Tutorial

November 22, 2019

1 Tutorial: Python para Data Science

1.1 Sumário

- ### 1.1.1 1. Um pouco sobre estruturas de dados no Python
- ### 1.1.2 2. Loops e condicionais
- ### 1.1.3 3. Pandas DataFrames, manipulação e visualização de dados no Seaborn
- ### 1.1.4 4. Regressão linear
- ### 1.1.5 5. Fluxo de Aprendizagem de Máquina

```
In [1]: import this
```

```
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
```

```
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

1.2 1. Um pouco sobre estruturas de dados no Python

1.2.1 1.1 Listas

```
In [3]: fibonnaci_lista = [1,1,2,3,5,8,13]
```

```
In [4]: fibonnaci_lista[0] #indexador começa em zero
```

```
Out[4]: 1
```

```
In [5]: fibonnaci_lista[2:3] #não inclui número à direita
```

```
Out[5]: [2]
```

```
In [6]: fibonnaci_lista[2:4]
```

```
Out[6]: [2, 3]
```

```
In [7]: fibonnaci_lista[2:]
```

```
Out[7]: [2, 3, 5, 8, 13]
```

1.2.2 1.2 Strings

```
In [8]: print('Hello World')
```

```
Hello World
```

```
In [9]: print('Hello'+'World')
```

```
HelloWorld
```

1.2.3 1.3 Tuplas

```
In [10]: fibonnaci_tupla = 1,1,2,3,5,8,13

In [11]: fibonnaci_tupla

Out[11]: (1, 1, 2, 3, 5, 8, 13)

In [12]: tupla = (1,1),(2,2)

In [13]: tupla

Out[13]: ((1, 1), (2, 2))
```

1.2.4 1.4 Dicionários

```
In [14]: dicionario = {'General Electric':'GE','Nokia':'Nok','Petrobras':'PBR'}

In [15]: dicionario

Out[15]: {'General Electric': 'GE', 'Nokia': 'Nok', 'Petrobras': 'PBR'}

In [16]: dicionario['General Electric']

Out[16]: 'GE'

In [17]: categoria = {'Renda alta':3,'Renda média':2,'Renda baixa':1}

In [18]: categoria

Out[18]: {'Renda alta': 3, 'Renda média': 2, 'Renda baixa': 1}

In [19]: categoria['Renda alta']

Out[19]: 3
```

1.3 2. Loops e condicionais

```
In [20]: for i in range(0,4):
           print(fibonnaci_lista[i])

1
1
2
3


In [21]: if fibonnaci_lista[2] == 2:
             print('True')
         else:
             print('False')

True


In [22]: fibonnaci_lista[2] == 2 #booleano

Out[22]: True
```

1.4 3. Pandas DataFrames, manipulação e visualização de dados

```
In [23]: import pandas as pd
         import numpy as np
         import matplotlib as plt
         import seaborn as sns
         %matplotlib inline
```

O seaborn é uma biblioteca de visualização de dados que tem alguns datasets 'built-in', por exemplo, o clássico iris:

```
In [24]: iris = sns.load_dataset('iris')
         iris.head()
```

```
Out[24]:    sepal_length  sepal_width  petal_length  petal_width species
         0           5.1          3.5           1.4          0.2  setosa
         1           4.9          3.0           1.4          0.2  setosa
         2           4.7          3.2           1.3          0.2  setosa
         3           4.6          3.1           1.5          0.2  setosa
         4           5.0          3.6           1.4          0.2  setosa
```

```
In [25]: len(iris)
```

```
Out[25]: 150
```

```
In [26]: iris.shape
```

```
Out[26]: (150, 5)
```

```
In [27]: iris.describe()
```

```
Out[27]:        sepal_length  sepal_width  petal_length  petal_width
         count    150.000000   150.000000    150.000000   150.000000
         mean       5.843333     3.057333      3.758000     1.199333
         std        0.828066     0.435866      1.765298     0.762238
         min        4.300000     2.000000      1.000000     0.100000
         25%        5.100000     2.800000      1.600000     0.300000
         50%        5.800000     3.000000      4.350000     1.300000
         75%        6.400000     3.300000      5.100000     1.800000
         max        7.900000     4.400000      6.900000     2.500000
```

```
In [28]: iris.species.unique()
```

```
Out[28]: array(['setosa', 'versicolor', 'virginica'], dtype=object)
```

1.4.1 Usando operadores lógicos

Por exemplo, queremos olhar os valores da variável *sepal_length*:

```
In [29]: iris['sepal_length'].values
```

```
Out[29]: array([5.1, 4.9, 4.7, 4.6, 5. , 5.4, 4.6, 5. , 4.4, 4.9, 5.4, 4.8, 4.8,
                4.3, 5.8, 5.7, 5.4, 5.1, 5.7, 5.1, 5.4, 5.1, 4.6, 5.1, 4.8, 5. ,
                5. , 5.2, 5.2, 4.7, 4.8, 5.4, 5.2, 5.5, 4.9, 5. , 5.5, 4.9, 4.4,
                5.1, 5. , 4.5, 4.4, 5. , 5.1, 4.8, 5.1, 4.6, 5.3, 5. , 7. , 6.4,
                6.9, 5.5, 6.5, 5.7, 6.3, 4.9, 6.6, 5.2, 5. , 5.9, 6. , 6.1, 5.6,
                6.7, 5.6, 5.8, 6.2, 5.6, 5.9, 6.1, 6.3, 6.1, 6.4, 6.6, 6.8, 6.7,
                6. , 5.7, 5.5, 5.5, 5.8, 6. , 5.4, 6. , 6.7, 6.3, 5.6, 5.5, 5.5,
                6.1, 5.8, 5. , 5.6, 5.7, 5.7, 6.2, 5.1, 5.7, 6.3, 5.8, 7.1, 6.3,
                6.5, 7.6, 4.9, 7.3, 6.7, 7.2, 6.5, 6.4, 6.8, 5.7, 5.8, 6.4, 6.5,
                7.7, 7.7, 6. , 6.9, 5.6, 7.7, 6.3, 6.7, 7.2, 6.2, 6.1, 6.4, 7.2,
                7.4, 7.9, 6.4, 6.3, 6.1, 7.7, 6.3, 6.4, 6. , 6.9, 6.7, 6.9, 5.8,
                6.8, 6.7, 6.7, 6.3, 6.5, 6.2, 5.9])
```

Imagine que queremos ver esses valores somente para a espécie Setosa. Podemos fazer o mesmo usando operadores lógicos ou usando o próprio Pandas.

