⇒ Week 8 : Unsupervised Unsupervised learning

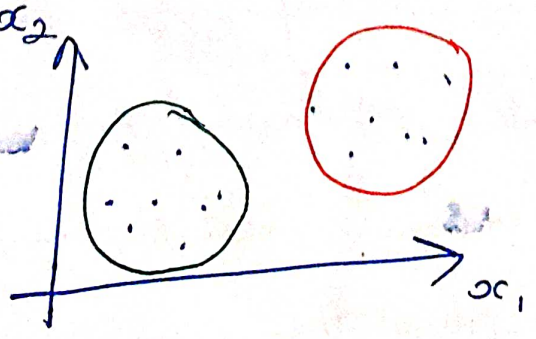
* In contrast to supervised learning, unsupervised learning uses an Unlabeled training set rather than a labeled one.

* In other words, we don't have the vector y of expected results, we only have a dataset of features where we need to find some structure.

⇒ Example of unsupervised learning : Clustering

* Clustering is the task of grouping a set of objects in such a way that objects in the same group are more similar to each other than to those in other groups.

eg:



Given: Training set : $\{x^1, x^2, x^3\}$

cluster the 2 groups

⇒ Applications of clustering

→ Market segmentation

→ Social network analysis

→ Organize computing clusters

→ Astronomical data analysis

⇒ K-Means clustering algorithm

→ Very popular clustering algorithm

→ K-Means is an iterative algs & it does 2 things:

* Cluster assignment step
* Move centroid step

→ Algorithmically,

Step-1: Randomly initialize 'K' points in the dataset called cluster centroids.

Step-2: Assign all examples into one of the 'K' groups based on which cluster centroid the example is closest to.

Step-3: Compute the averages (or means) for all the points inside each of the K clusters & then move the cluster centroids to those averages.

⇒ Explanations for the algorithm

→ 1st for loop : Cluster assignment step

* Vector c : c^i represents the cluster (or formally its index) assigned to example x^i

* Mathematically, the operation of cluster assignment is as follows :

$$c^i = \text{argmin}_k \|x^i - \mu_k\|^2$$

More simply, out of all possible values of $k \in \{1, 2, \ldots, K\}$ choose the value which minimizes $\|x^i - \mu_k\|^2$ & assign that value to c^i.

→ Rerun step-2 & 3 until we've found our clusters

⇒ Inputs:

→ K : number of clusters

→ Training set $\{x^1, x^2, \ldots, x^m\}$ → No. of training examples

* $x^i \in \mathbb{R}^n$ → No. of features

* Notice we dropped the term $x_0 = 1$ in our training set

⇒ The algorithm

Randomly initialize K cluster centroids $\mu_1, \mu_2, \ldots, \mu_K \in \mathbb{R}^n$

Repeat:

```
% Cluster assignment step
for i = 1 to m:
    c^i := index (from 1 to K) of cluster
           centroid closest to x^i

% Move centroid step
for k = 1 to K:
    μ_k := average (mean) of points assigned to
           cluster k
```

* Why squared distance?

By squaring the Euclidean distance, we make the function we are trying to minimize more sharply increasing, which in turns makes our minimization quicker.

* $\|x^i - \mu_k\|^2 = (x_1{}^i - \mu_{1(k)})^2 + \ldots + (x_n{}^i - \mu_{n(k)})^2$

Moreover, we don't have to calculate the square root too, which reduces our computation.

→ 2nd for loop: Move Centroid Step

* In this step, we move each centroid to the average of its group.

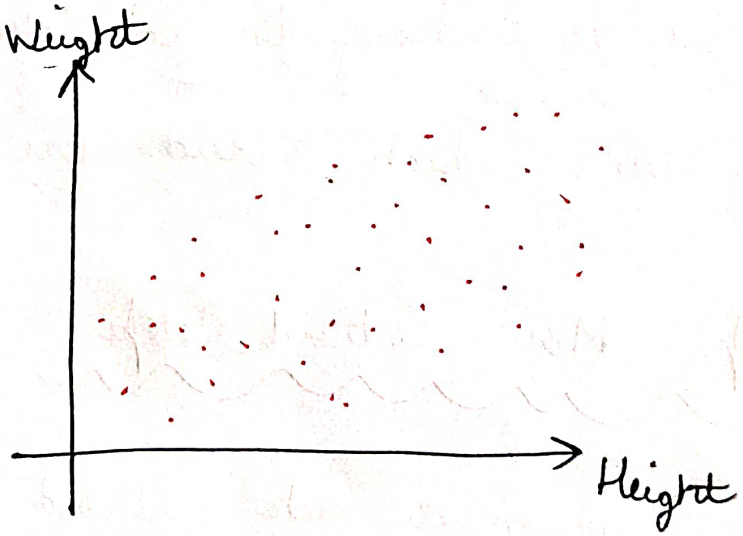
* Mathematically, for a cluster centroid k containing points $\{x^{k_1}, x^{k_2}, \ldots, x^{k_n}\}$ in its cluster, the foll. operation takes place: $\mu_k = \frac{1}{n}[x^{k_1} + x^{k_2} + \ldots + x^{k_n}]$

* If you have a cluster with 0 points assigned to it, you can randomly re-initialize that centroid to a new point.

You can also simply eliminate that cluster group.

⇒ K-means for non-separated clusters

Some datasets like the following:



(T-Shirt sizing)

have no real inner separation or natural structure. K-means can still evenly segment your data into K subsets, so can still be useful in these cases.

