Sistemas Inteligentes - Machine Learning

Alberto Barbosa

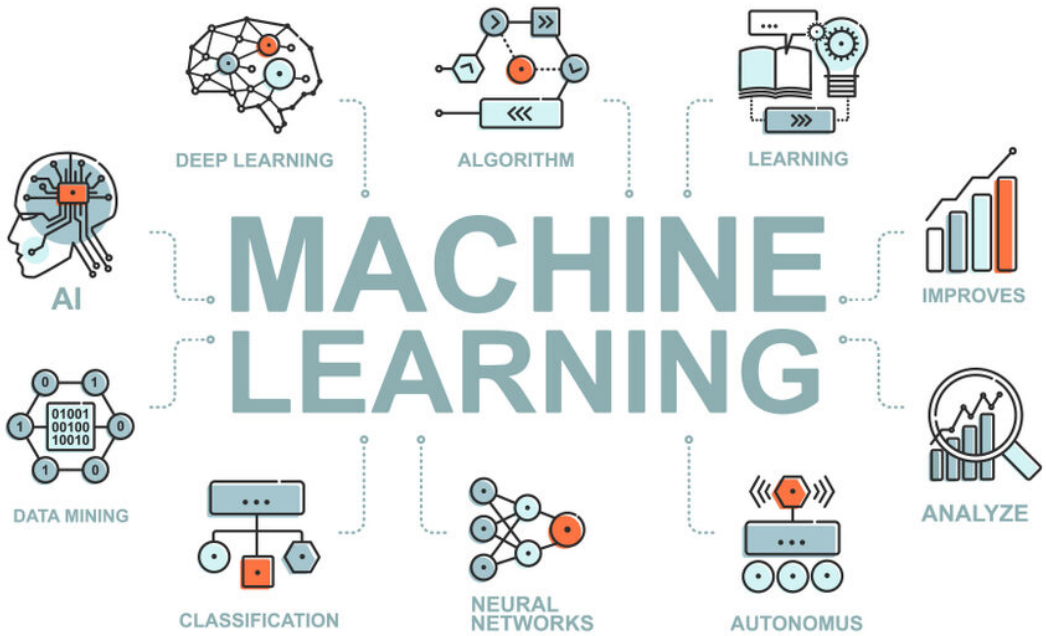
Universidade da Maia

alberto.barbosa@umaia.pt

May 26, 2025

Machine Learning

Podemos dizer que um agente inteligente está a aprender quando melhora o seu comportamento após fazer observações sobre o mundo.
Quando este agente é um computador, falamos de **machine learning**, onde um computador **analisa** dados, constrói um **modelo** baseado nesses dados e usa o modelo como uma **hipótese** sobre o mundo.



Machine Learning: Porquê?

Porquê deixar a máquina aprender quando podemos simplesmente fazer um programá-la da forma correcta para resolver o problema?
Primeiro, podemos estar a falar de um problema com **demasiada variabilidade**, o que torna impossível a capacidade de **antecipar** todos os casos.
Segundo, podemos **não saber** como resolver o problema, pelo que tentemos que a máquina aprenda a existência de **padrões** que nós não conseguimos descortinar ainda.

Tarefas em Machine Learning

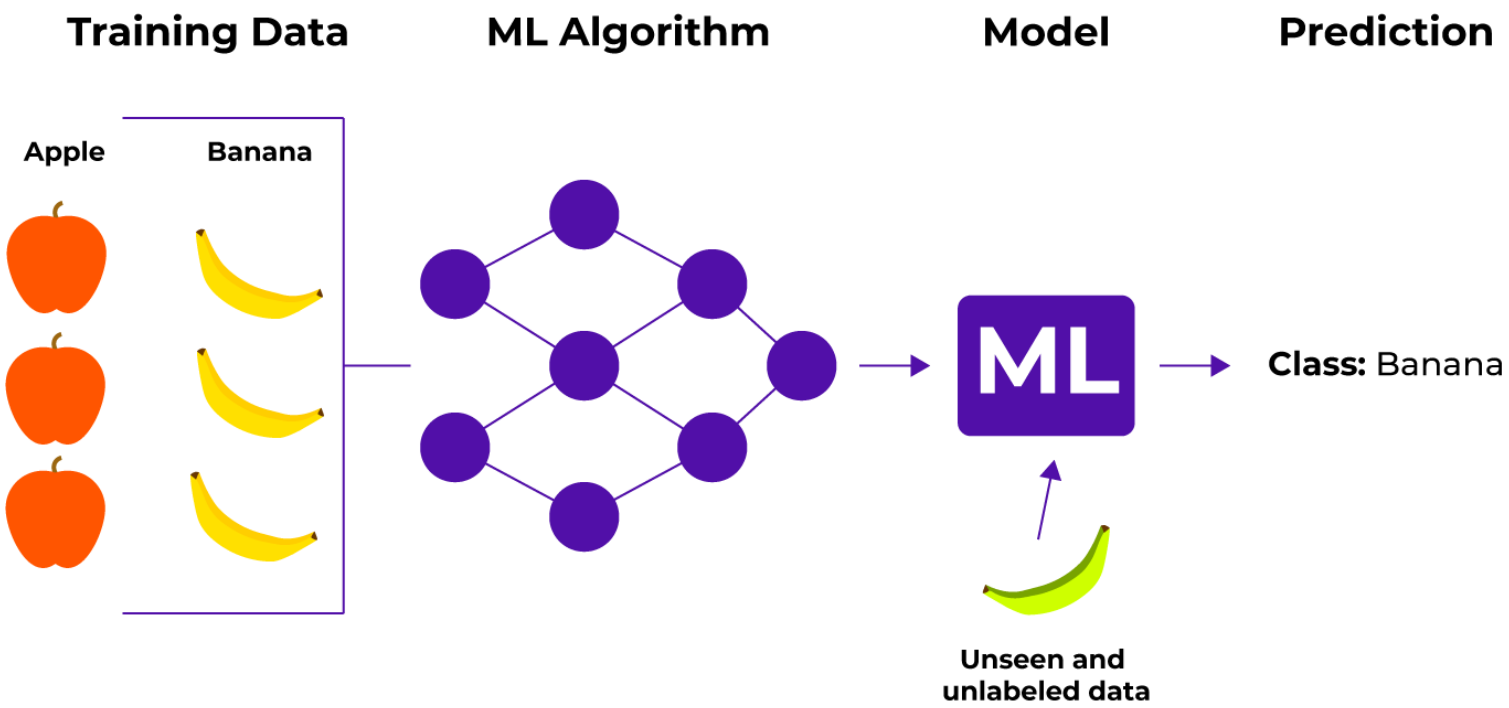
Genericamente podemos falar de três tipos de tarefas de aprendizagem:

- Aprendizagem supervisionada.
- Aprendizagem não supervisionada.
- Reinforcement Learning.

Aprendizagem Supervisionada

Na aprendizagem **supervisionada**, o agente observa **pares** de input-output e aprende uma **função** que **mapeia** o input no output. Aos outputs chamamos **labels**.
O agente irá aprender uma função que quando recebe um novo elemento de input, prevê uma nova label para esse input.
Um exemplo deste tipo de aprendizagem é receber uma imagem e dizer se a imagem retrata um gato.