```
In [30]: setosa = iris["species"] == 'setosa'
         iris[setosa].head()
```

```
Out[30]:    sepal_length  sepal_width  petal_length  petal_width species
         0           5.1          3.5           1.4          0.2  setosa
         1           4.9          3.0           1.4          0.2  setosa
         2           4.7          3.2           1.3          0.2  setosa
         3           4.6          3.1           1.5          0.2  setosa
         4           5.0          3.6           1.4          0.2  setosa
```

```
In [31]: iris.petal_length[setosa].values
```

```
Out[31]: array([1.4, 1.4, 1.3, 1.5, 1.4, 1.7, 1.4, 1.5, 1.4, 1.5, 1.5, 1.6, 1.4,
                1.1, 1.2, 1.5, 1.3, 1.4, 1.7, 1.5, 1.7, 1.5, 1. , 1.7, 1.9, 1.6,
                1.6, 1.5, 1.4, 1.6, 1.6, 1.5, 1.5, 1.4, 1.5, 1.2, 1.3, 1.4, 1.3,
                1.5, 1.3, 1.3, 1.3, 1.6, 1.9, 1.4, 1.6, 1.4, 1.5, 1.4])
```

```
In [32]: iris['petal_length'][setosa].values
```

```
Out[32]: array([1.4, 1.4, 1.3, 1.5, 1.4, 1.7, 1.4, 1.5, 1.4, 1.5, 1.5, 1.6, 1.4,
                1.1, 1.2, 1.5, 1.3, 1.4, 1.7, 1.5, 1.7, 1.5, 1. , 1.7, 1.9, 1.6,
                1.6, 1.5, 1.4, 1.6, 1.6, 1.5, 1.5, 1.4, 1.5, 1.2, 1.3, 1.4, 1.3,
                1.5, 1.3, 1.3, 1.3, 1.6, 1.9, 1.4, 1.6, 1.4, 1.5, 1.4])
```

```
In [33]: iris['petal_length'].value_counts(ascending=False)
```

```
Out[33]: 1.5    13
         1.4    13
         5.1     8
         4.5     8
         1.3     7
         1.6     7
         5.6     6
```

```
4.0     5
4.9     5
4.7     5
4.8     4
1.7     4
4.4     4
4.2     4
5.0     4
4.1     3
5.5     3
4.6     3
6.1     3
5.7     3
3.9     3
5.8     3
1.2     2
1.9     2
6.7     2
3.5     2
5.9     2
6.0     2
5.4     2
5.3     2
3.3     2
4.3     2
5.2     2
6.3     1
1.1     1
6.4     1
3.6     1
3.7     1
3.0     1
3.8     1
6.6     1
6.9     1
1.0     1
Name: petal_length, dtype: int64
```

Podemos também querer visualizar dados das espécies Setosa **ou** Versicolor:

```
In [36]: setosa = iris["species"] == 'setosa'
         versicolor = iris["species"] == 'versicolor'
         iris['petal_length'][setosa|versicolor].values #operador or
```

```
Out[36]: array([1.4, 1.4, 1.3, 1.5, 1.4, 1.7, 1.4, 1.5, 1.4, 1.5, 1.5, 1.6, 1.4,
                1.1, 1.2, 1.5, 1.3, 1.4, 1.7, 1.5, 1.7, 1.5, 1. , 1.7, 1.9, 1.6,
                1.6, 1.5, 1.4, 1.6, 1.6, 1.5, 1.5, 1.4, 1.5, 1.2, 1.3, 1.4, 1.3,
                1.5, 1.3, 1.3, 1.3, 1.6, 1.9, 1.4, 1.6, 1.4, 1.5, 1.4, 4.7, 4.5,
```

```
4.9, 4. , 4.6, 4.5, 4.7, 3.3, 4.6, 3.9, 3.5, 4.2, 4. , 4.7, 3.6,
4.4, 4.5, 4.1, 4.5, 3.9, 4.8, 4. , 4.9, 4.7, 4.3, 4.4, 4.8, 5. ,
4.5, 3.5, 3.8, 3.7, 3.9, 5.1, 4.5, 4.5, 4.7, 4.4, 4.1, 4. , 4.4,
4.6, 4. , 3.3, 4.2, 4.2, 4.2, 4.3, 3. , 4.1])
```

Crosstabs são tabelas cruzadas uteis para verificar correlações possíveis

```
In [37]: cross = pd.crosstab(iris['species'], iris['petal_width'])
```

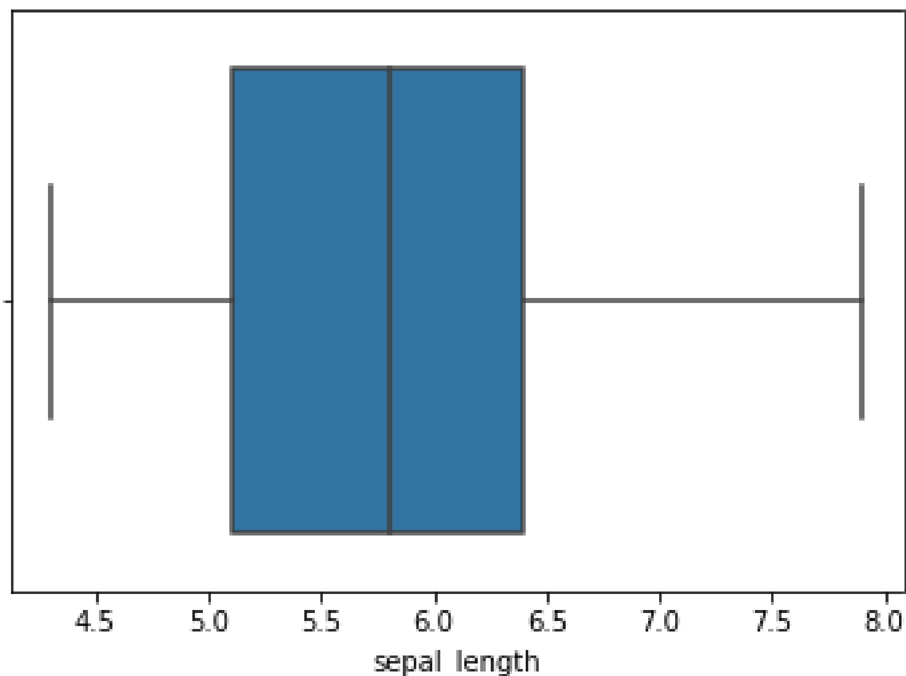
```
In [38]: cross.head()
```

```
Out[38]: petal_width  0.1  0.2  0.3  0.4  0.5  0.6  1.0  1.1  1.2  1.3  ...  1.6  1.7  \
         species                                                        ...
         setosa         5   29    7    7    1    1    0    0    0    0  ...    0    0
         versicolor     0    0    0    0    0    0    7    3    5   13  ...    3    1
         virginica      0    0    0    0    0    0    0    0    0    0  ...    1    1

         petal_width  1.8  1.9  2.0  2.1  2.2  2.3  2.4  2.5
         species
         setosa         0    0    0    0    0    0    0    0
         versicolor     1    0    0    0    0    0    0    0
         virginica     11    5    6    6    3    8    3    3

         [3 rows x 22 columns]
```

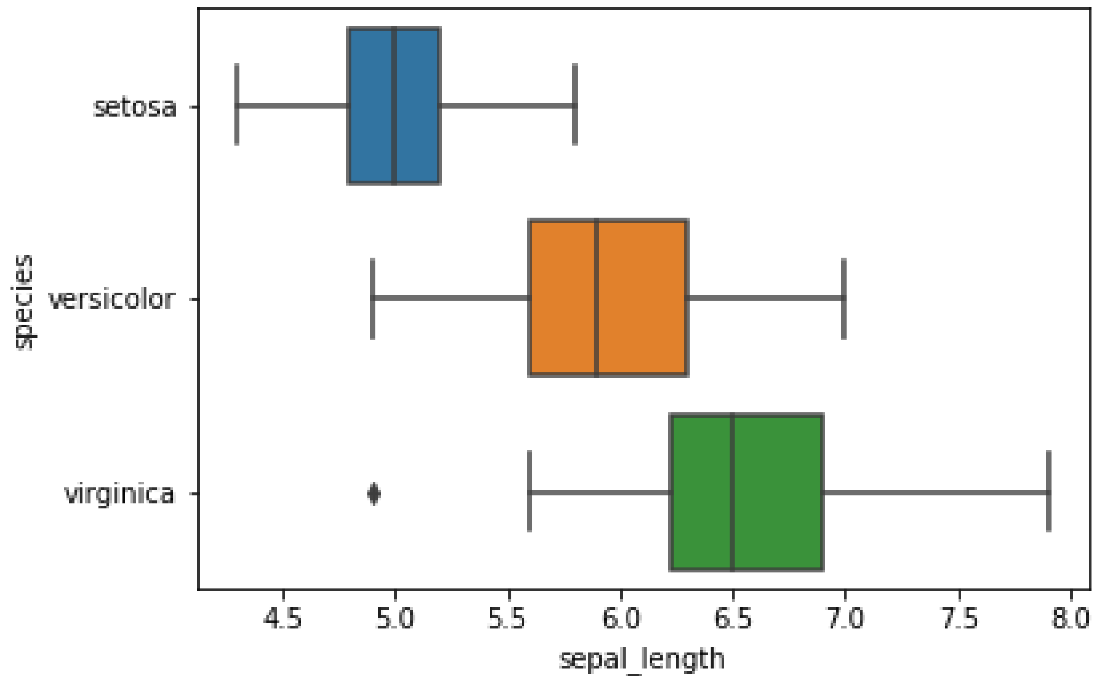
Em geral, para qualquer modelagem de dados, é interessante analisar a distribuição das variáveis. Uma boa prática de análise exploratória é analisar os histogramas e boxplots:

```
In [39]: sns.boxplot(iris['sepal_length'])
```

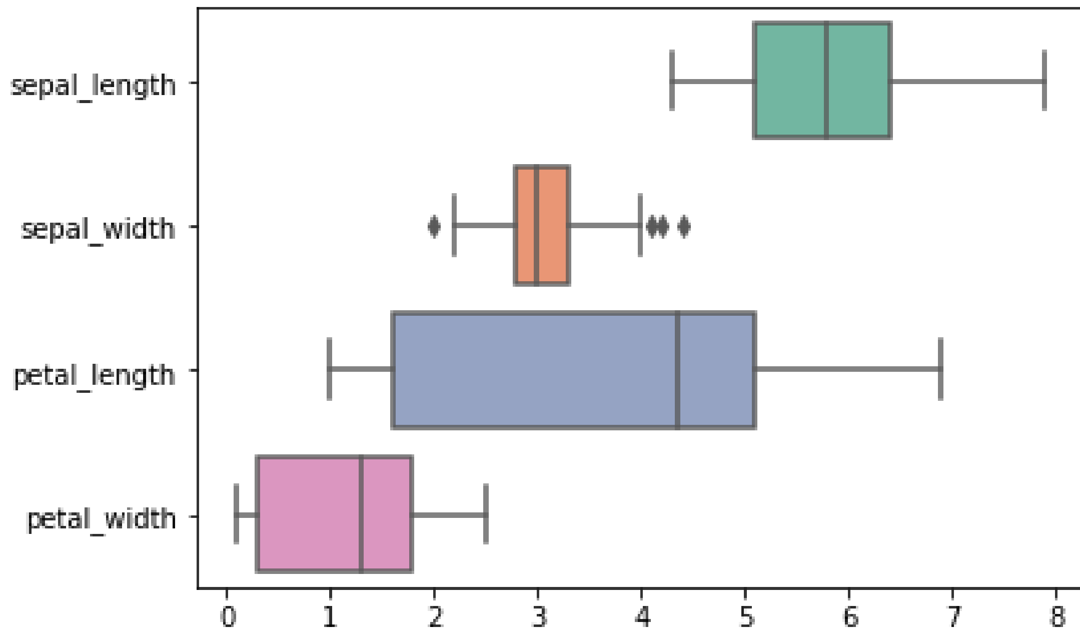
```
Out[39]: <matplotlib.axes._subplots.AxesSubplot at 0x1a1bbd8f98>
```



```
In [40]: sns.boxplot(x='sepal_length',y='species',data=iris)

Out[40]: <matplotlib.axes._subplots.AxesSubplot at 0x1a1bed6d68>
```



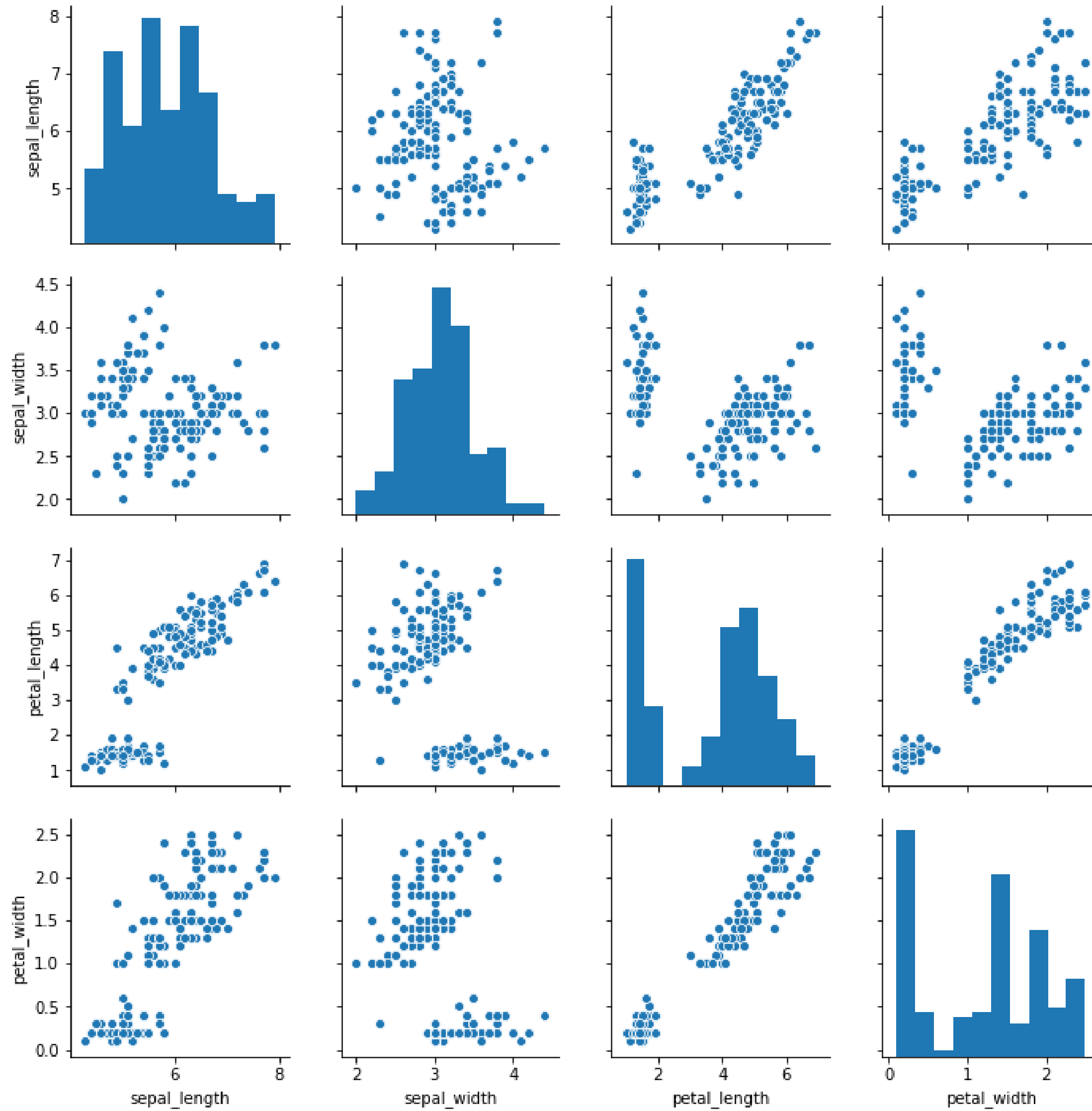

