7_decomp

February 6, 2024

1 Matrix Decompositions

variationalform https://variationalform.github.io/

Just Enough: progress at pace https://variationalform.github.io/

https://github.com/variationalform

Simon Shaw https://www.brunel.ac.uk/people/simon-shaw.

This document uses python

and also makes use of LaTeX

in Markdown

1.1 What this is about:

You will be introduced to …

- Methods for decomposing matrices in to alternative forms.
- Eigenvalues and Eigenvectors of square matrices.
- The Singular Value Decomposition (SVD) for non-square matrices
- Using `numpy` to compute these decompositions.

As usual our emphasis will be on *doing* rather than *proving*: *just enough: progress at pace*

1.2 Assigned Reading

For this worksheet you should read Chapter 7 of [FCLA], and Chapters 4 of [MML].

- MML: Mathematics for Machine Learning, by Marc Peter Deisenroth, A. Aldo Faisal, and Cheng Soon Ong. Cambridge University Press. https://mml-book.github.io.
- FCLA: A First Course in Linear Algebra, by Ken Kuttler, https://math.libretexts.org/Bookshelves/Linear_Algebra/A_First_Course_in_Linear_Algebra_(Kuttler)

All of the above can be accessed legally and without cost.

There are also these useful references for coding:

- PT: `python`: https://docs.python.org/3/tutorial

- NP: `numpy`: https://numpy.org/doc/stable/user/quickstart.html
- MPL: `matplotlib`: https://matplotlib.org

1.3 Eigenvalues and Eigenvectors

Consider this matrix and vector,

$$\boldsymbol{B} = \begin{pmatrix} 3 & -2 & 4 \\ -6 & 6 & -11 \\ 6 & 2 & 5 \end{pmatrix} \qquad \text{and} \qquad \boldsymbol{x} = \begin{pmatrix} -2 \\ 3 \\ 1 \end{pmatrix} \qquad \text{then} \qquad \boldsymbol{Bx} = \begin{pmatrix} -8 \\ 19 \\ -1 \end{pmatrix}$$

and notice that $\boldsymbol{B}$ *mixes up* $\boldsymbol{x}$ to such an extent that $\boldsymbol{Bx}$ bears no relationship to the original $\boldsymbol{x}$. This isn't so surprising when you think about it.

We can do this in `python` using `numpy` as follows…

[ ]:
```
import numpy as np
B = np.array( [[3, -2, 4],[-6, 6, -11],[ 6, 2, 5 ]])
x = np.array([[-2], [3], [1]])
f = B.dot(x)
print('f = \n', f)
```

On the other hand, consider this matrix with a different vector,

$$\boldsymbol{B} = \begin{pmatrix} 3 & -2 & 4 \\ -6 & 6 & -11 \\ 6 & 2 & 5 \end{pmatrix} \qquad \text{and} \qquad \boldsymbol{w} = \begin{pmatrix} -2 \\ 5 \\ 2 \end{pmatrix}.$$

This time,

$$\boldsymbol{Bw} = \begin{pmatrix} 3 & -2 & 4 \\ -6 & 6 & -11 \\ 6 & 2 & 5 \end{pmatrix} \begin{pmatrix} -2 \\ 5 \\ 2 \end{pmatrix} = \begin{pmatrix} -8 \\ 20 \\ 8 \end{pmatrix} = 4 \begin{pmatrix} -2 \\ 5 \\ 2 \end{pmatrix} = 4\boldsymbol{w}$$

So $\boldsymbol{Bw} = 4\boldsymbol{w}$, and all $\boldsymbol{B}$ does is magnify $\boldsymbol{w}$ to be 4 times longer.

Here it is again,

$$\boldsymbol{Bw} = \begin{pmatrix} 3 & -2 & 4 \\ -6 & 6 & -11 \\ 6 & 2 & 5 \end{pmatrix} \begin{pmatrix} -2 \\ 5 \\ 2 \end{pmatrix} = \begin{pmatrix} -8 \\ 20 \\ 8 \end{pmatrix} = 4 \begin{pmatrix} -2 \\ 5 \\ 2 \end{pmatrix} = 4\boldsymbol{w}$$

1.3.1 Exercise:

use `python` to show that $\frac{1}{4}\boldsymbol{Bw} - \boldsymbol{w} = \boldsymbol{0}$

