## 4. K-Vizinhos Mais Próximos, Validação Cruzada e *GridSearch*

Nesta trilha você vai aprender:

* O Modelo de Classificação K-Vizinhos Mais Próximos
* Conjunto de Validação e Teste, e como aplicar a Validação Cruzada de Dados
* Como selecionar melhores hiperparâmetros dos Modelos

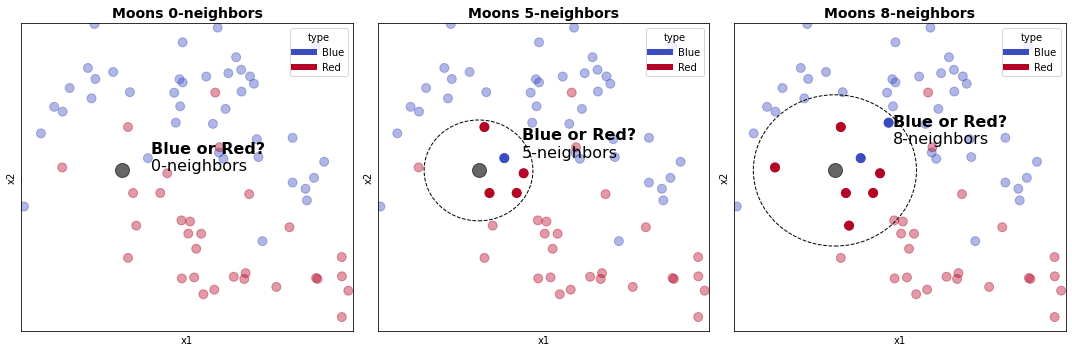
Nesta e na próxima trilha vamos nos deter unicamente em modelos para classificação de dados e aqui vamos explorar um único modelo, o de K-Vizinhos Mais Próximos. A partir de um problema bastante simples vamos incrementar esse modelo para que você possa explorar seus conceitos, mas também vários outros conceitos e técnicas de uso geral no aprendizado de máquina e classificadores, como o emprego de estimadores para normalização e encode, conjuntos de validação e teste, validação cruzada e seleção de hiperparâmetros.

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

# K-Vizinhos Mais Próximos

O K-Vizinhos mais Próximos, ou Knn (do inglês, K nearest neighbors) é um dos modelos mais simples de classificação, mas também bastante empregado. Seu funcionamento se baseia em um princípio muito simples que nós mesmos adotamos frequentemente no dia a dia, nós, por exemplo, nos assemelhamos às pessoas mais próximas de nós. Assim, é razoável supor que podemos classificar uma instância de acordo com a classe de seus vizinhos mais próximos.

# you can skip this code!  
  
from sklearn.datasets import make\_moons  
from sklearn.neighbors import DistanceMetric  
cmap\_data = plt.cm.coolwarm   
  
X, y = make\_moons(n\_samples=100, noise=0.25, random\_state=1234)  
moons = pd.DataFrame({'x1':X[:, 0], 'x2':X[:, 1], 'y':y})  
  
X, y = make\_moons(n\_samples=1, noise=0.25, random\_state=123)  
# amoon = pd.DataFrame({'x1':X[:, 0], 'x2':X[:, 1], 'y':y})  
amoon = pd.DataFrame({'x1':[-.2], 'x2':[.3]})  
  
dist = DistanceMetric.get\_metric('euclidean')  
moons['distance'] = dist.pairwise(moons[['x1','x2']],amoon[['x1','x2']])  
  
f, ax = plt.subplots(1,3,figsize=(15,5))  
cmap\_data = plt.cm.coolwarm   
  
for i in range(3):  
 neighbors = moons.nsmallest([0,5,8][i],'distance')  
 ax[i].scatter(moons.x1, moons.x2, c=moons.y, cmap=cmap\_data, alpha=0.4, s=80)  
 ax[i].scatter(neighbors.x1, neighbors.x2, c=neighbors.y, cmap=cmap\_data, s=80)  
 ax[i].plot(amoon.x1, amoon.x2, 'ko', markersize=14, alpha=0.6)  
  
 a\_circle = plt.Circle((amoon.x1, amoon.x2), neighbors.distance.max()+0.05, edgecolor='k', linestyle='dashed',fill=False)  
 ax[i].add\_artist(a\_circle)  
  
 ax[i].text(amoon.x1+0.2+i/10, amoon.x2+i/10,'Blue or Red?\n', weight='bold', fontsize=16)  
 ax[i].text(amoon.x1+0.2+i/10, amoon.x2+i/10,' \n' + str([0,5,8][i]) + '-neighbors', fontsize=16)  
 ax[i].set\_title('Moons ' + str([0,5,8][i]) + '-neighbors',weight='bold',fontsize=14)  
 ax[i].set\_xlabel("x1")  
 ax[i].set\_ylabel("x2")  
  
 custom\_lines = [Line2D([0], [0], color=cmap\_data(0.), lw=6),  
 Line2D([0], [0], color=cmap\_data(1.), lw=6)]  
 ax[i].legend(custom\_lines, ['Blue', 'Red'], loc='upper right',title='type')  
  
 ax[i].set\_xlim([-0.9,1.4])  
 ax[i].set\_ylim([-0.9,1.4])  
 ax[i].set\_xticks([])  
 ax[i].set\_yticks([])  
 # ax[i].axis('equal')  
  
plt.tight\_layout()  
plt.show()



O número k define quantos vizinhos queremos empregar na classificação. No exemplo acima, considerando 5 vizinhos, você pode verificar que há uma chance de de que a moon *cinza* seja uma moon *red*, e com 8 vizinhos a chance é de de ser *red*. Assim, assumimos nos dois casos a classe *red* para prever a classe do ponto selecionado.

## Knn *by scratch*

O conceito do Knn é bastante simples o que permite implementar o algoritmo e verificar o seu funcionamento sem qualquer API ou pacote adicional. Basicamente o modelo consiste na execução de 3 passos:

1. Calcular as distâncias do elemento desejado para os demais
2. Encontrar os k-vizinhos mais próximos
3. Retornar a classe mais frequente entre dos k-vizinhos

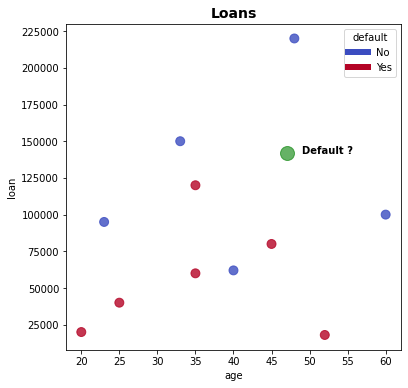
O exemplo a seguir é um *exemplo de brinquedo* e foi livremente adaptado de [Sayad (2021)](https://www.saedsayad.com/data_mining_map.htm). A ideia é prever, com base na idade, valor e retorno de empréstimos realizados anteriormente, a possibilidade de fazermos um bom ou mal empréstimo para um novo caso que desejamos avaliar.

loans = pd.DataFrame({'age':[25,35,45,20,35,52,23,40,60,48,33],  
 'loan':[40000,60000,80000,20000,120000,18000,95000,62000,100000,220000,150000],  
 'default':[1,1,1,1,1,1,0,0,0,0,0] }) # 1='yes'  
  
case = pd.DataFrame({'age':[47],'loan':[142000]})  
  
display(loans)  
display(case)

age loan default  
0 25 40000 1  
1 35 60000 1  
2 45 80000 1  
3 20 20000 1  
4 35 120000 1  
5 52 18000 1  
6 23 95000 0  
7 40 62000 0  
8 60 100000 0  
9 48 220000 0  
10 33 150000 0

age loan  
0 47 142000

f, ax = plt.subplots(1,1,figsize=(6,6))  
cmap\_data = plt.cm.coolwarm   
  
ax.scatter(loans.age, loans.loan, c=loans.default, cmap=cmap\_data, alpha=0.8, s=80)  
ax.plot(case.age, case.loan, 'go', markersize=14, alpha=0.6)  
  
plt.text(case.age+2, case.loan,'Default ?', weight='bold')  
plt.title('Loans',weight='bold',fontsize=14)  
plt.xlabel("age")  
plt.ylabel("loan")  
  
custom\_lines = [Line2D([0], [0], color=cmap\_data(0.), lw=6),  
 Line2D([0], [0], color=cmap\_data(1.), lw=6)]  
plt.legend(custom\_lines, ['No', 'Yes'], loc='upper right',title='default')  
  
plt.show()



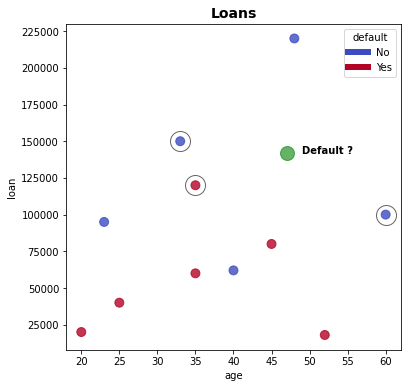
*Default* representa os casos em que o empréstimo não foi pago e, portanto, uma operação ruim. E nosso objetivo é prever, com base no conjunto de treinamento, qual classe (*Default Yes* ou *No*) de um novo empréstimo de para um cliente com anos.

