Universidade Federal do Rio de Janeiro

Álgebra Linear II – Professor: Thadeu Dias

Aluno: Miguel Badany Cerne

DRE: 123370433

Dia: 19/12/2023

Trabalho 2: Relatório

Todo o projeto foi implementado com o auxílio das bibliotecas matplotlib, numpy e Random.

O meu código foi seccionado em alguns arquivos:

- imageHandler.py: responsável pela conversão da imagem em preto e branco e a sua importação em formato de matriz, além da funcionalidade de exibir as imagens na tela, antes e depois do processamento;

- importer.py: simplesmente um arquivo que usei para importar todas as dependências em outros arquivos;

- main.py: arquivo que é responsável pela execução geral do programa. Nele, o usuário decide se fará a execução da primeira ou da segunda parte do trabalho;

- matricesgeneration.py: responsável pela geração de matrizes e vetores aleatórios utilizados na primeira parte do trabalho;

- part1.py: implementa toda a lógica necessária para a primeira parte do projeto;

- part2.py: implementa toda a lógica necessária para a segunda parte do projeto;

- plotters.py: arquivo auxiliar para plottar gráficos – para a parte 2 do trabalho;

- svd.py: arquivo auxiliar para aplicar o svd nas matrizes e para “transformar a matriz em uma imagem novamente”.

1 – Inicialização do Projeto

Para executar o projeto, todos arquivos .py devem estar em um mesmo diretório e devemos executar o arquivo main.py. Assim, o programa exige que o usuário indique que parte do projeto deseja executar (“1” ou “2”).

Em todos os casos,

1.1 – Parte 1

Caso o usuário escolha a primeira parte do projeto, o programa executará os quatro casos que o trabalho exige nos formatos de uma matriz aleatória e um outro vetor, também aleatório. Obedecendo as seguintes regras:

```
rows =    [3, 4, 3, 4]
columns = [3, 3, 4, 3]
ranks =   [3, 3, 3, 2]
```

Caso 1: 3x3 com ranque completo;

Caso 2: 4x3 com ranque completo;

Caso 3: 3x4 com ranque completo;

Caso 4: 4x3 com ranque incompleto.

Para cada um desses casos, o programa imprime na tela:

1. A Matriz aleatória gerada;
2. O vetor aleatório gerado;
3. O vetor b – solução do sistema;
4. O produto Ax para checar se resulta no vetor b;
5. O produto USVh;
6. A norma de x;
7. A compatibilidade entre Ax e b;
8. A compatibilidade entre USVh e A.

Note: Para todas as compatibilidades, foram considerados como aceitáveis, erros de 1e-16:

```
print("Compatibility Test: Ax vs. b")
print(np.allclose(Ax_product, vector, atol=1e-16))
print()

print("Compatibility Test: USVh vs. A")
print(np.allclose(matrix, USVh_product, atol=1e-16))
print()
```

1.2 – Parte 2

Caso o usuário escolha a segunda parte do projeto, o programa executará o SVD para a imagem em preto e branco para diferentes K's:

```
for i in range(40):
    k_value = (i*(i+1)//2+1)
    reconstructed_image = apply_svd_to_image(bw_matrix, k_value)
```

Por fim, o programa finaliza mostrando o gráfico Norma Frobenius de A-Matriz Obtida por K.

2 – Análise: Parte 1

O programa nos dá como saídas as matrizes: U, S - $\sum$ -, V^H, onde:

- U: a matriz ortogonal de tamanho mxm

- $\sum$: a matriz diagonal composta por valores reais não negativos, onde seus valores são os valores singulares da matriz A de tamanho mxn

- V^H: a matriz conjugada transposta da matriz V unitária de tamanho nxn

2.1 – Caso 1: Matriz Quadrada e de Ranque Completo

Neste caso, utilizamos uma matriz quadrada 3x3:

```
Generated Matrix:
[[ 230   99   -1]
 [ -91 -153 -178]
 [  -3 -147 -165]]
```

O vetor b:

```
Generated vector:
[[1]
 [6]
 [1]]
```

O vetor x – resultado:

```
Solution:
[[-0.04835842]
 [ 0.12130487]
 [-0.11325298]]
```

O Produto entre A e x, cujo valor esperado é o vetor b:

```
Product A @ x:
[[1.]
 [6.]
 [1.]]
```

O Produto $U\sum V^H$, cujo valor esperado é a matriz A:

```
Product U @ S @ Vh:
[[ 230.   99.   -1.]
 [ -91. -153. -178.]
 [  -3. -147. -165.]]
```

Por fim, a norma de x e as compatibilidades:

```
Vector X norm:
0.17285729843514158

Compatibility Test: Ax vs. b
True

Compatibility Test: USVh vs. A
True
```

Nesse caso, como a matriz é quadrada e tem ranque completo, temos que o vetor x sempre pertencerá ao espaço gerado pela própria matriz e, portanto, o sistema tem soluções exatas, como confirmado pela exatidão do produto Ax e de $U\sum V^H$.

