

Machine Learning na Prática

Modelos em Python

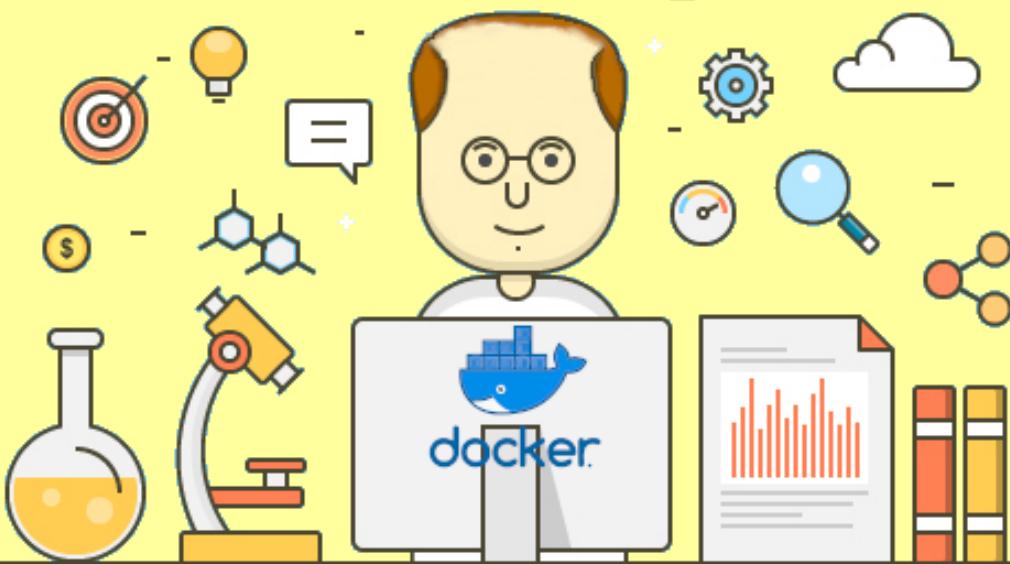
Fernando Anselmo

Copyright © 2020 Fernando Anselmo - v1.0

PUBLICAÇÃO INDEPENDENTE

<http://fernandoanselmo.orgfree.com>

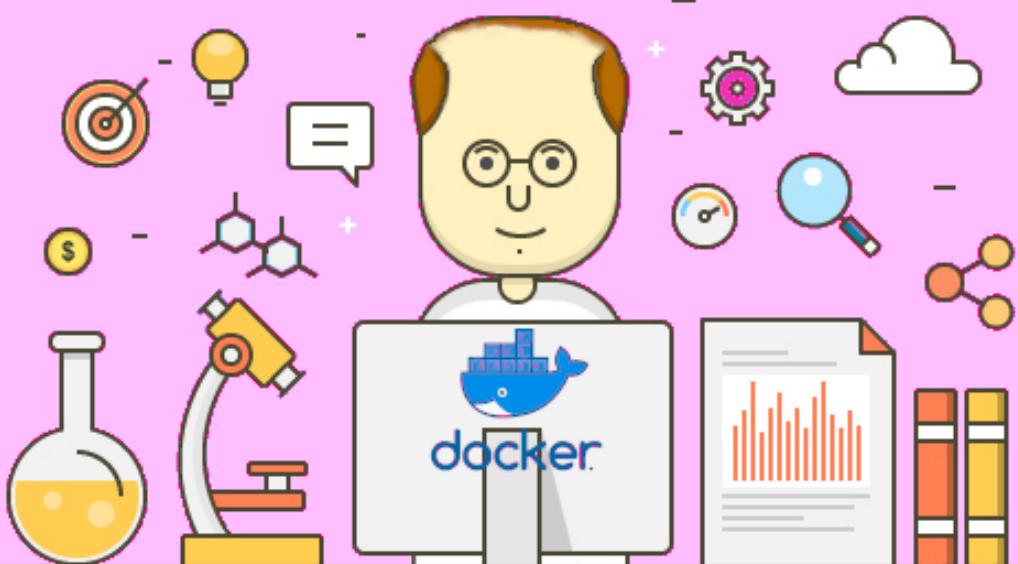
É permitido a total distribuição, cópia e compartilhamento deste arquivo, desde que se preserve os seguintes direitos, conforme a licença da *Creative Commons 3.0*. Logos, ícones e outros itens inseridos nesta obra, são de responsabilidade de seus proprietários. Não possuo a menor intenção em me apropriar da autoria de nenhum artigo de terceiros. Caso não tenha citado a fonte correta de algum texto que coloquei em qualquer seção, basta me enviar um e-mail que farei as devidas retratações, algumas partes podem ter sido cópias (ou baseadas na ideia) de artigos que li na Internet e que me ajudaram a esclarecer muitas dúvidas, considere este como um documento de pesquisa que resolvi compartilhar para ajudar os outros usuários e não é minha intenção tomar crédito de terceiros.



Sumário

1	Modelos Iniciais	5
1.1	K-Means	5
1.2	Aplicação da Técnica	6
1.3	Plotagem do Resultado do Modelo	8
1.4	K-Nearest Neighbors	9
1.4.1	Predição com K-Nearest Neighbors	11
1.5	Análise de Cluster	12
1.6	Clusterização Hierárquica	15
1.6.1	Clusterização Hierárquica versus K-Nearest Neighbors	18
1.7	Regressão Linear	19
1.7.1	Aplicar a Regressão Linear	20
1.7.2	Regressão Linear com mais de um Preditor	21
1.7.3	Regressão Linear e Limpeza dos Dados	22
1.7.4	Separação e treino	24

1.8	Árvore de Decisão	25
1.8.1	Critérios Gini e Entropia	27
1.9	Floresta Aleatória	29
A	Considerações Finais	33
A.1	Sobre o Autor	34



1. Modelos Iniciais

F Na vida, não existe nada a temer, mas a entender. (Marie Curie - Cientista e Vencedora 2 vezes do Prêmio Nobel)

1.1 K-Means

Acredito que K-Means seja o modelo mais simples para começarmos, este é um algoritmo de Aprendizado Não Supervisionado, ou seja, não necessita de atributos alvo para agir, sua função é de separar as observações em grupos de modo que possamos observar melhor os dados.

Sendo assim, nosso problema para usar esse algoritmo é exatamente achar esse k ideal de modo que os grupos sejam separados coerentemente. Para isso existe uma técnica interessante chamada "Técnica do Cotovelo"(Elbow Technique).

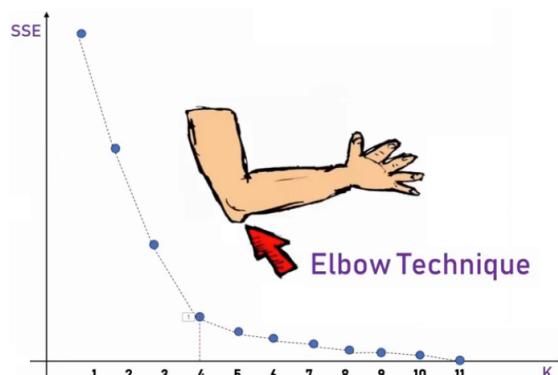


Figura 1.1: Técnica do Cotovelo

Exatamente na posição 4 existe uma "quebra" para passar ao próximo valor, usamos para definir essa quebra o SSE¹ (*Sum Squared Error*).

1.2 Aplicação da Técnica

Para achar o k ideal vamos ativar nosso JupyterLab personalizado que criamos com o Docker e na primeira célula importamos as bibliotecas necessárias:

```

1 import pandas as pd
2 import numpy as np
3 from sklearn.preprocessing import scale
4 from sklearn.cluster import KMeans
5 from matplotlib import pyplot as plt
6
7 %matplotlib inline

```

Importamos a biblioteca Pandas e a Numpy para manipularmos os dados, a Scikit-Learn para usarmos o modelo K-Means e Matplot para vermos o resultado em um gráfico. A última linha é utilizada para mostrar os gráficos no Jupyter. Próximo passo consiste em ler os dados, baixamos o arquivo **gameML.csv** e na posição do nosso arquivo **.ipynb** criamos uma subpasta chamada **bases** e nesta colocamos o arquivo.

```

1 df = pd.read_csv('bases/gameML.csv', delimiter=';')
2 df.head()

```

E como resultado da execução dessa célula devemos ter:

	Nome	Idade	Salário
0	Daenerys Targaryen	27	70000
1	Jon Snow	29	90000
2	Gregor Ciegane	29	61000
3	Arya Stark	28	60000
4	Tyrion Lannister	42	150000

Figura 1.2: Idades e Salários da Empresa GameML

No arquivo existem 3 campos: nome do funcionário, idade e salário, se plotarmos os dados entre idade e salário em gráfico:

```
1 plt.scatter(df['Idade'], df['Salário'])
```

Obtemos como resultado:

¹Soma Residual dos Quadrados, é a soma dos resíduos elevado por 2. É uma medida da discrepância entre os dados e um modelo de estimativa. Um valor pequeno SSE indica um ajuste apertado do modelo aos dados.

