Universidade de São Paulo
Escola Politécnica

Classificação de dígitos - Machine Learning
Cálculo númerico

Professor Dr. Pedro Peixoto
Ariel Guerchenzon - N°USP: 10335552
Matheus José Oliveira dos Santos - N°USP: 10335826

São Paulo
2019

1 INTRODUÇÃO

1.1 comentários iniciais

“Com o aumento da quantidade de dados circulando em todo o mundo, há uma crescente necessidade de se analisar dados automaticamente, sem a intervenção humana. Um dos principais desafios é o de encontrar padrões em quantidades enormes de dados, para obtermos formas de classificar e resumir os dados para que possam ser processadas por um humano.” (MAP3121 - Cálculo Númerico, enunciado do Exercício Programa 1).

Com a ascensão da era dos dados surgem novas tecnologias para tratá-los, com eles, conseguimos ensinar o computador a aprender tarefas as quais não seria possível programar diretamente. Esse tipo de programação é conhecido como Aprendizagem de Máquina, ou Machine Learning. Segundo Tom M. Mitchell: "Diz-se que um programa de computador aprende pela experiência *E,* com respeito a algum tipo de tarefa *T* e performance *P*, se sua performance *P* nas tarefas em *T*, na forma medida por *P*, melhoram com a experiência *E*."

Vale ressaltar que algorítmo de Machine Learning podem ser divididos basicamente em dois tipos: supervisionados e não supervisionados. Supervisionados são os algoritmos que recebem dados etiquetados, enquanto os não supervisionados encontram relações entre os dados sem essas supostas etiquetas. O algoritmo o qual este relatório descreve, se trata de um do tipo supervisionado, pois é conhecido os valores dos números utilizados durante o treinamento.

Este relatório tem como objetivo apresentar os resultados do Exercício Programa 1 de Cálculo numérico oferecido a Escola Politécnica para obtenção da nota parcial, além claro, de seu respectivo aprendizado. O problema consiste em classificar dígitos utilizando métodos numéricos aprendidos na disciplina, de tal forma que o problema é dividido em 3 tarefas.

A primeira tarefa baseia-se na solução de sistemas lineares utilizam fatoração QR, além da adaptação do algoritmo para resolução de vários sistemas simultâneos.

A segunda tarefa baseia-se em realizar uma fatoração não-negativa decompondo A em WH, através da utilização da função de resolver sistemas simultâneos desenvolvida na Tarefa 1.

E por fim, a tarefa principal realiza a classificação dos dígitos em si utilizando matrizes W_d treinadas com a base de dígitos manuscritos MNIST.

1.2 Estrutura de arquivos

A pasta .zip entregue contém os seguintes arquivos:

- TarefaPrincipal.py
- Tarefa2.py
- Tarefa1.py
- LEIAME.txt
- dados_mnist
    - train_dig0.txt
    - train_dig1.txt
    - train_dig2.txt
    - train_dig3.txt
    - train_dig4.txt
    - train_dig5.txt
    - train_dig6.txt
    - train_dig7.txt
    - train_dig8.txt
    - train_dig9.txt
    - test_index.txt
    - test_images.txt

2 PRIMEIRA TAREFA

2.1 Objetivo

Nesta tarefa, o objetivo é resolver sistemas lineares e resolver sistemas lineares simultâneos, através do método de decomposição **QR**. Para isso, dada uma matriz **W**, fatora-se **W** utilizando sucessivas rotações de Givens tal que **W=QR** onde **R** é uma matriz triangular superior. Com isso podemos resolver o sistema **Wx=b** conforme as equações abaixo:

$$\mathbf{Wx = b}$$

Mas, como **W** pode ser fatorado como **W** = **QR**, segue que

$$\mathbf{QRx = b}$$
$$\mathbf{Rx = Q^{-1}b}$$
$$\mathbf{Rx = b'}$$

Com $\mathbf{b' = Q^{-1}b}$

O que torna o sistema extremamente simples de ser resolvido.

Além disso, também podemos resolver sistemas lineares simultâneos, de tal forma que cada coluna da matriz **A** fosse equivalente ao vetor **b**. Segue abaixo o equacionamento:

$$\mathbf{WX = A}$$
$$\mathbf{QRX = A}$$
$$\mathbf{RX = Q^{-1}A}$$
$$\mathbf{RX = A'}$$

Onde **A** = $[b_1,b_2,b_3...b_n]$ tal que cada b_i seja um vetor a ser determinado e **X** = $[x_1,x_2,x_3...x_n]$.

No algoritmo sugerido no enunciado, aplicamos a rotação de given tanto a **W** quanto a **b** ou **A**. Dessa forma, não há necessidade de obter a matriz **Q**, pois ao aplicar a rotação de givens a **b** ou **A**, obtemos direto **b'** ou **A'**.

2.2 Código

O código foi modulado em funções para que possam ser usadas de forma independente. A única biblioteca utilizada na tarefa 1 foi o numpy. Segue abaixo o código com suas respectivas explicações.

2.2.1 getSenCos

A função *getSenCos* recebe dois parâmetros: w_{ik} e w_{jk} identificados por a e b, sendo esses, elementos da matriz **W** nas posições respectivas. Nessa função, foi escrito os dois métodos os quais o enunciado cita para a obtenção do $\sin\theta$ e $\cos\theta$ utilizados para aplicar a Rotação de Givens. Observa-se que o método utilizado abaixo, apresenta maior estabilidade numérico, sendo menos suscetível a erros de computação. A função retorna sin e cos como s e c.

```
# Recebe os itens w_ik e w_jk de [W] e retorna o sen(theta) e cos(theta) da Rotação de Givens.
def getSenCos(a,b):

    #Método numericamente mais estável para obtenção de sin e cos
    tau, c, s = 0, 0, 0
    if (abs(a) > abs(b)):
        tau = -(b/a)
        c = 1/(np.sqrt(1 + tau**2))
        s = c*tau
    else:
        tau = -(a/b)
        s = 1/(np.sqrt(1 + tau**2))
        c = s*tau

    #Método númericamente mais instável
    # r = (a**2 + b**2)**(1/2)
    # c = a/r
    # s = -b/r

    return s, c
```

2.2.2 Rot_givens

A função *Rot_givens* recebe como parâmetros a matriz qualquer (ou vetor) **W** a qual será aplicada um único passo da Rotação de Givens nas linhas i e j de **W**. E como dito anteriormente, precisa dos parâmetros s e c da função *getSenCos*. Observa-se que apenas as linhas i e j são alteradas.

Há um try-except no código, o qual possui como função identificar se **W** é uma matriz ou um vetor e aplicar os passos necessários para alterá-la. Verifica-se que a forma apresentada dentro do "try" substitui o for do algoritmo sugerido, mantendo o código mais compacto.

```
#realoza a rotação de givens numa W qualquer
def Rot_givens(W,i,j,s,c):
    #O try-except aqui foi utilizado para verificar se W é um vetor ou uma matriz
    #Dessa forma funciona tanto na resolver sistemas quanto na resolver sist. simultâneos.
    try:
        #Sendo uma matriz
        m = W.shape[1]
        W[i,:], W[j,:] = c*W[i,:] - s*W[j,:], s*W[i,:] + c*W[j,:]
    except:
        #Sendo um vetor
        aux = c*W[i] - s*W[j]
        W[j] = s*W[i] + c*W[j]
        W[i] = aux

    return W
```

2.2.3 getQRfactors

A função *getQRfactors* recebe como parâmetros a matriz **W** e uma matriz **b** que pode ser tanto b, como dito em 2.1, ou uma matriz **A**. A função aplica a Rotação de Given simultâneamente as matrizes **W** e **b**, não sendo necessário a matriz **Q**. Originalmente, tínhamos escritos essa função para obter tanto **Q** quanto **R**, devido a isso, manteve-se o nome da função *"getQRfactors"*. Vale ressaltar que **W** é uma matriz triangular superior.

```
#Realiza a rotação de givens
def getQRfactors(W,b):
    #b pode ser tanto um vetor quanto uma matriz
    b = b.transpose()
    for k in range(0,W.shape[1]):
        for j in range(W.shape[0]-1,k,-1):
            i = j-1
            if W[j][k] != 0:
                s, c = getSenCos(W[i][k],W[j][k])
                W = Rot_givens(W,i,j,s,c)
                b = Rot_givens(b,i,j,s,c)

    return W,b
```

