ANDREW NG COURSE

- **Introduction**

Machine learning gives computers the ability to learn without being explicity programmed. (Arthur Samuel)

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

Supervised Learning: Divides into regression and classification.

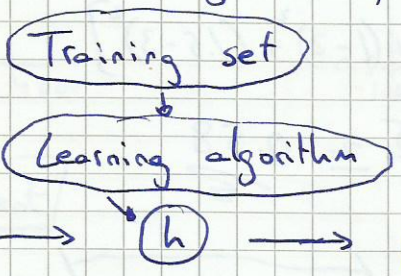
Regression ⇒ continuous output

Classification ⇒ discrete output

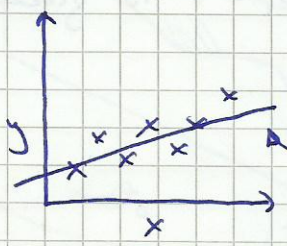
Unsupervised Learning: Allows to approach problems with little or no idea what results should look like

- **Model Representation**

$x^{(i)}$ = input variables $y^{(i)}$ = output variables

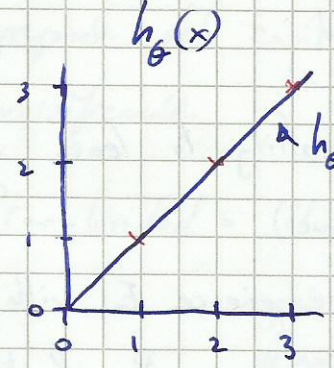


Cost function. Squarred error function or Mean squared error.

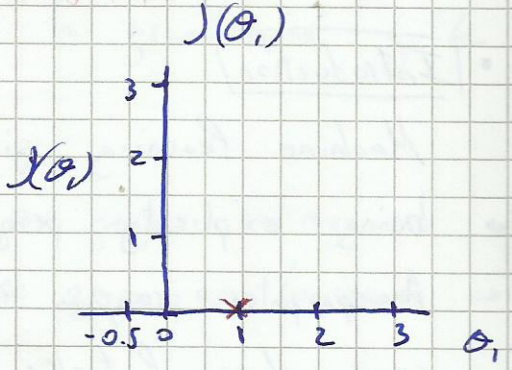


$$\underset{\theta_0,\theta_1}{\text{minimize}} \quad \frac{1}{2m}\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)^2$$

Idea: Choose θ_0, θ_1 so that $h_\theta(x)$ is close to y for our training examples (x, y)



$\theta_1 = 1$

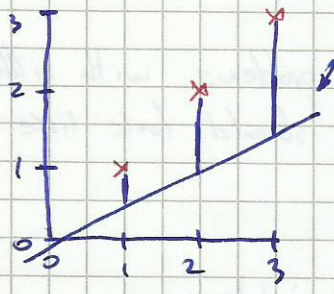


$$J(\theta_1) = \frac{1}{2m} \sum_{i=1}^{m} \left(\theta_1 x^{(i)} - y^{(i)}\right)^2 =$$

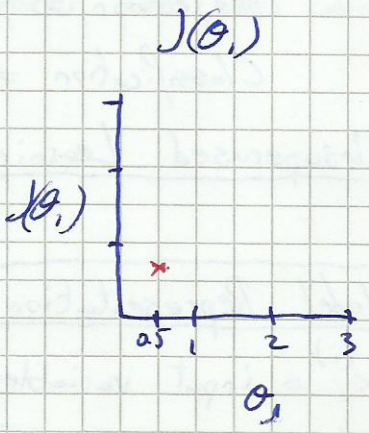
$$= \frac{1}{2m}\left(0^2 + 0^2 + 0^2\right) = 0^2$$

$J(1) = 0$

$\theta_1 = 0.5$



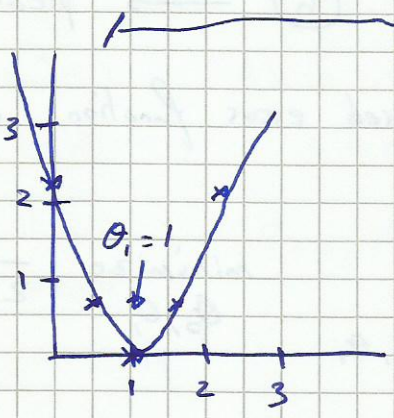
$\theta_1 = 0.5$



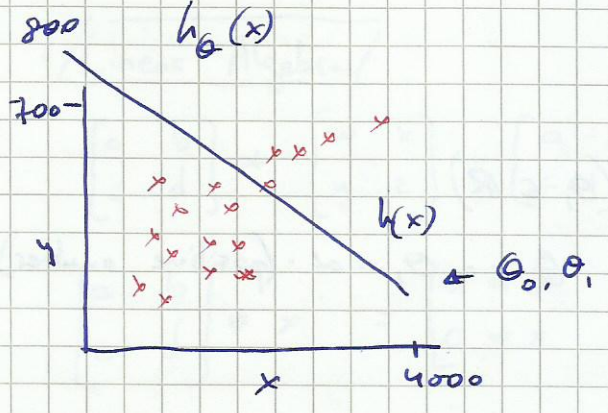
$$J(0.5) = \frac{1}{2m}\left[(0.5-1)^2 + (1-2)^2 + (1.5-3)^2\right]$$

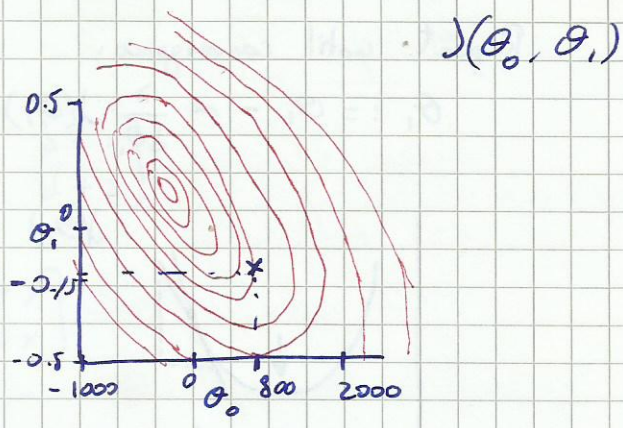
$$\frac{1}{2 \cdot 3}(3.5) = \frac{3.5}{6} \approx 0.58$$

