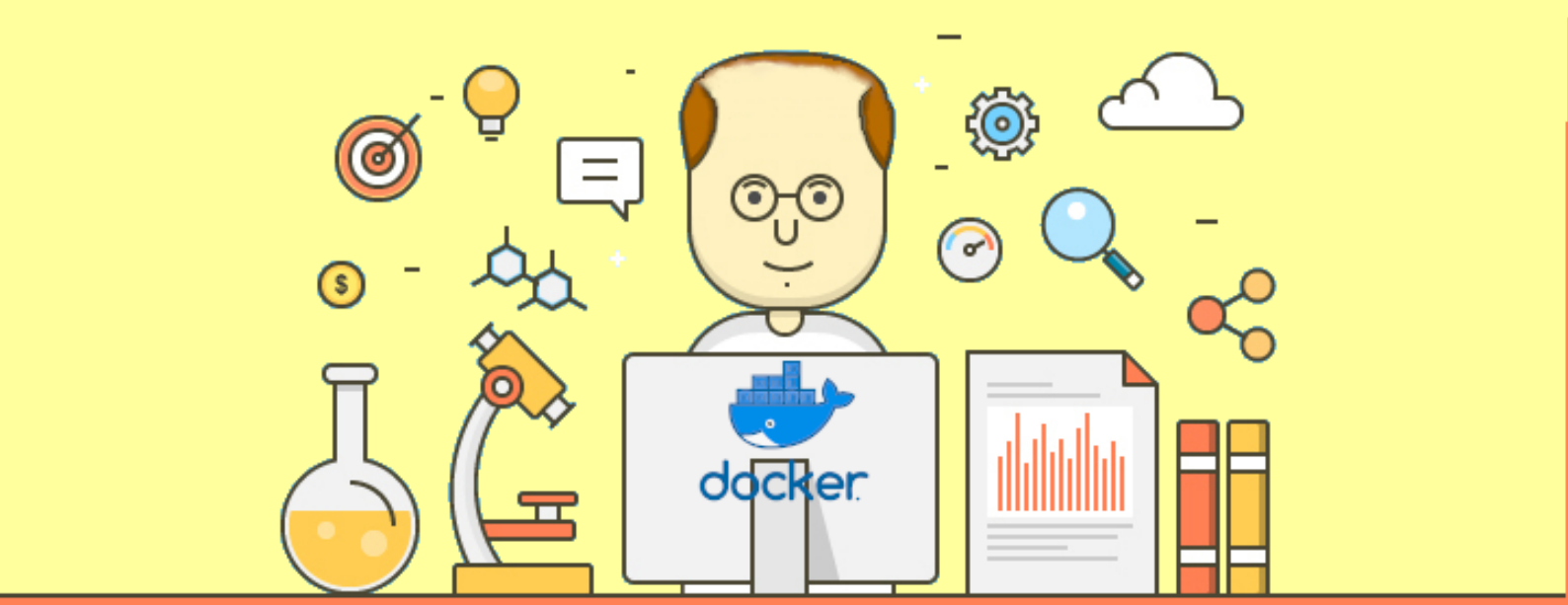
Machine Learning na Prática

Modelos em Python

Fernando Anselmo

PUBLICAÇÃO INDEPENDENTE

http:\fernandoanselmo.orgfree.com



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

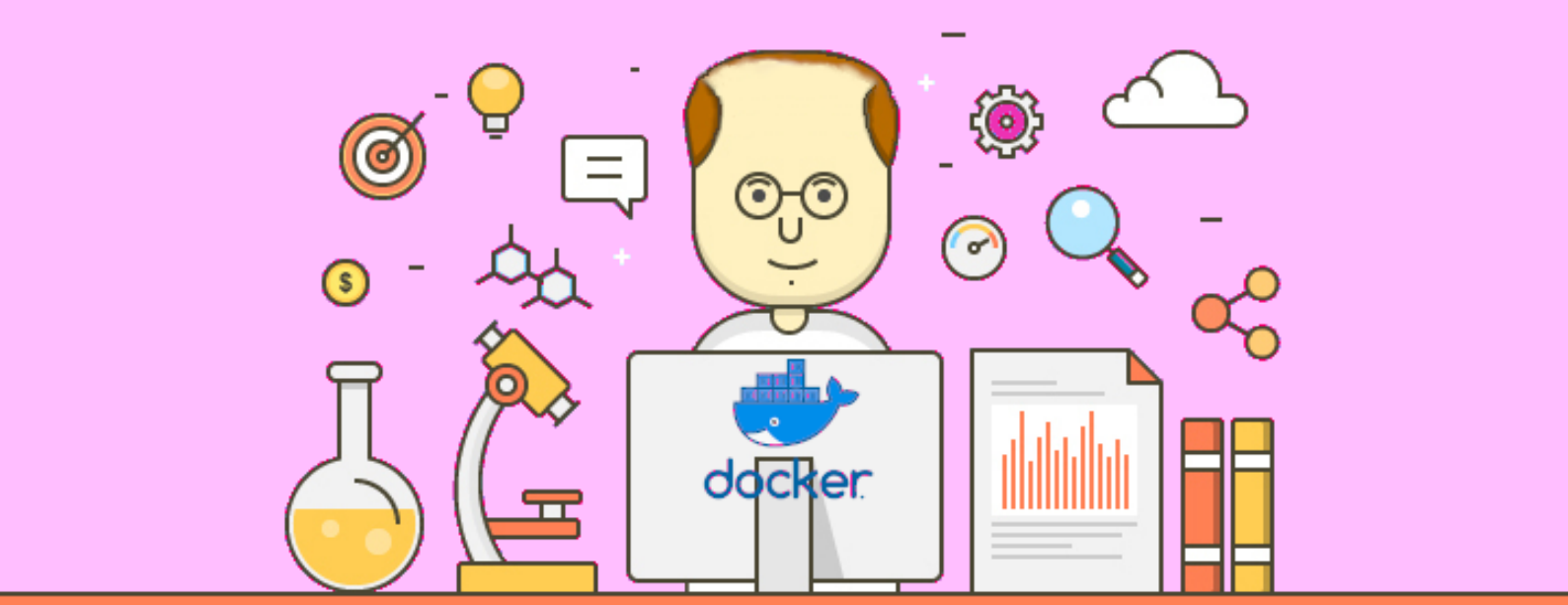
1.7.2 Regressão Linear com mais de um Preditor 21

1.7.3 Regressão Linear e Limpeza dos Dados 22

1.7.4 Separação e treino 24

A Considerações Finais 27

A.1 Sobre o Autor 28



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).

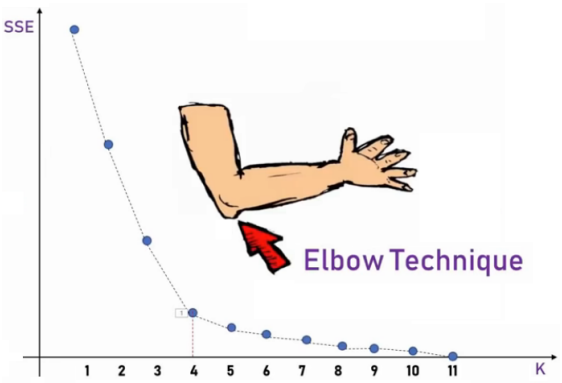


Figura 1.1: Técnica do Cotovelo

Exatamente na posição 4 existe uma "quebra"para passar ao próximo valor, usamos para definir essa quebra o SSE[1] (*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:

```
import pandas as pd
import numpy as np
from sklearn.preprocessing import scale
from sklearn.cluster import KMeans
from matplotlib import pyplot as plt

%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.

```
df = pd.read_csv('bases/gameML.csv', delimiter=';')
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:

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

Obtemos como resultado:

[1] Soma 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.

