## 7. Seleção de Modelos

Nesta aula você vai aprender:

* Como selecionar melhores modelos de Classificação e Regressão
* Sobre Outros Estimadores importantes
* Como empregar modelos de aprendizado supervisionado para a Classificação de Imagens de Dígitos e Séries Temporais

Nesta aula vamos consolidar o que você aprendeu sobre modelos de aprendizado supervisionado trabalhando dois conjuntos de dados um pouco diferentes, e que darão uma visão mais ampla do que você pode fazer com esses modelos e do que você verá na sequência do seu curso em outras disciplinas.

Vamos fazer aqui modelos completos de aprendizado supervisionado, tanto para Classificação (predição de classes) como para Regressão (predição de valores), selecionando não só os **melhores hiperparâmetros dos modelos**, como também a **seleção de modelos que competem** entre si. Você também terá contato com modelos importantes, como **modelos neurais** e **regressores não lineares** que, embora não venhamos a detalhar aqui os seus princípios e funcionamento, você verá que podem ser facilmente aplicados com o que você aprendeu aqui.

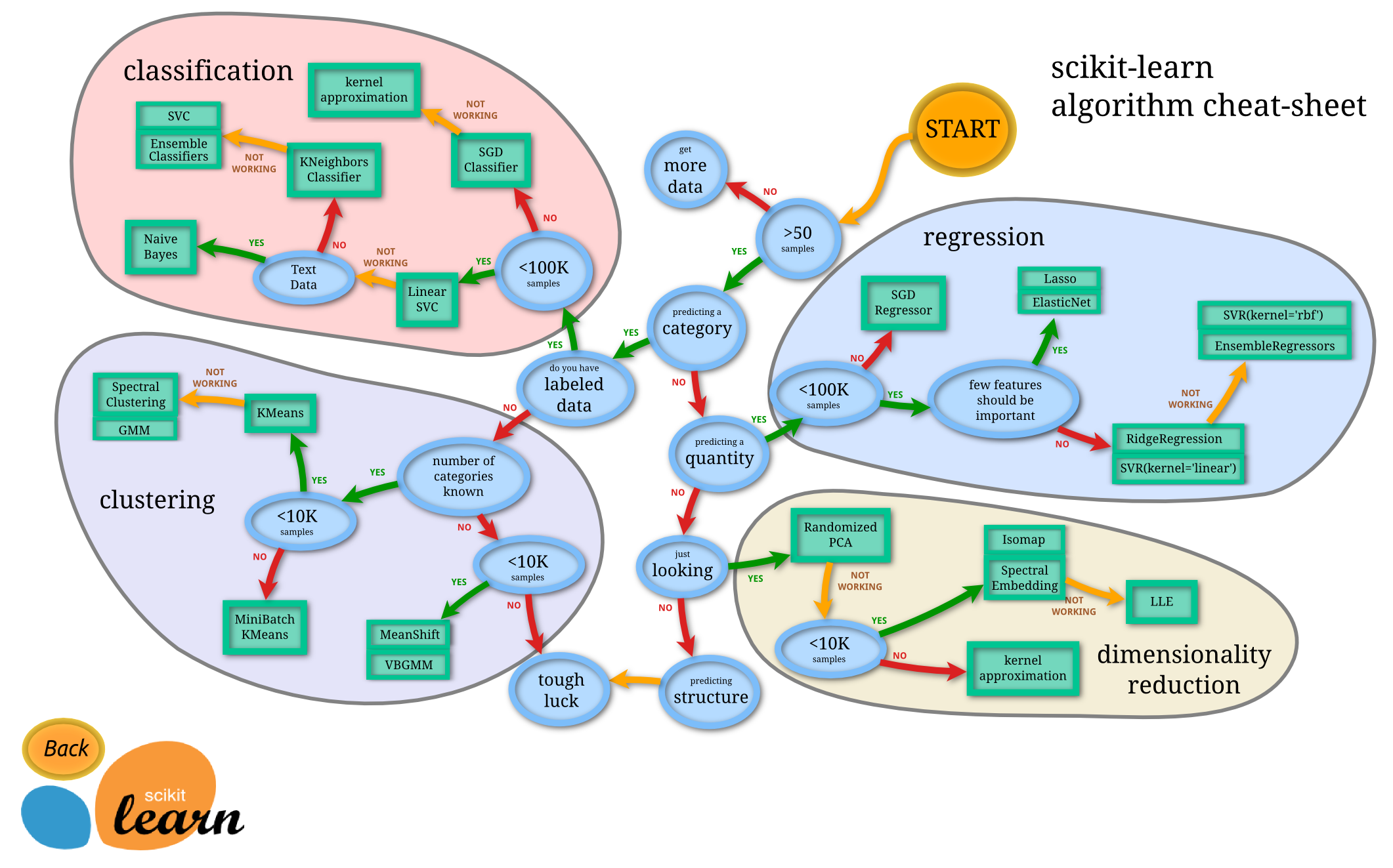
Ao final você terá uma receita de código completa para buscar melhores modelos tanto para problemas de classificação como de estimativa de valores.

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

# Selecionando Diferentes Modelos

Vamos começar entendendo como funciona a seleção de diferentes modelos e seus parâmetros para problemas de classificação em um exemplo de brinquedo, o nosso conhecido *moons*. Mais adiante aplicaremos os mesmos princípios para um problema de classificação de imagem de dígito e para regressores.

Diferentes modelos criam fronteiras de decisão diferentes para os mesmos conjuntos de dados. Eles empregam critérios diferentes e, por isso, produzem classificações de modo diferente o que torna difícil você escolher empregar um modelo ou outro, muitas vezes, essa é a parte mais difícil de resolver um problema de aprendizado de máquina: *qual o estimador certo para empregar?* Não há um melhor estimador *apriori* para quaisquer dados e estimadores diferentes são mais adequados para diferentes tipos de dados e problemas. O scikit-learn fornece em <https://scikit-learn.org/stable/tutorial/machine_learning_map/index.html> um diagrama, e que fornecemos abaixo, que funciona como um guia, embora aproximado, de como abordar problemas com relação a quais estimadores empregar.



**Figura 8. Diagrama do scikit-learn para seleção de modelos.**

(Fonte: <https://scikit-learn.org/>)

De qualquer modo uma abordagem comum e que você encontra em sistemas de *Auto ML*, consiste em adotarmos alguma métrica, como a acuracidade ou a precisão, ou múltiplas métricas, para selecionar diferentes modelos, experimentando cada um deles, muito à exemplo do que fizemos na aula anterior para a escolha dos melhores hiperparâmetros.

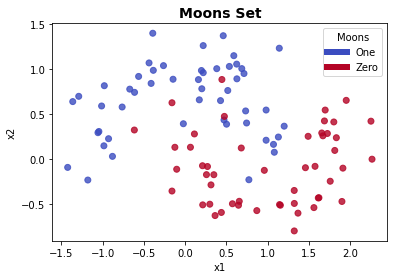
# Selecionando Classificadores

Vamos adaptar o código modelo final da aula anterior para explorar agora, não só diferentes hiperparâmetros, mas diferentes estimadores.

Abaixo os principais classificadores do scikit-learn e os que empregaremos, com (\*), para seleção de um estimador para o nosso conjunto de dados *moons*:

from sklearn.linear\_model import LogisticRegression # (\*)  
from sklearn.neighbors import KNeighborsClassifier # (\*)  
from sklearn.tree import DecisionTreeClassifier # (\*)  
from sklearn.model\_selection import KFold  
from sklearn.model\_selection import cross\_val\_score  
from sklearn.naive\_bayes import GaussianNB  
from sklearn.tree import DecisionTreeClassifier  
from sklearn.svm import SVC  
from sklearn.neural\_network import MLPClassifier  
from sklearn.ensemble import RandomForestClassifier  
from sklearn.ensemble import AdaBoostClassifier  
from sklearn.ensemble import GradientBoostingClassifier # (\*)

from sklearn.datasets import make\_moons  
cmap\_data = plt.cm.coolwarm   
  
X, y = make\_moons(n\_samples=100, noise=0.25, random\_state=1234)  
  
df = pd.DataFrame({'x1':X[:, 0], 'x2':X[:, 1], 'y':y})  
  
plt.scatter(df.x1, df.x2, c=df.y, cmap=cmap\_data, alpha=0.8)  
plt.title('Moons Set',weight='bold',fontsize=14)  
plt.xlabel("x1")  
plt.ylabel("x2")  
  
custom\_lines = [Line2D([0], [0], color=cmap\_data(0.), lw=6),  
 Line2D([0], [0], color=cmap\_data(1.), lw=6)]  
plt.legend(custom\_lines, ['One', 'Zero'], loc='upper right',title='Moons')  
  
plt.show()  
  
print(df.head())



x1 x2 y  
0 -0.763251 0.577354 0  
1 -0.391942 1.395641 0  
2 1.324561 -0.492319 1  
3 0.271295 -0.082532 1  
4 2.253887 0.420281 1

df.to\_csv('moons.csv',index=None)

def border(clf):  
 x\_min, x\_max = X['x1'].min() - .5, X['x1'].max() + .5  
 y\_min, y\_max = X['x2'].min() - .5, X['x2'].max() + .5  
 xx, yy = np.meshgrid(np.arange(x\_min, x\_max, 0.2),  
 np.arange(y\_min, y\_max, 0.2))  
  
 if hasattr(clf, "decision\_function"):  
 Z = clf.decision\_function(np.c\_[xx.ravel(), yy.ravel()])  
 else:  
 Z = clf.predict\_proba(np.c\_[xx.ravel(), yy.ravel()])[:, 1]  
  
