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

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

Python

```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

```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

```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

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

% Transpose A
A_trans = A'
```

Python

```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

```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

```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.

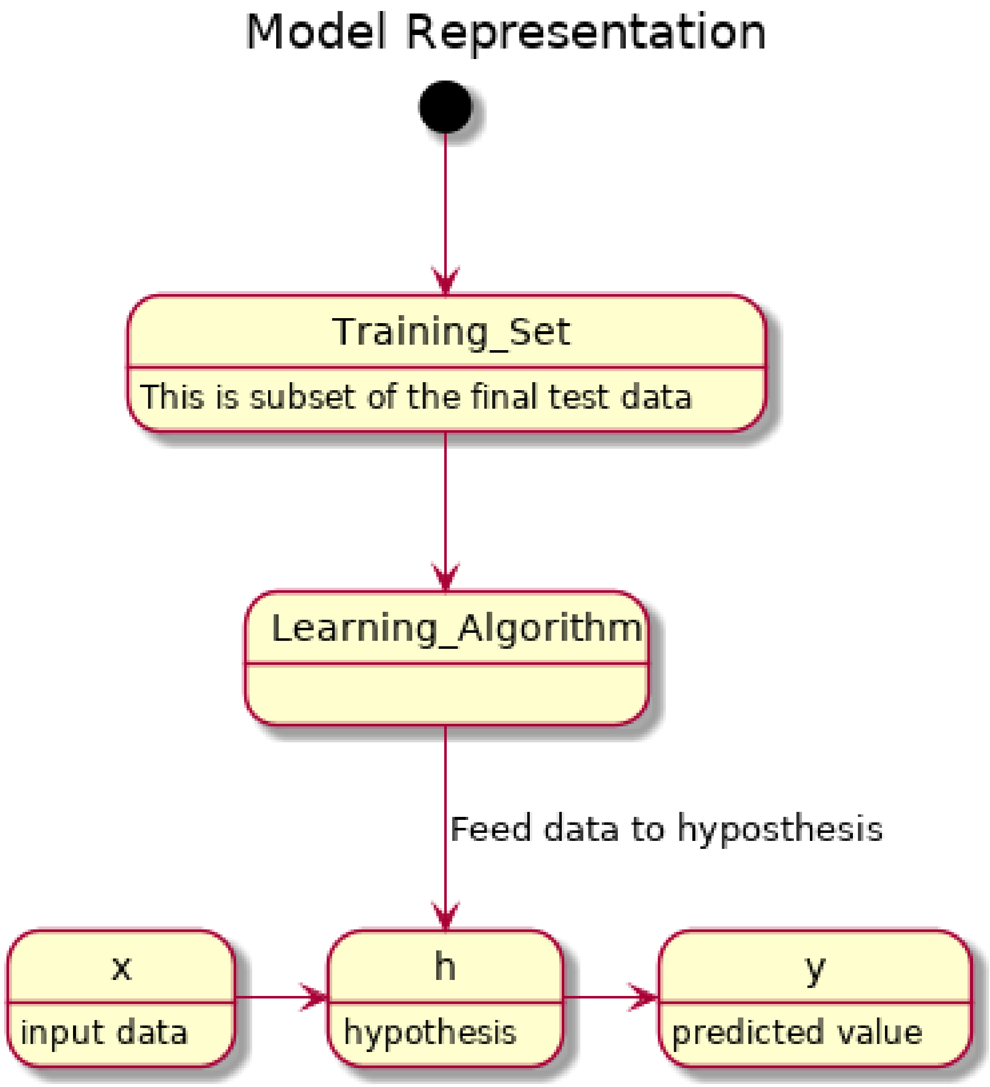
2.1.3 Model

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.

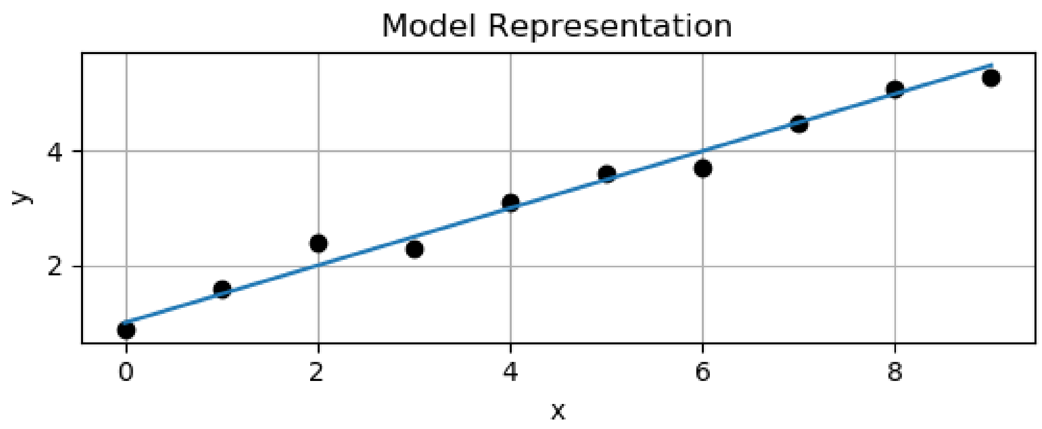
To establish notation for future use, we'll use $x^{(i)}$ to denote the **"input"** variables (living area in this example), also called input features, and $y^{(i)}$ to denote the **"output"** or target variable that we are trying to predict (price). A pair $(x^{(i)}, y^{(i)})$ is called a **training 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

