Lab 4: Python, NumPy, Vectorization and Multiple Linear Regression

This lab consists of two parts:

1. Python, numpy, and vectorization
2. Multiple variable linear regression

Goals of Lab

In this lab, you will:

- Review the features of NumPy and Python that are used in Course

- Extend our regression model routines to support multiple features

Guidelines for Completing the Lab

In order to complete this lab, you need to run and understand the codes that are provided in this lab. Furthermore, the you need to complete following four tasks in this:

1. Task 1: Implement the dot product using for loop
2. Task 2: Implement the cost function for multiple variables
3. Task 3: Complete the function to compute gradients
4. Task 4: Complete the function to compute gradient descent for multiple variables

Part 1: Python, NumPy and Vectorization

A brief introduction to some of the scientific computing used in this course. In particular the NumPy scientific computing package and its use with python.

Outline

- Goals

- 1.2 Useful References

- 2 Python and NumPy

- 3 Vectors

-3.1 Abstract

- 3.2 NumPy Arrays

- 3.3 Vector Creation

- 3.4 Operations on Vectors

- 4 Matrices

- 4.1 Abstract

- 4.2 NumPy Arrays

- 4.3 Matrix Creation

- 4.4 Operations on Matrices

```
import numpy as np    # it is an unofficial standard to use np for numpy
import time
```

1.1 Goals

In this lab, you will:

- Review the features of NumPy and Python that are used in Course

1.2 Useful References

- NumPy Documentation including a basic introduction: [NumPy.org] (https://NumPy.org/doc/stable/)

- A challenging feature topic: [NumPy Broadcasting](https://NumPy.org/doc/stable/user/basics.broadcasting.html)

2 Python and NumPy

Python is the programming language we will be using in this course. It has a set of numeric data types and arithmetic operations. NumPy is a library that extends the base capabilities of python to add a richer data set including more numeric types, vectors, matrices, and many matrix functions. NumPy and python work together fairly seamlessly. Python arithmetic operators work on NumPy data types and many NumPy functions will accept python data types.

3 Vectors

3.1 Abstract

Vectors, as you will use them in this course, are ordered arrays of numbers. In notation, vectors are denoted with lower case bold letters such as **x**. The elements of a vector are all the same type. A vector does not, for example, contain both characters and numbers. The number of elements in the array is often referred to as the *dimension* though mathematicians may prefer *rank*. The vector shown has a dimension of n. The elements of a vector can be referenced with an index. In math settings, indexes typically run from 1 to n. In computer science and these labs, indexing will typically run from 0 to $n-1$. In notation, elements of a vector, when referenced individually will indicate the index in a subscript, for example, the 0-th element, of the vector **x** is x_0. Note, the x is not bold in this case.

3.2 NumPy Arrays

NumPy's basic data structure is an indexable, n-dimensional *array* containing elements of the same type (`dtype`). Right away, you may notice we have overloaded the term 'dimension'. Above, it was the number of elements in the vector, here, dimension refers to the number of indexes of an array. A one-dimensional or 1-D array has one index. In Course 1, we will represent vectors as NumPy 1-D arrays.

- 1-D array, shape $(n,)$: n elements indexed [0] through [n-1]

3.3 Vector Creation

Data creation routines in NumPy will generally have a first parameter which is the shape of the object. This can either be a single value for a 1-D result or a tuple (*n,m*,...) specifying the shape of the result. Below are examples of creating vectors using these routines.

```
# NumPy routines which allocate memory and fill arrays with value

a = np.zeros(4);                print(f"np.zeros(4) :   a = {a}, a shape
= {a.shape}, a data type = {a.dtype}")

a = np.zeros((4,));             print(f"np.zeros(4,) :  a = {a}, a shape
= {a.shape}, a data type = {a.dtype}")

a = np.random.random_sample(4); print(f"np.random.random_sample(4): a =
{a}, a shape = {a.shape}, a data type = {a.dtype}")
```

