CS 613 Machine Learning

Most Machine Learning algorithms require some idea of similarity and/or distance between observations.

Which function you choose to use often depends on the nature of the data, the equation you're using it in, etc...

As a reference here's some of the most common similarity/distance functions:

Cosine Similarity

Aims to measure similarity based on the angle between two vectors relative to some origin. The function uses the dot product definition of cosine. Due to the nature of cosine the similarity value will be in the range of [-1, 1].

$$similarity(A, B) = \cos(\theta) = \frac{A \cdot B}{||A||||B||} = \frac{\sum_{i=1}^{D} A_i B_i}{\sqrt{\sum_{i=1}^{D} A_i^2} \sqrt{\sum_{i=1}^{D} B_i^2}}$$

Gaussian Kernel

The Gaussian Kernel, sometimes known as the *radial basis function kernel* measures the similarity between two objects. The numerator of the exponent is the squared Euclidean distance so when this distance is zero, the similarity is one. Likewise when the squared Euclidean distance is infinite, the similarity is zero.

We can control the fall-off rate by specifying the σ value; a larger sigma will result is slower fall-off.

$$similarity(A, B) = e^{-\frac{\sum_{i=1}^{D}(A_i - B_i)^2}{2\sigma^2}} = e^{-\frac{||A-B||^2}{2\sigma^2}}$$

Histogram Intersection

As the name implied, histogram intersection is often used to compute the intersection of two histograms, resulting in a similarity measurement:

$$similarity(A, B) = \sum_{i=1}^{D} \min(A_i, B_,)$$

Euclidean Distance

The Euclidean Distance, sometimes referred to as the L2-Norm, is the straight line distance between two points.

$$distance(A, B) = \sqrt{\sum_{i=1}^{D} (A_i - B_i)^2}$$

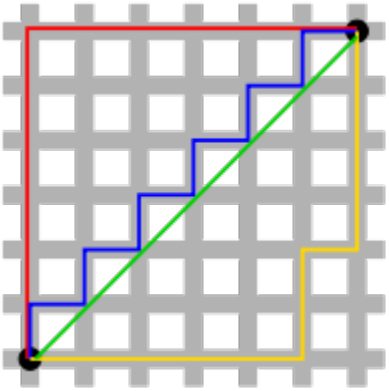
This is typically a good choice if doing things like rotation in the space has some meaning.

Manhattan Distance

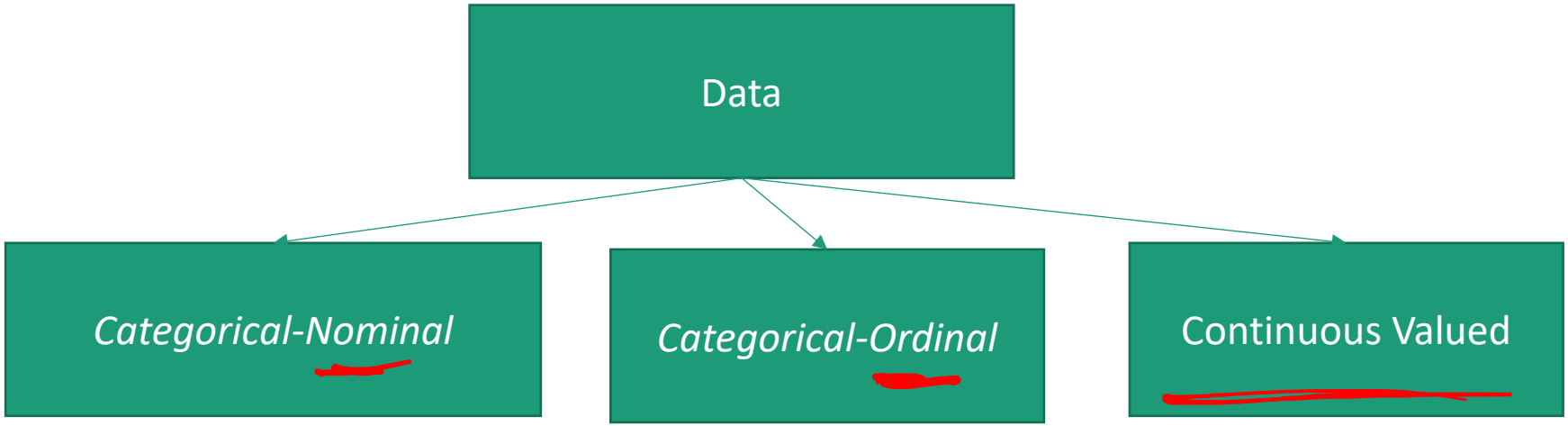
The Manhattan Distance, also referred to as the City Block Distance, Taxicab Distance, or L1 Norm, measures the distance between two points as the sum of their difference along each axis.

$$distance(A, B) = \sum_{i=1}^{D} |A_i - B_i|$$

This is typically a good choice if doing things like rotation in the space does **not** make sense.

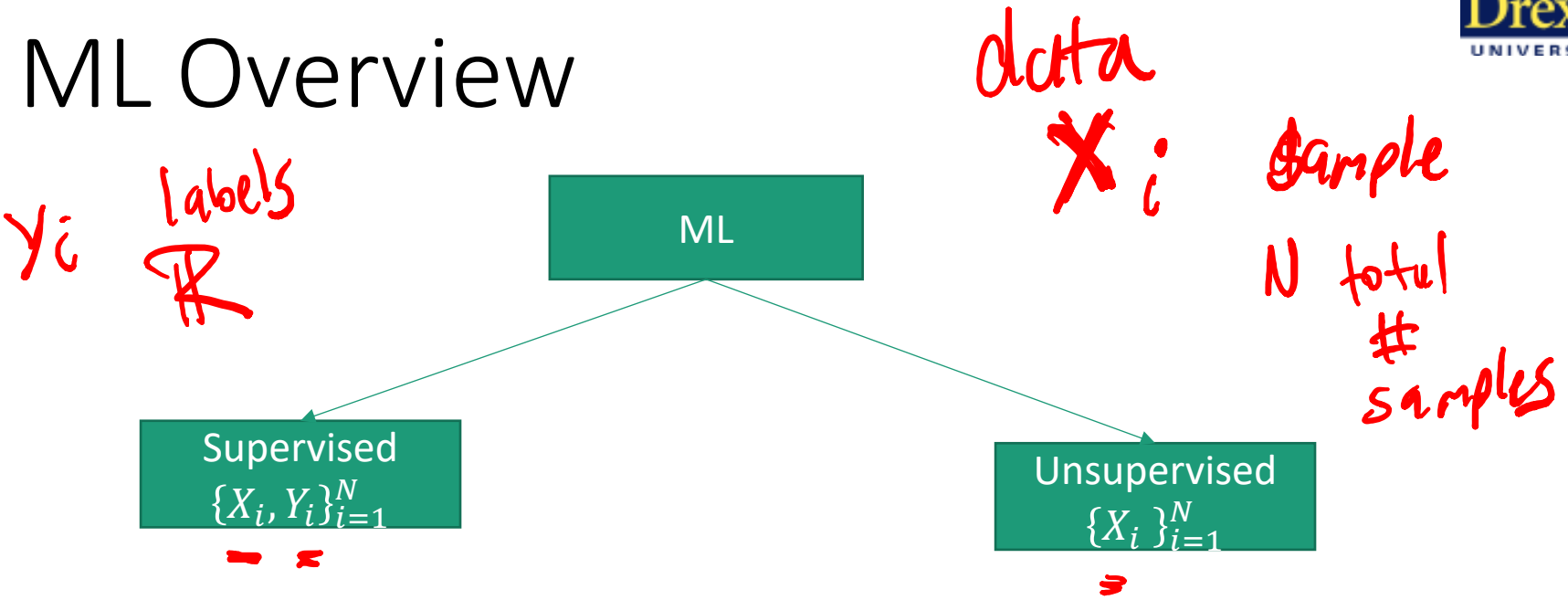


Types of Data



- Each piece of information pertaining to an observation can fall into one of three categories:
  - *Categorical-Nominal (unordered)*
    - Examples: Car Model, School
  - *Categorical-Ordinal (can be ordered)*
    - Examples: Colors, small < medium < large
  - *Continuous Valued*
    - Examples: Blood Pressure, Height

ML Overview



- We can basically break machine learning tasks into two categories
  1. Supervised Learning
  2. Unsupervised Learning
- *Supervised learning*
  - Data X_i and correct answer (label) Y_i given for each example $i \in \{1, \dots, N\}$
- *Unsupervised learning*
  - Only data given for each example, X_i

Types of Machine Learning

- Supervised Learning
  - Learning by example
- Unsupervised Learning
  - Learning by exposure to the raw data distribution
- Reinforcement Learning
  - Learning from right/wrong feedback
- Transfer Learning
  - Learning one task helping you to learn a related task

least important future

Percentage Use

96%

3%

1%

2%

ML Algorithms

- Here's a list of algorithms we'll look at in the class

1. Linear Regression
2. Classification
    a. Probabilistic Decisions (Inference, Bayesian Learning, Decision Trees)
    b. Nearest Neighbors
    c. Support Vector Machines
    d. Logistic Regression
    e. Introduction to Artificial/Deep Networks
3. Feature Reduction (Feature Selection, Feature Projection)
4. Unsupervised Learning
    a. K-Means
    b. Mixture Models
5. Neuro-inspired Machine Learning Data
    a. Neural networks
    b. Sparse Coding
    c. Intro to Recurrent Neural Networks (RNNs)

No Free Lunch Theorem

- The "free lunch" refers to the once-common tradition of saloons in the United States providing a "free" lunch to patrons who had purchased at least one drink. Many foods on offer were high in salt (e.g., ham, cheese, and salted crackers), so those who ate them ended up buying a lot of beer.

