1. Supervised learning involves finding patterns in unlabelled data, whereas unsupervised learning involves generalizing over new examples after presentation with labeled data

TRUE

FALSE

Unsupervised learning can be used for dimensionality reduction

TRUE

FALSE

Image segmentation is one thing that can be achieved by supervised learning only

TRUE

FALSE

Outlier detection is one application of clustering

TRUE

FALSE

Which of the following is NOT an application of unsupervised learning?

a. Getting Google image search categories

b. Author clustering

c. Dimensionality reduction

d. Predicting future stock prices

Not only can clustering be used prior to using a supervised learning technique, this is often very sensible

TRUE

FALSE

We can only identify patterns we already knew existed with clustering algorithms if we are to identify any at all

TRUE

FALSE

Which of the following techniques or procedures involves supervised, not unsupervised, learning?

a. Support Vector Machines

b. K-Means

c. Mean-shift clustering

d. Hierarchical clustering

What does dimensionality reduction always consist in?

a. Reducing the number of fields in the data

b. Reducing the number of rows in the data

c. Reducing the number of fields and the number of rows in the data

d. None of the above

Finding the location for a new hospital counts as a possible application of clustering

TRUE

FALSE

March 28, 2017 / [#Machine Learning](https://www.freecodecamp.org/news/tag/machine-learning/)

# An introduction to clustering algorithms



Take a look at the image below. It’s a collection of bugs and creepy-crawlies of different shapes and sizes. Take a moment to categorize them by similarity into a number of groups.

This isn’t a trick question. Start with grouping the spiders together.

Images via Google Image Search, labelled for reuse

Done? While there’s not necessarily a “correct” answer here, it’s most likely you split the bugs into four *clusters*. The spiders in one cluster, the pair of snails in another, the butterflies and moth into one, and the trio of wasps and bees into one more.

That wasn’t too bad, was it? You could probably do the same with twice as many bugs, right? If you had a bit of time to spare — or a passion for entomology — you could probably even do the same with a hundred bugs.

For a machine though, grouping ten objects into however many meaningful clusters is no small task, thanks to a mind-bending branch of maths called [combinatorics](https://en.wikipedia.org/wiki/Bell_number), which tells us that are 115,975 different possible ways you could have grouped those ten insects together.

Had there been twenty bugs, there would have been [over fifty trillion](https://www.wolframalpha.com/input/?i=bell%27s+number+(20)) possible ways of clustering them.

With a hundred bugs — there’d be many times more solutions than there are [particles in the known universe](https://www.wolframalpha.com/input/?i=particles+in+universe).

How many times more? By my calculation, approximately [five hundred million billion billion times more](https://www.wolframalpha.com/input/?i=BellB%5B100%5D+%2F+number+of+particles+in+universe). In fact, there are more than [four million billion *googol*](https://www.wolframalpha.com/input/?i=BellB%5B100%5D+%2F+googol) solutions ([what’s a googol?](https://www.wolframalpha.com/input/?i=googol)).

For just a hundred objects.

Almost all of those solutions would be meaningless — yet from that unimaginable number of possible choices, you pretty quickly found one of the very few that clustered the bugs in a useful way.

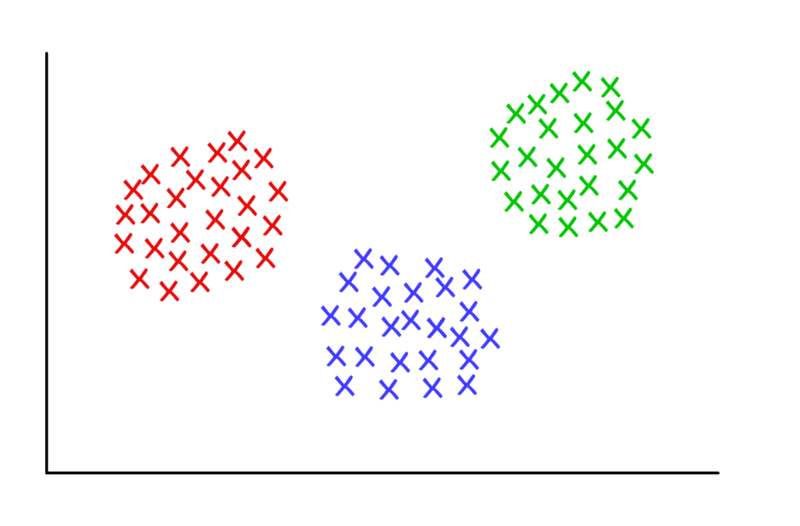
Us humans take it for granted how good we are categorizing and making sense of large volumes of data pretty quickly. Whether it’s a paragraph of text, or images on a screen, or a sequence of objects — humans are generally fairly efficient at making sense of whatever data the world throws at us.

Given that a key aspect of developing A.I. and machine learning is getting machines to quickly make sense of large sets of input data, what shortcuts are there available?

Here, you can read about three clustering algorithms that machines can use to quickly make sense of large datasets. This is by no means an exhaustive list — there are other algorithms out there — but they represent a good place to start!

You’ll find for each a quick summary of when you might use them, a brief overview of how they work, and a more detailed, step-by-step worked example. I believe it helps to understand an algorithm by actually carrying out yourself.

If you’re *really keen*, you’ll find the best way to do this is with pen and paper. Go ahead — nobody will judge!

Three suspiciously neat clusters, with K = 3

**K-means clustering**

**Use when...**

…you have an idea of how many groups you’re expecting to find *a priori*.

**How it works**

The algorithm randomly assigns each observation into one of *k* categories, then calculates the *mean* of each category. Next, it reassigns each observation to the category with the closest mean before recalculating the means. This step repeats over and over until no more reassignments are necessary.

**Worked Example**

Take a group of 12 football (or ‘soccer’) players who have each scored a certain number of goals this season (say in the range 3–30). Let’s divide them into separate clusters — say three.

**Step 1** requires us to randomly split the players into three groups and calculate the means of each.

Group 1

Player A (5 goals),

Player B (20 goals),

Player C (11 goals)

Group Mean = (5 + 20 + 11) / 3 = 12 goals

Group 2

Player D (5 goals),

Player E (3 goals),

Player F (19 goals)

Group Mean = 9 goals

Group 3

Player G (30 goals),

Player H (3 goals),

Player I (15 goals)

Group Mean = 16 goals

**Step 2:** For each player, reassign them to the group with the closest mean. E.g., Player A (5 goals) is assigned to Group 2 (mean = 9). Then recalculate the group means.

Group 1 (Old Mean = 12 goals)

Player C (11 goals)

New Mean = 11 goals

Group 2 (Old Mean = 9 goals)

Player A (5 goals),

Player D (5 goals),

Player E (3 goals),

Player H (3 goals)

New Mean = 4 goals

Group 3 (Old Mean = 16 goals)

Player G (30 goals),

Player I (15 goals),

Player B (20 goals),

Player F (19 goals)

New Mean = 21 goals

**Repeat** Step 2 over and over until the group means no longer change. For this somewhat contrived example, this happens on the next iteration. **Stop!** You have now formed three clusters from the dataset!

Group 1 (Old Mean = 11 goals)

Player C (11 goals),

Player I (15 goals)

Final Mean = 13 goals

Group 2 (Old Mean = 4 goals)

Player A (5 goals),

Player D (5 goals),

Player E (3 goals),

Player H (3 goals)

Final Mean = 4 goals

Group 3 (Old Mean = 21 goals)

Player G (30 goals),

Player B (20 goals),

Player F (19 goals)

Final Mean = 23 goals

With this example, the clusters could correspond to the players’ positions on the field — such as defenders, midfielders and attackers.

K-means works here because we could have reasonably expected the data to fall naturally into these three categories.

In this way, given data on a range of performance statistics, a machine could do a reasonable job of estimating the positions of players from any team sport — useful for sports analytics, and indeed any other purpose where classification of a dataset into predefined groups can provide relevant insights.

**Finer details**

There are several variations on the algorithm described here. The initial method of ‘seeding’ the clusters can be done in one of several ways.

Here, we randomly assigned every player into a group, then calculated the group means. This causes the initial group means to tend towards being similar to one another, which ensures greater repeatability.

An alternative is to seed the clusters with just one player each, then start assigning players to the nearest cluster. The returned clusters are more sensitive to the initial seeding step, reducing repeatability in highly variable datasets.

However, this approach may reduce the number of iterations required to complete the algorithm, as the groups will take less time to diverge.

An obvious limitation to K-means clustering is that you have to provide *a priori* assumptions about how many clusters you’re expecting to find.

There are methods to assess the fit of a particular set of clusters. For example, the Within-Cluster [Sum-of-Squares](https://en.wikipedia.org/wiki/Partition_of_sums_of_squares) is a measure of the variance within each cluster.

The ‘better’ the clusters, the lower the overall WCSS.

**Hierarchical clustering**

**Use when...**

…you wish to uncover the underlying relationships between your observations.

**How it works**

A distance matrix is computed, where the value of cell (*i, j)* is a distance metric between observations *i* and *j*.

Then, pair the closest two observations and calculate their average. Form a new distance matrix, merging the paired observations into a single object.

From this distance matrix, pair up the closest two observations and calculate their average. Repeat until all observations are grouped together.

**Worked example**

Here’s a super-simplified dataset about a selection of whale and dolphin species. As a trained biologist, I can assure you we normally use much more detailed datasets for things like [reconstructing phylogeny](https://en.wikipedia.org/wiki/Phylogenetic_tree).

For now though, we’ll just look at the typical body lengths for these six species. We’ll be using just two repeated steps.

Species Initials Length(m)

Bottlenose Dolphin BD 3.0

Risso's Dolphin RD 3.6

Pilot Whale PW 6.5

Killer Whale KW 7.5

Humpback Whale HW 15.0

Fin Whale FW 20.0

**Step 1:** compute a distance matrix between each species. Here, we’ll use the [Euclidean distance](https://en.wikipedia.org/wiki/Euclidean_distance) — how far apart are the data points?

Read this exactly as you would a distance chart in a road atlas. The difference in length between any pair of species can be looked up by reading the value at the intersection of the relevant row and column.

BD RD PW KW HW

RD 0.6

PW 3.5 2.9

KW 4.5 3.9 1.0

HW 12.0 11.4 8.5 7.5

FW 17.0 16.4 13.5 12.5 5.0

**Step 2:** Pair up the two closest species. Here, this will be the Bottlenose & Risso’s Dolphins, with an average length of 3.3m.

**Repeat** Step 1 by recalculating the distance matrix, but this time merge the Bottlenose & Risso’s Dolphins into a single object with length 3.3m.

[BD, RD] PW KW HW

PW 3.2

KW 4.2 1.0

HW 11.7 8.5 7.5

FW 16.7 13.5 12.5 5.0

**Next**, repeat Step 2 with this new distance matrix. Here, the smallest distance is between the Pilot & Killer Whales, so we pair them up and take their average — which gives us 7.0m.

**Then**, we repeat Step 1 — recalculate the distance matrix, but now we’ve merged the Pilot & Killer Whales into a single object of length 7.0m.

[BD, RD] [PW, KW] HW

[PW, KW] 3.7

HW 11.7 8.0

FW 16.7 13.0 5.0

**Next**, repeat Step 2 with this distance matrix. The smallest distance (3.7m) is between the two merged objects — so now merge them into an even bigger object, and take the average (which is 5.2m).

**Then**, repeat Step 1 and compute a new distance matrix, having merged the Bottlenose & Risso’s Dolphins with the Pilot & Killer Whales.

[[BD, RD] , [PW, KW]] HW

HW 9.8

FW 14.8 5.0

**Next**, repeat Step 2. The smallest distance (5.0m) is between the Humpback & Fin Whales, so merge them into a single object, and take the average (17.5m).

**Then**, it’s back to Step 1 — compute the distance matrix, having merged the Humpback & Fin Whales.

[[BD, RD] , [PW, KW]]

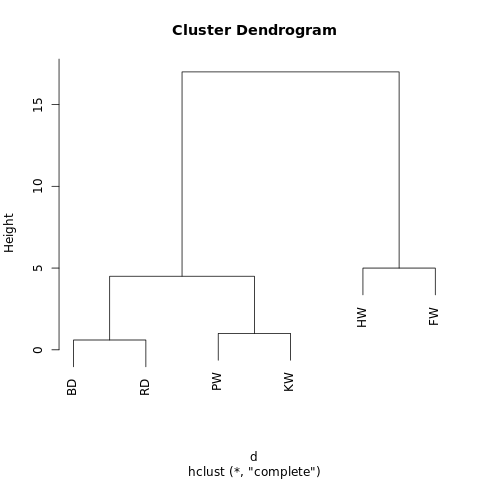
[HW, FW] 12.3

**Finally,** repeat Step 2 — there is only one distance (12.3m) in this matrix, so pair everything into one big object. Now you can **stop!** Look at the final merged object:

[[[BD, RD],[PW, KW]],[HW, FW]]

It has a nested structure (think [JSON](http://json.org/example.html)), which allows it to be drawn up as a tree-like graph, or 'dendrogram'.

It reads in much the same way a family tree might. The nearer two observations are on the tree, the more similar or closely-related they are taken to be.

A no-frills dendrogram generated at [R-Fiddle.org](http://www.r-fiddle.org/)

The structure of the dendrogram gives insight into how the dataset is structured.

In this example, there are two main branches, with Humpback Whale and Fin Whale on one side, and the Bottlenose Dolphin/Risso’s Dolphin and Pilot Whale/Killer Whale on the other.

In evolutionary biology, much larger datasets with many more specimens and measurements are used in this way to infer taxonomic relationships between them.

Outside of biology, hierarchical clustering has applications in data mining and machine learning contexts.

The cool thing is that this approach requires no assumptions about the number of clusters you’re looking for.

You can split the returned dendrogram into clusters by “cutting” the tree at a given height. This height can be chosen in a number of ways, depending on the resolution at which you wish to cluster the data.

For instance, looking at the dendrogram above, if we draw a horizontal line at height = 10, we’d intersect the two main branches, splitting the dendrogram into two sub-graphs. If we cut at height = 2, we’d be splitting the dendrogram into three clusters.

**Finer details**

There are essentially three aspects in which hierarchical clustering algorithms can vary to the one given here.

Most fundamental is the approach — here, we have used an *agglomerative* process, whereby we start with individual data points and iteratively cluster them together until we’re left with one large cluster.

An alternative (but more computationally intensive) approach is to start with one giant cluster, and then proceed to divide the data into smaller and smaller clusters until you’re left with isolated data points.

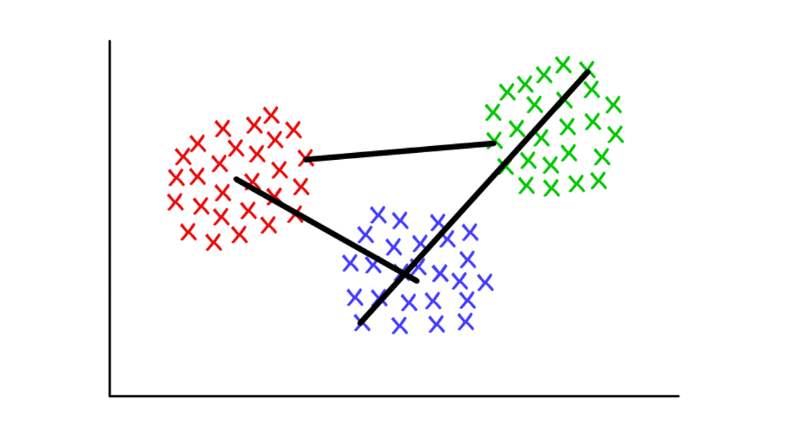
There are also a range of methods that can be used to calculate the distance matrices. For many purposes, the Euclidean distance (think Pythagoras’ Theorem) will suffice, but there are [alternatives](https://en.wikipedia.org/wiki/Metric_(mathematics)) that may be more applicable in some circumstances.

Finally, the *linkage criterion* can also vary. Clusters are linked according to how close they are to one another, but the way in which we define ‘close’ is flexible.

In the example above, we measured the distances between the means (or ‘centroids’) of each group and paired up the nearest groups. However, you may want to use a different definition.

For example, each cluster is made up of several discrete points. You could define the distance between two clusters to be the minimum (or maximum) distance between any of their points — as illustrated in the figure below.

There are still other ways of defining the linkage criterion, which may be suitable in different contexts.

Red/Blue: centroid linkage; Red/Green: minimum linkage; Green/Blue: maximum linkage

**Graph Community Detection**

**Use when**

…you have data that can be represented as a network, or ‘graph’.

**How it works**

A *graph community* is very generally defined as a subset of vertices which are more connected to each other than with the rest of the network.

Various algorithms exist to identify communities, based upon more specific definitions. Algorithms include, but are not limited to: Edge Betweenness, Modularity-Maximsation, Walktrap, Clique Percolation, Leading Eigenvector…

**Worked example**

[Graph theory](https://en.wikipedia.org/wiki/Graph_theory), or the mathematical study of networks, is a fascinating branch of mathematics that lets us model complex systems as an abstract collection of ‘dots’ (or *vertices*) connected by ‘lines’ (or *edges*).

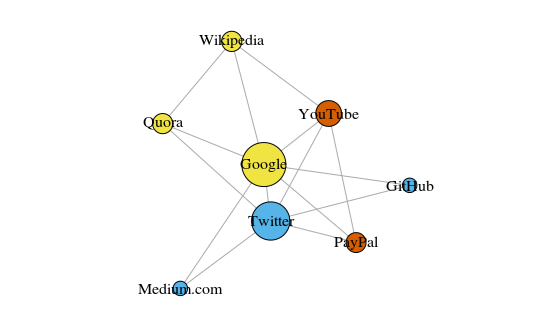
Perhaps the most intuitive case-studies are social networks.

Here, the vertices represent people, and edges connect vertices who are friends/followers. However, any system can be modelled as a network if you can justify a method to meaningfully connect different components.

Among the more innovative applications of graph theory to clustering include feature extraction from image data, and analysing gene regulatory networks.

As an entry-level example, take a look at this quickly put-together graph. It shows the eight websites I most recently visited, linked according to whether their respective Wikipedia articles link out to one another.

You could assemble this data manually, but for larger-scale projects, it’s much quicker to write a Python script to do the same. [Here’s one I wrote earlier](https://raw.githubusercontent.com/pg0408/Medium-articles/master/graph_maker.py).

Graph plotted with ‘igraph’ package for R version 3.3.3

The vertices are colored according to their community membership, and sized according to their *centrality*. See how Google and Twitter are the most central?

Also, the clusters make pretty good sense in the real-world (always an important performance indicator).

The yellow vertices are generally reference/look-up sites; the blue vertices are all used for online publishing (of articles, tweets, or code); and the red vertices include YouTube, which was of course founded by former PayPal employees. Not bad deductions for a machine.

Aside from being a useful way to visualize large systems, the real power of networks comes from their mathematical analysis. Let’s start by translating our nice picture of the network into a more mathematical format. Below is the *adjacency matrix* of the network.

GH Gl M P Q T W Y

GitHub 0 1 0 0 0 1 0 0

Google 1 0 1 1 1 1 1 1

Medium 0 1 0 0 0 1 0 0

PayPal 0 1 0 0 0 1 0 1

Quora 0 1 0 0 0 1 1 0

Twitter 1 1 1 1 1 0 0 1

Wikipedia 0 1 0 0 1 0 0 0

YouTube 0 1 0 1 0 1 0 0

The value at the intersection of each row and column records whether there is an edge between that pair of vertices.

For instance, there is an edge between Medium and Twitter (surprise, surprise!), so the value where their rows/columns intersect is 1. Similarly, there is no edge between Medium and PayPal, so the intersection of their rows/columns returns 0.

Encoded within the adjacency matrix are all the properties of this network — it gives us the key to start unlocking all manner of valuable insights.

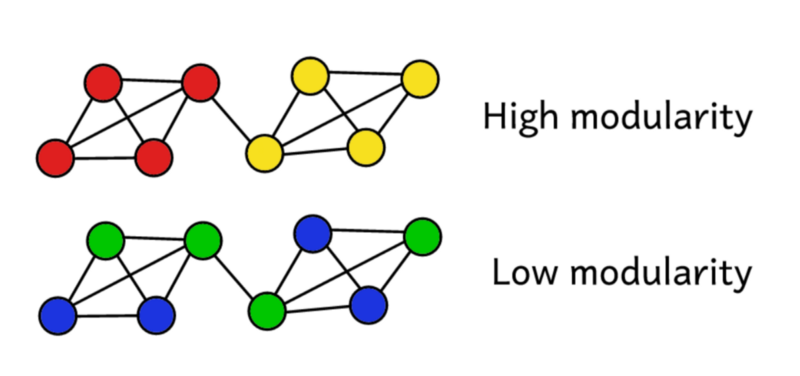
For a start, summing any column (or row) gives you the *degree* of each vertex — i.e., how many others it is connected to. This is commonly denoted with the letter *k*.

Likewise, summing the degrees of every vertex and dividing by two gives you *L*, the number of edges (or ‘links’) in the network. The number of rows/columns gives us *N*, the number of vertices (or ‘nodes’) in the network.

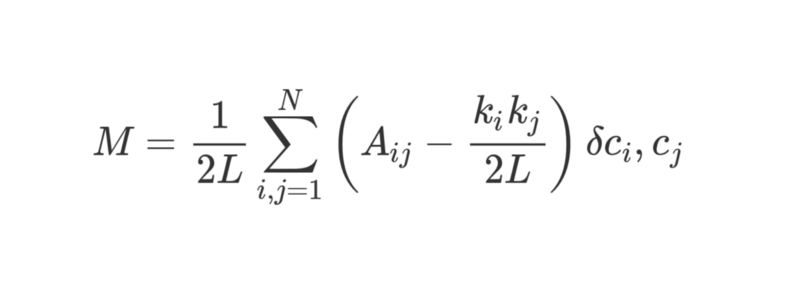
Knowing just *k*, *L, N* and the value of each cell in the adjacency matrix *A* lets us calculate the [*modularity*](https://en.wikipedia.org/wiki/Modularity_(networks))ofany given clustering of the network.

Say we’ve clustered the network into a number of communities. We can use the modularity score to assess the ‘quality’ of this clustering.

A higher score will show we’ve split the network into ‘accurate’ communities, whereas a low score suggests our clusters are more random than insightful. The image below illustrates this.

Modularity serves as a measure of the ‘quality’ of a partition.

Modularity can be calculated using the formula below:



That’s a fair amount of math, but we can break it down bit by bit and it’ll make more sense.

*M* is of course what we’re calculating — modularity.

1/2*L* tells us to divide everything that follows by 2*L*, i.e., twice the number of edges in the network. So far, so good.

The **Σ** symbol tells us we’re summing up everything to the right, and lets us iterate over every row and column in the adjacency matrix *A*.

For those unfamiliar with sum notation, the *i, j = 1* and the *N* work much like nested for-loops in programming. In Python, you’d write it as follows:

sum = 0

for i in range(1,N):

for j in range(1,N):

ans = #stuff with i and j as indices

sum += ans

So what is #stuff with i and j in more detail?

Well, the bit in brackets tells us to subtract ( *k\_i k\_j ) / 2L* from *A\_ij*.

*A\_ij* is simply the value in the adjacency matrix at row *i*, column *j*.

The values of *k\_i* and *k\_j* are the degrees of each vertex — found by adding up the entries in row *i* and column *j* respectively. Multiplying these together and dividing by 2*L* gives us the expected number of edges between vertices *i* and *j* if the network were randomly shuffled up.

Overall, the term in the brackets reveals the difference between the network’s real structure and the expected structure it would have if randomly reassembled.

Playing around with the values shows that it returns its highest value when *A\_ij* = 1, and ( *k\_i k\_j ) / 2L* is low. This means we see a higher value if there is an ‘unexpected’ edge between vertices *i* and *j.*

Finally, we multiply the bracketed term by whatever the last few symbols refer to.

The ?c*\_i,* c*\_j i*s the fancy-sounding but totally harmless Kronecker-delta function. Here it is, explained in Python:

def kroneckerDelta(ci, cj):

if ci == cj:

return 1

else:

return 0

kroneckerDelta("A","A")

#returns 1

kroneckerDelta("A","B")

#returns 0

Yes — it really is that simple. The Kronecker-delta function takes two arguments, and returns 1 if they are identical, otherwise, zero.

This means that if vertices *i* and *j* have been put in the same cluster, then ?c*\_i,* c*\_j = 1*. Otherwise, if they are in different clusters, the function returns zero.

As we are multiplying the bracketed term by this Kronecker-delta function, we find that for the nested sum **Σ**, the outcome is highest when there are lots of ‘unexpected’ edges connecting vertices assigned to the same cluster.

As such, modularity is a measure of how well-clustered the graph is into separate communities.

Dividing by *2L* bounds the upper value of modularity at 1. Modularity scores near to or below zero indicate the current clustering of the network is really no use. The higher the modularity, the better the clustering of the network into separate communities.

By maximising modularity, we can find the best way of clustering the network.

Notice that we have to pre-define how the graph is clustered to find out how ‘good’ that clustering actually is.

Unfortunately, employing brute force to try out every possible way of clustering the graph to find which has the highest modularity score would be computationally impossible beyond a very limited sample size.

[Combinatorics](https://en.wikipedia.org/wiki/Partition_of_a_set#Counting_partitions) tells us that for a network of just eight vertices, there are 4140 different ways of clustering them. A network twice the size would have over ten billion possible ways of clustering the vertices.

Doubling the network again (to a very modest 32 vertices) would give 128 septillion possible ways, and a network of eighty vertices would be cluster-able in more ways than there are [atoms in the observable universe](https://www.wolframalpha.com/input/?i=991267988808424794443839434655920239360814764000951599022939879419136287216681744888844&lk=1&rawformassumption=%22ClashPrefs%22+-%3E+%7B%22Math%22%7D).

Instead, we have to turn to a *heuristic* method that does a reasonably good job at estimating the clusters that will produce the highest modularity score, without trying out every single possibility.

This is an algorithm called *Fast-Greedy Modularity-Maximization,* and it’s somewhat analogous to the agglomerative hierarchical clustering algorithm describe above. Instead of merging according to distance, ‘Mod-Max’ merges communities according to changes in modularity.

Here’s how it goes:

**Begin** by initially assigning every vertex to its own community, and calculating the modularity of the whole network, *M*.

