

13: Clustering

[Previous](#) [Next](#) [Index](#)

Please follow me on LinkedIn for more
Steve Nouri
<https://www.linkedin.com/in/stevenouri/>

Unsupervised learning - introduction

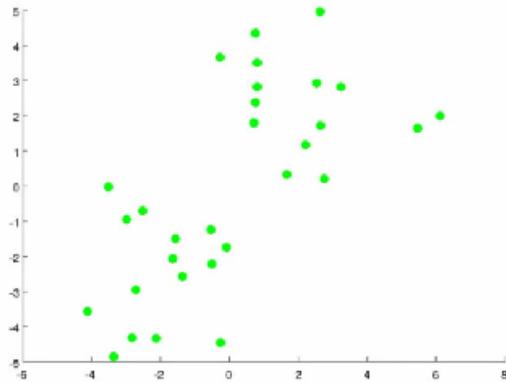
- Talk about **clustering**
 - **Learning from unlabeled data**
- Unsupervised learning
 - Useful to contrast with supervised learning
- Compare and contrast
 - Supervised learning
 - Given a set of labels, fit a hypothesis to it
 - Unsupervised learning
 - Try and determine structure in the data
 - Clustering algorithm groups data together based on data features
- What is clustering good for
 - **Market segmentation** - group customers into different market segments
 - **Social network analysis** - Facebook "smartlists"
 - **Organizing computer clusters** and data centers for network layout and location
 - **Astronomical data analysis** - Understanding galaxy formation

K-means algorithm

- Want an algorithm to automatically group the data into coherent clusters
- K-means is **by far** the most widely used clustering algorithm

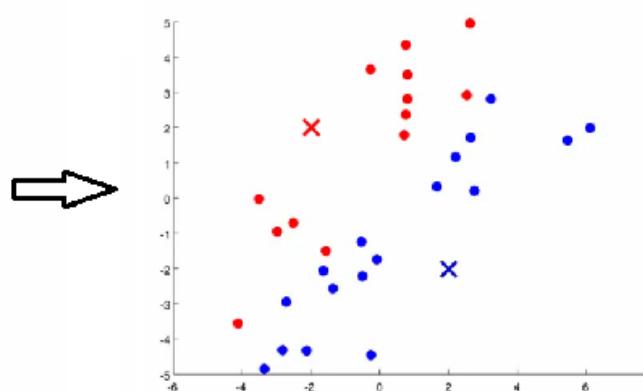
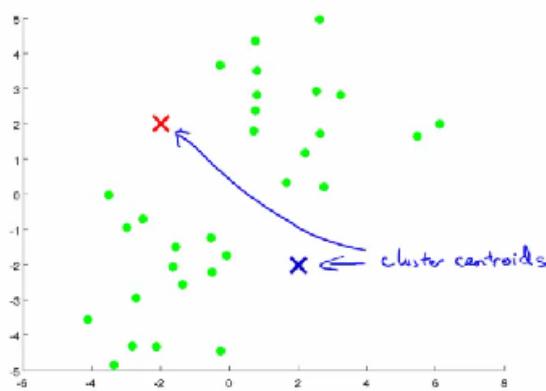
Overview

- Take unlabeled data and group into two clusters



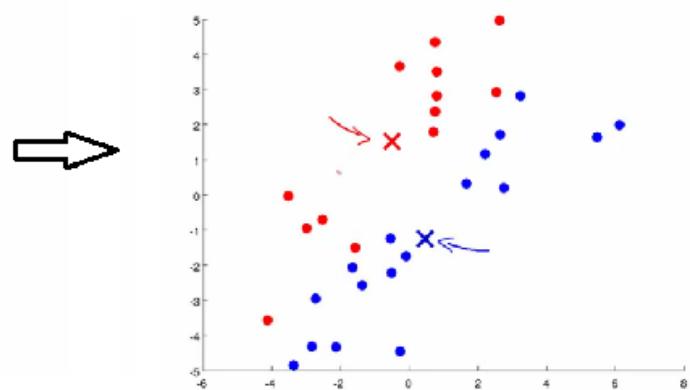
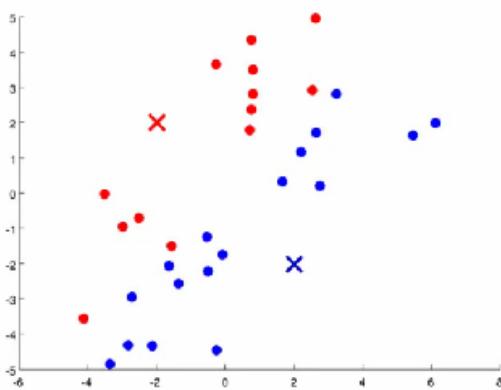
- Algorithm overview
 - 1) Randomly allocate two points as the **cluster centroids**
 - Have as many cluster centroids as clusters you want to do (K cluster centroids, in fact)
 - In our example we just have two clusters
 - 2) Cluster assignment step
 - Go through each example and depending on if it's closer to the red or blue centroid assign each point to one of the two clusters

- To demonstrate this, we've gone through the data and "colour" each point red or blue



- 3) Move centroid step

- Take each centroid and move to the average of the correspondingly assigned data-points



- Repeat 2) and 3) until convergence

- More formal definition

- Input:**

- K (number of clusters in the data)
- Training set $\{x^1, x^2, x^3 \dots, x^n\}$

- Algorithm:**

- Randomly initialize K cluster centroids as $\{\mu_1, \mu_2, \mu_3 \dots \mu_K\}$

Repeat {

for $i = 1$ to m

$c^{(i)} :=$ index (from 1 to K) of cluster centroid
closest to $x^{(i)}$

for $k = 1$ to K

$\mu_k :=$ average (mean) of points assigned to cluster k }

- Loop 1

- This inner loop repeatedly sets the $c^{(i)}$ variable to be the index of the closest variable of cluster centroid closest to x^i
- i.e. take i^{th} example, measure squared distance to each cluster centroid, assign $c^{(i)}$ to the cluster closest

$$\min_{c^{(i)}} \|x^{(i)} - \mu_k\|^2$$

- Loop 2

- Loops over each centroid calculate the average mean based on all the points associated with each centroid from $c^{(i)}$

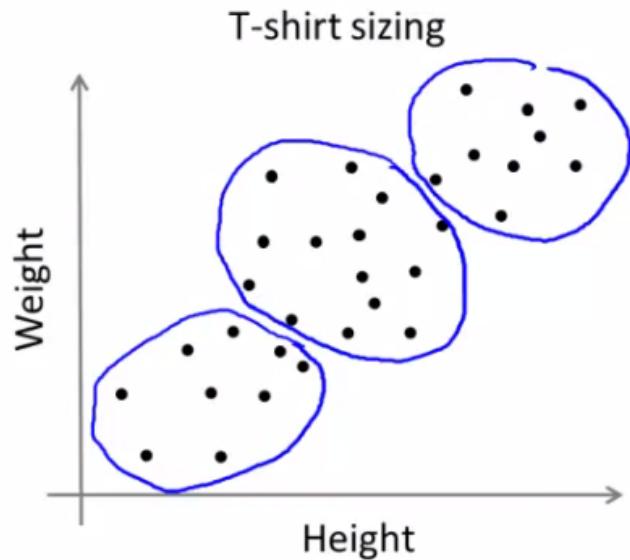
- What if there's a centroid with no data
 - Remove that centroid, so end up with K-1 classes
 - Or, randomly reinitialize it
 - Not sure when though...

K-means for non-separated clusters

- So far looking at K-means where we have well defined clusters
- But often K-means is applied to datasets where there aren't well defined clusters
 - e.g. T-shirt sizing



- Not obvious discrete groups
- Say you want to have three sizes (S,M,L) how big do you make these?
 - One way would be to run K-means on this data
 - May do the following



- So creates three clusters, even though they aren't really there
- Look at first population of people
 - Try and design a small T-shirt which fits the 1st population
 - And so on for the other two
- This is an example of market segmentation
 - Build products which suit the needs of your subpopulations

K means optimization objective

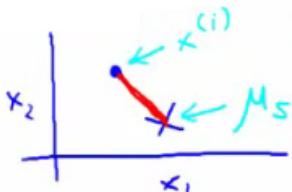
- Supervised learning algorithms have an optimization objective (cost function)
 - K-means does too
- K-means has an optimization objective like the supervised learning functions we've seen

- Why is this good?
- Knowing this is useful because it helps for debugging
- Helps find better clusters
- While K-means is running we keep track of two sets of variables
 - c^i is the index of clusters $\{1, 2, \dots, K\}$ to which x^i is currently assigned
 - i.e. there are $m c^i$ values, as each example has a c^i value, and that value is one of the clusters (i.e. can only be one of K different values)
 - μ_k , is the cluster associated with centroid k
 - Locations of cluster centroid k
 - So there are K
 - So these are the centroids which exist in the training data space
 - μ_{c^i} , is the cluster centroid of the cluster to which example x^i has been assigned to
 - This is more for convenience than anything else
 - You could look up that example i is indexed to cluster j (using the c vector), where j is between 1 and K
 - Then look up the value associated with cluster j in the μ vector (i.e. what are the features associated with μ_j)
 - But instead, for easy description, we have this variable which gets exactly the same value
 - Lets say x^i has been assigned to cluster 5
 - Means that
 - $c^i = 5$
 - $\mu_{c^i} = \mu_5$

- Using this notation we can write the optimization objective;

$$J(c^{(1)}, \dots, c^{(m)}, \mu_1, \dots, \mu_K) = \frac{1}{m} \sum_{i=1}^m \|x^{(i)} - \mu_{c^{(i)}}\|^2$$

- i.e. squared distances between training example x^i and the cluster centroid to which x^i has been assigned to
 - This is just what we've been doing, as the visual description below shows;



- The red line here shows the distances between the example x^i and the cluster to which that example has been assigned
 - Means that when the example is very close to the cluster, this value is small
 - When the cluster is very far away from the example, the value is large

- This is sometimes called the **distortion** (or **distortion cost function**)

- So we are finding the values which minimizes this function;

$$\min_{\substack{c^{(1)}, \dots, c^{(m)}, \\ \mu_1, \dots, \mu_K}} J(c^{(1)}, \dots, c^{(m)}, \mu_1, \dots, \mu_K)$$

