Unsupervised learning
by datacamp

finds patterns in data
using techniques such as ‘clustering’ and ‘dimension reduction’
supervised is learning with ‘labels’
unsupervised is learning without ‘labels’

Always reminding
columns are the features (or measurements)
rows are the samples (or observations)

Things get interesting quick
each feature creates a dimension
dimension = number of features
we cannot visualize past three dimensions
but we can get insights of these higher dimensions with our models

simple example using the Iris dataset (3 iris types, 4 features)
dataset named ’samples’
using k-means clustering
from sklearn.cluster import KMeans
#instantiate model
#specify number of clusters, here we chose 3 because we know that there is 3 iris types
model = KMeans(n_clusters=3)
#fit 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
model.fit(samples)
#predict > here this returns a cluster label for each sample indicating which cluster a sample belongs
labels = model.predict(samples)
print(labels)

Nice feature
k-means can determine where new samples belong without starting over
does this by remembering the mean of the samples in each cluster
these means are called ‘centroids’
new samples are assigned to the cluster whose centrold is closest

How to pass new samples
pass the array of new samples to the predict method of the kmeans model
new_labels = model.predict(new_samples)
print(new_labels)

Visualize with scatter plot
import matplotlib.pyplot as plt
#sepal length is in the 0th column of the array
xs = samples[:,0]
#petal length is in the 2nd column
ys = samples[:,2]
plt.scatter(xs, ys, c=labels)
#c=labels allows us to color by cluster label
plt.show()

Nice simple example:
# 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()

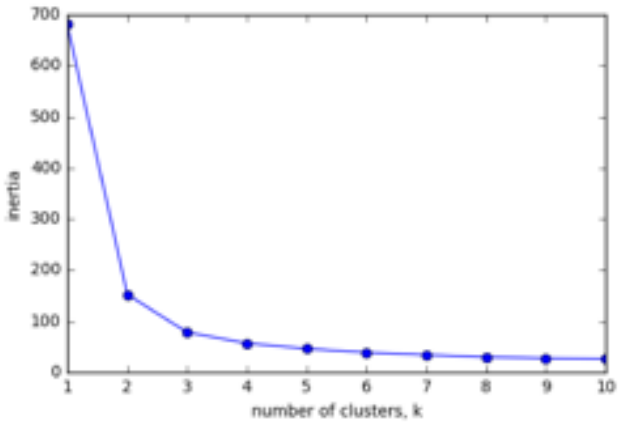
Evaluating a clustering
measure quality of a clustering
this informs the choice of how many clusters to look for
this can be easy depending on our dataset
we'll first look at a scenario where we know the groupings
we'll continue looking at the iris dataset

we can form tables that show relationships
in our example a table that compares clusters vs species
call this ‘cross-tabulation’
can create this in pandas
df = pd.DataFrame({‘labels’:labels, ‘species’:species})
ct = pd.crosstab(df[‘labels’], df[‘species’])
print(ct)

species	setosa	versicolor	virginica
labels			
0	0	2	36
1	50	0	0
2	0	48	14

How to measure when samples are not labelled?
good clustering has tight clusters
how spread out the samples within each cluster ar can be measured by the ‘inertia’
inertia measures how are samples are from their centroids
lower values of inertia are better (represent clusters that are not spread out)
automatically measure with KMeans, can be called as intertia attribute
model = KMeans(n_clusters=3)
model.fit(samples)
print(model.inertia_)

How many clusters to choose?
*a trade-off as usual, no discrete answer, must subjectively choose
but guidance:
low inertia but not too many clusters
plot out inertia and choose somewhere on the ‘elbow’
our example:



n_clusters 3 looks to be the ideal spot

Nice KMeans example

```
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()
```

Transforming features for better clusterings

Feature variances

variance of a feature measures the spread of its values

Need to scale

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
Our friend, StandardScaler transforms every feature to have mean 0 and variance 1
from sklearn.preprocesing import StandardScaler
scaler = StandardScaler()
scaler.fit(samples)
StandardScaler(copy=True, with_mean=True, with_std=True)
samples_scaled = scaler.transform(samples)
#our transfom() method can now be used to standardize any samples (ie new ones)
#'with_mean' argument, if True the mean is subtracted, making the scaled data have 0 mean
#'with_std' must be set to True (default) otherwise data does not scale; as True data is scaled by dividing it by the standard deviation
#this ensurese that the transformed data will a standard deviation of 1

Sidebar - what does 'copy' argument do:
The `copy` parameter in `StandardScaler` controls whether a new copy of the data should be created during scaling. It determines whether the original data should be modified in place or whether a new array should be created with the scaled values. The default value is `True`, which means that a copy is made by default.

There are a few reasons why you might want to use a copy of the data instead of modifying the original data in place:

1. Data Integrity: If you want to preserve the integrity of the original data and avoid any unintended modifications, using a copy is a safer option. Modifying the original data directly may lead to unintended consequences, especially if you are dealing with large datasets or shared data in multiple parts of your code.

2. Reusability: By creating a copy, you can reuse the original unscaled data later in your code or analysis. This is useful if you need to compare the scaled and unscaled data or if you want to apply different scaling techniques to the same data at different points in your workflow.

3. Debugging: When troubleshooting issues related to scaling or data preprocessing, having the original data as a reference can help in debugging and understanding the transformations applied to the data.

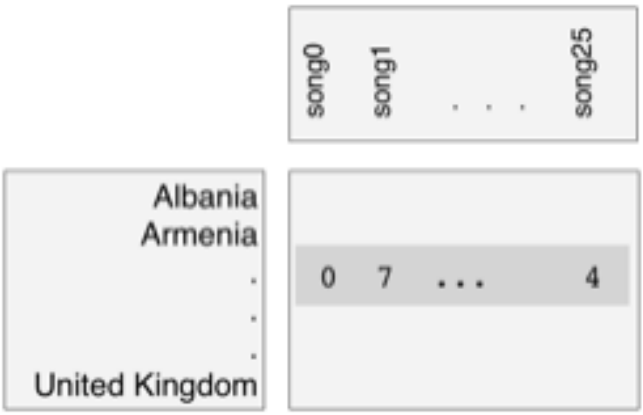
However, there are scenarios where using a copy might not be necessary or even less memory-efficient, especially when dealing with large datasets. In such cases, you can set `copy=False` to perform the scaling in place, which saves memory

and reduces the overhead of creating a new array. Just keep in mind that modifying the data in place might lead to unexpected results if the original data is needed elsewhere in your code or analysis.

