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 [1]:
```
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

import warnings
warnings.filterwarnings("ignore")
```

In [2]:
```
%%javascript
IPython.OutputArea.prototype._should_scroll = function(lines) {
    return false;
}
```

In [3]:
```
iris_url = "https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.dat
```

In [4]:
```
# loading dataset into Pandas DataFrame
df_iris = pd.read_csv(iris_url, names=['sepal length', 'sepal width', 'petal lengt
```

In [5]:
```
df_iris.head(15)
```

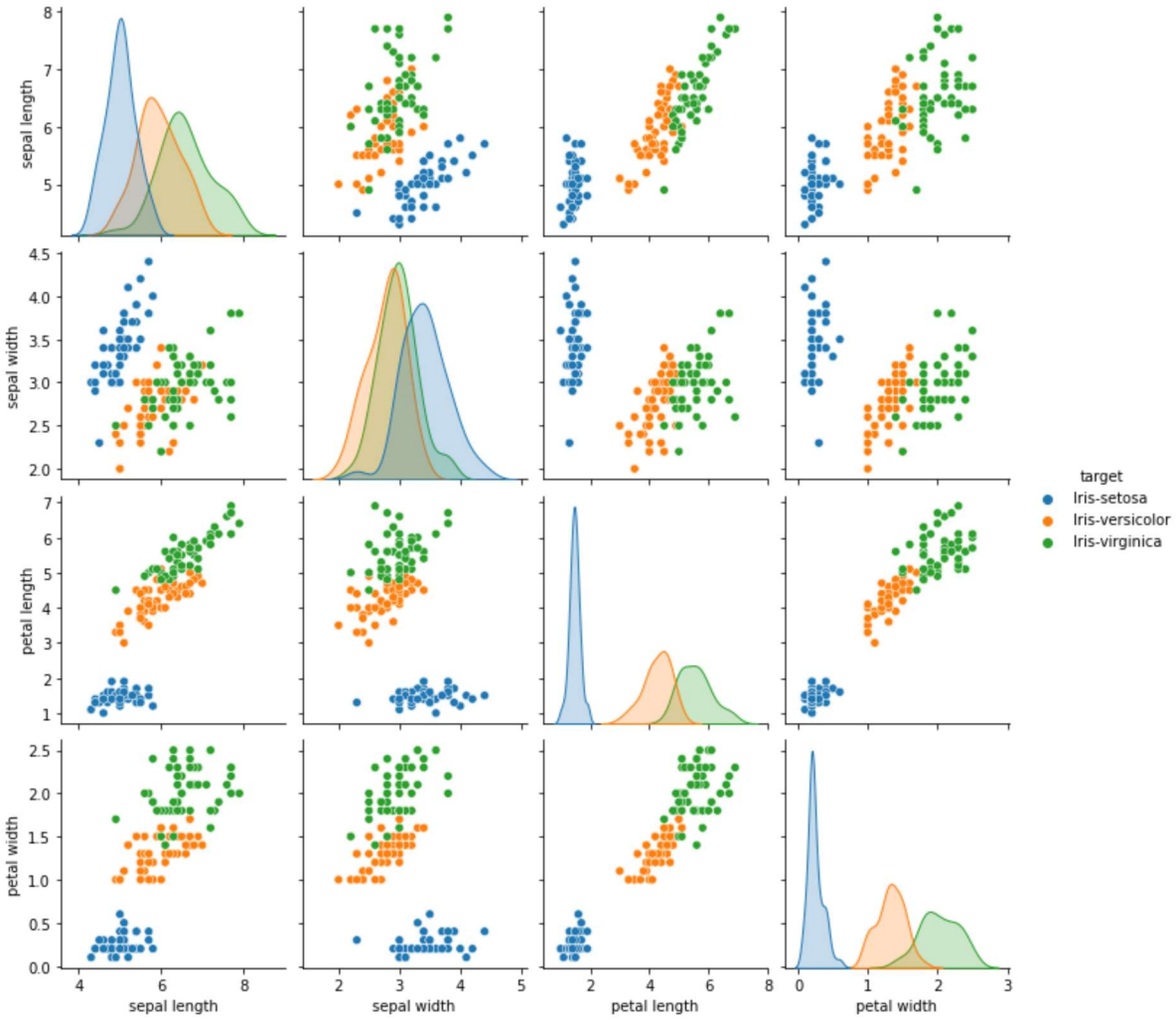
Out[5]:

	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

In [6]:
```
sns.pairplot(df_iris, hue='target')
```

Out[6]:
```
<seaborn.axisgrid.PairGrid at 0x7f3565470e50>
```



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 [7]:
```
features_iris = ['sepal length', 'sepal width', 'petal length', 'petal width']
x_iris = df_iris.loc[:, features_iris].values
```

In [8]:
```
y_iris = df_iris.loc[:, ['target']].values
```

In [9]:
```
x_iris = StandardScaler().fit_transform(x_iris)
```

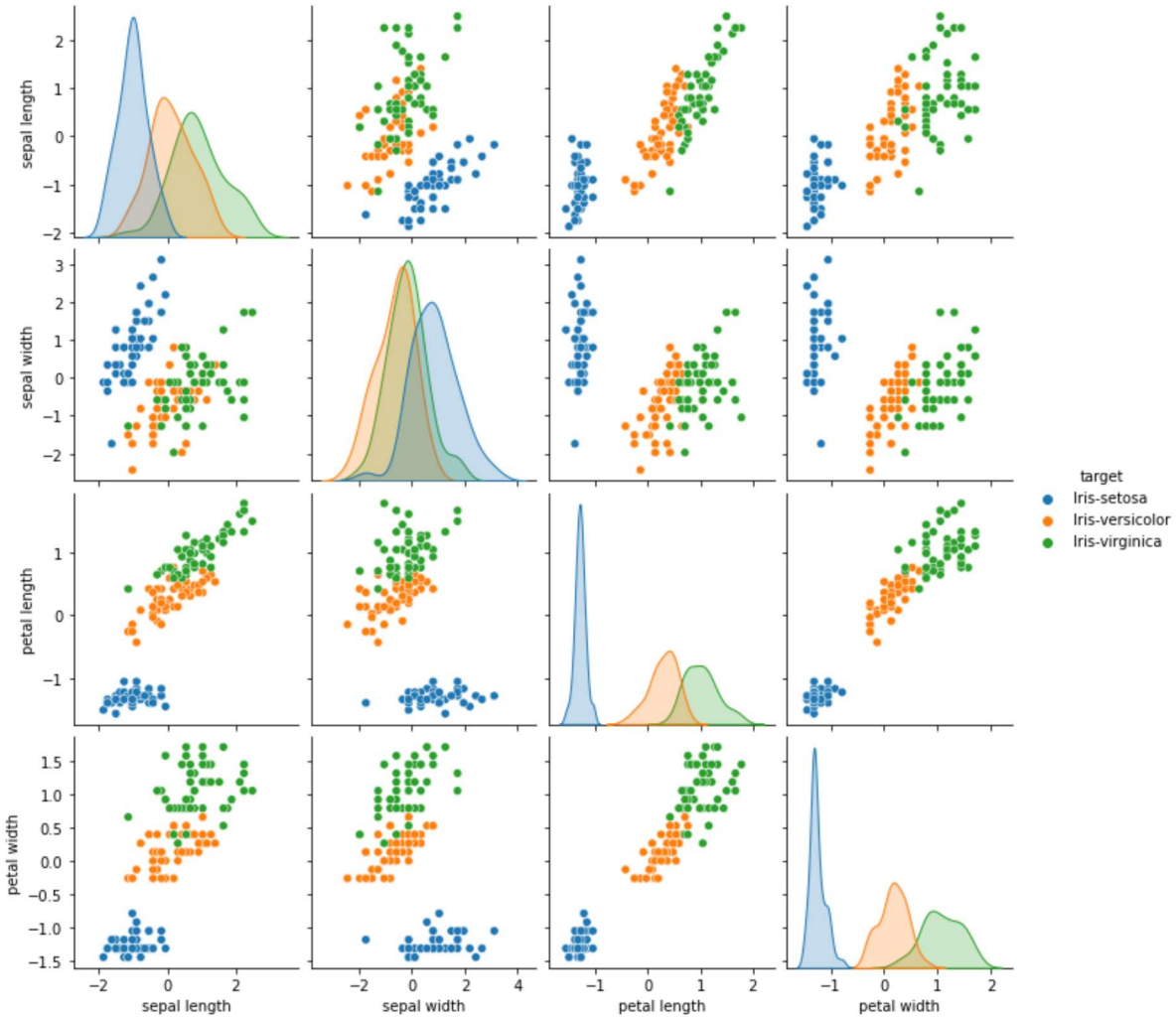
In [10]:
```
df_iris_standarize = pd.DataFrame(data=x_iris, columns=features_iris)
df_iris_standarize['target'] = df_iris['target']
df_iris_standarize.head(15)
```

Out[10]:

	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 [11]:
```
sns.pairplot(df_iris_standarize, hue='target')
```

Out[11]:
```
<seaborn.axisgrid.PairGrid at 0x7f356070f640>
```



We can see that the distributions are now standardized

PCA Projection to 2D

In [12]:
```
pca_iris = PCA(n_components=2)
```

In [13]:
```
principalComponents_iris = pca_iris.fit_transform(x_iris)
```

In [14]:
```
principalDf_iris = pd.DataFrame(data=principalComponents_iris,
                                columns=['principal component 1', 'principal compor
```

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

Out[15]:

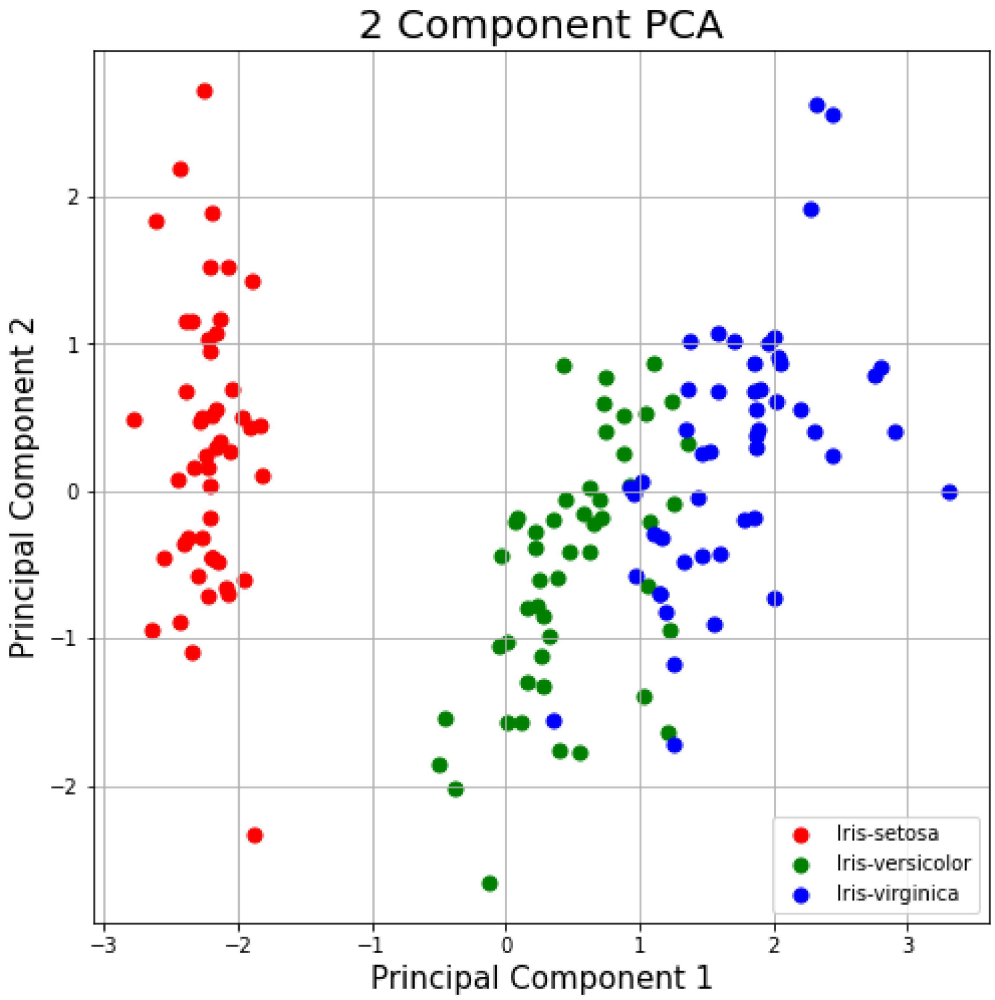
	principal component 1	principal component 2	target
0	-2.264542	0.505704	Iris-setosa
1	-2.086426	-0.655405	Iris-setosa
2	-2.367950	-0.318477	Iris-setosa
3	-2.304197	-0.575368	Iris-setosa
4	-2.388777	0.674767	Iris-setosa
5	-2.070537	1.518549	Iris-setosa
6	-2.445711	0.074563	Iris-setosa
7	-2.233842	0.247614	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

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 [16]:

```
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 [17]:
```
pca_iris.explained_variance_ratio_
```

Out[17]:
```
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 [18]:
```
from sklearn.datasets import load_breast_cancer
```

Załadowanie zbioru danych breast_cancer oraz zmapowanie etykiety target.

In [19]:
```
def sklearn_dataset_to_pandas(sklearn_dataset, target_map = None):
    cancer_features = sklearn_dataset['data']
    cancer_features_norm = StandardScaler().fit_transform(cancer_features)

    pandas_df = pd.DataFrame(data=cancer_features_norm, columns=sklearn_dataset['f
    pandas_df['target'] = pd.Series(sklearn_dataset['target'])

    if target_map is not None:
        pandas_df['target'] = pandas_df['target'].map(lambda t: target_map[t])

    return pandas_df
```

In [20]:
```
cancer_dataset = load_breast_cancer()
cancer_target_map = {0: 'benign', 1: 'malignant'}
cancer_df = sklearn_dataset_to_pandas(cancer_dataset, cancer_target_map)
cancer_df.head(5)
```

Out[20]:

	mean radius	mean texture	mean perimeter	mean area	mean smoothness	mean compactness	mean concavity	mean concave points	syr
0	1.097064	-2.073335	1.269934	0.984375	1.568466	3.283515	2.652874	2.532475	2
1	1.829821	-0.353632	1.685955	1.908708	-0.826962	-0.487072	-0.023846	0.548144	0
2	1.579888	0.456187	1.566503	1.558884	0.942210	1.052926	1.363478	2.037231	0
3	-0.768909	0.253732	-0.592687	-0.764464	3.283553	3.402909	1.915897	1.451707	2
4	1.750297	-1.151816	1.776573	1.826229	0.280372	0.539340	1.371011	1.428493	-0

5 rows × 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 ())

In [21]:
```
cancer_df.corr()
```

Out[21]:

	mean radius	mean texture	mean perimeter	mean area	mean smoothness	mean compactness	mean concavity	c
mean radius	1.000000	0.323782	0.997855	0.987357	0.170581	0.506124	0.676764	0.
mean texture	0.323782	1.000000	0.329533	0.321086	-0.023389	0.236702	0.302418	0.
mean perimeter	0.997855	0.329533	1.000000	0.986507	0.207278	0.556936	0.716136	0.
mean area	0.987357	0.321086	0.986507	1.000000	0.177028	0.498502	0.685983	0.
mean smoothness	0.170581	-0.023389	0.207278	0.177028	1.000000	0.659123	0.521984	0.
mean compactness	0.506124	0.236702	0.556936	0.498502	0.659123	1.000000	0.883121	0.
mean concavity	0.676764	0.302418	0.716136	0.685983	0.521984	0.883121	1.000000	0.
mean concave points	0.822529	0.293464	0.850977	0.823269	0.553695	0.831135	0.921391	1.
mean symmetry	0.147741	0.071401	0.183027	0.151293	0.557775	0.602641	0.500667	0.
mean fractal dimension	-0.311631	-0.076437	-0.261477	-0.283110	0.584792	0.565369	0.336783	0.
radius error	0.679090	0.275869	0.691765	0.732562	0.301467	0.497473	0.631925	0.
texture error	-0.097317	0.386358	-0.086761	-0.066280	0.068406	0.046205	0.076218	0.
perimeter error	0.674172	0.281673	0.693135	0.726628	0.296092	0.548905	0.660391	0.
area error	0.735864	0.259845	0.744983	0.800086	0.246552	0.455653	0.617427	0.
smoothness error	-0.222600	0.006614	-0.202694	-0.166777	0.332375	0.135299	0.098564	0.
compactness error	0.206000	0.191975	0.250744	0.212583	0.318943	0.738722	0.670279	0.
concavity error	0.194204	0.143293	0.228082	0.207660	0.248396	0.570517	0.691270	0.
concave points error	0.376169	0.163851	0.407217	0.372320	0.380676	0.642262	0.683260	0.
symmetry error	-0.104321	0.009127	-0.081629	-0.072497	0.200774	0.229977	0.178009	0.
fractal dimension error	-0.042641	0.054458	-0.005523	-0.019887	0.283607	0.507318	0.449301	0.
worst radius	0.969539	0.352573	0.969476	0.962746	0.213120	0.535315	0.688236	0.
worst texture	0.297008	0.912045	0.303038	0.287489	0.036072	0.248133	0.299879	0.
worst perimeter	0.965137	0.358040	0.970387	0.959120	0.238853	0.590210	0.729565	0.
worst area	0.941082	0.343546	0.941550	0.959213	0.206718	0.509604	0.675987	0.

	mean radius	mean texture	mean perimeter	mean area	mean smoothness	mean compactness	mean concavity	c
worst smoothness	0.119616	0.077503	0.150549	0.123523	0.805324	0.565541	0.448822	0.
worst compactness	0.413463	0.277830	0.455774	0.390410	0.472468	0.865809	0.754968	0.
worst concavity	0.526911	0.301025	0.563879	0.512606	0.434926	0.816275	0.884103	0.
worst concave points	0.744214	0.295316	0.771241	0.722017	0.503053	0.815573	0.861323	0.
worst symmetry	0.163953	0.105008	0.189115	0.143570	0.394309	0.510223	0.409464	0.
worst fractal dimension	0.007066	0.119205	0.051019	0.003738	0.499316	0.687382	0.514930	0.

30 rows × 30 columns

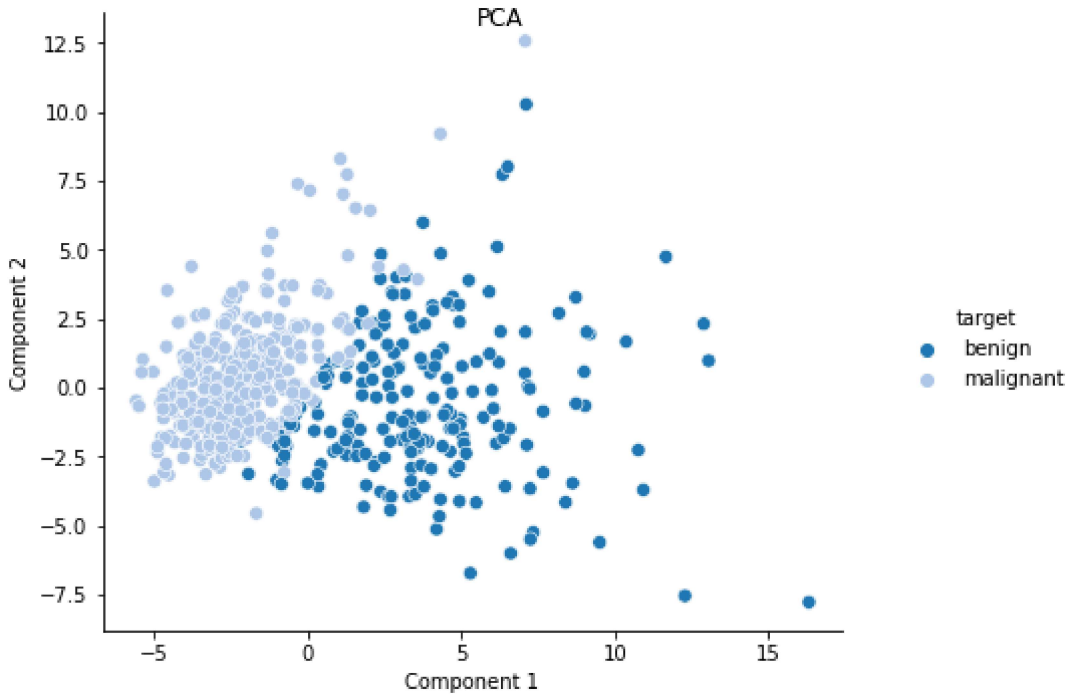
- Perform PCA and visualize the data

Zdefiniowanie metody do redukcji wymiaru danych przy pomocy PCA.

In [22]:
```
def create_pca(n_components, df):
    pca = PCA(n_components=n_components)
    pca_components = pca.fit_transform(df.loc[:, df.columns != 'target'])

    columns_names = ['Component ' + str(i) for i in range(1, n_components+1)]
    pca_df = pd.DataFrame(data=pca_components, columns=columns_names)
    pca_df = pd.concat([pca_df, df['target']], axis=1)

    return pca_df, pca
```

In [23]:
```
def visualize_pca(df, fig_size=(8, 5)):
    g= sns.relplot(data=df, x='Component 1', y='Component 2', hue=df['target'], pa
    g.figure.set_size_inches(fig_size)
    g.fig.suptitle("PCA")
    sns.despine()
```

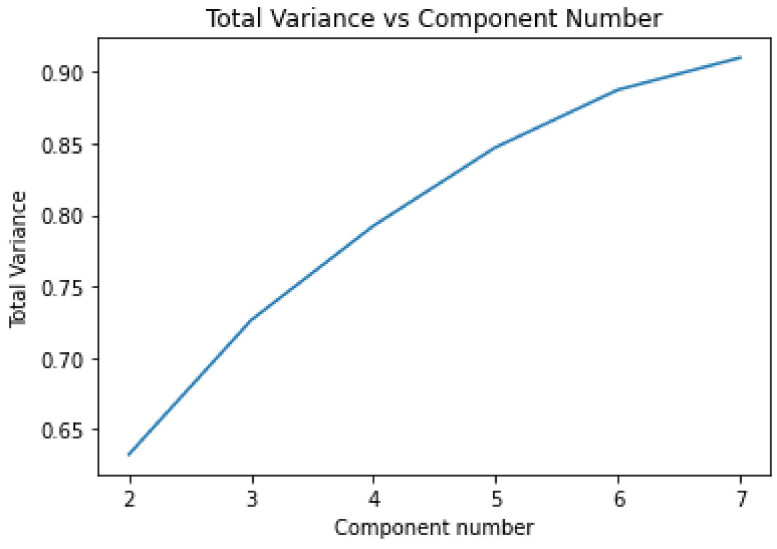
In [24]:
```
pca_cancer_df, pca_cancer = create_pca(2, cancer_df)
visualize_pca(pca_cancer_df)
```



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

In [25]:
```
def compare_pc(df, n=8):
    x, y = [], []
    for i in range(2, n):
        pca_df, pca = create_pca(i, df)
        total_variance = np.sum(pca.explained_variance_ratio_)
        x.append(i)
        y.append(total_variance)

    plt.title("Total Variance vs Component Number")
    plt.xlabel("Component number")
    plt.ylabel("Total Variance")
    plt.plot(x, y)
    plt.show()
```

In [26]:
```
compare_pc(cancer_df)
```



Wraz ze zwiększaniem parametru n_components wzrastała nam całkowita wariancja. Wynika to z faktu, że zwiększa się wielkośc wektora wariancji, a każda z jego składowych zawiera niezerowe wartości. W związku z

tym wraz ze zwiększaniem n_components, do sumy wariancji dochodzą nam nowe składowe. Wartość sumy wariancji dąży do 1.00, każda kolejna składowa w wektore wariancji ma coraz mniejsze wartości.

- 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 [27]:
```
from sklearn.feature_selection import RFE
from sklearn.tree import DecisionTreeClassifier
from operator import itemgetter
```

Z naszych cech wybieramy tylko 15 'najelpszych' przy użyciu algorytmu RFE.

