Katarzyna Skoczylas
Ada Wojterska

ROZKŁAD SVD

W naszym projekcie zajmiemy się analizą zastosowań rozkładu macierzy według wartości osobliwych (singular value decomposition - SVD).

Rozważmy macierz A (m x n) z elementami należącymi do liczb rzeczywistych. Aby znaleźć jej rozkład SVD potrzebujemy trzech macierzy. Są to:

- macierz lewostronna wektorów osobliwych U (m x m),
- macierz wartości osobliwych ∑ (m x n), czyli pierwiastków wartości własnych macierzy A^TA (Jest ona symetryczna i nieujemnie określona, a jej wszystkie wartości własne są nieujemne). Jednak w niektórych źródłach jako wartości osobliwe rozumiane są tylko pierwiastki z dodatnich wartości własnych,
- macierz prawostronna wektorów osobliwych V (n x n).

Rozkład SVD macierzy A można zapisać jako:

$$A = U\Sigma V^T.$$

Warto wymienić kilka znaczących własności tego rozkładu:

- U i V to macierze kwadratowe, niezależnie od tego czy A jest kwadratowa, czy nie.
- Macierze wektorów osobliwych U i V są unitarne, jednak my będziemy zajmować się tylko przypadkiem rzeczywistym, więc będą one ortogonalne. Oznacza to, że $U^TU = I \; i \; V^TV = I.$
- Macierz wartości osobliwych ∑ to macierz o tych samych wymiarach jak A. Wartości osobliwe są ułożone na „diagonali" w kolejności od największej (lewy górny róg) do najmniejszej (prawy dolny róg). Jeśli A nie jest macierzą kwadratową to musimy dodać blok zerowy po prawej (jeśli liczba kolumn macierzy A jest mniejsza niż liczba wierszy) lub poniżej diagonali (jeśli liczba kolumn macierzy A jest większa niż liczba wierszy).
- Pierwsze n kolumn macierzy U to znormalizowany i ortogonalny układ wektorów własnych AA^T. Jeśli A nie jest kwadratowa, to pozostałe kolumny to znormalizowane dopełnienie ortogonalne pierwszych r kolumn.
- Pierwsze n wierszy macierzy V^T to znormalizowany i ortogonalny układ wektorów własnych A^TA. Jeśli A nie jest kwadratowa, to pozostałe wiersze to znormalizowane dopełnienie ortogonalne pierwszych r wierszy.
- Wszystkie wartości osobliwe są rzeczywiste oraz nieujemne.
- Liczba niezerowych wartości osobliwych jest równa rzędowi macierzy, czyli pierwiastków z dodatnich wartości własnych macierzy A^TA.

Opisana jako ostatnia własność wynika z faktu, że przestrzeń kolumnowa i wierszowa macierzy to odpowiednio lewostronne i prawostronne wektory osobliwe, które są jedynie mnożone przez odpowiadające im niezerowe wartości osobliwe.

Pokażemy teraz jak przebiega mnożenie $A = U\Sigma V^T$ i wynikające z tego ciekawe wnioski.

Widzimy więc, że wyjściową macierz A można przedstawić w postaci sumy macierzy:

$A = \sum_{i=1}^{k} u_i \sigma_i v_i^T$, gdzie k to liczba niezerowych wartości osobliwych.

Rząd każdej z nich jest równy 1. Wiemy, że wartości osobliwe w macierzy ∑ są uporządkowane malejąco, a więc każda kolejna macierz A_i będzie zazwyczaj niosła mniej informacji o macierzy wyjściowej A. Zatem A_1, czyli macierz odpowiadająca największej z wartości osobliwej σ_i, jest najważniejszą „warstwą" macierzy A. Nie musimy jednak korzystać ze wszystkich warstw i zamiast tego utworzyć inną macierz A' zawierającą pierwsze $l \leq k$ warstw. Taką macierz nazywamy **aproksymacją macierzy A macierzami niskiego rzędu**. Otrzymujemy wtedy