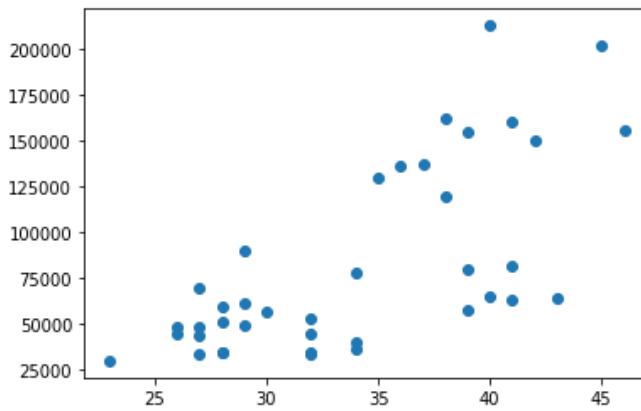


Figura 1.3: Idades e Salários da Empresa GameML

Quantos grupos de dados podemos distinguir? Para localizarmos a quantidade ideal aplicamos a técnica do cotovelo que consiste de:

```

1 k_rng = range(1,10)
2 sse = []
3 for k in k_rng:
4     km = KMeans(n_clusters=k)
5     km.fit(df[['Idade', 'Salário']])
6     sse.append(km.inertia_)
7 plt.xlabel('K')
8 plt.ylabel('SSE (Sum Squared Error)')
9 plt.plot(k_rng, sse)

```

Criar um range de 1 a 10 (um simples número máximo de possíveis *clusters*), para cada valor treinamos o modelo com as variáveis e obtemos o valor do atributo **inércia**. O algoritmo agrupa dados e procura separar amostras em n grupos de igual variação, minimizando um critério conhecido como inércia ou **RSS** dentro do *cluster*. O que estamos fazendo na prática é colocar o valor 1 para o **k** e guardar esse valor, em seguida o valor 2 e assim sucessivamente. Por fim plotamos esse valor em um gráfico e obtemos como resultado:

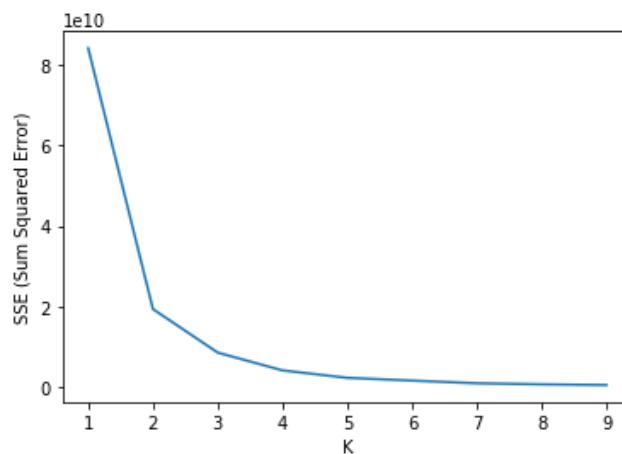


Figura 1.4: Gráfico com os valores de Inércia

E vemos nosso "cotovelo" da curva bem na posição **3**, marcando assim o número ideal de clusters.

1.3 Plotagem do Resultado do Modelo

Um detalhe interessante que para usarmos o algoritmo K-Means, devemos colocar os dados em "escala", vamos tentar usar o modelo sem proceder dessa forma:

```
1 km = KMeans(n_clusters=3)
2 y_predict = km.fit_predict(df[['Idade', 'Salário']])
3 df['ypred'] = y_predict
4 df.head()
```

Já sabemos que o valor de 3 clusters é o ideal, então realizamos o treinamento com os atributos Idade e Salário para montarmos um novo atributo com o resultado dessa predição (somente para que o gráfico apareça separado por cores). E plotamos o gráfico:

```
1 cores = np.array(['green', 'red', 'blue'])
2 plt.scatter(x=df['Idade'],
3 y=df['Salário'],
4 c=cores[df.ypred], s=50)
```

Obtemos como resultado:

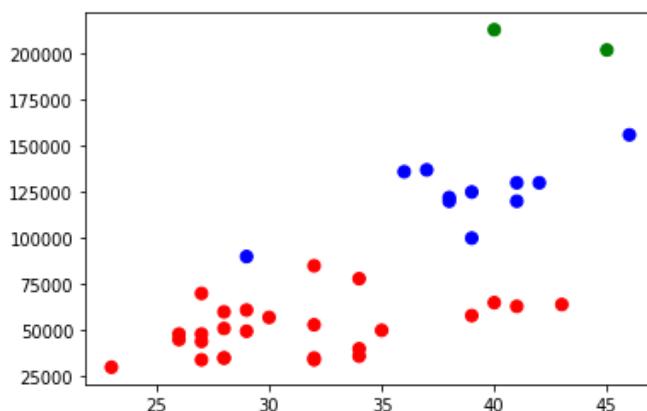


Figura 1.5: Separados por Grupo

E parece que obtemos algo bem errado com alguns *outliers* aparecendo, observamos o ponto azul no meio dos vermelhos e um outro azul isolado perto dos verdes. Então antes de treinarmos esse algoritmo devemos colocar os dados na mesma escala, isso é feito assim:

```
1 df['Salário'] = scale(df.Salário)
2 df['Idade'] = scale(df.Idade)
3 df.head()
```

Os atributos **idade** e **salário** possuem valores bem diferentes e distantes e isso gera problemas para nosso resultado final, colocar em escala e aproximar (sem modificar o resultado final) os valores seria algo criar

um modelo de um prédio porém mantendo as mesmas proporções do prédio original.

A função da **Scikit-Learn** que realiza este processo é chamada *scale()* e colocamos em escala os atributos se visualizarmos nossos dados agora veremos que o atributo **idade** possui valores entre -2 e 2 enquanto que **salário** entre -1.5 e 3 (são diferentes exatamente para manter a proporcionalidade). Retornamos ao mesmo processo de treinamento:

```

1 km = KMeans(n_clusters=3)
2 y_predict = km.fit_predict(df[['Idade', 'Salário']])
3 df['ypred'] = y_predict
4 df.head()

```

Plotamos novamente o gráfico e agora como resultado obtemos:

```

1 cores = np.array(['green', 'red', 'blue'])
2 plt.scatter(x=df['Idade'],
3 y=df['Salário'],
4 c=cores[df.ypred], s=50)

```

E obtemos como resultado:

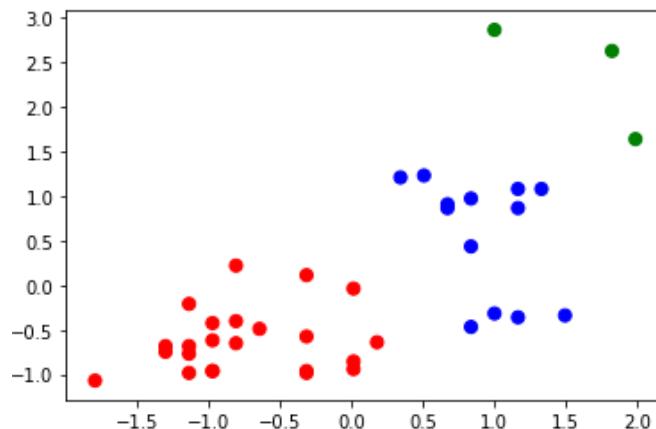


Figura 1.6: Separados por Grupo em Escala

Que é um resultado bem mais coerente.

1.4 K-Nearest Neighbors

Ou simplesmente KNN. Modelos assim existem pois muitas pessoas pensam que separar em *clusters* não auxilia na predição, pois bem nosso próximo modelo é um supervisionado e destinado a Predição por Clusterização (ou se prefere por proximidade dos grupos). KNN que normalmente é usado para a predição de imagens como: Isso é um Gato? Ou não é um Gato? Porém ao invés de imagens, vamos usar uma base bem conhecida chamada **Flores Íris** para entendermos seu comportamento.

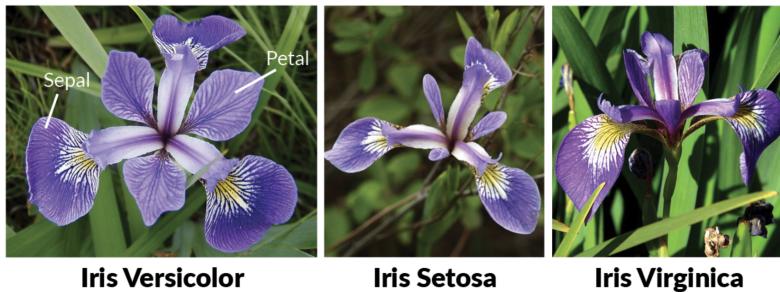


Figura 1.7: Flores Iris

Nessa base existem três espécies separadas: Versicolor, Setosa e Virgíñica. E para distingui-las utilizamos 2 medidas da sépala e da pétala (largura e altura de cada). O problema é que algumas espécies causam as maiores confusões em nossos modelos. Para realizarmos uma predição sobre essa base importamos nossas bibliotecas:

```

1 import numpy as np
2 from matplotlib import pyplot as plt
3 from sklearn import datasets
4 from sklearn.model_selection import train_test_split
5 from sklearn import neighbors
6
7 %matplotlib inline

```

Usamos a **NumPy** para gerenciamento dos dados. **Matplotlib** para plotarmos os gráficos. Da **Scikit-Learn** obtemos os nossos dados através do pacote **datasets** e para separar uma massa de teste contamos com o *train_test_split*. E a *neighbors* contém o nosso algoritmo. O próximo passo consiste na preparação dos dados:

```

1 iris = datasets.load_iris()
2 X, y = iris.data, iris.target
3
4 X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
   random_state=1234)

```

O método *load_iris()* traz a nossa base em uma matriz de dados. Nossa base está dividida em *data* que contém os *features* preditores (tamanho e largura da sépala e tamanho e largura da pétala, que colocaremos em *X*) e *target, feature* que contém a definição da espécie (0 representa **Setosa**, 1 para **Versicolor** e 2 para **Virgíñica** que colocaremos em *y*). Usamos o método *train_test_split* para retirar 20% dos dados como amostra de teste e obtemos quatro agrupamentos:

- **X_train**, com os dados para treino do algoritmo.
- **X_test**, com os dados para teste.
- **y_train**, com o resultado para o treino.
- **y_test**, com o resultado para o teste.

Com nossos dados preparados vamos treinar o modelo:

```

1 clf = neighbors.KNeighborsClassifier()
2 clf.fit(X_train, y_train)

```

```
3 print(clf.score(X_test, y_test))
```

E conseguimos uma boa acurácia com incríveis 96% de precisão, agora é vermos na prática como isso funciona.

