## 5. Árvores de Decisão e Seleção de Atributos

Nesta aula você vai aprender:

* Sobre Classificador de Árvores de Decisão
* O que é Ganho de Informação ou Informação Mútua
* Como empregar esses e outros conceitos na Seleção de Atributos para o Aprendizado

Nas aulas anteriores você já aprendeu sobre os classificadores logístico e k-vizinhos mais próximos e como empregá-los. Mas existem muitos outros classificadores que empregam outros princípios para chegar a um resultado de classificação dos dados.

from sklearn.neural\_network import MLPClassifier  
from sklearn.neighbors import KNeighborsClassifier  
from sklearn.svm import SVC  
from sklearn.gaussian\_process import GaussianProcessClassifier  
from sklearn.gaussian\_process.kernels import RBF  
from sklearn.tree import DecisionTreeClassifier  
from sklearn.ensemble import RandomForestClassifier, AdaBoostClassifier  
from sklearn.naive\_bayes import GaussianNB  
from sklearn.discriminant\_analysis import QuadraticDiscriminantAnalysis

O código acima exibe as funções de apenas alguns desses classificadores que você encontra no scikit-learn (acesse <https://scikit-learn.org/stable/auto_examples/classification/plot_classifier_comparison.html>).

As Árvores de Decisão são um modelo bastante importante e poderoso e é o modelo que vamos estudar nesta aula. Entre as diferenças das Árvores de Decisão com relação aos modelos anteriores está no fato de ser um modelo cujo princípio está em empregar a probabilidade (ou proporção) dos valores dos atributos no lugar dos próprios valores (pense em um atributo gênero, com valores Masculino e Feminino, a proporção dos valores é a probabilidade de você encontrar ao acaso o valor Masculino ou Feminino nos dados). Uma grande vantagem disso é que, não empregando os valores diretamente, mas apenas as suas proporções ou probabilidades, podemos empregar atributos categóricos diretamente sem a preocupação de transformá-los em numéricos (o modelo NaiveBayes é outro modelo que emprega o mesmo princípio). Isso também permitirá introduzirmos o conceito de ganho de informação, importante princípio que pode ser aplicado à seleção de atributos (a decisão de que atributos empregar no aprendizado).

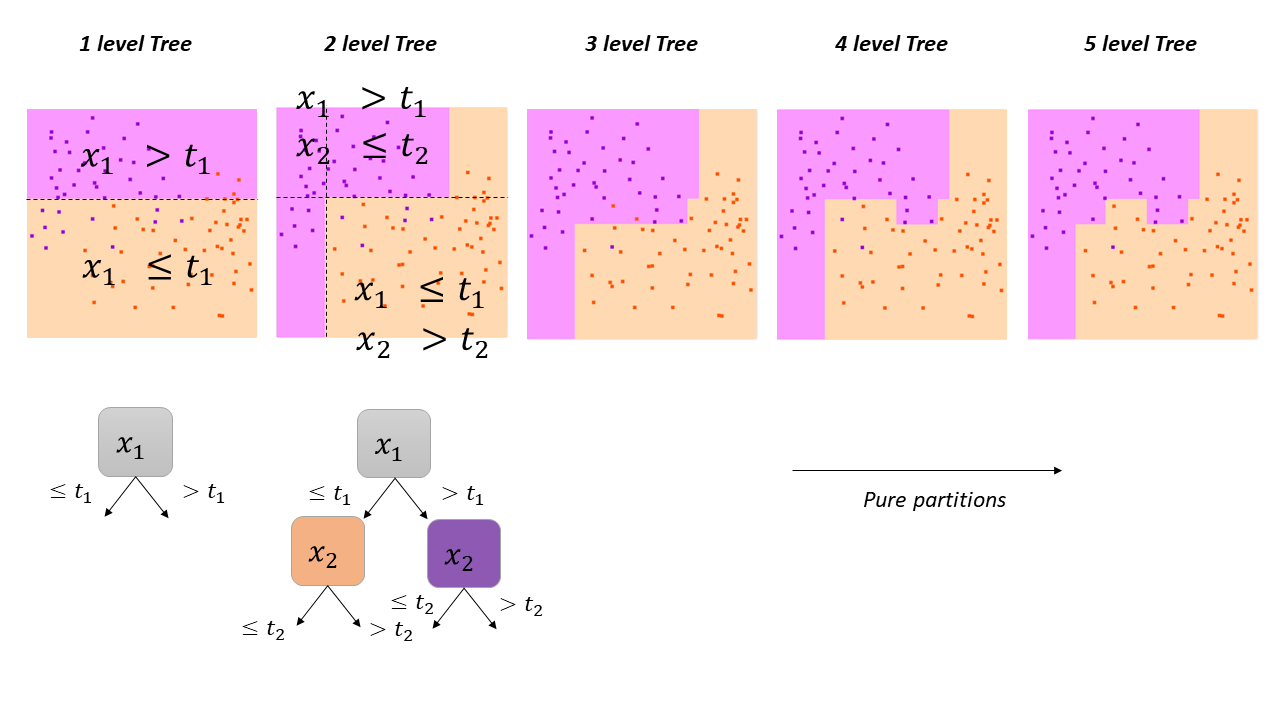
import numpy as np  
import pandas as pd  
import matplotlib.pyplot as plt  
%matplotlib inline  
from matplotlib.lines import Line2D  
import seaborn as sns

# Árvores de Decisão e o Método Partitivo

O classificador de k-vizinhos mais próximos é bastante poderoso. Ele pode lidar de várias classes de problemas e fazer classificações com limites de decisão não lineares (diferentemente da regressão logística). O Knn, entretanto, precisa armazenar todos os dados de treinamento e, quanto mais dados de treinamento, maior o custo do processamento e mais lento ele se torna pois é preciso calcular a distância para todas as entradas de treinamento, mesmo existindo estruturas de dados eficientes para pesquisa de vizinhos mais próximos. O uso de uma métrica de distância e a necessidade de atributos numéricos (ou transformados em numéricos) pode, em alguns casos, ser também uma limitação.

Você deve lembrar que os algoritmos de aprendizado de máquina buscam por padrões nos dados, uma estrutura inerente que é inerente a eles. O Knn, por exemplo, parte da ideia de que semelhança entre itens vizinhos e que pontos de dados de várias classes não são espalhados aleatoriamente. Pelo contrário, as classes aparecem em grupos mais ou menos homogêneos. Assim se você sabe que um ponto de teste cai em um grupo com vários pontos de uma mesma classe, você sabe a classe de seus vizinhos mesmo sem calcular as distâncias para cada deles. É, neste caso, suficiente saber que o ponto de teste está em uma área onde todos os vizinhos são de uma classe. As árvores de decisão exploram exatamente isso.

As árvores de decisão estão explorando exatamente isso. Aqui, não são armazenados os dados de treinamento mas uma estrutura de árvore *'modela'* os dados e divide recursivamente o espaço em regiões com classes semelhantes. Essa estrutura faz particionamentos sucessivos dos dados em que as partições são cada vez mais *puras*, isto é, são cada vez mais homogêneas no sentido das classes que contêm.



**Figura 1. Particionamentos Sucessivos das Árvores de Decisão.**

O nó raiz da árvore representa todo o conjunto de dados. Este conjunto é então dividido aproximadamente ao meio ao longo de uma dimensão (um atributo) por um limite simples. Todos os pontos que possuem um valor caem no nó filho direito e todos os demais no nó filho esquerdo. O limiar a dimensão é escolhida de forma que os nós filhos resultantes sejam mais puros em termos de associação de classe. O ideal é que todos os pontos positivos (por exemplo, pontos laranja na figura acima) caiam em um nó filho e todos os pontos negativos (roxos) no outro. Se for esse o caso, a árvore está pronta. Caso contrário, os nós folha são novamente divididos até que eventualmente todas as folhas, ou nós terminais, sejam puras com todos seus pontos com o mesmo rótulo ou não podem ser mais divididos (dois pontos idênticos com rótulos diferentes, o que em casos reais podemos de fato ter).

Uma vez que a árvore é construída os dados de treinamento não precisam ser mais armazenados (como no Knn) pois a árvore *captura* todo o padrão dos dados, e você pode pensar em como uma única fórmula y = a0 + a1 x modela um conjunto de dados em uma regressão linear. A árvore passa a ser nossa '*fórmula*' de predição, as entradas de teste simplesmente precisam descer da árvore até uma folha que contendo a predição da classe, e são muito eficientes. Além disso as árvores de decisão não requerem métricas porque as divisões são baseadas em limites dos valores dos atributos (ou ainda suas proporções ou probabilidades) e não em distâncias.

# Construindo uma Árvore de Decisão: Algoritmo de Hunt

Vamos começar com um exemplo e entender como construir uma árvore de decisão a partir dos dados.

