

My Notes for Machine Learning

Philip Tracton

CONTENTS

1	Math Review	2
1.1	Linear Algebra	2
1.1.1	Matrix	2
1.1.2	Matrix Operations	4
1.2	Calculus	10
1.3	Probability	10
1.4	Statistics	10
2	Introduction to Machine Learning	11
2.1	Introduction to Machine Learning	11
2.1.1	Supervised Learning	11
2.1.2	Unsupervised Learning	11
2.1.3	Model	12
2.1.4	Cost Functions	14
2.1.5	Gradient Descent	20
2.1.6	Multivariate Linear Regression	25
2.1.7	Gradient Descent for Multiple Variables	25
2.1.8	Feature Scaling	26
2.1.9	Gradient Descent Learning Rate	27
2.1.10	Features and Polynomial Regression	27
2.1.11	Normal Equation	27

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.

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

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.

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

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

Python

```

1  % The ; denotes we are going back to
   ↪ a new row.
2  A = [1, 2, 3; 4, 5, 6; 7, 8, 9; 10,
   ↪ 11, 12]
3
4  % Initialize a vector
5  v = [1;2;3]
6
7  % Get the dimension of the matrix A
   ↪ where m = rows and n = columns
8  [m,n] = size(A)
9
10 % You could also store it this way
11 dim_A = size(A)
12
13 % Get the dimension of the vector v
14 dim_v = size(v)
15
16 % Now let's index into the 2nd row
   ↪ 3rd column of matrix A
17 A_23 = A(2,3)

```

```

1  #! /usr/bin/env python3
2
3  import numpy as np
4
5  A = np.array([[1, 2, 3], [3, 4, 5],
   ↪ [7, 8, 9]])
6  rows, cols = np.shape(A)
7  print("\nA= {}".format(A))
8  print("Rows = {} Cols =
   ↪ {}".format(rows, cols))
9  print("Location 2,3 =
   ↪ {}".format(A[1][2]))
10
11 V = np.array([[1], [2], [3], [4],
   ↪ [5], [6], [7], [8]])
12 rows, cols = np.shape(V)
13 print("\nV= {}".format(V))
14 print("Rows = {} Cols =
   ↪ {}".format(rows, cols))
15 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**.

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

Matlab

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

Python

```
1  #!/usr/bin/env python3
2
3  import numpy as np
4
5  A = np.array([[1, 2], [3, 4]])
6  B = np.array([[11, 12], [13, 14]])
7  C = A + B
8  print("A = {}".format(A))
9  print("B = {}".format(B))
10 print("C = {}".format(C))
```

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

Matlab

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

Python

```
1  #!/usr/bin/env python3
2
3  import numpy as np
4
5  A = np.array([[1, 2], [3, 4]])
6  B = np.array([[11, 12], [13, 14]])
7  C = A - B
8  print("A = {}".format(A))
9  print("B = {}".format(B))
10 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:

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

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

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

Matlab

```
1 A = [10 20; 30 40]
2 x = 10
3 C = A*x
4 D = A/x
```

Python

```
1  #!/usr/bin/env python3
2
3  import numpy as np
4
5  A = np.array([[10, 20], [30, 40]])
6  x = 10
7  C = A * x
8  D = A/x
9  print("A = {}".format(A))
10 print("C = {}".format(C))
11 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.

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

Matrix Matrix Multiplication

This is also known as the dot product.

An $m \times n$ matrix multiplied by an $n \times o$ matrix results in an $m \times o$ matrix. In the example, a 3×2 matrix times a 2×2 matrix resulted in a 3×2 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. $A * B$ is not $B * A$

Multiplication is associative. $(A * B) * C = A * (B * C)$

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

Matlab

```
1 A = [10 20; 30 40; 50 60]
2 B = [5; 10]
3 C = A * B
4 % The line below fails since A*B is
  ↪ not B*A
5 %D = B * A
```

Python

```
1 #! /usr/bin/env python3
2
3 import numpy as np
4
5 A = np.array([[10, 20], [30, 40],
  ↪ [50, 60]])
6 B = np.array([[5], [10]])
7 C = A.dot(B)
8 print("A = {}".format(A))
9 print("B = {}".format(B))
10 print("C = {}".format(C))
11 # The line below fails since A*B is
  ↪ not B*A
12 #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 diagonal 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**.

