## Clustering and dimensionality reduction: k-means clustering, hierarchical clustering, principal component analysis (PCA), singular value decomposition (SVD) are all examples of Unsupervised Learning.

**Think of all of the ways that we have looked at for finding the hidden structure of datasets.** How do you summarize that and group the data so that it fits most usefully? How do you effectively represent data in a compressed format? These are the goals of unsupervised learning, which is called “unsupervised” because you start with **unlabeled data** (there’s no Y in Unsupervised Learning Use Cases).

The two unsupervised learning tasks we will investigate are **clustering** which is putting the data into groups by similarity. We will also look at r**educing dimensionality** to compress the data while maintaining its structure and usefulness.

There are some Examples of where unsupervised learning methods might be useful:

- Let’s say you want to build An advertising platform that segments the U.S. population into smaller groups with similar demographics and purchasing habits so that advertisers can reach their target market with relevant ads.

- Airbnb presently groups its housing listings into neighborhoods so that users can navigate listings more easily.

- A data science team reduces the number of dimensions in a large data set to simplify modeling and reduce file size.

When compared to supervised learning, it’s not always easy to come up with metrics for how well an unsupervised learning algorithm is doing. “Performance” is often subjective and domain-specific.

Despite the success of supervised machine learning and deep learning, there is a widespread belief that unsupervised learning has even greater potential. The learning of a supervised learning system is limited by its training. A supervised learning system can learn only those tasks that it’s trained for. An unsupervised system could theoretically achieve “artificial general intelligence,” that was discussed earlier which means that it has the ability to learn any task a human can learn. However, the technology isn’t there yet.

The biggest problem with [supervised learning](https://www.infoworld.com/article/3403403/supervised-learning-explained.html) is the expense of labeling the training data. I mean expense in terms of both time and cost. On the other hand, the biggest problem with unsupervised learning (where the data is not labeled) is that it often times just doesn’t work very well. Nevertheless, unsupervised learning does have its uses: It can sometimes be good for reducing the dimensionality of a data set, exploring the pattern and structure of the data, finding groups of similar objects, and detecting outliers and other noise in the data.

So with that said, it is worth trying unsupervised learning methods as part of your exploratory data analysis to discover patterns and clusters, to reduce the dimensionality of your data, to maybe discover latent features, or to remove outliers in your data. Whether you then need to move on to supervised learning or to using pre-trained models to do predictions depends on your goals and your data.

## **So What is the best way to explain unsupervised learning?**

Let’s Think about how human children learn. I am a parent and a teacher and I obviously don’t need to show my kids… especially my young children every breed of dog and cat there is to teach them to recognize dogs and cats. They can learn from a few examples, without a lot of explanation, and generalize on their own. Again, my job as a parent is to provide my kids with examples and a little context and let them generalize the concept so they own it. They might mistakenly call a Serval a Dog or a chihauha a cat the first time or two they see one, but they can be corrected pretty quickly.

Children intuitively lump groups of things they see into classes. This is an a priori concept that all children posess. The brain is pre-wired with this feature. One of the major goals of unsupervised learning is to essentially allow computers to develop this same ability. There is a great blog by Alex Graves and Kelly Clancy of DeepMind called “[Unsupervised learning: the curious pupil](https://deepmind.com/blog/unsupervised-learning/),” In it they say:

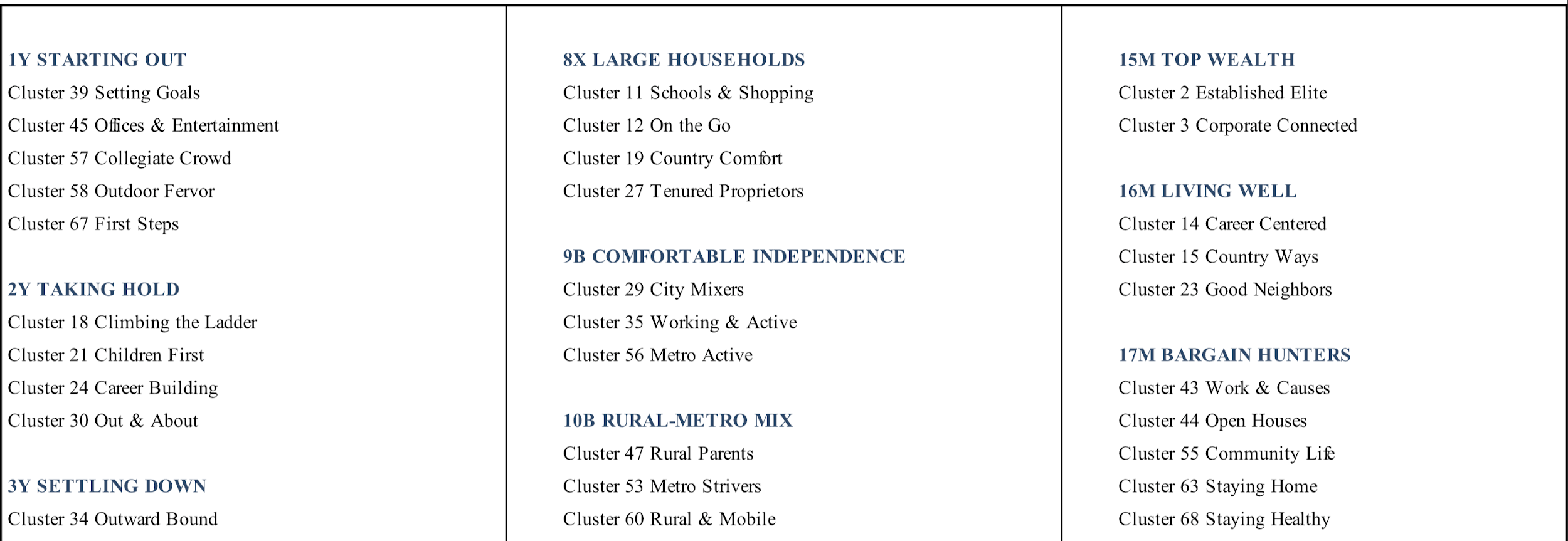
Unsupervised learning is a paradigm designed to create autonomous intelligence by rewarding agents (that is, computer programs) for learning about the data they observe without a particular task in mind. In other words, the agent learns for the sake of learning.

The potential of an agent that learns for the sake of learning is far greater than a system that reduces complex pictures to a binary decision (for examle, a dog or cat). Discovering patterns in data as opposed to carrying out a pre-defined task can often times produce surprising and useful results. I’ll just say this, researchers at Lawrence Berkeley Lab [ran a text processing algorithm (Word2vec) on several million material science abstracts](https://techxplore.com/news/2019-07-machine-learning-algorithms-uncover-hidden-scientific.html) and they were then able to predict discoveries of new thermoelectric materials.

---------------------------Clustering is next

# **Clustering**

An interesting example of clustering in the real world is marketing data provider Acxiom’s life stage clustering system, Personicx. This service segments U.S. households into 70 distinct clusters within 21 life stage groups that are used by advertisers when targeting Facebook ads, display ads, direct mail campaigns, etc.



A selection of Personicx demographic clusters

Their [white paper](http://c.ymcdn.com/sites/dema.site-ym.com/resource/resmgr/Member_Resources/Lifestage_Clustering.pdf) reveals that they used **centroid clustering** and **principal component analysis**, both of which are techniques covered in this series.

