Overview of dimensionality reduction methods

This personal project aims to explore existing methods of dimensionality reduction. While the Principal Component Analysis (PCA) is most useful for identifying patterns in data, finding their similarities and differences, the same task can prove difficult for image processing. In practice a dataset to be decomposed could be too large to fit into computer's memory, as is the case in image collections and visual processing in general.

Moreover, in practice, especially in visual processing, data may be available only incrementally (e.g. in gesture recognition). Additionally, in practice data are more likely to have non-linear relationships, yet PCA is limited to capturing only linear relationships. The nonlinearity is commonly handled with the use of non-linear kernels in PCA. However, to my knowledge this approach of appliying non-linear kernels in case of IPCA (and incremental data) has not been done before, thus this is a secondary objective of the project.

This study could be applied to any domain that works with high-dimensional data, incremental data, and non-linear data. Since I am highly interested in the Robotics Field, then a good combination of two would be visual data in Computer Vision.

This research projects aims to expand on the Machine Learning course content by exploring, comparing and testing various existing methods of dimentionality reduction on non-linear data and data that is either available incrementally or is too large to fit in memory.

To use this notebook, please refer to the github repository that contains all necessary data.

Table of Contents

Datasets

To study and compare various methods of Dimentionality Reduction I use the same datasets for all the considered methods.

The considered datasets are:

- Flower dataset consisting of 5 types of flower images.
- Star dataset consisting of various physical properties of stars that could determine the type of the star
- Fashion MNIST dataset in `.csv` format
- MNIST digits dataset in `.csv` format.

All datasets are designed for a classification task.

Import libraries

In [ ]:
```
from sklearn.decomposition import PCA, IncrementalPCA, KernelPCA
from sklearn.preprocessing import LabelEncoder

import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import cv2
import os
from tqdm import tqdm
```

Data Preprocessing

Helper function for plotting and dataset imports

In [ ]:
```
def plotting(reduced_data: np.array, labels: np.array ,title: str, label_dict: dict) -> None:
    plt.figure(figsize=(8,5))  # set up the figure and its size
    scatter = plt.scatter(reduced_data[:,0], reduced_data[:,1],  c=labels, cmap='jet')

    plt.legend(scatter.legend_elements()[0], label_dict.values(), loc="lower right", title="Classes")
    plt.title(title)

    plt.show()

def plot_comparison(reduced_data: list, num_plots: int, targets: list ,title: str, subtitles: list, label_dict:
    fig, ax = plt.subplots(1,num_plots, figsize=(8+2*num_plots,5))
    for i in range(num_plots):
        scatter = ax[i].scatter(reduced_data[i][:,0], reduced_data[i][:,1], c=targets[i], cmap='jet')
        ax[i].set_title(subtitles[i])
        ax[i].legend(scatter.legend_elements()[0], label_dict[i].values(), loc="lower right", title="Classes")
    fig.suptitle(title)

    plt.show()



def label_assignment(img,label):
    return label

def training_data(label,data_dir, imgsize=150):
    for img in tqdm(os.listdir(data_dir)):
        label = label_assignment(img,label)
        path = os.path.join(data_dir,img)
        img = cv2.imread(path,cv2.IMREAD_COLOR)
        img = cv2.resize(img,(imgsize,imgsize))
        #print(np.array(img).shape)

        X.append(np.array(img))
        Z.append(str(label))
```

Flower Dataset

To test how the considered methods behave with high-resolution image dataset, I am using the Flower dataset for image classification. The dataset is stored in `/flowers` directory, which consists of 5 sub-directories for each class respectively.

Code for image parsing taken from here

In [ ]:
```
# (flower) train set paths
daisy_dir = './datasets/flowers/daisy/'
dandelion_dir = './datasets/flowers/dandelion/'
rose_dir = './datasets/flowers/rose/'
sunflower_dir = './datasets/flowers/sunflower/'
tulip_dir = './datasets/flowers/tulip/'

# set up constants
imgsize = 350     # resize large images to 150x150
RANDOM_STATE = 42
n_components = 2  # used for most methods to display of a 2D-plot
```

In [ ]:
```
# Label dictionary for plot's legend
flower_dict = {
    0: 'daisy',
    1: 'dandelion',
    2: 'rose',
    3: 'sunflower',
    4: 'tulip',
}
```

In [ ]:
```
# Set up lists to store data in
X, Z = [], []

# load train data (display progress for all folders)
training_data('daisy',daisy_dir, imgsize=imgsize)
training_data('dandelion',dandelion_dir, imgsize=imgsize)
training_data('rose',rose_dir, imgsize=imgsize)
training_data('sunflower',sunflower_dir, imgsize=imgsize)
training_data('tulip',tulip_dir, imgsize=imgsize)
```

```
  7%|█         | 36/501 [00:00<00:02, 186.37it/s]100%|██████████| 501/501 [00:02<00:00, 248.28it/s]
100%|██████████| 646/646 [00:02<00:00, 277.33it/s]
100%|██████████| 497/497 [00:01<00:00, 299.20it/s]
100%|██████████| 495/495 [00:01<00:00, 270.83it/s]
100%|██████████| 607/607 [00:02<00:00, 282.94it/s]
```

In [ ]:
```
# extract labels (flower types)
label_encoder= LabelEncoder()
flower_labels = label_encoder.fit_transform(Z)
X = np.array(X)
```

To understand, how the flower dataset is stored, display its shape.

In [ ]:
```
X.shape
```

Out[ ]:
```
(2746, 350, 350, 3)
```

For the PCA to work, the number of the dataset's dimensions should be ≤ 2.

As displayed above, `X` has 4 dimensions:

- The first represents the number of images in the dataset
- The second and the third - the size of a single image
- The last - number of color channels in a single image (RGB)

Therefore, to fit the data for the PCA:

- Convert all images to grayscale to reduce the number of color channels, subsuquently reducing the number dimensions by 1
- Flatten the images from 2-dimensional 150 x 150 sized images to 1-D $22500 = 150^2$ sized arrays.

Formula for converting a RGB image to grayscale source

In [ ]:
```
def rgb2gray(rgb_image: np.array) -> np.array:
    r, g, b = rgb_image[:,:,0], rgb_image[:,:,1], rgb_image[:,:,2]
    gray = 0.2989 * r + 0.5870 * g + 0.1140 * b  # see formula source above
    return gray


def flatten_and_gray(img: np.array) -> np.array:
    gray = rgb2gray(img)
    flat_img = gray.flatten()
    return flat_img
```

In [ ]:
```
# Apply flattening and grayscaling to all high-resolution images

flower_train = []

# save flatten and grayscale images
for i in range(X.shape[0]):
    flower_train.append(flatten_and_gray(X[i]))

flower_train = np.array(flower_train)
flower_train.shape
```

Out[ ]:
```
(2746, 122500)
```

Fashion MNIST

In [ ]:
```
# read data
path = "./datasets/fashion-mnist_train.csv"
fashion_train_large = pd.read_csv(path)
fashion_train_large.head() # 28x28 images (= 784 pixels + 1 label column)
```

Out[ ]:

	label	pixel1	pixel2	pixel3	pixel4	pixel5	pixel6	pixel7	pixel8	pixel9	...	pixel775	pixel776	pixel777	pixel778	pixel779	pi
0	2	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	
1	9	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	
2	6	0	0	0	0	0	0	0	5	0	...	0	0	0	30	43	
3	0	0	0	0	1	2	0	0	0	0	...	3	0	0	0	0	
4	3	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	

5 rows × 785 columns

In [ ]:
```python
# separate labels from the dataframe
fashion_labels_large = fashion_train_large['label']
del fashion_train_large['label']
```

