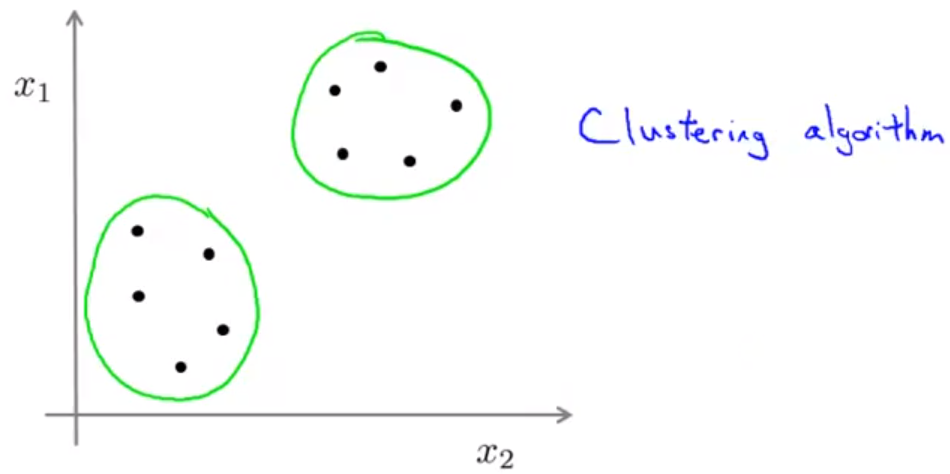
**Machine Learning – Lecture Notes – Part III**

**Week 8**

**Unsupervised Learning**

**Introduction**

* In a supervised learning problem, we are given a set of labels and we must fit a hypothesis to it.
* In an unsupervised learning problem, our training data has no labels. We simply ask unsupervised learning algorithms to find some structure in the data for us.
* We can use a clustering algorithm to find clusters in data as follows.



* Some applications of clustering include market segmentation, social network analysis, organizing computing clusters, astronomical data analysis, etc.

**K–Means Algorithm**

* Given an unlabelled data set, we first create **cluster centroids** randomly in the data set, where is the number of clusters we want to group the data into. Then, there are 2 main steps in the algorithm.

1. First, we colour each of the data points based on which of the centroids it’s closest to.
2. Then, we move each of the centroids to the average of its coloured data points and we repeat the process.

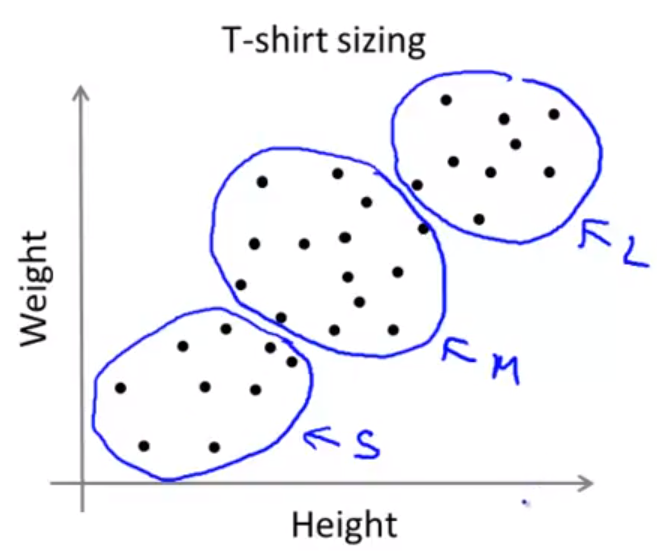
* We iterate through these steps until the cluster centroids are at the centroids of their clusters.

K–Means Formalized

* Input
  + (number of clusters)
  + Training set , and we say (drop the convention)
* Randomly initialize cluster centroids .
* Repeat {
  + for
    - index (from 1 to ) of cluster centroid closest to using .
  + for to
    - average (mean) of points assigned to cluster
  + }
* If a certain cluster has no points closer to it, then you can either (a) eliminate that cluster so there are clusters or (b) reinitialize that cluster centroid at a new location.

K–Means for Non–Separated Clusters

* Even when it looks like there are no clusters in the data, we can make clusters out of it.
* For example, given people’s heights and weights, we can segment the market for T-shirts into 3 different classes of data – small, medium and large, and design T-shirts based on those market segmentations.



**Optimization Objective**

K-Means Optimization Objective

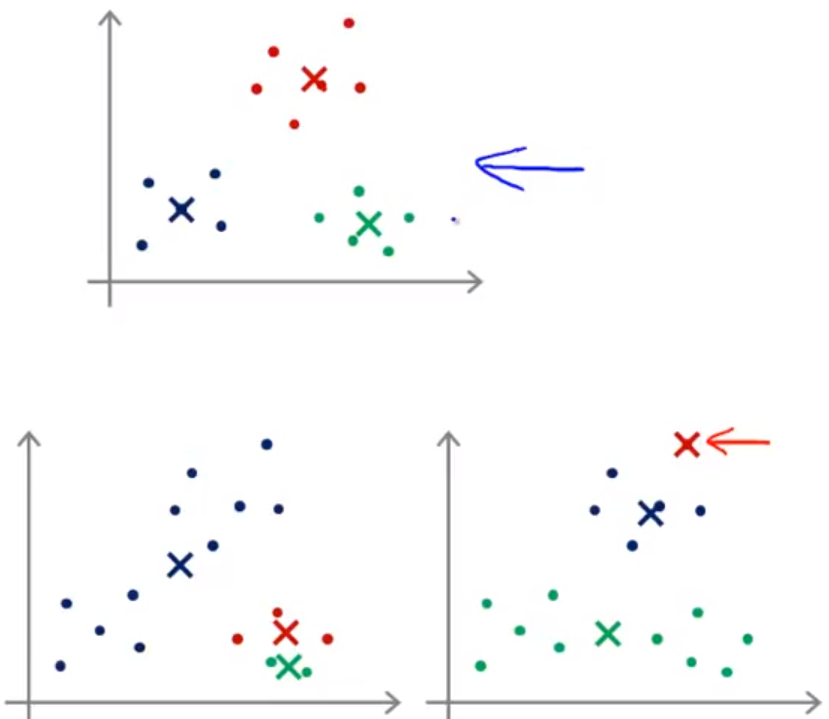
* Notation:
  + index of cluster () to which example is currently assigned
  + cluster centroid , where
  + cluster centroid of cluster to which example has been assigned
* Optimization objective:
* The K-means algorithm first minimizes with respect to and then with respect to .