You can imagine how having access to these clusters is extremely useful for advertisers who want to (1) understand their existing customer base and (2) use their ad spend effectively by targeting potential new customers with relevant demographics, interests, and lifestyles.

A clustering problem is an unsupervised learning problem that asks the model to find groups of similar data points. There are a number of clustering algorithms currently in use, which tend to have slightly different characteristics. In general, clustering algorithms look at the metrics or distance functions between the feature vectors of the data points, and then group the ones that are “near” each other. Clustering algorithms work best if the classes do not overlap.



From the graphic on your screen, You can actually find out which cluster you personally would belong to by answering a few simple questions in Acxiom’s [“What’s My Cluster?” tool](https://isapps.acxiom.com/personicx/personicx.aspx).

In the next episodes, we’ll walk through a couple of clustering methods to develop your skills for how this task can be performed.

-------KMeans is next

# **k-means clustering**

*“And* k *rings were given to the race of Centroids, who above all else, desire power.”*

The goal of clustering is to create groups of data points so that points in different clusters are dissimilar while points within a cluster are similar.

With **k-means clustering**, we want to cluster our data points into *k* groups. A larger *k* creates smaller groups with more granularity, a lower *k* means larger groups and less granularity.

The output of the algorithm would be a set of “labels” assigning each data point to one of the *k* groups. In k-means clustering, the way these groups are defined is by creating a **centroid** for each group. The centroids are like the heart of the cluster, they “capture” the points closest to them and add them to the cluster.

Think of these as the people who show up at a party and soon become the centers of attention because they’re so magnetic. If there’s just one of them, everyone will gather around; if there are lots, many smaller centers of activity will form.

Here are the steps to k-means clustering:

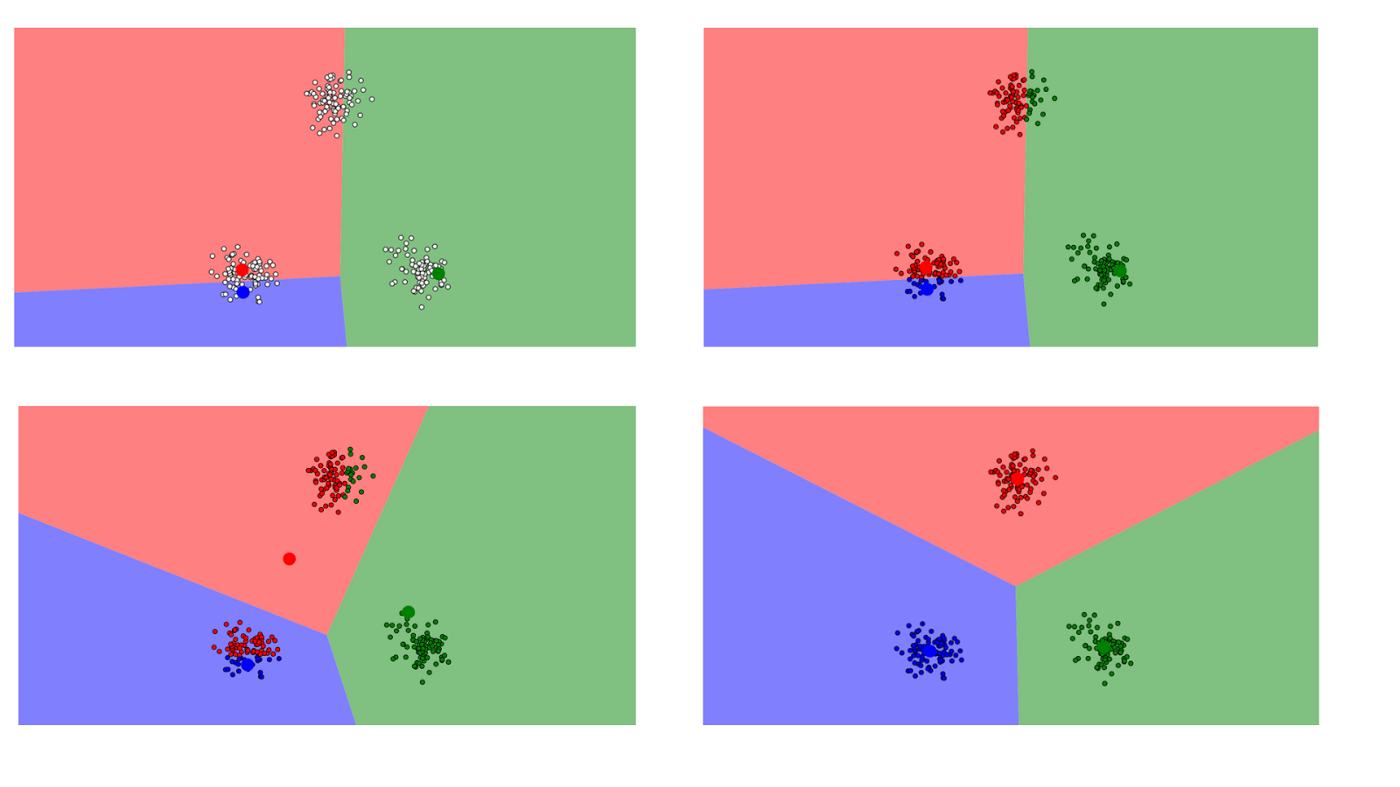
**1. Define the *k* centroids**. Initialize these at random (there are also fancier algorithms for initializing the centroids that end up converging more effectively).

**2. Find the closest centroid & update cluster assignments.** Assign each data point to one of the *k* clusters. Each data point is assigned to the nearest centroid’s cluster. Here, the measure of “nearness” is a hyperparameter — often Euclidean distance.

**3. Move the centroids to the center of their clusters.** The new position of each centroid is calculated as the average position of all the points in its cluster.

Keep repeating steps 2 and 3 until the centroid stop moving a lot at each iteration (i.e., until the algorithm converges).

That, in short, is how k-means clustering works! Check out this [visualization](https://www.naftaliharris.com/blog/visualizing-k-means-clustering/) of the algorithm — read it like a comic book. Each point in the plane is colored according the centroid that it is closest to at each moment. You’ll notice that the centroids (the larger blue, red, and green circles) start randomly and then quickly adjust to capture their respective clusters.



Another real-life application of k-means clustering is classifying handwritten digits. Suppose we have images of the digits as a long vector of pixel brightnesses. Let’s say the images are black and white and are 64x64 pixels. Each pixel represents a dimension. So the world these images live in has 64x64=4,096 dimensions. In this 4,096-dimensional world, k-means clustering allows us to group the images that are close together and assume they represent the same digit, which can achieve [pretty good results](http://ieeexplore.ieee.org/document/6755106/?reload=true) for digit recognition.

So in summary, The k-means clustering problem attempts to divide *n* number of observations into *k* clusters using the Euclidean distance metric, with the objective of minimizing the variance (sum of squares) within each cluster. It is a method of vector quantization, and is useful for feature learning.

Lloyd’s algorithm (iterative cluster agglomeration with centroid updates) is the most common heuristic used to solve the problem, and is relatively efficient, but doesn’t guarantee global convergence. To improve that, people often run the algorithm multiple times using random initial cluster centroids generated by the Forgy or Random Partition methods.