```
In [41]: # Podemos utilizar o dataset inteiro e plotar o boxplot para todas as variáveis numér
         sns.boxplot(data=iris, orient="h", palette="Set2")

Out[41]: <matplotlib.axes._subplots.AxesSubplot at 0x1a1bff1198>
```



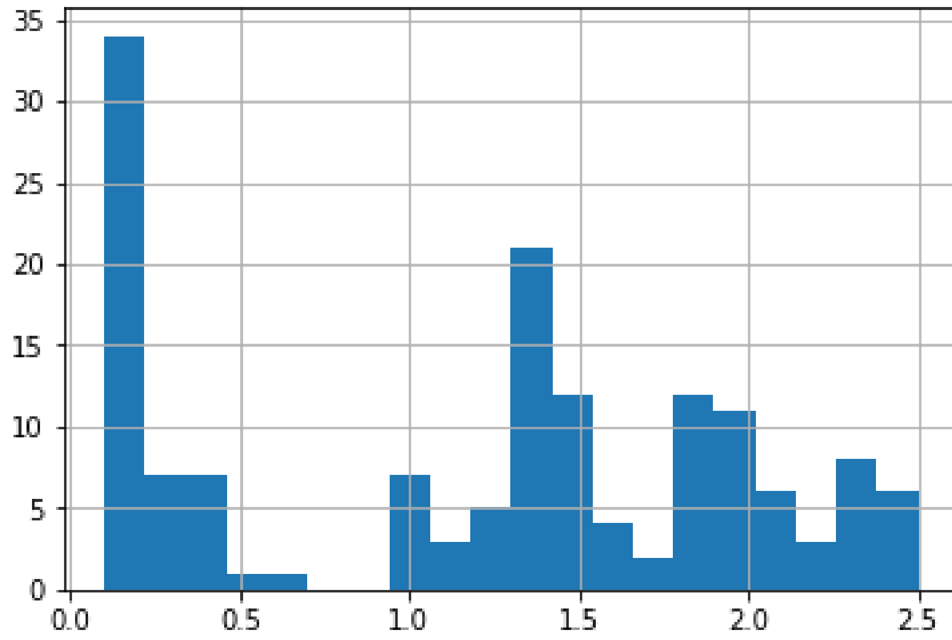
Um gráfico interessante que o pacote seaborn oferece é o pairplot, uma mistura de histograma com scatterplot:

```
In [42]: sns.pairplot(iris)
```

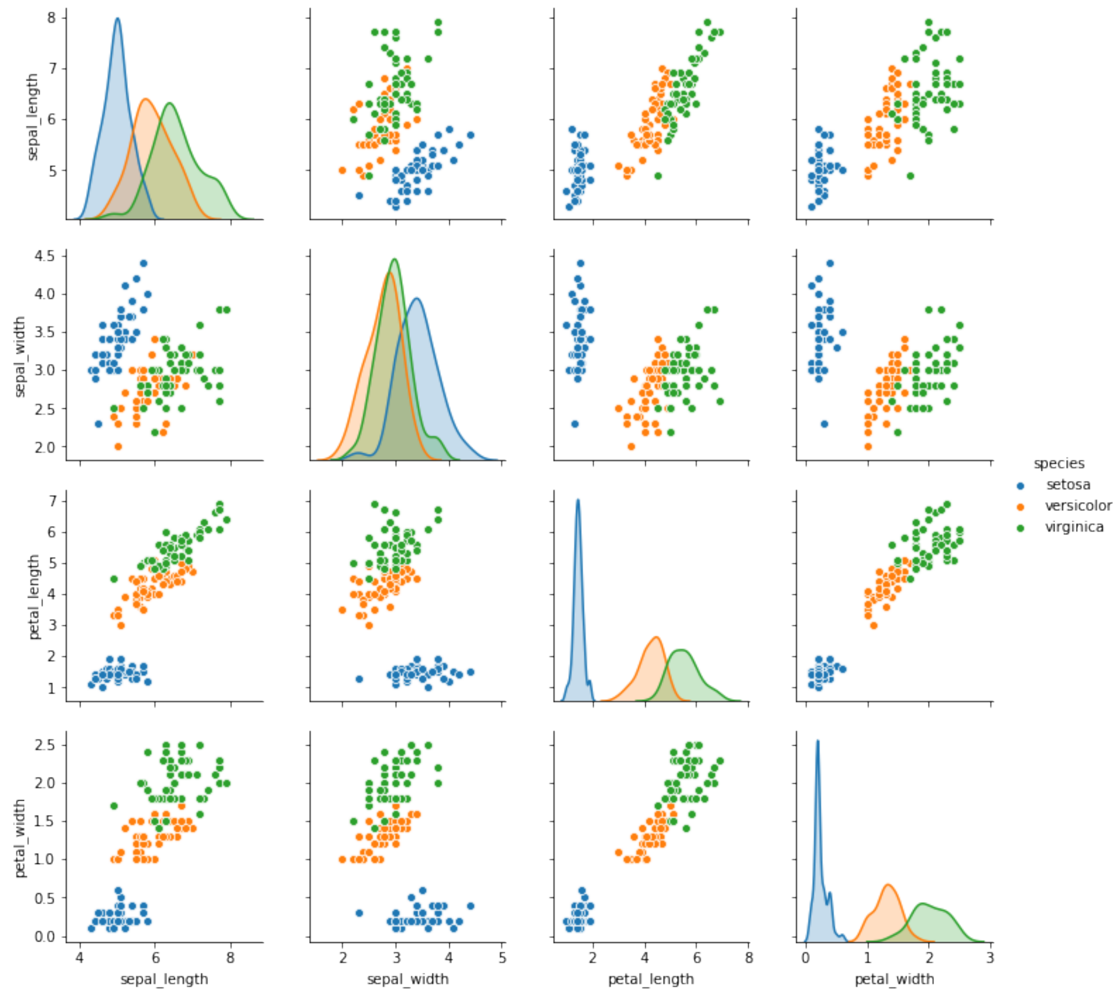
```
Out[42]: <seaborn.axisgrid.PairGrid at 0x1a1bece898>
```




```
In [43]: iris['petal_width'].hist(bins=20)

Out[43]: <matplotlib.axes._subplots.AxesSubplot at 0x1a1bece3c8>
```



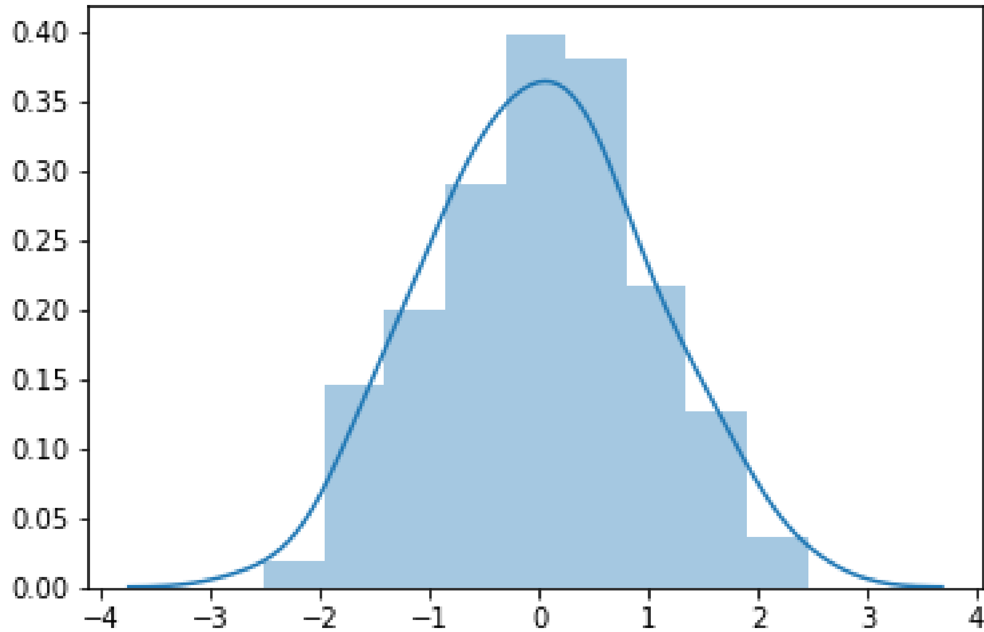

