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!

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). 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 are 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.

In [2]:
```
import pandas as pd
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
from sklearn.datasets import load_boston
#from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.feature_selection import RFE
#from sklearn.linear_model import RidgeCV, LassoCV, Ridge, Lasso


%matplotlib inline
```

In [3]:
```
%%javascript
IPython.OutputArea.prototype._should_scroll = function(lines) {
    return false;
}
```

In [4]:
```
iris_url = "https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data"
```

In [5]:
```
# loading dataset into Pandas DataFrame
df_iris = pd.read_csv(iris_url ,names=['sepal length','sepal width','petal length','petal
```

In [6]:
```
df_iris.head(15)
```

Out[6]:

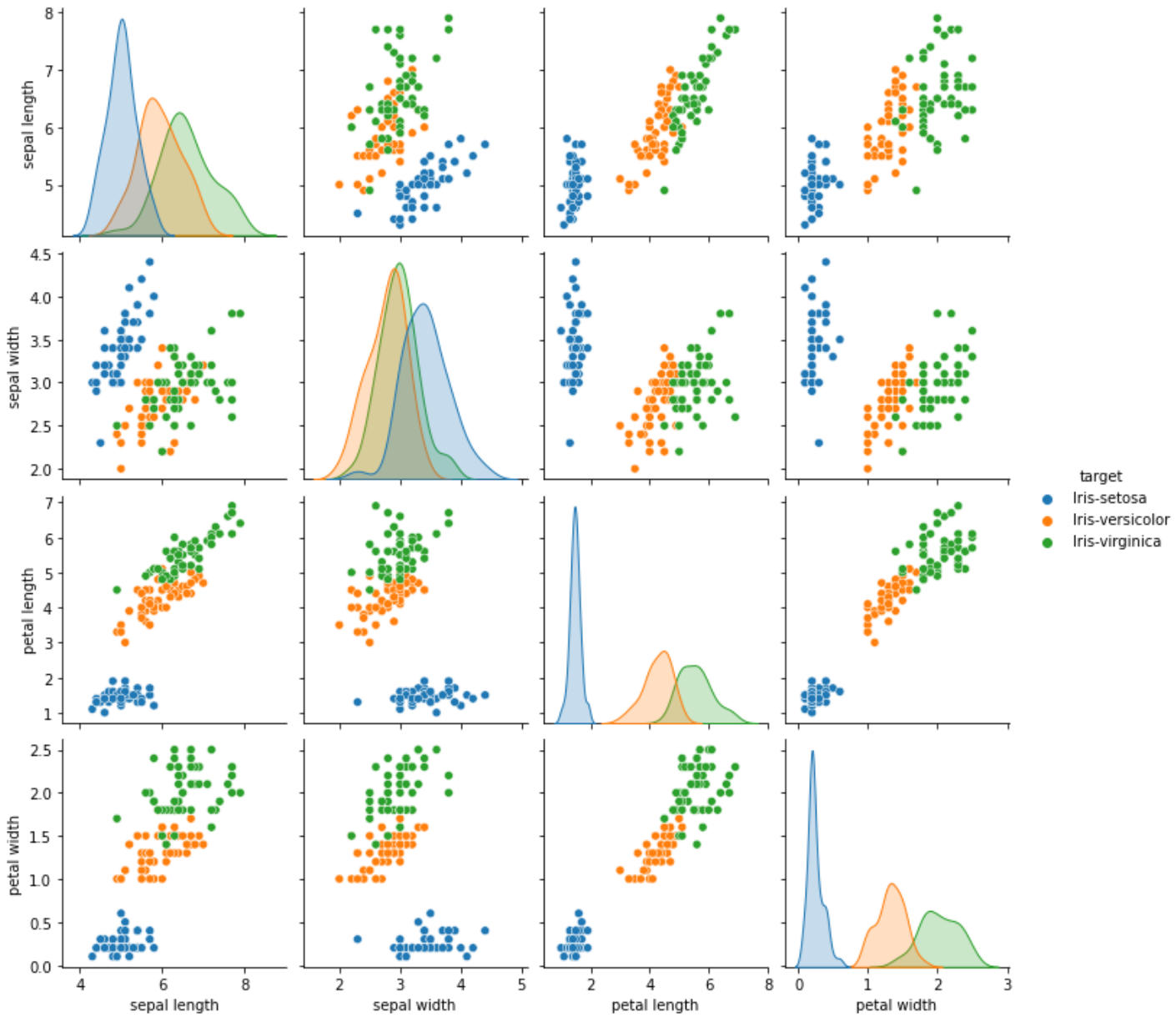
	sepal length	sepal width	petal length	petal width	target
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa
5	5.4	3.9	1.7	0.4	Iris-setosa
6	4.6	3.4	1.4	0.3	Iris-setosa
7	5.0	3.4	1.5	0.2	Iris-setosa

	sepal length	sepal width	petal length	petal width	target
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

In [7]:
```
sns.pairplot(df_iris, hue='target')
```

Out[7]:
```
<seaborn.axisgrid.PairGrid at 0x13882bbb0>
```



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

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.

In [8]:
```
features_iris = ['sepal length', 'sepal width', 'petal length', 'petal width']
x_iris = df_iris.loc[:, features_iris].values
```

In [9]:
```
y_iris = df_iris.loc[:,['target']].values
```

In [10]:
```
x_iris = StandardScaler().fit_transform(x_iris)
```

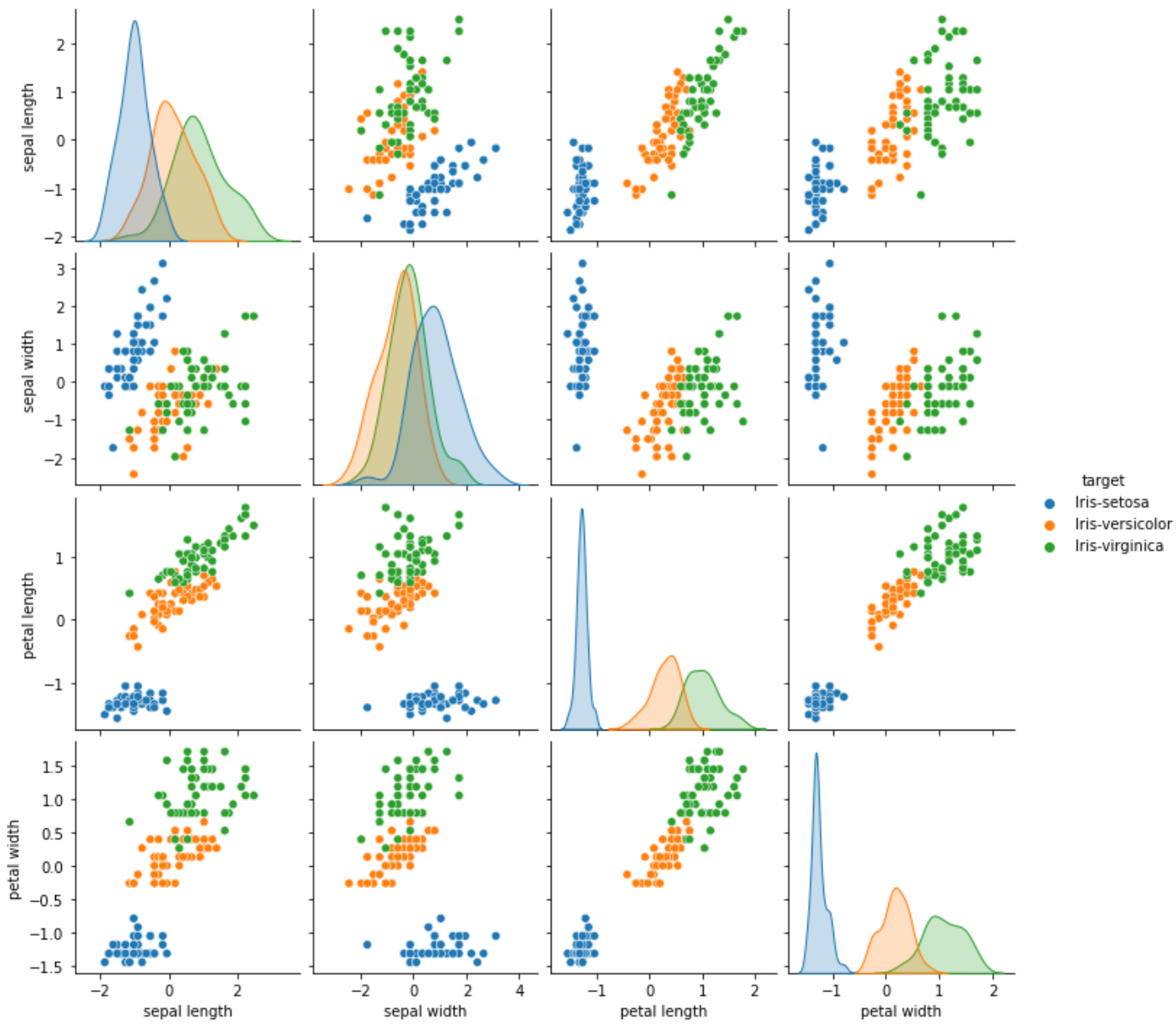
In [11]:
```
df_iris_standarize = pd.DataFrame(data = x_iris, columns = features_iris)
df_iris_standarize['target'] = df_iris['target']
df_iris_standarize.head(15)
```

Out[11]:

	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

In [12]:
```
sns.pairplot(df_iris_standarize, hue='target')
```

Out[12]:
```
<seaborn.axisgrid.PairGrid at 0x13b0d72e0>
```



We can see that the distributions are now standardized

PCA Projection to 2D

In [13]:
```
pca_iris = PCA(n_components=2)
```

In [14]:
```
principalComponents_iris = pca_iris.fit_transform(x_iris)
```

In [15]:
```
principal_df_cancer = pd.DataFrame(data = principalComponents_iris, columns = ['principal
```

In [16]:
```
finalDf_iris = pd.concat([principal_df_cancer, df_iris[['target']]], axis = 1)
finalDf_iris.head(15)
```

Out[16]:

	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

	principal component 1	principal component 2	target
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