# Exemplo livremente adaptado de:   
# Weinberger, Kilian. \*\*Machine Learning for Intelligent Systems\*\*, Lecture 28: Decision / Regression Trees   
#  
  
df = pd.read\_csv('http://meusite.mackenzie.br/rogerio/TIC/comics.csv',sep=';')  
display(df)  
  
cases = pd.read\_csv('http://meusite.mackenzie.br/rogerio/TIC/comics\_cases.csv',sep=';')  
display(cases)

Name sex smokes tie mask cape ears class  
0 Batman male no no yes yes yes good  
1 Robin male no no yes yes yes good  
2 Catwoman female no no yes no yes bad  
3 Joker male no no no no no bad  
4 Alfred male no yes no no no good  
5 Penguin male yes yes no no no bad

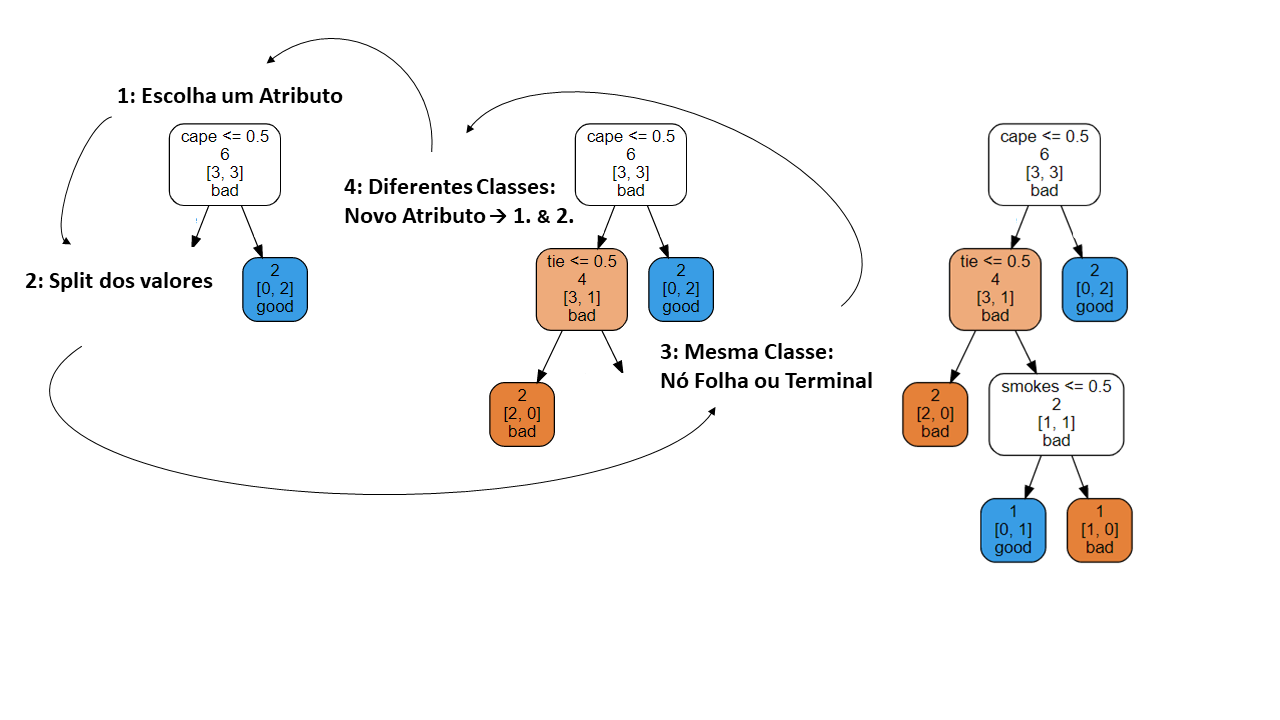
Name sex smokes tie mask cape ears class  
0 Batgirl female yes yes no yes no ?  
1 Riddler male yes no no no no ?

O algoritmo de Hunt cria uma árvore de decisão de maneira recursiva, particionando os registros de treinamento em subconjuntos sucessivamente mais puros. Seja o conjunto de registros de treinamento que atingem um nó . O procedimento recursivo é, então, o seguinte:

1. **Se contém registros que pertencem à mesma classe , então é um nó folha rotulado como**
2. **Se for um conjunto vazio, então é um nó folha rotulado pela classe padrão,**
3. **Se contiver registros que pertencem a mais de uma classe, use um teste de atributo para dividir os dados em subconjuntos menores.**

Ele aplica recursivamente o procedimento a cada subconjunto até que todos os registros no subconjunto pertençam à mesma classe. O algoritmo de Hunt assume que cada combinação de conjuntos de atributos possui um rótulo de classe exclusivo durante o procedimento. Se todos os registros associados a tiverem valores de atributo idênticos, exceto para o rótulo da classe, não será possível dividir esses registros no futuro. Nesse caso, o nó é classificado como um nó folha com o mesmo rótulo de classe que a classe principal de registros de treinamento associados a este nó.

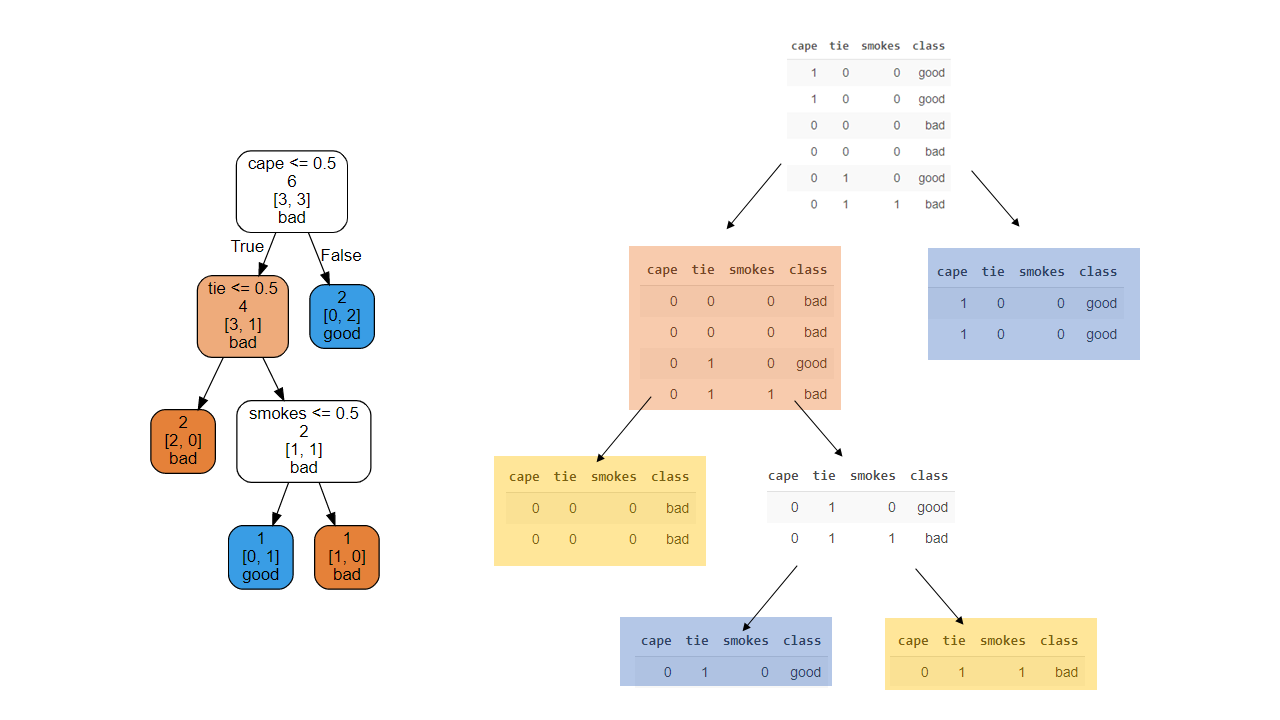
O procedimento talvez seja mais claro de *ver* do que *ler*...



