Machine Learning workshop

Prof. Daniele Bonacorsi

ISGC 2021

March 22, 2021

Lab on **Regression**

[<u>credits</u>: A. Geron, "Hands-On Machine Learning With Scikit-Learn and Tensorflow"]

Steps to follow

Steps:

- frame your problem
- select a performance measure
- get the data
- descriptive statistics → discover and visualise the data to gain insights
- data pre-processing → prepare the data for ML algos
- model selection, model training
- model fine-tuning
- solution presentation
- (launch, monitor, maintain your newly deployed system)

The goal

Build a model of housing prices in California using the California census data.

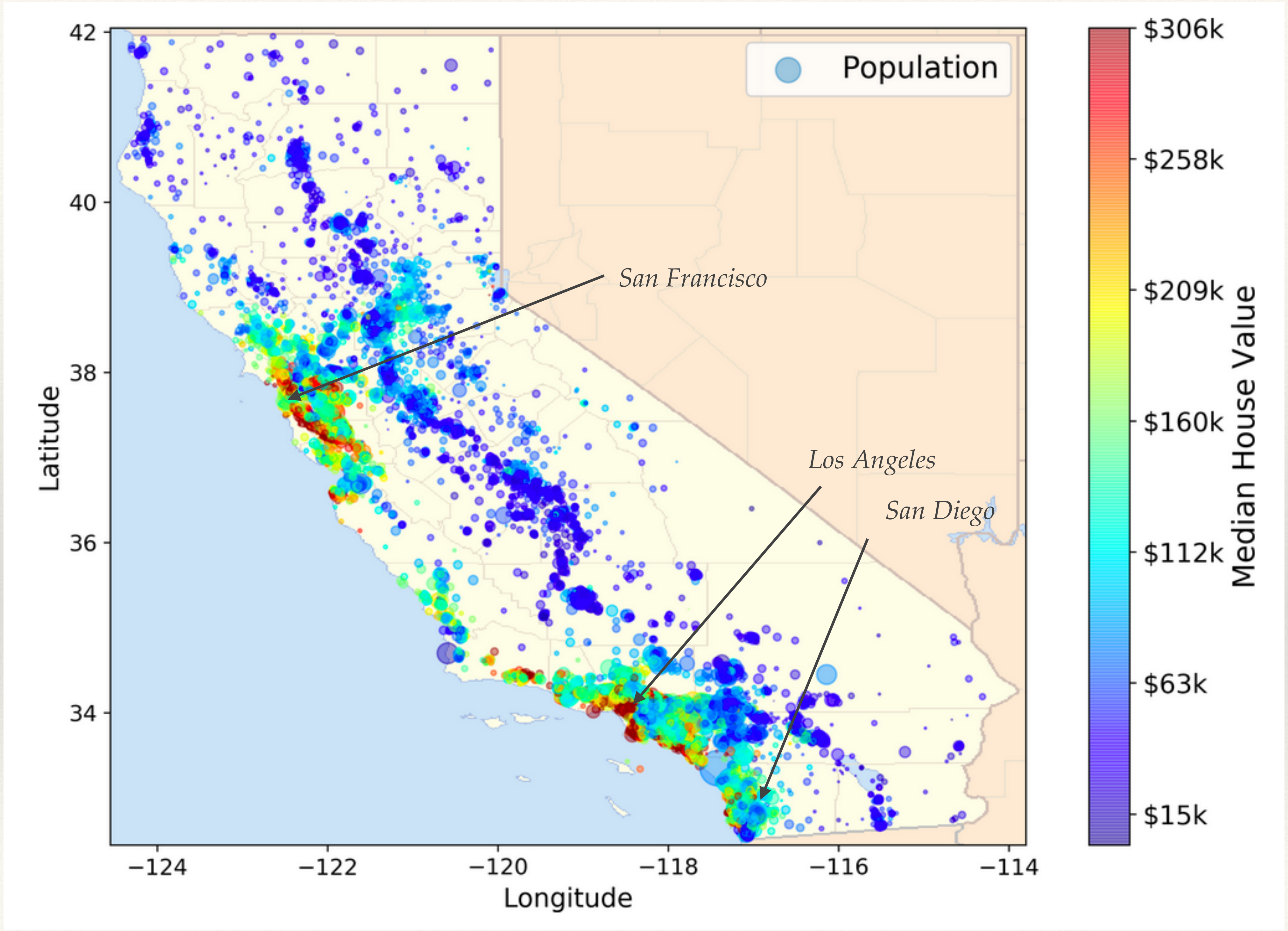
Pretend to be a recently-hired data scientist in a real estate company in California, and you are asked to predict the price of a house given various parameters, having at your disposal the **California Housing Prices dataset**:

- from the StatLib repository: *R. Kelley Pace and Ronald Barry, "Sparse Spatial Autoregressions," Statistics & Probability Letters 33, no. 3 (1997): 291–297*

From this dataset, you know:

- population, median income, median housing price, much more.. for each block group (of 600-3000 people) - called "**districts**"
  - caveats: not updated (data from the 90s) and minor mods (added a categorical attribute, removed a few features for teaching purposes)

Your model should learn from this data and be able to predict the median housing price in any district, given all the other metrics


Population
Median House Value
$306k
$258k
$209k
$160k
$112k
$63k
$15k
San Francisco
Los Angeles
San Diego
Latitude
42
40
38
36
34
Longitude
−124
−122
−120
−118
−116
−114

Frame the problem / **ask questions** - step 1

Building a model is not the goal. **Ask for the goal(s)**. You do not work in an isolated system: stand up and ASK.

Good questions are:

- *"what is my model being used for, eventually?"*
    - this tells you how you concretely organise the **approach** to the problem, what **algorithms** you will select, what **performance** measure you will use to evaluate your model, how much **effort** you should invest in each (sub-)part of the work
- *"what the current status of study of this problem is (if any)?"*
    - this gives you a reference performance, as well as insights on how to solve the problem
- *"what the expected full data pipeline which my solution will insert in?"*
    - Data pipeline as a sequence of data processing components. Very common in ML. Async and self-contained components, data store as the only interface, different teams on different components, tactics for broken components, monitoring, etc

Frame the problem / **make assumptions** - step 2

Is it Supervised, Unsupervised, or Reinforcement Learning? Is it a classification task or a regression task? Should you use batch learning or online learning techniques?

(… think …)

Frame the problem / **make assumptions** - step 2

Is it Supervised, Unsupervised, or Reinforcement Learning? Is it a classification task or a regression task? Should you use batch learning or online learning techniques?

It is a **supervised** learning task..

- you are given labeled training examples: each instance comes with the expected output, i.e. the district's median housing price

Frame the problem / **make assumptions** - step 2

Is it Supervised, Unsupervised, or Reinforcement Learning? Is it a classification task or a regression task? Should you use batch learning or online learning techniques?

It is a **supervised** learning task..

- you are given labeled training examples: each instance comes with the expected output, i.e. the district's median housing price

It is a **univariate regression** task..

- you are asked to predict a value, and a single one per district

Frame the problem / **make assumptions** - step 2

Is it Supervised, Unsupervised, or Reinforcement Learning? Is it a classification task or a regression task? Should you use batch learning or online learning techniques?

It is a **supervised** learning task..

- you are given labeled training examples: each instance comes with the expected output, i.e. the district's median housing price