2.2.4 Sol_linSis

A função *Sol_linSis* recebe como parâmetros a matriz **W** um vetor **b**. Aplica as Rotações de Givens correspondentes para transformar **W** numa triangular superior **R** e **b** em **b'**. Dessa forma, a resolução da equação **Rx=b'** é trivial, dado pelo algoritmo sugerido no enunciado do problema.

```
#Recebendo as matrizes A (triangular superior) e b (vetor), resolve o sistema linear
def Sol_linSis(W,b):
    n = W.shape[0] #Número de linhas de A
    m = W.shape[1] #Número de colunas de A

    R, b = getQRfactors(W,b)
    #Q, R = np.linalg.qr(W)
    X = np.zeros(m)
    #b = np.dot(Q.transpose(),b)

    for k in range(m-1,-1,-1):
        somatorio = 0
        for j in range(k+1,m):
            #print(R[k])
            somatorio += R[k][j]*X[j]
        if R[k][k] != 0:
            X[k] = (b[k] - somatorio)/R[k][k]
    return X
```

2.2.5 Sol_linSis_simult

Por fim, a função *Sol_linSis_simult* recebe como parâmetros as matrizes **W** a **A** já mencionadas, além do parâmetro p, que se trata do número de linhas da matriz resposta **X.** É obtido a matriz triangular superior **R** através de sucessivas aplicações da Rotações de Givens, depois disso, o sistema é resolvido de forma trivial como o algoritmo sugerido no enunciado.

```python
#Resolve sistemas simultâneos
def Sol_linSis_simult(W,A,p):
    m = A.shape[1]
    R, A = getQRfactors(W.copy(),A.transpose())
    X = np.zeros((p,m))
    for k in range(p-1,-1,-1):
        for j in range(m):
            somatorio = 0
            for t in range(k+1,p):
                somatorio += R[k][t]*X[t][j]
            if R[k][k] != 0:
                X[k][j] = (A[k][j] - somatorio)/R[k][k]
    return X
```

2.3 Resultados dos testes

2.3.1 Teste para a função Sol_linSin

Como teste inicial dos algoritmos, considere os casos (descreva os resultados no relatório):
a) n = m = 64, W i,i = 2, i = 1, n, W i,j = 1, se |i − j| = 1 e W i,j = 0, se |i − j| > 1. Use b(i) = 1, i = 1, n.

Cria-se as matrizes usadas no teste.

```python
W = np.eye(64)
W = W*2
for i in range(64):
    for j in range(64):
        if abs(i-j) == 1:
            W[i][j] = 1
        elif abs(i-j)>1:
            W[i][j] = 0
        else:
            pass
b = np.ones(64)
```

O resultado obtido foi o vetor:

x = [0.49230769 0.01538462 0.47692308 0.03076923 0.46153846 0.04615385
 0.44615385 0.06153846 0.43076923 0.07692308 0.41538462 0.09230769
 0.4 0.10769231 0.38461538 0.12307692 0.36923077 0.13846154
 0.35384615 0.15384615 0.33846154 0.16923077 0.32307692 0.18461538
 0.30769231 0.2 0.29230769 0.21538462 0.27692308 0.23076923

0.26153846 0.24615385 0.24615385 0.26153846 0.23076923 0.27692308
0.21538462 0.29230769 0.2 0.30769231 0.18461538 0.32307692
0.16923077 0.33846154 0.15384615 0.35384615 0.13846154 0.36923077
0.12307692 0.38461538 0.10769231 0.4 0.09230769 0.41538462
0.07692308 0.43076923 0.06153846 0.44615385 0.04615385 0.46153846
0.03076923 0.47692308 0.01538462 0.49230769]

Ao fazer a multiplicação *np.dot(W,x)* para obter novamente o vetor de 1's. Obtém-se o seguinte resultado:

b = [1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]

Resultado perfeito e dentro do esperado, de acordo com o **b** original.

b) n = 20, m = 17, W i,j = 1/(i + j − 1), se |i − j| ≤ 4 e W i,j = 0, se |i − j| > 4. Use b(i) = i, i = 1, n.

Cria-se as matrizes usadas no teste.

```
n = 20 #número de linhas
m = 17 #número de colunas
W = np.zeros((n,m))
for i in range(n):
    for j in range(m):
        if abs(i-j)>4:
            W[i][j] = 0
        else:
            W[i][j] = 1/(i+j-1+2) #+2 faz a correção pois i e j começam do zero.
b = np.ones(n)
```

O resultado obtido foi:

x = [2.88155063 -1.83376253 -1.51398904 -1.52190762 -0.45379461 5.85669889
3.42192891 3.65656219 1.20368454 6.12534249 -2.47971396 -1.47793147
-1.20390258 0.12841469 6.501589 11.49105599 14.58052582]

Ao fazer a multiplicação *np.dot(W,x)* para obter novamente o vetor de 1's. Obtém-se o seguinte resultado:

b = [0.98877052 0.90768119 0.99310105 1.1525159 1.10779778 0.92750055
0.96801475 0.98253494 0.98945734 0.96516387 0.80186148 0.99797545
1.27710297 1.17679316 0.88153264 0.94973366 0.9734249 0.98436988

0.95157654 0.73333049]

Resultado bastante aproximado do **b** original. Levantamos a hipótese da diferença entre o **b** obtido e o esperado ser devido a matriz **W** não ser quadrada.

2.3.2 Teste para a função Sol_linSin_simult:

c) n = m = 64, W i,i = 2, i = 1, n, W i,j = 1, se |i − j| = 1 e W i,j = 0, se |i − j| > 1. Defina p = 3, resolvendo 3 sistemas simultâneos, com A(i, 1) = 1, A(i, 2) = i, A(i, 3) = 2i − 1, i = 1, n.

Cria-se as matrizes usadas no teste.

```
W = np.eye(64)
W = W*2
for i in range(64):
    for j in range(64):
        if abs(i-j) == 1:
            W[i][j] = 1
        elif abs(i-j)>1:
            W[i][j] = 0
        else:
            pass

b = np.ones((3,64))
for i in range(64):
    b[1][i] = i+1
    b[2][i] = 2*(i+1) - 1
b = b.transpose()
```

O resultado obtido foi:

X = [[4.92307692e-01 -2.19456019e-14 -4.92307692e-01]
 [1.53846154e-02 1.00000000e+00 1.98461538e+00]
 [4.76923077e-01 -6.58314386e-14 -4.76923077e-01]
 [3.07692308e-02 2.00000000e+00 3.96923077e+00]
 [4.61538462e-01 -1.09788788e-13 -4.61538462e-01]
 [4.61538462e-02 3.00000000e+00 5.95384615e+00]
 [4.46153846e-01 -1.53778435e-13 -4.46153846e-01]
 [6.15384615e-02 4.00000000e+00 7.93846154e+00]
 [4.30769231e-01 -1.97156903e-13 -4.30769231e-01]
 [7.69230769e-02 5.00000000e+00 9.92307692e+00]
 [4.15384615e-01 -2.42929631e-13 -4.15384615e-01]
 [9.23076923e-02 6.00000000e+00 1.19076923e+01]

[4.00000000e-01 -2.87217996e-13 -4.00000000e-01]
[1.07692308e-01 7.00000000e+00 1.38923077e+01]
[3.84615385e-01 -3.28300313e-13 -3.84615385e-01]
[1.23076923e-01 8.00000000e+00 1.58769231e+01]
[3.69230769e-01 -3.66065711e-13 -3.69230769e-01]
[1.38461538e-01 9.00000000e+00 1.78615385e+01]
[3.53846154e-01 -3.92206420e-13 -3.53846154e-01]
[1.53846154e-01 1.00000000e+01 1.98461538e+01]
[3.38461538e-01 -4.04849623e-13 -3.38461538e-01]
[1.69230769e-01 1.10000000e+01 2.18307692e+01]
[3.23076923e-01 -4.03822377e-13 -3.23076923e-01]
[1.84615385e-01 1.20000000e+01 2.38153846e+01]
[3.07692308e-01 -3.95710424e-13 -3.07692308e-01]
[2.00000000e-01 1.30000000e+01 2.58000000e+01]
[2.92307692e-01 -3.77142114e-13 -2.92307692e-01]
[2.15384615e-01 1.40000000e+01 2.77846154e+01]
[2.76923077e-01 -3.61595674e-13 -2.76923077e-01]
[2.30769231e-01 1.50000000e+01 2.97692308e+01]
[2.61538462e-01 -3.49168379e-13 -2.61538462e-01]
[2.46153846e-01 1.60000000e+01 3.17538462e+01]
[2.46153846e-01 -3.39935101e-13 -2.46153846e-01]
[2.61538462e-01 1.70000000e+01 3.37384615e+01]
[2.30769231e-01 -3.27139018e-13 -2.30769231e-01]
[2.76923077e-01 1.80000000e+01 3.57230769e+01]
[2.15384615e-01 -3.14196876e-13 -2.15384615e-01]
[2.92307692e-01 1.90000000e+01 3.77076923e+01]
[2.00000000e-01 -2.87442450e-13 -2.00000000e-01]
[3.07692308e-01 2.00000000e+01 3.96923077e+01]
[1.84615385e-01 -2.46819796e-13 -1.84615385e-01]
[3.23076923e-01 2.10000000e+01 4.16769231e+01]
[1.69230769e-01 -2.12884720e-13 -1.69230769e-01]
[3.38461538e-01 2.20000000e+01 4.36615385e+01]
[1.53846154e-01 -1.85690856e-13 -1.53846154e-01]
[3.53846154e-01 2.30000000e+01 4.56461538e+01]
[1.38461538e-01 -1.58396239e-13 -1.38461538e-01]
[3.69230769e-01 2.40000000e+01 4.76307692e+01]
[1.23076923e-01 -1.24117372e-13 -1.23076923e-01]
[3.84615385e-01 2.50000000e+01 4.96153846e+01]
[1.07692308e-01 -1.03550590e-13 -1.07692308e-01]
[4.00000000e-01 2.60000000e+01 5.16000000e+01]
[9.23076923e-02 -8.98398020e-14 -9.23076923e-02]
[4.15384615e-01 2.70000000e+01 5.35846154e+01]