**Random Initialization**

* Should have .
* Randomly pick training examples , where is the random training example.
* Set for these examples.

Local Optima

* However, it is very possible that K-means gets stuck at a bad local optima that prevents it from reaching the global optimum.

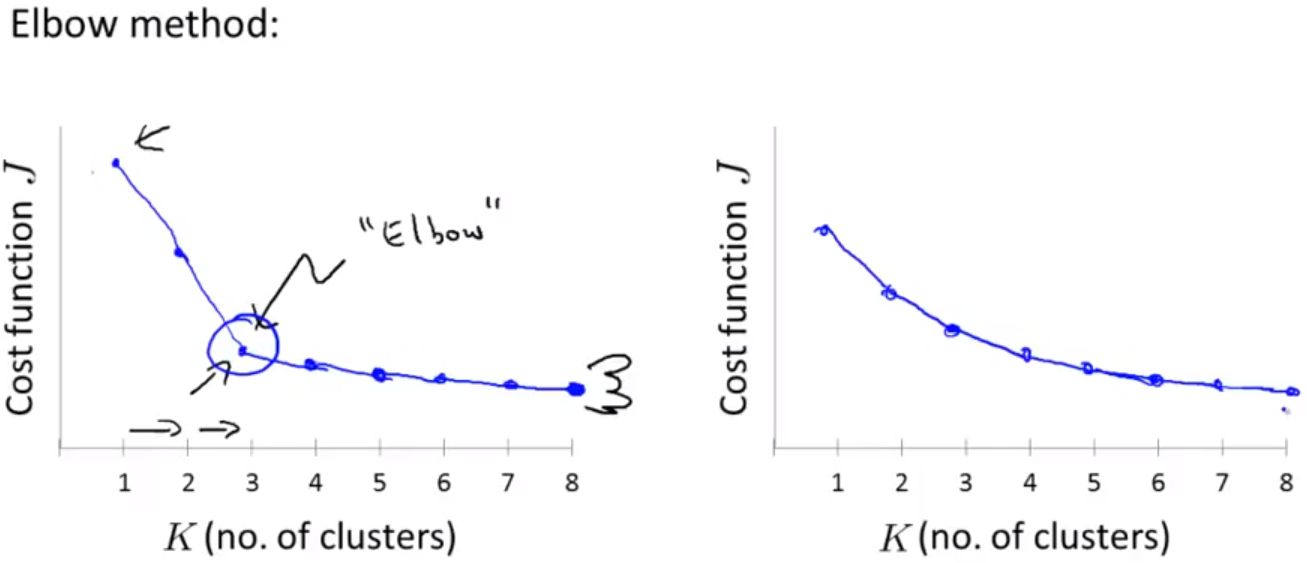


* To prevent this issue or get higher chances of remedying it, we can run random initialization many times and compare the cost function at each iteration and keep the best one.
* for to {
  + randomly initialize K-means
  + run K-means and get
  + compute cost function
  + compare to the previous cost function and keep the parameters that yield lowest cost
  + }

**Choosing the Number of Clusters**

Elbow Method

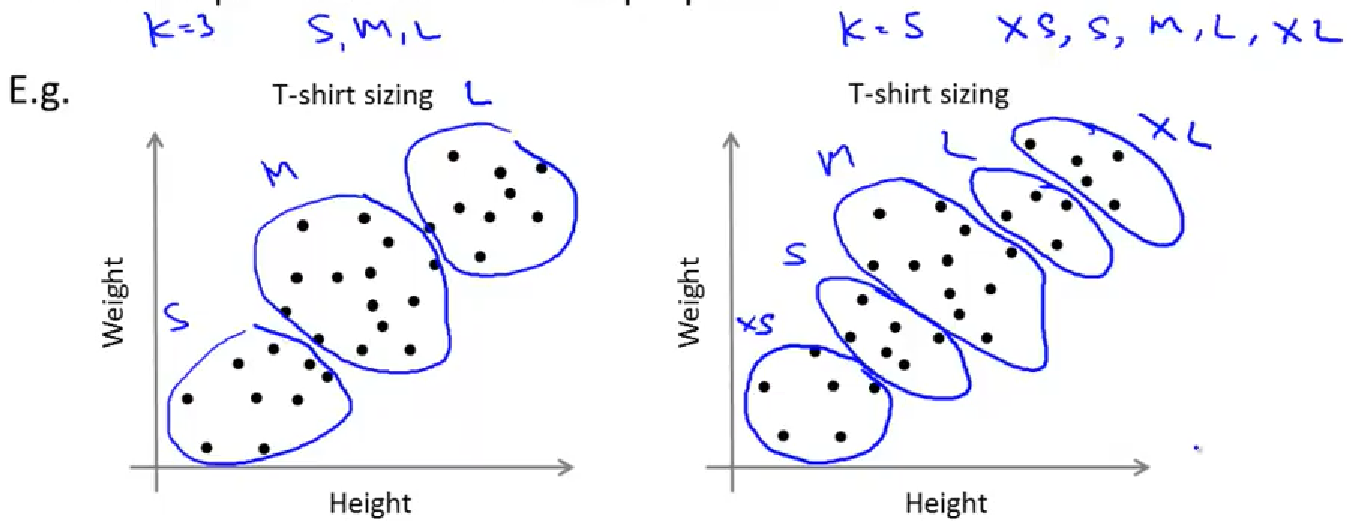
* We run the K-means algorithm with varying number of clusters and plot the minimum cost .
* Ideally, there is an “elbow” at which point, from the left, the cost function is decreasing rapidly, and from the right the cost function is decreasing slowly.



* However, we often end up with a curve on the right, where the elbow of the curve is ambiguous.