**Step 1** requires that for each community pair linked by at least a single edge, the algorithm calculates the resultant change in modularity Δ*M* if the two communities were merged into one.

**Step 2** then takes the pair of communities that produce the biggest increase in Δ*M,* which are then merged. Calculate the new modularity *M* for this clustering, and keep a record of it.

**Repeat** steps 1 and 2 — each time merging the pair of communities for which doing so produces the biggest gain in Δ*M,* then recording the new clustering pattern and its associated modularity score *M*.

**Stop** when all the vertices are grouped into one giant cluster. Now the algorithm checks the records it kept as it went along, and identifies the clustering pattern that returned the highest value of *M*. This is the returned community structure.

**Finer details**

Whew! That was computationally intensive, at least for us humans.

Graph theory is a rich source of computationally challenging, often NP-hard problems — yet it also has incredible potential to provide valuable insights into complex systems and datasets.

Just ask Larry Page, whose eponymous PageRank algorithm — which helped propel Google from start-up to basically world domination in less than a generation — was based entirely in graph theory.

Community detection is a major focus of current research in graph theory, and there are plenty of alternatives to Modularity-Maximization, which while useful, does have some drawbacks.

For a start, its agglomerative approach often sees small, well-defined communities swallowed up into larger ones. This is known as the *resolution limit* — the algorithm will not find communities below a certain size.

Another challenge is that rather than having one distinct, easy-to-reach global peak, the Mod-Max approach actually tends to produce a wide ‘plateau’ of many similar high modularity scores — making it somewhat difficult to truly identify the absolute maximum score.

Other algorithms use different ways to define and approach community detection.

*Edge-Betweenness* is a divisive algorithm, starting with all vertices grouped in one giant cluster. It proceeds to iteratively remove the least ‘important’ edges in the network, until all vertices are left isolated. This produces a hierarchical structure, with similar vertices closer together in the hierarchy.

Another algorithm is *Clique Percolation*, which takes into account possible overlap between graph communities.

Yet another set of algorithms are based on [random-walks](https://en.wikipedia.org/wiki/Random_walk) across the graph, and then there are [*spectral clustering*](https://en.wikipedia.org/wiki/Spectral_clustering) methods which start delving into the eigendecomposition of the adjacency matrix and other matrices derived therefrom. These ideas are used in feature extraction in, for example, areas such as computer vision.

It’d be well beyond the scope of this article to give each algorithm its own in-depth worked example. Suffice to say that this is an active area of research, providing powerful methods to make sense of data that even a generation ago would have been extremely difficult to process.

**Conclusion**

Hopefully this article has informed and inspired you to better understand how machines can make sense of data. The future is a rapidly changing place, and many of those changes will be driven by what technology becomes capable of in the next generation or two.

As outlined in the introduction, machine learning is an extraordinarily ambitious field of research, in which massively complex problems require solving in as accurate and as efficient a way possible. Tasks that come naturally to us humans require innovative solutions when taken on by machines.

There’s still plenty of progress to be made, and whoever contributes the next breakthrough idea will no doubt be generously rewarded. Maybe someone reading this article will be behind the next powerful algorithm?

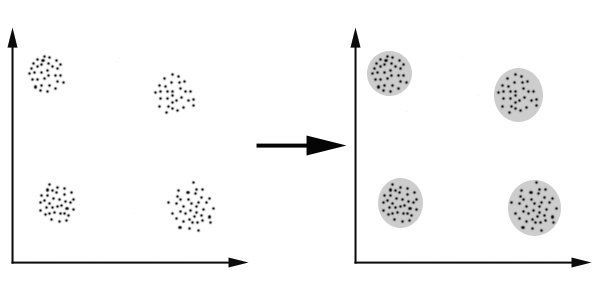
All great ideas have to start somewhere!

***A Tutorial on Clustering Algorithms***

Introduction | [K-means](https://matteucci.faculty.polimi.it/Clustering/tutorial_html/kmeans.html) | [Fuzzy C-means](https://matteucci.faculty.polimi.it/Clustering/tutorial_html/cmeans.html) | [Hierarchical](https://matteucci.faculty.polimi.it/Clustering/tutorial_html/hierarchical.html) | [Mixture of Gaussians](https://matteucci.faculty.polimi.it/Clustering/tutorial_html/mixture.html) | [Links](https://matteucci.faculty.polimi.it/Clustering/tutorial_html/links.html)

**Clustering: An Introduction**

*What is Clustering?*  
Clustering can be considered the most important *unsupervised learning* problem; so, as every other problem of this kind, it deals with finding a *structure* in a collection of unlabeled data.  
A loose definition of clustering could be “the process of organizing objects into groups whose members are similar in some way”.  
A *cluster* is therefore a collection of objects which are “similar” between them and are “dissimilar” to the objects belonging to other clusters.  
We can show this with a simple graphical example:



In this case we easily identify the 4 clusters into which the data can be divided; the similarity criterion is *distance*: two or more objects belong to the same cluster if they are “close” according to a given distance (in this case geometrical distance). This is called *distance-based clustering*.  
Another kind of clustering is *conceptual clustering*: two or more objects belong to the same cluster if this one defines a concept *common* to all that objects. In other words, objects are grouped according to their fit to descriptive concepts, not according to simple similarity measures.

*The Goals of Clustering*  
So, the goal of clustering is to determine the intrinsic grouping in a set of unlabeled data. But how to decide what constitutes a good clustering? It can be shown that there is no absolute “best” criterion which would be independent of the final aim of the clustering. Consequently, it is the user which must supply this criterion, in such a way that the result of the clustering will suit their needs.  
For instance, we could be interested in finding representatives for homogeneous groups (*data reduction*), in finding “natural clusters” and describe their unknown properties (*“natural” data types*), in finding useful and suitable groupings (*“useful” data classes*) or in finding unusual data objects (*outlier detection*).

*Possible Applications*  
Clustering algorithms can be applied in many fields, for instance:

* *Marketing*: finding groups of customers with similar behavior given a large database of customer data containing their properties and past buying records;
* *Biology*: classification of plants and animals given their features;
* *Libraries*: book ordering;
* *Insurance*: identifying groups of motor insurance policy holders with a high average claim cost; identifying frauds;
* *City-planning*: identifying groups of houses according to their house type, value and geographical location;
* *Earthquake studies*: clustering observed earthquake epicenters to identify dangerous zones;
* *WWW*: document classification; clustering weblog data to discover groups of similar access patterns.

*Requirements*  
The main requirements that a clustering algorithm should satisfy are:

* scalability;
* dealing with different types of attributes;
* discovering clusters with arbitrary shape;
* minimal requirements for domain knowledge to determine input parameters;
* ability to deal with noise and outliers;
* insensitivity to order of input records;
* high dimensionality;
* interpretability and usability.

*Problems*  
There are a number of problems with clustering. Among them:

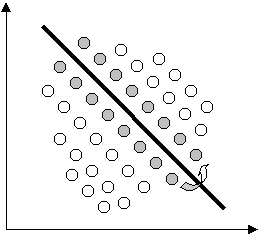
* current clustering techniques do not address all the requirements adequately (and concurrently);
* dealing with large number of dimensions and large number of data items can be problematic because of time complexity;
* the effectiveness of the method depends on the definition of “distance” (for distance-based clustering);
* if an *obvious* distance measure doesn’t exist we must “define” it, which is not always easy, especially in multi-dimensional spaces;
* the result of the clustering algorithm (that in many cases can be arbitrary itself) can be interpreted in different ways.

**Clustering Algorithms**

*Classification*  
Clustering algorithms may be classified as listed below:

* Exclusive Clustering
* Overlapping Clustering
* Hierarchical Clustering
* Probabilistic Clustering

In the first case data are grouped in an exclusive way, so that if a certain datum belongs to a definite cluster then it could not be included in another cluster. A simple example of that is shown in the figure below, where the separation of points is achieved by a straight line on a bi-dimensional plane.  
On the contrary the second type, the overlapping clustering, uses fuzzy sets to cluster data, so that each point may belong to two or more clusters with different degrees of membership. In this case, data will be associated to an appropriate membership value.



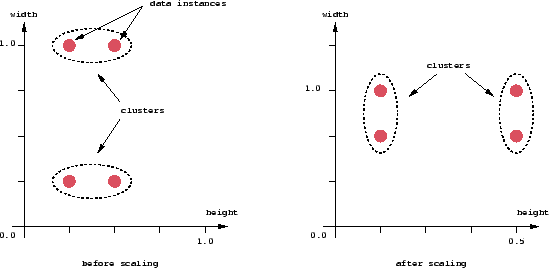
Instead, a hierarchical clustering algorithm is based on the union between the two nearest clusters. The beginning condition is realized by setting every datum as a cluster. After a few iterations it reaches the final clusters wanted.  
Finally, the last kind of clustering use a completely probabilistic approach.

In this tutorial we propose four of the most used clustering algorithms:

* K-means
* Fuzzy C-means
* Hierarchical clustering
* Mixture of Gaussians

Each of these algorithms belongs to one of the clustering types listed above. So that, [K-means](https://matteucci.faculty.polimi.it/Clustering/tutorial_html/kmeans.html) is an *exclusive clustering* algorithm, [Fuzzy C-means](https://matteucci.faculty.polimi.it/Clustering/tutorial_html/cmeans.html) is an *overlapping clustering* algorithm, [Hierarchical clustering](https://matteucci.faculty.polimi.it/Clustering/tutorial_html/hierarchical.html) is obvious and lastly [Mixture of Gaussian](https://matteucci.faculty.polimi.it/Clustering/tutorial_html/mixture.html) is a *probabilistic clustering* algorithm. We will discuss about each clustering method in the following paragraphs.

*Distance Measure*  
An important component of a clustering algorithm is the distance measure between data points. If the components of the data instance vectors are all in the same physical units then it is possible that the simple Euclidean distance metric is sufficient to successfully group similar data instances. However, even in this case the Euclidean distance can sometimes be misleading. Figure shown below illustrates this with an example of the width and height measurements of an object. Despite both measurements being taken in the same physical units, an informed decision has to be made as to the relative scaling. As the figure shows, different scalings can lead to different clusterings.



Notice however that this is not only a graphic issue: the problem arises from the mathematical formula used to combine the distances between the single components of the data feature vectors into a unique distance measure that can be used for clustering purposes: different formulas leads to different clusterings.  
Again, domain knowledge must be used to guide the formulation of a suitable distance measure for each particular application.

**Minkowski Metric**For higher dimensional data, a popular measure is the Minkowski metric,



where *d* is the dimensionality of the data. The *Euclidean* distance is a special case where *p*=2, while *Manhattan* metric has *p*=1. However, there are no general theoretical guidelines for selecting a measure for any given application.

It is often the case that the components of the data feature vectors are not immediately comparable. It can be that the components are not continuous variables, like length, but nominal categories, such as the days of the week. In these cases again, domain knowledge must be used to formulate an appropriate measure.

*Bibliography*

* Tariq Rashid: “Clustering”  
  <http://www.cs.bris.ac.uk/home/tr1690/documentation/fuzzy_clustering_initial_report/node11.html>
* Osmar R. Zaïane: “Principles of Knowledge Discovery in Databases - Chapter 8: Data Clustering”  
  [http://www.cs.ualberta.ca/~zaiane/courses/cmput690/slides/Chapter8/index.html](http://www.cs.ualberta.ca/%7Ezaiane/courses/cmput690/slides/Chapter8/index.html)
* Pier Luca Lanzi: “Ingegneria della Conoscenza e Sistemi Esperti – Lezione 2: Apprendimento non supervisionato”  
  <http://www.elet.polimi.it/upload/lanzi/corsi/icse/2002/Lezione%202%20-%20Apprendimento%20non%20supervisionato.pdf>

***A Tutorial on Clustering Algorithms***

[Introduction](https://matteucci.faculty.polimi.it/Clustering/tutorial_html/index.html) | K-means | [Fuzzy C-means](https://matteucci.faculty.polimi.it/Clustering/tutorial_html/cmeans.html) | [Hierarchical](https://matteucci.faculty.polimi.it/Clustering/tutorial_html/hierarchical.html) | [Mixture of Gaussians](https://matteucci.faculty.polimi.it/Clustering/tutorial_html/mixture.html) | [Links](https://matteucci.faculty.polimi.it/Clustering/tutorial_html/links.html)

**K-Means Clustering**

*The Algorithm*  
K-means ([MacQueen, 1967](https://matteucci.faculty.polimi.it/Clustering/tutorial_html/kmeans.html#macqueen)) is one of the simplest unsupervised learning algorithms that solve the well known clustering problem. The procedure follows a simple and easy way to classify a given data set through a certain number of clusters (assume k clusters) fixed a priori. The main idea is to define k centroids, one for each cluster. These centroids shoud be placed in a cunning way because of different location causes different result. So, the better choice is to place them as much as possible far away from each other. The next step is to take each point belonging to a given data set and associate it to the nearest centroid. When no point is pending, the first step is completed and an early groupage is done. At this point we need to re-calculate k new centroids as barycenters of the clusters resulting from the previous step. After we have these k new centroids, a new binding has to be done between the same data set points and the nearest new centroid. A loop has been generated. As a result of this loop we may notice that the k centroids change their location step by step until no more changes are done. In other words centroids do not move any more.  
Finally, this algorithm aims at minimizing an *objective function*, in this case a squared error function. The objective function

,

where is a chosen distance measure between a data point and the cluster centre , is an indicator of the distance of the *n* data points from their respective cluster centres.

The algorithm is composed of the following steps:

|  |
| --- |
| 1. *Place K points into the space represented by the objects that are being clustered. These points represent initial group centroids.* 2. *Assign each object to the group that has the closest centroid.* 3. *When all objects have been assigned, recalculate the positions of the K centroids.* 4. *Repeat Steps 2 and 3 until the centroids no longer move. This produces a separation of the objects into groups from which the metric to be minimized can be calculated.* |

Although it can be proved that the procedure will always terminate, the k-means algorithm does not necessarily find the most optimal configuration, corresponding to the global objective function minimum. The algorithm is also significantly sensitive to the initial randomly selected cluster centres. The k-means algorithm can be run multiple times to reduce this effect.

K-means is a simple algorithm that has been adapted to many problem domains. As we are going to see, it is a good candidate for extension to work with fuzzy feature vectors.

*An example*  
Suppose that we have n sample feature vectors **x**1, **x**2, ..., **x**n all from the same class, and we know that they fall into k compact clusters, k < n. Let **m**i be the mean of the vectors in cluster i. If the clusters are well separated, we can use a minimum-distance classifier to separate them. That is, we can say that **x** is in cluster i if || **x** - **m**i || is the minimum of all the k distances. This suggests the following procedure for finding the k means:

* Make initial guesses for the means **m**1, **m**2, ..., **m**k
* Until there are no changes in any mean
  + Use the estimated means to classify the samples into clusters
  + For i from 1 to k
    - Replace **m**i with the mean of all of the samples for cluster i
  + end\_for
* end\_until

Here is an example showing how the means **m**1 and **m**2 move into the centers of two clusters.

A picture containing black, darkness

Description automatically generated

**Remarks**  
This is a simple version of the k-means procedure. It can be viewed as a greedy algorithm for partitioning the n samples into k clusters so as to minimize the sum of the squared distances to the cluster centers. It does have some weaknesses:

* The way to initialize the means was not specified. One popular way to start is to randomly choose k of the samples.
* The results produced depend on the initial values for the means, and it frequently happens that suboptimal partitions are found. The standard solution is to try a number of different starting points.
* It can happen that the set of samples closest to **m**i is empty, so that **m**i cannot be updated. This is an annoyance that must be handled in an implementation, but that we shall ignore.
* The results depend on the metric used to measure || **x** - **m**i ||. A popular solution is to normalize each variable by its standard deviation, though this is not always desirable.
* The results depend on the value of k.

This last problem is particularly troublesome, since we often have no way of knowing how many clusters exist. In the example shown above, the same algorithm applied to the same data produces the following 3-means clustering. Is it better or worse than the 2-means clustering?

A picture containing black, darkness

Description automatically generated

Unfortunately there is no general theoretical solution to find the optimal number of clusters for any given data set. A simple approach is to compare the results of multiple runs with different k classes and choose the best one according to a given criterion (for instance the Schwarz Criterion - see [Moore's slides](https://matteucci.faculty.polimi.it/Clustering/tutorial_html/kmeans.html#moore)), but we need to be careful because increasing k results in smaller error function values by definition, but also an increasing risk of overfitting.

*Bibliography*

* J. B. MacQueen (1967): "Some Methods for classification and Analysis of Multivariate Observations, *Proceedings of 5-th Berkeley Symposium on Mathematical Statistics and Probability"*, Berkeley, University of California Press, 1:281-297
* Andrew Moore: “K-means and Hierarchical Clustering - Tutorial Slides”  
  [http://www-2.cs.cmu.edu/~awm/tutorials/kmeans.html](http://www-2.cs.cmu.edu/%7Eawm/tutorials/kmeans.html)
* Brian T. Luke: “K-Means Clustering”  
  [http://fconyx.ncifcrf.gov/~lukeb/kmeans.html](http://fconyx.ncifcrf.gov/%7Elukeb/kmeans.html)
* Tariq Rashid: “Clustering”  
  <http://www.cs.bris.ac.uk/home/tr1690/documentation/fuzzy_clustering_initial_report/node11.html>
* Hans-Joachim Mucha and Hizir Sofyan: “Nonhierarchical Clustering”  
  [http://www.quantlet.com/mdstat/scripts/xag/html/xaghtmlframe149.ht](http://www.quantlet.com/mdstat/scripts/xag/html/xaghtmlframe149.html)

Clustering Questions

Which instructions correctly capture the K-means algorithm, as described in the lecture?

A. Choose k random data points Assign cluster centers to those data points Assign each cluster to the nearest data point Update the cluster centers

B. Choose k random positions Assign cluster centers to these positions Compute distances of all data points to the cluster centers Assign each data point to the nearest cluster center Update cluster centers (as the mean of those)

C. Choose k random positions Compute distances of the data points to those positions Assign cluster centers to those positions Assign each data point to the nearest cluster center Update cluster centers (as the mean of those)

D. Update the cluster centers Assign cluster centers to those data points Assign each cluster to the nearest data point Choose k random data points

The K-Means algorithm isn’t guaranteed to converge

TRUE

FALSE

The output of the K-Means algorithm doesn’t depend on initialization

TRUE

FALSE

Which of the following counts as a way to deal with the K-Means algorithm’s sensitivity to outliers?

a. Using the mean instead of the median for updates

b. Using the mode instead of the median for updates

c. Using the median instead of the mean for updates

d. Using the mean instead of the mode for updates

What happens if we set k to 1?

a. The algorithm crashes

b.The algorithm becomes more efficient

c. Only one cluster center results

d. Infinitely-many cluster centers result

We should choose a value for k such that:

a. Decreasing k doesn’t model the data any better

b. Increasing k doesn’t model the data any better

c. Decreasing k would model the data slightly better

d. Increasing k would model the data slightly better

What happens if k is set too high?

a. We explain too little of the variance

b. We explain too little of the error

c. We explain too much of the variance

d. We explain too much of the error

Which of the following is NOT part of any initialization method for the K-Means algorithm?

a. Starting several times

b. Choosing random positions

c. Using random data points as centers

d. Picking a training and test set

Non-mathematically, what is the Within Groups Sum of Squares (WGSS)?

a. The distance of all the points to their closest centers

b. The distance of all the centers to data point furthest away from them

c. The sum of the squares of the distances between each data point and every other data point

d. None of the above

What is the purpose of the ‘Knee’ or ‘Elbow’ method?

a. To find out how long the K-Means algorithm will take to execute

b. To find out whether the problem is indeed one that could be solved by K-Means

c. To help us pick an initial value for ‘K’

d. None of the above

What will be the value of the Within Groups Sum of Squares (WGSS) if K is initialized to exactly the number of data points? Will there be a value?

a. 1

b. 0

c. NA

d. The exact number of data points

Which of the following is a good method by which to check whether the algorithm has converged?

a. Check how many cluster centers have moved; specify a value for ε; if they’ve moved less than ε then the algorithm has converged; else it hasn’t.

b. Check how many cluster centers have moved; specify a value for ε; if they’ve moved more than ε then the algorithm has converged; else it hasn’t.

c. Look at the number of clusters. If there are just as many clusters as there are data points, then the algorithm has converged.

d. Check how many cluster centers have moved; specify a value for ε; the algorithm has converged if and only if they’ve moved by exactly ε; else it hasn’t.

Which of the following claims is true?

a. Cross-validation is not used when we apply clustering to unseen data

b. Cross-validation is used when we apply clustering to unseen data

c. Cross-validation never involves partitioning

d. All of the above claims are false

The following is a correct account of the method of cross-validation: I. Partition data into n folds II. Cluster on (n-1) folds III. Compute sum of squared distances to centroids for validation set

TRUE

FALSE

What is the main motivation behind Mean-Shift Clustering?

a. Making clustering more accurate than K-Means

b. Making clustering faster than K-Means

c. Making clustering a form of supervised learning

d. Avoiding the need to choose K

The following is the correct way to execute Mean-Shift Clustering: i. Put a window around each data point ii. Compute mean of points in the frame iii. Repeat until convergence iv. Shift the window to the mean

TRUE

FALSE

Why is Mean-Shift Clustering computationally hard?

a. We have to specify window size in advance and keep it constant for all executions

b. We have to determine a cluster for each point

c. We need to specify K in advance and use the same value each time we execute it

d. None of the above

What is the main drawback of Mean-Shift Clustering?

a. It’s a more difficult algorithm to write than K-Means

b. We still have to initialize K

c. We still have to pick values for the window sizes

d. None of the above

*This notebook was put together by* [*Jake Vanderplas*](http://www.vanderplas.com) *for PyCon 2014. Source and license info is on* [*GitHub*](https://github.com/jakevdp/sklearn_pycon2014/)*.*

# Unsupervised Learning In-depth: PCA and K-Means

Here we'll briefly go into a bit of depth on some important unsupervised learning techniques: **Principal Component Analysis (PCA)** and **K-Means**.

By the end of this section you should

* be able to describe how PCA reduces dimensionality
* be able to describe how K Means defines clusters
* see how these can be applied to several interesting problems

%matplotlib inline

import matplotlib.pyplot as plt

import numpy as np

## Principal Component Analysis

Principal Component Analysis is a very powerful unsupervised method for dimensionality reduction in data. It's easiest to visualize by looking at a two-dimensional dataset:

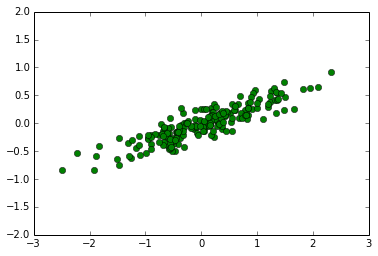
np.random.seed(1)

X = np.dot(np.random.random(size=(2, 2)), np.random.normal(size=(2, 200))).T

plt.plot(X[:, 0], X[:, 1], 'og')

plt.axis('equal')

(-3.0, 3.0, -1.0, 1.0)



We can see that there is a definite trend in the data. What PCA seeks to do is to find the **Principal Axes** in the data, and explain how important those axes are in describing the data distribution:

from sklearn.decomposition import PCA

pca = PCA(n\_components=2)

pca.fit(X)

print(pca.explained\_variance\_)

print(pca.components\_)

[ 0.75871884 0.01838551]

[[ 0.94446029 0.32862557]

[-0.32862557 0.94446029]]

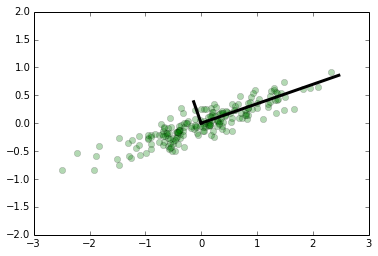
plt.plot(X[:, 0], X[:, 1], 'og', alpha=0.3)

plt.axis('equal')

for length, vector in zip(pca.explained\_variance\_, pca.components\_):

v = vector \* 3 \* np.sqrt(length)

plt.plot([0, v[0]], [0, v[1]], '-k', lw=3)



Notice that one direction is very important, while the other direction is not. This shows us that the second principal component could be **completely ignored** without much loss of information! Let's see what our data look like if we only keep 95% of the variance:

clf = PCA(0.95)

X\_trans = clf.fit\_transform(X)

print(X.shape)

print(X\_trans.shape)

(200, 2)

(200, 1)

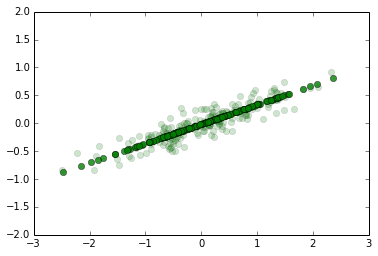
By specifying that we want to throw away 5% of the variance, the data is now compressed by a factor of 50%! Let's see what the data look like after this compression:

X\_new = clf.inverse\_transform(X\_trans)

plt.plot(X[:, 0], X[:, 1], 'og', alpha=0.2)

plt.plot(X\_new[:, 0], X\_new[:, 1], 'og', alpha=0.8)

plt.axis('equal');



The light points are the original data, while the dark points are the projected version. We see that after truncating 5% of the variance of this dataset and then reprojecting it, the "most important" features of the data are maintained, and we've compressed the data by 50%!

### Application of PCA to Digits

This might seem a bit strange in two dimensions, but the projection and dimensionality reduction can be extremely useful when visualizing high-dimensional data. Let's take a quick look at the application of PCA to the digits data we looked at before:

from sklearn.datasets import load\_digits

digits = load\_digits()

X = digits.data

y = digits.target

pca = PCA(2) # project from 64 to 2 dimensions

Xproj = pca.fit\_transform(X)

print(X.shape)

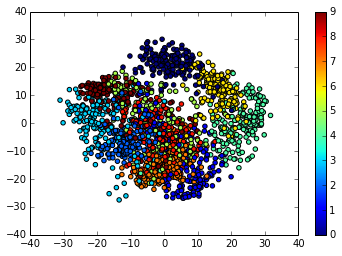
print(Xproj.shape)

(1797, 64)

(1797, 2)

plt.scatter(Xproj[:, 0], Xproj[:, 1], c=y)

plt.colorbar();



This gives us an idea of the relationship between the digits. Essentially, we have found the optimal rotation in 64-dimensional space that allows us to see the layout of the digits, **without reference** to the labels.