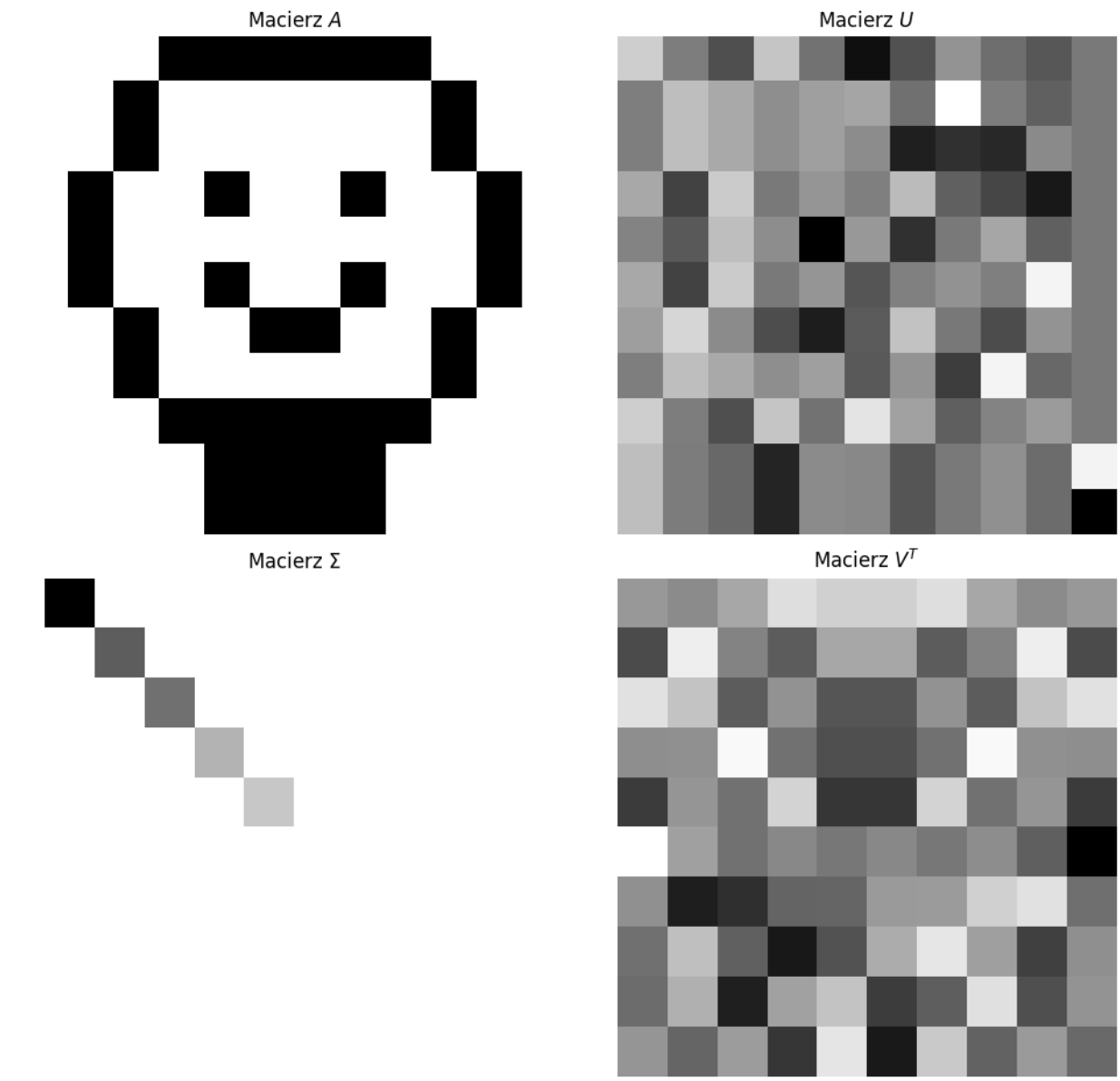
przybliżenie rzędu l. Operacja ta ma wiele praktycznych zastosowań, takich jak oczyszczanie szumów i kompresja danych, które postaramy się przedstawić w dalszej części naszej pracy.

CZĘŚĆ 1. WPROWADZENIE

Na przykładzie prostej macierzy pokażemy, jak działa SVD i jak zmienia się dokładność przybliżenia w zależności od ilości sumowanych „warstw".

Na początku stworzyłyśmy przykładową macierz A, przeprowadziłyśmy rozkład SVD i w obrazowy sposób przedstawiłyśmy macierze U, $\sum$, V^T.

```python
import matplotlib.pyplot as plt
import numpy as np

4 usages  kasia2004
def rysuj_macierz(ax, macierz, tytul, min, max):
    ax.imshow(macierz, cmap='gray_r', interpolation='nearest', vmin=min, vmax=max)
    ax.set_title(tytul)
    ax.axis('off')

kasia2004 *
def main():
    # Przykładowa macierz A
    A = np.array([
        [0, 0, 1, 1, 1, 1, 1, 1, 0, 0],
        [0, 1, 0, 0, 0, 0, 0, 0, 1, 0],
        [0, 1, 0, 0, 0, 0, 0, 0, 1, 0],
        [1, 0, 0, 1, 0, 0, 1, 0, 0, 1],
        [1, 0, 0, 0, 0, 0, 0, 0, 0, 1],
        [1, 0, 0, 1, 0, 0, 1, 0, 0, 1],
        [0, 1, 0, 0, 1, 1, 0, 0, 1, 0],
        [0, 1, 0, 0, 0, 0, 0, 0, 1, 0],
        [0, 0, 1, 1, 1, 1, 1, 1, 0, 0],
        [0, 0, 0, 1, 1, 1, 1, 0, 0, 0],
        [0, 0, 0, 1, 1, 1, 1, 0, 0, 0]
    ])

    # Przeprowadzenie rozkładu SVD
    U, s, Vt = np.linalg.svd(A)
    # Stworzenie macierzy diagonalnej S
    S = np.diag(s)
    print(U, s, Vt)
    # Tworzenie rysunków na jednej figurze
    fig, axes = plt.subplots( nrows: 2,  ncols: 2, figsize=(10, 10))

    rysuj_macierz(axes[0, 0], A,  tytul: "Macierz $A$", np.min(A), np.max(A))
    rysuj_macierz(axes[0, 1], U,  tytul: "Macierz $U$", np.min(U), np.max(U))
    rysuj_macierz(axes[1, 0], S,  tytul: "Macierz $\\Sigma$", np.min(S), np.max(S))
    rysuj_macierz(axes[1, 1], Vt,  tytul: "Macierz $V^T$", np.min(Vt), np.max(Vt))

    plt.tight_layout()
    #plt.savefig("wyjsciowe_macierze.png")
    plt.show()
```



