Centro Universitário FEI

***Perceptron***

**Aula 6 - Exercícios**

Disciplina:

Tópicos Especiais em Aprendizagem (Prof. Reinaldo A. C. Bianchi)

Aluno:

Claudio Aparecido Borges Junior (RA 120122-7)

São Bernardo do Campo

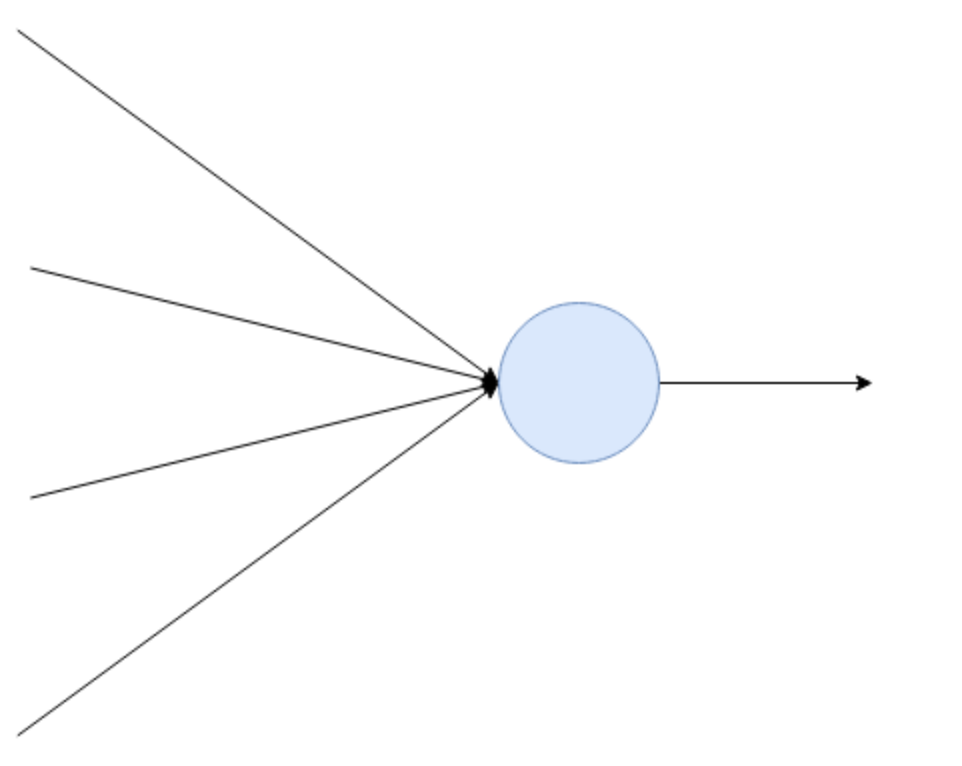
2o semestre / 2020

**EXERCÍCIO 1:** *Implemente o Perceptron para classificar o dataset Iris, separando as classes “setosa” e “versicolor”.*