But how much information have we thrown away? We can figure this out by looking at the variance:

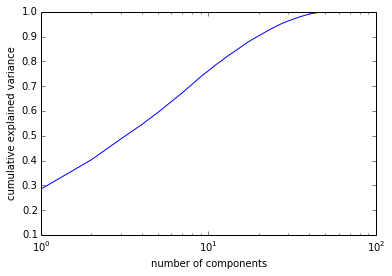
pca = PCA(64).fit(X)

plt.semilogx(np.cumsum(pca.explained\_variance\_ratio\_))

plt.xlabel('number of components')

plt.ylabel('cumulative explained variance')

<matplotlib.text.Text at 0x10809a990>



Here we see that our two-dimensional projection loses a lot of information (as measured by the explained variance) and that we'd need about 20 components to retain 90% of the variance. Looking at this plot for a high-dimensional dataset can help you understand the level of redundancy present in multiple observations.

Note that scikit-learn contains many other unsupervised dimensionality reduction routines: some you might wish to try are Other dimensionality reduction techniques which are useful to know about:

* [sklearn.decomposition.PCA](http://scikit-learn.org/0.13/modules/generated/sklearn.decomposition.PCA.html): Principal Component Analysis
* [sklearn.decomposition.RandomizedPCA](http://scikit-learn.org/0.13/modules/generated/sklearn.decomposition.RandomizedPCA.html): fast non-exact PCA implementation based on a randomized algorithm
* [sklearn.decomposition.SparsePCA](http://scikit-learn.org/0.13/modules/generated/sklearn.decomposition.SparsePCA.html): PCA variant including L1 penalty for sparsity
* [sklearn.decomposition.FastICA](http://scikit-learn.org/0.13/modules/generated/sklearn.decomposition.FastICA.html): Independent Component Analysis
* [sklearn.decomposition.NMF](http://scikit-learn.org/0.13/modules/generated/sklearn.decomposition.NMF.html): non-negative matrix factorization
* [sklearn.manifold.LocallyLinearEmbedding](http://scikit-learn.org/0.13/modules/generated/sklearn.manifold.LocallyLinearEmbedding.html): nonlinear manifold learning technique based on local neighborhood geometry
* [sklearn.manifold.IsoMap](http://scikit-learn.org/0.13/modules/generated/sklearn.manifold.Isomap.html): nonlinear manifold learning technique based on a sparse graph algorithm

## K Means

K Means is an algorithm for **unsupervised clustering**: that is, finding clusters in data based on the data attributes alone (not the labels).

K Means is a relatively easy-to-understand algorithm. It searches for cluster centers which are the mean of the points within them, such that every point is closest to the cluster center it is assigned to.

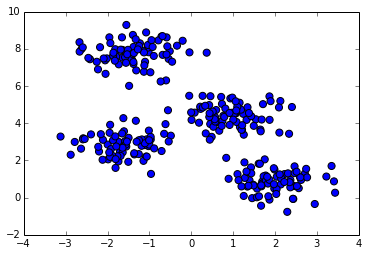
Let's look at how KMeans operates on the simple clusters we looked at previously. To emphasize that this is unsupervised, we'll not plot the colors of the clusters:

from sklearn.datasets.samples\_generator import make\_blobs

X, y = make\_blobs(n\_samples=300, centers=4,

random\_state=0, cluster\_std=0.60)

plt.scatter(X[:, 0], X[:, 1], s=50);



By eye, it is relatively easy to pick out the four clusters. If you were to perform an exhaustive search for the different segmentations of the data, however, the search space would be exponential in the number of points. Fortunately, there is a well-known Expectation Maximization (EM) procedure which scikit-learn implements, so that KMeans can be solved relatively quickly.

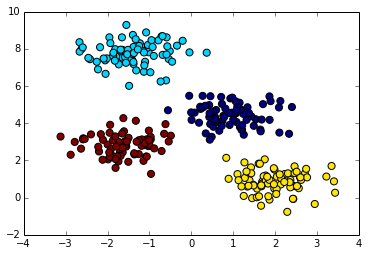
from sklearn.cluster import KMeans

est = KMeans(4) # 4 clusters

est.fit(X)

y\_kmeans = est.predict(X)

plt.scatter(X[:, 0], X[:, 1], c=y\_kmeans, s=50);



The algorithm identifies the four clusters of points in a manner very similar to what we would do by eye!

### Application of KMeans to Digits

For a closer-to-real-world example, let's again take a look at the digits data. Here we'll use KMeans to automatically cluster the data in 64 dimensions, and then look at the cluster centers to see what the algorithm has found.

est = KMeans(n\_clusters=10)

clusters = est.fit\_predict(digits.data)

est.cluster\_centers\_.shape

(10, 64)

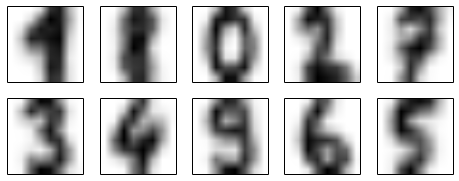
We see ten clusters in 64 dimensions. Let's visualize each of these cluster centers to see what they represent:

fig = plt.figure(figsize=(8, 3))

for i in range(10):

ax = fig.add\_subplot(2, 5, 1 + i, xticks=[], yticks=[])

ax.imshow(est.cluster\_centers\_[i].reshape((8, 8)), cmap=plt.cm.binary)



We see that even without the labels, KMeans is able to find clusters whose means are recognizable digits (with apologies to the number 8).

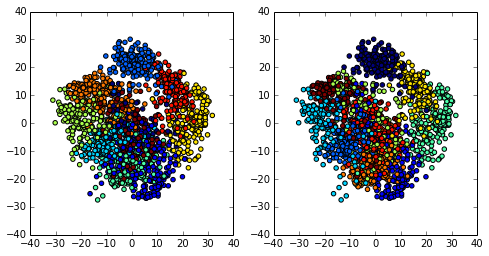
For good measure, let's use our PCA visualization and look at the true cluster labels and K-means cluster labels:

X = PCA(2).fit\_transform(digits.data)

fig, ax = plt.subplots(1, 2, figsize=(8, 4))

ax[0].scatter(X[:, 0], X[:, 1], c=clusters)

ax[1].scatter(X[:, 0], X[:, 1], c=digits.target);



Though the colors are permuted, we see that in general (at least in a straightforward by-eye comparison) the KMeans clusters tend to reflect the true clustering.

## Exercise: KMeans for Color Compression

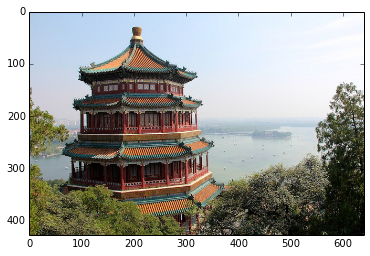
One interesting application of clustering is in color image compression. For example, imagine you have an image with millions of colors. In most images, a large number of the colors will be unused, and conversely a large number of pixels will have similar or identical colors.

Scikit-learn has a number of images that you can play with, accessed through the datasets module. For example:

from sklearn.datasets import load\_sample\_image

china = load\_sample\_image("china.jpg")

plt.imshow(china);



The image itself is stored in a 3-dimensional array, of size (height, width, RGB):

china.shape

(427, 640, 3)

We can envision this image as a cloud of points in a 3-dimensional color space. We'll rescale the colors so they lie between 0 and 1, then reshape the array to be a typical scikit-learn input:

X = (china / 255.0).reshape(-1, 3)

print(X.shape)

(273280, 3)

We now have 273,280 points in 3 dimensions. Your task is to use KMeans to compress the 2563

colors into a smaller number (say, 64 colors). Basically, you want to find *Ncolor*

clusters in the data, and create a new image where the true input color is replaced by the color of the closest cluster.

Your goal is to fill-in the following function:

def compress\_image(image, n\_colors):

"""Compress an image

Parameters

==========

image : numpy array

array of shape (height, width, 3) with values between 0 and 1

n\_colors : integer

the number of colors in the final compressed image

(i.e. the number of KMeans clusters to fit).

Returns

=======

new\_image : numpy array

array representing the new image, compressed via KMeans clustering.

It has the same shape as the input image, but contains only

``n\_colors`` distinct colors.

"""

X = (image / 255.0).reshape(-1, 3)

new\_image = image.copy()

#------------

# Your KMeans code goes here!

#------------

# if you convert back to integer, make sure it's the correct type!

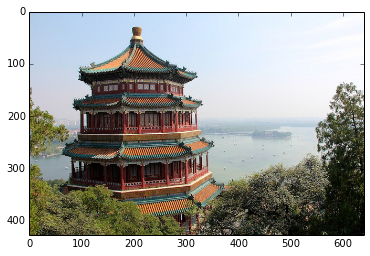
# i.e. new\_image = (255 \* new\_image).astype(np.uint8)

return new\_image

# create and plot the new image

new\_image = compress\_image(china, 64)

plt.imshow(new\_image);



How does the color fidelity compare to the original?

Hint: because the calculation takes a long time, you may wish to test it on only a subset of the image. For example,

new\_image = compress\_image(china[::5, ::5], 64)

Give this a good try! If you would like to load the solution, uncomment the following command:

# %load solutions/05\_color\_compression.py

## 1. Unsupervised Learning

00:00 - 00:09

Hi! My name is Ben Wilson and I'm a Data Scientist and mathematician. We're here to learn about unsupervised learning in Python.

## 2. Unsupervised learning

00:09 - 00:33

Unsupervised learning is a class of machine learning techniques for discovering patterns in data. For instance, finding the natural "clusters" of customers based on their purchase histories, or searching for patterns and correlations among these purchases, and using these patterns to express the data in a compressed form. These are examples of unsupervised learning techniques called "clustering" and "dimension reduction".

## 3. Supervised vs unsupervised learning

00:33 - 01:13

Unsupervised learning is defined in opposition to supervised learning. An example of supervised learning is using the measurements of tumors to classify them as benign or cancerous. In this case, the pattern discovery is guided, or "supervised", so that the patterns are as useful as possible for predicting the label: benign or cancerous. Unsupervised learning, in contrast, is learning without labels. It is pure pattern discovery, unguided by a prediction task. You'll start by learning about clustering. But before we begin, let's introduce a dataset and fix some terminology.

## 4. Iris dataset

01:13 - 01:30

The iris dataset consists of the measurements of many iris plants of three different species. There are four measurements: petal length, petal width, sepal length and sepal width. These are the features of the dataset.

1. 1 https://scikit-learn.org/stable/modules/generated/sklearn.datasets.load\_iris.html

## 5. Arrays, features & samples

01:30 - 01:50

Throughout this course, datasets like this will be written as two-dimensional numpy arrays. The columns of the array will correspond to the features. The measurements for individual plants are the samples of the dataset. These correspond to rows of the array.

## 6. Iris data is 4-dimensional

01:50 - 02:07

The samples of the iris dataset have four measurements, and so correspond to points in a four-dimensional space. This is the dimension of the dataset. We can't visualize four dimensions directly, but using unsupervised learning techniques we can still gain insight.

## 7. k-means clustering

02:07 - 02:27

In this chapter, we'll cluster these samples using k-means clustering. k-means finds a specified number of clusters in the samples. It's implemented in the scikit-learn or "sklearn" library. Let's see kmeans in action on some samples from the iris dataset.

## 8. k-means clustering with scikit-learn

02:27 - 03:13

The iris samples are represented as an array. To start, import kmeans from scikit-learn. Then create a kmeans model, specifying the number of clusters you want to find. Let's specify 3 clusters, since there are three species of iris. Now call the fit method of the model, passing the array of samples. This fits the model to the data, by locating and remembering the regions where the different clusters occur. Then we can use the predict method of the model on these same samples. This returns a cluster label for each sample, indicating to which cluster a sample belongs. Let's assign the result to labels, and print it out.

## 9. Cluster labels for new samples

03:13 - 03:35

If someone comes along with some new iris samples, k-means can determine to which clusters they belong without starting over. k-means does this by remembering the mean (or average) of the samples in each cluster. These are called the "centroids". New samples are assigned to the cluster whose centroid is closest.

## 10. Cluster labels for new samples

03:35 - 03:50

Suppose you've got an array of new samples. To assign the new samples to the existing clusters, pass the array of new samples to the predict method of the kmeans model. This returns the cluster labels of the new samples.

## 11. Scatter plots

03:50 - 04:17

In the next video, you'll learn how to evaluate the quality of your clustering. But for now, let's visualize our clustering of the iris samples using scatter plots. Here is a scatter plot of the sepal length vs petal length of the iris samples. Each point represents an iris sample, and is colored according to the cluster of the sample. To create a scatter plot like this, use PyPlot.

## 12. Scatter plots

04:17 - 04:49

Firstly, import PyPlot. It is conventionally imported as plt. Now get the x- and y- co-ordinates of each sample. Sepal length is in the 0th column of the array, while petal length is in the 2nd column. Now call the plt dot scatter function, passing the x- and y- co-ordinates and specifying c=labels to color by cluster label. When you are ready to show your plot, call plt dot show.

## 13. Let's practice!

04:49 - 04:58

It's time to take your first steps in unsupervised learning. Have fun!

Daily XP50

## Exercise

# How many clusters?

You are given an array points of size 300x2, where each row gives the (x, y) co-ordinates of a point on a map. Make a scatter plot of these points, and use the scatter plot to guess how many clusters there are.

matplotlib.pyplot has already been imported as plt. In the IPython Shell:

* Create an array called xs that contains the values of points[:,0] - that is, column 0 of points.
* Create an array called ys that contains the values of points[:,1] - that is, column 1 of points.
* Make a scatter plot by passing xs and ys to the plt.scatter() function.
* Call the plt.show() function to show your plot.

How many clusters do you see?

### Possible answers

2

3

300

Correct! The scatter plot suggests that there are 3 distinct clusters.

import matplotlib.pyplot as plt xs = points[:,0] ys = points[:,1] plt.scatter(xs,ys) plt.show()

# Clustering 2D points

From the scatter plot of the previous exercise, you saw that the points seem to separate into 3 clusters. You'll now create a KMeans model to find 3 clusters, and fit it to the data points from the previous exercise. After the model has been fit, you'll obtain the cluster labels for some new points using the .predict() method.

You are given the array points from the previous exercise, and also an array new\_points.

## Instructions

100 XP

* Import KMeans from sklearn.cluster.
* Using KMeans(), create a KMeans instance called model to find 3 clusters. To specify the number of clusters, use the n\_clusters keyword argument.
* Use the .fit() method of model to fit the model to the array of points points.
* Use the .predict() method of model to predict the cluster labels of new\_points, assigning the result to labels.
* Hit submit to see the cluster labels of new\_points.
* # Import KMeans
* from sklearn.cluster import KMeans
* # Create a KMeans instance with 3 clusters: model
* model = KMeans(n\_clusters=3)
* # Fit model to points
* model.fit(points)
* # Determine the cluster labels of new\_points: labels
* labels = model.predict(new\_points)
* # Print cluster labels of new\_points
* print(labels)
* #import matplotlib.pyplot as plt
* #xs = points[:,0]
* #ys = points[:,1]
* #plt.scatter(xs,ys)
* #plt.show()

# Import KMeans

from sklearn.cluster import KMeans

# Create a KMeans instance with 3 clusters: model

model = KMeans(n\_clusters=3)

# Fit model to points

model.fit(points)

# Determine the cluster labels of new\_points: labels

labels = model.predict(new\_points)

# Print cluster labels of new\_points

print(labels)

#import matplotlib.pyplot as plt

#xs = points[:,0]

#ys = points[:,1]

#plt.scatter(xs,ys)

#plt.show()

[1 2 0 1 2 1 2 2 2 0 1 2 2 0 0 2 0 0 2 2 0 2 1 2 1 0 2 0 0 1 1 2 2 2 0 1 2

2 1 2 0 1 1 0 1 2 0 0 2 2 2 2 0 0 1 1 0 0 0 1 1 2 2 2 1 2 0 2 1 0 1 1 1 2

1 0 0 1 2 0 1 0 1 2 0 2 0 1 2 2 2 1 2 2 1 0 0 0 0 1 2 1 0 0 1 1 2 1 0 0 1

0 0 0 2 2 2 2 0 0 2 1 2 0 2 1 0 2 0 0 2 0 2 0 1 2 1 1 2 0 1 2 1 1 0 2 2 1

0 1 0 2 1 0 0 1 0 2 2 0 2 0 0 2 2 1 2 2 0 1 0 1 1 2 1 2 2 1 1 0 1 1 1 0 2

2 1 0 1 0 0 2 2 2 1 2 2 2 0 0 1 2 1 1 1 0 2 2 2 2 2 2 0 0 2 0 0 0 0 2 0 0

2 2 1 0 1 1 0 1 0 1 0 2 2 0 2 2 2 0 1 1 0 2 2 0 2 0 0 2 0 0 1 0 1 1 1 2 0

0 0 1 2 1 0 1 0 0 2 1 1 1 0 2 2 2 1 2 0 0 2 1 1 0 1 1 0 1 2 1 0 0 0 0 2 0

0 2 2 1]

Great work! You've successfully performed k-Means clustering and predicted the labels of new points. But it is not easy to inspect the clustering by just looking at the printed labels. A visualization would be far more useful. In the next exercise, you'll inspect your clustering with a scatter plot!

# Inspect your clustering

Let's now inspect the clustering you performed in the previous exercise!

A solution to the previous exercise has already run, so new\_points is an array of points and labels is the array of their cluster labels.

## Instructions

100 XP

* Import matplotlib.pyplot as plt.
* Assign column 0 of new\_points to xs, and column 1 of new\_points to ys.
* Make a scatter plot of xs and ys, specifying the c=labels keyword arguments to color the points by their cluster label. Also specify alpha=0.5.
* Compute the coordinates of the centroids using the .cluster\_centers\_ attribute of model.
* Assign column 0 of centroids to centroids\_x, and column 1 of centroids to centroids\_y.
* Make a scatter plot of centroids\_x and centroids\_y, using 'D' (a diamond) as a marker by specifying the marker parameter. Set the size of the markers to be 50 using s=50.

# Import pyplot

\_\_\_\_

# Assign the columns of new\_points: xs and ys

xs = \_\_\_\_

ys = \_\_\_\_

# Make a scatter plot of xs and ys, using labels to define the colors

\_\_\_\_

# Assign the cluster centers: centroids

centroids = \_\_\_\_

# Assign the columns of centroids: centroids\_x, centroids\_y

centroids\_x = centroids[:,0]

centroids\_y = centroids[:,1]

# Make a scatter plot of centroids\_x and centroids\_y

\_\_\_\_

plt.show()

# Import pyplot

import matplotlib.pyplot as plt

# Assign the columns of new\_points: xs and ys

xs = new\_points[:, 0]

ys = new\_points[:,1]

# Make a scatter plot of xs and ys, using labels to define the colors

plt.scatter(xs,ys, c=labels, alpha=0.5)

# Assign the cluster centers: centroids

centroids = model.cluster\_centers\_

# Assign the columns of centroids: centroids\_x, centroids\_y

centroids\_x = centroids[:,0]

centroids\_y = centroids[:,1]

# Make a scatter plot of centroids\_x and centroids\_y

plt.scatter(centroids\_x, centroids\_y, marker='D', s=50)

plt.show()

# Import pyplot import matplotlib.pyplot as plt # Assign the columns of new\_points: xs and ys xs = new\_points[:, 0] ys = new\_points[:,1] # Make a scatter plot of xs and ys, using labels to define the colors plt.scatter(xs,ys, c=labels, alpha=0.5) # Assign the cluster centers: centroids centroids = model.cluster\_centers\_ # Assign the columns of centroids: centroids\_x, centroids\_y centroids\_x = centroids[:,0] centroids\_y = centroids[:,1] # Make a scatter plot of centroids\_x and centroids\_y plt.scatter(centroids\_x, centroids\_y, marker='D', s=50) plt.show()

Fantastic! The clustering looks great! But how can you be sure that 3 clusters is the correct choice? In other words, how can you evaluate the quality of a clustering? Tune into the next video in which Ben will explain how to evaluate a clustering!

## 1. Evaluating a clustering

00:00 - 00:11

In the previous video, we used k-means to cluster the iris samples into three clusters. But how can we evaluate the quality of this clustering?

## 2. Evaluating a clustering

00:11 - 00:32

A direct approach is to compare the clusters with the iris species. You'll learn about this first, before considering the problem of how to measure the quality of a clustering in a way that doesn't require our samples to come pre-grouped into species. This measure of quality can then be used to make an informed choice about the number of clusters to look for.

## 3. Iris: clusters vs species

00:32 - 01:18

Firstly, let's check whether the 3 clusters of iris samples have any correspondence to the iris species. The correspondence is described by this table. There is one column for each of the three species of iris: setosa, versicolor and virginica, and one row for each of the three cluster labels: 0, 1 and 2. The table shows the number of samples that have each possible cluster label/species combination. For example, we see that cluster 1 corresponds perfectly with the species setosa. On the other hand, while cluster 0 contains mainly virginica samples, there are also some virginica samples in cluster 2.

## 4. Cross tabulation with pandas

01:18 - 01:32

Tables like these are called "cross-tabulations". To construct one, we are going to use the pandas library. Let's assume the species of each sample is given as a list of strings.

## 5. Aligning labels and species

01:32 - 01:48

Import pandas, and then create a two-column DataFrame, where the first column is cluster labels and the second column is the iris species, so that each row gives the cluster label and species of a single sample.

## 6. Crosstab of labels and species

01:48 - 02:12

Now use the pandas crosstab function to build the cross tabulation, passing the two columns of the DataFrame. Cross tabulations like these provide great insights into which sort of samples are in which cluster. But in most datasets, the samples are not labelled by species. How can the quality of a clustering be evaluated in these cases?

## 7. Measuring clustering quality

02:12 - 02:30

We need a way to measure the quality of a clustering that uses only the clusters and the samples themselves. A good clustering has tight clusters, meaning that the samples in each cluster are bunched together, not spread out.

## 8. Inertia measures clustering quality

02:30 - 03:06

How spread out the samples within each cluster are can be measured by the "inertia". Intuitively, inertia measures how far samples are from their centroids. You can find the precise definition in the scikit-learn documentation. We want clusters that are not spread out, so lower values of the inertia are better. The inertia of a kmeans model is measured automatically when any of the fit methods are called, and is available afterwards as the inertia attribute. In fact, kmeans aims to place the clusters in a way that minimizes the inertia.

## 9. The number of clusters

03:06 - 03:26

Here is a plot of the inertia values of clusterings of the iris dataset with different numbers of clusters. Our kmeans model with 3 clusters has relatively low inertia, which is great. But notice that the inertia continues to decrease slowly. So what's the best number of clusters to choose?

## 10. How many clusters to choose?

03:26 - 03:53

Ultimately, this is a trade-off. A good clustering has tight clusters (meaning low inertia). But it also doesn't have too many clusters. A good rule of thumb is to choose an elbow in the inertia plot, that is, a point where the inertia begins to decrease more slowly. For example, by this criterion, 3 is a good number of clusters for the iris dataset.

## 11. Let's practice!

03:53 - 04:10

In this video, you've learned ways to evaluate the quality of a clustering. In the next video, you'll learn to use feature scaling to make your clusterings even better. For now, let's practice!

1. [SPRINGbOARD](https://github.com/Amit-DataScience/SPRINGbOARD/tree/master)

/

**Euclidean\_and\_Manhattan\_Distances\_Case\_Study (1).ipynb**

## Euclidean and Manhattan Distance Calculations

While working on this quick case study, you'll see examples and comparisons of distance measures. Specifically, you'll visually compare the Euclidean distance to the Manhattan distance measures. Distance measures have a multitude of uses in data science and are the foundations of many algorithms you'll be using, including Prinical Components Analysis.

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

import matplotlib.cm as cm

%matplotlib inline

plt.style.use('ggplot')

# Load Course Numerical Dataset

df = pd.read\_csv('distance\_dataset.csv',index\_col=0)

df.head()

|  | **X** | **Y** | **Z** | **ClusterID** |
| --- | --- | --- | --- | --- |
| **0** | 5.135779 | 4.167542 | 5.787635 | 4 |
| **1** | 4.280721 | 5.770909 | 6.091044 | 4 |
| **2** | 8.329098 | 7.540436 | 3.247239 | 2 |
| **3** | 5.470224 | 5.069249 | 5.768313 | 4 |
| **4** | 2.381797 | 2.402374 | 3.879101 | 1 |

### Euclidean Distance

Let's visualize the difference between the Euclidean and Manhattan distance.

Please use pandas to load the dataset .CSV file and Numpy to compute the **Euclidean distance** to the point (Y=5, Z=5) that we've chosen as a reference. On the left, note the dataset projected onto the YZ plane and color coded per the Euclidean distance we just computed. As we are used to, points that lie at the same Euclidean distance define a regular 2D circle.

Note that the **SciPy library** comes with optimized functions written in C to compute distances (in the scipy.spatial.distance module) that are much faster than our (naive) implementation.

# In the Y-Z plane, we compute the distance to ref point (5,5)

distEuclid1 = np.sqrt((df.Z - 5)\*\*2 + (df.Y - 5)\*\*2)

distEuclid

0 3.022262

1 4.151203

2 4.547162

3 3.456205

4 1.063003

...

1995 2.721163

1996 5.479017

1997 3.834477

1998 3.229189

1999 5.350437

Length: 2000, dtype: float64

**Create a distance to reference point (3,3) matrix similar to the above example.**

distEuclid2 = np.sqrt((df.Z - 3)\*\*2 + (df.Y - 3)\*\*2)

distEuclid

0 3.022262

1 4.151203

2 4.547162

3 3.456205

4 1.063003

...