K-means assumes spherical clusters that are separable so that the mean converges towards the cluster center, and also assumes that the ordering of the data points does not matter. The clusters are expected to be of similar size, so that the assignment to the nearest cluster center is the correct assignment.

The heuristics for solving k-means clusters are usually similar to the expectation-maximization (EM) algorithm for Gaussian mixture models.

-------------Hierarchical clustering is next

# **\*\*Hierarchical clustering**

*“Hierarchical clustering says Let’s make a million options become seven options. Or five. Or twenty? whatever, we can decide later.”*

Hierarchical clustering is similar to regular clustering, except that you’re aiming to build a hierarchy of clusters. This can be useful when you want flexibility in how *many* clusters you ultimately want. For example, imagine grouping items on an online marketplace like Etsy or Amazon. On the homepage you’d want a few broad categories of items for simple navigation, but as you go into more specific shopping categories you’d want increasing levels of granularity, i.e. more distinct clusters of items.

In terms of outputs from the algorithm, in addition to cluster assignments you also build a nice tree that tells you about the hierarchies between the clusters. You can then pick the number of clusters you want from this tree.

Here are the steps for hierarchical clustering:

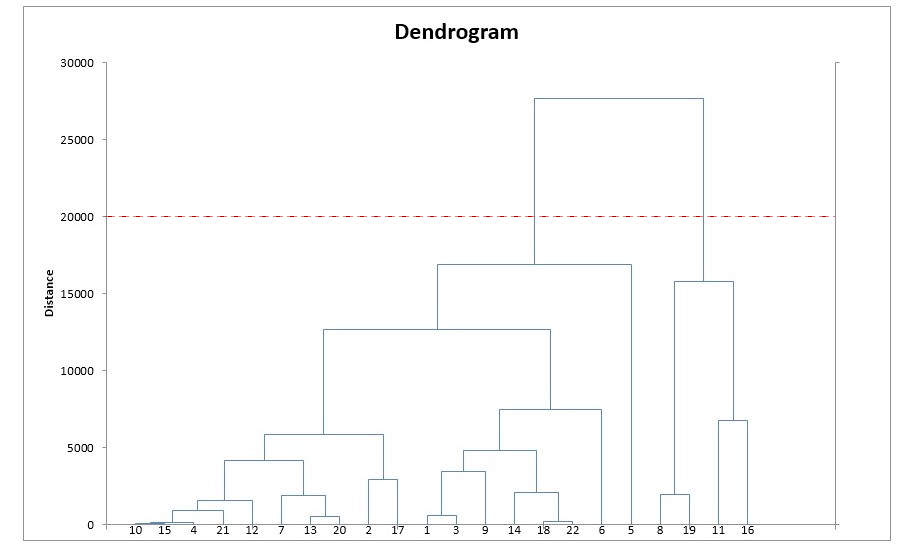
**1. Start with N clusters,** one for each data point.

**2. Merge the two clusters that are closest to each other**.Now you have N-1 clusters.

**3. Recompute the distances between the clusters.** There are several ways to do this (see this [tutorial](https://home.deib.polimi.it/matteucc/Clustering/tutorial_html/hierarchical.html) for more details). One of them (called average-linkage clustering) is to consider the distance between two clusters to be the average distance between all their respective members.

**4. Repeat steps 2 and 3 until you get one cluster of N data points.** You get a tree (also known as a **dendrogram**) like the one below.

**5. Pick a number of clusters and draw a horizontal line in the dendrogram.** For example, if you want k=2 clusters, you should draw a horizontal line around “distance=20000.” You’ll get one cluster with data points 8, 9, 11, 16 and one cluster with the rest of the data points. In general, the number of clusters you get is the number of intersection points of your horizontal line with the vertical lines in the dendrogram.



Hierarchical cluster analysis (HCA) can be agglomerative (you build the clusters bottom-up starting with individual points and ending with a single cluster) or divisive (you start with a single cluster and break it up until you wind up with individual points). If you’re lucky you can find an intermediate stage of the clustering process that reflects a meaningful classification.

The clustering process is usually displayed as a dendrogram (tree diagram that you see on your screen). HCA algorithms tend to use significant compute time and memory resources. Because of this, the algorithm is usually relegated to relatively small data sets.

HCA algorithms can use various metrics and linkage criteria. Euclidian distance and squared Euclidian distance are both common for numeric data; Hamming distance and Levenshtein distance are common for non-numeric data. Single-linkage and complete linkage are common; both of these can simplify the clustering algorithms (SLINK and CLINK respectively). SLINK is one of the few clustering algorithms guaranteed to find an optimum solution.

------------------------------Dimensionality Reduction is next

# **Dimensionality reduction**

*“It is not the daily increase, but the daily decrease. Hack away at the unessential.” — A quote by Bruce Lee that explains*

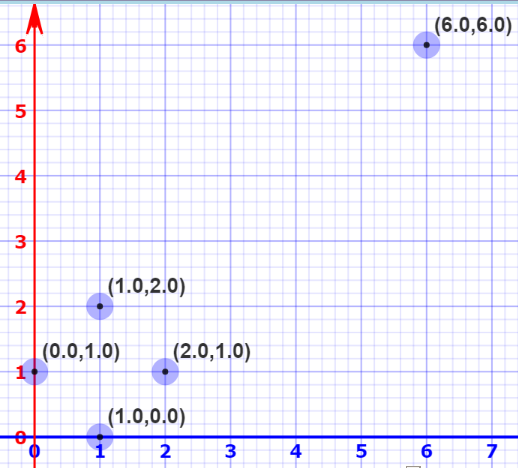
Dimensionality reduction looks a lot like compression. This is about trying to reduce the complexity of the data while keeping as much of the relevant structure as possible. If you take a simple 128 x 128 x 3 pixels image (length x width x RGB value), that’s 49,152 dimensions of data. If you’re able to reduce the dimensionality of the space in which these images live without destroying too much of the meaningful content in the images, then you’ve done a good job at dimensionality reduction.

We’ll take a look at two common techniques in practice: **principal component analysis** and **singular value decomposition**.

# **Principal component analysis (PCA)**

First, a little linear algebra refresher — let’s talk about **spaces** and **bases**.

You’re familiar with the coordinate plane with origin O(0,0) and **basis vectors** i(1,0) and j(0,1). It turns out you can choose a completely different basis and still have all the math work out. For example, you can keep O as the origin and choose the basis to vectors i’=(2,1) and j’=(1,2). If you have the patience for it, you’ll convince yourself that the point labeled (2,2) in the i’, j’ coordinate system is labeled (6, 6) in the i, j system.



Plotted using Mathisfun’s “[Interactive Cartesian Coordinates](https://www.mathsisfun.com/data/cartesian-coordinates-interactive.html)”

This means we can change the basis of a space. Now imagine much higher-dimensional space. Like, 50K dimensions. You can select a basis for that space, and then select only the 200 most significant vectors of that basis. These basis vectors are called **principal components**, and the subset you select constitute a new space that is smaller in dimensionality than the original space but maintains as much of the complexity of the data as possible.

To select the most significant principal components, we look at how much of the data’s variance they capture and order them by that metric.

