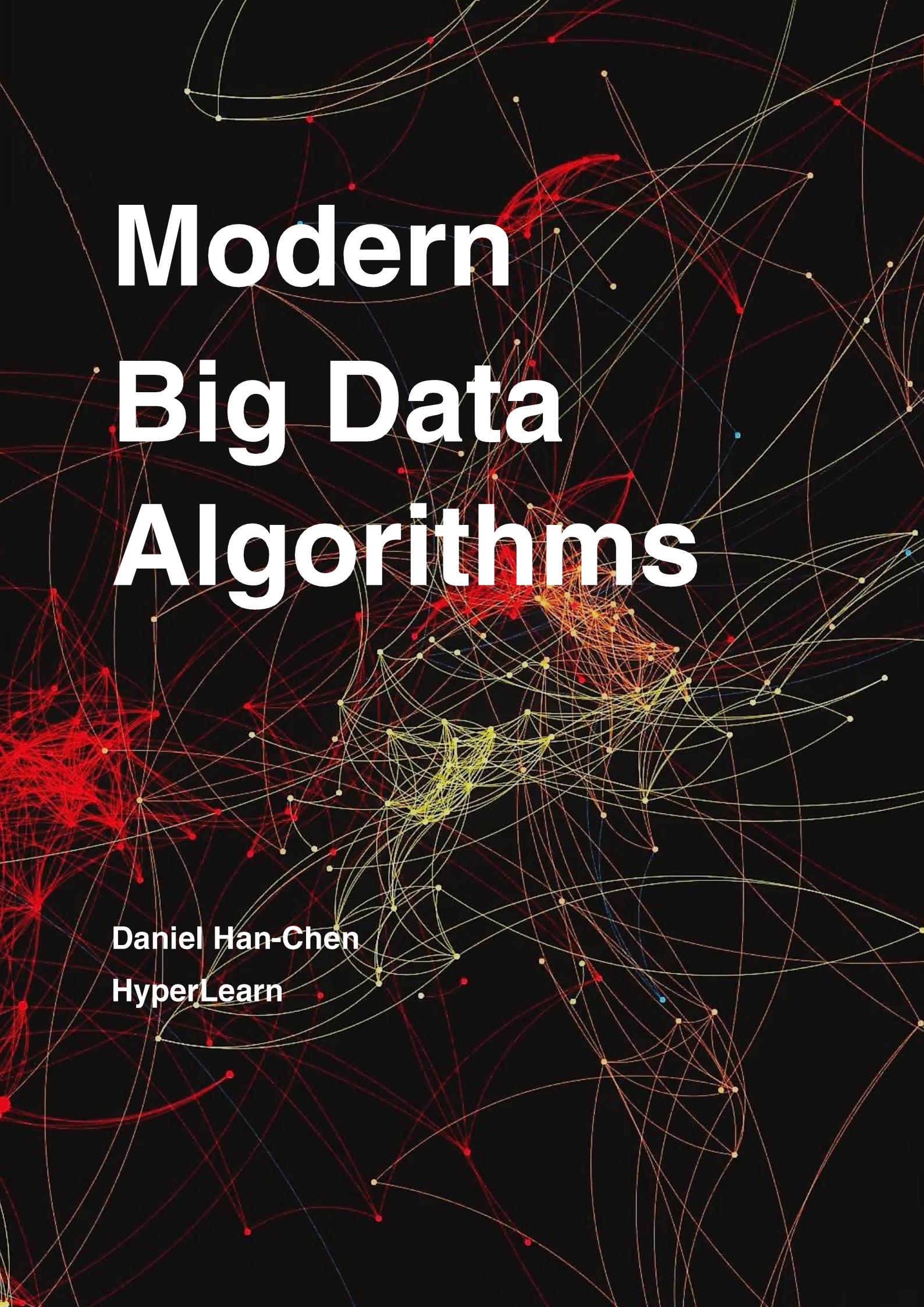


Daniel Han-Chen
HyperLearn

Welcome!

As we enter the first quarter of the 21st century, datasets are getting larger and wider.

Running primitive and slow algorithms will cause headaches, productivity and economic losses. By optimising algorithms used in stock market predictions, climate change modelling, artificial intelligence and cancer research, the world can benefit dramatically from faster and more accurate numerical methods.

Modern Big Data Algorithms is a comprehensive collection of Faster Machine Learning techniques. Many algorithms are at github.com/danielhanchen/hyperlearn

I thank **Aleksandar Ignjatovic** (UNSW) for allowing to write this in COMP4121!

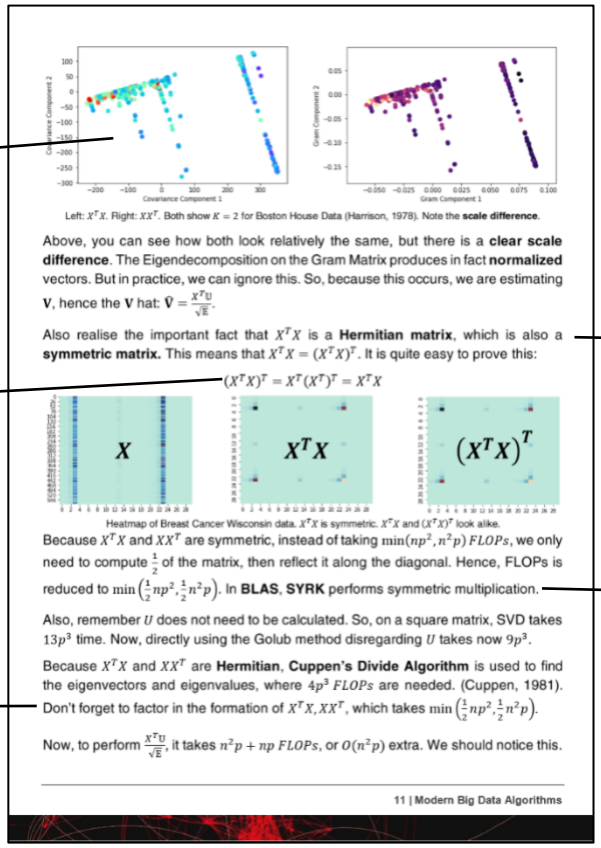
Graphics to showcase algorithms

Algorithmic details are included

FLOPs, time complexity analysis

Linear Algebra concepts

LAPACK, BLAS functions



Left: X^TX. Right: XX^T. Both show $K = 2$ for Boston House Data (Harrison, 1978). Note the **scale difference**.

Above, you can see how both look relatively the same, but there is a **clear scale difference**. The Eigendecomposition on the Gram Matrix produces in fact **normalized** vectors. But in practice, we can ignore this. So, because this occurs, we are estimating $\mathbf{V}$, hence the $\mathbf{V}$ hat: $\hat{\mathbf{V}} = \frac{X^T U}{\sqrt{E}}$.

Also realise the important fact that X^TX is a **Hermitian matrix**, which is also a **symmetric matrix.** This means that $X^TX = (X^TX)^T$. It is quite easy to prove this:

$$(X^TX)^T = X^T(X^T)^T = X^TX$$

X X^TX $(X^TX)^T$

Heatmap of Breast Cancer Wisconsin data. X^TX is symmetric. X^TX and $(X^TX)^T$ look alike.

Because X^TX and XX^T are symmetric, instead of taking $\min(np^2, n^2p)$ $FLOPs$, we only need to compute $\frac{1}{2}$ of the matrix, then reflect it along the diagonal. Hence, FLOPs is reduced to $\min\left(\frac{1}{2}np^2, \frac{1}{2}n^2p\right)$. In **BLAS**, **SYRK** performs symmetric multiplication.

Also, remember U does not need to be calculated. So, on a square matrix, SVD takes $13p^3$ time. Now, directly using the Golub method disregarding U takes now $9p^3$.

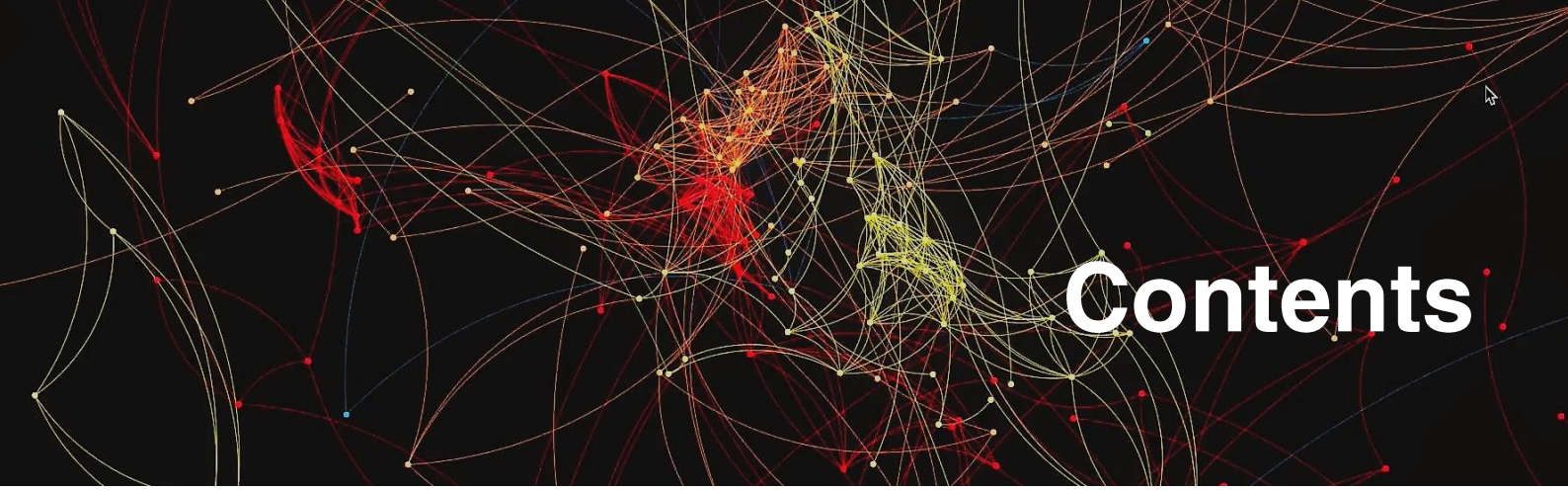
Because X^TX and XX^T are **Hermitian**, **Cuppen's Divide Algorithm** is used to find the eigenvectors and eigenvalues, where $4p^3$ $FLOPs$ are needed. (Cuppen, 1981). Don't forget to factor in the formation of X^TX, XX^T, which takes $\min\left(\frac{1}{2}np^2, \frac{1}{2}n^2p\right)$.

Now, to perform $\frac{X^TU}{\sqrt{E}}$, it takes $n^2p + np$ $FLOPs$, or $O(n^2p)$ extra. We should notice this.

11 | Modern Big Data Algorithms

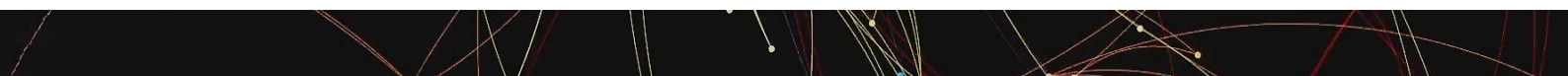
Modern Big Data Algorithms
Daniel Han-Chen
HyperLearn
University of New South Wales

danielhanchen@gmail.com
reddit.com/user/danielhanchen
linkedin.com/in/danielhanchen



Contents

Novel new algorithms are **pink**. Revamped algorithms are **green**. means it's implemented online as well on **github.com/danielhanchen/hyperlearn**. **Blue** means it will be published later, and **Dark Blue** means it's a novel new algorithm.



1 Singular Value Decomposition

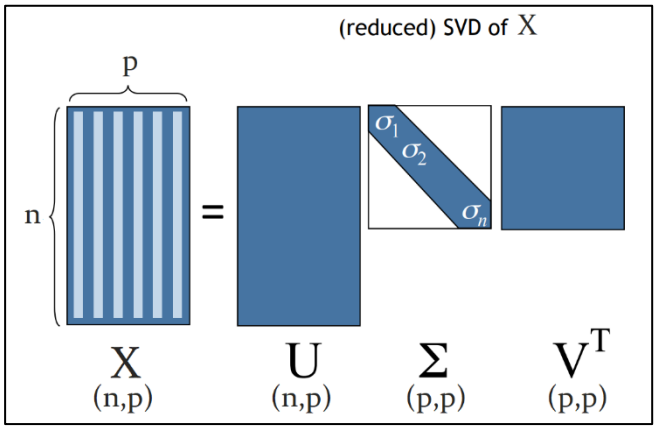
- *I have a 100MB image. I'm uploading it, but it says I need to wait 10 minutes.*
- *I'm trying to understand a dataset, but I can't graph all 1,000 columns on 1 graph.*
- *Why is SVD slow? I want the top K components containing 90% of information.*
- *How can I reconstruct images or data that have missing pixels or information?*

Singular Value Decomposition is a linear algebra algorithm that can solve these 4 problems! It can be applied to any numerical dataset (be it images, tables, signals). It reduces a dataset into 3 components: U, Σ, V^T. (Strang, 2016).

The **SVD** is a critical algorithm in Linear Algebra, as it used in Big Data Analysis (PCA, LSI), Image Compression, Physics, Genomics, Finance and Artificial Intelligence.

SVD on "Pillars of Creation" by the Hubble Space Telescope. SVD is used to remove unimportant features in images. From left: original image, 50%, 99% compression. (HST 1995)

SVD factors a matrix X into $U\Sigma V^T$. U is a rotation matrix, and is **orthogonal**. This means $U^T U = I$. Σ is a stretching matrix, and is diagonal. V^T is also a rotation matrix, and is also orthogonal, so $V^T V = I$. An important realisation is when applying **SVD** on the transpose of the data matrix X. (Sorkine-Hornung, 2016)



$$X = U\Sigma V^T$$

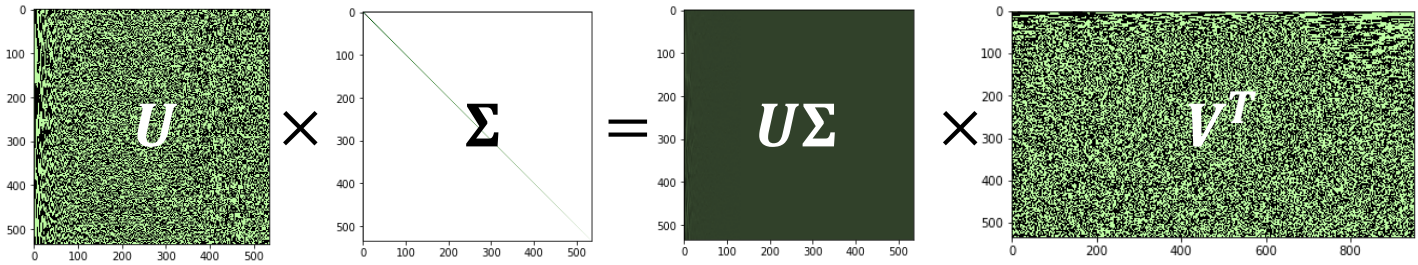
$$X^T = (U\Sigma V^T)^T$$

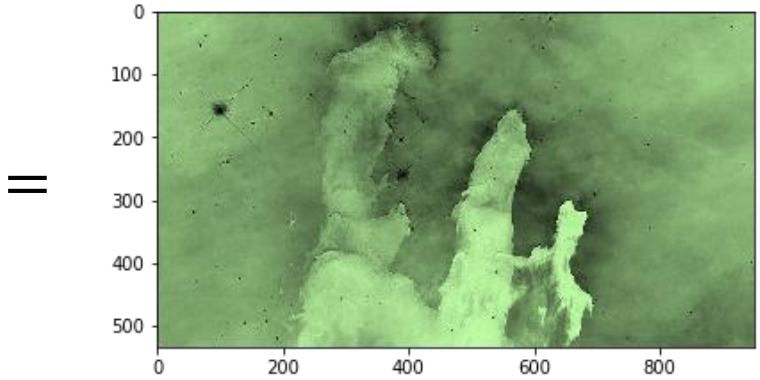
$$X^T = (V^T)^T \Sigma^T U^T \quad (1)$$

$$X^T = V\Sigma U^T \quad (2)$$

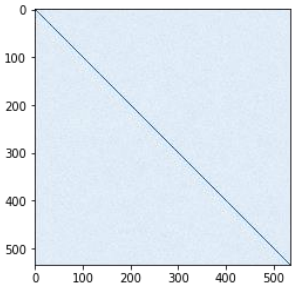
(1.) Transpose of a chain of matrices reverses the chain and transposes each element. Transpose of a transpose is itself. (2.) Diagonal matrix transpose is still itself.

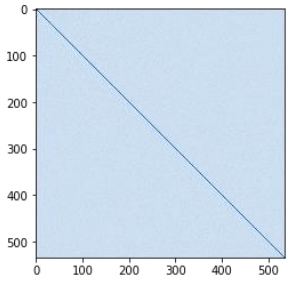
But how does the 3 components U, Σ, V^T look like? By isolating only the green channel of the Pillars of Creation, you can see how at first, U and V^T look like a mess. But when multipled together in the correct way, an image appears!



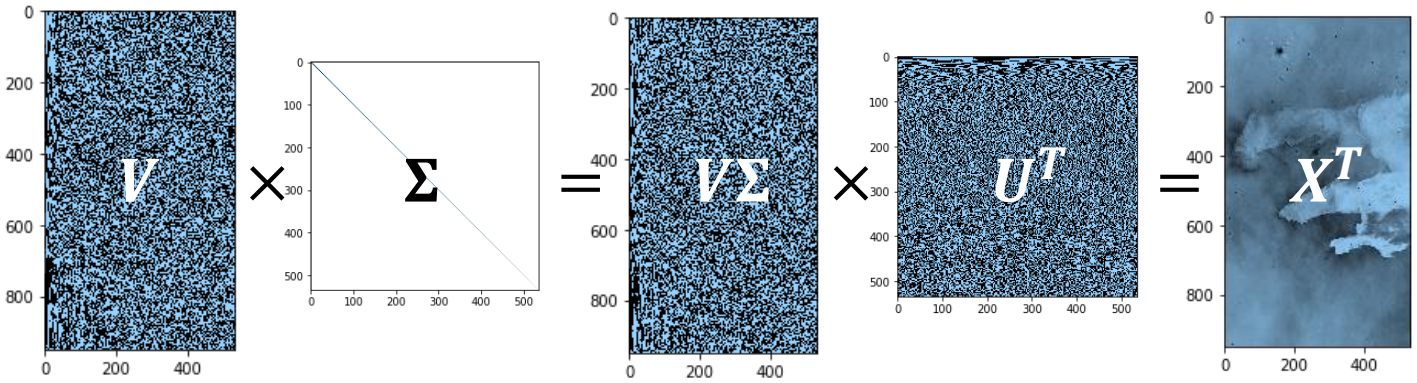


To confirm the 3 facts before - $U^T U = I, V^T V = I$ and $X^T = V \Sigma U^T$, we can easily check this with the images again by rounding U, V to the nearest 1 decimal place and verifying that a diagonal is still apparent after multiplying.

$U^T U =$ 

$V^T V =$ 

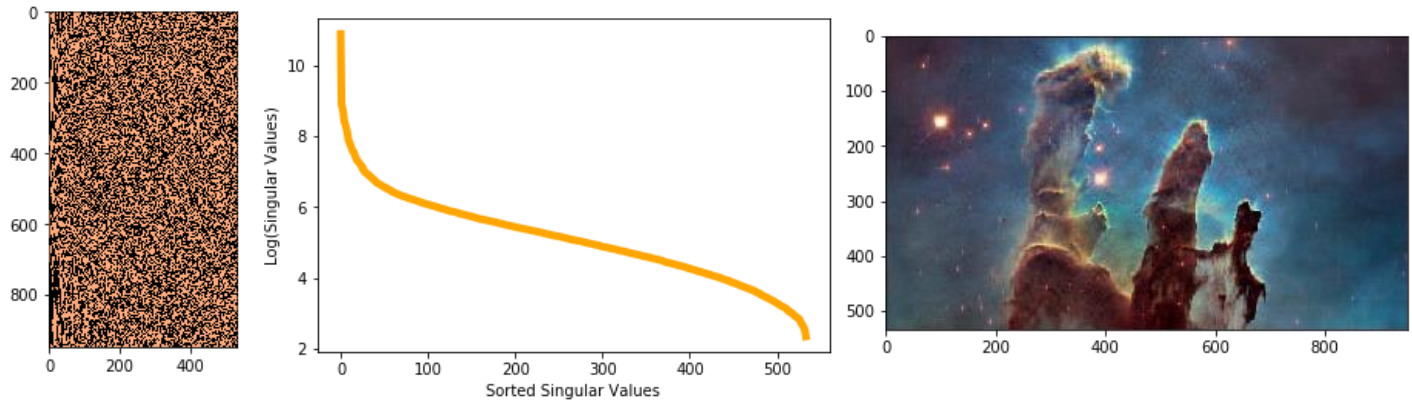
Likewise, to check our claim that $X^T = V \Sigma U^T$,



We can clearly see how all my claims seem to hold. These properties are very important when we want to compute the **SVD** of X much faster than before.

One thing you may have noticed is the odd $V\Sigma$, as it looks like the left columns have dark streaks, but the right looks like noise. This is because **SVD** produces **sorted singular values** in Σ. Singular values are positive numbers showcasing how much information each **SVD** component it has. A higher number = more information.

It's not abnormal for some of these to be small! That's why the right looks unimportant. You can see how the top maybe 50 singular values seem to capture the most information about the image, when we plot a line graph for the singular values.



- *I have a 100MB image. I'm uploading it, but it says I need to wait 10 minutes.*

We can solve this! Without losing a lot of image quality, we can **discard** all singular values that are small! The image on the right shows keeping only 50 singular values.

By discarding 494 singular values, we keep the first 50 columns of U, and 50 rows of V^T. Originally, $n^2 + p^2 + np = 1{,}077{,}612$ bytes were needed to store the 3 parts.

Now, only $\frac{50}{534}n^2 + \left(\frac{50}{534}p\right)^2 + \frac{50}{534}np = 76{,}700$ bytes are needed. So, a 93% size reduction! Instead of sending 100MB, send 7MB with 99% quality retainment!