⇒ Optimization Objective

→ Recall some parameters which we used in our algo:

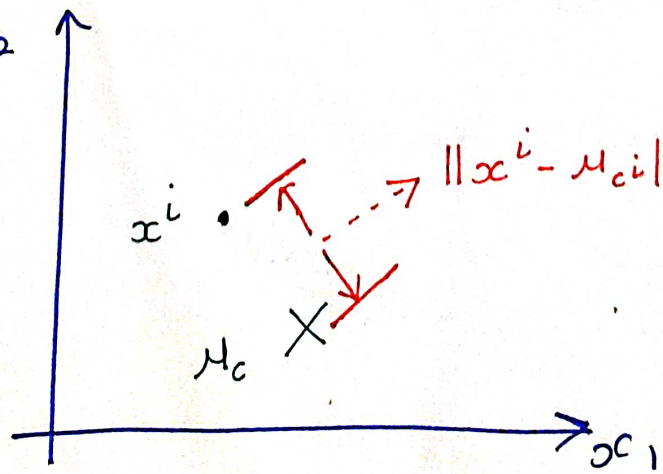
* c^i = index of cluster $(1, 2, \ldots, K)$ to which example x^i is a currently assigned
* μ_k = cluster centroid k ; $\mu_k \in \mathbb{R}^n$, $k \in [1, K]$
* μ_{c^i} = cluster centroid of cluster to which example x^i has been assigned.

eg: Suppose $x^i \to 5^{th}$ cluster then $c^i = 5$ & $\mu_{c^i} = \mu_5$

→ Using these variables, we can define our cost function:

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

eg:



$\|x^i - \mu_{c^i}\|$ = Euclidean distance b/w x^i & its cluster centroid μ_{c^i}

→ The above cost function is often called the 'distortion' of the training examples

→ Our optimization objective is as follows :

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

i.e. from all the values of set $c \in [c^1, \ldots, c^m]$ representing our clusters, & set $\mu \in [\mu_1, \ldots, \mu_K]$ representing all our centroids, we'll choose/find the values which will minimize the average of the distances of every training example to its corresponding cluster centroid.

→ In the first for loop (i.e. the cluster assignment step), our goal is to:

> Minimize J with $c^1, \ldots, c^m$ while holding $\mu_1, \ldots, \mu_K$ fixed

Intuitively, this step assigns each point to a cluster centroid closed to it, because that's what minimizes the square of distances b/w the points & their cluster centroid.

→ In the 2nd for loop (i.e. the move centroid step), our goal is to:

> Minimize J with $\mu_1, \ldots \mu_K$

Intuitively, this chooses the centroid positions which minimize J.

→ Note : The K-Means' cost function is convex in nature. So, while minimizing, it is not possible for it to sometimes increase all of a sudden. It should always descend.

⇒ Random initialization

There's one particular recommended method for randomly initializing cluster centroids :

1) Have $K < m$. That is, make sure no. of clusters is less than no. of training examples.

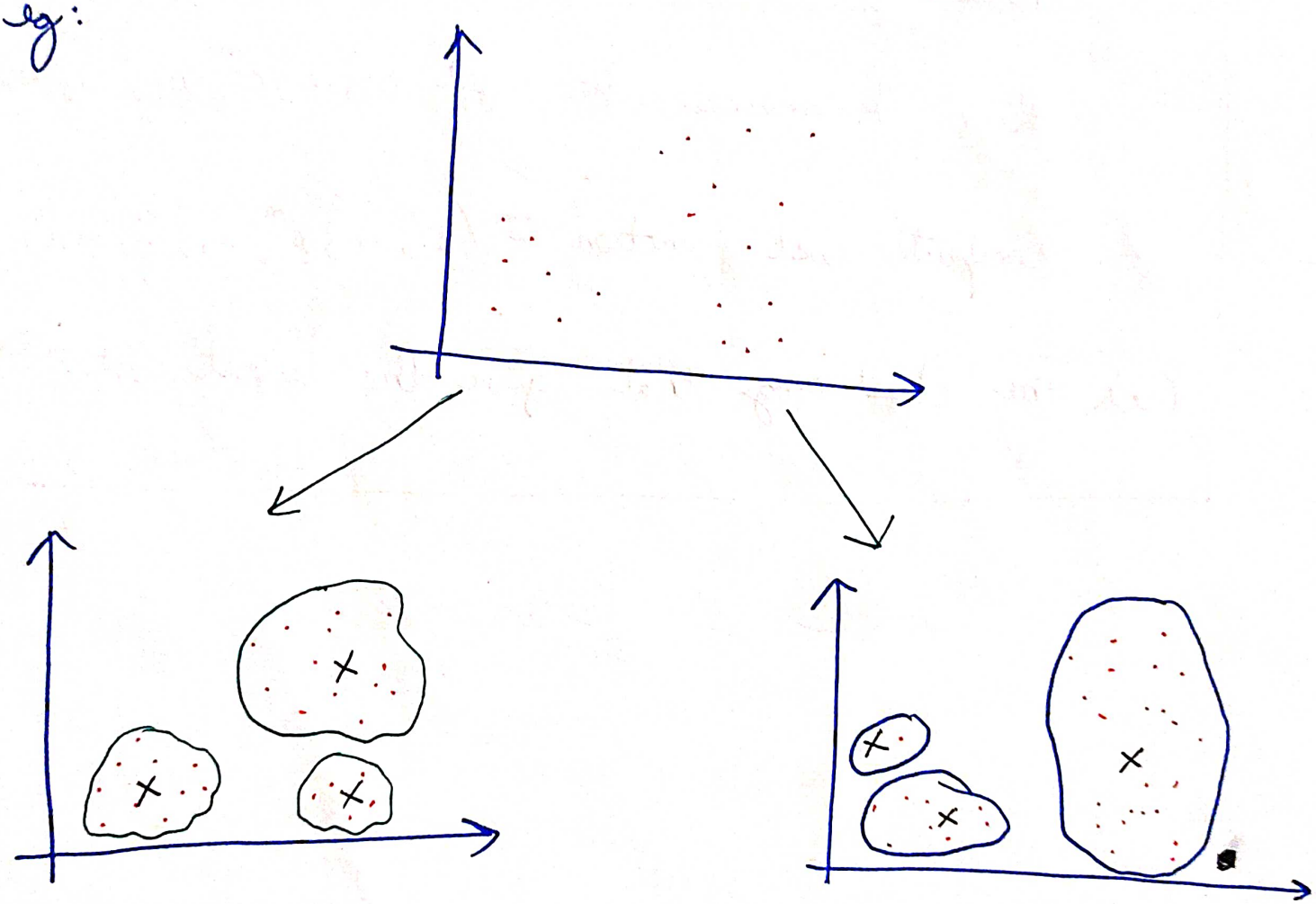
2) Randomly pick K 'unique' training examples as cluster centroids.