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

Matlab

```

1 % Initialize random matrices A and B
  ↪
2 A = [1,2;4,5]
3 B = [1,1;0,2]
4
5 % Initialize a 2 by 2 identity
  ↪ matrix
6 I = eye(2)
7
8 % The above notation is the same as I
  ↪ = [1,0;0,1]
9
10 % What happens when we multiply I*A ?
  ↪
11 IA = I*A
12
13 % How about A*I ?
14 AI = A*I
15
16 % Compute A*B
17 AB = A*B
18
19 % Is it equal to B*A?
20 BA = B*A
21
22 % Note that IA = AI but AB != BA

```

Python

```

1 #!/usr/bin/env python3
2
3 import numpy as np
4
5 # Initialize random matrices A and B
6 A = np.array([[1, 2],
7               [4, 5]])
8 B = np.array([[1, 1],
9               [0, 2]])
10
11 # Initialize a 2 by 2 identity
12 ↪ matrix
13 I = np.eye(2)
14
15 # The above notation is the same as I
16 ↪ = [1, 0
17 #
18 ↪ 0, 1]
19
20 # What happens when we multiply I*A
21 ↪ ?
22 IA = I.dot(A)
23
24 # How about A*I ?
25 AI = A.dot(I)
26
27 # Compute A*B
28 AB = A.dot(B)
29
30 # Is it equal to B*A?
31 BA = B.dot(A)
32
33 # Note that IA = AI but AB != BA
34 print("A = {}".format(A))
35 print("B = {}".format(B))
36 print("IA = {}".format(IA))
37 print("AI = {}".format(AI))
38 print("AB = {}".format(AB))
39 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: $A_{ij} = A_{ji}^T$

Matlab

```
1 % Initialize matrix A
2 A = [1,2,0;0,5,6;7,0,9]
3
4 % Transpose A
5 A_trans = A'
```

Python

```
1 #! /usr/bin/env python3
2
3 import numpy as np
4
5 A = np.array([[1, 2, 0], [0, 5, 6],
6              ↪ [7, 0, 9]])
7 A_trans = A.transpose()
8 print("A = {}".format(A))
9 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.

$$I = A * A^{-1} \quad (1.10)$$

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

```

1 % Initialize matrix A
2 A = [1,2,0;0,5,6;7,0,9]
3
4 % Take the inverse of A
5 A_inv = inv(A)
6
7 % What is  $A^{-1} \cdot A$ ?
8 A_invA = inv(A)*A