Choosing the Value of K

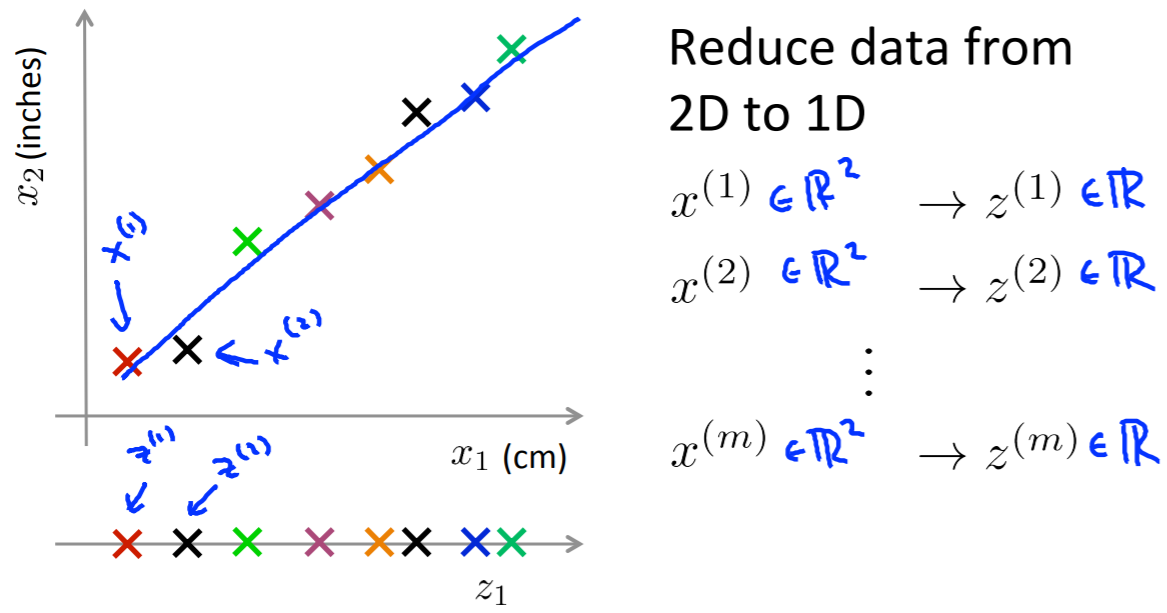
* We can train a current dataset based on how many clusters we will look for eventually. For example, in the market segmentation for T-shirts, we group the heights and weights of individuals into 3 categories to make a S, M, and L size for people.
* If I want more categories, namely XS, S, M, L, XL, then I pick .



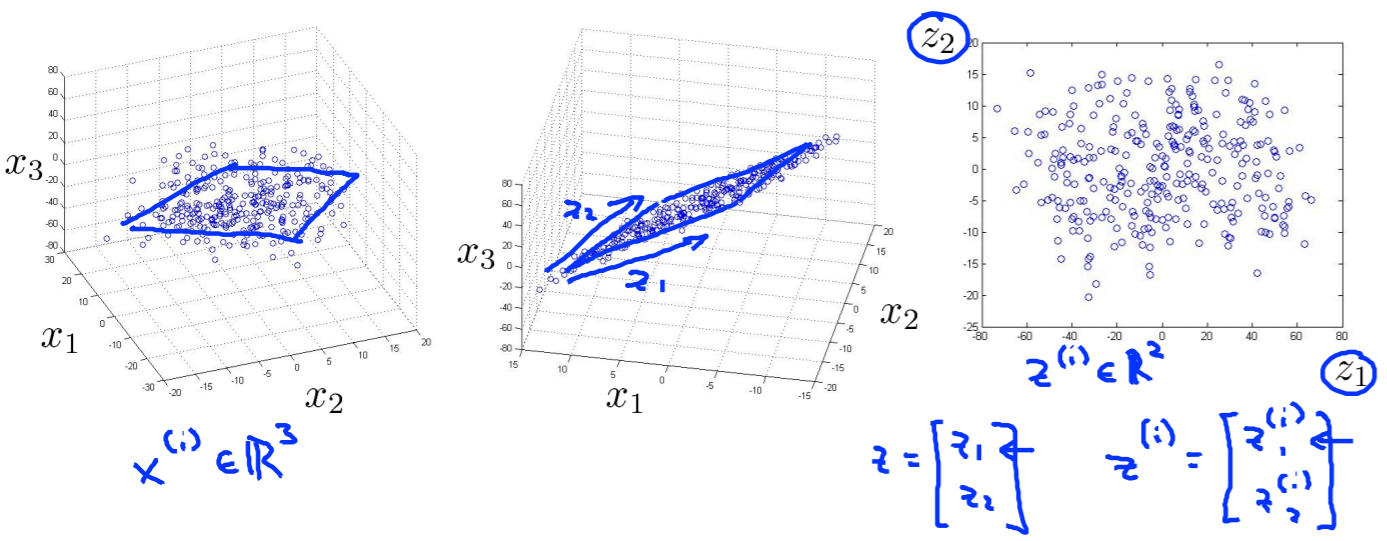
**Motivation I: Data Compression**

Dimensionality Reduction

* Reducing data from a higher dimension to a lower dimension (e.g. 2D 🡪 1D).
* If most data points sit on a straight line, then we can approximate both features by just using the *x-*value of the data points.



