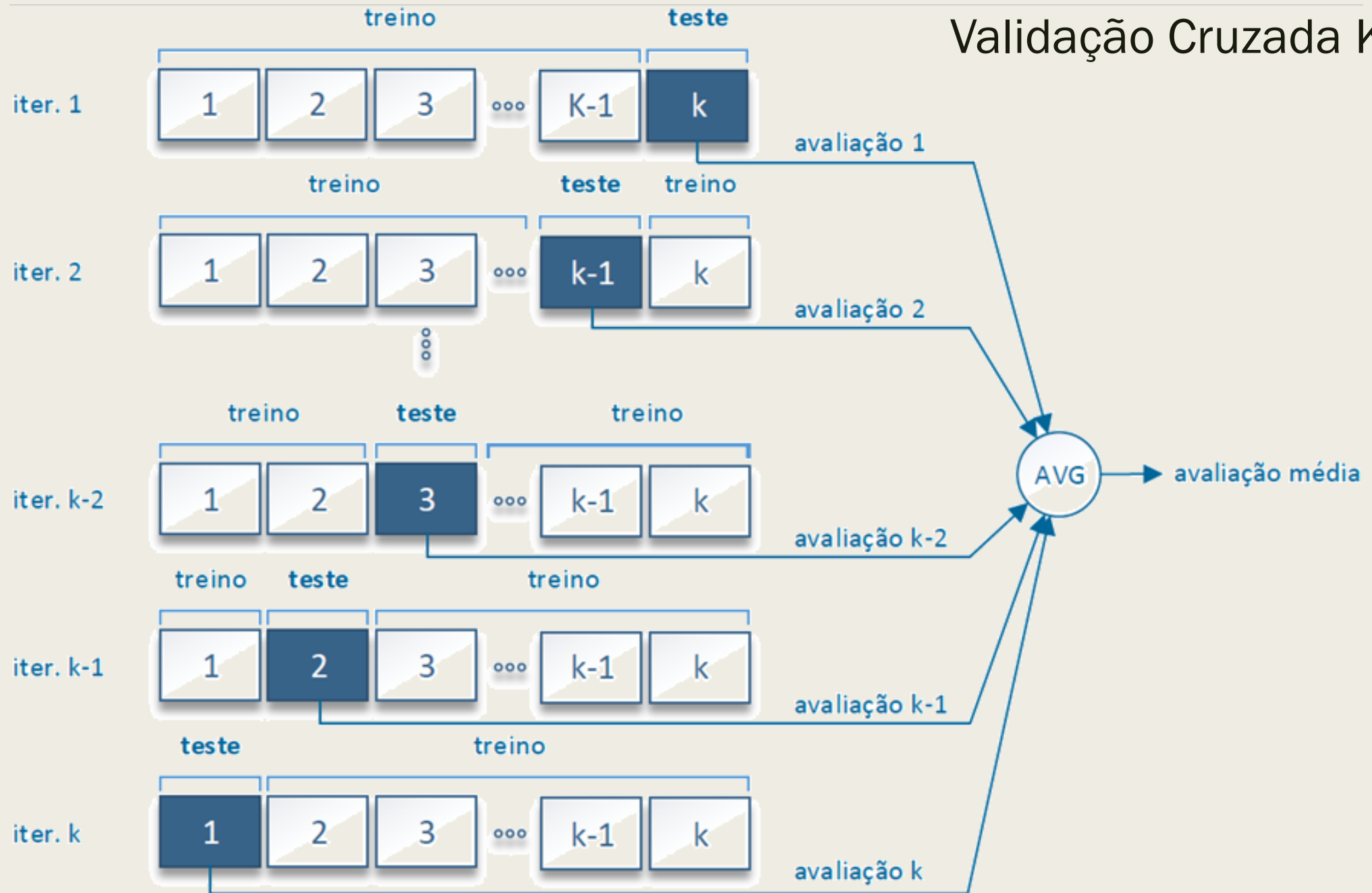


# Validação Cruzada K-folds

A literatura de *data mining* sugere que se calculem as métricas anteriores pelo método de validação cruzada

- A validação cruzada é uma técnica muito usada sempre que se pretende obter estimativas de desempenho preditivo mais **estáveis e fiáveis**,
  - *sendo especialmente indicada para a fase de **afinação** dos modelos (tuning)*
- No método de validação cruzada *K-folds*, o *dataset* inicial é dividido em **k partições** iguais (subconjuntos)
  - *Depois, iterativamente, pega-se em cada uma das k partições:*
    - e usa-se essa partição para teste e todas as restantes para treino
  - *O desempenho final do modelo é obtido pela média dos desempenhos observados sobre cada subconjunto de teste,*
    - conseguindo-se desta forma uma estimativa de desempenho que se julga mais consistente
- Um dos valores mais usados para o número de partições é o  $K=10$ .
- Esta técnica tem como inconveniente, relativamente ao particionamento simples, o aumento do esforço computacional