**Figura 2. Construção de uma Árvore de Decisão pelo algoritmo de Hunt.**

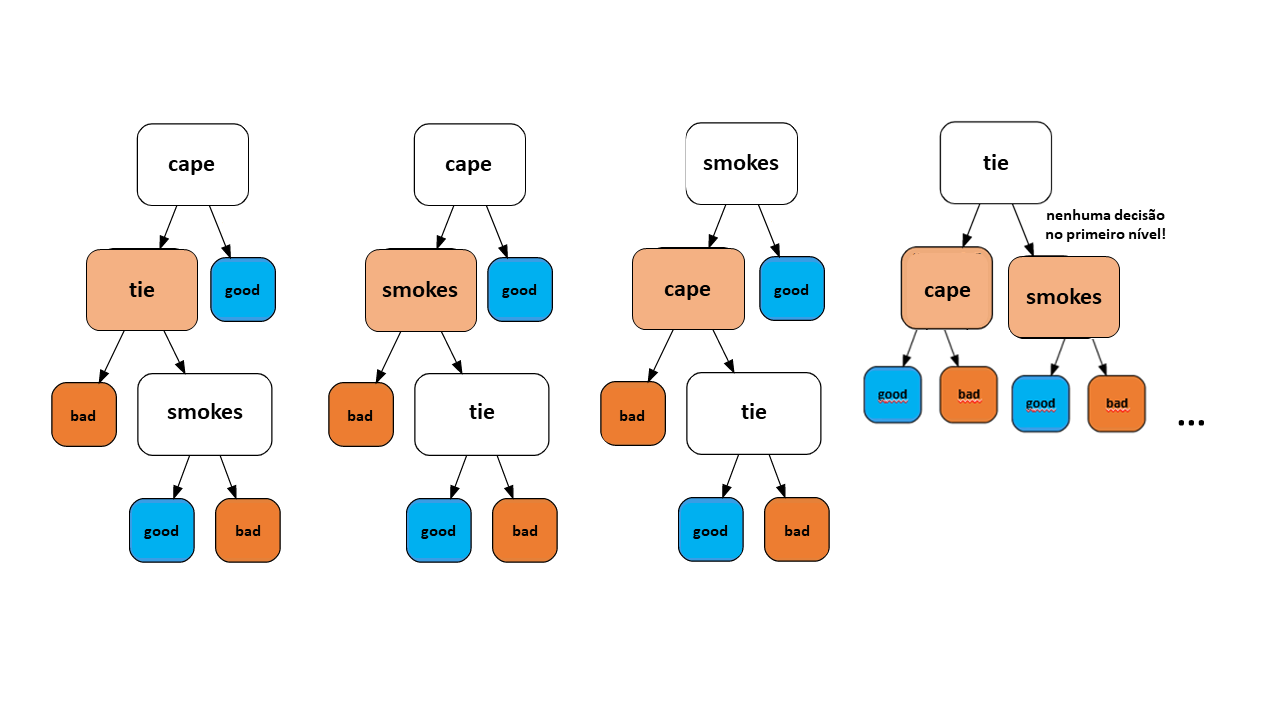
No exemplo acima escolhemos arbitrariamente iniciar o particionamento dos dados pelo atributo cape, como para todos os casos cape = yes temos somente alunos good esse ramo se torna um nó folha ou terminal, e dados que seguem nesse ramo já estão classificados e não serão mais considerados. Para os casos cape = no há alunos good e bad e, portanto, não podemos decidir e é necessário adicionarmos um novo nó com outro atributo. Note que nesse ramo só teremos agora dados com cape = no. Escolhemos novamente de modo arbitrário o atributo tie de realizamos novamente o *split* dos dados. Agora, os casos com tie = no (e cape = no!, pois é o ramo que estamos) são todos da mesma classe e são, portanto, um nó terminal. Para tie = yes (e ainda cape = no!) precisamos adicionar ainda um novo particionamento com o atributo smokes.

Note que o que a estrutura de árvore faz é simplesmente definir sucessivos particionamentos dos dados buscando partições cada vez mais *puras* (somente classes *good* ou *bad*).



**Figura 3. Árvore de Decisão e o particionamento dos dados.**

No exemplo acima fizemos escolhas arbitrárias dos atributos para a construção da árvore. Se escolhêssemos outros atributos ou em uma ordem diferentes chegaríamos a outras árvores de decisão que poderíamos igualmente empregar para a classificação dos dados.



**Figura 4. Diferentes possíveis Árvores de Decisão**

**para um mesmo conjunto de dados.**

Mas, como você pode ver, existem árvores melhores e piores. O atributo tie na raiz, por exemplo, não permite particionarmos dados de forma a eliminar quaisquer dos dados e, portanto, *ele não fornece qualquer informação útil para determinar a classe dos dados*. A ideia é de árvores mais eficientes possam *otimizar* a cada nível a *pureza* das partições. Você entenderá como podemos fazer isso mais adiante em que empregaremos o *ganho de informação* como uma medida para definir melhores partições dos dados a cada nível.

# Outros Algoritmos

Os algoritmos de Árvore de Decisão constroem as Árvores do mesmo modo recursivo que construímos aqui. Todos eles buscam particionamentos que maximizem a decisão, tornando a Árvores e os caminhos que temos que percorrer menores. Eles buscam a cada nível da Árvore (partição) uma maior informação para “pureza” das partições dos dados em termos de sua classe.

**Diferentes algoritmos de Árvores de Decisão e seus critérios para seleção dos nós:**

* **ID3 algorithm (Claude Shannon Entropy): Ganho do de Informação**
* **C4.5 algorithm: Gain ratio**
* **CART algorithm: Gini index**

Seja qual for o método empregado a ideia é a mesma. Existem particionamentos que alteram pouco a distribuição dos dados entre as classes e trazem, portanto, pouca informação adicional para a classificação, e outros que proporcionam informação maior para a decisão.

O pseudo código abaixo é o algoritmo clássico apresentado por Mitchell, Tom Michael (1997) em seu clássico livro de Aprendizado de Máquina.

%%script false  
# Adaptado de, Mitchell, Tom Michael (1997). Machine Learning.   
  
def ID3 (Examples, Target\_Attribute, Attributes):   
   
 Create a root node for the tree  
 if all examples are positive, return the single-node tree Root, with label = +.  
 if all examples are negative, return the single-node tree Root, with label = -.  
   
 if number of predicting attributes is empty:  
 return the single node tree Root, with label = most common value of the target attribute in the examples.  
 else:  
 A ← The Attribute that best classifies examples.  
 Decision Tree attribute for Root = A.  
 for each possible value, vi, of A,  
 Add a new tree branch below Root, corresponding to the test A = vi.  
 Let Examples(vi) be the subset of examples that have the value vi for A  
 if Examples(vi) is empty  
 below this new branch add a leaf node with label = most common target value in the examples  
 else below this new branch add the subtree ID3 (Examples(vi), Target\_Attribute, Attributes – {A})  
 End

# Decision Tree, scikit-learn

Antes de prosseguirmos vamos explorar o classificador de árvore de decisão do scikit-learn no nosso exemplo de brinquedo.

Os modelos de árvore de decisão podem lidar com variáveis categóricas sem codificá-las de uma só vez. Veja que nosso exemplo os valores atributos tinham valores 'yes' ou 'no' e, em nenhum momento precisamos supor ou transformá-los em numéricos. Entretanto as implementações mais populares de árvores de decisão, incluindo o scikit-learn não honram esse fato e requerem o encode dos dados categóricos, apesar da degradação do desempenho do modelo de árvore que isso traz. Desse modo precisamos proceder o encode dos dados para empregarmos o classificador de árvore de decisão do scikit-learn. Mas lembre-se, que isso é uma limitação apenas da implementação.

from sklearn.preprocessing import OneHotEncoder, LabelEncoder  
  
df\_labels = df.drop(columns=['Name','class']).apply(LabelEncoder().fit\_transform)  
cases\_labels = cases.drop(columns=['Name','class']).apply(LabelEncoder().fit\_transform)  
  
display( pd.concat([df.Name, df\_labels, df['class']], axis=1) )  
display( pd.concat([cases.Name, cases\_labels, cases['class']], axis=1) )

Name sex smokes tie mask cape ears class  
0 Batman 1 0 0 1 1 1 good  
1 Robin 1 0 0 1 1 1 good  
2 Catwoman 0 0 0 1 0 1 bad  
3 Joker 1 0 0 0 0 0 bad  
4 Alfred 1 0 1 0 0 0 good  
5 Penguin 1 1 1 0 0 0 bad

Name sex smokes tie mask cape ears class  
0 Batgirl 0 0 1 0 1 0 ?  
1 Riddler 1 0 0 0 0 0 ?

Neste exemplo simples, havendo para todos os atributos somente dois valores, podemos fazer o LabelEncoder dos dados, tendo o mesmo resultado que teríamos ao empregar o OneHotEncoder(drop='if-binary').

Como estamos interessados apenas em *explorar* o modelo, por hora podemos deixar de lado a separação de dados de treinamento e teste. O estimador DecisionTreeClassifier() segue então a mesma forma de uso que empregamos até agora.

from sklearn.tree import DecisionTreeClassifier  
  
X = df\_labels  
y = df['class']  
  
clf = DecisionTreeClassifier(criterion='entropy')   
  
clf.fit(X,y)  
y\_pred = clf.predict(X)  
print( clf.score(X,y) )

1.0

E podemos então fazer predições a partir do modelo construído.

A picture containing graphical user interface

Description automatically generated

Diagram

Description automatically generated

**Figura 5. Personagens para predição**.

(Fonte: imagens de divulgação, <https://universoheroico.com.br> e <https://images6.fanpop.com>)

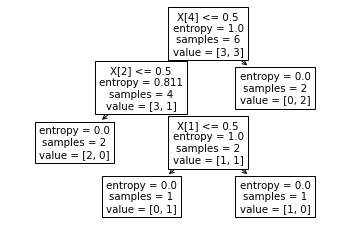
y\_pred = clf.predict(cases\_labels)  
cases['class'] = y\_pred  
display( cases )

Name sex smokes tie mask cape ears class  
0 Batgirl female yes yes no yes no good  
1 Riddler male yes no no no no bad

Uma das grandes vantagens dos modelos de Árvore de Decisão é sua *interpretabilidade*. Um tema cada vez mais exigido dos modelos de aprendizado de máquina à medida que eles são mais e mais empregados em decisões que influenciam o dia a dia e o destino das pessoas (por exemplo, aplicações na no processo seletivo das empresas ou a previsibilidade de crimes).