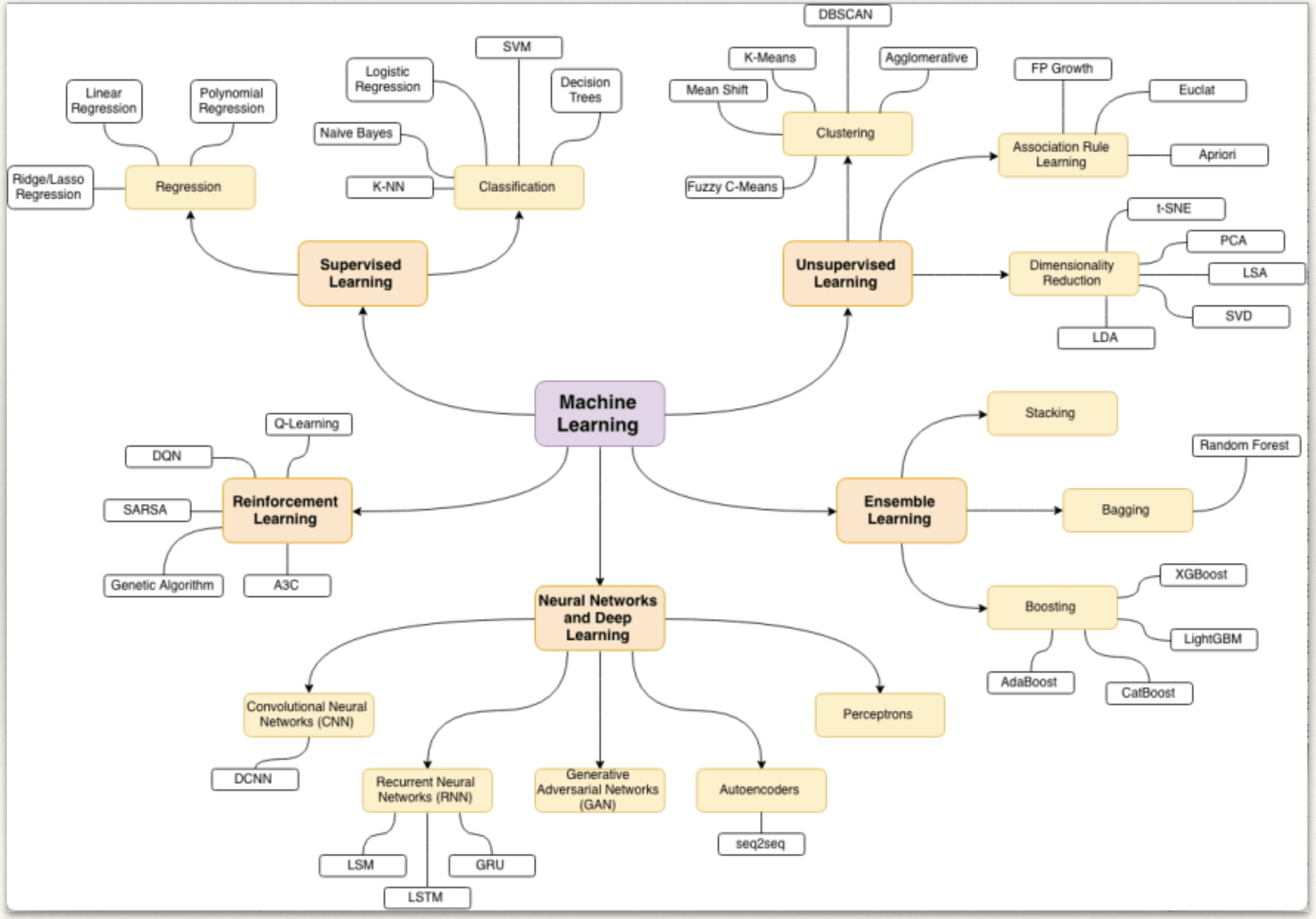
It is a **univariate regression** task..

- you are asked to predict a value, and a single one per district

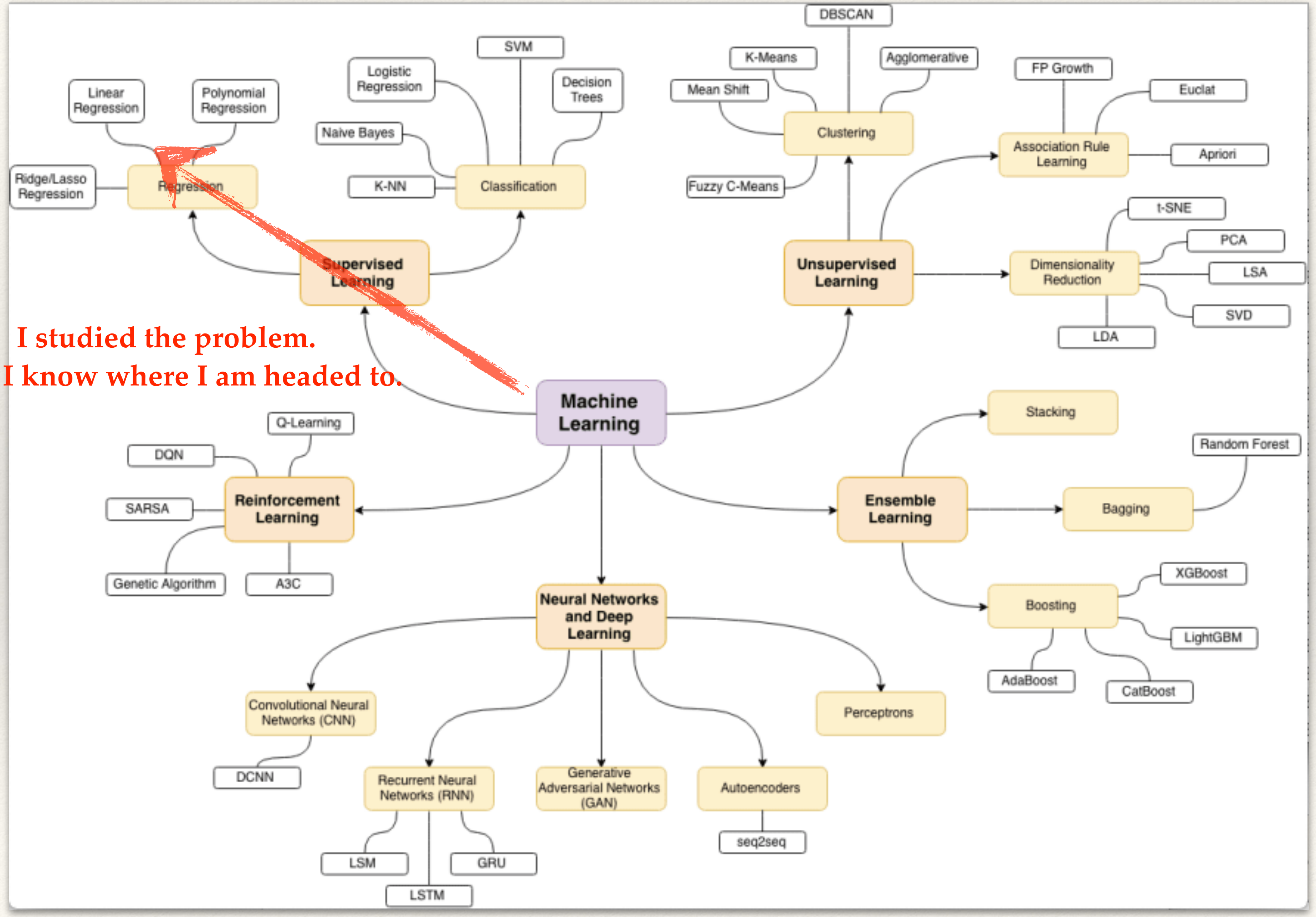
Batch learning techniques should work just fine..

- data is small enough to fit in memory, there is no continuous flow of data coming in the system, there is no particular need to adjust to rapidly changing data

"I have the feeling I did nothing so far…"



“Quite the opposite! you did something already!”



I studied the problem.
Now I know where I am headed to.

Frame the problem / **check assumptions** - step 3

After knowing the full pipeline (from external input).. now recheck your assumptions.

So far: **Supervised. Univariate regression. Batch learning**

- "is any other components in the overall work pipeline making my assumptions unnecessary or tactically wrong?"

Example:

Suppose your value predictions are going to be clustered into coarse-grain categories (e.g. just "cheap", "medium","expensive"). Then, getting the price perfectly right is not important at all, you just need to get the category right, and your task should have been framed as a classification task instead!

Notation

m	# of **instances**, i.e. examples in the training dataset
x	"input" variables, or "**features**" (*) (a vector per example)
y	"output" variable, or "**label**" (a.k.a "target") per example
$(\mathbf{x}^{(i)}, y^{(i)})$	the single ith training example (ith row)

Example:

$$\mathbf{x}^{(1)} = \begin{pmatrix} -118.29 \\ 33.91 \\ 1,416 \\ 38,372 \end{pmatrix}$$

$$y^{(1)} = 156,400$$

(*) strictly speaking, *attribute* != *feature* ..

Select a **performance measure**

A typical performance measure for regression problems is the Root Mean Square Error (**RMSE**)

- it gives an idea of how much error the system typically makes in its predictions: the smaller it is the better

Or the Mean Absolute Error (**MAE**) - aka Average Absolute Deviation

$$\mathrm{RMSE}(\mathbf{X}, h) = \sqrt{\frac{1}{m} \sum_{i=1}^{m} \left(h\left(\mathbf{x}^{(i)}\right) - y^{(i)}\right)^2}$$

$$\mathrm{MAE}(\mathbf{X}, h) = \frac{1}{m} \sum_{i=1}^{m} \left|h\left(\mathbf{x}^{(i)}\right) - y^{(i)}\right|$$

h → my hypothesis for y

Both are ways to measure the distance between two vectors (predictions and labels). Various distance measures, or norms, are possible:

- RMSE → root of sum of squares → Euclidean norm, or ℓ_2 norm, often noted $\| \|_2$
- MAE → sum of absolutes → Manhattan norm, or ℓ_1 norm, often noted $\| \|_1$

In general:

- ℓ_k norm of a vector **v** containing n elements: $\|\mathbf{v}\|_k = (|v_0|^k + |v_1|^k + .. + |v_n|^k)^{1/k}$
- ℓ_0 gives the # non-zero elements; ℓ_∞ gives the max absolute value in the vector

Which one?

- The higher the norm index, the more it focuses on large values and neglects small ones. This is why RMSE is more sensitive to outliers than the MAE (if you have outliers, use MAE; when outliers are exponentially rare (like in a bell-shaped curve), RMSE performs very well and is generally preferred

Time to code!

You should start your code with a **set-up** part.