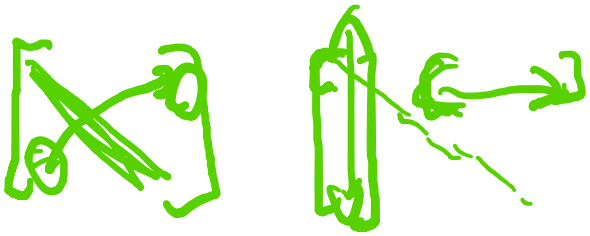
- No one method dominates all others over all possible datasets. On a particular dataset, one specific method may work best, but some other method may work better on a similar but different dataset
- To get superior flexibility on a specific problem you make assumptions
- This reduces the generability of the algorithm

No Free Lunch Theorem

- Questions we should ask ourselves are:
  - Do we have labels?
  - Do we want to do clustering?
  - Do we know the number of clusters?
  - Do we want to do regression?
  - Do we want to do classification?
  - Is understanding/interpreting the learned system important?
  - What are our memory and/or processing time limitations?
  - Do we have categorical or continuous-valued data?
  - Do I have enough data?
  - Do I have enough features? Too many?
  - Is linear ok or do I need non-linear?
  - And more…

Line – 60%

- Y = mx + b



dot product

$\theta_1 x_1$ $\theta_0 \cdot 1$

slope intercept

$$\theta = \begin{bmatrix} \theta_1 \\ \theta_2 \\ \vdots \\ \theta_N \end{bmatrix} \quad X = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_N \end{bmatrix}$$

$\theta_1 x_1 + \theta_2 x_2 \ldots \theta_N x_N$

$ax + by = c$

$by = -ax + c$

$y = -\frac{a}{b}x + \frac{c}{b}$

neural net curves

$2D \to 3D \to 4D \to 5D$ hyperplane

$ax + by + cz = d$

$\theta_1 x_1 + \theta_2 x_2 + \ldots \theta_N x_N$

$\theta \cdot X$

$\theta^T X$

50 [ ] 50 [ ]

50 x 1 · 50 x 1

1 x 50 50 x 1

1 x 1 scalar

$\theta^T x + b$ perceptron

$\theta^T x$ ← linear regression

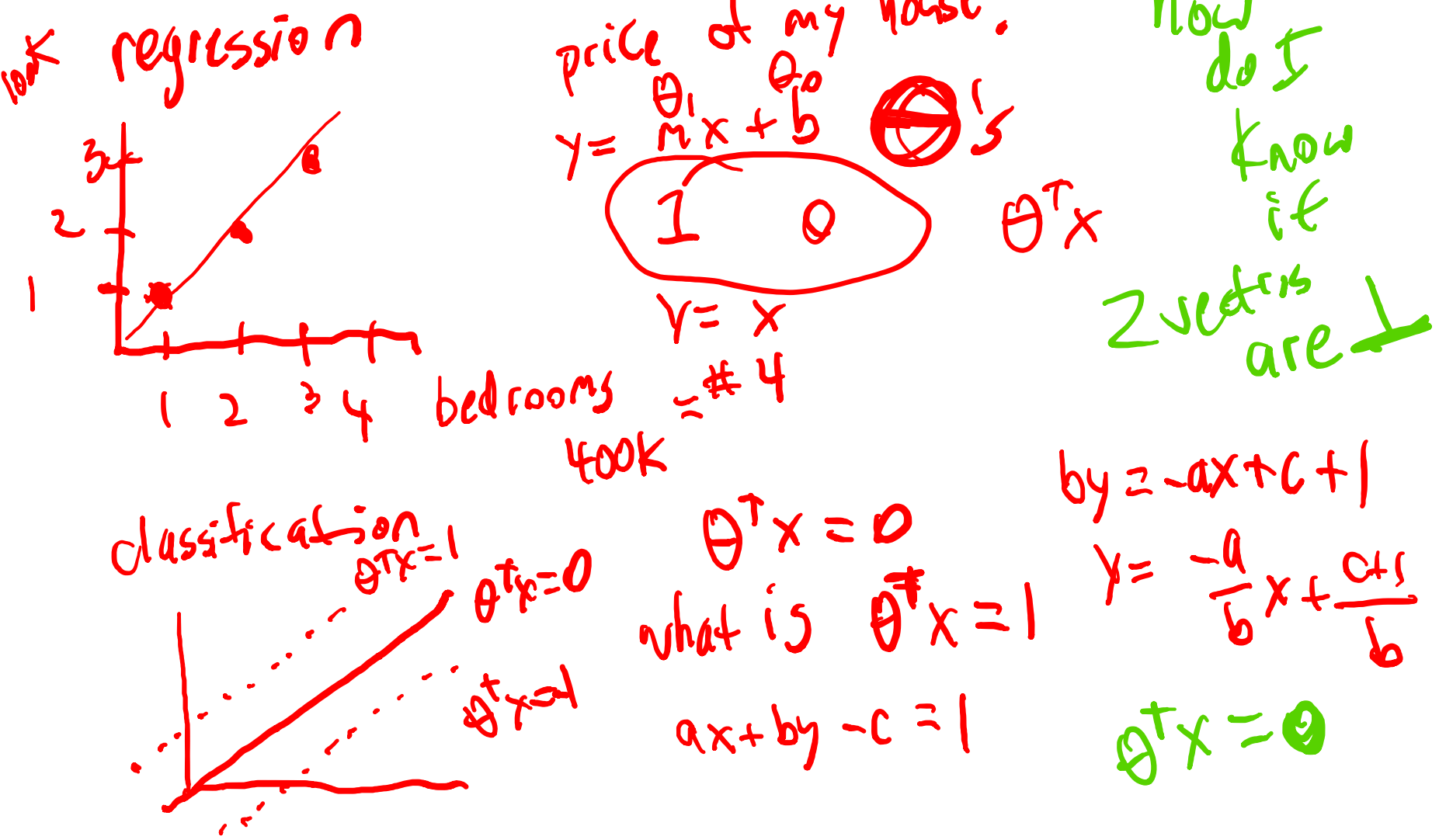
$ax + by - c = 0$

$\theta^T x = 0$ logistic regression

$$\theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \vdots \\ \theta_N \end{bmatrix} \quad x = \begin{bmatrix} 1 \\ x_1 \\ \vdots \\ x_N \end{bmatrix}$$

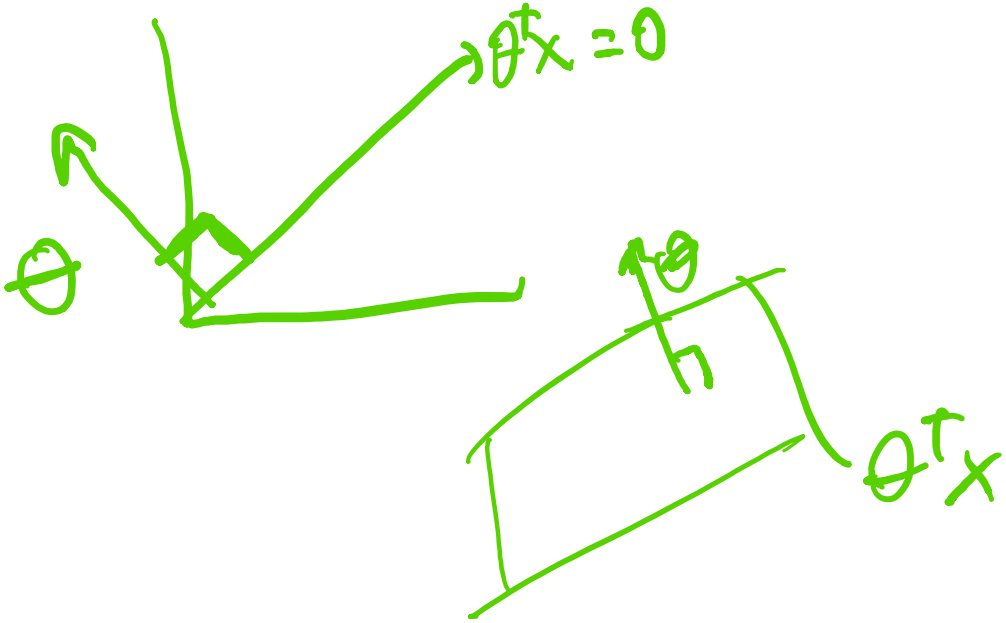
sum

How to use lines



Dot product = 0, when…

- 2 vectors are perpendicular



$b = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$

θ

$a = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$

$0 \cdot 1 + 1 \cdot 0 = 0$

$a^T b = \|a\| \|b\| \cos\theta$

$\cos(90)$

θ

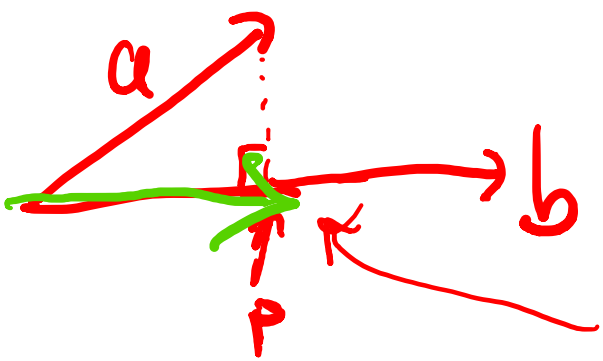
$a = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix}$

