AGH UNIVERSITY OF KRAKOW

Department of Computer Science

Project nr 2
Visualization of structured and unstructured data

"PCA kernelPCA"

Author: *Jacek Tyszkiewicz*
Degree programme: *Computer Science*
Supervisor:

Krakow, 2025

Table of Contents

1. PCA kernelPCA

1.1. Perform PCA for Breast Cancer Dataset (3 points)

– You can find this dataset in the scikit-learn library, import it and convert to pandas dataframe. Original labels are ‘0‘ and ‘1‘; for better readability change these names to: ‘benign‘ and ‘malignant‘.

– Visualize correlations between pairs of features (due to the greater number of features use pandas `corr()` function instead of `pairplot` and seaborn `heatmap()`).

– Examine (zbadaj) explained variance, draw a plot showing relation between total explained variance and number of principal components used.

– Use recursive feature elimination (available in scikit-learn module) or another feature ranking algorithm to split 30 features into 15 “more important” and 15 “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.

– You can find this dataset in the scikit-learn library, import it and convert to pandas dataframe. Original labels are ‘0‘ and ‘1‘; for better readability change these names to: ‘benign‘ and ‘malignant‘.

```
from IPython.display import display
pd.set_option('display.max_columns', None)
pd.set_option('display.width', None)
pd.set_option('display.max_colwidth', None)
```

```
from sklearn.datasets import load_breast_cancer
import pandas as pd

data = load_breast_cancer()
df = pd.DataFrame(data.data, columns=data.feature_names)


df['target'] = data.target
df['target'] = df['target'].map({0: 'malignant', 1: 'benign'})

display(df.head())
```

```
corr_matrix = df.drop(columns='target').corr()

```

```
plt.figure(figsize=(16, 12))
sns.heatmap(corr_matrix, annot=False, cmap='coolwarm', center=0)
plt.title('Feature Correlation Matrix (Breast Cancer Dataset) ')
plt.show()
```

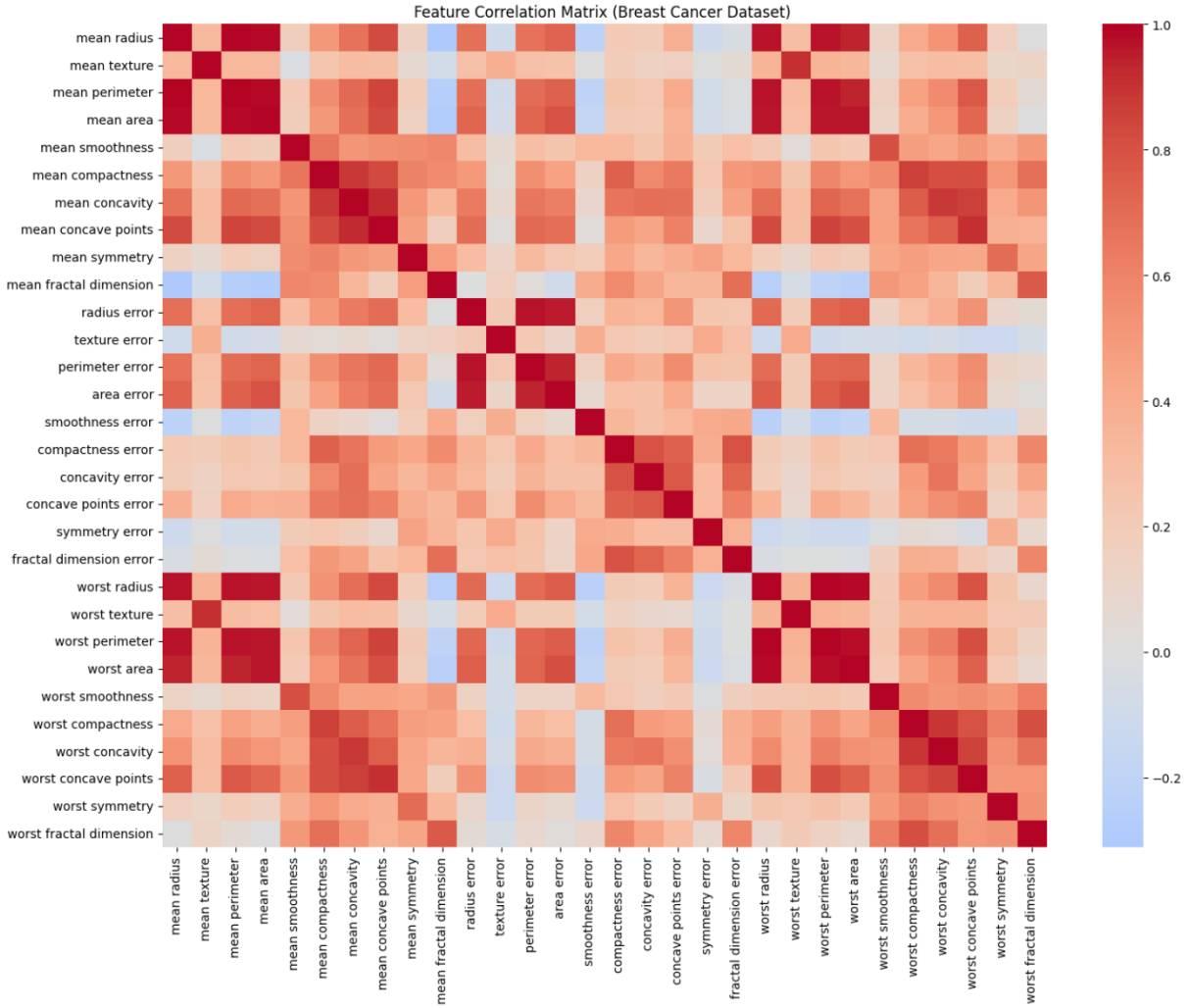

Fig. 1.1. Feature Heatmap

```
X = data.data

X_scaled = StandardScaler().fit_transform(X)

pca = PCA()
X_pca = pca.fit_transform(X_scaled)

explained_variance_ratio = pca.explained_variance_ratio_
cumulative_variance = np.cumsum(explained_variance_ratio)

plt.figure(figsize=(10, 6))
plt.plot(range(1, len(cumulative_variance)+1), cumulative_variance, marker='o',
    linestyle='--')
plt.title('Explained variance for number of principal components used')
plt.xlabel('number of principal components')
plt.ylabel('cumulative explained variance')
plt.grid(True)
plt.show()
```