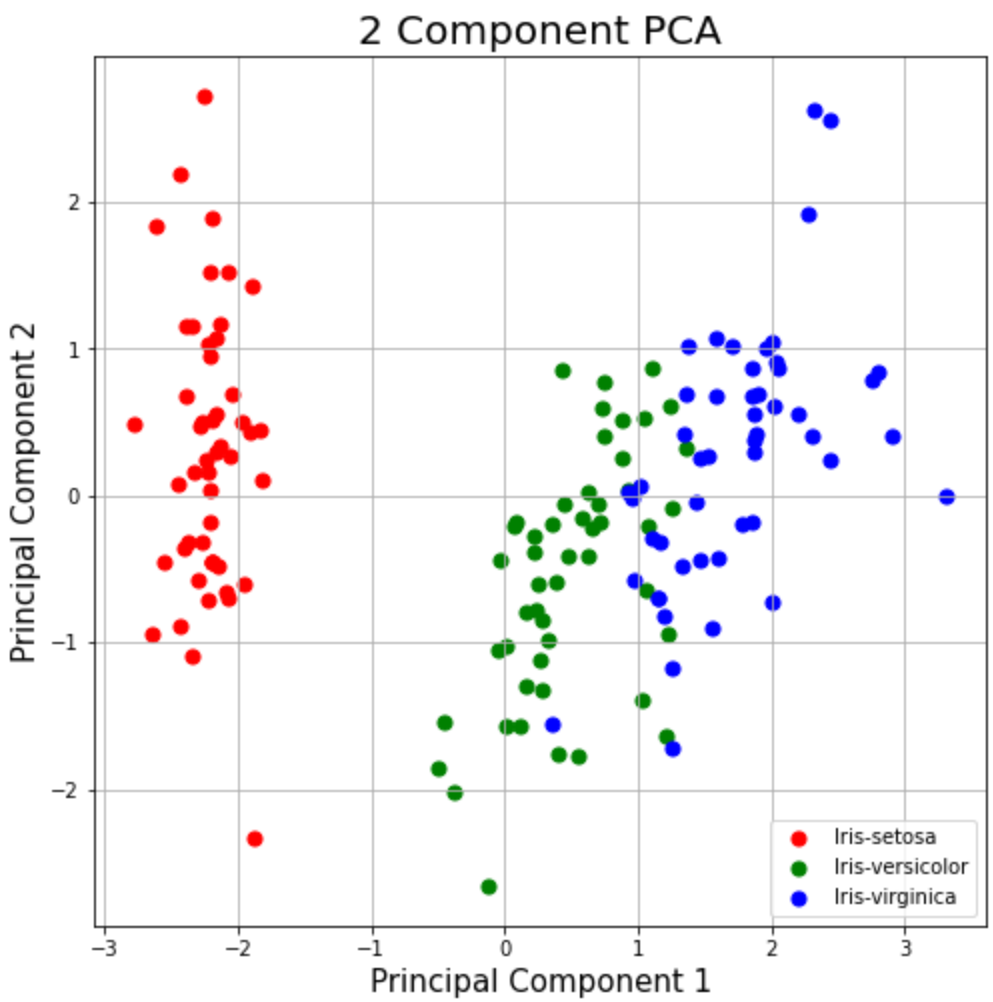
Visualize 2D Projection

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

In [17]:

```
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):
    indicesToKeep = finalDf_iris['target'] == target
    ax.scatter(finalDf_iris.loc[indicesToKeep, 'principal component 1']
               , finalDf_iris.loc[indicesToKeep, 'principal component 2']
               , c = color
               , s = 50)
ax.legend(iris_targets)
ax.grid()
```



iris-setosa is linearry separablo from others class

Explained Variance

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

In [18]:
```
pca_iris.explained_variance_ratio_
```

Out[18]:
```
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.

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.

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'

In [19]:
```
from sklearn.datasets import load_breast_cancer

cancer = load_breast_cancer()
df_cancer = pd.DataFrame(cancer.data, columns=[cancer.feature_names])
df_cancer['target'] = pd.Categorical(pd.Series(cancer.target).map(lambda x: cancer.target_

df_cancer.head(15)
```

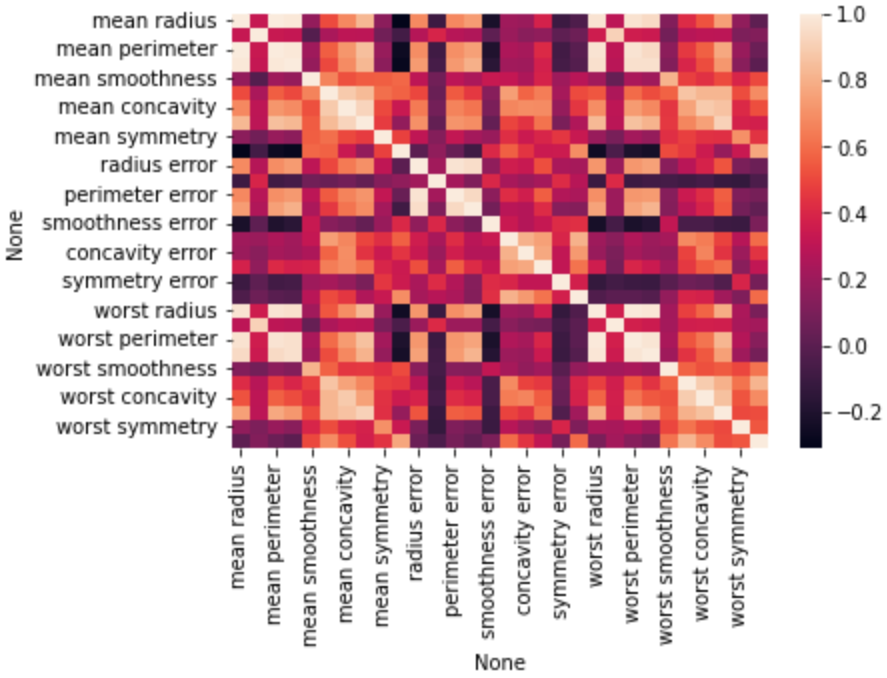
Out[19]:

	mean radius	mean texture	mean perimeter	mean area	mean smoothness	mean compactness	mean concavity	mean concave points	mean symmetry	mean fractal dimension
0	17.99	10.38	122.80	1001.0	0.11840	0.27760	0.30010	0.14710	0.2419	0.07871
1	20.57	17.77	132.90	1326.0	0.08474	0.07864	0.08690	0.07017	0.1812	0.05667
2	19.69	21.25	130.00	1203.0	0.10960	0.15990	0.19740	0.12790	0.2069	0.05999
3	11.42	20.38	77.58	386.1	0.14250	0.28390	0.24140	0.10520	0.2597	0.09744
4	20.29	14.34	135.10	1297.0	0.10030	0.13280	0.19800	0.10430	0.1809	0.05883
5	12.45	15.70	82.57	477.1	0.12780	0.17000	0.15780	0.08089	0.2087	0.07613
6	18.25	19.98	119.60	1040.0	0.09463	0.10900	0.11270	0.07400	0.1794	0.05742
7	13.71	20.83	90.20	577.9	0.11890	0.16450	0.09366	0.05985	0.2196	0.07451
8	13.00	21.82	87.50	519.8	0.12730	0.19320	0.18590	0.09353	0.2350	0.07389
9	12.46	24.04	83.97	475.9	0.11860	0.23960	0.22730	0.08543	0.2030	0.08243
10	16.02	23.24	102.70	797.8	0.08206	0.06669	0.03299	0.03323	0.1528	0.05697
11	15.78	17.89	103.60	781.0	0.09710	0.12920	0.09954	0.06606	0.1842	0.06082
12	19.17	24.80	132.40	1123.0	0.09740	0.24580	0.20650	0.11180	0.2397	0.07800
13	15.85	23.95	103.70	782.7	0.08401	0.10020	0.09938	0.05364	0.1847	0.05338
14	13.73	22.61	93.60	578.3	0.11310	0.22930	0.21280	0.08025	0.2069	0.07682

15 rows × 31 columns

In [20]:
```
sns.heatmap(df_cancer.corr())
```

Out[20]:
```
<AxesSubplot:xlabel='None', ylabel='None'>
```



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

In [21]:

```
breast_cancer_pca = PCA(n_components=2)
df_cancer_data = df_cancer.loc[:, df_cancer.columns != ('target',)]
df_cancer_standardized = StandardScaler().fit_transform(df_cancer_data)
pc_breast_cancer = breast_cancer_pca.fit_transform(df_cancer_standardized)
principal_df_cancer = pd.DataFrame(data=pc_breast_cancer,
                                   columns=['principal component 1', 'principal component

final_df_cancer = pd.concat([principal_df_cancer, df_cancer[['target']]], axis=1)
final_df_cancer.head(3)
```

```
/usr/local/lib/python3.9/site-packages/sklearn/utils/validation.py:1675: FutureWarning: Fe
ature names only support names that are all strings. Got feature names with dtypes: ['tupl
e']. An error will be raised in 1.2.
  warnings.warn(
/usr/local/lib/python3.9/site-packages/sklearn/utils/validation.py:1675: FutureWarning: Fe
ature names only support names that are all strings. Got feature names with dtypes: ['tupl
e']. An error will be raised in 1.2.
  warnings.warn(
```

Out[21]:

	principal component 1	principal component 2	(target,)
0	9.192837	1.948583	malignant
1	2.387802	-3.768172	malignant
2	5.733896	-1.075174	malignant

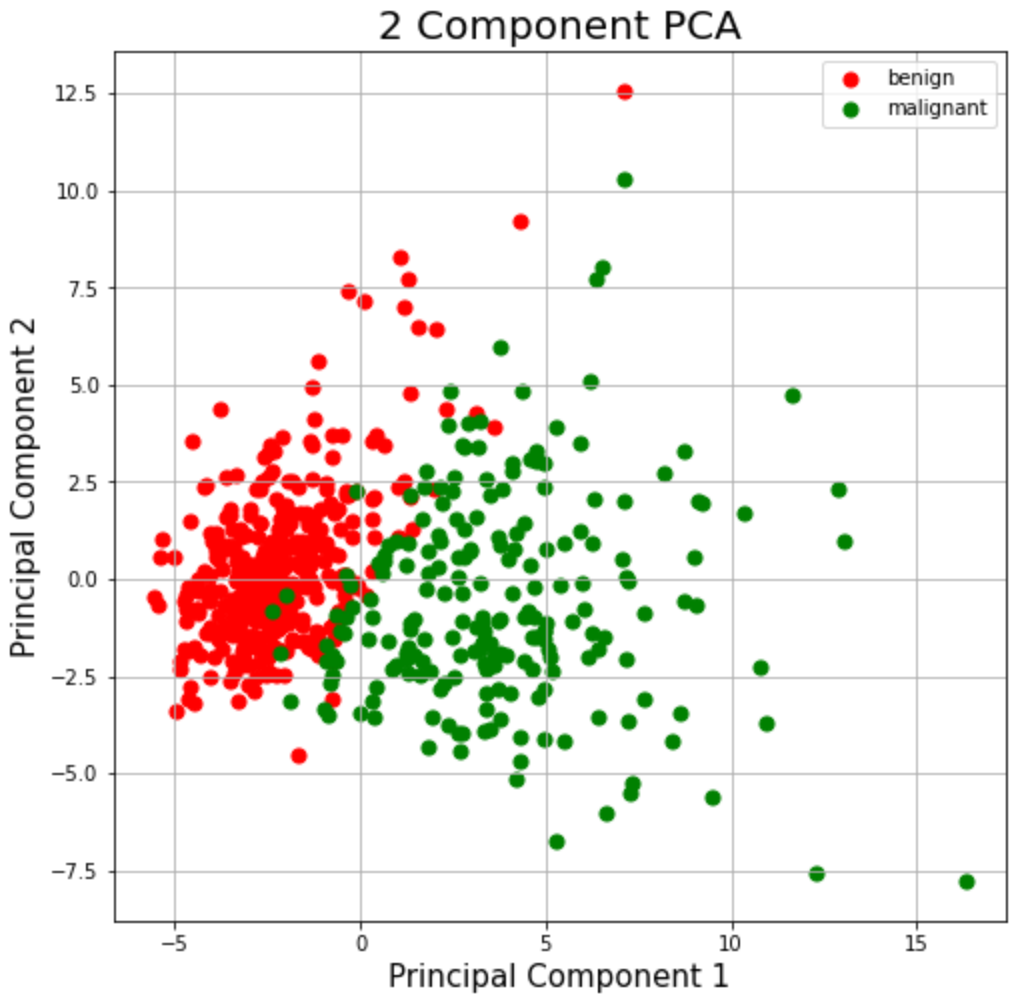
- Perform PCA and visualize the data

In [22]:

```
# TODO
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)

cancer_target_values = np.unique(df_cancer[['target']].values)
```

```
colors = ['r', 'g']
for target, color in zip(cancer_target_values,colors):
    indicesToKeep = final_df_cancer[('target',)] == target
    ax.scatter(final_df_cancer.loc[indicesToKeep, 'principal component 1']
               , final_df_cancer.loc[indicesToKeep, 'principal component 2']
               , c = color
               , s = 50)
ax.legend(cancer_target_values)
ax.grid()
```



In [22]:

In [23]:
```
# TODO
breast_cancer_pca.explained_variance_ratio_
```

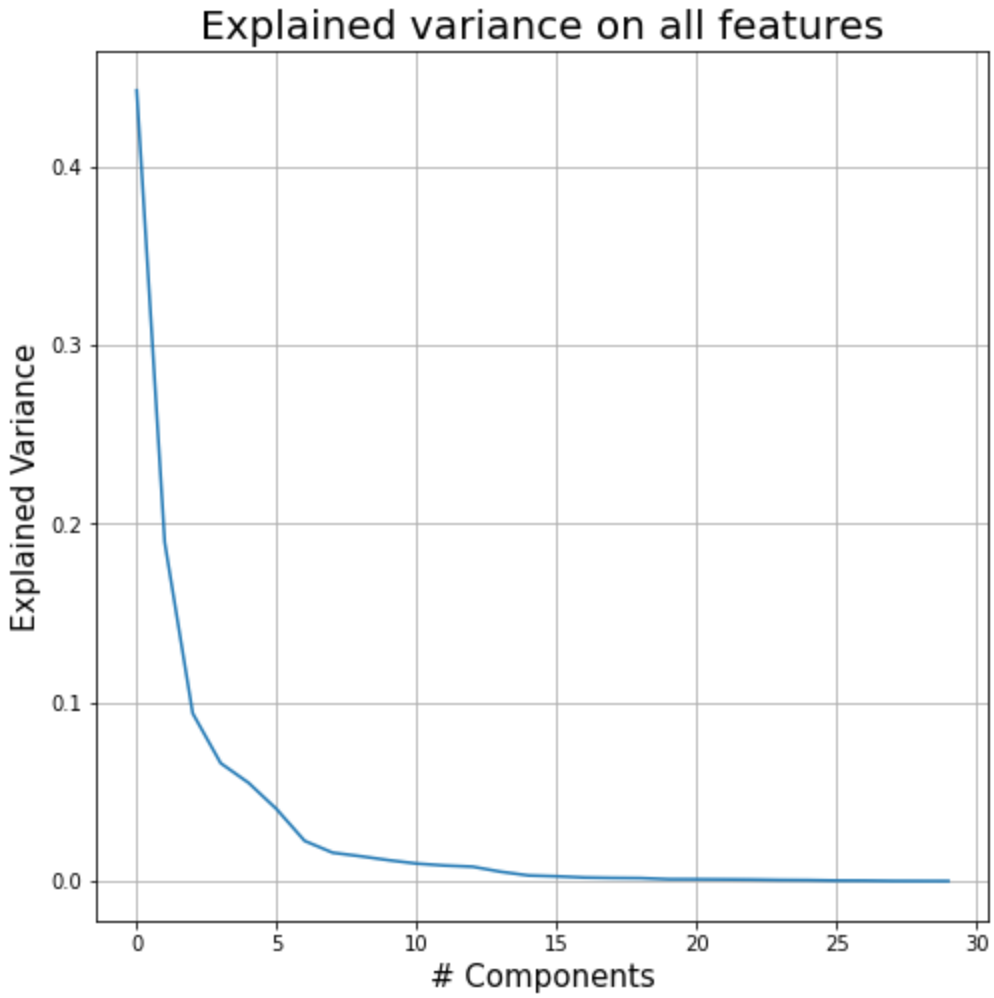
Out[23]:
```
array([0.44272026, 0.18971182])
```

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

In [24]:
```
pca = PCA()
pca.fit_transform(df_cancer_standardized)

fig = plt.figure(figsize = (8,8))
ax = fig.add_subplot(1,1,1)
ax.set_xlabel('# Components', fontsize = 15)
ax.set_ylabel('Explained Variance', fontsize = 15)
ax.set_title('Explained variance on all features', fontsize = 20)
```

```
ax.plot(range(len(pca.explained_variance_)), pca.explained_variance_ratio_)
ax.grid()
```



- Use recursive feature elimination (available in scikit-learn module) or another feature ranking algorithm to split 30 features to on 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.

In [25]:
```
# TODO
estimator = LinearRegression()
selector = RFE(estimator, n_features_to_select=15, step=1)

def target_mapper(target_value):
    if target_value == 'malignant':
        return 1
    else:
        return 0

target_normalized = df_cancer[['target']].apply(lambda x: x.map(target_mapper))
selector.fit(df_cancer_standardized, target_normalized)
```

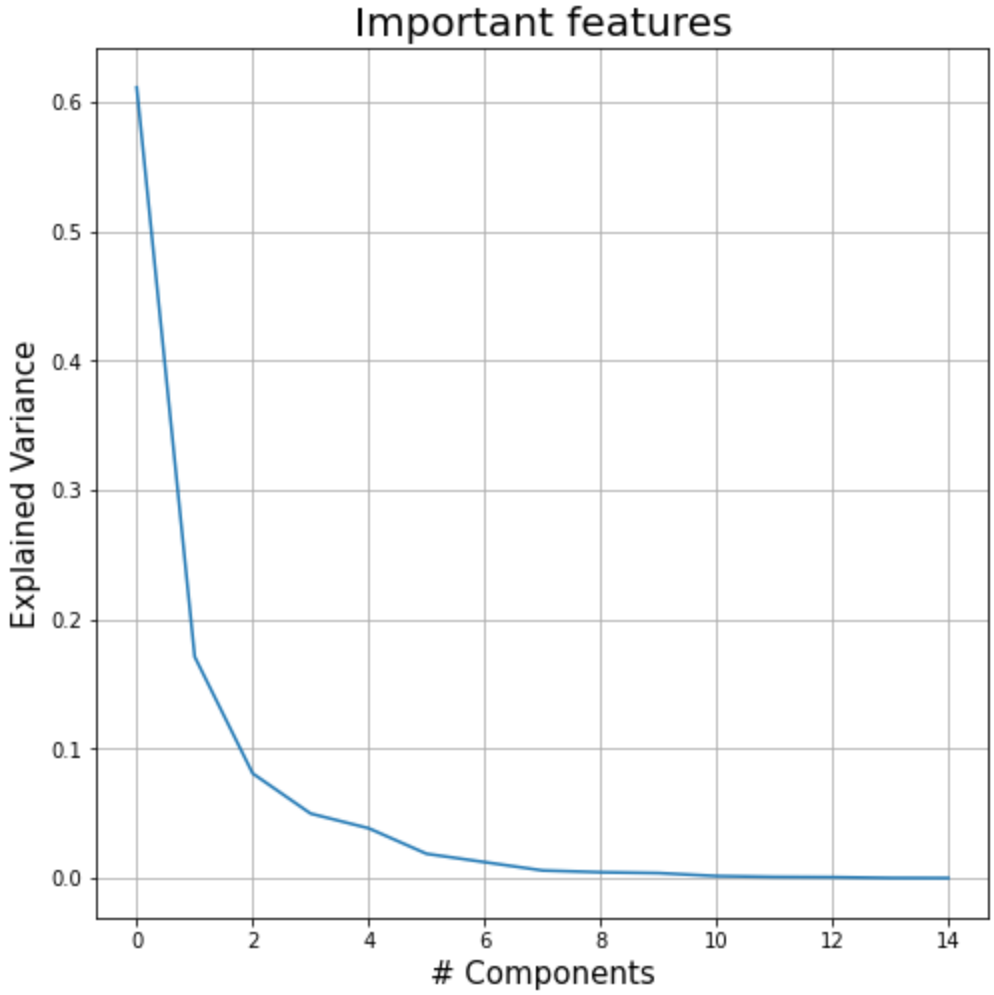
Out[25]:
```
RFE(estimator=LinearRegression(), n_features_to_select=15)
```

In [26]:
```
df_important_features = df_cancer_standardized[:, selector.support_]

pca = PCA()
pca.fit_transform(df_important_features)

fig = plt.figure(figsize = (8,8))
```

```
ax = fig.add_subplot(1,1,1)
ax.set_xlabel('# Components', fontsize = 15)
ax.set_ylabel('Explained Variance', fontsize = 15)
ax.set_title('Important features', fontsize = 20)
ax.plot(range(len(pca.explained_variance_)), pca.explained_variance_ratio_)
ax.grid()
```

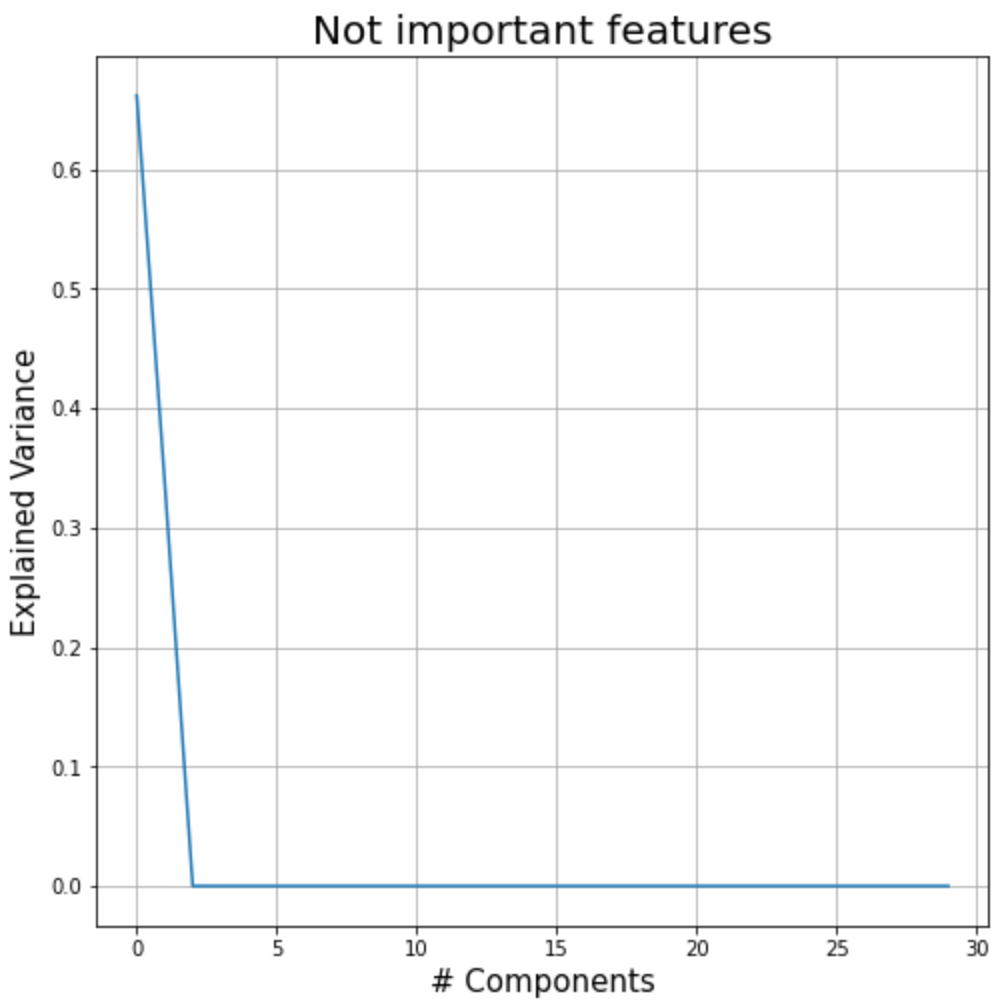


In [27]:

```
df_not_important_features = df_cancer_standardized[:, 1 - selector.support_]

pca = PCA()
pca.fit_transform(df_not_important_features)

fig = plt.figure(figsize = (8,8))
ax = fig.add_subplot(1,1,1)
ax.set_xlabel('# Components', fontsize = 15)
ax.set_ylabel('Explained Variance', fontsize = 15)
ax.set_title('Not important features', fontsize = 20)
ax.plot(range(len(pca.explained_variance_)), pca.explained_variance_ratio_)
ax.grid()
```



We can see that when we perform PCA on important features, we can see that even though explained variance is diminishing for more components it's still positive. That means that those feautes still hold some meanigful information. When we look at PCA performed on non-important features, we can see that almost entire variance is explained by a couple features. That's due to the fact that other features simply don't convey a lot of meaningful information (as suggested by the estimator - I chose linear regression).

Kernel PCA

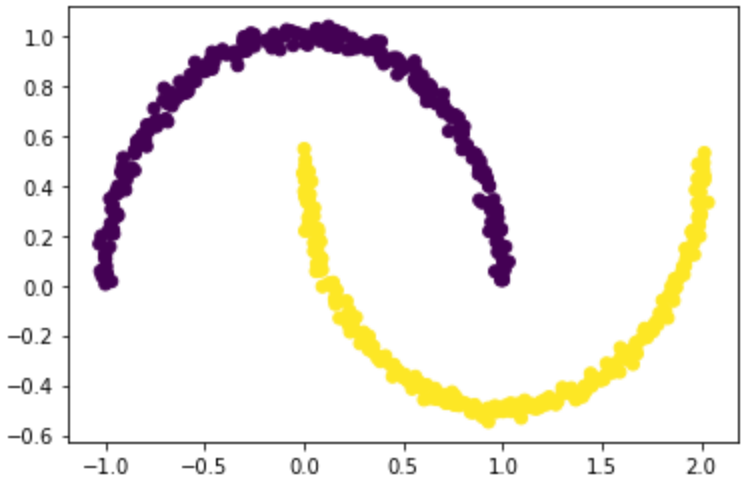
PCA is a linear method. That is it can only be applied to datasets which are linearly separable. It does an excellent job for datasets, which are linearly separable. But, if we use it to non-linear datasets, we might get a result which may not be the optimal dimensionality reduction. Kernel PCA uses 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.

In [28]:

```
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()
```

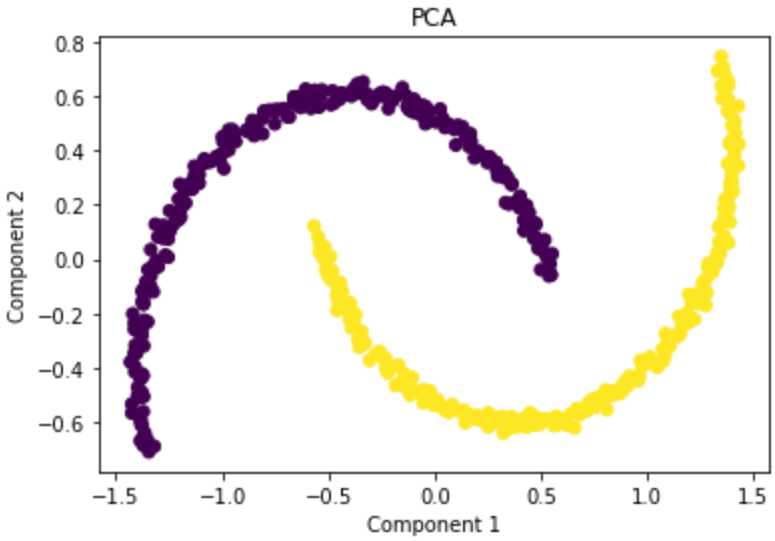


Let's apply PCA on this dataset

In [29]:

```
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()
```

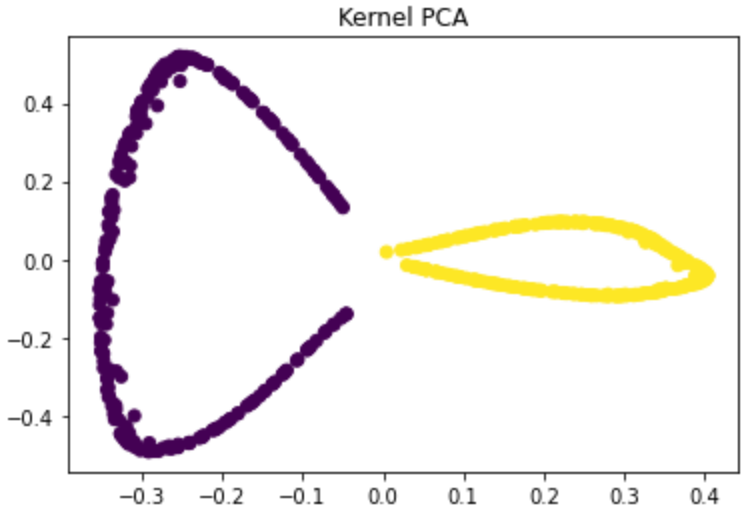


PCA failed to distinguish the two classes

In [30]:

```
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

KernelPCA exercises

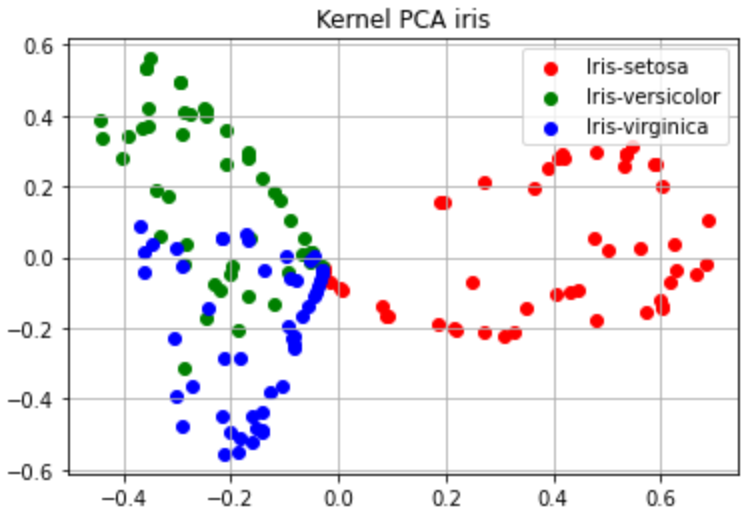
- Visualize in 2d datasets used in this labs, experiment with the parameters of the KernelPCA method change kernel and gamma params. Docs: https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.KernelPCA.html

In [31]:

```
# TODO, iris
from sklearn.decomposition import KernelPCA
kpca = KernelPCA(kernel ='rbf', gamma = 2.05)
# kpca = KernelPCA(kernel ='poly', gamma = 2.9)
X_kpca = kpca.fit_transform(x_iris)

plt.title("Kernel PCA iris")
iris_targets = ['Iris-setosa', 'Iris-versicolor', 'Iris-virginica']
colors = ['r', 'g', 'b']
for target, color in zip(iris_targets, colors):
    indicesToKeep = df_iris['target'] == target
    plt.scatter(X_kpca[indicesToKeep, 0], X_kpca[indicesToKeep, 1], c = color)

plt.legend(iris_targets)
plt.grid()
plt.show()
```



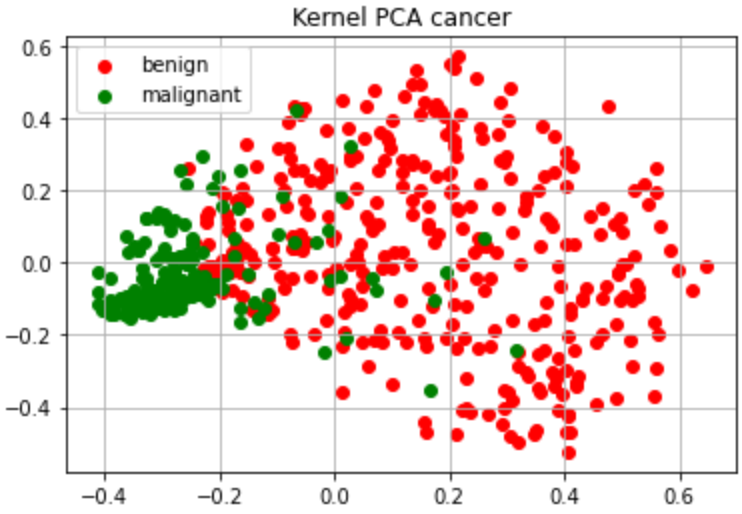
In [32]:

```
# TODO, cancer
kpca = KernelPCA(kernel ='rbf', gamma = 0.1)
```

```
# kpca = KernelPCA(kernel ='poly', gamma = 0.8)
X_kpca = kpca.fit_transform(df_cancer_standardized)

plt.title("Kernel PCA cancer")
colors = ['r', 'g']
for target, color in zip(cancer_target_values, colors):
    indicesToKeep = final_df_cancer[('target',)] == target
    plt.scatter(X_kpca[indicesToKeep, 0], X_kpca[indicesToKeep, 1], c = color)

plt.legend(cancer_target_values)
plt.grid()
plt.show()
```



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.Kernell

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

Visualizing mnist

In [179…

```
import mnist
plt.rcParams["figure.figsize"] = [16, 9]

def tiles(examples, space_between_tiles=2):
    rows_count = examples.shape[0]
    cols_count = examples.shape[1]
    tile_height = examples.shape[2]
    tile_width = examples.shape[3]

    img_height = (tile_height + space_between_tiles) * (rows_count - 1) + tile_height
    img_width = (tile_width + space_between_tiles) * (cols_count - 1) + tile_width
    img_matrix = np.ones(shape=(img_height, img_width))
    for tile_row_idx in range(rows_count):
        for tile_col_idx in range(cols_count):
            start_row_idx = (tile_height + space_between_tiles) * tile_row_idx
            end_row_idx = start_row_idx + tile_height
            start_col_idx = (tile_width + space_between_tiles) * tile_col_idx
            end_col_idx = start_col_idx + tile_width
            img_matrix[start_row_idx:end_row_idx, start_col_idx:end_col_idx] = examples[ti

    return img_matrix
```

In [180…

```
digits = np.reshape(mnist.train_images()[:12*24], newshape=(12, 24, 28, 28))
img = tiles(digits)

plt.matshow(img, cmap='gray', interpolation='none')
plt.axis('off')
plt.show()
```

In [108…

```
X = mnist.train_images().astype(np.float32) / 255.0
y = mnist.train_labels()
```

```
_, img_height, img_width = X.shape
X = X.reshape(-1, img_height * img_width)

# limit the size of data
SAMPLES_LIMIT = 2000
X = X[:SAMPLES_LIMIT]
y = y[:SAMPLES_LIMIT]
```

In [109…

```
def plot_2d_mnist_scatter(X, y, title):
    fig, plot = plt.subplots()
    fig.set_size_inches(16, 16)
    plt.prism()

    for i in range(10):
        digit_indices = (y == i)
        dim1 = X[digit_indices][:, 0]
        dim2 = X[digit_indices][:, 1]
        plot.scatter(dim1, dim2)

    plot.set_xticks(())
    plot.set_yticks(())

    plt.tight_layout()
    plt.legend(labels = [i for i in range(10)])
    plt.title(title, fontsize=36)
    plt.show()


def draw_explained_variance(X, pca, title):
    pca.fit_transform(X)

    fig, ax = plt.subplots()
    ax.set_xlabel('# Components', fontsize = 15)
    ax.set_ylabel('Explained Variance', fontsize = 15)
    ax.set_title(title, fontsize = 36)
    ax.plot(range(len(pca.explained_variance_)), pca.explained_variance_ratio_)
    ax.grid()
```

