[Andrew Ng - Machine Learning Course](https://www.coursera.org/learn/machine-learning/home/welcome)

Title 2

Title 3

Mustafa Çağrı Ardıç

Start Date: 30/10/2018

Contents

[1 Week 1 3](#_Toc528878658)

[1.1 Introduction 3](#_Toc528878659)

[1.1.1 Welcome 3](#_Toc528878660)

[1.1.2 What is Machine Learning? 3](#_Toc528878661)

[1.1.3 Supervised Learning 4](#_Toc528878662)

[1.1.4 Unsupervised Learning 6](#_Toc528878663)

[1.2 Linear Regression with One Variable - Model and Cost Function 8](#_Toc528878664)

[1.2.1 Model Representation 8](#_Toc528878665)

[1.2.2 Cost Function 9](#_Toc528878666)

[1.2.3 Cost Function – Intuition I 10](#_Toc528878667)

[1.2.4 Cost Function - Intuition II 12](#_Toc528878668)

[1.3 Linear Regression with One Variable – Parameter Learning 14](#_Toc528878669)

[1.3.1 Gradient Descent 14](#_Toc528878670)

[1.3.2 Gradient Descent Intuition 16](#_Toc528878671)

[1.3.3 Gradient Descent for Linear Regression 18](#_Toc528878672)

# Week 1

Welcome to Machine Learning! This week, we introduce the core idea of teaching a computer to learn concepts using data—without being explicitly programmed.

We are going to start by covering linear regression with one variable. Linear regression predicts a real-valued output based on an input value. We discuss the application of linear regression to housing price prediction, present the notion of a cost function, and introduce the gradient descent method for learning.

We’ll also have optional lessons that provide a refresher on linear algebra concepts. Basic understanding of linear algebra is necessary for the rest of the course, especially as we begin to cover models with multiple variables.



## Introduction

### [Welcome](https://www.coursera.org/learn/machine-learning/lecture/RKFpn/welcome)

Applications of Machine Learning:

* Database mining
* Large datasets from growth of automation/web;
* E.g., Web click data (*to understand the user behaviour*), medical records (*to understand diseases better*), biology, engineering (*gene sequences, dna sequences*).
* Applications that cannot be programmed by hand:
* E.g., Autonomous helicopter (*They just did not know how to write a computer program to make this helicopter fly. The computer learn by itself to how to fly this helicopter. There is an interesting* [*video online*](https://www.youtube.com/watch?v=M-QUkgk3HyE)*.*);
* Handwriting recognition (*how the handwritten letters are routed; the envelopes*), most of Natural Language Processing (NLP), Computer Vision.
* Self-customizing programs
* E.g., Amazon, Netflix product recommendations
* Understanding human learning (brain, real AI)

### [What is Machine Learning?](https://www.coursera.org/learn/machine-learning/lecture/Ujm7v/what-is-machine-learning)

There is no consensus among the researchers, but couple of definitions that are widely used are:

* Arthur Samuel (1959). Machine Learning: Field of study that gives computers the ability to learn without being explicitly programmed.
  + *Samuel wrote a* [*checkers playing program*](http://infolab.stanford.edu/pub/voy/museum/samuel.html)*.* *The checker program learned what the good board positions are and what the bad board positions are. Eventually the algorithm bet Arthur Samuel.*
* Tom Mitchell (1998).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.

Example: playing checkers.

E = the experience of playing many games of checkers

T = the task of playing checkers.

P = the probability that the program will win the next game.

In general, any machine learning problem can be assigned to one of two broad classifications:

* Supervised learning
* Unsupervised learning

**Others:** Reinforcement learning, recommender systems.

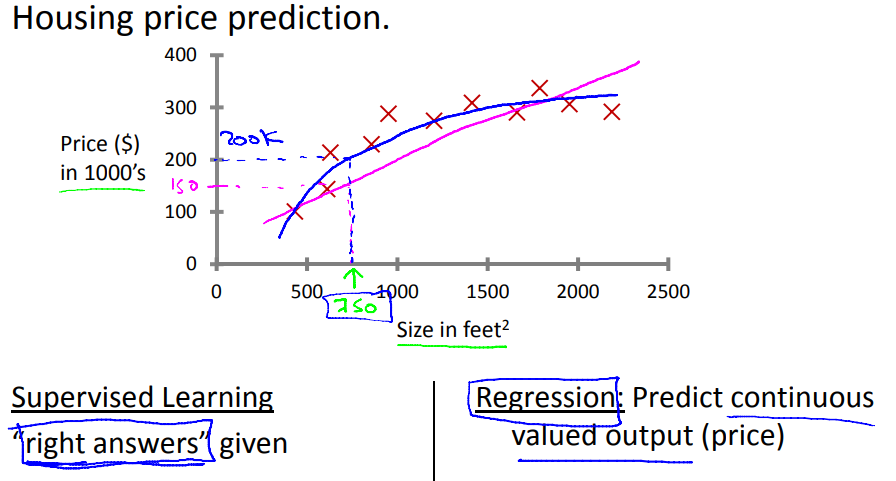
**We wil also talk about:** Practical advice for applying learning algorithms.

### [Supervised Learning](https://www.coursera.org/learn/machine-learning/lecture/1VkCb/supervised-learning)

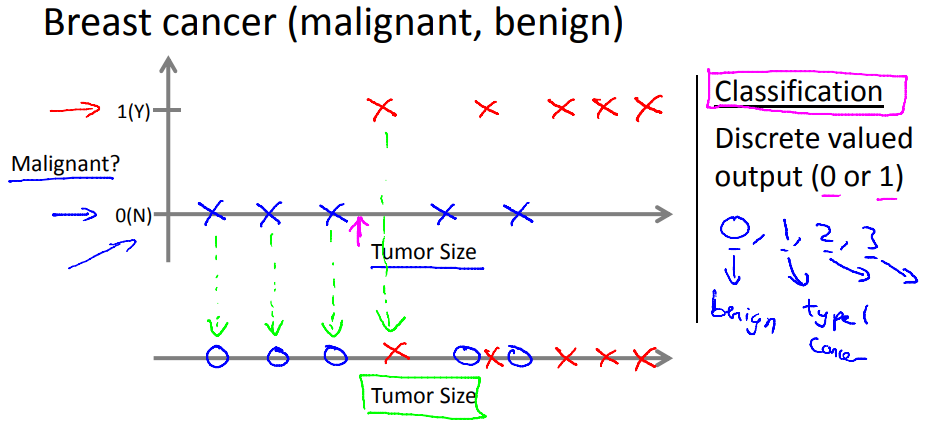
In supervised learning, we are given a data set and already know what our correct output should look like, having the idea that there is a relationship between the input and the output.

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.

**Example 1:** Predicting the price of a house. Let’s say we have a friend how owns a house that is 750 square feet. So, how much money can he get for his house?

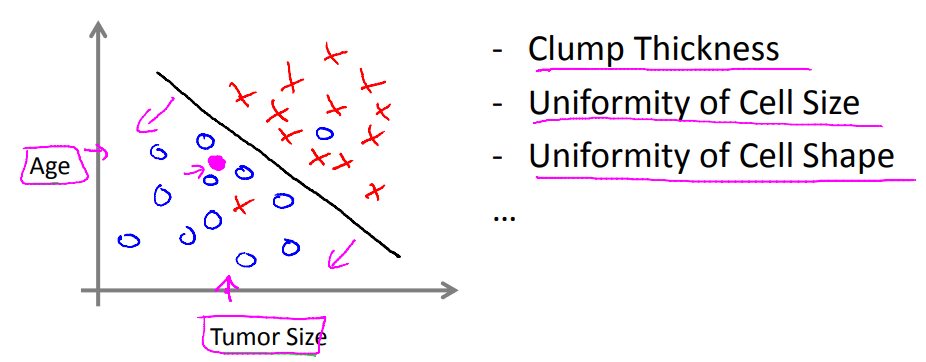
Purple Curve – linear curve  
Blue Curve – quadratic curve

**Example 2:**



This is a classification problem. Because the algorithm is trying to predict whether a tumour is malignant or benign, depending on its size. We can see from the image above that the larger the tumour size is higher the chance of that tumour being a malignant one. We can have more than two target variables as well (e.g., cancer stages).

**Example 3:**



Instead of having one variable, we can have multiple variables. The Age is added as a variable. We can see that the younger the person is, less of a chance that s/he has a malignant tumour.

The learning algorithm fits a curve that separates the classes. The black curve above in our case.

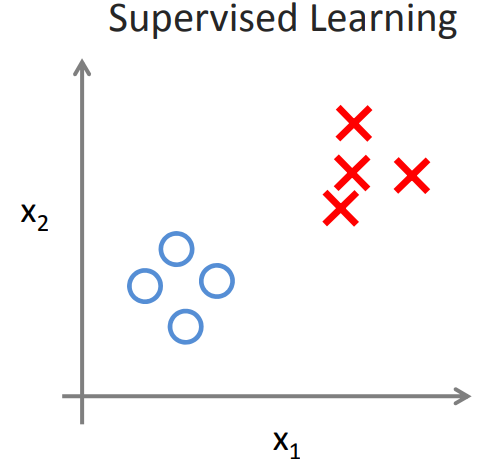
If there are infinite number of features, there is an algorithm called Support Vector Machines that can separate the classes by using a neat trick.

### [Unsupervised Learning](https://www.coursera.org/learn/machine-learning/lecture/olRZo/unsupervised-learning)

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.

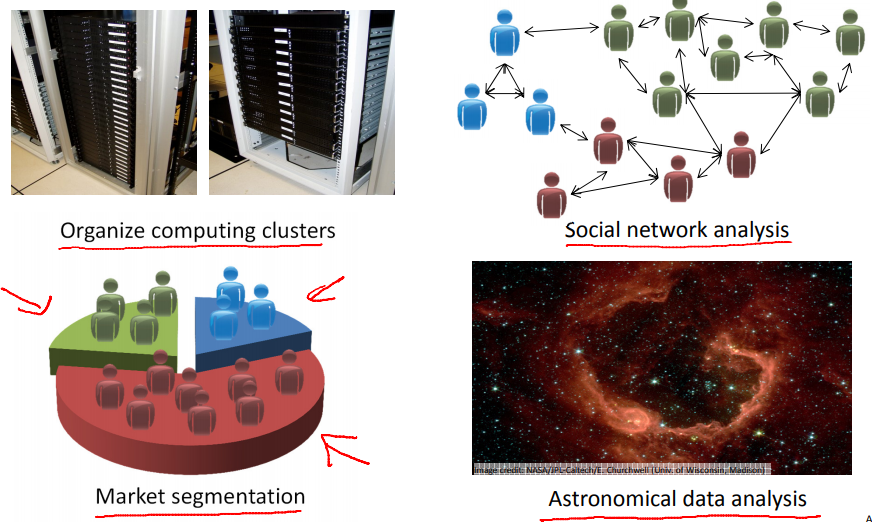
We can derive this structure by clustering the data based on relationships among the variables in.