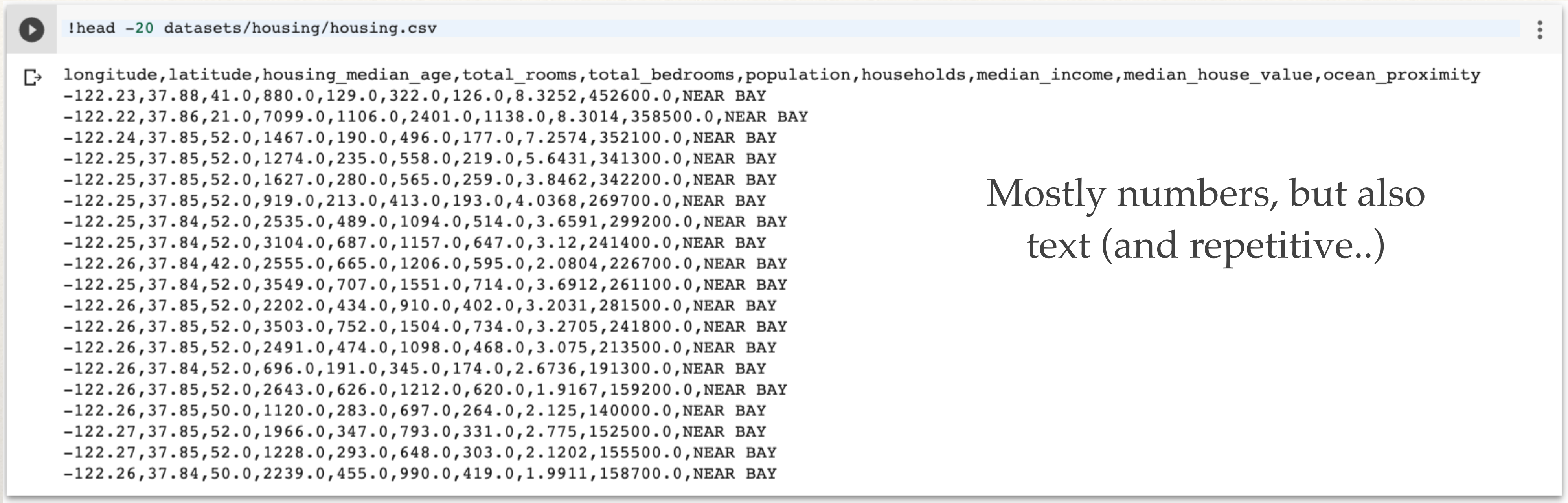
*If you are trying all this real-time during the workshop, this means you should go to **colab***

Then, it is time to **import the data** and check if this worked.

Practice R2 | 1 minute | Time to code!

Inspect the data

Once data source has been downloaded, inspect it straight

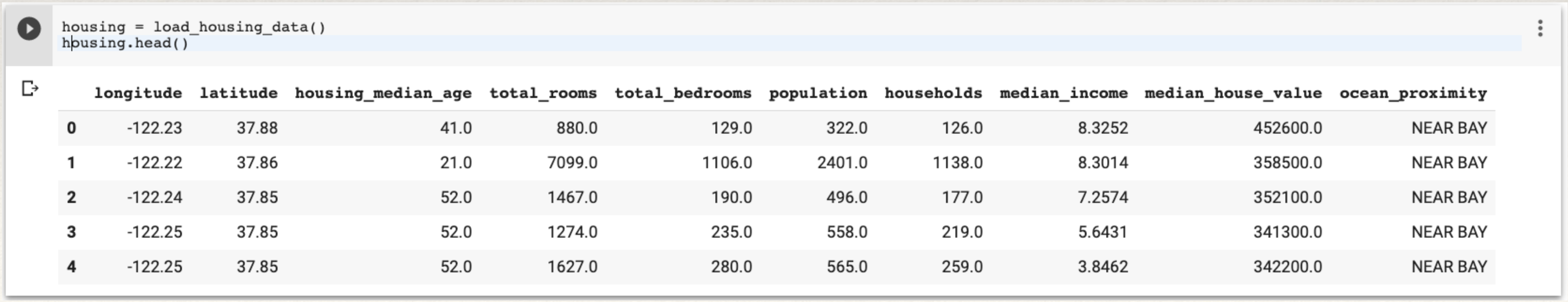


```
!head -20 datasets/housing/housing.csv
```

```
longitude,latitude,housing_median_age,total_rooms,total_bedrooms,population,households,median_income,median_house_value,ocean_proximity
-122.23,37.88,41.0,880.0,129.0,322.0,126.0,8.3252,452600.0,NEAR BAY
-122.22,37.86,21.0,7099.0,1106.0,2401.0,1138.0,8.3014,358500.0,NEAR BAY
-122.24,37.85,52.0,1467.0,190.0,496.0,177.0,7.2574,352100.0,NEAR BAY
-122.25,37.85,52.0,1274.0,235.0,558.0,219.0,5.6431,341300.0,NEAR BAY
-122.25,37.85,52.0,1627.0,280.0,565.0,259.0,3.8462,342200.0,NEAR BAY
-122.25,37.85,52.0,919.0,213.0,413.0,193.0,4.0368,269700.0,NEAR BAY
-122.25,37.84,52.0,2535.0,489.0,1094.0,514.0,3.6591,299200.0,NEAR BAY
-122.25,37.84,52.0,3104.0,687.0,1157.0,647.0,3.12,241400.0,NEAR BAY
-122.26,37.84,42.0,2555.0,665.0,1206.0,595.0,2.0804,226700.0,NEAR BAY
-122.25,37.84,52.0,3549.0,707.0,1551.0,714.0,3.6912,261100.0,NEAR BAY
-122.26,37.85,52.0,2202.0,434.0,910.0,402.0,3.2031,281500.0,NEAR BAY
-122.26,37.85,52.0,3503.0,752.0,1504.0,734.0,3.2705,241800.0,NEAR BAY
-122.26,37.85,52.0,2491.0,474.0,1098.0,468.0,3.075,213500.0,NEAR BAY
-122.26,37.84,52.0,696.0,191.0,345.0,174.0,2.6736,191300.0,NEAR BAY
-122.26,37.85,52.0,2643.0,626.0,1212.0,620.0,1.9167,159200.0,NEAR BAY
-122.26,37.85,50.0,1120.0,283.0,697.0,264.0,2.125,140000.0,NEAR BAY
-122.27,37.85,52.0,1966.0,347.0,793.0,331.0,2.775,152500.0,NEAR BAY
-122.27,37.85,52.0,1228.0,293.0,648.0,303.0,2.1202,155500.0,NEAR BAY
-122.26,37.84,50.0,2239.0,455.0,990.0,419.0,1.9911,158700.0,NEAR BAY
```

Mostly numbers, but also text (and repetitive..)

Or use pandas to deal with a DataFrame object - much easier to view and manipulate:



```
housing = load_housing_data()
housing.head()
```

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	median_income	median_house_value	ocean_proximity
0	-122.23	37.88	41.0	880.0	129.0	322.0	126.0	8.3252	452600.0	NEAR BAY
1	-122.22	37.86	21.0	7099.0	1106.0	2401.0	1138.0	8.3014	358500.0	NEAR BAY
2	-122.24	37.85	52.0	1467.0	190.0	496.0	177.0	7.2574	352100.0	NEAR BAY
3	-122.25	37.85	52.0	1274.0	235.0	558.0	219.0	5.6431	341300.0	NEAR BAY
4	-122.25	37.85	52.0	1627.0	280.0	565.0	259.0	3.8462	342200.0	NEAR BAY

Inspect the data.. further

Inspect the data further:

- use the pandas info() method on your dataframe (tip: it is called "housing")
- use value_counts() to inspect special attributes (tip: "ocean proximity")
- use describe() to get a summary of the numerical attributes
- plot all attributes using **matplotlib**
- review all the output you got, and try to extract interesting observations

Practice R3 | 2 minutes | Time to code!

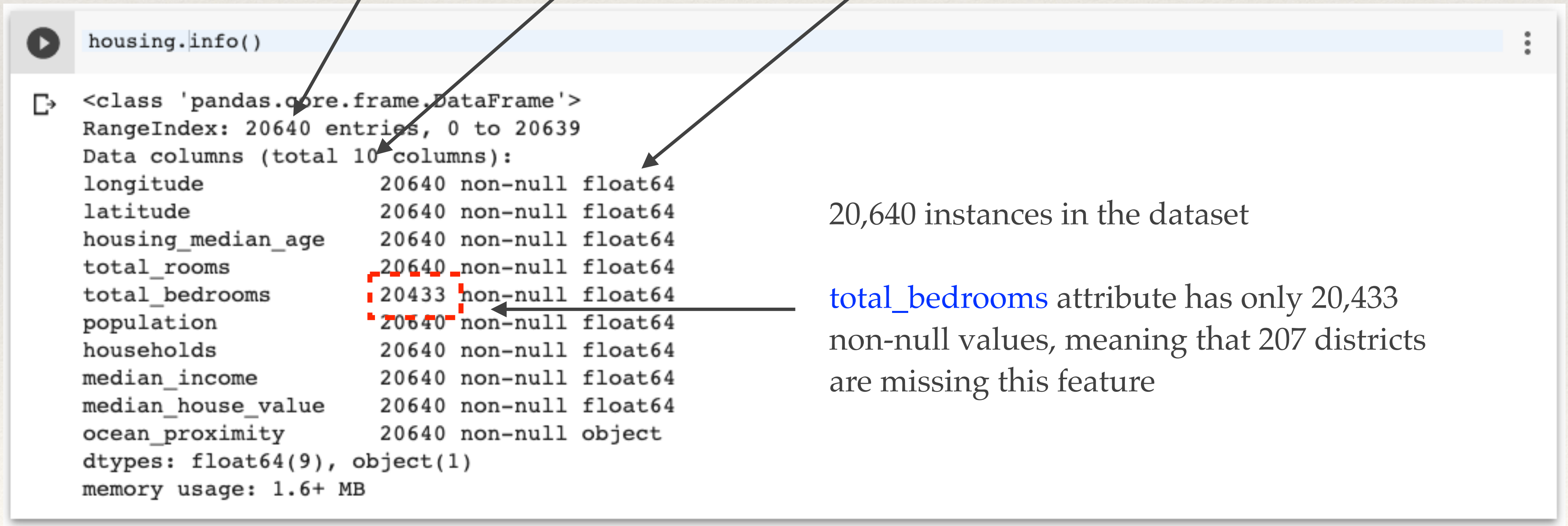
Inspect the data

Each row represents one district. There are 10 attributes (columns):

