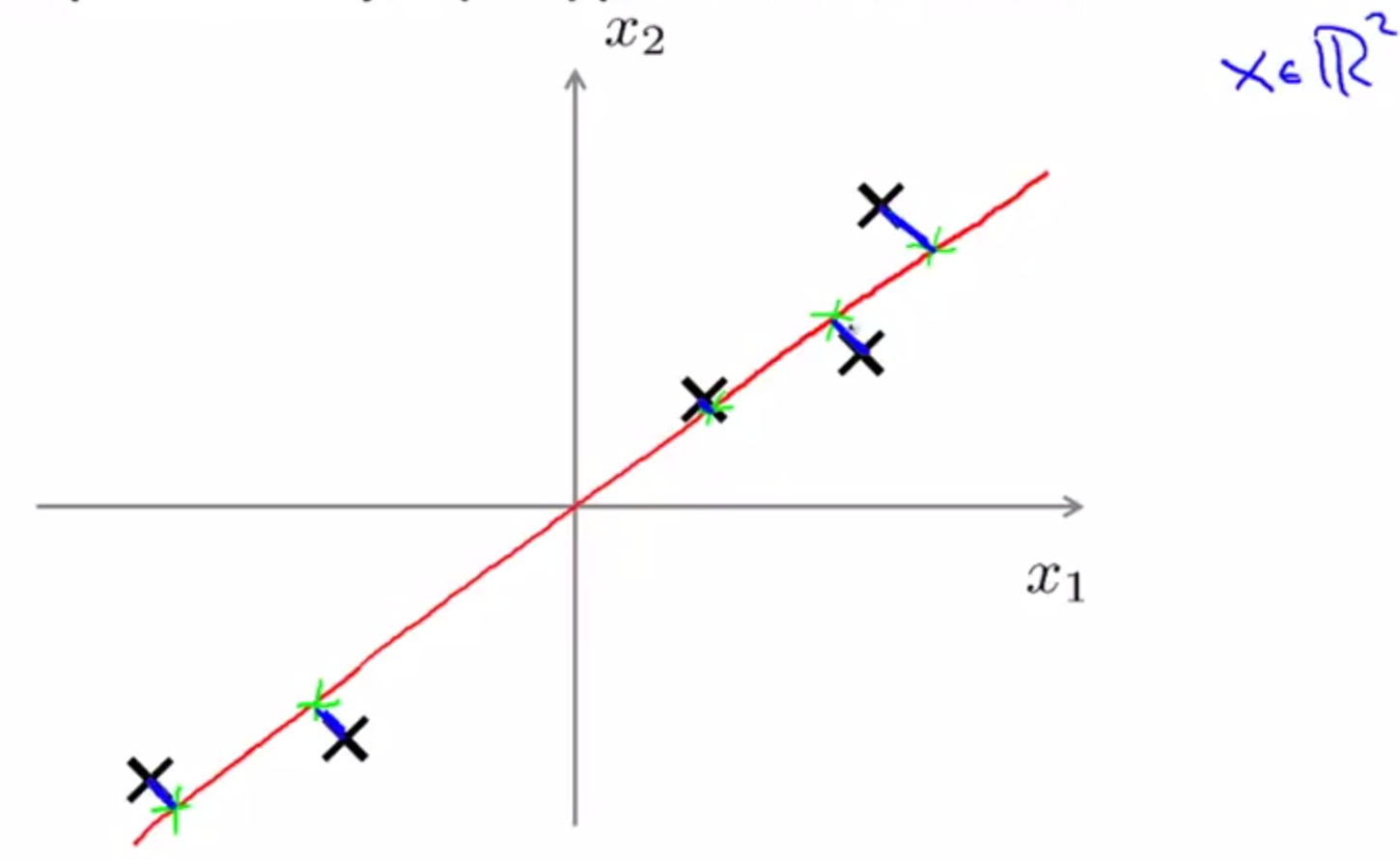
***Dimensionality Reduction***

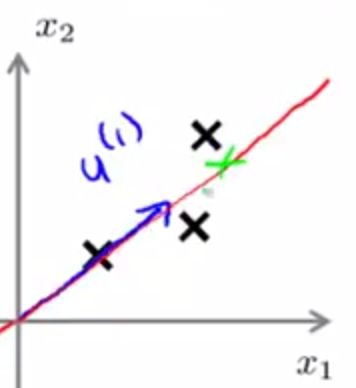
***Principal Component Analysis***

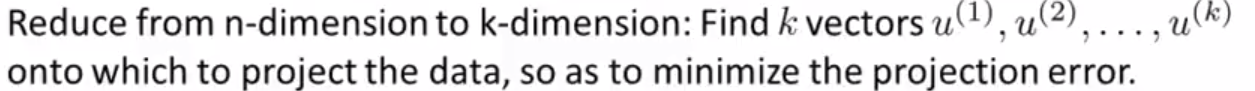
**I. PRINCIPAL COMPONENT ANALYSIS PROBLEM FORMULATION**

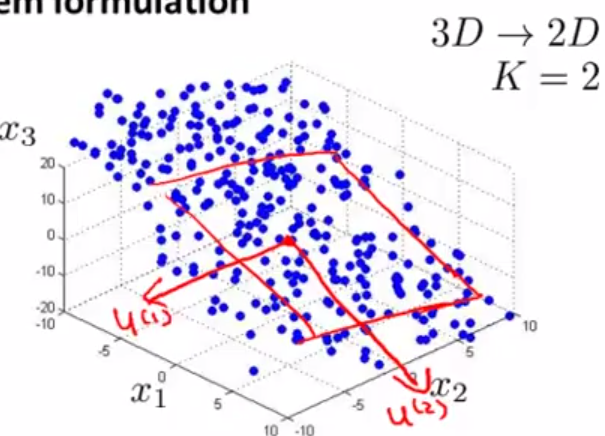
* For the problem of dimensionality reduction, by far the most commonly used algorithm is **principle components analysis** (**PCA**)
* Ex: Data set of examples x in Ɍ2 + we want to reduce the dimensions of the data from 2D to 1D (find a line onto which to project the data)



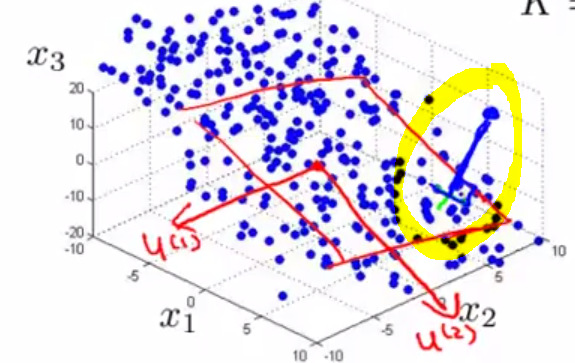
* A good line onto which to project the data might be like above b/c for the projected versions of the points (green), the distance between each point + the projected version (blue) is pretty small/short
* What PCA does is try to find a lower-dimensional surface (a line in this case) onto which to project the data so that the sum of squares (**SSE**) of these segments is minimized (**the projection error**).
* Before applying PCA, it's standard practice to first perform **mean normalization** + **feature scaling** so features x1 + x2 should have 0 mean + comparable ranges of values.
