1 Introduction to Machine Learning

` Machine learning is a program that, broadly speaking, allows us to "teach" computers how to perform tasks by providing examples of how they should be done. For example, suppose we wish to write a program to distinguish between valid email messages and unwanted spam. We could try to write a set of simple rules, for example, flagging messages that contain certain features (such as the word "lottery" or obviously-fake headers). However, writing rules to accurately distinguish which text is valid can actually be quite difficult to do well, resulting either in many missed spam messages, or, worse, many lost emails. Worse, the spammers will actively adjust the way they send spam in order to trick these strategies (e.g., writing "lØttěry"). Writing effective rules — and keeping them up-to-date — quickly becomes an insurmountable task. Fortunately, machine learning has provided a solution. Modern spam filters are "learned" from examples: we provide the learning algorithm with example emails which we have manually labeled as "ham" (valid email) or "spam" (unwanted email), and the algorithms learn to distinguish between them automatically. Machine learning is a diverse and exciting field, and there are multiple ways of defining it:

1. **The Artificial Intelligence View.** Learning is central to human knowledge and intelligence, and, likewise, it is also essential for building intelligent machines. Years of effort in AI have shown that trying to build intelligent computers by programming all the rules cannot be done; automatic learning is crucial. For example, we humans are not born with the ability to understand language — we learn it — and it makes sense to try to have computers learn language instead of trying to program it all it.
2. **The Software Engineering View.** Machine learning allows us to program computers by example, which can be easier than writing code the traditional way.
3. **The Statistics View.** Machine learning is the union of computer science and statistics: computational techniques are applied to statistical problems. Machine learning has been applied to a vast number of problems in many contexts, beyond the typical statistics problems. Machine learning is often designed with different considerations than statistics (e.g., speed is often more important than accuracy).

Often, machine learning methods are broken into two phases:

1. **Training:** A model is learned from a collection of **training data**.

2. **Application:** The model is used to make decisions about some new **test data**.

For example, in the spam filtering case, the training data constitutes email messages labeled as ham or spam, and each new email message that we receive (and which to classify) is test data. However, there are other ways in which machine learning is used as well.

Types of Machine Learning

Some of the main types of machine learning are:

1. Supervised Learning, in which the training data is labeled with the correct answers, e.g., "spam" or "ham." The two most common types of supervised learning are classification (where the outputs are discrete labels, as in spam filtering) and regression (where the outputs are real-valued).
2. Unsupervised learning, in which we are given a collection of unlabeled data, which we wish to analyze and discover patterns within. The two most important examples are dimension reduction and clustering.
3. Reinforcement learning, in which an agent (e.g., a robot or controller) seeks to learn the optimal actions to take based the outcomes of past actions.

There are many other types of machine learning as well, for example:

1. Semi-supervised learning, in which only a subset of the training data is labeled
2. Time-series forecasting, such as in financial markets
3. Anomaly detection such as used for fault-detection in factories and in surveillance
4. Active learning, in which obtaining data is expensive, and so an algorithm must determine which training data to acquire and many others.

Model building

There are infinitely many curves that fit the given data, and, because the data might be noisy, we might not even want to fit the data precisely. Hence, machine learning requires that we make certain choices in building model [Ref slides also]:

1. How do we parameterize the model we fit? How do we parameterize the curve; should we try to explain the data with a linear function, a quadratic, or a sinusoidal curve?
2. What criteria (e.g., objective function) do we use to judge the quality of the fit? For example, when fitting a curve to noisy data, it is common to measure the quality of the fit in terms of the squared error between the data we are given and the fitted curve. When minimizing the squared error, the resulting fit is usually called a least-squares estimate.
3. Some types of models and some model parameters can be very expensive to optimize well. How long are we willing to wait for a solution, or can we use approximations (or hand tuning) instead?
4. Ideally we want to find a model that will provide useful predictions in future situations. That is, although we might learn a model from training data, we ultimately care about how well it works on future test data. When a model fits training data well, but performs poorly on test data, we say that the model has over fit the training data; i.e., the model has fit properties of the input that are not particularly relevant to the task at hand .Such properties are referred to as noise. When this happens we say that the model does not generalize well to the test data. Rather it produces predictions on the test data that are much less accurate than you might have hoped for given the fit to the training data.

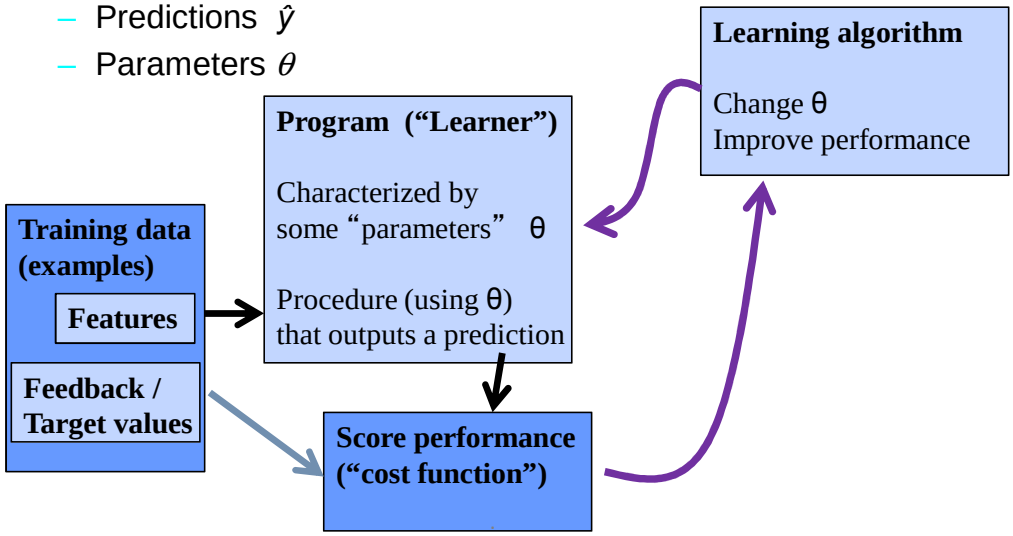
**** [Chapter 1 Ref all slides- examples, types of learning, learning system, decision trees, model building]**

Supervised Learning
Linear Regression, Logistic Regression

The goal in supervised learning is to make *predictions from data*. For example, one popular application of supervised learning is email spam filtering. Here, an email (the data instance) needs to be classified as *spam* or *not-spam*. Following the approach of traditional computer science, one might be tempted to write a carefully designed program that follows some rules to decide if an email is spam or not. Although such a program might work reasonably well for a while, it has significant draw-backs. As email spam changes it would have to be re-written. Spammers could attempt to reverse engineer the software and design messages that circumvent it. And even if it is successful, it could probably not easily be applied to different languages. Machine Learning uses a different approach to generate a program that can make predictions from data. Instead of programming it by hand it is *learned* from past data. This process works if we have data instances for which we know exactly what the right prediction would have been. For example past data might be user-annotated as spam or not-spam. A machine learning algorithm can utilize such data to learn a program, a *classifier*, to predict the correct *label* of each annotated data instance. Other successful applications of machine learning include web-search ranking (predict which web-page the user will click on based on his/her search query), placing of online advertisements (predict the expected revenue of an ad, when placed on a homepage, which is seen by a specific user), visual object recognition (predict which object is in an image - e.g. a camera mounted on a self-driving car), face-detection (predict if an image patch contains a human face or not).

Supervised learning

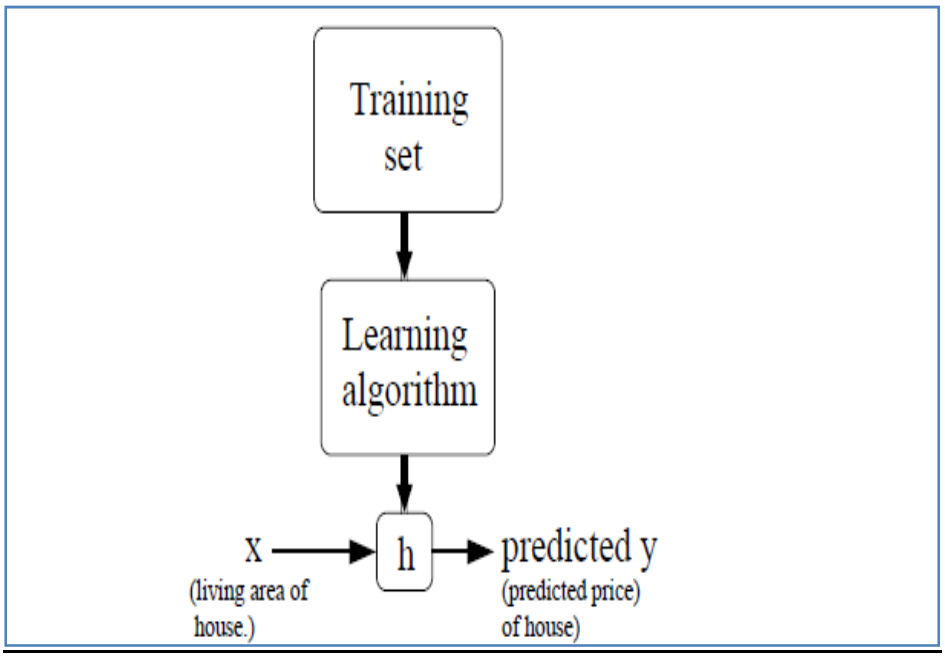
- Notation
  - Features x
  - Targets y
  - Predictions $\hat{y}$
  - Parameters θ



Recall our supervised learning paradigm – we have some observed features "x" which we would like to use to predict a target value "y". We create a flexible function, called the "learner", whose input/output relationships can be adjusted through parameters theta. Then, a collection of training examples are used to fit the parameters until the learner is able to make good predictions on those examples.

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

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



Linear Regression for Machine Learning

Regression is a means of predicting a dependent variable based on one or more independent variables.

This is done by fitting a line or surface to the data points
that minimizes the total error.

The line or surface is called the regression model

The goals are:

- The linear regression model used to create representation and learning algorithms

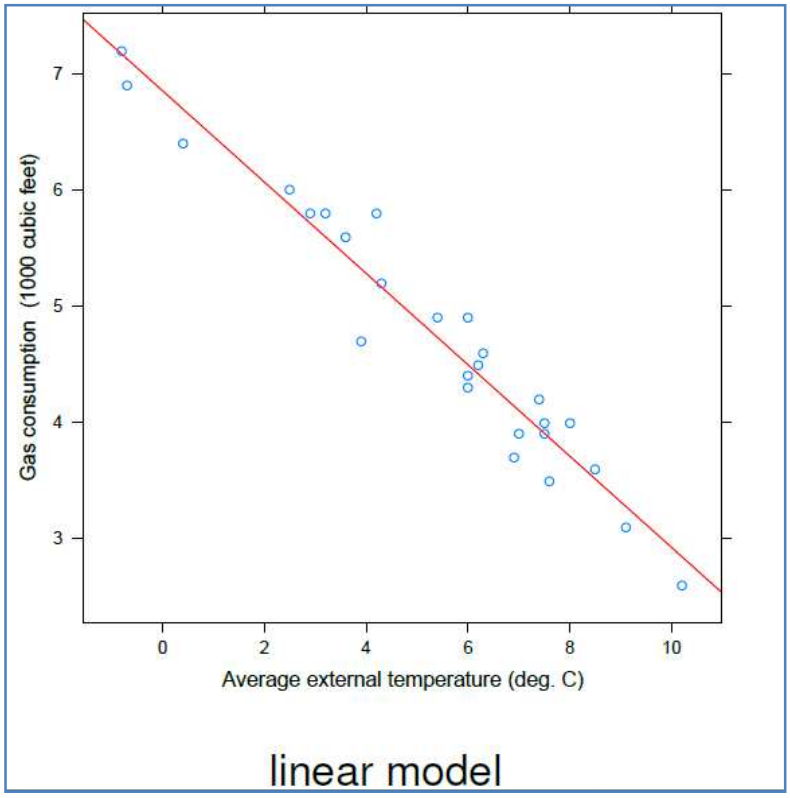
- How to best prepare data when modeling using linear regression.