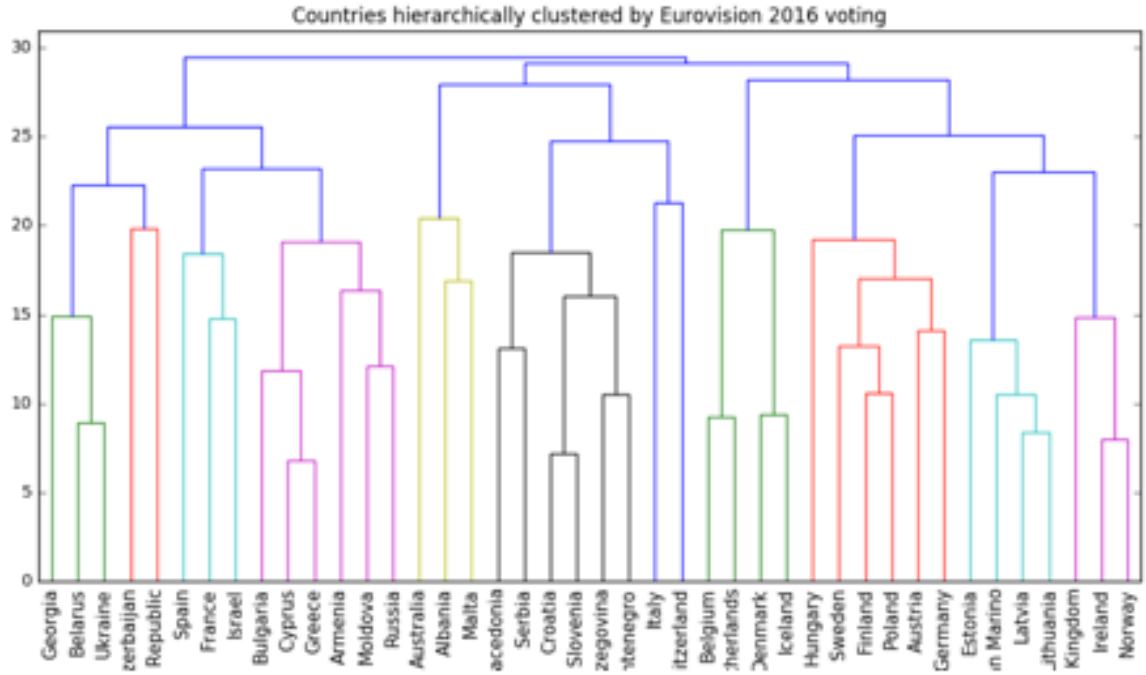
Ultimately, the choice of using a copy or not depends on your specific use case and the trade-offs between data integrity, reusability, memory efficiency, and ease of debugging. If you're unsure, using the default behavior with `copy=True` is generally a safer option.

Example of scaler k-means pipeline
```
from sklearn.preprocessing import StandardScaler
from sklearn.cluster import KMeans
from sklearn.pipeline import make_pipeline
#scaler must come first
scaler = StandardScaler()
kmeans = KMeans(n_clusters=3)
pipeline = make_pipeline(scaler, kmeans)
#arguments are the steps that you want to compose
#fit both scaler and kmeans, done off of pipeline object
pipline.fit(samples)
#use .predict to obtain the cluster labels
labels = pipeline.predict(samples)
```

Hierarchical clustering
arranges samples into a hierarchy of clusters
example - Eurovision scoring dataset to help visualize the process



Here is the result of applyinghierarchical clustering to the Eurovision scores:



this tree-like diagram is called a ‘dendrogram’
Different types of clustering
‘agglomerative’ clustering > each unit starts as a cluster, then the two closest merge, and continues until a single cluster
also ‘divisive’ clustering which works in reverse
above is agglomerative
this is a bottom up process
clusters are represented as vertical lines
joining of vertical lines indicates a merging of clusters

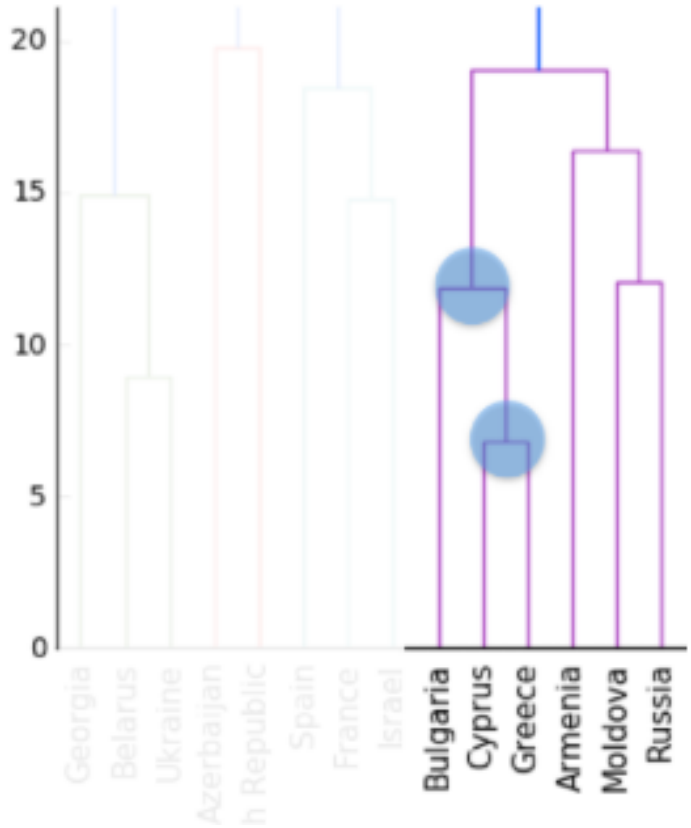
#plt to show dendrogram
import matplotlib.pyplot as plt
from scipy.cluster.hierarchy import linkage, dendrogram
#linkage function performs the hierarchical clustering
mergings = linkage(samples, method=‘complete’)
dendrogram(mergings, labels=country_names, leaf_rotation=90, leaf_font_size=6)
plt.show()