Naszym następnym celem było zwizualizowanie wkładu kolejnych macierzy A_i w ostateczną postać macierzy A. Doszłyśmy do wniosku, że funkcja rysująca macierz musi przyjmować dwa parametry - minimum i maksimum, ponieważ inaczej skala odcieni czerni i bieli jest dostosowywana przy każdym wywołaniu indywidualnie dla każdej macierzy. Nasze „warstwy" zawierają inne wartości, w tym różnią się wartością największą i najmniejszą, więc aby ich wizualne porównywanie miało sens trzeba dostosować skalę kolorów w taki sam sposób dla wszystkich macierzy. W przeciwnym wypadku, jeśli wartości najmniejsze i największe dwóch macierzy to odpowiednio: -3 i 5 oraz 1 i 2, to wartość 0 w obu macierzach będzie reprezentowana przez inny odcień. Najbardziej znaczące są wartości jak najdalsze od 0, zarówno te dodatnie i jak i ujemne. Z tego powodu jako minimum przyjmujemy 0, a jako maksimum moduł z wartości najbardziej odległej od 0. Coraz ciemniejszy kolor oznacza większy „wpływ" na końcową postać macierzy.

```
    if i >= len(U):
        Ai = np.zeros_like(U)
        return Ai
    ui = U[:, i].reshape(-1, 1)
    vti = Vt[i, :].reshape(1, -1)
    s = S[i][i]
    Ai = ui * s @ vti
    return Ai


1 usage   kasia2004
def stworz_skale(lista):
    min = np.inf
    max = -np.inf
    for i in range(len(lista)):
        if np.min(lista[i]) < min:
            min = np.min(lista[i])
    for i in range(len(lista)):
        if np.max(lista[i]) > max:
            max = np.max(lista[i])
    if abs(min) > abs(max):
        return abs(min)
    else:
        return abs(max)
```

```
    A1 = stworz_Ai(U, S, Vt,  i: 0)
    A2 = stworz_Ai(U, S, Vt,  i: 1)
    A3 = stworz_Ai(U, S, Vt,  i: 2)
    A4 = stworz_Ai(U, S, Vt,  i: 3)
    A5 = stworz_Ai(U, S, Vt,  i: 4)
    A6 = stworz_Ai(U, S, Vt,  i: 5)
    A7 = stworz_Ai(U, S, Vt,  i: 6)
    A8 = stworz_Ai(U, S, Vt,  i: 7)
    A9 = stworz_Ai(U, S, Vt,  i: 8)
    A10 = stworz_Ai(U, S, Vt,  i: 9)

    lista_macierzy = [A1, A2, A3, A4, A5, A6, A7, A8, A9, A10]
    max = stworz_skale(lista_macierzy)

    fig, axes = plt.subplots( nrows: 3,  ncols: 2, figsize=(12, 10))

    for i in range(6):
        rysuj_macierz(axes[i // 2, i % 2], np.abs(lista_macierzy[i]),  tytul: f"Macierz $A_{i + 1}$",  min: 0, max)

    plt.tight_layout()
    plt.savefig("macierze_A1_A6.png")
    plt.show()
```

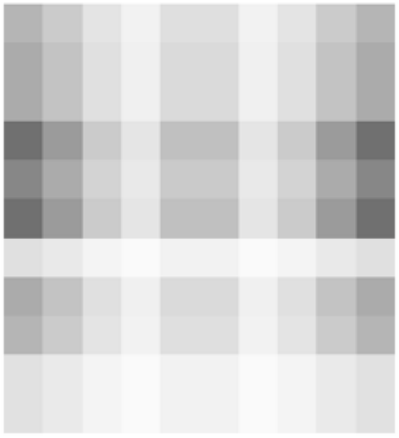
Poniżej zaprezentowane jest 6 macierzy – kolejnych „warstw” macierzy wyjściowej. Jak łatwo zauważyć, nasze przypuszczenie się potwierdziło – każda kolejna macierz, odpowiadająca coraz mniejszej wartości osobliwej, ma jaśniejszy kolor. Macierz A_6 jest cała biała, co oznacza, że nie ma ona żadnego wpływu na wartości macierzy A. Tak samo będzie dla wszystkich kolejnych „warstw”.

Warto wspomnieć, że w rzeczywistości często trudno jest odróżnić bardzo małe wartości osobliwe od zera. Podczas pisania programu stosuje się tolerancję, czyli określony próg, aby zadecydować, czy dana wartość powinna być uznawana za zero.