1.4.1 Predição com K-Nearest Neighbors

Primeiro vamos mostrar os dados:

```
1 cores = np.array(['green', 'red', 'blue'])
2 subplot1 = plt.scatter(x=x[:, 0], y=x[:, 1], c=cores[y], s=50)
```

Pegamos as duas primeiras variáveis tamanho e largura da sépala e obtemos como resultado:

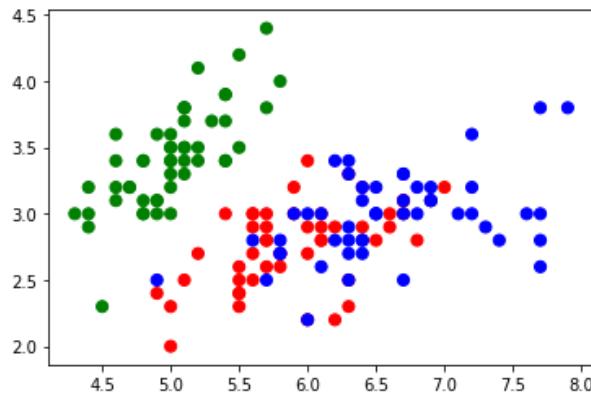


Figura 1.8: Comparar tamanho e largura da Sépala

Não nos perdemos nas cores **Verde** é Setosa, **Vermelho** é Versicolor e **Azul** é Virgínica. Agora vamos pensar em um ponto qualquer nesse espaço, por exemplo:

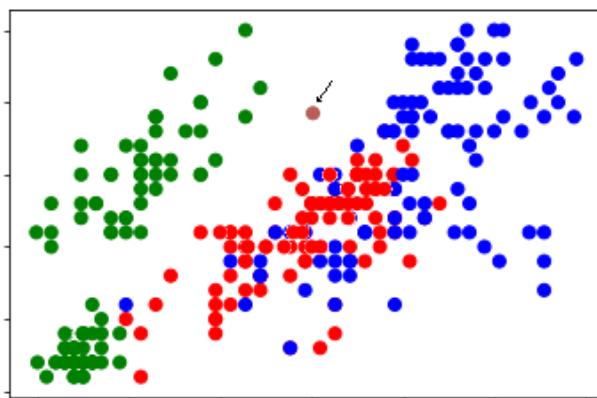


Figura 1.9: Localizar o Ponto Roxo

O ponto roxo fica na interseção do 4º valor de X e y qual cor real ele seria? Observamos no gráfico anterior que os pontos são 6,0 e 4,0 porém nos falta o valor para mais dois atributos tamanho e largura da pétala:

```

1 cores = np.array(['green', 'red', 'blue'])
2 subplot1 = plt.scatter(x=X[:, 2], y=X[:, 3], c=cores[y], s=50)

```

E obtemos como resultado:

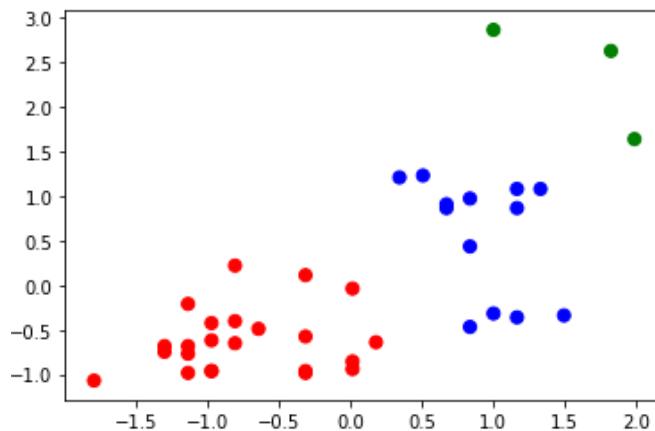


Figura 1.10: Comparar tamanho e largura da Pétala

E verificamos que na interseção do 4º valor de X e y obtemos os valores 4,0 e 2,0. Agora que obtemos os quatro valores podemos realizar uma predição:

```

1 predicao = clf.predict([[6.0, 4.0, 4.0, 2.0]])
2 print(predicao)

```

E resulta que o modelo prevê que é do tipo [1], ou seja, um ponto vermelho da espécie **Versicolor**.

1.5 Análise de Cluster

Então sabemos agora que ambos modelos K-Means e KNN trabalham utilizando *clusters* (agrupamentos) sendo que o primeiro é do tipo não supervisionado destinado a separação com base em um número de centroides (k) presentes e os valores médios mais próximos (isso representa uma distância Euclidiana entre as observações). Porém é necessário colocar os dados em escala para verificar se não ocorre nenhuma perturbação nesse centroide. Vamos importar algumas bibliotecas para realizarmos mais testes:

```

1 import numpy as np
2 import pandas as pd
3 import seaborn as sns
4 import matplotlib.pyplot as plt
5 from sklearn import datasets
6 from sklearn.preprocessing import scale
7 from sklearn.cluster import KMeans
8 from sklearn.metrics import classification_report
9
10 %matplotlib inline

```

Já passamos por todas e não desejo ser repetitivo porém dessa vez vamos utilizar a Pandas para manipular os dados e a classe *metrics* da SciKit-Learn para mostrar o comportamento do nosso modelo. Iremos continuar usando a base Iris e construímos um *DataFrame* somente com os dados dos atributos preditores porém guardaremos o atributo alvo para verificar como nosso modelo se comportou:

```

1 iris = datasets.load_iris()
2 X = scale(iris.data)
3 y = pd.DataFrame(iris.target)
4 y.columns = ['Targets']
5 variable_names = iris.feature_names
6 iris_df = pd.DataFrame(iris.data)
7 iris_df.columns = variable_names
8 iris_df.head()

```

E nosso *DataFrame* se apresenta da seguinte maneira:

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)
0	5.1	3.5	1.4	0.2
1	4.9	3.0	1.4	0.2
2	4.7	3.2	1.3	0.2
3	4.6	3.1	1.5	0.2
4	5.0	3.6	1.4	0.2

Figura 1.11: DataFrame com os dados dos Atributos Preditores

O próximo passo é construir e treinar nosso modelo:

```

1 clustering = KMeans(n_clusters=3, random_state=5).fit(X)

```

Normalmente para treinar um modelo passamos dois conjuntos de dados, porém o K-Means só recebe um único conjunto, exatamente por não realizar previsões precisa apenas dos dados para separá-los em conjuntos. Mas como será que foi seu comportamento? Descobrimos isso comparando dois gráficos:

```

1 cores = np.array(['green', 'red', 'blue'])
2 relabel = np.choose(clustering.labels_, [1, 0, 2]).astype(np.int64)
3 plt.figure(figsize = [15, 5])
4
5 plt.subplot(1, 4, 1)
6 plt.scatter(x=iris_df['petal length (cm)'],
7 y=iris_df['petal width (cm)'],
8 c=cores[iris.target], s=50)
9 plt.title('Real (Pétala)')
10
11 plt.subplot(1, 4, 2)
12 plt.scatter(x=iris_df['petal length (cm)'],
13 y=iris_df['petal width (cm)'],
14 c=relabel, s=50)
15 plt.title('KMeans (Pétala)')
16
17 plt.subplot(1, 4, 3)
18 plt.scatter(x=iris_df['sepal length (cm)'],
19 y=iris_df['sepal width (cm)'],
20 c=cores[iris.target], s=50)

```

```

21 plt.title('Real (Sépala)')
22
23 plt.subplot(1, 4, 1)
24 plt.scatter(x=iris_df['sepal length (cm)'],
25 y=iris_df['sepal width (cm)'],
26 c=cores[relabel], s=50)
27 plt.title('KMeans (Sépala)')
28
29 plt.show()

```

Usamos os mesmos conjuntos de cores para cada espécie, obtemos quatro gráficos comparativos: 1º largura e altura da Pétala e a cor será mostrada com base em nosso atributo alvo (ou seja o valor real), 2º o que o modelo achou que seria o correto, 3º largura e altura da Sépala e o 4º novamente como o modelo separou. E obtemos como resultado:

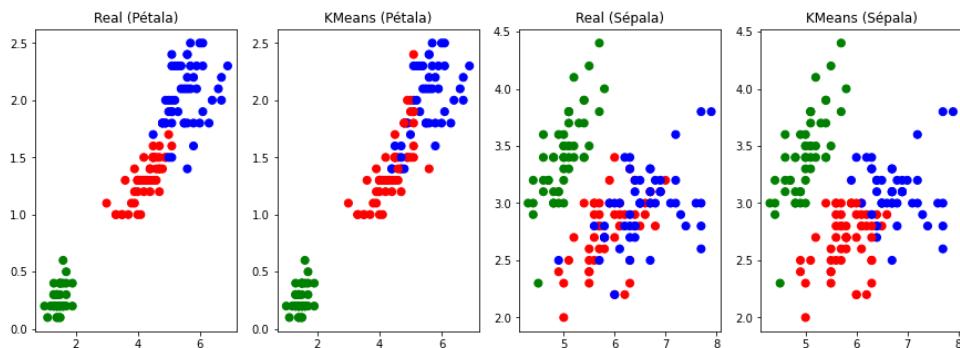


Figura 1.12: Comparativo entre o Real e o KMeans

Para pétala o **K-Means** quase acertou a posição de cada espécie, porém para Sépala aconteceram as maiores confusões, isso se deve ao fato do centroide. Para melhor avaliarmos nosso modelo precisamos de mais medidas: *Precision* (precisão) é a medida de relevância do modelo, *Recall* (revocação ou sensibilidade) se trata da medida de completude do modelo e *F1 Score* se trata de uma média ponderada entre *precision* e *recall*. Podemos obtê-las da seguinte forma:

```

1 metricas = classification_report(y, relabel)
2 print(metricas)

```

Obtemos como resultado:

	precision	recall	f1-score	support
0	1.00	1.00	1.00	50
1	0.74	0.78	0.76	50
2	0.77	0.72	0.74	50
accuracy			0.83	150
macro avg	0.83	0.83	0.83	150
weighted avg	0.83	0.83	0.83	150

Figura 1.13: Relatório de Performance do K-Means

Precision - É a razão entre as observações positivas previstas corretamente e o total de observações positivas previstas. Calculada com a fórmula: $TP \div (TP + FP)$.

Recall - É a razão entre as observações positivas previstas corretamente e todas as observações da classe real. Calculada com a fórmula: $TP \div (TP + FN)$

F1 Score - Essa pontuação leva em consideração tanto os falsos positivos quanto os negativos. Intuitivamente, não é tão fácil entender como precisão, mas F1 é geralmente mais útil que *precision*, especialmente se estivermos com uma distribuição de classe desigual. $2 \times (recall \times precision) \div (recall + precision)$

Acurácia funciona melhor se os falsos positivos e negativos tiverem um custo semelhante. Se o custo for muito diferente, é melhor olharmos essas métricas.

