def gradient\_descent(X, y, theta, alpha, num\_iters):

"""

Learn the parameters of the model using gradient descent using

the \*training set\*. Gradient descent is an optimization algorithm

used to minimize some (loss) function by iteratively moving in

the direction of steepest descent as defined by the negative of

the gradient. We use gradient descent to update the parameters

(weights) of our model.

Input:

- X: Inputs (n features over m instances).

- y: True labels (1 value over m instances).

- theta: The parameters (weights) of the model being learned.

- alpha: The learning rate of your model.

- num\_iters: The number of updates performed.

Returns two values:

- theta: The learned parameters of your model.

- J\_history: the loss value for every iteration.

"""

J\_history = [] # Use a python list to save cost in every iteration

theta = theta.copy() # avoid changing the original thetas

###########################################################################

# TODO: Implement the gradient descent optimization algorithm. #

###########################################################################

for k in range(num\_iters):

J\_history.append(compute\_cost(X, y, theta))

temp\_theta = np.array([])

for i in range(len(theta)):

theta\_i = 0

for j, vec in enumerate(X):

theta\_j = 0

theta\_j = np.inner(vec, theta)

theta\_j -= y[j]

theta\_j = theta\_j \* vec[i]

theta\_i += theta\_j

theta\_i = theta\_i \* (alpha / len(y))

temp\_theta = np.append(temp\_theta, theta\_i)

theta -= temp\_theta

###########################################################################

# END OF YOUR CODE #

###########################################################################

return theta, J\_history

**efficient**

def efficient\_gradient\_descent(X, y, theta, alpha, num\_iters):

"""

Learn the parameters of your model using the \*training set\*, but stop

the learning process once the improvement of the loss value is smaller

than 1e-8. This function is very similar to the gradient descent

function you already implemented.

Input:

- X: Inputs (n features over m instances).

- y: True labels (1 value over m instances).

- theta: The parameters (weights) of the model being learned.

- alpha: The learning rate of your model.

- num\_iters: The number of updates performed.

Returns two values:

- theta: The learned parameters of your model.

- J\_history: the loss value for every iteration.

"""

J\_history = [] # Use a python list to save cost in every iteration

theta = theta.copy() # avoid changing the original thetas

###########################################################################

# TODO: Implement the gradient descent optimization algorithm. #

###########################################################################

k = 0

curr\_cost = compute\_cost(X, y, theta)

J\_history.append(0)

while (abs(curr\_cost - J\_history[len(J\_history) - 1]) > 1E-8) and (k < num\_iters):

if k == 0:

J\_history.pop()

k += 1

J\_history.append(curr\_cost)

temp\_theta = np.array([])

for i in range(len(theta)):

theta\_i = 0

for j, vec in enumerate(X):

theta\_j = 0

theta\_j = np.inner(vec, theta)

theta\_j -= y[j]

theta\_j = theta\_j \* vec[i]

theta\_i += theta\_j

theta\_i = theta\_i \* (alpha / len(y))

temp\_theta = np.append(temp\_theta, theta\_i)

theta -= temp\_theta

curr\_cost = compute\_cost(X, y, theta)

###########################################################################

# END OF YOUR CODE #

###########################################################################

return theta, J\_history