High dimensional data

Dimensionality reduction

- **PCA**
  - Principal Component Analysis Explained Visually (https://setosa.io/ev/principal-component-analysis/)
  - The Beginner's Guide to Dimensionality Reduction (https://dimensionality-reduction-293e465c2a3443e8941b016d.vercel.app/), by Matthew Conlen and Fred Hohman
  - A Tutorial on Principal Component Analysis (c08_pca.pdf)
- **t-SNE**
  - Visualizing Data using t-SNE (the original paper) (https://www.jmlr.org/papers/volume9/vandermaaten08a/vandermaaten08a.pdf)
  - How to Use t-SNE Effectively (https://distill.pub/2016/misread-tsne/)
  - An illustrated introduction to the t-SNE algorithm (https://www.oreilly.com/content/an-illustrated-introduction-to-the-t-sne-algorithm/)

In [2]:
```
import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sns
import numpy as np
import scipy.stats as ss

sns.set_style('white')
```

```
c:\Users\Stefan\AppData\Local\Programs\Python\Python310\lib\site-packages\matplotlib\projections\__init__.py:63: UserWarning: Unable to i
mport Axes3D. This may be due to multiple versions of Matplotlib being installed (e.g. as a system package and as a pip package). As a re
sult, the 3D projection is not available.
  warnings.warn("Unable to import Axes3D. This may be due to multiple versions of "
```

Scatterplot matrix for low-high dimensional data

In many cases, the number of dimensions is not too large. For instance, the "Iris" dataset (https://en.wikipedia.org/wiki/Iris_flower_data_set) contains four dimensions of measurements on the three types of iris flower species. It's more than two dimensions, yet still manageable.

In [3]:
```
iris = sns.load_dataset('iris')
iris.head(2)
```

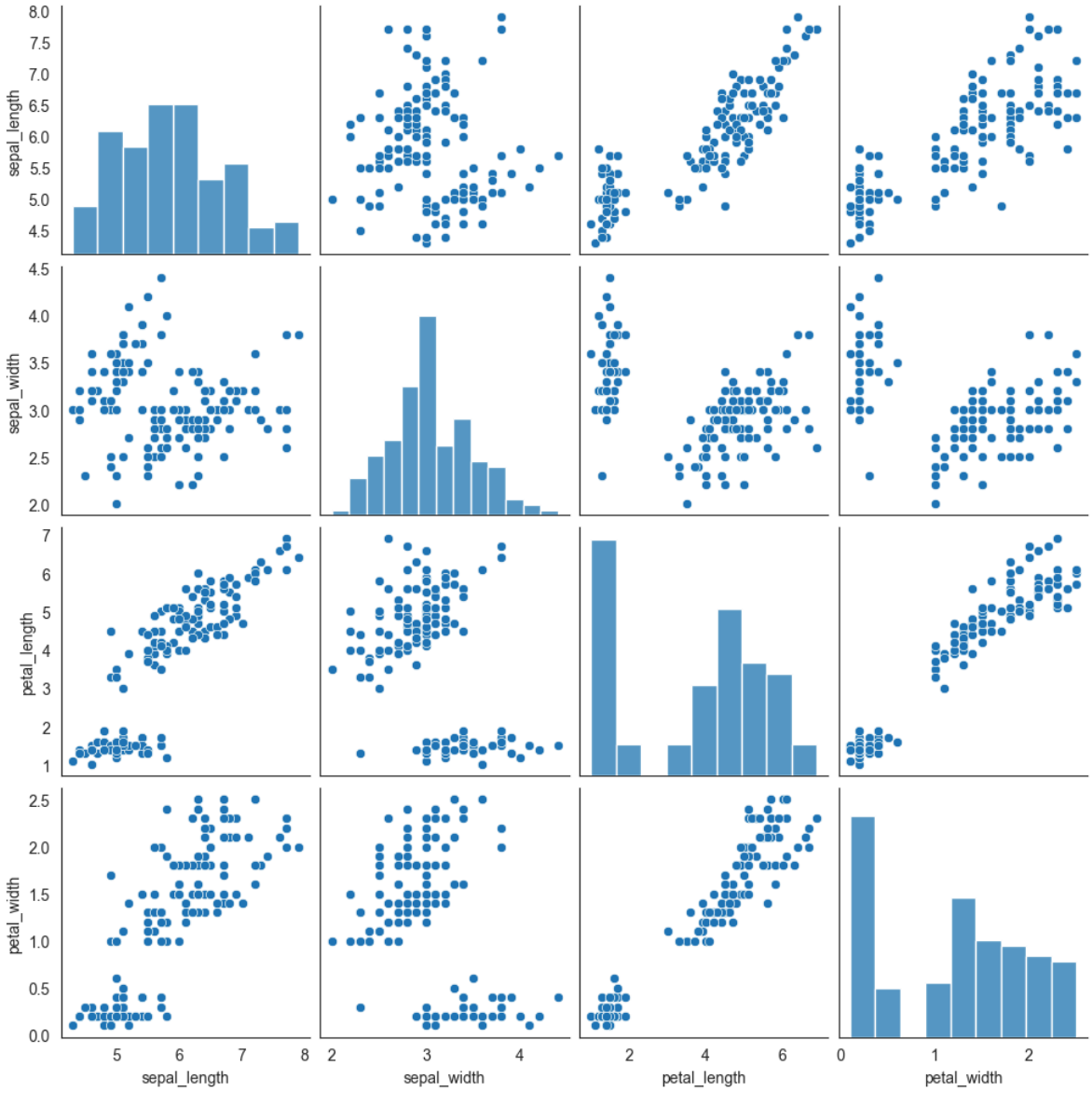
Out[3]:

	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	setosa
1	4.9	3.0	1.4	0.2	setosa

We get four dimensions (sepal_length, sepal_width, petal_length, petal_width). One direct way to visualize them is to have a scatter plot for each pair of dimensions. We can use the **`pairplot()`** (http://stanford.edu/~mwaskom/software/seaborn/generated/seaborn.pairplot.html) function in seaborn to do this.

```
In [4]: sns.pairplot(iris)
```