 Z = Z.reshape(xx.shape)  
 plt.contourf(xx, yy, Z, cmap=cmap\_data, alpha=0.8)   
 plt.scatter(X['x1'], X['x2'], c=y, cmap=cmap\_data, alpha=0.8)  
 plt.title(str(clf)[0:str(clf).index('(')] + ' Moons Predictions',weight='bold',fontsize=14)  
 plt.xlabel("x1")  
 plt.ylabel("x2")  
  
 custom\_lines = [Line2D([0], [0], color=cmap\_data(0.), lw=6),  
 Line2D([0], [0], color=cmap\_data(1.), lw=6)]  
  
 plt.legend(custom\_lines, ['One', 'Zero'], loc='upper right',title='Moons')  
  
 plt.show()  
 return

Para obter os hiperparâmetros de cada modelo você pode empregar:

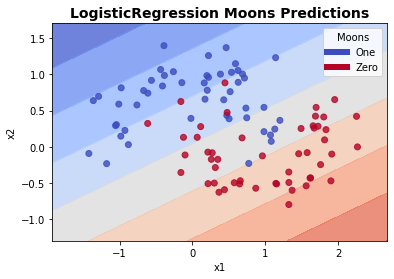
from sklearn.ensemble import GradientBoostingClassifier  
GradientBoostingClassifier().get\_params

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

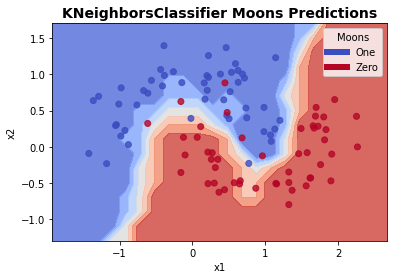
Abaixo um esquema geral de como avaliar os diferentes regressores com seus diferentes parâmetros e você poderá empregar outros se quiser.

from sklearn.linear\_model import LogisticRegression  
from sklearn.neighbors import KNeighborsClassifier  
from sklearn.tree import DecisionTreeClassifier  
from sklearn.model\_selection import KFold  
from sklearn.model\_selection import cross\_val\_score  
from sklearn.naive\_bayes import GaussianNB  
from sklearn.tree import DecisionTreeClassifier  
from sklearn.svm import SVC  
from sklearn.neural\_network import MLPClassifier  
from sklearn.ensemble import RandomForestClassifier  
from sklearn.ensemble import AdaBoostClassifier  
from sklearn.ensemble import GradientBoostingClassifier  
  
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  
  
X = df[['x1','x2']]  
y = df.y  
  
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(),  
 DecisionTreeClassifier(),  
 RandomForestClassifier(),  
 GradientBoostingClassifier()]  
   
   
param\_grids = [ {},  
 {'n\_neighbors': [3,4,5], 'metric': ['euclidean','manhattan']},  
 {'max\_depth': [2,3,4,5]},  
 {'n\_estimators':[3,4,5,6]},   
 {'n\_estimators':[3,4,5,6]}]   
   
for i in range(len(base\_estimators)):  
 clf = GridSearchCV(base\_estimators[i], param\_grids[i], cv=5, scoring='accuracy')  
 clf.fit(X\_train, y\_train)  
 # print(clf.cv\_results\_)  
 print(clf.best\_estimator\_)  
 border(clf.best\_estimator\_)  
 print()  
 print("Detailed classification report:")  
 print()  
 y\_pred = clf.predict(X\_test)  
 print(classification\_report(y\_test, y\_pred))  
 print()

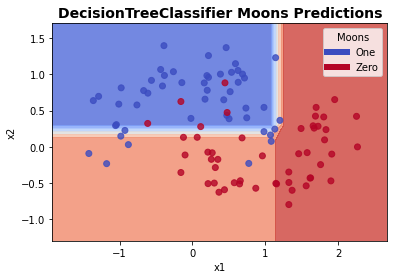
LogisticRegression()



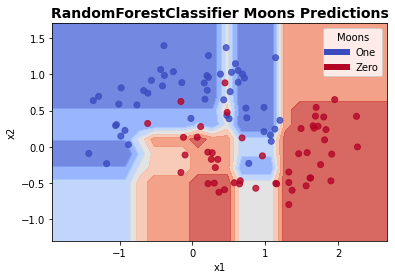
Detailed classification report:  
  
 precision recall f1-score support  
  
 0 0.93 0.93 0.93 15  
 1 0.93 0.93 0.93 15  
  
 accuracy 0.93 30  
 macro avg 0.93 0.93 0.93 30  
weighted avg 0.93 0.93 0.93 30  
  
  
KNeighborsClassifier(metric='euclidean', n\_neighbors=4)



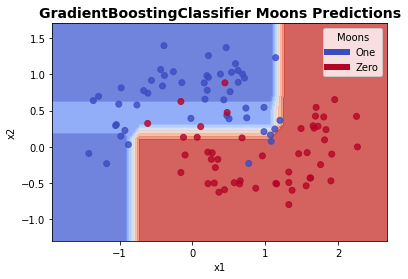
Detailed classification report:  
  
 precision recall f1-score support  
  
 0 0.83 1.00 0.91 15  
 1 1.00 0.80 0.89 15  
  
 accuracy 0.90 30  
 macro avg 0.92 0.90 0.90 30  
weighted avg 0.92 0.90 0.90 30  
  
  
DecisionTreeClassifier(max\_depth=2)



Detailed classification report:  
  
 precision recall f1-score support  
  
 0 0.92 0.80 0.86 15  
 1 0.82 0.93 0.87 15  
  
 accuracy 0.87 30  
 macro avg 0.87 0.87 0.87 30  
weighted avg 0.87 0.87 0.87 30  
  
  
RandomForestClassifier(n\_estimators=6)



Detailed classification report:  
  
 precision recall f1-score support  
  
 0 0.68 1.00 0.81 15  
 1 1.00 0.53 0.70 15  
  
 accuracy 0.77 30  
 macro avg 0.84 0.77 0.75 30  
weighted avg 0.84 0.77 0.75 30  
  
  
GradientBoostingClassifier(n\_estimators=3)



Detailed classification report:  
  
 precision recall f1-score support  
  
 0 0.93 0.93 0.93 15  
 1 0.93 0.93 0.93 15  
  
 accuracy 0.93 30  
 macro avg 0.93 0.93 0.93 30  
weighted avg 0.93 0.93 0.93 30

Analisando os dados os melhores resultados são os resultados dos modelos Gradiente Boosting e Regressão Logística e, neste caso, deveríamos empregar o modelo mais simples de Regressão Logística seguindo o princípio da parcimônia. Este, entretanto, é um exemplo de brinquedo, mas serve para mostrar que nem sempre modelos mais elaborados fornecerão a melhor predição dos dados.

# CASO: Classificando Imagens de Dígitos ()

Agora você pode aplicar as técnicas seleção de classificadores que aprendeu acima para um problema mais interessante. Vamos empregar isso para a classificação de imagens de dígitos escritos à mão, algo que é importante parte do problema de reconhecimento óptico de caracteres. Um conjunto de dígitos exemplo já pré-formatados pode ser encontrado como parte da biblioteca Scikit-Learn, contendo 797 amostras de dígitos em uma matriz de 8 × 8 pixels (um vetor de 64 posições).

## Entendendo os Dados

from sklearn.datasets import load\_digits  
digits = load\_digits()  
digits.keys()

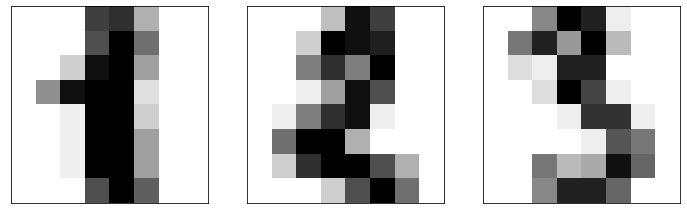
dict\_keys(['data', 'target', 'frame', 'feature\_names', 'target\_names', 'images', 'DESCR'])

digits['data'].shape

(1797, 64)

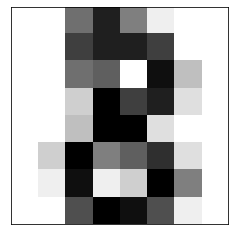
Podemos ver as imagens, por exemplo, dos dígitos 1, 2 e 3 que fazem parte desses exemplos.

fig = plt.figure(figsize=(12, 4))  
  
for i in range(1,4):  
 plt.subplot(1,3,i)  
 plt.imshow(digits.images[i], cmap=plt.cm.binary, interpolation='nearest')  
 plt.xticks([])  
 plt.yticks([])  
plt.show()