# Validação Cruzada K-folds



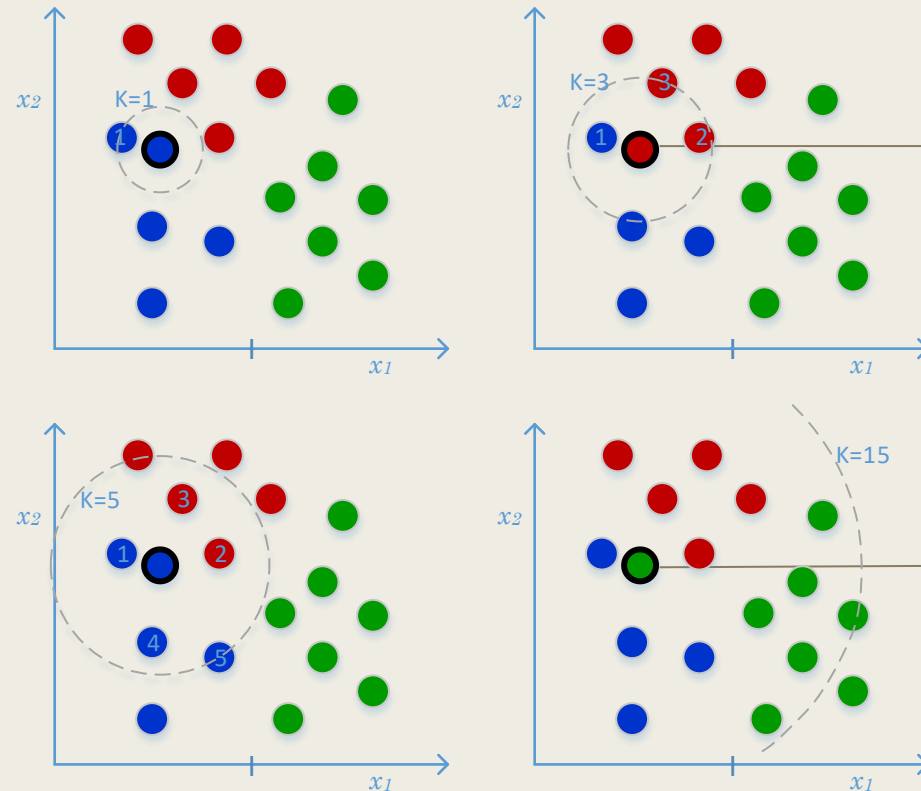
# Algoritmos de Machine Learning

- Seguem-se alguns dos algoritmos mais usados nos modelos de aprendizagem supervisionada
  - *Regressão Linear (para regressão)*
  - *Regressão Logística (para classificação)*
  - *Os K Vizinhos Mais Próximos (K-Nearest Neighbors – KNN)*
  - *Árvores de decisão*
  - *Florestas Aleatórias (Random Forests)*
  - *Máquinas de Vetores de Suporte (Support Vector Machines – SVM)*
  - *Redes Neurais (Neuronal Networks – NN)*
- Existem versões da maior parte destes algoritmos quer para regressão quer para classificação
- Todos estes algoritmos encontram-se disponíveis no *package* do Python Scikit-learn.

# KNN – Os K Vizinhos Mais Próximos

- O KNN (K-Nearest Neighbors), apesar de ser um dos algoritmos de classificação mais simples, apresenta, em determinados problemas, um desempenho até bastante aceitável
- Tal como o seu nome indicia, é um método que classifica os exemplos de teste com base na sua proximidade aos exemplos de treino
  - *Mais concretamente, na sua versão mais simples, atribui a cada exemplo de teste a classe mais frequente entre os  $K$  exemplos de treino que lhes sejam mais próximos*

Ilustração do KNN, para instâncias com 2 atributos,  $x_1$  e  $x_2$ , de 3 classes diferentes (vermelha, verde e azul), e para diferentes valores de  $K$  (quantidade de vizinhos considerados)



Exemplo de teste classificado como pertencendo à classe dos “vermelhos”, dado que é a classe predominante nos 3 exemplos de treino mais próximos

Exemplo de teste classificado como pertencendo à classe dos “verdes”, dado que é a classe predominante nos 15 exemplos de treino mais próximos

# KNN – Os K Vizinhos Mais Próximos

- A proximidade das instâncias de treino à instância de teste que se pretende classificar é determinada por uma **medida de distância**, sendo a mais comum a Euclidiana
- Repare-se que é muito fácil interpretar este tipo de algoritmo, percebendo-se de forma clara qual a regra que é usada na escolha da classe de cada exemplo de teste
  - *Porém, esta não é a realidade de outros algoritmos de ML mais complexos, como é o caso das redes neurais artificiais e das máquinas de vetores de suporte (SVM), sendo, por isso, conhecidos por algoritmos de “caixa preta”*
- Neste classificador não há propriamente a indução prévia de um modelo a partir dos dados de treino
  - *A “aprendizagem” ocorre em simultâneo com a classificação dos exemplos de teste (aprende como classificar cada exemplo de teste, comparando-o com os exemplos de treino)*
- O não treinamento prévio do modelo, tem custos computacionais
  - ***Grande parte do esforço é deferido para a fase de teste,***
  - *quando, na generalidade dos modelos ML, é o treino que requer a maior parte do esforço de processamento, sendo a fase de teste muito rápida*
    - É normalmente na fase de teste (a fase de aplicação do modelo) que a eficiência de execução se pode assumir como um fator crítico (exemplo: necessidade de classificar em tempo real)

# KNN – Quantos vizinhos? (K=?)

No exemplo ilustrado foram usados K=1, K=3, K=5 e K=15 vizinhos para classificar o exemplo de teste

- Neste caso, parece terem sido as escolhas K=1 e K=5 que levaram à classificação correta
  - *Mesmo a classe escolhida com K=3 parece aceitável, atendendo à proximidade da instância de teste com as de treino da classe vermelha*
- Mas já a classificação que se obtém com K=15 será claramente inadequada, atendendo ao grande distanciamento da instância em relação a todas as instâncias da classe a que foi associada (classe verde)
- Percebe-se, por isso, a importância do parâmetro K
  - **demasiados** vizinhos favorecem o **underfitting** (ajustamento pobre do modelo),
  - mas **poucos** vizinhos também conduzem, pelo contrário, ao **overfitting** (sobre-ajuste), tornando o modelo bastante sensível ao ruído
    - Por exemplo, para o caso extremo K=1 (a classe escolhida é a do vizinho mais próximo), o modelo faz depender a sua classificação de um único exemplo de treino, errando provavelmente na sua decisão sempre que esse exemplo seja um *outlier* e, por isso, ocupar a zona de influência de outra classe
    - Repare-se que com K=1 comporta-se sempre perfeitamente com os dados de treino (100% de acerto), mas terá certamente alguma dificuldade em lidar com dados novos
- Portanto, a afinação (*tuning*) dum modelo KNN passa essencialmente pela escolha do melhor valor para o parâmetro K, também designado **hiperparâmetro**, no contexto da ML
  - *Por norma, os algoritmos de ML incluem dois tipos de parâmetros: aqueles que são ajustados automaticamente durante a fase de treino (aprendizagem), e os hiperparâmetros, que teremos de ser nós a ajustá-los “manualmente”.*

# KNN – Afinar com dados de validação

- Prevendo-se que um modelo (KNN ou qualquer outro) vai ser sujeito a afinação, convém logo à partida particionar o *dataset* inicial em 3 subconjuntos (a não ser que se use validação cruzada):
  - *dados de treino* – *será com eles que o modelo vai aprender (habitualmente contém pelo menos metade dos exemplos do dataset inicial)*
  - *dados de validação* – *será com estes dados que o modelo vai ser avaliado nas várias iterações em que se testam diferentes valores de  $K$ , ou de outros hiperparâmetros (como estes dados vão influenciar o modelo, não servirão para o teste final)*
  - *dados de teste* – *para avaliar a verdadeira capacidade de generalização do modelo (teste derradeiro)*
    - sendo estes, dados nunca antes mostrados ao modelo, só eles nos darão uma indicação da sua verdadeira capacidade (o modelo final a que se chega, acaba por ser influenciado pelos dados de validação)
- De uma forma geral, sempre que se pretenda aperfeiçoar em várias iterações o modelo em desenvolvimento (seja através do ajuste de hiperparâmetros, seleção de características, de técnicas ou de outras operações, e estando em causa ou não o KNN), dever-se-á usar um subconjunto de dados específico para os sucessivos testes intermédios – que designamos **dados de validação**
- Por norma, no particionamento, a seleção dos exemplos para cada um dos subconjuntos deve ser feita de forma aleatória
  - *e estratificada, particularmente em datasets desbalanceados (com classes muito mais frequentes do que outras),*  
*de forma a garantir a mesma proporção de cada classe nos vários subconjuntos*



