Curso: Machine Learning Básico com Python

Aula 02 – Bibliotecas PYTHON para Inteligência Artificial

Roteiro 04 – Bibliotecas para IA

2. Pandas

O Pandas é o pacote para a manipulação dos dados de nossa investigação usando Inteligência Artificial em Python. É uma biblioteca Python que fornece uma série de recursos de leitura e manipulação de grandes bases de dados com uma grande variedade de comandos.

De início aprenderemos com abrir bases de dados (datasets) armazenados em Excel (.xlsx) e CSV (.csv) e depois criaremos nosso próprio arquivo de dados como um DataFrame.

Assim como para usar o NumPy precisamos importar ele, aqui também teremos que importar o Pandas. A maneira de fazê-lo é tão tranquila quanto o NumPy aqui no Colab, já que ambos estão pré-instalados:

```
import pandas as pd
```

Assim como demos o apelido de np para o NumPy, demos o apelido de pd para o Pandas. Isso facilita chamar ele toda hora.

A primeira importação de dados que faremos será uma planilha Excel. Para isso vá ao Excel e cria uma planilha como a da figura 1 dando o nome nela de “dados.xlsx”, extensão padrão do Excel hoje.

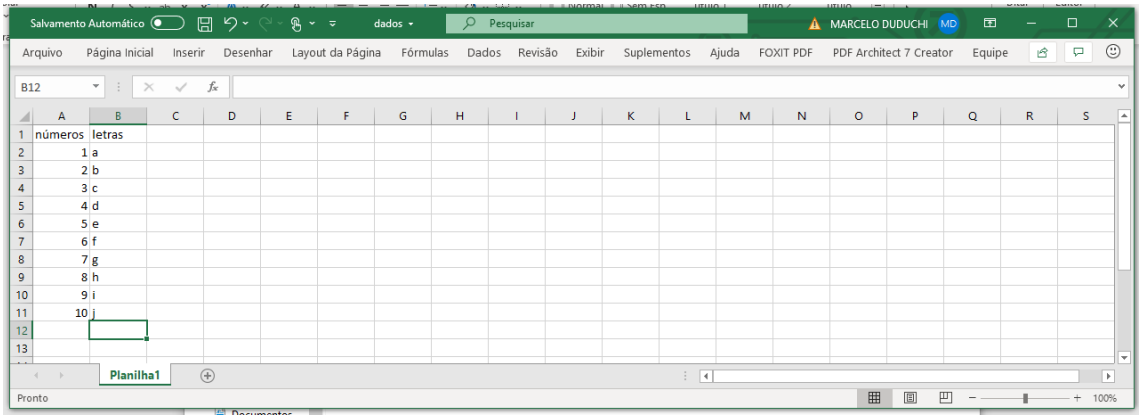

Figura 1. Planilha Excel básica

Agora que temos nossa planilha com dados criada vamos carregá-la no Colab para depois podermos usada. Esse processo é um pouco diferente que em outros ambientes. As instruções para isso são:

```
from google.colab import files
uploaded = files.upload()
```

Assim que você mandar executar as instruções ele solicitará que você faça a seleção do ou dos arquivos a fazer o upload. No caso você deve selecionar o arquivo criado no excel “dados.xlsx”. Veja na figura 2. O dicionário “uploaded” terá as chaves dos nomes de arquivos selecionados -

portanto, se, por exemplo, você selecionar um arquivo “dados.xlsx”, então você acessaria esse arquivo usando “uploaded['dados.xlsx']”.

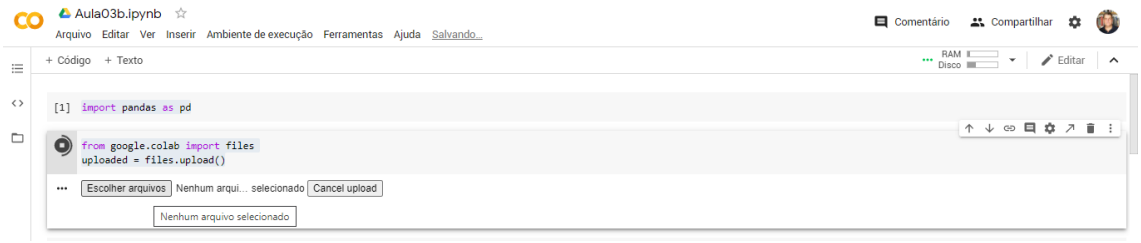

Figura 2. Seleção dos arquivos para upload.

