



**Treinamento Python Aplicado ao Big Data – Python Aplicado a Big Data - Analisando Dados com Numpy e Pandas**

Conteudo Teorico/Prático

## Sumário

Introdução ao Matplotlib .....	3
Tipos de plotagem .....	3
Seaborn Data Visualization .....	18
Heat Map (Visualização com mapa de calor) .....	18
Criando um mapa de calor .....	18
Mapa de cores sequencial (Sequential colormaps) .....	22
Seaborn heatmap colorbar .....	25
O que é NumPy? .....	30
NumPy Matriz(Array) .....	31
Operações de Matrizes .....	34
Multiplicação de duas matrizes .....	35
Acessando elementos de uma matriz: colunas e linhas .....	37
Fatiamento (slicing) de uma matriz np .....	40
Pandas .....	42
Explorando Dados com Dataframes .....	42
Ler arquivo Excel .....	42
Importar arquivo CSV .....	43
Ler arquivo de texto .....	44
Aplicar uma função a colunas / linhas .....	47
Classificar valores / classificar por coluna .....	48
Eliminar / remover duplicados .....	50
Excluir uma coluna .....	52
Excluir linhas .....	53
Somar uma coluna .....	56
Contagem de valores únicos .....	56
Subconjunto de linhas .....	57
Gerar arquivo Excel .....	59
Gerar arquivo CSV .....	60
Gerar arquivo HTML .....	61
Pivot Table em Pandas .....	63
Acessar os dados .....	64
“Pivotando” os dados .....	65
Columns vs Values .....	69

Filtros avançados.....	76
GropuBy.....	77
Aprofundando Data Analysis.....	79
Análise da Distribuição .....	84
Análise de Variáveis Categóricas .....	87
Munging de dados com Python: Usando Pandas.....	91
Tratando grande volumes de dados .....	96
Lendo o arquivo.....	96
Utilizando o Pandas .....	97
Formatando o resultado.....	104
MatrizDe Correlação .....	106
Qual é o coeficiente de correlação?.....	107
Traçando a matriz de correlação.....	109
Interpretando a matriz de correlação .....	112
Adicionando título e rótulos ao gráfico.....	113
Classificando a matriz de correlação .....	115
Seleção de pares de correlação negativa .....	118
Introdução ao Machine Learning .....	124
Como iniciar um projeto de aprendizado de máquina em Python? .....	124
Referencias:.....	144

## Introdução ao Matplotlib

Matplotlib é uma biblioteca em Python que cria gráficos 2D para visualizar dados. A visualização sempre ajuda na melhor análise dos dados e aumenta a capacidade de tomada de decisão do usuário. Neste passo-a-passo matplotlib, vamos traçar alguns gráficos e alterar algumas propriedades como fontes, rótulos, intervalos, etc.,

Primeiro, vamos instalar o matplotlib; caso a distribuição Anaconda esteja sendo usada, não é necessária nenhuma instalação. Começaremos a traçar alguns gráficos. Vamos gerar alguns dos gráficos que matplotlib pode desenhar.

## Tipos de plotagem

Existem vários tipos de gráficos diferentes em matplotlib. Esta seção explica brevemente alguns tipos de plotagem em matplotlib.

### Gráfico de linha

Um gráfico de linha é uma linha 2D simples no gráfico.

### Contorno e Pseudocolor

Podemos representar um matrizbidimensional em cores usando a função `pcolormesh()` mesmo se as dimensões estiverem espaçadas de forma desigual. Da mesma forma, a função `contour()` faz o mesmo trabalho.

### Histogramas

Para retornar as contagens de bin e probabilidades na forma de um histograma, usamos a função `hist()`.

### Paths

Para adicionar um caminho arbitrário no Matplotlib, usamos o módulo ***matplotlib.path***.

## Streamplot

Podemos usar a função `streamplot()` para traçar as linhas de fluxo de um vetor. Também podemos mapear as cores e a largura dos diferentes parâmetros, como velocidade, tempo etc.

## Gráficos de barra

Podemos usar a função `bar()` para fazer gráficos de barras com muitas personalizações.

## Outros tipos

Alguns outros exemplos de gráficos em Matplotlib incluem:

- Elipses
- Gráfico de setores
- Tabelas
- Gráficos de dispersão (Scatter Plots)
- Widgets GUI
- Curvas preenchidas (Filled curves)
- Tratamento de data (Date handling)
- Log Plots
- Legendas
- TeX- Notações para objetos de texto
- Renderização TeX nativa
- EEG GUI
- Plotagens de esboço estilo XKCD (style sketch plots)

Para importar o pacote em seu projeto Python, use a seguinte instrução:

```
import matplotlib.pyplot as plt
```

**Matplotlib** é a biblioteca, **pyplot** é um pacote que inclui todas as funções MATLAB para usar funções MATLAB em Python.

Finalmente, podemos usar **plt** para chamar funções dentro do arquivo python.

## Linha vertical

Para plotar uma linha vertical com pyplot, você pode usar a função **`axvline`** ().

A sintaxe de `axvline` é a seguinte:

**`plt.axvline (x = 0, ymin = 0, ymax = 1, **kwargs)`**

Nesta sintaxe: `x` é a coordenada para o eixo x. Este ponto é de onde a linha seria gerada verticalmente. `ymin` é a parte inferior do gráfico; `ymax` é o topo do gráfico. `**kwargs` são as propriedades da linha, como cor, rótulo, estilo de linha, etc.

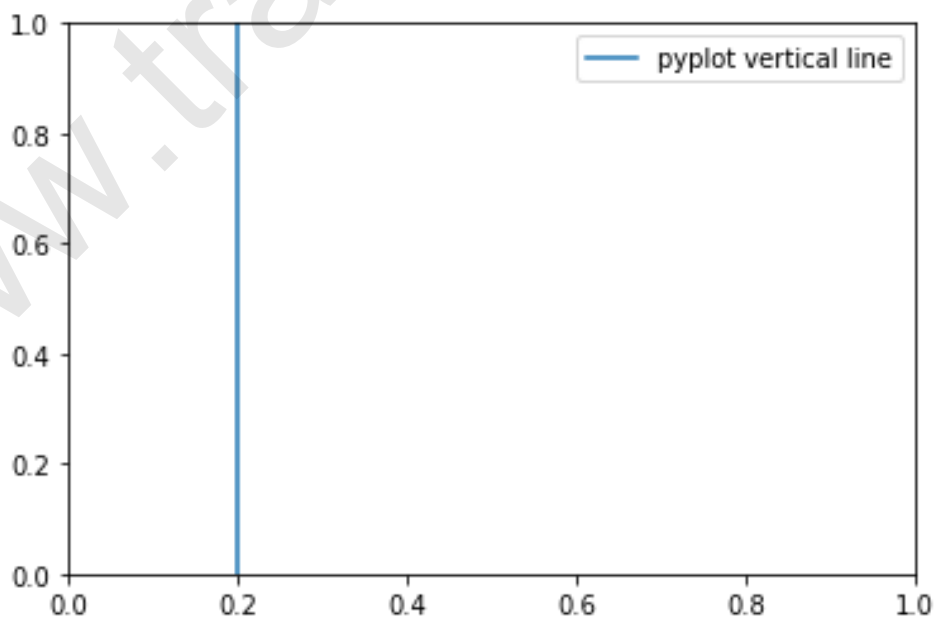
Neste exemplo, desenhamos uma linha vertical. 0.2 significa que a linha será desenhada no ponto 0,2 no gráfico. 0 e 1 são `ymin` e `ymax` respectivamente.

O rótulo (`label`) é uma das propriedades da linha. `legend ()` é a função MATLAB que ativa a etiqueta no gráfico. Finalmente, `show ()` irá abrir o gráfico ou a tela do gráfico. Vamos testar o código abaixo:

```
plt.axvline ( 0.2 , 0 , 1 , label = 'pyplot vertical line' )
```

```
plt.legend ()
```

```
plt.show ()
```



Linha horizontal O `axhline()` traça uma linha horizontal ao longo. A sintaxe para `axhline()` é a seguinte:

**`plt.axhline (y = 0, xmin = 0, xmax = 1, **kwargs)`**

Na sintaxe: `y` são as coordenadas ao longo do eixo `y`. Esses pontos são de onde a linha seria gerada horizontalmente. `xmin` é a esquerda do gráfico; `xmax` é o lado direito do gráfico. `**kwargs` são as propriedades da linha, como cor, rótulo, estilo de linha, etc.

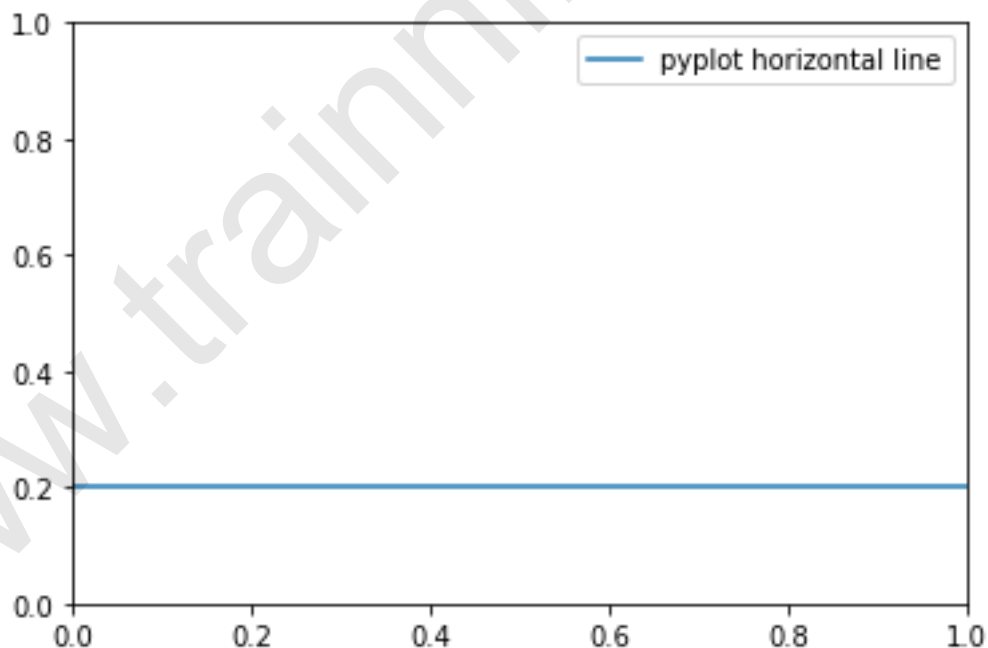
Substituindo `axvline()` por `axhline()` no exemplo anterior, você terá uma linha horizontal no gráfico:

```
ypoints = 0.2

plt.axhline (ypoints, 0 , 1 , label = 'pyplot horizontal line' )

plt.legend ()

plt.show ()
```



### Múltiplas Linhas

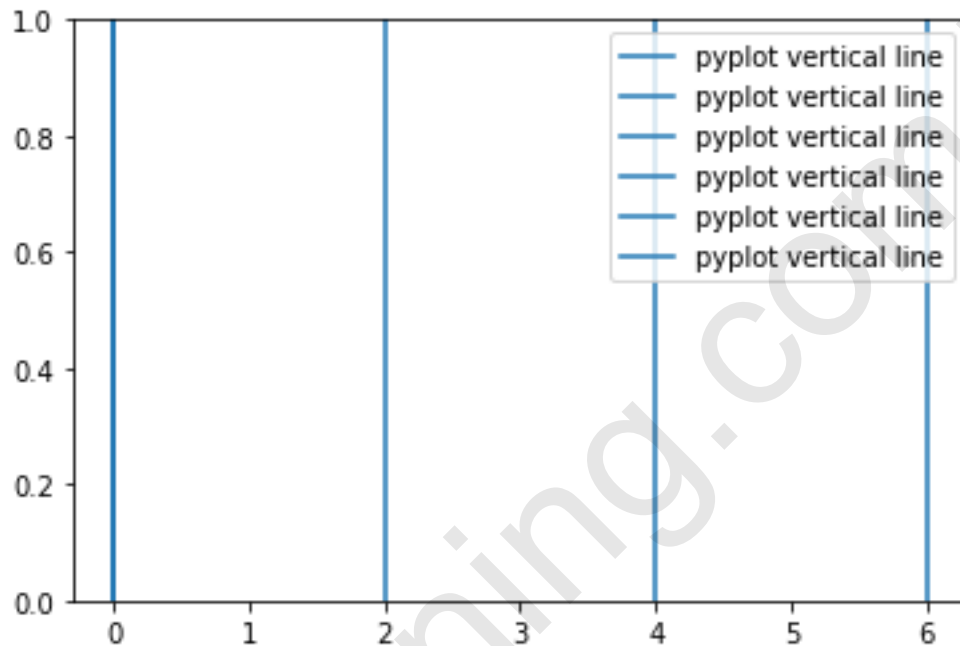
Para plotar várias linhas verticais, podemos criar uma matriz de `x` pontos / coordenadas e, em seguida, iterar em cada elemento da matriz para plotar mais de uma linha. O resultado será:

```
xpoints = [ 0,2 , 0,4 , 0,6 ]

for p in xpoints:
    plt.axvline (p, label = 'pyplot vertical line' )

plt.legend ()

plt.show ()
```



A saída acima não parece muito atraente; podemos usar cores diferentes para cada linha também no gráfico.

Considere o exemplo abaixo:

```
xpoints = [0.2, 0.4, 0.6]

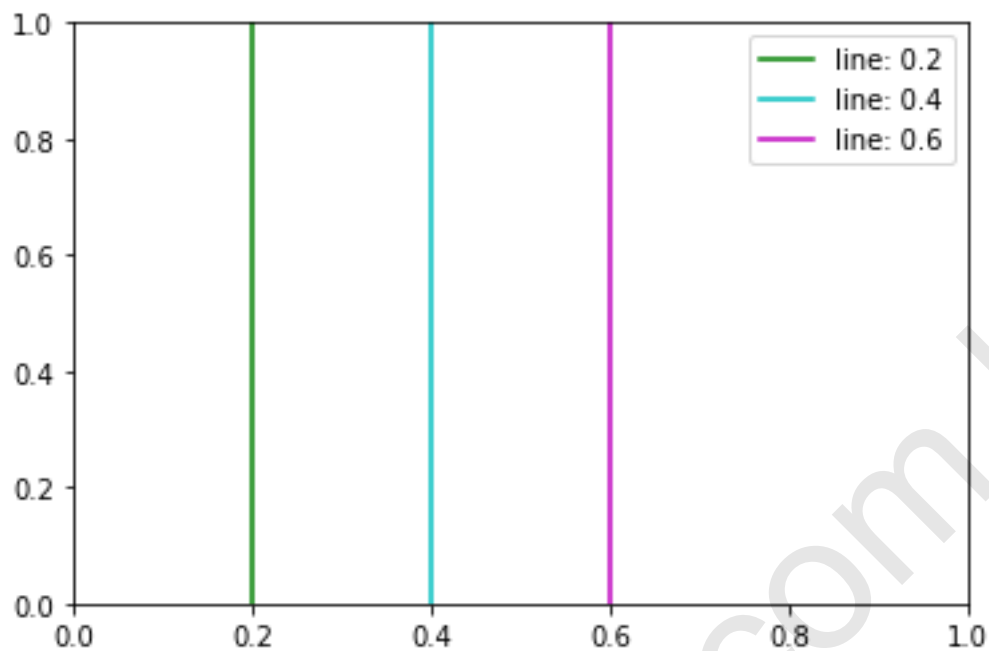
colors = ['g', 'c', 'm']

for p, c in zip(xpoints, colors):
    plt.axvline(p, label='line: {}'.format(p), c=c)

plt.legend()

plt.show()
```



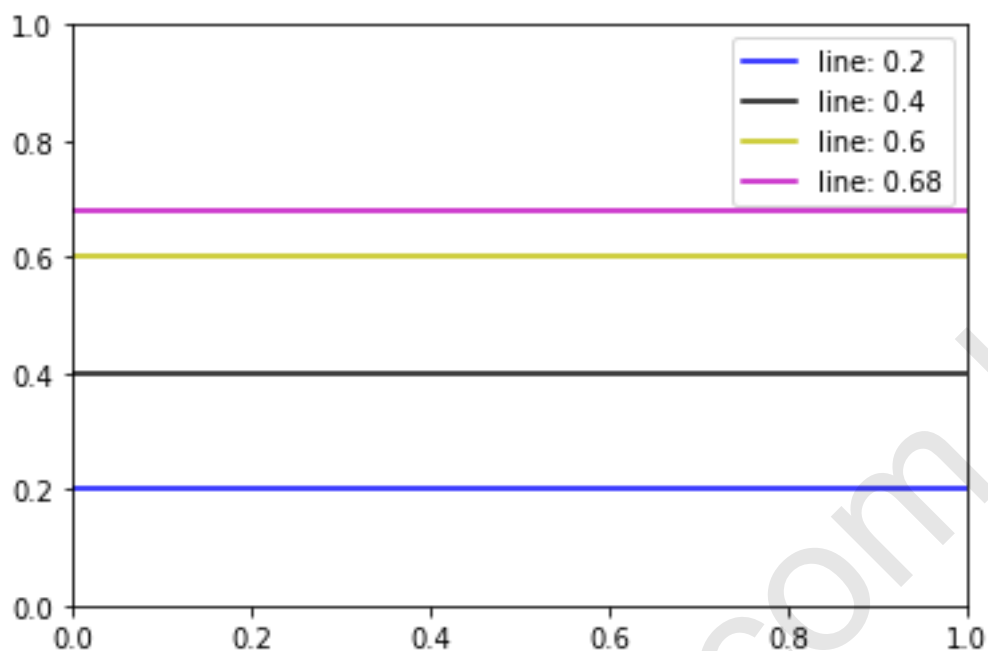


Neste exemplo, acima, temos uma matrizde linhas e uma matrizde símbolos de cores Python. Usando a função `zip ()` , ambos os arrays são mesclados: o primeiro elemento de `xpoints [ ]` com o primeiro elemento do `matrizcolor [ ]` . Dessa forma, a primeira linha = verde, segunda linha = ciano, etc.

As chaves `{}` atuam como um marcador para adicionar variáveis Python à impressão com a ajuda da função `format ()` . Portanto, temos `xpoints [ ]` no gráfico.

Basta substituir `axvline ()` por `axhline ()` no exemplo anterior, e você terá várias linhas horizontais no gráfico. Implemente o código abaixo:

```
ypoints = [0.2, 0.4, 0.6, 0.68]
colors = ['b', 'k', 'y', 'm']
for p, c in zip(ypoints, colors):
    plt.axhline(p, label='line: {}'.format(p), c=c)
plt.legend()
plt.show()
```



O código, acima, é o mesmo; temos uma matriz de quatro pontos do eixo y e cores diferentes neste momento. Ambas as matrizes são mescladas com a função `zip()`, iteradas na matriz final e `axhline()` plota as linhas conforme mostrado na saída acima.

### Salvar Figura

Depois de traçar seu gráfico, como salvar o gráfico de saída?

Para salvar o gráfico, use `savefig()` do pyplot.

#### **`plt.savefig ( fname , **kwargs )`**

Onde `fname` é o nome do arquivo, o destino ou caminho também pode ser especificado junto com o nome do arquivo. O parâmetro `kwargs` é opcional. Você pode usá-lo para alterar a orientação, formato, cor do rosto, qualidade, dpi, etc.

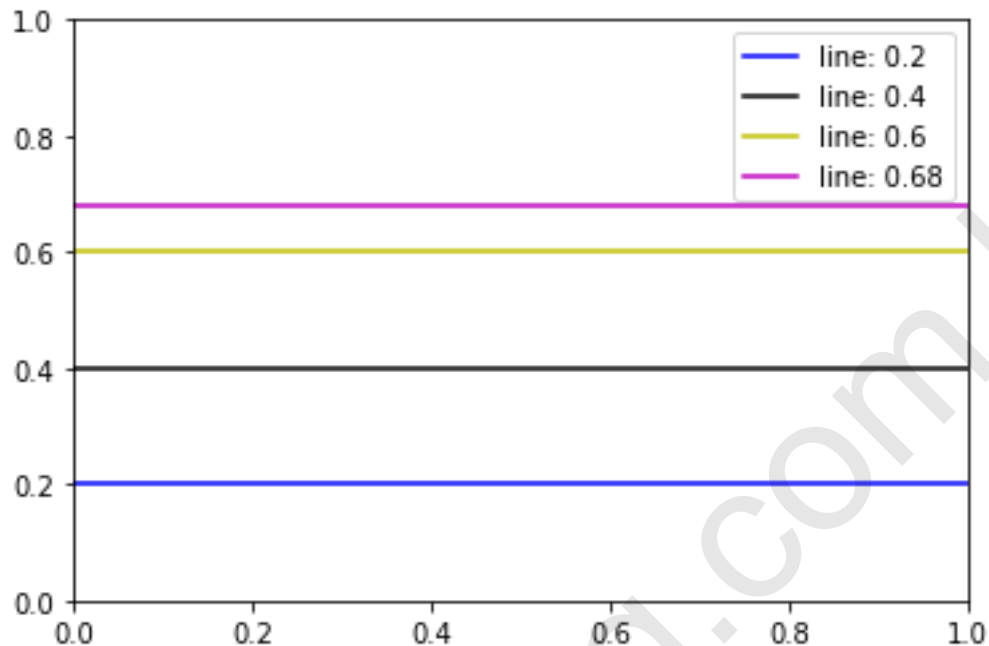
```
ypoints = [0.2, 0.4, 0.6, 0.68]
colors = ['b', 'k', 'y', 'm']

for p, c in zip(ypoints, colors):
    plt.axhline(p, label='line: {}'.format(p), c=c)

plt.savefig('horizontal_lines.png')
```

```
plt.legend()
```

```
plt.show()
```



O nome do arquivo é `horizontal_lines.png`; o arquivo estará no mesmo diretório de trabalho que contem seu Jupyter Notebook.

### Plots múltiplos

Todos os exemplos anteriores foram sobre plotagem em um gráfico. Também é possível plotar vários gráficos dentro de uma mesma figura.

Você pode gerar vários gráficos na mesma figura com a ajuda da função `subplot()` do Python `pyplot`.

**`matplotlib.pyplot.subplot ( nrows , ncols , index , ** kwargs )`**

Em argumentos, temos três inteiros para especificar, o número de gráficos em uma linha e em uma coluna e, em seguida, em qual índice o gráfico deve estar. Você pode considerá-lo como uma grade, e estamos desenhando em suas células.

O primeiro número seria `nrows` o número de linhas; o segundo seria `ncols` o número de colunas e então o índice. Outros argumentos opcionais (`** kwargs`) incluem cor, rótulo, título, snap, etc.

Observe o código a abaixo para plotar mais de um gráfico em uma figura.

```

from matplotlib import pyplot as plt

plt.subplot(1, 2, 1)

x1 = [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]

y1 = [5, 10, 15, 20, 25, 30, 35, 40, 45, 50]

plt.plot(x1, y1, color = "c")

plt.subplot(1, 2, 2)

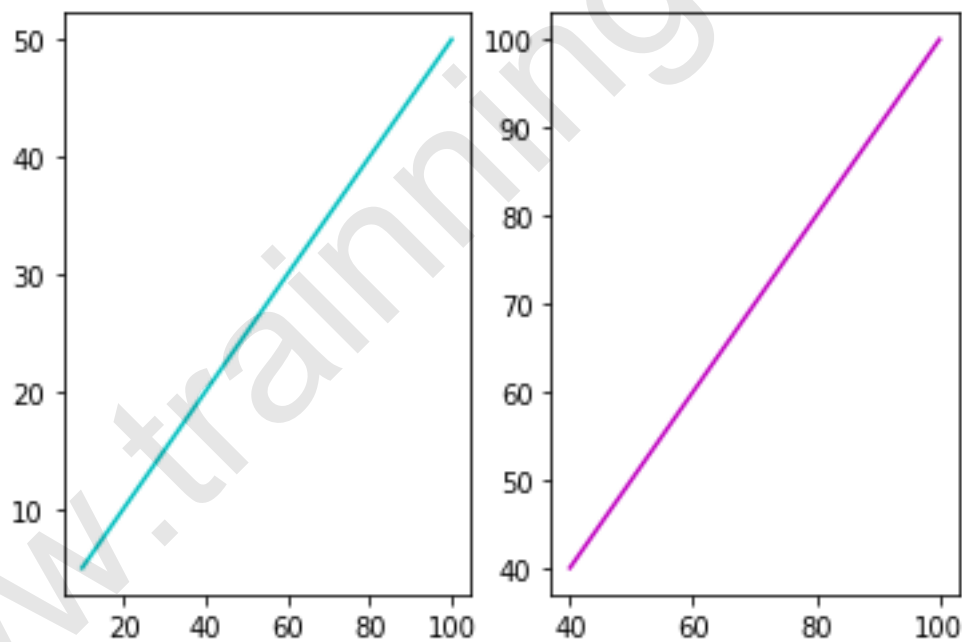
x2 = [40, 50, 60, 70, 80, 90, 100]

y2 = [40, 50, 60, 70, 80, 90, 100]

plt.plot(x2, y2, color = "m")

plt.show()

```



Para plotar gráficos horizontais, altere os valores das linhas e colunas do subplot como:

**`plt.subplot(2, 1, 1)` `plt.subplot(2, 1, 2)`**

Isso significa que temos 2 linhas e 1 coluna. Implemente e execute o código para observar a saída:

```
plt.subplot(2, 1, 1)

x1 = [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]

y1 = [5, 10, 15, 20, 25, 30, 35, 40, 45, 50]

plt.plot(x1, y1, color = "c")

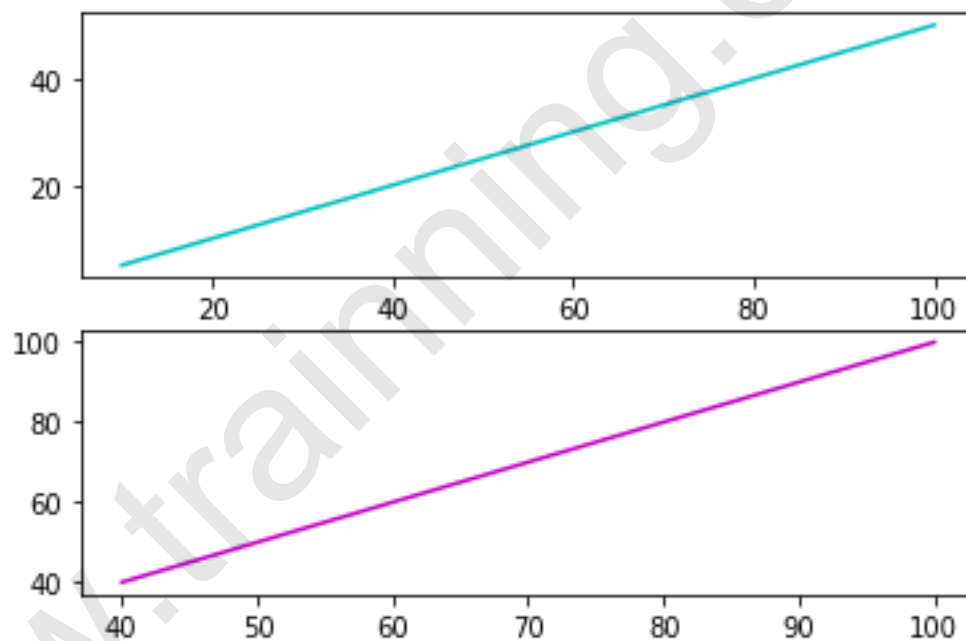
plt.subplot(2, 1, 2)

x2 = [40, 50, 60, 70, 80, 90, 100]

y2 = [40, 50, 60, 70, 80, 90, 100]

plt.plot(x2, y2, color = "m")

plt.show()
```



Agora vamos criar uma grade de gráficos 2 × 2.

Considere o código abaixo. Implementando e executando o código abaixo resultado será o seguinte:

```
from matplotlib import pyplot as plt

plt.subplot(2, 2, 1)

x1 = [40, 50, 60, 70, 80, 90, 100]
```

```
y1 = [40, 50, 60, 70, 80, 90, 100]

plt.plot(x1, y1, color = "c")

plt.subplot(2, 2, 2)

x2 = [40, 50, 60, 70, 80, 90, 100]

y2 = [40, 50, 60, 70, 80, 90, 100]

plt.plot(x2, y2, color = "m")

plt.subplot(2, 2, 3)

x3 = [40, 50, 60, 70, 80, 90, 100]

y3 = [40, 50, 60, 70, 80, 90, 100]

plt.plot(x3, y3, color = "g")

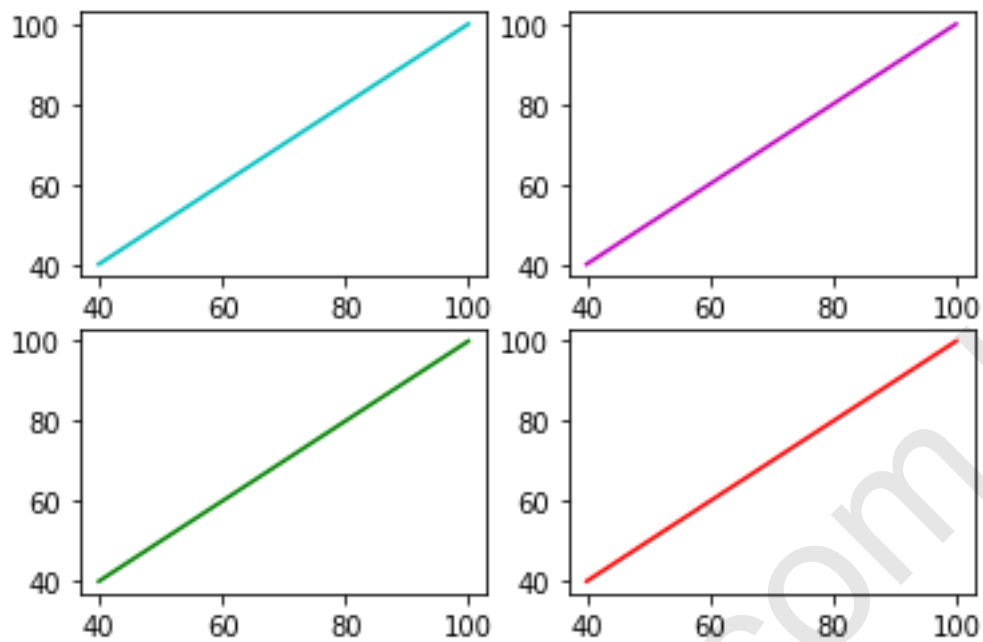
plt.subplot(2, 2, 4)

x4 = [40, 50, 60, 70, 80, 90, 100]

y4 = [40, 50, 60, 70, 80, 90, 100]

plt.plot(x4, y4, color = "r")

plt.show()
```



Neste código acima, 2,2,1 significa 2 linhas, 2 colunas e o gráfico estará no índice 1. Da mesma forma, 2,2,2 significa 2 linhas, 2 colunas e o gráfico estará no índice 2 da grade .

