Linear Regression

THE 2ND BRAIN, THE 3RD EYE

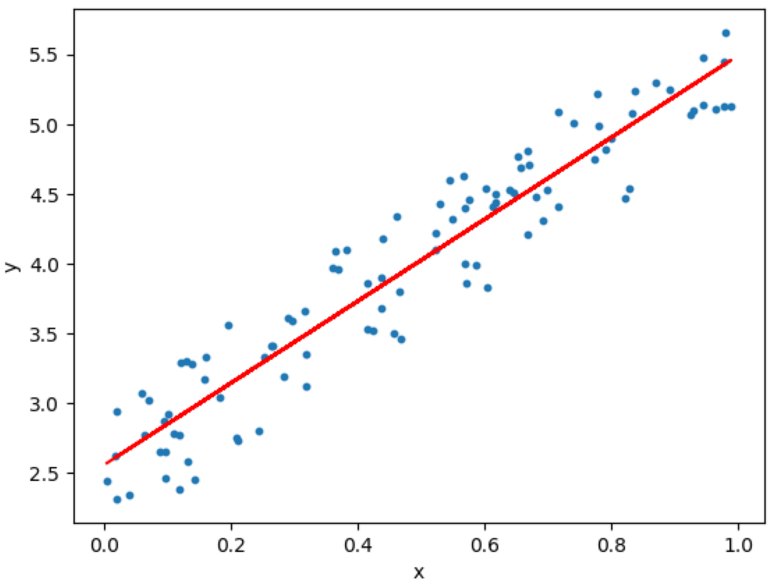
JP @ 2B3E

Contents

- Linear regression
- Multi-variate linear regression
- Gradient descent for linear regression

Linear regression

- Solving a regression problem, with a linear function



Linear regression with multiple features

- Multiple variables = multiple features
- House price prediction with one feature
  - X = house size, use this to predict
  - y = house price
- If we have more than one feature (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)

Linear regression with multiple features

- More notation **n**
  - number of features (n = 4)
- **m**
  - number of examples (i.e. number of rows in a table)
- $\mathbf{x}^i$
  - 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 i-th training example
  - So
    - x_2^3 is, for example, the number of bedrooms in the third house

Linear regression with multiple features

- hypothesis
  - Hypothesis with one feature
    - $h_\theta(x) = \theta_0 + \theta_1 x$
      - two parameters (including the bias)
      - One variable x
  - Hypothesis with 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

Linear regression with multiple features

- For convenience of notation, $x_0 = 1$
  - Bias: an additional 0th feature for each example
  - **feature vector**
    - n + 1 dimensional feature vector indexed from 0
    - a column **X**: each example has a column vector associated with it
  - **Parameters**
    - are n+1 dimensional vector
    - This is also a column vector called $\boldsymbol{\theta}$
    - This vector is the same for each example
- Considering this, hypothesis can be written
  - $h_\theta(x) = \theta_0 x_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 + \theta_4 x_4$

Linear regression with multiple features

- Vector form: $h_\theta(x) = \theta^T X$
  - θ^T is an [1 x n+1] matrix
  - θ: a column vector —> θ^T a row vector
  - θ: [(n+1) x 1]
  - θ^T : [1 x (n+1)]
  - $\theta^T X$
    - [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 is an example of multivariate linear regression

Gradient descent for multiple variables

- Fitting parameters for the hypothesis with gradient descent
  - Parameters are θ_0 to θ_n
  - Instead of thinking about this as *n* separate values, think about the parameters as a single vector (θ)
    - Where θ is n+1 dimensional
- Our cost function is

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

Gradient descent for multiple variables

- Instead of thinking of J as a function of the n+1 numbers, J() is just a function of the parameter vector J(θ)

Repeat $\{$

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

$\}$ (simultaneously update for every $j = 0, \ldots, n$)

- This is $\theta_j = \theta_j$ - learning rate (α) times the partial derivative of J(θ) with respect to $\theta_{J(\ldots)}$
- We do this through a **simultaneous update** of every θ_j value