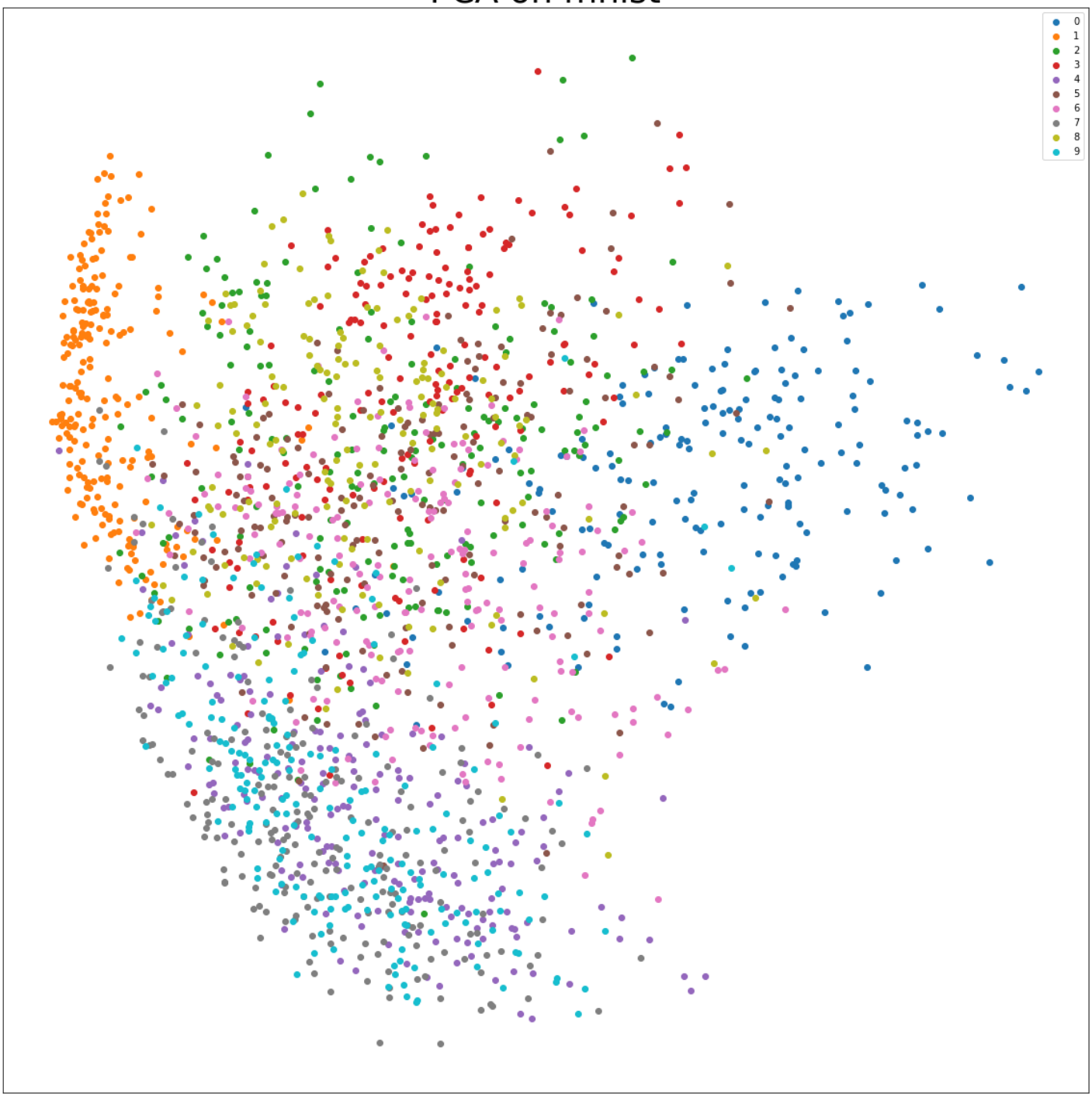
Visualizing PCA

In [58]:

```
pca = PCA()
X_pca_embedded = pca.fit_transform(X, y)
```

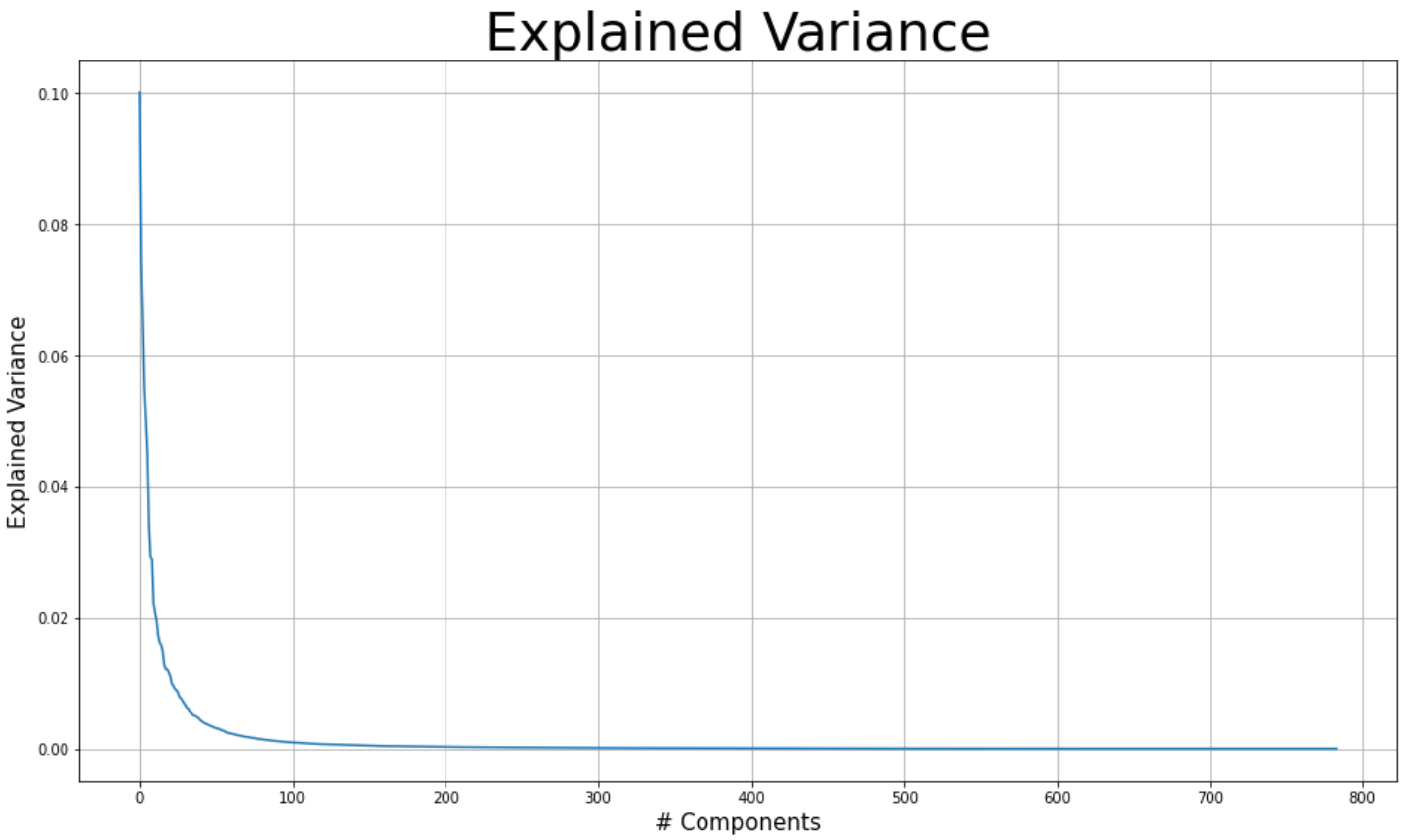
In [59]:

```
plot_2d_mnist_scatter(X_pca_embedded, y, 'PCA on mnist')
```



In [96]:

```
draw_explained_variance(X, pca, 'Explained Variance')
```



In [97]:

```
def find_number_of_components(pca, cumulative_explained_variance):
    return np.argmax(np.cumsum(pca.explained_variance_ratio_) > cumulative_explained_varia

def find_required_components(pca):
    variance = [.5, .85, .9, .95, .99]
    n_comps = []
    for cumulative_explained_variance in variance:
        n_comps.append(find_number_of_components(pca, cumulative_explained_variance))

    df = pd.DataFrame()
    df['explained variance'] = variance
    df['components'] = np.array(n_comps)

    return df
```

In [98]:

```
find_required_components(pca)
```

Out[98]:

	explained variance	components
0	0.50	9
1	0.85	55
2	0.90	81
3	0.95	140
4	0.99	303

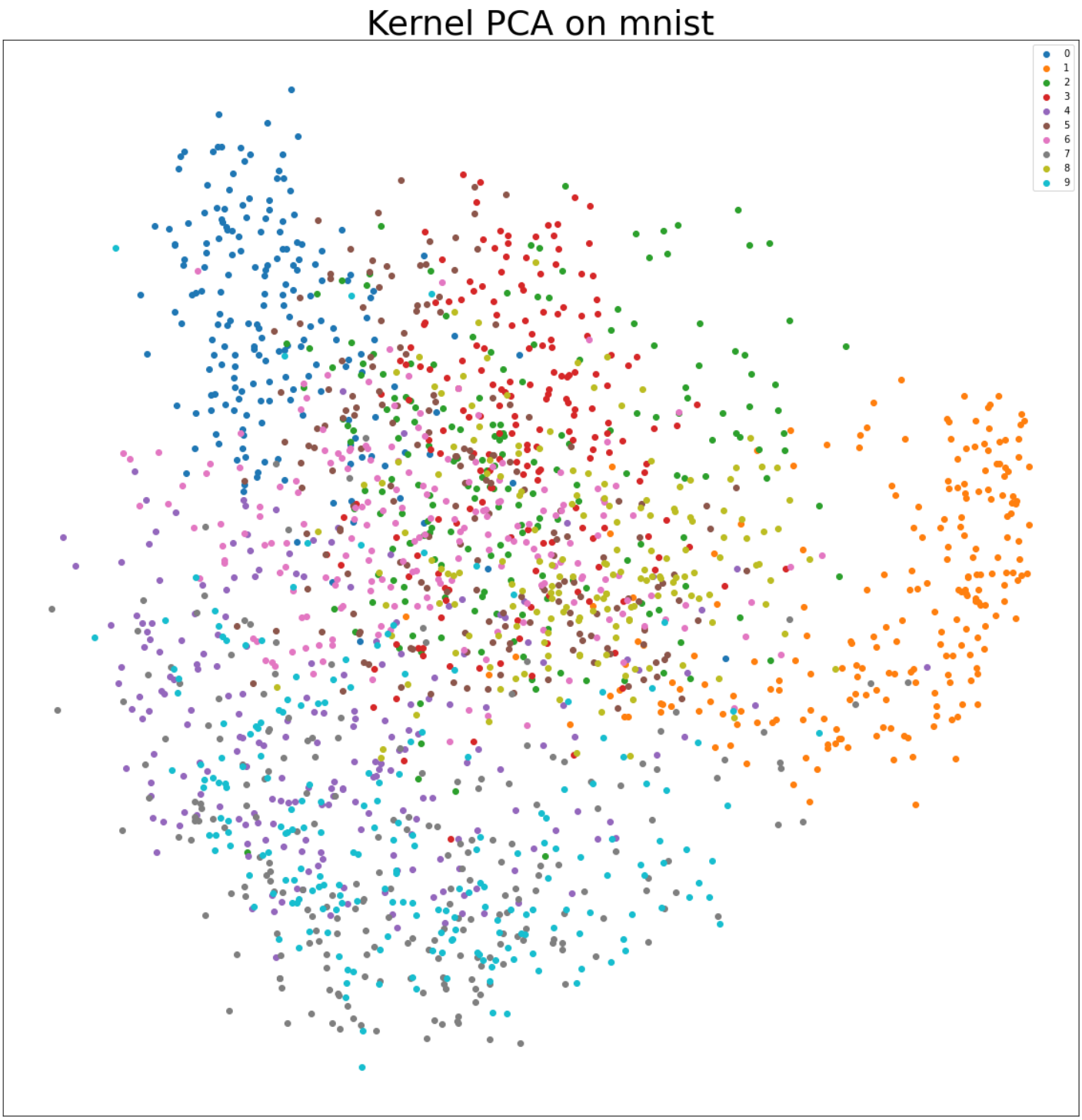
Visualizing Kernel PCA

In [ ]:

```
# kernel_pca = KernelPCA(kernel ='rbf', gamma = 0.03)
kernel_pca = KernelPCA(kernel ='cosine', gamma = 0.5)
X_pca_embedded = kernel_pca.fit_transform(X, y)
```

In [156…

```
plot_2d_mnist_scatter(X_pca_embedded, y, 'Kernel PCA on mnist')
```



Transforming data back and forth

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_tran
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.

