PACKAGES FOR MACHINE LEARNING

Pandas Package

The Pandas package - A 1st view[1]

- Pandas (panel data analysis), being a newer package, is supported in NumPy itself.
- One of its main features is dataframes

  (type of structure already familiar to R programmers)

  - *these are objects that represent two-dimensional data properly labeled or labeled, somewhat similar to the tables we get used to using in other environments, such as Excel, SQL, etc.*
  - *provide us, in essence, with an interface for data, which greatly facilitates its representation and manipulation*
- A DataFrame takes on the following aspect:

```
import pandas as pd
df=pd.DataFrame({'label':['A','B','C','A','B','C'], 'value':[1,2,3,4,5,6]})
df
```

	label	value
0	A	1
1	B	2
2	C	3
3	A	4
4	B	5
5	C	6

Note that the initial content is provided to the DataFrame in dictionary form

[1] *With content adapted from "A Whirlwind Tour of Python", of Jake Vanderplas, O'Reilly, 2016 (with licensing CC0)*

A first view of Pandas[1]

- With this interface for the data, we may simply perform a huge set of operations, such as,
    - *select columns by name:*

```
df['label']
```

```
0    A
1    B
2    C
3    A
4    B
5    C
```

- *select lines with the method loc()*

```
df.loc[2]
```

```
label    C
value    3
```

- *apply string manipulation operations to columns with data of this type:*

```
df['label'].str.lower()
```

```
0    a
1    b
2    c
3    a
4    b
5    c
```

A first view of Pandas[1]

- *apply aggregation functions to columns with numeric data:*

```
df['value'].sum()
```

```
21
```

- *and, not least, perform aggregation operations by category in a specific column:*

```
df.groupby('label').sum()
```

label	value
A	5
B	7
C	9

In the last example, the sums of all the values that share the same label are calculated on each line, something that would be much more laborious (and less efficient) if we used the features of Python or even NumPy.

Understanding Pandas Better

- In most of the problems addressed in Data Science, the data under analysis are presented in the form of
  1. *where each column has a name that identifies it and a data type of its own (i.e., each column can store a different data type)*

     (this column characteristics cannot be ensured by NumPy arrays, as you easily see)
  2. *and where each row contains data for an object, instance, individual, record or observation*

nome	idade	presencas	notaIA
Ana	20	20	12.2
Rui	25	3	5.3
Gil	27	28	15.7
Zé	23	17	15.9
Tó	20	28	19.2

- To easily handle data in the form of tables, Pandas introduces a new data type: the DataFrame
  - *it is a data structure that allows you to manipulate in a simple, efficient and flexible way data tables in python environment*

Series

- In addition to DataFrames, Pandas introduces another data structure: the **series**
  - *a series is similar to a one-dimensional NumPy array*
  - *but where its elements, being by default indexed by an integer according to their position (0, 1, 2, …), can also be indexed by custom indexes by the user*

```
s0=pd.Series([0,10,20,30,40]); print(s0)
```

```
0     0
1    10
2    20
3    30
4    40
```

```
s1=pd.Series([0,10,20,30,40], index=['A','A','B','C','D']); print(s1)
```

```
A     0
A    10
B    20
C    30
D    40
```

imagine defining new integer indexes, but not starting at 0

- Access to the elements of a series is similar to arrays
  - *position or label can be used, with the* ***iloc()*** *and* ***loc()*** *methods if necessary*

```
print(s1[2], s1['B'], s1.iloc[2], s1.loc['B'])
```

```
20 20 20 20
```

  - *and slicing operations are also possible*

```
print(s1[2:4])
```

```
B    20
C    30
```

```
print(s1.iloc[2:4])
```

```
B    20
C    30
```

And when there are repeated labels?

```
print(s1['A'])
```

```
A     0
A    10
```