* The goal of PCA, if we want to reduce data from 2D to 1D, is to try find a direction/vector u1 in R(n) onto which to project the data to minimize the projection error even when extending out the vector



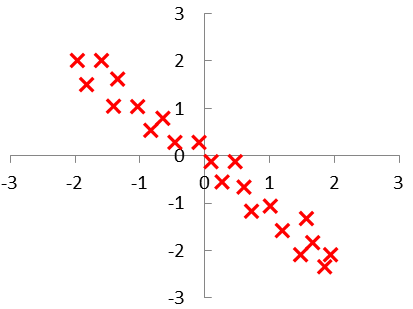
* ***Note:*** Whether PCA gives a positive or negative vector doesn't matter b/c each vector defines the same line onto which we’re projecting data.
* 
* In the more general case of n-dimensional data reduced to k-dimensions, we want to find not just a *single* vector but to find k vectors/dimensions onto which to project the data
* If we have a 3D point cloud like this, maybe we want to do is find a pair of vectors that can define a plane/2D surface onto which I am going to project my data.



* The formal definition of this: Find the **set** of vectors u(1), u(2), … u(k) + project the data onto the **linear subspace** *spanned by this set of k vectors.*
* We want to find a k-dimensional surface (a 2D plane in this case) where we can define the position of the examples in a plane using k directions to minimize the of projection distance
* In a 3D example, given a point, we would take it + project it onto a 2D surface + the projection error would be the distance between the point + where it gets projected down to my 2D surface.