Você pode, então, calcular diretamente a distância do par para todos os demais dados e, na sequência, ordenar os dados para identificar os k-vizinhos mais próximos e a classe mais frenquente dentre eles. Aqui empregamos vizinhos mais próximos.

from sklearn.neighbors import DistanceMetric  
  
dist = DistanceMetric.get\_metric('euclidean')  
d = dist.pairwise(loans[['age','loan']],case[['age','loan']])  
  
loans['distance'] = d  
display(loans.sort\_values('distance'))

age loan default distance  
10 33 150000 0 8000.012250  
4 35 120000 1 22000.003273  
8 60 100000 0 42000.002012  
6 23 95000 0 47000.006128  
2 45 80000 1 62000.000032  
9 48 220000 0 78000.000006  
7 40 62000 0 80000.000306  
1 35 60000 1 82000.000878  
0 25 40000 1 102000.002373  
3 20 20000 1 122000.002988  
5 52 18000 1 124000.000101

k3\_neighbors = loans.nsmallest(3,'distance')  
  
f, ax = plt.subplots(1,1,figsize=(6,6))  
cmap\_data = plt.cm.coolwarm   
  
ax.scatter(k3\_neighbors.age, k3\_neighbors.loan, edgecolors='k', c='w', alpha=0.6, s=400)  
ax.scatter(loans.age, loans.loan, c=loans.default, cmap=cmap\_data, alpha=0.8, s=80)  
ax.plot(case.age, case.loan, 'go', markersize=14, alpha=0.6)  
  
plt.text(case.age+2, case.loan,'Default ?', weight='bold')  
plt.title('Loans',weight='bold',fontsize=14)  
plt.xlabel("age")  
plt.ylabel("loan")  
  
custom\_lines = [Line2D([0], [0], color=cmap\_data(0.), lw=6),  
 Line2D([0], [0], color=cmap\_data(1.), lw=6)]  
plt.legend(custom\_lines, ['No', 'Yes'], loc='upper right',title='default')  
  
plt.show()  
  
display(k3\_neighbors)



age loan default distance  
10 33 150000 0 8000.012250  
4 35 120000 1 22000.003273  
8 60 100000 0 42000.002012

Com base nisso podemos estimar que a classe desse empréstimo é *Default No* (zeros) com probabilidade de e, portanto, deveríamos conceder o empréstimo.

## Revisando o Modelo: Normalizando os Dados

O cálculo de distâncias como medida de similaridade (menor distância indicando maior similaridade) pode, entretanto, apresentar grandes desvios quando empregamos variáveis ​com escalas muito diferentes ou variáveis ​​numéricas e categóricas em conjunto.

Veja que no nosso exemplo os valores dos empréstimos encontram-se em uma escala de valores vezes maior que a idade e, por isso, idade teve pouca ou nenhuma influência nas distâncias e poderíamos ter o mesmo resultado empregando somente o os valores dos empréstimos.

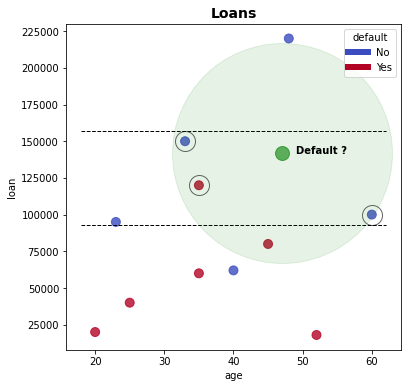
Vários modelos de aprendizado de máquina são baseados em distância como medida de similaridade e são portanto *sensíveis* à normalização dos dados e devemos aplicá-la quando empregados dados em diferentes escalas.

dist = DistanceMetric.get\_metric('euclidean')  
d = dist.pairwise(loans[['loan']],case[['loan']])  
  
loans['distance\_loans'] = d  
display(loans.sort\_values('distance'))

age loan default distance distance\_loans  
10 33 150000 0 8000.012250 8000.0  
4 35 120000 1 22000.003273 22000.0  
8 60 100000 0 42000.002012 42000.0  
6 23 95000 0 47000.006128 47000.0  
2 45 80000 1 62000.000032 62000.0  
9 48 220000 0 78000.000006 78000.0  
7 40 62000 0 80000.000306 80000.0  
1 35 60000 1 82000.000878 82000.0  
0 25 40000 1 102000.002373 102000.0  
3 20 20000 1 122000.002988 122000.0  
5 52 18000 1 124000.000101 124000.0

A maior escala dos valores de loan faz com que o '*peso*' dessa variável no cálculo das distâncias praticamente '*anule*' a importância da variável *age*.

f, ax = plt.subplots(1,1,figsize=(6,6))  
cmap\_data = plt.cm.coolwarm   
  
ax.plot(case.age, case.loan, 'go', markersize=220, alpha=0.10)  
ax.scatter(k3\_neighbors.age, k3\_neighbors.loan, edgecolors='k', c='w', alpha=0.6, s=400)  
ax.scatter(loans.age, loans.loan, c=loans.default, cmap=cmap\_data, alpha=0.8, s=80)  
ax.plot(case.age, case.loan, 'go', markersize=14, alpha=0.6)  
  
plt.hlines(157000,18,62,linestyles='dashed',lw=1)  
plt.hlines(93000,18,62,linestyles='dashed',lw=1)  
  
plt.text(case.age+2, case.loan,'Default ?', weight='bold')  
plt.title('Loans',weight='bold',fontsize=14)  
plt.xlabel("age")  
plt.ylabel("loan")  
  
custom\_lines = [Line2D([0], [0], color=cmap\_data(0.), lw=6),  
 Line2D([0], [0], color=cmap\_data(1.), lw=6)]  
plt.legend(custom\_lines, ['No', 'Yes'], loc='upper right',title='default')  
  
plt.show()  
  
display(k3\_neighbors)



age loan default distance  
10 33 150000 0 8000.012250  
4 35 120000 1 22000.003273  
8 60 100000 0 42000.002012

A solução é normalizar os dados do conjunto de treinamento. Existem várias técnicas de normalização e aqui vamos empregar a normalização *min\_max* que leva todos os valores ao intervalo .

### Normalização com scikit-learn: Função *vs* Estimador

Você pode simplesmente empregar uma função minmax\_scale do scikit-learn para fazer a normalização e existem outras funções para os demais tipos de normalização.

from sklearn.preprocessing import minmax\_scale  
  
minmax\_scale(loans[['age','loan']])

array([[0.125 , 0.10891089],  
 [0.375 , 0.20792079],  
 [0.625 , 0.30693069],  
 [0. , 0.00990099],  
 [0.375 , 0.5049505 ],  
 [0.8 , 0. ],  
 [0.075 , 0.38118812],  
 [0.5 , 0.21782178],  
 [1. , 0.40594059],  
 [0.7 , 1. ],  
 [0.325 , 0.65346535]])

Mas o uso de funções não permite aplicarmos a mesma regra de normalização em outros conjuntos de dados!

minmax\_scale(case)

array([[0., 0.]])

Neste caso, é melhor empregarmos um *estimador* do scikit-learn. O cálculo é o mesmo da função minmaxscaler, mas o uso de um estimador permite salvarmos os parâmetros empregados para a mesma aplicação em outros conjuntos de dados.

*Na normalização de dados com o scikit-learn é preferível uso de estimadores no lugar de funções para que a mesma transformação possa ser efetuada em outros conjuntos de dados. O mesmo princípio se aplica a outras transformações, como os encodes.*

Vamos então aplicar o estimador MinMaxScaler no lugar da função.