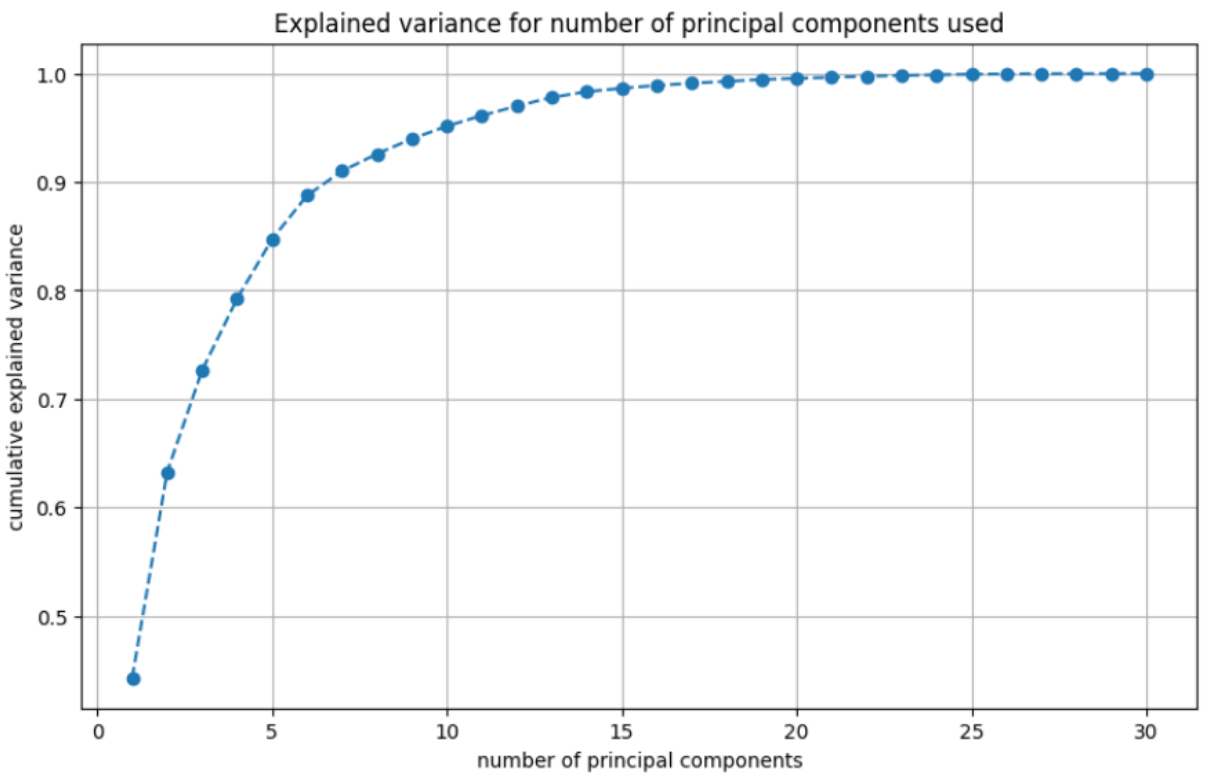

Fig. 1.2. Feature Heatmap

```
X = data.data
y = data.target

X = StandardScaler().fit_transform(X)

estimator = LogisticRegression(max_iter=1000)
selector = RFE(estimator, n_features_to_select=15)
selector.fit(X, y)

X_all = X
X_top15 = X[:, selector.support_]
X_bottom15 = X[:, ~selector.support_]


def explained_variance(X_data):
    pca = PCA().fit(X_data)
    return np.cumsum(pca.explained_variance_ratio_)

var_all = explained_variance(X_all)
var_top15 = explained_variance(X_top15)
var_bottom15 = explained_variance(X_bottom15)


plt.figure(figsize=(8, 5))
plt.plot(range(1, len(var_all) + 1), var_all, label="All features (30)")
plt.plot(range(1, len(var_top15) + 1), var_top15, label="Top 15 features (RFE)"
    )
plt.plot(range(1, len(var_bottom15) + 1), var_bottom15, label="Bottom 15
    features (RFE)")

plt.xlabel("number of principal components")
plt.ylabel("cumulative explained variance")
plt.title("PCA: Explained Variance (Breast Cancer Dataset)")
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```

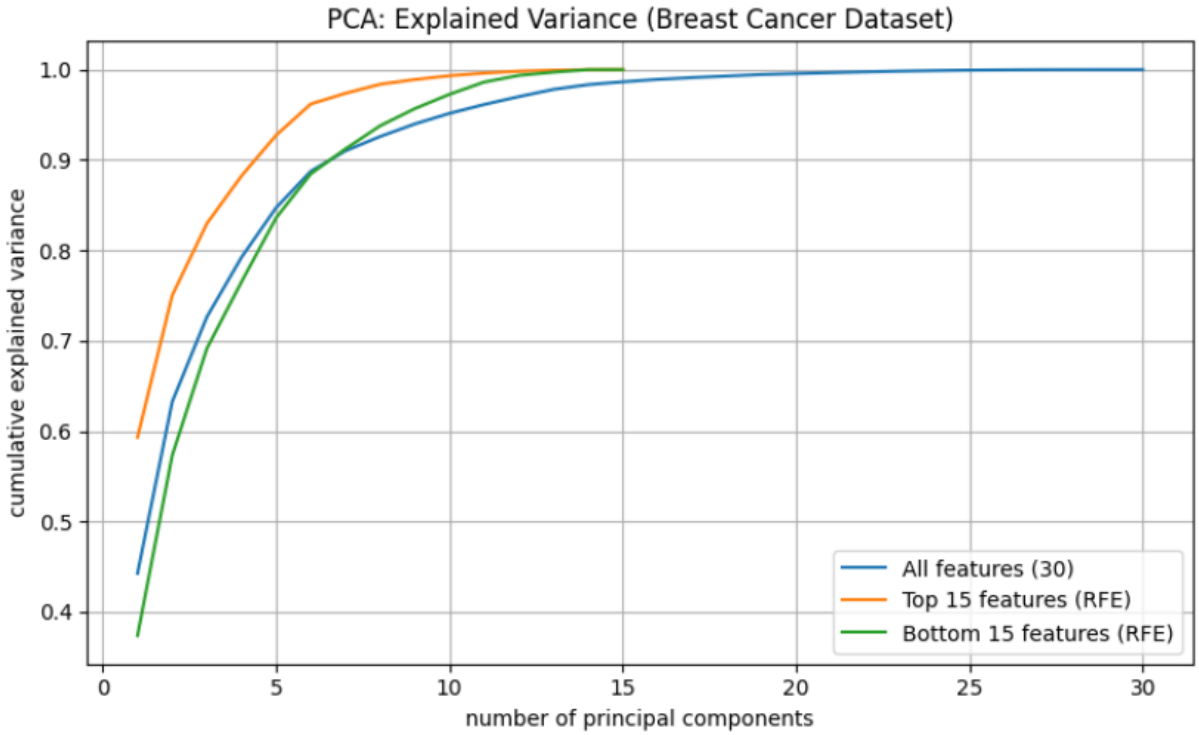

Fig. 1.3. Feature Heatmap

1.2. KernelPCA (3 points)

– Visualize in 2D datasets used in this lab, experiment with the parameters of the KernelPCA method. Change `kernel` and `gamma` params.

Documentation: https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.KernelPCA.html