* PCA tries to find the line/plane/whatever onto which to project data to try to minimize that error
* *How does PCA relate to linear regression*?
* When explaining PCA, sometimes it looks a bit like linear regression.
* Despite some cosmetic similarity, these are totally different algorithms.
* If doing linear regression, we’d be trying to predict the value of some variable Y given some info features X by fitting a straight line so as to minimize the squared error between a point + that straight line
* These error lines are the vertical distance between the point and the value predicted by the hypothesis.
* Whereas in contrast, in PCA, it tries to minimize the magnitude of these lines *drawn at an angle,* the **shortest orthogonal distances**/distance between the point x and this red line.
* This gives very different effects depending on the dataset.
* More generally, w/ linear regression, there is a distinguished variable Y we're trying to predict.
* We’re taking all the values of X and try to use that to predict Y.
* Whereas in PCA, there is no distinguished variable Y we're trying to predict + instead we have a list of features X-X(n) + all these features are treated equally
* 



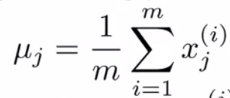
* 

**II. PRINCIPAL COMPONENT ANALYSIS ALGORITHM**

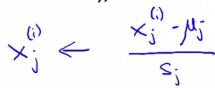
* Before applying PCA, there is a data pre-processing step which you should always do.

Given the training set of m unlabeled examples, it’s important to always perform **mean normalization**, + then, depending on your data, maybe perform **feature scaling** as well.

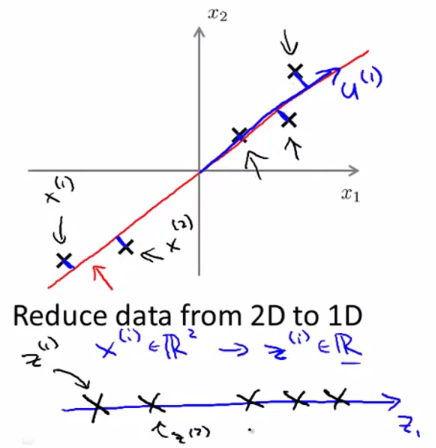
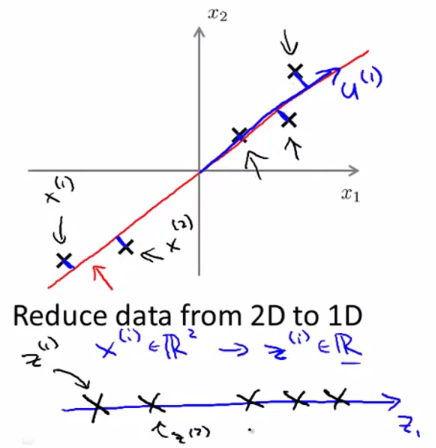
* Very similar to mean normalization + feature scaling processes for supervised learning
* Exact same procedure except now doing it to our unlabeled data, X1-Xm.
* For mean normalization: 1st compute the mean of each feature



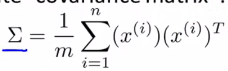
* Then replace each feature, X(j) w/ **X(j) – μ(j)** to make each feature now have exactly **μ = 0**
* Then if different features have very different scales (sf. ft. of a house vs. # of bedrooms), then also scale each feature to have a comparable range of values.
* Similar to supervised learning, take X(j)(i) (ith example of feature j), subtract **μ**, + then divide by s(j), some measure of the range of values of feature j (could be range, more commonly is SD)



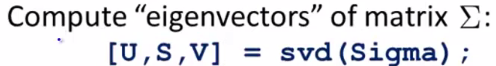
* Having done this data pre-processing, the PCA algorithm then tries to find a lower dimensional sub-space onto which to project data so as to minimize the sum of the squared projection errors
* Want to specifically find a vector, u1, which specifies a direction in 1D, or in the 2D case we want to find 2 vectors, u1 + u2, to define a surface onto which to project data.
* Reduction from 2D to 1D: Given examples x(i) in Ɍ2 🡪 want to find a set of numbers z(i) in Ɍ to represent our data

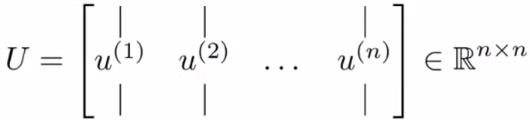
* What PCA has to come up w/ a way to compute 2 things: To compute vectors (u1, or u1 and u2) + to compute the numbers Z. S
* If reducing data from 3D/Ɍ3 to z(i) in 2D/Ɍ2, the z vectors would now be 2D 🡺 [z1 z2]
* How do you compute all of these quantities?
* There is mathematical derivation/proof to find the right values U1, U2, Z1, Z2, and so on, is very complicated and beyond the scope of the course.
* But once you've done all that math, the procedure to actually find the value of u1 you want is not that hard, even though the mathematical proof that such a value is the correct one is somewhat more involved
* Let's say we want to reduce the data to n dimensions to k dimension
* 1st, compute something called the **covariance matrix, Σ**



