Supervised Machine Learning Project

Welcome to my First Machine Learning Project by a Lab! In this lab, we will implement linear regression with one variable to predict profits for a restaurant franchise.

Outline

NOTE: To prevent any type of errors, you are not allowed to edit or delete non-graded cells in this notebook .

1 - Packages

First, let's run the cell below to import all the packages that we will need during this lab project.

- [numpy (www.numpy.org)](www.numpy.org) is the fundamental package for working with matrices in Python.
- [matplotlib (http://matplotlib.org)](http://matplotlib.org) is a famous library to plot graphs in Python.
- `utils.py` contains helper functions for this assignment. You do not need to modify code in this file.

In [ ]:

```python
import numpy as np
import matplotlib.pyplot as plt
from utils import *
import copy
import math
%matplotlib inline
```

2 - Problem Statement

Suppose you are the CEO of a restaurant franchise and are considering different cities for opening a new outlet.

- You would like to expand your business to cities that may give your restaurant higher profits.
- The chain already has restaurants in various cities and you have data for profits and populations from the cities.
- You also have data on cities that are candidates for a new restaurant.
    - For these cities, you have the city population.

We will use the data to help us identify which cities may potentially give your business higher profits.

3 - Dataset

We will start by loading the dataset for this task.

- The `load_data()` function shown below loads the data into variables `x_train` and `y_train`
    - `x_train` is the population of a city
    - `y_train` is the profit of a restaurant in that city. A negative value for profit indicates a loss.
    - Both `X_train` and `y_train` are numpy arrays.

In [ ]:

```python
# load the dataset
x_train, y_train = load_data()
```

View the variables

Before starting on any task, it is useful to get more familiar with our dataset.

- A good place to start is to just print out each variable and see what it contains.

The code below prints the variable `x_train` and the type of the variable.

In [ ]:

```python
# print x_train
print("Type of x_train:",type(x_train))
print("First five elements of x_train are:\n", x_train[:5])
```

`x_train` is a numpy array that contains decimal values that are all greater than zero.

- These values represent the city population times 10,000
- For example, 6.1101 means that the population for that city is 61,101

Now, let's print `y_train`

In [ ]:

```python
# print y_train
print("Type of y_train:",type(y_train))
print("First five elements of y_train are:\n", y_train[:5])
```

Similarly, `y_train` is a numpy array that has decimal values, some negative, some positive.

- These represent your restaurant's average monthly profits in each city, in units of $10,000.
  - For example, 17.592 represents $175,920 in average monthly profits for that city.
  - -2.6807 represents -$26,807 in average monthly loss for that city.

Check the dimensions of your variables

Another useful way to get familiar with your data is to view its dimensions.

Please print the shape of `x_train` and `y_train` and see how many training examples you have in your dataset.

In [ ]:

```python
print ('The shape of x_train is:', x_train.shape)
print ('The shape of y_train is: ', y_train.shape)
print ('Number of training examples (m):', len(x_train))
```

The city population array has 97 data points, and the monthly average profits also has 97 data points. These are NumPy 1D arrays.

Visualize your data

It is often useful to understand the data by visualizing it.

- For this dataset, we can use a scatter plot to visualize the data, since it has only two properties to plot (profit and population).
- Many other problems that we will encounter in real life have more than two properties (for example, population, average household income, monthly profits, monthly sales).When we have more than two properties, we can still use a scatter plot to see the relationship between each pair of properties.

In [ ]:

```python
# Create a scatter plot of the data. To change the markers to red "x",
# we used the 'marker' and 'c' parameters
plt.scatter(x_train, y_train, marker='x', c='r')

# Set the title
plt.title("Profits vs. Population per city")
# Set the y-axis label
plt.ylabel('Profit in $10,000')
# Set the x-axis label
plt.xlabel('Population of City in 10,000s')
plt.show()
```

Our goal is to build a linear regression model to fit this data.

- With this model, we can then input a new city's population, and have the model estimate your restaurant's potential monthly profits for that city.

4 - Refresher on linear regression

In this project, we will fit the linear regression parameters (w, b) to your dataset.