```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

```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, 5.6]'
theta = zeros(2,1); % 2 rows x 1
 ↪  column of 0

m = length(Y_GOLDEN);
x = [0:1:m]';
x = [ones(m,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.

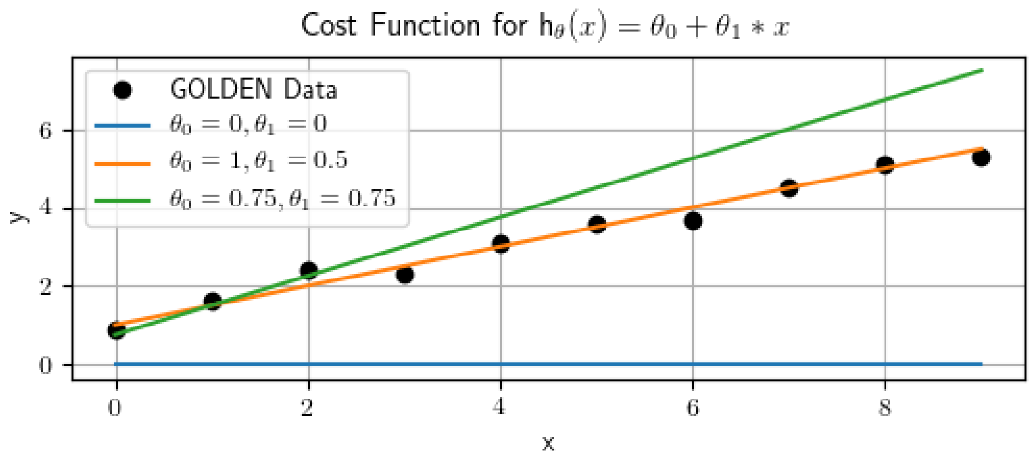
Matlab

Python

```
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
#! /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! 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

```matlab
function [final_cost, final_theta0, final_theta1] = gradient_descent(theta,epsilon, golden, MAX_STEPS)

m = length(golden);
x = [0:1:m]';
x = [ones(m+1,1), x(:,1)];
theta0_range = [theta(1): epsilon: theta(1)+MAX_STEPS];
theta1_range = [theta(2): epsilon: theta(2)+MAX_STEPS];

final_cost = 10000000;
for z0 = 1:length(theta0_range)
    for z1 = 1:length(theta1_range)
        sum_diff_squared = 0;
        cost = 0;
        diff = 0;
        diff_squarded = 0;
        theta_values = [theta0_range(z0), theta1_range(z1)];
        for i =1:length(golden)
          diff = (theta0_range(z0) + x(i)*theta1_range(z1)) - golden(i);
          diff_squared = diff * diff;
          sum_diff_squared = sum_diff_squared + diff_squared;
        end
        cost = (1/(2*m)) * sum_diff_squared;
        if (cost(1) < final_cost)
            printf("\n")
            final_cost = cost(1)
            final_theta0 = theta0_range(z0)
            final_theta1 = theta1_range(z1)
            endif
        end
    end
end
```

Python

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

import numpy as np
import matplotlib.pyplot as plt


def hypothesis_linear(theta0=None,
    theta1=None, x=None):
    """
    Calculate h_{\theta} (x^{(i)}}
    """
    return theta0 + x*theta1


def gradient_descent(theta0=None,
    theta1=None, epsilon=None,
    golden=None, MAX_STEPS=10):
    """
    Calculate the cost function
    J(\theta_{0}, \theta_{1}) =
    \frac{1}{2m} \sum_{i=1}^{m}
    (h_{\theta}(x^{i})-y^{i})^2
    """


    print("Cost Function theta0={}
        theta1={} epsilon={}
        golden={}".format(
        theta0, theta1, epsilon,
            golden))
    # Number of training elements
    m = len(golden)

    x = np.arange(0, m, 1)
    theta0_range = np.arange(theta0,
        theta0+MAX_STEPS, epsilon)
    theta1_range = np.arange(theta1,
        theta1+MAX_STEPS, epsilon)

    # Start this out as HUGE so we
        should always find a lower
        cost than this
    final_cost = 10000000000

    # Go through all possible values
     ↪  for theta0 and theta1
    # This is not efficient, this is
     ↪  just trying
     ↪  everything........
    for z0 in theta0_range:
        for z1 in theta1_range:
            # Clear out the per loop
             ↪  values
            sum_diff_squared = 0

            # This is the
             ↪  \sum_{i=1}^{m} loop
             ↪  element
            for i in range(m):
                # find the difference
                 ↪  between the
                 ↪  calculated value
                 ↪  from
                 ↪  hypothesis_linear
                 ↪  and
                # the actual value
                 ↪  stored in the
                 ↪  golden array
                diff = hypothe-
                 ↪  sis_linear(z0,
                 ↪  z1, x[i]) -
                 ↪  golden[i]
                # Square and sum to
                 ↪  avoid some being
                 ↪  too high and some
                 ↪  being too low.
                diff_squared = diff *
                 ↪  diff
                sum_diff_squared =
                 ↪  sum_diff_squared
                 ↪  + diff_squared
            cost = 1/(2*m) *
             ↪  sum_diff_squared
            # print ("Z0 = {:02f} Z1
             ↪  = {:02f} Cost =
             ↪  {:02f}".format(z0,
             ↪  z1, cost))

            # Find the lowest
             ↪  possible cost
            if (cost < final_c
                final_cost = cost
                final_theta0 = z0
                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 $x_0, x_1, x_2 ... x_n$ 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

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

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

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

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

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

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

2.1.7 Gradient Descent for Multiple Variables

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

paramters: $\theta_0, \theta_1 ... \theta_n$

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

$J(\Theta)$ 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:

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

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 sq feet
$x_2 = 1$-5 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:
$x_1 = \frac{size in feet^2}{2000}$
$x_2 = \frac{number of bedrooms}{5}$
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:
$x_1 = \frac{size - 1000}{2000}$

