Open in Colab

Ungraded lab: Algorithmic Dimensionality Reduction

Welcome, during this ungraded lab you are going to perform several algorithms that aim to reduce the dimensionality of data. This topic is very important because it is not uncommon that reduced models outperform the ones trained on the raw data because noise and redundant information are present in most datasets. This will also allow your models to train and make predictions faster, which might be really important depending on the problem you are working on. In particular you will:

1. Use Principal Component Analysis (**PCA**) to reduce the dimensionality of a dataset that classifies celestial bodies.
2. Use Single Value Decomposition (**SVD**) to create low level representations of images of handwritten digits.
3. Use Non-negative Matrix Factorization (**NMF**) to segment text into topics.

Let's get started!

In [1]:
```
# General use imports
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
```

Principal Components Analysis - PCA

This is an unsupervised algorithm that creates linear combinations of the original features. PCA is a widely used technique for dimension reduction since it is fast and easy to implement. PCA aims to keep as much variance as possible from the original data in a lower dimensional space. It finds the best axis to project the data so that the variance of the projections is maximized.

In the lecture you saw PCA applied to the Iris dataset. This dataset has been used extensively to showcase PCA so here you are going to do something different. You are going to use the HTRU_2 dataset which describes several celestial objects and the idea is to be able to classify if an object is a pulsar star or not.

Begin by downloading the dataset:

In [2]:
```
# Download zip file
!wget https://archive.ics.uci.edu/ml/machine-learning-databases/00372/HTRU2

# Unzip it
!unzip HTRU2.zip
```

```
--2021-08-16 11:02:14--  https://archive.ics.uci.edu/ml/machine-learning-da
tabases/00372/HTRU2.zip
Resolving archive.ics.uci.edu (archive.ics.uci.edu)... 128.195.10.252
Connecting to archive.ics.uci.edu (archive.ics.uci.edu)|128.195.10.252|:44
3... connected.
HTTP request sent, awaiting response... 200 OK
Length: 1563015 (1.5M) [application/x-httpd-php]
Saving to: 'HTRU2.zip'

HTRU2.zip           100%[===================>]   1.49M  7.81MB/s    in 0.2s

2021-08-16 11:02:15 (7.81 MB/s) - 'HTRU2.zip' saved [1563015/1563015]

Archive:  HTRU2.zip
  inflating: HTRU_2.csv
  inflating: HTRU_2.arff
  inflating: Readme.txt
```

Load the data into a dataframe for easier inspection:

In [3]:
```
# Load data into a pandas dataframe
data = pd.read_csv("HTRU_2.csv", names=['mean_ip', 'sd_ip', 'ec_ip',
                                        'sw_ip', 'mean_dm', 'sd_dm',
                                        'ec_dm', 'sw_dm', 'pulsar'])

# Take a look at the data
data.head()
```

Out[3]:

	mean_ip	sd_ip	ec_ip	sw_ip	mean_dm	sd_dm	ec_dm	sw_dm	pulsa
0	140.562500	55.683782	-0.234571	-0.699648	3.199833	19.110426	7.975532	74.242225	
1	102.507812	58.882430	0.465318	-0.515088	1.677258	14.860146	10.576487	127.393580	
2	103.015625	39.341649	0.323328	1.051164	3.121237	21.744669	7.735822	63.171909	
3	136.750000	57.178449	-0.068415	-0.636238	3.642977	20.959280	6.896499	53.593661	
4	88.726562	40.672225	0.600866	1.123492	1.178930	11.468720	14.269573	252.567306	

This dataset has 8 numerical features (the "pulsar" column is the label). Now you are going to perform PCA reduce this 8th-dimensional input space to a lower dimensional one.

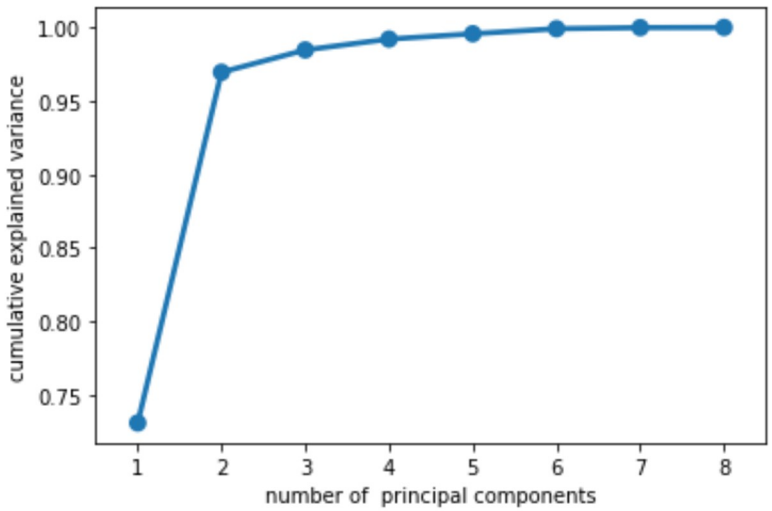
But first, scale the data. If you do an exploratory analysis of the data you will see that this dataset has a lot of outliers. Because of this you are going to use a RobustScaler, which scales features using statistics that are robust to outliers.