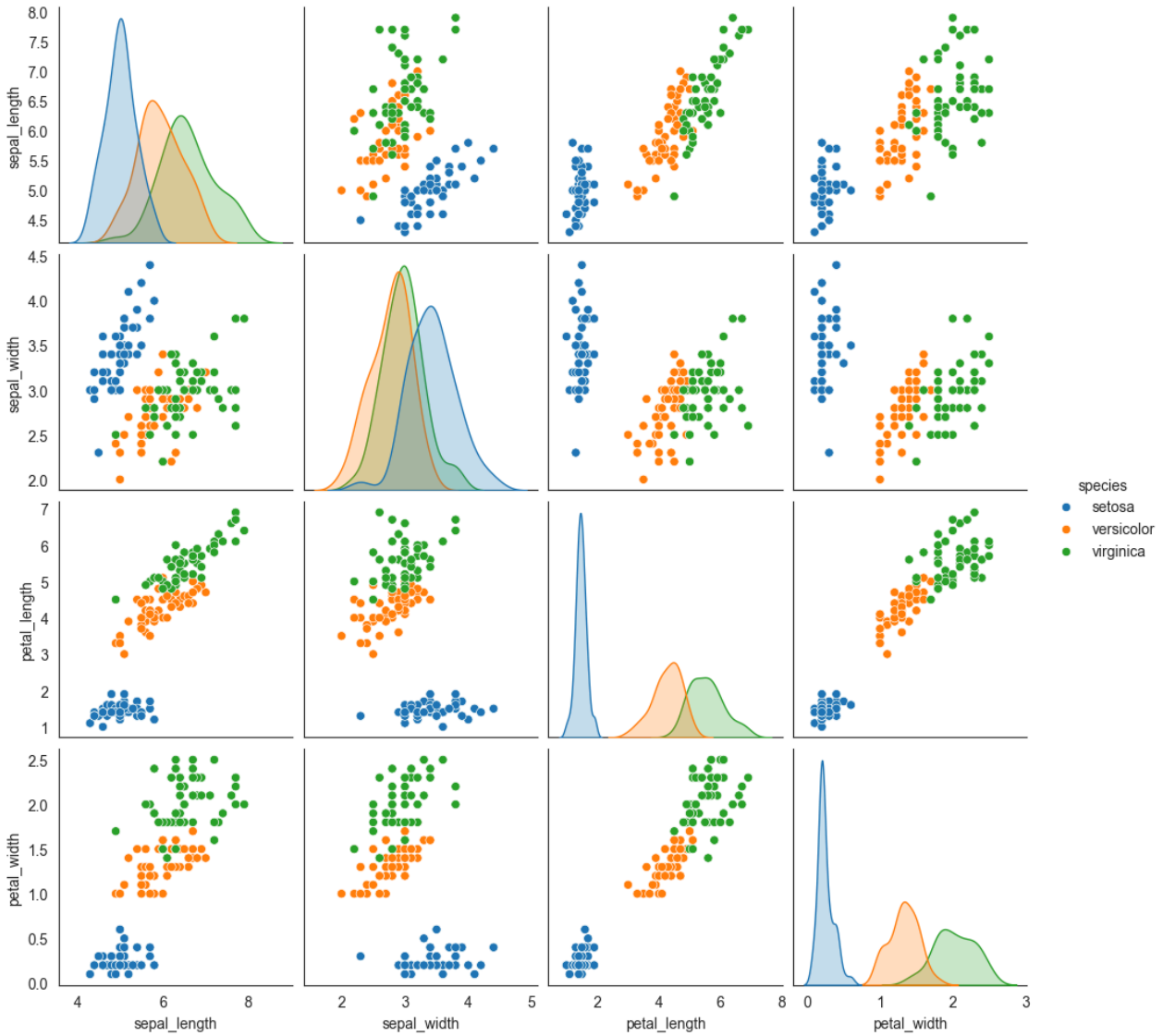
```
Out[4]: <seaborn.axisgrid.PairGrid at 0x24baf768a90>
```



By using colors, you can get a much more useful plot.

```
In [5]: sns.pairplot(iris, hue='species')
```

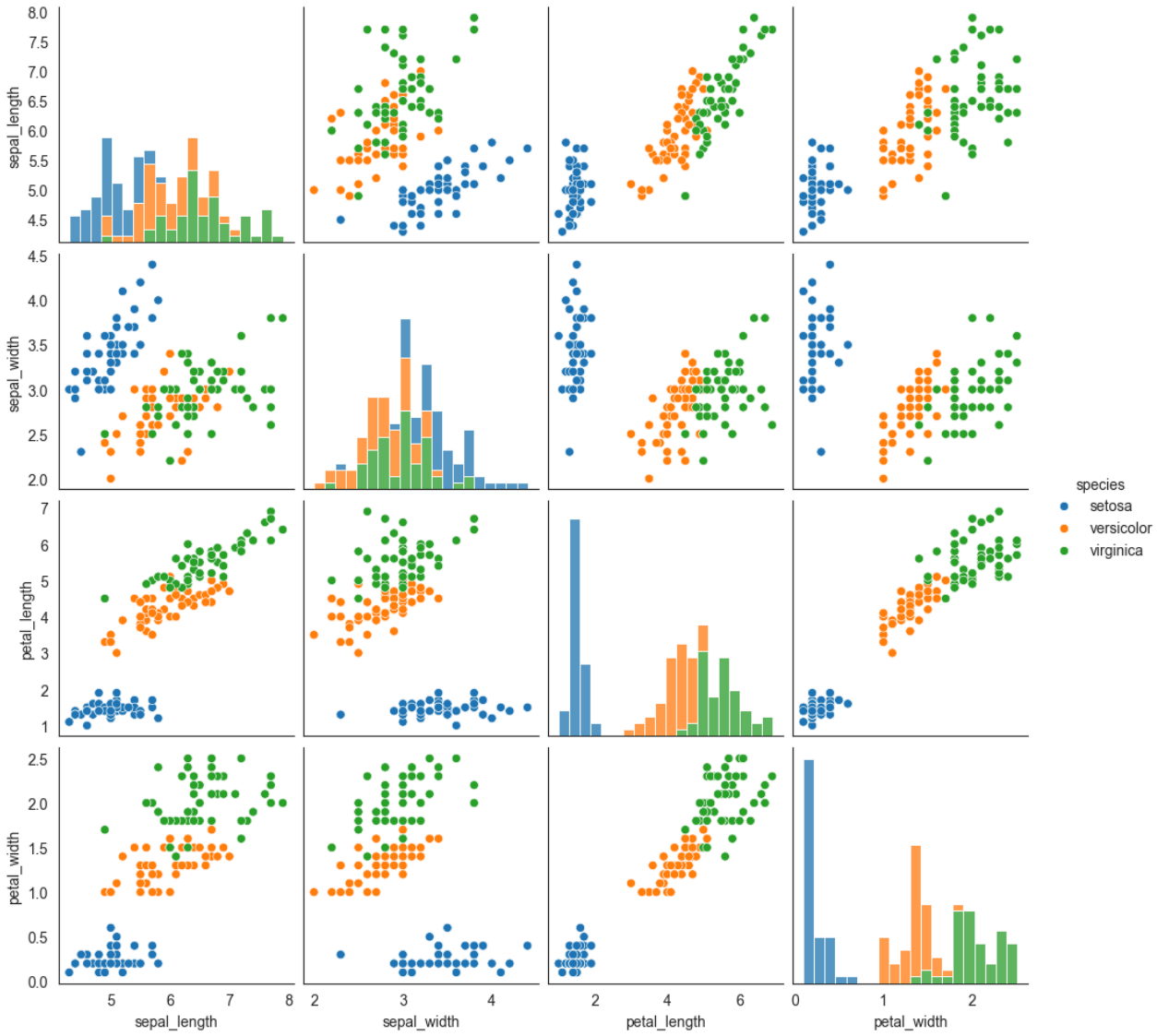
```
Out[5]: <seaborn.axisgrid.PairGrid at 0x24ba44d0520>
```



Seaborn also lets us to specify what to put in the diagonal. When `hue` is used, it defaults to KDE plot. We can change it back to histogram. See: https://seaborn.pydata.org/generated/seaborn.pairplot.html (https://seaborn.pydata.org/generated/seaborn.pairplot.html)