3) Set $\mu_1, \ldots, \mu_K$ equal to these K examples.

⇒ Local optimums (Bad clustering)

* Due to some unfortunate initialization of centroids, K-Means can sometimes result in clusters that converge to local optimums.

eg:



* To decrease a chance of this happening, we can run this algo on many different random initializations

→ In cases where $K < 10$, it is strongly recommended to run a loop of random initializations.

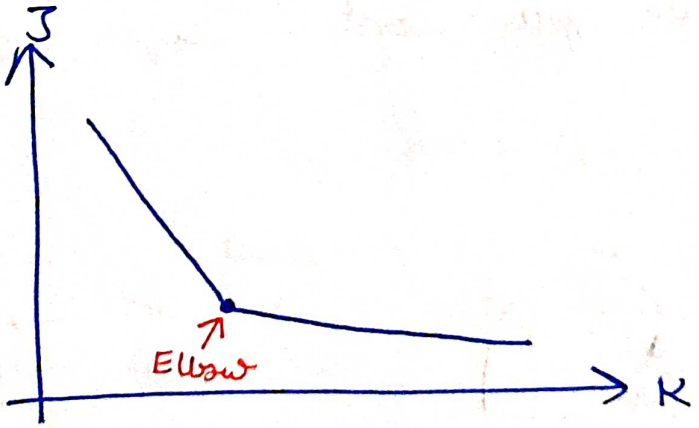
```
for i = 1 to 100:
    Randomly initialize k-means
    Run k-means to get c^1, ..., c^m, μ_1, ..., μ_K
    Compute cost function J(c^1, ..., c^m, μ_1, ..., μ_K)
Pick the clustering that gave the lowest cost.
```

⇒ Choosing the no. of clusters (K)

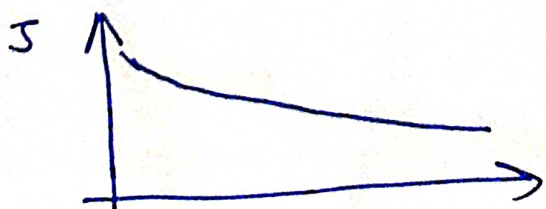
→ Choosing K can be really arbitrary & ambiguous.

→ Elbow method

* Plot the cost J & the number of clusters K
* The cost function should ~~increase as~~ reduce as we increase the no. of clusters K, & then flatten out.
* Choose K at the points where the cost function J starts to flatten out.



* However, many times the curve is very gradual with no clear elbow.

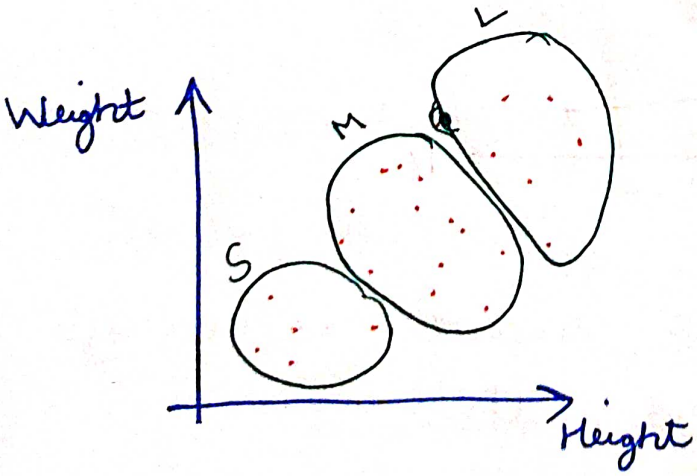


→ J will always decrease as K is increased unless it gets stuck at a bad local optimum.

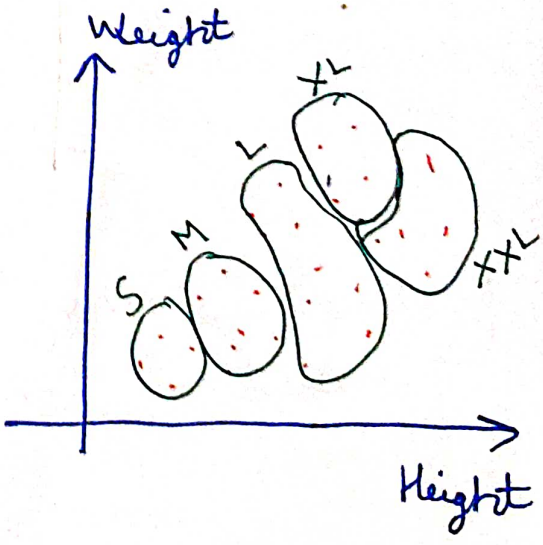
→ Another way to choose K is to observe how well it performs on a 'downstream' purpose.

In other words, choose K that proves to be most useful for some goal you're trying to achieve from using these clusters.

eg: Whether you want 3 sizes or 5 sizes for a t-shirt.



or



⇒ Dimensionality reduction

→ Another type of unsupervised learning problem.