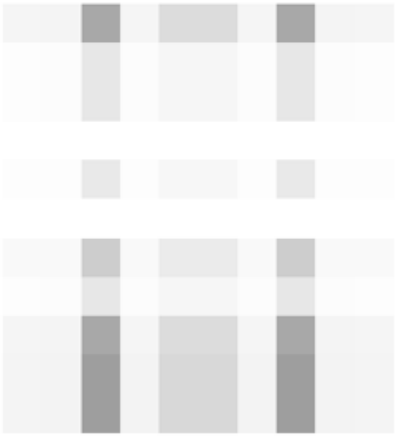
Macierz A_1

Macierz A_2

Macierz A_3



Macierz A_4

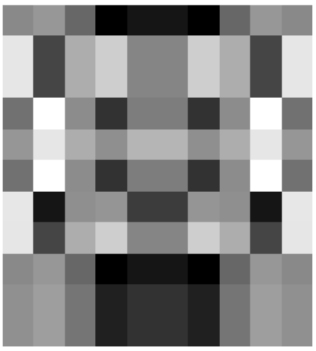


Macierz A_5

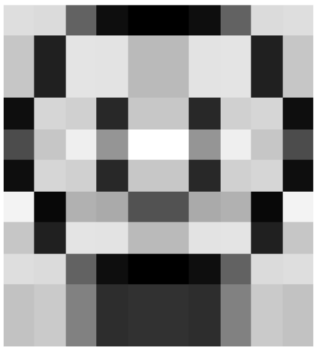
Macierz A_6

Kolejnym krokiem było pokazanie, że sumując poszczególne warstwy, rzeczywiście otrzymamy oryginalną macierz, bądź też odpowiednio dokładne jej przybliżenie. Patrząc na rysunki poniżej widzimy, że czym więcej „warstw” zsumujemy, tym rysunek bliższy jest wyjściowemu obrazowi.