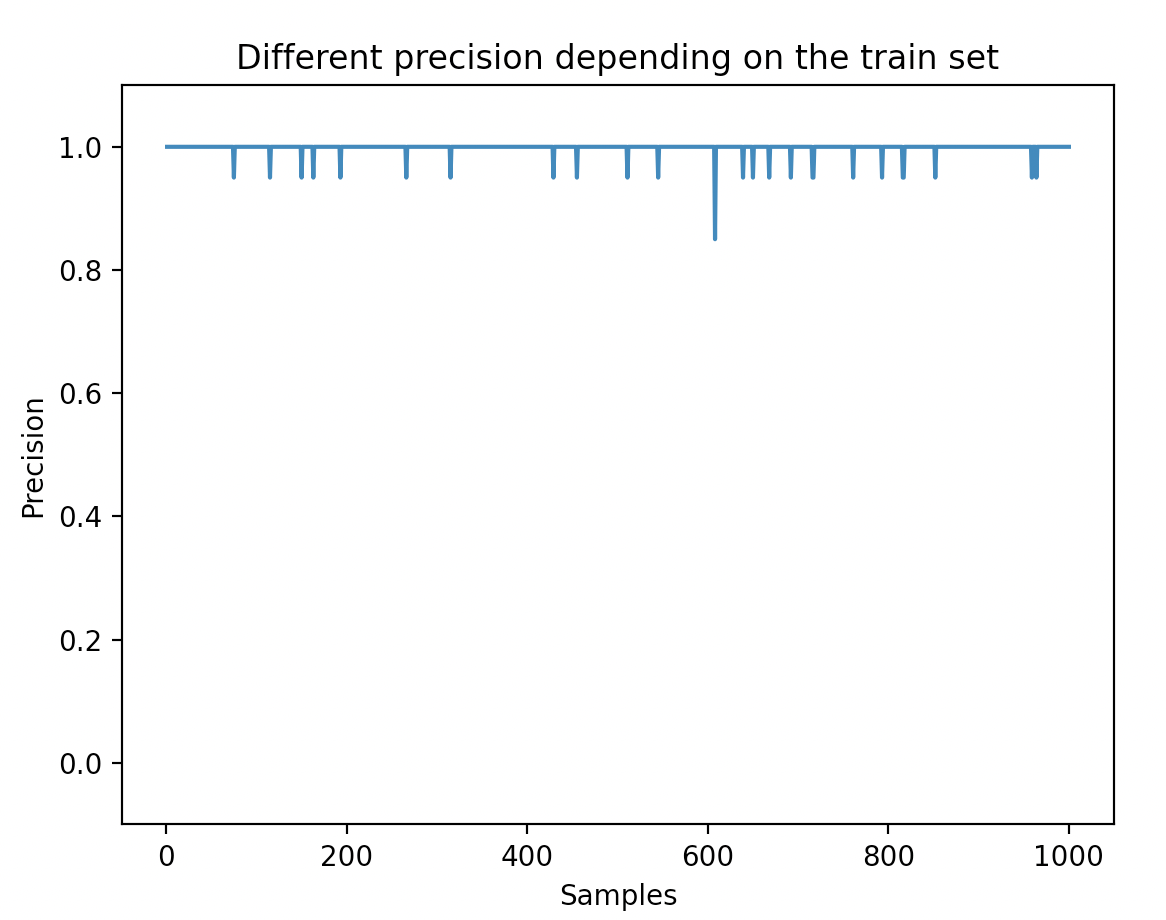
A implantação foi realizada utilizando a linguagem Python versão 3.6, com função de entrada ex1\_iris\_2\_classes() presente no Anexo 1. A classe Perceptron foi criada para encapsular as ações necessárias para treinar e classificar utilizando neurônios perceptron. Foi criada a classe Neurônio dentro da classe Perceptron para facilitar os casos de existência de mais de uma classe de saída, assim cada neurônio é responsável por classificar apenas 1 classe.

Dois métodos são utilizados, o train e o predict. O primeiro treina o modelo utilizando um conjunto de N entradas de M atributos cada e uma lista de N classificações de K classes. O segundo prediz para cada neurônio K se ele está ativo ou não, ou seja, se a entrada é da classe pertencente por ele.

Como o conjunto de dados da Iris possui 4 variáveis de entrada, o neurônio possuirá 4 entradas, além do viés.