1995 2.721163

1996 5.479017

1997 3.834477

1998 3.229189

1999 5.350437

Length: 2000, dtype: float64

**Replace the value set to 'c' in the plotting cell below with your own distance matrix and review the result to deepen your understanding of Euclidean distances.**

figEuclid = plt.figure(figsize=[10,8])

plt.scatter(df.Y - 5, df.Z-5, c=distEuclid1, s=20)

plt.ylim([-4.9,4.9])

plt.xlim([-4.9,4.9])

plt.xlabel('Y - 5', size=14)

plt.ylabel('Z - 5', size=14)

plt.title('Euclidean Distance')

cb = plt.colorbar()

cb.set\_label('Distance from (5,5)', size=14)

#figEuclid.savefig('Euclidean.png')

A picture containing screenshot, text, circle, colorfulness

Description automatically generated

### Manhattan Distance

Manhattan distance is simply the sum of absolute differences between the points coordinates. This distance is also known as the taxicab or city block distance as it measures distances along the coorinate axis, which creates "paths" that look like a cab's route on a grid-style city map.

We display the dataset projected on the XZ plane here color coded per the Manhattan distance to the (X=5, Z=5) reference point. We can see that points lying at the same distance define a circle that looks like a Euclidean square.

# In the Y-Z plane, we compute the distance to ref point (5,5)

distManhattan = np.abs(df.X - 5) + np.abs(df.Z - 5)

distManhattan

0 0.923415

1 1.810324

2 5.081859

3 1.238538

4 3.739102

...

1995 0.906694

1996 3.321762

1997 4.779125

1998 0.408304

1999 2.889332

Length: 2000, dtype: float64

**Create a Manhattan distance to reference point (4,4) matrix similar to the above example and replace the value for 'c' in the plotting cell to view the result.**

distManhattan = np.abs(df.X - 4) + np.abs(df.Z - 4)

distManhattan

0 2.923415

1 2.371765

2 5.081859

3 3.238538

4 1.739102

...

1995 2.139184

1996 4.828132

1997 4.779125

1998 1.591696

1999 4.800204

Length: 2000, dtype: float64

figEuclid = plt.figure(figsize=[10,8])

plt.scatter(df.X - 4, df.Z-4, c=distManhattan, s=20)

plt.ylim([-4.9,4.9])

plt.xlim([-4.9,4.9])

plt.xlabel('Y - 4', size=14)

plt.ylabel('Z - 4', size=14)

plt.title('Manhuttan Distance')

cb = plt.colorbar()

cb.set\_label('Distance from (4,4)', size=14)

#figEuclid.savefig('Euclidean.png')

A picture containing text, screenshot, colorfulness, diagram

Description automatically generated

Now let's create distributions of these distance metrics and compare them. We leverage the scipy dist function to create these matrices similar to how you manually created them earlier in the exercise.

import scipy.spatial.distance as dist

mat = df[['X','Y','Z']].to\_numpy()

DistEuclid = dist.pdist(mat,'euclidean')

DistManhattan = dist.pdist(mat, 'cityblock')

largeMat = np.random.random((10000,100))

DistEuclid

array([1.84227209, 5.29408225, 0.96192617, ..., 4.69183711, 4.2880246 ,

3.02005239])

**Plot histograms of each distance matrix for comparison.**

plt.hist(DistEuclid,bins=5)

(array([325486., 493219., 799453., 235697., 145145.]),

array([6.77647886e-03, 1.99122937e+00, 3.97568226e+00, 5.96013516e+00,

7.94458805e+00, 9.92904094e+00]),

<a list of 5 Patch objects>)

A picture containing text, screenshot, diagram, plot

Description automatically generated

plt.hist(DistManhattan,bins=5)

(array([413332., 557218., 719302., 256388., 52760.]),

array([9.86792833e-03, 3.33444799e+00, 6.65902804e+00, 9.98360810e+00,

1.33081882e+01, 1.66327682e+01]),

<a list of 5 Patch objects>)

A picture containing text, screenshot, diagram, plot

Description automatically generated

# How many clusters of grain?

In the video, you learned how to choose a good number of clusters for a dataset using the k-means inertia graph. You are given an array samples containing the measurements (such as area, perimeter, length, and several others) of samples of grain. What's a good number of clusters in this case?

KMeans and PyPlot (plt) have already been imported for you.

This dataset was sourced from the [UCI Machine Learning Repository](https://archive.ics.uci.edu/ml/datasets/seeds).

## Instructions

100 XP

* For each of the given values of k, perform the following steps:
* Create a KMeans instance called model with k clusters.
* Fit the model to the grain data samples.
* Append the value of the inertia\_ attribute of model to the list inertias.
* The code to plot ks vs inertias has been written for you, so hit submit to see the plot!

ks = range(1, 6)

inertias = []

for k in ks:

    # Create a KMeans instance with k clusters: model

    \_\_\_\_

    # Fit model to samples

    \_\_\_\_

    # Append the inertia to the list of inertias

    \_\_\_\_

# Plot ks vs inertias

plt.plot(ks, inertias, '-o')

plt.xlabel('number of clusters, k')

plt.ylabel('inertia')

plt.xticks(ks)

plt.show()

ks = range(1, 6)

inertias = []

for k in ks:

    # Create a KMeans instance with k clusters: model

    model = KMeans(n\_clusters=k)

    # Fit model to samples

    model.fit(samples)

    # Append the inertia to the list of inertias

    inertias.append(model.inertia\_)

# Plot ks vs inertias

plt.plot(ks, inertias, '-o')

plt.xlabel('number of clusters, k')

plt.ylabel('inertia')

plt.xticks(ks)

plt.show()

ks = range(1, 6) inertias = [] for k in ks: # Create a KMeans instance with k clusters: model model = KMeans(n\_clusters=k) # Fit model to samples model.fit(samples) # Append the inertia to the list of inertias inertias.append(model.inertia\_) # Plot ks vs inertias plt.plot(ks, inertias, '-o') plt.xlabel('number of clusters, k') plt.ylabel('inertia') plt.xticks(ks) plt.show()

Excellent job! The inertia decreases very slowly from 3 clusters to 4, so it looks like 3 clusters would be a good choice for this data.

# Evaluating the grain clustering

In the previous exercise, you observed from the inertia plot that 3 is a good number of clusters for the grain data. In fact, the grain samples come from a mix of 3 different grain varieties: "Kama", "Rosa" and "Canadian". In this exercise, cluster the grain samples into three clusters, and compare the clusters to the grain varieties using a cross-tabulation.

You have the array samples of grain samples, and a list varieties giving the grain variety for each sample. Pandas (pd) and KMeans have already been imported for you.

## Instructions

100 XP

* Create a KMeans model called model with 3 clusters.
* Use the .fit\_predict() method of model to fit it to samples and derive the cluster labels. Using .fit\_predict() is the same as using .fit() followed by .predict().
* Create a DataFrame df with two columns named 'labels' and 'varieties', using labels and varieties, respectively, for the column values. This has been done for you.
* Use the pd.crosstab() function on df['labels'] and df['varieties'] to count the number of times each grain variety coincides with each cluster label. Assign the result to ct.
* Hit submit to see the cross-tabulation!
* # Create a KMeans model with 3 clusters: model
* model = \_\_\_\_
* # Use fit\_predict to fit model and obtain cluster labels: labels
* labels = \_\_\_\_
* # Create a DataFrame with labels and varieties as columns: df
* df = pd.DataFrame({'labels': labels, 'varieties': varieties})
* # Create crosstab: ct
* ct = \_\_\_\_
* # Display ct
* print(ct)

# Create a KMeans model with 3 clusters: model

model = KMeans(n\_clusters=3)

# Use fit\_predict to fit model and obtain cluster labels: labels

labels = model.fit\_predict(samples)

# Create a DataFrame with labels and varieties as columns: df

df = pd.DataFrame({'labels': labels, 'varieties': varieties})

# Create crosstab: ct

ct = pd.crosstab(df['labels'], df['varieties'])

# Display ct

print(ct)

Great work! The cross-tabulation shows that the 3 varieties of grain separate really well into 3 clusters. But depending on the type of data you are working with, the clustering may not always be this good. Is there anything you can do in such situations to improve your clustering? You'll find out in the next video!

# Create a KMeans model with 3 clusters: model

model = KMeans(n\_clusters=3)

# Use fit\_predict to fit model and obtain cluster labels: labels

labels = model.fit\_predict(samples)

# Create a DataFrame with labels and varieties as columns: df

df = pd.DataFrame({'labels': labels, 'varieties': varieties})

# Create crosstab: ct

ct = pd.crosstab(df['labels'], df['varieties'])

# Display ct

print(ct)

varieties Canadian wheat Kama wheat Rosa wheat

labels

0 0 1 60

1 68 9 0

2 2 60 10

<script.py> output:

varieties Canadian wheat Kama wheat Rosa wheat

labels

0 0 1 60

1 68 9 0

2 2 60 10

## 1. Transforming features for better clusterings

00:00 - 00:04

Let's look now at another dataset,

## 2. Piedmont wines dataset

00:04 - 00:23

the Piedmont wines dataset. We have 178 samples of red wine from the Piedmont region of Italy. The features measure chemical composition (like alcohol content) and visual properties like color intensity. The samples come from 3 distinct varieties of wine.

1. 1 Source: https://archive.ics.uci.edu/ml/datasets/Wine

## 3. Clustering the wines

00:23 - 00:30

Let's take the array of samples and use KMeans to find 3 clusters.

## 4. Clusters vs. varieties

00:30 - 00:46

There are three varieties of wine, so let's use pandas crosstab to check the cluster label - wine variety correspondence. As you can see, this time things haven't worked out so well. The KMeans clusters don't correspond well with the wine varieties.

## 5. Feature variances

00:46 - 01:00

The problem is that the features of the wine dataset have very different variances. The variance of a feature measures the spread of its values. For example, the malic acid feature has a higher variance

## 6. Feature variances

01:00 - 01:12

than the od280 feature, and this can also be seen in their scatter plot. The differences in some of the feature variances is enormous, as seen here, for example, in the scatter plot of the od280 and proline features.

## 7. StandardScaler

01:12 - 01:46

In KMeans clustering, the variance of a feature corresponds to its influence on the clustering algorithm. To give every feature a chance, the data needs to be transformed so that features have equal variance. This can be achieved with the StandardScaler from scikit-learn. It transforms every feature to have mean 0 and variance 1. The resulting "standardized" features can be very informative. Using standardized od280 and proline, for example, the three wine varieties are much more distinct.

## 8. sklearn StandardScaler

01:46 - 02:07

Let's see the StandardScaler in action. First, import StandardScaler from sklearn.preprocessing. Then create a StandardScaler object, and fit it to the samples. The transform method can now be used to standardize any samples, either the same ones, or completely new ones.

## 9. Similar methods

02:07 - 02:26

The APIs of StandardScaler and KMeans are similar, but there is an important difference. StandardScaler transforms data, and so has a transform method. KMeans, in contrast, assigns cluster labels to samples, and this done using the predict method.

## 10. StandardScaler, then KMeans

02:26 - 02:54

Let's return to the problem of clustering the wines. We need to perform two steps. Firstly, to standardize the data using StandardScaler, and secondly to take the standardized data and cluster it using KMeans. This can be conveniently achieved by combining the two steps using a scikit-learn pipeline. Data then flows from one step into the next, automatically.

## 11. Pipelines combine multiple steps

02:54 - 03:25

The first steps are the same: creating a StandardScaler and a KMeans object. After that, import the make\_pipeline function from sklearn.pipeline. Apply the make\_pipeline function to the steps that you want to compose in this case, the scaler and the kmeans objects. Now use the fit method of the pipeline to fit both the scaler and kmeans, and use its predict method to obtain the cluster labels.

## 12. Feature standardization improves clustering

03:25 - 03:45

Checking the correspondence between the cluster labels and the wine varieties reveals that this new clustering, incorporating standardization, is fantastic. Its three clusters correspond almost exactly to the three wine varieties. This is a huge improvement on the clustering without standardization.

## 13. sklearn preprocessing steps

03:45 - 03:58

StandardScaler is an example of a "preprocessing" step. There are several of these available in scikit-learn, for example MaxAbsScaler and Normalizer.

## 14. Let's practice!

03:58 - 04:06

You've learned a lot this video. Now put it into practice and cluster some datasets!

# Scaling fish data for clustering

You are given an array samples giving measurements of fish. Each row represents an individual fish. The measurements, such as weight in grams, length in centimeters, and the percentage ratio of height to length, have very different scales. In order to cluster this data effectively, you'll need to standardize these features first. In this exercise, you'll build a pipeline to standardize and cluster the data.

These fish measurement data were sourced from the [Journal of Statistics Education](http://ww2.amstat.org/publications/jse/jse_data_archive.htm).

## Instructions

100 XP

* Import:
  + make\_pipeline from sklearn.pipeline.
  + StandardScaler from sklearn.preprocessing.
  + KMeans from sklearn.cluster.
* Create an instance of StandardScaler called scaler.
* Create an instance of KMeans with 4 clusters called kmeans.
* Create a pipeline called pipeline that chains scaler and kmeans. To do this, you just need to pass them in as arguments to make\_pipeline().

# Perform the necessary imports

from \_\_\_\_ import \_\_\_\_

from \_\_\_\_ import \_\_\_\_

from \_\_\_\_ import \_\_\_\_

# Create scaler: scaler

scaler = \_\_\_\_

# Create KMeans instance: kmeans

kmeans = \_\_\_\_

# Create pipeline: pipeline

pipeline = \_\_\_\_

# Perform the necessary imports

from sklearn.pipeline import make\_pipeline

from sklearn.preprocessing import StandardScaler

from sklearn.cluster import KMeans

# Create scaler: scaler

scaler = StandardScaler()

# Create KMeans instance: kmeans

kmeans = KMeans(n\_clusters=4)

# Create pipeline: pipeline

pipeline = make\_pipeline(scaler, kmeans)

# Perform the necessary imports from sklearn.pipeline import make\_pipeline from sklearn.preprocessing import StandardScaler from sklearn.cluster import KMeans # Create scaler: scaler scaler = StandardScaler() # Create KMeans instance: kmeans kmeans = KMeans(n\_clusters=4) # Create pipeline: pipeline pipeline = make\_pipeline(scaler, kmeans)

Great work! Now that you've built the pipeline, you'll use it in the next exercise to cluster the fish by their measurements.

# Clustering the fish data

You'll now use your standardization and clustering pipeline from the previous exercise to cluster the fish by their measurements, and then create a cross-tabulation to compare the cluster labels with the fish species.

As before, samples is the 2D array of fish measurements. Your pipeline is available as pipeline, and the species of every fish sample is given by the list species.

## Instructions

100 XP

* Import pandas as pd.
* Fit the pipeline to the fish measurements samples.
* Obtain the cluster labels for samples by using the .predict() method of pipeline.
* Using pd.DataFrame(), create a DataFrame df with two columns named 'labels' and 'species', using labels and species, respectively, for the column values.
* Using pd.crosstab(), create a cross-tabulation ct of df['labels'] and df['species'].
* # Import pandas
* import pandas as pd
* # Fit the pipeline to samples
* \_\_\_\_
* # Calculate the cluster labels: labels
* labels = \_\_\_\_
* # Create a DataFrame with labels and species as columns: df
* df = \_\_\_\_
* # Create crosstab: ct
* ct = \_\_\_\_
* # Display ct
* print(ct)

# Import pandas

import pandas as pd

# Fit the pipeline to samples

pipeline.fit(samples)

# Calculate the cluster labels: labels

labels = pipeline.predict(samples)

# Create a DataFrame with labels and species as columns: df

df = pd.DataFrame({'labels':labels, 'species':species})

# Create crosstab: ct

ct = pd.crosstab(df['labels'], df['species'])

# Display ct

print(ct)

# Import pandas

import pandas as pd

# Fit the pipeline to samples

pipeline.fit(samples)

# Calculate the cluster labels: labels

labels = pipeline.predict(samples)

# Create a DataFrame with labels and species as columns: df

df = pd.DataFrame({'labels':labels, 'species':species})

# Create crosstab: ct

ct = pd.crosstab(df['labels'], df['species'])

# Display ct

print(ct)

species Bream Pike Roach Smelt

labels

0 0 17 0 0

1 33 0 1 0

2 0 0 0 13

3 1 0 19 1

<script.py> output:

species Bream Pike Roach Smelt

labels

0 0 0 0 13

1 33 0 1 0

2 0 17 0 0

3 1 0 19 1

Excellent! It looks like the fish data separates really well into 4 clusters!

# Clustering stocks using KMeans

In this exercise, you'll cluster companies using their daily stock price movements (i.e. the dollar difference between the closing and opening prices for each trading day). You are given a NumPy array movements of daily price movements from 2010 to 2015 (obtained from Yahoo! Finance), where each row corresponds to a company, and each column corresponds to a trading day.

Some stocks are more expensive than others. To account for this, include a Normalizer at the beginning of your pipeline. The Normalizer will separately transform each company's stock price to a relative scale before the clustering begins.

Note that Normalizer() is different to StandardScaler(), which you used in the previous exercise. While StandardScaler() standardizes **features** (such as the features of the fish data from the previous exercise) by removing the mean and scaling to unit variance, Normalizer() rescales **each sample** - here, each company's stock price - independently of the other.

KMeans and make\_pipeline have already been imported for you.

## Instructions

100 XP

* Import Normalizer from sklearn.preprocessing.
* Create an instance of Normalizer called normalizer.
* Create an instance of KMeans called kmeans with 10 clusters.
* Using make\_pipeline(), create a pipeline called pipeline that chains normalizer and kmeans.
* Fit the pipeline to the movements array.
* # Import Normalizer
* \_\_\_\_
* # Create a normalizer: normalizer
* normalizer = \_\_\_\_
* # Create a KMeans model with 10 clusters: kmeans
* kmeans = \_\_\_\_
* # Make a pipeline chaining normalizer and kmeans: pipeline
* pipeline = \_\_\_\_
* # Fit pipeline to the daily price movements
* \_\_\_\_

# Import Normalizer

from sklearn.preprocessing import Normalizer

# Create a normalizer: normalizer

normalizer = Normalizer()

# Create a KMeans model with 10 clusters: kmeans

kmeans = KMeans(n\_clusters=10)

# Make a pipeline chaining normalizer and kmeans: pipeline

pipeline = make\_pipeline(normalizer, kmeans)

# Fit pipeline to the daily price movements

pipeline.fit(movements)

# Import Normalizer

from sklearn.preprocessing import Normalizer

# Create a normalizer: normalizer

normalizer = Normalizer()

# Create a KMeans model with 10 clusters: kmeans

kmeans = KMeans(n\_clusters=10)

# Make a pipeline chaining normalizer and kmeans: pipeline

pipeline = make\_pipeline(normalizer, kmeans)

# Fit pipeline to the daily price movements

pipeline.fit(movements)

Pipeline(steps=[('normalizer', Normalizer()),

('kmeans', KMeans(n\_clusters=10))])

Great work - you're really getting the hang of this. Now that your pipeline has been set up, you can find out which stocks move together in the next exercise!

# Which stocks move together?

In the previous exercise, you clustered companies by their daily stock price movements. So which company have stock prices that tend to change in the same way? You'll now inspect the cluster labels from your clustering to find out.

Your solution to the previous exercise has already been run. Recall that you constructed a Pipeline pipeline containing a KMeans model and fit it to the NumPy array movements of daily stock movements. In addition, a list companies of the company names is available.

## Instructions

100 XP

* Import pandas as pd.
* Use the .predict() method of the pipeline to predict the labels for movements.
* Align the cluster labels with the list of company names companies by creating a DataFrame df with labels and companies as columns. This has been done for you.
* Use the .sort\_values() method of df to sort the DataFrame by the 'labels' column, and print the result.
* Hit submit and take a moment to see which companies are together in each cluster!

# Import pandas

import pandas as pd

# Predict the cluster labels: labels

labels = \_\_\_\_

# Create a DataFrame aligning labels and companies: df

df = pd.DataFrame({'labels': labels, 'companies': companies})

# Display df sorted by cluster label

print(\_\_\_\_)

# Import pandas

import pandas as pd

# Predict the cluster labels: labels

labels = pipeline.predict(movements)

# Create a DataFrame aligning labels and companies: df

df = pd.DataFrame({'labels': labels, 'companies': companies})

# Display df sorted by cluster label

print(df.sort\_values('labels'))

# Import pandas

import pandas as pd

# Predict the cluster labels: labels

labels = pipeline.predict(movements)

# Create a DataFrame aligning labels and companies: df

df = pd.DataFrame({'labels': labels, 'companies': companies})

# Display df sorted by cluster label

print(df.sort\_values('labels'))

<script.py> output:

labels companies

59 0 Yahoo

15 0 Ford

35 0 Navistar

26 1 JPMorgan Chase

16 1 General Electrics

58 1 Xerox

11 1 Cisco

18 1 Goldman Sachs

20 1 Home Depot

5 1 Bank of America

3 1 American express

55 1 Wells Fargo

1 1 AIG

38 2 Pepsi

40 2 Procter Gamble

28 2 Coca Cola

27 2 Kimberly-Clark

9 2 Colgate-Palmolive

54 3 Walgreen

36 3 Northrop Grumman

29 3 Lookheed Martin

4 3 Boeing

0 4 Apple

47 4 Symantec

33 4 Microsoft

32 4 3M

31 4 McDonalds

30 4 MasterCard

50 4 Taiwan Semiconductor Manufacturing

14 4 Dell

17 4 Google/Alphabet

24 4 Intel

23 4 IBM

2 4 Amazon

51 4 Texas instruments

43 4 SAP

45 5 Sony

48 5 Toyota

21 5 Honda

22 5 HP

34 5 Mitsubishi

7 5 Canon

56 6 Wal-Mart

57 7 Exxon

44 7 Schlumberger

8 7 Caterpillar

10 7 ConocoPhillips

12 7 Chevron

13 7 DuPont de Nemours

53 7 Valero Energy

39 8 Pfizer

41 8 Philip Morris

25 8 Johnson & Johnson

49 9 Total

46 9 Sanofi-Aventis

37 9 Novartis

42 9 Royal Dutch Shell

19 9 GlaxoSmithKline

52 9 Unilever

6 9 British American Tobacco

## 1. Visualizing hierarchies

00:00 - 00:08

A huge part of your work as a data scientist will be the communication of your insights to other people.

## 2. Visualizations communicate insight

00:08 - 00:37

Visualizations are an excellent way to share your findings, particularly with a non-technical audience. In this chapter, you'll learn about two unsupervised learning techniques for visualization: t-SNE and hierarchical clustering. t-SNE, which we'll consider later, creates a 2d map of any dataset, and conveys useful information about the proximity of the samples to one another. First up, however, let's learn about hierarchical clustering.

## 3. A hierarchy of groups

00:37 - 01:08

You've already seen many hierarchical clusterings in the real world. For example, living things can be organized into small narrow groups, like humans, apes, snakes and lizards, or into larger, broader groups like mammals and reptiles, or even broader groups like animals and plants. These groups are contained in one another, and form a hierarchy. Analogously, hierarchical clustering arranges samples into a hierarchy of clusters.

## 4. Eurovision scoring dataset

01:08 - 01:37

Hierarchical clustering can organize any sort of data into a hierarchy, not just samples of plants and animals. Let's consider a new type of dataset, describing how countries scored performances at the Eurovision 2016 song contest. The data is arranged in a rectangular array, where the rows of the array show how many points a country gave to each song. The "samples" in this case are the countries.

1. 1 https://www.eurovision.tv/page/results

## 5. Hierarchical clustering of voting countries

01:37 - 02:17

The result of applying hierarchical clustering to the Eurovision scores can be visualized as a tree-like diagram called a "dendrogram". This single picture reveals a great deal of information about the voting behavior of countries at the Eurovision. The dendrogram groups the countries into larger and larger clusters, and many of these clusters are immediately recognizable as containing countries that are close to one another geographically, or that have close cultural or political ties, or that belong to single language group. So hierarchical clustering can produce great visualizations. But how does it work?

## 6. Hierarchical clustering

02:17 - 02:59

Hierarchical clustering proceeds in steps. In the beginning, every country is its own cluster - so there are as many clusters as there are countries! At each step, the two closest clusters are merged. This decreases the number of clusters, and eventually, there is only one cluster left, and it contains all the countries. This process is actually a particular type of hierarchical clustering called "agglomerative clustering" - there is also "divisive clustering", which works the other way around. We haven't defined yet what it means for two clusters to be close, but we'll revisit that later on.

## 7. The dendrogram of a hierarchical clustering

02:59 - 03:23

The entire process of the hierarchical clustering is encoded in the dendrogram. At the bottom, each country is in a cluster of its own. The clustering then proceeds from the bottom up. Clusters are represented as vertical lines, and a joining of vertical lines indicates a merging of clusters. To understand better, let's zoom in

## 8. The dendrogram of a hierarchical clustering

03:23 - 03:26

and look at just one part of this dendrogram.

## 9. Dendrograms, step-by-step

03:26 - 03:33

In the beginning, there are six clusters, each containing only one country.

## 10. Dendrograms, step-by-step

03:33 - 03:41

The first merging is here, where the clusters containing Cyprus and Greece are merged together in a single cluster.

## 11. Dendrograms, step-by-step

03:41 - 03:46

Later on, this new cluster is merged with the cluster containing Bulgaria.

## 12. Dendrograms, step-by-step

03:46 - 03:52

Shortly after that, the clusters containing Moldova and Russia are merged,

## 13. Dendrograms, step-by-step

03:52 - 03:57

which later is in turn merged with the cluster containing Armenia.

## 14. Dendrograms, step-by-step

03:57 - 04:03

Later still, the two big composite clusters are merged together. This process continues

## 15. Dendrograms, step-by-step

04:03 - 04:09

until there is only one cluster left, and it contains all the countries.

## 16. Hierarchical clustering with SciPy

04:09 - 04:52

We'll use functions from scipy to perform a hierarchical clustering on the array of scores. For the dendrogram, we'll also need a list of country names. Firstly, import the linkage and dendrogram functions. Then, apply the linkage function to the sample array. Its the linkage function that performs the hierarchical clustering. Notice there is an extra method parameter - we'll cover that in the next video. Now pass the output of linkage to the dendrogram function, specifying the list of country names as the labels parameter. In the next video, you'll learn how to extract information from a hierarchical clustering,

## 17. Let's practice!

04:52 - 05:01

But for now, let's see what hierarchical clustering can do with some real-world datasets.

# How many merges?

If there are 5 data samples, how many merge operations will occur in a hierarchical clustering? (To help answer this question, think back to the video, in which Ben walked through an example of hierarchical clustering using 6 countries.)

##### Answer the question

**50XP**

#### Possible Answers

* 

4 merges.

press1

* 

3 merges.

press2

* 

This can't be known in advance.

