My Notes for Machine Learning

Philip Tracton

Contents

1 Math Review

1.1 Linear Algebra

1.1.1 Matrix

A matrix is a 2-dimensional array of numbers.

N is number of rows. **M** is number of columns.

Matrices are specified as NxM.

A_{ij} is how you specify a single location in a matrix. It is row I and column J.

$$\mathbf{A} = \begin{bmatrix} x_{11} & x_{12} & x_{13} & \dots & x_{1m} \\ x_{21} & x_{22} & x_{23} & \dots & x_{2m} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ x_{N1} & x_{N2} & x_{N3} & \dots & x_{NM} \end{bmatrix} \tag{1.1}$$

A vector is a matrix with one column and many rows and specified as Nx1

v_i is the i^{th} element of the vector V.

$$\mathbf{v} = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_N \end{bmatrix} \tag{1.2}$$

Matrices are usually denoted by uppercase names while vectors are lowercase.

Scalar means that an object is a single value, not a vector or matrix.

$\mathbb{R}$ refers to the set of scalar real numbers.

$\mathbb{R}^n$ refers to the set of n-dimensional vectors of real numbers.

Matlab (or Octave) and Python are 2 very common languages for doing machine learning and math work in general. They are very different languages. Matlab is a number crunching specific language. This is pretty much all it does. Python is a general purpose language with a lot of built up libraries, Numpy in particular, to support doing this type of work.

This document will try to go over all concepts in both languages in order to better understand how the math works from 2 different perspectives.

One of the issues to keep track of is that Matlab starts counting at 1 and Python starts counting at 0! Notice in the code below that A(2,3) in Matlab will get you the same location as A[1][2] in Python for the same matrices

This is an example of creating a simple 3x3 matrix in both Matlab and Python. Notice that Matlab is a little bit simpler. There is no need to import a library for it. Python doesn't need one either technically, but numpy will be used a lot for machine learning work and we should just start with it.

Matlab

```
% The ; denotes we are going back to a new row.
A = [1, 2, 3; 4, 5, 6; 7, 8, 9; 10, 11, 12]

% Initialize a vector
v = [1;2;3]

% Get the dimension of the matrix A where m = rows and n = columns
[m,n] = size(A)

% You could also store it this way
dim_A = size(A)

% Get the dimension of the vector v
dim_v = size(v)

% Now let's index into the 2nd row 3rd column of matrix A
A_23 = A(2,3)
```

Python

```
#! /usr/bin/env python3

import numpy as np

A = np.array([[1, 2, 3], [3, 4, 5], [7, 8, 9]])
rows, cols = np.shape(A)
print("\nA= {}".format(A))
print("Rows = {} Cols = {}".format(rows, cols))
print("Location 2,3 = {}".format(A[1][2]))

V = np.array([[1], [2], [3], [4], [5], [6], [7], [8]])
rows, cols = np.shape(V)
print("\nV= {}".format(V))
print("Rows = {} Cols = {}".format(rows, cols))
print("Location 6,1 = {}".format(V[5][0]))
```

1.1.2 Matrix Operations

Addition and Subtraction

Addition and subtraction are element-wise, so you simply add or subtract each corresponding element:
To add or subtract two matrices, their dimensions must be the **same**.

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} + \begin{bmatrix} w & x \\ y & z \end{bmatrix} = \begin{bmatrix} a+w & b+x \\ c+y & d+z \end{bmatrix} \tag{1.3}$$

Matlab

```
A = [1 2; 3 4]
B = [11 12; 13 14]
C = A + B
```

Python

```
#! /usr/bin/env python3

import numpy as np

A = np.array([[1, 2], [3, 4]])
B = np.array([[11, 12], [13, 14]])
C = A + B
print("A = {}".format(A))
print("B = {}".format(B))
print("C = {}".format(C))
```