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

For example [google news](https://news.google.com/?hl=tr&gl=TR&ceid=TR:tr); everyday it looks at tens of thousands of news and it groups them into cohesive news stories based on some similarity measure. In our case the similarity measure is the title of the news.

The applications of unsupervised learning:



This second application is on social network analysis; so given knowledge about which friends you email the most or given your Facebook friends or your Google+ circles, can we automatically identify which are cohesive groups of friends, also which are groups of people that all know each other?

Market segmentation; many companies have huge databases of customer information. So, can you look at this customer data set and automatically discover market segments and automatically group your customers into different market segments so that you can automatically and more efficiently sell or market your different market segments together?

**More examples:**

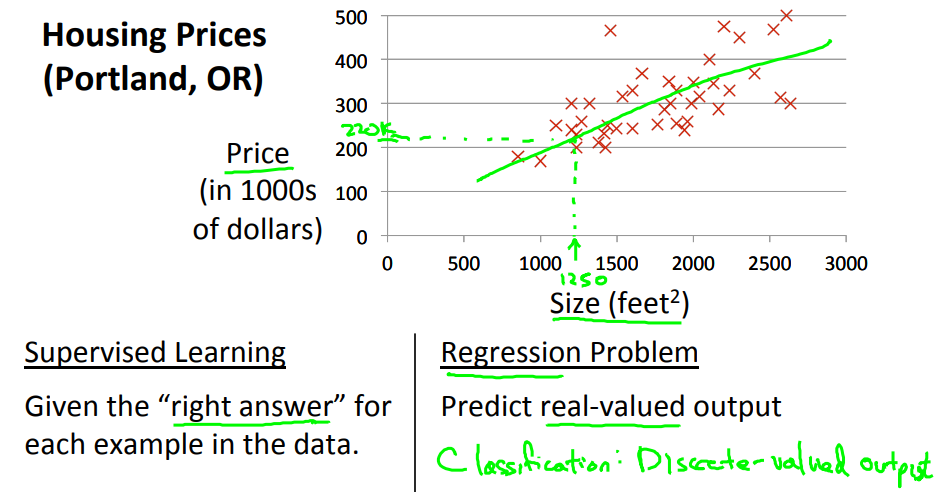
Clustering: Take a collection of 1,000,000 different genes, and find a way to automatically group these genes into groups that are somehow similar or related by different variables, such as lifespan, location, roles, and so on.

Non-clustering: The "Cocktail Party Algorithm", allows you to find structure in a chaotic environment. (i.e. identifying individual voices and music from a mesh of sounds at a cocktail party). [Check cocktail party algorithm.](https://www.technologyreview.com/s/537101/deep-learning-machine-solves-the-cocktail-party-problem/)

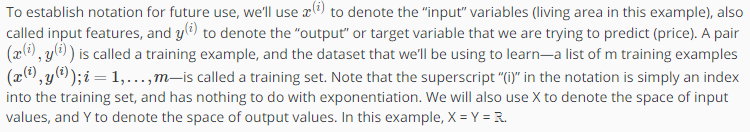
## Linear Regression with One Variable - Model and Cost Function

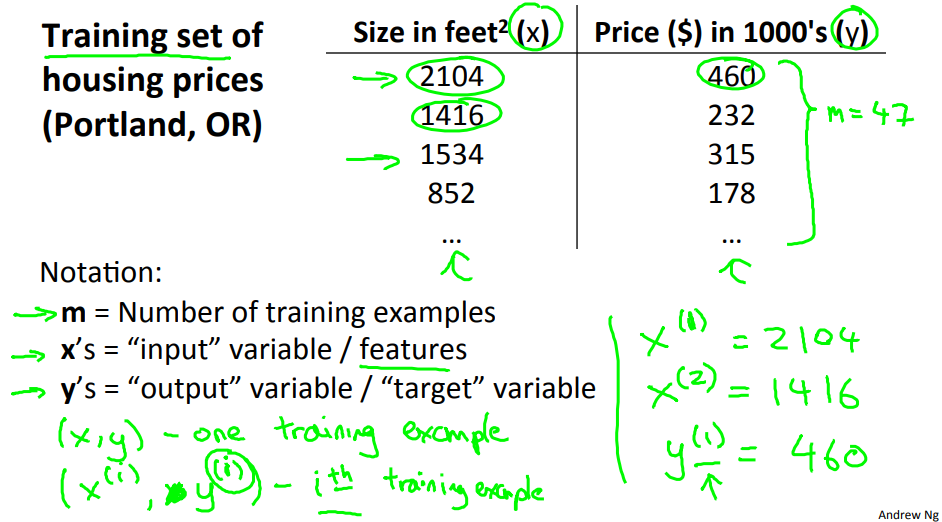
[Link to slides.](https://d3c33hcgiwev3.cloudfront.net/_ec21cea314b2ac7d9e627706501b5baa_Lecture2.pdf?Expires=1541116800&Signature=ZjXEACTHqg~3gYTADWEOiI9yv2zcEZWmX2rSwLz9uHPYxsZUhSYAJ~YbNQ1zrXh23w5xEZkafflWuxF7jG3~e3Aohj4D3svJrXXEEQ933J6eiR~07pWR9ouCJr~ZalLKd5Eq9Fd2oi6jLtiYl~lgmEgrH6P3hKAbpKnJXPhCCG4_&Key-Pair-Id=APKAJLTNE6QMUY6HBC5A)

### [Model Representation](https://www.coursera.org/learn/machine-learning/lecture/db3jS/model-representation)

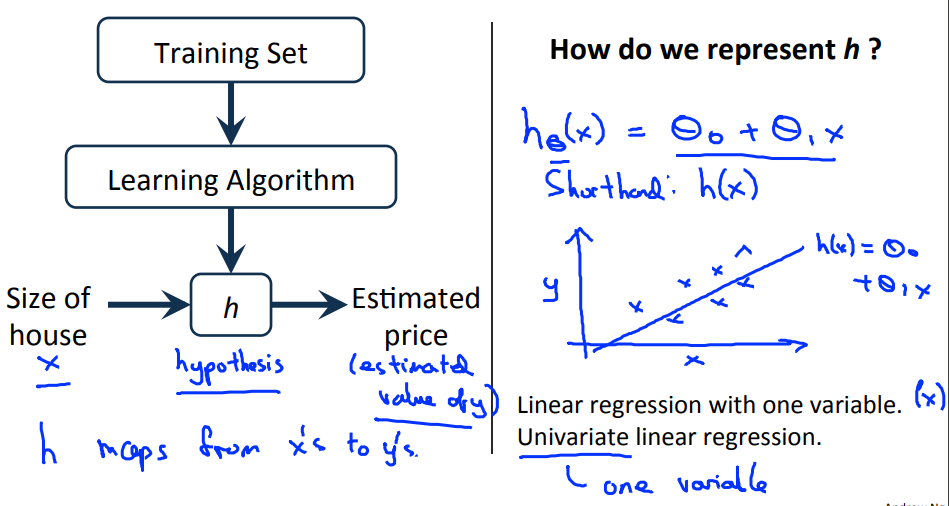


**Notations**







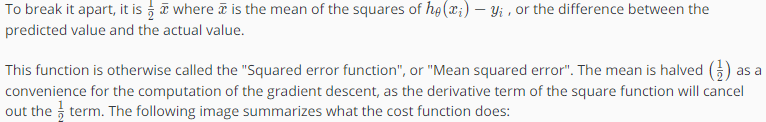


When the target variable that we’re trying to predict is continuous, such as in our housing example, we call the learning problem a **regression problem**. When y can take on only a small number of discrete values (such as if, given the living area, we wanted to predict if a dwelling is a house or an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

### [Cost Function](https://www.coursera.org/learn/machine-learning/lecture/rkTp3/cost-function)

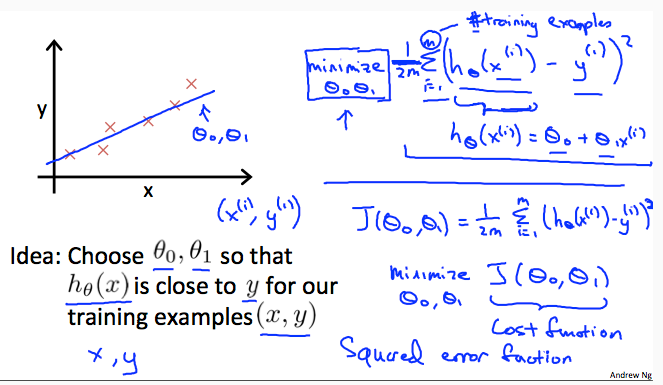
**We can measure the accuracy of our hypothesis function by using a cost function.** This takes an average difference (actually a fancier version of an average) of all the results of the hypothesis with inputs from x's and the actual output y's.



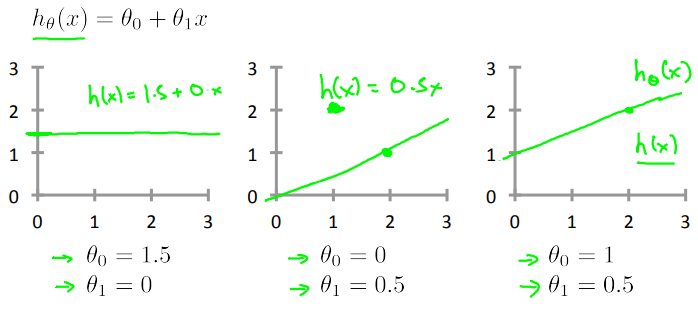


Why there is a in front the summation ? 🡪 because when we take the derivative of the cost function (f2) the 2 and 1/2 cancels each other out.