Some data creation routines do not take a shape tuple:

```
# NumPy routines which allocate memory and fill arrays with value but
do not accept shape as input argument

a = np.arange(4.);              print(f"np.arange(4.):     a = {a}, a
shape = {a.shape}, a data type = {a.dtype}")

a = np.random.rand(4);          print(f"np.random.rand(4): a = {a}, a
shape = {a.shape}, a data type = {a.dtype}")
```

values can be specified manually as well.

```
# NumPy routines which allocate memory and fill with user specified
values

a = np.array([5,4,3,2]);  print(f"np.array([5,4,3,2]):  a = {a},     a
shape = {a.shape}, a data type = {a.dtype}")

a = np.array([5.,4,3,2]); print(f"np.array([5.,4,3,2]): a = {a}, a shape
= {a.shape}, a data type = {a.dtype}")
```

These have all created a one-dimensional vector `a` with four elements. `a.shape` returns the dimensions. Here we see a.shape = `(4,)` indicating a 1-d array with 4 elements.

3.4 Operations on Vectors

Let's explore some operations using vectors.

3.4.1 Indexing

Elements of vectors can be accessed via indexing and slicing. NumPy provides a very complete set of indexing and slicing capabilities. We will explore only the basics needed for the course here. Reference [Slicing and Indexing](https://NumPy.org/doc/stable/reference/arrays.indexing.html) for more details.

Indexing means referring to *an element* of an array by its position within the array.

Slicing means getting a *subset* of elements from an array based on their indices.

NumPy starts indexing at zero so the 3rd element of an vector **a** is `a[2]`.

```
#vector indexing operations on 1-D vectors

a = np.arange(10)

print(a)


#access an element

print(f"a[2].shape: {a[2].shape} a[2]  = {a[2]}, Accessing an element returns a scalar")


# access the last element, negative indexes count from the end

print(f"a[-1] = {a[-1]}")


#indexs must be within the range of the vector or they will produce and error

try:

    c = a[10]
```

```
except Exception as e:
    print("The error message you'll see is:")
    print(e)
```

3.4.2 Slicing

Slicing creates an array of indices using a set of three values (`start:stop:step`). A subset of values is also valid. Its use is best explained by example:

```
#vector slicing operations
a = np.arange(10)
print(f"a        = {a}")

#access 5 consecutive elements (start:stop:step)
c = a[2:7:1];     print("a[2:7:1] = ", c)

# access 3 elements separated by two
c = a[2:7:2];     print("a[2:7:2] = ", c)

# access all elements index 3 and above
c = a[3:];        print("a[3:]    = ", c)

# access all elements below index 3
c = a[:3];        print("a[:3]    = ", c)

# access all elements
c = a[:];         print("a[:]     = ", c)
```

3.4.3 Single vector operations

There are a number of useful operations that involve operations on a single vector.

```
a = np.array([1,2,3,4])
print(f"a             : {a}")
# negate elements of a
b = -a
print(f"b = -a        : {b}")

# sum all elements of a, returns a scalar
b = np.sum(a)
print(f"b = np.sum(a) : {b}")

b = np.mean(a)
print(f"b = np.mean(a): {b}")

b = a**2
print(f"b = a**2      : {b}")
```

3.4.4 Vector Vector element-wise operations

Most of the NumPy arithmetic, logical and comparison operations apply to vectors as well. These operators work on an element-by-element basis. For example

$$\mathbf{a} + \mathbf{b} = \sum_{i=0}^{n-1} a_i + b_i$$

```
a = np.array([ 1, 2, 3, 4])
b = np.array([-1,-2, 3, 4])
print(f"Binary operators work element wise: {a + b}")
```

Of course, for this to work correctly, the vectors must be of the same size:

```
#try a mismatched vector operation
c = np.array([1, 2])
try:
    d = a + c
except Exception as e:
    print("The error message you'll see is:")
    print(e)
```

3.4.5 Scalar Vector operations

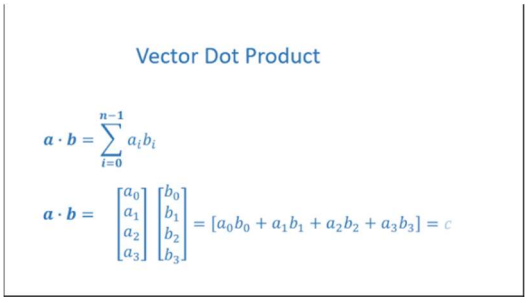
Vectors can be 'scaled' by scalar values. A scalar value is just a number. The scalar multiplies all the elements of the vector.

```
a = np.array([1, 2, 3, 4])

# multiply a by a scalar
b = 5 * a
print(f"b = 5 * a : {b}")
```

3.4.6 Vector Vector dot product

The dot product is a mainstay of Linear Algebra and NumPy. This is an operation used extensively in this course and should be well understood. The dot product is shown below.