# apenas mostrando o resultado do estimador  
from sklearn.preprocessing import MinMaxScaler  
  
scaler = MinMaxScaler()  
print(scaler.fit(loans[['age','loan']]))  
  
print(scaler.data\_max\_)  
  
print(scaler.transform(loans[['age','loan']]))  
print(scaler.transform(case))

MinMaxScaler()  
[6.0e+01 2.2e+05]  
[[0.125 0.10891089]  
 [0.375 0.20792079]  
 [0.625 0.30693069]  
 [0. 0.00990099]  
 [0.375 0.5049505 ]  
 [0.8 0. ]  
 [0.075 0.38118812]  
 [0.5 0.21782178]  
 [1. 0.40594059]  
 [0.7 1. ]  
 [0.325 0.65346535]]  
[[0.675 0.61386139]]

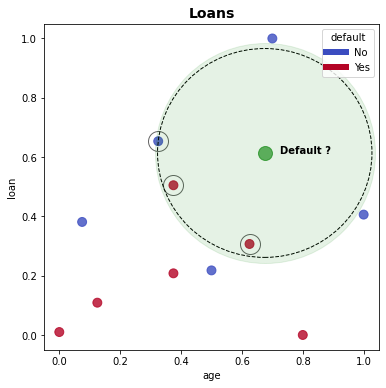
# criando os dados normalizados, 'scaled'  
loans\_scaled = pd.DataFrame(scaler.transform(loans[['age','loan']]),columns=loans.columns[0:2])  
loans\_scaled = pd.concat([loans\_scaled,loans[['default']]],axis=1)  
display(loans\_scaled)  
  
case\_scaled = pd.DataFrame(scaler.transform(case),columns=case.columns[0:2])  
display(case\_scaled)

age loan default  
0 0.125 0.108911 1  
1 0.375 0.207921 1  
2 0.625 0.306931 1  
3 0.000 0.009901 1  
4 0.375 0.504950 1  
5 0.800 0.000000 1  
6 0.075 0.381188 0  
7 0.500 0.217822 0  
8 1.000 0.405941 0  
9 0.700 1.000000 0  
10 0.325 0.653465 0

age loan  
0 0.675 0.613861

Podemos agora refazer o cálculo das distância empregando os dados normalizados e identificar os k-vizinhos mais próximos e a classe mais frenquente dentre eles.

dist = DistanceMetric.get\_metric('euclidean')  
d = dist.pairwise(loans\_scaled[['age','loan']],case\_scaled[['age','loan']])  
  
loans\_scaled['distance'] = d  
  
k3\_neighbors\_scaled = loans\_scaled.nsmallest(3,'distance')  
  
f, ax = plt.subplots(1,1,figsize=(6,6))  
cmap\_data = plt.cm.coolwarm   
  
ax.plot(case\_scaled.age, case\_scaled.loan, 'go', markersize=220, alpha=0.10)  
ax.scatter(k3\_neighbors\_scaled.age, k3\_neighbors\_scaled.loan, edgecolors='k', c='w', alpha=0.6, s=400)  
ax.scatter(loans\_scaled.age, loans\_scaled.loan, c=loans\_scaled.default, cmap=cmap\_data, alpha=0.8, s=80)  
ax.plot(case\_scaled.age, case\_scaled.loan, 'go', markersize=14, alpha=0.6)  
  
a\_circle = plt.Circle((case\_scaled.age, case\_scaled.loan), k3\_neighbors\_scaled.distance.max(), edgecolor='k', linestyle='dashed',fill=False)  
ax.add\_artist(a\_circle)  
  
plt.text(case\_scaled.age+0.05, case\_scaled.loan,'Default ?', weight='bold')  
plt.title('Loans',weight='bold',fontsize=14)  
plt.xlabel("age")  
plt.ylabel("loan")  
  
custom\_lines = [Line2D([0], [0], color=cmap\_data(0.), lw=6),  
 Line2D([0], [0], color=cmap\_data(1.), lw=6)]  
plt.legend(custom\_lines, ['No', 'Yes'], loc='upper right',title='default')  
  
plt.show()  
  
display(k3\_neighbors\_scaled)



age loan default distance  
2 0.625 0.306931 1 0.310977  
4 0.375 0.504950 1 0.319158  
10 0.325 0.653465 0 0.352234

Como você pode ver o resultado agora é outro e a classe prevista do empréstimo será *Default Yes* (zeros), com probabilidade de e, portanto, não deveríamos conceder esse empréstimo.

# Knn scikit-learn

Vamos empregar agora o estimador KNeighborsClassifier do scikit-learn para o mesmo problema acima. O modo de uso do estimador é o mesmo que empregamos para o classificador logístico e será empregado em todos os demais modelos supervisionados que iremos implementar.

from sklearn import neighbors  
from sklearn.preprocessing import MinMaxScaler  
  
X = loans[['age','loan']]   
y = loans.default   
  
scaler = MinMaxScaler()  
scaler.fit(X)  
X = scaler.transform(X)   
case\_scaled = scaler.transform(case)  
  
clf = neighbors.KNeighborsClassifier(n\_neighbors = 3)  
  
clf.fit(X, y)   
  
y\_pred = clf.predict(case\_scaled)  
  
default\_pred = ['No','Yes'][y\_pred[0]]  
print('Default? ', default\_pred)

Default? Yes

## Empregando Valores Categóricos

A distância de vetores de características é uma forma de medirmos similiridade entre os dados e é bastante empregado em uma série de modelos. Atributos categóricos, não numéricos, são muitas vezes muito importantes tendo uma forte influência na predição da classe dos dados, mas o cálculo de uma função distância, como a distância euclidiana que empregamos aqui, requer que os atributos sejam numéricos. A solução, neste caso, é fazermos o *encode* dos dados, transformando-os para numéricos.

Vamos alterar nosso exemplo anterior, incluindo um atributo de duração do empréstimo, e entender como podemos aplicar o modelo Knn também com atributos categóricos.

loans['Duration'] = ['Short','Long','Short','Undefined','Long','Short','Long','Short','Undefined','Long','Short']   
  
case['Duration'] = ['Short']   
  
display(loans)  
display(case)

age loan default distance distance\_loans Duration  
0 25 40000 1 102000.002373 102000.0 Short  
1 35 60000 1 82000.000878 82000.0 Long  
2 45 80000 1 62000.000032 62000.0 Short  
3 20 20000 1 122000.002988 122000.0 Undefined  
4 35 120000 1 22000.003273 22000.0 Long  
5 52 18000 1 124000.000101 124000.0 Short  
6 23 95000 0 47000.006128 47000.0 Long  
7 40 62000 0 80000.000306 80000.0 Short  
8 60 100000 0 42000.002012 42000.0 Undefined  
9 48 220000 0 78000.000006 78000.0 Long  
10 33 150000 0 8000.012250 8000.0 Short

age loan Duration  
0 47 142000 Short

Para transformar o atributo categórico vamos fazer o *hot encode* do atributo Duration. Essa técnica consiste em criarmos novos atributos binários (numéricos, 0 ou 1) para cada categoria dos dados e associar o valor 1 para categoria verdadeira.