- If we consider the k-means algorithm

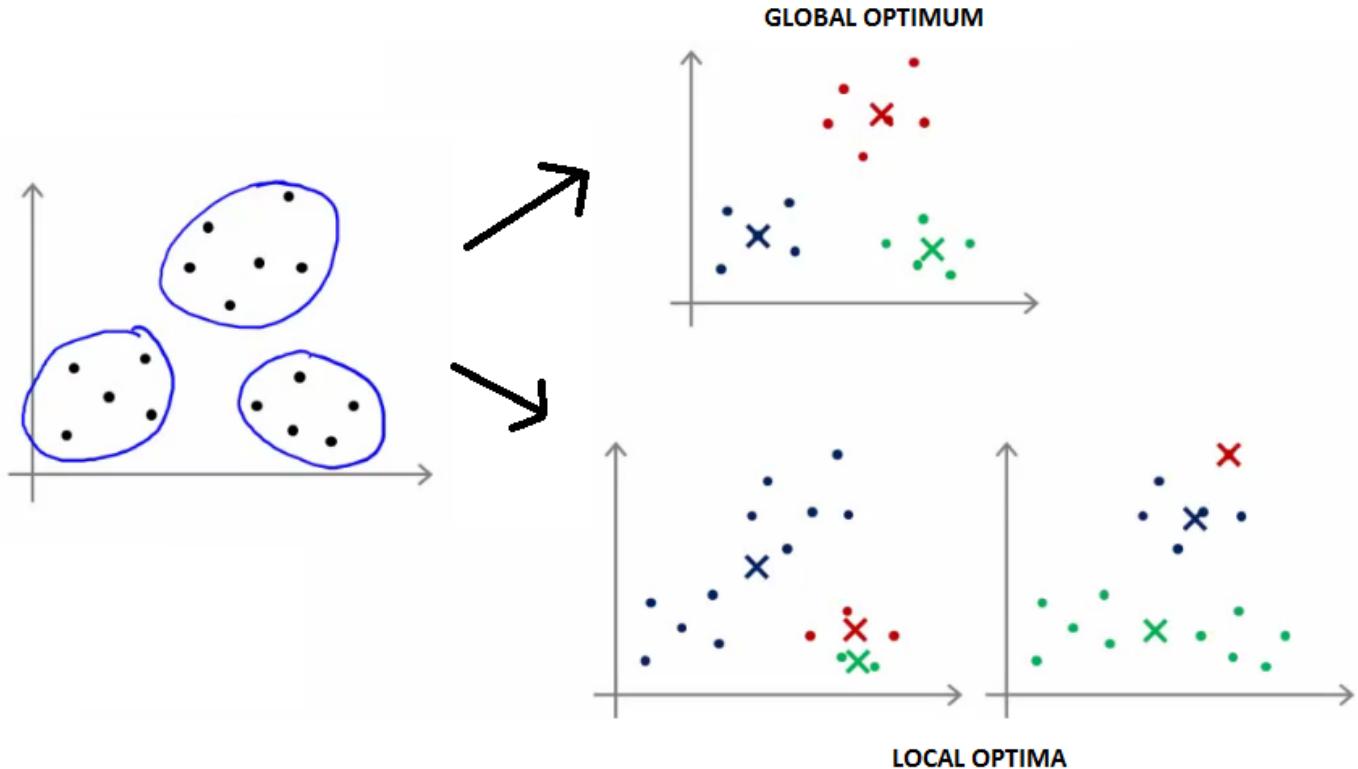
- The **cluster assigned step** is minimizing $J(\dots)$ with respect to $c^1, c^2 \dots c^i$
 - i.e. find the centroid closest to each example
 - Doesn't change the centroids themselves
- The **move centroid step**
 - We can show this step is choosing the values of μ which minimizes $J(\dots)$ with respect to μ
- So, we're partitioning the algorithm into two parts
 - First part minimizes the c variables
 - Second part minimizes the J variables

- We can use this knowledge to help debug our K-means algorithm

Random initialization

- How we initialize K-means
 - And how avoid local optimum

- Consider clustering algorithm
 - Never spoke about how we initialize the centroids
 - A few ways - one method is most recommended
- Have number of centroids set to less than number of examples ($K < m$) (if $K > m$ we have a problem)
 - Randomly pick K training examples
 - Set μ_1 up to μ_K to these example's values
- K means can converge to different solutions depending on the initialization setup
 - Risk of local optimum



- The local optimum are valid convergence, but local optimum not global ones
- If this is a concern
 - We can do multiple random initializations
 - See if we get the same result - many same results are likely to indicate a global optimum
- Algorithmically we can do this as follows;

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)

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

- A typical number of times to initialize K-means is 50-1000
- Randomly initialize K-means
 - For each 100 random initialization run K-means
 - Then compute the distortion on the set of cluster assignments and centroids at convergent
 - End with 100 ways of cluster the data
 - Pick the clustering which gave the lowest distortion
- If you're running K means with 2-10 clusters can help find better global optimum
 - If K is larger than 10, then multiple random initializations are less likely to be necessary
 - First solution is probably good enough (better granularity of clustering)

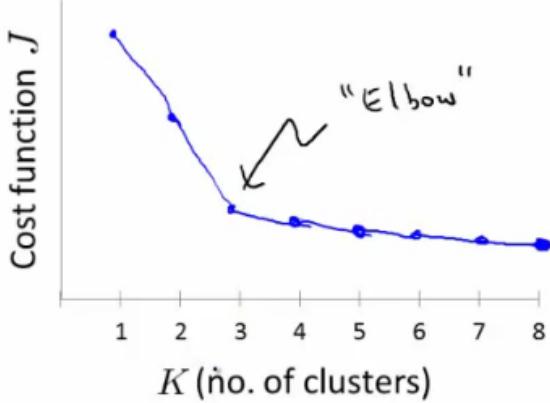
How do we choose the number of clusters?

- Choosing K?
 - Not a great way to do this automatically
 - Normally use visualizations to do it manually
- What are the intuitions regarding the data?
- Why is this hard

- Sometimes very ambiguous
 - e.g. two clusters or four clusters
 - Not necessarily a correct answer
- This is why doing it automatic this is hard

Elbow method

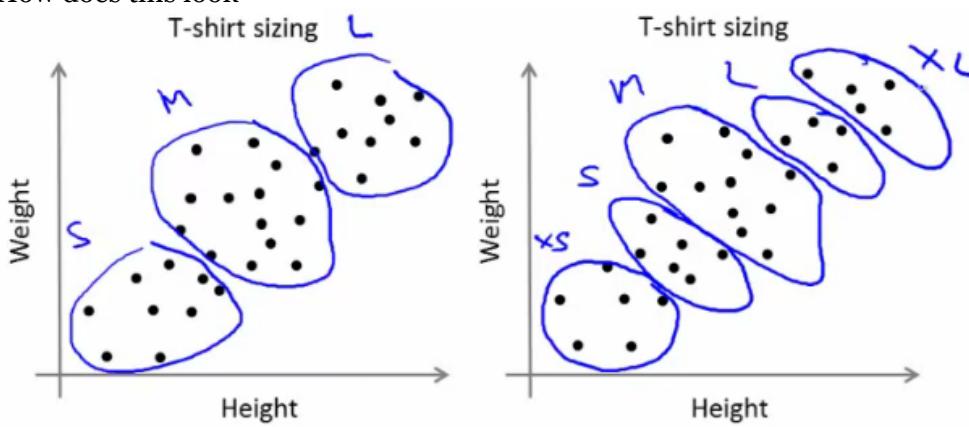
- Vary K and compute cost function at a range of K values
- As K increases $J(\dots)$ minimum value should decrease (i.e. you decrease the granularity so centroids can better optimize)
 - Plot this (K vs $J()$)
- Look for the "elbow" on the graph



- Choose the "elbow" number of clusters
- If you get a nice plot this is a reasonable way of choosing K
- Risks
 - Normally you don't get a nice line -> no clear elbow on curve
 - Not really that helpful

Another method for choosing K

- Using K-means for market segmentation
- Running K-means for a later/downstream purpose
 - See how well different number of clusters serve your later needs
- e.g.
 - T-shirt size example
 - If you have three sizes (S,M,L)
 - Or five sizes (XS, S, M, L, XL)
 - Run K-means where $K = 3$ and $K = 5$
 - How does this look



- This gives a way to choose the number of clusters
 - Could consider the cost of making extra sizes vs. how well distributed the products are
 - How important are those sizes though? (e.g. more sizes might make the customers happier)
 - So applied problem may help guide the number of clusters

14: Dimensionality Reduction (PCA)

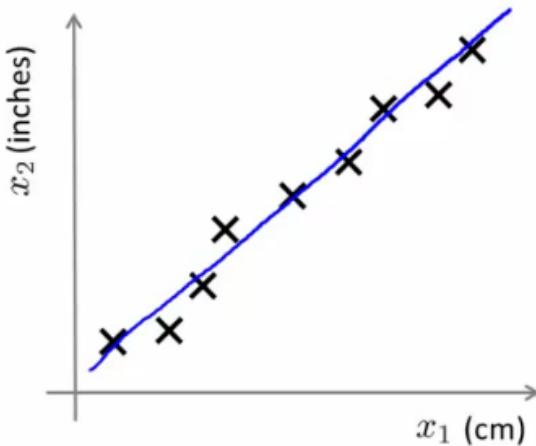
[Previous](#) [Next](#) [Index](#)

Motivation 1: Data compression

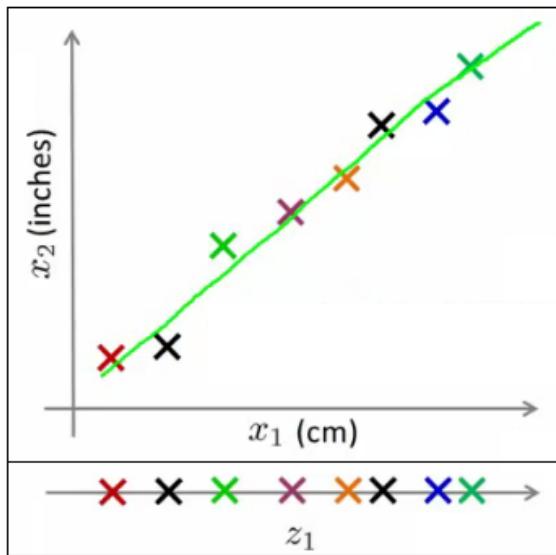
- Start talking about a second type of unsupervised learning problem - **dimensionality reduction**
 - Why should we look at dimensionality reduction?