In [28]:
```
rfe = RFE(estimator=DecisionTreeClassifier(), n_features_to_select=15)
rfe.fit(cancer_df[cancer_dataset['feature_names']].values, cancer_df['target'])
```

Out[28]:
```
RFE(estimator=DecisionTreeClassifier(), n_features_to_select=15)
```

In [29]:
```
sorted_predictors_names = []
for x, y in (sorted(zip(rfe.ranking_ , cancer_dataset['feature_names']), key=itemg
    print(x, y)
    sorted_predictors_names.append(y)
```

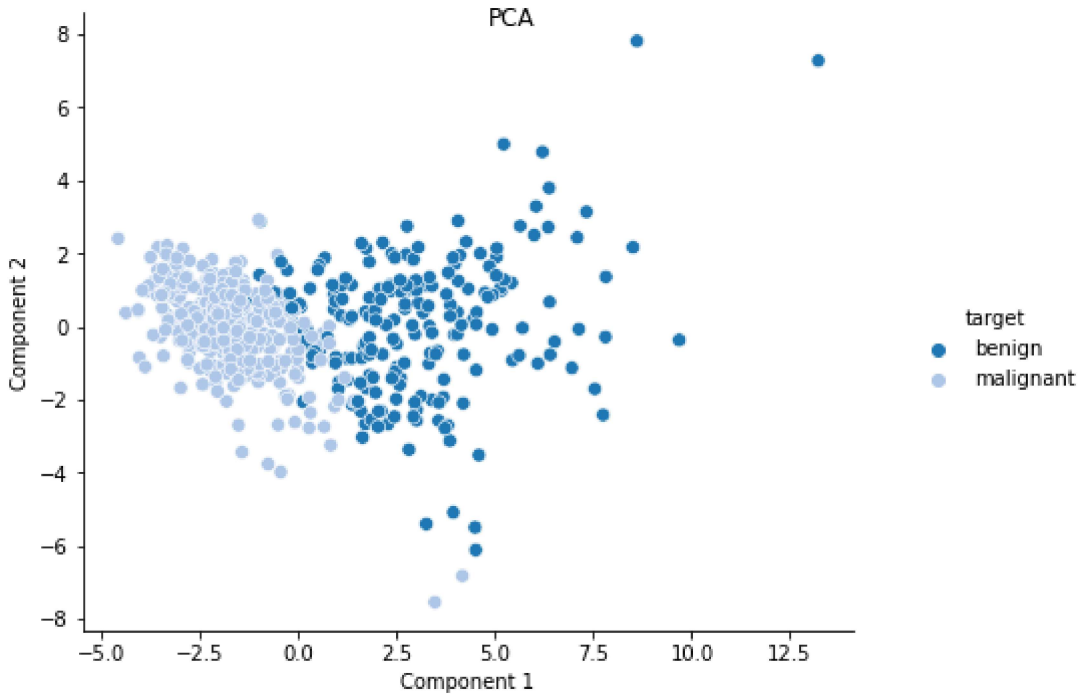
```
1 mean concave points
1 perimeter error
1 area error
1 smoothness error
1 concavity error
1 concave points error
1 worst radius
1 worst texture
1 worst perimeter
1 worst area
1 worst smoothness
1 worst compactness
1 worst concavity
1 worst concave points
1 worst symmetry
2 texture error
3 mean fractal dimension
4 mean symmetry
5 fractal dimension error
6 mean concavity
7 worst fractal dimension
8 symmetry error
9 mean compactness
10 mean smoothness
11 mean area
12 mean perimeter
13 mean texture
14 radius error
15 compactness error
16 mean radius
```

In [30]:
```
cancer_predictors = rfe.transform(cancer_df[cancer_dataset['feature_names']].value

cancer_df_reduce = pd.DataFrame(data=cancer_predictors, columns=sorted_predictors_
cancer_df_reduce['target'] = cancer_df['target']
cancer_df_reduce.head(5)
```

Out[30]:

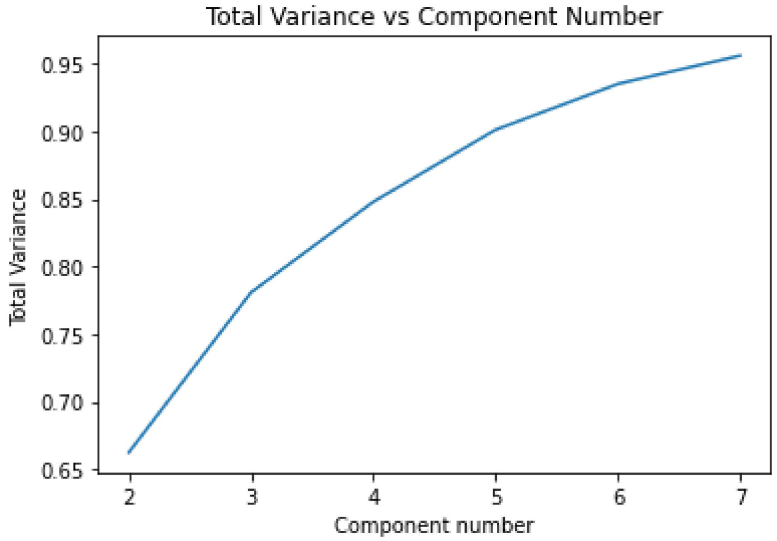
	mean concave points	perimeter error	area error	smoothness error	concavity error	concave points error	worst radius	worst texture	wo perime
0	2.532475	2.833031	2.487578	-0.214002	0.724026	0.660820	1.886690	-1.359293	2.3036
1	0.548144	0.263327	0.742402	-0.605351	-0.440780	0.260162	1.805927	-0.369203	1.5351
2	2.037231	0.850928	1.181336	-0.297005	0.213076	1.424827	1.511870	-0.023974	1.3474
3	1.451707	0.286593	-0.288378	0.689702	0.819518	1.115007	-0.281464	0.133984	-0.2499
4	1.428493	1.273189	1.190357	1.483067	0.828471	1.144205	1.298575	-1.466770	1.3385

In [31]:
```
pca_cancer_reduce_df, pca_cancer_reduce = create_pca(2, cancer_df_reduce)
visualize_pca(pca_cancer_reduce_df)
```



Podobnie jak w przypadku PCA z wykorzystaniem wszystkich paramnetrów, widać że zbiory 'benign' oraz 'malignant' są od siebie odseparowane.

In [32]:
```
compare_pc(cancer_df_reduce)
```

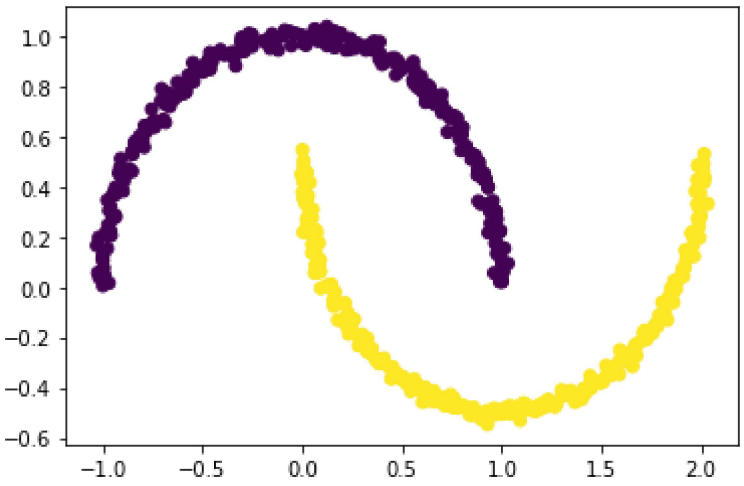


Algorytm PCA uzyskła lepszy efekt w przypadku, gdy skorzystaliśmy z 15 najlepsyzch cech wybranych przez algorytm RFE. W poprzednim przykładzie osiągnieliśmy sumy wariancji na poziomie : (0.6, 0.9), a dla modelu z 15 najlepszymi parametrami jest to : (0.7, 0. 95). Wynika to z faktu, że nie braliśmy pod uwagę cech, ktore są słabowe skorelowane z target.

Kernel PCA

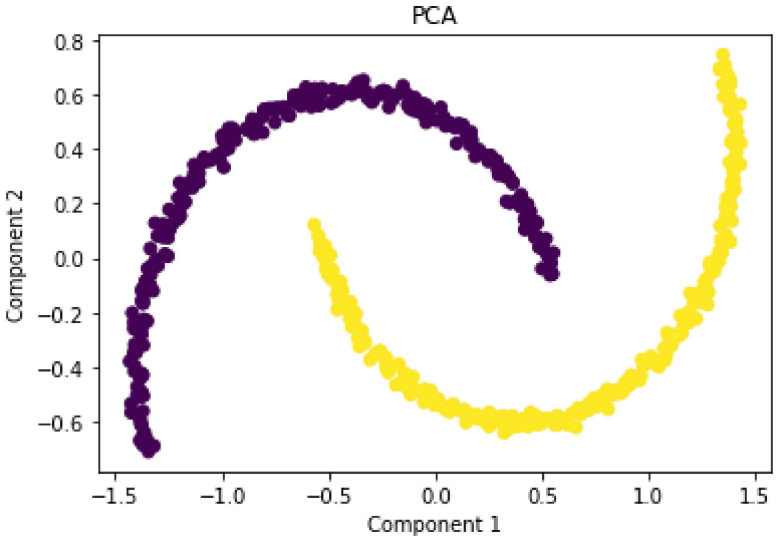
PCA is a linear method. That is it can only be applied to datasets which are linearly separable. It does an excellent job for datasets, which are linearly separable. But, if we use it to non-linear datasets, we might get a result which may not be the optimal dimensionality reduction. Kernel PCA uses 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 [33]:
```
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()
```



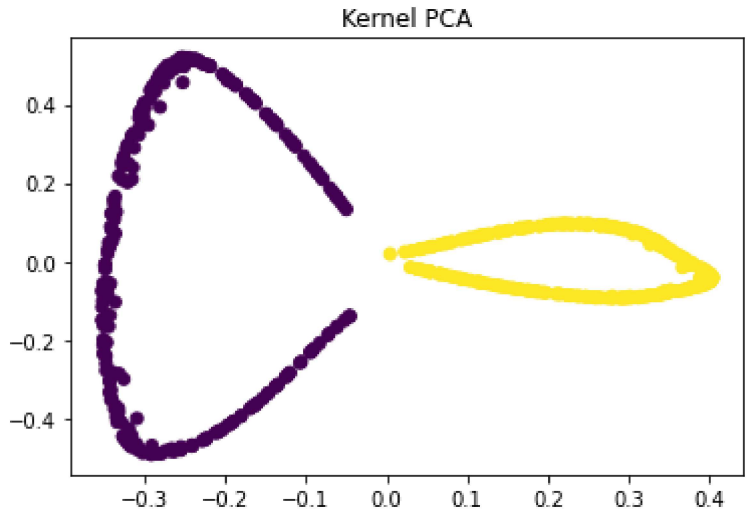
Let's apply PCA on this dataset

In [34]:
```
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 [35]:
```
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

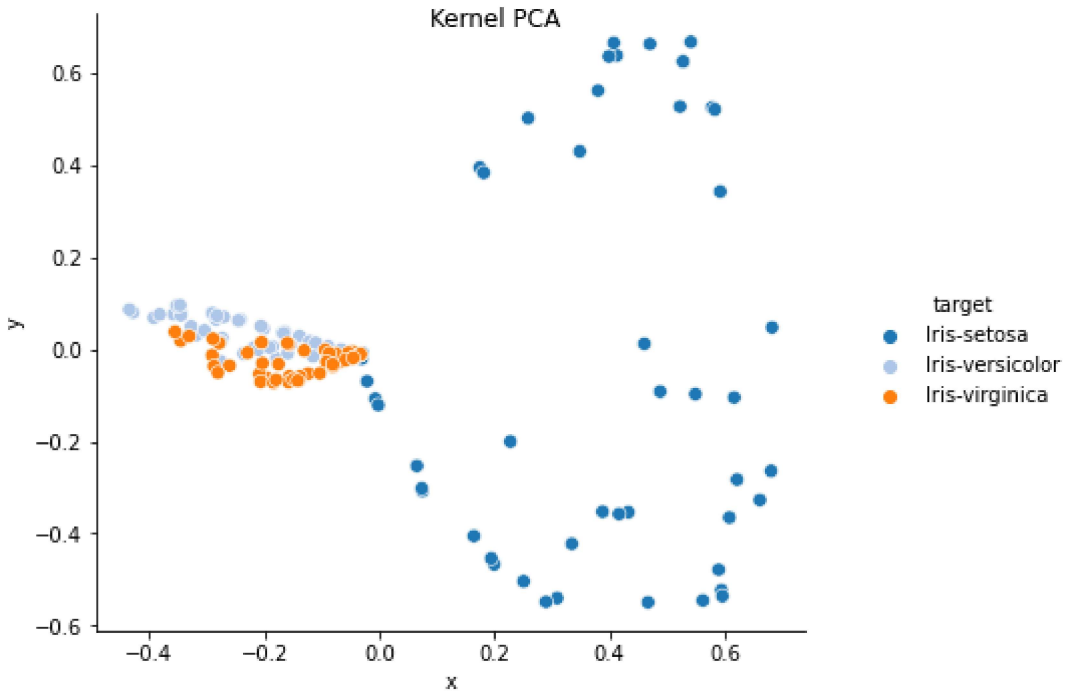
- Visualize in 2d datasets used in this labs, experiment with the parameters of the KernelPCA method change kernel and gamma params. Docs: https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.KernelPCA.html

Zaczniemy od KernelPCA dla Iris Dataset

In [36]:
```
def apply_kernel_pca(df, gamma=15, fig_size=(8,5)):
    features = df.loc[:, df.columns != 'target']
    k_pca = KernelPCA(kernel='rbf', gamma=gamma)
    features_k_pca = k_pca.fit_transform(features)
```

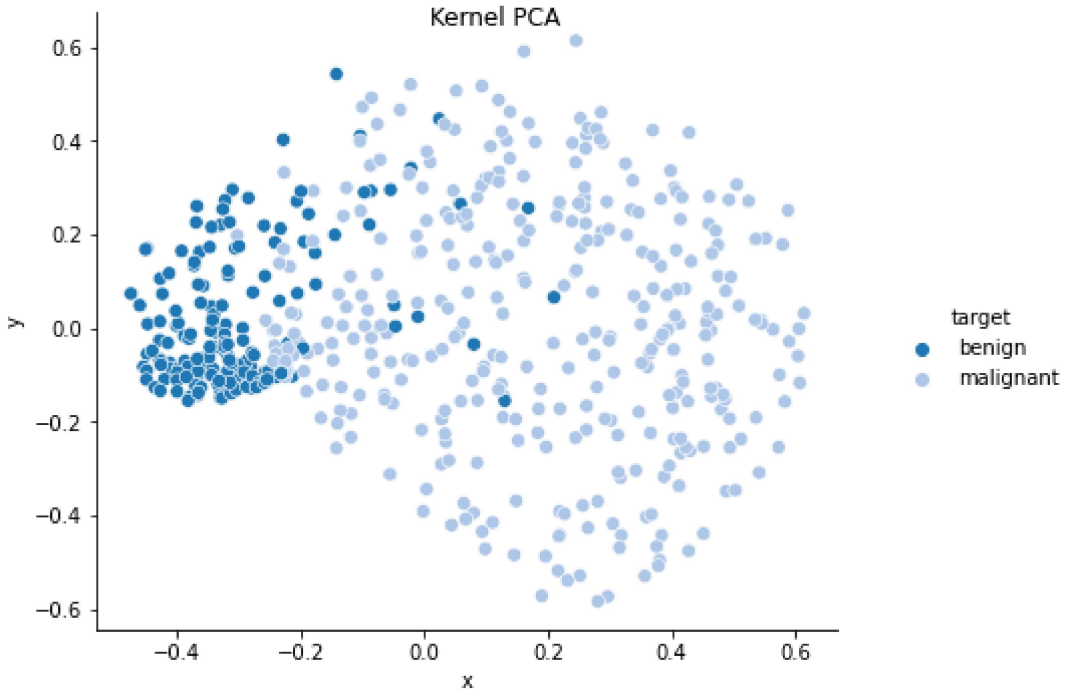
```
    pca_df = pd.DataFrame(data=features_k_pca[:,0:2], columns=['x', 'y'])
    pca_df = pd.concat([pca_df, df['target']], axis=1)

    g= sns.relplot(data=pca_df, x='x', y='y', hue=pca_df['target'], palette='tab20
    g.figure.set_size_inches(fig_size)
    g.fig.suptitle("Kernel PCA")
    sns.despine()
```

In [37]:
```
apply_kernel_pca(df_iris_standarize, gamma=2.2)
```



Podobnie jak w przypadku PCA widzimy, że Iris-setosa jest odseparowany od dwóch pozostałych klas.

In [38]:
```
apply_kernel_pca(cancer_df_reduce, gamma=0.2)
```



Widzimy, że przy pomocy Kernel PCA klasy dla zbioru cancerDataset zostały dobrze odseparowane. Wartości dla notowtorów złośliwych są bardziej rozproszone niż dla nowotworów łagodnych.

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.decompositic 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.decom

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

Przygotowanie zbioru danych.

In [39]:
```
from sklearn.datasets import load_digits
```

In [40]:
```
mnist_dataset = load_digits()
mnist_df = sklearn_dataset_to_pandas(mnist_dataset)
mnist_df.head(5)
```

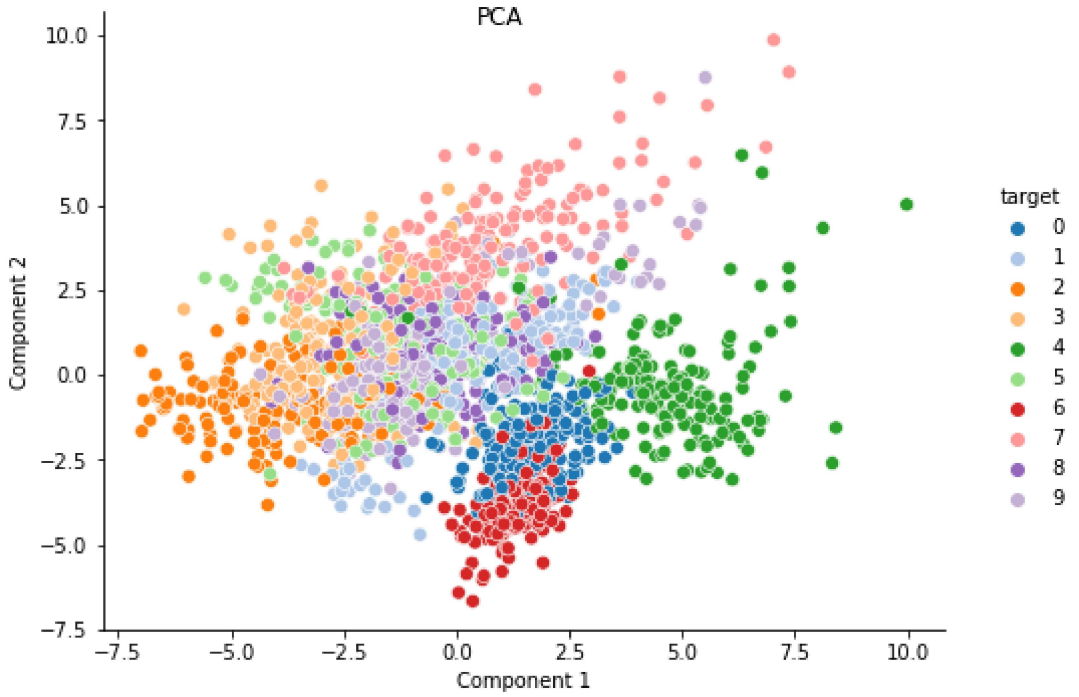
Out[40]:

	pixel_0_0	pixel_0_1	pixel_0_2	pixel_0_3	pixel_0_4	pixel_0_5	pixel_0_6	pixel_0_7	pixel_1_0
0	0.0	-0.335016	-0.043081	0.274072	-0.664478	-0.844129	-0.409724	-0.125023	-0.059078
1	0.0	-0.335016	-1.094937	0.038648	0.268751	-0.138020	-0.409724	-0.125023	-0.059078
2	0.0	-0.335016	-1.094937	-1.844742	0.735366	1.097673	-0.409724	-0.125023	-0.059078
3	0.0	-0.335016	0.377661	0.744919	0.268751	-0.844129	-0.409724	-0.125023	-0.059078
4	0.0	-0.335016	-1.094937	-2.551014	-0.197863	-1.020657	-0.409724	-0.125023	-0.059078

5 rows × 65 columns

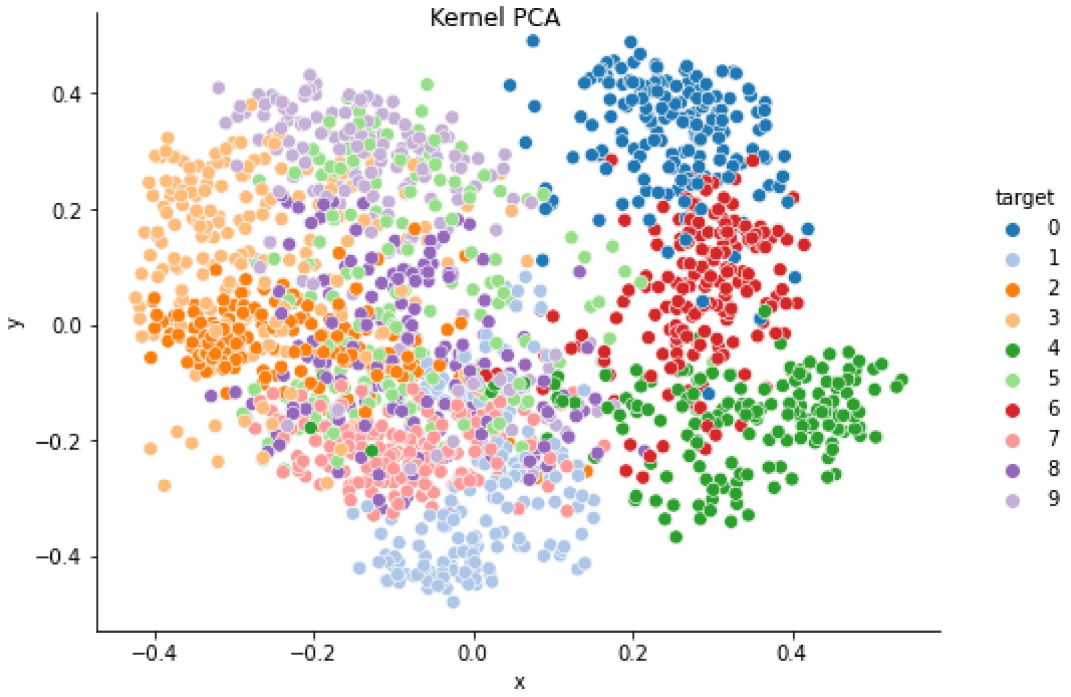
Użycie algorytmu PCA na zbiorze MNIST

In [41]:
```
pca_mnist_df, pca_mnist = create_pca(2, mnist_df)
visualize_pca(pca_mnist_df)
```



Ogólnie rzecz biorąc to ciężko dopatrzeć się jakiejś separacji danych klas. Napewno można dostrzec, ża najbardziej odseparowane od pozostałych sa klasa 4 oraz 6.Użycie algorytmu Kernel PCA na zbiorze MNIST

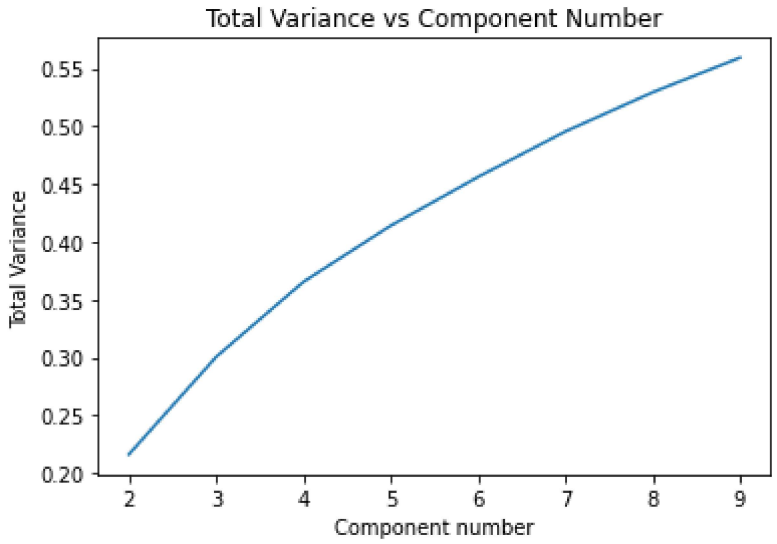
In [42]:
```
apply_kernel_pca(mnist_df, gamma=0.01)
```



Kernel PCA zadziałał już lepiej, eksperymentując z paramterem gamma można uzyskać dość widoczną separacją na dwie części : część po prawej (0, 6 ,4) i cześć po lewej. Dodatkowo w części po prawej mamy całkiem ładną rozróżnienie 3 różnych klas.

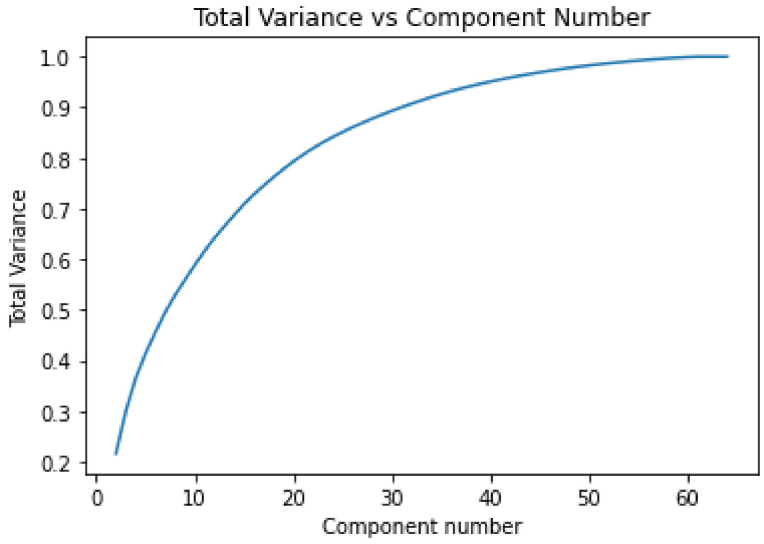
Porównanie total variance w przypadku użycia różnej liczby n_components.

In [43]:
```
compare_pc(mnist_df, n=10)
```



Total variance rośnie, ale widać że jest słabo. Dla n=10 użyskaliśmy tylko total_variance na poziomie 55 %.