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} - \begin{bmatrix} w & x \\ y & z \end{bmatrix} = \begin{bmatrix} a-w & b-x \\ c-y & d-z \end{bmatrix} \tag{1.4}$$

Matlab

```
A = [1 2; 3 4]
B = [11 12; 13 14]
C = A - B
```

Python

```
#! /usr/bin/env python3

import numpy as np

A = np.array([[1, 2], [3, 4]])
B = np.array([[11, 12], [13, 14]])
C = A - B
print("A = {}".format(A))
print("B = {}".format(B))
print("C = {}".format(C))
```

For matrix addition/subtraction there is not much of a difference. Python takes a little more set up in that you need to import numpy, but the actual operational step is identical.

Scalar Multiplication and Division

In scalar multiplication, we simply multiply every element by the scalar value:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} * x = \begin{bmatrix} a*x & b*x \\ c*x & d*x \end{bmatrix} \tag{1.5}$$

In scalar division, we simply divide every element by the scalar value:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} / x = \begin{bmatrix} a/x & b/x \\ c/x & d/x \end{bmatrix} \tag{1.6}$$

Matlab

```
A = [10 20; 30 40]
x = 10
C = A*x
D = A/x
```

Python

```
#! /usr/bin/env python3

import numpy as np

A = np.array([[10, 20], [30, 40]])
x = 10
C = A * x
D = A/x
print("A = {}".format(A))
print("C = {}".format(C))
print("D = {}".format(D))
```

Matrix Vector Multiplication

The result is a **vector**. The number of **columns** of the matrix must equal the number of **rows** of the vector.

An **m x n matrix** multiplied by an **n x 1** vector results in an **m x 1** vector.

Some more Math Insights

Below is an example of a matrix-vector multiplication. Make sure you understand how the multiplication works.

$$\begin{bmatrix} a & b \\ c & d \\ e & f \end{bmatrix} * \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} a*x+b*y \\ c*x+d*y \\ e*x+f*y \end{bmatrix} \tag{1.7}$$

Matrix Matrix Multiplication

This is also known as the dot product.

An **m x n** matrix multiplied by an **n x o** matrix results in an **m x o** matrix. In the example, a 3 x 2 matrix times a 2 x 2 matrix resulted in a 3 x 2 matrix.

To multiply two matrices, the number of columns of the first matrix must equal the number of rows of the second matrix. The process is to take each row of the first matrix and multiply it by each column of the second matrix. Iterate this through each row in the first matrix with each column in the second matrix.

You can **NOT** reverse the order. $\mathbf{A} * \mathbf{B}$ is not $\mathbf{B} * \mathbf{A}$
Multiplication is associative. $(\mathbf{A} * \mathbf{B}) * \mathbf{C} = \mathbf{A} * (\mathbf{B} * \mathbf{C})$

$$\begin{bmatrix} a & b \\ c & d \\ e & f \end{bmatrix} * \begin{bmatrix} w & x \\ y & z \end{bmatrix} = \begin{bmatrix} a*w+b*y & a*x+b*z \\ c*w+d*y & c*x+d*z \\ e*w+f*y & e*x+f*z \end{bmatrix} \tag{1.8}$$

Matlab

```
A = [10 20; 30 40; 50 60]
B = [5; 10]
C = A* B
% The line below fails since A*B is not B*A
%D = B *A
```

Python

```
#! /usr/bin/env python3

import numpy as np

A = np.array([[10, 20], [30, 40], [50, 60]])
B = np.array([[5], [10]])
C = A.dot(B)
print("A = {}".format(A))
print("B = {}".format(B))
print("C = {}".format(C))
# The line below fails since A*B is not B*A
#D = B.dot(A)
```

Notice the syntax is starting to differ more. For Matlab, you can just multiply the vectors like any other variable. In python we need to use the dot method. Notice it is A that calls dot with B as a parameter.