Essa interpretabilidade é em grande parte pela possibilidade de podermos *enxergar* como é tomada a decisão do classificador observando a estrutura da árvore ou, ao menos, a cadeia de particionamentos criada. O scikit-learn fornece algumas formas de visualização da árvore criada sendo a mais simples, o plot\_tree().

from sklearn import tree  
  
tree.plot\_tree(clf)  
plt.show()



Mas resultados muito melhores podem ser obtidos com o pacote graphviz.

import graphviz   
dot\_data = tree.export\_graphviz(clf, out\_file=None,   
 feature\_names=list(df\_labels.columns.values),   
 class\_names=list(sorted(df['class'].unique())),   
 filled=True, rounded=True,   
 special\_characters=False,  
 proportion=False, impurity=False, node\_ids=False,label=None)   
graph = graphviz.Source(dot\_data)   
graph.render('graph\_cartoons') # para gravação em .pdf  
graph

A picture containing text, clock

Description automatically generated

Revisando a árvore acima podemos reproduzir os particionamentos criados e notar que o 'aprendizado' do modelo pode ser facilmente interpretado por uma série de condições (*ifs*) encadeados.

df\_tree = pd.concat([df\_labels, df['class']], axis=1)  
  
print('Root level')  
print(60\*'-')  
display(df\_tree)  
print('Continue Next Level...')  
display(df\_tree[df\_tree.cape<=0.5])  
print('Terminal')  
display(df\_tree[df\_tree.cape>0.5])  
  
print('\n\n\nSecond level')  
print(60\*'-')  
display(df\_tree[df\_tree.cape<=0.5])  
print('Continue Next Level...')  
display(df\_tree[(df\_tree.cape<=0.5) & (df\_tree.tie>0.5)])  
print('Terminal')  
display(df\_tree[(df\_tree.cape<=0.5) & (df\_tree.tie<=0.5)])  
  
print('\n\n\nThird level')  
print(60\*'-')  
display(df\_tree[(df\_tree.cape<=0.5) & (df\_tree.tie>0.5)])  
print('Terminal')  
display(df\_tree[(df\_tree.cape<=0.5) & (df\_tree.tie>0.5) & (df\_tree.smokes<=0.5) ])  
print('Terminal')  
display(df\_tree[(df\_tree.cape<=0.5) & (df\_tree.tie>0.5) & (df\_tree.smokes>0.5)])

Root level  
------------------------------------------------------------

sex smokes tie mask cape ears class  
0 1 0 0 1 1 1 good  
1 1 0 0 1 1 1 good  
2 0 0 0 1 0 1 bad  
3 1 0 0 0 0 0 bad  
4 1 0 1 0 0 0 good  
5 1 1 1 0 0 0 bad

Continue Next Level...

sex smokes tie mask cape ears class  
2 0 0 0 1 0 1 bad  
3 1 0 0 0 0 0 bad  
4 1 0 1 0 0 0 good  
5 1 1 1 0 0 0 bad

Terminal

sex smokes tie mask cape ears class  
0 1 0 0 1 1 1 good  
1 1 0 0 1 1 1 good

Second level  
------------------------------------------------------------

sex smokes tie mask cape ears class  
2 0 0 0 1 0 1 bad  
3 1 0 0 0 0 0 bad  
4 1 0 1 0 0 0 good  
5 1 1 1 0 0 0 bad

Continue Next Level...

sex smokes tie mask cape ears class  
4 1 0 1 0 0 0 good  
5 1 1 1 0 0 0 bad

Terminal

sex smokes tie mask cape ears class  
2 0 0 0 1 0 1 bad  
3 1 0 0 0 0 0 bad

Third level  
------------------------------------------------------------

sex smokes tie mask cape ears class  
4 1 0 1 0 0 0 good  
5 1 1 1 0 0 0 bad

Terminal

sex smokes tie mask cape ears class  
4 1 0 1 0 0 0 good

Terminal

sex smokes tie mask cape ears class  
5 1 1 1 0 0 0 bad

# Um Exemplo Clássico

Antes de prosseguir talvez você queira revisar aqui alguns pontos de como empregar o classificador DecisionTreeClassifier no exemplo clássico de Mitchell, Tom Michael (1997). Machine Learning. Como no exemplo anterior não faremos a separação do conjunto de dados de treinamento e teste, pois nosso foco aqui é na aplicação do estimador.

df\_playball = pd.read\_csv('http://meusite.mackenzie.br/rogerio/ML/PlayBallcsv.csv')  
df\_playball.head()

Day Outlook Temperature Humidity Wind Play ball  
0 D1 Sunny Hot High Weak No  
1 D2 Sunny Hot High Strong No  
2 D3 Overcast Hot High Weak Yes  
3 D4 Rain Mild High Weak Yes  
4 D5 Rain Cool Normal Weak Yes

Primeiramente você precisa fazer o encode dos dados, uma restrição da implementação do scikit-learn.

***Note: Árvores de Decisão, como você observou através do exemplo de sua construção não requerem valores numéricos e podem empregar valores categóricos diretamente. Entretanto, o estimador do scikit-learn não está implementado deste modo e requer que os valores dos atributos sejam numéricos. Por isso, é necessário fazer o encode dos dados antes de aplicar o estimador.***

from sklearn.preprocessing import OneHotEncoder, LabelEncoder  
  
df\_labels\_playball = df\_playball.drop(columns=['Day','Play ball']).apply(LabelEncoder().fit\_transform)  
  
pd.concat([ df\_labels, df\_playball[['Play ball']] ], axis=1).head()

sex smokes tie mask cape ears Play ball  
0 1.0 0.0 0.0 1.0 1.0 1.0 No  
1 1.0 0.0 0.0 1.0 1.0 1.0 No  
2 0.0 0.0 0.0 1.0 0.0 1.0 Yes  
3 1.0 0.0 0.0 0.0 0.0 0.0 Yes  
4 1.0 0.0 1.0 0.0 0.0 0.0 Yes

O uso do estimador segue do mesmo modo que já empregamos para o estimador logístico, o Knn e é a forma de empregarmos qualquer modelo de aprendizado supervisionado.

from sklearn.tree import DecisionTreeClassifier  
  
X = df\_labels\_playball  
y = df\_playball['Play ball']  
  
clf = DecisionTreeClassifier()   
  
clf.fit(X,y)  
y\_pred = clf.predict(X)  
print( clf.score(X,y) )

1.0

Você pode achar útil visualizar a árvore de decisão criada com o graphviz.

import graphviz   
dot\_data = tree.export\_graphviz(clf, out\_file=None,   
 feature\_names=list(df\_labels\_playball.columns.values),   
 class\_names=list(sorted(df\_playball['Play ball'].unique())),   
 filled=True, rounded=True,   
 special\_characters=False,  
 proportion=False, impurity=False, node\_ids=False,label=None)   
graph = graphviz.Source(dot\_data)   
graph.render('graph\_cartoons') # para gravação em .pdf  
graph

# Diagram Description automatically generated

# Entropia e Ganho de Informação

Voltando ao nosso exemplo dos personagens, a árvore construída foi a mesma que construímos antes (lembre-se, poderíamos ter outras árvores construídas) mas a escolha não foi aqui arbitrária. A escolha dos atributos para a definição dos nós da árvore seguiu o critério empregar os atributos com maior *ganho de informação* para maximizar a *pureza* dos particionamentos.

Você pode entender o *ganho de informação* como uma medida de quanto a informação de um atributo contribui para diminuir a incerteza sobre outro, assim, o ganho de informação de cape para determinar class, , deve ser maior que o ganho de informação dos demais atributos.

Mas para entendermos o ganho de informação precisamos antes entender o conceito de entropia.

## Entropia

Na teoria da informação, a entropia é uma medida de quantidade de informação introduzido por Claude Shannon em seu artigo de 1948 *A Mathematical Theory of Communication*. A entropia de um atributo é o nível médio de informação, surpresa, ou ainda incerteza, inerente aos resultados possíveis desse atributo. Ela é definida como:

onde é a probabilidade dos valores de .

Um *bit*, por exemplo, é a unidade de informação exatamente por que carrega *1* de entropia:

onde e são respectivamente as probabilidades do bit e .

Assim, podemos ver também que a entropia de uma constante é zero, pois não há qualquer incerteza sobre seu valor:

e é exatamente isso que queremos buscar em um particionamento ao construirmos uma árvore de decisão.