E cada imagem traz ainda o *rótulo* de classificação do dígito escrito à mão:

i = 8 # entre com um valor entre 0 e 796  
fig = plt.figure(figsize=(4, 4))  
  
plt.imshow(digits.images[i], cmap=plt.cm.binary, interpolation='nearest')  
plt.xticks([])  
plt.yticks([])  
plt.show()  
  
print('Este dígito é um: ', digits.target[i])



Este dígito é um: 8

Resumindo, temos aqui um conjunto de treinamento com 797 exemplos de dígitos à mão rotulados. Os atributos de entrada,X do nosso conjunto, são os valores de pixels da imagem 8x8 digits.imagens. Por comodidade vamos empregar os valores reformatados para um vetor 64x1 digits.data, o que também poderia ser obtido com um reshape() do Numpy sobre a imagem. Os rótulos de cada imagem, nossa variável objetivo y, encontra-se em digits.target.

digits.images[i]

array([[ 0., 0., 9., 14., 8., 1., 0., 0.],  
 [ 0., 0., 12., 14., 14., 12., 0., 0.],  
 [ 0., 0., 9., 10., 0., 15., 4., 0.],  
 [ 0., 0., 3., 16., 12., 14., 2., 0.],  
 [ 0., 0., 4., 16., 16., 2., 0., 0.],  
 [ 0., 3., 16., 8., 10., 13., 2., 0.],  
 [ 0., 1., 15., 1., 3., 16., 8., 0.],  
 [ 0., 0., 11., 16., 15., 11., 1., 0.]])

digits.images[i].reshape(64)

array([ 0., 0., 9., 14., 8., 1., 0., 0., 0., 0., 12., 14., 14.,  
 12., 0., 0., 0., 0., 9., 10., 0., 15., 4., 0., 0., 0.,  
 3., 16., 12., 14., 2., 0., 0., 0., 4., 16., 16., 2., 0.,  
 0., 0., 3., 16., 8., 10., 13., 2., 0., 0., 1., 15., 1.,  
 3., 16., 8., 0., 0., 0., 11., 16., 15., 11., 1., 0.])

digits.data[i]

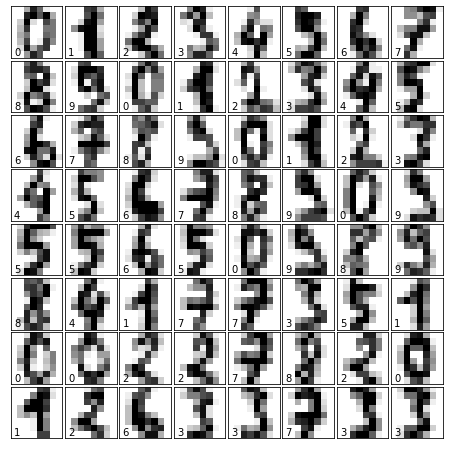
array([ 0., 0., 9., 14., 8., 1., 0., 0., 0., 0., 12., 14., 14.,  
 12., 0., 0., 0., 0., 9., 10., 0., 15., 4., 0., 0., 0.,  
 3., 16., 12., 14., 2., 0., 0., 0., 4., 16., 16., 2., 0.,  
 0., 0., 3., 16., 8., 10., 13., 2., 0., 0., 1., 15., 1.,  
 3., 16., 8., 0., 0., 0., 11., 16., 15., 11., 1., 0.])

digits.target[i]

8

Observe aqui outros exemplos dos dados.

#@markdown you can skip this code  
# source: https://jakevdp.github.io/PythonDataScienceHandbook/05.08-random-forests.html  
# set up the figure  
fig = plt.figure(figsize=(6, 6)) # figure size in inches  
fig.subplots\_adjust(left=0, right=1, bottom=0, top=1, hspace=0.05, wspace=0.05)  
  
# plot the digits: each image is 8x8 pixels  
for i in range(64):  
 ax = fig.add\_subplot(8, 8, i + 1, xticks=[], yticks=[])  
 ax.imshow(digits.images[i], cmap=plt.cm.binary, interpolation='nearest')  
   
 # label the image with the target value  
 ax.text(0, 7, str(digits.target[i]))



Nosso problema agora é construir um melhor modelo de classificação de dígitos escritos à mão com o nosso conjunto de treinamento.

## Aplicando os Classificadores

Vamos inicialmente aplicar somente os classificadores que já conhecemos como o classificador logístico, o Knn, a Árvore de Decisão e a Floresta Aleatória.

Vamos variar esses modelos criando uma lista desses estimadores e implementando um laço de treinamento sobre essa lista, assim como fizemos na aula 5. Dentro de laço de treinamento implementamos uma busca em grade dos melhores hiperparâmetros de cada modelo da nossa lista. Acumulando as métricas de cada resultado, como a acuracidade por exemplo, você poderá escolher o melhor modelo com os melhores hiperparâmetros para essa métrica.

from sklearn.linear\_model import LogisticRegression  
from sklearn.neighbors import KNeighborsClassifier  
from sklearn.tree import DecisionTreeClassifier  
from sklearn.ensemble import RandomForestClassifier  
  
from sklearn.model\_selection import train\_test\_split  
from sklearn.neighbors import KNeighborsClassifier  
from sklearn.model\_selection import GridSearchCV  
from sklearn.metrics import classification\_report  
  
X = digits.data  
y = digits.target  
  
X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, stratify=y, test\_size=0.3, random\_state=0)  
  
base\_estimators = [ LogisticRegression(max\_iter=10000),  
 KNeighborsClassifier(),  
 DecisionTreeClassifier(),  
 RandomForestClassifier(random\_state=0) ]  
   
param\_grids = [ {},  
 {'n\_neighbors': [3,4,5,6,7]},  
 {'max\_depth': [2,3,4,5,6,7,8,9,10], 'criterion': ['gini','entropy']},  
 {'n\_estimators':[3,4,5,6],'max\_depth': [2,3,4,5,6,7,8,9,10], 'criterion': ['gini','entropy']}]   
   
save\_estimators = []  
  
for i in range(len(base\_estimators)):  
 clf = GridSearchCV(base\_estimators[i], param\_grids[i], cv=5, scoring='accuracy')  
 clf.fit(X\_train, y\_train)  
 # print(clf.cv\_results\_)  
 print(clf.best\_estimator\_)  
 save\_estimators.append(clf.best\_estimator\_)  
 print()  
 print("Detailed classification report:")  
 print()  
 y\_pred = clf.predict(X\_test)  
 print(classification\_report(y\_test, y\_pred))  
 print()

LogisticRegression(max\_iter=10000)  
  
Detailed classification report:  
  
 precision recall f1-score support  
  
 0 1.00 1.00 1.00 54  
 1 0.90 0.98 0.94 55  
 2 1.00 0.98 0.99 53  
 3 0.95 0.96 0.95 55  
 4 1.00 0.96 0.98 54  
 5 0.95 0.96 0.95 55  
 6 1.00 0.98 0.99 54  
 7 0.98 0.98 0.98 54  
 8 0.96 0.90 0.93 52  
 9 0.94 0.94 0.94 54  
  
 accuracy 0.97 540  
 macro avg 0.97 0.97 0.97 540  
weighted avg 0.97 0.97 0.97 540  
  
  
KNeighborsClassifier()  
  
Detailed classification report:  
  
 precision recall f1-score support  
  
 0 1.00 1.00 1.00 54  
 1 0.93 1.00 0.96 55  
 2 1.00 0.98 0.99 53  
 3 0.96 0.95 0.95 55  
 4 1.00 0.96 0.98 54  
 5 0.96 1.00 0.98 55  
 6 1.00 1.00 1.00 54  
 7 0.98 1.00 0.99 54  
 8 0.96 0.94 0.95 52  
 9 1.00 0.96 0.98 54  
  
 accuracy 0.98 540  
 macro avg 0.98 0.98 0.98 540  
weighted avg 0.98 0.98 0.98 540  
  
  
DecisionTreeClassifier(criterion='entropy', max\_depth=10)  
  
Detailed classification report:  
  
 precision recall f1-score support  
  
 0 1.00 0.96 0.98 54  
 1 0.90 0.82 0.86 55  
 2 0.86 0.83 0.85 53  
 3 0.79 0.84 0.81 55  
 4 0.84 0.89 0.86 54  
 5 0.82 0.84 0.83 55  
 6 0.91 0.93 0.92 54  
 7 0.94 0.89 0.91 54  
 8 0.72 0.81 0.76 52  
 9 0.85 0.81 0.83 54  
  
 accuracy 0.86 540  
 macro avg 0.86 0.86 0.86 540  
weighted avg 0.86 0.86 0.86 540  
  
  
RandomForestClassifier(criterion='entropy', max\_depth=10, n\_estimators=6,  
 random\_state=0)  
  
Detailed classification report:  
  
 precision recall f1-score support  
  
 0 0.98 1.00 0.99 54  
 1 0.86 0.93 0.89 55  
 2 0.94 0.94 0.94 53  
 3 0.85 0.91 0.88 55  
 4 0.95 0.96 0.95 54  
 5 0.90 0.85 0.88 55  
 6 1.00 1.00 1.00 54  
 7 0.94 0.94 0.94 54  
 8 0.82 0.79 0.80 52  
 9 0.90 0.81 0.85 54  
  
 accuracy 0.91 540  
 macro avg 0.91 0.91 0.91 540  
weighted avg 0.91 0.91 0.91 540

Com os parâmetros acima o melhor estimador obtido foi o Knn com um score de 0.979. Bem difícil de ser melhorado... rs.