Another way of thinking about this is that PCA remaps the space in which our data exists to make it more compressible. The transformed dimension is smaller than the original dimension.

By making use of the first several dimensions of the remapped space only, we can start gaining an understanding of the dataset’s organization. This is the promise of dimensionality reduction: reduce complexity (dimensionality in this case) while maintaining structure (variance).

Let’s go over PCA again… Oh… and this is you warning that there is math involved… so … umm. Math alert!

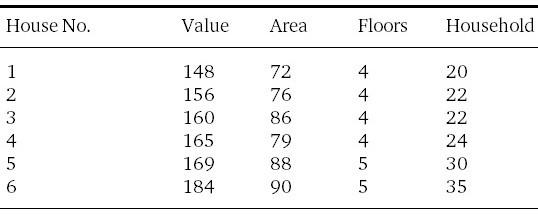
Humans have many senses but we are at our core visual creatures. Seeing is believing is human mantra. When you have a dataset with more than three dimensions, it becomes impossible to visualize what is going on with our eyes. But who said that these extra dimensions are *really* necessary? Isn’t there a way to somehow reduce it to one, two, or three humanly dimensions? It turns out there is.

The technique that we’ve been discussing: Principal Component Analysis (PCA) is one of the better techniques. It is simple and elegant. Unfortunately, simple doesn’t mean easy to see through and really understand what’s going on. If you have scoured blogs or read a fully mathematical and abstract treatment with no intuition of its significance then you know what I mean. Or it may be the opposite where it was explained with such simplicity, and with no mathematical treatment so it appeared to be “magic”. Or maybe a little bit of both, with a conceptual gap in the middle that doesn’t connect intuition with rigor. I’ll try to avoid this here.

The main objective of dimensionality reduction is this: **find a low-dimensional representation of the data that retains as much information as possible.** If you are taking notes then that is a point that you want to hold on to. Dimensionality reduction throuh PCA allows you to **find a low-dimensional representation of the data that retains as much information as possible. If I am trying to predict what type of flower exists based on** That’s a pretty bold statement. Let’s see what it means.

# **It simply means that we are Getting Rid Of The Unnecessary.**

Suppose we have the following dataset for house values, or prices, in thousand dollars, of a certain district:



You can see that we have a Dataset with 4 features per item on your screen.

This data set has 4 dimensions and is impossible to visualize graphically as a whole. However, if you carefully study the relationship of the features to each other and to themselves, you’ll notice that not all features are equally important.

For instance, can you characterize each house by its floor count? Does the number of floors **help to distinguish** one house from another? It doesn’t seem so, since they are almost equal, that is, they have **low variance**, namely, *(sigma squared)*σ² = 0.2,andtherefore this data point isn’t very helpful. What about households? It doesn’t vary much, but its variance is certainly more than floors ((sigma squared) σ² = 28), and thus is **more** helpful. Now, for the last two features, area (σ² = 43) and value (σ² = 127), they vary much more, and hence they are more representative than the other two of our data.

But there’s something we can do to fully exploit each feature to the maximum without sacrificing much accuracy. So far, we have studied each feature individually. What about their **relation** to each other?If you look carefully at the first two features, value and area, you’ll notice that the value is **roughly double** the area. This is incredibly useful, for we can now deduce one feature from another, and need only one instead of two. This property is called **covariance**. The higher the covariance, the more correlated the two features are, which implies redundancy in the data, since there’s more information than needed because we can deduce one feature from another. Covariance is a concept that you are going to want to remember.

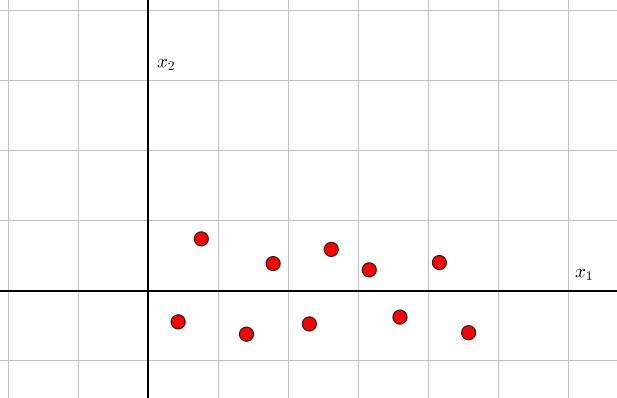
From the what I just said, it should be obvious that:

* It’s a *good thing* to have features with *high variance*, since they will be more informative and more important when it comes to making a prediction.
* It’s a *bad thing* to have highly correlated features, or *high covariance*, since they can be deduced from one another with little loss in information, and thus keeping them together is redundant.

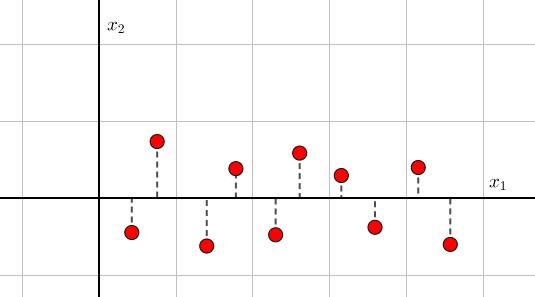
And that’s just what PCA does. It tries to find another representation of the data (another set of features), such that the features in this representation have the highest possible variance and lowest possible covariance. But this shouldn’t make sense now, until we see it with our eyes…

# [**Hussein Abdullatif**](https://towardsdatascience.com/@abdullatif.h?source=post_page-----a8c9ec7b7e79----------------------) **had a great exercise that explains PCA. He said: Let’s imagine that I am interviewing Ms. PCA**

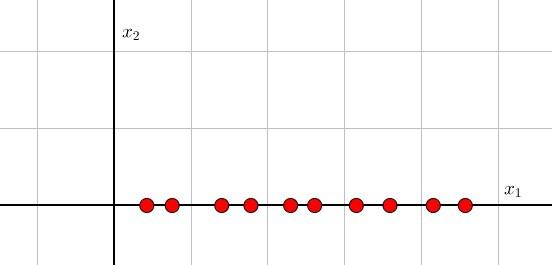
In an interview with PCA, I asked her the following question: “What does your perfect dataset look like?” She showed me this:



And it makes sense — the data can essentially be **reduced to a single line.**Observe: the variance along the *x*1 *direction* is very high compared to the *x*2 direction, which means we can safely drop the *x*2 features without much damage (by saying the “variance along *x*1 direction”, I mean the variance of the first feature, since we chose to represent it with the *x*1-axis, same for *x*2). Moreover, *x*1 doesn’t seem to depend on *x*2 at all, it just goes on increasing regardless of the value of *x*2, which implies relatively *low covariance*. Now as to why this is actually perfect for PCA, it is simply because all she needs to do is this:



Projecting the dataset on the x1 axis.



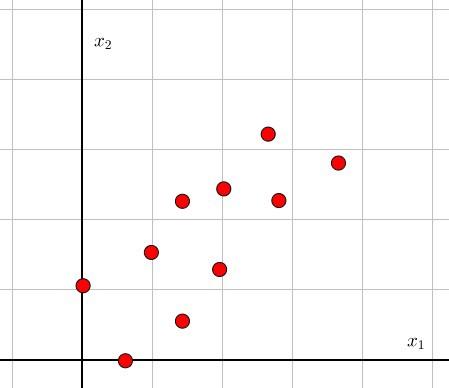
After projection, the data only has one dimension.