But, you can reduce the size even further! The power lies in the **Eckart–Young–Mirsky Theorem,** which says that SVD provides the best approximation of a compressed matrix when compared to the full matrix. (C. Eckart)

It says the **Frobenius Norm** of the compressed matrix and the full is minimised when using **SVD**. The Frobenius Norm is the sum of the element wise squared difference between the old matrix and the new one. (:k means up to component k)

$$\min\left\|X - \widehat{X_{:k}}\right\|_F^2 = \min\left\|X - U_{:k}\Sigma_{:k}V_{:k}^T\right\|_F^2 \quad for\ rank\ k$$

The hat $\widehat{X_{:k}}$ means approximation. We can see that the information held by each component $X_k = U_k\Sigma_k V_k^T$ is directly proportional to its own singular value squared.

$$\boldsymbol{Info}(X_k) = \frac{\Sigma_k^2}{n-1}$$

$$\%\boldsymbol{Info}(X_k) = \frac{\Sigma_k^2}{n-1} \div \sum \boldsymbol{Info}(X_i) = \Sigma_k^2 \div \sum \Sigma_i^2$$

Using some Linear Algebra, we can also see that

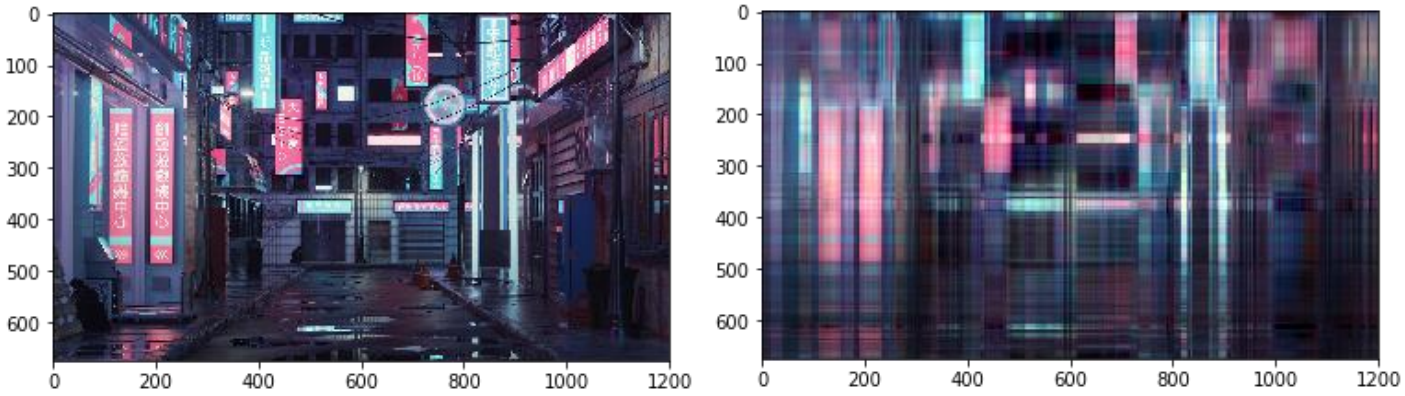
$$\boldsymbol{trace}(X^T X) = \sum \lambda_i = \sum \Sigma_{\mathrm{i}}^2$$

$$\boldsymbol{trace}(X^T X) = \sum \lambda_i = \sum \boldsymbol{Info}(X_i)$$

$$\%\boldsymbol{Info}(X_k) = \frac{\Sigma_k^2}{n-1} \div \boldsymbol{trace}(X^T X) = \Sigma_k^2 \div \sum \Sigma_i^2$$

So, the **trace**, or diagonal sum of the **covariance matrix** $X^T X$ is the sum of the eigenvalues of $X^T X$, which is the sum of the singular values squared. (Angell, 2010). Notice in statistics, $\boldsymbol{Info}(X_k), \boldsymbol{Var}(X_k)$ are used interchangeably (they're the same).

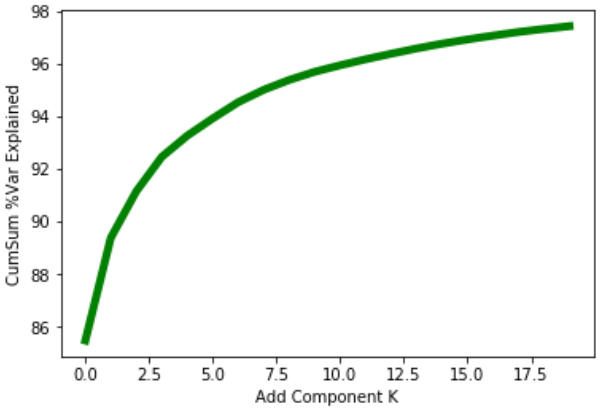
So, $\%\boldsymbol{Info}(X_k) = \%\boldsymbol{Var}(X_k)$ is a number from 0 to 1 and shows the % of information each component explains. This means we can compress the image knowing how much % components you want to keep!



SVD on "City Night" by Stijn Orlans. Left is original. Right P = 0.01 (keep 1% information) (Orlans)

SVD Image Compression Algorithm

P = 0.01 (Information kept)
Info = 0 (%Info Explained)
For every channel in (R,G,B):
    U, S, VT = **SVD**(channel) (1)
    K = **P***length(S)
    Var = S^2/(length(U)-1) / trace(channel.T @ channel) (2)
    Info += **CSum**(Var)[K] / 3
    compressed = U[:,:K] * S[:K] @ VT[:K] (3)
approximation = merge(compressed)



Notice **Cumulative Sum (CSum)** works when a list is sorted. Singular Values are sorted from largest to smallest. If I have a list = [1,0,3,8], then the CSum is [1,1,4,11].

Plotting CSum $\%\boldsymbol{Info}$, the first 5 components hold 94% of the image's information.

(1.) Reduced SVD (2.) @ is matrix multiplication. * is element wise multiply. (3.) U[:,:K] is first K columns. V[:K] first K rows.

a. Fast Singular Value Decomposition

SVD is a rather hard problem to solve. Given a matrix X of size (n,p), where n is the number of rows, and p = number of parameters or columns, **SVD** runs in $O(np^2)$. (Trefethen, 1996).

One issue with **Big O** notation is that it hides the constant factors. According to Trefethen, **SVD's** total computational FLOPs (floating point operations) is approx $2np^2 + 11p^3\ FLOPs$. (Lower is faster).

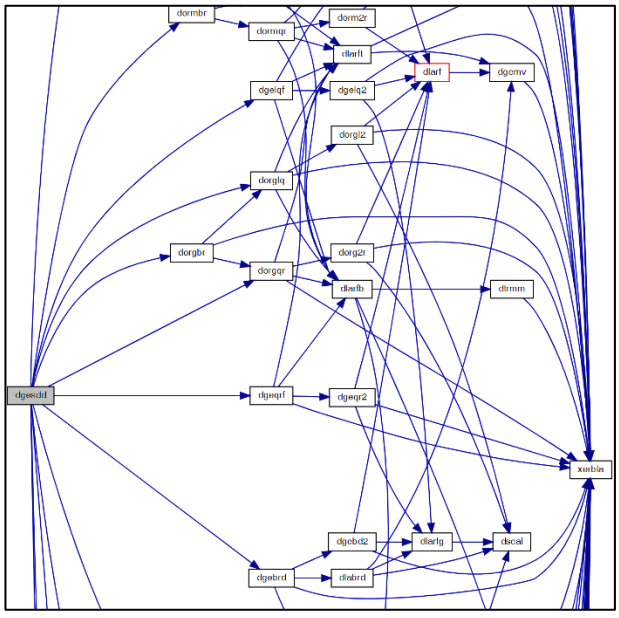
Modern day **SVD** algorithms are implemented in 2 steps:

1. Reduce X to a bidiagonal matrix (Householder Reflections)
2. Divide n Conquer Eigenvalue Algorithm

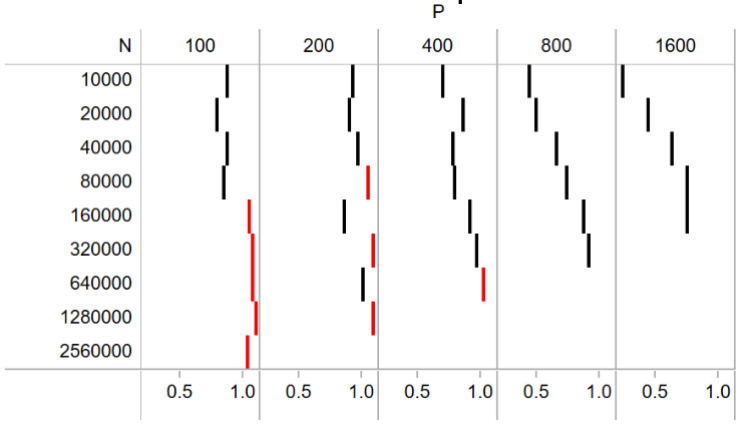
Older algorithms use a Golub QR Iteration (Golub & Kahan, 1965) instead of a Divide n Conquer for step 2. (Cuppen, 1981).

The Divide n Conquer **SVD** Algorithm computes the **SVD** on 2 halves of the matrix, after it has been processed to become tridiagonal. Each part can be solved in parallel.

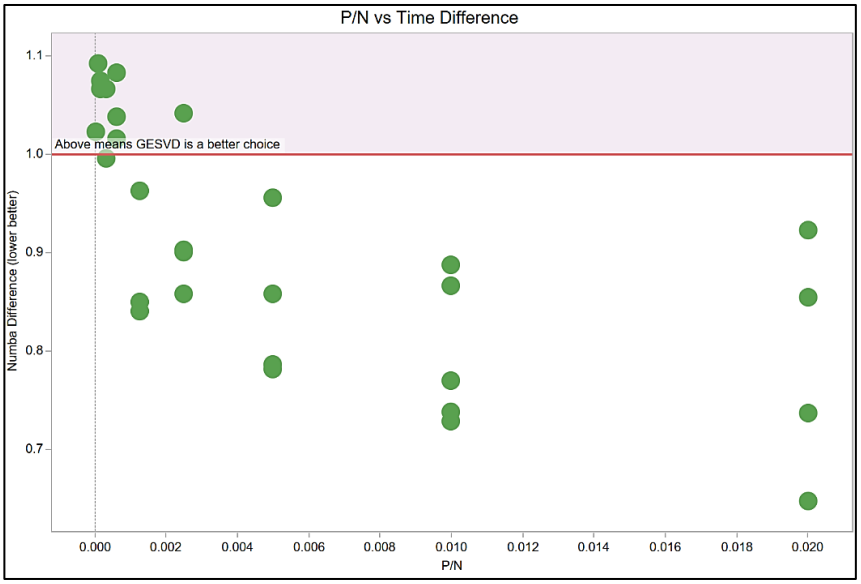
By using the **Master Theorem** and $T(n) = 2T\left(\frac{n}{2}\right) + O(n^2)$, Cuppen's Divide n Conquer **SVD** has complexity $O(n^2)$. In the modern linear algebra package LAPACK, both Golub (**GESVD**) and Cuppen (**GESDD**) algorithms are provided. (Blackford, 1999). 2 steps look easy right? Below shows LAPACK's SVD's computation tree.



LAPACK's SVD is complicated (Blackford, 1999)



Now, is Divide n Conquer **SVD** always faster than Golub QR? Above, when N is large and when P is small, GESVD seems to win. When you plot $\frac{P}{N}$ vs $\frac{GESDD}{GESVD}$, you can see a trend that when there are few columns, GESVD wins.



Above 1 means GESVD is faster. Graphs: Tableau. Test: Microsoft Surface Core i5 4 cores, 8GB RAM, Windows 10.

Clearly, we have devised a faster **SVD** algorithm already!

By using $\frac{P}{N}$, we can approximately compute **SVD** faster using both GESVD and GESDD. Obviously, it's not guaranteed to always have lower running times. We can see that if $\frac{P}{N} < 0.001$, it is most likely better to use Golub **SVD** instead of Divide n Conquer **SVD**.

The Fast-SVD Algorithm

N, P = shape(X)
Ratio = P/N
return **GESVD**(X) if **Ratio** < 0.001 else **GESDD**(X)

I will attempt to explain why GESVD is faster when the number of columns is small. GESDD divides the matrix into halves and works best when both are relatively square. When a matrix is **thin**, dividing the matrix into 2 provides clearly non-square halves.

How about when a matrix is **wide**? (When $P > N$). Remember that $X^T = V\Sigma U^T$, and $U^T U = I, V^T V = I$?. We also know that the complexity of SVD is approx. $O(np^2)$, so why not transpose X, compute the SVD, then return the components? If you transpose the matrix, $N \leftrightarrow P$ is swapped. So, the new complexity is $O[\min(np^2, n^2 p)]$. If a matrix is wide, we compute svd(X^T), and else svd(X).

The Even Faster-SVD Algorithm

N, P = shape(X)
If P > N:
    svdTranspose = TRUE
    X = X.T
N, P = shape(X)
Ratio = P/N
U, S, VT = **GESVD**(X) if **Ratio** < 0.001 else **GESDD**(X)
Return U, S, VT if **svdTranspose** = FALSE else VT.T, S, U.T

One thing to note is when you compute svd(X^T), you must return $(V^T)^T, \Sigma, U^T$. You can see that only n, p is compared, thus complexity remains at $O[\min(np^2, n^2 p)]$.

$$X^T = \mathbb{U}\mathbb{S}\mathbb{V}^T$$
$$X = \mathbb{V}\mathbb{S}\mathbb{U}^T$$
$$So\ \mathbb{V}\mathbb{S}\mathbb{U}^T = U\Sigma V^T, so\ U = (\mathbb{V}^T)^T, \Sigma = \mathbb{S}, V^T = \mathbb{U}^T$$

b. Principal Component Analysis

Fast-SVD is good enough for most tasks in astronomy, physics and computer vision. But, it can be made even faster! In the field of Artificial Intelligence and data science, **Principal Component Analysis (PCA)** is an important tool, as it can compress large datasets into a few components, where each component explains some information about the dataset.

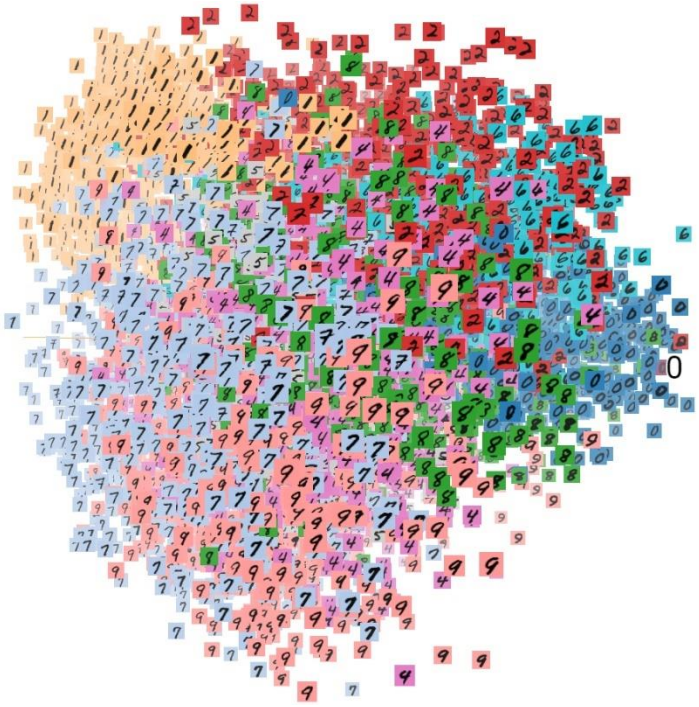
PCA is just the singular value decomposition of a **centred matrix** (remove column mean) and provides to the user the **eigenvectors** and **eigenvalues** of that matrix. We can easily see that: (Strang, 2016)

$$X = U\Sigma V^T$$
$$X^T X = V\Sigma U^T U\Sigma V^T = V\Sigma\Sigma V^T$$
$$X^T X = V\Sigma^2 V^T$$

Since $U^T U = I$ and a diagonal matrix Σ multiplied by itself is just itself squared. Clearly, the SVD includes the eigenvectors and eigenvalues for $X^T X$, since $V\Sigma^2 V^T$ looks suspiciously like the **diagonalized Eigendecomposition** for symmetric matrices:

$$A = V\Lambda V^{-1} = \mathbf{V\Lambda V^T}$$

So, we can see that Λ: the eigenvalues of $X^T X$ are just the singular values Σ squared.



PCA with 3 components on MNIST Digits data. Digits images are clustered quite well. (Google, n.d.)

If you apply SVD on **non-centred data**, for example, on **bag of words** text data, then this algorithm is called **Latent Sematic Indexing (LSI)**. (Deerwester, 1988)

You can find the eigenvalues, eigenvectors easily via SVD:

$$U, \Sigma, V^T = SVD(X - \bar{X})$$
$$\mathbf{\Lambda} = \Sigma^2$$
$$\mathbf{V} = (V^T)^T$$

$\bar{X}$ is column mean. So, PCA can be computed in $2np^2 + 11p^3 + 2np$ FLOPs, after realising mean removal.

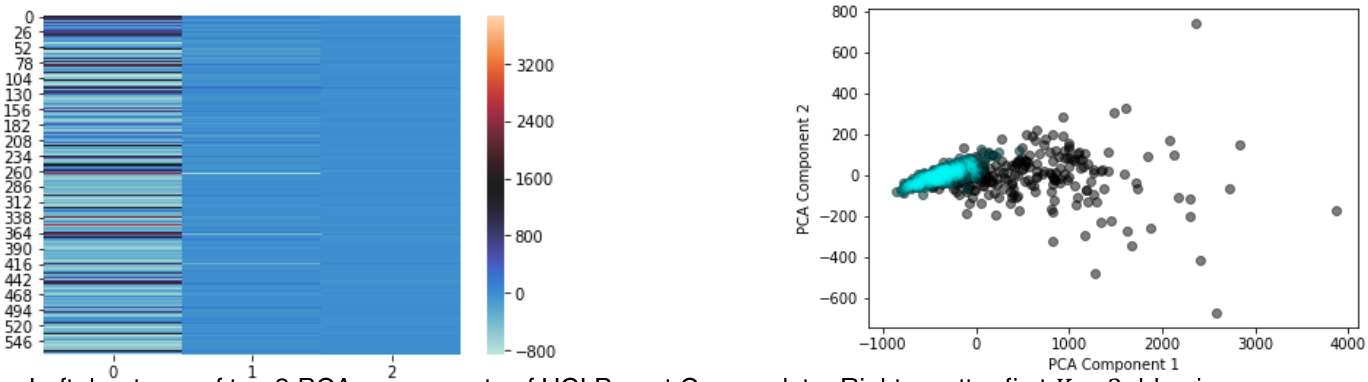
The eigenvalues Λ are discarded (used in $\%\boldsymbol{Info}$ analysis), and the eigenvectors V becomes the **transformation matrix**. This means that given new data A:

$$A_{transformed} = (A - \bar{X})\mathbf{V}$$

Then, invoke the **Eckart–Young–Mirsky Theorem**, which says that each component K provides the minimal **Frobenius Norm**. So $\sum(X - U_{:K}\Sigma_{:K}V_{:K}^T)^2$ is minimised.

- *I'm trying to understand a dataset, but I can't graph all 1,000 columns on 1 graph.*

Pretend you want to construct 3 columns which describes 1000 columns of data. You can easily create 3 random columns. But, we want the 3 random columns to capture as much information as possible as 1000 columns would. So, we select the top 3 components from the transformed A, and this is **Principal Component Analysis**.



Left: heatmap of top 3 PCA components of UCI Breast Cancer data. Right: scatter first $K = 2$, blue is cancerous.

PCA is the same as centred SVD. So, the complexity is $O[\min(np^2, n^2p)]$. There's an issue! PCA provides eigenvectors for X^TX, the covariance matrix. This forces the complexity to be $O[\min(np^2)]$, as $SVD(X^T)$ will showcase incorrect eigenvectors.

Remember PCA finds the eigenvectors for X^TX, and $\mathbf{\Lambda} = \Sigma^2$. Then:

$$X^TX = V\Sigma^2V^T = \mathbf{V\Lambda V^T}$$

$$X\mathbf{V} = U\Sigma\mathbf{V^T V} = U\sqrt{\mathbf{\Lambda}}$$

$$\text{Also, } X^T = V\Sigma U^T$$

$$XX^T = U\Sigma V^T V\Sigma U^T = U\Sigma^2U^T$$

Now XX^T's or the **gram matrix** has eigenvectors $\mathbb{U}$ and eigenvalues $\mathbb{E}$.

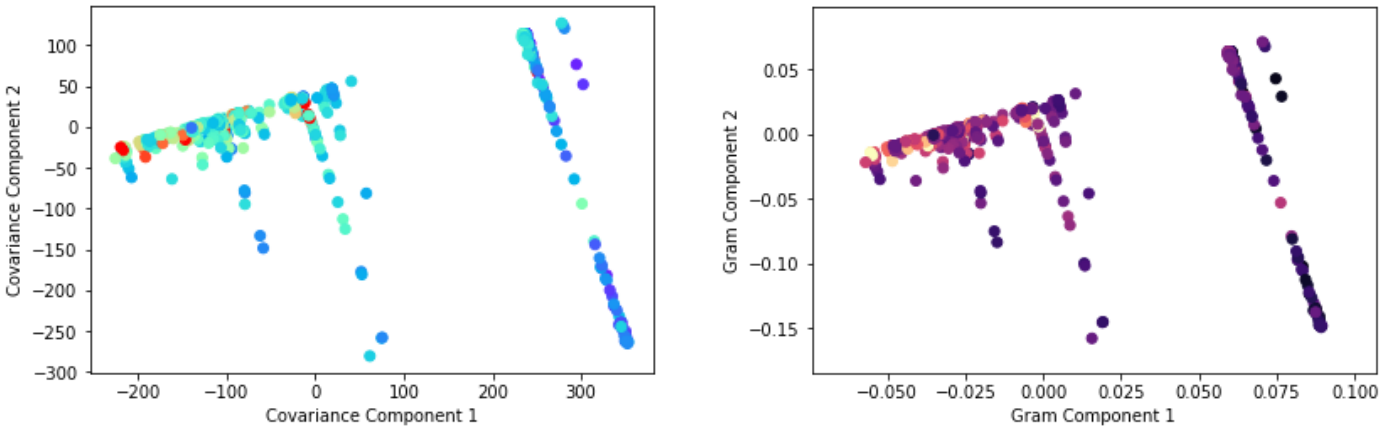
$$XX^T = U\Sigma^2U^T = \mathbb{U}\mathbb{E}\mathbb{U}^T$$

Our aim is to find $U\Sigma = A_{transformed} = (A - \bar{X})\mathbf{V}$. So, we want to eliminate V^T. How?

$$X^T\mathbb{U} = V\Sigma U^T\mathbb{U} = \mathbf{V}\Sigma$$

$$\text{Clearly, } \mathbf{V} = \frac{X^T\mathbb{U}}{\Sigma} = \frac{X^T\mathbb{U}}{\sqrt{\mathbb{E}}}$$

So, by using XX^T, we obtain $\mathbf{V}$ naturally**.** (Good, 1969)

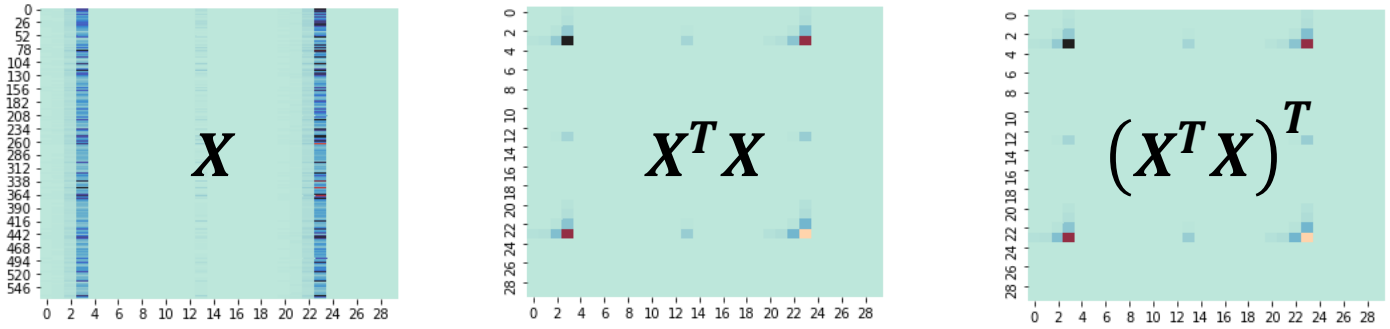


Left: X^TX. Right: XX^T. Both show $K = 2$ for Boston House Data (Harrison, 1978). Note the **scale difference**.

