Lecture 4: More PCA, LSA, and NMF

Contents

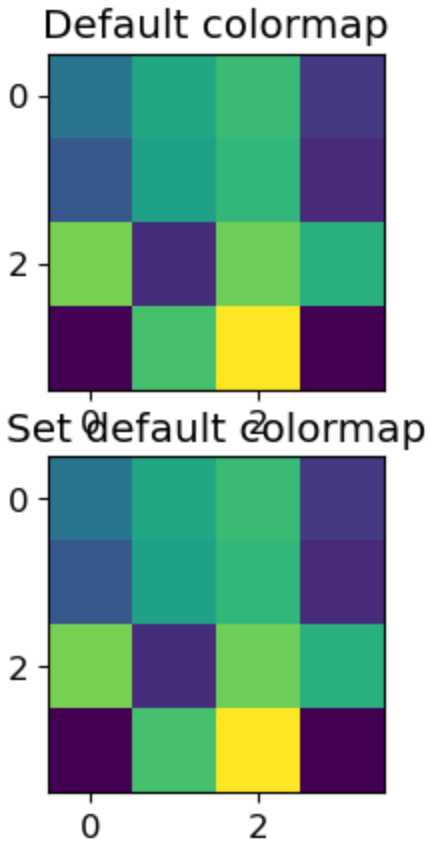
DSCI 563 Unsupervised Learning

UBC Master of Data Science program, 2024-25

Imports and learning outcomes

Imports

```
import os
import random
import sys
import time

import numpy as np

sys.path.append(os.path.join(os.path.abspath(".."), "code"))
import matplotlib.pyplot as plt
import seaborn as sns
import torch
from plotting_functions import *
from sklearn.decomposition import PCA
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler

plt.rcParams["font.size"] = 16
# plt.style.use("seaborn")
%matplotlib inline
pd.set_option("display.max_colwidth", 0)

DATA_DIR = os.path.join(os.path.abspath(".."), "data/")
```



Learning outcomes

From this lecture, students are expected to be able to:

- Explain how to write a training example as a linear combination of components, in the context of PCA;
- Use `explained_variance_ratio_` of PCA to choose k in the context of dimensionality reduction using PCA.
- Explain the geometry of PCA for different values for n and d.
- Explain similarities and differences between K-Means and PCA.
- State similarities and differences between `PCA`, `TruncatedSVD`, and `NMF`.
- Interpret components givens given by PCA, Truncated SVD and NMF.
- Recognize the scenarios for using `TruncatedSVD` and `NMF` and use them with `scikit-learn`.

1. PCA recap

A commonly used dimensionality reduction technique.

- PCA input
  - $X_{n \times d}$
  - the number of components k
- PCA output
  - $W_{k \times d}$ with k principal components (also called "parts")
  - $Z_{n \times k}$ transformed data (also called "part weights")
- Usually $k << d$.
- PCA is useful for dimensionality reduction, visualization, feature extraction etc.

Important matrices involved in PCA

PCA learns a k-dimensional subspace of the original d-dimensional space. Here are the main matrices involved in PCA.

- $X \rightarrow$ Original data matrix
- $W \rightarrow$ Principal components
  - the best lower-dimensional hyper-plane found by the PCA algorithm
- $Z \rightarrow$ Transformed data with reduced dimensionality
  - the co-ordinates in the lower dimensional space
- $X_{hat} \rightarrow$ Reconstructed data
  - reconstructions using Z and W in the original co-ordinate system

How can we use W and Z?

- Dimension reduction: compress data into limited number of components.
- Outlier detection: it might be an outlier if isn't a combination of usual components.
- Supervised learning: we could use Z as our features.
- Visualization: if we have only 2 components, we can view data as a scatterplot.
- Interpretation: we can try and figure out what the components represent.

- PCA reduces the dimensionality by learning a k-dimensional subspace of the original d-dimensional space.

- When going from higher dimensional space to lower dimensional space, PCA still tries to capture the topology of the points in high dimensional space, making sure that we are not losing some of the important properties of the data.
- So Points which are nearby in high dimensions are still nearby in low dimension.

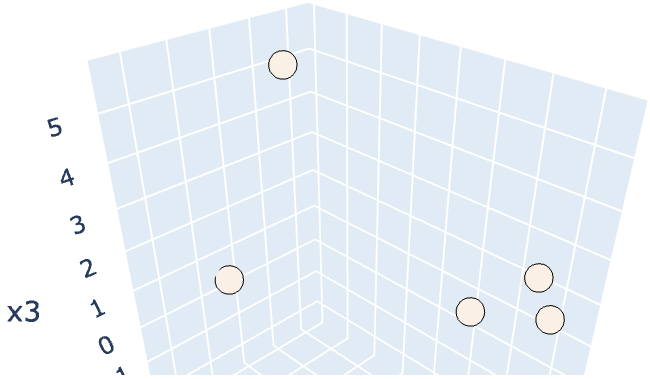
- In PCA, we find a lower dimensional subspace so that the squared error of reconstruction, i.e., elements of X and elements of ZW is minimized.
- The goal is to find the two best matrices such that when we multiply them we get a matrix that's closest to the data.
- A common way to learn PCA is using singular value decomposition (SVD).

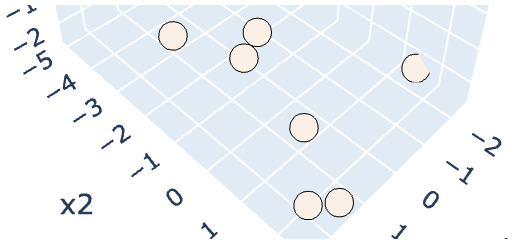
PCA examples

- Let's look at examples of reducing dimensionality using PCA from 3 to 2 and from 3 to 1.

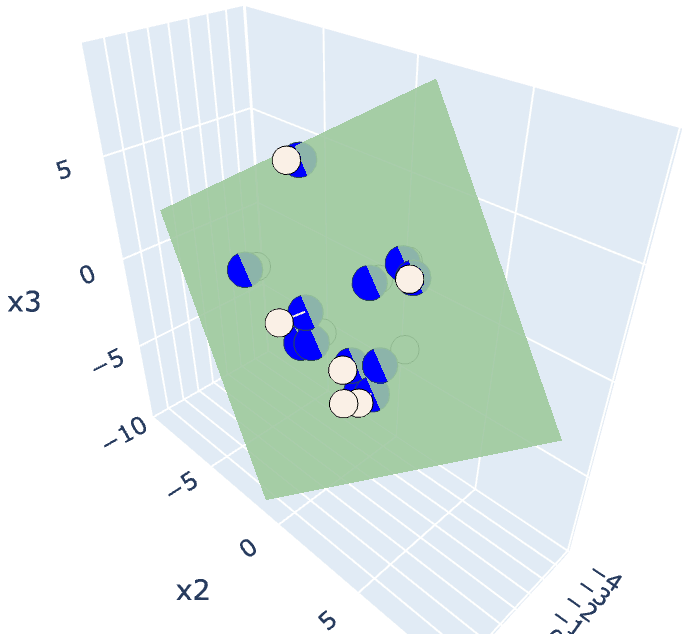
```
n = 12
d = 3

x1 = np.linspace(0, 5, n) + np.random.randn(n) * 0.05
x2 = -x1 * 0.1 + np.random.randn(n) * 2
x3 = x1 * 0.7 + np.random.randn(n) * 3

X = np.concatenate((x1[:, None], x2[:, None], x3[:, None]), axis=1)
X = X - np.mean(X, axis=0)
```

```
import plotly.express as px

plot_interactive_3d(X)
```





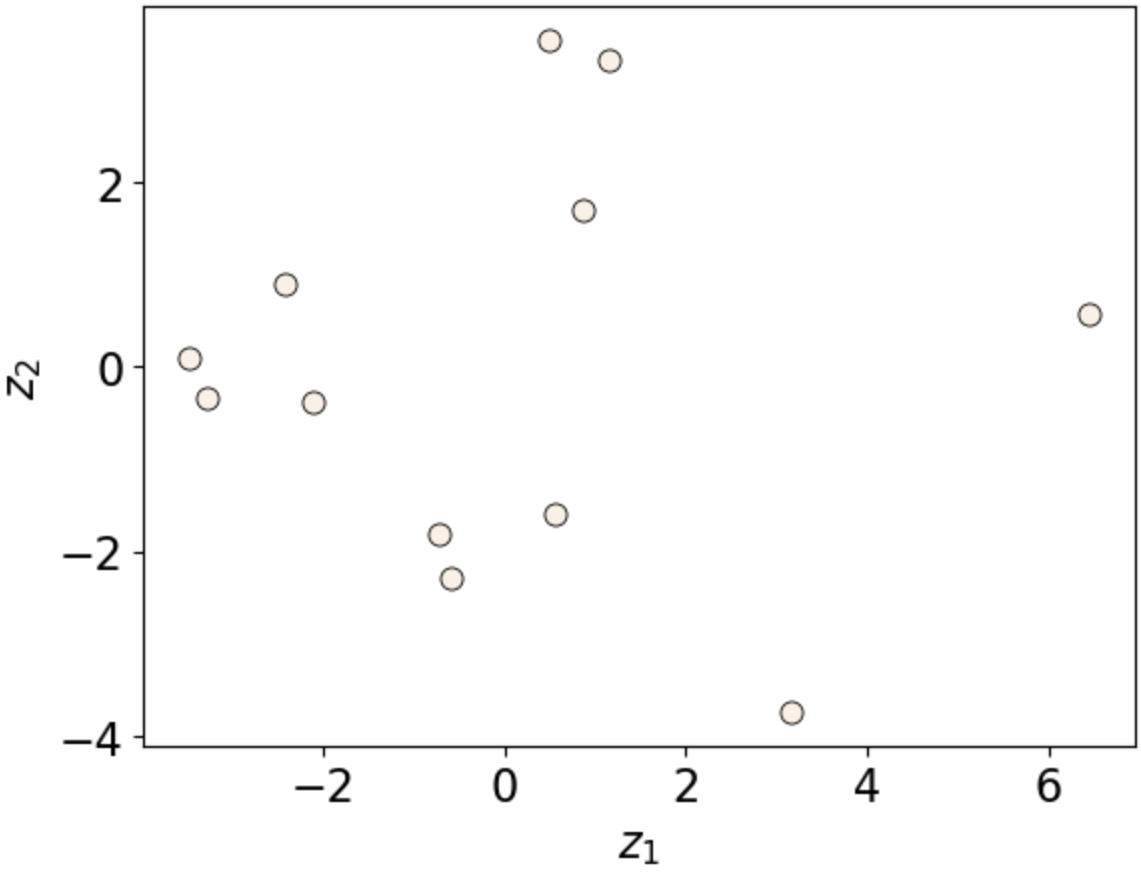

```
pca = PCA(n_components=2)
pca.fit(X)
plot_3d_2k(X, pca)
```



- The plane corresponds to W.
- Z_i are the co-ordinates of the X_i projected onto the plane.
- It gives us projected data in the new 2D coordinate system.
- X_{hat} are reconstructions in the original 3D coordinate system.

We can also plot the transformed `Z` values in two dimensions.

```
Z = pca.transform(X)
discrete_scatter(Z[:, 0], Z[:, 1])
plt.xlabel("$z_1$")
plt.ylabel("$z_2$");
```



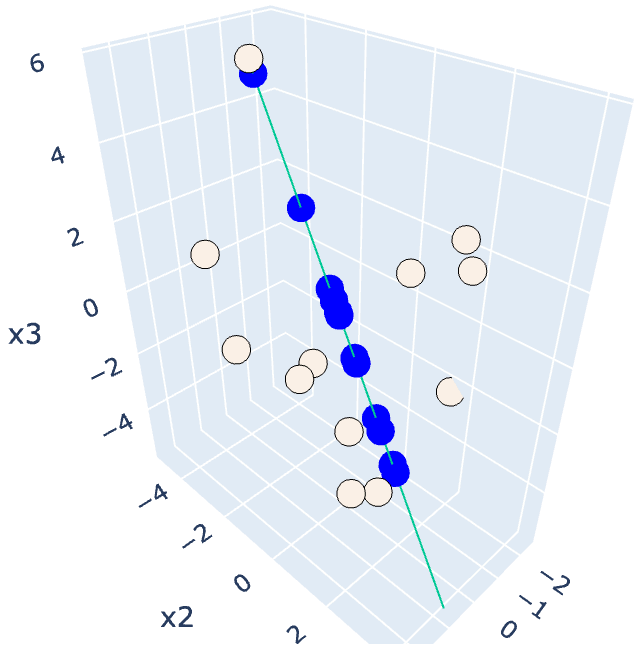
How does this relate to maximum variance view of PCA?

- To describe a two dimensional plane we need two basis vectors.
- The first basis vector is in the direction of the maximum variance of the data and the second basis vector is in the second maximum variance in the data.

Moving to $d = 3, k = 1$

- Let's reduce dimensionality from 3 dimensions to 1 dimension.
- Since $k = 1$, the W matrix defines a line.

```
pca = PCA(n_components=1)
pca.fit(X)
plot_3d_1k(X, pca)
```



- The line corresponds to the W vector.
- Z_i are the co-ordinates of the X_i projected onto the line.
- It gives us projected data in the new 1D coordinate system.
- X_{hat} are reconstructions in the original 3D coordinate system.

Parts and part weights

- In PCA terminology, the weight matrix W is often called **parts** and their corresponding values in Z are called **part weights**.
- We could represent an example as a linear combinations of components.