It’s crucial to understand what happened. Because *x*1 is much more important than *x*2 (according to the two criteria stated earlier), we have decided to keep only *x*1, by **projecting** the data points on its axis, which is equivalent to keeping only the *x*1-coordinate of the points, which is also equivalent to dropping the “unimportant” feature *x*2. And now we have a 1D dataset instead of 2D!

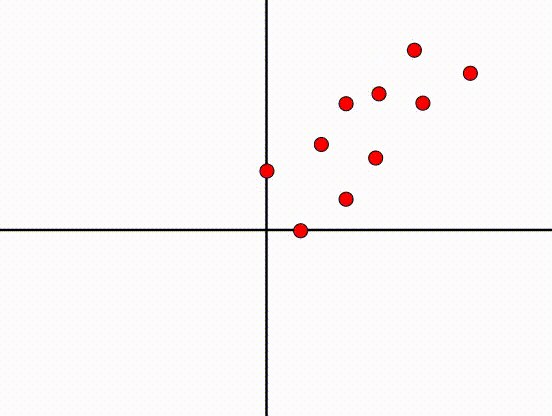
This is essentially **Dimensionality Reduction:** finding the best low dimensional representation of the data. Of course, there’ll be some error due to neglecting the second feature, which is represented by the dashed lines above. But we know this error is kept to a minimum since the data lies almost on a line, and it’s the best we can actually do with the given information (more on this later).

# **The Plot Thickens**

“So what about your not-so-perfect dataset? What does that look like?” I asked PCA. Without hesitation, she said: “I really don’t believe there’s such a thing, you know. Everyone is perfect, you only need to change your perspective. Here, take a look at this.”



“Not so easy as before, don’t you think? Nope. Tilt your head”, she said.



“It’s actually the same ‘perfect’ dataset I showed you previously, just rotated 45 degrees. All we have to do is rotate our own axes to align with the dataset, and proceed as before: projecting on the *new x*1 axis, and omitting the *new x*2 axis”

It’s crucial to pause and ponder about this unexpected rotation of events. Like what engineers usually do, PCA reduces the not-so-perfect problem of a dataset that isn’t “aligned”, to a perfect, “aligned” problem that is easy to solve.

How does this happen? Essentially, PCA tries to find another set of axes such that the variance along that axis is as large as possible. When I said “variance along *x*1 axis”, I meant the variance of the feature *x*1. But after rotating our axes, the axes lost their meaning — they no longer represent *x*1 or *x*2. Rather, they represent a **linear combination** of both. To see how, notice that the new *x*1 axis in the above figure is a line that satisfies the equation : *x*2=*x*1 or *x*1-*x*2 = 0 (with respect to the old feature axes). And *x*1-*x*2=0 is nothing more than a linear combination of the features, each weighed by 1 and -1. This rotated axis now represents a **new feature**, call it *z*1, that is:



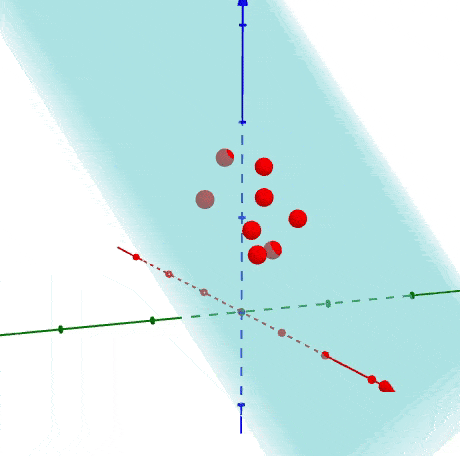
Likewise, the rotated *x*2-axis now represents a new feature, call it *z*2:



These two new directions, *z*1 and *z*2, are the **Principal Components** of the dataset. The *1st principal component*, *z*1, is the one with the highest variance, and hence the one that is most important, carries the most information, and in a sense that which the data *hinges* on. The variance in the direction of the first principal component is now interpreted as the variance of the new, made-up feature *z*1. As for the *2nd principal component*, *z*2, it’s just the one with the second most variance that is perpendicular to the 1st.

As its name implies, Principal Component Analysis is all about finding these principal components, so that we may utilize the first few of them that carry the most variance to represent our data, just like what we did when we projected the prefect dataset into a line.

It’s easy to see how this can generalize to more than two dimensions. Instead of projecting into lines, we project into planes, and our perfect dataset in 3D is now lying *approximately* on a plane (or maybe, even better, a line):



# **Magical Lines And Where To Find Them**

Now comes the real deal: How to find these principal components?

[I’ll leave that matter to the next](https://towardsdatascience.com/svd-8c2f72e264f) episode on SVD, where we shall develop that master formula:



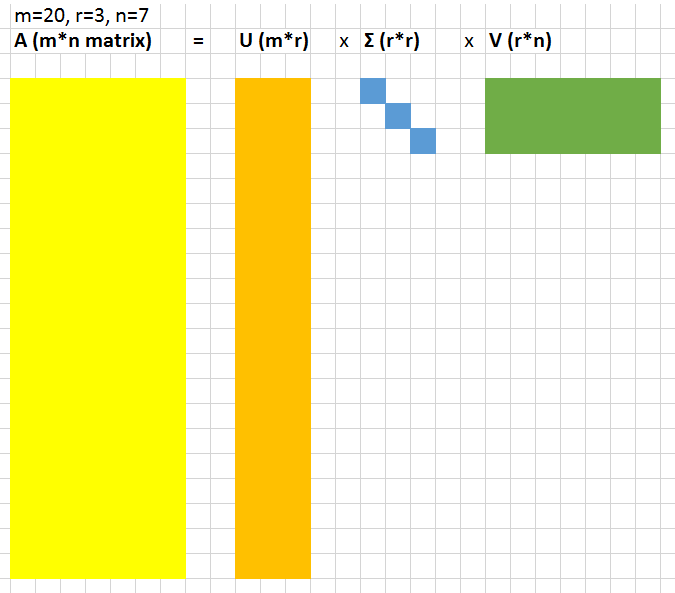
And along the way, we will discover deep insights on how we can turn our qualitative intuition developed here into an elegant mathematical structure that generalizes to all dimensions.

---------SVD is next

**Singular value decomposition (SVD)**

Let’s represent our data like a big A = *m* x *n* matrix. SVD is a computation that allows us to decompose that big matrix into a product of 3 smaller matrices (U=*m* x *r*, diagonal matrix Σ=*r* x *r*, and V=*r* x *n* where r is a small number).