- *longitude, latitude, housing_median_age, total_rooms, total_bedrooms, population, households, median_income, median_house_value, ocean_proximity*

The pandas info() method is useful to get a quick description of the data

- total number of rows and columns, each attribute's type, # non-null values



```
housing.info()
```

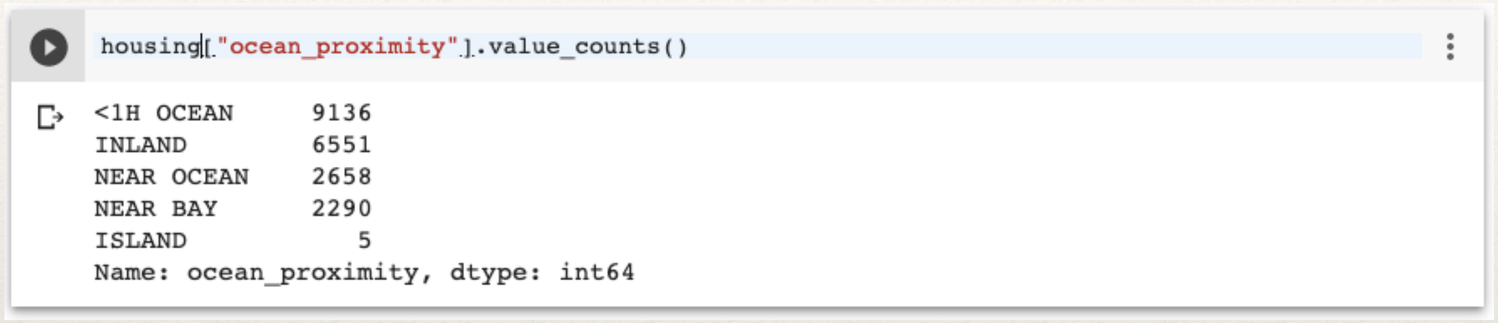
```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 20640 entries, 0 to 20639
Data columns (total 10 columns):
longitude             20640 non-null float64
latitude              20640 non-null float64
housing_median_age    20640 non-null float64
total_rooms           20640 non-null float64
total_bedrooms        20433 non-null float64
population            20640 non-null float64
households            20640 non-null float64
median_income         20640 non-null float64
median_house_value    20640 non-null float64
ocean_proximity       20640 non-null object
dtypes: float64(9), object(1)
memory usage: 1.6+ MB
```

20,640 instances in the dataset

total_bedrooms attribute has only 20,433 non-null values, meaning that 207 districts are missing this feature

Inspect the data

All attributes numerical, except ocean_proximity. Its type is object, so it could hold any kind of py object, but you loaded from a CSV file so you know that it must be a text attribute. Probably categorical: use value_counts()

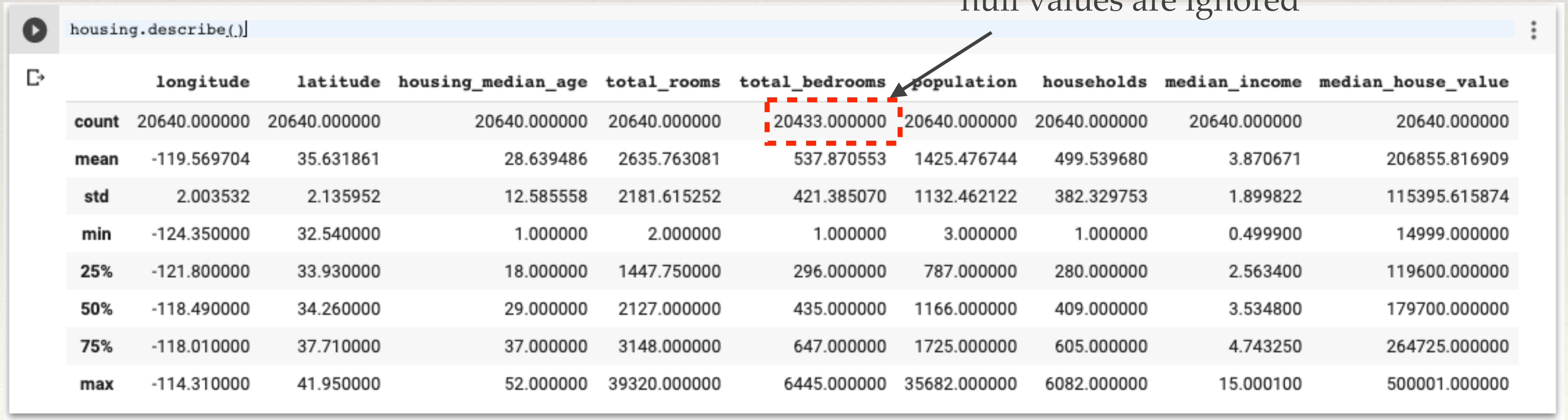


```
housing["ocean_proximity"].value_counts()
```

```
<1H OCEAN     9136
INLAND        6551
NEAR OCEAN    2658
NEAR BAY      2290
ISLAND           5
Name: ocean_proximity, dtype: int64
```

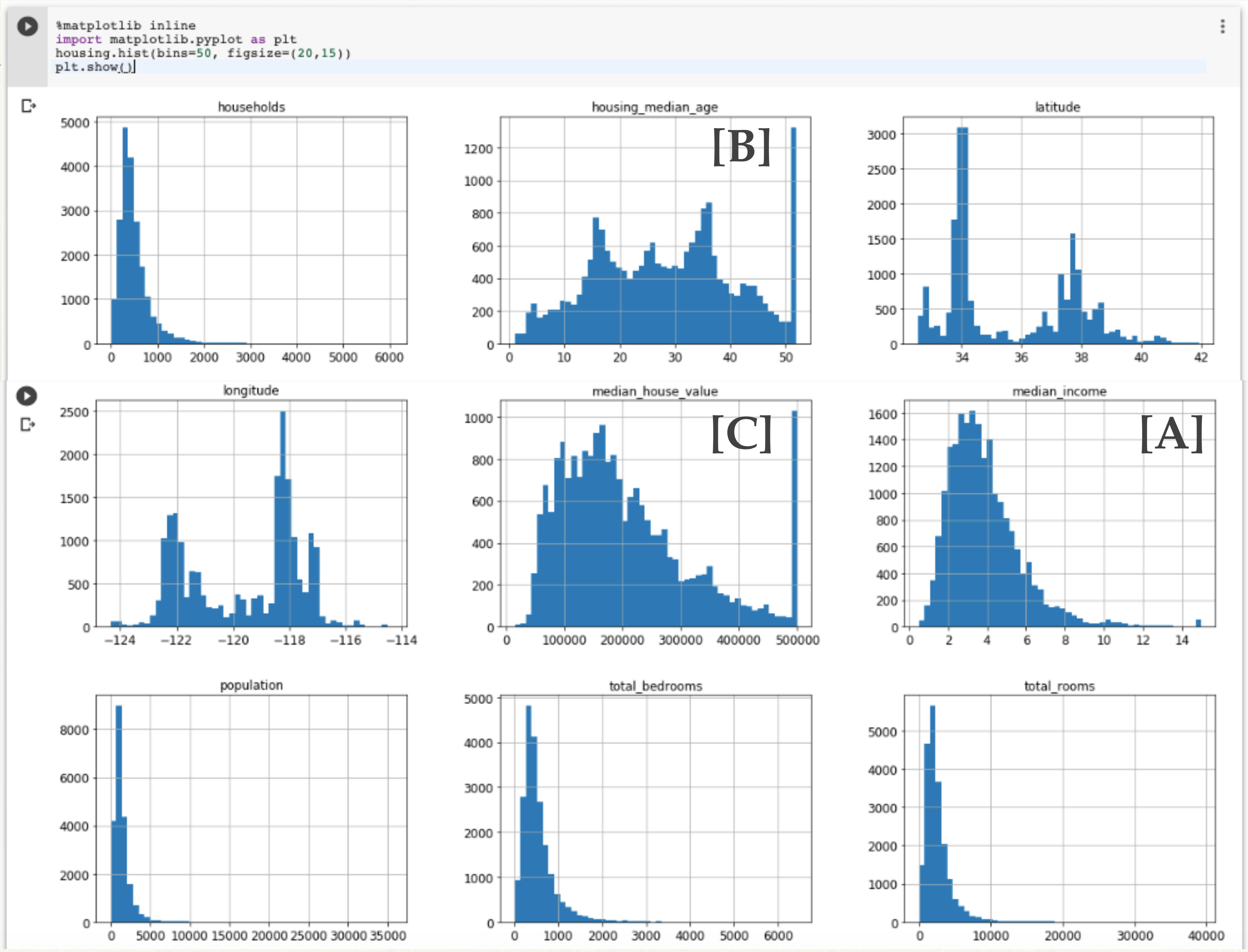
The describe() method shows a summary of the numerical attributes