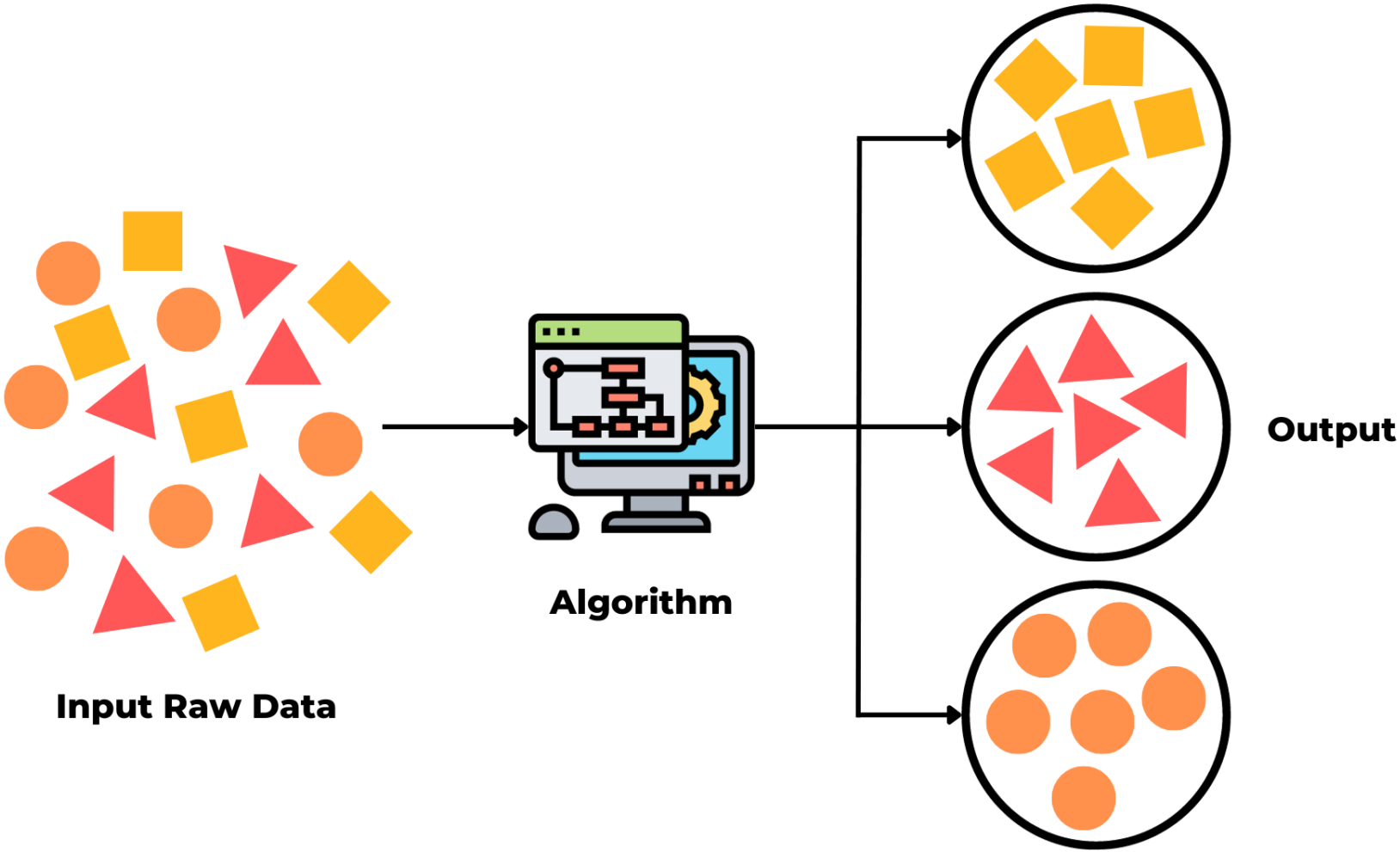
Aprendizagem Supervisionada



Aprendizagem não supervisionada

Na aprendizagem **não supervisionada**, o agente aprende padrões no input **sem qualquer label** a suportar a sua decisão.
Um exemplo deste tipo de aprendizagem é **clustering** onde o conjunto de objectos dados como input é divido em grupos de acordo com um determinado critério.

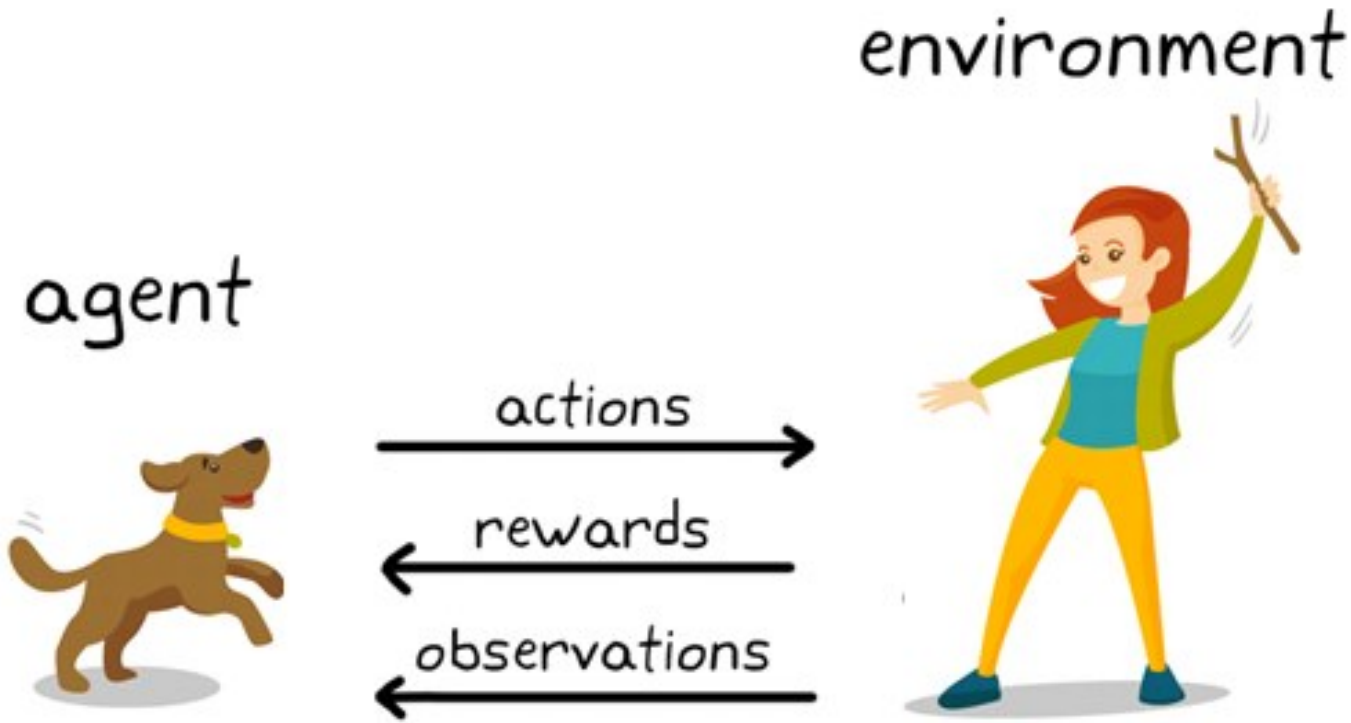
Aprendizagem não supervisionada



Reinforcement Learning

No caso de **reinforcement learning**, o agente aprende a partir de **reforço positivo e negativo**.
Por exemplo, no final de um jogo de xadrez o agente é reforçado positivamente caso ganhe e reforçado negativamente caso perca.

Reinforcement Learning



Aprendizagem Supervisionada

Uma tarefa de aprendizagem supervisionada pode ser formalmente definida da seguinte forma:
Dado um conjunto de treino com N pares input-output $(x_1, y_1), (x_2, y_2), \ldots (x_N, y_N)$ onde cada par foi gerado por uma função desconhecida $y = f(x)$, queremos descobrir uma função h que aproxima a verdadeira função f.

Aprendizagem Supervisionada

A essa função h chamamos **hipótese** e é parte integrante do espaço de hipóteses $\mathcal{H}$ de possíveis funções.
Também chamamos h de **modelo** dos dados, obtido através da classe de modelos $\mathcal{H}$. Ao output y_i chamamos **ground truth** pois é a resposta verdadeira que estamos a tentar que o nosso modelo preveja.

Avaliação do modelo

A melhor forma de avaliar o nosso modelo não será pela sua performance no conjunto de treino, mas sim como ele se comporta em dados **não vistos anteriormente**. Assim sendo, por norma existe um segundo conjunto de dados chamado **conjunto de teste** composto por pares (x_i, y_i) semelhante ao conjunto de treino.
Caso o modelo preveja de forma correcta os outputs no conjunto de teste, dizemos que o modelo **generaliza** bem.