# KNN – variantes do modelo base

- Na versão mais simples de classificação de um exemplo de teste, a sua classe é escolhida por uma votação simples,
  - *sendo eleita a mais frequente entre o grupo de  $k$  vizinhos mais próximos,*
  - *através duma votação em que cada um dos  $k$  vizinhos tem o mesmo peso (importância), não importando a que proximidade se encontre da instância a classificar (mas dentro do grupo de  $k$  vizinhos, há sempre uns mais vizinhos do que outros...)*
- Versões mais evoluídas do algoritmo, usam uma votação ponderada, atribuindo maior peso aos votos dos vizinhos mais próximos
  - *como, por exemplo, o peso dado pelo inverso do quadrado da sua distância à instância a classificar,*
  - *conseguindo-se, com isso, uma menor sensibilidade do algoritmo à escolha do  $K$*
- Ainda que o KNN seja frequentemente ilustrado com problemas de classificação, à semelhança de muitos outros algoritmos de ML, pode ser facilmente adaptado a regressão
  - *Para isso, bastará escolher para a variável alvo (valor numérico) da instância de teste a média (que poderá ser ponderada) dos valores resposta associados às  $K$  instâncias vizinhas (em vez de se escolher a classe predominante)*



# KNN – Exemplo de aplicação

Com o auxílio do Scikit-learn, vamos aplicar o KNN na classificação das flores do *dataset* iris, para exemplificar, quer o algoritmo em si, quer o processo de afinação que normalmente é realizado na procura do melhor modelo

- Começemos por carregar o dataset iris e recordemos o que contém

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

iris.data

```
array([[5.1, 3.5, 1.4, 0.2],
       [4.9, 3. , 1.4, 0.2],
       [4.7, 3.2, 1.3, 0.4],
       [4.6, 3.1, 1.5, 0.5]])
```

```
iris.data.shape
```

 $(150, 4)$ 

```
iris.feature_names
```

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

iris.target

```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  
        ^   ^   ^   1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,  
          ^   ^   ^   1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,  
            ^   ^   ^   1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
```

valores da  
variável  
resposta

```
iris.target_names
```

```
array(['setosa', 'versicolor', 'virginica'], dtype='<U10')
```

iris.target.shape

(150,)

# Exemplo de aplicação – Particionamento do *dataset*

- Representemos as variáveis explicativas por X e a de resposta por y

```
X=iris.data  
y=iris.target
```

- Uma vez que pretendemos afinar o modelo de classificação, tentando encontrar o melhor valor para o hiperparâmetro K (i.e., o que maximize o desempenho do modelo), devemos começar por particionar o dataset *inicial* em 3 subconjuntos: de treino, de validação e de teste
  - Esses 3 subconjuntos podem ser obtidos executando a função train\_test\_split() duas vezes (na 1ª o dataset inicial é dividido em duas parcelas; na 2ª é a segunda parcela que é dividida em duas partes)*

Começamos por retirar para treino 50% dos exemplos

Depois, metade dos restantes vão para validação (25%) e a outra metade para teste (25%)

```
from sklearn.model_selection import train_test_split
```

```
Xtreino,Xrest,ytreino,yrest = train_test_split(X, y, test_size=0.5,  
random_state=1234, stratify=y)  
print(Xtreino.shape, Xrest.shape, ytreino.shape, yrest.shape)
```

para garantir as mesmas proporções de y nas várias partições

```
(75, 4) (75, 4) (75,) (75,)
```

para garantir que gera sempre as mesmas partições

```
Xvalid,Xteste,yvalid,yteste = train_test_split(Xrest, yrest, test_size = 0.5,  
random_state=1234, stratify=yrest)  
print(Xvalid.shape, Xteste.shape, yvalid.shape, yteste.shape)
```

```
(37, 4) (38, 4) (37,) (38,)
```

```
del X, y, Xrest, yrest
```

variáveis que não são mais necessárias

# Exemplo de aplicação – Afinação do modelo (*tuning*)

- A classe KNeighborsClassifier do Scikit-learn, permite-nos facilmente criar um modelo de classificação baseado no algoritmo KNN

```
from sklearn.neighbors import KNeighborsClassifier
```

- É na sua instanciação que deveremos definir o valor do hiperparâmetro K, indicando a quantidade de vizinhos a usar na classificação
- Como pretendemos otimizar o modelo, avaliamos o seu desempenho para diferentes valores de K

```
for k in range(1,11):  
    modelo = KNeighborsClassifier(n_neighbors=k)  
    modelo.fit(X=Xtreino, y=ytreino)  
    acuracia=modelo.score(X=Xvalid, y=yvalid)  
    print("Tx de acerto considerando k={} vizinhos: {:.1f}%".format(k,acuracia*100))
```

```
Tx de acerto considerando k=1 vizinhos: 94.6%  
Tx de acerto considerando k=2 vizinhos: 94.6%  
Tx de acerto considerando k=3 vizinhos: 94.6%  
Tx de acerto considerando k=4 vizinhos: 97.3%  
Tx de acerto considerando k=5 vizinhos: 97.3%  
Tx de acerto considerando k=6 vizinhos: 100.0%  
Tx de acerto considerando k=7 vizinhos: 97.3%  
Tx de acerto considerando k=8 vizinhos: 97.3%  
Tx de acerto considerando k=9 vizinhos: 97.3%  
Tx de acerto considerando k=10 vizinhos: 97.3%
```

- Pelos resultados obtidos percebe-se que o melhor desempenho é alcançado com 6 vizinhos (K=6)

Para métrica de desempenho usou-se a taxa de acerto. Sendo 100%, significa que todas as flores do conjunto de validação foram corretamente classificadas (numa das 3 espécies)

# Exemplo – Teste do modelo

- Para conseguirmos avaliar a capacidade de generalização do modelo encontrado, precisamos de testá-lo com dados que nunca viu...
  - *Esses dados ficaram resguardados, precisamente nas variáveis Xteste e yteste, para poderem agora ser usados*