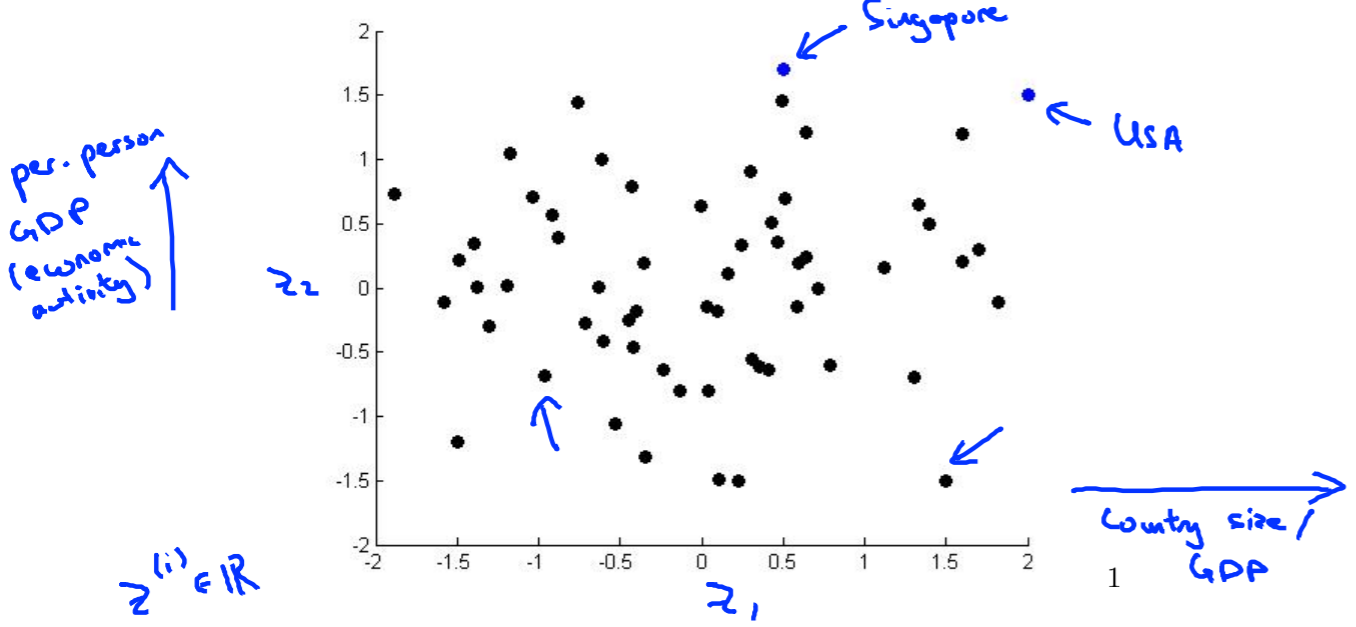
* Similarly, we can map 3D 🡪 2D if all/most of the data points lie on a single plane. We can make two axes to represent the distance along each of the axes of the plane.



* Reducing the dimension data both saves storage space because less numbers are needed for the same data and makes learning algorithms more efficient since they require less computation.

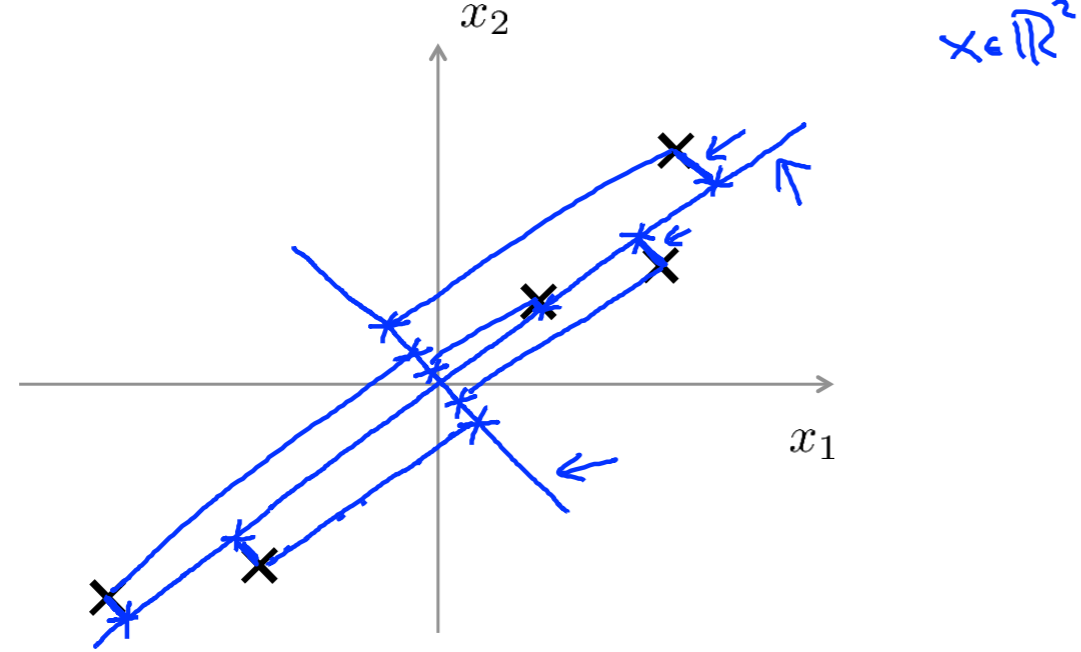
Motivation II: Data Visualization

* When given data of high dimensions, such as 50-dimensional data, it is hard to visualize it.
* Using dimensionality reduction will help group certain features together to get lower dimensional data (i.e. 2D or 3D data) that can be plotted. This gives a better understanding of the data.
* For example, given many statistics about different countries like GDP, GDP per capita, HDI, life expectancy, Gini coefficient, mean household income, etc., we can reduce these figures into 2 categories: country size & per-person GDP (economic activity), yielding a graph like below.

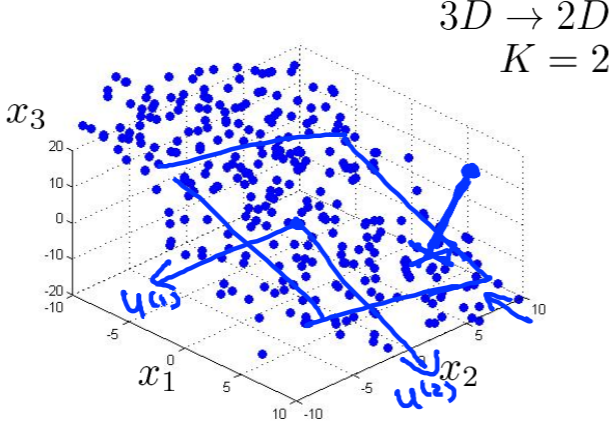


**Principal Component Analysis (PCA) Problem Formulation**

* The PCA problem is to reduce higher dimensional data into lower dimensional data.
* For example, with training examples , we can simplify this data into a line if there is a straight line that adheres to the data.

