```
from __future__ import division
from random import seed, uniform
from math import exp
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn import linear_model

def getData(file):
    data = np.array(pd.read_csv(file).dropna())
    classStr = data[:,-1]
    classMap = {'Iris-setosa': 0, 'Iris-versicolor': 1, 'Iris-virginica': 2}
    classNum = np.zeros(len(classStr))
    for i in range(len(classStr)):
        classNum[i] = classMap[classStr[i]]
        data[i,-1] = int(classNum[i])
    dataTrain, dataTest, classTrain, classTest = train_test_split(data, data[:, -1], test_size=0.3, random_state=3)
    return data, dataTrain, dataTest

# Calculate neuron activation as the sum of the weighted inputs to the neuron, i.e.
#     activation = weight[1] * input[1] + ... + weight[n-1] * input[n-1] + weight[n]
# where weight[n] = bias
def calculateActivation(weights, inputs):
    activation = 0
    for i in range(len(weights)):
        if i != len(weights)-1:
            activation += weights[i] * inputs[i]
        else:
            activation += weights[i]
    return activation

# Calculate neuron output as the sigmoid function of the neuron activation, i.e.
#     output = 1 / (1 + exp(-activation))
def calculateOutput(activation):
    output = 1.0 / (1.0 + exp(-activation))
    return output

# Calculate derivative of the neuron output as the derivative of the sigmoid function, i.e.
#     derivative = output * (1 - output)
def calculateDerivative(output):
    derivative = output * (1.0 - output)
    return derivative

# Forward propagate the outputs of each layer to the inputs to the next layer. That is,
# for each layer in the network, calculate the activation to and output of each neuron.
# Then forward the outputs to the inputs of the next layer.
def forwardPropagate(network, inputs):
    for layer in network:
        outputs = [0] * len(layer)
        for i, neuron in enumerate(layer):
            activation = calculateActivation(neuron['weights'], inputs)
            neuron['output'] = calculateOutput(activation)
            outputs[i] = neuron['output']
        inputs = outputs
```

```
    return outputs

# Backward propagate the errors from each layer to train the weights of the previous layer.
# That is, calculate the error for each neuron in each layer, starting with the output layer
# and proceeding backwards through the hidden layers. For the output layer, the error is
# is the difference between the correct value and the output value. For the hidden layers,
# the error is the product of the error for the next layer and the derivative of the output,
# multiplied by the weight of the connection between the neuron in the current layer and the
# next layer.
def backwardPropagate(network, correct):
    for i in reversed(range(len(network))):
        layer = network[i]
        errors = [0] * len(layer)
        if i == len(network)-1:
            for j in range(len(layer)):
                neuron = layer[j]
                errors[j] = correct[j] - neuron['output']
        else:
            for j in range(len(layer)):
                error = 0.0
                for neuron in network[i + 1]:
                    error += neuron['weights'][j] * neuron['delta']
                errors[j] = error
        for j in range(len(layer)):
            neuron = layer[j]
            derivative = calculateDerivative(neuron['output'])
            neuron['delta'] = errors[j] * derivative

def updateWeights(network, observation, eta):
    for i in range(len(network)):
        if i == 0:
            inputs = observation[:i-1]
        else:
            inputs = [neuron['output'] for neuron in network[i-1]]
        for neuron in network[i]:
            for j in range(len(inputs)+1):
                if j < len(inputs):
                    neuron['weights'][j] += eta * neuron['delta'] * inputs[j]
                else:
                    neuron['weights'][j] += eta * neuron['delta']

def createNetwork(numInput, numHidden, numOutput):
    hiddenLayer = [{'weights':[uniform(0, 1) for i in range(numInput + 1)], 'delta':0} for j in range(numHidden)]
    outputLayer = [{'weights':[uniform(0, 1) for i in range(numHidden + 1)], 'delta':0} for j in range(numOutput)]
    network = [hiddenLayer, outputLayer]
    return network

def trainNetwork(network, train, eta, numEpoch, numOutput):
    for epoch in range(numEpoch):
        totalError = 0
        for observation in train:
            correct = [0 for i in range(numOutput)]

            # Translate class values from integer representation {0, 1, 2, ...}
            # to one-hot representation {[1 0 0 ...], [0 1 0 ...], [0 0 1 ...], ...}
```

```
            # by setting the i-th digit of the one-hot representation to 1,
            # where i is the value of the integer representation,
            # and setting all other bits of the one-hot representation to 0
            correct[observation[-1]] = 1
            output = forwardPropagate(network, observation)
            for i in range(len(correct)):
                totalError += (correct[i]-output[i])**2
            backwardPropagate(network, correct)
            updateWeights(network, observation, eta)
        print "Epoch:", epoch, "Total error:", totalError

def getEstimate(network, observation):
    outputs = forwardPropagate(network, observation)
    return outputs.index(max(outputs))

print "Begin Neural Network"

# Load and pre-process data, i.e. map class label strings to integers, and split data into training and testing sets
data, dataTrain, dataTest = getData('iris.csv')
print "dataTrain[0:20, :]:\n", dataTrain[0:20, :]
print "dataTest[0:20, :]:\n", dataTest[0:20, :]

# Train neural network
data = dataTrain
numInput = len(data[0]) - 1
numHidden = 20
numOutput = len(set([observation[-1] for observation in data]))
seed(3)
network = createNetwork(numInput, numHidden, numOutput)
print "numInput:", numInput
print "numOutput:", numOutput

eta = 0.25
numEpoch = 200
trainNetwork(network, data, eta, numEpoch, numOutput)

for layer in network:
    print layer

# Test neural network
data = dataTest
numObservations = len(data[:,0])
numErrors = 0
for observation in data:
    estimate = getEstimate(network, observation)
    if estimate != observation[-1]: numErrors += 1
    print "Correct value:", observation[-1], "Estimated value:", estimate
errorRate = numErrors / numObservations
accuracy = 1 - errorRate
print "Accuracy:", accuracy

print "End Neural Network"
```