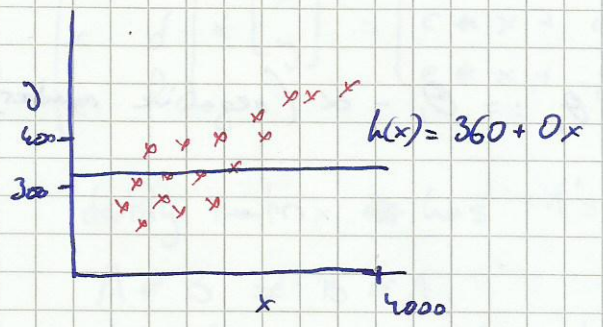
$J(0.5) = 0.58$

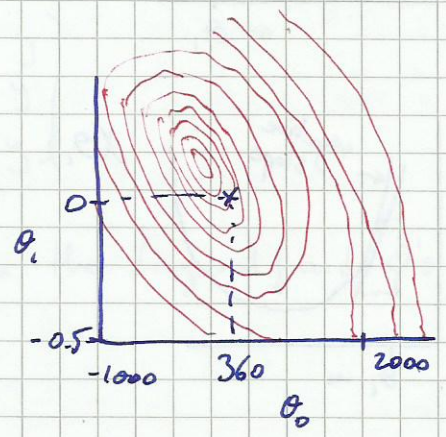


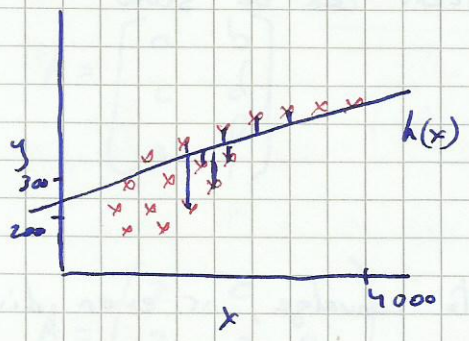
$\theta_1 = 1$ is global minimum

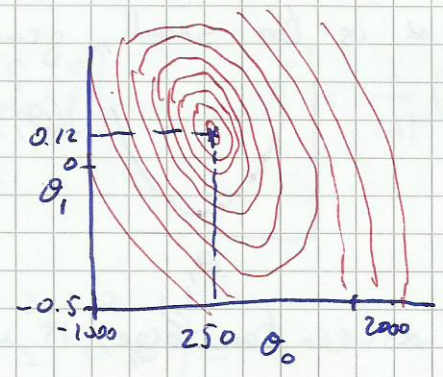








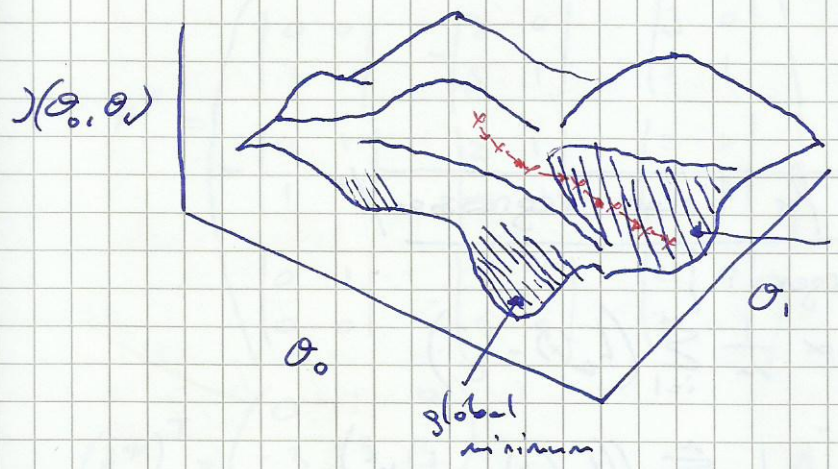




Gradient Descent

repeat until convergence:

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



$$temp0 := \theta_0 - \alpha \frac{\partial}{\partial \theta_0} J(\theta_0, \theta_1)$$

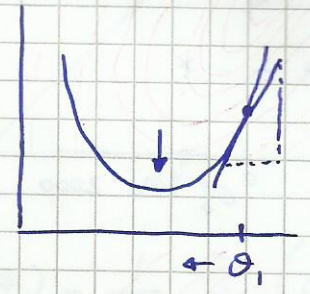
$$temp1 := \theta_1 - \alpha \frac{\partial}{\partial \theta_1} J(\theta_0, \theta_1)$$

$$\theta_0 := temp0$$

$$\theta_1 := temp1$$

Repeat until convergence:

$$\theta_1 := \theta_1 - \alpha \frac{\partial}{\partial \theta_1} J(\theta_1)$$

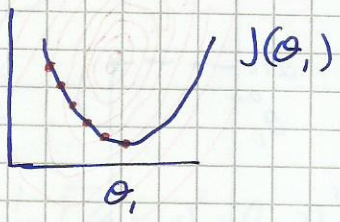


$(\theta_1 \in \mathbb{R})$

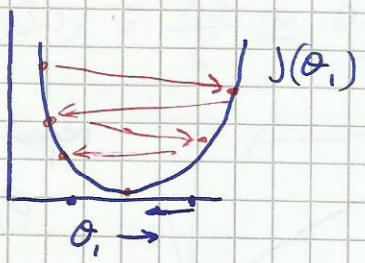
$\theta_1 := \theta_1 - \alpha \cdot$ (positive number)

$\theta_1 := \theta_1 - \alpha$ (negative number)