null values are ignored



```
housing.describe()
```

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	median_income	median_house_value
count	20640.000000	20640.000000	20640.000000	20640.000000	20433.000000	20640.000000	20640.000000	20640.000000	20640.000000
mean	-119.569704	35.631861	28.639486	2635.763081	537.870553	1425.476744	499.539680	3.870671	206855.816909
std	2.003532	2.135952	12.585558	2181.615252	421.385070	1132.462122	382.329753	1.899822	115395.615874
min	-124.350000	32.540000	1.000000	2.000000	1.000000	3.000000	1.000000	0.499900	14999.000000
25%	-121.800000	33.930000	18.000000	1447.750000	296.000000	787.000000	280.000000	2.563400	119600.000000
50%	-118.490000	34.260000	29.000000	2127.000000	435.000000	1166.000000	409.000000	3.534800	179700.000000
75%	-118.010000	37.710000	37.000000	3148.000000	647.000000	1725.000000	605.000000	4.743250	264725.000000
max	-114.310000	41.950000	52.000000	39320.000000	6445.000000	35682.000000	6082.000000	15.000100	500001.000000



```
%matplotlib inline
import matplotlib.pyplot as plt
housing.hist(bins=50, figsize=(20,15))
plt.show()
```

Inspect the data

Notice a few things in these histograms:

- **[A]** attribute not expressed in the **expected units** (USD). Ask who gave you the data..
    - it is in tens of thousands of USD
- **[B] [C]** were **capped**. Note you have a label here. It could be a problem. Your ML algo may learn that prices never go beyond that limit, which is wrong. Check if you need precise predictions also in those ranges. If yes, either you collect proper labels for the districts whose labels were capped, or you remove those examples from the training set (and the test set)
- The attributes (e.g. **[A] [C]**) have very **different scales** → **feature scaling**
- Many histograms are **skewed**: this may make it a bit harder for some ML algos to detect patterns → **transform attributes** to get symmetric distributions
- …

<u>**You are not wasting time!**</u> Some steps here go towards a better understanding (and preprocessing) of the data you are feeding to ML

Train / Test splitting

Split into train and test set:

- in the standard way
- (NOT MANDATORY) using stratification

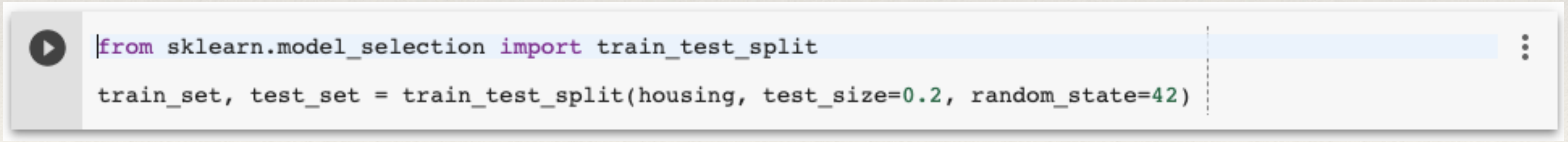
Practice R4 | 3 minutes | Time to code!

Create a test set

Take a subset of your data and put it aside.

Why? Because your brain is an amazing pattern detection system and you should avoid it to trick you!

- i.e. your brain is highly prone to overfitting. Looking at the test set, you might see some patterns and be biassed towards some ML model. Then, when you estimate the generalisation error using the test set, your estimate will be too optimistic and you will launch a system that will eventually perform on new data much worse than expected → "**data snooping bias**"



```
from sklearn.model_selection import train_test_split
train_set, test_set = train_test_split(housing, test_size=0.2, random_state=42)
```

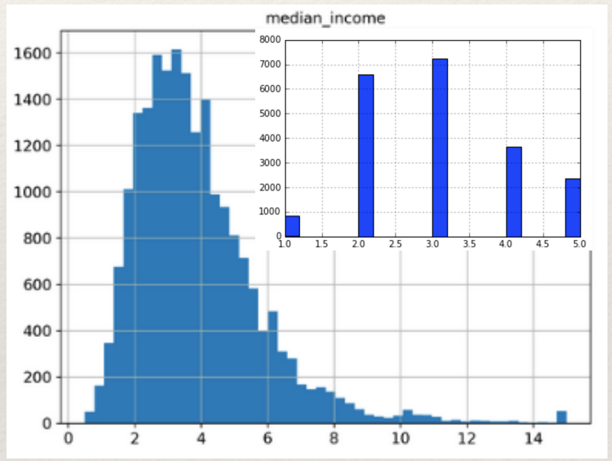
>90% of ML practitioners do this.. but..

Sampling: purely random vs stratified

Purely random sampling works OK only if the dataset is large enough. If not, you risk to introduce a significant sampling bias

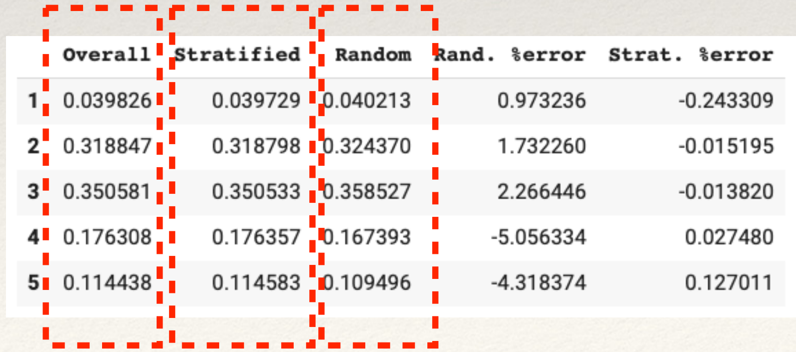
Best is to use **stratified sampling**

- as the population is divided into homogeneous subgroups called **strata**, sample **the right number of instances from each stratum** to guarantee that the test set you build is **representative of the overall population**



E.g. if you are told that median income is an important attribute to predict house prices in a district, make sure you represent all categories of salary in your test set

- create e.g. 5 categories, assign examples to each
- then, use the sklearn's StratifiedShuffleSplit class
- compare the income category proportions in the overall dataset, in the test set generated with stratified sampling, and in a test set generated using purely random sampling



	Overall	Stratified	Random	Rand. %error	Strat. %error
1	0.039826	0.039729	0.040213	0.973236	-0.243309
2	0.318847	0.318798	0.324370	1.732260	-0.015195
3	0.350581	0.350533	0.358527	2.266446	-0.013820
4	0.176308	0.176357	0.167393	-5.056334	0.027480
5	0.114438	0.114583	0.109496	-4.318374	0.127011

Visualize the data

Visualize to gain insight:

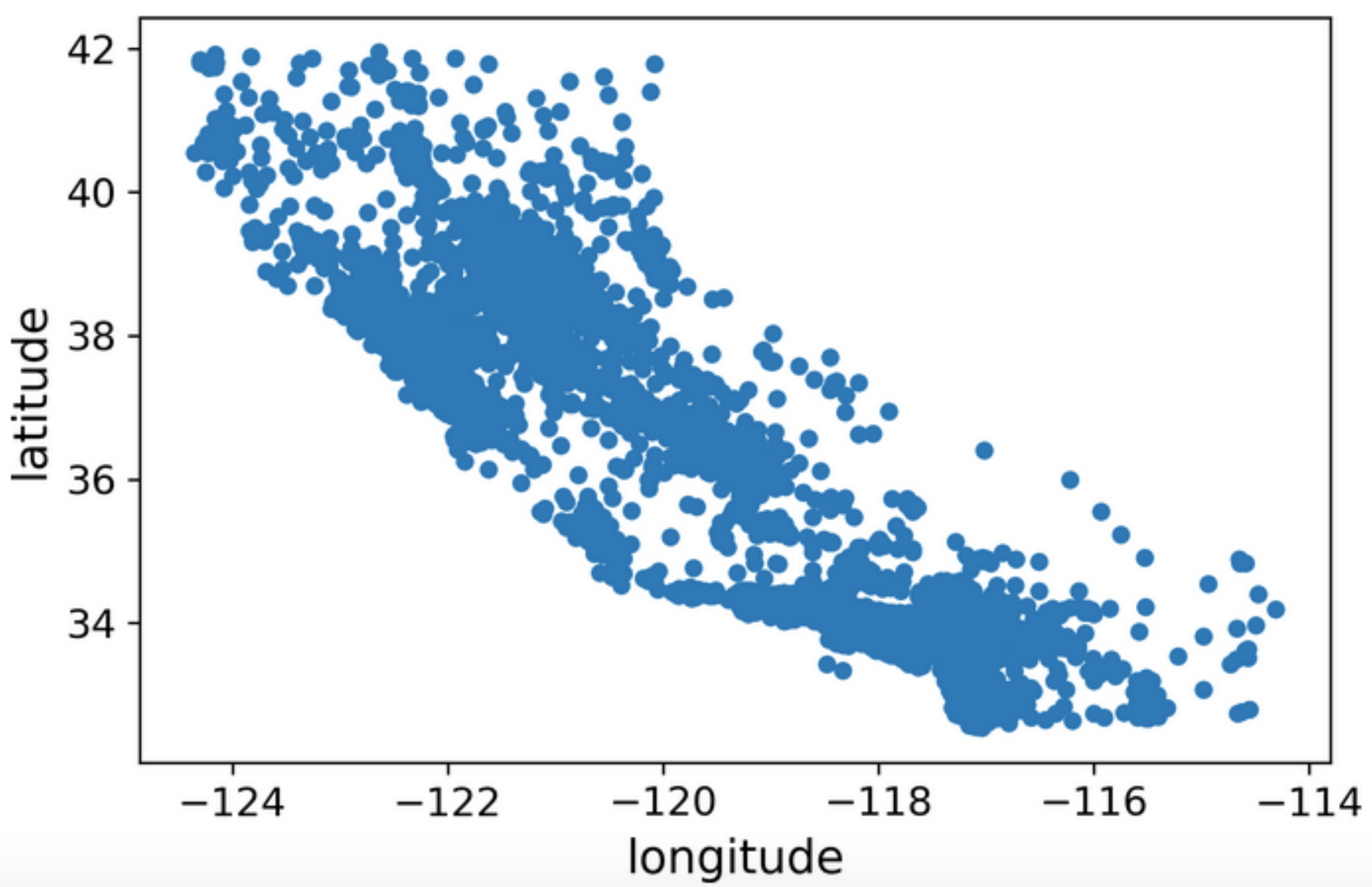
- hint: use the previous dataframe, and .plot() with proper options
  - e.g. scatter kind, and x and y should be the GPS coordinates

Practice R5 | 3 minutes | Time to code!

Visualising Geographical Data

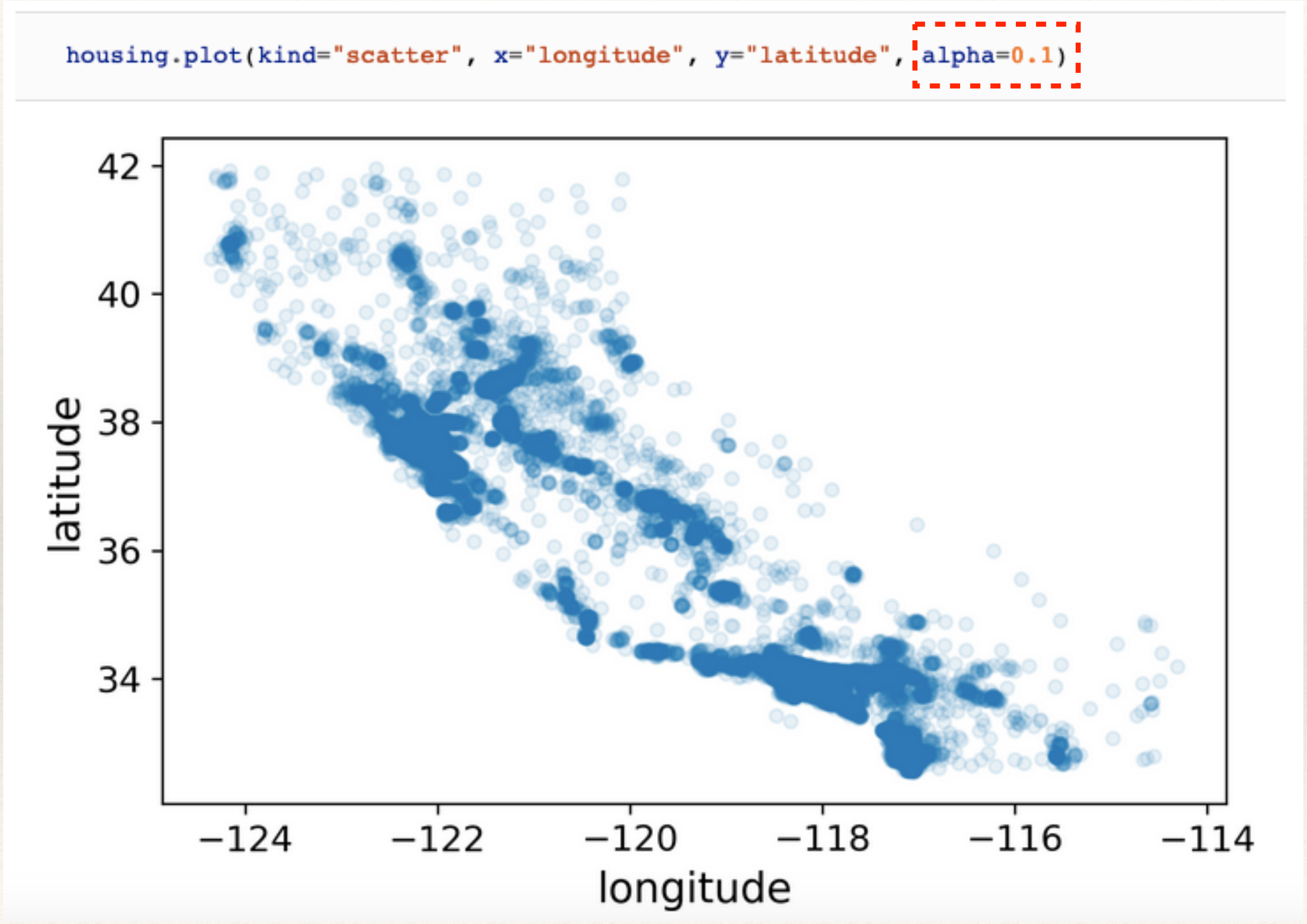
For geolocated data, this often gives useful insights.

```
housing.plot(kind="scatter", x="longitude", y="latitude")
```



Visualising Geographical Data

Use **matplotlib** features to highlight high density patterns.

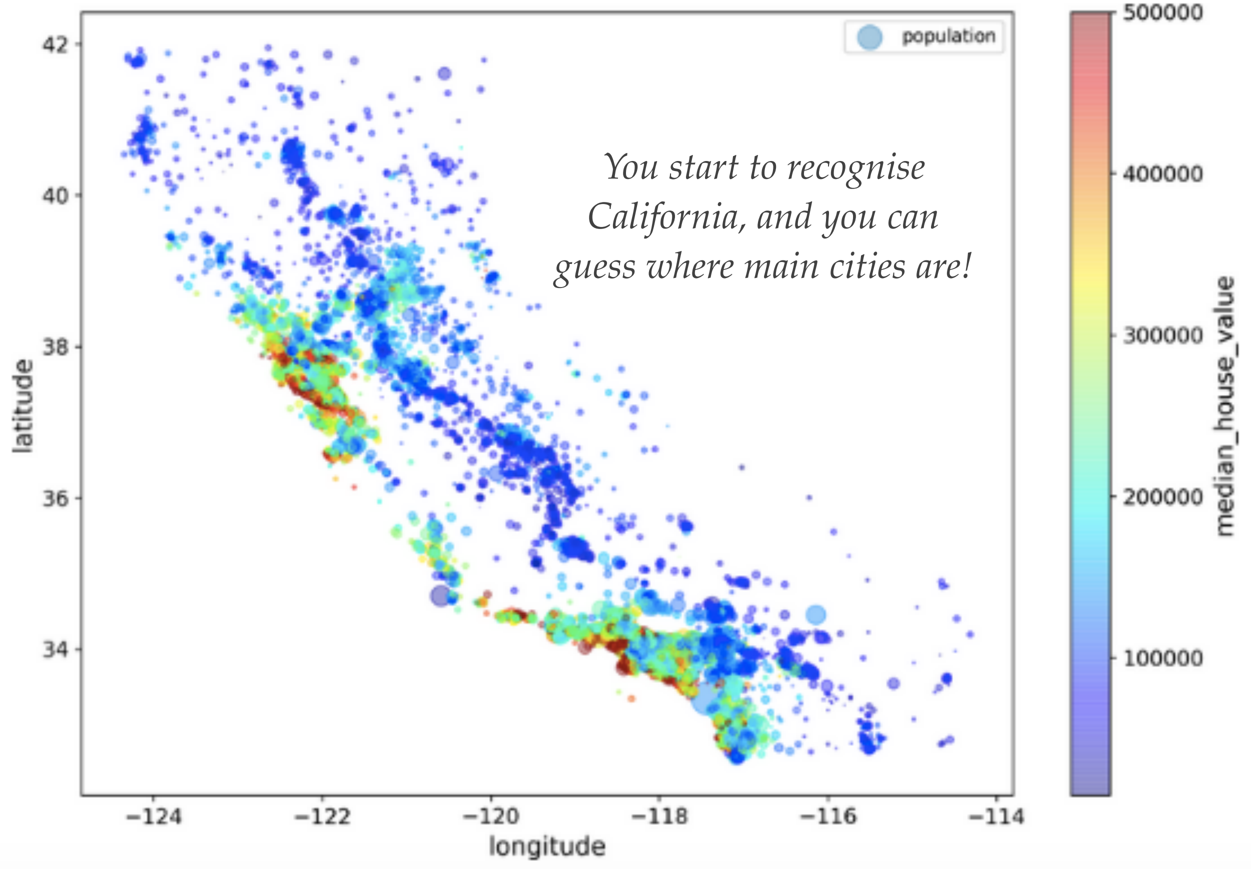


```
housing.plot(kind="scatter", x="longitude", y="latitude", alpha=0.1)
```

Visualising Geographical Data

Display population by circles' size and house price by colour on a **heatmap**.

```
housing.plot(kind="scatter", x="longitude", y="latitude", alpha=0.4,
    s=housing["population"]/100, label="population", figsize=(10,7),
    c="median_house_value", cmap=plt.get_cmap("jet"), colorbar=True,
)
plt.legend()
```



You start to recognise California, and you can guess where main cities are!