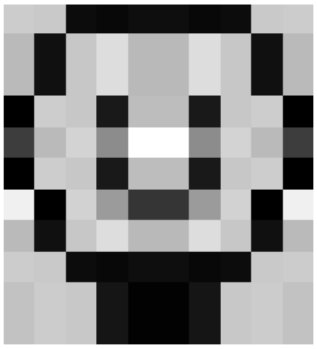
Macierz A1 + A2



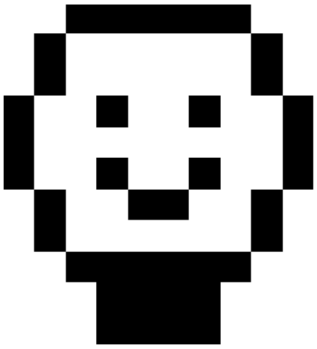
Macierz A1 + A2 + A3



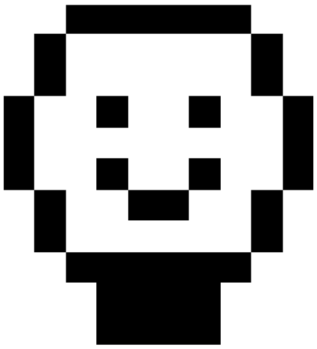
Macierz A1 + A2 + A3 + A4



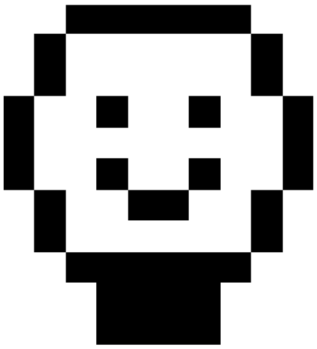
Macierz A1 + A2 + A3 + A4 + A5



Macierz A1 + A2 + A3 + A4 + A5 + A6

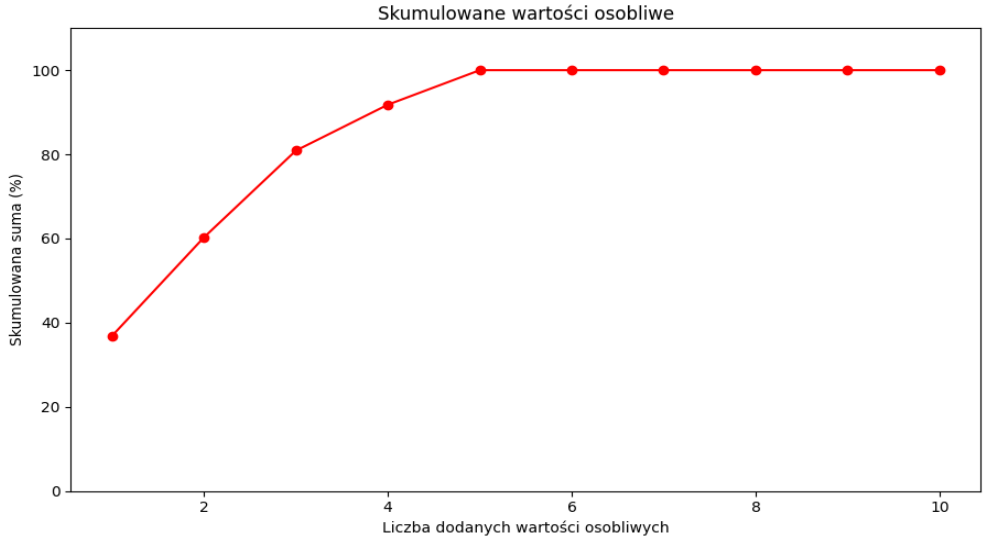


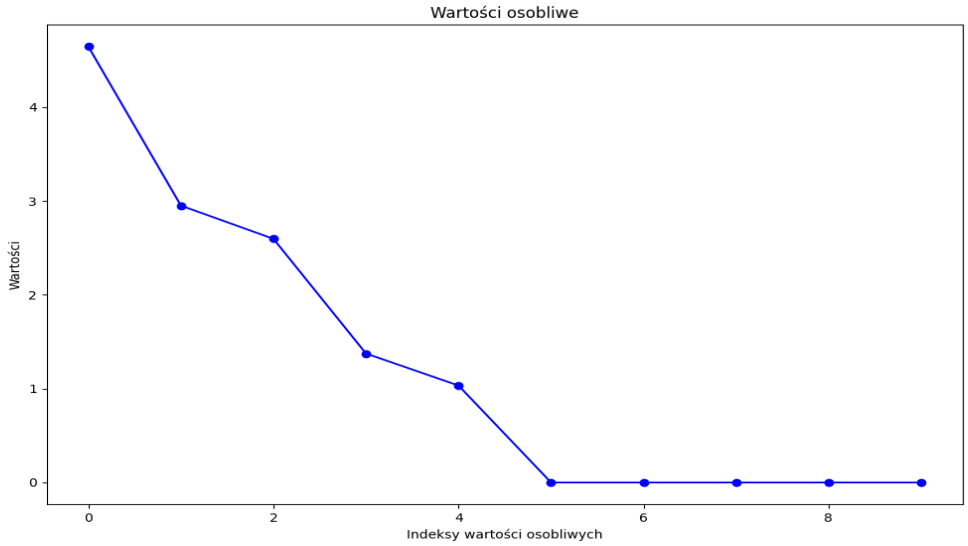
Macierz A1 + A2 + A3 + A4 + A5 + A6 + A7



```
rysuj_macierz(axes[0,0], A1 + A2, tytul: "Macierz A1 + A2", np.min(A1 + A2), np.max(A1 + A2))
rysuj_macierz(axes[0,1], A1 + A2 + A3, tytul: "Macierz A1 + A2 + A3", np.min(A1 + A2 + A3), np.max(A1 + A2 + A3))
rysuj_macierz(axes[1,0], A1 + A2 + A3 + A4, tytul: "Macierz A1 + A2 + A3 + A4", np.min(A1 + A2 + A3 + A4),
              np.max(A1 + A2 + A3 + A4))
rysuj_macierz(axes[1,1], A1 + A2 + A3 + A4 + A5, tytul: "Macierz A1 + A2 + A3 + A4 + A5", np.min(A1 + A2 + A3 + A4 + A5),
              np.max(A1 + A2 + A3 + A4 + A5))
rysuj_macierz(axes[2,0], A1 + A2 + A3 + A4 + A5 + A6, tytul: "Macierz A1 + A2 + A3 + A4 + A5 + A6", np.min(A1 + A2 + A3 + A4 + A5 + A6),
              np.max(A1 + A2 + A3 + A4 + A5 + A6))
rysuj_macierz(axes[2,1], A1 + A2 + A3 + A4 + A5 + A6 + A7, tytul: "Macierz A1 + A2 + A3 + A4 + A5 + A6 + A7",
              np.min(A1 + A2 + A3 + A4 + A5 + A6 + A7),
              np.max(A1 + A2 + A3 + A4 + A5 + A6 + A7))

plt.tight_layout()
plt.savefig("macierze_sumy.png")
plt.show()
```

Postanowiłyśmy także stworzyć wykresy podsumowujące analizę naszego obrazu. Ciekawym wydało nam się to, że już pierwsza wartość osobliwa, to prawie 40% sumy wszystkich z nich , a co za tym idzie pierwsza „warstwa” odpowiada za około 40% tworzonego obrazu, a zsumowanie połowy z nich pozwala nam osiągnąć niemal 100% dokładność przybliżanego obrazu.






```
plt.figure(1)
plt.plot( *args: s, marker='o', linestyle='-',
         color='b')
plt.title('Wartości osobliwe')
plt.xlabel('Indeksy wartości osobliwych')
plt.ylabel('Wartości')
plt.show()
cumulative_sum = np.cumsum(s) / np.sum(s)
cumulative_sum_scaled = cumulative_sum * 100
plt.figure(2)
plt.plot( *args: range(1, len(cumulative_sum_scaled) + 1), cumulative_sum_scaled, marker='o', linestyle='-', color='r')
plt.title('Skumulowane wartości osobliwe')
plt.xlabel('Liczba dodanych wartości osobliwych')
plt.ylabel('Skumulowana suma (%)')
plt.ylim( *args: 0, 110)
plt.show()
```

CZĘŚĆ 2. RZECZYWISTE OBRAZY

Kompresja obrazów to rodzaj kompresji danych stosowany do obrazów cyfrowych, mający na celu zmniejszenie kosztów ich przechowywania lub transmisji. W kontekście obrazów cyfrowych macierz obrazu (gdzie każdy piksel jest reprezentowany przez wartość) może być poddana SVD, co pozwala na oddzielenie istotnych informacji (które zajmują mniej miejsca) od tych mniej istotnych (które mogą być pominięte lub skompresowane). Redukując rząd macierzy możemy zrekonstruować pierwotny obraz. Mimo że będzie on mniej szczegółowy, różnica jest często niezauważalna dla ludzkiego oka.

Rozmiar każdej ze zredukowanych macierzy dla k największych wartości osobliwych to:

- $U_K: m \times k$
- $\Sigma_k: k \times k$
- $V_k: n \times k$
- Stosunek łącznej ilość pamięci zajmowanej przez aproksymację rzędu do rozkładu początkowej macierzy:
  stosunek_zajmowanej_pamięci = $(m \times k + k \times k + n \times k) / (m \times m + m \times n + n \times n)$.