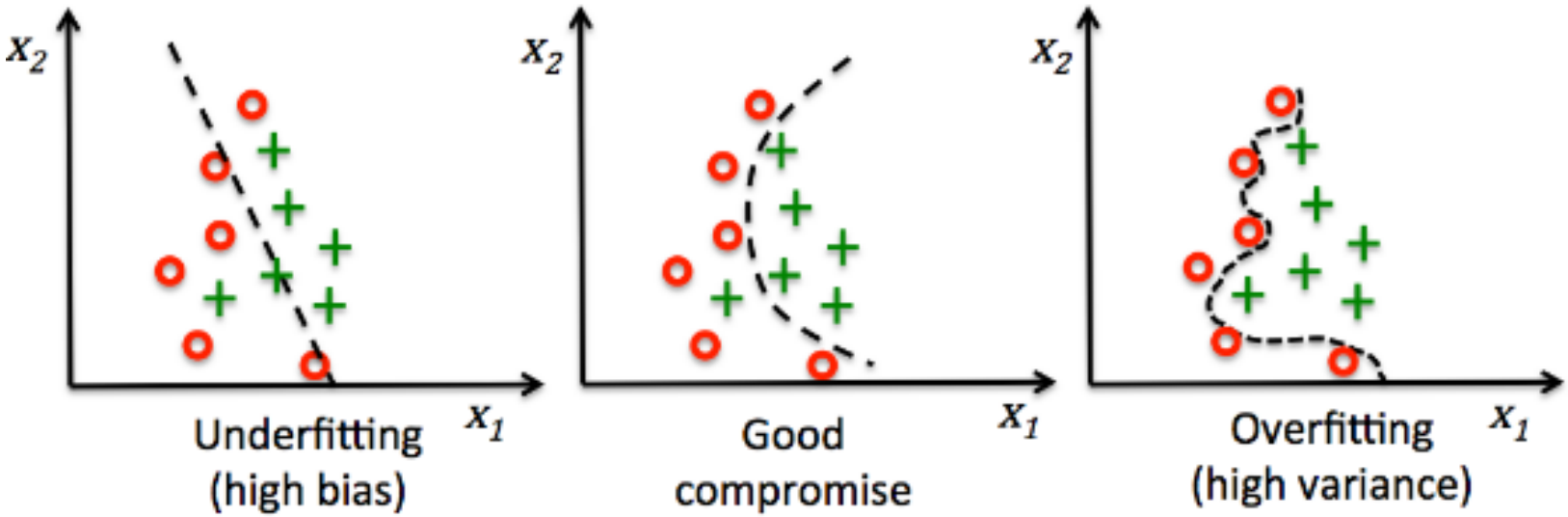
Bias e Variância

Dois conceitos importantes em Machine Learning são o **bias** e a **variância**. Bias é a tendência de um modelo se desviar de um valor esperado quando exposto a diferentes conjuntos de treino. O bias normalmente resulta de **restrições** impostas pelo **espaço de possíveis modelos**.
Por exemplo, o espaço de hipóteses das funções lineares impõe um bias aos modelos na medida em que o próprio modelo apenas pode ser constituído por uma linha recta. Caso existam padrões nos dados não capturados por uma linha recta, o modelo não será capaz de os representar e aprender.
Chamamos de **underfitting** à incapacidade do modelo de encontrar os devidos padrões nos dados.

Bias e Variância

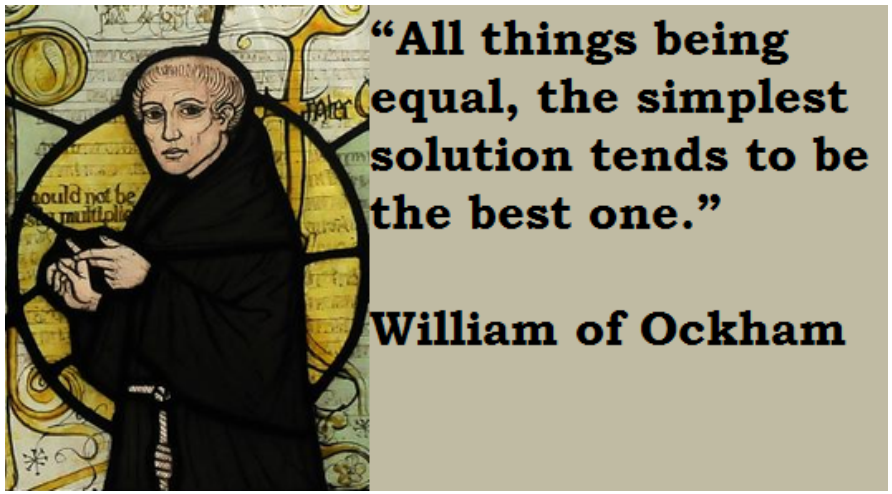
A variância consiste na quantidade de mudanças nos dados devido a **flutuações nos dados de treino**.
Chamamos de **overfitting** à excessiva capacidade de adaptação do modelo a um conjunto de dados em particular no qual foi treinado, fazendo com que o modelo tenha baixa performance em dados novos.

Bias e Variância



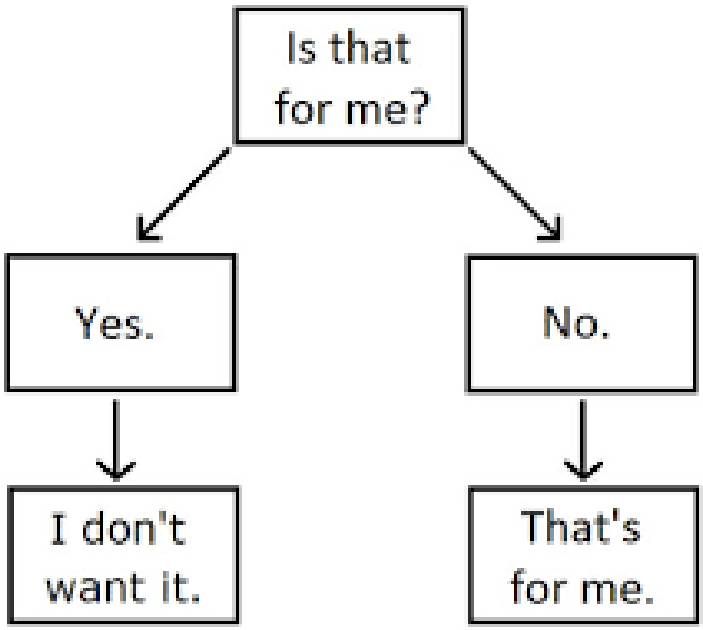
Bias e Variância

Normalmente tentamos equilibrar estes dois componentes, tendo em conta o **bias-variance tradeoff**: uma escolha entre modelos **mais complexos** com **baixo bias** que se adaptam bem aos dados de treino e modelos **mais simples** com **pouca variância** que generalizem melhor.
Um princípio útil de seguir é o da **Ockham's Razor**: Se temos duas possíveis respostas, mas uma é mais simples, provavelmente é a mais correcta.



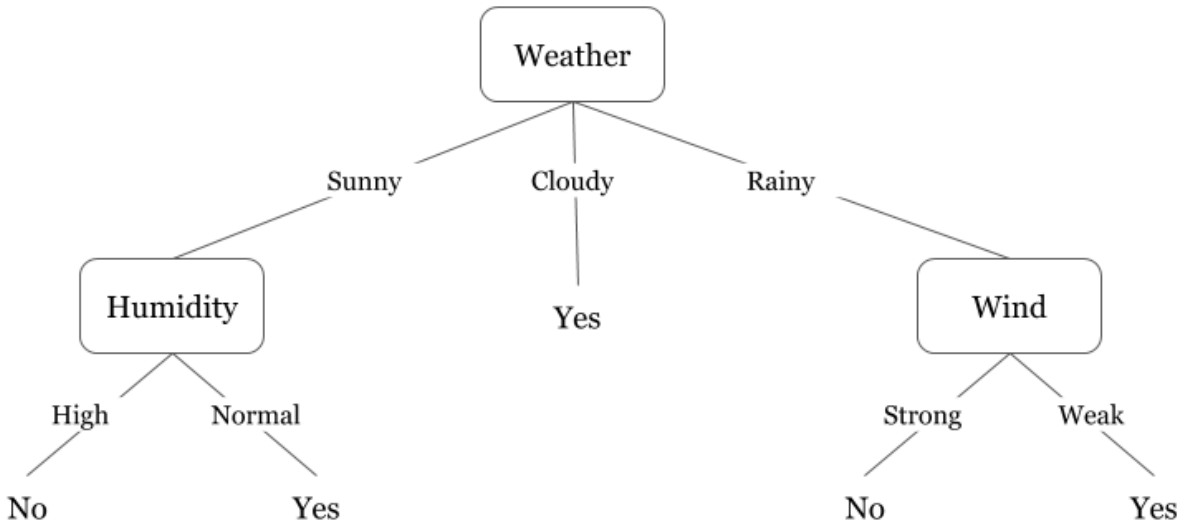
Decision Trees

Uma Decision Tree é uma representação de uma função que mapeia um vector de valores de atributos num único valor de output.
Uma árvore de decisão chega a um output após fazer uma sequência de testes começando na raíz da árvore e percorrendo os ramos certos até chegar a uma folha.