In [4]:
```
from sklearn.preprocessing import RobustScaler

# Split features from labels
features = data[[col for col in data.columns if col != "pulsar"]]
labels = data["pulsar"]

# Scale data
robust_data = RobustScaler().fit_transform(features)
```

Now perform PCA using sklearn. In this first iteration you are going to create a principal component for each one of the features so there is no dimensionality reduction:

In [5]:

```
from sklearn.decomposition import PCA

# Instantiate PCA without specifying number of components
pca_all = PCA()

# Fit to scaled data
pca_all.fit(robust_data)

# Save cumulative explained variance
cum_var = (np.cumsum(pca_all.explained_variance_ratio_))
n_comp = [i for i in range(1, pca_all.n_components_ + 1)]

# Plot cumulative variance
ax = sns.pointplot(x=n_comp, y=cum_var)
ax.set(xlabel='number of  principal components', ylabel='cumulative explair
plt.show()
```

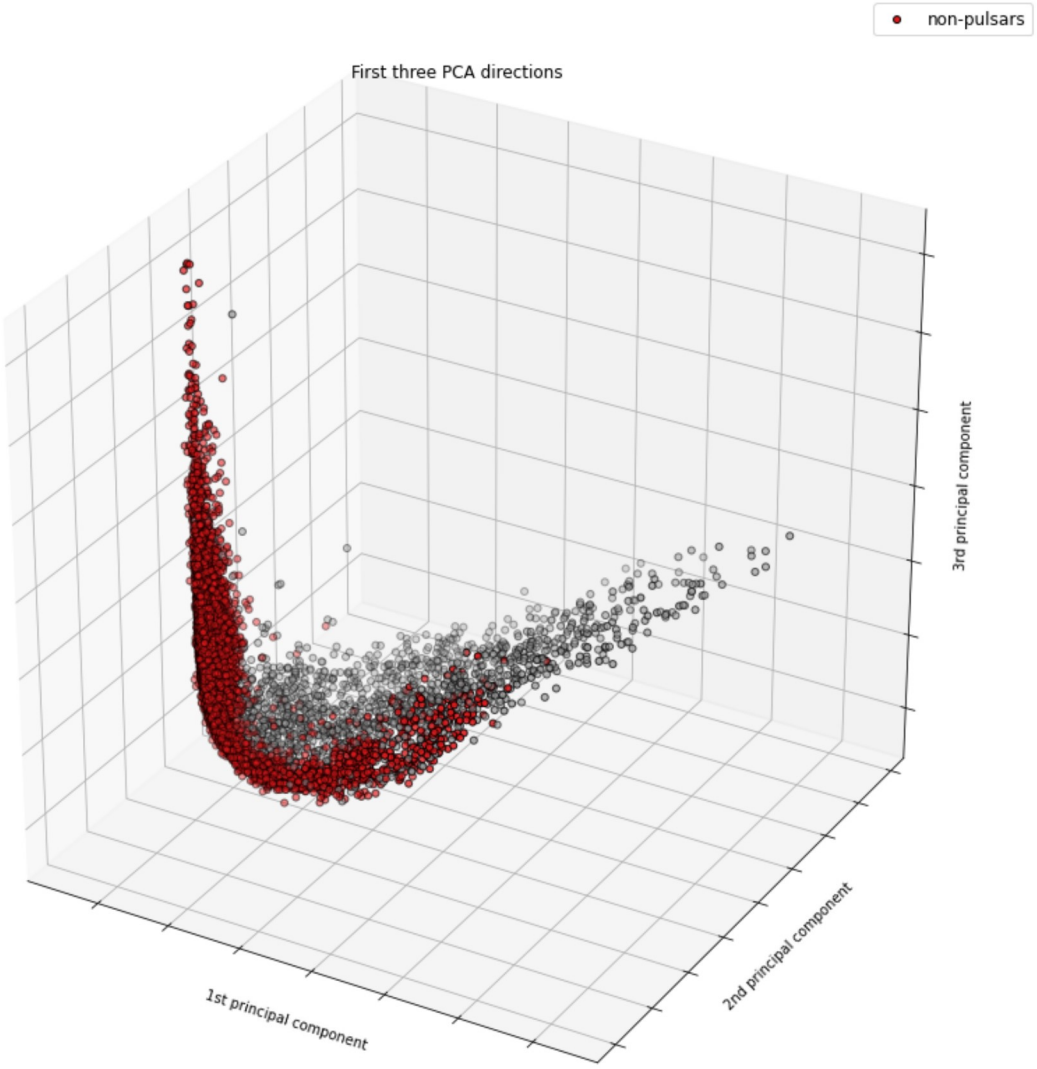


Wow! With just 3 components almost all of the variance of the original data is explained! This makes you think that there were some highly correlated features in the original data.