If α is too small, gradient descent can be slow



If α is too large, it may fail to converge, or even diverge



Gradient Descent for linear regression

repeat until convergence: {

$$\theta_0 := \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^i) - y^i \right)$$

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

• **Linear Algebra**

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

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} * x = \begin{bmatrix} a*x & b*x \\ c*x & d*x \end{bmatrix}$$

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

Identity matrix has 1's on the diagonal and 0's everywhere

$$A*B \neq B*A$$

$$(A*B)*C = A*(B*C)$$

$$A = \begin{bmatrix} a & b \\ c & d \\ e & f \end{bmatrix} \qquad A^T = \begin{bmatrix} a & c & e \\ b & d & f \end{bmatrix} \qquad A^{-1} = \frac{1}{|A|}(A^*)^t$$

$$A = \begin{pmatrix} 2 & 0 & 1 \\ 3 & 0 & 0 \\ 5 & 1 & 1 \end{pmatrix} \qquad A = \begin{vmatrix} 2 & 0 & 1 \\ 3 & 0 & 0 \\ 5 & 1 & 1 \end{vmatrix} = 3+0+0-(0+0+0) = 3$$

$$A^* = \begin{pmatrix} \begin{vmatrix} 0 & 0 \\ 1 & 1 \end{vmatrix} & -\begin{vmatrix} 3 & 0 \\ 5 & 1 \end{vmatrix} & \begin{vmatrix} 3 & 0 \\ 5 & 1 \end{vmatrix} \\ -\begin{vmatrix} 0 & 1 \\ 1 & 1 \end{vmatrix} & \begin{vmatrix} 2 & 1 \\ 5 & 1 \end{vmatrix} & -\begin{vmatrix} 2 & 0 \\ 5 & 1 \end{vmatrix} \\ \begin{vmatrix} 0 & 1 \\ 0 & 0 \end{vmatrix} & -\begin{vmatrix} 2 & 1 \\ 3 & 0 \end{vmatrix} & \begin{vmatrix} 2 & 0 \\ 3 & 0 \end{vmatrix} \end{pmatrix} = \begin{pmatrix} 0 & -3 & 3 \\ 1 & -3 & -2 \\ 0 & 3 & 0 \end{pmatrix}$$

$$(A^*)^T = \begin{pmatrix} 0 & 1 & 0 \\ -3 & -3 & 3 \\ 3 & -2 & 0 \end{pmatrix}$$

$$A^{-1} = \frac{1}{3}\begin{pmatrix} 0 & 1 & 0 \\ -3 & -3 & 3 \\ 3 & -2 & 0 \end{pmatrix} = \begin{pmatrix} 0 & \frac{1}{3} & 0 \\ -1 & -1 & 1 \\ 1 & -\frac{2}{3} & 0 \end{pmatrix}$$

• Multiple Features

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

$x_j^{(i)}$ = value of feature j in the i^{th} training example.

$x^{(i)}$ = the input (features) of the i^{th} training example

m = the number of training examples

n = the number of features

The multivariable form of the hypothesis function:

$h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_n x_n$ is equivalent to

$$h_\theta(x) = \begin{bmatrix} \theta_0 & \dots & \theta_n \end{bmatrix} \begin{bmatrix} x_0 \\ \vdots \\ x_n \end{bmatrix} = \theta^T x$$

• Gradient Descent For Multiple Variables

repeat until convergence: {

$$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right) \cdot x_j^{(i)} \quad \text{for } j := 0 \dots n$$

}

The difference between multiple variables and 1 variable is that:

- With 1 variable: $(n = 1)$

  θ_0 has x_0 / θ_1 has x_1

- With multiple variables: $(n \geq 1)$

  $x_0^{(i)} = 1$

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 range. And so will oscillate inefficiently down to the optimum when the variables are very uneven.

To prevent this is to modify ranges of our input variables

$-1 \leq x_{(i)} \leq 1$ or $-0.5 \leq x_{(i)} \leq 0.5$

2 techiques to help are

- Feature scaling: $x_{(i)} / \max(x_{(i)}) - \min(x_{(i)})$
- Mean normalization: $(x_{(i)} - \text{average}(i)) / \underbrace{\max(x_{(i)}) - \min(x_{(i)})}_{\text{Can be replaced by standard desviation}}$

- Debugging Gradient Descent. Make a plot with number of iterations on the x-axis. $J(\theta)$ over the number of iterations of gradient descent. If $J(\theta)$ ever increases, you need to decrease α.
Declare convergence if $J(\theta)$ decreases by less than 10^{-3} in one iteration.
  - For sufficiently small α, $J(\theta)$ should decrease on every iteratio
  - If α is too small, gradient descent can be slow to converge.
- Polynomial regression
We can combine multiple features into one. Our hypothesis function needn't be linear if that doesn't fit the data well.
To make it a square root function, we could do:
$$h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 \sqrt{x_1}$$

- **Normal Equation**
We will minimize J by explicitly taking its derivates with respect to the θ_j's, and setting them to zero. This allows us to find the optimum theta without iteration.
$$\theta = (X^T X)^{-1} X^T y$$

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

Gradient Descent	Normal equation
Need to choose α	No need to choose α
Need many iterations	No need to iterate
$O(kn^2)$	$O(n^3)$, need calculate $(X^T X)^{-1}$
With n is large	With n is short

If X^TX is non invertible, the common causes might be having:

- Redundant features: 2 features are very closely related (linearly dependent)
- Too many features ($m \leq n$): Delete some features or use regularization

- **Classification**

We want to predict take on only a small number of discrete values.

For binary classification problems, $y \in \{0, 1\}$

+ Hypothesis representation

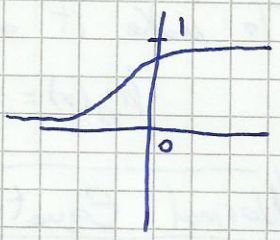
It doesn't make sense for $h_\theta(x)$ to take values larger than 1 or smaller than 0 when we know that $y \in \{0, 1\}$.