```

Python

```

1 #!/usr/bin/env python3
2
3 import numpy as np
4 import numpy.linalg
5
6 A = np.array([[1, 2, 0], [0, 5, 6],
7              ↪ [7, 0, 9]])
8 A_inv = numpy.linalg.inv(A)
9 A_invA = A.dot(A_inv)
10 print("A = {}".format(A))
11 print("A_inv = {}".format(A_inv))
12 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.

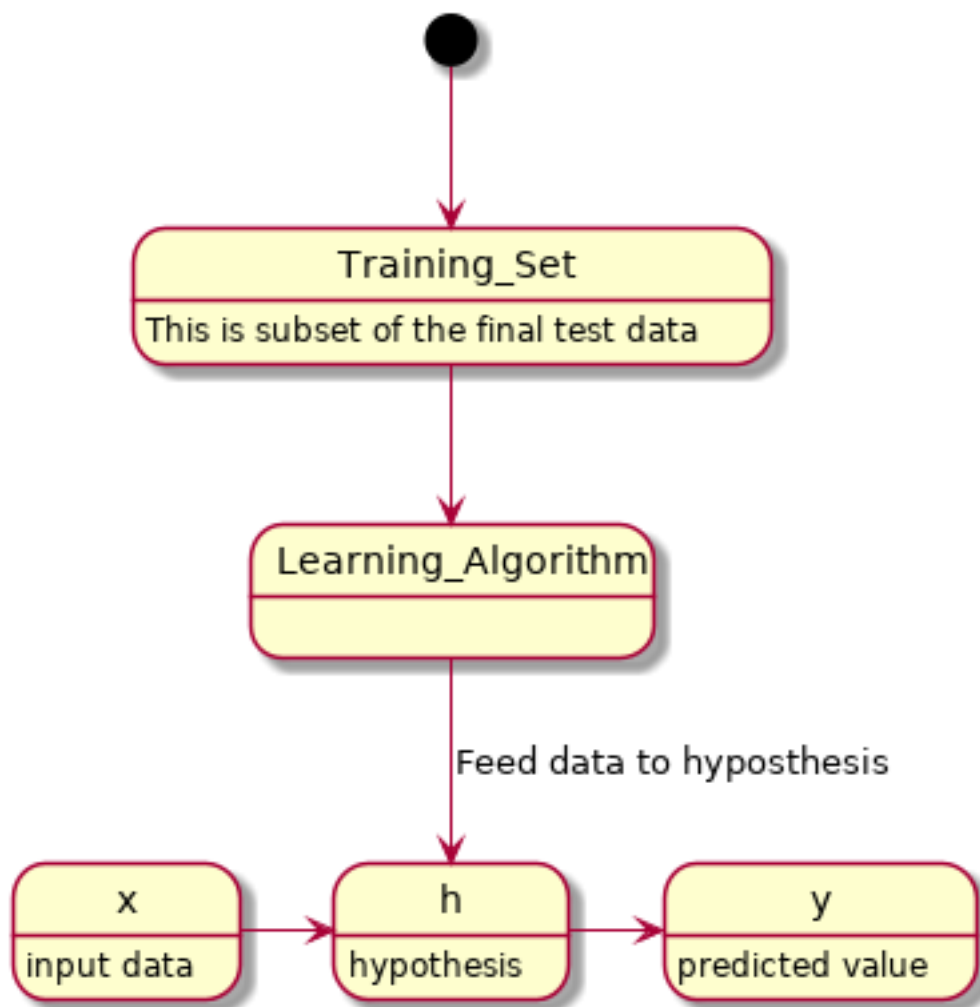
To establish notation for future use, we'll use $x^{(i)}$ to denote the “**input**” variables (living area in this example), also called input features, and $y^{(i)}$ to denote the “**output**” or target variable that we are trying to predict (price). A pair $(x^{(i)}, y^{(i)})$ is called a **training example**, and the dataset that we'll be using to learn—a list of m training examples $(x^{(i)}, y^{(i)})$; $i=1, \dots, m$ —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 \mathbf{X} to denote the space of input values, and \mathbf{Y} to denote the space of output values. In this example, $\mathbf{X} = \mathbf{Y} = \mathbb{R}$.

To describe the supervised learning problem slightly more formally, our goal is, given a training set, to learn a function

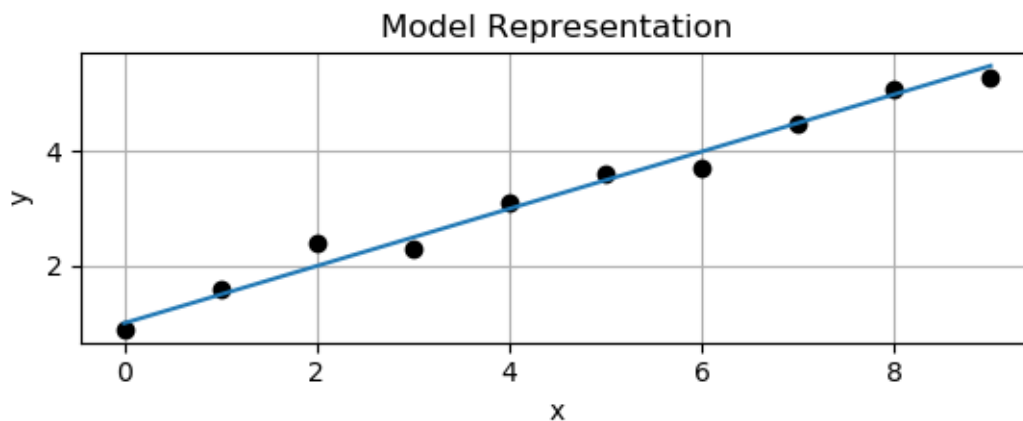
$$h : \mathbf{X} \rightarrow \mathbf{Y} \quad (2.1)$$

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:

Model Representation



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 function 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.

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

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

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

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