$x_2 = \frac{bedrooms-2}{5}$
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_1 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

Normal equation is a method to solve for θ analytically instead of iteratively. One step and it's done, no process or convergence.
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 θjs, and setting them to zero. This allows us to find the optimum theta without iteration. The normal equation formula is given below:

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

EX: For 1-D

$$J(\theta) = a\theta^2 + b\theta + c\frac{d}{d\theta}J(\theta) = 2a\theta + b = 0 thensolvefor\theta$$

EX: For more than 1-D Take the partial for J for each θ , set to 0 and solve

$$J(\theta_0....\theta_m) = \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^i)^2 \frac{\partial}{\partial\theta_j} = ... = 0 for each j solve for all the \theta values \tag{2.11}$$

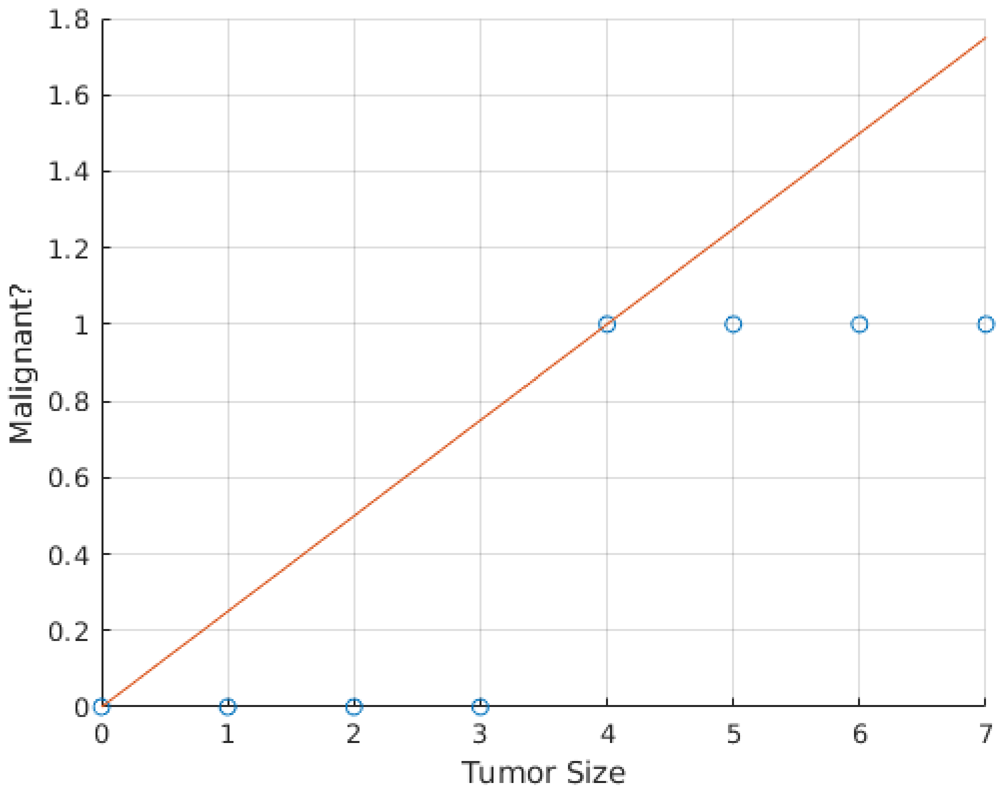
3 Classification

3.1 Classification

To attempt classification, one method is to use linear regression and map all predictions greater than 0.5 as a 1 and all less than 0.5 as a 0. However, this method doesn't work well because classification is not actually a linear function.

The classification problem is just like the regression problem, except that the values we now want to predict take on only a small number of discrete values. For now, we will focus on the binary classification problem in which y can take on only two values, 0 and 1. (Most of what we say here will also generalize to the multiple-class case.) For instance, if we are trying to build a spam classifier for email, then $x^{(i)}$ may be some features of a piece of email, and y may be 1 if it is a piece of spam mail, and 0 otherwise. Hence, $y \in 0, 1$. 0 is also called the negative class, and 1 the positive class, and they are sometimes also denoted by the symbols "-" and "+." Given $x^{(i)}$ the corresponding $y^{(i)}$ is also called the label for the training example.

Classification problems:
Email: Spam/Not Spam
Online Transaction: Valid/Fraud
Tumor: Malignant/Benign
$y \in 0, 1$ where 0 is the negative class and 1 is the positive class.



For a few points that split up like this easily, the linear fit is not good.

Threshold Classifier output $h_\theta(x)$ at 0.5
if $h_\theta(x) \geq 0.5$ y=1
if $h_\theta(x) < 0.5$ y= 0
This is not perfect, still can mis-classify somethings. It might be a reasonable solution. An outlier can skew the line. This can move the thresold to the wrong location. Linear regression is not a great idea for classification.
Logistic Regression outputs are always between 0 and 1. The name includes regression but it is a classification algorithm