Time series

- A **time series** can be understood as a series in which its elements are indexed by indexes that represent moments of time
- Suppose we want to record our daily weight over the course of a week
  - *We can then build a time series, starting with the definition of the set of indexes to be used, through the* ***date_range()***

```
dias = pd.date_range('26-10-2020', periods=7); print(dias)
```

number of the items create

```
DatetimeIndex(['2020-10-26', '2020-10-27', '2020-10-28', '2020-10-29',
               '2020-10-30', '2020-10-31', '2020-11-01'],
              dtype='datetime64[ns]', freq='D')
```

1º Instant

'D': daily frequency. There are many other periodicities that can be chosen

  - *Assuming that the following values correspond to the 7 weighings*

```
pesagens=np.round(np.random.random(7)+75,2) #simulação de valores
print(pesagens)
```

```
[75.48 75.71 75.17 75.73 75.21 75.59 75.57]
```

  - *easily create the corresponding time series*

```
pesos=pd.Series(pesagens, index=dias); print(pesos)
```

```
2020-10-26    75.48
2020-10-27    75.71
2020-10-28    75.17
2020-10-29    75.73
2020-10-30    75.21
2020-10-31    75.59
2020-11-01    75.57
Freq: D, dtype: float64
```

Indexes with date/time

- In addition to the date, we may also include the time in the indexes of a time series
  - *As an example, let's create a set of indexes with date/time*

```
horas = pd.date_range('26-10-2020 15:20', periods=7); print(horas)
```

```
DatetimeIndex(['2020-10-26 15:20:00', '2020-10-27 15:20:00',
               '2020-10-28 15:20:00', '2020-10-29 15:20:00',
               '2020-10-30 15:20:00', '2020-10-31 15:20:00',
               '2020-11-01 15:20:00'],
              dtype='datetime64[ns]', freq='D')
```

In addition to this type of format, Pandas can interpret several other date and time formats

  - *We can now assign this new set of indexes to the existing weight series*

```
pesos.index=horas; print(pesos)
```

```
2020-10-26 15:20:00    75.48
2020-10-27 15:20:00    75.71
2020-10-28 15:20:00    75.17
2020-10-29 15:20:00    75.73
2020-10-30 15:20:00    75.21
2020-10-31 15:20:00    75.59
2020-11-01 15:20:00    75.57
Freq: D, dtype: float64
```

- Note that while weights are a series, days and hours are just indexes

```
print(type(pesos),type(dias),type(horas), sep='\n')
```

```
<class 'pandas.core.series.Series'>
<class 'pandas.core.indexes.datetimes.DatetimeIndex'>
<class 'pandas.core.indexes.datetimes.DatetimeIndex'>
```

Let's go back to DataFrames

- As previously said, a **DataFrame** allows us to represent a data table in a Python environment

DataFrame

Columns

Indexes

	nome	idade	presencas	notaIA
0	Ana	20	20	12.2
1	Rui	25	3	5.3
2	Gil	27	28	15.7
3	Zé	23	17	15.9
4	Tó	20	28	19.2

Lines

Series

Series

Series

Series

As illustrated, each individual column forms, with the index column, a series

- The DataFrame is the ideal framework for representing data science and ML data
  - *faithfully models the way data is represented in real life*
    - it is customary for data processed in ML to be found in SQL tables, in tables stored in CSV files, or even in Excel tables
- That's why Panda DataFrames are so used in ML
  - *It is typically in this format that the ML algorithms of the Scikit-learn package expect to receive the data*

How to create a DataFrame

- The previously illustrated DataFrame can be created by instantiating the Pandas DataFrame class,
  - *providing the contents of the table in dictionary form*

```
alunos = pd.DataFrame({'nome': ['Ana', 'Rui', 'Gil', 'Zé', 'Tó'],
'idade': [20,25,27,23,20], 'presencas': [20,3,28,17,28], 'notaIA':
[12.2,5.3,15.7,15.9,19.2]})
```

  - *or by providing the data and column names in separate lists*

```
data=[['Ana',20,20,12.2], ['Rui',25,3,5.3], ['Gil',27,28,15.7],
['Zé',23,17,15.9], ['Tó',20,28,19.2]]
lables=['nome', 'idade', 'presencas', 'notaIA']
```

```
alunos=pd.DataFrame(data, columns=lables)
```