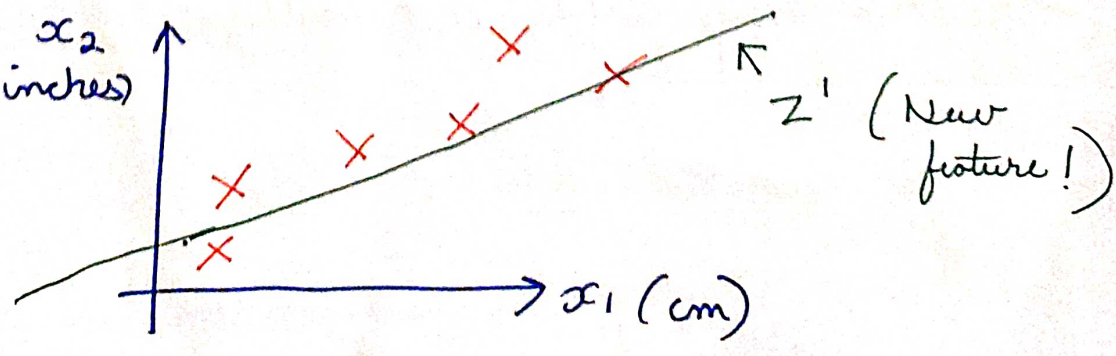
⇒ Motivation-1 : Data compression

→ We may want to reduce the dimension of our features if we have a lot of redundant data.

eg : Suppose there are 2 features x_1 (cm) & x_2 (inches). Both of them represent the same thing!

→ For reducing 2D → 1D, we find 2 highly correlated features, plot them, and make a new line that seems to describe both features accurately.

eg :



→ Doing this sort of compression :

* Reduces the total data to be stored in memory
* Speeds up our learning algorithm
* Gives a 'lossy' compression but an acceptable loss

→ Note: In dimensionality reduction we are reducing our features rather than our number of examples.

* Our variable m stays the same
* The number of features ϕ which each example carries (n), will be reduced.

→ In reality, we'd normally try & do 1000D → 100D.

→ Motivation-2: Visualization

→ It is not easy to visualise data that is more than 3 dimensions.

→ We can try to reduce the dimensions of our data to 3 or less in order to plot.

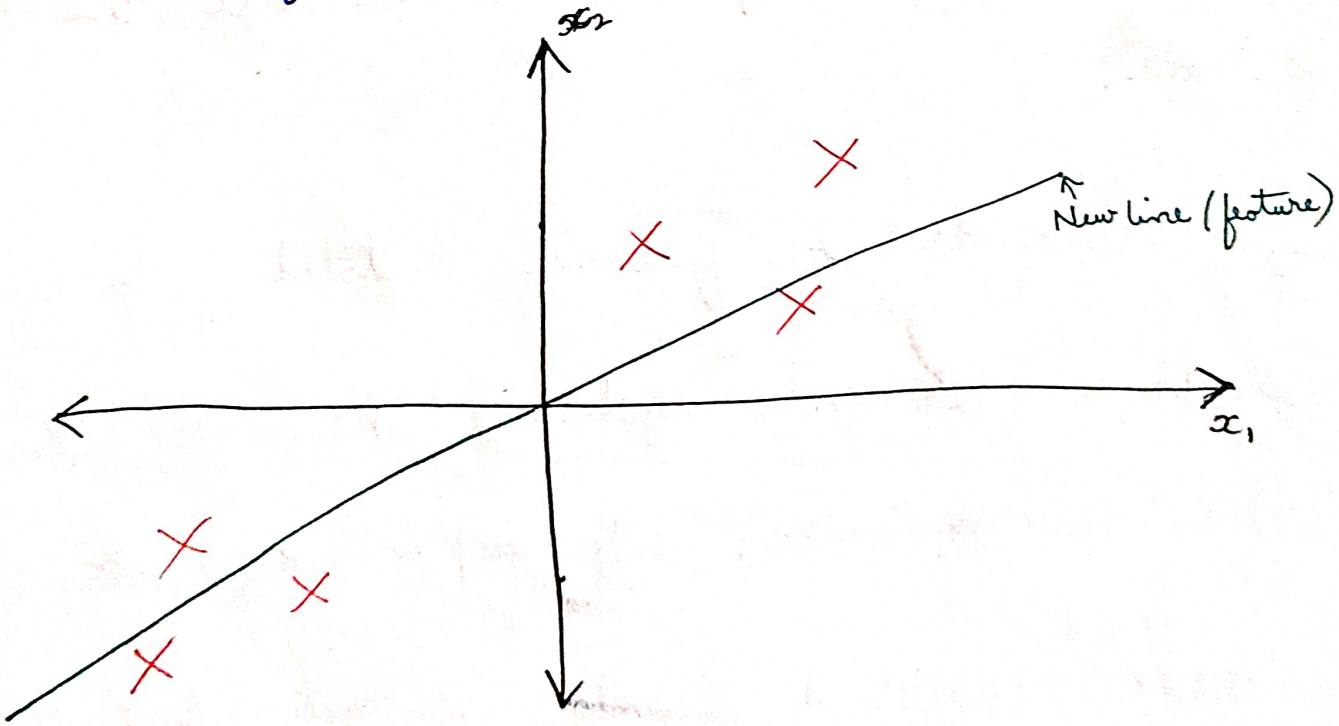
eg: For this we need suitable features z_1, z_2 (& perhaps z_3) that can effectively summarize all the other features.

eg: Hundreds of features related to a country's economic system, like GDP, per-capita GDP, Poverty index, Life expectancy, ~~etc.~~ etc. can be combined into 2 new features, like z_1 → Overall country size / economic activity & z_2 → per-person well being / economic activity

⇒ Principal Component analysis (PCA) Problem formulation

→ PCA is a really popular algorithm for dimensionality reduction.

→ Problem formulation : Given 2 features x_1 & x_2, we want to find a single line that effectively describes both features at once. We then map our old features onto this new line to get a new single feature.