In [ ]:
```python
# Choose only some portion of data so that most dimension reduction algorithms execute fast
# But save the original large data for future IPCA and PCA comparison
fashion_train = fashion_train_large.iloc[0:5000,:]
fashion_labels = fashion_labels_large.iloc[0:5000]
```

In [ ]:
```python
# Translating numerical labels into an understandable labels using a dictionary
fashion_dict = {
    0: 'Top',
    1: 'Trouser',
    2: 'Pullover',
    3: 'Dress',
    4: 'Coat',
    5: 'Sandal',
    6: 'Shirt',
    7: 'Sneaker',
    8: 'Bag',
    9: 'Boot'
}
```

Digits Dataset

In [ ]:
```python
# read data
path = "./datasets/digits_train.csv"
digits_train = pd.read_csv(path)
digits_train.head() # 28x28 images (= 784 pixels + 1 label column)
```

Out[ ]:

	label	pixel0	pixel1	pixel2	pixel3	pixel4	pixel5	pixel6	pixel7	pixel8	...	pixel774	pixel775	pixel776	pixel777	pixel778	pi
0	1	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	
1	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	
2	1	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	
3	4	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	
4	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	

5 rows × 785 columns

In [ ]:
```python
# separate target column from the dataframe
digits_target = digits_train['label']
del digits_train['label']
```

In [ ]:
```python
# The quality of the wine varies from 3 to 8 in the original dataset
digits_dict = {i:i for i in range(10)} # dummy dictionary for digits labels
```

Star dataset

In [ ]:
```python
# read data
star_path = "./datasets/stars.csv"
star_train = pd.read_csv(star_path)
star_train.head()
```

Out[ ]:

	Temperature (K)	Luminosity(L/Lo)	Radius(R/Ro)	Absolute magnitude(Mv)	Star type	Star color	Spectral Class
0	3068	0.002400	0.1700	16.12	0	Red	M
1	3042	0.000500	0.1542	16.60	0	Red	M
2	2600	0.000300	0.1020	18.70	0	Red	M
3	2800	0.000200	0.1600	16.65	0	Red	M
4	1939	0.000138	0.1030	20.06	0	Red	M

In [ ]:
```
# separate target column from the dataframe
star_target = star_train['Star type']
del star_train['Star type']
```

In [ ]:
```
from sklearn.preprocessing import OrdinalEncoder

# Encode the categorical columns
encoder = OrdinalEncoder()
categ_cols = ["Star color", 'Spectral Class']
star_train[categ_cols] = encoder.fit_transform(star_train[categ_cols])
star_train.head()
```

Out[ ]:

	Temperature (K)	Luminosity(L/Lo)	Radius(R/Ro)	Absolute magnitude(Mv)	Star color	Spectral Class
0	3068	0.002400	0.1700	16.12	10.0	5.0
1	3042	0.000500	0.1542	16.60	10.0	5.0
2	2600	0.000300	0.1020	18.70	10.0	5.0
3	2800	0.000200	0.1600	16.65	10.0	5.0
4	1939	0.000138	0.1030	20.06	10.0	5.0

In [ ]:
```
# Label dictionary for plot's legend
star_dict = {
    0: 'Brown Dwarf',
    1: 'Red Dwarf',
    2: 'White Dwarf',
    3: 'Main Sequence',
    4: 'Supergiant',
    5: 'Hypergiant'
}
```

Principal Component Analysis

Classic PCA

PCA is a technique used to reduce data dimensionality of a set of correlated variables to project it to a lower dimensional space. The input data is centered but not scaled for each feature before applying the SVD.

It identifies the directions (principal components) in which the data varies the most and projects the data onto these components. The resulting components are orthogonal and capture the maximum amount of variance in the data. PCA can be used for data visualization, noise reduction, and feature extraction.

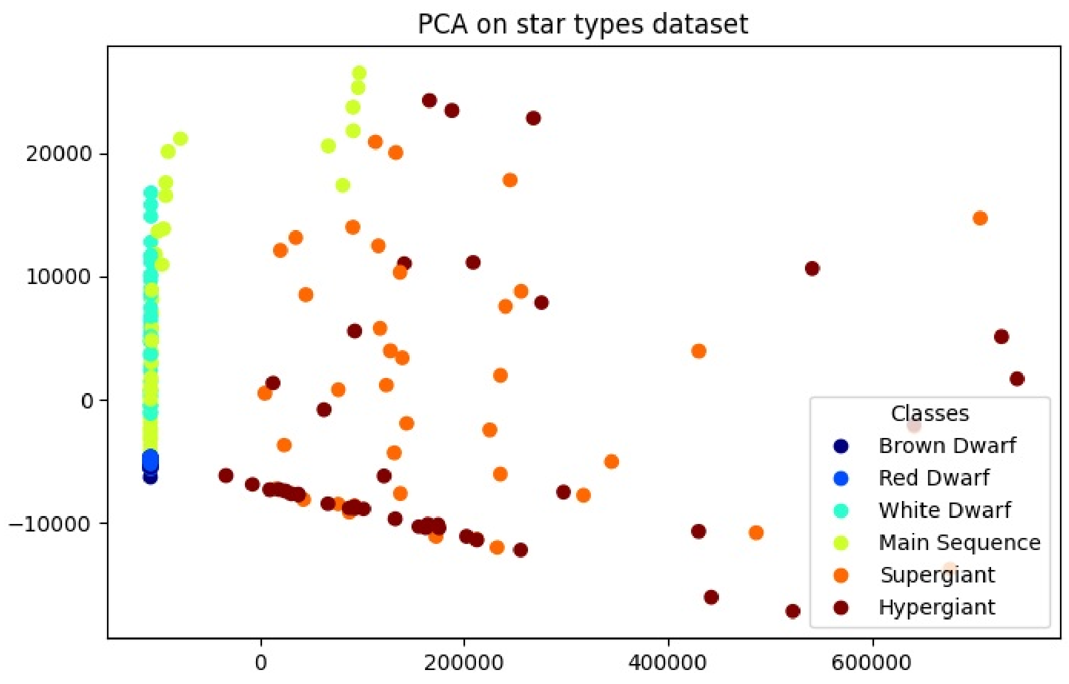
Source: scikit-documentation

In [ ]:
```
# Initialize PCA to use on all datasets
pca = PCA(n_components=n_components, random_state=RANDOM_STATE, whiten=False)
```

PCA on Star Types dataset

In [ ]:
```
# Fit the initialized PCA to star data
star_pca = pca.fit_transform(star_train)
```

In [ ]:
```
plotting(star_pca, star_target, 'PCA on star types dataset', star_dict)
```



According to PCA - super- or hypergiants are very different from normal stars and dwarfs! And Hypergiants seem to loosely follow a line.