To fix it, $0 \leq h_\theta(x) \leq 1$

Our new form uses the "Sigmoid Function" or also called the "Logistic function"

$$h_\theta(x) = g(\theta^T x) \qquad z = \theta^T x$$

$$g(z) = \frac{1}{1 + e^{-z}}$$



$h_\theta(x)$ give us the probability that our output is 1.

$$h_\theta(x) = P(y = 1 / x; \theta) = 1 - P(y = 0 / x; \theta)$$

+ Decision Boundary

$h_\theta(x) \geq 0.5 \rightarrow y = 1$

$h_\theta(x) < 0.5 \rightarrow y = 0$

// $g(z) \geq 0.5$ when $z \geq 0$

* Reminder

$z = 0, \; e^0 = 1 \Rightarrow g(z) = 0.5$

$z \to \infty, \; e^{-\infty} \to 0 \Rightarrow g(z) = 1$

$z \to -\infty, \; e^{\infty} \to \infty \Rightarrow g(z) = 0$

If $g = \theta^T x$, then: $h_\theta(x) = g(\theta^T x) \geq 0.5$ when $\theta^T x \geq 0$

From these statements, we can say:

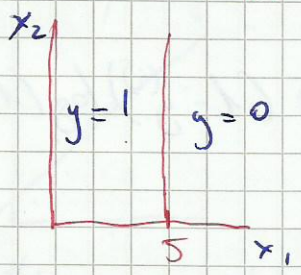
$\theta^T x \geq 0 \Rightarrow y = 1$ && $\theta^T x < 0 \Rightarrow y = 0$

Example: $\theta = \begin{bmatrix} 5 \\ -1 \\ 0 \end{bmatrix}$ $y = 1$ if $5 + (-1)x_1 + 0x_2 \geq 0$

$5 - x_1 \geq 0$

$-x_1 \geq -5$

$x_1 \leq 5$



the input to the sigmoid function $g(z)$ doesn't need to be linear, and could be a function that describes a circle

- **Cost Function**

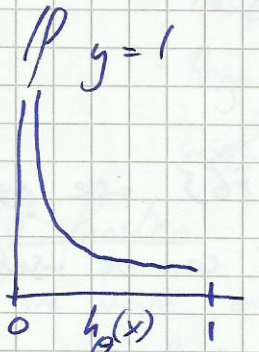
We can't use the same cost function that we use for linear regression because the Logistic Function will cause the output to be wavy. In other words, it will not be a convex function. To fix this: ↓

Our cost function looks like:

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

$$Cost(h_\theta(x), y) = -\log(h_\theta(x)) \quad \text{if } y = 1$$

$$Cost(h_\theta(x), y) = -\log(1 - h_\theta(x)) \quad \text{if } y = 0$$



$Cost(h_\theta(x), y) = 0$ if $h_\theta(x) = y$

$Cost(h_\theta(x), y) \to \infty$ if $h_\theta(x) \to 1$ && $y = 0$

$Cost(h_\theta(x), y) \to \infty$ if $h_\theta(x) \to 0$ && $y = 1$

- **Simplified Cost Function & Gradient Descent**

We can compress our cost function's 2 conditional cases into one case:

$$\text{Cost}(h_\theta(x), y) = \underbrace{-y \log(h_\theta(x))}_{y=0} \underbrace{- (1-y) \log(1-h_\theta(x))}_{y=1}$$

$$\Updownarrow$$

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

A vectorized implementation is:

$$h = g(X\theta)$$

$$J(\theta) = \frac{1}{m} \left(-y^T \log(h) - (1-y)^T \log(1-h) \right)$$

+ **Gradient Descent**

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

Repeat {

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

}

this algorith is identical to used in linear regression

We still have to update all values in theta.

A vectorized implementation is:

$$\theta := \theta - \frac{\alpha}{m} X^T \left(g(X\theta) - \vec{y} \right)$$

+ **Advanced Optimization**

Conjugate gradient, BFGS, L-BFGS are more complex, faster ways to optimize θ that can be used instead of gradient descent.

- **Multiclass Classification: One vs all**

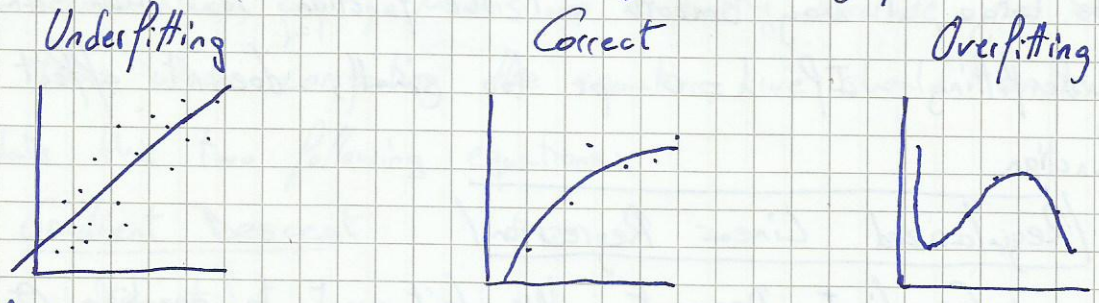
Now $y \in \{1 \dots n\}$. We divide our problem in n parts.

Train a logistic regression classifier $h_\theta(x)$ for each class to predic

the probability that $y = i$. To make a prediction on a new x, pick the class that maximize $h_\theta(x)$

- /Solving the Problem of Overfitting/

It might seem that the more features we add, better. However, there is also a danger in adding too many features.



Underfitting, or high bias. It's usually caused by a function that is too simple or uses too few features.

Overfitting, or high variance. It's caused by doesn't generalize well to predict new data. It's usually caused by a complicated function that creates a lot of unnecessary curves and angles unrelated to the data.

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
2) Regularization
   - Keep all the features, but reduce the magnitude of parameters
   - Regularization works well when we have a lot of slightly useful features.