1.6 Clusterização Hierárquica

Este é um modelo alternativo ao particionamento de *cluster* no conjunto de dados, pode ser aplicado para encontrar a distância entre cada ponto e seus vizinhos mais próximos e conectá-lo de forma ideal. Podemos mostrar o número de subgrupos com o auxílio de um Dendrograma².

É útil pois não existe necessidade de especificar o número de *clusters* (ou K) antes da análise e o dendrograma fornece uma representação visual desses. Vamos trazer para o conjunto de bibliotecas visto anteriormente mais três:

```

1 from scipy.cluster.hierarchy import dendrogram, linkage
2 from sklearn.cluster import AgglomerativeClustering
3 from sklearn.metrics import accuracy_score

```

Para este exemplo vamos utilizar outra base que está contida no arquivo **mtcars.csv** (trazer essa para a subpasta **/base**). E carregamos os dados do seguinte modo:

```

1 carros = pd.read_csv('bases/mtcars.csv')
2 carros.columns = ['name', 'mpg', 'cil', 'disp', 'hp', 'drat', 'wt', 'qsec', 'vs',
   'am', 'gear', 'carb']
3 X = carros[['mpg', 'disp', 'hp', 'wt']].values
4 y = carros['am'].values

```

Essa base contém 32 modelos de carros com os seguintes atributos: Nome, autonomia em milhas por galão, número de cilindros, deslocamento (medida de poder do carro em polegada cúbica), cavalos de força, relação do eixo traseiro, peso (em libras), eficiência do gasto de combustível (por 1/4 milha), motor (0 = V-shaped, 1 = straight), câmbio (0 = automática, 1 = manual), total de marchas e carburadores. Porém para não trabalharmos com tantos atributos vamos usar somente: consumo de gasolina (mpg), deslocamento (disp), cavalos de força (hp) e peso (wt) e o nosso objetivo e descobrir se o carro possui um câmbio manual ou automático.

Podemos montar o dendrograma do seguinte modo:

```

1 z = linkage(X, 'ward')
2 dendrogram(z, truncate_mode='lastp', p=12, leaf_rotation=45, leaf_font_size=15,

```

²É um gráfico em formato de árvore que mostra visualmente os relacionamentos entre as observações.

```

show_contracted=True)

3
4 plt.title('Dendograma')
5 plt.xlabel('Tamanho do Cluster')
6 plt.ylabel('Distância')
7 plt.axhline(y=500)
8 plt.axhline(y=150)
9 plt.show()

```

Obtemos como resultado:

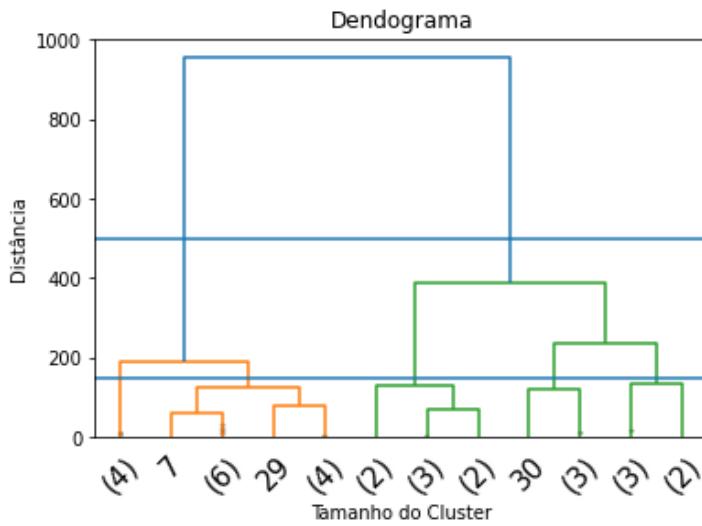


Figura 1.14: Dendrograma dos tamanhos do Cluster

O dendrograma mostra como cada *cluster* é composto e desenha um link em forma de U entre cada cluster e seus filhos. A parte superior indica uma mesclagem. Cada perna indica quais foram mesclados. O comprimento das pernas e do U representa a distância entre os filhos.

Para mesclar recursivamente o par de *clusters* e aumentar minimamente a distância de ligação utilizamos a função `AgglomerativeClustering()`. Essa possui dois parâmetros básicos: *affinity* e *linkage*.

affinity: métrica utilizada para calcular a ligação. Possui as seguintes opções:

- *euclidean* - é o único que aceita o parâmetro *linkage* como *ward*. Refere-se a distância euclidiana que pode ser provada pela aplicação repetida do teorema de Pitágoras.
- *l1* - critério de erro absoluto.
- *l2* - critério de erros quadrados (lembra do RSS).
- *manhattan* - distância euclidiana ao quadrado.
- *cosine* - também chamada de Similaridade do Cosseno. É a distância do cosseno entre duas variáveis.
- *precomputed* - necessita de uma matriz de distância (em vez de similaridade) como entrada para o método de ajuste, pois X será considerado uma matriz.

linkage: define qual o critério de ligação usar. Determina qual distância usar entre os conjuntos de observação. Possui as seguintes opções:

- *ward* - minimiza a variação dos *clusters* que estão sendo mesclados.
- *average* - média das distâncias de cada observação dos conjuntos.
- *complete* - distâncias máximas entre todas as observações dos dois conjuntos.
- *single* - mínimo das distâncias entre todas as observações dos dois conjuntos.

Como escolher os parâmetros ideais? Fácil, testemos várias combinações e veremos qual possui uma melhor acurácia para os dados que estamos tratando:

```

1 hclusters1 = AgglomerativeClustering(n_clusters=2, affinity='euclidean',
2     linkage='ward').fit(X)
3 print('Método 1:', accuracy_score(y, hclusters1.labels_))
4
5 hclusters2 = AgglomerativeClustering(n_clusters=2, affinity='euclidean',
6     linkage='complete').fit(X)
7 print('Método 2:', accuracy_score(y, hclusters2.labels_))
8
9 hclusters3 = AgglomerativeClustering(n_clusters=2, affinity='euclidean',
10    linkage='average').fit(X)
11 print('Método 3:', accuracy_score(y, hclusters3.labels_))
12
13 hclusters4 = AgglomerativeClustering(n_clusters=2, affinity='manhattan',
14    linkage='single').fit(X)
15 print('Método 4:', accuracy_score(y, hclusters4.labels_))
16
17 hclusters5 = AgglomerativeClustering(n_clusters=2, affinity='manhattan',
18    linkage='complete').fit(X)
19 print('Método 5:', accuracy_score(y, hclusters5.labels_))
20
21 hclusters6 = AgglomerativeClustering(n_clusters=2, affinity='manhattan',
22    linkage='average').fit(X)
23 print('Método 6:', accuracy_score(y, hclusters6.labels_))
24
25 hclusters7 = AgglomerativeClustering(n_clusters=2, affinity='cosine',
26    linkage='single').fit(X)
27 print('Método 7:', accuracy_score(y, hclusters7.labels_))
28
29 hclusters8 = AgglomerativeClustering(n_clusters=2, affinity='cosine',
30    linkage='complete').fit(X)
31 print('Método 8:', accuracy_score(y, hclusters8.labels_))
32
33 hclusters9 = AgglomerativeClustering(n_clusters=2, affinity='cosine',
34    linkage='average').fit(X)
35 print('Método 9:', accuracy_score(y, hclusters9.labels_))

```

Obtemos como resultado:

Método 1: 0.78125
 Método 2: 0.4375
 Método 3: 0.78125
 Método 4: 0.625
 Método 5: 0.71875
 Método 6: 0.71875
 Método 7: 0.3125
 Método 8: 0.28125

```
Método 9: 0.1875
```

Assim para esse caso *Euclidian/Ward* ou *Manhattan/Complete* são os que melhor responderam ao nosso conjunto de dados com uma acurácia de 78,12%. Podemos inclusive tirar um relatório mais completo (como já vimos):

```
1 print(classification_report(y, hclusters1.labels_))
```

Obtemos como resultado:

	precision	recall	f1-score	support
0	0.88	0.74	0.80	19
1	0.69	0.85	0.76	13
accuracy			0.78	32
macro avg	0.78	0.79	0.78	32
weighted avg	0.80	0.78	0.78	32

Figura 1.15: Relatório de Performance da Clusterização Hierárquica

Só que ficou uma pergunta no ar, esse método se comporta melhor que um modelo de clusterização preditivo como o KNN?

1.6.1 Clusterização Hierárquica versus K-Nearest Neighbors

Para verificar como o KNN se comporta com os dados dos carros adicionamos mais quatro bibliotecas:

```
1 from sklearn.neighbors import KNeighborsClassifier
2 from sklearn import preprocessing
3 from sklearn.model_selection import train_test_split
4 from sklearn.metrics import classification_report
```

Como já obtemos nossos dados, vamos apenas separá-los em bases de treino e teste:

```
1 X = preprocessing.scale(X)
2 X_treino, X_teste, y_treino, y_teste = train_test_split(X, y, test_size=.20,
   random_state=17)
```

Porém devemos sempre lembrar que os modelos de clusterização trabalham melhor quando os dados estão em escala, assim acertamos os atributos preditores antes de realizar a separação de 80% dos dados para treino e 20% para teste.

```
1 clf = KNeighborsClassifier()
2 clf.fit(X_treino, y_treino)
```

Treinamos nosso modelo e podemos avaliar o resultado:

```
1 y_predito = clf.predict(X_teste)
2 print(classification_report(y_teste, y_predito))
```

Obtemos como resultado:

	precision	recall	f1-score	support
0	0.80	1.00	0.89	4
1	1.00	0.67	0.80	3
accuracy			0.86	7
macro avg	0.90	0.83	0.84	7
weighted avg	0.89	0.86	0.85	7

Figura 1.16: Relatório de Performance do K-Nearest Neighbors

E na média percebemos que este se comporta melhor pois atinge resultados acima dos 80%.