[7.69230769e-02 -7.60937651e-14 -7.69230769e-02]
[4.30769231e-01 2.80000000e+01 5.55692308e+01]
[6.15384615e-02 -6.92400886e-14 -6.15384615e-02]
[4.46153846e-01 2.90000000e+01 5.75538462e+01]
[4.61538462e-02 -4.85098719e-14 -4.61538462e-02]
[4.61538462e-01 3.00000000e+01 5.95384615e+01]
[3.07692308e-02 -2.77422951e-14 -3.07692308e-02]
[4.76923077e-01 3.10000000e+01 6.15230769e+01]
[1.53846154e-02 -1.38816408e-14 -1.53846154e-02]
[4.92307692e-01 3.20000000e+01 6.35076923e+01]]

Ao fazer a multiplicação *np.dot(W,X)* para obter novamente a matriz original A, Obtém-se o seguinte resultado:

A = [[1. 1. 1.]
[1. 2. 3.]
[1. 3. 5.]
[1. 4. 7.]
[1. 5. 9.]
[1. 6. 11.]
[1. 7. 13.]
[1. 8. 15.]
[1. 9. 17.]
……..
[1. 60. 119.]
[1. 61. 121.]
[1. 62. 123.]
[1. 63. 125.]
[1. 64. 127.]]

A coincide perfeitamente com a matriz **A** original.

d) n = 20, m = 17, W i,j = 1/(i + j − 1), se |i − j| ≤ 4 e W i,j = 0, se |i − j| > 4. Defina p = 3, resolvendo 3 sistemas simultâneos, com A(i, 1) = 1, A(i, 2) = i, A(i, 3) = 2i − 1, i = 1, n.

Cria-se as matrizes usadas no teste.

```
n = 20 #número de linhas
m = 17 #número de colunas
W = np.zeros((n,m))
for i in range(n):
    for j in range(m):
        if abs(i-j)>4:
            W[i][j] = 0
        else:
            W[i][j] = 1/(i+j-1+2) #+2 faz a correção pois i e j começam do zero.

b = np.ones((3,n))
for i in range(n):
    b[1][i] = i+1
    b[2][i] = 2*(i+1) - 1
b = b.transpose()
```

O resultado obtido foi:

X = [[2.88155063e+00 5.63578080e+01 1.09834065e+02]
 [-1.83376253e+00 -4.58748489e+01 -8.99159352e+01]
 [-1.51398904e+00 -4.34890237e+01 -8.54640584e+01]
 [-1.52190762e+00 -4.85765491e+01 -9.56311906e+01]
 [-4.53794614e-01 -3.01402000e+01 -5.98266054e+01]
 [5.85669889e+00 8.98121189e+01 1.73767539e+02]
 [3.42192891e+00 4.87136480e+01 9.40053671e+01]
 [3.65656219e+00 5.92392471e+01 1.14821932e+02]
 [1.20368454e+00 1.14463570e+01 2.16890294e+01]
 [6.12534249e+00 1.09162736e+02 2.12200129e+02]
 [-2.47971396e+00 -7.28727548e+01 -1.43265796e+02]
 [-1.47793147e+00 -5.43628927e+01 -1.07247854e+02]
 [-1.20390258e+00 -5.14444172e+01 -1.01684932e+02]
 [1.28414690e-01 -2.50148146e+01 -5.01580440e+01]
 [6.50158900e+00 9.85719388e+01 1.90642289e+02]
 [1.14910560e+01 2.18658859e+02 4.25826661e+02]
 [1.45805258e+01 2.98400228e+02 5.82219929e+02]]

Ao fazer a multiplicação *np.dot(W,X)* para obter novamente a matriz original A, Obtém-se o seguinte resultado:

A = [[0.98877052 0.75186507 0.51495961]
 [0.90768119 0.10665801 -0.69436518]
 [0.99310105 2.85672709 4.72035313]
 [1.1525159   7.19515482 13.23779375]
 [1.10779778 7.21887431 13.32995083]
 [0.92750055 4.4750944   8.02268825]

[0.96801475 6.33226369 11.69651264]
[0.98253494 7.63676415 14.29099336]
[0.98945734 8.78132851 16.57319969]
[0.96516387 9.2397249 17.51428592]
[0.80186148 6.93986267 13.07786386]
[0.99797545 11.95799118 22.91800692]
[1.27710297 18.80334155 36.32958013]
[1.17679316 17.63728064 34.09776811]
[0.88153264 12.50894908 24.13636552]
[0.94973366 14.95092908 28.95212449]
[0.9734249 16.44748786 31.92155082]
[0.98436988 17.67594688 34.36752389]
[0.95157654 17.94389214 34.93620774]
[0.73333049 14.53629117 28.33925185]]

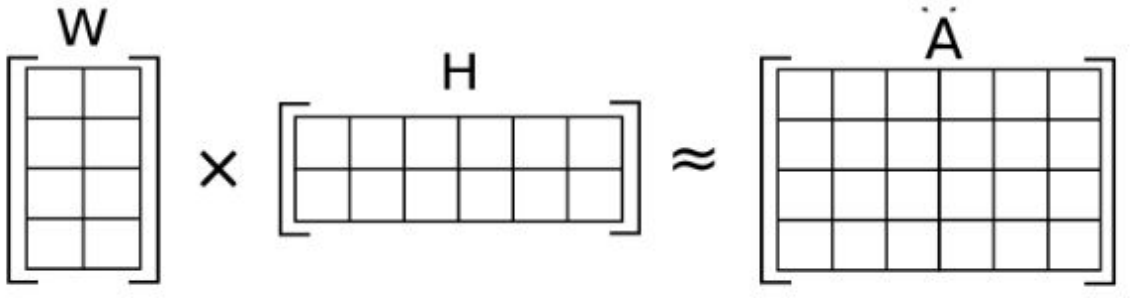
Observa-se uma maior discrepância dos resultados desse item em comparação aos itens anteriores. Novamente, temos a hipótese que a diferença seja devido a matriz W não ser quadrada.

3 SEGUNDA TAREFA

3.1 Objetivo

A tarefa 2 teve como objetivo fazer um algoritmo que descobre a fatoração positiva **WH** a partir de uma matriz $\mathbf{A}_{(n \times m)}$ dada. A matriz $\mathbf{W}_{(n \times p)}$ possui mais linhas que colunas (n > p) e a matriz $\mathbf{H}_{(p \times m)}$ (m > p) possui mais colunas que linhas. O produto matricial **WH** deve ser igual a matriz original $\mathbf{A}_{(n \times m)}$, porém **WH** não são necessariamente únicas. Na tarefa principal vemos que a matriz **W** representa o que foi "aprendido" do treinamento de um dígito.

$$\mathbf{W}_{(n \times p)} \times \mathbf{H}_{(p \times m)} = \mathbf{A}_{(n \times m)}$$



3.2 Código

O código foi dividido em uma série de funções para o código ser mais modular e conciso.

3.2.1 normaliza

A função normaliza(W) recebe uma matriz W e retorna W com as suas colunas normalizadas, ou seja, de módulo igual a 1. Isso é feito da seguinte forma: W é transposta, para que seja mais fácil trabalhar com as colunas; com um loop, para cada linha de W^t é calculada o módulo da linha, caso o módulo não seja nulo, a linha é dividida pelo módulo. Posteriormente, a W^t é transposta novamente para se obter W com as colunas normalizadas.

```
#Normaliza (tranforma em módulo 1) cada coluna de W
def normaliza(W): #Funciona
    W = W.transpose()
    for i in range(W.shape[0]):
        divisor = np.sum(np.square(W[i]))**(1/2)
        if divisor != 0:
            W[i] = W[i]/divisor
    return W.transpose()
```

3.2.2 normaErro

A função normaErro(A, W, H) recebe a matriz original A e as suas fatorações não negativas W e H e retorna a soma dos quadrados das diferenças de todas as entradas entre a matriz A com a matriz resultante do produto WH.

```
#Calcula a norma quadrática do Erro entre os termos da Matriz original com os das matrizes fatoradas
def normaErro(A, W, H):
    E = A - np.dot(W,H)
    E = np.sum(np.square(E))
    return E
```

3.2.3 main

A função main da Tarefa 2 foi feita com base no enunciado. Ela recebe uma matriz A e um número inteiro p. Esse inteiro diz qual vai ser a dimensão das matrizes fatoradas provenientes de A.