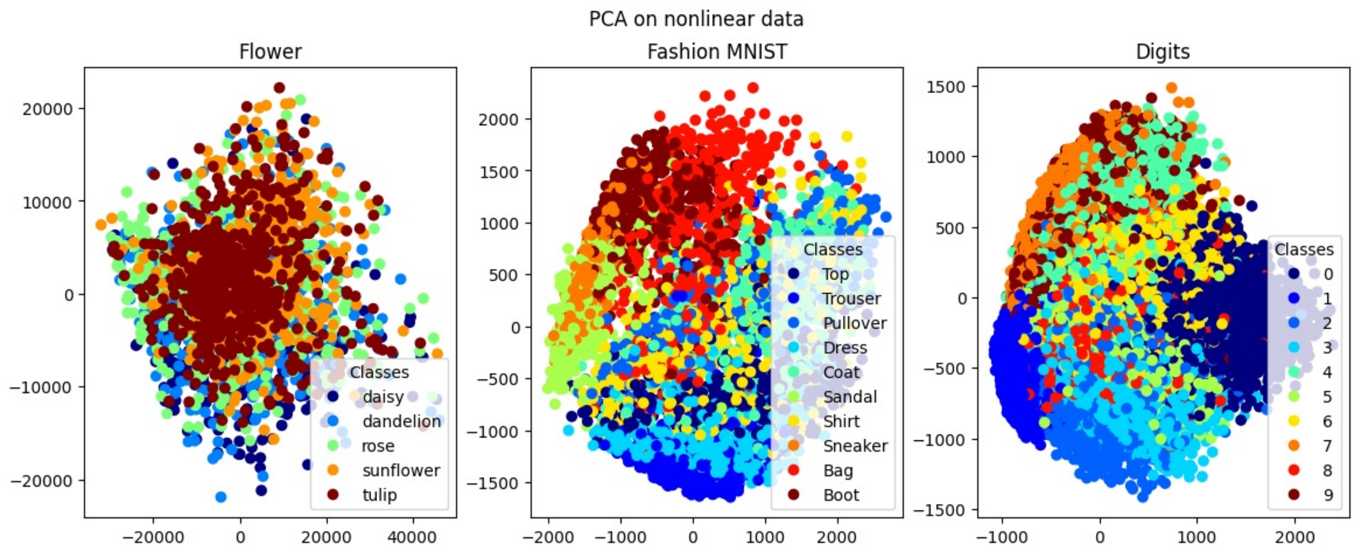
However, that's all the information we can get.

PCA on images

```
In [ ]: # fit PCA to image datasets

        digits_pca = pca.fit_transform(digits_train)
        flower_pca = pca.fit_transform(flower_train)
        fashion_pca = pca.fit_transform(fashion_train)
```

For the nonlinear data: while some clusters of classes are visible, they are still heavily intertwined with other data, with no clear borders

```
In [ ]: plot_comparison(
            reduced_data=[flower_pca, fashion_pca, digits_pca],
            num_plots=3,
            targets=[flower_labels, fashion_labels, digits_target],
            title = 'PCA on nonlinear data',
            subtitles=['Flower', 'Fashion MNIST', 'Digits'],
            label_dict=[flower_dict, fashion_dict, digits_dict]
            )
```



As PCA is not well fit for non-linear data, a different dimensionality reduction method should be considered. The most straightforward option is to introduce a different kernel to the PCA to work around the nonlinearity with `KernelPCA`

Kernel PCA

By the use of integral operator kernel functions, one can effciently compute principal components in high-dimensional feature spaces, related to input space by some nonlinear map. For instance the space of all possible d-pixel products in images.

Source: Kernel Principal Component Analysis

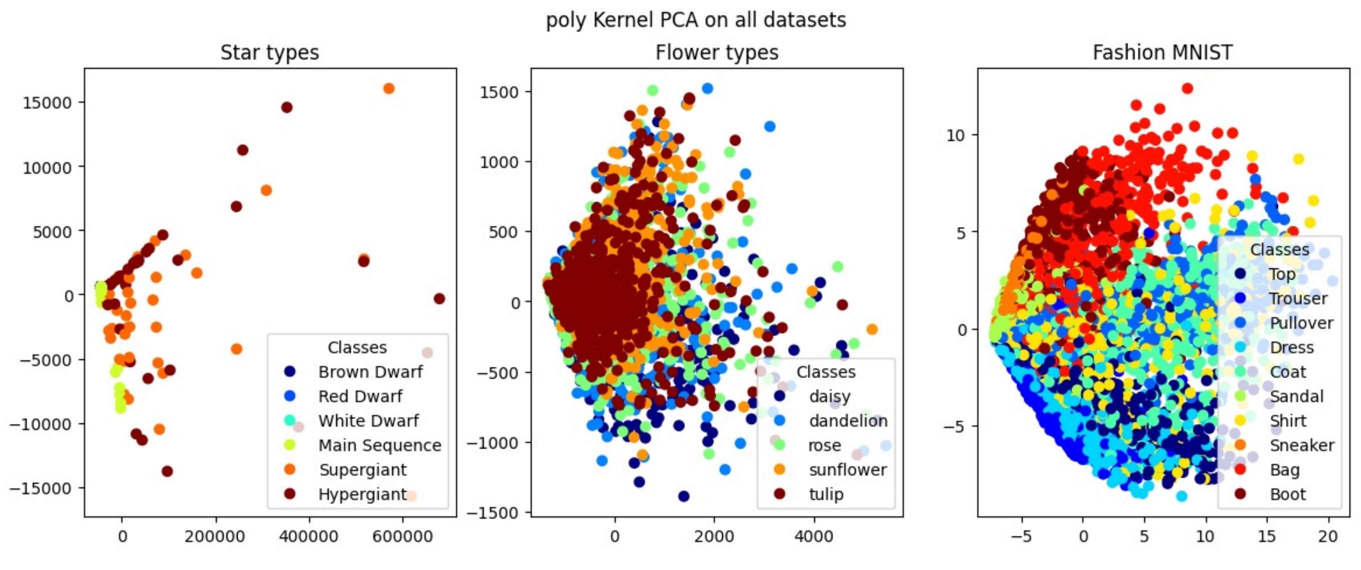
The following is a list of available kernels used for PCA:

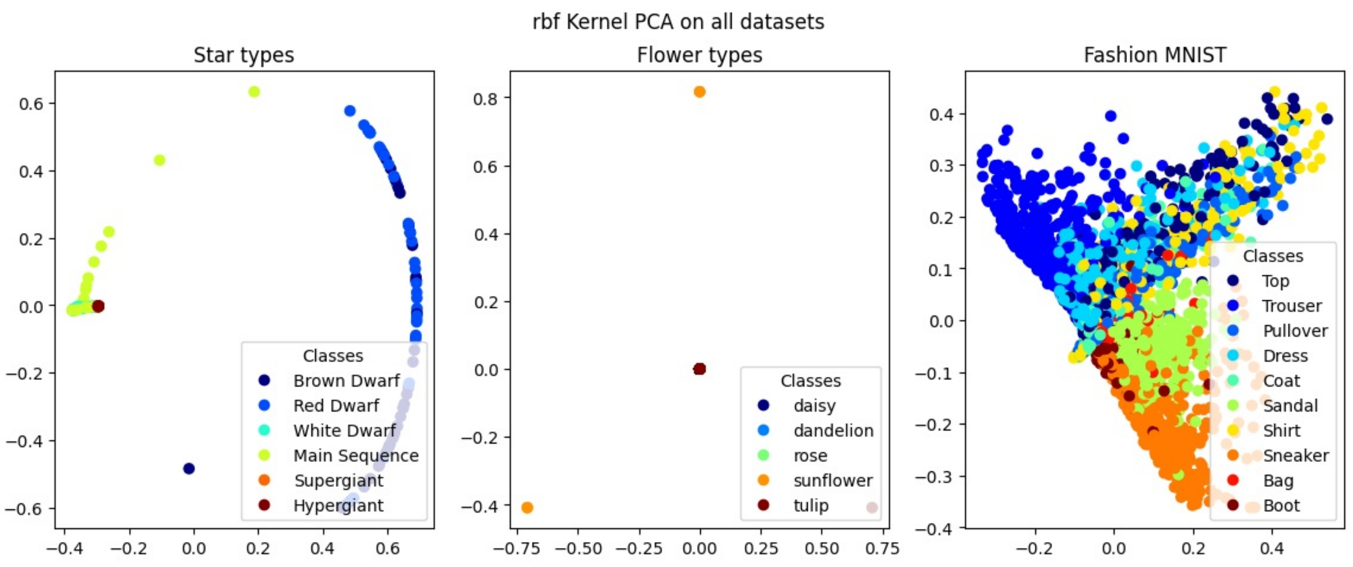
- **Linear** (default, same as PCA)
- **Radial Basis Function (RBF)** - represented as feature vector in some input space and defined as the squared Euclidean distance between the two feature vectors. It has a γ parameter, which determines the influence of individual training samples.
- **Polynomial** - it operates by mapping the data into a higher-dimentional space using a polynomial kernel function of a specified degree, allowing for the identification of non-linear patterns and relationships within the data. Similarly to RBF, it also has a γ parameter that affects the kernel coefficient.
- **Sigmoid** - sigmoid kernel function is defined as $K(x, y) = tanh(\gamma x^T + c)$. The kernel parameter γ affects the shape and apread of the kernel function. It heavily influences the complexity of the transformation performed by the sigmoid kernel.
- **Cosine** - based on the cosine similarity measure. The cosine kernel function is $K(\mathbf{x}, \mathbf{y}) = \frac{\mathbf{x} \cdot \mathbf{y}}{||\mathbf{x}|| \cdot ||\mathbf{y}||}$, where $\mathbf{x}, \mathbf{y}$ are input vectors. The cosine kernel is used, when the angle between data points is more relevant than the distance between them. Thus it is often used in natural language processing tasks. In image processing it does not see much use, which is demonstated with my considered datasets - the use of cosine kernel does not improve the data representation.

