pca_kernelpca_lab

March 24, 2024

1 Principal Component Analysis

Principal Component Analysis (PCA) is a linear dimensionality reduction technique that can be utilized for extracting information from a high-dimensional space by projecting it into a lower-dimensional sub-space. It tries to preserve the essential parts that have more variation of the data and remove the non-essential parts with fewer variation.

Dimensions are nothing but features that represent the data. For example, A 28 X 28 image has 784 picture elements (pixels) that are the dimensions or features which together represent that image.

One important thing to note about PCA is that it is an Unsupervised dimensionality reduction technique, you can cluster the similar data points based on the feature correlation between them without any supervision (or labels), and you will learn how to achieve this practically using Python in later sections of this tutorial!

PCA is a statistical procedure that uses an orthogonal transformation to convert a set of observations of possibly correlated variables (entities each of which takes on various numerical values) into a set of values of linearly uncorrelated variables called principal components. Features, Dimensions, and Variables are all referring to the same thing in this notebook.

Main usage of PCA

- Data Visualization When working on any data related problem, extensive data exploration like finding out how the variables are correlated or understanding the distribution of a few variables is crucial. Considering that there is a large number of variables or dimensions along which the data is distributed, visualization can be a challenge and almost impossible. Using dimensionality reduction, data can be projected into a lower dimension, thereby allowing you to visualize the data in a 2D or 3D space.

- Speeding Machine Learning Algorithm Since PCA's main idea is dimensionality reduction, you can leverage that to speed up your machine learning algorithm's training and testing time considering your data has a lot of features, and the ML algorithm's learning is too slow.

Principal Component Principal components are the key to PCA; they represent what's underneath the hood of your data. In a layman term, when the data is projected into a lower dimension (assume three dimensions) from a higher space, the three dimensions are nothing but the three Principal Components that captures (or holds) most of the variance (information) of your data.

Principal components have both direction and magnitude. The direction represents across which principal axes the data is mostly spread out or has most variance and the magnitude signifies the amount of variance that Principal Component captures of the data when projected onto that axis. The principal components are a straight line, and the first principal component holds the most variance in the data. Each subsequent principal component is orthogonal to the last and has a lesser variance. In this way, given a set of x correlated variables over y samples you achieve a set of u uncorrelated principal components over the same y samples.

The reason you achieve uncorrelated principal components from the original features is that the correlated features contribute to the same principal component, thereby reducing the original data features into uncorrelated principal components; each representing a different set of correlated features with different amounts of variation.

Each principal component represents a percentage of total variation captured from the data.

PCA on iris dataset In this section we will decompose with PCA very simple 4-dimensional data set. This is ono eg the best known pattern recognition dataset. The data set contains 3 classes of 50 instances each, where each class refers to a type of iris plant. One class is linearly separable from the other 2; the latter are NOT linearly separable from each other.

[ ]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LinearRegression
from sklearn.feature_selection import RFE

%matplotlib inline
```

[ ]:
```
%%javascript
IPython.OutputArea.prototype._should_scroll = function(lines) {
    return false;
}
```

```
<IPython.core.display.Javascript object>
```

[ ]:
```
iris_url = "https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.
 ↪data"
```

[ ]:
```
# loading dataset into Pandas DataFrame
df_iris = pd.read_csv(iris_url, names=['sepal length', 'sepal width', 'petal␣
 ↪length', 'petal width', 'target'])
```

[ ]:
```
df_iris.head(15)
```

[ ]:

	sepal length	sepal width	petal length	petal width	target
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa

3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa
5	5.4	3.9	1.7	0.4	Iris-setosa
6	4.6	3.4	1.4	0.3	Iris-setosa
7	5.0	3.4	1.5	0.2	Iris-setosa
8	4.4	2.9	1.4	0.2	Iris-setosa
9	4.9	3.1	1.5	0.1	Iris-setosa
10	5.4	3.7	1.5	0.2	Iris-setosa
11	4.8	3.4	1.6	0.2	Iris-setosa
12	4.8	3.0	1.4	0.1	Iris-setosa
13	4.3	3.0	1.1	0.1	Iris-setosa
14	5.8	4.0	1.2	0.2	Iris-setosa

In the case that the dimensionality of the data allows it, it is good practice to see how each pair of features correlate with each other. In the followinglink you will find more methods for visualizing multidimensional data using matplotlib and seaborn libraries https://towardsdatascience.com/the-art-of-effective-visualization-of-multi-dimensional-data-6c7202990c57

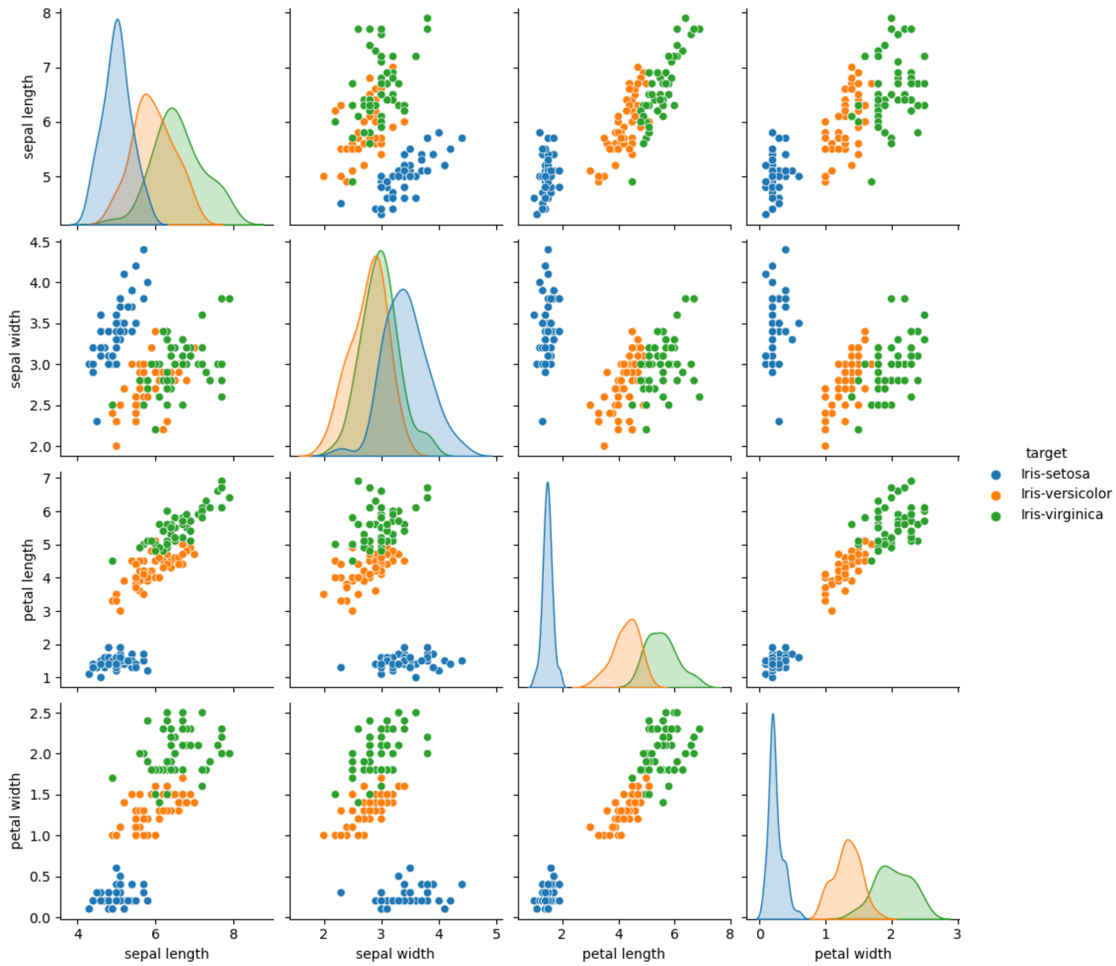
[ ]:
```
sns.pairplot(df_iris, hue='target')
```

```
c:\ProgramData\anaconda3\envs\tensorflow\lib\site-
packages\seaborn\_oldcore.py:1119: FutureWarning: use_inf_as_na option is
deprecated and will be removed in a future version. Convert inf values to NaN
before operating instead.
  with pd.option_context('mode.use_inf_as_na', True):
c:\ProgramData\anaconda3\envs\tensorflow\lib\site-
packages\seaborn\_oldcore.py:1075: FutureWarning: When grouping with a length-1
list-like, you will need to pass a length-1 tuple to get_group in a future
version of pandas. Pass `(name,)` instead of `name` to silence this warning.
  data_subset = grouped_data.get_group(pd_key)
c:\ProgramData\anaconda3\envs\tensorflow\lib\site-
packages\seaborn\_oldcore.py:1075: FutureWarning: When grouping with a length-1
list-like, you will need to pass a length-1 tuple to get_group in a future
version of pandas. Pass `(name,)` instead of `name` to silence this warning.
  data_subset = grouped_data.get_group(pd_key)
c:\ProgramData\anaconda3\envs\tensorflow\lib\site-
packages\seaborn\_oldcore.py:1075: FutureWarning: When grouping with a length-1
list-like, you will need to pass a length-1 tuple to get_group in a future
version of pandas. Pass `(name,)` instead of `name` to silence this warning.
  data_subset = grouped_data.get_group(pd_key)
c:\ProgramData\anaconda3\envs\tensorflow\lib\site-
packages\seaborn\_oldcore.py:1119: FutureWarning: use_inf_as_na option is
deprecated and will be removed in a future version. Convert inf values to NaN
before operating instead.
  with pd.option_context('mode.use_inf_as_na', True):
c:\ProgramData\anaconda3\envs\tensorflow\lib\site-
packages\seaborn\_oldcore.py:1075: FutureWarning: When grouping with a length-1
list-like, you will need to pass a length-1 tuple to get_group in a future
```

```
version of pandas. Pass `(name,)` instead of `name` to silence this warning.
  data_subset = grouped_data.get_group(pd_key)
c:\ProgramData\anaconda3\envs\tensorflow\lib\site-
packages\seaborn\_oldcore.py:1075: FutureWarning: When grouping with a length-1
list-like, you will need to pass a length-1 tuple to get_group in a future
version of pandas. Pass `(name,)` instead of `name` to silence this warning.
  data_subset = grouped_data.get_group(pd_key)
c:\ProgramData\anaconda3\envs\tensorflow\lib\site-
packages\seaborn\_oldcore.py:1075: FutureWarning: When grouping with a length-1
list-like, you will need to pass a length-1 tuple to get_group in a future
version of pandas. Pass `(name,)` instead of `name` to silence this warning.
  data_subset = grouped_data.get_group(pd_key)
c:\ProgramData\anaconda3\envs\tensorflow\lib\site-
packages\seaborn\_oldcore.py:1119: FutureWarning: use_inf_as_na option is
deprecated and will be removed in a future version. Convert inf values to NaN
before operating instead.
  with pd.option_context('mode.use_inf_as_na', True):
c:\ProgramData\anaconda3\envs\tensorflow\lib\site-
packages\seaborn\_oldcore.py:1075: FutureWarning: When grouping with a length-1
list-like, you will need to pass a length-1 tuple to get_group in a future
version of pandas. Pass `(name,)` instead of `name` to silence this warning.
  data_subset = grouped_data.get_group(pd_key)
c:\ProgramData\anaconda3\envs\tensorflow\lib\site-
packages\seaborn\_oldcore.py:1075: FutureWarning: When grouping with a length-1
list-like, you will need to pass a length-1 tuple to get_group in a future
version of pandas. Pass `(name,)` instead of `name` to silence this warning.
  data_subset = grouped_data.get_group(pd_key)
c:\ProgramData\anaconda3\envs\tensorflow\lib\site-
packages\seaborn\_oldcore.py:1075: FutureWarning: When grouping with a length-1
list-like, you will need to pass a length-1 tuple to get_group in a future
version of pandas. Pass `(name,)` instead of `name` to silence this warning.
  data_subset = grouped_data.get_group(pd_key)
c:\ProgramData\anaconda3\envs\tensorflow\lib\site-
packages\seaborn\_oldcore.py:1119: FutureWarning: use_inf_as_na option is
deprecated and will be removed in a future version. Convert inf values to NaN
before operating instead.
  with pd.option_context('mode.use_inf_as_na', True):
c:\ProgramData\anaconda3\envs\tensorflow\lib\site-
packages\seaborn\_oldcore.py:1075: FutureWarning: When grouping with a length-1
list-like, you will need to pass a length-1 tuple to get_group in a future
version of pandas. Pass `(name,)` instead of `name` to silence this warning.
  data_subset = grouped_data.get_group(pd_key)
c:\ProgramData\anaconda3\envs\tensorflow\lib\site-
packages\seaborn\_oldcore.py:1075: FutureWarning: When grouping with a length-1
list-like, you will need to pass a length-1 tuple to get_group in a future
version of pandas. Pass `(name,)` instead of `name` to silence this warning.
  data_subset = grouped_data.get_group(pd_key)
c:\ProgramData\anaconda3\envs\tensorflow\lib\site-
```