In [44]:
```
compare_pc(mnist_df, n=65)
```



In [45]:
```
def is_equal(num1, num2, epsilon = 0.1):
    if (abs(num1 - num2) < epsilon):
        return True
    return False


def find_number_of_components_for_goal_variance(df, max_number_of_components, goal_
    x, y = [], []
    for i in range(2, max_number_of_components):
        pca_df, pca = create_pca(i, df)
        total_variance = np.sum(pca.explained_variance_ratio_)
        if (is_equal(goal_variance, total_variance)):
            print(f'{i} PCA components for {goal_variance} ')
            return


find_number_of_components_for_goal_variance(mnist_df, 65, 0.99)
find_number_of_components_for_goal_variance(mnist_df, 65, 0.95)
find_number_of_components_for_goal_variance(mnist_df, 65, 0.90)
find_number_of_components_for_goal_variance(mnist_df, 65, 0.85)
```

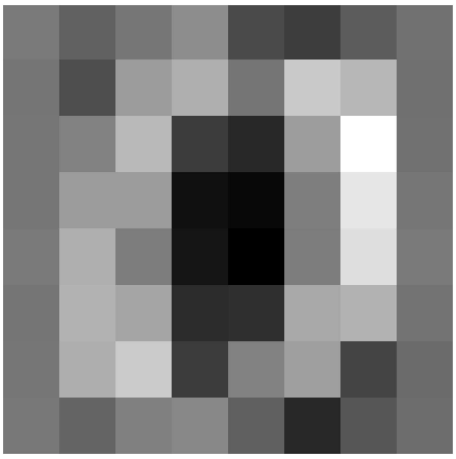
```
30 PCA components for 0.99
25 PCA components for 0.95
21 PCA components for 0.9
18 PCA components for 0.85
```

In [46]:
```
# W tym miejscu zorientowaliśmy się, że zamiast MNIST wczytaliśmy inny zbiór liczb
# https://scikit-learn.org/stable/datasets/toy_dataset.html#optical-recognition-of-
# jednak raczej w niczym to nie przeszkadza :)
```

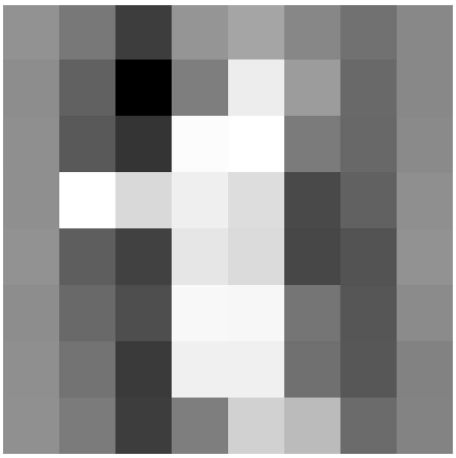
In [ ]:
```
```

In [47]:
```
def draw_digit(digit, width_height_pixels=8):
    '''
    digit: 1-row df
    '''
    digit = digit.to_numpy()
    img = np.reshape(digit, (width_height_pixels, width_height_pixels))
    plt.matshow(img, cmap='gray', interpolation='none')
    plt.axis('off')
    plt.show()


# MNIST digits
mnist_df_images = mnist_df.drop(columns="target")
```

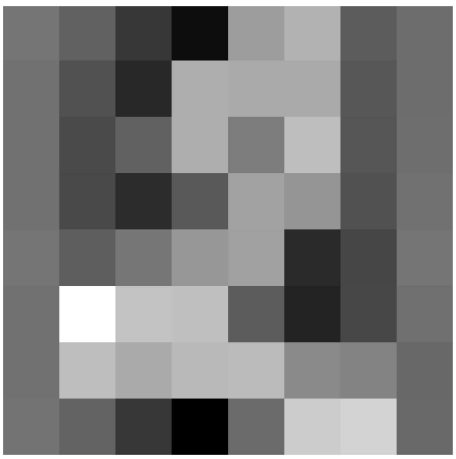
```
mnist_df_labels = mnist_df["target"]

randoms = [0,1,2,3,4,5,6,7,8,9]
for i in randoms:
    draw_digit(mnist_df_images.iloc[[i]])
    print(int(mnist_df_labels.iloc[[i]]))
# why I can't simply querry by row mnist_df_images[1]??
```



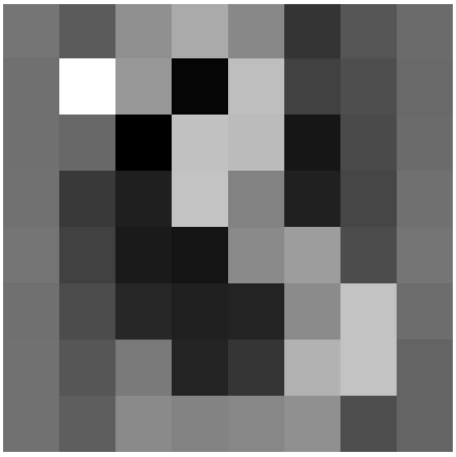
0



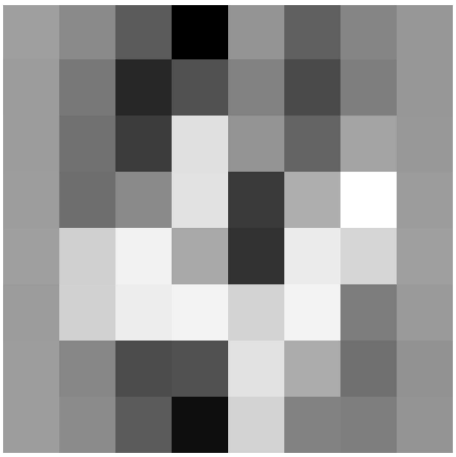
1



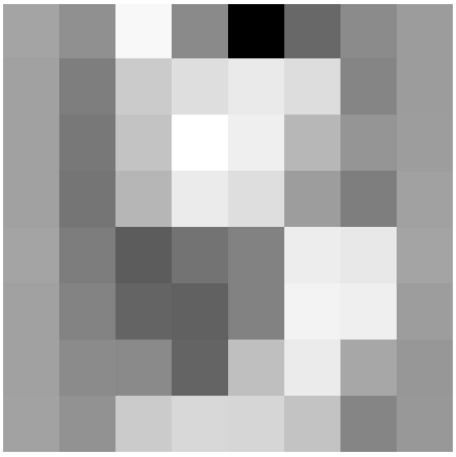
2



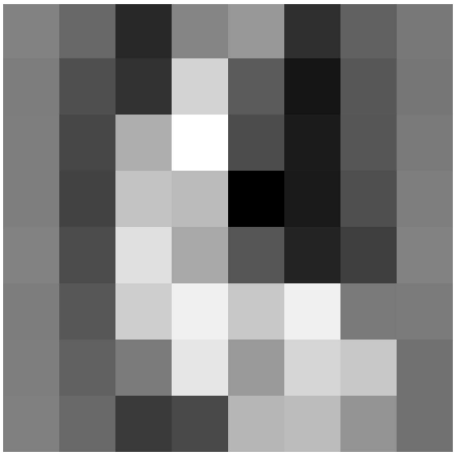
3



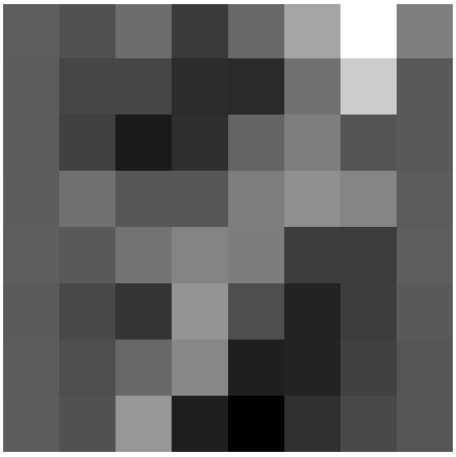
4



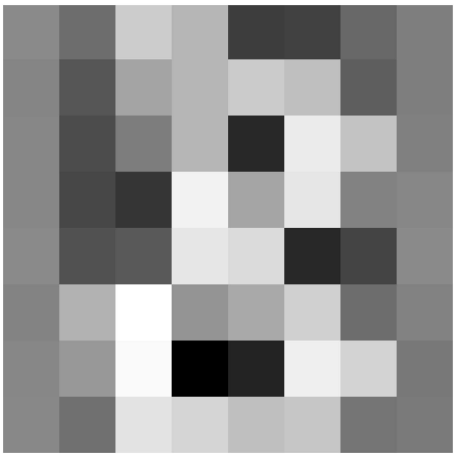
5



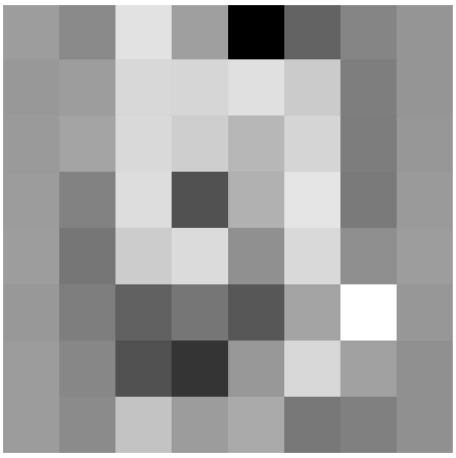
6



7



8

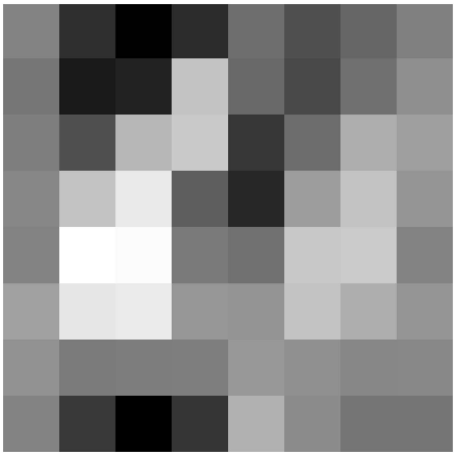


9

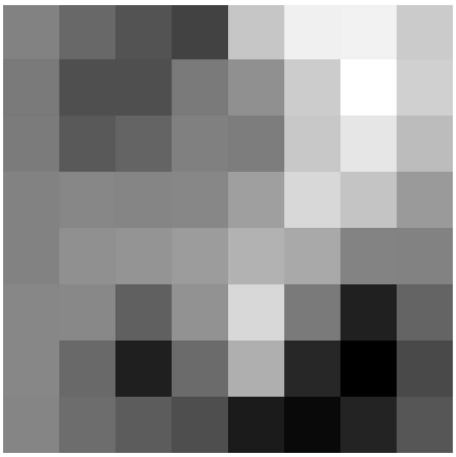
In [48]:
```
def draw_digit_pca(digit, width_height_pixels=8):
    '''
    digit: np array
    label: 1-row df with int as a label
    '''
    img = np.reshape(digit, (width_height_pixels, width_height_pixels))
    plt.matshow(img, cmap='gray', interpolation='none')
    plt.axis('off')
    plt.show()
```

In [49]:
```
# PCA digits transfomed back
inversed_mnist = pca_mnist.inverse_transform(pca_mnist_df.loc[:, pca_mnist_df.colu
randoms = [0,1,8,9]
for i in randoms:
    draw_digit_pca(inversed_mnist[i])
```

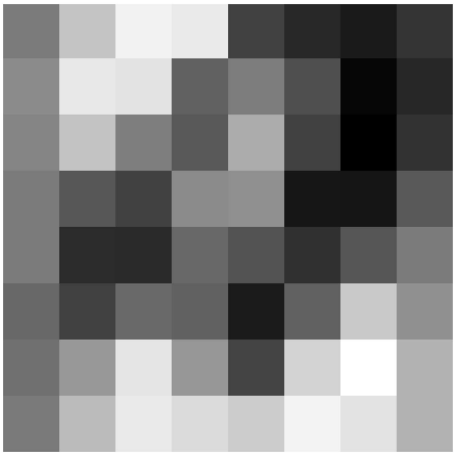
```
    print(i)
```



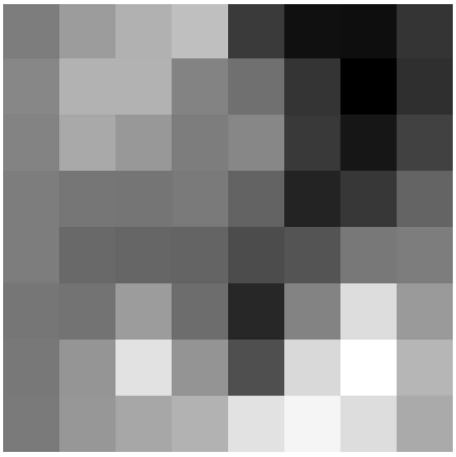
0



1



8

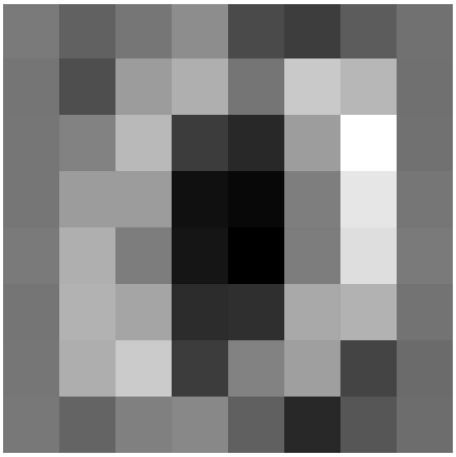


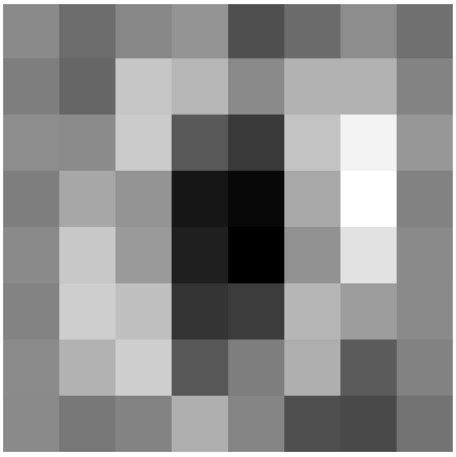
9

Powyższe wykresy przedsawiają PCA tylko z dwóch składowych liczby są zupenie nierozpoznawlne

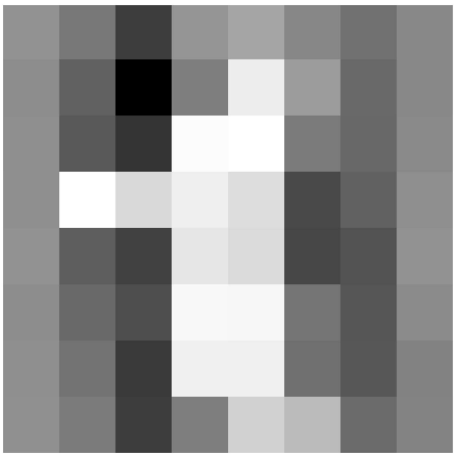
In [50]:
```
def compare_original_number_and_PCA_reconstuction(pca_components, input_df):
    pca_df, pca = create_pca(pca_components, input_df)
    input_df_images = mnist_df.drop(columns="target")
    input_df_labels = mnist_df["target"]
    inversed_pca = pca.inverse_transform(pca_df.loc[:, pca_df.columns != 'target'])

    randoms = [0,1,8,9]
    print(f'### Number of components: {pca_components}  ###')
    for i in randoms:
        print(f'*** comparing number: {int(input_df_labels.iloc[[i]])} *** ')
        draw_digit(input_df_images.iloc[[i]])
        draw_digit_pca(inversed_pca[i])

compare_original_number_and_PCA_reconstuction(30, mnist_df)
compare_original_number_and_PCA_reconstuction(25, mnist_df)
compare_original_number_and_PCA_reconstuction(21, mnist_df)
compare_original_number_and_PCA_reconstuction(18, mnist_df)
```

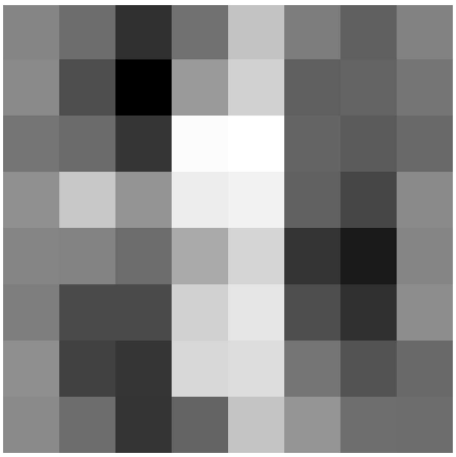
```
### Number of components: 30  ###
*** comparing number: 0 ***
```



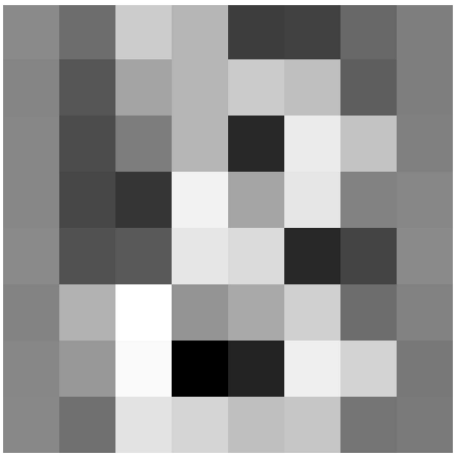


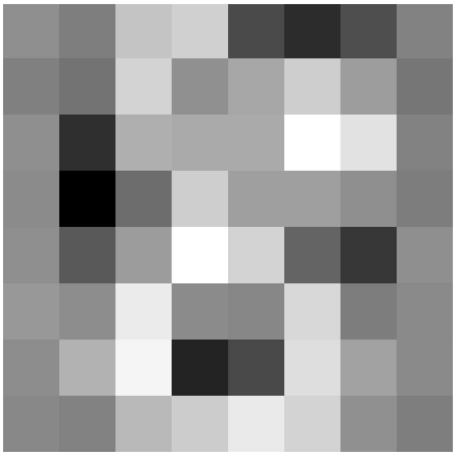
```
*** comparing number: 1 ***
```



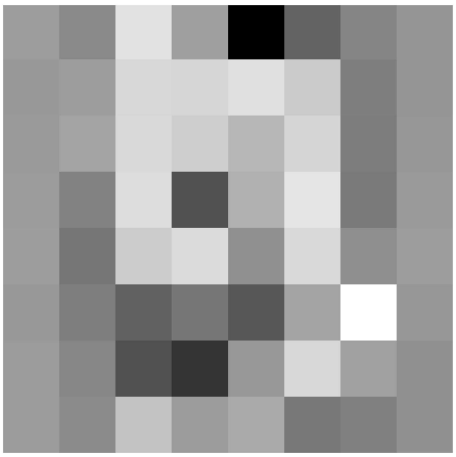


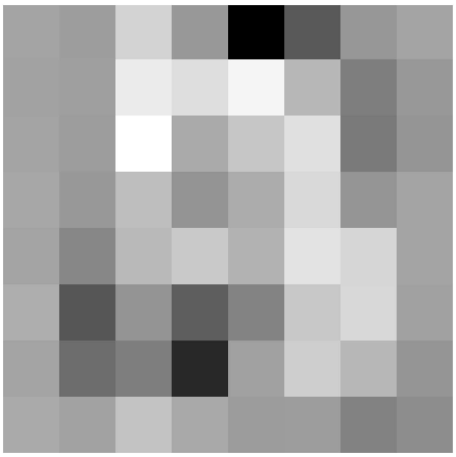
```
*** comparing number: 8 ***
```



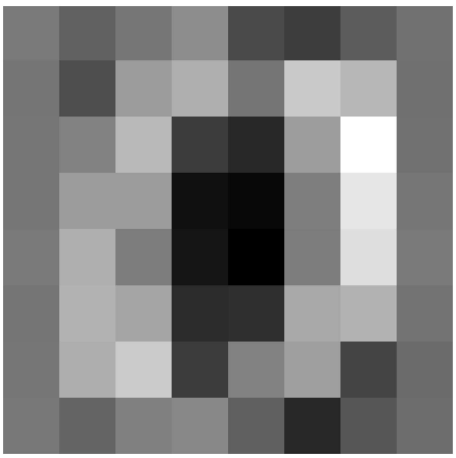


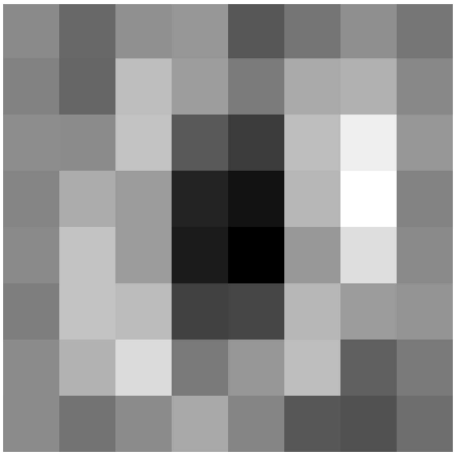
```
*** comparing number: 9 ***
```



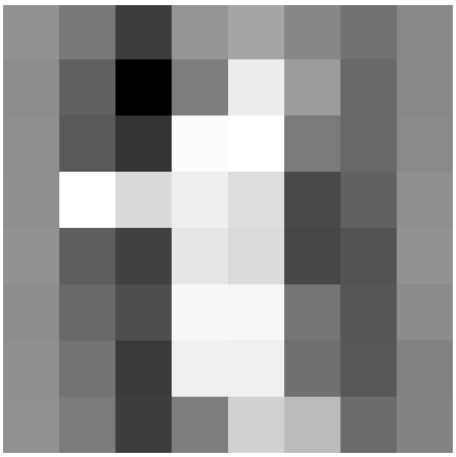


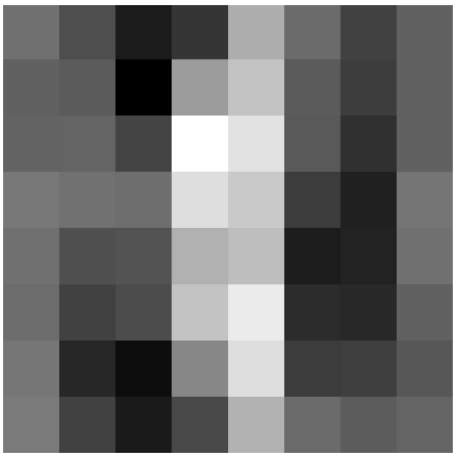
```
### Number of components: 25  ###
*** comparing number: 0 ***
```



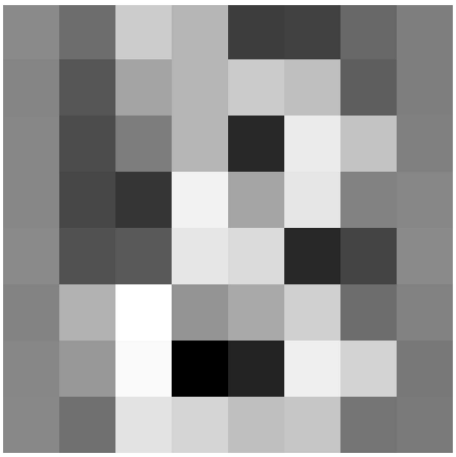


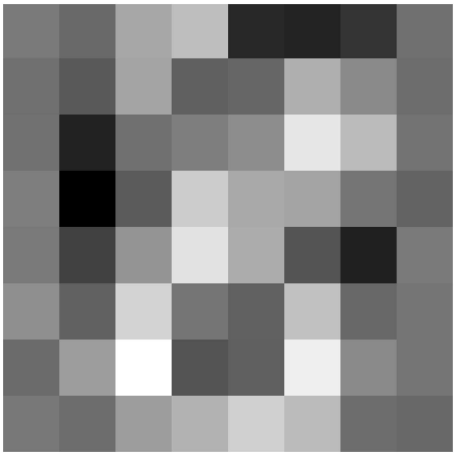
```
*** comparing number: 1 ***
```



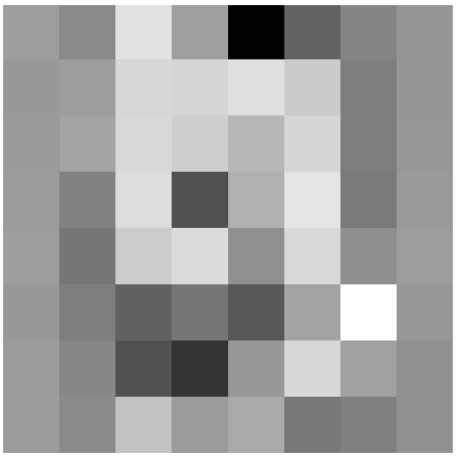


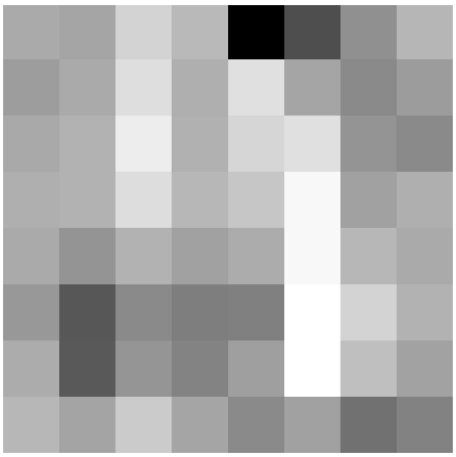
```
*** comparing number: 8 ***
```



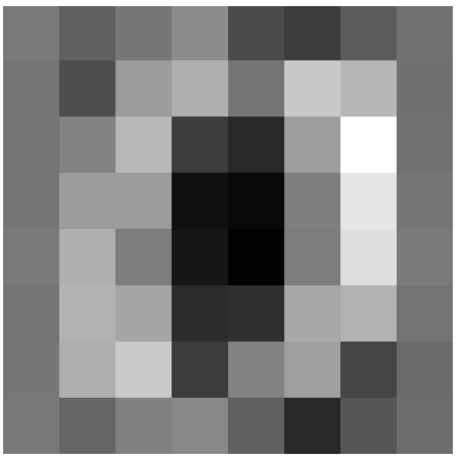


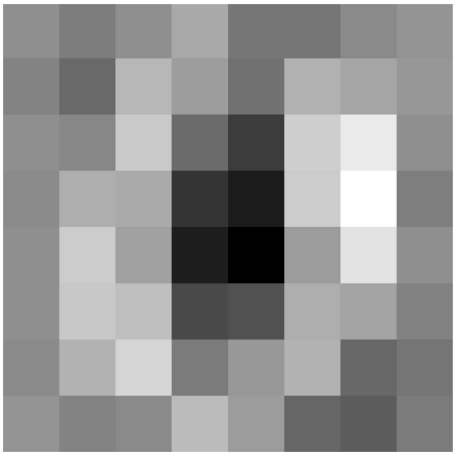
```
*** comparing number: 9 ***
```



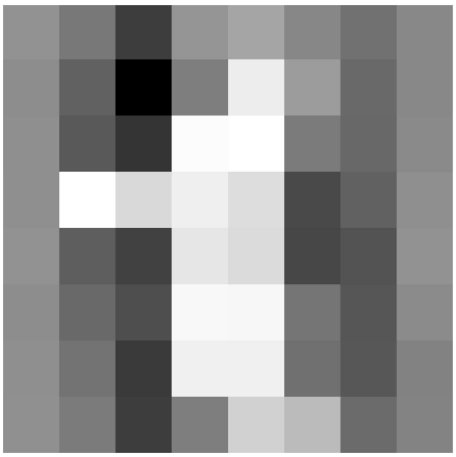


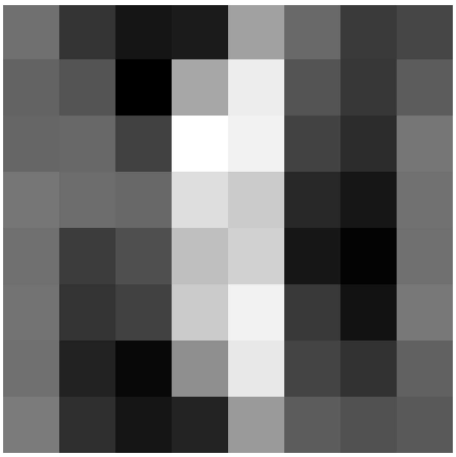
```
### Number of components: 21  ###
*** comparing number: 0 ***
```