### Axis Range

É possível definir o intervalo ou limite dos eixos x e y usando as funções `xlim ()` e `ylim ()` do pyplot, respectivamente.

***`matplotlib.pyplot.xlim([starting_point, ending_point])`***

***`matplotlib.pyplot.ylim([starting_point, ending_point])`***

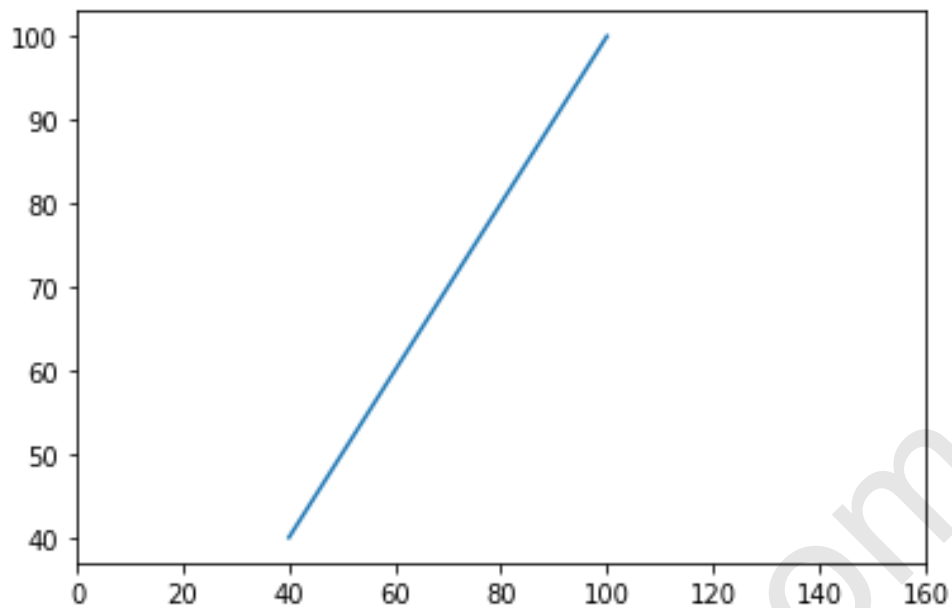
O exemplo abaixo permite definir o limite do eixo x para o gráfico:

```
from matplotlib import pyplot as plt
x1 = [40, 50, 60, 70, 80, 90, 100]
y1 = [40, 50, 60, 70, 80, 90, 100]

plt.plot(x1, y1)

plt.xlim([0,160])

plt.show()
```



No grafico acima, os pontos no eixo x irão de 0 a 160.

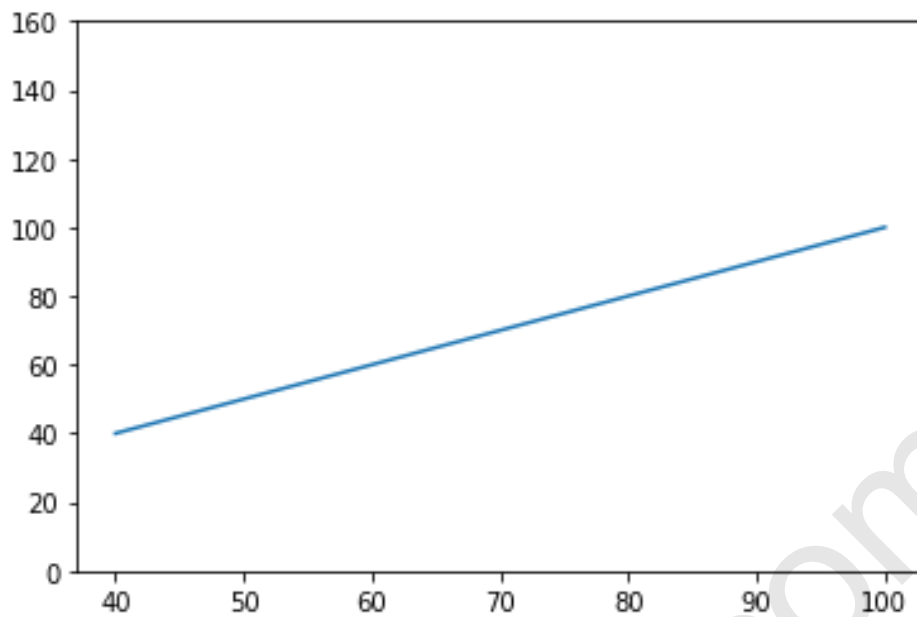
Da mesma forma, para limitar as coordenadas do eixo y, você colocará a seguinte linha de código:

**`plt.ylim([0,160])`**

Após a implementação e execução do código abaixo o resultado serão seguinte:

```
from matplotlib import pyplot as plt
x1 = [40, 50, 60, 70, 80, 90, 100]
y1 = [40, 50, 60, 70, 80, 90, 100]
plt.plot(x1, y1)
plt.ylim([0,160])
plt.show()
```





### Label Axis

Você pode criar os rótulos para os eixos x e y usando as funções `xlabel()` e `ylabel()` do `pyplot`.

**`matplotlib.pyplot.xlabel(labeltext, labelfontdict, *kwargs)`**

**`matplotlib.pyplot.ylabel(labeltext, labelfontdict, *kwargs)`**

Na sintaxe acima, `labeltext` é o texto do rótulo e é uma string; `labelfont` descreve o tamanho da fonte, peso, família do texto do rótulo é opcional.

```
x1 = [40, 50, 60, 70, 80, 90, 100]
```

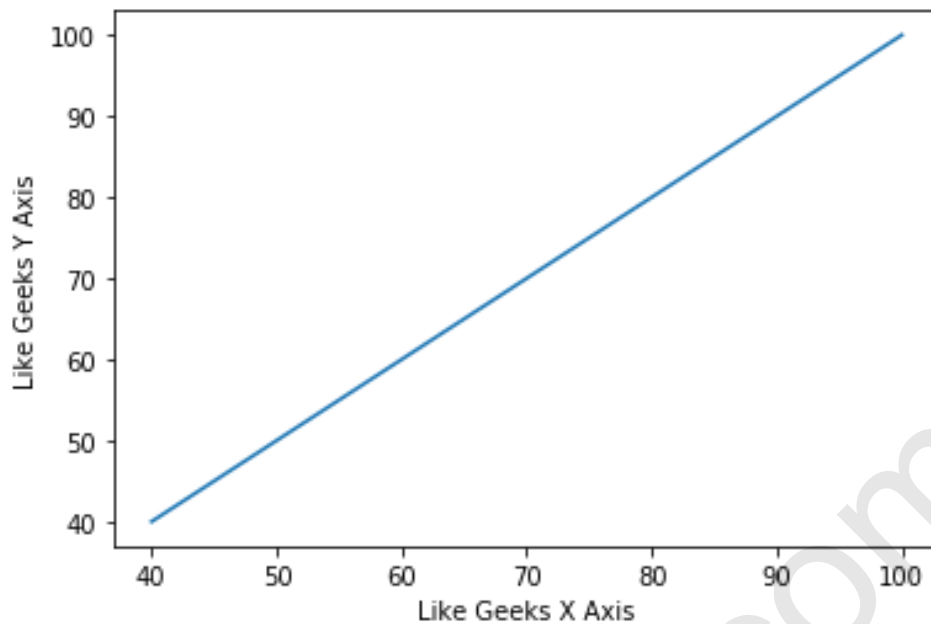
```
y1 = [40, 50, 60, 70, 80, 90, 100]
```

```
plt.plot(x1, y1)
```

```
plt.xlabel('Like Geeks X Axis')
```

```
plt.ylabel('Like Geeks Y Axis')
```

```
plt.show()
```



No exemplo acima, temos matriz x e y regulares para as coordenadas x e y, respectivamente. Então `plt.xlabel ()` gera um texto para o eixo x e `plt.ylabel ()` gera um texto para o eixo y.

### Clear (limpar) Plot

A função `clf ()` do pyplot limpa o gráfico.

#### **`matplotlib.pyplot.clf()`**

Na função `clf ()`, não temos argumentos.

```
x1 = [40, 50, 60, 70, 80, 90, 100]
```

```
y1 = [40, 50, 60, 70, 80, 90, 100]
```

```
plt.plot(x1, y1)
```

```
plt.xlabel('Like Geeks X Axis')
```

```
plt.ylabel('Like Geeks Y Axis')
```

```
plt.clf()
```

```
plt.show()
```

```
<Figure size 432x288 with 0 Axes>
```

Neste código acima, criamos um gráfico e também definimos rótulos. Depois disso, usamos a função `clf ()` para limpar o gráfico.

## Seaborn Data Visualization

### Heat Map (Visualização com mapa de calor)

Neste projeto, vamos representar os dados em um mapa de calor usando uma biblioteca Python chamada seaborn. Esta biblioteca é usada para visualizar dados baseados no Matplotlib .

Vamos entender o que é um mapa de calor, como criá-lo, como alterar suas cores, ajustar o tamanho da fonte e muito mais.

### O que é um mapa de calor?

O mapa de calor é uma forma de representar os dados em uma formato bidimensional. Os valores dos dados são representados como cores no gráfico. O objetivo do mapa de calor é fornecer um resumo visual colorido das informações.

### Criando um mapa de calor

Para criar um mapa de calor, usaremos a biblioteca seaborn com Python. Seaborn é baseada em Matplotlib. Fornece uma interface de visualização de dados de alto nível onde podemos desenhar qualquer array.

Seaborn suporta as seguintes polts:

- Plot distribuição (Distribution plots)
- Matrix Plots
- Gráficos de regressão (Regression Plots)
- Gráficos de séries temporais (Time series plots)
- Gráficos categóricos (Categorical Plots)

Importe os seguintes módulos necessários:

```
import numpy as np

import seaborn as sb

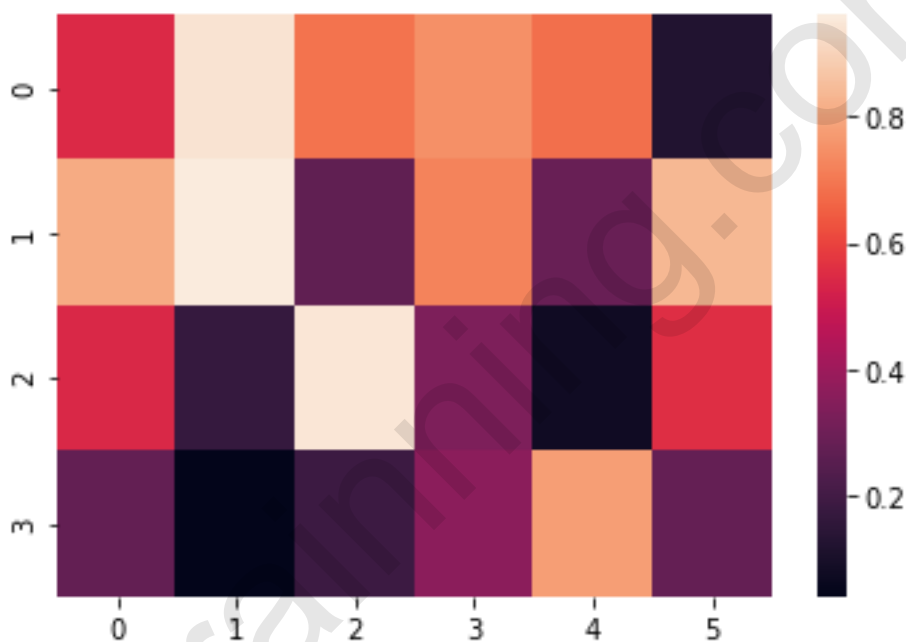
import matplotlib.pyplot as plt
```

Importamos o módulo numpy para gerar uma matriz de números aleatórios entre um determinado intervalo, que será plotada como um mapa de calor.

```
data = np.random.rand(4, 6)
```

Isso criará uma matriz bidimensional com quatro linhas e seis colunas. Agora vamos armazenar esses valores de matriz no mapa de calor. Podemos criar um mapa de calor usando a função `heatmap` da biblioteca Seaborn. Então, passaremos os dados usando `matplotlib`, exibiremos o mapa de calor na saída:

```
heat_map = sb.heatmap (data)
plt.show ()
```



#### Remover heatmap 'x' tick labels

Os valores no eixo 'x' e 'y' para cada bloco no mapa de calor são chamados de rótulos de escala (tick labels). Seaborn adiciona os tick labels por padrão. Se quisermos remove-los, podemos definir o atributo `xticklabel` ou `yticklabel` do mapa de calor do mar como `False`, conforme abaixo:

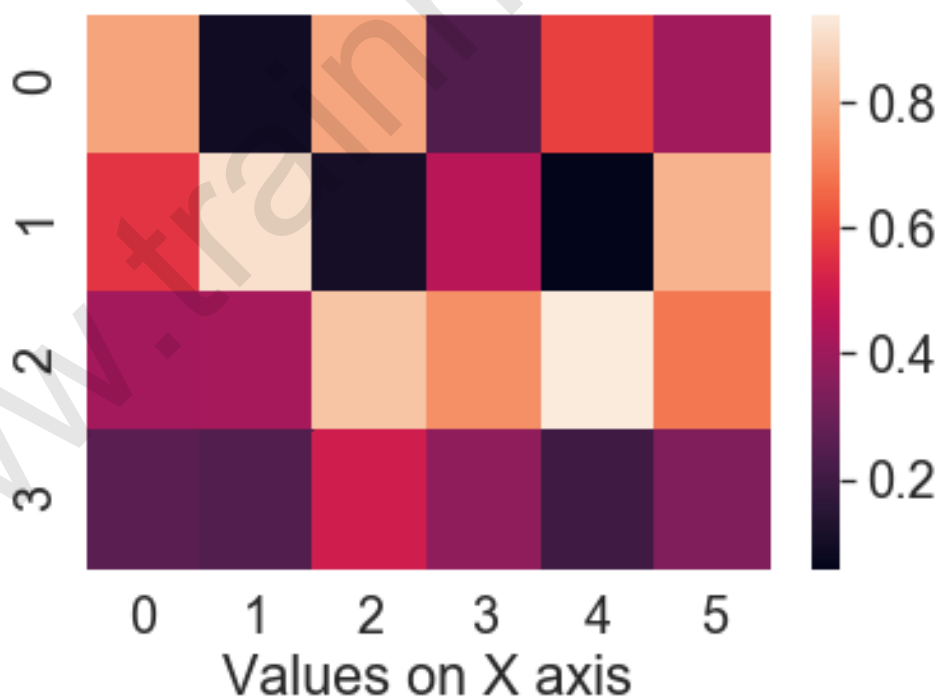
```
heat_map = sb.heatmap(data, xticklabels=False, yticklabels=False)
```



### Definir rótulo do eixo 'x' do mapa de calor

Podemos adicionar um rótulo no eixo x usando o atributo xlabel de Matplotlib conforme mostrado no código a seguir:

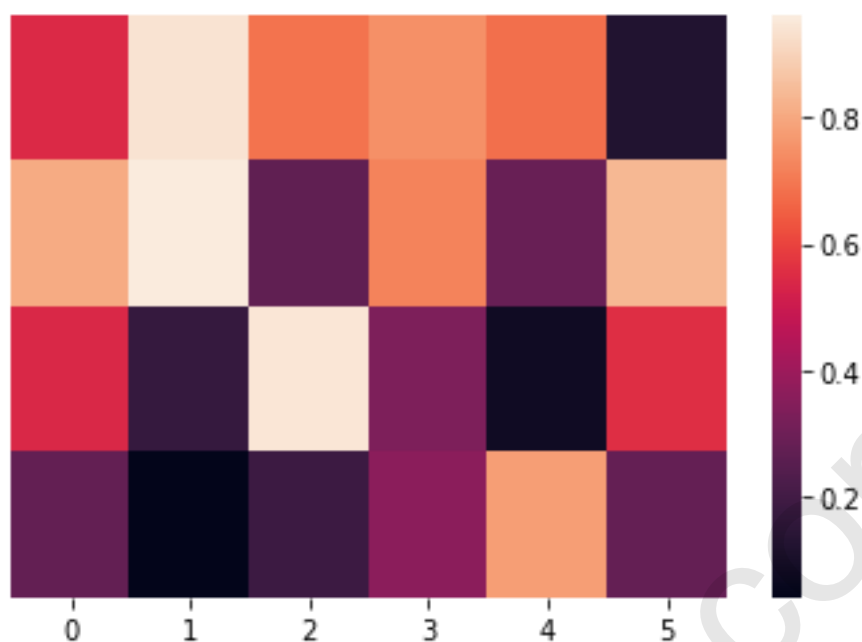
```
heat_map = sb.heatmap(data)
plt.xlabel("Values on X axis")
plt.show()
```



### Remover heatmap 'y' tick labels

Seaborn adiciona os rótulos para o eixo y por padrão. Para removê-los, podemos definir os yticklabels como false.

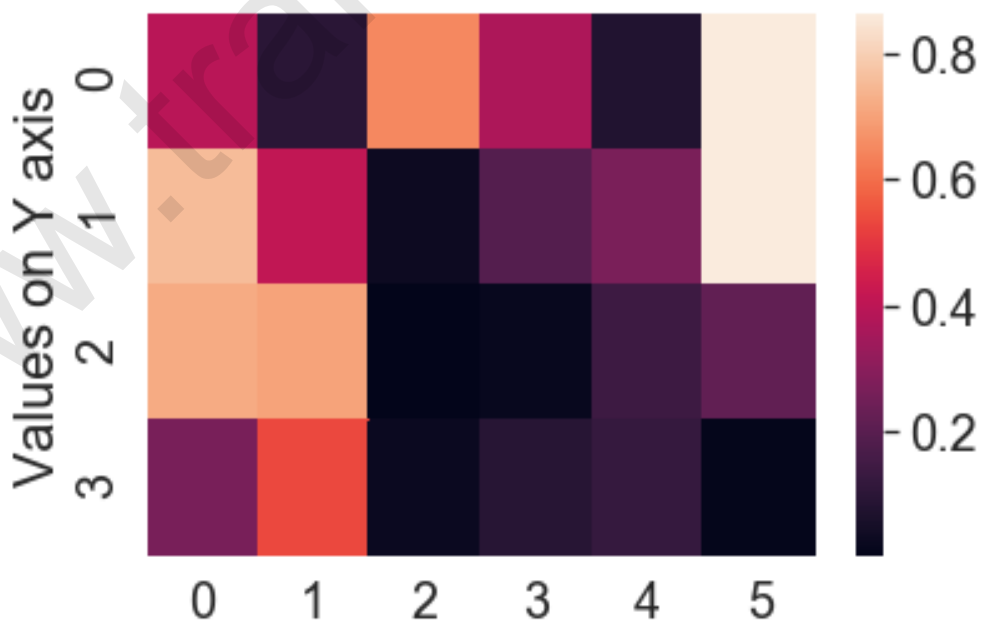
```
heat_map = sb.heatmap(data, yticklabels=False)
```



### Definir rótulo do eixo 'y' do mapa de calor

Podemos adicionar um rótulo no eixo y usando o atributo ylabel de Matplotlib conforme mostrado no código a seguir:

```
data = np.random.rand(4, 6)
heat_map = sb.heatmap(data)
plt.ylabel('Values on Y axis')
plt.show()
```

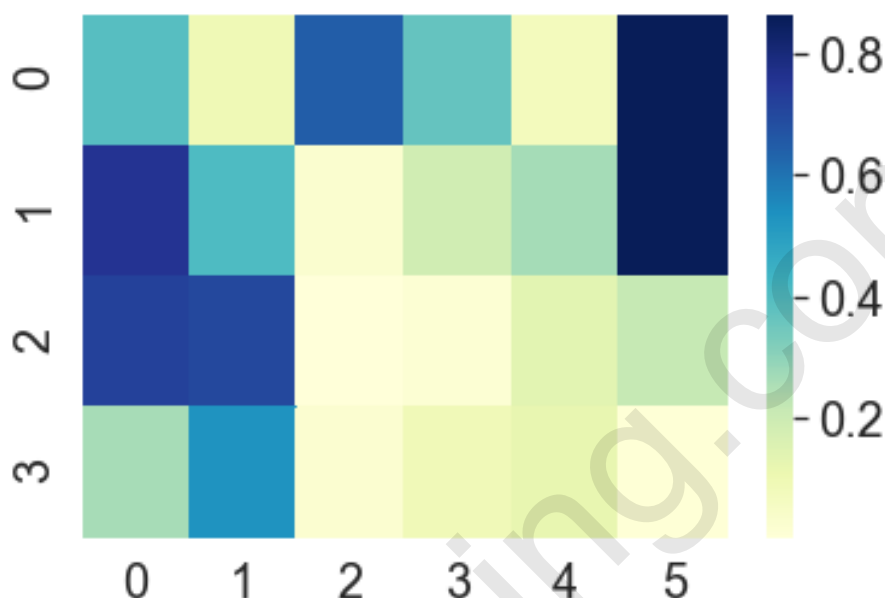


### Alterando a cor do mapa de calor

Você pode alterar a cor do mapa de calor de origem marinha usando o mapa de cores usando o atributo `cmap`.

Considere o código abaixo:

```
heat_map = sb.heatmap(data, cmap="YlGnBu")
plt.show()
```



Aqui, `cmap` é igual a 'YlGnBu', que representa a seguinte cor:

No mapa de calor Seaborn, temos três tipos diferentes de mapas de cores.

- Mapas de cores sequenciais (Sequential colormaps)
- Paleta de cores divergentes (Diverging color palette)
- Dados discretos (Discrete Data)

### Mapa de cores sequencial (Sequential colormaps)

Você pode usar o mapa de cores sequencial (Sequential colormaps) quando os dados variam de um valor baixo a um valor alto. Os códigos de cores sequenciais do mapa de cores podem ser usados com a função `heatmap()` ou `kdeplot()`. O mapa de cores sequencial contém as seguintes cores:

 Esta imagem foi tirada de Matplotlib.org.

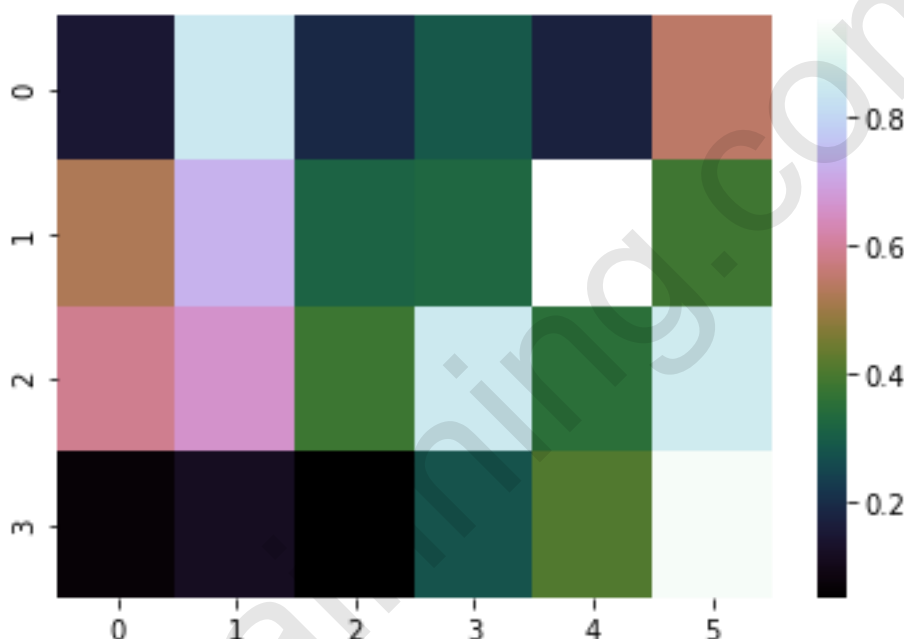
### Paleta de cubo sequencial

O cubehelix é uma forma de mapa de cores sequencial. Você pode usá-lo quando o brilho for aumentado linearmente e quando houver uma ligeira diferença no matiz.

**A paleta cubehelix se parece com o seguinte:**

**e resultado será:**

```
heat_map = sb.heatmap(data, cmap="cubehelix")
```



### Paleta de cores divergentes (Diverging color palette)

Você pode usar a paleta de cores divergentes quando os valores alto e baixo são importantes no mapa de calor.

A paleta divergente cria uma paleta entre duas cores HUSL. Isso significa que a paleta divergente contém dois tons diferentes em um gráfico.

Você pode criar a paleta divergente, usando seaborn, como mostrado abaixo. A saída é obtida usando a seguinte linha de código:

```
sb.palplot(sb.mpl_palette("Set3", 11))
plt.show()
```





O argumento 'Set3' é o nome da paleta e 11 é o número de cores discretas na paleta. O método palplot do seaborn plota os valores em uma matriz horizontal da paleta de cores fornecida.

### Adicionar texto sobre mapa de calor

Para adicionar texto sobre o mapa de calor, podemos usar o atributo 'annot'. Se 'annot' for definido como True, o texto será escrito em cada célula. Se os rótulos de cada célula forem definidos, você pode atribuir os rótulos ao atributo 'annot'.

Considere o código abaixo - ele nos trará a seguinte saída:

```
data = np.random.rand(4, 6)
heat_map = sb.heatmap(data, annot=True)
plt.show()
```



Podemos personalizar o valor da anotação.

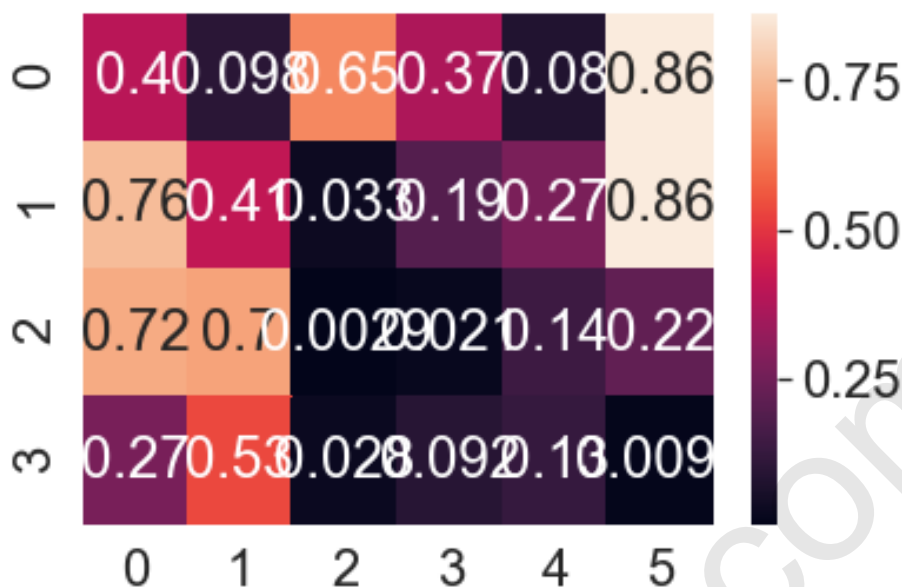
### Ajustar o tamanho da fonte do mapa de calor

Podemos ajustar o tamanho da fonte do texto do mapa de calor usando o atributo 'font\_scale' da biblioteca Seaborn desta forma:

```
sb.set(font_scale=2)
```

Agora, defina e mostre o mapa de calor. Após defini-lo, terá a seguinte aparência - após aumentar o tamanho da fonte:

```
heat_map = sb.heatmap(data, annot=True)
plt.show()
```



### Seaborn heatmap colorbar

A barra de cores no mapa de calor é semelhante a esta que observamos na figura plotada acima. O atributo `cbar` do mapa de calor é um atributo booleano; diz se deve ou não aparecer no gráfico. Se o atributo `cbar` não for definido, a barra de cores será exibida no gráfico por padrão. Para remove-la, defina `cbar` como `False`:

```
sb.set(font_scale=1)
heat_map = sb.heatmap(data, annot=True, cbar=False)
plt.show()
```



Para adicionar um título de barra de cores, podemos usar o atributo `cbar_kws`.