```
packages\seaborn\_oldcore.py:1075: FutureWarning: When grouping with a length-1
list-like, you will need to pass a length-1 tuple to get_group in a future
version of pandas. Pass `(name,)` instead of `name` to silence this warning.
  data_subset = grouped_data.get_group(pd_key)
```

```
[ ]: <seaborn.axisgrid.PairGrid at 0x1fe563e1940>
```



You can immediately see that the features petal length and petal width are strongly correlated

1.0.1 Standardize the Data

Since PCA yields a feature subspace that maximizes the variance along the axes, it makes sense to standardize the data, especially, if it was measured on different scales. Although, all features in the Iris dataset were measured in centimeters, let us continue with the transformation of the data onto unit scale (mean=0 and variance=1), which is a requirement for the optimal performance of many machine learning algorithms.

[ ]:
```
features_iris = ['sepal length', 'sepal width', 'petal length', 'petal width']
X_iris = df_iris.loc[:, features_iris].values
```

[ ]:
```
y_iris = df_iris.loc[:, ['target']].values
```

[ ]:
```
X_iris = StandardScaler().fit_transform(X_iris)
```

[ ]:
```
df_iris_standarize = pd.DataFrame(data=X_iris, columns=features_iris)
df_iris_standarize['target'] = df_iris['target']
df_iris_standarize.head(15)
```

[ ]:

	sepal length	sepal width	petal length	petal width	target
0	-0.900681	1.032057	-1.341272	-1.312977	Iris-setosa
1	-1.143017	-0.124958	-1.341272	-1.312977	Iris-setosa
2	-1.385353	0.337848	-1.398138	-1.312977	Iris-setosa
3	-1.506521	0.106445	-1.284407	-1.312977	Iris-setosa
4	-1.021849	1.263460	-1.341272	-1.312977	Iris-setosa
5	-0.537178	1.957669	-1.170675	-1.050031	Iris-setosa
6	-1.506521	0.800654	-1.341272	-1.181504	Iris-setosa
7	-1.021849	0.800654	-1.284407	-1.312977	Iris-setosa
8	-1.748856	-0.356361	-1.341272	-1.312977	Iris-setosa
9	-1.143017	0.106445	-1.284407	-1.444450	Iris-setosa
10	-0.537178	1.494863	-1.284407	-1.312977	Iris-setosa
11	-1.264185	0.800654	-1.227541	-1.312977	Iris-setosa
12	-1.264185	-0.124958	-1.341272	-1.444450	Iris-setosa
13	-1.870024	-0.124958	-1.511870	-1.444450	Iris-setosa
14	-0.052506	2.189072	-1.455004	-1.312977	Iris-setosa

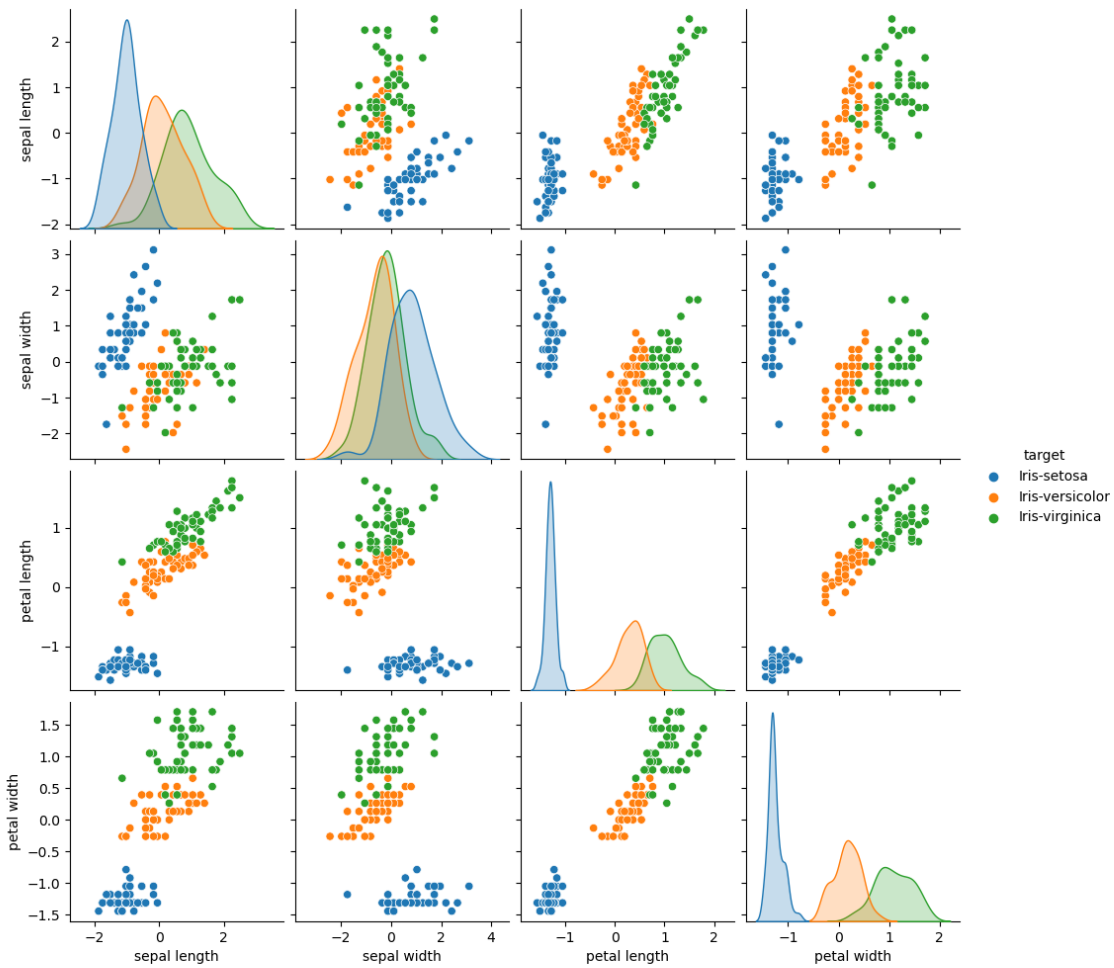
[ ]:
```
sns.pairplot(df_iris_standarize, hue='target')
```

```
c:\ProgramData\anaconda3\envs\tensorflow\lib\site-
packages\seaborn\_oldcore.py:1119: FutureWarning: use_inf_as_na option is
deprecated and will be removed in a future version. Convert inf values to NaN
before operating instead.
  with pd.option_context('mode.use_inf_as_na', True):
c:\ProgramData\anaconda3\envs\tensorflow\lib\site-
packages\seaborn\_oldcore.py:1075: FutureWarning: When grouping with a length-1
list-like, you will need to pass a length-1 tuple to get_group in a future
version of pandas. Pass `(name,)` instead of `name` to silence this warning.
  data_subset = grouped_data.get_group(pd_key)
c:\ProgramData\anaconda3\envs\tensorflow\lib\site-
packages\seaborn\_oldcore.py:1075: FutureWarning: When grouping with a length-1
list-like, you will need to pass a length-1 tuple to get_group in a future
version of pandas. Pass `(name,)` instead of `name` to silence this warning.
  data_subset = grouped_data.get_group(pd_key)
c:\ProgramData\anaconda3\envs\tensorflow\lib\site-
packages\seaborn\_oldcore.py:1075: FutureWarning: When grouping with a length-1
list-like, you will need to pass a length-1 tuple to get_group in a future
```

```
version of pandas. Pass `(name,)` instead of `name` to silence this warning.
  data_subset = grouped_data.get_group(pd_key)
c:\ProgramData\anaconda3\envs\tensorflow\lib\site-
packages\seaborn\_oldcore.py:1119: FutureWarning: use_inf_as_na option is
deprecated and will be removed in a future version. Convert inf values to NaN
before operating instead.
  with pd.option_context('mode.use_inf_as_na', True):
c:\ProgramData\anaconda3\envs\tensorflow\lib\site-
packages\seaborn\_oldcore.py:1075: FutureWarning: When grouping with a length-1
list-like, you will need to pass a length-1 tuple to get_group in a future
version of pandas. Pass `(name,)` instead of `name` to silence this warning.
  data_subset = grouped_data.get_group(pd_key)
c:\ProgramData\anaconda3\envs\tensorflow\lib\site-
packages\seaborn\_oldcore.py:1075: FutureWarning: When grouping with a length-1
list-like, you will need to pass a length-1 tuple to get_group in a future
version of pandas. Pass `(name,)` instead of `name` to silence this warning.
  data_subset = grouped_data.get_group(pd_key)
c:\ProgramData\anaconda3\envs\tensorflow\lib\site-
packages\seaborn\_oldcore.py:1075: FutureWarning: When grouping with a length-1
list-like, you will need to pass a length-1 tuple to get_group in a future
version of pandas. Pass `(name,)` instead of `name` to silence this warning.
  data_subset = grouped_data.get_group(pd_key)
c:\ProgramData\anaconda3\envs\tensorflow\lib\site-
packages\seaborn\_oldcore.py:1119: FutureWarning: use_inf_as_na option is
deprecated and will be removed in a future version. Convert inf values to NaN
before operating instead.
  with pd.option_context('mode.use_inf_as_na', True):
c:\ProgramData\anaconda3\envs\tensorflow\lib\site-
packages\seaborn\_oldcore.py:1075: FutureWarning: When grouping with a length-1
list-like, you will need to pass a length-1 tuple to get_group in a future
version of pandas. Pass `(name,)` instead of `name` to silence this warning.
  data_subset = grouped_data.get_group(pd_key)
c:\ProgramData\anaconda3\envs\tensorflow\lib\site-
packages\seaborn\_oldcore.py:1075: FutureWarning: When grouping with a length-1
list-like, you will need to pass a length-1 tuple to get_group in a future
version of pandas. Pass `(name,)` instead of `name` to silence this warning.
  data_subset = grouped_data.get_group(pd_key)
c:\ProgramData\anaconda3\envs\tensorflow\lib\site-
packages\seaborn\_oldcore.py:1075: FutureWarning: When grouping with a length-1
list-like, you will need to pass a length-1 tuple to get_group in a future
version of pandas. Pass `(name,)` instead of `name` to silence this warning.
  data_subset = grouped_data.get_group(pd_key)
c:\ProgramData\anaconda3\envs\tensorflow\lib\site-
packages\seaborn\_oldcore.py:1119: FutureWarning: use_inf_as_na option is
deprecated and will be removed in a future version. Convert inf values to NaN
before operating instead.
  with pd.option_context('mode.use_inf_as_na', True):
c:\ProgramData\anaconda3\envs\tensorflow\lib\site-
```

```
packages\seaborn\_oldcore.py:1075: FutureWarning: When grouping with a length-1
list-like, you will need to pass a length-1 tuple to get_group in a future
version of pandas. Pass `(name,)` instead of `name` to silence this warning.
  data_subset = grouped_data.get_group(pd_key)
c:\ProgramData\anaconda3\envs\tensorflow\lib\site-
packages\seaborn\_oldcore.py:1075: FutureWarning: When grouping with a length-1
list-like, you will need to pass a length-1 tuple to get_group in a future
version of pandas. Pass `(name,)` instead of `name` to silence this warning.
  data_subset = grouped_data.get_group(pd_key)
c:\ProgramData\anaconda3\envs\tensorflow\lib\site-
packages\seaborn\_oldcore.py:1075: FutureWarning: When grouping with a length-1
list-like, you will need to pass a length-1 tuple to get_group in a future
version of pandas. Pass `(name,)` instead of `name` to silence this warning.
  data_subset = grouped_data.get_group(pd_key)
```