Machine learning, more specifically the field of predictive modeling is primarily concerned with minimizing the error of a model or making the most accurate predictions possible, at the expense of explainability. As such, linear regression was developed in the field of statistics and is studied as a model for understanding the relationship between input and output numerical variables, but has been borrowed by machine learning. It is both a statistical algorithm and a machine learning algorithm.

Linear regression is a **linear model**, e.g. a model that assumes a linear relationship between the input variables (x) and the single output variable (y). More specifically, that y can be calculated from a linear combination of the input variables (x).

When there is a single input variable (x), the method is referred to as **simple linear regression**. When there are **multiple input variables**, literature from statistics often refers to the method as multiple linear regression.

Ex : Example: how does gas consumption depend on external temperature?



Variable Types

The most common variable types:

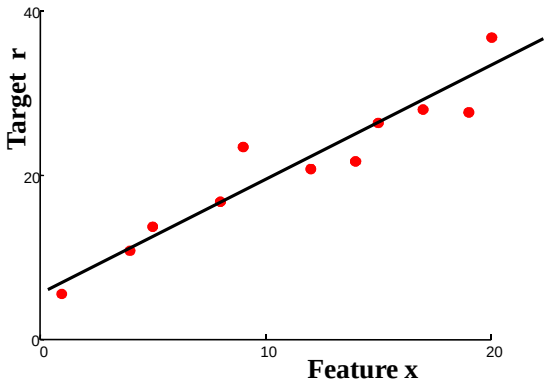
numerical / interval-scaled / quantitative where differences and quotients etc. are meaningful, usually with domain, Y is X := R,
e.g., temperature, size, weight.

nominal / discret / categorical / qualitative / factor where differences and quotients are not defined, usually with a finite, enumerated domain,
e.g., X := {red, green, blue}
or X := {a, b, c, . . . , y, z}.

ordinal / ordered categorical where levels are ordered, but differences and quotients are not defined, usually with a finite, enumerated domain,
e.g., X := {small, medium, large}

If Y is numerical, the task is called **regression**.
If Y is nominal, the task is called **classification**.

Linear regression



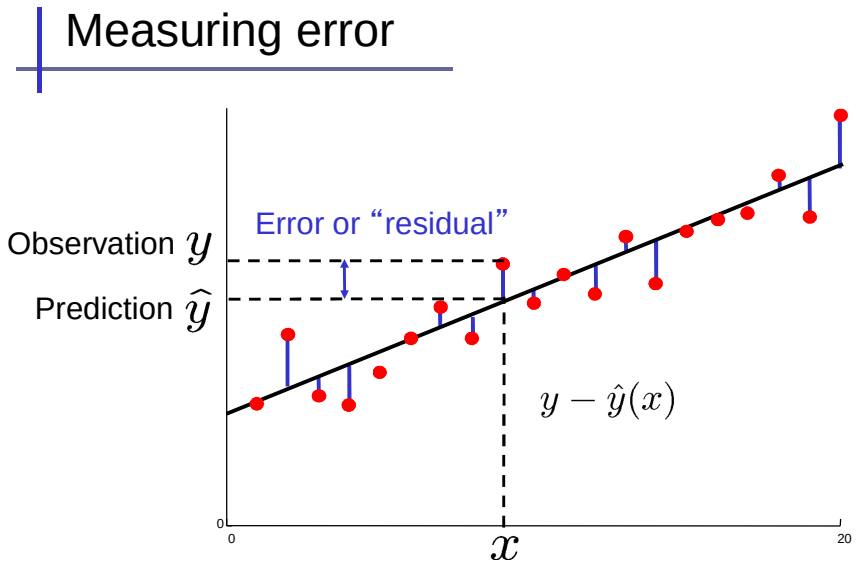
"Predictor":
Evaluate line:

$$r = \theta_0 + \theta_1 x_1$$

return r

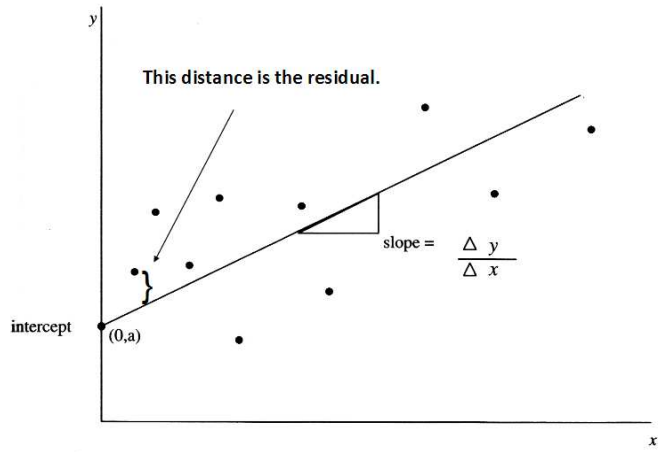
- Define form of function f(x) explicitly
- Find a good f(x) within that family

In linear regression, our learner consists of an explicit functional form f(x): in other words, we define a family of functions parameterized by theta, and any particular value of theta defines a member of that family. Learning then finds a good member within that family (good value of theta) using the training dataset D



To start with Linear Regression, you must be aware of a few basic concepts of statistics. i.e,

- Correlation (r) – Explains the relationship between two variables, possible values -1 to +1
- Variance (σ2)– Measure of spread in your data
- Standard Deviation (σ) – Measure of spread in your data (Square root of Variance)
- Normal distribution
- Residual (error term) – {Actual value – Predicted value}



Each observation of the dependent (y) variable may be expressed as the predicted value + a residual (error).

Error – the difference between the true value and the predicted value. The way we determine which line (there are an infinite number of potential lines) is the best fit is easy. We need to define a line that passes through the point determined by the mean x value and the mean y value. The slope of this line needs to minimize the residual error

Fitting the curve

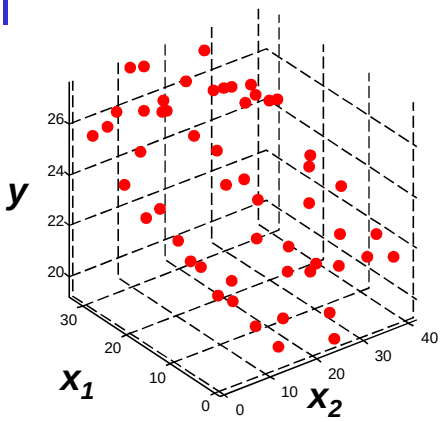
Key Points:

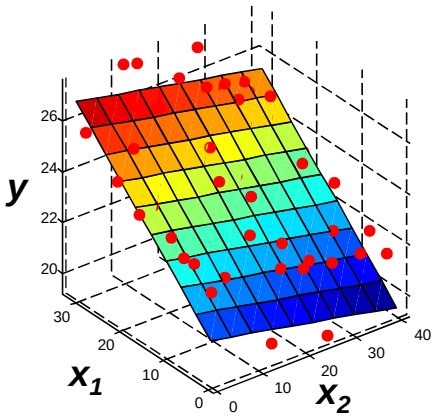
We are trying to fit a line through a set of plotted points that minimizes the residuals (errors).

This line is called the line of best fit

We fit this line in such a way that the sum of the squared residuals is minimized

More dimensions?





$$\hat{y}(x) = \underline{\theta} \cdot \underline{x}^T$$

$$\underline{\theta} = [\theta_0 \ \theta_1 \ \theta_2]$$
$$\underline{x} = [1 \ x_1 \ x_2]$$

In a multidimensional feature space, our linear regression remains a linear function – so, a plane for two features, a hyper plane for more. Otherwise, everything remains essentially the same.

Notation

$$\hat{y}(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \ldots$$

Define "feature" $x_0 = 1$ (constant)
Then

$$\hat{y}(x) = \theta \, x^T$$

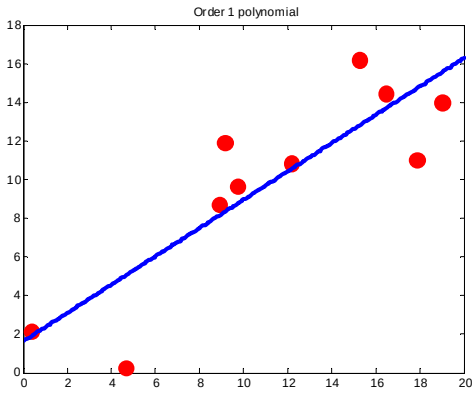
$$\underline{\theta} = [\theta_0, \ldots, \theta_n]$$
$$\underline{x} = [1, x_1, \ldots, x_n]$$

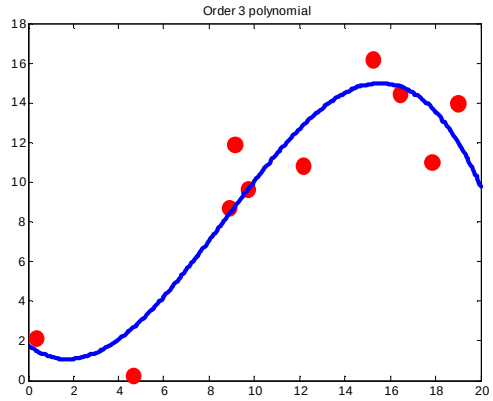
Some quick notation – our prediction yhat is a linear function, theta0 + theta1 times feature 1, plus theta2 times feature 2, etc.

For notational compactness, let's define a "feature 0" that is always value 1; then yhat is simply the vector product between a parameter vector theta and a feature vector x.

Nonlinear functions

- What if our hypotheses are not lines?
  - Ex: higher-order polynomials





Nonlinear functions

- Single feature x, predict target y:

$$D = \{(x^{(j)}, y^{(j)})\} \qquad \hat{y}(x) = \theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3$$

⇓ Add features:

$$D = \{([x^{(j)}, (x^{(j)})^2, (x^{(j)})^3], y^{(j)})\} \qquad \hat{y}(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3$$

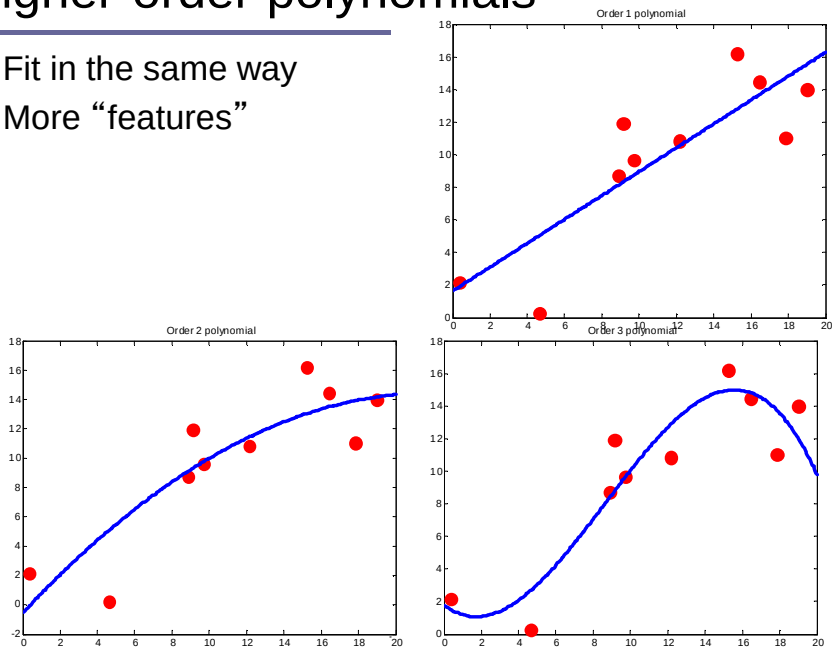
Linear regression in new features

- Sometimes useful to think of "feature transform"

$$\Phi(x) = [1\,,\, x\,,\, x^2\,,\, x^3\,,\, \ldots] \qquad \hat{y}(x) = \underline{\theta} \cdot \Phi(x)$$

Higher-order polynomials

- Fit in the same way
- More "features"



Features

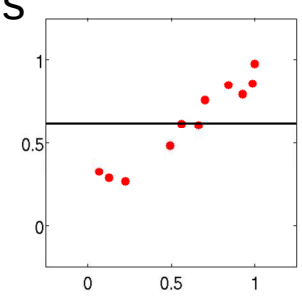
- In general, can use any features we think are useful
- Other information about the problem
  - Sq. footage, location, age, …
- Polynomial functions
  - Features [1, x, x^2, x^3, …]
- Other functions
  - 1/x, sqrt(x), $x_1 * x_2$, …
- "Linear regression" = linear in the parameters
  - Features we can make as complex as we want!