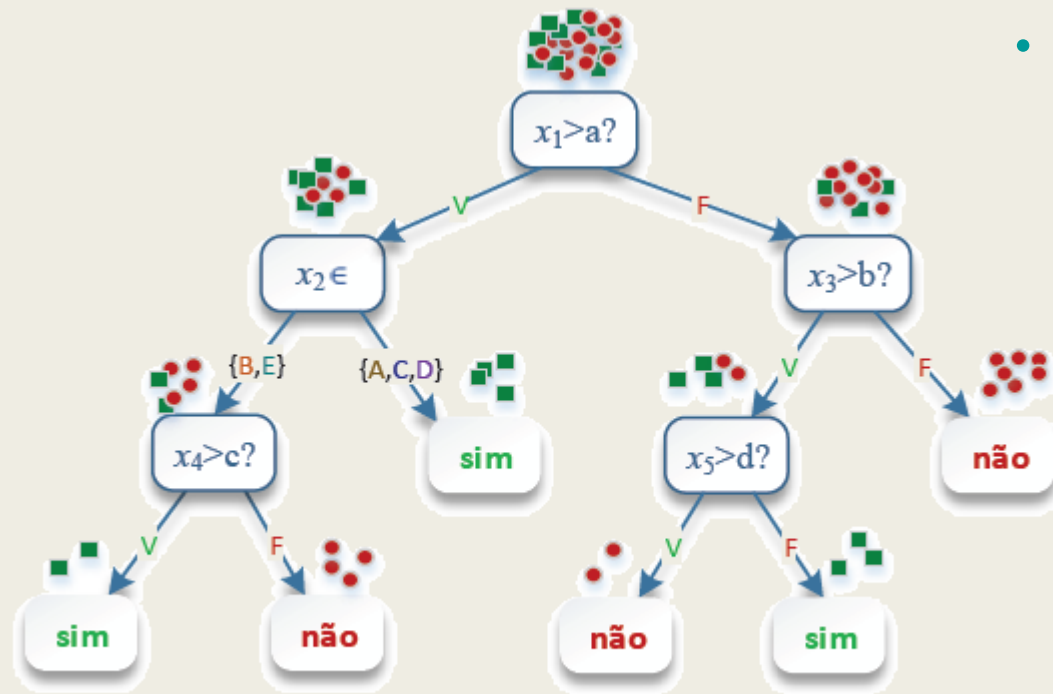
```
k=6
modelo = KNeighborsClassifier(n_neighbors=k)
modelo.fit(X=Xtreino, y=ytreino)
acuracia=modelo.score(X=Xteste, y=yteste)
print("Tx de acerto do melhor modelo (k={}), no conjunto de teste: {:.1f}%".format(k,
acuracia*100))
```

Tx de acerto do melhor modelo (k=6), no conjunto de teste: 97.4%

- Como a taxa de acerto nos dados de teste é 97.4%, pode concluir-se que o modelo encontrado tem uma boa capacidade de generalização (que é o que sempre se pertente neste tipo de modelos),
  - *Mantem um desempenho muito elevado mesmo quando aplicado a dados novos (a sua acurácia baixou apenas 2.6%)*
- Saliente-se porém que há sempre algum grau de subjetividade nos resultados dos modelos de ML, não devendo, por isso, ser interpretados com excessiva rigidez...
  - *dada a quantidade e a representatividade dos exemplos dos datasets raramente ser a ideal (150 exemplos de classificações de flores, no dataset iris, é manifestamente pouco)*
  - *e devido à componente estocástica que normalmente caracteriza, quer o particionamento dos datasets, quer os próprios algoritmos de ML (nem todos)*

# Árvores de decisão

- Uma árvore de decisão apresenta um conjunto de regras de classificação organizadas segundo uma estrutura em forma de árvore
- As regras, encontrando-se organizadas hierarquicamente, vão separando os dados iterativamente
  - *Cada nodo da árvore representa um critério de separação com base no valor que assuma um atributo específico*
  - *e cada ramo que sai desse nodo representa um dos possíveis resultados desse critério.*



Árvore de decisão para classificação binária.

- No treinamento (construção automática) da árvore, o algoritmo começa por escolher para o nodo raiz da árvore o atributo e o critério que melhor separem os dados de treino,
  - *após o qual, e segundo esse critério de separação, os dados são divididos em 2 subconjuntos.*
  - Depois, para cada um desses subconjuntos cria-se um novo nodo, descendente do primeiro, e aplica-se o mesmo procedimento, escolhendo o atributo e critério que melhor separem o subconjunto, novamente em 2. E assim sucessivamente, até que os dados do subconjunto façam todos parte de uma mesma classe, ou se atinja um qualquer outro critério de paragem (número máximo de níveis, p. ex.).

# Funcionamento duma árvore de decisão

- Para um melhor entendimento do funcionamento deste tipo de classificador, foquemo-nos na árvore de decisão para classificação binária ilustrada na figura:
  - *Cada variável  $x_i$  representa um dos atributos que caracterizam as instâncias do dataset.*
  - *O atributo usado em cada nodo e o respetivo critério de separação são escolhidos de forma a maximizar a separação, em dois subgrupos, das instâncias de classe positiva das de classe negativa*
    - *para avaliar a qualidade da separação em cada nodo são usadas métricas próprias, como é o caso da entropia e do índice de Gini, que medem, ambas, a impureza dos subconjuntos.*
  - *Um atributo pode ocorrer mais do que uma vez na mesma árvore, podendo ser numérico ou categórico.*
  - *No exemplo, a variável  $x_2$  representa um atributo categórico nominal que pode assumir 5 classes distintas (A, B, C, D e E), e todas as restantes são atributos, necessariamente, numéricos ou de tipo categórico ordinal.*
  - *Na ilustração, junta-se ainda a cada nodo um conjunto de instâncias positivas e negativas, de forma ilustrar as sucessivas divisões que vão sofrendo os subconjuntos de dados à medida que lhes vão sendo aplicados os respetivos critérios de separação.*
  - *Neste exemplo, a árvore desempenha a sua função na perfeição, separando completamente as duas classes*
    - *Mas como o mais habitual é não ser possível alcançar a separação perfeita das classes, o algoritmo tem de incluir critérios adicionais de paragem, como a imposição de um número máximo de níveis que a árvore possa atingir ou quando se tornar impossível separar ainda mais as instâncias do subconjunto.*



