

# Home

## ENVS615 - Data Science Studio

- [Dani Arribas-Bel. \[@darribas\].](#)

## Contact

Dani Arribas-Bel - D.Arribas-Bel [at] liverpool.ac.uk  
Senior Lecturer in Geographic Data Science  
Office 508, Roxby Building,  
University of Liverpool - 74 Bedford St S,  
Liverpool, L69 7ZT,  
United Kingdom.

## Overview

### Delivery modes

The course is delivered following a flipped, hybrid model by which some of the content is available to students *before* class time, and some is delivered in group sessions together with the instructor. The former is called *asynchronous* delivery, while the latter is referred to as *synchronous*.

Throughout the course content, each learning object (typically subsections in the materials) is tagged with one of the following two labels for reference:

- [Async] To be completed asynchronously, on your own, and *before* class
- [Sync] To be completed synchronously, with course instructor, remotely through video link

Similarly, the overview below contains them for a quick, general sense at what is expected from you before we meet in synchronous sessions.

## Schedule

### Day 1 - Introduction

- [Async] Introduction to the course: [data, data, data](#)
- [Async] [Computational building blocks](#)

### Day 2 - Tabular data

- [Sync] [Manipulation](#)
- [Sync] [Visualisation](#)
- [Sync] [Advanced manipulation](#)

### Day 3 - Unsupervised learning

- [Unsupervised learning](#)

☰ Contents

Course Specs

[Overview](#)

[Infrastructure](#)

[Data](#)

[Assessment](#)

[Bibliography](#)

Introduction

["Data, data, data"](#)

[Computational building blocks](#)

Data Wrangling

[Reading & Manipulating](#)

[Tabular Data](#)

[Visualising Tabular Data](#)

[Advanced Tabular](#)

[Manipulation](#)

Machine Learning

[Unsupervised Learning](#)

[Supervised Learning](#)

[Inference](#)

[Overfitting & Cross-Validation](#)

- [Async] [Overview](#)
- [Sync] [Hands-on](#)

## Day 4 - Supervised learning

- [Introduction](#)
  - [Async] [Overview](#)
  - [Sync] [Hands-on](#)
- [Sync] [Inference](#)
- [Sync] [Overfitting & cross-validation](#)

## Day 5

- [Data Studio](#)
  - [Async] [Assignment brief](#)
  - [Sync] [Studio time](#)

## Programming Language

This course is entirely taught in the [Python](#) programming language.

Source: [XKCD](#)

There are several reasons why we have made this choice. Many of them are summarised nicely in [this article by The Economist](#).

## Infrastructure

This page covers a few technical aspects on how the course is built, kept up to date, and how you can create a computational environment to run all the code it includes.

## Software stack

This course is best followed if you can not only read its content but also interact with its code and even branch out to write your own code and play on your own. For that, you will need to have installed on your computer a series of interconnected software packages; this is what we call a *stack*.

Instructions on how to install a software stack that allows you to run the materials of this course depend on the operating system you are using. Detailed guides are available for the main systems on the following resource, provided by the [Geographic Data Science Lab](#):



## Github repository

All the materials for this course and this website are available on the following Github repository:



If you are interested, you can download a compressed .zip file with the most up-to-date version of all the materials, including the HTML for this website at:



## Containerised backend

The course is developed, built and tested using the [gds env](#), a containerised platform for Geographic Data Science. You can read more about the [gds env](#) project at:



## Binder

[Binder](#) is a service that allows you to run scientific projects in the cloud for free. Binder can spin up “ephemeral” instances that allow you to run code on the browser without any local setup. It is possible to run the course on Binder by clicking on the button below:



### Warning

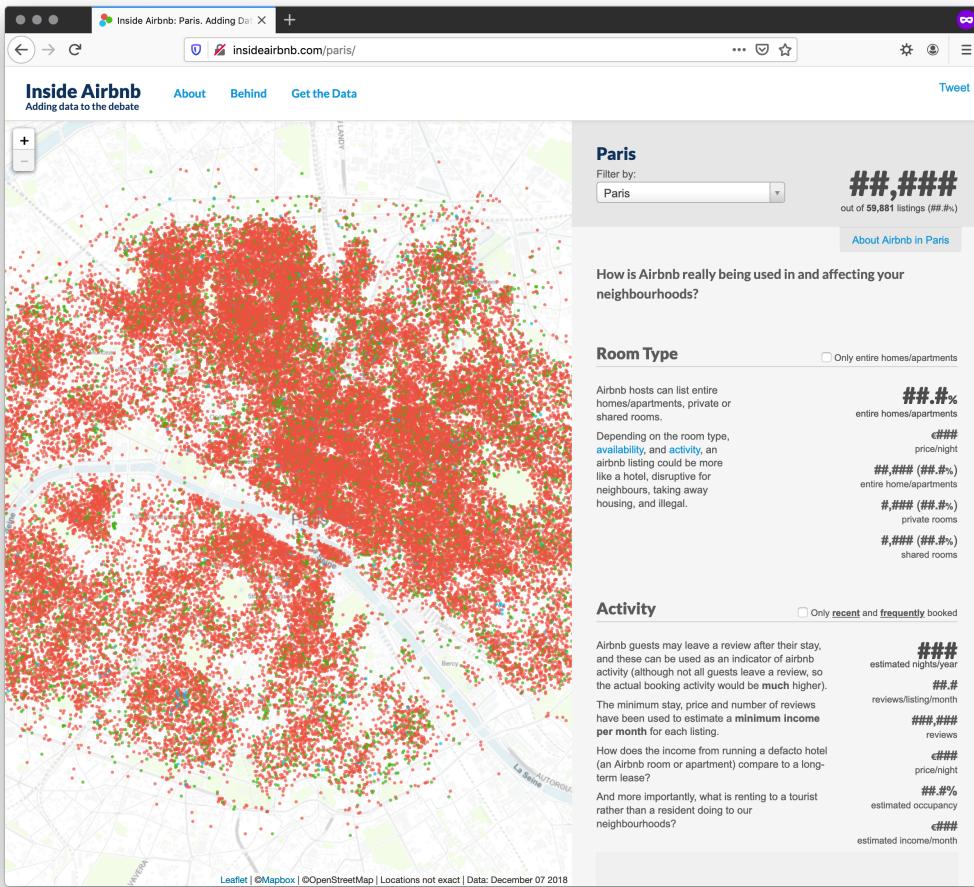
It is important to note Binder instances are *ephemeral* in the sense that the data and content created in a session is **NOT** saved anywhere and is deleted as soon as the browser tab is closed.

Binder is also the backend this website relies on when you click on the rocket icon () on a page with code. Remember, you can play with the code interactively but, once you close the tab, all the changes are lost.

## Data

### *What*

To maintain consistency and allow you to focus on the concepts we will be exploring throughout this course, we will stick to the same data throughout.



## Source

The dataset we will use comes from [InsideAirBnb](#) and contains real estate properties listed for the area of Paris (France). Essentially every bit of information you can inquire about an Airbnb property is recorded in the original dataset. From there, we have processed it to simplify it so it is easier to illustrate the concepts we will be learning in the course.

## Access

All the data files required for this course is contained in the zip compressed [bundled download](#) provided via Github.

## Prep code

Data preparation code is available, if you are so inclined, for you to peruse over [here](#).

## Assessment

Key information:

- Type: Coursework
- [Equivalent to 5,000 words] Up to five figures and three tables + code + comments + up to 2,000
- Chance to be reassessed
- Due on **14:00, March 9th 2021**
- Electronic submission through Turnitin/CANVAS. Static HTML with NO interactive cells

This module is assessed through a *computational essay*. To complete it successfully, you will need to demonstrate aptitude in at least three areas:

1. Data audacity
2. Python data skills
3. Machine learning and inference literacy

These translate in the following components of the computational essay:

## 1 - Find, prepare & explore a dataset

Find a dataset you are excited about and that meets the following characteristics:

- It contains several characteristics (features) for a number of observations (samples)
- At least two characteristics are continuous and at least two are categorical
- You can think of ways in which clustering the observations based on their characteristics could tell an interesting story
- You can imagine a situation in which one of the continuous characteristics can be explained in a supervised model as a function of some of the other characteristics

**NOTE:** please discuss with Dani the choice of dataset before the course finishes

With the dataset at hand:

1. Prepare it for analysis
2. Explore the dataset visually, identifying interesting patterns

## 2 - Unsupervised learning

Perform a clustering exercise & analyse the results. You are expected to try several clustering models, choose a preferred one, and present a critical argument about why that is your choice. To build your argument, you may rely on graphics, performance scores, and substantive reasoning. Demonstrate that you understand not only how the mechanics of the algorithms work but that you are able to translate those into an applied context to make sense of data.

## 3 - Supervised learning

Finally, build a predictive model based on linear regression and:

- Interpret the coefficient
- Evaluate its predictive performance both with and without cross-validation
- Reflect on the differences between assessing the performance of a model cross-validating and not.

Similarly to the previous point, demonstrate that you both understand the workings of the algorithms and techniques but also how you can make the most of it to learn about your data. Critical thinking is critical.

## Bibliography

### References

[B+01]

Leo Breiman and others. Statistical modeling: the two cultures (with comments and a rejoinder by the author). *Statistical science*, 16(3):199–231, 2001.

### [Don17]

David Donoho. 50 years of data science. *Journal of Computational and Graphical Statistics*, 26(4):745–766, 2017.

### [MS17]

Sendhil Mullainathan and Jann Spiess. Machine learning: an applied econometric approach. *Journal of Economic Perspectives*, 31(2):87–106, 2017.

### [RABWng]

Sergio J. Rey, Daniel Arribas-Bel, and Levi J. Wolf. *Geographic Data Science with PySAL and the PyData stack*. CRC press, forthcoming.

## Further materials

A list of further materials where you can continue learning is available at:

[http://darribas.org/gds19/further\\_resources.html](http://darribas.org/gds19/further_resources.html)

## “Data, data, data”

### [\[Async\]](#)

We kick off the course with an overview of what Data Science, as a discipline, is. There are two tasks for you to do ahead of class:

- Watch the following introductory clip:

### Slides

The slides used in the clip are available at:

- [\[HTML\]](#).
- [\[PDF\]](#).



Although not required, the clip mentions Lazer & Radford (2017) [lazer14032014] which, although not fully required for this course, is well worth your time if you are interested in new forms of data in the Social Sciences.

- Read “50 Years of Data Science”, by David Donoho [\[Don17\]](#).

The paper, published under an Open Access license, is available [here](#)

Once you have read Donoho [\[Don17\]](#), the following table below will show you how the [outline of this course](#) can be mapped to his “Greater Data Science” goals:

### Donoho’s Greater Data Science

1. Data Gathering, Preparation, and Exploration

Course Blocks	Goal	
<a href="#">Computational building blocks</a>	GDS3	2. Data Representation and Transformation 3. Computing with Data 4. Data Visualisation and Presentation 5. Data Modeling 6. Science about Data Science
Data Manipulation ( <a href="#">Intro</a> + <a href="#">Advanced</a> )	GDS1 + GDS2	
<a href="#">Exploring data visually</a>	GDS4	
<a href="#">Unsupervised learning</a>	GDS5	
[Supervised learning ( <a href="#">Overview</a> + <a href="#">Inference</a> + <a href="#">Cross-Validation</a> )	GDS5	

## Computational building blocks

[\[Async\]](#)

### Background

Data Science is as much a process as it is an outcome. And *how* that process gets done matters. In this section, we cover the main practices and tools on top of which this course is built.

There are three sets of materials you are expected to go over to complete this section:

1. [Read] "Geospatial Computational Environment", Chapter 2 of the upcoming book by [\[RABWng\]](#).
2. [Watch] Go over the three videos in Block B of the GDS course ([\[URL\]](#)), covering an overview of the main tools we will use in the course.

The chapter can be read online [here](#)

### Software stack

#### ⚠ Attention

Before you move on to the next section, make sure you have all the software stack installed, up and running on your computer so you can follow on the rest of the course interactively

Let's install everything you will need to successfully complete this course! To do so, please follow the appropriate guide for your case from those offered at:

[https://gds-ul.github.io/soft\\_install/](https://gds-ul.github.io/soft_install/)

They will all help you install the same software, but they vary based on the operating system you are running (Windows/macOS/Linux).

## Reading & Manipulating Tabular Data

[\[Sync\]](#)

```
import pandas
import numpy as np
from sqlalchemy import create_engine
```

To note:

- matplotlib magic

- Library import and alias

## Data

- Read in .csv

```
db = pandas.read_csv("../data/paris_abb.csv.zip")
```

- Connect to SQLite and query

```
engine = create_engine("sqlite:///../data/paris_abb_mini.db")
```

```
qry = """
SELECT *
FROM db
LIMIT 5;
"""

dbs = pandas.read_sql(qry, engine)
dbs
```

	<b>id</b>	<b>neighbourhood_cleansed</b>	<b>property_type</b>	<b>room_type</b>	<b>accommodates</b>
0	3109	Observatoire	Apartment	Entire home/apt	2
1	5396	Hôtel-de-Ville	Apartment	Entire home/apt	2
2	7397	Hôtel-de-Ville	Apartment	Entire home/apt	4
3	7964	Opéra	Apartment	Entire home/apt	2
4	9952	Popincourt	Apartment	Entire home/apt	2

```
qry = """
SELECT id, Price
FROM db
WHERE neighbourhood_cleansed = 'Popincourt';
"""

dbs = pandas.read_sql(qry, engine)
dbs.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5137 entries, 0 to 5136
Data columns (total 2 columns):
 #   Column  Non-Null Count  Dtype  
---  -- 
 0   id      5137 non-null   int64  
 1   Price    5137 non-null   float64 
dtypes: float64(1), int64(1) 
memory usage: 80.4 KB
```

- Explore

```
db.head()
```

	<b>id</b>	<b>neighbourhood_cleansed</b>	<b>property_type</b>	<b>room_type</b>	<b>accommodates</b>
0	3109	Observatoire	Apartment	Entire home/apt	2
1	5396	Hôtel-de-Ville	Apartment	Entire home/apt	2
2	7397	Hôtel-de-Ville	Apartment	Entire home/apt	4
3	7964	Opéra	Apartment	Entire home/apt	2
4	9952	Popincourt	Apartment	Entire home/apt	2

```
db.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 50280 entries, 0 to 50279
Data columns (total 10 columns):
 #   Column           Non-Null Count  Dtype  
 ---  --  
 0   id               50280 non-null   int64  
 1   neighbourhood_cleansed 50280 non-null   object  
 2   property_type     50280 non-null   object  
 3   room_type         50280 non-null   object  
 4   accommodates      50280 non-null   int64  
 5   bathrooms          50280 non-null   float64 
 6   bedrooms           50280 non-null   float64 
 7   beds               50280 non-null   float64 
 8   bed_type           50280 non-null   object  
 9   Price              50280 non-null   float64 
dtypes: float64(4), int64(2), object(4)
memory usage: 3.8+ MB
```

```
db.describe()
```

	<b>id</b>	<b>accommodates</b>	<b>bathrooms</b>	<b>bedrooms</b>	<b>price</b>
<b>count</b>	5.028000e+04	50280.000000	50280.000000	50280.000000	50280.000000
<b>mean</b>	1.968671e+07	3.063425	1.117045	1.077804	1.66e+06
<b>std</b>	1.140675e+07	1.556731	0.687370	1.001000	1.16e+06
<b>min</b>	3.109000e+03	1.000000	0.000000	0.000000	0.000000
<b>25%</b>	9.340209e+06	2.000000	1.000000	1.000000	1.00e+06
<b>50%</b>	1.999153e+07	2.000000	1.000000	1.000000	1.00e+06
<b>75%</b>	2.987492e+07	4.000000	1.000000	1.000000	2.00e+06
<b>max</b>	3.858893e+07	17.000000	50.000000	50.000000	50.00e+06

DataFrame objects can hold different types of data

- “Whole” numbers (int)
- Decimals (float)
- Categorical (pandas.Category)
- Dates (pandas.Timestamp)
- Geo (geopandas.GeoDataFrame)
- ...