Se tudo der certo será apresentado o processo de carregamento até os 100% e aparecerá a informação que o arquivo foi carregado conforme figura 3.

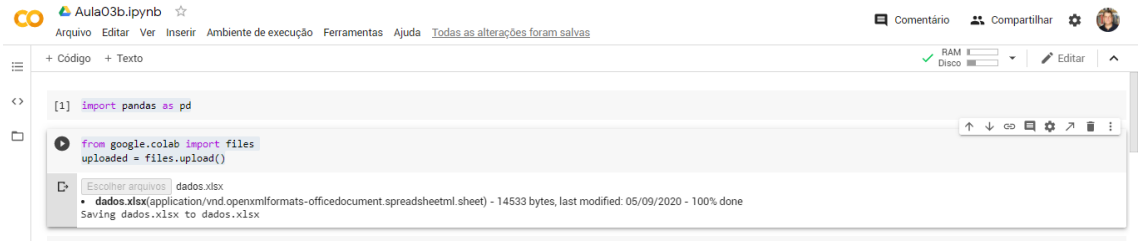

Figura 3. Tela do carregamento do arquivo.

Cuidado! Não crie nome de arquivos com espaço em branco. Eles podem dar problema ao carregar ou na hora de indicar a leitura. Além disso sempre verifique o nome assumido ao salvar o arquivo. Se o nome do arquivo já existia outro nome automaticamente é dado com o sufixo “ (n)” dependendo de quantos arquivos com o mesmo nome já existem.

Assim como usamos o “upload” podemos usar o “download” para baixar os arquivos do Colab no nosso computador:

```
files.download("dados.xlsx")
```

A partir do momento que os arquivos estão no Colab alguns comandos simples e úteis são “!ls” (mostra os arquivos armazenados) “!rm” <nome do arquivo> (remove o arquivo).

Além disso você pode clicar no símbolo de pasta à esquerda da tela (seta vermelha em destaque na figura 4) que apresentará os arquivos por onde também podem gerenciar os arquivos de dados.

A Figura 4 mostra também o comando “!ls” executado mostrando os arquivos existentes e a aba de arquivos aberta à esquerda.

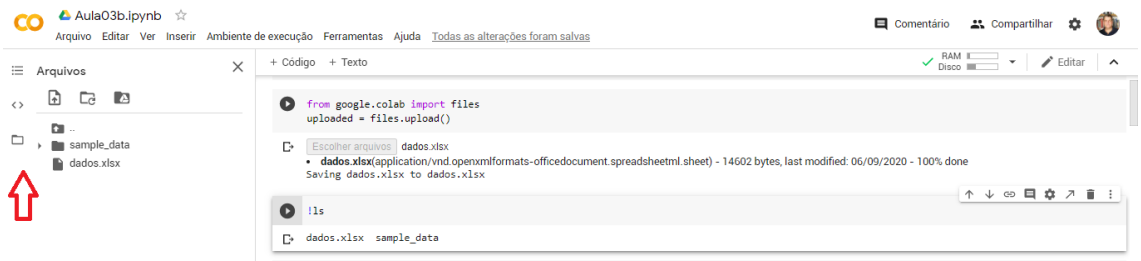

Figura 4. Tela do Colab mostrando a aba de arquivos a esquerda e a execução de “!ls”

É possível também saber quais os arquivos que carregamos e referenciados no dicionário “uploaded usando as instruções a seguir:

```
for fn in uploaded.keys():
  print('Arquivo "{name}" com tamanho de {length} bytes'.format(
      name=fn, length=len(uploaded[fn])))
```

```
Arquivo "dados.xlsx" com tamanho de 14602 bytes
```

Depois de carregar o arquivo no Colab (em outros ambientes este passo pode ser suprimido) precisamos de preparar o arquivo para leitura. A instrução a seguir é responsável por isso:

```
dados1 = pd.read_excel('dados.xlsx')
```

Existem diversos outros possíveis parâmetros a serem passados na função “read_excel”, mas por enquanto não vamos utilizá-los. Se você quiser pode consultar estes na documentação do Pandas disponível em:

https://pandas.pydata.org/

Caso queira consulta-los, a documentação em pdf da última versão 1.1.1 de 20 de agosto de 2020 encontra-se disponível em:

https://pandas.pydata.org/docs/pandas.pdf

Sempre que abrimos um novo arquivo duas funções são as mais usadas para conhecer um pouco sobre o arquivo aberto. O “shape” que mostra uma tupla com o número de linhas e colunas do arquivo e “head” que mostra os primeiros registros do dataset.