# Treino e uso duma árvore de decisão

- A árvore de decisão foi uma das primeiras estruturas a suportar aprendizagem automática
  - *cada nodo que lhe é acrescentado de forma automática, com base nos dados de treino, traduz-se em mais conhecimento que ela adquire*
  - *espera-se que a árvore depois de contruída, nesse processo de treino supervisionado que vai acrescentado nodos um a um, fique habilitada a classificar instâncias futuras com um elevado grau de acerto*
- Já na fase de previsão, a árvore de decisão irá classificar cada exemplo, de acordo com o caminho que satisfizer as condições desde o nodo raiz até ao nodo terminal, sendo o exemplo classificado de acordo com a classe associada a esse último nodo.
  - *Por vezes, após a criação da árvore, são-lhe aplicadas técnicas de “poda”, de forma a expurgá-la de possíveis impurezas, contribuindo-se assim para que somente a informação considerada relevante seja usada na tomada de decisão (uma árvore demasiado complexa propicia o overfitting)*
- Uma árvore de decisão pode também ser adaptada para problemas de regressão, prevendo valores contínuos em vez de classes
  - *funciona de maneira semelhante, mas em vez de avaliar a pureza dos subconjuntos com base em métricas como o índice de Gini ou a entropia, utiliza métricas que avaliam a dispersão dos valores (como a MSE).*
  - *Para se fazer uma previsão, são então percorridos os ramos até se chegar a uma folha, sendo depois escolhida a média ou a mediana dos valores alvo dos exemplos de treino que foram parar a essa folha.*



# Vantagens e desvantagens das árvores de decisão

- Vantagens
  - *produzem regras de classificação fáceis de interpretar*
  - *podem ser adaptadas a problemas de regressão*
  - *são bastante eficientes na construção dos modelos*
  - *não são dependentes da escala das variáveis numéricas (não requerem normalização)*
  - *apresentam robustez à presença de outliers e a atributos redundantes ou irrelevantes.*
- Desvantagem
  - *perturbações do conjunto de treino podem provocar alterações consideráveis no modelo induzido*

# Árvores de Decisão– Exemplo de aplicação

Classifiquemos novamente as flores do *dataset* iris, mas usando agora uma árvore de decisão (AD), de forma a exemplificarmos, quer o algoritmo em si, quer a utilização do método de validação cruzada (VC) no processo de afinação do respetivo modelo

- *O Scikit-learn disponibiliza-nos tanto o algoritmo da AD como a própria ferramenta para fazer o tuning com VC*
- Começemos por criar as variáveis X e y, com os preditores e a variável a prever, respetivamente.

```
from sklearn import datasets
iris=datasets.load_iris()
X=iris.data; y=iris.target
```

- Como vamos usar VC, não devemos criar um subconjunto de dados específico para validação, uma vez que o próprio algoritmo de VC usa os dados de “treino”, quer para treino, quer para validação

```
from sklearn.model_selection import train_test_split
Xtreino,Xteste,ytreino,yteste = train_test_split(X, y, test_size=0.25,
random_state=1234, stratify=y)
print(Xtreino.shape, Xteste.shape, ytreino.shape, yteste.shape)
```

```
(112, 4) (38, 4) (112,) (38,)
```

# Exemplo de aplicação – Uma 1ª versão não afinada

Podemos começar por criar uma 1ª versão não afinada do modelo de classificação

- A classe DecisionTreeClassifier do Scikit-learn, permite-nos facilmente criar um classificador baseado numa AD

```
from sklearn.tree import DecisionTreeClassifier  
modelo = DecisionTreeClassifier()
```

- Uma avaliação preliminar pode ser realizada com o método score() do modelo

```
modelo.fit(Xtreino, ytreino)  
modelo.score(X=Xteste, y=yteste)
```

0.9473684210526315

- Mesmo sem afinação, este modelo já apresenta um desempenho bastante interessante
    - *Porém, vamos tentar otimizar ainda mais o modelo, para percebermos, com este exemplo, de que modo pode ser realizado o processo de afinação com VC no Scikit-learn*
- (naturalmente, esse processo de afinação com VC, mas com diferentes hiperparâmetros, pode depois vir a ser replicado no desenvolvimento de um qualquer outro modelo de ML)*

# Exemplo de aplicação – Afinação do modelo (*tuning*)

- Como pretendemos otimizar o modelo, avaliamos o seu desempenho para diferentes valores de alguns dos seus hiperparâmetros
  - A classe *GridSearchCV* ajuda-nos nesse processo, usando na procura do melhor modelo todas as combinações possíveis dos valores dos hiperparâmetros que lhe dermos, e assegurando ela própria que as avaliações nesse processo são realizadas por VC

```
from sklearn.model_selection import GridSearchCV
```

- Fornecemos-lhe, na forma de dicionário, os valores dos hiperparâmetros que pertentemos testar

```
grelha_valores={'criterion':['gini','entropy'], 'max_depth':[2,3,4,5,10]}
```

função que mede a qualidade da separação,  
usada para critério em cada nodo

altura máxima  
da árvore

```
modelo = DecisionTreeClassifier()  
procura_modelo = GridSearchCV(modelo,param_grid=grelha_valores,cv=5)  
procura_modelo.fit(X=Xtreino, y=ytreino)
```

nº de partições  
usadas na VC

# Exemplo de aplicação – O teste final

- Os hiperparâmetros ótimos podem então ser consultados através do atributo `best_params_`

```
procura_modelo.best_params_  
{'criterion': 'gini', 'max_depth': 2}
```

e o respetivo modelo no atributo `best_estimator_`

```
modelo_otimo=procura_modelo.best_estimator_
```

- A capacidade de generalização do modelo encontrado pode ser finalmente avaliada com os dados de teste

```
modelo_otimo.score(X=Xteste, y=yteste)  
  
0.9736842105263158
```

- Portanto, com a afinação do modelo, melhorou-se ainda mais a sua *performance*, conseguindo-se a taxa de acerto que já se tinha obtido com o algoritmo KNN.
- Como a procura exaustiva realizada pela classe `GridSearchCV` nem sempre é praticável, devido ao esforço computacional envolvido (quando usados vários hiperparâmetros com muitos valores e, particularmente, outros algoritmos de ML mais complexos), temos também a possibilidade de optar por uma procura mais eficiente (ainda que um pouco menos assertiva), usando a classe `RandomizedSearchCV`.
- Recorde-se, por fim, que a taxa de acerto dado pelo método `score()` não é a métrica mais indicada para avaliar um classificador (tente descobrir como usar o valor AUC na VC)