2.2 – Caso 2: Matriz Alta e Estreita e de Ranque Completo

Neste caso, utilizamos uma matriz 4x3:

```
Generated Matrix:
[[ -76    5 -124]
 [ -31 -137 -119]
 [-157 -103 -328]
 [ 186  -66  129]]
```

O vetor b:

```
Generated vector:
[[5]
 [6]
 [5]
 [5]]
```

O vetor x – resultado:

```
Solution:
[[ 0.05391182]
 [-0.00734816]
 [-0.04425553]]
```

O Produto entre A e x, cujo valor esperado é o vetor b:

```
Product A @ x:
[[1.35364725]
 [4.60184063]
 [6.80852062]
 [4.80361283]]
```

O Produto $U\sum V^H$, cujo valor esperado é a matriz A:

```
Product U @ S @ Vh:
[[ -76.    5. -124.]
 [ -31. -137. -119.]
 [-157. -103. -328.]
 [ 186.  -66.  129.]]
```

Por fim, a norma de x e as compatibilidades:

```
Vector X norm:
0.07013580807143154

Compatibility Test: Ax vs. b
False

Compatibility Test: USVh vs. A
True
```

Neste caso, diferentemente do caso da matriz quadrada de ranque completo, temos que, como $\dim(R(A))$ é menor que a dimensão do vetor b, é impossível que o vetor x pertença ao espaço formado pela própria matriz A. Quando isso ocorre, no entanto, a solução de SVD garante que o vetor x é a aproximação por mínimos quadrados.

Notamos, que – mesmo assim – o resultado da multiplicação $U\sum V^H$ ainda nos retorna o valor original de A. Isso ocorre, pois a presença de zeros na matriz $\sum$ não interfere na obtenção da matriz original, pois esses zeros representam dimensões nulas adicionadas para torná-la uma matriz quadrada, o que pouco interfere na reconstrução da matriz original.

2.3 – Caso 3: Matriz Baixa e Larga e de Ranque Completo

Neste caso, utilizamos uma matriz 3x4:

```
Generated Matrix:
[[ -81  -76    5 -124]
 [ 351  -31 -137 -119]
 [ 153 -157 -103 -328]]
```

O vetor b:

```
Generated vector:
[[3]
 [8]
 [3]]
```

O vetor x – resultado:

```
Solution:
[[-0.09630842]
 [-0.7875052 ]
 [-0.56020428]
 [ 0.4987932 ]]
```

O Produto entre A e x, cujo valor esperado é o vetor b:

```
Product A @ x:
[[3.]
 [8.]
 [3.]]
```

O Produto $U\sum V^H$, cujo valor esperado é a matriz A:

```
Product U @ S @ Vh:
[[ -81.  -76.    5. -124.]
 [ 351.  -31. -137. -119.]
 [ 153. -157. -103. -328.]]
```

Por fim, a norma de x e as compatibilidades:

```
Vector X norm:
1.0918164886841768

Compatibility Test: Ax vs. b
True

Compatibility Test: USVh vs. A
True
```

Neste caso, semelhantemente ao caso 1, temos que o sistema é “bem comportado”, ou seja, dimensão de A comporta o vetor b. Isto é, o vetor x ocupa o espaço gerado pela matriz A e, portanto, tanto o produto Ax quanto o produto $U\sum V^H$ dão valores exatos.

2.4 – Caso 4: Matriz Alta e Estreita e de Ranque Incompleto

Neste caso, utilizamos uma matriz 4x3:

```
Generated Matrix:
[[ -34  -19  -76]
 [-150  -69 -255]
 [-178  -91 -352]
 [  60    6  -15]]
```

O vetor b:

```
Generated vector:
[[5]
 [6]
 [3]
 [3]]
```

O vetor x – resultado:

```
Solution:
[[ 6.78350903e+13]
 [-4.64134828e+14]
 [ 8.56864298e+13]]
```

O Produto entre A e x, cujo valor esperado é o vetor b:

```
Product A @ x:
[[ -7.1875  ]
 [-28.484375]
 [-29.5     ]
 [ 13.265625]]
```

O Produto $U\sum V^H$, cujo valor esperado é a matriz A:

```
Product U @ S @ Vh:
[[ -34.  -19.  -76.]
 [-150.  -69. -255.]
 [-178.  -91. -352.]
 [  60.    6.  -15.]]
```

Por fim, a norma de x e as compatibilidades:

```
Vector X norm:
476827958894953.9

Compatibility Test: Ax vs. b
False

Compatibility Test: USVh vs. A
True
```

Por fim, temos o caso em que a matriz tem ranque incompleto, isto é, a dimensão do espaço gerado pela matriz A é menor que as dimensões da matriz A em si. Ou seja, o vetor x nos retorna, novamente, um valor inexato, que representa a solução de mínimos quadrados do respectivo sistema.