$$0 \leq h_\theta(x) \leq 1$$

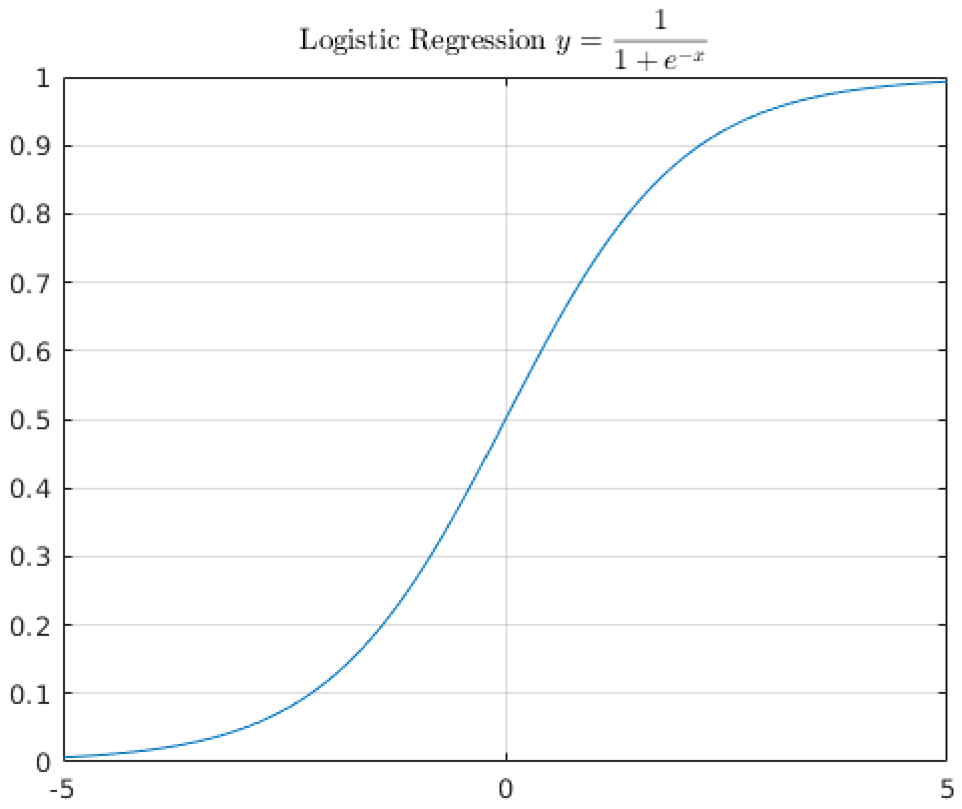
Hypothesis Representation

We could approach the classification problem ignoring the fact that y is discrete-valued, and use our old linear regression algorithm to try to predict y given x. However, it is easy to construct examples where this method performs very poorly. Intuitively, it also doesn't make sense for $h_\theta(x)$ to take values larger than 1 or smaller than 0 when we know that $y \in 0, 1$. To fix this, let's change the form for our hypotheses $h_\theta(x)$ satisfy $0 \leq h_\theta(x) \leq 1$. This is accomplished by plugging $\theta^T x$ into the Logistic Function.

Our new form uses the ”Sigmoid Function,” also called the ”Logistic Function”:

$$\begin{aligned} h_\theta(x) &= g(\theta^T x) \\ z &= \theta^T x \\ g(z) &= \frac{1}{1+e^{-z}} \\ h_\theta(x) &= \frac{1}{1+e^{-\theta^T x}} \end{aligned} \tag{3.1}$$

$g(z)$ is the Sigmoid or Logistic Function. This is where the name Logistic Regression comes from.



The function g(z), shown here, maps any real number to the (0, 1) interval, making it useful for transforming an arbitrary-valued function into a function better suited for classification.

$h_\theta(x)$ will give us the probability that our output is 1. For example, $h_\theta(x) = 0.7$ gives us a probability of 70% that our output is 1. Our probability that our prediction is 0 is just the complement of our probability that it is 1 (e.g. if probability that it is 1 is 70%, then the probability that it is 0 is 30

$$\begin{aligned} h_\theta(x) = P(y=1||x;\theta) \quad &= 1 - P(y=0|x;\theta) \\ P(y=0||x;\theta) + P(y=1|x;\theta) = 1 \end{aligned} \tag{3.2}$$

Fit parameters θ to our data.
$h_\theta(x)$ = estimate probability that y=1 on input x

Example:
$x = \begin{bmatrix} x_0 \\ x_1 \end{bmatrix} = \begin{bmatrix} 1 \\ tumorsize \end{bmatrix}$
If $h_\theta(x) = 0.7$, then it is a 70% chance of being a tumor.

Decision Boundary

In order to get our discrete 0 or 1 classification, we can translate the output of the hypothesis function as follows:

$$\begin{aligned} h_\theta(x) \geq 0.5 \rightarrow y = 1 \\ h_\theta(x) < 0.5 \rightarrow y = 0 \end{aligned} \tag{3.3}$$

The way our logistic function g behaves is that when its input is greater than or equal to zero, its output is greater than or equal to 0.5:

$$\begin{aligned} & g(z) \geq 0.5 \\ & when\ z \geq 0 \end{aligned} \tag{3.4}$$

Remember.

$$\begin{aligned} z = 0, e^0 = 1 \Rightarrow g(z) = 1/2 \\ z \rightarrow \infty, e^{-\infty} \rightarrow 0 \Rightarrow g(z) = 1 \\ z \rightarrow -\infty, e^{\infty} \rightarrow \infty \Rightarrow g(z) = 0 \end{aligned} \tag{3.5}$$

So if our input to g is $\theta^T X$ then that means:

$$\begin{aligned} & h_\theta(x) = g(\theta^T x) \geq 0.5 \\ & when\ \theta^T x \geq 0 \end{aligned} \tag{3.6}$$

From these statements we can now say:

$$\begin{aligned} \theta^T x \geq 0 \Rightarrow y = 1 \\ \theta^T x < 0 \Rightarrow y = 0 \end{aligned} \tag{3.7}$$

The **decision boundary** is the line that separates the area where y = 0 and where y = 1. It is created by our hypothesis function.
Example:

$$\begin{aligned} & \theta = \begin{bmatrix} 5 \\ -1 \\ 0 \end{bmatrix} \\ & y = 1\ if\ 5 + (-1)x_1 + 0x_2 \geq 0 \\ & 5 - x_1 \geq 0 \\ & -x_1 \geq -5 \\ & x_1 \leq 5 \end{aligned} \tag{3.8}$$

In this case, our decision boundary is a straight vertical line placed on the graph where x_1 =5, and everything to the left of that denotes y = 1, while everything to the right denotes y = 0.