Isso é feito da seguinte forma: Inicializa-se uma matriz $W_{(n\,x\,p)}$ com números aleatórios; guarda-se uma cópia de A. Em seguida, dentro de um loop de no máximo 100 iterações: a matriz W é normalizada usando a função “normaliza”; resolve-se um sistema linear simultâneo $WH = A$ para descobrir-se $H_{(p\,x\,m)}$; H então é redefinida para que suas entradas negativas tornarem-se nulas; resolve-se um segundo sistema linear simultâneo $H^tW^t = A^t$ para descobrir-se $W^t_{(p\,x\,n)}$; transpõe-se W^t para chegar-se em W; W então é redefinida para que suas

entradas negativas tornarem-se nulas; por fim, é calculada a norma do erro usando a função “normaErro”, caso o erro quadrático seja menor que 10^{-5}, então o laço for é encerrado. Então, a função retorna as matrizes W e H obtidas por todas essas operações.

```python
#Main foi feita seguindo o enunciado
def main(A, p):
    E_0 = 1
    n = A.shape[0] #Número de linhas de A
    m = A.shape[1] #Número de colunas de A

    #Inicialize um W(nxp) randômico com valores positivos
    W = np.random.rand(n,p)

    copia = np.copy(A) #Cópia de A

    #Enquanto diferença da norma dos erros consecutivos > 10^-5 ou itmax = 100
    itmax = 100
    for i in range(0, itmax):
        W = normaliza(W) #Normalize W

        A = np.copy(copia)
        H = Tarefa1.Sol_linSis_simult(W,A,p) #Resolva MMQ WH = A, determinando H

        #Redefina H
        H[H<0] = 0

        A = np.copy(copia)
        #Resolva MMQ H_t W_t = A_t, determinando W_t
        W = Tarefa1.Sol_linSis_simult(H.transpose(),A.transpose(),p)

        W = W.transpose()
        W[W<0] = 0 #Redefina W

        #Calcular erro
        E_1 = normaErro(A, W, H)
        if ((abs(E_1 - E_0) < (10 ** -5)) & (i != 0)):
            break
        else:
            E_0 = E_1
    return W, H
```

3.3 Resultado dos testes

O teste apresentado no enunciado pedia que fosse feita a fatoração positiva da seguinte matriz:

$$A = \begin{pmatrix} 3/10 & 3/5 & 0 \\ 1/2 & 0 & 1 \\ 4/10 & 4/5 & 0 \end{pmatrix}$$

$$W = \begin{pmatrix} 3/5 & 0 \\ 0 & 1 \\ 4/5 & 0 \end{pmatrix}$$

$$H = \begin{pmatrix} 1/2 & 1 & 0 \\ 1/2 & 0 & 1 \end{pmatrix}$$

Os resultados obtidos foram:

```
W
[[2.48112638e-05 6.00024805e-01]
 [9.99999969e-01 2.35894291e-17]
 [3.30816850e-05 8.00033073e-01]]
H
[[0.50000002 0.         1.00000003]
 [0.49987595 1.         0.        ]]
O resultado da fatoração WH é:
[[2.99950377e-01 6.00024805e-01 2.48112645e-05]
 [5.00000000e-01 2.35894291e-17 1.00000000e+00]
 [3.99933837e-01 8.00033073e-01 3.30816861e-05]]
```

Vale ressaltar que, em relação às matrizes WH dadas no enunciado, as matrizes obtidas estão com as colunas (na matriz W) e linhas (na matriz H) invertidas. Porém, em outras execuções do código, obteve-se um resultado exatamente igual ao do enunciado, ou seja, dentro da margem de erro de 10^{-5}. Isso se deve ao fato de que as fatorações WH não serem únicas.

Abaixo estão expostos alguns outros exemplos de testes que foram realizados e seus resultados.

- Teste 2:
  - Matriz: $[[4, 6, 0], [6, 4, 0], [0, 0, 0]]$
  - $p = 2$ e $p = 3$:

```
O resultado da fatoração WH é:
[[4. 6. 0.]
 [6. 4. 0.]
 [0. 0. 0.]]
```

- Teste 3:
  - Matriz: $[[1, 2, 3, 5], [2, 4, 8, 12], [3, 6, 7, 13]]$
  - $p = 2$:

```
O resultado da fatoração WH é:
[[ 1.00004944  2.00009888  3.00012351  4.99987644]
 [ 2.00019711  4.00039423  8.00049245 11.99950738]
 [ 3.00005008  6.00010017  7.00012512 12.99987483]]
```

  - $p = 3$:

```
O resultado da fatoração WH é:
[[ 1.          2.          3.          5.        ]
 [ 2.00001506  4.00003012  8.00001402 12.00004414]
 [ 3.          6.          7.         13.        ]]
```

- Teste 4:
  - Matriz:
    $[[4, 6, 8, 5, 7], [2, 3, 4, 5, 1], [4, 6, 8, 5, 7], [2, 3, 4, 5, 1], [4, 6, 8, 5, 7]]$
  - $p = 2$:

```
O resultado da fatoração WH é:
[[4. 6. 8. 5. 7.]
 [2. 3. 4. 5. 1.]
 [4. 6. 8. 5. 7.]
 [2. 3. 4. 5. 1.]
 [4. 6. 8. 5. 7.]]
```

  - $p = 3$:

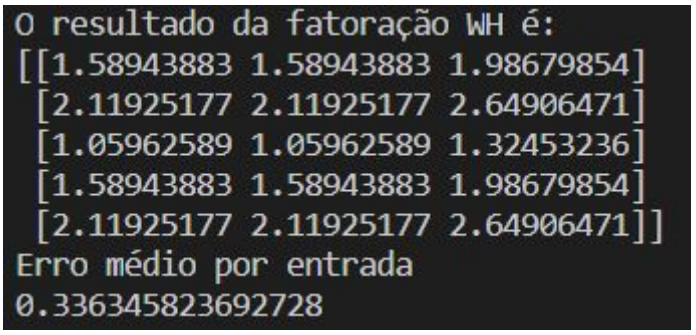
```
O resultado da fatoração WH é:
[[3.99998247 5.99997371 7.99996494 5.00004236 7.00004236]
 [1.99994285 2.99991427 3.9998857  5.0001381  1.00013812]
 [3.99998247 5.99997371 7.99996494 5.00004236 7.00004236]
 [1.99994285 2.99991427 3.9998857  5.0001381  1.00013812]
 [3.99998247 5.99997371 7.99996494 5.00004236 7.00004236]]
```

  - $p = 4$:

```
O resultado da fatoração WH é:
[[3.99996193 5.99994289 7.99992386 5.000092   7.000092  ]
 [1.99997147 2.99995721 3.99994294 5.00006894 1.00006894]
 [3.99996193 5.99994289 7.99992386 5.000092   7.000092  ]
 [1.99997147 2.99995721 3.99994294 5.00006894 1.00006894]
 [3.99996193 5.99994289 7.99992386 5.000092   7.000092  ]]
```

- Teste 5:

- Matriz: [[1,1,4],[3,2,3],[1,3,1],[1,4,3],[5,2,2]]
- $p = 2$:



Em relação aos testes, os 4 primeiros foram um grande sucesso, obtendo aproximações excelentes para a matriz original usando a fatoração WH. Porém, o teste 5 não apresenta uma grande precisão, o que leva a crer que os erros inerentes da aritmética de ponto flutuante causam grandes distorções no resultado das operações.

Para calcular o erro médio por entrada, presente no teste 5, subtraiu-se o resultado do produto das matrizes W e H pela matriz original. Então as diferenças das entradas foram elevadas ao quadrado, somadas e tiradas a raiz quadrada, depois divididas pelo número de entradas. No teste 5 vemos que o erro médio ficou bem acima da precisão estabelecida de 10^{-5}, isso se deve ao fato do loop principal da tarefa 2 ter sido limitado a 100 iterações. Logo, quando esse número máximo foi atingido, a precisão ainda não era a desejada, por isso obteve-se o resultado um pouco diferente. Caso o número de iterações fosse maior, pode-se ter certeza que a precisão também seria.