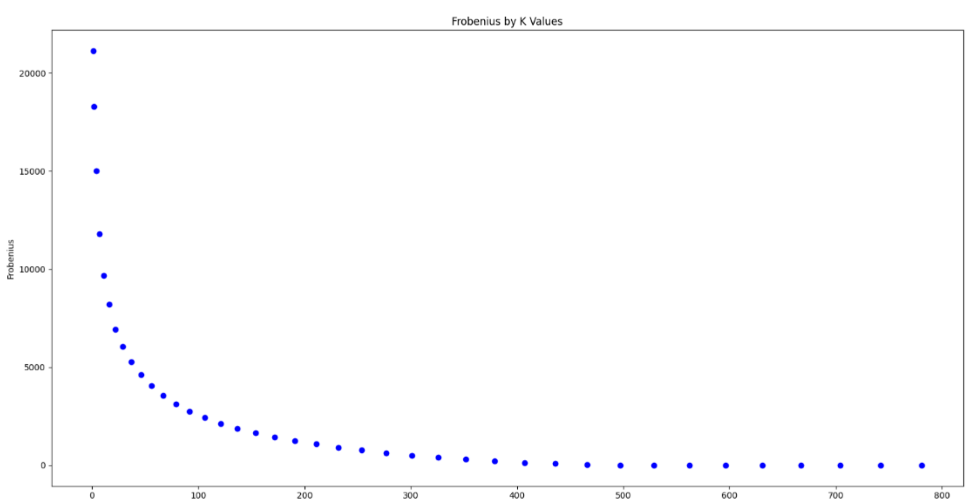
3 – Análise: Parte 2

Ao importarmos a imagem para o programa e passarmos de uma imagem colorida para uma imagem preta e branca obtemos o seguinte resultado:

Numericamente, essa importação resulta em uma matriz 512x512, onde cada elemento compõe um pixel, cujo valor representa a "claridade" do tom de cinza de cada pixel. Neste contexto, aplicaremos a fatoração SVD, novamente com auxílio de np.linalg.svd.

Então, utilizamos uma função auxiliar para somar os K primeiros valores singulares – onde K é um valor arbitrário configurado no programa – para reconstituir a imagem, ou seja, comprimindo-a.

Assim, temos, para cada K, uma matriz A_K, que representa a imagem composta pelos primeiros K valores singulares, que se aproximam cada vez mais da imagem original. Sobre a diferença $A-A_K$ aplicamos a Norma Frobenius, para cada K e plottamos estes resultados em um gráfico:



Ao observarmos o gráfico, intuímos que – supostamente – a imagem se aproxima seguindo uma escala assintótica, onde as melhorias começam rápidas mas perdem uma certa “velocidade” com o aumento de K. Ou seja, a partir uma certa aproximação A_K em específico, a diferença entre a imagem original e a imagem comprimida se torna insignificante e pode-se guardar a mesma informação utilizando um menor espaço de armazenamento.

Para confirmarmos esta intuição, dispomos de algumas amostras de imagens:

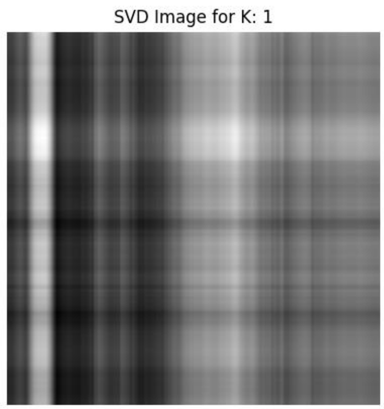


Imagem com K = 1, não dá para compreender nada na imagem.

Imagem com K = 10, algumas formas podem ser identificadas, mas o resultado ainda não é satisfatório.

Imagem com K=100, a imagem está compreensível mas ainda há um granulado claro na qualidade da imagem.

Imagem K=220, a partir de um coeficiente – aproximado – de 220, não há distinções significativas entre a imagem comprimida e a imagem original. A partir daí, já estamos no trecho do gráfico em que, com o crescimento de K, não temos uma melhoria significativa.

Ao transmitirmos a imagem original, necessitamos de 512x512 = 262.144 coeficientes, enquanto – como abordado no texto base para o trabalho – necessitamos de K(M+N+1) coeficientes para transmitir a imagem comprimida pelo SVD.

Assim, se considerarmos aceitável o granulado de K=100, precisamos de 100(512+512+1) = 102.500 coeficientes, poupando mais de 50% do espaço com relação à imagem original.

Considerando o resultado ideal como o utilizado, ou seja, K=200, teremos a necessidade de 200(512+512+1) = 205.000 coeficientes, ainda sim, poupando mais de 60.000 coeficientes (aproximadamente 20% do espaço ocupado pela imagem original).