[ ]:
```
B = np.array( [[3, -2, 4],[-6, 6, -11],[ 6, 2, 5 ]])
w = np.array([[-2], [5], [2]])
f = 0.25*B.dot(w)
print('f - w = \n', f-w)
```

```
# or many other variants, such as
print('result = \n', 0.25*B.dot(w)-w)
```

$$\boldsymbol{Bw} = \begin{pmatrix} 3 & -2 & 4 \\ -6 & 6 & -11 \\ 6 & 2 & 5 \end{pmatrix} \begin{pmatrix} -2 \\ 5 \\ 2 \end{pmatrix} = \begin{pmatrix} -8 \\ 20 \\ 8 \end{pmatrix} = 4 \begin{pmatrix} -2 \\ 5 \\ 2 \end{pmatrix} = 4\boldsymbol{w}$$

This seems more surprising. Here 4 is called an *eigenvalue* ('own value') of $\boldsymbol{B}$ and $\boldsymbol{w}$ is the corresponding *eigenvector*.

In general, for any square matrix $\boldsymbol{B}$ the problem of finding scalars λ and vectors $\boldsymbol{v}$ such that

$$\boldsymbol{Bv} = \lambda\boldsymbol{v}$$

is called an *eigenvalue problem*. The following facts are known to be true:

> **The Eigenvalue Theorem.** Every square matrix of dimension n has n eigenvalue-eigenvector pairs, $(\lambda_1, \boldsymbol{v}_1), (\lambda_2, \boldsymbol{v}_2), \ldots, (\lambda_n, \boldsymbol{v}_n)$. The eigenvalues need not be distinct, and the eigenvector lengths are arbitrary. **A matrix which has one or more zero eigenvalues is not invertible.** On the other hand, **If the eigenvalues of a matrix are all non-zero then that matrix is invertible**, and it has **full rank**. The determinant of a matrix is the product of its eigenvalues.

NOTE: if $\boldsymbol{Bv} = \lambda\boldsymbol{v}$ then it can be shown that we need $\det(\boldsymbol{B} - \lambda\boldsymbol{I}) = 0$. This is not practically useful, but is of central importance for theory.

1.3.2 Example 1

For $\boldsymbol{B}$ above we have that

$$\boldsymbol{Bv} = \lambda\boldsymbol{v}$$

for $(\lambda_1, \boldsymbol{v}_1), (\lambda_2, \boldsymbol{v}_2), (\lambda_3, \boldsymbol{v}_3)$ given by

$$9 \text{ with } \begin{pmatrix} 17 \\ -45 \\ 3 \end{pmatrix}, \qquad 1 \text{ with } \begin{pmatrix} -1 \\ 1 \\ 1 \end{pmatrix} \qquad \text{and} \qquad 4 \text{ with } \begin{pmatrix} -2 \\ 5 \\ 2 \end{pmatrix}.$$

We have already seen the case $\lambda = 4$ and $\boldsymbol{v} = (-2, 5, 2)^T$ above.

The eigenvectors are not unique - they can be multiplied by an arbitrary (non-zero) scalar and they remain eigenvectors. For that reason it is usual to *normalize* an eigenvector by dividing through by its length, as given by the (Euclidean, Pythagorean) 2-norm, $\|\cdot\|_2$. For example, for $\boldsymbol{v} = (-2, 5, 2)^T$ we have

$$\|\boldsymbol{v}\|_2 = \sqrt{(-2)^2 + 5^2 + 2^2} = \sqrt{33}$$

which means that

$$\boldsymbol{v} = \frac{1}{\sqrt{33}} \begin{pmatrix} -2 \\ 5 \\ 2 \end{pmatrix} = \begin{pmatrix} -0.348155311911396 \\ 0.870388279778489 \\ 0.348155311911395 \end{pmatrix}$$

is also an eigenvector for the eigenvalue $\lambda = 4$.

> **THINK ABOUT:** An eigenpair satisfies $\boldsymbol{Bv} = \lambda\boldsymbol{v}$. Choose a non-zero real number α and write $\boldsymbol{w} = \alpha\boldsymbol{v}$. Is it true that $\boldsymbol{Bw} = \lambda\boldsymbol{w}$? Can you see why eigenvectors are not unique in length?
>
> **THINK ABOUT:** If we choose $\alpha = \|\boldsymbol{v}\|_2^{-1}$ above what can you say about the value of $\|\boldsymbol{w}\|_2$?

