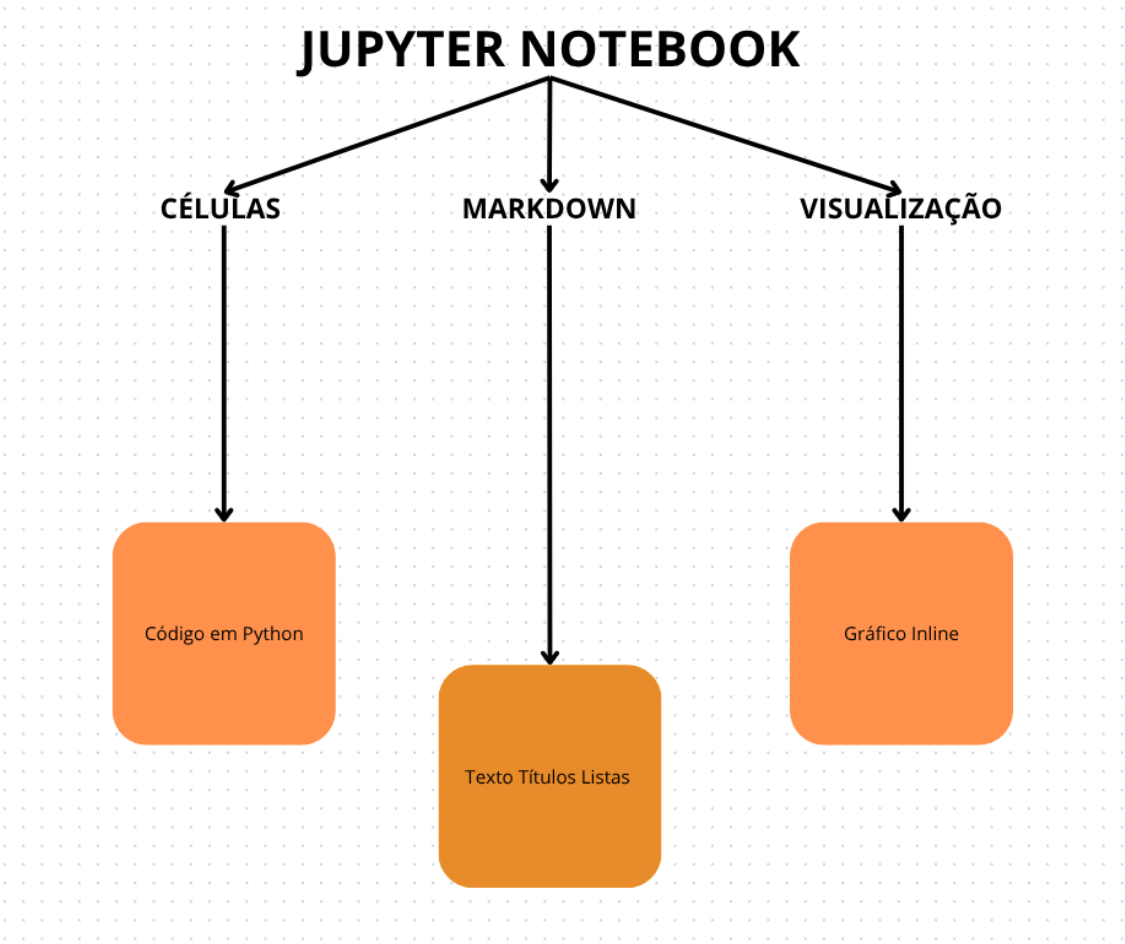
4 - Prática: Principais Bibliotecas e Ferramentas Python para Aprendizado de Máquina (I)

Ronny Gabryel Colatino de Souza

Descrição da atividade:

Jupyter Notebook

Logo de cara, o que mais me impressionou no Jupyter foi como ele mistura código com explicação. Diferente de escrever código num editor separado e depois ter que documentar tudo, aqui você vai escrevendo e explicando junto.



Células independentes: Posso rodar pedaços do código sem executar tudo. Isso é perfeito quando estou testando uma ideia ou debugando algo específico.

Gráficos na hora: Quando faço um plot, ele aparece ali mesmo, embaixo do código. Não preciso ficar abrindo janela separada.

Markdown integrado: Escrevo texto normal, títulos, listas, tudo misturado com código. Fica organizado e fácil de entender depois.