Indices

```
db.head(2)
```

```
   id neighbourhood_cleansed property_type room_type accommodates
0  3109          Observatoire     Apartment    Entire home/apt      2
1  5396        Hôtel-de-Ville     Apartment    Entire home/apt      2
```

```
db.set_index("id").head(2)
```

```
neighbourhood_cleansed property_type room_type accommodates bat
id
3109          Observatoire     Apartment    Entire home/apt      2
5396        Hôtel-de-Ville     Apartment    Entire home/apt      2
```

```
dbi = db.set_index("id")
```

```
dbi.index
```

```
Int64Index([ 3109,      5396,      7397,      7964,      9952,     10710,
           11170,     11213,     11265,     11798,
...,
38485563, 38489275, 38513797, 38516223, 38516341, 38517692,
38520175, 38537044, 38546594, 38588929],
dtype='int64', name='id', length=50280)
```

```
dbi.columns
```

```
Index(['neighbourhood_cleansed', 'property_type', 'room_type', 'accommodates',
       'bathrooms', 'bedrooms', 'beds', 'bed_type', 'Price'],
      dtype='object')
```

## Slicing and Dicing Data

### Index-based queries

- Columns

```
db[ "Price" ].head()
```

```
0    60.0
1   115.0
2   119.0
3   130.0
4    75.0
Name: Price, dtype: float64
```

- Generic point-querying

```
db.loc[0, "Price"]
```

```
60.0
```

- Full slice of one dimension

```
db.loc[0, :]
```

```
id                      3109
neighbourhood_cleansed    Observatoire
property_type              Apartment
room_type                  Entire home/apt
accommodates                   2
bathrooms                     1
bedrooms                      0
beds                          1
bed_type                      Real Bed
Price                         60
Name: 0, dtype: object
```

```
db.loc[:, "Price"]
```

```
0      60.0
1      115.0
2      119.0
3      130.0
4      75.0
...
50275   250.0
50276   40.0
50277   60.0
50278   65.0
50279   69.0
Name: Price, Length: 50280, dtype: float64
```

- Range queries

```
db.loc[0:5, "neighbourhood_cleansed"]
```

```
0      Observatoire
1      Hôtel-de-Ville
2      Hôtel-de-Ville
3      Opéra
4      Popincourt
5      Élysée
Name: neighbourhood_cleansed, dtype: object
```

```
db.loc[5, "property_type": "bed_type"]
```

```
property_type          Apartment
room_type              Entire home/apt
accommodates                     4
bathrooms                        1
bedrooms                           1
beds                            2
bed_type                      Real Bed
Name: 5, dtype: object
```

- List-based queries

```
db.loc[[0, 49, 19, 29, 39, 9], ["Price", "id"]]
```

	Price	id
0	60.0	3109
49	128.0	32082
19	65.0	17919
29	80.0	21264
39	49.0	26567
9	120.0	11798

## Order-based queries

```
dbi.iloc[0:5, :]
```

	neighbourhood_cleansed	property_type	room_type	accommodates	bat
id					
3109	Observatoire	Apartment	Entire home/apt	2	
5396	Hôtel-de-Ville	Apartment	Entire home/apt	2	
7397	Hôtel-de-Ville	Apartment	Entire home/apt	4	
7964	Opéra	Apartment	Entire home/apt	2	
9952	Popincourt	Apartment	Entire home/apt	2	

## EXERCISE:

- Slice the dataset to keep only properties with the following IDs, in that order: 38520175, 619716, and 37847454
- Extract the section of the dataset that includes the 50th to the 100th rows, and the room\_type and bedrooms columns

## Conditional queries

- Using loc

```
db.loc[db["neighbourhood_cleansed"] == "Observatoire",
       ["neighbourhood_cleansed", "Price"]]\n       .head()
```

	neighbourhood_cleansed	Price
0	Observatoire	60.0
47	Observatoire	90.0
52	Observatoire	150.0
104	Observatoire	84.0
144	Observatoire	140.0

```
db.loc[db["Price"] < 100, ["id", "neighbourhood_cleansed"]].head()
```

	id	neighbourhood_cleansed
0	3109	Observatoire
4	9952	Popincourt
5	10710	Élysée
6	11170	Panthéon
10	11848	Popincourt

```
db.loc[(db["Price"] < 100) & \
       (db["bathrooms"] >= 8),  
:]
```

	<b>id</b>	<b>neighbourhood_cleansed</b>	<b>property_type</b>	<b>room_type</b>	<b>accomr</b>
<b>12066</b>	8876983	Luxembourg	Boutique hotel	Private room	
<b>22964</b>	18766792	Luxembourg	Boutique hotel	Private room	
<b>24788</b>	19819352	Luxembourg	Boutique hotel	Private room	
<b>25798</b>	20433587	Luxembourg	Boutique hotel	Private room	
<b>26048</b>	20691340	Luxembourg	Boutique hotel	Private room	
<b>26108</b>	20747725	Luxembourg	Boutique hotel	Private room	
<b>26121</b>	20768013	Luxembourg	Boutique hotel	Private room	

- Conditional filters

```
fltr = db[ "bathrooms" ] > 8  
fltr.head()
```

```
0    False  
1    False  
2    False  
3    False  
4    False  
Name: bathrooms, dtype: bool
```

```
db[fltr]
```

	<b>id</b>	<b>neighbourhood_cleansed</b>	<b>property_type</b>	<b>room_type</b>	<b>accomr</b>
<b>12066</b>	8876983	Luxembourg	Boutique hotel	Private room	
<b>22964</b>	18766792	Luxembourg	Boutique hotel	Private room	
<b>24788</b>	19819352	Luxembourg	Boutique hotel	Private room	
<b>25798</b>	20433587	Luxembourg	Boutique hotel	Private room	
<b>26048</b>	20691340	Luxembourg	Boutique hotel	Private room	
<b>26108</b>	20747725	Luxembourg	Boutique hotel	Private room	
<b>26121</b>	20768013	Luxembourg	Boutique hotel	Private room	

- Concatenated queries

```
db.loc[(db["Price"] < 100) & \
       (db["bathrooms"] >= 8),  
:]
```

	<b>id</b>	<b>neighbourhood_cleansed</b>	<b>property_type</b>	<b>room_type</b>	<b>accomr</b>
<b>12066</b>	8876983	Luxembourg	Boutique hotel	Private room	
<b>22964</b>	18766792	Luxembourg	Boutique hotel	Private room	
<b>24788</b>	19819352	Luxembourg	Boutique hotel	Private room	
<b>25798</b>	20433587	Luxembourg	Boutique hotel	Private room	
<b>26048</b>	20691340	Luxembourg	Boutique hotel	Private room	
<b>26108</b>	20747725	Luxembourg	Boutique hotel	Private room	
<b>26121</b>	20768013	Luxembourg	Boutique hotel	Private room	

```
db.loc[(db["Price"] < 5) | \  
       (db["Price"] > 5000),  
:]
```

	<b>id</b>	<b>neighbourhood_cleansed</b>	<b>property_type</b>	<b>room_type</b>	<b>accommr</b>
8149	6088687		Temple	Apartment	Entire home/apt
10113	7225849	Buttes-Montmartre		Apartment	Entire home/apt
11286	8093890		Passy	Apartment	Entire home/apt
25106	19974916	Buttes-Montmartre		Condominium	Entire home/apt
25558	20219162		Buttes-Chaumont	Apartment	Entire home/apt
25676	20291987		Passy	Apartment	Entire home/apt
25697	20313940		Temple	Apartment	Entire home/apt
26899	21422028		Popincourt	Apartment	Entire home/apt
32190	25448670		Vaugirard	Apartment	Entire home/apt
35291	27608896		Observatoire	Apartment	Private room
43961	34380017		Bourse	Serviced apartment	Hotel room
46767	36019554		Louvre	Serviced apartment	Hotel room
47574	36402651	Batignolles-Monceau		House	Entire home/apt

- Using query

```
db.query("neighbourhood_cleansed == 'Observatoire'")\
[["neighbourhood_cleansed", "Price"]]\n.head()
```

	<b>neighbourhood_cleansed</b>	<b>Price</b>
0	Observatoire	60.0
47	Observatoire	90.0
52	Observatoire	150.0
104	Observatoire	84.0
144	Observatoire	140.0

```
db.query("Price < 100")\
[["id", "neighbourhood_cleansed"]]\n.head()
```

```

      id neighbourhood_cleansed
0    3109          Observatoire
4    9952        Popincourt
5   10710         Élysée
6   11170       Panthéon
10  11848        Popincourt

```

```
db.query("(Price < 100) & (bathrooms >= 8)")
```

	id	neighbourhood_cleansed	property_type	room_type	accommr
<b>12066</b>	8876983	Luxembourg	Boutique hotel	Private room	
<b>22964</b>	18766792	Luxembourg	Boutique hotel	Private room	
<b>24788</b>	19819352	Luxembourg	Boutique hotel	Private room	
<b>25798</b>	20433587	Luxembourg	Boutique hotel	Private room	
<b>26048</b>	20691340	Luxembourg	Boutique hotel	Private room	
<b>26108</b>	20747725	Luxembourg	Boutique hotel	Private room	
<b>26121</b>	20768013	Luxembourg	Boutique hotel	Private room	

## Editing tables

- Modifying single values

```
db.loc[1, "bed_type"]
```

```
'Pull-out Sofa'
```

```
db.loc[1, "bed_type"] = "Pullout Sofa"
```

```
db.loc[1, "bed_type"]
```

```
'Pullout Sofa'
```

- Modifying blocks of values

```
db.neighbourhood_cleansed.unique()
```

```
array(['Observatoire', 'Hôtel-de-Ville', 'Opéra', 'Popincourt', 'Élysée',
       'Panthéon', 'Entrepôt', 'Buttes-Montmartre', 'Gobelins',
       'Buttes-Chaumont', 'Luxembourg', 'Louvre', 'Palais-Bourbon',
       'Reuilly', 'Bourse', 'Ménilmontant', 'Vaugirard',
       'Batignolles-Monceau', 'Temple', 'Passy'], dtype=object)
```

```
db.loc[db["neighbourhood_cleansed"] == "Hôtel-de-Ville",
      "neighbourhood_cleansed"] = "City Council"
```

```
db.neighbourhood_cleansed.unique()
```

```
array(['Observatoire', 'City Council', 'Opéra', 'Popincourt', 'Élysée',  
       'Panthéon', 'Entrepôt', 'Buttes-Montmartre', 'Gobelins',  
       'Buttes-Chaumont', 'Luxembourg', 'Louvre', 'Palais-Bourbon',  
       'Reuilly', 'Bourse', 'Ménilmontant', 'Vaugirard',  
       'Batignolles-Monceau', 'Temple', 'Passy'], dtype=object)
```

- Creating new columns and rows

```
db["more_beds_than_accomodates"] = db["beds"] > db["accommodates"]
```

## EXERCISE

- Find how many properties have more than ten bathrooms
- Can you rent an AirBnb in Paris with only one bed but three bedrooms?
- In which neighbourhoods can you rent an “Airbed”?

## Writing Data

```
[i for i in dir(db) if i[:3]=="to_"]
```

```
['to_clipboard',  
 'to_csv',  
 'to_dict',  
 'to_excel',  
 'to_feather',  
 'to_gbq',  
 'to_hdf',  
 'to_html',  
 'to_json',  
 'to_latex',  
 'to_markdown',  
 'to_numpy',  
 'to_parquet',  
 'to_period',  
 'to_pickle',  
 'to_records',  
 'to_sql',  
 'to_stata',  
 'to_string',  
 'to_timestamp',  
 'to_xarray']
```

## Further Resources

Similar introduction:

```
http://darribas.org/gds19/content/labs/lab\_01.html
```

More stuff on indices:

```
http://pandas.pydata.org/pandas-docs/stable/indexing.html
```

And for the pros:

```
http://pandas.pydata.org/pandas-docs/stable/advanced.html
```

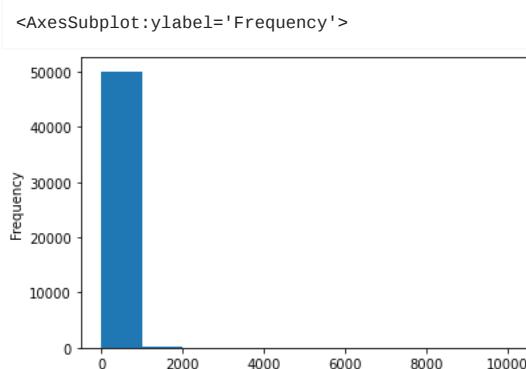
## Visualising Tabular Data

[\[Sync\]](#)

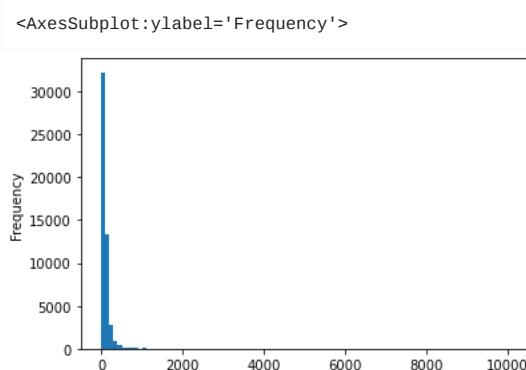
```
%matplotlib inline  
  
import pandas  
import seaborn as sns  
import matplotlib.pyplot as plt  
  
db = pandas.read_csv("../data/paris_abb.csv.zip")
```

## Simple & Quick (pandas)

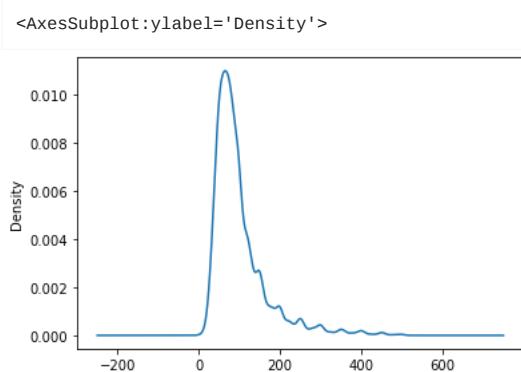
```
db["Price"].plot.hist()
```



```
db["Price"].plot.hist(bins=100)
```

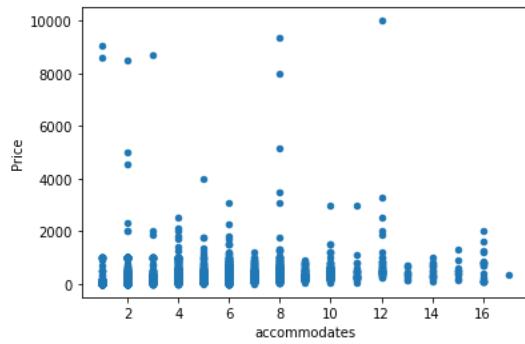


```
db.query("Price < 500")["Price"].plot.kde()
```



```
db.plot.scatter("accommodates", "Price")
```

```
<AxesSubplot:xlabel='accommodates', ylabel='Price'>
```



## EXERCISE

- Create a histogram of the distribution of number of people a property accommodates (accommodates)
- Create a scatter plot of the number of beds and number of bedrooms

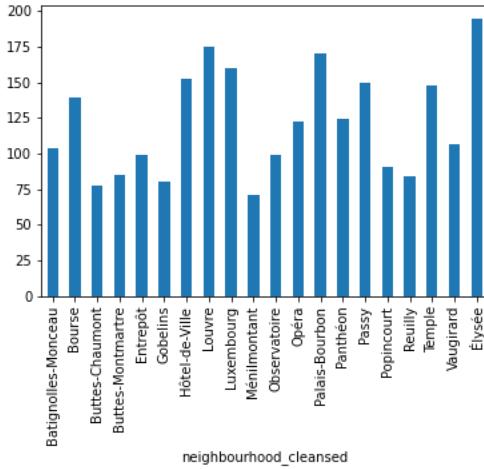
## More

```
https://pandas.pydata.org/pandas-docs/stable/user\_guide/visualization.html
```

We can also combine this with groupings from the previous notebook:

```
db.groupby("neighbourhood_cleansed")\
    ["Price"]\n    .mean()\n    .plot.bar()
```

```
<AxesSubplot:xlabel='neighbourhood_cleansed'>
```



## EXERCISE

Create a bar plot of the average review\_scores\_rating by neighbourhood, sorted by average price.

For a “pro” touch (optional), subtract 90 from the reviews score before plotting.