Again, the input to the sigmoid function g(z) (e.g. $\theta^T X$ doesn't need to be linear, and could be a function that describes a circle (e.g. z = $\theta_0 + \theta_1 x_1^2 + \theta_2 x_2^2$ or any shape to fit our data.
Example:
If $h_\theta(x) = g(\theta_0 + \theta_1 x_1 + \theta_2 x_2)$ then predict y =1.
For given $\theta = \begin{bmatrix} -3 \\ 1 \\ 1 \end{bmatrix}$
then $-3 + x_1 + x_2 \geq 0$.
This means $\theta^T x = -3 + x_1 + x_2 + \theta_3 x_1^2 + \theta_4 x_2^2)$

Non-Linear Decision Boundaries $h_\theta(x) = g(\theta_0 + \theta_1 x_1 + \theta_2 x_2$
$\theta = \begin{bmatrix} -1 \\ 0 \\ 0 \\ 1 \\ 1 \end{bmatrix}$
This will predict y =1 if $-1 + x_1^2 + x_2^2 \geq 0$
Decision Boundary is property not of training set but of the hypothesis and parameters.

You can use even higher order polynomials leading to very complex decision boundaries.

3.1.1 Logistic Regression Model

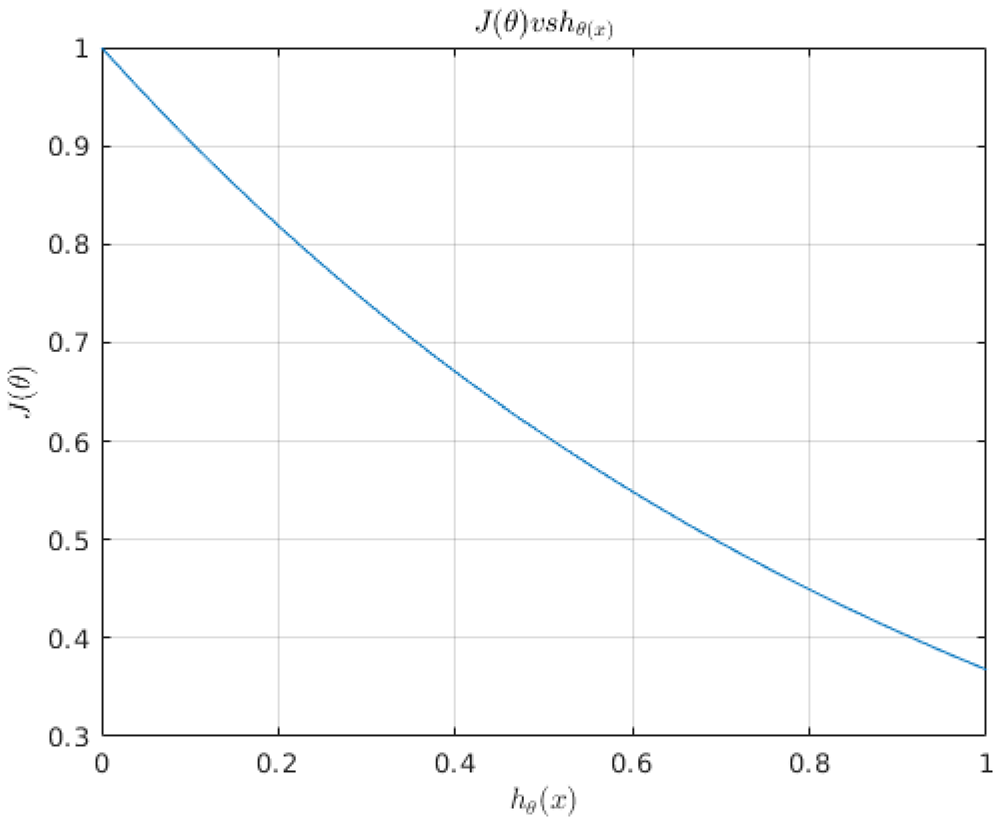
Cost Function

We cannot use the same cost function that we use for linear regression because the Logistic Function will cause the output to be wavy, causing many local optima. In other words, it will not be a convex function.

