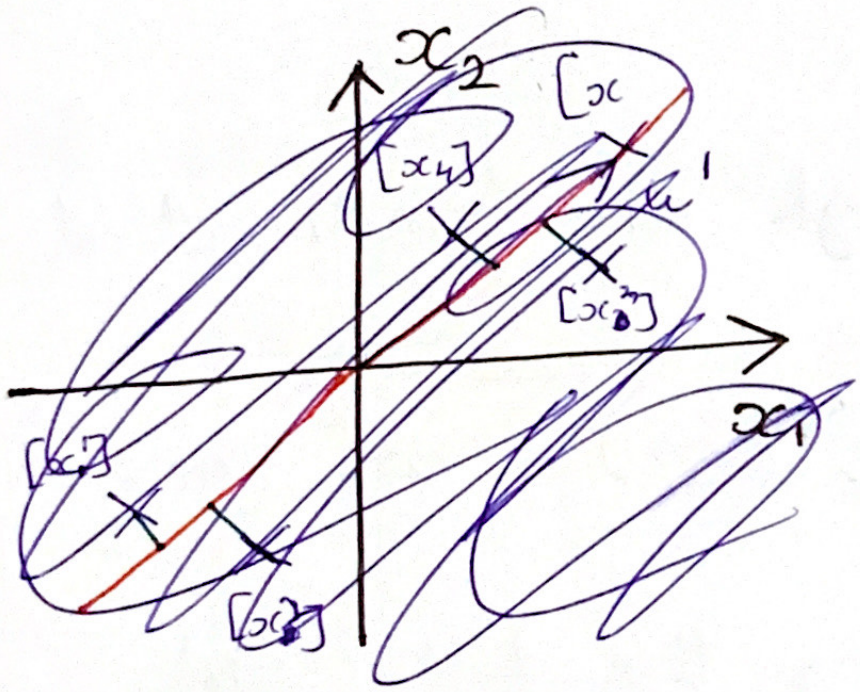
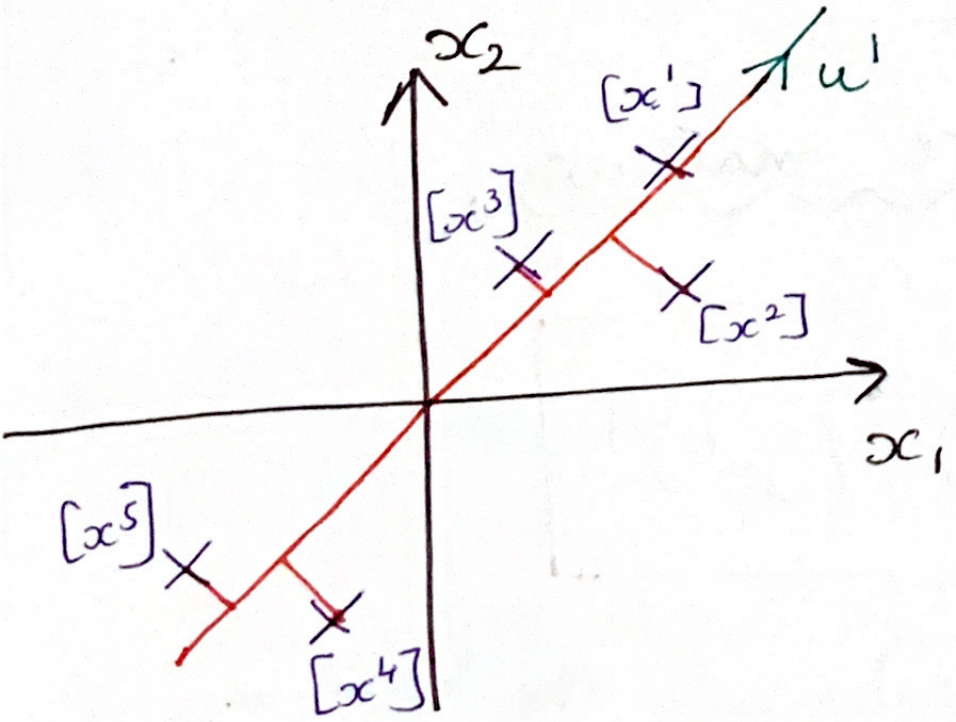
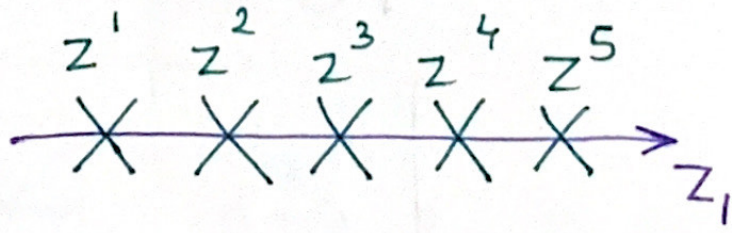
→ With preprocessing done, PCA finds the lower dimensional sub-space which minimizes the projection error.