Above, you can see how both look relatively the same, but there is a **clear scale difference**. The Eigendecomposition on the Gram Matrix produces in fact **normalized** vectors. But in practice, we can ignore this. So, because this occurs, we are estimating $\mathbf{V}$, hence the $\mathbf{V}$ hat: $\widehat{\mathbf{V}} = \frac{X^T\mathbb{U}}{\sqrt{\mathbb{E}}}$.

Also realise the important fact that X^TX is a **Hermitian matrix**, which is also a **symmetric matrix.** This means that $X^TX = (X^TX)^T$. It is quite easy to prove this:

$$(X^TX)^T = X^T(X^T)^T = X^TX$$



Heatmap of Breast Cancer Wisconsin data. X^TX is symmetric. X^TX and $(X^TX)^T$ look alike.

Because X^TX and XX^T are symmetric, instead of taking $\min(np^2, n^2p)\ FLOPs$, we only need to compute $\frac{1}{2}$ of the matrix, then reflect it along the diagonal. Hence, FLOPs is reduced to $\min\left(\frac{1}{2}np^2, \frac{1}{2}n^2p\right)$. In **BLAS**, **SYRK** performs symmetric multiplication.

Also, remember U does not need to be calculated. So, on a square matrix, SVD takes $13p^3$ time. Now, directly using the Golub method disregarding U takes now $9p^3$.

Because X^TX and XX^T are **Hermitian**, **Cuppen's Divide Algorithm** is used to find the eigenvectors and eigenvalues, where $4p^3\ FLOPs$ are needed. (Cuppen, 1981). Don't forget to factor in the formation of X^TX, XX^T, which takes $\min\left(\frac{1}{2}np^2, \frac{1}{2}n^2p\right)$.

Now, to perform $\frac{X^T\mathbb{U}}{\sqrt{\mathbb{E}}}$, it takes $n^2p + np\ FLOPs$, or $O(n^2p)$ extra. We should add this into our complexity analysis.

Full SVD	No U SVD	Cuppen's Algorithm
$\min \begin{pmatrix} 2np^2 + 11p^3 \\ 2n^2p + 11n^3 \end{pmatrix}$	$\min \begin{pmatrix} \frac{1}{2}np^2 + 9p^3 \\ \frac{3}{2}n^2p + 9n^3 + np \end{pmatrix}$	$\min \begin{pmatrix} \frac{1}{2}np^2 + 4p^3 \\ \frac{3}{2}n^2p + 4n^3 + np \end{pmatrix}$

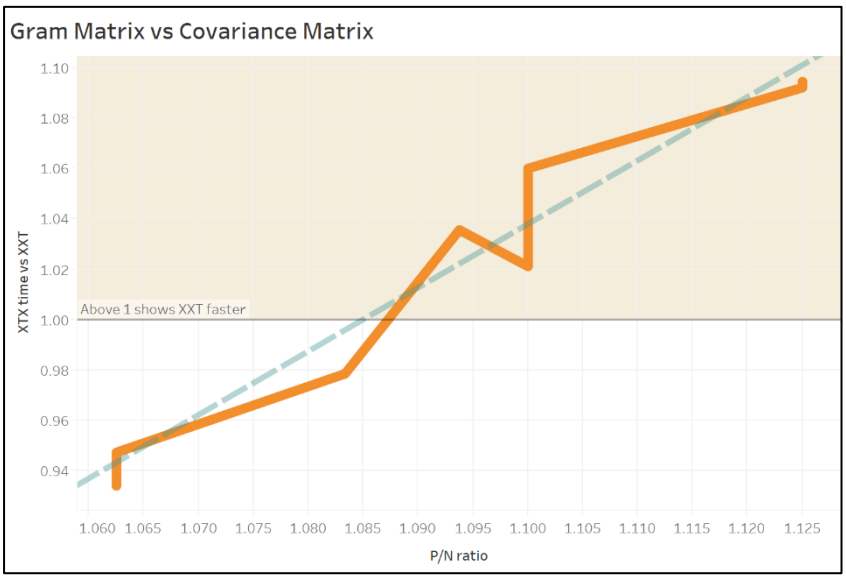
So, there's a dilemma - which is faster when $p > n$? Let's first remove **Full SVD**:

$$2n^2p + 11n^3 > \frac{3}{2}n^2p + 4n^3 + np$$
$$\frac{1}{2}n^2p + 7n^3 > np$$
$$\frac{1}{2}np + 7n^2 > p$$
$$\frac{1}{2}n + \frac{7n^2}{p} > 1$$
$$\text{Take } n = 3, p \to \infty, \text{ then } \frac{3}{2} > 1$$

So, **Full SVD** slower than Cuppen for all $n \geq 3$. Now, which is faster when using the **gram matrix** instead of the **covariance matrix**?

$$\frac{3}{2}n^2p + 4n^3 + np > \frac{1}{2}np^2 + 4p^3$$

Let's take a wild $n = 100$ and $p = 101$. Then, $5{,}525{,}100 > 4{,}631{,}254$. If we make $p = 150$, then: $6{,}255{,}000 < 14{,}625{,}000$. Clearly, it shows that if the difference between n and p is small, then X^TX wins. So, we should test this using **LAPACK's SYEVD**.



Above 1 means XX^T is faster. Graphs: Tableau. Test: Microsoft Surface Core i5 4 cores, 8GB RAM, Windows 10.

Clearly, as $\frac{p}{n}$ increases (more columns than rows), using XX^T wins rather than X^TX.

However, there is an issue. By fitting a linear line on the data, we see that:

$$\frac{time(X^TX)}{time(XX^T)} = 2.5\frac{p}{n} - 1.75$$

We want $\frac{time(X^TX)}{time(XX^T)} > 1$, so:

$$\frac{time(X^TX)}{time(XX^T)} = 2.5\frac{p}{n} - 1.75 > 1 \text{ implying that } \frac{p}{n} > 1.1.$$

This shows that approx. the no. of columns must exceed $1.1n$ for XX^T to win. Hence, we can devise our **Fast-Eigendecomposition Algorithm** with these insights!

d. Epsilon Jitter, QR-SVD Fast-PCA Algorithms

By combining our insights about fast symmetric matrix multiplication and using **LAPACK's** Divide n Conquer **SYEVD**, we now devise our **Fast-Eigen** algorithm:

The Fast-Eigendecomposition Algorithm

N, P = shape(X)
A = X – **colMean**(X) (1)
If P > 1.1 * N:
    AAT = **SYRK**(A.T, trans = 1) (2)
    V, Λ = **SYEVD**(AAT)
    $\Lambda[\Lambda < 0] = 0$ (3)
    V = (A.T @ V) / $\sqrt{\Lambda}$ (4)
Else:
    ATA = **SYRK**(A.T, trans = 0)
    V, Λ = **SYEVD**(ATA)
Return V, Λ

(1.) **colMean** means to remove the mean of each column (2.) **SYRK** is **BLAS's** symmetric multiply. (3.) Eigenvalues can be negative. Set them to zero. (4.) @ is matrix multiplication. / is element wise division.

One **flaw** with this algorithm is that <u>all matrices</u> are SVD factorizable, hence the PCA must exist. However, a matrix can be un-diagonalizable if it's **singular** or **defective**. (Weisstein, n.d.) Each column must be **linearly independent**. This means if $X_{col1} + X_{col2} = X_{col3}$, then that's not diagonalizable. This issue occurs in **OneHot Encoding**.

Determining whether a matrix has linearly independent columns is tedious. By using **L2 regularization**, we can now force a matrix to have linearly independent columns:

$$\lim_{\lambda \to 0}(X^T X + \lambda I)$$

The trick is we add some **small machine epsilon jitter** to the diagonal, hence making it 100% linearly independent. If the algorithm fails to converge, add a larger jitter. Start off with $jitter = \lambda = 0.0001$, then multiply by 10 until success.

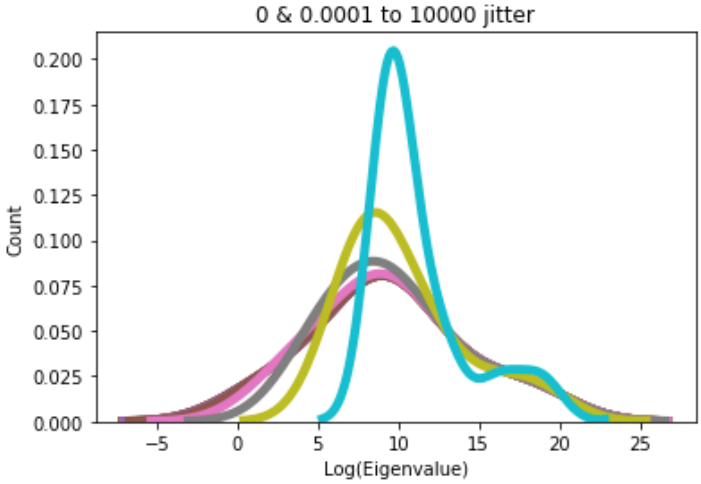
The Epsilon Jitter Algorithm

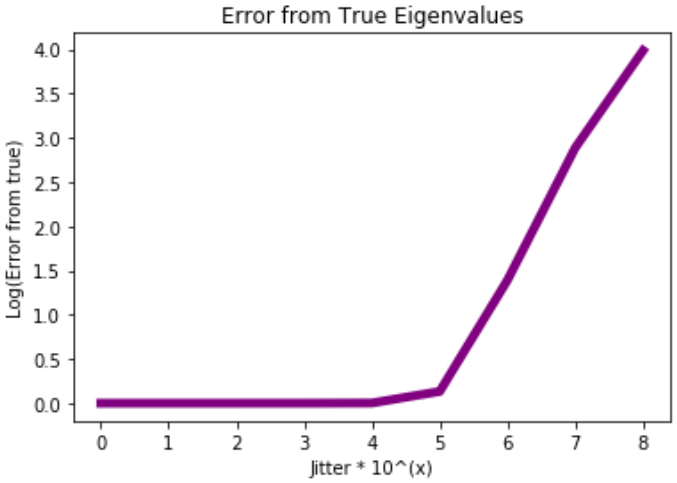
jitter = 0, **converged** = FALSE
While not **converged**:
    Try:
        V, Λ = **Fast-Eigen[** min(A.T @ A + **jitter***eye OR A @ A.T + **jitter***eye) **]**
    Except:
        jitter = **jitter***10 if **jitter** > 0 else 0.0001
Return V, Λ

When adding some **jitter**, doesn't it change the final eigenvalues? **YES**, but by a negligible amount, so don't worry! In fact, let us quantify this error using:

$$\boldsymbol{Error}(\boldsymbol{\Lambda}, \widehat{\boldsymbol{\Lambda}}) = \sum \log^2(\boldsymbol{\Lambda} - \widehat{\boldsymbol{\Lambda}})$$

$\widehat{\boldsymbol{\Lambda}}$ is the estimated eigenvalues using our **Epsilon Jitter Algorithm**, and $\boldsymbol{\Lambda}$ is the true eigenvalues for Boston data (Harrison, 1978). If you plot a **kernel density** and **line plot**, we see that as $\lambda \to 0, \boldsymbol{Error} \to \mathbf{0}$. But, as $\lambda \to \infty$, and the distribution shifts.

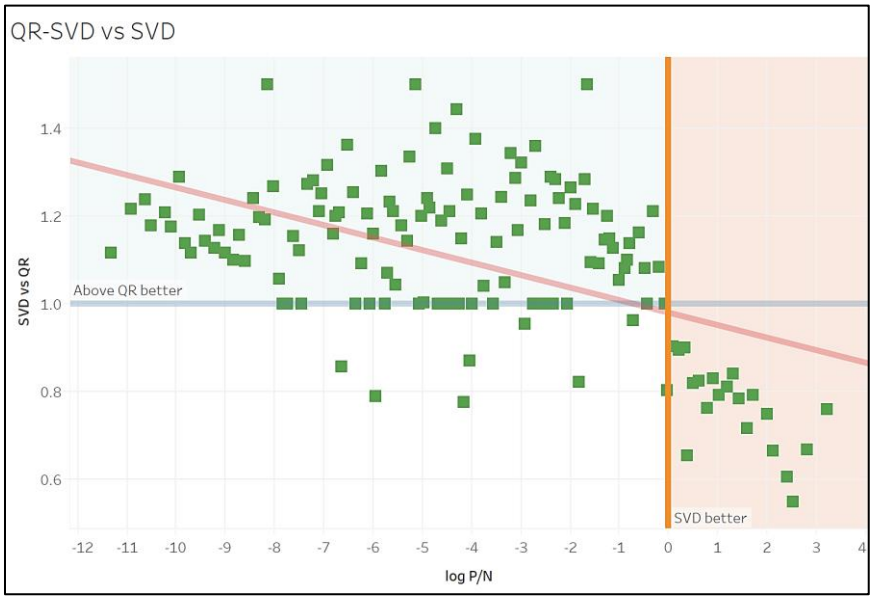




Clearly, when jitter is small, the algorithm seems to have no clear disadvantages. Another algorithm worth mentioning is from Holub's Matrix Computation (Holub). It showcases that if $n \geq \frac{5}{3}p$, then the following **QR Algorithm** wins:

The QR-SVD Algorithm

N, P = A.shape
If N >= 5/3*P:
    Q, R = **QR** (A)
    U, S, VT = **Fast-SVD** (R)
Else: U, S, VT = **Fast-SVD (**A)
V, Λ = S^2, VT.T
Return V, Λ

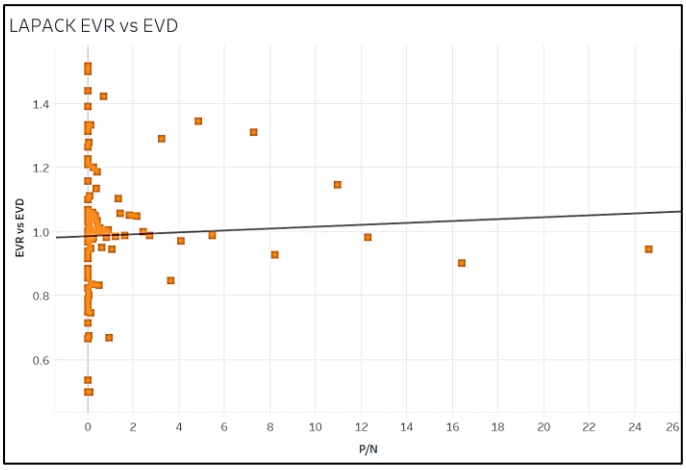


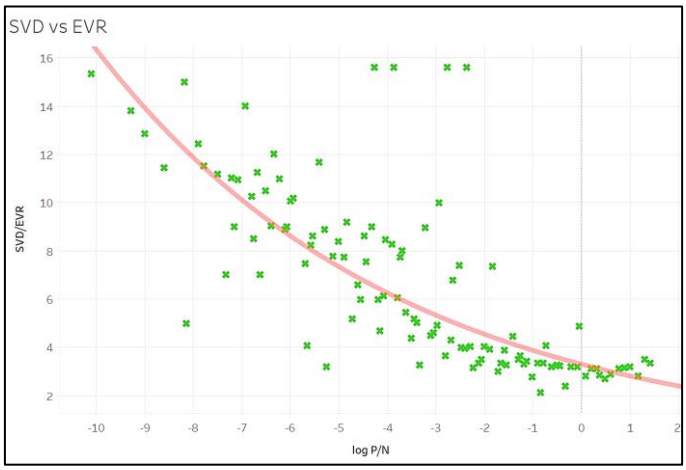
And I tested that claim out. Clearly, **<u>Holub isn't ambitious enough</u>**. It is apparent that if $n > p$, the **QR-SVD** Algorithm is better. So, the QR Algorithm should be invoked if $n > p$ and not if $n \geq \frac{5}{3}p$.

Left: SVD/QR vs log(P/N). Graphs: Tableau. Test: Microsoft Surface Core i5 4 cores, 8GB RAM, Windows 10.

There is one more important **LAPACK** consideration. I am currently timing **SYEVD** (or Cuppen's Divide n Conquer Eigenvalue Algorithm). There is another eigendecomp algorithm called **SYEVR** (**Relatively Robust Representations**), which in time complexity shows $O(n^2)$ rather than $O(n^3)$ complexity.

According to James W. and co. (Demmel, 2008), SYEVD (Divide) is sometimes faster than SYEVR. So, it is unclear which is faster, however, SYEVD (Divide)'s accuracy is $O(\sqrt{n}\varepsilon)$, whilst SYEVR is $O(n\varepsilon)$. Clearly, SYEVD's error is smaller.





Many random matrix tests. Graphs: Tableau. Test: Microsoft Surface Core i5 4 cores, 8GB RAM, Windows 10.

I test this myself, and it is apparent there exists no clear trend. However, it seems on average, SYEVR is slower than SYEVD. On the right, we see SVD vs SYEVR. Clearly, pure SVD is always slower. So, if we combine our findings on **QR-SVD** and **SYEVD**:

The Fast-PCA Algorithm

```
useSVD = FALSE, jitter = 0, converged = FALSE                    (1)
N, P = A.shape
If useSVD:
        If N >= P:                                               (2)
                Q, R = QR (A)
                U, S, VT = Fast-SVD (R)
        Else:   U, S, VT = Fast-SVD (A)
        V, Λ = S^2, VT.T
Else:
        While not converged:
                Try:
                        If P > 1.1 * N:
                                AAT = SYRK(A.T, trans = 1)
                                V, Λ = SYEVD(AAT)
                                Λ[Λ < 0] = 0
                                V = (A.T @ V) / √Λ
                        Else:
                                ATA = SYRK(A.T, trans = 0)
                                V, Λ = SYEVD(ATA)
                Except: jitter = jitter*10 if jitter > 0 else 0.0001
Return V, Λ
```

(1.) **useSVD** is a user defined option. Use SVD for better accuracy. (2.) Notice N >= P, not N >= 5/3*P.

d.Fast Truncated, Randomized SVD

- *Why is SVD slow? I want the top K components containing 90% of information.*

This is the problem I haven't considered yet! A fundamental problem with both **Fast-SVD** and **Fast-PCA** Algorithms is that is it rather a waste of time to compute all eigenvalues and eigenvectors. We normally only want the **top K components**. Sometimes, top 3, or 2. So, is there a faster way to compute the top 5 eigenvectors?

The **Power Iteration** can be used to find the **top 1 eigenvector** with has the maximum $\%\boldsymbol{Info}(X_k)$ explained. It is relatively short: (Mises, 1929)

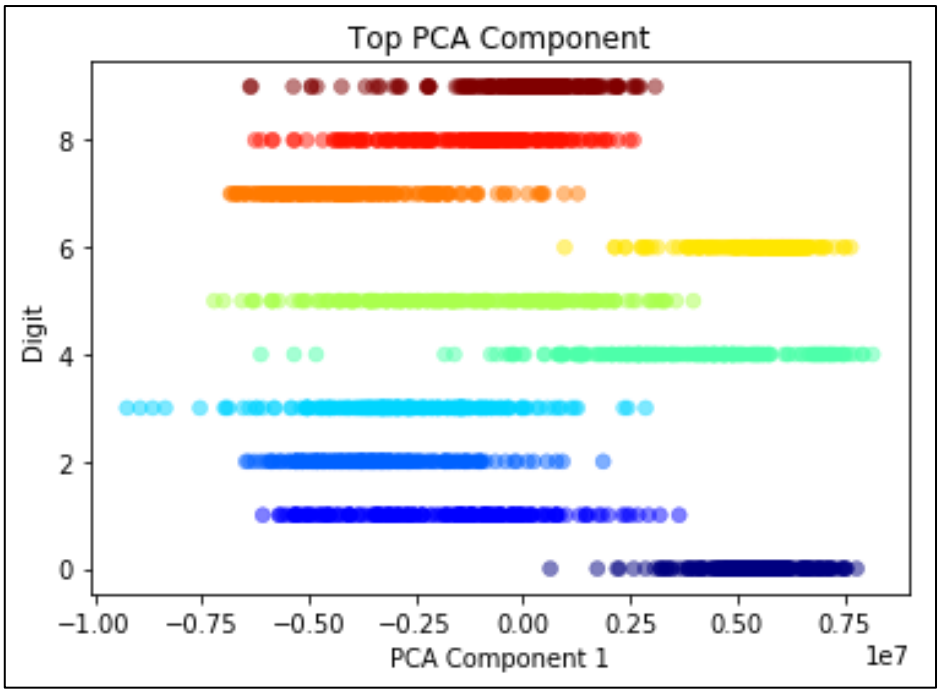
The Power Iteration Algorithm

```
N , P = A.shape, iterations = 2                    (1)
v = randomVector(size = P)                         (2)
ATA = SYRK (A.T, trans = 0)
For iteration in range(iterations):
        v = ATA @ v
        v = v / norm(v)                            (3)
Return v
```

(1.) **iterations** is user defined. (2.) **randomVector** creates random entries. (3.) Normalize vector by norm

Also remember that the **norm** of a vector is calculated by:

$$\boldsymbol{norm}(v) = \sqrt{\sum v_i^2} = \|v\|$$



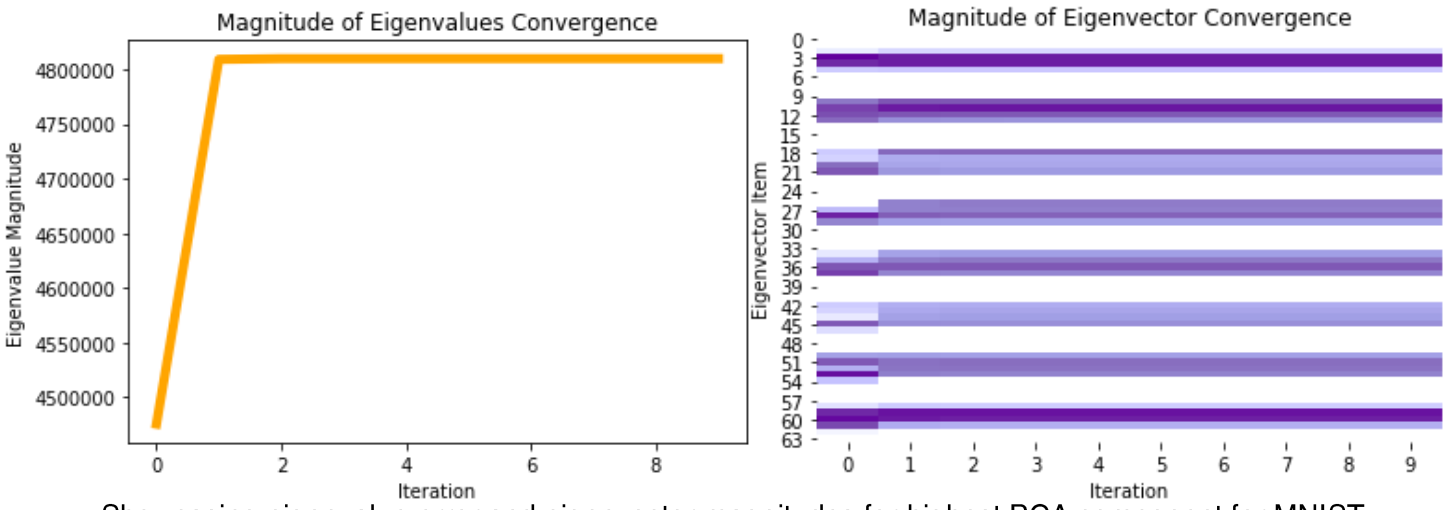
Plotting a **density scatter** of the top PCA component for MNIST digits. PCA captures similarities between classes. See 0 and 6 vs 3.