Well done! With 5 data samples, there would be 4 merge operations, and with 6 data samples, there would be 5 merges, and so on.

# Hierarchical clustering of the grain data

In the video, you learned that the SciPy linkage() function performs hierarchical clustering on an array of samples. Use the linkage() function to obtain a hierarchical clustering of the grain samples, and use dendrogram() to visualize the result. A sample of the grain measurements is provided in the array samples, while the variety of each grain sample is given by the list varieties.

## Instructions

100 XP

* Import:
  + linkage and dendrogram from scipy.cluster.hierarchy.
  + matplotlib.pyplot as plt.
* Perform hierarchical clustering on samples using the linkage() function with the method='complete' keyword argument. Assign the result to mergings.
* Plot a dendrogram using the dendrogram() function on mergings. Specify the keyword arguments labels=varieties, leaf\_rotation=90, and leaf\_font\_size=6.
* # Perform the necessary imports
* from \_\_\_\_ import \_\_\_\_, \_\_\_\_
* import \_\_\_\_ as \_\_\_\_
* # Calculate the linkage: mergings
* mergings = \_\_\_\_
* # Plot the dendrogram, using varieties as labels
* dendrogram(\_\_\_\_,
* labels=\_\_\_\_,
* leaf\_rotation=\_\_\_\_,
* leaf\_font\_size=\_\_\_\_,
* )
* plt.show()

# Perform the necessary imports

from scipy.cluster.hierarchy import linkage, dendrogram

import matplotlib.pyplot as plt

# Calculate the linkage: mergings

mergings = linkage(samples, method='complete')

# Plot the dendrogram, using varieties as labels

dendrogram(mergings,

           labels=varieties,

           leaf\_rotation=90,

           leaf\_font\_size=6,

)

plt.show()

# Perform the necessary imports from scipy.cluster.hierarchy import linkage, dendrogram import matplotlib.pyplot as plt # Calculate the linkage: mergings mergings = linkage(samples, method='complete') # Plot the dendrogram, using varieties as labels dendrogram(mergings, labels=varieties, leaf\_rotation=90, leaf\_font\_size=6, ) plt.show()

Superb! Dendrograms are a great way to illustrate the arrangement of the clusters produced by hierarchical clustering.

# Hierarchies of stocks

In chapter 1, you used k-means clustering to cluster companies according to their stock price movements. Now, you'll perform hierarchical clustering of the companies. You are given a NumPy array of price movements movements, where the rows correspond to companies, and a list of the company names companies. SciPy hierarchical clustering doesn't fit into a sklearn pipeline, so you'll need to use the normalize() function from sklearn.preprocessing instead of Normalizer.

linkage and dendrogram have already been imported from scipy.cluster.hierarchy, and PyPlot has been imported as plt.

## Instructions

100 XP

* Import normalize from sklearn.preprocessing.
* Rescale the price movements for each stock by using the normalize() function on movements.
* Apply the linkage() function to normalized\_movements, using 'complete' linkage, to calculate the hierarchical clustering. Assign the result to mergings.
* Plot a dendrogram of the hierarchical clustering, using the list companies of company names as the labels. In addition, specify the leaf\_rotation=90, and leaf\_font\_size=6 keyword arguments as you did in the previous exercise.
* # Import normalize
* \_\_\_\_
* # Normalize the movements: normalized\_movements
* normalized\_movements = \_\_\_\_
* # Calculate the linkage: mergings
* mergings = \_\_\_\_
* # Plot the dendrogram
* \_\_\_\_
* plt.show()
* # Import normalize
* from sklearn.preprocessing import normalize
* # Normalize the movements: normalized\_movements
* normalized\_movements = normalize(movements)
* # Calculate the linkage: mergings
* mergings = linkage(normalized\_movements, method='complete')
* # Plot the dendrogram
* dendrogram(mergings, labels=companies, leaf\_rotation=90, leaf\_font\_size=6)
* plt.show()

# Import normalize from sklearn.preprocessing import normalize # Normalize the movements: normalized\_movements normalized\_movements = normalize(movements) # Calculate the linkage: mergings mergings = linkage(normalized\_movements, method='complete') # Plot the dendrogram dendrogram(mergings, labels=companies, leaf\_rotation=90, leaf\_font\_size=6) plt.show()

Great work! You can produce great visualizations such as this with hierarchical clustering, but it can be used for more than just visualizations. You'll find out more about this in the next video!

**Hierarchical Clustering Questions**

When using clustering for outlier detection in the multi-feature object trajectory clustering for video analysis example, how do the outliers present themselves?

a. As data points in their own right

b.As absences of data points

c. As clusters

d. As absences of clusters

Hierarchical clustering requires us to specify neither k nor a window size, in advance

TRUE

FALSE

In what situation is hierarchical clustering typically used?

a. When you have almost no data

b. When you know a reasonable value for k

c. When you know a reasonable value for a window size

d. When you have all the data

One can sensibly and informatively model the animal kingdom with hierarchical clustering

TRUE

FALSE

The following is a good non-mathematical representation of the hierarchical clustering algorithm: 1. Make each point is its own cluster 2. Look at two points that have the shortest distance (this requires computing the distance between every point) 3. Group those two points together 4. Look at the next pair 5. Group those two points together 6. Continue in this vein until a point is assigned to a tuple (like a pair or triple) 7. Choose a threshold in the tree where the algorithm cuts off 8. Finish

TRUE

FALSE

Hierarchical clustering produces an incomplete structure with a predefined number of clusters

TRUE

FALSE

What is single-linkage?

a. The minimum distance between any two points in two different clusters

b. The smallest maximum distance between points in two different clusters

c. A compromise between (A) and (B)

d. None of the above

What is the problem with complete linkage as a means of measuring the similarity between clusters?

a. It makes long chains

b. It’s sensitive to outliers

c. It takes too long to acquire

d. None of the above

What is the Rand index?

a. (TruePositive + TrueNegative) / (TruePositive + TrueNegative + FalsePositive + FalseNegative)

b. (TruePositive + FalsePositive) / (TruePositive + TrueNegative + FalsePositive + FalseNegative)

c. (TruePositive + TrueNegative) / (TruePositive \* TrueNegative \* FalsePositive \* FalseNegative)

d. (FalsePositive + FalseNegative) / (TruePositive + TrueNegative + FalsePositive + FalseNegative)

What is the driving motivation behind the evaluation metric of stability?

a. The idea that clustering solutions ought to generalize to new, unseen data

b. The idea that clustering solutions should not have many outliers

c. The idea that clustering solutions should be accurate

d. The idea that clustering solutions should be robust

***A Tutorial on Clustering Algorithms***

[Introduction](https://matteucci.faculty.polimi.it/Clustering/tutorial_html/index.html) | [K-means](https://matteucci.faculty.polimi.it/Clustering/tutorial_html/kmeans.html) | [Fuzzy C-means](https://matteucci.faculty.polimi.it/Clustering/tutorial_html/cmeans.html) | Hierarchical | [Mixture of Gaussians](https://matteucci.faculty.polimi.it/Clustering/tutorial_html/mixture.html) | [Links](https://matteucci.faculty.polimi.it/Clustering/tutorial_html/links.html)

**Hierarchical Clustering Algorithms**

*How They Work*  
Given a set of N items to be clustered, and an N\*N distance (or similarity) matrix, the basic process of hierarchical clustering (defined by [S.C. Johnson in 1967](https://matteucci.faculty.polimi.it/Clustering/tutorial_html/hierarchical.html#johnson)) is this:

1. Start by assigning each item to a cluster, so that if you have N items, you now have N clusters, each containing just one item. Let the distances (similarities) between the clusters the same as the distances (similarities) between the items they contain.
2. Find the closest (most similar) pair of clusters and merge them into a single cluster, so that now you have one cluster less.
3. Compute distances (similarities) between the new cluster and each of the old clusters.
4. Repeat steps 2 and 3 until all items are clustered into a single cluster of size N. (\*)

Step 3 can be done in different ways, which is what distinguishes *single-linkage* from *complete-linkage* and *average-linkage* clustering.  
In *single-linkage* clustering (also called the *connectedness* or *minimum* method), we consider the distance between one cluster and another cluster to be equal to the shortest distance from any member of one cluster to any member of the other cluster. If the data consist of similarities, we consider the similarity between one cluster and another cluster to be equal to the greatest similarity from any member of one cluster to any member of the other cluster.  
In *complete-linkage* clustering (also called the *diameter* or *maximum* method), we consider the distance between one cluster and another cluster to be equal to the greatest distance from any member of one cluster to any member of the other cluster.  
In *average-linkage* clustering, we consider the distance between one cluster and another cluster to be equal to the average distance from any member of one cluster to any member of the other cluster.  
A variation on average-link clustering is the UCLUS method of [R. D'Andrade (1978)](https://matteucci.faculty.polimi.it/Clustering/tutorial_html/hierarchical.html#dandrade) which uses the median distance, which is much more outlier-proof than the average distance.

This kind of hierarchical clustering is called *agglomerative* because it merges clusters iteratively. There is also a *divisive* hierarchical clustering which does the reverse by starting with all objects in one cluster and subdividing them into smaller pieces. Divisive methods are not generally available, and rarely have been applied.

(\*) Of course there is no point in having all the N items grouped in a single cluster but, once you have got the complete hierarchical tree, if you want k clusters you just have to cut the k-1 longest links.

*Single-Linkage Clustering: The Algorithm*  
Let’s now take a deeper look at how Johnson’s algorithm works in the case of single-linkage clustering.  
The algorithm is an agglomerative scheme that erases rows and columns in the proximity matrix as old clusters are merged into new ones.

The N\*N proximity matrix is D = [d(i,j)]. The clusterings are assigned sequence numbers 0,1,......, (n-1) and L(k) is the level of the kth clustering. A cluster with sequence number m is denoted (m) and the proximity between clusters (r) and (s) is denoted d [(r),(s)].

The algorithm is composed of the following steps:

|  |
| --- |
| 1. *Begin with the disjoint clustering having level L(0) = 0 and sequence number m = 0.* 2. *Find the least dissimilar pair of clusters in the current clustering, say pair (r), (s), according to  d[(r),(s)] = min d[(i),(j)]  where the minimum is over all pairs of clusters in the current clustering.* 3. *Increment the sequence number : m = m +1. Merge clusters (r) and (s) into a single cluster to form the next clustering m. Set the level of this clustering to  L(m) = d[(r),(s)]* 4. *Update the proximity matrix, D, by deleting the rows and columns corresponding to clusters (r) and (s) and adding a row and column corresponding to the newly formed cluster. The proximity between the new cluster, denoted (r,s) and old cluster (k) is defined in this way:  d[(k), (r,s)] = min d[(k),(r)], d[(k),(s)]* 5. *If all objects are in one cluster, stop. Else, go to step 2.* |

*An Example*  
Let’s now see a simple example: a hierarchical clustering of distances in kilometers between some Italian cities. The method used is single-linkage.

**Input distance matrix** (L = 0 for all the clusters):

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **BA** | **FI** | **MI** | **NA** | **RM** | **TO** |
| **BA** | 0 | 662 | 877 | 255 | 412 | 996 |
| **FI** | 662 | 0 | 295 | 468 | 268 | 400 |
| **MI** | 877 | 295 | 0 | 754 | 564 | 138 |
| **NA** | 255 | 468 | 754 | 0 | 219 | 869 |
| **RM** | 412 | 268 | 564 | 219 | 0 | 669 |
| **TO** | 996 | 400 | 138 | 869 | 669 | 0 |

A map of italy with black dots

Description automatically generated with low confidence

The nearest pair of cities is MI and TO, at distance 138. These are merged into a single cluster called "MI/TO". The level of the new cluster is L(MI/TO) = 138 and the new sequence number is m = 1.  
Then we compute the distance from this new compound object to all other objects. In single link clustering the rule is that the distance from the compound object to another object is equal to the shortest distance from any member of the cluster to the outside object. So the distance from "MI/TO" to RM is chosen to be 564, which is the distance from MI to RM, and so on.

After merging MI with TO we obtain the following matrix:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **BA** | **FI** | **MI/TO** | **NA** | **RM** |
| **BA** | 0 | 662 | 877 | 255 | 412 |
| **FI** | 662 | 0 | 295 | 468 | 268 |
| **MI/TO** | 877 | 295 | 0 | 754 | 564 |
| **NA** | 255 | 468 | 754 | 0 | 219 |
| **RM** | 412 | 268 | 564 | 219 | 0 |

A picture containing map, text

Description automatically generated

min d(i,j) = d(NA,RM) = 219 => merge NA and RM into a new cluster called NA/RM  
L(NA/RM) = 219  
m = 2

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **BA** | **FI** | **MI/TO** | **NA/RM** |
| **BA** | 0 | 662 | 877 | 255 |
| **FI** | 662 | 0 | 295 | 268 |
| **MI/TO** | 877 | 295 | 0 | 564 |
| **NA/RM** | 255 | 268 | 564 | 0 |

A map of italy with black lines

Description automatically generated with low confidence

min d(i,j) = d(BA,NA/RM) = 255 => merge BA and NA/RM into a new cluster called BA/NA/RM  
L(BA/NA/RM) = 255  
m = 3

|  |  |  |  |
| --- | --- | --- | --- |
|  | **BA/NA/RM** | **FI** | **MI/TO** |
| **BA/NA/RM** | 0 | 268 | 564 |
| **FI** | 268 | 0 | 295 |
| **MI/TO** | 564 | 295 | 0 |

A map of italy with black lines

Description automatically generated with low confidence

min d(i,j) = d(BA/NA/RM,FI) = 268 => merge BA/NA/RM and FI into a new cluster called BA/FI/NA/RM  
L(BA/FI/NA/RM) = 268  
m = 4

|  |  |  |
| --- | --- | --- |
|  | **BA/FI/NA/RM** | **MI/TO** |
| **BA/FI/NA/RM** | 0 | 295 |
| **MI/TO** | 295 | 0 |

A map of italy with black lines

Description automatically generated with low confidence

Finally, we merge the last two clusters at level 295.

The process is summarized by the following hierarchical tree:

A picture containing diagram, sketch, line, technical drawing

Description automatically generated

*Problems*  
The main weaknesses of agglomerative clustering methods are:

* they do not scale well: time complexity of at least *O(n2)*, where n is the number of total objects;
* they can never undo what was done previously.

*Bibliography*

* S. C. Johnson (1967): "Hierarchical Clustering Schemes" *Psychometrika*, 2:241-254
* R. D'andrade (1978): "U-Statistic Hierarchical Clustering" *Psychometrika*, 4:58-67
* Andrew Moore: “K-means and Hierarchical Clustering - Tutorial Slides”  
  [http://www-2.cs.cmu.edu/~awm/tutorials/kmeans.html](http://www-2.cs.cmu.edu/%7Eawm/tutorials/kmeans.html)
* Osmar R. Zaïane: “Principles of Knowledge Discovery in Databases - Chapter 8: Data Clustering”  
  [http://www.cs.ualberta.ca/~zaiane/courses/cmput690/slides/Chapter8/index.html](http://www.cs.ualberta.ca/%7Ezaiane/courses/cmput690/slides/Chapter8/index.html)
* Stephen P. Borgatti: “How to explain hierarchical clustering”  
  <http://www.analytictech.com/networks/hiclus.htm>
* Maria Irene Miranda: “Clustering methods and algorithms”  
  [http://www.cse.iitb.ac.in/dbms/Data/Courses/CS632/1999/clustering/dbms.htm](http://www.cse.iitb.ac.in/dbms/Data/Courses/CS632/1999/clustering/dbms.html)

**Three Popular Clustering Methods and When to Use Each**

[[](https://medium.com/@odsc?source=post_page-----4227c80ba2b6--------------------------------)](https://medium.com/@odsc?source=post_page-----4227c80ba2b6--------------------------------)

[[Predict](https://medium.com/predict?source=post_page-----4227c80ba2b6--------------------------------)](https://medium.com/predict?source=post_page-----4227c80ba2b6--------------------------------)

[ODSC - Open Data Science](https://medium.com/@odsc?source=post_page-----4227c80ba2b6--------------------------------)

·

Published in

[Predict](https://medium.com/predict?source=post_page-----4227c80ba2b6--------------------------------)

·

6 min read

·

Sep 21, 2018

A picture containing diagram, sketch, drawing, art

Description automatically generated

In the mad rush to find new ways of teasing apart labeled data, we often forget about everything we can do with unsupervised learning. Unsupervised [**machine learning**](https://opendatascience.com/tag/machine-learning/) can be very powerful in its own right, and clustering is by far the most common expression of this group of problems.

This is a quick run-down of three of the most popular clustering approaches and what types of situations each is best-suited to. The one thing clustering has in common with supervised problems is that there is no silver bullet; each algorithm will have its time and place depending on what you’re trying to accomplish. This should give you some intuition as far as when to use each, with only a smidge of math.

**Hierarchical Clustering**

Imagine you have some number of clusters *k* you’re interested in finding. All you know is that you can probably break up your dataset into that many distinct groups at the top level, but you might also be interested in the groups inside your groups, or the groups inside of those groups. To get that kind of structure, we use hierarchical clustering.

We begin with *n* different points and *k* different clusters we want to discover; for our purposes, *n* = 4, and *k* = 2.

A picture containing sketch, drawing, black and white, illustration

Description automatically generated

Start by treating each point as if it were its own cluster.

A picture containing sketch, silhouette

Description automatically generated

We then start merging each of our single-point clusters into larger clusters, using the clusters that are closest together. We find the smallest distance from the pairwise distance matrix — which is just a table of the distance of every cluster from every other cluster.

This is initialized as the Euclidean distance between each point, but after that, we switch to one of a variety of different ways of measuring cluster distance. If you’re interested in learning more about these techniques, look up single link clustering, complete link clustering, clique margins, and Ward’s Method.

Anyways, we start merging the clusters that are closest together. Say we know that x1 and x3are closest together. We then merge those two into a new cluster, *ca.*

A picture containing sketch

Description automatically generated

We now recalculate ca’s distance from every other cluster, using whichever method we prefer from the aforementioned grab-bag. Then we repeat, merging our clusters over and over until we get *k* top-level clusters — in our example, two clusters. Say we figure out that x2 is closer to ca than to x4.

A picture containing sketch, line, diagram

Description automatically generated

We now have two top-level clusters, cb and x4 (remember that each point starts as its own cluster). We can now search the tree structure we’ve created to search for sub-clusters, which in our original 2-D view would look like this:

A picture containing clipart, illustration, emoticon, cartoon

Description automatically generated

**Density-Based Clustering**

Hierarchical clustering is advantageous for understanding any hidden structures in your data, but it has a major pitfall. In the version shown above, we assume that every data point is relevant — which is almost never the case in the real world.

Density-based clustering methods provide a safety valve. Instead of assuming that every point is part of some cluster, we only look at points that are tightly packed and assume everything else is noise.

This approach requires two parameters: a radius 𝜖 and a neighborhood density 𝛳. For every point, we calculate Neps(x) — the number of points that are at most 𝜖 away from x. If Neps(x) ≥ 𝛳, not counting x, then x is considered a *core point*. If a point isn’t a core point, but it is a member of a core point’s neighborhood, then it is considered a *border point*. Everything else is just considered *noise.*

One of the most common and, indeed, performative implementations of density-based clustering is Density-based Spatial Clustering of Applications with Noise, better known as DBSCAN. DBSCAN works by running a connected components algorithm across the different core points. If two core points share border points, or a core point is a border point in another core point’s neighborhood, then they’re part of the same connected component, which forms a cluster.

Say we have a relatively small 𝜖 and a decently large 𝛳. We might wind up with clusters like this:

A picture containing design

Description automatically generated with low confidence

See? Those two lonely points are very far away from the two clusters, and there are too few for them to really mean anything — so they’re just noise.

Notice that we’re also able to discover non-convex clusters this way (see the orange arc). Pretty neat!

We don’t need to specify a number of clusters we’re interested in for density-based clustering to work — it will automatically discover some number of clusters based on your 𝜖 and 𝛳. This is especially useful when you expect all of your clusters to have a similar density.

**K-Means Clustering**

Hierarchical clustering excels at discovering embedded structures in the data, and density-based approaches excel at finding an unknown number of clusters of similar density. However, both fail at finding a ‘consensus’ across the full dataset. Hierarchical clustering can put together clusters that seem close, but no information about other points is considered. Density-based methods only look at a small neighborhood of nearby points and similarly fail to consider the full dataset.

That’s where K-means clustering comes in. In a sense, K-means considers every point in the dataset and uses that information to evolve the clustering over a series of iterations.

K-means works by selecting *k* central points, or *means*, hence K-Means. These means are then used as the centroid of their cluster: any point that is closest to a given mean is assigned to that mean’s cluster.

Once all points are assigned, move through each cluster and take the average of all points it contains. This new ‘average’ point is the new mean of the cluster.

Just repeat these two steps over and over again until the point assignments stop changing!

A picture containing diagram, sketch, drawing, design

Description automatically generated

Once the point assignments have stopped changing, the algorithm is said to have converged.

We will now have *k* different clusters, each of which has a centroid closer to every point in its cluster than any other centroid. Calculating the centroid again won’t change the assignments, so we stop. That’s really all there is to K-means, but it’s a very powerful method for finding a known number of clusters while considering the entire dataset.

There are numerous approaches to initializing your means. The Forgy Method randomly selects *k* random observations from the data and uses these as the starting points. The Random Partition Method will assign every point in the dataset to a random cluster, then calculate the centroid from these and resume the algorithm.

While K-means is an NP-hard problem, heuristic methods are capable of finding decent approximations to the global optimum in polynomial time and are able to handle big datasets efficiently, making it a solid choice over hierarchical clustering in some cases.

Clustering is a strange world, with an even stranger collection of techniques. These three approaches are only some of the most popular, but they will get you a long way in discovering unknown groupings in your data. Clustering is very useful in exploratory data analysis, finding initialization points for other analyses, and is also incredibly simple to deploy. Used wisely, clustering can provide surprising insights into your data. Consider it another notch in your belt.

## 1. Cluster labels in hierarchical clustering

00:00 - 00:05

In the previous video, we employed hierarchical clustering

## 2. Cluster labels in hierarchical clustering

00:05 - 00:28

to create a great visualization of the voting behavior at the Eurovision. But hierarchical clustering is not only a visualization tool. In this video, you'll learn how to extract the clusters from intermediate stages of a hierarchical clustering. The cluster labels for these intermediate clusterings can then be used in further computations, such as cross tabulations, just like the cluster labels from k-means.

## 3. Intermediate clusterings & height on dendrogram

00:28 - 00:49

An intermediate stage in the hierarchical clustering is specified by choosing a height on the dendrogram. For example, choosing a height of 15 defines a clustering in which Bulgaria, Cyprus and Greece are in one cluster, Russia and Moldova are in another, and Armenia is in a cluster on its own. But what is the meaning of the height?

## 4. Dendrograms show cluster distances

00:49 - 01:04

The y-axis of the dendrogram encodes the distance between merging clusters. For example, the distance between the cluster containing Cyprus and the cluster containing Greece was approximately 6 when they were merged into a single cluster.

## 5. Dendrograms show cluster distances

01:04 - 01:11

When this new cluster was merged with the cluster containing Bulgaria, the distance between them was 12.

## 6. Intermediate clusterings & height on dendrogram

01:11 - 01:24

So the height that specifies an intermediate clustering corresponds to a distance. This specifies that the hierarchical clustering should stop merging clusters when all clusters are at least this far apart.

## 7. Distance between clusters

01:24 - 01:50

The distance between two clusters is measured using a "linkage method". In our example, we used "complete" linkage, where the distance between two clusters is the maximum of the distances between their samples. This was specified via the "method" parameter. There are many other linkage methods, and you'll see in the exercises that different linkage methods give different hierarchical clusterings!

## 8. Extracting cluster labels

01:50 - 02:02

The cluster labels for any intermediate stage of the hierarchical clustering can be extracted using the fcluster function. Let's try it out, specifying the height of 15.

## 9. Extracting cluster labels using fcluster

02:02 - 02:22

After performing the hierarchical clustering of the Eurovision data, import the fcluster function. Then pass the result of the linkage function to the fcluster function, specifying the height as the second argument. This returns a numpy array containing the cluster labels for all the countries.

## 10. Aligning cluster labels with country names

02:22 - 02:52

To inspect cluster labels, let's use a DataFrame to align the labels with the country names. Firstly, import pandas, then create the data frame, and then sort by cluster label, printing the result. As expected, the cluster labels group Bulgaria, Greece and Cyprus in the same cluster. But do note that the scipy cluster labels start at 1, not at 0 like they do in scikit-learn.

## 11. Let's practice!

02:52 - 03:03

Now that you've learned how to extract cluster labels from a hierarchical clustering, let's put your new skills into practice!

# Which clusters are closest?

In the video, you learned that the linkage method defines how the distance between clusters is measured. In complete linkage, the distance between clusters is the distance between the furthest points of the clusters. In single linkage, the distance between clusters is the distance between the closest points of the clusters.

Consider the three clusters in the diagram. Which of the following statements are true?

A picture containing text, screenshot, diagram, circle

Description automatically generated

**A.** In single linkage, Cluster 3 is the closest cluster to Cluster 2.

**B.** In complete linkage, Cluster 1 is the closest cluster to Cluster 2.

##### Answer the question

**50XP**

#### Possible Answers

* Neither A nor B.
* A only
* Both A and B.

# Different linkage, different hierarchical clustering!

In the video, you saw a hierarchical clustering of the voting countries at the Eurovision song contest using 'complete' linkage. Now, perform a hierarchical clustering of the voting countries with 'single' linkage, and compare the resulting dendrogram with the one in the video. Different linkage, different hierarchical clustering!

You are given an array samples. Each row corresponds to a voting country, and each column corresponds to a performance that was voted for. The list country\_names gives the name of each voting country. This dataset was obtained from [Eurovision](https://www.eurovision.tv/page/results).

