# What is Machine Learning?

Two definitions of Machine Learning are offered. Arthur Samuel described it as: "the field of study that gives computers the ability to learn without being explicitly programmed." This is an older, informal definition.

Tom Mitchell provides a more modern definition: "A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in 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, OR
* unsupervised learning.

# 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 acontinuous 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. Here is a description on Math is Fun on Continuous and Discrete Data.

Example 1:

Given data about the size of houses on the real estate market, try to predict their price. Price as a function of size is a continuous output, so this is a regression problem.

We could turn this example into a classification problem by instead making our output about whether the house "sells for more or less than the asking price." Here we are classifying the houses based on price into two discrete categories.

Example 2: (a)Regression - Given a picture of Male/Female, We have to predict his/her age on the basis of given picture. (b)Classification - Given a picture of Male/Female, We have to predict Whether He/She is of High school, College, Graduate age. Another Example for Classification - Banks have to decide whether or not to give a loan to someone on the basis of his credit history.

# Unsupervised Learning

Unsupervised learning, on the other hand, 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, i.e., there is no teacher to correct you.

Example:

Clustering: Take a collection of 1000 essays written on the US Economy, and find a way to automatically group these essays into a small number that are somehow similar or related by different variables, such as word frequency, sentence length, page count, and so on.

# **ML:Linear Regression with One Variable**

# **Model Representation**

Recall that in regression problems, we are taking input variables and trying to fit the output onto a continuous expected result function.

Linear regression with one variable is also known as "univariate linear regression."

Univariate linear regression is used when you want to predict a single output value *y* from a single input value *x*. We're doing supervised learning here, so that means we already have an idea about what the input/output cause and effect should be.

# The Hypothesis Function

Our hypothesis function has the general form:

*y*ˆ=*h**θ*(*x*)=*θ*0+*θ*1*x*

Note that this is like the equation of a straight line. We give to *h**θ*(*x*) values for *θ*0 and *θ*1 to get our estimated output *y*ˆ. In other words, we are trying to create a function called *h**θ* that is trying to map our input data (the x's) to our output data (the y's).

### **Example:**

Suppose we have the following set of training data:

|  |  |
| --- | --- |
| input x | output y |
| 0 | 4 |
| 1 | 7 |
| 2 | 7 |
| 3 | 8 |

Now we can make a random guess about our *h**θ* function: *θ*0=2 and *θ*1=2. The hypothesis function becomes *h**θ*(*x*)=2+2*x*.

So for input of 1 to our hypothesis, y will be 4. This is off by 3. Note that we will be trying out various values of *θ*0 and *θ*1 to try to find values which provide the best possible "fit" or the most representative "straight line" through the data points mapped on the x-y plane.

# Cost Function

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

*J*(*θ*0,*θ*1)=12*m*∑*i*=1*m*(*y*ˆ*i*−*y**i*)2=12*m*∑*i*=1*m*(*h**θ*(*x**i*)−*y**i*)2

To break it apart, it is 12*x*ˉ where *x*ˉ is the mean of the squares of *h**θ*(*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 (12*m*) as a convenience for the computation of the gradient descent, as the derivative term of the square function will cancel out the 12 term.

Now we are able to concretely measure the accuracy of our predictor function against the correct results we have so that we can predict new results we don't have.

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 straight line (defined by *h**θ*(*x*)) which passes through this scattered set of data. 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. In the best case, the line should pass through all the points of our training data set. In such a case the value of *J*(*θ*0,*θ*1) will be 0.

# ML: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 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). This can be kind of confusing; we are moving up to a higher level of abstraction. We are not graphing x and y itself, but the parameter range of our hypothesis function and the cost resulting from selecting 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.

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 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, and the size of each step is determined by the parameter *α*, which is called the learning rate.

The gradient descent algorithm is:

repeat until convergence:

*θ**j* :=*θ**j*−*α*∂∂*θ**j**J*(*θ*0,*θ*1)

where

*j*=0,1 represents the feature index number.

Intuitively, this could be thought of as:

repeat until convergence:

*θ**j* :=*θ**j*−*α*[Slope of tangent aka derivative in j dimension]

# 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 (the derivation of the formulas are out of the scope of this course, but a really great one can be [found here](https://math.stackexchange.com/questions/70728/partial-derivative-in-gradient-descent-for-two-variables/189792" \l "189792)):

repeat until convergence: {*θ*0 :=*θ*1 :=}*θ*0−*α*1*m*∑*i*=1*m*(*h**θ*(*x**i*)−*y**i*)*θ*1−*α*1*m*∑*i*=1*m*((*h**θ*(*x**i*)−*y**i*)*x**i*)

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

Note that we have separated out the two cases for *θ**j* into separate equations for *θ*0 and *θ*1; and that for *θ*1 we are multiplying *x**i* at the end due to the derivative.