Veja o uso do “shape”:

```
dados1.shape
```

```
(10, 2)
```

Veja o uso do “head”:

```
dados1.head()
```

---

	números	letras
0	1	a
1	2	b
2	3	c
3	4	d
4	5	e

O padrão do “head” é mostrar os primeiros 5 registros, mas se você passar a quantidade de registros ele apresentará a quantidade solicitada:

```
dados1.head(8)
```

---

	números	letras
0	1	a
1	2	b
2	3	c
3	4	d
4	5	e
5	6	f
6	7	g
7	8	h

Esses comandos iniciais nos permitem ter uma ideia da base de dados que abrimos e por isso vale a pena executá-los.

A seguir carregaremos um arquivo CSV. Para isso selecionamos a base que usaremos em nosso projeto. Ela é uma base aberta disponível para download e deixamos disponível no Google Classroom. Um lugar interessante para pegarmos bases de dados abertas para fazermos testes é o Kaggle (pronuncia-se “kégol”):

https://www.kaggle.com/

Este site possui diversos *datasets* que podem ser usados para os exercícios que vamos propor. Vale a pena dar uma olhada. Para usar basta fazer uma inscrição simples usando seu próprio usuário Google. A forma de se inscrever aparecerá logo no primeiro acesso do lado inferior esquerdo da tela inicial.

A figura 5 mostra a tela inicial do Kaggle Já logada. O acesso aos *datasets* você tem clicando na opção *data* em destaque (seta vermelha). Lá você pode vagar pelos *datasets*. Clicando em Quick Look você tem um resumo sobre o *dataset,* alguns dados sobre ele como o tamanho, as colunas disponíveis, os dados e se quiser ainda pode baixar em “download”.

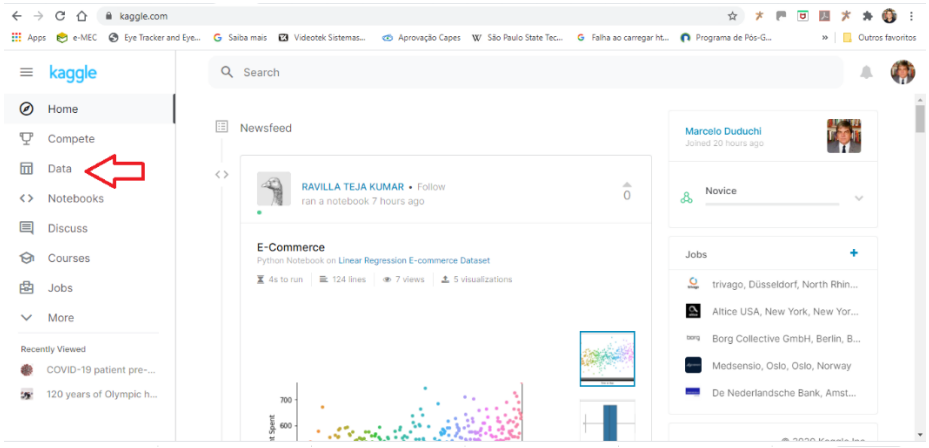

Figura 5. Tela Inicial do Kaggle.

Para carregar a base CSV seguimos os mesmos passos que com a base Excel apresentados anteriormente. Se você ainda não iniciou o Pandas e o "files" não pode esquecer de fazê-lo. No caso de estar começando a usar agora deveria usar todas as funções que usamos antes...

Em seguida carregar o arquivo com:

```
uploaded.update(files.upload())
```

O "update" fica por conta de o "files.upload()" retornar um dicionário e para fundir os dois dicionários em "uploaded" usamos o "update". Se não quisermos este controle em uploaded basta simplesmente carregarmos o arquivo com:

```
files.upload()
```

Com o arquivo carregado basta usarmos uma função bem parecida com a usada para arquivos Excel agora para o arquivo CSV:

```
dados2 = pd.read_csv('basecensus.csv')
```

Para conhecer melhor o arquivo, como anteriormente usamos o "shape" e o "head":

```
dados2.shape
```

```
(30162, 16)
```