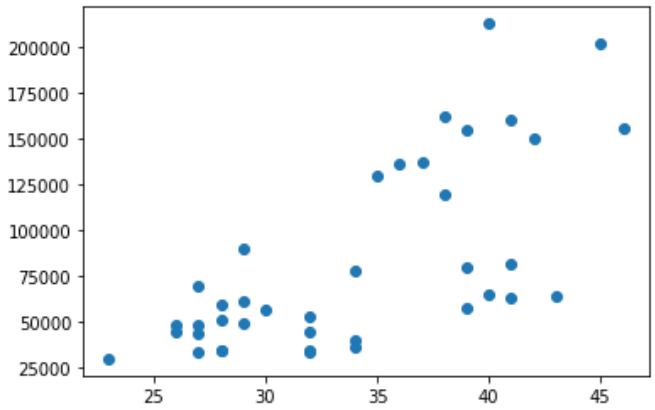


Figura 1.3: Idades e Salários da Empresa GameML

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

```
k_rng = range(1,10)
sse = []
for k in k_rng:
  km = KMeans(n_clusters=k)
  km.fit(df[['Idade','Salário']])
  sse.append(km.inertia_)
plt.xlabel('K')
plt.ylabel('SSE (Sum Squared Error)')
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 e 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:

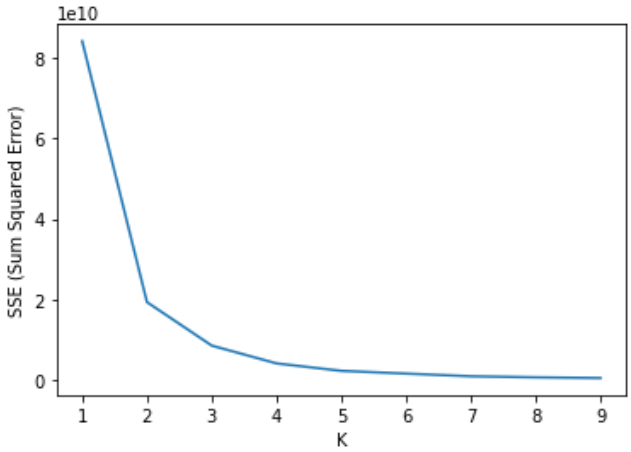


Figura 1.4: Gráfico com os valores de Inércia

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 algorítimo K-Means, devemos colocar os dados em "escala", vamos tentar usar o modelo sem proceder dessa forma:

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

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

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

Obtemos como resultado:

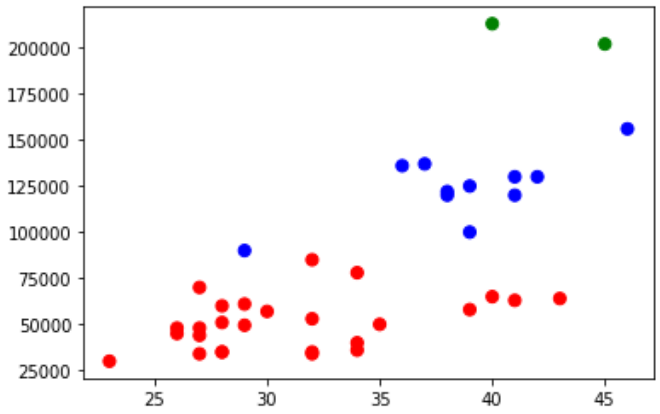


Figura 1.5: Separados por Grupo

E parece que temos 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 em escala, isso é feito assim:

```
df['Salário'] = scale(df.Salário)
df['Idade'] = scale(df.Idade)
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:

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

Plotamos novamente o gráfico e agora como resultado teremos:

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

E obtemos como resultado:

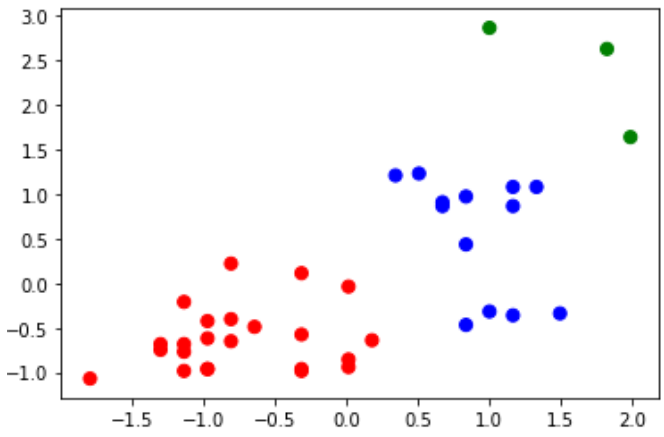


Figura 1.6: Separados por Grupo em Escala

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.