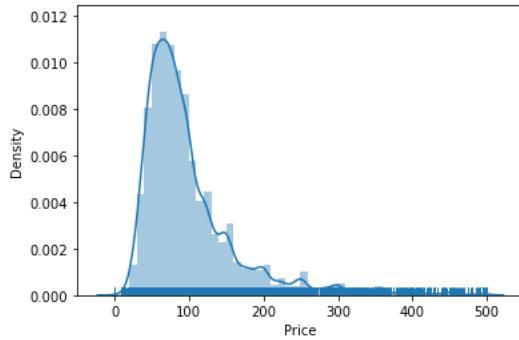
## seaborn

- Univariate

```
sns.distplot(db.query("Price < 500")["Price"],  
            kde=True,  
            rug=True)
```

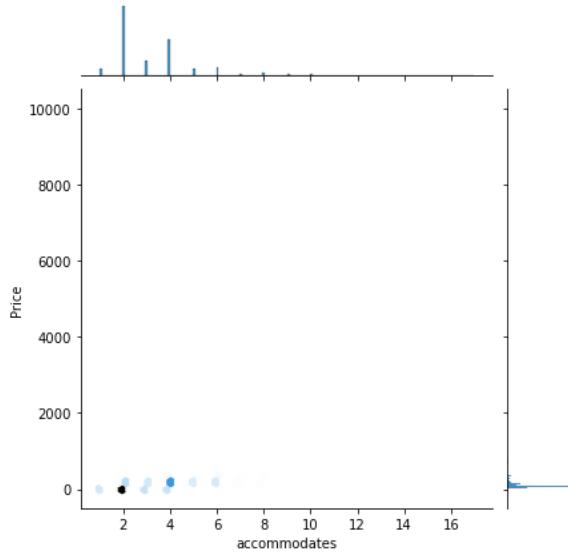
```
/opt/conda/lib/python3.8/site-packages/seaborn/distributions.py:2551: FutureWarning:  
`distplot` is a deprecated function and will be removed in a future version. Please  
adapt your code to use either `displot` (a figure-level function with similar  
flexibility) or `histplot` (an axes-level function for histograms).  
warnings.warn(msg, FutureWarning)  
/opt/conda/lib/python3.8/site-packages/seaborn/distributions.py:2055: FutureWarning:  
The `axis` variable is no longer used and will be removed. Instead, assign variables  
directly to `x` or `y`.  
warnings.warn(msg, FutureWarning)
```

```
<AxesSubplot:xlabel='Price', ylabel='Density'>
```

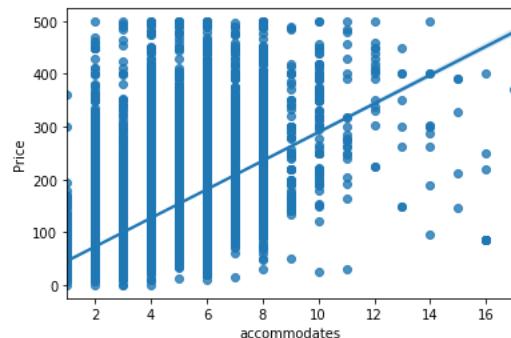


- Bivariate

```
sns.jointplot(x = "accommodates",  
               y = "Price",  
               data=db,  
               kind="hex");
```

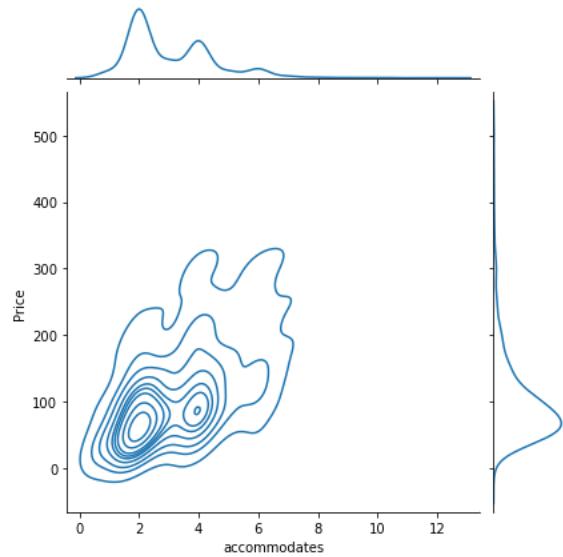


```
sns.regplot(x = "accommodates",  
            y = "Price",  
            data=db.query("Price < 500"))
```



```
sns.jointplot(x = "accommodates",
               y = "Price",
               data=db.query("Price < 500")\
                     .sample(1000),
               kind="kde");

```



## EXERCISE

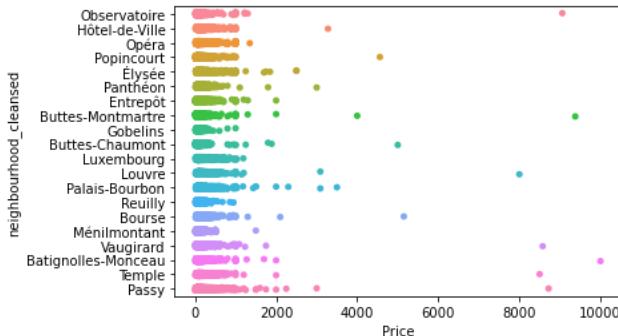
Explore the documentation of `jointplot` and create a figure similar to the one above where you replace the hexagonal binning for a KDE. Since this will probably take too long, subset your data before plotting to only properties cheaper than \$500, and then randomly sample 1,000 observations (tip: check out the [sample](#) method).

## More

<http://seaborn.pydata.org/tutorial.html>

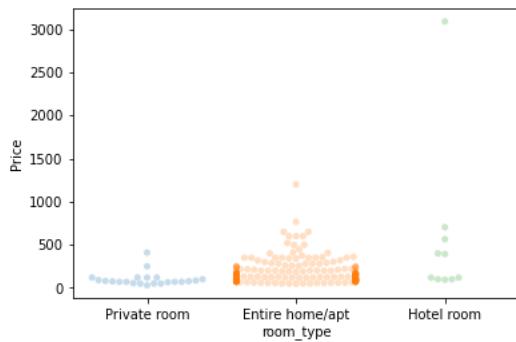
- Categorical

```
sns.stripplot(x = "Price",
               y = "neighbourhood_cleaned",
               data=db
               );
```



```
sns.swarmplot(x = "room_type",
               y = "Price",
               data=db.query("neighbourhood_cleansed == 'Louvre'")\
                     .sample(250),
               alpha=0.25
);
```

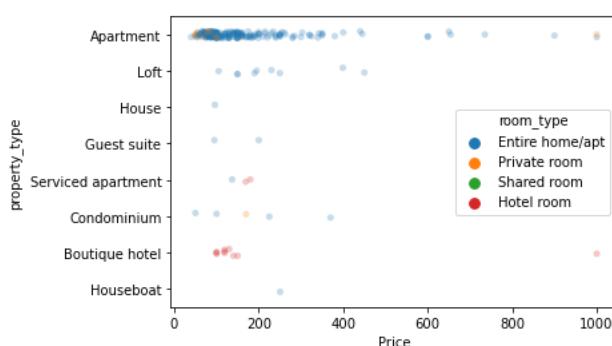
/opt/conda/lib/python3.8/site-packages/seaborn/categorical.py:1296: UserWarning:  
 58.3% of the points cannot be placed; you may want to decrease the size of the  
 markers or use stripplot.  
 warnings.warn(msg, UserWarning)



To note:

- With larger datasets, it's hard to see any pattern
- This is true even if you jitter the points around to avoid overlap and/or you play with transparency (alpha)
- Algorithms to separate out dots exist but they're computationally intensive and can only do so much

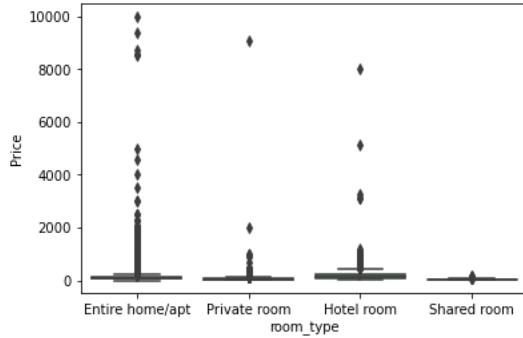
```
sns.stripplot(x = "Price",
               y = "property_type",
               hue = "room_type",
               data=db.query("neighbourhood_cleansed == 'Louvre'")\
                     .sample(250),
               alpha=0.25
);
```



```

sns.boxplot(x = "room_type",
            y = "Price",
            data = db
            );

```

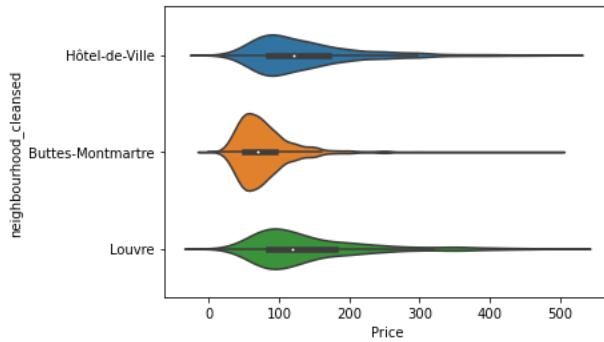


```

nei_list = ["Hôtel-de-Ville", "Louvre", "Buttes-Montmartre"]
sub = db["neighbourhood_cleaned"].isin(nei_list)

sns.violinplot(x = "Price",
                y = "neighbourhood_cleaned",
                data = db[sub].query("Price < 500")
                );

```



## EXERCISE

Explore the distribution of price by property type

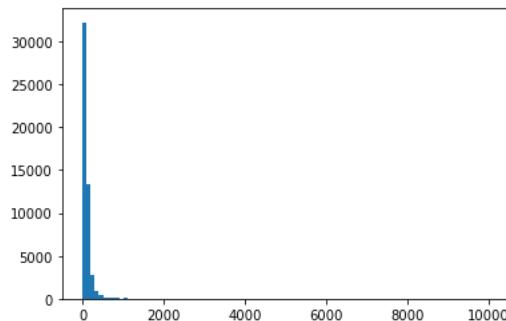
## Full control (matplotlib)

One

```

f, ax = plt.subplots(1)
ax.hist(db['Price'], bins=100)
plt.show()

```



```

sub = db.query("Price < 500")\
    .sample(1000)

f, ax = plt.subplots(1)

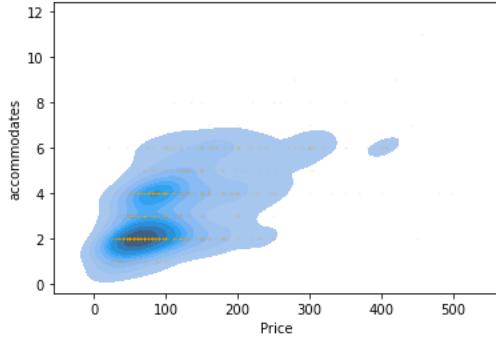
sns.kdeplot(sub['Price'], sub['accommodates'],
            shade=True, ax=ax)
ax.scatter(sub['Price'], sub['accommodates'],
           alpha=0.1, s=0.75, color='orange')

plt.show()

```

/opt/conda/lib/python3.8/site-packages/seaborn/\_decorators.py:36: FutureWarning: Pass the following variable as a keyword arg: y. From version 0.12, the only valid positional argument will be `data`, and passing other arguments without an explicit keyword will result in an error or misinterpretation.

```
warnings.warn(
```



```

f, ax = plt.subplots(1, figsize=(4, 4))

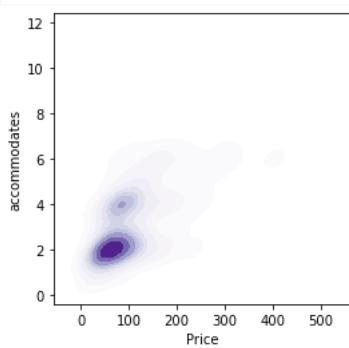
sns.kdeplot(sub['Price'],
            sub['accommodates'],
            shade=True, ax=ax, cmap='Purples')

plt.show()

```

/opt/conda/lib/python3.8/site-packages/seaborn/\_decorators.py:36: FutureWarning: Pass the following variable as a keyword arg: y. From version 0.12, the only valid positional argument will be `data`, and passing other arguments without an explicit keyword will result in an error or misinterpretation.

```
warnings.warn(
```



**CHALLENGE** - Create a visualisation that includes a KDE and a scatter plot and that explores the relationship between number of beds and number of people it accommodates, for a random sample of 500 properties.

Two or more

```

f, axs = plt.subplots(1, 2, figsize=(12, 6))

sns.distplot(sub['Price'],
             kde=False,
             rug=True,
             ax=axs[0]
            )

sns.distplot(sub['accommodates'],
             hist=False,
             kde=True,
             rug=True,
             ax=axs[1]
            )

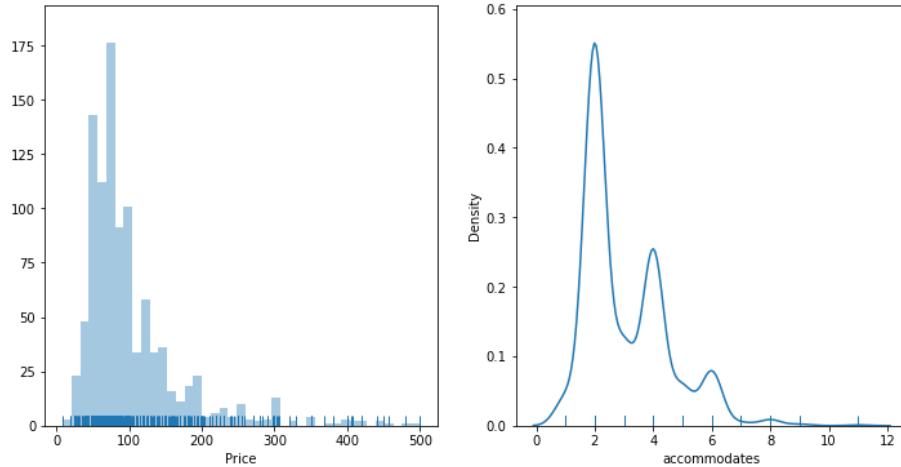
plt.show()

```

```

/opt/conda/lib/python3.8/site-packages/seaborn/distributions.py:2551: FutureWarning:
`distplot` is a deprecated function and will be removed in a future version. Please
adapt your code to use either `displot` (a figure-level function with similar
flexibility) or `histplot` (an axes-level function for histograms).
warnings.warn(msg, FutureWarning)
/opt/conda/lib/python3.8/site-packages/seaborn/distributions.py:2055: FutureWarning:
The `axis` variable is no longer used and will be removed. Instead, assign variables
directly to `x` or `y`.
warnings.warn(msg, FutureWarning)
/opt/conda/lib/python3.8/site-packages/seaborn/distributions.py:2551: FutureWarning:
`distplot` is a deprecated function and will be removed in a future version. Please
adapt your code to use either `displot` (a figure-level function with similar
flexibility) or `kdeplot` (an axes-level function for kernel density plots).
warnings.warn(msg, FutureWarning)
/opt/conda/lib/python3.8/site-packages/seaborn/distributions.py:2055: FutureWarning:
The `axis` variable is no longer used and will be removed. Instead, assign variables
directly to `x` or `y`.
warnings.warn(msg, FutureWarning)

```



**CHALLENGE** - Create a visualisation for all of the properties with three subplots:

1. Histogram of price
2. Scatter plot of price Vs number of people it accommodates
3. Histogram of number of people the property accommodates

## Advanced Tabular Manipulation

[\[Sync\]](#)

```

%matplotlib inline
import pandas
db = pandas.read_csv("../data/paris_abb.csv.zip")

```

Sorting

- By values

```
# Top-5 cheapes properties
db.sort_values("Price")\
    .head(5)
```

		id	neighbourhood_cleansed	property_type	room_type	accomr
25676	20291987			Passy	Apartment	Entire home/apt
25697	20313940			Temple	Apartment	Entire home/apt
26899	21422028			Popincourt	Apartment	Entire home/apt
25558	20219162			Buttes-Chaumont	Apartment	Entire home/apt
25106	19974916			Buttes-Montmartre	Condominium	Entire home/apt

```
# Top-5 most expensive properties
db.sort_values("Price", ascending=False)\n    .head(5)
```

		id	neighbourhood_cleansed	property_type	room_type	accomr
47574	36402651		Batignolles-Monceau		House	Entire home/apt
10113	7225849		Buttes-Montmartre		Apartment	Entire home/apt
35291	27608896		Observatoire		Apartment	Private room
11286	8093890			Passy	Apartment	Entire home/apt
32190	25448670			Vaugirard	Apartment	Entire home/apt

- By index

```
tmp = db.set_index("property_type")\
    .sort_index()
tmp.head()
```

	id	neighbourhood_cleansed	room_type	accommodates
<b>property_type</b>				
Aparthotel	31728955	Vaugirard	Hotel room	2
Aparthotel	13003443	Vaugirard	Entire home/apt	4
Aparthotel	31733851	Temple	Hotel room	4
Aparthotel	17630233	Bourse	Hotel room	4
Aparthotel	24387710	Observatoire	Private room	12

(Useful for quick subsetting:)

```
tmp.loc["Tiny house",
        ["id", "neighbourhood_cleansed", "Price"]]
```

	<b>id</b>	<b>neighbourhood_cleansed</b>	<b>Price</b>
	<b>property_type</b>		
Tiny house	20976623	Panthéon	110.0
Tiny house	6838781	Entrepôt	60.0
Tiny house	29322690	Passy	30.0
Tiny house	18919023	Louvre	100.0
Tiny house	4191080	Louvre	80.0
Tiny house	9582468	Popincourt	80.0
Tiny house	37312602	Ménilmontant	40.0
Tiny house	2555221	Buttes-Montmartre	114.0
Tiny house	34572791	Passy	35.0
Tiny house	37174177	Vaugirard	50.0
Tiny house	26292760	Ménilmontant	75.0
Tiny house	36864294	Passy	38.0
Tiny house	20273867	Panthéon	100.0
Tiny house	34816113	Observatoire	40.0
Tiny house	768986	Temple	79.0
Tiny house	13220094	Buttes-Chaumont	160.0
Tiny house	5699102	Buttes-Montmartre	71.0
Tiny house	36048024	Buttes-Montmartre	70.0
Tiny house	20688679	Buttes-Montmartre	73.0
Tiny house	15157047	Temple	42.0
Tiny house	12588228	Passy	75.0

## Joinning

- Additional data (linked through ID!)

```
reviews = pandas.read_csv("../data/paris_abb_review.csv.zip")
reviews.head()
```

	<b>id</b>	<b>review_scores_rating</b>	<b>review_scores_accuracy</b>	<b>review_scores_cleanline</b>
0	3109	100.0	10.0	10
1	5396	90.0	9.0	8
2	7397	94.0	10.0	9
3	7964	96.0	10.0	10
4	9952	98.0	10.0	10

- Join to original table:

```
dbj1 = db.join(reviews.set_index("id"),
               on = "id"
              )
dbj1.head()
```