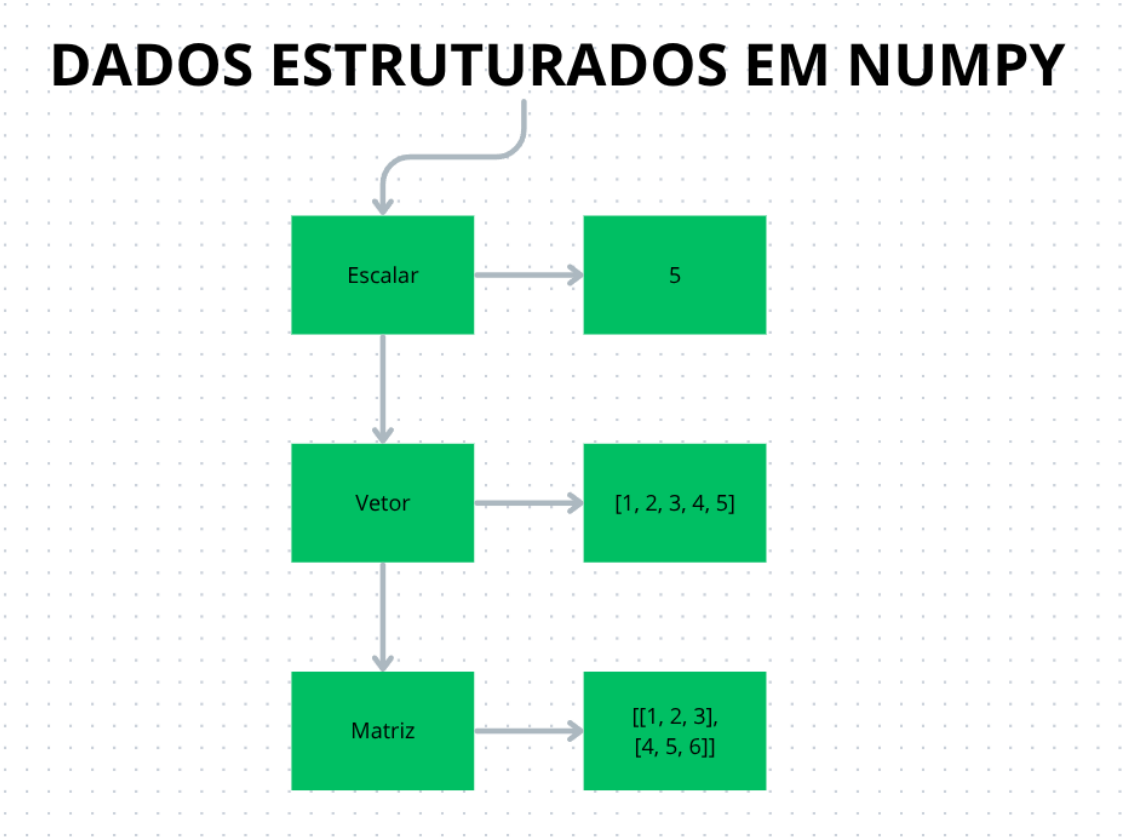
Comparando com Google Colab, o Jupyter roda local na minha máquina, o que significa controle total dos dados e não depender de internet. Mas o Colab tem a vantagem de já vir com tudo instalado e GPU gratuita.

Senti que o Jupyter transforma programação em algo mais parecido com escrever um relatório científico. Você vai contando a história da sua análise enquanto mostra o código e os resultados.

NumPy

NumPy foi uma revelação. Vindo de outras linguagens, ver como Python consegue ser rápido para matemática me surpreendeu.

Minha Visualização da Hierarquia NumPy:



Arrays vs Listas Python

O grande diferencial está na performance. Uma lista Python normal é lenta para operações matemáticas porque cada elemento pode ser qualquer coisa. NumPy força todos os elementos a serem do mesmo tipo, permitindo otimizações.

Exemplo prático:

```python
# Lista Python normal - lenta
lista = [1, 2, 3, 4, 5]
nova_lista = [x * 2 for x in lista]  # Precisa de loop
```

```python
# Array NumPy - rápido
array = np.array([1, 2, 3, 4, 5])
novo_array = array * 2  # Operação direta, sem loop
```

NumPy trabalha com arrays de qualquer dimensão:

- **1D**: Como uma lista simples [1, 2, 3]
- **2D**: Como uma tabela com linhas e colunas
- **3D**: Para dados mais complexos como imagens coloridas

Criação rápida que usei:

```python
np.zeros(5) # Array de zeros: [0. 0. 0. 0. 0.]
```

```python
np.ones(3) # Array de uns: [1. 1. 1.]
```

```python
np.arange(0, 10) # Sequência: [0 1 2 3 4 5 6 7 8 9]
```