```
data = load_breast_cancer()
X = data.data
y = data.target
X_scaled = StandardScaler().fit_transform(X)

gammas = [0.001, 0.01, 0.1, 1, 10]

fig, axes = plt.subplots(1, len(gammas), figsize=(18, 4))
fig.suptitle("KernelPCA (RBF) - effect of the gamma parameter", fontsize=16)

# Pętla po gamma
for i, gamma in enumerate(gammas):
    kpca = KernelPCA(n_components=2, kernel='rbf', gamma=gamma)
    X_kpca = kpca.fit_transform(X_scaled)

    ax = axes[i]
    ax.scatter(X_kpca[:, 0], X_kpca[:, 1], c=y, cmap='coolwarm', s=10)
    ax.set_title(f"gamma = {gamma}")
    ax.set_xticks([])
    ax.set_yticks([])

plt.tight_layout(rect=[0, 0, 1, 0.93])
plt.show()
```

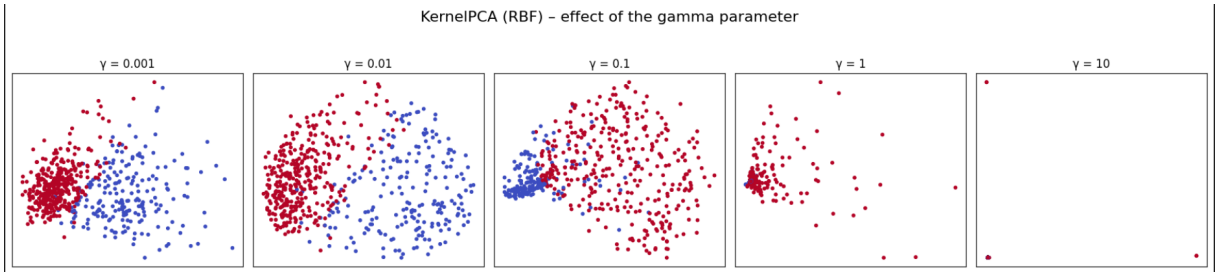

Fig. 1.4. Feature Heatmap

```python
gammas = [0.00001,0.0001, 0.001, 0.01, 0.1]
degrees = [2, 3, 5,6,7]


fig, axes = plt.subplots(len(degrees), len(gammas), figsize=(16, 9))
fig.suptitle("KernelPCA (poly) - influence of gamma and polynomial degree",
    fontsize=16)

for i, degree in enumerate(degrees):
    for j, gamma in enumerate(gammas):
        kpca = KernelPCA(n_components=2, kernel='poly', gamma=gamma, degree=
            degree, coef0=1)
        X_kpca = kpca.fit_transform(X_scaled)

        ax = axes[i, j]
        ax.scatter(X_kpca[:, 0], X_kpca[:, 1], c=y, cmap='coolwarm', s=10)
        ax.set_title(f"gamma={gamma}, deg={degree}")
        ax.set_xticks([])
        ax.set_yticks([])

plt.tight_layout(rect=[0, 0, 1, 0.95])
plt.show()
```

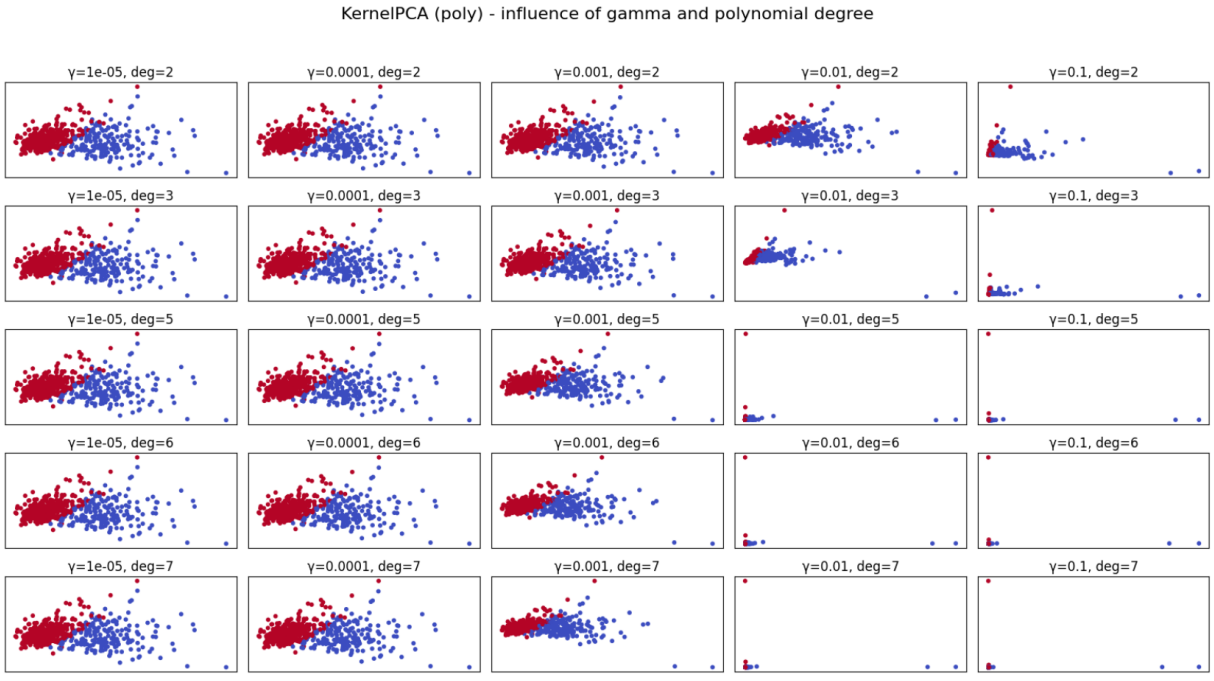

Fig. 1.5. Feature Heatmap

```
kpca = KernelPCA(n_components=2, kernel='cosine')
X_kpca = kpca.fit_transform(X_scaled)


