Relatório sobre Fatoração de Matrizes para Sistemas de Classificação de *Machine Learning*

Heloisa de Lazari Bento
Nusp: 11093093

Lucca Alipio Pagnan
Nusp: 11259329

Novembro de 2020

1 Introdução

Esse exercício programa (EP) escrito em *Python 3.8.6* teve como objetivo classificar dígitos a partir de imagens dos mesmos escritos a mão. Para isso, foi implementada uma técnica de *Mechine Learning* com fatoração de matrizes.

Além disso, foram utilizadas as bibliotecas *NumPy* para operações básicas com matrizes e vetores, *time* para saber o tempo utilizado para realização de determinados processos, *pdb* para o debugger e *cProfile* para uma análise mais completa do tempo de todos os processos para facilitar a otimização. Houve também o uso de *linters* e *fixers* para uma melhor formatação do código.

Foram escritos 3 programas no total, *EP-T1.py*, *EP-T2.py* e *IdentificadorDeDígitos.py*, que equivalem a primeira, segunda e a tarefa principal. Todas os tempos e testes descritos foram feitos em um laptop com processador INTEL i5 de quarta geração, resultados podem variar.

2 Programas Auxiliares

2.1 Primeira Tarefa

Na primeira tarefa, foi necessário escrever um programa que inicialmente resolvia um sistema no estilo $Wx = b$ com W e b dados e o último sendo um vetor. Para isso, utilizamos o método **Rot-Givens**, que faz rotações nas matrizes a fim de transformar W em triangular superior para uma resolução mais eficiente.

O algorítimo em si, tem origens no arquivo *vector_ oper.py* disponibilizado pelo professor, e o método escolhido por nós, mostrado abaixo, era o mais eficiente em nossas máquinas.

```
def Rotgivens5(W, n, m, i, j, c, s):
    W[i, 0:m], W[j, 0:m] = (
    c * W[i, 0:m] - s * W[j, 0:m],
    s * W[i, 0:m] + c * W[j, 0:m])
    return W
```

O resto do código foi escrito de maneira bem compartimentalizada, há funções que aplicam o método de **Rot-Givens** em W e b, uma segunda que calcula as constantes para a primeira e outra responsável por encontrar x a partir dos vetores transformados.

Aplicando esses código para o exemplo **a**, encontramos x rapidamente, aproximadamente 0,019s, e concluímos que esse é um sistema determinado, pois Wx calculado com W original e x encontrado é igual ao vetor b original Já o exemplo **b**, realizado em menos de 0,015s, nunca resultava no vetor b após a multiplicação Wx, nos levando a concluir que esse era um sistema sobredeterminado.

Após esses exemplos e segundo a lógica do exercício, concluímos que uma alteração no nosso código original podia ser feito para que ele suportasse não só o b em forma de vetor mas também na de uma matriz com n colunas, ou seja, n sistemas simultâneos, assim os exemplos **c** e **d** poderiam ser testados.

Os resultados para ambos foi como o esperado, com tempos menores que 0.018s. Concluímos também que todos os sistemas de **c** são determinados, enquanto os de **d** não.

Assim, poderíamos utilizar o código para a construção da segunda tarefa auxiliar.

2.2 Segunda Tarefa

Nessa segunda tarefa o objetivo era fatorar uma matriz A em W e H positivos, para isso, foi utilizado o algorítimo dos mínimos quadrados. A solução de equações a partir de **Rot-Givens** foi utilizado para uma melhor otimização.

O método foi escrito se baseando no pseudo código disponibilizado, onde é necessário a resolução de 2 sistemas, um para a determinação de H e outro para a determinação de W. Ele se repetia até que o quadrado da diferença entre o erro da fatoração (WH) atual e anterior fosse mínimo se comparado a A original, e isso geralmente ocorria para erros absolutos pequenos.

Não houve muitos problemas nessa aplicação e os testes com as matrizes disponibilizadas ocorriam em menos de 0.01s. Acreditamos que esse tempo era bem menor do que os vistos na primeira tarefa, mesmo ele sendo utilizado nesse código, pois a matriz do exemplo é significantemente menor.

3 Programa Principal

Nesse programa, implementamos cálculos de matrizes para a criação de uma AI a partir de Machine Learning que é capaz de identificar dígitos escritos a mão a partir de uma imagem quadrada de 28 pixeis. Para isso, é preciso que um banco de dados com vários exemplos dos dignos manuscritos já categorizados sejam utilizados pelo programa para a criação de uma matriz para cada digito que serão a passe de um espaço vetorial.

Ao compararmos uma matriz criada a partir da imagem analisada com cada uma das bases, podemos ver com qual ela mais se assemelha para identificarmos o dígito nela escrito.

3.1 Os passos

Todos os dados de um dígito para os treinamentos são colocados em uma matriz T onde cada coluna representa uma imagem com $28^2 = 784$ colunas. Essas matrizes passam pelo processamento da imagem, normalizando seus valores.