*O* ***Hot Encode*** *(também Dummy Encode ou ainda One Code) é uma transformação importante e a melhor forma transformar dados categóricos para numéricos, e preferível a empregar o* ***Label Encode*** *(isto é, associar valores ordinais como 0,1,2,3... para cada categoria).*

A função get\_dummies() do Pandas é um meio simples e bastante comum para fazermos o hot encode de dados.

pd.get\_dummies(loans,prefix='Duration')

age loan default ... Duration\_Long Duration\_Short Duration\_Undefined  
0 25 40000 1 ... 0 1 0  
1 35 60000 1 ... 1 0 0  
2 45 80000 1 ... 0 1 0  
3 20 20000 1 ... 0 0 1  
4 35 120000 1 ... 1 0 0  
5 52 18000 1 ... 0 1 0  
6 23 95000 0 ... 1 0 0  
7 40 62000 0 ... 0 1 0  
8 60 100000 0 ... 0 0 1  
9 48 220000 0 ... 1 0 0  
10 33 150000 0 ... 0 1 0  
  
[11 rows x 8 columns]

Mas, novamente como na normalização, o uso de uma função para esse tipo de transformação apresenta dificuldade para a aplicação em novos conjuntos de dados.

pd.get\_dummies(case,prefix='Duration')

age loan Duration\_Short  
0 47 142000 1

É preferível, assim, também empregarmos um estimador neste caso.

from sklearn.preprocessing import OneHotEncoder, LabelEncoder  
  
hot\_encode = OneHotEncoder(handle\_unknown='ignore')   
hot\_encode = hot\_encode.fit(loans[['Duration']])  
  
display(hot\_encode.categories\_)  
  
display( pd.DataFrame(hot\_encode.transform(loans[['Duration']] ).toarray(), columns = hot\_encode.categories\_) )  
display( pd.DataFrame(hot\_encode.transform(case[['Duration']] ).toarray(), columns = hot\_encode.categories\_) )

[array(['Long', 'Short', 'Undefined'], dtype=object)]

Long Short Undefined  
0 0.0 1.0 0.0  
1 1.0 0.0 0.0  
2 0.0 1.0 0.0  
3 0.0 0.0 1.0  
4 1.0 0.0 0.0  
5 0.0 1.0 0.0  
6 1.0 0.0 0.0  
7 0.0 1.0 0.0  
8 0.0 0.0 1.0  
9 1.0 0.0 0.0  
10 0.0 1.0 0.0

Long Short Undefined  
0 0.0 1.0 0.0

Podemos, então, construir um exemplo completo do uso do modelo Knn para a predição do empréstimo, incluindo a normalização e o hot encode de dados.

from sklearn import neighbors  
from sklearn.preprocessing import MinMaxScaler  
from sklearn.preprocessing import OneHotEncoder, LabelEncoder  
  
X = loans.drop(columns=['default','distance','distance\_loans'])   
y = loans.default   
  
hot\_encode = OneHotEncoder(handle\_unknown='ignore')   
hot\_encode = hot\_encode.fit(X[['Duration']])  
X = pd.concat([ X, pd.DataFrame(hot\_encode.transform(loans[['Duration']] ).toarray(), columns = hot\_encode.categories\_) ], axis=1 )  
X = X.drop(columns=['Duration'])  
case = pd.concat([ case, pd.DataFrame(hot\_encode.transform(case[['Duration']] ).toarray(), columns = hot\_encode.categories\_) ], axis=1 )  
case = case.drop(columns=['Duration'])  
  
display(X)  
display(case)  
  
scaler = MinMaxScaler()  
scaler.fit(X)  
X = scaler.transform(X)   
case\_scaled = scaler.transform(case)  
  
clf = neighbors.KNeighborsClassifier(n\_neighbors = 3)  
  
clf.fit(X, y)   
  
y\_pred = clf.predict(case\_scaled)  
  
default\_pred = ['No','Yes'][y\_pred[0]]  
print('Default? ', default\_pred)

age loan (Long,) (Short,) (Undefined,)  
0 25 40000 0.0 1.0 0.0  
1 35 60000 1.0 0.0 0.0  
2 45 80000 0.0 1.0 0.0  
3 20 20000 0.0 0.0 1.0  
4 35 120000 1.0 0.0 0.0  
5 52 18000 0.0 1.0 0.0  
6 23 95000 1.0 0.0 0.0  
7 40 62000 0.0 1.0 0.0  
8 60 100000 0.0 0.0 1.0  
9 48 220000 1.0 0.0 0.0  
10 33 150000 0.0 1.0 0.0

age loan (Long,) (Short,) (Undefined,)  
0 47 142000 0.0 1.0 0.0

Default? No

# CASO: Breast Cancer biopsy

Este banco de dados de câncer de mama foi obtido dos Hospitais da Universidade de Wisconsin, que avaliou biópsias de tumores de mama em 699 pacientes até 1992;. Cada um dos nove atributos foi pontuado em uma escala de 1 a 10.

Fonte: <https://vincentarelbundock.github.io/Rdatasets/csv/MASS/biopsy.csv>

Documentação: <https://stat.ethz.ch/R-manual/R-devel/library/MASS/html/biopsy.html>

df = pd.read\_csv('https://vincentarelbundock.github.io/Rdatasets/csv/MASS/biopsy.csv',index\_col=0)  
df.head()

ID V1 V2 V3 V4 V5 V6 V7 V8 V9 class  
1 1000025 5 1 1 1 2 1.0 3 1 1 benign  
2 1002945 5 4 4 5 7 10.0 3 2 1 benign  
3 1015425 3 1 1 1 2 2.0 3 1 1 benign  
4 1016277 6 8 8 1 3 4.0 3 7 1 benign  
5 1017023 4 1 1 3 2 1.0 3 1 1 benign

## Preparação dos Dados

As variáveis preditoras são todas numéricas e não há, portanto, necessidade do hot encode dos atributos. Há entretanto valores ausentes.

df.isnull().sum() / len(df)

ID 0.00000  
V1 0.00000  
V2 0.00000  
V3 0.00000  
V4 0.00000  
V5 0.00000  
V6 0.02289  
V7 0.00000  
V8 0.00000  
V9 0.00000  
class 0.00000  
dtype: float64

O percentual é pequeno e assim vamos simplesmente excluir os dados ausentes.

df['V6'] = df[['V6']].fillna(df['V6'].mean())  
df.isnull().sum() / len(df)

ID 0.0  
V1 0.0  
V2 0.0  
V3 0.0  
V4 0.0  
V5 0.0  
V6 0.0  
V7 0.0  
V8 0.0  
V9 0.0  
class 0.0  
dtype: float64

## Aplicando o Modelo Knn

No nosso exemplo anterior, não fizemos separação de conjuntos de treinamento e teste, afinal era um exemplo simples apenas para entendermos os princípios do modelo.

Vamos agora implementar o modelo empregando o esquema mais geral de aprendizado supervisionado incluindo o uso dos conjuntos de treinamento e teste, e as avaliações do modelo.

from sklearn.model\_selection import train\_test\_split  
from sklearn import neighbors  
from sklearn.preprocessing import MinMaxScaler  
from sklearn.preprocessing import OneHotEncoder, LabelEncoder  
  
X = df.drop(columns=['ID','class'])  
y = df['class']  
  
scaler = MinMaxScaler()  
scaler.fit(X)  
X = scaler.transform(X)   
  
X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, stratify=y, test\_size=0.3, random\_state=123)  
  
clf = neighbors.KNeighborsClassifier(n\_neighbors = 3)  
  
clf.fit(X\_train, y\_train)   
  
y\_pred = clf.predict(X\_test)  
  
print( y\_pred[0:10], '...' )  
print( clf.score(X,y) )

['benign' 'benign' 'malignant' 'benign' 'benign' 'malignant' 'benign'  
 'benign' 'benign' 'benign'] ...  
0.9699570815450643

## Predição de Novos Casos

Se considerarmos 0.96 um bom resultado podemos então aplicar o modelo para novos casos. Por exemplo, podemos fazer predição considerando pacientes hipotéticos valores das medidas v1-v9 dos tumores nos percentis .

X\_new = pd.DataFrame( df.drop(columns=['ID','class']).quantile([0.10, 0.25, 0.75, 0.90]) ).reset\_index(drop=True)  
display(X\_new)  
X\_new\_scaled = scaler.transform(X\_new)

V1 V2 V3 V4 V5 V6 V7 V8 V9  
0 1.0 1.0 1.0 1.0 2.0 1.0 1.0 1.0 1.0  
1 2.0 1.0 1.0 1.0 2.0 1.0 2.0 1.0 1.0  
2 6.0 5.0 5.0 4.0 4.0 5.0 5.0 4.0 1.0  
3 9.0 9.0 8.0 8.0 6.0 10.0 7.0 9.0 3.0

clf.predict(X\_new\_scaled)

array(['benign', 'benign', 'malignant', 'malignant'], dtype=object)

X\_new['predicted'] = clf.predict(X\_new\_scaled)  
X\_new

V1 V2 V3 V4 V5 V6 V7 V8 V9 predicted  
0 1.0 1.0 1.0 1.0 2.0 1.0 1.0 1.0 1.0 benign  
1 2.0 1.0 1.0 1.0 2.0 1.0 2.0 1.0 1.0 benign  
2 6.0 5.0 5.0 4.0 4.0 5.0 5.0 4.0 1.0 malignant  
3 9.0 9.0 8.0 8.0 6.0 10.0 7.0 9.0 3.0 malignant

# Métricas de Distância

Para medir a distância entre vetores característicos empregamos até aqui a distância Euclidiana. Ela é a distância mais comum, mas existem várias funções distância que podem ser aplicadas, não só aqui mas também em outros modelos (*knn*, *kmeans* etc.). Embora a distância euclidiana pareça ser a mais aplicada, outras funções distância encontram mais uso em contextos específicos como a distância de **Hamming** para dados binários e cadeias de strings ou a distância **Coseno** para análise de dados de linguagem natural, textos e documentos.