Compression

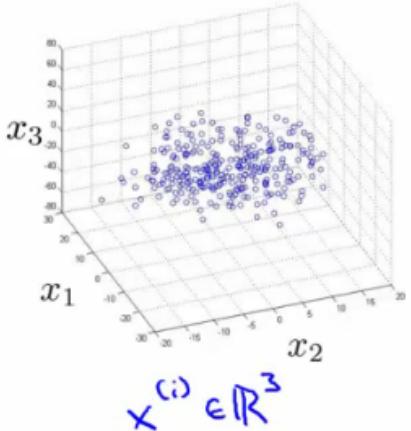
- Speeds up algorithms
- Reduces space used by data for them
- What is dimensionality reduction?
 - So you've collected many features - maybe more than you need
 - Can you "simply" your data set in a rational and useful way?
 - Example
 - Redundant data set - different units for same attribute
 - Reduce data to 1D (2D->1D)



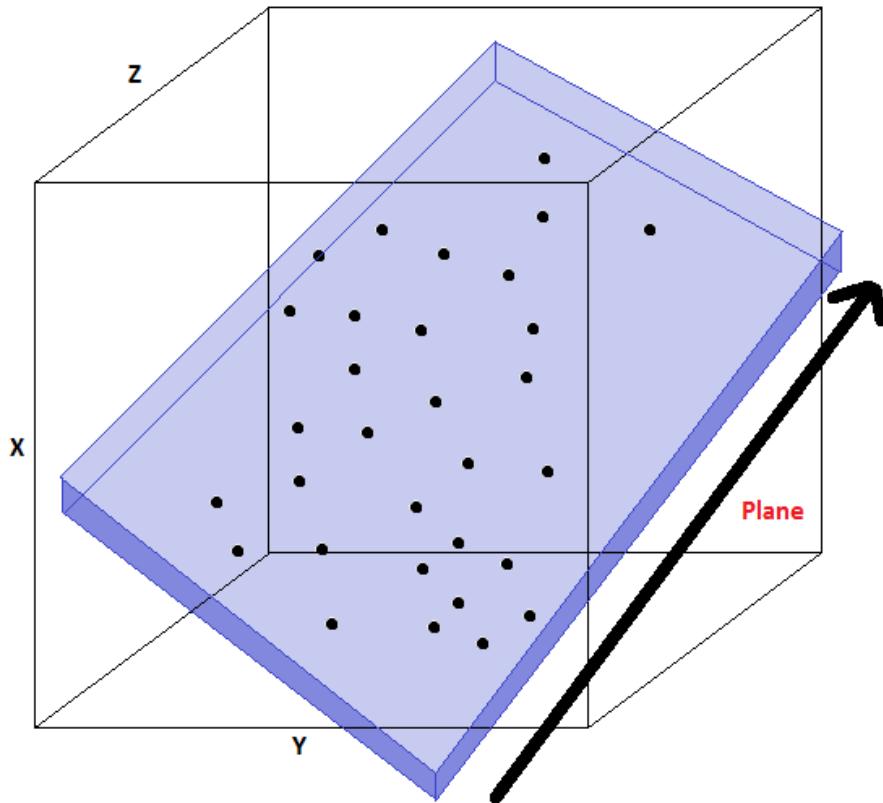
- Example above isn't a perfect straight line because of round-off error
- Data redundancy can happen when different teams are working independently
 - Often generates redundant data (especially if you don't control data collection)
- Another example
 - Helicopter flying - do a survey of pilots (x_1 = skill, x_2 = pilot enjoyment)
 - These features may be highly correlated
 - This correlation can be combined into a single attribute called aptitude (for example)
- What does dimensionality reduction mean?
 - In our example we plot a line
 - Take exact example and record position on that line



- So before x^1 was a 2D feature vector (X and Y dimensions)
- Now we can represent x^1 as a 1D number (Z dimension)
- So we approximate original examples
 - Allows us to halve the amount of storage
 - Gives lossy compression, but an acceptable loss (probably)
 - The loss above comes from the rounding error in the measurement, however
- Another example 3D \rightarrow 2D
 - So here's our data

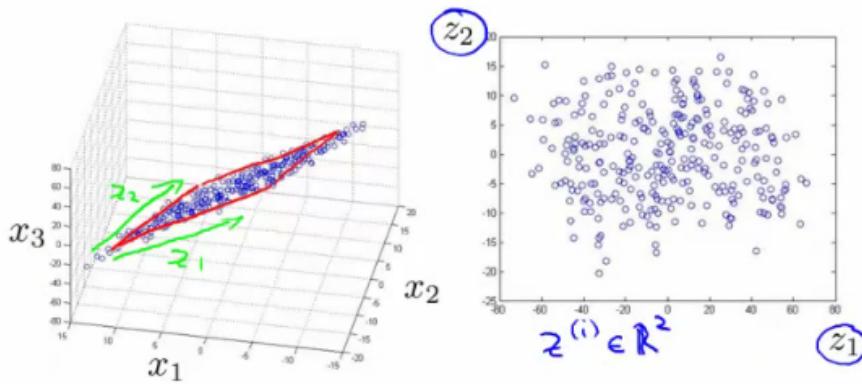


- Maybe all the data lies in one plane
 - This is sort of hard to explain in 2D graphics, but that plane may be aligned with one of the axis
 - Or or may not...
 - Either way, the plane is a small, a constant 3D space
 - In the diagram below, imagine all our data points are sitting "inside" the blue tray (has a dark blue exterior face and a light blue inside)



- Because they're all in this relative shallow area, we can basically ignore one of the dimension, so we draw two new lines (z_1 and z_2) along the x and y planes of the box, and plot the locations in that box
- i.e. we loose the data in the z-dimension of our "shallow box" (NB "z-dimensions" here refers to the dimension relative to the box (i.e it's depth) and NOT the z dimension of the axis we've got drawn above) but because the box is shallow it's OK to lose this. Probably....

- Plot values along those projections



- So we've now reduced our 3D vector to a 2D vector

- In reality we'd normally try and do 1000D → 100D

Motivation 2: Visualization

- It's hard to visualize highly dimensional data
 - Dimensionality reduction can improve how we display information in a tractable manner for human consumption
 - Why do we care?
 - Often helps to develop algorithms if we can understand our data better
 - Dimensionality reduction helps us do this, see data in a helpful
 - Good for explaining something to someone if you can "show" it in the data
- Example;
 - Collect a large data set about many facts of a country around the world

Country	GDP (trillions of US\$)	Per capita GDP (thousands of intl. \$)	Human Development Index	Life expectancy	Poverty Index (Gini as percentage)	Mean household income (thousands of US\$)	...
Canada	1.577	39.17	0.908	80.7	32.6	67.293	...
China	5.878	7.54	0.687	73	46.9	10.22	...
India	1.632	3.41	0.547	64.7	36.8	0.735	...
Russia	1.48	19.84	0.755	65.5	39.9	0.72	...
Singapore	0.223	56.69	0.866	80	42.5	67.1	...
USA	14.527	46.86	0.91	78.3	40.8	84.3	...
...

- So
 - $x_1 = \text{GDP}$
 - ...
 - $x_6 = \text{mean household}$
- Say we have 50 features per country
- How can we understand this data better?
 - Very hard to plot 50 dimensional data
- Using dimensionality reduction, instead of each country being represented by a 50-dimensional feature vector

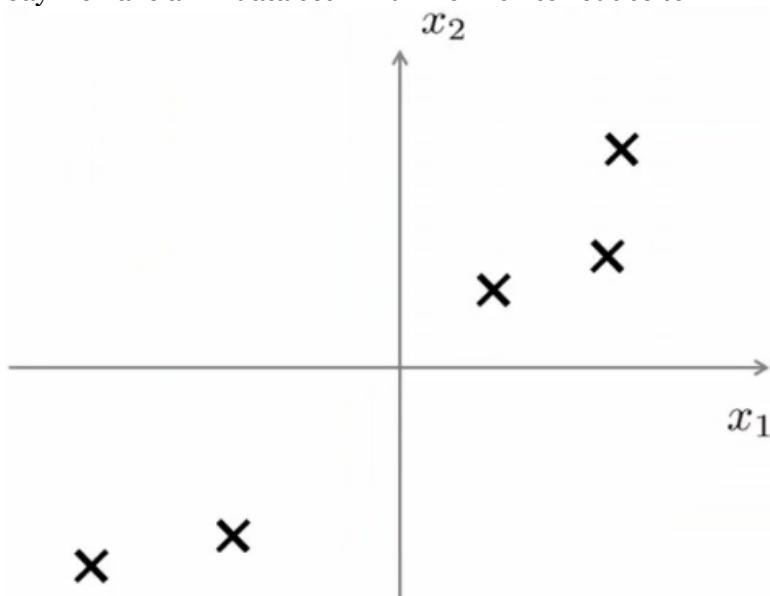
- Come up with a different feature representation (z values) which summarize these features

Country	z_1	z_2
Canada	1.6	1.2
China	1.7	0.3
India	1.6	0.2
Russia	1.4	0.5
Singapore	0.5	1.7
USA	2	1.5
...

- This gives us a 2-dimensional vector
 - Reduce 50D -> 2D
 - Plot as a 2D plot
- Typically you don't generally ascribe meaning to the new features (so we have to determine what these summary values mean)
 - e.g. may find horizontal axis corresponds to overall country size/economic activity
 - and y axis may be the per-person well being/economic activity
- So despite having 50 features, there may be two "dimensions" of information, with features associated with each of those dimensions
 - It's up to you to assess what of the features can be grouped to form summary features, and how best to do that (feature scaling is probably important)
- Helps show the two main dimensions of variation in a way that's easy to understand

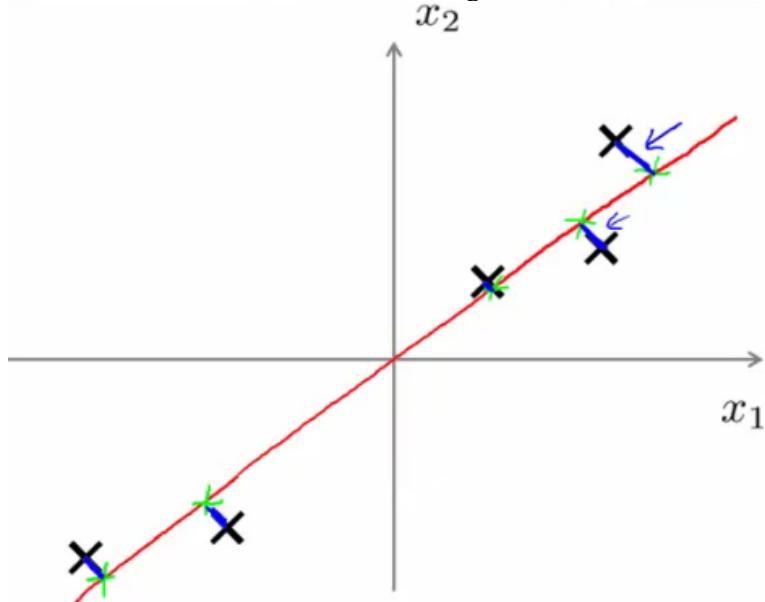
Principle Component Analysis (PCA): Problem Formulation

- For the problem of dimensionality reduction the most commonly used algorithm is **PCA**
 - Here, we'll start talking about how we formulate precisely what we want PCA to do
- So
 - Say we have a 2D data set which we wish to reduce to 1D



- In other words, find a single line onto which to project this data
 - How do we determine this line?