Podemos empregar o pacote scipy para determinar a entropia de dados com base nas probabilidades dos diferentes valores e, abaixo, você pode ver a entropia do *bit* e de uma constante.

from scipy.stats import entropy  
E = entropy( [1/2, 1/2] , base=2 )  
E

1.0

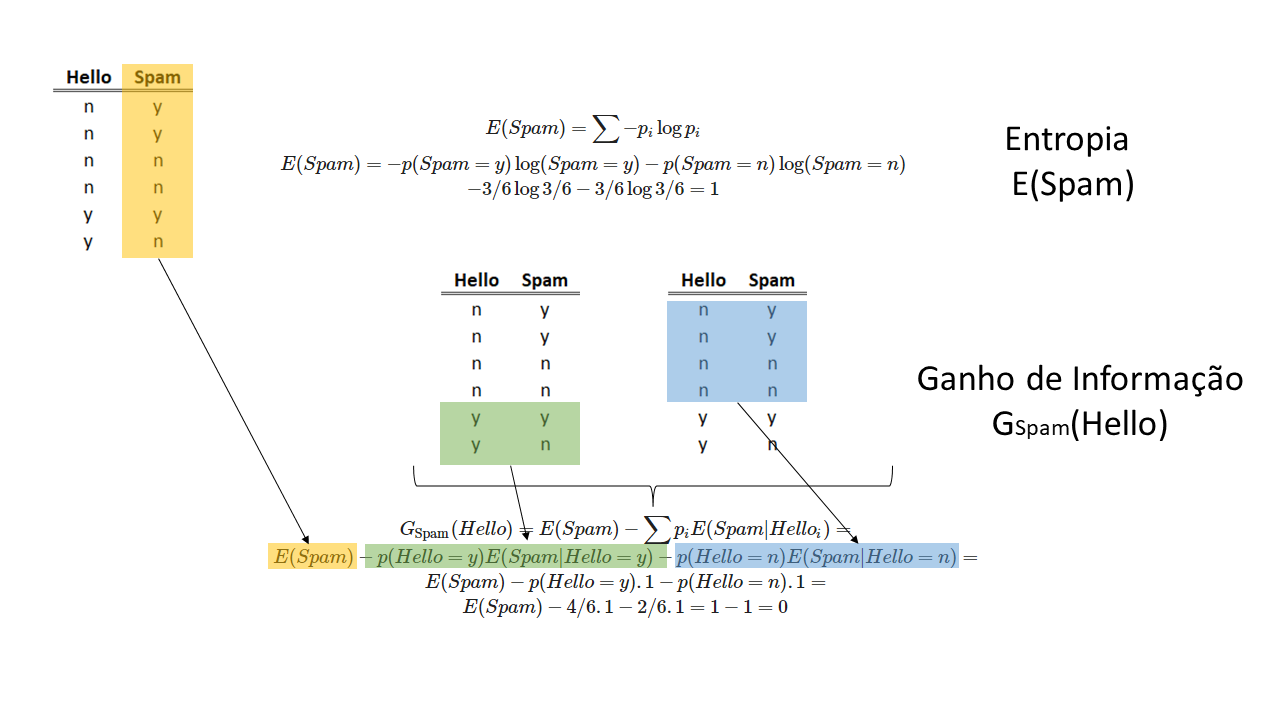
E = entropy( 1 , base=2 )  
E

0.0

## Ganho de Informação

O ganho de informação de um atributo com relação a uma variável classe objetivo (, Target) é, portanto, uma medida de quanto a informação desse atributo diminui a incerteza sobre a classe dos dados, ou seu valor no caso de uma regressão. O ganho de informação pode ser então expresso em termos da entropia como:

Onde é a entropia do atributo objetivo e a somatória é a média ponderada da entropia do atributo target para os diferentes conjuntos de dados com valor .



**Figura 6. Exemplo do Cálculo de Entropia e Ganho de Informação.**

O exemplo acima mostra o cálculo do ganho de informação do atributo *Hello*, indicando emails iniciando ou não por *Hello*, para a determinação da classe desses emails, *Spam* ou *Not Spam*.

# Ganho de Informação, Índice Gini e Classification Error

Você deve ter notado que ao empregarmos o classificador DecisionTreeClassifier() empregamos um parâmetro criterion, DecisionTreeClassifier(criterion='entropy'). Isso porque existem outras métricas de cálculo do ganho de informação como o Índice Gini e o Classification Error sendo o padrão do scikit-learn o emprego do Índice Gini, DecisionTreeClassifier(criterion='gini). Essas medidas são definidas como:

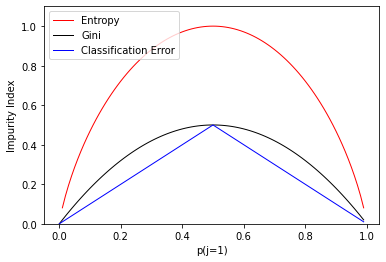
e

E são empregadas dos mesmo modo que a entropia para o cálculo do ganho de informação:

e

Tanto a entropia como o índice gini e o classification error são igualmente medidas de impuridade (0, para nenhuma incerteza dos dados) sendo apenas funções com diferentes medidas mas que preservam a mesma ordem de medida, isto é, um maior valor de entropia sempre leva a maiores valores do índice gini e do classification error, como você pode observar abaixo.

#@markdown you can skip this code!  
  
# Adaptado de: https://www.bogotobogo.com/python/scikit-learn/scikt\_machine\_learning\_Decision\_Tree\_Learning\_Informatioin\_Gain\_IG\_Impurity\_Entropy\_Gini\_Classification\_Error.php  
import matplotlib.pyplot as plt  
import numpy as np  
  
def gini(p):  
 return (p)\*(1 - (p)) + (1 - p)\*(1 - (1-p))  
  
def entropy(p):  
 return - p\*np.log2(p) - (1 - p)\*np.log2((1 - p))  
  
def classification\_error(p):  
 return 1 - np.max([p, 1 - p])  
  
x = np.arange(0.0, 1.0, 0.01)  
ent = [entropy(p) if p != 0 else None for p in x]  
c\_err = [classification\_error(i) for i in x]  
  
fig = plt.figure()  
ax = plt.subplot(111)  
  
for j, lab, c, in zip(  
 [ent, gini(x), c\_err],  
 ['Entropy', 'Gini', 'Classification Error'],  
 ['red', 'black', 'blue']):  
 line = ax.plot(x, j, label=lab, lw=1, color=c)  
  
ax.legend(loc='upper left', ncol=1, fancybox=True, shadow=False)  
  
plt.ylim([0, 1.1])  
plt.xlabel('p(j=1)')  
plt.ylabel('Impurity Index')  
plt.show()



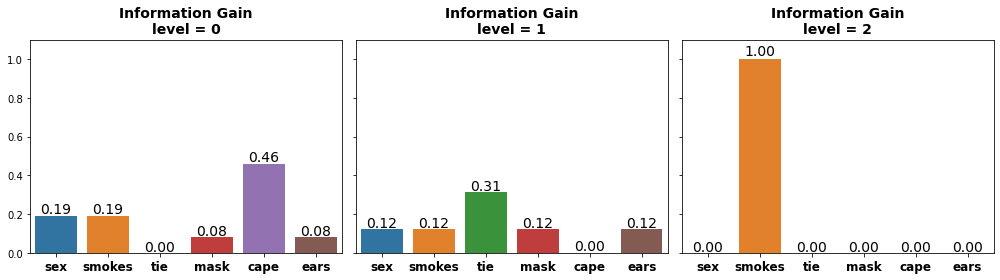
# Entropia e Ganho de Informação com Python

Você pode agora empregar as definições acima para verificar o ganho de informação de cada atributo do nosso exemplo de brinquedo.

def show\_values(axs, orient="v", space=.01):  
 def \_single(ax):  
 if orient == "v":  
 for p in ax.patches:  
 \_x = p.get\_x() + p.get\_width() / 2  
 \_y = p.get\_y() + p.get\_height() + (p.get\_height()\*0.01)  
 value = '{:.2f}'.format(p.get\_height())  
 ax.text(\_x, \_y+0.01, value, ha="center", fontsize=14)   
  
 if isinstance(axs, np.ndarray):  
 for idx, ax in np.ndenumerate(axs):  
 \_single(ax)  
 else:  
 \_single(axs)

# you can skip this code!  
  
from scipy.stats import entropy  
  
def informationgain(df,target,base=2):  
 E = entropy( df[target].value\_counts() / len(df) , base=base )  
  
 E\_x = {}  
  
 for c in df.drop(columns=[target]):  
 E\_x[c] = []  
 for v in df[c].unique():  
 e = len(df[df[c]==v])/len(df) \* entropy( df[df[c]==v]['class'].value\_counts() / len(df[df[c]==v]), base=base )  
 E\_x[c].append(np.round(e,2))  
  
 ig = {}  
  
 for c, p in E\_x.items():  
 ig[c] = E - sum(p)  
  
 return ig

# you can skip this code!  
  
ig\_root = informationgain(df=df.drop(columns=['Name']),target='class')  
ig\_level\_1 = informationgain(df=df.drop(columns=['Name'])[df.cape == 'no'],target='class')  
ig\_level\_2 = informationgain(df=df.drop(columns=['Name'])[ (df.cape == 'no') & (df.tie == 'yes') ],target='class')  
  
f , ax = plt.subplots(1,3,figsize=(14,4),sharey=True)  
  
for i, ig\_ in enumerate([ ig\_root, ig\_level\_1, ig\_level\_2 ]):  
 sns.barplot(x=list(ig\_.keys()), y=list(ig\_.values()), ax=ax[i])  
 ax[i].set\_title('Information Gain\nlevel = ' + str(i), fontsize=14, weight='bold')  
 ax[i].set\_ylim([0,1.1])  
  
 ax[i].set\_xticklabels(ax[i].get\_xticklabels(), fontsize=12, weight='bold')  
 show\_values(ax[i])  
  
plt.tight\_layout()  
plt.show()



Como você pode ver as escolhas dos atributos em cada nível segue exatamente o maior ganho de informação recalculado a cada nível da árvore.