- Cost function

If we have overfitting from our hypothesis function. We can modify our cost function

$$\min_\theta \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)}\right)^2 + 1000\,\theta_3^2 + 1000\,\theta_4^2$$

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

$$\min_\theta \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right)^2 + \lambda \sum_{j=1}^{n} \theta_j^2$$

The λ, or lambda, is the regularization parameter. If λ is too large, it may smooth out the function too much and cause underfitting. If $\lambda = 0$ or too small, doesn't affect to the function

- **Regularized Linear Regression**
  - + Gradient Descent. We don't want to penalize θ_0

Repeat {

$$\theta_0 := \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right) x_0^{(i)}$$

$$\theta_j := \theta_j - \alpha \left[\frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right) x_j^{(i)} + \frac{\lambda}{m} \theta_j \right]$$

$$j \in \{1, 2 \dots n\}$$

}

The term $\frac{\lambda}{m}\theta_j$ performs our regularization. With some manipulation our update rule can also be represented as:

$$\theta_j := \theta_j \left(1 - \alpha \frac{\lambda}{m}\right) - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right) x_j^{(i)}$$

$1 - \alpha \frac{\lambda}{m}$ will always be less than 1. You can see it as reducing the value of θ_j, by some amount on every update.

  - + Normal equation

$$\theta = \left(X^T X + \lambda L \right)^{-1} X^T y \quad \text{where} \quad L = \begin{bmatrix} 0 & & & \\ & 1 & & \\ & & 1 & \\ & & & 1 \end{bmatrix}_{(n+1)\times(n+1)}$$

Recall that if $m \leq n$, then $X^T X$ is non-invertible. However, when we add the term λL, then $X^T X + \lambda L$ becomes invertible.

Regularized Logistic Regression

We can avoid overfitting, we can regularize the cost function by adding a term to the end:

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)} \log\left(h_\theta(x^{(i)})\right) + (1-y^{(i)}) \log\left(1-h_\theta(x^{(i)})\right)\right] + \frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2$$

the second sum, $\sum_{j=1}^{n}\theta_j^2$ means to explicitly exclude the bias term, θ_0. Thus, when computing the equation, we should continuously update the two following equations:

Gradient Descent

Repeat {

$$\theta_0 := \theta_0 - \alpha\frac{1}{m}\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)x_0^{(i)}$$

$$\theta_j := \theta_j - \alpha\left[\frac{1}{m}\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)x_j^{(i)} + \frac{\lambda}{m}\theta_j\right]$$

$$(j = 1 \dots n)$$

}

Neural Networks

Visually, a simplistic representation looks like:

$$\begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} \rightarrow [\ ] \rightarrow h_\theta(x)$$

input layer → hidden layer → output layer

$a_i^{(j)}$ = "activation" of unit i in layer j

$\Theta^{(j)}$ = matrix of weights controlling function mapping from layer j to layer $j+1$

If we have one hidden layer:

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

$$a_1^{(2)} = g\left(\Theta_{10}^{(1)}x_0 + \Theta_{11}^{(1)}x_1 + \Theta_{12}^{(1)}x_2 + \Theta_{13}^{(1)}x_3\right)$$

$$a_2^{(2)} = g\left(\Theta_{20}^{(1)}x_0 + \Theta_{21}^{(1)}x_1 + \Theta_{22}^{(1)}x_2 + \Theta_{23}^{(1)}x_3\right)$$

$$a_3^{(2)} = g\left(\Theta_{30}^{(1)}x_0 + \Theta_{31}^{(1)}x_1 + \Theta_{32}^{(1)}x_2 + \Theta_{33}^{(1)}x_3\right)$$

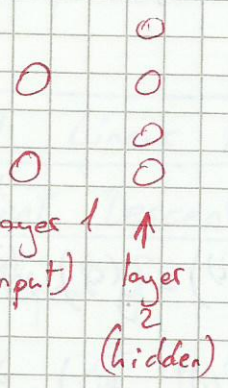
$$h_\Theta(x) = a_1^{(3)} = g\left(\Theta_{10}^{(2)}a_0^{(2)} + \Theta_{11}^{(2)}a_1^{(2)} + \Theta_{12}^{(2)}a_2^{(2)} + \Theta_{13}^{(2)}a_3^{(2)}\right)$$

this is saying that we compute our activation nodes by using a 3x4 matrix of parameters. Each layer gets its own matrix of weights, $\Theta^{(j)}$

If network has s_j units in layer j and s_{j+1} units in layer $j+1$, then $\Theta^{(j)}$ will be of dimension $s_{j+1} \times (s_j + 1)$

The +1 comes the "bias nodes"

Example:



$\Theta^{(1)}$ where $s_j = 2$, $s_{j+1} = 4$

4×3

$s_{j+1} \times (s_j + 1)$ $\quad$ j = layer

We'll do a vectorized implementation of the above functions.

We're going to define a new variable $z_k^{(j)}$ that encompasses the parameters inside our g function.

$$a_1^{(2)} = g(z_1^{(2)}) \; ; \quad a_2^{(2)} = g(z_2^{(2)}) \; ; \quad a_3^{(2)} = g(z_3^{(2)})$$

For layer $j=2$ and node k, the variable z will be:

$$z_k^{(2)} = \Theta_{k,0}^{(1)} x_0 + \Theta_{k,1}^{(1)} x_1 + \dots + \Theta_{k,n}^{(1)} x_n$$

- Examples

$\Theta^{(1)} = [-30 \quad 20 \quad 20]$

$h_\Theta(x) = g(-30 + 20x_1 + 20x_2)$

$x_1 = 0$ AND $x_2 = 0$ then $g(-30) \approx 0$

$x_1 = 0$ AND $x_2 = 1$ then $g(-10) \approx 0$

$x_1 = 1$ AND $x_2 = 0$ then $g(-10) \approx 0$

$x_1 = 1$ AND $x_2 = 1$ then $g(10) \approx 1$

- **Cost Function in neural networks**

L = layers in the network

S_l = Number of units in layer l

K = Number of output units/classes

The cost function for regularized logistic regression was:

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)} \log\left(h_\theta(x^{(i)})\right) + (1-y^{(i)}) \log\left(1-h_\theta(x^{(i)})\right)\right] + \frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2$$

For neural networks:

$$J(\Theta) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K}\left[y_k^{(i)} \log\left(\left(h_\Theta(x^{(i)})\right)_k\right) + (1-y_k^{(i)}) \log\left(1-\left(h_\Theta(x^{(i)})\right)_k\right)\right] + \cdots$$

(cost function)

$$\cdots \underbrace{\frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{S_l}\sum_{j=1}^{S_{l+1}}\left(\Theta_{j,i}^{(l)}\right)^2}$$

- the double sum simply adds up the logistic regression costs calculated for each cell in the output layer.
- The triple sum simply adds up the squares of all the individual Θs in the entire network
- the i in the triple sum doesn't refer to training example i

- **Backpropagation algorithm**

Training set $\{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$

Set $\Delta_{ij}^{(l)} = 0$ (for all l, i, j) *Used to compute $\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta)$

For $i = 1$ to m

1. Set $a^{(1)} = x^{(i)}$
2. Perform forward propagation to compute $a^{(l)}$ for $l = 2, 3, \ldots, L$
3. Using $y^{(i)}$, compute $\delta^{(L)} = a^{(L)} - y^{(i)}$
4. Compute $\delta^{(L-1)}, \delta^{(L-2)}, \ldots, \delta^{(2)}$
5. $\Delta_{ij}^{(l)} := \Delta_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$

$$D_{ij}^{(l)} := \frac{1}{m}\Delta_{ij}^{(l)} + \lambda \Theta_{ij}^{(l)} \quad \text{if } j \neq 0$$

$$D_{ij}^{(l)} := \frac{1}{m}\Delta_{ij}^{(l)} \quad \text{if } j = 0$$

$$\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = D_{ij}^{(l)}$$

2. Given one training example (x, y):

Forward propagation:

$a^{(1)} = x$

$z^{(2)} = \Theta^{(1)} a^{(1)}$

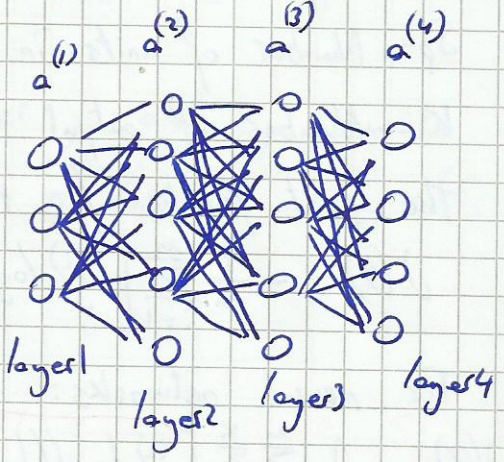
$a^{(2)} = g(z^{(2)})$ (add $a_0^{(2)}$)

$z^{(3)} = \Theta^{(2)} a^{(2)}$

$a^{(3)} = g(z^{(3)})$ (add $a_0^{(3)}$)

$z^{(4)} = \Theta^{(3)} a^{(3)}$

$a^{(4)} = h_\Theta(x) = g(z^{(4)})$



3. Using $y^{(t)}$, compute $\delta^{(L)} = a^{(L)} - y^{(t)}$

L = Total number of layers

$a^{(L)}$ = Vector of outputs of the activation units for the last layer.

Our "error values" for the last layer are simply the differences of our actual results in the last layer and the correct outputs in y

4. Compute $\delta^{(L-1)}, \delta^{(L-2)}, \ldots, \delta^{(2)}$ using:

$$\delta^{(l)} = ((\Theta^{(l)})^T \delta^{(l+1)}) .* \underbrace{a^{(l)} .* (1 - a^{(l)})}_{g'(z^{(l)})}$$

5. A vectorized form is $\Delta^{(l)} := \Delta^{(l)} + \delta^{(l+1)} (a^{(l)})^T$

Unrolling Parameters

Learning algorithm

Have initial parameters $\Theta^{(1)}, \Theta^{(2)}, \Theta^{(3)}$

Unroll to get initialTheta to pass to

fminunc(@costFunction, initialTheta, options)

function [jval, gradientVec] = costFunction (thetaVec)

From thetaVec, get $\Theta^{(1)}, \Theta^{(2)}, \Theta^{(3)}$

Use forward prop/back prop to compute $D^{(1)}, D^{(2)}, D^{(3)}$ and J

Unroll $D^{(1)}, D^{(2)}, D^{(3)}$ to get gradientVec

- **Gradient Checking**

We can approximate the derivate of our cost function with:

$$\frac{\partial}{\partial \Theta} J(\Theta) \simeq \frac{J(\Theta+\epsilon) - J(\Theta-\epsilon)}{2\epsilon}$$

A small value for ϵ such as $\epsilon = 10^{-4}$, guarantees that the math works out properly. If the value for ϵ is too small, we can end up with numerical problems

- **Random Initialization**

Initializing all theta weights to zero doesn't work with neural networks.