- However, because the datasets used in ML are usually quite big, DataFrames end up, for the most part, being automatically loaded from an external source
  - *it is often uploaded, for example, to load a DataFrame from a CSV text file using the read_csv()*

```
alunos=pd.read_csv('dados.csv')
```

In case the contents of the table are in the 'dados.csv' file

Customize indexes in a DataFrame(DF)

- As with series, we may customize the indexes of a DF
  - *In df students we can, for example, use for indexing the lines the mechanographic numbers of the respective students*

```
alunos.index=[31234,33333,40000,44444,30000]
```

```
alunos
```

	nome	idade	presencas	notaIA
31234	Ana	20	20	12.2
33333	Rui	25	3	5.3
40000	Gil	27	28	15.7
44444	Zé	23	17	15.9
30000	Tó	20	28	19.2

- If we show the df students again, it is perceived that the lines have effectively started to have a new form of indexing

- Through the attribute's 'index', 'columns' and 'values' we can easily consult (or change) the 3 important components of DF

```
alunos.index
```

```
Int64Index([31234, 33333, 40000, 44444, 30000], dtype='int64')
```

```
alunos.columns
```

```
Index(['nome', 'idade', 'presencas', 'notaIA'], dtype='object')
```

```
alunos.values
```

```
array([['Ana', 20, 20, 12.2],
       ['Rui', 25, 3, 5.3],
       ['Gil', 27, 28, 15.7],
       ['Zé', 23, 17, 15.9],
       ['Tó', 20, 28, 19.2]], dtype=object)
```

Quick query of a DataFrame

- As the datasets treated in ML are large, very rarely do you choose to show the totality of the lines that make up the corresponding DF
  - *It is often enough to consult some of your rows for a first check and validation of the table contents*
  - *To this end, the DFs have the two methods,* ***head()*** *and* ***tail()****, which allow us to quickly consult the contents of the first and last lines, respectively.*

```
alunos.head(2)
```

	nome	idade	presencas	notaIA
31234	Ana	20	20	12.2
33333	Rui	25	3	5.3

```
alunos.tail(2)
```

	nome	idade	presencas	notaIA
44444	Zé	23	17	15.9
30000	Tó	20	28	19.2

  - *Both the head() and the tail(), when invoked without arguments, present, by default, 5 lines.*

- The DFs also contain a very useful method, the **describe()**, which allows to generate various statistical data related to the numerical values saved:
  - *quantity of items, mean, standard deviation, minimum value, 1st quartile, median, 3rd quartile and maximum value*

```
alunos.describe()
```

	idade	presencas	notaIA
count	5.000000	5.000000	5.000000
mean	23.000000	19.200000	13.660000
std	3.082207	10.281051	5.288951
min	20.000000	3.000000	5.300000
25%	20.000000	17.000000	12.200000
50%	23.000000	20.000000	15.700000
75%	25.000000	28.000000	15.900000
max	27.000000	28.000000	19.200000

How to Select Specific Columns or Rows

- One or more specific columns can be selected using their names as an index

```
alunos['idade']
```

```
31234    20
33333    25
40000    27
44444    23
30000    20
```

```
alunos[['nome','idade']]
```

	nome	idade
31234	Ana	20
33333	Rui	25

- We can also access the column through its attribute

```
alunos.idade
```

```
31234    20
33333    25
40000    27
44444    23
30000    20
```

  - *When we select a single column, we get a serie as a result; but when several are selected, the result is already a DF*