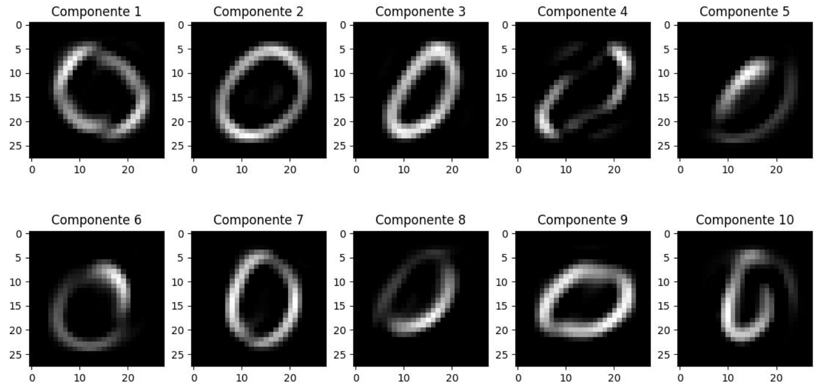
4 TAREFA PRINCIPAL

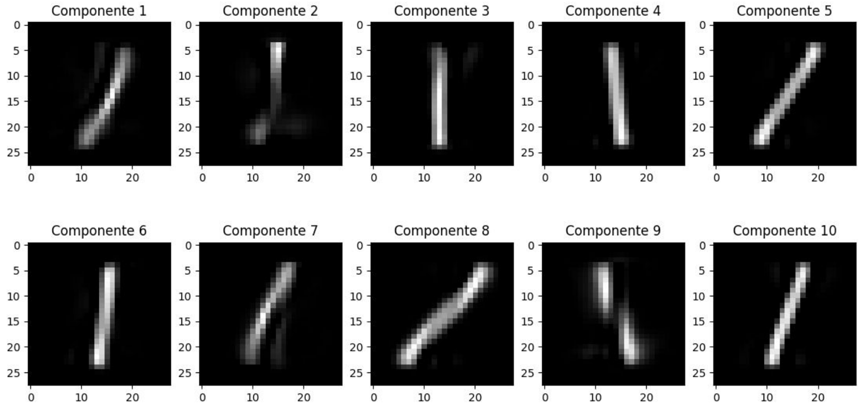
4.1 Objetivo

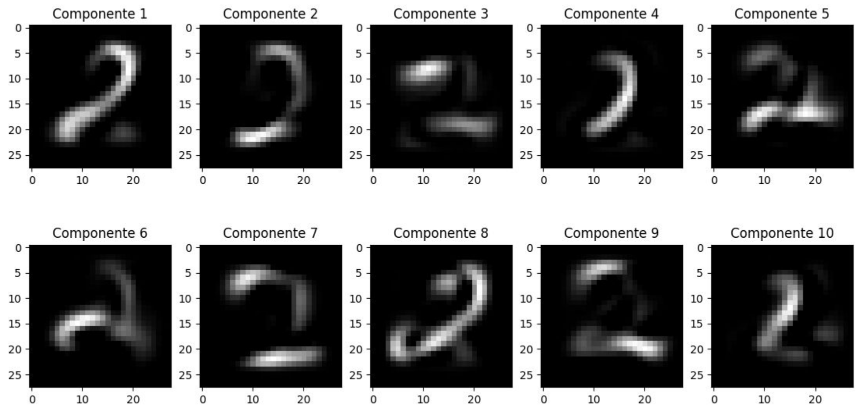
O objetivo da tarefa principal é integrar tanto a Tarefa 1 quanto a Tarefa 2 para a classificação dos dígitos, tal que haja utilização da base dados MNIST. Além disso, esta tarefa pode ser dividida em duas etapas principais: Aprendizagem e classificação. Aqui que se encontra o algoritmo de Machine Learning em si.

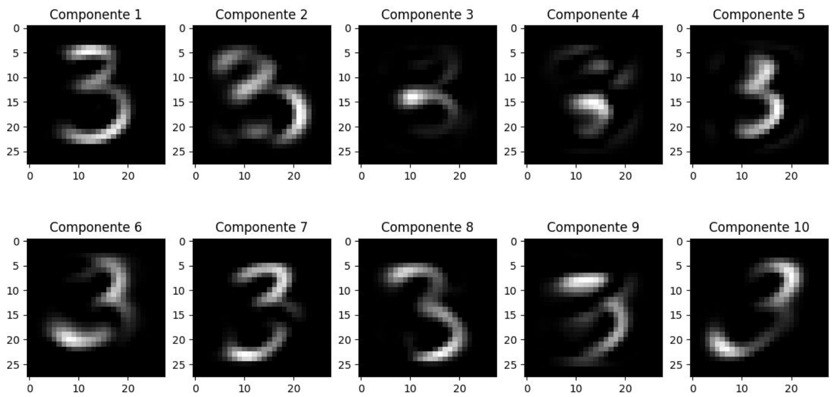
No treinamento, abrimos o arquivo *train_dig0.txt* contendo a matriz A, ou seja, uma matriz com centenas de colunas, onde cada coluna representa uma imagem. Realizamos a decomposição **A = WH** apresentado na Tarefa 2, descartamos **H**, e obtemos $\mathbf{W}_0$, ou seja, uma representação do aprendizado do que é o dígito 0. Realizamos isso para todos os outros dígitos.

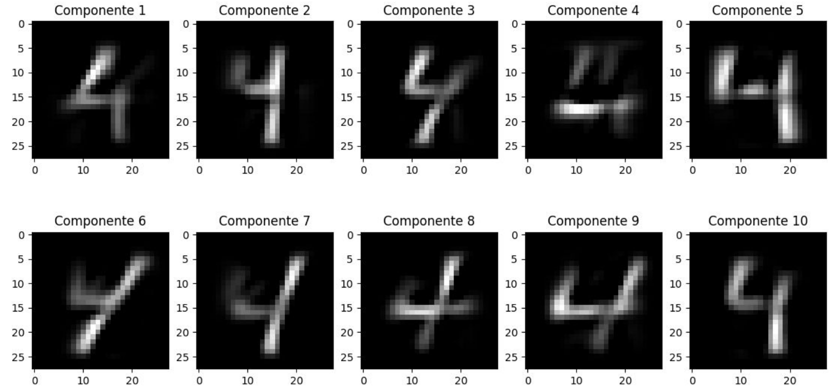
Se as dimensões de W forem nxp, teremos p componentes do dígito d de W. Como representado nas imagens a seguir, para todos os dígitos.

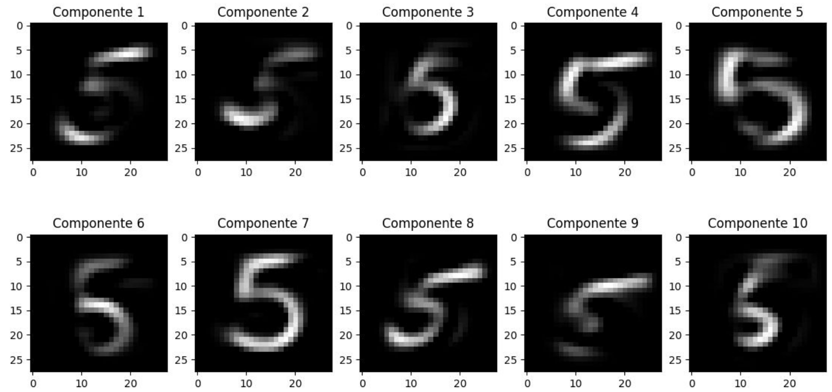

Componente 1
Componente 2
Componente 3
Componente 4
Componente 5
Componente 6
Componente 7
Componente 8
Componente 9
Componente 10

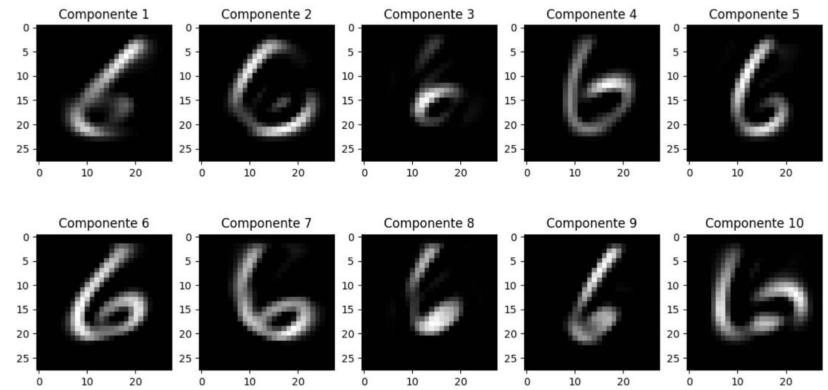

Componente 1
Componente 2
Componente 3
Componente 4
Componente 5
Componente 6
Componente 7
Componente 8
Componente 9
Componente 10