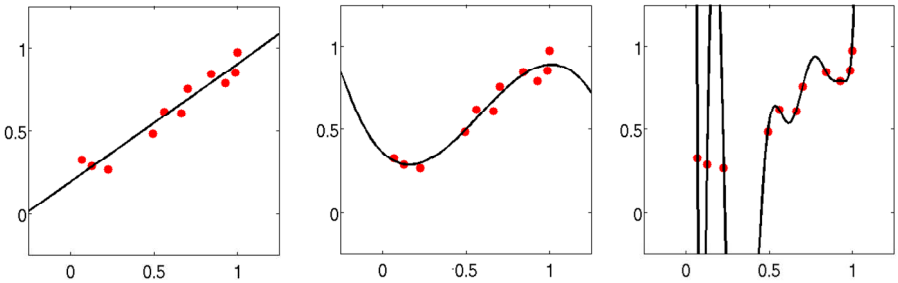
More generally, we can use any features that we might like.

For example, in addition to collecting more information about each data example, we can use whatever nonlinear transform Phi we think would be useful. Thus, when we say “linear regression”, we usually *mean* linear in the parameters theta, even if we may have used transformations to learn a function that is not actually linear in the input features x. Useful transforms, then, will give us *features* that are linearly predictive of the target y.

Higher-order polynomials

- Are more features better?
- “Nested” hypotheses
  - 2nd order more general than 1st,
  - 3rd order “ “ than 2nd, …
- Fits the observed data better

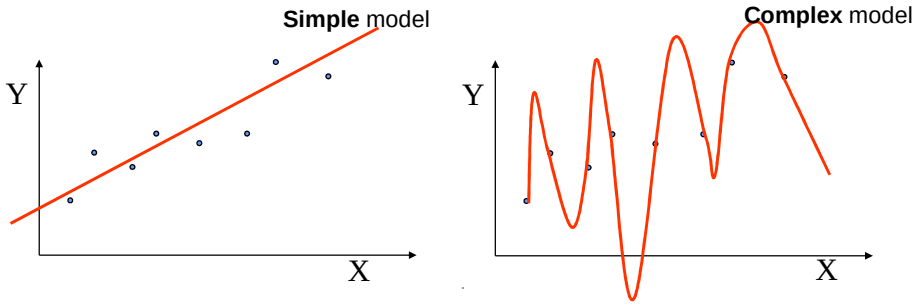




Once we are able to generate more features, such as polynomials, should we? Where should we stop? Notice that increasing the polynomial degree of our fit creates a nested sequence of models – for example, out of all lines, the constant predictor is one possibility – so the best line will always fit as well or better than the best constant. Similarly, all cubics include all lines, so a cubic fit will be as good or better than the best line. We get an improvement in fit quality with increasing features. At an extreme, given enough features, we will fit the data exactly (N eqs)

Overfitting and complexity

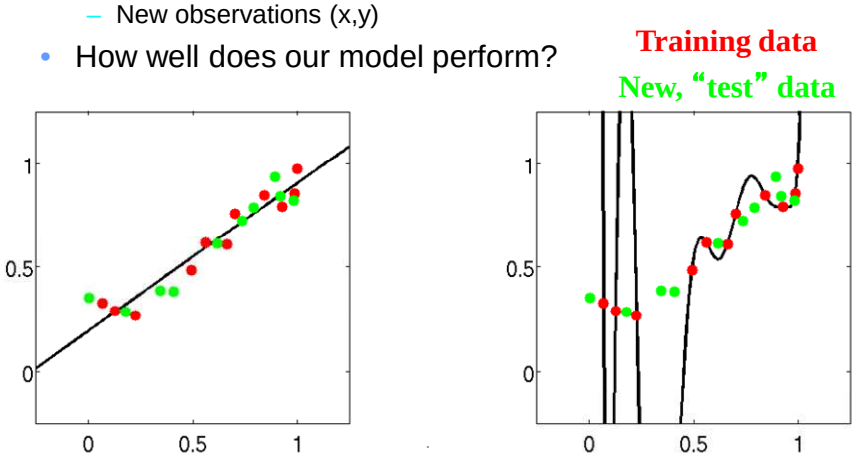
- More complex models will always fit the training data better
- But they may "overfit" the training data, learning complex relationships that are not really present



But, this is not such a good thing – remember our example, where we could explain the data with a line plus noise, or as a complex function that hits all the data. Although more complex models will always fit the training examples better, they may not preform as well in the future.

Test data

- After training the model
- Go out and get more data from the world
  - New observations (x,y)
- How well does our model perform?



Once we observe new data, we may discover that the complex model performs worse – over fitting! One way to assess this is through holding out validation data, or using cross-validation. We explicitly hide some data (the green points) from our learning algorithm, and then use these points to estimate its performance on future data.

Linear regression with multiple features

- Multiple variables = multiple features
- In regression with single variable(example 1)we had
  - X = house size, use this to predict
  - y = house price
- If in a new scheme we have more variables (such as number of bedrooms, number floors, age of the home)
  - x_1, x_2, x_3, x_4 are the four features
    - x_1 - size (feet squared)
    - x_2 - Number of bedrooms
    - x_3 - Number of floors
    - x_4 - Age of home (years)
  - y is the output variable (price)

- More notation
  - **n**
    - number of features (n = 4)
  - **m**
    - number of examples (i.e. number of rows in a table)
  - $\mathbf{x}^{\mathbf{i}}$
    - vector of the input for an example (so a vector of the four parameters for the i^{th} input example)
    - i is an index into the training set
    - So

        - x is an n-dimensional feature vector
        - x^3 is, for example, the 3rd house, and contains the four features associated with that house
    - $\mathbf{x_j^i}$
        - The value of feature j in the ith training example
        - So
            - x_2^3 is, for example, the number of bedrooms in the third house
- Now we have multiple features
    - What is the form of our hypothesis?
    - Previously our hypothesis took the form;
        - $h_\theta(x) = \theta_0 + \theta_1 x$
            - Here we have two parameters (theta 1 and theta 2) determined by our cost function
            - One variable x
    - Now we have multiple features
        - $h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 + \theta_4 x_4$
    - For example

        - $h_\theta(x) = 80 + 0.1x_1 + 0.01x_2 + 3x_3 - 2x_4$
            - An example of a hypothesis which is trying to predict the price of a house
            - Parameters are still determined through a cost function

    - For convenience of notation, $x_0 = 1$
        - For every example i you have an additional 0th feature for each example
        - So now your **feature vector** is n + 1 dimensional feature vector indexed from 0
            - This is a column vector called x
            - Each example has a column vector associated with it

        - So let's say we have a new example called "X"
    - **Parameters** are also in a 0 indexed n+1 dimensional vector
        - This is also a column vector called θ
        - This vector is the same for each example
- Considering this, hypothesis can be written
    - $h_\theta(x) = \theta_0 x_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 + \theta_4 x_4$
- If we do
    - $h_\theta(x) = \theta^T X$
        - θ^T is an [1 x n+1] matrix
        - In other words, because θ is a column vector, the transposition operation transforms it into a row vector
        - So before
            - θ was a matrix [n + 1 x 1]
        - Now
            - θ^T is a matrix [1 x n+1]
        - Which means the inner dimensions of θ^T and X match, so they can be multiplied together as
            - [1 x n+1] * [n+1 x 1]
            - = $h_\theta(x)$
            - So, in other words, the transpose of our parameter vector * an input example X gives you a predicted hypothesis which is [1 x 1] dimensions (i.e. a single value)
    - This $x_0 = 1$ lets us write this like this
- This is an example of multivariate linear regression

Gradient Descent algorithm and cost function

Linear regression - implementation (cost function)

- A cost function lets us figure out how to fit the best straight line to our data
- Choosing values for θ_i (parameters)
  - Different values give you different functions
  - If θ_0 is 1.5 and θ_1 is 0 then we get straight line parallel with X along 1.5 @ y
  - If θ_1 is > 0 then we get a positive slope
- Based on our training set we want to generate parameters which make the straight line
  - Chosen these parameters so $h_\theta(x)$ is close to y for our training examples
    - Basically, uses xs in training set with $h_\theta(x)$ to give output which is as close to the actual y value as possible
    - Think of $h_\theta(x)$ as a "y imitator" - it tries to convert the x into y, and considering we already have y we can evaluate how well $h_\theta(x)$ does this
- To formalize this;
  - We want to want to solve a **minimization problem**
  - Minimize $(h_\theta(x) - y)^2$
    - i.e. minimize the difference between h(x) and y for each/any/every example
  - Sum this over the training set

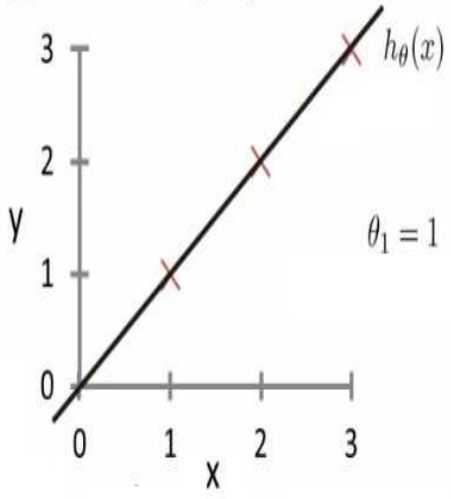
$$\frac{1}{2m}\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)^2$$

- Minimize squared different between predicted house price and actual house price
  - 1/2m
    - 1/m - means we determine the average
    - 1/2m the 2 makes the math a bit easier, and doesn't change the constants we determine at all (i.e. half the smallest value is still the smallest value!)
  - Minimizing θ_0/θ_1 means we get the values of θ_0 and θ_1 which find on average the minimal deviation of x from y when we use those parameters in our hypothesis function
- More cleanly, this is a cost function

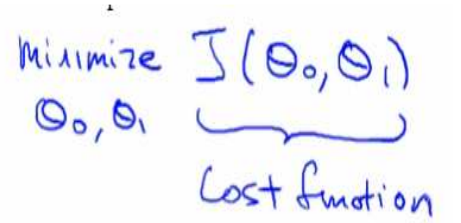
$$J(\theta_0,\theta_1) = \frac{1}{2m}\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)^2$$

- And we want to minimize this cost function

- And we want to minimize this cost function
  - Our cost function is (because of the summartion term) inherently looking at ALL the data in the training set at any time
- **So to recap**
  - **Hypothesis** - is like your prediction machine, throw in an *x* value, get a putative *y* value



  - **Cost** - is a way to, using your training data, determine values for your θ values which make the hypothesis as accurate as possible



- This cost function is also called the squared error cost function
  - This cost function is reasonable choice for most regression functions
  - Probably most commonly used function
- In case $J(\theta_0,\theta_1)$ is a bit abstract, going into what it does, why it works and how we use it in the coming sections

Cost function - a deeper look

- Lets consider some intuition about the cost function and why we want to use it
  - The cost function determines parameters
  - The value associated with the parameters determines how your hypothesis behaves, with different values generate different
- Simplified hypothesis
  - Assumes $\theta_0 = 0$

$$h_\theta(x) = \theta_1 x$$

$\theta_0 = 0$

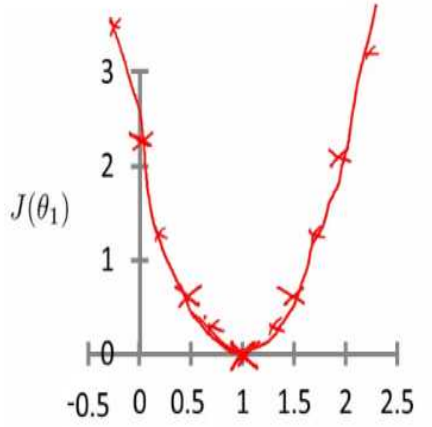
θ_1

$$J(\theta_1) = \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)}\right)^2$$

$$\underset{\theta_1}{\text{minimize}}\ J(\theta_1)$$

- Cost function and goal here are very similar to when we have θ_0, but with a simpler parameter
  - Simplified hypothesis makes visualizing cost function J() a bit easier