	<b>id</b>	<b>neighbourhood_cleansed</b>	<b>property_type</b>	<b>room_type</b>	<b>accommodates</b>
0	3109	Observatoire	Apartment	Entire home/apt	2
1	5396	Hôtel-de-Ville	Apartment	Entire home/apt	2
2	7397	Hôtel-de-Ville	Apartment	Entire home/apt	4
3	7964	Opéra	Apartment	Entire home/apt	2
4	9952	Popincourt	Apartment	Entire home/apt	2

```
dbj2 = db.set_index("id")\
         .join(reviews.set_index("id"))
dbj2.head()
```

<b>id</b>	<b>neighbourhood_cleansed</b>	<b>property_type</b>	<b>room_type</b>	<b>accommodates</b>	<b>bat</b>
3109	Observatoire	Apartment	Entire home/apt	2	
5396	Hôtel-de-Ville	Apartment	Entire home/apt	2	
7397	Hôtel-de-Ville	Apartment	Entire home/apt	4	
7964	Opéra	Apartment	Entire home/apt	2	
9952	Popincourt	Apartment	Entire home/apt	2	

Note:

- Left can choose index/column, right needs to be index (results “almost” the same)
- For more flexibility, check out `merge`:

```
https://pandas.pydata.org/pandas-docs/stable/user\_guide/merging.html
```

## Grouping

- Get the mean price for "Louvre":

```
db.query("neighbourhood_cleansed == 'Louvre'")\
    ["Price"]\
    .mean()
```

```
175.2634194831014
```

- Get the mean price for "Luxembourg":

```
db.query("neighbourhood_cleansed == 'Luxembourg'")\
    ["Price"]\
    .mean()
```

```
160.02262142381903
```

- Get the mean price for "Palais-Bourbon":

```
db.query("neighbourhood_cleansed == 'Palais-Bourbon'")\
    ["Price"]\
    .mean()
```

```
169.8526690391459
```

- For every neighbourhood???

```
db.groupby("neighbourhood_cleansed")\
    ["Price"]\
    .mean()
```

```
neighbourhood_cleansed
Batignolles-Monceau      103.355239
Bourse                   138.898017
Buttes-Chaumont          76.971745
Buttes-Montmartre        84.867211
Entrepôt                 99.130493
Gobelins                 80.000000
Hôtel-de-Ville            152.547723
Louvre                   175.263419
Luxembourg                160.022621
Ménilmontant              71.074504
Observatoire               98.907340
Opéra                     122.495921
Palais-Bourbon             169.852669
Panthéon                  124.683662
Passy                     149.237946
Popincourt                 90.832782
Reuilly                   84.225256
Temple                     147.904592
Vaugirard                  106.269580
Élysée                    194.158416
Name: Price, dtype: float64
```

## MultiIndex Tables

Grouping can be based on more than one variable only...

```
nr = dbj1.groupby(["neighbourhood_cleansed", "room_type"])\
    [["Price", "review_scores_rating"]]\
    .mean()
```

This generates a MultiIndex:

```
nr.head()
```

neighbourhood_cleansed	room_type	Price	review_scores_rating
Batignolles-Monceau	Entire home/apt	103.501922	92.484096
	Hotel room	343.660714	93.607143
	Private room	58.698305	95.064407
	Shared room	53.142857	90.214286
Bourse	Entire home/apt	139.618020	91.608503

nr.index

```

MultiIndex([('Batignolles-Monceau', 'Entire home/apt'),
            ('Batignolles-Monceau', 'Hotel room'),
            ('Batignolles-Monceau', 'Private room'),
            ('Batignolles-Monceau', 'Shared room'),
            ('Bourse', 'Entire home/apt'),
            ('Bourse', 'Hotel room'),
            ('Bourse', 'Private room'),
            ('Bourse', 'Shared room'),
            ('Buttes-Chaumont', 'Entire home/apt'),
            ('Buttes-Chaumont', 'Hotel room'),
            ('Buttes-Chaumont', 'Private room'),
            ('Buttes-Chaumont', 'Shared room'),
            ('Buttes-Montmartre', 'Entire home/apt'),
            ('Buttes-Montmartre', 'Hotel room'),
            ('Buttes-Montmartre', 'Private room'),
            ('Buttes-Montmartre', 'Shared room'),
            ('Entrepôt', 'Entire home/apt'),
            ('Entrepôt', 'Hotel room'),
            ('Entrepôt', 'Private room'),
            ('Entrepôt', 'Shared room'),
            ('Gobelins', 'Entire home/apt'),
            ('Gobelins', 'Hotel room'),
            ('Gobelins', 'Private room'),
            ('Gobelins', 'Shared room'),
            ('Hôtel-de-Ville', 'Entire home/apt'),
            ('Hôtel-de-Ville', 'Hotel room'),
            ('Hôtel-de-Ville', 'Private room'),
            ('Hôtel-de-Ville', 'Shared room'),
            ('Louvre', 'Entire home/apt'),
            ('Louvre', 'Hotel room'),
            ('Louvre', 'Private room'),
            ('Louvre', 'Shared room'),
            ('Luxembourg', 'Entire home/apt'),
            ('Luxembourg', 'Hotel room'),
            ('Luxembourg', 'Private room'),
            ('Luxembourg', 'Shared room'),
            ('Ménilmontant', 'Entire home/apt'),
            ('Ménilmontant', 'Hotel room'),
            ('Ménilmontant', 'Private room'),
            ('Ménilmontant', 'Shared room'),
            ('Observatoire', 'Entire home/apt'),
            ('Observatoire', 'Hotel room'),
            ('Observatoire', 'Private room'),
            ('Observatoire', 'Shared room'),
            ('Opéra', 'Entire home/apt'),
            ('Opéra', 'Hotel room'),
            ('Opéra', 'Private room'),
            ('Opéra', 'Shared room'),
            ('Palais-Bourbon', 'Entire home/apt'),
            ('Palais-Bourbon', 'Hotel room'),
            ('Palais-Bourbon', 'Private room'),
            ('Palais-Bourbon', 'Shared room'),
            ('Panthéon', 'Entire home/apt'),
            ('Panthéon', 'Hotel room'),
            ('Panthéon', 'Private room'),
            ('Panthéon', 'Shared room'),
            ('Passy', 'Entire home/apt'),
            ('Passy', 'Hotel room'),
            ('Passy', 'Private room'),
            ('Passy', 'Shared room'),
            ('Popincourt', 'Entire home/apt'),
            ('Popincourt', 'Hotel room'),
            ('Popincourt', 'Private room'),
            ('Popincourt', 'Shared room'),
            ('Reuilly', 'Entire home/apt'),
            ('Reuilly', 'Hotel room'),
            ('Reuilly', 'Private room'),
            ('Reuilly', 'Shared room'),
            ('Temple', 'Entire home/apt'),
            ('Temple', 'Hotel room'),
            ('Temple', 'Private room'),
            ('Temple', 'Shared room'),
            ('Vaugirard', 'Entire home/apt'),
            ('Vaugirard', 'Hotel room'),
            ('Vaugirard', 'Private room'),
            ('Vaugirard', 'Shared room'),
            ('Élysée', 'Entire home/apt'),
            ('Élysée', 'Hotel room'),
            ('Élysée', 'Private room'),
            ('Élysée', 'Shared room')],
            names=['neighbourhood_cleaned', 'room_type'])

```

These indices allow us to do several more things than single index objects. For example:

- One-level queries:

```
nr.xs("Bourse", level="neighbourhood_cleaned")
```

	Price	review_scores_rating
room_type		
Entire home/apt	139.618020	91.608503
Hotel room	321.244898	93.367347
Private room	71.088710	93.225806
Shared room	35.062500	90.875000

```
nr.xs("Shared room", level="room_type")
```

	Price	review_scores_rating
neighbourhood_cleaned		
Batignolles-Monceau	53.142857	90.214286
Bourse	35.062500	90.875000
Buttes-Chaumont	25.228571	90.400000
Buttes-Montmartre	42.555556	89.333333
Entrepôt	30.458333	90.458333
Gobelins	35.600000	92.533333
Hôtel-de-Ville	64.000000	95.375000
Louvre	72.000000	93.400000
Luxembourg	55.000000	88.666667
Ménilmontant	34.923077	90.730769
Observatoire	41.000000	97.142857
Opéra	44.125000	94.500000
Palais-Bourbon	35.400000	95.000000
Panthéon	73.800000	95.200000
Passy	39.785714	93.785714
Popincourt	31.880952	89.261905
Reuilly	30.300000	93.200000
Temple	30.833333	92.833333
Vaugirard	51.333333	94.750000
Élysée	70.000000	96.000000

```
nr.loc[("Bourse", "Shared room"), :]
```

Price	35.0625
review_scores_rating	90.8750
Name:	(Bourse, Shared room), dtype: float64

But also “unstack” it so we can cross-tab:

```
unstacked = nr.unstack()  
unstacked
```

room_type	Price				rev	
	Entire home/apt	Hotel room	Private room	Shared room	Ent ho	Ent ho
<b>neighbourhood_cleansed</b>						
Batignolles-Monceau	103.501922	343.660714	58.698305	53.142857	92.	
Bourse	139.618020	321.244898	71.088710	35.062500	91.	
Buttes-Chaumont	82.568452	117.947368	45.866667	25.228571	93.	
Buttes-Montmartre	87.478750	118.342105	55.217899	42.555556	92.	
Entrepôt	101.621981	311.736842	55.144893	30.458333	92.	
Gobelins	86.284188	106.961538	53.238671	35.600000	91.	
Hôtel-de-Ville	154.687543	412.291667	79.372881	64.000000	93.	
Louvre	165.077367	426.830508	102.828947	72.000000	91.	
Luxembourg	164.863531	204.746667	87.226562	55.000000	92.	
Ménilmontant	75.727787	128.200000	45.138425	34.923077	93.	
Observatoire	95.354978	306.525000	90.521401	41.000000	92.	
Opéra	115.688708	287.089655	75.561905	44.125000	93.	
Palais-Bourbon	170.620910	450.722222	76.144144	35.400000	92.	
Panthéon	129.989522	141.925000	69.246753	73.800000	92.	
Passy	155.033317	331.766667	77.497674	39.785714	91.	
Popincourt	95.073171	138.333333	53.348606	31.880952	93.	
Reuilly	89.299656	104.727273	53.468165	30.300000	93.	
Temple	150.560496	241.977273	77.474453	30.833333	92.	
Vaugirard	108.483296	288.891892	58.145729	51.333333	92.	
Élysée	194.715942	300.590361	94.835165	70.000000	91.	

This in turn creates a MultiIndex on the columns instead, which works similarly:

```
unstacked["Price"]
```

room_type	Entire home/apt	Hotel room	Private room	Shared room
<b>neighbourhood_cleansed</b>				
<b>Batignolles-Monceau</b>	103.501922	343.660714	58.698305	53.142857
<b>Bourse</b>	139.618020	321.244898	71.088710	35.062500
<b>Buttes-Chaumont</b>	82.568452	117.947368	45.866667	25.228571
<b>Buttes-Montmartre</b>	87.478750	118.342105	55.217899	42.555556
<b>Entrepôt</b>	101.621981	311.736842	55.144893	30.458333
<b>Gobelins</b>	86.284188	106.961538	53.238671	35.600000
<b>Hôtel-de-Ville</b>	154.687543	412.291667	79.372881	64.000000
<b>Louvre</b>	165.077367	426.830508	102.828947	72.000000
<b>Luxembourg</b>	164.863531	204.746667	87.226562	55.000000
<b>Ménilmontant</b>	75.727787	128.200000	45.138425	34.923077
<b>Observatoire</b>	95.354978	306.525000	90.521401	41.000000
<b>Opéra</b>	115.688708	287.089655	75.561905	44.125000
<b>Palais-Bourbon</b>	170.620910	450.722222	76.144144	35.400000
<b>Panthéon</b>	129.989522	141.925000	69.246753	73.800000
<b>Passy</b>	155.033317	331.766667	77.497674	39.785714
<b>Popincourt</b>	95.073171	138.333333	53.348606	31.880952
<b>Reuilly</b>	89.299656	104.727273	53.468165	30.300000
<b>Temple</b>	150.560496	241.977273	77.474453	30.833333
<b>Vaugirard</b>	108.483296	288.891892	58.145729	51.333333
<b>Élysée</b>	194.715942	300.590361	94.835165	70.000000

## EXERCISE

Create a table that shows the average price for properties by room and property type

More at:

[https://pandas.pydata.org/pandas-docs/stable/user\\_guide/advanced.html](https://pandas.pydata.org/pandas-docs/stable/user_guide/advanced.html)

## Unsupervised Learning

To do before...

[[\[Async\]](#)

Before we jump on the action, we are going to get some background on unsupervised learning. Please complete watching the following three clips, all sourced from the [Geographic Data Science course](#):

1. [The need to group data](#), which describes *why* unsupervised learning is useful
2. [Non-spatial clustering](#), which describes *what* unsupervised learning is and illustrates it

3. [K-Means](#), which introduces the technique we'll use in class and, by far, most popular one to cluster data

## Action!

[\[Sync\]](#)

```
import pandas
from numpy.random import seed

from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_score, calinski_harabasz_score
from sklearn.decomposition import PCA
from sklearn.preprocessing import scale, MinMaxScaler

import seaborn as sns
import matplotlib.pyplot as plt

orig = pandas.read_csv("../data/paris_abb.csv.zip")
reviews = pandas.read_csv("../data/paris_abb_review.csv.zip")
db = orig.join(reviews.set_index("id"), on="id")
```

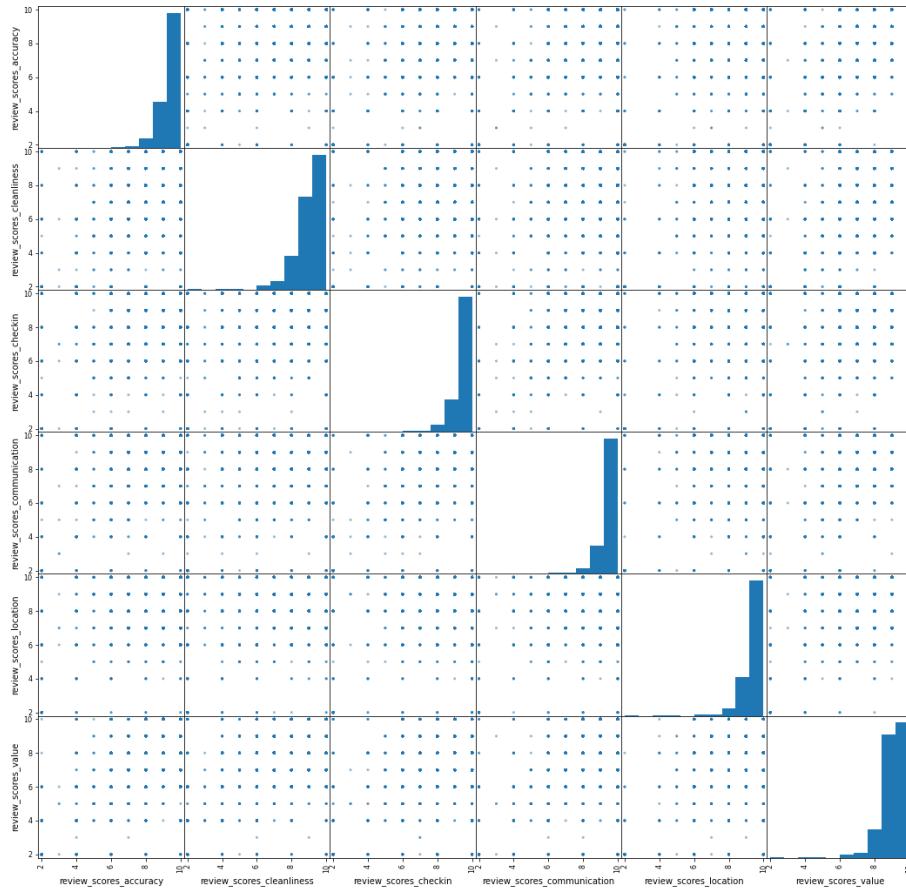
## Explore

```
review_areas = ["review_scores_accuracy",
                 "review_scores_cleanliness",
                 "review_scores_checkin",
                 "review_scores_communication",
                 "review_scores_location",
                 "review_scores_value"
                ]
```

```
db[review_areas].describe().T
```

	count	mean	std	min	25%	50%
review_scores_accuracy	50280.0	9.576929	0.824413	2.0	9.0	10.0
review_scores_cleanliness	50280.0	9.178540	1.107574	2.0	9.0	9.0
review_scores_checkin	50280.0	9.660302	0.769064	2.0	10.0	10.0
review_scores_communication	50280.0	9.703520	0.727020	2.0	10.0	10.0
review_scores_location	50280.0	9.654018	0.696448	2.0	9.0	10.0
review_scores_value	50280.0	9.254515	0.930852	2.0	9.0	9.0

```
pandas.plotting.scatter_matrix(db[review_areas],
                               figsize=(18, 18));
```



This , into a composite index.