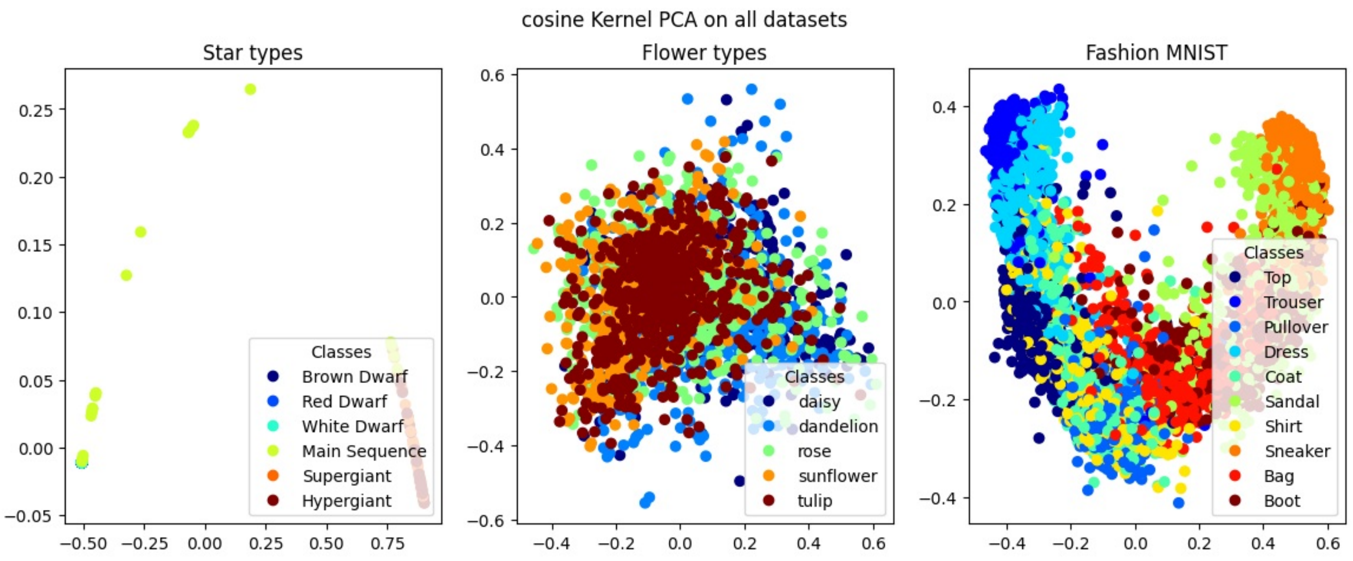
In [ ]:
```python
# Function to create and plot PCA with different kernels
# Digits dataset is skipped as it exceeds memory

def kernel_plot(kernel):

    # initialize the kernel PCA
    kernel_pca = KernelPCA(
        n_components=None, kernel=kernel,
        gamma=1E-6,  # Kernel coefficient for rbf, poly and sigmoid kernels. Ignored by other kernels (default:
        fit_inverse_transform=True, alpha=0.1,
        degree=2,    # for 'poly'-kernel - indicates a square polynomial
        random_state=RANDOM_STATE
        )

    # fit all data
    star_kernel_pca = kernel_pca.fit_transform(star_train)
    flower_kernel_pca = kernel_pca.fit_transform(flower_train)
    fashion_kernel_pca = kernel_pca.fit_transform(fashion_train)
    #digits_kernel_pca = kernel_pca.fit_transform(digits_train)  # skipped (out-of-memory)

    # plot the results
    plot_comparison(
        reduced_data=[star_kernel_pca, flower_kernel_pca, fashion_kernel_pca],
        num_plots=3,
        targets=[star_target, flower_labels, fashion_labels],
        title = f'{kernel} Kernel PCA on all datasets',
        subtitles=['Star types', 'Flower types', 'Fashion MNIST'],
        label_dict=[star_dict,flower_dict, fashion_dict]
        )
```

In [ ]:
```python
# initialize the Kernel PCA for all mentioned kernels
kernels = ['poly','rbf','cosine']
```

In [ ]:
```python
# Plot KernelPCA reduction with different kernels and different datasets for comparison
for kernel in kernels:
    kernel_plot(kernel)
```







Most curious is the RBF Kernel PCA: it was able to separate boot-like objects from various tops in the Fashion MNIST dataset and dwarf stars from the rest in the Star Types dataset.

Meanwhile, Cosine Kernel RBF had little effect on image data, it was capable of separating giant stars from the rest.

In [ ]:
```
# -----skipped (out-of-memory)--------
#digits_kernel_pca = kernel_pca.fit_transform(digits_train)
```

The Kernel PCA does not work on my machine for the Digits dataset and crashes my kernel, as the data is too large to fit in the memory of my computer. As a workaround - introduce the Incremental PCA

Incremental PCA

When the dataset to be decomposed is too large to fit in memory with PCA, a replacement is typically used - Incremental Principal Component Analysis (IPCA), which, as the name suggests, splits the data into a given number of batches and incremenatlly performs partial fit on each of the batches. IPCA builds a low-rank approximation for the input data using an amount of memory which is independent of the number of input data samples. It is still dependent on the input data features, but changing the batch size allows for control of memory usage.

Notably, this yields stochastic results, thus may lead to some imprecisions over time. It is still dependent on the input data features, but changing the batch size allows for control of memory usage.

Depending on the size of the input data, this algorithm can be much more memory efficient than a PCA, and allows sparse input.