Identity

The identity matrix is a square matrix (m=n) that has 1's along the diagnal and zeros everywhere else and is usually denoted by the letter **I**.
The identity matrix, when multiplied by any matrix of the same dimensions, results in the original matrix. It's just like multiplying numbers by 1. The identity matrix simply has 1's on the diagonal (upper left to lower right diagonal) and 0's elsewhere.

When multiplying the identity matrix after some matrix $(A * I)$, the square identity matrix's dimension should match the other matrix's **columns**. When multiplying the identity matrix before some other matrix (I*A), the square identity matrix's dimension should match the other matrix's **rows**.

$$\mathbf{I} = \begin{bmatrix} 1 & 0 & 0 & \dots & 0 \\ 0 & 1 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 1 \end{bmatrix} \tag{1.9}$$

Matlab

```
% Initialize random matrices A and B
A = [1,2;4,5]
B = [1,1;0,2]

% Initialize a 2 by 2 identity matrix
I = eye(2)

% The above notation is the same as I = [1,0;0,1]

% What happens when we multiply I*A ?
IA = I*A

% How about A*I ?
AI = A*I

% Compute A*B
AB = A*B

% Is it equal to B*A?
BA = B*A

% Note that IA = AI but AB != BA
```

Python

```
#! /usr/bin/env python3

import numpy as np

# Initialize random matrices A and B
A = np.array([[1, 2],
              [4, 5]])
B = np.array([[1, 1],
              [0, 2]])

# Initialize a 2 by 2 identity matrix
I = np.eye(2)

# The above notation is the same as I = [1, 0
#
 0, 1]

# What happens when we multiply I*A ?
IA = I.dot(A)

# How about A*I ?
AI = A.dot(I)

# Compute A*B
AB = A.dot(B)

# Is it equal to B*A?
BA = B.dot(A)

# Note that IA = AI but AB != BA
print("A = {}".format(A))
print("B = {}".format(B))
print("IA = {}".format(IA))
print("AI = {}".format(AI))
print("AB = {}".format(AB))
print("BA = {}".format(BA))
```

Again the syntax is still mostly the same. Matlab multiplies are calls to the dot method in Python. The eye() function in Matlab does the same thing as the numpy version of eye(). The

difference is that in Matlab it is a built in function and in Python you need to call the method inside the Numpy library.

Transpose

The **transposition** of a matrix is like rotating the matrix 90° in clockwise direction and then reversing it. We can compute transposition of matrices in matlab with the transpose(A) function or A'

In other words: $A_{ij} = A^T_{ji}$

Matlab

```
% Initialize matrix A
A = [1,2,0;0,5,6;7,0,9]

% Transpose A
A_trans = A'
```

Python

```
#! /usr/bin/env python3

import numpy as np

A = np.array([[1, 2, 0], [0, 5, 6],
 ↪  [7, 0, 9]])
A_trans = A.transpose()
print("A = {}".format(A))
print("A_trans = {}".format(A_trans))
```

In Matlab the ' operator will transpose a matrix. There is a transpose function in Matlab but the ' notation is very common and easier.

In Python we must call the transpose method on our matrix.

Inverse

The inverse of a matrix A is denoted A^{-1}. Multiplying by the inverse results in the identity matrix.

$$I = A * A^{-1} \tag{1.10}$$

A non square matrix does not have an inverse matrix. We can compute inverses of matrices in Octave with the pinv(A) function and in Matlab with the inv(A) function. Matrices that don't have an inverse are singular or degenerate.

Matlab

```
% Initialize matrix A
A = [1,2,0;0,5,6;7,0,9]

% Take the inverse of A
A_inv = inv(A)

% What is A^(-1)*A?
A_invA = inv(A)*A
```

Python

```
#! /usr/bin/env python3

import numpy as np
import numpy.linalg

A = np.array(([1, 2, 0], [0, 5, 6],
 ↪  [7, 0, 9]))
A_inv = numpy.linalg.inv(A)
A_invA = A.dot(A_inv)
print("A = {}".format(A))
print("A_inv = {}".format(A_inv))
print("A_invA = {}".format(A_invA))
```

The Matlab code is pretty straight forward. You can call the inv() function on a matrix. You can multiply the original matrix by its inverse and get the identity.