Uma função é uma função distância se atende 4 propriedades:

## Algumas funções distância comuns

Distância Euclidiana

Distância Euclidiana Quadrática

Distância de Manhattan

Distância Máxima

Distância Minkowski

### Distância de Hamming para Strings

A distância de Hamming tem aplicação para cadeias de símbolos de mesmo comprimento e, por exemplo, pode ser empregada para medir a distância entre cadeias de DNA.

def hamming\_distance(string1, string2):  
 dist\_counter = 0  
 for n in range(len(string1)):  
 if string1[n] != string2[n]:  
 dist\_counter += 1  
 return dist\_counter  
  
hamming\_distance('ACGTACGT','ACGTTACG')

4

def hamming\_distance2(string1, string2):  
 return sum(xi != yi for xi, yi in zip(string1, string2))  
  
hamming\_distance2('ACGTACGT','ACGTTACG')

4

### Distância Cosseno

A Distância Cosseno encontra aplicação no tratamento de textos em que os textos têm uma representação vetorial baseada na frequência dos termos (ou palavras) como as representações *bow*, *tf-idf* etc. Ela mede o ângulo formado pelos vetores e a distância de dois vetores e pode ser obtida a partir da expressão:

Definimos a *similaridade* de dois vetores como:

E a distância:

O exemplo a seguir ilustra o uso dessa forma de distância para a busca de documentos similares e compara os resultados com o uso da distância euclidiana.

# apenas como empregar as funções cosine\_similarity e cosine\_distances  
from sklearn.metrics.pairwise import cosine\_distances, cosine\_similarity  
  
print( cosine\_similarity([[0,1,1]],[[0,2,2]]) )  
print( cosine\_distances([[0,1,1]],[[0,2,2]]) )

[[1.]]  
[[2.22044605e-16]]

Textos com representação vetorial BOW (Bag of Words), o valor na matriz sendo a quantidade de vezes que o termo aparece no documento

dfd = pd.read\_excel('http://meusite.mackenzie.br/rogerio/FCI22020/TFIDFExample2.xlsx',  
 skiprows=range(0,3),nrows=10,index\_col=0,usecols=range(0,2),header=None).reset\_index()  
dfd.columns = ['tf(i,j)','Text']   
df = pd.read\_excel('http://meusite.mackenzie.br/rogerio/FCI22020/TFIDFExample2.xlsx',  
 skiprows=range(0,16),nrows=9,index\_col=0,usecols=range(0,14))  
df[['Text']] = dfd[['Text']]  
display(df)

tf(i,j) system ... time Text  
1 d1 0 ... 0 Human machine interface for ABC computer appli...  
2 d2 1 ... 1 A survey of user opinion of computer system re...  
3 d3 1 ... 0 The EPS user interface management system.  
4 d4 2 ... 0 System and human system engineering testing in...  
5 d5 0 ... 1 Relation to user perceived response time to er...  
6 d6 0 ... 0 The generation of random, binary, ordered trees.  
7 d7 0 ... 0 The intersection graph of paths in trees.  
8 d8 0 ... 0 Graph minors IV: Widths of trees and well-quas...  
9 d9 0 ... 0 Graph minors: A survey.  
  
[9 rows x 14 columns]

Abaixo, as distâncias coseno e euclidiana dos documentos.

from sklearn.metrics.pairwise import cosine\_distances, cosine\_similarity  
  
docs = df.drop(columns=['tf(i,j)','Text'])  
doc = df[df['tf(i,j)'] == 'd1'].drop(columns=['tf(i,j)','Text'])  
  
print( cosine\_distances(docs, doc) )

[[0. ]  
 [0.76429774]  
 [0.71132487]  
 [0.76429774]  
 [1. ]  
 [1. ]  
 [1. ]  
 [1. ]  
 [1. ]]

dist = DistanceMetric.get\_metric('euclidean')  
print( dist.pairwise(docs, doc) )

[[0. ]  
 [2.64575131]  
 [2.23606798]  
 [2.64575131]  
 [2.44948974]  
 [2. ]  
 [2.23606798]  
 [2.44948974]  
 [2.44948974]]

*Embora não possamos desenvolver aqui os vários tópicos que envolvem o processamento de textos e de linguagem natural, o exemplo acima permite entender que podemos aplicar as mesmas técnicas de aprendizado supervisionado que aplicamos até aqui em dados tabulares, em textos e documentos a partir de uma representação vetorial desses elementos (e o mesmo é válido para outros modelos). Em outros termos, podemos aplicar técnicas como Knn, Regressão Logística, e mesmo outras que veremos a seguir, para classificar também documentos, e-mails ou páginas de um site.*

# Seleção de Hiperparâmetros

Em todos exemplos anteriores empregamos o modelo Knn com k=3 e a função de distância euclidiana. Será que haveriam *hiperparâmetros* melhores com desempenho melhor?

A escolha de melhores *hiperparâmetros* é em geral por experimentação uma vez que não existem hiperparâmetros melhores *apriori* para quaisquer conjuntos de dados. A ideia, então, é criarmos os diferentes modelos e avaliarmos o desempenho de cada um para obtermos os melhores hiperparâmetros.

Podemos então adaptar o nosso código do modelo Knn anterior para, por exemplo, variar os hiperparâmetros k, no range de valores de 2 a 10 e experimentar o resultado das funções distância 'euclidean' e 'manhattan'. manhattan

from sklearn.model\_selection import train\_test\_split  
from sklearn import neighbors  
from sklearn.preprocessing import MinMaxScaler  
from sklearn.preprocessing import OneHotEncoder, LabelEncoder  
  
df = pd.read\_csv('https://vincentarelbundock.github.io/Rdatasets/csv/MASS/biopsy.csv',index\_col=0)  
df['V6'] = df[['V6']].fillna(df['V6'].mean())  
  
X = df.drop(columns=['ID','class'])  
y = df['class']  
  
scaler.fit(X)  
X = scaler.transform(X)   
  
X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, stratify=y, test\_size=0.3, random\_state=123)  
  
for k, d in [(k,d) for k in range(2,11) for d in ['euclidean','manhattan']]:  
  
 clf = neighbors.KNeighborsClassifier(n\_neighbors = k, metric= d )  
  
 clf.fit(X\_train, y\_train)   
  
 y\_pred = clf.predict(X\_test)  
  
 print( k, d, np.round( clf.score(X\_test,y\_test), 4) )

2 euclidean 0.9095  
2 manhattan 0.9286  
3 euclidean 0.9429  
3 manhattan 0.9381  
4 euclidean 0.9381  
4 manhattan 0.9333  
5 euclidean 0.9429  
5 manhattan 0.9333  
6 euclidean 0.9381  
6 manhattan 0.9286  
7 euclidean 0.9429  
7 manhattan 0.9333  
8 euclidean 0.9381  
8 manhattan 0.9333  
9 euclidean 0.9381  
9 manhattan 0.9381  
10 euclidean 0.9381  
10 manhattan 0.9429

Embora tendo escolhido os conjuntos de treinamento e teste de forma aleatória o resultado acima, pode depender do par (treinamento, teste) escolhido. Para não considerarmos o resultado de uma única amostra, podemos fazer várias execuções a obter a média dos valores sobre várias amostras, o que será uma medida dependente de um par específico de dados e uma melhor aproximação do resultado esperado do modelo.

scores\_means = {}  
  
for k, d in [(k,d) for k in range(2,9) for d in ['euclidean','manhattan']]:  
   
 scores = []  
 for i in range(0,25):  
  
 X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, stratify=y, test\_size=0.3, random\_state=123)  
  
 clf = neighbors.KNeighborsClassifier(n\_neighbors = k, metric= d )  
  
 clf.fit(X\_train, y\_train)   
  
 y\_pred = clf.predict(X\_test)  
   
 scores.append(clf.score(X\_test,y\_test))  
  
 scores\_means[(k,d)] = np.mean(scores)  
   
scores\_means = pd.DataFrame(scores\_means.values(), index=scores\_means.keys()).reset\_index()  
scores\_means.columns = ['k','metric','score']  
  
display(scores\_means)  
print('\nBest result:\n')  
display(scores\_means.nlargest(1,'score'))

k metric score  
0 2 euclidean 0.909524  
1 2 manhattan 0.928571  
2 3 euclidean 0.942857  
3 3 manhattan 0.938095  
4 4 euclidean 0.938095  
5 4 manhattan 0.933333  
6 5 euclidean 0.942857  
7 5 manhattan 0.933333  
8 6 euclidean 0.938095  
9 6 manhattan 0.928571  
10 7 euclidean 0.942857  
11 7 manhattan 0.933333  
12 8 euclidean 0.938095  
13 8 manhattan 0.933333

Best result:

k metric score  
2 3 euclidean 0.942857

Desse modo, concluímos que para o nosso conjunto de dados os melhores resultados com o modelo knn são obtidos com os parâmetros acima.