```
[ ]: <seaborn.axisgrid.PairGrid at 0x1fe5fc75df0>
```



We can see that the distributions are now standardized

1.0.2 PCA Projection to 2D

[ ]:
```
pca_iris = PCA(n_components=2)
```

[ ]:
```
principal_components_iris = pca_iris.fit_transform(X_iris)
```

[ ]:
```
principal_df_iris = pd.DataFrame(data=principal_components_iris,␣
↪columns=['principal component 1', 'principal component 2'])
```

[ ]:
```
final_df_iris = pd.concat([principal_df_iris, df_iris[['target']]], axis=1)
final_df_iris.head(15)
```

[ ]:

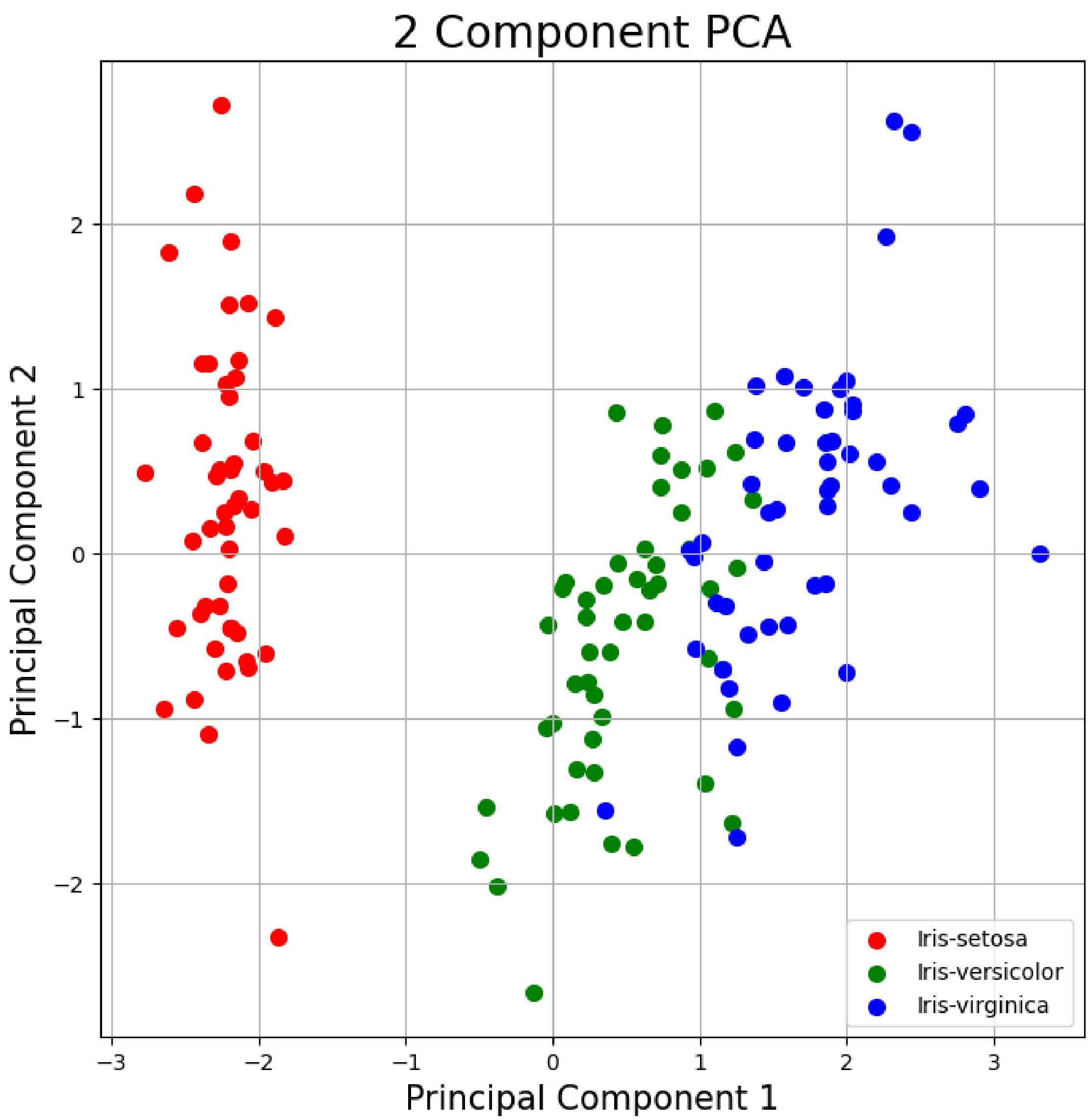
	principal component 1	principal component 2	target
0	-2.264542	0.505704	Iris-setosa
1	-2.086426	-0.655405	Iris-setosa
2	-2.367950	-0.318477	Iris-setosa
3	-2.304197	-0.575368	Iris-setosa
4	-2.388777	0.674767	Iris-setosa
5	-2.070537	1.518549	Iris-setosa
6	-2.445711	0.074563	Iris-setosa
7	-2.233842	0.247614	Iris-setosa
8	-2.341958	-1.095146	Iris-setosa
9	-2.188676	-0.448629	Iris-setosa
10	-2.163487	1.070596	Iris-setosa
11	-2.327378	0.158587	Iris-setosa
12	-2.224083	-0.709118	Iris-setosa
13	-2.639716	-0.938282	Iris-setosa
14	-2.192292	1.889979	Iris-setosa

1.0.3 Visualize 2D Projection

Use a PCA projection to 2d to visualize the entire data set. You should plot different classes using different colors or shapes.

[ ]:
```
fig = plt.figure(figsize=(8, 8))
ax = fig.add_subplot(1, 1, 1)
ax.set_xlabel('Principal Component 1', fontsize=15)
ax.set_ylabel('Principal Component 2', fontsize=15)
ax.set_title('2 Component PCA', fontsize=20)

iris_targets = ['Iris-setosa', 'Iris-versicolor', 'Iris-virginica']
colors = ['r', 'g', 'b']

for target, color in zip(iris_targets, colors):
    indices_to_keep = final_df_iris['target'] == target
    ax.scatter(
        final_df_iris.loc[indices_to_keep, 'principal component 1'],
        final_df_iris.loc[indices_to_keep, 'principal component 2'],
```

```
        c=color,
        s=50
    )

ax.legend(iris_targets)
ax.grid()
```



iris-setosa is linearry separable from others class

1.0.4 Explained Variance

The explained variance tells us how much information (variance) can be attributed to each of the principal components.

[ ]:
```
pca_iris.explained_variance_ratio_
```

[ ]:
```
array([0.72770452, 0.23030523])
```

Together, the first two principal components contain 95.80% of the information. The first principal component contains 72.77% of the variance and the second principal component contains 23.03% of the variance. The third and fourth principal component contained the rest of the variance of the dataset.

1.0.5 limitations of PCA

- PCA is not scale invariant. check: we need to scale our data first.
- The directions with largest variance are assumed to be of the most interest
- Only considers orthogonal transformations (rotations) of the original variables
- PCA is only based on the mean vector and covariance matrix. Some distributions (multivariate normal) are characterized by this, but some are not.
- If the variables are correlated, PCA can achieve dimension reduction. If not, PCA just orders them according to their variances.

1.0.6 Exercises - Perform PCA for breast cancer dataset

- You can find this dataset it in the scikit learn library, import it and convert to pandas dataframe, original label are '0' and '1' for better readability change these names to: 'benign' and 'malignant'

[ ]:
```
import sklearn.datasets
```

[ ]:
```
data = sklearn.datasets.load_breast_cancer()
df_cancer = pd.DataFrame(data.data, columns=data.feature_names)

df_cancer
```

[ ]:
```
     mean radius  mean texture  mean perimeter  mean area  mean smoothness  \
0          17.99         10.38          122.80     1001.0          0.11840
1          20.57         17.77          132.90     1326.0          0.08474
2          19.69         21.25          130.00     1203.0          0.10960
3          11.42         20.38           77.58      386.1          0.14250
4          20.29         14.34          135.10     1297.0          0.10030
..           …             …               …          …                …
564        21.56         22.39          142.00     1479.0          0.11100
565        20.13         28.25          131.20     1261.0          0.09780
566        16.60         28.08          108.30      858.1          0.08455
567        20.60         29.33          140.10     1265.0          0.11780
568         7.76         24.54           47.92      181.0          0.05263

     mean compactness  mean concavity  mean concave points  mean symmetry  \
0             0.27760         0.30010              0.14710         0.2419
```

```
1            0.07864        0.08690             0.07017        0.1812
2            0.15990        0.19740             0.12790        0.2069
3            0.28390        0.24140             0.10520        0.2597
4            0.13280        0.19800             0.10430        0.1809
..               …              …                 …              …
564          0.11590        0.24390             0.13890        0.1726
565          0.10340        0.14400             0.09791        0.1752
566          0.10230        0.09251             0.05302        0.1590
567          0.27700        0.35140             0.15200        0.2397
568          0.04362        0.00000             0.00000        0.1587

     mean fractal dimension  …  worst radius  worst texture  \
0                   0.07871  …        25.380          17.33
1                   0.05667  …        24.990          23.41
2                   0.05999  …        23.570          25.53
3                   0.09744  …        14.910          26.50
4                   0.05883  …        22.540          16.67
..                      …  …           …             …
564                 0.05623  …        25.450          26.40
565                 0.05533  …        23.690          38.25
566                 0.05648  …        18.980          34.12
567                 0.07016  …        25.740          39.42
568                 0.05884  …         9.456          30.37

     worst perimeter  worst area  worst smoothness  worst compactness  \
0             184.60      2019.0           0.16220            0.66560
1             158.80      1956.0           0.12380            0.18660
2             152.50      1709.0           0.14440            0.42450
3              98.87       567.7           0.20980            0.86630
4             152.20      1575.0           0.13740            0.20500
..               …          …               …                 …
564           166.10      2027.0           0.14100            0.21130
565           155.00      1731.0           0.11660            0.19220
566           126.70      1124.0           0.11390            0.30940
567           184.60      1821.0           0.16500            0.86810
568            59.16       268.6           0.08996            0.06444

     worst concavity  worst concave points  worst symmetry  \
0             0.7119                0.2654          0.4601
1             0.2416                0.1860          0.2750
2             0.4504                0.2430          0.3613
3             0.6869                0.2575          0.6638
4             0.4000                0.1625          0.2364
..               …                   …              …
564           0.4107                0.2216          0.2060
565           0.3215                0.1628          0.2572
566           0.3403                0.1418          0.2218
```

```
567           0.9387                 0.2650           0.4087
568           0.0000                 0.0000           0.2871

     worst fractal dimension
0                    0.11890
1                    0.08902
2                    0.08758
3                    0.17300
4                    0.07678
..                       …
564                  0.07115
565                  0.06637
566                  0.07820
567                  0.12400
568                  0.07039

[569 rows x 30 columns]
```

[ ]:
```
df_cancer['target'] = pd.Series(data.target).map({0: 'benign', 1: 'malignant'})

df_cancer.head(15)
```

[ ]:
```
    mean radius  mean texture  mean perimeter  mean area  mean smoothness  \
0         17.99         10.38          122.80     1001.0          0.11840
1         20.57         17.77          132.90     1326.0          0.08474
2         19.69         21.25          130.00     1203.0          0.10960
3         11.42         20.38           77.58      386.1          0.14250
4         20.29         14.34          135.10     1297.0          0.10030
5         12.45         15.70           82.57      477.1          0.12780
6         18.25         19.98          119.60     1040.0          0.09463
7         13.71         20.83           90.20      577.9          0.11890
8         13.00         21.82           87.50      519.8          0.12730
9         12.46         24.04           83.97      475.9          0.11860
10        16.02         23.24          102.70      797.8          0.08206
11        15.78         17.89          103.60      781.0          0.09710
12        19.17         24.80          132.40     1123.0          0.09740
13        15.85         23.95          103.70      782.7          0.08401
14        13.73         22.61           93.60      578.3          0.11310