In Python it is a little more complicated. We need to bring in the numpy Linear Algebra library. Once this library is brought in, we can use the inv() method in it on our matrix. From there we can do all the same things as in Matlab but with our Python syntax.

1.2 Calculus

1.3 Probability

1.4 Statistics

2 Introduction to Machine Learning

2.1 Introduction to Machine Learning

Machine Learning: Field of study that gives computers the ability to learn without being explicitly programmed.

Well-posed Learning Problem: A computer program is said to learn from experience E with respect to some task T and some performance measure P, if its performance on T, as measured by P, improves with experience E.

Machine Learning Algorithms

- Supervised Learning
- Unsupervised Learning
- Reinforcement Learning
- Recommender Systems

2.1.1 Supervised Learning

In supervised learning, we are given a data set and already know what our correct output should look like, having the idea that there is a relationship between the input and the output.

Supervised learning problems are categorized into ”regression” and ”classification” problems. In a regression problem, we are trying to predict results within a continuous output, meaning that we are trying to map input variables to some continuous function. In a classification problem, we are instead trying to predict results in a discrete output. In other words, we are trying to map input variables into discrete categories.

In **supervised learning**, the ”right answers” are known.

A **Regression** is predicting a value! For example Given a picture of a person, we have to predict their age on the basis of the given picture

A **classification** is breaking into groups or discrete values. For example Given a patient with a tumor, we have to predict whether the tumor is malignant or benign.

2.1.2 Unsupervised Learning

In **unsupervised learning**, the ”right answers” are **not** known.

Unsupervised learning allows us to approach problems with little or no idea what our results should look like. We can derive structure from data where we don't necessarily know the effect of the variables.

We can derive this structure by clustering the data based on relationships among the variables in the data.

With unsupervised learning there is no feedback based on the prediction results.

