

O *package* Scikit-learn

- Com os 4 *packages* estudados e com o conhecimento que temos da linguagem Python, já seria possível desenvolvermos os nossos próprios modelos de ML
 - *porém, isso iria requerer um esforço muito elevado e um conhecimento profundo de toda a complexa fundamentação matemática que suporta os atuais algoritmos de ML*
- Felizmente, para que não tenhamos que ser nós a implementá-los à mão, o *package* Scikit-learn disponibiliza-nos grande parte desses sofisticados – mas também complexos – algoritmos
- Usando o *package* Scikit-learn, em vez de gastarmos parte importante do nosso tempo a tentar replicar algoritmos já disponíveis e devidamente consolidados, vamos poder concentrar o nosso esforço nas importantes tarefas de **preparação** dos *datasets* (tarefas de pré-processamento), de **afinação** dos algoritmos (ajuste dos seus principais parâmetros de configuração) e de **avaliação** e **interpretação** dos resultados alcançados
- Segue-se apenas uma breve introdução ao *package* Scikit-learn, uma vez que será com ele que iremos explorar todos os algoritmos de *Machine Learning* na importante caminhada que temos ainda pela frente
- Para além dos principais algoritmos e de outras ferramentas importantes para a ML, o Scikit-learn faculta-nos ainda alguns *datasets* que poderemos usar como matéria prima na experimentação dos modelos de ML que iremos aprender a desenvolver
 - *mas será em repositórios online que poderemos encontrar um grande manancial de datasets para os nossos estudos*

Datasets para experimentação

- No processo de desenvolvimento de um modelo de ML, utiliza-se, para entrada do próprio modelo, um determinado *dataset*
 - *se esse modelo estiver a ser desenvolvido apenas num contexto de aprendizagem das técnicas de ML, não havendo um conjunto de dados reais a serem analisados, torna-se imperativo encontrar datasets com um volume suficiente de dados que tornem a experimentação o mais realista possível*
 - alguns desses datasets, bastante populares entre a comunidade da Data Science, encontramos-los nas próprias bibliotecas que estamos a usar, como é o caso da Seaborn e em particular da Scikit-learn
 - mas, como se disse antes, é no ciberespaço que temos ao nosso dispor enormes repositórios de *datasets* para experimentação
- Para uma primeira incursão na Scikit-learn e na ML, vamos começar por analisar um dos seus *datasets* mais populares
 - *Informação mais completa sobre todos os datasets disponibilizados pela Scikit-learn, pode ser obtida na sua documentação oficial:*

<http://scikit-learn.org/stable/datasets/index.html>
- Prosseguimos depois com a implementação dum primeiro modelo de ML através da biblioteca ‘Scikit-Learn’, de forma a ilustrarmos todas as fases de desenvolvimento incluídas neste tipo de projetos.

O dataset Iris

- Um dos *datasets* mais populares entre a comunidade da Data Science é o Iris
- Trata-se de um dataset criado pelo britânico Ronald Fisher, contendo registros de três espécies de plantas iris (versicolor, virginica, setosa)
 - sendo formado por 150 linhas, com o registo das caraterísticas de 50 exemplares de cada tipo de planta
 - e por 5 colunas, com os valores de comprimento e largura quer das pétalas, quer das sépalas, e com a classificação da planta numa das três espécies
- Este *dataset* é muito usado para construir modelos de ML que, com base nos 4 atributos de cada exemplar, classifique a planta numa das 3 espécies



O dataset Iris da Scikit-learn

- O Iris é um dos datasets incluídos no Scikit_learn
 - *para obtê-lo, invocamos a função load_iris() do módulo 'datasets'*

```
from sklearn import datasets
iris = datasets.load_iris()
```

```
type(iris)
sklearn.utils.Bunch
```

- O objeto carregado não é um DataFrame
 - *trata-se de um objeto especial que, tendo a estrutura de um dicionário, permite que acessemos a cada uma das suas partes através de atributos (chaves)*

```
dir(iris)
```

```
['DESCR', 'data', 'feature_names', 'filename', 'frame', 'target', 'target_names']
```

- Através do atributo 'feature_names', temos acesso ao nome das colunas (características)

```
iris.feature_names
```

```
['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal width (cm)']
```

- e através do atributo 'data' aos valores dessas colunas (características de cada exemplar)

```
iris.data
```

```
array([[5.1, 3.5, 1.4, 0.2],
       [4.9, 3. , 1.4, 0.2],
       [4.7, 3.2,
       [4.6
```

```
iris.data.shape
```

```
(150, 4)
```


Outros *datasets* do Scikit-learn

- Par além do Iris, o Scikit-learn disponibiliza-nos muitos outros datasets interessantes, como por exemplo:

- *dados do cancro da mama (para classificação)*

```
ds=datasets.load_breast_cancer()  
print('{} linhas e {} colunas.'.format(ds.data.shape[0],ds.data.shape[1]))
```

569 linhas e 30 colunas.

- *dados da diabetes (para regressão)*

```
ds=datasets.load_diabetes()  
print('{} linhas e {} colunas.'.format(ds.data.shape[0],ds.data.shape[1]))
```

442 linhas e 10 colunas.

- *imagens de dígitos escritos à mão (para classificação)*

```
ds=datasets.load_digits()  
print('{} linhas e {} colunas.'.format(ds.data.shape[0],ds.data.shape[1]))
```

1797 linhas e 64 colunas.