plt.figure(figsize=(6, 5))
plt.scatter(X_kpca[:, 0], X_kpca[:, 1], c=y, cmap='coolwarm', s=15)
plt.title("KernelPCA - kernel='cosine'")
plt.xlabel("PC1")
plt.ylabel("PC2")
plt.grid(True)
plt.tight_layout()
plt.show()
```

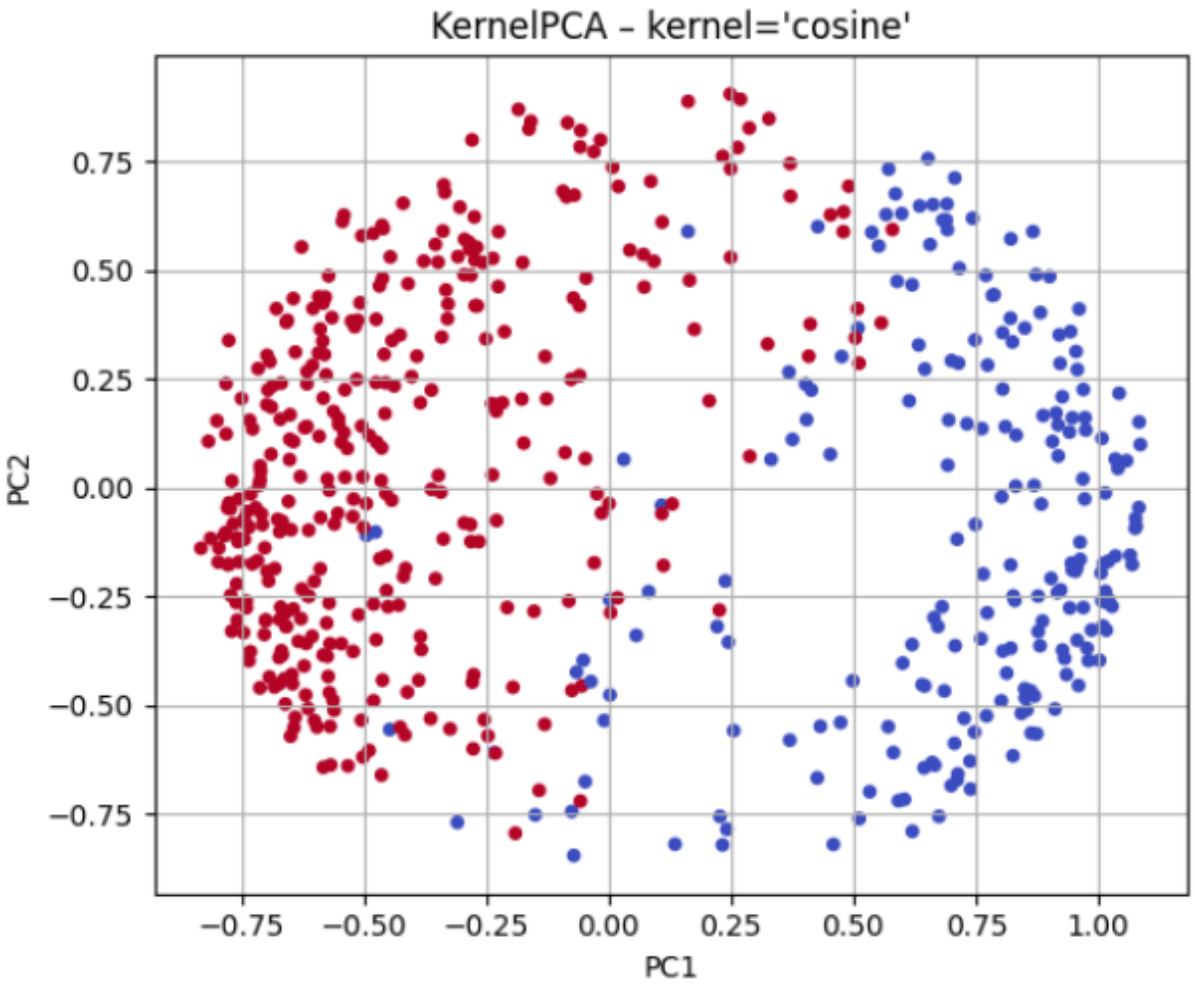

Fig. 1.6. Feature Heatmap

```
kpca = KernelPCA(n_components=2, kernel='linear')
X_kpca = kpca.fit_transform(X_scaled)


plt.figure(figsize=(6, 5))
plt.scatter(X_kpca[:, 0], X_kpca[:, 1], c=y, cmap='coolwarm', s=15)
plt.title("KernelPCA - kernel='linear'")
plt.xlabel("PC1")
plt.ylabel("PC2")
plt.grid(True)
plt.tight_layout()
plt.show()
```

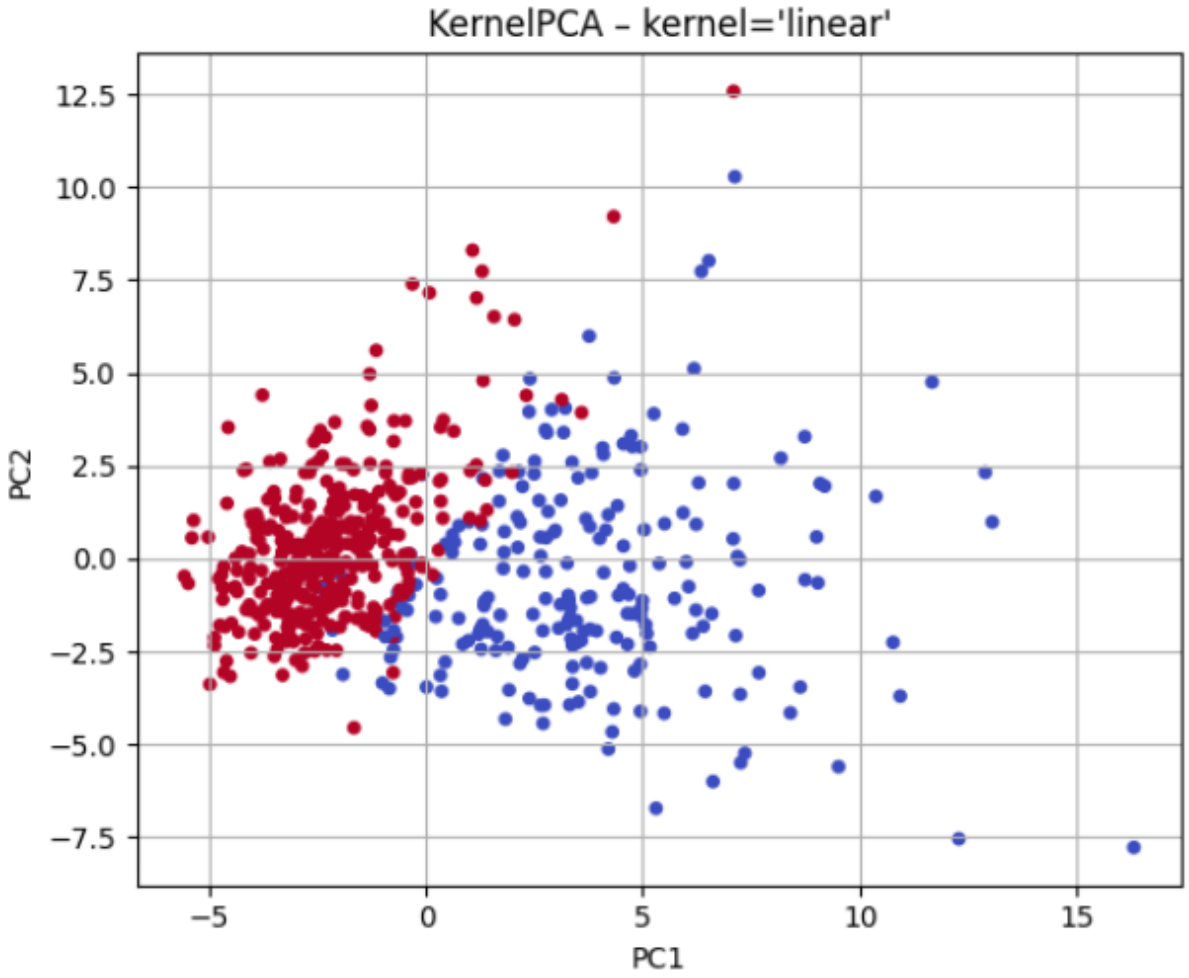

Fig. 1.7. Feature Heatmap