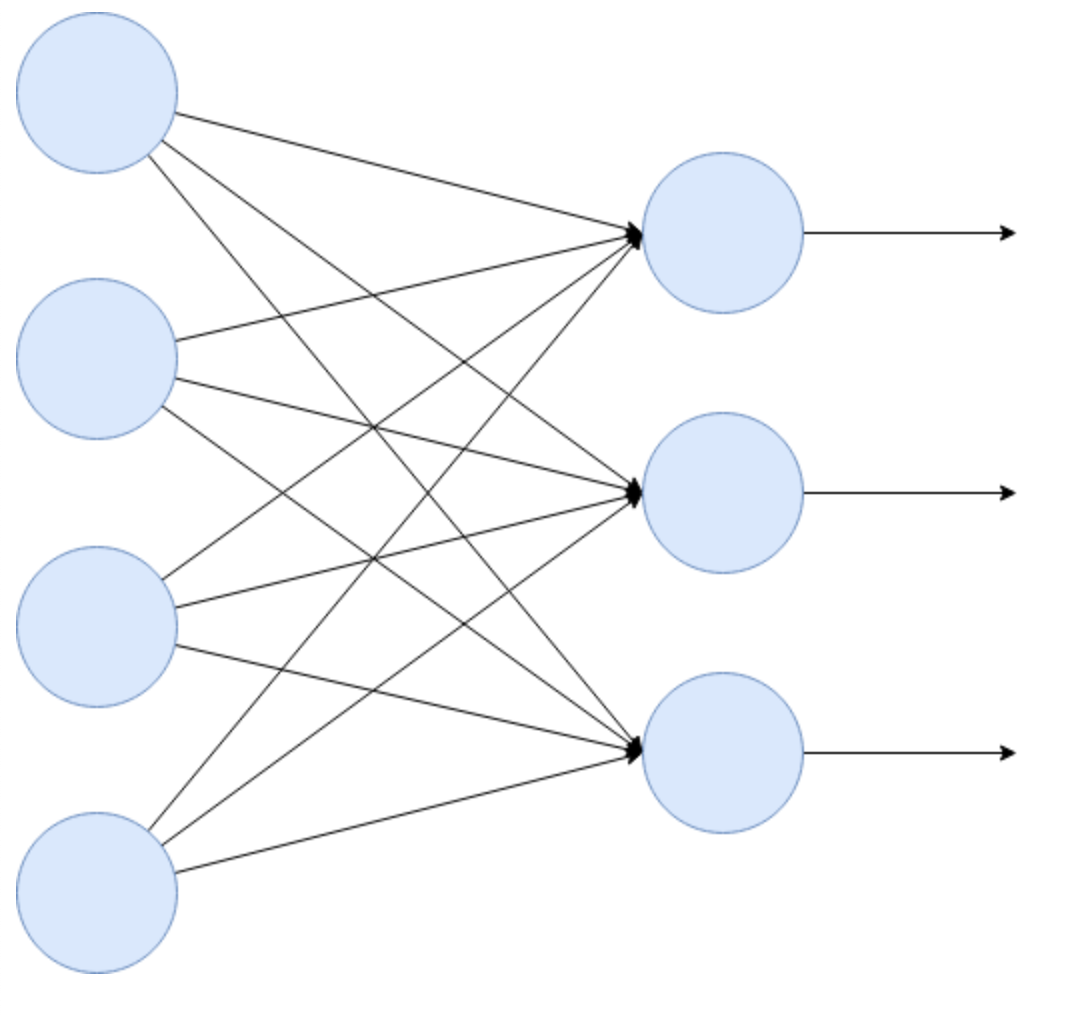


Abaixo é apresentado a resultado da precisão randomizando os dados de entrada para 1000 amostras. O menor valor ocorrido foi de 0.85 e o maior valor foi 1.0. A taxa de aprendizado utilizada foi de 1e-4 e a tolerância para convergência de 1e-8.



**EXERCÍCIO 2:** *Implemente o Perceptron Multiclasses euse no datasert Iris, separando as 3 classes (é possível?).*

A implantação foi realizada utilizando a linguagem Python versão 3.6, com função de entrada ex1\_iris\_3\_classes() presente no Anexo 1. Assim como no exercício anterior, duas classes foram utilizadas a Perceptron e a Perceptron.Neuron e o treinamento foi realizado por neurônio. Nesse caso, cada neurônio classifica uma classe distinta do conjunto de dados da Iris conforme figura abaixo:

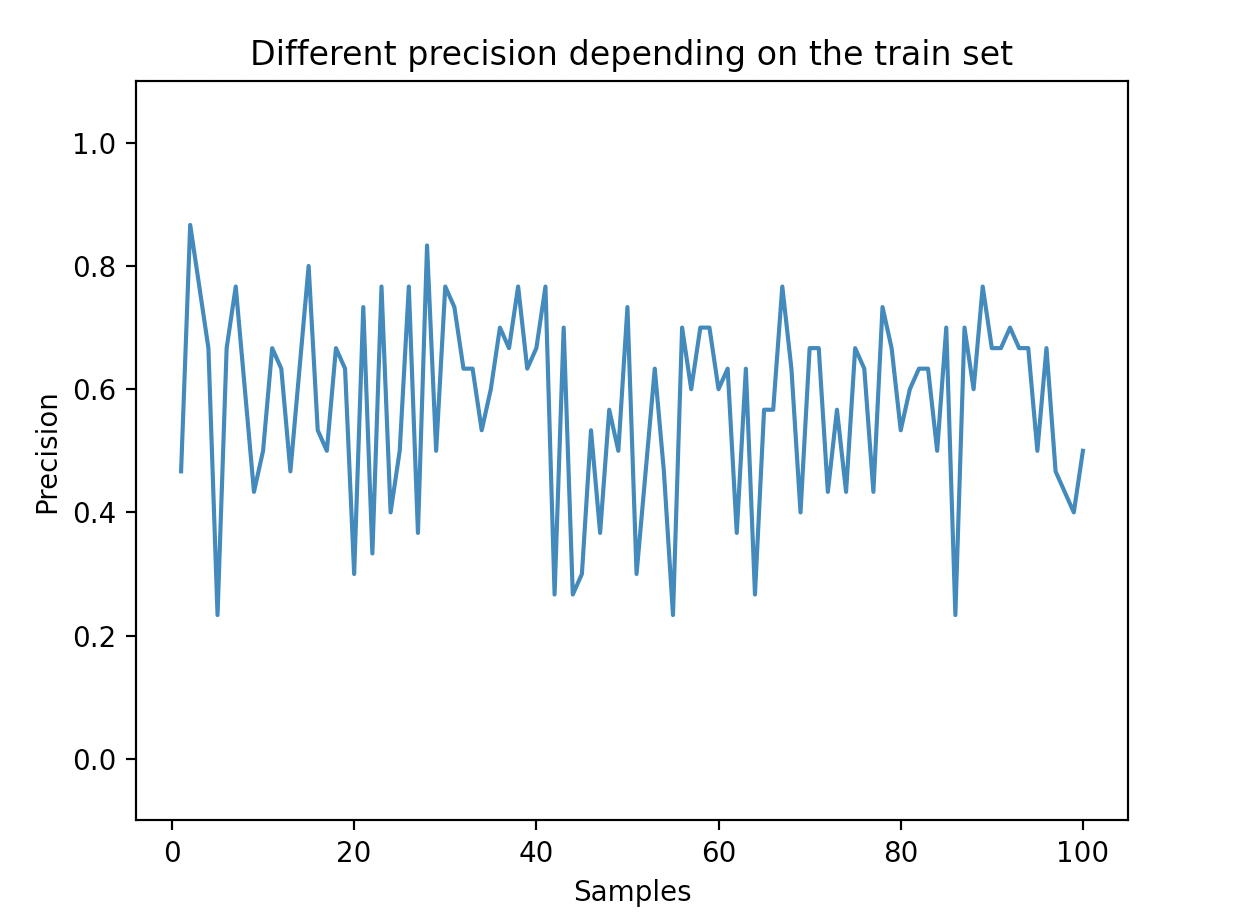


Um efeito que acontece é o de que em alguns momentos nenhum ou mais do que um neurônio pode estar ativo, impossibilitando a classificação. Em uma execução realizada foi encontrada a seguinte métrica de ativação:

Activations for the same input: Counter({1: 23, 2: 4, 0: 3})

Indicando que em 30 casos 2 neurônios estavam ativos ao mesmo tempo e nenhum neurônio não ficou ativo em 3 casos.

Abaixo é apresentado a resultado da precisão randomizando os dados de entrada para 100 amostras. Foi considerado divergente quando mais do que um ou nenhum neurônio se ativaram. O menor valor ocorrido foi de 0.23 e o maior valor foi 0.87. A taxa de aprendizado utilizada foi de 1e-4 e a tolerância para convergência de 1e-4.



**ANEXO I – CÓDIGO FONTE**

import numpy as np  
import copy  
from collections import Counter  
import matplotlib.pyplot as plt  
  
  
class Perceptron():  
 *""" Implements a single layer perceptron.  
 Perceptron is a binaty classifier that maps an input $X\_i$ to a value  
 $f(x)$ using he function:  
 f(x) = { 1 if $w$ x $x$ + b > 0  
 { -1 otherwise  
 being:  
 $w$: weight vector  
 $x$: input vector  
 $b$: bias  
 """* class Neuron():  
 *""" Represents a perceptron neuron.  
  