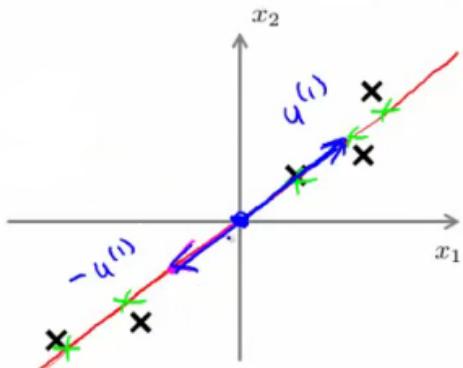
- The distance between each point and the projected version should be small (blue lines below are short)
- PCA tries to find a lower dimensional surface so the sum of squares onto that surface is minimized
- The blue lines are sometimes called the **projection error**
 - PCA tries to find the surface (a straight line in this case) which has the minimum projection error



- As an aside, you should normally do **mean normalization** and **feature scaling** on your data before PCA

- A more formal description is

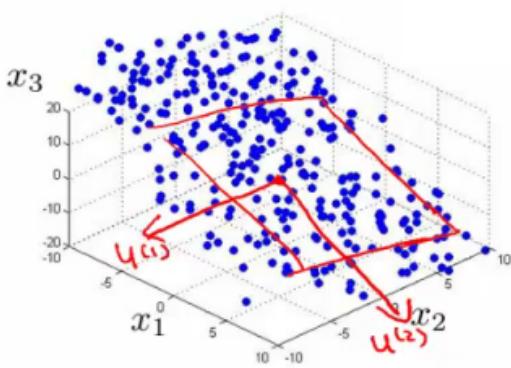
- For 2D-1D, we must find a vector $u^{(1)}$, which is of some dimensionality
- Onto which you can project the data so as to minimize the projection error



- $u^{(1)}$ can be positive or negative ($-u^{(1)}$) which makes no difference
 - Each of the vectors define the same red line

- In the more general case

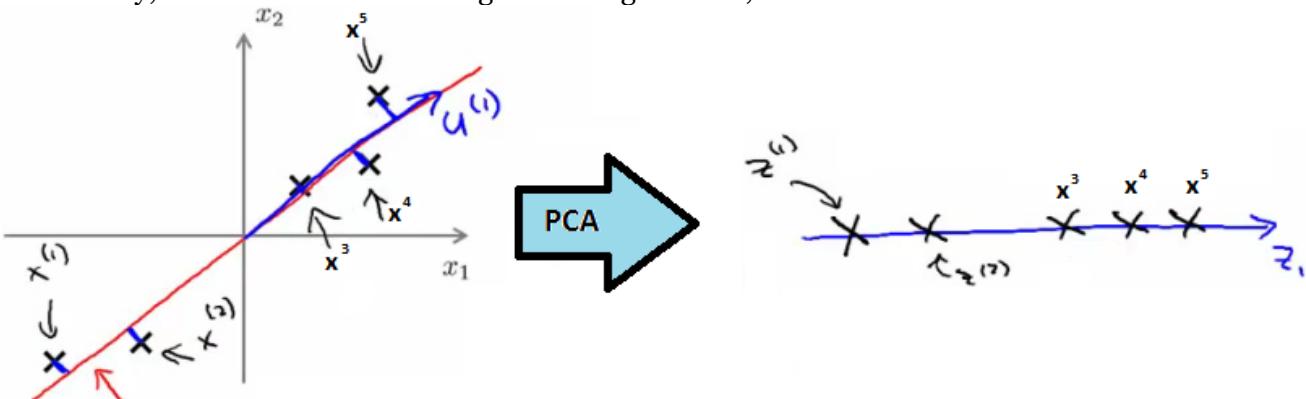
- To reduce from nD to kD we
 - Find k vectors ($u^{(1)}, u^{(2)}, \dots, u^{(k)}$) onto which to project the data to minimize the projection error
 - So lots of vectors onto which we project the data
 - Find a set of vectors which we project the data onto the linear subspace spanned by that set of vectors
 - We can define a point in a plane with k vectors
- e.g. 3D->2D
 - Find pair of vectors which define a 2D plane (surface) onto which you're going to project your data
 - Much like the "shallow box" example in compression, we're trying to create the shallowest box possible (by defining two of its three dimensions, so the box's depth is minimized)



- How does PCA relate to linear regression?
 - PCA is **not** linear regression
 - Despite cosmetic similarities, very different
 - For linear regression, fitting a straight line to minimize the **straight line** between a point and a squared line
 - NB - **VERTICAL distance** between point
 - For PCA minimizing the magnitude of the shortest **orthogonal distance**
 - Gives very different effects
 - More generally
 - With linear regression we're trying to predict "y"
 - With PCA there is no "y" - instead we have a list of features and all features are treated equally
 - If we have 3D dimensional data 3D->2D
 - Have 3 features treated symmetrically

PCA Algorithm

- Before applying PCA must do data preprocessing
 - Given a set of m unlabeled examples we must do
 - **Mean normalization**
 - Replace each x_j^i with $x_j^i - \mu_j$,
 - In other words, determine the mean of each feature set, and then for each feature subtract the mean from the value, so we re-scale the mean to be 0
 - **Feature scaling (depending on data)**
 - If features have very different scales then scale so they all have a comparable range of values
 - e.g. x_j^i is set to $(x_j^i - \mu_j) / s_j$
 - Where s_j is some measure of the range, so could be
 - Biggest - smallest
 - Standard deviation (more commonly)
 - With preprocessing done, PCA finds the lower dimensional sub-space which minimizes the sum of the square
 - In summary, for 2D->1D we'd be doing something like this;



- Need to compute two things;
 - Compute the **u vectors**
 - The new planes
 - Need to compute the **z vectors**
 - z vectors are the new, lower dimensionality feature vectors
- A mathematical derivation for the u vectors is very complicated
 - But once you've done it, the procedure to find each u vector is not that hard

Algorithm description

- Reducing data from n -dimensional to k -dimensional
 - Compute the covariance matrix

$$\Sigma = \frac{1}{m} \sum_{i=1}^n (x^{(i)})(x^{(i)})^T$$

- This is commonly denoted as Σ (greek upper case sigma) - NOT summation symbol
- $\Sigma = \text{sigma}$
 - This is an $[n \times n]$ matrix
 - Remember that x^i is a $[n \times 1]$ matrix
- In MATLAB or octave we can implement this as follows;

$$\text{sigma} = (1/m) * (X^T * X)$$

- Compute eigenvectors of matrix Σ

- `[U, S, V] = svd(sigma)`
 - svd = singular value decomposition
 - More numerically stable than `eig`
 - `eig` = also gives eigenvector

- U, S and V are matrices

- U matrix is also an $[n \times n]$ matrix
- Turns out the columns of U are the u vectors we want!
- So to reduce a system from n -dimensions to k -dimensions
 - Just take the first k -vectors from U (first k columns)

$$U = \begin{bmatrix} u^{(1)} & u^{(2)} & \dots & u^{(n)} \\ | & | & & | \end{bmatrix} \in \mathbb{R}^{n \times n}$$

- Next we need to find some way to change x (which is n dimensional) to z (which is k dimensional)

- (reduce the dimensionality)
- Take first k columns of the u matrix and stack in columns
 - $n \times k$ matrix - call this U_{reduce}

- We calculate z as follows

- $z = (U_{\text{reduce}})^T * x$
 - So $[k \times n] * [n \times 1]$
 - Generates a matrix which is
 - $k \times 1$
 - If that's not witchcraft I don't know what is!

- Exactly the same as with supervised learning except we're now doing it with unlabeled data

- So in summary

- Preprocessing
- Calculate sigma (covariance matrix)
- Calculate eigenvectors with `svd`
- Take k vectors from U ($U_{\text{reduce}} = U(:,1:k)$)
- Calculate z ($z = U_{\text{reduce}}^T * x$)

- No mathematical derivation

- Very complicated
- But it works

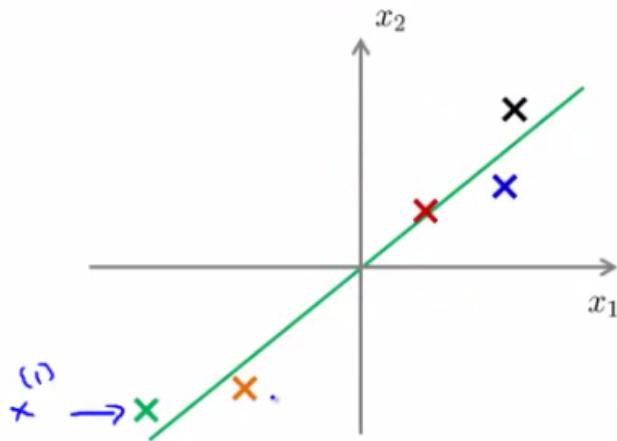
Reconstruction from Compressed Representation

- Earlier spoke about PCA as a compression algorithm

- If this is the case, is there a way to **decompress** the data from low dimensionality back to a higher dimensionality format?

- Reconstruction

- Say we have an example as follows

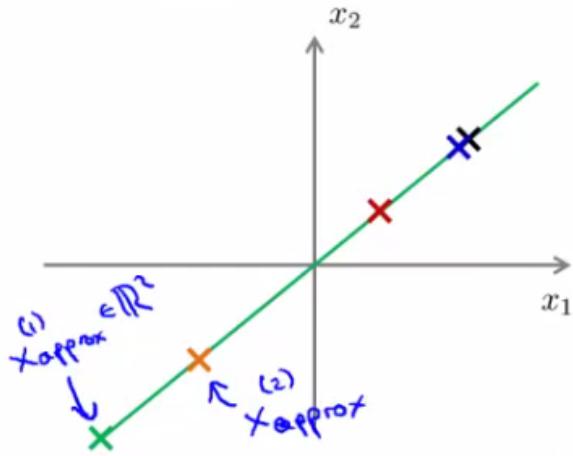


$$z = U_{reduce}^T x$$



- We have our examples (x^1, x^2 etc.)
- Project onto z -surface
- Given a point z^1 , how can we go back to the 2D space?

- Considering
 - z (vector) = $(U_{reduce})^T * x$
- To go in the opposite direction we must do
 - $x_{approx} = U_{reduce} * z$
 - To consider dimensions (and prove this really works)
 - $U_{reduce} = [n \times k]$
 - $z [k \times 1]$
 - So
 - $x_{approx} = [n \times 1]$
- So this creates the following representation



- We lose some of the information (i.e. everything is now perfectly on that line) but it is now projected into 2D space

Choosing the number of Principle Components

- How do we chose k ?
 - $k = \text{number of principle components}$
 - Guidelines about how to chose k for PCA
- To chose k think about how PCA works

- PCA tries to minimize averaged squared projection error

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

- Total variation in data can be defined as the average over data saying how far are the training examples from the origin

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

- When we're choosing k typical to use something like this

$$\frac{\frac{1}{m} \sum_{i=1}^m \|x^{(i)} - x_{approx}^{(i)}\|^2}{\frac{1}{m} \sum_{i=1}^m \|x^{(i)}\|^2} \leq 0.01 \quad (1\%)$$

- Ratio between averaged squared projection error with total variation in data
 - Want ratio to be small - means we retain 99% of the variance
- If it's small (o) then this is because the numerator is small
 - The numerator is small when $x^{(i)} = x_{approx}^{(i)}$
 - i.e. we lose very little information in the dimensionality reduction, so when we decompress we regenerate the same data

- So we chose k in terms of this ratio

- Often can significantly reduce data dimensionality while retaining the variance

- How do you do this

Algorithm:

Try PCA with $k = 1$

Compute $U_{reduce}, z^{(1)}, z^{(2)}, \dots, z^{(m)}, x_{approx}^{(1)}, \dots, x_{approx}^{(m)}$

Check if

$$\frac{\frac{1}{m} \sum_{i=1}^m \|x^{(i)} - x_{approx}^{(i)}\|^2}{\frac{1}{m} \sum_{i=1}^m \|x^{(i)}\|^2} \leq 0.01?$$