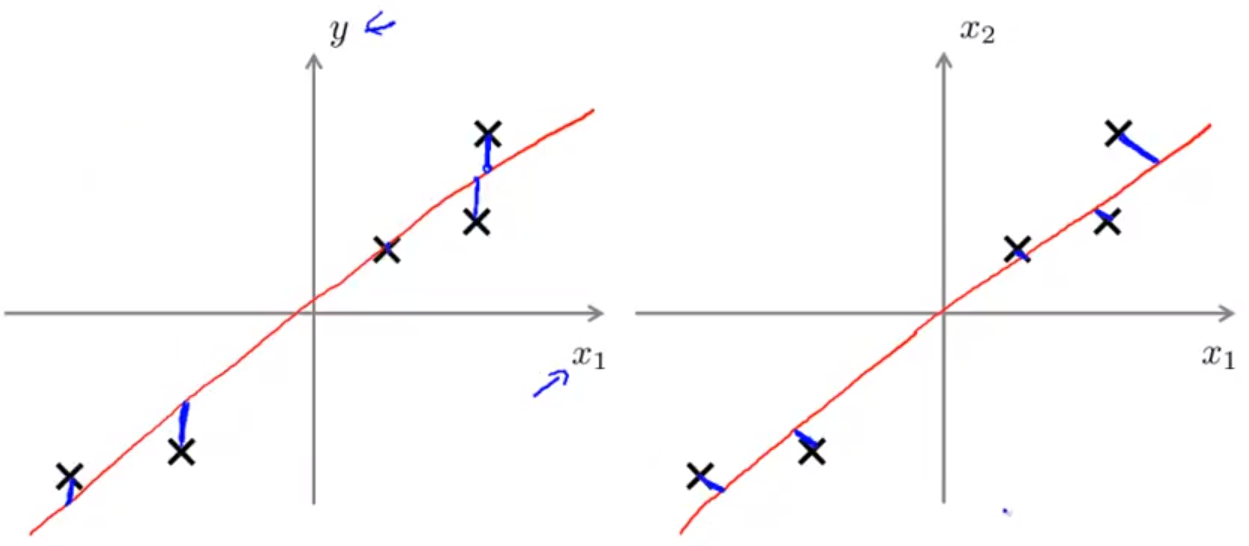


* Reduce from 2D 🡪 1D: Find a direction vector onto which to project the data as to minimize projection error.
* Reduce from -dimensional 🡪 -dimensional: Find vectors onto which to project the data as to minimize projection error.



PCA is **not** Linear Regression

* Although PCA looks like linear regression, their goals are quite different.
* Linear regression tries to predict a variable based on an input and calculates the error as the sum of squares of the **vertical distance** from each point to the line.
* PCA tries to find a line to represent the relationship between variables and and calculates the error as the sum of squares of the **orthogonal distance** from each point to the line.



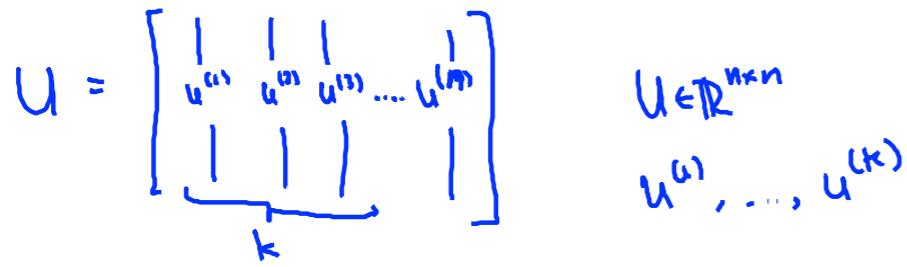
**Principal Component Analysis (PCA) Algorithm**

Data Preprocessing

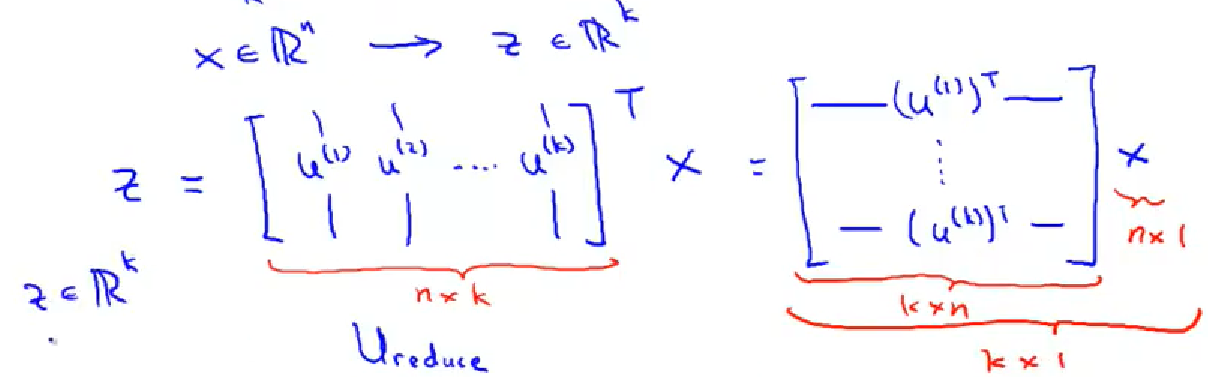
* Given a training set , we perform feature scaling & mean normalization:
  + Replace each with .
  + If different features are on different scales (e.g. size of house (ft2), # of bedrooms), scale features to be on similar ranges (e.g. -1 to 1, or 0 to 1).