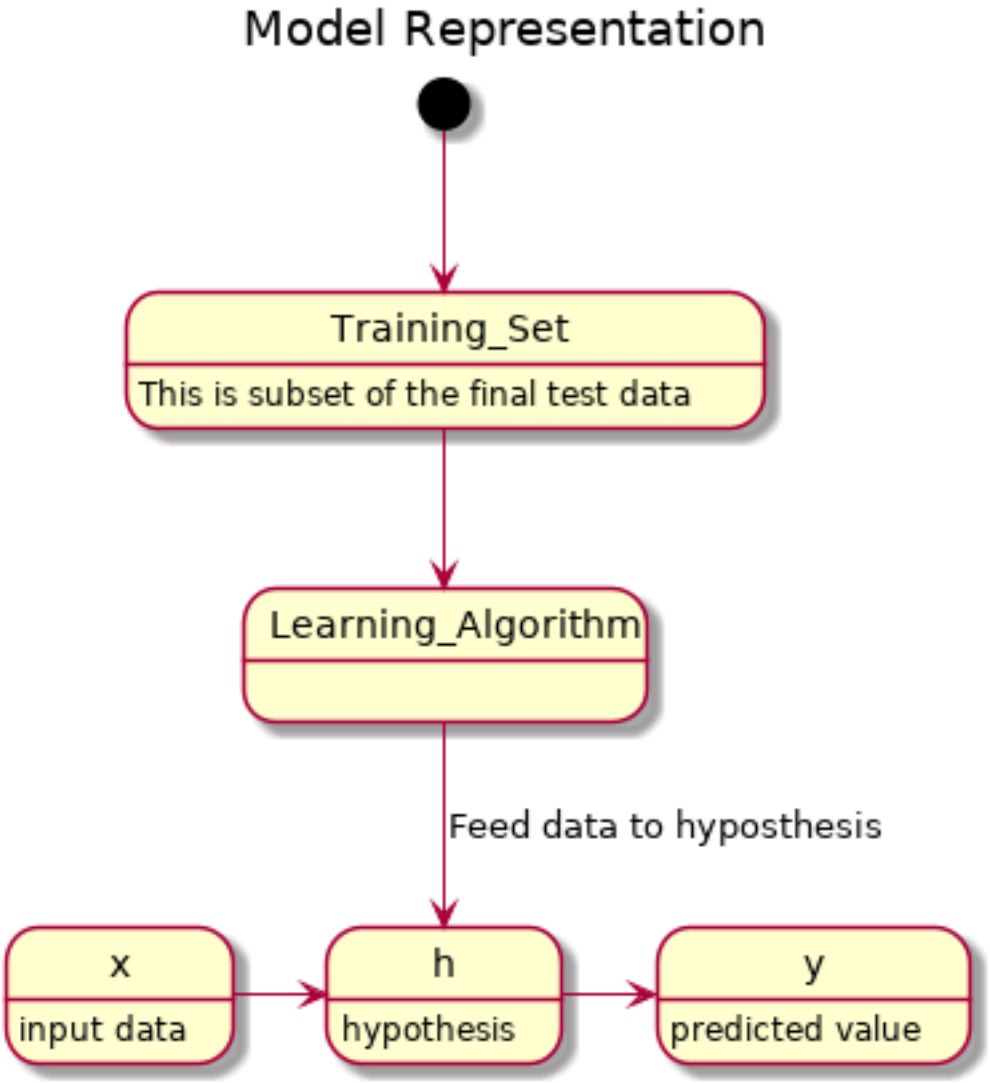
2.1.3 Model

Input variables or input features are denoted as $x^{(i)}$
Output or target variable we are trying to predict are denoted as $y^{(i)}$
The pair $(x^{(i)}, y^{(i)})$ is called a training example.

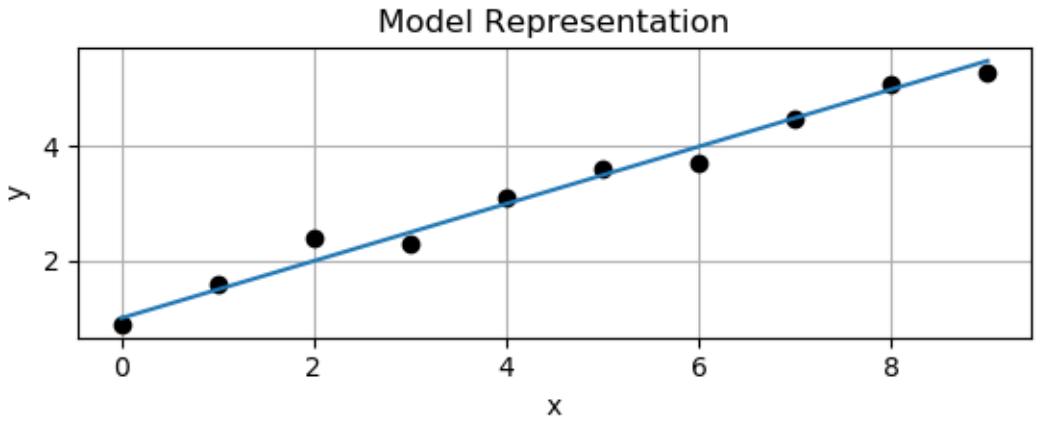
To establish notation for future use, we'll use $x^{(i)}$ to denote the **"input"** variables (living area in this example), also called input features, and $y^{(i)}$ to denote the **"output"** or target variable that we are trying to predict (price). A pair $(x^{(i)}, y^{(i)})$ is called a **training example**, and the dataset that we'll be using to learn—a list of m training examples $(x^{(i)}, y^{(i)})$; i=1,...,m—is called a **training set**. Note that the superscript "(i)" in the notation is simply an index into the training set, and has nothing to do with exponentiation. We will also use **X** to denote the space of input values, and **Y** to denote the space of output values. In this example, X = Y = $\mathbb{R}$.
To describe the supervised learning problem slightly more formally, our goal is, given a training set, to learn a function

$$h : X \to Y \tag{2.1}$$

so that h(x) is a "good" predictor for the corresponding value of y. For historical reasons, this function h is called a hypothesis. Seen pictorially, the process is therefore like this:



When the target variable that we're trying to predict is continuous, such as in our housing example, we call the learning problem a regression problem. When y can take on only a small number of discrete values (such as if, given the living area, we wanted to predict if a dwelling is a house or an apartment, say), we call it a classification problem.



This is an attempt at a linear fit for this data. It is not quite a match, but it is very close. The plot for the line is $y = 0.5x + 1$ which is our hypothesis fuction h.

2.1.4 Cost Functions

We can measure the accuracy of our hypothesis function by using a **cost function**. This takes an average difference (actually a fancier version of an average) of all the results of the hypothesis with inputs from x's and the actual output y's.

$$J(\theta_0, \theta_1) = \frac{1}{2m} \sum_{i=1}^{m} (\hat{y} - y^i)^2 = \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x^i) - y^i)^2 \tag{2.2}$$

m is the number of training examples. The goal is to minimize $J(\theta_0, \theta_1)$.
To break it apart, it is $\frac{1}{2}\bar{x}$ where $\bar{x}$ is the mean of the squares of $h_\theta(x^i) - y^i$, or the difference between the predicted value and the actual value.

This function is otherwise called the **Squared error function**, or **Mean squared error**. The mean is halved $\frac{1}{2}$ as a convenience for the computation of the gradient descent, as the derivative term of the square function will cancel out the $\frac{1}{2}$ term.

For our previous example of $y = 0.5x + 1$, we have $h_\theta(x^i) = \theta_0 + \theta_1 x^i$. This leads to our cost function of

$$J(\theta_0, \theta_1) = \frac{1}{2m} \sum_{i=1}^{m} (\theta_0 + \theta_1 x^i - y^i)^2 \tag{2.3}$$

Choose θ_0, θ_1 so that $h_\theta(x)$ is close to y for training examples (x, y)

KEY NOTE: The cost function calculates a single value based on a specific θ_0, θ_1 pair. It does not find the best θ_0, θ_1 values. These values are supplied. Gradient Descent is the process of finding the best θ_0, θ_1 values.
If we try to think of it in visual terms, our training data set is scattered on the x-y plane. We are trying to make a straight line (defined by $h_\theta(x)$) which passes through these scattered data points.

Our objective is to get the best possible line. The best possible line will be such so that the average squared vertical distances of the scattered points from the line will be the least. Ideally, the line should pass through all the points of our training data set. In such a case, the value of $J(\theta_0, \theta_1)$ will be 0.

This is a cost function implementation for the simple linear equation of $h_{theta}(x) = \theta_0 + \theta_1 x$

Matlab

```
function J = cost_function(X, y,
 ↪  theta)
% This function computes the cost for
 ↪  this specific pair of theta
 ↪  values.
% It is expecting theta to be a 2
 ↪  element vector.
 theta

  % Initialize some useful values
  m = length(y); % number of training
   ↪  examples

  % You need to return the following
   ↪  variables correctly
  J = 0;

  i = 1:m;
  J = (1/(2*m)) * sum(
   ↪  ((theta(1).*X(i,1) + theta(2)
   ↪  .* X(i,2)) - y(i)) .^ 2);

end
```