If we normalize each of the eigenvectors above with their own length we get something like this (the decimals may go on for ever - why?):

$$9 \text{ with } \begin{pmatrix} 0.3527\ldots \\ -0.9336\ldots \\ 0.0622\ldots \end{pmatrix}, \qquad 1 \text{ with } \begin{pmatrix} -0.5773\ldots \\ 0.5773\ldots \\ 0.5773\ldots \end{pmatrix} \qquad \text{and} \qquad 4 \text{ with } \begin{pmatrix} -0.3481\ldots \\ 0.8703\ldots \\ 0.3481\ldots \end{pmatrix}.$$

Let's use `numpy` to calculate the eigensystem for $\boldsymbol{B}$. It goes like this…

[ ]:
```
w, V = np.linalg.eig(B)
print(w)
print(V)
```

We can see that two quantities are returned, `w` and `V`. The eigenvalues are collected in `w` and the corresponding eigenvectors are the columns of `V`.

Note that these columns of `V` agree with our calculations above for the normalized eigenvectors.

1.4 The Eigen-System

As indicated by this computation, we can stack the length-normalized eigenvectors together next to each other, with the eigenvalues on the leading diagonal of an otherwise zero matrix. We also make sure that the eigenvectors appear in the same order as the corresponding eigenvalues and then use the fact that $\boldsymbol{Bv} = \lambda\boldsymbol{v}$ for each eigen-pair. Then entire eigen-system can then be represented in one equation. For example,

$$\begin{pmatrix} 3 & -2 & 4 \\ -6 & 6 & -11 \\ 6 & 2 & 5 \end{pmatrix} \begin{pmatrix} 0.3527\ldots & -0.5773\ldots & -0.3481\ldots \\ -0.9336\ldots & 0.5773\ldots & 0.8703\ldots \\ 0.0622\ldots & 0.5773\ldots & 0.3481\ldots \end{pmatrix}$$

$$= \begin{pmatrix} 0.3527\ldots & -0.5773\ldots & -0.3481\ldots \\ -0.9336\ldots & 0.5773\ldots & 0.8703\ldots \\ 0.0622\ldots & 0.5773\ldots & 0.3481\ldots \end{pmatrix} \begin{pmatrix} 9 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 4 \end{pmatrix}$$

We can write this as

$$\boldsymbol{BV} = \boldsymbol{VD}$$

where the columns of $\boldsymbol{V}$ are the eigenvectors of $\boldsymbol{B}$, and the diagonal matrix $\boldsymbol{D}$ has the eigenvalues on the leading diagonal. The left-to-right order of the eigenvalues in $\boldsymbol{D}$ matches the order that the eigenvectors appear in $\boldsymbol{V}$.

All three of these matrices are square and they each have the same dimension as $\boldsymbol{B}$.

Now, suppose that $\det(\boldsymbol{V}) \neq 0$, then $\boldsymbol{V}^{-1}$ exists and we can (pre-)multiply both sides of $\boldsymbol{BV} = \boldsymbol{VD}$ by $\boldsymbol{V}^{-1}$ and get,

$$\boldsymbol{V}^{-1}\boldsymbol{BV} = \boldsymbol{V}^{-1}\boldsymbol{VD} = \boldsymbol{D}.$$

We see that this has produced a diagonal matrix $\boldsymbol{V}^{-1}\boldsymbol{BV}$ that is similar to $\boldsymbol{B}$ in the sense that it has the same eigenvalues (why are they the same? What can you say about the eigenvalues of diagonal matrices?). Such an operation is called a *similarity transformation*.

On the other hand, we can (post-)multiply both sides of $\boldsymbol{BV} = \boldsymbol{VD}$ by $\boldsymbol{V}^{-1}$ and get,

$$\boldsymbol{B} = \boldsymbol{BVV}^{-1} = \boldsymbol{VDV}^{-1}.$$

One reason why this is useful is that we can now easily raise $\boldsymbol{B}$ to powers. For example,

$$\boldsymbol{B}^2 = (\boldsymbol{VDV}^{-1})(\boldsymbol{VDV}^{-1}) = \boldsymbol{VD}^2\boldsymbol{V}^{-1}.$$

and so on.

However, to do this we needed to assume that $\det(\boldsymbol{V}) \neq 0$, and this need not be the case. Matrices for which this is true are called *diagonalizable*, and matrices for which it isn't true are called *defective*.