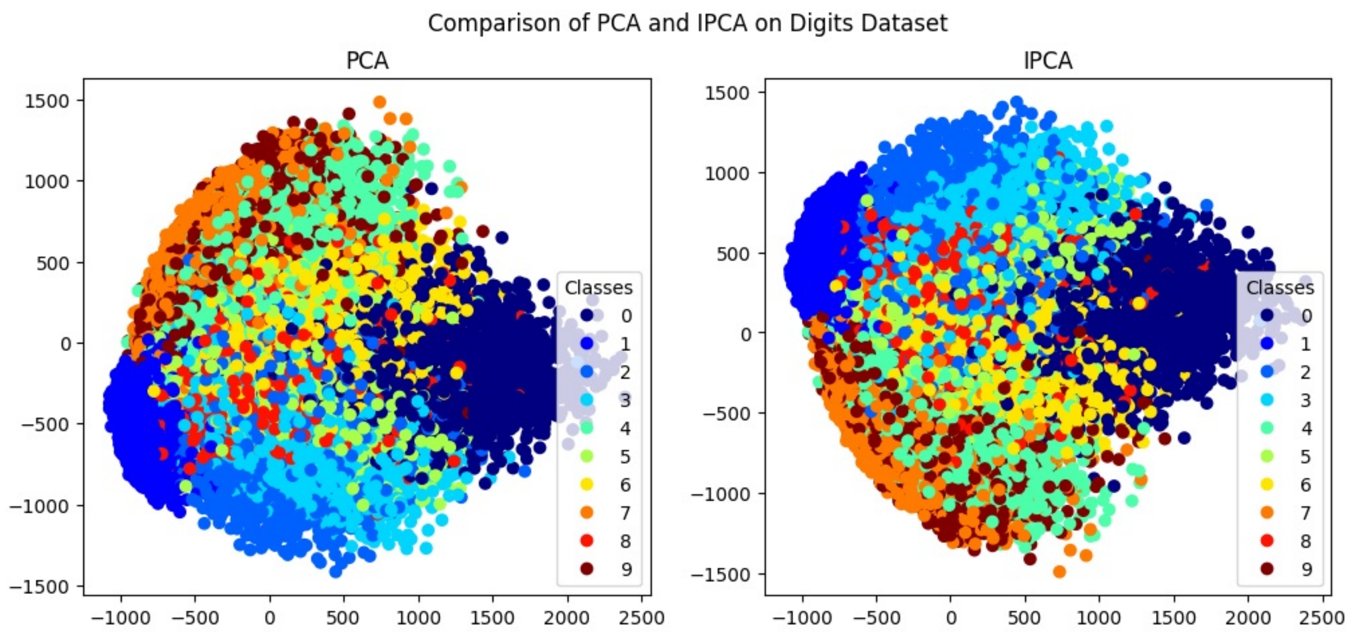
In [ ]:
```
batch_size = 5
ipca = IncrementalPCA(n_components=n_components, whiten=False)
```

This algorithm has constant memory complexity, on the order of `batch_size * n_features`, enabling use of np.memmap files without loading the entire file into memory. For sparse matrices, the input is converted to dense in batches (in order to be able to subtract the mean) which avoids storing the entire dense matrix at any one time.

Source: scikit documentation

```
In [ ]: digits_ipca = ipca.fit_transform(digits_train)
```

```
In [ ]: # IPCA does the same as PCA
        plot_comparison(
            reduced_data=[digits_pca, digits_ipca],
            num_plots=2,
            targets=[digits_target, digits_target],
            title = 'Comparison of PCA and IPCA on Digits Dataset',
            subtitles=['PCA', 'IPCA'],
            label_dict=[digits_dict, digits_dict]
            )
```



Curiosly, the results above are vertically mirrowed. Since both PCA and IPCA perform the same operations for finding principal components, this happened possible because after centering the data the positive or negative direction of the second PC does not relate to the variance of the data, making both options plausible.

IPCA on large data

Primary usage of IPCA is to replace PCA when the dataset to be decomposed is too large to fit in memory.

Original sized (60000 images) Fashion MNIST dataset is used here to demonstrate this feature.

```
In [ ]: %%timeit
        fashion_pca_large = pca.fit_transform(fashion_train_large)
```

```
1.2 s ± 32.1 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

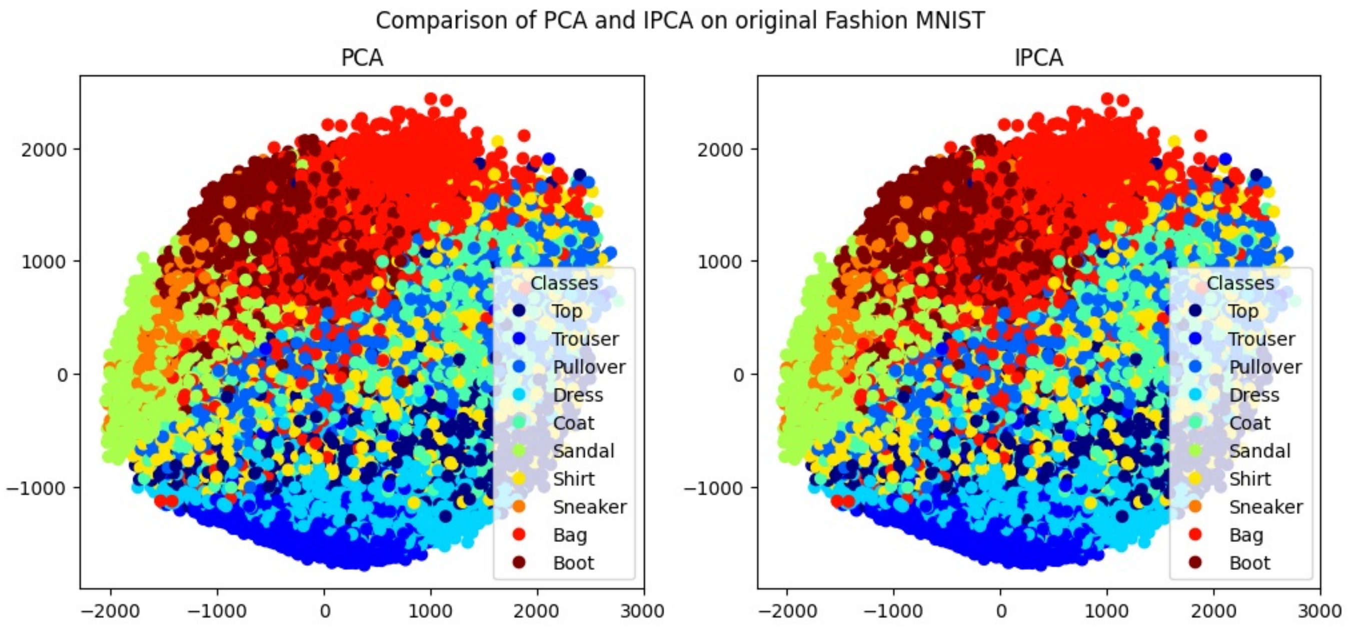
While IPCA allows a machine to fit the data efficiently in terms of memory, however the trade-off is execution time

```
In [ ]: %%timeit
        fashion_ipca_large = ipca.fit_transform(fashion_train_large)
```

```
8.71 s ± 131 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
In [ ]: fashion_pca_large = pca.fit_transform(fashion_train_large)
        fashion_ipca_large = ipca.fit_transform(fashion_train_large)
```

```
In [ ]: # IPCA does the same as PCA
        plot_comparison(
            reduced_data=[fashion_pca_large, fashion_ipca_large],
            num_plots=2,
            targets=[fashion_labels_large, fashion_labels_large],
            title = 'Comparison of PCA and IPCA on original Fashion MNIST',
            subtitles=['PCA', 'IPCA'],
            label_dict=[fashion_dict, fashion_dict]
            )
```



The results shown are visibly the same!

Manifold Learning

Manifold learning is an approach to non-linear dimensionality reduction. Algorithms for this task are based on the idea that the dimensionality of many data sets is only artificially high.

While data in two or three dimensions can be plotted to show the inherent structure of the data, equivalent high-dimensional plots are much less intuitive. To aid visualization of the structure of a dataset, the dimension must be reduced.

To address this concern, a number of supervised and unsupervised linear dimensionality reduction frameworks have been designed, such as Principal Component Analysis (PCA), Independent Component Analysis, Linear Discriminant Analysis, and others. These algorithms define specific rubrics to choose an “interesting” linear projection of the data. These methods can be powerful, but often miss important non-linear structure in the data.

Source: scikit documentation