Q: draw a pairplot with hue and histogram on the diagonal

```
In [6]: sns.pairplot(iris, hue='species', diag_kind='hist', diag_kws={'multiple': 'stack', 'bins':20})
Out[6]: <seaborn.axisgrid.PairGrid at 0x24b9cca2e90>
```



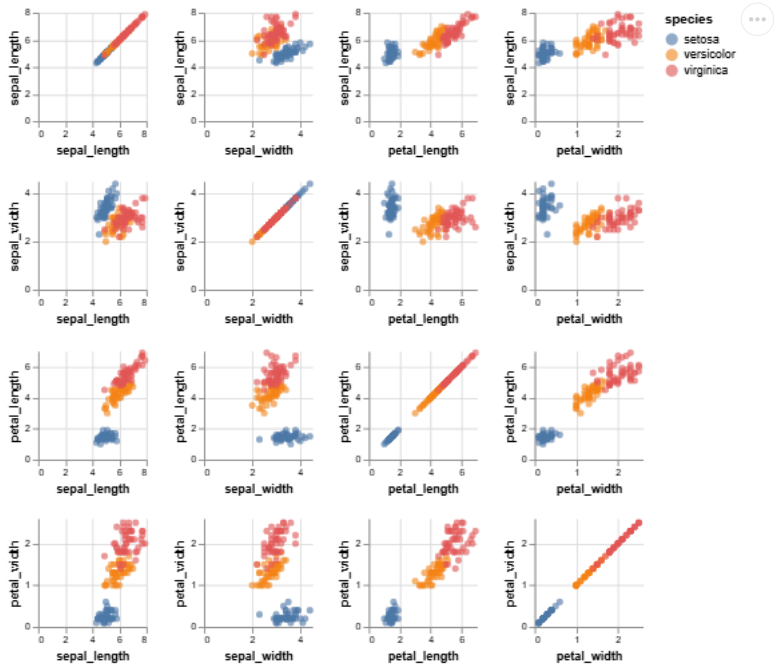
We can use altair to create an interactive scatterplot matrix. Can you create a scatterplot matrix of iris dataset by consulting https://altair-viz.github.io/gallery/scatter_matrix.html? (https://altair-viz.github.io/gallery/scatter_matrix.html?)

Q: Draw an interactive scatterplot matrix for iris dataset in altair

In [ ]:
```
import altair as alt

features = ['sepal_length', 'sepal_width', 'petal_length', 'petal_width']

charts = []

for y in features:
    row = []
    for x in features:
        chart = alt.Chart(iris).mark_circle(opacity=0.6).encode(
            x=alt.X(f'{x}:Q', title=x),
            y=alt.Y(f'{y}:Q', title=y),
            color=alt.Color('species:N')
        ).properties(width=90, height=90)
        row.append(chart)
    charts.append(alt.hconcat(*row))

pairplot = alt.vconcat(*charts).configure_axis(
    labelFontSize=8,
    titleFontSize=10
).configure_view(
    stroke=None
)

pairplot
```

```
c:\Users\Stefan\AppData\Local\Programs\Python\Python310\lib\site-packages\altair\utils\core.py:410: FutureWarning: the convert_dtype para
meter is deprecated and will be removed in a future version.  Do ``ser.astype(object).apply()`` instead if you want ``convert_dtype=False
``.
  col = df[col_name].apply(to_list_if_array, convert_dtype=False)
```

Out[ ]:



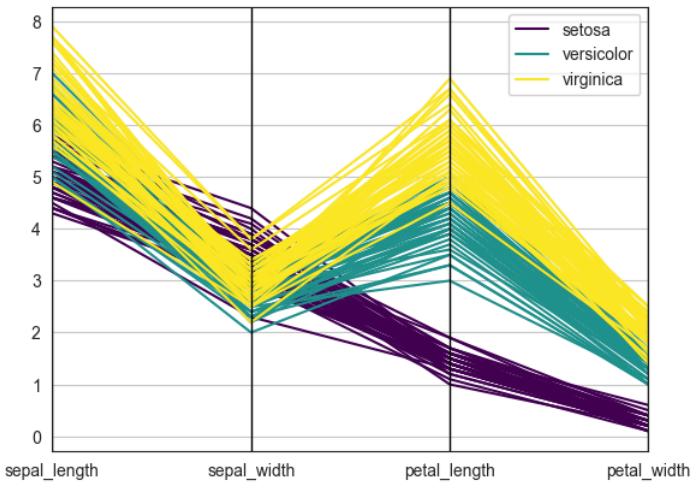
Parallel coordinates

Another useful visualization you can create with not-so-high-dimensional datasets is parallel coordinate visualization. Actually pandas supports parallel coordinate plots as well as "Andrews curve" (you can think of it as a smooth version of parallel coordinate.

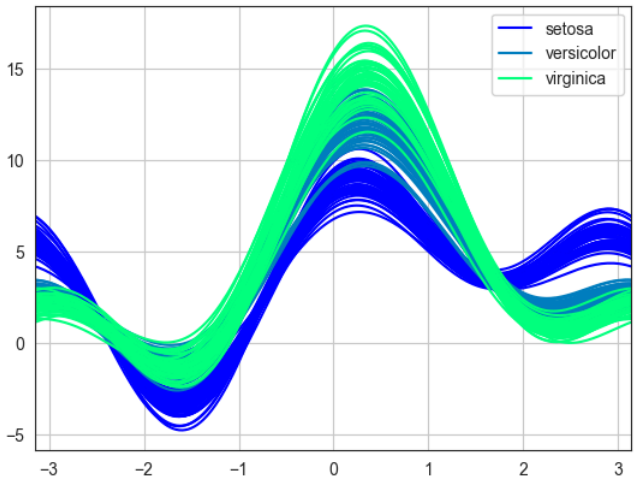
- https://pandas.pydata.org/pandas-docs/stable/visualization.html#parallel-coordinates (https://pandas.pydata.org/pandas-docs/stable/visualization.html#parallel-coordinates)
- https://pandas.pydata.org/pandas-docs/stable/visualization.html#andrews-curves (https://pandas.pydata.org/pandas-docs/stable/visualization.html#andrews-curves)

Q: Can you draw a parallel coordinate plot and a andrews curve plot of iris dataset? (The examples use the `viridis` and `winter` colormap.)

In [25]:
```
from pandas.plotting import parallel_coordinates, andrews_curves

parallel_coordinates(iris, "species", colormap='viridis');
```



In [26]:
```
andrews_curves(iris, "species", colormap="winter");
```

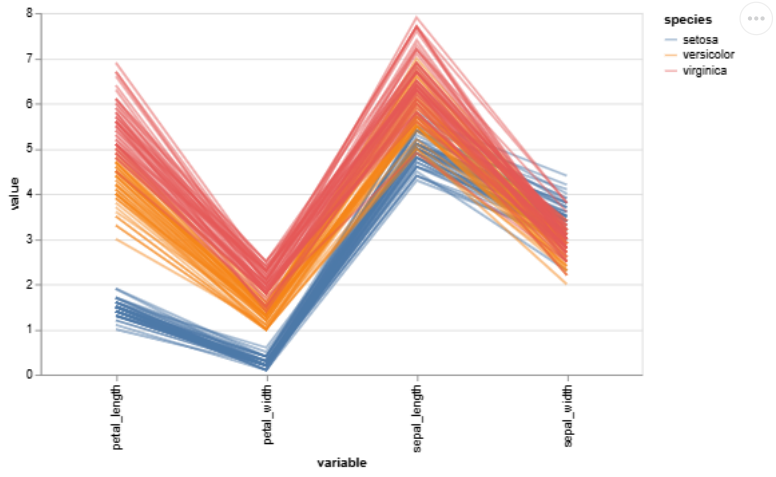


We can also use altair.