PCA reconstruction

- We can get $\hat{X}_{n\times d}$ (reconstructed X) by matrix multiplication of $Z_{n\times k}$ and $W_{k\times d}$. $

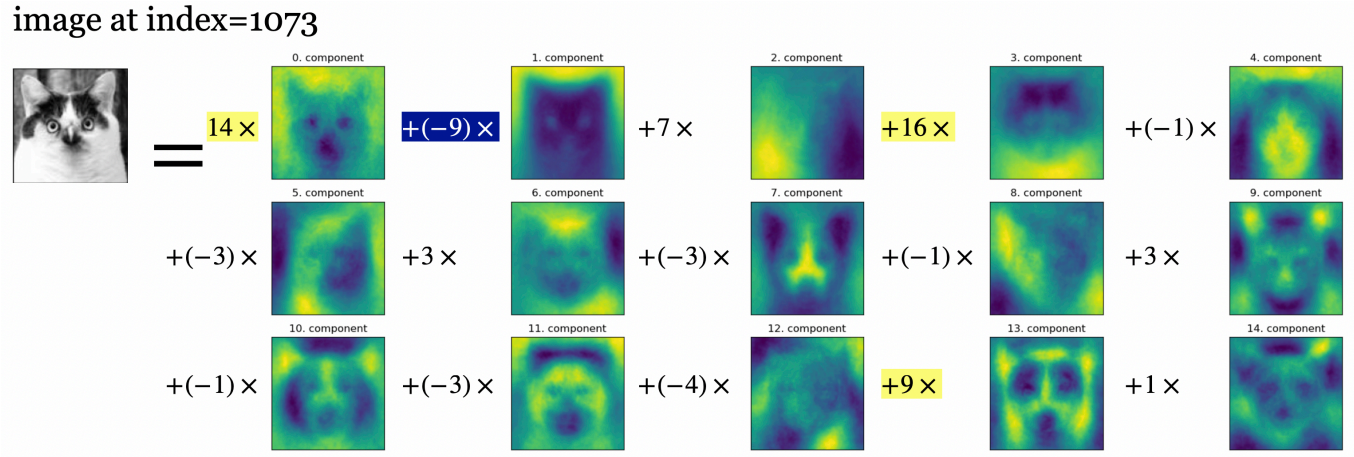
$$\hat{X}_{n\times d} = Z_{n\times k}W_{k\times d} = \begin{bmatrix} z_{11} & \dots & z_{1k} \\ z_{21} & \dots & z_{2k} \\ & \vdots & \\ z_{i1} & \dots & z_{ik} \\ & \vdots & \\ z_{n1} & \dots & z_{nk} \end{bmatrix}_{n\times k} \begin{bmatrix} w_{11} & \dots & w_{1d} \\ w_{21} & \dots & w_{2d} \\ & \vdots & \\ w_{k1} & \dots & w_{kd} \end{bmatrix}_{k\times d} \quad \$$

- For instance, you can reconstruct an example $\hat{x_i}$ as follows:

$$\hat{x_i} = \begin{bmatrix} z_{i1}w_{11} + z_{i2}w_{21} + \dots + z_{ik}w_{k1} \\ z_{i1}w_{12} + z_{i2}w_{22} + \dots + z_{ik}w_{k2} \\ \vdots \\ z_{i1}w_{1d} + z_{i2}w_{2d} + \dots + z_{ik}w_{kd} \end{bmatrix}_{d\times 1}$$

Rearranging the terms, we can see that z_i values tell us how much each principal component to use to create this example. Each z_i value act as weights in the linear combination of the principal components. A higher absolute value of z_i means that the first principal component contributes more significantly to the reconstruction of the example. Conversely, a value close to zero would indicate a minimal contribution from the first principal component to the reconstruction of that example.

- Below, I'm writing the image of this funny cat from your lab as a linear combination of principal components ("parts").
- The coefficients in Z tell us how much of each principal component to use to create this image.
- In this image, components 0, 1, 3, 13 seem to be playing an important role.



Row for index 1073 from Z

	Z_0	Z_1	Z_2	Z_3	Z_4	Z_5	Z_6	Z_7	Z_8	Z_9	Z_10	Z_11	Z_12	Z_13	Z_14
1073	14.0	-9.0	7.0	16.0	-1.0	-3.0	3.0	-3.0	-1.0	3.0	-1.0	-3.0	-4.0	9.0	1.0

2. Choosing the number of components k

- Similar to K-Means there is no definitive answer.
- But we can use some strategies to help us out:
  - Look at the total explained variance by k components.
  - Look at the reconstructions with k components.
- In PCA, principal components in W are ordered by how much variance of the data they cover.
- We can look at cumulative variance ratio (`explained_variance_ratio_` in `sklearn`) to select how many components we want.
- When the data is human interpretable (e.g., images), you can also look at the reconstructions to help make this decision.
- When we train a PCA model, it returns the amount of variance explained by each of the components.
- In `sklearn`, we can access this using the `explained_variance_` attribute of the pca object.

- And we can access the percentage of variance explained by each of the selected components with `explained_variance_ratio_`
- Let's explore this on our toy pizza example.

```
pizza_df = pd.read_csv(DATA_DIR + "Pizza.csv")
pizza_df.head()
```

	brand	id	mois	prot	fat	ash	sodium	carb	cal
0	A	14069	27.82	21.43	44.87	5.11	1.77	0.77	4.93
1	A	14053	28.49	21.26	43.89	5.34	1.79	1.02	4.84
2	A	14025	28.35	19.99	45.78	5.08	1.63	0.80	4.95
3	A	14016	30.55	20.15	43.13	4.79	1.61	1.38	4.74
4	A	14005	30.49	21.28	41.65	4.82	1.64	1.76	4.67

```
X_pizza = pizza_df.drop(columns=["id", "brand"])
y_pizza = pizza_df["brand"]
X_pizza.head()
```

	mois	prot	fat	ash	sodium	carb	cal
0	27.82	21.43	44.87	5.11	1.77	0.77	4.93
1	28.49	21.26	43.89	5.34	1.79	1.02	4.84
2	28.35	19.99	45.78	5.08	1.63	0.80	4.95
3	30.55	20.15	43.13	4.79	1.61	1.38	4.74
4	30.49	21.28	41.65	4.82	1.64	1.76	4.67

```
n_components = 2
pipe_pca = make_pipeline(StandardScaler(), PCA(n_components=n_components))
pipe_pca.fit(X_pizza)
W = pipe_pca.named_steps["pca"].components_
```

```
pipe_pca.named_steps["pca"].explained_variance_
```

```
array([4.18573434, 2.29811778])
```

```
pipe_pca.named_steps["pca"].explained_variance_ratio_
```

```
array([0.59596884, 0.3272082 ])
```

```
pipe_pca.named_steps["pca"].explained_variance_ratio_.sum()
```

```
np.float64(0.9231770406002876)
```

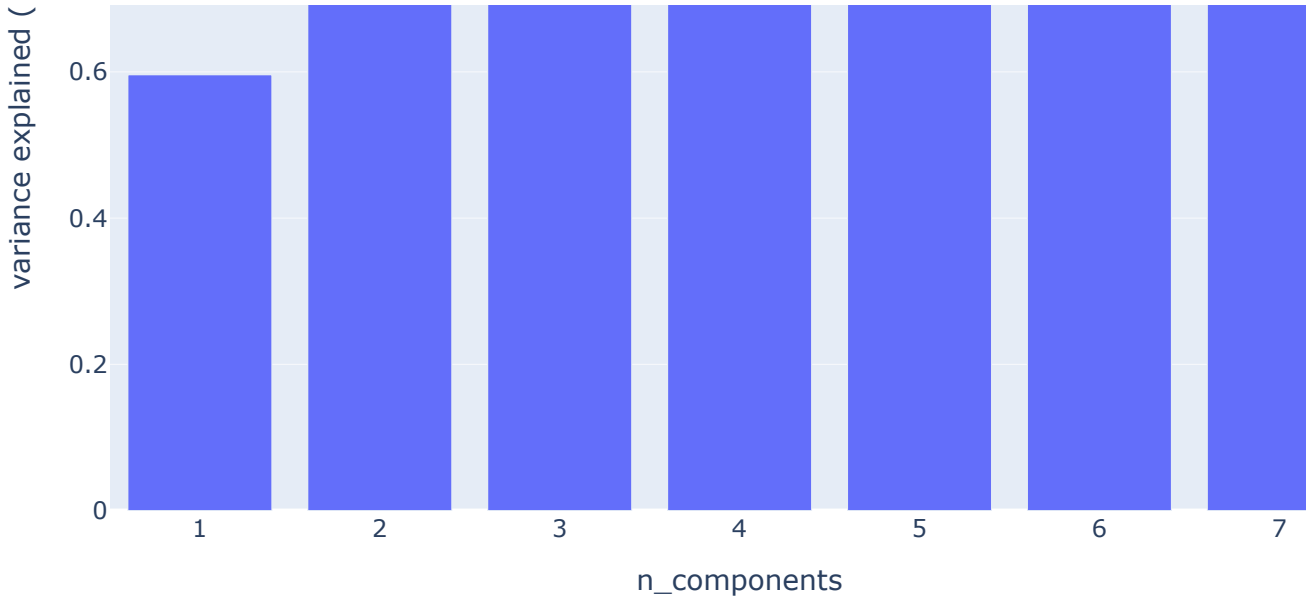
We are capturing 92.31% of the information using only two of these newly created features!!

Let's look at how much "information" we can capture with different number of components.

```
n_components = 7
pipe_pca = make_pipeline(StandardScaler(), PCA(n_components=n_components))
pipe_pca.fit(X_pizza)
W = pipe_pca.named_steps["pca"].components_
```

```
df = pd.DataFrame(
    data=np.cumsum(pipe_pca["pca"].explained_variance_ratio_),
    columns=["variance_explained (%)"],
    index=range(1, n_components + 1),
)
df.index.name = "n_components"
```

```
simple_bar_plot(
    x=df.index.tolist(),
    y=df["variance_explained (%)"],
    x_title="n_components",
    y_title="variance explained (%)",
)
```




```
pipe_pca.named_steps["pca"].explained_variance_
```

```
array([4.18573434e+00, 2.29811778e+00, 4.15948838e-01, 9.54925358e-02,
       2.77695834e-02, 3.38738483e-04, 9.55061572e-06])
```

Here is the percentage variance explained by the model.

```
pipe_pca.named_steps["pca"].explained_variance_ratio_.sum()
```

```
np.float64(1.0)
```

The sum of explained variance ratio covered by all components should be close to 1.

Let's try it on a larger dataset. Let's bring back the human face images dataset, a small subset of Human Faces dataset (available here).

```

```

```
import numpy as np
import pandas as pd
import random
import os
import torch
from torchvision import datasets, models, transforms, utils
from PIL import Image
from torchvision import transforms

from torchvision.models import vgg16
import matplotlib.pyplot as plt
```

```
mpl.rcParams.update(mpl.rcParamsDefault)
plt.rcParams["image.cmap"] = "gray"
```

```
import torchvision
```

```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
```

```
def set_seed(seed=42):
    torch.manual_seed(seed)
    np.random.seed(seed)
    random.seed(seed)
```

```
set_seed(seed=42)
```

```

```

```
import glob
IMAGE_SIZE = 200
def read_img_dataset(data_dir):
    data_transforms = transforms.Compose(
        [
            transforms.Resize((IMAGE_SIZE, IMAGE_SIZE)),
            transforms.Grayscale(num_output_channels=1),
            transforms.ToTensor(),
            transforms.Lambda(torch.flatten)
        ])

    image_dataset = datasets.ImageFolder(root=data_dir, transform=data_transfo
    dataloader = torch.utils.data.DataLoader(
         image_dataset, batch_size=BATCH_SIZE, shuffle=True, num_workers=0
    )
    dataset_size = len(image_dataset)
    class_names = image_dataset.classes
    inputs, classes = next(iter(dataloader))
    return inputs, classes
```

```
image_shape = (IMAGE_SIZE, IMAGE_SIZE)
```

```
def plot_sample_bw_imgs(inputs):
    fig, axes = plt.subplots(1, 5, figsize=(10, 8), subplot_kw={"xticks": (),
    for image, ax in zip(inputs, axes.ravel()):
        ax.imshow(image.reshape(image_shape))
    plt.show()
```

```
data_dir = DATA_DIR + "people"
file_names = [image_file for image_file in glob.glob(data_dir + "/*/*.jpg")]
n_images = len(file_names)
BATCH_SIZE = n_images  # because our dataset is quite small
faces_inputs, classes = read_img_dataset(data_dir)
```

```
X_faces = faces_inputs.numpy()
X_faces.shape
```

```
(367, 40000)
```

```
plot_sample_bw_imgs(X_faces[40:])
```

Let's try 200 components.

```
n_components = 200
pca = PCA(n_components=n_components, whiten=True, random_state=0)
pca.fit(X_faces)
```

```
PCA(n_components=200, random_state=0, whiten=True)
```

Let's visualize the first few components.