- *caraterísticas de vinhos de 3 produtores diferentes (para classificação)*

```
ds=datasets.load_wine()  
print('{} linhas e {} colunas.'.format(ds.data.shape[0],ds.data.shape[1]))
```

178 linhas e 13 colunas.

Primeira incursão na ML com a Scikit-learn

- A forma mais fácil de iniciarmo-nos na ML é através da regressão linear
 - *trata-se de um método linear que modela a relação entre uma variável dependente numérica (variável alvo) e uma ou mais variáveis independentes (variáveis explicativas)*
 - *diz-se linear porque a relação entre as variáveis independentes X e a variável dependente y é linear: $y=a+bx_1+cx_2+dx_3+...$*
- Por exemplo, duas variáveis que estão certamente correlacionadas são as horas de estudo dedicadas a uma unidade curricular e a respetiva nota
 - *se assumirmos que essa relação é aproximadamente linear, podemos então desenvolver um modelo da regressão linear que nos ajude a capturar essa relação*
- Na exposição que se segue vamos então usar regressão linear para resolver o seguinte problema:
 - *partindo-se de um histórico de notas e horas de estudo despendidas por 15 alunos (valor demasiado baixo para se poder induzir um bom modelo) numa determinada unidade curricular, criar um modelo que, com base nesse histórico, consiga prever qual a nota esperada para um aluno que se apresente a exame com 2 semanas de estudo*

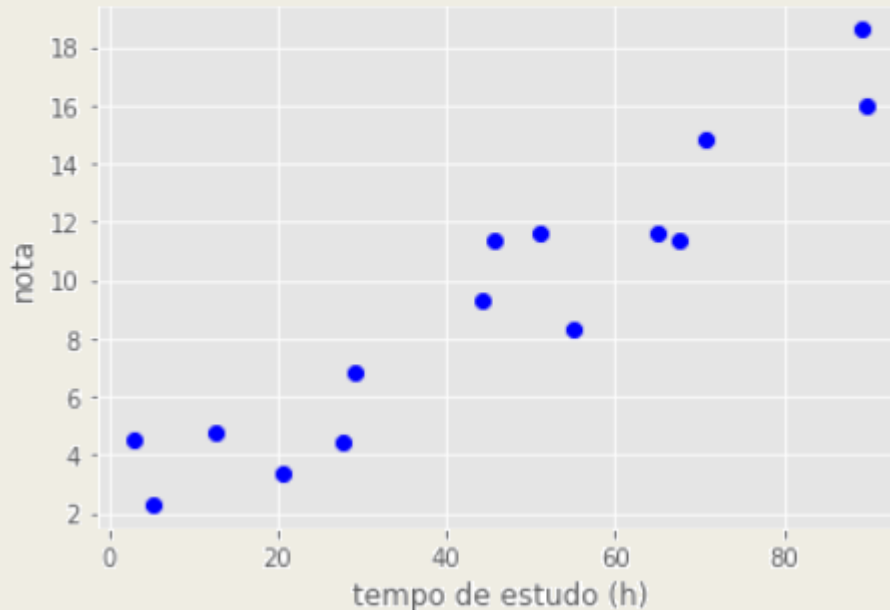
O dataset com notas e horas de estudo

- Considere-se que se dispõe do registo das horas de estudo e respetivas notas de 15 alunos

```
horas=[55.1,70.8,29.1,51.1,89.3,89.6,12.6,20.7,5.1,44.1,3.0,45.7,64.9,27.8,67.6]
```

```
notas=[8.3,14.8,6.8,11.6,18.6,16.0,4.8,3.4,2.3,9.3,4.5,11.4,11.6,4.4,11.4]
```

```
import matplotlib.pyplot as plt
plt.plot(horas, notas, 'ob')
plt.xlabel('tempo de estudo (h)'); plt.ylabel('nota');
```



Repare-se que neste exemplo há só 1 variável independente, mas poderiam ser várias (a ajudar a explicar a nota...)

Tipicamente, os algoritmos de ML do Scikit-learn requerem os dados no formato bidimensional

- convertamos, por isso, as nossas listas para arrays 2D, com 15 linhas e 1 coluna

```
import numpy as np
horas=np.array(horas).reshape(-1,1)
horas.shape
```

(15, 1)

```
notas=np.array(notas).reshape(-1,1)
notas.shape
```

(15, 1)

Dados de treino e dados de teste

- Uma etapa importante na criação dum modelo de ML é a sua fase de treino
 - *é nesta fase que o modelo aprende com os dados, ajustando a sua configuração de forma automática*
- Para além de se construir o modelo, vai ser necessário depois avaliar, de alguma forma, a sua eficácia (a sua capacidade de acerto), comparando as suas previsões com resultados conhecidos
 - *Essa avaliação será mais fiável se for realizada com dados que não foram usados no treino do modelo,*
 - *ou seja, a avaliação de desempenho do modelo deverá ser feita com dados completamente novos (dados que o modelo nunca viu)*
 - *Só assim se conseguirá medir devidamente a sua capacidade de **generalização***
- Repare-se que é natural que o modelo tenha um melhor desempenho com os dados de treino, uma vez que se ajustou a eles
 - *Porém, o que verdadeiramente interessa é a sua capacidade de aplicar e adaptar para novos dados o conhecimento que adquiriu com os dados de treino*
 - *ou seja, ter capacidade de generalização*
- Quantos mais dados de treino se usarem, maior será a capacidade de generalização do modelo
 - *Com poucos dados de treino, o modelo vai ajustar-se demasiado a esses dados (**overfitting**), comprometendo a sua capacidade de generalização (bom desempenho no treino, mas péssimo no teste)*