Symmetry breaking

Initialize each $\Theta_{ij}^{(l)}$ to a random value in $[-\epsilon, \epsilon]$

- **Putting it together**

- Number of input units = dimension of features $x^{(i)}$
- Number of output units = number of classes
- Number of hidden units per layer = usually more the better
- Defaults: 1 hidden layer, If you have more than 1 hidden layer, you have the same number of units in every hidden layer

Training a Neural Network

1. Randomly initialize the weights
2. Implement forward propagation to get $h_\Theta(x^{(i)})$ for any $x^{(i)}$
3. Implement the cost function
4. Implement backpropagation to compute partial derivatives
5. Use gradient checking to confirm that your backpropagation works. Then disable gradient checking.
6. Use gradient descent or a built-in optimization function to minimize the cost function with the weights in theta

Evaluating a hypothesis

Once we have done some trouble shooting for errors in our predictions by:

- Getting more training examples
- Trying smaller sets of features
- Trying additional features
- Trying polynomial features
- Increasing or decreasing λ

We can move on to evaluate our new hypothesis

Training set: 70% Test set: 30%

The new procedure using these two sets is then:

1. Learn Θ and minimize $J_{train}(\Theta)$ using the training set
2. Compute the test set error $J_{test}(\Theta)$

The test set error

1. For linear regression: $J_{test}(\Theta) = \frac{1}{2m_{test}} \sum_{i=1}^{m_{test}} \left(h_\Theta(x_{test}^{(i)}) - y_{test}^{(i)} \right)^2$
2. For classification ~ Misclassification

$$err(h_\Theta(x), y) = \begin{cases} 1 \text{ if } h_\Theta(x) \geq 0.5 \text{ \&\& } y = 0 \\ \quad \text{or} \\ \quad h_\Theta(x) < 0.5 \text{ \&\& } y = 1 \\ 0 \quad \text{otherwise} \end{cases}$$

The average test error for the test set is

$$\text{Test Error} = \frac{1}{m_{test}} \sum_{i=1}^{m_{test}} err\left(h_\Theta(x_{test}^{(i)}, y_{test}^{(i)} \right)$$

One way to break down our dataset into the 3 sets:

- Training set 60%
- Cross validation set 20%
- Test set 20%

We can now calculate three separate error values for the 3 different sets using the following method:

1. Optimize the parameters in Θ using the training set for each polynomial degree.
2. Find the polynomial degree d with the least error using the cross validation set
3. Estimate the generalization error using the test set with $J_{test}(\Theta^{(d)})$ (d = theta from polynomial with lower error)

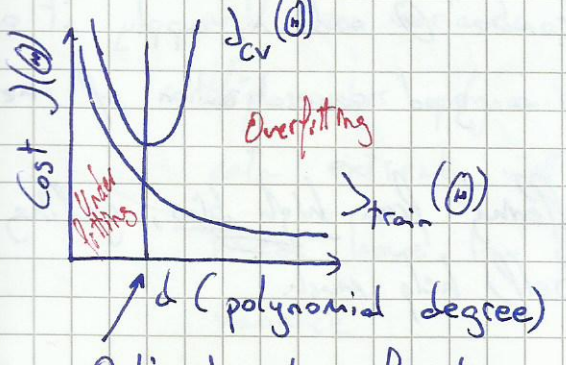
the degree of the polynomial d hasn't been trained using the test set.

- **Diagnosing Bias vs Variance**

High bias → Underfitting
High variance → Overfitting

the training error will tend to decrease as we increase the degree d of the polynomial.
At the same time, the cross validation error will tend to decrease as we increase d up to a point, and then it will increase as d is increased, forming a convex curve.

High bias (underfitting): $J_{CV}(\Theta)$ and $J_{train}(\Theta)$ will be high
High variance (overfitting): $J_{train}(\Theta)$ will be low and $J_{CV}(\Theta)$ will be much greater than $J_{train}(\Theta)$



Regularization and Bias/Variance

Model: $h_\theta(x) = \theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 + \theta_4 x^4$

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

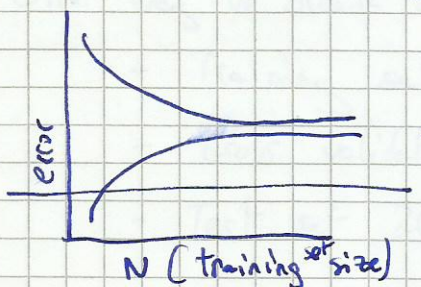
Large λ = high bias (underfit)

Small λ = high variance (overfit)

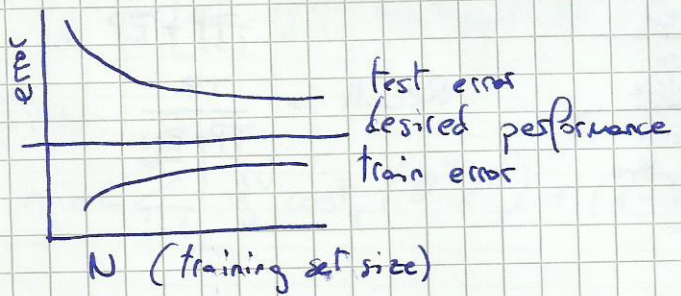
To choose the model and the regularization term λ, we need to:

1. Create a list of lambdas
   (i.e. $\lambda \in \{0, 0.01, 0.02, 0.04, 0.08, 0.16, 0.32, 0.64, 1.28, 2.56, 5.12, 10.24\}$
2. Create a set of models with different degrees or any other variants
3. Iterate through the λs and for each λ go through all the models to learn some Θ.
4. Compute the cross validation error using the learned Θ (computed with λ) on the $J_{cv}(\Theta)$ without regularization or $\lambda = 0$
5. Select the best combo that produces the lowest error on the cross validation set
6. Using the best combo Θ and λ, apply it on $J_{test}(\Theta)$ to see if it has a good generalization of the problem

If a learning algorithm is suffering from high bias, getting more training data will not (by itself) help much.



If a learning algorithm is suffering from high variance, getting more training data is likely to help



- Deciding what to do next revisted

Our decission process can be broken down as follows:

- Getting more training examples. Fixes high variance
- Trying smaller sets of features. Fixes high variance
- Adding features. Fixes high bias
- Adding polynomial features. Fixes high bias
- Decreasing λ. Fixes high bias
- Increasing λ. Fixes high variance

A neural network with fewer parameters is prone to underfitting. It's also computationally cheaper

A large neural network with more parameters is prone to overfitting.