- The model function for linear regression, which is a function that maps from x (city population) to y (your restaurant's monthly profit for that city) is represented as

$$f_{w,b}(x) = wx + b$$

- To train a linear regression model, we want to find the best (w, b) parameters that fit your dataset.
    - To compare how one choice of (w, b) is better or worse than another choice, we can evaluate it with a cost function $J(w, b)$
        - J is a function of (w, b). That is, the value of the cost $J(w, b)$ depends on the value of (w, b).
    - The choice of (w, b) that fits your data the best is the one that has the smallest cost $J(w, b)$.

- To find the values (w, b) that gets the smallest possible cost $J(w, b)$, you can use a method called **gradient descent**.
    - With each step of gradient descent, your parameters (w, b) come closer to the optimal values that will achieve the lowest cost $J(w, b)$.

- The trained linear regression model can then take the input feature x (city population) and output a prediction $f_{w,b}(x)$ (predicted monthly profit for a restaurant in that city).

5 - Compute Cost

Gradient descent involves repeated steps to adjust the value of your parameter (w, b) to gradually get a smaller and smaller cost $J(w, b)$.

- At each step of gradient descent, it will be helpful for us to monitor our progress by computing the cost $J(w, b)$ as (w, b) gets updated.
- In this section, we will implement a function to calculate $J(w, b)$ so that we can check the progress of our gradient descent implementation.

Cost function

For one variable, the cost function for linear regression $J(w, b)$ is defined as

$$J(w, b) = \frac{1}{2m} \sum_{i=0}^{m-1} (f_{w,b}(x^{(i)}) - y^{(i)})^2$$

- We can think of $f_{w,b}(x^{(i)})$ as the model's prediction of your restaurant's profit, as opposed to $y^{(i)}$, which is the actual profit that is recorded in the data.
- m is the number of training examples in the dataset

Model prediction

- For linear regression with one variable, the prediction of the model $f_{w,b}$ for an example $x^{(i)}$ is respresentented as:

$$f_{w,b}(x^{(i)}) = wx^{(i)} + b$$

This is the equation for a line, with an intercept b and a slope w

Implementation

Look at the `compute_cost()` function below to compute the cost $J(w, b)$.

Exercise 1

Complete the `compute_cost` below to:

- Iterate over the training examples, and for each example, compute:
  - The prediction of the model for that example

$$f_{wb}(x^{(i)}) = wx^{(i)} + b$$

  - The cost for that example

$$cost^{(i)} = (f_{wb} - y^{(i)})^2$$

- Return the total cost over all examples

$$J(\mathbf{w}, b) = \frac{1}{2m} \sum_{i=0}^{m-1} cost^{(i)}$$

  - Here, m is the number of training examples and $\sum$ is the summation operator

If you are not able to understand the code, you can check out the hints presented after the cell below to help you with the implementation.

In [ ]:

```
# UNQ_C1
# GRADED FUNCTION: compute_cost

def compute_cost(x, y, w, b):
    """
    Computes the cost function for linear regression.

    Args:
        x (ndarray): Shape (m,) Input to the model (Population of cities)
        y (ndarray): Shape (m,) Label (Actual profits for the cities)
        w, b (scalar): Parameters of the model

    Returns
        total_cost (float): The cost of using w,b as the parameters for linear regression
               to fit the data points in x and y
    """
    # number of training examples
    m = x.shape[0]

    # We need to return this variable correctly
    total_cost = 0

    ### Here We Start ###

    for i in range(m):

        f_wb = w * x[i] + b

        cost = (f_wb - y[i])**2

        total_cost += cost

    total_cost = total_cost / (2 * m)

    return total_cost
```

▶

We will check if our implementation was correct by running the following test code:

In [ ]:

```
# Compute cost with some initial values for paramaters w, b
initial_w = 2
initial_b = 1

cost = compute_cost(x_train, y_train, initial_w, initial_b)
print(type(cost))
print(f'Cost at initial w: {cost:.3f}')

# Public tests
from public_tests import *
compute_cost_test(compute_cost)
```

Expected Output:

6 - Gradient descent

In this section, we will implement the gradient for parameters w, b for linear regression.