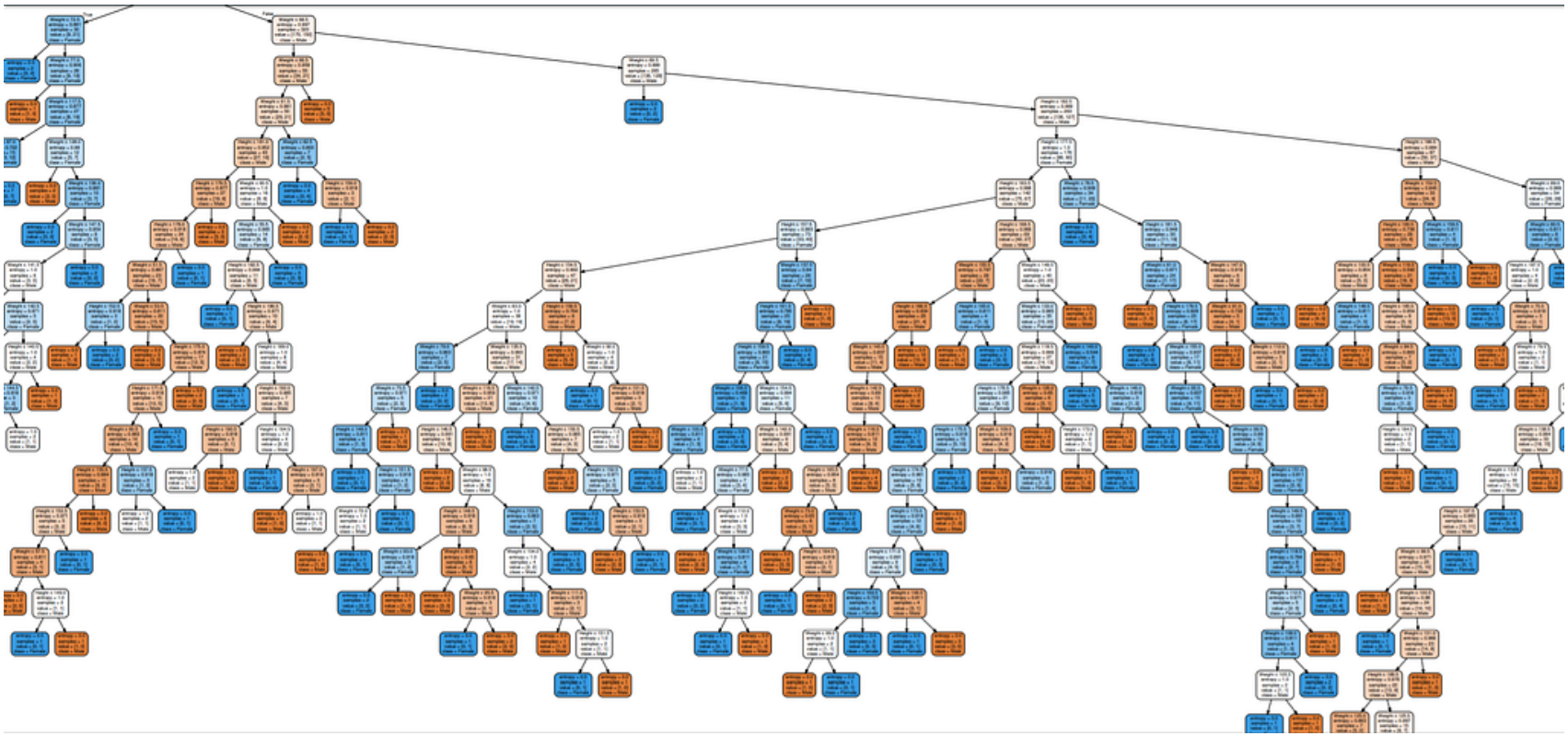
Decision Trees

Cada nó interno corresponde a um teste ao valor de um dos atributos, os ramos correspondem aos possíveis valores desse atributo e os as folhas especificam que valor deve ser retornado pela função.
Tanto o input como o output de uma árvore de decisão pode ser um valor discreto ou contínuo.



Expressividade

Para muitos casos, as árvores de decisão representam um resultado bastante fácil de perceber.
No entanto, existem casos onde as árvores de decisão não são a melhor forma de representação do modelo (função de paridade, por exemplo).



Como gerar árvores de decisão

Vamos analisar o seguinte conjunto de dados:

Example	Input Attributes										Goal
	Alt	*Bar*	*Fri*	*Hun*	*Pat*	*Price*	*Rain*	*Res*	*Type*	*Est*	*WillWait*
$\mathbf{x}_1$	*Yes*	*No*	*No*	*Yes*	*Some*	*$$$*	*No*	*Yes*	*French*	*0–10*	$y_1 = Yes$
$\mathbf{x}_2$	*Yes*	*No*	*No*	*Yes*	*Full*	*$*	*No*	*No*	*Thai*	*30–60*	$y_2 = No$
$\mathbf{x}_3$	*No*	*Yes*	*No*	*No*	*Some*	*$*	*No*	*No*	*Burger*	*0–10*	$y_3 = Yes$
$\mathbf{x}_4$	*Yes*	*No*	*Yes*	*Yes*	*Full*	*$*	*Yes*	*No*	*Thai*	*10–30*	$y_4 = Yes$
$\mathbf{x}_5$	*Yes*	*No*	*Yes*	*No*	*Full*	*$$$*	*No*	*Yes*	*French*	*>60*	$y_5 = No$
$\mathbf{x}_6$	*No*	*Yes*	*No*	*Yes*	*Some*	*$$*	*Yes*	*Yes*	*Italian*	*0–10*	$y_6 = Yes$
$\mathbf{x}_7$	*No*	*Yes*	*No*	*No*	*None*	*$*	*Yes*	*No*	*Burger*	*0–10*	$y_7 = No$
$\mathbf{x}_8$	*No*	*No*	*No*	*Yes*	*Some*	*$$*	*Yes*	*Yes*	*Thai*	*0–10*	$y_8 = Yes$
$\mathbf{x}_9$	*No*	*Yes*	*Yes*	*No*	*Full*	*$*	*Yes*	*No*	*Burger*	*>60*	$y_9 = No$
$\mathbf{x}_{10}$	*Yes*	*Yes*	*Yes*	*Yes*	*Full*	*$$$*	*No*	*Yes*	*Italian*	*10–30*	$y_{10} = No$
$\mathbf{x}_{11}$	*No*	*No*	*No*	*No*	*None*	*$*	*No*	*No*	*Thai*	*0–10*	$y_{11} = No$
$\mathbf{x}_{12}$	*Yes*	*Yes*	*Yes*	*Yes*	*Full*	*$*	*No*	*No*	*Burger*	*30–60*	$y_{12} = Yes$

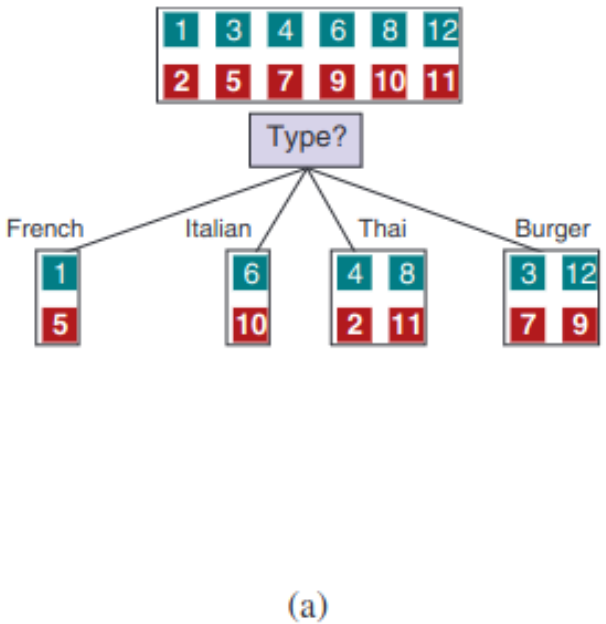
Figure 18.3 Examples for the restaurant domain.

Como gerar árvores de decisão

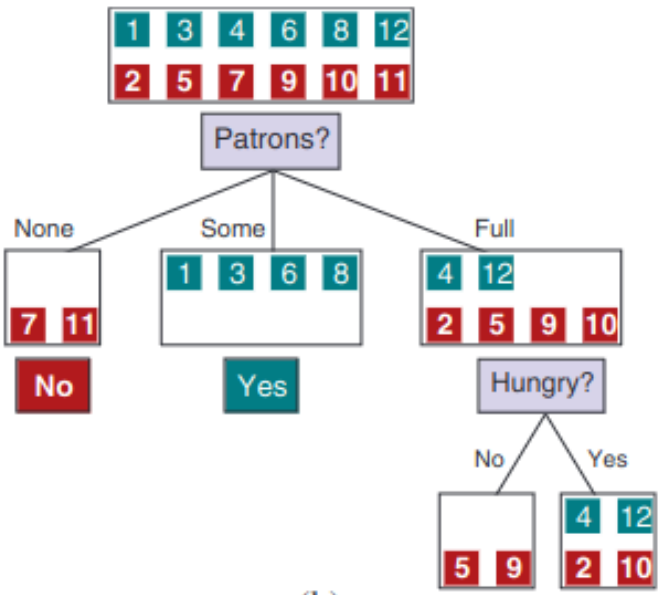
O nosso objectivo é gerar uma árvore de decisão que seja consistente com o conjunto de dados que temos e que seja o mais pequena possível.
Vamos falar de um algoritmo greedy que irá escolher sempre o atributo mais importante para estar mais perto da raíz da árvore e depois irá construir a restante árvore recursivamente.
O atributo mais importante é aquele que melhor nos ajuda a classificar os exemplos.