Let's plot the first 3 principal components:

In [6]:

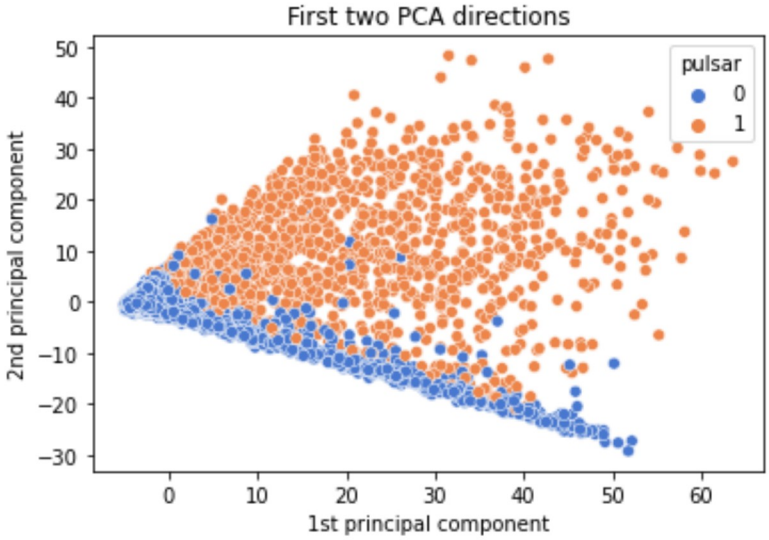
```
from mpl_toolkits.mplot3d import Axes3D

# Instantiate PCA with 3 components
pca_3 = PCA(3)

# Fit to scaled data
pca_3.fit(robust_data)

# Transform scaled data
data_3pc = pca_3.transform(robust_data)

# Render the 3D plot
fig = plt.figure(figsize=(15,15))
ax = fig.add_subplot(111, projection='3d')


ax.scatter(data_3pc[:, 0], data_3pc[:, 1], data_3pc[:, 2], c=labels,
           cmap=plt.cm.Set1, edgecolor='k', s=25, label=data['pulsar'])

ax.legend(["non-pulsars"], fontsize="large")

ax.set_title("First three PCA directions")
ax.set_xlabel("1st principal component")
ax.w_xaxis.set_ticklabels([])
ax.set_ylabel("2nd principal component")
ax.w_yaxis.set_ticklabels([])
ax.set_zlabel("3rd principal component")
ax.w_zaxis.set_ticklabels([])

plt.show()
```



It is possible to visualize a plane that would be able to separate both classes since non-pulsars tend to group on the edge of this surface while pulsars are mostly located on the inner side of the surface.

In this case it is reasonable to think that the dimension can be reduced even more since with 2 principal components more than 95% of the variance of the original data is explained. Now let's plot just the first two principal components:

In [7]:
```
# Instantiate PCA with 2 components
pca_2 = PCA(2)

# Fit and transform scaled data
pca_2.fit(robust_data)
data_2pc = pca_2.transform(robust_data)

# Render the 2D plot
ax = sns.scatterplot(x=data_2pc[:,0],
                     y=data_2pc[:,1],
                     hue=labels,
                     palette=sns.color_palette("muted", n_colors=2))

ax.set(xlabel='1st principal component', ylabel='2nd principal component',
plt.show()
```



Even in 2D the 2 classes look linearly separable (not perfectly, of course) but this is quite remarkable considering that the initial space was 8th dimensional.

Using PCA you've successfully reduced the dimensionality from 8 to 2 while maintaining a lot of the variance of the original data!

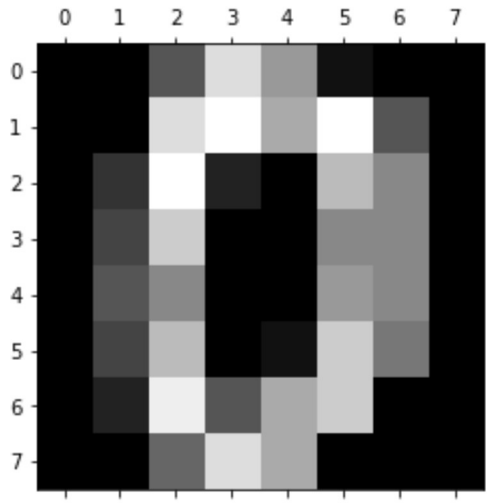
Singular Value Decomposition - SVD

SVD is one way to decompose matrices. Remember that matrices can be seen as linear transformations in space. PCA relies on eigendecomposition, which can only be done for square matrices. However you don't always have square matrices, and sometimes you have really sparse matrices.