Gradient descent for multiple variables

- Implementing this algorithm
  - When n = 1

Repeat {

$$\theta_0 := \theta_0 - \alpha \underbrace{\frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})}_{\frac{\partial}{\partial \theta_0} J(\theta)}$$

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

(simultaneously update θ_0, θ_1) }

- Above, we have slightly different update rules for θ_0 and θ_1
  - Actually they're the same, except the end has a previously undefined $x_0^{(i)}$ as 1, so wasn't shown
- —> almost identical rule for multivariate gradient descent

Gradient descent for multiple variables

New algorithm $(n \geq 1)$:

Repeat {

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

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

(simultaneously update θ_j for $j = 0, \ldots, n$) }

- We're doing this for each j (0 until n) as a simultaneous update (like when n = 1)
- So, we re-set θ_j to
  - θ_j minus the learning rate (α) times the partial derivative of the θ vector with respect to θ_j

Gradient descent for multiple variables

- In non-calculus words, this means that we do
  - Learning rate
  - Times 1/m (makes the maths easier)
  - Times the sum of
    - The hypothesis taking in the variable vector, minus the actual value, times the j-th value in that variable vector for EACH example
- It's important to remember that
- $$\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)} = \frac{\partial}{\partial\theta_j}J(\theta)$$

Gradient Decent in practice

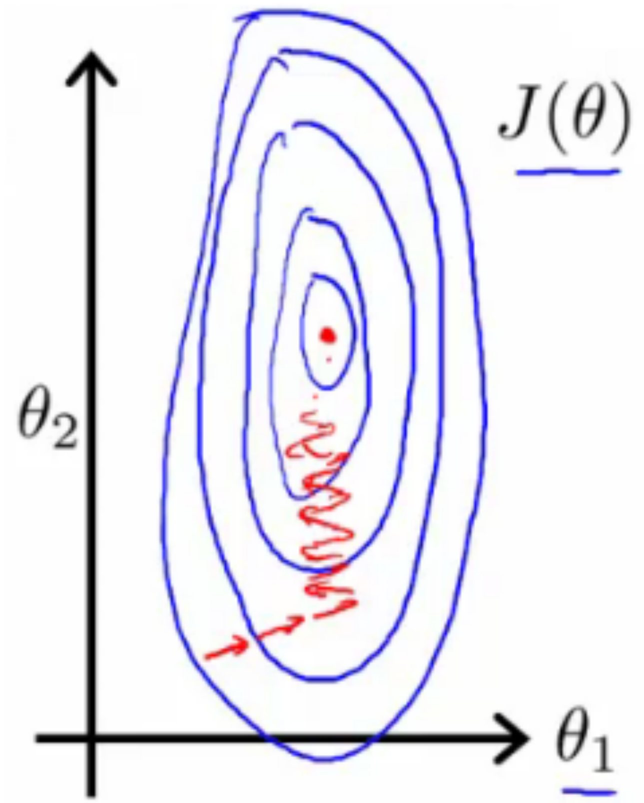
- Feature scaling
- Learning rate

Feature Scaling

- Having covered the theory
    - —> move on to learn about some of the practical tricks
- Feature scaling
    - If you have a problem with multiple features
    - You should make sure those features have a similar scale
        - Means gradient descent will converge more quickly
    - e.g.
        - x1 = size (0 - 2000 feet)
        - x2 = number of bedrooms (1-5)
        - Means the contours generated if we plot θ_1 vs. θ_2 give a very tall and thin shape due to the huge range difference
    - Running gradient descent on this kind of cost function can take a long time to find the global minimum