Como gerar árvores de decisão

Neste caso, podemos ver que o atributo mais importante dos dois é o Patrons.
Com Type, mantemos exactamente a mesma distribuição após a separação dos exemplos pelos valores de atributos.
Com Patrons, conseguimos logo dar uma resposta relativamente a bastantes exemplos, ficando apenas por resolver o caso em que o valor de Patrons é "Full".



(a)



(b)

Como gerar árvores de decisão: ID3

Após escolhermos o atributo mais importante, existem quatro casos possíveis:

- Se todos os exemplos que restam são todos positivos ou negativos, então já podemos dar uma resposta: Sim ou Não. Por exemplo, $Patrons = None$.
- Se existem alguns exemplos positivos ou negativos então voltamos a escolher o melhor atributo para os separar. Por exemplo em $Patrons = Full$ é escolhido o atributo *Hungry*.
- Se já não existem exemplos, então quer dizer que ainda não foi visto nenhum caso com aquela combinação de atributos. Assim sendo, retornamos o output mais comum do conjunto de exemplos que foi usado na construção do nó pai.
- Se já não existem atributos para usar, mas ainda temos exemplos positivos e negativos, então quer dizer que estes exemplos têm a mesma descrição, mas diferentes classificações. Neste caso, retornamos o valor de output mais comum neste conjunto de exemplos.

Algoritmo ID3

Algorithm 1 ID3(*examples*,*attributes*,*parent_examples*)

if *examples* is empty **then**
  return PLURALITY-VALUE(*parent_examples*)
end if
if all *examples* have the same classification **then**
  return the classification
end if
if *attributes* is empty **then**
  return PLURALITY-VALUE(*examples*)
end if
$A \leftarrow \text{argmax}_{a \in attributes}$ IMPORTANCE(a,*examples*)
tree $\leftarrow$ a new decision tree with root test A
for all value v of A **do**
  $exs \leftarrow \{e : e \in examples \textbf{ and } e.A = v\}$
  subtree $\leftarrow$ LEARN-DECISION-TREE(*exs*,*attributes* $- A$,*examples*)
  add a branch to *tree* with label $(A = v)$ and subtree *subtree*
end for
return *tree*

Como determinar a importância

Para determinar a importância, vamos usar a noção de ganho de informação, que deriva da definição de entropia.
Entropia é uma medida de incerteza de uma variável aleatória: quanto mais informação, menos entropia.
Se uma variável aleatória tem apenas um valor possível, a entropia é 0.
Uma moeda equilibrada com 2 faces tem 1 bit de entropia (2 resultados igualmente possíveis).
Se uma moeda está viciada e 99% das vezes que a atiramos, recebemos "cara", então a entropia será muito menor que 1, uma vez que a incerteza é também ela muito menor.

Como determinar a importância

Podemos definir entropia como:

$$H(V) = -\sum_k P(v_k) log_2 P(v_k) \tag{1}$$

onde V é uma variável aleatória com valores v_k com probabilidades $P(v_k)$
Podemos calcular a entropia de uma moeda equilibrada:

$$H(equilibrada) = -(0.5 log_2 0.5 + 0.5 log_2 0.5) = 1 \tag{2}$$

Para uma moeda viciada:

$$H(viciada) = -(0.99 log_2 0.99 + 0.01 log_2 0.01) = 0.08 \tag{3}$$

Como determinar a importância

Vamos então definir a seguinte função para ajudar na representação o cálculo do ganho de informação de uma variável:

$$B(q) = -(q log_2 q + (1-q) log_2 (1-q)) \tag{4}$$

Assim, podemos dizer que $H(viciada) = B(0.99) = 0.08$. Se tivermos um dataset com p exemplos positivos e n exemplos negativos, então a entropia da variável de output no conjunto inteiro é:

$$H(output) = B(\frac{p}{p+n}) \tag{5}$$

No exemplo do restaurante, temos que $p = 6$ e $n = 6$. Logo, a entropia pode ser calculada como $B(0.5) = 1$.

Como determinar a importância

A ideia é escolher um atributo A de tal forma que a entropia do conjunto de dados desça. Medimos esta redução calculando a entropia que resta depois de efectuado o teste ao atributo.
Um atributo A com d valores diferentes divide o conjunto de treino E em subconjuntos $E_1, \ldots, E_d$. Cada subconjunto E_k tem p_k exemplos positivos e n_k exemplos negativos, pelo que precisaremos de $B(\frac{p_k}{p_k+n_k})$ bits de informação para responder à questão.
Um exemplo escolhido aleatoriamente tem probabilidade $\frac{p_k+n_k}{p+n}$ de pertencer a E_k, pelo que a restante entropia depois de escolhido o atributo pode ser calculada da seguinte forma:

$$Resto(A) = \sum_{k=1}^{d} \frac{p_k + n_k}{p + n} B(\frac{p_k}{p_k + n_k}) \tag{6}$$

Como determinar a importância

O ganho de informação é então calculado da seguinte forma para um atributo A:

$$Ganho(A) = B(\frac{p}{p+n}) - Resto(A) \tag{7}$$

A função $Ganho(A)$ é, na verdade a função IMPORTANCE. Voltando ao exemplo do restaurante:

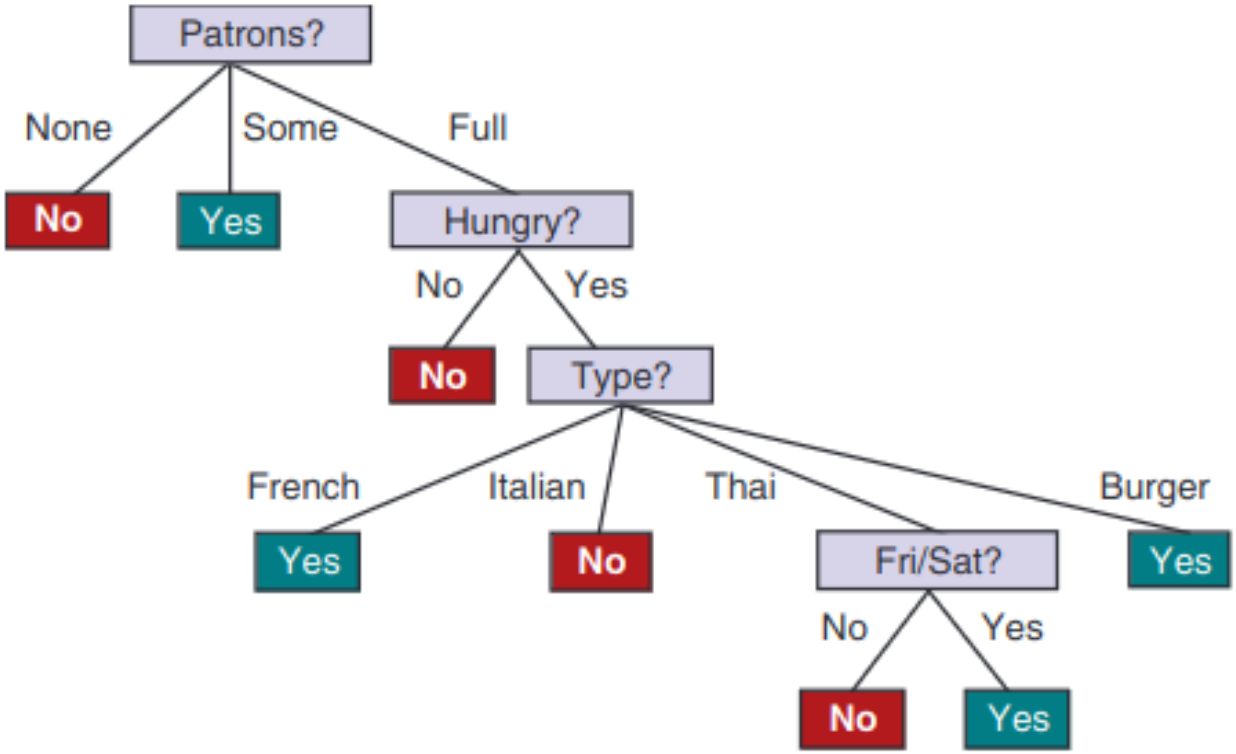
$$Ganho(Patrons) = 1 - [\frac{2}{12}B(\frac{0}{2}) + \frac{4}{12}B(\frac{4}{4}) + \frac{6}{12}B(\frac{2}{6})] = 0.541 \tag{8}$$

$$Ganho(Type) = 1 - [\frac{2}{12}B(\frac{1}{2}) + \frac{2}{12}B(\frac{1}{2}) + \frac{4}{12}B(\frac{2}{4}) + \frac{4}{12}B(\frac{2}{4})] = 0 \tag{9}$$

Confirmando a intuição de que Patrons é um atributo mais importante que Type.