$\|a\| = \sqrt{a_1^2 + a_2^2 + a_3^2 \dots a_N^2}$

Euclidean distance

L_2 norm

Projections → shadows of one vector onto another



$a \cdot b = \|a\| \|b\| \cos\theta$

$\frac{a \cdot b}{\|b\|} = \|a\| \cos\theta$

scalar projection

vector projection

unit vector

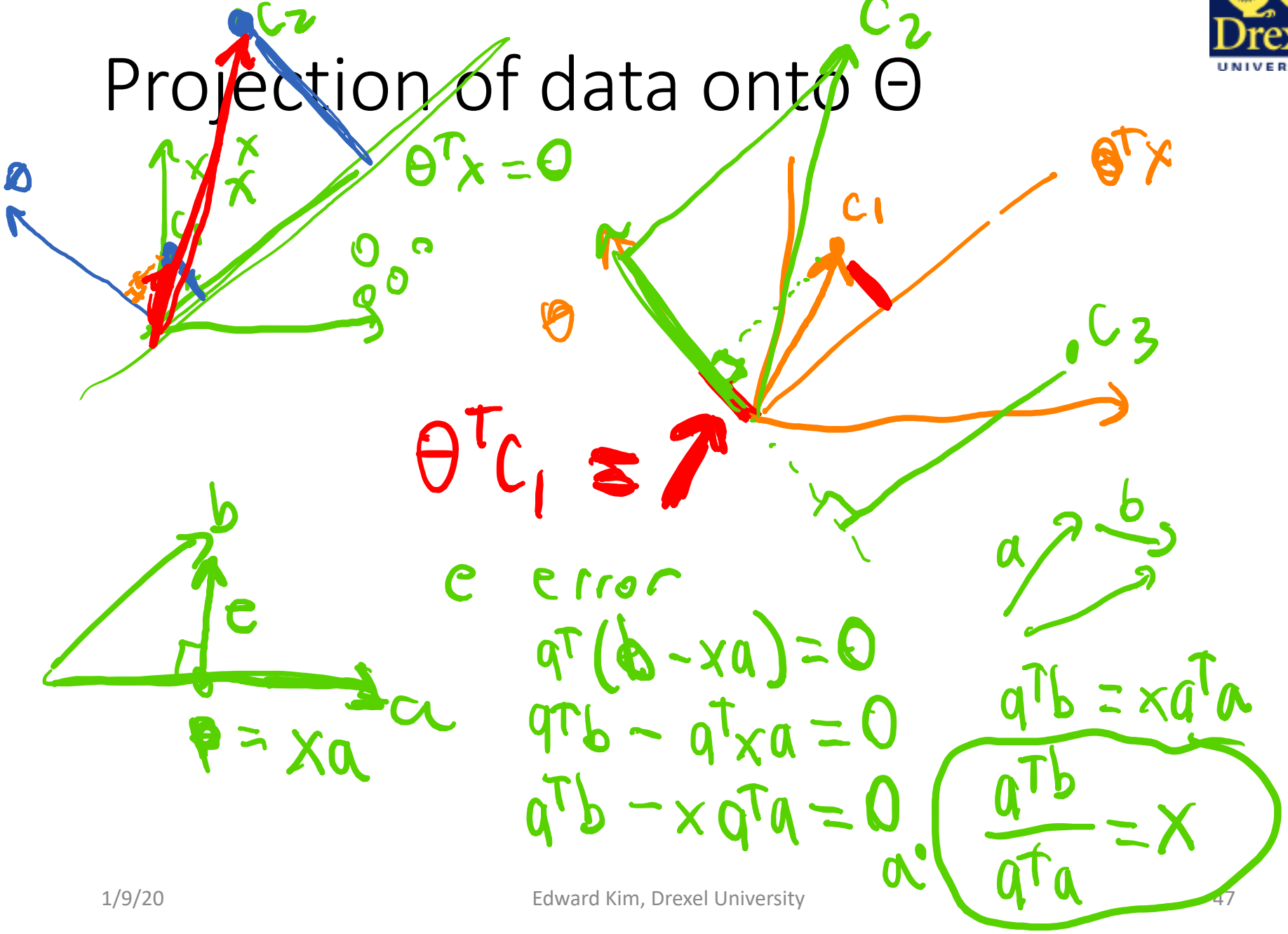
what if b is already unit vector

$\theta^T x$ ~~θx~~ $a \cdot b \Leftrightarrow b \cdot a$

$\frac{a \cdot b}{\|b\|} \cdot \frac{b}{\|b\|} \Rightarrow \frac{\|b\|^2}{b^T b}$ $\|b\|^2 = b_1^2 + \dots b_n^2$

$\frac{a^T b}{b^T b} \cdot b$

Projection of data onto Θ



$\theta^T x$

$(X^T X)^{-1} X^T y$ → vector projection

linear regression

$\frac{a^T b}{q^T a}$

Preparing Data

$Ax = b$

A^{-1} doesn't exist

A is not square

↑

$(A^T A)^{-1} (A^T A)x = A^T b$

pseudo inverse

square invertible

$(A^T A)^{-1} A^T$

Parsing Text Files

np.genfromtext
pandas

- The first thing we'll need to do in this class is get data!
- We'll get data from text files
- But unfortunately they may be in all different formats ☹
  - If we're lucky it may be in .csv format and we can use a library to read this (like python's `csv` library)
  - But even then there may be mixed data types making it impossible.
- So one of the first things you'll want to do is write a file parser.

Feature Types

isCat $\in [0,1]$ 0.8
isDog $\in [0,1]$ 0.7
isMouse $\in [0,1]$ 0.3
| Binary

- Let's assume that we now have our data in our system.
- The next thing to consider is the nature of our data.
- As previously mentioned, we categorize each feature as one of three types:
  1. Continuous valued → floating
  2. Categorical-Nominal → "name" → {cat, dog, mouse} → 0 1 2
  3. Categorical-Ordinal → "ordered" → {small, medium, large} → 0 1 2
- Depending on the algorithm being used we may need to
  - Convert categorical features into a set of binary features
  - "Standardize" features so that they have the same range

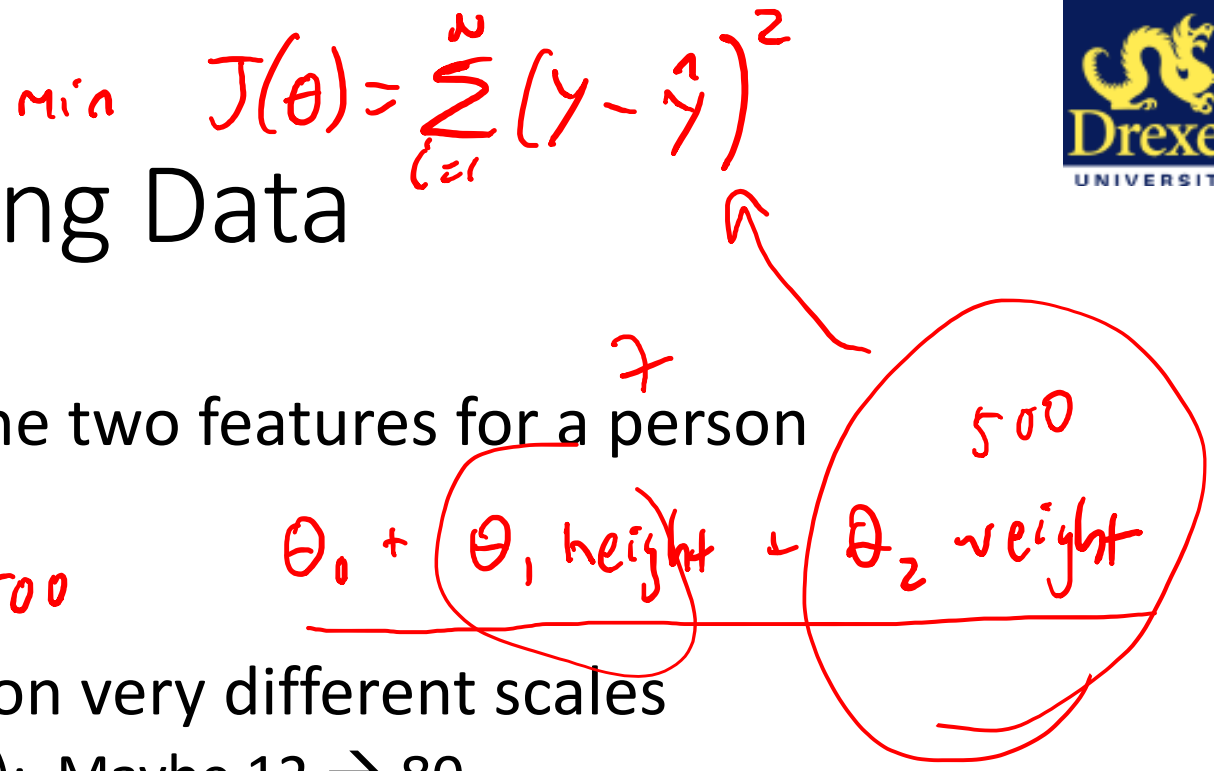
1.5 ?