1.5 Eigen-systems of Symmetric Matrices

The eigenvalues and eigenvectors of a general square matrix could be **complex numbers** (which we aren't going to be too concerned with), and for large matrices the inverse $\boldsymbol{V}^{-1}$ could be hard to find explicitly.

However, in the special case of a symmetric matrix $\boldsymbol{A}$, the eigensystem $\boldsymbol{Av} = \lambda\boldsymbol{A}$ is made up exclusively of *real numbers*.

Furthermore, the matrix of normalized eigenvectors, $\boldsymbol{V}$, is an orthogonal matrix. This means that $\boldsymbol{V}^{-1} = \boldsymbol{V}^T$ - which is very easy to calculate once $\boldsymbol{V}$ is known.

This is called the *Spectral Theorem* - see [MML, Theorem 4.15]

> **Spectral Theorem (for matrices)** If $\boldsymbol{A}$ is real and symmetric (hence square) then its eigenvalues are all real and its eigenvector matrix $\boldsymbol{V}$ can be taken as *orthogonal* so that $\boldsymbol{V}^{-1} = \boldsymbol{V}^T$.

Let's see an example of this.

$$\text{if } \boldsymbol{A} = \begin{pmatrix} 3 & -2 & 4 \\ -2 & 6 & 2 \\ 4 & 2 & 5 \end{pmatrix}$$

then (with some rounding),

$$\boldsymbol{D} \approx \begin{pmatrix} -1.217 & 0 & 0 \\ 0 & 8.217 & 0 \\ 0 & 0 & 7 \end{pmatrix} \quad \text{and} \quad \boldsymbol{V} \approx \begin{pmatrix} .726 & .522 & -.447 \\ .363 & .261 & .894 \\ -.584 & .812 & 0 \end{pmatrix}$$

We'll take a look at how to do that in the lab.

We will verify that $\boldsymbol{V}^T\boldsymbol{V} = \boldsymbol{I}$ up to rounding error.

We'll also confirm that $\boldsymbol{AV} = \boldsymbol{VD}$.

1.6 The Eigen-Decomposition

Next, by post-multiplying by $\boldsymbol{V}^{-1}$ we note that we can also write $\boldsymbol{AV} = \boldsymbol{VD}$ as $\boldsymbol{A} = \boldsymbol{VDV}^T$ which, in expanded form, is,

$$\boldsymbol{A} = (\boldsymbol{v}_1\ \boldsymbol{v}_2\ \boldsymbol{v}_3\ \ldots) \begin{pmatrix} \lambda_1 & & & \\ & \lambda_2 & & \\ & & \lambda_3 & \\ & & & \ddots \end{pmatrix} \begin{pmatrix} \boldsymbol{v}_1^T \\ \boldsymbol{v}_2^T \\ \boldsymbol{v}_3^T \\ \vdots \end{pmatrix}$$

(we haven't shown all the zero elements). Then simplifying this we get

$$\boldsymbol{A} = (\boldsymbol{v}_1\ \boldsymbol{v}_2\ \boldsymbol{v}_3\ \ldots) \begin{pmatrix} \lambda_1\boldsymbol{v}_1^T \\ \lambda_2\boldsymbol{v}_2^T \\ \lambda_3\boldsymbol{v}_3^T \\ \vdots \end{pmatrix} = \lambda_1\boldsymbol{v}_1\boldsymbol{v}_1^T + \lambda_2\boldsymbol{v}_2\boldsymbol{v}_2^T + \lambda_3\boldsymbol{v}_3\boldsymbol{v}_3^T + \cdots = \sum_{k=1}^{n} \lambda_k\boldsymbol{v}_k\boldsymbol{v}_k^T.$$

Let's see this in python for a specific example. We start by getting the eigen-system for $\boldsymbol{A}$ as above.

[ ]:
```
A = np.array([[3,-2,4],[-2,6,2],[4,2,5]])
w, V = np.linalg.eig(A)
D=np.diag(w)
print('D = \n', D)
```

Let's look at each decomposed term in turn. First, for the $k = 1$ term,

[ ]:
```
print( D[0,0]*V[:,0:1]*V[:,0:1].T )
```

Let's think about what is going on here. We are printing out the quantity

```
D[0,0]*V[:,0:1]*V[:,0:1].T
```

In this, the `D[0,0]` factor is just the eigenvalue from the top left (first row, first column) of the $\boldsymbol{D}$ matrix.