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.

# ML:Linear Regression with Multiple Variables

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.

*x*(*i*)*j**x*(*i*)*m**n*=value of feature *j* in the *i**t**h* training example=the column vector of all the feature inputs of the *i**t**h* training example=the number of training examples=∣∣*x*(*i*)∣∣ ;(the number of features)

Now define the multivariable form of the hypothesis function as follows, accomodating these multiple features:

*h**θ*(*x*)=*θ*0+*θ*1*x*1+*θ*2*x*2+*θ*3*x*3+⋯+*θ**n**x**n*

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:

*h**θ*(*x*)=[*θ*0*θ*1...*θ**n*]⎡⎣⎢⎢⎢⎢*x*0*x*1⋮*x**n*⎤⎦⎥⎥⎥⎥=*θ**T**x*

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 Mr. Ng assumes *x*(*i*)0=1 for (*i*∈1,…,*m*)

[Note: So that we can do matrix operations with theta and x, we will set *x*(*i*)0=1, for all values of *i*. This makes the two vectors 'theta' and *x*(*i*) match each other element-wise (that is, have the same number of elements: *n*+1).]

The training examples are stored in *X* row-wise, like such:

*X*=⎡⎣⎢⎢⎢*x*(1)0*x*(2)0*x*(3)0*x*(1)1*x*(2)1*x*(3)1⎤⎦⎥⎥⎥,*θ*=[*θ*0*θ*1]

You can calculate the hypothesis as a column vector of size (m x 1) with:

*h**θ*(*X*)=*X**θ*

For the rest of this page, and other pages of the wiki, *X* will represent a matrix of training examples *x*(*i*) stored row-wise.

# Cost function

For the parameter vector *θ* (of type ℝ*n*+1 or in ℝ(*n*+1)×1), the cost function is:

*J*(*θ*)=12*m*∑*i*=1*m*(*h**θ*(*x*(*i*))−*y*(*i*))2

The vectorized version is:

*J*(*θ*)=12*m*(*X**θ*−*y*⃗ )*T*(*X**θ*−*y*⃗ )

Where *y*⃗  denotes the vector of all y values.

# 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:

}repeat until convergence:{*θ*0 :=*θ*0−*α*1*m*∑*i*=1*m*(*h**θ*(*x*(*i*))−*y*(*i*))⋅*x*(*i*)0*θ*1 :=*θ*1−*α*1*m*∑*i*=1*m*(*h**θ*(*x*(*i*))−*y*(*i*))⋅*x*(*i*)1*θ*2 :=*θ*2−*α*1*m*∑*i*=1*m*(*h**θ*(*x*(*i*))−*y*(*i*))⋅*x*(*i*)2⋯

In other words:

}repeat until convergence:{*θ**j* :=*θ**j*−*α*1*m*∑*i*=1*m*(*h**θ*(*x*(*i*))−*y*(*i*))⋅*x*(*i*)*j*for j := 0..n

## Matrix Notation

The Gradient Descent rule can be expressed as:

*θ* :=*θ*−*α*∇*J*(*θ*)

Where ∇*J*(*θ*) is a column vector of the form:

∇*J*(*θ*)=⎡⎣⎢⎢⎢⎢⎢∂*J*(*θ*)∂*θ*0∂*J*(*θ*)∂*θ*1⋮∂*J*(*θ*)∂*θ**n*⎤⎦⎥⎥⎥⎥⎥

The j-th component of the gradient is the summation of the product of two terms:

∂*J*(*θ*)∂*θ**j*==1*m*∑*i*=1*m*(*h**θ*(*x*(*i*))−*y*(*i*))⋅*x*(*i*)*j*1*m*∑*i*=1*m**x*(*i*)*j*⋅(*h**θ*(*x*(*i*))−*y*(*i*))

Sometimes, the summation of the product of two terms can be expressed as the product of two vectors.

Here, *x*(*i*)*j*, for *i*=1,...,*m*, represents the *m* elements of the *j*-th column, *x**j*⃗ , of the training set *X*.

The other term (*h**θ*(*x*(*i*))−*y*(*i*)) is the vector of the deviations between the predictions *h**θ*(*x*(*i*)) and the true values *y*(*i*). Re-writing ∂*J*(*θ*)∂*θ**j*, we have:

∂*J*(*θ*)∂*θ**j*∇*J*(*θ*)==1*m**x**j*⃗ *T*(*X**θ*−*y*⃗ )1*m**X**T*(*X**θ*−*y*⃗ )

Finally, the matrix notation (vectorized) of the Gradient Descent rule is:

*θ* :=*θ*−*α**m**X**T*(*X**θ*−*y*⃗ )

# Feature Normalization

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.