- Selecting specific rows requires using the **loc()** and **iloc()** methods
  - *loc() if the respective label is used as an index*
  - *iloc() If the relative position is used as an index*

```
alunos.loc[40000] # o mesmo que alunos.iloc[2]
```

```
nome         Gil
idade         27
presencas     28
notaIA      15.7
```

```
alunos.iloc[[1,3]] #o mesmo que alunos.loc[[33333,44444]]
```

	nome	idade	presencas	notaIA
33333	Rui	25	3	5.3
44444	Zé	23	17	15.9

Slicing in DataFrames

- The slicing operation in DF can be carried out based on,

lines,

```
alunos.iloc[1:3,:]
```

	nome	idade	presencas	notaIA
33333	Rui	25	3	5.3
40000	Gil	27	28	15.7

or in both

```
alunos.iloc[1:3,2:]
```

	presencas	notaIA
33333	3	5.3
40000	28	15.7

or in columns

```
alunos.iloc[:,0:2]
```

	nome	idade
31234	Ana	20
33333	Rui	25
40000	Gil	27
44444	Zé	23
30000	Tó	20

- In the slicing operation, index labels can also be used instead of positions on both axes (with the loc() method, of course)

```
alunos.loc[33333:40000,'presencas':]
```

	presencas	notaIA
33333	3	5.3
40000	28	15.7

But be careful: if we use labels in the slicing operation, the upper limit is included in the selection

Boolean indexing and individual cells

- If the goal is to select lines from a DF based on the value of cells, we can always use Boolean indexing
  - *For example, students over the age of 20 under 25 are as follows:*

```
alunos[(alunos.presencas<25) & (alunos.idade>20)]
```

	nome	idade	presencas	notaIA
33333	Rui	25	3	5.3
44444	Zé	23	17	15.9

- Of course, you can also access the value of a single cell
  - *For example, if we want to know the 40000 student grade, we can use the at() method*

```
alunos.at[40000,'notaIA']  #o mesmo que alunos.loc[40000,'notaIA'] e que alunos.iloc[2,3]
```

```
15.7
```

For the rest, the at() method is more efficient because it only allows access to one element

Transposed DataFrame

- As in arrays, you can use the transpose() method in a DF if you want to swap the rows for the columns
  - *This method can also be accessed through the T property*

```
alunos.T  #o mesmo que alunos.transpose()
```

	31234	33333	40000	44444	30000
nome	Ana	Rui	Gil	Zé	Tó
idade	20	25	27	23	20
presencas	20	3	28	17	28
notaIA	12.2	5.3	15.7	15.9	19.2

Sort a DataFrame by indexes

1. With the method sort_index(), we can sort the rows of a DF by the index column

```
alunos.sort_index(axis=0)
```

	nome	idade	presencas	notaIA
30000	Tó	20	28	19.2
31234	Ana	20	20	12.2
33333	Rui	25	3	5.3
40000	Gil	27	28	15.7
44444	Zé	23	17	15.9

2. But we can also sort the DF columns by the header row (column names)

```
alunos.sort_index(axis=1)
```

	idade	nome	notaIA	presencas
31234	20	Ana	12.2	20
33333	25	Rui	5.3	3
40000	27	Gil	15.7	28
44444	23	Zé	15.9	17
30000	20	Tó	19.2	28

3. However, the sort_index() returns an ordered DF, not changing the original DF

```
alunos
```

	nome	idade	presencas	notaIA
31234	Ana	20	20	12.2
33333	Rui	25	3	5.3
40000	Gil	27	28	15.7
44444	Zé	23	17	15.9
30000	Tó	20	28	19.2

4. To change the original DF, in this, as in other functions, we must use 'inplace'

```
alunos.sort_index(axis=0, inplace=True)
alunos
```

	nome	idade	presencas	notaIA
30000	Tó	20	28	19.2
31234	Ana	20	20	12.2
33333	Rui	25	3	5.3
40000	Gil	27	28	15.7
44444	Zé	23	17	15.9

Sort a DataFrame by the values

- With the method sort_values() we were able to sort a DF by the values of the cells
  - *To sort the rows of a DF by the values of a specific column, we only have to provide the name of that column*

```
alunos.sort_values('notaIA', ascending=False)
```

	nome	idade	presencas	notaIA
30000	Tó	20	28	19.2
44444	Zé	23	17	15.9
40000	Gil	27	28	15.7
31234	Ana	20	20	12.2
33333	Rui	25	3	5.3

in this example, students are ordered in descending order of their grade

In this, as in the other sortmethods, we can reverse the order with the parameter 'ascending'