O código será semelhante ao seguinte:

```
heat_map = sb.heatmap(data, annot=True, cbar_kws={'label': 'My Colorbar'})
plt.show()
```

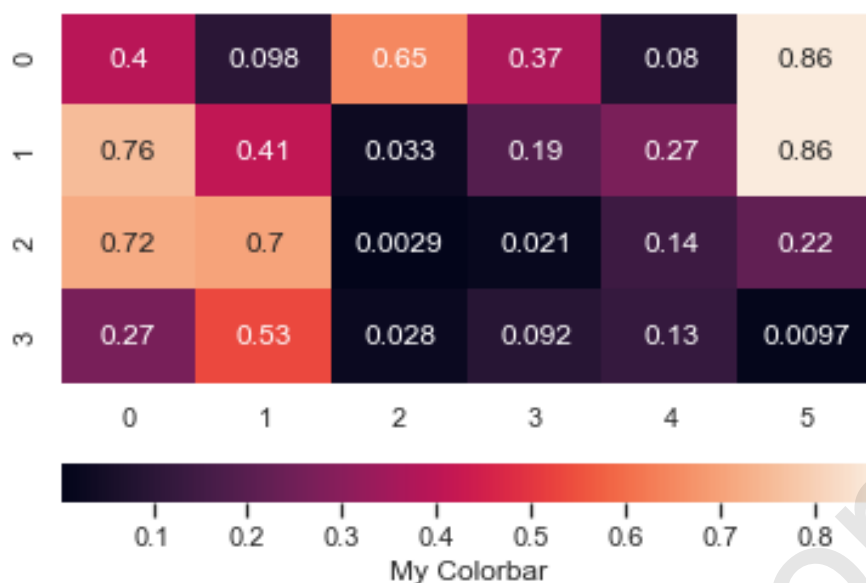


No `cbar_kws`, temos que especificar a qual atributo da barra de cores estamos nos referindo. Em nosso exemplo, estamos nos referindo ao rótulo (título) da barra de cores.

Da mesma forma, podemos mudar a orientação da cor. A orientação padrão é vertical como no exemplo acima.

Para criar uma barra de cores horizontal, defina o atributo de orientação de `cbar_kws` da seguinte maneira:

```
heat_map = sb.heatmap(data, annot=True, cbar_kws={'label': 'My Colorbar', 'orientation': 'horizontal'})
plt.show()
```

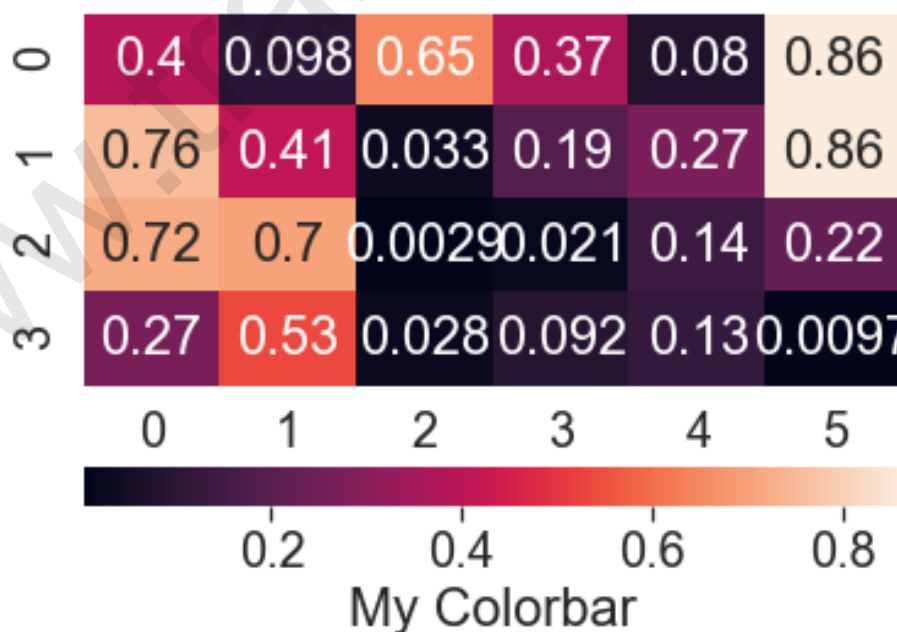


### Alterar o tamanho da fonte da barra de cores do mapa de calor

Se precisarmos alterar o tamanho da fonte de todos os componentes do seaborn, você pode usar o atributo `font_scale` do Seaborn.

Vamos definir a escala para 1,8 e comparar uma escala 1 - acima - com 1,8. A escala de 1,8 será parecida com esta:

```
sb.set(font_scale=1.8)
heat_map = sb.heatmap(data, annot=True, cbar_kws={'label': 'My Colorbar', 'orientation': 'horizontal'})
plt.show()
```



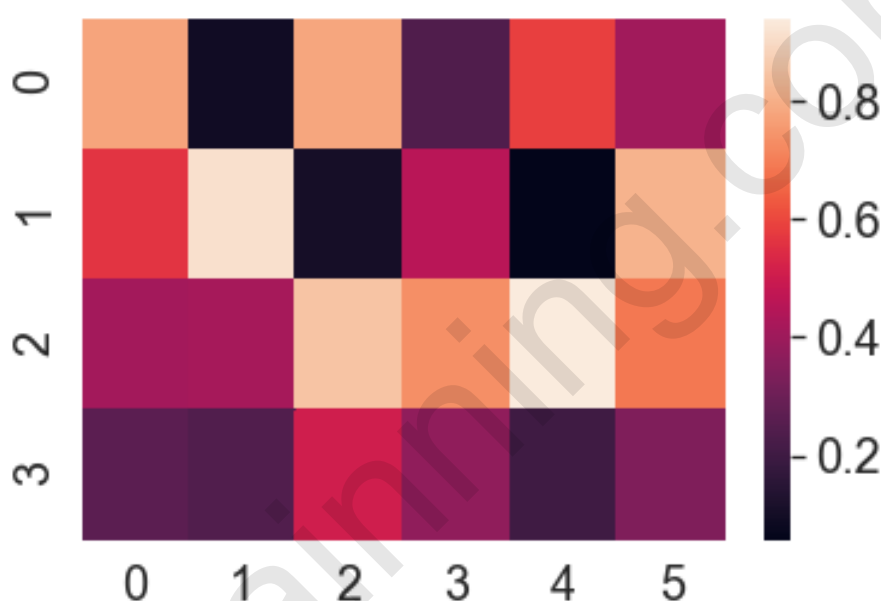
## Alterar a rotação do tick axis

Podemos alterar a rotação dos rótulos de escala (tick labels) usando o atributo de rotação dos rótulos `ytick` ou `xtick` necessários.

Primeiro, definimos o mapa de calor; nosso gráfico é regular com dados aleatórios conforme definido na anteriormente.

Observe os `yticklabels` originais na imagem a seguir:

```
heat_map = sb.heatmap(data)
plt.show()
```



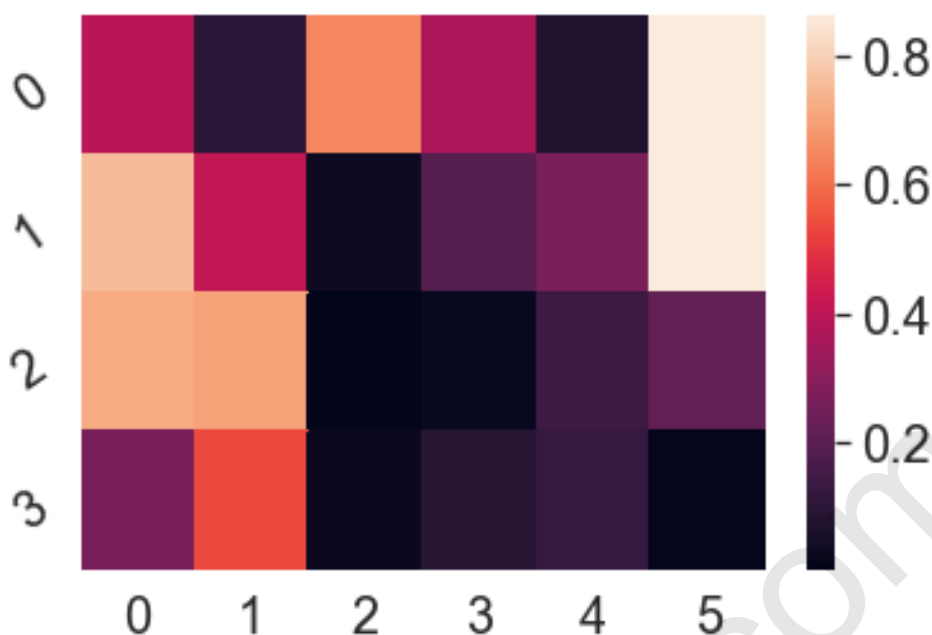
Para girá-los, primeiro obteremos os `yticklabels` do mapa de calor e, em seguida, definiremos a rotação como 0:

```
heat_map = sb.heatmap(data)
heat_map.set_yticklabels(heat_map.get_yticklabels(), rotation=0)
```

*#0 atributo de rotação pode ser qualquer ângulo -aqui, giramos em 35°:*

```
heat_map.set_yticklabels(heat_map.get_yticklabels(), rotation=35)
```

```
[Text(0, 0.5, '0'), Text(0, 1.5, '1'), Text(0, 2.5, '2'), Text(0, 3.5, '3')]
```



### Adicionar texto e valores no mapa de calor

Na passo anterior, adicionamos apenas valores ao mapa de calor. Nesta seção, adicionaremos valores junto com o texto no mapa de calor.

Considere o seguinte exemplo: vamos criar uma matriz para o texto que escreveremos no mapa de calor. Observe o código abaixo:

```
text = np.asarray([[ 'a', 'b', 'c', 'd', 'e', 'f'], ['g', 'h', 'i', 'j', 'k', 'l'], ['m', 'n', 'o', 'p', 'q', 'r'], ['s', 't', 'u', 'v', 'w', 'x']])
```

Agora temos que combinar o texto com os valores e adicionar o resultado ao mapa de calor como um rótulo:

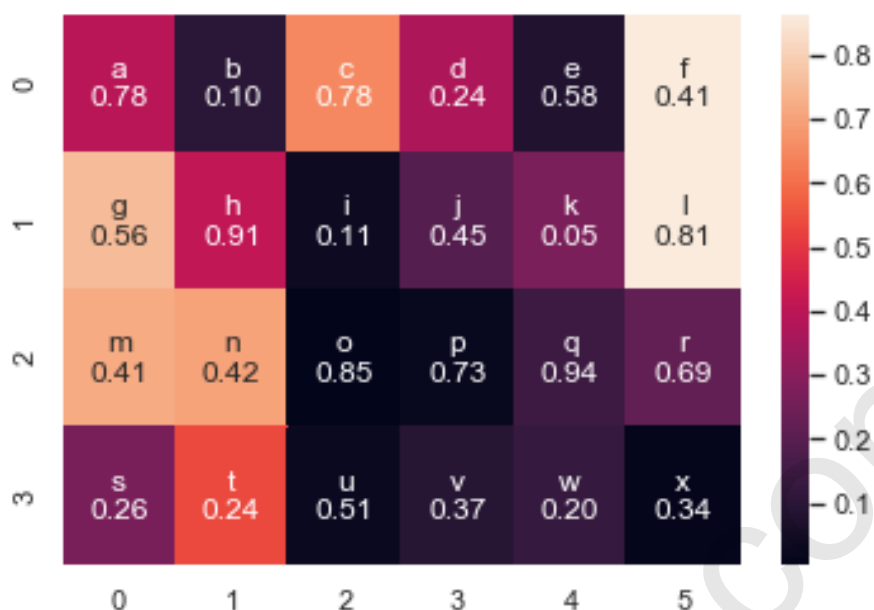
```
labels = (np.asarray(["{0}\n{1:.2f}".format(text, data) for text, data in zip(text.flatten(), data.flatten())])).reshape(4, 6)
```

Aqui, passamos os dados na matriz de texto e na matriz de dados e, em seguida, 'compactamos' (usando a instrução zip) as duas matrizes em um texto mais simples e os compactamos juntos. O resultante é então remodelado para criar outra matriz do mesmo tamanho, que agora contém texto e dados.

A nova matriz é armazenada em uma variável chamada rótulos. A variável de rótulos será adicionada ao mapa de calor usando anotação:

```
sb.set(font_scale=1)
heat_map = sb.heatmap(data, annot=labels, fmt='')

```



## O que é NumPy?

NumPy é o pacote fundamental para computação científica em Python. É uma biblioteca Python que fornece um objeto de matriz multidimensional, vários objetos derivados (como matriz e matriz mascaradas) e uma variedade de rotinas para operações rápidas em matriz, incluindo matemática, lógica, manipulação de forma, classificação, seleção, E / S, transformadas discretas de Fourier, álgebra linear básica, operações estatísticas básicas, simulação aleatória e muito mais.

No núcleo do pacote NumPy, está o objeto `ndmatrix`. Isso encapsula matriz n- dimensionais de tipos de dados homogêneos, com muitas operações sendo realizadas em código compilado para desempenho. Existem várias diferenças importantes entre as matriz NumPy e as sequências Python padrão:

- Os arrays NumPy têm um tamanho fixo na criação, ao contrário das listas Python (que podem crescer dinamicamente). Alterar o tamanho de um `ndmatrix` criará um novo matriz e excluirá o original.
- Os elementos em uma matriz NumPy devem ser todos do mesmo tipo de dados e, portanto, terão o mesmo tamanho na memória. A

exceção: pode-se ter arrays de objetos (Python, incluindo NumPy), permitindo assim arrays de elementos de tamanhos diferentes.

- Os arrays NumPy facilitam operações matemáticas avançadas e outros tipos de operações em um grande número de dados. Normalmente, essas operações são executadas de forma mais eficiente e com menos código do que é possível usando as sequências integradas do Python.
- Uma crescente abundância de pacotes científicos e matemáticos baseados em Python está usando matriz NumPy; embora eles normalmente suportem entrada de sequência Python, eles convertem essa entrada em matriz NumPy antes do processamento e frequentemente geram matriz NumPy. Em outras palavras, para usar com eficiência muito (talvez até a maioria) do software científico / matemático de hoje baseado em Python, apenas saber como usar os tipos de sequência integrados do Python é insuficiente - também é necessário saber como usar matriz NumPy.

Os pontos sobre o tamanho e a velocidade da sequência são particularmente importantes na computação científica.

### **NumPy Matriz(Array)**

NumPy é um pacote para computação científica que tem suporte para um poderoso objeto de matriz N-dimensional. Antes de usar o NumPy, você precisa instalá-lo.

NumPy fornece uma matriz multidimensional de números (que na verdade é um objeto). Vamos dar um exemplo:

```
import numpy as np
a = np.array([1, 2, 3])
print(a)
print(type(a))
```



Observe a saída:

```
import numpy as np
a = np.array([1, 2, 3])
print(a)
print(type(a))

[1 2 3]
<class 'numpy.ndarray'>
```

Como podemos observar, a classe de matrizdo NumPy é chamada ndarray.

### Como criar uma matrizNumPy?

Existem várias maneiras de criar arrays NumPy.

### Matriz de inteiros, flutuantes (float) e números complexos

```
import numpy as np

A = np.array([[1, 2, 3], [3, 4, 5]])
print(A)

A = np.array([[1.1, 2, 3], [3, 4, 5]]) # Matrizde floats
print(A)

A = np.array([[1, 2, 3], [3, 4, 5]], dtype = complex)
# Matrizde números complexos
print(A)
```

Quando você executa o programa, a saída será:

```
import numpy as np

A = np.array([[1, 2, 3], [3, 4, 5]])
print(A)

A = np.array([[1.1, 2, 3], [3, 4, 5]]) # Array de floats
print(A)

A = np.array([[1, 2, 3], [3, 4, 5]], dtype = complex) # Array de numeros complexos
print(A)
```

[[1 2 3]  
[3 4 5]]  
[[1.1 2. 3. ]  
[3. 4. 5. ]]  
[[1.+0.j 2.+0.j 3.+0.j]  
[3.+0.j 4.+0.j 5.+0.j]]

## Matriz de zeros e uns

```
import numpy as np

zeros_matriz= np.zeros( (2, 3) )
print(zeros_array)

uns_matriz= np.ones( (1, 5), dtype=np.int32 ) #
especificando o dtype
print(uns_array)
```

Aqui, especificamos dtype para 32 bits (4 bytes). A saída será:

```
import numpy as np

zeros_array = np.zeros( (2, 3) )
print(zeros_array)

uns_array = np.ones( (1, 5), dtype=np.int32 ) # especificando o dtype
print(uns_array) # Output: [[1 1 1 1 1]]
```

[[0. 0. 0.]  
[0. 0. 0.]]  
[[1 1 1 1 1]]

## Usando arange () e shape ()

```
import numpy as np

A = np.arange(4)
print('A =', A)

B = np.arange(12).reshape(2, 6)
print('B =', B)
```

A saída será:

```
import numpy as np

A = np.arange(4)
print('A =', A)

B = np.arange(12).reshape(2, 6)
print('B =', B)

A = [0 1 2 3]
B = [[ 0  1  2  3  4  5]
     [ 6  7  8  9 10 11]]
```

## Operações de Matrizes

Acima, demos a você 3 exemplos: adição de duas matriz, multiplicação de duas matriz e transposição de uma array. Usamos listas aninhadas antes para escrever esses programas. Vamos ver como podemos fazer a mesma tarefa usando o matrizNumPy.

### Adição de duas matrizes

Usamos o operar + para adicionar elementos correspondentes de duas matrizes NumPy.

```
import numpy as np

A = np.array([[2, 4], [5, -6]])
B = np.array([[9, -3], [3, 6]])
C = A + B      # adição dos elementos-matrizes
print(C)
```

Após a execução a saída será:

```
import numpy as np

A = np.array([[2, 4], [5, -6]])
B = np.array([[9, -3], [3, 6]])
C = A + B      # adição dos elementos-matrizes
print(C)
```

```
[[11  1]
 [ 8  0]]
```

### Multiplicação de duas matrizes

Para multiplicar duas matrizes, usamos o método dot().

Obs.: \* (asterisco) é usado para multiplicação de matriz (multiplicação de elementos correspondentes de duas matrizes) e não para multiplicação de matrizes.

```
import numpy as np

A = np.array([[3, 6, 7], [5, -3, 0]])
B = np.array([[1, 1], [2, 1], [3, -3]])
C = A.dot(B)
print(C)
```

A saída será:

```
import numpy as np

A = np.array([[3, 6, 7], [5, -3, 0]])
B = np.array([[1, 1], [2, 1], [3, -3]])
C = A.dot(B)
print(C)
```

```
[[ 36 -12]
 [ -1   2]]
```

Transpor uma matriz

Usamos ***numpy.transpose*** para calcular a transposição de uma matriz.

```
import numpy as np

A = np.array([[1, 1], [2, 1], [3, -3]])
print(A.transpose())
```

A saída será:

```
import numpy as np

A = np.array([[1, 1], [2, 1], [3, -3]])
print(A.transpose())
```

```
[[ 1  2  3]
 [ 1  1 -3]]
```

Como é possível observar, o NumPy tornou na tarefa de transposição em algo fácil de executar.

## Acessando elementos de uma matriz: colunas e linhas

**Acessando elementos** - semelhante as listas (python lists), podemos acessar os elementos da matriz usando seu índice (index). Vamos começar com uma matriz NumPy unidimensional.

```
import numpy as np
A = np.array([2, 4, 6, 8, 10])

print("A[0] =", A[0])      # Primeiro elemento
print("A[2] =", A[2])      # terceiro elemento
print("A[-1] =", A[-1])    # Ultimo elemento
```

Quando você executa o programa, a saída será:

```
import numpy as np
A = np.array([2, 4, 6, 8, 10])

print("A[0] =", A[0])      # Primeiro elemento
print("A[2] =", A[2])      # terceiro elemento
print("A[-1] =", A[-1])    # Ultimo elemento
```

```
A[0] = 2
A[2] = 6
A[-1] = 10
```

Agora, vamos ver como podemos acessar os elementos de uma matriz bidimensional (que é basicamente uma matriz).

```
import numpy as np

A = np.array([[1, 4, 5, 12],
              [-5, 8, 9, 0],
              [-6, 7, 11, 19]])

# Primeiro elemento da primeira linha
print("A[0][0] =", A[0][0])
```

```
# Terceiro elemento da segunda linha
print("A[1][2] =", A[1][2])

# Ultimo elemento da ultima linha
print("A[-1][-1] =", A[-1][-1])
```

Quando executamos o programa, a saída será:

```
import numpy as np
A = np.array([2, 4, 6, 8, 10])

print("A[0] =", A[0])      # Primeiro elemento
print("A[2] =", A[2])      # terceiro elemento
print("A[-1] =", A[-1])    # Ultimo elemento

A[0] = 2
A[2] = 6
A[-1] = 10
```

### Acessar linhas de uma matriz

```
import numpy as np

A = np.array([[1, 4, 5, 12],
              [-5, 8, 9, 0],
              [-6, 7, 11, 19]])

print("A[0] =", A[0]) # Primeira linha
print("A[2] =", A[2]) # Segunda linha
print("A[-1] =", A[-1]) # Ultima linha (ou terceira
linha)
```

Quando executamos o programa, a saída será:

```
import numpy as np

A = np.array([[1, 4, 5, 12],
              [-5, 8, 9, 0],
              [-6, 7, 11, 19]])

print("A[0] =", A[0]) # Primeira linha
print("A[2] =", A[2]) # Segunda linha
print("A[-1] =", A[-1]) # Última linha (ou terceira linha)

A[0] = [ 1  4  5 12]
A[2] = [-6  7 11 19]
A[-1] = [-6  7 11 19]
```

### Acessar colunas de uma matriz

```
import numpy as np

A = np.array([[1, 4, 5, 12],
              [-5, 8, 9, 0],
              [-6, 7, 11, 19]])

print("A[:,0] =", A[:,0]) # Primeira Coluna
print("A[:,3] =", A[:,3]) # Quarta coluna
print("A[:, -1] =", A[:, -1]) # Última coluna (ou quarta
coluna)
```

Quando executamos o programa, a saída será:

```
import numpy as np

A = np.array([[1, 4, 5, 12],
              [-5, 8, 9, 0],
              [-6, 7, 11, 19]])

print("A[:,0] =", A[:,0]) # Primeira Coluna
print("A[:,3] =", A[:,3]) # Quarta coluna
print("A[:, -1] =", A[:, -1]) # Última coluna (ou quarta coluna)

A[:,0] = [ 1 -5 -6]
A[:,3] = [12  0 19]
A[:, -1] = [12  0 19]
```



## Fatiamento (slicing) de uma matriz np

O fatiamento de uma matriz NumPy unidimensional é semelhante a uma lista.

Vamos observar o código abaixo:

```
import numpy as np
letters = np.array([1, 3, 5, 7, 9, 7, 5])

# do terceiro ao quinto elementos
print(letters[2:5])

# do primeiro ao quarto elementos
print(letters[:-5])

# do sexto ao ultimo elementos
print(letters[5:])

# do primeiro ao ultimo elementos
print(letters[:])

# revertendo a lista (invertendo a busca - do ultimo ao primeiro elementos)
print(letters[::-1])
```

Ao executarmos o código a saída será:

```
import numpy as np
letters = np.array([1, 3, 5, 7, 9, 7, 5])

# do terceiro ao quinto elementos
print(letters[2:5])

# do primeiro ao quarto elementos
print(letters[:-5])

# do sexto ao ultimo elementos
print(letters[5:])

# do primeiro ao ultimo elementos
print(letters[:])

# revertendo a lista (invertendo a busca - do ultimo ao primeiro elementos)
print(letters[::-1])

[5 7 9]
[1 3]
[7 5]
[1 3 5 7 9 7 5]
[5 7 9 7 5 3 1]
```

Agora, vamos entender como podemos fatiar (slice) uma matriz. Observe o código abaixo:

```
import numpy as np

A = np.array([[1, 4, 5, 12, 14],
              [-5, 8, 9, 0, 17],
              [-6, 7, 11, 19, 21]])

print(A[:2, :4]) # duas linhas, quatro colunas

print(A[:1,]) # primeira linha, todas as colunas

print(A[:,2]) # todas as linhas, segunda coluna

print(A[:, 2:5]) # todas as linhas, da terceira a
quinta colunas
```

A saída será:

```
import numpy as np

A = np.array([[1, 4, 5, 12, 14],
              [-5, 8, 9, 0, 17],
              [-6, 7, 11, 19, 21]])

print(A[:2, :4]) # duas linhas, quatro colunas
|
print(A[:1,]) # primeira linha, todas as colunas

print(A[:,2]) # todas as linhas, segunda coluna

print(A[:, 2:5]) # todas as linhas, da terceira a quinta colunas

[[ 1  4  5 12]
 [-5  8  9  0]]
[[ 1  4  5 12 14]]
[ 5  9 11]
[[ 5 12 14]
 [ 9  0 17]
 [11 19 21]]

'Output:\n[[ 5 12 14]\n [ 9  0 17]\n [11 19 21]]\n'
```

O uso de NumPy (em vez de listas aninhadas) torna muito mais fácil trabalhar com matrizes.

## Pandas

### Explorando Dados com Dataframes

Pandas é uma biblioteca Python de código aberto que fornece análise e manipulação de dados em **programação Python**.

É uma biblioteca de excelente resultados quando o tema é representação de dados, filtragem e programação estatística. A peça mais importante no pandas é o DataFrame, onde é possível construir e armazenar conjuntos de dados com diferentes origens.

Se a distribuição Anaconda estiver em uso não é necessário nenhuma instalação. Caso contrario, é possível instalar a biblioteca Pandas através da instrução: `pip install pandas`

```
pip install pandas
```

### Ler arquivo Excel

É possível ler de um arquivo do Excel usando o método `read_excel()` do pandas. Para isso, você precisa importar mais um módulo chamado `xlrd`. Caso não esteja instalado, é possível instalá-lo através do comando `pip` abaixo:

```
pip install xlrd
```

1. O exemplo abaixo demonstra como ler em uma planilha do Excel:
2. Vamos carregar uma planilha excel dentro de nosso programa Python da seguinte forma:
3. Importar o módulo pandas conforme indicado na instrução abaixo:

```
import pandas
```

Passaremos o nome do arquivo Excel e o número da planilha da qual precisamos ler os dados para o método `read_excel()`.

```
pandas.read_excel('nomeDoArquivo.xlsx', 'Sheet1')
```

O snippet acima irá gerar a seguinte saída:

```
import pandas
```

```
pandas.read_excel('Dataset.xlsx', 'Planilha1')
```

Python Data Analytics	
0	Numpy
1	Pandas
2	Matplotlib
3	Seaborn
4	Machine Learning

Se você verificar o tipo de saída usando a palavra-chave type, terá o seguinte resultado:

```
<class 'pandas.core.frame.DataFrame'>
```

É chamado de **DataFrame** ! Essa é a unidade básica de pandas com a qual vamos lidar.

O DataFrame é uma estrutura rotulada de 2 dimensões, onde podemos armazenar dados de diferentes tipos. DataFrame é semelhante a uma tabela SQL ou uma planilha Excel.

### Importar arquivo CSV

Para ler um arquivo CSV, você pode usar o método read\_csv () do pandas.

Importe o módulo pandas:

```
import pandas
```

Agora chame o método `read_csv()` da seguinte maneira:

```
pandas.read_csv('Quadrinhos.csv')
```

O código irá gerar o seguinte DataFrame:

```
pandas.read_csv("Quadrinhos.csv")
```

	Name	Publiser
0	Spider-Man	Marvel Comics
1	Super-Man	DC Comics
2	Iron-Man	Marvel Comics
3	Wonder Woman	DC Comics
4	Batman	DC Comics

### Ler arquivo de texto

Também podemos usar o método `read_csv` do pandas para ler um arquivo de texto; considere o seguinte exemplo:

```
import pandas
```

```
pandas.read_csv('quadrinhos.txt')
```

A saída do código acima será:

```
pandas.read_csv('quadrinhos.txt')
```

	Name	Publiser
0	Spider-Man	Marvel Comics
1	Super-Man	DC Comics
2	Iron-Man	Marvel Comics
3	Wonder Woman	DC Comics
4	Batman	DC Comics

O Pandas trata o arquivo como um arquivo CSV porque temos elementos separados por vírgulas. O arquivo também pode usar outro delimitador, como ponto-e-vírgula, tabulação etc.

Suponha que temos um delimitador de tabulação e o arquivo se parece com isto:

### Selecionando as linhas por valor

Primeiro, criaremos um DataFrame a partir do qual selecionaremos as linhas.

Para criar um DataFrame, considere o código abaixo:

```
import pandas

frame_data = {'nome': ['Berry', 'Kara', 'Clark'],
              'idade': [28, 32, 52], 'função': ['Office Boy',
              'Enfermeira', 'Ascensorista']}

df = pandas.DataFrame(frame_data)
```

Neste código, criamos um DataFrame com três colunas e três linhas usando o método DataFrame () dos pandas. O resultado será o seguinte:

```
import pandas

frame_data = {'nome': ['Berry', 'Kara', 'Clark'], 'idade': [28, 32, 52], 'função': ['Office Boy', 'Enfermeira', 'Ascensorista']}

df = pandas.DataFrame(frame_data)
```

	nome	idade	função
0	Berry	28	Office Boy
1	Kara	32	Enfermeira
2	Clark	52	Ascensorista

Para selecionar uma linha com base no valor, execute a seguinte instrução:

```
df.loc[df['nome'] == 'Kara']
```

df.loc [] ou DataFrame.loc [] é uma matrizbooleana que você pode usar para acessar linhas ou colunas por valores ou rótulos. No código acima, ele selecionará linhas em que o nome seja igual a Jason.

O resultado será:

```
df.loc[df['nome'] == 'Kara']
```

	nome	idade	função
1	Kara	32	Enfermeira

### Selecionar linha por índice

Para selecionar uma linha por seu índice, podemos usar o operador Slicing (:) ou o matriz `df.loc []`.