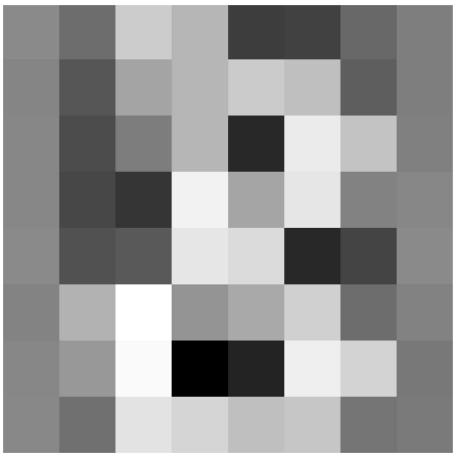


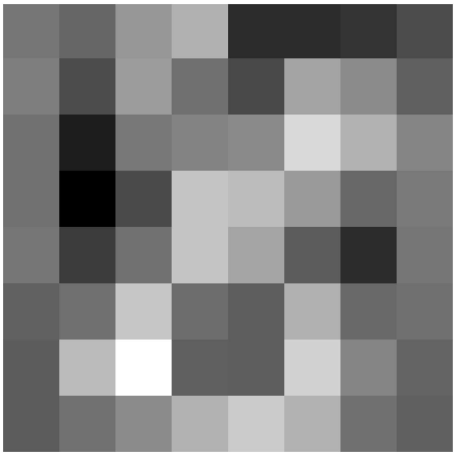
```
*** comparing number: 1 ***
```



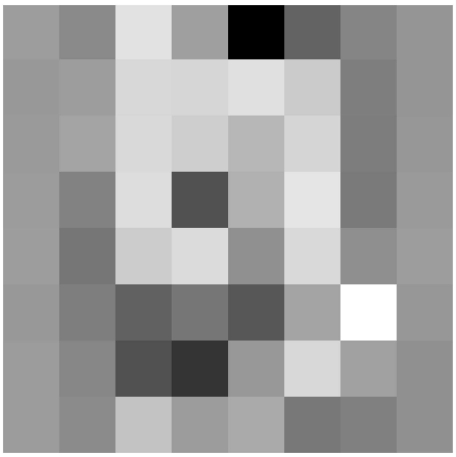


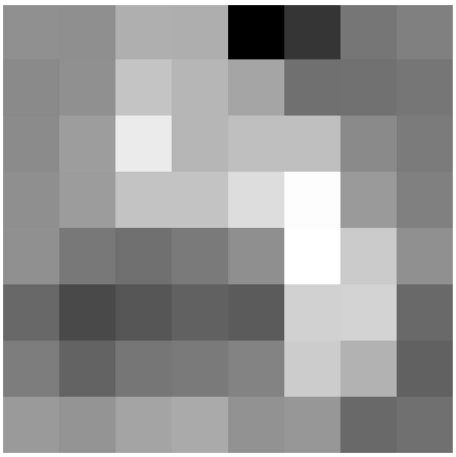
```
*** comparing number: 8 ***
```



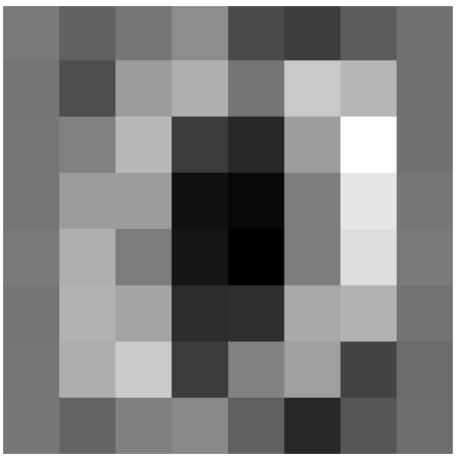


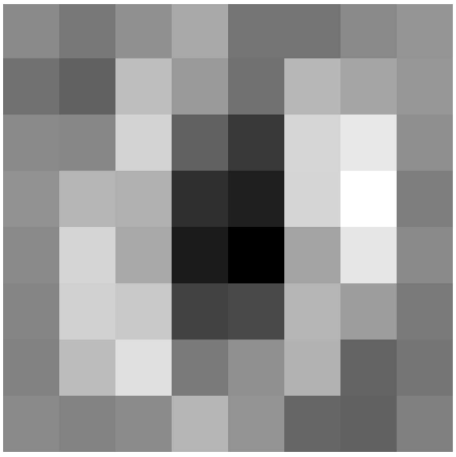
```
*** comparing number: 9 ***
```



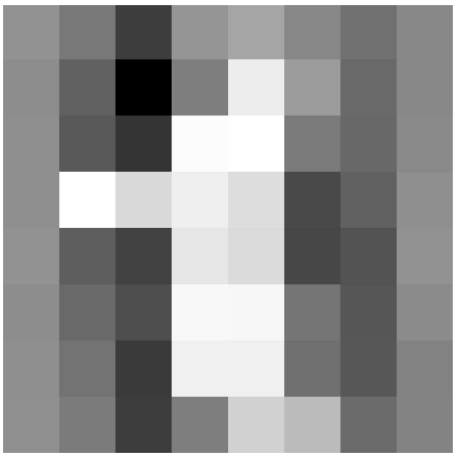


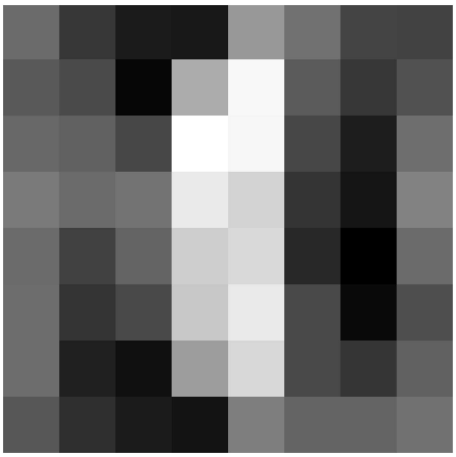
```
### Number of components: 18  ###
*** comparing number: 0 ***
```



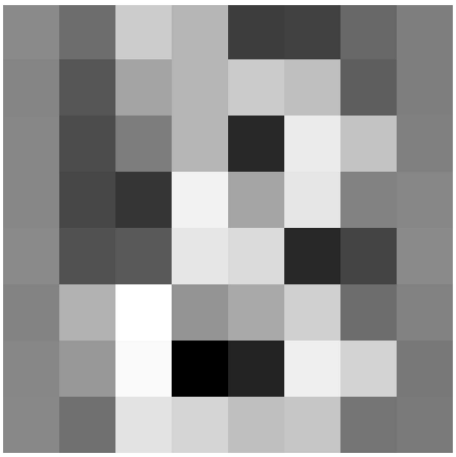


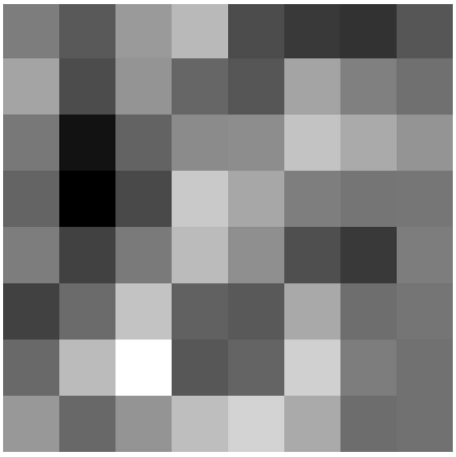
```
*** comparing number: 1 ***
```



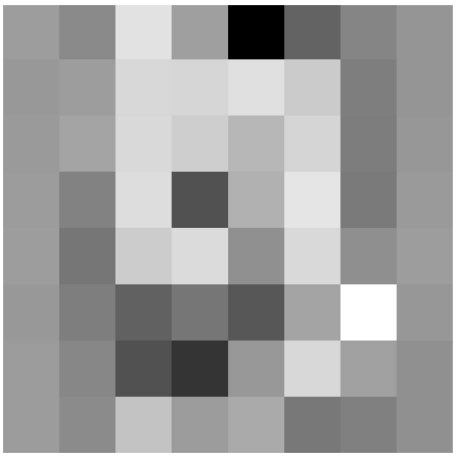


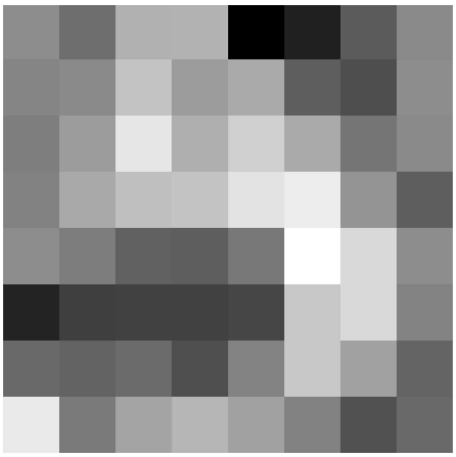
```
*** comparing number: 8 ***
```





```
*** comparing number: 9 ***
```





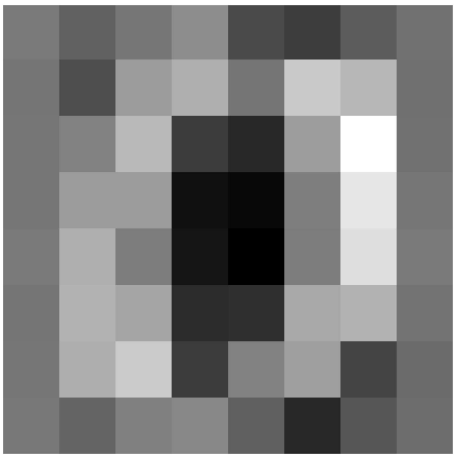
Dla liczby komponentów równej 30 (czyli odpowiadającej 99% wariancji) liczby i liczby otworzone z wariancji są identyczne. Natomiast dla liczby komponentór równej 18 (85% wariancji) widzimy, że liczby odtworzone z wariancji mają tylko najbardziej charakterystyczne cechy danej liczby - dla jedynki jest to prosta linia na pikselach środkowych.

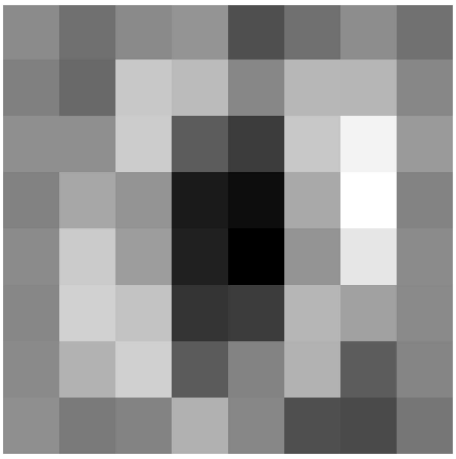
In [51]:
```
def apply_kernel_pca_mnist(df, n_comp, gamma=15, fig_size=(8,5)):
    features = df.loc[:, df.columns != 'target']
    k_pca = KernelPCA(kernel='rbf', gamma=gamma, n_components=n_comp, fit_inverse_t
    features_k_pca = k_pca.fit_transform(features)

    pca_df = pd.DataFrame(data=features_k_pca[:,0:2], columns=['x', 'y'])
    pca_df = pd.concat([pca_df, df['target']], axis=1)

    g= sns.relplot(data=pca_df, x='x', y='y', hue=pca_df['target'], palette='tab20
    g.figure.set_size_inches(fig_size)
```

```
    g.fig.suptitle("Kernel PCA")
    sns.despine()
```

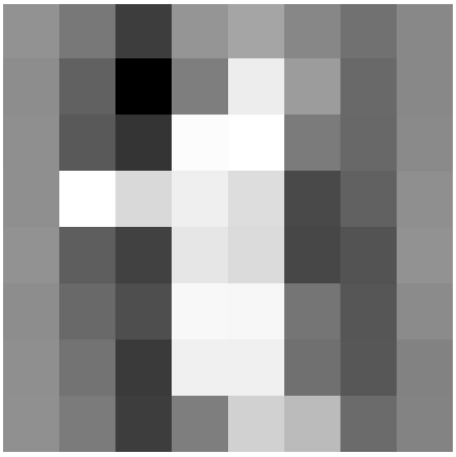
In [52]:
```
def compare_original_number_and_PCA_kernel_reconstuction(pca_components, input_df)
    pca_df, pca_kernel = apply_kernel_pca_mnist(input_df, pca_components, gamma=2.
    input_df_images = mnist_df.drop(columns="target")
    input_df_labels = mnist_df["target"]
    inversed_pca = pca_kernel.inverse_transform(pca_df.loc[:, pca_df.columns != 'ta

    randoms = [0,1,2,8,9]
    print(f'### Number of components: {pca_components}  ###')
    for i in randoms:
        print(f'*** comparing number: {int(input_df_labels.iloc[[i]])} *** ')
        draw_digit(input_df_images.iloc[[i]])
        draw_digit_pca(inversed_pca[i])

compare_original_number_and_PCA_reconstuction(30, mnist_df)
compare_original_number_and_PCA_reconstuction(25, mnist_df)
compare_original_number_and_PCA_reconstuction(21, mnist_df)
compare_original_number_and_PCA_reconstuction(18, mnist_df)
```

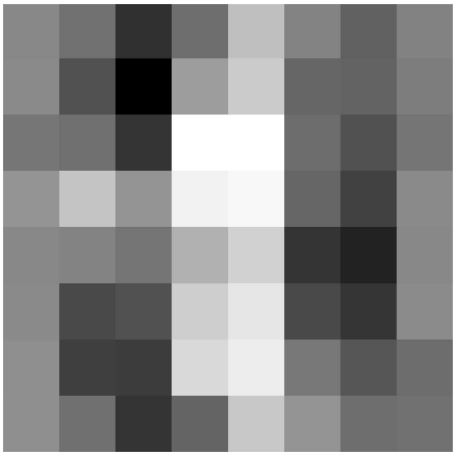
```
### Number of components: 30  ###
*** comparing number: 0 ***
```



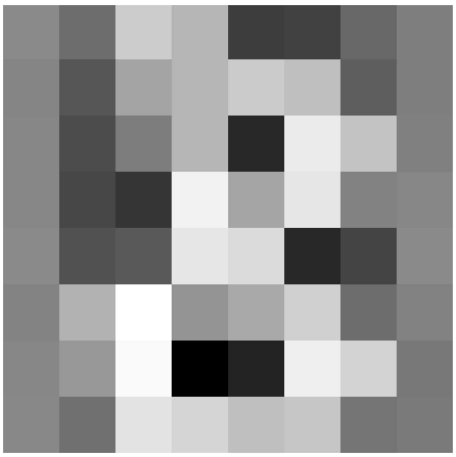


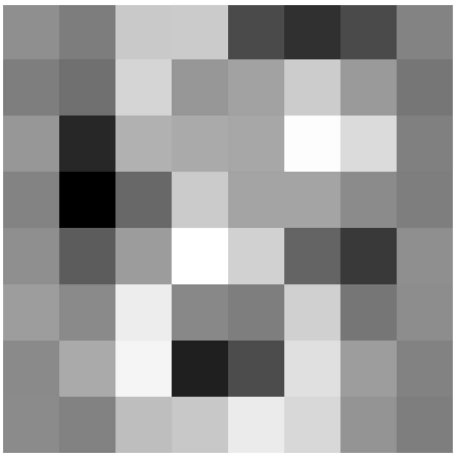
```
*** comparing number: 1 ***
```



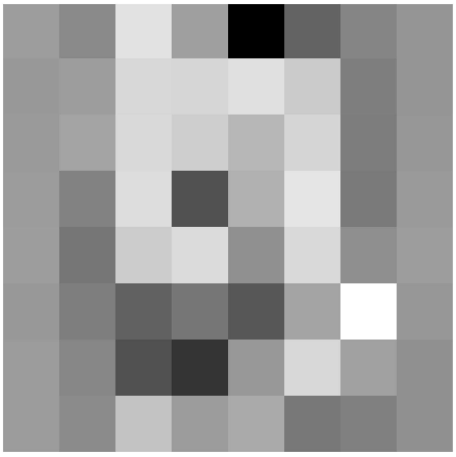


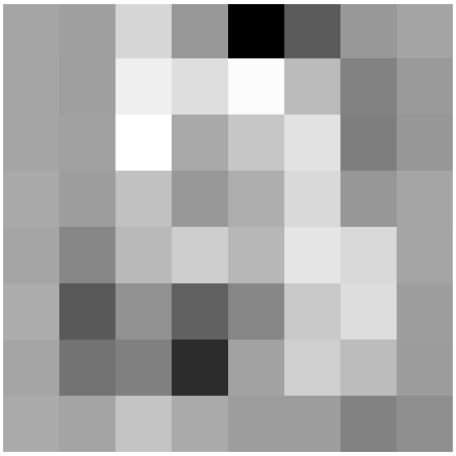
```
*** comparing number: 8 ***
```



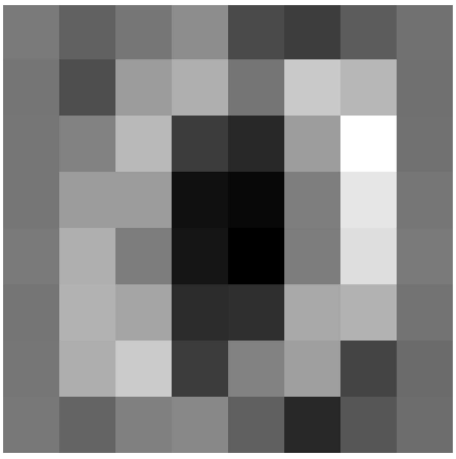


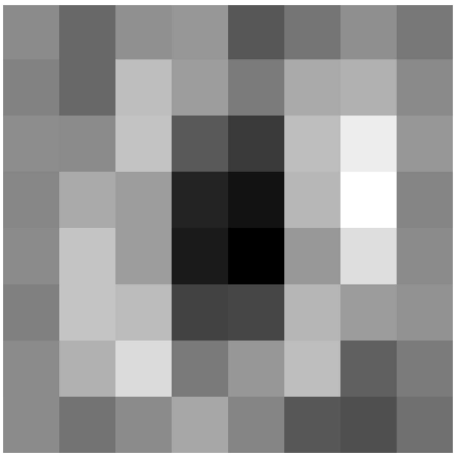
```
*** comparing number: 9 ***
```



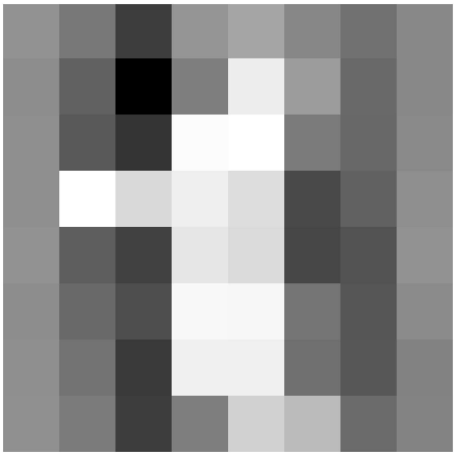


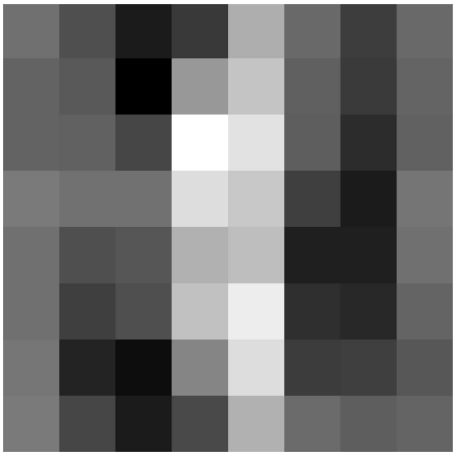
```
### Number of components: 25  ###
*** comparing number: 0 ***
```



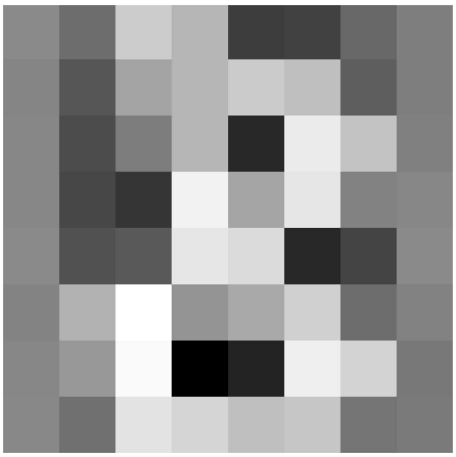


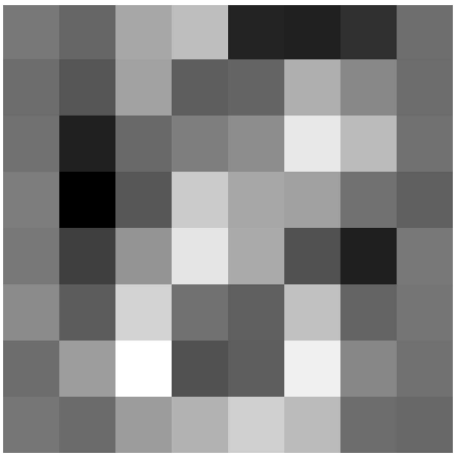
```
*** comparing number: 1 ***
```



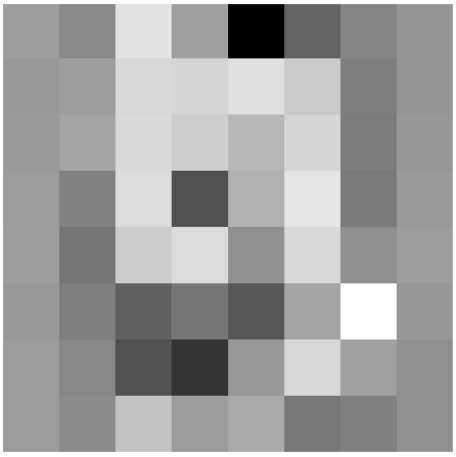


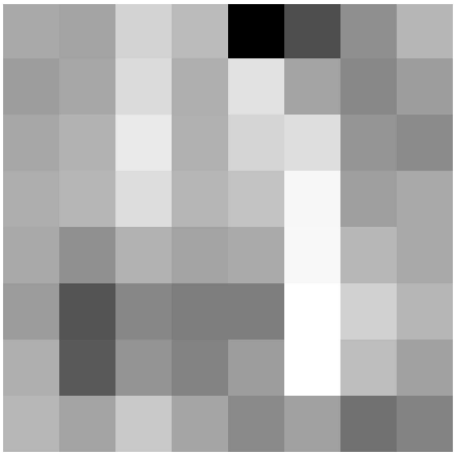
```
*** comparing number: 8 ***
```



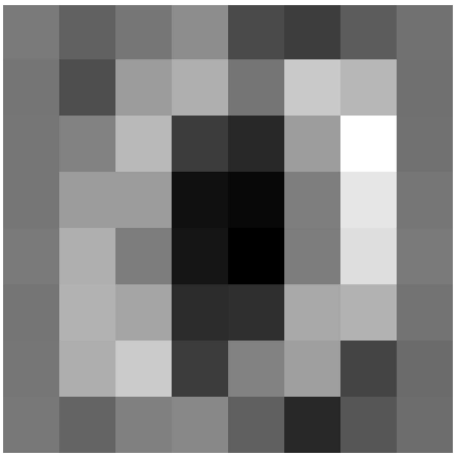


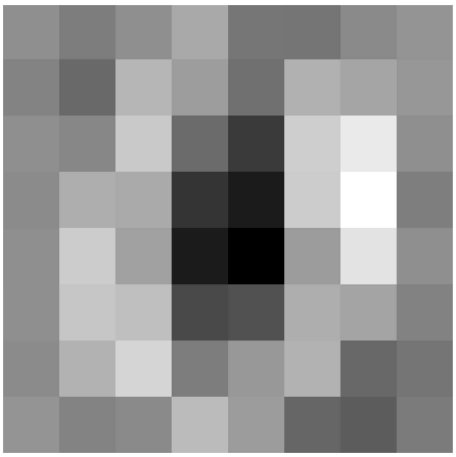
```
*** comparing number: 9 ***
```



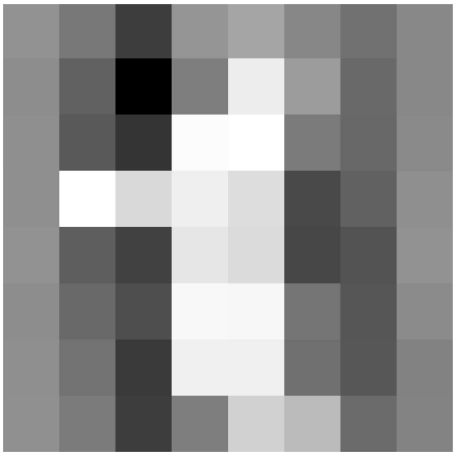


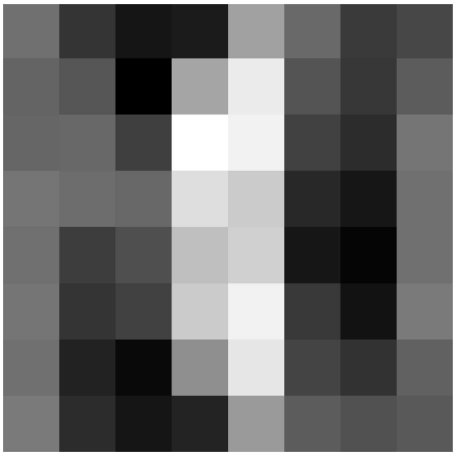
```
### Number of components: 21  ###
*** comparing number: 0 ***
```