1.5

Prepping Data

- Categorical → Binary
  - If our algorithm computes some floating value/location of a feature then it wouldn't make sense to have a floating point value of a categorical feature
    - Imagine if a feature is car make $x_1 \in \{Honda, Ford, Toyota\} \rightarrow \{0,1,2\}$
    - What would the meaning of 0.7 be?
  - Instead we'll want to create a set of binary features from our enumerations
    - $isHonda \in [0,1], isFord \in [0,1], isToyota \in [0,1]$
- Standardizing Features
  - If our algorithm incrementally updates parameters, then we may want all features to have the same variance.
  - Otherwise ones with grader range will have greater influence than ones with a smaller range

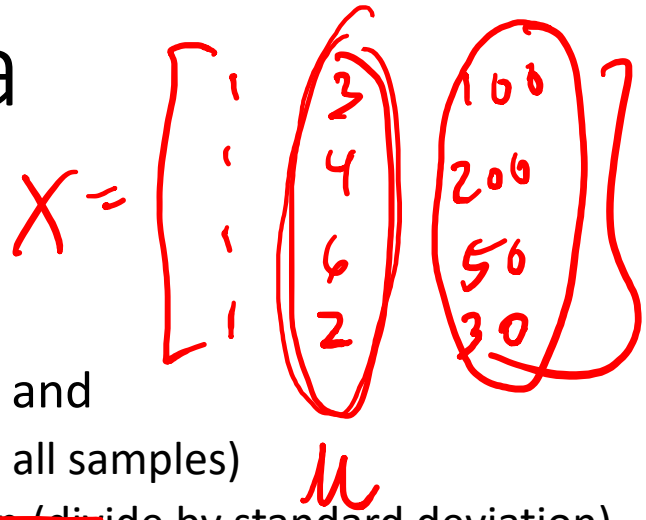
Standardizing Data



- Example: Imagine two features for a person
  - Height 0 – 7
  - Weight 0 – 500
- These are both on very different scales
  - Height (inches): Maybe 12 → 80
  - Weight (lbs): Maybe 5→400
- If we used the data as-is, then one feature may have more influence than the other
  - Depends on the algorithm

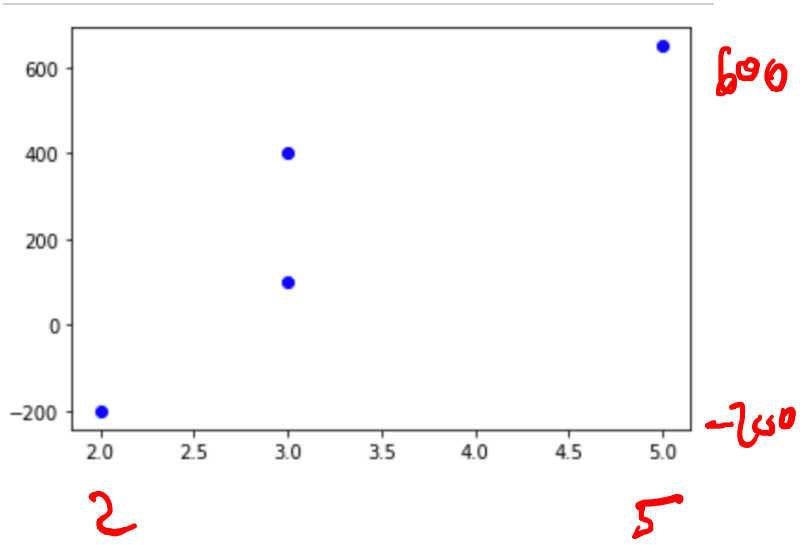
Standardizing Data

- Standardized data has
  - Zero mean
  - Unit deviation
- We treat each feature independently and
  1. Center it (subtract the mean from all samples)
  2. Make them all have the same span (divide by standard deviation).
- Example



```
import numpy as np
import matplotlib.pyplot as plt

X = [[3,400],
     [2,-200],
     [3, 100],
     [5,650]]
X = np.array(X)
plt.plot(X[:,0], X[:,1],'bo')
plt.show()
```

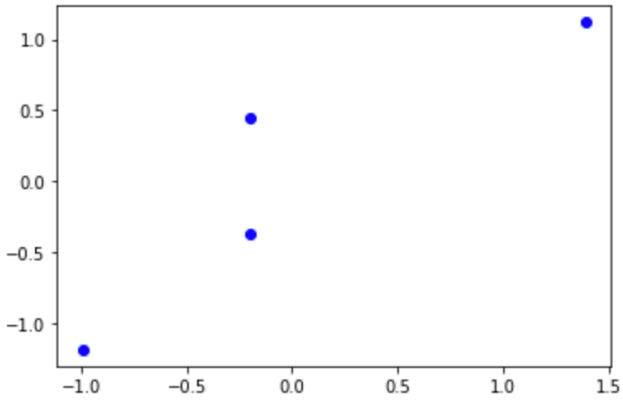


Standardizing Data

- We can compute the mean and standard deviation of each feature easily and then subtract the means from each observation and divide each (centered) observation by the standard deviation of each feature.
- Would it make sense to do this with categorical data?
- How about if it was made into a binary feature set?

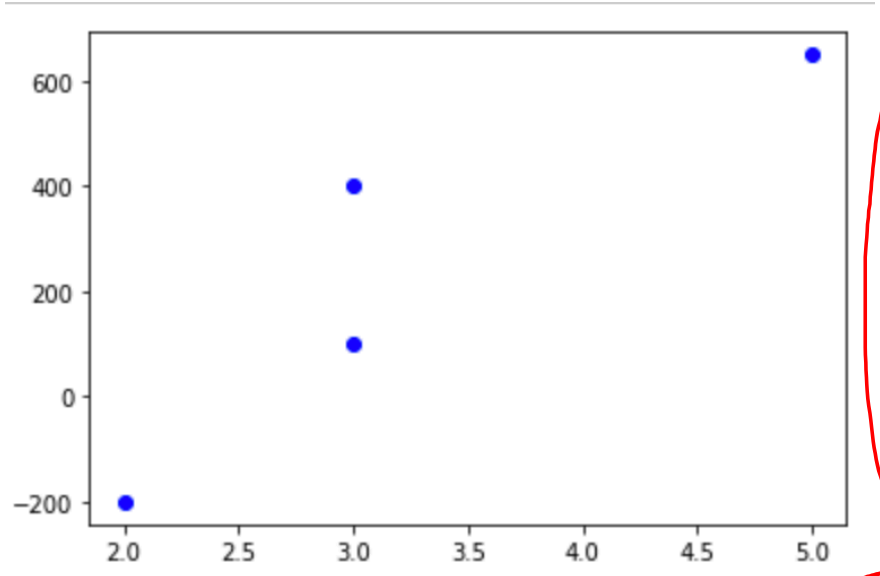
```
mean = np.mean(X, axis=0)
std = np.std(X, axis=0, ddof=1)
# print(mean, std)
# [ 3.25 237.5 ] [ 1.25830574
368.27299657]

sX = (X-mean)/std
plt.plot(sX[:,0], sX[:,1],'bo')
plt.show()
```