[x.score(X\_test, y\_test) for x in save\_estimators]

[0.9666666666666667,  
 0.9796296296296296,  
 0.8611111111111112,  
 0.9148148148148149]

best = np.array( [x.score(X\_test, y\_test) for x in save\_estimators] ).argmax()  
save\_estimators[best]

KNeighborsClassifier()

O estimador está usando os parâmetros padrão, como por exemplo a quantidade de vizinhos mais próximos 5 e a métrica de distância minkowski.

**n\_neighbors=5, metric='minkowski'**

Mas você pode ainda aplicar esse mesmo procedimento para outros modelos que você venha a ter interesse, embora não entremos em detalhe aqui. Por exemplo, abaixo, vamos aplicar os classificadores de Gradient Boosting, Support Vector Machines (SVC) e uma rede neural multicamadas (MLP), poderosos classificadores empregados para muitos problemas complexos principalmente envolvendo dados com grande dimensionalidade como imagens e vídeos.

from sklearn.linear\_model import LogisticRegression   
from sklearn.neighbors import KNeighborsClassifier   
from sklearn.tree import DecisionTreeClassifier   
from sklearn.model\_selection import KFold  
from sklearn.model\_selection import cross\_val\_score  
from sklearn.naive\_bayes import GaussianNB  
from sklearn.tree import DecisionTreeClassifier  
from sklearn.svm import SVC  
from sklearn.neural\_network import MLPClassifier  
from sklearn.ensemble import RandomForestClassifier  
from sklearn.ensemble import AdaBoostClassifier  
from sklearn.ensemble import GradientBoostingClassifier   
  
X = digits.data  
y = digits.target  
  
X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, stratify=y, test\_size=0.3, random\_state=0)  
  
base\_estimators = [ MLPClassifier(max\_iter=5000),  
 GradientBoostingClassifier(),  
 SVC() ]  
   
param\_grids = [ {'hidden\_layer\_sizes':[(20,),(50,),(100,),(50,50)], 'activation': ['relu','tanh']},  
 {},  
 {}]   
   
save\_estimators = []  
  
for i in range(len(base\_estimators)):  
 clf = GridSearchCV(base\_estimators[i], param\_grids[i], cv=5, scoring='accuracy')  
 clf.fit(X\_train, y\_train)  
 # print(clf.cv\_results\_)  
 print(clf.best\_estimator\_)  
 save\_estimators.append(clf.best\_estimator\_)  
 print()  
 print("Accuracy score:")  
 print()  
 y\_pred = clf.predict(X\_test)  
 print(clf.score(X\_test, y\_test))  
 print()

MLPClassifier(activation='tanh', max\_iter=5000)  
  
Accuracy score:  
  
0.9814814814814815  
  
GradientBoostingClassifier()  
  
Accuracy score:  
  
0.9629629629629629  
  
SVC()  
  
Accuracy score:  
  
0.987037037037037

Acima empregamos uma rede neural variando o número de camadas da rede e a função de ativação empregadas. Todos tiveram um desempenho próximo ou superior ao Knn anterior. De fato, esses modelos são bastante poderosos e populares, mas cuidado, o resultado melhor aqui não é uma regra! Isso nem sempre se verifica e você já aprendeu aqui que o melhor modelo depende mesmo sempre dos dados! Não é surpresa, entretanto, o melhor desempenho do modelo de Máquinas de Vetores de Suporte (SVC()) que se popularizou exatamente pelo seu desempenho superior na classificação de dígitos nos anos 90.

# CASO: Classificando o MNIST ()

Um conjunto mais interessante de dígitos escritos à mão é o famoso **MNIST** (<http://yann.lecun.com/exdb/mnist/>) sendo um conjunto de dados *benchmark* para algoritmos de classificação bastante empregado.

Ele é constituído por 70000 imagens de dígitos rotulados escritos à mão em imagens de 28x28 pixels. Um conjunto bastante mais complexo que o anterior. Vamos aplicar aqui apenas os modelos que encontramos os melhores resultados antes, como o K-vizinhos mais próximos, o modelo neural e o de vetores de suporte. A exploração dos dados e a aplicação dos modelos do código anterior é bastante direta e dispensa comentários para código. A execução, entretanto, pode levar aqui alguns minutos.

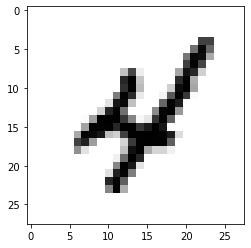
%%time  
from sklearn.datasets import fetch\_openml  
mnist = fetch\_openml('mnist\_784')

CPU times: user 46.3 s, sys: 2.9 s, total: 49.2 s  
Wall time: 1min 4s

mnist.data.shape

(70000, 784)

i = 9 # entre com um valor entre 0 e 796  
fig = plt.figure(figsize=(4, 4))  
  
image = mnist.data.to\_numpy()  
plt.imshow((image[i].reshape(28,28)), cmap=plt.cm.gray\_r, interpolation='nearest')  
plt.show()  
  
print('Este dígito é um: ', mnist.target[i])



Este dígito é um: 4

mnist.data.head()

pixel1 pixel2 pixel3 pixel4 pixel5 pixel6 pixel7 pixel8 pixel9 \  
0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0   
1 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0   
2 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0   
3 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0   
4 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0   
  
 pixel10 ... pixel775 pixel776 pixel777 pixel778 pixel779 pixel780 \  
0 0.0 ... 0.0 0.0 0.0 0.0 0.0 0.0   
1 0.0 ... 0.0 0.0 0.0 0.0 0.0 0.0   
2 0.0 ... 0.0 0.0 0.0 0.0 0.0 0.0   
3 0.0 ... 0.0 0.0 0.0 0.0 0.0 0.0   
4 0.0 ... 0.0 0.0 0.0 0.0 0.0 0.0   
  
 pixel781 pixel782 pixel783 pixel784   
0 0.0 0.0 0.0 0.0   
1 0.0 0.0 0.0 0.0   
2 0.0 0.0 0.0 0.0   
3 0.0 0.0 0.0 0.0   
4 0.0 0.0 0.0 0.0   
  
[5 rows x 784 columns]

mnist.target.head()

0 5  
1 0  
2 4  
3 1  
4 9  
Name: class, dtype: category  
Categories (10, object): ['0', '1', '2', '3', ..., '6', '7', '8', '9']

%%time  
from sklearn.neighbors import KNeighborsClassifier   
from sklearn.svm import SVC  
from sklearn.neural\_network import MLPClassifier  
  
X = mnist.data[0:100] # retire os [] se tiver certeza de querer executar com os 70K dados. Deve levar mais de 30min de execução.  
y = mnist.target[0:100]  
  
X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, stratify=y, test\_size=0.3, random\_state=0)  
  
base\_estimators = [ MLPClassifier(max\_iter=5000),  
 KNeighborsClassifier(),  
 SVC() ]  
   
param\_grids = [ {'hidden\_layer\_sizes':[(100,),(32,32),(32,64,32)], 'activation': ['relu','tanh']},  
 {'n\_neighbors':[3,4,5,6,7,8,9,10]},  
 {}]   
   
save\_estimators = []  
  
for i in range(len(base\_estimators)):  
 clf = GridSearchCV(base\_estimators[i], param\_grids[i], cv=5, scoring='accuracy')  
 clf.fit(X\_train, y\_train)  
 # print(clf.cv\_results\_)  
 print(clf.best\_estimator\_)  
 save\_estimators.append(clf.best\_estimator\_)  
 print()  
 print("Accuracy score:")  
 print()  
 y\_pred = clf.predict(X\_test)  
 print(clf.score(X\_test, y\_test))  
 print()

MLPClassifier(max\_iter=5000)  
  
Accuracy score:  
  
0.7666666666666667

KNeighborsClassifier(n\_neighbors=3)  
  
Accuracy score:  
  
0.6  
  
SVC()  
  
Accuracy score:  
  
0.7333333333333333  
  
CPU times: user 25.1 s, sys: 16.7 s, total: 41.8 s  
Wall time: 22.4 s

# Selecionando Regressores

A mesma abordagem empregada para selecionar modelos e hiperparâmetros de classificadores pode ser empregada para a seleção de regressores. A regressão linear foi um dos primeiros modelos supervisionados que aprendemos aqui. Você deve lembrar: a regressão linear busca fazer a aproximação de uma função desconhecida:

Mas para a qual temos um conjunto de dados de treinamento (exemplos de entradas e as respectivas saídas da função), para uma função linear da forma:

Onde , são os coeficientes que minimizam o erro,

Isso resolve uma série de problemas bastante interessantes como a predição de preço de imóveis ou de veículos. Mas há uma série de problemas, até mesmo simples, onde um regressor linear não é suficientemente adequado. Existem assim inúmeros outros regressores que podem ser aplicados do mesmo modo que a regressão linear para obter uma estimativa **não linear** dos dados.