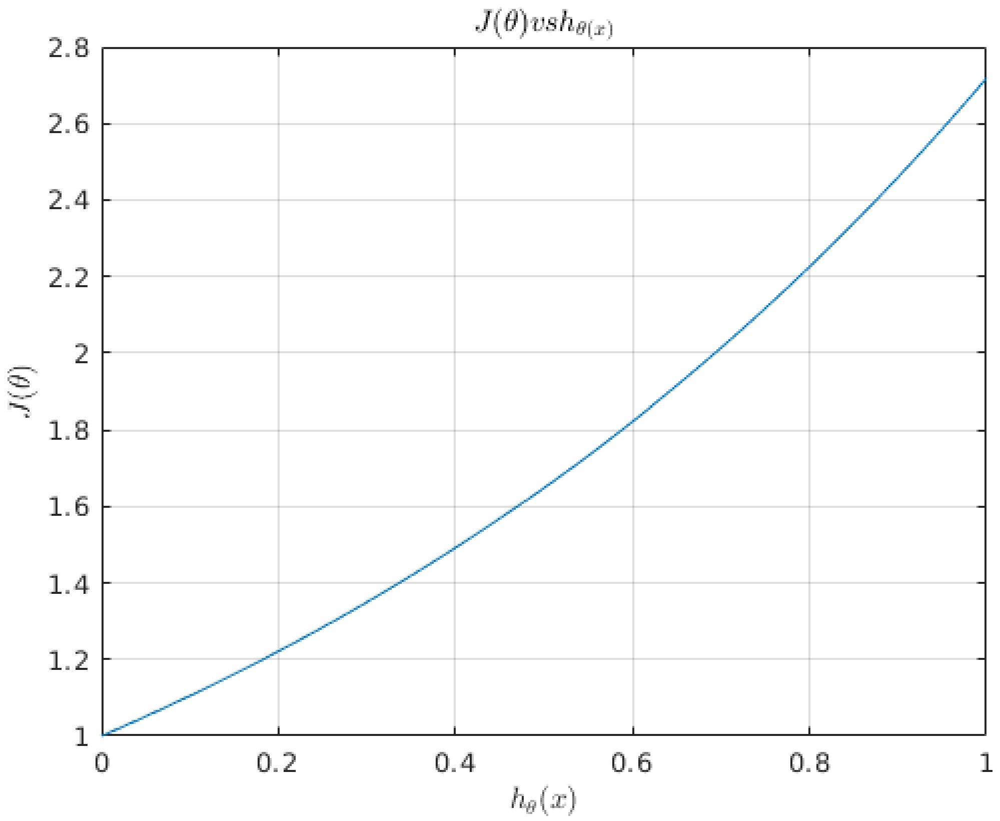
Instead, our cost function for logistic regression looks like:

$$\begin{aligned} J(\theta) &= \frac{1}{m}\sum_{i=1}^{m} \mathrm{Cost}(h_\theta(x^{(i)}), y^{(i)}) \\ \mathrm{Cost}(h_\theta(x), y) &= -\log(h_\theta(x)) \quad \text{if y = 1} \\ \mathrm{Cost}(h_\theta(x), y) &= -\log(1 - h_\theta(x)) \quad \text{if y = 0} \end{aligned} \tag{3.9}$$

When y = 1, we get the following plot for $J(\theta) vs h_\theta(x)$



When y = 0, we get the following plot for $J(\theta) vs h_\theta(x)$



$$\text{Cost}(h_\theta(x), y) = 0 \text{ if } h_\theta(x) = y \quad \text{Cost}(h_\theta(x), y) \to \infty \text{ if } y = 0 \text{ and } h_\theta(x) \to 1 \text{Cost}(h_\theta(x), y) \to \infty \text{ if } y = 1 \text{ and } h_\theta(x \tag{3.10}$$

If our correct answer 'y' is 0, then the cost function will be 0 if our hypothesis function also outputs 0. If our hypothesis approaches 1, then the cost function will approach infinity.

If our correct answer 'y' is 1, then the cost function will be 0 if our hypothesis function outputs 1. If our hypothesis approaches 0, then the cost function will approach infinity.

Note that writing the cost function in this way guarantees that $J(\theta)$ is convex for logistic regression.

Problem:
How to choose parameters θ?
Training Set: $(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), (x^{(3)}, y^{(3)}) \ldots (x^{(m)}, y^{(m)})$
m example $x \in \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_n \end{bmatrix}$
With $x_0 = 1$ and $y \in 0, 1$ this mean y must be 0 or 1.
$h_\theta(x) = \frac{1}{1+e^{\theta^T X}}$

Linear Regression Cost:

$$J(\theta) = \frac{1}{m} \sum_{i=1}^{m} \frac{1}{2}(h_\theta(x^{(i)}) - y^{(i)})^2 \tag{3.11}$$

Redefine this as:

$$Cost(h_{theta}(x^{(i)}), y^{(i)}) = \frac{1}{2}(h_\theta(x^{(i)}) - y^{(i)})^2 \tag{3.12}$$

This doesn't work right for logistic regression. It would be non-convex. Create many local optima. Gradient descent is not guaranteed to find global minima.
Logistic Regression Cost Function

$$Cost(h_{theta}(x), y) = \begin{cases} -log(h_\theta(x)) & if y = 1 \\ -log(1 - h_\theta(x)) & if y = 0 \end{cases} \tag{3.13}$$

Cost =0 if y=1 and $h_\theta(x)$=1 but as $h_\theta(x) \to 0$, cost $\to \infty$
This captures the idea that if $h_\theta(x)$=0 but y =1, we will penalize the algorithm by a large cost.

Simplified Cost Function and Gradient Descent

We can compress our cost function's two conditional cases into one case:
$Cost(h_\theta(x), y) = -ylog(h_\theta(x)) - (1 - y)log(1 - h_\theta(x))$

Notice that when y is equal to 1, then the second term $(1-y)\log(1-h_\theta(x))$ will be zero and will not affect the result. If y is equal to 0, then the first term $-y\log(h_\theta(x))$ will be zero and will not affect the result.
We can fully write out our entire cost function as follows:

$$\mathrm{Cost}(h_\theta(x), y) = -y\ \log(h_\theta(x)) - (1-y)\log(1-h_\theta(x)) \tag{3.14}$$

A vectorized implementation is:

$$\begin{aligned} h &= g(X\theta) \\ J(\theta) &= \frac{1}{m}\cdot\left(-y^T\log(h) - (1-y)^T\log(1-h)\right) \end{aligned} \tag{3.15}$$

Remember that the general form of gradient descent is:

$$\begin{aligned} &Repeat\ \{ \\ &\theta_j := \theta_j - \alpha\frac{\partial}{\partial\theta_j}J(\theta) \\ &\} \end{aligned} \tag{3.16}$$

We can work out the derivative part using calculus to get:

$$\begin{aligned} &Repeat\ \{ \\ &\theta_j := \theta_j - \frac{\alpha}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)} \\ &\} \end{aligned} \tag{3.17}$$

Notice that this algorithm is identical to the one we used in linear regression. We still have to simultaneously update all values in theta.
A vectorized implementation is:

$$\theta := \theta - \frac{\alpha}{m}X^T(g(X\theta) - \overrightarrow{y}) \tag{3.18}$$

Advanced Optimization

"Conjugate gradient", "BFGS", and "L-BFGS" are more sophisticated, faster ways to optimize θ that can be used instead of gradient descent. We suggest that you should not write these more sophisticated algorithms yourself (unless you are an expert in numerical computing) but use the libraries instead, as they're already tested and highly optimized. Octave provides them.