To decompose these types of matrices, which can't be decomposed with eigendecomposition, you can use techniques such as Singular Value Decomposition. SVD decomposes the original dataset into its constituents, resulting in a reduction of dimensionality. It is used to remove redundant features from the dataset.

To check SVD you are going to use the digits dataset, which is made up of 1797 8x8 images of handwritten digits:

In [8]:
```
from sklearn.datasets import load_digits

# Load the digits dataset
digits = load_digits()

# Plot first digit
image = digits.data[0].reshape((8, 8))
plt.matshow(image, cmap = 'gray')
plt.show()
```



You might think that since every digit is 8x8 this will be a square matrix and thus PCA might be a better choice. However, each digit is represented as a 1x64 array. Also you might wonder why the first example worked with PCA if the data had far more observations than features. The reason is that when performing PCA you end up using the matrix product X^tX which is a square matrix.

The above is a consequence of the nature of the problems. The pulsar star dataset had numerical data to represent each data point. On the other hand, this dataset represents images through pixels.

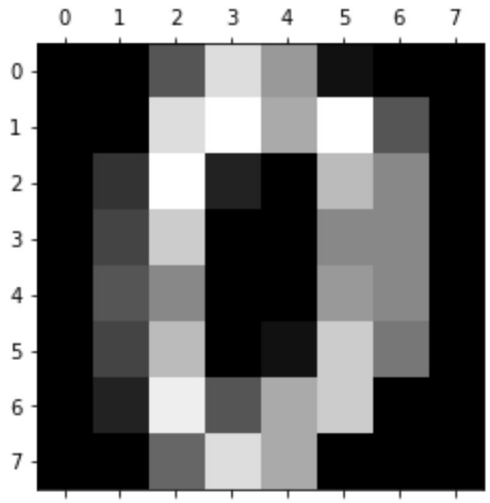
Let's continue by normalizing the data and checking its dimensions:

In [9]:
```
# Save data into X variable
X = digits.data

# Normalize pixel values
X = X/255

# Print shapes of dataset and data points
print(f"Digits data has shape {X.shape}\n")
print(f"Each data point has shape {X[0].shape}\n")
```

```
Digits data has shape (1797, 64)

Each data point has shape (64,)
```

Plot the first digit to check how normalization affects the images:

In [10]:
```
image = X[0].reshape((8, 8))
plt.matshow(image, cmap = 'gray')
plt.show()
```

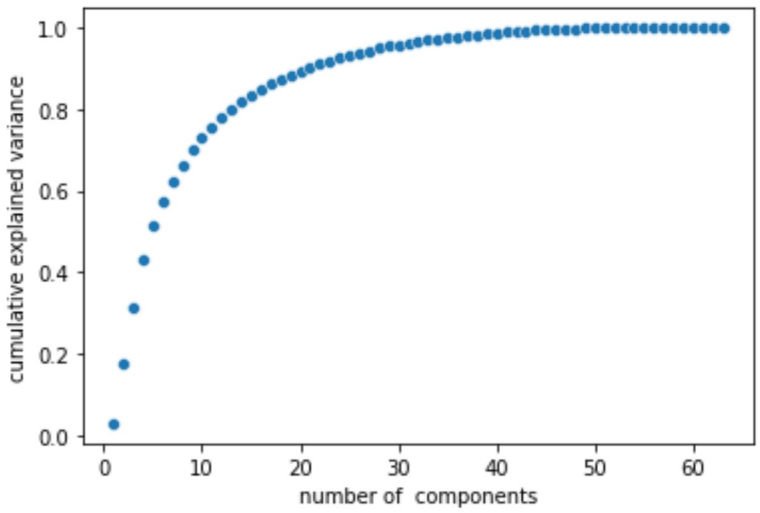


The image should be identical to the one without normalization. This is because the relative brightness of each pixel with the others is maintained. Normalization is done as a preprocessing step when feeding the data into a Neural Network. Here it is done since it is a step that is usually always done when working with image data.

Now perform SVD on the data and plot the cumulative amount of explained variance for every number of components. Note that the TruncatedSVD needs a number of components strictly lower to the number of features.