Particionar o *dataset*

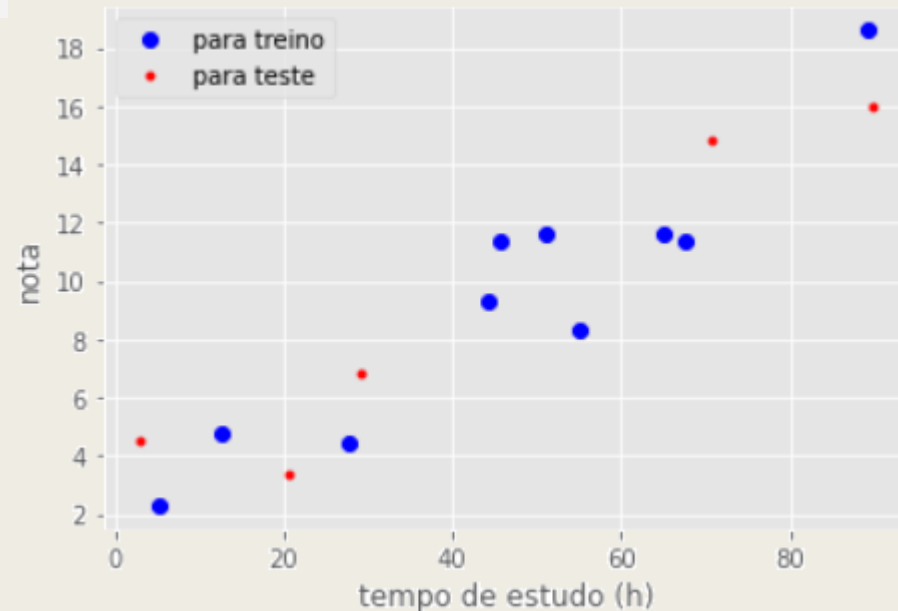
- Em vez de usarmos todos os dados disponíveis para treinar o nosso modelo, devemos então pôr alguns de parte para que, com eles, venhamos a conseguir testar de forma adequada o modelo final
 - *Felizmente, o Scikit-learn contém uma função, no seu módulo model_selection, que nos permite facilmente particionar o nosso dataset em 2 subconjuntos*

```
from sklearn.model_selection import train_test_split
horas_train, horas_test, notas_train, notas_test = train_test_split(horas, notas, test_size=0.33)
```

Por norma, usam-se mais dados para treino e menos para teste (por exemplo, entre 10 e 30%)

```
plt.plot(horas_train, notas_train, 'ob', label='para treino')
plt.plot(horas_test, notas_test, '.r', label='para teste')
plt.xlabel('tempo de estudo (h)'); plt.ylabel('nota');
plt.legend();
```

A função `train_test_split()` recebe o array com os valores das variáveis explicativas (*horas*) e o array com os valores da variável alvo (*notas*), e devolve cada um desses arrays desdobrado em dois, com os dados que serão usados para treino e para teste separados aleatoriamente na proporção indicada pelo parâmetro `test_size` (33% para teste)



Criar, treinar e prever

- Instanciando a classe LinearRegression do módulo linear_model da Scikit-learn, criamos um modelo de regressão linear

```
from sklearn.linear_model import LinearRegression  
modelo=LinearRegression()
```

- Depois, através do seu método fit(), treinamos o modelo com os subconjuntos das horas e das notas que selecionámos para esse fim

```
modelo.fit(X=horas_train, y=notas_train)
```

- Uma vez criado e treinado o modelo, podemos já pô-lo a fazer previsões através do seu método predict(), encontrando a resposta para a questão colocada inicialmente

```
modelo.predict([[80]]) #nota prevista para duas semanas de estudo
```

```
array([[15.2]])
```

Nada mau, para quem trabalhou tão pouco....

- Mas tão ou mais importante que a previsão em si, é inteirarmo-nos da real capacidade deste estimador para fazer previsões fiáveis

```
notas_estimadas=modelo.predict(horas_test)
```

- Só agora os dados deixados para teste devem ser usados, pondo o modelo a fazer a previsão com esses dados

horas_test
array([[89.6], [70.8], [20.7], [29.1], [3.]])

notas_test
array([[16.], [14.8], [3.4], [6.8], [4.5]])

notas_estimadas
array([[16.8], [13.6], [5.], [6.4], [1.9]])

Interpretação gráfica dos dados do modelo

- Para um melhor entendimento do modelo de regressão linear que foi criado, visualizemos, juntamente com os dados de treino e de teste, a reta de regressão definida pelo modelo e as notas por si previstas para as horas de estudo do conjunto de treino

```
plt.plot(horas_train, notas_train, 'ob', label='para treino')
plt.plot(horas_test, notas_test, '.r', label='para teste')
plt.plot(horas_test, notas_estimadas, label='reta de regressão')
plt.plot(horas_test, notas_estimadas, 'g.', label='notas estimadas')
plt.xlabel('tempo de estudo (h)'); plt.ylabel('nota');
plt.legend();
```



O desempenho do modelo será tanto maior quanto menor forem os erros cometidos nas suas previsões.

