.3, 1.8]    --> It can be: virginica species.
Testing set: [5.7, 2.5, 5.0, 2.0]    --> It can be: virginica species.
Testing set: [6.3, 2.7, 4.9, 1.8]    --> It can be: virginica species.
Testing set: [7.2, 3.2, 6.0, 1.8]    --> It can be: virginica species.
Testing set: [6.2, 2.8, 4.8, 1.8]    --> It can be: virginica species.
Testing set: [6.4, 2.8, 5.6, 2.1]    --> It can be: virginica species.
Testing set: [7.4, 2.8, 6.1, 1.9]    --> It can be: virginica species.
Testing set: [7.9, 3.8, 6.4, 2.0]    --> It can be: virginica species.
Testing set: [6.4, 2.8, 5.6, 2.2]    --> It can be: virginica species.
Testing set: [6.3, 2.8, 5.1, 1.5]    --> It can be: versicolor species.
Testing set: [6.9, 3.1, 5.4, 2.1]    --> It can be: virginica species.
Testing set: [6.9, 3.1, 5.1, 2.3]    --> It can be: virginica species.
Testing set: [5.8, 2.7, 5.1, 1.9]    --> It can be: virginica species.
Testing set: [6.7, 3.3, 5.7, 2.5]    --> It can be: virginica species.
Testing set: [5.9, 3.0, 5.1, 1.8]    --> It can be: virginica speci
Accuracy: 0.980
```

THE END