- So hypothesis pass through 0,0
- Two key functins we want to understand
  - $h_\theta(x)$
    - Hypothesis is a function of x - function of what the size of the house is
  - $J(\theta_1)$
    - Is a function of the parameter of θ_1
  - So for example
    - $\theta_1 = 1$
    - $J(\theta_1) = 0$
  - Plot
    - θ_1 vs $J(\theta_1)$
    - Data
      - 1)
        - $\theta_1 = 1$
        - $J(\theta_1) = 0$
      - 2)
        - $\theta_1 = 0.5$
        - $J(\theta_1) = \sim 0.58$
      - 3)
        - $\theta_1 = 0$
        - $J(\theta_1) = \sim 2.3$
  - If we compute a range of values plot
    - $J(\theta_1)$ vs θ_1 we get a polynomial (looks like a quadratic)



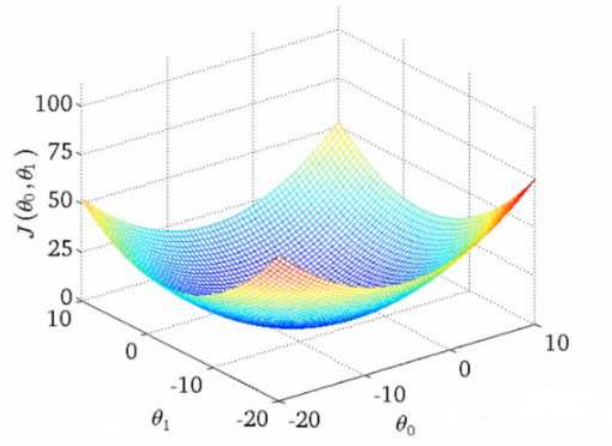
θ_1

- The optimization objective for the learning algorithm is find the value of θ_1 which minimizes $J(\theta_1)$
  - So, here $\theta_1 = 1$ is the best value for θ_1

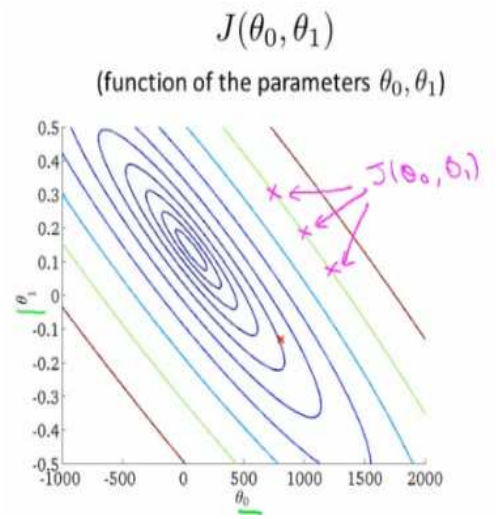
A deeper insight into the cost function - simplified cost function

- Assume you're familiar with contour plots or contour figures
  - Using same cost function, hypothesis and goal as previously
  - It's OK to skip parts of this section if you don't understand cotour plots
- Using our original complex hyothesis with two pariables,
  - So cost function is
    - $J(\theta_0, \theta_1)$
- Example,
  - Say
    - $\theta_0 = 50$
    - $\theta_1 = 0.06$
  - Previously we plotted our cost function by plotting
    - θ_1 vs $J(\theta_1)$
  - Now we have two parameters
    - Plot becomes a bit more complicated

- Generates a 3D surface plot where axis are
  - $X = \theta_1$
  - $Z = \theta_0$
  - $Y = J(\theta_0,\theta_1)$



- We can see that the height (y) indicates the value of the cost function, so find where y is at a minimum
- Instead of a surface plot we can use a **contour figures/plots**
  - Set of ellipses in different colors
  - Each colour is the same value of $J(\theta_0, \theta_1)$, but obviously plot to different locations because θ_1 and θ_0 will vary
  - Imagine a bowl shape function coming out of the screen so the middle is the concentric circles



- Each point (like the red one above) represents a pair of parameter values for Θ0 and Θ1
  - Our example here put the values at
    - θ_0 = ~800
    - θ_1 = ~-0.15
  - Not a good fit
    - i.e. these parameters give a value on our contour plot far from the center
  - If we have
    - θ_0 = ~360
    - θ_1 = 0
    - This gives a better hypothesis, but still not great - not in the center of the countour plot
  - Finally we find the minimum, which gives the best hypothesis
- Doing this by eye/hand is a pain in the ass
  - What we really want is an efficient algorithm fro finding the minimum for θ_0 and θ_1

Gradient descent algorithm

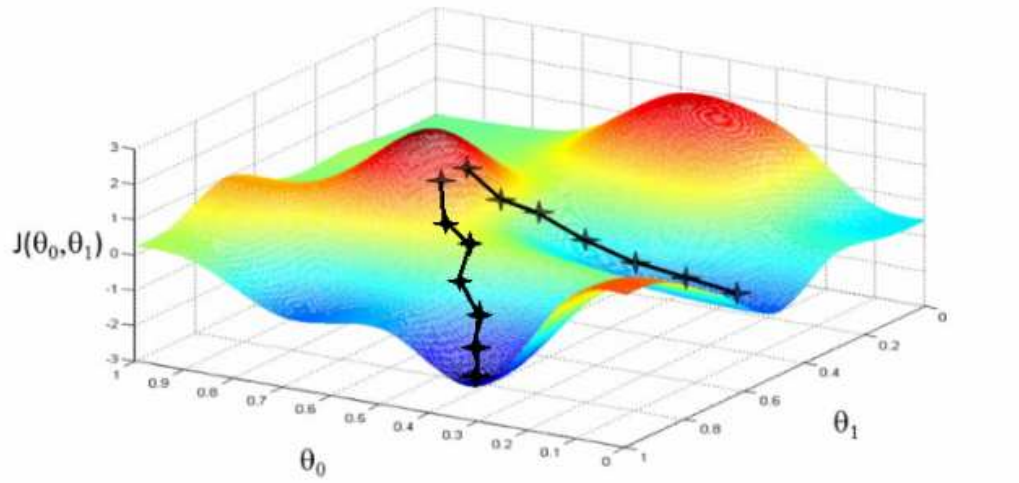
At a theoretical level, gradient descent is an algorithm that minimizes functions. Given a function defined by a set of parameters, gradient descent starts with an initial set of parameter values and iteratively moves toward a set of parameter values that minimize the function. This iterative minimization is achieved using calculus, taking steps in the negative direction of the function gradient.

Gradient descent algorithm

- Minimize cost function J
- Gradient descent
  - Used all over machine learning for minimization
- Start by looking at a general J() function
- Problem
  - We have $J(\theta_0, \theta_1)$
  - We want to get **min $J(\theta_0, \theta_1)$**
- Gradient descent applies to more general functions
  - $J(\theta_0, \theta_1, \theta_2 .... \theta_n)$
  - min $J(\theta_0, \theta_1, \theta_2 .... \theta_n)$

How does it work?

- Start with initial guesses
  - Start at 0,0 (or any other value)
  - Keeping changing θ_0 and θ_1 a little bit to try and reduce $J(\theta_0,\theta_1)$
- Each time you change the parameters, you select the gradient which reduces $J(\theta_0,\theta_1)$ the most possible
- Repeat
- Do so until you converge to a local minimum
- Has an interesting property
  - Where you start can determine which minimum you end up



  - Here we can see one initialization point led to one local minimum
  - The other led to a different one

A more formal definition

- Do the following until covergence

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1) \quad \text{(for } j = 0 \text{ and } j = 1\text{)}$$

- What does this all mean?
  - Update θ_j by setting it to (θ_j - α) times the partial derivative of the cost function with respect to θ_j
- Notation
  - :=
    - Denotes assignment
    - NB a = b is a *truth assertion*
  - α (alpha)
    - Is a number called the **learning rate**
    - Controls how big a step you take
      - If α is big have an aggressive gradient descent
      - If α is small take tiny steps

- Derivative term

$$\frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1)$$

    - Not going to talk about it now, derive it later
- There is a subtly about how this gradient descent algorithm is implemented
    - Do this for θ_0 and θ_1
    - For j = 0 and j = 1 means we **simultaneously** update both
    - How do we do this?
        - Compute the right hand side for both θ_0 and θ_1
            - So we need a temp value
        - Then, update θ_0 and θ_1 at the same time
        - We show this graphically below

$$\begin{aligned}
&\text{temp0} := \theta_0 - \alpha \tfrac{\partial}{\partial \theta_0} J(\theta_0, \theta_1) \\
&\text{temp1} := \theta_1 - \alpha \tfrac{\partial}{\partial \theta_1} J(\theta_0, \theta_1) \\
&\theta_0 := \text{temp0} \\
&\theta_1 := \text{temp1}
\end{aligned}$$

- If you implement the non-simultaneous update it's not gradient descent, and will behave weirdly
    - But it might look sort of right - so it's important to remember this!

Understanding the algorithm

- To understand gradient descent, we'll return to a simpler function where we minimize one parameter to help explain the algorithm in more detail
    - min θ_1 $J(\theta_1)$ where θ_1 is a real number
- Two key terms in the algorithm
    - Alpha
    - Derivative term
- Notation nuances
    - Partial derivative vs. derivative
        - Use partial derivative when we have multiple variables but only derive with respect to one
        - Use derivative when we are deriving with respect to all the variables
- Derivative term

$$\frac{\partial}{\partial \theta_j} J(\theta_1)$$

    - Derivative says
        - Lets take the tangent at the point and look at the slope of the line
        - So moving towards the mimum (down) will greate a negative derivative, alpha is always positive, so will update $j(\theta_1)$ to a smaller value
        - Similarly, if we're moving up a slope we make $j(\theta_1)$ a bigger numbers
- Alpha term (α)
    - What happens if alpha is too small or too large
    - Too small
        - Take baby steps
        - Takes too long
    - Too large
        - Can overshoot the minimum and fail to converge
- When you get to a local minimum

- Gradient of tangent/derivative is 0
- So derivative term = 0
- alpha * 0 = 0
- So $\theta_1 = \theta_1 - 0$
- So θ_1 remains the same

- As you approach the global minimum the derivative term gets smaller, so your update gets smaller, even with alpha is fixed
  - Means as the algorithm runs you take smaller steps as you approach the minimum
  - So no need to change alpha over time

Linear regression with gradient descent

- Apply gradient descent to minimize the squared error cost function $J(\theta_0, \theta_1)$
- Now we have a partial derivative

$$\frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1) = \frac{2}{2\theta_j} \cdot \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right)^2$$

$$= \frac{2}{2\theta_j} \frac{1}{2m} \sum_{i=1}^{m} \left(\theta_0 + \theta_1 x^{(i)} - y^{(i)} \right)^2$$

- So here we're just expanding out the first expression
  - $J(\theta_0, \theta_1) = 1/2m....$
  - $h_\theta(x) = \theta_0 + \theta_1 * x$
- So we need to determine the derivative for each parameter - i.e.
  - When j = 0
  - When j = 1
- Figure out what this partial derivative is for the θ_0 and θ_1 case
  - When we derive this expression in terms of j = 0 and j = 1 we get the following

$$j = 0 : \frac{\partial}{\partial \theta_0} J(\theta_0, \theta_1) = \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right)$$

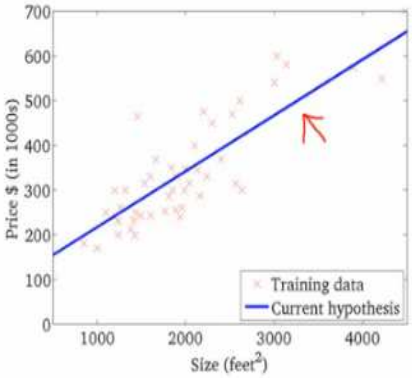
$$j = 1 : \frac{\partial}{\partial \theta_1} J(\theta_0, \theta_1) = \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right) \cdot x^{(i)}$$