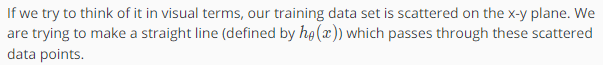
How to choose the coefficients/weights in the linear regression equation; θ’s

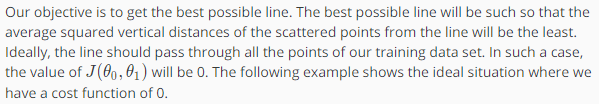


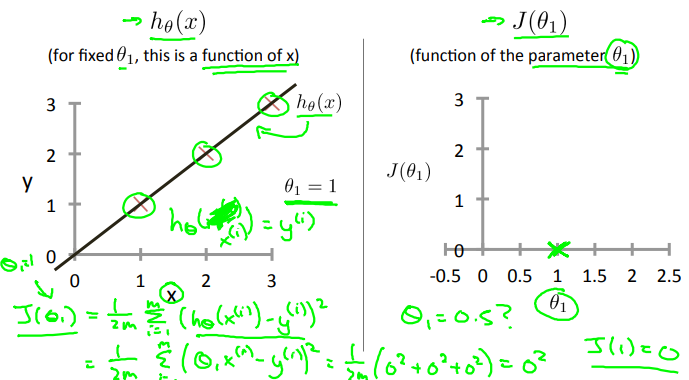
Some first order linear equation examples:



### [Cost Function – Intuition I](https://www.coursera.org/learn/machine-learning/lecture/N09c6/cost-function-intuition-i)

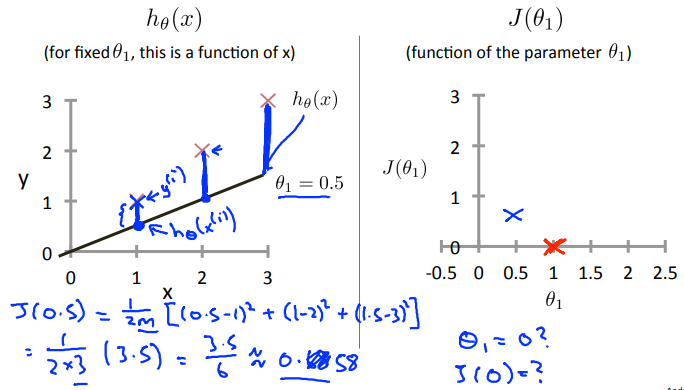




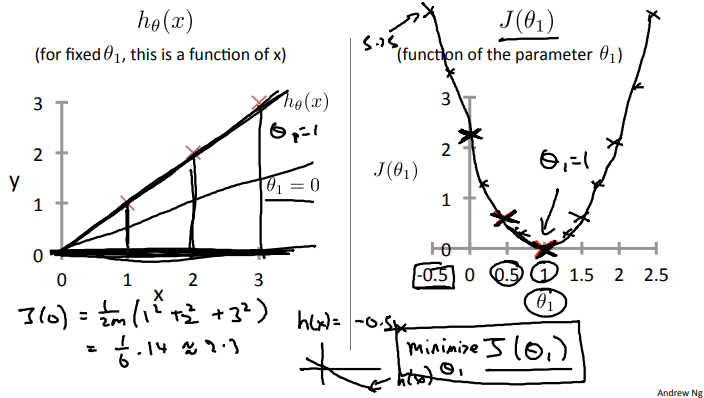


Cost Function vs θ1 value



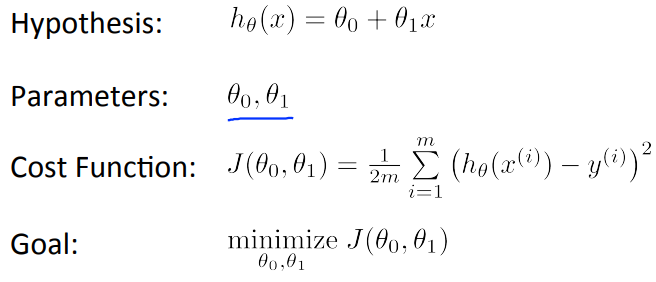


This increases our cost function to 0.58. Plotting several other points yields to the following graph:

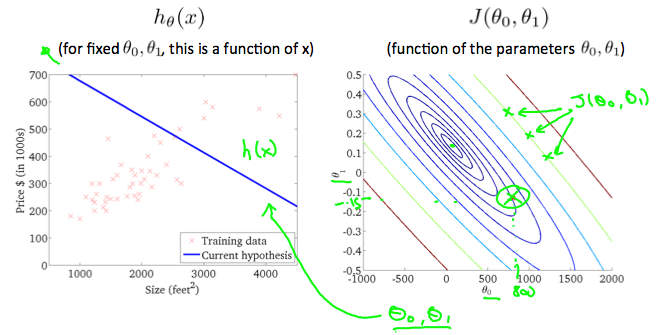


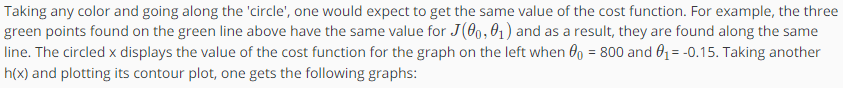


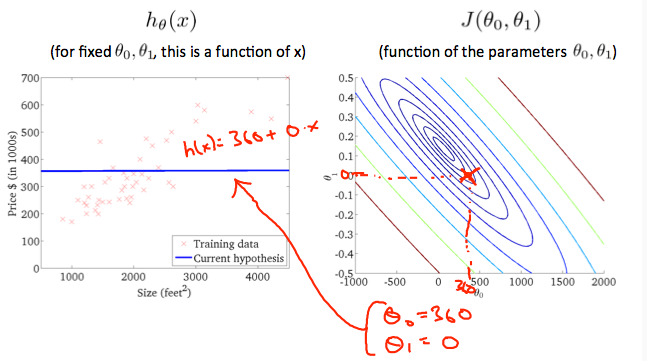
### [Cost Function - Intuition II](https://www.coursera.org/learn/machine-learning/lecture/nwpe2/cost-function-intuition-ii)



A [contour plot](https://en.wikipedia.org/wiki/Contour_line) is a graph that contains many contour lines. A contour line of a two variable function has a constant value at all points of the same line. An example of such a graph is the one to the right below.









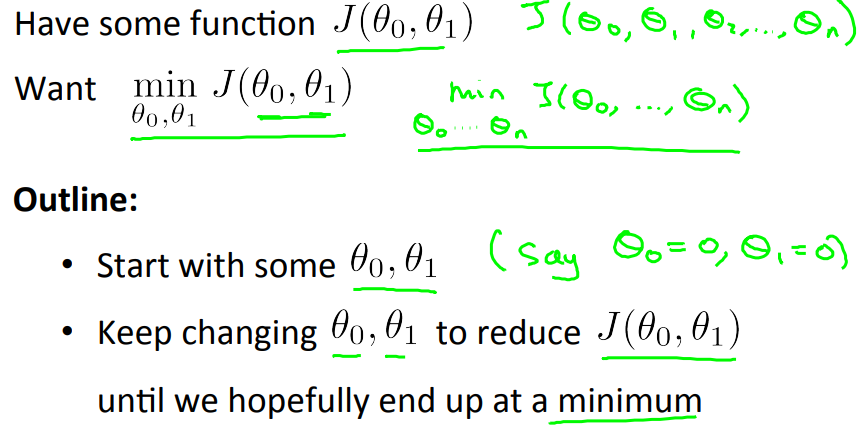


The graph above minimizes the cost function as much as possible and consequently, the result of θ1​ and θ0​ tend to be around 0.12 and 250 respectively. Plotting those values on our graph to the right seems to put our point in the center of the inner most 'circle'.

## Linear Regression with One Variable – Parameter Learning

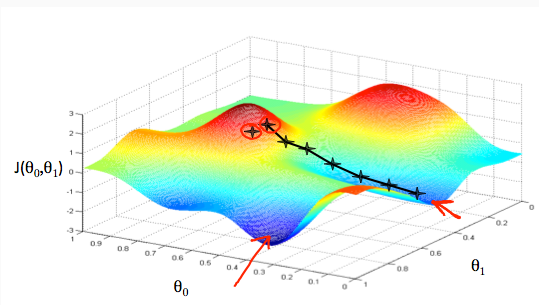
### [Gradient Descent](https://www.coursera.org/learn/machine-learning/lecture/8SpIM/gradient-descent)

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.



Explanation from the [video](https://www.coursera.org/learn/machine-learning/lecture/8SpIM/gradient-descent) (minute 2.15).

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.

#### Learning Rate

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*(*θ*0​,*θ*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.

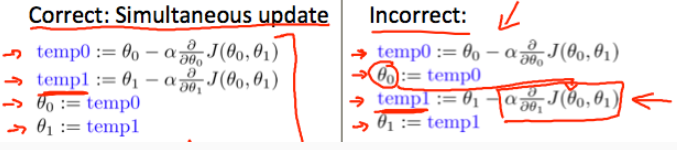
The gradient descent algorithm is:

repeat until convergence:



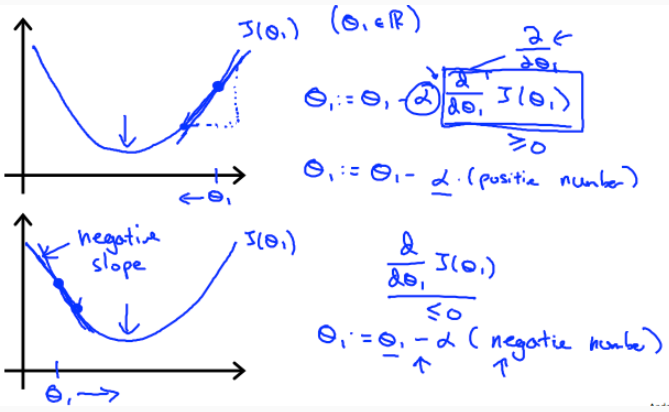
Where j=0,1 represents the feature index number and := is the assignment operator.

At each iteration j, one should simultaneously update the parameters *θ*1​,*θ*2​,...,*θn*​. Updating a specific parameter prior to calculating another one on the *j*(*th*) iteration would yield to a wrong implementation. On the right side (incorrect), we can see that *θ*0 is updated before *θ*1. This effects the next value of *θ*1. The *θ*1 values on the left and the right side are different.



### [Gradient Descent Intuition](https://www.coursera.org/learn/machine-learning/lecture/GFFPB/gradient-descent-intuition)

Here we explore the scenario where there is one parameter θ1 and we plot its cost function to implement a gradient descent. Our formula for a single parameter is :



Regardless of the slope's sign for  ​ *J*(*θ*1​), *θ*1​ eventually converges to its minimum value. The following graph shows that when the slope is negative, the value of *θ*1​ increases and when it is positive, the value of *θ*1​ decreases.



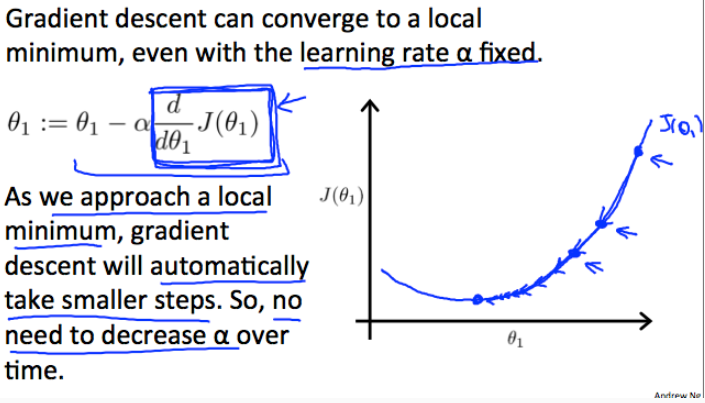
On a side note, we should adjust our parameter α to ensure that the gradient descent algorithm converges in a reasonable time. Failure to converge or too much time to obtain the minimum value imply that our step size is wrong.