In [181…

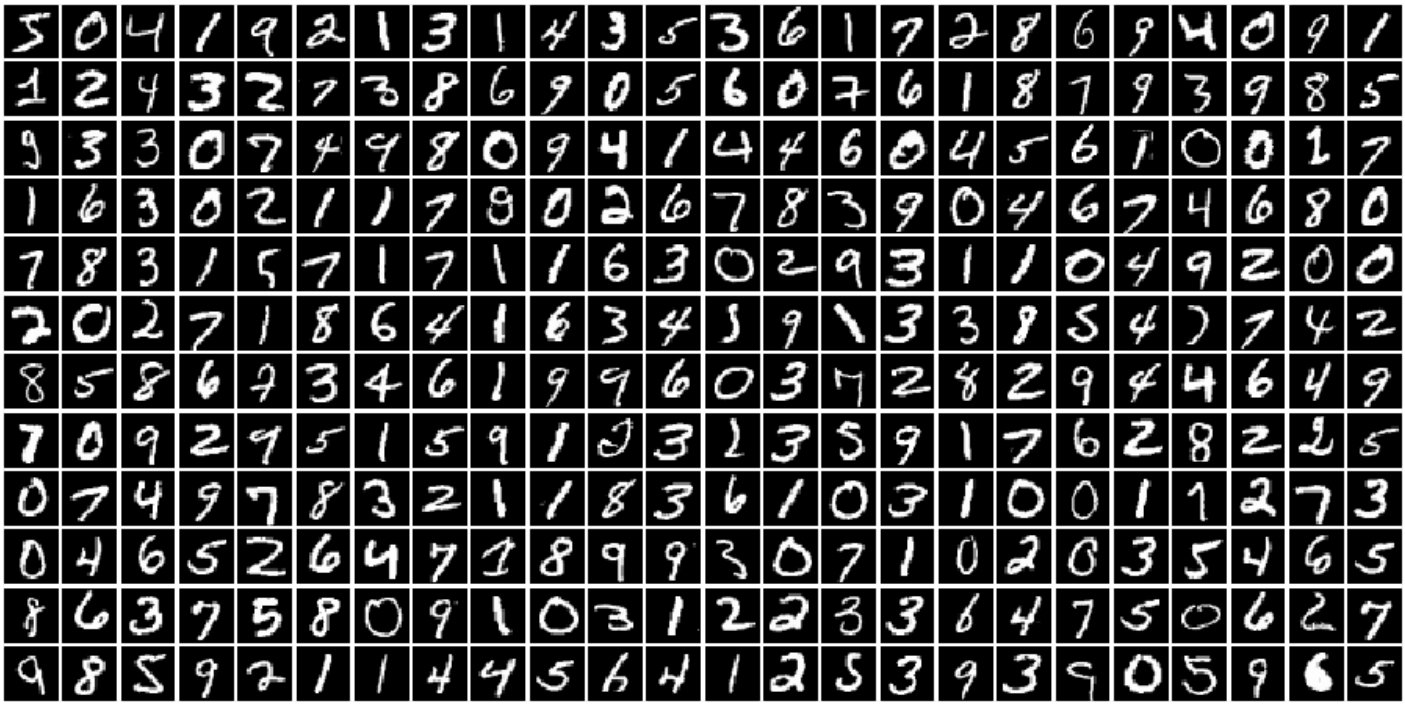
```
def draw_sample_digits(X):
    SAMPLE_SIZE = 12 * 24
```

```
    examples = np.reshape(X[:SAMPLE_SIZE], newshape=(12, 24, 28, 28))

    img = tiles(examples)
    plt.matshow(img, cmap='gray', interpolation='none')
    plt.axis('off')
    plt.show()
```

In [182…

```
draw_sample_digits(X)
```

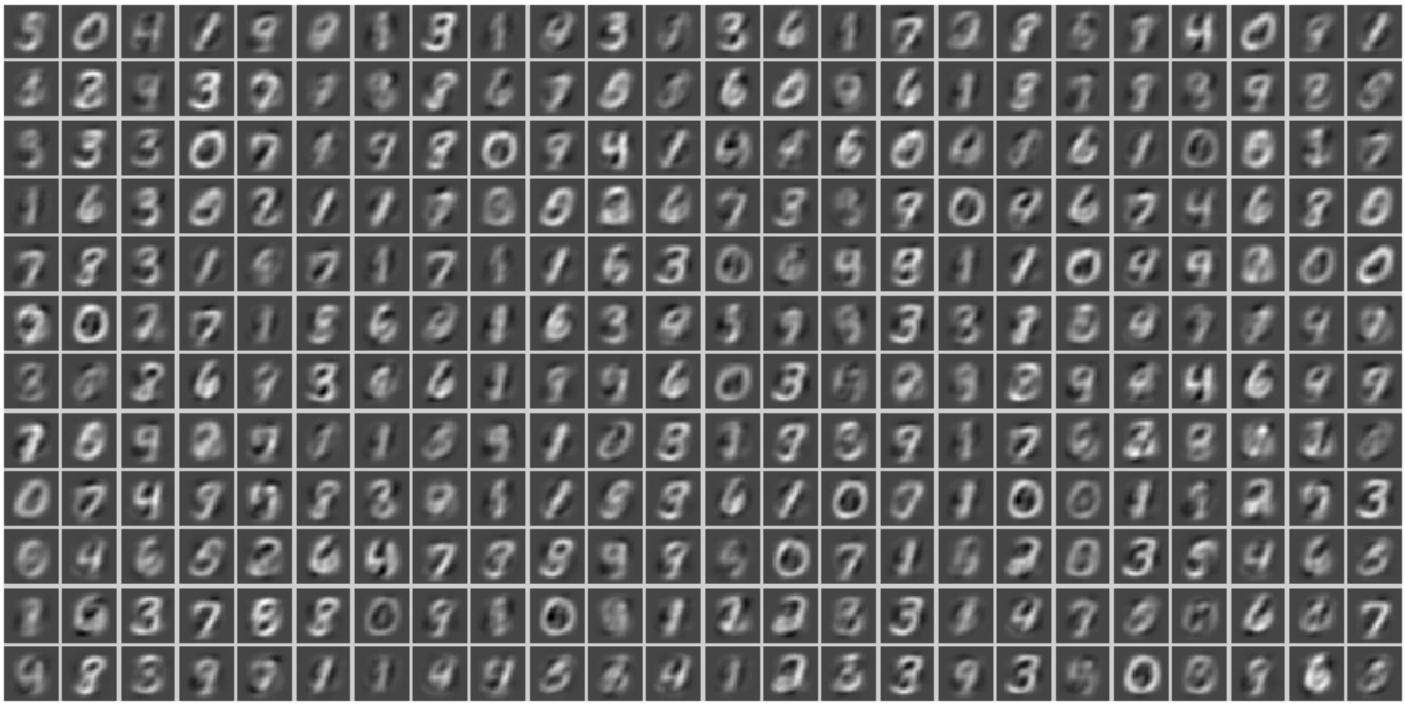


In [188…

```
def draw_digits_for_n_components(X, n_comp):
    pca_n_comp = PCA(n_components=n_comp)
    X_pca_embedded_n_comp = pca_n_comp.fit_transform(X)
    X_transformed = pca_n_comp.inverse_transform(X_pca_embedded_n_comp)
    draw_sample_digits(X_transformed)
```

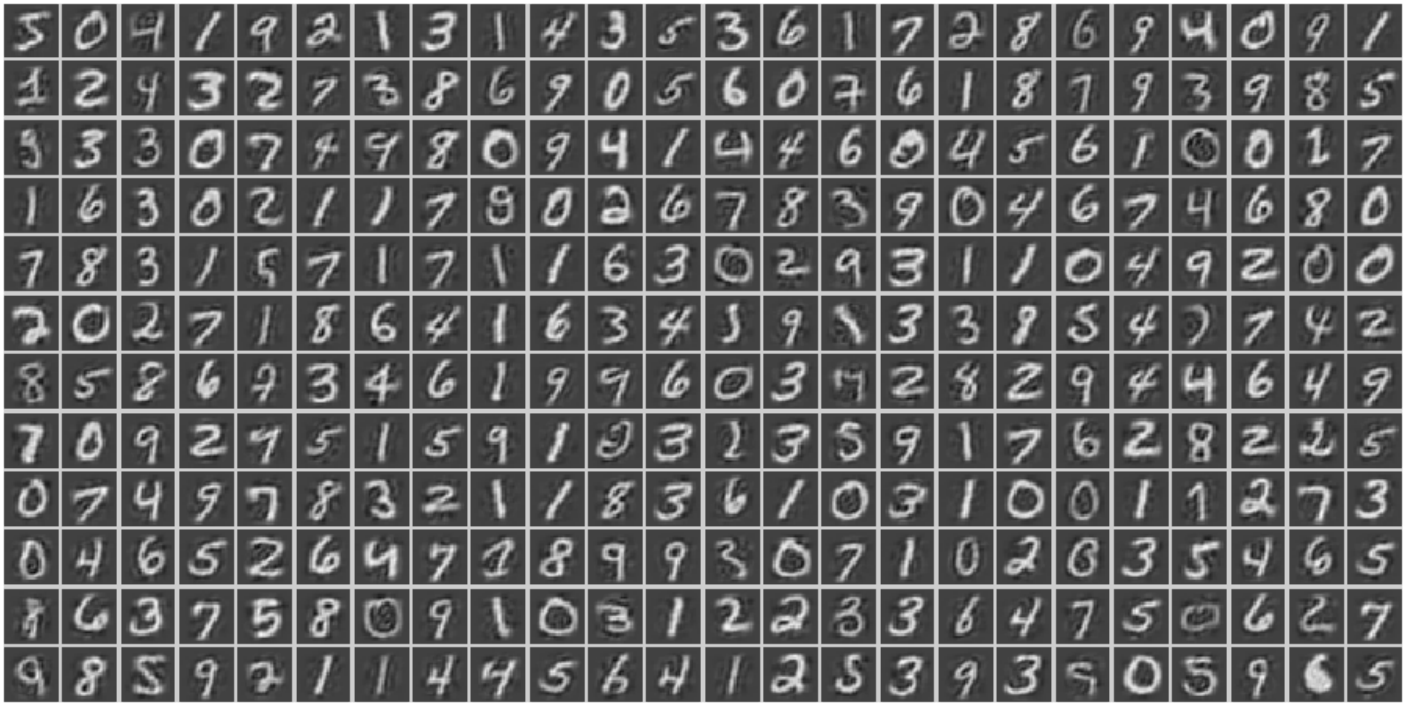
In [190…

```
draw_digits_for_n_components(X, 9)
```