## Instructions

100 XP

* Import linkage and dendrogram from scipy.cluster.hierarchy.
* Perform hierarchical clustering on samples using the linkage() function with the method='single' keyword argument. Assign the result to mergings.
* Plot a dendrogram of the hierarchical clustering, using the list country\_names as the labels. In addition, specify the leaf\_rotation=90, and leaf\_font\_size=6 keyword arguments as you have done earlier.
* # Perform the necessary imports
* import matplotlib.pyplot as plt
* from \_\_\_\_ import \_\_\_\_, \_\_\_\_
* # Calculate the linkage: mergings
* mergings = \_\_\_\_
* # Plot the dendrogram
* \_\_\_\_
* plt.show()

# Perform the necessary imports

import matplotlib.pyplot as plt

from scipy.cluster.hierarchy import linkage, dendrogram

# Calculate the linkage: mergings

mergings = linkage(samples, method='single')

# Plot the dendrogram

dendrogram(mergings, labels= country\_names, leaf\_rotation=90, leaf\_font\_size=6)

plt.show()

# Perform the necessary imports import matplotlib.pyplot as plt from scipy.cluster.hierarchy import linkage, dendrogram # Calculate the linkage: mergings mergings = linkage(samples, method='single') # Plot the dendrogram dendrogram(mergings, labels= country\_names, leaf\_rotation=90, leaf\_font\_size=6) plt.show()

Great work! As you can see, performing single linkage hierarchical clustering produces a different dendrogram!

# Intermediate clusterings

Displayed on the right is the dendrogram for the hierarchical clustering of the grain samples that you computed earlier. If the hierarchical clustering were stopped at height 6 on the dendrogram, how many clusters would there be?

## Instructions

50 XP

### Possible answers

1

3

as many as there were at the beginning

# Extracting the cluster labels

In the previous exercise, you saw that the intermediate clustering of the grain samples at height 6 has 3 clusters. Now, use the fcluster() function to extract the cluster labels for this intermediate clustering, and compare the labels with the grain varieties using a cross-tabulation.

The hierarchical clustering has already been performed and mergings is the result of the linkage() function. The list varieties gives the variety of each grain sample.

## Instructions

100 XP

* Import:
  + pandas as pd.
  + fcluster from scipy.cluster.hierarchy.
* Perform a flat hierarchical clustering by using the fcluster() function on mergings. Specify a maximum height of 6 and the keyword argument criterion='distance'.
* Create a DataFrame df with two columns named 'labels' and 'varieties', using labels and varieties, respectively, for the column values. This has been done for you.
* Create a cross-tabulation ct between df['labels'] and df['varieties'] to count the number of times each grain variety coincides with each cluster label.

# Perform the necessary imports

import \_\_\_\_ as \_\_\_\_

from \_\_\_\_ import \_\_\_\_

# Use fcluster to extract labels: labels

labels = \_\_\_\_

# Create a DataFrame with labels and varieties as columns: df

df = pd.DataFrame({'labels': labels, 'varieties': varieties})

# Create crosstab: ct

ct = \_\_\_\_

# Display ct

print(ct)

# Perform the necessary imports

import pandas as pd

from scipy.cluster.hierarchy import fcluster

# Use fcluster to extract labels: labels

labels = fcluster(mergings, 6, criterion='distance')

# Create a DataFrame with labels and varieties as columns: df

df = pd.DataFrame({'labels': labels, 'varieties': varieties})

# Create crosstab: ct

ct = pd.crosstab(df['labels'], df['varieties'])

# Display ct

print(ct)

# Perform the necessary imports

import pandas as pd

from scipy.cluster.hierarchy import fcluster

# Use fcluster to extract labels: labels

labels = fcluster(mergings, 6, criterion='distance')

# Create a DataFrame with labels and varieties as columns: df

df = pd.DataFrame({'labels': labels, 'varieties': varieties})

# Create crosstab: ct

ct = pd.crosstab(df['labels'], df['varieties'])

# Display ct

print(ct)

<script.py> output:

varieties Canadian wheat Kama wheat Rosa wheat

labels

1 14 3 0

2 0 0 14

3 0 11 0

Fantastic - you've now mastered the fundamentals of k-Means and agglomerative hierarchical clustering. Next, you'll learn about t-SNE, which is a powerful tool for visualizing high dimensional data.

## 1. t-SNE for 2-dimensional maps

00:00 - 00:07

In this video, you'll learn about an unsupervised learning method for visualization called "t-SNE".

## 2. t-SNE for 2-dimensional maps

00:07 - 00:37

t-SNE stands for "t-distributed stochastic neighbor embedding". It has a complicated name, but it serves a very simple purpose. It maps samples from their high-dimensional space into a 2- or 3-dimensional space so they can visualized. While some distortion is inevitable, t-SNE does a great job of approximately representing the distances between the samples. For this reason, t-SNE is an invaluable visual aid for understanding a dataset.

## 3. t-SNE on the iris dataset

00:37 - 01:13

To see what sorts of insights are possible with t-SNE, let's look at how it performs on the iris dataset. The iris samples are in a four dimensional space, where each dimension corresponds to one of the four iris measurements, such as petal length and petal width. Now t-SNE was given only the measurements of the iris samples. In particular it wasn't given any information about the three species of iris. But if we color the species differently on the scatter plot, we see that t-SNE has kept the species separate.

## 4. Interpreting t-SNE scatter plots

01:13 - 01:44

This scatter plot gives us a new insight, however. We learn that there are two iris species, versicolor and virginica, whose samples are close together in space. So it could happen that the iris dataset appears to have two clusters, instead of three. This is compatible with our previous examples using k-means, where we saw that a clustering with 2 clusters also had relatively low inertia, meaning tight clusters.

## 5. t-SNE in sklearn

01:44 - 02:02

t-SNE is available in scikit-learn, but it works a little differently to the fit/transform components you've already met. Let's see it in action on the iris dataset. The samples are in a 2-dimensional numpy array, and there is a list giving the species of each sample.

## 6. t-SNE in sklearn

02:02 - 02:22

To start with, import TSNE and create a TSNE object. Apply the fit\_transform method to the samples, and then make a scatter plot of the result, coloring the points using the species. There are two aspects that deserve special attention: the fit\_transform method, and the learning rate.

## 7. t-SNE has only fit\_transform()

02:22 - 02:46

t-SNE only has a fit\_transform method. As you might expect, the fit\_transform method simultaneously fits the model and transforms the data. However, t-SNE does not have separate fit and transform methods. This means that you can't extend a t-SNE map to include new samples. Instead, you have to start over each time.

## 8. t-SNE learning rate

02:46 - 03:10

The second thing to notice is the learning rate. The learning rate makes the use of t-SNE more complicated than some other techniques. You may need to try different learning rates for different datasets. It is clear, however, when you've made a bad choice, because all the samples appear bunched together in the scatter plot. Normally it's enough to try a few values between 50 and 200.

## 9. Different every time

03:10 - 03:42

A final thing to be aware of is that the axes of a t-SNE plot do not have any interpretable meaning. In fact, they are different every time t-SNE is applied, even on the same data. For example, here are three t-SNE plots of the scaled Piedmont wine samples, generated using the same code. Note that while the orientation of the plot is different each time, the three wine varieties, represented here using colors, have the same position relative to one another.

## 10. Let's practice!

03:42 - 03:52

You are now equipped to use t-SNE to gain insight into some real-world datasets. Let's get some practice!

# t-SNE visualization of grain dataset

In the video, you saw t-SNE applied to the iris dataset. In this exercise, you'll apply t-SNE to the grain samples data and inspect the resulting t-SNE features using a scatter plot. You are given an array samples of grain samples and a list variety\_numbers giving the variety number of each grain sample.

## Instructions

100 XP

* Import TSNE from sklearn.manifold.
* Create a TSNE instance called model with learning\_rate=200.
* Apply the .fit\_transform() method of model to samples. Assign the result to tsne\_features.
* Select the column 0 of tsne\_features. Assign the result to xs.
* Select the column 1 of tsne\_features. Assign the result to ys.
* Make a scatter plot of the t-SNE features xs and ys. To color the points by the grain variety, specify the additional keyword argument c=variety\_numbers.
* # Import TSNE
* \_\_\_\_
* # Create a TSNE instance: model
* model = \_\_\_\_
* # Apply fit\_transform to samples: tsne\_features
* tsne\_features = \_\_\_\_
* # Select the 0th feature: xs
* xs = tsne\_features[:,0]
* # Select the 1st feature: ys
* ys = tsne\_features[:,1]
* # Scatter plot, coloring by variety\_numbers
* \_\_\_\_
* plt.show()

# Import TSNE

from sklearn.manifold import TSNE

# Create a TSNE instance: model

model = TSNE(learning\_rate=200)

# Apply fit\_transform to samples: tsne\_features

tsne\_features = model.fit\_transform(samples)

# Select the 0th feature: xs

xs = tsne\_features[:,0]

# Select the 1st feature: ys

ys = tsne\_features[:,1]

# Scatter plot, coloring by variety\_numbers

plt.scatter(xs, ys, c=variety\_numbers)

plt.show()

# Import TSNE from sklearn.manifold import TSNE # Create a TSNE instance: model model = TSNE(learning\_rate=200) # Apply fit\_transform to samples: tsne\_features tsne\_features = model.fit\_transform(samples) # Select the 0th feature: xs xs = tsne\_features[:,0] # Select the 1st feature: ys ys = tsne\_features[:,1] # Scatter plot, coloring by variety\_numbers plt.scatter(xs, ys, c=variety\_numbers) plt.show()

Excellent! As you can see, the t-SNE visualization manages to separate the 3 varieties of grain samples. But how will it perform on the stock data? You'll find out in the next exercise!

# A t-SNE map of the stock market

t-SNE provides great visualizations when the individual samples can be labeled. In this exercise, you'll apply t-SNE to the company stock price data. A scatter plot of the resulting t-SNE features, labeled by the company names, gives you a map of the stock market! The stock price movements for each company are available as the array normalized\_movements (these have already been normalized for you). The list companies gives the name of each company. PyPlot (plt) has been imported for you.

## Instructions

100 XP

* Import TSNE from sklearn.manifold.
* Create a TSNE instance called model with learning\_rate=50.
* Apply the .fit\_transform() method of model to normalized\_movements. Assign the result to tsne\_features.
* Select column 0 and column 1 of tsne\_features.
* Make a scatter plot of the t-SNE features xs and ys. Specify the additional keyword argument alpha=0.5.
* Code to label each point with its company name has been written for you using plt.annotate(), so just hit submit to see the visualization!
* # Import TSNE
* \_\_\_\_
* # Create a TSNE instance: model
* model = \_\_\_\_
* # Apply fit\_transform to normalized\_movements: tsne\_features
* tsne\_features = \_\_\_\_
* # Select the 0th feature: xs
* xs = \_\_\_\_
* # Select the 1th feature: ys
* ys = tsne\_features[:,1]
* # Scatter plot
* \_\_\_\_
* # Annotate the points
* for x, y, company in zip(xs, ys, companies):
* plt.annotate(company, (x, y), fontsize=5, alpha=0.75)
* plt.show()

# Import TSNE

from sklearn.manifold import TSNE

# Create a TSNE instance: model

model = TSNE(learning\_rate=50)

# Apply fit\_transform to normalized\_movements: tsne\_features

tsne\_features = model.fit\_transform(normalized\_movements)

# Select the 0th feature: xs

xs = tsne\_features[:,0]

# Select the 1th feature: ys

ys = tsne\_features[:,1]

# Scatter plot

plt.scatter(xs, ys, alpha=0.5)

# Annotate the points

for x, y, company in zip(xs, ys, companies):

    plt.annotate(company, (x, y), fontsize=5, alpha=0.75)

plt.show()

# Import TSNE from sklearn.manifold import TSNE # Create a TSNE instance: model model = TSNE(learning\_rate=50) # Apply fit\_transform to normalized\_movements: tsne\_features tsne\_features = model.fit\_transform(normalized\_movements) # Select the 0th feature: xs xs = tsne\_features[:,0] # Select the 1th feature: ys ys = tsne\_features[:,1] # Scatter plot plt.scatter(xs, ys, alpha=0.5) # Annotate the points for x, y, company in zip(xs, ys, companies): plt.annotate(company, (x, y), fontsize=5, alpha=0.75) plt.show()

Fantastic! It's visualizations such as this that make t-SNE such a powerful tool for extracting quick insights from high dimensional data.

## 1. Visualizing the PCA transformation

00:00 - 00:05

In the next two chapters you'll learn techniques for dimension reduction.

## 2. Dimension reduction

00:05 - 00:39

Dimension reduction finds patterns in data, and uses these patterns to re-express it in a compressed form. This makes subsequent computation with the data much more efficient, and this can be a big deal in a world of big datasets. However, the most important function of dimension reduction is to reduce a dataset to its "bare bones", discarding noisy features that cause big problems for supervised learning tasks like regression and classification. In many real-world applications, it's dimension reduction that makes prediction possible.

## 3. Principal Component Analysis

00:39 - 01:02

In this chapter, you'll learn about the most fundamental of dimension reduction techniques. It's called "Principal Component Analysis", or "PCA" for short. PCA performs dimension reduction in two steps, and the first one, called "de-correlation", doesn't change the dimension of the data at all. It's this first step that we'll focus on in this video.

## 4. PCA aligns data with axes

01:02 - 01:32

In this first step, PCA rotates the samples so that they are aligned with the coordinate axes. In fact, it does more than this: PCA also shifts the samples so that they have mean zero. These scatter plots show the effect of PCA applied to two features of the wine dataset. Notice that no information is lost - this is true no matter how many features your dataset has. You'll practice visualizing this transformation in the exercises.

## 5. PCA follows the fit/transform pattern

01:32 - 01:59

scikit-learn has an implementation of PCA, and it has fit and transform methods just like StandardScaler. The fit method learns how to shift and how to rotate the samples, but doesn't actually change them. The transform method, on the other hand, applies the transformation that fit learned. In particular, the transform method can be applied to new, unseen samples.

## 6. Using scikit-learn PCA

01:59 - 02:19

Let's see PCA in action on the some features of the wine dataset. Firstly, import PCA. Now create a PCA object, and fit it to the samples. Then use the fit PCA object to transform the samples. This returns a new array of transformed samples.

## 7. PCA features

02:19 - 02:40

This new array has the same number of rows and columns as the original sample array. In particular, there is one row for each transformed sample. The columns of the new array correspond to "PCA features", just as the original features corresponded to columns of the original array.

## 8. PCA features are not correlated

02:40 - 03:01

It is often the case that the features of a dataset are correlated. This is the case with many of the features of the wine dataset, for instance. However, PCA, due to the rotation it performs, "de-correlates" the data, in the sense that the columns of the transformed array are not linearly correlated.

## 9. Pearson correlation

03:01 - 03:20

Linear correlation can be measured with the Pearson correlation. It takes values between -1 and 1, where larger values indicate a stronger correlation, and 0 indicates no linear correlation. Here are some examples of features with varying degrees of correlation.

## 10. Principal components

03:20 - 03:41

Finally, PCA is called "principal component analysis" because it learns the "principal components" of the data. These are the directions in which the samples vary the most, depicted here in red. It is the principal components that PCA aligns with the coordinate axes.

## 11. Principal components

03:41 - 03:53

After a PCA model has been fit, the principal components are available as the components attribute. This is numpy array with one row for each principal component.

## 12. Let's practice!

03:53 - 04:01

You've met several new ideas in this video. Let's put them into practice!

# Correlated data in nature

You are given an array grains giving the width and length of samples of grain. You suspect that width and length will be correlated. To confirm this, make a scatter plot of width vs length and measure their Pearson correlation.

## Instructions

100 XP

* Import:
  + matplotlib.pyplot as plt.
  + pearsonr from scipy.stats.
* Assign column 0 of grains to width and column 1 of grains to length.
* Make a scatter plot with width on the x-axis and length on the y-axis.
* Use the pearsonr() function to calculate the Pearson correlation of width and length.
* # Perform the necessary imports
* \_\_\_\_
* \_\_\_\_
* # Assign the 0th column of grains: width
* width = \_\_\_\_
* # Assign the 1st column of grains: length
* length = \_\_\_\_
* # Scatter plot width vs length
* plt.scatter(\_\_\_\_, \_\_\_\_)
* plt.axis('equal')
* plt.show()
* # Calculate the Pearson correlation
* correlation, pvalue = \_\_\_\_
* # Display the correlation
* print(correlation)

# Perform the necessary imports

import matplotlib.pyplot as plt

from scipy.stats import pearsonr

# Assign the 0th column of grains: width

width = grains[:,0]

# Assign the 1st column of grains: length

length = grains[:,1]

# Scatter plot width vs length

plt.scatter(width, length)

plt.axis('equal')

plt.show()

# Calculate the Pearson correlation

correlation, pvalue = pearsonr(width, length)

# Display the correlation

print(correlation)

# Perform the necessary imports

import matplotlib.pyplot as plt

from scipy.stats import pearsonr

# Assign the 0th column of grains: width

width = grains[:,0]

# Assign the 1st column of grains: length

length = grains[:,1]

# Scatter plot width vs length

plt.scatter(width, length)

plt.axis('equal')

plt.show()

# Calculate the Pearson correlation

correlation, pvalue = pearsonr(width, length)

# Display the correlation

print(correlation)

0.8604149377143469

<script.py> output:

0.8604149377143469

Great work! As you would expect, the width and length of the grain samples are highly correlated.

Daily XP400

## Exercise

## Exercise

# Decorrelating the grain measurements with PCA

You observed in the previous exercise that the width and length measurements of the grain are correlated. Now, you'll use PCA to decorrelate these measurements, then plot the decorrelated points and measure their Pearson correlation.

## Instructions

100 XP

* Import PCA from sklearn.decomposition.
* Create an instance of PCA called model.
* Use the .fit\_transform() method of model to apply the PCA transformation to grains. Assign the result to pca\_features.
* The subsequent code to extract, plot, and compute the Pearson correlation of the first two columns pca\_features has been written for you, so hit submit to see the result!
* # Import PCA
* \_\_\_\_
* # Create PCA instance: model
* model = \_\_\_\_
* # Apply the fit\_transform method of model to grains: pca\_features
* pca\_features = \_\_\_\_
* # Assign 0th column of pca\_features: xs
* xs = pca\_features[:,0]
* # Assign 1st column of pca\_features: ys
* ys = pca\_features[:,1]
* # Scatter plot xs vs ys
* plt.scatter(xs, ys)
* plt.axis('equal')
* plt.show()
* # Calculate the Pearson correlation of xs and ys
* correlation, pvalue = pearsonr(xs, ys)
* # Display the correlation
* print(correlation)

# Import PCA

from sklearn.decomposition import PCA

# Create PCA instance: model

model = PCA()

# Apply the fit\_transform method of model to grains: pca\_features

pca\_features = model.fit\_transform(grains)

# Assign 0th column of pca\_features: xs

xs = pca\_features[:,0]

# Assign 1st column of pca\_features: ys

ys = pca\_features[:,1]

# Scatter plot xs vs ys

plt.scatter(xs, ys)

plt.axis('equal')

plt.show()

# Calculate the Pearson correlation of xs and ys

correlation, pvalue = pearsonr(xs, ys)

# Display the correlation

print(correlation)

# Import PCA

from sklearn.decomposition import PCA

# Create PCA instance: model

model = PCA()

# Apply the fit\_transform method of model to grains: pca\_features

pca\_features = model.fit\_transform(grains)

# Assign 0th column of pca\_features: xs

xs = pca\_features[:,0]

# Assign 1st column of pca\_features: ys

ys = pca\_features[:,1]

# Scatter plot xs vs ys

plt.scatter(xs, ys)

plt.axis('equal')

plt.show()

# Calculate the Pearson correlation of xs and ys

correlation, pvalue = pearsonr(xs, ys)

# Display the correlation

print(correlation)

5.4909917646575975e-17

<script.py> output:

5.4909917646575975e-17

Excellent! You've successfully decorrelated the grain measurements with PCA!

# Principal components

On the right are three scatter plots of the same point cloud. Each scatter plot shows a different set of axes (in red). In which of the plots could the axes represent the principal components of the point cloud?

Recall that the principal components are the directions along which the the data varies.

## Instructions

50 XP

### Possible answers

None of them

both plot1 and plot3

plot2

**Well done! You've correctly inferred that the principal components have to align with the axes of the point cloud. This happens in both plot 1 and plot 3.**

## 1. Intrinsic dimension

00:00 - 00:01

## 2. Intrinsic dimension of a flight path

00:01 - 00:27

Consider this dataset with 2 features: latitude and longitude. These two features might track the flight of an airplane, for example. This dataset is 2-dimensional, yet it turns out that it can be closely approximated using only one feature: the displacement along the flight path. This dataset is intrinsically one-dimensional.

## 3. Intrinsic dimension

00:27 - 00:51

The intrinsic dimension of a dataset is the number of features required to approximate it. The intrinsic dimension informs dimension reduction, because it tells us how much a dataset can be compressed. In this video, you'll gain a solid understanding of the intrinsic dimension, and be able to use PCA to identify it in real-world datasets that have thousands of features.

## 4. Versicolor dataset

00:51 - 01:15

To better illustrate the intrinsic dimension, let's consider an example dataset containing only some of the samples from the iris dataset. Specifically, let's take three measurements from the iris versicolor samples: sepal length, sepal width, and petal width. So each sample is represented as a point in 3-dimensional space.

## 5. Versicolor dataset has intrinsic dimension 2

01:15 - 01:35

However, if we make a 3d scatter plot of the samples, we see that they all lie very close to a flat, 2-dimensional sheet. This means that the data can be approximated by using only two coordinates, without losing much information. So this dataset has intrinsic dimension 2.

## 6. PCA identifies intrinsic dimension

01:35 - 02:04

But scatter plots are only possible if there are 3 features or less. So how can the intrinsic dimension be identified, even if there are many features? This is where PCA is really helpful. The intrinsic dimension can be identified by counting the PCA features that have high variance. To see how, let's see what happens when PCA is applied to the dataset of versicolor samples.

## 7. PCA of the versicolor samples

02:04 - 02:16

PCA rotates and shifts the samples to align them with the coordinate axes. This expresses the samples using three PCA features.

## 8. PCA features are ordered by variance descending

02:16 - 02:47

The PCA features are in a special order. Here is a bar graph showing the variance of each of the PCA features. As you can see, each PCA feature has less variance than the last, and in this case the last PCA feature has very low variance. This agrees with the scatter plot of the PCA features, where the samples don't vary much in the vertical direction. In the other two directions, however, the variance is apparent.

## 9. Variance and intrinsic dimension

02:47 - 03:07

The intrinsic dimension is the number of PCA features that have significant variance. In our example, only the first two PCA features have significant variance. So this dataset has intrinsic dimension 2, which agrees with what we observed when inspecting the scatter plot.

## 10. Plotting the variances of PCA features

03:07 - 03:24

Let's see how to plot the variances of the PCA features in practice. Firstly, make the necessary imports. Then create a PCA model, and fit it to the samples. Now create a range enumerating the PCA features,

## 11. Plotting the variances of PCA features

03:24 - 03:31

and make a bar plot of the variances; the variances are available as the explained\_variance attribute of the PCA model.

## 12. Intrinsic dimension can be ambiguous

03:31 - 03:54

The intrinsic dimension is a useful idea that helps to guide dimension reduction. However, it is not always unambiguous. Here is a graph of the variances of the PCA features for the wine dataset. We could argue for an intrinsic dimension of 2, of 3, or even more, depending upon the threshold you chose.

## 13. Let's practice!

03:54 - 04:08

In the next video, you'll learn to use the intrinsic dimension for dimension reduction. But for now, let's get some practice discovering the intrinsic dimension of some real-world datasets!

Daily XP600

## Exercise

## Exercise

# The first principal component

The first principal component of the data is the direction in which the data varies the most. In this exercise, your job is to use PCA to find the first principal component of the length and width measurements of the grain samples, and represent it as an arrow on the scatter plot.

The array grains gives the length and width of the grain samples. PyPlot (plt) and PCA have already been imported for you.

## Instructions

100 XP

* Make a scatter plot of the grain measurements. This has been done for you.
* Create a PCA instance called model.
* Fit the model to the grains data.
* Extract the coordinates of the mean of the data using the .mean\_ attribute of model.
* Get the first principal component of model using the .components\_[0,:] attribute.
* Plot the first principal component as an arrow on the scatter plot, using the plt.arrow() function. You have to specify the first two arguments - mean[0] and mean[1].

Daily XP600

## Exercise

## Exercise

# The first principal component

The first principal component of the data is the direction in which the data varies the most. In this exercise, your job is to use PCA to find the first principal component of the length and width measurements of the grain samples, and represent it as an arrow on the scatter plot.

The array grains gives the length and width of the grain samples. PyPlot (plt) and PCA have already been imported for you.

## Instructions

100 XP

* Make a scatter plot of the grain measurements. This has been done for you.
* Create a PCA instance called model.
* Fit the model to the grains data.
* Extract the coordinates of the mean of the data using the .mean\_ attribute of model.
* Get the first principal component of model using the .components\_[0,:] attribute.
* Plot the first principal component as an arrow on the scatter plot, using the plt.arrow() function. You have to specify the first two arguments - mean[0] and mean[1].

# Make a scatter plot of the untransformed points

plt.scatter(grains[:,0], grains[:,1])

# Create a PCA instance: model

model = \_\_\_\_

# Fit model to points

\_\_\_\_

# Get the mean of the grain samples: mean

mean = \_\_\_\_

# Get the first principal component: first\_pc

first\_pc = \_\_\_\_

# Plot first\_pc as an arrow, starting at mean

plt.arrow(\_\_\_\_, \_\_\_\_, first\_pc[0], first\_pc[1], color='red', width=0.01)

# Keep axes on same scale

plt.axis('equal')

plt.show()

# Make a scatter plot of the untransformed points

plt.scatter(grains[:,0], grains[:,1])

# Create a PCA instance: model

model = PCA()

# Fit model to points

model.fit(grains)

# Get the mean of the grain samples: mean

mean = model.mean\_

# Get the first principal component: first\_pc

first\_pc = model.components\_[0,:]

# Plot first\_pc as an arrow, starting at mean

plt.arrow(mean[0], mean[1], first\_pc[0], first\_pc[1], color='red', width=0.01)

# Keep axes on same scale

plt.axis('equal')

plt.show()

# Make a scatter plot of the untransformed points plt.scatter(grains[:,0], grains[:,1]) # Create a PCA instance: model model = PCA() # Fit model to points model.fit(grains) # Get the mean of the grain samples: mean mean = model.mean\_ # Get the first principal component: first\_pc first\_pc = model.components\_[0,:] # Plot first\_pc as an arrow, starting at mean plt.arrow(mean[0], mean[1], first\_pc[0], first\_pc[1], color='red', width=0.01) # Keep axes on same scale plt.axis('equal') plt.show()

Excellent job! This is the direction in which the grain data varies the most.