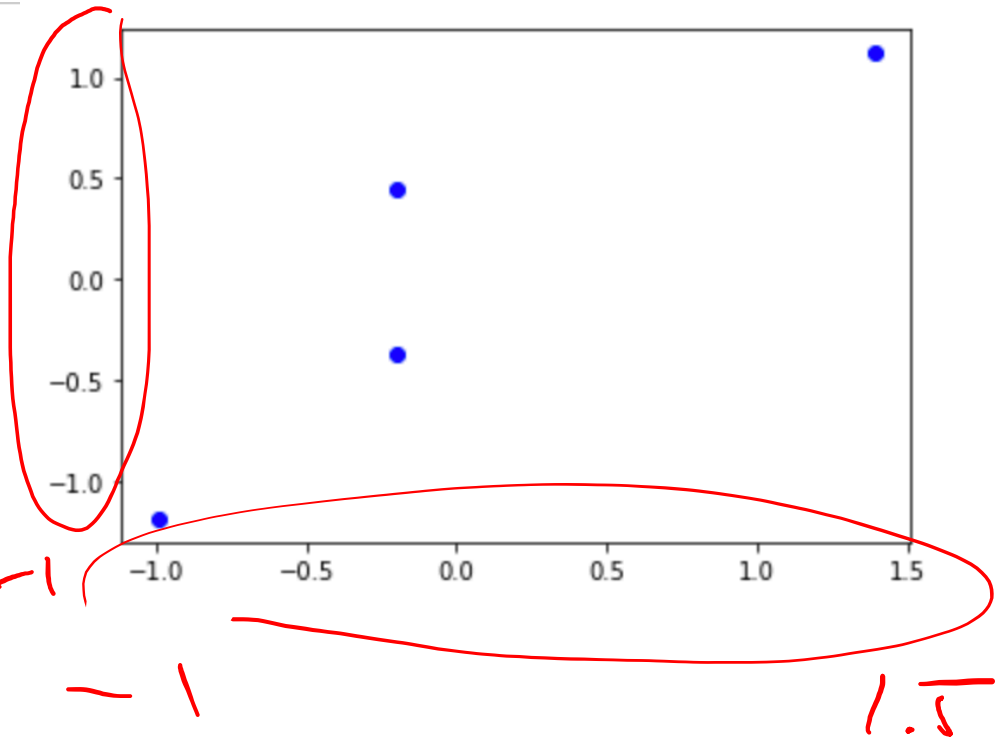


Standardizing Data

Original



Standardized



Note on Standard Deviation

- There are two commonly used, slightly different computations for standard deviation:

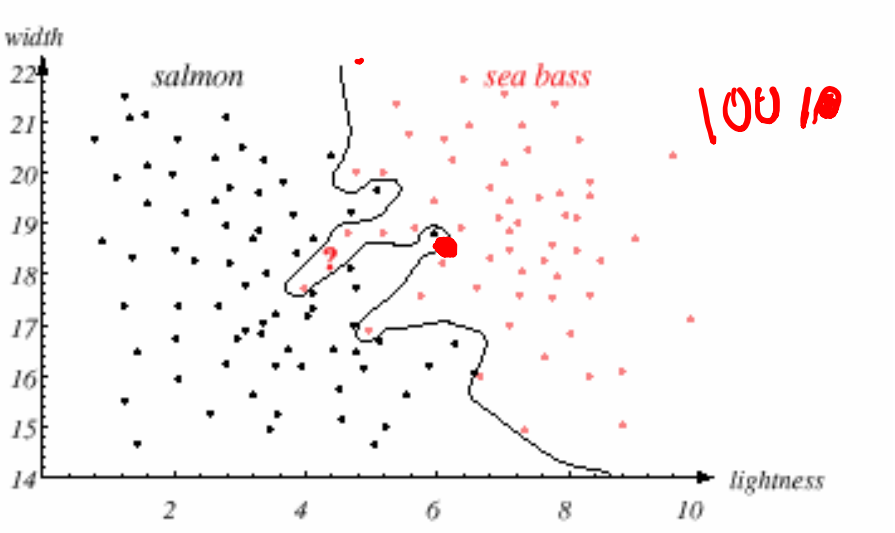
$$\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^2}$$

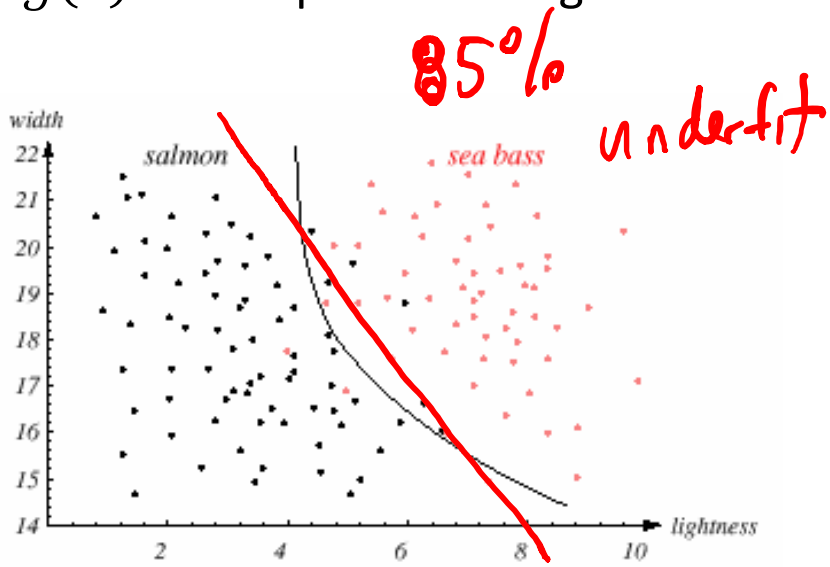
$$\sigma = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(x_i - \mu)^2}$$

- Typically the first version is to be used if we got the mean, μ, somehow other than computing it from the data itself.
- The second version adjusts for this lack of independence of σ and , u if in fact μ was computed as $\mu = \frac{1}{N}\sum_{i=1}^{N} x_i$
- For this course, we will typically want to use the version that divides by $N - 1$

Over/Underfitting

- The biggest problem with supervised learning is **over** or **under-fitting**
- If we over-fit our training data, then we're finding a function for the training data, not the function of the entire system.
  - Therefore may not fit the general data
- If we under-fit our data then our model $g(x)$ is not specific enough to be an approximation to $f(z)$.

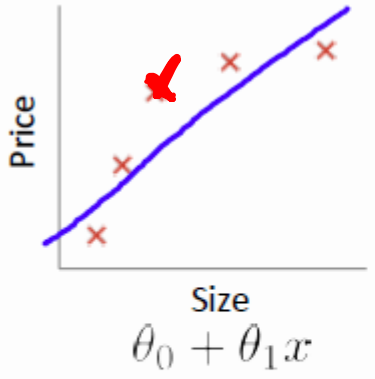




Bias vs Variance

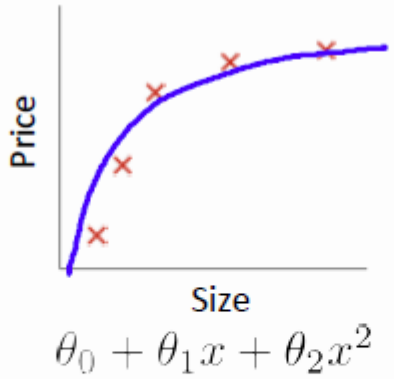
- Sometimes instead of talking about over-fitting and under-fitting we talk about *bias* and *variance*
- If our trained model is very dependent on the training data set (that is it fits it very well), then there will be a large ***variance*** in the models given different training sets.
  - Therefore we say variance and overfitting are related
- Conversely, if our model barely changes at all when it sees different training data, then we say it is **biased**.
  - Which is not necessarily due to underfitting. But it could be…

Over/Under Fitting



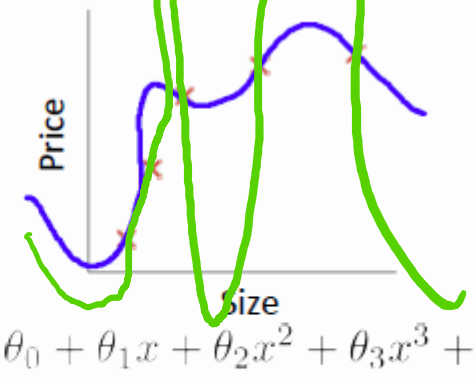
$\theta_0 + \theta_1 x$

High bias
(underfit)



$\theta_0 + \theta_1 x + \theta_2 x^2$

"Just right"



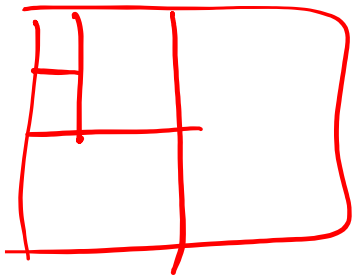
$\theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 + \theta_4 x^4$

High variance
(overfit)