Python

```
#! /usr/bin/env python3


def hypothesis_linear(theta0=None,
 ↪  theta1=None, x=None):
    return theta0 + x*theta1


def cost_function(X=None, Y=None,
 ↪  theta=None):
    # print("Cost Function X={} Y={}
     ↪  theta={}".format(
    #    X, Y, theta))
    m = len(Y)

    sum_diff_squared = 0
    cost = 0
    diff = 0
    diff_squared = 0
    for i in range(m):
        diff = hypothe-
         ↪  sis_linear(theta[0],
         ↪  theta[1], X[i]) - Y[i]
        diff_squared = diff * diff
        sum_diff_squared =
         ↪  sum_diff_squared +
         ↪  diff_squared
    cost = 1/(2*m) * sum_diff_squared
    return cost
```

These are the test scripts for driving the cost_function implementations. Both are fed the same input data and come back with the same cost value of 0.019.

Matlab

```
clear;

Y_GOLDEN = [0.9, 1.6, 2.4, 2.3, 3.1, 3.6, 3.7, 4.5, 5.1, 5.3]'
theta = zeros(2,1); % 2 rows x 1 column of 0

m = length(Y_GOLDEN);
x = [0:1:m]';
x = [ones(m+1,1), x(:,1)]
J_min = cost_function(x, Y_GOLDEN, [1, 0.5])
```

Python

```
#! /usr/bin/env python3

import numpy as np
import cost_function

Y_GOLDEN = [0.9, 1.6, 2.4, 2.3, 3.1, 3.6, 3.7, 4.5, 5.1, 5.3]
X = np.arange(0, 10, 1)
my_cost = cost_function.cost_function(X, Y_GOLDEN, (1, 0.5))
print("FOUND: Cost = {:02f}".format(my_cost))
```

This is an example of trying different values for $\theta_0 and \theta_1$ and plotting the results.

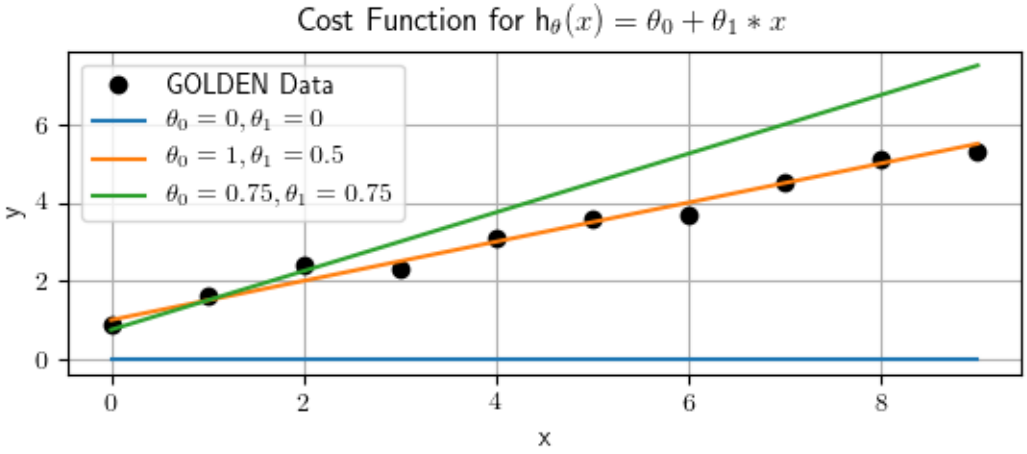
Matlab

```
clear;

theta = zeros(2,1); % 2 rows x 1 column of 0
Y_GOLDEN = [0.9, 1.6, 2.4, 2.3, 3.1, 3.6, 3.7, 4.5, 5.1, 5.3]';
m = length(Y_GOLDEN);
x = [0:1:m]';
x = [ones(m,1), x(1:10,1)];
Y1 = hypothesis_linear(0,0, x);
Y2 = hypothesis_linear(1,0.5, x);
Y3 = hypothesis_linear(0.75,0.75, x);

figure;
plot(x(:,2), Y_GOLDEN, 'ko', 'DisplayName', 'Y_GOLDEN')
grid on;
hold on;
plot(x(:,2), Y1)
plot(x(:,2), Y2)
plot(x(:,2), Y3)
hold off
legend("Y\_GOLDEN", '\theta_{0} =0, \theta_{1} = 0', ['\theta_{0}=1, ' ...
                    '\theta_{1} = 0.5'], '\theta_{0} =0.75, \theta_{1} = 0.75')
xlabel("x")
ylabel("y")
title('Cost Function for h_{\theta}(x) = \theta_{0} + \theta_{1} * x')
saveas(gcf, 'plot_cost_function.png')
```