We first need to provide a function that evaluates the following two functions for a given input value θ:

$J(\theta)$
$\frac{\partial}{\partial\theta_j}J(\theta)$

```
function [jVal, gradient] = costFunction(theta)
  jVal = [...code to compute J(theta)...];
  gradient = [...code to compute derivative of J(theta)...];
end
```

Then we can use octave's ”fminunc()” optimization algorithm along with the ”optimset()” function that creates an object containing the options we want to send to ”fminunc()”.

```
options = optimset('GradObj', 'on', 'MaxIter', 100);
initialTheta = zeros(2,1);
   [optTheta, functionVal, exitFlag] = fminunc(@costFunction, initialTheta,
    ↪  options);
```

We give to the function ”fminunc()” our cost function, our initial vector of theta values, and the ”options” object that we created beforehand.

3.1.2 Multiclass Classification

Example:
Auto sort email into folders: Work, Friends, Family, Other.
Medical Diagnosis: Not Ill, Cold, Flu
Weather: Sunny, Cloudy, Rain, Snow

Y can have a small number of discrete values.

If we have 3 classes, go through all possible combinations of 1 vs rest
$h_\theta^{(1)}x$ = class 1 vs all others, learns to detect class 1
$h_\theta^{(2)}x$ = class 2 vs all others, learns to detect class 2
$h_\theta^{(3)}x$ = class 3 vs all others, learns to detect class 3
$h_\theta^{(i)}x = P(y = i|x;\theta)$ for i = 1,2,3
This is about finding the probability of a particular class.
Pick the best i

Now we will approach the classification of data when we have more than two categories. Instead of y = 0,1 we will expand our definition so that y = 0,1...n.

Since y = 0,1...n, we divide our problem into n+1 (+1 because the index starts at 0) binary classification problems; in each one, we predict the probability that 'y' is a member of one of our classes.

We are basically choosing one class and then lumping all the others into a single second class. We do this repeatedly, applying binary logistic regression to each case, and then use the hypothesis that returned the highest value as our prediction.

Train a logistic regression classifier $h_\theta(x)$ for each class to predict the probability that y = i. To make a prediction on a new x, pick the class that maximizes $h_\theta(x)$

One Vs All

3.1.3 Overfitting

Cost Function

Regularized Linear Regression

Regularized Logistic Regression

4 Neural Nets

4.1 Motivation

4.1.1 Non-Linear Hypothesis

Neural nets fell out of favor for a while, but it is currently the state of the art technique for many problems.
Needed for non-linear problems. With many features the previous systems (linear/sigmoid) would need too many terms and would become a mess and may not work. For n = 100 features, if you only consider the first order terms you get about 5000 elements to your equation. If you consider second order also, it gets out to about 170,000. This is getting to large.

Computer vision is hard. You and I see a car, the computer sees a matrix of numbers, pixel values. For a 50x50 pixel image, you have n=2500 (7500 is using RGB). This leads to about 3million features. Simple logistic with quadratic equations can not handle something this large.

4.1.2 Neurons and the Brain

Original motivation was a machine that can mimic the brain. Widely used in 80s and 90s but waned in the late 90s. Major resurgence when computers became big and fast enough to handle this.

The brain has ”one learning algorithm”. The brain can be re-wired and restore abilities. Same piece of brain tissue and can sight or sound or touch. Blind people can develop a sonar like skill.

4.2 Neural Networks

4.2.1 Model Representation I

Let's examine how we will represent a hypothesis function using neural networks. At a very simple level, neurons are basically computational units that take inputs (dendrites) as electrical inputs (called ”spikes”) that are channeled to outputs (axons). In our model, our dendrites are like the input features $x_1...x_n$, and the output is the result of our hypothesis function. In this model our x_0 input node is sometimes called the ”bias unit.” It is always equal to 1. In neural networks, we use the same logistic function as in classification, $\frac{1}{1+e^{-\theta^T x}}$, yet we sometimes call it a sigmoid (logistic) activation function. In this situation, our ”theta” parameters are sometimes called ”weights”.

$$\begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_n \end{bmatrix} \rightarrow [\,] \rightarrow h_\theta(x) \tag{4.1}$$

Our input nodes (layer 1), also known as the "input layer", go into another node (layer 2), which finally outputs the hypothesis function, known as the "output layer".

We can have intermediate layers of nodes between the input and output layers called the "hidden layers."