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:

*x**i* :=*x**i*−*μ**i**s**i*

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

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

Example: *x**i* is housing prices with range of 100 to 2000, with a mean value of 1000. Then, *x**i* :=*p**r**i**c**e*−10001900.

# Gradient Descent Tips

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. Andrew Ng recommends decreasing *α* by multiples of 3.

# 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.

### **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 is *h**θ*(*x*)=*θ*0+*θ*1*x*1 then we can create additional features based on *x*1, to get the quadratic function *h**θ*(*x*)=*θ*0+*θ*1*x*1+*θ*2*x*21 or the cubic function *h**θ*(*x*)=*θ*0+*θ*1*x*1+*θ*2*x*21+*θ*3*x*31

In the cubic version, we have created new features *x*2 and *x*3 where *x*2=*x*21 and *x*3=*x*31.

To make it a square root function, we could do: *h**θ*(*x*)=*θ*0+*θ*1*x*1+*θ*2*x*1‾‾√

Note that at 2:52 and through 6:22 in the "Features and Polynomial Regression" video, the curve that Prof Ng discusses about "doesn't ever come back down" is in reference to the hypothesis function that uses the sqrt() function (shown by the solid purple line), not the one that uses *s**i**z**e*2 (shown with the dotted blue line). The quadratic form of the hypothesis function would have the shape shown with the blue dotted line if *θ*2 was negative.

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 - 1000 then range of *x*21 becomes 1 - 1000000 and that of *x*31 becomes 1 - 1000000000

# Normal Equation

The "Normal Equation" is a method of finding the optimum theta without iteration.

*θ*=(*X**T**X*)−1*X**T**y*

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

Mathematical proof of the Normal equation requires knowledge of linear algebra and is fairly involved, so you do not need to worry about the details.

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

|  |  |
| --- | --- |
| Gradient Descent | Normal Equation |
| Need to choose alpha | No need to choose alpha |
| Needs many iterations | No need to iterate |
| *O* (*k**n*2) | *O* (*n*3), need to calculate inverse of *X**T**X* |
| Works well when n is large | Slow if n is very large |

With the normal equation, computing the inversion has complexity (*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.

### **Normal Equation Noninvertibility**

When implementing the normal equation in octave we want to use the 'pinv' function rather than 'inv.'

*X**T**X* may be noninvertible. The common causes are:

* 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.

# ML:Logistic Regression

Now we are switching from regression problems to classification problems. Don't be confused by the name "Logistic Regression"; it is named that way for historical reasons and is actually an approach to classification problems, not regression problems.

# Binary Classification

Instead of our output vector *y* being a continuous range of values, it will only be 0 or 1.

*y*∈{0,1}

Where 0 is usually taken as the "negative class" and 1 as the "positive class", but you are free to assign any representation to it.

We're only doing two classes for now, called a "Binary Classification Problem."

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. This method doesn't work well because classification is not actually a linear function.

# Hypothesis Representation

Our hypothesis should satisfy:

0≤*h**θ*(*x*)≤1

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

*h**θ*(*x*)=*g*(*θ**T**x*)*z*=*θ**T**x**g*(*z*)=11+*e*−*z*



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. Try playing with interactive plot of sigmoid function [here](https://www.desmos.com/calculator/bgontvxotm).

We start with our old hypothesis (linear regression), except that we want to restrict the range to 0 and 1. This is accomplished by plugging *θ**T**x* into the Logistic Function.

*h**θ* will give us the probability that our output is 1. For example, *h**θ*(*x*)=0.7 gives us the probability of 70% that our output is 1.

*h**θ*(*x*)=*P*(*y*=1∣∣*x* ;*θ*)=1−*P*(*y*=0∣∣*x* ;*θ*)*P*(*y*=0∣∣*x*;*θ*)+*P*(*y*=1∣∣*x* ;*θ*)=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

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

*h**θ*(*x*)≥0.5→*y*=1*h**θ*(*x*)<0.5→*y*=0

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:

*g*(*z*)≥0.5*w**h**e**n**z*≥0

Remember.-

*z*=0,*e*0=1⇒*g*(*z*)=1/2*z*→∞,*e*−∞→0⇒*g*(*z*)=1*z*→−∞,*e*∞→∞⇒*g*(*z*)=0

So if our input to *g* is *θ**T**X*, then that means:

*h**θ*(*x*)=*g*(*θ**T**x*)≥0.5*w**h**e**n**θ**T**x*≥0

From these statements we can now say:

*θ**T**x*≥0⇒*y*=1*θ**T**x*<0⇒*y*=0

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:

*w**h**e**n**θ*=⎡⎣⎢⎢5−10⎤⎦⎥⎥*y*=1

5+(−1)*x*1+0*x*25−*x*1−*x*1*x*1≥0⇔≥0⇔≥−5⇔≤5

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. *θ**T**X*) doesn't need to be linear, and could be a function that describes a circle (e.g. *z*=*θ*0+*θ*1*x*21+*θ*2*x*22) or any shape to fit our data.