Considere o código abaixo:

```
import pandas

frame_data = {'nome': ['Berry', 'Kara', 'Clark'],
              'idade': [28, 32, 52], 'função': ['Office Boy',
              'Enfermeira', 'Ascensorista']}

df = pandas.DataFrame(frame_data)
```

Criamos um DataFrame. Agora vamos acessar uma linha usando `df.loc []`:

```
df.loc[1]
```

```
df.loc[1]
nome      Kara
idade      32
função    Enfermeira
Name: 1, dtype: object
```

Como você pode ver, uma linha é obtida. Podemos fazer o mesmo usando o operador de fatiamento da seguinte maneira:

```
df[1:2]
```

```
df[1:2]
```

	nome	idade	função
1	Kara	32	Enfermeira

### Aplicar uma função a colunas / linhas

Para aplicar uma função em uma coluna ou linha, você pode usar o método `apply()` de `DataFrame`.

Considere o seguinte exemplo:

```
frame_data = {'A': [1, 2, 3], 'B': [18, 20, 22], 'C': [54, 12, 13]}

df = pandas.DataFrame(frame_data)
```

Criamos um `DataFrame` e adicionamos valores do tipo inteiro nas linhas. Para aplicar uma função, por exemplo, raiz quadrada nos valores, importaremos o módulo **numpy** para usar a função `sqrt` a partir dele, desta forma:

```
import numpy as np

df.apply(np.sqrt)
```

A saída será:

```
import numpy as np
df.apply(np.sqrt)
```

	A	B	C
0	1.000000	4.242641	7.348469
1	1.414214	4.472136	3.464102
2	1.732051	4.690416	3.605551

Para aplicar a função de soma, o código será:



```
df.apply(np.sum)
```

```
df.apply(np.sum)
```

```
A      6
B     60
C     79
dtype: int64
```

Para aplicar a função a uma coluna específica, você pode especificar a coluna assim:

```
df['A'].apply(np.sqrt)
```

```
df['A'].apply(np.sqrt)
```

```
0      1.000000
1      1.414214
2      1.732051
Name: A, dtype: float64
```

### Classificar valores / classificar por coluna

Para classificar valores em um DataFrame, use o método `sort_values()` do DataFrame.

Crie um DataFrame com valores inteiros:

```
frame_data = {'A': [23, 12, 30], 'B': [18, 20, 22],
              'C': [54, 112, 13]}

df = pandas.DataFrame(frame_data)
```

Agora, para classificar os valores:

```
df.sort_values(by=['A'])
```

A saída será:

```
df.sort_values(by=['A'])
```

	A	B	C
1	12	20	112
0	23	18	54
2	30	22	13

No código acima, os valores são classificados pela coluna A. Para classificar por várias colunas; o código será:

```
df.sort_values(by=['A', 'B'])
```

```
df.sort_values(by=['A', 'B'])
```

	A	B	C
1	12	20	112
0	23	18	54
2	30	22	13

Se quiser classificar em ordem decrescente, defina os atribuídos crescentes de **set\_values** como False da seguinte maneira:

```
df.sort_values(by=['A'], ascending=False)
```

A saída será

```
df.sort_values(by=['A'], ascending=False)
```

	A	B	C
2	30	22	13
0	23	18	54
1	12	20	112

## Eliminar / remover duplicados

Para eliminar linhas duplicadas de um DataFrame, use o método ***drop\_duplicates()*** do DataFrame.

Considere o seguinte exemplo:

```
frame_data = {'nome': ['Berry', 'Kara', 'Clark', 'Berry'], 'idade': [28, 32, 52, 28], 'função': ['Office Boy', 'Enfermeira', 'Ascensorista', 'Office Boy']}

df = pandas.DataFrame(frame_data)
```

```
frame_data = {'nome': ['Berry', 'Kara', 'Clark', 'Berry'], 'idade': [28, 32, 52, 28], 'função': ['Office Boy', 'Enfermeira', 'Ascensorista', 'Office Boy']}

df = pandas.DataFrame(frame_data)
```

df

	nome	idade	função
0	Berry	28	Office Boy
1	Kara	32	Enfermeira
2	Clark	52	Ascensorista
3	Berry	28	Office Boy

Acima, criamos um DataFrame com uma linha duplicada. Para verificar se linhas duplicadas estão presentes no DataFrame, use o método ***df.duplicated()*** do DataFrame.

```
df.duplicated()
```

A saída será:

```
df.duplicated()
```

```
0    False
1    False
2    False
3     True
dtype: bool
```

Pode-se ver que a última linha é uma duplicata. Para descartar ou remover esta linha, execute a seguinte linha de código:

```
df.drop_duplicates()
```

A saída será:

```
df.drop_duplicates()
```

	nome	idade	função
0	Berry	28	Office Boy
1	Kara	32	Enfermeira
2	Clark	52	Ascensorista

### Eliminar duplicidade por coluna

Às vezes, temos dados onde os valores das colunas são os mesmos e desejamos excluí-los. Podemos eliminar uma linha por coluna passando o nome da coluna que precisamos excluir.

Por exemplo, temos o seguinte DataFrame:

```
frame_data = {'nome': ['Berry', 'Kara', 'Clark', 'Berry'],
               'idade': [28, 32, 52, 28],
               'função': ['Office Boy', 'Enfermeira', 'Ascensorista', 'Office Boy']}

df = pandas.DataFrame(frame_data)
```

```
frame_data = {'nome': ['Berry', 'Kara', 'Clark', 'Berry'], 'idade': [28, 32, 52, 28],
              'função': ['Office Boy', 'Enfermeira', 'Ascensorista', 'Office Boy']}

df = pandas.DataFrame(frame_data)
```

df

	nome	idade	função
0	Berry	28	Office Boy
1	Kara	32	Enfermeira
2	Clark	52	Ascensorista
3	Berry	28	Office Boy

Aqui é possível observar que Berry se repete duas vezes. Se desejar remover duplicidade por coluna, basta passar o nome da coluna da seguinte forma:

```
df.drop_duplicates(['nome'])
```

A saída será:

```
df.drop_duplicates(['nome'])
```

	nome	idade	função
0	Berry	28	Office Boy
1	Kara	32	Enfermeira
2	Clark	52	Ascensorista

### Excluir uma coluna

Para excluir uma coluna ou linha inteira, podemos usar o método `drop()` do `DataFrame` especificando o nome da coluna ou linha.

Considere o seguinte exemplo:

```
df.drop(['função'], axis=1)
```

Nesta linha de código, estamos excluindo a coluna chamada 'função'. O argumento do eixo é necessário aqui. Se o valor do eixo for 1, significa que queremos excluir colunas. Se o valor do eixo for 0, significa que a linha será excluída. Em valores de eixo, 0 é para índice e 1 é para colunas.

O resultado será:

```
df.drop(['função'], axis=1)
```

	nome	idade
0	Berry	28
1	Kara	32
2	Clark	52
3	Berry	28

### Excluir linhas

Podemos usar o método `drop()` para eliminar ou deletar uma linha passando o índice da linha.

Vamos construir o seguinte DataFrame:

```
import pandas

frame_data = {'nome': ['Berry', 'Kara', 'Clark'],
              'idade': [28, 32, 52], 'função': ['Office Boy',
              'Enfermeira', 'Ascensorista']}

df = pandas.DataFrame(frame_data)
```

Para eliminar uma linha com índice 0 em que o nome é James, a idade é 18 e o trabalho é Assistant, use o seguinte código:

```
df.drop([0])
```

```
df.drop([0])
```

	nome	idade	função
1	Kara	32	Enfermeira
2	Clark	52	Ascensorista

Vamos criar um DataFrame onde os índices são os nomes:

```
frame_data = {'nome': ['Berry', 'Kara', 'Clark'],
              'idade': [28, 32, 52], 'função': ['Office Boy',
              'Enfermeira', 'Ascensorista']}

df = pandas.DataFrame(frame_data, index = ['Berry',
              'Kara', 'Clark'])
```

```
df
```

	nome	idade	função
Berry	Berry	28	Office Boy
Kara	Kara	32	Enfermeira
Clark	Clark	52	Ascensorista

Agora podemos deletar uma linha com um certo valor. Por exemplo, se quisermos excluir uma linha em que o nome é Rogers, o código será:

```
df.drop(['Kara'])
```

A saída será:

```
df.drop(['Kara'])
```

	nome	idade	função
Berry	Berry	28	Office Boy
Clark	Clark	52	Ascensorista

Você também pode excluir um intervalo de linha como:

```
df.drop(df.index[[0, 1]])
```

Isso excluirá linhas do índice 0 a 1 e apenas uma linha restante, pois nosso DataFrame é composto por 3 linhas:

```
df.drop(df.index[[0, 1]])
```

	nome	idade	função
Clark	Clark	52	Ascensorista

Se você deseja excluir a última linha do DataFrame e não sabe qual é o número total de linhas, você pode usar a indexação negativa conforme abaixo:

```
df.drop(df.index[-1])
```

-1 exclui a última linha. Da mesma forma, -2 excluirá as duas últimas linhas e assim por diante.

```
df.drop (df.index [-1] )
```

	nome	idade	função
0	Berry	28	Office Boy
1	Kara	32	Enfermeira



### Somar uma coluna

É possível usar o método `sum()` do `DataFrame` para somar os itens da coluna.

Vamos construir o seguinte `DataFrame`

```
frame_data = {'A': [23, 12, 12], 'B': [18, 18, 22],  
'C': [13, 112, 13]}  
  
df = pandas.DataFrame(frame_data)
```

Agora, para somar os itens da coluna A, use a seguinte linha de código:

```
df['A'].sum()
```

```
df['A'].sum()
```

```
47
```

Também podemos usar o método `apply()` do `DataFrame` e passar o método `sum` de `numpy` para somar os valores.

### Contagem de valores únicos

Para contar valores exclusivos em uma coluna, é possível usar o método `nunique()` do `DataFrame`.

Vamos construir o `DataFrame` como abaixo:

```
frame_data = {'A': [23, 12, 12], 'B': [18, 18, 22],
               'C': [13, 112, 13]}

df = pandas.DataFrame(frame_data)
```

Para contar os valores únicos na coluna A:

```
df['A'].unique()
```

```
df['A'].unique()
2
```

Como você pode ver, a coluna A tem apenas dois valores exclusivos 23 e 12, e outro 12 é uma duplicidade. É por isso que temos o valor 2 exibido na saída.

Se quiser contar todos os valores em uma coluna, use o método `count()` da seguinte maneira. Observe o código abaixo:

```
df['A'].count()
```

```
df['A'].count()
3
```

### Subconjunto de linhas

Para selecionar um subconjunto de um DataFrame, você pode usar os colchetes.

Por exemplo, temos um DataFrame que contém alguns inteiros. Podemos selecionar ou subdividir uma linha como esta:

```
df.[start:count]
```

O ponto inicial (`start`) será incluído no subconjunto, mas o ponto final não será incluído. Por exemplo, para selecionar três linhas começando na primeira linha, você escreverá:

```
df[0:3]
```

A saída será

```
df[0:3]
```

	A	B	C
0	23	18	13
1	12	18	112
2	12	22	13

Esse código significa começar na primeira linha, que é 0, e selecionar três linhas.

Da mesma forma, para selecionar as duas primeiras linhas, você escreverá:

```
df[0:2]
```

```
df[0:2]
```

	A	B	C
0	23	18	13
1	12	18	112

Para selecionar ou subdividir a última linha, use a indexação negativa como:

```
df[-1:]
```

```
df[-1:]
```

	A	B	C
2	12	22	13

## Gerar arquivo Excel

Para escrever um DataFrame em uma planilha Excel, podemos usar o método `to_excel()`.

Caso a distribuição Anaconda esteja em uso não é necessária nenhuma instalação; caso contrario será necessário instalar o módulo – via instrução `pip` – como sugerido abaixo. Observe:

```
pip install openpyxl
```

Vamos construir o seguinte Dataframe:

```
# usar este import somente se foi feita a instalação
# via comando pip

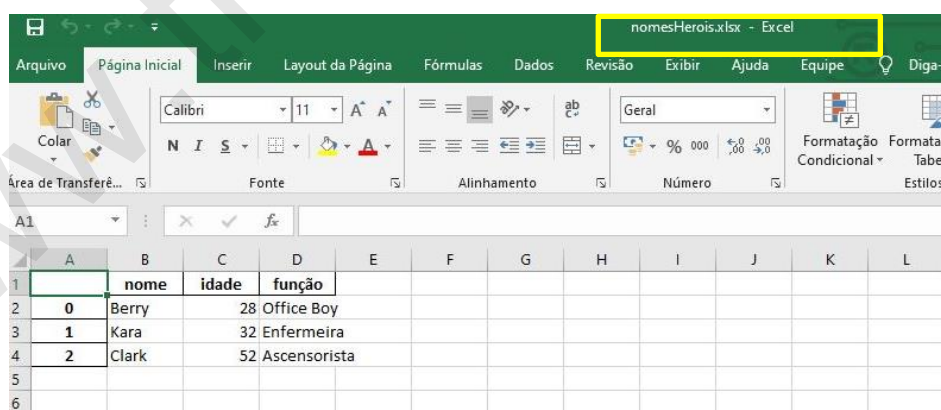
import openpyxl

frame_data = {'nome': ['Berry', 'Kara', 'Clark'],
              'idade': [28, 32, 52], 'função': ['Office Boy',
              'Enfermeira', 'Ascensorista']}

df = pandas.DataFrame(frame_data)

df.to_excel("nomesHeroes.xlsx", "Planilha1")
```

O arquivo Excel terá a seguinte aparência:



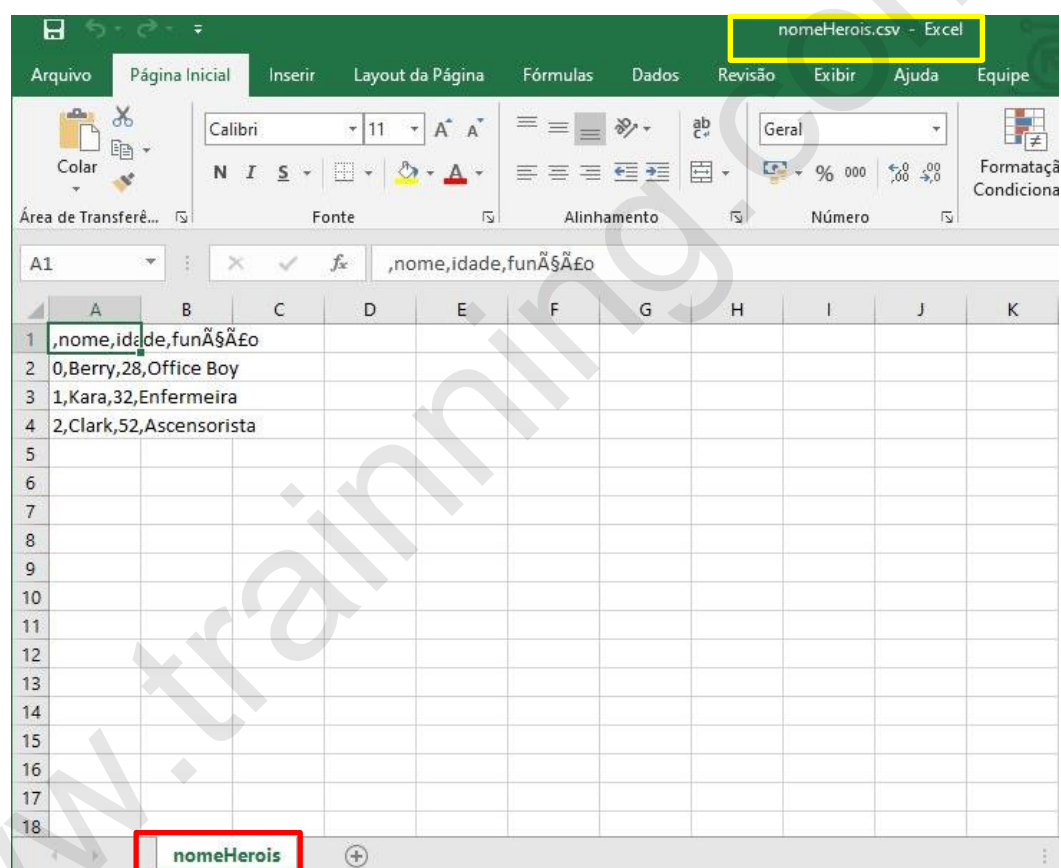
	A	B	C	D	E	F	G	H	I	J	K	L
1		nome	idade	função								
2	0	Berry	28	Office Boy								
3	1	Kara	32	Enfermeira								
4	2	Clark	52	Ascensorista								
5												
6												

## Gerar arquivo CSV

Da mesma forma, para gerar um DataFrame no formato CSV, use o método `to_csv()` como na linha de código a seguir.

```
df.to_csv("nomeHerois.csv")
```

O arquivo de saída será como o seguinte:



## Gerar arquivo Json

É possível usar o método `to_json()` do DataFrame para gerar um arquivo JSON:

```
df.to_json("nomeHerois.json")
```

Nesta linha de código, o nome do arquivo JSON é passado como um argumento. O DataFrame será armazenado no arquivo deste arquivo JSON. O arquivo terá o seguinte conteúdo:

```
1 {"nome":{"0":"Berry","1":"Kara","2":"Clark"},  
2  "idade":{"0":28,"1":32,"2":52},  
3  "fun\u00e7\u00e3o":{"0":"Office Boy","1":"Enfermeira","2":"Ascensorista"}}
```

### Gerar arquivo HTML

Também é possível gerar um arquivo HTML a partir do nosso Dataframe. Para isso, use o método `to_html()`

Implmente o código abaixo:

```
df.to_html("nomeHerois.html")
```

O arquivo resultante terá o seguinte conteúdo:

```

nomeHerois.html
1 <table border="1" class="dataframe">
2   <thead>
3     <tr style="text-align: right;">
4       <th></th>
5       <th>nome</th>
6       <th>idade</th>
7       <th>função</th>
8     </tr>
9   </thead>
10  <tbody>
11    <tr>
12      <th>0</th>
13      <td>Berry</td>
14      <td>28</td>
15      <td>Office Boy</td>
16    </tr>
17    <tr>
18      <th>1</th>
19      <td>Kara</td>
20      <td>32</td>
21      <td>Enfermeira</td>
22    </tr>
23    <tr>
24      <th>2</th>
25      <td>Clark</td>
26      <td>52</td>
27      <td>Ascensorista</td>
28    </tr>
29  </tbody>
30 </table>

```

Quando você abre o arquivo HTML no navegador, ele se parece com o seguinte:



	nome	idade	função
0	Berry	28	Office Boy
1	Kara	32	Enfermeira
2	Clark	52	Ascensorista

Trabalhar com pandas é muito fácil. É como trabalhar com planilhas do Excel! O pandas DataFrame é uma biblioteca de peças muito flexível que você pode usar.

## Pivot Table em Pandas

A maioria das pessoas certamente tem mais experiência com pivot tables no Excel. Pandas provém uma função similar chamada (apropriadamente) de `pivot_table`. Esse exercício focará em mostrar a função `pivot_table` do Pandas e como usá-la para análise de dados.

Os dados Um dos desafios de usar o `pivot_table` do pandas é se assegurar que se entenda, da melhor maneira possível, os dados disponíveis e as questões que tentamos responder utilizando pivot table. Pivot table é uma função aparentemente simples mas que pode produzir análises profundas e, principalmente, de modo rápido.

Nesse cenário, acompanharemos um pipeline de vendas (que também pode ser chamado de funil de vendas). O problema básico: alguns ciclos de venda são muito longos (por exemplo, software corporativo, equipamento pesado, entres outros) e a direção que se quer observar em mais detalhes para entender o que acontece ao longo do ano.

Questões típicas incluem:

- Quanto temos de receita potencial no pipeline (funil)?
- Quais produtos estão no pipeline (funil)?
- Quem tem quais produtos e em qual estágio?
- Quais as chances de fechar negócios até o final do ano?

Muitas empresas usam ferramentas de CRM ou outros softwares de acompanhamento do processo de vendas. Ainda que tenham ferramentas úteis para analisar os dados, inevitavelmente alguém vai exportar os dados para o Excel e usar uma pivot table para sumariá-los.

Usar a pivot table do Pandas pode ser uma boa alternativa. Algumas características:

- Mais rápida (uma vez instalada)
- Auto documentável (olhe o código e você saberá o que ela faz)
- Fácil de usar para gerar relatórios ou emails



- Mais flexível – você pode definir funções de agregação customizadas

## Acessar os dados

Primeiramente, vamos preparar o ambiente primeiro.

Se você quiser acompanhar, baixe o arquivo em Excel:

Importando as Bibliotecas

```
import pandas as pd
import numpy as np
```

A API `pivot_table` tem mudado ao longo do tempo, por isso por favor assegure-se que você tem uma versão recente do pandas (> 0.15) instalada para que tudo funcione corretamente. Esse exercício também usa o tipo de dados categórico o que também requer uma versão recente.

Ler os dados do funil de vendas no nosso DataFrame:

```
df = pd.read_excel("C:/Users/SeuUsuario/caminhodasuapa
sta/Analise Pivot Tables/sales-funnel.xlsx")
df.head()
```

A	N	F	M	Pr	Q	F	S
ccount	ame	ep	anager	oduct	uantity	rice	tatus
7	T	(	D	C	1	3	p
14466	rantow-	raig	ebra	PU		0000	resented
	Barrows	Booker	Henley				
7	T	(	D	So	1	1	p
14466	rantow-	raig	ebra	ftware		0000	resented
	Barrows	Booker	Henley				
7	T	(	D	M	2	3	p
14466	rantow-	raig	ebra	aintenance		000	ending
	Barrows	Booker	Henley				
7	F	(	D	C	1	3	d
37550	ritsch,	raig	ebra	PU		5000	eclined
	Russel	Booker	Henley				
	and						

Anderso							
n							
1	K	I	D	C	2	€	w
46832	iehn-	aniel	ebra	PU		5000	on
	Spinka	Hilton	Henley				

Por conveniência, vamos definir a coluna 'Status' como uma categoria e determinar a ordem em que queremos ver.

Isso não é estritamente necessário mas vai nos ajudar a manter a ordem que queremos enquanto trabalhamos na análise dos dados.

```
df["Status"] = df["Status"].astype("category")
df["Status"].cat.set_categories(["won", "pending", "pres
ented", "declined"], inplace=True)
```

### “Pivotando” os dados

Para se construir uma pivot table é importante dar uma passo de cada vez. Adicione itens e verifique cada passo para confirmar se está obtendo os resultados esperados. A pivot table mais simples precisa ter um dataframe e um índice (index). Aqui, vamos usar Name como índice.

```
pd.pivot_table(df, index=["Name"])
```

	Account	Price	Quantity
Name			
Barton LLC	740150	35000	1.000000
Fritsch, Russel and Anderson	737550	35000	1.000000
Herman LLC	141962	65000	2.000000
Jerde-Hilpert	412290	5000	2.000000
Kassulke, Ondricka and Metz	307599	7000	3.000000
Keeling LLC	688981	100000	5.000000
Kiehn-Spinka	146832	65000	2.000000
Koepp Ltd	729833	35000	2.000000
Kulas Inc	218895	25000	1.500000

<b>Purdy-Kunde</b>	163416	30000	1.000000
<b>Stokes LLC</b>	239344	7500	1.000000
<b>Trantow-Barrows</b>	714466	15000	1.333333

A maioria dos argumentos de uma pivot table aceita múltiplos valores na forma de uma lista.

```
pd.pivot_table(df, index=["Name", "Rep", "Manager"])
```

			Account	Price	Quantity
Name	Rep	Manager			
Barton LLC	John Smith	Debra Henley	740150	35000	1.000000
Fritsch, Russell and Anderson	Craig Booker	Debra Henley	737550	35000	1.000000
Hermann LLC	Cedric Moss	Fred Anderson	141962	65000	2.000000
Jerde-Hilpert	John Smith	Debra Henley	412290	5000	2.000000
Kassulke, Ondricka and Metz	Wendy Yule	Fred Anderson	307599	7000	3.000000
Keeling LLC	Wendy Yule	Fred Anderson	688981	10000	5.000000
Kiehn-Spinka	Daniel Hilton	Debra Henley	146832	65000	2.000000
Koepp Ltd	Wendy Yule	Fred Anderson	729833	35000	2.000000
Kulas Inc	Daniel Hilton	Debra Henley	218895	25000	1.500000

<b>Purdy-Kunde</b>	Cedric Moss	Fred Anderson	163416	30000	1.000000
<b>Stokes LLC</b>	Cedric Moss	Fred Anderson	239344	7500	1.000000
<b>Trantow-Barrows</b>	Craig Booker	Debra Henley	714466	15000	1.333333

Isso é interessante. O que, possivelmente, precisamos fazer é olhar por gerente (Manager) e por representante (Rep). É fácil fazer isso; basta alterar o índice.

```
pd.pivot_table(df, index=["Manager", "Rep"])
```

		Account	Price	Quantity
Manager	Rep			
Debra Henley	Craig Booker	720237.0	20000.00000	1.25000
	Daniel Hilton	194874.0	38333.33333	1.66666
	John Smith	576220.0	20000.00000	1.50000
Fred Anderson	Cedric Moss	196016.5	27500.00000	1.25000
	Wendy Yule	614061.5	44250.00000	3.00000

É possível ver que a pivot table é inteligente o bastante para começar a agregar e sumarizar os dados, agrupando os representantes com seus respectivos gerentes. Agora começamos a perceber o que uma pivot table pode fazer por nós.

Para esse fim, as colunas conta (Account) e quantidade (Quantity) não são muito úteis. Vamos removê-las, definindo explicitamente as colunas que queremos usando o campo values.

```
pd.pivot_table(df, index=["Manager", "Rep"], values=["Price"])
```

		Price
Manager	Rep	
Debra Henley	Craig Booker	20000.000000
	Daniel Hilton	38333.333333
	John Smith	20000.000000
Fred Anderson	Cedric Moss	27500.000000
	Wendy Yule	44250.000000

A coluna preço (Price) automaticamente tirou a média dos dados, mas podemos obter a contagem ou a soma. Adicioná-los é simples ao usar aggfunc e np.sum.

```
pd.pivot_table(df, index=["Manager", "Rep"], values=["Price"], aggfunc=np.sum)
```

		Price
Manager	Rep	
Debra Henley	Craig Booker	80000
	Daniel Hilton	115000
	John Smith	40000
Fred Anderson	Cedric Moss	110000
	Wendy Yule	177000

aggfunc pode receber uma lista de funções. Vamos tentar a média usando função mean do numpy e len para obter a contagem.

```
pd.pivot_table(df, index=["Manager", "Rep"], values=["Price"], aggfunc=[np.mean, len])
```

		mean	len
		Price	Price
Manager	Rep		
Debra Henley	Craig Booker	20000.000000	4
	Daniel Hilton	38333.333333	3
	John Smith	20000.000000	2
Fred Anderson	Cedric Moss	27500.000000	4
	Wendy Yule	44250.000000	4

Se quisermos ver as vendas em termos de produtos, a variável colunas (columns) nos permite definir uma ou mais colunas.