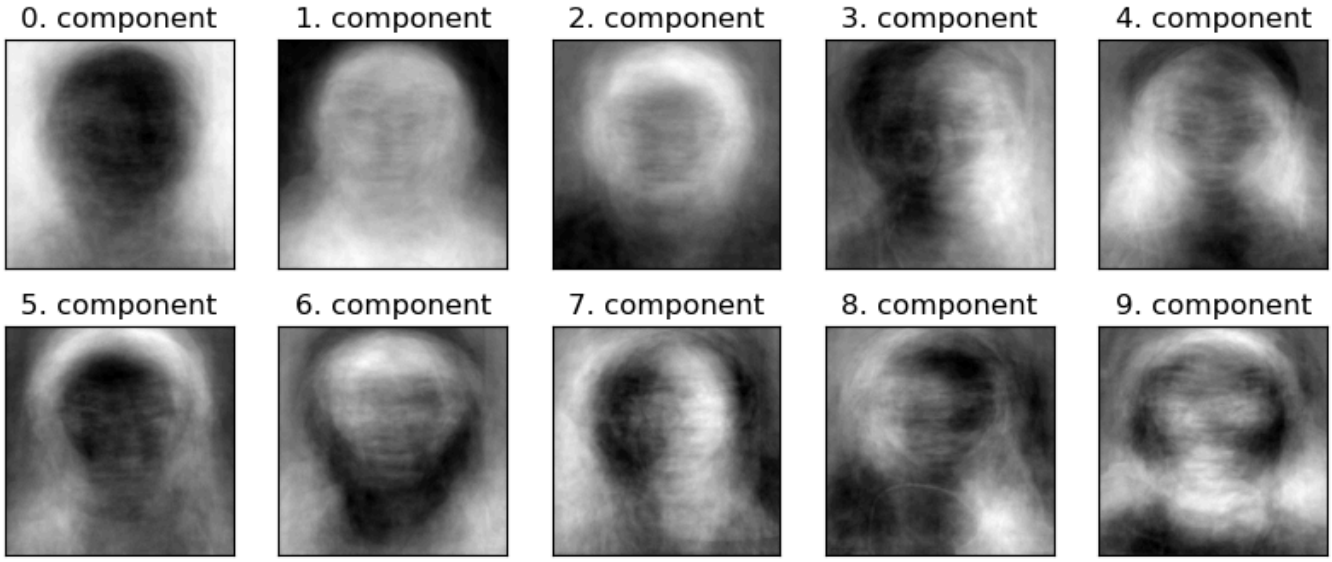
```
Z = pca.transform(X_faces)  # Transform the data
W = pca.components_  # principal components
X_hat = pca.inverse_transform(Z)  # reconstructions
```

```
image_shape
```

```
(200, 200)
```

```
def plot_components(W):
    fig, axes = plt.subplots(2, 5, figsize=(10, 4), subplot_kw={"xticks": (),
    for i, (component, ax) in enumerate(zip(W, axes.ravel())):
        ax.imshow(component.reshape(image_shape))
        ax.set_title("{}. component".format((i)))
    plt.show()
```

```
plot_components(W)
```



How much variance is covered by the first 200 components?

```
df = pd.DataFrame(
    data=np.cumsum(pca.explained_variance_ratio_),
    columns=["cummulative variance_explained (%)"],
    index=range(1, n_components + 1),
)
df.index.name = "n_components"
df
```

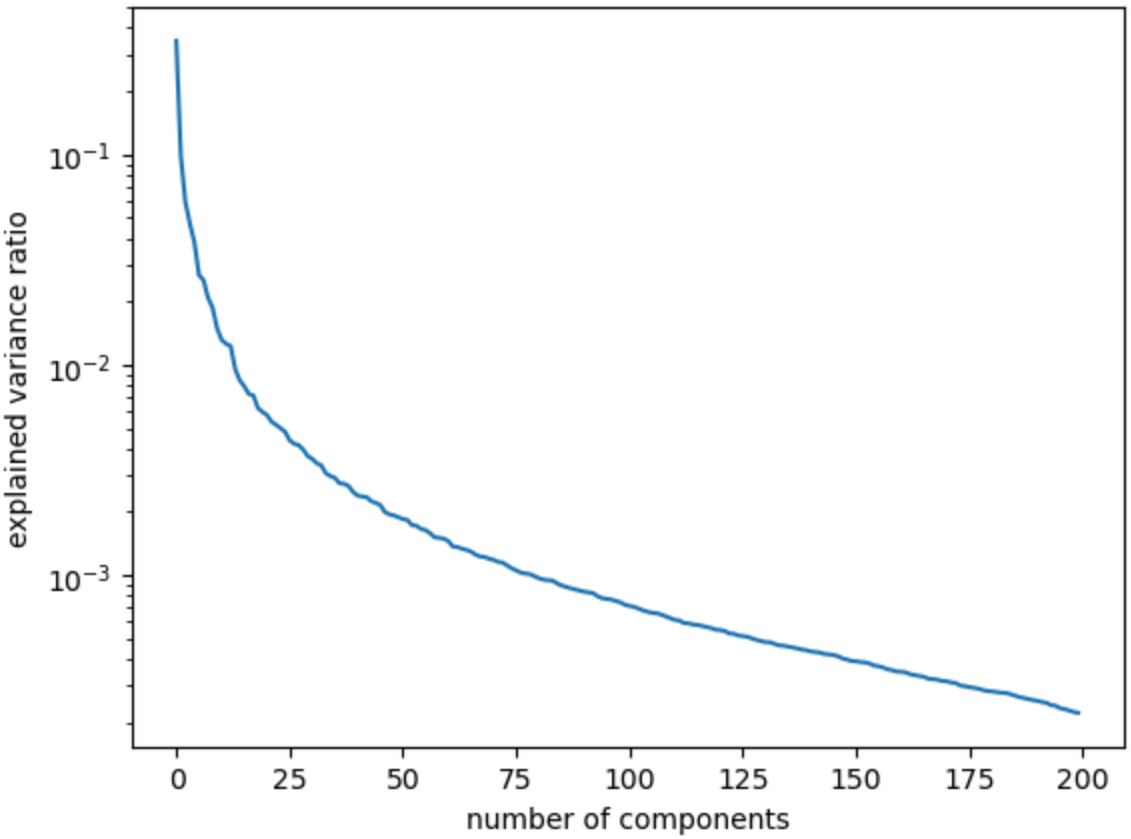
n_components	cummulative variance_explained (%)
1	0.346940
2	0.445209
3	0.504754
4	0.551640
5	0.589835
...	...
196	0.982022
197	0.982252
198	0.982479
199	0.982702
200	0.982924

200 rows × 1 columns

It doesn't make sense to plot a bar chart here, as we have so many components.

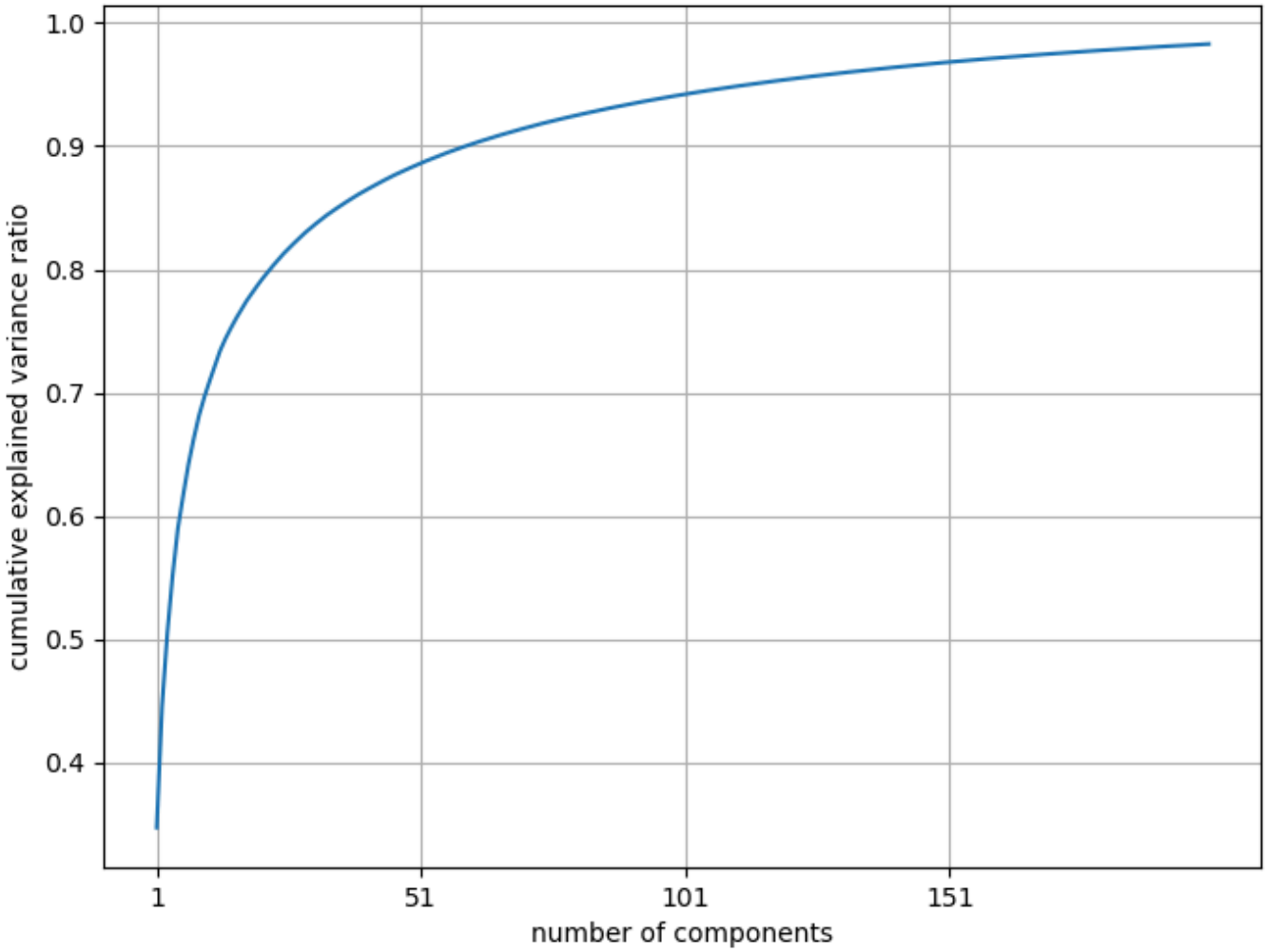
How about plotting number of components vs. explained variance?

```
plt.semilogy(pca.explained_variance_ratio_)
plt.xlabel("number of components")
plt.ylabel("explained variance ratio");
plt.show()
```



A more useful plot is number of components vs. cumulative explained variance ratio.

```
plt.figure(figsize=(8, 6))
plt.plot(range(1, 201), np.cumsum(pca.explained_variance_ratio_))
plt.xticks(range(1, 201, 50))
plt.xlabel("number of components")
plt.ylabel("cumulative explained variance ratio")
plt.grid();
plt.show()
```



Seems like we are capturing ~90% variance with only ~50 components! Remember that our dimensionality in the original dataset is 200 by 200 = 40000.

How do we choose the best value for the number of components? This will be application dependent.

3. Miscellaneous PCA-related things

3.1 PCA and multicollinearity

- How do you think multicollinearity would affect PCA?

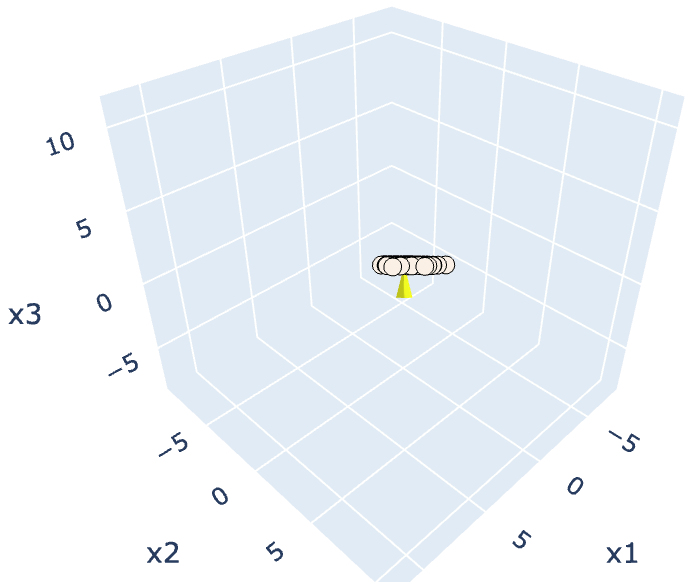
```
from scipy.stats import multivariate_normal, norm
X_mc = multivariate_normal.rvs(
    mean=[0, 0], cov=[[12, 11.5], [11.5, 12]], size=200, random_state=42
)

X_mc = np.column_stack((X_mc, np.sum(X_mc,axis=1)/2)) # Create a correlated fe
```

```
all((X_mc[:, 0] + X_mc[:, 1])/2 == X_mc[:, 2])
```

```
True
```

Do we really need the third dimension to describe the data? How would PCA handle this colinearity?

```
pca = PCA(n_components=3).fit(X_mc)
plot_multicollinearity_3d(X_mc, pca)
```



What's the variance associated with each component?

```
pca.explained_variance_
```

```
array([3.23380503e+01, 4.65171412e-01, 1.75508741e-15])
```

The variance associated third component is very close to zero.

```
np.allclose(pca.explained_variance_[2], np.zeros(1))
```

```
True
```

So when we train PCA on $X_{n \times d}$ with multicolinearity,

- We will get d orthogonal vectors.
- But the variance explained in some of the components would be zero.

(Optional) The relationship between the singular values and explained variance

```
singular_vals = pca.singular_values_
singular_vals
```

```
array([8.02201472e+01, 9.62128427e+00, 5.90984260e-07])
```

The relationship between the singular value σ_i and the explained variance is

$$\text{explained variance of } i^{th} \text{ component} = \frac{\sigma_i^2}{N-1}$$

- $N \rightarrow$ the number of data points

Let's get the explained variance by manipulating the singular values.

In our case, N=200. So the explained variance can be calculated as:

```
(pca.singular_values_)**2/199
```

```
array([3.23380503e+01, 4.65171412e-01, 1.75508741e-15])
```

These values are exactly the same as explained variance.

```
pca.explained_variance_
```

```
array([3.23380503e+01, 4.65171412e-01, 1.75508741e-15])
```

3.2 PCA and outliers

- PCA may or may not maintain outliers after applying transformation.
- In the anomaly detection application, our assumption is that PCA is not able to maintain outliers and hence the reconstruction error for them would be higher compared to normal examples.

> Robust PCA is more appropriate for anomaly detection task. We're not using it in lab because it's not implemented in `sklearn`. That said there are some external packages available. For example see this implementation of robust PCA. Also, `sklearn` might have it sometime soon. See here.