The dot product multiplies the values in two vectors element-wise and then sums the result.

Vector dot product requires the dimensions of the two vectors to be the same.

Task 1: Implement the dot product using for loop

Using a for loop, implement a function which returns the dot product of two vectors. The function to return given inputs *a* and *b*:

$$x = \sum_{i=0}^{n-1} a_i b_i$$

Assume both `a` and `b` are the same shape.

Complete the following function.

```
def my_dot(a, b):
    """
   Compute the dot product of two vectors

    Args:
      a (ndarray (n,)):  input vector
      b (ndarray (n,)):  input vector with same dimension as a

    Returns:
      x (scalar):
    """
```

Note, the dot product is expected to return a scalar value.

Let's try the same operations using `np.dot`.

```
# test 1-D
a = np.array([1, 2, 3, 4])
b = np.array([-1, 4, 3, 2])
c = np.dot(a, b)
print(f"NumPy 1-D np.dot(a, b) = {c}, np.dot(a, b).shape = {c.shape} ")
c = np.dot(b, a)
print(f"NumPy 1-D np.dot(b, a) = {c}, np.dot(a, b).shape = {c.shape} ")
```

Above, you will note that the results for 1-D matched our implementation.

3.4.7 The Need for Speed: vector vs for loop

We utilized the NumPy library because it improves speed memory efficiency. Let's demonstrate:

```
np.random.seed(1)
a = np.random.rand(10000000)  # very large arrays
b = np.random.rand(10000000)

tic = time.time()  # capture start time
c = np.dot(a, b)
toc = time.time()  # capture end time

print(f"np.dot(a, b) =  {c:.4f}")
print(f"Vectorized version duration: {1000*(toc-tic):.4f} ms ")

tic = time.time()  # capture start time
```

```
c = my_dot(a,b)
toc = time.time()  # capture end time

print(f"my_dot(a, b) =  {c:.4f}")
print(f"loop version duration: {1000*(toc-tic):.4f} ms ")

del(a);del(b)  #remove these big arrays from memory
```

So, vectorization provides a large speed up in this example. This is because NumPy makes better use of available data parallelism in the underlying hardware. GPU's and modern CPU's implement Single Instruction, Multiple Data (SIMD) pipelines allowing multiple operations to be issued in parallel. This is critical in Machine Learning where the data sets are often very large.

3.4.8 Vector Vector operations in Course

Vector Vector operations will appear frequently in course. Here is why:

- Going forward, our examples will be stored in an array, `X_train` of dimension (m,n). This will be explained more in context, but here it is important to note it is a 2 Dimensional array or matrix (see next section on matrices).

- `w` will be a 1-dimensional vector of shape (n,).

- we will perform operations by looping through the examples, extracting each example to work on individually by indexing X. For example:`X[i]`

- `X[i]` returns a value of shape (n,), a 1-dimensional vector. Consequently, operations involving `X[i]` are often vector-vector.

That is a somewhat lengthy explanation, but aligning and understanding the shapes of your operands is important when performing vector operations.