# 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:

*J*(*θ*)=1*m*∑*i*=1*m*Cost(*h**θ*(*x*(*i*)),*y*(*i*))Cost(*h**θ*(*x*),*y*)=−log(*h**θ*(*x*))Cost(*h**θ*(*x*),*y*)=−log(1−*h**θ*(*x*))if y = 1if y = 0

 

The more our hypothesis is off from y, the larger the cost function output. If our hypothesis is equal to y, then our cost is 0:

Cost(*h**θ*(*x*),*y*)=0 if *h**θ*(*x*)=*y*Cost(*h**θ*(*x*),*y*)→∞ if *y*=0and*h**θ*(*x*)→1Cost(*h**θ*(*x*),*y*)→∞ if *y*=1and*h**θ*(*x*)→0

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*(*θ*) is convex for logistic regression.

# Simplified Cost Function and Gradient Descent

We can compress our cost function's two conditional cases into one case:

Cost(*h**θ*(*x*),*y*)=−*y*log(*h**θ*(*x*))−(1−*y*)log(1−*h**θ*(*x*))

Notice that when y is equal to 1, then the second term ((1−*y*)log(1−*h**θ*(*x*))) will be zero and will not affect the result. If y is equal to 0, then the first term (−*y*log(*h**θ*(*x*))) will be zero and will not affect the result.

We can fully write out our entire cost function as follows:

*J*(*θ*)=−1*m*∑*i*=1*m*[*y*(*i*)log(*h**θ*(*x*(*i*)))+(1−*y*(*i*))log(1−*h**θ*(*x*(*i*)))]

A vectorized implementation is:

*h*=*g*(*X**θ*)*J*(*θ*)=1*m*⋅(−*y**T*log(*h*)−(1−*y*)*T*log(1−*h*))

### **Gradient Descent**

Remember that the general form of gradient descent is:

*R**e**p**e**a**t*{*θ**j* :=*θ**j*−*α*∂∂*θ**j**J*(*θ*)}

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

*R**e**p**e**a**t*{*θ**j* :=*θ**j*−*α**m*∑*i*=1*m*(*h**θ*(*x*(*i*))−*y*(*i*))*x*(*i*)*j*}

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:

*θ* :=*θ*−*α**m**X**T*(*g*(*X**θ*)−*y*⃗ )

# Advanced Optimization

"Conjugate gradient", "BFGS", and "L-BFGS" are more sophisticated, faster ways to optimize *θ* that can be used instead of gradient descent. A. Ng suggests not to 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*(*θ*)∂∂*θ**j**J*(*θ*)

We can write a single function that returns both of these:

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()". (Note: the value for MaxIter should be an integer, not a character string - errata in the video at 7:30)

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.

# Multiclass Classification: One-vs-all

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

In this case 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.

*y*∈{0,1...*n*}*h*(0)*θ*(*x*)=*P*(*y*=0∣∣*x* ;*θ*)*h*(1)*θ*(*x*)=*P*(*y*=1∣∣*x* ;*θ*)⋯*h*(*n*)*θ*(*x*)=*P*(*y*=*n*∣∣*x* ;*θ*)prediction=max*i*(*h*(*i*)*θ*(*x*))

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.

# ML:Regularization

# The Problem of Overfitting

Regularization is designed to address the problem of overfitting.

High bias or underfitting is when the form of our hypothesis function *h* maps poorly to the trend of the data. It is usually caused by a function that is too simple or uses too few features. eg. if we take *h**θ*(*x*)=*θ*0+*θ*1*x*1+*θ*2*x*2 then we are making an initial assumption that a linear model will fit the training data well and will be able to generalize but that may not be the case.

At the other extreme, overfitting or high variance is caused by a hypothesis function that fits the available data but does not generalize well to predict new data. It is usually caused by a complicated function that creates a lot of unnecessary curves and angles unrelated to the data.

This terminology is applied to both linear and logistic regression.

There are two main options to address the issue of overfitting:

1. Reduce the number of features.
   * Manually select which features to keep.
   * Use a model selection algorithm (studied later in the course).
2. Regularization
   * Keep all the features, but reduce the parameters *θ**j*.