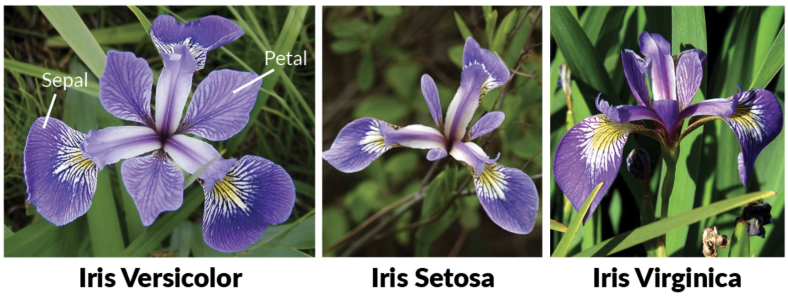


Figura 1.7: Flores Iris

Nessa base existem três espécies separadas: Versicolor, Setosa e Virgínica. 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:

```
import numpy as np
from matplotlib import pyplot as plt
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn import neighbors

%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:

```
iris = datasets.load_iris()
X, y = iris.data, iris.target

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 **Virgínica** que colocaremos em y). Usamos o método *train_test_split* para retirar 20% dos dados como amostra de teste e assim teremos quatro agrupamentos:

- **X_train**, com os dados para treino do algorítimo.
- **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:

```
clf = neighbors.KNeighborsClassifier()
clf.fit(X_train, y_train)
```

```
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:

```
cores = np.array(['green', 'red', 'blue'])
subplt1 = 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:

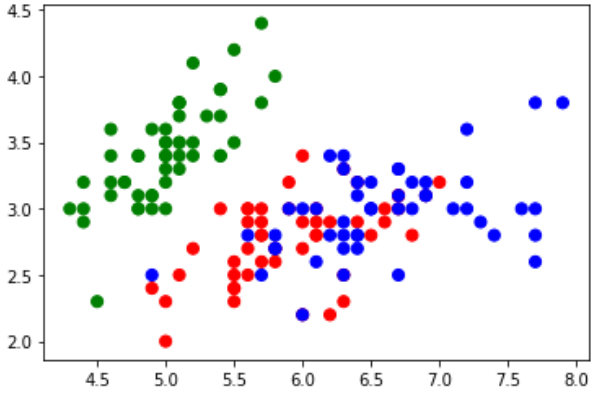


Figura 1.8: Comparar tamanho e largura da Sépala

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

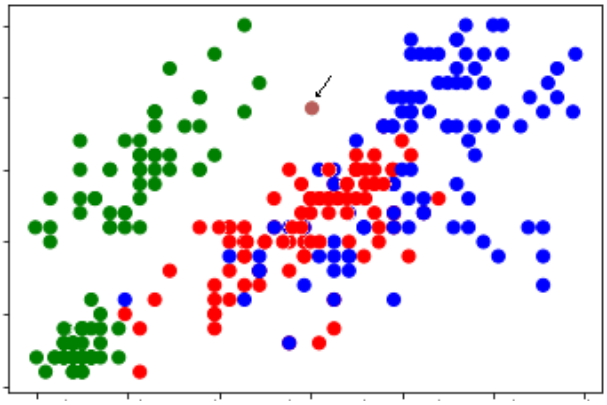


Figura 1.9: Localizar o Ponto Roxo

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:

```
cores = np.array(['green', 'red', 'blue'])
subplt1 = plt.scatter(x=X[:, 2], y=X[:, 3], c=cores[y], s=50)
```

E obtemos como resultado:

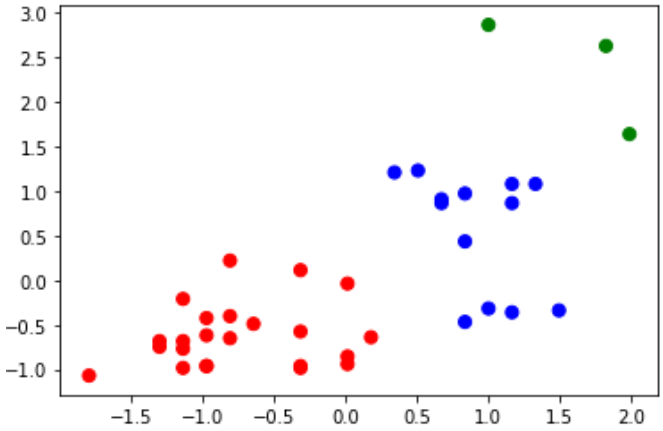


Figura 1.10: Comparar tamanho e largura da Pétala

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

```
predicao = clf.predict([[6.0, 4.0, 4.0, 2.0]])
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 pertubação nesse centroide. Vamos importar algumas bibliotecas para realizarmos mais testes:

```
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn import datasets
from sklearn.preprocessing import scale
from sklearn.cluster import KMeans
from sklearn.metrics import classification_report

%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:

```
iris = datasets.load_iris()
X = scale(iris.data)
y = pd.DataFrame(iris.target)
y.columns = ['Targets']
variable_names = iris.feature_names
iris_df = pd.DataFrame(iris.data)
iris_df.columns = variable_names
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:

```
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 predições precisa apenas dos dados para separá-los em conjuntos. Mas como será que foi seu comportamento? Descobrimos isso comparando dois gráficos:

```
cores = np.array(['green', 'red', 'blue'])
relabel = np.choose(clustering.labels_, [1, 0, 2]).astype(np.int64)
plt.figure(figsize = [15, 5])

plt.subplot(1, 4, 1)
plt.scatter(x=iris_df['petal length (cm)'],
y=iris_df['petal width (cm)'],
c=cores[iris.target], s=50)
plt.title('Real (Pétala)')

plt.subplot(1, 4, 2)
plt.scatter(x=iris_df['petal length (cm)'],
y=iris_df['petal width (cm)'],
c=cores[relabel], s=50)
plt.title('KMeans (Pétala)')

plt.subplot(1, 4, 3)
plt.scatter(x=iris_df['sepal length (cm)'],
y=iris_df['sepal width (cm)'],
c=cores[iris.target], s=50)
```

```
plt.title('Real (Sépala)')

plt.subplot(1, 4, 4)
plt.scatter(x=iris_df['sepal length (cm)'],
y=iris_df['sepal width (cm)'],
c=cores[relabel], s=50)
plt.title('KMeans (Sépala)')

plt.show()
```

Usamos os mesmos conjuntos de cores para cada espécie, teremos 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:

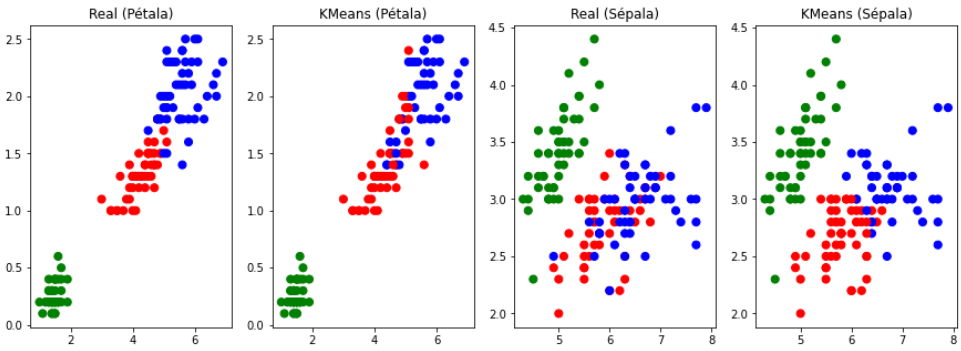


Figura 1.12: Comparativo entre o Real e o KMeans

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:

```
metricas = classification_report(y, relabel)
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 Dendrograma[2].

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:

```
from scipy.cluster.hierarchy import dendrogram, linkage
from sklearn.cluster import AgglomerativeClustering
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:

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

Essa base contém 32 modelos de carros com os seguintes atributos: Nome, quilometragem, número de cilindros, deslocamento (medida de poder do carro em polegada cúbica), cavalos de força, relação do eixo traseiro, peso (em libras), comparativo de eficiência de 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:

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

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

```
    show_contracted=True)

plt.title('Dendograma')
plt.xlabel('Tamanho do Cluster')
plt.ylabel('Distancia')
plt.axhline(y=500)
plt.axhline(y=150)
plt.show()
```

Obtemos como resultado:

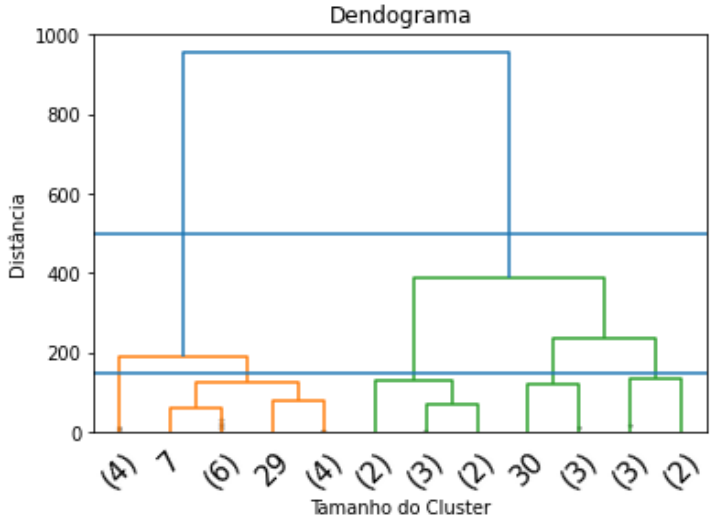


Figura 1.14: Dendrograma dos tamanhos do Cluster

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 (lembramos 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:

```
hclusters1 = AgglomerativeClustering(n_clusters=2, affinity='euclidean',
    linkage='ward').fit(X)
print('Método 1:', accuracy_score(y, hclusters1.labels_))

hclusters2 = AgglomerativeClustering(n_clusters=2, affinity='euclidean',
    linkage='complete').fit(X)
print('Método 2:', accuracy_score(y, hclusters2.labels_))

hclusters3 = AgglomerativeClustering(n_clusters=2, affinity='euclidean',
    linkage='average').fit(X)
print('Método 3:', accuracy_score(y, hclusters3.labels_))

hclusters4 = AgglomerativeClustering(n_clusters=2, affinity='manhattan',
    linkage='single').fit(X)
print('Método 4:', accuracy_score(y, hclusters4.labels_))

hclusters5 = AgglomerativeClustering(n_clusters=2, affinity='manhattan',
    linkage='complete').fit(X)
print('Método 5:', accuracy_score(y, hclusters5.labels_))

hclusters6 = AgglomerativeClustering(n_clusters=2, affinity='manhattan',
    linkage='average').fit(X)
print('Método 6:', accuracy_score(y, hclusters6.labels_))

hclusters7 = AgglomerativeClustering(n_clusters=2, affinity='cosine',
    linkage='single').fit(X)
print('Método 7:', accuracy_score(y, hclusters7.labels_))

hclusters8 = AgglomerativeClustering(n_clusters=2, affinity='cosine',
    linkage='complete').fit(X)
print('Método 8:', accuracy_score(y, hclusters8.labels_))

hclusters9 = AgglomerativeClustering(n_clusters=2, affinity='cosine',
    linkage='average').fit(X)
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):

```
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:

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn import preprocessing
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report
```

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

```
X = preprocessing.scale(X)
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.