```
In [ ]: from sklearn.manifold import TSNE, Isomap, MDS
```

T-SNE - T-distributed Stohastic Neighbor Embedding

T-distributed Stohastic Neighbor Embedding (T-SNE) is a technique to visualize high-dimensional data by giving each data point a location in a $2-$ or $3-D$ map (in this project, only $2-D$ representation is considered for all dimentionality reduction techniques). It converts similarities between data points to joint probabilities and tries to minimize the Kullback-Leibler divergence (a measure of how one probability distribution is different from a second) between the joint probabilities of the low-dimensional embedding and the high-dimensional data.

T-SNE is particularly effective at revealing structure at many different scales, making it suitable for high-dimensional data that lie on different, but related, low-dimentional manifolds. Note, that it reduced tendency to crowd points together in the center of the map, which is the most common tendency in all considered methods so far.

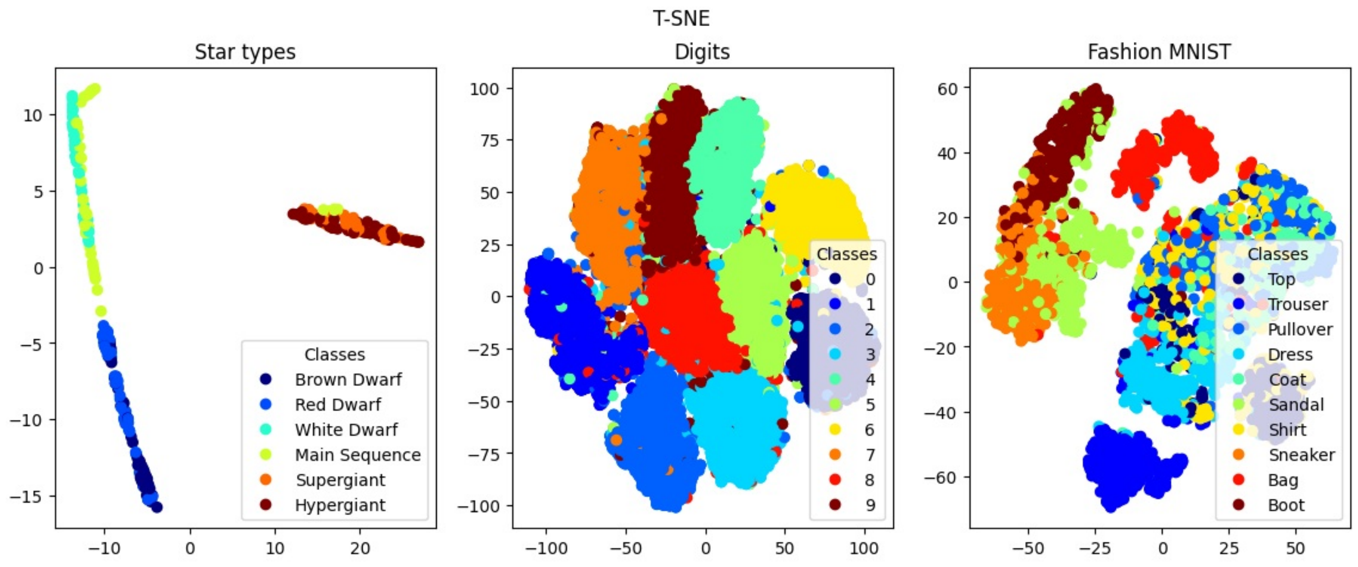
T-SNE has a cost function that is not convex, i.e. with different initializations we can get different results.

Source: scikit Documentation

```
In [ ]: t_sne = TSNE(
            n_components=n_components,
            perplexity=30,
            early_exaggeration=10,
            n_iter=1000,
            random_state=RANDOM_STATE,
        )
```

```
In [ ]: star_tsne = t_sne.fit_transform(star_train)
        flower_tsne = t_sne.fit_transform(flower_train)
        fashion_tsne = t_sne.fit_transform(fashion_train)
        digits_tsne = t_sne.fit_transform(digits_train)
```

In [ ]:
```
plot_comparison(
    reduced_data=[star_tsne, digits_tsne, fashion_tsne],
    num_plots=3,
    targets=[star_target, digits_target, fashion_labels],
    title = 'T-SNE',
    subtitles=['Star types', 'Digits', 'Fashion MNIST'],
    label_dict=[star_dict, digits_dict, fashion_dict]
    )
```



T-SNE performed very well on both image datasets and the Star Types dataset, separating most classes into clusters with negligible errors.

Note, the rightmost cluster in Fashion MNIST represents the top clothing (T-shirts, shirts, pullovers etc.), which have not been separated. Possibly, this is caused but faulty data, as some images cannot be distingushed even by a human.

In this example images of T-shirt and Shirt are hardly distingushable.

Isomap

Isometric Mapping (Isomap) seeks a lower-dimensional embedding which maintains geodesic distances between all points. It determines the neighbors of each point, constructing a neighboring graph, finding the shortest path between nodes, then computing a lower-dimentional embedding.

Isomap can be viewed as an extension of Multi-dimensional Scaling (MDS) or Kernel PCA. Isomap seeks a lower-dimensional embedding which maintains geodesic distances between all points.

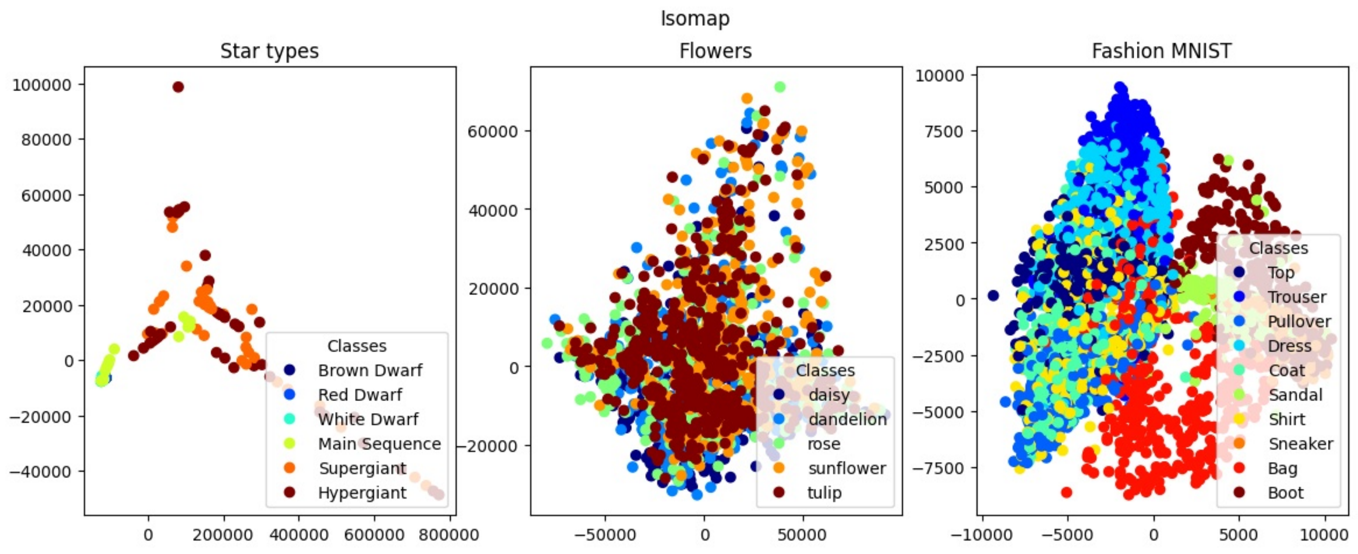
Source: scikit documentation

In [ ]:
```
n_neighbors = 5
p = 2  # Parameter for the Minkowski metric
isomap = Isomap(n_neighbors=n_neighbors, n_components=n_components, p=2)
```

In [ ]:
```
flower_isomap = isomap.fit_transform(flower_train)
fashion_isomap = isomap.fit_transform(fashion_train)
star_isomap = isomap.fit_transform(star_train)
```