3.3 K-Means and PCA

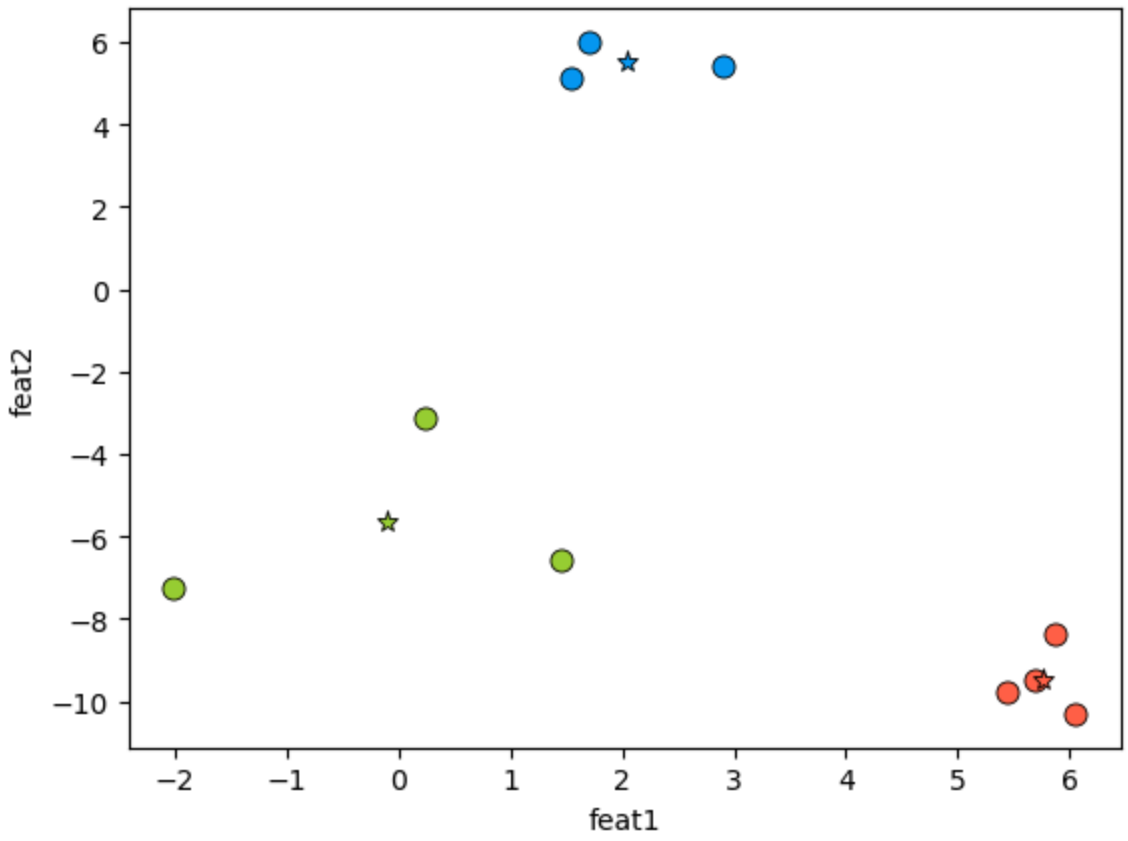
- You can think of PCA as generalization of K-Means.
- In case of K-Means we learn a set of cluster centers and memorize these K centers.
- We represent the dataset in terms of those centers.
- W in K-Means has K d-dimensional cluster centers.

- Z_i tells us which center the point belongs to. Each row of Z has one element equal to one and the other are zero, indicating the cluster of that row.

```
X, y = make_blobs(n_samples=10, centers=3, n_features=2, random_state=10)
pd.DataFrame(X, columns=["feat1", "feat2"])
```

	feat1	feat2
0	5.691924	-9.476412
1	1.707899	6.004352
2	0.236210	-3.119100
3	2.901595	5.421215
4	5.859439	-8.381924
5	6.047749	-10.305047
6	-2.007588	-7.247439
7	1.454677	-6.583872
8	1.536362	5.111215
9	5.430704	-9.759561

```
from sklearn.cluster import KMeans
n_clusters = 3
km = KMeans(n_clusters=n_clusters, n_init='auto')
km.fit(X)
centers=km.cluster_centers_
cluster_labels=km.labels_
discrete_scatter(X[:, 0], X[:, 1], cluster_labels, markers="o")
discrete_scatter(centers[:, 0], centers[:, 1], [0,1,2], markers="*")
plt.xlabel("feat1")
plt.ylabel("feat2")
plt.show()
```



You can think of cluster centers as the W matrix of K-Means

```
W_km = km.cluster_centers_
pd.DataFrame(W_km, columns=["feat1", "feat2"])
```

	feat1	feat2
0	2.048619	5.512261
1	-0.105567	-5.650137
2	5.757454	-9.480736

You can think of the hard cluster assignments as part weights associated with components.

```
Z_km = np.zeros((len(X), n_clusters), dtype="int32")
for i in range(len(X)):
    Z_km[i, cluster_labels[i]] = 1

col_names = ["center" + str(i) for i in range(n_clusters)]
pd.DataFrame(Z_km, columns=col_names)
```

	center0	center1	center2
0	0	0	1
1	1	0	0
2	0	1	0
3	1	0	0
4	0	0	1
5	0	0	1
6	0	1	0
7	0	1	0
8	1	0	0
9	0	0	1

- So we decompose X as $$X_{(n \times d)} \approx Z_{(n \times k)} W_{(k \times d)}$$
- You can think of reconstruction of each point as $\hat{X}_i = Z_i W$
- The reconstruction error would be high because we are capturing all data points in terms of just K cluster centers.
- Z in not continuous.
- PCA is a generalization that allows continuous values of Z_is.
- Does it make sense to `fit` a PCA model here with `n_components=3`?

❓❓ Questions for you

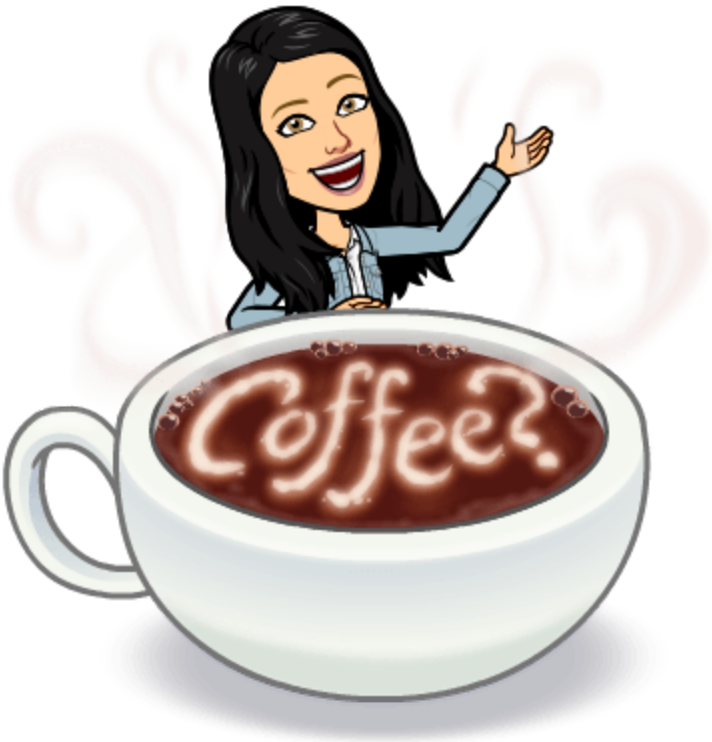
4.1 Select all of the following statements which are **True** (iClicker)

- (A) Each column in Z represents the extent to which the corresponding principal component contributes to each example in the transformed space.

- (B) A larger magnitude of Z_{i1} in the transformed data Z indicates that component 1 has a strong influence on the reconstruction of example $\hat{x}_i$.
- (C) In K-Means clustering, each example is assigned to exactly one cluster, whereas in PCA, each example is represented as a linear combination of all principal components.
- (D) If two features in the original dataset are perfectly collinear, PCA would only be able to find $d - 1$ meaningful principal components, as the remaining component(s) will capture zero variance.
- (E) The principal components found by PCA are statistically uncorrelated.

V's Solutions!

Break



4. Varieties/extensions of PCA

4.1 PCA extensions

There are a number of PCA generalizations and extensions. Here are some examples.

- TuncatedSVD or Latent Semantic Analysis (LSA)
- Non negative matrix factorization (NMF)
- Sparse PCA or Mini-batch sparse PCA
- KernelPCA
- IncrementalPCA
- Robust PCA (not implemented in sklearn yet)
- ...

In the next sections we'll see examples of `LSA` and NMF.

4.2 Latent Semantic Analysis (LSA)

- Similar to PCA, but it **does not center the data** before applying SVD.
- Particularly useful for extracting semantically meaningful components from high-dimensional sparse data.
- Why not PCA?
    - In such cases, subtracting the mean is problematic because it destroys sparsity, forcing a dense representation that may be impractical to store or process efficiently.
- Useful in the context of text data because BOW representation of text data is usually sparse and high dimensional.
- This is also referred to as Latent Semantic Indexing (LSI) in the context of information retrieval.
- In `scikit-learn` the way to do this is using `TruncatedSVD`.
    - Contrary to PCA, this algorithm does not center the data before calling `SVD`.
    - So it's suitable for sparse, high-dimensional matrices (e.g., features extracted by `CountVectorizer`).

- You can think of LSA as a tool for extracting semantic features or "topics" from a given set of documents.
- It creates a dense representation of documents.

$$X_{n \times d} \approx ZW$$

- Let's get an intuition for LSA with a toy example.

```
import wikipedia
from nltk.tokenize import sent_tokenize, word_tokenize

queries = [
    "raspberry",
    "pineapple juice",
    "mango fruit",
    "Vancouver",
    "London",
    "Toronto",
]

wiki_dict = {"wiki query": [], "text": [], "n_words": []}
for i in range(len(queries)):
    sent = sent_tokenize(wikipedia.page(queries[i]).content)[0]
    wiki_dict["text"].append(sent)
    wiki_dict["n_words"].append(len(word_tokenize(sent)))
    wiki_dict["wiki query"].append(queries[i])

mixed_sent = "Mango and pineapple are tropical. It's fun to eat fresh fruit in
wiki_dict["text"].append(mixed_sent)
wiki_dict["n_words"].append(len(word_tokenize(mixed_sent)))
wiki_dict["wiki query"].append('mixed')

wiki_df = pd.DataFrame(wiki_dict)
wiki_df
```

	wiki query	text	n_words
0	raspberry	The raspberry is the edible fruit of several plant species in the genus Rubus of the rose family, most of which are in the subgenus Idaeobatus.	28
1	pineapple juice	Pineapple juice is a juice made from pressing the natural liquid out from the pulp of the pineapple (a fruit from a tropical plant).	27
2	mango fruit	A mango is an edible stone fruit produced by the tropical tree Mangifera indica.	15
3	Vancouver	Vancouver is a major city in Western Canada, located in the Lower Mainland region of British Columbia.	19
4	London	London is the capital and largest city of both England and the United Kingdom, with a population of 8,866,180 in 2022.	23
5	Toronto	Toronto is the most populous city in Canada and the capital city of the Canadian province of Ontario.	19
6	mixed	Mango and pineapple are tropical. It's fun to eat fresh fruit in the city of Vancouver.	19

Let's get BOW representation of the documents.

```
from sklearn.decomposition import TruncatedSVD
from sklearn.feature_extraction.text import CountVectorizer

cv = CountVectorizer(stop_words="english")
bow = cv.fit_transform(wiki_df["text"]).toarray()
bow_df = pd.DataFrame(bow, columns=cv.get_feature_names_out(), index=wiki_df["
bow_df
```

	180	2022	866	british	canada	canadian	capital	city	columbi
text									
The raspberry is the edible fruit of several plant species in the genus Rubus of the rose family, most of which are in the subgenus Idaeobatus.	0	0	0	0	0	0	0	0	0
Pineapple juice is a juice made from pressing the natural liquid out from the pulp of the pineapple (a fruit from a tropical plant).	0	0	0	0	0	0	0	0	0
A mango is an edible stone fruit produced by the tropical tree Mangifera indica.	0	0	0	0	0	0	0	0	0
Vancouver is a major city in Western Canada, located in the Lower Mainland	0	0	0	1	1	0	0	1	

	180	2022	866	british	canada	canadian	capital	city	columbi
text									
region of British Columbia.									
London is the capital and largest city of both England and the United Kingdom, with a population of 8,866,180 in 2022.	1	1	1	0	0	0	1	1	(
Toronto is the most populous city in Canada and the capital city of the Canadian province of Ontario.	0	0	0	0	1	1	1	2	(
Mango and pineapple are tropical. It's fun to eat fresh fruit in the city of Vancouver.	0	0	0	0	0	0	0	1	(

7 rows × 53 columns

```
bow_df.shape
```

```
(7, 53)
```

- Each document is represented with 68 features, i.e., words.