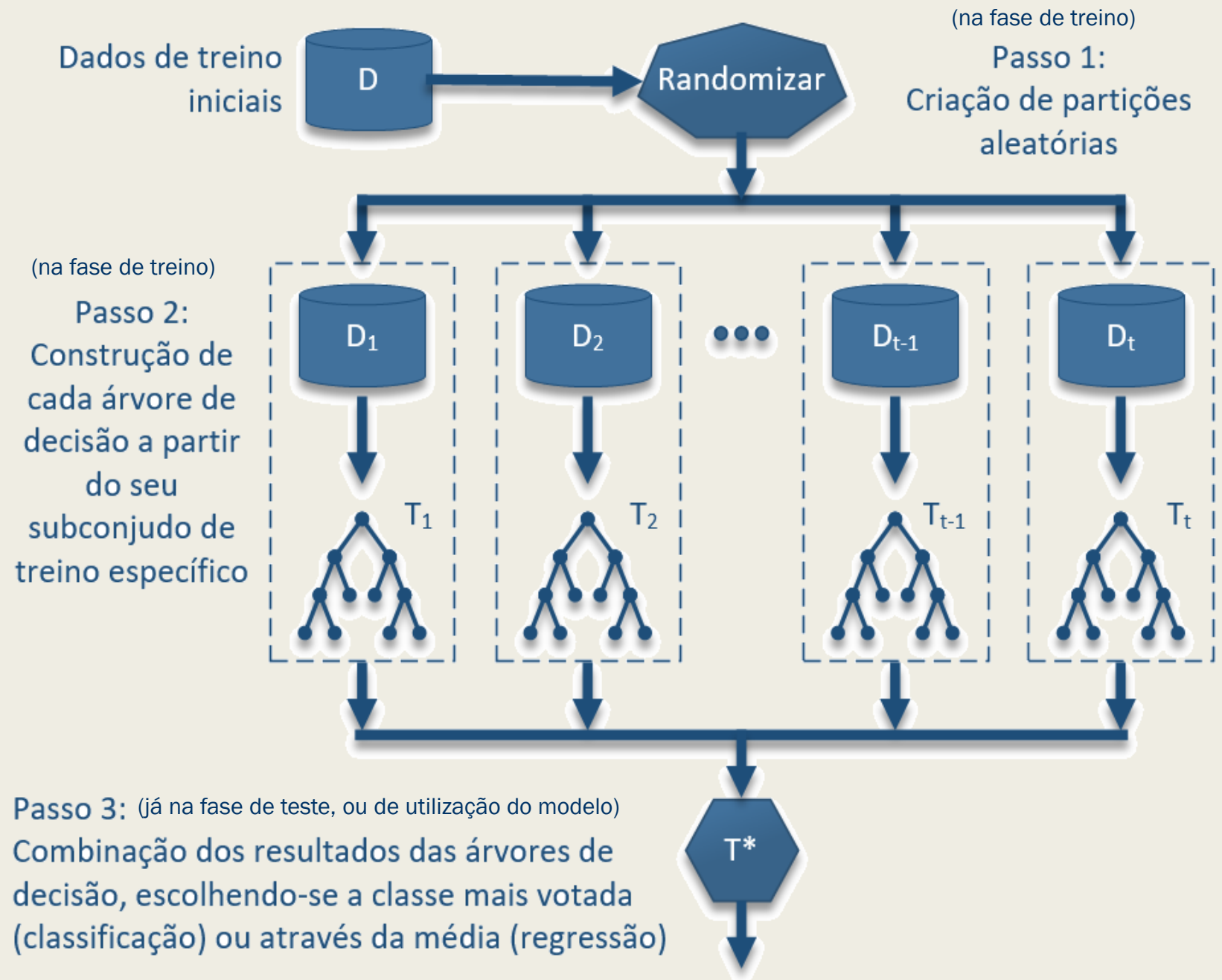
# Random Forest

- O algoritmo *random forest* (RF), proposto por Breiman<sup>1</sup>, é um método de aprendizagem baseada em comités<sup>2</sup>, que gera múltiplas árvores de decisão durante o treino
  - *Tem por base a premissa de que um conjunto de classificadores fracos pode criar um classificador forte*
  - *As RF combinam os resultados de múltiplas árvores de decisão treinadas individualmente, com padrões de erro diferentes, para tentar otimizar o desempenho preditivo global*
- Tal como ilustrado no esquema que se segue, para induzir individualmente cada uma das árvores da floresta, o algoritmo particiona aleatoriamente o conjunto de dados de treino inicial  $D$  (que contém  $n$  exemplos e  $d$  atributos) em múltiplos subconjuntos de treino de menor dimensão e dimensionalidade,  $D_1 \dots D_t$ ,
  - *cada um obtido de forma independente e por reamostragem aleatória, com reposição, do conjunto original.*
  - *Cada uma destas múltiplas partições, que contém  $m$  exemplos e  $i$  atributos, em que  $m < n$  e  $i < d$ , é utilizada para induzir uma diferente árvore da floresta*
  - *Os exemplos do conjunto de treino inicial  $D$  que não surjam no subconjunto de treino de uma árvore individual, os denominados dados out-of-bag, são utilizados como dados de teste para estimar o desempenho dessa árvore durante a fase de treino, conseguindo-se com isso uma estimativa fiável, uma vez que se tratam de dados novos*
  - *O processo de amostragem aleatória, quer dos exemplos, quer dos atributos, vai provocar o aumento da variabilidade das árvores da floresta, conseguindo-se, por essa via, reduzir a variância e o overfitting do modelo final.*

<sup>1</sup> Leo Breiman. Random forests. Machine learning, 45(1):5–32, 2001

<sup>2</sup> também designados métodos de conjunto ou ainda mistura de especialistas

# Random Forest





# Hiperparâmetros

- Entre os parâmetros mais importantes a afinar nas Random Forest, na procura do modelo com melhor desempenho, encontram-se:
  - *o número de árvores da floresta;*
  - *a altura máxima das árvores (profundidade);*
  - *o número de variáveis (escolhidas aleatoriamente) a considerar na procura do critério de separação em cada nodo – usam-se, como valores típicos, a raiz quadrada (em classificadores) e um terço do total de variáveis (em modelos de regressão);*
  - *a métrica para avaliar a qualidade da separação em cada nodo.*
- Para além dos hiperparâmetros, o método Random Forest do Scikit-Learn, tratando-se de um algoritmo altamente paralelizável, possui ainda um importante parâmetro, o `n_jobs`, que se destina a acelerar o treino do modelo,
  - *controla o número de tarefas a serem executadas em paralelo durante o treinamento das árvores;*
  - *quando positivo, especifica o número de árvores que são treinadas em paralelo;*
  - *se -1, o número de árvores treinadas em paralelo é igual ao número de núcleos do processador;*
  - *por omissão, não paraleliza.*

*(Este parâmetro também está presente nos K Vizinhos Mais Próximos e noutros algoritmos passíveis de paralelização)*