#### How does gradient descent converge with a fixed step size α?

The intuition behind the convergence is that   ​ *J*(*θ*1​) approaches 0 as we approach the bottom of our convex function. At the minimum, the derivative will always be 0 and thus we get:





5th

4th

3rd

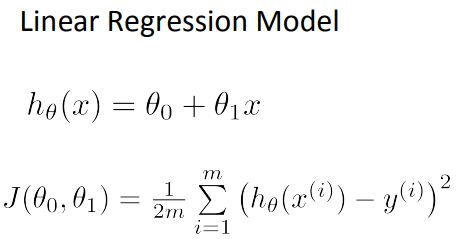
2nd

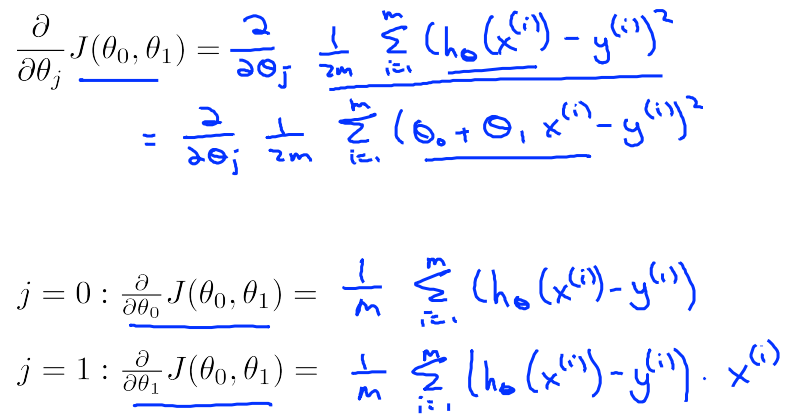
1st

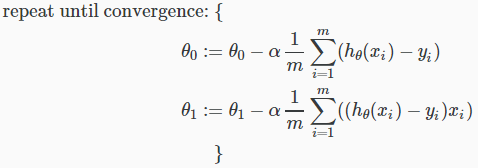
At the 1st step, the derivative of *J*(*θ*1​) is a large value. Therefore α. *J*(*θ*1​) is a large value. At the 2nd step, the derivative is smaller therefore, α. *J*(*θ*1​) is smaller than the 1st step (α is fixed). Eventhough the a stays the same, since derivative is becoming smaller, gradient descent converges to a local minimum.

### [Gradient Descent for Linear Regression](https://www.coursera.org/learn/machine-learning/lecture/kCvQc/gradient-descent-for-linear-regression)

When specifically applied to the case of linear regression, a new form of the gradient descent equation can be derived. We can substitute our actual cost function and our actual hypothesis function and modify the equation to :

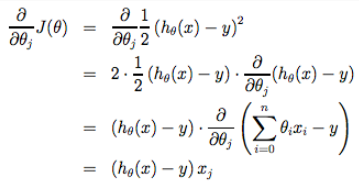






where m is the size of the training set, *θ*0​ a constant that will be changing simultaneously with *θ*1​ and xi​,yi ​are values of the given training set (data).

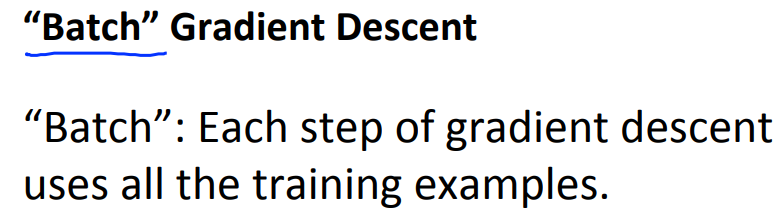
Note that we have separated out the two cases for *θj*​ into separate equations for *θ*0​ and *θ*1​; and that for *θ*1​ we are multiplying *xi*​ at the end due to the derivative. The following is a derivation of  ​*J*(*θ*) for a single example:



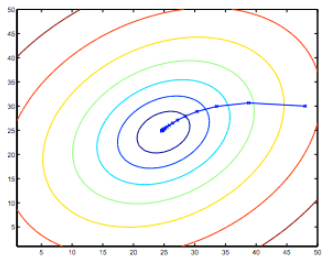
The point of all this is that if we start with a guess for our hypothesis and then repeatedly apply these gradient descent equations, our hypothesis will become more and more accurate.

#### Batch Gradient Decent

So, this is simply gradient descent on the original cost function J. This method looks at every example in the entire training set on every step, and is called **batch gradient descent.**



Note that, while gradient descent can be susceptible to local minima in general, the optimization problem we have posed here for linear regression has only one global, and no other local, optima; thus gradient descent always converges (assuming the learning rate α is not too large) to the global minimum. Indeed, J is a convex quadratic function. Here is an example of gradient descent as it is run to minimize a quadratic function.



The ellipses shown above are the contours of a quadratic function. Also shown is the trajectory taken by gradient descent, which was initialized at (48,30). The x’s in the figure (joined by straight lines) mark the successive values of θ that gradient descent went through as it converged to its minimum. Check the [video](https://www.coursera.org/learn/machine-learning/lecture/kCvQc/gradient-descent-for-linear-regression) to see how the coefficients are updated after a gradient decent step (5.35).

# Week 2

Welcome to week 2! I hope everyone has been enjoying the course and learning a lot! This week we’re covering linear regression with multiple variables. we’ll show how linear regression can be extended to accommodate multiple input features. We also discuss best practices for implementing linear regression.

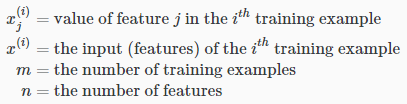


## Multivariate Linear Regression

### [Multiple Features](https://www.coursera.org/learn/machine-learning/lecture/6Nj1q/multiple-features)

Linear regression with multiple variables is also known as "multivariate linear regression".

We now introduce notation for equations where we can have any number of input variables.

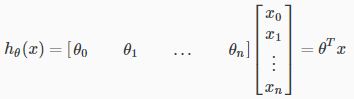


The multivariable form of the hypothesis function accommodating these multiple features is as follows:

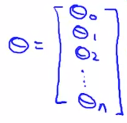


In order to develop intuition about this function, we can think about *θ*0​ as the basic price of a house, *θ*1​ as the price per square meter,  *θ*2​ as the price per floor, etc. *x*1​ will be the number of square meters in the house, *x*2​ the number of floors, etc.

Using the definition of matrix multiplication, our multivariable hypothesis function can be concisely represented as:



Please note that *θ* is in the following format:

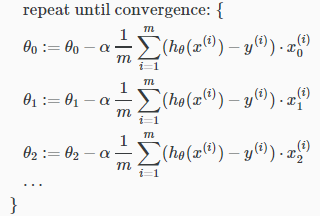


This is a vectorization of our hypothesis function for one training example; see the lessons on vectorization to learn more.

**Remark:** Note that for convenience reasons in this course we assume ***x*0(*i*) =1** for (*i*∈1,…,*m*). This allows us to do matrix operations with *θ* and x. Hence making the two vectors '*θ*' and  *x*(*i*) match each other element-wise (that is, have the same number of elements: n+1).]

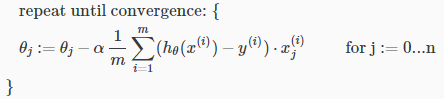
### [Gradient Descent for Multiple Variables](https://www.coursera.org/learn/machine-learning/lecture/Z9DKX/gradient-descent-for-multiple-variables)

The gradient descent equation itself is generally the same form; we just have to repeat it for our 'n' features:

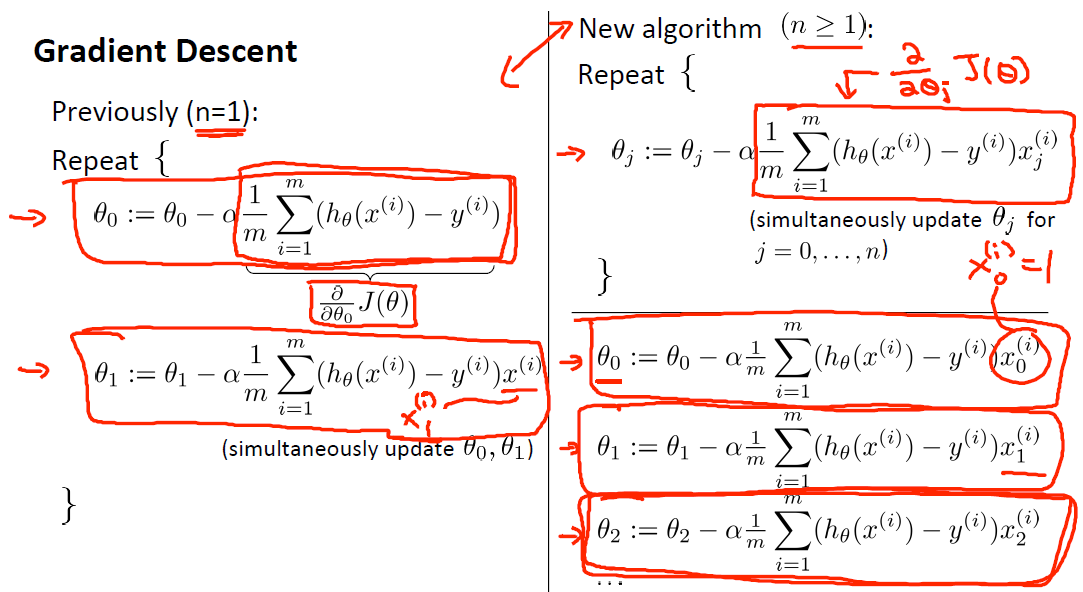


X0(i)=1

In other words:



The following image compares gradient descent with one variable to gradient descent with multiple variables:



### [Gradient Descent in Practice I – Feature Scaling](https://www.coursera.org/learn/machine-learning/lecture/xx3Da/gradient-descent-in-practice-i-feature-scaling)

We can speed up gradient descent by having each of our input values in roughly the same range. This is because θ will descend quickly on small ranges and slowly on large ranges, and so will oscillate inefficiently down to the optimum when the variables are very uneven.