- To check this you need to know multivariate calculus
  - So we can plug these values back into the gradient descent algorithm
- How does it work
  - Risk of meeting different local optimum
  - The linear regression cost function is always a **convex function** - always has a single minimum
    - Bowl shaped
    - One global optima

- So gradient descent will always converge to global optima
- In action
  - Initialize values to
    - $\theta_0 = 900$
    - $\theta_1 = -0.1$

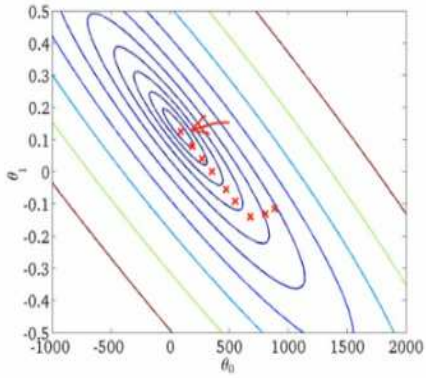
$h_\theta(x)$

(for fixed θ_0, θ_1, this is a function of x)



$J(\theta_0, \theta_1)$

(function of the parameters θ_0, θ_1)



- End up at a global minimum
- This is actually **Batch Gradient Descent**
  - Refers to the fact that over each step you look at all the training data
    - Each step compute over m training examples
  - Sometimes non-batch versions exist, which look at small data subsets
    - We'll look at other forms of gradient descent (to use when m is too large) later in the course
- There exists a numerical solution for finding a solution for a minimum function
  - **Normal equations** method
  - Gradient descent scales better to large data sets though
  - Used in lots of contexts and machine learning

What's next - important extensions
Two extension to the algorithm

- **1) Normal equation for numeric solution**
  - To solve the minimization problem we can solve it [min $J(\theta_0, \theta_1)$] exactly using a numeric method which avoids the iterative approach used by gradient descent
  - Normal equations method
  - Has advantages and disadvantages
    - Advantage
      - No longer an alpha term
      - Can be much faster for some problems
    - Disadvantage
      - Much more complicated
  - We discuss the normal equation in the **linear regression with multiple features** section
- **2) We can learn with a larger number of features**
  - So may have other parameters which contribute towards a prize
    - e.g. with houses
      - Size
      - Age
      - Number bedrooms
      - Number floors
    - x1, x2, x3, x4
  - With multiple features becomes hard to plot
    - Can't really plot in more than 3 dimensions
    - Notation becomes more complicated too
      - Best way to get around with this is the notation of linear algebra
      - Gives notation and set of things you can do with matrices and vectors
      - e.g. Matrix

$$X = \begin{bmatrix} 2104 & 5 & 1 & 45 \\ 1416 & 3 & 2 & 40 \\ 1534 & 3 & 2 & 30 \\ 852 & 2 & 1 & 36 \end{bmatrix} \qquad y = \begin{bmatrix} 460 \\ 232 \\ 315 \\ 172 \end{bmatrix}$$

- We see here this matrix shows us
  - Size
  - Number of bedrooms
  - Number floors
  - Age of home
- All in one variable
  - Block of numbers, take all data organized into one big block
- Vector
  - Shown as y
  - Shows us the prices
- Need linear algebra for more complex linear regression modles
- Linear algebra is good for making computationally efficient models (as seen later too)
  - Provide a good way to work with large sets of data sets
  - Typically vectorization of a problem is a common optimization technique

Simple Linear Regression Stochastic Gradient Descent (How to solve)

The coefficients used in simple linear regression can be found using stochastic gradient descent.

Linear regression is a linear system and the coefficients can be calculated analytically using linear algebra. Stochastic gradient descent is not used to calculate the coefficients for linear regression in practice (in most cases).

Linear regression does provide a useful exercise for learning stochastic gradient descent which is an important algorithm used for minimizing cost functions by machine learning algorithms.

As stated above, our linear regression model is defined as follows:

$$y = \theta_0 + \theta_1 * x$$

Gradient Descent Iteration #1

Let's start with values of 0.0 for both coefficients.

$$\theta_0 = 0.0$$

$$\theta_1 = 0.0$$

$$y = 0.0 + 0.0 * x$$

We can calculate the error for a prediction as follows:

$$error = p(i) - y(i)$$

Where p(i) is the prediction for the i'th instance in our dataset and y(i) is the i'th output variable for the instance in the dataset.

We can now calculate he predicted value for y using our starting point coefficients for the first training instance:

$$x=1, y=1$$

$$p(i) = 0.0 + 0.0 * 1$$

$$p(i) = 0$$

Using the predicted output, we can calculate our error:

$$error = 0 – 1$$

$$error = -1$$

We can now use this error in our equation for gradient descent to update the weights. We will start with updating the intercept first, because it is easier.

We can say that θ_0 is accountable for all of the error. This is to say that updating the weight will use just the error as the gradient. We can calculate the update for the B0 coefficient as follows:

$$\theta_0 (t+1) = \theta_0 (t) – alpha * error$$

Where θ_0 (t+1) is the updated version of the coefficient we will use on the next training instance, θ_0 (t) is the current value for θ_0 alpha is our learning rate and error is the error we calculate for the training instance. Let's use a small learning rate of 0.01 and plug the values into the equation to work out what the new and slightly optimized value of θ_0 will be:

$$\theta_0 (t+1) = 0.0 – 0.01 * -1.0$$

$$\theta_0 (t+1) = 0.01$$

Now, let's look at updating the value for θ_1. We use the same equation with one small change. The error is filtered by the input that caused it. We can update B1 using the equation:

$$\theta_1 (t+1) = \theta_1 (t) – alpha * error * x$$

Where θ_1 (t+1) is the update coefficient, θ_1 (t) is the current version of the coefficient, alpha is the same learning rate described above, error is the same error calculated above and x is the input value.

We can plug in our numbers into the equation and calculate the updated value for θ_1:

$$\theta_1 (t+1) = 0.0 – 0.01 * -1 * 1$$

$$\theta_1 (t+1) = 0.01$$

We have just finished the first iteration of gradient descent and we have updated our weights to be θ_0 =0.01 and θ_1 =0.01. This process must be repeated for the remaining 4 instances from our dataset.

One pass through the training dataset is called an epoch.

Gradient Descent Iterations

This is an iterative process. There is a list of all of the values for the coefficients over the 20 iterations that we get:

```
0.01    0.01
0.0397  0.0694
0.066527    0.176708
0.08056049  0.21880847
0.1188144616    0.410078328
0.1235255337    0.4147894001
0.1439944904    0.4557273134
0.1543254529    0.4970511637
0.1578706635    0.5076867953
0.1809076171    0.6228715633
0.1828698253    0.6248337715
0.1985444516    0.6561830242
0.2003116861    0.6632519622
0.1984110104    0.657549935
0.2135494035    0.7332419008
0.2140814905    0.7337739877
0.2272651958    0.7601413984
0.2245868879    0.7494281668
0.219858174 0.7352420252
0.230897491 0.7904386102
```

The values should match closely, but may have minor differences due to different precisions. You can plug each pair of coefficients back into the simple linear regression equation. This is useful because we can calculate a prediction for each training instance and in turn calculate the error.

Below is a plot of the error for each set of coefficients as the learning process unfolded. This is a useful graph as it shows us that error was decreasing with each iteration and starting to bounce around a bit towards the end. The final coefficients have the values θ_0 =0.230897491 and θ_0

=0.7904386102 referring error plot.

Logistic regression and classification

Logistic Regression Model

The logistic regression model takes real-valued inputs and makes a prediction as to the probability of the input belonging to the default class (class 0).

If the probability is > 0.5 we can take the output as a prediction for the default class (class 0), otherwise the prediction is for the other class (class 1).

For this dataset, the logistic regression has three coefficients just like linear regression, for example:

$$output = \theta_0 + \theta_1 * x1 + \theta_2 * x2$$

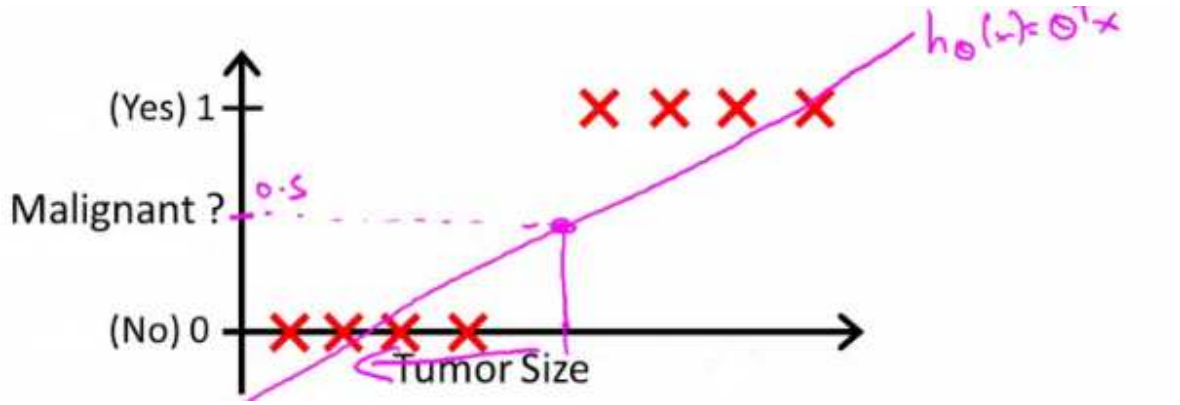
The job of the learning algorithm will be to discover the best values for the coefficients (θ_0 , θ_1 and θ_2) based on the training data.

Unlike linear regression, the output is transformed into a probability using the logistic function:

$$p(class=0) = 1 / (1 + e^{(-output)})$$

Classification

- Where y is a discrete value
  - Develop the logistic regression algorithm to determine what class a new input should fall into
- Classification problems
  - Email -> spam/not spam?
  - Online transactions -> fraudulent?
  - Tumor -> Malignant/benign
- Variable in these problems is Y
  - Y is either 0 or 1
    - 0 = negative class (absence of something)
    - 1 = positive class (presence of something)
- Start with **binary class problems**
  - Later look at multiclass classification problem, although this is just an extension of binary classification
- How do we develop a classification algorithm?
  -
  - Tumour size vs malignancy (0 or 1)
  - We *could* use linear regression
    -
      - Then threshold the classifier output (i.e. anything over some value is yes, else no)
      - In our example below linear regression with thresholding seems to work



- We can see above this does a reasonable job of stratifying the data points into one of two classes
  - But what if we had a single Yes with a very small tumour
  - This would lead to classifying all the existing yeses as nos
- Another issues with linear regression
  - We know Y is 0 or 1
  - Hypothesis can give values large than 1 or less than 0
- So, logistic regression generates a value where is always either 0 or 1
  -
  - Logistic regression is a **classification algorithm** - don't be confused

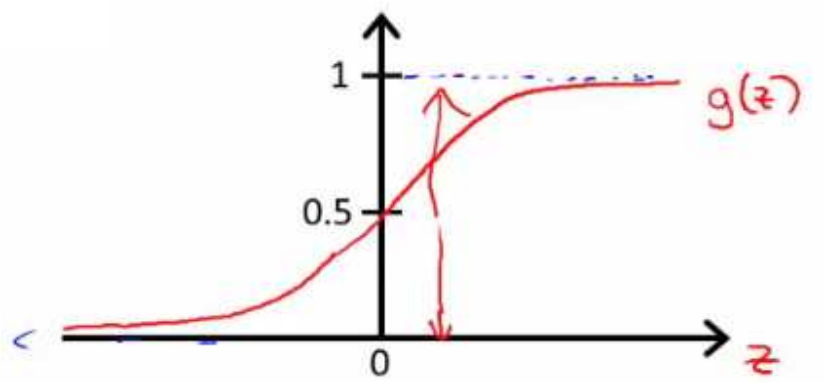
Hypothesis representation

- What function is used to represent our hypothesis in classification
- We want our classifier to output values between 0 and 1
  -
  - When using linear regression we did $h_\theta(x) = (\theta^T x)$
  - For classification hypothesis representation we do $h_\theta(x) = g((\theta^T x))$