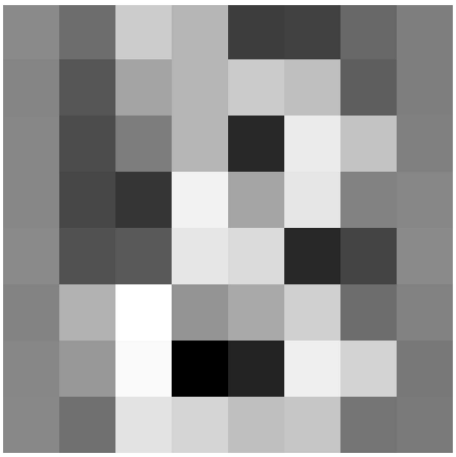


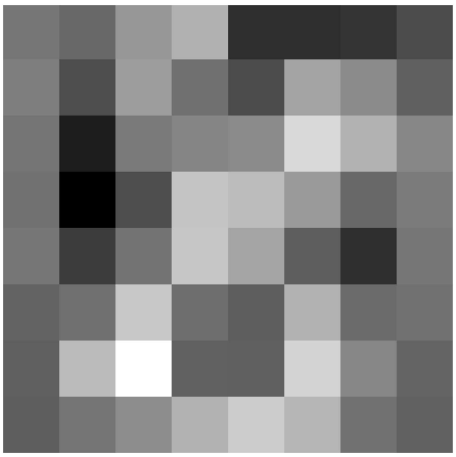
```
*** comparing number: 1 ***
```



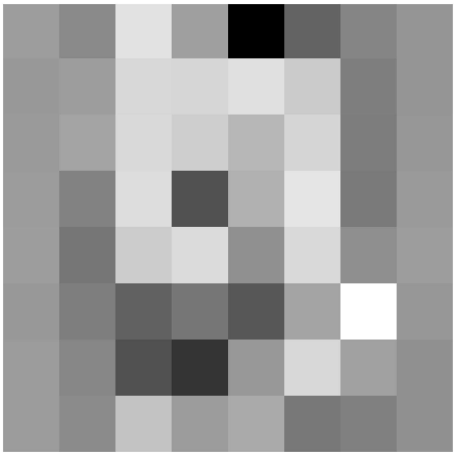


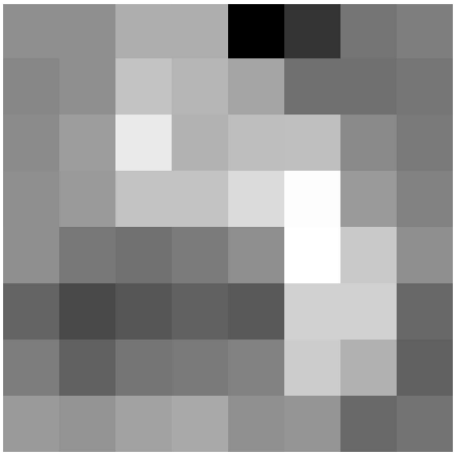
```
*** comparing number: 8 ***
```



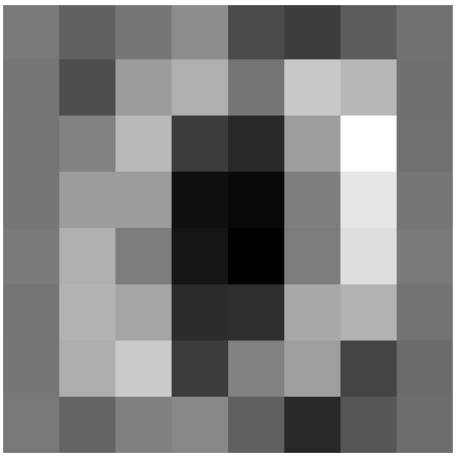


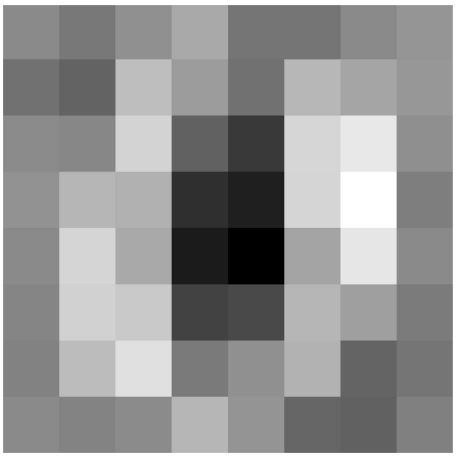
```
*** comparing number: 9 ***
```



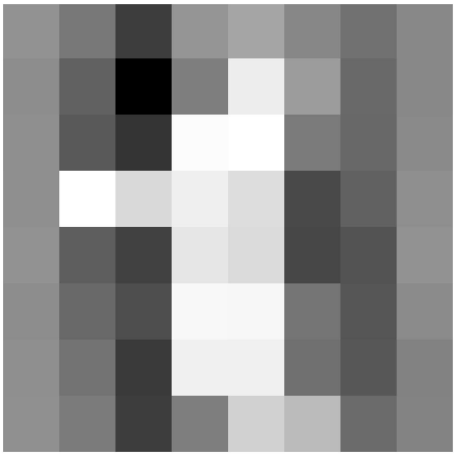


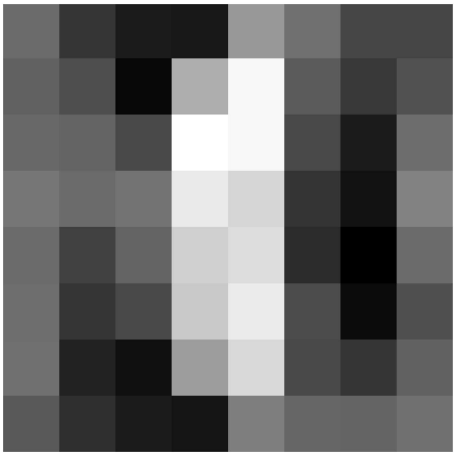
```
### Number of components: 18  ###
*** comparing number: 0 ***
```



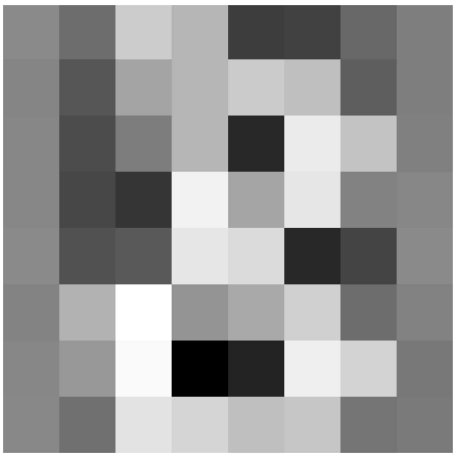


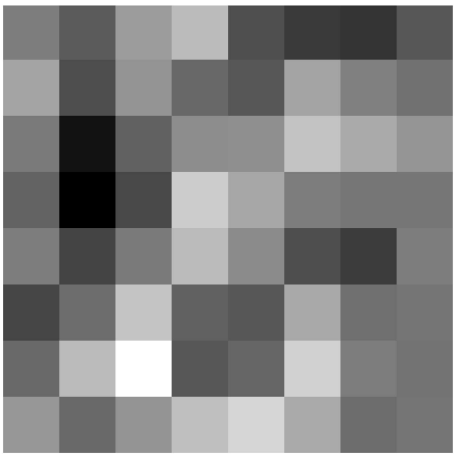
```
*** comparing number: 1 ***
```



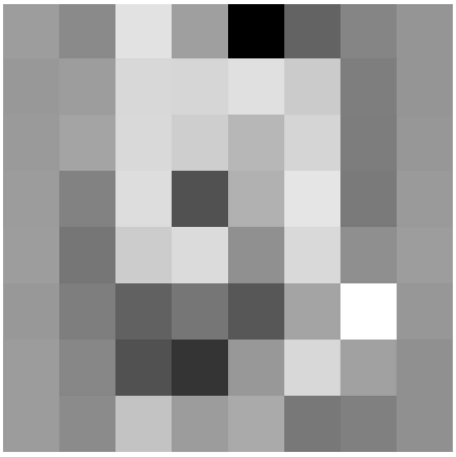


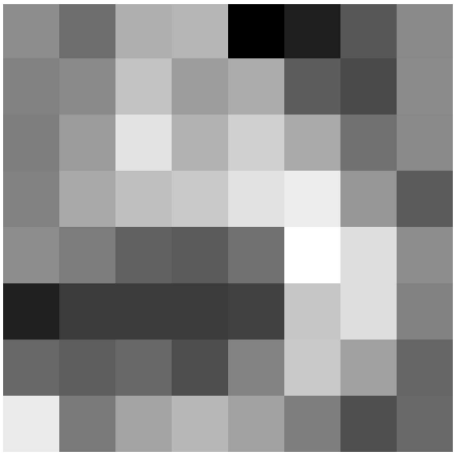
```
*** comparing number: 8 ***
```





```
*** comparing number: 9 ***
```





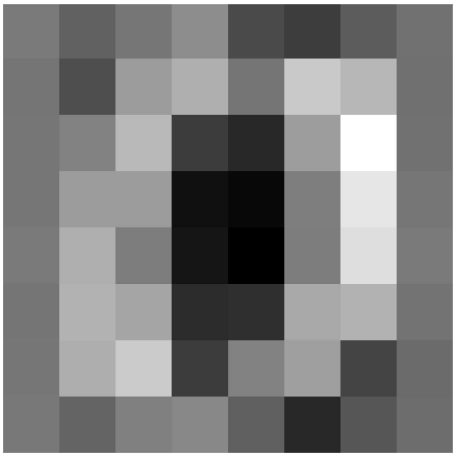
Zachowanie PCA i PCA-Kernel jest bardzo podobne - dokładna analiza wymaga dodatkowego zestawienia:

In [53]:
```
def apply_kernel_pca_mnist_2(df, n_comp, gamma=15, fig_size=(8,5)):
    features = df.loc[:, df.columns != 'target']
    k_pca = KernelPCA(kernel='rbf', gamma=gamma, n_components=n_comp, fit_inverse_t
    features_k_pca = k_pca.fit_transform(features)

    return features_k_pca, k_pca
```

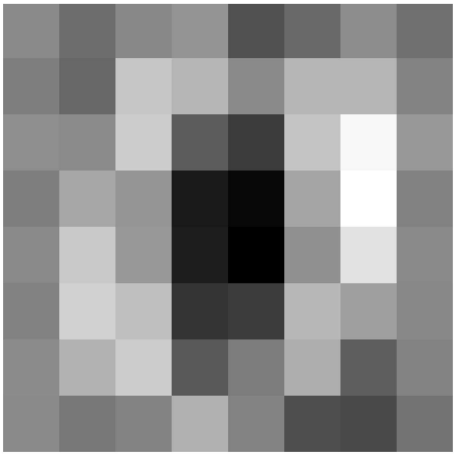
In [54]:
```
def compare_originalNumber_PCA_PCAkernel_reconstuction(pca_components, input_df):
    pca_kernel_df, pca_kernel = apply_kernel_pca_mnist_2(input_df, pca_components,
    inversed_pca_kernel= pca_kernel.inverse_transform(pca_kernel_df)

    pca_df, pca = create_pca(pca_components, input_df)
    inversed_pca = pca.inverse_transform(pca_df.loc[:, pca_df.columns != 'target'])

    input_df_images = mnist_df.drop(columns="target")
    input_df_labels = mnist_df["target"]

    randoms = [0,1,2,3,4,5,6,7,8,9]
    print(f'### Number of components: {pca_components}  ###')
    for i in randoms:
        print(f'*** comparing number: {int(input_df_labels.iloc[[i]])} *** ')
        draw_digit(input_df_images.iloc[[i]])
        print('^ original number ^')
        draw_digit_pca(inversed_pca[i])
        print('^ PCA number ^')
        draw_digit_pca(inversed_pca_kernel[i])
        print('^ kernelPCA number ^\n\n')
```

```
compare_originalNumber_PCA_PCAkernel_reconstuction(30, mnist_df)
compare_originalNumber_PCA_PCAkernel_reconstuction(25, mnist_df)
compare_originalNumber_PCA_PCAkernel_reconstuction(21, mnist_df)
compare_originalNumber_PCA_PCAkernel_reconstuction(18, mnist_df)
```

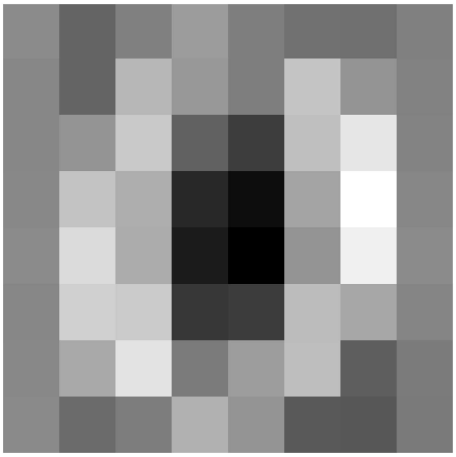
```
### Number of components: 30  ###
*** comparing number: 0 ***
```



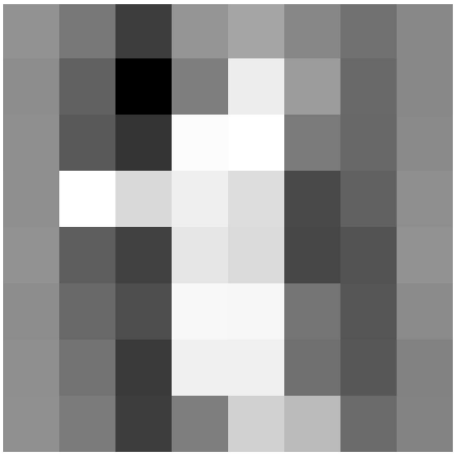
```
^ original number ^
```



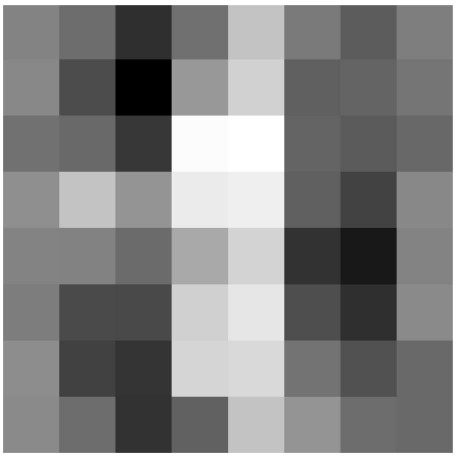
```
^ PCA number ^
```



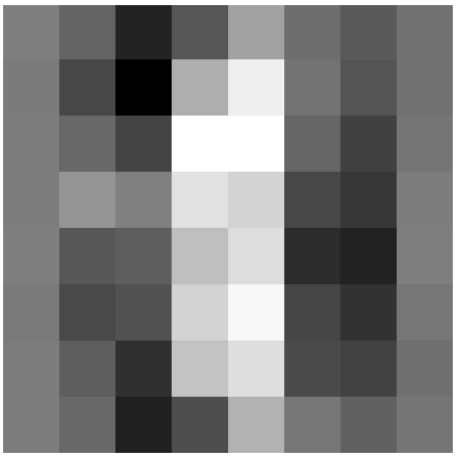
```
^ kernelPCA number ^


*** comparing number: 1 ***
```