```python
gammas = [0.001, 0.01, 0.1, 1]
coef0s = [-1, 0, 1]


fig, axes = plt.subplots(len(coef0s), len(gammas), figsize=(16, 9))
fig.suptitle("Effect of gamma and coef0 in KernelPCA (sigmoid)", fontsize=16)

for i, coef0 in enumerate(coef0s):
    for j, gamma in enumerate(gammas):
        kpca = KernelPCA(n_components=2, kernel='sigmoid', gamma=gamma, coef0=
            coef0)
        X_kpca = kpca.fit_transform(X_scaled)

        ax = axes[i, j]
        ax.scatter(X_kpca[:, 0], X_kpca[:, 1], c=y, cmap='coolwarm', s=10)
        ax.set_title(f"gamma={gamma}, coef0={coef0}")
        ax.set_xticks([])
        ax.set_yticks([])

plt.tight_layout(rect=[0, 0, 1, 0.95])
plt.show()
```

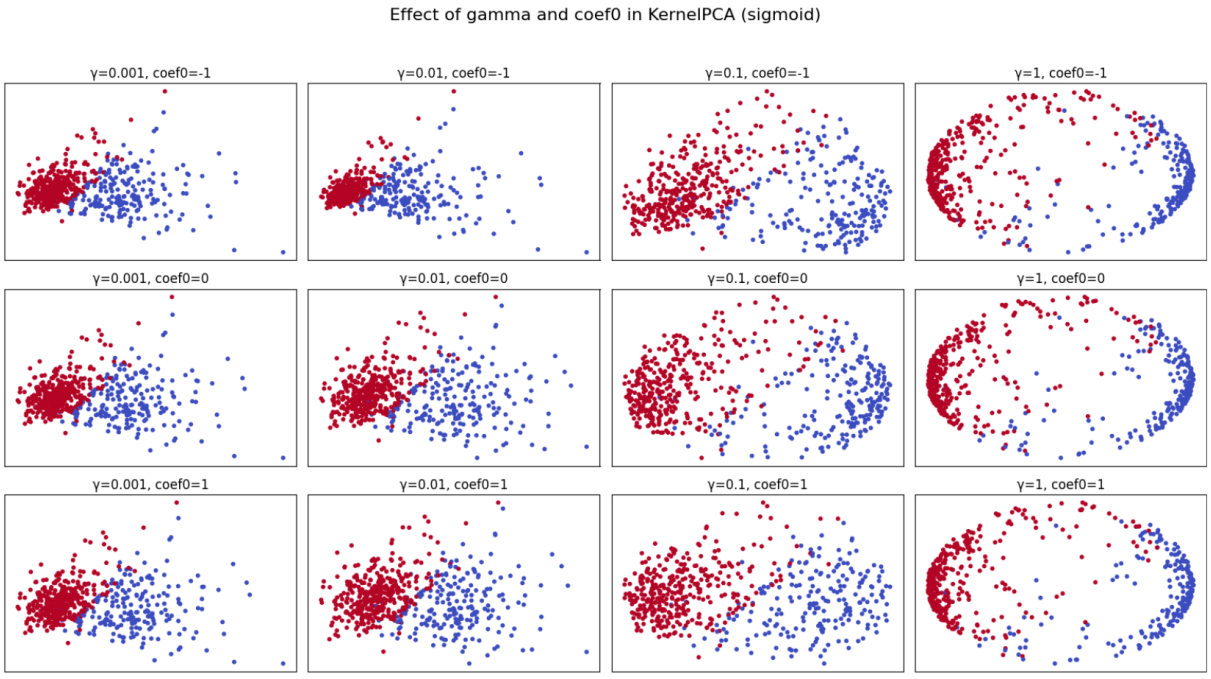

Fig. 1.8. Feature Heatmap

1.3. Classic PCA Analysis on MNIST (4 points)

- Download the MNIST data set (there is a function to load it 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 do its original space. Make an inverse (odwrotną) transformation for number of components (komponentów) corresponding (odpowiadających) 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).

1.3.1. Important:

Write a brief analysis (200–500 words) addressing:

1. **What are the key strengths and limitations of the PCA variant you explored?**
2. **In what real-world scenarios would this approach be particularly useful?**
3. **How does the computational complexity scale with dataset size?**
4. **What improvements or modifications would you suggest to the method?**

```
from sklearn.datasets import fetch_openml
mnist = fetch_openml('mnist_784', version=1, as_frame=False)


X, y = mnist.data, mnist.target.astype(int)


X_scaled = StandardScaler().fit_transform(X)

X_sample = X_scaled[:2000]
y_sample = y[:2000]

pca = PCA(n_components=2)
X_pca = pca.fit_transform(X_sample)

kpca = KernelPCA(n_components=2, kernel='rbf', gamma=0.01)
X_kpca = kpca.fit_transform(X_sample)

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 6))

scatter1 = ax1.scatter(X_pca[:, 0], X_pca[:, 1], c=y_sample, cmap='tab10', s=3)
ax1.set_title("PCA - MNIST projection to 2D")

scatter2 = ax2.scatter(X_kpca[:, 0], X_kpca[:, 1], c=y_sample, cmap='tab10', s
    =3)
ax2.set_title("KernelPCA (rbf) - MNIST projection to 2D")