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

Treinamos nosso modelo e podemos avaliar o resultado:

```
y_predito = clf.predict(X_teste)
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	72179226
1	1961	74311343
2	1962	76514328
3	1963	78772657
4	1964	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: temos atributos numéricos, "Ano"e "População". Para entendermos o relacionamento entre os atributos, plotamos esses em um gráfico:

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

Obtemos como resultado:

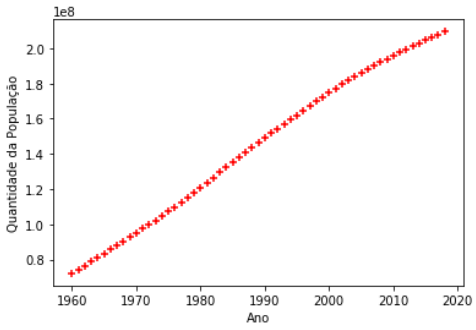


Figura 1.18: Dados da População Brasileira

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 temos nossos atributos conferidos, basta treinarmos nosso modelo e obtermos nossa previsão:

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

E teremos 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:

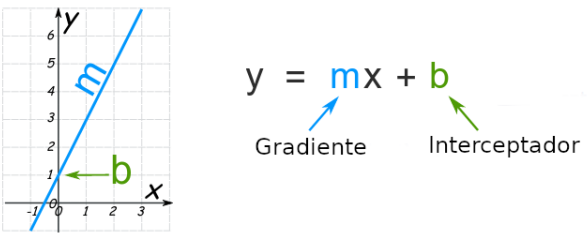


Figura 1.19: Base da Regressão Linear

> **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:

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

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

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

Obtemos como resultado:

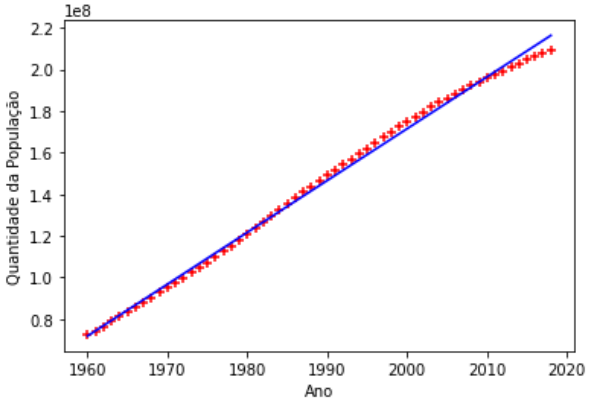


Figura 1.20: Dados da População Brasileira com a Previsão

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:

```
import pandas as pd
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split
from matplotlib import pyplot as plt

%matplotlib inline
```

E ler nossa base de dados:

```
df = pd.read_csv('bases/marketSales.csv')
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 código identificador 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).

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

Cuidado pois se aplicamos um corte seco como: `df.dropna(how='any',inplace=True)` teremos 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.

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

Não temos mais a presença de nulos, mas ainda temos 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:

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

```
df['Item_Fat_Content'] = df['Item_Fat_Content'].map({'LF': 1, 'Low Fat': 1, 'low
    fat': 1, 'reg': 2, 'Regular': 2})
df['Item_Fat_Content'] = df['Item_Fat_Content'].astype(pd.Int64Dtype())
df['Outlet_Location_Type'] = df['Outlet_Location_Type'].map({'Tier 1': 1, 'Tier 2':
    2, 'Tier 3': 3})
df['Outlet_Location_Type'] = df['Outlet_Location_Type'].astype(pd.Int64Dtype())
df['Outlet_Type'] = df['Outlet_Type'].map({'Supermarket Type1': 1, 'Supermarket
    Type2': 2, 'Supermarket Type3': 3, 'Grocery Store': 4})
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. Porém ainda temos o caso de tipo do item que vamos trocá-lo de 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:

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

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

```
le_Item_Type = LabelEncoder()
df['Item_Type'] = le_Item_Type.fit_transform(df['Item_Type'])
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:

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

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

```
clf = LinearRegression()
clf.fit(X_train, y_train)
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:

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

Obtemos como resultado:

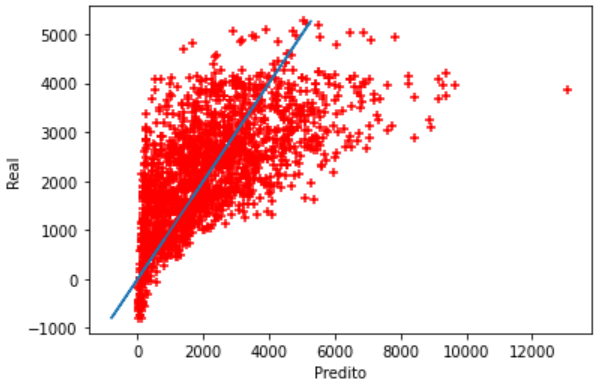
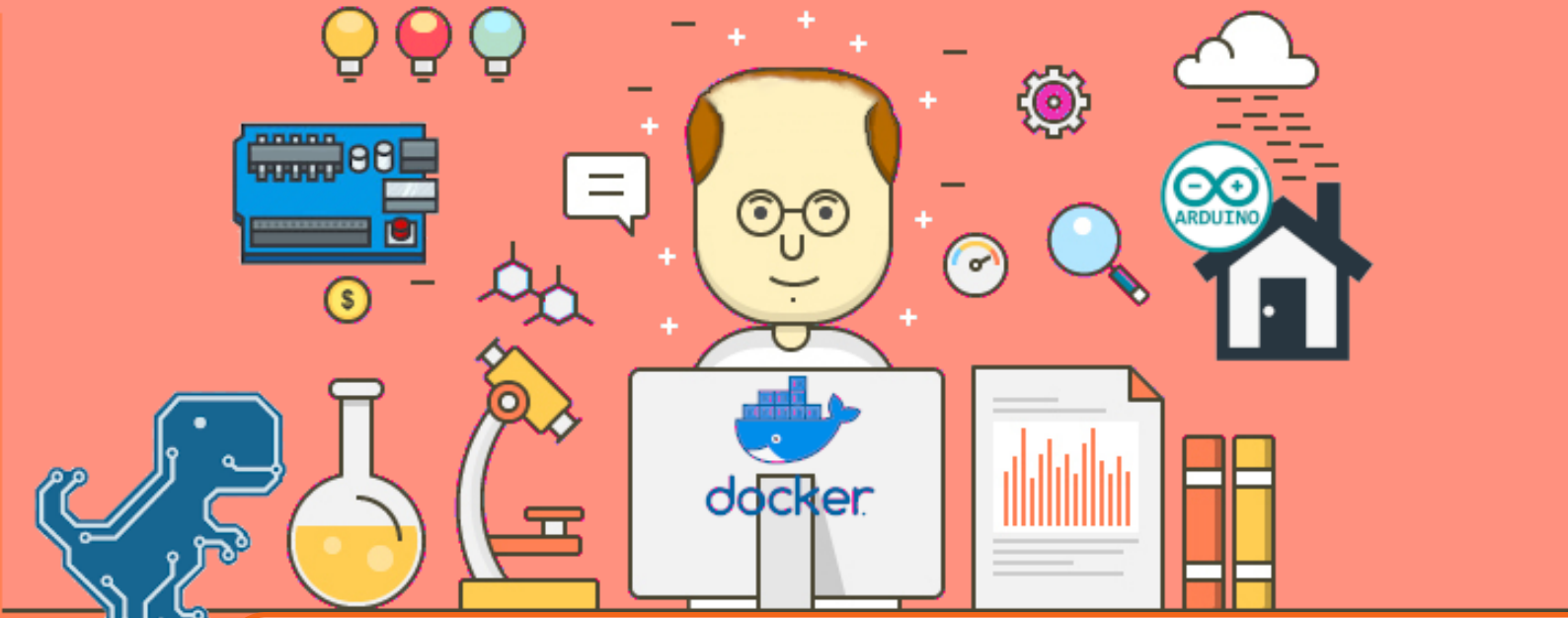


Figura 1.22: Regressão Linear aplicada a vários atributos

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



A. Considerações Finais

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.freebase.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 analise 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