Árvore de Decisão

A árvore gerada pelo algoritmo para este exemplo seria:



Regressão Linear

Vamos começar por ver o caso mais simples de uma regressão linear: regressão linear com uma função linear univariada, ou seja, vamos modelar um dados com uma linha recta.

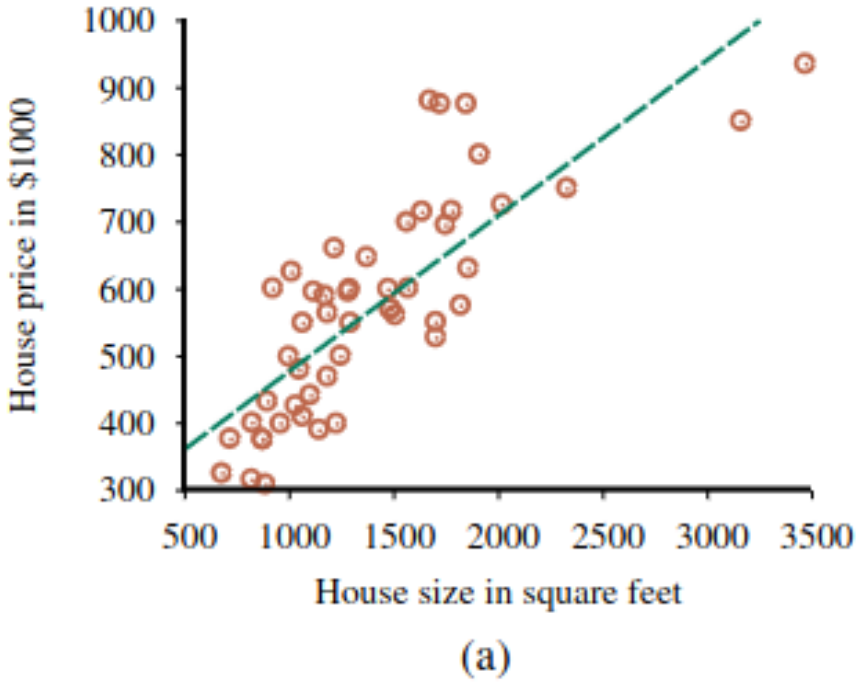
Regressão Linear

Uma função linear univariada com input x e output y tem a forma $y = w_1 x + w_0$, onde w_0 e w_1 são coeficientes que temos de determinar. Estes coeficientes funcionam como pesos: o valor de y varia consoante o peso relativo de um termo ou outro.
Vamos assumir que w é o vector $\langle w_0, w_1 \rangle$ e a função linear com esses pesos é:

$$h_w(x) = w_1 x + w_0 \tag{10}$$

Regressão Linear

Vamos explorar este exemplo onde cada ponto relaciona o tamanho e o preço de uma casa para venda.



O objectivo é encontrar a função $h_w(X)$ que melhor se ajusta aos dados. A esta tarefa chamamos regressão linear.

Regressão Linear

A tarefa resume-se a encontrar valores para os pesos $\langle w_0, w_1 \rangle$ que minimizem uma loss function.
Uma loss function clássica em casos de regressão linear é a Squared-Error:

$$Loss(h_w) = \sum_{j=1}^{N} (y_j - (w_1 x_j + w_0))^2 \tag{11}$$

O objectivo é minimizar a função $Loss(h_w)$. A função é mínima quando as suas derivadas parciais são zero:

$$\frac{\partial}{\partial w_0} \sum_{j=1}^{N} (y_j - (w_1 x_j + w_0))^2 = 0 \tag{12}$$

$$\frac{\partial}{\partial w_1} \sum_{j=1}^{N} (y_j - (w_1 x_j + w_0))^2 = 0 \tag{13}$$

Regressão Linear

Estas equações têm uma solução única:

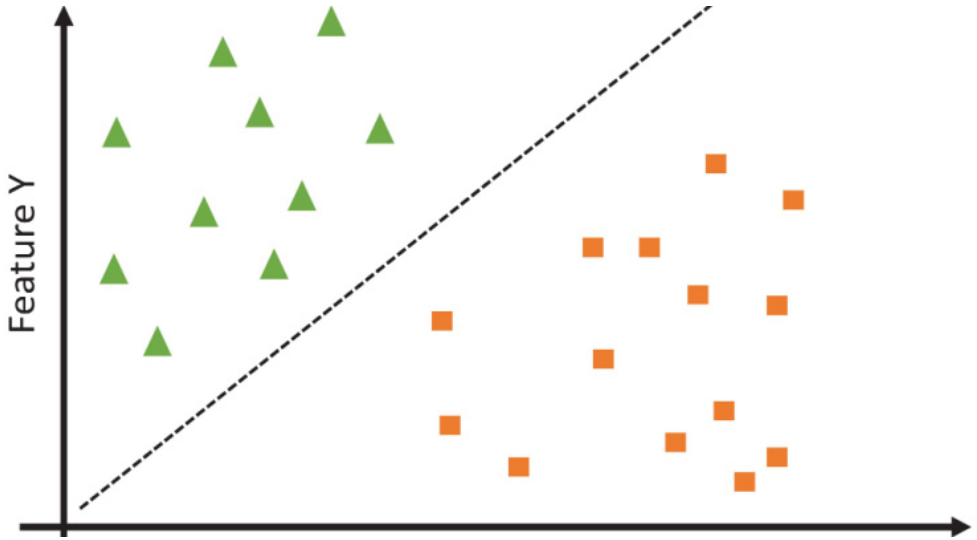
$$w_1 = \frac{N(\sum x_j y_j) - (\sum x_j)(\sum y_j)}{N(\sum (x_j)^2) - (\sum x_j)^2} \tag{14}$$

$$w_0 = (\sum y_j - w_1(\sum x_j))/N \tag{15}$$

Para o exemplo das casas, a solução seria $w_1 = 0.232$ e $w_0 = 246$ e a linha está representada a tracejado na figura.
Com as devidas extensões, é possível aplicar regressões lineares a domínios com mais variáveis.

Support Vector Machines

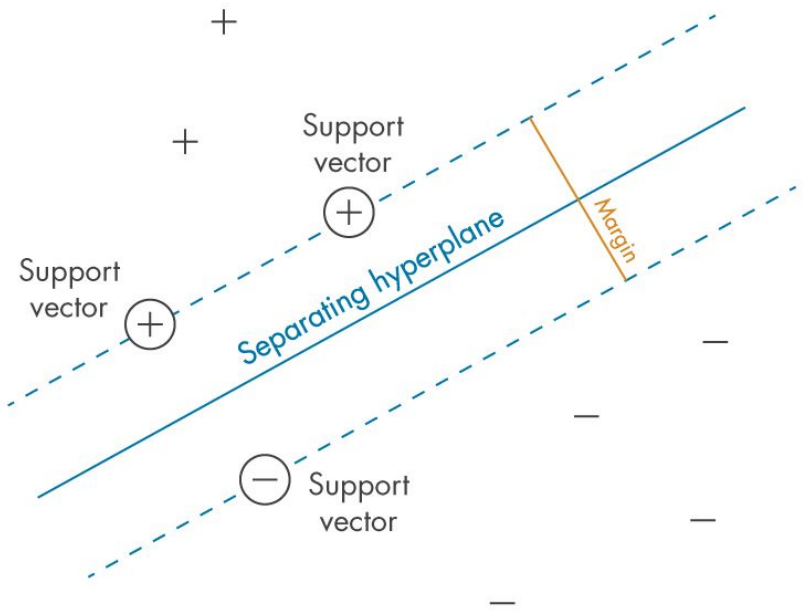
SVMs são uma classe muito flexível e poderosa de algoritmos para classificação e regressão.
O objectivo é tentar encontrar uma linha (no caso de dados bidimensionais) ou uma variedade (em dados multidimensionais) capaz de dividir as classes.



A tarefa da SVM será então encontrar uma linha que separe os dois conjuntos de dados. No entanto, temos um problema: existem formas possíveis de traçar uma linha entre os dois conjuntos de dados. Isso poderá influenciar a forma como iremos atribuir labels a dados novos.