```
In [44]: sns.pairplot(iris, hue="species", height=2.5);
```



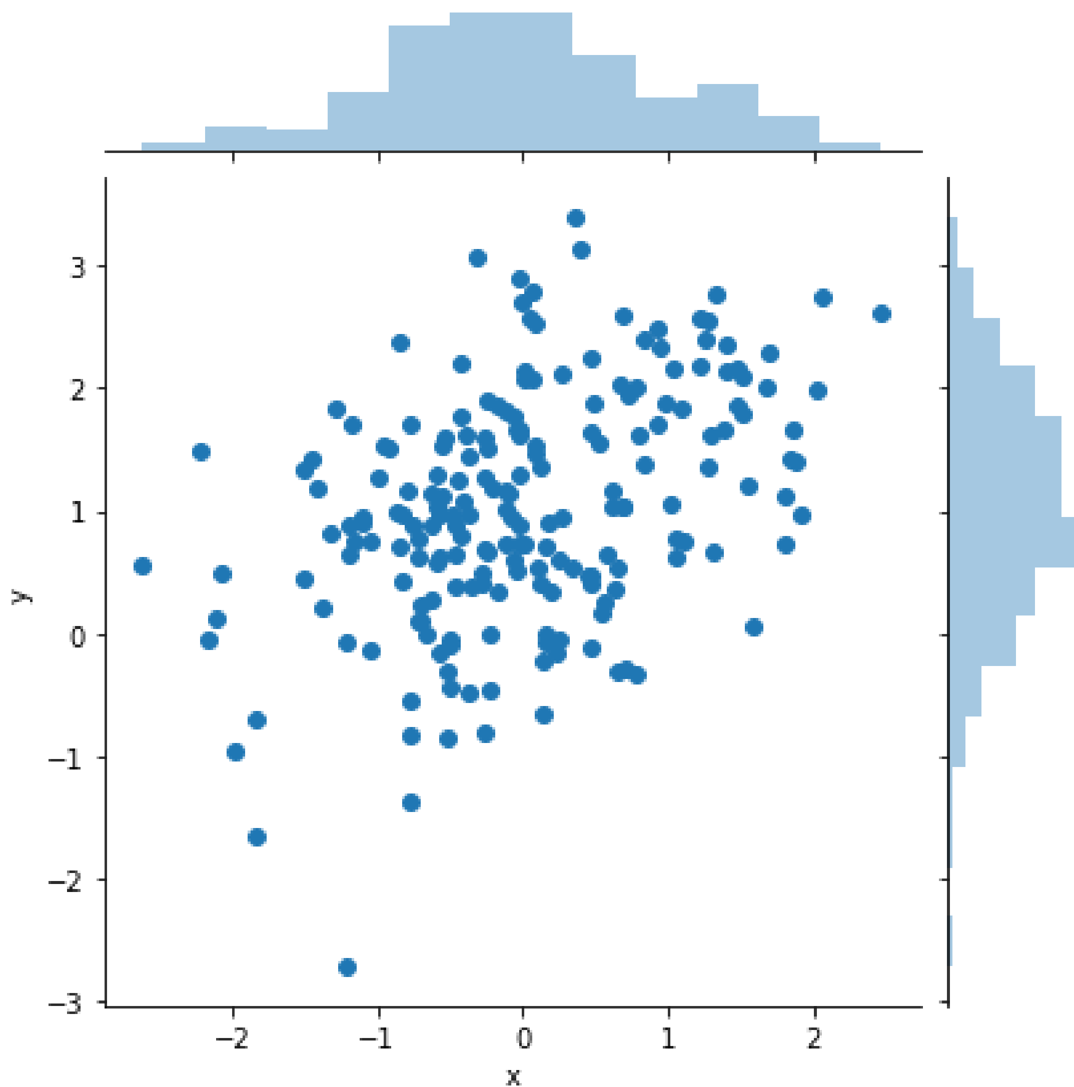
1.4.2 Simulações usando Numpy

Algo interessante a se aprender sobre os pacotes Numpy e Pandas é a simulação de dados i.i.d. de distribuições paramétricas conhecidas

```
In [45]: x = np.random.normal(size=100)
         sns.distplot(x);
```




```
In [46]: np.random.normal #apertar tab

Out[46]: <function RandomState.normal>

In [47]: mean, cov = [0, 1], [(1, .5), (.5, 1)]
         data = np.random.multivariate_normal(mean, cov, 200)
         df = pd.DataFrame(data, columns=["x", "y"]) #estruturando os dados como um Pandas Data

In [48]: df.head()

Out[48]:           x         y
         0 -0.028608  1.297633
         1 -0.547837  1.540190
         2 -0.032653  2.891551
         3  1.257769  2.390749
         4 -1.386497  0.220500

In [49]: # Outro gráfico interessante do Seaborn
         sns.jointplot(x="x", y="y", data=df);
```



1.5 4. Regressão

Para o exercícios seguintes vamos usar os dados sobre jogadores Premium League

```
In [50]: df = pd.read_csv('http://bit.ly/pd-premier-league')

In [51]: df.columns

Out[51]: Index(['name', 'club', 'age', 'position', 'position_cat', 'market_value',
                'page_views', 'fpl_value', 'fpl_sel', 'fpl_points', 'region',
                'nationality', 'new_foreign', 'age_cat', 'club_id', 'big_club',
                'new_signing'],
               dtype='object')
```

1.5.1 O que significa cada coluna?

- **name** : Nome do jogador
- **club** : Clube do jogador
- **age** : Idade do jogador
- **position** : Posição no campo no qual mais atua
- **position_cat** :
  - 1 atacante
  - 2 meio-campista
  - 3 defensor
  - 4 goleiro
- **market_value** : Valor em dólares no site transfermrkt.com em Julho de 2017
- **page_views** : Média de visitas diárias na página do Wikipedia entre Set 2016 e May 2017
- **fpl_value** : Valor no Fantasy Game da Premier League em Julho de 2017
- **fpl_sel** : % de times escalados com o jogador em Julho de 2017
- **fpl_points** : Pontos acumulados no Fantasy Game em 2016
- **region**:
  - 1 for Inglaterra
  - 2 for Europa
  - 3 for Américas
  - 4 for Restante do Mundo
- **nationality** : País de nascimento
- **new_foreign** : Se tá vindo de uma liga diferente nesse ano
- **age_cat** : Categorização da idade ("17-21","22-25","26-28","29-31","32-38")
- **club_id** : Identificador único do clube
- **big_club**: Se o clube é considerado um dos 6 grandes clubes (United, City, Chelsea, Arsenal, Liverpool and Tottenham)
- **new_signing**: Se assinou um novo contrato nesse ano.

In [52]:
```
df.head()
```

Out[52]:
```
                name     club  age position  position_cat  market_value  \
0     Alexis Sanchez  Arsenal   28       LW             1          65.0
1         Mesut Ozil  Arsenal   28       AM             1          50.0
2          Petr Cech  Arsenal   35       GK             4           7.0
3       Theo Walcott  Arsenal   28       RW             1          20.0
4  Laurent Koscielny  Arsenal   31       CB             3          22.0