Arrays 2D:

```python
np.zeros((3, 4)) # Matriz 3x4 de zeros
```

```python
np.eye(3) # Matriz identidade 3x3
```

Arrays aleatórios:

```
np.random.rand(3)     # 3 números aleatórios entre 0 e 1
```

```
np.random.randint(0, 10, 5)  # 5 números inteiros entre 0 e 9
```

Comparação Performance (Minha Experiência):

Operação	Lista Python	NumPy Array	Velocidade
Soma 1M elementos	100ms	2ms	50x
Multiplicação	Loop manual	Operação direta	100x
Estatísticas	Cálculo manual	Integrado	80x

Indexação e Seleção:

```
arr = np.arange(0, 10)
```

```
arr[2] # Elemento na posição 2
```

```
np.int64(2)
```

```
arr[2:5] # Elementos da posição 2 até 4
```

```
array([2, 3, 4])
```

```
arr[arr > 5]  # Todos elementos maiores que 5
```

```
array([6, 7, 8, 9])
```

Seleção inteligente que me impressionou:

```
dados = np.array([1, 5, 3, 8, 2, 9])
```

```
# Pegar só os números pares
```

```
pares = dados[dados % 2 == 0]  # [8, 2]
```

Operações Matemáticas:

```
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])
```

```
a + b # [5, 7, 9]
```

```
array([5, 7, 9])
```

```
a * b    # [4, 10, 18]
```

```
array([ 4, 10, 18])
```

```
a ** 2   # [1, 4, 9]
```

```
array([1, 4, 9])
```

```
# Com escalares
a * 10   # [10, 20, 30]
```

```
array([10, 20, 30])
```

Funções Estatísticas

```
dados = np.random.rand(100)
```

```
dados.mean() # Média
```

```
np.float64(0.4856770154970958)
```

```
dados.std() # Desvio padrão
```

```
np.float64(0.2770212714729233)
```

```
dados.max() # Máximo
```

```
np.float64(0.9873127505313937)
```

```
dados.min() # Mínimo
```

```
np.float64(0.013241011086396526)
```

```
dados.sum() # Soma total
```

```
np.float64(48.56770154970958)
```

O que mais me impressiona é como NumPy elimina a necessidade de loops manuais. Operações que em C precisariam de for são feitas em uma linha.

Pandas:

Se NumPy é bom para arrays, Pandas é perfeito para dados do mundo real: tabelas, planilhas, CSVs, essas coisas.

Minha Representação Visual da Estrutura Pandas:

Series 1D

Index	Data
0	10
1	20
2	30
3	40

Dataframe 2D

	Nome	Idade	Cidade
0	Ana	25	SP
1	João	30	RJ
2	Maria	35	MG

Fluxograma das Operações Pandas que Mais foi utilizado:

Series: A Base do Pandas

Series é como uma coluna de dados com índice:

```
# Criando Series
```

```
vendas = pd.Series([100, 200, 300], index=['Jan', 'Fev', 'Mar'])
```

```
print(vendas['Jan'])  # 100
```

```
100
```

```
# Operações com Series
```

```
vendas * 1.1  # Aumentar 10%
```

```
Jan    110.0
Fev    220.0
Mar    330.0
dtype: float64
```

```
vendas[vendas > 150]  # Filtrar valores
```

```
Fev    200
Mar    300
dtype: int64
```

DataFrame: A Estrutura Principal

DataFrame é como uma planilha Excel, mas que você controla por código. Tem linhas, colunas, e cada coluna pode ter um tipo de dado diferente.

```
# Criando DataFrame
```

```
dados = {
    'Nome': ['Ana', 'João', 'Maria'],
    'Idade': [25, 30, 28],
    'Salário': [5000, 6000, 5500]
}
```

```
df = pd.DataFrame(dados)
```

Seleção e Filtros:

```
# Selecionar colunas
df['nome']
```

```
0    Alice
1    Bruno
2    Carla
Name: nome, dtype: object
```

```
df[['nome', 'idade']] # Múltiplas colunas
```

	nome	idade
0	Alice	25.0
1	Bruno	NaN
2	Carla	30.0

```
# Selecionar linhas
df.loc[0] # Primeira linha
```

```
nome     Alice
idade     25.0
Name: 0, dtype: object
```

```
df.iloc[0:2] # Primeiras duas linhas
```

	nome	idade
0	Alice	25.0
1	Bruno	NaN

```
# Filtros condicionais
df[df['idade'] > 25]  # Pessoas com mais de 25 anos
```