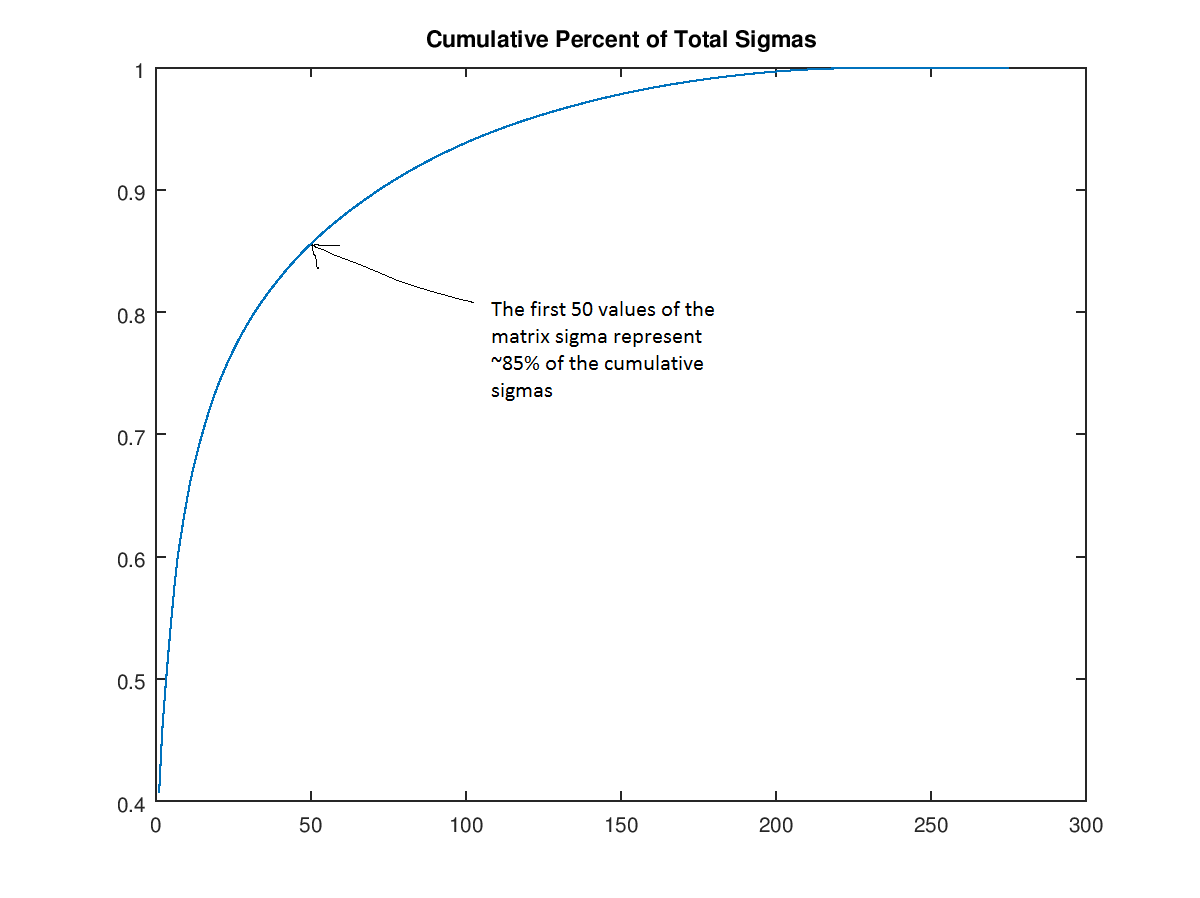
Here’s a more visual illustration of that product to start with:



The values in the r\*r diagonal matrix Σ are called singular values. What’s cool about them is that these singular values can be used to compress the original matrix. If you drop the smallest 20% of singular values and the associated columns in matrices U and V, you save quite a bit of space and still get a decent representation of the underlying matrix.

To examine what that means more precisely, let’s work with this image of a dog:

We’ll use the code written in Andrew Gibiansky’s [post](http://andrew.gibiansky.com/blog/mathematics/cool-linear-algebra-singular-value-decomposition/) on SVD. First, we show that if we rank the singular values (the values of the matrix Σ) by magnitude, the first 50 singular values contain 85% of the magnitude of the whole matrix Σ.



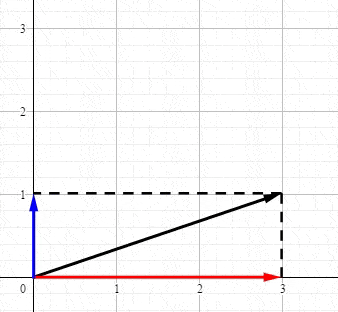
We can use this fact to discard the next 250 values of sigma (i.e., set them to 0) and just keep a “rank 50” version of the image of the dog. Here, we create a rank 200, 100, 50, 30, 20, 10, and 3 dog. Obviously, the picture is smaller, but let’s agree that the rank 30 dog is still good. Now let’s see how much compression we achieve with this dog. The original image matrix is 305\*275 = 83,875 values. The rank 30 dog is 305\*30+30+30\*275=17,430 — almost 5 times fewer values with very little loss in image quality. The reason for the calculation above is that we also discard the parts of the matrix U and V that get multiplied by zeros when the operation UΣ’V is carried out (where Σ’ is the modified version of Σ that only has the first 30 values in it).



Unsupervised learning is often used to preprocess the data. Usually, that means compressing it in some meaning-preserving way like with PCA or SVD before feeding it to a deep neural net or another supervised learning algorithm.

If you are ok so far then buckle up… we are going to take a more math centric view of SVD in this review.

Back in elementary mechanics, you learned that any force vector can be decomposed into its components along the *x* and *y* axes:



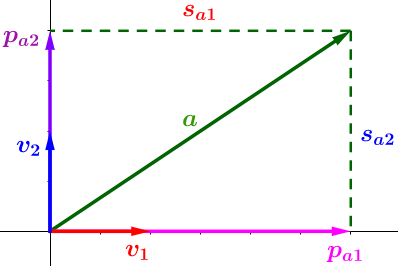
***Congratulations.*** *Now you know what singular value decomposition is.*

For it’s disappointing that almost every tutorial of SVD **makes it more complicated than necessary, while the core idea is most simple.**

Since mathematics is just the art of assigning different names to the same concept, **SVD is nothing more than decomposing vectors onto orthogonal axes** — we just decided it may need a more deluxe name.

Let’s see how this is the case.

When the vector (***a***) is decomposed, we get 3 pieces of information:



1. The **directions** of projection — the **unit** vectors (***v*₁** and ***v*₂**) **representing the directions** onto which we project (decompose). In the above they’re the *x* and *y* axes, but can be any other orthogonal axes.
2. The **lengths** of projection (the **line segments** s***ₐ***₁ and s***ₐ***₂) — which tell us how much of the vector is **contained** in each direction of projection (more of vector ***a*** is leaning on the direction ***v*₁** than it is on ***v*₂**, hence s***ₐ***₁>s***ₐ***₂).
3. The **vectors** of projection (***pₐ*₁** and ***pₐ*₂**) — which are used to **reconstruct** the original vector ***a*** by adding them together (as a vector sum), and for which it’s easy to verify that ***pₐ*₁**=s***ₐ***₁\****v*₁** and ***pₐ*₂**=s***ₐ₂***\****v*₂ — So they’re redundant, as they can be deduced from the former 2 pieces.**

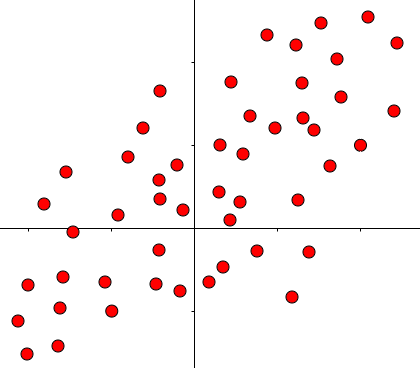
# **Critical Conclusion:**

Any vector can be expressed in terms of:

1. Projection directions unit vectors (v₁, v₂, …).

2. The lengths of projections onto them (sₐ₁, sₐ₂, …).

All what SVD does is **extend this conclusion** to more than one vector (or point) and to all dimensions :



An example of a dataset (**a point can be considered a vector through the origin**).