    mean compactness  mean concavity  mean concave points  mean symmetry  \
0            0.27760         0.30010              0.14710         0.2419
1            0.07864         0.08690              0.07017         0.1812
2            0.15990         0.19740              0.12790         0.2069
3            0.28390         0.24140              0.10520         0.2597
4            0.13280         0.19800              0.10430         0.1809
5            0.17000         0.15780              0.08089         0.2087
6            0.10900         0.11270              0.07400         0.1794
```

7	0.16450	0.09366	0.05985	0.2196
8	0.19320	0.18590	0.09353	0.2350
9	0.23960	0.22730	0.08543	0.2030
10	0.06669	0.03299	0.03323	0.1528
11	0.12920	0.09954	0.06606	0.1842
12	0.24580	0.20650	0.11180	0.2397
13	0.10020	0.09938	0.05364	0.1847
14	0.22930	0.21280	0.08025	0.2069

	mean fractal dimension	…	worst texture	worst perimeter	worst area \
0	0.07871	…	17.33	184.60	2019.0
1	0.05667	…	23.41	158.80	1956.0
2	0.05999	…	25.53	152.50	1709.0
3	0.09744	…	26.50	98.87	567.7
4	0.05883	…	16.67	152.20	1575.0
5	0.07613	…	23.75	103.40	741.6
6	0.05742	…	27.66	153.20	1606.0
7	0.07451	…	28.14	110.60	897.0
8	0.07389	…	30.73	106.20	739.3
9	0.08243	…	40.68	97.65	711.4
10	0.05697	…	33.88	123.80	1150.0
11	0.06082	…	27.28	136.50	1299.0
12	0.07800	…	29.94	151.70	1332.0
13	0.05338	…	27.66	112.00	876.5
14	0.07682	…	32.01	108.80	697.7

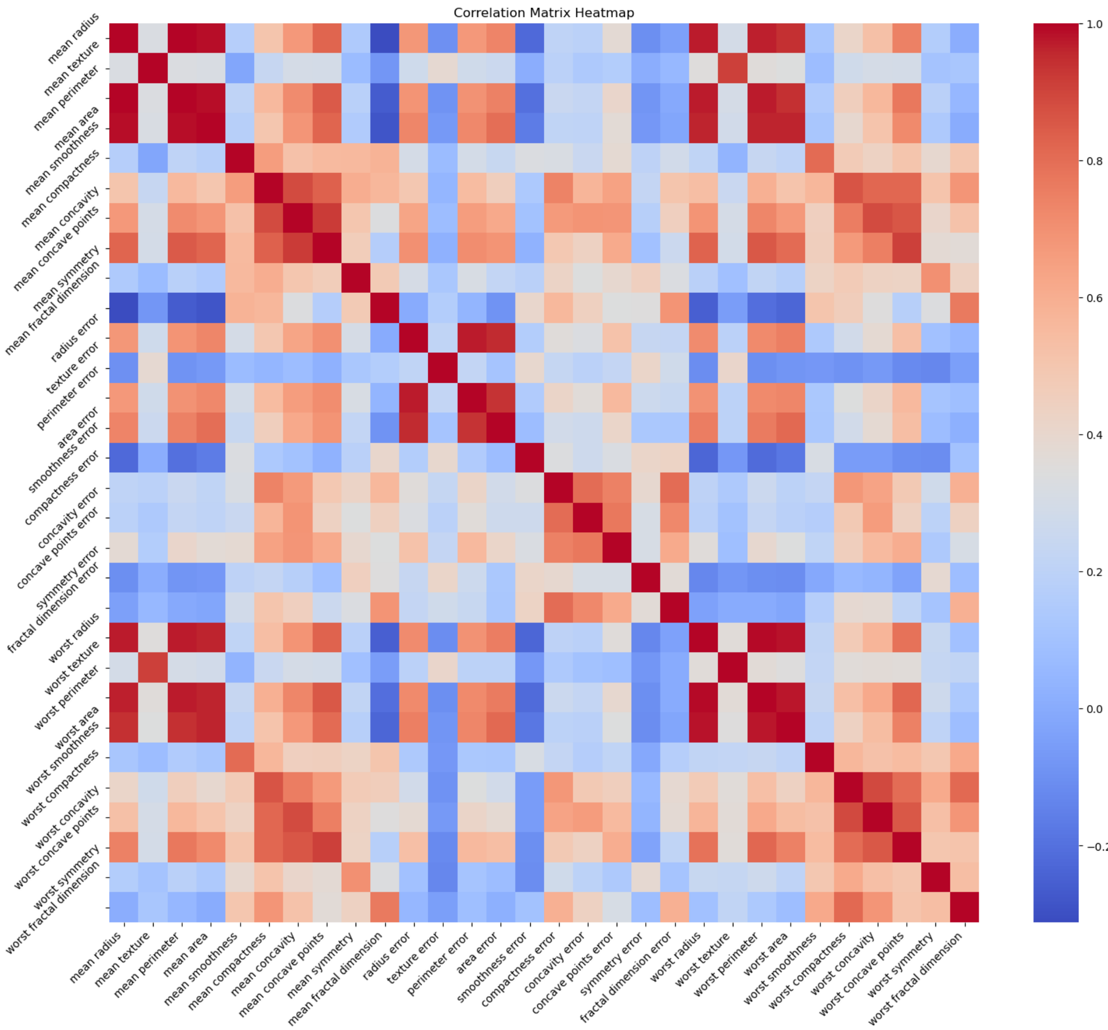
	worst smoothness	worst compactness	worst concavity \
0	0.1622	0.6656	0.7119
1	0.1238	0.1866	0.2416
2	0.1444	0.4245	0.4504
3	0.2098	0.8663	0.6869
4	0.1374	0.2050	0.4000
5	0.1791	0.5249	0.5355
6	0.1442	0.2576	0.3784
7	0.1654	0.3682	0.2678
8	0.1703	0.5401	0.5390
9	0.1853	1.0580	1.1050
10	0.1181	0.1551	0.1459
11	0.1396	0.5609	0.3965
12	0.1037	0.3903	0.3639
13	0.1131	0.1924	0.2322
14	0.1651	0.7725	0.6943

	worst concave points	worst symmetry	worst fractal dimension	target
0	0.26540	0.4601	0.11890	benign
1	0.18600	0.2750	0.08902	benign
2	0.24300	0.3613	0.08758	benign

```
3                  0.25750           0.6638                    0.17300  benign
4                  0.16250           0.2364                    0.07678  benign
5                  0.17410           0.3985                    0.12440  benign
6                  0.19320           0.3063                    0.08368  benign
7                  0.15560           0.3196                    0.11510  benign
8                  0.20600           0.4378                    0.10720  benign
9                  0.22100           0.4366                    0.20750  benign
10                 0.09975           0.2948                    0.08452  benign
11                 0.18100           0.3792                    0.10480  benign
12                 0.17670           0.3176                    0.10230  benign
13                 0.11190           0.2809                    0.06287  benign
14                 0.22080           0.3596                    0.14310  benign

[15 rows x 31 columns]
```

- Visualizes correlations between pairs of features (due to the greater number of features use pandas corr () function instead of pairplot instead of seaborn heatmap ())

[ ]:
```
corr_matrix = df_cancer.drop('target', axis=1).corr()

plt.figure(figsize=(18, 15))

sns.heatmap(corr_matrix, fmt=".2f", cmap='coolwarm', xticklabels=corr_matrix.
 ↪columns, yticklabels=corr_matrix.columns)

plt.xticks(rotation=45, ha="right")
plt.yticks(rotation=45)
plt.title('Correlation Matrix Heatmap')

plt.show()
```



- Perform PCA and visualize the data

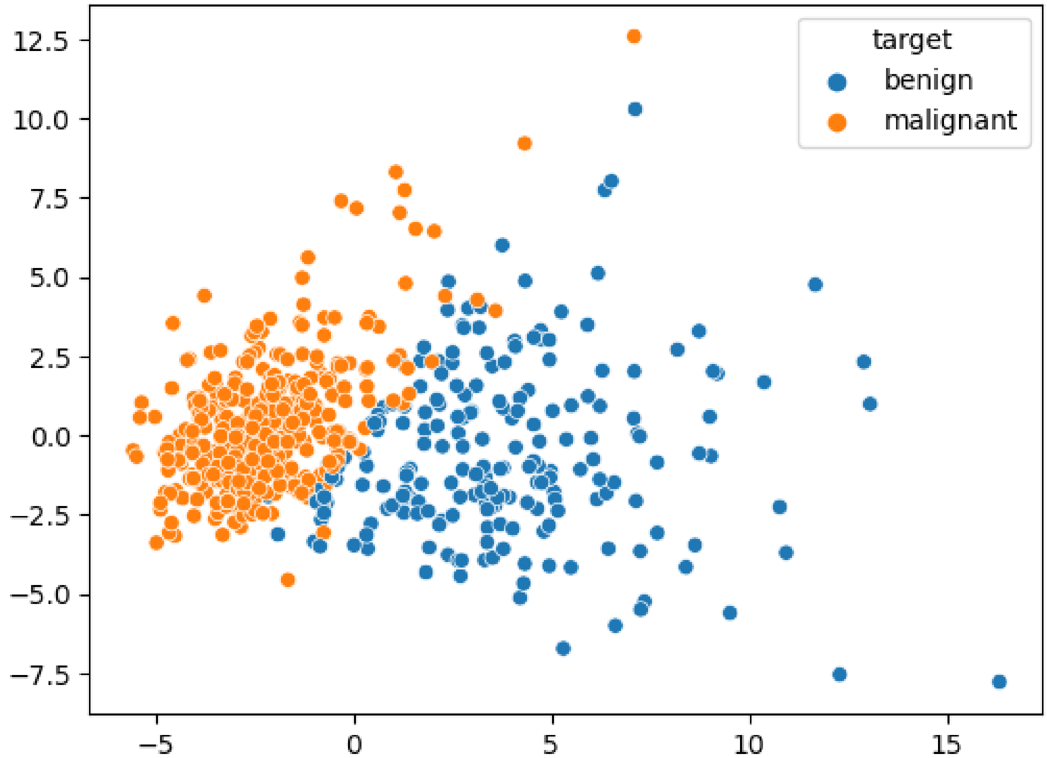
[ ]:
```
X_cancer = df_cancer.drop('target', axis=1)
y_cancer = df_cancer['target']

X_cancer = StandardScaler().fit_transform(X_cancer)
```

[ ]:
```
pca_cancer = PCA()
```

[ ]:
```
principal_components_cancer = pca_cancer.fit_transform(X_cancer)
```

[ ]:
```
sns.scatterplot(x=principal_components_cancer[:, 0],␣
↪y=principal_components_cancer[:, 1], hue=y_cancer)
```

[ ]:
```
<Axes: >
```



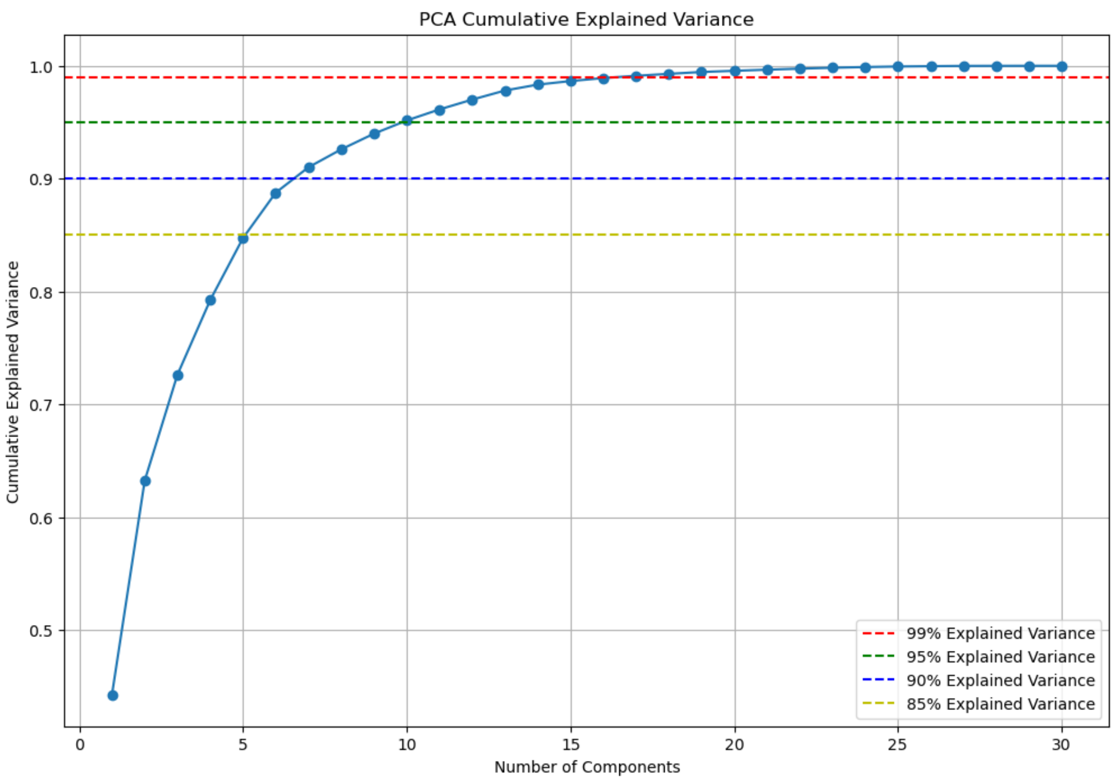

```
[ ]: pca_cancer.explained_variance_ratio_ # Show variance ratio for all components␣
     ↪(the number of components is the same as the number of features)
```

```
[ ]: array([4.42720256e-01, 1.89711820e-01, 9.39316326e-02, 6.60213492e-02,
            5.49576849e-02, 4.02452204e-02, 2.25073371e-02, 1.58872380e-02,
            1.38964937e-02, 1.16897819e-02, 9.79718988e-03, 8.70537901e-03,
            8.04524987e-03, 5.23365745e-03, 3.13783217e-03, 2.66209337e-03,
            1.97996793e-03, 1.75395945e-03, 1.64925306e-03, 1.03864675e-03,
            9.99096464e-04, 9.14646751e-04, 8.11361259e-04, 6.01833567e-04,
            5.16042379e-04, 2.72587995e-04, 2.30015463e-04, 5.29779290e-05,
            2.49601032e-05, 4.43482743e-06])
```

- Examine explained variance, draw a plot showing relation between total explained variance and number of principal components used