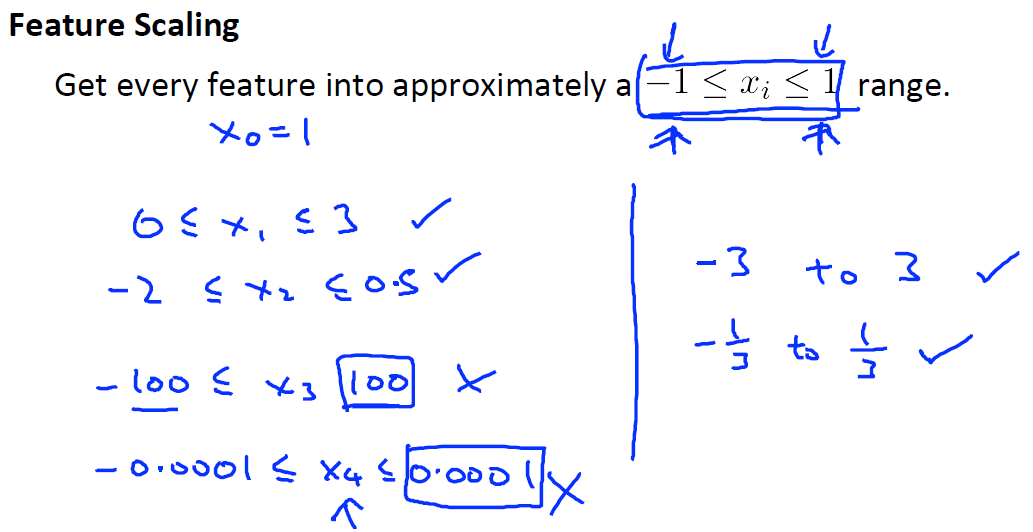
The way to prevent this is to modify the ranges of our input variables so that they are all roughly the same. Ideally:

−1 ≤  *x*(*i*)​ ≤ 1

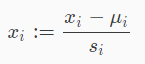
or

−0.5 ≤  *x*(*i*)​ ≤ 0.5

**These aren't exact requirements;** we are only trying to speed things up. The goal is to get all input variables into roughly one of these ranges, give or take a few. *The ranges can also be as follows (the incorrect ones are highlighted by crosses):*



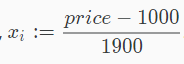
Two techniques to help with this are **feature scaling** and **mean normalization**. Feature scaling involves dividing the input values by the range (i.e. the maximum value minus the minimum value) of the input variable, resulting in a new range of just 1. Mean normalization involves subtracting the average value for an input variable from the values for that input variable resulting in a new average value for the input variable of just zero. To implement both of these techniques, adjust your input values as shown in this formula:



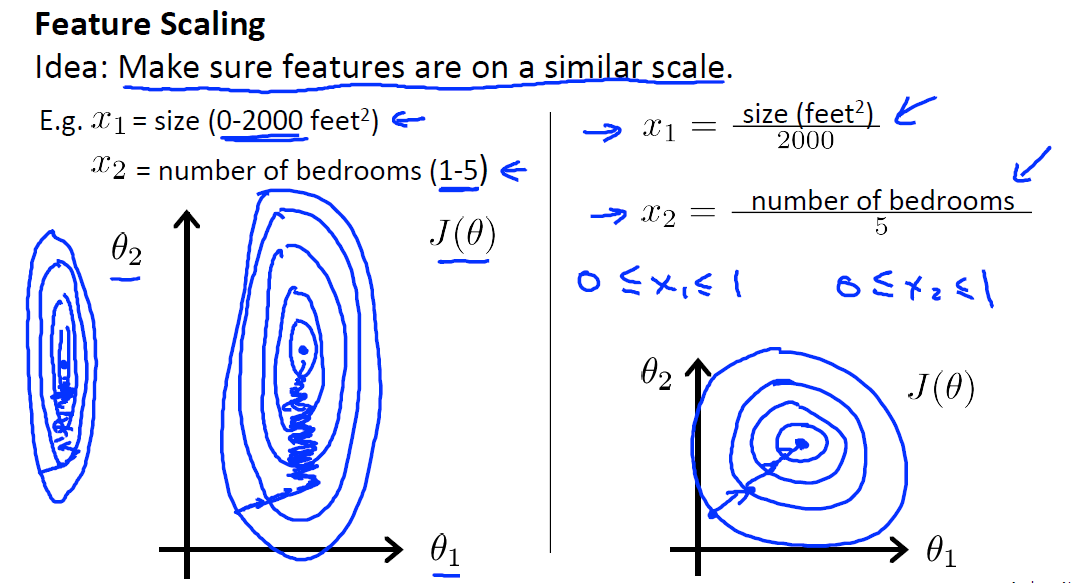
Where *μi*​ is the **average** of all the values for feature (i) and *si*​ is the range of values (max - min), or *si*​ is the standard deviation.

Note that dividing by the range, or dividing by the standard deviation, give different results. The quizzes in this course and the programming exercises use standard deviation.

For example, if *xi*​ represents housing prices with a range of 100 to 2000 and a mean value of 1000, then:



*The illustration that highlights the importance of feature scaling:*



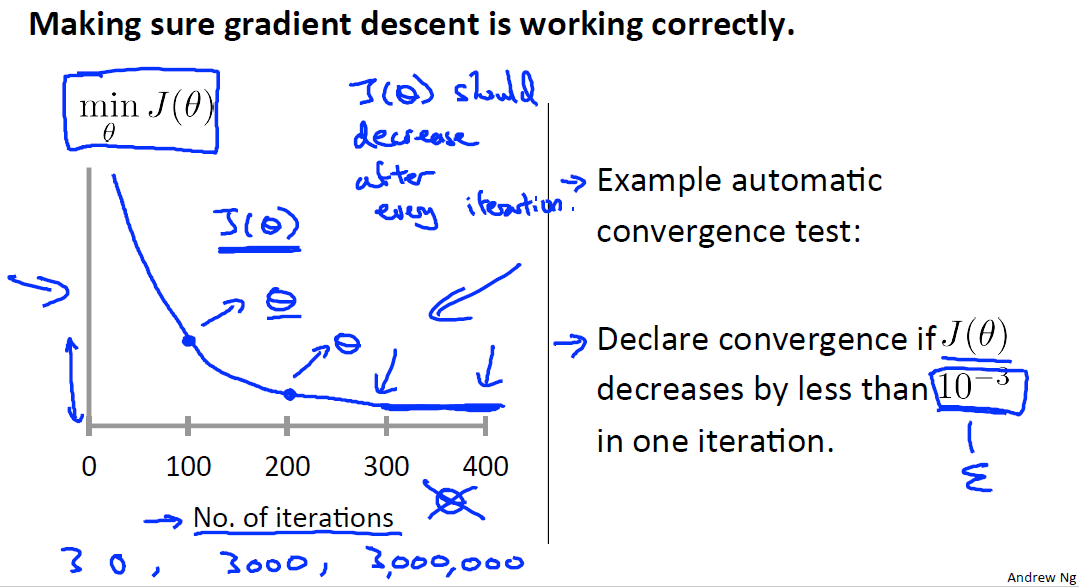
*On the left side of the image we can see that θ2 corresponds to a variable with large numbers (size in our case 0-2000), whereas, θ1 corresponds to a variable with a small numbers (number of bedrooms 1-5). We can see that the contour plot is is very thin and tall (could have been other way around; fat and short). This causes gradient descent algorithm to take quite a long time to minimise the cost function.*

*On the right side, when both features are standardised, we can see that the contour plot is more circle like shape and gradiendt descent algorithm minimises the cost function faster.*

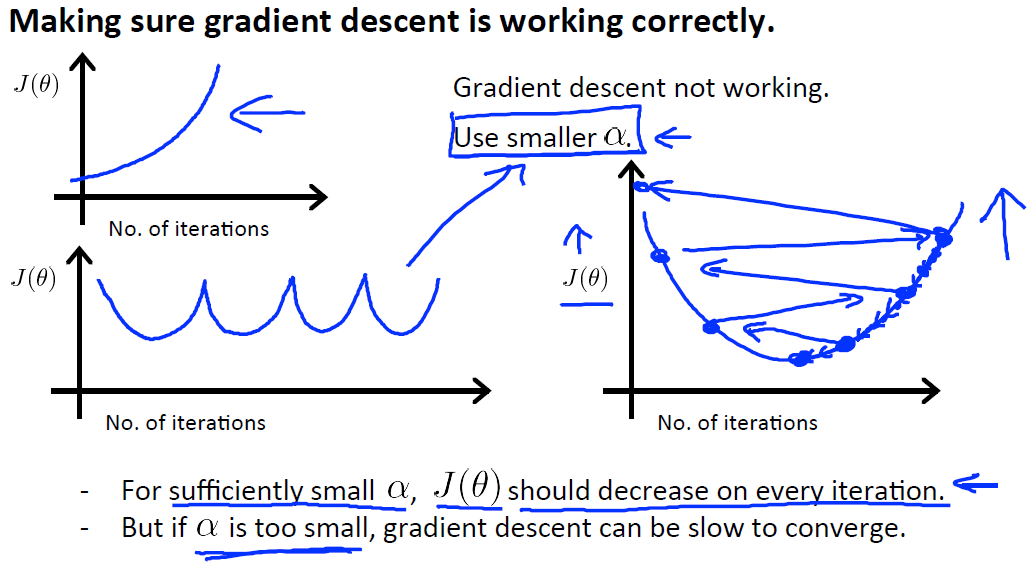
### [Gradient Descent in Practice II – Learning Rate](https://www.coursera.org/learn/machine-learning/lecture/3iawu/gradient-descent-in-practice-ii-learning-rate)

**Debugging gradient descent.** Make a plot with *number of iterations* on the x-axis. Now plot the cost function, J(θ) over the number of iterations of gradient descent. If J(θ) ever increases, then you probably need to decrease α.

**Automatic convergence test.** Declare convergence if J(θ) decreases by less than E in one iteration, where E is some small value such as 10−3. **However in practice it's difficult to choose this threshold value.**



It has been proven that if learning rate α is sufficiently small, then J(θ) will decrease on every iteration.

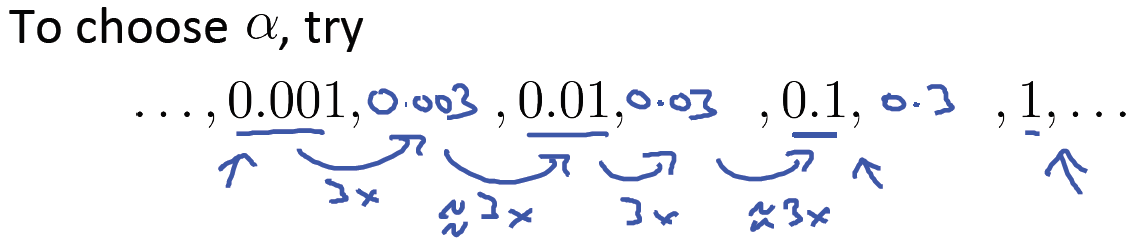


To summarize:

**If *α* is too small:** slow convergence.

**If *α* is too large:** may not decrease on every iteration and thus may not converge.

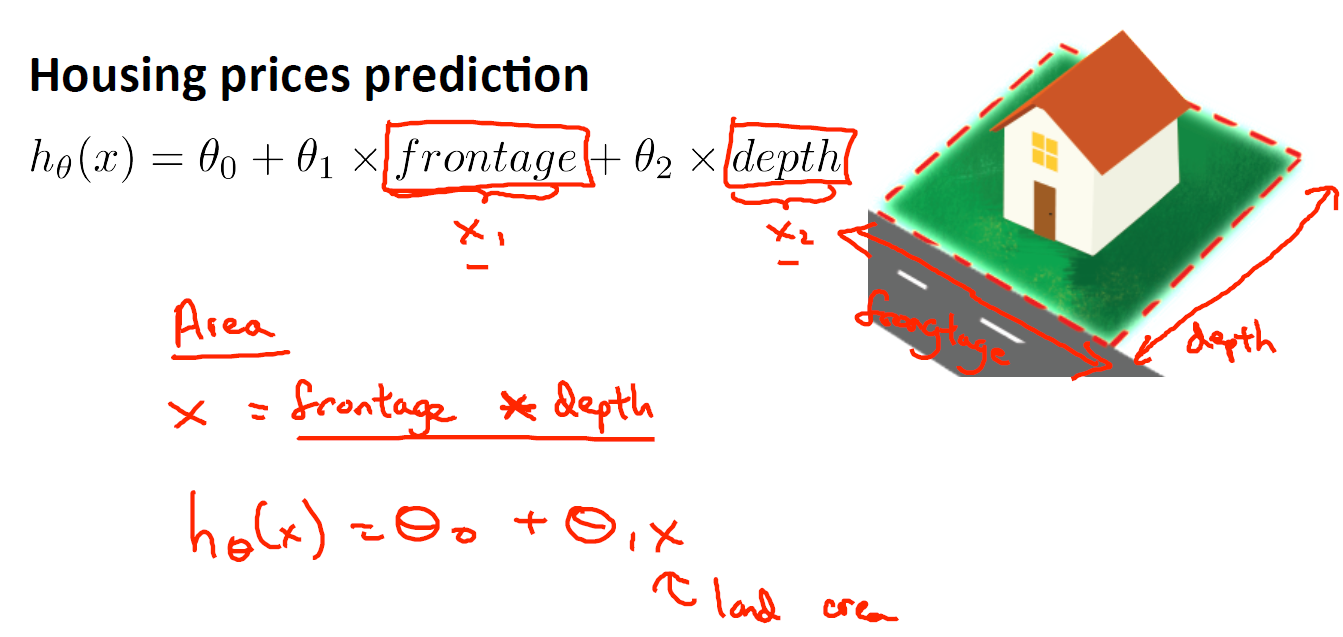
Try range of values for *α*  and plot the cost function against number of iterations for every selected *α* :



### [Features and Polynomial Regression](https://www.coursera.org/learn/machine-learning/lecture/Rqgfz/features-and-polynomial-regression)

We can improve our features and the form of our hypothesis function in a couple different ways.