Existem várias formas de contabilizar esses erros, a que se dá o nome de métricas

Métricas de avaliação de desempenho

Seguem-se duas das métricas mais usadas na avaliação de modelos de regressão:

- *Raiz do Erro Quadrático Médio (RMSE – Root Mean Square Error)*

$$RMSE = \sqrt{\frac{\sum_{i=1}^n (y_i - f(x_i))^2}{n}}$$

- *Coeficiente de Determinação (R^2)*

$$R^2 = 1 - \frac{\sum_{i=1}^n (y_i - f(x_i))^2}{\sum_{i=1}^n (y_i - \bar{y})^2}$$

sendo:

- n o número de observações usadas para teste
- y_i o i -ésimo valor real da variável dependente
- \bar{y} a média dos valores reais da variável dependente
- x_i o i -ésimo valor real da variável independente
- $f(x_i)$ o i -ésimo valor previsto da variável dependente (o valor previsto pelo modelo para y_i)
- $y_i - f(x_i)$ o i -ésimo erro cometido pelo modelo de previsão

Avaliação do modelo com o conjunto de teste – usando a métrica RMSE

- Para uma melhor compreensão, vamos aplicar a fórmula da RMSE em vários passos:

```
print(np.ravel(notas_test))
```

```
[16.  14.8  3.4  6.8  4.5]
```

Notas reais

```
print(np.ravel(notas_estimadas))
```

```
[16.8 13.6  5.   6.4  1.9]
```

Notas
previstas
pelo modelo

A função `ravel()` converte um array multidimensional num array com uma única dimensão (vetor) – usamos aqui esta função com o único propósito de mostrar o conjunto de valores numa só linha

```
erros=notas_test-notas_estimadas #erros das estimativas  
print(np.ravel(erros))
```

```
[-0.8  1.2 -1.6  0.4  2.6]
```

```
erros_quadraticos=erros**2 #quadrado dos erros  
print(np.ravel(erros_quadraticos))
```

```
[0.7 1.5 2.5 0.2 6.6]
```

```
                                #(Erro Quadrático Médio)  
print(round(erros_quadraticos.mean(),2))
```

```
2.28
```

```
RMSE=np.sqrt(erros_quadraticos.mean()) #Raíz do Erro Quadrático Médio  
print(round(RMSE,2))
```

```
1.51
```

Na prática, o erro RMSE é facilmente obtido por uma função do Scikit-learn, tal como se ilustra no fim do próximo diapositivo

Portanto o erro RMSE cometido pelo modelo de previsão foi de 1.51 (uma valor aceitável, dado o intervalo de variação da nota)

Avaliação do modelo com o conjunto de teste – usando a métrica R^2

- Para obtermos o Coeficiente de Determinação, basta invocarmos o método `score()` do próprio modelo:

```
R2=modelo.score(horas_test,notas_test)
print(round(R2,2))
```

0.92

- Ou seja, obteve-se um R^2 de 92%
 - que revela um ótimo desempenho(um R^2 de 0.92 significa que 92% da nota (variável dependente) é explicada pelo tempo de estudo (variável independente))
 - Note-se, no entanto, que este é apenas um exemplo meramente acadêmico, com um conjunto muito reduzido de dados simulados, que não permite que estes resultados possam ser levados demasiado a sério
 - em vez de datasets de treino e de teste de 10 e 5 alunos, respetivamente, exemplos mais realistas envolveriam normalmente centenas ou milhares de instâncias em ambos os conjuntos
- O Coeficiente de Determinação é, provavelmente, a medida de desempenho mais usada entre a comunidade da Data Science na avaliação de modelos de regressão
- Estas e muitas outras métricas, quer de regressão, quer de classificação, estão disponíveis no módulo ‘metrics’ da Scikit-learn

Para calcular o R^2 , o método `score()` começa por calcular as estimativas para as notas com base nas horas que recebe para, só depois, usando essas estimativas juntamente com as notas reais que também recebe, determinar R^2

```
from sklearn.metrics import r2_score, mean_squared_error
r2_score(notas_test,notas_estimadas) #devolve 0.92
mean_squared_error(notas_test,notas_estimadas,squared=False) #devolve 1.51
```

Com `squared=True`, calcula a MSE em vez da RMSE

Persistência do modelo

- Uma vez criado e treinado o modelo, é necessário preservá-lo para futuras utilizações
 - *Gravando o modelo, o mesmo ficará disponível para que futuras previsões se possam fazer rapidamente e sem necessidade de novos treinamentos*
- Dispomos de duas formas de gravar o modelo treinado, tornando-o persistente
 - *usando o módulo Pickle do Python,*
 - *ou através do módulo Joblib do Scikit-learn, que grava de forma otimizada modelos de machine learning treinados e outros objetos complexos como os arrays NumPy.*
- Como gravar o modelo com o Pickle

```
import pickle #para gravar em disco
f = open('modelo_horas_estudo.pck', 'wb')
pickle.dump(modelo, f)
f.close()
```

```
import pickle #para ler do disco
f = open('modelo_horas_estudo.pck', 'rb')
modelo = pickle.load(f)
f.close()
```

- Como gravar o modelo com o Joblib

```
import joblib #para gravar em disco
joblib.dump(modelo, 'modelo_horas_estudo.jbl')
```

Não precisamos de ser nós a abrir e a fechar o ficheiro

Com o Joblib acaba por ser mais simples

```
import joblib #para ler do disco
modelo = joblib.load('modelo_horas_estudo.jbl')
```