## Classify

`scikit-learn` has a very consistent API (learn it once, use it across). It comes in a few flavors:

- `fit`
- `fit\_transform`
- Direct method

- All raw

```
estimator = KMeans(n_clusters = 5)
estimator
```

```
KMeans(n_clusters=5)
```

```
seed(12345)
estimator.fit(db[review_areas])
```

```
KMeans(n_clusters=5)
```

```
k5_raw = pandas.Series(estimator.labels_,
                       index=db.index
                      )
k5_raw.head()
```

0	3
1	0
2	1
3	3
4	3

dtype: int32

## NOTE fit

- All standardised

```
# Minus mean, divided by std
db_stded = scale(db[review_areas])
pandas.DataFrame(db_stded,
                 index = db.index,
                 columns = review_areas
                ).describe()\
                .reindex(["mean", "std"])
```

	review_scores_accuracy	review_scores_cleanliness	review_scores_checkin
mean	2.450638e-14	1.191311e-14	-3.519940e-1
std	1.000010e+00	1.000010e+00	1.000010e+0

## NOTE scale API

```
# Range scale
range_scaler = MinMaxScaler()
db_scaled = range_scaler.fit_transform(db[review_areas])
pandas.DataFrame(db_scaled,
                 index = db.index,
                 columns = review_areas
                ).describe()\
                .reindex(["min", "max"])
```

	review_scores_accuracy	review_scores_cleanliness	review_scores_checkin
min	0.0	0.0	0.0
max	1.0	1.0	1.0

## NOTE fit\_transform

```
seed(12345)

estimator = KMeans(n_clusters = 5)

estimator.fit(db_stded)

k5_std = pandas.Series(estimator.labels_,
                       index=db.index
                      )
k5_std.head()
```

0	0
1	4
2	0
3	0
4	0

dtype: int32

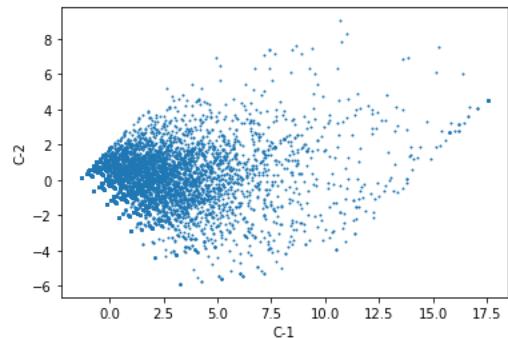
- Projected to lower dimension

```
pca_estimator = PCA(n_components=2)
pca_estimator
```

```
PCA(n_components=2)
```

```
components = pca_estimator.fit_transform(db[review_areas])
components = pandas.DataFrame(components,
                             index = db.index,
                             columns = ["C-1", "C-2"]
                            )
```

```
components.plot.scatter("C-1",
                       "C-2",
                       s=1
                      );
```



Now we cluster the two components instead of all the input variables:

```
seed(12345)
estimator = KMeans(n_clusters = 5)
estimator.fit(components)
k5_pca = pandas.Series(estimator.labels_,
                       index=components.index
                      )
```

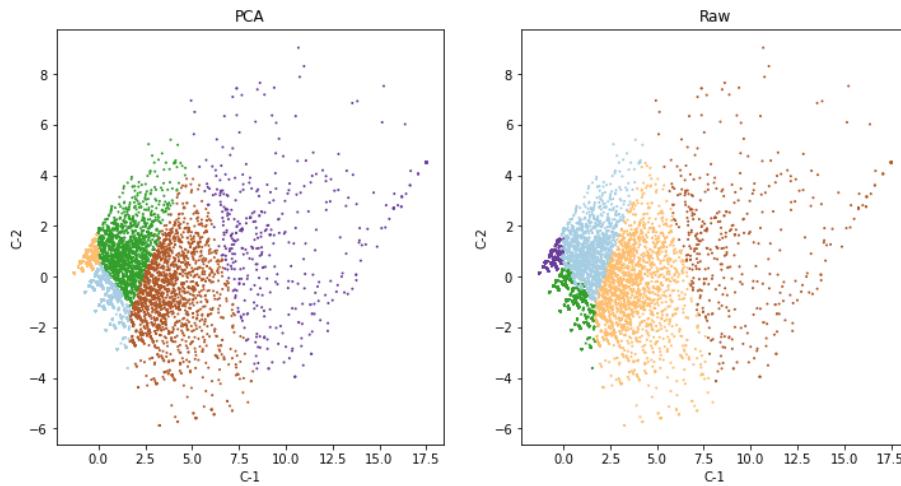
We can compare how both solutions relate to each other (or not):

```
f, axs = plt.subplots(1, 2, figsize=(12, 6))

ax = axs[0]
components.assign(labels=k5_pca)\n    .plot.scatter("C-1",
                  "C-2",
                  c="labels",
                  s=1,
                  cmap="Paired",
                  colorbar=False,
                  ax=ax
                 )
ax.set_title("PCA")

ax = axs[1]
components.assign(labels=k5_raw)\n    .plot.scatter("C-1",
                  "C-2",
                  c="labels",
                  s=1,
                  cmap="Paired",
                  colorbar=False,
                  ax=ax
                 )
ax.set_title("Raw")

plt.show()
```



Actually pretty similar (which is good!). But remember that our original input was expressed in the same units anyway, so it makes sense.

**EXERCISE:** Add a third plot to the figure above visualising the labels with the range-scaled transformation.

Explore the classification

- Quality of clustering ([Calinski and Harabasz score](#), the ratio of between over within dispersion)

```
chs_raw = calinski_harabasz_score(db[review_areas],
                                    k5_raw
                                   )
```

```
chs_std = calinski_harabasz_score(db[review_areas],
                                    k5_std
                                   )
```

```
chs_pca = calinski_harabasz_score(db[review_areas],
                                    k5_pca
                                   )
```

```
pandas.Series({"Raw": chs_raw,
               "Standardised": chs_std,
               "PCA": chs_pca,
              })
```

Raw	16548.489419
Standardised	14674.706963
PCA	16519.822001
dtype: float64	

The higher, the better, so either the original or PCA.

We can also use this to “optimise” (or at least explore its behaviour) the number of clusters. Let’s pick the original input as the scores suggest are more desirable:

```

%%time
seed(12345)

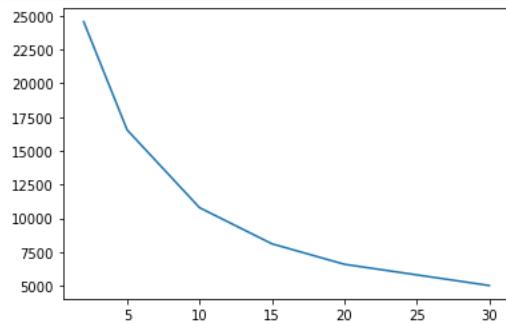
chss = {}
for i in [2, 5, 10, 15, 20, 30]:
    estimator = KMeans(n_clusters=i)
    estimator.fit(components)
    chs = calinski_harabasz_score(db[review_areas],
                                  estimator.labels_)
    chss[i] = chs
chss = pandas.Series(chss)

```

CPU times: user 2min 32s, sys: 1min 9s, total: 3min 42s  
Wall time: 14.6 s

```
chss.plot.line()
```

<AxesSubplot:>



5 clusters?

- Quality of clustering ([silhouette scores](#))

```

%%time
sil_raw = silhouette_score(db[review_areas],
                           k5_raw,
                           metric="euclidean"
                           )

```

CPU times: user 1min 3s, sys: 4min 38s, total: 5min 41s  
Wall time: 41.5 s

```

%%time
sil_std = silhouette_score(db[review_areas],
                           k5_std,
                           metric="euclidean"
                           )

```

CPU times: user 1min 4s, sys: 4min 28s, total: 5min 33s  
Wall time: 41.3 s

```

%%time
sil_pca = silhouette_score(db[review_areas],
                           k5_pca,
                           metric="euclidean"
                           )

```

CPU times: user 1min 5s, sys: 4min 22s, total: 5min 28s  
Wall time: 41 s

```

pandas.Series({"Raw": sil_raw,
               "Standardised": sil_std,
               "PCA": sil_pca,
               })

```

```
Raw          0.314448
Standardised 0.304222
PCA          0.314263
dtype: float64
```

For a graphical analysis of silhouette scores, see here:

```
https://scikit-learn.org/stable/auto\_examples/cluster/plot\_kmeans\_silhouette\_analysis.html
```

**EXERCISE** Compare silhouette scores across our three original approaches for three and 20 clusters

- Characterise

Internally:

```
g = db[review_areas]\n    .groupby(k5_pca)
```

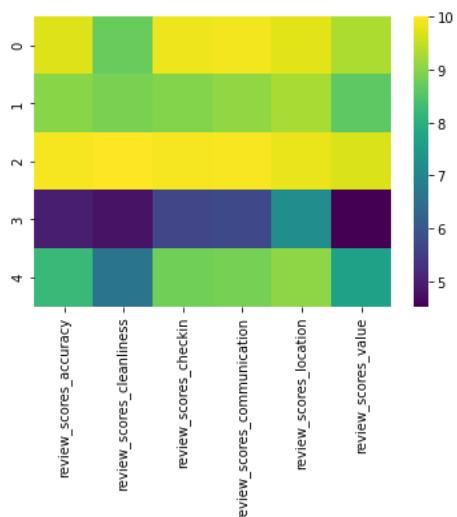
```
g.size()\n    .sort_values()
```

```
3      437\n4     2850\n1     7752\n0    16468\n2    22773\ndtype: int64
```

```
g.mean()
```

	review_scores_accuracy	review_scores_cleanliness	review_scores_checkin	r
0	9.705186	8.728564	9.867379	
1	9.020253	8.901961	8.986068	
2	9.931366	10.000000	9.923989	
3	4.993135	4.789474	5.665904	
4	8.220702	6.640000	8.803158	

```
sns.heatmap(g.mean(), cmap='viridis');
```



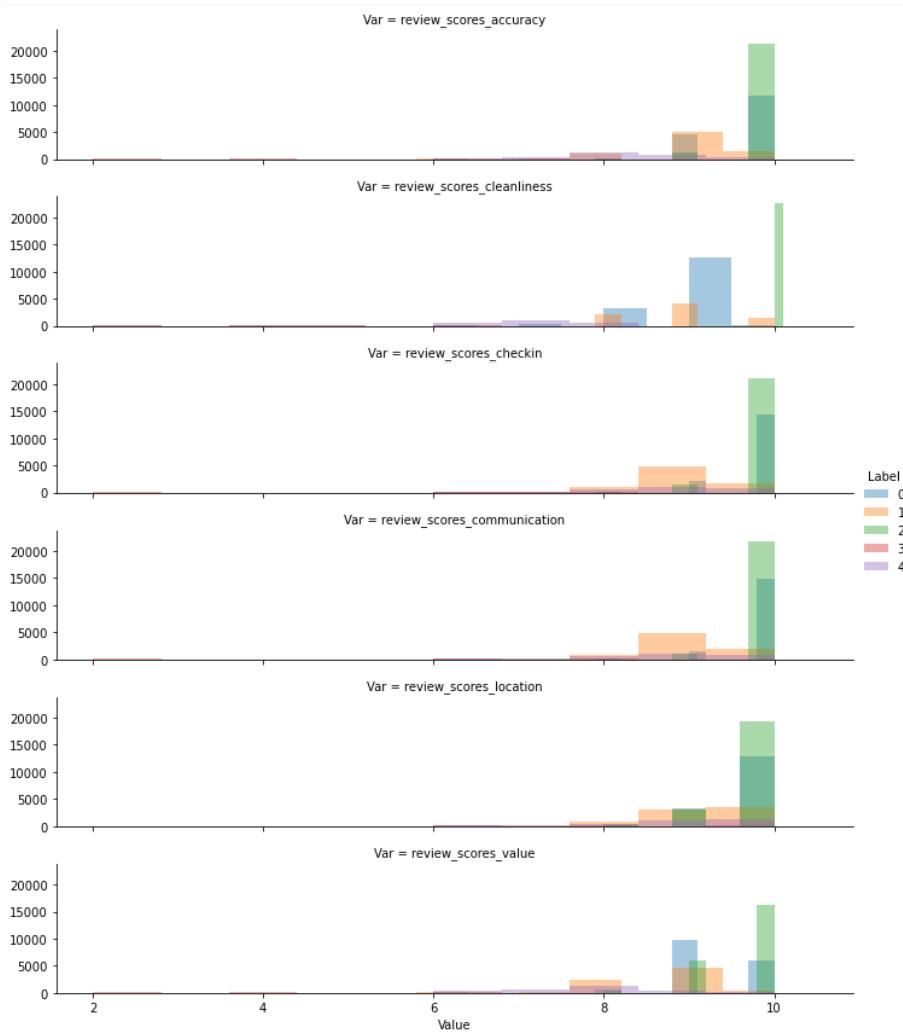
To explore the *distribution* of the values inside each cluster, rather than their mean, we can use the fancy FaceGrid approach:

```
tidy_db = db[review_areas]\n    .stack()\n    .reset_index()\n    .rename(columns={"level_1": "Var",\n                    "level_0": "ID",\n                    0: "Value"\n                })\n    .join(pandas.DataFrame({"Label": k5_pca}),\n          on="ID")\ntidy_db.head()
```

ID		Var	Value	Label
0	0	review_scores_accuracy	10.0	2
1	0	review_scores_cleanliness	10.0	2
2	0	review_scores_checkin	10.0	2
3	0	review_scores_communication	10.0	2
4	0	review_scores_location	10.0	2

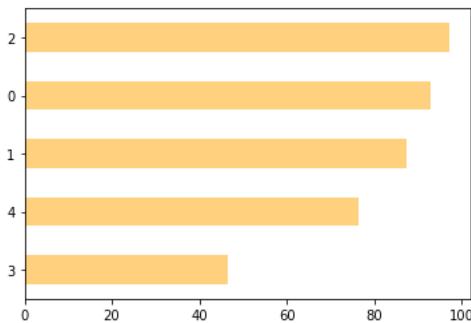
```
g = sns.FacetGrid(tidy_db,\n                  row="Var",\n                  hue="Label",\n                  height=2,\n                  aspect=5\n                 )\ng.map(sns.distplot,\n       "Value",\n       hist=True,\n       bins=10,\n       kde=False,\n       rug=False\n      )\ng.add_legend();
```

```
/opt/conda/lib/python3.8/site-packages/seaborn/distributions.py:2551: FutureWarning:
'distplot' is a deprecated function and will be removed in a future version. Please
adapt your code to use either `displot` (a figure-level function with similar
flexibility) or `histplot` (an axes-level function for histograms).
warnings.warn(msg, FutureWarning)
```



Externally:

```
# Cross with review_scores_rating
db.groupby(k5_pca)\n    ["review_scores_rating"]\n    .mean()\n    .sort_values()\n    .plot.barr(color="orange",\n               alpha=0.5\n               );
```



**EXERCISE** Can you cross the clustering results with property prices? Create:

- A bar plot with the average price by cluster
- A plot with the distribution (KDE/hist) of prices within cluster

Before we move on, let's also save the labels of the results:

```
pandas.DataFrame({"k5_pca": k5_pca})\n    .to_parquet("../data/k5_pca.parquet")
```

## Supervised Learning

To do before...

[\[Async\]](#)

Background

Here is a taster for supervised learning to make you want more, which we'll do on the live session. To prepare for this part of the course (which includes the [section below](#), as well as the one on [inference](#) and [overfitting/CV](#)), here are two tasks you will need to complete *before* the live session:

First, watch the following clip which sets up the scene for what will be to come

### Slides

The slides used in the clip are available at:

- [\[HTML\]](#)
- [\[PDF\]](#)



Second, read the following two articles, all mentioned in the clip:

- *Statistical Modeling: the Two Cultures* ([\[B+01\]](#))
- *Machine Learning: An Applied Econometric Approach\** ([\[MS17\]](#))

An open access, PDF version is available [here](#)

An open access, PDF version is available [here](#)

## Regression trees

Most of this block relies on linear regression. Linear regression is one of the simplest but also most powerful supervised algorithms. Later in the section, we will contrast its use with another technique that has gained much popularity since its inception at the turn of the millenium: random forests. A detailed understanding of random forests goes beyond the scope of this course. However, its intuition will be useful and will be covered here.

To understand random forests, we first need to learn about regression trees:

## Lecture 21: Regression Trees



A forest is no more than a collection of trees. A random forest thus will “grow” several trees, and then aggregate the predictions of each into a single, better prediction. This is what we call “bagging”:

## Lecture 21: Bagging



Building on these two ideas, the notion of a “random” forest makes a bit more sense:

## Lecture 21: Regression Trees



Action!