### Columns vs Values

Um dos pontos de confusão no pivot\_table é o uso de columns e values. Lembre-se, columns é opcional – columns dá uma forma adicional de segmentar os valores efetivos que te interessam. As funções de agregação se aplicam aos valores que você listar.

```
pd.pivot_table(df, index=["Manager", "Rep"], values=["Price"], columns=["Product"], aggfunc=[np.sum])
```

		sum				
		Price				
		Prod	CPU	Mainten	Mon	Soft
		uct	ance	itor	ware	
Man	Rep					
ager						
Debra Henley	Craig Booker	6500	0.0	5000.0	NaN	1000
	Daniel Hilton	1050	00.0	NaN	NaN	1000
						0.0

	John Smith	3500.0	5000.0	NaN	NaN
Fred Anderson	Cedric Moss	9500.0	5000.0	NaN	1000.0
	Wendy Yule	1650.0	7000.0	500.0	NaN

Alguns valores NaN estão presentes. Se quisermos removê-los, podemos usar `fill_value` e atribuir-lhes o valor 0.

```
In [13]:
pd.pivot_table(df, index=["Manager", "Rep"], values=["Price"], columns=["Product"], aggfunc=[np.sum], fill_value=0)
```

Out[13]:

		sum				
		Price				
		Product	CP	Maintenance	Monitors	Software
Manager	Rep					
Debra Henley	Craig Booker	650.00	5000	0	10000	
	Daniel Hilton	105.000	0	0	10000	
	John Smith	350.00	5000	0	0	
Fred Anderson	Cedric Moss	950.00	5000	0	10000	
	Wendy Yule	165.000	7000	5000	0	

Para que a análise fique mais interessante seria útil incluir a quantidade, também. Adicione `Quantity` à lista de valores.

```
pd.pivot_table(df, index=["Manager", "Rep"], values=["Price", "Quantity"], columns=["Product"], aggfunc=[np.sum], fill_value=0)
```

		sum			
		Price			
	Product	CPU	Maintenance	Monitor	Software
Manager	Rep				
Debra Henley	Craig Booker	65000	5000	0	10000
	Daniel Hilton	105000	0	0	10000
	John Smith	35000	5000	0	0
Fred Anderson	Cedric Moss	95000	5000	0	10000
	Wendy Yule	165000	7000	5000	0

O que pode chamar a atenção é que podemos mover itens para o índice para obter uma representação visual diferente. Remova Product das colunas e adicione ao índice(Index).

```
pd.pivot_table(df, index=["Manager", "Rep", "Product"], values=["Price", "Quantity"], aggfunc=[np.sum], fill_value=0)
```

			sum	
			Price	Quantity
Manager	Rep	Product		
Debra Henley	Craig Booker	CPU	65000	2
		Maintenance	5000	2
		Software	10000	1



Fred Anderson	Daniel	CPU	105000	4
	Hilton	Software	10000	1
	John	CPU	35000	1
	Smith	Maintenance	5000	2
	Cedric	CPU	95000	3
	Moss	Maintenance	5000	1
		Software	10000	1
	Wendy	CPU	165000	7
	Yule	Maintenance	7000	3
		Monitor	5000	2

Para esse conjunto de dados, essa representação faz mais sentido. Agora, e se quisermos ver os totais? `margins=True` faz isso para nós.

```
pd.pivot_table(df, index=["Manager", "Rep", "Product"], values=["Price", "Quantity"], aggfunc=[np.sum, np.mean], fill_value=0, margins=True)
```

			sum		mean	
			Price	Quantity	Price	Quantity
Manager	Rep	Product				
Debra Henley	Craig Booker	CPU	65000	2	32500	1.00000
		Maintenance	5000	2	5000	2.00000
		Software	10000	1	10000	1.00000

	Daniel Hilton	D	CPU	10	4	5	2.0
				5000		2500	00000
			Software	10000	1	10000	1.00000
	John Smith	J	CPU	35	1	3	1.0
				000		5000	00000
			Maintenance	5000	2	5000	2.00000
Fred Anderson	Fredric Moss	F	CPU	95	3	4	1.5
				000		7500	00000
			Maintenance	5000	1	5000	1.00000
	Wendy Yule		Software	10000	1	10000	1.00000
		W	CPU	16	7	8	3.5
				5000		2500	00000
		Maintenance	7000	3	7000	3.00000	
		Monitor	5000	2	5000	2.00000	
	All			52000	30	30705	1.764706

Vamos incrementar a análise e olhar o nosso pipeline ao nível de gerentes. Note como o status é ordenado com base na nossa definição prévia de categoria.

```
pd.pivot_table(df, index=["Manager", "Status"], values=["Price"], aggfunc=[np.sum], fill_value=0, margins=True)
```

		sum
		Price
Manager	Status	

<b>Debra Henley</b>	won	65000
	pending	50000
	presented	50000
	declined	70000
<b>Fred Anderson</b>	won	172000
	pending	5000
	presented	45000
	declined	65000
<b>All</b>		522000

Uma funcionalidade bem útil é a capacidade de mandar um dicionário para a `aggfunc` de forma que você possa "performar" diferentes funções em cada um dos valores selecionados. Isso tem o efeito colateral de deixar os rótulos (labels) mais limpos.

```
pd.pivot_table(df, index=["Manager", "Status"], columns=[
    "Product"], values=["Quantity", "Price"], aggfunc={"Quantity":
    len, "Price": np.sum}, fill_value=0)
```

		Price				
		Product	CPU	Maintenance	Monitor	Software
Manager	Status					
Debra Henley	won	65000	0	0	0	
	pending	40000	10000	0	0	
	presented	30000	0	0	20000	
	declined	70000	0	0	0	
Fred Anderson	won	165000	7000	0	0	
	pending	0	5000	0	0	
	presented	30000	0	5000	10000	

	declined	65000	0	0	0
--	----------	-------	---	---	---

Você pode fornecer uma lista de aggfunctions a ser aplicada a cada valor também:

```
table = pd.pivot_table(df, index=["Manager", "Status"], columns=["Product"], values=["Quantity", "Price"], aggfunc={"Quantity": len, "Price": [np.sum, np.mean]}, fill_value=0)
table
```

		Price					Quantity				
		mean					sum				
Manager	Status	Product	PU	Maintenance	Monitor	Software	PU	Maintenance	Monitor	Software	PU
		Product	PU	Maintenance	Monitor	Software	PU	Maintenance	Monitor	Software	PU
Debra Henley	on		5000				5000				0
	ending		0000	000			0000	0000			2
	resented		0000			0000	0000			0000	0
	declined		5000				0000				0
Fred Anderson	on		2500	000			6500	000			1
	ending			000				000			1
	resented		0000		000	0000	0000		000	0000	0
	declined		5000				5000				0

## Filtros avançados

Uma vez que você tenha gerado seus dados, eles estão num DataFrame que você pode filtrar usando funções padrão do DataFrame.

Caso queira ver apenas um gerente:

Você pode olhar todas as negociações (Deals) pendentes e fechadas.

```
table.query('Status == ["pending", "won"]')
```

		Price				Quantity						
		mean		sum		len						
	roduct	f PU	M aintenance	onitor	oftware	PU aintenance	onitor	oftware	PU aintenance	Ma onitor	I oftware	S
M anager												
D ebra Henley	wo		0			(		(	(	0	(	0
	n	50				5						
		00				0						
						0						
	pe		5			(		(	(	2	(	0
	nding	00	000			0	0000					
		00				0						
						0						
F red Anderson	wo		7			(		(	(	1	(	0
	n	25	000			6	000					
		00				5						
						0						
	pe		5			(		(	(	1	(	0
	nding		000				000					

Essa é uma funcionalidade poderosa do `pivot_table`, então não se esqueça que você tem toda a força do pandas a seu dispor, uma vez que tenha seus dados no formato `pivot_table` que necessita.

## GropuBy

O termo group by é muito popular para quem trabalha com base de dados. Quando temos repetições para o elemento chave e queremos fazer um resumo, um agrupamento, é esse o comando a ser utilizado. Um exemplo clássico é quando você tem os dados dos gastos feitos por clientes de uma loja e sua base contém um gasto por linha. Para obter o total gasto por cada cliente, você irá recorrer ao group by.

No Python, quando temos um dataframe da biblioteca Pandas, podemos chamar o comando pelo seu nome e utilizando a sintaxe mais clássica da linguagem que é o nome do objeto (o dataframe) seguido de group by e da função que deve ser aplicada (como a média ou a soma). Vejamos o exemplo da base com os gastos de cada cliente:

```
# carrega biblioteca pandas

import pandas as pd

# cria dataset
compras = {'Id': ['AA2930', 'AA2930', 'CC2139', 'CC2139',
                  'CC9999', 'AA2930'],
            'Data': ['2019-01-01', '2019-01-30', '2019-01-30', '2019-02-01',
                    '2019-02-20', '2019-03-15'],
            'Valor': [200, 100, 400, 150, 10, 25]}

compras = pd.DataFrame(compras, columns = ['Id', 'Data', 'Valor'])

# traz a soma de gastos de cada cliente (Id)

compras.groupby(['Id'])['Valor'].sum()

Id
AA2930    325
CC2139    550
CC9999     10
Name: Valor, dtype: int64
```

Veja que primeiro carregamos a biblioteca pandas a qual vamos nos referir com a abreviação pd, criamos um dataframe e, em seguida, somamos o campo Valor, agrupando-o por cada Id. Poderíamos ter obtido o gasto médio de cada cliente trocando mean() por sum():

```
compras.groupby(['Id'])['Valor'].mean()
```

```
Id
AA2930    108.333333
CC2139    275.000000
CC9999     10.000000
Name: Valor, dtype: float64
```

Agora, imagine que por algum motivo você queira fazer um agrupamento utilizando uma tabela com algum filtro. Por exemplo, você quer trazer o gasto médio por cliente mas sem considerar compras que para sua loja são irrisórias, como as que são abaixo de 30 reais. Neste caso, é intuitivo: você deve fazer o comando de filtro normalmente e na frente dele você deve incluir o comando de agrupamento. Veja primeiro como seria fazendo um passo por vez:

```
# cria um novo dataframe só com compras acima de 30 reais
```

```
compras_v2 = compras[compras['Valor'] > 30]
```

```
compras_v2.groupby(['Id'])['Valor'].sum()
```

```
Id
AA2930    300
CC2139    550
Name: Valor, dtype: int64
```

Para economizar linhas de código, podemos simplesmente colocar tudo em uma única linha:

```
compras[compras['Valor'] > 30].groupby(['Id'])['Valor'].sum()
```

```
Id
AA2930    300
CC2139    550
Name: Valor, dtype: int64
```

De forma análoga, você pode querer trazer somente clientes que fizeram mais de uma compra na sua loja. Ou seja, você vai fazer o group by somente com os clientes que aparecem mais de uma vez na sua base:

```
compras[compras.groupby('Id')['Valor'].transform('size') > 1].groupby(['Id'])['Valor'].sum()

Id
AA2930    325
CC2139    550
Name: Valor, dtype: int64
```

Se quiser, troque `sum()` e `mean()` por outras funções como `count()`, de acordo com a sua necessidade.

## Aprofundando Data Analysis

Prática com conjunto de dados – Problema de previsão de Empréstimo

### Descrição das variáveis

- Loan\_ID | ID único do empréstimo (loan)
- Gender | Masculino/Feminino (Male/Female)
- Married | Casado - sim ou não (Y/N)
- Dependents | Numero de dependentes
- Education | Grau escolar (Graduate/ Under Graduate)
- Self\_Employed | Auto empregado - sim ou não (Y/N)
- ApplicantIncome | Renda do aplicante
- CoapplicantIncome | Renda do co-aplicante
- LoanAmount | Montante do empréstimo (loan), em milhares
- Loan\_Amount\_Term | Prazo do empréstimo (loan), em meses
- Credit\_History | Histórico de crédito corresponde aos critérios
- Property\_Area | Localização da propriedade (Urban/Semi Urban/Rural)
- Loan\_Status | Empréstimo (loan) aprovado - sim ou não (Y/N)

```
import pandas as pd
import numpy as np
import matplotlib as plt
import matplotlib.pyplot as plt
```

```
df = pd.read_csv("C:/Users/SeuUsuario/CaminhodaSuaPasta/Arquivos-
Treino/Arquivo-Treino.csv") #Lendo o conjunto de dados em um dataframe
do Pandas
```



Depois de importar a biblioteca, leia o conjunto de dados usando a função `read_csv()`.

Depois de ler o conjunto de dados, você pode dar uma olhada nas linhas iniciais usando a função `head()`.

```
df.head(10)
```

loan_id	ender	married	dependents	education	self_Employed	applicantIncome	applicantIncome	loanAmount	loanAmount_Term	credit_History	property_Area	loan_Status
---------	-------	---------	------------	-----------	---------------	-----------------	-----------------	------------	-----------------	----------------	---------------	-------------

P001002	male	no		graduate	no	849	0	NaN	60.0	0	urban	
---------	------	----	--	----------	----	-----	---	-----	------	---	-------	--

P001003	male	yes		graduate	no	583	508.0	28.0	60.0	0	suburban	
---------	------	-----	--	----------	----	-----	-------	------	------	---	----------	--

P001005	male	yes		graduate	yes	000	0	6.0	60.0	0	urban	
---------	------	-----	--	----------	-----	-----	---	-----	------	---	-------	--

P001006	male	yes		not Graduate	no	583	358.0	20.0	60.0	0	urban	
---------	------	-----	--	--------------	----	-----	-------	------	------	---	-------	--

oa n_I D	en de r	arr ied	epe nde nts	duc atio n	elf_ Emp loye d	pplic antIn com e	oappl icantl ncom e	oan Am oun t	oan_ Amo unt_T erm	redit _His tory	rope rty_ Area	oan _St atu s
----------------	---------------	------------	-------------------	------------------	--------------------------	----------------------------	------------------------------	-----------------------	-----------------------------	-----------------------	----------------------	------------------------

P0 010 08	ale	o		rad uat e	o	000	.0	41.0	60.0	.0	rban	
-----------------	-----	---	--	-----------------	---	-----	----	------	------	----	------	--

P0 010 11	ale	es		rad uat e	es	417	196.0	67.0	60.0	.0	rban	
-----------------	-----	----	--	-----------------	----	-----	-------	------	------	----	------	--

P0 010 13	ale	es		ot Gra dua te	o	333	516.0	5.0	60.0	.0	rban	
-----------------	-----	----	--	------------------------	---	-----	-------	-----	------	----	------	--

P0 010 14	ale	es	+	rad uat e	o	036	504.0	58.0	60.0	.0	emiu rban	
-----------------	-----	----	---	-----------------	---	-----	-------	------	------	----	--------------	--

P0 010 18	ale	es		rad uat e	o	006	526.0	68.0	60.0	.0	rban	
-----------------	-----	----	--	-----------------	---	-----	-------	------	------	----	------	--

P0	ale	es		rad	o	2841	0968. 0	49.0	60.0	.0	emiu rban	
----	-----	----	--	-----	---	------	------------	------	------	----	--------------	--

```

loan_id    gender    married    dependents    education    self_employed    applicant_income    coapplicant_income    loan_amount    loan_amount_term    credit_history    property_area    loan_status

010
20
          uat
          e

```

10 linhas devem aparecer. Alternativamente, você também pode olhar para mais linhas do conjunto de dados.

Em seguida, você pode olhar para um resumo dos campos numéricos usando a função `describe()`.

```
df.describe()
```

	ApplicantIncome	CoapplicantIncome	LoanAmount	Loan_Amount_Term	Credit_History
count	614.00000	614.00000	592.000000	600.00000	564.000000
mean	5403.459283	1621.245798	146.412162	342.00000	0.842199
std	6109.041673	2926.248369	85.587325	65.12041	0.364878
min	150.00000	0.000000	9.00000	12.00000	0.000000
25%	2877.500000	0.000000	100.000000	360.00000	1.000000

	ApplicantIncome	CoapplicantIncome	LoanAmount	Loan_Amount_Term	Credit_History
5	3812.5	1188.50	128.	360.000	1.00
0%	00000	0000	000000	00	0000
7	5795.0	2297.25	168.	360.000	1.00
5%	00000	0000	000000	00	0000
max	81000.	41667.0	700.	480.000	1.00
	000000	00000	000000	00	0000

A função `describe()` fornece contagem, média, desvio padrão (STD), mínimo, quartis e máximo (Leia este artigo para atualizar-se em estatísticas básicas para compreender a distribuição de uma população).

Aqui estão algumas inferências, você pode desenhar olhando para a saída de função `describe()`:

- LoanAmount tem 22 valores faltantes (614 – 592).
- Loan\_Amount\_Term tem 14 valores faltantes (614 – 600).
- Credit\_History tem 50 valores faltantes (614 – 564).

Podemos também verificar que cerca de 84% dos candidatos têm histórico de crédito. Como? A média do campo `Credit_History` é 0,85 (Lembre-se, `Credit_History` tem um valor 1 para aqueles que têm um histórico de crédito e 0 caso contrário).

A distribuição `ApplicantIncome` parece estar em linha com a expectativa. O mesmo com a `CoapplicantIncome`.

Note por favor que podemos ter uma ideia de uma possível distorção nos dados comparando a média com a mediana, isto é, o valor de 50%.

Para os valores não-numéricos (por exemplo, localização da Propriedade, histórico de crédito etc.), podemos olhar para a distribuição de frequência para

entender se elas fazem sentido ou não. A tabela de frequências pode ser exibida pelo seguinte comando:

```
df['Property_Area'].value_counts()
Semiurban    233
Urban        202
Rural        179
Name: Property_Area, dtype: int64
```

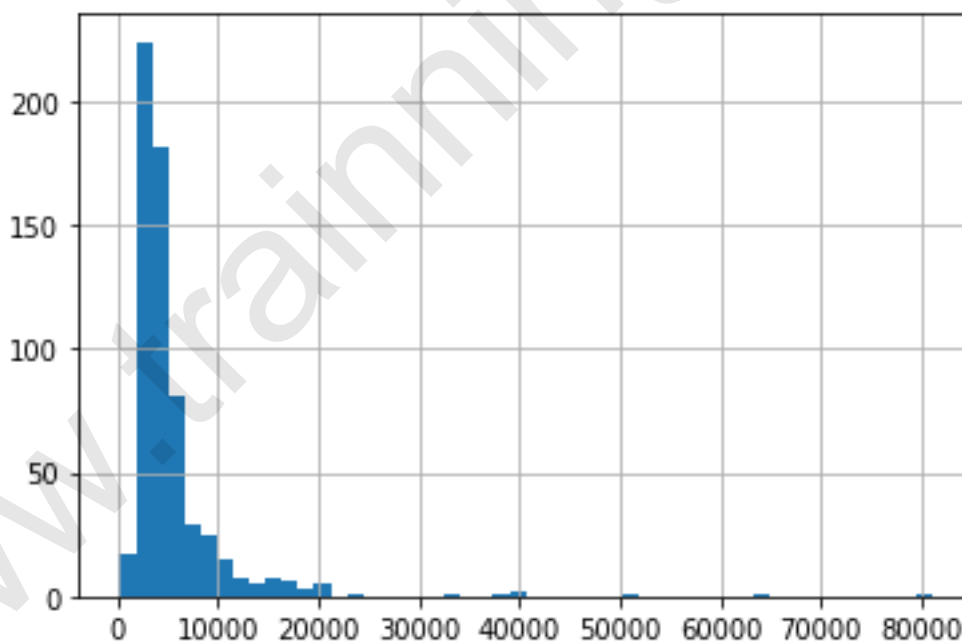
### Análise da Distribuição

Agora que estamos familiarizados com as características básicas dos dados, vamos estudar a distribuição de diversas variáveis. Vamos começar com as variáveis numéricas – ApplicantIncome e LoanAmount.

Vamos começar plotando o histograma do ApplicantIncome usando o seguinte comando:

```
df['ApplicantIncome'].hist(bins=50)

<matplotlib.axes._subplots.AxesSubplot at 0xdf8a70>
```

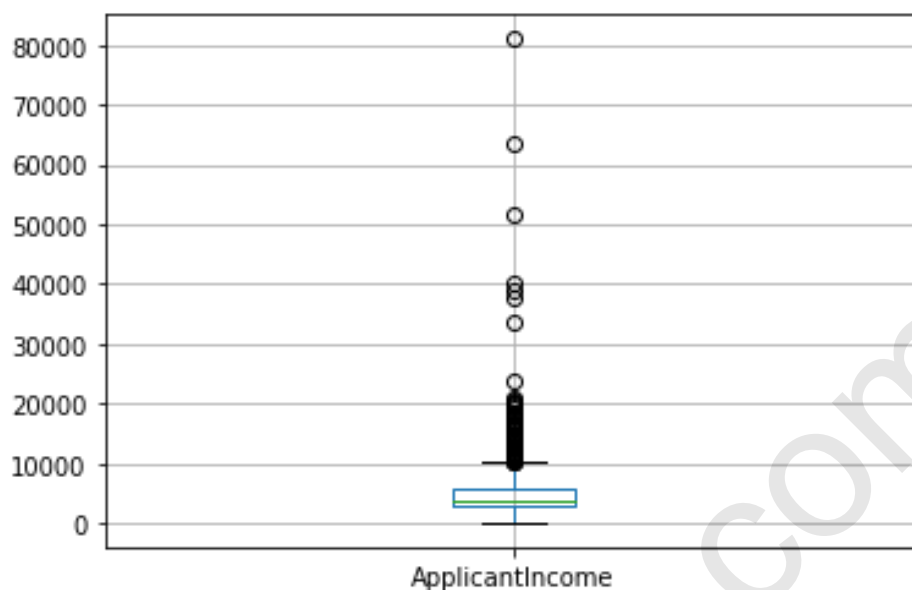


Observamos que há poucos valores extremos. Esta é também a razão pela qual 50 caixas (bins) são necessários para representar a distribuição claramente.

Em seguida, olhemos para o box plots para compreender as distribuições. Box plots para renda do aplicante pode ser traçado por:

```
df.boxplot(column='ApplicantIncome')
```

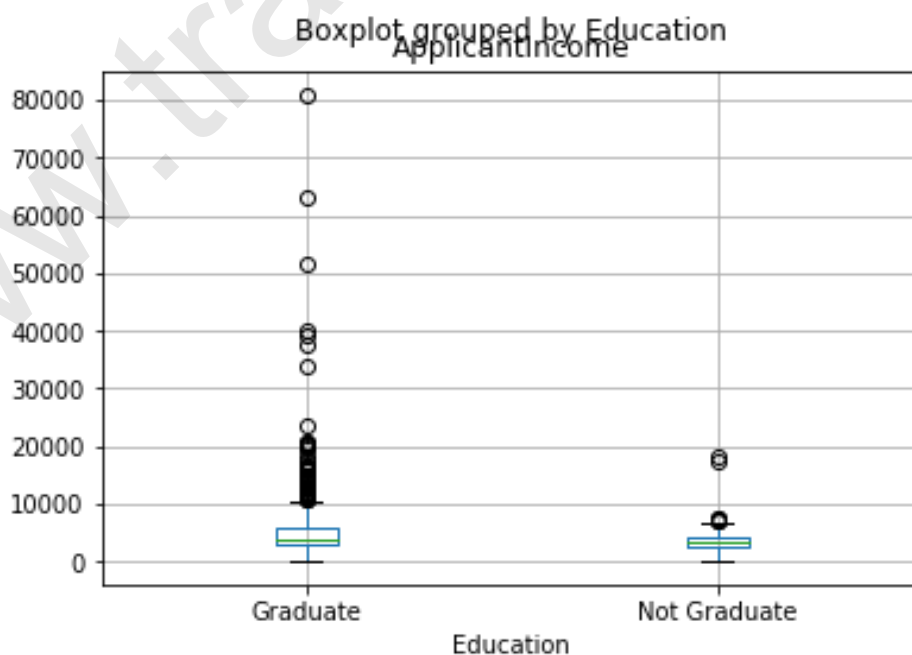
```
<matplotlib.axes._subplots.AxesSubplot at 0xad84c50>
```



Isso confirma a presença de uma grande quantidade de outliers/valores extremos. Isso pode ser atribuído à disparidade de renda na sociedade. Parte disso pode ser impulsionado pelo fato de que nós estamos olhando para as pessoas com diferentes níveis de ensino. Vamos segregá-los por Educação:

```
df.boxplot(column='ApplicantIncome', by = 'Education')
```

```
<matplotlib.axes._subplots.AxesSubplot at 0xdfe8570>
```

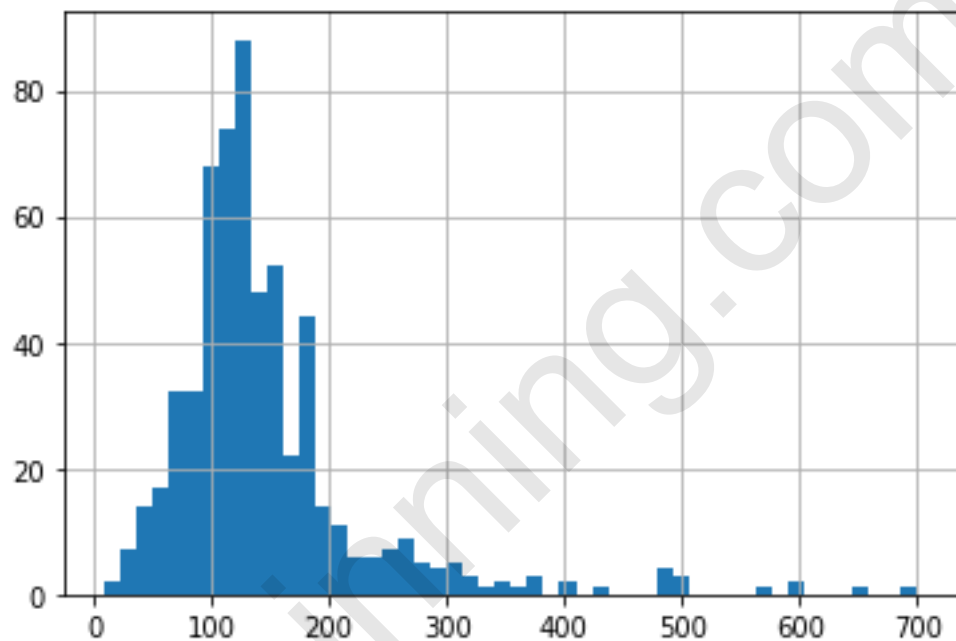


Podemos ver que não há nenhuma diferença substancial entre a renda média de pós-graduação e de não-graduados. Mas há um maior número de diplomados com rendimentos muito elevados que parecem ser os outliers.

Agora, vamos olhar para o histograma e o box plot do valor do empréstimo usando o seguinte comando:

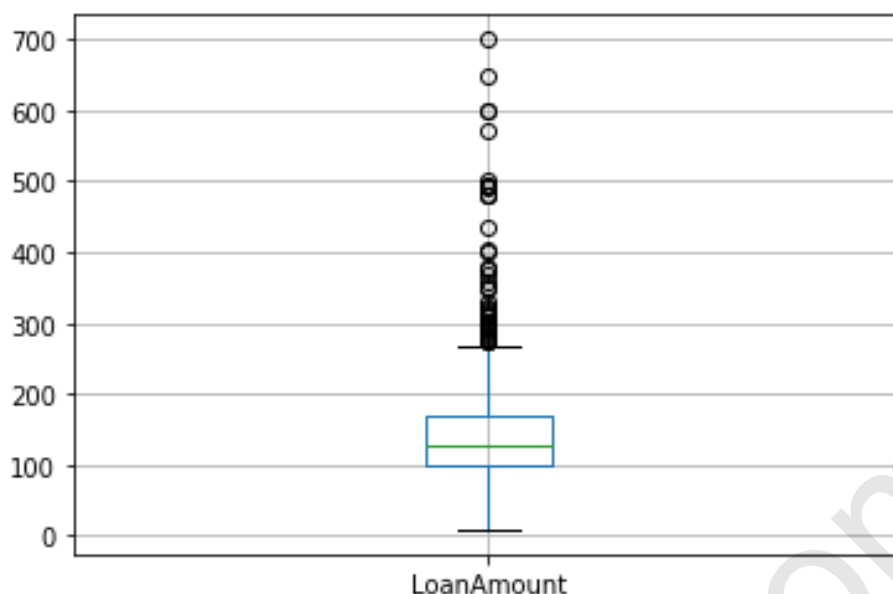
```
df['LoanAmount'].hist(bins=50)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0xe3b82d0>
```



```
df.boxplot(column='LoanAmount')
```

```
<matplotlib.axes._subplots.AxesSubplot at 0xe43d090>
```



Mais uma vez, há alguns valores extremos. Claramente, tanto ApplicantIncome e LoanAmount exigem uma certa quantidade de 'data munging'.

LoanAmount tem valores faltantes e valores extremos, enquanto ApplicantIncome tem alguns valores extremos, que exigem compreensão mais profunda. Vamos avaliar isto nas próximas seções.

### Análise de Variáveis Categóricas

Agora que entendemos as distribuições do ApplicantIncome e do LoanIncome, vamos entender as variáveis categóricas com mais detalhes. Usaremos tabela estilo pivot table do Excel e tabulação cruzada. Por exemplo, vamos olhar para as chances de conseguir um empréstimo com base no histórico de crédito.