 This neuron have X inputs and 1 output (Y). It have W weights with the  
 same dimension as the X.  
 """* def \_\_init\_\_(self, dim, learning\_rate, tolerance):  
 *""" Initialize the Perceptron neuron with a dimension.* ***:param*** *dim: Dimension of the weights. Must be the same number  
 of the input number and bias, if applicable* ***:param*** *learn\_rate: Learn rate for the neuron  
 """* # Initialize the weights (w) with a random value between 0 and 1  
 self.w = np.random.rand(dim)  
 self.learning\_rate = learning\_rate  
 self.tolerance = tolerance  
  
 def \_train\_step(self, x, t):  
 *""" Make a train step  
 A train step is done when presenting a single input (with all  
 dimensions) and a single expected output. The process will  
 eventually modify the weigths matrix* ***:param*** *x: input* ***:param*** *t: target  
 """* # Output of the perceptron  
 y = self.predict(x)  
 # Error of the perceptron  
 error = t - y  
 # Delta step based  
 delta = self.learning\_rate \* np.array([xi \* error for xi in x])  
 # The new value of the weight  
 self.w = self.w + delta  
  
 def train(self, xs, ys):  
 *""" Train the neuron until it converge  
 """* # Add a convergence tries threshold  
 i = 0  
 max\_steps = 1e4  
 w\_diff = self.tolerance + 1  
 # Keep training until not converged  
 while (i < max\_steps) and (w\_diff > self.tolerance):  
 w\_old = copy.deepcopy(self.w)  
 for x, y in zip(xs, ys):  
 self.\_train\_step(x, y)  
 # Calculate the diff between the old weights and the new ones  
 # by taking the maximum absolute  
 w\_diff = np.max(np.abs(self.w - w\_old))  
 i += 1  
 if w\_diff > self.tolerance:  
 # If it stopped because it did not converge, print a message  
 print('Train timeout')  
  
 def predict(self, x):  
 *""" Predict an output* ***:param*** *x: input* ***:return*** *the prediction as -1 or 1  
 """* y = sum(x \* self.w.transpose())  
 return 1 if y > 0 else -1  
  
 def \_\_init\_\_(self, tolerance=1e-6, learning\_rate=1e-4):  
 *""" Initialize the perceptron* ***:param*** *tolerance: minimum tolerance for training* ***:param*** *learning\_rate: convergence rate  
 """* self.tolerance = tolerance  
 self.learning\_rate = learning\_rate  
 # There will be 1 neuron for each possible output. If only 2 classes  
 # are needed, 2 neurons will be created and the application can  
 # choose whatever neuron it prefer (however, nothing garantees the  
 # precision are the same).  
 self.neurons = None  
  
 def train(self, xs, ys):  
 # Each possible output will be a unique neuron, if the neuron is active  
 # means that it has predicted the output associated with it  
  
 # Add a singular column for the bias  
 xs\_bias = np.c\_[np.ones(len(xs)), xs]  
 # The dimension considers the first element as bias  
 dim = len(xs\_bias[0])  
 # Identify unique classes  
 classes = set(ys[:, 0])  
 # Create one neuron for each class (because of binary classification)  
 self.neurons = {  
 cls: Perceptron.Neuron(dim, self.learning\_rate, self.tolerance)  
 for cls in classes}  
  
 # Train each neuron apart  
 for cls, neuron in self.neurons.items():  
 # Convert each output to real values (-1 or 1)  
 y\_real = [1 if y == cls else -1 for y in ys[:, 0]]  
 # Train the neuron for each sample  
 neuron.train(xs\_bias, y\_real)  
  
 def predict(self, xs):  
 # Add a singular column for the bias  
 xs\_bias = np.c\_[np.ones(len(xs)), xs]  
 # Predict the output for each neuron and return a list of the  
 # predictions and the neurons  
 return {cls: [True if neuron.predict(x) > 0 else False for x in xs\_bias]  
 for cls, neuron in self.neurons.items()}  
  
 @staticmethod  
 def split\_train\_test(x, y, train\_size=0.8):  
 *""" Split the input and the output into train and test datasets* ***:param*** *x: dataset inputs* ***:param*** *y: dataset output* ***:param*** *train\_size: ranging from 0 to 1* ***:return****: the train and test x and y's  
 """* cutoff\_index = int(len(x) \* train\_size)  
 x\_train = x[:cutoff\_index, :]  
 y\_train = y[:cutoff\_index, :]  
 x\_test = x[cutoff\_index:, :]  
 y\_test = y[cutoff\_index:, :]  
 return x\_train, y\_train, x\_test, y\_test,  
  
  
def calc\_precision(real, prediction):  
 *""" Calculate how good the prediction is base of the precision.* ***:param*** *real: real values* ***:param*** *prediction: predicted values* ***:return****: quality measure ranging from [0 - 1] (inclusive)  
 """* err = sum(  
 [1 for a, b in zip(real, prediction) if a == b])  
 return err / len(prediction)  
  
  
def ex1\_iris\_2\_classes():  
 # Read the iris dataset  
 data = np.genfromtxt('datasets/iris.data', delimiter=',',  
 dtype=None,  
 encoding=None)  
  