Clearly, Power Iteration has complexity $iterations \times (p^2 + 2p) + np^2$. Now, there is a problem. We only get the highest eigenvector. How about the eigenvalue? We can use **Rayleigh's Quotient**! (Parlet N.)

$$\lambda = \frac{v^T (X^T X) v}{\|v\|^2}$$

Notice its **norm squared**. If you noticed, there is a user input **iteration parameter.** What's considered good?

I test multiple iterations, and it is very clear below that after 2 iterations or so, the **Power Iteration** converges. So, constrained complexity is $2p^2 + 4p + np^2$.



Showcasing eigenvalue error and eigenvector magnitudes for highest PCA component for MNIST.

Knowing the top eigenvector is useful for some applications – like **PageRank**. However, normally, we want > 1 eigenvectors. But, we do not want all p eigenvectors, but just a smaller subset. So, how do we extend the Power Iteration algorithm? The **QR Iteration algorithm** (Francis, 1961) allows us to compute K top eigenvectors!

The QR Iteration Algorithm

N , P = A.shape, **iterations** = 10 (1)
Q = **randomVector**(size = (P, K)) (2)
ATA = **SYRK** (A.T, trans = 0)
For iteration in range(**iterations**):
    Q, R = **QR** (ATA @ Q) (3)
Λ , V = Q, **diag**(R)
Return Λ , V

(1.) **iterations** is user defined. (2.) random matrix P rows, K columns. (3.) QR Decomposition

However, one issue with the QR Iteration is its **convergence speed**. Also, notice $X^T X$ must be explicitly computed beforehand which can sometimes be very memory consuming. Notice, at each step, QR decomposition takes $2np^2 - \frac{2}{3}p^3\ FLOPS$. $A^T AQ$ is of size (p, k), where k = number of eigenvectors you want.

So, taking $i = iterations$, the total complexity is $i\left(2pk^2 - \frac{2}{3}k^3\right) + np^2\ FLOPS$, after considering the computation time of $A^T A$. Notice, approximately, $i \geq 10$ is needed for OK convergence, or else the eigenvalues diverge too much.

So, to remove these problems, Halko and co developed a fast **Randomized SVD algorithm**. (Halko, 2009). It uses QR Iteration as its base but extends it by projecting X^T and X onto the temporary eigenspace at each iteration. It is especially fast and can converge relatively well in $i = 7$ iterations. Notice, $X^T X$ does NOT have to be computed! Also, **LU decomposition** is used, taking approx. $np^2 - 1/3p^3\ FLOPS$.

Randomized SVD Algorithm

N , P = A.shape, **iterations** = 7
Q = **randomVector**(size = (P, K))
For iteration in range(**iterations**):
    Q, __ = **LU** (X @ Q) (1)
    Q, __ = **LU** (X.T @ Q) (2)
Q, __ = **QR** (X @ Q)
U*, S, VT = **Fast-SVD** (Q.T @ X)
U = Q@U*
Return U, S, VT

(1.) **LU Decomposition:** $X = LU$, where L is lower triangular, and U is upper triangular. (2.) Discard U.

Clearly, XQ takes npk operations. X^TQ also takes npk operations. The first **LU** takes $nk^2 - \frac{1}{3}k^3\ FLOPS$, and the second **LU** takes $pk^2 - \frac{1}{3}k^3\ FLOPS$. (Notice the first one is nk^2, but second is pk^2. So, the iteration step takes $i\left(2npk + nk^2 + pk^2 - \frac{2}{3}k^3\right)$.

But, we need to factor in the QR step, and SVD step. XQ takes npk. QR takes $2nk^2 - \frac{2}{3}k^3$. Q^TX takes npk. SVD takes approx. $2kp^2 + 11p^3$. And, QU^* takes npk.

So, the outside loop takes $3npk + 2nk^2 - \frac{2}{3}k^3 + 2kp^2 + 11p^3\ FLOPs$. Thus:

$$i\left(2npk + nk^2 + pk^2 - \frac{2}{3}k^3\right) + 3npk + 2nk^2 - \frac{2}{3}k^3 + 2kp^2 + 11p^3 =$$
$$npk(2i+3) + nk^2(i+2) - \frac{2}{3}k^3(i+1) + kp^2(i+2) + 11p^3$$

QR Iteration takes $i\left(2pk^2 - \frac{2}{3}k^3\right) + np^2 = 2pk^2i - \frac{2}{3}k^3i + np^2$. Also plug in i = 10&7.

$$2pk^2i - \frac{2}{3}k^3i + np^2 > npk(2i+3) + nk^2(i+2) - \frac{2}{3}k^3(i+1) + kp^2(i+2) + 11p^3$$
$$20pk^2 - \frac{20}{3}k^3 + np^2 > 17npk + 9nk^2 - \frac{16}{3}k^3 + 9kp^2 + 11p^3$$
$$20pk^2 - \frac{4}{3}k^3 + np^2 > 17npk + 9nk^2 + 9kp^2 + 11p^3$$

Now take $k = 3$ to graph top 3 components.

$$180p - \frac{4}{3}27 + np^2 > 41np + 81n + 27p^2 + 11p^3$$
$$180p - 36 + p^2(n-27) - 41np - 81n - 11p^3 > 0$$
$$180 - \frac{36}{p} + p(n-27) - 41n - \frac{81n}{p} - 11p^2 > 0$$

Now, remove the constants, and 36/p is relatively not useful, so worst case p = 1.

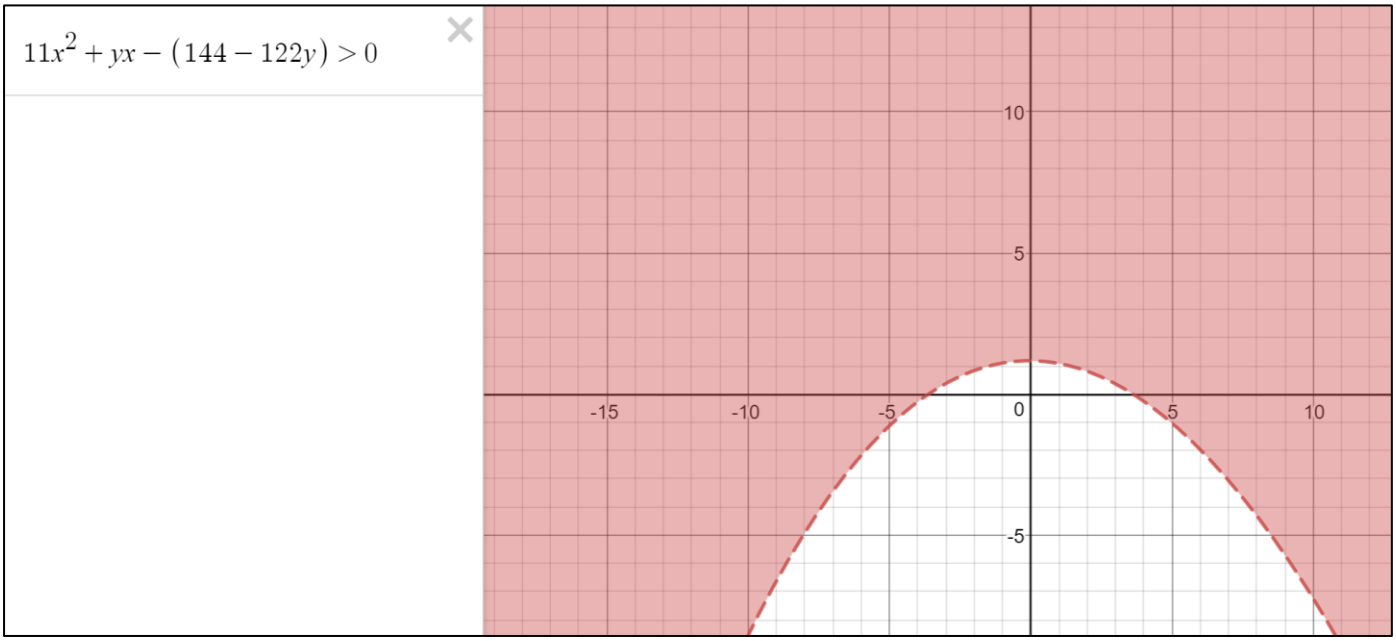
$$180 - 36 - n\left(p + 41 + \frac{81}{p}\right) - 11p^2 > 0$$

Likewise, 81/p is insignificant, so worst case p = 1 is added.

$$144 - n(p + 122) - 11p^2 > 0$$

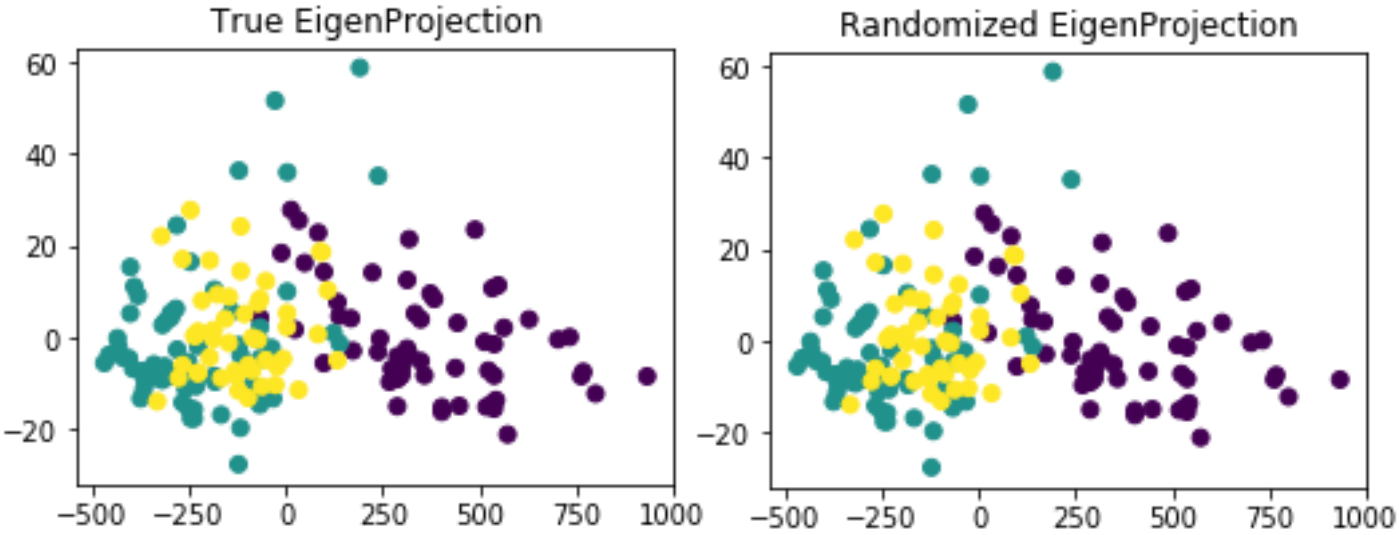
$$11p^2 + np - (144 - 122n) > 0$$

Clearly, for all $n > 1, p > 3$, Randomized SVD is always faster than QR Iteration even in the worst case if p = 1. We can confirm this via **desmos.com**!



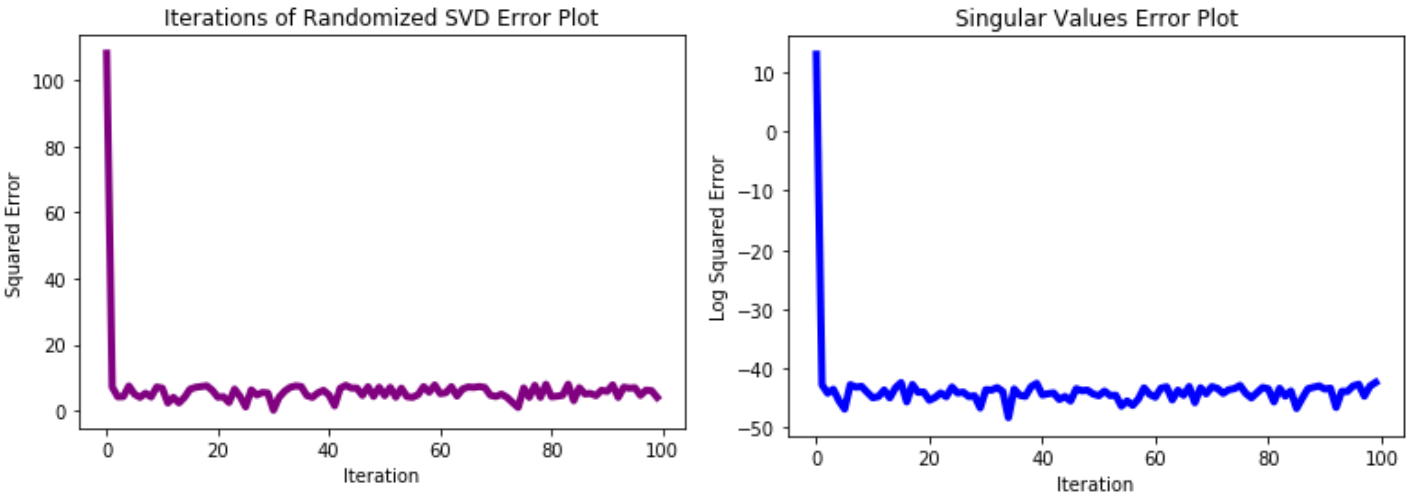
As you can see, the surface is satisfied iff x axis (=p) > 3, and y axis (=n) > 1. Which is obvious for all datasets, since clearly you can't have super small datasets.

So now, let us test if **RandomizedSVD** produces similar eigenvalues and eigenvectors on a real dataset – Wine dataset. (Lichman, 2013)



It looks wonderfully similar! But, we should also check the error for the eigenvectors given say iterations is set from 1 to 100. We should also check the singular values.

$$\boldsymbol{Error_i}(\mathbf{V}, \widehat{\mathbf{V}}) = \sum \sum (\mathbf{V} - \widehat{\mathbf{V}})^2 \qquad \boldsymbol{Error_i}(\boldsymbol{\Sigma}, \widehat{\boldsymbol{\Sigma}}) = \sum \log^2(\boldsymbol{\Sigma} - \widehat{\boldsymbol{\Sigma}})$$



Clearly, the error drops dramatically after just 2 iterations! In fact, for most datasets, providing iterations = 5 or so should be enough!

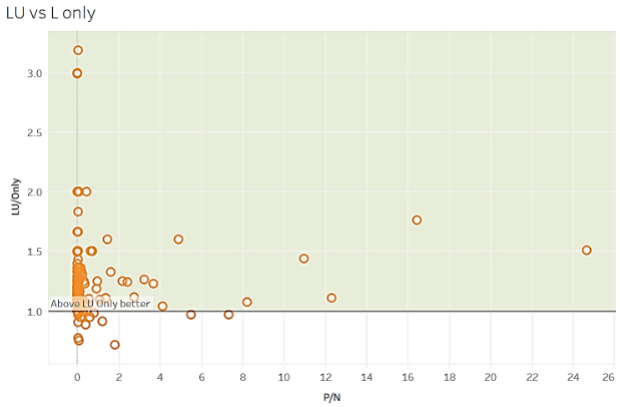
Notice we can make Randomized SVD even faster! Notice in the above algorithm, both LU and QR Decompositions do not require the U or R matrix, so why not just not calculate it? In LAPACK, **GETRF** is used for LU and **GEQRF** is for QR decompositions. However, GETRF does not permute the matrix L. So, we must call **LASWP** to permute the matrix L. For QR decomposition, we use **ORGQR** to get Q.

For LU, LAPACK returns a matrix where the top triangular is U and bottom is L. In Scipy, both U and L are made, hence a memory copy is needed. This also happens in QR decomposition. So, instead, if we only need L or Q, we can overwrite it!

Fast Randomized SVD Algorithm

N , P = A.shape, **iterations** = 5
Q = **randomVector**(size = (P, K))
For iteration in range(**iterations**):
    Q, P = **GETRF** (X @ Q) (1)
    Q = **LASWP** (Q, P) (2)
    Q, P = **GETRF** (X.T @ Q)
    Q = **LASWP** (Q, P)
Q = **GEQRF** (X @ Q) (3)
Q = **ORGQR** (Q) (4)
U*, S, VT = **Fast-SVD** (Q.T @ X)
U = Q@U*
Return U, S, VT

(1.) LU Decomp. P is permutation vector. (2.) Permute L (3.) QR Decomp. (4.) Get Q, discard R.



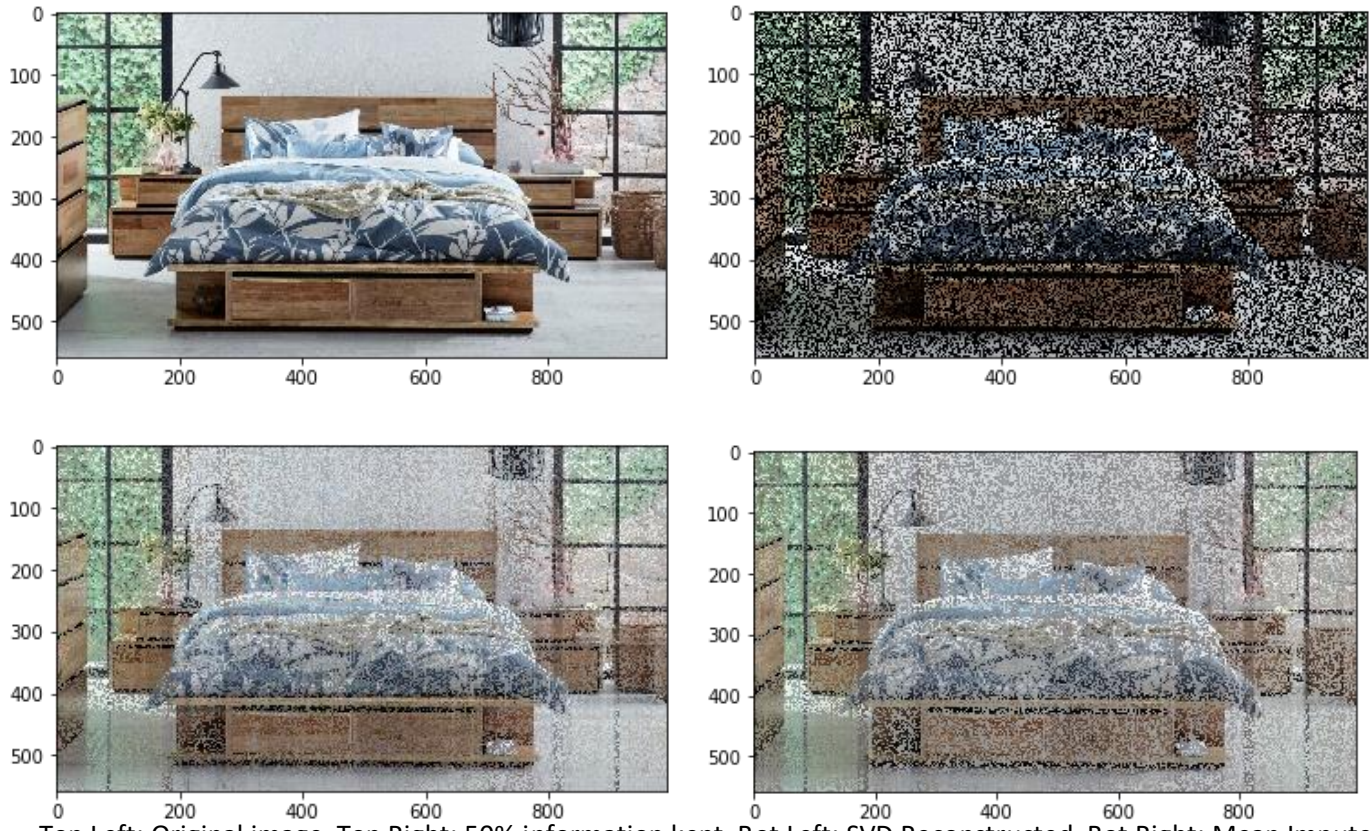
Clearly, on the left, outputting only L or Q from the decompositions is much faster. Likewise, notice how I changed the iterations from the default 7 as suggested by (Halko, 2009) to now 5, as I can clearly see the error is not reducing even after 2 iterations!

d.Reconstruction SVD

• *How can I reconstruct images or data that have missing pixels or information?*

The final issue I have not addressed! In real world data, missing values is always seen. So, how do we reconstruct the data to its best original state by using the non-null data cells? This problem is called **missing value imputation** or **inference**.