In [27]:
```
iris_transformed = iris.reset_index().melt(['species', 'index'])
alt.Chart(iris_transformed).mark_line().encode(
    x='variable:N',
    y='value:Q',
    color='species:N',
    detail='index:N',
    opacity=alt.value(0.5),
).properties(width=500)
```

```
c:\Users\Stefan\AppData\Local\Programs\Python\Python310\lib\site-packages\altair\utils\core.py:410: FutureWarning: the convert_dtype parameter is deprecated and will be removed in a future version.  Do ``ser.astype(object).apply()`` instead if you want ``convert_dtype=False``.
  col = df[col_name].apply(to_list_if_array, convert_dtype=False)
c:\Users\Stefan\AppData\Local\Programs\Python\Python310\lib\site-packages\altair\utils\core.py:410: FutureWarning: the convert_dtype parameter is deprecated and will be removed in a future version.  Do ``ser.astype(object).apply()`` instead if you want ``convert_dtype=False``.
  col = df[col_name].apply(to_list_if_array, convert_dtype=False)
```

Out[27]:



Q: can you explain how `iris_transformed` is different from the original `iris` dataset and why do we need to transform in this way?

In [33]:
```
iris_transformed.head(6)
```

Out[33]:

	species	index	variable	value
0	setosa	0	sepal_length	5.1
1	setosa	1	sepal_length	4.9
2	setosa	2	sepal_length	4.7
3	setosa	3	sepal_length	4.6
4	setosa	4	sepal_length	5.0
5	setosa	5	sepal_length	5.4

In [35]:
```
iris_transformed.query('index==0').head(6)
```

Out[35]:

	species	index	variable	value
0	setosa	0	sepal_length	5.1
150	setosa	0	sepal_width	3.5
300	setosa	0	petal_length	1.4
450	setosa	0	petal_width	0.2

The transformed representation assigns an index to each data point in the original frame, then breaks down the table into pairs (species, index, feature_name, value). It is useful when we want to treat multiple columns as the same data type or we need to deal with tidy data (which is what altair expects to be used with)

PCA

The principal component analysis (PCA) is the most basic dimensionality reduction method. For example, in the Iris dataset we have four variables (`sepal_length` , `sepal_width` , `petal_length` , `petal_width`). If we can reduce the number of variables to two, then we can easily visualize them in two dimensions. (See here (http://setosa.io/ev/principal-component-analysis/) a nice visual example of PCA)

PCA is already implemented in the scikit-learn (http://scikit-learn.org/stable/) package, a machine learning library in Python, which should have been included in Anaconda. If you don't have it, install it with:

```
conda install scikit-learn
```

or

```
pip install scikit-learn
```

In [36]:
```
iris.head(2)
```

Out[36]:

	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	setosa
1	4.9	3.0	1.4	0.2	setosa

This is a four dimensional data. To run the PCA we want to isolate only the numerical columns.

In [37]:
```
features = ['sepal_length', 'sepal_width', 'petal_length', 'petal_width']
iris_only_features = iris[features]
iris_only_features.head()
```

Out[37]:

	sepal_length	sepal_width	petal_length	petal_width
0	5.1	3.5	1.4	0.2
1	4.9	3.0	1.4	0.2
2	4.7	3.2	1.3	0.2
3	4.6	3.1	1.5	0.2
4	5.0	3.6	1.4	0.2

We should first create a PCA object and specify the number of components to obtain. Note that you can obtain more than two principal components.

In [38]:
```
from sklearn.decomposition import PCA
pca = PCA(n_components=2) # set the number of components to 2
```

Now you can run `fit()` method to identify principal components.

In [39]:
```
pca_iris_fitted = pca.fit(iris_only_features)
```

An important set of numbers that you want to look at is the *explained variance ratio*.

In [40]:
```
pca_iris_fitted.explained_variance_ratio_
```

Out[40]:
```
array([0.92461872, 0.05306648])
```

It tells you how much of the variance in the original dataset is explained by the principal components that you obtained. It seems like the first two components capture more than 95% of the variance in original dataset. This means that the PCA is very effective on this dataset and just using two principal components is a very good approximation to use all dimensions. Now you can use the result to transform the original dataset.

In [41]:
```
iris_pca = pca_iris_fitted.transform(iris_only_features)
iris_pca[:5]
```

Out[41]:
```
array([[-2.68412563,  0.31939725],
       [-2.71414169, -0.17700123],
       [-2.88899057, -0.14494943],
       [-2.74534286, -0.31829898],
       [-2.72871654,  0.32675451]])
```

A convenient way to do both fitting and transforming is

In [42]:
```
iris_pca = pca.fit_transform(iris_only_features)
iris_pca[:5]
```

Out[42]:
```
array([[-2.68412563,  0.31939725],
       [-2.71414169, -0.17700123],
       [-2.88899057, -0.14494943],
       [-2.74534286, -0.31829898],
       [-2.72871654,  0.32675451]])
```

You can see that this transformed matrix has two columns. Each column corresponds to the "loading" for one of the principal components.

In [49]:
```
iris_pca_df = pd.DataFrame(data=iris_pca, columns=['PC1', 'PC2'])
iris_pca_df.head()
```

Out[49]:

	PC1	PC2
0	-2.684126	0.319397
1	-2.714142	-0.177001
2	-2.888991	-0.144949
3	-2.745343	-0.318299
4	-2.728717	0.326755

Let's add the species information to the dataframe.

Q: add species column to `iris_pca_df` .

In [50]:
```
iris_pca_df.insert(2, "species", iris['species'])
iris_pca_df.head(5)
```

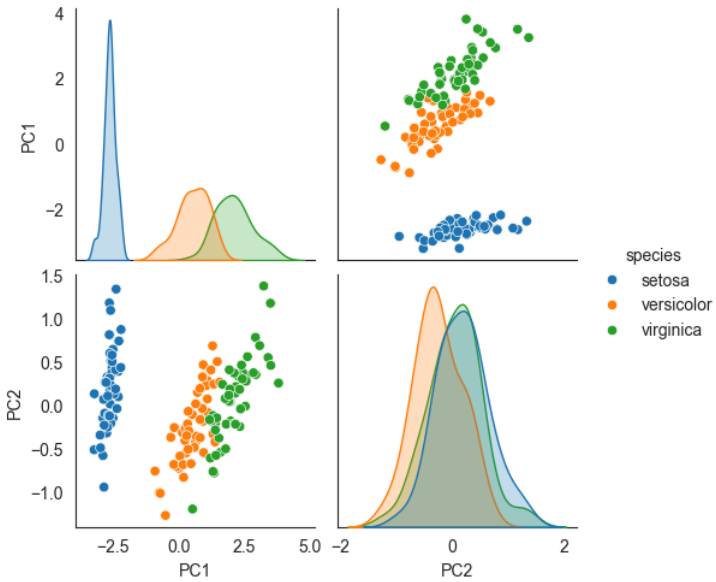
Out[50]:

	PC1	PC2	species
0	-2.684126	0.319397	setosa
1	-2.714142	-0.177001	setosa
2	-2.888991	-0.144949	setosa
3	-2.745343	-0.318299	setosa
4	-2.728717	0.326755	setosa

Now we can produce a scatterplot based on the two principal components. Let's just draw a pairplot.