1.7 Regressão Linear

A regressão linear tenta modelar o relacionamento entre dois atributos, através de ajustes sob uma equação linear dos dados observados. Um atributo é considerado **explicativo** e a outro **dependente**. Para simplificar um pouco, é uma técnica que utiliza valores de entrada para predizer os de saída (como por exemplo, prever o crescimento da população de um País) através da aplicação dos coeficientes (também chamados de peso) da equação linear.

Comecemos com a importação das bibliotecas que necessitamos:

```

1 import pandas as pd
2 import numpy as np
3 from matplotlib import pyplot as plt
4 from sklearn.linear_model import LinearRegression
5
6 %matplotlib inline

```

A classe *Linear Model* da *Scikit-Learn* o método *LinearRegression* para realizar nosso trabalho. Baixar a base de dados **PopBrasil.csv** que contém as observações de Crescimento da População Brasileira.

```

1 df = pd.read_csv('bases/PopBrasil.csv')
2 df.head()

```

Obtemos como resultado:

Ano	Populacao
0	1960
1	72179226
2	1961
3	74311343
4	1962
5	76514328
6	1963
7	78772657
8	1964
9	81064571

Figura 1.17: Dados da População Brasileira

Devemos saber que o modelo trabalha com a relação entre atributos numéricos: explanatórios X e

dependentes y . Utiliza somente esse tipo devido aos ajustes matemáticos que são realizados e os pesos criados conforme a função minimiza os erros. Nossas observações são bem simples: obtemos atributos numéricos, "Ano" e "População". Para entendermos o relacionamento entre os atributos, plotamos esses em um gráfico:

```

1 plt.xlabel('Ano')
2 plt.ylabel('Quantidade da População')
3 plt.scatter(df.Anو, df.Populacao, color='red', marker='+')
```

Obtemos como resultado:

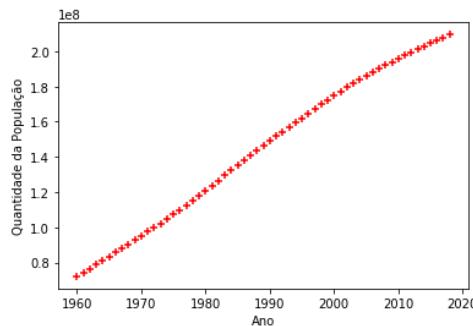


Figura 1.18: Dados da População Brasileira

E esta é a parte mais importante na execução desse modelo, a medida que alteramos o valor de "Ano" o valor de "População" também é afetado, ou seja, existe um relacionamento linear. Essa é a premissa básica para se usar este algoritmo, o relacionamento forte entre os atributos deve existir.

1.7.1 Aplicar a Regressão Linear

Agora que obtemos nossos atributos conferidos, basta treinarmos nosso modelo e obtermos nossa previsão:

```

1 reg = LinearRegression()
2 reg.fit(df[['Ano']], df.Populacao)
3 prev = reg.predict([[2020]])
4 print("Previsão 2020 é: %d" % prev)
```

E obtemos a predição da população brasileira para o ano de 2020, que é 221.322.254 de habitantes. Como a magia acontece? Pura matemática que é fornecida pela seguinte fórmula:

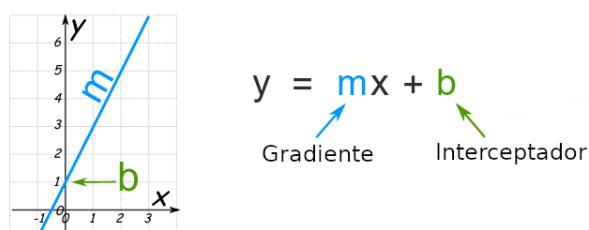


Figura 1.19: Base da Regressão Linear

Dica 1.1: Para saber mais. Se deseja conhecer mais sobre o assunto, visite a página: <https://www.mathsisfun.com/algebra/linear-equations.html> aonde se obtém uma explicação mais completa.

E podemos reproduzir esse resultado pois o objeto treinado nos fornece tanto o valor do Gradiente (`coef_[0]`) quanto do Interceptador (`intercept_`). Então:

```
1 m = reg.coef_[0]
2 b = reg.intercept_
3 prev2020 = m * 2020 + b
4 print("Previsão 2020 é: %d" % prev2020)
```

E obtemos exatamente o mesmo resultado. Podemos traçar a "Reta da Regressão Linear", pois o modelo consegue predizer os resultados de cada ano:

```
1 plt.xlabel('Ano')
2 plt.ylabel('Quantidade da População')
3 plt.scatter(df.Ano, df.Populacao, color='red', marker='+')
4 plt.plot(df.Ano, reg.predict(df[['Ano']]), color='blue')
```

Obtemos como resultado:

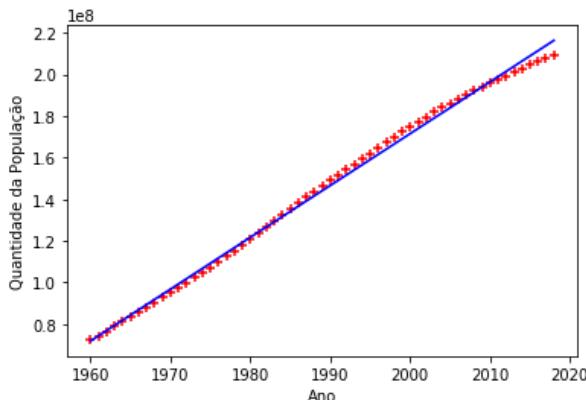


Figura 1.20: Dados da População Brasileira com a Previsão

Vamos praticar nossos novos "poderes de futurólogo", junto a essa base encontramos outra chamada ExpecVida.csv, com ela, tente prever qual será a Expectativa de Vida do brasileiro no ano de 2020.

1.7.2 Regressão Linear com mais de um Preditor

Vimos como usar o modelo de Regressão Linear, porém apenas a título de facilitação do entendimento, somente um atributo preditor. Mas o que acontece quando o alvo é influenciado por mais de um preditor? Vamos entender na prática como isso acontece.

Pensemos em um caso do Varejo, vamos utilizar um conjunto de observações chamado **marketSales.csv** que como o nome sugere, são compostos por transações de vendas. Sabemos que várias coisas influenciam

a saída de um determinado produto, tais como, o grau de visibilidade, peso, se possui muita ou pouca quantidade de gordura, tamanho do mercado ou outros.

Começamos com a importação da bibliotecas necessárias:

```

1 import pandas as pd
2 from sklearn.linear_model import LinearRegression
3 from sklearn.preprocessing import LabelEncoder
4 from sklearn.model_selection import train_test_split
5 from matplotlib import pyplot as plt
6
7 %matplotlib inline

```

E ler nossa base de dados:

```

1 df = pd.read_csv('bases/marketSales.csv')
2 df.head()

```

Até o momento nada de novo, nosso problema começa ao repararmos nas observações:

	Item_Identifier	Item_Weight	Item_Fat_Content	Item_Visibility	Item_Type	Item_MRP	Outlet_Identifier	Outlet_Establishment_Year
0	FDA15	9.30	Low Fat	0.016047	Dairy	249.8092	OUT049	1999
1	DRC01	5.92	Regular	0.019278	Soft Drinks	48.2692	OUT018	2009
2	FDN15	17.50	Low Fat	0.016760	Meat	141.6180	OUT049	1999
3	FDX07	19.20	Regular	0.000000	Fruits and Vegetables	182.0950	OUT010	1998
4	NCD19	8.93	Low Fat	0.000000	Household	53.8614	OUT013	1987

Figura 1.21: Observações sobre Vendas de Produtos

Sabemos que os modelos de regressão só trabalham com tipos numéricos, muito pior existe o caso de nulos entre algumas outras inconsistências nessas 14.204 observações.

1.7.3 Regressão Linear e Limpeza dos Dados

Sejamos francos, maior parte de trabalho do Cientista de Dados é arrumar os dados que sofridamente conseguiu para realizar o trabalho, então começaremos a compreender como uma parte disso funciona. Primeiro detalhe vamos tratar os atributos indesejáveis, nulos e que não contribuem em absolutamente em nada para o aumento/diminuição das vendas. Atributos como o código identificador do produto (*Item_Identifier*) e do mercado (*Outlet_Identifier*) - por esse motivo que o Cientista de Dados deve entender do negócio.

Ao verificarmos a função *info()* descobrimos ainda que o atributo alvo (*Item_Outlet_Sales*) que indica a quantidade de produtos vendidos possui dados nulos (ou seja, também não servem para previsão).

```

1 df = df.drop(df[df['Item_Outlet_Sales'].isnull()].index)
2 df = df.drop(columns=['Item_Identifier', 'Outlet_Identifier'], axis=1)

```

Cuidado pois se aplicamos um corte seco como: `df.dropna(how='any', inplace=True)` obtemos somente 4.650 observações (devido a eliminação dos valores nulos contidos em outros atributos) - ou seja

perdemos quase 10.000 observações. Lembrar que o tratamento dos nulos deve ser cirúrgico e criterioso. Ao aplicar o corte corretamente somente do atributo alvo ficamos com 8.523 observações. Além disso removemos os preditores que não serviam.