We can **combine** multiple features into one. For example, we can combine  *x*1​ and  *x*2​ into a new feature *x*3​ by taking *x*1​⋅*x*2​. (*Area is the created feature by taking the product of frontage and depth*):

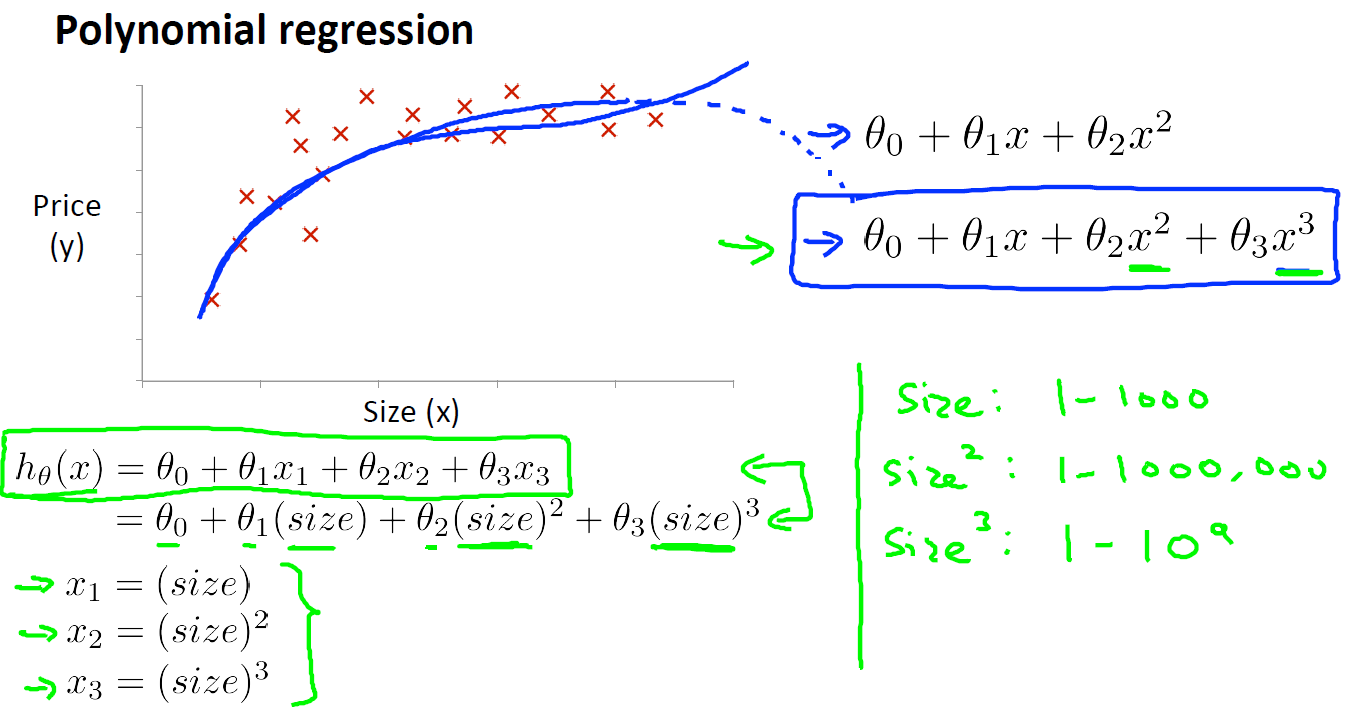


#### Polynomial Regression

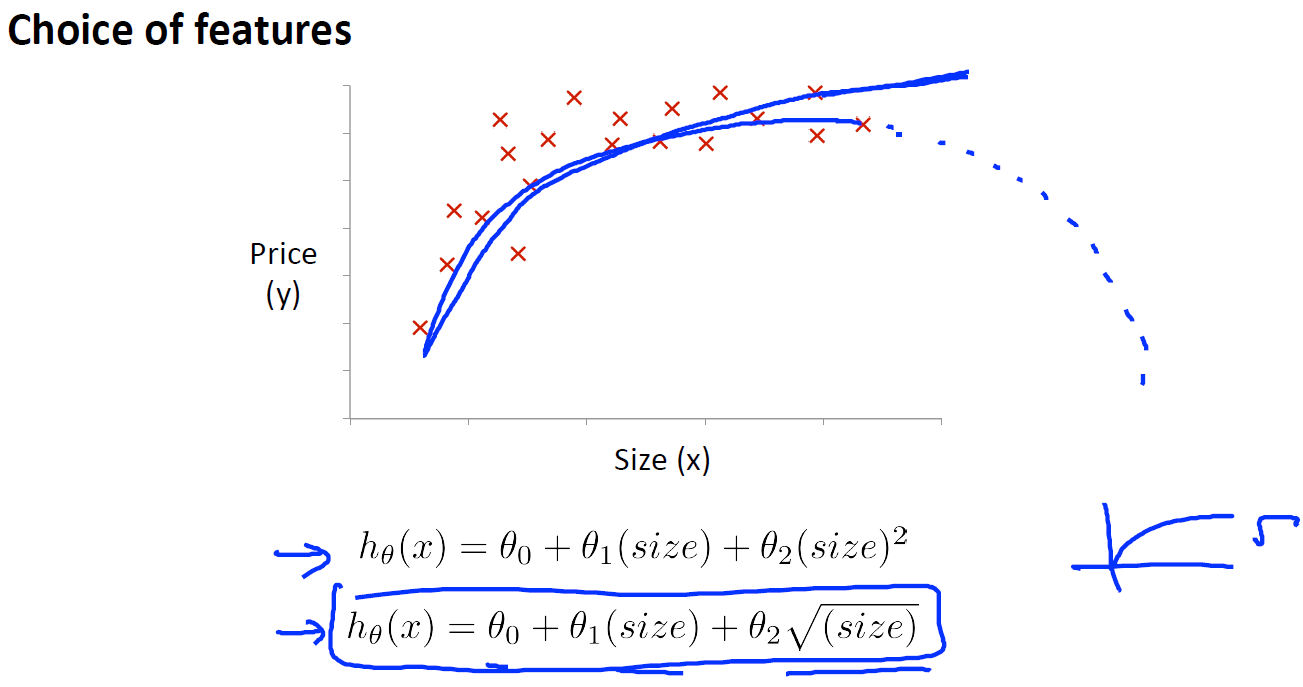
Our hypothesis function need not be linear (a straight line) if that does not fit the data well.

We can **change the behavior or curve** of our hypothesis function by making it a quadratic, cubic or square root function (or any other form).

For example, if our hypothesis function *hθ*​(*x*)=*θ*0​+*θ*1​*x*1​ then we can create additional features based on *x*1​, to get the quadratic function *hθ*​(*x*)=*θ*0​+*θ*1​*x*1​+*θ*2​*x*12​ or the cubic function *hθ*​(*x*)=*θ*0​+*θ*1​*x*1​+*θ*2​*x*12​+*θ*3​*x*13​:

  
In the cubic version, we have created new features *x*2​ and *x*3​ where *x*2​=*x*12​ and *x*3​=*x*13​.

To make it a square root function, we could do: *hθ*​(*x*)=*θ*0​+*θ*1​*x*1​+*θ*2​*x*1​​1/2 (*The increment of price decreases as the size increases if square function is used which is not very realistic*):



One important thing to keep in mind is, if you choose your features this way then feature scaling becomes very important.

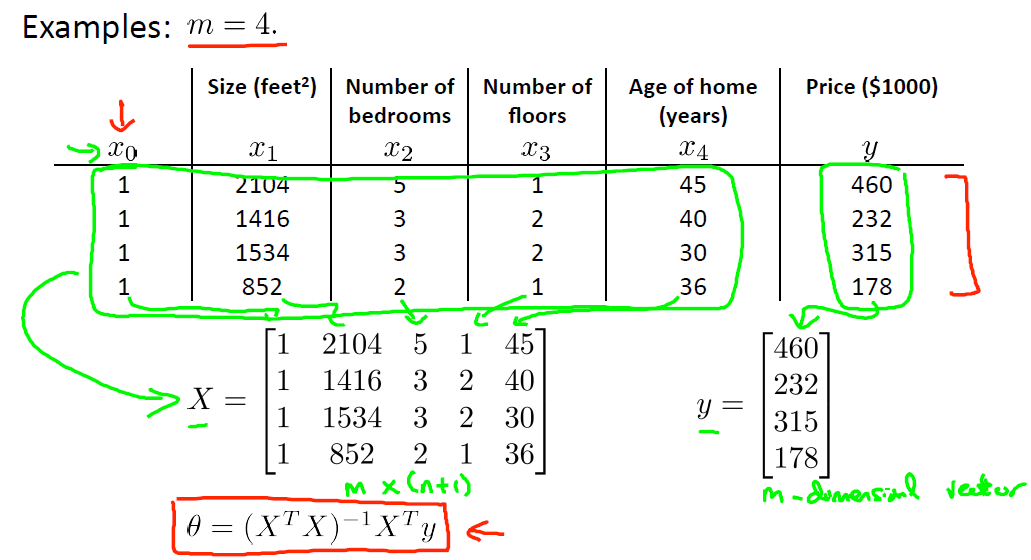
eg. If *x*1​ has range 1 – 103 then range of *x*12​ becomes 1 - 106 and that of *x*13​ becomes 1 – 109.