Advice for Applying PCA

- Can use PCA to speed up algorithm running time
 - Explain how
 - And give general advice

Speeding up supervised learning algorithms

- Say you have a supervised learning problem
 - Input x and y
 - x is a 10 000 dimensional feature vector
 - e.g. 100 x 100 images = 10 000 pixels
 - Such a huge feature vector will make the algorithm slow
 - With PCA we can reduce the dimensionality and make it tractable
 - How
 - 1) Extract xs
 - So we now have an unlabeled training set
 - 2) Apply PCA to x vectors
 - So we now have a reduced dimensional feature vector z
 - 3) This gives you a new training set
 - Each vector can be re-associated with the label

- 4) Take the reduced dimensionality data set and feed to a learning algorithm
 - Use y as labels and z as feature vector
- 5) If you have a new example map from higher dimensionality vector to lower dimensionality vector, then feed into learning algorithm
- PCA maps one vector to a lower dimensionality vector
 - $x \rightarrow z$
 - Defined by PCA **only** on the training set
 - The mapping computes a set of parameters
 - Feature scaling values
 - U_{reduce}
 - Parameter learned by PCA
 - Should be obtained only by determining PCA on your training set
 - So we use those learned parameters for our
 - Cross validation data
 - Test set
- Typically you can reduce data dimensionality by 5-10x without a major hit to algorithm

Applications of PCA

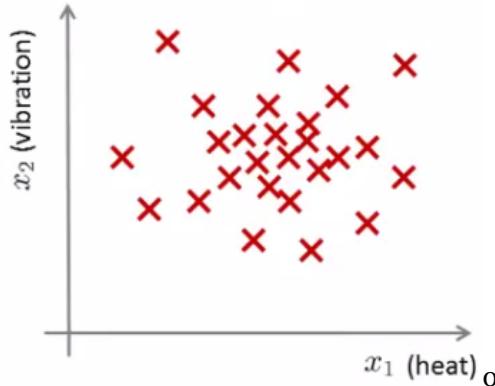
- **Compression**
 - Why
 - Reduce memory/disk needed to store data
 - Speed up learning algorithm
 - How do we chose k ?
 - % of variance retained
- **Visualization**
 - Typically chose $k = 2$ or $k = 3$
 - Because we can plot these values!
- One thing often done wrong regarding PCA
 - A bad use of PCA: Use it to prevent over-fitting
 - Reasoning
 - If we have x^i we have n features, z^i has k features which can be lower
 - If we *only* have k features then maybe we're less likely to over fit...
 - This doesn't work
 - BAD APPLICATION
 - Might work OK, but not a good way to address over fitting
 - Better to use regularization
 - PCA throws away some data without knowing what the values it's losing
 - Probably OK if you're keeping most of the data
 - But if you're throwing away some crucial data bad
 - So you have to go to like 95-99% variance retained
 - So here regularization will give you AT LEAST as good a way to solve over fitting
 - A second PCA myth
 - Used for compression or visualization - good
 - Sometimes used
 - Design ML system with PCA from the outset
 - But, what if you did the whole thing without PCA
 - See how a system performs without PCA
 - ONLY if you have a reason to believe PCA will help should you then add PCA
 - PCA is easy enough to add on as a processing step
 - Try without first!

15: Anomaly Detection

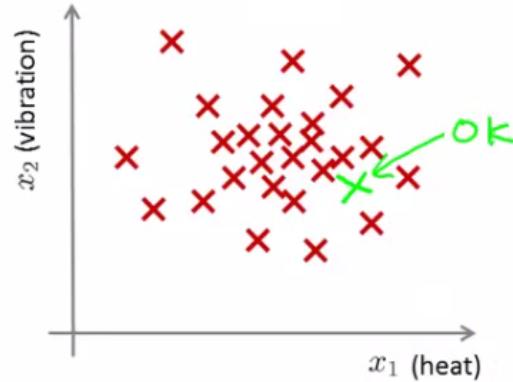
[Previous](#) [Next](#) [Index](#)

Anomaly detection - problem motivation

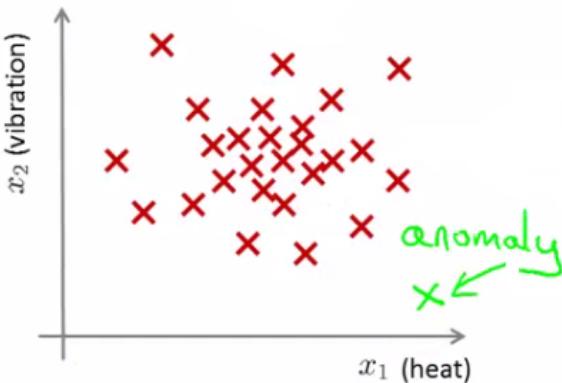
- Anomaly detection is a reasonably commonly used type of machine learning application
 - Can be thought of as a solution to an unsupervised learning problem
 - But, has aspects of supervised learning
- What is anomaly detection?
 - Imagine you're an aircraft engine manufacturer
 - As engines roll off your assembly line you're doing QA
 - Measure some features from engines (e.g. heat generated and vibration)
 - You now have a dataset of x^1 to x^m (i.e. m engines were tested)
 - Say we plot that dataset



- Next day you have a new engine
 - An anomaly detection method is used to see if the new engine is anomalous (when compared to the previous engines)
- If the new engine looks like this;



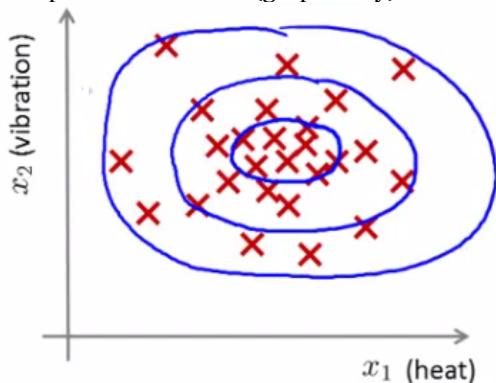
- Probably OK - looks like the ones we've seen before
- But if the engine looks like this



- Uh oh! - this looks like an **anomalous data-point**

- More formally
 - We have a dataset which contains **normal** (data)
 - How we ensure they're normal is up to us
 - In reality it's OK if there are a few which aren't actually normal
 - Using that dataset as a reference point we can see if other examples are **anomalous**
- How do we do this?
 - First, using our training dataset we build a model

- We can access this model using $p(\mathbf{x})$
 - This asks, "What is the probability that example \mathbf{x} is normal?"
- Having built a model
 - if $p(\mathbf{x}_{\text{test}}) < \varepsilon \rightarrow$ flag this as an anomaly
 - if $p(\mathbf{x}_{\text{test}}) \geq \varepsilon \rightarrow$ this is OK
 - ε is some threshold probability value which we define, depending on how sure we need/want to be
- We expect our model to (graphically) look something like this;



- i.e. this would be our model if we had 2D data

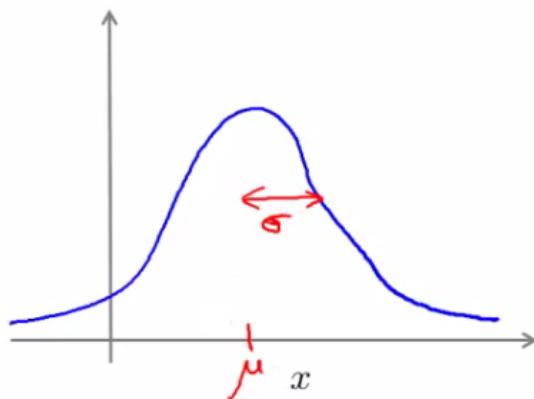
Applications

- Fraud detection
 - Users have activity associated with them, such as
 - Length on time on-line
 - Location of login
 - Spending frequency
 - Using this data we can build a model of what normal users' activity is like
 - What is the probability of "normal" behavior?
 - Identify unusual users by sending their data through the model
 - Flag up anything that looks a bit weird
 - Automatically block cards/transactions
- Manufacturing
 - Already spoke about aircraft engine example
- Monitoring computers in data center
 - If you have many machines in a cluster
 - Computer features of machine
 - x_1 = memory use
 - x_2 = number of disk accesses/sec
 - x_3 = CPU load
 - In addition to the measurable features you can also define your own complex features
 - x_4 = CPU load/network traffic
 - If you see an anomalous machine
 - Maybe about to fail
 - Look at replacing bits from it

The Gaussian distribution (optional)

- Also called the **normal distribution**
- Example
 - Say \mathbf{x} (data set) is made up of real numbers
 - Mean is μ
 - Variance is σ^2
 - σ is also called the **standard deviation** - specifies the width of the Gaussian probability
 - The data has a Gaussian distribution
 - Then we can write this $\sim N(\mu, \sigma^2)$
 - \sim means = is distributed as
 - N (should really be "script" N (even curlier!) \rightarrow means normal distribution)
 - μ, σ^2 represent the mean and variance, respectively
 - These are the two parameters a Gaussian means

- Looks like this;

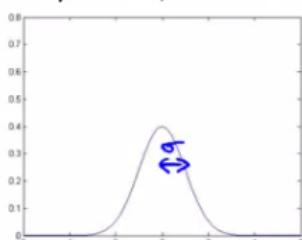


- This specifies the probability of x taking a value
 - As you move away from μ
- Gaussian equation is
 - $P(x : \mu, \sigma^2)$ (probability of x , parameterized by the mean and squared variance)

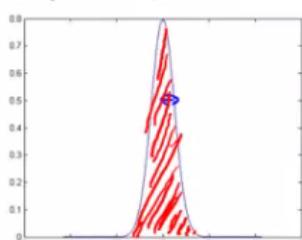
$$= \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$$

- Some examples of Gaussians below
 - Area is always the same (must = 1)
 - But width changes as standard deviation changes

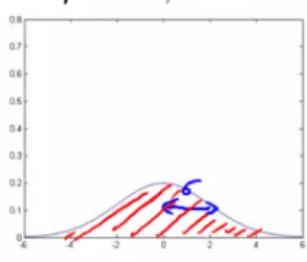
$\mu = 0, \sigma = 1$



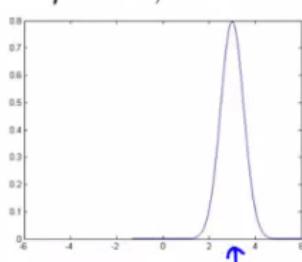
$\mu = 0, \sigma = 0.5$



$\mu = 0, \sigma = 2$

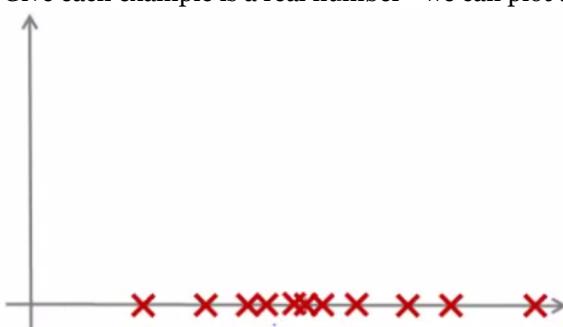


$\mu = 3, \sigma = 0.5$



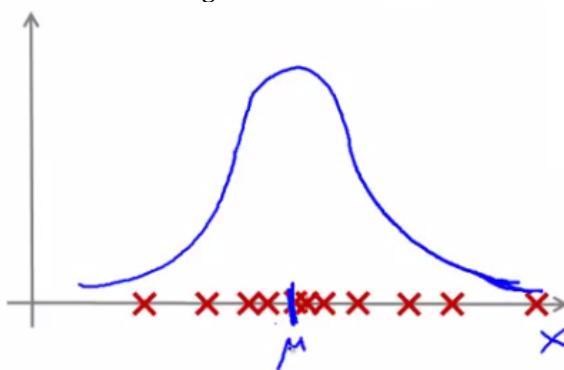
Parameter estimation problem

- What is it?
 - Say we have a data set of m examples
 - Give each example is a real number - we can plot the data on the x axis as shown below



- Problem is - say you suspect these examples come from a Gaussian
 - Given the dataset can you estimate the distribution?