	nome	idade	Salario
1	João	30	6000
2	Maria	28	5500

```
df[df['Salario'] > 5000]  # Com acento
```

	nome	idade	Salario
1	João	30	6000
2	Maria	28	5500

Estatísticas Descritivas:

```
df.describe() # Estatísticas gerais
```

	idade	Salario
count	3.000000	3.0
mean	27.666667	5500.0
std	2.516611	500.0
min	25.000000	5000.0
25%	26.500000	5250.0
50%	28.000000	5500.0
75%	29.000000	5750.0
max	30.000000	6000.0

```
df['Salario'].mean() # Média de salários
```

```
np.float64(5500.0)
```

```
df.info() # Informações sobre os dados
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3 entries, 0 to 2
Data columns (total 3 columns):
 #   Column   Non-Null Count  Dtype
---  ------   --------------  -----
 0   nome     3 non-null      object
 1   idade    3 non-null      int64
 2   Salario  3 non-null      int64
dtypes: int64(2), object(1)
memory usage: 204.0+ bytes
```

```
df.head()           # Primeiras 5 linhas
```

	nome	idade	Salario
0	Ana	25	5000
1	João	30	6000
2	Maria	28	5500

GroupBy: Agrupamento de Dados:

Uma das coisas mais úteis é agrupar dados por categoria:

```
# Dados de vendas por região
vendas = pd.DataFrame({
    'Região': ['Norte', 'Sul', 'Norte', 'Sul'],
    'Vendas': [100, 200, 150, 300],
    'Produto': ['A', 'B', 'A', 'B']
})
```

```
# Agrupar por região
vendas.groupby('Região')['Vendas'].sum()
```

```
Região
Norte    250
Sul      500
Name: Vendas, dtype: int64
```

Dados Ausentes (NaN)

Pandas lida bem com dados que estão faltando:

```
# Dados com valores faltando
df_com_nan = pd.DataFrame({
    'A': [1, 2, np.nan],
    'B': [5, np.nan, np.nan],
    'C': [1, 2, 3]
})
```

```
# Remover linhas com NaN
df_com_nan.dropna()
```

	A	B	C
0	1.0	5.0	1

```
# Preencher NaN com um valor
df_com_nan.fillna(0)
```

	A	B	C
0	1.0	5.0	1
1	2.0	0.0	2
2	0.0	0.0	3

```
# Preencher com a média
df_com_nan['A'].fillna(df_com_nan['A'].mean())
```

```
0    1.0
1    2.0
2    1.5
Name: A, dtype: float64
```

Leitura e Escrita de Arquivos:

```
# Ler arquivos
df = pd.read_csv('dados.csv')
```

```
df = pd.read_excel('dados.xlsx')
```

```
# Salvar arquivos
df.to_csv('resultado.csv', index=False)
```

```
df.to_excel('resultado.xlsx', index=False)
```

Achei o Pandas intuitivo porque ele pensa como a gente pensa sobre dados. Quando você quer "mostrar só as vendas de São Paulo", a sintaxe fica bem parecida com isso.

Integração das Ferramentas:

Na prática, essas três ferramentas trabalham juntas:

Fluxo Típico de Trabalho

Jupyter: Ambiente para documentar todo o processo
Pandas: Carregar e limpar os dados
NumPy: Operações matemáticas mais pesadas
Pandas: Análise e visualização dos resultados

Conclusões

Essas três ferramentas revolucionaram minha forma de trabalhar com dados. O Jupyter Notebook não é apenas um editor, é uma ferramenta de storytelling para análise de dados. Posso contar uma história completa, desde a hipótese inicial até os resultados finais, tudo em um lugar só.

NumPy me mostrou que Python pode ser rápido para matemática quando usado corretamente. A sintaxe simples esconde uma engine poderosa que aproveita otimizações de baixo nível. Arrays NumPy são a base para praticamente todas as outras bibliotecas de ciência de dados.

Pandas é como ter Excel com superpoderes. A facilidade para filtrar, agrupar e transformar dados é impressionante. O que mais me chamou atenção foi como operações complexas de dados se tornam intuitivas com a sintaxe do Pandas.

A integração entre essas ferramentas é natural. NumPy fornece a base matemática, Pandas constrói estruturas mais complexas em cima dele, e Jupyter documenta todo o processo. É um ecossistema que faz sentido e prepara o terreno para ferramentas mais avançadas de machine learning.

Para mim que e alguém que entende bem pouco de ciencia de dados, essas tres bibliotecas vejo que são essenciais e me ajudaram a pensar de uma forma diferente sobre os dados

Referencias

A pasta com os links do curso