Principle Component Analysis (PCA) Algorithm

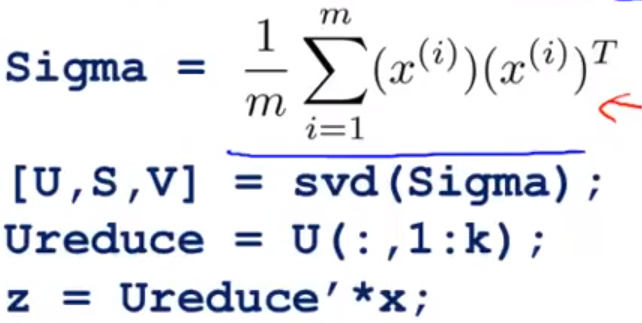
* Goal is to reduce data from -dimensions to -dimensions.
* Compute the “covariance matrix”:
* (The matrix will be , since is and is .)
* Now compute “eigenvectors” of matrix by using [U, S, V] = svd(Sigma);, where svd stands for singular value decomposition.
* Now we have the matrix and we can take the first columns to get our direction vectors upon which to map the data.



* Then, to compute the vector (our reduced input vector), we calculate this via , where is the first columns of the matrix.

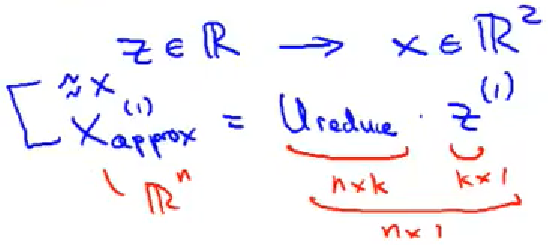


* The vectorized implementation is as follows.



**Reconstruction from Compressed Representation**

* Once we have found , there should be some way to go back to find . Since we have , we can calculate an approximation .



* What you find is that the approximate data fits the line perfectly because all the errors have disappeared. Thus, it is most effective to implement the PCA algorithm when the sum of squares of **orthogonal distances** is small.

**Choosing the Number of Principal Components**

* The PCA algorithm tries to minimize the average squared projection error.
* We also have something called the total variation in the data.
* Typically, we choose , the number of principal components, to be the smallest value so that:
* The above means that “99% of the variance is retained”.

Efficient Algorithm

* There is a better way of computing . After calling the SVD function on , we get [U, S, V] = svd(Sigma), and the matrix S is a diagonal matrix so that only the elements on the diagonal are non-zero.
* Then, for given , to assure that we have variance, we can test if
* We can do this by creating a for loop and checking at each value of from 1 to , if this value is .

**Advice for Applying PCA**

* Given a training set , if each feature vector , then it will be hard to run a supervised learning algorithm on it.
* Then, we apply PCA to get reduced feature vectors that retain 99% of variance, for example.
* Then, we have a new training set . Then, train our hypothesis using our prediction function .

Applications of PCA

* Compression
  + Reduce memory/disk space needed to store data
  + Speed up learning algorithms
* Visualization (usually choose or )
* An **incorrect use** of PCA is to prevent overfitting. Use instead of to reduce the number of features to .
* The problem is that this algorithm doesn’t know the output labels , and so it could be throwing away useful information. While you can guarantee that “99% of the variance will be retained”, regularization is usually better because the algorithm knows the training outputs .