 # Filter out non setosa and versicolor as per exercise requirement  
 sub\_data = [[elm for elm in row] for row in data if  
 row[-1] in ('Iris-setosa', 'Iris-versicolor')]  
  
 # Suffle the data to decrease the training bias  
 np.random.shuffle(sub\_data)  
 # Split the into X an Y where X are the input samples and Y are the  
 # output samples (what we want to classify)  
 x = np.array(  
 [[elm for i, elm in enumerate(row) if i < len(row) - 1] for row in  
 sub\_data])  
 y = np.array(  
 [[elm for i, elm in enumerate(row) if i == len(row) - 1] for row in  
 sub\_data])  
  
 # Split the dataset into training and test  
 x\_train, y\_train, x\_test, y\_test, = Perceptron.split\_train\_test(x, y)  
  
 perceptron = Perceptron(tolerance=1e-3, learning\_rate=1e-4)  
 perceptron.train(x\_train, y\_train)  
 y\_classified = perceptron.predict(x\_test)  
  
 # Adjust classification because 2 neurons were used and we need just one  
 y\_classified\_adj = ['Iris-setosa' if i else 'Iris-versicolor' for i in  
 y\_classified['Iris-setosa']]  
  
 # Calculated the precision of the classified output  
 precision = calc\_precision(y\_test.transpose()[0], y\_classified\_adj)  
 print("Precision ex1: ", precision)  
 precisions.append(precision)  
  
  
def exi\_iris\_3\_classes():  
 # Read the iris dataset  
 data = np.genfromtxt('datasets/iris.data', delimiter=',',  
 dtype=None,  
 encoding=None)  
  
 # Suffle the data to decrease the training bias  
 np.random.shuffle(data)  
 # Split the into X an Y where X are the input samples and Y are the  
 # output samples (what we want to classify)  
 x = np.array(  
 [[elm for i, elm in enumerate(row) if i < len(row) - 1] for row in  
 data])  
 y = np.array(  
 [[elm for i, elm in enumerate(row) if i == len(row) - 1] for row in  
 data])  
  
 # Split the dataset into training and test  
 x\_train, y\_train, x\_test, y\_test, = Perceptron.split\_train\_test(x, y)  
  
 perceptron = Perceptron(tolerance=1e-3, learning\_rate=1e-4)  
 perceptron.train(x\_train, y\_train)  
 y\_classified = perceptron.predict(x\_test)  
  
 # Verify if more than one neuron activates at the same time  
 setosa = y\_classified['Iris-setosa']  
 virginica = y\_classified['Iris-virginica']  
 versicolor = y\_classified['Iris-versicolor']  
 conjunction = [x + y + z for x, y, z in zip(setosa, virginica, versicolor)]  
 counter = Counter(conjunction)  
 print('Activations for the same input: ', counter)  
 # Calculate global precision  
 global\_result = []  
 for i in range(len(y\_test.transpose()[0])):  
 if conjunction[i] != 1:  
 global\_result.append('')  
 elif setosa[i]:  
 global\_result.append('Iris-setosa')  
 elif virginica[i]:  
 global\_result.append('Iris-virginica')  
 else:  
 global\_result.append('Iris-versicolor')  
 precision = calc\_precision(y\_test.transpose()[0], global\_result)  
 print("Global precision: ", precision)  
  
 # Calculate the precision for each neuron  
 for cls, results in y\_classified.items():  
 # Adjust the expected value to be true if one class and false if  
 # another one  
 y\_test\_adj = [True if y == cls else False for y in  
 y\_test.transpose()[0]]  
 # Calculated the precision of the classified output  
 precision = calc\_precision(y\_test\_adj, results)  
 print("Precision of ", cls, ': ', precision)  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 ex1\_iris\_2\_classes()  
 exi\_iris\_3\_classes()