[\[Sync\]](#)

```
%matplotlib inline
from IPython.display import Image

import pandas
import numpy as np

db = pandas.read_csv("../data/paris_abb.csv.zip")
```

Linear Regression

$$P_i = \alpha + \beta X + \epsilon$$

```
import statsmodels.formula.api as sm
```

- Raw price

```
f = "Price ~ bathrooms + bedrooms + room_type"
lm_raw = sm.ols(f, db)\n    .fit()
lm_raw
```

```
<statsmodels.regression.linear_model.RegressionResultsWrapper at 0x7f458e6d1cd0>
```

```
lm_raw.summary()
```

<b>Dep. Variable:</b>	Price	<b>R-squared:</b>	0.103
<b>Model:</b>	OLS	<b>Adj. R-squared:</b>	0.103
<b>Method:</b>	Least Squares	<b>F-statistic:</b>	1151.
<b>Date:</b>	Thu, 07 Jan 2021	<b>Prob (F-statistic):</b>	0.00
<b>Time:</b>	17:24:33	<b>Log-Likelihood:</b>	-3.2312e+05
<b>No. Observations:</b>	50280	<b>AIC:</b>	6.462e+05
<b>Df Residuals:</b>	50274	<b>BIC:</b>	6.463e+05
<b>Df Model:</b>	5		
<b>Covariance Type:</b>	nonrobust		

#### OLS Regression Results

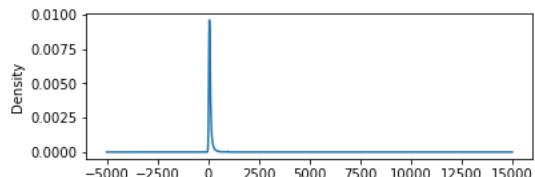
	coef	std err	t	P> t	[0.025	0.975]
<b>Intercept</b>	69.3589	1.295	53.573	0.000	66.821	71.896
<b>room_type[T.Hotel room]</b>	148.7074	4.779	31.120	0.000	139.341	158.073
<b>room_type[T.Private room]</b>	-51.5793	2.202	-23.425	0.000	-55.895	-47.264
<b>room_type[T.Shared room]</b>	-71.5010	8.692	-8.226	0.000	-88.537	-54.465
<b>bathrooms</b>	-2.4081	1.367	-1.762	0.078	-5.087	0.271
<b>bedrooms</b>	43.3410	0.939	46.174	0.000	41.501	45.181
<b>Omnibus:</b>	135133.909	<b>Durbin-Watson:</b>	1.923			
<b>Prob(Omnibus):</b>	0.000	<b>Jarque-Bera (JB):</b>	6632879884.203			
<b>Skew:</b>	32.680	<b>Prob(JB):</b>	0.00			
<b>Kurtosis:</b>	1781.140	<b>Cond. No.</b>	27.2			

#### Notes:

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

Now, *Price* is rather skewed:

```
db.Price.plot.kde(figsize=(6, 2));
```



We could take the log to try to obtain a better fit:

```
f = "np.log1p(Price) ~ bathrooms + bedrooms + room_type"
lm_log = sm.ols(f, db)\n    .fit()
lm_log
```

```
<statsmodels.regression.linear_model.RegressionResultsWrapper at 0x7f458e1484c0>
```

```
lm_log.rsquared
```

```
0.31515872008420753
```

```
lm_raw.rsquared
```

```
0.10269412705703151
```

That is *inference* though. We're here to “machine learn”!

```
# Get predictions
yp_raw = lm_raw.fittedvalues
yp_raw.head()
```

```
0    66.950833
1    66.950833
2    153.632865
3    110.291849
4    110.291849
dtype: float64
```

Or...

```
lm_raw.predict(db[["bathrooms",
                   "bedrooms",
                   "room_type"]
                  ])\n    .head()
```

```
0    66.950833
1    66.950833
2    153.632865
3    110.291849
4    110.291849
dtype: float64
```

The Machine Learner way

We need to:

- Get the dummies ourselves first

```
room_type_ds = pandas.get_dummies(db["room_type"])
room_type_ds.head(2)
```

```
Entire home/apt Hotel room Private room Shared room
```

	Entire home/apt	Hotel room	Private room	Shared room
0	1	0	0	0
1	1	0	0	0

- Prep x and y

```
x = pandas.concat([db[["bathrooms", "bedrooms"]],  
                   room_type_ds.drop("Entire home/apt",  
                                      axis=1)  
                  ], axis=1  
                 )  
x.head()
```

	bathrooms	bedrooms	Hotel room	Private room	Shared room
0	1.0	0.0	0	0	0
1	1.0	0.0	0	0	0
2	1.0	2.0	0	0	0
3	1.0	1.0	0	0	0
4	1.0	1.0	0	0	0

- Set up a model

```
from sklearn.linear_model import LinearRegression  
regressor = LinearRegression()
```

- Train the model (see the use of fit)

```
regressor.fit(X, db["Price"])
```

```
LinearRegression()
```

And voila, we have our results!

```
pandas.Series(regressor.coef_,  
              index=X.columns  
             )
```

```
bathrooms      -2.408064  
bedrooms       43.341016  
Hotel room     148.707387  
Private room   -51.579347  
Shared room    -71.501010  
dtype: float64
```

```
regressor.intercept_
```

```
69.35889717336228
```

```
lm_raw.params
```

```
Intercept          69.358897  
room_type[T.Hotel room] 148.707387  
room_type[T.Private room] -51.579347  
room_type[T.Shared room] -71.501010  
bathrooms        -2.408064  
bedrooms         43.341016  
dtype: float64
```

## Tree-based approaches: the Random Forest

```
from sklearn.ensemble import RandomForestRegressor
```

Very similar API (as throughout sklearn). Two parameters to set (see [here](#) for guidance):

```
rf_raw = RandomForestRegressor(n_estimators=100,
                               max_features=None
                               )
%time rf_raw.fit(X, db["Price"])
```

```
CPU times: user 750 ms, sys: 0 ns, total: 750 ms
Wall time: 749 ms
```

```
RandomForestRegressor(max_features=None)
```

To recover the predictions, we need to rely on predict:

```
rf_raw_lbls = rf_raw.predict(X)
rf_raw_lbls
```

```
array([ 71.98778067,  71.98778067, 140.74061093, ...,  93.62649936,
       71.98778067,  93.62649936])
```

For completeness, let's quickly fit a Random Forest on the log of price:

```
rf_log = RandomForestRegressor(n_estimators=100,
                               max_features=None
                               )
%time rf_log.fit(X, np.log1p(db["Price"]))
rf_log_lbls = rf_log.predict(X)
```

```
CPU times: user 590 ms, sys: 0 ns, total: 590 ms
Wall time: 589 ms
```

## EXERCISE

- Train a random forest on the price using the number of beds and the property type instead

Before we move on, let's record all the predictions in a single table for convenience:

```
res = pandas.DataFrame({
    "LM-Raw": lm_raw.fittedvalues,
    "LM-Log": lm_log.fittedvalues,
    "RF-Raw": rf_raw_lbls,
    "RF-Log": rf_log_lbls,
    "Truth": db["Price"]
})
res.head()
```

	LM-Raw	LM-Log	RF-Raw	RF-Log	Truth
0	66.950833	4.198030	71.987781	4.181397	60.0
1	66.950833	4.198030	71.987781	4.181397	115.0
2	153.632865	4.850742	140.740611	4.817468	119.0
3	110.291849	4.524386	93.626499	4.444683	130.0
4	110.291849	4.524386	93.626499	4.444683	75.0

And write it out to a file:

```
res.to_parquet("../data/lm_results.parquet")
```

An example with categorical outcomes

Let's bring back the classification we did in the previous session:

```
k5_pca = pandas.read_parquet("../data/k5_pca.parquet")\
    .reindex(db.index)
k5_pca.head()
```

```
k5_pca
```

0	2
1	1
2	0
3	2
4	2

Now we might conceive cases where we want to build a model to predict these classes based on some house characteristics. To illustrate it, let's consider the same variables as above. In this case, however, we want a *classifier* rather than a *regressor*, as the response is categorical.

```
from sklearn.ensemble import RandomForestClassifier
```

But the training is very similar:

```
%%time
classifier = RandomForestClassifier(n_estimators=100,
                                    max_features="sqrt")
classifier.fit(X, k5_pca["k5_pca"])

pred_lbls = pandas.Series(classifier.predict(X),
                           index=k5_pca.index)
```

```
CPU times: user 1.23 s, sys: 0 ns, total: 1.23 s
Wall time: 1.23 s
```

```
class_res = pandas.DataFrame({"Truth": k5_pca["k5_pca"],
                               "Predicted": pred_lbls
                             })
class_res.apply(pandas.Categorical)
class_res.describe()
```

	Truth	Predicted
count	50280	50280
unique	5	5
top	2	2
freq	22773	49801

## Inference

[\[Sync\]](#)

```
%matplotlib inline
from IPython.display import Image

import pandas
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
import statsmodels.formula.api as sm
from sklearn.preprocessing import scale
from sklearn import metrics
```

```
db = pandas.read_csv('../data/paris_abb.csv.zip')
db['l_price'] = np.log1p(db['Price'])
db.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 50280 entries, 0 to 50279
Data columns (total 11 columns):
 #   Column           Non-Null Count  Dtype  
 ---  --  
 0    id               50280 non-null   int64  
 1    neighbourhood_cleansed 50280 non-null   object  
 2    property_type      50280 non-null   object  
 3    room_type          50280 non-null   object  
 4    accommodates       50280 non-null   int64  
 5    bathrooms          50280 non-null   float64 
 6    bedrooms           50280 non-null   float64 
 7    beds               50280 non-null   float64 
 8    bed_type           50280 non-null   object  
 9    Price              50280 non-null   float64 
 10   l_price            50280 non-null   float64 
dtypes: float64(5), int64(2), object(4)
memory usage: 4.2+ MB
```

## Baseline model

$$\log(P) = \alpha + X\beta + \epsilon$$

$X$ :

- Bathrooms
- Bedrooms
- Beds
- Room type

```
m1 = sm.ols('l_price ~ bedrooms + bathrooms + beds', db).fit()
m1.summary()
```



	<b>bedrooms</b>	<b>bathrooms</b>	<b>beds</b>
<b>count</b>	5.028000e+04	5.028000e+04	5.028000e+04
<b>mean</b>	1.748138e-14	1.621070e-15	-5.895298e-15
<b>std</b>	1.000010e+00	1.000010e+00	1.000010e+00
<b>min</b>	-1.076738e+00	-1.625116e+00	-1.429042e+00
<b>25%</b>	-7.772733e-02	-1.702805e-01	-5.706025e-01
<b>50%</b>	-7.772733e-02	-1.702805e-01	-5.706025e-01
<b>75%</b>	-7.772733e-02	-1.702805e-01	2.878366e-01
<b>max</b>	4.887380e+01	7.111664e+01	4.149292e+01

```
m2 = sm.ols('l_price ~ bedrooms + bathrooms + beds',
            data=scX.join(db['l_price']))\
            .fit()
m2.summary()
```

**Dep. Variable:** l\_price **R-squared:** 0.245  
**Model:** OLS **Adj. R-squared:** 0.245  
**Method:** Least Squares **F-statistic:** 5433.  
**Date:** Thu, 07 Jan 2021 **Prob (F-statistic):** 0.00  
**Time:** 17:25:24 **Log-Likelihood:** -38592.  
**No. Observations:** 50280 **AIC:** 7.719e+04  
**Df Residuals:** 50276 **BIC:** 7.723e+04  
**Df Model:** 3  
**Covariance Type:** nonrobust

OLS Regression Results

	coef	std err	t	P> t	[0.025	0.975]
<b>Intercept</b>	4.4837	0.002	1928.462	0.000	4.479	4.488
<b>bedrooms</b>	0.2255	0.004	57.760	0.000	0.218	0.233
<b>bathrooms</b>	-0.1061	0.003	-32.373	0.000	-0.113	-0.100
<b>beds</b>	0.1628	0.003	50.554	0.000	0.156	0.169
<b>Omnibus:</b>	11384.882	<b>Durbin-Watson:</b>		1.846		
<b>Prob(Omnibus):</b>	0.000	<b>Jarque-Bera (JB):</b>	585415.122			
<b>Skew:</b>	-0.096	<b>Prob(JB):</b>		0.00		
<b>Kurtosis:</b>	19.715	<b>Cond. No.</b>		3.11		

Notes:

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

Let's bring both sets of results together:

```
pandas.DataFrame({"Baseline": m1.params,  
                  "X Std.": m2.params  
                 })
```

	Baseline	X Std.
Intercept	4.180599	4.483657
bedrooms	0.225325	0.225548
bathrooms	-0.154385	-0.106119
beds	0.139759	0.162806

To note:

How does interpretation of the coefficients change?

- Meaning of intercept when  $X$  is demeaned
- Units in which  $\beta$  are interpreted

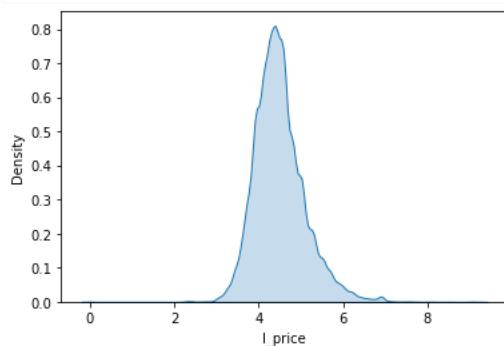
## Predictive checking

Is the model picking up the overall “shape of data”?

- Important to know how much we should trust our inferences
- Crucial if we want to use the model to predict!

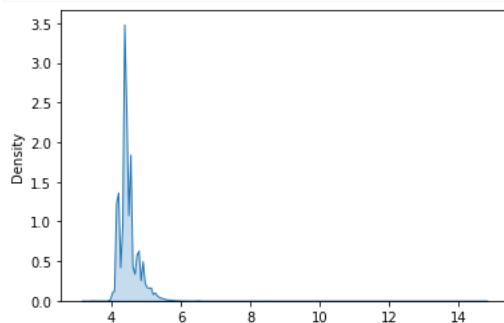
```
sns.kdeplot(db['l_price'], shade=True)
```

```
<AxesSubplot:xlabel='l_price', ylabel='Density'>
```

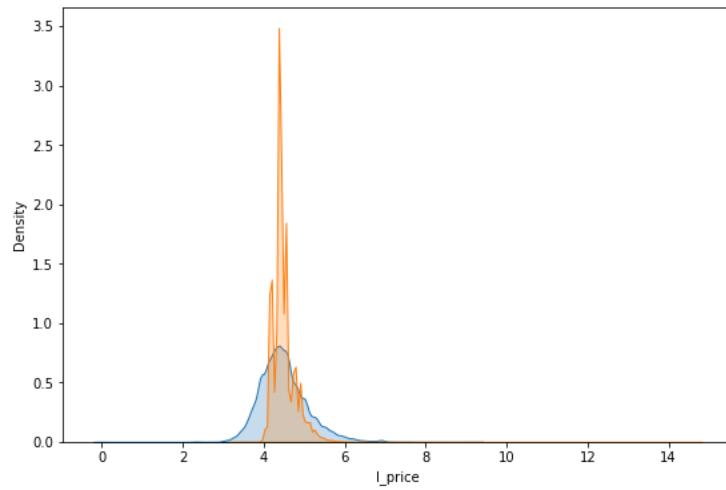


```
sns.kdeplot(m1.fittedvalues, shade=True)
```

```
<AxesSubplot:ylabel='Density'>
```



```
f, ax = plt.subplots(1, figsize=(9, 6))  
sns.kdeplot(db['l_price'], shade=True, ax=ax, label='$y$')  
sns.kdeplot(m1.fittedvalues, shade=True, ax=ax, label='$\hat{y}$')  
plt.show()
```



To note:

- Not a terrible start
- How could we improve it?

```
m3 = sm.ols('l_price ~ bedrooms + bathrooms + beds + room_type', db).fit()
m3.summary()
```

**Dep. Variable:** l\_price      **R-squared:** 0.340  
**Model:** OLS      **Adj. R-squared:** 0.340  
**Method:** Least Squares      **F-statistic:** 4314.  
**Date:** Thu, 07 Jan 2021      **Prob (F-statistic):** 0.00  
**Time:** 17:25:25      **Log-Likelihood:** -35210.  
**No. Observations:** 50280      **AIC:** 7.043e+04  
**Df Residuals:** 50273      **BIC:** 7.050e+04  
**Df Model:** 6  
**Covariance Type:** nonrobust

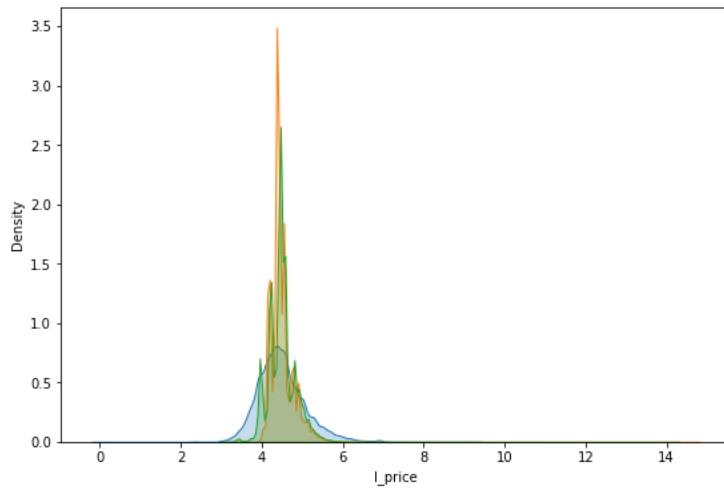
OLS Regression Results

	coef	std err	t	P> t	[0.025	0.975]
<b>Intercept</b>	4.2417	0.005	899.941	0.000	4.232	4.251
<b>room_type[T.Hotel room]</b>	0.5548	0.016	35.619	0.000	0.524	0.585
<b>room_type[T.Private room]</b>	-0.4862	0.007	-66.604	0.000	-0.501	-0.472
<b>room_type[T.Shared room]</b>	-1.0320	0.028	-36.320	0.000	-1.088	-0.976
<b>bedrooms</b>	0.2388	0.004	65.134	0.000	0.232	0.246
<b>bathrooms</b>	-0.1440	0.004	-32.285	0.000	-0.153	-0.135
<b>beds</b>	0.1144	0.003	43.386	0.000	0.109	0.120
<b>Omnibus:</b>	11695.728	<b>Durbin-Watson:</b>	1.878			
<b>Prob(Omnibus):</b>	0.000	<b>Jarque-Bera (JB):</b>	680804.847			
<b>Skew:</b>	-0.027	<b>Prob(JB):</b>	0.00			
<b>Kurtosis:</b>	21.027	<b>Cond. No.</b>	37.2			

Notes:

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

```
f, ax = plt.subplots(1, figsize=(9, 6))
sns.kdeplot(db['l_price'], shade=True, ax=ax, label='$y$')
sns.kdeplot(m1.fittedvalues, shade=True, ax=ax, label='$\hat{y}_1$')
sns.kdeplot(m3.fittedvalues, shade=True, ax=ax, label='$\hat{y}_2$')
plt.show()
```



To note:

- This is better!
- But these are only point predictions. Sometimes that's good enough.
- Usually however, we want a model to capture the underlying process instead of the particular realisation observed (ie. dataset).
- Then we need to think about the uncertainty embedded in the model we are estimating

## Inferential Vs Predictive uncertainty

[See more in Chapter 7.2 of [Gelman & Hill 2006](#) ]

- Two types of uncertainty in our model
  - Predictive ( $\epsilon$ )
  - Inferential ( $\beta$ )
- Both affect the final predictions we make

```
Image("../figs/abb_room.png", retina=True)
```

[Beautiful flat next to Montparnasse Station](#)

Share Save

**Beautiful flat next to Montparnasse Station**

Paris

5 guests   2 bedrooms   3 beds   1 bathroom

Entire home  
You'll have the apartment to yourself.