   page_views  fpl_value fpl_sel  fpl_points  region     nationality  \
0        4329       12.0  17.10%         264     3.0           Chile
1        4395        9.5   5.60%         167     2.0         Germany
2        1529        5.5   5.90%         134     2.0  Czech Republic
3        2393        7.5   1.50%         122     1.0         England
4         912        6.0   0.70%         121     2.0          France

   new_foreign  age_cat  club_id  big_club  new_signing
```

```
0            0      4      1      1        0
1            0      4      1      1        0
2            0      6      1      1        0
3            0      4      1      1        0
4            0      4      1      1        0
```

In [53]:
```
df.describe()
```

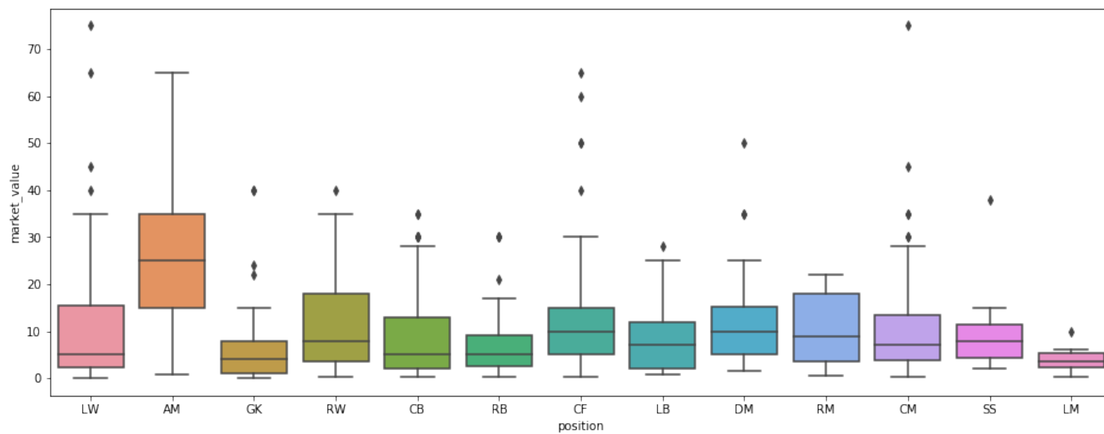
Out[53]:
```
              age  position_cat  market_value   page_views   fpl_value  \
count  461.000000    461.000000    461.000000   461.000000  461.000000
mean    26.804772      2.180043     11.012039   763.776573    5.447939
std      3.961892      1.000061     12.257403   931.805757    1.346695
min     17.000000      1.000000      0.050000     3.000000    4.000000
25%     24.000000      1.000000      3.000000   220.000000    4.500000
50%     27.000000      2.000000      7.000000   460.000000    5.000000
75%     30.000000      3.000000     15.000000   896.000000    5.500000
max     38.000000      4.000000     75.000000  7664.000000   12.500000

       fpl_points      region  new_foreign     age_cat     club_id  \
count  461.000000  460.000000   461.000000  461.000000  461.000000
mean    57.314534    1.993478     0.034707    3.206074   10.334056
std     53.113811    0.957689     0.183236    1.279795    5.726475
min      0.000000    1.000000     0.000000    1.000000    1.000000
25%      5.000000    1.000000     0.000000    2.000000    6.000000
50%     51.000000    2.000000     0.000000    3.000000   10.000000
75%     94.000000    2.000000     0.000000    4.000000   15.000000
max    264.000000    4.000000     1.000000    6.000000   20.000000

         big_club  new_signing
count  461.000000   461.000000
mean     0.303688     0.145336
std      0.460349     0.352822
min      0.000000     0.000000
25%      0.000000     0.000000
50%      0.000000     0.000000
75%      1.000000     0.000000
max      1.000000     1.000000
```

In [54]:
```
# um comando interessante para analisar se existem dados faltantes é o seguinte, util
df.apply(lambda x: sum(x.isnull()),axis=0)
```

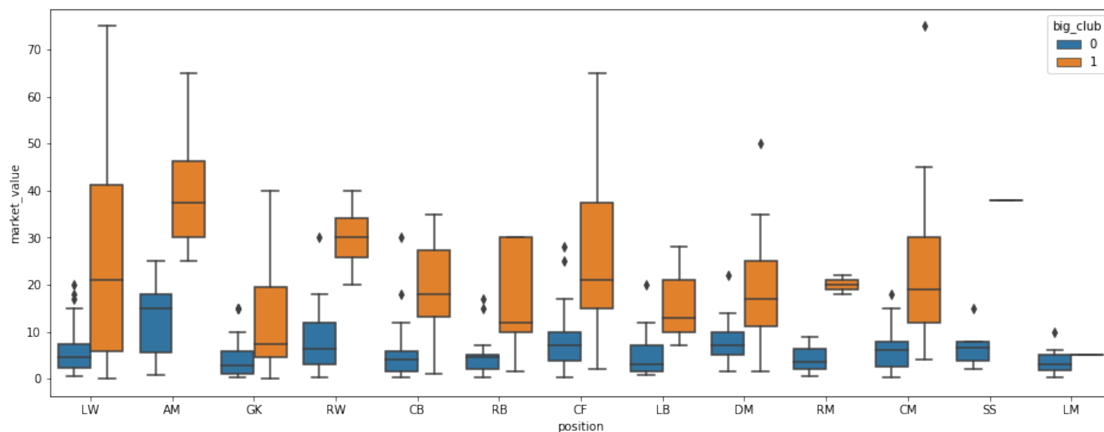
Out[54]:
```
name            0
club            0
age             0
position        0
position_cat    0
market_value    0
page_views      0
fpl_value       0
```