```
c:\Users\kovan\miniconda3\Lib\site-packages\sklearn\manifold\_isomap.py:383: UserWarning: The number of connecte
d components of the neighbors graph is 4 > 1. Completing the graph to fit Isomap might be slow. Increase the num
ber of neighbors to avoid this issue.
  self._fit_transform(X)
c:\Users\kovan\miniconda3\Lib\site-packages\scipy\sparse\_index.py:100: SparseEfficiencyWarning: Changing the sp
arsity structure of a csr_matrix is expensive. lil_matrix is more efficient.
  self._set_intXint(row, col, x.flat[0])
c:\Users\kovan\miniconda3\Lib\site-packages\scipy\sparse\_index.py:100: SparseEfficiencyWarning: Changing the sp
arsity structure of a csr_matrix is expensive. lil_matrix is more efficient.
  self._set_intXint(row, col, x.flat[0])
c:\Users\kovan\miniconda3\Lib\site-packages\scipy\sparse\_index.py:100: SparseEfficiencyWarning: Changing the sp
arsity structure of a csr_matrix is expensive. lil_matrix is more efficient.
  self._set_intXint(row, col, x.flat[0])
c:\Users\kovan\miniconda3\Lib\site-packages\scipy\sparse\_index.py:100: SparseEfficiencyWarning: Changing the sp
arsity structure of a csr_matrix is expensive. lil_matrix is more efficient.
  self._set_intXint(row, col, x.flat[0])
c:\Users\kovan\miniconda3\Lib\site-packages\scipy\sparse\_index.py:100: SparseEfficiencyWarning: Changing the sp
arsity structure of a csr_matrix is expensive. lil_matrix is more efficient.
  self._set_intXint(row, col, x.flat[0])
c:\Users\kovan\miniconda3\Lib\site-packages\scipy\sparse\_index.py:100: SparseEfficiencyWarning: Changing the sp
arsity structure of a csr_matrix is expensive. lil_matrix is more efficient.
  self._set_intXint(row, col, x.flat[0])
```

In [ ]:
```python
plot_comparison(
    reduced_data=[star_isomap, flower_isomap, fashion_isomap],
    num_plots=3,
    targets=[star_target, flower_labels, fashion_labels],
    title = 'Isomap',
    subtitles=['Star types', 'Flowers', 'Fashion MNIST'],
    label_dict=[star_dict,flower_dict, fashion_dict]
    )
```



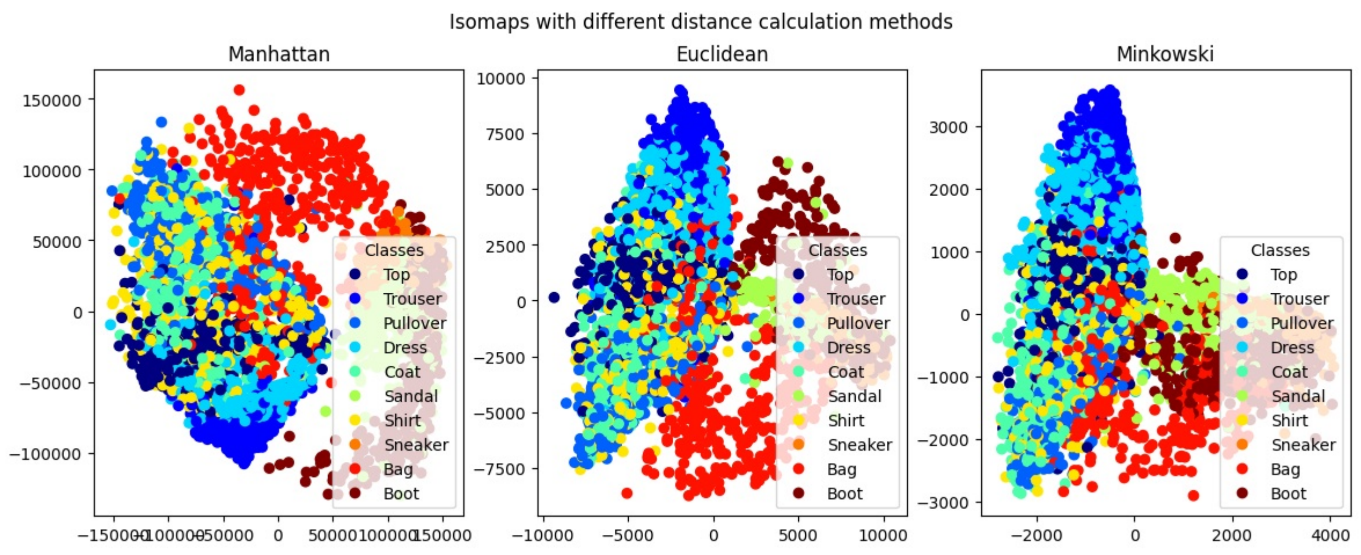
Isomap supports different methods of calculating distances (pairwise), namely:

- $p = 1$ -> manhattan_distance;
- $p = 2$ -> euclidean_distance;
- otherwise -> minkowski_distance.

Below is their comparison (based on Fashion MNIST dataset)

In [ ]:
```python
isomap_comparison = []  # list to store isomaps with different parameters
for param in range(3):
    isomap = Isomap(n_neighbors=n_neighbors, n_components=n_components, p=param+1)
    isomap_comparison.append(isomap.fit_transform(fashion_train))
```

In [ ]:
```python
plot_comparison(
    reduced_data=isomap_comparison,
    num_plots=3,
    targets=[fashion_labels, fashion_labels, fashion_labels],
    title = 'Isomaps with different distance calculation methods',
    subtitles=['Manhattan', 'Euclidean', 'Minkowski'],
    label_dict=[fashion_dict,fashion_dict, fashion_dict]
    )
```



UMAP - Uniform Manifold Approximation and Projection

Performance

UMAP is a general purpose manifold learning and dimension reduction algorithm. Similar to PCA and t-SNE, UMAP is a technique for visualizing highly dimensional data. It reduces a dataset down to a specified number of components. In UMAP's case, it uses manifold learning - a technique that works especially well when there are non-linear relationships that exist within the data.

While UMAP yields similar visual results to T-SNE, it has superior time performance.

Source: UMAP documentation

In [ ]:
```
%pip install umap-learn
```