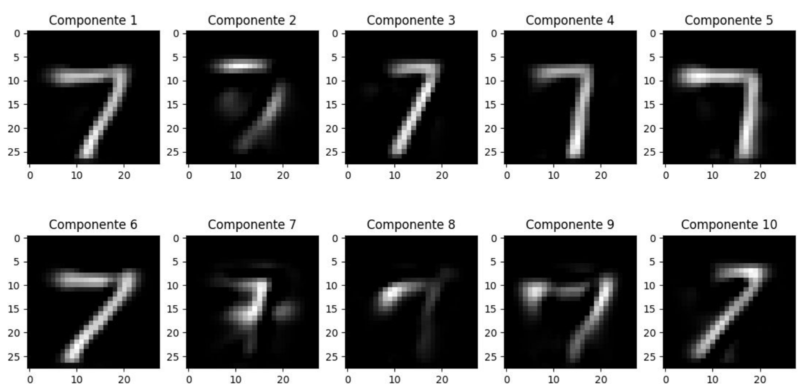

Componente 1
Componente 2
Componente 3
Componente 4
Componente 5
Componente 6
Componente 7
Componente 8
Componente 9
Componente 10

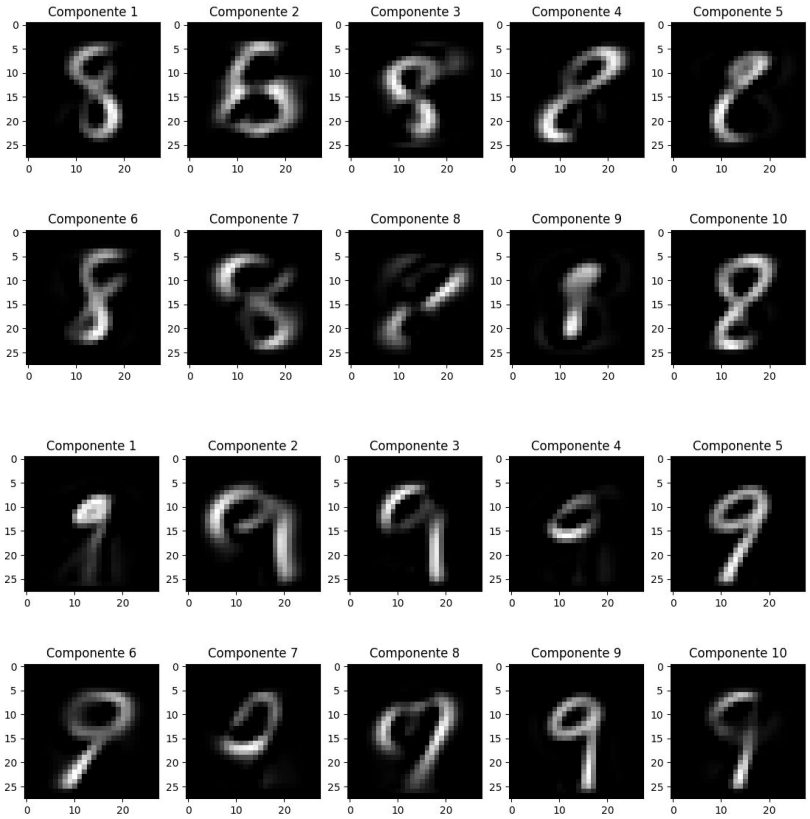

Componente 1
Componente 2
Componente 3
Componente 4
Componente 5
Componente 6
Componente 7
Componente 8
Componente 9
Componente 10


Componente 1
Componente 2
Componente 3
Componente 4
Componente 5
Componente 6
Componente 7
Componente 8
Componente 9
Componente 10


Componente 1
Componente 2
Componente 3
Componente 4
Componente 5
Componente 6
Componente 7
Componente 8
Componente 9
Componente 10


Componente 1
Componente 2
Componente 3
Componente 4
Componente 5
Componente 6
Componente 7
Componente 8
Componente 9
Componente 10


Componente 1
Componente 2
Componente 3
Componente 4
Componente 5
Componente 6
Componente 7
Componente 8
Componente 9
Componente 10



Após a etapa de treinamento, é realizado a classificação. Para isso, utiliza-se a Tarefa 1 para resolver sistemas simultâneos determinando a matriz **H** como decomposição de **A** em $\mathbf{W}_d\mathbf{H}$, tal que $\mathbf{W}_d$ represente uma das matrizes da etapa de treinamento e **A** o conjunto de imagens que se deseja classificar.

$$\mathbf{W}_d\mathbf{H} = \mathbf{A}$$

Com isso, definimos

$$\mathbf{C} = \mathbf{A} - \mathbf{W}_d\mathbf{H}$$

onde **C** é composto por colunas $\mathbf{c}_j$, tal que a norma quadrática de $\mathbf{c}_j$ represente o erro entre uma imagem $\mathbf{a}_i$ e um dígito **d**. Fazemos isso para todos os dígitos e todas as imagens. Para cada imagem, indicamos que ela representa um dígito **d**, se o seu respectivo erro for o menor.

Por fim, comparando com o índice dos dados, é possível calcular uma taxa de acerto.

4.2 Código

4.2.1 treinamento

A função *treinamento* recebe uma série de parâmetros. O primeiro, p, é o número de componentes que se deseja obter no treinamento. Depois, três parâmetros booleanos, salvar (se deseja salvar o treinamento ou não), abrir (se deseja abrir um treinamento já salvo), visualizar (caso deseje usar o matplotlib para visualizar o treinamento. Foi feito apenas para p = 10, não usar com p diferente de 10). Além disso, temos o endereço do arquivo que se salva/abre e qtd como quantidade de dígitos que se deseja abrir, começando do 0.

Após o treinamento, a função retorna a *array* **W** que se trata de um *array* onde cada índice representa o treinamento do dígito d, ou seja, a decomposição $\mathbf{A} = \mathbf{W}_d\mathbf{H}$ onde **A** representa o conjunto de imagens para treinamento e $\mathbf{W}_d$ a resultante.

```
def treinamento(p,salvar,abrir,visualizar,arquivo,qtd,ndig_treino):

    Wtemp = 0
    if abrir == True:
        W = np.zeros((qtd,784,p))
        for i in range(qtd):
            Wtemp = np.loadtxt(arquivo+str(i)+'.txt') #Abre o treinamento
            W[i] = Wtemp
    else:
        W = np.zeros((qtd,784,p))
        for d in range(qtd):
            A = np.loadtxt('dados_mnist/train_dig'+str(d)+'.txt')[:,:ndig_treino]
            Wtemp , H = fatorWH(A,p)
            W[d] = Wtemp
            print(str(d+1)+'/'+str(qtd)+' Treinos realizados')

        if salvar == True:
            print('Salvando treino realizado')
            for d in range(qtd):
                np.savetxt(arquivo+str(d)+'.txt',W[d])

    if visualizar == True: #Só pode ser usado com p = 10
        print('visualizando dados')
        #fig, axs = plt.subplots(2,5,figsize=(10,10))
        for d in range(qtd):
            k=0
            print('visualizando dígito '+str(d+1)+'/'+str(qtd))
            fig, axs = plt.subplots(2,5,figsize=(28,28))
            for i in range(2):
                for j in range(5):
                    try:
                        Aux = W[d].transpose()[k].reshape((28,28))*255
                        axs[i,j].imshow(Aux, cmap = 'gray')
                        axs[i,j].set_title("Componente "+str(k+1))
                    except:
                        print('Deu ruim')
                        pass
                    k+=1
            plt.show()

    return W
```

4.2.2 classifDigitos

A outra parte principal desta etapa, a função *classifDigitos,* que recebe como parâmetros a *array* retornada da função treinamento, **W**, a matriz que contém os dígitos os quais se deseja classificar **A** e o número de componentes p de **W**. Com isso, segue o procedimento apresentado no item 4.1 para realizar essa classificação.