Nota: aqui, o status de empréstimo foi codificado como 1 para Sim e 0 para Não. Assim, a média representa a probabilidade de obtenção de empréstimo.

```
temp1 = df['Credit_History'].value_counts(ascending=True)
temp2 = df.pivot_table(values='Loan_Status', index=['Credit_History'], aggfunc=lambda x: x.map({'Y':1, 'N':0}).mean())
print('Frequency Table for Credit History:')
```



```

print(temp1)
print('\nProbability of getting loan for each Credit History class:')
print(temp2)

```

Frequency Table for Credit History:

0.0      89

1.0      475

Name: Credit\_History, dtype: int64

Probability of getting loan for each Credit History class:

Credit_History	Loan_Status
0.0	0.078652
1.0	0.795789

Agora, podemos observar que temos uma tabela dinâmica semelhante ao MS Excel. Isto pode ser plotado como um gráfico de barras usando a biblioteca “matplotlib” com o seguinte código:

```

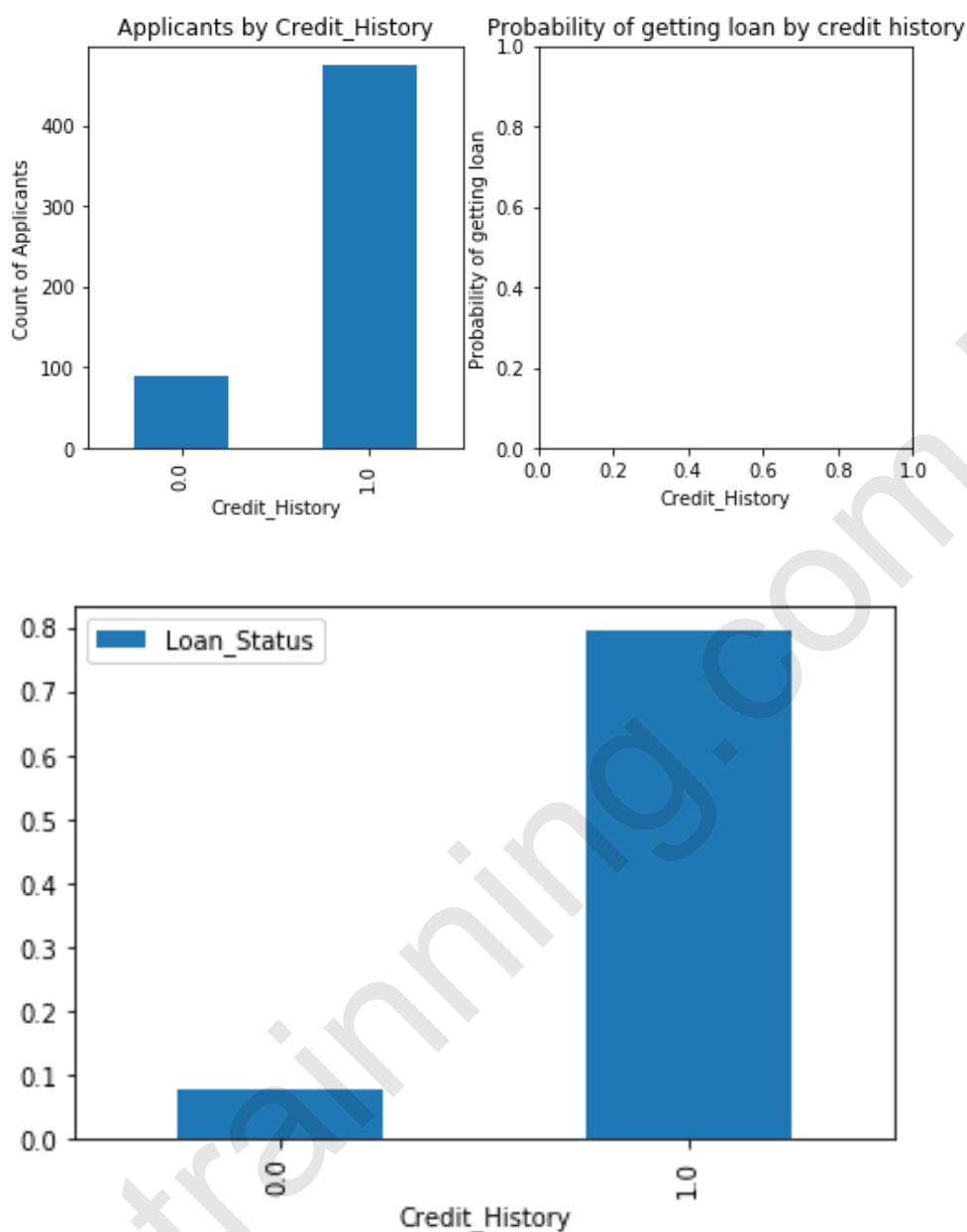
fig = plt.figure(figsize=(8,4))
ax1 = fig.add_subplot(121)
ax1.set_xlabel('Credit_History')
ax1.set_ylabel('Count of Applicants')
ax1.set_title("Applicants by Credit_History")
temp1.plot(kind='bar')
ax2 = fig.add_subplot(122)
temp2.plot(kind = 'bar')
ax2.set_xlabel('Credit_History')
ax2.set_ylabel('Probability of getting loan')
ax2.set_title("Probability of getting loan by credit history")

```

```

Text(0.5, 1.0, 'Probability of getting loan by credit history')

```



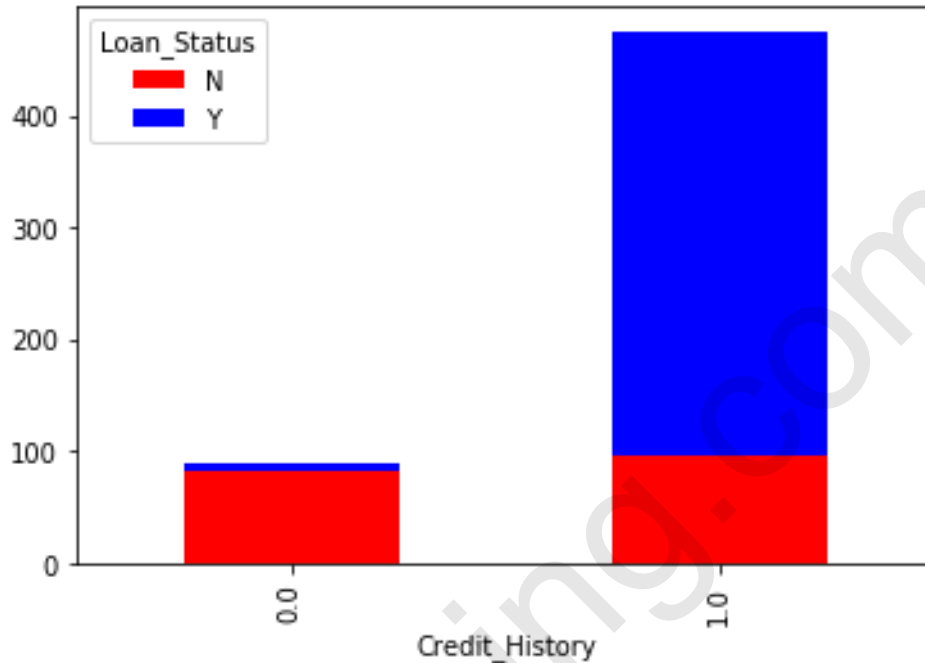
Isso mostra que a chance de conseguir um empréstimo é oito vezes se o requerente tiver um histórico de crédito válido. Pode-se traçar gráficos semelhantes por Estado civil, Profissional Autônomo, Localização de propriedade, etc.

Como alternativa, estes dois plots também podem ser visualizados por combinação em um gráfico empilhado.

Você também pode adicionar gênero na mistura (semelhante a tabela dinâmica em Excel):

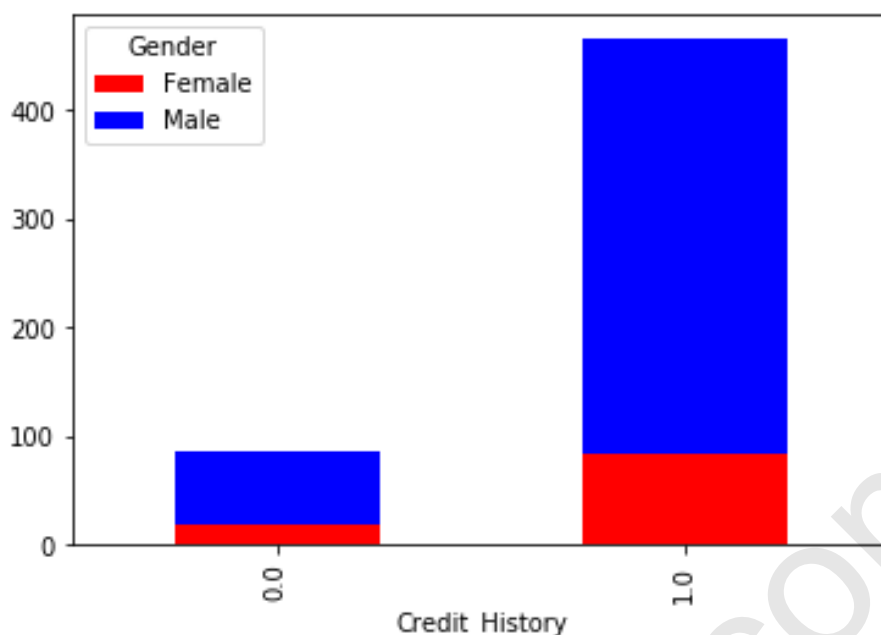
```
temp3 = pd.crosstab(df['Credit_History'],
                    df['Loan_Status'])
temp3.plot(kind='bar', stacked=True, color=['red', 'blue'], grid=False)
```

<matplotlib.axes.\_subplots.AxesSubplot at 0xe49b790>



```
temp3 = pd.crosstab(df['Credit_History'],
                    df['Gender'])
temp3.plot(kind='bar', stacked=True, color=['red', 'blue'], grid=False)
```

<matplotlib.axes.\_subplots.AxesSubplot at 0xabd9030>



Acabamos de criar dois algoritmos de classificação básicos aqui, um baseado no histórico de crédito e outro em 2 variáveis categóricas (incluindo sexo).

Em seguida, vamos explorar variáveis ApplicantIncome e LoanStatus e ainda, realizar munging dos dados e criar um conjunto de dados para a aplicação de várias técnicas de modelagem. Eu recomendo fortemente que você pegue um outro conjunto de dados e outro problema e faça uma exploração de dados independente, como exemplo, antes de continuar lendo.

### **Munging de dados com Python: Usando Pandas**

Data munging – por quê da necessidade

Durante a exploração dos dados, encontramos alguns problemas que precisam ser resolvidos antes que os dados estejam prontos para serem modelados. Este exercício é normalmente referido como “Data Munging”. Aqui estão os problemas, já estamos cientes de que:

- Há valores faltando em algumas variáveis. Devemos estimar esses valores, dependendo da quantidade de valores faltantes e da importância esperada das variáveis.
- Quando se olha para as distribuições, vê-se que ApplicantIncome e LoanAmount parecem conter valores extremos em cada ponta.

Embora intuitivamente eles possam fazer sentido, devem ser tratados de forma adequada.

Além desses problemas com campos numéricos, devemos também olhar para os campos não-numéricos. Ou seja, Gênero, Property\_Tree, Estado civil, Educação e Dependentes, para ver se eles contêm alguma informação útil.

### Verifique os Valores Ausentes no Conjunto de Dados

Vejam os valores ausentes em todas as variáveis, porque a maioria dos modelos não funciona com dados ausentes. E mesmo se funcionarem, isso ajuda mais frequentemente. Então, vamos verificar o número de nulos e de NaNs no conjunto de dados.

O comando abaixo deve nos dizer o número de valores faltantes em cada coluna, pois `isnull()` retorna 1 se o valor é nulo.

*# Munging de dados com Python: Usando Pandas*

```
df.apply(lambda x: sum(x.isnull()), axis=0)
```

```
Loan_ID      0
Gender       13
Married      3
Dependents   15
Education    0
Self_Employed 32
ApplicantIncome 0
CoapplicantIncome 0
LoanAmount   22
Loan_Amount_Term 14
Credit_History 50
Property_Area 0
Loan_Status  0
dtype: int64
```

Embora os valores ausentes não sejam muito elevados em número, muitas variáveis têm valores ausentes e cada um deles deve ser estimado e adicionado aos dados.

Nota: Lembre-se que os valores ausentes podem nem sempre ser NaNs. Por exemplo, se o valor de prazo do empréstimo for 0, isso faz sentido? Ou você

considera que está ausente? Podemos supor que a resposta seja que é ausente; está certo. Assim, devemos verificar se há valores que não fazem sentido.

Como preencher valores ausentes no LoanAmount, valor do empréstimo?

Existem inúmeras maneiras de preencher os valores ausentes dos empréstimos – a mais simples é substituí-los pela média, o que pode ser feito pelo seguinte código:

```
df['LoanAmount'].fillna(df['LoanAmount'].mean(), inplace=True)
df['LoanAmount']
```

0	146.412162
1	128.000000
2	66.000000
3	120.000000
4	141.000000
	...
609	71.000000
610	40.000000
611	253.000000
612	187.000000
613	133.000000

Name: LoanAmount, Length: 614, dtype: float64

O outro extremo poderia ser a construção de um modelo de aprendizagem supervisionada para prever o montante do empréstimo com base em outras variáveis e, em seguida, utilizar a idade, juntamente com outras variáveis, para prever a sobrevida.

Como o objetivo agora é trazer as etapas do munging de dados, estamos usando uma abordagem que se encontra em algum ponto entre esses 2 extremos. A hipótese principal é que se o nível de escolaridade ou o trabalho por conta própria podem se combinar para dar uma boa estimativa do montante do empréstimo.

Como dissemos anteriormente, Self\_Employed tem alguns valores ausentes. Vamos olhar para a tabela de frequência:

```
df['Self_Employed'].value_counts()
```

```
No      500
```

```
Yes      82
```

```
Name: Self_Employed, dtype: int64
```

Como 86% dos valores são “não”, é seguro calcular os valores ausentes como “Não” pois há uma alta probabilidade de sucesso. Isso pode ser feito usando o seguinte código:

```
df['Self_Employed'].fillna('No', inplace=True)
```

Agora, vamos criar uma tabela dinâmica que nos forneça valores médios para todos os grupos de valores exclusivos de características de Self\_Employed e Educação. Em seguida, vamos definir uma função, que retorna os valores dessas células e aplicá-los para preencher os valores ausentes de valor do empréstimo:

```
table = df.pivot_table(values='LoanAmount', index='Self_Employed', columns='Education', aggfunc=np.median)
```

```
# Define a função que retorna o valor da tabela pivot
def fage(x):
    return table.loc[x['Self_Employed'], x['Education']]
# Substitui valores faltantes
df['LoanAmount'].fillna(df[df['LoanAmount'].isnull()
]).apply(fage, axis=1, inplace=True)
```

Esta é uma boa maneira de imputar os valores ausentes do montante do empréstimo.

### **Como tratar valores extremos na distribuição de valor do empréstimo (LoanAmount) e Solicitante de renda (ApplicantIncome)?**

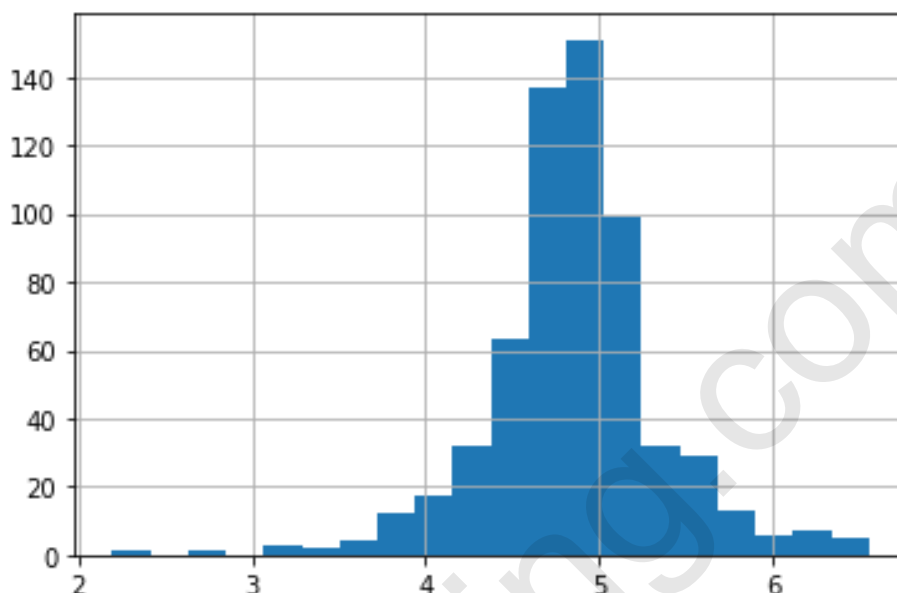
Vamos analisar os valores do empréstimo em primeiro lugar. Valores extremos são possíveis, ou seja, algumas pessoas podem solicitar empréstimos de alto valor devido a necessidades específicas. Então, ao invés de tratá-los como valores atípicos, vamos tentar uma transformação log para anular os seus efeitos:

```
df['LoanAmount_log'] = np.log(df['LoanAmount'])
```

Olhando novamente para o Histograma:

```
df['LoanAmount_log'].hist(bins=20)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0xac24890>
```



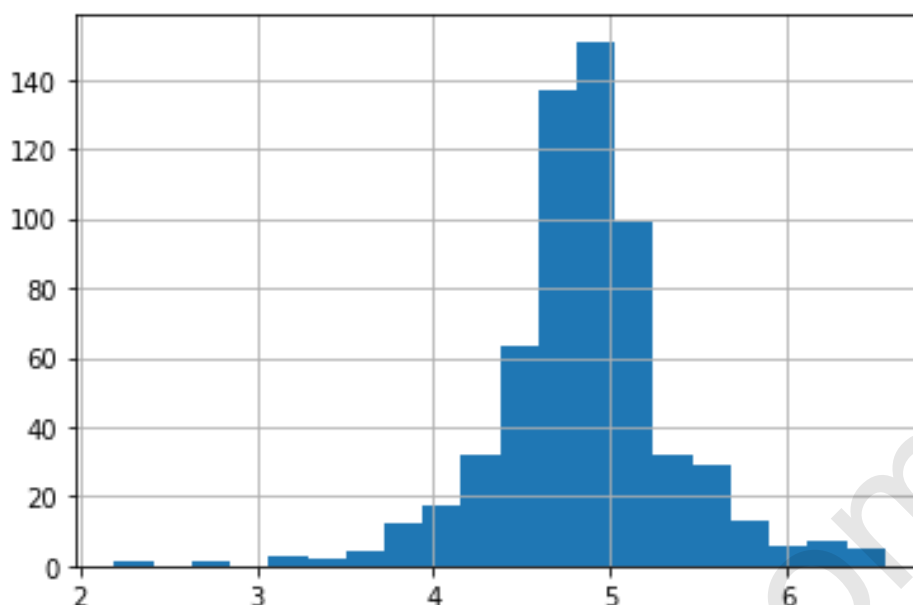
Agora, a distribuição parece muito mais próxima ao normal e os efeitos dos valores extremos foi significativamente diminuído.

Chegando à Renda do Solicitante (ApplicantIncome). Uma intuição pode ser que alguns candidatos tenham renda baixa, mas apoio de co-candidatos fortes. Assim, pode ser uma boa ideia combinar ambos os rendimentos como renda total e fazer a mesma transformação log.

```
df['TotalIncome'] = df['ApplicantIncome'] + df['CoapplicantIncome']
df['TotalIncome_log'] = np.log(df['TotalIncome'])
df['LoanAmount_log'].hist(bins=20)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0xe4e6ad0>
```





Vemos que a distribuição ficou muito melhor que antes. Vou deixar para você imputar os valores ausentes para Sexo, Casado, Dependentes, Prazo do valor do empréstimo e Histórico de crédito. Além disso, encorajo você a pensar sobre possíveis informações adicionais que podem ser derivadas a partir dos dados. Por exemplo, criar uma coluna para  $\text{LoanAmount} / \text{TotalIncome}$  pode fazer sentido, uma vez que dá uma ideia se o candidato tem condições adequadas para pagar seu empréstimo.

## Tratando grande volumes de dados

Neste passo-a-passo, você aprenderá a trabalhar com grandes arquivos Excel no Pandas, focando na leitura e análise de um arquivo xls e, em seguida, trabalhando com um subconjunto dos dados originais.

### Lendo o arquivo

O primeiro arquivo com o qual trabalharemos é uma compilação de todos os acidentes de carro na Inglaterra de 1979 a 2004, para extrair todos os acidentes que aconteceram em Londres no ano de 2000.

### Excel

Vamos acessar o conteúdo do arquivo ZIP que nos dará o dataset para que nossa aplicação possa consumir – podemos acessar o link em [data.gov.uk](https://data.gov.uk) -

e extraia o conteúdo. Em seguida, tente abrir Accidents7904.csv no Excel. Seja cuidadoso. Se você não tiver memória suficiente, isso pode muito bem travar seu computador. Você deve ver um erro “Arquivo não carregado completamente”, pois o Excel só pode lidar com um milhão de linhas por vez.

Se você estiver usando o LibreOffice também receberá um erro semelhante - “Os dados não puderam ser carregados completamente porque o número máximo de linhas por folha foi excedido”.

### Utilizando o Pandas

Vamos construir o script. Crie um arquivo notebook (no Jupyter Notebook) chamado e adicione o seguinte código:

```
import pandas as pd

# Lendo o arquivo

data = pd.read_csv("Accidents7904.csv",
low_memory=False)

# Saída exibindo o número de linhas

print("Total de linhas: {0}".format(len(data)))

# Observar quais headers são válidos

print(list(data))
```

Aqui, importamos Pandas, lemos o arquivo - o que pode levar algum tempo, dependendo de quanta memória seu sistema possui - e geramos o número total de linhas que o arquivo possui, bem como os cabeçalhos disponíveis (por exemplo, títulos de colunas).

Quando executado, você verá:

```
Total de linhas: 6224198
```

```
['\xef\xbb\xbfAccident_Index',
'Location_Easting_OSGR', 'Location_Northing_OSGR',
'Longitude', 'Latitude', 'Police_Force',
'Accident_Severity', 'Number_of_Vehicles',
'Number_of_Casualties', 'Date', 'Day_of_Week',
'Time', 'Local_Authority_(District)',
'Local_Authority_(Highway)', '1st_Road_Class',
'1st_Road_Number', 'Road_Type',
'Speed_limit', 'Junction_Detail', 'Junction_Control',
'2nd_Road_Class',
'2nd_Road_Number', 'Pedestrian_Crossing-
Human_Control',
'Pedestrian_Crossing-Physical_Facilities',
'Light_Conditions', 'Weather_Conditions',
'Road_Surface_Conditions',
'Special_Conditions_at_Site', 'Carriageway_Hazards',
'Urban_or_Rural_Area',
'Did_Police_Officer_Attend_Scene_of_Accident',
'LSOA_of_Accident_Location']
```

Portanto, existem mais de seis milhões de linhas. Volte sua atenção para a lista de cabeçalhos, o primeiro em particular:

```
'\xef\xbb\xbfAccident_Index',
```

Isso deve ler Accident\_Index. O que há com o extra \xef\xbb\xbf no início? Bem, na verdade significa que o valor é hexadecimal, que é uma Marca de Ordem de Byte, indicando que o texto é Unicode.

Por que isso é importante para nós?

Você não pode presumir que os arquivos que lê estão limpos. Eles podem conter símbolos extras como este que podem confundir seus scripts.

Este arquivo é bom, no sentido de que está limpo - mas muitos arquivos têm dados ausentes, dados em formato interno inconsistente, etc. Portanto, sempre que você tiver um arquivo para analisar, a primeira coisa que você deve fazer é limpá-lo. Quanta limpeza? O suficiente para permitir que você faça algumas análises.

Que tipo de limpeza você pode exigir?

- Corrija a data / hora. O mesmo arquivo pode ter datas em formatos diferentes, como os formatos americano (mm-dd-aa) ou europeu (dd-mm-aa). Eles precisam ser colocados em um formato comum.
- Remova todos os valores vazios. O arquivo pode ter colunas e / ou linhas em branco, e isso aparecerá como NaN (não é um número) no Pandas. O Pandas oferece uma maneira simples de removê-los: a função **dropna()**.
- Remova qualquer valor de lixo que tenha entrado nos dados. Esses são valores que não fazem sentido (como a marca de ordem de bytes que vimos anteriormente). Às vezes, pode ser possível contorná-los. Por exemplo, pode haver um conjunto de dados onde a idade foi inserida como um número de ponto flutuante (por engano). A função `int()` então pode ser usada para garantir que todas as idades estejam no formato inteiro.

## Analizando

Para aqueles que conhecem SQL, você pode usar as instruções SELECT, WHERE, AND / OR com diferentes palavras-chave para refinar sua pesquisa. Podemos fazer o mesmo no Pandas e de uma forma mais amigável para o programador .

Para começar, vamos ver todos os acidentes que aconteceram em um domingo. Olhando para os cabeçalhos acima, há um campo Day\_of\_Week, o usaremos.

No arquivo ZIP que você baixou, há um arquivo chamado Road-Accident-Safety-Data-Guide-1979-2004.xls , que contém informações extras sobre os códigos usados. Se você abrir, verá que o domingo tem o código 1.

```

print("\nAcidentes")

print("-----")

# Acidentes que ocorreram em um Domingo

accidents_sunday = data[data.Day_of_Week == 1]

print("Acidentes que ocorreram em um Domingo:
{0}".format(

    len(accidents_sunday)))

```

Aqui, direcionamos o Day\_of\_Week campo e retornamos um DataFrame com a condição que verificamos - day of week == 1.

Quando executado, você verá:

```

Acidentes
-----
Acidentes que ocorreram em um Domingo: 693847

```

Como você pode ver, houve 693.847 acidentes ocorridos em um domingo.

Vamos complicar nossa consulta: Descubra todos os acidentes que aconteceram em um domingo e envolveram mais de vinte carros:

```

# Acidentes que ocorreram no Domingo, > 20 carros

accidents_sunday_twenty_cars = data[

    (data.Day_of_Week == 1) & (data.Number_of_Vehicles
> 20)]

print("Acidentes que ocorreram em um Domingo
envolvendo > 20 carros: {0}".format(

    len(accidents_sunday_twenty_cars)))

```

Execute o script. Agora temos 10 acidentes:

```

Acidentes
-----

Acidentes que ocorreram em um Domingo: 693847

Acidentes que ocorreram em um Domingo envolvendo > 20
carros: 10

```

Vamos adicionar outra condição: clima.

Abra o **Road-Accident-Safety-Data-Guide-1979-2004.xls** e vá para a planilha meteorológica. Você verá que o código 2 significa “**Chovendo sem ventos fortes**” – tradução livre.

Adicione isso à nossa consulta:

```

# Acidentes que ocorreram no Domingo, > 20 carros, na
chuva

accidents_sunday_twenty_cars_rain = data[

    (data.Day_of_Week == 1) & (data.Number_of_Vehicles
> 20) &

    (data.Weather_Conditions == 2)]

print("Acidentes que ocorreram em um Domingo
envolvendo > 20 carros na chuva: {0}".format(

    len(accidents_sunday_twenty_cars_rain)))

```

Portanto, houve quatro acidentes ocorridos em um domingo, envolvendo mais de vinte carros, enquanto estava chovendo:

```

Acidentes
-----

Acidentes que ocorreram em um Domingo: 693847

Acidentes que ocorreram em um Domingo envolvendo > 20
carros: 10

```

```
Acidentes que ocorreram em um Domingo envolvendo > 20
carross na chuva: 4
```

Poderíamos continuar tornando isso cada vez mais extenso e complexo, conforme necessário. Vamos voltar nossa observação para o principal ponto de interesse: observar, analisar e explorar os acidentes na cidade de Londres.

Se você olhar para o arquivo **Road-Accident-Safety-Data-Guide-1979-2004.xls** novamente, há uma aba chamada **Police\_Force**. O código para 1 diz, “**Metropolitan Police**” - comumente conhecido como Scotland Yard, e é a força policial responsável pela maior parte (embora não por toda) de Londres. Para o nosso caso, isso é bom o suficiente, e podemos extrair essas informações assim:

```
# Acidentes em Londres em um Domingo

london_data = data[data['Police_Force'] == 1 &
(data.Day_of_Week == 1)]

print("\nAcidents em Londres de 1979-2004 ocorridos em
um Domingo: {0}".format(

    len(london_data)))
```

Execute o script. Isso criou um novo DataFrame com os acidentes tratados pela “Polícia Metropolitana” de 1979 a 2004 em um domingo:

```
Acidentes
-----

Acidentes que ocorreram em um Domingo: 693847

Acidentes que ocorreram em um Domingo envolvendo > 20
carros: 10

Acidentes que ocorreram em um Domingo envolvendo > 20
carross na chuva: 4

Acidentes em Londres de 1979-2004 ocorridos em um
Domingo: 114624
```

E se você quisesse criar um novo DataFrame que contivesse apenas acidentes no ano 2000?

A primeira coisa que precisamos fazer é converter o formato de data para um que o Python possa entender usando a função **`pd.to_datetime()`**. Isso pega uma data em qualquer formato e a converte para um formato que possamos entender (aaaa-mm-dd). Então, podemos criar outro DataFrame que contém apenas acidentes de 2000:

```
# Convertendo datas para o Pandas date/time

london_data_2000 = london_data[

    (pd.to_datetime(london_data['Date'],
erros='coerce') >

        pd.to_datetime('2000-01-01', erros='coerce'))
&

    (pd.to_datetime(london_data['Date'],
erros='coerce') <

        pd.to_datetime('2000-12-31', erros='coerce'))

]

print("Acidentes em Londres no ano 2000 ocorridos em
um Domingo: {0}".format(

    len(london_data_2000)))
```

Quando executado, você verá:

```
Acidentes que ocorreram em um Domingo: 693847

Acidentes que ocorreram em um Domingo envolvendo > 20
carros: 10

Acidentes que ocorreram em um Domingo envolvendo > 20
carross na chuva: 4

Acidentes em Londres de 1979-2004 ocorridos em um
Domingo: 114624

Acidentes em Londres no 2000 ocorridos em um Domingo:
3889
```



Normalmente, para filtrar uma matriz, você apenas usaria um **forloop** com uma condicional:

```
for data in array:
    if data > X and data < X:
        # Ocorre algo
```

No entanto, você realmente não deve definir seu próprio loop, já que muitas bibliotecas de alto desempenho, como o Pandas, possuem funções auxiliares. Nesse caso, o código acima percorre todos os elementos e filtra os dados fora das datas definidas e, em seguida, retorna os pontos de dados que se enquadram nas datas.

### Formatando o resultado

Primeiro, precisamos fazer uma limpeza. Lembra da marca de ordem de bytes que vimos antes? Isso causa problemas ao gravar esses dados em um arquivo do Excel - o Pandas lança um UnicodeDecodeError . Por quê? Porque o resto do texto é decodificado como ASCII, mas os valores hexadecimais não podem ser representados em ASCII.

Poderíamos escrever tudo como Unicode, mas lembre-se que essa marca de ordem de bytes é um extra desnecessário (para nós) que não queremos ou precisamos. Portanto, vamos nos livrar disso renomeando o cabeçalho da coluna:

```
london_data_2000.rename(
    columns={'\xef\xbb\xbfAccident_Index':
'Accident_Index'},
    inplace=True)
```

Esta é a maneira de renomear uma coluna no Pandas; um pouco complicado, para ser honesto. `inplace = True` é necessário porque queremos modificar a estrutura existente, e não criar uma cópia, que é o que o Pandas faz por padrão.

Agora podemos salvar os dados no Excel:

```
# Salvando como arquivo no formato Excel

writer = pd.ExcelWriter(

    'Acid_Londres_Domingo_2000.xlsx',
engine='xlsxwriter')

london_data_2000.to_excel(writer, 'Planilha1')

writer.save()
```

Certifique-se de instalar o XlsxWriter – caso não esteja usando a distribuição Anaconda - antes de executar:

```
$ pip install XlsxWriter
```

Se tudo tiver corrido bem, isso deve ter criado um arquivo chamado **Acid\_Londres\_Domingo\_2000.xlsx** e, em seguida, salvo nossos dados na Planilha1 . Abra este arquivo no Excel ou LibreOffice e confirme se os dados estão corretos.