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





PCA



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

→ 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

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

* svd() → singular value decomposition
* eig() → also gives eigenvector
* ~~s~~ svd is more numerically stable than eig.
* From $[U, S, V] = svd(sigma)$, 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.

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

→ 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 :

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

&

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

*

$$z = \begin{bmatrix} u^1 & u^2 & \cdots & u^k \end{bmatrix}^T x$$

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

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

⇒ 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

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

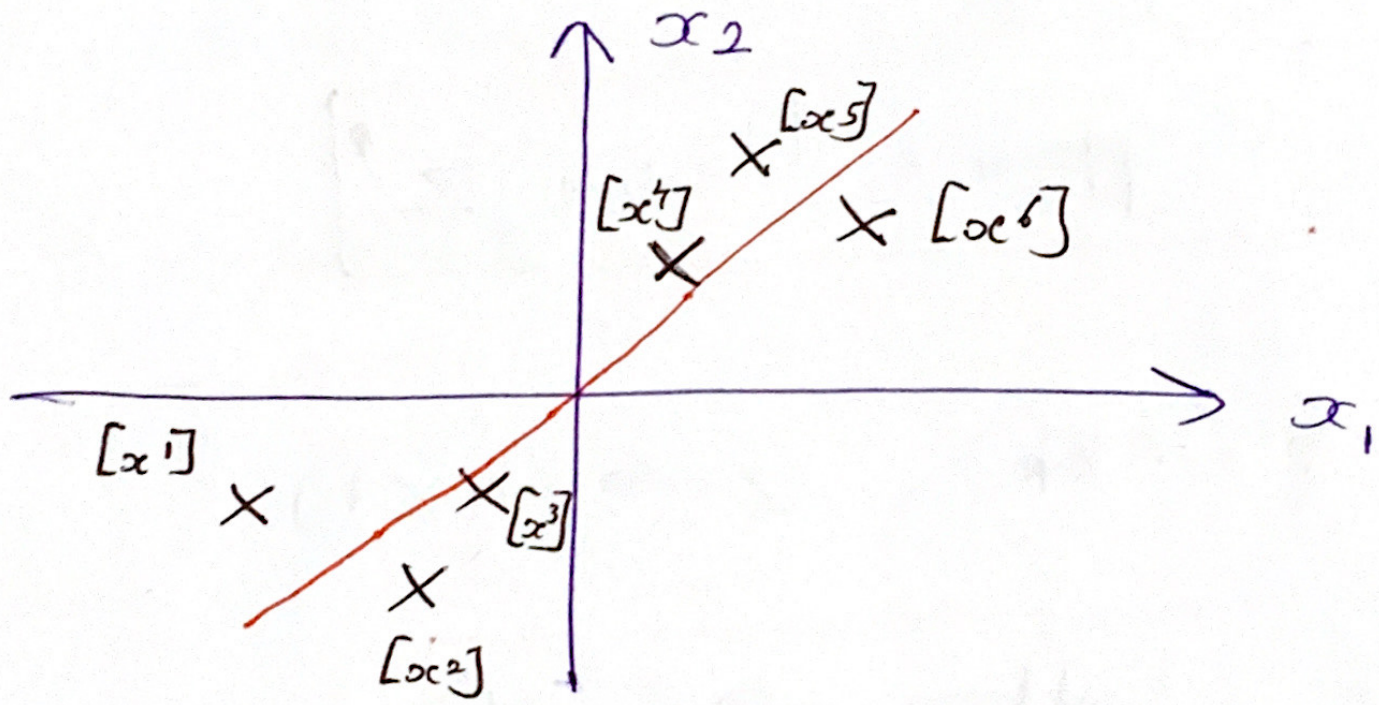
$$z = \begin{bmatrix} | & | & & | \\ u^1 & u^2 & \cdots & u^k \\ | & | & & | \end{bmatrix}^T x = \begin{bmatrix} -(u^1)^T- \\ -(u^2)^T- \\ \vdots \\ -(u^k)^T- \end{bmatrix} x$$