## Computing Parameters Analytically

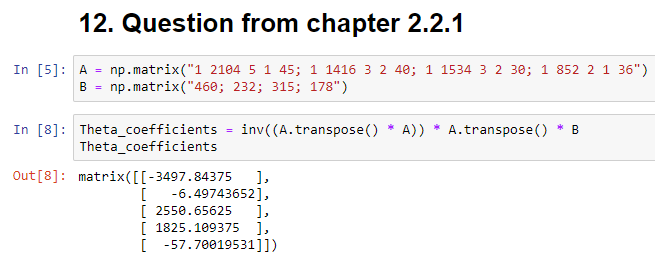
### [Normal Equation](https://www.coursera.org/learn/machine-learning/lecture/2DKxQ/normal-equation)

Gradient descent gives one way of minimizing J (cost function). 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 θj ’s, and setting them to zero. This allows us to find the optimum theta without iteration. The normal equation formula is given below:

θ=(XTX)−1.XTy

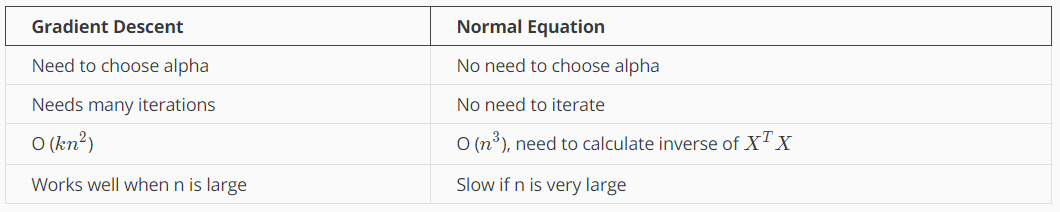


*The θ coefficients of the dataset above are:*



There is no need to do feature scaling with the normal equation.

The following is a comparison of gradient descent and the normal equation:

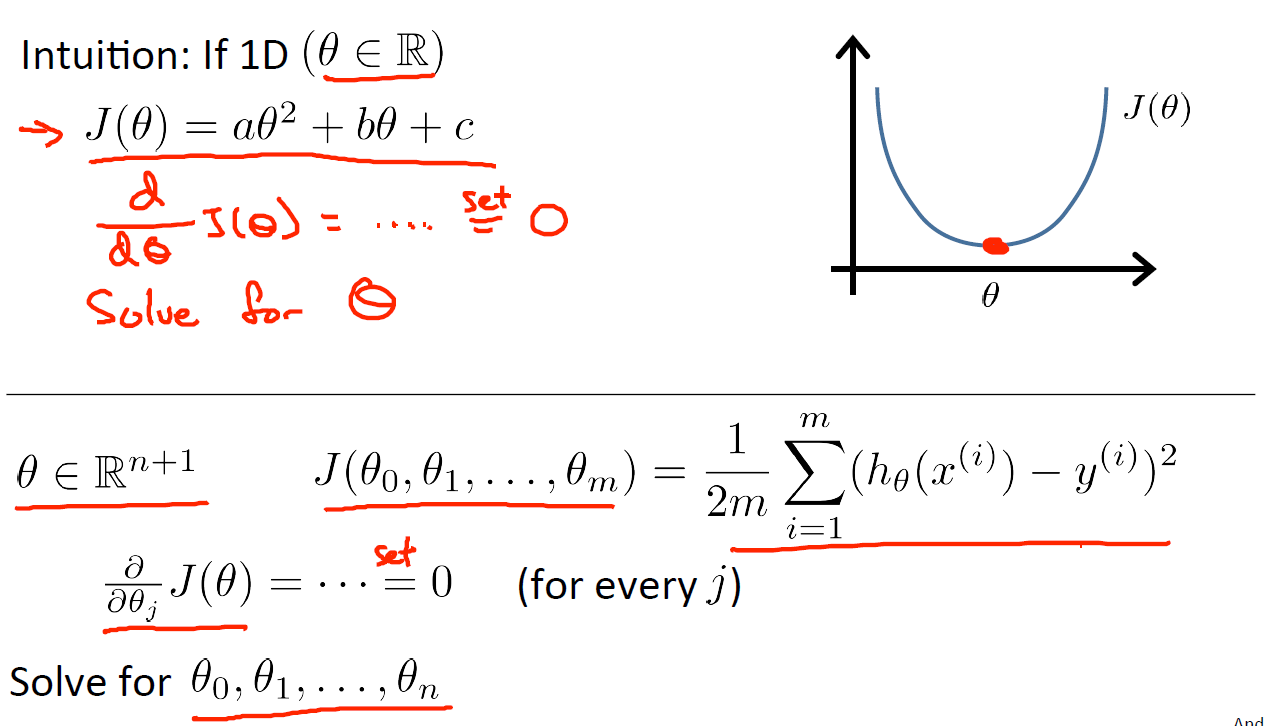


With the normal equation, computing the inversion has complexity O(*n*3). So if we have a very large number of features, the normal equation will be slow. In practice, when n exceeds 10,000 it might be a good time to go from a normal solution to an iterative process.

Also some matrices are not invertable. Therefore, we simply cannot use normal equation in those cases (θ=(XTX)−1.XTy). However, please bear in mind that pinv ([pseudo inverse function](https://docs.scipy.org/doc/numpy-1.15.1/reference/generated/numpy.linalg.pinv.html)) can still take the inverse of a non-invertable function.

#### Intuition

*Take the derivative of J(θ) for every θ value and set it to 0. Then solve the equation. This gives us the minimum point (As highlighted by aqua colour) .*



### [Normal Equation Noninvertability](https://www.coursera.org/learn/machine-learning/lecture/zSiE6/normal-equation-noninvertibility)

If XTX is noninvertible, the common causes might be having:

* Redundant features, where two features are very closely related (i.e. they are linearly dependent)
* Too many features (e.g. m ≤ n). In this case, delete some features or use "regularization" (to be explained in a later lesson).

Solutions to the above problems include deleting a feature that is linearly dependent with another or deleting one or more features when there are too many features.

# ASSESMENT 1

**The assignment brief**

****

**The assigment data files**

** <----> **

**My code single variable**

** <----> **

**My code multiple variables**

** <----> **

**The example code**

 **<----> **

# Week 3

Welcome to week 3! This week, we’ll be covering logistic regression. Logistic regression is a method for classifying data into discrete outcomes. For example, we might use logistic regression to classify an email as spam or not spam. In this module, we introduce the notion of classification, the cost function for logistic regression, and the application of logistic regression to multi-class classification.

We are also covering regularization. Machine learning models need to generalize well to new examples that the model has not seen in practice. We’ll introduce regularization, which helps prevent models from overfitting the training data.

## Logistic Regression - Classification and Representation



### [Classification](https://www.coursera.org/learn/machine-learning/lecture/wlPeP/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.



B

A

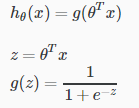
*We can see from the image above that when the new malignant observation is introduced to the model (marked by blue rectangular), the regression line changes from A to B which causes the highlighted (orange rectangular) observations to be misclassfied as not malignant by the model when they clearly are malignant. Hence, in the case of classification, Linear Regression is not quite suitable.*

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∈{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.

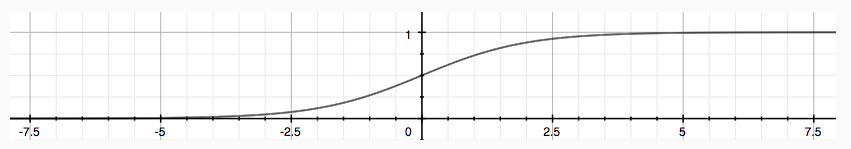
### [Hypothesis Representation](https://www.coursera.org/learn/machine-learning/lecture/RJXfB/hypothesis-representation)

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 *hθ*​(*x*) to take values larger than 1 or smaller than 0 when we know that y ∈ {0, 1}. To fix this, let’s change the form for our hypotheses *hθ*​(*x*) to satisfy 0≤*hθ*​(*x*)≤1. This is accomplished by plugging *θTx* into the Logistic Function.

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



The following image shows us what the sigmoid function looks like:



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.

*hθ*​(*x*) will give us the **probability** that our output is 1. For example, *hθ*​(*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%).



### [Decision Boundary](https://www.coursera.org/learn/machine-learning/lecture/WuL1H/decision-boundary)

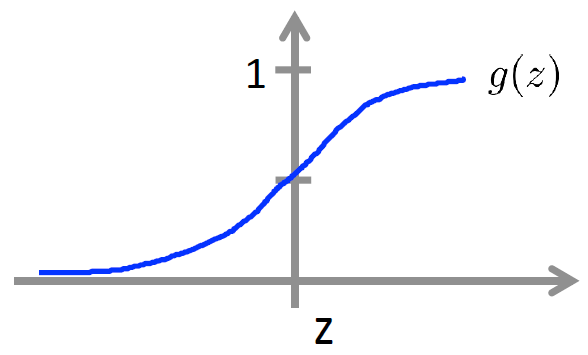
In order to get our discrete 0 or 1 classification, we can translate the output of the hypothesis function as follows:



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:



This is because:



z=*θTx* – we can see from the sigmoid function graph that when z > 0 (highlighted by red circle), g(z) >0.5

0.5

Remember.



So if our input to g is *θTX*, then that means:

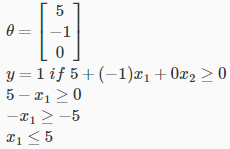


From these statements we can now say:



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**:

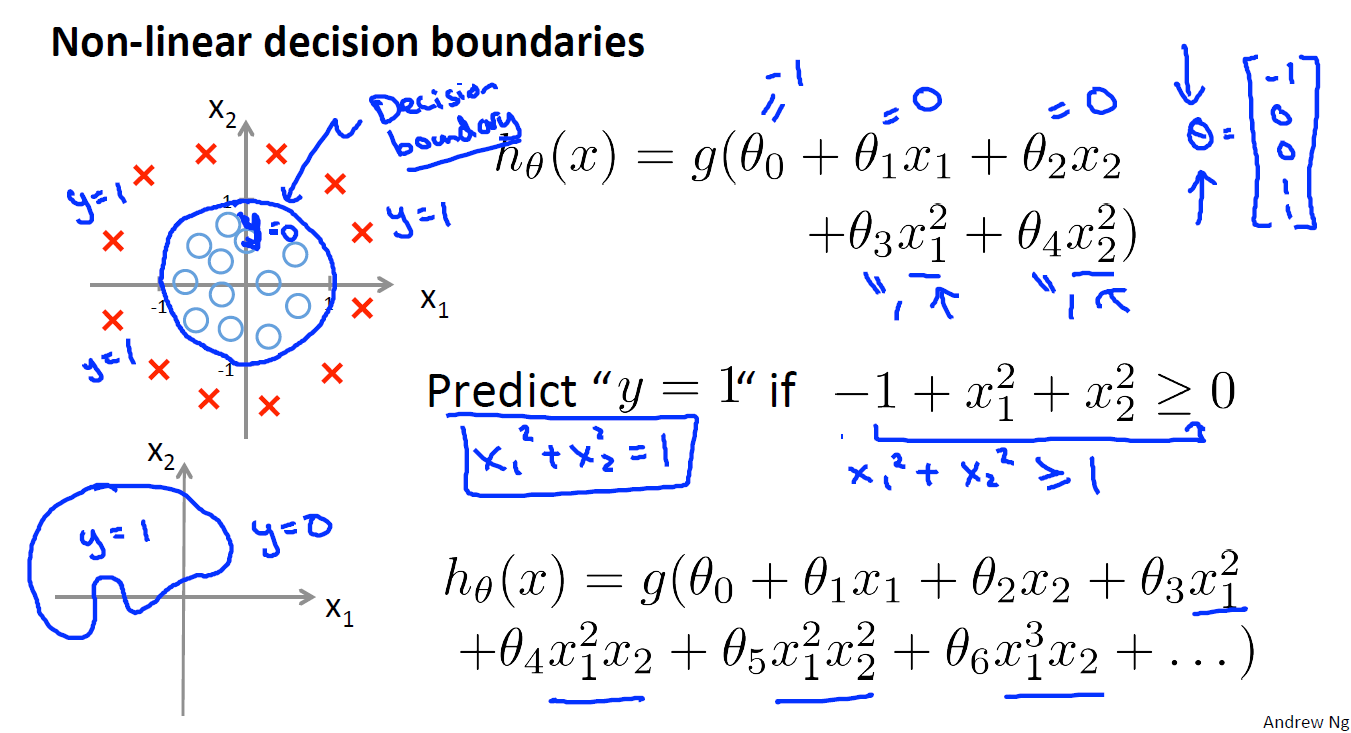


In this case, our decision boundary is a straight vertical line placed on the graph where *x*1=5, and everything to the left of that denotes y=1, while everything to the right denotes y=0.