Daily XP700

## Exercise

## Exercise

# Variance of the PCA features

The fish dataset is 6-dimensional. But what is its intrinsic dimension? Make a plot of the variances of the PCA features to find out. As before, samples is a 2D array, where each row represents a fish. You'll need to standardize the features first.

## Instructions

100 XP

* Create an instance of StandardScaler called scaler.
* Create a PCA instance called pca.
* Use the make\_pipeline() function to create a pipeline chaining scaler and pca.
* Use the .fit() method of pipeline to fit it to the fish samples samples.
* Extract the number of components used using the .n\_components\_ attribute of pca. Place this inside a range() function and store the result as features.
* Use the plt.bar() function to plot the explained variances, with features on the x-axis and pca.explained\_variance\_ on the y-axis.
* # Perform the necessary imports
* from sklearn.decomposition import PCA
* from sklearn.preprocessing import StandardScaler
* from sklearn.pipeline import make\_pipeline
* import matplotlib.pyplot as plt
* # Create scaler: scaler
* scaler = \_\_\_\_
* # Create a PCA instance: pca
* pca = \_\_\_\_
* # Create pipeline: pipeline
* pipeline = \_\_\_\_
* # Fit the pipeline to 'samples'
* \_\_\_\_
* # Plot the explained variances
* features = \_\_\_\_
* plt.bar(\_\_\_\_, \_\_\_\_)
* plt.xlabel('PCA feature')
* plt.ylabel('variance')
* plt.xticks(features)
* plt.show()

# Perform the necessary imports

from sklearn.decomposition import PCA

from sklearn.preprocessing import StandardScaler

from sklearn.pipeline import make\_pipeline

import matplotlib.pyplot as plt

# Create scaler: scaler

scaler = StandardScaler()

# Create a PCA instance: pca

pca = PCA()

# Create pipeline: pipeline

pipeline = make\_pipeline(scaler, pca)

# Fit the pipeline to 'samples'

pipeline.fit(samples)

# Plot the explained variances

features = range(pca.n\_components\_)

plt.bar(features, pca.explained\_variance\_)

plt.xlabel('PCA feature')

plt.ylabel('variance')

plt.xticks(features)

plt.show()

# Perform the necessary imports from sklearn.decomposition import PCA from sklearn.preprocessing import StandardScaler from sklearn.pipeline import make\_pipeline import matplotlib.pyplot as plt # Create scaler: scaler scaler = StandardScaler() # Create a PCA instance: pca pca = PCA() # Create pipeline: pipeline pipeline = make\_pipeline(scaler, pca) # Fit the pipeline to 'samples' pipeline.fit(samples) # Plot the explained variances features = range(pca.n\_components\_) plt.bar(features, pca.explained\_variance\_) plt.xlabel('PCA feature') plt.ylabel('variance') plt.xticks(features) plt.show()

Great work! It looks like PCA features 0 and 1 have significant variance.

# Intrinsic dimension of the fish data

In the previous exercise, you plotted the variance of the PCA features of the fish measurements. Looking again at your plot, what do you think would be a reasonable choice for the "intrinsic dimension" of the fish measurements? Recall that the intrinsic dimension is the number of PCA features with significant variance.

## Instructions

50 XP

### Possible answers

1

2

5

Great job! Since PCA features 0 and 1 have significant variance, the intrinsic dimension of this dataset appears to be 2.

## 1. Dimension reduction with PCA

00:00 - 00:02

## 2. Dimension reduction

00:02 - 00:16

Dimension reduction represents the same data using less features and is vital for building machine learning pipelines using real-world data. Finally, in this video, you'll learn how to perform dimension reduction using PCA.

## 3. Dimension reduction with PCA

00:16 - 00:33

We've seen already that the PCA features are in decreasing order of variance. PCA performs dimension reduction by discarding the PCA features with lower variance, which it assumes to be noise, and retaining the higher variance PCA features, which it assumes to be informative.

## 4. Dimension reduction with PCA

00:33 - 00:54

To use PCA for dimension reduction, you need to specify how many PCA features to keep. For example, specifying n\_components=2 when creating a PCA model tells it to keep only the first two PCA features. A good choice is the intrinsic dimension of the dataset, if you know it. Let's consider an example right away.

## 5. Dimension reduction of iris dataset

00:54 - 01:27

The iris dataset has 4 features representing the 4 measurements. Here, the measurements are in a numpy array called samples. Let's use PCA to reduce the dimension of the iris dataset to only 2. Begin by importing PCA as usual. Create a PCA model specifying n\_components=2, and then fit the model and transform the samples as usual. Printing the shape of the transformed samples, we see that there are only two features, as expected.

## 6. Iris dataset in 2 dimensions

01:27 - 01:55

Here is a scatterplot of the two PCA features, where the colors represent the three species of iris. Remarkably, despite having reduced the dimension from 4 to 2, the species can still be distinguished. Remember that PCA didn't even know that there were distinct species. PCA simply took the 2 PCA features with highest variance. As we can see, these two features are very informative.

## 7. Dimension reduction with PCA

01:55 - 02:12

PCA discards the low variance features, and assumes that the higher variance features are informative. Like all assumptions, there are cases where this doesn't hold. As we saw with the iris dataset, however, it often does in practice.

## 8. Word frequency arrays

02:12 - 02:44

In some cases, an alternative implementation of PCA needs to be used. Word frequency arrays are a great example. In a word-frequency array, each row corresponds to a document, and each column corresponds to a word from a fixed vocabulary. The entries of the word-frequency array measure how often each word appears in each document. Only some of the words from the vocabulary appear in any one document, so most entries of the word frequency array are zero.

## 9. Sparse arrays and csr\_matrix

02:44 - 02:58

Arrays like this are said to be "sparse", and are often represented using a special type of array called a "csr\_matrix". csr\_matrices save space by remembering only the non-zero entries of the array.

## 10. TruncatedSVD and csr\_matrix

02:58 - 03:20

Scikit-learn's PCA doesn't support csr\_matrices, and you'll need to use TruncatedSVD instead. TruncatedSVD performs the same transformation as PCA, but accepts csr matrices as input. Other than that, you interact with TruncatedSVD and PCA in exactly the same way.

## 11. Let's practice!

03:20 - 03:32

In this video, you've learned how to perform dimension reduction using PCA. Now it's time to practice dimension reduction in the wild!

# Dimension reduction of the fish measurements

In a previous exercise, you saw that 2 was a reasonable choice for the "intrinsic dimension" of the fish measurements. Now use PCA for dimensionality reduction of the fish measurements, retaining only the 2 most important components.

The fish measurements have already been scaled for you, and are available as scaled\_samples.

## Instructions

100 XP

* Import PCA from sklearn.decomposition.
* Create a PCA instance called pca with n\_components=2.
* Use the .fit() method of pca to fit it to the scaled fish measurements scaled\_samples.
* Use the .transform() method of pca to transform the scaled\_samples. Assign the result to pca\_features.
* # Import PCA
* \_\_\_\_
* # Create a PCA model with 2 components: pca
* pca = \_\_\_\_
* # Fit the PCA instance to the scaled samples
* \_\_\_\_
* # Transform the scaled samples: pca\_features
* pca\_features = \_\_\_\_
* # Print the shape of pca\_features
* print(pca\_features.shape)

# Import PCA

from sklearn.decomposition import PCA

# Create a PCA model with 2 components: pca

pca = PCA(n\_components=2)

# Fit the PCA instance to the scaled samples

pca.fit(scaled\_samples)

# Transform the scaled samples: pca\_features

pca\_features = pca.transform(scaled\_samples)

# Print the shape of pca\_features

print(pca\_features.shape)

(85, 2)

<script.py> output:

(85, 2)

Superb! You've successfully reduced the dimensionality from 6 to 2.

Daily XP300

## Exercise

## Exercise

# A tf-idf word-frequency array

In this exercise, you'll create a tf-idf word frequency array for a toy collection of documents. For this, use the TfidfVectorizer from sklearn. It transforms a list of documents into a word frequency array, which it outputs as a csr\_matrix. It has fit() and transform() methods like other sklearn objects.

You are given a list documents of toy documents about pets. Its contents have been printed in the IPython Shell.

## Instructions

100 XP

* Import TfidfVectorizer from sklearn.feature\_extraction.text.
* Create a TfidfVectorizer instance called tfidf.
* Apply .fit\_transform() method of tfidf to documents and assign the result to csr\_mat. This is a word-frequency array in csr\_matrix format.
* Inspect csr\_mat by calling its .toarray() method and printing the result. This has been done for you.
* The columns of the array correspond to words. Get the list of words by calling the .get\_feature\_names() method of tfidf, and assign the result to words.
* # Import TfidfVectorizer
* from sklearn.feature\_extraction.text import TfidfVectorizer
* # Create a TfidfVectorizer: tfidf
* tfidf = TfidfVectorizer()
* # Apply fit\_transform to document: csr\_mat
* csr\_mat = tfidf.fit\_transform(documents)
* # Print result of toarray() method
* print(csr\_mat.toarray())
* # Get the words: words
* words = tfidf.get\_feature\_names()
* # Print words
* print(words)

['cats say meow', 'dogs say woof', 'dogs chase cats']

# Import TfidfVectorizer

from sklearn.feature\_extraction.text import TfidfVectorizer

# Create a TfidfVectorizer: tfidf

tfidf = TfidfVectorizer()

# Apply fit\_transform to document: csr\_mat

csr\_mat = tfidf.fit\_transform(documents)

# Print result of toarray() method

print(csr\_mat.toarray())

# Get the words: words

words = tfidf.get\_feature\_names()

# Print words

print(words)

[[0.51785612 0. 0. 0.68091856 0.51785612 0. ]

[0. 0. 0.51785612 0. 0.51785612 0.68091856]

[0.51785612 0.68091856 0.51785612 0. 0. 0. ]]

['cats', 'chase', 'dogs', 'meow', 'say', 'woof']

<script.py> output:

[[0.51785612 0. 0. 0.68091856 0.51785612 0. ]

[0. 0. 0.51785612 0. 0.51785612 0.68091856]

[0.51785612 0.68091856 0.51785612 0. 0. 0. ]]

['cats', 'chase', 'dogs', 'meow', 'say', 'woof']

Great work! You'll now move to clustering Wikipedia articles!

# Clustering Wikipedia part I

You saw in the video that TruncatedSVD is able to perform PCA on sparse arrays in csr\_matrix format, such as word-frequency arrays. Combine your knowledge of TruncatedSVD and k-means to cluster some popular pages from Wikipedia. In this exercise, build the pipeline. In the next exercise, you'll apply it to the word-frequency array of some Wikipedia articles.

Create a Pipeline object consisting of a TruncatedSVD followed by KMeans. (This time, we've precomputed the word-frequency matrix for you, so there's no need for a TfidfVectorizer).

The Wikipedia dataset you will be working with was obtained from [here](https://blog.lateral.io/2015/06/the-unknown-perils-of-mining-wikipedia/).

## Instructions

100 XP

* Import:
  + TruncatedSVD from sklearn.decomposition.
  + KMeans from sklearn.cluster.
  + make\_pipeline from sklearn.pipeline.
* Create a TruncatedSVD instance called svd with n\_components=50.
* Create a KMeans instance called kmeans with n\_clusters=6.
* Create a pipeline called pipeline consisting of svd and kmeans.
* # Perform the necessary imports
* from \_\_\_\_ import \_\_\_\_
* from \_\_\_\_ import \_\_\_\_
* from \_\_\_\_ import \_\_\_\_
* # Create a TruncatedSVD instance: svd
* svd = \_\_\_\_
* # Create a KMeans instance: kmeans
* kmeans = \_\_\_\_
* # Create a pipeline: pipeline
* pipeline = \_\_\_\_

# Perform the necessary imports

from sklearn.decomposition import TruncatedSVD

from sklearn.cluster import KMeans

from sklearn.pipeline import make\_pipeline

# Create a TruncatedSVD instance: svd

svd = TruncatedSVD(n\_components=50)

# Create a KMeans instance: kmeans

kmeans = KMeans(n\_clusters=6)

# Create a pipeline: pipeline

pipeline = make\_pipeline(svd, kmeans)

# Perform the necessary imports from sklearn.decomposition import TruncatedSVD from sklearn.cluster import KMeans from sklearn.pipeline import make\_pipeline # Create a TruncatedSVD instance: svd svd = TruncatedSVD(n\_components=50) # Create a KMeans instance: kmeans kmeans = KMeans(n\_clusters=6) # Create a pipeline: pipeline pipeline = make\_pipeline(svd, kmeans)

Excellent! Now that you have set up your pipeline, you will use it in the next exercise to cluster the articles.

# Clustering Wikipedia part II

It is now time to put your pipeline from the previous exercise to work! You are given an array articles of tf-idf word-frequencies of some popular Wikipedia articles, and a list titles of their titles. Use your pipeline to cluster the Wikipedia articles.

A solution to the previous exercise has been pre-loaded for you, so a Pipeline pipeline chaining TruncatedSVD with KMeans is available.

## Instructions

100 XP

* Import pandas as pd.
* Fit the pipeline to the word-frequency array articles.
* Predict the cluster labels.
* Align the cluster labels with the list titles of article titles by creating a DataFrame df with labels and titles as columns. This has been done for you.
* Use the .sort\_values() method of df to sort the DataFrame by the 'label' column, and print the result.
* Hit submit and take a moment to investigate your amazing clustering of Wikipedia pages!
* # Import pandas
* \_\_\_\_
* # Fit the pipeline to articles
* \_\_\_\_
* # Calculate the cluster labels: labels
* labels = \_\_\_\_
* # Create a DataFrame aligning labels and titles: df
* df = pd.DataFrame({'label': labels, 'article': titles})
* # Display df sorted by cluster label
* print(\_\_\_\_)

# Import pandas

import pandas as pd

# Fit the pipeline to articles

pipeline.fit(articles)

# Calculate the cluster labels: labels

labels = pipeline.predict(articles)

# Create a DataFrame aligning labels and titles: df

df = pd.DataFrame({'label': labels, 'article': titles})

# Display df sorted by cluster label

print(df.sort\_values('label'))

**# Import pandas**

**import pandas as pd**

**# Fit the pipeline to articles**

**pipeline.fit(articles)**

**# Calculate the cluster labels: labels**

**labels = pipeline.predict(articles)**

**# Create a DataFrame aligning labels and titles: df**

**df = pd.DataFrame({'label': labels, 'article': titles})**

**# Display df sorted by cluster label**

**print(df.sort\_values('label'))**

**label article**

**59 0 Adam Levine**

**50 0 Chad Kroeger**

**51 0 Nate Ruess**

**52 0 The Wanted**

**53 0 Stevie Nicks**

**58 0 Sepsis**

**55 0 Black Sabbath**

**56 0 Skrillex**

**57 0 Red Hot Chili Peppers**

**54 0 Arctic Monkeys**

**21 1 Michael Fassbender**

**28 1 Anne Hathaway**

**27 1 Dakota Fanning**

**26 1 Mila Kunis**

**25 1 Russell Crowe**

**24 1 Jessica Biel**

**23 1 Catherine Zeta-Jones**

**22 1 Denzel Washington**

**20 1 Angelina Jolie**

**29 1 Jennifer Aniston**

**18 2 2010 United Nations Climate Change Conference**

**10 2 Global warming**

**11 2 Nationally Appropriate Mitigation Action**

**12 2 Nigel Lawson**

**13 2 Connie Hedegaard**

**19 2 2007 United Nations Climate Change Conference**

**15 2 Kyoto Protocol**

**16 2 350.org**

**17 2 Greenhouse gas emissions by the United States**

**14 2 Climate change**

**40 3 Tonsillitis**

**48 3 Gabapentin**

**47 3 Fever**

**46 3 Prednisone**

**45 3 Hepatitis C**

**44 3 Gout**

**43 3 Leukemia**

**42 3 Doxycycline**

**41 3 Hepatitis B**

**49 3 Lymphoma**

**39 4 Franck Ribéry**

**37 4 Football**

**36 4 2014 FIFA World Cup qualification**

**35 4 Colombia national football team**

**34 4 Zlatan Ibrahimović**

**33 4 Radamel Falcao**

**32 4 Arsenal F.C.**

**31 4 Cristiano Ronaldo**

**30 4 France national football team**

**38 4 Neymar**

**1 5 Alexa Internet**

**2 5 Internet Explorer**

**3 5 HTTP cookie**

**4 5 Google Search**

**8 5 Firefox**

**6 5 Hypertext Transfer Protocol**

**7 5 Social search**

**9 5 LinkedIn**

**5 5 Tumblr**

**0 5 HTTP 404**

Fantastic! Take a look at the cluster labels and see if you can identify any patterns!

## 1. Non-negative matrix factorization (NMF)

00:00 - 00:00

## 2. Non-negative matrix factorization

00:00 - 00:30

NMF stands for "non-negative matrix factorization". NMF, like PCA, is a dimension reduction technique. In constract to PCA, however, NMF models are interpretable. This means an NMF models are easier to understand yourself, and much easier for you to explain to others. NMF can not be applied to every dataset, however. It is required that the sample features be "non-negative", so greater than or equal to 0.

## 3. Interpretable parts

00:30 - 00:42

NMF achieves its interpretability by decomposing samples as sums of their parts. For example, NMF decomposes documents as combinations of common themes,

## 4. Interpretable parts

00:42 - 00:51

and images as combinations of common patterns. You'll learn about both these examples in detail later. For now, let's focus on getting started.

## 5. Using scikit-learn NMF

00:51 - 01:09

NMF is available in scikit learn, and follows the same fit/transform pattern as PCA. However, unlike PCA, the desired number of components must always be specified. NMF works both with numpy arrays and sparse arrays in the csr\_matrix format.

## 6. Example word-frequency array

01:09 - 01:50

Let's see an application of NMF to a toy example of a word-frequency array. In this toy dataset, there are only 4 words in the vocabulary, and these correspond to the four columns of the word-frequency array. Each row represents a document, and the entries of the array measure the frequency of each word in the document using what's known as "tf-idf". "tf" is the frequency of the word in the document. So if 10% of the words in the document are "datacamp", then the tf of "datacamp" for that document is point-1. "idf" is a weighting scheme that reduces the influence of frequent words like "the".

## 7. Example usage of NMF

01:50 - 02:10

Let's now see how to use NMF in Python. Firstly, import NMF. Create a model, specifying the desired number of components. Let's specify 2. Fit the model to the samples, then use the fit model to perform the transformation.

## 8. NMF components

02:10 - 02:37

Just as PCA has principal components, NMF has components which it learns from the samples, and as with PCA, the dimension of the components is the same as the dimension of the samples. In our example, for instance, there are 2 components, and they live in 4 dimensional space, corresponding to the 4 words in the vocabulary. The entries of the NMF components are always non-negative.

## 9. NMF features

02:37 - 02:57

The NMF feature values are non-negative, as well. As we saw with PCA, our transformed data in this example will have two columns, corresponding to our two new features. The features and the components of an NMF model can be combined to approximately reconstruct the original data samples.

## 10. Reconstruction of a sample

02:57 - 03:20

Let's see how this works with a single data sample. Here is a sample representing a document from our toy dataset, and here are its NMF feature values. Now if we multiply each NMF components by the corresponding NMF feature value, and add up each column, we get something very close to the original sample.

## 11. Sample reconstruction

03:20 - 03:43

So a sample can be reconstructed by multiplying the NMF components by the NMF feature values of the sample, and adding up. This calculation also can be expressed as what is known as a product of matrices. We won't be using that point of view, but that's where the "matrix factorization", or "MF", in NMF comes from.

## 12. NMF fits to non-negative data only

03:43 - 04:09

Finally, remember that NMF can only be applied to arrays of non-negative data, such as word-frequency arrays. In the next video, you'll construct another example by encoding collections of images as non-negative arrays. There are many other great examples as well, such as arrays encoding audio spectrograms, and arrays representing the purchase histories on e-Commerce sites.

## 13. Let's practice!

04:09 - 04:16

In this video, you've learned the basics of NMF. Now let's practice using it.

# Non-negative data

Which of the following 2-dimensional arrays are examples of non-negative data?

1. A tf-idf word-frequency array.
2. An array daily stock market price movements (up and down), where each row represents a company.
3. An array where rows are customers, columns are products and entries are 0 or 1, indicating whether a customer has purchased a product.

##### Answer the question

**50XP**

#### Possible Answers

**1 only**

**2 and 3**

**1 and 3**

Well done! Stock prices can go down as well as up, so an array of daily stock market price movements is not an example of non-negative data.

# NMF applied to Wikipedia articles

In the video, you saw NMF applied to transform a toy word-frequency array. Now it's your turn to apply NMF, this time using the tf-idf word-frequency array of Wikipedia articles, given as a csr matrix articles. Here, fit the model and transform the articles. In the next exercise, you'll explore the result.

## Instructions

100 XP

* Import NMF from sklearn.decomposition.
* Create an NMF instance called model with 6 components.
* Fit the model to the word count data articles.
* Use the .transform() method of model to transform articles, and assign the result to nmf\_features.
* Print nmf\_features to get a first idea what it looks like (.round(2) rounds the entries to 2 decimal places.)

# Import NMF

\_\_\_\_

# Create an NMF instance: model

model = \_\_\_\_

# Fit the model to articles

\_\_\_\_

# Transform the articles: nmf\_features

nmf\_features = \_\_\_\_

# Print the NMF features

print(nmf\_features.round(2))

# Import NMF

from sklearn.decomposition import NMF

# Create an NMF instance: model

model = NMF(n\_components=6)

# Fit the model to articles

model.fit(articles)

# Transform the articles: nmf\_features

nmf\_features = model.transform(articles)

# Print the NMF features

print(nmf\_features.round(2))

# Import NMF

from sklearn.decomposition import NMF

# Create an NMF instance: model

model = NMF(n\_components=6)

# Fit the model to articles

model.fit(articles)

# Transform the articles: nmf\_features

nmf\_features = model.transform(articles)

# Print the NMF features

print(nmf\_features.round(2))

[[0. 0. 0. 0. 0. 0.44]

[0. 0. 0. 0. 0. 0.57]

[0. 0. 0. 0. 0. 0.4 ]

[0. 0. 0. 0. 0. 0.38]

[0. 0. 0. 0. 0. 0.49]

[0.01 0.01 0.01 0.03 0. 0.33]

[0. 0. 0.02 0. 0.01 0.36]

[0. 0. 0. 0. 0. 0.49]

[0.02 0.01 0. 0.02 0.03 0.48]

[0.01 0.03 0.03 0.07 0.02 0.34]

[0. 0. 0.53 0. 0.03 0. ]

[0. 0. 0.36 0. 0. 0. ]

[0.01 0.01 0.31 0.06 0.01 0.02]

[0. 0.01 0.34 0.01 0. 0. ]

[0. 0. 0.43 0. 0.04 0. ]

[0. 0. 0.48 0. 0. 0. ]

[0.01 0.02 0.38 0.03 0. 0.01]

[0. 0. 0.48 0. 0. 0. ]

[0. 0.01 0.55 0. 0. 0. ]

[0. 0. 0.47 0. 0. 0. ]

[0. 0.01 0.02 0.52 0.06 0.01]

[0. 0. 0. 0.51 0. 0. ]

[0. 0.01 0. 0.42 0. 0. ]

[0. 0. 0. 0.44 0. 0. ]

[0. 0. 0. 0.5 0. 0. ]

[0.1 0.09 0. 0.38 0. 0.01]

[0. 0. 0. 0.57 0. 0.01]

[0.01 0.01 0. 0.47 0. 0.01]

[0. 0. 0. 0.58 0. 0. ]

[0. 0. 0. 0.53 0.01 0.01]

[0. 0.41 0. 0. 0. 0. ]

[0. 0.61 0. 0.01 0. 0. ]

[0.01 0.26 0. 0.02 0.01 0. ]

[0. 0.64 0. 0. 0. 0. ]

[0. 0.61 0. 0. 0. 0. ]

[0. 0.34 0. 0. 0. 0. ]

[0.01 0.32 0.02 0. 0.01 0. ]

[0.01 0.21 0.01 0.05 0.02 0.01]

[0.01 0.47 0. 0.02 0. 0. ]

[0. 0.64 0. 0. 0. 0. ]

[0. 0. 0. 0. 0.48 0. ]

[0. 0. 0. 0. 0.49 0. ]

[0. 0. 0. 0. 0.38 0.01]

[0. 0. 0. 0.01 0.54 0. ]

[0. 0. 0.01 0. 0.42 0. ]

[0. 0. 0. 0. 0.51 0. ]

[0. 0. 0. 0. 0.37 0. ]

[0. 0. 0.04 0. 0.23 0. ]

[0.01 0. 0.02 0.01 0.33 0.04]

[0. 0. 0. 0. 0.42 0. ]

[0.31 0. 0. 0. 0. 0. ]

[0.37 0. 0. 0. 0. 0. ]

[0.4 0.03 0. 0.02 0. 0.02]

[0.38 0. 0. 0.04 0. 0.01]

[0.44 0. 0. 0. 0. 0. ]

[0.46 0. 0. 0. 0. 0. ]

[0.28 0. 0. 0.05 0. 0.02]

[0.45 0. 0. 0. 0.01 0. ]

[0.29 0.01 0.01 0.01 0.19 0.01]

[0.38 0.01 0. 0.1 0.01 0. ]]

<script.py> output:

Fantastic - let's explore the meaning of these features in the next exercise!

## Exercise

## Exercise

# NMF features of the Wikipedia articles

Now you will explore the NMF features you created in the previous exercise. A solution to the previous exercise has been pre-loaded, so the array nmf\_features is available. Also available is a list titles giving the title of each Wikipedia article.

When investigating the features, notice that for both actors, the NMF feature 3 has by far the highest value. This means that both articles are reconstructed using mainly the 3rd NMF component. In the next video, you'll see why: NMF components represent topics (for instance, acting!).