- Where we define g(z)
  - z is a real number
- $g(z) = 1/(1 + e^{-z})$
  - This is the **sigmoid function**, or the **logistic function**
- If we combine these equations we can write out the hypothesis as

$$h_\theta(x) = \frac{1}{1 + e^{-\theta^T x}}$$

- What does the sigmoid function look like
- Crosses 0.5 at the origin, then flattens out]
  - Asymptotes at 0 and 1



- Given this we need to fit θ to our data

Interpreting hypothesis output

- When our hypothesis ($h_\theta(x)$) outputs a number, we treat that value as the estimated probability that y=1 on input x
  - Example
    - If X is a feature vector with $x_0 = 1$ (as always) and x_1 = tumourSize
    - $h_\theta(x) = 0.7$
      - Tells a patient they have a 70% chance of a tumor being malignant

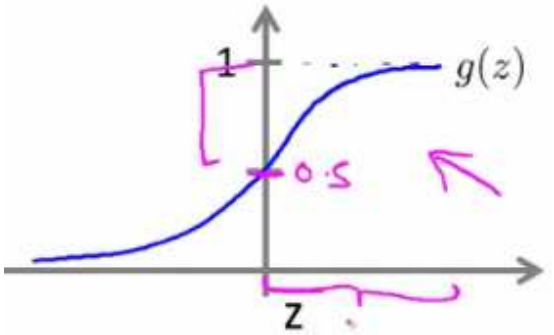
- We can write this using the following notation
  - $h_\theta(x) = P(y=1|x\ ;\ \theta)$
- What does this mean?
  - Probability that y=1, given x, parameterized by θ

- Since this is a binary classification task we know y = 0 or 1
  - So the following must be true
    - $P(y=1|x\ ;\ \theta) + P(y=0|x\ ;\ \theta) = 1$
    - $P(y=0|x\ ;\ \theta) = 1 - P(y=1|x\ ;\ \theta)$

Decision boundary

- Gives a better sense of what the hypothesis function is computing
- Better understand of what the hypothesis function looks like
  - One way of using the sigmoid function is;
    - When the probability of y being 1 is greater than 0.5 then we can predict y = 1
    - Else we predict y = 0
  - When is it exactly that $h_\theta(x)$ is greater than 0.5?
    - Look at sigmoid function

- g(z) is greater than or equal to 0.5 when z is greater than or equal to 0

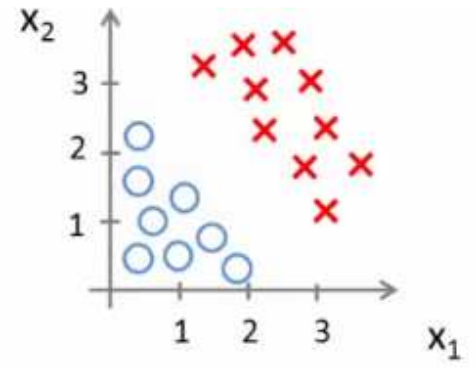


  - So if z is positive, g(z) is greater than 0.5
    - $z = (\theta^T x)$
  - So when

        - $\theta^T x >= 0$
    - Then $h_\theta >= 0.5$
- So what we've shown is that the hypothesis predicts y = 1 when $\theta^T x >= 0$
    - The corollary of that when $\theta^T x <= 0$ then the hypothesis predicts y = 0
    - Let's use this to better understand how the hypothesis makes its predictions

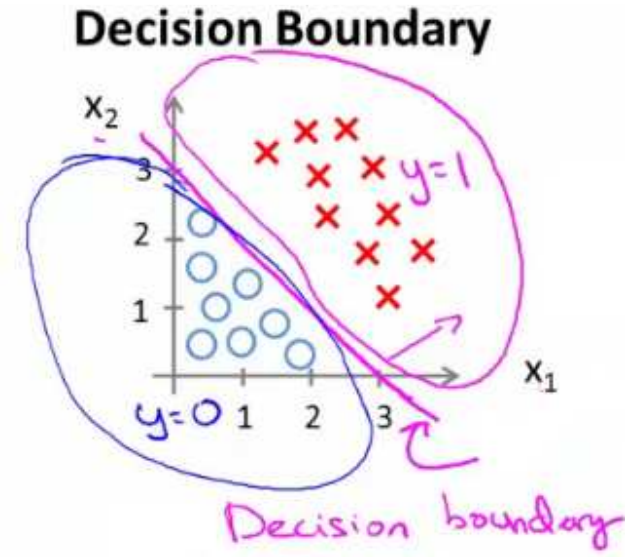
Decision boundary

- $h_\theta(x) = g(\theta_0 + \theta_1 x_1 + \theta_2 x_2)$



- So, for example
    - $\theta_0 = -3$
    - $\theta_1 = 1$
    - $\theta_2 = 1$
- So our parameter vector is a column vector with the above values
    - So, θ^T is a row vector = [-3,1,1]
- What does this mean?
    -
    - The z here becomes $\theta^T x$
    - We predict "y = 1" if
        - $-3x_0 + 1x_1 + 1x_2 >= 0$
        - $-3 + x_1 + x_2 >= 0$

- We can also re-write this as
  - If ($x_1 + x_2 >= 3$) then we predict y = 1
  - If we plot
    - $x_1 + x_2 = 3$ we graphically plot our **decision boundary**

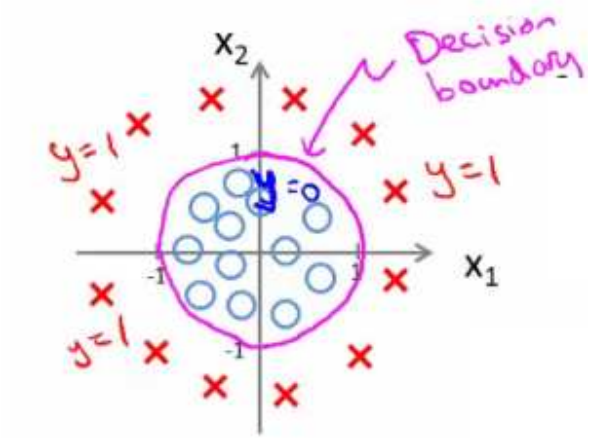


- Means we have these two regions on the graph
  -
  - Blue = false
  - Magenta = true
  - Line = decision boundary
    -
      - Concretely, the straight line is the set of points where $h_\theta(x) = 0.5$ exactly
  - The decision boundary is a property of the hypothesis
    - Means we can create the boundary with the hypothesis and parameters without any data
      - Later, we use the data to determine the parameter values

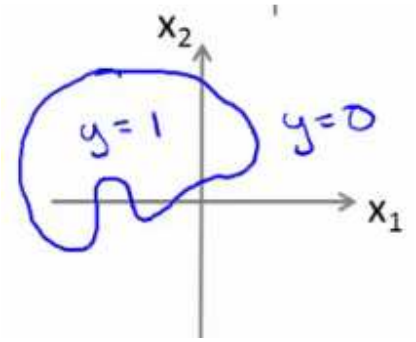
- i.e. y = 1 if
  - $5 - x_1 > 0$
  - $5 > x_1$

Non-linear decision boundaries

- Get logistic regression to fit a complex non-linear data set
  -
  - Like polynomial regress add higher order terms
  - So say we have
    -
      - $h_\theta(x) = g(\theta_0 + \theta_1 x_1 + \theta_3 x_1^2 + \theta_4 x_2^2)$
      - We take the transpose of the θ vector times the input vector
        -
        - Say θ^T was [-1,0,0,1,1] then we say;
        - Predict that "y = 1" *if*
          -
          - $-1 + x_1^2 + x_2^2 >= 0$
            or
          - $x_1^2 + x_2^2 >= 1$
        - If we plot $x_1^2 + x_2^2 = 1$
          - This gives us a circle with a radius of 1 around 0



- Mean we can build more complex decision boundaries by fitting complex parameters to this (relatively) simple hypothesis
- More complex decision boundaries?
  -
    - By using higher order polynomial terms, we can get even more complex decision boundaries



Cost function for logistic regression

- Fit θ parameters
- Define the optimization object for the cost function we use the fit the parameters
  -
    - Training set of *m* training examples
      -
        - Each example has is n+1 length column vector

Training set: $\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \cdots, (x^{(m)}, y^{(m)})\}$

m examples $x \in \begin{bmatrix} x_0 \\ x_1 \\ \cdots \\ x_n \end{bmatrix}$ $x_0 = 1, y \in \{0, 1\}$

$$h_\theta(x) = \frac{1}{1 + e^{-\theta^T x}}$$

- This is the situation
  - Set of m training examples
  - Each example is a feature vector which is n+1 dimensional
  - $x_0 = 1$
  - $y \in \{0,1\}$
  - Hypothesis is based on parameters (θ)
    - Given the training set how to we chose/fit θ?
- Linear regression uses the following function to determine θ

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

- Instead of writing the squared error term, we can write
  -
  - If we define "cost()" as;
    - $\text{cost}(h_\theta(x^i), y) = 1/2(h_\theta(x^i) - y^i)^2$
    - Which evaluates to the cost for an individual example using the same measure as used in linear regression
  - We can **redefine J(θ) as**

$$J(\theta) = \frac{1}{m} \sum_{i=1}^{m} \text{Cost}(h_\theta(x^{(i)}), y^{(i)})$$

    - Which, appropriately, is the sum of all the individual costs over the training data (i.e. the same as linear regression)
- To further simplify it we can get rid of the superscripts
  - So

$$J(\theta) = \frac{1}{m} \sum_{i=1}^{m} \text{Cost}(h_\theta(x), y)$$

- What does this actually mean?
  - This is the cost you want the learning algorithm to pay if the outcome is $h_\theta(x)$ and the actual outcome is y
  - If we use this function for logistic regression this is a **non-convex function** for parameter optimization
    - Could work....
- What do we mean by non convex?
  -
  - We have some function - $J(\theta)$ - for determining the parameters
  - Our hypothesis function has a non-linearity (sigmoid function of $h_\theta(x)$)
    - This is a complicated non-linear function
  - If you take $h_\theta(x)$ and plug it into the Cost() function, and them plug the Cost() function into $J(\theta)$ and plot $J(\theta)$ we find many local optimum -> *non convex function*
  - Why is this a problem
    - Lots of local minima mean gradient descent may not find the global optimum - may get stuck in a global minimum
  - We would like a convex function so if you run gradient descent you converge to a global minimum

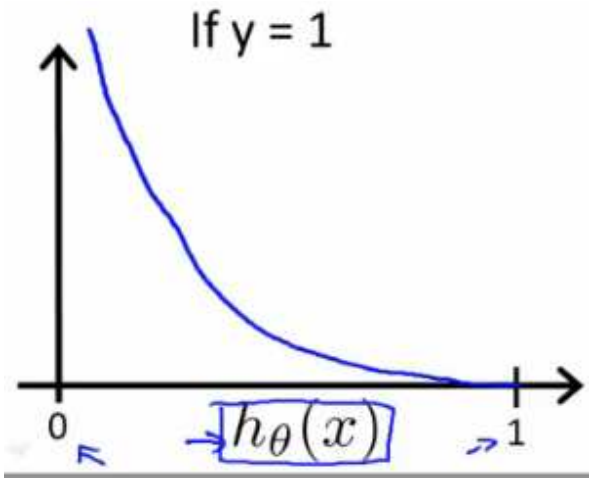
A convex logistic regression cost function

- To get around this we need a different, convex Cost() function which means we can apply gradient descent

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

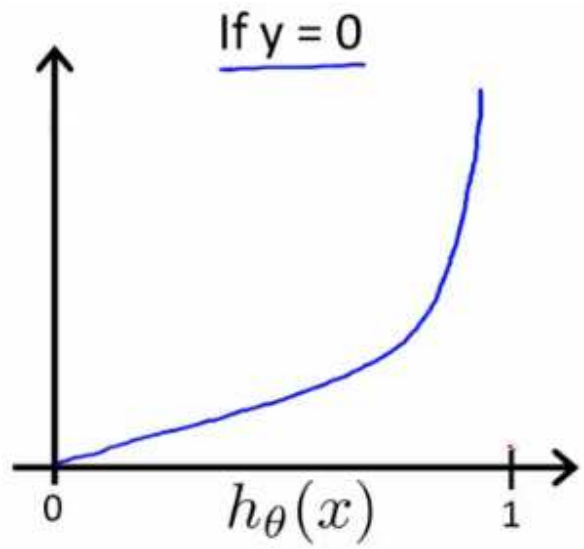
- **This is our logistic regression cost function**
  -