Nosso próximo problema com nulos é nos atributos: peso do item (*Item_Weight*) e tamanho da loja (*Outlet_Size*). Em um caso de dados real devemos procurar preencher esses valores solicitando a informação necessária aos responsáveis, porém para fins desse trabalho iremos remover essa colunas também.

```
1 df = df.drop(columns=['Item_Weight', 'Outlet_Size'], axis=1)
```

Não obtemos mais a presença de nulos, mas ainda existem problemas, precisamos verificar os atributos não numéricos das observações, isto é: conteúdo de gordura (*Item_Fat_Content*), tipo do item (*Item_Type*), localização da loja (*Outlet_Location_Type*) e tipo da loja (*Outlet_Type*). Para isso:

```
1 print("Gordura:", df['Item_Fat_Content'].unique())
2 print("Tipo:", df['Item_Type'].unique())
3 print("Loc. Loja:", df['Outlet_Location_Type'].unique())
4 print("Tipo Loja:", df['Outlet_Type'].unique())
```

O atributo *Item_Fat_Content* possui uma faixa com os seguintes valores: '*LF*', '*Low Fat*', '*Regular*', '*low fat*' ou '*reg*'. Obviamente só existem dois tipos: '*Low Fat*' e '*Regular*' os outros três são variações desses valores. Para corrigir isso e realizar sua conversão:

```
1 df['Item_Fat_Content'] = df['Item_Fat_Content'].map({'LF': 1, 'Low Fat': 1, 'low
   fat': 1, 'reg': 2, 'Regular': 2})
2 df['Item_Fat_Content'] = df['Item_Fat_Content'].astype(pd.Int64Dtype())
3 df['Outlet_Location_Type'] = df['Outlet_Location_Type'].map({'Tier 1': 1, 'Tier 2':
   2, 'Tier 3': 3})
4 df['Outlet_Location_Type'] = df['Outlet_Location_Type'].astype(pd.Int64Dtype())
5 df['Outlet_Type'] = df['Outlet_Type'].map({'Supermarket Type1': 1, 'Supermarket
   Type2': 2, 'Supermarket Type3': 3, 'Grocery Store': 4})
6 df['Outlet_Type'] = df['Outlet_Type'].astype(pd.Int64Dtype())
```

Criamos um dicionário com as faixas, repetimos os mesmos valores para os tipos que são semelhantes e realizamos a troca dos elementos no *DataFrame*. Aplicamos também a mesma prática para a localização e tipo da loja que possui poucos valores. Vamos agora com o caso de tipo do item que devemos trocá-lo para uma forma diferente (é ideal quando existem muitos valores diferentes).

Cada atributo tem um tipo determinado, por exemplo, *float* aceita números com pontos decimais, *int* numéricos inteiros, *string* caracteres, além disso *Python* trabalha com um tipo especial denominado *category*. Corresponde a uma determinada faixa de valores. Converter tipo do item em atributo categórico:

```
1 df['Item_Type'] = df.Item_Type.astype('category')
```

Uma vez realizado esse processo podemos "codificá-lo":

```
1 le_Item_Type = LabelEncoder()
2 df['Item_Type'] = le_Item_Type.fit_transform(df['Item_Type'])
3 df.head()
```

Para cada valor categorizado é atribuído um valor numérico (ou seja o mesmo trabalho que tivemos para o mapa). Usamos as funções `info()` e `describe()` e podemos partir para a próxima etapa sem quaisquer problemas com os dados, pois agora são todos numéricos e não possuem qualquer valor nulo.

1.7.4 Separação e treino

Separar em treino e teste (para avaliarmos nosso modelo) e remover o atributo alvo:

```
1 target = df['Item_Outlet_Sales']
2 df = df.drop(columns=['Item_Outlet_Sales'], axis=1)
3 X_train, X_test, y_train, y_test = train_test_split(df, target, test_size = .2)
4 print('Amostra de Treino:', X_train.shape)
5 print('Amostra de Teste:', X_test.shape)
```

Usamos um valor de 20% para nosso teste e obtemos: 6.818 observações para treino e 1.705 de teste. Treinamos nosso modelo e verificamos seu resultado:

```
1 clf = LinearRegression()
2 clf.fit(X_train, y_train)
3 print('Acurácia: ', clf.score(X_test, y_test))
```

E obtemos uma acurácia aproximada de 42% e qual o motivo dessa discrepância tão grande? Simples, estamos cada vez mais perto da realidade e podemos verificar que realizar previsões com altos scores e poucas ações não existe. Pois se fosse assim: Corramos para treinar um modelo que nos dará os seis números da MegaSena. Ou ao menos nos dizer quando vai chover corretamente com muito pouco trabalho. Por fim podemos ver como os dados estão bem discrepantes em relação ao que foi predito e o real:

```
1 y_pred = model.predict(X_test)
2 plt.plot(y_test, y_test)
3 plt.scatter(y_test, y_pred, c = 'red', marker='+')
4 plt.ylabel('Real')
5 plt.xlabel('Predito')
6 plt.show()
```

Obtemos como resultado:

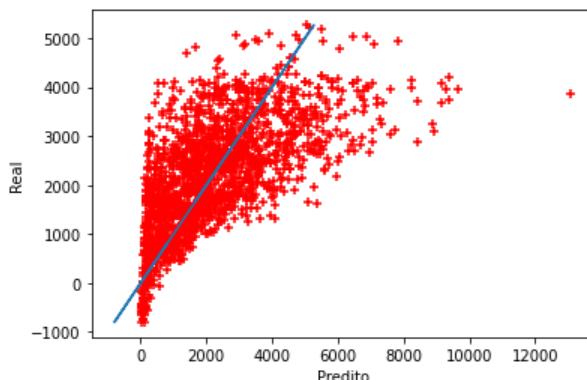


Figura 1.22: Regressão Linear aplicada a vários atributos

Em vermelho são a relação entre o valor real e o que foi predito, a linha azul mostra a Reta da Regressão. Verificamos a presença de um ponto bem isolado? Pode ser um *outlier*? Exatamente por esse motivo que passamos um bom tempo em EDA.

1.8 Árvore de Decisão

Se existe uma unanimidade entre os modelos que todo o Cientista de Dados conhece *Decision Tree* seria provavelmente o grande campeão (ou estaria entre os primeiros colocados), é um modelo com base em perguntas e respostas. Vamos começar com a importação das nossas bibliotecas básicas:

```
1 import pandas as pd
2 import numpy as np
3 from sklearn.preprocessing import LabelEncoder
4 from sklearn import tree
```

Além da Pandas e NumPy, que serão básicas para todos os modelos que vemos, necessitamos também da *LabelEncoder* que responde as manipulações no DataFrame e do objeto *tree* por conter a classe para executar o algoritmo. Próximo passo é buscarmos os nossos dados:

```
1 df = pd.read_csv('bases/salaries.csv')
2 df.head()
```

Essa base de salários é composta por 3 empresas: Google, Facebook e ABC Pharma. Temos ainda Cargo, Grau de Ensino e Salário (Anual). E chegamos na parte essencial por adotar esse modelo: **O que desejamos responder?**, exatamente isso o modelo responde a uma pergunta que deve ser formulada de forma binária, ou seja que tal na empresa X, com um cargo Y e curso Z conseguimos ganhar um salário maior que 100 mil anuais?

Precisamos preparar nossa base de dados e adicionar uma resposta a nossa pergunta, assim criamos a variável **desejo**:

```
1 desejo = pd.Series(np.where(df['salary']>=100000, 1, 0))
2 print(desejo)
```

Observamos que desejo possui somente os valores 0 e 1 que indica falso ou verdadeiro para cada um dos itens da nossa tabela original. Por exemplo: trabalhar na Google, com um curso de Bacharel no cargo Executivo de Vendas não fará ganhar um salário de 100 mil anuais, mas no cargo Gerente de Negócios sim.

Para criarmos nossa Árvore de Decisão, precisamos que todos os campos (que participarão de sua composição) sejam variáveis numéricas categóricas, sua formação se dará da seguinte forma:

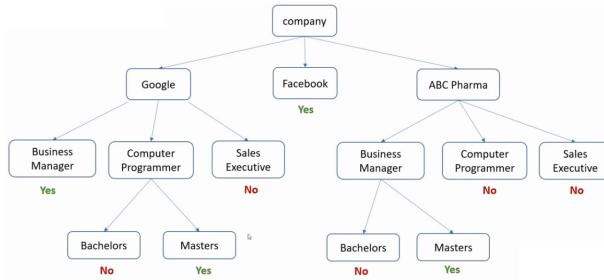


Figura 1.23: Estrutura da Árvore Montada

A função da biblioteca LabelEncoder é fazer exatamente isso, ou seja, transformar uma variável caractere, criar uma categoria e atribuir uma valor para ela, então:

```

1 le_company = LabelEncoder()
2 le_job = LabelEncoder()
3 le_degree = LabelEncoder()
4
5 df['company_n'] = le_company.fit_transform(df['company'])
6 df['job_n'] = le_job.fit_transform(df['job'])
7 df['degree_n'] = le_degree.fit_transform(df['degree'])
8 df.head()
  
```

Criamos 3 novas séries e adicionamos ao nosso DataFrame original, vemos que foi atribuído um valor numérico para cada categoria, por exemplo Google é 2, Business Manager (Gerente de Negócios) é 0. Dessa forma que o algorítimo monta nossa árvore, a partir dessas três variáveis. Porém precisamos de um *DataFrame* composto somente por elas:

```

1 entradas = df.drop(['company', 'job', 'degree', 'salary'], axis='columns')
2 entradas.head()
  
```

De um modo mais simples, criamos um novo *DataFrame* (entradas) sem as variáveis descritivas (Empresa, Cargo e Grau de Ensino) e a numérica (Salário). Tudo está pronto, com esses dois novos objetos (**desejo** e **entradas**) podemos treinar nosso algorítimo:

```

1 model = tree.DecisionTreeClassifier()
2 model.fit(entradas, desejo)
  
```

E podemos realizar nossas perguntas, por exemplo, alguém da **Google** (2) que trabalha no cargo **Executivo de Vendas** (2) e fez **Mestrado** (1) recebe um salário maior que 100 mil anuais?

```

1 model.predict([[2, 2, 1]])
  
```

E obtemos como resposta o valor **0** indicando que não. Este é um dos principais modelos utilizados em Ciência de Dados tornou-se a porta de entrada para muitos outros. Podemos nos enganar com o pensamento que bastaria dar uma olhada nos dados para respondermos a pergunta, em uma base pequena realmente isso pode até ser viável, porém em uma base com muitos dados seria impraticável.

1.8.1 Critérios Gini e Entropia

Uma árvore de decisão se enquadra nos Algoritmos para Aprendizado de Máquina supervisionados e pode ser usada tanto para classificação quanto regressão - embora principalmente para o primeiro tipo. Como vimos este modelo obtém uma instância, atravessa uma árvore e compara recursos importantes com uma determinada declaração condicional. Se ele desce para o ramo filho esquerdo ou direito depende do resultado. Normalmente, os recursos mais importantes estão próximos a raiz.