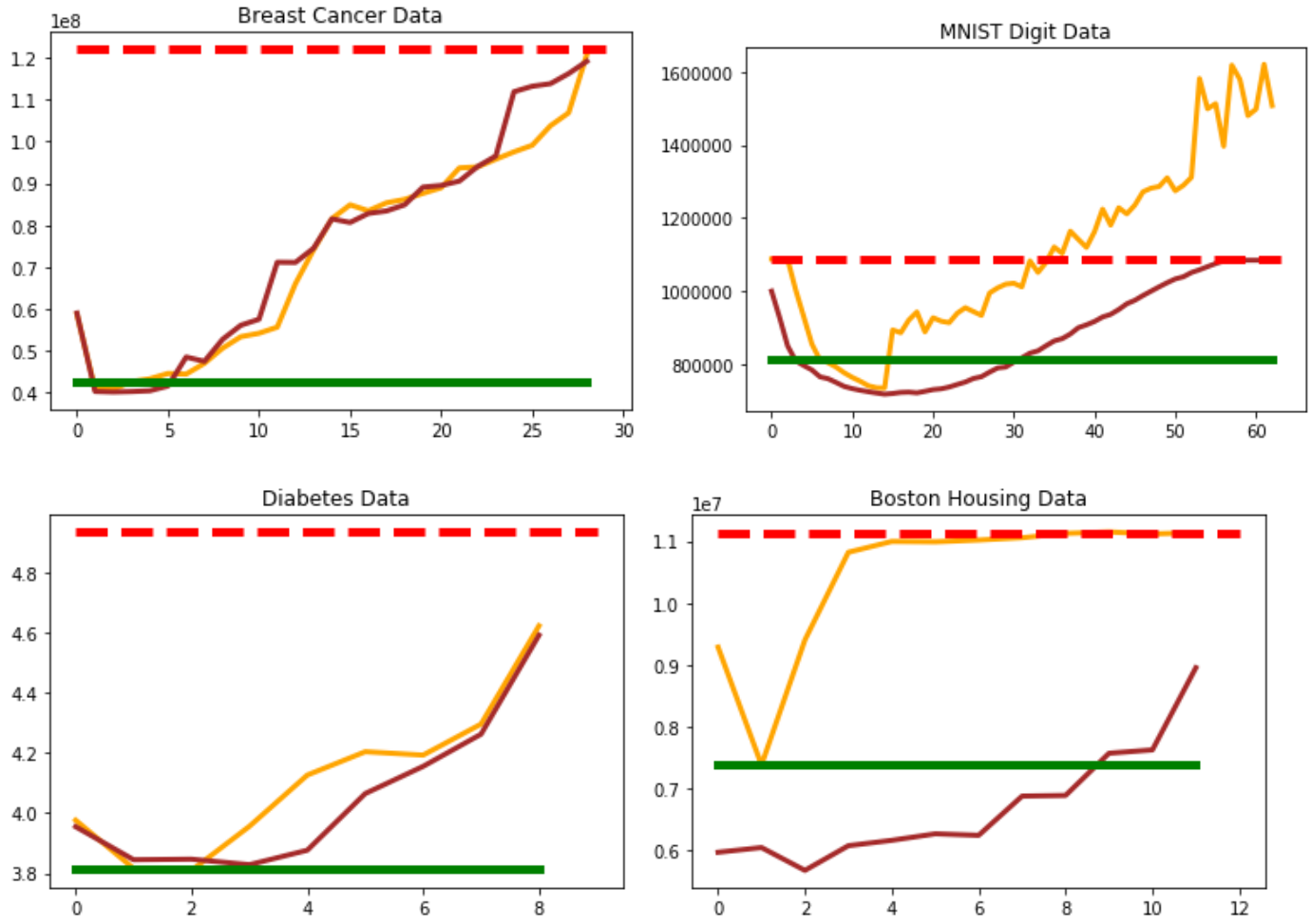
One of the most basic algorithms is called **mean column imputation**, which just fills each missing value with it's column mean or median. This method is easy and fast, but only considers the information within 1 column. It doesn't care about the information in each row.



Top Left: Original image. Top Right: 50% information kept. Bot Left: SVD Reconstructed. Bot Right: Mean Impute

By using the above real-life example, you can see that using only column mean imputation produces a blurrier image on the one reconstructed with SVD (a new method). First notice that the SVD reconstruction of the image uses mean imputation as its initial conditions, then through Randomized SVD, produces the top K components, and reconstructing the matrix via $\hat{X} = U_{:K}\Sigma_{:K}V_{:K}^{T}$.

This method is powerful, as SVD considers both rows and columns in conjunction.



Above showcases the error of reconstruction from the ground truth using SVD. **GREEN** shows the new heuristic (number of components is automatically chosen) on 50% of the training data. **BROWN** shows the full dataset if used instead. **ORANGE** shows 50% of the training data where the number of components change. **RED** is error where only mean of columns is used. The lower, the better.

Notice how I mentioned a heuristic! In the real world, we **cannot analyse error spectrums**, so I must provide a guess to how many components we should use. Using a full reconstruction yields the same results as normal mean impute and using too few components isn't helpful. So, after checking on many datasets, the best heuristic is:

$$k = number\ of\ components = \min\left(\sqrt{p} - 1, 1\right)$$

Also, I noticed that Hastie and co (Hastie, 2014) produced a similar approach. However, their method is limited, as given **test data**, (ie data not seen), their SVD approach cannot work, as the old data AND new test data must be recombined and imputed together (which is clearly wasting resources).

Also, their methods are relatively slow, as you need multiple iterations to get the converged solution. However, my new algorithm can be performed quickly.

SVD Reconstruction Algorithm - Train

```
n, p = X.shape
K = min( sqrt( p ) – 1, 1 )
mask = isnull(X)                                    (1)
mean = nanmean(X)                                   (2)
std = nanstd(X)
X -= mean; X /= std
X[mask] = 0
U, S, VT = Fast – RandomizedSVD ( X, K )
Xhat = U * S @ VT
Xhat *= std; Xhat += mean
Return Xhat, S, VT, mean, std
```

(1.) Boolean matrix showing where null values are (2.) Calculates mean disregarding null values.

In order to transform a new test data matrix, the below algorithm is executed:

SVD Reconstruction Algorithm - Test

```
n, p = X.shape
K = min( sqrt( p ) – 1, 1 )
mask = isnull(X)
X -= mean; X /= std
X[mask] = 0
newX = vstack ( S*VT ,  X  )                        (1)
U, S, VT = Fast – RandomizedSVD ( X, K )
Xhat = U[K:, :K] * S[:K] @ VT[:K]
Xhat *= std; Xhat += mean
Return Xhat
```

(1.) **Vstack** means to vertically stack matrices together.

Notice how we vertically stack the training Σ, V^T together, courtesy of **incremental SVD / PCA**, as described by Ross and co. (Ross, 2008) This allows us to utilise the old learnt singular values and eigenvectors, and project the new data into the space.

Clearly, this is much faster than stacking the 2 train and test matrices and relearning. This will also cause other **pipeline problems** as well. (Say we created a machine learning model on the training data. If we re-fit the SVD on the entire dataset, then we have to re-train the ML model, which is infeasible and silly).

To calculate the complexity of this algorithm, we split the analysis into:

1. Mean, Standard Deviation, Mask array $\boldsymbol{3np^2}$
2. Inplace updating of matrix $\boldsymbol{3np^2}$
3. RandomizedSVD $\boldsymbol{13npk + 7nk^2 - 4k^3 + 7kp^2 + 11p^3}$ (iterations = 5)
   Also, $k \approx \sqrt{p}$, so complexity is $\boldsymbol{13np\sqrt{p} + 7np - 4p\sqrt{p} + 7\sqrt{p}p^2 + 11p^3}$
4. Reconstruction $\boldsymbol{2n\sqrt{p} + np\sqrt{p}}$

Notice how the Reconstruction step is NOT $nk + nk^2 + np\sqrt{p} = n\sqrt{p} + np + np\sqrt{p}$. Clearly, when we perform $U \times \Sigma$, we do not have to perform matrix multiplication at all! We notice quite easily how $U \times \Sigma = U \circ \Sigma$, which is the **Hadamard Product**, or **element wise multiplication**. Since the singular values are a diagonal matrix, it is equivalent to computing element wise multiplication rather than a full matrix multiply.

$$nk + nk^2 + np\sqrt{p} = n\sqrt{p} + np + np\sqrt{p} > 2n\sqrt{p} + np\sqrt{p}$$
$$n\sqrt{p} + np > 2n\sqrt{p}$$
$$np > n\sqrt{p}$$
$$p > \sqrt{p}$$

So, the Hadamard Product can reduce computation times! Likewise, if we have a new test dataset, assume it is of the same size as the training n.

1. Mean, Standard Deviation, Mask array $\boldsymbol{3np^2}$
2. Inplace updating of matrix $\boldsymbol{3np^2}$
3. RandomizedSVD $\boldsymbol{13(n+k)pk + 7(n+k)k^2 - 4k^3 + 7kp^2 + 11p^3}$
   (n+k) is seen, since vstack stacks the 5 scaled eigenvectors on top.
   So, complexity is $\boldsymbol{13(n+\sqrt{p})p\sqrt{p} + 7(n+\sqrt{p})p - 4p\sqrt{p} + 7\sqrt{p}p^2 + 11p^3}$
4. Reconstruction $\boldsymbol{2(n+\sqrt{p})\sqrt{p} + (n+\sqrt{p})p\sqrt{p}}$

Clearly the formulas look yuck. Take $\boldsymbol{(n+\sqrt{p}) = N}$
Then, $13Np\sqrt{p} + 7Np - 4p\sqrt{p} + 7\sqrt{p}p^2 + 11p^3$, $2N\sqrt{p} + Np\sqrt{p}$ is seen. Notice how normally K is exceptionally small when compared to n or p, hence when considering large scales, the testing and training complexities are nearly identical.

However, Hastie's algorithm is clearly slower and computationally more expensive. Assuming they perform i iterations, and use **normal SVD**, then their complexity still explodes. Notice they **DO NOT** use the heuristic $k = \min(\sqrt{p} - 1, 1)$, but rather p.

1. Mean, Standard Deviation, Mask array $\boldsymbol{3np^2}$
2. Inplace updating of matrix $\boldsymbol{3np^2}$
3. Normal SVD: $\boldsymbol{2np^2 + 11p^3}$
4. Reconstruction $\boldsymbol{2np + np^2}$

So, Hastie's algorithm takes $9np^2 + 11p^3 + 2np\ FLOPS$, which is not nice at all.

On testing data, you have to recompute the entire matrix with the old data, thus tripling the final FLOPS to be $27np^2 + 33p^3 + 6np$. Clearly, mine takes approx. $6np^2 + 14np\sqrt{p} + 7np - 4p\sqrt{p} + 7p^2\sqrt{p} + 11p^3 + 2n\sqrt{p}$ to train, and hence double this for testing and training combined.

2 Linear Least Squares

- *How do I find the best linear model to map features $X \rightarrow y$ quickly and efficiently?*
- *Can we increase a linear model's robustness to outliers automatically?*
- *How do I obtain the exact least squares solution if data comes in streams?*

Linear Least Squares is a class of linear algebra techniques that can solve these 3 problems! It maps some input features X to targets y. Pretend you wanted to predict a countries' GDP from say it's Income Inequality, Unemployment and Population. Then, we build a linear model such that:

$$\_Income\ Inequality + \_Unemployment + \_Population = GDP$$

$$X = [Income\ Inequality \quad Unemployment \quad Population]$$

$$y = GDP$$

$$X \overset{\theta}{\rightarrow} y$$

Notice the __ underscores. This is where we want some arbitrary numbers to multiply each quantity to make Income. Say maybe this is plausible:

$$-10.3 Income\ Inequality - 3.4 Unemployment + 5.4 Population = GDP$$

In fact, we want to find these coefficients, which we call θ.

$$\theta = \begin{bmatrix} -10.3 \\ -3.4 \\ 5.4 \end{bmatrix}$$

But, how do we find these coefficients? Well, we get lots of data! We get pairs of X, y for a lot of countries, say for example:

$$X = \begin{bmatrix} 3 & 6.6 & 10{,}000{,}000 \\ 4 & 5.5 & 20{,}000{,}000 \\ 2 & 5.4 & 50{,}000{,}000 \end{bmatrix}, \quad y = \begin{bmatrix} 40{,}000{,}000 \\ 50{,}000{,}000 \\ 180{,}000{,}000 \end{bmatrix}$$

Notice each column in X signifies each column feature (Income Inequality, Unemployment, Population), and y signifies GDP.

So, we want to build a linear model satisfying:

$$y = \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 + \cdots + \theta_p x_p + \varepsilon$$

Where ε is the errors which are normally distributed: $\varepsilon \sim \boldsymbol{N}(0, \sigma^2)$

In linear algebra terms:

$$y = \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 + \cdots + \theta_p x_p + \varepsilon = X\theta + \varepsilon$$

$$y = X\theta + \varepsilon$$

Hence, we want to estimate θ. So, to get estimates:

$$\hat{y} = X\hat{\theta}$$

To solve this, we notice the squared error criterion, which I used a lot before.

$$\hat{y} = X\hat{\theta}$$

$$Error(y, \hat{y}) = \sum (y - \hat{y})^2$$

$$Error(y, \hat{y}) = \sum \left(y - X\hat{\theta}\right)^2$$

So, we can find $\hat{\theta}$, by simply differentially and setting the error derivative to 0:

$$\frac{\partial\, Error(y, \hat{y})}{\partial\, \hat{\theta}} = -\sum X^T \left(y - X\hat{\theta}\right)$$

$$0 = \frac{\partial\, Error(y, \hat{y})}{\partial\, \hat{\theta}} = \sum X^T y - X^T\, \hat{\theta}$$

$$0 = \sum X^T y - X^T X\, \hat{\theta}$$

$$0 = nX^T y - nX^T X\hat{\theta}$$

$$X^T X\hat{\theta} = X^T y$$

$$\hat{\theta} = (X^T X)^{-1} X^T y$$

So, we have derived the linear least squares solution! Notice a fundamental problem is calculating the inverse. The inverse approximately takes $O(n^3)$ time via **Gauss Elimination,** but with blockwise inversion, this can be reduced to $\Omega(n^2 \log n)$. (Raz)

But, that's not the main problem. The inverse **cannot be computed** if:

1. The matrix $X^T X$ is **NOT** invertible:
    a. If the $determinant\ or\ \det(X^T X) = 0$
    b. If there are linearly dependent columns.

In practice, matrices always become NOT invertible, and so we need another way to calculate the inverse. However, if it does work, then in total, $X^T X$ takes $np^2\ FLOPS$, inverse $\frac{2}{3}p^3$, $X^T y\ np$ and $(X^T X)^{-1} X^T y\ p^2$. So total $np^2 + \frac{2}{3}p^3 + np + p^2\ FLOPS$.

a. Moore Penrose Pseudoinverse

The most popular algorithm to solve the linear least squares solution isn't a direct inverse of $X^T X$, but rather using SVD! The reason is because SVD is much more numerically stable, and easily bypasses the uninvertible problems. So what is the SVD Least Squares Algorithm?

$$\hat{\theta} = (X^T X)^{-1} X^T y$$

$$X = U\Sigma V^T$$

$$X^T X = V\Sigma U^T U\, \Sigma V^T = V\Sigma^2 V^T$$

$$(X^T X)^{-1} = V\Sigma^{-2} V^T$$

$$(X^T X)^{-1} X^T y = V\Sigma^{-2} V^T V\Sigma U^T y$$

$$(X^T X)^{-1} X^T y = V\Sigma^{-2}\Sigma U^T y$$

$$(X^T X)^{-1} X^T y = V\Sigma^{-1} U^T y$$

Clearly, since SVD exists for all matrices, the above formula highlights how SVD based Linear Least Squares works on **ALL matrices.** However, the time complexity and FLOP count isn't great. SVD takes $2np^2 + 11p^3\ FLOPS$, Σ^{-1} takes $p\ FLOPS$, and the multiplication takes $p^3 + np^2 + np$.

So, in all, $3np^2 + 12p^3 + np\ FLOPS$ is needed, which is clearly not good at all.

Notice above, we can make the multiplication faster if we use **Hadamard Products** and **Matric Chain Multiplication (Dynamic Programming)**. Clearly, computing $U^T y$ first is more efficient, so it should be:

$$\hat{\theta} = (V\Sigma^{-1}(U^T y))$$

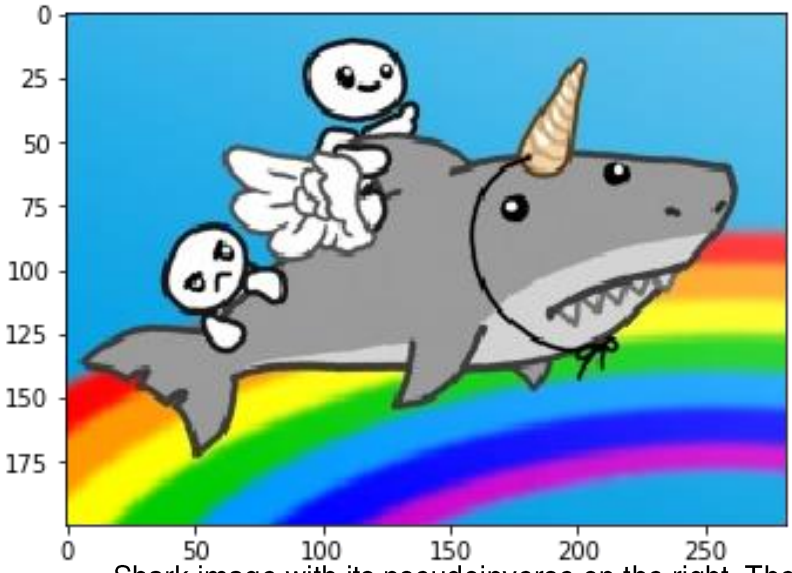
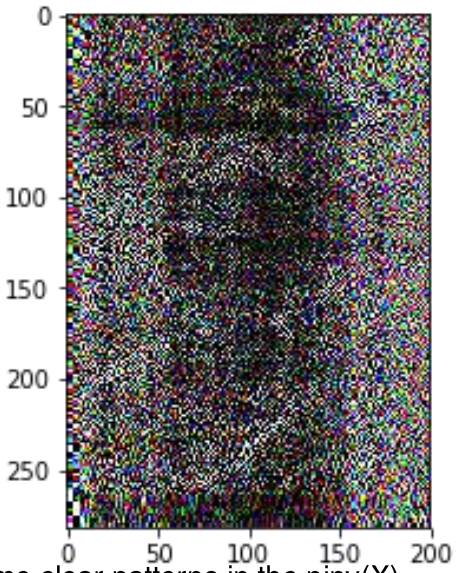
So, now the multiplication steps are reduced to $p^3 + p^2 + np$, so np^2 is shaved off. Notice Σ^{-1} is a diagonal matrix, hence:

$$V\Sigma^{-1} = V \circ \Sigma^{-1}$$

Where the dot is the **element wise Hadamard Product** mentioned previously. The complexity drops further to $2p^2 + np$, so p^3 is converted to a p^2.

Hence, final complexity is $2np^2 + 2p^2 + np + 11p^3$ for:

$$\hat{\theta} = (V \circ \Sigma^{-1}(U^T y))$$

Shark image with its pseudoinverse on the right. There are some clear patterns in the pinv(X).

Notice, the **pseudoinverse** of a matrix is in fact:

$$X^{\dagger} = \boldsymbol{pinv}(X) = V \circ \Sigma^{-1} U^{T}$$

The pseudoinverse symbol is called “X dagger”. It provides the best approximation st

$$XX^{\dagger} \approx I_{n,n}$$

$$X^{\dagger}X \approx I_{p,p}$$

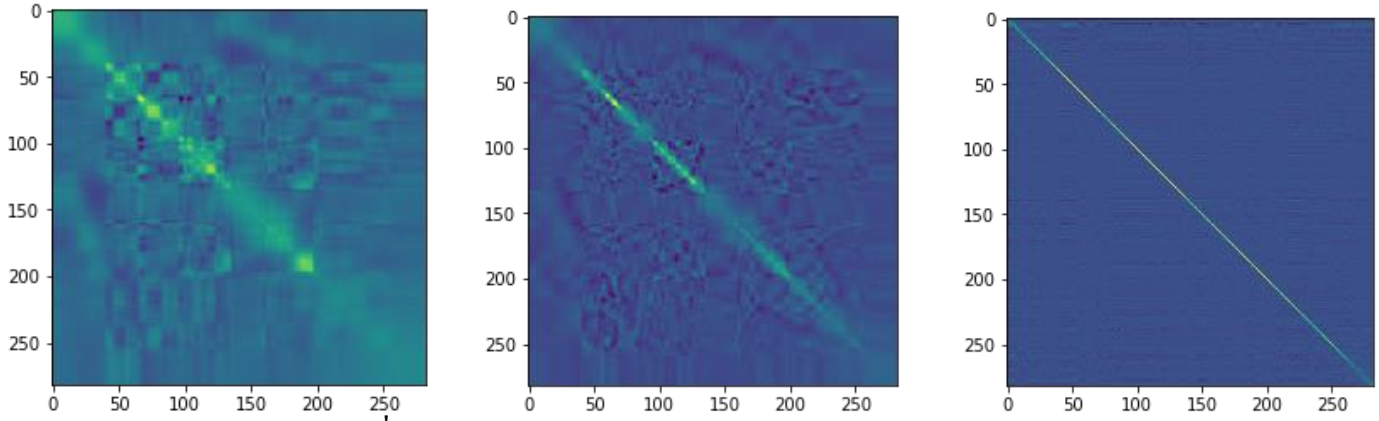
The pseudoinverse exists for all matrices, courtesy of the SVD existing for all matrices. In fact, it extends or generalises the normal inverse, since:

$$XX^{-1} = I_{n,n} = X^{-1}X$$

Notice instead, the pseudoinverse finds the **left and right inverses**. Clearly, this is extremely powerful, as now, an inverse exists for **ALL MATRICES**!!! Hence:

$$\hat{\theta} = (V \circ \Sigma^{-1}(U^{T}y)) = X^{\dagger}y$$

You may also notice how in **Reconstruction SVD**, we approximated the reconstruction using $\hat{X} = U_{:K}\Sigma_{:K}V_{:K}^{T}$. Well, we can do the same for $X^{\dagger}$! Clearly, $\widehat{X^{\dagger}} = V_{:K}\Sigma_{:K}^{-1}U_{:K}^{T}$, so choose the top ½ components to approximate the pseudoinverse.



Top 10 components $XX_{:10}^{\dagger}$, then top 10 and finally all components. Should look like 1 diagonal.

b.Ridge Regularization via SVD

- *Can we increase a linear model's robustness to outliers automatically?*

Now, let us answer this question! The fundamental problem of closed form linear regression via SVD is that the solution vector isn't very robust to large changes in coefficients. Pretend this occurs:

$$1000000A + 20B - 10203C = Y$$

Clearly, a small change in A vastly alters the entire solution dramatically. So, how do we adjust the coefficients such that the largeness of them is penalised? IE: Can we produce coefficients such that the squared error is reduced AND the coefficient magnitude is reduced? In terms of an optimisation statement:

$$Error_{robust}(y, \hat{y}) = \sum \left(y - X\hat{\theta}\right)^2 + \sum \hat{\theta}^2$$

So, we want to minimise this new error to obtain the robust coefficients. This is called **Ridge Regression**, **L2** or **Tikhonov Regularization**. (Tikhonov, 1977)

The closed form solution is:

$$\hat{\theta}_{L2} = \left(X^T X + \alpha I_{p,p}\right)^{-1} X^T y$$

Where α is a chosen **regularization parameter**. It can be any number, normally chosen as 1 or 0.1.