    - This is the penalty the algorithm pays
    - Plot the function
- Plot y = 1
    -
    - So $h_\theta(x)$ evaluates as -log($h_\theta(x)$)



- So when we're right, cost function is 0
    - Else it slowly increases cost function as we become "more" wrong
    - X axis is what we predict
    - Y axis is the cost associated with that prediction
- This cost functions has some interesting properties
    - If y = 1 and $h_\theta(x) = 1$
        - If hypothesis predicts exactly 1 and thats exactly correct then that corresponds to 0 (exactly, not nearly 0)
    - As $h_\theta(x)$ goes to 0
        - Cost goes to infinity
        - This captures the intuition that if $h_\theta(x) = 0$ (predict $P(y=1|x;\theta) = 0$) but y = 1 this will penalize the learning algorithm with a massive cost
- What about if y = 0
- then cost is evaluated as -log(1- $h_\theta(x)$)
    -

- Just get inverse of the other function



- Now it goes to plus infinity as $h_\theta(x)$ goes to 1
- With our particular cost functions J(θ) is going to be convex and avoid local minimum

Simplified cost function and gradient descent

- Define a simpler way to write the cost function and apply gradient descent to the logistic regression
  - By the end should be able to implement a fully functional logistic regression function
- Logistic regression cost function is as follows

$$J(\theta) = \frac{1}{m} \sum_{i=1}^{m} \text{Cost}(h_\theta(x^{(i)}), y^{(i)})$$

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

Note: $y = 0$ or 1 always

- This is the cost for a single example
  -
  - For binary classification problems y is always 0 or 1
    -
      - Because of this, we can have a simpler way to write the cost function
        -
        - Rather than writing cost function on two lines/two cases
        - Can compress them into one equation - more efficient
  - Can write cost function is
    - **cost(h_θ, (x),y) = -ylog(h_θ(x)) - (1-y)log(1- h_θ(x))**
      - This equation is a more compact of the two cases above
  - We know that there are only two possible cases
    -
      - y = 1
        - Then our equation simplifies to
          - -log(h_θ(x)) - (0)log(1 - h_θ(x))
            - -log(h_θ(x))
            - Which is what we had before when y = 1
      - y = 0
        - Then our equation simplifies to
          - -(0)log(h_θ(x)) - (1)log(1 - h_θ(x))

        - = -log(1- $h_\theta(x)$)
        - Which is what we had before when y = 0
    - Clever!
- So, in summary, our cost function for the θ parameters can be defined as

$$J(\theta) = -\frac{1}{m}[\sum_{i=1}^{m} y^{(i)} \log h_\theta(x^{(i)}) + (1 - y^{(i)}) \log (1 - h_\theta(x^{(i)}))$$

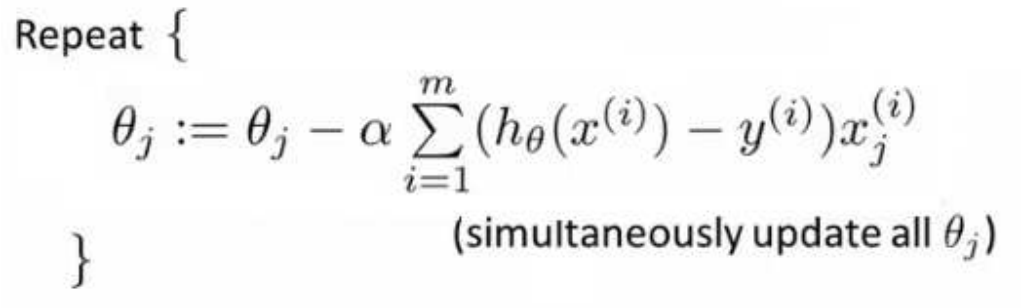
- Why do we chose this function when other cost functions exist?
    - This cost function can be derived from statistics using the principle of **maximum likelihood estimation**
        - Note this does mean there's an underlying Gaussian assumption relating to the distribution of features
    - Also has the nice property that it's convex
- To fit parameters θ:
    - Find parameters θ which minimize J(θ)
    - This means we have a set of parameters to use in our model for future predictions
- Then, if we're given some new example with set of features x, we can take the θ which we generated, and output our prediction using

$$h_\theta(x) = \frac{1}{1+e^{-\theta^T x}}$$

    -
    - This result is
        -
        - p(y=1 | x ; θ)
            -
            - Probability y = 1, given x, parameterized by θ

How to minimize the logistic regression cost function

- Now we need to figure out how to minimize J(θ)
  -
  - Use gradient descent as before
  - Repeatedly update each parameter using a learning rate



- If you had *n* features, you would have an n+1 column vector for θ
- This equation is the same as the linear regression rule
  - The only difference is that our definition for the hypothesis has changed
- Previously, we spoke about how to monitor gradient descent to check it's working
  - Can do the same thing here for logistic regression
- When implementing logistic regression with gradient descent, we have to update all the θ values (θ_0 to θ_n) simultaneously
  - Could use a for loop
  - Better would be a vectorized implementation
- Feature scaling for gradient descent for logistic regression also applies here

Advanced optimization

- Previously we looked at gradient descent for minimizing the cost function
- Here look at advanced concepts for minimizing the cost function for logistic regression
  - Good for large machine learning problems (e.g. huge feature set)
- *What is gradient descent actually doing?*
  -
  - We have some cost function J(θ), and we want to minimize it
  - We need to write code which can take θ as input and compute the following
    -
    - J(θ)
    - Partial derivative if J(θ) with respect to j (where j=0 to j = n)

$$J(\theta)$$
$$\frac{\partial}{\partial \theta_j} J(\theta) \quad \text{(for } j = 0, 1, \ldots, n \text{)}$$

- Given code that can do these two things
  - Gradient descent repeatedly does the following update

$$\text{Repeat } \{ \theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta) \}$$

- So update each j in θ sequentially
- So, we must;
  - Supply code to compute J(θ) and the derivatives
  - Then plug these values into gradient descent
- Alternatively, instead of gradient descent to minimize the cost function we could use
  - **Conjugate gradient**

    - **BFGS** (Broyden-Fletcher-Goldfarb-Shanno)
    - **L-BFGS** (Limited memory - BFGS)
- These are more optimized algorithms which take that same input and minimize the cost function
- These are *very* complicated algorithms
- Some properties
    -
    - **Advantages**
        - No need to manually pick alpha (learning rate)
            - Have a clever inner loop (line search algorithm) which tries a bunch of alpha values and picks a good one
        - Often faster than gradient descent
            - Do more than just pick a good learning rate
        - Can be used successfully without understanding their complexity
    - **Disadvantages**
        -
        - Could make debugging more difficult
        - Should not be implemented themselves
        - Different libraries may use different implementations - may hit performance

Using advanced cost minimization algorithms

- How to use algorithms
    -
    - Say we have the following example

Example:

$$\theta = \begin{bmatrix} \theta_1 \\ \theta_2 \end{bmatrix}$$

$$J(\theta) = (\theta_1 - 5)^2 + (\theta_2 - 5)^2$$

$$\frac{\partial}{\partial \theta_1} J(\theta) = 2(\theta_1 - 5)$$

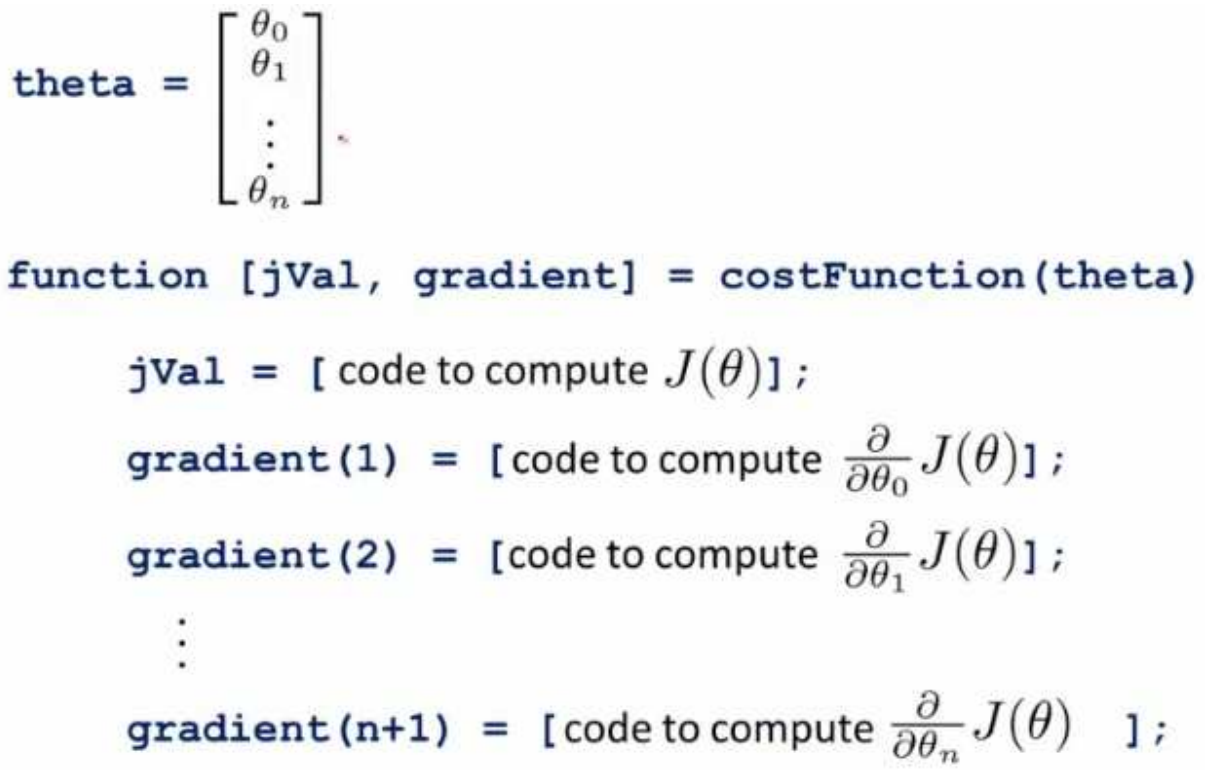
$$\frac{\partial}{\partial \theta_2} J(\theta) = 2(\theta_2 - 5)$$

- Example above
  - θ_1 and θ_2 (two parameters)
  - Cost function here is $J(\theta) = (\theta_1 - 5)^2 + (\theta_2 - 5)^2$
  - The derivatives of the $J(\theta)$ with respect to either θ_1 and θ_2 turns out to be the $2(\theta_i - 5)$
- First we need to define our cost function, which should have the following signature

function [jval, gradient] = costFunction(THETA)

- Input for the cost function is **THETA**, which is a vector of the θ parameters
- Two return values from **costFunction** are
  - **jval**
    - How we compute the cost function θ (the underived cost function)
      - In this case = $(\theta_1 - 5)^2 + (\theta_2 - 5)^2$
  - **gradient**
    - 2 by 1 vector
    - 2 elements are the two partial derivative terms
    - i.e. this is an n-dimensional vector
      - Each indexed value gives the partial derivatives for the partial derivative of $J(\theta)$ with respect to θ_i
      - Where i is the index position in the **gradient** vector

- With the cost function implemented, we can call the advanced algorithm using
- How do we apply this to logistic regression?
  -
  - Here we have a vector