```
fpl_sel          0
fpl_points       0
region           1
nationality      0
new_foreign      0
age_cat          0
club_id          0
big_club         0
new_signing      0
dtype: int64
```

In [55]:
```
# Podemos fazer alguns boxplots interessantes para analisar o dataset
import matplotlib.pyplot as plt
plt.figure(figsize=(16, 6))
sns.boxplot(x='position',y='market_value',data=df);
```



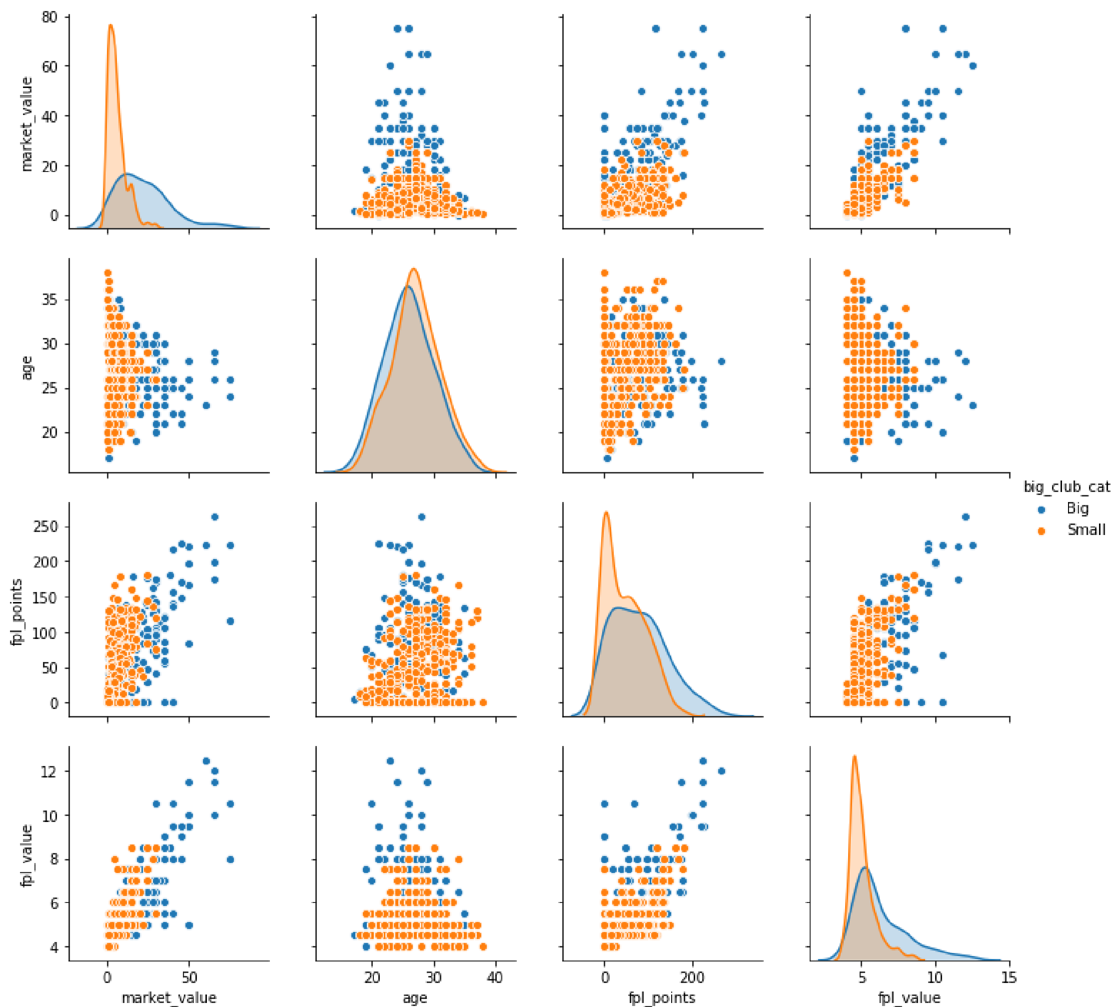
In [56]:
```
import matplotlib.pyplot as plt
plt.figure(figsize=(16, 6))
sns.boxplot(x='position',y='market_value',hue='big_club',data=df);
```



Vamos supor que queremos estimar o valor de mercado de um jogador. É muito provável que este dependa de algumas variáveis como *posição do jogador* (position_cat), *idade* (age), *pontos acumulados* (fpl_point). Para isso, podemos plotar alguns gráficos:

```
In [57]: # Criando uma função a ser aplicada por linha
         def variable(row):
             if row.big_club == 1:
                 return 'Big'
             else:
                 return 'Small'
         df["big_club_cat"] = df.apply(variable, axis=1)
```

```
In [58]: sns.pairplot(df[['market_value','age','fpl_points','fpl_value','big_club_cat']],hue='l
```



```
In [59]: from sklearn.linear_model import LinearRegression
         model = LinearRegression()
         x = df[['age','fpl_value','big_club']]
         y = df['market_value']
         fit = model.fit(x, y)
```

```
In [60]: print('Intercept: \n', fit.intercept_)
         print('Coefficients: \n', fit.coef_)
         print('R2: \n',fit.score(x, y))
```

```
Intercept:
 -21.423211406320924
Coefficients:
 [-0.0925666   5.94430027  8.33847031]
R2:
 0.7048801254252627
```

1.5.2 Uma outra alternativa é usar o Statsmodels que já retorna um resumo bem completo:

```
In [61]: import statsmodels.api as sm
         xlinha = sm.add_constant(x) # adding a constant
         model = sm.OLS(y, xlinha).fit()
         print_model = model.summary()
         print(print_model)
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:           market_value   R-squared:                       0.705
Model:                            OLS   Adj. R-squared:                  0.703
Method:                 Least Squares   F-statistic:                     363.8
Date:                Fri, 22 Nov 2019   Prob (F-statistic):          1.13e-120
Time:                        10:09:05   Log-Likelihood:                -1527.7
No. Observations:                 461   AIC:                             3063.
Df Residuals:                     457   BIC:                             3080.
Df Model:                           3
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const        -21.4232      2.595     -8.256      0.000     -26.522     -16.324
age           -0.0926      0.079     -1.167      0.244      -0.248       0.063
fpl_value      5.9443      0.256     23.251      0.000       5.442       6.447
big_club       8.3385      0.750     11.117      0.000       6.864       9.812
==============================================================================
Omnibus:                      151.227   Durbin-Watson:                   1.726
Prob(Omnibus):                  0.000   Jarque-Bera (JB):              919.433
Skew:                           1.275   Prob(JB):                    2.23e-200
Kurtosis:                       9.431   Cond. No.                         231.
```

```
==============================================================================

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.


/Users/palomavaissman/anaconda3/lib/python3.7/site-packages/numpy/core/fromnumeric.py:2389: Fu
  return ptp(axis=axis, out=out, **kwargs)
```

1.6 5. Fluxo de Aprendizagem de Máquina:

1.6.1 O fluxo de contrução de um modelo por ser sumarizado no que chamamos CRISP-DM:

Abreviação de **CRoss Industry Standard Process** for Data Mining, que pode ser traduzido como Processo Padrão Inter-Indústrias para Mineração de Dados. É um modelo de processo de mineração de dados que descreve abordagens comumente usadas por especialistas em mineração de dados para atacar problemas.

Ou seja, a construção de um modelo envolve etapas como: * **Entendimento do problema de negócio**: um modelo deve agregar valor ao negócio, resolvendo problemas. * **Preparação dos dados**: etapas de visualização, manipulação e tratamento dos dados. * **Modelagem**: escolha da metodologia adequada, treinamento, otimização e validação do modelo. * **Validação**: validação em forma de testes A/B ou outras formas para verificar se atende aos problemas do negócio. * **Produção**: modelo é de fato implantado em sistemas de forma que o cliente possa usar.

Dentro do Python, em geral, utilizam-se as bibliotecas Numpy e Pandas para preparação dos dados e o Scikit-Learn para a modelagem.

```
In [62]: import sklearn

In [63]: help sklearn?

In [ ]: help sklearn
```

Vamos utilizar alguns datasets do próprio **sklearn** para realizar as etapas de modelagem mais comuns. Para um modelo de classificação podemos usar o famoso dataset **MNIST**, composto por 70.000 imagens de dígitos escritos por estudantes de high school e empregados do US Census Bureau.

```
In [65]: from sklearn.datasets import fetch_openml
         mnist = fetch_openml('mnist_784',version=1)
         mnist.keys()

Out[65]: dict_keys(['data', 'target', 'feature_names', 'DESCR', 'details', 'categories', 'url']
```

Os datasets do sklearn tem todos a mesma estrutura: * Uma chave **DESCR** que descreve o dataset; * Uma chave **data** que contém um array de uma linha por **instância** e uma coluna por **feature** * Um **target** que contém um array com os **labels**

O dataset tem 70.000 imagens, cada uma com 28 x 28 pixels, cada uma das features representa uma intensidade do pixel, que pode variar entre 0 (branco) e 255 (preto).