Regularization works well when we have a lot of slightly useful features.

# Cost Function

If we have overfitting from our hypothesis function, we can reduce the weight that some of the terms in our function carry by increasing their cost.

Say we wanted to make the following function more quadratic:

*θ*0+*θ*1*x*+*θ*2*x*2+*θ*3*x*3+*θ*4*x*4

We'll want to eliminate the influence of *θ*3*x*3 and *θ*4*x*4. Without actually getting rid of these features or changing the form of our hypothesis, we can instead modify ourcost function:

*m**i**n**θ* 12*m*∑*m**i*=1(*h**θ*(*x*(*i*))−*y*(*i*))2+1000⋅*θ*23+1000⋅*θ*24

We've added two extra terms at the end to inflate the cost of *θ*3 and *θ*4. Now, in order for the cost function to get close to zero, we will have to reduce the values of *θ*3 and *θ*4 to near zero. This will in turn greatly reduce the values of *θ*3*x*3 and *θ*4*x*4 in our hypothesis function.

We could also regularize all of our theta parameters in a single summation:

*m**i**n**θ* 12*m* [∑*m**i*=1(*h**θ*(*x*(*i*))−*y*(*i*))2+*λ* ∑*n**j*=1*θ*2*j*]

The *λ*, or lambda, is the regularization parameter. It determines how much the costs of our theta parameters are inflated. You can visualize the effect of regularization in this interactive plot [here](https://www.desmos.com/calculator/1hexc8ntqp)

Using the above cost function with the extra summation, we can smooth the output of our hypothesis function to reduce overfitting. If lambda is chosen to be too large, it may smooth out the function too much and cause underfitting.

# Regularized Linear Regression

We can apply regularization to both linear regression and logistic regression. We will approach linear regression first.

### **Gradient Descent**

We will modify our gradient descent function to separate out *θ*0 from the rest of the parameters because we do not want to penalize *θ*0.

Repeat {    *θ*0 :=*θ*0−*α* 1*m* ∑*i*=1*m*(*h**θ*(*x*(*i*))−*y*(*i*))*x*(*i*)0    *θ**j* :=*θ**j*−*α* [(1*m* ∑*i*=1*m*(*h**θ*(*x*(*i*))−*y*(*i*))*x*(*i*)*j*)+*λ**m**θ**j*]}          *j*∈{1,2...*n*}

The term *λ**m**θ**j* performs our regularization.

With some manipulation our update rule can also be represented as:

*θ**j* :=*θ**j*(1−*α**λ**m*)−*α*1*m*∑*m**i*=1(*h**θ*(*x*(*i*))−*y*(*i*))*x*(*i*)*j*

The first term in the above equation, 1−*α**λ**m* will always be less than 1. Intuitively you can see it as reducing the value of *θ**j* by some amount on every update.

Notice that the second term is now exactly the same as it was before.

### **Normal Equation**

Now let's approach regularization using the alternate method of the non-iterative normal equation.

To add in regularization, the equation is the same as our original, except that we add another term inside the parentheses:

*θ*=(*X**T**X*+*λ*⋅*L*)−1*X**T**y*where  *L*=⎡⎣⎢⎢⎢⎢011⋱1⎤⎦⎥⎥⎥⎥

*L* is a matrix with 0 at the top left and 1's down the diagonal, with 0's everywhere else. It should have dimension (*n*+1)×(*n*+1). Intuitively, this is the identity matrix (though we are not including *x*0), multiplied with a single real number *λ*.

Recall that if *m*≤*n*, then *X**T**X* is non-invertible. However, when we add the term *λ*⋅*L*, then *X**T**X* + *λ*⋅*L* becomes invertible.

# Regularized Logistic Regression

We can regularize logistic regression in a similar way that we regularize linear regression. Let's start with the cost function.

### **Cost Function**

Recall that our cost function for logistic regression was:

*J*(*θ*)=−1*m*∑*m**i*=1[*y*(*i*) log(*h**θ*(*x*(*i*)))+(1−*y*(*i*)) log(1−*h**θ*(*x*(*i*)))]

We can regularize this equation by adding a term to the end:

*J*(*θ*)=−1*m*∑*m**i*=1[*y*(*i*) log(*h**θ*(*x*(*i*)))+(1−*y*(*i*)) log(1−*h**θ*(*x*(*i*)))]+*λ*2*m*∑*n**j*=1*θ*2*j*

Note Well: The second sum, ∑*n**j*=1*θ*2*j* means to explicitly exclude the bias term, *θ*0. I.e. the *θ* vector is indexed from 0 to *n* (holding *n*+1 values, *θ*0 through *θ**n*), and this sum explicitly skips *θ*0, by running from 1 to *n*, skipping 0.