Example
# 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()
```

Cluster labels in hierarchical clustering
how to extract clusters from intermediate stages
which can then be used for further computations, such as cross tabulations
specified by choosing a height on the dendrogram
what does the height mean?
height on dendrogram = distance between merging clusters
what does this mean



Cyprus and Greece were clustered at 6
Bulgaria merged when the distance between them was 12
choosing a height specifies that the hierarchical clustering should stop merging clusters when are clusters are at said height
*this distance is measured using the linkage() method
above we used method='complete' where the distance between two clusters is the maximum of the distances between their samples

Extracting cluster labels
for intermediate stage using the fcluster() function
returns a NumPy array of cluster labels
from scipy.cluster.hierarchy import linkage
mergings = linkage(samples, method='complete')
from scipy.cluster.hierarchy import fcluster
labels = fcluster(mergings, 15, criterion='distance')
#15 is the specified height, see graph above
print(labels)

Inspecting and aligning cluster labels
import pandas as pd
pairs = pd.DataFrame({'labels': labels, 'countries': country_names})
print(pairs.sort_values('labels'))
output >

	countries	labels
5	Belarus	1
40	Ukraine	1
...		
36	Spain	5
8	Bulgaria	6
19	Greece	6
10	Cyprus	6
28	Moldova	7
...		

*scipy cluster labels start at 1, not at 0 like they do in scikit-learn

Example - continue to us on dataset 'mergings'
# 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)
```

t-SNE for 2-dimensional maps
t-SNE stands for 't-distributed stochastic neighboer embedding'
maps samples from high-dimensional space into a 2 or 3D space so that they can be visualized
distortion is inevitable but t-SNE does a great job of approximately representing the distances between the samples
example - using the iris data set (named samples)

```
import matplotlib.pyplot as plt
from sklearn.manifold import TSNE
model = TSNE(learning_rate=100)
transformed = model.fit_transform(samples)
xs = transformed[:,0]
ys = transformed[:,1]
plt.scatter(xs, ys, c=species)
plt.show()
```

t-SNE only has a fit_transform() method which simultaneously fits the model and transforms the data
*fit and transform are not separate, which means you can't extend a t-SNE map to include new samples
which means you have to start over each time
'learning rate' requires some trial and error
no direct technique on choosing
normally a good range is between 50 and 200
when you've made a bad choice your visualization will show your data points bunched together on the scatter plot
**axes of a t-SNE plot do not have any interpretable meaning
they will be different every time, even on the same data
**key is that while the orientation of the plot is different each time, the data points have the same position relative to one another

Example

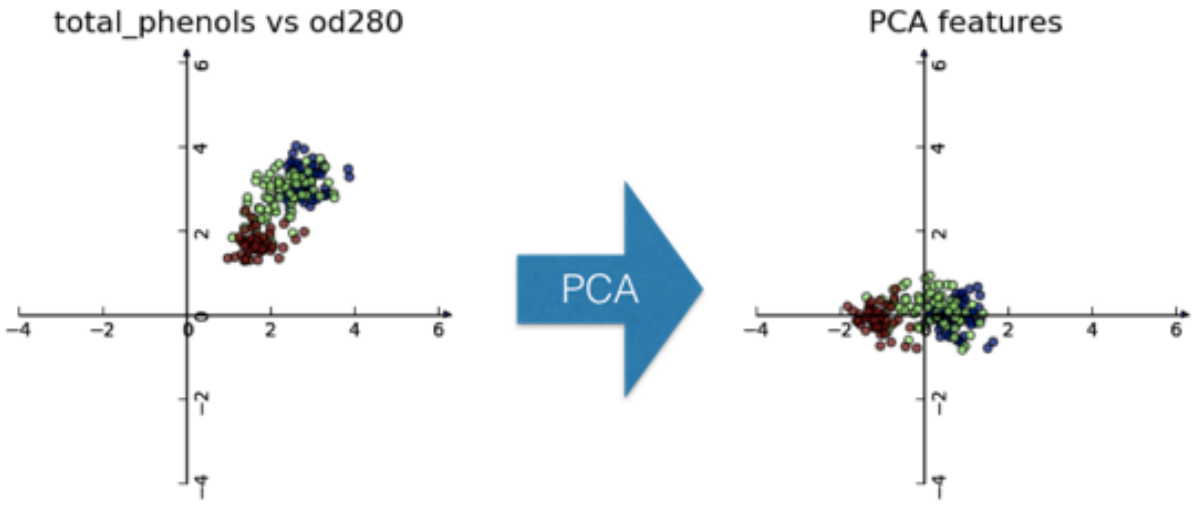
```
# Import TSNE
from sklearn.manifold import TSNE

# 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()
```

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
this is big in a world of big data
**dimension reduction's most important function is to reduce a dataset to its 'bare bones'
meaning discarding noisy features that cause big problems for supervised learning tasks like regression and classification
in the real world, dimension reduction often is what makes predictin possible

Principal Component Analysis (PCA)
most fundamental of dimension reduction techniques
does dimension reduction in two steps
first step > decorrelation (doesn't change the dimension of the data)
second step > reduces dimension

First step
PCA aligns data with axes
rotates the samples so that they are aligned with the coordinate axes
also shifts the samples so that they have mean zero
no information is lost - this is true no matter how many features your dataset has
visual explaining:



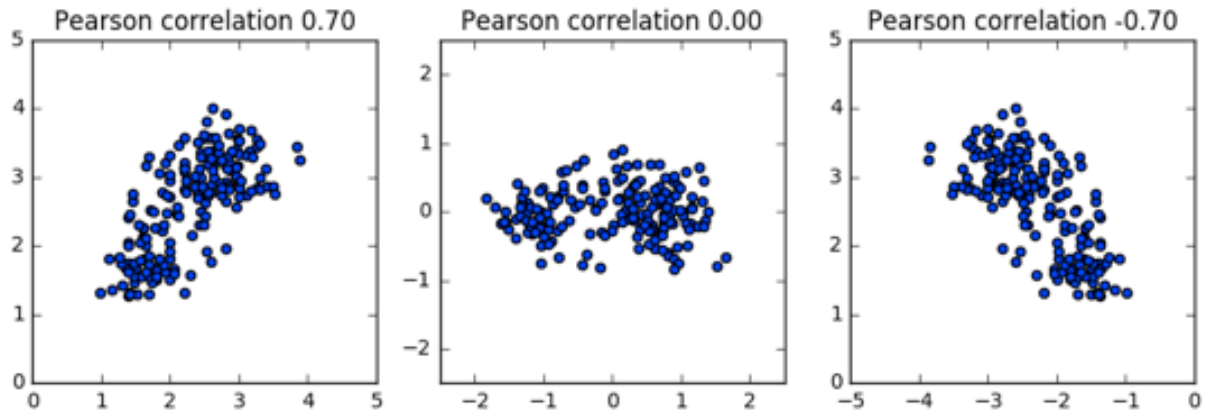
PCA has a fit and transform method just like StandardScaler()
fit() learns how to shift and how to rotate the samples, but doesn't actually change them
fit() learns the transformation from the given data
transform() applies the transformation that fit learned
*transform() can be applied to new and unseen samples

example - using 'samples' an array of two features in wine dataset (total_phenols and od280)
from sklearn.decomposition import PCA
model = PCA()
model.fit(samples)
transformed = model.transform(samples)
print(transformed)
output > new array but with same amount of rows and columns as the original sample array
has a row for each transformed sample
columns correspond to 'PCA features'

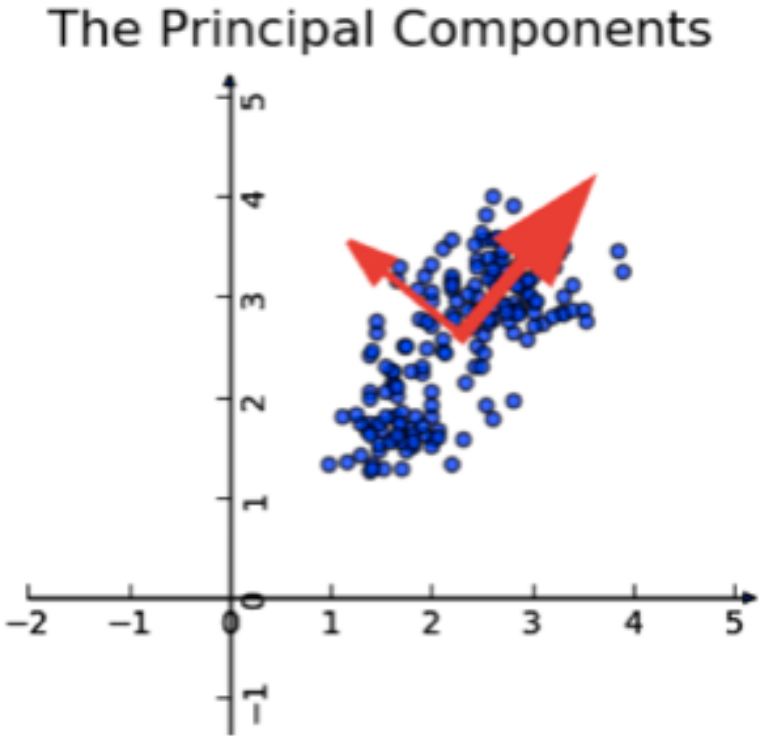
PCA features are not correlated
often features of a dataset are correlated
*with PCA features this is not the case
reason is due to the rotation it performs
this action 'de-correlates' the data in the sense that the columns of the transformed array are not linearly correlated

Pearson correlation
measures linear correlation
takes values between -1 and 1
larger values indicate a stronger correlation

0 indicates no correlation



Why is it called principal component anaylsis?
because it learns the ‘principal components’ of the data
these are the directions in which the samples vary the most
*it is the principal components that PCA aligns with the coordinate axes



once a PCA model has been fit, the principal components are available as the components attribute
this is a NumPy array with one row for each principal component

Example
# 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)
```

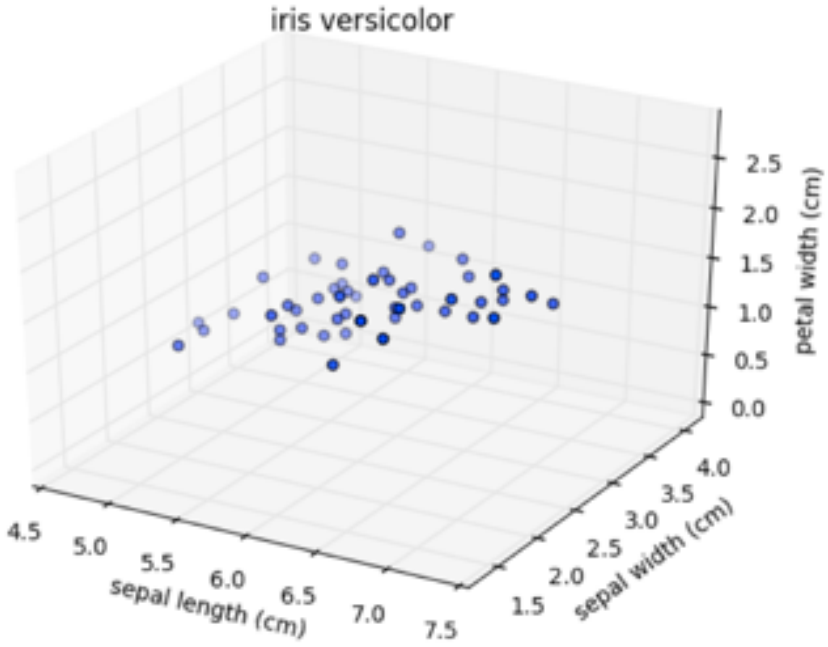
Example - PCA