  - *In ordering the columns of a DF by the values of a specific row, we must indicate their label and choose the second axis (axis=1)*
    - But this ordering only makes sense if all the columns are of the same type

we anonymize the DF students so that they are only numerical columns, so that we can sort

```
anonimos=alunos.iloc[:,1:]
```

	idade	presencas	notaIA
30000	20	28	19.2
31234	20	20	12.2
33333	25	3	5.3
40000	27	28	15.7
44444	23	17	15.9

```
anonimos.sort_values(33333, axis=1)
```

	presencas	notaIA	idade
30000	28	19.2	20
31234	20	12.2	20
33333	3	5.3	25
40000	28	15.7	27
44444	17	15.9	23

even so, it won't make much sense to sort values with different meanings

Apply a function to a DataFrame

With the apply() method, you can apply a function to the values of a df column,

1. whether it's a function of ours

```
quad = lambda x: x**2
```

```
alunos.presencas.apply(quad)
```

```
30000    784
31234    400
33333      9
40000    784
44444    289
```

2. whether it's a Python function

```
alunos.notaIA.apply(round)
```

```
30000    19
31234    12
33333     5
40000    16
44444    16
```

If the function has additional parameters, they can be provided via a tuplo. For example, to round with 1 decimal place:

```
alunos.notaIA.apply(round, args=(1,))
```

3. It should be noted, however, that df is not changed

```
alunos
```

	nome	idade	presencas	notaIA
30000	Tó	20	28	19.2
31234	Ana	20	20	12.2
33333	Rui	25	3	5.3
40000	Gil	27	28	15.7
44444	Zé	23	17	15.9

4. But if the goal is to change the DF, reflecting in itself the results of the application of the function, just assign those results to the respective column
   - *let's start by duplicating the df students, through an in-depth copy*

```
al=alunos.copy()
```

If we wanted a shallow copy, we could use the 'deep' parameter:

```
#al=alunos.copy(deep=False)
```

Apply a function to a DataFrame

5. The rounding of the df 'al' notes, with alteration of the DF itself, would be made as follows

```
al.notaIA=al.notaIA.apply(round)
```

```
al
```

	nome	idade	presencas	notaIA
30000	Tó	20	28	19
31234	Ana	20	20	12
33333	Rui	25	3	5
40000	Gil	27	28	16
44444	Zé	23	17	16

6. If we want to apply a function to multiple columns of a DF, or to all, we can do so by iteseeing

```
for c in alunos.columns[1:]:
    al[c]=al[c].apply(quad)
```

```
al
```

	nome	idade	presencas	notaIA
30000	Tó	400	784	361
31234	Ana	400	400	144
33333	Rui	625	9	25
40000	Gil	729	784	256
44444	Zé	529	289	256

7. Finally, it is finally notethat it is also possible to apply a function to a row, rather than to a column, simply by indexing the row with the loc method (providing the label) or iloc (providing the position)

```
anonimos.loc[40000].apply(quad)
```

```
idade        729.00
presencas    784.00
notaIA       246.49
```

```
anonimos.iloc[3].apply(quad)
```

```
idade        729.00
presencas    784.00
notaIA       246.49
```

Insert or remove a column in a DataFrame

1. Suppose that you want to add a new column to the DF, indicating whether or not the student has a previous year's attendance

```
alunos['freqAnt']=[False,False,True,False,True]
```

	nome	idade	presencas	notaIA	freqAnt
30000	Tó	20	28	19.2	False
31234	Ana	20	20	12.2	False
33333	Rui	25	3	5.3	True
40000	Gil	27	28	15.7	False
44444	Zé	23	17	15.9	True