A função retorna dois vetores:

- *dmpM*: da sigla, dígito mais provável matriz, ou seja, um array que contém o resultado da classificação, o dígito com o menor erro quadrático para cada imagem.
- *menorErroM*: matriz que contém o respectivo menor erro para cada dígito apresentado na *dmpM*.

```
def classifDigitos(W,A,p):

    qtd = W.shape[0]
    #H = np.zeros((W[0].shape[1],A.shape[1],10))
    # print(A.shape)
    C = np.zeros((qtd,A.shape[0],A.shape[1]))
    for d in range(qtd):
        H = Tarefa1.Sol_linSis_simult(W[d].copy(),A.copy(),p)
        C[d] = A.copy() - np.dot(W[d].copy(),H)
        # print(C.shape)
    qtdanalisar = C.shape[2]

    dmpM = np.zeros(qtdanalisar) #matriz dígito mais provável
    menorErroM = np.zeros(qtdanalisar) #Matriz com o menor erro do correspondente dígito mais provável
    for i in range(qtdanalisar): #varre imagens, ou seja, varre colunas, 10000 colunas
        menorErro = np.inf #número infinito
        dmp = -1 # dígito mais provavél

        for d in range(qtd): # Varre os dígitos 0 à qtd
            C_temp = C[d].transpose() #Dessa forma consigo pegar uma coluna cj como se fosse uma linha

            cj = C_temp[i]
            norma = np.sum(np.square(cj))**(1/2) #Calcula norma quadrática

            if norma < menorErro:
                menorErro = norma
                dmp = d

        dmpM[i] = dmp #Salva dmp na matriz
        menorErroM[i] = menorErro #Salva menor erro na matriz

    return dmpM, menorErroM
```

4.2.3 CalcularTaxas

A função *CalcularTaxas* recebe como parâmetros o *dmpM* e um vetor *labels* que contém o índice dos resultados esperados. Inicialmente conta-se o número de dígitos classificados corretamente para posteriormente ser possível a comparação destes com o número de dígitos correspondentes no vetor *labels,* obtendo as taxas.

Esta função printa as taxas no terminal.

```
def CalcularTaxas(dmpM,labels):
    cont = 0
    contd = np.zeros(10)
    for i in range(dmpM.shape[0]):
        if dmpM[i] == labels[i]:
            cont +=1
            d = int(dmpM[i])
            contd[d] +=1
    taxa = round(cont/labels.shape[0]*100,2)

    print('Percentual Acertos total: ' + str(taxa)+'%')
    print('Acertos absolutos: '+str(cont))

    for d in range(10):
        qtdNaLabels = list(labels.flatten()).count(d)
        taxa = round(contd[d]/qtdNaLabels*100,2)
        print('Percentual acertos dígito ' + str(d) + ': ' + str(taxa)+'% '+
            '('+str(int(contd[d]))+'/'+str(qtdNaLabels)+')')

    return
```

4.2.4 main

Por fim, a função main, que sintetiza todo programa. Vale ressaltar que qualquer alteração nos parâmetros de treinamento, nome do arquivo, se deseja salvar/abrir ou visualizar o arquivo devem ser feitas aqui. Essa função recebe como parâmetros p e ndig_treino, que são variados nos testes a seguir.

```
def main(p,ndig_treino):
    print('Início do programa')
    print('Carregando matriz com imagens teste...')
    A = np.loadtxt('dados_mnist/test_images.txt')
    print('Realizando treinamento...')
    W = treinamento(p,True,False,False,'treinamento',10,ndig_treino) #Treina e Salva
    #W = treinamento(p,False,True,False,'treinamento',10,ndig_treino) #Abre
    print('Classificando digítos...')
    dmpM, menorErroM = classifDigitos(W,A,p)
    print('Calculando taxas...')
    print('Parâmetros')
    print('p = ' + str(p))
    print('ndig_treino: '+str(ndig_treino))
    labels = np.loadtxt('dados_mnist/test_index.txt')
    CalcularTaxas(dmpM,labels)

    print('Fim do programa')
    return
```

4.3 Resultado dos testes

4.3.1 Teste (p = 5; ndig_treino = 100)

Percentual Acertos total: 87.71%
Acertos absolutos: 8771
Percentual acertos dígito 0: 97.14% (952/980)

Percentual acertos dígito 1: 99.47% (1129/1135)
Percentual acertos dígito 2: 84.98% (877/1032)
Percentual acertos dígito 3: 83.56% (844/1010)
Percentual acertos dígito 4: 78.82% (774/982)
Percentual acertos dígito 5: 83.41% (744/892)
Percentual acertos dígito 6: 93.01% (891/958)
Percentual acertos dígito 7: 87.84% (903/1028)
Percentual acertos dígito 8: 79.57% (775/974)
Percentual acertos dígito 9: 87.41% (882/1009)

4.3.2 Teste (p = 10; ndig_treino = 100)

Percentual Acertos total: 89.93%
Acertos absolutos: 8993
Percentual acertos dígito 0: 97.65% (957/980)
Percentual acertos dígito 1: 99.3% (1127/1135)
Percentual acertos dígito 2: 89.73% (926/1032)
Percentual acertos dígito 3: 84.36% (852/1010)
Percentual acertos dígito 4: 89.31% (877/982)
Percentual acertos dígito 5: 86.43% (771/892)
Percentual acertos dígito 6: 93.63% (897/958)
Percentual acertos dígito 7: 90.86% (934/1028)
Percentual acertos dígito 8: 79.36% (773/974)
Percentual acertos dígito 9: 87.12% (879/1009)

4.3.3 Teste (p = 15; ndig_treino = 100)

Percentual Acertos total: 90.43%
Acertos absolutos: 9043
Percentual acertos dígito 0: 98.06% (961/980)
Percentual acertos dígito 1: 99.56% (1130/1135)
Percentual acertos dígito 2: 90.02% (929/1032)
Percentual acertos dígito 3: 84.95% (858/1010)
Percentual acertos dígito 4: 87.68% (861/982)
Percentual acertos dígito 5: 84.3% (752/892)
Percentual acertos dígito 6: 95.62% (916/958)
Percentual acertos dígito 7: 94.55% (972/1028)
Percentual acertos dígito 8: 80.6% (785/974)
Percentual acertos dígito 9: 87.12% (879/1009)

4.3.4 Teste (p = 5; ndig_treino = 1000)

Percentual Acertos total: 90.79%
Acertos absolutos: 9079
Percentual acertos dígito 0: 97.55% (956/980)
Percentual acertos dígito 1: 99.12% (1125/1135)
Percentual acertos dígito 2: 87.6% (904/1032)
Percentual acertos dígito 3: 90.89% (918/1010)
Percentual acertos dígito 4: 84.62% (831/982)
Percentual acertos dígito 5: 88.34% (788/892)
Percentual acertos dígito 6: 96.24% (922/958)
Percentual acertos dígito 7: 89.3% (918/1028)
Percentual acertos dígito 8: 85.93% (837/974)
Percentual acertos dígito 9: 87.22% (880/1009)

4.3.5 Teste (p = 10; ndig_treino = 1000)

Percentual Acertos total: 92.69%
Acertos absolutos: 9269
Percentual acertos dígito 0: 98.57% (966/980)
Percentual acertos dígito 1: 99.38% (1128/1135)
Percentual acertos dígito 2: 89.92% (928/1032)
Percentual acertos dígito 3: 91.39% (923/1010)
Percentual acertos dígito 4: 92.87% (912/982)
Percentual acertos dígito 5: 88.34% (788/892)
Percentual acertos dígito 6: 95.09% (911/958)
Percentual acertos dígito 7: 91.63% (942/1028)
Percentual acertos dígito 8: 88.5% (862/974)
Percentual acertos dígito 9: 90.09% (909/1009)

4.3.6 Teste (p = 15; ndig_treino = 1000)

Percentual Acertos total: 93.39%
Acertos absolutos: 9339
Percentual acertos dígito 0: 98.78% (968/980)
Percentual acertos dígito 1: 99.38% (1128/1135)
Percentual acertos dígito 2: 91.86% (948/1032)
Percentual acertos dígito 3: 91.49% (924/1010)
Percentual acertos dígito 4: 91.75% (901/982)
Percentual acertos dígito 5: 90.02% (803/892)
Percentual acertos dígito 6: 96.35% (923/958)
Percentual acertos dígito 7: 93.09% (957/1028)
Percentual acertos dígito 8: 87.99% (857/974)

Percentual acertos dígito 9: 92.17% (930/1009)

4.3.7 Teste (p = 5; ndig_treino = 4000)

Percentual Acertos total: 91.76%
Acertos absolutos: 9176
Percentual acertos dígito 0: 97.55% (956/980)
Percentual acertos dígito 1: 99.47% (1129/1135)
Percentual acertos dígito 2: 89.92% (928/1032)
Percentual acertos dígito 3: 93.17% (941/1010)
Percentual acertos dígito 4: 87.88% (863/982)
Percentual acertos dígito 5: 87.56% (781/892)
Percentual acertos dígito 6: 96.87% (928/958)
Percentual acertos dígito 7: 88.52% (910/1028)
Percentual acertos dígito 8: 89.01% (867/974)
Percentual acertos dígito 9: 86.52% (873/1009)