Then, next, `V[:,0:1]` is a **`numpy` slice**.

In this type of expression `[c,a:b]` means take the elements in row `c` that occupy columns `a` through to `b-1`. The expression `[:,a:b]` means take **all of** the rows. Remember that column and row numbering starts at zero in `numpy`.

So, `V[:,0:1]` says take the first column of `V` - a column vector, and `V[:,0:1].T` says take the transpose of the first solumn of `V`.

It is important to note that we have to write our slicing expressions in the form `V[:,0:1]` rather than `V[:,0]`, otherwise we lose the shape. See e.g. https://stackoverflow.com/questions/29635501/row-vs-column-vector-return-during-numpy-array-slicing

Here is a demo:

[ ]:
```
print('V = \n', V)
print('V[:,0:1] = \n', V[:,0:1])
print('V[:,0] = \n', V[:,0])
```

Therefore, for $k = 1$ we can write $\lambda_k \boldsymbol{v}_k \boldsymbol{v}_k^T$ in code as `D[0,0]*V[:,0:1]*V[:,0:1].T`.

The full reconstruction of $\boldsymbol{A}$ is then…

[ ]:
```
print('\n The reconstruction of A ...')
print(D[0,0]*V[:,0:1]*V[:,0:1].T + D[1,1]*V[:,1:2]*V[:,1:2].T + D[2,2]*V[:,2:
↪3]*V[:,2:3].T)
```

1.7 Implication: approximation of matrices

This is quite a big deal, it means in effect that we can break a square symmetric matrix into pieces and consider as many or as few of those pieces as we wish.

This point of view really suggests an approximation scheme. If $\boldsymbol{A}$ is $n \times n$ and symmetric then,

$$\boldsymbol{A} = \sum_{k=1}^{n} \lambda_k \boldsymbol{v}_k \boldsymbol{v}_k^T \qquad \text{which suggests that} \qquad \boldsymbol{A} \approx \sum_{k=1}^{m} \lambda_k \boldsymbol{v}_k \boldsymbol{v}_k^T$$

for $m < n$. We would want to sort the eigenvalues so that the most dominant ones come first in this sum, which is the same as saying

$$|\lambda_1| \geq |\lambda_2| \geq |\lambda_3| \geq \cdots$$

We'll look more closely at this in the workshop session where we will see the effect on the amount of data storage we can save.

Building on these examples we can infer that if a large matrix can be well approximated by just a few terms in the eigen-expansion then the amount of storage required in computer memory can be vastly reduced.

This is very useful. But it only applies to symmetric square matrices.

We can extend it to non-symmetric matrices by introducing complex numbers but we can't extend this to non-square matrices because they don't have eigenvalues.

Fortunately, there is another - even more powerful - tool at our disposal.

1.8 SVD: The Singular Value Decomposition

Only square matrices have eigenvalues, and not all square matrices are diagonalizable via the similarity transform. For general matrices (containing real numbers for us) there is a tool called the **SVD** - the *Singular Value Decomposition*.

Let $\boldsymbol{K}$ be an n-row by m-column matrix of real numbers.

Then $\boldsymbol{K} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^T$ - this is called the *Singular Value Decomposition* of $\boldsymbol{K}$. In this:

- $\boldsymbol{U}$ is an $n \times n$ *orthogonal square* matrix
- $\boldsymbol{\Sigma}$ is an $n \times m$ *rectangular diagonal* matrix
- $\boldsymbol{V}^T$ is an $m \times m$ *orthogonal square* matrix

The entries on the diagonal of $\boldsymbol{\Sigma}$ are called the *singular values* of $\boldsymbol{K}$ and the number of non-zero singular values gives the rank of $\boldsymbol{K}$.

The columns of $\boldsymbol{U}$ (resp. $\boldsymbol{V}$) are called the left (resp. right) singular vectors of $\boldsymbol{K}$.