Existem dois critérios que são comumente passados, são consideradas métricas padrão para calcular "impureza" ou "nível de informação". Orientam a divisão de um nó na árvore de decisão apenas com base nas informações existentes nesse nó. São calculados conforme as seguintes fórmulas:

- Gini: $1 - \sum_{j=1}^c (p_j)^2$
- Entropy: $-\sum_{j=1}^c p_j \times \log(p_j)$

Gini é a probabilidade de uma amostra aleatória ser classificada incorretamente se escolhermos aleatoriamente um *label* de acordo com a distribuição em um ramo. **Entropia** é uma medida de informação (ou melhor, a falta dela). Ao calcular o ganho de informação sobre uma divisão. Qual é a diferença em entropias. Isso mede como se reduz a incerteza sobre um *label*.

Vamos começar um outro projeto, porém veremos algo bem diferente, não iremos nos preocupar em explanar os dados que exploraremos. Vamos iniciar pelas bibliotecas:

```

1 import pandas as pd
2 from sklearn.model_selection import train_test_split
3 from sklearn.tree import DecisionTreeClassifier
4 from sklearn.metrics import accuracy_score
5 from sklearn.metrics import classification_report
6 from sklearn.metrics import confusion_matrix

```

A grande diferença está aqui: criamos um conjunto de funções para realizar todo o trabalho de leitura, separação e execução do modelo. Recomendo esse método quando for criar seus projetos, pois simplifica muito a manutenção dos mesmos:

```

1 def obterDados(base):
2     balance_data = pd.read_csv(base, sep = ',', header = None)
3     print('Tamanho:', balance_data.shape)
4     print(balance_data.head())
5     return balance_data
6
7 def separarDados(dados):
8     x = dados.values[:,1:5]
9     y = dados.values[:,0]
10    x_train, x_test, y_train, y_test = train_test_split(x, y, test_size = 0.3,
11                                                       random_state = 100)
12    return x, y, x_train, x_test, y_train, y_test
13
14 def treinar(x_train, y_train, criterio):
15     clf = DecisionTreeClassifier(criterion = criterio, random_state = 100, max_depth =
16         3, min_samples_leaf = 5)
17     clf.fit(x_train,y_train)

```

```

16     return clf
17
18 def predizer(x_test, clf):
19     y_pred = clf.predict(x_test)
20     print(f"Valores Preditos: {y_pred}")
21     return y_pred
22
23 def acuracia(y_test, y_pred):
24     print("Acurácia:", accuracy_score(y_test, y_pred, average='weighted',
25         zero_division=1) * 100)
26     print(confusion_matrix(y_test, y_pred))
27     print(classification_report(y_test, y_pred))

```

O *obterDados()* é responsável por ler e devolver as informações da nossa base de dados, *separarDados()* por dividir corretamente nossa base em treino e teste, *treinar()* realiza o treinamento da base conforme determinado critério, *predizer()* obtém as previsões com base na massa de teste em conjunto com o classificador e *acuracia()* verifica os resultados obtidos.

Com algumas mudanças podemos adaptar esses métodos e começar a pensar em criar uma biblioteca de modo a facilitar nossos projetos futuros. Veja como é mais simples com seu uso.

Para ler os dados:

```
1 data = obterDados('bases/balance-scale.data')
```

Para separar os dados:

```
1 x, y, x_train, x_test, y_train, y_test = separarDados(data)
```

Treinar com o critério **Gini**:

```
1 clf = treinar(x_train, y_train, 'gini')
2 y_pred_gini=prediction(x_test, clf)
```

Obter o resultado:

```
1 cal_accuracy(y_test, y_pred_gini)
```

E obtemos como resultado:

	Acurácia: [100. 72.82608696 73.95833333]
	[[0 6 7] [0 67 18] [0 19 71]]
	precision recall f1-score support
B	0.00 0.00 0.00 13
L	0.73 0.79 0.76 85
R	0.74 0.79 0.76 90
accuracy	0.73 188
macro avg	0.49 0.53 0.51 188
weighted avg	0.68 0.73 0.71 188

Figura 1.24: Resultado para o Critério Gini

Treinar com o critério **Entropia**:

```
1 clf = treinar(x_train, y_train, 'entropy')
2 y_pred_entr = prediction(x_test, clf)
```

Obter o resultado:

```
1 acuracia(y_test, y_pred_entr)
```

E obtemos como resultado:

Acurácia: [100. 70.78651685 70.70707071]	
	[[0 6 7] [0 63 22] [0 20 70]]
precision	
B	0.00
L	0.71
R	0.71
recall	
B	0.00
L	0.74
R	0.78
f1-score	
B	0.00
L	0.72
R	0.74
support	
B	13
L	85
R	90
accuracy	0.71
macro avg	0.47
weighted avg	0.66
	188
	0.49
	188
	0.68
	188

Figura 1.25: Resultado para o Critério Entropia

E observamos que neste caso, Gini possui uma melhor resposta aos dados.

1.9 Floresta Aleatória

Random Forest (por caridade *Florest* não existe na língua inglesa) é um modelo supervisionado que pode ser considerado como um aprimoramento da Árvore de Decisão, vamos recapitular seu funcionamento.

Através de um conjunto definido de regras, um novo valor passará por uma série de perguntas, os chamados ramos da árvore, para prevermos em qual melhor resposta se encaixa. Agora vamos pensar que podemos separar os dados em várias partes definidas aleatoriamente e usarmos uma árvore para cada uma dessas partes e para tomarmos uma decisão apenas computamos qual foi a resposta mais votada?

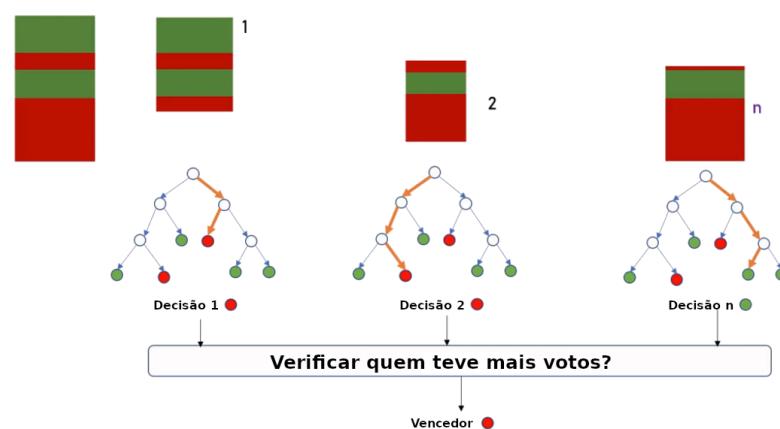


Figura 1.26: Funcionamento da Floresta Aleatória

E esse é o conceito da **Floresta Aleatória**, para aprendermos como é esse modelo na prática. Começamos com a importação das bibliotecas:

```

1 import pandas as pd
2 from sklearn.model_selection import train_test_split
3 from sklearn.datasets import load_digits
4 from sklearn.ensemble import RandomForestClassifier
5 from sklearn.metrics import confusion_matrix
6 import matplotlib.pyplot as plt
7 import seaborn as sn
8
9 plt.figure(figsize=(10,7))
10 %matplotlib inline

```

A única aquisição nova é o método *RandomForestClassifier* que vem da ensemble contida na biblioteca *Scikit-learn*. Como fonte de dados usamos os dados de *load_digits* (isso mesmo, finalmente vamos trabalhar com imagens). Lemos nossos dados:

```
1 digits = load_digits()
```

E verificamos um determinado dígito, por exemplo o 8:

```

1 plt.matshow(digits.images[8])
2 plt.show()

```

E obtemos como resultado:

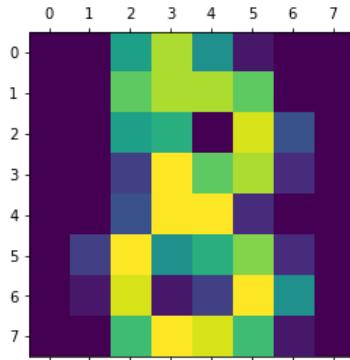


Figura 1.27: Número 8 da load_digits

Uma imagem é formada por vários pontos que denominamos de pixels, cada um desses possuem valor em uma escala de cores, no caso dessa base obtemos 64 colunas identificando cada um desses pontos. Vamos criar um DataFrame com esses dados:

```

1 df = pd.DataFrame(digits.data)
2 df.head()

```

E obtemos como resultado:

	0	1	2	3	4	5	6	7	8	9	...	54	55	56	57	58	59	60	61	62	63
0	0.0	0.0	5.0	13.0	9.0	1.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	6.0	13.0	10.0	0.0	0.0	0.0
1	0.0	0.0	0.0	12.0	13.0	5.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	11.0	16.0	10.0	0.0	0.0
2	0.0	0.0	0.0	4.0	15.0	12.0	0.0	0.0	0.0	0.0	...	5.0	0.0	0.0	0.0	0.0	3.0	11.0	16.0	9.0	0.0
3	0.0	0.0	7.0	15.0	13.0	1.0	0.0	0.0	0.0	8.0	...	9.0	0.0	0.0	0.0	7.0	13.0	13.0	9.0	0.0	0.0
4	0.0	0.0	0.0	1.0	11.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	2.0	16.0	4.0	0.0	0.0

Figura 1.28: Dados das Imagens

O problema é que não existe como coluna o resultado e precisamos colocá-lo, esse valor está no elemento target (não pense que existem apenas 9 imagens, são 1.997):

```
1 df['valor'] = digits.target
```

Separamos nossos dados na massa de treino e teste (20% do total):

```
1 X_train, X_test, y_train, y_test = train_test_split(df.drop(['valor']),
2 axis='columns'), df['valor'], test_size = .2)
2 print(len(X_train), len(X_test))
```

E treinamos nosso modelo:

```
1 clf = RandomForestClassifier()
2 clf.fit(X_train, y_train)
```