- Could be something like this



- Seems like a reasonable fit - data seems like a higher probability of being in the central region, lower probability of being further away

- Estimating μ and σ^2

- μ = average of examples
- σ^2 = standard deviation squared

$$\sigma^2 = \frac{1}{m} \sum_{i=1}^m (x^{(i)} - \mu)^2$$

- As a side comment

- These parameters are the maximum likelihood estimation values for μ and σ^2
- You can also do $1/(m)$ or $1/(m-1)$ doesn't make too much difference
 - Slightly different mathematical problems, but in practice it makes little difference

Anomaly detection algorithm

- Unlabeled training set of m examples

- Data = $\{x^1, x^2, \dots, x^m\}$
 - Each example is an n -dimensional vector (i.e. a feature vector)
 - We have n features!
- Model $P(x)$ from the data set
 - What are high probability features and low probability features
 - x is a vector
 - So model $p(x)$ as
 - $= p(x_1; \mu_1, \sigma_1^2) * p(x_2; \mu_2, \sigma_2^2) * \dots * p(x_n; \mu_n, \sigma_n^2)$
 - Multiply the probability of each features by each feature
 - We model each of the features by assuming each feature is distributed according to a Gaussian distribution
 - $p(x_i; \mu_i, \sigma_i^2)$
 - The probability of feature x_i given μ_i and σ_i^2 , using a Gaussian distribution
 - As a side comment
 - Turns out this equation makes an **independence assumption** for the features, although algorithm works if features are independent or not
 - Don't worry too much about this, although if your features are tightly linked you should be able to do some dimensionality reduction anyway!
 - We can write this chain of multiplication more compactly as follows;

$$= \prod_{j=1}^n p(x_j; \mu_j, \sigma_j^2)$$
 - Capital PI (Π) is the product of a set of values
 - The problem of estimating this distribution is sometimes called the problem of **density estimation**

Algorithm

1. Choose features x_i that you think might be indicative of anomalous examples.

2. Fit parameters $\mu_1, \dots, \mu_n, \sigma_1^2, \dots, \sigma_n^2$

$$\mu_j = \frac{1}{m} \sum_{i=1}^m x_j^{(i)}$$

$$\sigma_j^2 = \frac{1}{m} \sum_{i=1}^m (x_j^{(i)} - \mu_j)^2$$

3. Given new example x , compute $p(x)$:

$$p(x) = \prod_{j=1}^n p(x_j; \mu_j, \sigma_j^2) = \prod_{j=1}^n \frac{1}{\sqrt{2\pi}\sigma_j} \exp\left(-\frac{(x_j - \mu_j)^2}{2\sigma_j^2}\right)$$

Anomaly if $p(x) < \varepsilon$

- 1 - Chose features

- Try to come up with features which might help identify something anomalous - may be unusually large or small values
- More generally, chose features which describe the general properties
- This is nothing unique to anomaly detection - it's just the idea of building a sensible feature vector

- 2 - Fit parameters

- Determine parameters for each of your examples μ_i and σ_i^2
 - Fit is a bit misleading, really should just be "Calculate parameters for 1 to n"
- So you're calculating standard deviation and mean for each feature
- You should of course used some vectorized implementation rather than a loop probably

- 3 - compute $p(x)$

- You compute the formula shown (i.e. the formula for the Gaussian probability)
- If the number is very small, very low chance of it being "normal"

Anomaly detection example

- x_1

- Mean is about 5
- Standard deviation looks to be about 2

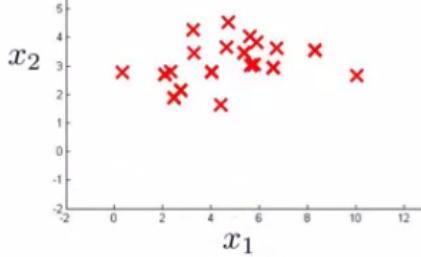
- x_2

- Mean is about 3
- Standard deviation about 1

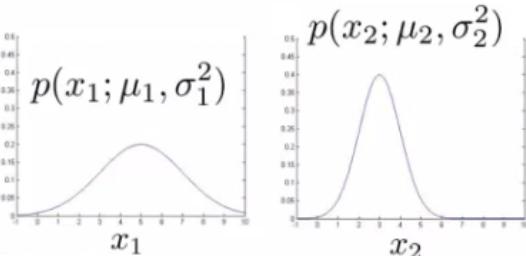
- So we have the following system

$$\mu_1 = 5, \sigma_1 = 2$$

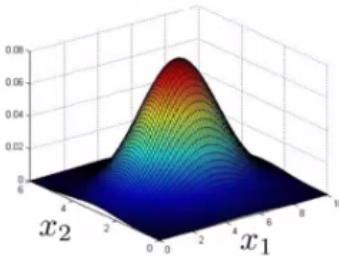
$$\mu_2 = 3, \sigma_2 = 1$$



- If we plot the Gaussian for x_1 and x_2 we get something like this



- If you plot the product of these things you get a surface plot like this



- With this surface plot, the height of the surface is the probability - $p(x)$
- We can't always do surface plots, but for this example it's quite a nice way to show the probability of a 2D feature vector
- Check if a value is anomalous
 - Set epsilon as some value
 - Say we have two new data points new data-point has the values
 - x^1_{test}
 - x^2_{test}
 - We compute
 - $p(x^1_{\text{test}}) = 0.436 \geq \epsilon$ (~40% chance it's normal)
 - Normal
 - $p(x^2_{\text{test}}) = 0.0021 < \epsilon$ (~0.2% chance it's normal)
 - Anomalous
 - What this is saying is if you look at the surface plot, all values above a certain height are normal, all the values below that threshold are probably anomalous

Developing and evaluating and anomaly detection system

- Here talk about developing a system for anomaly detection
 - How to evaluate an algorithm
- Previously we spoke about the importance of real-number evaluation
 - Often need to make a lot of choices (e.g. features to use)
 - Easier to evaluate your algorithm if it returns a **single number** to show if changes you made improved or worsened an algorithm's performance
 - To develop an anomaly detection system quickly, would be helpful to have a way to evaluate your algorithm
- Assume we have some labeled data
 - So far we've been treating anomalous detection with unlabeled data
 - If you have labeled data allows evaluation
 - i.e. if you think something is anomalous you can be sure if it is or not
- So, taking our engine example
 - You have some labeled data
 - Data for engines which were non-anomalous $\rightarrow y = 0$
 - Data for engines which were anomalous $\rightarrow y = 1$
 - Training set is the collection of normal examples
 - OK even if we have a few anomalous data examples
 - Next define
 - Cross validation set
 - Test set
 - For both assume you can include a few examples which have anomalous examples
 - Specific example
 - Engines
 - Have 10 000 good engines
 - OK even if a few bad ones are here...
 - LOTS of $y = 0$
 - 20 flawed engines
 - Typically when $y = 1$ have 2-50
 - Split into
 - Training set: 6000 good engines ($y = 0$)
 - CV set: 2000 good engines, 10 anomalous
 - Test set: 2000 good engines, 10 anomalous
 - Ratio is 3:1:1
 - Sometimes we see a different way of splitting
 - Take 6000 good in training
 - Same CV and test set (4000 good in each) different 10 anomalous,
 - Or even 20 anomalous (same ones)
 - This is bad practice - should use different data in CV and test set
 - Algorithm evaluation
 - Take trainings set $\{x^1, x^2, \dots, x^m\}$
 - Fit model $p(x)$

- On cross validation and test set, test the example x
 - $y = 1$ if $p(x) < \text{epsilon}$ (anomalous)
 - $y = 0$ if $p(x) \geq \text{epsilon}$ (normal)
- Think of algorithm trying to predict if something is anomalous
 - But you have a label so can check!
 - Makes it look like a supervised learning algorithm
- What's a good metric to use for evaluation
 - $y = 0$ is very common
 - So classification would be bad
 - Compute fraction of true positives/false positive/false negative/true negative
 - Compute precision/recall
 - Compute F1-score
- Earlier, also had **epsilon** (the threshold value)
 - Threshold to show when something is anomalous
 - If you have CV set you can see how varying epsilon effects various evaluation metrics
 - Then pick the value of epsilon which maximizes the score on your CV set
 - Evaluate algorithm using cross validation
 - Do final algorithm evaluation on the test set

Anomaly detection vs. supervised learning

- If we have labeled data, we not use a supervised learning algorithm?
 - Here we'll try and understand when you should use supervised learning and when anomaly detection would be better

Anomaly detection

- **Very small number of positive examples**
 - Save positive examples just for CV and test set
 - Consider using an anomaly detection algorithm
 - Not enough data to "learn" positive examples
- **Have a very large number of negative examples**
 - Use these negative examples for $p(x)$ fitting
 - Only need negative examples for this
- **Many "types" of anomalies**
 - Hard for an algorithm to learn from positive examples when anomalies may look nothing like one another
 - So anomaly detection doesn't know what they look like, but knows what they *don't* look like
 - When we looked at SPAM email,
 - Many types of SPAM
 - For the spam problem, usually enough positive examples
 - So this is why we usually think of SPAM as supervised learning
- Application and why they're anomaly detection
 - **Fraud detection**
 - Many ways you may do fraud
 - If you're a major on line retailer/very subject to attacks, sometimes might shift to supervised learning
 - **Manufacturing**
 - If you make HUGE volumes maybe have enough positive data -> make supervised
 - Means you make an assumption about the kinds of errors you're going to see
 - It's the unknown unknowns we don't like!
 - **Monitoring machines in data**

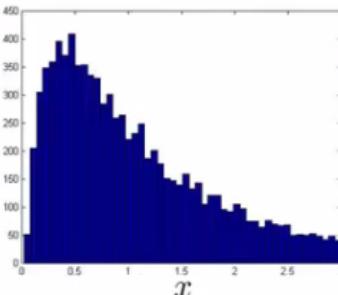
Supervised learning

- **Reasonably large number of positive and negative examples**
- Have enough positive examples to give your algorithm the opportunity to see what they look like
 - If you expect anomalies to look anomalous in the same way
- Application
 - Email/SPAM classification
 - Weather prediction
 - Cancer classification