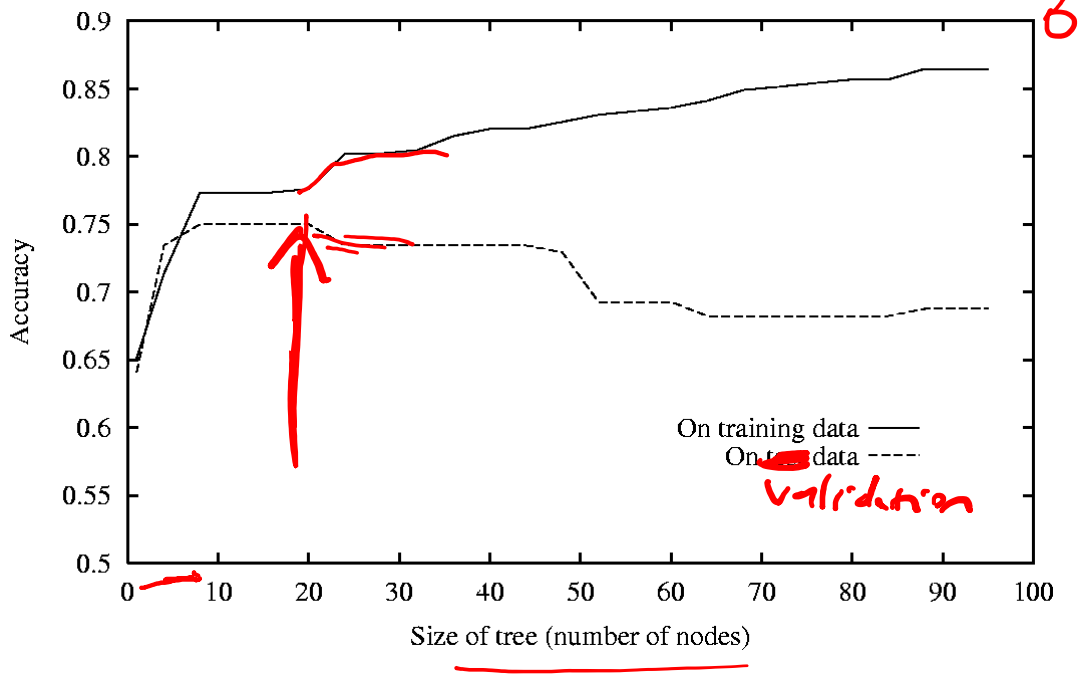
Detecting Over/Under-Fitting

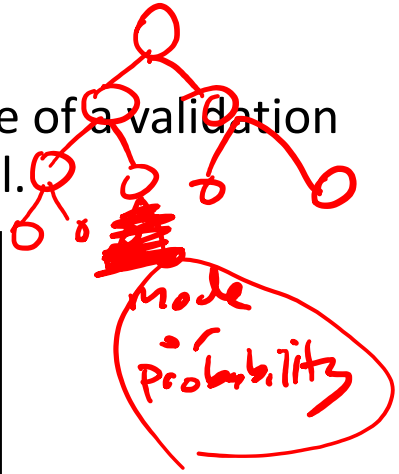
- How can we detect under-fitting?
  - If we don't do well on either the training or the testing sets
- How can we detect over-fitting?
  - If we do well on the training set but poorly on the testing set.

Overfitting



- What's the problem with fitting our data as closely as possible?
  - We may overfit the data!
- Since this is an iterative (or recursive) algorithm, the use of a validation set to decide between different versions is quite natural.

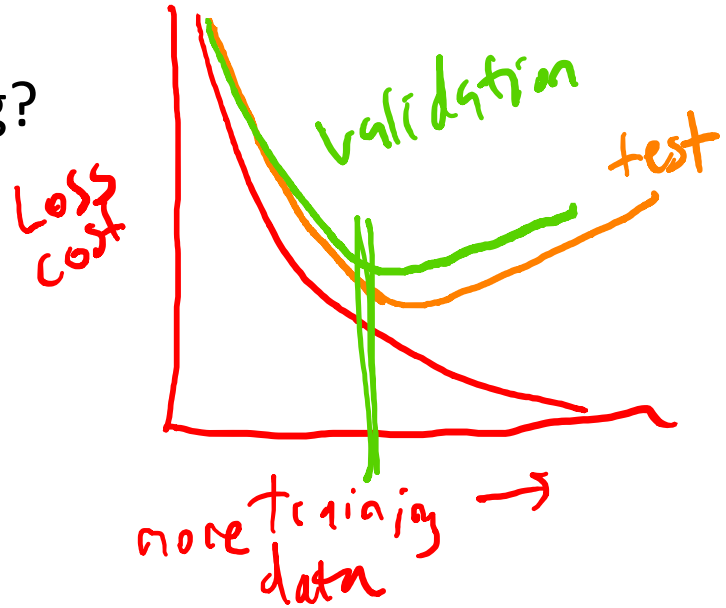




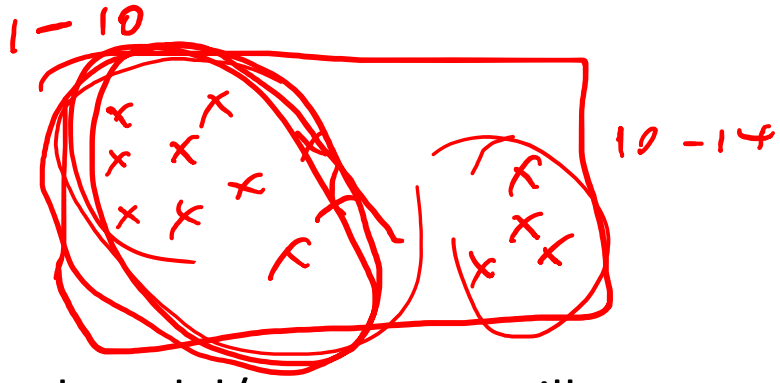
http://jmvidal.cse.sc.edu/talks/decisiontrees/allslides.html

Dealing with Over/Under-Fitting

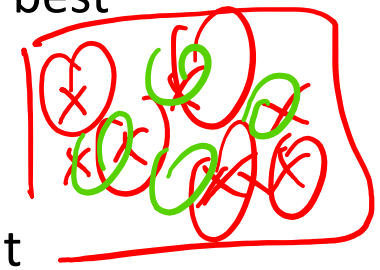
- How can we deal with under-fitting?
  - Make a more complex model
    - May involve need more features
  - Trying a different algorithm
- How can we deal with over-fitting?
  - Use a less complex model
    - May involve using less features
  - Try a different algorithm.
  - Get more data
  - Use a third set to choose between hypothesis (called a *validation set)*.
  - Add a penalization (or regularization) term to the equations to penalize model complexity.



Data Sets



- For constructing our final model/system we will want to use all of our data
- However, often we need to figure out which model is best
- Therefore we will split our data into two groups:
  1. **Training Data**
  2. **Testing Data**



- Typically this is done as a 2/3 training, 1/3 testing split
- We then build/train our system using the training data and test our system using the testing data
- Now we can compare this system against others!
- Once we've chosen our model then we can use all the data and know what the upper-bound on the error should be
- The key is to have the training data and testing data pulled from the same distribution