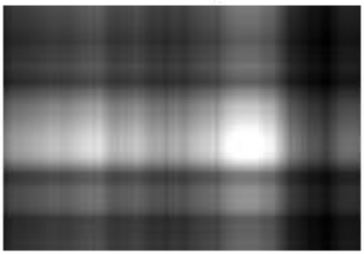
Aby sprawdzić te informacje w praktyce, postanowiłyśmy przeanalizować bardziej szczegółowe, rzeczywiste zdjęcia. Modelami zostali kolejno: Kluska, Stefan, Kopytko i Prążek

```python
def sumuj_do_Ai(kot, i):
    # Przeprowadzenie rozkładu SVD
    U, s, Vt = np.linalg.svd(kot)
    # Stworzenie macierzy diagonalnej S
    S = np.diag(s)
    K = np.zeros_like(kot, dtype=np.float64)  # Rzutowanie na float64
    for j in range(i):
        Ai = stworz_Ai(U, S, Vt, j)
        Ai= np.pad(Ai, pad_width: ((0, 0), (0, kot.shape[1] - Ai.shape[1])), mode='constant')
        K += Ai.astype(np.float64)  # Rzutowanie Ai na float64 przed dodaniem
        K = np.clip(K, a_min: 0, a_max: 255)
    return K.astype(np.uint8)  # Rzutowanie na uint8 przed zwróceniem
```

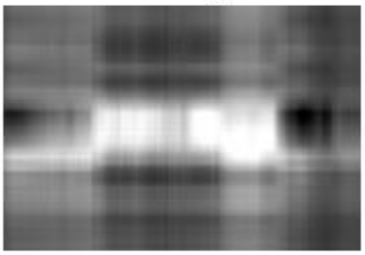
```python
    kot = np.array(image_gray)

    lista_indeksow = [1, 2, 4, 5, 10, 20, 50, 200, 700, 1000]
    fig, axes = plt.subplots( nrows: 5,  ncols: 2, figsize=(12, 8))

    for i in range(len(lista_indeksow)):
        row = i // 2
        col = i % 2
        # Rysowanie każdej macierzy na osobnym subplotcie
        kot_i = sumuj_do_Ai(kot, lista_indeksow[i])
        rysuj_macierz(axes[row, col], kot_i,  tytul: f"kot {lista_indeksow[i]}", np.min(kot_i), np.max(kot_i))

    plt.tight_layout()
    plt.savefig("kotek_ady.png")
    plt.show()
```

```python
def main():
    image_path = 'kot.jpeg'

    # Załadowanie obrazu
    image = Image.open(image_path)

    # Konwertowanie obrazu na odcienie szarości
    image_gray = image.convert('L')

    image_matrix = np.array(image_gray)

    print(image_matrix)

    fig, ax = plt.subplots( nrows: 1,  ncols: 1, figsize=(12, 8))

    rysuj_macierz(ax, image_matrix,  tytul: "kot", np.min(image_matrix), np.max(image_matrix))

    plt.show()
```