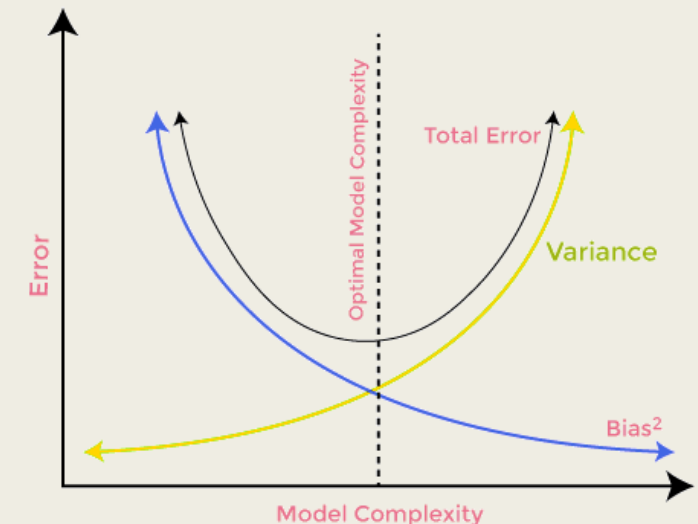
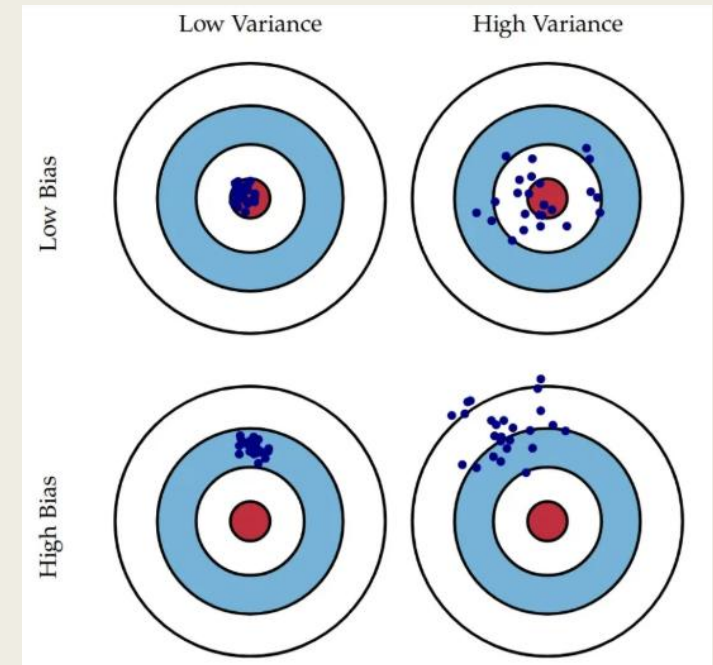
# Vantagens e desvantagens das random forests

Pode-se afirmar que as random forests são um dos algoritmos que melhores resultados oferece para um conjunto vasto de aplicações, mesmo que envolvam conjuntos de dados de grande dimensão (muitas instâncias) e de elevada dimensionalidade (muitos atributos)

- Vantagens
  - *geram estimativas com um bom equilíbrio entre enviesamento e variância (ver diapositivo seguinte)*
  - *não requerem a normalização das variáveis contínuas*
  - *capacidade de modelar relações não lineares*
  - *resistência ao overfitting*
  - *aptidão para lidar com dados categóricos (mas não as do sklearn – ver a seguir como resolver)*
  - *possibilidade de estimar a importância das variáveis preditivas*
  - *rapidez de construção e eficiência do estimador, mesmo em conjuntos de elevada dimensão e dimensionalidade (quando comparado com as SVM e as redes neuronais)*
- Desvantagens
  - *sensível a pequenas mudanças nos datasets*
  - *não lidam bem com atributos categóricos com elevado número de classes distintas*

# Variância vs Envieçamento

- Variância
  - Mede a sensibilidade do modelo às flutuações nos dados de treino.
  - Modelos com alta variância tendem a sobreajustar os dados de treino (overfitting), chegando a capturar até o "ruído" nos dados – mais frequente em modelos complexos.
- Envieçamento (Viés, Bias)
  - Tendência sistemática do modelo em errar em uma dada direção.
  - Modelos com alto envieçamento não capturam corretamente a relação subjacente dos dados de treino, podendo subajustar os dados (underfitting) – mais frequente em modelos simples que nem sequer conseguem aprender a estrutura dos dados de treino
- Trade-off (compromisso):
  - Por norma, quando se tenta diminuir um, aumenta-se o outro.
  - O objetivo é encontrar um equilíbrio em que o viés e a variância sejam suficientemente baixos para se obter uma boa generalização.