In [11]:

```
from sklearn.decomposition import TruncatedSVD

# Instantiate Truncated SVD with (original dimension - 1) components
org_dim = X.shape[1]
tsvd = TruncatedSVD(org_dim - 1)
tsvd.fit(X)

# Save cumulative explained variance
cum_var = (np.cumsum(tsvd.explained_variance_ratio_))
n_comp = [i for i in range(1, org_dim)]

# Plot cumulative variance
ax = sns.scatterplot(x=n_comp, y=cum_var)
ax.set(xlabel='number of  components', ylabel='cumulative explained varianc
plt.show()
```



Looking at the plot it can be seen that with only 5 components near the 50% of the variance

of the original data is explained.

Let's double check this:

In [12]:
```
print(f"Explained variance with 5 components: {float(cum_var[4:5])*100:.2f}
```

```
Explained variance with 5 components: 51.53%
```

It is not a lot but let's check what you get when performing SVD with only 5 components:

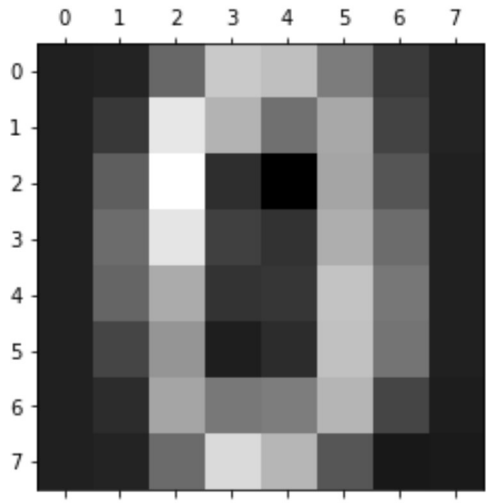
In [13]:
```
# Instantiate a Truncated SVD with 5 components
tsvd = TruncatedSVD(n_components=5)

# Get the transformed data
X_tsvd = tsvd.fit_transform(X)

# Print shapes of dataset and data points
print(f"Original data points have shape {X[0].shape}\n")
print(f"Transformed data points have shape {X_tsvd[0].shape}\n")
```

```
Original data points have shape (64,)

Transformed data points have shape (5,)
```

By doing this you are now representing each digit using 5 dimensions instead of the original 64! Isn't that amazing?

Now check how this looks like visually:

In [14]:
```
image_reduced_5 = tsvd.inverse_transform(X_tsvd[0].reshape(1, -1))
image_reduced_5 = image_reduced_5.reshape((8, 8))
plt.matshow(image_reduced_5, cmap = 'gray')
plt.show()
```



It looks blurry but you can still tell this is a zero.

Using more components

Let's try again, only, this time, we use half the number of features in the original data.

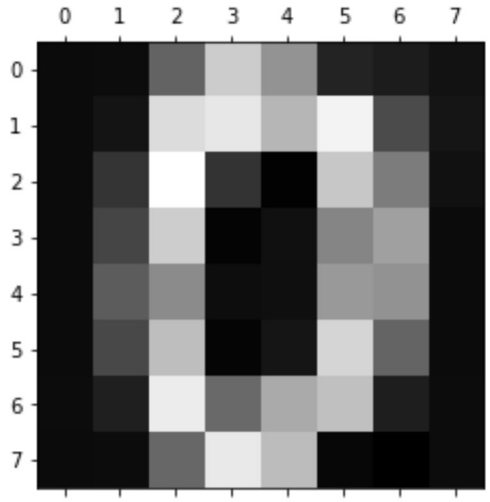
But first define a function that performs this process for any number of components:

In [15]:
```
def image_given_components(n_components, verbose=True):
  tsvd = TruncatedSVD(n_components=n_components)
  X_tsvd = tsvd.fit_transform(X)
  if verbose:
    print(f"Explained variance with {n_components} components: {float(tsvd.
  image = tsvd.inverse_transform(X_tsvd[0].reshape(1, -1))
  image = image.reshape((8, 8))
  return image
```

Use the function to generate the image that use 32 components:

In [16]:
```
image_reduced_32 = image_given_components(32)
plt.matshow(image_reduced_32, cmap = 'gray')
plt.show()
```

```
Explained variance with 32 components: 96.63%
```



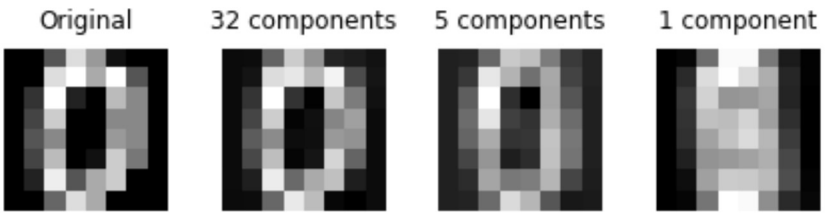
Wow! This image looks very similar to the original one (no wonder since more than 95% of the original variance is explained) but the dimensionality of the representations have been cut in half!