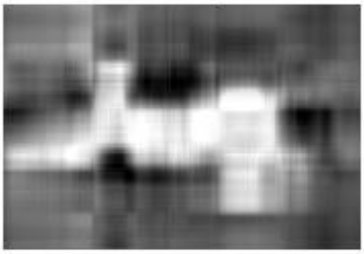
warstwy: 1



warstwy: 2



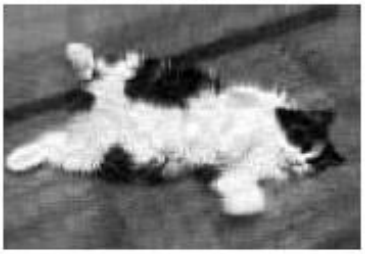
warstwy: 4



warstwy: 5

warstwy: 10

warstwy: 20



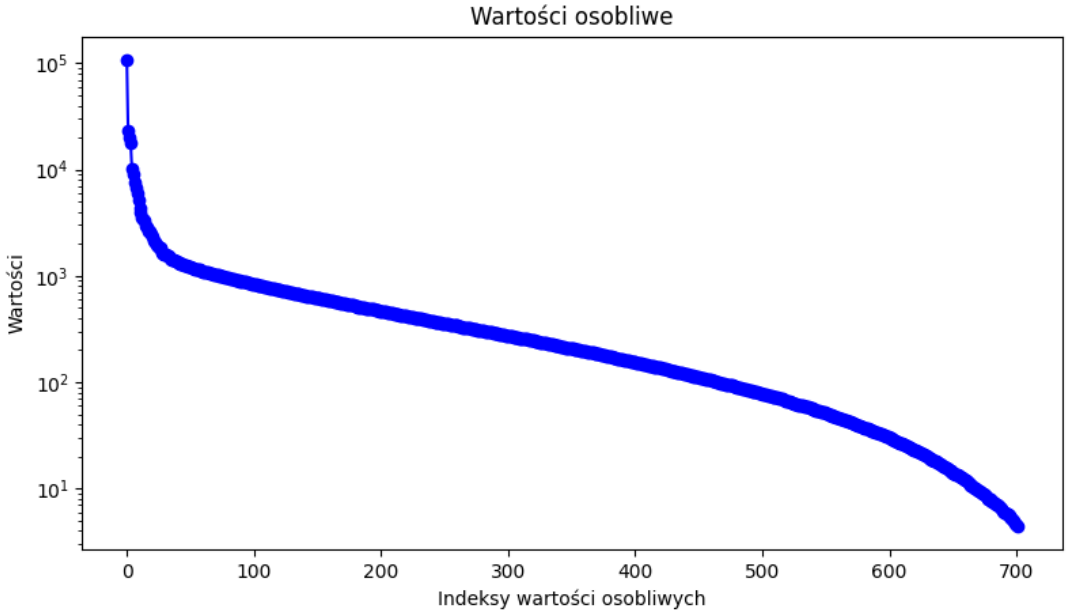
warstwy: 50

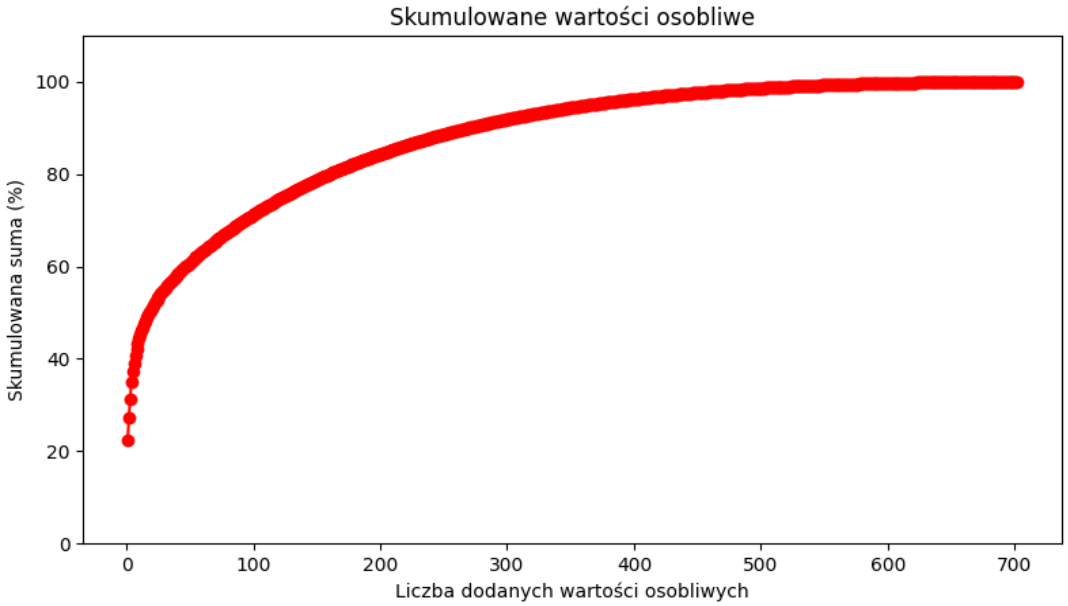
warstwy: 200

warstwy: 700

pierwotne zdjęcie:

Patrząc na wygenerowane rysunki, widzimy, że kot przedstawiony za pomocą sumy 50 „warstw” wygląda dostatecznie dobrze i w wielu przypadkach może on być satysfakcjonującym nas przybliżeniem. Zgodnie z przedstawionymi wykresami jest to około 7% wszystkich wartości osobliwych, które odpowiadają aż za blisko 60% całego obrazu. Warto zauważyć, że wartości osobliwe są przedstawione są przedstawione na wykresie w skali logarytmicznej, ponieważ maleją one tak szybko, że wykorzystanie liniowej skali bardzo zmniejszało czytelność danych. Ponadto musiałyśmy zastosować rzutowanie typów zmiennych między float64 i unit8, ponieważ w przeciwnym wypadku generowane były błędy, np. w postaci plam na rysunku.

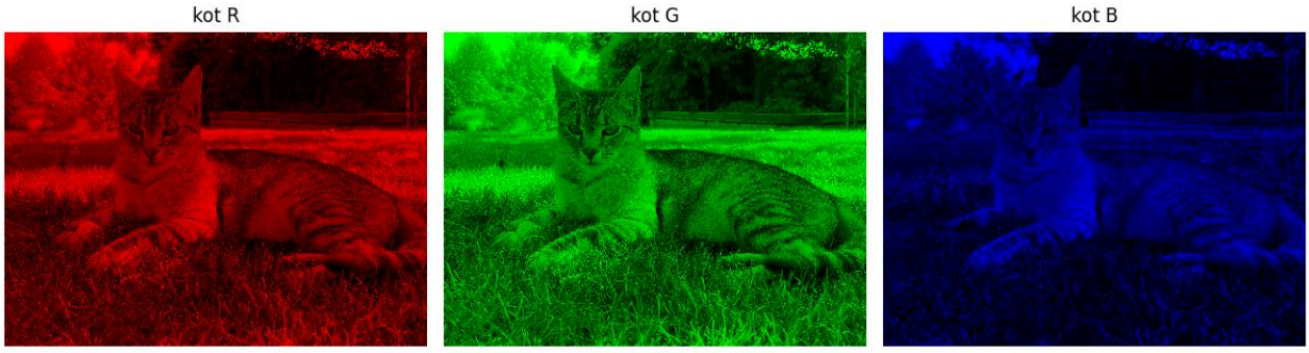




Korzystając ze przedstawionego wcześniej wzoru obliczyłyśmy stosunek ilość pamięci zajmowanej przez aproksymację 50. rzędu do rozkładu początkowej macierzy. Wynik to około 1/25, a więc 4%. Jak widać jest to znaczne zmiejszenie rozmiaru danych.