Housing prices are very much related to location (e.g. close to the ocean) and to the population density.

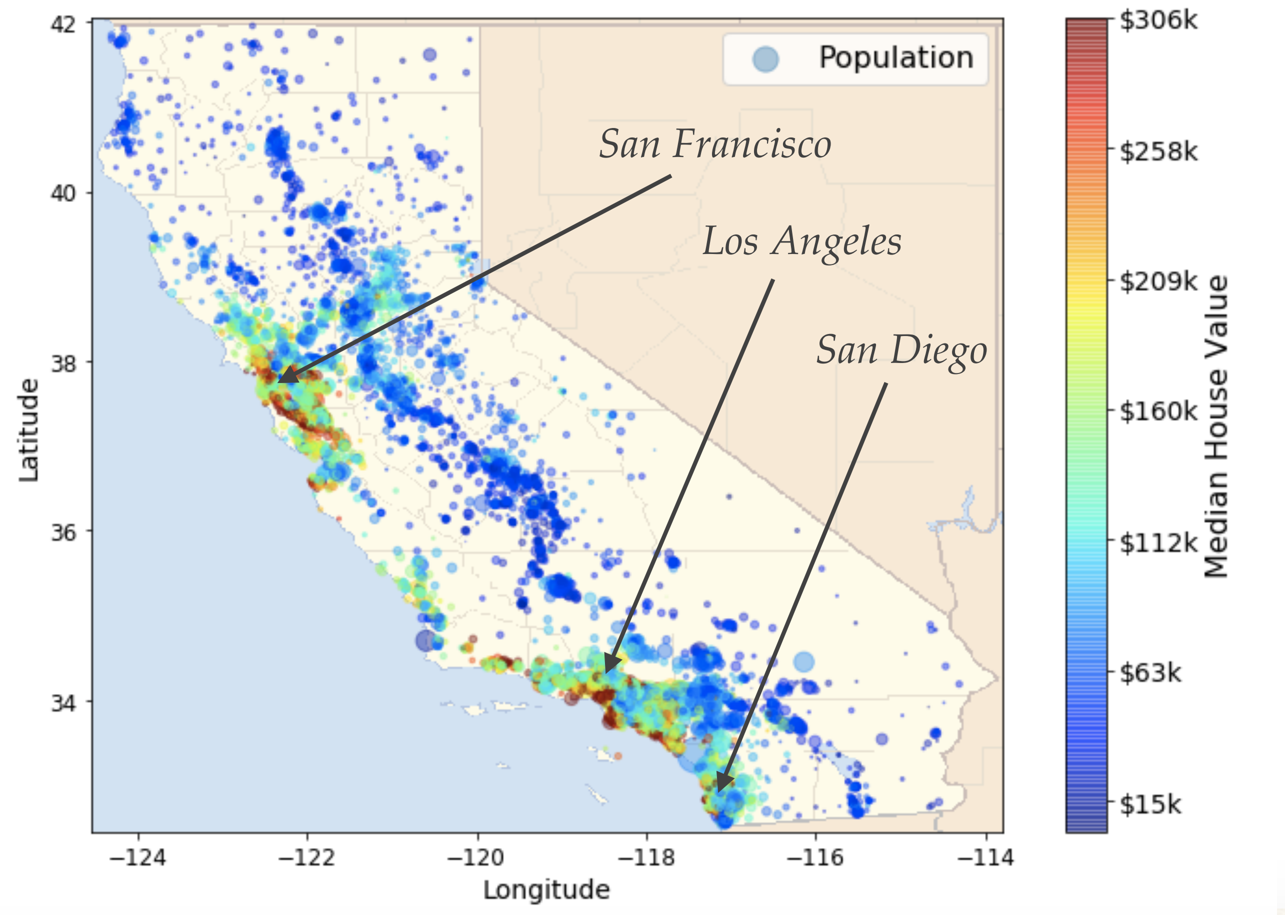
Use a clustering algo to detect the main clusters?

Careful about ocean proximity attribute: perhaps useful but works different North vs South, so not a simple rule..

Visualising Geographical Data

Display population by circles' size and house price by colour on a **heatmap**.

```
housing.plot(kind="scatter", x="longitude", y="latitude", alpha=0.4,
    s=housing["population"]/100, label="population", figsize=(10,7),
    c="median_house_value", cmap=plt.get_cmap("jet"), colorbar=True,
)
plt.legend()
```



Housing prices are very much related to location (e.g. close to the ocean) and to the population density.

Use a clustering algo to detect the main clusters?

Careful about ocean proximity attribute: perhaps useful but works different North vs South, so not a simple rule..

Correlations

Inspect correlations

- hint: start with the housing dataframe, and explore .corr()

Code available also to do it visually, afterwards

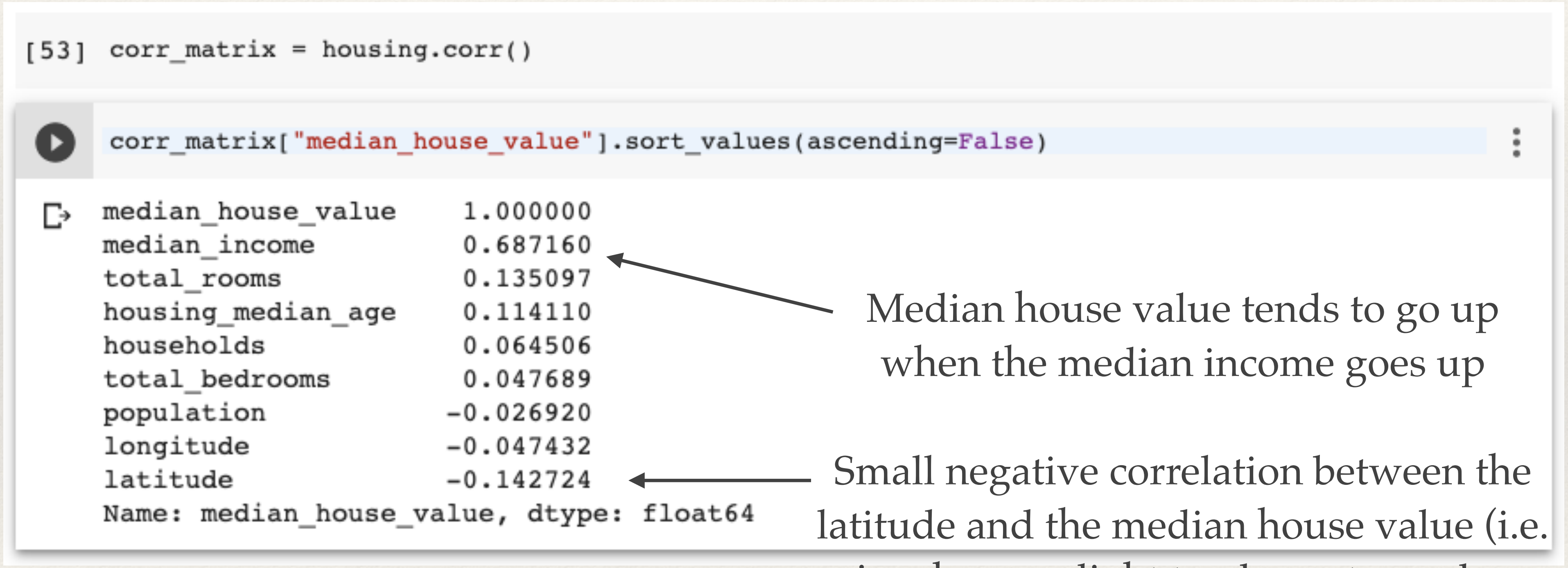
- here, try to run the code, create the plots and examine them

Practice R6 | 3 minutes | Time to code!