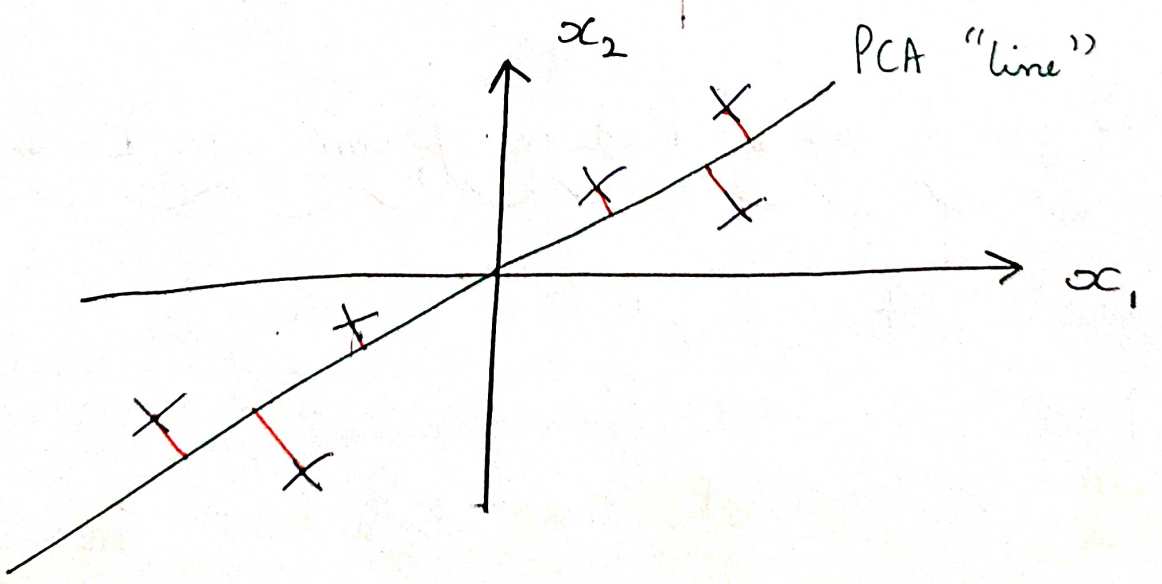
⇒ PCA problem formulation

→ Problem: Given 2 features x_1 & x_2, we want to find a single line that effectively describes both features at once.

We then map our old features onto this line to get a new feature.

→ PCA tries to find a lower dimensional surface so that the sum of squares onto that surface is minimized. (projection error)

eg:

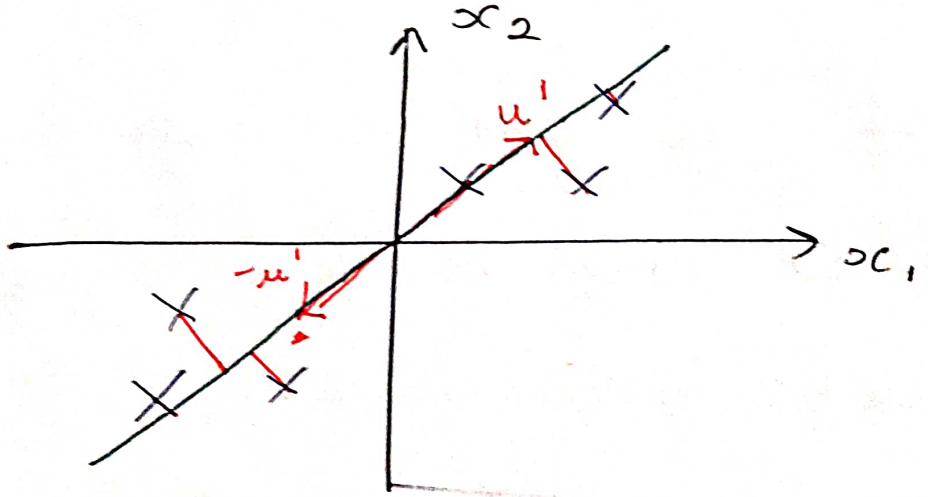


→ The red lines are sometimes called the projection error.

⇒ More formally defined,

→ Reduce from 2-D to 1-D :

Find a direction (a vector $u^1 \in \mathbb{R}^n$) onto which to project the data so as to minimize the projection error.



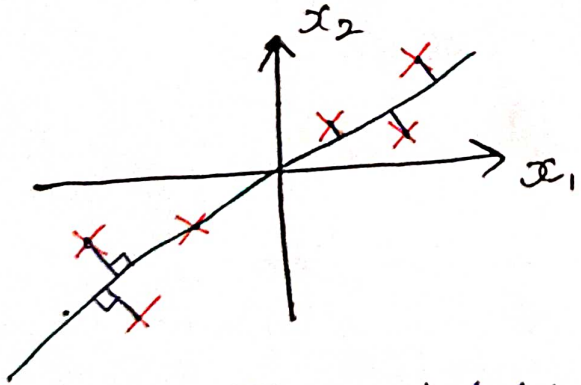
→ *More general case : Reduce from n-D to k-D

Find k vectors $u^1, u^2, \ldots, u^k$ onto which to project the data so as to minimize the projection error.

→ The formal definition or goal of PCA is to find the set of k vectors $\{u^1, u^2, \ldots, u^k\}$ s we're going to project the n points $\{x_1, x_2, \ldots, x_n\}$ onto the linear subspace spanned by those k vectors, s.t. projection error is low.

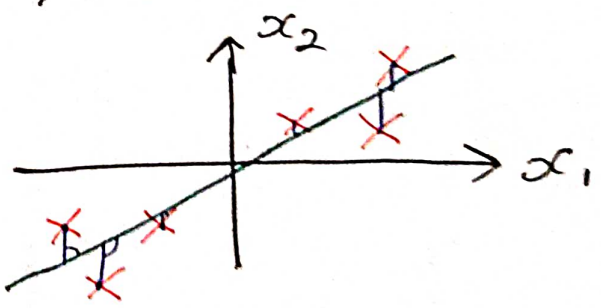
⇒ PCA is not linear regression

PCA



→ Here, we're minimizing the shortest distance, or shortest orthogonal distances, to our data points. We aren't trying to predict any result & ain't applying any weights to our inputs.