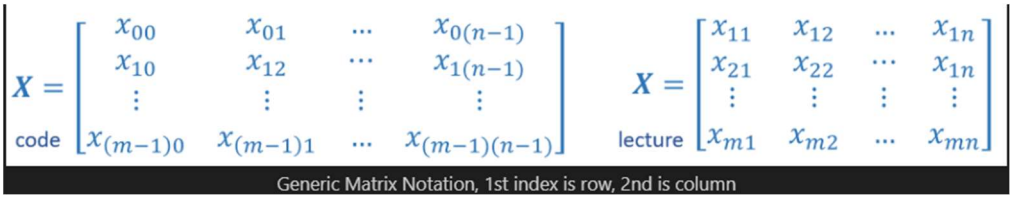
```
# show common Course example
X = np.array([[1],[2],[3],[4]])
w = np.array([2])
c = np.dot(X[1], w)
```

```
print(f"X[1] has shape {X[1].shape}")
print(f"w has shape {w.shape}")
print(f"c has shape {c.shape}")
```

4 Matrices

4.1 Abstract

Matrices, are two dimensional arrays. The elements of a matrix are all of the same type. In notation, matrices are denoted with capitol, bold letter such as **X**. In this and other labs, `m` is often the number of rows and `n` the number of columns. The elements of a matrix can be referenced with a two dimensional index. In math settings, numbers in the index typically run from 1 to n. In computer science and these labs, indexing will run from 0 to n-1.



Generic Matrix Notation, 1st index is row, 2nd is column

4.2 NumPy Arrays

NumPy's basic data structure is an indexable, n-dimensional *array* containing elements of the same type (`dtype`). These were described earlier. Matrices have a two-dimensional (2-D) index [m,n].

In Course 1, 2-D matrices are used to hold training data. Training data is m examples by n features creating an (m,n) array. Course 1 does not do operations directly on matrices but typically extracts an example as a vector and operates on that. Below you will review:

- data creation

- slicing and indexing

4.3 Matrix Creation

The same functions that created 1-D vectors will create 2-D or n-D arrays. Here are some examples

Below, the shape tuple is provided to achieve a 2-D result. Notice how NumPy uses brackets to denote each dimension. Notice further than NumPy, when printing, will print one row per line.

```
a = np.zeros((1, 5))
print(f"a shape = {a.shape}, a = {a}")

a = np.zeros((2, 1))
print(f"a shape = {a.shape}, a = {a}")

a = np.random.random_sample((1, 1))
print(f"a shape = {a.shape}, a = {a}")
```

One can also manually specify data. Dimensions are specified with additional brackets matching the format in the printing above.

```
# NumPy routines which allocate memory and fill with user specified values
a = np.array([[5], [4], [3]]);   print(f" a shape = {a.shape}, np.array: a = {a}")
a = np.array([[5],   # One can also
              [4],   # separate values
              [3]]); #into separate rows
print(f" a shape = {a.shape}, np.array: a = {a}")
```

4.4 Operations on Matrices

Let's explore some operations using matrices.

4.4.1 Indexing

Matrices include a second index. The two indexes describe [row, column]. Access can either return an element or a row/column. See below:

```
#vector indexing operations on matrices

a = np.arange(6).reshape(-1, 2)   #reshape is a convenient way to create matrices

print(f"a.shape: {a.shape}, \na= {a}")


#access an element

print(f"\na[2,0].shape:      {a[2,  0].shape},  a[2,0]  =  {a[2,  0]},  type(a[2,0]) = {type(a[2, 0])} Accessing an element returns a scalar\n")


#access a row

print(f"a[2].shape:    {a[2].shape}, a[2]    = {a[2]}, type(a[2])    = {type(a[2])}")
```

It is worth drawing attention to the last example. Accessing a matrix by just specifying the row will return a *1-D vector*.

Reshape

The previous example used [reshape](https://numpy.org/doc/stable/reference/generated/numpy.reshape.html) to shape the array.

`a = np.arange(6).reshape(-1, 2) `

This line of code first created a *1-D Vector* of six elements. It then reshaped that vector into a *2-D* array using the reshape command. This could have been written:

`a = np.arange(6).reshape(3, 2) `

To arrive at the same 3 row, 2 column array. The -1 argument tells the routine to compute the number of rows given the size of the array and the number of columns.