Como é possível perceber tem 30.162 linhas e 16 colunas.

```
dados2.head()
```

	Unnamed: 0	age	workclass	final.weight	education	education.num	marital.status	occupation	relationship	race	sex	capital.gain	capital.loos	hour.per.week	native.country	income
0	1	39	State-gov	77516	Bachelors	13	Never-married	Adm-clerical	Not-in-family	White	Male	2174	0	40	United-States	<=50K
1	2	50	Self-emp-not-inc	83311	Bachelors	13	Married-civ-spouse	Exec-managerial	Husband	White	Male	0	0	13	United-States	<=50K
2	3	38	Private	215646	HS-grad	9	Divorced	Handlers-cleaners	Not-in-family	White	Male	0	0	40	United-States	<=50K
3	4	53	Private	234721	11th	7	Married-civ-spouse	Handlers-cleaners	Husband	Black	Male	0	0	40	United-States	<=50K
4	5	28	Private	338409	Bachelors	13	Married-civ-spouse	Prof-specialty	Wife	Black	Female	0	0	40	Cuba	<=50K

E aí estão as 16 colunas com os 5 primeiras linhas. Assim podemos ter uma ideia do *dataset* alvo do nosso projeto. A partir daí é possível excluir colunas que não nos interessam, repartir, fatiar para realizarmos tudo o que pretendemos com o *dataset*.

A terceira forma de trabalhar com *datasets* é criarmos ele a partir de um dicionário. O dicionário deve ter a característica de apresentar-se com uma chave e uma lista. Veja o exemplo de um dicionário de alunos:

```
alunos = {'Nome':['Marcelo','Lucas','Cynthia','Helena'],
          'Media':[4,7,5.5,9],
          'Status':['Reprovado','Aprovado','Reprovado','Aprovado']}
```

Para transformar este dicionário em um dataframe é a função "DataFrame". Veja:

```
df1 = pd.DataFrame(alunos)
```

A partir de agora podemos usar "alunos" como os outros *datasets*:

```
df1.shape
```

```
(4, 3)
```

```
df1.head()
```

	Nome	Media	Status
0	Marcelo	4.0	Reprovado
1	Lucas	7.0	Aprovado
2	Cynthia	5.5	Reprovado
3	Helena	9.0	Aprovado

Ou ainda:

```
print(df1)
```

	Nome	Media	Status
0	Marcelo	4.0	Reprovado
1	Lucas	7.0	Aprovado
2	Cynthia	5.5	Reprovado
3	Helena	9.0	Aprovado

Além do “shape” e “head()” existem algumas outras funções interessantes para explorarmos nossas bases de dados. A primeira delas é a “describe()” que trás de uma vez uma série de informações importantes (quantidade, média, valor máximo, valor mínimo, desvio padrão e percentis) sobre as colunas numéricas do *dataframe*:

```
df1.describe()
```

	Media
count	4.000000
mean	6.375000
std	2.136001
min	4.000000
25%	5.125000
50%	6.250000
75%	7.500000
max	9.000000

Para filtrar colunas específicas basta indicar o nome do *dataframe* com a coluna entre parênteses:

```
df1[['Nome']]
```

	Nome
0	Marcelo
1	Lucas
2	Cynthia
3	Helena

Para filtrar linhas podemos usar o “loc” indicando entre colchetes o, ou os, índices de interesse:

```
df1.loc[[1,3]]
```

	Nome	Media	Status
1	Lucas	7.0	Aprovado
3	Helena	9.0	Aprovado

Ou ainda:

```
df1.loc[1:3]
```

	Nome	Media	Status
1	Lucas	7.0	Aprovado
2	Cynthia	5.5	Reprovado
3	Helena	9.0	Aprovado

Podemos ainda estabelecer condições no “loc”. Veja os exemplos:

```
df1.loc[df1['Status'] == 'Aprovado']
```

	Nome	Media	Status
1	Lucas	7.0	Aprovado
3	Helena	9.0	Aprovado

```
df1.loc[df1['Media'] < 7.0]
```

	Nome	Media	Status
0	Marcelo	4.0	Reprovado
2	Cynthia	5.5	Reprovado

Além de filtrar é possível também construir novos *dataframes* a partir de condições específicas. Para isso basta atribuir a um novo *dataframe*. Veja:

```
exame = df1.loc[df1['Media'] < 7.0]
```

```
print(exame)
```