```
[ ]: total_explained_variance = pca_cancer.explained_variance_ratio_.cumsum()
     number_of_components = np.arange(1, len(total_explained_variance) + 1)

     plt.figure(figsize=(12, 8))
     plt.plot(number_of_components, total_explained_variance, marker='o',␣
      ↪linestyle='-')
```

```
plt.xlabel('Number of Components')
plt.ylabel('Cumulative Explained Variance')
plt.title('PCA Cumulative Explained Variance')

plt.grid(True)
plt.axhline(y=0.99, color='r', linestyle='--', label='99% Explained Variance')
plt.axhline(y=0.95, color='g', linestyle='--', label='95% Explained Variance')
plt.axhline(y=0.90, color='b', linestyle='--', label='90% Explained Variance')
plt.axhline(y=0.85, color='y', linestyle='--', label='85% Explained Variance')
plt.legend(loc='best')

plt.show()
```



- Use recursive feature elimination (available in scikit-learn module) or another feature ranking algorithm to split 30 features to 15 "more important" and "less important" features. Then repeat the last step from the full data set - draw a plot showing relation between total explained variance and number of principal components used for all 3 cases. Explain the result briefly.

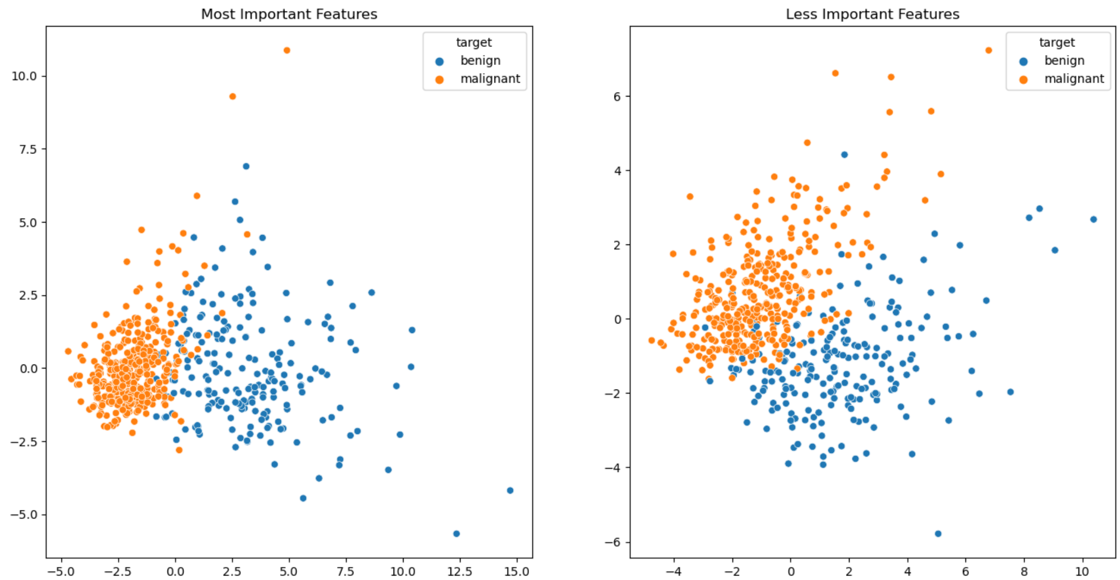
[ ]:
```
from sklearn.preprocessing import LabelEncoder

label_encoder = LabelEncoder()
```

```
y_cancer_encoded = label_encoder.fit_transform(y_cancer)

estimator = LinearRegression()
rfe = RFE(estimator, n_features_to_select=15, step=1)
rfe.fit(X_cancer, y_cancer_encoded)

rfe.support_
```

[ ]:
```
array([ True, False,  True,  True, False,  True,  True,  True, False,
       False,  True, False,  True, False, False, False,  True,  True,
       False, False,  True,  True, False,  True, False, False,  True,
       False, False,  True])
```

[ ]:
```
X_more_important = X_cancer[:,rfe.support_]
X_less_important = X_cancer[:,np.invert(rfe.support_)]
```

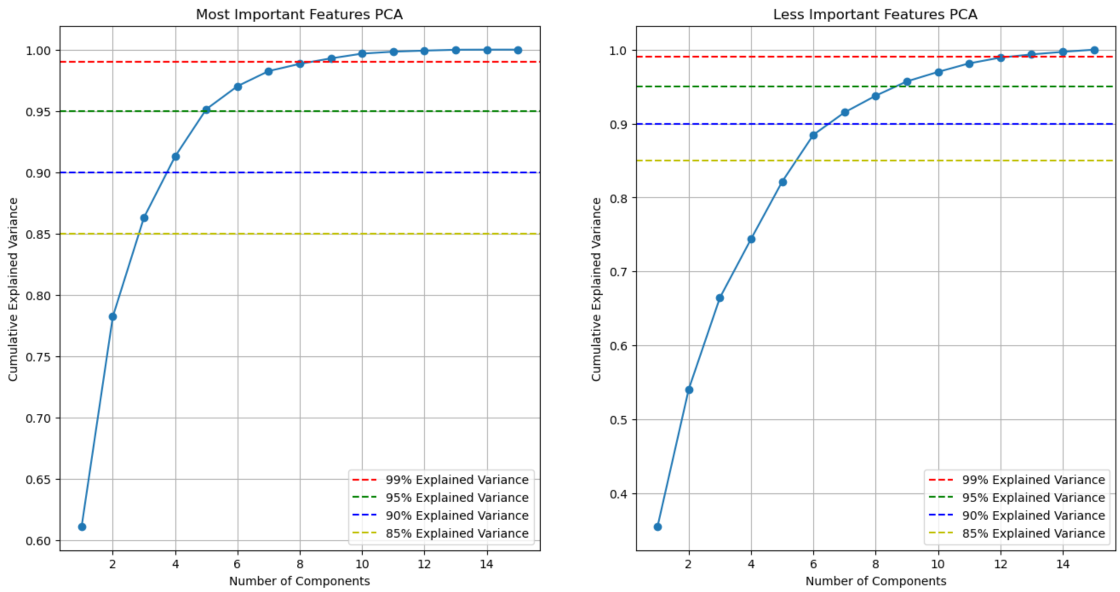
[ ]:
```
fig, ax = plt.subplots(1, 2, figsize=(16, 8))

more_important_pca = PCA()
principal_components_more_important = more_important_pca.
 ↪fit_transform(X_more_important)
sns.scatterplot(x=principal_components_more_important[:, 0],␣
 ↪y=principal_components_more_important[:, 1], hue=y_cancer, ax=ax[0])
ax[0].set_title('Most Important Features')

less_important_pca = PCA()
principal_components_less_important = less_important_pca.
 ↪fit_transform(X_less_important)
sns.scatterplot(x=principal_components_less_important[:, 0],␣
 ↪y=principal_components_less_important[:, 1], hue=y_cancer, ax=ax[1])
ax[1].set_title('Less Important Features')

plt.show()
```



[ ]:

```python
fig, ax = plt.subplots(1, 2, figsize=(16, 8))

def plot_variance_ratio(pca, ax):
    total_explained_variance = pca.explained_variance_ratio_.cumsum()
    number_of_components = np.arange(1, len(total_explained_variance) + 1)

    ax.plot(number_of_components, total_explained_variance, marker='o', linestyle='-')
    ax.set_xlabel('Number of Components')
    ax.set_ylabel('Cumulative Explained Variance')
    ax.grid(True)
    ax.axhline(y=0.99, color='r', linestyle='--', label='99% Explained Variance')
    ax.axhline(y=0.95, color='g', linestyle='--', label='95% Explained Variance')
    ax.axhline(y=0.90, color='b', linestyle='--', label='90% Explained Variance')
    ax.axhline(y=0.85, color='y', linestyle='--', label='85% Explained Variance')
    ax.legend(loc='best')

plot_variance_ratio(more_important_pca, ax[0])
ax[0].set_title('Most Important Features PCA')

plot_variance_ratio(less_important_pca, ax[1])
ax[1].set_title('Less Important Features PCA')
```

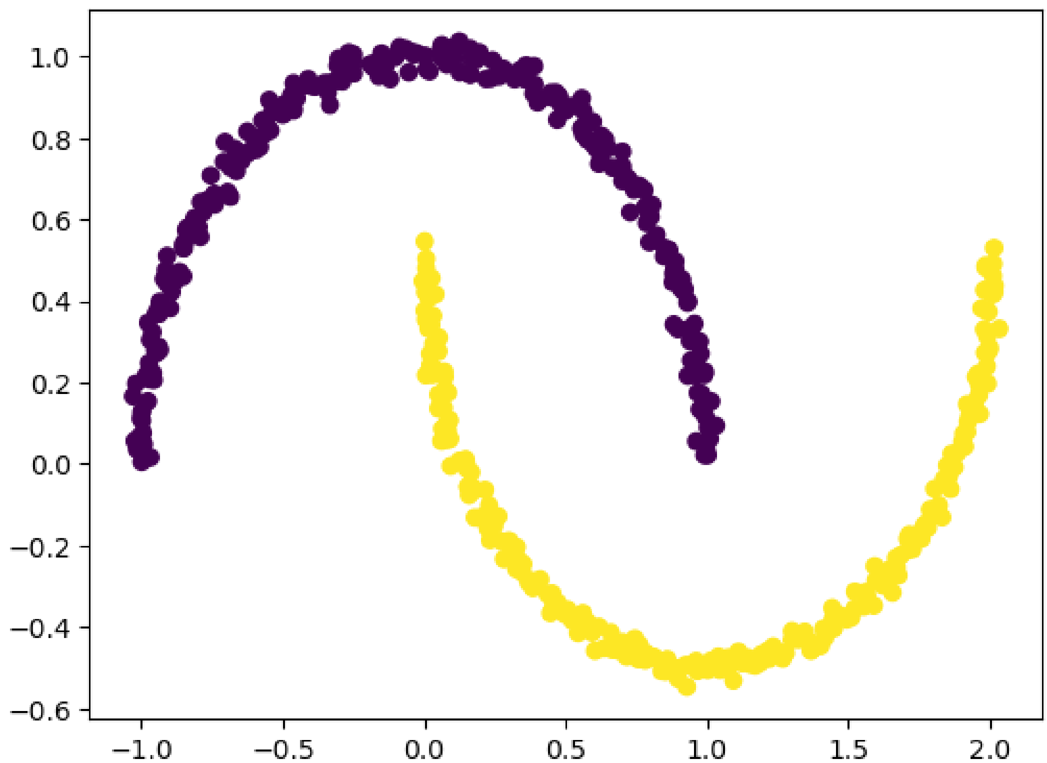
```
plt.show()
```



1.1 Kernel PCA

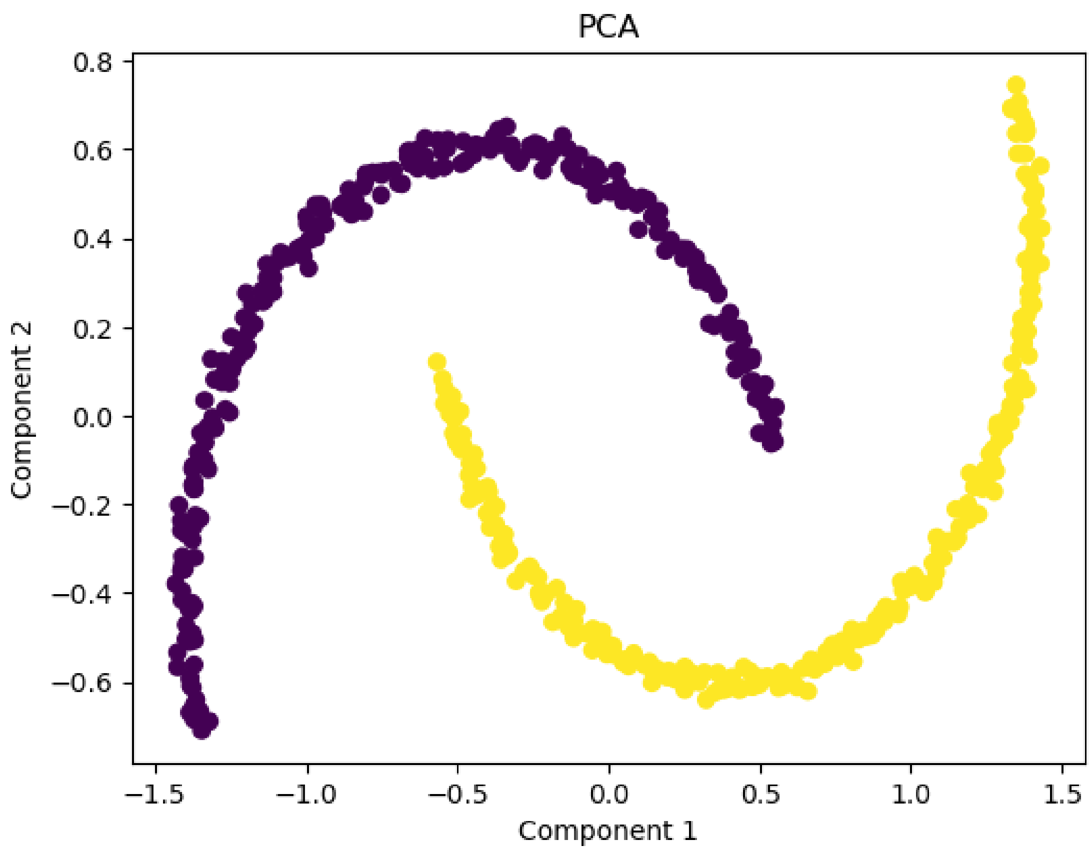
PCA is a linear method. That is it can only be applied to datasets which are linearly separable. It does an excellent job for datasets, which are linearly separable. But, if we use it to non-linear datasets, we might get a result which may not be the optimal dimensionality reduction. Kernel PCA uses a kernel function to project dataset into a higher dimensional feature space, where it is linearly separable. It is similar to the idea of Support Vector Machines.

[ ]:
```
import matplotlib.pyplot as plt
from sklearn.datasets import make_moons