The gradient descent algorithm is:

$$\text{repeat until convergence: } \{$$
$$b := b - \alpha \frac{\partial J(w, b)}{\partial b}$$
$$w := w - \alpha \frac{\partial J(w, b)}{\partial w}$$
$$\}$$

where, parameters w, b are both updated simultaniously and where

$$\frac{\partial J(w, b)}{\partial b} = \frac{1}{m} \sum_{i=0}^{m-1} (f_{w,b}(x^{(i)}) - y^{(i)})$$

$$\frac{\partial J(w, b)}{\partial w} = \frac{1}{m} \sum_{i=0}^{m-1} (f_{w,b}(x^{(i)}) - y^{(i)})x^{(i)}$$

- m is the number of training examples in the dataset

- $f_{w,b}(x^{(i)})$ is the model's prediction, while $y^{(i)}$, is the target value

We will implement a function called `compute_gradient` which calculates $\frac{\partial J(w)}{\partial w}, \frac{\partial J(w)}{\partial b}$

Exercise 2

Please complete the `compute_gradient` function to:

- Iterate over the training examples, and for each example, compute:
    - The prediction of the model for that example

$$f_{wb}(x^{(i)}) = wx^{(i)} + b$$

    - The gradient for the parameters w, b from that example

$$\frac{\partial J(w, b)^{(i)}}{\partial b} = (f_{w,b}(x^{(i)}) - y^{(i)})$$

$$\frac{\partial J(w, b)^{(i)}}{\partial w} = (f_{w,b}(x^{(i)}) - y^{(i)})x^{(i)}$$

- Return the total gradient update from all the examples

$$\frac{\partial J(w, b)}{\partial b} = \frac{1}{m} \sum_{i=0}^{m-1} \frac{\partial J(w, b)^{(i)}}{\partial b}$$

$$\frac{\partial J(w, b)}{\partial w} = \frac{1}{m} \sum_{i=0}^{m-1} \frac{\partial J(w, b)^{(i)}}{\partial w}$$

    - Here, m is the number of training examples and $\sum$ is the summation operator

```python
# UNQ_C2
# GRADED FUNCTION: compute_gradient
def compute_gradient(x, y, w, b):
    """
    Computes the gradient for linear regression
    Args:
      x (ndarray): Shape (m,) Input to the model (Population of cities)
      y (ndarray): Shape (m,) Label (Actual profits for the cities)
      w, b (scalar): Parameters of the model
    Returns
      dj_dw (scalar): The gradient of the cost w.r.t. the parameters w
      dj_db (scalar): The gradient of the cost w.r.t. the parameter b
    """

    # Number of training examples
    m = x.shape[0]

    # We need to return the following variables correctly
    dj_dw = 0
    dj_db = 0

    ### HERE WE GO ###

    for i in range(m):

        f_wb = w * x[i] + b

        dj_dw += (f_wb - y[i]) * x[i]

        dj_db += (f_wb - y[i])


    dj_dw = dj_dw / m

    dj_db = dj_db / m

    return dj_dw, dj_db
```

▶

Let's Run the cells below to check our implementation of the `compute_gradient` function with two different initializations of the parameters w,b.

```python
# Compute and display gradient with w initialized to zeroes
initial_w = 0
initial_b = 0

tmp_dj_dw, tmp_dj_db = compute_gradient(x_train, y_train, initial_w, initial_b)
print('Gradient at initial w, b (zeros):', tmp_dj_dw, tmp_dj_db)

compute_gradient_test(compute_gradient)
```

Now let's run the gradient descent algorithm implemented above on our dataset.

Expected Output:

Gradient at initial , b (zeros)	-65.32884975 -5.83913505154639

```python
# Compute and display cost and gradient with non-zero w
test_w = 0.2
test_b = 0.2
tmp_dj_dw, tmp_dj_db = compute_gradient(x_train, y_train, test_w, test_b)

print('Gradient at test w, b:', tmp_dj_dw, tmp_dj_db)
```

Expected Output:

Gradient at test w	-47.41610118 -4.007175051546391

2.6 Learning parameters using batch gradient descent

We will now find the optimal parameters of a linear regression model by using batch gradient descent. Batch refers to running all the examples in one iteration.

- We don't need to implement anything for this part. Simply run the cells below.
- A good way to verify that gradient descent is working correctly is to look at the value of $J(w, b)$ and check that it is decreasing with each step.
- Assuming we have implemented the gradient and computed the cost correctly and we have an appropriate value for the learning rate alpha, $J(w, b)$ should never increase and should converge to a steady value by the end of the algorithm.