4.3.8 Teste (p = 10; ndig_treino = 4000)

Percentual Acertos total: 93.4%
Acertos absolutos: 9340
Percentual acertos dígito 0: 98.88% (969/980)
Percentual acertos dígito 1: 99.56% (1130/1135)
Percentual acertos dígito 2: 91.57% (945/1032)
Percentual acertos dígito 3: 93.27% (942/1010)
Percentual acertos dígito 4: 93.79% (921/982)
Percentual acertos dígito 5: 89.57% (799/892)
Percentual acertos dígito 6: 96.66% (926/958)
Percentual acertos dígito 7: 91.05% (936/1028)
Percentual acertos dígito 8: 88.71% (864/974)
Percentual acertos dígito 9: 89.99% (908/1009)

4.3.9 Teste (p = 15; ndig_treino = 4000)

Percentual Acertos total: 93.67%
Acertos absolutos: 9367
Percentual acertos dígito 0: 98.88% (969/980)
Percentual acertos dígito 1: 99.38% (1128/1135)
Percentual acertos dígito 2: 91.96% (949/1032)
Percentual acertos dígito 3: 91.68% (926/1010)
Percentual acertos dígito 4: 93.58% (919/982)
Percentual acertos dígito 5: 91.59% (817/892)
Percentual acertos dígito 6: 96.87% (928/958)
Percentual acertos dígito 7: 90.66% (932/1028)

Percentual acertos dígito 8: 89.84% (875/974)
Percentual acertos dígito 9: 91.58% (924/1009)

5 CONCLUSÃO

O reconhecimento de imagens e a teoria em que se baseia, chamada de *machine learning* (aprendizagem de máquina)*,* é uma das áreas do conhecimento em maior expansão nos últimos tempos. O futuro apresenta uma série de problemas que precisarão dessa tecnologia como solução, exemplos disso são carros autônomos, reconhecimento facial e muitos outros.

O exercício de programação apresentado tem como objetivo a realização de um algoritmo que reconhece e classifica dígitos manuscritos (provenientes da base de dados MNIST), por meio de *machine learning.* Isso foi feito em três partes, ou tarefas. A tarefa 1 pedia que fosse feito um algoritmo que transforma matrizes em triangulares superiores por meio do método de Rotação de Givens. A segunda tarefa pedia que fosse feito um algoritmo que descobre as fatorações não negativas WH de uma matriz, usando a tarefa 1. Por fim, a tarefa principal pedia que fosse feito o treinamento, classificação e avaliação do método de classificação de dígitos manuscritos.

Dois parâmetros importantes do problema são *p* e *ndig_treino*, onde *p* é o número de componentes resultantes do treinamento e *ndig_treino* é o número de dígitos da base de dados que serão usados no treinamento. Ambos os parâmetros estão relacionados à qualidade e a precisão do treinamento do programa desenvolvido, pois quanto maior o p mais informação será guardada do treinamento e portanto mais precisa será a classificação. Da mesma forma, quanto mais dígitos forem usados no treinamento, ou seja ndig_treino maior, maior será a quantidade de informação usada no treino e mais precisa será a classificação. Porém, um incremento em qualquer um dos dois parâmetros leva a um aumento do tempo de execução do código. Abaixo está uma tabela que relaciona a quantidade de acertos de classificação com os parâmetros usados:

Número absoluto de acertos	ndig_treino = 100	ndig_treino =1000	ndig_treino = 4000
p = 5	8771	9079	9176
p = 10	8993	9269	9340
p = 15	9043	9339	9367

Como se pode ver, a precisão é proporcional a *p* e a *ndig_treino.*

Outro detalhe da tarefa principal que merece ser mencionado é que, claramente, o dígito “8” foi o que o algoritmo teve maior dificuldade de classificar corretamente, obtendo os menores índices de acerto. Isso se evidencia na seção 4.3, onde se percebe que o dígito “8” teve o menor percentual de acerto em relação aos outros dígitos em todos os testes realizados, exceto nos testes 4.3.6 e 4.3.7. Especula-se que tem a ver com os formatos diferentes que o número “8” pode assumir, como pode ser observado na página 21, que pode ser confundido com um “0”, “1”, “2” e com “7”, dependendo das distorções caligráficas.

Sobre a tarefa 2 e a fatoração *WH*, observou-se que os resultados mais precisos da fatoração se obtiveram quando *W* era uma matriz quadrada. Isso se deve à própria característica da fatoração, onde se acha duas matrizes de dimensões iguais ou menores que a matriz original que quando multiplicadas, obtém-se a matriz original. Quando a matriz *W* é quadrada, perde-se menos informação da matriz original, pois as dimensões serão mais semelhantes à original. Caso contrário, quando *W* não é quadrada, perde-se informação das linhas originais, pois *W* sempre tem um número de linhas maior ou igual ao de colunas.

Ainda sobre a tarefa 2, observou-se que para matrizes grandes (Ex.: 20x8), o número de iterações requeridas para que o erro se estabilize ultrapassava o número máximo pré-definido de iterações para o loop principal do método. Ou seja, a precisão quadrática não fica abaixo de 10^{-5} em matrizes grandes, o que leva a distorções. Também testou-se com um número máximo maior de iterações, resultando no erro ainda assim não se estabilizando. Acredita-se que isso tem a ver com os erros de arredondamento inerentes dos métodos numéricos. Para matrizes pequenas, o erro se estabiliza rapidamente

Recapitulando o primeiro parágrafo da conclusão, *machine learning* está em plena ascensão e a carreira de cientista de dados está entre as com maior média salarial. Achamos que essa tarefa foi extremamente relevante para o nosso aprendizado e nossa formação pois, apesar de nós dois cursarmos engenharia mecânica, possuímos grande interesse pela área de computação e acreditamos que habilidades nessa área valerão muito para nosso futuro. Portanto, gostamos da tarefa e aprendemos muito sobre programação e aprendizado de máquina.

APÊNDICE A

Reprodução do arquivo LEIAME.txt, presente na pasta com os arquivos do código.

Exercício de Programação 1 - Métodos Numéricos e Aplicações [MAP3121]

Engenharia Mecânica - Turma 11
Ariel Guerchenzon - N°USP: 10335552

Matheus José Oliveira dos Santos - N°USP: 10335826

Instruções para compilação e execução do código desenvolvido:

Programas necessários:
- Python 3.7
- Numpy
- Matplotlib

Interpretador recomendado:
- IDLE Python 3.7 64-bit [IDLE];

Primeiro passo:
Tenha os arquivos "Tarefa1.py", "Tarefa2.py" e "TarefaPrincipal.py" dentro da mesma pasta.
Também tenha dentro desta pasta uma outra pasta que contém os dados_mnist, que deve ser chamada de "dados_mnist".

Instruções de execução:
Dentro do arquivo "TarefaPrincipal.py", dentro da função main(p,ndig_treino):, há uma linha de código da seguinte forma:
"W = treinamento(p,True,False,False,'treinamento',10,ndig_treino) #Treina e Salva"
Como se pode perceber, o 2º, 3º e 4º argumento da função são booleanos, eles significam que:
- 2º: True salva o arquivo resultante do treinamento, para não ter que treinar novamente na próxima vez que se rodar o código. Caso já tenha um arquivo treinado salvo, pode deixar como False;
- 3º: True abre o arquivo de treinamento já salvo. Se for a primeira vez que estiver rodando o código, deixe como False, pois não há nada a ser lido. Caso já tenha um arquivo salvo, pode deixar como True que esse arquivo será lido;
- 4º: True abre a janela do matplotlib para visualizar o treinamento. Caso não queira ver os dígitos resultantes do treinamento, deixe como False.
Os outros parâmetros são:
- p: número de componentes que serão geradas no treinamento;
- 'treinamento': nome do arquivo que será aberto/salvo o treinamento dependendo das suas escolhas nas variáveis booleanas;
- ndig_treino: número de dígitos manuscritos que serão lidos da base de dados para treinamento.

Como rodar o código:
[VSCode] - Abra o terminal.

Em seguida, por meio da linha de comando abra a pasta em que está salvo os arquivos ".py". (Ex.: cd ep1-numerico)

Então, digite "py TarefaPrincipal.py" (nem sempre o comando inicial é "py", dependendo da sua versão pode ser "python3" ou apenas "python").

O código rodará.

[IDLE] - Abra o arquivo TarefaPrincipal.py" no editor de texto.

Aperte o comando F5 ou selecione "Run" na barra acima.

O código rodará.

APÊNDICE B

Lista de funções do numpy utilizadas:

- np.sqrt()
- np.transpose()
- np.zeros()
- np.copy()
- np.eye()
- np.ones()
- np.dot()
- np.random.rand()
- np.sum()
- np.loadtxt()
- np.savetxt()
- np.reshape()

Lista de funções do matplotlib utilizadas:

- plt.subplots()
- plt.imshow()
- plt.show()
- plt.set_title()