Notice how we can express the top in SVD:

$$\left(X^T X + \alpha I_{p,p}\right)^{-1} X^T y = \left(V\Sigma^2 V^T + \alpha I_{p,p}\right)^{-1} V\Sigma U^T y$$

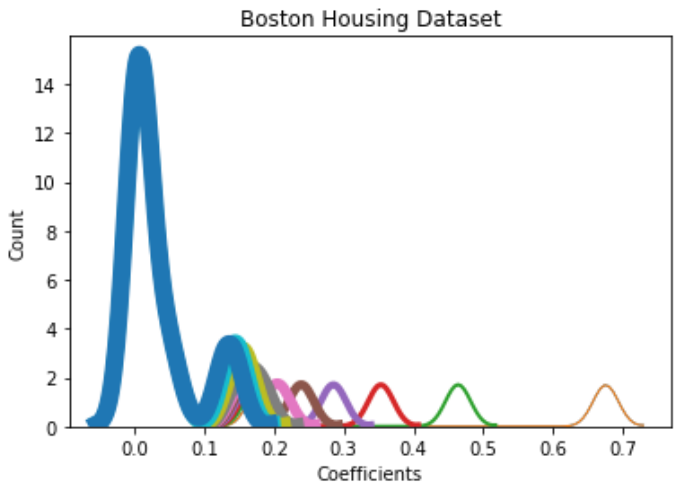
$$= V\left(\Sigma^2 V^T + \alpha I_{p,p} V^T\right)^{-1} V\Sigma U^T y$$

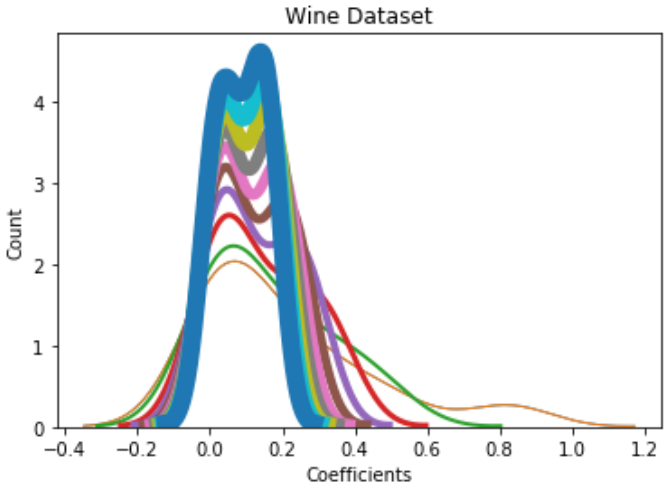
$$= V\left(\Sigma^2 + \alpha I_{p,p}\right)^{-1} V^T V\Sigma U^T y$$

$$= V\left(\Sigma^2 + \alpha I_{p,p}\right)^{-1} \Sigma U^T y$$

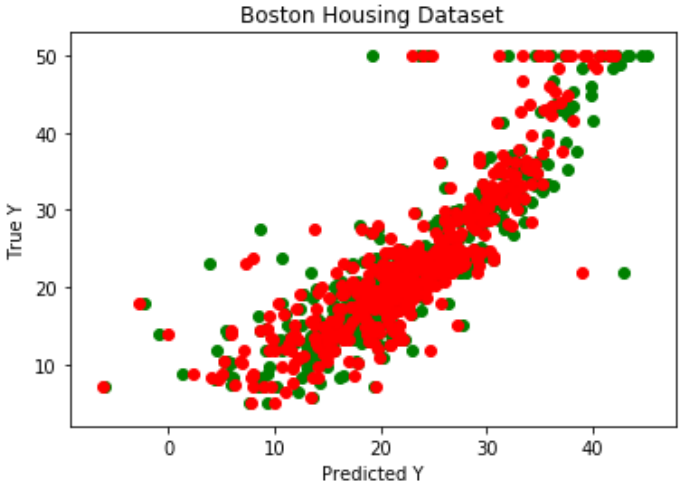
$$= V \frac{\Sigma}{\Sigma^2 + \alpha I_{p,p}} U^T y$$

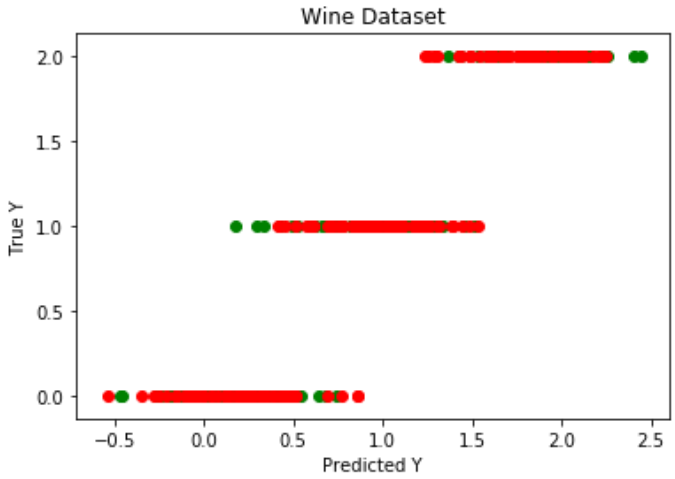
Hence, we can see the singular values are **reduced** or **shrunken**. This is the power of Ridge Regularization, as it penalises larger singular values, which capture a lot of information.





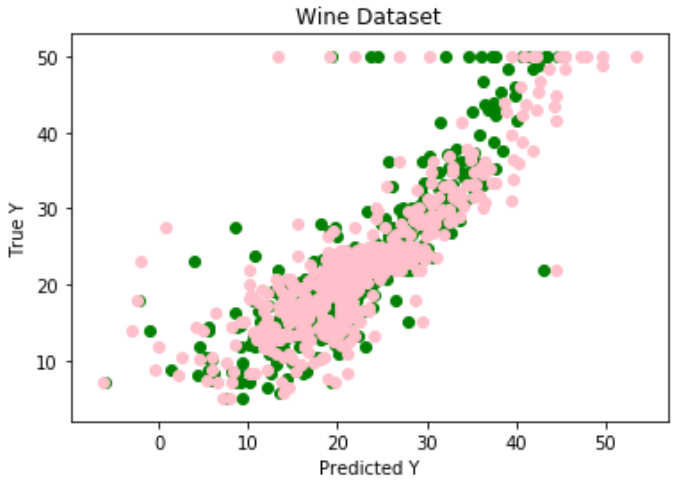
As you can see, Ridge Regularization shifts the coefficient distribution to become more "even". Before, many coefficients were very large, and Ridge penalised these, by shrinking the large singular values down. You can see the dark shaded distributions corresponding to $\alpha = 10$ showcase more equality of magnitudes.





Likewise, after choosing $\alpha = 100$, the prediction vs true value plots don't look too different. Green = no L2 regularization. Red = regularization.

Let's try something novel! What happens if we make the singular values more powerful? Instead we perform **reverse L2 regularization**, thus minimising:



$$Error_{robust}(y, \hat{y}) = \sum \left(y - X\hat{\theta}\right)^2 - \sum \hat{\theta}^2$$

Notice we can see in the graph it has the opposite effect.

Clearly, compare the top right. The predicted values diverge to the right, whilst in L2 Regularization, they diverge to the left. Anyways, this was just for fun!

c. Epsilon Jitter Cholesky Solver

- *How do I find the best linear model to map features $X \rightarrow y$ quickly and efficiently?*

Clearly, using SVD to solve the linear least squares problem is universally accepted, but **SLOW**. Very slow. So, our problem is 3 fold:

1. Can we just use $O(n^3)$ **Gauss Elimination** to compute the inverse?
2. How can we enforce robustness effectively and automatically?
3. How can we force non-invertible matrices to be invertible?

Well, we already solved 1 problem – 2! Essentially to answer the other 2:

1. Yes!
2. Already can be done via **Ridge Regularization**.
3. Yes!

The algorithm which can do this the **Epsilon Jitter Cholesky Least Squares Solver**. It does the following:

1. Start with no regularization
2. Add epsilon jitter regularization until a solution is found.
3. Solve via Gauss Elimination.

Epsilon Jitter Cholesky Least Squares Solver

```
Alpha = 0
While no solution:
        Add regularization
        Theta = Solve ( X, y, alpha )
        Break if solution
        Else:
                If Alpha == 0: Alpha = 0.0001
                Else: Alpha *= 10
Return Theta
```

Notice this looks suspiciously similar to my old **Epsilon PCA** algorithm! Well it's the same, but wrapped for different use cases! In fact, this algorithm is so efficient, that I am a bit shocked it provides close to no error changes (some machine precision errors), and also does automatic ridge regularization (the jitter / alpha acts as it).

In fact, notice how $X^T X + \alpha I$ is a **symmetric matrix**! We proved this way back. Clearly, to compute the inverse, we should use specialised decomposition methods. LU Decomposition is normally used, where $\frac{2}{3}n^3$ FLOP operations are used.

But, because of this symmetrical property, we can use **Cholesky Decomposition**! This factors a symmetric matrix into either a Lower OR Upper Triangular Matrix:

$$X = U^T U = LL^T$$

The FLOP count is now $\frac{1}{3}n^3\ FLOPS$ or ½ that for LU Decomposition! This is phenomenal! Likewise to get the inverse:

$$X^{-1} = (U^T U)^{-1} = U^{-1}(U^T)^{-1}$$

$$X^{-1} = U^{-1}(U^{-1})^T$$

Clearly, the inverse on U, which is an upper triangular matrix can be easily found, as **back substitution** is cheaply done. Also notice:

$$(X^T X + \alpha I)^{-1} X^T y = (U^T U)^{-1} X^T y$$

$$= U^{-1}(U^{-1})^T X^T y$$

Likewise, we can just apply the back substitution on the normal equations:

$$U^T U = X^T y$$

This allows the final complexity to be a mere $np^2 + \frac{1}{2}p^3 + np + p^2\ FLOPS$. Compared with LU Decomposition, a total $np^2 + \frac{2}{3}p^3 + np + p^2\ FLOPS$ is needed. Notice the $\frac{1}{2}p^3$ versus the old $\frac{2}{3}p^3$. But using back substitution, we can further chop the complexity even further. In **LAPACK**, **POTRF** is for Cholesky decomposition, **POTRS** is for Cholesky Back Substitution.

Clearly, what I aim to do is:

$$\lim_{\alpha \to 0} (X^T X + \alpha I) \to X^T X$$

So, we start off with some $\alpha = 0.0001$. We keep multiplying by 10 until we find a solution. This essentially causes **linearly dependent** columns to become uncoupled.

$$X = \begin{bmatrix} 1 & 2 & -1 \\ 2 & 4 & -2 \\ 1 & 2 & -1 \end{bmatrix}$$

Clearly the above matrix is **NOT invertible**. Column 1 = ½ Column 2 = -1 Column 3. This is a linearly dependent matrix. So, we enforce linear independence by adding 0.0001 along the diagonal:

$$X = \begin{bmatrix} 1.0001 & 2 & -1 \\ 2 & 4.0001 & -2 \\ 1 & 2 & -1.0001 \end{bmatrix}$$

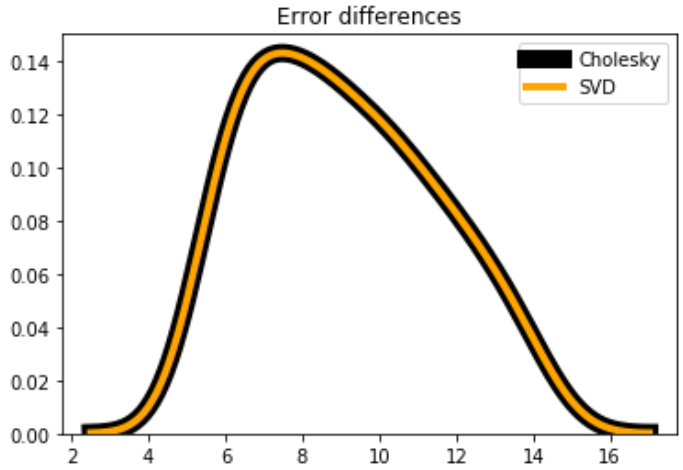
This forces the matrix to be linearly independent, which is the crux of this algorithm.

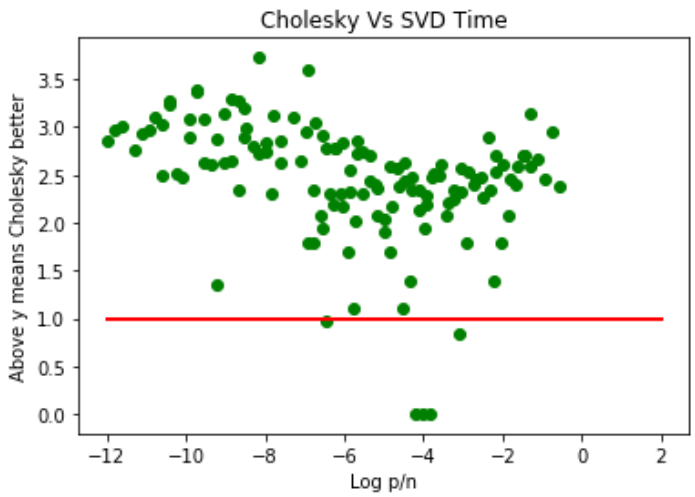
Epsilon Jitter Cholesky Least Squares Solver

```
Alpha = 0
XTX = SYRK ( X.T, trans = 0 )                                    (1)
While no solution:
        U = POTRF ( XTX )                                        (2)
        If failure:
                Diagonal ( XTX ) -= Alpha
                If Alpha == 0: Alpha = 0.0001
                Else: Alpha *= 10
                Diagonal ( XTX ) += Alpha                        (3)
                Continue
        Theta = POTRS ( U, y )                                   (4)
Diagonal ( XTX ) -= Alpha
Return Theta
```

(1.) SYRK: BLAS fast symmetric multiplication. (2.) POTRF: Cholesky Decomposition. (3.) Add alpha to diagonal. (4.) Back substitution solve.

Notice in the above algorithm how we add and minus alpha continuously. We have to do this or else the matrix would get added alpha multiple times. Furthermore, a good feature of this algorithm is it's memory usage or space complexity. SVD takes $4\min(n,p)^2 + \max(n,p) + 9\min(n,p)$ space. (Scipy, 2018). On the other hand, Cholesky performs in-place, so only $2p^2$ or less memory is needed.





Clearly, the errors between Epsilon Cholesky and SVD is nearly invisible, In fact, this was a on a log scale, so it should exacerbate small differences. Timing showcases Cholesky is consistently as least 2 to 3 times faster than SVD on a log scale, so actually, it's e^2 to e^3 times faster!

So it's quite clear that the Epsilon Jitter Cholesky Solver has multiple advantages:

1. Is **faster by 7.4 to 20.1 times** and uses only $np^2 + \frac{1}{2}p^3 + np + p^2\ FLOPS$
2. Automatic ridge regularization is seen with limiting behaviour of epsilon jitter.
3. Has no error differences, and uses only $2p^2$ memory.

d. Streaming Algorithms & Gradient Descent

- *How do I obtain the exact least squares solution if data comes in streams?*

A **streaming algorithm** is one that iteratively finds a solution after seeing some new data. It needs to satisfy 3 things:

1. Does not store the full data matrix in memory.
2. Can produce a good solution without even seeing the new data.
3. Can easily produce the solution with no complex data structures.

I have already introduced Haiko, Ross and co's (Halko, 2009) (Ross, 2008) **Incremental SVD / PCA**, which does just this! We can clearly use SVD to calculate the linear least squares solution, but there must be a better way!

By referring to Andrew Ng, (Ng, 2009), Gradient Descent can be used to iteratively find the solution to the least squares problem. Remember:

$$0 = \frac{\partial\, Error(y, \hat{y})}{\partial\, \hat{\theta}} = \sum X^T y - X^T \hat{\theta}$$

$$0 = \sum X^T y - X^T X \hat{\theta}$$

$$X^T X \hat{\theta} = X^T y$$

Well, instead of computing the inverse, we can perform gradient updates, where:

$$\hat{\theta}_{t+1} \Leftarrow \hat{\theta}_t - \lambda\, X^T \left(y - X\hat{\theta}_t\right)$$

Notice $\Leftarrow$ means to update, and λ is the step size or learning rate, normally chosen to be less than 1 (0.1, 0.001). This allows us to only perform matrix multiplies and additions, without ever having to perform matrix inversions or decompositions. However, you have to do this iteratively for quite a long time (maybe >= 100 iterations) until a solution has been well defined. If you notice the above:

$$\hat{\theta}_{t+1} \Leftarrow \hat{\theta}_t - \lambda\, X^T \left(y - X\hat{\theta}_t\right)$$

$$\hat{\theta}_{t+1} \Leftarrow \hat{\theta}_t - \lambda\left(\, X^T y - X^T X \hat{\theta}_t\right)$$

Now, we want $\hat{\theta}_t = \hat{\theta}_{t+1}$, since at convergence, nothing will change. So:

$$\hat{\theta}_t \Leftarrow \hat{\theta}_t - \lambda\left(\, X^T y - X^T X \hat{\theta}_t\right)$$

$$0 = -\lambda\left(X^T y - X^T X\hat{\theta}_t \right)$$

$$0 = \left(X^T y - X^T X\hat{\theta}_t \right)$$

$$X^T X\hat{\theta}_t = X^T y$$

So essentially, via gradient descent, we can see it can produce the optimal least squares solution is many iterations are provided. Notice we start with an initial guess or initial solution $\hat{\theta}_t$, which can be randomly generated.

Now, **batch gradient descent** is an extension to the above **gradient descent algorithm.** It splits the data into portions (maybe 10), and performs gradient descent on each portion separately, after reducing each step size by 10. Hence:

$$Batch\ i\text{:}\ \hat{\theta}_t \Leftarrow \hat{\theta}_t - \frac{\lambda}{10}\left(X^T y - X^T X\hat{\theta}_t \right)$$

This allows possible faster convergence (you can make the divisor = 1), and is much more robust to outliers, as it averages across all updates. So having B batches:

$$\hat{\theta}_t \Leftarrow \hat{\theta}_t - \frac{\lambda}{B}\left(X^T y - X^T X\hat{\theta}_t \right)$$

$$\hat{\theta}_{t+B} \Leftarrow\Leftarrow \hat{\theta}_t - \lambda \sum_{i=1}^{i=B} \frac{1}{B}\left(X^T y - X^T X\hat{\theta}_{t+i} \right)$$

So, why not use this algorithm for streaming algorithms? This method is very memory efficient, only ever needing to store 1 vector: $\hat{\theta}_t$. This takes only a mere p space, which is phenomenally small. What we do, is after we have the coefficients after seeing a batch of data, we average it with the second batch.

$$\hat{\theta}_{t+1} \Leftarrow \frac{\hat{\theta}_t + \hat{\theta}_{t-1}}{2}$$

Or to be more precise:

$$\hat{\theta}_{t+1} \Leftarrow \frac{n_t\hat{\theta}_t + n_{t-1}\hat{\theta}_{t-1}}{n_t + n_{t-1}}$$

However, we don't want this to occur:

$$\hat{\theta}_{t+1\ FINAL} \Leftarrow \frac{n_t\hat{\theta}_t + n_{t-1}\hat{\theta}_{t-1} + n_{t-2}\hat{\theta}_{t-2} + \cdots + n_0\hat{\theta}_0}{n_t + n_{t-1} + n_{t-2} + \cdots + n_0}$$

As we'll be storing tp memory. Instead, we approximate the solution through updating the number of elements seen directly. Hence:

$$\hat{\theta}_{t+1\ FINAL} \Leftarrow \frac{n_t\hat{\theta}_t + (n_{t-1} + n_{t-2} + \cdots + n_0)\hat{\theta}_{t-1}}{n_t + (n_{t-1} + n_{t-2} + \cdots + n_0)}$$

Essentially, if we store $N = n_{t-1} + n_{t-2} + \cdots + n_0$, then

$$\hat{\theta}_{t+1\ FINAL} \Leftarrow \frac{n_t\hat{\theta}_t + N_{t-1}\hat{\theta}_{t-1\ FINAL}}{n_t + N_{t-1}}$$

Thus, only p memory is needed.

$$\hat{\theta}_{1\ FINAL} \Leftarrow \frac{n_1\hat{\theta}_1 + n_0\hat{\theta}_0}{n_1 + n_0} = \frac{n_1\hat{\theta}_1 + n_0\hat{\theta}_0}{N_1}$$

$$\hat{\theta}_{2\ FINAL} \Leftarrow \frac{n_2\hat{\theta}_2 + n_1\hat{\theta}_1 + n_0\hat{\theta}_0}{n_2 + n_1 + n_0} = \frac{n_2\hat{\theta}_2}{n_2 + n_1 + n_0} + \frac{n_1\hat{\theta}_1 + n_0\hat{\theta}_0}{n_2 + n_1 + n_0}$$

$$Notice \quad N_1\hat{\theta}_{1\ FINAL} = n_1\hat{\theta}_1 + n_0\hat{\theta}_0$$

$$\hat{\theta}_{2\ FINAL} \Leftarrow \frac{n_2\hat{\theta}_2}{n_2 + N_1} + \frac{n_1\hat{\theta}_1 + n_0\hat{\theta}_0}{n_2 + N_1}$$

$$= \frac{n_2\hat{\theta}_2}{n_2 + N_1} + \frac{N_1\hat{\theta}_{1\ FINAL}}{n_2 + N_1} = \frac{n_2\hat{\theta}_2 + N_1\hat{\theta}_{1\ FINAL}}{n_2 + N_1}$$

Hence, we have that:

$$\hat{\theta}_{t+1\ FINAL} \Leftarrow \frac{n_t\hat{\theta}_t + N_{t-1}\hat{\theta}_{t-1\ FINAL}}{n_t + N_{t-1}}$$

This means that we have proven that this holds! We can further evaluate this claim by **induction**, but it is quite clear.

$$Let \quad \hat{\theta}_{k\ FINAL} \Leftarrow \frac{n_k\hat{\theta}_k + \cdots + n_0\hat{\theta}_0}{n_k + \cdots + n_0} = \frac{n_k\hat{\theta}_k + N_{k-1}\hat{\theta}_{k-1\ FINAL}}{n_k + N_{k-1}} \quad be\ TRUE$$

Then prove for k+1

$$\hat{\theta}_{k+1\ FINAL} \Leftarrow \frac{n_{k+1}\hat{\theta}_{k+1} + n_k\hat{\theta}_k + \cdots + n_0\hat{\theta}_0}{n_{k+1} + n_k + \cdots + n_0}$$

$$= \frac{n_{k+1}\hat{\theta}_{k+1}}{n_{k+1} + n_k + \cdots + n_0} + \frac{n_k\hat{\theta}_k + \cdots + n_0\hat{\theta}_0}{n_{k+1} + n_k + \cdots + n_0}$$

But, $\hat{\theta}_{k\ FINAL}(n_k + N_{k-1}) = n_k\hat{\theta}_k + N_{k-1}\hat{\theta}_{k-1\ FINAL} = n_k\hat{\theta}_k + \cdots + n_0\hat{\theta}_0$

$$= \frac{n_{k+1}\hat{\theta}_{k+1}}{n_{k+1} + n_k + \cdots + n_0} + \frac{\hat{\theta}_{k\ FINAL}(n_k + N_{k-1})}{n_{k+1} + n_k + \cdots + n_0}$$

$$= \frac{n_{k+1}\hat{\theta}_{k+1} + n_k\hat{\theta}_{k\,FINAL} + N_{k-1}\hat{\theta}_{k\,FINAL}}{n_{k+1} + n_k + \cdots + n_0} = \frac{n_{k+1}\hat{\theta}_{k+1} + N_k\hat{\theta}_{k\,FINAL}}{n_{k+1} + N_k}$$

Since $n_k + N_{k-1} = N_k$. Clearly, **proof by induction** is seen for all $k \geq 0$.