Linear regression



→ Here, we're minimizing the squared error from every point to our predictor line. These are vertical distances.

⇒ Principal component analysis : Algorithm

→ Before we can apply PCA, there is a data pre-processing step we must perform :

Training set : $x^1, x^2, \ldots, x^m$

Preprocessing : Feature Scaling / Mean normalization

* Let μ_j = mean of j^{th} feature/datapoint over all m examples

$$\Rightarrow \mu_j = \frac{\sum_{i=1}^{m} x_j^{\,i}}{m}$$

* Replace $x_j^{\,i}$ by $(x_j^{\,i} - \mu_j)$

* It is possible that a difference might exist between the scales of diff. features/datapoints.

In this case, to have a comparable range of values, we first subtract the mean of each feature from the original feature. Then we scale all the features as:

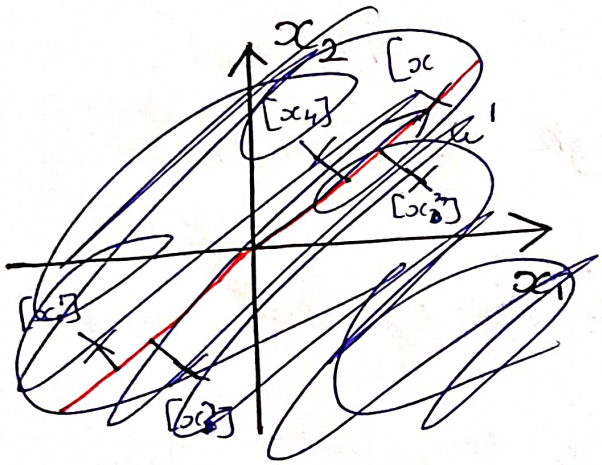
$$x_j^i = \frac{x_j^i - \mu_j}{s_j}$$

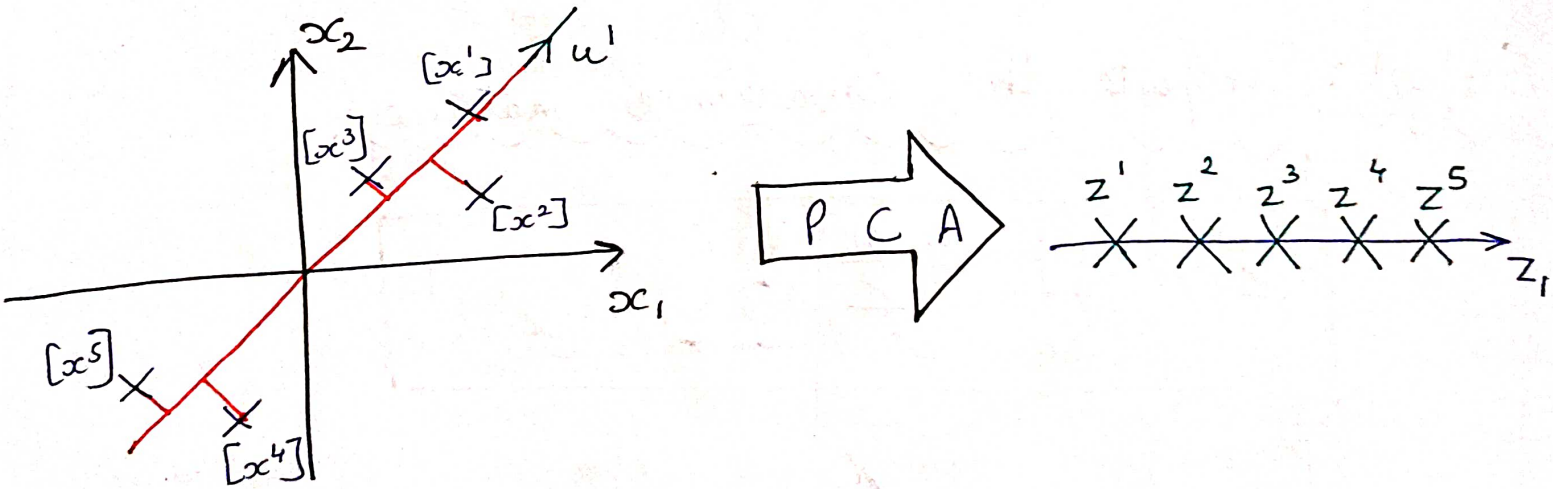
where s_j can represent: Range = (Max val − Min val)

, or Standard deviation

→ With preprocessing done, PCA finds the lower dimensional sub-space which minimizes the projection error.

→ In summary, (for 2D → 1D, say) we're doing the following:





$$x^i \in \mathbb{R}^2 \xrightarrow{PCA} z^i \in \mathbb{R}^1$$

→ So, we need to compute 2 things:

1) the u^i vectors: which "represent" the lower dimensional curve/surface/hyperplane.

2) the z^i vectors: which are the new, low dimensional mappings of our features/datapoints.

⇒ Algorithm description: Though the mathematical proof is a bit complicated (:-), we can still apply PCA by doing the foll. steps:

Step-1: Compute the "covariance matrix"

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

→ This capital sigma sign represents the covariance matrix. (Diff. from summation)

→ $x^i \rightarrow (n \times 1)$ matrix

∴ $\Sigma \rightarrow (n \times n)$ matrix

→ Step-2 : Compute "eigenvectors" of matrix Σ

$$[U, S, V] = \text{svd}(\text{sigma})$$

* svd() → singular value decomposition
* eig() → also gives eigenvector
* svd is more numerically stable than eig.
* From $[U, S, V] = \text{svd}(\text{sigma})$, we get 3 matrices U, S & V.

Turns out the columns of U are the u vectors we want!