```

1 function J = cost_function(X, y,
    ↪ theta)
2 % This function computes the cost for
    ↪ this specific pair of theta
    ↪ values.
3 % It is expecting theta to be a 2
    ↪ element vector.
4 theta
5
6 % Initialize some useful values
7 m = length(y); % number of training
    ↪ examples
8
9 % You need to return the following
    ↪ variables correctly
10 J = 0;
11
12 i = 1:m;
13 J = (1/(2*m)) * sum(
    ↪ ((theta(1).*X(i,1) + theta(2)
    ↪ .* X(i,2)) - y(i)) .^ 2);
14
15 end

```

Python

```

1  #!/usr/bin/env python3
2
3
4  def hypothesis_linear(theta0=None,
    ↪ theta1=None, x=None):
5      return theta0 + x*theta1
6
7
8  def cost_function(X=None, Y=None,
    ↪ theta=None):
9      # print("Cost Function X={} Y={}
    ↪ theta={}").format(
10         # X, Y, theta))
11         m = len(Y)
12
13         sum_diff_squared = 0
14         cost = 0
15         diff = 0
16         diff_squared = 0
17         for i in range(m):
18             diff = hypothe-
    ↪ sis_linear(theta[0],
    ↪ theta[1], X[i]) - Y[i]
19             diff_squared = diff * diff
20             sum_diff_squared =
    ↪ sum_diff_squared +
    ↪ diff_squared
21         cost = 1/(2*m) * sum_diff_squared
22         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

```

1 clear;
2
3 Y_GOLDEN = [0.9, 1.6, 2.4, 2.3, 3.1,
4   ↪ 3.6, 3.7, 4.5, 5.1, 5.3, 5.6]';
5 theta = zeros(2,1); % 2 rows x 1
6   ↪ column of 0
7
8 m = length(Y_GOLDEN);
9 x = [0:1:m]';
10 x = [ones(m,1), x(:,1)]
11 J_min = cost_function(x, Y_GOLDEN,
12   ↪ [1, 0.5])

```

Python

```

1 #!/usr/bin/env python3
2
3 import numpy as np
4 import cost_function
5
6 Y_GOLDEN = [0.9, 1.6, 2.4, 2.3, 3.1,
7   ↪ 3.6, 3.7, 4.5, 5.1, 5.3]
8 X = np.arange(0, 10, 1)
9 my_cost =
10   ↪ cost_function.cost_function(X,
11   ↪ Y_GOLDEN, (1, 0.5))
12 print("FOUND: Cost = {:.02f}"
13   ↪ ".format(my_cost))

```

This is an example of trying different values for θ_0 and θ_1 and plotting the results.