```
# 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)

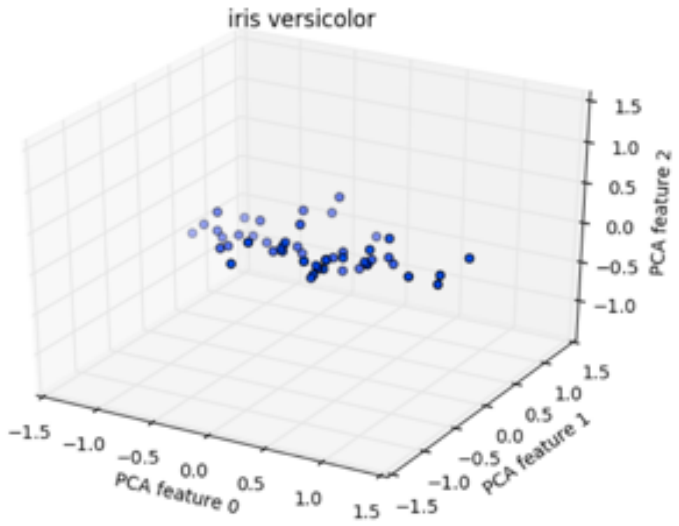
Intrinsic dimension
what does this mean?
best to describe with an example
have 2 features > longitude and latitude along a flight path
however it can be closely approximated using only one feature: the displacement along the flight path
this dataset is intrinsically one-dimensional
intrinsic dimension = number of features needed to approximate the dataset
informs dimension reduction because it tells us how much a dataset can be compressed

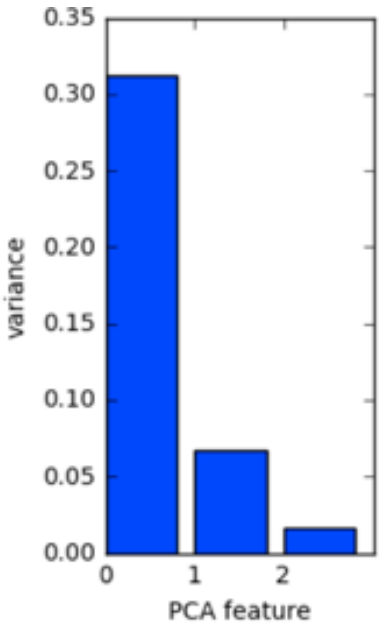
Understanding intrinsic dimension visually
example iris versicolor, only took three features, plotted on a 3D scatter plot



in this visualization you can see that most of the data points lie in a 2D sheet
*this means that the data can likely be approximated using only two features without losing much information
this example has intrinsic dimension 2

What happens when the dataset has more than 3 dimensions?
this is where PCA can be helpful
intrinsic dimension can be identified by counting the PCA features that have high or significant variance
example using the above example



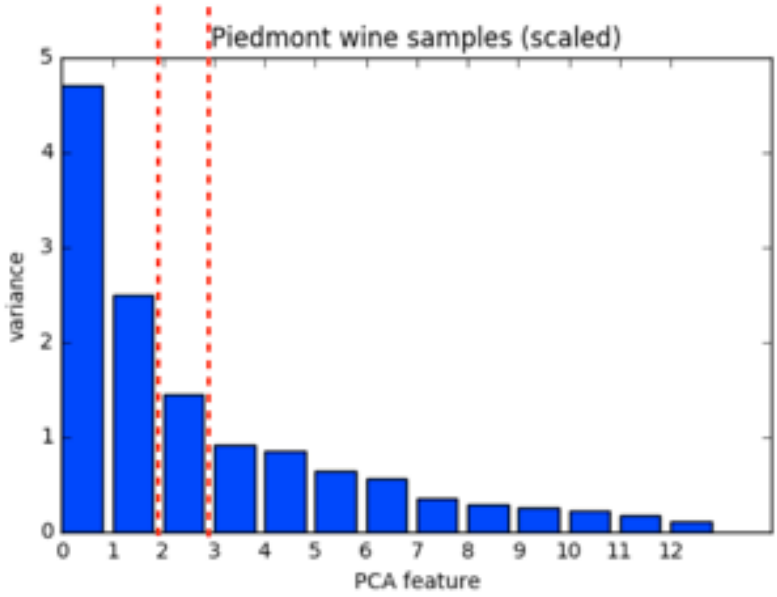


the 3D scatterplot shows the first step the decorrelation of PCA
the bar graph represents the variance from great to least
feature 2 has low variance

example

```
import matplotlib.pyplot as plt
from sklearn.decompostion import PCA
pca = PCA()
pca.fit(samples)
#create a range enumerating the PCA features
features = range(pca.n_components_)
#make a bar plot of the variances of the PCA features
plt.bar(features, pca.explained_variance_)
plt.xticks(features)
plt.ylabel('variance')
plt.xlabel('PCA feature')
plt.show()
```

intrinsic dimension is a useful idea that helps to guide dimension reduction
it is an idealization
like everything it is not always unambiguous
example