	Nome	Media	Status
0	Marcelo	4.0	Reprovado
2	Cynthia	5.5	Reprovado

Ou ainda selecionando colunas:

```
exame = df1[['Nome','Media']].loc[df1['Media'] < 7.0]
print(exame)
```

```
      Nome  Media
0  Marcelo    4.0
2  Cynthia    5.5
```

Muitas vezes ao manipularmos os dados para realizar análises separamos colunas específicas. Além dessas existem outras oportunidades onde queremos ter esta coluna atrelada a um índice numérico como os *dataframes*. O Pandas tem um tipo pronto para isso que são as Séries. Podemos criar séries a partir de listas de Python ou até de arrays de NumPy. Observe os exemplos:

```
s1 = pd.Series([2,5,3,34,54,23,1,16])
print(s1)
```

```
0     2
1     5
2     3
3    34
4    54
5    23
6     1
7    16
dtype: int64
```

```
import numpy as np
array1 = np.array([2,5,3,34,54,23,1,16])
s2 = pd.Series(array1)
print(s2)
```

```
0     2
1     5
2     3
3    34
4    54
5    23
6     1
7    16
dtype: int64
```

A série na verdade tem comandos muito próximos do *dataframe*. É como se fosse um *dataframe* de uma coluna só.

Note que se fizermos determinados comandos de manipulação de fatiamento de dataframes obtemos séries:

```
s3 = df1['Nome']
print(s3)
type(s3)
```

```
0    Marcelo
1      Lucas
2    Cynthia
3     Helena
Name: Nome, dtype: object
pandas.core.series.Series
```

Note que "`df1[['Nome']]`" != "`df1['Nome']`"

Por fim consideraremos algumas manipulações com colunas de dataframes.

Para renomear uma coluna por exemplo basta usar o "rename". Você indica entre parênteses os nomes atuais das colunas e após ":" o novo nome:

```
dados1.rename(columns={"números":"nr","letras":"ltr"})
```

```
Para voltarmos aos nomes originais podemos renomear novamente,
agora usando o inplace para não mostrar o data frame após alterar:
```

```
dados1.rename(columns={"nr":"números","ltr":"letras"},inplace=True)
```

Você também pode checar quantos valores tem de cada ocorrência em cada coluna. No nosso caso não tem muito sentido porque só colocamos uma ocorrência de cada valor:

```
dados1['letras'].value_counts()
```

```
j    1
g    1
e    1
i    1
h    1
a    1
f    1
c    1
d    1
b    1
Name: letras, dtype: int64
```

Só note um detalhe, em algumas bases vão aparecer valores como NaN que vão significar valor não informado e podem inclusive ser alvo de nossas tarefas em IA no futuro. Essa função é muito boa para vasculhar um *dataset* como o do projeto de vocês.

Por último veremos como excluir uma coluna que pode não ser de nosso interesse com o "drop". Temos que indicar qual a coluna alvo (no caso a "números"), que nosso alvo é uma coluna ("axis=1") e que não queremos mostrar os dados ("Inplace=True"):

```
dados1.drop('números',axis=1,inplace=True)
print(dados1)
```

```
  letras
0      a
1      b
2      c
3      d
4      e
5      f
6      g
7      h
8      i
9      j
```

Não é o caso desta base, mas poderíamos retirar as linhas que tem dados faltantes com para realizar alguma operação que pudesse depois trabalhar estes dados para conseguir completar os dados NaN. Para fazer isso poderíamos usar:

```
teste = dados1.dropna()
```

No nosso caso não será excluído nada, mas teste com dataframe com dados ausentes.

Ao invés de excluí-los poderíamos verificar quais estão faltantes com a função "isnull":

```
nulos = dados1.isnull()
print(nulos)
```

```
   letras
0   False
1   False
2   False
3   False
4   False
5   False
6   False
7   False
8   False
9   False
```

Usando o conceito do "isnull" podemos trabalhar a quantidade de dados faltantes:

```
qtdnulos = dados1.isnull().sum()
print(qtdnulos)
```

```
letras    0
dtype: int64
```

Ou ainda verificar o percentual disso considerando o total de linhas que pode ser verificado a partir de uma coluna qualquer (no caso só temos a coluna letras) pela função “len(“:

```
len(dados1['letras'])
```

É possível também preencher os dados faltantes com algum valor. Por exemplo:

```
dados1['letras'].fillna('semdados',inplace = True)
```

Exercícios:

Teste estas manipulações com dataframes na base de dados “basecensus.csv”