Matlab

Python

```

1 clear;
2
3 theta = zeros(2,1); % 2 rows x 1
   ↪ column of 0
4 Y_GOLDEN = [0.9, 1.6, 2.4, 2.3, 3.1,
   ↪ 3.6, 3.7, 4.5, 5.1, 5.3]';
5 m = length(Y_GOLDEN);
6 x = [0:1:m]';
7 x = [ones(m,1), x(1:10,1)];
8 Y1 = hypothesis_linear(0,0, x);
9 Y2 = hypothesis_linear(1,0.5, x);
10 Y3 = hypothesis_linear(0.75,0.75, x);
11
12 figure;
13 plot(x(:,2), Y_GOLDEN, 'ko',
   ↪ 'DisplayName', 'Y_GOLDEN')
14 grid on;
15 hold on;
16 plot(x(:,2), Y1)
17 plot(x(:,2), Y2)
18 plot(x(:,2), Y3)
19 hold off
20 legend("Y\GOLDEN", '\theta_0 = 0,
   ↪ \theta_1 = 0', ['\theta_0 = 1,
   ↪ ' ...
21                               '\theta_1 =
   ↪ 0.5'],
   ↪ '\theta_0 =
   ↪ 0.75,
   ↪ \theta_1 =
   ↪ 0.75')
22 xlabel("x")
23 ylabel("y")
24 title('Cost Function for
   ↪ h_{\theta}(x) = \theta_0 +
   ↪ \theta_1 * x')
25 saveas(gcf, 'plot_cost_function.png')

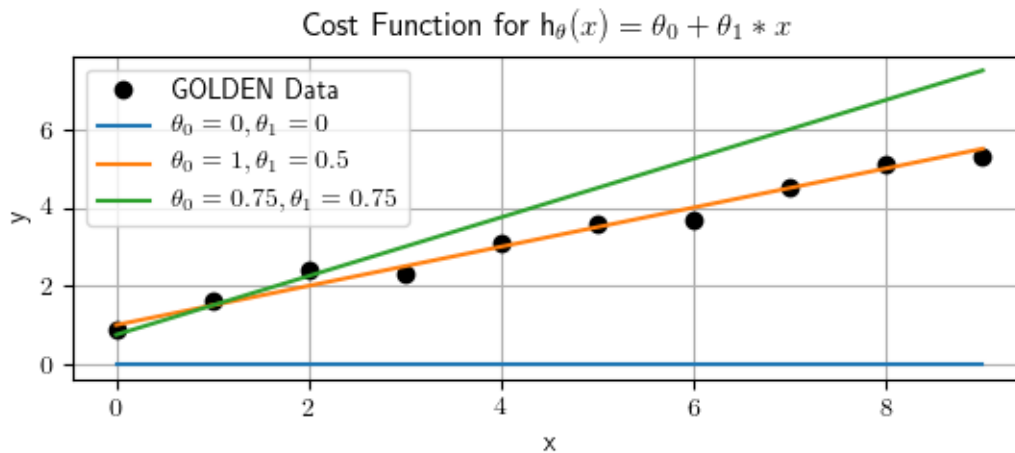
```

```

1  #!/usr/bin/env python3
2
3  import cost_function
4
5  import numpy as np
6  import matplotlib
7  matplotlib.rcParams['text.usetex'] =
    ↪ True
8  import matplotlib.pyplot as plt
9
10 Y_GOLDEN = [0.9, 1.6, 2.4, 2.3, 3.1,
    ↪ 3.6, 3.7, 4.5, 5.1, 5.3]
11 X = np.arange(0, 10, 1)
12
13 Y1 =
    ↪ cost_function.hypothesis_linear(0,
    ↪ 0, X)
14 Y2 =
    ↪ cost_function.hypothesis_linear(1,
    ↪ 0.5, X)
15 Y3 =
    ↪ cost_function.hypothesis_linear(0.75,
    ↪ 0.75, X)
16
17 fig = plt.figure()
18 ax1 = fig.add_subplot(211)
19 ax1.set_ylabel('y')
20 ax1.set_xlabel('x')
21 ax1.set_title(r'Cost Function for
    ↪  $h_{\theta}(x) = \theta_0 +$ 
    ↪  $\theta_1 * x$ ')
22
23 ax1.grid(True)
24 line0, = ax1.plot(X, Y_GOLDEN, 'ko',
    ↪ label="GOLDEN Data")
25 line1, = ax1.plot(X, Y1,
    ↪ label=r' $\theta_0 = 0, \theta_1$ 
    ↪  $= 0$ ')
26 line2, = ax1.plot(X, Y2,
    ↪ label=r' $\theta_0 = 1, \theta_1$ 
    ↪  $= 0.5$ ')
27 line3, = ax1.plot(X, Y3,
    ↪ label=r' $\theta_0 = 0.75,$ 
    ↪  $\theta_1 = 0.75$ ')
28 plt.legend()
29 # plt.show()
30 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 hypothesis 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 Algorithm is:

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

Repeat this until it converges! Make sure to only update all θ values once all calculations are done. We want to keep them constant while looping through everything.

Gradient Descent Implementation

Matlab