```
Requirement already satisfied: umap-learn in c:\users\kovan\miniconda3\lib\site-packages (0.5.5)
Requirement already satisfied: numpy>=1.17 in c:\users\kovan\miniconda3\lib\site-packages (from umap-learn) (1.2
6.2)
Requirement already satisfied: scipy>=1.3.1 in c:\users\kovan\miniconda3\lib\site-packages (from umap-learn) (1.
11.4)
Requirement already satisfied: scikit-learn>=0.22 in c:\users\kovan\miniconda3\lib\site-packages (from umap-lear
n) (1.3.2)
Requirement already satisfied: numba>=0.51.2 in c:\users\kovan\miniconda3\lib\site-packages (from umap-learn) (0
.58.1)
Requirement already satisfied: pynndescent>=0.5 in c:\users\kovan\miniconda3\lib\site-packages (from umap-learn)
(0.5.11)
Requirement already satisfied: tqdm in c:\users\kovan\miniconda3\lib\site-packages (from umap-learn) (4.65.0)
Requirement already satisfied: llvmlite<0.42,>=0.41.0dev0 in c:\users\kovan\miniconda3\lib\site-packages (from n
umba>=0.51.2->umap-learn) (0.41.1)
Requirement already satisfied: joblib>=0.11 in c:\users\kovan\miniconda3\lib\site-packages (from pynndescent>=0.
5->umap-learn) (1.3.2)
Requirement already satisfied: threadpoolctl>=2.0.0 in c:\users\kovan\miniconda3\lib\site-packages (from scikit-
learn>=0.22->umap-learn) (3.2.0)
Requirement already satisfied: colorama in c:\users\kovan\miniconda3\lib\site-packages (from tqdm->umap-learn) (
0.4.6)
Note: you may need to restart the kernel to use updated packages.
```

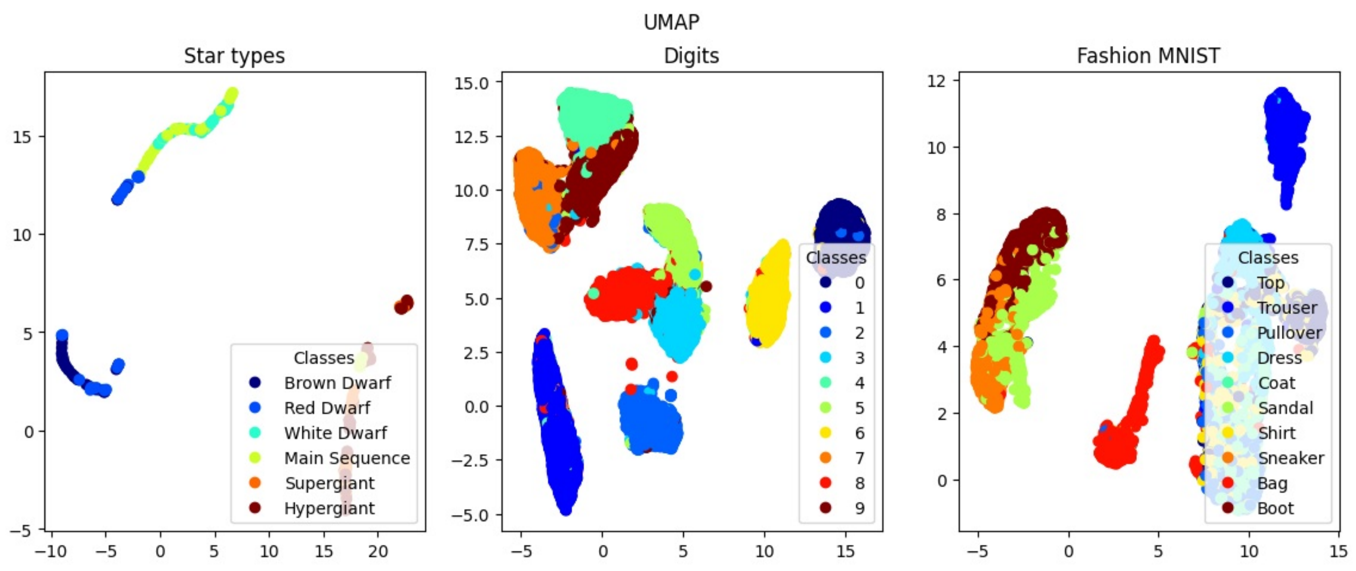
In [ ]:
```
import umap
```

```
c:\Users\kovan\miniconda3\Lib\site-packages\tqdm\auto.py:21: TqdmWarning: IProgress not found. Please update jup
yter and ipywidgets. See https://ipywidgets.readthedocs.io/en/stable/user_install.html
  from .autonotebook import tqdm as notebook_tqdm
```

In [ ]:
```
reducer = umap.UMAP()
```

In [ ]:
```
fashion_umap = reducer.fit_transform(fashion_train)
star_umap = reducer.fit_transform(star_train)
digits_umap = reducer.fit_transform(digits_train)
```

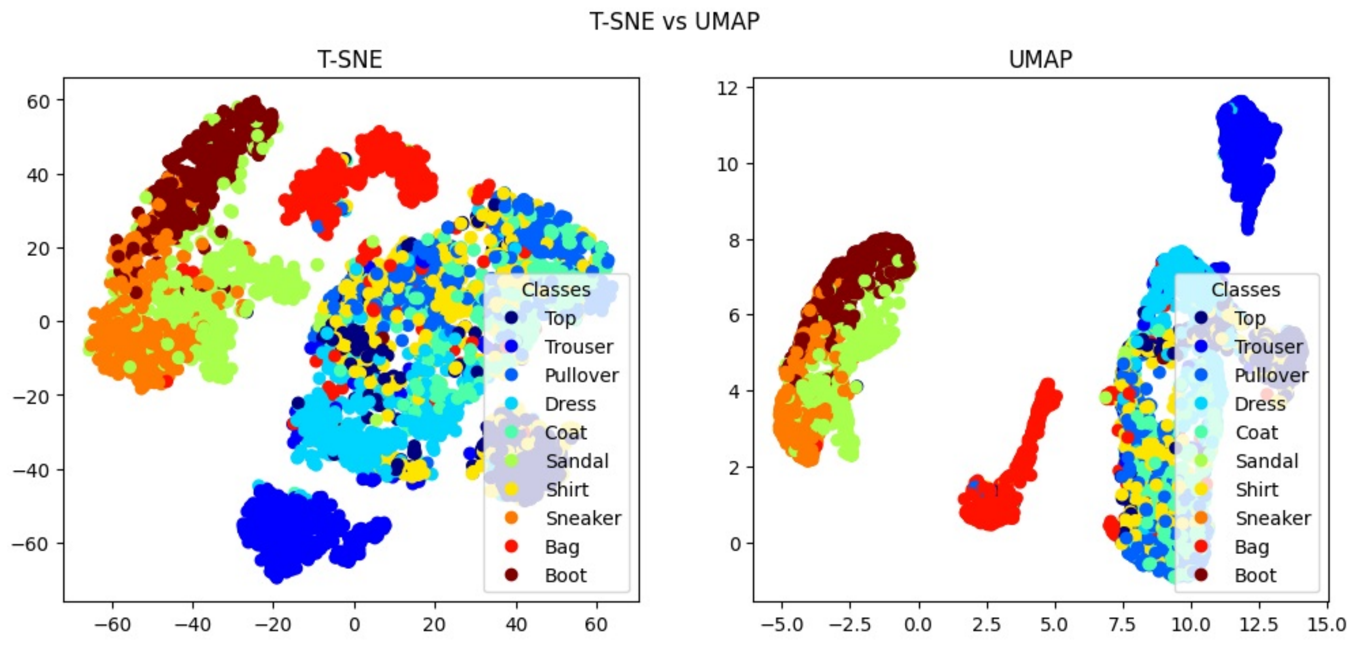
In [ ]:
```
plot_comparison(
    reduced_data=[star_umap, digits_umap, fashion_umap],
    num_plots=3,
    targets=[star_target, digits_target, fashion_labels],
    title = 'UMAP',
    subtitles=['Star types', 'Digits', 'Fashion MNIST'],
    label_dict=[star_dict, digits_dict, fashion_dict]
    )
```



T-SNE vs UMAP

Like mentioned before, UMAP performs operations similarly to T-SNE, which is most visible on Fashion MNIST

```
In [ ]: plot_comparison(
            reduced_data=[fashion_tsne, fashion_umap],
            num_plots=2,
            targets=[fashion_labels, fashion_labels],
            title = 'T-SNE vs UMAP',
            subtitles=['T-SNE', 'UMAP'],
            label_dict=[fashion_dict, fashion_dict]
            )
```



Conclusion

Dimensionality reduction methods play a crucial role in extracting meaningful insights from high-dimensional data by transforming it into a lower-dimensional space while preserving essential characteristics.

Techniques such as Principal Component Analysis (PCA), t-Distributed Stochastic Neighbor Embedding (t-SNE), Isomap, and UMAP offer diverse approaches to this challenge, each with its unique strengths and applications. PCA provides a linear transformation that captures the most significant variance, while t-SNE focuses on preserving local and global structures for effective visualization. Isomap and UMAP, on the other hand, address the nonlinear nature of data, offering robust solutions for manifold learning and dimension reduction.

These methods are instrumental in various domains, including machine learning, data analysis, and visualization, empowering researchers and practitioners to make sense of complex, high-dimensional datasets, therefore being familiar with them is vital in data analysis.