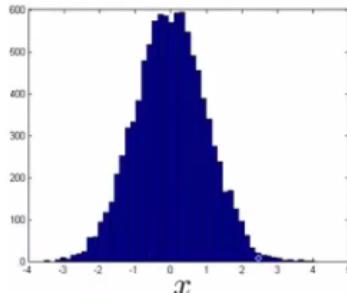
Choosing features to use

- One of the things which has a huge effect is which features are used
- **Non-Gaussian features**
 - Plot a histogram of data to check it has a Gaussian description - nice sanity check
 - Often still works if data is non-Gaussian
 - Use **hist** command to plot histogram

- Non-Gaussian data might look like this



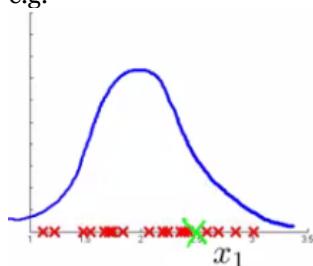
- Can play with different transformations of the data to make it look more Gaussian
- Might take a log transformation of the data
 - i.e. if you have some feature x_1 , replace it with $\log(x_1)$



- This looks much more Gaussian
- Or do $\log(x_1+c)$
 - Play with c to make it look as Gaussian as possible
- Or do $x^{1/2}$
- Or do $x^{1/3}$

Error analysis for anomaly detection

- Good way of coming up with features
- Like supervised learning error analysis procedure
 - Run algorithm on CV set
 - See which one it got wrong
 - Develop new features based on trying to understand *why* the algorithm got those examples wrong
- Example
 - $p(x)$ large for normal, $p(x)$ small for abnormal
 - e.g.

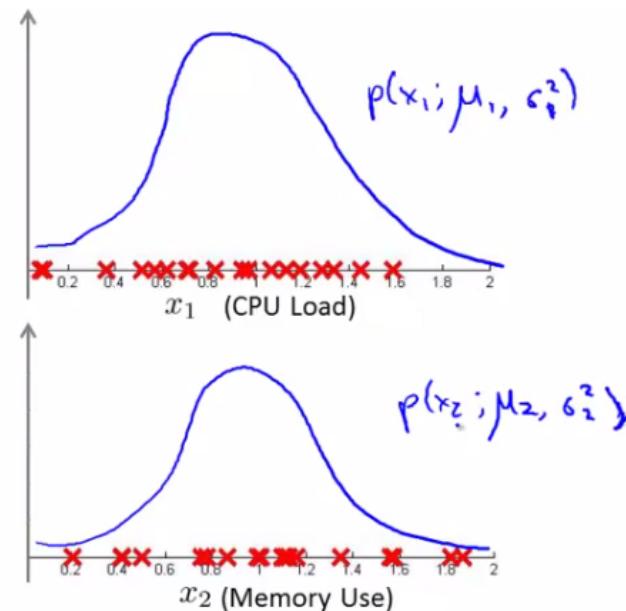
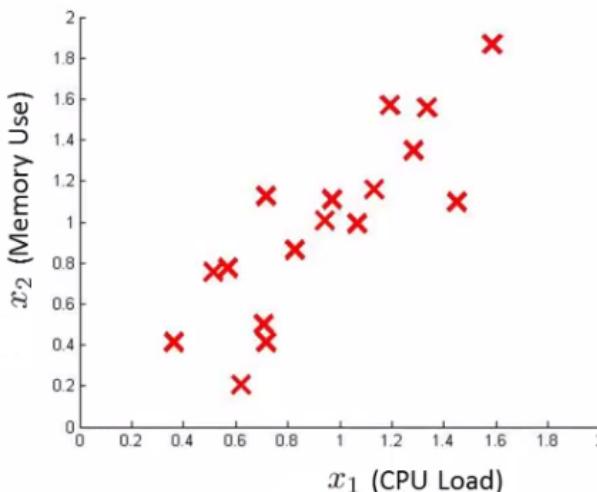


- Here we have one dimension, and our anomalous value is sort of buried in it (in green - Gaussian superimposed in blue)
 - Look at data - see what went wrong
 - Can looking at that example help develop a new feature (x_2) which can help distinguish further anomalies
- Example - data center monitoring
 - Features
 - x_1 = memory use
 - x_2 = number of disk access/sec
 - x_3 = CPU load
 - x_4 = network traffic
 - We suspect CPU load and network traffic grow linearly with one another
 - If server is serving many users, CPU is high and network is high
 - Fail case is infinite loop, so CPU load grows but network traffic is low
 - New feature - CPU load/network traffic
 - May need to do feature scaling

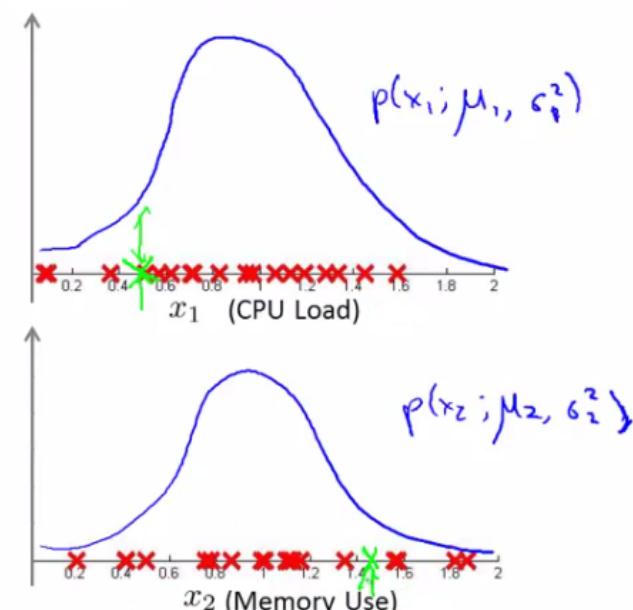
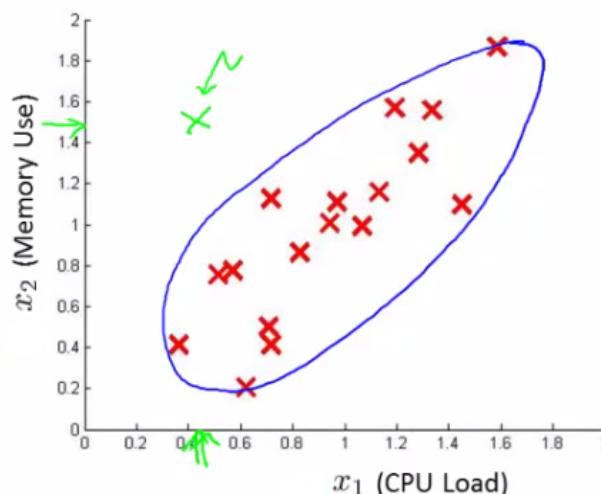
Multivariate Gaussian distribution

- Is a slightly different technique which can sometimes catch some anomalies which non-multivariate Gaussian distribution anomaly detection fails to

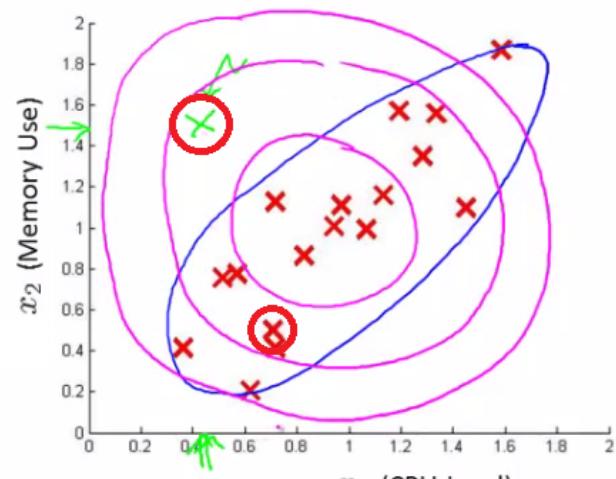
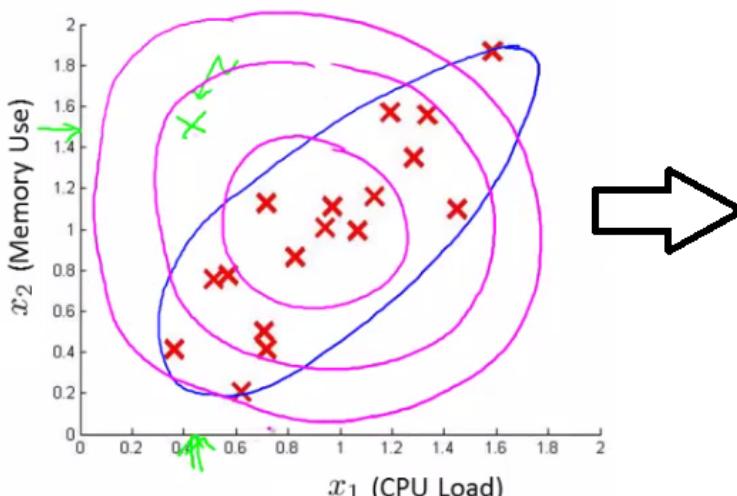
- Unlabeled data looks like this



- Say you can fit a Gaussian distribution to CPU load and memory use
- Lets say in the test set we have an example which looks like an anomaly (e.g. $x_1 = 0.4$, $x_2 = 1.5$)
 - Looks like most of data lies in a region far away from this example
 - Here memory use is high and CPU load is low (if we plot x_1 vs. x_2 our green example looks miles away from the others)
- Problem is, if we look at each feature individually they may fall within acceptable limits - the issue is we know we shouldn't don't get those kinds of values **together**
 - But individually, they're both acceptable



- This is because our function makes probability prediction in concentric circles around the the means of both



- Probability of the two red circled examples is basically the same, even though we can clearly see the green one as an outlier

- Doesn't understand the meaning

Multivariate Gaussian distribution model

- To get around this we develop the **multivariate Gaussian distribution**
 - Model p(x) all in one go, instead of each feature separately
 - What are the parameters for this new model?
 - μ - which is an n dimensional vector (where n is number of features)
 - Σ - which is an $[n \times n]$ matrix - the **covariance matrix**
- For the sake of completeness, the formula for the multivariate Gaussian distribution is as follows

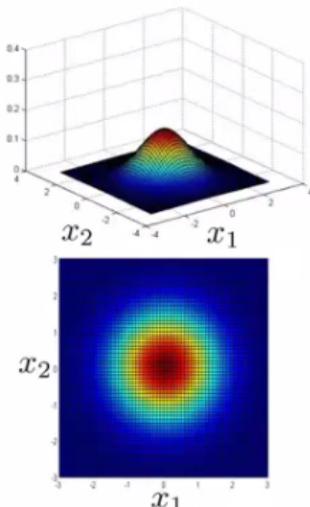
$$p(x; \mu, \Sigma) = \frac{1}{(2\pi)^{n/2} |\Sigma|^{1/2}} \exp(-\frac{1}{2} (x-\mu)^T \Sigma^{-1} (x-\mu))$$