Pré-processamento

- Neste exemplo que acabámos de ilustrar, partimos de um *dataset* fictício, contendo, por isso, dados completos, sem ruído e no formato adequado para serem de imediato analisados
 - *Porém essa não é a realidade na maior parte dos datasets reais*
- Por norma, ainda antes da aplicação das técnicas de ML, os datasets precisam de ser devidamente preparados, envolvendo tarefas como
 - *seleção dos dados*
 - *limpeza dos dados*
 - *adaptação e transformação dos dados*
- Este conjunto de tarefas, que se destinam a preparar e a adequar os dados para a ML, constituem a fase de pré-processamento
 - *Trata-se de uma etapa vital em todo o processo de aquisição automática de conhecimento, por representar a maior parte do esforço despendido neste tipo de problemas (80% do tempo, segundo alguns estudos), e, principalmente, pelo importante impacto que inevitavelmente acaba por ter na qualidade do modelo desenvolvido*
- Para ilustrarmos as principais operações que podem ser realizadas durante a fase de pré-processamento, vamos usar, como exemplo, o *dataset* dos alunos de IA, mas desta vez, incluindo-lhe intencionalmente um conjunto de imperfeições que vão ter que ser corrigidas.

Dataset inicial

- Introduziram-se algumas imperfeições no *dataset* alunos, para o tornar mais realista...

alunos								
	nome	genero	freq	idade	presencas	freqAnt	notaIA	aprovado
30000	Tó	M	ordin	20	28.0	False	19.2	1
31234	Ana	F	trab	20	20.0	False	12.2	0
33333	Rui	M	erasm	25	3.0	True	5.3	0
40000	Gil	M	ordin	27	NaN	False	NaN	1
44444	Zé	M	ordin	23	NaN	True	15.9	1
34567	Ivo	M	trab	21	27.0	False	14.0	1
35000	José	M	ordin	21	28.0	False	14.0	1
36000	Joel	M	ordin	22	26.0	True	12.0	1
37000	Bia	F	ordin	65	27.0	True	9.0	0
38000	Luís	M	erasm	20	25.0	False	21.5	0
39000	Rita	F	erasm	21	27.0	False	18.0	1
41000	Lara	F	trab	23	5.0	True	7.0	0
99999	Lara	F	trab	23	5.0	True	7.0	0

Repare-se que, se a ideia for construir um modelo que venha a prever as notas dos alunos de IA (variável *notaIA*), esse modelo será de regressão; mas se o objetivo for prever se o aluno aprova ou não (variável *aprovado*), o modelo já será de classificação.

Limpeza dos dados

- Uma das primeiras tarefas que é necessário realizar na ML é a limpeza dos dados, eliminando ou substituindo dados inválidos, atípicos ou com valores omissos ou repetidos, que possam existir no *dataset*
- Nos *DataFrames* do *Pandas* e nos *arrays* do *NumPy*, os dados omissos encontram-se normalmente representados por *NaN* (Not a Number),
 - significando que o valor está ausente, indefinido ou não representável
- Os valores inválidos ou omissos, dependendo da situação, podem ser
 - eliminados, removendo as respectivas linhas,
 - ou substituídos por outros valores específicos que sejam válidos, normalmente inferidos a partir dos restantes valores da sua coluna, usando medidas de localização (média, mediana, moda)
- Seguidamente exemplificam-se diferentes tarefas de limpeza que podem ser realizadas sobre os dados

Substituição dos NaNs pela média da coluna

- Inspeccionando o DataFrame alunos, sendo de pequena dimensão, rapidamente conseguimos localizar as células com valores omissos (NaN)

presencas	freqAnt	notaIA
NaN	False	NaN
NaN	True	15.9
22.0	False	14.0

- Mas, tratando-se de datasets reais, com largas centenas ou milhares de linhas (alunos registados), o método booleano isnull() revela-se de grande utilidade,

- podendo , inclusivamente, ser combinado com o método de agregação sum(), caso pretendamos saber rapidamente quantos valores omissos ou nulos temos em cada uma das colunas*

```
alunos.isnull().sum()
nome          0
genero        0
freq          0
idade         0
presencas     2
freqAnt       0
notaIA        1
aprovado      0
```

- Uma forma de lidar com os NaN (valores omissos), é substituir cada um deles, por exemplo, pela média dos restantes valores da mesma coluna
 - para isso usa-se o método fillna() da respetiva coluna*
- Talvez esta não seja a opção indicada para a coluna notaIA, dado tratar-se, muito provavelmente, da variável alvo cujo valor não deverá de maneira alguma ser ficcionado
 - Mas já poderá ser um boa opção para a coluna ‘presencas’, principalmente se não se tratar de uma variável demasiado explicativa*

```
alunos.presencas = alunos.presencas.fillna(alunos.presencas.mean())
```

Após substituição dos NaNs da coluna 'presenças'

alunos								
	nome	genero	freq	idade	presenças	freqAnt	notaIA	aprovado
30000	Tó	M	ordin	20	28.000000	False	19.2	1
31234	Ana	F	trab	20	20.000000	False	12.2	0
33333	Rui	M	erasm	25	3.000000	True	5.3	0
40000	Gil	M	ordin	27	20.090909	False	NaN	1
44444	Zé	M	ordin	23	20.090909	True	15.9	1
34567	Ivo	M	trab	21	27.000000	False	14.0	1
35000	José	M	ordin	21	28.000000	False	14.0	1
36000	Joel	M	ordin	22	26.000000	True	12.0	1
37000	Bia	F	ordin	65	27.000000	True	9.0	0
38000	Luís	M	erasm	20	25.000000	False	21.5	0
39000	Rita	F	erasm	21	27.000000	False	18.0	1
41000	Lara	F	trab	23	5.000000	True	7.0	0
99999	Lara	F	trab	23	5.000000	True	7.0	0