^ original number ^

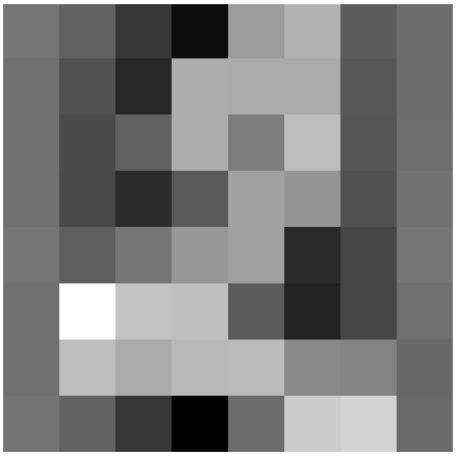


^ PCA number ^

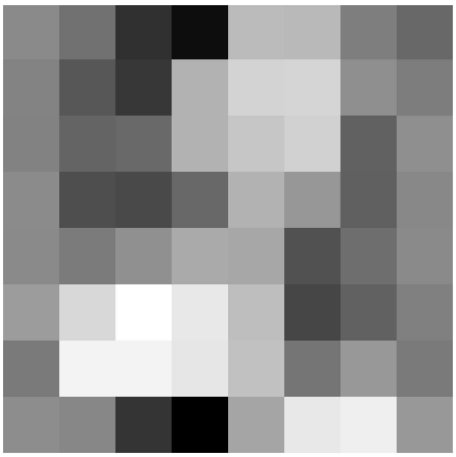


^ kernelPCA number ^

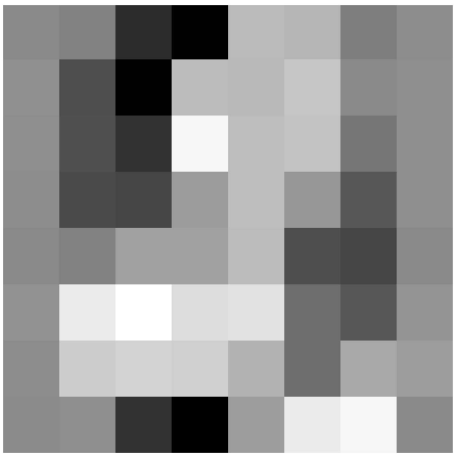
*** comparing number: 2 ***



^ original number ^

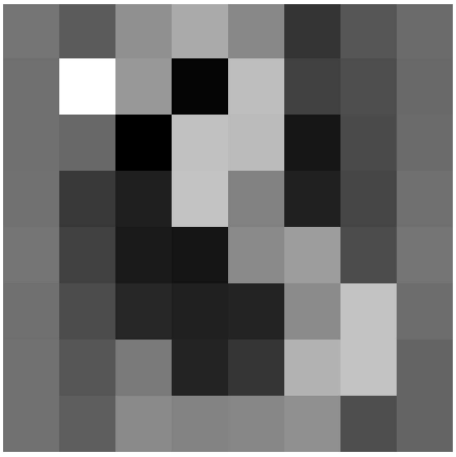


^ PCA number ^

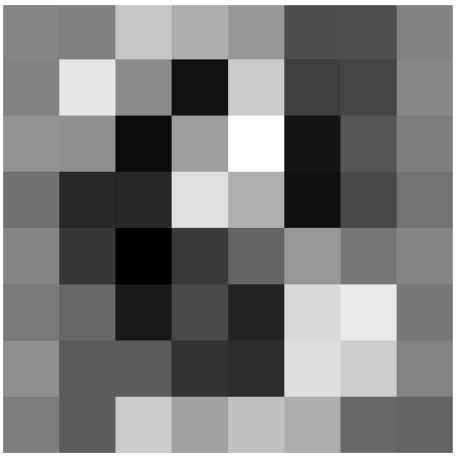


^ kernelPCA number ^

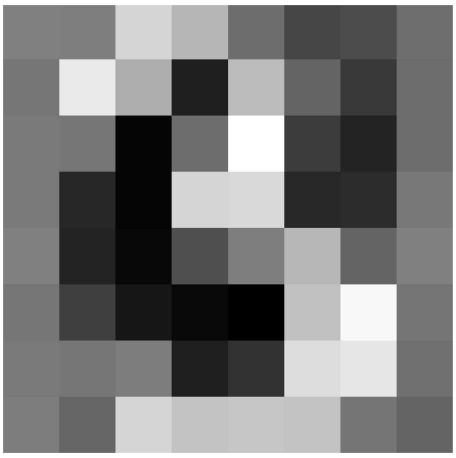
*** comparing number: 3 ***



^ original number ^

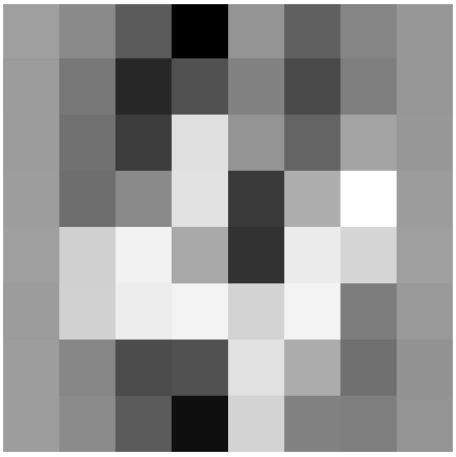


^ PCA number ^

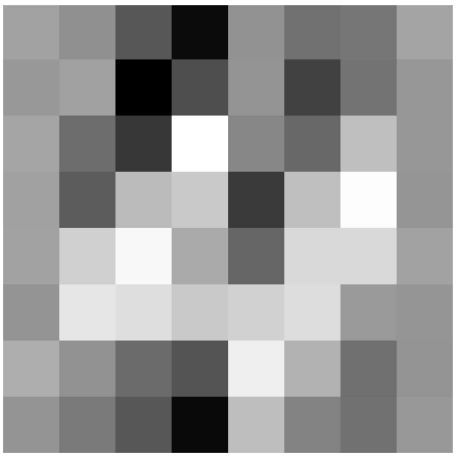


^ kernelPCA number ^

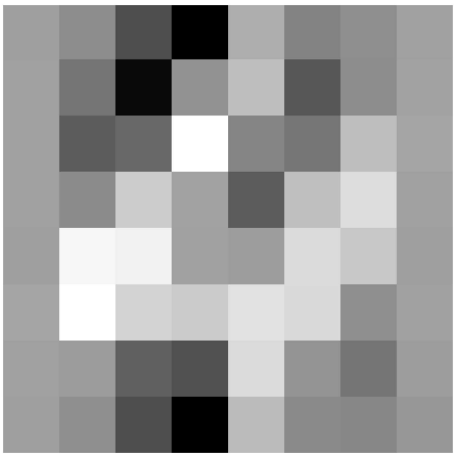
*** comparing number: 4 ***



^ original number ^

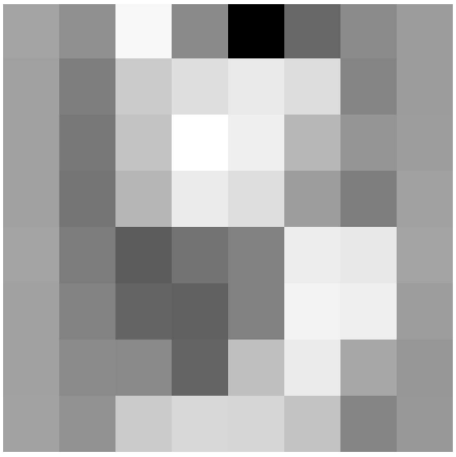


^ PCA number ^

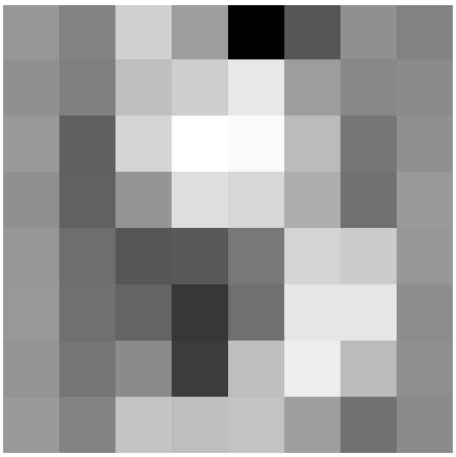


^ kernelPCA number ^

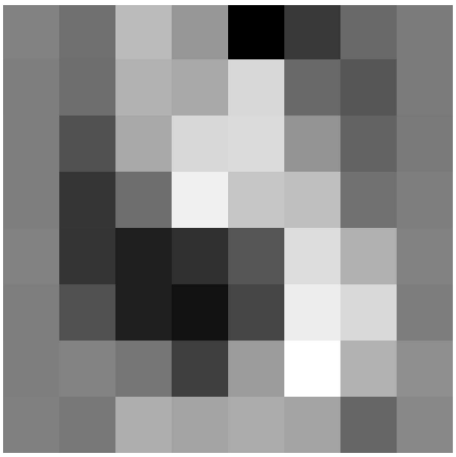
*** comparing number: 5 ***



^ original number ^

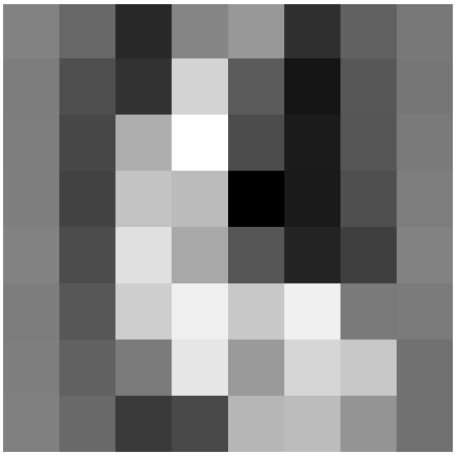


^ PCA number ^

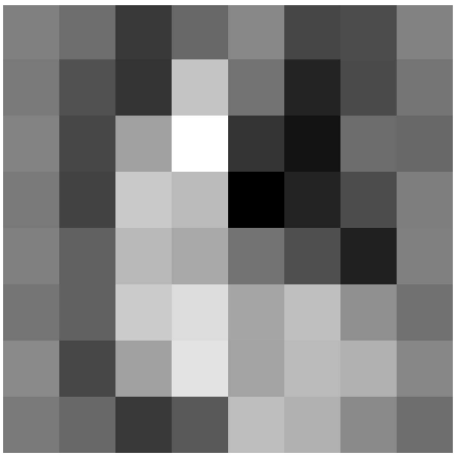


^ kernelPCA number ^

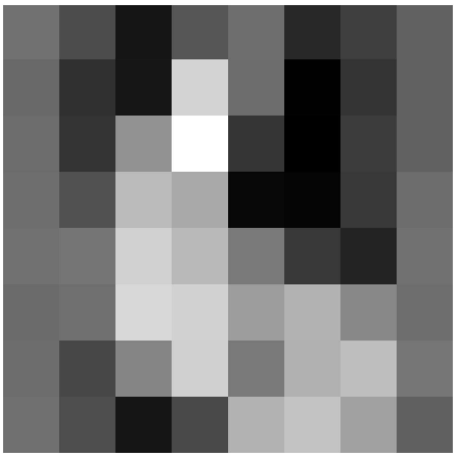
*** comparing number: 6 ***



^ original number ^

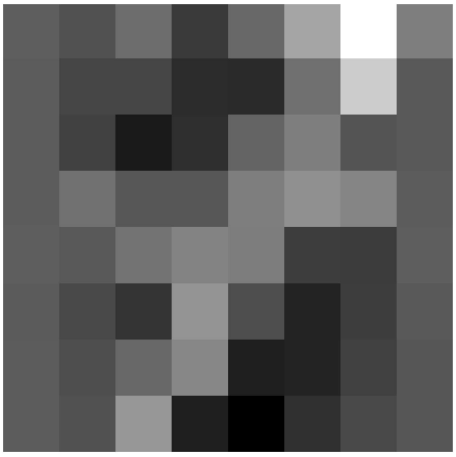


^ PCA number ^

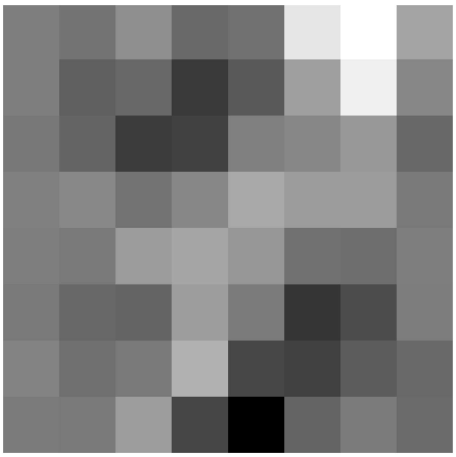


^ kernelPCA number ^

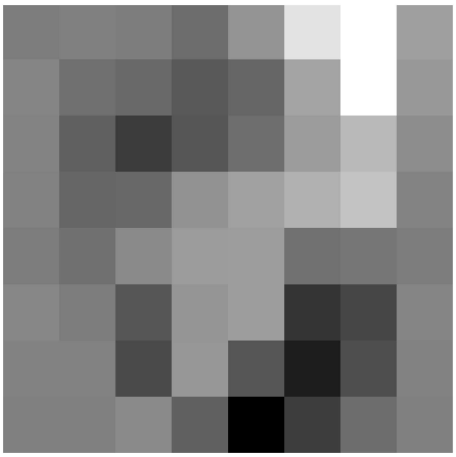
*** comparing number: 7 ***



^ original number ^

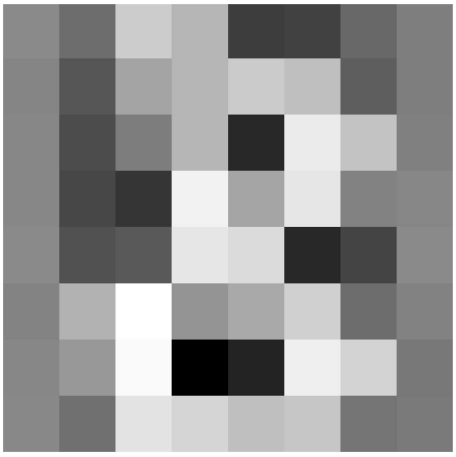


^ PCA number ^

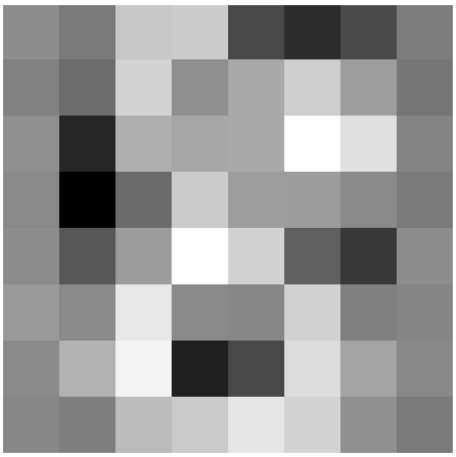


^ kernelPCA number ^

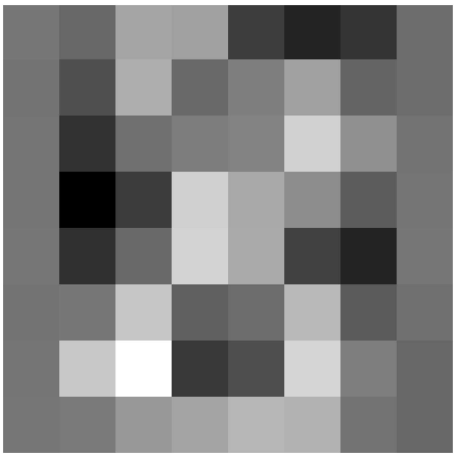
*** comparing number: 8 ***



^ original number ^

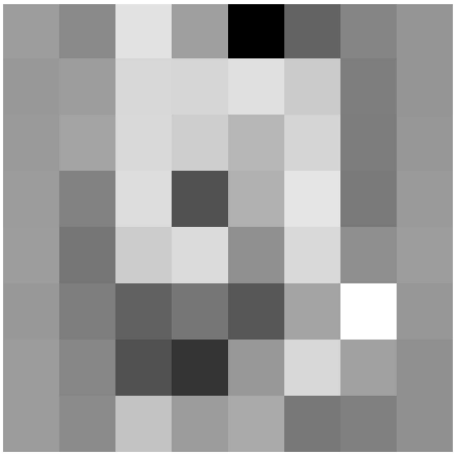


^ PCA number ^

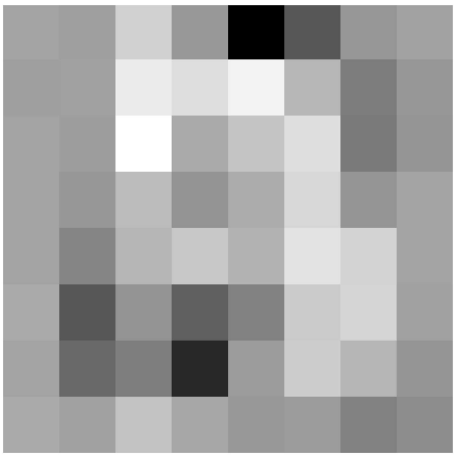


^ kernelPCA number ^

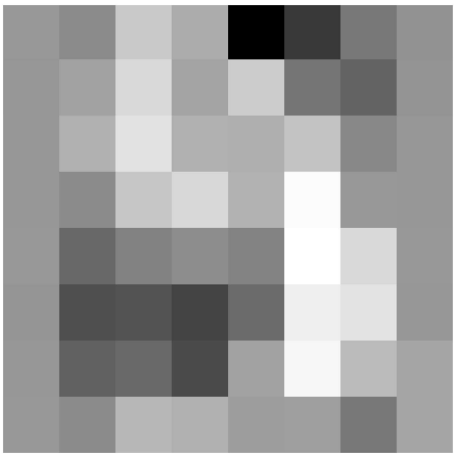
*** comparing number: 9 ***



^ original number ^

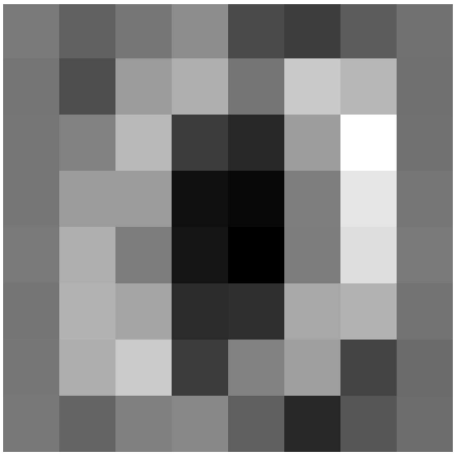


^ PCA number ^

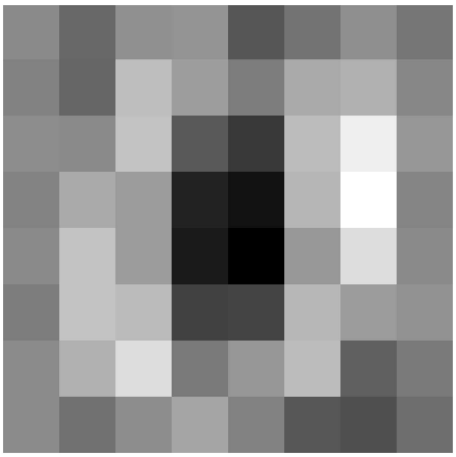


^ kernelPCA number ^

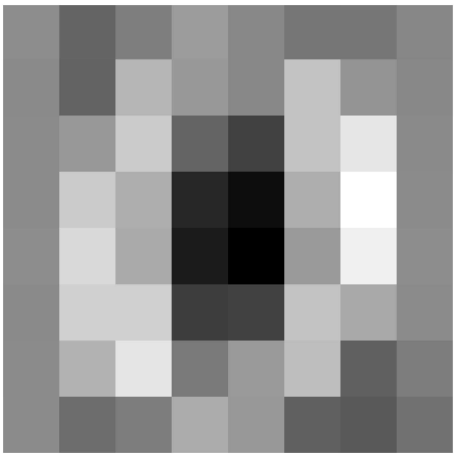
```
### Number of components: 25  ###
*** comparing number: 0 ***
```