plt.tight_layout()
plt.show()
```

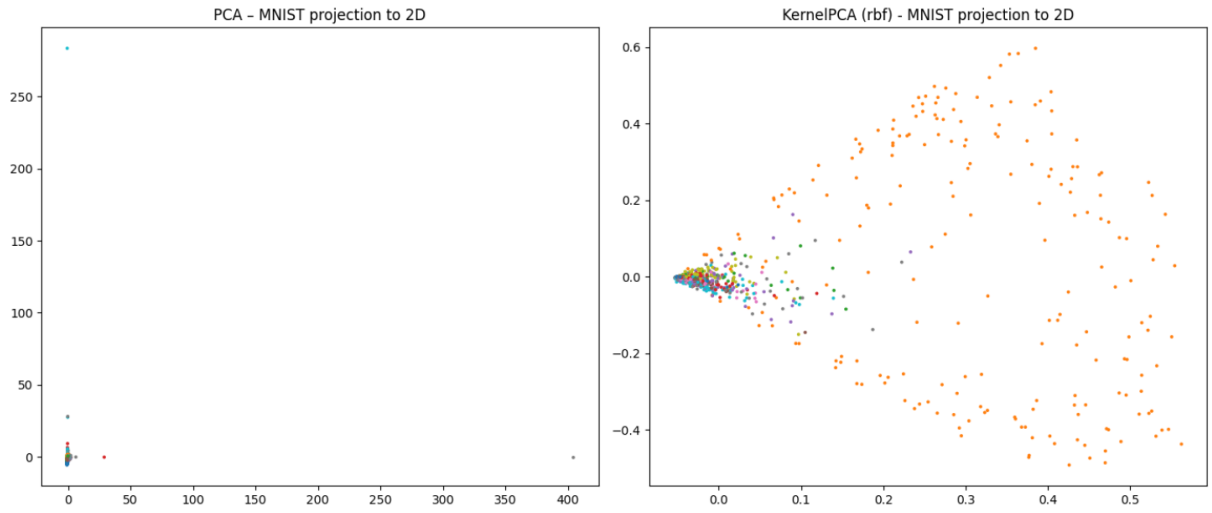

Fig. 1.9. Feature Heatmap

```python
explained_variance = np.cumsum(pca.explained_variance_ratio_)


progi = [0.85, 0.90, 0.95, 0.99]
for prog in progi:
    liczba = np.argmax(explained_variance >= prog) + 1
    print(f'Dla    {int(prog*100)}% wariancji potrzeba: {liczba} komponentów')

plt.figure(figsize=(10, 5))
plt.plot(range(1, len(explained_variance) + 1), explained_variance, marker='o')
plt.title('Skumulowana wariancja wyjaśniona przez PCA')
plt.xlabel('Liczba głównych składowych')
plt.ylabel('Skumulowana wyjaśniona wariancja')
plt.grid(True)
plt.tight_layout()
plt.show()
```

```
To explain ≥ 85% of variance, you need: 1 components
To explain ≥ 90% of variance, you need: 1 components
To explain ≥ 95% of variance, you need: 1 components
To explain ≥ 99% of variance, you need: 1 components
```

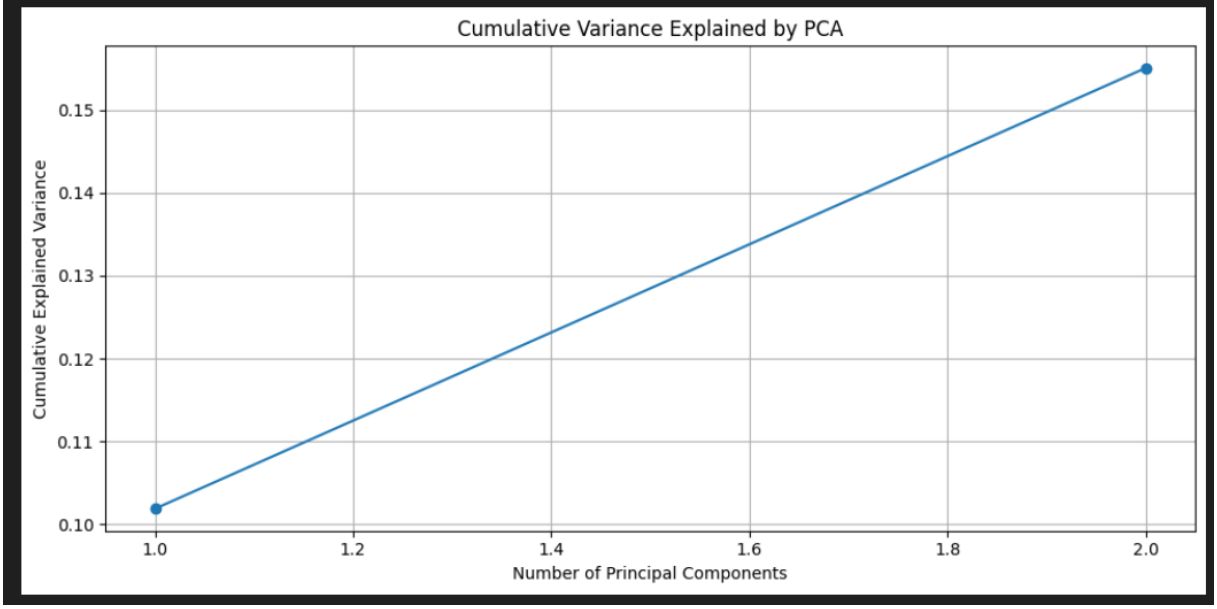

Fig. 1.10. Feature Heatmap

```
X = mnist.data[:10]
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

n_components = 10
pca = PCA(n_components=n_components)
X_reduced = pca.fit_transform(X_scaled)
X_reconstructed = pca.inverse_transform(X_reduced)

X_original = scaler.inverse_transform(X_scaled)
X_reconstructed = scaler.inverse_transform(X_reconstructed)

fig, axes = plt.subplots(2, 10, figsize=(15, 4))
for i in range(10):
    axes[0, i].imshow(X_original[i].reshape(28, 28), cmap="gray")
    axes[0, i].axis("off")
    axes[0, i].set_title("Original")

    axes[1, i].imshow(X_reconstructed[i].reshape(28, 28), cmap="gray")
    axes[1, i].axis("off")
    axes[1, i].set_title(f"{n_components} PCs")

plt.suptitle("MNIST image reconstruction using PCA")
plt.tight_layout()
plt.show()
```

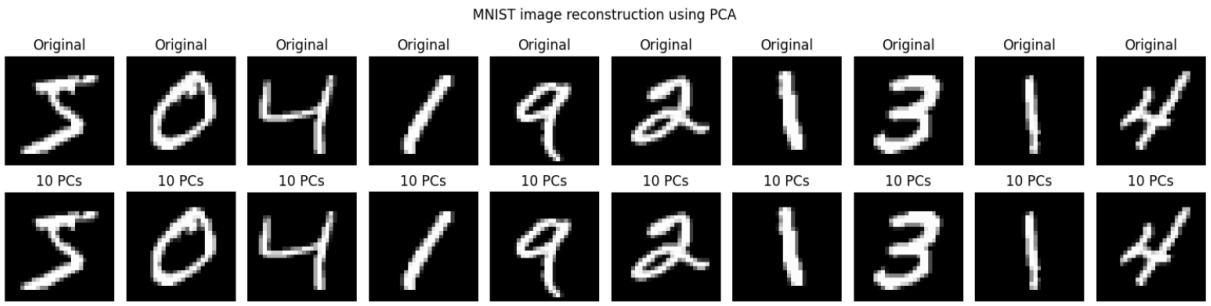

Fig. 1.11. Feature Heatmap