Add dates for prices Add dates

Raphaëlle

```
room = db.loc[db['id']==35607436, :]  
room.T
```

**45979**

<b>id</b>	35607436
<b>neighbourhood_cleansed</b>	Luxembourg
<b>property_type</b>	Apartment
<b>room_type</b>	Entire home/apt
<b>accommodes</b>	5
<b>bathrooms</b>	1
<b>bedrooms</b>	2
<b>beds</b>	3
<b>bed_type</b>	Real Bed
<b>Price</b>	90
<b>I_price</b>	4.51086

```
rid = room.index[0]
db.loc[rid, :]
```

```
id                      35607436
neighbourhood_cleansed    Luxembourg
property_type              Apartment
room_type                  Entire home/apt
accommodates                   5
bathrooms                     1
bedrooms                      2
beds                         3
bed_type                      Real Bed
Price                         90
l_price                       4.51086
Name: 45979, dtype: object
```

$$\log(\hat{P}_i) = \alpha + \sum_k \beta_k * X_k$$

```
m1.params['Intercept'] + db.loc[rid, cols].dot(m1.params[cols])
```

```
4.896141184855212
```

To note:

- What does dot do?

```
m1.fittedvalues[rid]
```

```
4.896141184855212
```

Point predictive simulation

```
%%time
# Parameters
## Number of simulations
r = 2000
# Pull out characteristics for house of interest
x_i = db.loc[rid, cols]
# Specify model engine
model = m1

# Place-holder
sims = np.zeros(r)
# Loop over number of replications
for i in range(r):
    # Get a random draw of betas
    rbs = np.random.normal(model.params, model.bse)
    # Get a random draw of epsilon
    re = np.random.normal(0, model.scale)
    # Obtain point estimate
    y_hr = rbs[0] + np.dot(x_i, rbs[1:]) + re
    # Store estimate
    sims[i] = y_hr
```

```
CPU times: user 637 ms, sys: 35.1 ms, total: 672 ms
Wall time: 632 ms
```

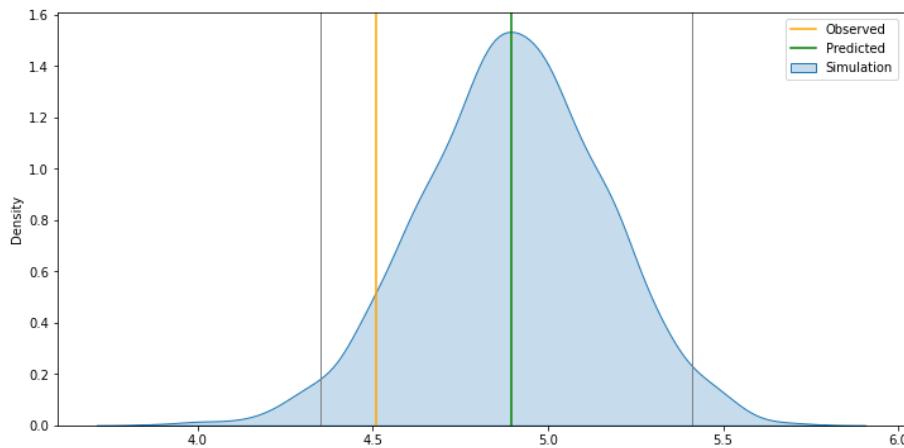
```

f, ax = plt.subplots(1, figsize=(12, 6))

sns.kdeplot(sims, shade=True, ax=ax, label='Simulation')
ax.axvline(db.loc[rid, 'l_price'], c='orange', label='Observed')
ax.axvline(model.fittedvalues[rid], c='green', label='Predicted')

lo, up = pandas.Series(sims)\n    .sort_values()\n    .iloc[[int(np.round(0.025 * r)), int(np.round(0.975 * r))]]\n\nax.axvline(lo, c='grey', linewidth=1)\nax.axvline(up, c='grey', linewidth=1)\n\nplt.legend()\nplt.show()

```



To note:

- Intuition of the simulation
- The for loop, deconstructed
- The graph, bit by bit
- If we did this for every observation, we'd expect 95% to be within the 95% bands

### Exercise

Explore with the code above and try to generate similar plots for:

- Different houses across locations and characteristics
- Different model

### Exercise+

Recreate the analysis above for observation 5389821. What happens? Why?

Now, we could do this for *all* the observations and get a sense of the overall distribution to be expected

```

%%time
# Parameters
## Number of observations & simulations
n = db.shape[0]
r = 200
# Specify model engine
model = m1

# Place-holder (N, r)
sims = np.zeros((n, r))
# Loop over number of replications
for i in range(r):
    # Get a random draw of betas
    rbs = np.random.normal(model.params, model.bse)
    # Get a random draw of epsilon
    re = np.random.normal(0*n, model.scale)
    # Obtain point estimate
    y_hr = rbs[0] + np.dot(db[cols], rbs[1:]) + re
    # Store estimate
    sims[:, i] = y_hr

```

CPU times: user 5.61 s, sys: 27.3 s, total: 32.9 s  
Wall time: 2.08 s

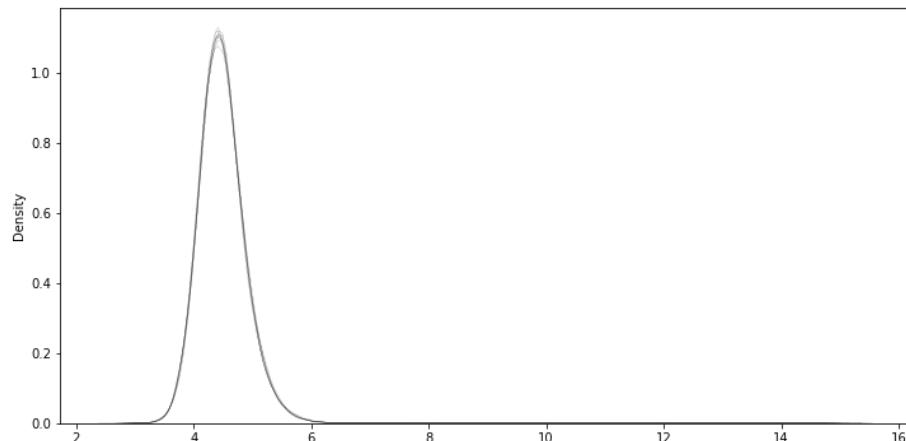
```

f, ax = plt.subplots(1, figsize=(12, 6))

for i in range(10):
    sns.kdeplot(sims[:, i], ax=ax, linewidth=0.1, alpha=0.1, color='k')

plt.show()

```



```

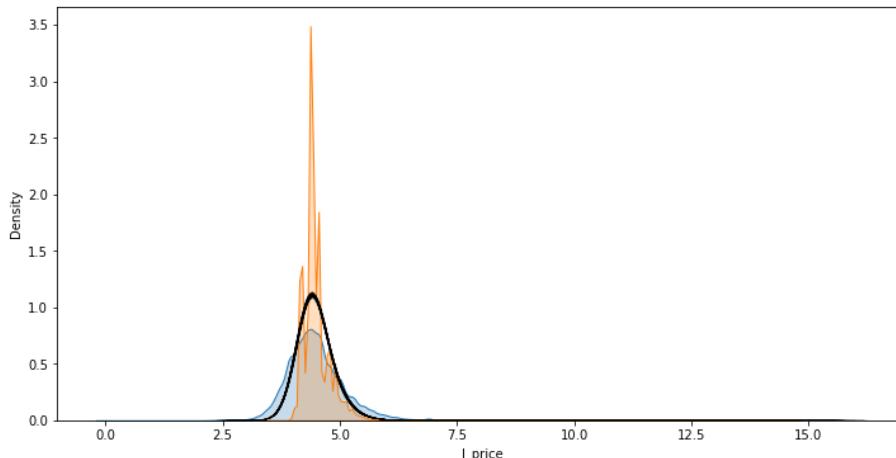
f, ax = plt.subplots(1, figsize=(12, 6))

sns.kdeplot(db['l_price'], shade=True, ax=ax, label='$y$')
sns.kdeplot(m1.fittedvalues, shade=True, ax=ax, label='$\hat{y}$')

for i in range(r):
    sns.kdeplot(sims[:, i], ax=ax, linewidth=0.1, alpha=0.1, color='k')

plt.show()

```



To note:

- Black line contains  $r$  thin lines that collectively capture the uncertainty behind the model

New data

Imagine we are trying to figure out how much should we charge for a property we want to put on Airbnb.

For example, let's assume our property is:

```
new = pandas.Series({'bedrooms': 4,
                     'bathrooms': 1,
                     'beds': 8})
```

```
%time
# Parameters
## Number of simulations
r = 2000
# Pull out characteristics for house of interest
x_i = new
# Specify model engine
model = m1

# Place-holder
sims = np.zeros(r)
# Loop over number of replications
for i in range(r):
    # Get a random draw of betas
    rbs = np.random.normal(model.params, model.bse)
    # Get a random draw of epsilon
    re = np.random.normal(0, model.scale)
    # Obtain point estimate
    y_hr = rbs[0] + np.dot(x_i, rbs[1:]) + re
    # Store estimate
    sims[i] = y_hr
```

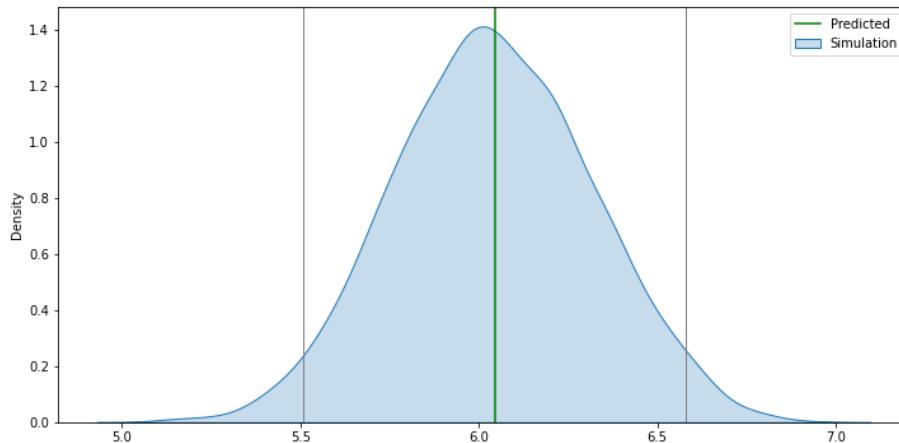
```
CPU times: user 640 ms, sys: 4.53 ms, total: 644 ms
Wall time: 643 ms
```

```
f, ax = plt.subplots(1, figsize=(12, 6))

sns.kdeplot(sims, shade=True, ax=ax, label='Simulation')
ax.axvline(model.params.iloc[0] + np.dot(new, model.params.iloc[1:]), \
           c='green', label='Predicted')

lo, up = pandas.Series(sims)\n            .sort_values()\n            .iloc[[int(np.round(0.025 * r)), int(np.round(0.975 * r))]]
ax.axvline(lo, c='grey', linewidth=1)
ax.axvline(up, c='grey', linewidth=1)

plt.legend()
plt.show()
```



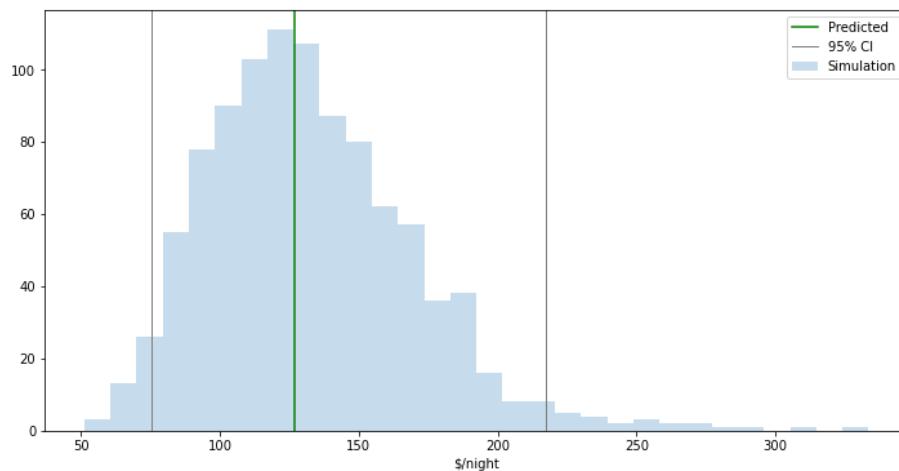
[Pro]

```

def predictor(bedrooms, bathrooms, beds):
    new = pandas.Series({'bedrooms': bedrooms,
                         'bathrooms': bathrooms,
                         'beds': beds
                        })
    r = 1000
    x_i = new
    model = m1
    y_hat = model.params.iloc[0] + np.dot(new, model.params.iloc[1:])
    # Simulation
    sims = np.zeros(r)
    for i in range(r):
        rbs = np.random.normal(model.params, model.bse)
        re = np.random.normal(0, model.scale)
        y_hr = rbs[0] + np.dot(x_i, rbs[1:]) + re
        sims[i] = y_hr
    sims = np.exp(sims)
    y_hat = np.exp(y_hat)
    # Bands
    lo, up = pandas.Series(sims)\n        .sort_values()\n        .iloc[[int(np.round(0.025 * r)), int(np.round(0.975 * r))]]\n    # Setup'n'draw figure\n    f, ax = plt.subplots(1, figsize=(12, 6))\n    ax.hist(sims, label='Simulation', alpha=0.25, bins=30)\n    ax.axvline(y_hat, c='green', label='Predicted')\n    ax.axvline(lo, c='grey', linewidth=1, label='95% CI')\n    ax.axvline(up, c='grey', linewidth=1)\n    #ax.set_xlim((0, 10))\n    # Dress up\n    ax.set_xlabel("$/night")\n    plt.legend()\n    return plt.show()

predictor(3, 1, 1)

```



```

# You might have to run this to make interactives work
# jupyter labextension install @jupyter-widgets/jupyterlab-manager
# From https://ipywidgets.readthedocs.io/en/latest/user_install.html#installing-the-jupyterlab-extension
# Then restart Jupyter Lab
from ipywidgets import interact, IntSlider

interact(predictor,
          bedrooms=IntSlider(min=1, max=10), \
          bathrooms=IntSlider(min=0, max=10), \
          beds=IntSlider(min=1, max=20)
         );

```

## Model performance

To note:

- Switch from inference to prediction
- Overall idea of summarising model performance
- $R^2$
- Error-based measures

```

# R^2
r2 = pandas.Series({'Baseline': metrics.r2_score(db['l_price'],
                                                m1.fittedvalues),
                     'Augmented': metrics.r2_score(db['l_price'],
                                                m3.fittedvalues)})
r2

```

```

Baseline    0.244826
Augmented   0.339876
dtype: float64

```

```

# MSE
mse = pandas.Series({'Baseline': metrics.mean_squared_error(db['l_price'],
                                                             m1.fittedvalues),
                      'Augmented': metrics.mean_squared_error(db['l_price'],
                                                             m3.fittedvalues)})
mse

```

```

Baseline    0.271771
Augmented   0.237565
dtype: float64

```

```

# MAE
mae = pandas.Series({'Baseline': metrics.mean_absolute_error(db['l_price'],
                                                               m1.fittedvalues),
                      'Augmented': metrics.mean_absolute_error(db['l_price'],
                                                               m3.fittedvalues)})
mae

```

```

Baseline    0.379119
Augmented   0.357287
dtype: float64

```

```

# All
perf = pandas.DataFrame({'MAE': mae,
                         'MSE': mse,
                         'R^2': r2})
perf

```

	MAE	MSE	$R^2$
--	-----	-----	-------

Baseline	0.379119	0.271771	0.244826
----------	----------	----------	----------

Augmented	0.357287	0.237565	0.339876
-----------	----------	----------	----------

# Overfitting & Cross-Validation

[Sync]

```
%matplotlib inline
import pandas
import seaborn as sns
import matplotlib.pyplot as plt
from numpy import exp, log1p, sqrt

base = pandas.read_csv("../data/paris_abb.csv.zip")
res = pandas.read_parquet("../data/lm_results.parquet")
db = base.join(res)
```

## Overfitting

So we have models but, are they any good?