```

1  function [final_cost, final_theta0,
   ↪ final_theta1] =
   ↪ gradient_descent(theta, epsilon,
   ↪ golden, MAX_STEPS)
2
3  m = length(golden);
4  x = [0:1:m]';
5  x = [ones(m+1,1), x(:,1)];
6  theta0_range = [theta(1): epsilon:
   ↪ theta(1)+MAX_STEPS];
7  theta1_range = [theta(2): epsilon:
   ↪ theta(2)+MAX_STEPS];
8
9  final_cost = 100000000;
10 for z0 = 1:length(theta0_range)
11     for z1 = 1:length(theta1_range)
12         sum_diff_squared = 0;
13         cost = 0;
14         diff = 0;
15         diff_squared = 0;
16         theta_values =
           ↪ [theta0_range(z0),
           ↪ theta1_range(z1)];
17         for i = 1:length(golden)
18             diff = (theta0_range(z0) +
               ↪ x(i)*theta1_range(z1))
               ↪ - golden(i);
19             diff_squared = diff * diff;
20             sum_diff_squared =
               ↪ sum_diff_squared +
               ↪ diff_squared;
21         end
22         cost = (1/(2*m)) *
           ↪ sum_diff_squared;
23         if (cost(1) < final_cost)
24             printf("\n")
25             final_cost = cost(1)
26             final_theta0 =
               ↪ theta0_range(z0)
27             final_theta1 =
               ↪ theta1_range(z1)
28         endif
29     end
30 end
31 end

```

Python

```

1  #!/usr/bin/env python3
2
3  import numpy as np
4  import matplotlib.pyplot as plt
5
6
7  def hypothesis_linear(theta0=None,
8      ↪ theta1=None, x=None):
9      """
10     Calculate  $h_{\theta}(x^{(i)})$ 
11     """
12
13     return theta0 + x*theta1
14
15 def gradient_descent(theta0=None,
16     ↪ theta1=None, epsilon=None,
17     ↪ golden=None, MAX_STEPS=10):
18     """
19     Calculate the cost function
20      $J(\theta_0, \theta_1) =$ 
21      $\frac{1}{2m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)})^2$ 
22     """
23
24     print("Cost Function theta0={}"
25         ↪ theta1={} epsilon={}
26         ↪ golden={}".format(
27             theta0, theta1, epsilon,
28             ↪ golden))
29
30     # Number of training elements
31     m = len(golden)
32
33     x = np.arange(0, m, 1)
34     theta0_range = np.arange(theta0,
35         ↪ theta0+MAX_STEPS, epsilon)
36     theta1_range = np.arange(theta1,
37         ↪ theta1+MAX_STEPS, epsilon)
38
39     # Start this out as HUGE so we
40     ↪ should always find a lower
41     ↪ cost than this

```