One clear issue is the size of the batches. If we have a stream of observations, is it wise to update incrementally for each observation? Or should we pool it into some larger group, then update it once say the pool size exceeds say 10?

Above, I used the Ames Housing Dataset courtesy of Kaggle (Cock, 2011) with multiple batch sizes. Green is the new streaming algorithm, and Red is the old popular averaging algorithm. When the batch sizes are small, the new algorithm does slightly worse at the beginning, but stabilises quickly. But, the old algorithm, although having a lower **Root Mean Squared Error** (RMSE) in the beginning stages, fluctuates a lot.

New Streaming Least Squares Algorithm

```
P = number of columns
Theta = zeros ( size = p )
N = 0

Function BatchUpdate ( X, y ):                                    (1)
        n = length ( X )
        Theta_t = Epsilon Jitter Cholesky Solve ( X, y )
        Theta = (Theta*N + Theta_t * n) / ( N + n )
        N += n
```

(1.) Function should be called every time when new data comes in

e. Exact Batch Linear Least Squares

However, notice the new streaming algorithm isn't perfect. You can clearly see how the error stabilises but doesn't seem to reduce further. Instead, if we want to only update say weekly or monthly, a new **Cholesky Exact Algorithm** must be made.

Remember that:

$$X^T X\hat{\theta} = X^T y$$

Well, clearly, if we add more rows, then:

$$X^T X = X_t^T X_t + X_{t+1}^T X_{t+1}$$

$$X^T y = X_t^T y_t + X_{t+1}^T y_{t+1}$$

This is quite clear, since matrix multiplication is just a linear operation, thus separating them does not cause any issues. Hence, if you know this, then we can store only $X^T X$ and $X^T y$, which will take $p^2 + p$ memory only. It's slightly larger than then p memory storage by the previous algorithm, but this method is **guaranteed to produce an exact solution**.

However, one must recompute the Cholesky Decomposition. However, one can also do a **rank-one update** of the Cholesky Decomposition, in which I will not discuss now. Remember it is super cheap to compute this decomposition $\frac{1}{3}p^3\ FLOPS$ is needed. So, recomputing the Cholesky Decomposition shouldn't take that much effort.

Also let us analyse the old Streaming Algorithm:

$$\hat{\theta}_{t+1} \Leftarrow \frac{\hat{\theta}_t + \hat{\theta}_{t-1}}{2}$$

At each step i, if $n = 1$, then $x^T x$ takes $p^2\ FLOPS$. $x^T y$ takes $p\ FLOPS$. Cholesky with inversion takes $\frac{1}{2}p^3\ FLOPS$, and multiplying takes $p^2\ FLOPS$. In total, $\frac{1}{2}p^3 + 2p^2 + p\ FLOPS$ is needed for each observation. (+ p for averaging). So given n iterations:

$$n\left(\frac{1}{2}p^3 + 2p^2 + p + p\right) = \frac{1}{2}np^3 + 2np^2 + 2np\ FLOPS$$

For the newer algorithm:

$$\hat{\theta}_{t+1\ FINAL} \Leftarrow \frac{n_t\hat{\theta}_t + N_{t-1}\hat{\theta}_{t-1\ FINAL}}{n_t + N_{t-1}}$$

FLOPS needed are similar, except we need to consider the explicit extra multiplies:

$$n\left(\frac{1}{2}p^3 + 2p^2 + p + p\right) + n(2p) = \frac{1}{2}np^3 + 2np^2 + 4np\ FLOPS$$

Although both only needs p memory, during the computation of the solution, p^2 memory is still needed. So:

1. During computation: p^2 memory is needed.
2. After computation: p memory is needed.

If this was done on one computer, then $p^2 + p$ memory is needed.

Now, notice our finding that:

$$X^T X = X_t^T X_t + X_{t+1}^T X_{t+1}$$

$$X^T y = X_t^T y_t + X_{t+1}^T y_{t+1}$$

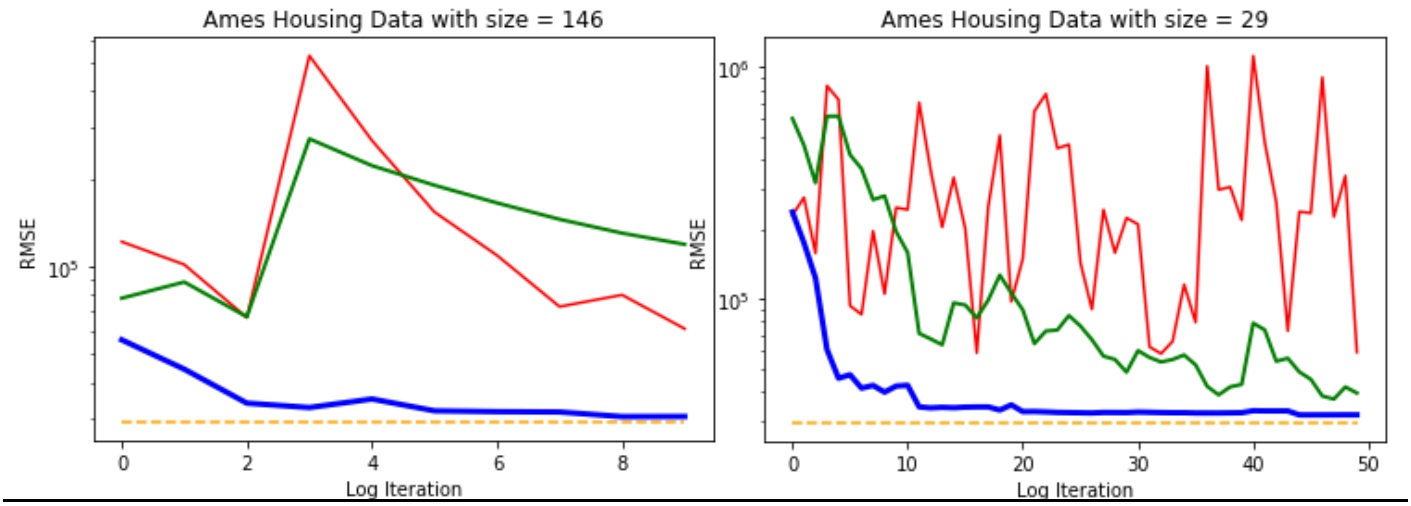
What this means, is why not continuously update $X^T X, X^T y$ directly? $p^2 + p$ memory is needed, and you don't need to store the coefficients.

Exact Batch Linear Least Squares

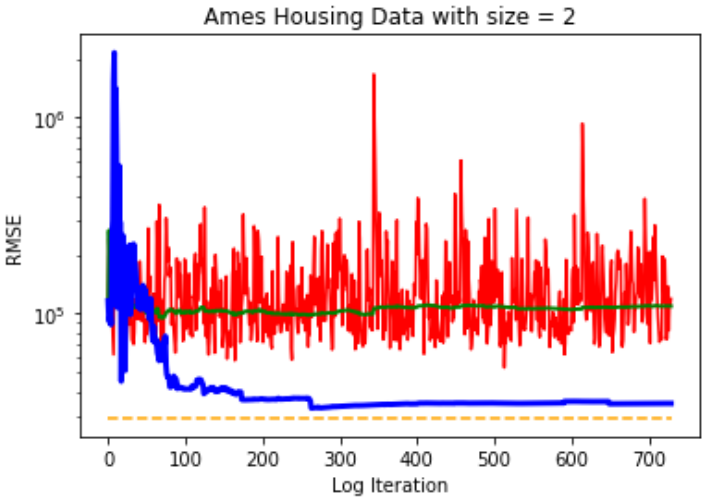
```
XTX = zeros ( shape = (p, p) )
XTy = zeros ( shape = p )

Function ExactUpdate ( x, y ):
        xtx = SYRK ( x.T, trans = 0 )                    (1)
        xty = x.T @ y
        XTX += xtx
        XTy += xty
        U = POTRF ( XTX )                                (2)
        Return POTRS ( U, XTy )                          (3)
```

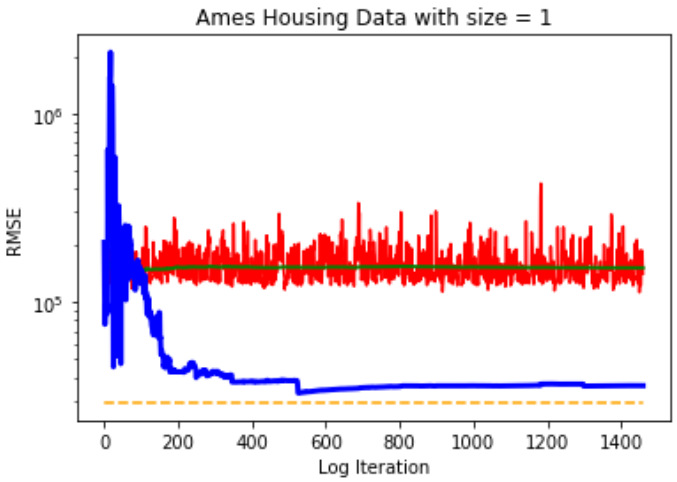
(1.) SYRK: BLAS Symmetric Matrix Multiply (2.) POTRF: Cholesky Decomp (3.) Back Substitution

Clearly, this new **Exact Batch Linear Least Squares** uses $\frac{1}{2}np^3 + 2np^2 + 4np\ FLOPS$, or the same FLOP count as the original algorithm. However, now, during computation AND non-computation steps, $p^2 + p$ memory is needed always.

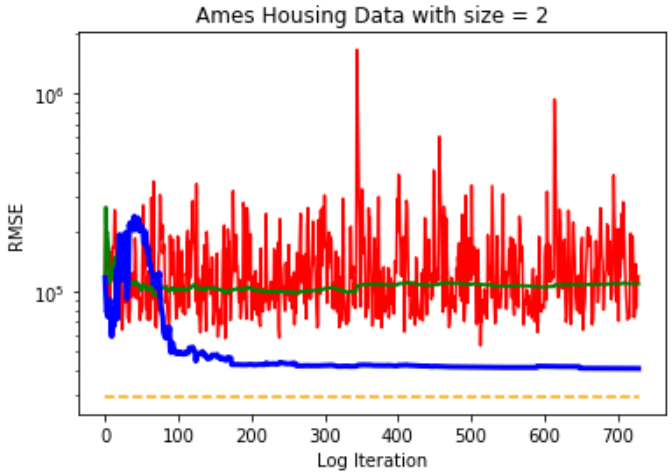


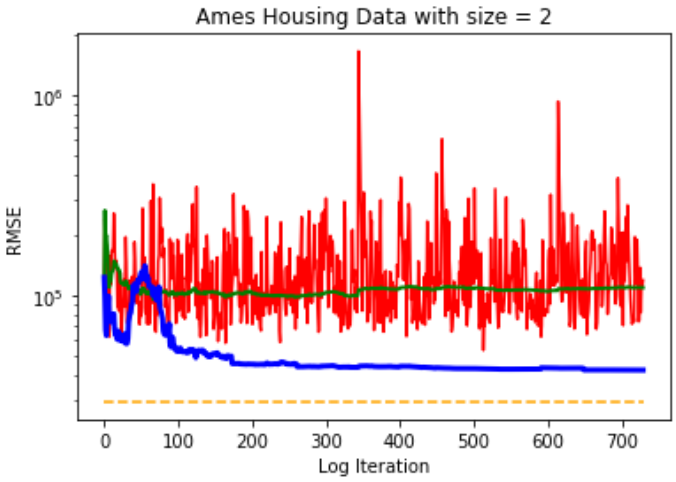
Red is first averaging algorithm. Green is new averaging algorithm. Blue is the exact least squares algorithm. The dotted line is the RMSE for the real final dataset when all is shown. The lower the error, the better.



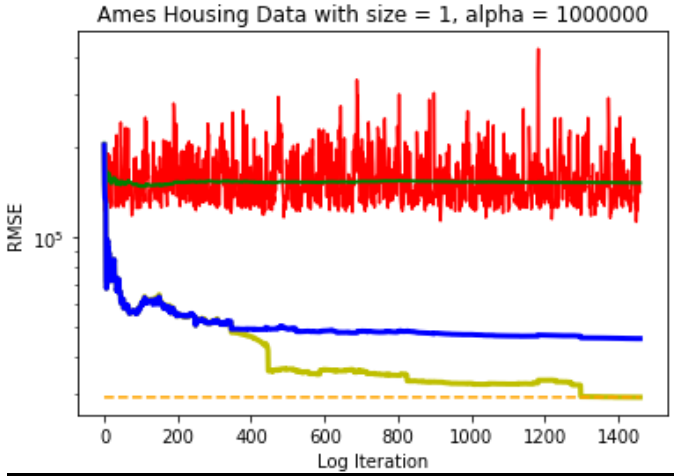


It is quite clear that the Exact Least Squares algorithm does much better in the long run but struggles severely when the observations are small. This is because the solution favours outliers more, and so we need a way to counteract this. So, we use **Ridge Regularization**! Essentially, we add some elements to the diagonal for $X^T X$.



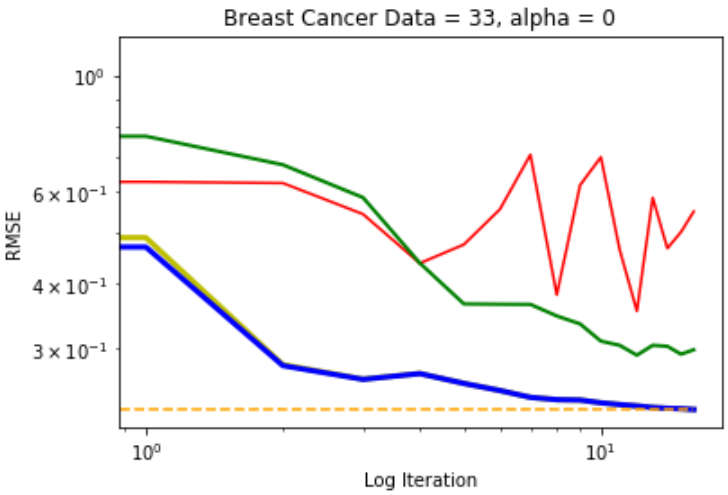


On the left, $\alpha = 10{,}000$. On the right, $\alpha = 100{,}000$. You can see how the fluctuations are controlled, but in the long run, the result goes further away from the optimum.



To fix this issue, at each iteration, we reduce the ridge regularization! This is essentially the **opposite** of **Epsilon Jitter Cholesky Solving**!

However, this method is **NOT GOOD**. In fact, the choice of the ridge parameter is a very important step. So, do not do this.



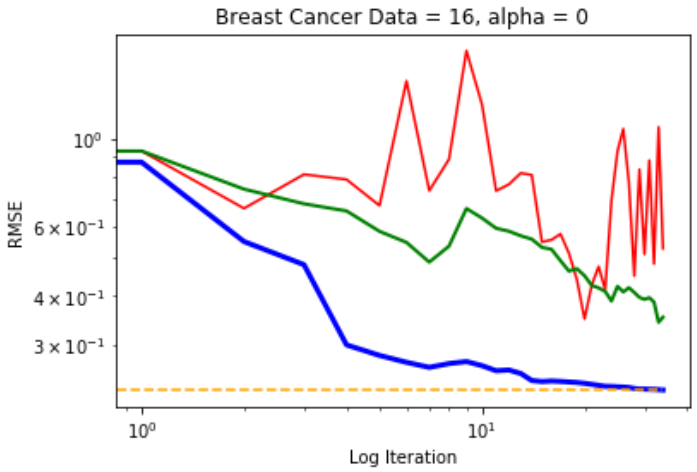
Instead what we do, is we borrow some batch sizes from the training of **Neural Networks**. Andrew Ng says batch sizes should be a power of 2, whereby it can easily be matrix multiplied via divide and conquer must faster. (Ng, 2009)

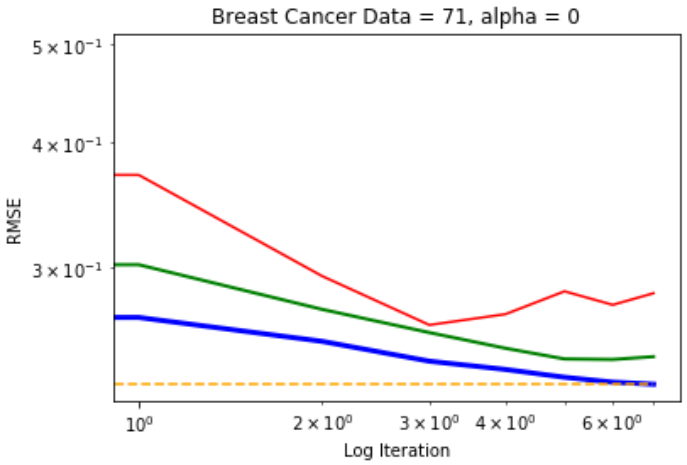
However, we are not exactly choosing a batch size for computational efficiency, but for stability.

So, choose a batch size as a power of two:

$$B = 2^i = (16, 32, 64, 128, \dots, 1024)$$

Above, you can see I chose a batch size of 32. Below, sizes of 16 and 64.





It is very apparent that the errors do not fluctuate anymore. Likewise, the new algorithm still does much better than gradient averaging. So, the final algorithm utilising both Exact Cholesky Solving and optimum batch sizes is as follows:

Batch Size Exact Batch Linear Least Squares

Batch_size = 32 (1)
XTX = zeros (shape = (p, p)) ; XTy = zeros (shape = p)
Current_count = 0 (2)
Theta = zeros (shape = p) (3)
Function **ExactUpdate** (x, y):
    xtx = **SYRK** (x.T, trans = 0)
    xty = x.T @ y
    XTX += xtx ; XTy += xty
    Current_count += len (X)
    If **Current_count** >= **Batch_size**:
        Current_count = 0
        U = **POTRF** (XTX)
        Theta = **POTRS** (U, XTy)
    Return **Theta**

(1.) User defined → should be power of 2. (2.) Temporary global counter. (3.) Extra memory needed.

3 NNMF, Positive Matrix Decomposition

Non-Negative Matrix Factorization or **Positive Matrix Decomposition** is a very popular technique which is used in astronomy, physics, recommender systems, or whenever data is strictly positive (ie: rank data, star signals).

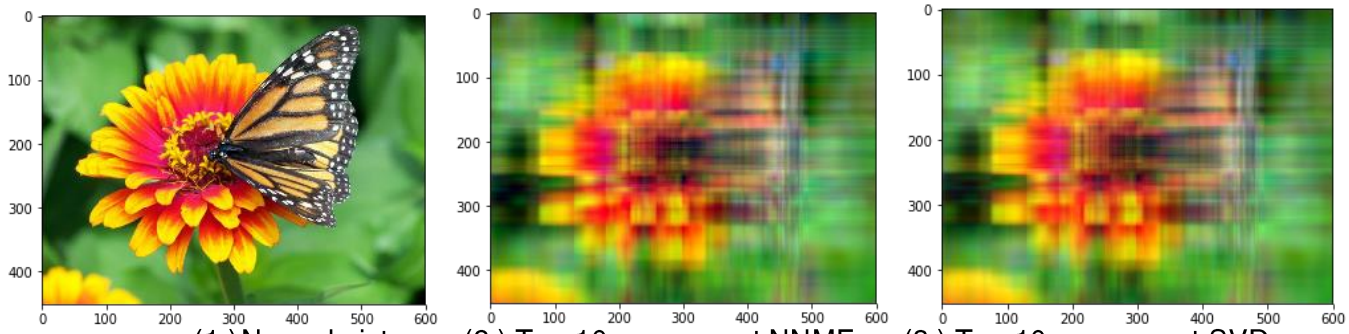
It is closely related to **Singular Value Decomposition** (SVD), whereby:

$$X = U\Sigma V^T$$

In NNMF, two matrices W, H are the factors, whereby:

$$X = WH$$

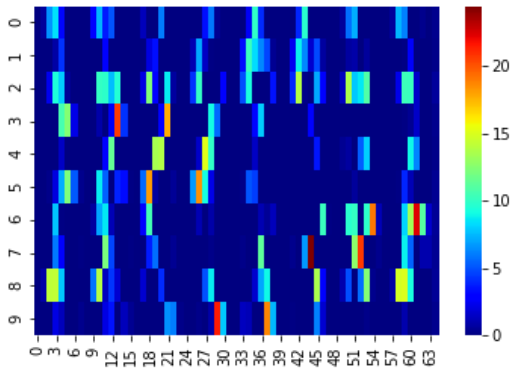
However, the constraint is that both W, H must have all positive elements. This constraint allows one to understand the basis vectors much more clearly.



(1.) Normal picture (2.) Top 10 component NNMF (3.) Top 10 component SVD

When used for image reconstruction, it is relatively unclear what's the point of NNMF. However, NNMF shines as factors positive matrices into a basis vector space and a columnar expression space.

The columns of W are called "features", and H showcases how much each of these features capture each original column of the data.