Onde agora é uma função não linear que queremos que o erro da função estimada seja mínimo por alguma métrica (isto é a diferença entre e ).

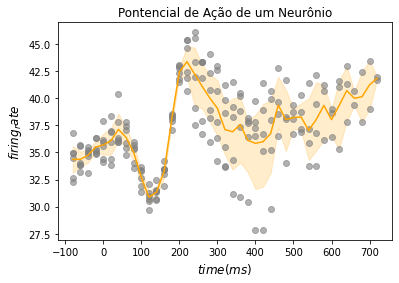
# CASO: Um Conjunto não Linear, Potencial de Ação de um Neurônio

Vamos empregar como exemplo um subconjunto de dados do Potencial de Ação de um neurônio. O Potencial de Ação é uma *função* que explica a ativação dos neurônios e que tem um papel importante nas neurociências e aplicações médicas. Essa é uma função claramente não linear como você pode observar pelo gráfico onde o sns.lineplot produz uma linha de tendência para os dados e seu intervalo de confiança. Não obstante haver uma série de modelos propostos, nós não conhecemos a *função real* desse potencial, mas podemos obter uma série de medidas no tempo (como o subconjunto empregado aqui) e buscar aproximar esses dados por algum modelo ou função.

import seaborn as sns  
df = sns.load\_dataset('dots')  
df = df[(df.choice == 'T2') & (df['align'] == 'dots')]  
df = df[['time','firing\_rate']].reset\_index(drop=True)  
df.head()

time firing\_rate  
0 -80 34.970107  
1 -80 36.785815  
2 -80 34.478506  
3 -80 34.991424  
4 -80 32.241533

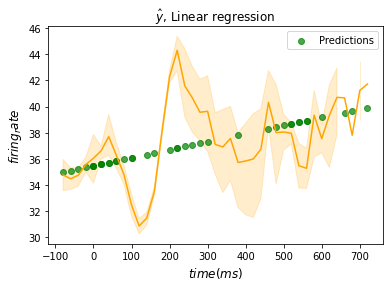
sns.lineplot(x=df.time, y=df.firing\_rate,color='orange')  
plt.scatter(x=df.time, y=df.firing\_rate,color='gray',alpha=0.6)  
  
plt.title('Pontencial de Ação de um Neurônio')  
plt.xlabel('$time (ms)$',fontsize=12)  
plt.ylabel('$firing\_rate$',fontsize=12)  
plt.show()



## Aplicando o Modelo Linear

Podemos aplicar diretamente o modelo de regressão linear como já fizemos antes e comparar nossas estimativas com os dados. Aplicamos aqui uma separação de dados de treinamento e teste para tornar a avaliação do modelo mais efetiva.

from sklearn import linear\_model  
from sklearn.model\_selection import train\_test\_split  
  
X = df[['time']]  
y = df.firing\_rate  
  
X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=1)  
  
regressor = linear\_model.LinearRegression()  
regressor.fit(X\_train, y\_train)  
  
y\_pred = regressor.predict(X\_test)  
  
sns.lineplot(x=X\_train.time,y=y\_train,color='orange')  
plt.scatter(X\_test,y\_pred,color='green',marker='o',alpha=0.7,label='Predictions')  
  
plt.title('$\\hat{y}$' + ', Linear regression')  
plt.xlabel('$time (ms)$',fontsize=12)  
plt.ylabel('$firing\_rate$',fontsize=12)  
plt.legend()  
  
plt.show()



Esse é o melhor modelo que a classe de regressores lineares pode fornecer e, de fato, você observa que ele se ajusta muito pouco aos dados. Modelos lineares não podem capturar a não linearidade da função de ativação!

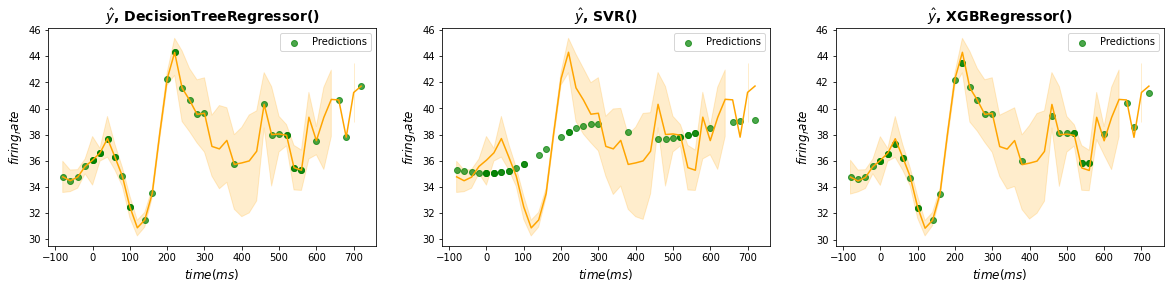
## Regressores não Lineares

Existem muitos outros modelos e que empregam paradigmas bastante diferentes do modelo Linear e permitem obter aproximações *não lineares* dos dados. Você pode por exemplo pensar que podemos aproximar uma função por diferentes polinômios, com diferentes graus, ou ainda por diferentes combinações de séries de dados como as séries de Taylor e Fourier, famosas séries para aproximação de funções na matemática.

Para nós é suficiente saber que existem muitos modelos e que, por partirem de princípios de construção diferentes, levam a diferentes soluções de aproximação da função original.

Você pode ver abaixo a aplicação de 3 modelos regressores bastante empregados o **Decision Tree Regressor**, um modelo baseado em **Máquinas de Vetores de Suporte** e **XGBoosting Regressor**. Todos tem a mesma lógica de aplicação dos estimadores do scikit-learn e da regressão linear. Você pode ainda alterar o código abaixo para experimentar outros modelos que encontram-se nos imports no início do código.

from sklearn.tree import DecisionTreeRegressor  
from sklearn.svm import SVR  
from xgboost.sklearn import XGBRegressor  
from sklearn.linear\_model import BayesianRidge  
from sklearn.linear\_model import ElasticNet  
from sklearn.kernel\_ridge import KernelRidge  
from sklearn.linear\_model import SGDRegressor  
from lightgbm import LGBMRegressor  
from sklearn.ensemble import GradientBoostingRegressor  
from sklearn.neural\_network import MLPRegressor # entradas precisam ser normalizadas  
  
base\_estimators = [ DecisionTreeRegressor(),  
 SVR(),  
 XGBRegressor() ]  
  
plt.figure(figsize=(20,4))  
k = 1  
for regressor in base\_estimators:  
  
 regressor.fit(X\_train, y\_train)  
  
 y\_pred = regressor.predict(X\_test)  
  
 plt.subplot(1,3,k)  
 k = k + 1  
 sns.lineplot(x=X\_train.time,y=y\_train,color='orange')  
 plt.scatter(X\_test,y\_pred,color='green',marker='o',alpha=0.7,label='Predictions')  
  
 plt.title('$\\hat{y}$, ' + str(regressor),fontsize=14,weight='bold')  
 plt.xlabel('$time (ms)$',fontsize=12)  
 plt.ylabel('$firing\_rate$',fontsize=12)  
 plt.legend()  
  
plt.show()



Neste exemplo omitimos por simplicidade a busca dos melhores modelos ou mesmo a a busca em grade por melhores hiperparâmetros, limitando-nos simplesmente a construir os diferentes modelos.

Mas você já pode perceber que podemos aplicar o mesmo raciocínio empregado antes para os classificadores, definindo uma métrica de regressão para os melhores modelos e selecionando assim modelos e hiperparâmetros que mais se ajustam aos dados. É o que fazemos a seguir.

from sklearn.tree import DecisionTreeRegressor  
from sklearn.svm import SVR  
from xgboost.sklearn import XGBRegressor  
from sklearn.linear\_model import BayesianRidge  
from sklearn.linear\_model import ElasticNet  
from sklearn.kernel\_ridge import KernelRidge  
from sklearn.linear\_model import SGDRegressor  
from lightgbm import LGBMRegressor  
from sklearn.ensemble import GradientBoostingRegressor  
from sklearn.neural\_network import MLPRegressor # entradas precisam ser normalizadas  
  
X = df[['time']]  
y = df.firing\_rate  
  
X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=1)  
  
base\_estimators = [ MLPRegressor(max\_iter=8000),  
 DecisionTreeRegressor(),  
 SVR(),  
 XGBRegressor() ]  
   
param\_grids = [ {'hidden\_layer\_sizes':[(32,),(8,8),(8,16,8)], 'activation': ['logistic','tanh'], 'solver':['sgd', 'adam']},  
 {'max\_depth':[3,4,5]},  
 {},  
 {}]   
   
save\_estimators = []  
  
for i in range(len(base\_estimators)):  
 reg = GridSearchCV(base\_estimators[i], param\_grids[i], cv=5, scoring='neg\_mean\_squared\_error')  
 reg.fit(X\_train, y\_train)  
 # print(clf.cv\_results\_)  
 print(reg.best\_estimator\_)  
 save\_estimators.append(reg.best\_estimator\_)  
 print()  
 print("MSE Mean Square Error:")  
 print()  
 y\_pred = reg.predict(X\_test)  
 print(reg.score(X\_test, y\_test))  
 print()

MLPRegressor(activation='logistic', hidden\_layer\_sizes=(32,), max\_iter=8000)  
  
MSE Mean Square Error:  
  
-9.001852217669798  
  
DecisionTreeRegressor(max\_depth=4)  
  
MSE Mean Square Error:  
  
-3.432988535863749  
  
SVR()  
  
MSE Mean Square Error:  
  
-7.82387506810357  
  
MSE Mean Square Error:  
  
-5.222817600405398

save\_estimators

[MLPRegressor(activation='logistic', hidden\_layer\_sizes=(32,), max\_iter=8000),  
 DecisionTreeRegressor(max\_depth=4),  
 SVR(),  
 XGBRegressor()]

Empregamos agora 4 diferentes regressores e diferentes hiperparâmetros. A escolha recai sobre a melhor métrica de **MSE (Mean Square Error)**, uma métrica comum para regressores onde medimos o erro médio com relação a predição e os dados do conjunto de teste. Diferentemente da acuracidade (uma métrica para classificação) aqui o melhor modelo é o que apresenta o menor erro médio. Nos estimadores acima nossa escolha recairia sobre o modelo DecisionTreeRegressor(max\_depth=4).