- Probably there is a set of features related to sports (e.g., *team, sport, ice, hockey, outdoor, played, skating*) and a set of features related to fruit (e.g., *fruit, juice, mango, tree, tropical*)?
- Can we extract these **latent features**, which are there in the data but haven't manifested yet?

```
lsa_pipe = make_pipeline(
    CountVectorizer(stop_words="english"), TruncatedSVD(n_components=2)
)

lsa_transformed = lsa_pipe.fit_transform(wiki_df["text"]);
```

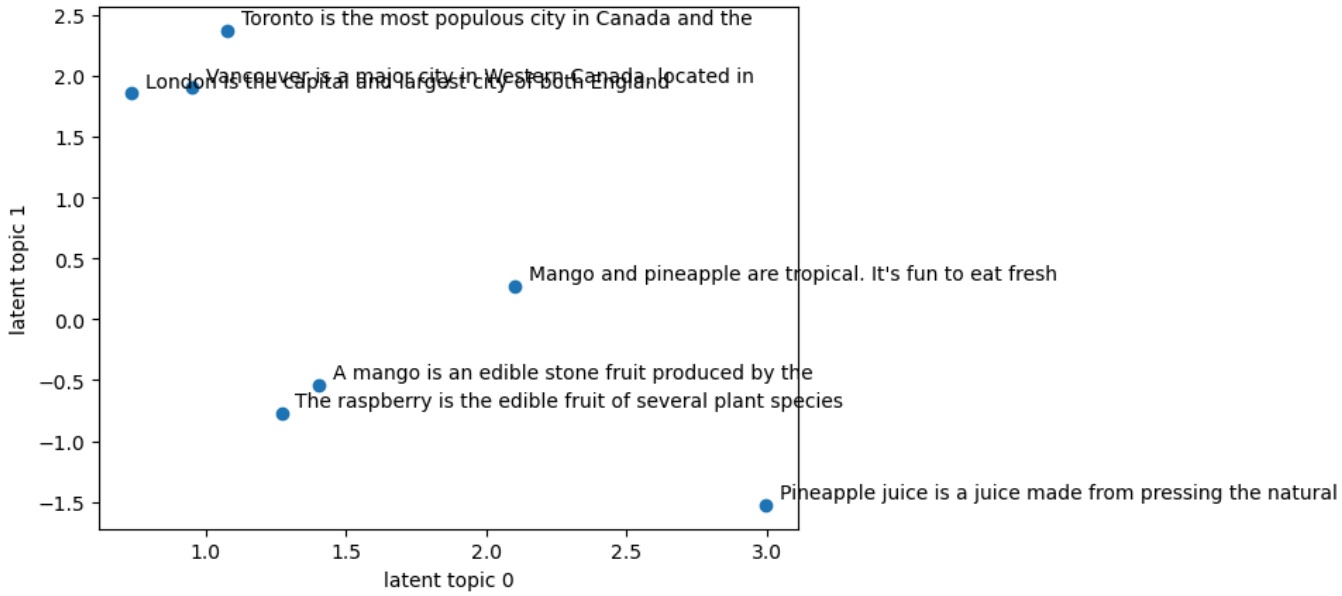
```
pd.DataFrame(
    np.round(lsa_transformed, 4),
    columns=["latent topic 0", "latent topic 1"],
    index=wiki_df["text"],
)
```

text	latent topic 0	latent topic 1
The raspberry is the edible fruit of several plant species in the genus Rubus of the rose family, most of which are in the subgenus Idaeobatus.	1.2696	-0.7712
Pineapple juice is a juice made from pressing the natural liquid out from the pulp of the pineapple (a fruit from a tropical plant).	2.9979	-1.5255
A mango is an edible stone fruit produced by the tropical tree Mangifera indica.	1.4019	-0.5387
Vancouver is a major city in Western Canada, located in the Lower Mainland region of British Columbia.	0.9514	1.9055
London is the capital and largest city of both England and the United Kingdom, with a population of 8,866,180 in 2022.	0.7322	1.8597
Toronto is the most populous city in Canada and the capital city of the Canadian province of Ontario.	1.0778	2.3706
Mango and pineapple are tropical. It's fun to eat fresh fruit in the city of Vancouver.	2.1023	0.2748

- Latent topic 0 seems to be dominant in the fruit-related documents.

- Latent topic 1 seems to be dominant in city-related documents.

```
plt.scatter(lsa_transformed[:, 0], lsa_transformed[:, 1])
plt.xlabel("latent topic 0")
plt.ylabel("latent topic 1")
x = lsa_transformed[:, 0]
y = lsa_transformed[:, 1]
for i, txt in enumerate(wiki_df["text"]):
    plt.annotate(" ".join(txt.split()[:10]), (x[i] + 0.05, y[i] + 0.05))
plt.show()
```



Let's examine the components learned by LSA.

- How much variance is covered by these two components?

```
lsa_pipe.named_steps["truncatedsvd"].explained_variance_ratio_.sum()
```

```
np.float64(0.30575326427035876)
```

Good to know!

- Our features are word counts.
- Which features have higher weights in latent topic 0 vs. latent topic 1?

What are the most important words for these latent topics?

```
sorting = np.argsort(lsa_pipe.named_steps["truncatedsvd"].components_, axis=1)

feature_names = np.array(
    lsa_pipe.named_steps["countvectorizer"].get_feature_names_out()
)
print_topics(
    topics=[0, 1], feature_names=feature_names, sorting=sorting, n_words=20
)
```

```
topic 0       topic 1
--------      --------
pineapple     city
fruit         canada
tropical      capital
juice         province
city          canadian
plant         toronto
mango         ontario
vancouver     populous
natural       vancouver
pulp          western
liquid        columbia
pressing      located
edible        lower
fresh         major
eat           mainland
fun           region
canada        british
capital       180
indica        kingdom
produced      london
```

This makes sense!!

- LSA has learned two useful “concepts” or latent features from 62 word count features: **cities** and **fruit**.
- A nice thing is that this is completely unsupervised.
- Instead of using BOW representation, we can create LSA representation of documents and pass it to supervised models.
- Truncated SVD does not find the directions of greatest variance when the data are not pre-centered. See here.
- Similar performance with less overfitting with a lot less features (8592 features vs. 2000 features).
- But it took a lot longer to extract these components.

- Another common way to preprocess counts data is using MaxAbsScaler, which scales each feature with its maximum absolute value.

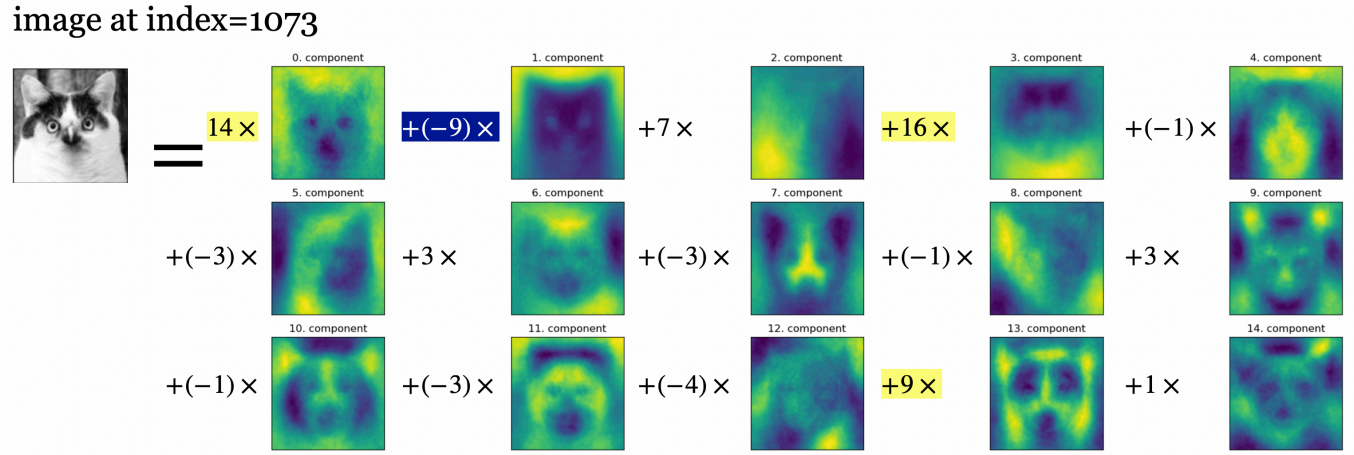
4.3 Non-negative matrix factorization (NMF)

- Published in Nature.
- Another unsupervised learning algorithm to extract useful features.
- Non-negative matrix factorization (NMF) uses a similar decomposition but with different constraints imposed on Z and W. $\$X_{(n\times d)} \approx Z_{(n\times k)}W_{(k\times d)}\$$
- Unlike PCA, the components are neither orthogonal to each other nor are they sorted by the variance explained by them.
- The constraint in NMF is that both Z and W will be allowed to contain only non-negative values.
- This means that it can only be applied to data where each feature is non-negative, as non-negative sum of non-negative components cannot become negative.

- The NMF "components" will only be adding things.
- Once something is added it cannot be removed because we cannot subtract anything.
- For example, in PCA you can have something like
  - $X_i = 14W_0 - 9W_2$, a cancellation effect from subtraction.
- In NMF, once you added $14W_0$ it cannot be cancelled out.
- This usually leads to more interpretable models, as there are no cancellation effects due to different signs.

PCA Representation vs. NMF representation

- PCA representation we saw before:

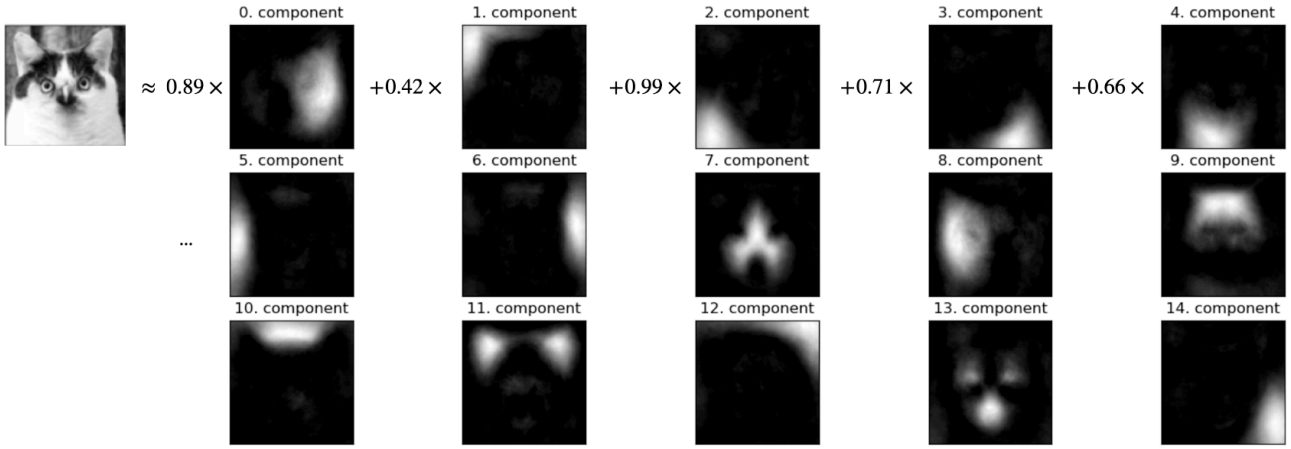


Row for index 1073 from Z

	Z_0	Z_1	Z_2	Z_3	Z_4	Z_5	Z_6	Z_7	Z_8	Z_9	Z_10	Z_11	Z_12	Z_13	Z_14
1073	14.0	-9.0	7.0	16.0	-1.0	-3.0	3.0	-3.0	-1.0	3.0	-1.0	-3.0	-4.0	9.0	1.0

In NMF,

- Everything is non-negative!
- No cancellation effects.

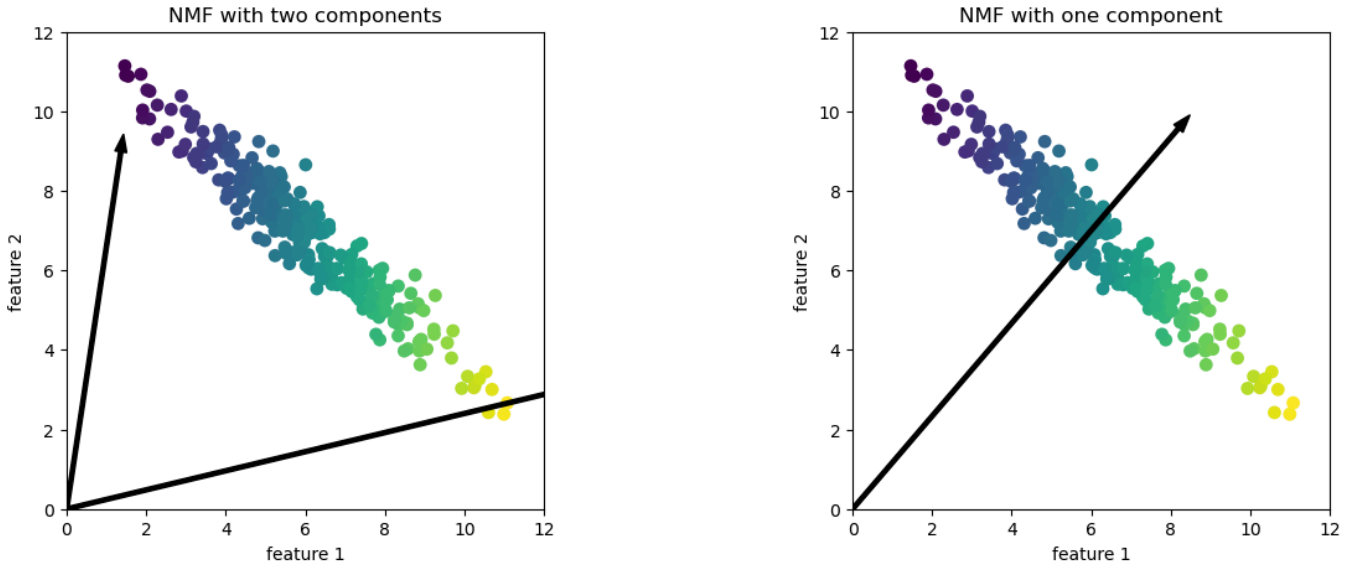


	NMF0	NMF1	NMF2	NMF3	NMF4	NMF5	NMF6	NMF7	NMF8	NMF9	NMF10	NMF11	NMF12	NMF13	NMF14
1053	0.894352	0.423642	0.991597	0.713478	0.667693	0.36344	0.034891	0.501453	0.377152	0.0	0.327164	0.845866	0.068551	0.0	0.333708

- Helpful in scenarios where data is created with several independent sources. For example:
  - audio track of multiple people speaking
  - music with many instruments
- The goal is to identify the original components that make up the combined data.
- Relates to the notion of combining parts to create a whole.

```
plot_nmf_illustration();
```

```
/Users/kvarada/miniforge3/envs/jbook/lib/python3.12/site-packages/sklearn/decor

Maximum number of iterations 200 reached. Increase it to improve convergence.
```



- In this case, if `n_components=2`, the components point towards the extremes of the data.
- If we only have 1 component, according to NMF, point toward the mean, which best explains the data.