Observamos no resultado o parâmetro "n_estimators=100", isso significa que são 100 árvores na nossa floresta, podemos aumentar ou diminuir (sem pensar em desmatamento) o número dessas árvores e observar se conseguimos aumentar nosso resultado. Verificamos a acurácia:

```
1 clf.score(X_test, y_test)
```

E obtemos significativos 98%, ou seja, nosso modelo erra apenas 2% das vezes, o que pode ser visto na Matriz de Confusão :

```
1 y_predicted = clf.predict(X_test)
2 cm = confusion_matrix(y_test, y_predicted)
3 sn.heatmap(cm, annot=True, cmap=plt.cm.Blues)
4 plt.xlabel('Predito')
5 plt.ylabel('Verdadeiro')
6 plt.show()
```

Obtemos como resultado:

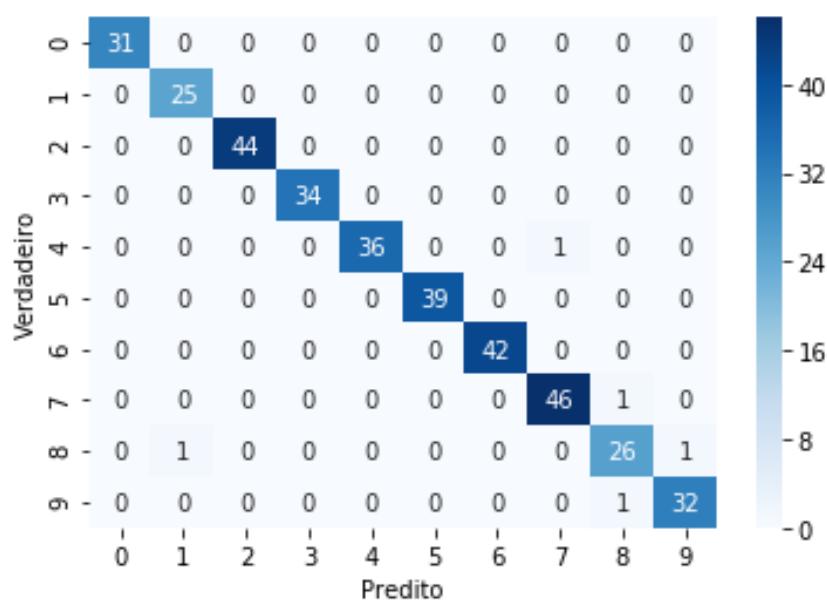
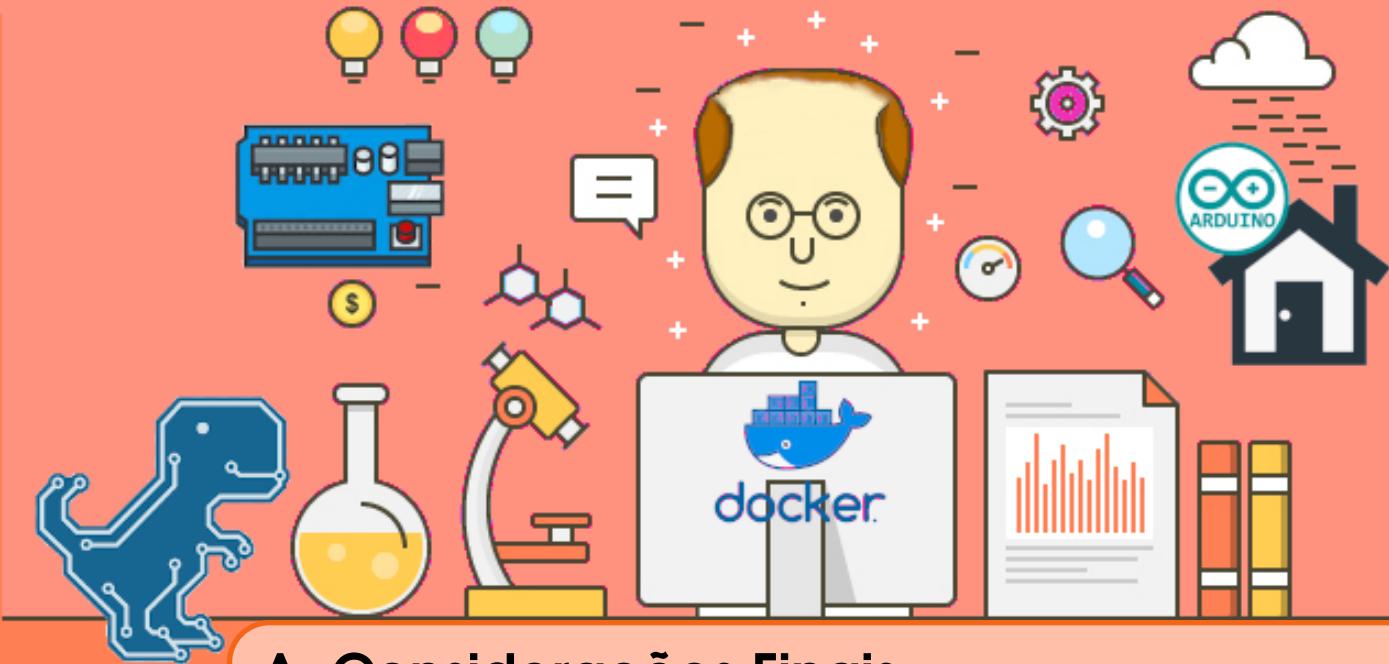


Figura 1.29: Matriz de Confusão no Seaborn



A. Considerações Finais

F Você não pode ensinar nada a ninguém, mas pode ajudar a pessoas a descobrirem por si mesmas.
(Galileu Galilei - Físico)

Os artigos deste livro foram selecionados das diversas publicações que fiz no Linkedin e encontradas em outros sites que foram nesta obra explicitamente citadas. Acredito que apenas com a prática podemos almejar o cargo de Cientista de Dados, então segue uma relação de boas bases que podemos encontrar na Internet:

- 20BN-SS-V2: <https://20bn.com/datasets/something-something>
- Actualitix: <https://pt.actualitix.com/>
- Banco Central do Brasil: <https://www3.bcb.gov.br>
- Banco Mundial: <http://data.worldbank.org>
- Censo dos EUA (População americana e mundial): <http://www.census.gov>
- Cidades Americanas: <http://datasf.org>
- Cidade de Chicago: <https://data.cityofchicago.org/>
- CIFAR-10: <https://www.cs.toronto.edu/~kriz/cifar.html>
- Cityscapes: <https://www.cityscapes-dataset.com/>
- Criptomoedas: <https://pro.coinmarketcap.com/migrate/>
- Dados da União Europeia: <http://open-data.europa.eu/en/data>
- Data 360: <http://www.data360.org>
- Datahub: <http://datahub.io/dataset>
- DBpedia: <http://wiki.dbpedia.org/>
- Diversas áreas de negócio e finanças: <https://www.quandl.com>
- Diversos assuntos: <http://www.firebaseio.com>
- Diversos países (incluindo o Brasil): <http://knoema.com>
- Fashion-MNIST: <https://www.kaggle.com/zalando-research/fashionmnist>
- Gapminder: <http://www.gapminder.org/data>

- Google Finance: <https://www.google.com/finance>
- Google Trends: <https://www.google.com/trends>
- Governo do Brasil: <http://dados.gov.br>
- Governo do Canadá (em inglês e francês): <http://open.canada.ca>
- Governo dos EUA: <http://data.gov>
- Governo do Reino Unido: <https://data.gov.uk>
- ImageNET: <http://www.image-net.org/>
- IPEA: <http://www.ipeadata.gov.br>
- IMDB-Wiki: <https://data.vision.ee.ethz.ch/cvl/rrothe/imdb-wiki/>
- Kinetics-700: <https://deepmind.com/research/open-source/kinetics>
- Machine Learning Databases: <https://archive.ics.uci.edu/ml/machine-learning-databases/>
- MEC Microdados INEP: <http://inep.gov.br/microdados>
- MS coco: <http://cocodataset.org/#home>
- MPII Human Pose: <http://human-pose.mpi-inf.mpg.de/>
- Músicas: <https://aws.amazon.com/datasets/million-song-dataset>
- NASA: <https://data.nasa.gov>
- Open Data Monitor: <http://opendatamonitor.eu>
- Open Data Network: <http://www.opendatanetwork.com>
- Open Images: <https://github.com/openimages/dataset>
- Portal de Estatística: <http://www.statista.com>
- Públicos da Amazon: <http://aws.amazon.com/datasets>
- R-Devel: <https://stat.ethz.ch/R-manual/R-devel/library/datasets/html/00Index.html>
- Reconhecimento de Faces: <http://www.face-rec.org/databases>
- Saúde: <http://www.healthdata.gov>
- Statsci: <http://www.statsci.org/datasets.html>
- Stats4stem: <http://www.stats4stem.org/data-sets.html>
- Stanford Large Network Dataset Collection: <http://snap.stanford.edu/data>
- Vincent Rdatasets: <https://vincentarelbundock.github.io/Rdatasets/datasets.html>
- Vitivinicultura Embrapa: <http://vitibrasil.cnpuv.embrapa.br/>

Esse não é o fim de uma jornada acredito ser apenas seu começo. Espero que este livro possa lhe servir para criar algo maravilhoso e fantástico que de onde estiver estarei torcendo por você.

A.1 Sobre o Autor

Fortes conhecimentos em linguagens de programação Java e Python. Especialista formado em Gestão da Tecnologia da Informação com forte experiência em Bancos Relacionais e não Relacionais. Possui habilidades analíticas necessárias para encontrar a providencial agulha no palheiro dos dados recolhidos pela empresa. Responsável pelo desenvolvimento de dashboards com a capacidade para análise de dados e detectar tendências, autor de 15 livros e diversos artigos em revistas especializadas, palestrante em seminários sobre tecnologia. Focado em aprender e trazer mudanças para a organização com conhecimento profundo do negócio.

- Perfil no Linkedin: <http://www.linkedin.com/pub/fernando-anselmo/23/236/bb4>
- Endereço do Git: <https://github.com/fernandoans/machinelearning>

Machine Learning na Prática

ESTE LIVRO PODE E DEVE SER DISTRIBUÍDO LIVREMENTE

Fernando Anselmo