Let's see an example of $\boldsymbol{K} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^T$.

$$\text{If } \boldsymbol{K} = \begin{pmatrix} 1 & 2 & 5 \\ 5 & -6 & 1 \end{pmatrix} \text{ then } \boldsymbol{U} = \begin{pmatrix} -0.06213\ldots & 0.99806\ldots \\ 0.99806\ldots & 0.06213\ldots \end{pmatrix},$$

$$\boldsymbol{\Sigma} = \begin{pmatrix} 7.88191\ldots & 0 & 0 \\ 0 & 5.46584\ldots & 0 \end{pmatrix} \text{ and } \boldsymbol{V} = \begin{pmatrix} 0.62525\ldots & 0.23944\ldots & -0.74278\ldots \\ -0.77553\ldots & 0.29699\ldots & -0.55708\ldots \\ 0.08720\ldots & 0.92437\ldots & 0.37139\ldots \end{pmatrix}$$

If we use these we can indeed check that

$$\begin{pmatrix} -0.062\ldots & 0.998\ldots \\ 0.998\ldots & 0.062\ldots \end{pmatrix} \begin{pmatrix} 7.881\ldots & 0 & 0 \\ 0 & 5.465\ldots & 0 \end{pmatrix} \begin{pmatrix} 0.625\ldots & 0.239\ldots & -0.742\ldots \\ -0.775\ldots & 0.296\ldots & -0.557\ldots \\ 0.087\ldots & 0.924\ldots & 0.371\ldots \end{pmatrix}^T = \begin{pmatrix} 1 & 2 & 5 \\ 5 & -6 & 1 \end{pmatrix}$$

as required. We aren't going to do it by hand though, that's what computers are for!

[ ]:
```
K = np.array([[1,2,5],[5,-6,1]])
U, S, VT = np.linalg.svd(K)
print(U)
print(S)
print(VT)
```

Note two things:

- `np.linalg.svd` returns $\boldsymbol{V}^T$, not $\boldsymbol{V}$.
- The shape of `S` doesn't agree with $\boldsymbol{\Sigma}$.

We'll need to pad `S` - and then we can check the reconstruction $\boldsymbol{K} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^T$.

The padding is a bit awkward - here it is...

[ ]:
```
S = np.hstack(( np.diag(S), np.zeros((2,1)) ))
print(S)
```

Now we can check the reconstruction $\boldsymbol{K} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^T$.

Note that we can also use `@` to perform matrix multiplication.

[ ]:
```
print(K - U @ S @ VT)
```

This is zero (to machine precision) as expected.

There is a great deal that can be said about the SVD, but we're going to stay narrowly focussed and explore its value in data science and machine learning.

Let's start with the following observation. If we denote the n-th column of $\boldsymbol{U}$ by $\boldsymbol{u}_n$, and the n-th column of $\boldsymbol{V}$ by $\boldsymbol{v}_n$, then the statement $\boldsymbol{K} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^T$ becomes (we saw something very similar to this above with eigenvalues),

$$\boldsymbol{K} = (\boldsymbol{u}_1\ \boldsymbol{u}_2\ \cdots\ \boldsymbol{u}_n)\begin{pmatrix}\sigma_1 & & & \\ & \sigma_2 & & \\ & & \ddots & \\ & & & \sigma_n\end{pmatrix}\begin{pmatrix}\boldsymbol{v}_1^T \\ \boldsymbol{v}_2^T \\ \vdots \\ \boldsymbol{v}_n^T\end{pmatrix} = (\boldsymbol{u}_1\ \boldsymbol{u}_2\ \cdots\ \boldsymbol{u}_n)\begin{pmatrix}\sigma_1\boldsymbol{v}_1^T \\ \sigma_2\boldsymbol{v}_2^T \\ \vdots \\ \sigma_n\boldsymbol{v}_n^T\end{pmatrix}$$

(we haven't shown all the zero elements of $\boldsymbol{\Sigma}$). Simplifying this further then gives,

$$\boldsymbol{K} = \sigma_1\boldsymbol{u}_1\boldsymbol{v}_1^T + \sigma_2\boldsymbol{u}_2\boldsymbol{v}_2^T + \cdots + \sigma_n\boldsymbol{u}_n\boldsymbol{v}_n^T$$

THINK ABOUT: $\boldsymbol{u}\boldsymbol{v}^T$ is a rank 1 matrix - why?

2 The SVD Theorem

Here are the full details. For $\boldsymbol{B} \in \mathbb{R}^{m\times n}$

$$\boldsymbol{B} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^T = \sum_{j=1}^{p}\sigma_j\boldsymbol{u}_j\boldsymbol{v}_j^T$$

where: $\boldsymbol{U} \in \mathbb{R}^{m\times m}$, $\boldsymbol{\Sigma} \in \mathbb{R}^{m\times n}$, $\boldsymbol{V} \in \mathbb{R}^{n\times n}$ and $p = \min\{m, n\}$.