Model evaluation the ML is all about predictive performance (remember the  $\hat{\beta}$  Vs  $\hat{y}$ , this is all about  $\hat{y}$ ).

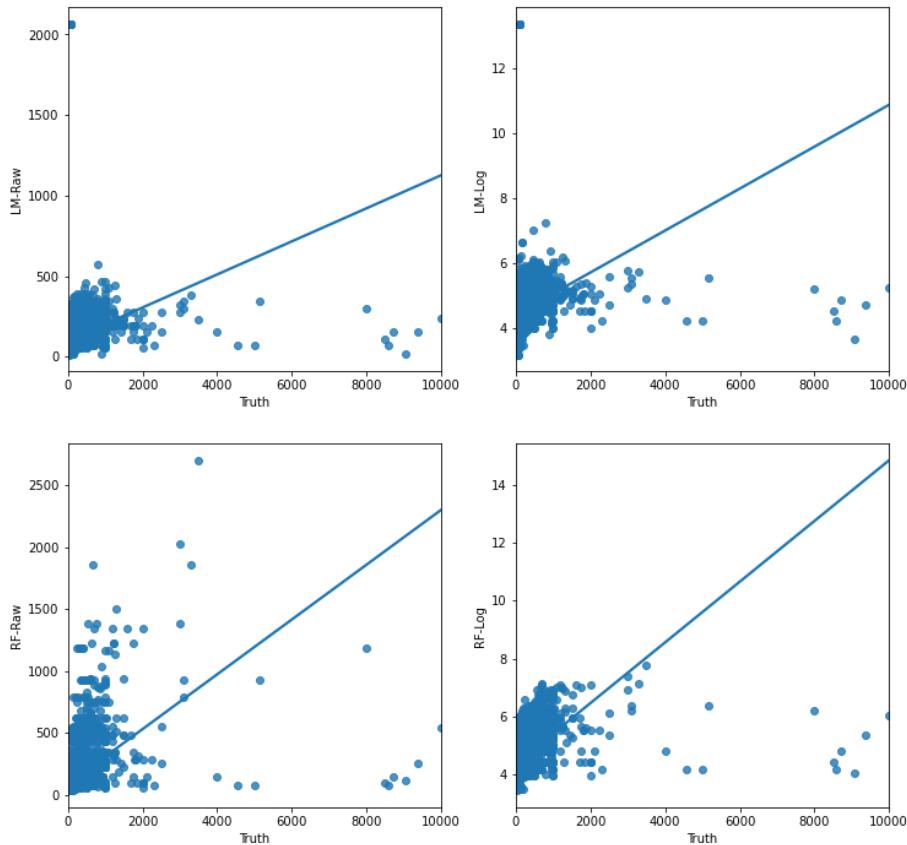
```
f, axs = plt.subplots(2, 2, figsize=(12, 12))
axs = axs.flatten()
for i, m in enumerate(res.columns.drop("Truth")):
    ax = axs[i]
    sns.regplot("Truth",
                m,
                res,
                ci=None,
                ax=ax
            )
f.suptitle(f"Observed Vs Predicted")
plt.show()
```

```

/opt/conda/lib/python3.8/site-packages/seaborn/_decorators.py:36: FutureWarning: Pass
the following variables as keyword args: x, y, data. From version 0.12, the only
valid positional argument will be 'data', and passing other arguments without an
explicit keyword will result in an error or misinterpretation.
    warnings.warn(
/opt/conda/lib/python3.8/site-packages/seaborn/_decorators.py:36: FutureWarning: Pass
the following variables as keyword args: x, y, data. From version 0.12, the only
valid positional argument will be 'data', and passing other arguments without an
explicit keyword will result in an error or misinterpretation.
    warnings.warn(
/opt/conda/lib/python3.8/site-packages/seaborn/_decorators.py:36: FutureWarning: Pass
the following variables as keyword args: x, y, data. From version 0.12, the only
valid positional argument will be 'data', and passing other arguments without an
explicit keyword will result in an error or misinterpretation.
    warnings.warn(
/opt/conda/lib/python3.8/site-packages/seaborn/_decorators.py:36: FutureWarning: Pass
the following variables as keyword args: x, y, data. From version 0.12, the only
valid positional argument will be 'data', and passing other arguments without an
explicit keyword will result in an error or misinterpretation.
    warnings.warn(

```

Observed Vs Predicted



There are several measures, we'll use two:

- $R^2$

```
from sklearn.metrics import r2_score
```

Let's compare apparent performance:

```

r2s = pandas.Series({ "LM-Raw": r2_score(db["Truth"],
                                         db["LM-Raw"]
                                         ),
                      "LM-Log": r2_score(log1p(db["Truth"]),
                                         db["LM-Log"]
                                         ),
                      "RF-Raw": r2_score(db["Truth"],
                                         db["RF-Raw"]
                                         ),
                      "RF-Log": r2_score(log1p(db["Truth"]),
                                         db["RF-Log"]
                                         ),
                    })
r2s

```

```

LM-Raw    0.102694
LM-Log    0.315159
RF-Raw    0.225217
RF-Log    0.443318
dtype: float64

```

- (R)MSE

```

from sklearn.metrics import mean_squared_error as mse

```

And a similar comparison (where we can convert all predictions to price units):

```

mses = pandas.Series({ "LM-Raw": mse(db["Truth"],
                                         db["LM-Raw"]
                                         ),
                      "LM-Log": mse(db["Truth"],
                                         exp(db["LM-Log"])
                                         ),
                      "RF-Raw": mse(db["Truth"],
                                         db["RF-Raw"]
                                         ),
                      "RF-Log": mse(db["Truth"],
                                         exp(db["RF-Log"])
                                         ),
                    }).apply(sqrt)
mses

```

```

LM-Raw    149.522153
LM-Log    7570.085725
RF-Raw    138.939346
RF-Log    140.904608
dtype: float64

```

Now this is great news for the random forest!

... or is it?

Let's do a thought experiment. Imagine that AirBnb, for some reason, did not initially move into the Passy neighbourhood.

Let's quickly retrain our models for that world. For convenience, let's focus on the raw models, not those based on logs:

```

room_type_ds = pandas.get_dummies(db["room_type"])
X = pandas.concat([db[["bathrooms", "bedrooms"]],
                  room_type_ds.drop("Entire home/apt",
                                     axis=1)
                  ],
                  axis=1
                )
# Passy IDs
pi = db.query("neighbourhood_cleaned == 'Passy'").index
# Non Passy IDs
npi = db.index.difference(pi)

# Non Passy data (same as in previous notebook)
X_np = X.reindex(npi)

```

```

from sklearn.ensemble import RandomForestRegressor
from sklearn.linear_model import LinearRegression

# Linear model
lm_raw_np_model = LinearRegression().fit(X_np,
                                         db.loc[npi, "Price"]
                                         )
lm_raw_np = lm_raw_np_model.predict(X_np)
# RF
rf_raw_np_model = RandomForestRegressor(n_estimators=100,
                                         max_features=None
                                         )\
    .fit(X_np,
         db.loc[npi, "Price"]
         )
rf_raw_np = rf_raw_np_model.predict(X_np)

```

The *apparent* performance of our model is similar, a bit better if anything:

```

mses_np = pandas.Series({"LM-Raw": mse(db.loc[npi, "Truth"],
                                         lm_raw_np
                                         ),
                         "RF-Raw": mse(db.loc[npi, "Truth"],
                                       rf_raw_np
                                       ),
                         }).apply(sqrt)
mses_np

```

```

LM-Raw      144.326619
RF-Raw      134.277649
dtype: float64

```

```
mses.reindex(mses_np.index)
```

```

LM-Raw      149.522153
RF-Raw      138.939346
dtype: float64

```

Now imagine that Passy comes on the AirBnb market and we need to provide price estimates for its properties. We can use our model to make predictions:

```

# Linear predictions
lm_raw_passy_pred = lm_raw_np_model.predict(X.reindex(pi))
lm_raw_passy_pred = pandas.Series(lm_raw_passy_pred,
                                   index=pi
                                   )
# RF predictions
rf_raw_passy_pred = rf_raw_np_model.predict(X.reindex(pi))
fr_raw_passy_pred = pandas.Series(rf_raw_passy_pred,
                                   index=pi
                                   )

```

How's our model doing now?

```

mses_p = pandas.Series({"LM-Raw": mse(db.loc[pi, "Truth"],
                                         lm_raw_passy_pred
                                         ),
                         "RF-Raw": mse(db.loc[pi, "Truth"],
                                       fr_raw_passy_pred
                                       ),
                         }).apply(sqrt)
mses_p

```

```

LM-Raw      235.291495
RF-Raw      218.293239
dtype: float64

```

```
mses_np
```

```
LM-Raw    144.326619
RF-Raw    134.277649
dtype: float64
```

Not that great on data the models have not seen before.

Linear regression is a particular case in that, given the features chosen, one cannot tweak much more; but the RF does allow us to tweak things slightly differently. How is a matter of (computational) brute force and a bit of savvy-ness in designing the data split.

## Cross-Validation to the rescue

- What it does: give you a better sense of the actual performance of your model
- What it doesn't (estimate a better model per-se)

Train/test split

```
from sklearn.model_selection import train_test_split

x_train, x_test, y_train, y_test = train_test_split(X,
                                                    db["Price"],
                                                    test_size=0.8
                                                   )
```

For the linear model:

```
lm_estimator = LinearRegression()
# Fit on X/Y train
lm_estimator.fit(x_train, y_train)
# Predict on X test
lm_y_pred = lm_estimator.predict(x_test)
# Evaluate on Y test
sqrt(mse(y_test, lm_y_pred))
```

```
146.63691841889124
```

For the random forest:

```
rf_estimator = RandomForestRegressor(n_estimators=100,
                                      max_features=None
                                     )
# Train on X/Y train
rf_estimator.fit(x_train, y_train)
# Predict on X test
rf_y_pred = rf_estimator.predict(x_test)
# Evaluate on Y test
sqrt(mse(y_test, rf_y_pred))
```

```
140.59053185334008
```

Worse than in-testing, but better than on an entirely new set of data with (potentially) different structure.

Now, the splitting that `train_test_split` does is random. What if it happens to be very particular? Can we trust the performance scores we recover?

*k*-fold CV

```
from sklearn.model_selection import cross_val_score
```

For the linear model:

```

lm_kcv_msres = cross_val_score(LinearRegression(),
                               X,
                               db[["Price"]],
                               cv=5,
                               scoring="neg_mean_squared_error"
                              )
# sklearn uses neg to optimisation is always maximisation
sqrt(-lm_kcv_msres).mean()

```

156.0444939069434

And for the random forest:

```

rf_estimator = RandomForestRegressor(n_estimators=100,
                                     max_features=None
                                    )
rf_kcv_msres = cross_val_score(rf_estimator,
                               X,
                               db[["Price"]],
                               cv=5,
                               scoring="neg_mean_squared_error"
                              )
# sklearn uses neg to optimisation is always maximisation
sqrt(-rf_kcv_msres).mean()

```

141.65779397663923

These would be more reliable measures of model performance. The difference between CV'ed estimates and original ones can be seen as an indication of overfitting.

mses

LM-Raw	149.522153
LM-Log	7570.085725
RF-Raw	138.939346
RF-Log	140.904608
dtype:	float64

In our case, since we're using very few features, both models probably already feature enough "regularisation" and the changes are not very dramatic. In other contexts, this can be a drastic change (e.g. [Arribas-Bel, Patino & Duque; 2017](#)).

### Parameter optimisation in scikit-learn

Random Forests really can be optimised over two key parameters, `n_estimators` and `max_features`, although there are more to tweak around:

```
RandomForestRegressor().get_params()
```

```
{
  'bootstrap': True,
  'ccp_alpha': 0.0,
  'criterion': 'mse',
  'max_depth': None,
  'max_features': 'auto',
  'max_leaf_nodes': None,
  'max_samples': 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': 100,
  'n_jobs': None,
  'oob_score': False,
  'random_state': None,
  'verbose': 0,
  'warm_start': False
}
```

Here we'll exhaustively test each combination for several values of the parameters:

```
param_grid = {  
    "n_estimators": [5, 25, 50, 75, 100, 150],  
    "max_features": [1, 2, 3, 4, 5]  
}
```

We will use `GridSearchCV` to automatically work over every combination, create cross-validated scores (MSE), and pick the preferred one.

```
from sklearn.model_selection import GridSearchCV
```

This can get fairly computationally intensive, and it is also “embarrassingly parallel”, so it's a good candidate to parallelise.

```
grid = GridSearchCV(RandomForestRegressor(),  
                    param_grid,  
                    scoring="neg_mean_squared_error",  
                    cv=5,  
                    n_jobs=-1  
)
```

Once defined, we `fit` to actually execute computations on a given set of data:

```
%%time  
grid.fit(X, db["Price"])
```

```
CPU times: user 631 ms, sys: 165 ms, total: 796 ms  
Wall time: 10.5 s
```

```
GridSearchCV(cv=5, estimator=RandomForestRegressor(), n_jobs=-1,  
            param_grid={'max_features': [1, 2, 3, 4, 5],  
                        'n_estimators': [5, 25, 50, 75, 100, 150]},  
            scoring='neg_mean_squared_error')
```

And we can explore the output from the same `grid` object:

```
grid_res = pandas.DataFrame(grid.cv_results_)  
grid_res.head()
```

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_max_fe
0	0.059308	0.002531	0.007023	0.000338	
1	0.260195	0.003964	0.019304	0.000458	
2	0.519160	0.003526	0.035556	0.000450	
3	0.818000	0.017073	0.054218	0.002030	
4	1.071697	0.034987	0.065108	0.001701	

And the winner is...

```
grid_res[grid_res["mean_test_score"] \n    == \n        grid_res["mean_test_score"].max() \n    ]["params"]
```

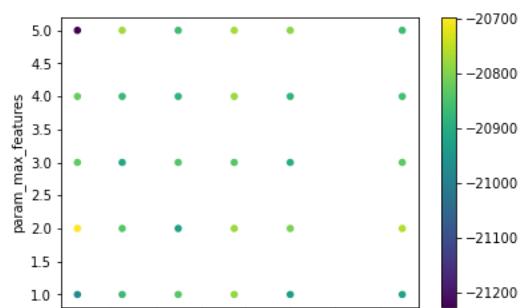
```
6      {'max_features': 2, 'n_estimators': 5}  
Name: params, dtype: object
```

Or, we can visualise the surface generated in the grid:

```

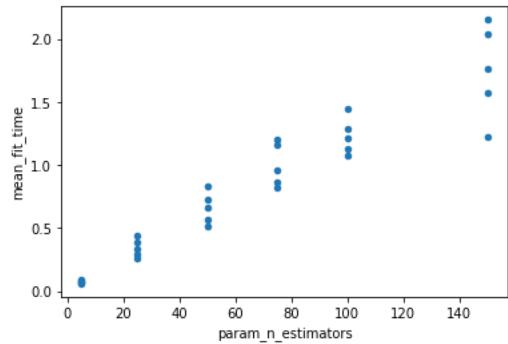
grid_res[["param_n_estimators", "param_max_features"]]\n    .astype(float)\n    .plot.scatter("param_n_estimators",\n                  "param_max_features",\n                  c=grid_res["mean_test_score"],\n                  cmap="viridis")\n);

```



It is also possible to see that, as expected, the more trees, the longer it takes to compute:

```
grid_res[["param_n_estimators", "mean_fit_time"]]\n    .astype(float)\n    .plot.scatter("param_n_estimators",\n                  "mean_fit_time")
```



**EXERCISE** What's the model performance (R/MSE) of the random forest setup we've used with the optimised set of parameters from the experiment above?

More at:

[https://scikit-learn.org/stable/modules/grid\\_search.html](https://scikit-learn.org/stable/modules/grid_search.html)

By Dani Arribas-Bel



Data Science Studio by [Dani Arribas-Bel](#) is licensed under a [Creative Commons Attribution-ShareAlike 4.0 International License](#).