# CASO: Bike Sharing Prediction

Vamos explorar aqui um caso mais interessante. Vamos aplicar os regressores para a predição de uma Série Temporal! Não vamos entrar em detalhes aqui sobre o que caracteriza uma Série Temporal e você deve estudar melhor isso em outras disciplinas ao longo do curso. Você também deve estudar muitos outros modelos estatísticos que permitem análise e previsão de Séries Temporais, modelos como AR (autoregressivos), MA (médias móveis), ARIMA (autoregressivo integrado com médias móveis) etc. Nossa abordagem aqui é, portanto, bastante simples e apenas visa mostrar para você a possibilidade de empregarmos o aprendizado de máquina para esse tipo de problema.

Para nós basta entendermos que Séries Temporais são um conjunto de valores que variam no tempo: o preço de uma ação na bolsa de valores, com diferentes valores de fechamento para cada dia do ano; o preço mensal do petróleo ao longo dos últimos 10 anos; a série de valores de temperatura média da terra, ou de emissões de CO2 ao longo dos últimos 30 anos; o volume de vendas diário da Amazon nos últimos 3 anos; são todos dados de Séries Temporais, que variam ao longo do tempo apresentando ciclos, tendências e sazonalidades. É também bastante claro a importância de fazermos previsões para esses valores (o valor de uma ação para os próximos 30 dias; a emissão de CO2 para os próximos 5 anos; ou ainda a previsão de vendas da Amazon para o mês de Dezembro). Assim, esse é um problema que pode nos interessar aplicar as técnicas de aprendizado supervisionado para fazer previsões, não obstante muitos outros métodos estatísticos disponíveis para isso.

O conjunto de dados Bike Sharing é uma Série bastante complexa, mas com características bastante comuns a um grande número de problemas, que envolvem demanda de produtos e serviços e vamos buscar fazer previsões do aluguel de bicicletas empregando regressores.

## Preparação dos Dados

Nosso arquivo traz dados hora a hora da demanda do aluguel de bicicletas e, após a indexação dos dados, vamos fazer um *resample* para transformar os dados horários em valores máximos por dia. A ideia é a predição dos valores máximos para que não haja falta da oferta do serviço.

***O Pacote Pandas traz várias facilidades de manipulação de dados de Séries Temporais quando empregamos índices do DataFrame com a dimensão de tempo (data, hora etc.). São operações que seriam difíceis de serem implementadas 'by scratch', como fazer o resample ou o shift dos dados.***

df = pd.read\_csv('https://github.com/Rogerio-mack/Temporal/raw/main/Data/london\_merged.csv')  
df.head()

timestamp cnt t1 t2 hum wind\_speed weather\_code \  
0 2015-01-04 00:00:00 182 3.0 2.0 93.0 6.0 3.0   
1 2015-01-04 01:00:00 138 3.0 2.5 93.0 5.0 1.0   
2 2015-01-04 02:00:00 134 2.5 2.5 96.5 0.0 1.0   
3 2015-01-04 03:00:00 72 2.0 2.0 100.0 0.0 1.0   
4 2015-01-04 04:00:00 47 2.0 0.0 93.0 6.5 1.0   
  
 is\_holiday is\_weekend season   
0 0.0 1.0 3.0   
1 0.0 1.0 3.0   
2 0.0 1.0 3.0   
3 0.0 1.0 3.0   
4 0.0 1.0 3.0

Embora existam muitos outros dados aqui, como os dados climático, e que poderiam ser também empregados para melhor o poder preditivo dos modelos vamos nessa introdução empregar apenas os valores de quantidade de bicicletas alugadas (cnt). Modelos que empregam uma única Série de dados para previsão são modelos univariados, enquanto modelos multivariados empregam múltiplas Séries. Mas não vamos chegar a estudar esses modelos aqui.

from datetime import datetime  
  
df.timestamp = pd.to\_datetime(df.timestamp)  
df['year'] = df.timestamp.dt.year  
df = df[df.year == 2015]  
df = df.set\_index('timestamp')

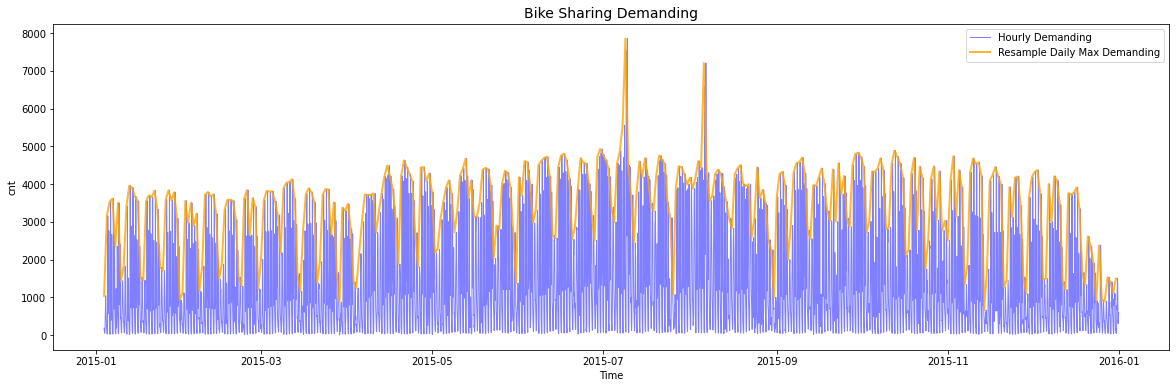
df\_resample = df.copy()  
df\_resample = df\_resample.resample('D').max()

df\_resample = df\_resample[['cnt']]  
df\_resample.head()

cnt  
timestamp   
2015-01-04 1039  
2015-01-05 3161  
2015-01-06 3534  
2015-01-07 3618  
2015-01-08 2351

A série de dados que temos interesse é agora a dos valores cnt, valores máximos de demanda de bicicletas por dia e representada pela linha amarela no gráfico abaixo.

plt.figure(figsize=(20,6))  
  
sns.lineplot(x = df.cnt.index, y = df.cnt, color='blue', alpha=0.5, label='Hourly Demanding',lw=1)  
sns.lineplot(x = df\_resample.cnt.index, y = df\_resample.cnt, color='orange', alpha=0.8, label='Resample Daily Max Demanding',lw=2)  
  
plt.title('Bike Sharing Demanding',fontsize=14)  
plt.xlabel('Days Hours')  
plt.xlabel('Time')  
plt.show()



## Estabelecendo o Problema

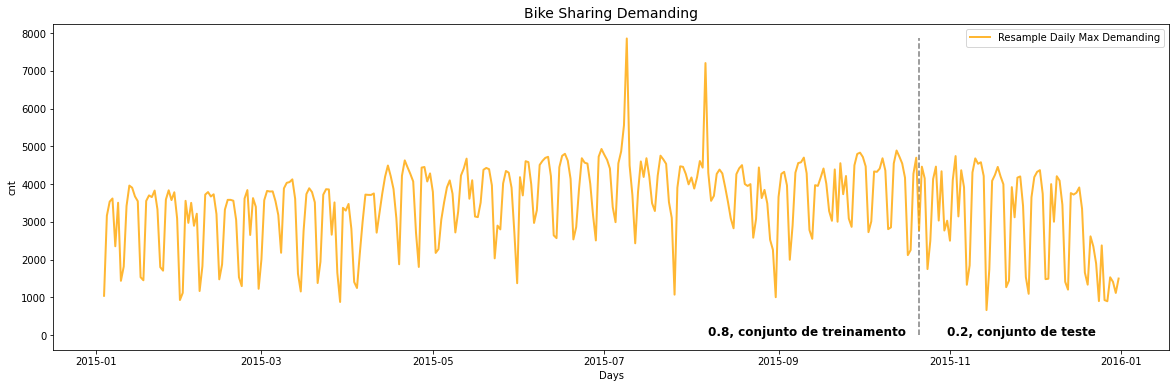
Nosso problema agora é fazer uma previsão no número de aluguéis de bicicletas diário (atributo cnt) criando um modelo que aproxima os valores históricos que temos no nosso conjunto de dados. Aqui também é útil separarmos nesse conjunto de treinamento uma porção dos dados para teste e, assim, podermos minimizar os efeitos de sobreajuste do modelo e sua perda de previsibilidade para novos dados. A representação da Série que desejamos aproximar encontra-se representada no gráfico a seguir.