```

30     final_cost = 10000000000
31
32     # Go through all possible values
33     ↪ for theta0 and theta1
34     # This is not efficient, this is
35     ↪ just trying
36     ↪ everything.....
37     for z0 in theta0_range:
38         for z1 in theta1_range:
39             # Clear out the per loop
40             ↪ values
41             sum_diff_squared = 0
42
43             # This is the
44             ↪ \sum_{i=1}^m loop
45             ↪ element
46             for i in range(m):
47                 # find the difference
48                 ↪ between the
49                 ↪ calculated value
50                 ↪ from
51                 ↪ hypothesis_linear
52                 ↪ and
53                 # the actual value
54                 ↪ stored in the
55                 ↪ golden array
56                 diff = hypothe-
57                 ↪ sis_linear(z0,
58                 ↪ z1, x[i]) -
59                 ↪ golden[i]
60                 # Square and sum to
61                 ↪ avoid some being
62                 ↪ too high and some
63                 ↪ being too low.
64                 diff_squared = diff *
65                 ↪ diff
66                 sum_diff_squared =
67                 ↪ sum_diff_squared
68                 ↪ + diff_squared
69                 cost = 1/(2*m) *
70                 ↪ sum_diff_squared
71                 # print ("Z0 = {:02f} Z1
72                 ↪ = {:02f} Cost =
73                 ↪ {:02f}".format(z0,
74                 ↪ z1, cost))
75
76             # Find the lowest
77             ↪ possible cost
78             if (cost < final_cost):
79                 final_cost = cost
80                 final_theta0 = z0
81                 final_theta1 = z1

```

The cost function for a linear function is always bowl shaped on the 3d plot with a single global optima.

2.1.6 Multivariate Linear Regression

We want to predict the output y . We will have multiple features $x_0, x_1, x_2 \dots x_n$ where n is the number of features. $x^{(i)}$ is the i^{th} training example. $x_j^{(i)}$ is the i^{th} training example for feature j m is the number of training examples.

Multivariate Linear Hypothesis is

$$h_{\theta}(x) = \theta_0 + \theta_1 * x_1 + \theta_2 * x_2 \dots + \theta_n * x_n \quad (2.5)$$

if we define $x_0 = 1$ we can redefine the equation as

$$h_{\theta}(x) = \theta_0 * x_0 + \theta_1 * x_1 + \theta_2 * x_2 \dots + \theta_n * x_n \quad (2.6)$$

Which can be expressed as vectors $\theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \vdots \\ \theta_n \end{bmatrix}$ and $x = \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_n \end{bmatrix}$

Which when put together in a linear algebra equation takes the form of

$$h_{\theta}(x) = \Theta^T x \quad (2.7)$$

In multivariate linear regression just multiple features in our linear regression to predict y .

2.1.7 Gradient Descent for Multiple Variables

hypothesis: $h_{\theta}(x) = \theta_0 * x_0 + \theta_1 * x_1 + \theta_2 * x_2 \dots + \theta_n * x_n$

parameters: $\theta_0, \theta_1 \dots \theta_n$

cost = $J(\theta_0, \theta_1 \dots \theta_n) = \frac{1}{2m} \sum_{i=1}^m (h_{\theta}(x^i) - y(i))^2$

$J(\Theta)$ usually means the vector Θ

The process is to repeat until convergence. We converge once the change in cost gets too small to really matter. The general form for more than 1 feature is:

$$\theta_j := \theta_j - \alpha * \frac{1}{m} \sum_{i=1}^m (h_{\theta}(x^i) - y^i) * x_j^i \quad (2.8)$$

This works out as:

$$\begin{aligned}