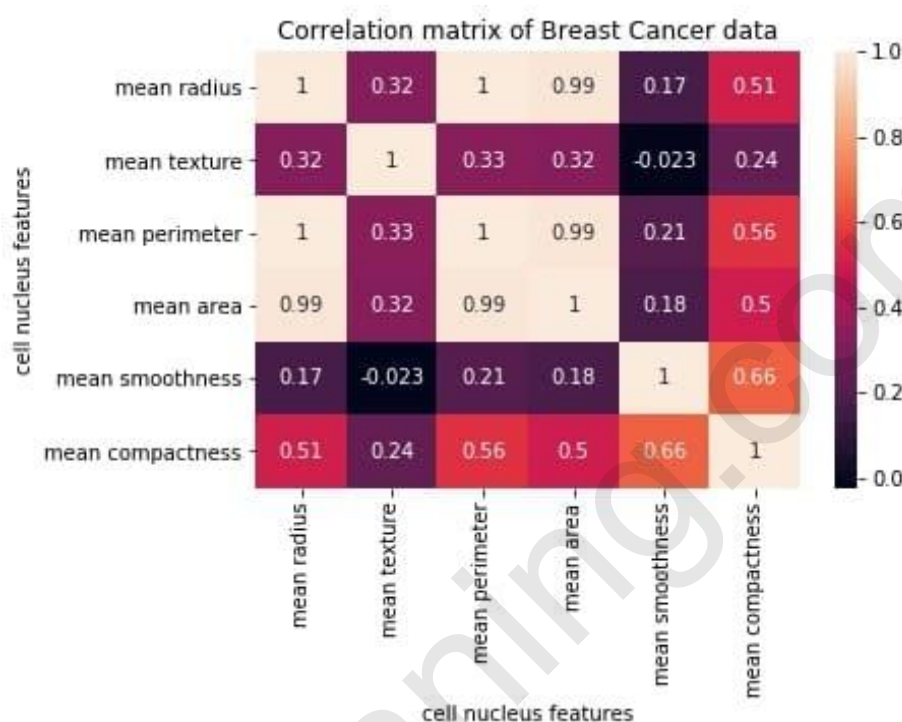
## Resumo

Então, o que realizamos? Acessamos um arquivo - com um volume de dados muito extenso que o Excel não conseguiu abrir e utilizamos o Pandas para:

1. Abrir o arquivo.
2. Executar consultas semelhantes a SQL nos dados.
3. Criar um novo arquivo XLSX com um subconjunto dos dados originais.

## MatrizDe Correlação

Uma matrizde correlação é um dado tabular que representa as 'correlações' entre pares de variáveis em um dado dado.



Cada linha e coluna representa uma variável, e cada valor nesta matriz é o coeficiente de correlação entre as variáveis representadas pela linha e coluna correspondentes.

A matrizde correlação é uma importante métrica de análise de dados que é calculada para resumir os dados para entender a relação entre várias variáveis e tomar decisões de acordo.

É também uma etapa de pré-processamento importante em pipelines de aprendizado de máquina para calcular e analisar a matriz de correlação onde a redução de dimensionalidade é desejada em dados de alta dimensão.

Mencionamos como cada célula na matriz de correlação é um ' **coeficiente de correlação** ' entre as duas variáveis correspondentes à linha e coluna da célula.

### Qual é o coeficiente de correlação?

Um coeficiente de correlação é um número que denota a força da relação entre duas variáveis.

Existem vários tipos de coeficientes de correlação, mas o mais comum deles é o coeficiente de Pearson denotado pela letra grega  $\rho$  (rho).

É definida como a covariância entre duas variáveis dividida pelo produto dos **desvios padrão** das duas variáveis.

$$\rho(X, Y) = \frac{COV(X, Y)}{\sigma_X \sigma_Y}$$

Aqui, a covariância entre X e Y  $COV(X, Y)$  é ainda definida como o 'valor esperado do produto dos desvios de X e Y de suas respectivas médias'. A fórmula para covariância tornaria isso mais claro.

$$COV(X, Y) = E[(X - \mu_X)(Y - \mu_Y)]$$

Portanto, a fórmula para a correlação de Pearson seria:

$$\rho(X, Y) = \frac{E[(X - \mu_X)(Y - \mu_Y)]}{\sigma_X \sigma_Y}$$

O valor de  $\rho$  está entre -1 e +1. Valores próximos de +1 indicam a presença de uma forte relação positiva entre X e Y, enquanto aqueles próximos de -1 indicam uma forte relação negativa entre X e Y. Valores próximos de zero significam que há ausência de qualquer relação entre X e Y.

### Buscando a matriz de correlação dos dados fornecidos

Vamos gerar dados aleatórios para duas variáveis e, em seguida, construir a matriz de correlação para elas.

```
import numpy as np

np.random.seed(10)

# generating 10 random values for each of the two
variables
X = np.random.randn(10)

Y = np.random.randn(10)

# computing the correlation matrix
C = np.corrcoef(X,Y)

print(C)
```

Saida:

```
[[1.          0.37258014]
 [0.37258014  1.          ]]
```

Como calculamos a matriz de correlação de 2 variáveis, suas dimensões são 2 x 2.

O valor 0,37 indica que não existe uma relação entre as duas variáveis. Isso era esperado, uma vez que seus valores foram gerados aleatoriamente.

Neste exemplo, usamos o método `corrcoef` de NumPy para gerar a matriz de correlação.

No entanto, este método tem uma limitação porque pode calcular a matriz de correlação entre 2 variáveis apenas.

Portanto, indo em frente, usaremos o **pandas DataFrames** para armazenar os dados e calcular a matriz de correlação neles.

### Traçando a matriz de correlação

Para esta explicação, usaremos um conjunto de dados que possui mais do que apenas dois recursos.

Usaremos os dados do câncer de mama, um dado popular de classificação binária usado nas aulas introdutórias de ML. Vamos carregar esse conjunto de dados do **dataset** módulo do scikit-learn . Ele é retornado na forma de **matriz NumPy** , mas iremos convertê-los em Pandas DataFrame.

```
from sklearn.datasets import load_breast_cancer

import pandas as pd

breast_cancer = load_breast_cancer()

data = breast_cancer.data

features = breast_cancer.feature_names

df = pd.DataFrame(data, columns = features)

print(df.shape)

print(features)
```

Saída:

```
(569, 30)
['mean radius' 'mean texture' 'mean perimeter' 'mean area'
 'mean smoothness' 'mean compactness' 'mean concavity'
 'mean concave points' 'mean symmetry' 'mean fractal dimension'
 'radius error' 'texture error' 'perimeter error' 'area error'
 'smoothness error' 'compactness error' 'concavity error'
 'concave points error' 'symmetry error' 'fractal dimension error'
 'worst radius' 'worst texture' 'worst perimeter' 'worst area'
 'worst smoothness' 'worst compactness' 'worst concavity'
 'worst concave points' 'worst symmetry' 'worst fractal dimension']
```

Existem 30 recursos nos dados, todos listados na saída acima.

Nosso objetivo agora é determinar a relação entre cada par dessas colunas. Faremos isso traçando a matriz de correlação.

Para manter as coisas simples, usaremos apenas as primeiras seis colunas e plotaremos sua matriz de correlação. Para plotar a array, usaremos uma biblioteca de visualização popular chamada seaborn, que é construída sobre matplotlib.

```
import seaborn as sns

import matplotlib.pyplot as plt

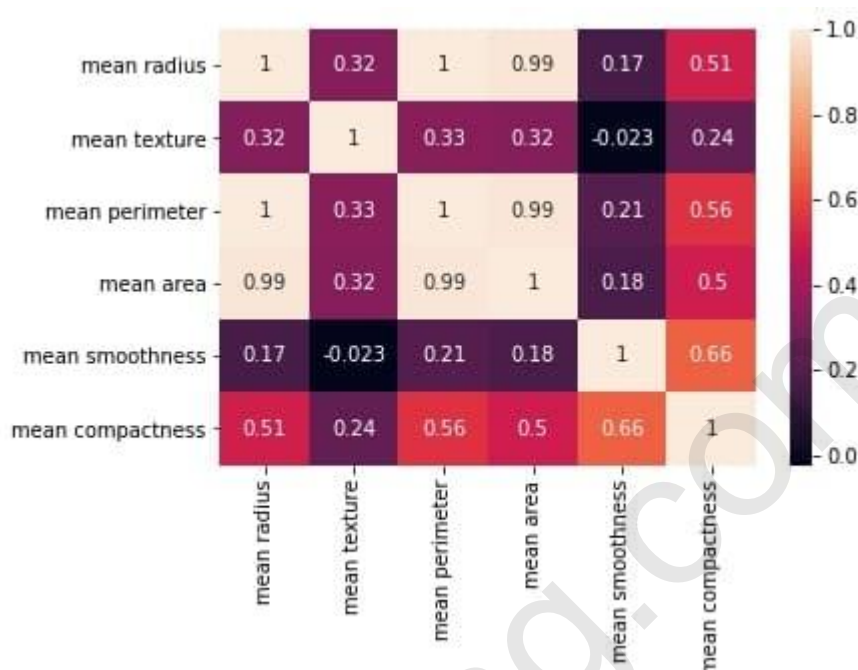
# taking all rows but only 6 columns
df_small = df.iloc[:, :6]

correlation_mat = df_small.corr()

sns.heatmap(correlation_mat, annot = True)

plt.show()
```

Saida:



O gráfico mostra uma matriz 6 x 6 e preenche com cor cada célula com base no coeficiente de correlação do par que a representa.

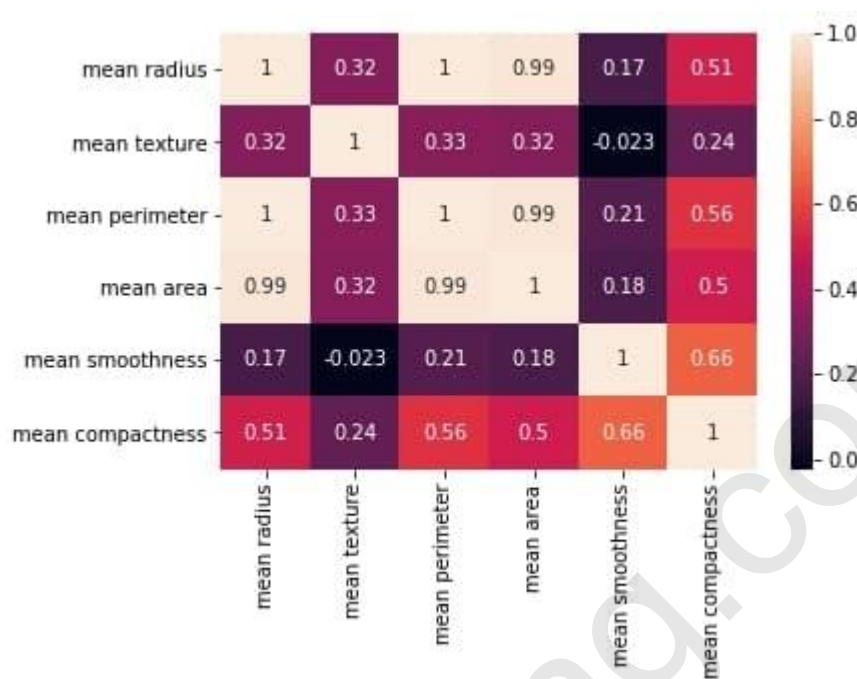
O método **corr()** do Pandas DataFrame é usado para calcular a array. Por padrão, ele calcula o coeficiente de correlação de Pearson. Também podemos usar outros métodos, como o coeficiente de Spearman ou o coeficiente de correlação de Kendall Tau, passando um valor apropriado para o parâmetro 'method'.

Usamos o método **heatmap()** do seaborn para traçar a array. O parâmetro 'annot=True' exibe os valores do coeficiente de correlação em cada célula.

Vamos observar e entender como interpretar a matriz de coeficientes de correlação traçada.



## Interpretando a matriz de correlação



Você deve manter os seguintes pontos em mente com relação à matriz de correlação:

1. Cada célula da grade representa o valor do coeficiente de correlação entre duas variáveis.
2. O valor na posição (a, b) representa o coeficiente de correlação entre os recursos na linha a e coluna b. Isso será igual ao valor na posição (b, a)
3. É uma **matriz quadrada** - cada linha representa uma variável e todas as colunas representam as mesmas variáveis que linhas, portanto, o número de linhas = número de colunas.
4. É uma **matriz simétrica** - isso faz sentido porque a correlação entre a, b será a mesma que entre b, a.
5. Todos os **elementos diagonais são 1**. Como os elementos diagonais representam a correlação de cada variável consigo mesma, ela sempre será igual a 1.

6. Os pontos dos eixos denotam o recurso que cada um deles representa.
7. Um grande valor positivo (próximo a 1,0) indica uma forte correlação positiva, ou seja, se o valor de uma das variáveis aumenta, o valor da outra variável também aumenta.
8. Um grande valor negativo (próximo a -1,0) indica uma forte correlação negativa, ou seja, o valor de uma variável diminui com o aumento da outra e vice-versa.
9. Um valor próximo a 0 (positivo ou negativo) indica a ausência de qualquer correlação entre as duas variáveis e, portanto, essas variáveis são independentes uma da outra.
10. Cada célula na matriz acima também é representada por tons de uma cor. Aqui, tons mais escuros da cor indicam valores menores, enquanto tons mais brilhantes correspondem a valores maiores (próximo a 1). Esta escala é dada com a ajuda de uma barra de cores no lado direito do gráfico.

### Adicionando título e rótulos ao gráfico

Podemos ajustar a matriz de correlação gerada, assim como qualquer outro gráfico Matplotlib. Vamos ver como podemos adicionar um título à matriz e rótulos aos eixos.

```
correlation_mat = df_small.corr()

sns.heatmap(correlation_mat, annot = True)

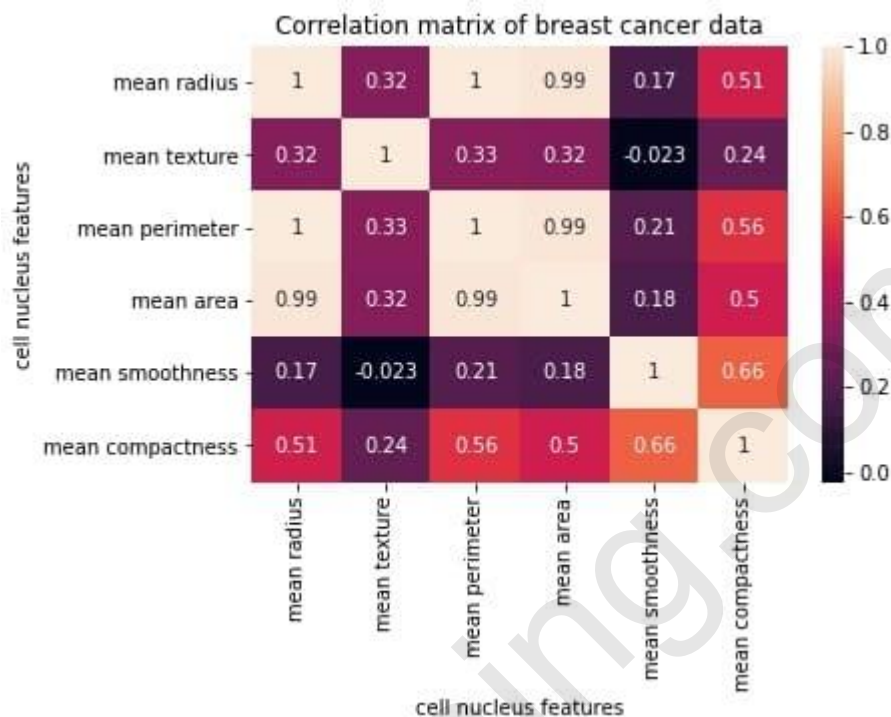
plt.title("Matriz de correlação dos dados de Cancer de
Mama")

plt.xlabel("características do núcleo da célula ")

plt.ylabel("características do núcleo da célula ")

plt.show()
```

Saida:



Também podemos alterar a posição do título para baixo, especificando a posição y.

```
correlation_mat = df_small.corr()

sns.heatmap(correlation_mat, annot = True)

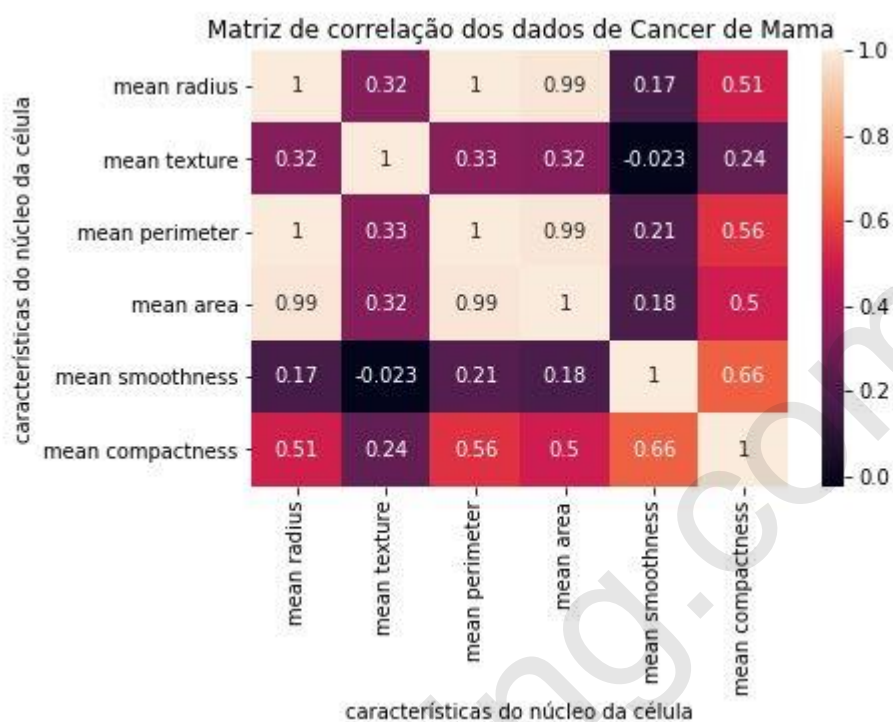
plt.title("Matriz de correlação dos dados de Cancer de
Mama", y=-0.75)

plt.xlabel("características do núcleo da célula")

plt.ylabel("características do núcleo da célula")

plt.show()
```

Saida:



### Classificando a matriz de correlação

Se os dados fornecidos tiverem um grande número de recursos, a matriz de correlação pode se tornar muito grande e, portanto, difícil de interpretar.

Às vezes, podemos querer classificar os valores na matriz para ver a força da correlação entre os vários pares de recursos em ordem crescente ou decrescente.

Vamos ver como podemos conseguir isso.

Primeiro, converteremos a matriz fornecida em uma série de valores unidimensional.

```
correlation_mat = df_small.corr()

corr_pairs = correlation_mat.unstack()

print(corr_pairs)
```

Saida:

```
mean radius    mean radius    1.000000
               mean texture    0.323782
               mean perimeter    0.997855
               mean area        0.987357
               mean smoothness  0.170581
               mean compactness 0.506124
mean texture    mean radius    0.323782
               mean texture    1.000000
               mean perimeter    0.329533
               mean area        0.321086
               mean smoothness -0.023389
               mean compactness 0.236702
mean perimeter  mean radius    0.997855
               mean texture    0.329533
               mean perimeter    1.000000
               mean area        0.986507
               mean smoothness  0.207278
               mean compactness 0.556936
mean area       mean radius    0.987357
               mean texture    0.321086
               mean perimeter    0.986507
               mean area        1.000000
               mean smoothness  0.177028
               mean compactness 0.498502
mean smoothness mean radius    0.170581
               mean texture    -0.023389
               mean perimeter    0.207278
               mean area        0.177028
               mean smoothness  1.000000
               mean compactness 0.659123
mean compactness mean radius    0.506124
               mean texture    0.236702
               mean perimeter    0.556936
               mean area        0.498502
               mean smoothness  0.659123
               mean compactness 1.000000

dtype: float64
```

O método `unstack` no Pandas `DataFrame` retorna uma Série com **MultilIndex**. Ou seja, cada valor na Série é representado por mais de um índice, que neste caso são os índices de linha e coluna que são os nomes dos recursos.

Vamos agora classificar esses valores usando o método `sort_values()` da Série Pandas.

```
sorted_pairs =  
corr_pairs.sort_values(kind="quicksort")  
  
print(sorted_pairs)
```

Saida:

```

mean texture      mean smoothness    -0.023389
mean smoothness   mean texture      -0.023389
mean radius       mean smoothness    0.170581
mean smoothness   mean radius       0.170581
mean area         mean smoothness    0.177028
mean smoothness   mean area         0.177028
                  mean perimeter    0.207278
mean perimeter    mean smoothness    0.207278
mean texture      mean compactness  0.236702
mean compactness  mean texture      0.236702
mean area         mean texture      0.321086
mean texture      mean area         0.321086
                  mean radius       0.323782
mean radius       mean texture      0.323782
mean texture      mean perimeter    0.329533
mean perimeter    mean texture      0.329533
mean compactness  mean area         0.498502
mean area         mean compactness  0.498502
mean compactness  mean radius       0.506124
mean radius       mean compactness  0.506124
mean perimeter    mean compactness  0.556936
mean compactness  mean perimeter    0.556936
                  mean smoothness    0.659123
mean smoothness   mean compactness  0.659123
mean perimeter    mean area         0.986507
mean area         mean perimeter    0.986507
                  mean radius       0.987357
mean radius       mean area         0.987357
mean perimeter    mean radius       0.997855
mean radius       mean perimeter    0.997855
                  mean radius       1.000000
mean area         mean area         1.000000
mean perimeter    mean perimeter    1.000000
mean texture      mean texture      1.000000
mean smoothness   mean smoothness    1.000000
mean compactness  mean compactness  1.000000
dtype: float64

```

Podemos ver que cada valor é repetido duas vezes na saída classificada. Isso ocorre porque nossa matriz de correlação era uma matriz simétrica e cada par de características ocorria duas vezes nela.

No entanto, agora temos os valores de coeficiente de correlação classificados de todos os pares de recursos e podemos tomar decisões de acordo.

### Seleção de pares de correlação negativa

Podemos querer selecionar pares de características com uma faixa particular de valores do coeficiente de correlação. Vamos ver como podemos escolher pares com uma correlação negativa dos pares classificados que geramos na seção anterior.



```
negative_pairs = sorted_pairs[sorted_pairs < 0]

print(negative_pairs)
```

Saida:

```
mean texture    mean smoothness  -0.023389
mean smoothness mean texture     -0.023389
dtype: float64
```

### Seleção de pares de correlação fortes (magnitude maior que 0,5)

Vamos usar a mesma abordagem para escolher recursos fortemente relacionados. Ou seja, tentaremos filtrar os pares de recursos cujos valores de coeficiente de correlação são maiores que 0,5 ou menores que -0,5.

```
strong_pairs = sorted_pairs[abs(sorted_pairs) > 0.5]

print(strong_pairs)
```

Saida:



```

mean compactness mean radius 0.506124
mean radius mean compactness 0.506124
mean perimeter mean compactness 0.556936
mean compactness mean perimeter 0.556936
mean smoothness mean smoothness 0.659123
mean smoothness mean compactness 0.659123
mean perimeter mean area 0.986507
mean area mean perimeter 0.986507
mean radius mean radius 0.987357
mean radius mean area 0.987357
mean perimeter mean radius 0.997855
mean radius mean perimeter 0.997855
mean radius mean radius 1.000000
mean area mean area 1.000000
mean perimeter mean perimeter 1.000000
mean texture mean texture 1.000000
mean smoothness mean smoothness 1.000000
mean compactness mean compactness 1.000000
dtype: float64

```

### Converter uma matrizde covariância na matrizde correlação

Anteriormente, vimos a relação entre a covariância e a correlação entre um par de variáveis. Observe. A figura abaixo:

$$\rho(X, Y) = \frac{COV(X, Y)}{\sigma_X \sigma_Y}$$

Vamos entender como podemos calcular a matrizde covariância de um dado em Python e então convertê-la em uma matrizde correlação. Vamos compará-la com a matrizde correlação que geramos usando uma chamada de método direta.

Em primeiro lugar, o Pandas não fornece um método para calcular a covariância entre todos os pares de variáveis, então usaremos o método `cov()` da dependencia NumPy .

```
cov = np.cov(df_small.T)

print(cov)
```

Saída:

```
[[ 1.24189201e+01  4.90758156e+00  8.54471417e+01  1.22448341e+03
   8.45445983e-03  9.41970568e-02]
 [ 4.90758156e+00  1.84989087e+01  3.44397592e+01  4.85993787e+02
  -1.41477877e-03  5.37668058e-02]
 [ 8.54471417e+01  3.44397592e+01  5.90440480e+02  8.43577235e+03
   7.08360652e-02  7.14714125e-01]
 [ 1.22448341e+03  4.85993787e+02  8.43577235e+03  1.23843554e+05
   8.76178126e-01  9.26493079e+00]
 [ 8.45445983e-03 -1.41477877e-03  7.08360652e-02  8.76178126e-01
  1.97799700e-04  4.89573915e-04]
 [ 9.41970568e-02  5.37668058e-02  7.14714125e-01  9.26493079e+00
   4.89573915e-04  2.78918740e-03]]
```

Estamos passando a matriz transposta porque o método espera uma matriz na qual cada um dos recursos é representado por uma linha em vez de uma coluna.

Portanto, acertamos nosso numerador. Agora, precisamos calcular uma matriz  $6 \times 6$  na qual o valor em  $i, j$  é o produto dos desvios-padrão dos recursos nas posições  $i$  e  $j$ .

Em seguida, dividiremos a matriz de covariância por esta matriz de desvios padrão para calcular a matriz de correlação.

Vamos primeiro construir a matriz de desvios padrão.

```
# calcular os desvios padrão de cada um dos 6 recursos
stds = np.std(df_small, axis = 0) #shape = (6,)

stds_matrix = np.array([[stds[i]*stds[j] for j in
range(6)] for i in range(6)])

print("matriz de desvios padrão da
forma:", stds_matrix.shape)
```

Saida:

```
matriz de desvios padrão da forma: (6, 6)
```

Agora que temos a matriz de covariância da forma (6,6) para as 6 características e o produto par a par da matriz das características da forma (6,6), podemos dividir as duas e ver se obtemos a matriz de correlação resultante desejada.

```
new_corr = cov/std_matrix
```

Armazenamos a nova matriz de correlação (derivada de uma matriz de covariância) na variável *new\_corr*.

Vamos verificar se acertamos traçando a matriz de correlação e justapondo-a com a anterior gerada diretamente pelo método Pandas *corr()*.

```
plt.figure(figsize=(18,4))

plt.subplot(1,2,1)

sns.heatmap(correlation_mat, annot = True)

plt.title("Matriz de correlação anterior (de Pandas)")

plt.xlabel("características do núcleo da célula")

plt.ylabel("características do núcleo da célula")

plt.subplot(1,2,2)

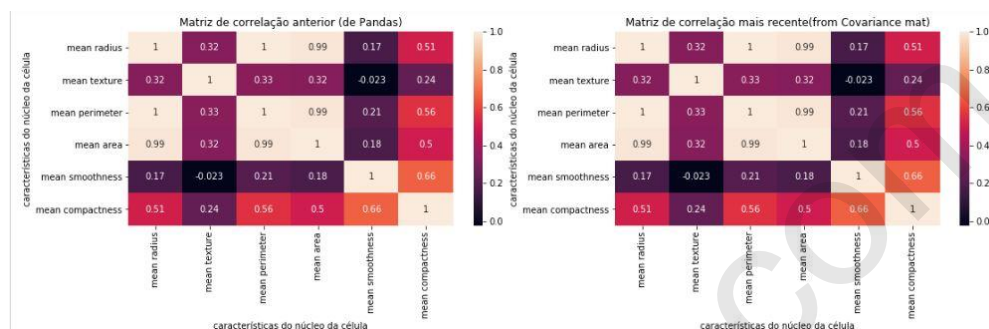
sns.heatmap(correlation_mat, annot = True)

plt.title("Matriz de correlação mais recente (from Covariance mat)")

plt.xlabel("características do núcleo da célula")
```

```
plt.ylabel("características do núcleo da célula")
plt.show()
```

Saída:



Podemos comparar as duas matrizes e notar que são idênticas.

Exportando a matriz de correlação para uma imagem

Traçar a matriz de correlação em um script Python não é suficiente. Podemos querer salvá-lo para uso posterior. Podemos salvar o gráfico gerado como um arquivo de imagem em disco usando o método ***plt.savefig()***.

```
correlation_mat = df_small.corr()
sns.heatmap(correlation_mat, annot = True)

plt.title("Matriz de correlação de dados de câncer de mama")

plt.xlabel("características do núcleo da célula")

plt.ylabel("características do núcleo da célula")
```

```
plt.savefig("breast_cancer_correlation.png")
```

Depois de executar este código, você pode ver um arquivo de imagem com o nome 'breast\_cancer\_correlation.png' no mesmo diretório de trabalho.

## Introdução ao Machine Learning

### Construindo um modelo ML passo-a-passo

Neste passo-a-passo, abordaremos:

1. Instalação do módulo Python SciPy – caso não esteja usando a distribuição Anaconda - para aprendizado de máquina em Python.
2. Carregar um conjunto de dados e entender sua estrutura usando resumos estatísticos e visualização de dados.
3. Criar 6 modelos de aprendizado de máquina, observa-los; calcular a precisão de cada um deles.

### Como iniciar um projeto de aprendizado de máquina em Python?

A melhor maneira de aprender o aprendizado de máquina é projetando e concluindo pequenos projetos.

Existem muitos módulos e bibliotecas – utilizando em Python - para escolher, oferecendo maneiras diferentes de fazer cada tarefa.