Support Vector Machines

Para que possamos desenhar a melhor linha possível, as SVMs oferecem uma estratégia: cada linha terá associada uma margem até ao seu ponto mais próximo e a linha que oferecer a maior margem será a escolhida.




```
from sklearn.svm import SVC

model = SVC(kernel='linear',C=1E10)
model.fit(X,y)
```

Support Vector Machines: Kernels

Para que possamos utilizar todo o poder que as SVMs nos podem dar, poderemos utilizar **kernels** que são funções capazes de mapear os nossos dados para espaços com mais dimensões.
Isto irá permitir que, por exemplo, possamos dividir classes suja separação não segue um padrão linear, mapeando-as numa dimensão onde essa divisão passe a ser linear.
Podemos utilizar diferentes **kernels** no scikit (que mapearão os nossos dados para outras dimensões de formas diferentes). Para tal, basta alterar o valor do argumento kernel.

```
model = SVC(kernel='rbf',C=1E6)
model.fit(X,y)
```

Support Vector Machines: Kernels

```
model = SVC(kernel='rbf',C=1E6)
model.fit(X,y)
```

Este uso de kernels é usado em métodos lineares para que possam também funcionar em ambientes não lineares como é o caso das SVMs e até das regressões lineares.
A variável **C** controla quão rígida é a **margem**. Isto é, quão tolerante a divisão será relativamente a pontos que possam não cumprir a rigidez da separação linear das classes. Um valor de C muito grande significa que a margem será **bastante rígida**; um valor de C mais baixo significará que a margem será **mais relaxada**.

Support Vector Machines

As SVMs têm algumas vantagens:

- São modelos muito **compactos**, necessitando de **pouca memória**, uma vez que dependem de um número relativamente pequeno de vectores de suporte.
- A previsão é normalmente muito **rápida**.
- Trabalham bem com dados com **muitas dimensões**, uma vez que apenas são afectadas por pontos perto da margem.
- A utilização de kernels torna-as muito **versáteis**.

No entanto, para datasets **muito grandes**, o processo de treino pode demorar bastante tempo, os resultados também dependem bastante do valor fornecido para a **constante C** e o modelo não é facilmente **interpretável**.

Random Forests

O conceito de Random Forests utiliza um método de **ensemble** chamado **bagging**. Bagging utiliza um conjunto de estimadores que potencialmente dão overfit aos dados e faz a **média** dos resultados para conseguir uma melhor classificação. Quando fazemos bagging a um conjunto de decision trees geradas para subconjuntos aleatório do conjunto de dados, temos uma Random Forest.
Podemos gerar uma Random Forest usando o Scikit da seguinte forma:

```
from sklearn.ensemble import RandomForestClassifier

model = RandomForestClassifier(n_estimators=100)
model.fit(X,y)
```

Random Forests

Também podemos usar Random Forests para regressão, utilizando o **RandomForestRegressor**:

```
from sklearn.ensemble import RandomForestRegressor

model = RandomForestRegressor(n_estimators = 100)
model.fit(X,y)
```

Random Forests

As Random Forests têm imensas vantagens:

- Treino e previsão são relativamente **rápidos**, pela simplicidade da geração das árvores de decisão.
- As árvores permitem até **classificações probabilísticas**, podendo estimar a probabilidade de cada label de acordo com os resultados de cada árvore individualmente.
- É um modelo muito **flexível**, podendo ser útil em tarefas em que facilmente existe um underfit por parte de outros modelos.

No entanto, ao utilizar Random Forests, perdemos bastante a interpretabilidade que as Decision Trees oferecem.

Principal Component Analysis

Um dos algoritmos **não supervisionados** mais utilizados é o **PCA**.
Essencialmente, o PCA reduz a dimensionalidade do nosso conjunto de dados, podendo também ser útil em tarefas de **redução de ruído** ou **extracção de features**, por exemplo.
Enquanto que num algoritmo superivisonado nós queremos prever valores para y através dos valores de X, aqui nós queremos perceber a relação entre x e y.
Assim sendo, nós tentamos encontrar quais são os **principais atributos** dos nossos dados e utilizamos esses atributos para descrever o dataset.

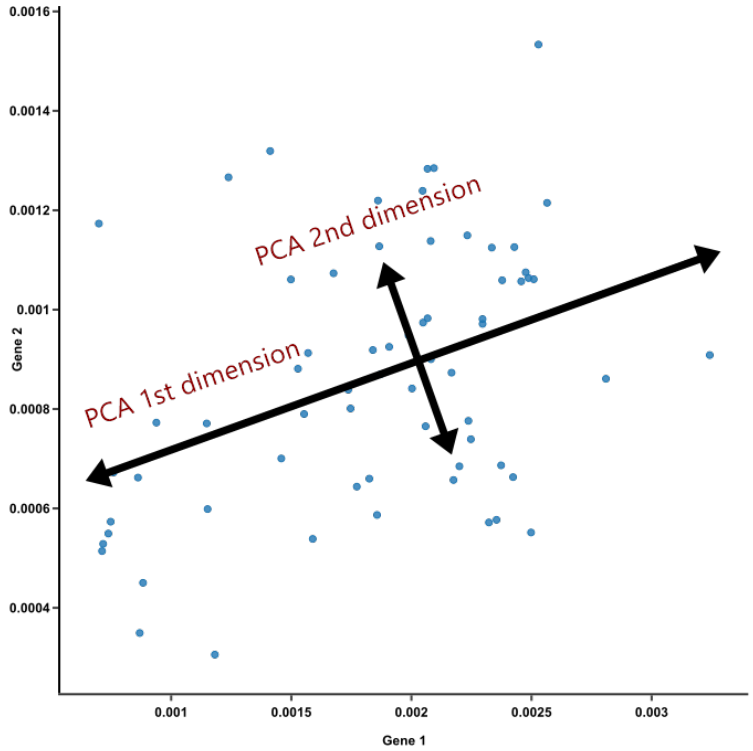
```
from sklearn.decomposition import PCA

pca = PCA(n_componentes = 2)
pca.fit(X)
```

Podemos aceder aos parâmetros mais importantes:

```
print(pca.components_)
print(pca.explained_variance_)
```

Principal Component Analysis



Os vectores obtidos são os **principais eixos** dos nosso dados e o tamanho mede a **"importância"** desse eixo na distribuição dos nossos dados, sendo uma **medida de variância** dos dados quando projectados nesse eixo.
A projecção de cada ponto nos eixos principais são os **componentes principais** dos dados e é feita através de uma **função afim**.

Principal Component Analysis

A utilização do PCA enquanto algoritmo para redução da dimensionalidade do dataset envolve descartar os componentes principais **mais pequenos**, resultando numa projecção dos dados para menos dimensões preservando a **variância maximal** dos mesmos.

```
pca = PCA(n_components=1)
pca.fit(X)
X_pca = pca.transform(X) #terá apenas uma dimensão
print(X_pca.shape) #será do tipo (L,1)
```

Embora tenhamos falado de PCA enquanto método de **redução de dimensionalidade**, ele também pode ser útil na **remoção de ruído** e para **feature selection**, por exemplo.
Uma desvantagem do PCA é que tende a ser bastante afectado por **outliers**. Podemos contrariar isto usando variantes mais robustas do PCA, como o **RandomizedPCA** ou **SparsePCA** do Scikit.

Clustering: k-means

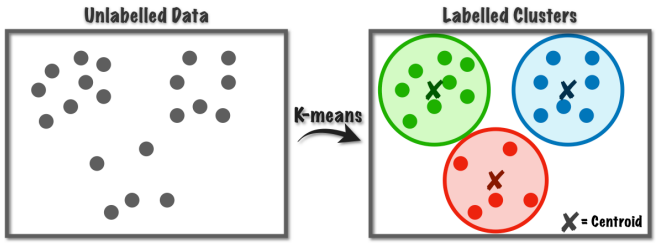
Outro tipo de algoritmos não supervisionados são os algoritmos de **clustering**.
Algoritmos de clustering tentam agrupar os pontos do nosso conjunto de dados através das propriedades dos mesmos sem recorrer a quaisquer **labels** predefinidas dos dados.
Existem imensos algoritmos de clustering disponíveis no Scikit. No entanto, um dos mais simples de perceber será o **k-means**, que está disponível na classe **sklearn.cluster.KMeans**.
O k-means procura um **número predeterminado** de clusters num conjunto de dados sem labels. Para tal, baseia-se nas seguintes noções de um **cluster óptimo**:

- O centro do cluster é a **média aritmética** de todos os pontos pertencentes ao mesmo.
- Cada ponto está **mais próximo do centro do seu cluster** do que do centro de outros clusters.

Clustering: k-means

O k-means procura um **número predeterminado** de clusters num conjunto de dados sem labels. Para tal, baseia-se nas seguintes noções de um **cluster óptimo**:

- O centro do cluster é a **média aritmética** de todos os pontos pertencentes ao mesmo.
- Cada ponto está **mais próximo do centro do seu cluster** do que do centro de outros clusters.