```
In [66]: X, y = mnist['data'], mnist['target']

In [67]: X.shape

Out[67]: (70000, 784)

In [68]: 28*28

Out[68]: 784

In [69]: len(y)

Out[69]: 70000
```

Vamos agora treinar um modelo **Random Forest** com a base MNIST e também a iris. Este algoritmo tem a seguinte estrutura: divide-se aleatoriamente o dataset de treino em n amostras e treina-se uma árvore de decisão com cada uma das amostras. Feito isso, o algoritmo promove uma votação, ou seja, aceita a classificação que tiver maioria dentre as árvores.

```
In [70]: # o dataset MNIST já vem com as bases de treino e teste estruturadas,
         # respectivamente as primeiras 60 mil e as últimas 10 mil observações
         X_train,X_test,y_train,y_test = X[:60000],X[60000:],y[:60000],y[60000:]

In [71]: # Vamos treinar uma floresta aleatória com 500 árvores, cada uma com no máximo 16 nós
         from sklearn.ensemble import RandomForestClassifier
         rf = RandomForestClassifier(n_estimators=500,max_leaf_nodes=16,n_jobs=-1)
         rf.fit(X_train,y_train)
         y_pred = rf.predict(X_test)
```

1.6.2 Medidas de performance em modelos de classificação

Primeiramente, é importante entender a matriz de confusão de um classificador:

Temos, portanto, a medida de **acurácia** (*trueness*),

a medida de **precisão** (daqueles que classifiquei como corretos, quantos efetivamente eram?),

o **recall** (quando realmente é da classe X, o quão frequente você classifica como X?).

Todos esses são resumidos na medida F1:

```
In [72]: # Avaliando a performance da previsão
         from sklearn import metrics
         print("Accuracy:",metrics.accuracy_score(y_test, y_pred))
         #preciso usar uma média ponderada pois o target é multiclasse
         print("Precison:",metrics.precision_score(y_test, y_pred,average='weighted'))
         print("Recall:",metrics.recall_score(y_test, y_pred,average='weighted'))
         print("F1:",metrics.f1_score(y_test, y_pred,average='weighted'))

Accuracy: 0.8275
Precison: 0.8362710774952071
Recall: 0.8275
F1: 0.8215357878351431
```

Agora vamos aplicar o classificador usando o dataset iris

```
In [73]: iris.shape

Out[73]: (150, 5)

In [74]: iris.iloc[:,0].head()

Out[74]: 0    5.1
         1    4.9
         2    4.7
         3    4.6
         4    5.0
         Name: sepal_length, dtype: float64
```

Como falamos, podemos usar o dataset iris já em cache via sklearn. Nesse caso, o dataset já vem estruturado em **data** e **target**. Outra possibilidade é usar a função *iloc* no Pandas DataFrame. Vamos então fazer o upload usando o sklearn:

```
In [76]: from sklearn.datasets import load_iris
         iris = load_iris()

In [77]: # Neste caso as bases de treino e teste não vem estruturadas, podemos usar uma função
         from sklearn.model_selection import train_test_split
         X = iris['data']
         y = iris['target']
         X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, random_state

In [78]: rf = RandomForestClassifier(n_estimators=500,n_jobs=-1)
         rf.fit(X_train,y_train)
         y_pred = rf.predict(X_test)
         print("Accuracy:",metrics.accuracy_score(y_test, y_pred))
         print("Recall:",metrics.recall_score(y_test, y_pred,average='weighted'))
         print("F1:",metrics.f1_score(y_test, y_pred,average='weighted'))

Accuracy: 0.98
Recall: 0.98
F1: 0.98
```

Algo interessante no modelo Random Forest é que este mostra o que chamamos de **Feature Importance**, que não são coeficientes como uma regressão, mas qual a importância percentual de cada **feature** na previsão do **target**. Com os dicionários contidos na estrutura de dados do **sklearn** podemos visualizar esses percentuais

```
In [79]: for name,score in zip(iris['feature_names'],rf.feature_importances_):
             print(name,score)
```

```
sepal length (cm) 0.09738530025794011
sepal width (cm) 0.03607559613096327
petal length (cm) 0.41509660404447224
petal width (cm) 0.45144249956662436
```

Por último, podemos usar a técnica de **cross-validation** (CV) para validar o modelo ainda mais, prevenindo o que chamamos de **overfitting**:

O CV é uma técnica que faz esse split de bases treino/teste várias vezes. No caso do CV K-fold, divide-se o dataset em K amostras (folds):

Podemos assim obter métricas após o CV, e isso é uma boa prática:

```
In [80]: from sklearn.model_selection import cross_val_score
         scores = cross_val_score(rf, iris.data, iris.target, cv=5)
         scores

Out[80]: array([0.96666667, 0.96666667, 0.93333333, 0.93333333, 1.        ])
```

Como default é utilizar o score da técnica (geralmente acurácia), podemos especificar se queremos uma outra métrica após o cross-validation, por exemplo:

```
In [81]: scores = cross_val_score(rf,iris.data, iris.target, cv=5, scoring='f1_macro')
         scores

Out[81]: array([0.96658312, 0.96658312, 0.93265993, 0.93333333, 1.        ])
```

É possível utilizar CV para fazer o que chamamos de **tuning de hiperparâmetros**, por exemplo, o número de árvores.

```
In [82]: rf_plain = RandomForestClassifier(random_state = 42)
         from pprint import pprint
         # Lista de hiperparâmetros possíveis
         print('Parameters currently in use:\n')
         pprint(rf_plain.get_params())

Parameters currently in use:

{'bootstrap': True,
 'class_weight': None,
 'criterion': 'gini',
 'max_depth': None,
 'max_features': 'auto',
 'max_leaf_nodes': None,
 'min_impurity_decrease': 0.0,
 'min_impurity_split': None,
 'min_samples_leaf': 1,
 'min_samples_split': 2,
 'min_weight_fraction_leaf': 0.0,
 'n_estimators': 'warn',
```

```
 'n_jobs': None,
 'oob_score': False,
 'random_state': 42,
 'verbose': 0,
 'warm_start': False}
```

```
In [83]: from sklearn.model_selection import GridSearchCV

         # Grid de parâmetros
         param_grid = {
             'bootstrap': [True],
             'max_depth': [10, 50, 100],
             'n_estimators': [100, 500,1000]
         }
         # Create a based model
         rf_plain = RandomForestClassifier()

         # Instantiate the grid search model
         grid_search = GridSearchCV(estimator = rf_plain, param_grid = param_grid,
                                   cv = 3, n_jobs = -1, verbose = 2)

         grid_search.fit(X, y)
         grid_search.best_params_
```

```
Fitting 3 folds for each of 9 candidates, totalling 27 fits


[Parallel(n_jobs=-1)]: Using backend LokyBackend with 4 concurrent workers.
[Parallel(n_jobs=-1)]: Done  20 out of  27 | elapsed:    2.8s remaining:    1.0s
[Parallel(n_jobs=-1)]: Done  27 out of  27 | elapsed:    4.2s finished
/Users/palomavaissman/anaconda3/lib/python3.7/site-packages/sklearn/model_selection/_search.py
  DeprecationWarning)
```

```
Out[83]: {'bootstrap': True, 'max_depth': 50, 'n_estimators': 100}
```

```
In [84]: rf_final = RandomForestClassifier(n_estimators=100,max_depth=10,n_jobs=-1)
         rf_final.fit(X_train,y_train)
         y_pred = rf_final.predict(X_test)
         print("Accuracy:",metrics.accuracy_score(y_test, y_pred))
         print("Recall:",metrics.recall_score(y_test, y_pred,average='weighted'))
         print("F1:",metrics.f1_score(y_test, y_pred,average='weighted'))
```

```
Accuracy: 0.98
Recall: 0.98
F1: 0.98
```

```
In [ ]:
```