# Conjuntos de Validação e Teste

Ao avaliar diferentes hiperparâmetros para os estimadores, como o valor de k e a métrica do estimador KNeighborsClassifier, traz entretanto o risco de overfitting no conjunto de teste, porque os hiperparâmetros podem ser ajustados *até que* o estimador tenha o desempenho ideal e, do mesmo modo que antes para o conjunto de treinamento, o conhecimento sobre o conjunto de teste pode *vazar* para o modelo e as métricas de avaliação mascarando o desempenho de generalização do modelo.

Deveríamos assim separar novamente uma outra parte do conjunto de dados, um *conjunto de validação* que seria empregado para validar o modelo durante a seleção dos hiperparâmetros: o treinamento, com diferentes hiperparâmetros, continua no conjunto de treinamento; em seguida a avaliação é feita no conjunto de validação; após os experimentos sucedidos, a avaliação final pode ser feita no conjunto de teste.

A implementação do nosso modelo seria então a seguinte:

scores\_means = {}  
  
X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, stratify=y, test\_size=0.3, random\_state=123)  
  
for k, d in [(k,d) for k in range(2,9) for d in ['euclidean','manhattan']]:  
   
 scores = []  
 for i in range(0,25):  
  
 X\_train\_val, X\_val, y\_train\_val, y\_val = train\_test\_split(X\_train, y\_train, stratify=y\_train, test\_size=0.2, random\_state=123)  
  
 clf = neighbors.KNeighborsClassifier(n\_neighbors = k, metric= d )  
  
 clf.fit(X\_train\_val, y\_train\_val)   
  
 y\_pred = clf.predict(X\_val)  
   
 scores.append(clf.score(X\_val,y\_val))  
  
 scores\_means[(k,d)] = np.mean(scores)  
   
scores\_means = pd.DataFrame(scores\_means.values(), index=scores\_means.keys()).reset\_index()  
scores\_means.columns = ['k','metric','score']  
  
display(scores\_means)  
print('\nBest result:\n')  
display(scores\_means.nlargest(1,'score'))

k metric score  
0 2 euclidean 0.928571  
1 2 manhattan 0.948980  
2 3 euclidean 0.948980  
3 3 manhattan 0.948980  
4 4 euclidean 0.948980  
5 4 manhattan 0.938776  
6 5 euclidean 0.959184  
7 5 manhattan 0.959184  
8 6 euclidean 0.959184  
9 6 manhattan 0.959184  
10 7 euclidean 0.959184  
11 7 manhattan 0.959184  
12 8 euclidean 0.959184  
13 8 manhattan 0.948980

Best result:

k metric score  
6 5 euclidean 0.959184

E podemos agora verificar o resultado desse melhor modelo no conjunto de teste para a avaliação final.

k = scores\_means.nlargest(1,'score').k.values[0]  
metric = scores\_means.nlargest(1,'score').metric.values[0]  
  
clf = neighbors.KNeighborsClassifier(n\_neighbors = k, metric= metric )  
print(clf)  
   
clf.fit(X\_train, y\_train)   
  
print("accuracy: %0.3f" % clf.score(X\_test,y\_test),'\n')

KNeighborsClassifier(metric='euclidean')  
accuracy: 0.943

Esperamos com esse modelo obter um resultado de 0.93 de acuracidade e esse é um valor metodologicamente mais correto que as estimativas anteriores.

# Cross Validation

Mas ao particionar os dados disponíveis em três conjuntos, reduzimos drasticamente o número de amostras que podem ser usadas para aprender o modelo.

Uma solução para este problema, e também para buscarmos um resultado menos dependente de uma escolha aleatória particular do par (treino, validação), é uma técnica denominada validação cruzada ou cross validation (CV). Um conjunto de teste ainda deve ser apresentado para avaliação final, mas o conjunto de validação não é mais necessário ao empregar o CV. Em sua forma mais simples, denominada *k-fold CV*, o conjunto de treinamento é dividido em k partições aleatórias menores e o treinamento ocorre do seguinte modo:

Para as partições:

1. O modelo é treinado empregando-se partições dos dados de treinamento;
2. O modelo resultante é validado na parte restante dos dados, isto é, é usado como conjunto de teste para calcular a métrica de desempenho, por exemplo a acuracidade ou a precisão.

A medida de desempenho do CV é, então, a média dos valores calculados para as medidas. Embora computacionalmente cara, o CV não desperdiça e é um procedimento mais sistemático que garante que todos os dados serão testados pelo modelo em algum momento.



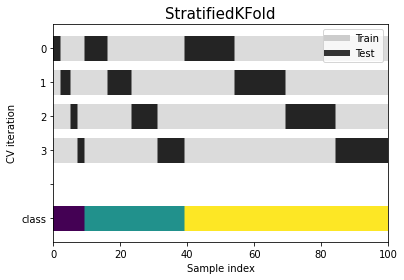
Figura 1. Esquema de Cross-Validation. (Fonte: <https://scikit-learn.org>)

O scikit-learn implementa diferentes tipos de CV que variam basicamente na forma com são selecionadas as partições. O KFold e o StratifiedKFold são as formas mais comuns e suas partições são exemplificadas abaixo.

# you can skip this code!  
  
# Código livremente adaptado de: https://scikit-learn.org/stable/auto\_examples/model\_selection/plot\_cv\_indices.html  
#  
cmap\_data = plt.cm.viridis   
cmap\_cv = plt.cm.binary  
  
def plot\_cv\_indices(cv, X, y, group, ax, n\_splits, lw=25):  
 """Create a sample plot for indices of a cross-validation object."""  
  
 # Generate the training/testing visualizations for each CV split  
 for ii, (tr, tt) in enumerate(cv.split(X=X, y=y, groups=group)):  
 # Fill in indices with the training/test groups  
 indices = np.array([np.nan] \* len(X))  
 indices[tt] = 1  
 indices[tr] = 0  
  
 # Visualize the results  
 ax.scatter(  
 range(len(indices)),  
 [ii + 0.5] \* len(indices),  
 c=indices,  
 marker="\_",  
 lw=lw,  
 cmap=cmap\_cv,  
 vmin=-0.2,  
 vmax=1.2,  
 )  
  
 custom\_lines = [Line2D([0], [0], color=cmap\_cv(0.2), lw=6),  
 Line2D([0], [0], color=cmap\_cv(0.8), lw=6)]  
 ax.legend(custom\_lines, ['Train', 'Test'], loc='upper right')  
  
 # Plot the data classes and groups at the end  
 ax.scatter(  
 range(len(X)), [ii + 2.5] \* len(X), c=y, marker="\_", lw=lw, cmap=cmap\_data  
 )  
  
 # Formatting  
 yticklabels = list(range(n\_splits)) + ["", "class"]  
 ax.set(  
 yticks=np.arange(n\_splits + 2) + 0.5,  
 yticklabels=yticklabels,  
 xlabel="Sample index",  
 ylabel="CV iteration",  
 ylim=[n\_splits + 2.2, -0.2],  
 xlim=[0, 100],  
 )  
 ax.set\_title("{}".format(type(cv).\_\_name\_\_), fontsize=15)  
 return ax

# you can skip this code!  
  
from sklearn.model\_selection import KFold, StratifiedKFold  
np.random.seed(1338)  
  
n\_splits = 4  
  
n\_points = 100  
X = np.random.randn(100, 10)  
  
percentiles\_classes = [0.1, 0.3, 0.6]  
y = np.hstack([[ii] \* int(100 \* perc) for ii, perc in enumerate(percentiles\_classes)])  
  
# Evenly spaced groups repeated once  
groups = np.hstack([[ii] \* 10 for ii in range(10)])  
  
fig, ax = plt.subplots()  
cv = KFold(n\_splits)  
plot\_cv\_indices(cv, X, y, groups, ax, n\_splits)  
plt.show()  
  
fig, ax = plt.subplots()  
cv = StratifiedKFold(n\_splits)  
plot\_cv\_indices(cv, X, y, groups, ax, n\_splits)  
plt.show()





Note que as faixas correspodem a índices aleatórios e não dados contíguos.