In [51]:
```
sns.pairplot(iris_pca_df, hue='species')
```

Out[51]:
```
<seaborn.axisgrid.PairGrid at 0x24bbcde6b90>
```



The PC1 seems to capture inter-species variation while PC2 seems to capture intra-species variation.

PCA with pictures

Let's play with PCA with some pictures.

In [52]:
```
from sklearn.datasets import fetch_olivetti_faces

dataset = fetch_olivetti_faces(shuffle=True)
faces = dataset.data
```

```
downloading Olivetti faces from https://ndownloader.figshare.com/files/5976027 to C:\Users\Stefan\scikit_learn_data
```

In [53]:
```
n_samples, n_features = faces.shape
print(n_samples)
print(n_features)
```

```
400
4096
```

So, this dataset contains 400 faces, and each of them has 4096 features (=pixels). Let's look at the first face:

In [54]:
```
print(faces[0].shape)
faces[0]
```

```
(4096,)
```

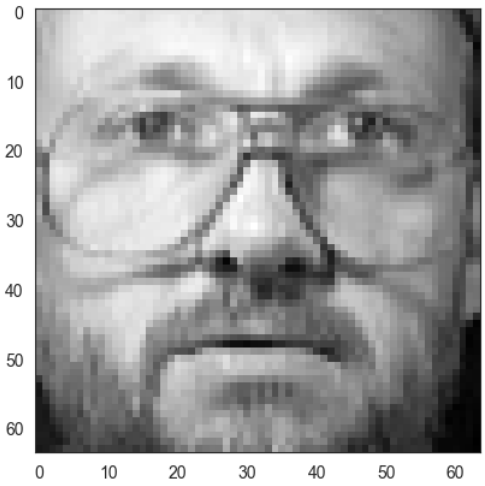
Out[54]:
```
array([0.6694215 , 0.6363636 , 0.6487603 , ..., 0.08677686, 0.08264463,
       0.07438017], dtype=float32)
```

It's an one-dimensional array with 4096 numbers. But a face should be a two-dimensional picture. Use `numpy` 's **`reshape()`** (http://docs.scipy.org/doc/numpy/reference/generated/numpy.reshape.html) function as well as `matplotlib` 's **`imshow()`** (http://matplotlib.org/api/pyplot_api.html#matplotlib.pyplot.imshow) function, transform this one-dimensional array into an appropriate 2-D matrix and draw it to show the face. You probably want to use `plt.cm.gray` as colormap.

Be sure to play with different shapes (e.g. 2 x 2048, 1024 x 4, 128 x 32, and so on) and think about why they look like what they look like. What is the right shape of the array?

Q: reshape the one-dimensional array into an appropriate two dimensional array and show the face

In [63]:
```
# plt.imshow(faces[0].reshape(16,256), cmap='gray')
# plt.imshow(faces[0].reshape(32,128), cmap='gray')
plt.imshow(faces[0].reshape(64,64), cmap='gray')
image_shape = (64,64)
```



Let's perform PCA on this dataset.

In [57]:
```
from sklearn.decomposition import PCA
```

Set the number of components to 6:

In [58]:
```
n_components=6
pca = PCA(n_components=n_components)
```

Fit the faces data:

In [59]:
```
pca.fit(faces)
```

Out[59]:
```
PCA(n_components=6)
```

PCA has an attribute called `components_` . It is a $\mathrm{n\_components} \times \mathrm{n\_features}$ matrix, in our case 6×4096. Each row is a component.

In [60]:
```
pca.components_
```

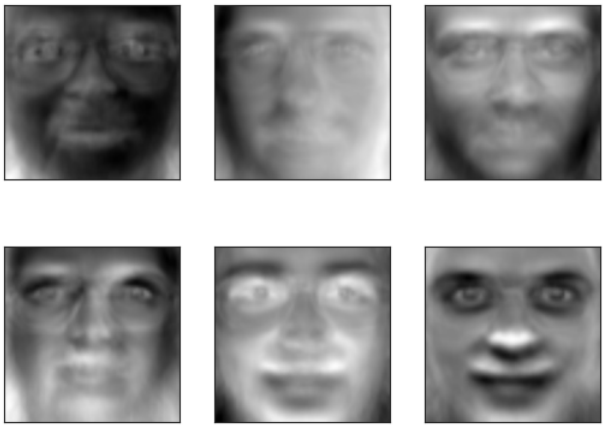
Out[60]:
```
array([[-0.00419107, -0.00710954, -0.00933614, ...,  0.00018519,
         0.00337968,  0.00318828],
       [-0.02859136, -0.03328836, -0.03784649, ...,  0.02962782,
         0.02721297,  0.02488898],
       [ 0.00135664, -0.00032584, -0.00019795, ..., -0.0154137 ,
        -0.01370983, -0.01188345],
       [ 0.00112415, -0.00179024, -0.01168216, ...,  0.02943   ,
         0.02781922,  0.02521859],
       [-0.02384371, -0.02359171, -0.02216185, ..., -0.04243895,
        -0.04007452, -0.04110378],
       [ 0.02909827,  0.03130254,  0.02877438, ..., -0.01635887,
        -0.01637537, -0.01491134]], dtype=float32)
```

In [61]:
```
pca.components_.shape
```

Out[61]:
```
(6, 4096)
```

We can display the 6 components as images:

In [64]:
```
for i, comp in enumerate(pca.components_, 1):
    plt.subplot(2, 3, i)
    plt.imshow(comp.reshape(image_shape), cmap=plt.cm.gray, interpolation='gaussian')
    plt.xticks(())
    plt.yticks(())
```



They are the "principal faces", which means that, by adding up these images with some appropriate weights, we can get a close approximation of the 400 images in the dataset!

We can get the coordinates of the 6 components to understand how each face is composed with the components.