* U is also an $[n \times n]$ matrix
* So, to reduce a system from n-D to k-D, we just take the first k vectors (columns) from U.

$$U = [u^1\ u^2 \dots u^n] \in \mathbb{R}^{n \times n} \longrightarrow U_{reduce} = [u^1\ u^2 \dots u^k] \in \mathbb{R}^{n \times k}$$

→ Step-3 : Take the first k columns of the U matrix & compute z

* After computing U_{reduce}, we can calculate the z vector in the foll. way :

$$z = (U_{reduce})^T \cdot x$$

&

$$z^{(i)} = (U_{reduce})^T \cdot x^{(i)}$$

* $z = [u^1 \; u^2 \; \dots \; u^k]^T \, x$

$$= \begin{bmatrix} (u^1)^T \\ \vdots \\ (u^k)^T \end{bmatrix} x$$

$k \times n$ $\quad$ $n \times 1$

$\therefore \boxed{z \in \mathbb{R}^{k \times 1}}$

⇒ Generalized MATLAB implementation

→ A tiny detail : Generally, we'll have m training examples having n features. We can represent them using X :

$$X = \begin{bmatrix} — (x^{(1)})^T — \\ — (x^{(2)})^T — \\ \vdots \\ — (x^{(m)})^T — \end{bmatrix}_{m \times n}$$

For computing sigma, we can use $\frac{1}{m}(X^T X)$.

→ Implementation :

```
sigma = (1/m) * X' * X    % Covariance matrix
[U, S, V] = svd(sigma)    % Projected directions
Ureduce = U(:, 1:k)       % Take first k directions
Z = X * Ureduce           % Projected data points
```

→ In summary, for a single training example

$$x = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} \in \mathbb{R}^{n \times 1}$$

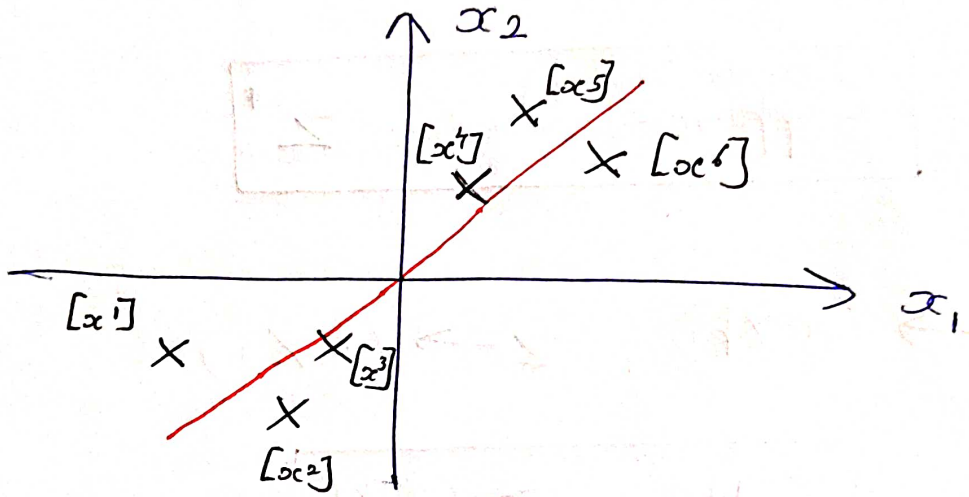
$$z = \begin{bmatrix} | & | & & | \\ u^1 & u^2 & \cdots & u^k \\ | & | & & | \end{bmatrix}^T x = \begin{bmatrix} -(u^1)^T- \\ -(u^2)^T- \\ \vdots \\ -(u^k)^T- \end{bmatrix} x$$

$$\therefore z_j = (u^j)^T \cdot x$$

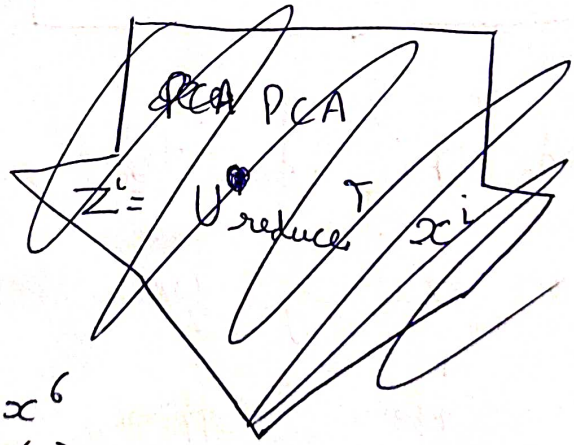
⇒ Reconstruction from compressed representation

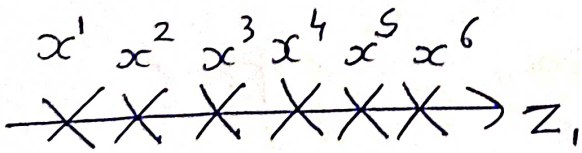
→ Is there a way, to decompress the compressed data, obtained from PCA back to the original?

→ eg:



PCA





→ For a single training example,

$$z^{(i)} = U_{reduce}^{T} x^{(i)}$$; $x^{(i)} \rightarrow (n \times 1)$, $U_{reduce}^{T} \rightarrow (k \times n)$

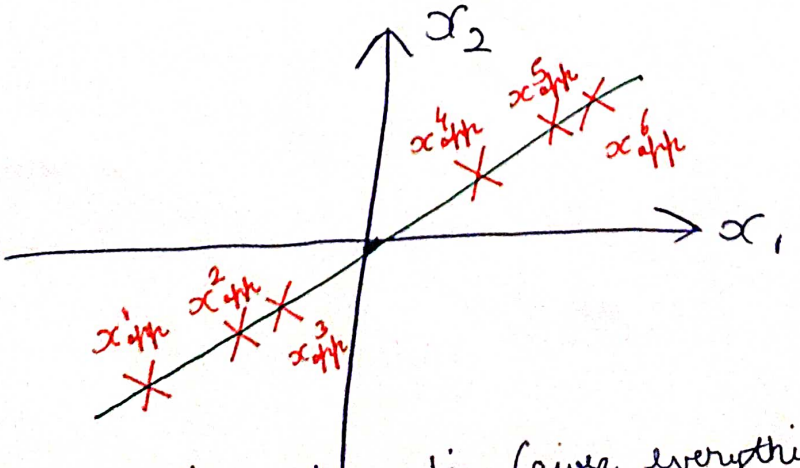
To go in the opp. dirn, we can do

$$x_{approx} = U_{reduce}\, z^{(i)}$$

→ $U_{reduce} \rightarrow (n \times k)$, $z \rightarrow (k \times 1)$.