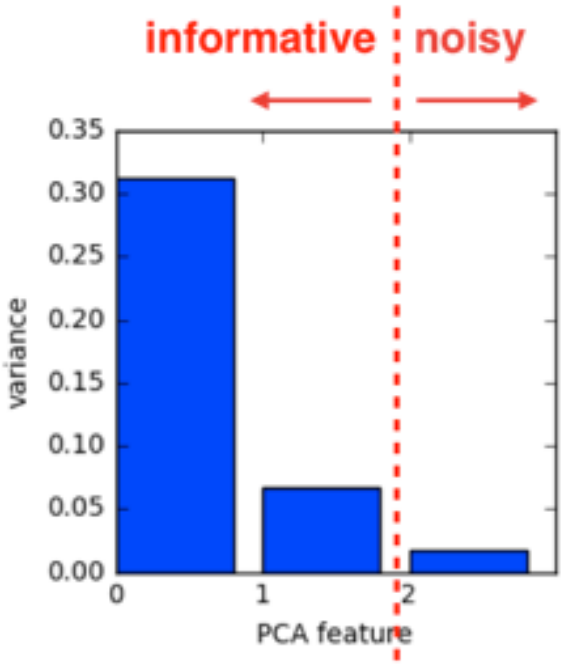
with the wine dataset and all the PCA features there could be an argument for an intrinsic dimension of 2 or 3 or 5
this all depends on the threshold you chose

example

```
# 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()
```

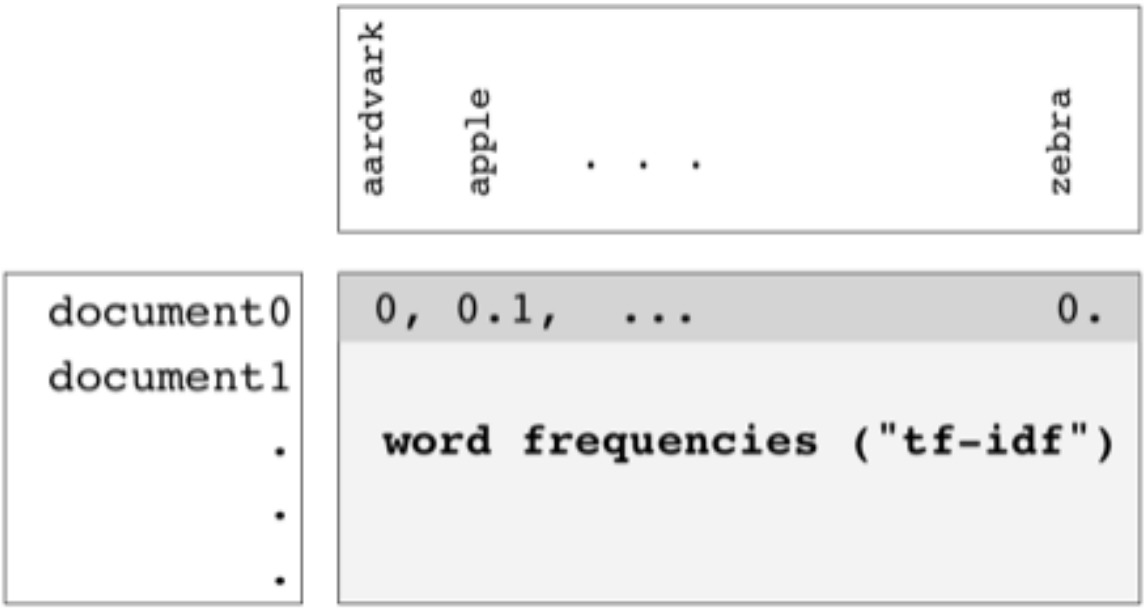
Dimension reduction with PCA

assumes the low variance features are ‘noise’
and high variance features are informative
*be aware sometimes this is not the case



Dimension reduction with PCA
specifiy how many features to keep
example PCA(n_components=2)
ideally this number is matched to the intrinsic dimension

Word frequency arrays



each row corresponds to a document
each column corresponds to a word from a fixed vocabulary
entries of the word-frequency array measure how often each word appears in each document
most often these are ‘sparse’ arrays
meaning most entries are 0
**instead of a NumPy array can use a special type of array called a ‘csr_matrix’
from scipy.sparse.csr_matrix
these save space by remembering only the non-zero entries of the array
*scikit PCA doesn’t support csr_matrix
use scikit TruncatedSVD instead
performs same transformation
*you interact with TruncatedSVD and PCA in exactly the same way
from sklearn.decomposition import TruncatedSVD
model = TruncatedSVD(n_components=3)
model.fit(documents) #documents in our csr-matrix
transformed = model.transform(documents)

Non-negative matrix factorization
NMF = ‘non-negative matrix factorization’
like PCA, is also a dimension reduction technique
unlike PCA, NMF models are interpretable
this means that NMF models are easier to understand
*key component - sample features must be ‘non-negative’, so greater than or equal to 0

NMF expresses documents as combinations of topics (or ‘themes’)
achieves its interpretability by decomposing samples as sums of their parts
decomposes documents as combinations of common themes
decomposes images as combinations of common patterns

example with tiny word-frequency array (only 4 words)
follows fit() and transform() pattern
*must specify number of components
works with NumPy arrays and sparse arrays
measure presence of words in each document using ’tf-idf’
’tf’ = frequency of word in document
‘idf’ = reduces influence of frequent words (like ‘the’)
our word-frequency array is called ‘samples’
from sklearn.decomposition import NMF
model = NMF(n_components=2)
model.fit(samples)
nmf_features = model.transform(samples)

print(model.components_)

```
[[ 0.01  0.    2.13  0.54]
 [ 0.99  1.47  0.    0.5 ]]
```

our example > creates 2 components that live in 4D space corresponding to the 4 words in the vocabulary
*dimension of components = dimension of samples
NMF like PCA creates its own features
the features and the components of an NMF model can be combined to approximately reconstruct the original data samples

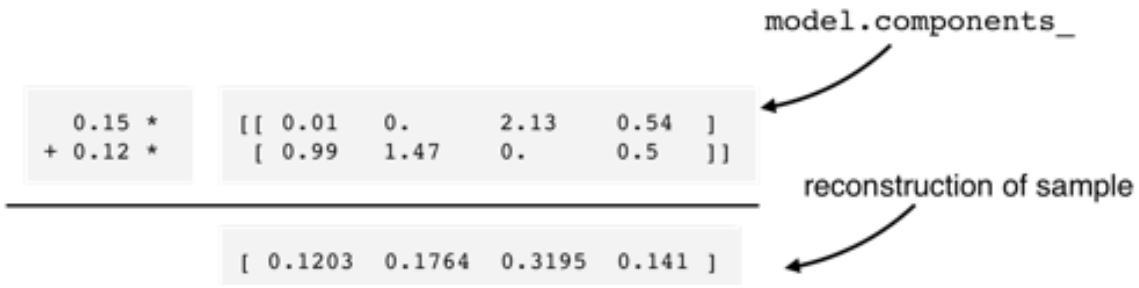
Reconstruction of a sample

```
print(samples[i,:])
```

```
[ 0.12  0.18  0.32  0.14]
```

```
print(nmf_features[i,:])
```