^ original number ^

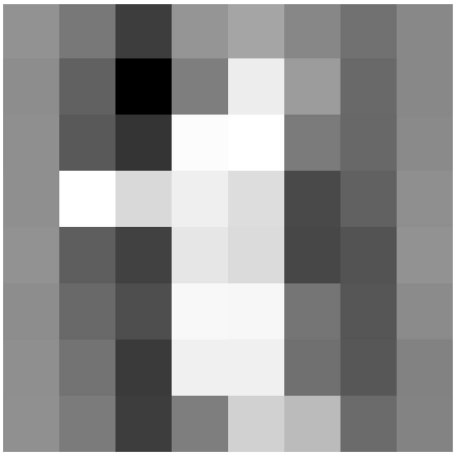


^ PCA number ^

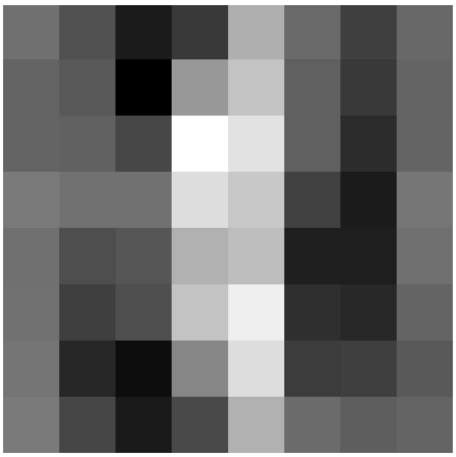


^ kernelPCA number ^

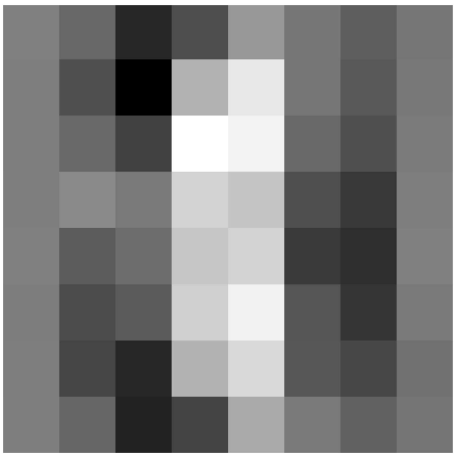
*** comparing number: 1 ***



^ original number ^

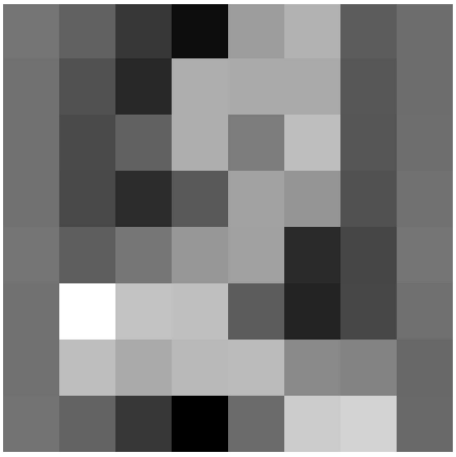


^ PCA number ^

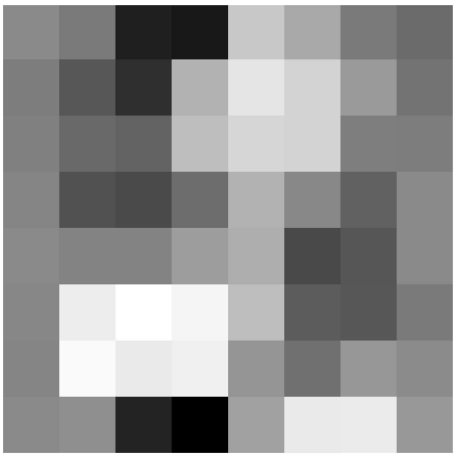


^ kernelPCA number ^

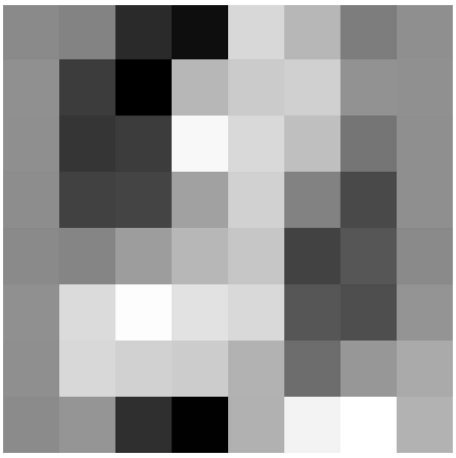
*** comparing number: 2 ***



^ original number ^

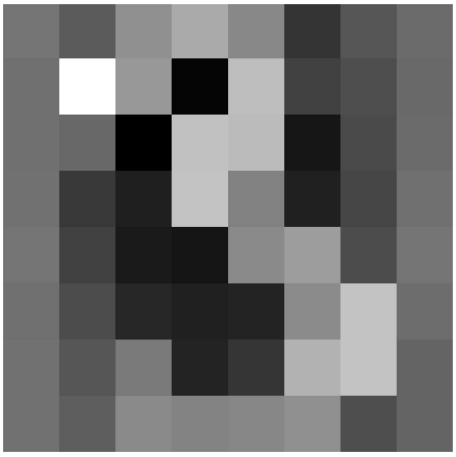


^ PCA number ^

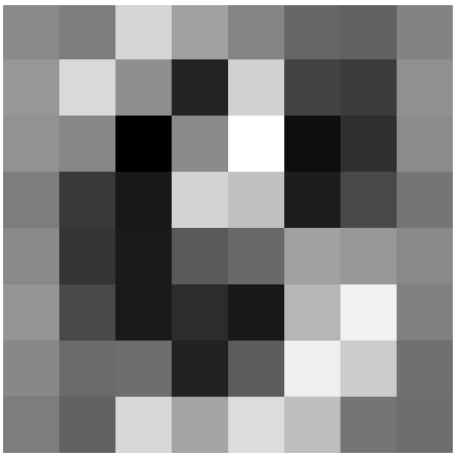


^ kernelPCA number ^

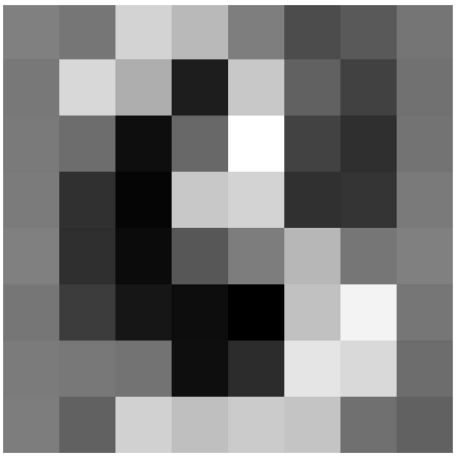
*** comparing number: 3 ***



^ original number ^

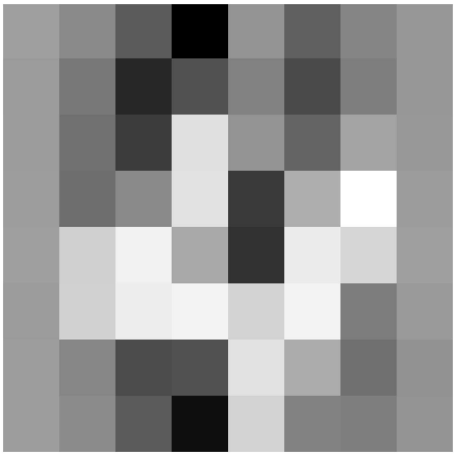


^ PCA number ^

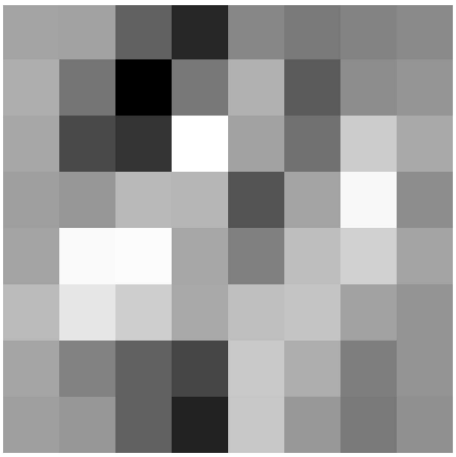


^ kernelPCA number ^

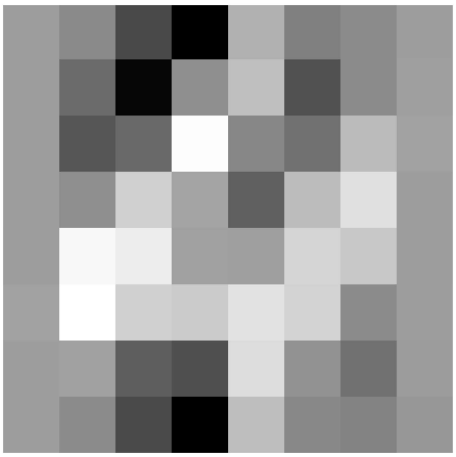
*** comparing number: 4 ***



^ original number ^

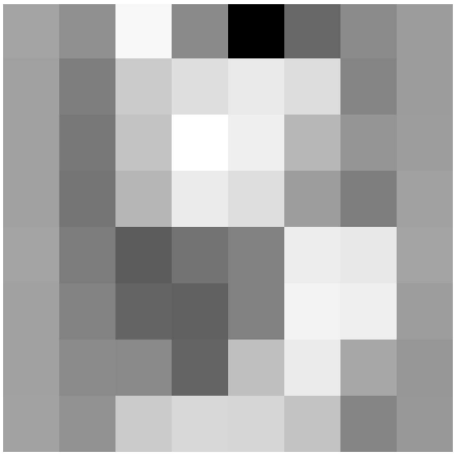


^ PCA number ^

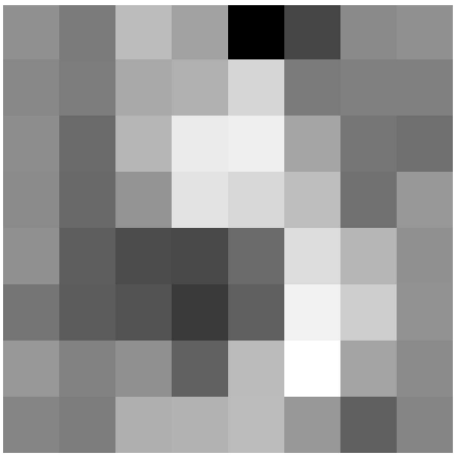


^ kernelPCA number ^

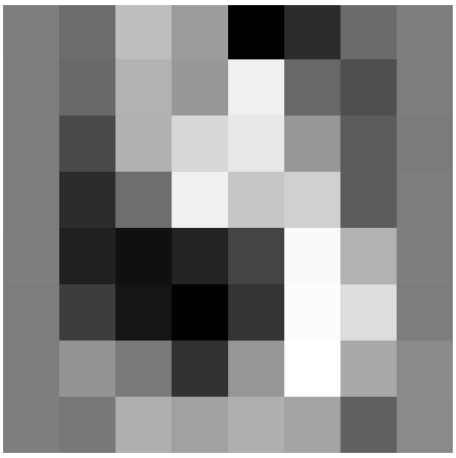
*** comparing number: 5 ***



^ original number ^

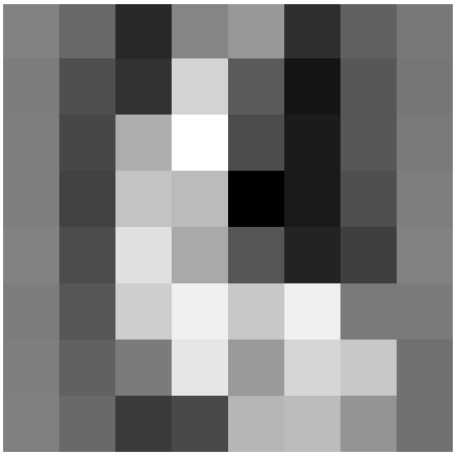


^ PCA number ^

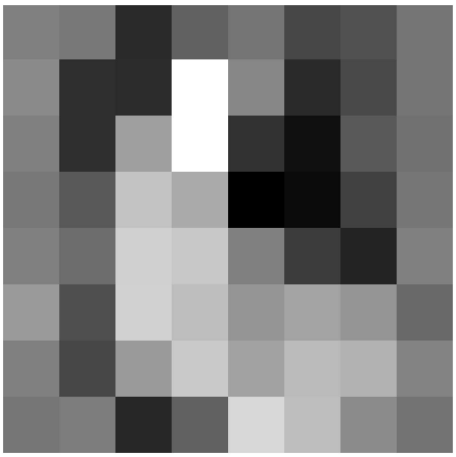


^ kernelPCA number ^

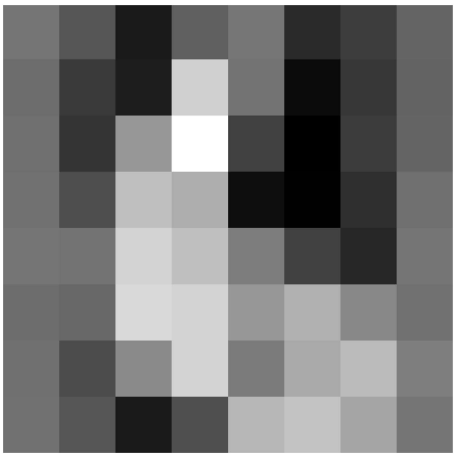
*** comparing number: 6 ***



^ original number ^

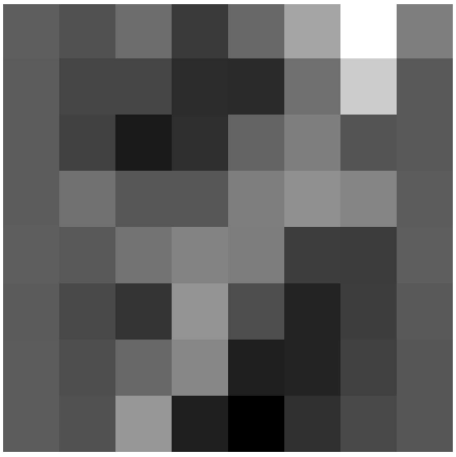


^ PCA number ^

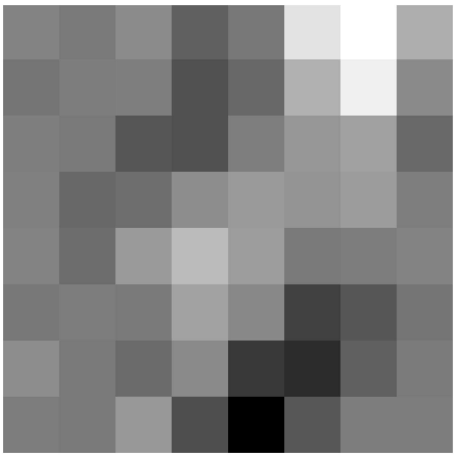


^ kernelPCA number ^

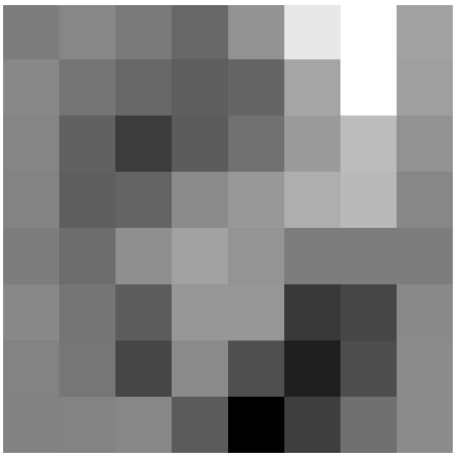
*** comparing number: 7 ***



^ original number ^

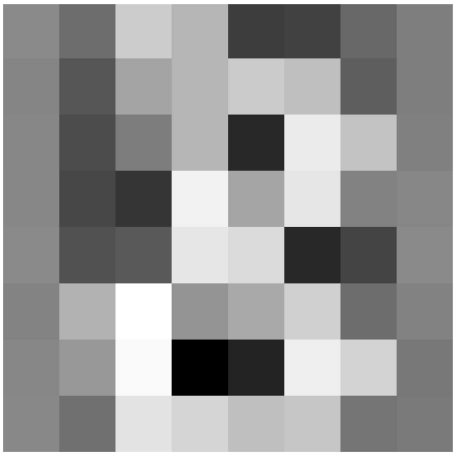


^ PCA number ^

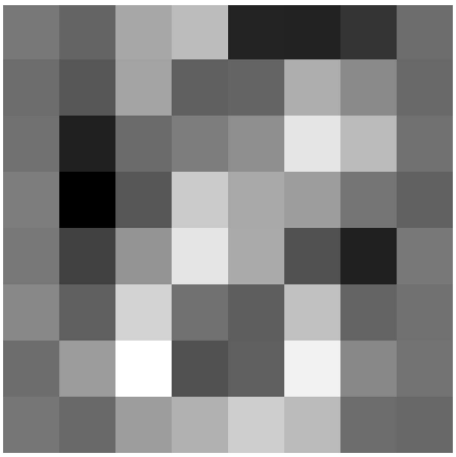


^ kernelPCA number ^

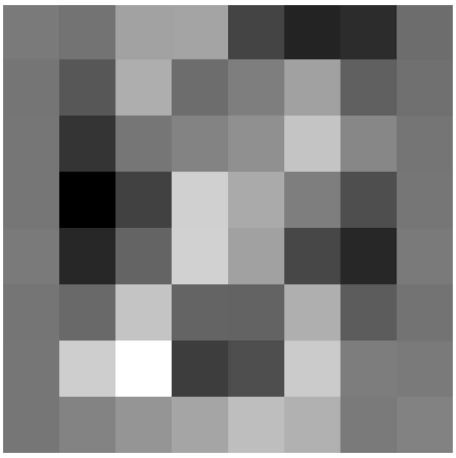
*** comparing number: 8 ***



^ original number ^

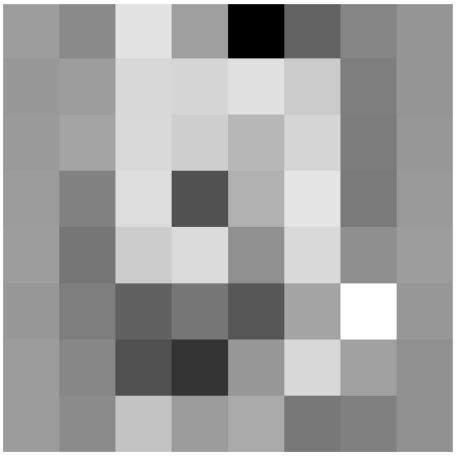


^ PCA number ^

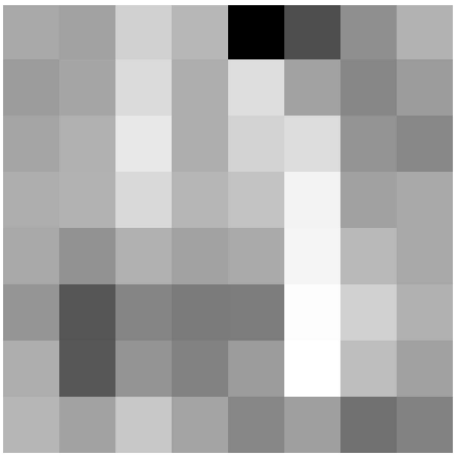


^ kernelPCA number ^

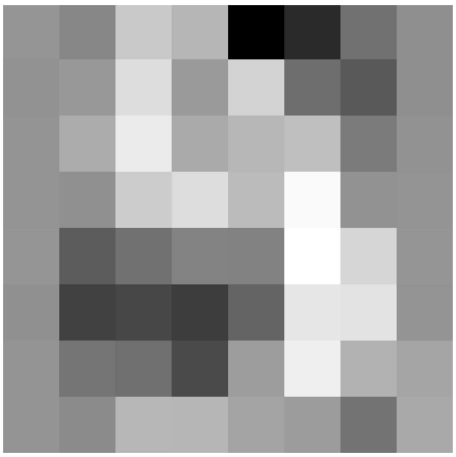
*** comparing number: 9 ***



^ original number ^

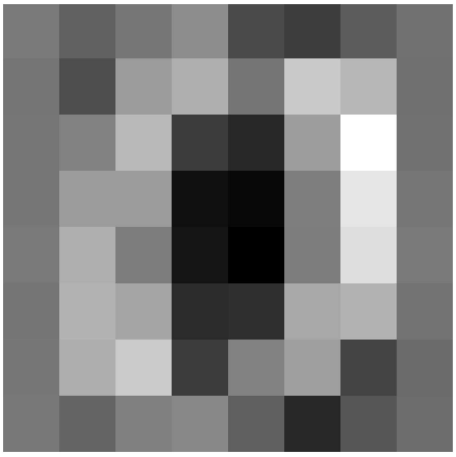


^ PCA number ^

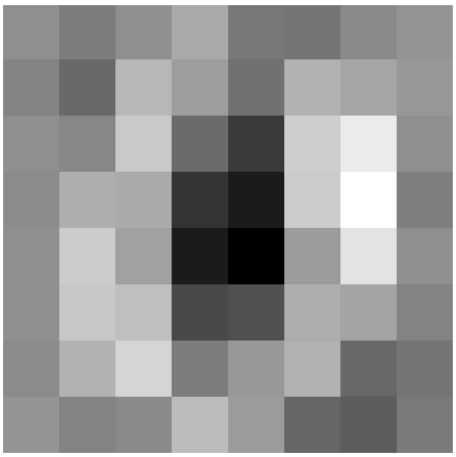


^ kernelPCA number ^

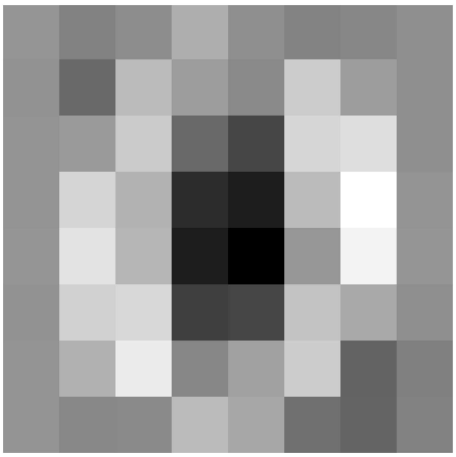
```
### Number of components: 21  ###
*** comparing number: 0 ***
```