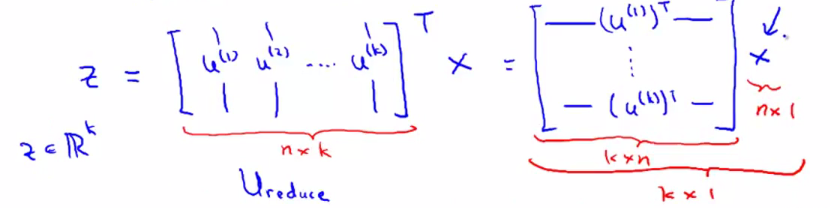
* Let's say we want to store this matrix in an octave variable called sigma.
* We’d need to do is compute the **eigenvectors** of the matrix



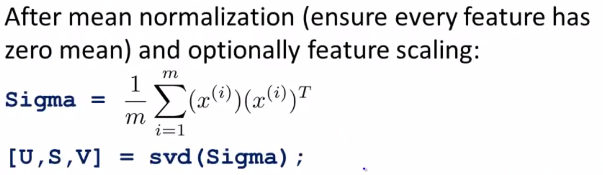
* **SVD = singular value decomposition:** much more advanced linear algebra than you actually need to know
* When sigma = a matrix, there are a few ways to compute these **eigenvectors**
* There is another octet function **eig(sigma)** which can also be used to compute the same thing
* SVD() + eig() give you the same vectors, but SVD is a little more numerically stable
* This b/c a covariance matrix always satisfies a mathematical property called **symmetric positive semi-definite**
* SVD() + eig() are different functions but when applied to a covariance matrix (which always satisfy this mathematical property), they'll always give you the same thing.
* If implementing this in a different language than Octave or MATLAB, find the numerical linear algebra library that can compute SVD
* Our covariance matrix will be an n\*n matrix, b/c x(i) is an n\*1 vector, so x(i)(t) is a 1\*n vector, so the product of these two things is going to be an n\*n matrix.
* SVD outputs 2 matrices, U, S, and V + the thing you really need out of SVD is the U matrix, also an n\*n matrix.
* If we look at the U matrix, the columns will be those vectors, u1, u2 + so on 🡺 U, will be n\*n matrix.



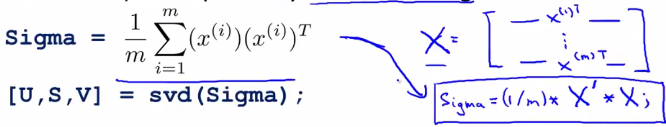
* To reduce the data from n dimensions down to k dimensions, we need to take the first k vectors (u1- u(k)) + this gives us the k directions onto which we want to project the data.
* So, from the SVD numerical linear algebra routine, we get the matrices U, S, and V + use the first k columns of U to get u1-u(k)
* Now the other thing to do: Take the original data set, X in Ɍ(n) + find a lower representation, Z in Ɍ(k)
* Take the first k columns of U to get an n\*k matrix called **U(reduce)**
* Then let Z = **U(reduce)(t)\*X**
* W/ U(reduce)(t), we end up w/ the vectors w/in U now vertical as rows.
* U(reduce)(t) is going to be k\*n + X is going to be n\*1, so the product will be k\*1.
* Therefore Z is a k-dimensional vector, which is exactly what we wanted.



* The X's here right can be examples in our training, CV, or test set
* PCA Algorithm Summary



* After pre-processing, compute the covariance matrix **Sigma**
* BTW: If your data is given as a matrix X w/ the training sets written in rows (from x1(t) down to x(n) transpose) this covariance matrix sigma actually has a nice vectorized implementation.
* You can implement in Octave:



* Run sigma = 1 / m \* X(t)\*X = the vectorized implementation of how to compute sigma.
* We then apply the SVD routine to get U, S, V, grab the first k columns of the U matrix you reduced
* Finally, we go from a feature vector X to this reduced dimension representation Z by multiplying X by the transposed reduced U matrix



* Similar to k Means, if applying PCA, apply it is w/ vectors X in Ɍn + not w/ X0 = 1
* There is a mathematical proof that this actually gives a projection of the data onto the k-dimensional subspace/surface that actually minimizes the squares projection error
* Fortunately, the PCA algorithm can be implemented in not too many lines of code (U1, U2 + so on + the Z you get out of this procedure are really the choices that minimize squared projection error)
* What PCA tries to do = find a surface/line onto which to project data so as to minimize to squared projection error.