### **Gradient Descent**

Just like with linear regression, we will want to separately update *θ*0 and the rest of the parameters because we do not want to regularize *θ*0.

Repeat {    *θ*0 :=*θ*0−*α* 1*m* ∑*i*=1*m*(*h**θ*(*x*(*i*))−*y*(*i*))*x*(*i*)0    *θ**j* :=*θ**j*−*α* [(1*m* ∑*i*=1*m*(*h**θ*(*x*(*i*))−*y*(*i*))*x*(*i*)*j*)+*λ**m**θ**j*]}  *j*∈{1,2...*n*}

This is identical to the gradient descent function presented for linear regression.

# ML:Anomaly Detection

# Problem Motivation

Just like in other learning problems, we are given a dataset *x*(1),*x*(2),…,*x*(*m*).

We are then given a new example, *x**t**e**s**t*, and we want to know whether this new example is abnormal/anomalous.

We define a "model" *p*(*x*) that tells us the probability the example is not anomalous. We also use a threshold *ϵ* (epsilon) as a dividing line so we can say which examples are anomalous and which are not.

A very common application of anomaly detection is detecting fraud:

*x*(*i*)= features of user *i*'s activities

Model *p*(*x*) from the data.

Identify unusual users by checking which have *p*(*x*)<*ϵ*.

If our anomaly detector is flagging too many anomalous examples, then we need to decrease our threshold *ϵ*

# Gaussian Distribution

The Gaussian Distribution is a familiar bell-shaped curve that can be described by a function (*μ*,*σ*2)

Let *x*∈ℝ. If the probability distribution of *x* is Gaussian with mean *μ*, variance *σ*2, then:

*x*∼(*μ*,*σ*2)

The little ∼ or 'tilde' can be read as "distributed as."

The Gaussian Distribution is parameterized by a mean and a variance.

Mu, or *μ*, describes the center of the curve, called the mean. The width of the curve is described by sigma, or *σ*, called the standard deviation.

The full function is as follows:

*p*(*x*;*μ*,*σ*2)=1*σ*(2*π*)‾‾‾‾√*e*−12(*x*−*μ**σ*)2

We can estimate the parameter *μ* from a given dataset by simply taking the average of all the examples:

*μ*=1*m*∑*i*=1*m**x*(*i*)

We can estimate the other parameter, *σ*2, with our familiar squared error formula:

*σ*2=1*m*∑*i*=1*m*(*x*(*i*)−*μ*)2

# Algorithm

Given a training set of examples, {*x*(1),…,*x*(*m*)} where each example is a vector, *x*∈ℝ*n*.

*p*(*x*)=*p*(*x*1;*μ*1,*σ*21)*p*(*x*2;*μ*2,*σ*22)⋯*p*(*x**n*;*μ**n*,*σ*2*n*)

In statistics, this is called an "independence assumption" on the values of the features inside training example *x*.

More compactly, the above expression can be written as follows:

=∏*j*=1*n**p*(*x**j*;*μ**j*,*σ*2*j*)

The algorithm

Choose features *x**i* that you think might be indicative of anomalous examples.

Fit parameters *μ*1,…,*μ**n*,*σ*21,…,*σ*2*n*.

Calculate *μ**j*=1*m*∑*i*=1*m**x*(*i*)*j*

Calculate *σ*2*j*=1*m*∑*i*=1*m*(*x*(*i*)*j*−*μ**j*)2

Given a new example *x*, compute *p*(*x*):

*p*(*x*)=∏*j*=1*n**p*(*x**j*;*μ**j*,*σ*2*j*)=∏*j*=1*n*12*π*‾‾‾√*σ**j**e**x**p*(−(*x**j*−*μ**j*)22*σ*2*j*)

Anomaly if *p*(*x*)<*ϵ*

A vectorized version of the calculation for *μ* is *μ*=1*m*∑*i*=1*m**x*(*i*). You can vectorize *σ*2 similarly.

# Developing and Evaluating an Anomaly Detection System

To evaluate our learning algorithm, we take some labeled data, categorized into anomalous and non-anomalous examples (*y*=0 if normal, *y*=1 if anomalous).

Among that data, take a large proportion of good, non-anomalous data for the training set on which to train *p*(*x*).

Then, take a smaller proportion of mixed anomalous and non-anomalous examples (you will usually have many more non-anomalous examples) for your cross-validation and test sets.

For example, we may have a set where 0.2% of the data is anomalous. We take 60% of those examples, all of which are good (*y*=0) for the training set. We then take 20% of the examples for the cross-validation set (with 0.1% of the anomalous examples) and another 20% from the test set (with another 0.1% of the anomalous).