In [65]:
```
faces_pca_transformed = pca.transform(faces)
```

In [66]:
```
faces_pca_transformed.shape
```

Out[66]:
```
(400, 6)
```

`faces_r` is a 400×6 matrix. Each row corresponds to one face, containing the coordinates of the 6 components. For instance, the coordinates for the first face is

In [67]:
```
faces_pca_transformed[0]
```

Out[67]:
```
array([ 0.81579113, -4.1440344 ,  2.4832683 , -0.9030864 ,  0.83138233,
        0.8863697 ], dtype=float32)
```

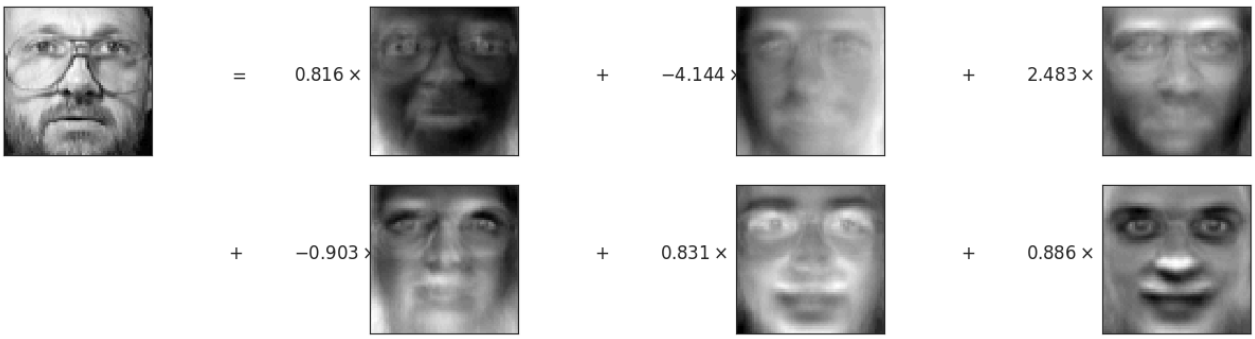
It seems that the second component (with coordinate 4.14403343) contributes the most to the first face. Let's display them together and see how similar they are:

In [68]:
```
# display the first face image
plt.subplot(1, 2, 1)
plt.imshow(faces[0].reshape(image_shape), cmap=plt.cm.gray, interpolation='gaussian')
plt.xticks(())
plt.yticks(())

# display the second component
plt.subplot(1, 2, 2)
plt.imshow(pca.components_[1].reshape(image_shape), cmap=plt.cm.gray, interpolation='gaussian')
plt.xticks(())
plt.yticks(())
```

Out[68]: ([], [])

We can display the composition of faces in an "equation" style:

In [69]:
```
from matplotlib import gridspec

def display_image(ax, image):
    ax.imshow(image, cmap=plt.cm.gray, interpolation='nearest')
    ax.set_xticks(())
    ax.set_yticks(())

def display_text(ax, text):
    ax.text(.5, .5, text, size=12)
    ax.axis('off')

face_idx = 0

plt.figure(figsize=(16,4))
gs = gridspec.GridSpec(2, 10, width_ratios=[5,1,1,5,1,1,5,1,1,5])

# display the face
ax = plt.subplot(gs[0])
display_image(ax, faces[face_idx].reshape(image_shape))

# display the equal sign
ax = plt.subplot(gs[1])
display_text(ax, r'$=$')

# display the 6 coordinates
for coord_i, gs_i in enumerate( [2,5,8,12,15,18] ):
    ax = plt.subplot(gs[gs_i])
    display_text( ax, r'$%.3f \times $' % faces_pca_transformed[face_idx][coord_i] )

# display the 6 components
for comp_i, gs_i in enumerate( [3,6,9,13,16,19] ):
    ax = plt.subplot(gs[gs_i])
    display_image( ax, pca.components_[comp_i].reshape(image_shape) )

# display the plus sign
for gs_i in [4,7,11,14,17]:
    ax = plt.subplot(gs[gs_i])
    display_text(ax, r'$+$')
```



We can directly see the results of this addition.

In [70]:
```
f, axes = plt.subplots(1, 6, figsize=(16,4))

faceid = 0

constructed_faces = []
for i in range(2, 10):
    constructed_faces.append(np.dot(faces_pca_transformed[faceid][:i], pca.components_[:i]))

# the face that we want to construct.
display_image(axes[0], faces[0].reshape(image_shape))

for idx, ax in enumerate(axes[1:]):
    display_image(ax, constructed_faces[idx].reshape(image_shape))
```

It becomes more and more real, although quite far with only several components.

NMF

There is another pretty cool dimensionality reduction method called NMF (Non-negative matrix factorization). It is widely used in many domains, such as identifying topics in documents, identifying key components in images, and so on. The key idea is by forcing every element in the decomposed matrices positive, NMF breaks something into **parts** that we can add together.

In [71]:
```
from sklearn.decomposition import NMF
n_components=20
nmf = NMF(n_components=n_components)
nmf_fitted = nmf.fit(faces)

for i, comp in enumerate(nmf_fitted.components_, 1):
    plt.subplot(4, 5, i)
    plt.imshow(comp.reshape(image_shape), cmap=plt.cm.gray, interpolation='gaussian')
    plt.xticks(())
    plt.yticks(())
```

```
c:\Users\Stefan\AppData\Local\Programs\Python\Python310\lib\site-packages\sklearn\decomposition\_nmf.py:1665: ConvergenceWarning: Maximum
number of iterations 200 reached. Increase it to improve convergence.
  warnings.warn(
```

As you can see here, each 'component' of NMF picks up a certain part of the face (light area), such as eyes, chin, nose, and so on. Very cool.

In [72]:
```
faces_nmf_tranformed = nmf_fitted.transform(faces)
```

In [73]:
```
faces_nmf_tranformed[0]
```

Out[73]:
```
array([4.8132297e-02, 5.4449666e-02, 0.0000000e+00, 6.3109472e-02,
       2.7885914e-02, 0.0000000e+00, 0.0000000e+00, 2.0505596e-02,
       3.3703959e-03, 4.0578868e-02, 5.2099656e-03, 1.4030471e-03,
       1.9796668e-02, 6.9160290e-02, 3.4936018e-02, 1.0364697e-01,
       7.6856710e-02, 3.4655118e-01, 2.2376062e-04, 2.1672850e-02],
      dtype=float32)
```

Can you show the reconstructed faces using the first n components, as we did for the PCA?

In [74]:
```
f, axes = plt.subplots(1, 8, figsize=(20,4))
faceid = 0
constructed_faces = []

for i in range(2, 10):
    constructed_faces.append(np.dot(faces_nmf_tranformed[faceid][:i], nmf.components_[:i]))

display_image(axes[0], faces[0].reshape(image_shape))

for idx, ax in enumerate(axes[1:]):
    display_image(ax, constructed_faces[idx].reshape(image_shape))
```

Unlike PCA that keeps superposing positive and negative images, NMF tends to gradually add multiple parts to the image. This is why it is widely used for many decomposing tasks such as detecting topics from documents.

t-SNE, Isomap, and MDS

Isomap, t-SNE, and MDS are nonlinear dimensionality reduction methods. Isomap preserves only the local relationships, MDS tries to preserve everything, and t-SNE is more flexible. t-SNE is very popular especially in machine learning.

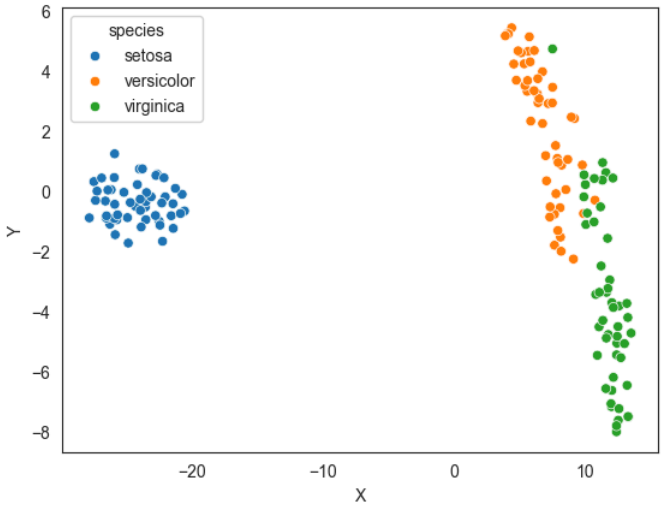
Some useful resources for t-SNE:

- Visualizing Data using t-SNE (http://www.jmlr.org/papers/volume9/vandermaaten08a/vandermaaten08a.pdf) (original paper)
- How to Use t-SNE Effectively (https://distill.pub/2016/misread-tsne/)
- An illustrated introduction to the t-SNE algorithm (https://www.oreilly.com/learning/an-illustrated-introduction-to-the-t-sne-algorithm)

Let's try t-SNE out with the iris data.