To better grasp how the images look like depending on the dimensionality of the representations, the next cell plots them side by side (the last figure has a parameter that you can tweak):

In [17]:

```
fig = plt.figure()

# Original image
ax1 = fig.add_subplot(1,4,1)
ax1.matshow(image, cmap = 'gray')
ax1.title.set_text('Original')
ax1.axis('off')

# Using 32 components
ax2 = fig.add_subplot(1,4,2)
ax2.matshow(image_reduced_32, cmap = 'gray')
ax2.title.set_text('32 components')
ax2.axis('off')

# Using 5 components
ax3 = fig.add_subplot(1,4,3)
ax3.matshow(image_reduced_5, cmap = 'gray')
ax3.title.set_text('5 components')
ax3.axis('off')

# Using 1 components
ax4 = fig.add_subplot(1,4,4)
ax4.matshow(image_given_components(1), cmap = 'gray') # Change this paramet
ax4.title.set_text('1 component')
ax4.axis('off')

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

```
Explained variance with 1 components: 2.87%
```



Notice how with 1 component it is not possible to determine that the image is a zero. What is the minimun number of components that are needed for this? Be sure to try out different values and see what you get!

Non-negative Matrix Factorization - NMF

NMF expresses samples as combinations of interpretable parts. For example, it represents documents as combinations of topics, and images in terms of commonly occurring visual patterns. NMF, like PCA, is a dimensionality reduction technique. In contrast to PCA, however, NMF models are interpretable. This means NMF models are easier to understand and much easier for us to explain to others. NMF can't be applied to every dataset, however. It requires the sample features be non-negative, so greater than or equal to 0.

To test NMF you will use the 20newsgroups dataset which comprises around 12000 newsgroups posts on 20 topics.

In [18]:
```
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.decomposition import NMF
from sklearn.datasets import fetch_20newsgroups

# Download data
data = fetch_20newsgroups(remove=('headers', 'footers', 'quotes'))

# Get the actual text data from the sklearn Bunch
data = data.get("data")
```

```
Downloading 20news dataset. This may take a few minutes.
Downloading dataset from https://ndownloader.figshare.com/files/5975967 (14
MB)
```

At this point you have the data in a list format. Let's check it out:

In [19]:
```
print(f"Data has {len(data)} elements.\n")
print(f"First 2 elements: \n")
for n, d in enumerate(data[:2], start=1):
  print("======"*10)
  print(f"Element number {n}:\n\n{d}\n")
```

```
Data has 11314 elements.

First 2 elements:

============================================================
Element number 1:

I was wondering if anyone out there could enlighten me on this car I saw
the other day. It was a 2-door sports car, looked to be from the late 60s/
early 70s. It was called a Bricklin. The doors were really small. In additi
on,
the front bumper was separate from the rest of the body. This is
all I know. If anyone can tellme a model name, engine specs, years
of production, where this car is made, history, or whatever info you
have on this funky looking car, please e-mail.

============================================================
Element number 2:

A fair number of brave souls who upgraded their SI clock oscillator have
shared their experiences for this poll. Please send a brief message detaili
ng
your experiences with the procedure. Top speed attained, CPU rated speed,
add on cards and adapters, heat sinks, hour of usage per day, floppy disk
functionality with 800 and 1.4 m floppies are especially requested.

I will be summarizing in the next two days, so please add to the network
knowledge base if you have done the clock upgrade and haven't answered this
poll. Thanks.
```

Notice that you only have the actual text without information of the topic it belongs to (labels).

Now you need to represent the text as vectors, for this you will use a TfidfVectorizer with `max_features` set to 500. This will be the original dimensionality of the data (which you will reduce via NMF).

In [20]:
```
# Instantiate vectorizer setting dimensionality of data
# The stop_words param refer to words (in english) that don't add much valu
vectorizer = TfidfVectorizer(max_features=500, stop_words='english')

# Vectorize original data
vect_data = vectorizer.fit_transform(data)


# Print dimensionality
print(f"Data has shape {vect_data.shape} after vectorization.")
print(f"Each data point has shape {vect_data[0].shape} after vectorization.
```

```
Data has shape (11314, 500) after vectorization.
Each data point has shape (1, 500) after vectorization.
```

Every one of the texts in the original data is represented as a 1x500 vector.

Now use NMF to reduce this dimensionality:

In [21]:
```
# Desired number of components
n_comp = 5

# Instantiate NMF with the desired number of components
nmf = NMF(n_components=n_comp, random_state=42)

# Apply NMF to the vectorized data
nmf.fit(vect_data)

reduced_vect_data = nmf.transform(vect_data)

# Print dimensionality
print(f"Data has shape {reduced_vect_data.shape} after NMF.")
print(f"Each data point has shape {reduced_vect_data[0].shape} after NMF.")