A partir disso, podemos calcular a matriz de classificação a partir do algorítimo criado na segunda tarefa, nele a matriz T é fatorada em H, que e descartada, e W funcionará como uma base do espaço vetorial de tamanho 784 por p que define o seu nível de precisão, classificando os dígitos manuscritos.

Em seguida, a matriz A guardará os 10000 dígitos que serão classificado e encontraremos x_i nas equações $A = W_i x_i$ onde i é o dígito a der analisado. Depois, se compararmos a norma da coluna k entre todas as matrizes x_i, podemos concluir o dígito manuscrito daquela coluna provavelmente é i onde a norma de k da matriz W_i é menor.

3.2 Otimização

Para otimizar esse código, alguns passos foram tomados, e entre elas, ouve a condensação das funções. Se comparado com funções similares nos programas auxiliares vemos que elas são mais segmentadas para uma maior flexibilidade do código, entretanto, no programa *IdentificadorDeDígitos.py* isso era prejudicial, pois o tempo perdido chamando funções auxiliares se somavam muito rapidamente.

3.3 Testes

Foram utilizados os parâmetros precisão de W (p) e número de imagens para treino (n) para a realização dos testes. A variável p poderia ter os valores 5, 10 e 15, e n poderia ser 100, 1000 e 4000 e os resultados podem ser vistos na tabela abaixo.

A partir dos dados das tabelas, podemos tirar algumas conclusões, entre elas uma da mais importante é que apos certo número de dígitos de trinamento o aumento da taxa de

	100	1000	4000
5	141s (2.3min)	471s (7.8min)	690s (11.5min)
10	304s (5min)	679s (11.3min)	1955s (32.5min)
15	427s (7.1min)	939s (15.6min)	2793s (46.5min)

Tabela 1: Tempo na realização do programa *IdentificadorDeDígitos.py*

Dígito	100	1000	4000
0	96.63265306122449%	97.55102040816327%	98.87755102040816%
1	99.47136563876651%	99.20704845814979%	99.38325991189427%
2	84.59302325581395%	87.79069767441861%	91.76356589147287%
3	85.64356435643565%	91.38613861386139%	91.2871287128713%
4	83.70672097759673%	86.65987780040734%	92.15885947046843%
5	83.85650224215246%	88.00448430493275%	88.00448430493275%
6	94.1544885177453%	96.24217118997912%	96.97286012526096%
7	88.13229571984435%	90.1750972762646%	93.28793774319067%
8	78.74743326488706%	86.13963039014374%	90.96509240246407%
9	86.32309217046581%	87.11595639246778%	92.07135777998018%
Total	88.5%	91.4%	91.67%

Tabela 2: Porcentagem de acerto para $p = 5$

Dígito	100	1000	4000
0	97.6530612244898%	98.57142857142858%	98.57142857142858%
1	99.38325991189427%	99.64757709251101%	99.55947136563876%
2	90.01937984496125%	90.50387596899225%	92.44186046511628%
3	86.03960396039604%	91.68316831683168%	93.06930693069307%
4	87.57637474541752%	92.4643584521385%	93.78818737270875%
5	86.88340807174887%	88.2286995515695%	90.02242152466367%
6	95.4070981210856%	96.4509394572025%	96.5553235908142%
7	89.6887159533074%	91.73151750972762%	92.50972762645915%
8	79.36344969199179%	87.47433264887063%	86.24229979466119%
9	88.10703666997026%	91.2784935579782%	91.67492566897918%
Total	90.16%	92.92%	93.55%

Tabela 3: Porcentagem de acerto com p = 10

Todos	100	1000	4000
0	97.24489795918367%	98.87755102040816%	98.67346938775509%
1	99.47136563876651%	99.38325991189427%	99.38325991189427%
2	88.85658914728683%	91.76356589147287%	93.02325581395348%
3	85.44554455445544%	91.2871287128713%	92.97029702970298%
4	88.59470468431772%	92.15885947046843%	93.58452138492872%
5	84.19282511210763%	88.00448430493275%	89.68609865470853%
6	95.61586638830897%	96.97286012526096%	96.86847599164928%
7	93.19066147859922%	93.28793774319067%	91.14785992217898%
8	80.39014373716633%	90.96509240246407%	89.01437371663245%
9	88.800792864222%	92.07135777998018%	91.87314172447968%
Total	90.36%	93.58%	93.72%

Tabela 4: Porcentagem de acerto para $p = 15$

sucesso é minimo se comparado com o tempo maior de execução. Entretanto, se valor de precisão de tende a aumentar, a precisão dos resultados aumenta em um maior grau.

4 Conclusões

A partir desse trabalho conseguimos desenvolver um conhecimento alto de *Python* e das técnicas de *Machine Learning*. Também percebemos que, as vezes, para conseguir melhores resultados em tempos razoáveis em programas desse tipo, a quantidade de dados para o treinamento é ofuscada pela quantidade de detalhamento desses dados.