4.4.2 Slicing

Slicing creates an array of indices using a set of three values (`start:stop:step`). A subset of values is also valid. Its use is best explained by example:

```
#vector 2-D slicing operations
```

```
a = np.arange(20).reshape(-1, 10)

print(f"a = \n{a}")


#access 5 consecutive elements (start:stop:step)

print("a[0, 2:7:1] = ", a[0, 2:7:1], ",  a[0, 2:7:1].shape =", a[0,
2:7:1].shape, "a 1-D array")


#access 5 consecutive elements (start:stop:step) in two rows

print("a[:, 2:7:1] = \n", a[:, 2:7:1], ",  a[:, 2:7:1].shape =", a[:,
2:7:1].shape, "a 2-D array")


# access all elements

print("a[:,:] = \n", a[:,:], ",  a[:,:].shape =", a[:,:].shape)


# access all elements in one row (very common usage)

print("a[1,:] = ", a[1,:], ",  a[1,:].shape =", a[1,:].shape, "a 1-D
array")

# same as

print("a[1]   = ", a[1],   ",  a[1].shape   =", a[1].shape, "a 1-D
array")
```

Part 2: Multiple Variable Linear Regression

In this lab part, you will extend the data structures and previously developed routines to support multiple features. Several routines are updated making the lab appear lengthy, but it makes minor adjustments to previous routines making it quick to review.

Outline

1 Goals

- Extend our regression model routines to support multiple features
  - Extend data structures to support multiple features
  - Rewrite prediction, cost and gradient routines to support multiple features
  - Utilize NumPy `np.dot` to vectorize their implementations for speed and simplicity

1.2 Tools

In this lab, we will make use of:

- NumPy, a popular library for scientific computing

- Matplotlib, a popular library for plotting data

```
import copy, math
import numpy as np
import matplotlib.pyplot as plt
plt.style.use('deeplearning.mplstyle')
np.set_printoptions(precision=2)  # reduced display precision on numpy arrays
```

2 Problem Statement

You will use the motivating example of housing price prediction. The training dataset contains three examples with four features (size, bedrooms, floors and, age) shown in the table below. Note that, unlike the earlier labs, size is in sqft rather than 1000 sqft. This causes an issue, which you will solve in the next lab!

Size (sqft)	No. of Bedrooms	No. of floors	Age of Home	Price (1000s dollars)
2104	5	1	45	460
1416	3	2	40	232
852	2	1	35	178

You will build a linear regression model using these values so you can then predict the price for other houses. For example, a house with 1200 sqft, 3 bedrooms, 1 floor, 40 years old.

Please run the following code to create your `x_train` and `y_train` variables.

```
x_train = np.array([[2104, 5, 1, 45], [1416, 3, 2, 40], [852, 2, 1, 35]])
y_train = np.array([460, 232, 178])
```

2.1 Matrix X containing our examples

Similar to the table above, examples are stored in a NumPy matrix `X_train`. Each row of the matrix represents one example. When you have m training examples (*m* is three in our example),

and there are *n* features (four in our example), **x** is a matrix with dimensions (*m,n*) (*m* rows, *n* columns).

$$\mathbf{X} = \begin{pmatrix} x_0^{(0)} & x_1^{(0)} & \cdots & x_{n-1}^{(0)} \\ x_0^{(1)} & x_1^{(1)} & \cdots & x_{n-1}^{(1)} \\ \cdots & & & \\ x_0^{(m-1)} & x_1^{(m-1)} & \cdots & x_{n-1}^{(m-1)} \end{pmatrix}$$

notation:

- $\mathbf{x}^{(i)}$ is vector containing example i. $\mathbf{x}^{(i)} = (x_0^{(i)}, x_1^{(i)}, \cdots, x_{n-1}^{(i)})$
- $x_j^{(i)}$ is element j in example i. The superscript in parenthesis indicates the example number while the subscript represents an element.

Display the input data.

```
# data is stored in numpy array/matrix
print(f"X Shape: {X_train.shape}, X Type:{type(X_train)})")
print(X_train)
print(f"y Shape: {y_train.shape}, y Type:{type(y_train)})")
print(y_train)
```

2.2 Parameter vector w, b

w is a vector with *n* elements.

- Each element contains the parameter associated with one feature.
- in our dataset, *n* is 4.
- notionally, we draw this as a column vector

$$\mathbf{w} = \begin{pmatrix} w_0 \\ w_1 \\ \cdots \\ w_{n-1} \end{pmatrix}$$

b is a scalar parameter.