A melhor maneira de começar a usar Python para aprendizado de máquina é concluir um projeto.

- Ele lhe dará uma visão panorâmica de como percorrer um pequeno projeto.
- Isso lhe dará confiança, talvez para prosseguir com seus próprios projetos.

Ao aplicar o aprendizado de máquina aos seus próprios conjuntos de dados, você está trabalhando em um projeto.

Um projeto de aprendizado de máquina pode não ser linear, mas tem várias etapas bem conhecidas:

1. Defina o problema.
2. Prepare os dados.
3. Avalie algoritmos.
4. Melhore os resultados.
5. Resultados presentes.

A melhor maneira de trabalhar em um projeto de aprendizado de máquina de ponta a ponta e cobrir as principais etapas. Ou seja, desde o carregamento de dados, resumindo dados, avaliando algoritmos e fazendo algumas previsões.

Dessa forma, é possível construir um modelo que pode usar em conjunto de dados - um após o outro.

O melhor pequeno projeto para começar em uma nova ferramenta é a classificação das flores da íris (por exemplo, o conjunto de dados da íris ).

Este é um bom projeto porque é muito bem compreendido:

- Os atributos são numéricos, então você precisa descobrir como carregar e manipular os dados.
- É um problema de classificação, permitindo que você pratique talvez um tipo mais fácil de algoritmo de aprendizagem supervisionada.
- É um problema de classificação multi-classe (multi-nominal) que pode requerer algum tratamento especializado.
- Possui apenas 4 atributos e 150 linhas, o que significa que é pequeno e cabe facilmente na memória (e em uma tela ou página A4).
- Todos os atributos numéricos estão nas mesmas unidades e na mesma escala, não exigindo qualquer escala ou transformação especial para começar.

Aqui está uma visão geral do que vamos cobrir:

1. Instalando a plataforma Python e SciPy.
2. Carregando o conjunto de dados.
3. Resumindo o conjunto de dados.
4. Visualizando o conjunto de dados.
5. Avaliando alguns algoritmos.
6. Fazendo algumas previsões.

Vamos trabalhar em cada etapa.

## 1. Baixando, instalando e iniciando o Python SciPy

Obtenha a plataforma Python e SciPy instalada em seu sistema, se ainda não estiver.

### 1.1 Instalar Bibliotecas SciPy

Existem 5 bibliotecas principais que você precisará instalar. Abaixo está uma lista das bibliotecas Python SciPy necessárias para este projeto:

- scipy
- numpy
- matplotlib
- pandas
- sklearn

Se estiver usando a distribuição **Ananconda**, essas bibliotecas já estarão instaladas. Caso contrario, instale-as usando o comando ***pip install***.

A [página de instalação do scipy](#) fornece instruções excelentes para instalar as bibliotecas acima em várias plataformas diferentes, como Linux, mac OS X e Windows. Se você tiver alguma dúvida ou dúvida, consulte este guia, ele tem sido seguido por milhares de pessoas.

- No Mac OS X, você pode usar macports para instalar o Python 3.\* e essas bibliotecas. Para obter mais informações sobre macports, [consulte a página inicial](#).
- No Linux, você pode usar seu gerenciador de pacotes, como o yum no Fedora para instalar RPMs.

## 1.2 Inicie o Python e verifique as versões

É uma boa ideia certificar-se de que seu ambiente Python foi instalado com êxito e está funcionando conforme o esperado.

O script a seguir ajudará você a testar seu ambiente. Ele importa cada biblioteca necessária neste tutorial e imprime a versão.

Mantenha as coisas simples e concentre-se no aprendizado de máquina, não no conjunto de ferramentas.

Implemente o seguinte script:

# Conferindo as versões das dependências que serão usadas

```
# Python version
import sys
print('Python: {}'.format(sys.version))
# scipy
import scipy
print('scipy: {}'.format(scipy.__version__))
# numpy
import numpy
print('numpy: {}'.format(numpy.__version__))
# matplotlib
import matplotlib
print('matplotlib: {}'.format(matplotlib.__version__))
# pandas
import pandas
print('pandas: {}'.format(pandas.__version__))
# scikit-learn
import sklearn
print('sklearn: {}'.format(sklearn.__version__))
```

Aqui está o resultado:

```
Python: 3.8.5 (default, Sep  4 2020, 02:22:02)
[Clang 10.0.0 ]
```



```
scipy: 1.5.2  
numpy: 1.19.2  
matplotlib: 3.3.2  
pandas: 1.1.3  
sklearn: 0.23.2
```

Idealmente, suas versões devem corresponder ou ser mais recentes. As APIs não mudam rapidamente, então não se preocupe se você estiver algumas versões atrasadas, tudo neste tutorial provavelmente ainda funcionará para você.

Se você receber um erro, pare. Agora é a hora de consertar.

Se você não conseguir executar o script acima de forma limpa, não poderá concluir este seu projeto machine learning.

## 2. Carregue os dados

Vamos usar o conjunto de dados de flores de íris. Este conjunto de dados é famoso porque é usado como o conjunto de dados "hello world" no aprendizado de máquina e nas estatísticas por quase todos.

O conjunto de dados contém 150 observações de flores de íris. Existem quatro colunas de medidas das flores em centímetros. A quinta coluna é a espécie da flor observada. Todas as flores observadas pertencem a uma das três espécies.

Nesta etapa, carregaremos os dados da íris do URL do arquivo CSV.

### 2.1 Importar bibliotecas

Primeiro, vamos importar todos os módulos, funções e objetos que vamos usar neste passo-a-passo.

```
# Carregando as bibliotecas de dependencia  
from pandas import read_csv  
from pandas.plotting import scatter_matrix  
from matplotlib import pyplot  
from sklearn.model_selection import train_test_split  
from sklearn.model_selection import cross_val_score  
from sklearn.model_selection import StratifiedKFold  
from sklearn.metrics import classification_report
```

```

from sklearn.metrics import confusion_matrix
from sklearn.metrics import accuracy_score
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.discriminant_analysis import
LinearDiscriminantAnalysis
from sklearn.naive_bayes import GaussianNB
from sklearn.svm import SVC

```

Tudo deve carregar sem erros. Se você tiver um erro, pare. Você precisa de um ambiente SciPy funcional antes de continuar.

## 2.2 Carregar conjunto de dados

Podemos carregar os dados diretamente do repositório do UCI Machine Learning.

Estamos usando o **pandas** para carregar os dados. Também usaremos o **pandas next** para explorar os dados tanto com estatísticas descritivas quanto com visualização de dados.

Observe que estamos especificando os nomes de cada coluna ao carregar os dados. Isso ajudará mais tarde, quando explorarmos os dados.

```

# carregando o dataset
url =
"https://raw.githubusercontent.com/jbrownlee/Datasets/master/iris.csv"
names = ['sepal-length', 'sepal-width', 'petal-length',
'petal-width', 'class']
dataset = read_csv(url, names=names)

```

O conjunto de dados deve carregar sem incidentes.

Se você tiver problemas de rede, pode baixar o arquivo [iris.csv](#) em seu diretório de trabalho e carregá-lo usando o mesmo método, alterando o URL para o nome do arquivo local.

## 3. Resuma o conjunto de dados

Agora é hora de dar uma olhada nos dados.

Nesta etapa, daremos uma olhada nos dados de algumas maneiras diferentes:

1. Dimensões do conjunto de dados.
2. Dê uma olhada nos próprios dados.
3. Resumo estatístico de todos os atributos.
4. Repartição dos dados pela variável de classe.

Não se preocupe, cada olhar para os dados é um comando. Esses são comandos úteis que você pode usar repetidamente em projetos futuros.

### 3.1 Dimensões do conjunto de dados

Podemos ter uma ideia rápida de quantas instâncias (linhas) e quantos atributos (colunas) os dados contêm com a propriedade de forma.

```
# shape (formato)
print(dataset.shape)
```

Você deve ver 150 instâncias e 5 atributos:

```
(150, 5)
```

### 3.2 Uma olhada rápida nos dados

Também é sempre uma boa ideia examinar seus dados.

```
# função head no modo padrão lendo 5 primeiras linhas
print(dataset.head())
```

Você deve ver as primeiras 5 linhas dos dados:

	sepal-length	sepal-width	petal-length	petal-width	class
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa

### 3.3 Resumo Estatístico

Agora podemos dar uma olhada em um resumo de cada atributo.

Isso inclui a contagem, a média, os valores mínimo e máximo, bem como alguns percentis.

```
# descrição dos dados carregados
print(dataset.describe())
```

A saída será:

	sepal-length	sepal-width	petal-length	petal-width
count	150.000000	150.000000	150.000000	150.000000
mean	5.843333	3.054000	3.758667	1.198667
std	0.828066	0.433594	1.764420	0.763161
min	4.300000	2.000000	1.000000	0.100000
25%	5.100000	2.800000	1.600000	0.300000
50%	5.800000	3.000000	4.350000	1.300000
75%	6.400000	3.300000	5.100000	1.800000
max	7.900000	4.400000	6.900000	2.500000

### 3.4 Distribuição de classes

Vamos agora dar uma olhada no número de instâncias (linhas) que pertencem a cada classe. Podemos ver isso como uma contagem absoluta.

```
# distribuição de classes
print(dataset.groupby('class').size())
```

Podemos ver que cada classe possui o mesmo número de instâncias (50 ou 33% do conjunto de dados).

```
class
Iris-setosa      50
Iris-versicolor  50
Iris-virginica   50
dtype: int64
```

### 3.5 O código – até este ponto

Para referência, podemos amarrar todos os elementos anteriores juntos em um único script.

O exemplo completo está listado abaixo.

```
# sumarizando os dados
from pandas import read_csv
# carregando o dataset
url =
"https://raw.githubusercontent.com/jbrownlee/Datasets/master/iris.csv"
names = ['sepal-length', 'sepal-width', 'petal-length',
'petal-width', 'class']
dataset = read_csv(url, names=names)
# formato - shape
print(dataset.shape)
# função head-padrão para ler as 5 primeiras linhas do
dataset
print(dataset.head())
# descrição dos dados
print(dataset.describe())
# distribuição de classes
print(dataset.groupby('class').size())
```

```
(150, 5)
  sepal-length  sepal-width  petal-length  petal-width  class
0           5.1           3.5           1.4           0.2  Iris-setosa
1           4.9           3.0           1.4           0.2  Iris-setosa
2           4.7           3.2           1.3           0.2  Iris-setosa
3           4.6           3.1           1.5           0.2  Iris-setosa
4           5.0           3.6           1.4           0.2  Iris-setosa
count      150.000000    150.000000    150.000000    150.000000
mean         5.843333     3.054000     3.758667     1.198667
std          0.828066     0.433594     1.764420     0.763161
min          4.300000     2.000000     1.000000     0.100000
25%          5.100000     2.800000     1.600000     0.300000
50%          5.800000     3.000000     4.350000     1.300000
75%          6.400000     3.300000     5.100000     1.800000
max          7.900000     4.400000     6.900000     2.500000
class
Iris-setosa      50
Iris-versicolor  50
Iris-virginica   50
dtype: int64
```

## 4. Visualização de dados

Agora temos uma ideia básica sobre os dados. Precisamos estender isso com algumas visualizações.

Vamos examinar dois tipos de gráficos:

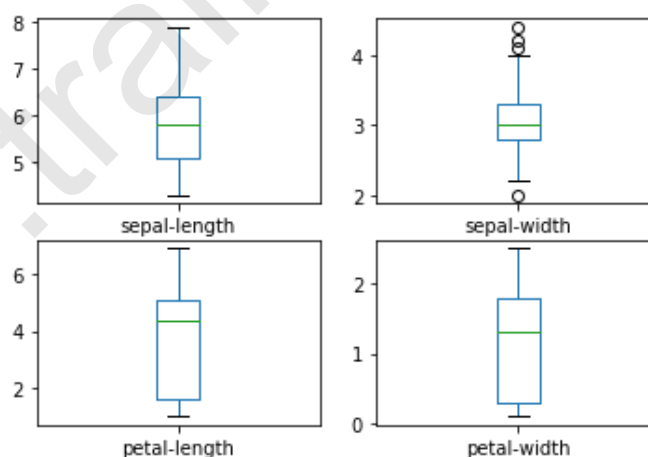
1. Plotagens univariadas para entender melhor cada atributo.
2. Gráficos multivariados para melhor compreender as relações entre os atributos.

### 4.1 Gráficos Univariados

Começamos com alguns gráficos univariados, ou seja, gráficos de cada variável individual. Dado que as variáveis de entrada são numéricas, podemos criar gráficos de caixa e bigode de cada uma.

```
# box e whisker plots
dataset.plot(kind='box', subplots=True, layout=(2,2),
sharex=False, sharey=False)
pyplot.show()
```

Isso nos dá uma ideia muito mais clara da distribuição dos atributos de entrada:

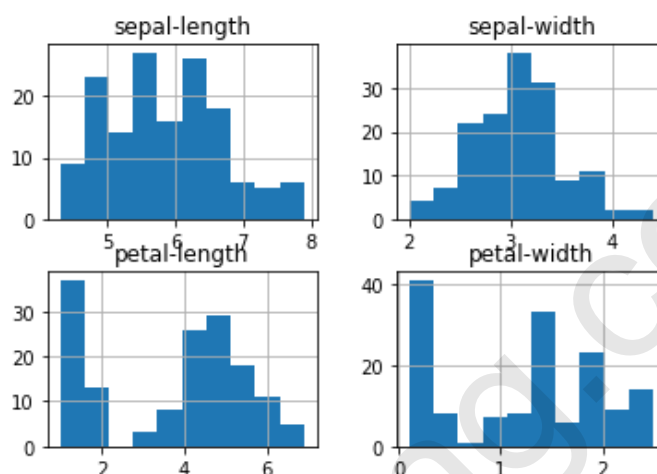


Plotagens de caixa e whiskers para cada variável de entrada para o conjunto de dados Iris Flowers

Também podemos criar um histograma de cada variável de entrada para ter uma ideia da distribuição.

```
# histogramas
dataset.hist()
pyplot.show()
```

Aparentemente duas das variáveis de entrada tenham uma distribuição gaussiana. É útil observar isso, pois podemos usar algoritmos que podem explorar essa suposição.



Gráficos de histograma para cada variável de entrada para o conjunto de dados  
Iris Flowers

## 4.2 Gráficos multivariados

Agora podemos examinar as interações entre as variáveis.

Primeiro, vamos examinar os gráficos de dispersão de todos os pares de atributos. Isso pode ser útil para identificar relacionamentos estruturados entre variáveis de entrada.

```
# scatter plot matrix
scatter_matrix(dataset)
pyplot.show()
```

Observe o agrupamento diagonal de alguns pares de atributos. Isso sugere uma alta correlação e um relacionamento previsível.

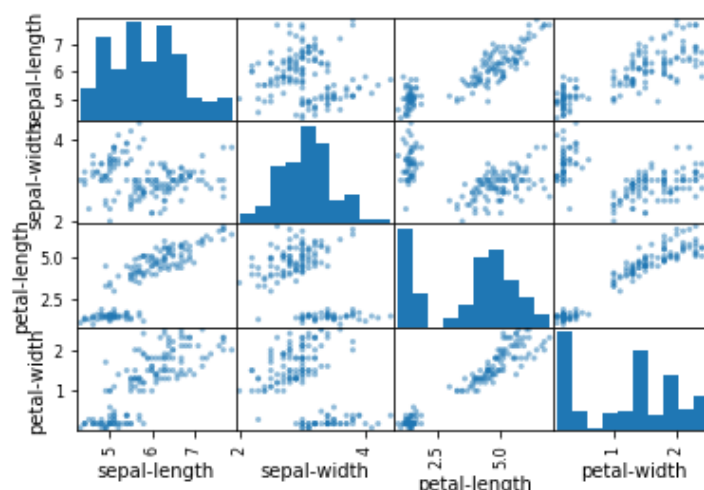


Gráfico de matriz de dispersão para cada variável de entrada para o conjunto de dados Iris Flowers

### 4.3 O código – até este ponto

Para referência, podemos amarrar todos os elementos anteriores juntos em um único script.

O exemplo completo está listado abaixo.

```
# visualizando os dados
from pandas import read_csv
from pandas.plotting import scatter_matrix
from matplotlib import pyplot

# carregando o dataset
url =
"https://raw.githubusercontent.com/jbrownlee/Datasets/master/iris.csv"
names = ['sepal-length', 'sepal-width', 'petal-length',
'petal-width', 'class']
dataset = read_csv(url, names=names)

# box e whisker plots
dataset.plot(kind='box', subplots=True, layout=(2,2),
sharex=False, sharey=False)
pyplot.show()

# histogramas
dataset.hist()
pyplot.show()

# scatter plot matrix
scatter_matrix(dataset)
pyplot.show()
```



## 5. Avaliando alguns algoritmos

Agora é o momento de criar alguns modelos dos dados e estimar sua precisão em dados não vistos. Abaixo, os tópicos que vamos cobrir nesta etapa:

1. Separar um conjunto de dados de validação.
2. Configurar o equipamento de teste para usar a validação cruzada de 10 vezes.
3. Construir vários modelos diferentes para prever espécies a partir de medições de flores
4. Selecionar o melhor modelo.

### 5.1 Criar um conjunto de dados de validação

Precisamos saber se o modelo que criamos é bom.

Posteriormente, usaremos métodos estatísticos para estimar a precisão dos modelos que criamos em dados não vistos. Também queremos uma estimativa mais concreta da precisão do melhor modelo em dados não vistos, avaliando-os em dados reais não vistos.

Ou seja, vamos reter alguns dados que os algoritmos não conseguirão ver e usaremos esses dados para obter uma segunda ideia independente de quão preciso o melhor modelo pode realmente ser.

Vamos dividir o conjunto de dados carregado em dois, 80% dos quais usaremos para treinar, avaliar e selecionar entre nossos modelos e 20% que iremos reter como um conjunto de dados de validação.

```
# Split-out (fatiamento) para validação do dataset
array = dataset.values
X = array[:,0:4]
y = array[:,4]
X_train, X_validation, Y_train, Y_validation =
train_test_split(X, y, test_size=0.20, random_state=1)
```

Agora você tem dados de treinamento no  $X_{train}$  e  $Y_{train}$  para a preparação de modelos e conjuntos de *validação*  $X_{validation}$  e  $Y_{validation}$  que podemos usar mais tarde.

Observe que usamos uma fatia Python para selecionar as colunas no array NumPy.

## 5.2 Teste Harness

Usaremos validação cruzada estratificada de 10 vezes para estimar a precisão do modelo.

Isso dividirá nosso conjunto de dados em 10 partes, treine em 9 e teste em 1 e repita para todas as combinações de divisões de teste de trem.

Estratificado significa que cada dobra ou divisão do conjunto de dados terá como objetivo ter a mesma distribuição de exemplo por classe que existe em todo o conjunto de dados de treinamento.

Definimos a semente aleatória por meio do argumento `random_state` para um número fixo para garantir que cada algoritmo seja avaliado nas mesmas divisões do conjunto de dados de treinamento.

Estamos usando a métrica de ' precisão ' para avaliar os modelos.

Esta é uma proporção do número de instâncias preditas corretamente dividido pelo número total de instâncias no conjunto de dados multiplicado por 100 para dar uma porcentagem (por exemplo, 95% de precisão). Estaremos usando a variável de pontuação quando executarmos a construção e avaliarmos cada modelo a seguir.

## 5.3 Construir Modelos

Não sabemos quais algoritmos seriam bons neste problema ou quais configurações usar.

A partir dos gráficos, temos uma ideia de que algumas das classes são parcialmente separáveis linearmente em algumas dimensões, portanto, esperamos resultados geralmente bons. Vamos testar 6 algoritmos diferentes:

- Regressão Logística (LR)
- Análise Discriminante Linear (LDA)
- K-Nearest Neighbors (KNN).
- Árvores de Classificação e Regressão (CART).
- Gaussian Naive Bayes (NB).
- Support Vector Machines (SVM).

Esta é uma boa mistura de algoritmos lineares simples (LR e LDA) e não lineares (KNN, CART, NB e SVM).

Vamos construir e avaliar nossos modelos:

```
# Algoritmo de verificação do modelo
models = []
models.append(('LR', LogisticRegression(solver='liblinear',
multi_class='ovr'))))
models.append(('LDA', LinearDiscriminantAnalysis()))
models.append(('KNN', KNeighborsClassifier()))
models.append(('CART', DecisionTreeClassifier()))
models.append(('NB', GaussianNB()))
models.append(('SVM', SVC(gamma='auto'))))
# evaluate each model in turn
results = []
names = []
for name, model in models:
    kfold = StratifiedKFold(n_splits=10, random_state=1,
shuffle=True)
    cv_results = cross_val_score(model, X_train,
Y_train, cv=kfold, scoring='accuracy')
    results.append(cv_results)
    names.append(name)
    print('%s: %f (%f)' % (name, cv_results.mean(),
cv_results.std()))
```

## 5.4 Selecionar o melhor modelo

Agora temos 6 modelos e estimativas de precisão para cada um. Precisamos comparar os modelos entre si e selecionar o mais preciso.

Executando o exemplo acima, obtemos os seguintes resultados brutos:

```
LR: 0.941667 (0.065085)
LDA: 0.975000 (0.038188)
KNN: 0.958333 (0.041667)
CART: 0.933333 (0.050000)
NB: 0.950000 (0.055277)
SVM: 0.983333 (0.033333)
```

**Obs.:** Seus resultados podem variar devido à natureza estocástica do algoritmo ou procedimento de avaliação, ou diferenças na precisão numérica. Considere executar o exemplo algumas vezes e compare o resultado médio.

Nesse caso, podemos ver que parece que Support Vector Machines (SVM) tem a maior pontuação de precisão estimada em cerca de 0,98 ou 98%.

Também podemos criar um gráfico dos resultados da avaliação do modelo e comparar a dispersão e a precisão média de cada modelo. Há uma população de medidas de precisão para cada algoritmo porque cada algoritmo foi avaliado 10 vezes (por meio de validação cruzada de 10 vezes).

Uma maneira útil de comparar as amostras de resultados de cada algoritmo é criar um gráfico de caixa e bigode para cada distribuição e comparar as distribuições.

```
# Algoritmo de comparação
pyplot.boxplot(results, labels=names)
pyplot.title('Algoritmo de comparação')
pyplot.show()
```

Podemos ver que os gráficos de caixa e bigode estão espremidos no topo da faixa, com muitas avaliações alcançando 100% de precisão e algumas chegando a 80% de precisão.

```
// inserir a saída aqui
```

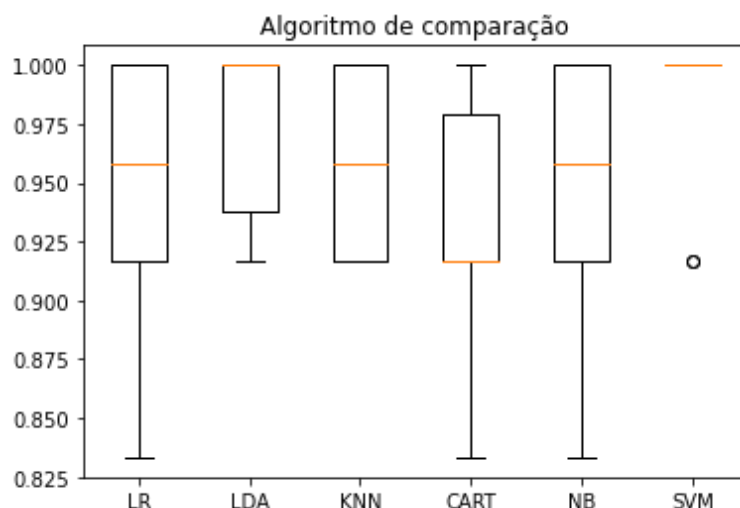


Gráfico de Box e Whisker Comparando Algoritmos de Aprendizado de Máquina no Conjunto de Dados Iris Flowers

## 5.5 O código – até este ponto

Para referência, podemos amarrar todos os elementos anteriores juntos em um único script.

O exemplo completo está listado abaixo:

```
# algoritmo para comparação dos modelos
from pandas import read_csv
from matplotlib import pyplot
from sklearn.model_selection import train_test_split
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import StratifiedKFold
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.naive_bayes import GaussianNB
from sklearn.svm import SVC

# carregando o dataset
url =
"https://raw.githubusercontent.com/jbrownlee/Datasets/master/iris.csv"
names = ['sepal-length', 'sepal-width', 'petal-length', 'petal-width',
'class']
dataset = read_csv(url, names=names)

# Fatando um conjunto para validação do dataset
array = dataset.values
X = array[:,0:4]
y = array[:,4]
X_train, X_validation, Y_train, Y_validation = train_test_split(X, y,
test_size=0.20, random_state=1, shuffle=True)
```

```

# Algoritmo que verifica os modelos
models = []
models.append(('LR', LogisticRegression(solver='liblinear',
multi_class='ovr'))))
models.append(('LDA', LinearDiscriminantAnalysis()))
models.append(('KNN', KNeighborsClassifier()))
models.append(('CART', DecisionTreeClassifier()))
models.append(('NB', GaussianNB()))
models.append(('SVM', SVC(gamma='auto'))))

# avaliando cada um dos modelos ML
results = []
names = []
for name, model in models:
    kfold = StratifiedKFold(n_splits=10, random_state=1,
shuffle=True)
    cv_results = cross_val_score(model, X_train, Y_train, cv=kfold,
scoring='accuracy')
    results.append(cv_results)
    names.append(name)
    print('%s: %f (%f)' % (name, cv_results.mean(),
cv_results.std()))

# Algoritmo para compara-los através do grafico
pyplot.boxplot(results, labels=names)
pyplot.title('Algoritmo de comparação')
pyplot.show()

```

## 6. Previsões (prediction)

Devemos escolher um algoritmo a ser usado para fazer previsões.

Os resultados da seção anterior sugerem que o SVM foi talvez o modelo mais preciso. Usaremos este modelo como nosso modelo final.

Agora queremos ter uma ideia da precisão do modelo em nosso conjunto de validação.

Isso nos dará uma verificação final independente sobre a precisão do melhor modelo. É importante manter um conjunto de validação para o caso de você cometer um deslize durante o treinamento, como overfitting no conjunto de treinamento ou vazamento de dados. Ambas as questões resultarão em um resultado excessivamente otimista.

## 6.1 Fazendo previsões

Podemos ajustar o modelo em todo o conjunto de dados de treinamento e fazer previsões no conjunto de dados de validação.

```
# Fazendo previsões (predictions) na validação do dataset
model = SVC(gamma='auto')
model.fit(X_train, Y_train)
predictions = model.predict(X_validation)
```

## 6.2 Avaliar previsões

Podemos avaliar as previsões comparando-as aos resultados esperados no conjunto de validação e, em seguida, calcular a precisão da classificação, bem como uma matriz de confusão e um relatório de classificação.

```
# avaliando previsões (predictions)
print(accuracy_score(Y_validation, predictions))
print(confusion_matrix(Y_validation, predictions))
print(classification_report(Y_validation, predictions))
```

Podemos ver que a precisão é de 0,966 ou cerca de 96% no conjunto de dados de retenção.

A matriz de confusão fornece uma indicação dos erros cometidos.

Finalmente, o relatório de classificação fornece uma análise de cada classe por precisão, recall, pontuação f1 e suporte, mostrando resultados excelentes (considerando que o conjunto de dados de validação era pequeno).

```
0.9666666666666667
[[11  0  0]
 [ 0 12  1]
 [ 0  0  6]]
```

	precision	recall	f1-score	support
Iris-setosa	1.00	1.00	1.00	11
Iris-versicolor	1.00	0.92	0.96	13
Iris-virginica	0.86	1.00	0.92	6
accuracy			0.97	30
macro avg	0.95	0.97	0.96	30
weighted avg	0.97	0.97	0.97	30

### 6.3 Exemplo Completo

Para referência, podemos amarrar todos os elementos anteriores juntos em um único script.

O exemplo completo está listado abaixo.

```
# fazendo previsões (predictions)
from pandas import read_csv
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix
from sklearn.metrics import accuracy_score
from sklearn.svm import SVC

# carregando o dataset
url =
"https://raw.githubusercontent.com/jbrownlee/Datasets/master/iris.csv"
names = ['sepal-length', 'sepal-width', 'petal-length', 'petal-width',
'class']
dataset = read_csv(url, names=names)

# fatiando os dados para validação - implementar teste e treino
array = dataset.values
X = array[:,0:4]
y = array[:,4]
X_train, X_validation, Y_train, Y_validation = train_test_split(X, y,
test_size=0.20, random_state=1)

# fazendo previsões (predictions) a partir dos dados validos
model = SVC(gamma='auto')
model.fit(X_train, Y_train)
predictions = model.predict(X_validation)

# avaliando as previsões (predictions)
print(accuracy_score(Y_validation, predictions))
print(confusion_matrix(Y_validation, predictions))
print(classification_report(Y_validation, predictions))
```

### Resumo

Neste passo-a-passo implementamos um primeiro projeto de aprendizado de máquina usando Python.

Concluimos um projeto de ponta a ponta, desde o carregamento dos dados até a realização de previsões.



**Referencias:**

<https://docs.scipy.org/doc/numpy-1.10.1/user/whatisnumpy.html>

<https://www.programiz.com/python-programming/matrix>

<https://likegeeks.com/pandas-passo-a-passo/>

<https://likegeeks.com/matplotlib-passo-a-passo/>

<https://likegeeks.com/seaborn-heatmap-passo-a-passo/>

<https://likegeeks.com/python-correlation-matrix/>

<https://realpython.com/working-with-large-excel-files-in-pandas/>

<https://machinelearningmastery.com/machine-learning-in-python-step-by-step/>