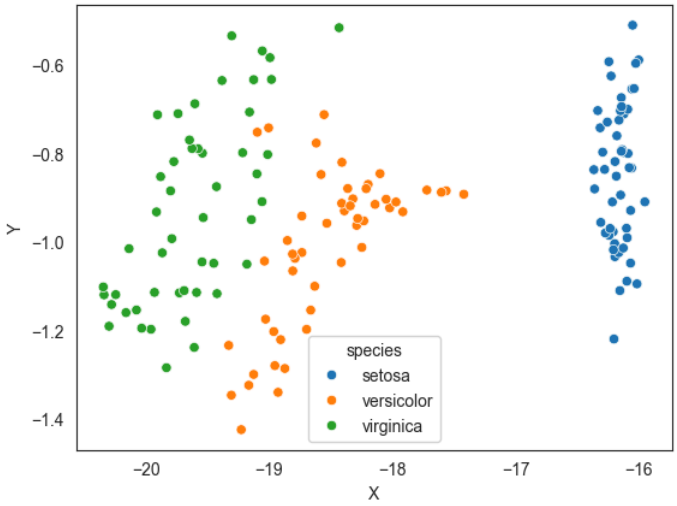
Q: Fit-transform the iris data with t-SNE and create a scatterplot of it.

In [90]:
```
from sklearn.manifold import TSNE
from sklearn.manifold import Isomap
from sklearn.manifold import MDS
from sklearn.datasets import load_iris

iris = sns.load_dataset('iris')

def run_tsne(perp=30):
    iris = sns.load_dataset('iris')

    tsne = TSNE(n_components=2, perplexity=perp)

    iris_tsne = tsne.fit_transform(iris_only_features)

    iris.head()

    iris_tsne_df = pd.DataFrame(data=iris_tsne, columns=['X', 'Y'])
    iris_tsne_df.head()
    iris_tsne_df.insert(2, "species", iris['species'])
    iris_tsne_df.head(5)

    sns.scatterplot(x='X', y='Y', data=iris_tsne_df, hue='species')
run_tsne()
```

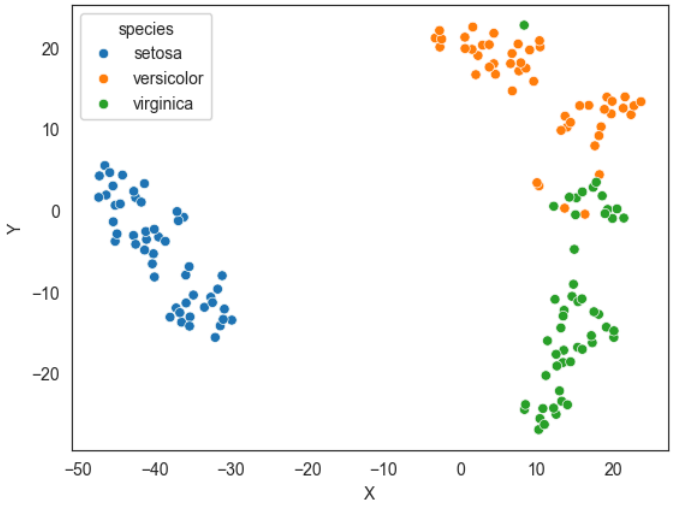


The hyperparameter `perplexity` determines how to balance attention between local and global aspects of your data. Changing this parameter (default is 30) may cause drastic changes in the output. Play with multiple values of perplexity.

In [91]:
```
run_tsne(100)
```



In [92]:
```
run_tsne(10)
```



If you want to learn more about t-SNE, play with https://distill.pub/2016/misread-tsne/ (https://distill.pub/2016/misread-tsne/) and https://experiments.withgoogle.com/visualizing-high-dimensional-space (https://experiments.withgoogle.com/visualizing-high-dimensional-space)

Visualizing the Digits dataset

This is a classic dataset of images of handwritten digits. It contains 1797 images with (8*8=64) pixels each.

In [93]:
```
from sklearn.datasets import load_digits

digits = load_digits()
digits.data.shape
```

Out[93]:
```
(1797, 64)
```

digits.data stores the images:

In [94]:
```
digits.data[0]
```

Out[94]:
```
array([ 0.,  0.,  5., 13.,  9.,  1.,  0.,  0.,  0.,  0., 13., 15., 10.,
       15.,  5.,  0.,  0.,  3., 15.,  2.,  0., 11.,  8.,  0.,  0.,  4.,
       12.,  0.,  0.,  8.,  8.,  0.,  0.,  5.,  8.,  0.,  0.,  9.,  8.,
        0.,  0.,  4., 11.,  0.,  1., 12.,  7.,  0.,  0.,  2., 14.,  5.,
       10., 12.,  0.,  0.,  0.,  0.,  6., 13., 10.,  0.,  0.,  0.])
```

and digits.target is the classes (or labels) that the images belong to. There are 10 classes in total.

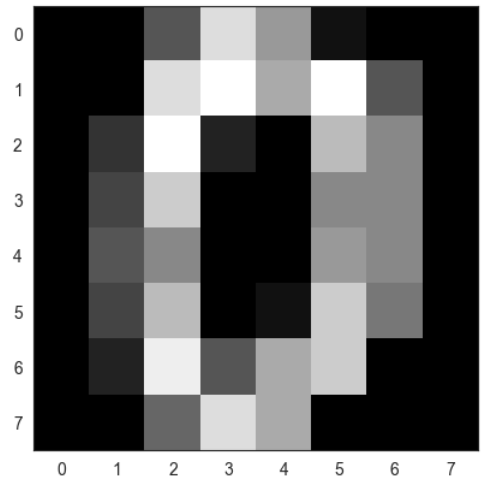
In [95]:
```
digits.target
```

Out[95]:
```
array([0, 1, 2, ..., 8, 9, 8])
```

Q: use `imshow` to display the first image.

In [99]:
```
# Implement
plt.imshow(digits.data[0].reshape(8,8), cmap='gray')
```

Out[99]:
```
<matplotlib.image.AxesImage at 0x24bc0829c30>
```



Let's first reorder the data points according to the handwritten numbers. We can use np.vstack (https://docs.scipy.org/doc/numpy/reference/generated/numpy.vstack.html) and np.hstack (https://docs.scipy.org/doc/numpy/reference/generated/numpy.hstack.html).

In [100]:
```
X = np.vstack([digits.data[digits.target==i]
               for i in range(10)])
y = np.hstack([digits.target[digits.target==i]
               for i in range(10)])
```

Then initialize a tsne model. For the meaning of the parameters see here (http://scikit-learn.org/stable/modules/generated/sklearn.manifold.TSNE.html).