```
X = mnist.data[:10]
y = mnist.target[:10].astype(int)

scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

kpca = KernelPCA(n_components=10, kernel='rbf', gamma=0.01,
    fit_inverse_transform=True)
X_reduced = kpca.fit_transform(X_scaled)
X_reconstructed = scaler.inverse_transform(kpca.inverse_transform(X_reduced))

fig, axes = plt.subplots(2, 10, figsize=(15, 3))
for i in range(10):
    axes[0, i].imshow(X[i].reshape(28, 28), cmap='gray')
    axes[0, i].axis('off')
    axes[0, i].set_title(f"Label: {y[i]}")

    axes[1, i].imshow(X_reconstructed[i].reshape(28, 28), cmap='gray')
    axes[1, i].axis('off')

plt.suptitle("KernelPCA: Original and Reconstruction (RBF, gamma=0.01)",
    fontsize=14)
plt.tight_layout()
plt.show()
```

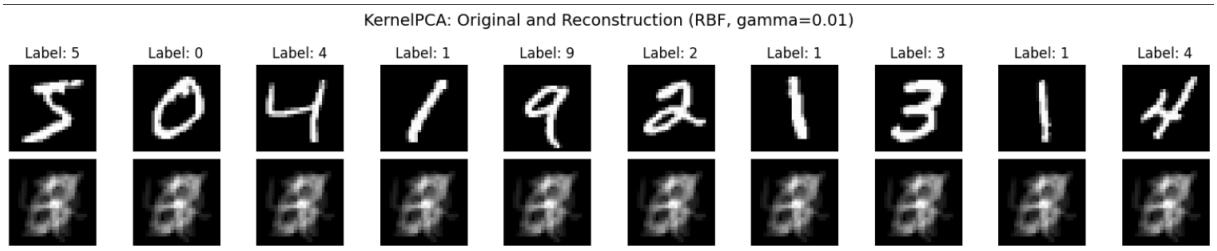

Fig. 1.12. Feature Heatmap

Important:

Write a brief analysis (200–500 words) addressing:

1. **What are the key strengths and limitations of the PCA variant you explored?**
2. **In what real-world scenarios would this approach be particularly useful?**
3. **How does the computational complexity scale with dataset size?**
4. **What improvements or modifications would you suggest to the method?**

Key strengths of the PCA variant I explored is that it extends traditional PCA to handle non-linear dimensionality reduction by implicitly mapping data into a higher-dimensional feature space. This is especially useful when the data is not linearly separable. The main limitation of this method is its high computational complexity. For example, when I tried to use the full MNIST dataset, the kernel matrix (K) had a size of $70{,}000 \times 70{,}000$, which is extremely memory-intensive.

This approach is especially useful in fields like image processing or finance, where datasets often contain many features and exhibit non-linear relationships. It is also valuable for visualizing high-dimensional data in two or three dimensions.

The computational complexity of PCA is determined by the algorithm used to compute the eigenvalues — in `scikit-learn` implementations, this is based on Singular Value Decomposition (SVD). For a data matrix $X \in \mathbb{R}^{n \times p}$, the complexity of full SVD is:

- $\mathcal{O}(np^2)$ when $n \geq p$
- $\mathcal{O}(n^2 p)$ when $p > n$

In the case of **Kernel PCA**, the method first computes a kernel (similarity) matrix $K \in \mathbb{R}^{n \times n}$, where n is the number of samples. This leads to:

- **Memory complexity:** $\mathcal{O}(n^2)$
- **Time complexity** for eigen-decomposition: up to $\mathcal{O}(n^3)$

I think we should reduce the time complexity, but it's not easy. One possible solution is to use a lower number of samples from the original data, but they should be representative. Another modification I would suggest is automatic hyperparameter optimization, which could be performed using methods like Bayesian Optimization (BOGP), Tree-structured Parzen Estimator (TPE), SMAC, or genetic algorithms. However, this requires defining a loss function.

(*) Optional: 4. Modern PCA Applications (2 points)

Choose **ONE** of the following advanced tasks:

Option A: Robust PCA for Noisy Image Reconstruction

1. Take 10 sample images from MNIST
2. Add random noise and some structured outliers (e.g., salt and pepper noise)
3. Apply both standard PCA and Robust PCA for denoising (odsłumiania)
4. Compare the results quantitatively (ilościowo) using metrics like PSNR, SSIM and qualitatively (jakościowo)

Option B: Incremental PCA for Large Datasets

1. Load the Fashion-MNIST dataset (a more complex alternative to MNIST)
2. Implement a streaming data scenario by processing the dataset in small batches
3. Apply Incremental PCA and compare its performance (both in terms of computation time and accuracy) with standard PCA
4. Visualize how the principal components evolve as more data is processed

Option C: PCA in Convolutional Autoencoder

1. Implement a simple convolutional autoencoder for MNIST digit reconstruction
2. Extract the bottleneck layer representations (latent space)
3. Compare these representations with traditional PCA by:
   - Visualizing the 2D projections of both
   - Measuring reconstruction quality at comparable compression ratios
   - Analyzing the feature importance in both approaches

```
X = mnist.data[:10].astype("float32") / 255.0
original_images = X.reshape(-1, 28, 28)
X_clean = X.reshape(10, -1)


noisy_images = np.array([random_noise(img, mode='s&p', amount=0.3) for img in
    original_images])
X_noisy = noisy_images.reshape(10, -1)

pca = PCA(n_components=5)
X_pca_recon = pca.inverse_transform(pca.fit_transform(X_noisy))

# Robust PCA denoising
_, X_rpca_recon, _, _ = robust_pca(X_noisy, n_components=5)

# PSNR & SSIM
pca_psnr = [psnr(orig, recon) for orig, recon in zip(X_clean.reshape(-1, 28,
    28), X_pca_recon.reshape(-1, 28, 28))]
rpca_psnr = [psnr(orig, recon) for orig, recon in zip(X_clean.reshape(-1, 28,
    28), X_rpca_recon.reshape(-1, 28, 28))]

pca_ssim = [ssim(orig, recon, data_range=1.0) for orig, recon in zip(X_clean.
    reshape(-1, 28, 28), X_pca_recon.reshape(-1, 28, 28))]
rpca_ssim = [ssim(orig, recon, data_range=1.0) for orig, recon in zip(X_clean.
    reshape(-1, 28, 28), X_rpca_recon.reshape(-1, 28, 28))]

# Table
import pandas as pd
df = pd.DataFrame({
    "Obraz": list(range(1, 11)),
    "PSNR_PCA": pca_psnr,
    "PSNR_RobustPCA": rpca_psnr,
    "SSIM_PCA": pca_ssim,
    "SSIM_RobustPCA": rpca_ssim
})
print(df.round(3))

# Wizualizacja
for i in range(3):
    fig, axs = plt.subplots(1, 4, figsize=(10, 3))
    axs[0].imshow(original_images[i], cmap="gray")
    axs[0].set_title("Original")

    axs[1].imshow(noisy_images[i], cmap="gray")
    axs[1].set_title("Noisy")

    axs[2].imshow(X_pca_recon[i].reshape(28, 28), cmap="gray")
    axs[2].set_title("PCA")

```