**Week 9**

**Anomaly Detection**

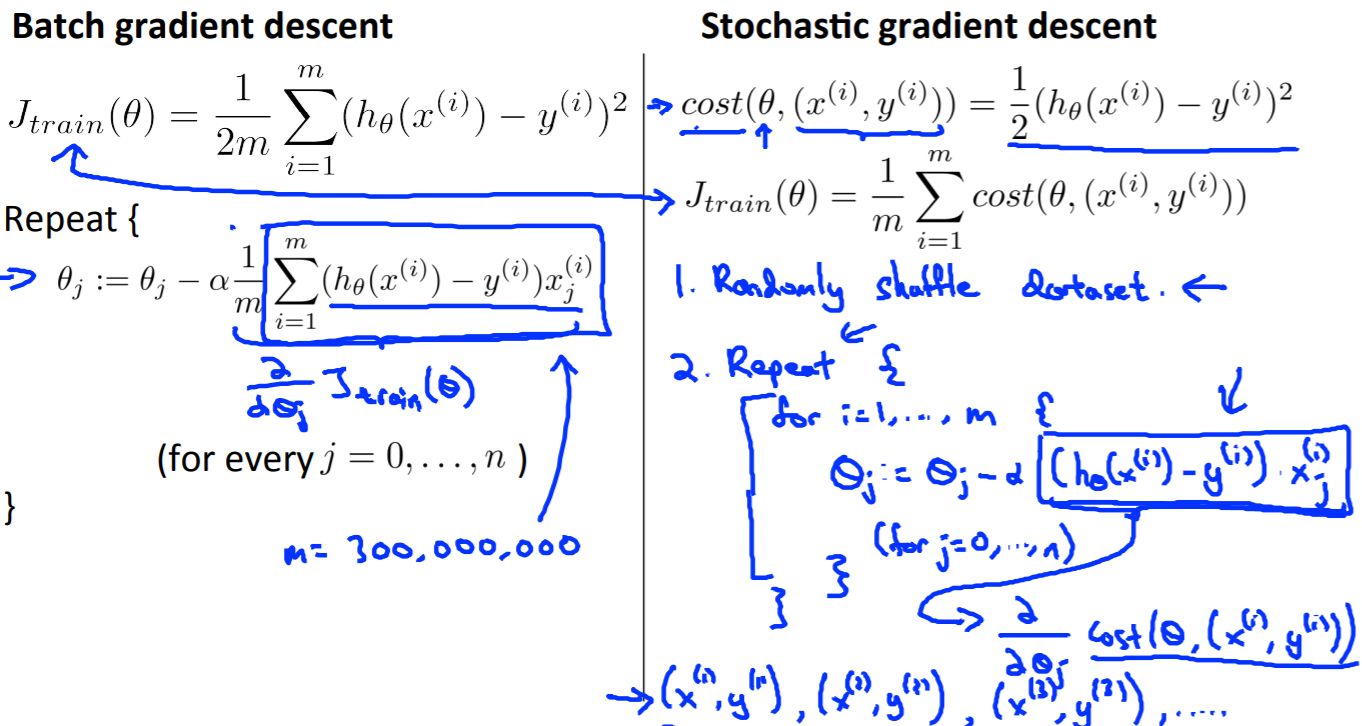
**Density Estimation – Motivation**

**Week 10**

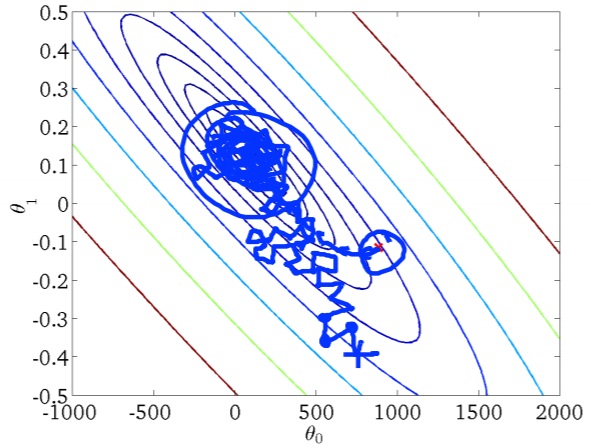
**Large Scale Machine Learning**

**Stochastic Gradient Descent**

* Below are the outlined differences between **batch gradient descent** and **stochastic gradient descent**.



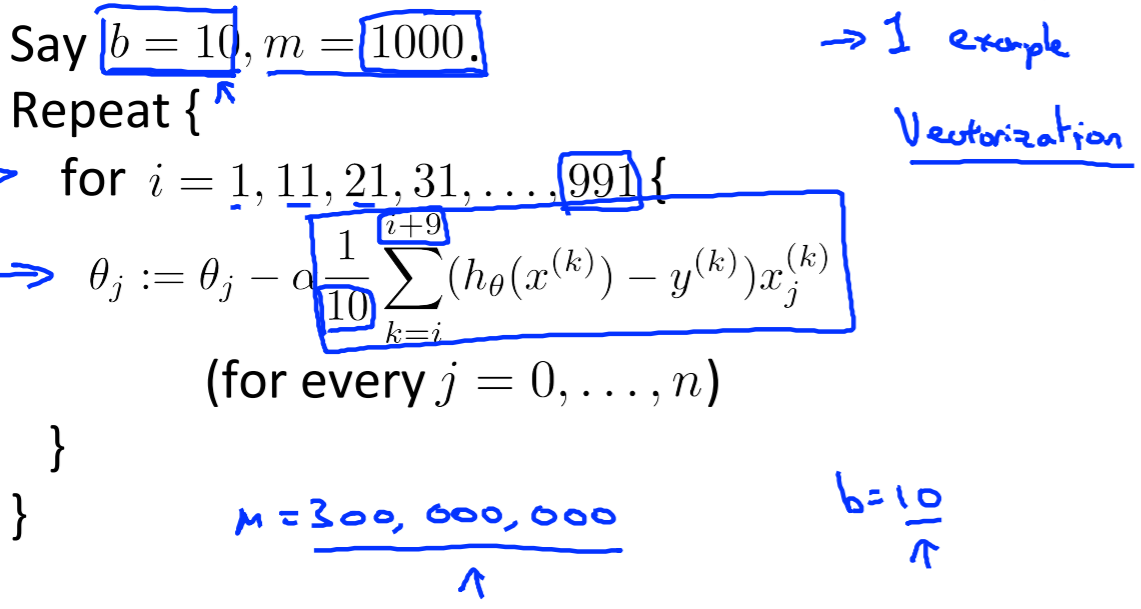
* Basically, on each iteration, the algorithm modifies the parameters a little bit after looking at only 1 training example.
* The dataset needs to be randomly shuffled to yield better chances of converging to the global optimum rather than a local optimum.
* The stochastic gradient descent algorithm takes a much more random path and converges close to the global minimum but doesn’t stay there.



* However, this is not a problem since it gets close enough to the global optimum, so the cost is relatively low.

**Mini-Batch Gradient Descent**

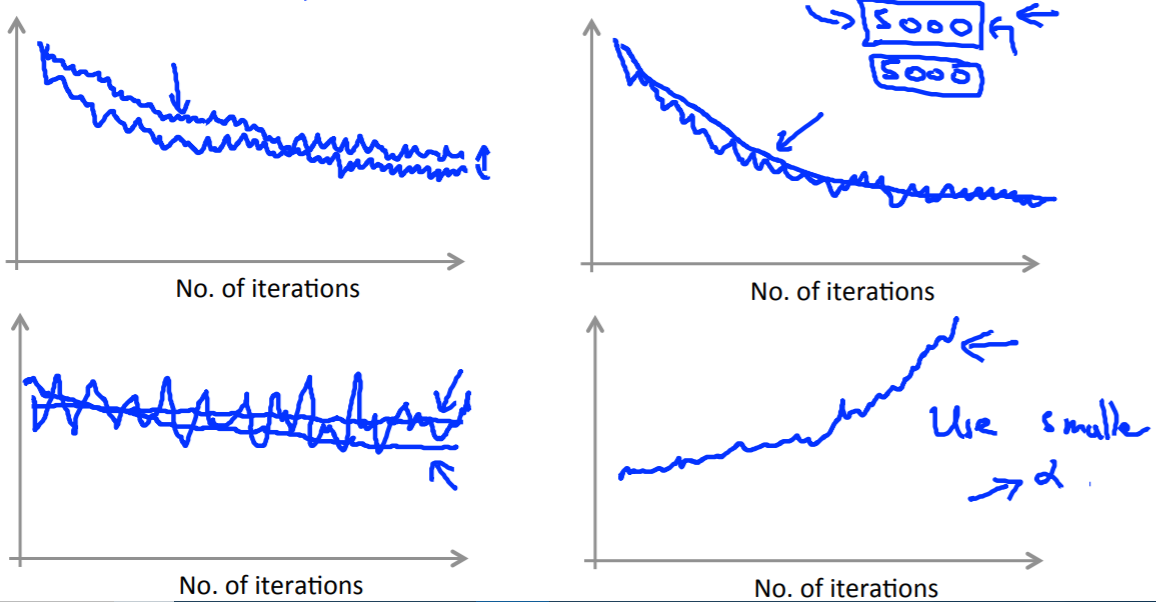
* Here are different classes of gradient descent:
  + **Batch Gradient Descent:** Use all examples in each iteration
  + **Stochastic Gradient Descent:** Use example in each iteration
  + **Mini-Batch Gradient Descent:** Use examples in each iteration
* In each “step” of mini-batch gradient descent, we train on examples in every step until we get through all examples, so there are usually steps.