For demonstration, w and b will be loaded with some initial selected values that are near the optimal. w is a 1-D NumPy vector.

```
b_init = 785.1811367994083
w_init = np.array([ 0.39133535, 18.75376741, -53.36032453, -
26.42131618])
print(f"w_init shape: {w_init.shape}, b_init type: {type(b_init)}")
```

3 Model Prediction with Multiple Variables

The model's prediction with multiple variables is given by the linear model:

$$f_{\mathbf{w},b}(\mathbf{x}) = w_0x_0 + w_1x_1 + ... + w_{n-1}x_{n-1} + b \tag{1}$$

or in vector notation:

$$f_{\mathbf{w},b}(\mathbf{x}) = \mathbf{w} \cdot \mathbf{x} + b \tag{2}$$

where . is a vector `dot product`

To demonstrate the dot product, we will implement prediction using (1) and (2).

3.1 Single Prediction element by element

Our previous prediction multiplied one feature value by one parameter and added a bias parameter. A direct extension of our previous implementation of prediction to multiple features would be to implement (1) above using loop over each element, performing the multiply with its parameter and then adding the bias parameter at the end.

```
def predict_single_loop(x, w, b):
    """
    single predict using linear regression

    Args:
```

```
      x (ndarray): Shape (n,) example with multiple features
      w (ndarray): Shape (n,) model parameters
      b (scalar):  model parameter

    Returns:
      p (scalar):  prediction
    """
    n = x.shape[0]
    p = 0
    for i in range(n):
        p_i = x[i] * w[i]
        p = p + p_i
    p = p + b
    return p

# get a row from our training data
x_vec = X_train[0,:]
print(f"x_vec shape {x_vec.shape}, x_vec value: {x_vec}")

# make a prediction
f_wb = predict_single_loop(x_vec, w_init, b_init)
print(f"f_wb shape {f_wb.shape}, prediction: {f_wb}")
```

3.2 Single Prediction, vector

Noting that equation (1) above can be implemented using the dot product as in (2) above. We can make use of vector operations to speed up predictions.

Recall from the Python/Numpy lab that NumPy `np.dot()` (https://numpy.org/doc/stable/reference/generated/numpy.dot.html)] can be used to perform a vector dot product.

```
def predict(x, w, b):
    """
    single predict using linear regression
    Args:
      x (ndarray): Shape (n,) example with multiple features
      w (ndarray): Shape (n,) model parameters
      b (scalar):             model parameter

    Returns:
      p (scalar):  prediction
    """
    p = np.dot(x, w) + b
    return p

# get a row from our training data
x_vec = X_train[0,:]
print(f"x_vec shape {x_vec.shape}, x_vec value: {x_vec}")

# make a prediction
f_wb = predict(x_vec,w_init, b_init)
print(f"f_wb shape {f_wb.shape}, prediction: {f_wb}")
```

The results and shapes are the same as the previous version which used looping. Going forward, `np.dot` will be used for these operations. The prediction is now a single statement. Most routines will implement it directly rather than calling a separate predict routine.

4 Compute Cost with Multiple Variables

The equation for the cost function with multiple variables $J(w,b)$ is:

$$J(\mathbf{w},b) = \frac{1}{2m}\sum_{i=0}^{m-1}(f_{\mathbf{w},b}(\mathbf{x}^{(i)}) - y^{(i)})^2 \quad (3)$$

Where:

$$f_{\mathbf{w},b}(\mathbf{x}^{(i)}) = \mathbf{w}\cdot\mathbf{x}^{(i)} + b \quad (4)$$

Task 2: Implement the cost function for multiple variables

In contrast to previous labs, w and $x^{(i)}$ are vectors rather than scalars supporting multiple features.