# Informação Mútua e Seleção de Atributos

Toda essa discussão sobre o ganho de informação talvez não fosse tão importante se isso estivesse limitado ao uso em modelos de Árvore de Decisão. O ganho de informação, entretanto, é útil para a construção de muitos outros modelos pois nos dá um meio de avaliarmos atributos que são potencialmente mais preditores que outros, algo que ainda não havíamos explorado. Dentro da seleção de atributos para modelos preditivos, ou *feature selection*, o ganho de informação é normalmente denominado informação mútua (*mutual information*), correspondendo ao mesmo cálculo que você viu do ganho de informação empregando a Entropia, para entropia com base ( ou ).

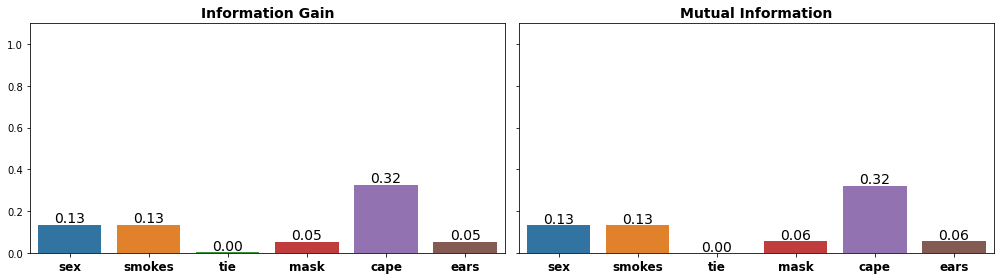
df\_tree

sex smokes tie mask cape ears class  
0 1 0 0 1 1 1 good  
1 1 0 0 1 1 1 good  
2 0 0 0 1 0 1 bad  
3 1 0 0 0 0 0 bad  
4 1 0 1 0 0 0 good  
5 1 1 1 0 0 0 bad

from sklearn.feature\_selection import mutual\_info\_classif  
  
print(mutual\_info\_classif(df\_tree.drop(columns=['class']), df\_tree['class'], discrete\_features=True))

[0.13230412 0.13230412 0. 0.05663301 0.31825708 0.05663301]

f , ax = plt.subplots(1,2,figsize=(14,4),sharey=True)  
  
ig\_root = informationgain(df=df.drop(columns=['Name']),target='class',base=None)  
  
i = 0  
sns.barplot(x=list(ig\_root.keys()), y=list(ig\_root.values()), ax=ax[i])  
ax[i].set\_title('Information Gain', fontsize=14, weight='bold')  
ax[i].set\_ylim([0,1.1])  
  
ax[i].set\_xticklabels(ax[i].get\_xticklabels(), fontsize=12, weight='bold')  
show\_values(ax[i])  
  
mutualinfo = mutual\_info\_classif(df\_tree.drop(columns=['class']), df\_tree['class'], discrete\_features=True)  
  
i = 1  
sns.barplot(x=df\_tree.drop(columns=['class']).columns, y=mutualinfo , ax=ax[i])  
ax[i].set\_title('Mutual Information', fontsize=14, weight='bold')  
ax[i].set\_ylim([0,1.1])  
  
ax[i].set\_xticklabels(ax[i].get\_xticklabels(), fontsize=12, weight='bold')  
show\_values(ax[i])  
  
plt.tight\_layout()  
plt.show()



# PlayBall

Retomando agora o exemplo clássico de Mitchell a informação mútua e o peso dos atributos no classificador também fornecem informações úteis para análise do modelo, e você pode notar que os primeiros dois níveis da Árvore são exatamente Outlook, Humidity e Wind, atributos que apresentam maior informação mútua a importância nesta ordem!

import graphviz   
dot\_data = tree.export\_graphviz(clf, out\_file=None,   
 feature\_names=list(df\_labels\_playball.columns.values),   
 class\_names=list(sorted(df\_playball['Play ball'].unique())),   
 filled=True, rounded=True,   
 special\_characters=False,  
 proportion=False, impurity=False, node\_ids=False,label=None)   
graph = graphviz.Source(dot\_data)   
graph.render('graph\_playball') # para gravação em .pdf  
graph

X = df\_labels\_playball  
y = df\_playball['Play ball']  
print(X.columns)  
print(mutual\_info\_classif(X, y, discrete\_features=True))  
print(clf.feature\_importances\_)

Index(['Outlook', 'Temperature', 'Humidity', 'Wind'], dtype='object')  
[0.17103394 0.02025554 0.10524435 0.03335912]  
[0.47111111 0. 0.28 0.24888889]

# A picture containing text, clock Description automatically generated

# Seleção de Atributos

A seleção de atributos para construção de um modelo tem muitos aspectos, incluindo muitos aspectos no domínio do problema (e não da solução!), e não seria possível abordarmos todos aqui. Mas vale a pena vermos algumas dessas técnicas e aprender a empregar o scikit-learn nessa tarefa.

Depois de abordagens que envolvem basicamente o entendimento dos dados, como excluir atributo único para cada instância ou um atributo com muitos dados ausentes, mas antes de empregar a informação mútua para a seleção de atributos, vale a pena removermos atributos de pouca variância.

## Removendo dados com pouca variância

O VarianceThreshold permite remover todos os atributos com variância abaixo de um limite e, por padrão, remove todos os atributos de variância zero (mesmo valor em todas as amostras), são os dados que não carregam nenhuma informação ou, como você viu entropia . O

df\_tree.var()

/usr/local/lib/python3.7/dist-packages/ipykernel\_launcher.py:1: FutureWarning: Dropping of nuisance columns in DataFrame reductions (with 'numeric\_only=None') is deprecated; in a future version this will raise TypeError. Select only valid columns before calling the reduction.  
 """Entry point for launching an IPython kernel.

sex 0.166667  
smokes 0.166667  
tie 0.266667  
mask 0.300000  
cape 0.266667  
ears 0.300000  
dtype: float64

from sklearn.feature\_selection import VarianceThreshold  
  
X = df\_tree.drop(columns=['class'])  
sel = VarianceThreshold(threshold=0.1)  
sel.fit\_transform(X)

array([[1, 0, 0, 1, 1, 1],  
 [1, 0, 0, 1, 1, 1],  
 [0, 0, 0, 1, 0, 1],  
 [1, 0, 0, 0, 0, 0],  
 [1, 0, 1, 0, 0, 0],  
 [1, 1, 1, 0, 0, 0]])

No nosso exemplo há poucos atributos e as variâncias próximas não permitem selecionar atributos para remoção. Mas o exemplo vale para mostrar como empregar o estimador.

## Selecionando atributos com o SelectKBest

***As técnicas de seleção de atributos são bastante sofisticadas. Não espero que você não dê muito atenção a elas agora, em especial para os detalhes de implementação. O importante é que você tenha esses conceitos de métricas que permitem identificar melhores atributos para problemas de classificação e regressão e que, sabendo existirem funções que implementam essa solução, lembrar de buscá-las quando for necessário. Nos problemas e exemplos que seguem no curso não voltaremos a empregar esses métodos.***

As classes do sklearn.feature\_selection podem ser usadas para seleção de atributos e redução de dimensionalidade em conjuntos de amostras, seja para melhorar o *score* dos estimadores ou para impulsionar seu desempenho em conjuntos de dados de dimensões muito altas.

A seleção univariada de recursos funciona selecionando os melhores atributos com base em testes estatísticos univariados, o que inclui a informação mútua. São 3 métodos, mas que tem o mesmo tipo de princípio:

* SelectKBest remove tudo, exceto os atributos de maior pontuação
* SelectPercentile remove todos os atributos, exceto a maior porcentagem de pontuação especificada
* GenericUnivariateSelectpermite realizar a seleção univariada de recursos com uma estratégia configurável

Aplicando o SelectKBest ao nosso exemplo teríamos:

df\_tree

sex smokes tie mask cape ears class  
0 1 0 0 1 1 1 good  
1 1 0 0 1 1 1 good  
2 0 0 0 1 0 1 bad  
3 1 0 0 0 0 0 bad  
4 1 0 1 0 0 0 good  
5 1 1 1 0 0 0 bad

from sklearn.feature\_selection import SelectKBest  
from sklearn.feature\_selection import chi2  
from sklearn.feature\_selection import mutual\_info\_classif  
  
X = df\_tree.drop(columns=['class'])  
y = df\_tree['class']  
  
select\_features = SelectKBest(mutual\_info\_classif, k=3).fit(X, y)  
print( select\_features.get\_support() )  
print( X.columns[select\_features.get\_support()] )  
  
X\_new = select\_features.transform(X)  
  
pd.DataFrame( X\_new, columns= X.columns[select\_features.get\_support()] )

[False False False True True True]  
Index(['mask', 'cape', 'ears'], dtype='object')

mask cape ears  
0 1 1 1  
1 1 1 1  
2 1 0 1  
3 0 0 0  
4 0 0 0  
5 0 0 0

Esses métodos podem ser empregados tanto para modelos de classificação como de regressão com diferentes métricas de pontuação:

* Para a regressão: **f\_regression, mutual\_info\_regression**
* Para a classificação: **chi2, f\_classif, mutual\_info\_classif**

E você deve ter cuidado para não empregar métricas de classificação em modelos de regressão e vice-versa.