Ten sam proces przeprowadziłyśmy na kolorowym Prążku. Tu jednak postanowiłyśmy wykorzystać rozkład na trzy podstawowe kolory – RGB i to na nich, oddzielnie przeprowadzić przybliżanie, po czym nałożyć na siebie stworzone „warstwy”.

```python
def main():
    image_path = 'prazek-min.jpg'

    # Załadowanie obrazu
    image = Image.open(image_path)
    image = image.convert('RGB')
    kot = np.array(image)

    # Wyodrębnienie poszczególnych kanałów
    R = kot[:, :, 0]
    G = kot[:, :, 1]
    B = kot[:, :, 2]

    fig1, axes1 = plt.subplots( nrows: 1,  ncols: 3, figsize=(12, 8))
    z = np.zeros_like(R)
    rysuj_macierz(axes1[0], np.stack( arrays: (R, z, z), axis=-1),  tytul: f"kot R", np.min(R), np.max(R))
    rysuj_macierz(axes1[1], np.stack( arrays: (z, G, z), axis=-1),  tytul: f"kot G", np.min(G), np.max(G))
    rysuj_macierz(axes1[2], np.stack( arrays: (z, z, B), axis=-1),  tytul: f"kot B", np.min(B), np.max(B))
    plt.tight_layout()
    plt.savefig("kotek_RGB.png")
    plt.show()
```



warstwy: 1

warstwy: 2

warstwy: 4

warstwy: 5

warstwy: 10

warstwy: 20

warstwy: 50

warstwy: 200

warstwy: 700

pierwotne zdjęcie:

CZĘŚĆ 3. REDUKCJA SZUMÓW

Ostatnim etapem naszej pracy było zastosowanie rozkładu SVD do redukcji szumów na przykładzie Kopytka i Stefana. Poprzez odpowiednią manipulację wykorzystywanymi „warstwami", można eliminować komponenty odpowiadające za szum, zachowując istotne informacje o obrazie. W procesie odszumiania za pomocą SVD najczęściej zeruje się lub redukuje mniejsze wartości osobliwe, które odpowiadają za szum, jednocześnie zachowując większe wartości osobliwe, które reprezentują główne struktury i cechy obrazu. Po redukcji szumu obraz jest rekonstruowany z przefiltrowanych wartości osobliwych. Metoda ta jest szczególnie efektywna w odszumianiu obrazów o ich niskiej zawartości. Aby sprawdzić jej działanie wykorzystałyśmy dwa przykładowe szumy – szum typu sól i pieprz oraz szum biały.

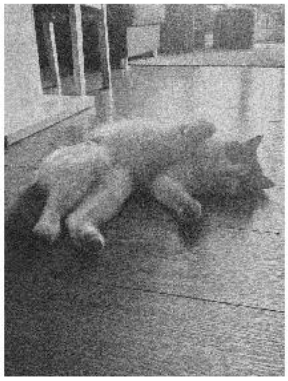
```python
def odszumianie(Mnoise, int):
    sigma = int
    U, S, Vt = np.linalg.svd(Mnoise, full_matrices=False)
    R = Mnoise.shape[0]
    cutoff = (4 / np.sqrt(3)) * np.sqrt(R) * sigma
    r = np.max(np.where(S > cutoff))

    # Odtworzenie obrazu za pomocą ograniczonej liczby wartości osobliwych
    Mclean = U[:, :r+1] @ np.diag(S[:r+1]) @ Vt[:r+1, :]
    return Mclean

new *
def dodaj_szum_solpieprz(M):
    Mnoise = np.copy(M)
    salt = np.random.rand(*M.shape) < 0.02
    pepper = np.random.rand(*M.shape) < 0.02
    Mnoise[salt] = 255
    Mnoise[pepper] = 0
    return Mnoise


1 usage  new *
def dodaj_szum_bialy(M, mean=0, std=35):
    noise = np.random.normal(mean, std, M.shape)
    Mnoise = M.astype(np.float64) + noise
    Mnoise = np.clip(Mnoise, a_min: 0, a_max: 255)
    Mnoise = Mnoise.astype(np.uint8)
    return Mnoise
```

```python
def main():
    image_path = 'kopytko2.jpg'
    image = Image.open(image_path)
    image_gray = image.convert('L')
    M = np.array(image_gray)

    fig, ax = plt.subplots( nrows: 2,  ncols: 2, figsize=(12, 8))

    rysuj_macierz(ax[0,0], M,  tytul: "Obraz wyjściowy", np.min(M), np.max(M))

    Mnoise = dodaj_szum_bialy(M)

    rysuj_macierz(ax[0,1], Mnoise,  tytul: "Obraz zaszumiony", np.min(Mnoise), np.max(Mnoise))

    Mclean1 = odszumianie(Mnoise,  int: 20)
    rysuj_macierz(ax[1,0], Mclean1,  tytul: "Obraz odszumiony 1", np.min(Mclean1), np.max(Mclean1))
    Mclean2 = odszumianie(Mnoise,  int: 20)
    rysuj_macierz(ax[1,1], Mclean2,  tytul: "Obraz odszumiony 2", np.min(Mclean2), np.max(Mclean2))

    plt.show()
```

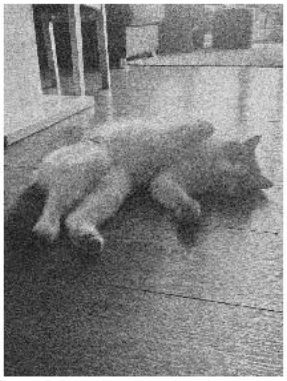
Obraz wyjściowy

Obraz zaszumiony

Obraz odszumiony 1



Obraz odszumiony 2



Obraz wyjściowy

Obraz zaszumiony

Obraz odszumiony 1

Obraz odszumiony 2