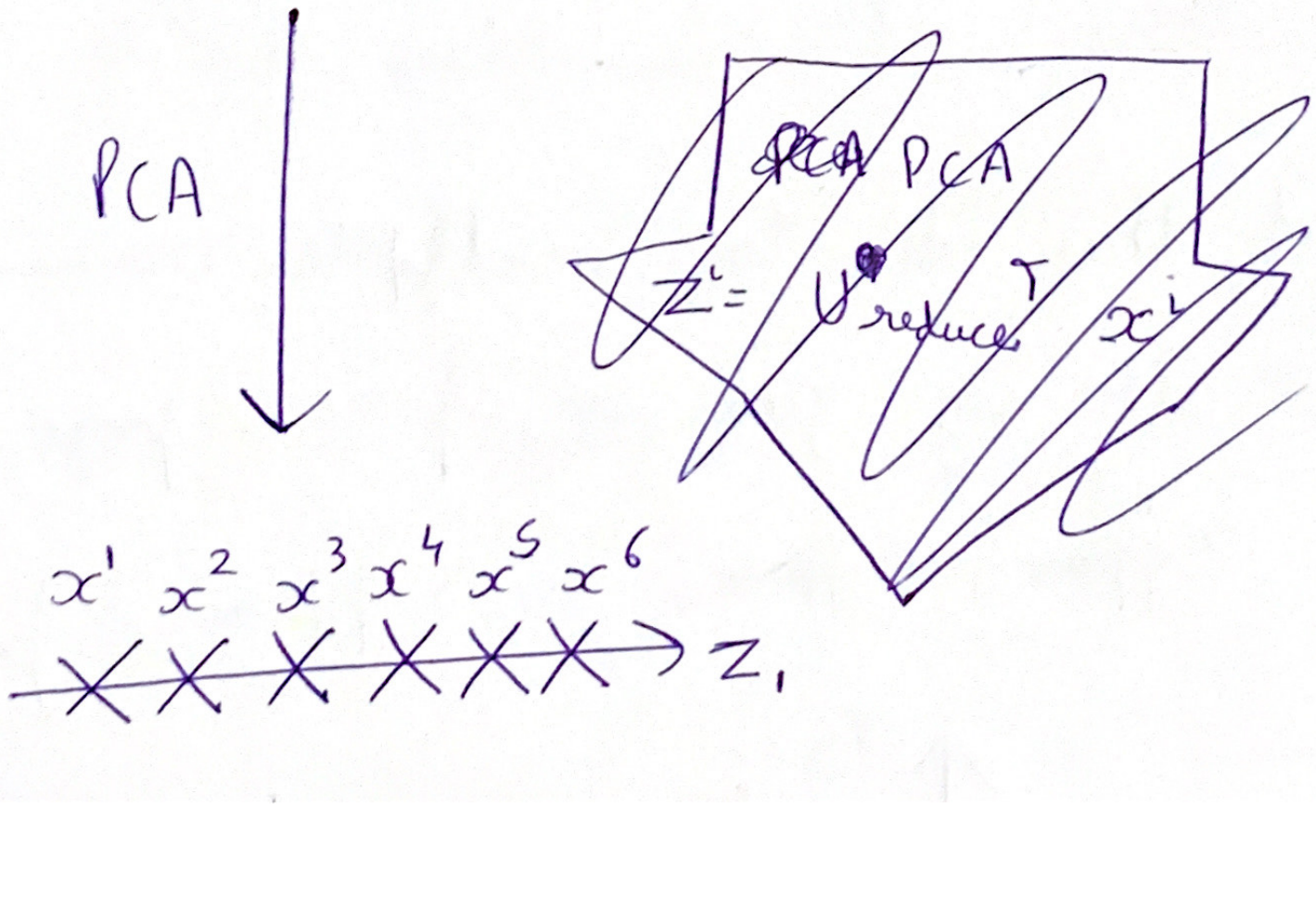
$$\therefore \boxed{z_j = (u^j)^T \cdot x}$$

⇒ Reconstruction from compressed representation

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

→ eg:





→ For a single training example,

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

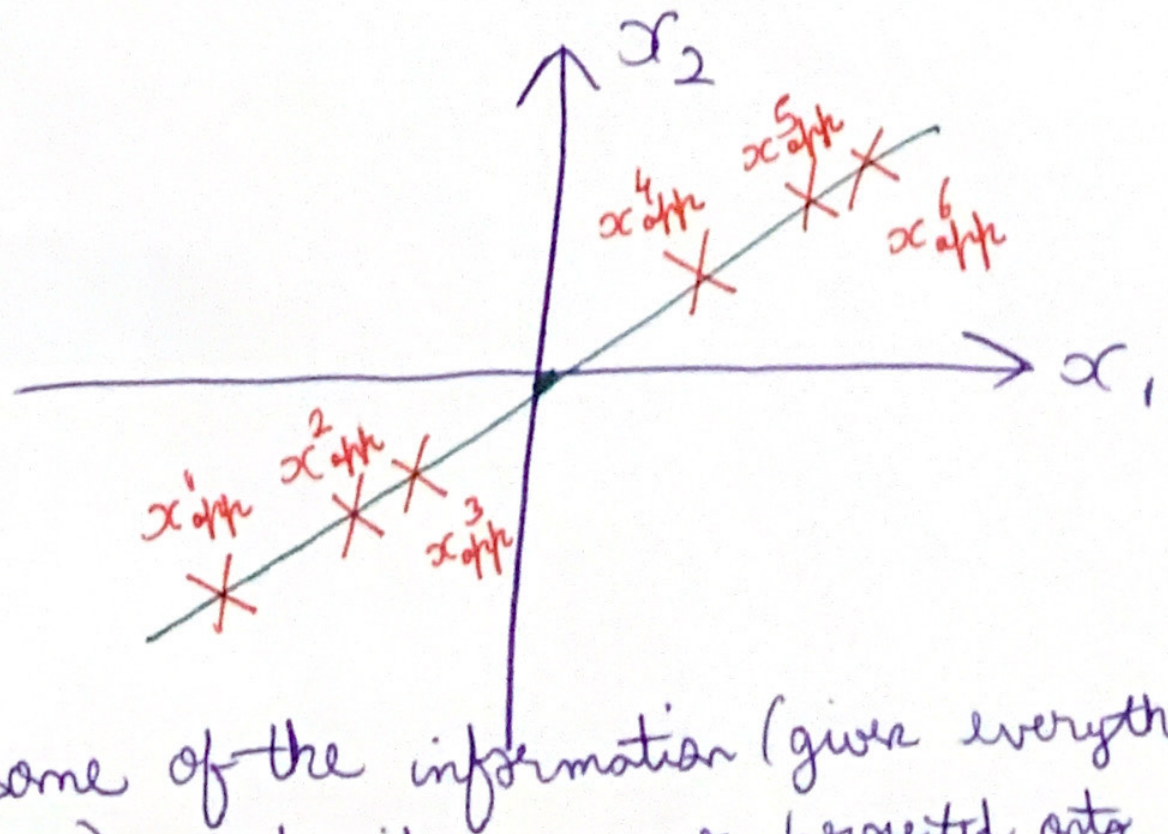
To go in the opp. dir^{n}, we can do

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

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

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

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



We lose some of the information (since everything is projected onto the 2-D space

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

⇒ $\boxed{U^{-1} = U^{\star}}$ where '$\star$' 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 choosing 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 & and 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 squred projection error divided by the total variation should be less than one percent, so that "99 % of the variance is retained".

~~* Simple theorem~~

~~Proof~~

* Simply said: If you project your points on a surface $\in \mathbb{R}^k$ (s.t. 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:

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

→ 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, ✓] = svd (Sigma)

Pick smallest value of k for which :

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

(99 % of variance retained)

Compute ~~Reduced~~ PCA for the selected val. 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 ($x^1, \ldots, x^m \in \mathbb{R}^{10000}$) we can use PCA to reduce the no. of features in each example of the training set ($z^1, \ldots, z^m \in \mathbb{R}^{1000}$)

→ Applying PCA to map $x^i \to z^i$ 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 commonly done to reduce data dimensionality by a factor of 5-10 without a major hit to the [illegible] learning (or classification) algorithm.

7 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

⇒ 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 4

* 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 !