## Instructions

100 XP

* Import pandas as pd.
* Create a DataFrame df from nmf\_features using pd.DataFrame(). Set the index to titles using index=titles.
* Use the .loc[] accessor of df to select the row with title 'Anne Hathaway', and print the result. These are the NMF features for the article about the actress Anne Hathaway.
* Repeat the last step for 'Denzel Washington' (another actor).

# Import pandas

\_\_\_\_

# Create a pandas DataFrame: df

df = \_\_\_\_

# Print the row for 'Anne Hathaway'

print(\_\_\_\_)

# Print the row for 'Denzel Washington'

print(\_\_\_\_)

# Import pandas

import pandas as pd

# Create a pandas DataFrame: df

df = pd.DataFrame(nmf\_features, index=titles)

# Print the row for 'Anne Hathaway'

print(df.loc['Anne Hathaway'])

# Print the row for 'Denzel Washington'

print(df.loc['Denzel Washington'])

# Import pandas

import pandas as pd

# Create a pandas DataFrame: df

df = pd.DataFrame(nmf\_features, index=titles)

# Print the row for 'Anne Hathaway'

print(df.loc['Anne Hathaway'])

# Print the row for 'Denzel Washington'

print(df.loc['Denzel Washington'])

0 0.004

1 0.000

2 0.000

3 0.576

4 0.000

5 0.000

Name: Anne Hathaway, dtype: float64

0 0.000

1 0.006

2 0.000

3 0.422

4 0.000

5 0.000

Name: Denzel Washington, dtype: float64

<script.py> output:

0 0.004

1 0.000

2 0.000

3 0.576

4 0.000

5 0.000

Name: Anne Hathaway, dtype: float64

0 0.000

1 0.006

2 0.000

3 0.422

4 0.000

5 0.000

Name: Denzel Washington, dtype: float64

Great work! Notice that for both actors, the NMF feature 3 has by far the highest value. This means that both articles are reconstructed using mainly the 3rd NMF component. In the next video, you'll see why: NMF components represent topics (for instance, acting!).

## Exercise

# NMF reconstructs samples

In this exercise, you'll check your understanding of how NMF reconstructs samples from its components using the NMF feature values. On the right are the components of an NMF model. If the NMF feature values of a sample are [2, 1], then which of the following is most likely to represent the original sample? A pen and paper will help here! You have to apply the same technique Ben used in the video to reconstruct the sample [0.1203 0.1764 0.3195 0.141].

### Possible answers

**[2.2, 1.1,2.1]**

**[0.5, 1.6, 3.1]**

**[-4.0, 1.0, -3.0]**

Well done, you've got it!

## 1. NMF learns interpretable parts

00:00 - 00:08

In this video, you'll learn that the components of NMF represent patterns that frequently occur in the samples.

## 2. Example: NMF learns interpretable parts

00:08 - 00:21

Let's consider a concrete example, where scientific articles are represented by their word frequencies. There are 20000 articles, and 800 words. So the array has 800 columns.

## 3. Applying NMF to the articles

00:21 - 00:31

Let's fit an NMF model with 10 components to the articles. The 10 components are stored as the 10 rows of a 2-dimensional numpy array.

## 4. NMF components are topics

00:31 - 00:45

The rows, or components, live in an 800-dimensional space - there is one dimension for each of the words. Aligning the words of our vocabulary with the columns of the NMF components allows them to be interpreted.

## 5. NMF components are topics

00:45 - 00:52

Choosing a component, such as this one, and looking at which words have the highest values,

## 6. NMF components are topics

00:52 - 01:00

we see that they fit a theme: the words are 'species', 'plant', 'plants', 'genetic', 'evolution' and 'life'.

## 7. NMF components are topics

01:00 - 01:04

The same happens if any other component is considered.

## 8. NMF components

01:04 - 01:42

So if NMF is applied to documents, then the components correspond to topics, and the NMF features reconstruct the documents from the topics. If NMF is applied to a collection of images, on the other hand, then the NMF components represent patterns that frequently occur in the images. In this example, for instance, NMF decomposes images from an LCD display into the individual cells of the display. This example you'll investigate for yourself in the exercises. To do this, you'll need to know how to represent a collection of images as a non-negative array.

## 9. Grayscale images

01:42 - 02:07

An image in which all the pixels are shades of gray ranging from black to white is called a "grayscale image". Since there are only shades of grey, a grayscale image can be encoded by the brightness of every pixel. Representing the brightness as a number between 0 and 1, where 0 is totally black and 1 is totally white, the image can be represented as 2-dimensional array of numbers.

## 10. Grayscale image example

02:07 - 02:12

Here, for example, is a grayscale photo of the moon!

## 11. Grayscale images as flat arrays

02:12 - 02:26

These 2-dimensional arrays of numbers can then be flattened by enumerating the entries. For instance, we could read-off the values row-by-row, from left-to-right and top to bottom.

## 12. Grayscale images as flat arrays

02:26 - 02:32

The grayscale image is now represented by a flat array of non-negative numbers.

## 13. Encoding a collection of images

02:32 - 03:00

A collection of grayscale images of the same size can thus be encoded as a 2-dimensional array, in which each row represents an image as a flattened array, and each column represents a pixel. Viewing the images as samples, and the pixels as features, we see that the data is arranged similarly to the word frequency array. Indeed, the entries of this array are non-negative, so NMF can be used to learn the parts of the images.

## 14. Visualizing samples

03:00 - 03:26

It's difficult to visualize an image by just looking at the flattened array. To recover the image, use the reshape method of the sample, specifying the dimensions of the original image as a tuple. This yields the 2-dimensional array of pixel brightnesses. To display the corresponding image, import pyplot, and pass the 2-dimensional array to the plt dot imshow function.

## 15. Let's practice!

03:26 - 03:40

In this video, you've seen how NMF components can be intepreted as patterns that frequently occur in the samples. It's time to get some practice and investigate this phenomenon for yourself.

## Exercise

## Exercise

# NMF learns topics of documents

In the video, you learned when NMF is applied to documents, the components correspond to topics of documents, and the NMF features reconstruct the documents from the topics. Verify this for yourself for the NMF model that you built earlier using the Wikipedia articles. Previously, you saw that the 3rd NMF feature value was high for the articles about actors Anne Hathaway and Denzel Washington. In this exercise, identify the topic of the corresponding NMF component.

The NMF model you built earlier is available as model, while words is a list of the words that label the columns of the word-frequency array.

After you are done, take a moment to recognize the topic that the articles about Anne Hathaway and Denzel Washington have in common!

## Instructions

100 XP

* Import pandas as pd.
* Create a DataFrame components\_df from model.components\_, setting columns=words so that columns are labeled by the words.
* Print components\_df.shape to check the dimensions of the DataFrame.
* Use the .iloc[] accessor on the DataFrame components\_df to select row 3. Assign the result to component.
* Call the .nlargest() method of component, and print the result. This gives the five words with the highest values for that component.

# Import pandas

import pandas as pd

# Create a DataFrame: components\_df

components\_df = \_\_\_\_

# Print the shape of the DataFrame

print(components\_df.shape)

# Select row 3: component

component = \_\_\_\_

# Print result of nlargest

print(component.nlargest())

# Import pandas

import pandas as pd

# Create a DataFrame: components\_df

components\_df = pd.DataFrame(model.components\_, columns=words)

# Print the shape of the DataFrame

print(components\_df.shape)

# Select row 3: component

component = components\_df.iloc[3]

# Print result of nlargest

print(component.nlargest())

**# Import pandas**

**import pandas as pd**

**# Create a DataFrame: components\_df**

**components\_df = pd.DataFrame(model.components\_, columns=words)**

**# Print the shape of the DataFrame**

**print(components\_df.shape)**

**# Select row 3: component**

**component = components\_df.iloc[3]**

**# Print result of nlargest**

**print(component.nlargest())**

**<script.py> output:**

**(6, 13125)**

**film 0.628**

**award 0.253**

**starred 0.245**

**role 0.211**

**actress 0.186**

**Name: 3, dtype: float64**

**Great work! Take a moment to recognise the topics that the articles about Anne Hathaway and Denzel Washington have in common!**

## Exercise

# Explore the LED digits dataset

In the following exercises, you'll use NMF to decompose grayscale images into their commonly occurring patterns. Firstly, explore the image dataset and see how it is encoded as an array. You are given 100 images as a 2D array samples, where each row represents a single 13x8 image. The images in your dataset are pictures of a LED digital display.

## Instructions

100 XP

* Import matplotlib.pyplot as plt.
* Select row 0 of samples and assign the result to digit. For example, to select column 2 of an array a, you could use a[:,2]. Remember that since samples is a NumPy array, you can't use the .loc[] or iloc[] accessors to select specific rows or columns.
* Print digit. This has been done for you. Notice that it is a 1D array of 0s and 1s.
* Use the .reshape() method of digit to get a 2D array with shape (13, 8). Assign the result to bitmap.
* Print bitmap, and notice that the 1s show the digit 7!
* Use the plt.imshow() function to display bitmap as an image.

# Import pyplot

from matplotlib import pyplot as plt

# Select the 0th row: digit

digit = \_\_\_\_

# Print digit

print(digit)

# Reshape digit to a 13x8 array: bitmap

bitmap = \_\_\_\_

# Print bitmap

print(bitmap)

# Use plt.imshow to display bitmap

plt.\_\_\_\_(\_\_\_\_, cmap='gray', interpolation='nearest')

plt.colorbar()

plt.show()

# Import pyplot

from matplotlib import pyplot as plt

# Select the 0th row: digit

digit = samples[0, :]

# Print digit

print(digit)

# Reshape digit to a 13x8 array: bitmap

bitmap = digit.reshape(13,8)

# Print bitmap

print(bitmap)

# Use plt.imshow to display bitmap

plt.imshow(bitmap, cmap='gray', interpolation='nearest')

plt.colorbar()

plt.show()

**# Import pyplot**

**from matplotlib import pyplot as plt**

**# Select the 0th row: digit**

**digit = samples[0, :]**

**# Print digit**

**print(digit)**

**# Reshape digit to a 13x8 array: bitmap**

**bitmap = digit.reshape(13,8)**

**# Print bitmap**

**print(bitmap)**

**# Use plt.imshow to display bitmap**

**plt.imshow(bitmap, cmap='gray', interpolation='nearest')**

**plt.colorbar()**

**plt.show()**

**<script.py> output:**

**[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 1. 1. 1. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0.**

**0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 1. 0.**

**0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 1. 0.**

**0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0.**

**0. 0. 0. 0. 0. 0. 0. 0.]**

**[[0. 0. 0. 0. 0. 0. 0. 0.]**

**[0. 0. 1. 1. 1. 1. 0. 0.]**

**[0. 0. 0. 0. 0. 0. 1. 0.]**

**[0. 0. 0. 0. 0. 0. 1. 0.]**

**[0. 0. 0. 0. 0. 0. 1. 0.]**

**[0. 0. 0. 0. 0. 0. 1. 0.]**

**[0. 0. 0. 0. 0. 0. 0. 0.]**

**[0. 0. 0. 0. 0. 0. 1. 0.]**

**[0. 0. 0. 0. 0. 0. 1. 0.]**

**[0. 0. 0. 0. 0. 0. 1. 0.]**

**[0. 0. 0. 0. 0. 0. 1. 0.]**

**[0. 0. 0. 0. 0. 0. 0. 0.]**

**[0. 0. 0. 0. 0. 0. 0. 0.]]**

**Excellent job! You'll explore this dataset further in the next exercise and see for yourself how NMF can learn the parts of images.**

Daily XP1200

## Exercise

## Exercise

# NMF learns the parts of images

Now use what you've learned about NMF to decompose the digits dataset. You are again given the digit images as a 2D array samples. This time, you are also provided with a function show\_as\_image() that displays the image encoded by any 1D array:

def show\_as\_image(sample):

bitmap = sample.reshape((13, 8))

plt.figure()

plt.imshow(bitmap, cmap='gray', interpolation='nearest')

plt.colorbar()

plt.show()

After you are done, take a moment to look through the plots and notice how NMF has expressed the digit as a sum of the components!

## Instructions

100 XP

* Import NMF from sklearn.decomposition.
* Create an NMF instance called model with 7 components. (7 is the number of cells in an LED display).
* Apply the .fit\_transform() method of model to samples. Assign the result to features.
* To each component of the model (accessed via model.components\_), apply the show\_as\_image() function to that component inside the loop.
* Assign the row 0 of features to digit\_features.
* Print digit\_features.
* # Import NMF
* \_\_\_\_
* # Create an NMF model: model
* model = \_\_\_\_
* # Apply fit\_transform to samples: features
* features = \_\_\_\_
* # Call show\_as\_image on each component
* for component in model.components\_:
* \_\_\_\_
* # Select the 0th row of features: digit\_features
* digit\_features = \_\_\_\_
* # Print digit\_features
* print(digit\_features)

# Import NMF

from sklearn.decomposition import NMF

# Create an NMF model: model

model = NMF(n\_components=7)

# Apply fit\_transform to samples: features

features = model.fit\_transform(samples)

# Call show\_as\_image on each component

for component in model.components\_:

    show\_as\_image(component)

# Select the 0th row of features: digit\_features

digit\_features = features[0, :]

# Print digit\_features

print(digit\_features)

**# Import NMF**

**from sklearn.decomposition import NMF**

**# Create an NMF model: model**

**model = NMF(n\_components=7)**

**# Apply fit\_transform to samples: features**

**features = model.fit\_transform(samples)**

**# Call show\_as\_image on each component**

**for component in model.components\_:**

**show\_as\_image(component)**

**# Select the 0th row of features: digit\_features**

**digit\_features = features[0, :]**

**# Print digit\_features**

**print(digit\_features)**

**<script.py> output:**

**[4.76823559e-01 0.00000000e+00 0.00000000e+00 5.90605054e-01**

**4.81559442e-01 0.00000000e+00 7.37551667e-16]**

**Great work! Take a moment to look through the plots and notice how NMF has expressed the digit as a sum of the components!**

## Exercise

## Exercise

# PCA doesn't learn parts

Unlike NMF, PCA doesn't learn the parts of things. Its components do not correspond to topics (in the case of documents) or to parts of images, when trained on images. Verify this for yourself by inspecting the components of a PCA model fit to the dataset of LED digit images from the previous exercise. The images are available as a 2D array samples. Also available is a modified version of the show\_as\_image() function which colors a pixel red if the value is negative.

After submitting the answer, notice that the components of PCA do not represent meaningful parts of images of LED digits!

## Instructions

100 XP

* Import PCA from sklearn.decomposition.
* Create a PCA instance called model with 7 components.
* Apply the .fit\_transform() method of model to samples. Assign the result to features.
* To each component of the model (accessed via model.components\_), apply the show\_as\_image() function to that component inside the loop.
* # Import PCA
* \_\_\_\_
* # Create a PCA instance: model
* model = \_\_\_\_
* # Apply fit\_transform to samples: features
* features = \_\_\_\_
* # Call show\_as\_image on each component
* for component in \_\_\_\_:
* \_\_\_\_

# Import PCA

from sklearn.decomposition import PCA

# Create a PCA instance: model

model = PCA(n\_components=7)

# Apply fit\_transform to samples: features

features = model.fit\_transform(samples)

# Call show\_as\_image on each component

for component in model.components\_:

    show\_as\_image(component)

# Import PCA from sklearn.decomposition import PCA # Create a PCA instance: model model = PCA(n\_components=7) # Apply fit\_transform to samples: features features = model.fit\_transform(samples) # Call show\_as\_image on each component for component in model.components\_: show\_as\_image(component)

Great work! Notice that the components of PCA do not represent meaningful parts of images of LED digits!

## 1. Building recommender systems using NMF

00:00 - 00:01

## 2. Finding similar articles

00:01 - 00:21

Suppose that you are an engineer at a large online newspaper. You've been given the task of recommending articles that are similar to the article currently being read by a customer. Given an article, how can you find articles that have similar topics? In this video, you'll learn how to solve this problem, and others like it, by using NMF.

## 3. Strategy

00:21 - 00:48

Our strategy for solving this problem is to apply NMF to the word-frequency array of the articles, and to use the resulting NMF features. You learned in the previous videos these NMF features describe the topic mixture of an article. So similar articles will have similar NMF features. But how can two articles be compared using their NMF features? Before answering this question, let's set the scene by doing the first step.

## 4. Apply NMF to the word-frequency array

00:48 - 01:08

You are given a word frequency array articles corresponding to the collection of newspaper articles in question. Import NMF, create the model, and use the fit\_transform method to obtain the transformed articles. Now we've got NMF features for every article, given by the columns of the new array.

## 5. Strategy

01:08 - 01:13

Now we need to define how to compare articles using their NMF features.

## 6. Versions of articles

01:13 - 01:27

Similar documents have similar topics, but it isn't always the case that the NMF feature values are exactly the same. For instance, one version of a document might use very direct language,

## 7. Versions of articles

01:27 - 01:40

whereas other versions might interleave the same content with meaningless chatter. Meaningless chatter reduces the frequency of the topic words overall, which reduces the values of the NMF features representing the topics.

## 8. Versions of articles

01:40 - 01:48

However, on a scatter plot of the NMF features, all these versions lie on a single line passing through the origin.

## 9. Cosine similarity

01:48 - 02:12

For this reason, when comparing two documents, it's a good idea to compare these lines. We'll compare them using what is known as the cosine similarity, which uses the angle between the two lines. Higher values indicate greater similarity. The technical definition of the cosine similarity is out the scope of this course, but we've already gained an intuition.

## 10. Calculating the cosine similarities

02:12 - 02:34

Let's see now how to compute the cosine similarity. Firstly, import the normalize function, and apply it to the array of all NMF features. Now select the row corresponding to the current article, and pass it to the dot method of the array of all normalized features. This results in the cosine similarities.

## 11. DataFrames and labels

02:34 - 03:03

With the help of a pandas DataFrame, we can label the similarities with the article titles. Start by importing pandas. After normalizing the NMF features, create a DataFrame whose rows are the normalized features, using the titles as an index. Now use the loc method of the DataFrame to select the normalized feature values for the current article, using its title 'Dog bites man'. Calculate the cosine similarities using the dot method of the DataFrame.

## 12. DataFrames and labels

03:03 - 03:17

Finally, use the nlargest method of the resulting pandas Series to find the articles with the highest cosine similarity. We see that all of them are concerned with 'domestic animals' and/or 'danger'!

## 13. Let's practice!

03:17 - 03:28

Now you've learned how to build a recommender system using NMF and the cosine similarity. Now it's time to get some practice recommending!

## Exercise

## Exercise

# Which articles are similar to 'Cristiano Ronaldo'?

In the video, you learned how to use NMF features and the cosine similarity to find similar articles. Apply this to your NMF model for popular Wikipedia articles, by finding the articles most similar to the article about the footballer Cristiano Ronaldo. The NMF features you obtained earlier are available as nmf\_features, while titles is a list of the article titles.

## Instructions

100 XP

* Import normalize from sklearn.preprocessing.
* Apply the normalize() function to nmf\_features. Store the result as norm\_features.
* Create a DataFrame df from norm\_features, using titles as an index.
* Use the .loc[] accessor of df to select the row of 'Cristiano Ronaldo'. Assign the result to article.
* Apply the .dot() method of df to article to calculate the cosine similarity of every row with article.
* Print the result of the .nlargest() method of similarities to display the most similar articles. This has been done for you, so hit 'Submit Answer' to see the result!
* # Perform the necessary imports
* import pandas as pd
* from \_\_\_\_ import \_\_\_\_
* # Normalize the NMF features: norm\_features
* norm\_features = \_\_\_\_
* # Create a DataFrame: df
* df = \_\_\_\_
* # Select the row corresponding to 'Cristiano Ronaldo': article
* article = df.loc[\_\_\_\_]
* # Compute the dot products: similarities
* similarities = \_\_\_\_
* # Display those with the largest cosine similarity
* print(similarities.nlargest())

# Perform the necessary imports

import pandas as pd

from sklearn.preprocessing import normalize

# Normalize the NMF features: norm\_features

norm\_features = normalize(nmf\_features)

# Create a DataFrame: df

df = pd.DataFrame(norm\_features, index=titles)

# Select the row corresponding to 'Cristiano Ronaldo': article

article = df.loc['Cristiano Ronaldo']

# Compute the dot products: similarities

similarities = df.dot(article)

# Display those with the largest cosine similarity

print(similarities.nlargest())

# Perform the necessary imports

import pandas as pd

from sklearn.preprocessing import normalize

# Normalize the NMF features: norm\_features

norm\_features = normalize(nmf\_features)

# Create a DataFrame: df

df = pd.DataFrame(norm\_features, index=titles)

# Select the row corresponding to 'Cristiano Ronaldo': article

article = df.loc['Cristiano Ronaldo']

# Compute the dot products: similarities

similarities = df.dot(article)

# Display those with the largest cosine similarity

print(similarities.nlargest())

Cristiano Ronaldo 1.0

Franck Ribéry 1.0

Radamel Falcao 1.0

Zlatan Ibrahimović 1.0

France national football team 1.0

dtype: float64

<script.py> output:

Cristiano Ronaldo 1.0

Franck Ribéry 1.0

Radamel Falcao 1.0

Zlatan Ibrahimović 1.0

France national football team 1.0

dtype: float64

Great work - although you may need to know a little about football (or soccer, depending on where you're from!) to be able to evaluate for yourself the quality of the computed similarities!

## xercise

## Exercise

# Recommend musical artists part I

In this exercise and the next, you'll use what you've learned about NMF to recommend popular music artists! You are given a sparse array artists whose rows correspond to artists and whose columns correspond to users. The entries give the number of times each artist was listened to by each user.

In this exercise, build a pipeline and transform the array into normalized NMF features. The first step in the pipeline, MaxAbsScaler, transforms the data so that all users have the same influence on the model, regardless of how many different artists they've listened to. In the next exercise, you'll use the resulting normalized NMF features for recommendation!

## Instructions

100 XP

* Import:
  + NMF from sklearn.decomposition.
  + Normalizer and MaxAbsScaler from sklearn.preprocessing.
  + make\_pipeline from sklearn.pipeline.
* Create an instance of MaxAbsScaler called scaler.
* Create an NMF instance with 20 components called nmf.
* Create an instance of Normalizer called normalizer.
* Create a pipeline called pipeline that chains together scaler, nmf, and normalizer.
* Apply the .fit\_transform() method of pipeline to artists. Assign the result to norm\_features.
* # Perform the necessary imports
* from \_\_\_\_ import \_\_\_\_
* from \_\_\_\_ import \_\_\_\_, \_\_\_\_
* from \_\_\_\_ import \_\_\_\_
* # Create a MaxAbsScaler: scaler
* scaler = \_\_\_\_
* # Create an NMF model: nmf
* nmf = \_\_\_\_
* # Create a Normalizer: normalizer
* normalizer = \_\_\_\_
* # Create a pipeline: pipeline
* pipeline = \_\_\_\_
* # Apply fit\_transform to artists: norm\_features
* norm\_features = \_\_\_\_

# Perform the necessary imports

from sklearn.decomposition import NMF

from sklearn.preprocessing import Normalizer, MaxAbsScaler

from sklearn.pipeline import make\_pipeline

# Create a MaxAbsScaler: scaler

scaler = MaxAbsScaler()

# Create an NMF model: nmf

nmf = NMF(n\_components=20)

# Create a Normalizer: normalizer

normalizer = Normalizer()

# Create a pipeline: pipeline

pipeline = make\_pipeline(scaler, nmf, normalizer)

# Apply fit\_transform to artists: norm\_features

norm\_features = pipeline.fit\_transform(artists)

# Perform the necessary imports from sklearn.decomposition import NMF from sklearn.preprocessing import Normalizer, MaxAbsScaler from sklearn.pipeline import make\_pipeline # Create a MaxAbsScaler: scaler scaler = MaxAbsScaler() # Create an NMF model: nmf nmf = NMF(n\_components=20) # Create a Normalizer: normalizer normalizer = Normalizer() # Create a pipeline: pipeline pipeline = make\_pipeline(scaler, nmf, normalizer) # Apply fit\_transform to artists: norm\_features norm\_features = pipeline.fit\_transform(artists)

Excellent work - now that you've computed the normalized NMF features, you'll use them in the next exercise to recommend musical artists!

Daily XP350

## Exercise

## Exercise

# Recommend musical artists part II

Suppose you were a big fan of Bruce Springsteen - which other musical artists might you like? Use your NMF features from the previous exercise and the cosine similarity to find similar musical artists. A solution to the previous exercise has been run, so norm\_features is an array containing the normalized NMF features as rows. The names of the musical artists are available as the list artist\_names.

## Instructions

100 XP

* Import pandas as pd.
* Create a DataFrame df from norm\_features, using artist\_names as an index.
* Use the .loc[] accessor of df to select the row of 'Bruce Springsteen'. Assign the result to artist.
* Apply the .dot() method of df to artist to calculate the dot product of every row with artist. Save the result as similarities.
* Print the result of the .nlargest() method of similarities to display the artists most similar to 'Bruce Springsteen'.
* # Import pandas
* \_\_\_\_
* # Create a DataFrame: df
* df = \_\_\_\_
* # Select row of 'Bruce Springsteen': artist
* artist = df.loc[\_\_\_\_]
* # Compute cosine similarities: similarities
* similarities = \_\_\_\_
* # Display those with highest cosine similarity
* \_\_\_\_
* # Import pandas
* import pandas as pd
* # Create a DataFrame: df
* df = pd.DataFrame(norm\_features, index=artist\_names)
* # Select row of 'Bruce Springsteen': artist
* artist = df.loc['Bruce Springsteen']
* # Compute cosine similarities: similarities
* similarities = df.dot(artist)
* # Display those with highest cosine similarity
* print(similarities.nlargest())

**# Import pandas**

**import pandas as pd**

**# Create a DataFrame: df**

**df = pd.DataFrame(norm\_features, index=artist\_names)**

**# Select row of 'Bruce Springsteen': artist**

**artist = df.loc['Bruce Springsteen']**

**# Compute cosine similarities: similarities**

**similarities = df.dot(artist)**

**# Display those with highest cosine similarity**

**print(similarities.nlargest())**

**<script.py> output:**

**Bruce Springsteen 1.000**

**Neil Young 0.956**

**Van Morrison 0.872**

**Leonard Cohen 0.865**

**Bob Dylan 0.859**

**dtype: float64**

**Well done, and congratulations on reaching the end of the course!**