X, y = make_moons(n_samples=500, noise=0.02, random_state=417)

plt.scatter(X[:, 0], X[:, 1], c=y)
plt.show()
```



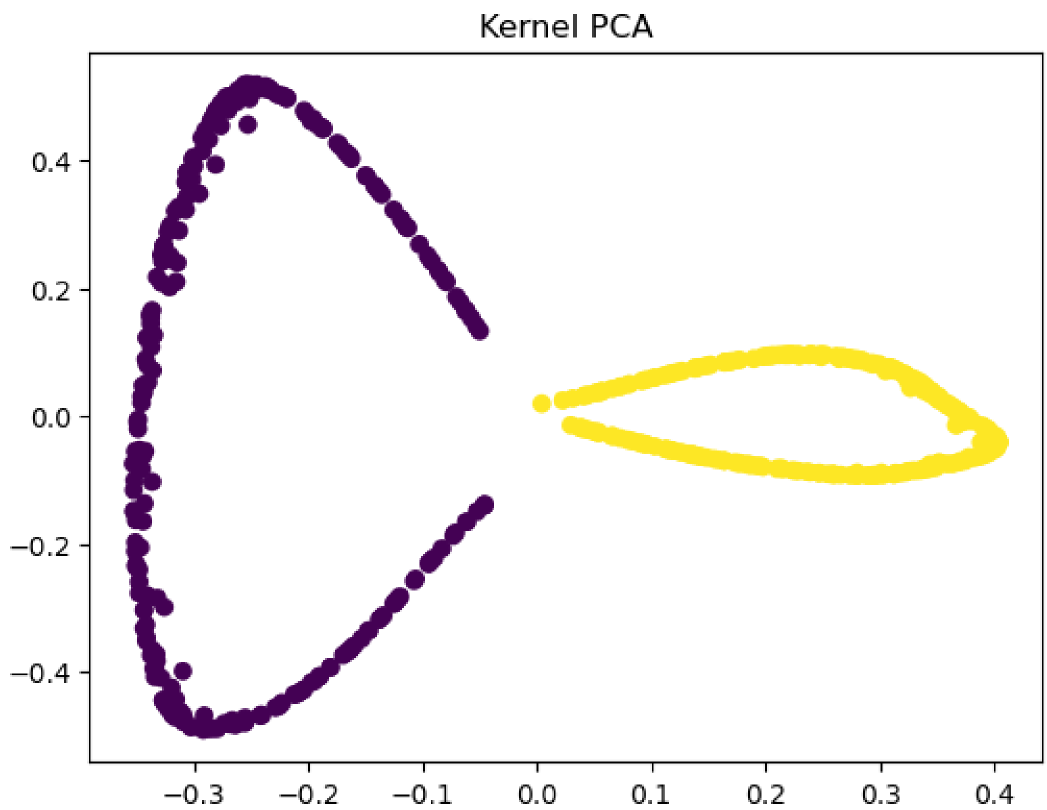
Let's apply PCA on this dataset

[ ]:
```
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X)

plt.title("PCA")
plt.scatter(X_pca[:, 0], X_pca[:, 1], c=y)
plt.xlabel("Component 1")
plt.ylabel("Component 2")
plt.show()
```



PCA failed to distinguish the two classes

[ ]:
```
from sklearn.decomposition import KernelPCA

kpca = KernelPCA(kernel='rbf', gamma=15)
X_kpca = kpca.fit_transform(X)

plt.title("Kernel PCA")
plt.scatter(X_kpca[:, 0], X_kpca[:, 1], c=y)
plt.show()
```



Applying kernel PCA on this dataset with RBF kernel with a gamma value of 15

1.1.1 KernelPCA exercises

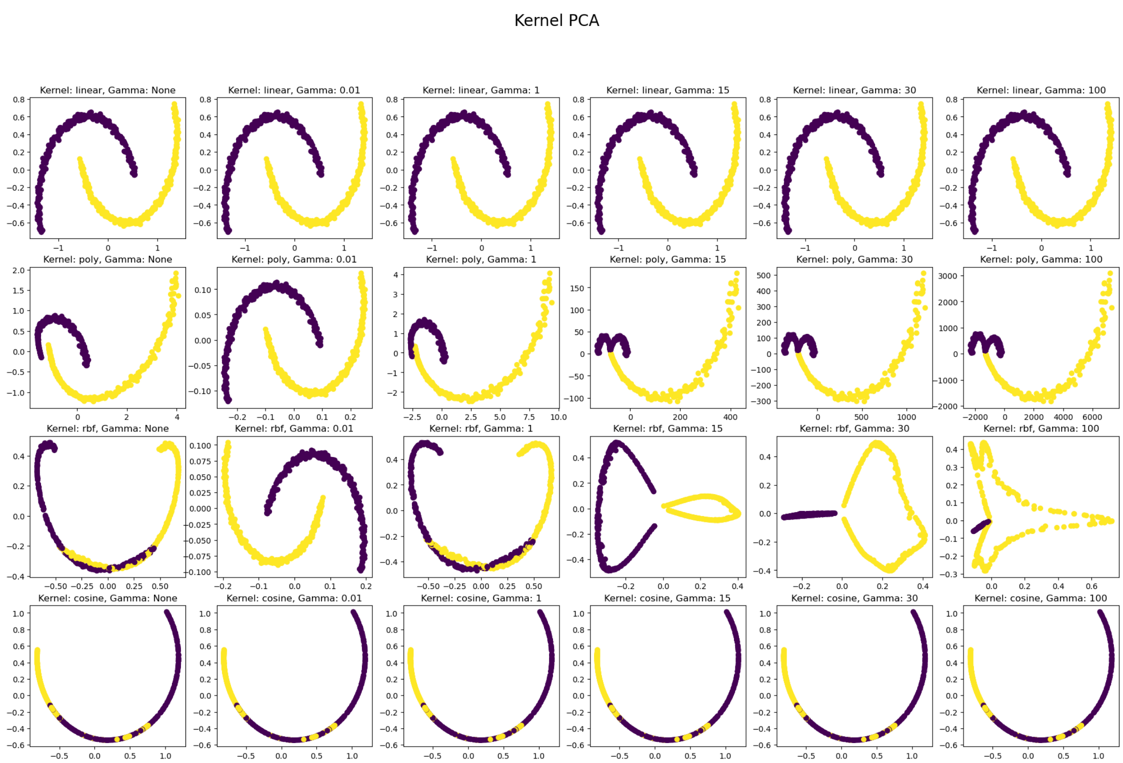
- Visualize in 2d datasets used in this labs, experiment with the parameters of the KernelPCA method, change kernel and gamma params. Docs: https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.KernelPCA.html

[ ]:
```
def compare_pca(X, y, kernels=['linear', 'poly', 'rbf', 'sigmoid', 'cosine',␣
↪'precomputed'], gammas=[None, .01, 1, 15, 30, 100]):
    fig, axes = plt.subplots(len(kernels), len(gammas), figsize=(25, 15),␣
↪squeeze=False)  # Ensure axes is always 2D
    fig.suptitle('Kernel PCA', fontsize=20)

    for i, kernel in enumerate(kernels):
        for j, gamma in enumerate(gammas):
            kpca = KernelPCA(kernel=kernel, gamma=gamma)
            X_kpca = kpca.fit_transform(X)

            # If axes is 2D, index it with two indices
```

```
            ax = axes[i, j] if len(kernels) > 1 and len(gammas) > 1 else axes[i][j] if len(gammas) > 1 else axes[j] if len(kernels) > 1 else axes[0]

            ax.scatter(X_kpca[:, 0], X_kpca[:, 1], c=y)
            ax.set_title(f'Kernel: {kernel}, Gamma: {gamma}')

    plt.show()
```

moons

[ ]:
```
compare_pca(X, y, kernels=['linear', 'poly', 'rbf', 'cosine'])
```



1.2 Homework

- Download the MNIST data set (there is a function to load this set in libraries such as scikit-learn, keras). It is a collection of black and white photos of handwritten digits with a resolution of 28x28 pixels. which together gives 784 dimensions.
- Try to visualize this dataset using PCA and KernelPCA, don't expect full separation of the data
- Similar to the exercises, examine explained variance. draw explained variance vs number of principal Components plot.
- Find number of principal components for 99%, 95%, 90%, and 85% of explained variance.

- Draw some sample MNIST digits and from PCA of its images transform data back to its original space (https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html#sklearn.decomposition.PCA.inverse_ Make an inverse transformation for number of components coresponding with explained variance shown above and draw the reconstructed images. The idea of this exercise is to see visually how depending on the number of components some information is lost.

- Perform the same reconstruction using KernelPCA (make comparisons for the same components number) https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.KernelPCA.html#sklearn.decomposition.Kernel

1.2.1 Data loading and visualization

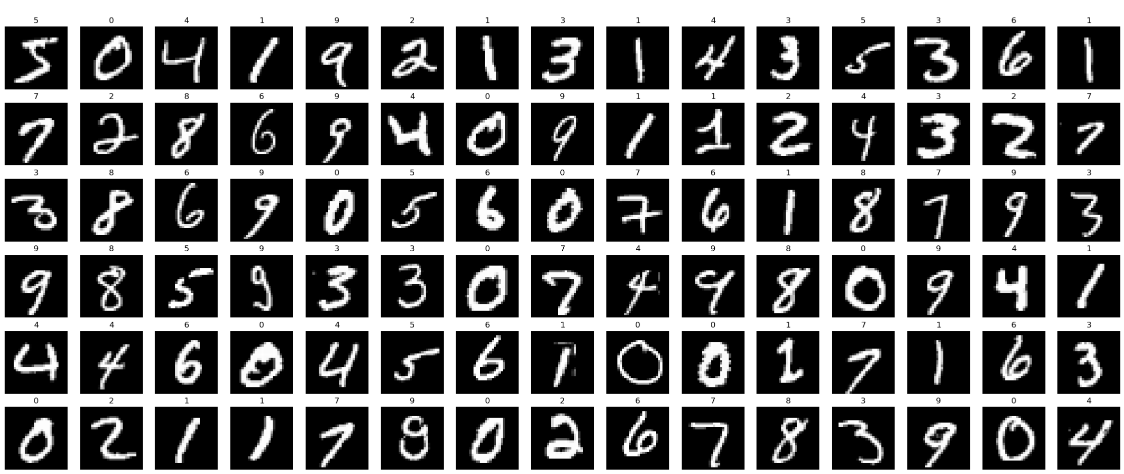
[ ]:
```
from keras.datasets import mnist
from sklearn.preprocessing import StandardScaler
from matplotlib import pyplot as plt
import pandas as pd
import torch
```

[ ]:
```
(X_train, y_train), (X_test, y_test) = mnist.load_data()
X = np.concatenate((X_train, X_test))
y = np.concatenate((y_train, y_test))
```

Let's visualize a few entries of our train data

[ ]:
```
def plot_images(images, labels, rows, cols):
    fig, axes = plt.subplots(rows, cols, figsize=(cols * 2, rows * 2))
    for i, ax in enumerate(axes.flat):
        ax.imshow(images[i], cmap='gray')
        ax.axis('off')
        ax.set_title(labels[i])
    plt.show()
```

[ ]:
```
plot_images(X, y, 6, 15)
```



1.2.2 Using PCA and KernelPCA to give better insight

First, we have to make the data 2-dimensional, so we reshape the original 3-dimensional array (each image has 2 dimensions and the original array is the array of images) into 2D array.

[ ]:
```
X_conv = X.reshape(X.shape[0], -1).astype('float32')