Python

```python
#! /usr/bin/env python3

import cost_function

import numpy as np
import matplotlib
matplotlib.rcParams['text.usetex'] = True
import matplotlib.pyplot as plt

Y_GOLDEN = [0.9, 1.6, 2.4, 2.3, 3.1, 3.6, 3.7, 4.5, 5.1, 5.3]
X = np.arange(0, 10, 1)

Y1 = cost_function.hypothesis_linear(0, 0, X)
Y2 = cost_function.hypothesis_linear(1, 0.5, X)
Y3 = cost_function.hypothesis_linear(0.75, 0.75, X)

fig = plt.figure()
ax1 = fig.add_subplot(211)
ax1.set_ylabel('y')
ax1.set_xlabel('x')
ax1.set_title(r'Cost Function for h_{\theta}(x) = \theta_{0} + \theta_{1} * x')

ax1.grid(True)
line0, = ax1.plot(X, Y_GOLDEN, 'ko', label="GOLDEN Data")
line1, = ax1.plot(X, Y1, label=r'\theta_{0} =0, \theta_{1} = 0')
line2, = ax1.plot(X, Y2, label=r'\theta_{0} =1, \theta_{1} = 0.5')
line3, = ax1.plot(X, Y3, label=r'\theta_{0} =0.75, \theta_{1} = 0.75')
plt.legend()
# plt.show()
plt.savefig("plot_cost_function.png")
```

This is the graph created by python, which looks better than the one from Matlab/Octave. There are 3 line and circles for the actual data. The first line is the blue one across the X-Axis since we are using 0,0. The best fit line in red has the ideal value for minimizing cost. The last line shows it being close, but a little off.



2.1.5 Gradient Descent

KEY NOTE: Gradient descent uses the hyposthesis function and cost function by iterating through various θ_0 and θ_1 until it finds the minimum value for the cost function. The θ_0 and θ_1 for that position are the ones we want!

So we have our hypothesis function and we have a way of measuring how well it fits into the data. Now we need to estimate the parameters in the hypothesis function. That's where gradient descent comes in.

Imagine that we graph our hypothesis function based on its fields θ_0 and θ_1 (actually we are graphing the cost function as a function of the parameter estimates). We are not graphing x and y itself, but the parameter range of our hypothesis function and the cost resulting from selecting a particular set of parameters.

We put θ_0 on the x axis and θ_1 on the y axis, with the cost function on the vertical z axis. The

points on our graph will be the result of the cost function using our hypothesis with those specific theta parameters. The graph below depicts such a setup.

PUT GRAPH HERE
We will know that we have succeeded when our cost function is at the very bottom of the pits in our graph, i.e. when its value is the minimum. The red arrows show the minimum points in the graph.

The way we do this is by taking the derivative (the tangential line to a function) of our cost function. The slope of the tangent is the derivative at that point and it will give us a direction to move towards. We make steps down the cost function in the direction with the steepest descent. The size of each step is determined by the parameter α, which is called the learning rate.

For example, the distance between each 'star' in the graph above represents a step determined by our parameter α. A smaller α would result in a smaller step and a larger α results in a larger step. The direction in which the step is taken is determined by the partial derivative of $J(\theta_0, \theta_1)$. Depending on where one starts on the graph, one could end up at different points. The image above shows us two different starting points that end up in two different places.

The Gradient Descent Algorightm is:

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1) \tag{2.4}$$

Repeat this until it converges!