Implement equations (3) and (4). using for loop over all `*m*` examples

```
def compute_cost(X, y, w, b):
    """
    compute cost
    Args:
      X (ndarray (m,n)): Data, m examples with n features
      y (ndarray (m,)) : target values
      w (ndarray (n,)) : model parameters
      b (scalar)       : model parameter

    Returns:
      cost (scalar): cost
    """
```

```
# Compute and display cost using our pre-chosen optimal parameters.
cost = compute_cost(X_train, y_train, w_init, b_init)
print(f'Cost at optimal w : {cost}')
```

5. Gradient Descent with Multiple Variables

Gradient descent for multiple variables:

$$\text{repeat until convergence: } \{$$

$$w_j = w_j - \alpha \frac{\partial J(\mathbf{w},b)}{\partial w_j} \quad \text{for j = 0..n-1} \tag{5}$$

$$b = b - \alpha \frac{\partial J(\mathbf{w},b)}{\partial b}$$

$$\}$$

where, n is the number of features, parameters w_j, b, are updated simultaneously and where

$$\frac{\partial J(\mathbf{w},b)}{\partial w_j} = \frac{1}{m} \sum_{i=0}^{m-1} (f_{\mathbf{w},b}(\mathbf{x}^{(i)}) - y^{(i)}) x_j^{(i)} \tag{6}$$

$$\frac{\partial J(\mathbf{w},b)}{\partial b} = \frac{1}{m} \sum_{i=0}^{m-1} (f_{\mathbf{w},b}(\mathbf{x}^{(i)}) - y^{(i)}) \tag{7}$$

- m is the number of training examples in the data set
- $f_{\mathbf{w},b}(\mathbf{x}^{(i)})$ is the model's prediction, while $y^{(i)}$ is the target value

5.1 Compute Gradient with Multiple Variables

An implementation for calculating the equations (6) and (7) is below. There are many ways to implement this. In this version, there is an

- outer loop over all m examples.
  - $\frac{\partial J(\mathbf{w},b)}{\partial b}$ for the example can be computed directly and accumulated
  - in a second loop over all n features:
    - $\frac{\partial J(\mathbf{w},b)}{\partial w_j}$ is computed for each w_j.

Task 3: Complete the following compute_gradient function

```
def compute_gradient(X, y, w, b):
    """
    Computes the gradient for linear regression
    Args:
      X (ndarray (m,n)): Data, m examples with n features
      y (ndarray (m,)) : target values
      w (ndarray (n,)) : model parameters
      b (scalar)       : model parameter
```

```
    Returns:
      dj_dw (ndarray (n,)): 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.
    """


#Compute and display gradient
tmp_dj_db,  tmp_dj_dw  =  compute_gradient(X_train,  y_train,  w_init,
b_init)
print(f'dj_db at initial w,b: {tmp_dj_db}')
print(f'dj_dw at initial w,b: \n {tmp_dj_dw}')
```

5.2 Gradient Descent with Multiple Variables

The routine below implements equation (5) above.

Task 4: Complete the gradient_descent function below using equation (5) above

```
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 (m,n))   : Data, m examples with n features
      y (ndarray (m,))    : target values
      w_in (ndarray (n,)) : initial model parameters
      b_in (scalar)       : initial model parameter
```

```
      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 (n,)) : Updated values of parameters
      b (scalar)       : Updated value of parameter
      """
```

In the code below you will test the implementation.

```
# initialize parameters
initial_w = np.zeros_like(w_init)
initial_b = 0.
# some gradient descent settings
iterations = 1000
alpha = 5.0e-7
# run gradient descent
w_final, b_final, J_hist = gradient_descent(X_train, y_train, initial_w,
initial_b,
                                                    compute_cost,
compute_gradient,
                                                    alpha, iterations)
print(f"b,w found by gradient descent: {b_final:0.2f},{w_final} ")
m,_ = X_train.shape
for i in range(m):
```

```
    print(f"prediction: {np.dot(X_train[i], w_final) + b_final:0.2f},
target value: {y_train[i]}")


# plot cost versus iteration

fig, (ax1, ax2) = plt.subplots(1, 2, constrained_layout=True,
figsize=(12, 4))

ax1.plot(J_hist)

ax2.plot(100 + np.arange(len(J_hist[100:])), J_hist[100:])

ax1.set_title("Cost vs. iteration");  ax2.set_title("Cost vs.
iteration (tail)")

ax1.set_ylabel('Cost')             ;  ax2.set_ylabel('Cost')

ax1.set_xlabel('iteration step')   ;  ax2.set_xlabel('iteration step')

plt.show()
```

Feature Scaling and Learning Rate

Feature Engineering and Polynomial Regression