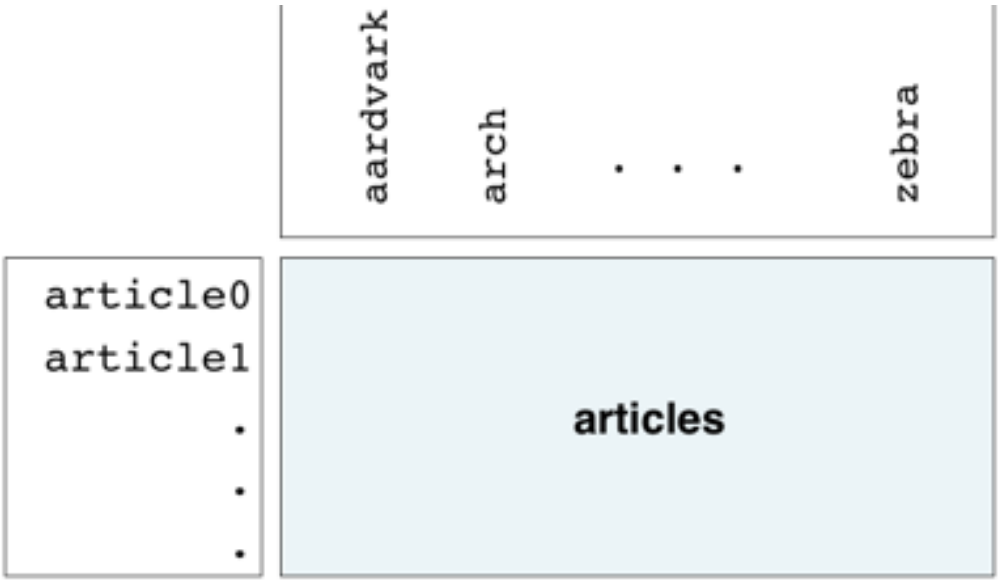
```
[ 0.15  0.12]
```



above - if we multiply each NMF components by the corresponding NMF feature value and add up each column >
we get an approximate of the original sample
*can also be expressed as a product of matrices
this is where ‘matrix factorization’ or the ‘MF’ of ‘NMF’ comes from
again remember NMR can only be applied to non-negative data
examples > word-frequency arrays, images, audio spectrograms

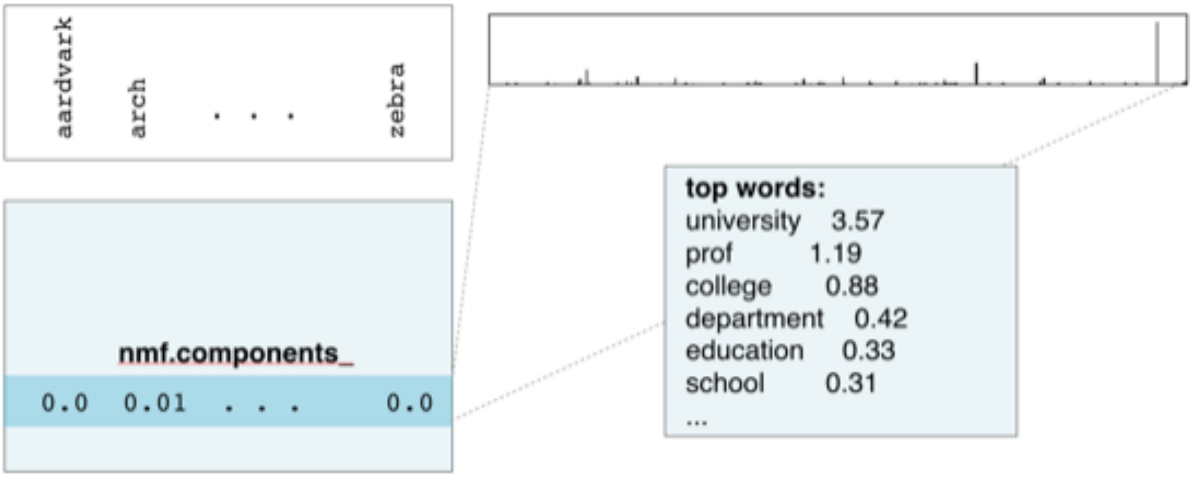
NMF learns interpretable parts
components of NMF represent patterns that frequently occur in samples

example - scientific article word-frequency array
rows = 20,000 scientific articles
columns = 800 words

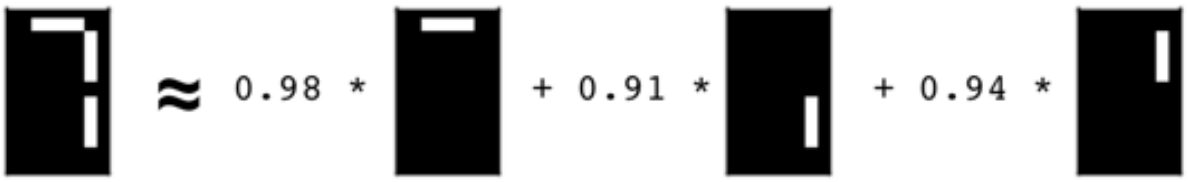


print(articles.shape)
output > (20000, 800)
from sklearn.decomposition import NMF
nmf = NMF(n_components=10)
nmf.fit(articles)
print(nmf.components_.shape)
output > (10, 800)
the 10 components are stored as the 10 rows of a 2D NumPy array