```
45      axs[3].imshow(X_rpca_recon[i].reshape(28, 28), cmap="gray")
46      axs[3].set_title("Robust PCA")
47
48      for ax in axs:
49          ax.axis("off")
50      plt.suptitle(f"Image #{i+1}")
51      plt.tight_layout()
52      plt.show()
```

```
   Obraz  PSNR_PCA  PSNR_RobustPCA  SSIM_PCA  SSIM_RobustPCA
0      1    10.291           9.099     0.338           0.451
1      2     8.687           8.218     0.478           0.484
2      3     9.850           9.273     0.285           0.352
3      4     8.948           8.096     0.185           0.228
4      5     9.618           8.635     0.277           0.336
5      6     8.737           8.531     0.401           0.403
6      7    10.132           8.641     0.201           0.231
7      8     8.433           8.358     0.392           0.410
8      9    10.985           8.492     0.182           0.196
9     10     8.256           7.809     0.288           0.300
C:\Users\jacek\AppData\Local\Temp\ipykernel_31040\2159679163.py:17: UserWarning: Inputs have mismatched dtype.  Setting data_range based on image_true.
  rpca_psnr = [psnr(orig, recon) for orig, recon in zip(X_clean.reshape(-1, 28, 28), X_rpca_recon.reshape(-1, 28, 28))]
```

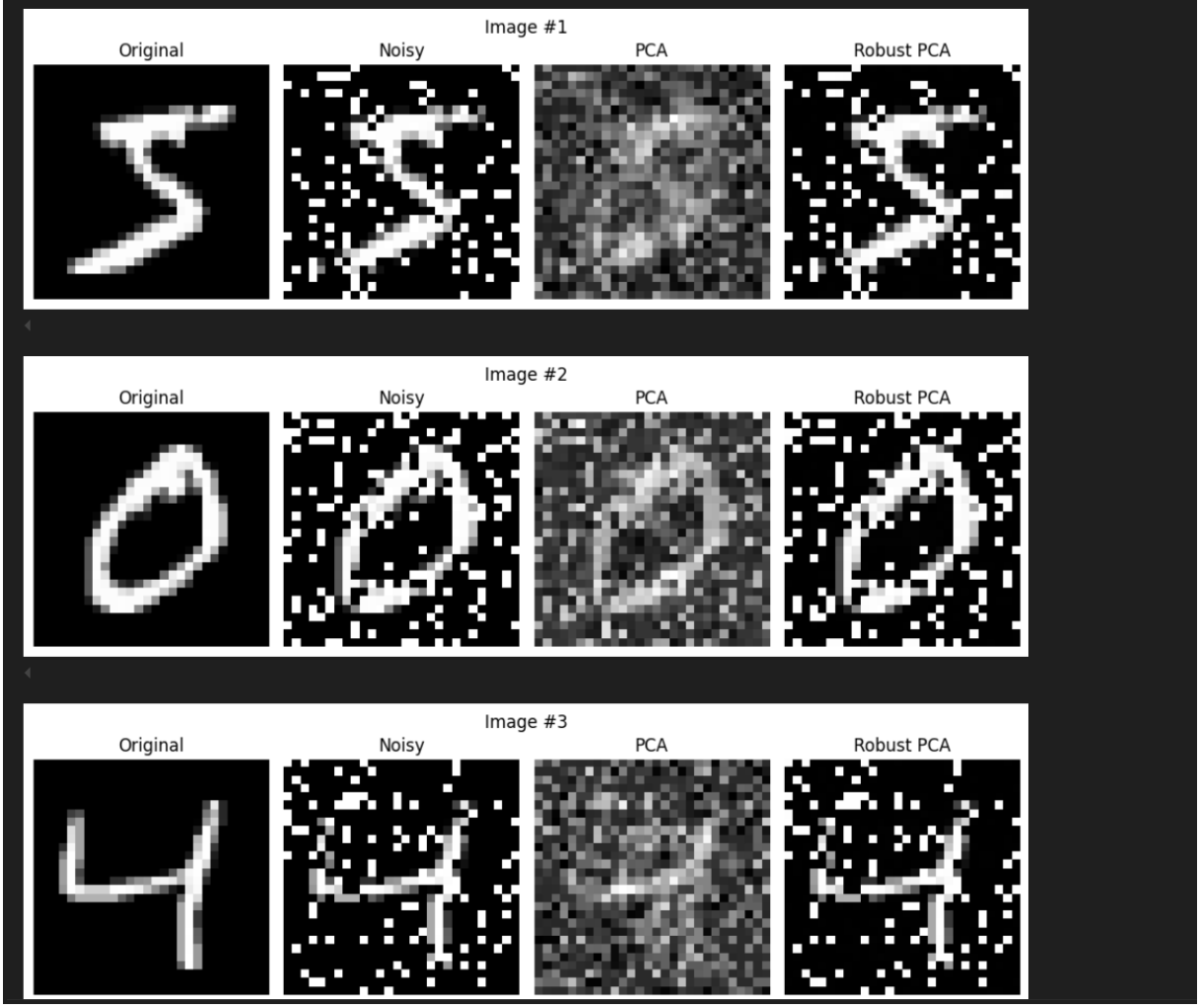

Fig. 1.13. Feature Heatmap

Analysis

PSNR is a metric that measures the pixel-wise difference between the original and reconstructed images. A higher PSNR indicates lower error and better quality, but it does not necessarily correspond to visual perception. On the other hand, SSIM is a perceptual metric that evaluates image similarity based on structure, luminance, and contrast. It better reflects human visual perception—higher SSIM values imply higher visual similarity.

We can see that PPCA achieves better PSNR than Robust PCA, but this doesn't reflect perceived visual quality. However, in SSIM, Robust PCA performs better than PCA. Since SSIM is more aligned with human visual perception, it is considered a more reliable metric. In my opinion, the image transformed by Robust PCA is clearer than the one produced by standard PCA.

A key strength of Robust PCA is that it preserves the important structure in data even under corruption, such as salt-and-pepper noise. It also achieves better visual quality in image reconstruction than standard PCA. However, the method comes with certain limitations. It typically requires more memory and longer convergence time due to iterative optimization. In real-world applications, Robust PCA proves useful in fields like medical imaging—where maintaining structural fidelity is essential—or in anomaly detection tasks.

Regarding computational complexity, Robust PCA tends to scale poorly with increasing dataset size, mainly due to its reliance on repeated SVD computations. This makes it more computationally intensive than traditional PCA. Reducing this complexity is challenging, but one potential direction is to work with a smaller subset of representative samples.

To improve the method further, automatic hyperparameter optimization could be introduced. Techniques such as Bayesian Optimization (BOGP), Tree-structured Parzen Estimator (TPE), SMAC, or genetic algorithms could be leveraged to fine-tune parameters like the regularization weights or convergence criteria. However, implementing such optimization requires a well-defined and effective loss function tailored to the reconstruction quality or task-specific requirements.