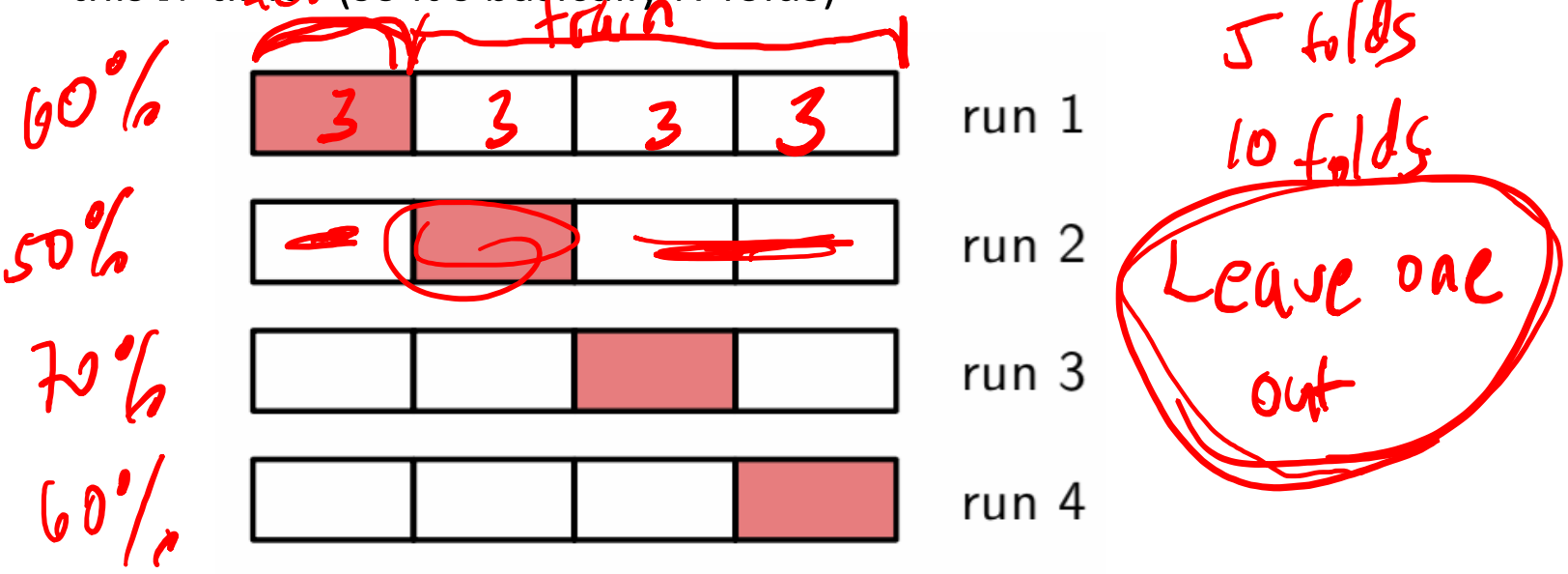
randomly shuffle data

Cross Validation

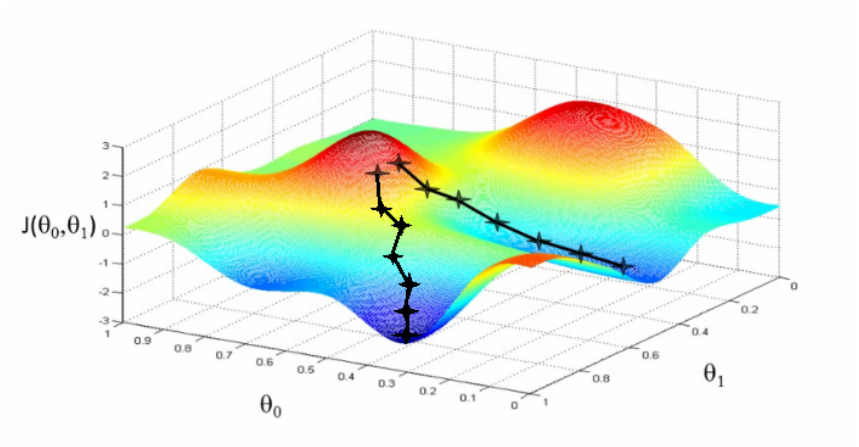
- What if we don't have that much data?
- Then we can do something called *cross-validation*
- Here we do several training(and validation, if necessary)/testing runs
  - Keeping track of all the errors
- We can then compute statistics for our this classifier based on the list of errors.
- Again, in the end our final system will be build using all the data.

S-Folds Cross Validation

- There are a few types of cross-validation
  - S-Folds: Here we'll divide our data up into S parts, train on $S-1$ of them and test the remaining part. Do this S times
  - Leave-one-out: If our data set is really small we may want to built our system on $N-1$ samples and test on just one sample. And do this N times (so it's basically N-folds)



Gradient Ascent/Descent



Gradient Ascent/Descent

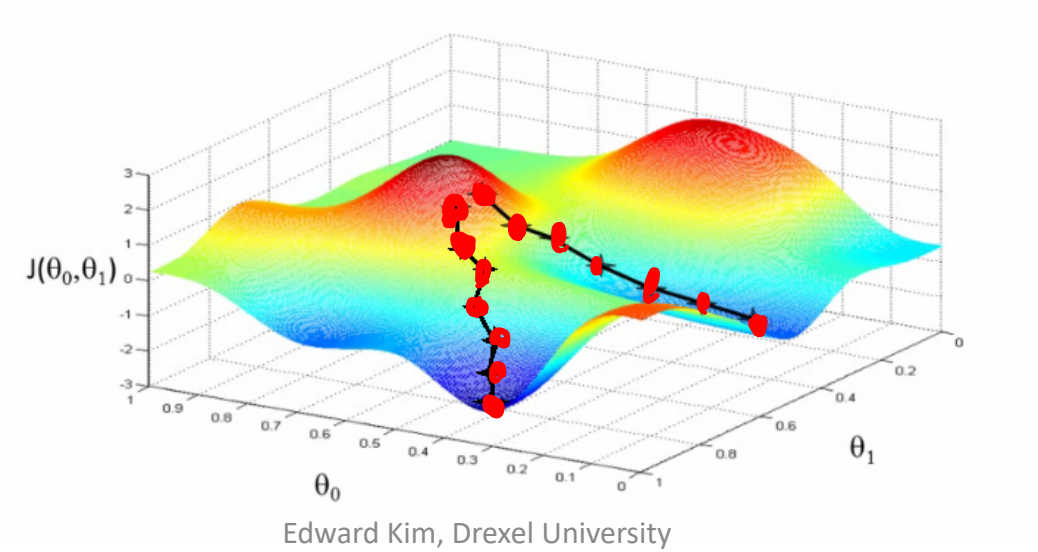
- The solution to linear regression that we just provided, $\theta = (X^T X)^{-1} X^T Y$, is called the *closed-form* solution to the problem.
  - We are able to use mathematics to come up with a direct solution to the minima/maxima problem.
- However, for some problems a closed-form solution/equation may not exist and/or if our matrix is large it may become infeasible to compute inverse
  - Or inverse may not exist in some cases

Gradient Ascent/Descent

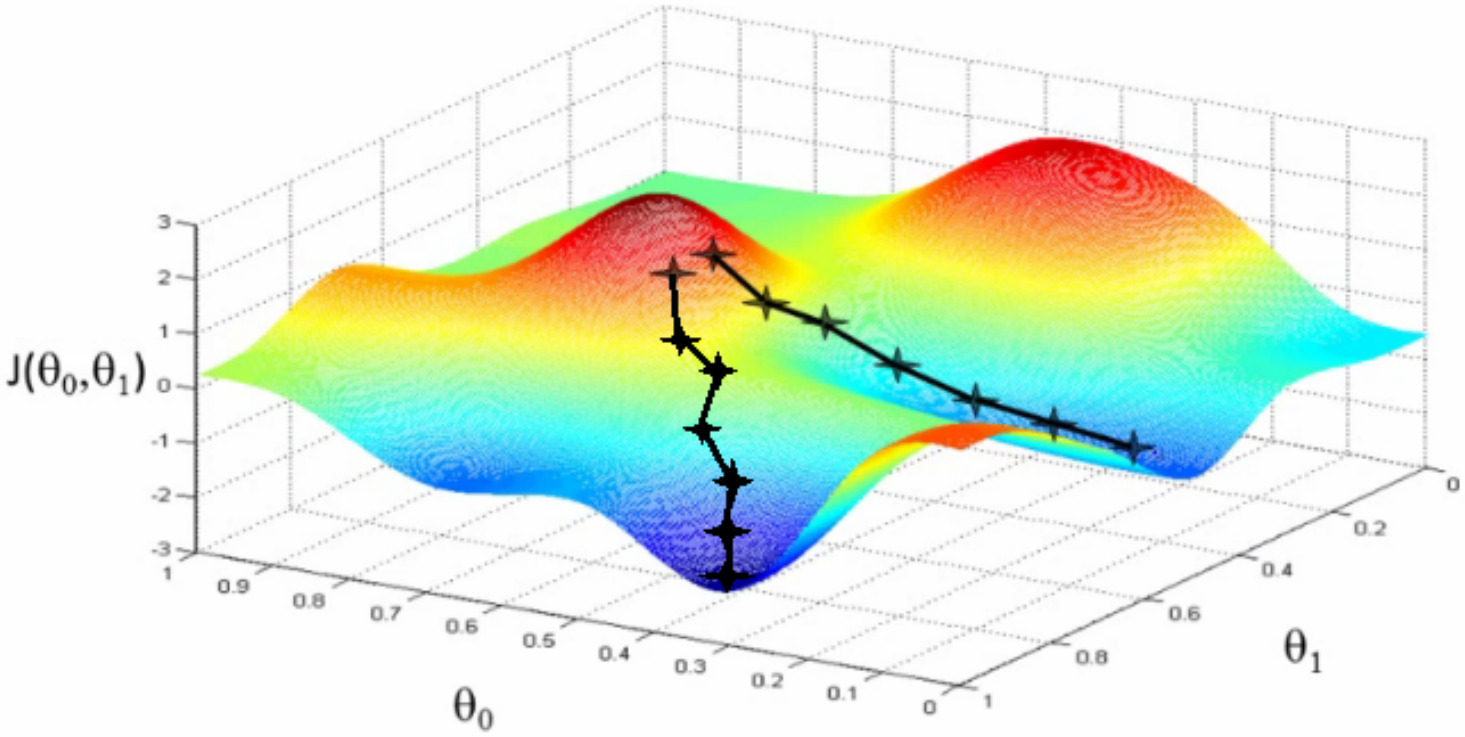
- Another approach to solving a minima/maxima problem is to compute the *gradient* of the equation with regards to each parameter in order to move our current parameter's value in that direction
  - And iteratively do this until we converge to a minima/maxima
- This approach is called ***gradient descent*** and generalizes nicely to lots of applications where we need to find the values of parameters to minimize or maximize some function

Gradient Descent/Assent

- The gradient of J can be thought of the way (vector) to go towards the maxima/minima
  - The gradient is synonymous with the slope.
- If we want to minimize J (if it's an error function), then we want to go "downhill"
  - Opposite direction of the gradient



Gradient Descent/Assent



The graphic depicts gradient assent with two different initial values of (θ_0, θ_1) and updating each parameter simultaneously