Eliminação das linhas com NaNs

- No caso de ser o valor da variável dependente `notaIA` a estar ausente, a decisão mais acertada será mesmo remover toda a linha (no exemplo, remoção do aluno Gil)

```
alunos.dropna(inplace=True)
```

elimina todas as linhas que contenham NaNs

alunos								
	nome	genero	freq	idade	presencas	freqAnt	notaIA	aprovado
30000	Tó	M	ordin	20	28.000000	False	19.2	1
31234	Ana	F	trab	20	20.000000	False	12.2	0
33333	Rui	M	erasm	25	3.000000	True	5.3	0
44444	Zé	M	ordin	23	20.090909	True	15.9	1
34567	Ivo	M	trab	21	27.000000	False	14.0	1
35000	José	M	ordin	21	28.000000	False	14.0	1
36000	Joel	M	ordin	22	26.000000	True	12.0	1
37000	Bia	F	ordin	65	27.000000	True	9.0	0
38000	Luís	M	erasm	20	25.000000	False	21.5	0
39000	Rita	F	erasm	21	27.000000	False	18.0	1
41000	Lara	F	trab	23	5.000000	True	7.0	0
99999	Lara	F	trab	23	5.000000	True	7.0	0

Se se pretendesse apenas eliminar as linhas com NaNs em colunas específicas, indicavam-se essas colunas através do parâmetro `subset`

a linha 40000 foi eliminada

Ainda há uma 3ª via para eliminar os NaNs, que deverá ser adotada em último recurso: excluir do estudo as variáveis (colunas) onde eles apareçam. Esta opção só fará sentido quando a coluna tiver demasiados NaNs

Eliminação de linhas duplicadas

- Por vezes, os datasets reais contêm linhas com conteúdos repetidos que convém retirar
- Para localizar essas linhas duplicadas, pode ser usado o método `duplicated()`, que tem um parâmetro `keep` que pode assumir um dos seguintes valores:

- *False* – todas as linhas duplicadas são sinalizadas (classificadas como *True*)
- *first* – apenas a 1ª linha duplicada não é sinalizada
- *last* – apenas a última linha duplicada não é sinalizada

```
alunos.duplicated(keep=False)
30000    False
31234    False
33333    False
44444    False
34567    False
35000    False
36000    False
37000    False
38000    False
39000    False
41000     True
99999     True
```

- Para seleccionarmos todas as linhas duplicadas, basta indexar a DataFrame com a série booleana (máscara) que se obteve com o método anterior

```
alunos[alunos.duplicated(keep=False)]
```

	nome	genero	freq	idade	presencas	freqAnt	notaIA	aprovado
41000	Lara	F	trab	23	5.0	True	7.0	0
99999	Lara	F	trab	23	5.0	True	7.0	0

Eliminação de linhas duplicadas

- Finalmente, para eliminar as linhas repetidas, por exemplo mantendo sempre a primeira delas, deverá ser usado o método `drop_duplicates()` com o parâmetro `Keep='first'`

```
alunos.drop_duplicates(keep='first', inplace=True)
```

alunos								
	nome	genero	freq	idade	presencas	freqAnt	notaIA	aprovado
30000	Tó	M	ordin	20	28.000000	False	19.2	1
31234	Ana	F	trab	20	20.000000	False	12.2	0
33333	Rui	M	erasm	25	3.000000	True	5.3	0
44444	Zé	M	ordin	23	20.090909	True	15.9	1
34567	Ivo	M	trab	21	27.000000	False	14.0	1
35000	José	M	ordin	21	28.000000	False	14.0	1
36000	Joel	M	ordin	22	26.000000	True	12.0	1
37000	Bia	F	ordin	65	27.000000	True	9.0	0
38000	Luís	M	erasm	20	25.000000	False	21.5	0
39000	Rita	F	erasm	21	27.000000	False	18.0	1
41000	Lara	F	trab	23	5.000000	True	7.0	0

Desapareceu a última Lara

Eliminação de linhas duplicadas

- Por vezes, queremos eliminar linhas que repitam valores apenas em algumas colunas específicas (por exemplo, colunas que representem atributos chaves)
 - *Para eliminarmos, por exemplo, as linhas de alunos que tenham simultaneamente a mesma idade e a mesma nota, só temos que indicar essas colunas através do parâmetro subset*

```
alunos.drop_duplicates(subset=['idade', 'notaIA'], keep='last')
```

	nome	genero	freq	idade	presencas	freqAnt	notaIA	aprovado
30000	Tó	M	ordin	20	28.000000	False	19.2	1
31234	Ana	F	trab	20	20.000000	False	12.2	0
33333	Rui	M	erasm	25	3.000000	True	5.3	0
44444	Zé	M	ordin	23	20.090909	True	15.9	1
35000	José	M	ordin	21	28.000000	False	14.0	1
36000	Joel	M	ordin	22	26.000000	True	12.0	1
37000	Bia	F	ordin	65	27.000000	True	9.0	0
38000	Luís	M	erasm	20	25.000000	False	21.5	0
39000	Rita	F	erasm	21	27.000000	False	18.0	1
41000	Lara	F	trab	23	5.000000	True	7.0	0