X_conv.shape
```

[ ]:
```
(70000, 784)
```

Standarize the data

[ ]:
```
X_scaled = StandardScaler().fit_transform(X_conv)
```

Because the MNIST dataset contains 70000 elements, we will use only a part of this dataset for PCA and kernel PCA. Running `KernelPCA` with `rbf` kernel on my PC with 16 GB of RAM was crashing on the whole dataset because of inability to allocate memory.

[ ]:
```
LIMIT = 10000

X_small = X_conv[:LIMIT]
y_small = y[:LIMIT]
```

Use PCA and KernelPCA to reduce the number of features to easy to visualize 2 dimensions

[ ]:
```
pca = PCA()
pca.fit(X_small)
```

[ ]:
```
PCA()
```

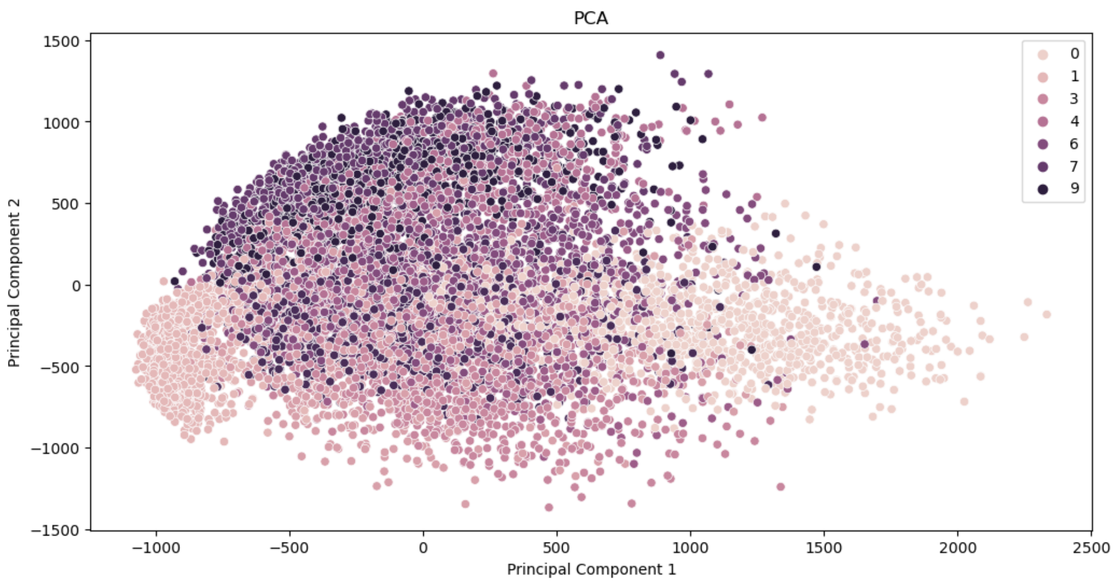
[ ]:
```
kpca = KernelPCA(kernel='rbf', gamma=15)
kpca.fit(X_small)
```

[ ]:
```
KernelPCA(gamma=15, kernel='rbf')
```

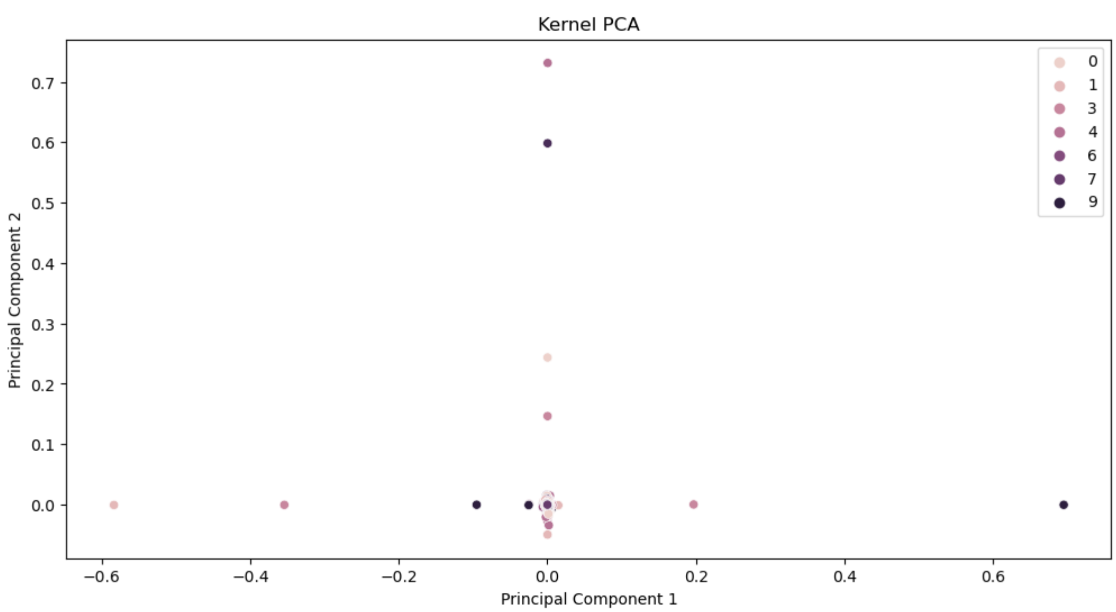
[ ]:
```
X_pca = pca.transform(X_small)
```

[ ]:
```
X_kpca = kpca.transform(X_small)
```

[ ]:
```
plt.figure(figsize=(12, 6))
sns.scatterplot(x=X_pca[:, 0], y=X_pca[:, 1], hue=y_small)
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.title('PCA')
plt.show()
```




```
[ ]: plt.figure(figsize=(12, 6))
sns.scatterplot(x=X_kpca[:, 0], y=X_kpca[:, 1], hue=y_small)
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.title('Kernel PCA')
plt.show()
```

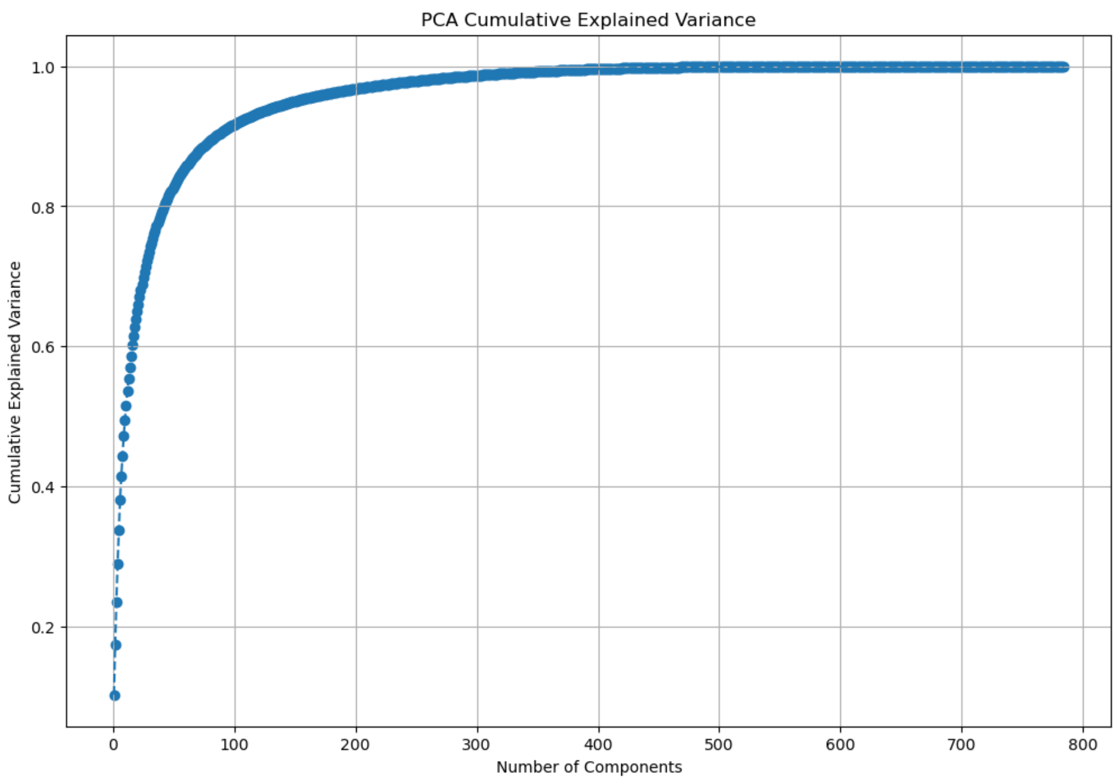


Neither of methods used (PCA and KernelPCA) could separate points properly. We can still see meny overlapping points on both scatter plots above.

1.2.3 Explained variance

PCA

[ ]:
```
total_explained_variance = pca.explained_variance_ratio_.cumsum()
number_of_components = np.arange(1, len(total_explained_variance) + 1)

plt.figure(figsize=(12, 8))
plt.scatter(number_of_components, total_explained_variance)
plt.plot(number_of_components, total_explained_variance, linestyle='--')
plt.xlabel('Number of Components')
plt.ylabel('Cumulative Explained Variance')
plt.title('PCA Cumulative Explained Variance')
plt.grid(True)
plt.show()
```



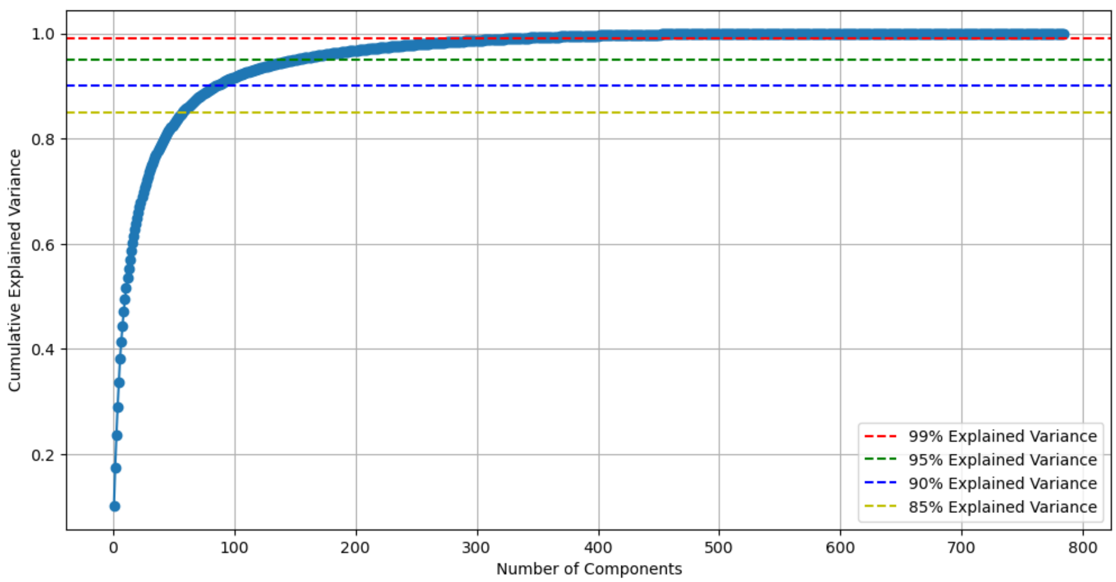
As we can see, the speed of the cumulative explained variance growth decreases significantly after the first 200 components. The curve, is not too steep, though, which means that there are no just a few components that explain most of the variance.

KernelPCA For Kernel PCA, creating explained variance plots is not straightforward or typically done. The eigenvalues from the kernel matrix do not represent variance in the original feature space, complicating the creation of explained variance plots.

1.2.4 Numbers of principal components for 99%, 95%, 90% and 85% of cumulative explained variance

[ ]:
```
fig = plt.figure(figsize=(12, 6))
plot_variance_ratio(pca, fig.add_subplot(1, 1, 1))