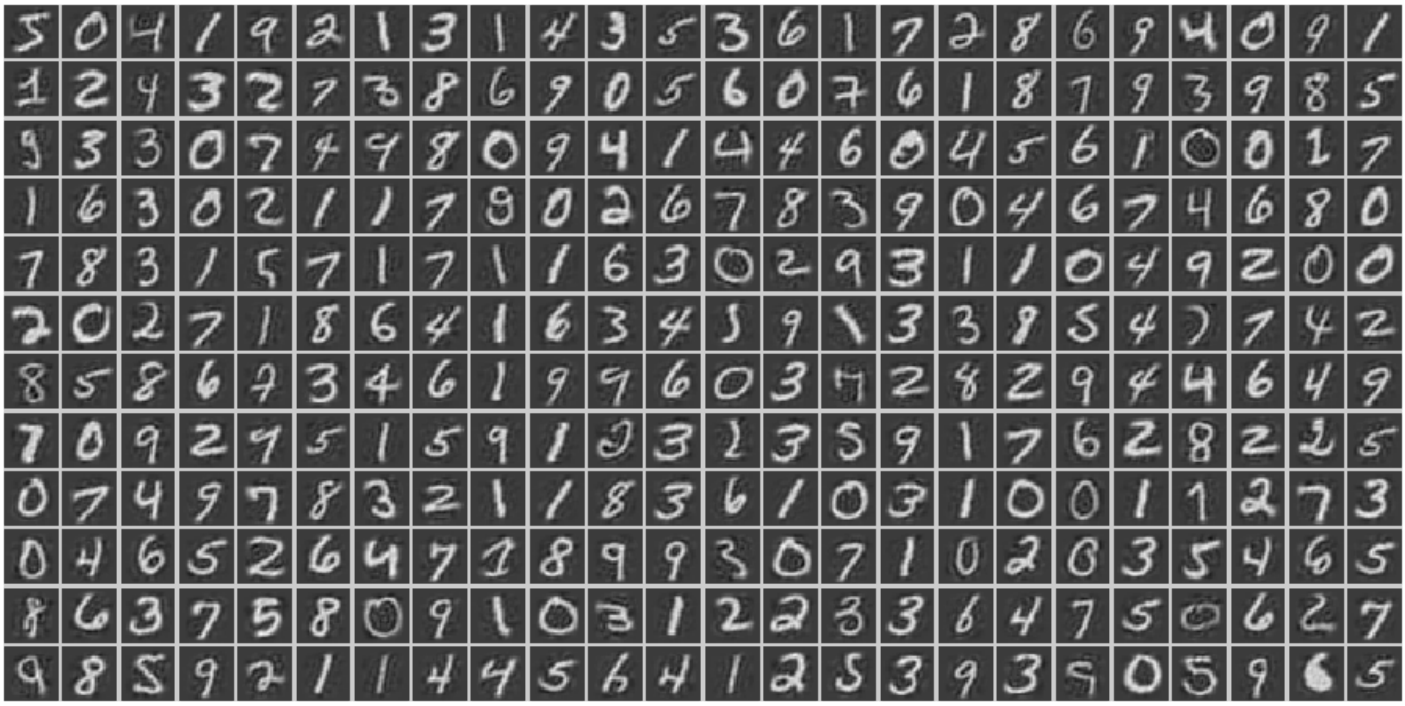
In [191…

```
draw_digits_for_n_components(X, 55)
```



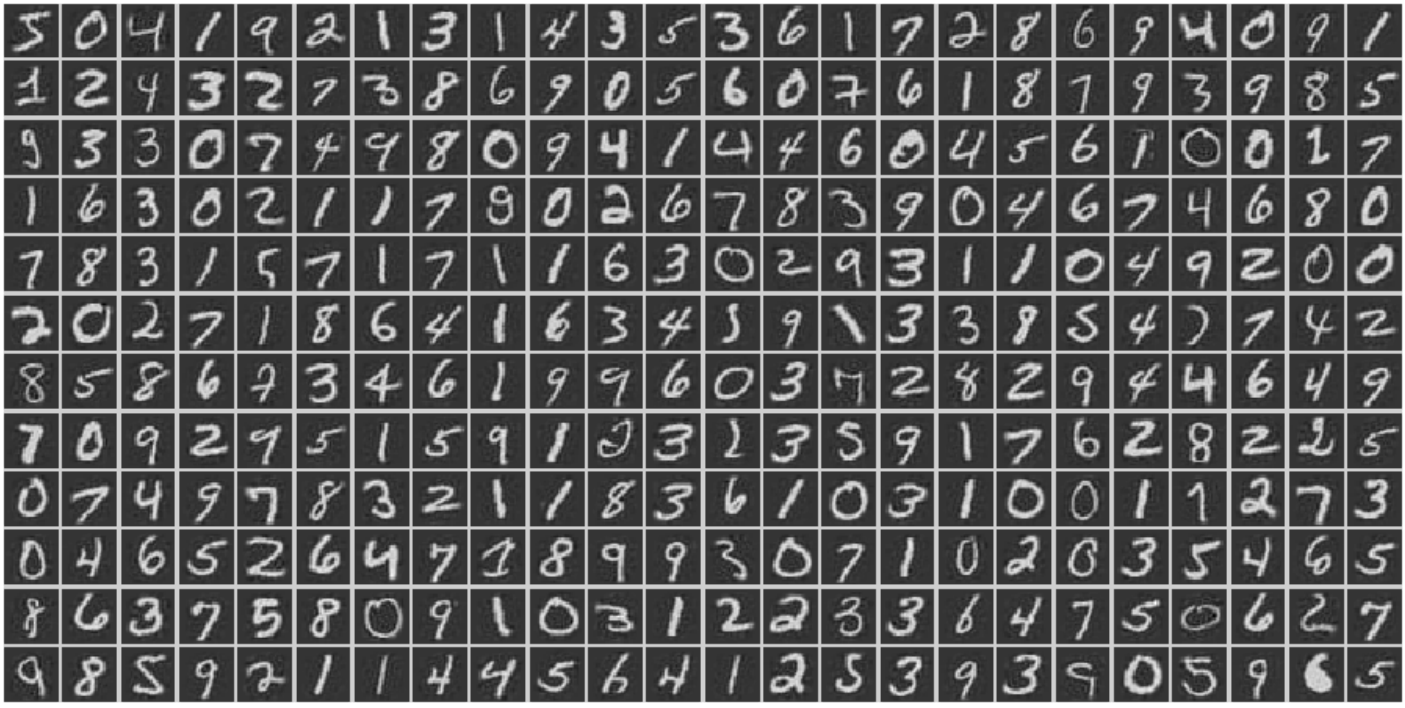
In [192…

```
draw_digits_for_n_components(X, 81)
```



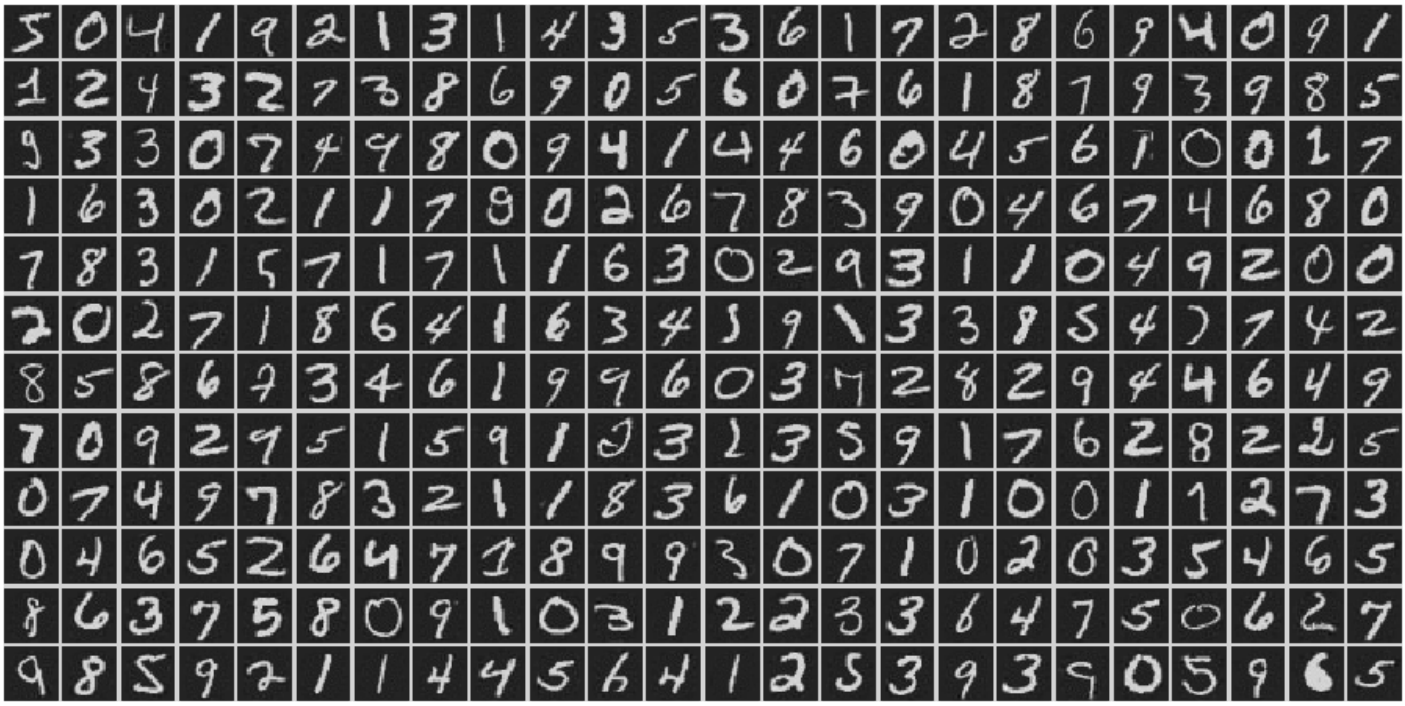
In [196…

```
draw_digits_for_n_components(X, 140)
```



In [195…

```
draw_digits_for_n_components(X, 303)
```



We can see that the number of principal components affects sharpness of our data but using only a couple of components we can get pretty good data approximation - using much fewer data! At some point increasing number of principal components doesn't visibly increase data quality. This means we could use PCA for lossy data compression.

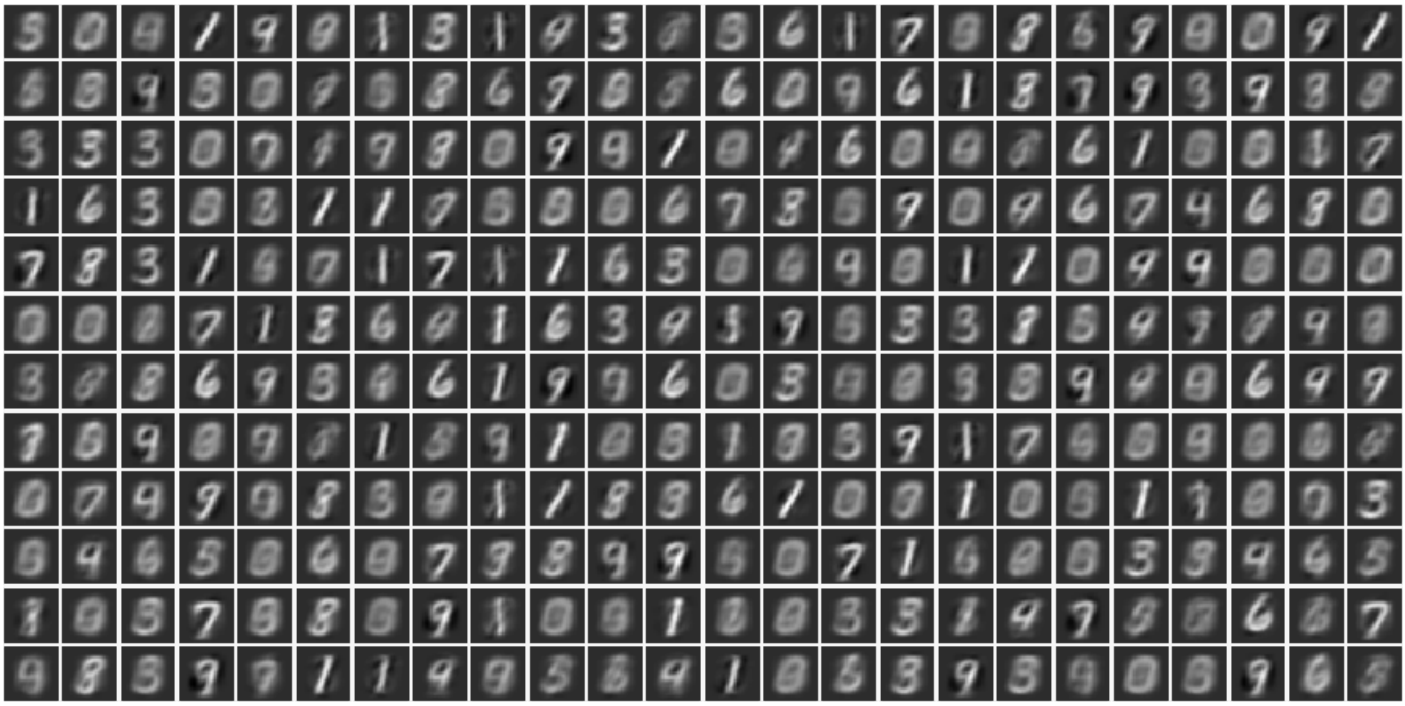
Reconstructing data using Kernel PCA

In [207…

```
def draw_digits_kernel_pca(X, n_comp):
    pca_n_comp = KernelPCA(kernel ='rbf', gamma = 0.03, n_components=n_comp, fit_inverse_t
    # pca_n_comp = KernelPCA(kernel ='cosine', gamma = 0.5, n_components=n_comp, fit_inve
    X_pca_embedded_n_comp = pca_n_comp.fit_transform(X)
    X_transformed = pca_n_comp.inverse_transform(X_pca_embedded_n_comp)
    draw_sample_digits(X_transformed)
```