Não aparece o Ivo, pois tinha a mesma idade e nota que o José (que era o last)

Repare-se que, como não se usou 'inplace=True', o DataFrame alunos não foi alterado. Pretendeu-se, com este comando, apenas exemplificar como se removeriam as linhas

Eliminação de linhas com valores inválidos

- Vimos como lidar com valores omissos e linhas repetidas. E se o nosso dataset contiver valores não admissíveis?
 - *Podemos começar por identificar as linhas com valores inválidos, usando o método `index()` nas linhas que forem seleccionadas pela condição que verifica se o valor é inválido*

```
labels_linhas=alunos[alunos.notaIA>20].index  
Int64Index([38000], dtype='int64')
```

e depois eliminá-las

```
alunos.drop(labels_linhas, inplace=True)
```

alunos

	nome	genero	freq	idade	presencas	freqAnt	notaIA	aprovado
30000	Tó	M	ordin	20	28.000000	False	19.2	1
31234	Ana	F	trab	20	20.000000	False	12.2	0
33333	Rui	M	erasm	25	3.000000	True	5.3	0
44444	Zé	M	ordin	23	20.090909	True	15.9	1
34567	Ivo	M	trab	21	27.000000	False	14.0	1
35000	José	M	ordin	21	28.000000	False	14.0	1
36000	Joel	M	ordin	22	26.000000	True	12.0	1
37000	Bia	F	ordin	65	27.000000	True	9.0	0
39000	Rita	F	erasm	21	27.000000	False	18.0	1
41000	Lara	F	trab	23	5.000000	True	7.0	0

*a linha
38000 foi
eliminada*

Eliminação de outliers

- Um *outlier* é um valor atípico que se encontra bastante distanciado dos outros valores observados
 - *Independente de resultarem ou não de falhas ou erros, tratam-se de valores atípicos que por norma convém retirar do dataset*
 - *a sua inclusão pode induzir negativamente o modelo com informação errada ou relacionada com casos muito particulares e improváveis que não convém considerar, sob pena de comprometer o desempenho geral do modelo*
- Dois dos métodos que podem ser usados para identificar os outliers são o Z-score e o Tukey Fences
 - *O método Z-score contabiliza quantos desvios padrão um dado valor se encontra distanciado da média*
$$Z_{score} = (x_i - \bar{x}) / \sigma$$
com x_i o valor a testar, \bar{x} a média e σ o desvio padrão
 - Se o resultado do Z-score for maior que 3 ou inferior a -3, o valor é considerado outlier
 - *O método Tukey Fences usa como medida de referência a distância entre o 1º (Q_1) e o 3º (Q_3) quartis, para classificar como outlier todo o valor que se distancie desses dois quartis uma vez e meia essa distância. Ou seja, é outlier se*
 - superar $Q_3 + 1.5 \times (Q_3 - Q_1)$,
 - ou se for inferior a $Q_1 - 1.5 \times (Q_3 - Q_1)$.
- Vamos exemplificar os dois métodos descritos, tentando remover com eles os outliers que possam existir na coluna 'idade'

Eliminação de *outliers* pelo Z-score

- Para aplicarmos a fórmula $Z_{score} = (x_i - \bar{x})/\sigma$, começemos por calcular a média e o desvio padrão das idades

```
media=alunos.idade.mean()  
media
```

26.1

```
desvpd=alunos.idade.std()  
desvpd
```

13.755402978870197

- O Z-score é então facilmente calculado

```
(alunos.idade-media)/desvpd
```

```
30000    -0.443462  
31234    -0.443462  
33333    -0.079969  
44444    -0.225366  
34567    -0.370763  
35000    -0.370763  
36000    -0.298065  
37000     2.827980  
39000    -0.370763  
41000    -0.225366
```

- Não sendo nenhum desses valores maior que 3 nem menor que -3, conclui-se que, de acordo com o critério Z-score, não há outliers nas idades
 - nem mesmo a da Bia, que já não é propriamente uma jovem...*

Eliminação de *outliers* pelo *Tukey Fences*

Para aplicarmos o método Tukey Fences, começamos por determinar o 1º e 3º quartis com a função `percentile()` do NumPy

```
import numpy as np
q1, q3 = np.percentile(alunos.idade, [25, 75])
```

Calculamos os respectivos limites,

```
lim_inf=q1-1.5*(q3-q1)
18.0
```

21.0 23.0

```
lim_sup=q3+1.5*(q3-q1)
26.0
```

para facilmente contruirmos a condição com o critério de *outlier* que será usada na indexação das respectivas linhas (as linhas que contêm os *outliers*)

```
alunos[(alunos.idade<lim_inf)|(alunos.idade>lim_sup)]
```

	nome	genero	freq	idade	presencas	freqAnt	notaIA	aprovado
37000	Bia	F	ordin	65	27.0	True	9.0	0

*Com este critério,
a Bia não passa
despercebida...*

Usando depois o método `index()` conseguimos saber os labels dessas linhas

```
labels_linhas=alunos[(alunos.idade<lim_inf)|(alunos.idade>lim_sup)].index
Int64Index([37000], dtype='int64')
```

que serão usados, finalmente, para a sua remoção através do método `drop()`.

```
alunos.drop(labels_linhas, inplace=True)
```