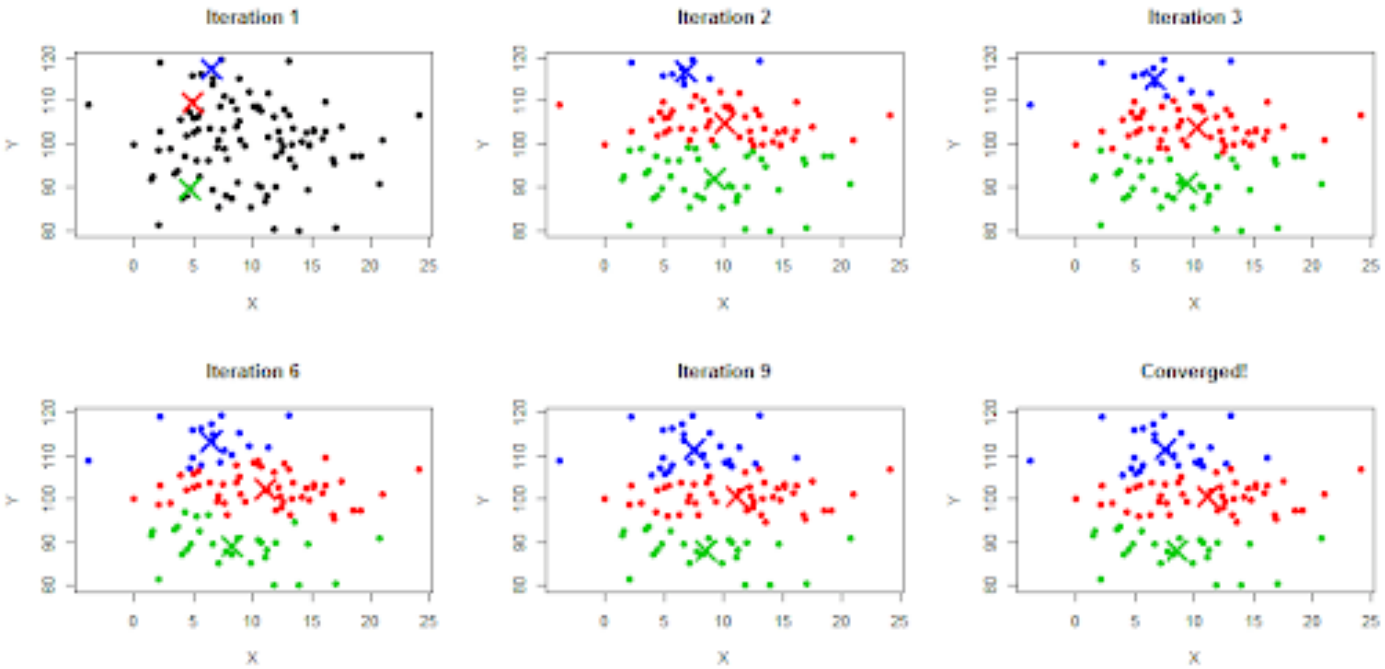



```
from sklearn.cluster import KMeans
kmeans = KMeans(n_clusters=4)
kmeans.fit(X)
preds = kmeans.predict(X)
```

Clustering: k-means

O k-means tem uma forma muito simples de funcionar:

- Inicialmente, define "aleatoriamente" alguns centros de clusters.
- De seguida, enquanto não convergir:
  - Atribui pontos ao cluster com o centro mais próximo.
  - Actualiza o centro dos clusters com a média dos seus pontos.



Clustering: k-means

Emnbora bastante intuitivo, o uso de k-means pode ter algumas limitações:

- O resultado óptimo pode não ser conseguido pois pode ficar preso em **máximos locais** e não chegar a um **máximo global**. Podemos mitigar isto permitindo ao algoritmo ter vários palpites iniciais.
- O número de clusters tem de ser definido previamente, não aprendendo o número de clusters dos dados. Outros algoritmos já aprendem isto como **DBSCAN** também disponível no Scikit.
- O k-means assume que os clusters são **separáveis linearmente**, não lidando bem com outras geometrias de clusters. O **SpectralClustering** do Scikit faz exactamente isso utilizando kernels.
- Pode ser lento para datasets muito grandes, uma vez que visita cada ponto uma vez durante cada iteração. Isto pode ser mitigado através de agoritmos k-means batch-based como o **MiniBatchKMeans** do Scikit, onde apenas um subconjunto dos dados é utilizado para actualizar os clusters.

Métricas de avaliação de classificadores

Os classificadores atribuem **labels** a cada uma das observações que lhes forem fornecidas. No entanto, precisamos de arranjar técnicas para medir **quão bem** estas atribuição estão a ser feitas.
Uma das medidas mais fáceis de entender é a **Accuracy**. Para efeitos de simplicidade vamos assumir que temos duas classes: Verdade e Falso.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{16}$$

Onde **TP** significa True Positives (valores que o modelo diz que são verdade e são de facto), **TN** significa True Negatives (valores que o modelo descreve como falso e são de facto), **FP** significa False Positives (valores que o modelo diz serem verdade, mas são falsos) e **FN** significa False Negatives (valores previstos como falsos, mas que são verdadeiros).
Um valor de 99% de **Accuracy** significa que o nosso modelo acertou em 99% dos casos que lhe foram dados a classificar.
No entanto, a Accuracy pode ser bastante enganadora, dependendo das características do conjunto de dados.

Métricas de avaliação de classificadores

Uma forma muito comum de avaliarmos os nossos classificadores é através da análise de uma **matriz de confusão**. A matriz de confusão combina os valores previstos e os valores verdadeiros das observações numa tabela.

		Predicted	
		Negative **(N)** -	Positive **(P)** +
Actual	Negative -	True Negative **(TN)**	**False Positive (FP) Type I Error**
	Positive +	**False Negative (FN) Type II Error**	True Positive **(TP)**

Através da análise desta tabela, é possível extrair várias métricas como Precision, Recall, Accuracy e AUC-ROC, entre outras.

```
from sklearn.metrics import confusion_matrix
trues = [2, 0, 2, 2, 0, 1]
preds = [0, 0, 2, 2, 0, 2]
confusion_matrix(trues,preds)
```

Precision

Calcula quantos dos casos em que o modelo acertou são positivos. É útil quando é preferível existirem Falsos Negativos a existirem Falsos Positivos.

$$Precision = \frac{TP}{TP + FP} \tag{17}$$

```
from sklearn.metrics import precision_score
trues = [2, 0, 2, 2, 0, 1]
preds = [0, 0, 2, 2, 0, 2]
precision_score(trues,preds)
```

Recall

Calcula quantos positivos fomos capaz de prever correctamente. É útil quando é preferível existirem Falsos Positivos a existirem Falsos Negativos.

$$Recall = \frac{TP}{TP + FN} \tag{18}$$

```
from sklearn.metrics import recall_score
trues = [2, 0, 2, 2, 0, 1]
preds = [0, 0, 2, 2, 0, 2]
recall_score(trues,preds)
```

F1 Score

Média harmónica de Precision e Recall. Máxima quando ambas as métricas são iguais. Bastante útil quando FP e FN são igualmente maus.

$$F1 = 2 \times \frac{Precision \times Recall}{Precision + Recall} \tag{19}$$

```
from sklearn.metrics import f1_score
trues = [2, 0, 2, 2, 0, 1]
preds = [0, 0, 2, 2, 0, 2]
f1_score(trues,preds)
```

Métricas de avaliação de regressores

Tal como acontece na classificação, é útil saber quão boas estão a ser as previsões dos nossos regressores.
Para tal, existem algumas métricas básicas e intuitivas que nos podem ajudar a perceber melhor quão boa é a performance do nosso modelo.

Métricas de avaliação de regressores

Uma das mais básicas é o **Mean Squared Error (MSE)**, também muitas vezes usada como **loss function** em alguns algoritmos de ML e representa a média do quadrado dos erros do nosso regressor:

$$MSE = \frac{1}{N}\sum_{i=1}^{n}(Y_i - TrueY_i)^2 \tag{20}$$

O facto de considerarmos o quadrado do erro, irá inflacionar erros muito grosseiros.

```
from sklearn.metrics import mean_squared_error

trues = [1,1,0,0,1,0]
preds = [0.95,0.85,0.9,0.8,0.7,0.3]
error = mean_squared_error(trues,preds)
```

Métricas de avaliação de regressores

Uma extensão desta métrica é a RMSE (Root Mean Squared Error) que corresponde à raíz quadrada do MSE. Podemos calcular esta função passando um argumento adicional à MSE do Scikit:

```
error = mean_squared_error(trues,preds,squared=False)
```

Métricas de avaliação de regressores

Outra métrica bastante comum é **Mean Absolute Error (MAE)** que ao contrário das suas métricas anteriores não penaliza com magnitudes diferentes erros de ordem diferentes (pois não faz o quadrado dos erros).

$$MAE = \frac{1}{N}\sum_{i=1}^{n} |Y_i - TrueY_i| \tag{21}$$

```
from sklearn.metrics import mean_absolute_error

trues = [1,1,0,0,1,0]
preds = [0.95,0.85,0.9,0.8,0.7,0.3]
error = mean_absolute_error(trues,preds)
```