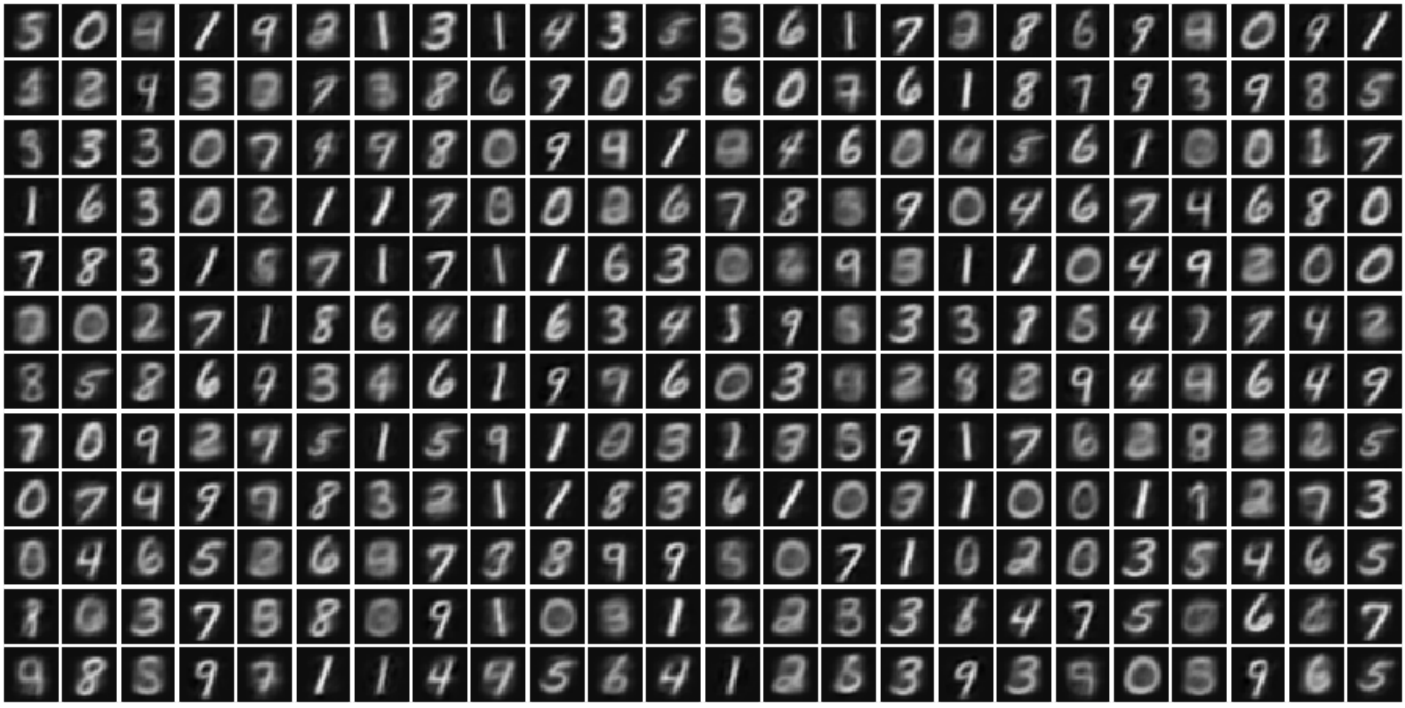
In [208…

```
draw_digits_kernel_pca(X, 9)
```



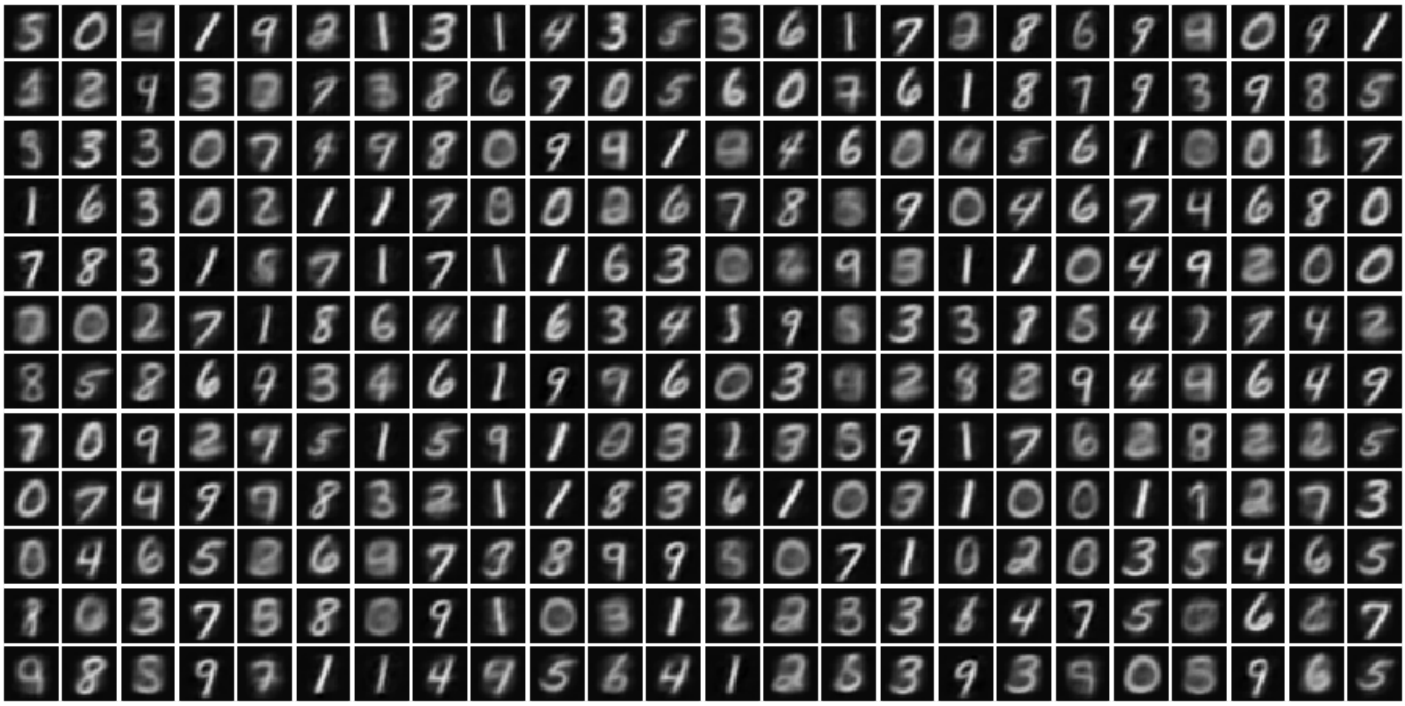
In [209…

```
draw_digits_kernel_pca(X, 55)
```



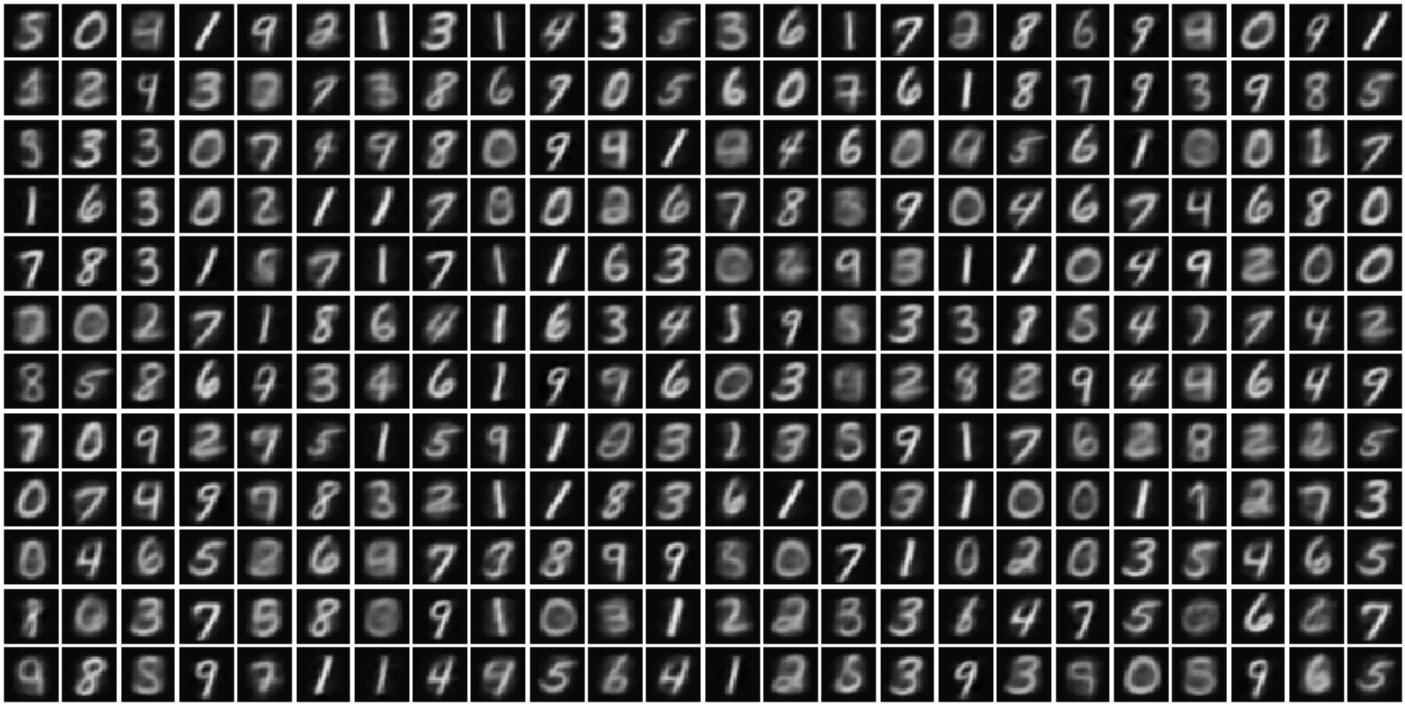
In [210…

```
draw_digits_kernel_pca(X, 81)
```



In [211…

```
draw_digits_kernel_pca(X, 148)
```



In [212…

```
draw_digits_kernel_pca(X, 303)
```