Feature Scaling

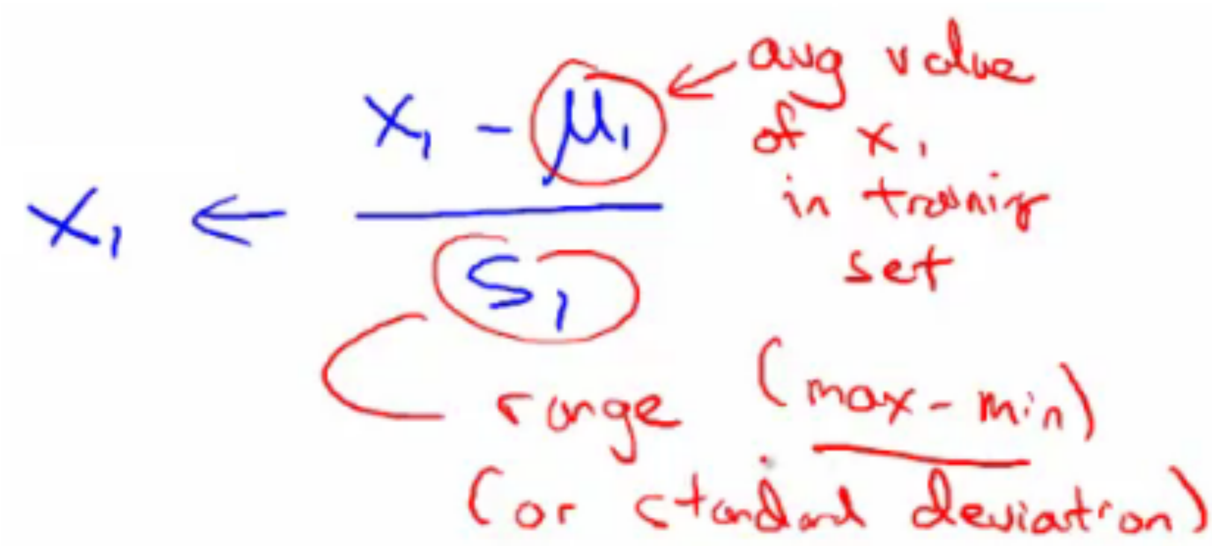
- Pathological input to gradient descent
- So we need to rescale this input so it's more effective
- So, if you define each value from x1 and x2 by dividing by the max for each feature
- Contours become more like circles (as scaled between 0 and 1)



Feature Scaling

- May want to get everything into -1 to +1 range (approximately)
  - Want to avoid large ranges, small ranges or very different ranges from one another
  - Rule a thumb regarding acceptable ranges
    - -3 to +3 is generally fine - any bigger bad
    - -1/3 to +1/3 is ok - any smaller bad
- Can do **mean normalization**
  - Take a feature x_i
    - Replace it by $(x_i$ - mean)/max
    - So your values all have an average about 0

Feature Scaling



- Instead of max can also use standard deviation

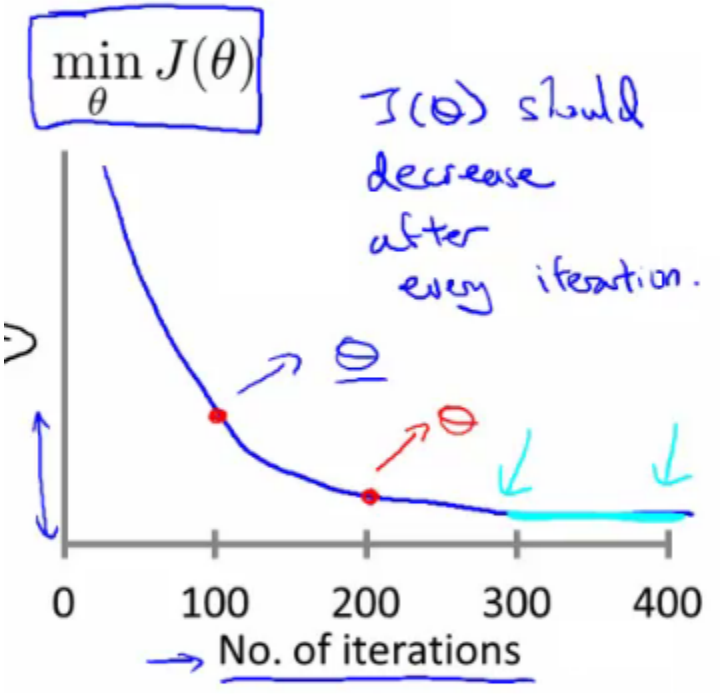
Learning Rate α

- Focus on the learning rate (α)
- Topics
  - Update rule
  - Debugging
  - How to chose α