Now it becomes a matter of knowing how to handle this mess.

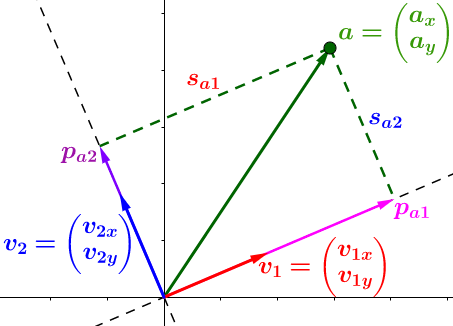
# **How To Handle This Mess**

We can’t handle that mess without first handling a single vector!

If you look at many generalizations done in mathematics, you’ll find they primarily utilize **matrices.**

So we have to find a way to express the operation of vector decomposition using matrices.

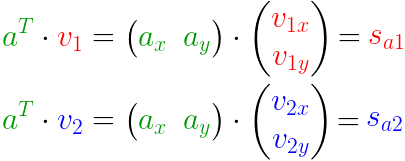
It turns out to be a natural thing to do:



Same figure as before, but tilting the axes of projection to convince you they aren’t confined to x and y. (aₓ and aᵧ are the coordinates of vector **a**, put into a column matrix (aka column vector), as per convention. Same for v₁ and v₂).

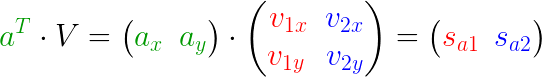
We want to decompose (project) the vector ***a*** along unit vectors ***v*₁** and ***v*₂**.

You may already know (especially if you’ve watched [this](https://www.youtube.com/watch?v=LyGKycYT2v0)) that projection is done by the **dot product** — it gives us the **lengths** of projection (s***ₐ***₁ and s***ₐ***₂):



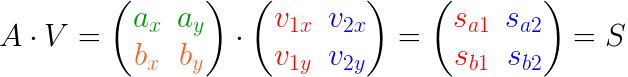
Projecting (a) onto v1 and v2.

But that’s redundant. We can utilize the efficiency of matrices…



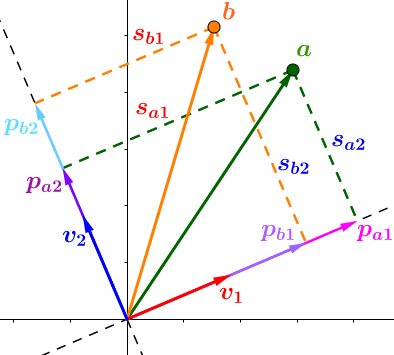
…to write both equations in one go, by adding an **extra column** for each unit vector.

We can even add more points…

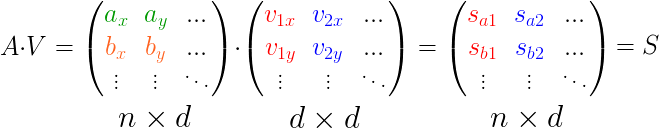


…by adding an **extra row** for each point. **S** is the matrix containing the lengths of projections.

Here’s how it looks like, after adding that point ***b***:

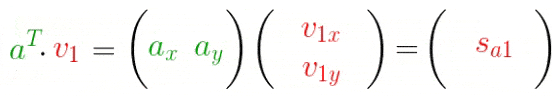


It’s now easy to generalize to any number of points and dimensions:

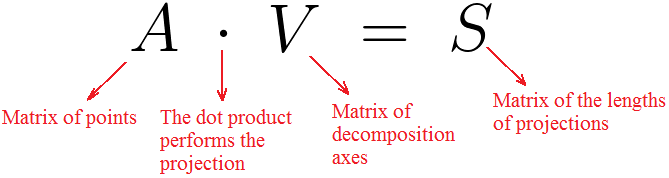


**n** = no. of points, **d** = no. of dimensions, **A** = matrix containing points, **V** = matrix containing the decomposition axes, **S** = matrix containing lengths of projection.

Mathematical elegance at its best.



# **Recap:-**

****

The dot product in this case is just **ordinary matrix multiplication**.

That’s exactly the same as saying:



Because **V** contains orthonormal columns, its inverse = its transpose (property of orthogonal matrices).

Which is all what SVD says (remember the Critical Conclusion):

Any set of vectors (A) can be expressed in terms of their lengths of projections (S) on some set of orthogonal axes (V).

**However, we are not quite there yet.** The conventional SVD formula says:



But that just means we want to see how:



And that’s what we’re going to do.

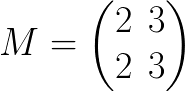
If you look carefully at the matrix ***S***, you’ll discover it consists of:



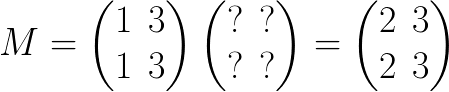
It turns out (for reasons to be seen later) that it’s best if we could **normalize** these column vectors, i.e. make them of **unit length**.

This is done by doing the equivalent of **dividing each column vector by its magnitude, but in matrix form**.

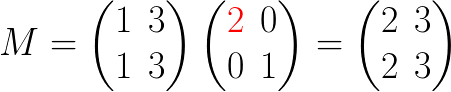
But first, a numerical example to see how this “division” thing is done.



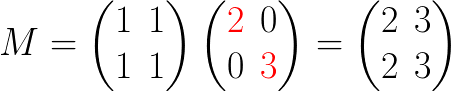
Let’s say we want to divide the ***1st*** column of ***M*** by **2**. Wewill surely have to **multiply by another matrix** to preserve the equality:



It’s straightforward to verify that the unknown matrix is nothing more than the **identity matrix, with the *1st* element replaced by the divisor** **= 2**:



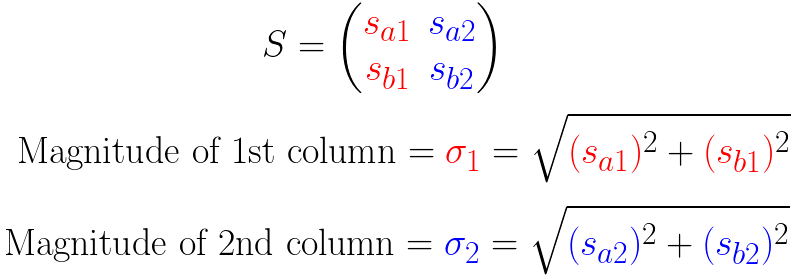
Dividing the ***2nd*** column by **3** now becomes a direct matter — just replace the ***2nd*** element of the identity matrix by **3**:



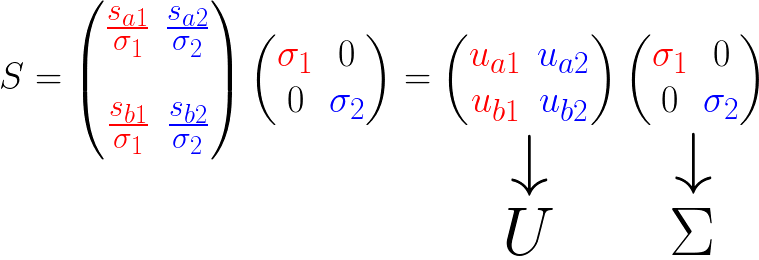
It should be obvious how this operation can be generalized to any matrix of any size.

