```
import numpy as np

############################################################
# Modeling: what we want to compute

#points = [(np.array([2]), 4), (np.array([4]), 2)]
#d = 1

# Generate artificial data to test whether the learning algorithm is working.
# Learning goes from data to parameters, but we can start with parameters
# (`true_w`) and generate data that will be fit well by these parameters.
true_w = np.array([1, 2, 3, 4, 5])  # Hidden from learning algorithm
d = len(true_w)
points = []
for i in range(500000):
    x = np.random.randn(d)
    y = true_w.dot(x) + np.random.randn()
    #print(x, y)
    points.append((x, y))

# For gradient descent
def F(w):
    return sum((w.dot(x) - y)**2 for x, y in points) / len(points)

def dF(w):
    return sum(2*(w.dot(x) - y) * x for x, y in points) / len(points)

# For stochastic gradient descent
def sF(w, i):
    x, y = points[i]
    return (w.dot(x) - y)**2

def sdF(w, i):
    x, y = points[i]
    return 2*(w.dot(x) - y) * x

############################################################
# Algorithms: how we compute it

def gradientDescent(F, dF, d):
    w = np.zeros(d)
    eta = 0.01
    for t in range(1000):
        value = F(w)
        gradient = dF(w)
        w = w - eta * gradient
        print('iteration {}: w = {}, F(w) = {}'.format(t, w, value))

def stochasticGradientDescent(sF, sdF, d, n):
    # Gradient descent
    w = np.zeros(d)
    eta = 1
    numUpdates = 0
    for t in range(1000):
        for i in range(n):  # For each data point...
            value = sF(w, i)
            gradient = sdF(w, i)
            numUpdates += 1
            eta = 1.0 / numUpdates  # Remember to do 1.0 instead of 1!
```

```
            w = w - eta * gradient
        print('iteration {}: w = {}, F(w) = {}'.format(t, w, value))

#gradientDescent(F, dF, d)
stochasticGradientDescent(sF, sdF, d, len(points))
```