df = df\_resample  
df.head()

cnt  
timestamp   
2015-01-04 1039  
2015-01-05 3161  
2015-01-06 3534  
2015-01-07 3618  
2015-01-08 2351

test\_size = int(0.2\*len(df))  
train\_size = len(df) - test\_size  
  
# Create Training and Test  
train = df.cnt[:train\_size]  
test = df.cnt[train\_size:]

plt.figure(figsize=(20,6))  
  
sns.lineplot(x = df.cnt.index, y = df.cnt, color='orange', alpha=0.8, label='Resample Daily Max Demanding',lw=2)  
  
plt.title('Bike Sharing Demanding',fontsize=14)  
plt.xlabel('Days Hours')  
plt.xlabel('Days')  
  
plt.vlines(x=df.cnt.index[train\_size] , ymin=0, ymax=df.cnt.max(), color='gray', linestyle='--')  
  
plt.text(df.cnt.index[train\_size - 75] , 0 , '0.8, conjunto de treinamento', fontsize=12, weight='bold')  
plt.text(df.cnt.index[train\_size + 10] , 0 , '0.2, conjunto de teste', fontsize=12, weight='bold')  
plt.legend()  
  
plt.show()



O conjunto de treinamento e teste aqui, entretanto, é construído de forma um pouco diferente que nas tarefas de classificação. No lugar de fazermos uma seleção aleatória, como queremos prever dados futuros com base em dados passados, escolhemos um intervalo contínuo de datas passadas para treinamento e o todo o intervalo restante (datas futuras) como teste. Existem uma série de outros procedimentos de validação e testes que poderíamos aplicar mas, para essa introdução, vamos nos limitar a aplicar esse modelo simples.

#@markdown  
def plot\_forecast(y, y\_pred, X\_train, title='Forecasting'):  
  
 plt.figure(figsize=(20,6))  
 plt.plot(y, color='orange', alpha=0.95, label='Real')  
 plt.plot(y\_pred, color='darkred', alpha=0.6, label='Forecast', marker='o')  
 plt.plot(y\_pred[0:X\_train.shape[0]], color='blue', alpha=0.6, label='Predicted', marker='o')  
  
 plt.vlines(x=y.index[X\_train.shape[0]] , ymin=0, ymax=y.max(), color='gray', linestyle='--')  
  
 plt.title(title, fontsize=14, weight='bold')  
 plt.xlabel('Days Hours')  
 plt.xlabel('Bike Demands')  
  
 plt.text(y.index[X\_train.shape[0] - 75] , 0 , '0.8, conjunto de treinamento', fontsize=12, weight='bold')  
 plt.text(y.index[X\_train.shape[0] + 10] , 0 , '0.2, conjunto de teste', fontsize=12, weight='bold')  
 plt.legend()  
 plt.show()  
  
 return

#@markdown  
# metrics  
from statsmodels.tsa.stattools import acf  
  
def forecast\_accuracy(forecast, actual, print\_flag=True):  
 mape = np.mean(np.abs(forecast - actual)/np.abs(actual)) # MAPE  
 me = np.mean(forecast - actual) # ME  
 mae = np.mean(np.abs(forecast - actual)) # MAE  
 mpe = np.mean((forecast - actual)/actual) # MPE  
 mse = np.mean((forecast - actual)\*\*2) # MSE  
 rmse = np.mean((forecast - actual)\*\*2)\*\*.5 # RMSE  
 corr = np.corrcoef(forecast, actual)[0,1] # CORR  
 acf1 = acf(forecast - actual)[1] # ACF1  
  
 metrics = {'MSE':mse, 'MAPE':mape, 'ME':me, 'MAE': mae, 'MPE': mpe, 'RMSE':rmse, 'ACF1':acf1, 'CORR':corr}  
  
 if print\_flag:  
 for key, value in metrics.items():  
 print(f'{key}: \t {value:.2f}')  
   
 return metrics

## Feature Engineering

Nossos dados agora têm o seguite aspecto:

df.head()

cnt  
timestamp   
2015-01-04 1039  
2015-01-05 3161  
2015-01-06 3534  
2015-01-07 3618  
2015-01-08 2351

**Como transformar isso em um problema de aproximação de mapeamento de entradas e saídas?** Isto é, como transformar isso em um problema que aproxima uma função,

dado um conjunto de dados , em que é o número de instâncias com atributos?

**Olhando nosso conjunto de dados df tem uma única variável! Se esses são os valores de onde estão os valores de ???**

Aqui é que entra a ideia de engenharia de atributos, ou *feature engineering*: **você vai criar o valor desses atributos**.

Uma ideia comum às Séries Temporais é a de que os valores presentes dependem dos valores passados da Série. Por exemplo, o valor do BitCoin hoje depende mais dos valores de ontem e de anteontem do que dos valores da semana passada, mas mais dos valores da semana passada do que dos valores de um ano atrás. O número de passagens aéreas apresenta *sazonalidades*, períodos diferentes do tempo como finais de semana e o final do ano influenciam fortemente no número de viagens e, assim, o número de passagens aéreas vendidas no final de semana ou no final do ano pode apresentar forte relação com os mesmos valores no final de semana passado no ou final do ano anterior. Você vai aproveitar essa ideia para construir as variáveis do seu modelo.

A ideia é criar novos atributos preditores de aprendizado com base no conhecimento que temos dos dados quanto as suas características. Analisar as características de uma Série Temporal não é uma tarefa fácil, mas adotaremos um procedimento simples aqui. Adotaremos que nossa Série Temporal tem um comportamento *autocorrelacionado*, isto é, o número de aluguéis de bicicletas hoje *depende* unicamente da quantidade de aluguéis nos dias anteriores (note, uma única variável correlacionada com seus próprios valores no tempo). Você vai construir assim atributos ... para criar um mapeamento de valores :

Isso é mais será mais fácil para você entender olhando os dados diretamente. Vamos empregar aqui uma janela de 7 dias, isto é, nós vamos considerar a previsão do número de aluguéis com base unicamente na influência do número de aluguéis nos últimos 7 dias. Essa escolha aqui e arbitrária e esse número poderia ser obtido através de métodos de análise de Séries Temporais, mas que estão fora do nosso escopo aqui. Você poderá, entretanto, experimentar outros valores.

A função shift do Pandas é útil para construir facilmente esses valores.

def ml\_timeseries\_df(df, n\_lags, prefix=''):  
  
 df\_ml = pd.DataFrame()  
  
 for i in range(n\_lags,0,-1):  
 df\_ml[prefix + 't-' + str(i)] = df.shift(i).values[:,0]  
 df\_ml['t'] = df.values[:,0]  
 df\_ml = df\_ml[n\_lags:]  
 df\_ml = df\_ml.reset\_index(drop=True)  
  
 return df\_ml   
  
df = ml\_timeseries\_df(df\_resample[['cnt']], n\_lags=7)  
df.head()

t-7 t-6 t-5 t-4 t-3 t-2 t-1 t  
0 1039.0 3161.0 3534.0 3618.0 2351.0 3505.0 1436.0 1813  
1 3161.0 3534.0 3618.0 2351.0 3505.0 1436.0 1813.0 3418  
2 3534.0 3618.0 2351.0 3505.0 1436.0 1813.0 3418.0 3960  
3 3618.0 2351.0 3505.0 1436.0 1813.0 3418.0 3960.0 3904  
4 2351.0 3505.0 1436.0 1813.0 3418.0 3960.0 3904.0 3673

Agora, nosso problema de Séries Temporais, foi transformado em um problema de mapeamento de valores de entrada para valores de saída e podemos, então, aplicar um modelo de Aprendizado Supervisionado para aproximarmos essa função.