In other words, we split the data 60/20/20 training/CV/test and then split the anomalous examples 50/50 between the CV and test sets.

Algorithm evaluation:

Fit model *p*(*x*) on training set {*x*(1),…,*x*(*m*)}

On a cross validation/test example *x*, predict:

If *p*(*x*)<*ϵ* (anomaly), then *y*=1

If *p*(*x*)≥*ϵ* (normal), then *y*=0

Possible evaluation metrics (see "Machine Learning System Design" section):

* True positive, false positive, false negative, true negative.
* Precision/recall
* *F*1 score

Note that we use the cross-validation set to choose parameter *ϵ*

# Anomaly Detection vs. Supervised Learning

When do we use anomaly detection and when do we use supervised learning?

Use anomaly detection when...

* We have a very small number of positive examples (*y*=1 ... 0-20 examples is common) and a large number of negative (*y*=0) examples.
* We have many different "types" of anomalies and it is hard for any algorithm to learn from positive examples what the anomalies look like; future anomalies may look nothing like any of the anomalous examples we've seen so far.

Use supervised learning when...

* We have a large number of both positive and negative examples. In other words, the training set is more evenly divided into classes.
* We have enough positive examples for the algorithm to get a sense of what new positives examples look like. The future positive examples are likely similar to the ones in the training set.

# Choosing What Features to Use

The features will greatly affect how well your anomaly detection algorithm works.

We can check that our features are gaussian by plotting a histogram of our data and checking for the bell-shaped curve.

Some transforms we can try on an example feature *x* that does not have the bell-shaped curve are:

* *l**o**g*(*x*)
* *l**o**g*(*x*+1)
* *l**o**g*(*x*+*c*) for some constant
* *x*√
* *x*1/3

We can play with each of these to try and achieve the gaussian shape in our data.

There is an error analysis procedure for anomaly detection that is very similar to the one in supervised learning.

Our goal is for *p*(*x*) to be large for normal examples and small for anomalous examples.

One common problem is when *p*(*x*) is similar for both types of examples. In this case, you need to examine the anomalous examples that are giving high probability in detail and try to figure out new features that will better distinguish the data.

In general, choose features that might take on unusually large or small values in the event of an anomaly.

# ML:Recommender Systems

# Problem Formulation

Recommendation is currently a very popular application of machine learning.

Say we are trying to recommend movies to customers. We can use the following definitions

*n**u*= number of users

*n**m*= number of movies

*r*(*i*,*j*)=1 if user *j* has rated movie *i*

*y*(*i*,*j*)= rating given by user *j* to movie *i* (defined only if *r*(*i*,*j*)=1)

# Content Based Recommendations

We can introduce two features, *x*1 and *x*2 which represents how much romance or how much action a movie may have (on a scale of 0−1).

One approach is that we could do linear regression for every single user. For each user *j*, learn a parameter *θ*(*j*)∈ℝ3. Predict user *j* as rating movie *i* with (*θ*(*j*))*T**x*(*i*)stars.

*θ*(*j*)= parameter vector for user *j*

*x*(*i*)= feature vector for movie *i*

For user *j*, movie *i*, predicted rating: (*θ*(*j*))*T*(*x*(*i*))

*m*(*j*)= number of movies rated by user *j*

To learn *θ*(*j*), we do the following

*m**i**n**θ*(*j*)=12∑*i*:*r*(*i*,*j*)=1((*θ*(*j*))*T*(*x*(*i*))−*y*(*i*,*j*))2+*λ*2∑*k*=1*n*(*θ*(*j*)*k*)2

This is our familiar linear regression. The base of the first summation is choosing all *i* such that *r*(*i*,*j*)=1.

To get the parameters for all our users, we do the following:

*m**i**n**θ*(1),…,*θ*(*n**u*)=12∑*j*=1*n**u*∑*i*:*r*(*i*,*j*)=1((*θ*(*j*))*T*(*x*(*i*))−*y*(*i*,*j*))2+*λ*2∑*j*=1*n**u*∑*k*=1*n*(*θ*(*j*)*k*)2

We can apply our linear regression gradient descent update using the above cost function.

The only real difference is that we eliminate the constant 1*m*.

# Collaborative Filtering

It can be very difficult to find features such as "amount of romance" or "amount of action" in a movie. To figure this out, we can use feature finders.

We can let the users tell us how much they like the different genres, providing their parameter vector immediately for us.