In [ ]:

```python
def gradient_descent(x, y, w_in, b_in, cost_function, gradient_function, alpha, num_iters):
    """
    Performs batch gradient descent to learn theta. Updates theta by taking
    num_iters gradient steps with learning rate alpha

    Args:
      x :      (ndarray): Shape (m,)
      y :      (ndarray): Shape (m,)
      w_in, b_in : (scalar) Initial values of parameters of the model
      cost_function: function to compute cost
      gradient_function: function to compute the gradient
      alpha : (float) Learning rate
      num_iters : (int) number of iterations to run gradient descent
    Returns
      w : (ndarray): Shape (1,) Updated values of parameters of the model after
          running gradient descent
      b : (scalar)                Updated value of parameter of the model after
          running gradient descent
    """

    # number of training examples
    m = len(x)

    # An array to store cost J and w's at each iteration — primarily for graphing later
    J_history = []
    w_history = []
    w = copy.deepcopy(w_in)  #avoid modifying global w within function
    b = b_in

    for i in range(num_iters):

        # Calculate the gradient and update the parameters
        dj_dw, dj_db = gradient_function(x, y, w, b )

        # Update Parameters using w, b, alpha and gradient
        w = w - alpha * dj_dw
        b = b - alpha * dj_db

        # Save cost J at each iteration
        if i<100000:      # prevent resource exhaustion
            cost =  cost_function(x, y, w, b)
            J_history.append(cost)

        # Print cost every at intervals 10 times or as many iterations if < 10
        if i% math.ceil(num_iters/10) == 0:
            w_history.append(w)
            print(f"Iteration {i:4}: Cost {float(J_history[-1]):8.2f}   ")

    return w, b, J_history, w_history #return w and J,w history for graphing
```

Now let's run the gradient descent algorithm above to learn the parameters for our dataset.

In [ ]:

```python
# initialize fitting parameters. Recall that the shape of w is (n,)
initial_w = 0.
initial_b = 0.

# some gradient descent settings
iterations = 1500
alpha = 0.01

w,b,_,_ = gradient_descent(x_train ,y_train, initial_w, initial_b,
                    compute_cost, compute_gradient, alpha, iterations)
print("w,b found by gradient descent:", w, b)
```

Expected Output:

<div align="center">

w, b found by gradient descent 1.16636235 -3.63029143940436

</div>

We will now use the final parameters from gradient descent to plot the linear fit.

The prediction for a single example $f(x^{(i)}) = wx^{(i)} + b$.

To calculate the predictions on the entire dataset, we can loop through all the training examples and calculate the prediction for each example. This is shown in the code block below.

In [ ]:

```python
m = x_train.shape[0]
predicted = np.zeros(m)

for i in range(m):
    predicted[i] = w * x_train[i] + b
```

We will now plot the predicted values to see the linear fit.

In [ ]:

```python
# Plot the linear fit
plt.plot(x_train, predicted, c = "b")

# Create a scatter plot of the data.
plt.scatter(x_train, y_train, marker='x', c='r')

# Set the title
plt.title("Profits vs. Population per city")
# Set the y-axis label
plt.ylabel('Profit in $10,000')
# Set the x-axis label
plt.xlabel('Population of City in 10,000s')
```

Your final values of w, b can also be used to make predictions on profits. Let's predict what the profit would be in areas of 35,000 and 70,000 people.

- The model takes in population of a city in 10,000s as input.
- Therefore, 35,000 people can be translated into an input to the model as `np.array([3.5])`
- Similarly, 70,000 people can be translated into an input to the model as `np.array([7.])`

In [ ]:

```python
predict1 = 3.5 * w + b
print('For population = 35,000, we predict a profit of $%.2f' % (predict1*10000))

predict2 = 7.0 * w + b
print('For population = 70,000, we predict a profit of $%.2f' % (predict2*10000))
```

Expected Output:

<div align="center">

For population = 35,000, we predict a profit of $4519.77

For population = 70,000, we predict a profit of $45342.45

</div>

**Thanks a lot for your precious time!

▶