- NB don't memorize this - you can always look it up
- What does this mean?
 - $|\Sigma|$ = absolute value of Σ (determinant of sigma)
 - This is a mathematic function of a matrix
 - You can compute it in MATLAB using `det(sigma)`

- More importantly, what does this $p(x)$ look like?
 - 2D example

$$\mu = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \quad \Sigma = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

- Sigma is sometimes call the identity matrix

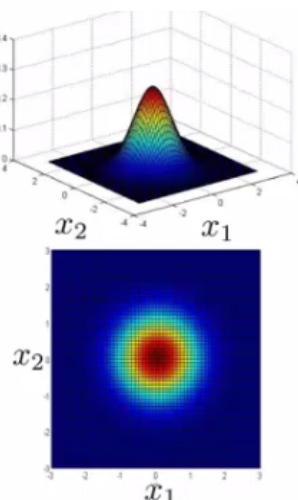


- $p(x)$ looks like this
 - For inputs of x_1 and x_2 the height of the surface gives the value of $p(x)$

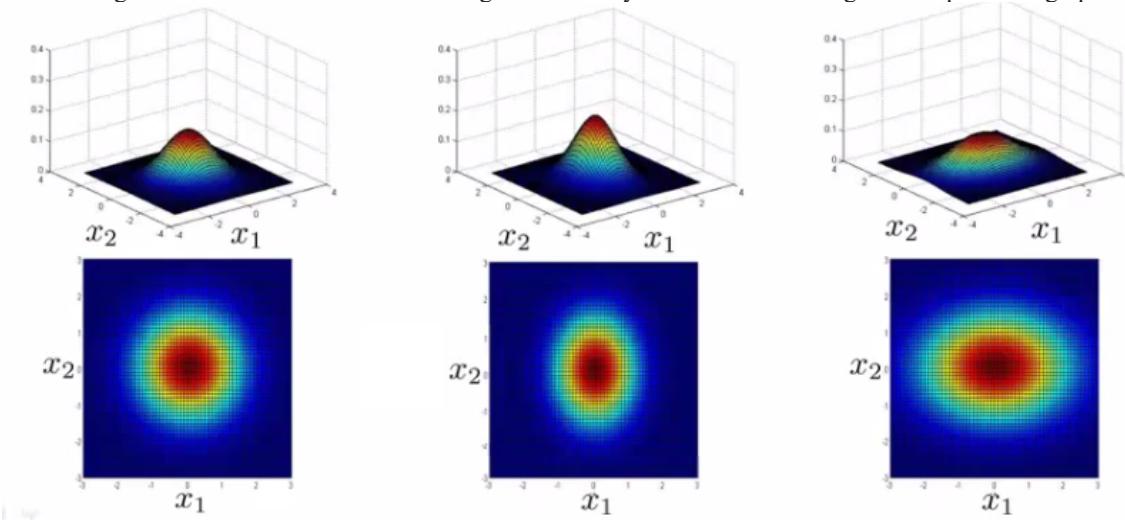
- What happens if we change Sigma?

$$\mu = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \quad \Sigma = \begin{bmatrix} 0.6 & 0 \\ 0 & 0.6 \end{bmatrix}$$

- So now we change the plot to

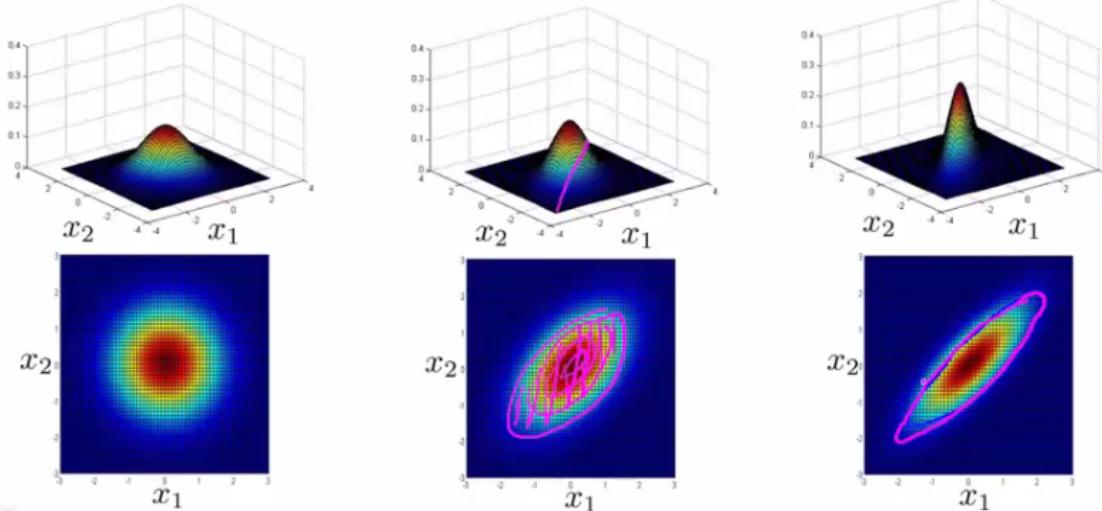


- Now the width of the bump decreases and the height increases
- If we set sigma to be different values this changes the identity matrix and we change the shape of our graph



- Using these values we can, therefore, define the shape of this to better fit the data, rather than assuming symmetry in every dimension
- One of the cool things is you can use it to model correlation between data
 - If you start to change the off-diagonal values in the covariance matrix you can control how well the various dimensions correlate

$$\mu = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \Sigma = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \quad \mu = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \Sigma = \begin{bmatrix} 1 & 0.5 \\ 0.5 & 1 \end{bmatrix} \quad \mu = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \Sigma = \begin{bmatrix} 1 & 0.8 \\ 0.8 & 1 \end{bmatrix}$$



- So we see here the final example gives a very tall thin distribution, shows a strong positive correlation
- We can also make the off-diagonal values negative to show a negative correlation
- Hopefully this shows an example of the kinds of distribution you can get by varying sigma
 - We can, of course, also move the mean (μ) which varies the peak of the distribution

Applying multivariate Gaussian distribution to anomaly detection

- Saw some examples of the kinds of distributions you can model
 - Now let's take those ideas and look at applying them to different anomaly detection algorithms
- As mentioned, multivariate Gaussian modeling uses the following equation;

$$p(x; \mu, \Sigma) = \frac{1}{(2\pi)^{\frac{n}{2}} |\Sigma|^{\frac{1}{2}}} \exp \left(-\frac{1}{2} (x - \mu)^T \Sigma^{-1} (x - \mu) \right)$$

- Which comes with the parameters μ and Σ
 - Where
 - μ - the mean (n-dimensional vector)
 - Σ - covariance matrix ([nxn] matrix)
- Parameter fitting/estimation problem
 - If you have a set of examples
 - $\{x^1, x^2, \dots, x^m\}$
 - The formula for estimating the parameters is

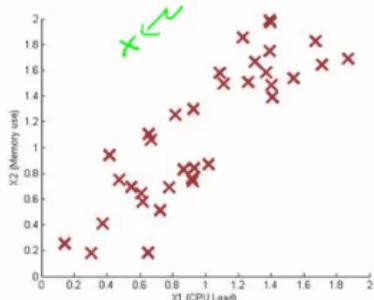
$$\mu = \frac{1}{m} \sum_{i=1}^m x^{(i)}$$

$$\Sigma = \frac{1}{m} \sum_{i=1}^m (x^{(i)} - \mu)(x^{(i)} - \mu)^T$$

- Using these two formulas you get the parameters

Anomaly detection algorithm with multivariate Gaussian distribution

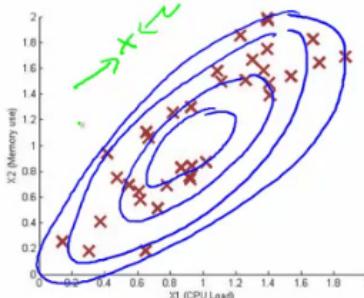
- 1) Fit model - take data set and calculate μ and Σ using the formula above
- 2) We're next given a new example (x_{test}) - see below



- For it compute $p(x)$ using the following formula for multivariate distribution

$$p(x) = \frac{1}{(2\pi)^{\frac{n}{2}} |\Sigma|^{\frac{1}{2}}} \exp \left(-\frac{1}{2}(x - \mu)^T \Sigma^{-1} (x - \mu) \right)$$

- 3) Compare the value with ϵ (threshold probability value)
 - if $p(x_{\text{test}}) < \epsilon \rightarrow$ flag this as an anomaly
 - if $p(x_{\text{test}}) \geq \epsilon \rightarrow$ this is OK
- If you fit a multivariate Gaussian model to our data we build something like this



- Which means it's likely to identify the green value as anomalous
- Finally, we should mention how multivariate Gaussian relates to our original simple Gaussian model (where each feature is looked at individually)
 - Original model corresponds to multivariate Gaussian where the Gaussians' contours are axis aligned
 - i.e. the normal Gaussian model is a special case of multivariate Gaussian distribution
 - This can be shown mathematically
 - Has this constraint that the covariance matrix sigma as ZEROs on the non-diagonal values

$$p(x; \mu, \Sigma) = \frac{1}{(2\pi)^{\frac{n}{2}} |\Sigma|^{\frac{1}{2}}} \exp \left(-\frac{1}{2}(x - \mu)^T \Sigma^{-1} (x - \mu) \right)$$

where $\Sigma = \begin{bmatrix} \sigma_1^2 & \dots & \dots \\ \vdots & \ddots & \vdots \\ \dots & \dots & \sigma_n^2 \end{bmatrix}$

- If you plug your variance values into the covariance matrix the models are actually identical

Original model vs. Multivariate Gaussian

Original Gaussian model

- Probably used more often
- There is a need to manually create features to capture anomalies where x_1 and x_2 take unusual combinations of values
 - So **need to make extra features**
 - Might not be obvious what they should be
 - This is always a risk - where you're using your own expectation of a problem to "predict" future anomalies
 - Typically, the things that catch you out aren't going to be the things you thought of
 - If you thought of them they'd probably be avoided in the first place
 - Obviously this is a bigger issue, and one which may or may not be relevant depending on your problem space

- Much **cheaper computationally**
- **Scales much better** to very large feature vectors
 - Even if $n = 100\ 000$ the original model works fine
- **Works well even with a small training set**
 - e.g. 50, 100
- Because of these factors it's used more often because it really represents a optimized but axis-symmetric specialization of the general model

Multivariate Gaussian model

- Used less frequently
- **Can capture feature correlation**
 - So no need to create extra values
- **Less computationally efficient**
 - Must compute inverse of matrix which is $[n \times n]$
 - So lots of features is bad - makes this calculation very expensive
 - So if $n = 100\ 000$ not very good
- **Needs for $m > n$**
 - i.e. number of examples must be greater than number of features
 - If this is not true then we have a singular matrix (non-invertible)
 - So should be used only in $m \gg n$
- If you find the matrix is non-invertible, could be for one of two main reasons
 - $m < n$
 - So use original simple model
 - Redundant features (i.e. linearly dependent)
 - i.e. two features that are the same
 - If this is the case you could use PCA or sanity check your data