## Aplicando o CV

A forma mais simples de empregar o CV é aplicar a função cross\_val\_score() sobre o estimador e o conjunto de dados. Ela por padrão irá retornar o *score* do estimador em uma estratégia de StratifiedKFold, mas outras métricas podem ser obtidas.

from sklearn.model\_selection import cross\_val\_score  
  
df = pd.read\_csv('https://vincentarelbundock.github.io/Rdatasets/csv/MASS/biopsy.csv',index\_col=0)  
df['V6'] = df[['V6']].fillna(df['V6'].mean())  
  
X = df.drop(columns=['ID','class'])  
y = df['class']  
  
scaler.fit(X)  
X = scaler.transform(X)   
  
X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, stratify=y, test\_size=0.3, random\_state=123)  
  
clf = neighbors.KNeighborsClassifier(n\_neighbors = 3, metric= 'euclidean' )  
  
acc\_scores = cross\_val\_score(clf, X\_train, y\_train, cv = 10)  
  
print(acc\_scores, '\n')  
print("accuracy: %0.3f +/- %0.3f" % (acc\_scores.mean(), acc\_scores.std() \* 2),'\n')  
  
for metric in ['accuracy','f1\_macro','precision\_macro','recall\_macro']:  
 scores = cross\_val\_score(clf, X\_train, y\_train, cv = 4, scoring=metric)  
 print(metric + ": %0.3f +/- %0.3f" % (scores.mean(), scores.std() \* 2))

[0.97959184 0.93877551 1. 0.97959184 0.95918367 0.95918367  
 1. 0.93877551 0.97959184 0.95833333]   
  
accuracy: 0.969 +/- 0.042   
  
accuracy: 0.971 +/- 0.008  
f1\_macro: 0.968 +/- 0.009  
precision\_macro: 0.970 +/- 0.009  
recall\_macro: 0.967 +/- 0.018

Aplicando agora o cross\_val\_score() ao nosso exemplo na busca de melhores hiperparâmetros você pode obter:

from sklearn.model\_selection import train\_test\_split  
from sklearn import neighbors  
from sklearn.preprocessing import MinMaxScaler  
from sklearn.preprocessing import OneHotEncoder, LabelEncoder  
  
df = pd.read\_csv('https://vincentarelbundock.github.io/Rdatasets/csv/MASS/biopsy.csv',index\_col=0)  
df['V6'] = df[['V6']].fillna(df['V6'].mean())  
  
X = df.drop(columns=['ID','class'])  
y = df['class']  
  
scaler.fit(X)  
X = scaler.transform(X)   
  
X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, stratify=y, test\_size=0.3, random\_state=123)  
  
scores\_means = {}  
  
X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, stratify=y, test\_size=0.3, random\_state=123)  
  
for k, d in [(k,d) for k in range(2,9) for d in ['euclidean','manhattan']]:  
   
 clf = neighbors.KNeighborsClassifier(n\_neighbors = k, metric= d )  
  
 acc\_scores = cross\_val\_score(clf, X\_train, y\_train, cv = 5)   
  
 scores\_means[(k,d)] = acc\_scores.mean()  
   
scores\_means = pd.DataFrame(scores\_means.values(), index=scores\_means.keys()).reset\_index()  
scores\_means.columns = ['k','metric','score']  
  
# display(scores\_means)  
print('\nBest result:\n')  
display(scores\_means.nlargest(1,'score'))

Best result:

k metric score  
7 5 manhattan 0.975489

E pode agora aplicar o melhor modelo obtido:

k = scores\_means.nlargest(1,'score').k.values[0]  
metric = scores\_means.nlargest(1,'score').metric.values[0]  
  
clf = neighbors.KNeighborsClassifier(n\_neighbors = k, metric= metric )  
print(clf)  
   
clf.fit(X\_train, y\_train)   
   
print("accuracy: %0.3f" % clf.score(X\_test,y\_test),'\n')

KNeighborsClassifier(metric='manhattan')  
accuracy: 0.933

Você deve notar que esse procedimento, variando hiperparâmetros do estimador, poderia ser igualmente empregado para avaliar e selecionar diferentes estimadores e você, por exemplo, poderia alterar o código acima para avaliar o desempenho dos estimadores KNeighborsClassifier e LogisticRegression a fim de escolher o melhor classificador.

# Usando o Grid Search

O último refinamento que faremos no procedimento de busca de melhores hiperparâmetros consiste em empregarmos GridSearchCV() do scikit-learn. Essa função automatiza a busca de melhores hiperparâmetros que fizemos acima implementando de forma manual os diferentes estimadores para um espaço de hiperparâmetros em for k, d in [(k,d) for k in range(2,9) for d in ['euclidean','manhattan']]:... e podendo ser aplicada a qualquer estimador.

Qualquer parâmetro de um estimador pode ser otimizado desta maneira e para encontrar os nomes e valores dos parâmetros de um determinado estimador você pode empregar o método estimator.get\_params().

clf = neighbors.KNeighborsClassifier()  
clf.get\_params

<bound method BaseEstimator.get\_params of KNeighborsClassifier()>

e do mesmo modo os hiperparâmetros da regressão logística podem também ser obtidos:

# e do mesmo modo os hiperparâmetros da regressão logística são também obtidos  
from sklearn.linear\_model import LogisticRegression   
LogisticRegression().get\_params

<bound method BaseEstimator.get\_params of LogisticRegression()>

Duas abordagens para a pesquisa de hiperparâmetros são fornecidas no scikit-learn para os valores fornecedos pelo usuário. O GridSearchCV considera exaustivamente todas as combinações de parâmetros e o RandomizedSearchCV emprega uma amostra de candidatos de um espaço de hiperparâmetros com uma distribuição específica. Nos dois casos a aplicação é bastante direta e a função apenas precisa receber o classificador, os valores dos hiperparâmetros desejados e o CV a ser empregado.

from sklearn.model\_selection import train\_test\_split  
from sklearn import neighbors  
from sklearn.preprocessing import MinMaxScaler  
from sklearn.preprocessing import OneHotEncoder, LabelEncoder  
from sklearn.model\_selection import GridSearchCV  
from sklearn.metrics import classification\_report  
  
df = pd.read\_csv('https://vincentarelbundock.github.io/Rdatasets/csv/MASS/biopsy.csv',index\_col=0)  
df['V6'] = df[['V6']].fillna(df['V6'].mean())  
  
X = df.drop(columns=['ID','class'])  
y = df['class']  
  
scaler.fit(X)  
X = scaler.transform(X)   
  
X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, stratify=y, test\_size=0.3, random\_state=123)  
  
base\_estimator = neighbors.KNeighborsClassifier()  
param\_grid = {'n\_neighbors': [3,4,5,6,7,8,9,10], 'metric': ['euclidean','manhattan']}  
  
clf = GridSearchCV(base\_estimator, param\_grid, cv=5, scoring='accuracy')  
clf.fit(X\_train, y\_train)  
  
# print(clf.cv\_results\_)  
print(clf.best\_estimator\_)  
  
print()  
print("Detailed classification report:")  
print()  
y\_pred = clf.predict(X\_test)  
print(classification\_report(y\_test, y\_pred))  
print()

KNeighborsClassifier(metric='manhattan')  
  
Detailed classification report:  
  
 precision recall f1-score support  
  
 benign 0.94 0.96 0.95 138  
 malignant 0.91 0.89 0.90 72  
  
 accuracy 0.93 210  
 macro avg 0.93 0.92 0.93 210  
weighted avg 0.93 0.93 0.93 210

Por padrão as funções de pesquisa em grade, como o GridSearchCV() empregam o score padrão do estimador como função de pontuação (no caso de classificação é o accuracy), mas deixamos explícito o parâmetro pois você poderia querer empregar uma função de pontuação dos estimadores baseada em outra métrica.