∴ $x_{approx} \rightarrow (n \times 1)$

→ We can extend this idea to multiple training examples! For the eg. given on prev. page, we can calculate $x_{approx}^{1}, x_{approx}^{2}, \ldots, x_{approx}^{6}$ & plot it:



We lose some of the information (since everything is projected onto the line) but it is now projected onto the 2-D space

→ U matrix has a special property that it is a unitary matrix

$$\Rightarrow \boxed{U^{-1} = U^{*}}$$ where '*' means "conjugate transpose"

→ Since we're dealing with real numbers, this is equivalent to:

$$\boxed{U^{-1} = U^{T}}$$

So, we could use the inverse as well, but it would be a waste of energy & compute cycles.

⇒ Choosing the no. of principal components (k)

→ To chose a suitable val. of k, think about how PCA works:

* It tries to minimize average squared projection error: $\frac{1}{m}\sum_{i=1}^{m} \|x^i - x^i_{approx}\|^2$

* We can also calculate the total variation in the data as: $\frac{1}{m}\sum_{i=1}^{m} \|x^i\|^2$

This basically calculates how far our datapoints are from the origin.

→ When chosing k, it's typically to use a ratio between average squared projection error & total variation with data:

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

→ Algorithm for choosing k

Try PCA with k = 1

Compute U_{reduce}, $\{z^1, z^2, \ldots, z^m\}$, $\{x^1_{approx}, \ldots, x^m_{approx}\}$

Compute the ratio & check if it is < 0.01

* If ratio > 0.01, then increment k & try again.

→ This procedure is actually horribly inefficient!

We will call SVD as:

$$[U, S, V] = svd(Sigma)$$

This returns a matrix S which is is a $(n \times n)$ matrix & structured in following way:

$$S = \begin{bmatrix} S_{11} & 0 & \cdots & 0 \\ 0 & S_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & S_{nn} \end{bmatrix}_{n \times n}$$

→ Diagonal matrix

→ We typically want this ratio to be < 0.01 or $< 1\%$ (when expressed in percentages).

* In other words, the squared projection error divided by the total variation should be less than one percent, so that "99% of the variance is retained".

* ~~Simple Bla~~

~~Whole~~

* Simply said mean: If you project your points on a surface $\in \mathbb{R}^k$ (s.t. ~~its~~ the ratio < 0.01 comes out), you will reconstruct the points with 99% accuracy. This is what is meant by "99% variance is retained".

→ So, now we have to ensure that:

$$1 - \left(\frac{\sum_{i=1}^{k} S_{ii}}{\sum_{i=1}^{n} S_{ii}} \right) < 0.01$$

→ Now, this is extremely efficient !! Why?

2 reasons:

1) Now, we don't need to evaluate SVD over & over for diff. values of k. We'll compute S once & keep using the rows from S corresponding to the value of k in the loop.

2) Since, we'll only proceed to calculate U, Z & x after the ratio is < 0.01, we don't have to calculate these as well for every value of k.

Updated algo :

$[U, S, V] = \text{svd}(\text{Sigma})$

Pick smallest value of k for which :

$$\frac{\sum_{i=1}^{k} S_{ii}}{\sum_{i=1}^{m}} \geq 0.99$$

(99% of variance retained)

Compute PCA for the selected vd. of k

⇒ Advice for applying PCA

⇒ Supervised learning speedup

→ The most common use of PCA is to speed up supervised learning.

→ Given a training set with a large no. of features (eg: $x^1, \ldots, x^m \in \mathbb{R}^{10000}$) we can use PCA to reduce the no. of features in each example of the training set (eg: $z^1, \ldots, z^m \in \mathbb{R}^{1000}$)

→ Applying PCA to map $x^i \rightarrow z^i$ should only done on the training set:

* ~~Parameters~~ The matrix U is "learned" through PCA.
* So, map z^i to your cross validation & test sets & ONLY after it (the matrix U) is defined on the training set.

→ Realistically, it is completely fine to reduce data dimensionality by a factor of 5-10 without a major hit to the learning (or classification) algorithm.

⇒ Applications of PCA

→ Compression:

* Reduce memory/disk needed to store data
* Speed up learning algorithm

→ Visualization: As we can plot only in 2D & 3D, we can use $k=2$ or $k=3$ to visualize larger dimensional datasets. PCA is often used in this scenario.

* From sc7 pdf: The PCA projection can be thought as a rotation that selects the view that maximizes the spread of the data, which often corresponds to the best view.

⇒ MISUSE of PCA : To prevent overfitting

→ A general misconception is that reducing the dimension of dataset by applying PCA can prevent ~~overfitting~~ overfitting as "fewer features, less likely to overfitting".

→ This MIGHT work but it isn't a good way to address overfitting. Use regularization instead. It'll work way better!

→ PCA doesn't consider the values of the outcomes y

* It is very possible that PCA throws away some useful data which might have led to better results.

* Regularization does a much better job as it keeps in mind, the effect of outcome values while preventing overfitting.

(4)

⇒ PCA is sometimes used where it shouldn't be

~~Before creating~~

→ Don't assume that you absolutely need to do PCA!

* Try out your full machine learning algorithm/pipeline/workflow without PCA first. Then use PCA if you find that you need it.

* Of course, if you have a dataset requiring more disk space than available, then you have to do it!