Effect of `n_components` on PCA vs. NMF

- PCA is a **global algorithm** in the sense that the first principal component always points toward the direction of maximum variance and the second principal component toward the direction of second maximum variance and so on.
- Changing the value for the number of components does not change this.
- In contrast, in NMF, with different number of components we get entirely different set of components.
- Components in NMF are also not ordered in any specific way and all components play an equal part.

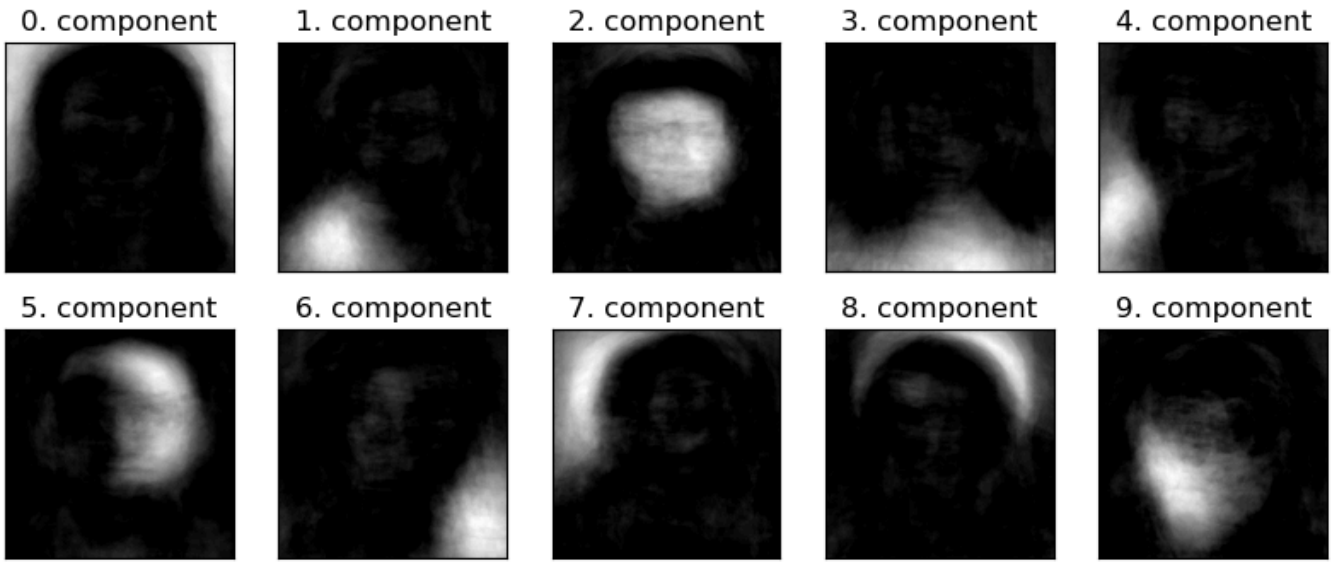
Let's go back to our people faces dataset.

```
from sklearn.decomposition import NMF

nmf = NMF(n_components=15, init="random", random_state=123, max_iter=5000)
nmf.fit(X_faces)
Z_nmf = nmf.transform(X_faces)
W_nmf = nmf.components_
```

NMF is slower than PCA.

Let's try to interpret the components given by NMF.

```
plot_components(W_nmf)
```




```
components=np.arange(0,10)
for compn in components:
    plot_strong_comp_images(X_faces, Z_nmf, W_nmf, compn=compn, image_shape=im
```

Take-away points

- For NMF everything is non-negative: the data, the components, and the transformed points.
- Terms cannot cancel each other out which makes it more interpretable.
- Unlike PCA, changing the number of components actually changes the directions of the other components.

❓❓ Questions for you

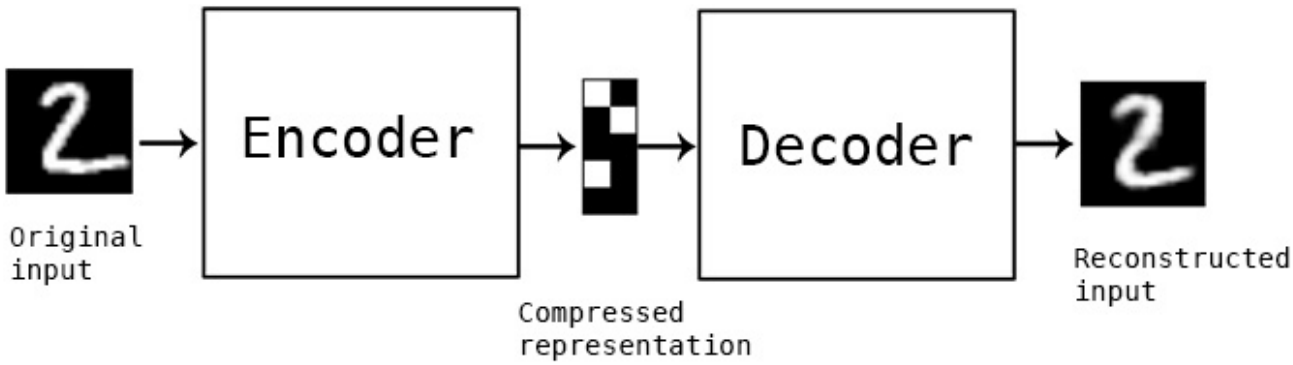
Exercise 4.2 Select all of the following statements which are **True** (iClicker)

- (A) Unlike bag-of-words representation, `TruncatedSVD` (LSA) gives a short and dense representation of documents.
- (B) In LSA, the weight vectors in W can be thought of as topics extracted from a set of documents.
- (C) In LSA, the coefficients from the transformed data tell us the proportion of each topic in a given document.
- (D) Suppose you are working with a dataset with 10 dimensions. You train `PCA` with `n_components=4` first. Then you train it with `n_components=1`. The absolute values in the first component vector are going to be exactly the same in both cases.
- (E) Suppose you are working with a dataset with 10 dimensions. You train `NMF` with `n_components=4` first. Now if you train `NMF` with `n_components=1`. The absolute values in the first component vector are going to be exactly the same in both cases.

V's Solutions!

5. (Optional) Autoencoders

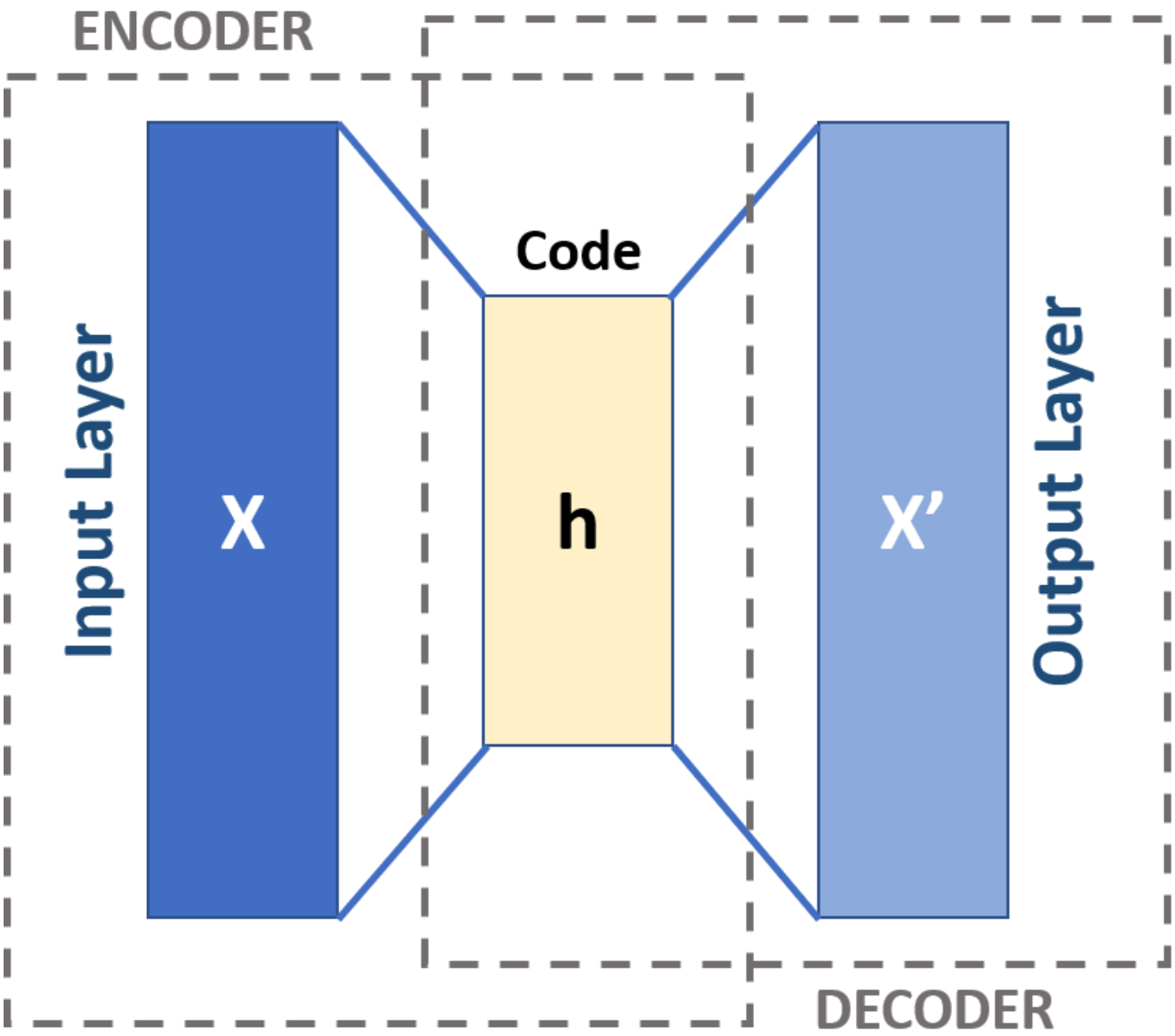
Autoencoders



Source

- Unsupervised learning technique for representation learning and dimensionality reduction.
- It's a neural network architecture.
- The idea is to create a "bottleneck" in the network so that the network is forced to learn compressed representation of the data.

Autoencoder architecture



Source

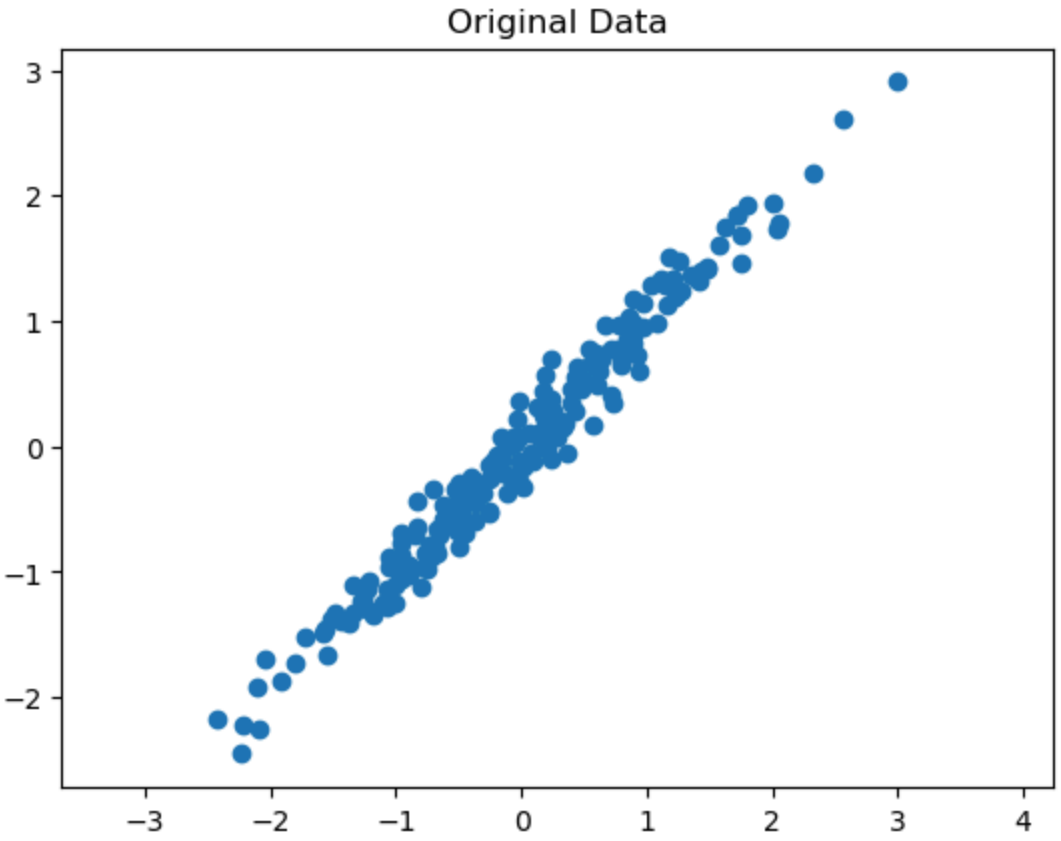
- The input layer is a vector representation of a data point.
- It has two phases: encoding and decoding.
- Encoding: The first layer transformed to a hidden layer.
- Decoding: The last layer converts the lower dimensional representation back to the original dimensionality.

What does it learn?

- An autoencoder aims to minimize the distance of this reconstructed vector and the original vector.
- And during this process, it creates good lower dimensional representations of the data.