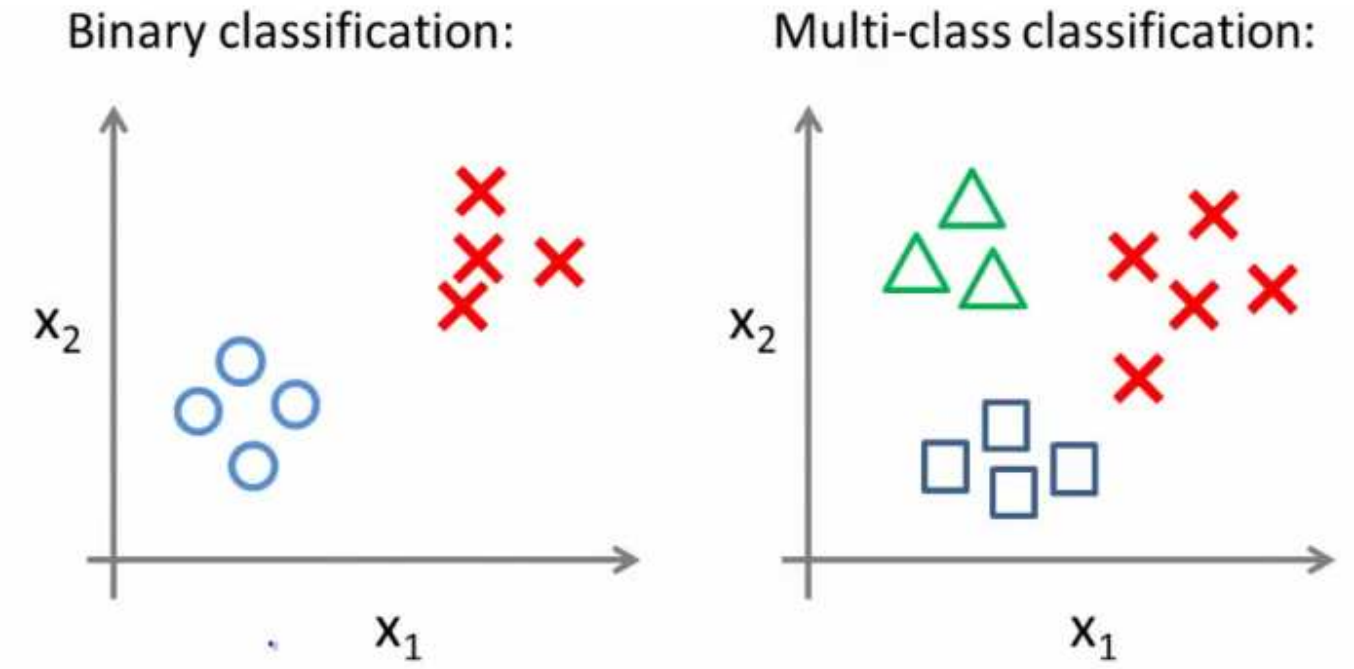


- Here
  - theta is a n+1 dimensional column vector
  - Octave indexes from 1, not 0
- Write a cost function which captures the cost function for logistic regression

Multiclass classification problems

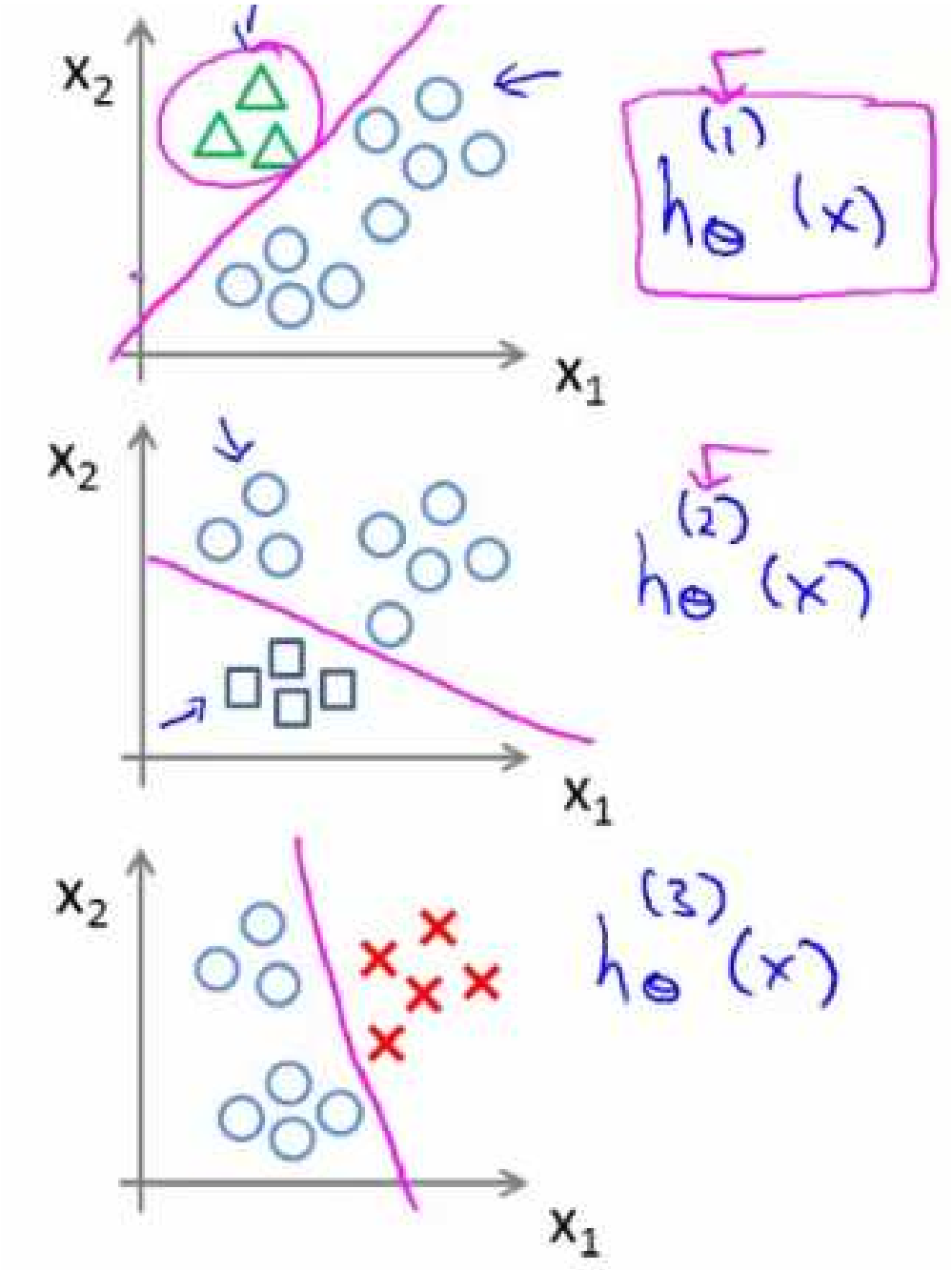
- Getting logistic regression for multiclass classification using **one vs. all**
- Multiclass - more than yes or no (1 or 0)

-
    - Classification with multiple classes for assignment



- Given a dataset with three classes, how do we get a learning algorithm to work?
    -
    - Use one vs. all classification make binary classification work for multiclass classification
- **One vs. all classification**
    -
    - Split the training set into three separate binary classification problems
        -
        - i.e. create a new fake training set
            -
            - Triangle (1) vs crosses and squares (0) $h_\theta^1(x)$
                - $P(y=1 \mid x_1; \theta)$
            - Crosses (1) vs triangle and square (0) $h_\theta^2(x)$

    - $P(y=1 \mid x_2; \theta)$
- Square (1) vs crosses and square (0) $h_\theta^3(x)$
    - $P(y=1 \mid x_3; \theta)$



- **Overall**
  - Train a logistic regression classifier $h_\theta^{(i)}(x)$ for each class i to predict the probability that y = i
  - On a new input, *x* to make a prediction, pick the class *i* that maximizes the probability that $h_\theta^{(i)}(x) = 1$

Regularization

In Machine learning and statistics, a common task is to fit a model to a set of training data. This model can be used later to make predictions or classify new data points. When the model fits the training data but does not have a good predicting performance and generalization power, we have an overfitting problem. Regularization is a technique used to avoid this overfitting problem. The idea behind regularization is that models that overfit the data are complex models that have for example too many parameters. In the example below we see how three different models fit the same dataset.

- Regularization refers to the method of preventing overfitting, by explicitly controlling the model complexity.
- It leads to smoothening of the regression line and thus prevents overfitting. It does so by penalizing the bent of the regression line that tries to closely match the noisy data points.
- There are couple of techniques to achieve regularization viz. L1 and L2, L2 regularization leads to small weights and L1 regularization leads to many zero weights (sparsity)

Simple example to explain regularization

Regularization artificially discourages complex or extreme explanations of the world even if they fit the has been observed better. The idea is that such explanations are unlikely to generalize well to the future; they may happen to explain a few data points from the past well, but this may just be because of accidents of the sample.

For example let's say you're training your guard robot to identify which neighborhood pets to scare off . You give the robot the following examples:

- Bo, snake, small, **FRIENDLY**
- Miles, dog, small, **FRIENDLY**
- Fifi, cat, small, *ENEMY*
- Muffy, cat, small, **FRIENDLY**
- Rufus, dog, large, **FRIENDLY**
- Jebediah, snail, small, **FRIENDLY**
- Aloysius, dog, large, *ENEMY*
- Tom, cat, large, *ENEMY*
  Your robot might happily learn the rule

  pets with up to four-letter names are enemies, as are large dogs with names beginning with 'A', except that small snakes are not enemies

  since it perfectly fits this data, but it feels clunky. Intuitively we'd suppose there's a simpler rule that fits as well. The reason we feel complexity is bad per se is that it seems unlikely to generalize well to new pets: are small snakes really not enemies? or is it just that they weren't in this input?

  In contrast, the rule

  large dogs and cats are enemies

  does not fit the data perfectly, but fits reasonably well. It's simpler, and maybe we therefore suppose it will be more correct for the thousands of other pets not in this sample.

  Loosely, regularization is the kind of thing that discourages complexity, even if it means picking a less-accurate rule according to the training data, in the math that might evaluate these rules.

One more example of a business scenario

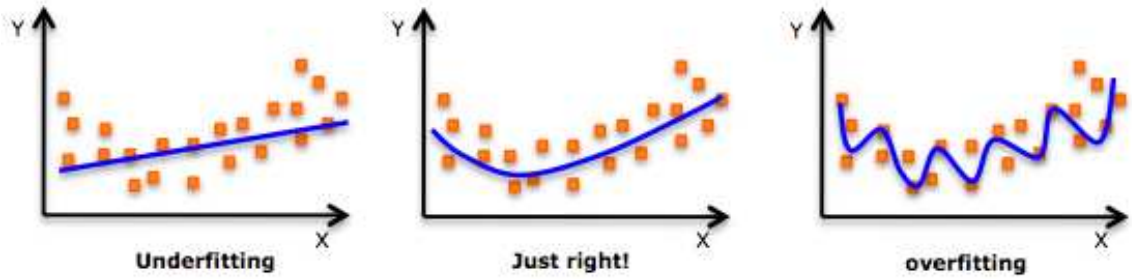
In the world of analytics, where we try to fit a curve to every pattern, Over-fitting is one of the biggest concerns. However, in general models are equipped enough to avoid over-fitting, but in general there is a manual intervention required to make sure the model does not consume more than enough attributes.

Let’s consider an example here, we have 10 students in a classroom. We intend to train a model based on their past score to predict their future score. There are 5 females and 5 males in the class. The average score of females is 60 whereas that of males is 80. The overall average of the class is 70.

Now, there are several ways to make the prediction:

- Predict the score as 70 for the entire class.
- Predict score of males = 80 and females = 60. This simplistic model which might give a better estimate than the first one.
- Now let’s try to overkill the problem. We can use the roll number of students to make a prediction and say that every student will exactly score same marks as last time. Now, this is unlikely to be true and we have reached such granular level that we can go seriously wrong.

The first case here is called under fit, the second being an optimum fit and last being an over-fit.



Methods to avoid Over-fitting:

Following are the commonly used methodologies :

1. **Cross-Validation** : Cross Validation in its simplest form is a one round validation, where we leave one sample as in-time validation and rest for training the model. But for keeping lower variance a higher fold cross validation is preferred.

2. **Early Stopping** : Early stopping rules provide guidance as to how many iterations can be run before the learner begins to over-fit.
3. **Pruning** : Pruning is used extensively while building CART models. It simply removes the nodes which add little predictive power for the problem in hand.
4. **Regularization** : This is the technique we are going to discuss in more details. Simply put, it introduces a cost term for bringing in more features with the objective function. Hence, it tries to push the coefficients for many variables to zero and hence reduce cost term.