In [101]:
```
tsne = TSNE(n_components=2, init='pca', random_state=0)
```

A models like tsne has a *lot* of parameters. By fitting it on the data, we are selecting the "best" parameters which minimize a certain objective function. For example, when fitting a linear regression model, we are selecting a line that minimizes the sum of squared errors. Here after we select the best parameters for tsne, we also obtain the clusters found by this model. calling `fit_transform` is equivilent to calling `fit` and then `transform` .

In [102]:
```
digits_proj = tsne.fit_transform(X)
```

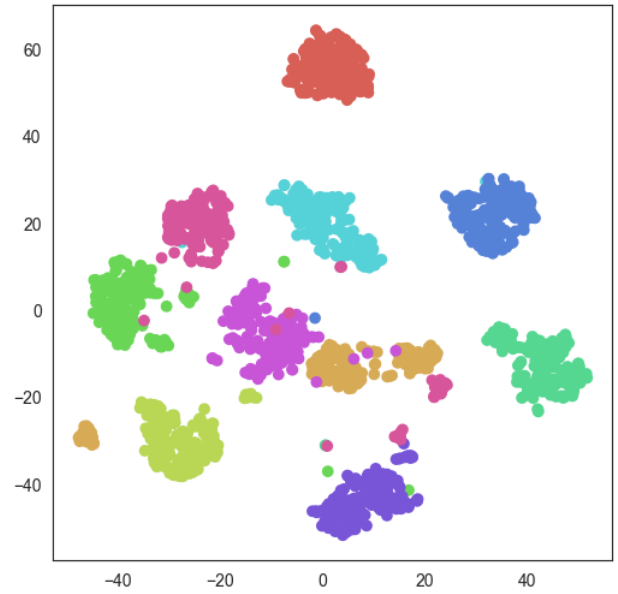
Plot the results. Seaborn's hls (http://seaborn.pydata.org/generated/seaborn.hls_palette.html#seaborn.hls_palette) palatte provides evenly spaced colors in HLS hue space, we can divide it into 10 colors.

In [103]:
```
palette = np.array(sns.color_palette("hls", 10))
```

Make a scatter plot of the first component against the second component, with color based on the numbers.

In [104]:
```
plt.figure(figsize = (6,6))
plt.scatter(digits_proj[:,0], digits_proj[:,1],c=palette[y])
```

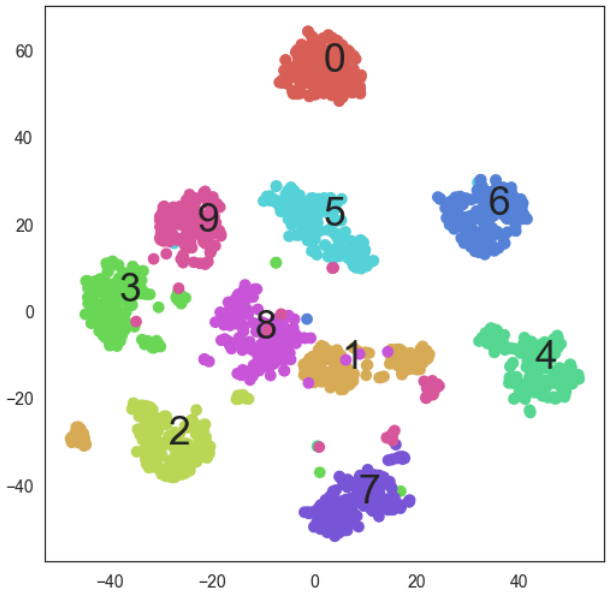
Out[104]:
```
<matplotlib.collections.PathCollection at 0x24bc085e020>
```



We can add some text for each cluster. The place of the text can be the center of the cluster. We can use `np.median` to find the centers. To simplify things, we can make the code into a function.

In [105]:
```
def plot_scatter(projection):
    plt.figure(figsize = (6,6))
    plt.scatter(projection[:,0], projection[:,1],c=palette[y])
    for i in range(10):
        # Position of each label.
        xtext, ytext = np.median(projection[y == i, :], axis=0)
        txt = plt.text(xtext, ytext, str(i), fontsize=24)
```

In [106]:
```
plot_scatter(digits_proj)
```



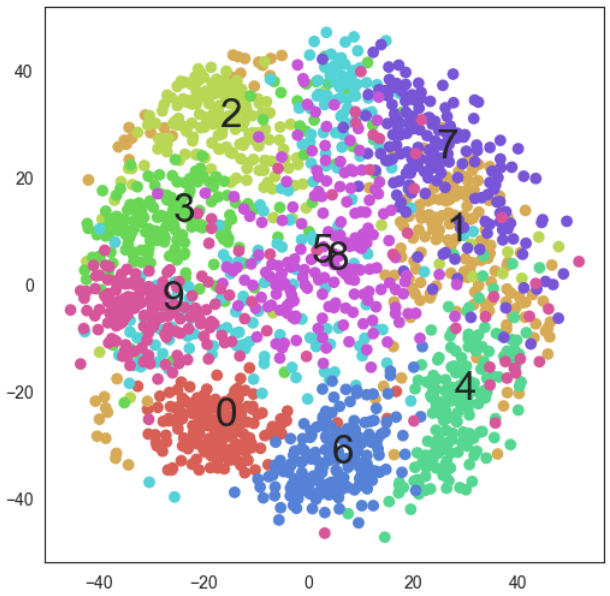
Comparison with Isomap and MDS

We talked about MDS and Isomap in class as two other manifold learning methods. Sklearn also has implementations for this two algorithms: MDS (http://scikit-learn.org/stable/modules/generated/sklearn.manifold.MDS.html) and Isomap (http://scikit-learn.org/stable/modules/generated/sklearn.manifold.Isomap.html), so the usage is very similar. Examples for using this methods can be found here (http://scikit-learn.org/stable/auto_examples/manifold/plot_lle_digits.html).

Can you make another two plots with these two methods? You only need to change the models and call the `plot_scatter` function,

In [ ]:
```
mds = MDS(n_components=2)
digits_proj = mds.fit_transform(X)
plot_scatter(digits_proj)
```

```
c:\Users\Stefan\AppData\Local\Programs\Python\Python310\lib\site-packages\sklearn\manifold\_mds.py:299: FutureWarning: The default value
of `normalized_stress` will change to ``'auto'`` in version 1.4. To suppress this warning, manually set the value of `normalized_stress`.
  warnings.warn(
```



In [108]:
```
iso = Isomap(n_components=2)
digits_proj = iso.fit_transform(X)
plot_scatter(digits_proj)
```

```
c:\Users\Stefan\AppData\Local\Programs\Python\Python310\lib\site-packages\sklearn\manifold\_isomap.py:373: UserWarning: The number of con
nected components of the neighbors graph is 2 > 1. Completing the graph to fit Isomap might be slow. Increase the number of neighbors to
avoid this issue.
  self._fit_transform(X)
c:\Users\Stefan\AppData\Local\Programs\Python\Python310\lib\site-packages\scipy\sparse\_index.py:103: SparseEfficiencyWarning: Changing t
he sparsity structure of a csr_matrix is expensive. lil_matrix is more efficient.
  self._set_intXint(row, col, x.flat[0])
```