Gradient Descent

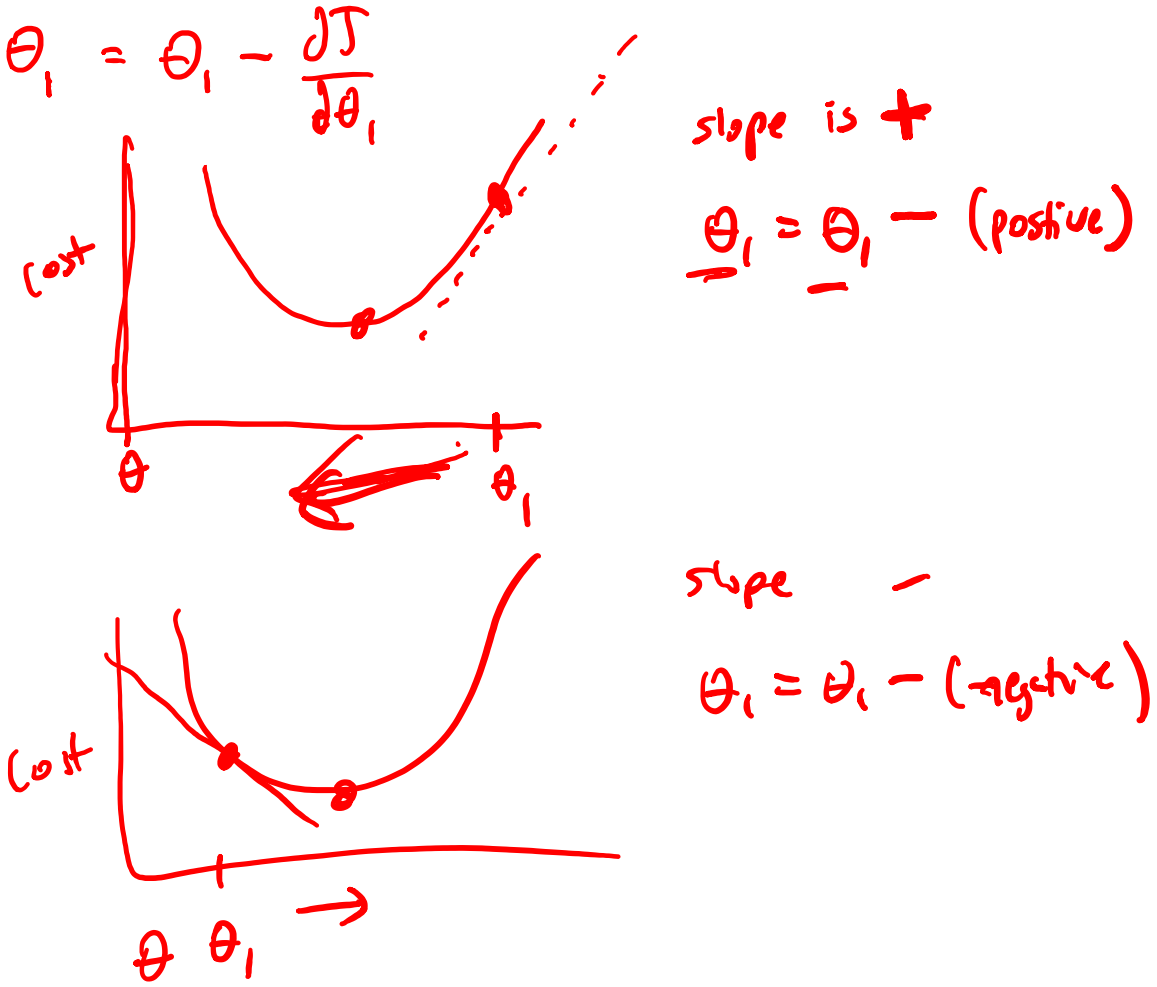
- To find the gradient of J we just take its derivative with respect to the variable we are solving for.
- We want to find the gradient with respect to each of our parameters, $\theta_0, \theta_1, \ldots, \theta_D$
- So the overall gradient, $\frac{\partial J}{\partial \theta}$, can be written as:

$$\frac{\partial J}{\partial \theta} = \begin{bmatrix} \frac{\partial J}{\partial \theta_0} \\ \frac{\partial J}{\partial \theta_1} \\ \cdots \\ \frac{\partial J}{\partial \theta_D} \end{bmatrix}$$

- Then we move our parameters in the direction of this gradient.

$$\theta = \theta - \frac{\partial J}{\partial \theta}$$

Intuition behind gradient update

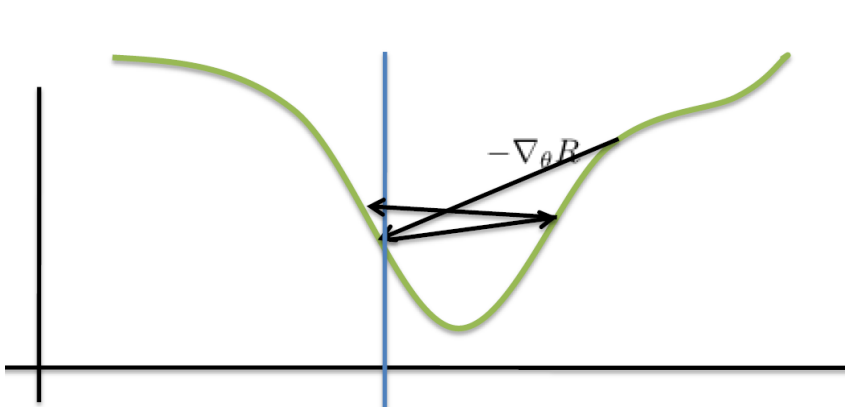


Iterative Gradient Descent Pseudocode

- Given:
  - Standardized data matrix X of size $N \times D$
  - Target value matrix Y of size $N \times 1$
- Add a bias column to X so that $X = [1, X]$ and has size $N \times (D + 1)$
- Initialize parameters θ_j for $j = 1, \ldots, D + 1$ to some random value. Note that θ will be a column vector of size $(D + 1) \times 1$
- Until convergence
  - $\forall x \in X$
    - Compute gradient $\frac{\partial J}{\partial \theta} = 2x^T(x\theta - y)$
    - Update θ such that $\theta = \theta - \eta \frac{\partial J}{\partial \theta}$

The Learning Rate

- If we just move our parameters according to $\frac{\partial J}{\partial \theta}$ we may either:
  - Go too slowly
  - Or *overjump* the desired minima/maxima.
- Therefore, we typically add a *hyperparameter*, η, called the *learning rate* that controls *how much* to go in the direction of the gradient.
- So now $\theta = \theta - \eta \frac{\partial J}{\partial \theta}$



Gradient Descent

- Termination Criteria:
  - Max number of iterations reached
  - Parameters change very little
  - Change in the training error is very little.

Variants of Gradient Descent

- This version of gradient descent, where we update the parameters one observation at a time, is called *iterative* or *online* gradient descent.
- The issue with iterative gradient descent is that it may take online to converge and/or be more likely to converge to some local solution.
- Instead we can update the parameters using the *average* of the gradients created by *all* the observations. This is called *batch gradient descent* and can be written as:

$$\theta = \theta - \frac{\eta}{N} X^T (X\theta - Y)$$

  - *Note: Here we let η absorb the other scalar, 2.*
- Of course batch gradient descent is that it requires all the data to be in memory at once
  - Which can be in issue with big data
- Therefore a variant called *stochastic mini-batch* gradient descent is more common
  - Each (outer) iteration, shuffle all the data
  - Break full data into smaller batches and use each to update parameters.

Least Squares Gradient Descent

- Let's first compute the gradient with respect to a single parameter, θ_i and a single observation, (x, y) and then *vectorize* our solution to update all parameters simultaneously.
- This approach, where we just use one observation at a time is called *iterative,* or *online* gradient learning.
- Returning to our least square linear regression problem, for a single observation the error is:

$$\mathrm{J} = (y - x\theta)^2$$

$\frac{\partial}{\partial \theta_1}$ ~~$\theta_0 + \theta_1 x_1 + \theta_2 x_2 \ldots \theta_n x_n$~~

Derive the update rule

$$\frac{\partial J}{\partial \theta_1} = (y - X\theta)^2 \Rightarrow 2(y - X\theta)(0 - x_1)$$

$$-2x_1(y - X\theta)$$

$$\frac{\partial J}{\partial \theta_1} = 2x_1(X\theta - y)$$

$$\frac{\partial J}{\partial \theta_2} = 2x_2(X\theta - y)$$

$$\frac{\partial J}{\partial \theta_j} = 2x_j(X\theta - y)$$

$$\frac{\partial J}{\partial \theta} = \begin{bmatrix} 2x_0(X\theta - y) \\ 2x_1(X\theta - y) \\ 2x_2(X\theta - y) \\ \vdots \end{bmatrix}$$

Least Squares Gradient Descent

$$J = (y - x\theta)^2$$

- Now let's take the gradient of this with respect to one of the parameters, θ_i

$$\frac{\partial J}{\partial \theta_i} = -2x_i y + 2x_i x\theta = 2x_i(x\theta - y)$$

- We can then vectorize this to get all the gradients at once for this observation:

$$\frac{\partial J}{\partial \theta} = 2x^T(x\theta - y) \Rightarrow 2x^T x\theta - 2x^T y = 0$$

$$2x^T x\theta = 2x^T y$$

$$\theta = (x^T x)^{-1} x^T y$$

- And update your parameters as:

$$\theta = \theta - 2x^T(x\theta - y)$$