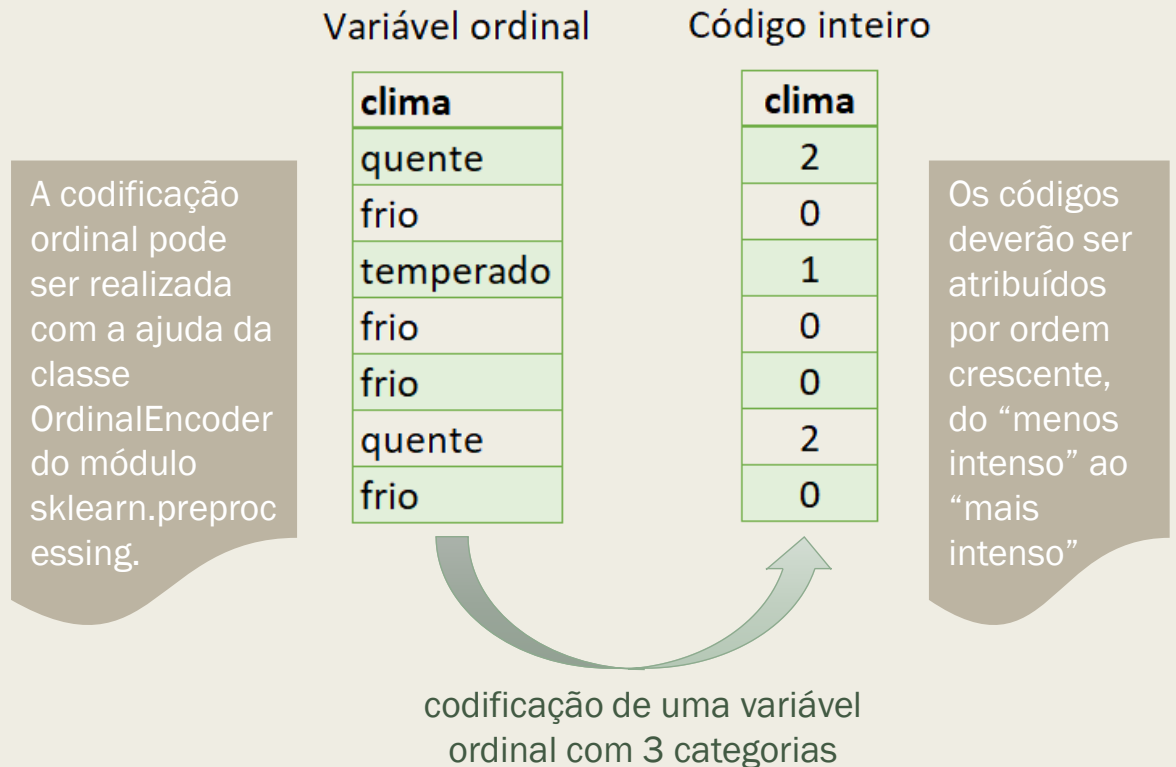


# Variáveis preditivas categóricas

- Muitos dos algoritmos de ML não conseguem lidar com preditores de tipo categórico
  - *é o caso das redes neuronais e das SVM, que estudaremos a seguir, uma vez que envolvem técnicas que esperam como entradas unicamente valores numéricos*
  - *Já os algoritmos baseados em árvores de decisão, tipicamente, funcionam bem com variáveis categóricas*
- Mas se estivermos a usar os Scikit-learn, então essa restrição é alargada a todos os seus algoritmos
  - *todos eles requerem que as entradas sejam numéricas (mesmo os que se baseiam em árvores de decisão)*
- O que fazer então às variáveis categóricas?
  - *Eliminá-las simplesmente, não é solução, pois, apesar da sua natureza, podem, tal como as numéricas, desempenhar um papel relevante na explicação da variável resposta*
  - *A única opção razoável passa então por convertê-las, de alguma forma, para valores numéricos – mas não de qualquer maneira...*

# Catégoricas ordinais vs nominais

- De acordo com a sua natureza, podemos ainda classificar as próprias variáveis catégoricas em duas tipologias diferentes
    - as ordinais – quando existe uma relação de ordem entre as várias categorias (labels/classes) assumidas pela variável, ao ponto de as podermos ordenar e comparar (Exemplo: ‘Mau’ < ‘Insuficiente’ < ‘Suficiente’ < ‘Bom’ < ‘Muito Bom’ < ‘Excelente’)*
    - e as nominais – quando não existe qualquer relação de ordem entre as várias categorias (Exemplo: ‘Vermelho’, ‘Verde’, ‘Azul’, ‘Branco’, ‘Preto’)*
  - Se a conversão para numérico se faz de forma simples no caso das variáveis ordinais,
    - bastando mapear para uma sequência de inteiros, de forma ordenada, as respetivas categorias (exemplo: ‘Mau’ → 0; ‘Insuficiente’ → 1; ‘Suficiente’ → 2; ‘Bom’ → 3; ‘Muito Bom’ → 4; ‘Excelente’ → 5),*
- o mesmo já não acontece com as nominais
- Para conversão das variáveis catégoricas nominais é normalmente usado um esquema de codificação mais elaborado, designado ‘one-hot encoding’, e que a seguir passamos a descrever*



# One-hot encoding

- Repare-se que se mapeássemos as categorias nominais diretamente para códigos inteiros induziríamos em erro o modelo de ML
  - *nessa situação, o algoritmo assumiria existir uma ordem natural entre as categorias, quando na verdade isso não acontece – todas as categorias estão ao mesmo nível (estariamos, no fundo, a ‘enganar’ o modelo)*
- O ‘one-hot encoding’ é então o esquema que normalmente se usa na ML para converter para numérico as variáveis categóricas nominais
- Nesse esquema de codificação, a variável nominal é desdobrada num conjunto de variáveis binárias mutuamente exclusivas, uma por cada categoria (label/classe)
  - *Mais concretamente, uma variável nominal de  $n$  categorias é substituída por  $n$  variáveis de 0s e 1s*
  - *Cada categoria (label/classe) dá origem a uma variável one-hot, ficando essa variável a 1 apenas nos exemplos do dataset que assumirem essa categoria*
- As novas variáveis que resultam do one-hot encoding são assim mutuamente exclusivas
  - *significando que, para cada exemplo do conjunto de dados, só uma delas ficará a 1 (todas as restantes estarão a 0)*

Variável nominal	One-hot encoding			
fruto	banana	laranja	maca	pera
pera	0	0	0	1
pera	0	0	0	1
laranja	0	1	0	0
banana	1	0	0	0
banana	1	0	0	0
pera	0	0	0	1
maca	0	0	1	0

codificação de uma variável nominal com 4 categorias

Para a codificação one-hot pode ser usada, quer a classe OneHotEncoder do módulo `sklearn.preprocessing`, quer a função `get_dummies()` do Pandas



# Variáveis *dummy*

- Vimos que a codificação *one-hot* cria uma variável binária por cada categoria
- Essa forma de representação envolve alguma redundância
  - *Como são mutuamente exclusivas entre si, uma delas acaba por ser dispensável*
  - *Por exemplo, o valor da 1ª variável pode ser sempre inferido a partir das restantes (será 1 quando todas as outras forem 0)*
- Quando se exclui uma das variáveis *one-hot* ficamos com um conjunto de  $n-1$  variáveis (sendo  $n$  o nº de categorias), a que se dá o nome de variáveis *dummy*
  - *Para além de se reduzir a redundância, alguns modelos de ML requerem que se usem estas variáveis, como é o caso da regressão linear*
- Será que às variáveis booleanas ou binárias também deve ser aplicada a codificação *one-hot*?
  - *Não, dado já estarem na representação *dummy* (uma variável de 0s e 1s a representar duas categorias)*
- No Scikit-Learn (e não só) é habitual os *datasets* representarem as variáveis nominais com códigos inteiros
  - *Dever-se-á ter especial atenção com estas situações, de forma a evitar que os algoritmos interpretem os códigos como grandezas numéricas (devemos continuar a olhar para elas como categóricas, que precisam de ser codificadas devidamente)*

A codificação para variáveis *dummy* pode ser realizada quer pela classe `OneHotEncoder` do módulo `sklearn.preprocessing`, usando o parâmetro `drop='first'`, quer pela função `get_dummies()` do `Pandas`, com o parâmetro `drop_first=True`.

Variável nominal	variáveis <i>dummy</i>		
fruto	laranja	maca	pera
pera	0	0	1
pera	0	0	1
laranja	1	0	0
banana	0	0	0
banana	0	0	0
pera	0	0	1
maca	0	1	0

codificação de uma variável nominal com 4 categorias