2. And if you want the column to be inserted before the notes column?
   - *For this we use the insert() method*
   - *But let's start by removing the column that was inserted*

```
alunos.drop('freqAnt', axis=1, inplace=True)
alunos
```

	nome	idade	presencas	notaIA
30000	Tó	20	28	19.2
31234	Ana	20	20	12.2
33333	Rui	25	3	5.3
40000	Gil	27	28	15.7
	Zé	23	17	15.9

3. Let us then reinsert the column, but now in position 3

```
alunos.insert(3,'freqAnt',[False,False,True,False,True])
alunos
```

	nome	idade	presencas	freqAnt	notaIA
30000	Tó	20	28	False	19.2
31234	Ana	20	20	False	12.2
33333	Rui	25	3	True	5.3
40000	Gil	27	28	False	15.7
44444	Zé	23	17	True	15.9

The drop() method, like others, is limited to returning a new DF as a result of.

- If the original DF is to be changed, it is necessary to use the 'inplace' parameter

If you want to delete multiple columns, simply provide the drop() method with a list of the names of those columns

Insert or remove a line in a DataFrame

1. Inserting a row, by indexing, requires using the loc() method with the new label

```
alunos.loc[34567]=['Ivo',21,27,False,14]
```

	nome	idade	presencas	freqAnt	notaIA
30000	Tó	20	28	False	19.2
31234	Ana	20	20	False	12.2
33333	Rui	25	3	True	5.3
40000	Gil	27	28	False	15.7
44444	Zé	23	17	True	15.9
34567	Ivo	21	27	False	14.0

2. When removing a line, it is sufficient to pass the label of the same

```
alunos.drop(33333)
```

	nome	idade	presencas	freqAnt	notaIA
30000	Tó	20	28	False	19.2
31234	Ana	20	20	False	12.2
40000	Gil	27	28	False	15.7
44444	Zé	23	17	True	15.9
34567	Ivo	21	27	False	14.0

- O método drop(), por defeito, remove linhas (axis=0 opcional)
- Ao não usarmos 'inplace', a remoção não se efetiva no próprio DF

3. Sometimes it is necessary to remove rows by the values of your cells
   - *Suppose, for example, that it is intended to remove from the DF students who fail for absences*
   - *which will be equivalent to selecting only the remaining*
   - *i.e. who have more than 24 attendances or who already have a frequency of the previous year*

```
alunos[(alunos.presencas>=25) | alunos.freqAnt]
```

	nome	idade	presencas	freqAnt	notaIA
30000	Tó	20	28	False	19.2
33333	Rui	25	3	True	5.3
40000	Gil	27	28	False	15.7
44444	Zé	23	17	True	15.9
34567	Ivo	21	27	False	14.0

Cross-reference table with DataFrames

- A cross-reference table shows the distribution of two or more categorical variables in a grouped way, enabling a quick consultation of the possible relationships between them
    - *To see the applicability of this type of table, let's start by adding two new columns of categorical (non-numeric) values to the student DF*

```
alunos.insert(1,'freq',['ordin','trab','erasm','ordin','ordin','trab'])
alunos['classif']=['Aprovado','reprovado','reprovado','Aprovado','Aprovado','Aprovado']
   alunos
```

	nome	freq	idade	presencas	freqAnt	notaIA	classif
30000	Tó	ordin	20	28	False	19.2	Aprovado
31234	Ana	trab	20	20	False	12.2	reprovado
33333	Rui	erasm	25	3	True	5.3	reprovado
40000	Gil	ordin	27	28	False	15.7	Aprovado
44444	Zé	ordin	23	17	True	15.9	Aprovado
34567	Ivo	trab	21	27	False	14.0	Aprovado

- Using these two columns in the crosstab() method, we can quickly see how the student's frequency type will be related to approval

```
   pd.crosstab(alunos.freq,alunos.classif)
```

classif	Aprovado	reprovado
freq		
erasm	0	1
ordin	3	0
trab	1	1

solve exercise #12

from the book of exercises