Let's look at a simple example of an autoencoder. Here is some synthetic data.

```
# Generate synthetic data
np.random.seed(0)
X = np.dot(np.random.rand(2, 2), np.random.randn(2, 200)).T
X = StandardScaler().fit_transform(X)  # Standardize data

plt.scatter(X[:, 0], X[:, 1])
plt.title("Original Data")
plt.axis('equal')
plt.show()
```



PCA

- Let's reduce the dimensionality from 2 to 1 with PCA

```
pca = PCA(n_components=1)
X_pca = pca.fit_transform(X)

# Project back to 2D
X_projected = pca.inverse_transform(X_pca)
```

```
!export MKL_VERBOSE=0
```

Linear autoencoder

```
import torch
import torch.nn as nn
import torch.optim as optim

# Convert numpy arrays to PyTorch tensors
X_torch = torch.tensor(X, dtype=torch.float32)

# Define the autoencoder
class LinearAutoencoder(nn.Module):
    def __init__(self):
        super(LinearAutoencoder, self).__init__()
        self.encoder = nn.Linear(2, 1, bias=False)  # Encoder
        self.decoder = nn.Linear(1, 2, bias=False)  # Decoder

    def forward(self, x):
        x = self.encoder(x)
        x = self.decoder(x)
        return x

# Initialize the autoencoder
autoencoder = LinearAutoencoder()

# Loss and optimizer
criterion = nn.MSELoss()
optimizer = optim.Adam(autoencoder.parameters(), lr=0.01)

# Train the autoencoder
for epoch in range(500):
    # Forward pass
    output = autoencoder(X_torch)
    loss = criterion(output, X_torch)
    print('Loss: ', loss.item())
    # Backward and optimize
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

# Use the autoencoder to reconstruct the data
X_torch_encoded = autoencoder.encoder(X_torch).detach().numpy()
X_torch_decoded = autoencoder(X_torch).detach().numpy()
```

```
Loss:  0.9991697072982788
Loss:  0.9973030090332031
Loss:  0.9960238933563232
Loss:  0.9945106506347656
Loss:  0.9925733804702759
Loss:  0.9901602268218994
Loss:  0.9872382283210754
Loss:  0.9837810397148132
Loss:  0.979766845703125
Loss:  0.9751763939857483
Loss:  0.969992995262146
Loss:  0.9642025828361511
Loss:  0.9577930569648743
Loss:  0.9507550001144409
Loss:  0.9430817365646362
Loss:  0.9347688555717468
Loss:  0.9258149862289429
Loss:  0.9162218570709229
Loss:  0.9059945940971375
Loss:  0.8951417803764343
Loss:  0.8836755156517029
Loss:  0.8716123700141907
Loss:  0.8589723706245422
Loss:  0.8457802534103394
Loss:  0.832064688205719
Loss:  0.8178590536117554
Loss:  0.8032008409500122
Loss:  0.7881320118904114
Loss:  0.7726986408233643
Loss:  0.7569507360458374
Loss:  0.7409417629241943
Loss:  0.724728524684906
Loss:  0.7083702683448792
Loss:  0.6919280290603638
Loss:  0.6754637956619263
Loss:  0.6590394377708435
Loss:  0.6427152156829834
Loss:  0.6265489459037781
Loss:  0.6105937361717224
Loss:  0.5948971509933472
Loss:  0.5794991850852966
Loss:  0.5644309520721436
Loss:  0.5497129559516907
Loss:  0.5353550910949707
Loss:  0.5213554501533508
Loss:  0.5077015161514282
Loss:  0.4943704903125763
Loss:  0.48133206367492676
Loss:  0.46855059266090393
Loss:  0.4559885263442993
Loss:  0.4436093270778656
Loss:  0.4313809275627136
Loss:  0.41927778720855713
Loss:  0.40728282928466797
Loss:  0.3953874111175537
```

```
Loss:  0.3835912048816681
Loss:  0.3719004690647125
Loss:  0.36032646894454956
Loss:  0.3488824963569641
Loss:  0.33758243918418884
Loss:  0.3264380991458893
Loss:  0.3154583275318146
Loss:  0.304647833108902
Loss:  0.29400742053985596
Loss:  0.2835339605808258
Loss:  0.2732214331626892
Loss:  0.2630622386932373
Loss:  0.25304803252220154
Loss:  0.2431708723306656
Loss:  0.23342452943325043
Loss:  0.22380468249320984
Loss:  0.2143096774816513
Loss:  0.2049403041601181
Loss:  0.19569988548755646
Loss:  0.18659375607967377
Loss:  0.17762908339500427
Loss:  0.16881416738033295
Loss:  0.16015802323818207
Loss:  0.1516702026128769
Loss:  0.1433602273464203
Loss:  0.13523761928081512
Loss:  0.12731164693832397
Loss:  0.11959125846624374
Loss:  0.1120852530002594
Loss:  0.10480210185050964
Loss:  0.09775014221668243
Loss:  0.09093751758337021
Loss:  0.08437228947877884
Loss:  0.0780622661113739
Loss:  0.07201497256755829
Loss:  0.06623761355876923
Loss:  0.06073679029941559
Loss:  0.055518437176942825
Loss:  0.050587624311447144
Loss:  0.045948371291160583
Loss:  0.0416034460067749
Loss:  0.03755425289273262
Loss:  0.03380069509148598
Loss:  0.030341045930981636
Loss:  0.027171937748789787
Loss:  0.024288278073072433
Loss:  0.02168325148522854
Loss:  0.019348392263054848
Loss:  0.017273636534810066
Loss:  0.015447412617504597
Loss:  0.013856757432222366
Loss:  0.012487523257732391
Loss:  0.01132452767342329
Loss:  0.010351761244237423
Loss:  0.009552622213959694
Loss:  0.008910141885280609
```

```
Loss:  0.008407240733504295
Loss:  0.008026957511901855
Loss:  0.00775268254801631
Loss:  0.0075683994218707085
Loss:  0.007458873558789492
Loss:  0.007409841287881136
Loss:  0.007408158853650093
Loss:  0.007441923022270203
Loss:  0.007500553037971258
Loss:  0.007574834860861301
Loss:  0.007656939793378115
Loss:  0.007740390487015247
Loss:  0.00782002042979002
Loss:  0.00789188127964735
Loss:  0.007953149266541004
Loss:  0.008002003654837608
Loss:  0.008037503808736801
Loss:  0.008059443905949593
Loss:  0.008068223483860493
Loss:  0.00806473009288311
Loss:  0.008050189353525639
Loss:  0.00802607648074627
Loss:  0.007994000799953938
Loss:  0.007955627515912056
Loss:  0.00791260227560997
Loss:  0.007866490632295609
Loss:  0.007818739861249924
Loss:  0.007770640775561333
Loss:  0.007723310962319374
Loss:  0.0076776850037276745
Loss:  0.007634514942765236
Loss:  0.007594372611492872
Loss:  0.007557656150311232
Loss:  0.007524615619331598
Loss:  0.007495358120650053
Loss:  0.007469875738024712
Loss:  0.007448062300682068
Loss:  0.0074297296814620495
Loss:  0.007414632011204958
Loss:  0.007402479648590088
Loss:  0.007392954081296921
Loss:  0.007385726552456617
Loss:  0.007380460388958454
Loss:  0.007376834284514189
Loss:  0.007374538108706474
Loss:  0.007373287808150053
Loss:  0.007372823543846607
Loss:  0.007372914347797632
Loss:  0.0073733641766011715
Loss:  0.007374000735580921
Loss:  0.007374688517302275
Loss:  0.00737531716004014
Loss:  0.007375804707407951
Loss:  0.007376091554760933
Loss:  0.0073761423118412495
Loss:  0.007375934161245823
```

```
Loss:  0.0073754675686359406
Loss:  0.007374743930995464
Loss:  0.007373786065727472
Loss:  0.007372615858912468
Loss:  0.00737126637250185
Loss:  0.007369766011834145
Loss:  0.007368151564151049
Loss:  0.007366452366113663
Loss:  0.0073647028766572475
Loss:  0.007362927310168743
Loss:  0.007361154537647963
Loss:  0.007359405979514122
Loss:  0.007357697933912277
Loss:  0.00735646046233326197
Loss:  0.0073544615879654884
Loss:  0.0073529495857656
Loss:  0.0073515139520168304
Loss:  0.007350157015025616
Loss:  0.007348875980824232
Loss:  0.007347668521106243
Loss:  0.0073465281166136265
Loss:  0.007345451042056084
Loss:  0.007344427052885294
Loss:  0.007343452423810959
Loss:  0.007342517375946045
Loss:  0.00734161539003253
Loss:  0.007340739481151104
Loss:  0.00733988406136632
Loss:  0.007339043077081442
Loss:  0.007338212337344885
Loss:  0.007337384391576052
Loss:  0.007336560636758804
Loss:  0.00733573455363512
Loss:  0.007334905676543713
Loss:  0.007334072142839432
Loss:  0.0073332334868609905
Loss:  0.007332389242947102
Loss:  0.007331539411097765
Loss:  0.007330684456974268
Loss:  0.00732982624322176
Loss:  0.007328961975872517
Loss:  0.007328095845878124
Loss:  0.007327227387577295
Loss:  0.007326358929276466
Loss:  0.007325490936636925
Loss:  0.007324622943997383
Loss:  0.007323755417019129
Loss:  0.007322890684008598
Loss:  0.007322028744965792
Loss:  0.0073211705312132835
Loss:  0.007320315111428499
Loss:  0.007319463416934013
Loss:  0.007318613585084677
Loss:  0.0073177688755095005
Loss:  0.007316927891224623
Loss:  0.007316090632230043
```

```
Loss:  0.007315254770219326
Loss:  0.007314424030482769
Loss:  0.0073135956190526485
Loss:  0.00731276907026767 7
Loss:  0.007311945781111717
Loss:  0.007311124354600906
Loss:  0.007310305256396532
Loss:  0.007309488020837307
Loss:  0.007308672182261944
Loss:  0.007307858671993017
Loss:  0.007307046093046665
Loss:  0.007306235376745462
Loss:  0.007305426988750696
Loss:  0.007304619997739792
Loss:  0.0073038130067288876
Loss:  0.007303008809685707
Loss:  0.007302205543965101
Loss:  0.007301404606550932
Loss:  0.0073006050661206245
Loss:  0.007299806922674179
Loss:  0.007299010641872883
Loss:  0.007298216689378023
Loss:  0.007297424599528313
Loss:  0.0072966329753398895
Loss:  0.00729584414511919
Loss:  0.00729505717754364
Loss:  0.007294272072613239
Loss:  0.007293488830327988
Loss:  0.00729270838201046
Loss:  0.0072919283993542194
Loss:  0.007291150745004415 5
Loss:  0.0072903758846223354
Loss:  0.007289601489901543
Loss:  0.00728883035480976 1
Loss:  0.007288060616701841
Loss:  0.007287293206900358
Loss:  0.007286528125405312
Loss:  0.007285764906555414
Loss:  0.007285003550350666
Loss:  0.007284244522452354
Loss:  0.007283487357199192
Loss:  0.007282731123268604
Loss:  0.007281978148967028
Loss:  0.007281226571649313
Loss:  0.0072804768569767475
Loss:  0.007279729936271906
Loss:  0.007278984412550926
Loss:  0.007278240751475096
Loss:  0.007277499884366989
Loss:  0.0072767604142427444
Loss:  0.007276023738086224
Loss:  0.007275287993252277
Loss:  0.00727455597370863
Loss:  0.0072738248854875565
Loss:  0.0072730956599116325
Loss:  0.0072723692283034325
```

```
Loss:  0.007271645590662956
Loss:  0.00727092195302248
Loss:  0.007270202040672302
Loss:  0.007269483990967274
Loss:  0.007268767338246107
Loss:  0.00726853945153952
Loss:  0.007267342414706945
Loss:  0.007266632746905088
Loss:  0.007265924941748381
Loss:  0.007265218999236822
Loss:  0.007264516316354275
Loss:  0.007263815030455589
Loss:  0.00726311607286334
Loss:  0.007262418977916241
Loss:  0.007261725142598152
Loss:  0.007261031772941351
Loss:  0.007260343059897423
Loss:  0.007259653881192207
Loss:  0.007258969359099865
Loss:  0.00725828530266881
Loss:  0.007257602643221617
Loss:  0.007256923709064722
Loss:  0.007256247568875551
Loss:  0.007255572825670242
Loss:  0.007254899945110083
Loss:  0.007254229858517647
Loss:  0.00725356163457036
Loss:  0.00725289573892951
Loss:  0.007252231705933809
Loss:  0.007251570001244545
Loss:  0.0072509110905230045
Loss:  0.00725025357678532 6
Loss:  0.00724959932267665 9
Loss:  0.007248946465551853
Loss:  0.007248296868056059
Loss:  0.007247648201882839
Loss:  0.007247002795338631
Loss:  0.007246358320116997
Loss:  0.007245717104524374
Loss:  0.007245076820254326
Loss:  0.007244440261274576
Loss:  0.007243806030601263
Loss:  0.007243173662573099
Loss:  0.0072425431571900845
Loss:  0.007241914980113506
Loss:  0.007241290062665939
Loss:  0.007240666542202234
Loss:  0.007240045815706253
Loss:  0.007239426486194134
Loss:  0.007238809950649738
Loss:  0.007238195743411779
Loss:  0.00723758339881897
Loss:  0.007236973848193884
Loss:  0.007236365228891373
Loss:  0.007235760800540447
Loss:  0.007235157303512096
```