## Limitações

Note que se fizermos a seleção de features como acima, teríamos eliminado o atributo tie, assim como o atributo smokes pelo critério de variância mínima, dois atributos que foram empregados no modelo de árvore. Isso porque esses critérios somente levam em consideração a variância e a informação mútua unicamente do atributo, sem levar em conta possíveis relações dos atributos para a predição. Assim, é importante você ter bastante cuidado ao remover dados de modelo a partir de critérios univariados.

## Eliminação recursiva de atributos

Uma última técnica que trataremos aqui é a eliminação recursiva de atributos. A maior parte dos estimadores fornece, após o treinamento, a importância de cada atributo (como estimador.coef\_ ou estimador.feature\_importances\_). A eliminação recursiva de recursos (RFE) consiste em selecionar atributos considerando recursivamente conjuntos cada vez menores na quantidade de atributos. Primeiro, o estimador é treinado no conjunto inicial, em seguida, os atributos menos importantes são removidos do conjunto atual repetindo-se o procedimento recursivamente até um número desejado.

RFECV executa RFE em um loop de validação cruzada para encontrar o número ideal de recursos e o exemplo poderá ser útil para aplicar em conjuntos de dados mais complexos que o nosso conjunto de dados de brinquedo.

from sklearn.tree import DecisionTreeClassifier  
from sklearn.feature\_selection import RFECV  
  
X = df\_labels  
y = df['class']  
  
clf = DecisionTreeClassifier(criterion='entropy')   
clf.fit(X,y)  
  
min\_features\_to\_select = 1 # Minimum number of features to consider  
rfecv = RFECV(  
 estimator=clf,  
 step=1,  
 cv=2,  
 scoring="accuracy",  
 min\_features\_to\_select=min\_features\_to\_select,  
)  
rfecv.fit(X, y)  
  
print("Optimal number of features : %d" % rfecv.n\_features\_)  
  
# Plot number of features VS. cross-validation scores  
plt.figure()  
plt.xlabel("Number of features selected")  
plt.ylabel("Cross validation score (accuracy)")  
plt.plot(  
 range(min\_features\_to\_select, len(rfecv.grid\_scores\_) + min\_features\_to\_select),  
 rfecv.grid\_scores\_,  
)  
plt.show()

Optimal number of features : 1



pd.DataFrame({'atributtes': df\_labels.columns, 'feature\_importances': clf.feature\_importances\_}).sort\_values('feature\_importances',ascending=False)

atributtes feature\_importances  
4 cape 0.459148  
1 smokes 0.333333  
2 tie 0.207519  
0 sex 0.000000  
3 mask 0.000000  
5 ears 0.000000

Como você pode notar os atributos selecionados são exatamente os que empregamos e foram gerados pelo modelo de Árvore de Decisão. Você pode aplicar esse procedimento ao exemplo do PlayBall, mas vou deixar isso com você.

# CASO: Classifying Political Parties Based on Congressional Votes

Vamos agora aplicar o modelo de Árvore de Decisão para um caso com dados reais e fazer o procedimento completo com o emprego de conjuntos de treinamento e teste, e as medidas de eficiência do modelo, para que você tenha um código completo para aplicar em outros problemas.

Este conjunto de dados inclui votos para cada um dos congressistas da Câmara dos Representantes dos EUA nos 16 votos principais identificados pelo CQA. Nosso objetivo é construir selecionar um melhor modelo para prever a filiação partidária dos congressistas com base nos registros de votação.

Fonte: <http://archive.ics.uci.edu/ml/datasets/Congressional+Voting+Records>

df = pd.read\_csv('http://archive.ics.uci.edu/ml/machine-learning-databases/voting-records/house-votes-84.data',header=None,prefix='Vote\_')  
df.rename(columns={'Vote\_0':'class'},inplace=True)  
df.head()

class Vote\_1 Vote\_2 Vote\_3 Vote\_4 Vote\_5 Vote\_6 Vote\_7 Vote\_8 Vote\_9 \  
0 republican n y n y y y n n n   
1 republican n y n y y y n n n   
2 democrat ? y y ? y y n n n   
3 democrat n y y n ? y n n n   
4 democrat y y y n y y n n n   
  
 Vote\_10 Vote\_11 Vote\_12 Vote\_13 Vote\_14 Vote\_15 Vote\_16   
0 y ? y y y n y   
1 n n y y y n ?   
2 n y n y y n n   
3 n y n y n n y   
4 n y ? y y y y

df.shape

(435, 17)

## Preparação dos Dados

Não existem dados ausentes, mas as abstenções '?' são um grande número de casos. Vamos esperar que elas possam também ajudar a identificar o partido do congressista uma vez que isso parece ser mais uma informação importante do que exatamente ausência de informação.

df.isnull().sum().sum()

0

(df == '?').sum()

class 0  
Vote\_1 12  
Vote\_2 48  
Vote\_3 11  
Vote\_4 11  
Vote\_5 15  
Vote\_6 11  
Vote\_7 14  
Vote\_8 15  
Vote\_9 22  
Vote\_10 7  
Vote\_11 21  
Vote\_12 31  
Vote\_13 25  
Vote\_14 17  
Vote\_15 28  
Vote\_16 104  
dtype: int64

Agora precisamos fazer o encode dos dados. A rigor deveríamos fazer o hot encode, mas por simplicidade será suficiente aqui fazermos apenas o label encode do dados.

from sklearn.preprocessing import OneHotEncoder, LabelEncoder  
  
df\_labels = df.drop(columns=['class']).apply(LabelEncoder().fit\_transform)  
  
pd.concat([ df\_labels, df[['class']] ], axis=1).head()

Vote\_1 Vote\_2 Vote\_3 Vote\_4 Vote\_5 Vote\_6 Vote\_7 Vote\_8 Vote\_9 \  
0 1 2 1 2 2 2 1 1 1   
1 1 2 1 2 2 2 1 1 1   
2 0 2 2 0 2 2 1 1 1   
3 1 2 2 1 0 2 1 1 1   
4 2 2 2 1 2 2 1 1 1   
  
 Vote\_10 Vote\_11 Vote\_12 Vote\_13 Vote\_14 Vote\_15 Vote\_16 class   
0 2 0 2 2 2 1 2 republican   
1 1 1 2 2 2 1 0 republican   
2 1 2 1 2 2 1 1 democrat   
3 1 2 1 2 1 1 2 democrat   
4 1 2 0 2 2 2 2 democrat

Antes de testarmos diferentes classificadores vamos aplicar um modelo básico para verificar o formato dos dados para aplicação do modelo.

from sklearn.tree import DecisionTreeClassifier  
from sklearn.model\_selection import train\_test\_split  
from sklearn import neighbors  
from sklearn.metrics import classification\_report  
  
X = df\_labels  
y = df['class']  
  
X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, stratify=y, test\_size=0.3, random\_state=123)  
  
clf = DecisionTreeClassifier()   
  
clf.fit(X\_train,y\_train)  
  
y\_pred = clf.predict(X\_test)  
  
accuracy = clf.score(X\_test, y\_test)  
print('\nScore de Acuracidade (2):\n')  
print(f'{accuracy:.2f}')  
  
print('\nClassification Report:\n')  
print(classification\_report(y\_test, y\_pred))

Score de Acuracidade (2):  
  
0.92  
  
Classification Report:  
  
 precision recall f1-score support  
  
 democrat 0.91 0.97 0.94 80  
 republican 0.96 0.84 0.90 51  
  
 accuracy 0.92 131  
 macro avg 0.93 0.91 0.92 131  
weighted avg 0.93 0.92 0.92 131

Parece tudo certo e podemos ainda verificar a árvore produzida neste treinamento.

import graphviz   
dot\_data = tree.export\_graphviz(clf, out\_file=None,   
 feature\_names=list(df\_labels.columns.values),   
 class\_names=list(sorted(df['class'].unique())),   
 filled=True, rounded=True,   
 special\_characters=False,  
 proportion=False, impurity=False, node\_ids=False,label=None)   
graph = graphviz.Source(dot\_data)   
graph.render('graph\_democratic') # para gravação em .pdf  
graph

Diagram, text

Description automatically generatedDiagram

Description automatically generated

e os valores de informações mútua e importância dos atributos mostram a concordância dos principais atributos que encabeçam a Árvore.

print('Mutual information and the attribute with higher Mutual Info: \n')  
print(mutual\_info\_classif(X, y, discrete\_features=True))  
print(X.columns[ mutual\_info\_classif(X, y, discrete\_features=True).argmax() ])  
print()  
print('Feature Importances and the attribute with higher Feature Importances: \n')  
print(clf.feature\_importances\_)  
print(X.columns[ clf.feature\_importances\_.argmax() ])

Mutual information and the attribute with higher Mutual Info:   
  
[8.73872293e-02 2.49962316e-04 2.99660509e-01 5.12951549e-01  
 2.92820364e-01 1.02055280e-01 1.37023508e-01 2.35826473e-01  
 2.15261646e-01 3.52248115e-03 7.43690939e-02 2.59411124e-01  
 1.57899640e-01 2.32400929e-01 1.52771138e-01 7.06865546e-02]  
Vote\_4  
  
Feature Importances and the attribute with higher Feature Importances:   
  
[0.00612052 0.0069473 0.01350356 0.84658144 0. 0.  
 0. 0.00752624 0. 0.02501028 0.03028702 0.00416838  
 0. 0.00926307 0.01378591 0.03680628]  
Vote\_4