Graphical user interface, text, application, table, Excel

Description automatically generated

**Fig. 3. Reconstrução da Série de Dados em uma Tabela de Entradas e Saídas. Os valores de saída sendo estimados a partir de 7 valores anteriores da série (7-lags).**

Organizado os dados desse modo podemos separar a primeira porção dos dados, , como o nosso conjunto de Treinamento e aplicar modelos regressores para a previsão ou *forecasting* do restante, ,dos dados da Série.

def split\_train\_test(X, y, test\_perc=0.2):  
   
 test\_size = int(len(df) \* test\_perc)  
 train\_size = len(df) - test\_size  
  
 X\_train, y\_train = X[:train\_size], y[:train\_size]  
 X\_test, y\_test = X[train\_size:], y[train\_size:]  
  
 return X\_train, y\_train, X\_test, y\_test  
  
X = df.drop(columns='t')  
y = df[['t']]   
  
X\_train, y\_train, X\_test, y\_test = split\_train\_test(X, y, test\_perc=0.2)

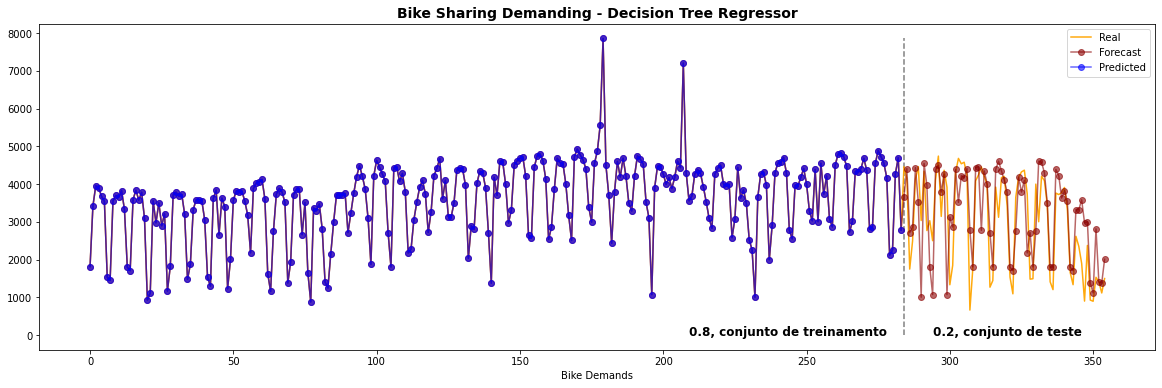
O código para aplicação do regressor é o mesmo que empregamos antes no exemplo da Função neuronal de Ativação, apenas que agora os dados de entrada e saída são outros, e podemos escolher dentre diversos regressores (preferencialmente, neste caso, não lineares).

### Decision Tree Regressor

Vamos primeiramente aplicar o regressor Decision Tree Regressor e verificar os resultados do modelo.

from sklearn.tree import DecisionTreeRegressor  
  
regressor = DecisionTreeRegressor(random\_state=0)  
  
regressor.fit(X\_train, y\_train)  
  
y\_pred = regressor.predict(X)

plot\_forecast(y, y\_pred, X\_train, title='Bike Sharing Demanding - Decision Tree Regressor')



dict\_metrics = {}  
  
cur\_metrics = forecast\_accuracy(y\_pred[len(y\_train):], y[len(y\_train):].values.reshape(-1),print\_flag=False)  
dict\_metrics['DecisionTreeRegressor'] = cur\_metrics  
  
print('MAE para DecisionTreeRegressor:', dict\_metrics['DecisionTreeRegressor']['MAE'])

MAE para DecisionTreeRegressor: 687.9577464788732

### 

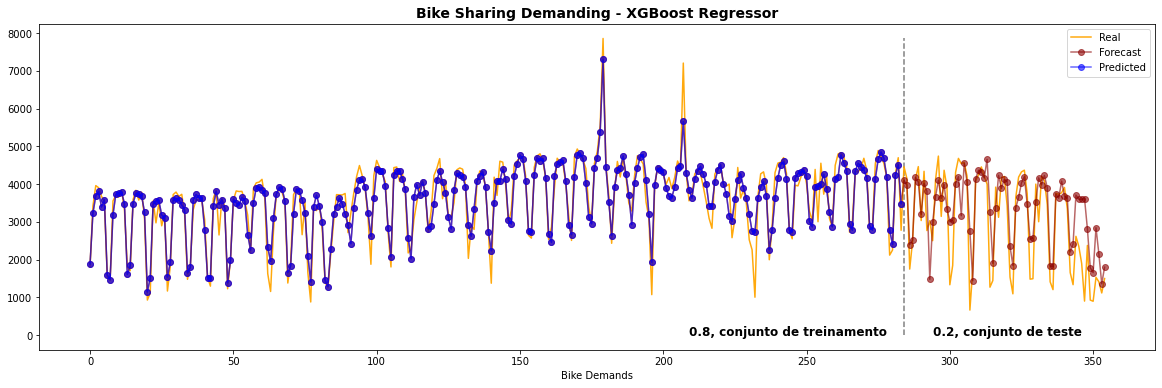
### XGBoost Regressor

Do mesmo modo podemos aplicar o regressor XGBoost Regressor você pode experimentar outros modelos se quiser.

from xgboost.sklearn import XGBRegressor  
  
regressor = XGBRegressor(random\_state=0)  
  
regressor.fit(X\_train,y\_train.values.reshape(-1))  
  
y\_pred = regressor.predict(X)

[14:31:41] WARNING: /workspace/src/objective/regression\_obj.cu:152: reg:linear is now deprecated in favor of reg:squarederror.

plot\_forecast(y, y\_pred, X\_train, title='Bike Sharing Demanding - XGBoost Regressor')   
  
cur\_metrics = forecast\_accuracy(y\_pred[len(y\_train):], y[len(y\_train):].values.reshape(-1),print\_flag=False)  
dict\_metrics['GradientBoostingRegressor'] = cur\_metrics  
  
print('MAE para GradientBoostingRegressor:', dict\_metrics['GradientBoostingRegressor']['MAE'])



MAE para GradientBoostingRegressor: 622.8062314315581

Selecionando a métrica MAE (erro quadrático médio) dos dois modelos você observa um menor erro do modelo GradientBoostingRegressor. É fácil ver que você pode aqui também aplicar o mesmo esquema de seleção de modelos e de hiperparâmetros com o GridSearchCV que empregamos antes para regressores para escolher entre inúmeros outros modelos e hiperparâmetros um melhor modelo, tendo em mãos uma solução bastante poderosa para a previsão de Séries Temporais com modelos de Aprendizado de Máquina. Essa tarefa, entretanto, vamos deixar para você! ;-)

# Sumário da Aula

Nesta aula você explorou modelos completos de aprendizado supervisionado, tanto para Classificação como para Regressão fazendo a Seleção de **melhores hiperparâmetros** dos modelos e a **seleção de diferentes modelos** que competem. Você também teve contato vários outros modelos importantes, como **modelos neurais** e **regressores não lineares**, e você pôde ver como eles podem ser **facilmente aplicados** com o que você já aprendeu aqui, mesmo sem ter aprendido completamente seus princípios (o que você verá em outras disciplinas mais adiante!).

Você também pôde trabalhar aqui com dois conjuntos de dados um pouco diferentes dos dados simplesmente tabulares que empregamos antes. Trabalhar **Dados de Imagens** e **Séries Temporais**, mesmo com modelos simples de aprendizado, permite você compreender melhor a capacidade e limitações desses modelos, abrindo o espaço para você entender modelos mais complexos que você verá na sequência do seu curso em outras disciplinas.

# Para Saber Mais

* Interessado em saber mais sobre o uso de Aprendizado de Máquina para previsão de Séries Temporais? Acesse **Kaggle Time Series** Um curso online com uma aborgem de Aprendizado de Máquina e Engenharia de Features aplicado à Séries Uni e Multivariadas <https://www.kaggle.com/learn/time-series>.
* **Modelos Estatísticos para Séries Temporais**. Previsões de Séries Temporais são feitas à muito tempo, mesmo antes de surgirem modelos de aprendizado de máquina. Aqui você pode explorar os modelos estatísticos tradicionais de Séries Temporais aplicados com Python. **Time Series Forecasting in Python** | Book Online & Videos <https://livebook.manning.com/book/time-series-forecasting-in-python-book/welcome/v-8/> <https://www.youtube.com/channel/UC-0lpiwlftqwC7znCcF83qg/featured>
* Acesse o Site de **THE MNIST DATABASE of handwritten digits** <http://yann.lecun.com/exdb/mnist/> para ver alguns dos benchmarks de classificação da base MNIST. **Yann LeCun** é um dos mais influentes cientistas de IA e Aprendizado de Máquina, um dos criadores dos modelos de Deep Learning e Cientista Chefe de IA da Meta (FaceBook).
* Explore diferentes modelos de classificação com o scikit-learn. Você pode começar por aqui:
* **Choosing the right estimator** <https://scikit-learn.org/stable/tutorial/machine_learning_map/index.html>, ou
* **Classifier comparison** <https://scikit-learn.org/stable/auto_examples/classification/plot_classifier_comparison.html>
* Que tal criar seus *próprios projetos* empregando os mesmos dados que empregamos aqui mas com outros modelos, ou empregue outros dados de interesse. Procure explorar modelos mais elaborados como SVC (Support Vector Machines) e AdaBoostClassifier, e empregar dados com mais classes de saída e maior volume de casos. Você também pode empregar modelos Árvore ou mesmo de K-vizinhos mais próximos para regressões não lineares. Os modelos código e o que você aprendeu até aqui são bastante suficientes para você avançar bastante rapidamente nesses projetos.

# Referências

* 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:15 de Maio de 2022.
* Oliveira, Rogério de; Albarracin, Orlando, Y.E.; Silva, Gustavo Rocha. **Séries Temporais para Engenharia e Outras Áreas**. (to be appear, 2023). Editora Mackenzie, Coleção Conexão Inicial.
* \_\_\_. **Tuning the hyper-parameters of an estimator**, Disponível em: <https://scikit-learn.org/stable/modules/grid_search.html#grid-search> Acesso em: 15 de Maio de 2022.
* \_\_\_. **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: 15 de Maio de 2022.
* \_\_\_. **Kaggle Time Series | Course Online**, Disponível em: <https://www.kaggle.com/learn/time-series>. Acesso em: 15 de Maio de 2022.