Normalização de colunas

- Frequentemente, torna-se necessário normalizar os valores das colunas numéricas do *dataset*
- Essa normalização tem como objetivo passar os valores de todas as colunas para uma mesma escala, sem, no entanto, se perder a diferenciação que existe entre valores duma mesma coluna
- Trata-se de uma técnica crucial para o desempenho de determinados algoritmos de ML, como é o caso das redes neurais artificiais e das SVM (máquinas de vetores de suporte)
 - *A normalização é necessária para evitar dificuldades de cálculo desnecessárias e, principalmente, para evitar que atributos com valores de grande amplitude ofusquem os de reduzida amplitude*
 - *Caso contrário, aqueles que assumissem valores de maior amplitude acabariam por ter uma maior preponderância no treinamento do modelo.*
- As normalizações min-max e Z-score são duas das técnicas mais usadas
 - *min-max (passa o valor para o intervalo de 0 a 1)*
Aplica-se a cada coluna a transformação $x' = (x - x_{min}) / (x_{max} - x_{min})$
 - *Z-score (representa o valor pela sua distância à média, em número de desvios padrão)*
Aplica-se a cada coluna a transformação $x' = (x - \bar{x}) / \sigma$

Normalização de colunas

- Para aplicarmos a normalização, devemos começar por seleccionar apenas as colunas numéricas

```
colunasnum=alunos.select_dtypes(include='number')
```

colunasnum				
	idade	presencas	notaIA	aprovado
30000	20	28.000000	19.2	1
31234	20	20.000000	12.2	0
33333	25	3.000000	5.3	0
44444	23	20.090909	15.9	1
34567	21	27.000000	14.0	1
35000	21	28.000000	14.0	1
36000	22	26.000000	12.0	1
39000	21	27.000000	18.0	1
41000	23	5.000000	7.0	0

- Instanciando a classe MinMaxScaler do package sklearn.preprocessing, obtém-se o objeto que será capaz de aplicar a transformação min-max (objeto 'scaler')

```
from sklearn.preprocessing import MinMaxScaler  
scaler = MinMaxScaler()
```

Normalização de colunas

- Através do seu método `fit_transform()`, o objeto 'escalador' aplica a transformação min-max ao DataFrame das colunas numéricas

- resultando um array NumPy (scaled_values) com os valores já normalizados*

```
scaled_values = scaler.fit_transform(colunasnum)
array([[0.      , 1.      , 1.      , 1.      ],
       [0.      , 0.68   , 0.4964 , 0.      ],
       [1.      , 0.      , 0.      , 0.      ],
       [0.6     , 0.6836 , 0.7626 , 1.      ],
       [0.2     , 0.96   , 0.6259 , 1.      ],
       [0.2     , 1.      , 0.6259 , 1.      ],
       [0.4     , 0.92   , 0.482  , 1.      ],
       [0.2     , 0.96   , 0.9137 , 1.      ],
       [0.6     , 0.08   , 0.1223 , 0.      ]])
```

- Atribuindo às colunas numéricas do DataFrame inicial, o array com os valores transformados, concluímos o processo de normalização.

```
alunos[colunasnum.columns]=scaled_values
```

	alunos							
	nome	genero	freq	idade	presencas	freqAnt	notaIA	aprovado
30000	Tó	M	ordin	0.0	1.000000	False	1.000000	1.0
31234	Ana	F	trab	0.0	0.680000	False	0.496403	0.0
33333	Rui	M	erasm	1.0	0.000000	True	0.000000	0.0
44444	Zé	M	ordin	0.6	0.683636	True	0.762590	1.0
34567	Ivo	M	trab	0.2	0.960000	False	0.625899	1.0
35000	José	M	ordin	0.2	1.000000	False	0.625899	1.0
36000	Joel	M	ordin	0.4	0.920000	True	0.482014	1.0
39000	Rita	F	erasm	0.2	0.960000	False	0.913669	1.0
41000	Lara	F	trab	0.6	0.080000	True	0.122302	0.0

Repare-se que todos os valores numéricos do dataset estão dentro do intervalo de 0 a 1.

Normalização de colunas

- Querendo-se aplicar a normalização Z-score, o que teria que se mudar era apenas o objeto 'escalador', que passaria a ser uma instância da classe StandardScaler

```
from sklearn.preprocessing import StandardScaler  
scaler = StandardScaler()
```

```
scaled_values = scaler.fit_transform(colunasnum)
```

```
scaled_values
```

```
array([[ -1.1487,  0.8136,  1.4082,  0.7071],  
       [ -1.1487, -0.049 , -0.199 , -1.4142],  
       [  2.0821, -1.882 , -1.7832, -1.4142],  
       [  0.7898, -0.0392,  0.6505,  0.7071],  
       [ -0.5026,  0.7057,  0.2143,  0.7071],  
       [ -0.5026,  0.8136,  0.2143,  0.7071],  
       [  0.1436,  0.5979, -0.2449,  0.7071],  
       [ -0.5026,  0.7057,  1.1326,  0.7071],  
       [  0.7898, -1.6663, -1.3928, -1.4142]])
```

```
alunos[colunasnum.columns]=scaled_values
```

Ficamos assim com um panorama geral, ainda que simplificado, de todo o processo de desenvolvimento de um modelo de ML (não fizemos, por exemplo, a sua afinação - tuning).