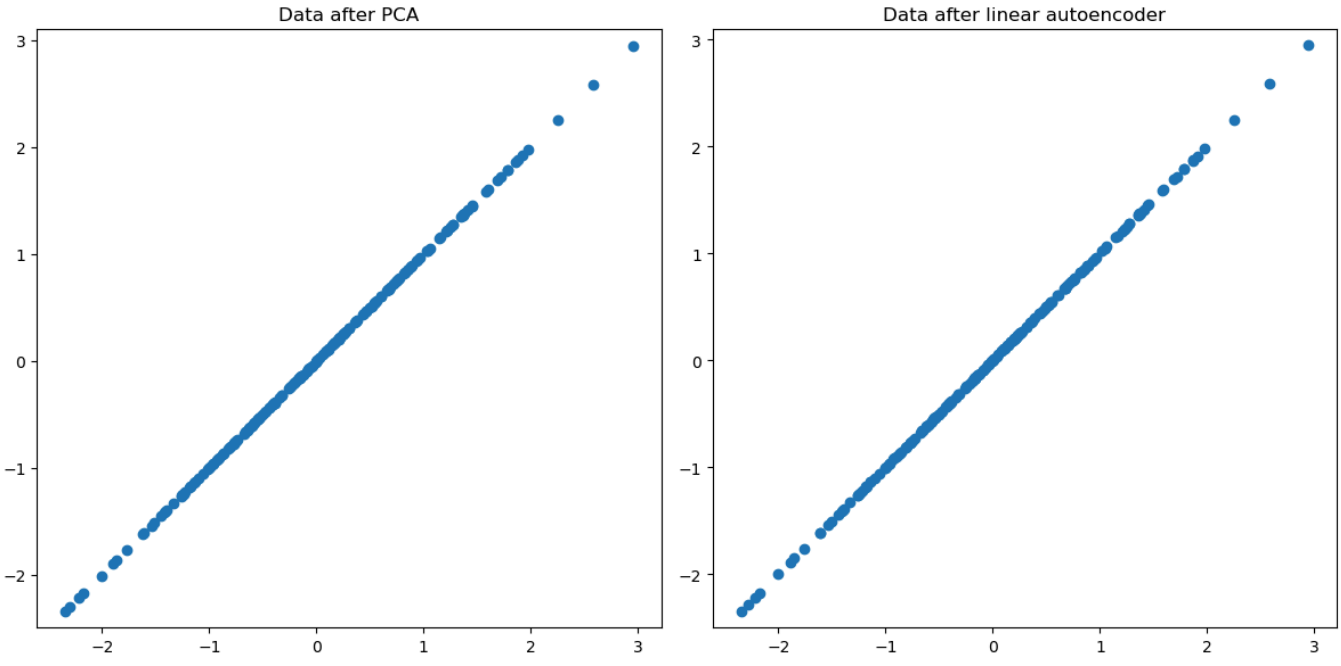
```
Loss:  0.007234557066112757
Loss:  0.00723357760035992
Loss:  0.00723336782265663
Loss:  0.0072327665984630585
Loss:  0.00723217474296689
Loss:  0.00723185215777159
Loss:  0.00723098482555151
Loss:  0.00723041314631700 5
Loss:  0.00722983060404658 3
Loss:  0.00722924945876002 3
Loss:  0.00722867110744118 7
Loss:  0.00722809555009007 45
Loss:  0.00722752138972282 4
Loss:  0.00722695002332329 75
Loss:  0.00722638005390763 3
Loss:  0.00722581334412097 9
Loss:  0.0072252475656569
Loss:  0.00722468551248312
Loss:  0.00722412485629320 14
Loss:  0.00722356699407100 7
Loss:  0.00722301146015524 9
Loss:  0.00722245685756206 5
Loss:  0.00722190551459789 3
Loss:  0.00722135649994015 7
Loss:  0.00722081027925014 5
Loss:  0.00722026544554399 5
Loss:  0.00721972249448299 4
Loss:  0.00721918232738971 7
Loss:  0.00721864402294158 9
Loss:  0.00721810804679989 8
Loss:  0.00721757357393330 356
Loss:  0.00721704261377453 8
Loss:  0.00721651362252155 64
Loss:  0.00721598695963621 1
Loss:  0.00721546169370412 8
Loss:  0.00721493922173976 9
Loss:  0.00721441814614672 72
Loss:  0.00721390079706907 3
Loss:  0.00721338391304016 1
Loss:  0.00721287075430154 8
Loss:  0.00721235852688550 95
Loss:  0.00721184955909848 2
Loss:  0.00721134245395660 4
Loss:  0.0072108383814278245
Loss:  0.0072103347629308 7
Loss:  0.00720983371138572 7
Loss:  0.00720933534538083 08
Loss:  0.00720883905887603 8
Loss:  0.00720834545791149 1
```

```
Loss:  0.007207853253930807
Loss:  0.00720736244693398 5
Loss:  0.0072068748995661736
Loss:  0.007206389214843512
Loss:  0.007205907255411148
Loss:  0.0072054252959787846
Loss:  0.007204946130514145
Loss:  0.0072044688276946545
Loss:  0.007203994784504175
Loss:  0.0072035216726362705
Loss:  0.00720305135473609
Loss:  0.007202582899481058
Loss:  0.0072021158412098885
Loss:  0.007201652508229017
Loss:  0.007201189175248146
Loss:  0.007200730498880148
Loss:  0.007200272288173437
Loss:  0.0071998173370957375
Loss:  0.007199362386018038
Loss:  0.007198910694569349
Loss:  0.007198461797088385
Loss:  0.007198014296591282
Loss:  0.007197569124400616
Loss:  0.007197126280516386
Loss:  0.007196684367954731
Loss:  0.007196245715022087
Loss:  0.0071958089247345924
Loss:  0.007195373065769672
Loss:  0.007194940932095051
Loss:  0.007194510195404291
Loss:  0.007194080390036106
Loss:  0.0071936543099582195
Loss:  0.0071932291612029076
Loss:  0.007192805875092745
Loss:  0.0071923863142728806
Loss:  0.007191966753453016
Loss:  0.0071915509179234505
Loss:  0.007191135082393885
Loss:  0.007190722972154617
Loss:  0.007190312258899212
Loss:  0.007189903873950243
Loss:  0.007189497351646423
Loss:  0.00718909315764904
Loss:  0.007188689894974232
Loss:  0.00718828896060586
Loss:  0.007187891751527786
Loss:  0.007187493611127138
Loss:  0.007187099661678076
Loss:  0.007186706177890301
Loss:  0.0071863154880702496
Loss:  0.007185926660895348
Loss:  0.0071855392307043076
Loss:  0.007185154594480991
Loss:  0.007184769958257675
Loss:  0.007184389978647232
```

```
Loss:  0.0071840109303593636
Loss:  0.007183633279055357
Loss:  0.0071832574903965
Loss:  0.007182884030044079
Loss:  0.0071825128979980946
Loss:  0.00718214269727468 5
Loss:  0.0071817748248577 12
Loss:  0.0071814088150858 88
Loss:  0.0071810455992817 88
Loss:  0.0071806819178164005
Loss:  0.007180322427302599
Loss:  0.00717996340245008 5
Loss:  0.0071796075171565294
Loss:  0.0071792523376643 66
Loss:  0.0071788998320698 74
Loss:  0.0071785487234592 44
Loss:  0.0071781994774937 63
Loss:  0.0071778511628508 57
Loss:  0.0071775056421756744
Loss:  0.007177161518484354
Loss:  0.0071768201887607574
Loss:  0.0071764797903597355
Loss:  0.00717614172026515
Loss:  0.0071758050471544 27
Loss:  0.0071754693053662 78
Loss:  0.0071751363575458 53
Loss:  0.0071748052723705 77
Loss:  0.0071744765155017 38
Loss:  0.0071741477586328 98
Loss:  0.0071738227270543575
Loss:  0.0071734981611371 04
Loss:  0.0071731759235262 87
Loss:  0.0071728541515767 57
Loss:  0.0071725356392562 39
Loss:  0.0071722185239195 82
Loss:  0.0071719032712280 75
Loss:  0.0071715889498591 42
Loss:  0.0071712774224579 33
Loss:  0.0071709663607180 12
Loss:  0.0071706580929458 14
Loss:  0.0071703507564961 91
Loss:  0.0071700457483530045
Loss:  0.007169742602854967
Loss:  0.0071694403886795 04
Loss:  0.0071691395714879 04
Loss:  0.0071688415482640 27
Loss:  0.0071685444563627 24
Loss:  0.0071682487614452 84
Loss:  0.0071679549291729 93
Loss:  0.0071676629595458 51
Loss:  0.0071673719212412 83
Loss:  0.0071670841425652 727
Loss:  0.0071667972952127 46
Loss:  0.0071665109135210 51
Loss:  0.0071662273257970 81
Loss:  0.0071659446693956 85
```

```
Loss:  0.007165663875639439
Loss:  0.007165384944528341
Loss:  0.007165107876062393
Loss:  0.007164831273257732
Loss:  0.007164556998759508
```

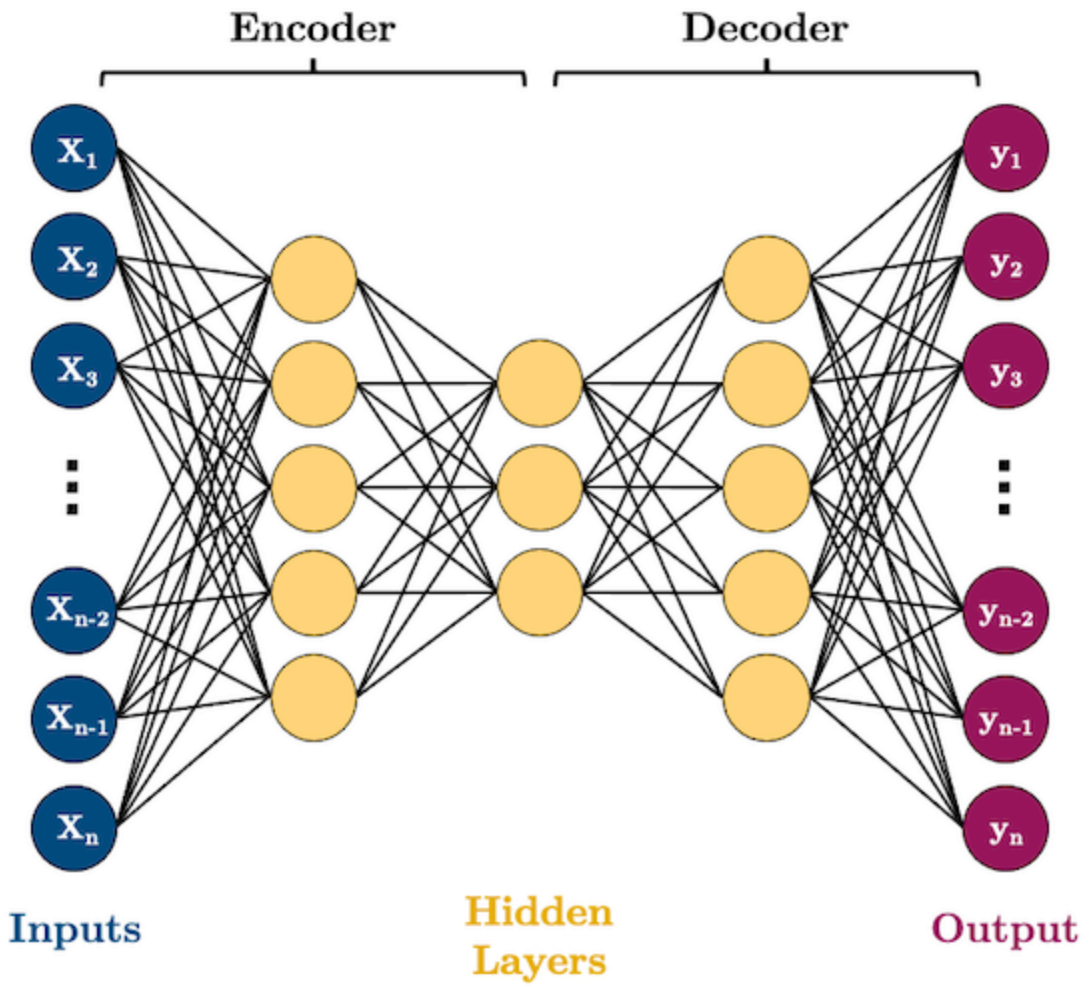
```
fig, ax = plt.subplots(1, 2, figsize=(12, 6))  # 1 row, 2 columns

# Plot 1: Data after PCA
ax[0].scatter(X_projected[:, 0], X_projected[:, 1])
ax[0].set_title("Data after PCA")
ax[0].axis('equal')

# Plot 2: Data after Autoencoder (PyTorch)
ax[1].scatter(X_torch_decoded[:, 0], X_torch_decoded[:, 1])
ax[1].set_title("Data after linear autoencoder")
ax[1].axis('equal')

plt.tight_layout()
plt.show()
```



- When we use a single hidden layer without any non-linear activation functions, it essentially performs the same function as PCA.
- However, neural networks allow for greater creativity and flexibility. We can incorporate multiple layers, and not all of these layers need to be linear. For example, we can add convolutional layers among others. You'll explore this in one of the challenging questions in the lab.

- Recall this picture of an autoencoder from 572.

- Here we have more than one hidden layers.



Non-linearity in autoencoders

- PCA is a linear dimensionality reduction technique.
- Autoencoders are non-linear because of non-linear activation functions.
- In fact, if we construct the network without non-linear activation function at each layer, we would observe similar dimensionality reduction as that of PCA.

5. Final comments, summary, reflection

Take-home message

- PCA can be thought of as generalization of K-Means.
- In general, linear dimensionality reduction techniques represent each example as a linear combinations of components in W.
- There are no definitive methods to decide the number of components. Two things which can help us out are
  - looking at the explained variance in case of PCA
  - looking at reconstructions
- There are many variants and extensions of PCA.
- `TruncatedSVD` or LSA is appropriate in the context of sparse text data, as it doesn't center the data.
- In NMF everything is non-negative.
- It is appropriate when you have non-negative data and interpretation is important for you.
- Also, it's useful when data is created as the addition of several independent sources (e.g., music with multiple sources) because NMF can identify the original components that make up the combined data.

Resources

- Matrix decompositions & latent semantic indexing
- NMF paper