X.columns[ np.argsort(- mutual\_info\_classif(X, y, discrete\_features=True))[0:5] ]

Index(['Vote\_4', 'Vote\_3', 'Vote\_5', 'Vote\_12', 'Vote\_8'], dtype='object')

X.columns[ np.argsort(- clf.feature\_importances\_)[0:5] ]

Index(['Vote\_4', 'Vote\_16', 'Vote\_11', 'Vote\_10', 'Vote\_15'], dtype='object')

# Comparando os Modelos

Vamos agora comparar os modelos que você já conhece de classificação e o modelo de Árvore de Decisão.

from sklearn.linear\_model import LogisticRegression  
from sklearn.neighbors import KNeighborsClassifier  
from sklearn.tree import DecisionTreeClassifier  
  
X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, stratify=y, test\_size=0.3, random\_state=123)  
  
base\_estimators = [ LogisticRegression(),  
 neighbors.KNeighborsClassifier(5),  
 DecisionTreeClassifier(),  
 DecisionTreeClassifier(max\_depth=2) ]  
  
best\_score = 0  
  
for estimator in base\_estimators:  
 clf = estimator  
 clf.fit(X\_train, y\_train)  
  
 y\_pred = clf.predict(X\_test)  
  
 print(clf)   
  
 accuracy = clf.score(X\_test, y\_test)  
 print('\nScore de Acuracidade (2):\n')  
 print(f'{accuracy:.2f}')  
  
 if accuracy > best\_score:  
 best\_score = accuracy  
 best\_model = clf  
  
 print('\nClassification Report:\n')  
 print(classification\_report(y\_test, y\_pred))  
  
  
print('\n Best Model: ' , best\_model, ' score = ', best\_score)

LogisticRegression()  
  
Score de Acuracidade (2):  
  
0.94  
  
Classification Report:  
  
 precision recall f1-score support  
  
 democrat 0.93 0.97 0.95 80  
 republican 0.96 0.88 0.92 51  
  
 accuracy 0.94 131  
 macro avg 0.94 0.93 0.93 131  
weighted avg 0.94 0.94 0.94 131  
  
KNeighborsClassifier()  
  
Score de Acuracidade (2):  
  
0.92  
  
Classification Report:  
  
 precision recall f1-score support  
  
 democrat 0.94 0.94 0.94 80  
 republican 0.90 0.90 0.90 51  
  
 accuracy 0.92 131  
 macro avg 0.92 0.92 0.92 131  
weighted avg 0.92 0.92 0.92 131  
  
DecisionTreeClassifier()  
  
Score de Acuracidade (2):  
  
0.92  
  
Classification Report:  
  
 precision recall f1-score support  
  
 democrat 0.91 0.97 0.94 80  
 republican 0.96 0.84 0.90 51  
  
 accuracy 0.92 131  
 macro avg 0.93 0.91 0.92 131  
weighted avg 0.93 0.92 0.92 131  
  
DecisionTreeClassifier(max\_depth=2)  
  
Score de Acuracidade (2):  
  
0.95  
  
Classification Report:  
  
 precision recall f1-score support  
  
 democrat 0.95 0.96 0.96 80  
 republican 0.94 0.92 0.93 51  
  
 accuracy 0.95 131  
 macro avg 0.95 0.94 0.94 131  
weighted avg 0.95 0.95 0.95 131  
  
  
 Best Model: DecisionTreeClassifier(max\_depth=2) score = 0.9465648854961832

Com os parâmetros acima o melhor estimador, levando em consideração apenas o critério de acuracidade (*score*), foi a Árvore de Decisão com o critério gini de seleção dos atributos (é o padrão) e com uma profundidade máxima de 2 da Árvore.

***Poda: A técnica de reduzir a profundidade da Árvore de Decisão é chamada de Poda e permite reduzir o overfitting (sobreajuste) dos modelos, um problema geralmente encontrado em Árvores de Decisão*.**

O melhor modelo produz então a seguinte Árvore de Decisão.

clf = best\_model  
  
import graphviz   
dot\_data = tree.export\_graphviz(clf, out\_file=None,   
 feature\_names=list(df\_labels.columns.values),   
 class\_names=list(sorted(df['class'].unique())),   
 filled=True, rounded=True,   
 special\_characters=False,  
 proportion=False, impurity=False, node\_ids=False,label=None)   
graph = graphviz.Source(dot\_data)   
graph.render('graph\_cartoons') # para gravação em .pdf  
graph

A picture containing text, clock, sign

Description automatically generated

Você pode acessar a documentação dos dados (<http://archive.ics.uci.edu/ml/machine-learning-databases/voting-records/house-votes-84.names>) e verificar a que eleições se referem os votos acima:

* **Vote\_4: physician-fee-freeze**
* **Vote\_3: adoption-of-the-budget-resolution**
* **Vote\_11: synfuels-corporation-cutback**

E, de fato, seria suficiente ainda somente a informação do Vote\_4: physician-fee-freeze para classificar os congressistas. Em 1984, o governo federal impõe um congelamento de um ano nos pagamentos para os serviços de saúde, sendo os serviços de saúde um tema polêmico que ainda hoje divide republicamos e democratas.

# Selecionando outros Modelos

***Você pode adaptar o código de seleção de modelos acima para explorar, por exemplo, diferentes valores de para o K-vizinhos mais Próximos ou ainda diferentes profundidades da Árvore. Para isso basta implementar no código original um procedimento mais ou menos como abaixo:***

%%script false  
  
for k in range(2,10):  
 clf = neighbors.KNeighborsClassifier(k)  
 ...  
  
# OU  
  
for deep in range(2,10):  
 clf = DecisionTreeClassifier(max\_depth=deep)  
 ...

Nas próximas aulas você aprenderá como fazer isso com uma função do Scikit-learn.

# Sumário da Aula

Aqui você conheceu e aprendeu a empregar o modelo de **Árvores de Decisão**. Árvores de Decisão são modelos bastante empregadas e muitas vezes são o primeiro modelo criado para proporcionar um *baseline* para modelos melhores. Esses modelos ainda permitiram introduzir uma série de conceitos importantes como **Entropia** e **Ganho de Informação** que desempenham um papel importante, sobretudo, na **Seleção de Atributos** para modelos de aprendizado.

Árvores de decisão podem ser empregadas tanto para modelos de classificação, como empregamos aqui, como também para criar estimadores de regressão não lineares. Elas são eficientes preditores, mas **Árvores muito profundas tendem a criar modelos com sobreajuste**. Para reduzir isso podemos fazer uma **poda** da Árvore, limitando a sua profundidade (parâmetro que empregamos aqui na seleção de melhores parâmetros do estimador), e outro método bastante empregado para reduzir o sobreajuste é empregar vários estimadores de Árvore não correlacionados, as **Florestas Aleatórias** ou **Random Forest**.

# Para Saber Mais

* Acesse [https://ml-playground.com/#](https://ml-playground.com/) e experimente alguns modelos de classificação sem qualquer programação. Se estiver interessado ainda pode verificar como exportar e importar dados do [https://ml-playground.com/#](https://ml-playground.com/) para o Python. Dica, os dados são salvos e carregados em formato .json.
* O exemplo clássico Árvores de Decisão de Mitchell, Tom Michael (1997) em **Machine Learning** é também explorado em Sayad, Saed (2021). **An Introduction to Data Science**, Disponível em: <https://www.saedsayad.com/data_mining_map.htm> e ajuda a entender em um passo a passo como é construída uma árvore de decisão e são calculados o ganho de informação.
* Acesse Jake VanderPlas. **Python Data Science Handbook** Disponível em: <https://jakevdp.github.io/PythonDataScienceHandbook/> e veja o capítulo **In-Depth: Decision Trees and Random Forests**, em especial o exemplo: **Example: Random Forest for Classifying Digits** classificam dados de imagens que não exploramos aqui.

# Referências

* Mitchell, Tom Michael (1997). **Machine Learning**. New York, NY: McGraw-Hill. ISBN 0070428077.
* Subasi, A. (2020). **Machine learning techniques. Practical Machine Learning for Data Analysis Using Python**, 91–202. <doi:10.1016/b978-0-12-821379-7.00003-5>
* Larose, Chantal D.; Larose, Daniel T. **Data Science Using Python and R** Hoboken: Wiley, c2019. E-book (259 p.) (Wiley Series on Methods and Applications in Data Mining Ser.). ISBN 9781119526834 (electronic bk.). Disponível em: <https://www3.mackenzie.br/biblioteca_virtual/index.php?tipoBiblio=ebookcentral&flashObg=n>
* Kotu, Vijay; Deshpande, Balachandre **Data Science: concepts and practice**. 2nd ed. Cambridge, [England]: Morgan Kaufmann, c2019. E-book (570 p.) ISBN 9780128147627 (electronic bk.). Disponível em: <http://pergamum.mackenzie.br:8080/pergamumweb/vinculos/00003c/00003cef.jpg>.
* Jake VanderPlas. **Python Data Science Handbook** O'Reilly Media, Inc. (2016). ISBN: 9781491912058. Disponível em: <https://jakevdp.github.io/PythonDataScienceHandbook/>. Acesso: 06 de Novembro de 2021.