 \theta_0 &:= \theta_0 - \alpha * \frac{1}{m} \sum_{i=1}^m (h_{\theta}(x^i) - y^i) * x_0^i \\
 \theta_1 &:= \theta_1 - \alpha * \frac{1}{m} \sum_{i=1}^m (h_{\theta}(x^i) - y^i) * x_1^i \\
 \theta_2 &:= \theta_2 - \alpha * \frac{1}{m} \sum_{i=1}^m (h_{\theta}(x^i) - y^i) * x_2^i \\
 &\vdots \\
 \theta_n &:= \theta_n - \alpha * \frac{1}{m} \sum_{i=1}^m (h_{\theta}(x^i) - y^i) * x_n^i
 \end{aligned} \tag{2.9}$$

Example

Housing Data

Size	Bedrooms	Floors	Age	Price
2104	5	1	45	460
1416	3	2	40	232
1534	3	2	30	315
852	2	1	36	178

$$x^2 = \begin{bmatrix} 1416 \\ 3 \\ 2 \\ 40 \end{bmatrix}$$

$$x_3^2 = 2$$

2.1.8 Feature Scaling

Idea: A problem with multiple features, make sure all features are on similar scales, Gradient Descent will converge more quickly.

EX:

$$x_1 = 0 - 2000 \text{ sq feet}$$

$$x_2 = 1-5 \text{ bedrooms}$$

These are very different ranges! Contours of cost function take on a very skewed elliptical shape. Can take a long time for gradients to find global minimum.

Scale features:

$$x_1 = \frac{\text{size in feet}^2}{2000}$$

$$x_2 = \frac{\text{number of bedrooms}}{5}$$

This puts everyone on a 0-1 scale so they are equally weighted. Gradient descent should work quicker.

Generally want all features in a $-1 \leq x_i \leq 1$. This is not a hard rule. Should just be similar in scale.

Mean Normalization

Replace x_i with $x_i - \mu_i$ to make features have approximately 0 mean. Do **NOT** apply to x_0

EX:

$$x_1 = \frac{\text{size} - 1000}{2000}$$

$$x_2 = \frac{\text{bedrooms} - 2}{5}$$

This puts x_1 and x_2 in between -0.5 and +0.5

$x_i = \frac{x_i - \mu_i}{s_i}$ Where μ_i is the average value and s_i is the range or the standard deviation.

2.1.9 Gradient Descent Learning Rate

Plotting $J(\theta)$ vs number of iterations, as gradient descent runs, it should be decreasing on each iteration. Once plot levels off, we have converged or are close enough. The number of iterations is hard to predict could be 30 or 3,000,000 or more. Automatic convergence tests are helpful. If cost function decreases by some small ϵ like 10^{-3} in an iteration. Finding this threshold is hard.

If $J(\theta)$ is increasing, gradient descent is not working and you should try a smaller α

If $J(\theta)$ is bouncing up and down, it is not working and you should try a smaller α

If α is too small, gradient descent will converge very slowly.

To choose α try 0.001, 0.01, 0.1, 1.....

Usually try 0.001, 0.003, 0.01, 0.03, 0.1, 0.3, 1 ...

2.1.10 Features and Polynomial Regression

Different learning algorithms based on different features.

Polynomial regression will let you fit non-linear functions.

Depends on insight in problem.

Polynomial Regress

If a linear fit does not appear to match data, you can use polynomials. A quadratic function would look like this: $h_{\theta}(x) = \theta_0 + \theta_1 * x + \theta_2 * x^2$ A cubic function would look like this: $h_{\theta}(x) = \theta_0 + \theta_1 * x + \theta_2 * x^2 + \theta_3 * x^3$

2.1.11 Normal Equation

Gradient descent gives one way of minimizing J. Let's discuss a second way of doing so, this time performing the minimization explicitly and without resorting to an iterative algorithm. In the "Normal Equation" method, we will minimize J by explicitly taking its derivatives with respect to the θ s, and setting them to zero. This allows us to find the optimum theta without iteration. The normal equation formula is given below:

$$\Theta = (X^T X)^{-1} X^T y \quad (2.10)$$