* We need a good **vectorized** implementation to outperform the **stochastic gradient descent** because it allows you to parallelize the computation.

**Stochastic Gradient Descent Convergence**

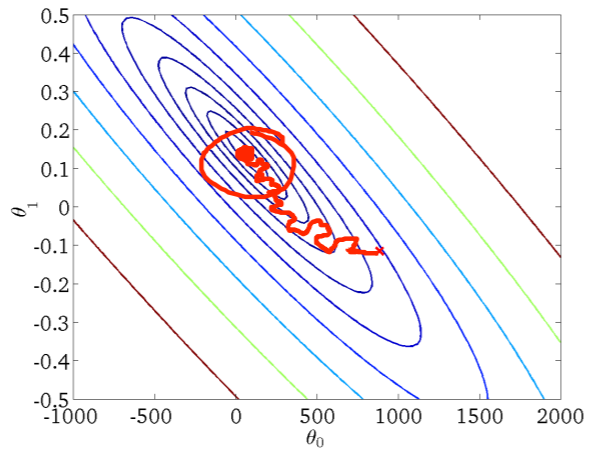
* In stochastic gradient descent, we compute the cost of the new training example before updating parameters .
* Then every 1000 iterations, for example, we plot averaged over the last 1000 examples processed by the algorithm.
* We can see 4 different types of graphs that occur when using stochastic/mini-batch gradient descent and plotting the cost function over many iterations.



* + **Top-Left:** If the original cost function looks like the top-left curve, you could try using a *smaller learning rate* to decrease cost, but this is already a good solution.
  + **Top-Right:** If the original cost function looks like the top-right curve, you might average over each 1000 examples and get that plot. You could choose to average over *more examples* and get a smoother curve as shown.
  + **Bottom-Left:** If the original cost function looks constant, you could try *average over more examples* to get a smoother curve that shows the decreasing cost function. The first line might have been very noisy. (Cost could also not be decreasing, which could be a problem, so use a *smaller learning rate*.)
  + **Bottom-Right:** If your original cost function is increasing, you should *use a smaller learning rate* and check your learning algorithm isn’t wrong.

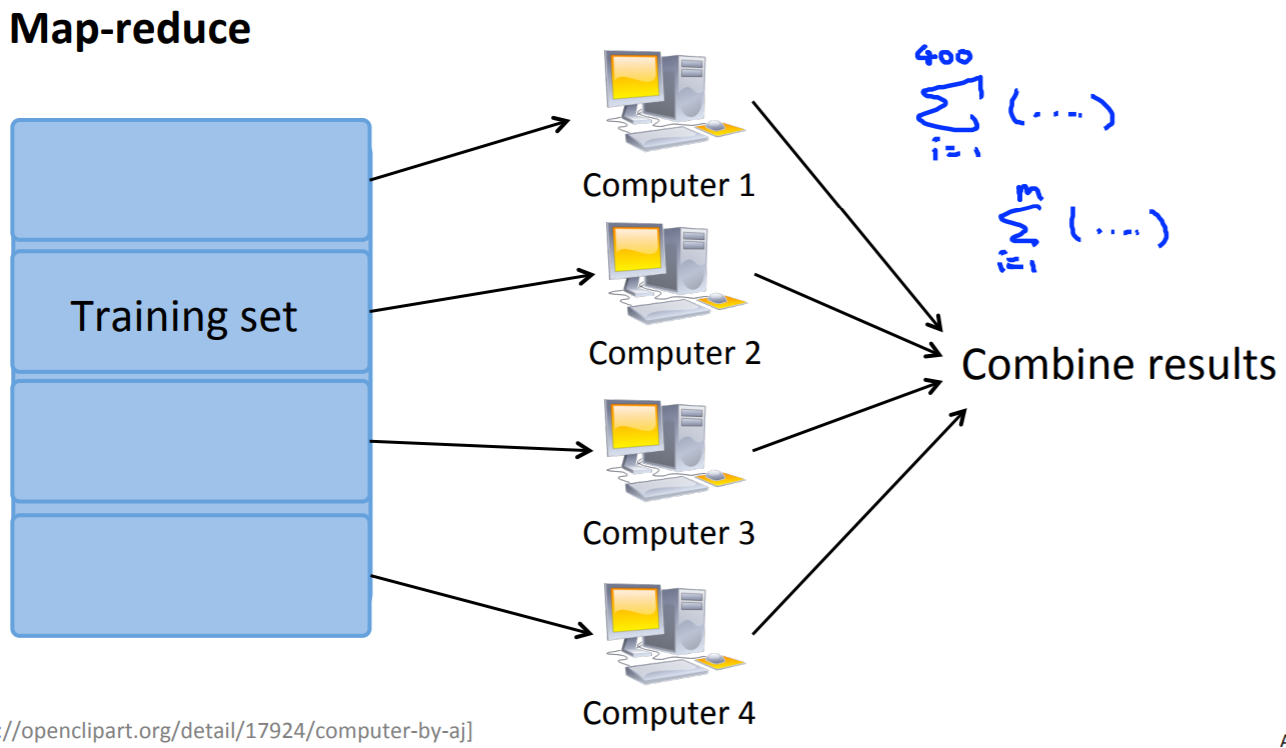
Learning Rate

* If you want your stochastic gradient descent to run faster and converge to the minimum, you could decrease your learning rate as you increase number of iterations. You could set , where is the iteration number and are constants.
* This way, the meanderings of gradient descent will become smaller as iterations increase, so it will look like the following curve.



**Map, Reduce and Data Parallelism**

* The idea is that we can divide up the workload of a machine learning algorithms on different machines. They can compute different portions of the training set in parallel and then put them back together.



* Even in a single computer, you can parallelize tasks by splitting a training set between multiple cores to train the machine learning algorithm faster.