Sendo o formato final do nosso procedimento também substituímos no código acima a exibição final do score que vínhamos empregando pelo classification\_report que exibe um resultado mais completo do modelo selecionado.

Em resumo, nosso procedimento de pesquisa de melhores hiperparâmetros de um estimador consiste em:

1. Selecionar um estimador (um classificador ou um regressor)
2. Definir um espaço de hiperparâmetros que desejamos avaliar
3. Definir uma função de pontuação (*score function*)
4. Empregar um esquema de validação cruzada

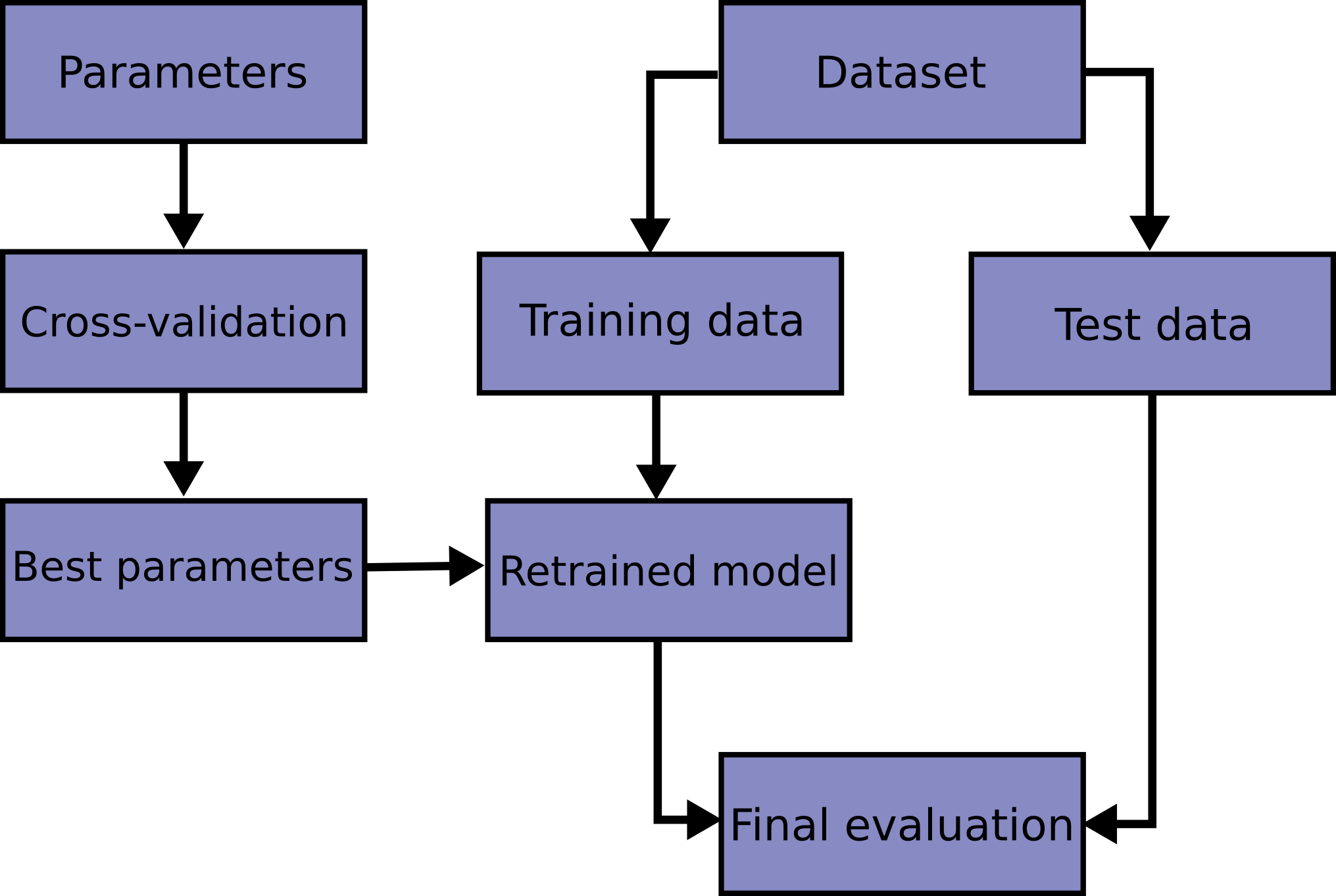


Figura 2. Esquema Geral da Avaliação de Modelos. (Fonte: <https://scikit-learn.org>)

# Síntese

Nesta trilha você aprendeu a empregar o modelo de KVizinhos mais Próximos para classificação. A partir de um problema inicial bastante simples e uma solução do zero você pode conhecer os princípios desse modelo. Em seguida, em uma sequência de passos, incrementamos nosso modelo para que você pudesse compreender vários conceito e técnicas importantes que podem ser aplicadas a estimadores e análises em geral:

* Na preparação dos dados
  + Normalização de Dados empregando estimadores
  + Encode de Dados empregando estimadores
* No desenvolvimento e avaliação dos modelos
  + Conjuntos de Validação e Teste
  + Validação Cruzada
  + Pesquisa em Grade para seleção de hiperparâmetros

Todos esses procedimentos, que aplicamos para o estimador específico do Knn, podem ser igualmente aplicados a quaisquer estimadores e o procedimento final dessa trilha fornece um modelo de código que você irá empregar em outros estimadores adiante.

# Para Saber Mais

* A documentação do scikit-learn é uma excelente fonte de consulta para todos as técnicas que empregamos aqui. Mas ela é também bastante extensa e selecionamos aqui alguns dos links principais que podem interessar:
  + **Cross-validation: evaluating estimator performance**, Disponível em: <https://scikit-learn.org/stable/modules/cross_validation.html#cross-validation>
  + **Tuning the hyper-parameters of an estimator**, Disponível em: <https://scikit-learn.org/stable/modules/grid_search.html#grid-search>
  + **Parameter estimation using grid search with cross-validation**, Disponível em: <https://scikit-learn.org/stable/auto_examples/model_selection/plot_grid_search_digits.html#sphx-glr-auto-examples-model-selection-plot-grid-search-digits-py>
* Mas para saber mais sobre esses procedimentos o melhor mesmo é praticar. Você pode empregar então o último código dessa trilha, que é um modelo final de código para seleção de hiperparâmetros de um classficador, para fazer o mesmo para o classificador logístico, um regressor linear ou ainda empregar um outro conjunto de dados de seu interesse. Essas são atividades que, de fato, irão ajudá-lo a saber mais sobre o que tratamos aqui.
* De um modo simples, dados são normalizados ou redimensionados para trazer todas as variáveis ​​em proporção umas com as outras. Existem diferentes formas de normalização e redimensionamento de dados e aqui aplicamos apenas uma delas. Acesse o artigo **When to perform a Feature Scaling?** de Raghav Vashisht, disponível em: <https://www.atoti.io/when-to-perform-a-feature-scaling/> para saber mais sobre isso e veja também **Compare the effect of different scalers on data with outliers** disponível em: <https://scikit-learn.org/stable/auto_examples/preprocessing/plot_all_scaling.html> para conhecer outros estimadores de normalização do scikit-learn.

# Referências

Sayad, Saed (2021). **An Introduction to Data Science**, Disponível em: <https://www.saedsayad.com/data_mining_map.htm> Acesso: 12 de Novembro de 2021.

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.

\_\_\_. **Cross-validation: evaluating estimator performance**, Disponível em: <https://scikit-learn.org/stable/modules/cross_validation.html#cross-validation> Acesso em: 13 de Novembro de 2021.

\_\_\_. **Tuning the hyper-parameters of an estimator**, Disponível em: <https://scikit-learn.org/stable/modules/grid_search.html#grid-search> Acesso em: 13 de Novembro de 2021.

\_\_\_. **Parameter estimation using grid search with cross-validation**, Disponível em: <https://scikit-learn.org/stable/auto_examples/model_selection/plot_grid_search_digits.html#sphx-glr-auto-examples-model-selection-plot-grid-search-digits-py> Acesso em: 13 de Novembro de 2021.