Note that $\boldsymbol{\Sigma} = \text{diag}(\sigma_1, \ldots, \sigma_p)$ and we can always arrange this so that $0 \leq \sigma_1 \leq \cdots \leq \sigma_p$.

$\boldsymbol{B}$ is real here (we aren't interested in complex matrices), then $\boldsymbol{U}$ and $\boldsymbol{V}$ are real and *orthogonal*.

If $\sigma_r \neq 0$ and $\sigma_p = 0$ for all $p > r$ then r is the rank of $\boldsymbol{B}$.

Note storage for $\boldsymbol{B}$ is mn. That for the SVD is $r(m+n+1)$. The ratio is

$$\frac{r(m+n+1)}{mn} = \frac{r}{n} + \frac{r}{m} + \frac{r}{mn}$$

2.1 The Thin SVD

We saw above that the zero columns were missing from $\boldsymbol{\Sigma}$ when we used `numpy` to calculate the SVD. This is called the *Thin SVD*.

Here we still have $\boldsymbol{B} \in \mathbb{R}^{m \times n}$ but with

$$\boldsymbol{B} = \boldsymbol{U}_1 \boldsymbol{\Sigma}_1 \boldsymbol{V}^T = \sum_{j=1}^{n} \sigma_j \boldsymbol{u}_j \boldsymbol{v}_j^T$$

where: $\boldsymbol{U}_1 \in \mathbb{R}^{m \times n}$, $\boldsymbol{\Sigma}_1 \in \mathbb{R}^{n \times n}$.

It is called the *thin SVD* because we drop the values that make no contribution (i.e. the zeros).

Don't worry too much about this. We'll let `numpy` do the hard work for us.

3 HOMEWORK - very important

In the lab we are going to see how the SVD can be used to compress data.

We'll use **image compression** as an example.

Take a good quality jpeg colour photo (e.g. on your phone) of something vivid, detailed and colourful and save it on your account (One Drive, for example) so that your Jupyter notebook in Anaconda can use it.

We are going to use the SVD to compress the image.

3.1 Review

- we understand that matrices can be decomposed into multiplicative factors.
- we saw how the spectral theorem allows us to approximate square symmteric matrices using the eigen-system.
- we have seen how rectangular matrices can be similarly expanded in terms of the singular value decomposition.
- we saw how we can access this functionality using `numpy` in `python`.

3.2 Technical Notes, Production and Archiving

Ignore the material below. What follows is not relevant to the material being taught.

Production Workflow

- Finalise the notebook material above
- Clear and fresh run of entire notebook
- Create html slide show:
  - `jupyter nbconvert --to slides 7_decomp.ipynb`
- Set `OUTPUTTING=1` below
- Comment out the display of web-sourced diagrams
- Clear and fresh run of entire notebook
- Comment back in the display of web-sourced diagrams
- Clear all cell output
- Set `OUTPUTTING=0` below
- Save
- git add, commit and push to FML
- copy PDF, HTML etc to web site
  - git add, commit and push
- rebuild binder

Some of this originated from

https://stackoverflow.com/questions/38540326/save-html-of-a-jupyter-notebook-from-within-the-n

These lines create a back up of the notebook. They can be ignored.

At some point this is better as a bash script outside of the notebook

[ ]:
```
%%bash
NBROOTNAME='7_decomp'
OUTPUTTING=1

if [ $OUTPUTTING -eq 1 ]; then
  jupyter nbconvert --to html $NBROOTNAME.ipynb
  cp $NBROOTNAME.html ../backups/$(date +"%m_%d_%Y-%H%M%S")_$NBROOTNAME.html
  mv -f $NBROOTNAME.html ./formats/html/

  jupyter nbconvert --to pdf $NBROOTNAME.ipynb
  cp $NBROOTNAME.pdf ../backups/$(date +"%m_%d_%Y-%H%M%S")_$NBROOTNAME.pdf
  mv -f $NBROOTNAME.pdf ./formats/pdf/

  jupyter nbconvert --to script $NBROOTNAME.ipynb
  cp $NBROOTNAME.py ../backups/$(date +"%m_%d_%Y-%H%M%S")_$NBROOTNAME.py
  mv -f $NBROOTNAME.py ./formats/py/
else
  echo 'Not Generating html, pdf and py output versions'
fi
```