Model complexity effects:

- Lower-order polynomials (low model complexity) have high bias and low variance. The model fits poorly consistenly
- High-order polynomials (high model complexity) fit the training data extremly well and the test data extremly poorly. These have low bias on the training data, but very high variance

μ_k — average (mean) of points assigned to cluster k

• Precision / Recall /

		Actual class: 1	Actual class: 0
Predicted class	1	True Positive	False positive
	0	False negative	True Negative

$$\text{Precision} = \frac{TP}{TP + FP}$$

$$\text{Recall} = \frac{TP}{TP + FN}$$

In regression:

$$\text{Precision} = \frac{TP}{\text{n° of predicted positives}}$$

$$\text{Recall} = \frac{TP}{\text{n° of actual positive}}$$

To compare precision and recall is necessary create

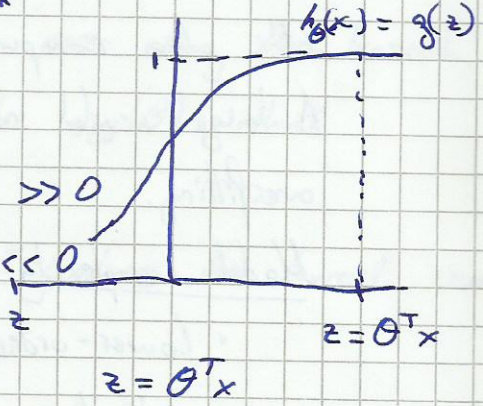
- score model (no average)

$$F_1 \text{ score} = 2\frac{P \cdot R}{P + R}$$

• Support Vector Machines

Alternative view of logistic regression

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



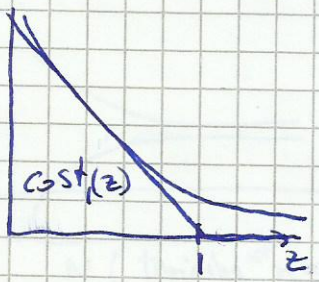
$z = \theta^T x$

- If $y = 1$, we want $h_\theta(x) \simeq 1$, $\theta^T x >> 0$
- If $y = 0$, we want $h_\theta(x) \simeq 0$, $\theta^T x << 0$

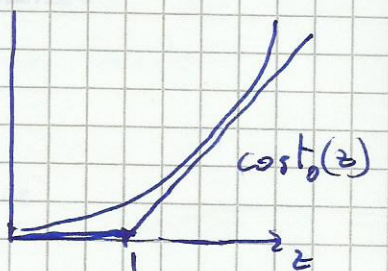
Cost of example: $-(y \log h_\theta(x) + (1 - y) \log(1 - h_\theta(x))) =$

$$= -y \underbrace{\log \frac{1}{1 + e^{-\theta^T x}}}_{\text{If } y = 1 \ (\theta^T x >> 0)} - (1 - y) \underbrace{\log\left(1 - \frac{1}{1 + e^{-\theta^T x}}\right)}_{\text{If } y = 0 \ (\theta^T x << 0)}$$

If $y = 1$ $(\theta^T x >> 0)$



If $y = 0$ $(\theta^T x << 0)$



Logistic regression:

$$\min_\theta \frac{1}{m}\left[\sum_{i=1}^{m} y^{(i)}\left(-\log h_\theta(x^{(i)})\right) + (1-y^{(i)})\left(\left(-\log(1-h_\theta(x^{(i)}))\right)\right)\right] + \frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2$$

SVM:

$$\min_\theta C\sum_{i=1}^{m}\left[y^{(i)}\,\text{cost}_1(\theta^T x^{(i)}) + (1-y^{(i)})\,\text{cost}_0(\theta^T x^{(i)})\right] + \frac{1}{2}\sum_{i=1}^{n}\theta_j^2$$

Kernel

Given x, $f_1 = \underset{\text{Kernel}}{\text{similarity}}(x, l^{(1)}) = \underset{\text{Gaussian Kernels}}{\exp}\left(-\frac{\|x-l^{(1)}\|^2}{2\sigma^2}\right)$

If $x \approx l^{(1)}$, $f_1 \approx 1$; If x is far from $l^{(1)}$, $f_1 \approx 0$

$C = \frac{1}{\lambda}$. Large C: lower bias, high variance

Small C: higher bias, low variance

Logistic regression vs SVMs

n = number of features ($x \in \mathbb{R}^{n+1}$)

m = number of training examples

If n is large (relative to m):

Use logistic regression, or SVMs without a kernel ("linear kernel")

If n is small, m is intermediate:

Use SVM with Gaussian kernel

If n is small, m is large:

Create/add more features, then use logistic regression or SVM without a kernel

- **Clustering**

K-means

Randomly initialize K cluster centroids $\mu_1, \mu_2, \ldots, \mu_K \in \mathbb{R}^n$

Repeat {

for $i = 1$ to m

$c^{(i)}$:= index of cluster centroid closest to $x^{(i)}$ $\left(\min_k \|x^{(i)} - \mu_k\|^2\right)$

for $k = 1$ to K

μ_k := average (mean) of points assigned to cluster k

}

Random initialization

For $i = 1$ to 100 {

Randomly initialize k-means

Run k-means. Get $c^{(1)}, \dots, c^{(m)}, \mu_1, \dots, \mu_k$

Compute cost function (distortion)

$J(c^{(1)}, \dots, c^{(m)}, \mu_1, \dots, \mu_k)$

}

To choose the number of clusters, we could use the "elbow metho

- ## Data Compression

Reduce from 2-dimension to 1-dimension:

Find a direction (a vector $a^{(1)} \in \mathbb{R}^n$) onto which to project the data so as to minimize the projection error

Typically, choose k to be smallest value so that

$$\frac{\frac{1}{m}\sum_{i=1}^{m} \| x^{(i)} - x_{approx}^{(i)} \|^2}{\frac{1}{m}\sum_{i=1}^{m} \| x^{(i)} \|^2}$$

99% variance is ~~retined~~ retained