In this example, we label these intermediate or "hidden" layer nodes $a_0^{(2)} ... a_n^{(2)}$ and call them "activation units."

$$\begin{aligned} &a_i^{(j)} = \text{"activation" of unit i in layer j} \\ &\Theta^{(j)} = \text{matrix of weights controlling function mapping from layer j to layer j+1} \end{aligned} \tag{4.2}$$

If we had one hidden layer, it would look like:

$$\begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} \rightarrow \begin{bmatrix} a_1^(2) \\ a_2^(2) \\ a_3^(2) \end{bmatrix} \rightarrow h_\theta(x) \tag{4.3}$$

The values for each of the "activation" nodes is obtained as follows. Remeber g is the sigmoid function.

$$\begin{aligned} a_1^{(2)} &= g(\Theta_{10}^{(1)} x_0 + \Theta_{11}^{(1)} x_1 + \Theta_{12}^{(1)} x_2 + \Theta_{13}^{(1)} x_3) \\ a_2^{(2)} &= g(\Theta_{20}^{(1)} x_0 + \Theta_{21}^{(1)} x_1 + \Theta_{22}^{(1)} x_2 + \Theta_{23}^{(1)} x_3) \\ a_3^{(2)} &= g(\Theta_{30}^{(1)} x_0 + \Theta_{31}^{(1)} x_1 + \Theta_{32}^{(1)} x_2 + \Theta_{33}^{(1)} x_3) \\ h_\Theta(x) = a_1^{(3)} &= g(\Theta_{10}^{(2)} a_0^{(2)} + \Theta_{11}^{(2)} a_1^{(2)} + \Theta_{12}^{(2)} a_2^{(2)} + \Theta_{13}^{(2)} a_3^{(2)}) \end{aligned} \tag{4.4}$$

This is saying that we compute our activation nodes by using a 3×4 matrix of parameters. We apply each row of the parameters to our inputs to obtain the value for one activation node. Our hypothesis output is the logistic function applied to the sum of the values of our activation nodes, which have been multiplied by yet another parameter matrix $\Theta^{(2)}$ containing the weights for our second layer of nodes.

Each layer gets its own matrix of weights, $\Theta^{(j)}$.

The dimensions of these matrices of weights is determined as follows:
If network has s_j units in layer j and $s_j + 1$ units in layer j+1, then $\Theta^{(j)}$ will be of dimension $s_{j+1}(s_j + 1)$.

The +1 comes from the addition in $\Theta^{(j)}$ of the "bias nodes," x_0 and $\Theta_0^{(j)}$. In other words the output nodes will not include the bias nodes while the inputs will. The following image summarizes our model representation:

4.2.2 Model Representation II

To re-iterate, the following is an example of a neural network:

$$\begin{aligned} a_1^{(2)} &= g(\Theta_{10}^{(1)}x_0 + \Theta_{11}^{(1)}x_1 + \Theta_{12}^{(1)}x_2 + \Theta_{13}^{(1)}x_3) \\ a_2^{(2)} &= g(\Theta_{20}^{(1)}x_0 + \Theta_{21}^{(1)}x_1 + \Theta_{22}^{(1)}x_2 + \Theta_{23}^{(1)}x_3) \\ a_3^{(2)} &= g(\Theta_{30}^{(1)}x_0 + \Theta_{31}^{(1)}x_1 + \Theta_{32}^{(1)}x_2 + \Theta_{33}^{(1)}x_3) \\ h_\Theta(x) = a_1^{(3)} &= g(\Theta_{10}^{(2)}a_0^{(2)} + \Theta_{11}^{(2)}a_1^{(2)} + \Theta_{12}^{(2)}a_2^{(2)} + \Theta_{13}^{(2)}a_3^{(2)}) \end{aligned} \tag{4.5}$$

In this section we'll do a vectorized implementation of the above functions. We're going to define a new variable $z_k^{(j)}$ that encompasses the parameters inside our g function. In our previous example if we replaced by the variable z for all the parameters we would get:

$$\begin{aligned} a_1^{(2)} &= g(z_1^{(2)}) \\ a_2^{(2)} &= g(z_2^{(2)}) \\ a_3^{(2)} &= g(z_3^{(2)}) \end{aligned} \tag{4.6}$$

The vector representation of x and z^j is:

$$x = \begin{bmatrix} x_0 \\ x_1 \\ \cdots \\ x_n \end{bmatrix} z^{(j)} = \begin{bmatrix} z_1^{(j)} \\ z_2^{(j)} \\ \cdots \\ z_n^{(j)} \end{bmatrix} \tag{4.7}$$

Setting x = $a^{(1)}$ we can rewrite the equation as: $z^{(j)} = \Theta^{(j-1)}a^{(j-1)}$
We are multiplying our matrix $\theta(j-1)$ with dimensions $s_j\times$ (n+1) (where s_j is the number of our activation nodes) by our vector $a^{(j-1)}$ with height (n+1). This gives us our vector $z^{(j)}$ with height s_j. Now we can get a vector of our activation nodes for layer j as follows:

$$a^{(j)} = g(z^j) \tag{4.8}$$

Where our function g can be applied element-wise to our vector $z^{(j)}$.
We can then add a bias unit (equal to 1) to layer j after we have computed $a^{(j)}$. This will be element $a_0^{(j)}$ and will be equal to 1. To compute our final hypothesis, let's first compute another z vector:

$$z^{(j+1)} = \Theta^{(j)}a^{(j)} \tag{4.9}$$

We get this final z vector by multiplying the next theta matrix after $\theta(j-1)$ with the values of all the activation nodes we just got. This last theta matrix Θ(j) will have only one row which is multiplied by one column $a^{(j)}$ so that our result is a single number. We then get our final result with:

$$h_\Theta(x) = a^{(j+1)} = g(z^{(j+1)}) \tag{4.10}$$

Notice that in this last step, between layer j and layer j+1, we are doing exactly the same thing as we did in logistic regression. Adding all these intermediate layers in neural networks allows us to more elegantly produce interesting and more complex non-linear hypotheses.

4.3 Applications

4.3.1 Examples and Intuition I

4.3.2 Examples and Intuition II