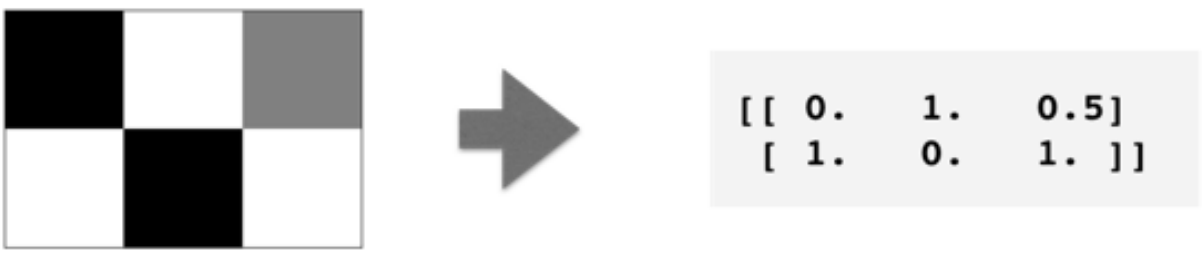
NMF components are topics
our example - rows live in an 800D space
one dimension for each of the words
aligning the words of our vocabulary with the columns of the NMF components
allows them to be interpreted



NMF components
for documents:
-NMF components represent topics
-NMF features combine topics into documents
for images:
-NMF components are parts of images
example LCD display
decomposes images from individual cells of the display



Grayscale images
only shades of gray
measures pixel brightness
represented with value between 0 and 1 (0 is black)
converts to 2D array



Grayscale images as flat arrays

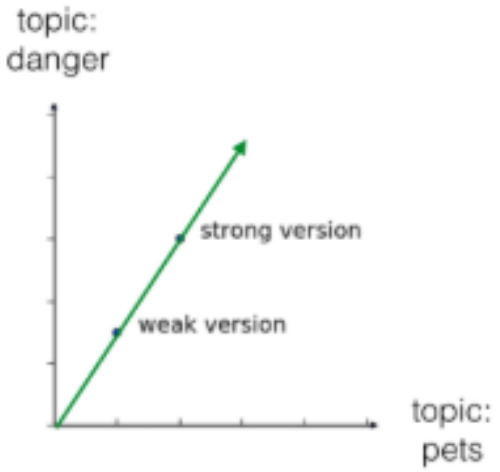
2D arrays can be ‘flattened’ by enumerating the entries
*row by row, left to right, top to bottom

Encoding a collection of images
*a collection of grayscale images of the same size can thus be encoded as a 2D array
where each row represents an image as a flattened array
and each column represents a pixel
the images are viewed as samples with pixels as the features
we can now apply NMF

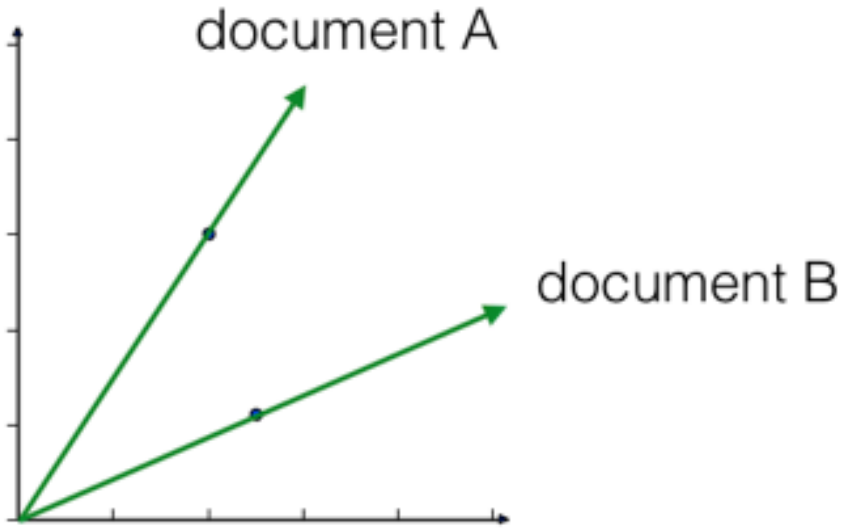
Visualizing samples
a bit complex for flattened arrays
to recover the image need to reshape the sample to the specified dimensions of the original image as a tuple
example
print(sample)
output > flat array
bitmap = sample.reshape((2,3))
print(bitmap)
output > back to a 2x3 array
from matplotlib import pyplot as plt
plt.imshow(bitmap, cmap=‘gray’, interpolation=‘nearest’)
plt.show()

Building recommender systesm using NMF
goal - recommend articles that are similar to the article currently being read
strategy
apply NMF to the word-frequency array
NMF feature values describe the topics
similar documents have similar NMF feature values
how can two articles be compared using their NMF features
example - array ‘articles’
from sklearn.decomposition import NMF

nmf = NMF(n_components=6)
nmf_features = nmf.fit_transform(articles)
versions of articles, ie similar docs have similar topics but exact feature values may and likely will be different
one may have direct language
others may add ‘meaningless chatter’
meaningless chatter reduces the frequency of the topic words overall
which reduces the values of the NMF features representing the topics
however, on a scatter plot of the NMF features, all these versions lie on a single line passing through the origin
example:



Cosine similarity
use when comparing documents
compare the origin lines
cosine similarity uses the angle between the two lines
higher values indicate greater similarity



calculate the cosine similarities
#use the normalize function
from sklearn.preprocessing import normalize
#instantiate the object and apply it to the array of all NMF features
norm_features = normalize(nmf_features)
#now select the row corresponding to the current article
#our example we will use index 23
current_article = norm_features[23,:]
pass it to the dot() method
similarities = norm_features.dot(current_article)
output is the cosine similarities

Label cosine similarities with the article titles
we do this using a DF
import pandas as pd
#normalize the NMF features
norm_features = normalize(nmf_features)
#create DF whose rows are the normalized features
#use the titles as an index
df = pd.DataFrame(norm_features, index=titles)
#use the loc method of the DF to select the normalized feature values for the current article, using its title 'Dog bites man'
current_article = df.loc['Dog bites man']
#calculate cosine similarities using the dot method
similarities = df.dot(current_article)
#use nlargest() method of the similarities Series to find the articles with the highest cosine similarity

example
# 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)
```