On the left, each row of H showcases how much of each column is expressed in the features. For example, take:

$$W = \begin{bmatrix} 1 & 2 \\ 0 & 1 \\ 1 & 1 \end{bmatrix}, H = \begin{bmatrix} 10 & 0 \\ 2 & 30 \end{bmatrix}$$

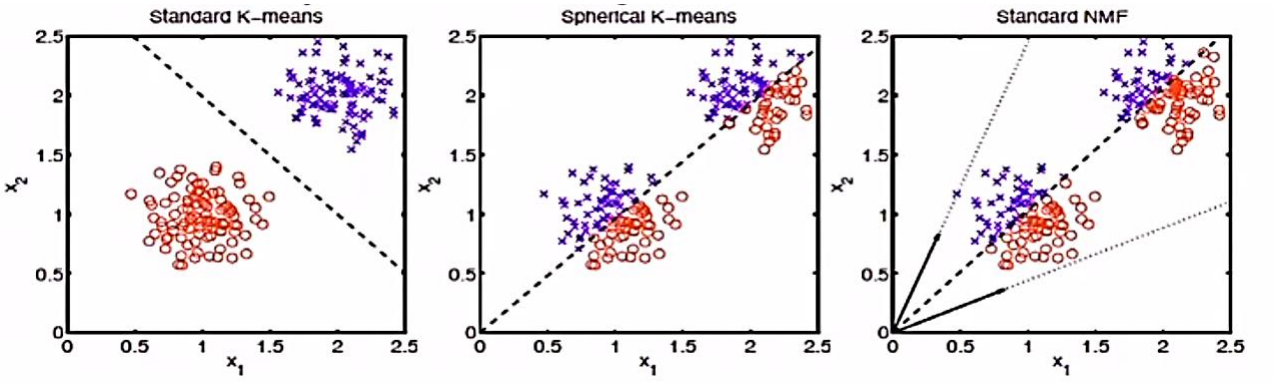
Then what this means is the feature column 1, or:

$$W_1 = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}$$

Captures the information mainly for column 1:

$$H_1 = [10 \quad 0]$$

So essentially, NNMF acts as a "**clustering**" method, whereby it tries to find combinations of the original column space to group similar observations together. This inherent clustering property is like **Spherical K-Means**. (Park, 2012)



Park notices that although NNMF's optimisation cost tries to minimise the **Frobenius Norm,** the resulting basis vectors seem to internally minimise **cosine distances**.

$$\min \|X - WH\|_F^2$$

This allows NNMF to work surprisingly well on text documents, since cosine distance is used continuously in **Natural Language Processing** as a method of comparing two sentence or paragraph vectors.

$$\boldsymbol{Cosine\ Similarity}\ (X, Y) = \frac{XY^T}{\|X\|\ \|Y\|}$$

$$\boldsymbol{Cosine\ Distance}\ (X, Y) = 1 - \boldsymbol{Cosine\ Similarity}\ (X, Y)$$

So in the end, we have that:

$$\hat{X} = W_1 H_1 + W_2 H_2 + \cdots + W_k H_k$$

This is true, since we showed in Linear Least Squares that:

$$X^T X = X_t^T X_t + X_{t+1}^T X_{t+1}$$

So, we choose a rank k, and we try to find both W, H where we want:

$$\min \|X - WH\|_F^2 \text{ subject to } W \geq 0, H \geq 0$$

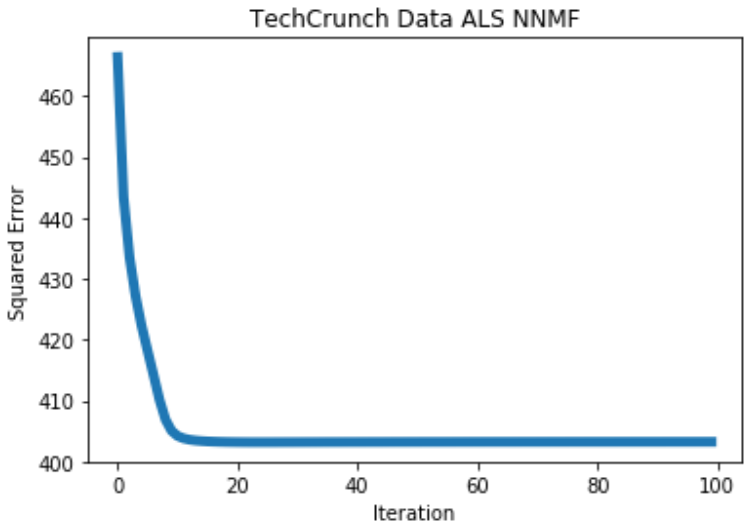
a. Alternating Least Squares & Fast HALS

A fundamental problem is that finding W, H is **not easy**. Since we have the 2 constraints where each element must be positive, it is relatively hard to find the best matrices which satisfy these conditions.

One way to find these solutions is via **Alternating Least Squares**. First proposed by Berry in 2006 (Berry, 2006), ALS is a popular technique used to find both W and H. It performs Least Squares on the problem, whereby:

1. Fix W, find H
2. Fix H, find W

The above steps are computed iteratively, and to consider the non-negativity constraints, all negative elements are set directly to 0.



When applying on **TechCrunch News Data**, (PromptCloudHQ, 2017), we can see that the error converges to the lowest error possible for ALS at around 10 iterations or so with 10 components.

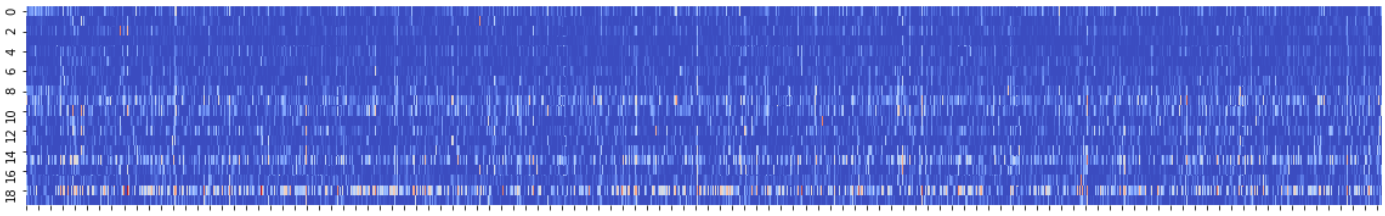
ALS seems to do OK, as it seems to minimise the errors. Let us confirm if ALS has "clustered" the news articles appropriately into groups.

If we analyse components 1 and 6:

$$H_1 = 20.6\ \text{Facebook} + 2.9\ \text{Ads} + 1.8\ \text{News} + 1.7\ \text{Feed} + 1.6\ \text{People} + 1.4\ \text{Users} \ldots$$

$$H_6 = 4.7\ \text{Iphone} + 4.1\ \text{IOS} + 3.4\ \text{Devices} + 3.0\ \text{Android} + 2.8\ \text{Device} + 2.3\ \text{Samsung} \ldots$$

Clearly, we can see ALS has shown that component 1 is about Facebook, and component 6 is about smartphones. Clearly, ALS seems to work for now.



Heatmap of H matrix from ALS. Each column is a word. Lighter = more prominent.

Alternating Least Squares NNMF

K = number of components (1)
n, p = X.shape
W = zeros (shape = (n, **K**))
H = zeros (shape = (**K**, p))

While not converged:
    H = max (**Epsilon Jitter Cholesky Solve (** W, X) , 0)
    W = max (**Epsilon Jitter Cholesky Solve** (H.T, X.T) , 0).T
Return W, H

Specify number of components <= p. Common is like 5, 10, 20, 30.

By analysing this algorithm, we see that the algorithm does this:

$$W\widehat{\boldsymbol{H}} = X$$

$$H^T\widehat{\boldsymbol{W^T}} = X^T$$

This allows the above equations to be solved via normal least squares where:

$$A\widehat{\boldsymbol{\theta}} = B$$

Essentially, the trick above is that we want W to exist on the RHS, so, we transpose:

$$X = WH$$

$$X^T = (WH)^T$$

$$X^T = H^T W^T$$

When we want to solve $W\widehat{\boldsymbol{H}} = X$,

$$\widehat{\boldsymbol{H}} = \max(\ (W^TW)^{-1}W^TX\ ,\quad 0)$$

Now, the above clearly takes nk^2 for W^TW, $\frac{1}{2}k^3$ for inverse Cholesky, and $(W^TW)^{-1}W^T$takes nk^2. Finally, $(W^TW)^{-1}W^TX$ takes npk.

Notice however, doing $(W^TW)^{-1}(W^TX)$ is **SLOWER**, thus utilising **Matrix Chain Multiplication** again, as we see this approach takes $npk + np^2$.

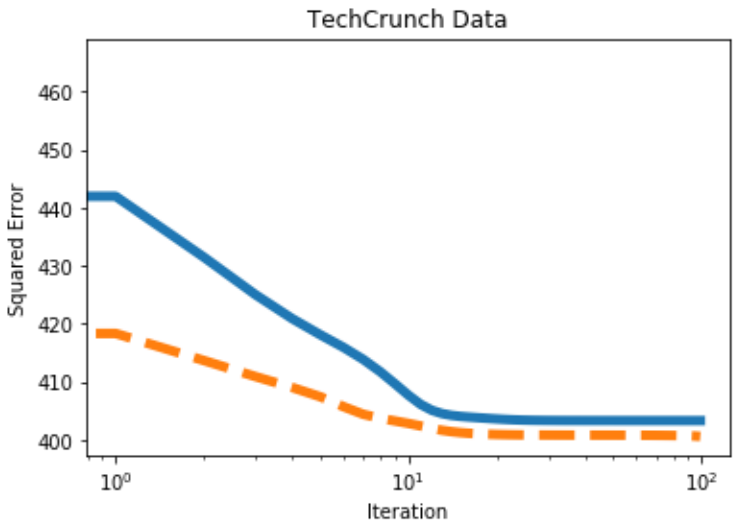
Notice the nk^2 vs np^2. Clearly, $k < p$, hence perform performs from left to right.

When we want to solve $H^T\widehat{\boldsymbol{W^T}} = X^T$:

$$\widehat{\boldsymbol{W^T}} = \max(\ (HH^T)^{-1}HX^T\ ,\quad 0)$$

$$\widehat{\boldsymbol{W}} = \widehat{\boldsymbol{W^T}}^T$$

The above takes pk^2 for HH^T, inverse takes $\frac{1}{2}k^3$, and $(HH^T)^{-1}HX^T$ takes $k^2p + npk$. Likewise, doing HX^T first is slower, causing extra nk^2 vs pk^2 steps.



However, in 2011, Hsieh and Dhillon introduced **Fast-HALS (Fast Hierarchical ALS)** (Hsieh, 2011) which phenomenally allowed NNMF to converge much faster.

On the left, the light orange dotted line showcases Fast HALS, and the dark blue is the old ALS algorithm. Clearly, Fast HALS converges much faster.

In **Sklearn**, Fast HALS is implemented. (Sklearn, 2018), and is extremely quick and powerful, as it uses **Coordinate Descent**.

Fast HALS

```
K = number of components
n, p = X.shape
W = zeros ( shape = (n, K) )
H = zeros ( shape = (K, p) )

Function CD_Update ( W, HHT, XHT, n, k ):
        XHT *= -1

        For i in range ( k ):
                For j in range ( n ):

                        gradient = XHT[ i , j ]
                        For m in range ( k ):
                                gradient += HHT[ i, m ] * W[ j , m ]

                        projected_gradient = min ( 0, gradient )   if W[ j ,i ] == 0  else  gradient
                        if gradient  != 0:
                                W[ j , i ] = max ( W[ j , i ] – gradient / HHT[ i, i ],    0   )

While not converged:
        CD_Update ( W,   H @ H.T  ,   X @ H.T  ,  n, k )
        CD_Update ( H.T,  W.T @ W  ,   X.T @ W  ,  p, k )
Return W, H
```

Notice above how there exists a lot of for loops! This **nested structure** is rather unfortunate, as it does not allow easy **parallelisation**. Above, you can see it goes through every column and then every row. However, remember that:

$$\hat{X} = W_1 H_1 + W_2 H_2 + \cdots + W_k H_k$$

This highlights that each column in W and each row in H are technically unconnected, and so with this in mind, I present **Parallel Fast HALS** in the next section.

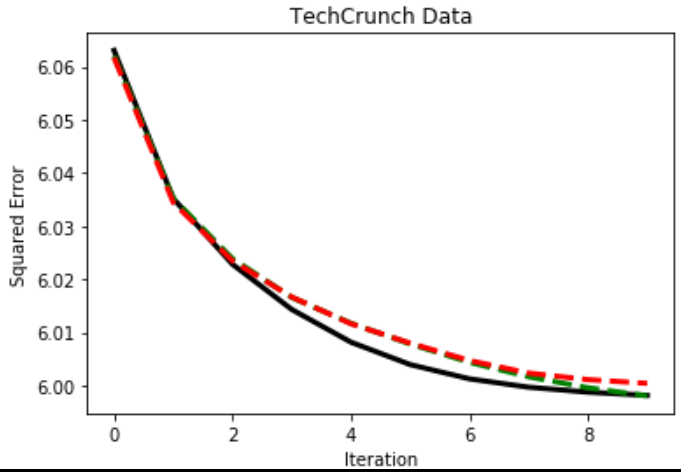
b. Parallel Fast HALS NNMF

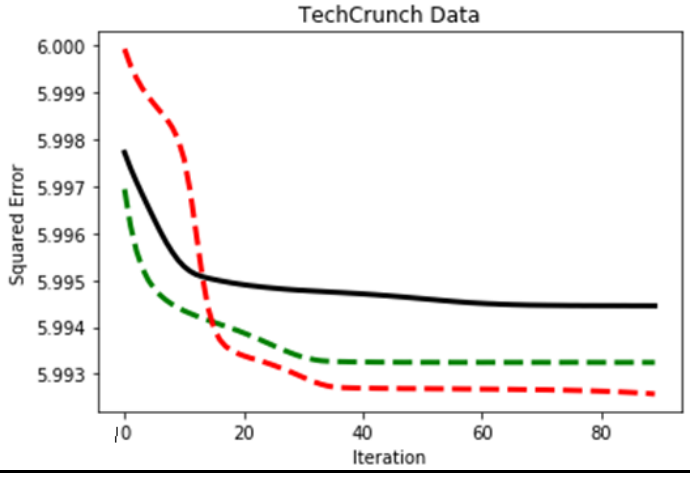
By noticing that W columns and H rows are essentially partially uncorrelated from one another, we can easily make the following changes in **blue**:

Parallel Fast HALS

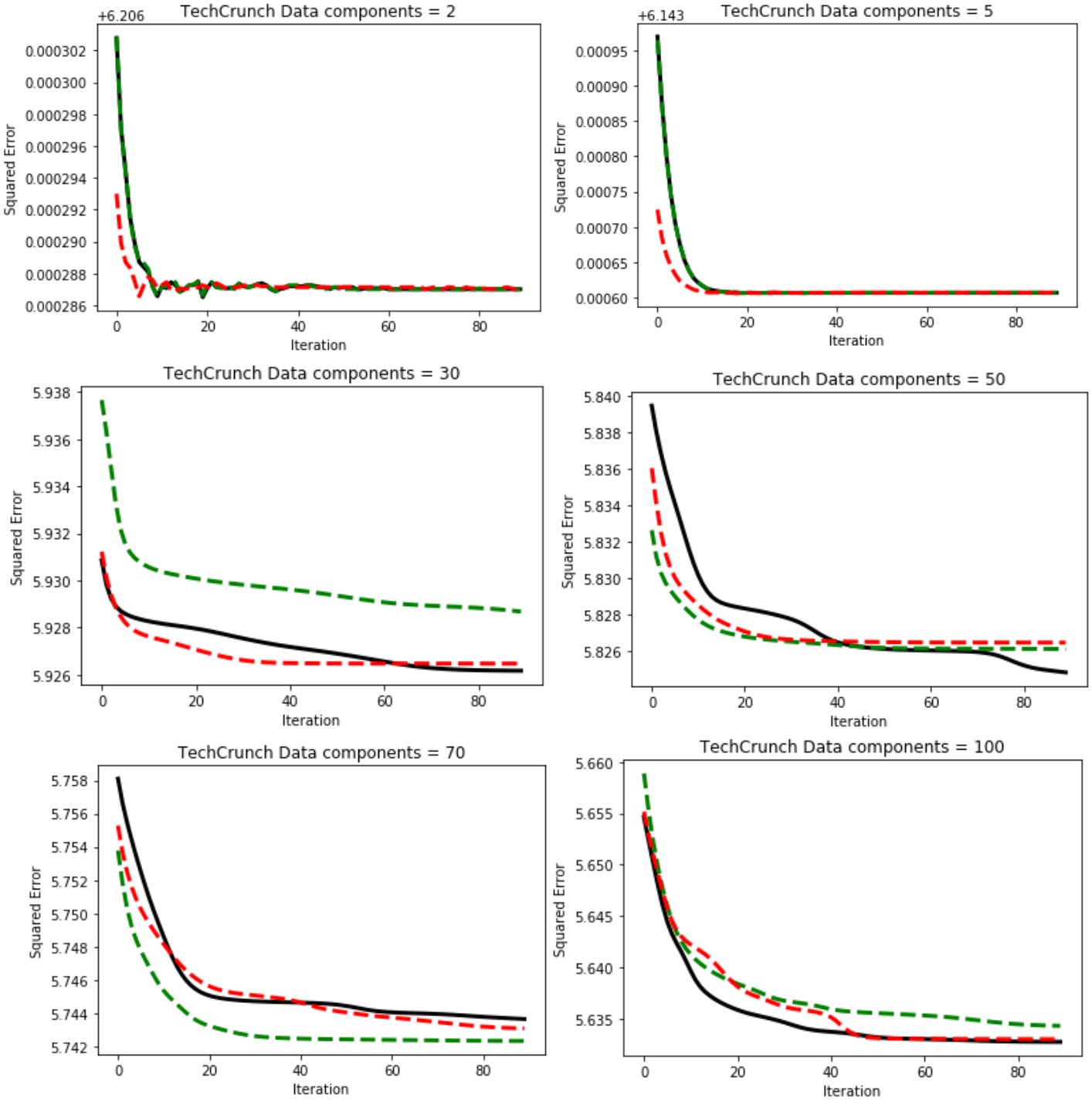
```
K = number of components
R = number of runs (default = 1)
n, p = X.shape
W = zeros ( shape = (n, K) )
H = zeros ( shape = (K, p) )

Function Fast_CD_Update ( W, HHT, XHT, n, k ):
        XHT *= -1
        Hessian = HHT[ i, i ]

        For run in range ( R ):
                For i in range ( k ) in parallel:
                        For j in range ( n ) in parallel:

                                gradient = XHT[ i , j ]
                                For m in range ( k ) in parallel:
                                        gradient += HHT[ i, m ] * W[ j , m ]

                                projected_gradient = min ( 0, gradient )   if W[ j ,i ] == 0  else  gradient
                                if Hessian != 0:
                                        if gradient  != 0:
                                                W[ j , i ] = max ( W[ j , i ] – gradient / Hessian ,  0  )

While not converged:
        Fast_CD_Update ( W,   H @ H.T  ,   X @ H.T  , n, k )
        Fast_CD_Update ( H.T,  W.T @ W  ,   X.T @ W  , p, k )
Return W, H
```

Essentially what I have done is to parallelise the gradient changes.





Fast HALS Original = **BLACK**
Parallel Fast HALS Runs = 1 = **GREEN**
Parallel Fast HALS Runs = 2 = **RED**
Astonishingly, when number of components = 20, in the first few iterations, Parallel Fast HALS did worse, as seen in the first 10 iterations. However, when the iterations grew, Parallel Fast HALS did even **BETTER** on the TechCrunch Data?? However, we must verify our claims. We shall try on multiple component numbers.



Clearly, on some components, Fast HALS has a lower error. On average, pure parallelisation with no runs is NOT good, and we can see that setting runs = 2 yields good results. Also, it might be good as iterations → ∞, switch back to Fast HALS.

References

Thanks so much for reading **Modern Big Data Algorithms**! I hope you like it! Many of the algorithms showcased are already implemented online at **github.com/danielhanchen/hyperlearn (HyperLearn).** Thanks a lot to all my supporters, and once again **Aleksandar Ignjatovic** for allowing me to complete this in COMP4121.

Thanks to these people:

Angell, D. (2010). *Mathematics Lecture Notes.* Retrieved from David Angell: http://web.maths.unsw.edu.au/~angell/

Berry, M. (2006). Algorithms and Applications for Approximate. *Elsevier*.

Blackford, S. (1999). *Singular Value Decomposition (SVD)*. Retrieved from LAPACK Netlib: https://www.netlib.org/lapack/lug/node32.html

C. Eckart, G. Y. (n.d.). The approximation of one matrix by another of lower rank. *Psychometrika*, 1936.

Cock, D. D. (2011). Ames, Iowa: Alternative to the Boston Housing Data as an. *Journal of Statistics Education*.

Cuppen, J. J. (1981). A divide and conquer method for the symmetric tridiagonal eigenproblem. *Numerical Math*.

Deerwester, S. e. (1988). Improving Information Retrieval with Latent Semantic Indexing. *Proceedings of the 51st Annual Meeting of the American Society for Information Science*.

Demmel, J. (2008). Performance and Accuracy of LAPACK's Symmetric Tridiagonal Eigensolvers. *SIAM J. Sci. Comput*.

Dhillon, C.-J. H. (2011). Fast Coordinate Descent Methods with Variable Selection. *Knowledge Discovery in Databases*, 11.

Francis, J. (1961). The QR Transformation, I. *The Computer Journal*.

Golub, G. H., & Kahan, W. (1965). *Calculating the singular values and pseudo-inverse of a matrix.* ournal of the Society for Industrial and Applied Mathematics.

Good, I. J. (1969). Some Applications of the Singular Decomposition of a Matrix. *Technometrics*.

Google. (n.d.). *Embedding Projector*. Retrieved from Google AI Experiments: http://projector.tensorflow.org/

Halko, e. a. (2009). Finding structure with randomness: Stochastic algorithms for constructing. *arXiv*.

Harrison, D. a. (1978). Hedonic prices and the demand for clean air. *J. Environ. Economics & Management*.

Hastie, T. (2014). Matrix Completion and Low-Rank SVD via Fast Alternating Least Squares. *arXiv*.

Holub. (n.d.). In G. H. C., *Matrix Computations* (pp. Chapter 5, section 5.4.4, pp 252-253).

Hsieh, D. (2011). Fast Coordinate Descent Methods with Variable Selection. *KDD*.

HST 1995, H. S. (n.d.). *Pillars of Creation.* NASA 1995.

Lichman, M. (2013). UCI Machine Learning Repository.

Mises, R. v. (1929). Praktische Verfahren der Gleichungsauflösung. *ZAMM - Zeitschrift für Angewandte Mathematik und Mechanik*.

Ng, A. (2009). CS229 Machine Learning. Stanford University.

Orlans, S. (n.d.). *City Night.* Retrieved from https://www.behance.net/gallery/51904795/City-Night

Park, H. (2012). Nonnegative Matrix Factorizations for Clustering. *Workshop on Algorithms for Modern Massive Data Sets.* Stanford University.

Parlet N., B. (n.d.). The symmetric eigenvalue problem. *SIAM, Classics in Applied Mathematics*, 1998.

PromptCloudHQ. (2017). *Article Titles from TechCrunch and VentureBeat.* Retrieved from Kaggle: https://www.kaggle.com/PromptCloudHQ/titles-by-techcrunch-and-venturebeat-in-2017

Raz, R. (n.d.). On the complexity of matrix product. *Proceedings of the thirty-fourth annual ACM symposium on Theory of computing*, 2002.

Ross, D. (2008). Incremental Learning for Robust Visual Tracking. *International Journal of Computer Vision*.

Scipy. (2018, May 5). *scipy.linalg.lapack.sgesdd*. Retrieved from Scipy.org Enthought: https://docs.scipy.org/doc/scipy/reference/generated/scipy.linalg.lapack.sgesdd.html#scipy.linalg.lapack.sgesdd

Sklearn. (2018). *sklearn.decomposition.NMF*. Retrieved from Sklearn: https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.NMF.html#sklearn.decomposition.NMF

Sorkine-Hornung, O. (2016). *SVD and Applications.* Retrieved from Swiss Federal Institute of Technology, Zurich: https://igl.ethz.ch/teaching/linear-algebra/la2016/material/SVD-applications.pdf

Strang, G. (2016). *Introduction to Linear Algebra. 5th ed.* Wellesley-Cambridge Press.

Tikhonov, A. (1977). Solution of Ill-posed Problems. *Washington: Winston & Sons*.

Trefethen, L. N. (1996). *Numerical linear algebra.*

Weisstein, E. W. (n.d.). *Singular Matrix*. Retrieved from WolframMathWorld: http://mathworld.wolfram.com/SingularMatrix.html

Modern Big Data Algorithms

Daniel Han-Chen

HyperLearn

Edition 1

2018 November