We now want to apply the above “division” concept to the matrix ***S***.

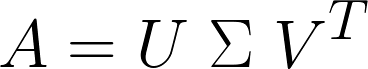
To normalize the columns of ***S***, we divide them by their magnitude…



…by doing with ***S*** what we did with ***M*** in the example above:



# **Finally…**

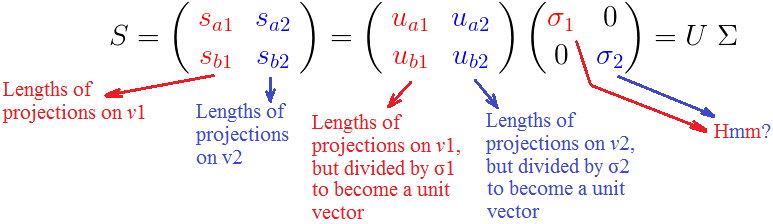
****

Singular Value Decomposition

Of course, some fine details and rigorous mathematics were, justifiably, swept under the rug, in order to *not* distract from the core concept.

# **Interpretation**

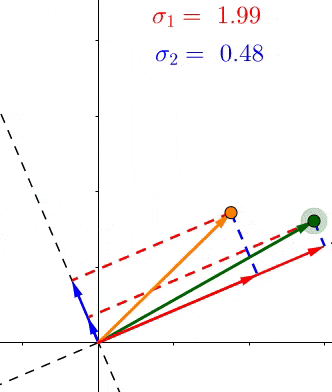
Let’s talk about this ***U*** and **Σ** …



What about the sigmas? Why did we burden ourselves with normalizing ***S*** to find them?

We’ve already seen that (σ***ᵢ***) is the **square root of the sum of squared projection lengths,** of all points,onto the *i*th unit vector ***vᵢ***.

What does that mean?



Red segments = projections on v1. Blue segments = projections on v2. **The closer the points to a specific axis of projection, the larger the value of the corresponding σ.**

Since the sigmas contain, in their definition, the sum of projection lengths onto a specific axis, **they represent how close all the points are to that axis.**

E.g. if σ₁ > σ₂, then most points are closer to ***v*₁** than ***v*₂**, and vice versa.

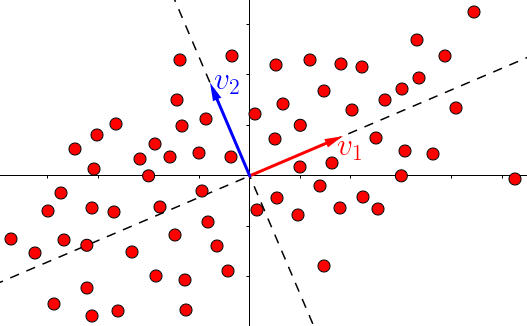
That’s of immense utility in the myriad applications of SVD.

# **The Main Application**

The algorithms of finding the SVD of a matrix don’t choose the projection directions (columns of matrix ***V***) randomly.

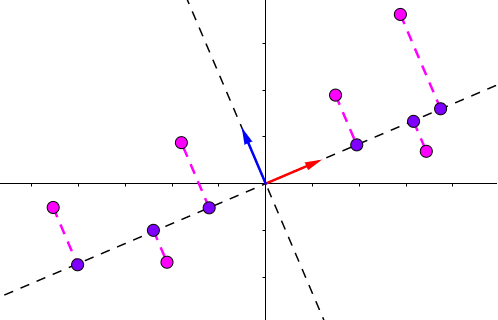
They choose them to be the Principal Components of the dataset (matrix A).

If you’ve read my [first article](https://towardsdatascience.com/https-medium-com-abdullatif-h-dimensionality-reduction-for-dummies-part-1-a8c9ec7b7e79), you know very well what the principal components are…



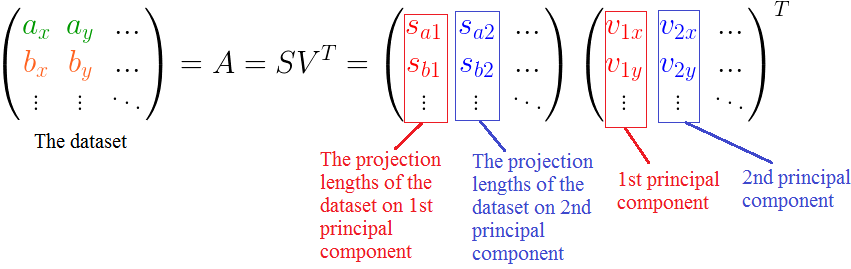
…they’re the lines of largest variation (largest variance).

You also know from [the same](https://towardsdatascience.com/https-medium-com-abdullatif-h-dimensionality-reduction-for-dummies-part-1-a8c9ec7b7e79) that the goal of **dimensionality reduction** is to **project the dataset on the line (or plane) of largest variance**:

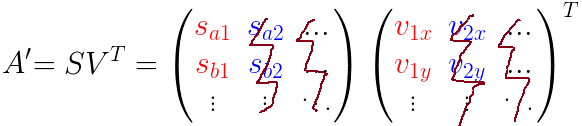


Magenta: points before projection. Violet: points after projection (reduced dimensionality).

Now the act of projecting the dataset using SVD becomes a snap, since **all the points are *already projected*** (decomposed) on all the principal components (the ***vᵢ*** unit vectors):



So, for example, to project the dataset on the 1st principal component…



**…all we have to do is remove all columns not related to the 1st principal component**. The projected dataset in now **A’**.

Multiplying the two matrices (S and V*ᵀ* above) results in the matrix A′ containing the projected points (violet) in the last graph.

And that’s it…for now.

# **Onwards!**

Now that you’ve finished this section, you’ve earned an awful, horrible, never-to-be-mentioned-again joke about unsupervised learning. Here goes…

*Person-in-joke-#1: Y would u ever need to use unsupervised tho?*

*Person-in-joke-#2: Y? there’s no Y.*

Next up… [Part 4: Neural Networks & Deep Learning](https://medium.com/@v_maini/neural-networks-deep-learning-cdad8aeae49b)!

# **Practice materials & further reading**

## **3a — k-means clustering**

*Play around with this clustering* [*visualization*](https://www.naftaliharris.com/blog/visualizing-k-means-clustering/) *to build intuition for how the algorithm works. Then, take a look at this implementation of* [*k-means clustering for handwritten digits*](https://github.com/Datamine/MNIST-K-Means-Clustering) *and the associated tutorial.*

## **3b — SVD**

*For a good reference on SVD, go no further than Andrew Gibiansky’s* [*post*](http://andrew.gibiansky.com/blog/mathematics/cool-linear-algebra-singular-value-decomposition/)*.*