# Save feature names for plotting
feature_names = vectorizer.get_feature_names()
```

```
Data has shape (11314, 5) after NMF.
Each data point has shape (5,) after NMF.
```

Now every data point is being represented by a vector of `n_comp` dimensions rather than the original 500!

In this case every component represents a topic and each data point is represented as a combination of those topics. The value for each topic can be interpreted as how strong the relationship between the text and that particular topic is.

Check this for the 1st element of the text data:

In [22]:
```
print(f"Original text:\n{data[0]}\n")

print(f"Representation based on topics:\n{reduced_vect_data[0]}")
```

```
Original text:
I was wondering if anyone out there could enlighten me on this car I saw
the other day. It was a 2-door sports car, looked to be from the late 60s/
early 70s. It was called a Bricklin. The doors were really small. In additi
on,
the front bumper was separate from the rest of the body. This is
all I know. If anyone can tellme a model name, engine specs, years
```

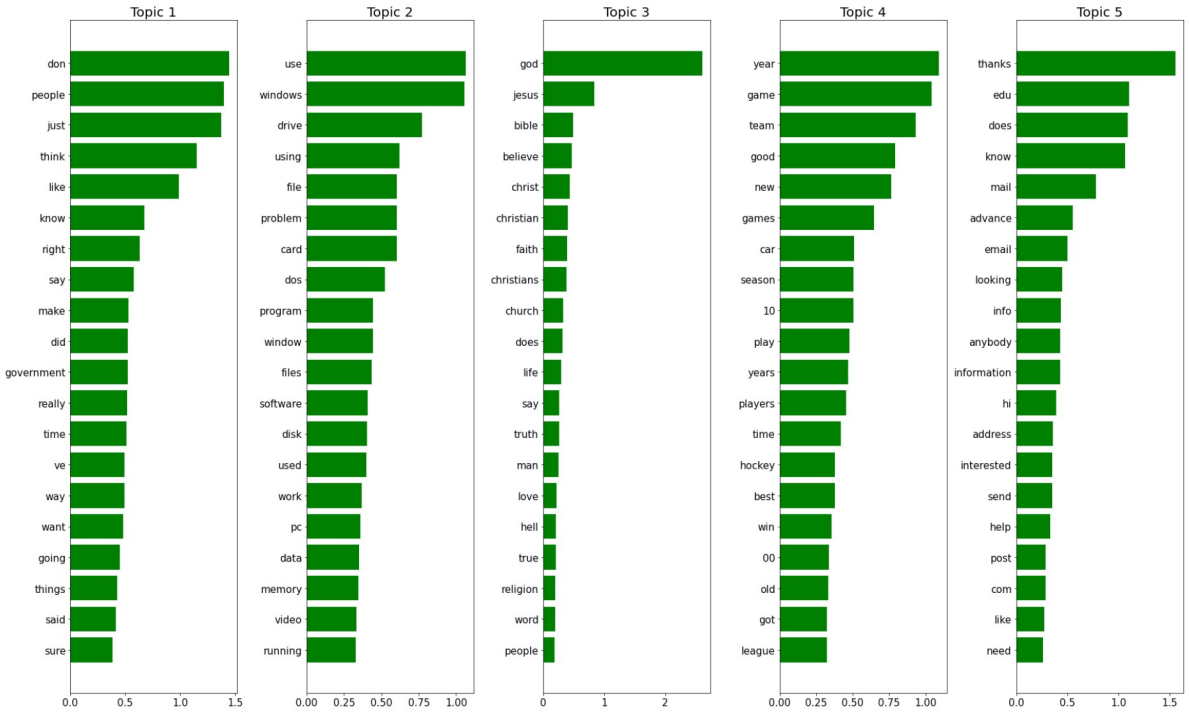
```
of production, where this car is made, history, or whatever info you
have on this funky looking car, please e-mail.