To infer the features from given parameters, we use the squared error function with regularization over all the users:

*m**i**n**x*(1),…,*x*(*n**m*)12∑*i*=1*n**m*∑*j*:*r*(*i*,*j*)=1((*θ*(*j*))*T**x*(*i*)−*y*(*i*,*j*))2+*λ*2∑*i*=1*n**m*∑*k*=1*n*(*x*(*i*)*k*)2

You can also randomly guess the values for theta to guess the features repeatedly. You will actually converge to a good set of features.

# Collaborative Filtering Algorithm

To speed things up, we can simultaneously minimize our features and our parameters:

*J*(*x*,*θ*)=12∑(*i*,*j*):*r*(*i*,*j*)=1((*θ*(*j*))*T**x*(*i*)−*y*(*i*,*j*))2+*λ*2∑*i*=1*n**m*∑*k*=1*n*(*x*(*i*)*k*)2+*λ*2∑*j*=1*n**u*∑*k*=1*n*(*θ*(*j*)*k*)2

It looks very complicated, but we've only combined the cost function for theta and the cost function for x.

Because the algorithm can learn them itself, the bias units where *x*0=1 have been removed, therefore *x*∈ℝ*n* and *θ*∈ℝ*n*.

These are the steps in the algorithm:

1. Initialize *x*(*i*),...,*x*(*n**m*),*θ*(1),...,*θ*(*n**u*) to small random values. This serves to break symmetry and ensures that the algorithm learns features *x*(*i*),...,*x*(*n**m*) that are different from each other.
2. Minimize *J*(*x*(*i*),...,*x*(*n**m*),*θ*(1),...,*θ*(*n**u*)) using gradient descent (or an advanced optimization algorithm).  
   E.g. for every *j*=1,...,*n**u*,*i*=1,...*n**m*:  
   *x*(*i*)*k* :=*x*(*i*)*k*−*α*⎛⎝⎜⎜∑*j*:*r*(*i*,*j*)=1((*θ*(*j*))*T**x*(*i*)−*y*(*i*,*j*))*θ*(*j*)*k*+*λ**x*(*i*)*k*⎞⎠⎟⎟  
   *θ*(*j*)*k* :=*θ*(*j*)*k*−*α*⎛⎝⎜⎜∑*i*:*r*(*i*,*j*)=1((*θ*(*j*))*T**x*(*i*)−*y*(*i*,*j*))*x*(*i*)*k*+*λ**θ*(*j*)*k*⎞⎠⎟⎟
3. For a user with parameters *θ* and a movie with (learned) features *x*, predict a star rating of *θ**T**x*.

# Vectorization: Low Rank Matrix Factorization

Given matrices *X* (each row containing features of a particular movie) and Θ (each row containing the weights for those features for a given user), then the full matrix *Y* of all predicted ratings of all movies by all users is given simply by: *Y*=*X*Θ*T*.

Predicting how similar two movies *i* and *j* are can be done using the distance between their respective feature vectors *x*. Specifically, we are looking for a small value of ∣∣∣∣*x*(*i*)−*x*(*j*)∣∣∣∣.

# Implementation Detail: Mean Normalization

If the ranking system for movies is used from the previous lectures, then new users (who have watched no movies), will be assigned new movies incorrectly. Specifically, they will be assigned *θ* with all components equal to zero due to the minimization of the regularization term. That is, we assume that the new user will rank all movies 0, which does not seem intuitively correct.

We rectify this problem by normalizing the data relative to the mean. First, we use a matrix Y to store the data from previous ratings, where the *i*th row of Y is the ratings for the *i*th movie and the *j*th column corresponds to the ratings for the *j*th user.

We can now define a vector

*μ*=[*μ*1,*μ*2,…,*μ**n**m*]

such that

*μ**i*=∑*j*:*r*(*i*,*j*)=1*Y**i*,*j*∑*j**r*(*i*,*j*)

Which is effectively the mean of the previous ratings for the *i*th movie (where only movies that have been watched by users are counted). We now can normalize the data by subtracting *u*, the mean rating, from the actual ratings for each user (column in matrix *Y*):

As an example, consider the following matrix *Y* and mean ratings *μ*:

*Y*=⎡⎣⎢⎢⎢54005 ?000 ?550040⎤⎦⎥⎥⎥,*μ*=⎡⎣⎢⎢⎢2.522.251.25⎤⎦⎥⎥⎥

The resulting *Y*′ vector is:

*Y*′=⎡⎣⎢⎢⎢2.52−.2.25−1.252.5 ?−2.25−1.25−2.5 ?3.753.75−2.5−21.25−1.25⎤⎦⎥⎥⎥

Now we must slightly modify the linear regression prediction to include the mean normalization term:

(*θ*(*j*))*T**x*(*i*)+*μ**i*

Now, for a new user, the initial predicted values will be equal to the *μ* term instead of simply being initialized to zero, which is more accurate.