Again, the input to the sigmoid function g(z) (e.g. *θTX*) doesn't need to be linear, and could be a function that describes a circle (e.g. z *z*=*θ*0​+*θ*1​*x*12​+*θ*2​*x*22​) or any shape to fit our data.

#### Non-linear decision boundaries

*It is also possible to create non-linear decision boundaries by using logistic regression. Please recall that in Linear Regression, it was possible to fit a* [*polynomial regression*](#_Polynomial_Regression) *curve to capture the non-linear manifolds. As logistic regression is the function of Linear Regression, if we choose to use a polynomial function instead of a linear function, then we would be able to create a non-linear decision boundary. From the example below we can see that by using a second order polynomial function, we can create a circle shaped decision boundary (1st). In fact, it is possible to introduce even higher order polynomial functions to create more complex decision boundaries (2nd).*



2nd

1st

## Regularization - Solving the Problem of Overfitting



### [The Problem of Overfitting](https://www.coursera.org/learn/machine-learning/lecture/ACpTQ/the-problem-of-overfitting)

# Appendices

## Appendix I – Linear Algrebra Review



### [Matrices and Vectors](https://www.coursera.org/learn/machine-learning/lecture/38jIT/matrices-and-vectors)

Matrices are 2-dimensional arrays:



The above matrix has four rows and three columns, so it is a 4 x 3 matrix.

A vector is a matrix with one column and many rows:

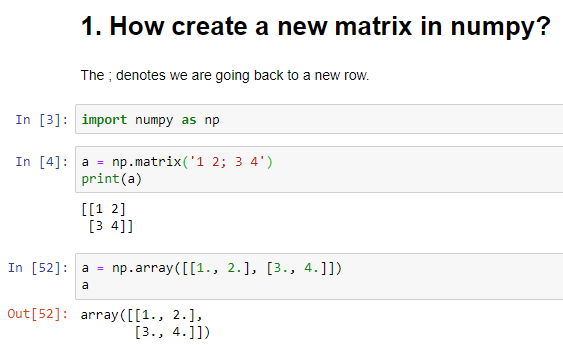


So vectors are a subset of matrices. The above vector is a 4 x 1 matrix.

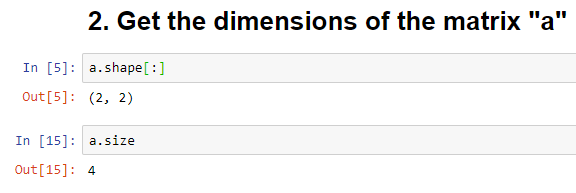
**Notation and terms**

* *Aij*​ refers to the element in the ith row and jth column of matrix A.
* A vector with 'n' rows is referred to as an 'n'-dimensional vector.
* *vi*​ refers to the element in the ith row of the vector.
* In general, all our vectors and matrices will be 1-indexed. Note that for some programming languages, the arrays are 0-indexed.
* 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.
* R refers to the set of scalar real numbers.
* R*n* refers to the set of n-dimensional vectors of real numbers.

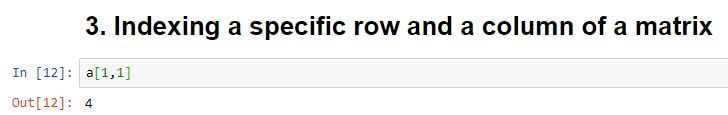
#### Creating a New Matrix in Numpy



#### Computing the Size of a Matrix



#### Indexing a specific row and a column



### [Addition and Scalar Multiplication](https://www.coursera.org/learn/machine-learning/lecture/R4hiJ/addition-and-scalar-multiplication)

Addition and subtraction are **element-wise**, so you simply add or subtract each corresponding element:



Subtracting Matrices:



To add or subtract two matrices, their dimensions must be **the same**.

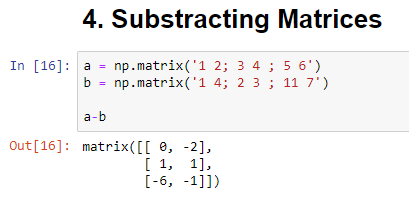
In scalar multiplication, we simply multiply every element by the scalar value:



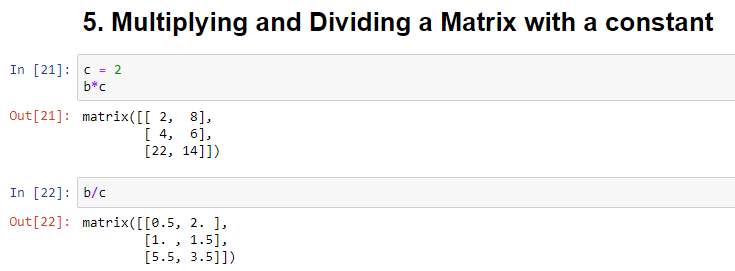
In scalar division, we simply divide every element by the scalar value:



#### Substracting Matrices

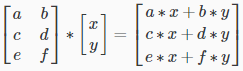


#### Multiplying and Dividing a Matrix with a Constant



### [Matrix-Vector Multiplication](https://www.coursera.org/learn/machine-learning/lecture/aQDta/matrix-vector-multiplication)

We map the column of the vector onto each row of the matrix, multiplying each element and summing the result.

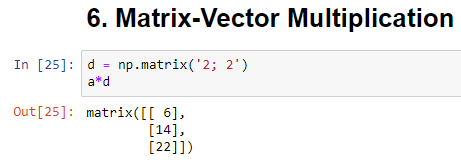


The result is a **vector**. The number of **columns** of the matrix must equal the number of **rows** of the vector.

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

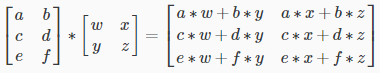
Below is an example of a matrix-vector multiplication. Make sure you understand how the multiplication works. Feel free to try different matrix-vector multiplications.

#### Multiplying a Matrix with a Vector



### [Matrix-Matrix Multiplication](https://www.coursera.org/learn/machine-learning/lecture/dpF1j/matrix-matrix-multiplication)

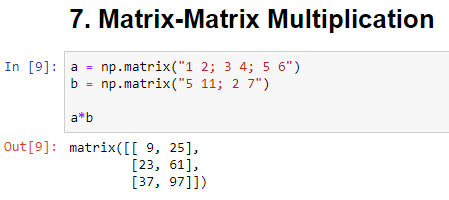
We multiply two matrices by breaking it into several vector multiplications and concatenating the result.



An **m x n matrix** multiplied by an **n x o matrix** results in an **m x o matrix**. In the above 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.

#### Multiplying a Matrix with a Matrix



### [Matrix Multiplication Properties](https://www.coursera.org/learn/machine-learning/lecture/W1LNU/matrix-multiplication-properties)

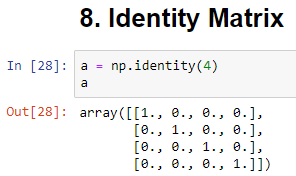
* Matrices are not commutative: *A*∗*B*≠*B*∗*A*
* Matrices are associative: (*A*∗*B*)∗*C*=*A*∗(*B*∗*C*)

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.

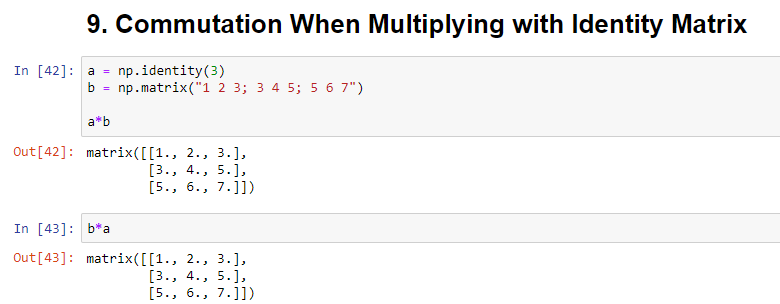


When multiplying the identity matrix after some matrix (A∗I), the square identity matrix's dimension should match the other matrix's **columns**. When multiplying the identity matrix before some other matrix (I∗A), the square identity matrix's dimension should match the other matrix's **rows**.

#### Identity Matrix

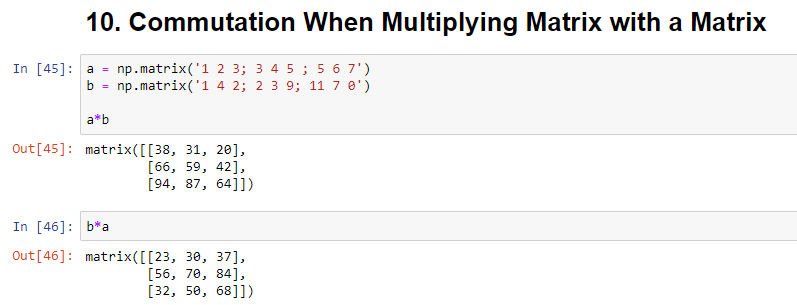


#### Commutation When Multiplying with Identity Matrix



#### Commutation When Multiplying Matrix with a Matrix

As can be seen, the results of a\*b and b\*a are different.

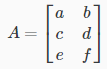


### [Inverse and Transpose](https://www.coursera.org/learn/machine-learning/lecture/FuSWY/inverse-and-transpose)

The **inverse** of a matrix A is denoted *A*−1. Multiplying by the inverse results in the identity matrix.

A non square matrix does not have an inverse matrix.

The **transposition** of a matrix is like rotating the matrix 90**°** in clockwise direction and then reversing it.



In other words:

*Aij*​=*AjiT*​

#### Inverse and Transpose of a Matrix