Representation based on topics:
[0.00605599 0.         0.         0.05396907 0.03956463]
```

Looks like this text can be expressed as a combination of the first, fourth and fifth topic. Specially the later two.

At this point you might wonder what these topics are. Since we didn't provide labels, these topics arised from the data. To have a sense of what these topics are, plot the top 20 words for each topic:

In [23]:

```
# Define function for plotting top 20 words for each topic
def plot_words_for_topics(n_comp, nmf, feature_names):
  fig, axes = plt.subplots(((n_comp-1)//5)+1, 5, figsize=(25, 15))
  axes = axes.flatten()

  for num_topic, topic in enumerate(nmf.components_, start=1):

    # Plot only the top 20 words

    # Get the top 20 indexes
    top_indexes = np.flip(topic.argsort()[-20:])

    # Get the corresponding feature name
    top_features = [feature_names[i] for i in top_indexes]

    # Get the importance of each word
    importance = topic[top_indexes]

    # Plot a barplot
    ax = axes[num_topic-1]
    ax.barh(top_features, importance, color="green")
    ax.set_title(f"Topic {num_topic}", {"fontsize": 20})
    ax.invert_yaxis()
    ax.tick_params(labelsize=15)

  plt.tight_layout()
  plt.show()

# Run the function
plot_words_for_topics(n_comp, nmf, feature_names)
```



Let's try to summarize each topic based on the top most common words for each one:

- The first topic is hard to describe but seems to be related to people and actions.
- The second one is clearly abouth tech stuff.
- Third one is about religion.
- Fourth one seems to revolve around sports and/or games.
- And the fifth one about education and/or information.

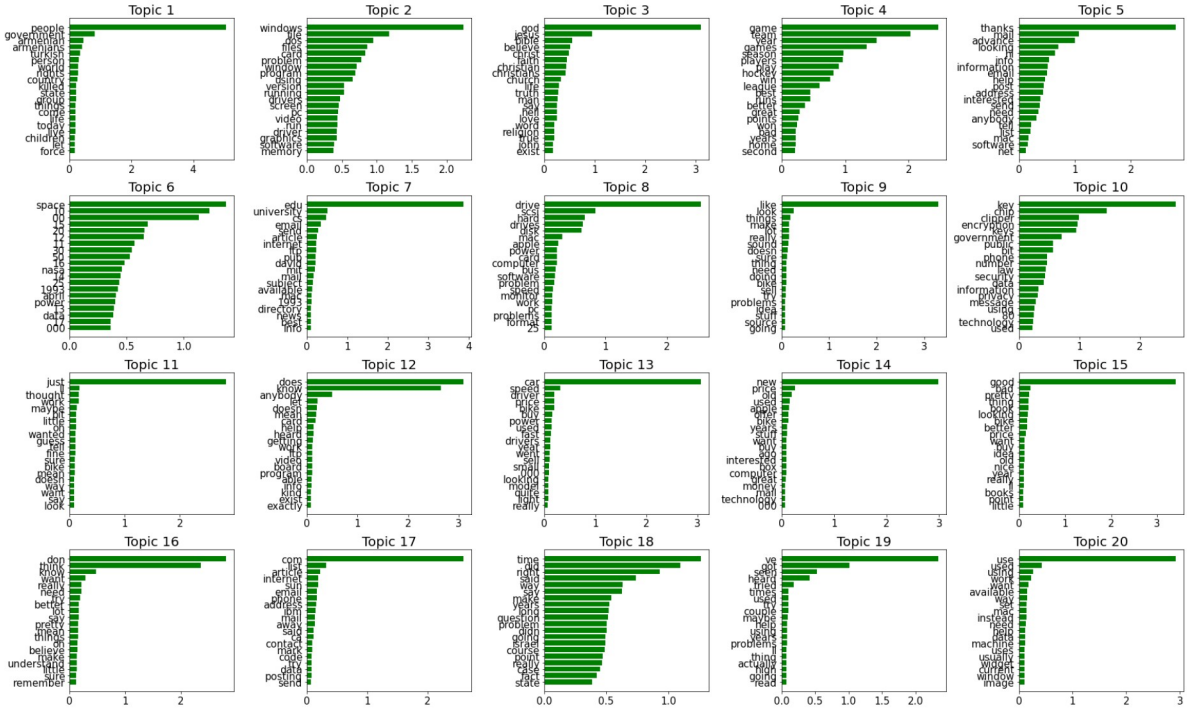
This makes sense considering the example with the first element of the text data. That text is mostly about cars (sports) and information.

Pretty cool, right?

The following function condenses the previously used code so you can play trying out with different number of components:

In [24]:
```
def try_NMF(n_comp):
  nmf = NMF(n_components=n_comp, random_state=42)
  nmf.fit(vect_data)
  feature_names = vectorizer.get_feature_names()
  plot_words_for_topics(n_comp, nmf, feature_names)
```

In [25]:
```
# Try different values!
try_NMF(20)
```



Congratulations on finishing this ungraded lab! Now you should have a clearer understanding of how to implement dimensionality reduction techniques.

The great thing about dimensionality reduction algorithms is that aside from making training and predicting faster, they perform some kind of automatic feature engineering by transforming the raw data into more meaningful representations.

Keep it up!