Learning Rate α

- **Make sure gradient descent is working**
  - Plot min J(θ) vs. # of iterations
    - (i.e. plotting J(θ) over the course of gradient descent
- If gradient descent is working then J(θ) should decrease after (almost) every iteration
- Can also show if you're not making huge gains after a certain number
  - Can apply heuristics to reduce number of iterations
  - If, for example, after 1000 iterations you reduce the parameters by nearly nothing you could chose to only run 1000 iterations in the future
  - Make sure you don't accidentally hard-code thresholds

Learning Rate α



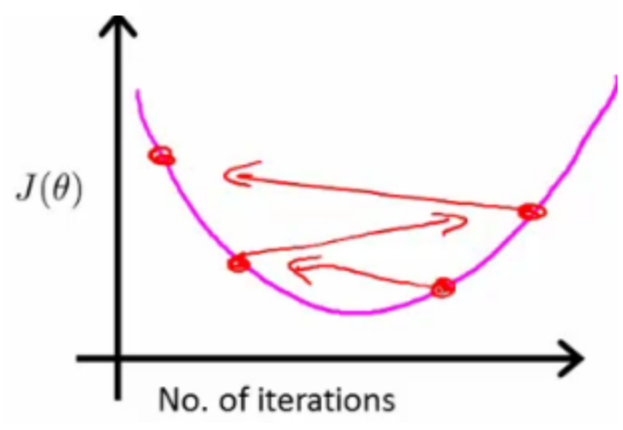
- Number of iterations varies a lot
    - 30 iterations
    - 3000 iterations
    - 3000 000 iterations
- Very hard to tell in advance how many iterations will be needed
- Can often make a guess based a plot like this after the first 100 or so iterations

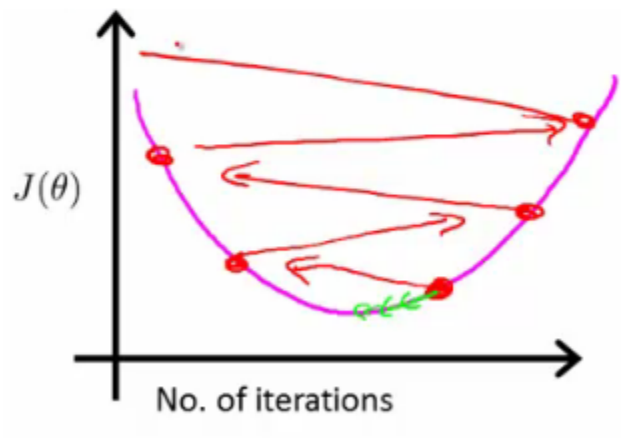
Learning Rate α

- Automatic convergence tests
  - Check if J(θ) changes by a small threshold or less
    - Choosing this threshold is hard
    - So often easier to check for a straight line
      - Why? - Because we're seeing the straightness in the context of the whole algorithm

Learning Rate α

- Checking its working
  - If you plot J(θ) vs iterations and see the value is increasing - means you probably need a smaller α
    - Cause is because your minimizing a function which looks like this





But you overshoot, so reduce learning rate so you actually reach the minimum (green line) → So, use a smaller α

Learning Rate α

- Another problem might be if J(θ) looks like a series of waves
  - Here again, you need a smaller α
- However
  - If α is small enough, J(θ) will decrease on every iteration
  - BUT, if α is too small then rate is too slow
    - A less steep incline is indicative of a slow convergence, because we're decreasing by less on each iteration than a steeper slope
- Typically
  - Try a range of alpha values
  - Plot J(θ) vs number of iterations for each version of alpha
  - Go for roughly threefold (or fivefold) increases
    - 0.001, 0.003, 0.01, 0.03. 0.1, 0.3