^ original number ^

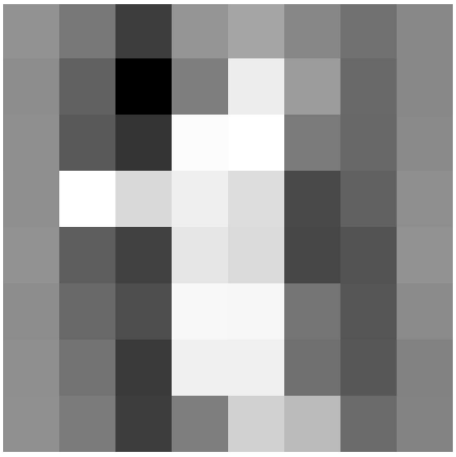


^ PCA number ^

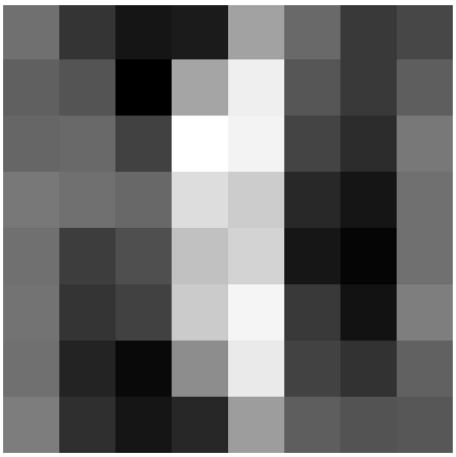


^ kernelPCA number ^

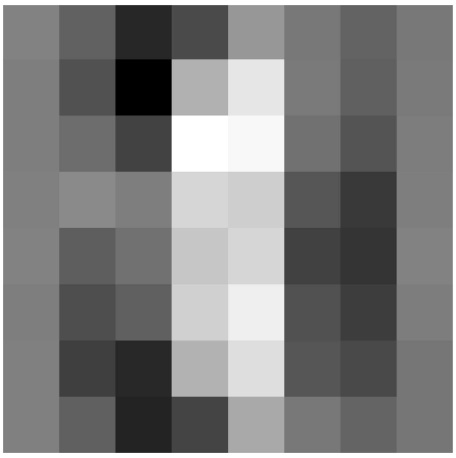
*** comparing number: 1 ***



^ original number ^

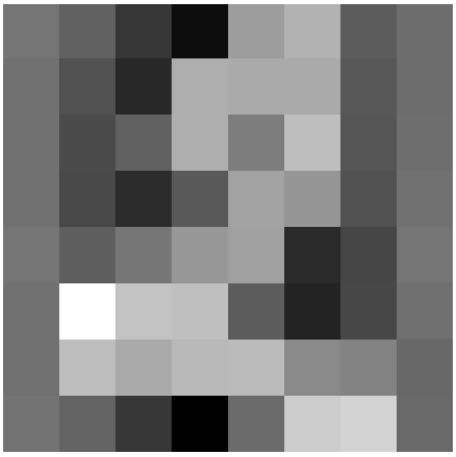


^ PCA number ^

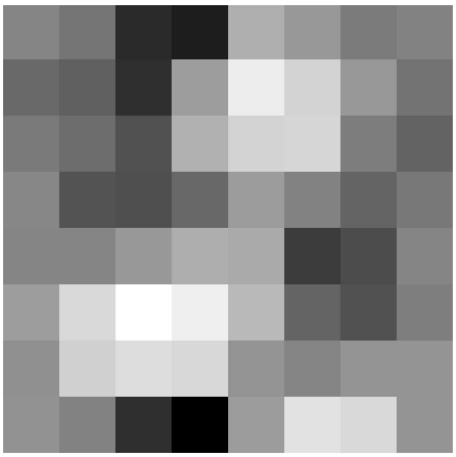


^ kernelPCA number ^

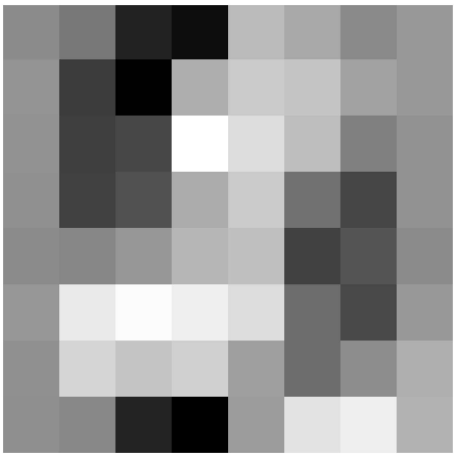
*** comparing number: 2 ***



^ original number ^

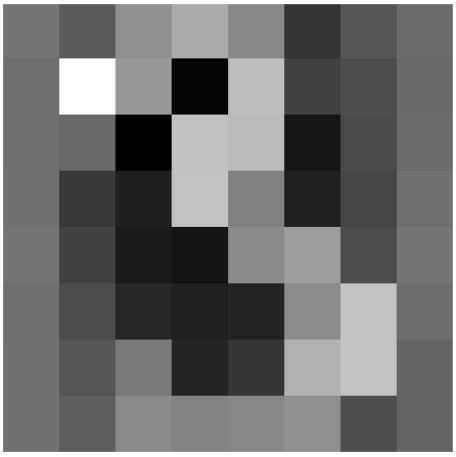


^ PCA number ^

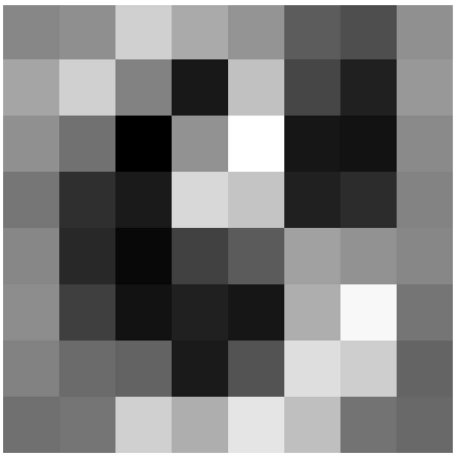


^ kernelPCA number ^

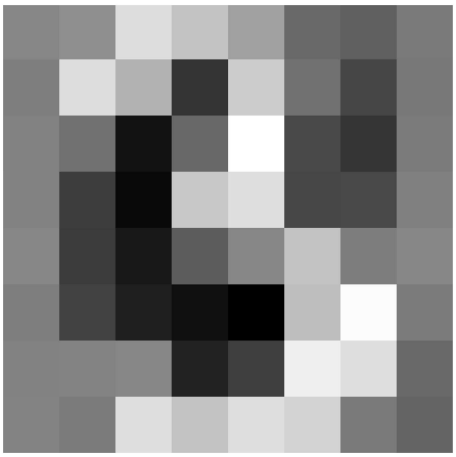
*** comparing number: 3 ***



^ original number ^

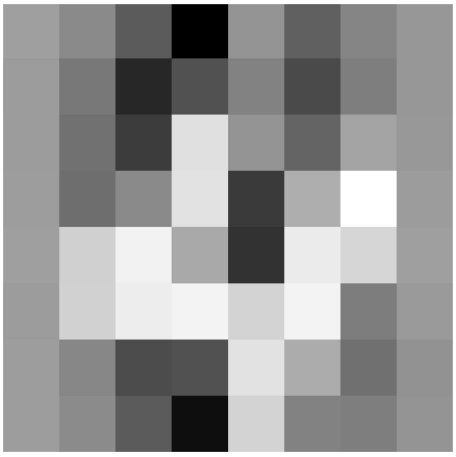


^ PCA number ^

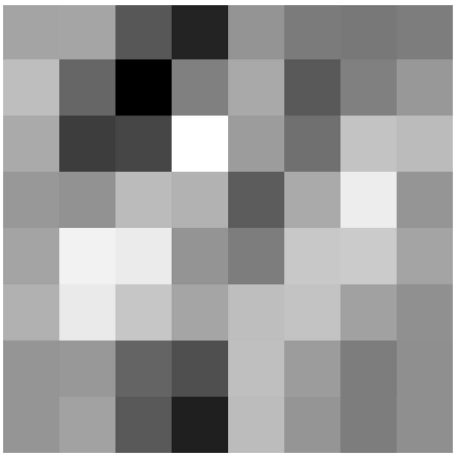


^ kernelPCA number ^

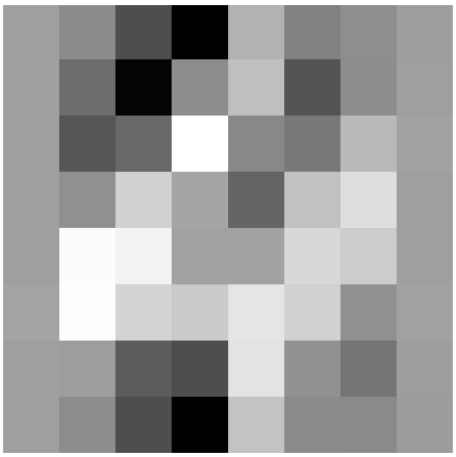
*** comparing number: 4 ***



^ original number ^

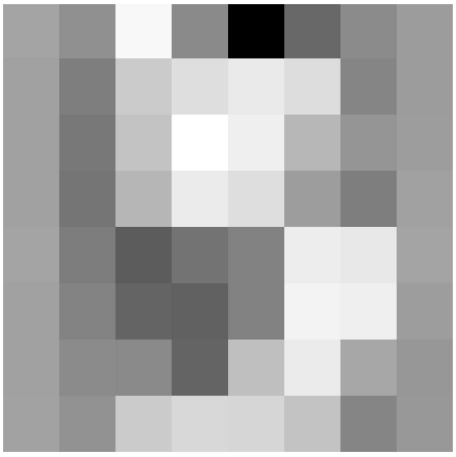


^ PCA number ^

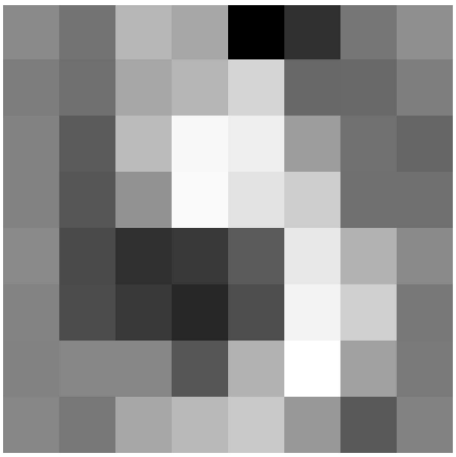


^ kernelPCA number ^

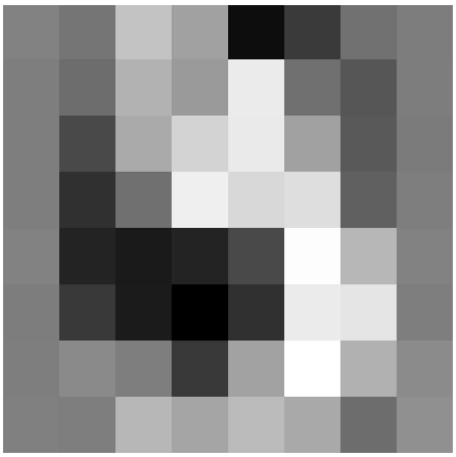
*** comparing number: 5 ***



^ original number ^

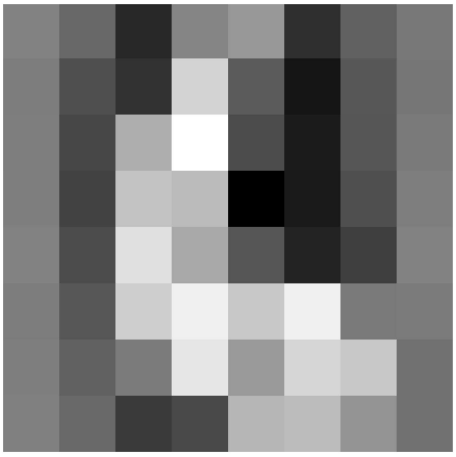


^ PCA number ^

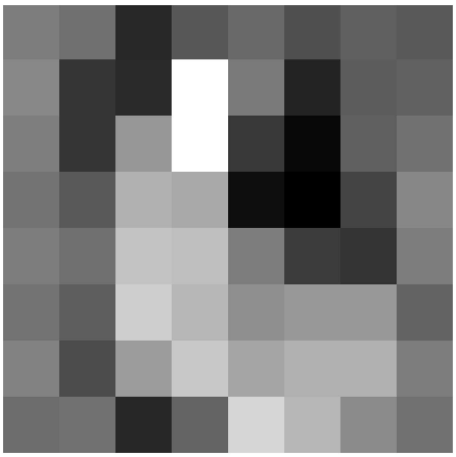


^ kernelPCA number ^

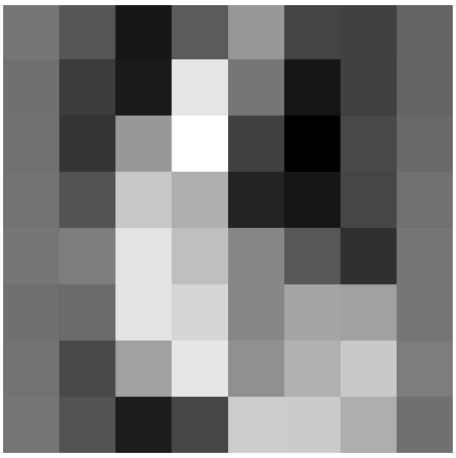
*** comparing number: 6 ***



^ original number ^

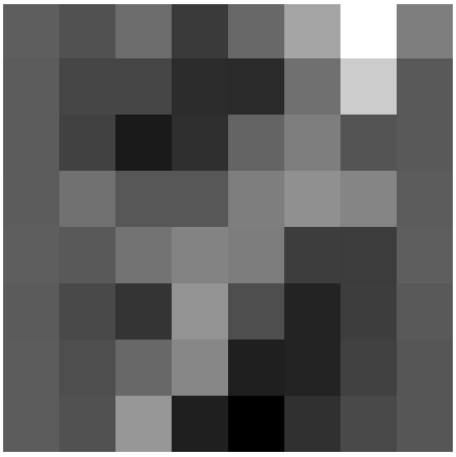


^ PCA number ^

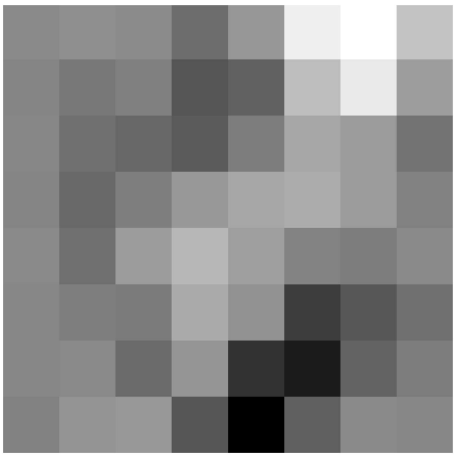


^ kernelPCA number ^

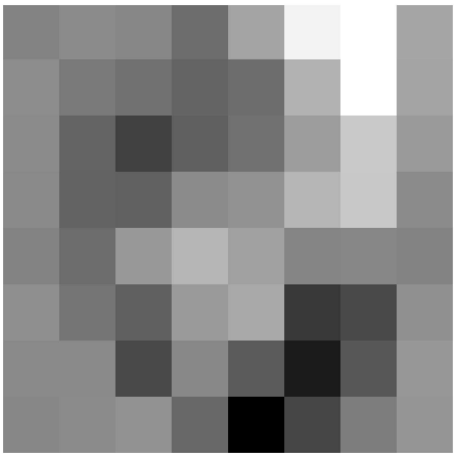
*** comparing number: 7 ***



^ original number ^

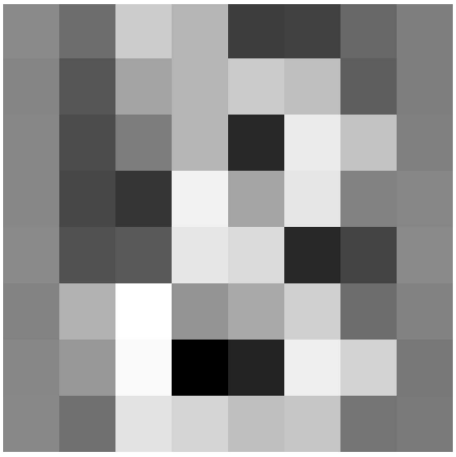


^ PCA number ^

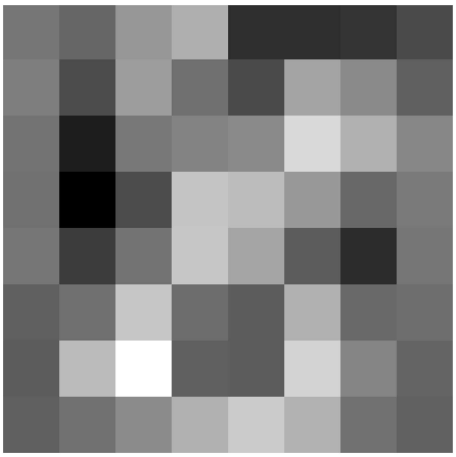


^ kernelPCA number ^

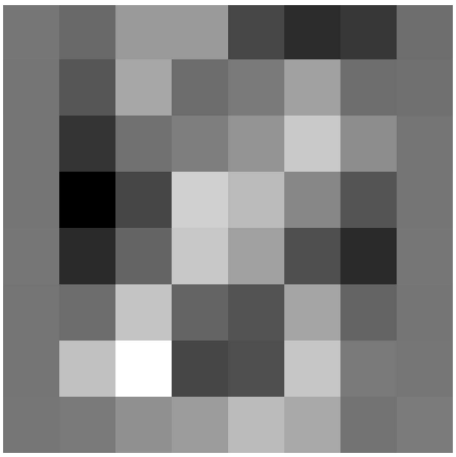
*** comparing number: 8 ***



^ original number ^

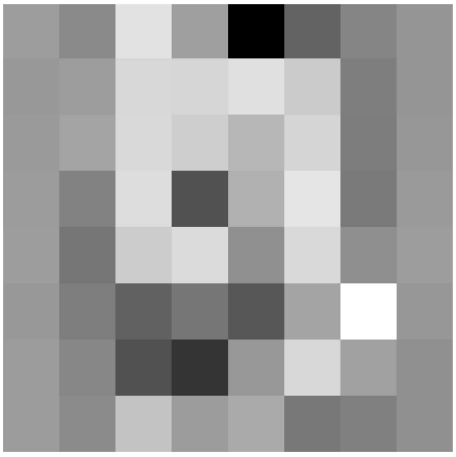


^ PCA number ^

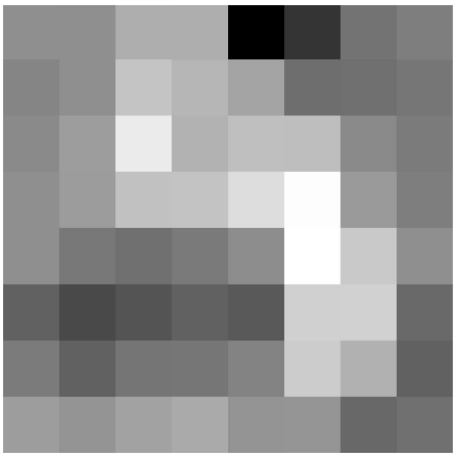


^ kernelPCA number ^

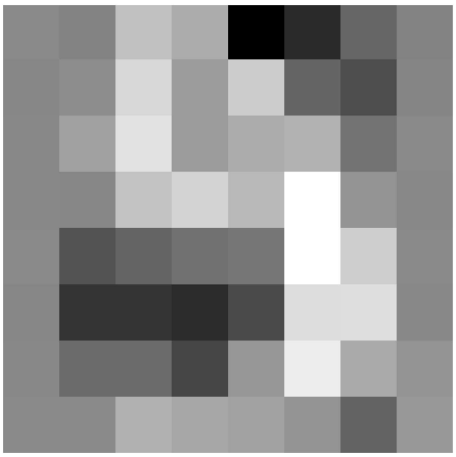
*** comparing number: 9 ***



^ original number ^

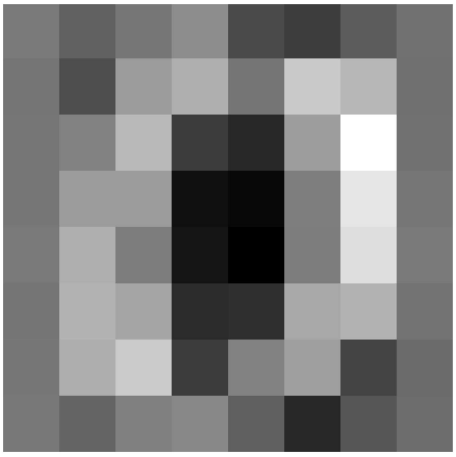


^ PCA number ^

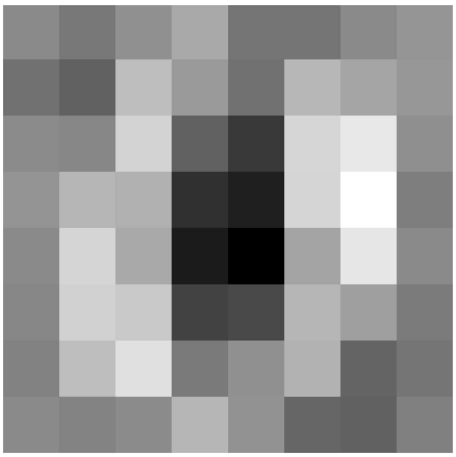


^ kernelPCA number ^

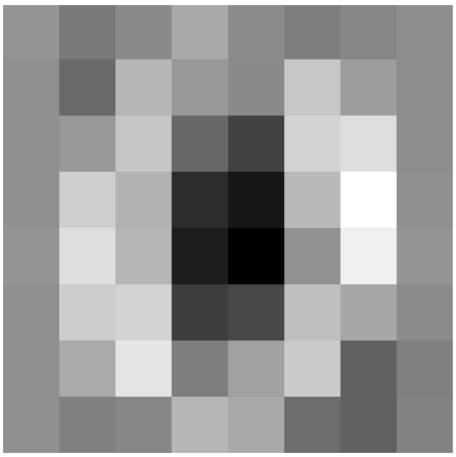
```
### Number of components: 18  ###
*** comparing number: 0 ***
```



^ original number ^

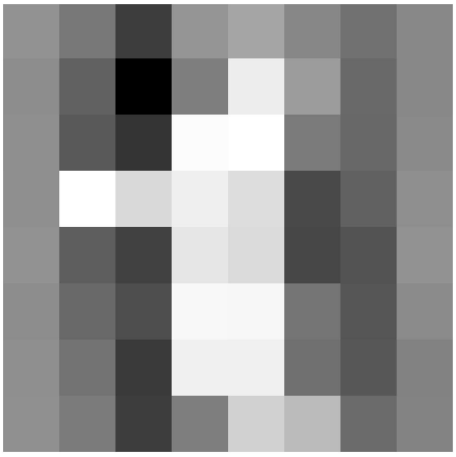


^ PCA number ^

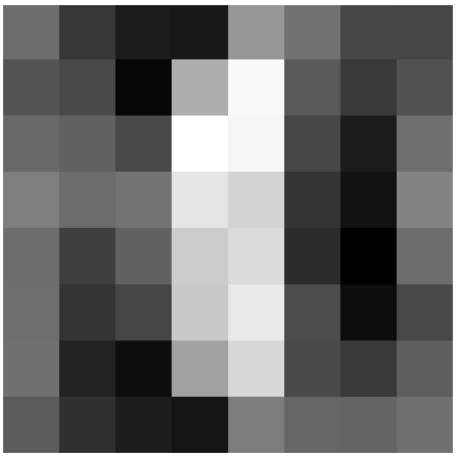


^ kernelPCA number ^

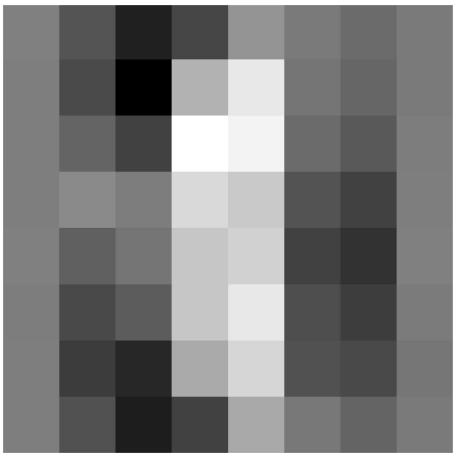
*** comparing number: 1 ***



^ original number ^

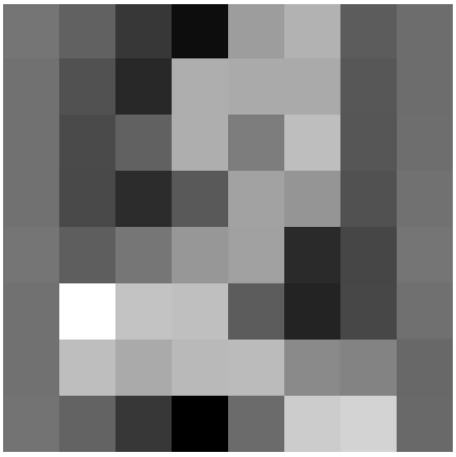


^ PCA number ^

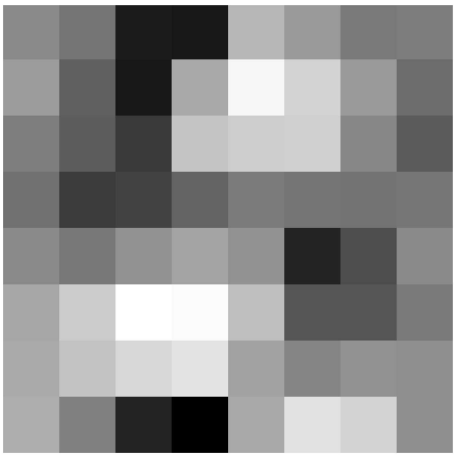


^ kernelPCA number ^

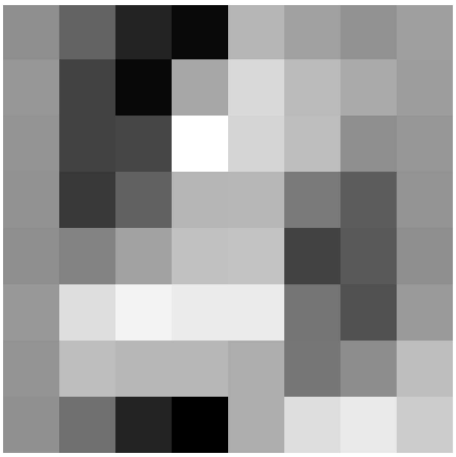
*** comparing number: 2 ***



^ original number ^

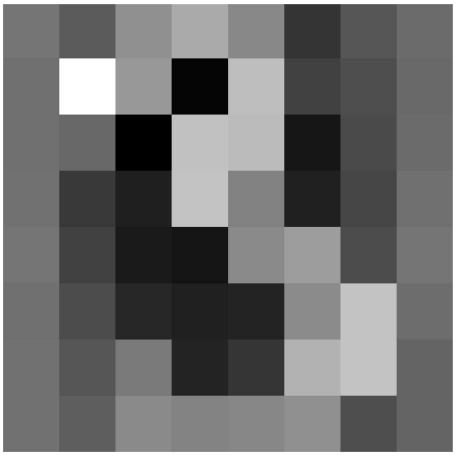


^ PCA number ^

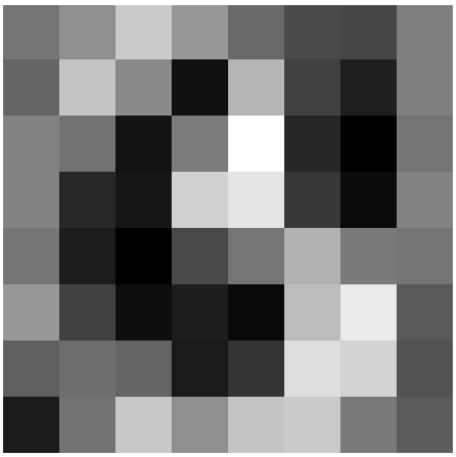


^ kernelPCA number ^

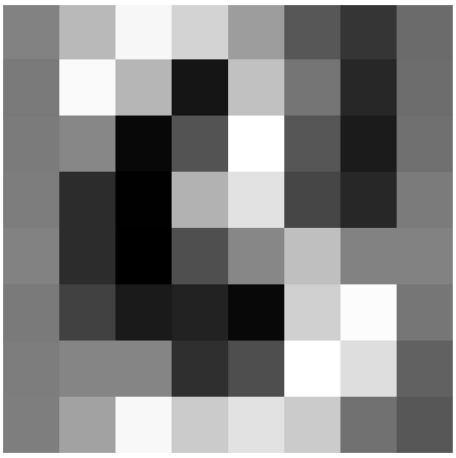
*** comparing number: 3 ***



^ original number ^

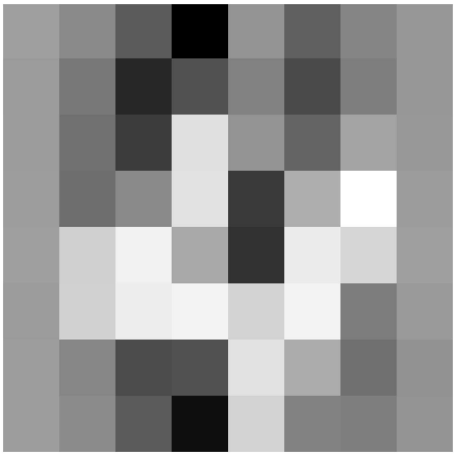


^ PCA number ^

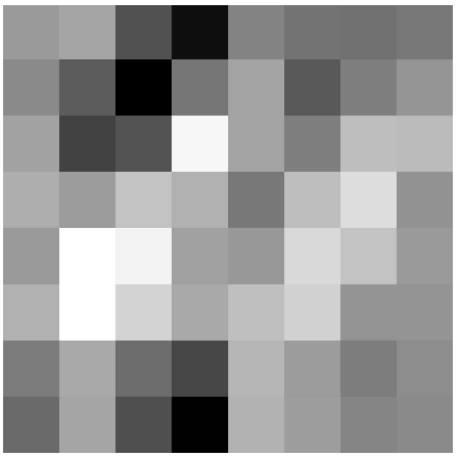


^ kernelPCA number ^

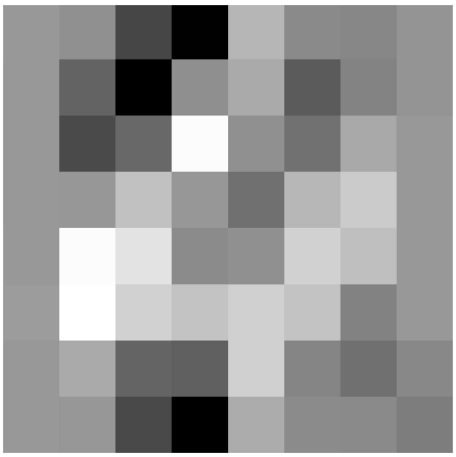
*** comparing number: 4 ***



^ original number ^

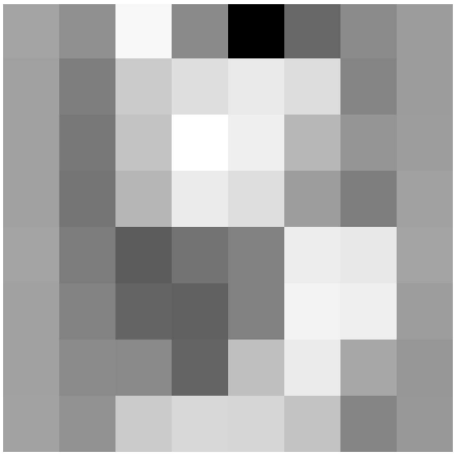


^ PCA number ^

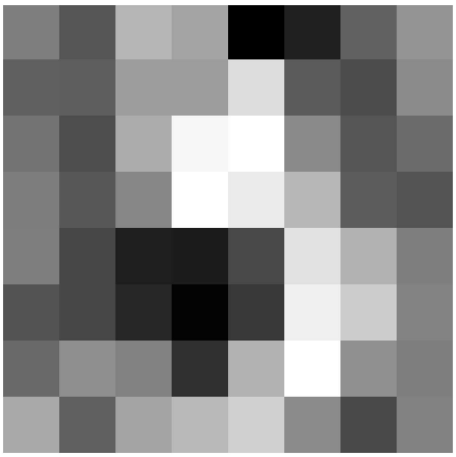


^ kernelPCA number ^

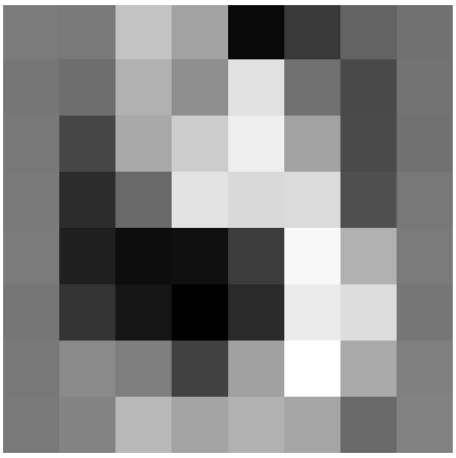
*** comparing number: 5 ***



^ original number ^

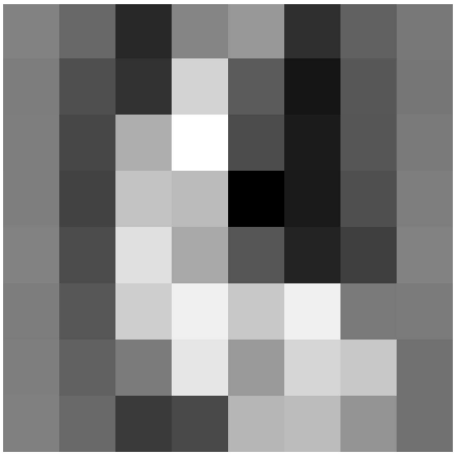


^ PCA number ^

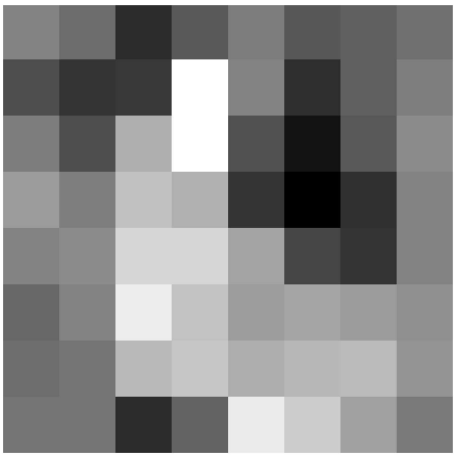


^ kernelPCA number ^

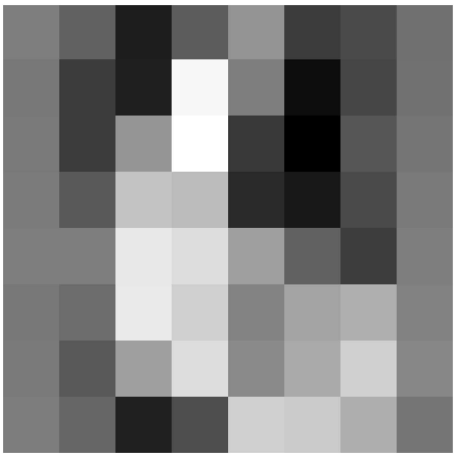
*** comparing number: 6 ***



^ original number ^

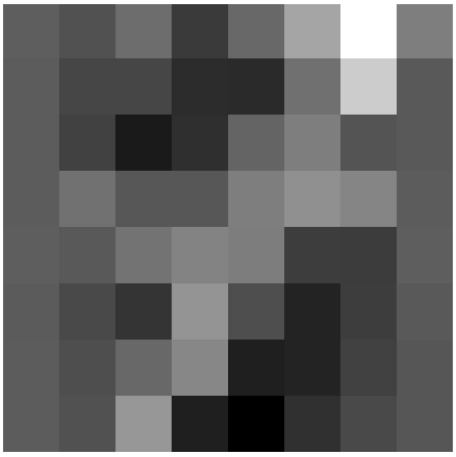


^ PCA number ^

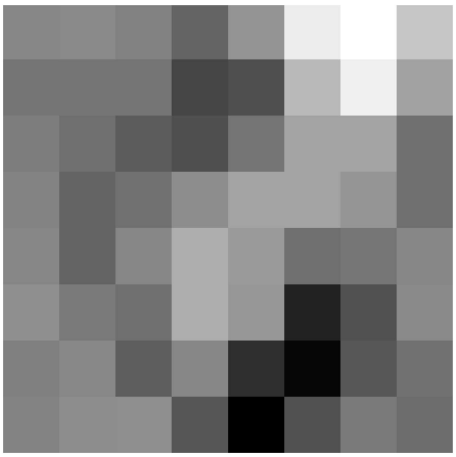


^ kernelPCA number ^

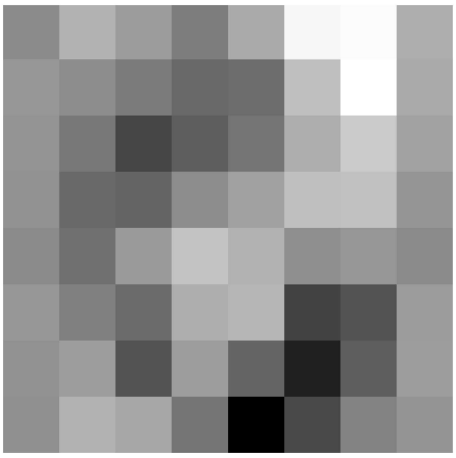
*** comparing number: 7 ***



^ original number ^

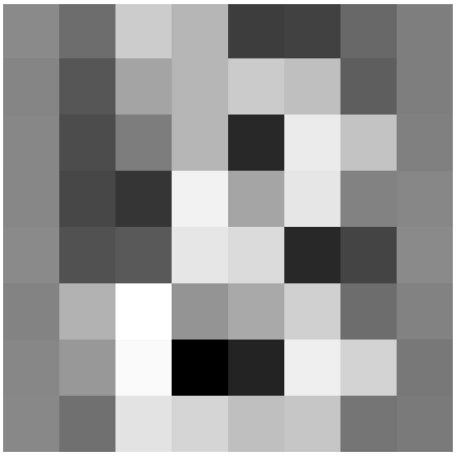


^ PCA number ^

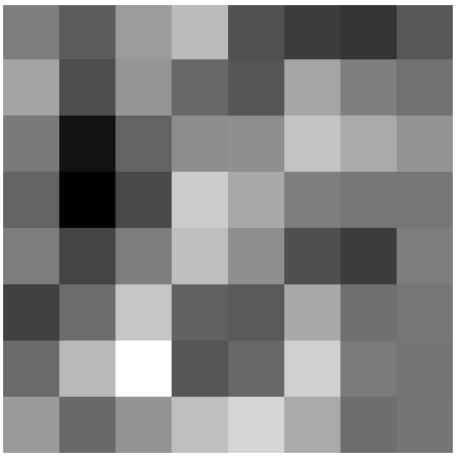


^ kernelPCA number ^

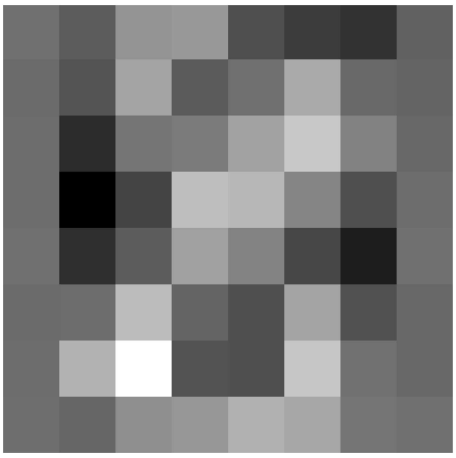
*** comparing number: 8 ***



^ original number ^

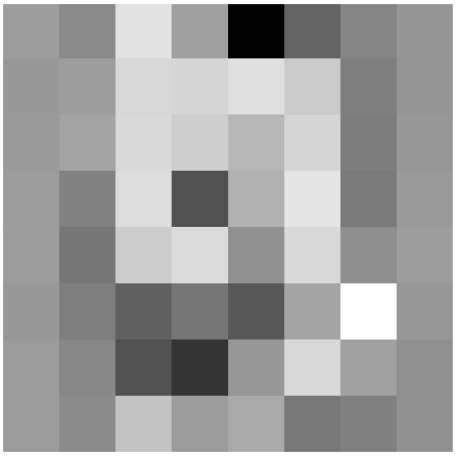


^ PCA number ^

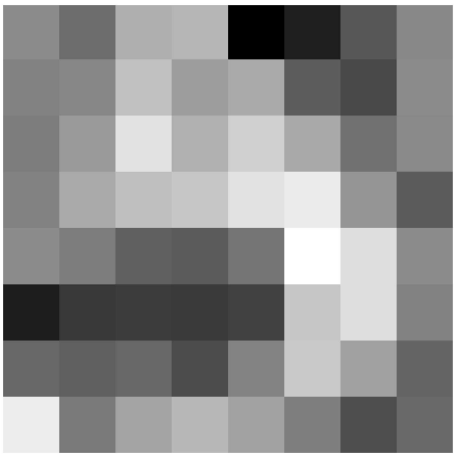


^ kernelPCA number ^

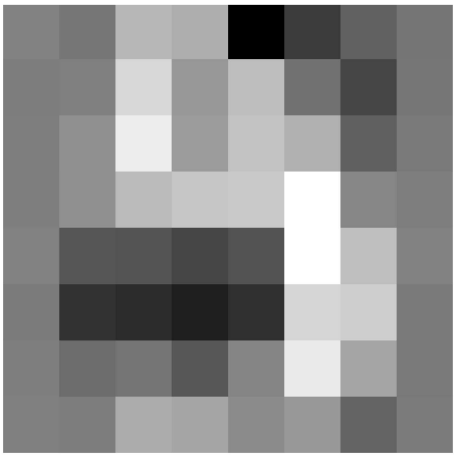
*** comparing number: 9 ***



^ original number ^



^ PCA number ^



^ kernelPCA number ^

Ciężko jednoznacznie prównać kernelPCA i PCA w przypadku zbioru Mnist wyniki zależą od bardzo zależą wyboru parametru gamma. Dla parameru 0.29 obrazy liczb otworzonych z kernelPCA i PCA są bardzo podobne. Natomiast dla parametru gamma 2.2 były dość różne. Ciężko było rozpoznać liczby odtoworzne z kernelPCA z parametrem gamma 2.2