thresholds = np.array([0.99, 0.95, 0.90, 0.85])
is_exceeded = total_explained_variance[:, np.newaxis] >= thresholds
components = np.argmax(is_exceeded, axis=0) + 1
for threshold, component in zip(thresholds, components):
    print(f'Number of components for {int(threshold * 100)}% explained variance:
 ↪ {component}')
```

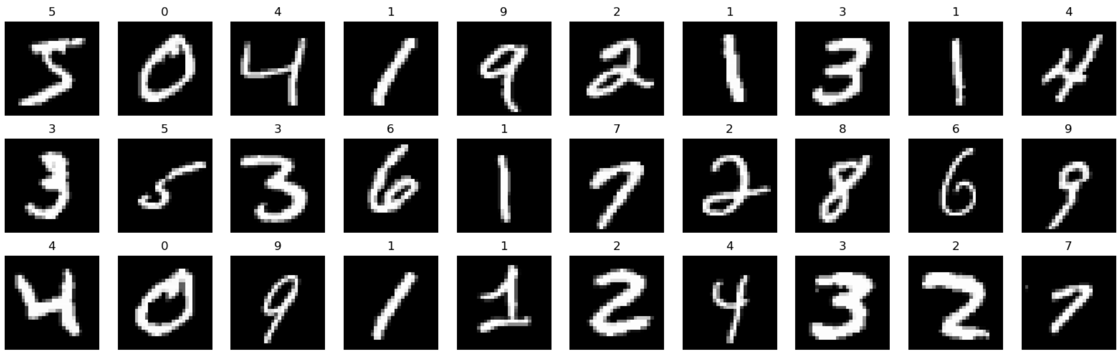
```
Number of components for 99% explained variance: 326
Number of components for 95% explained variance: 150
Number of components for 90% explained variance: 85
Number of components for 85% explained variance: 58
```



1.2.5 Image reconstruction for threshold values of number of PCA components

Let's draw a few digits we want to use as a reference after reconstruction from PCA

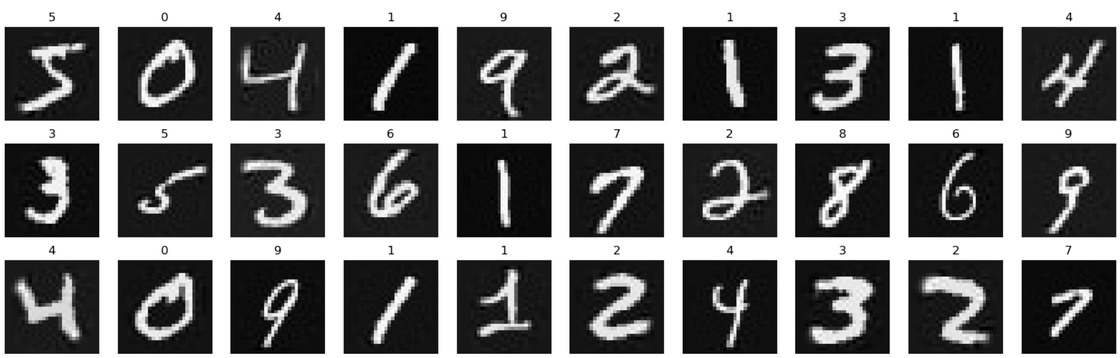
[ ]:
```
plot_images(X, y, 3, 10)
```



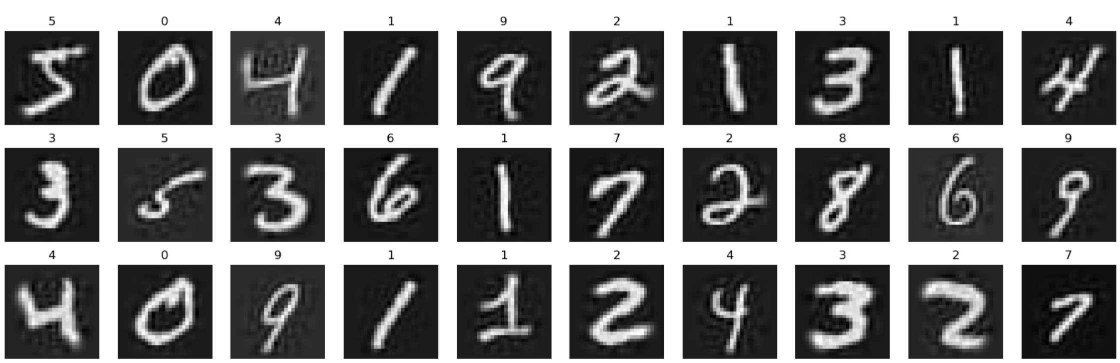
Draw reconstructed digits from PCA transformation

[ ]:
```
for n in components:
    pca = PCA(n_components=n)
    X_pca = pca.fit_transform(X_small)
    X_reconstructed = pca.inverse_transform(X_pca)
    print(f'Number of components: {n}')
    plot_images(X_reconstructed.reshape(-1, 28, 28), y_small, 3, 10)
```

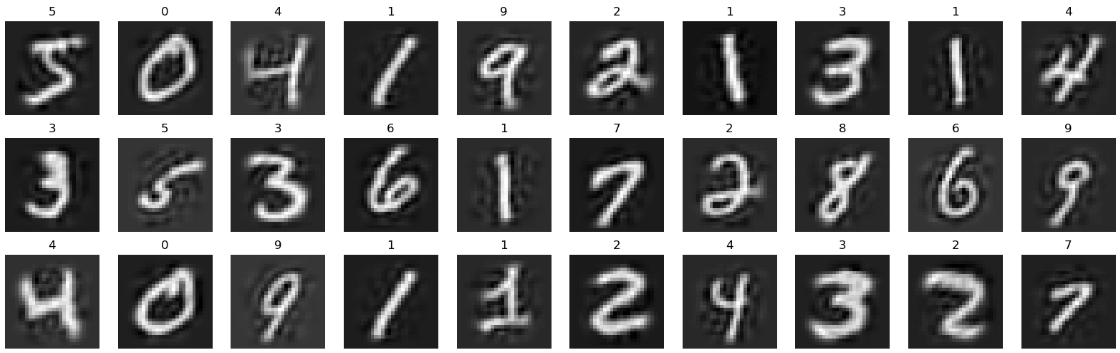
```
Number of components: 326
```



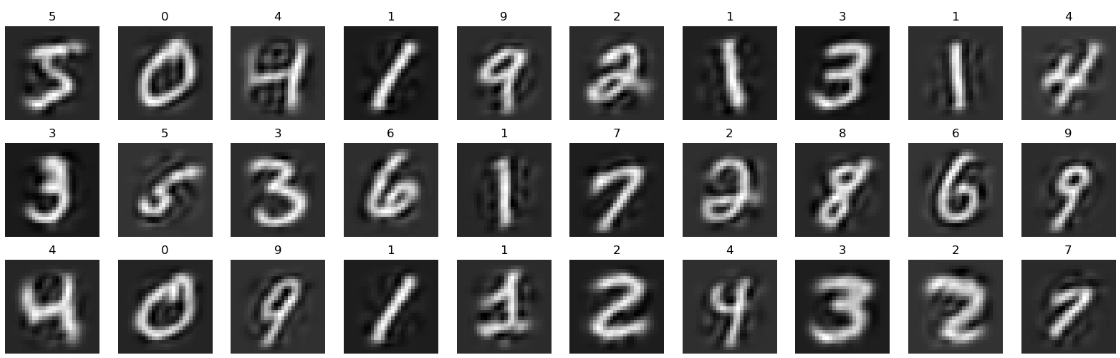

```
Number of components: 150
```



Number of components: 85



Number of components: 58

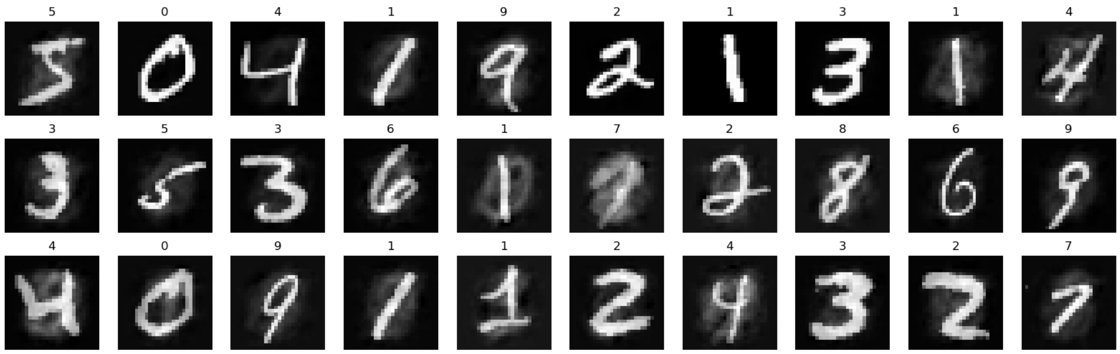


We can see that even for 58 components digits are readable and easily recognizable. In the direct comparison, changes are visible between ach number of principal components we used above (mostly between 326 and 58 components).

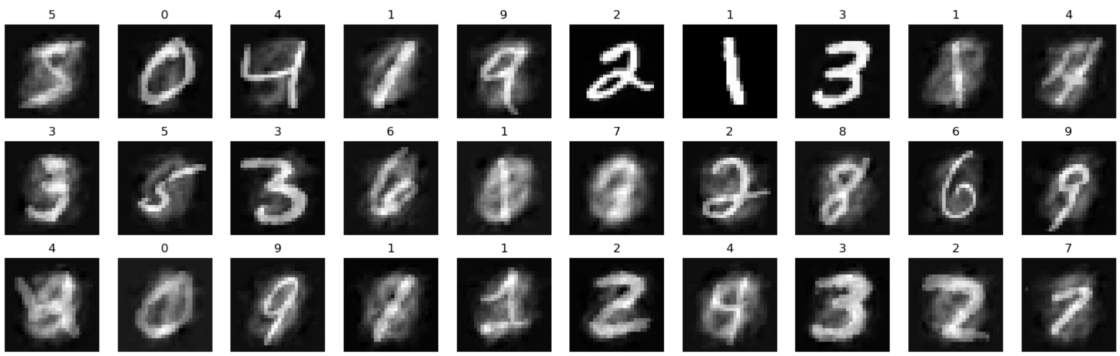
1.2.6 Reconstruction for Kernel PCA

[ ]:
```
for n in components:
    kpca = KernelPCA(n_components=n, kernel='rbf', gamma=15,␣
 ↪fit_inverse_transform=True)
    X_kpca = kpca.fit_transform(X_small)
    X_reconstructed = kpca.inverse_transform(X_kpca)
    print(f'Number of components: {n}')
    plot_images(X_reconstructed.reshape(-1, 28, 28), y_small, 3, 10)
```

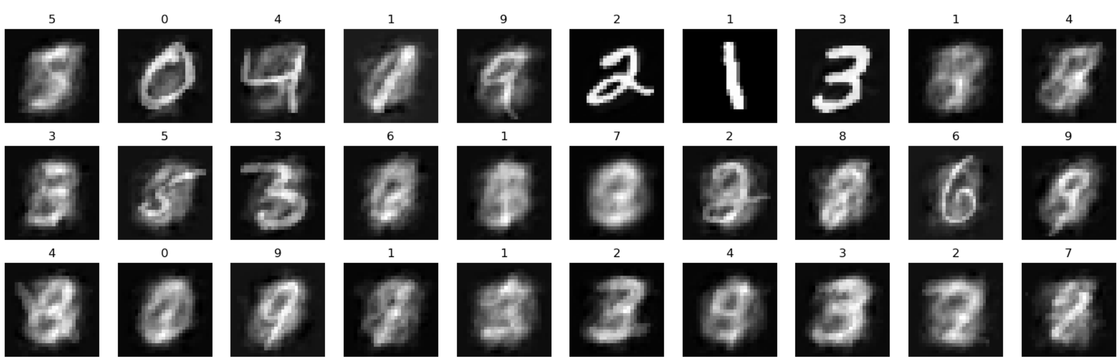
Number of components: 326



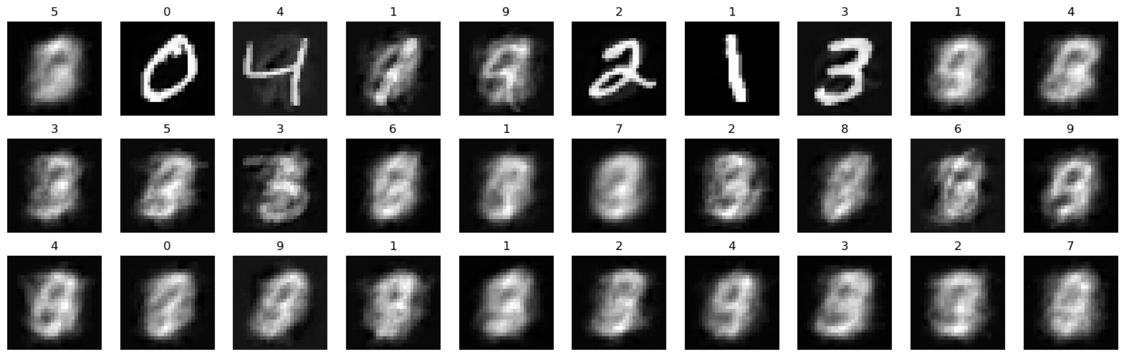
Number of components: 150



Number of components: 85



Number of components: 58



KernelPCA gives much worse results for the MNIST dataset. For the 326 components numbers are readable but there is a slight blur in the center (seems like features from other digits are displayed there as well). When the number of principal components decreases, the results are getting much worse.

1.3 Useful links

https://scikit-learn.org

https://towardsdatascience.com/introduction-to-principal-component-analysis-pca-with-python-code-69d3fcf19b57

https://towardsdatascience.com/kernel-pca-vs-pca-vs-ica-in-tensorflow-sklearn-60e17eb15a64