Looking for **correlations** (numerically)

Dataset not huge → compute the **standard correlation coefficient** (aka **Pearson's r**) between every pair of attributes

- e.g. check how much each attribute correlates with the median house value



```
[53] corr_matrix = housing.corr()
```

```
corr_matrix["median_house_value"].sort_values(ascending=False)
```

```
median_house_value    1.000000
median_income         0.687160
total_rooms           0.135097
housing_median_age    0.114110
households            0.064506
total_bedrooms        0.047689
population           -0.026920
longitude            -0.047432
latitude             -0.142724
Name: median_house_value, dtype: float64
```

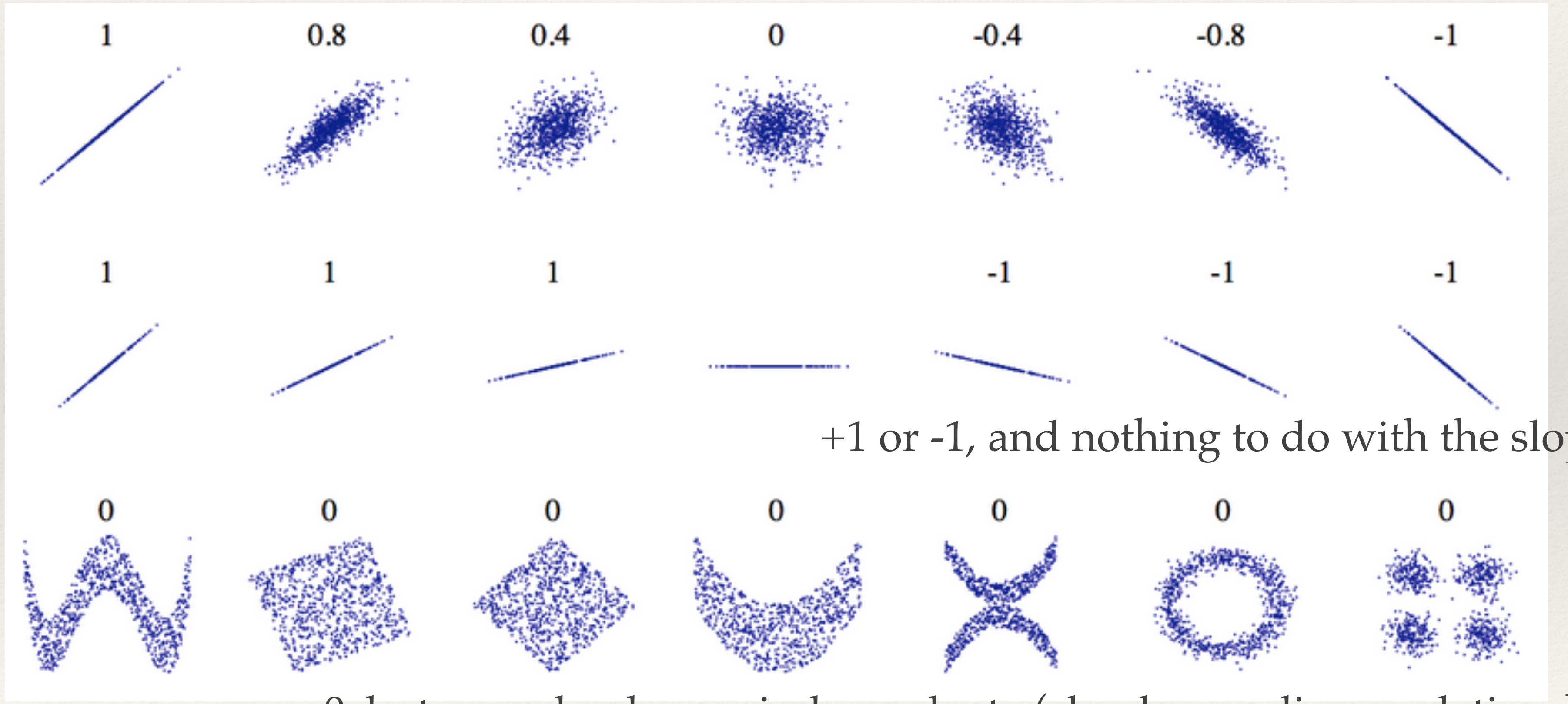
Median house value tends to go up when the median income goes up

Small negative correlation between the latitude and the median house value (i.e. prices have a slight tendency to go down when you go north)

Standard correlation coefficient of various datasets

The correlation coefficient only measures linear correlations

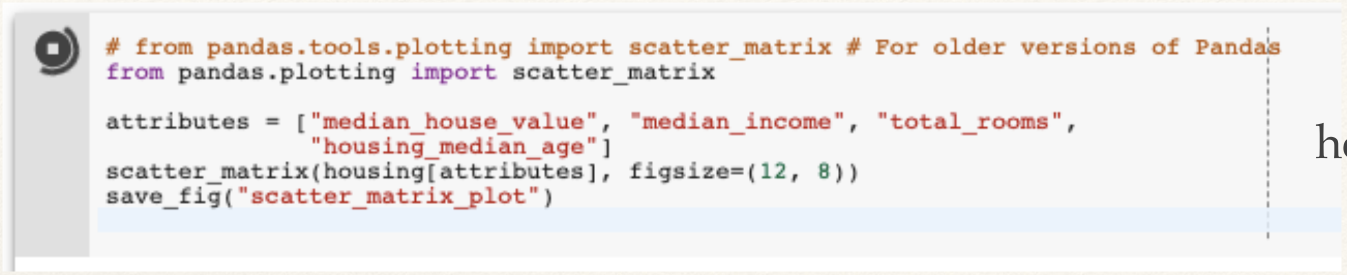
- it may completely miss out on nonlinear relationships (e.g., "if x is close to zero then y generally goes up").



+1 or -1, and nothing to do with the slope

0, but axes clearly non independent.. (clearly, non-linear relationship)

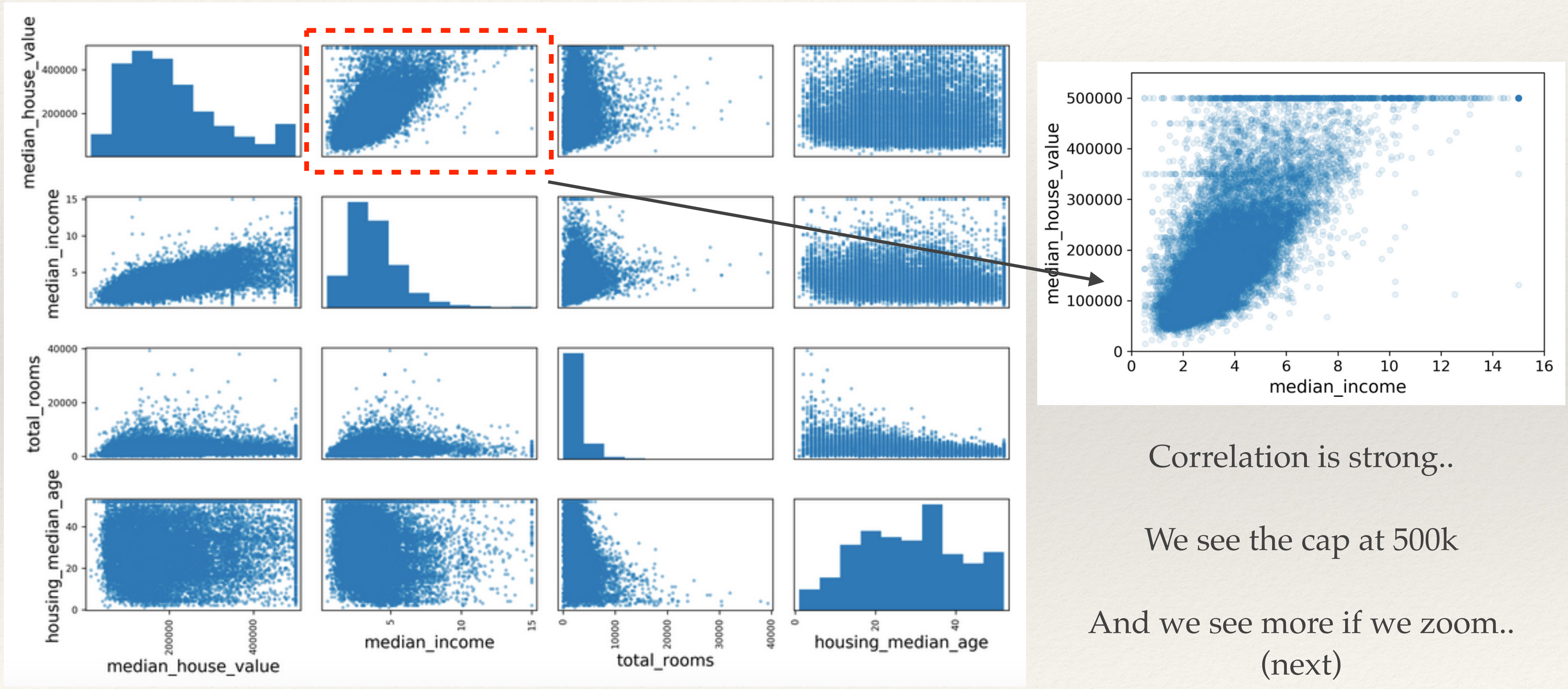
Looking for **correlations** (visually)



```
# from pandas.tools.plotting import scatter_matrix # For older versions of Pandas
from pandas.plotting import scatter_matrix

attributes = ["median_house_value", "median_income", "total_rooms",
              "housing_median_age"]
scatter_matrix(housing[attributes], figsize=(12, 8))
save_fig("scatter_matrix_plot")
```

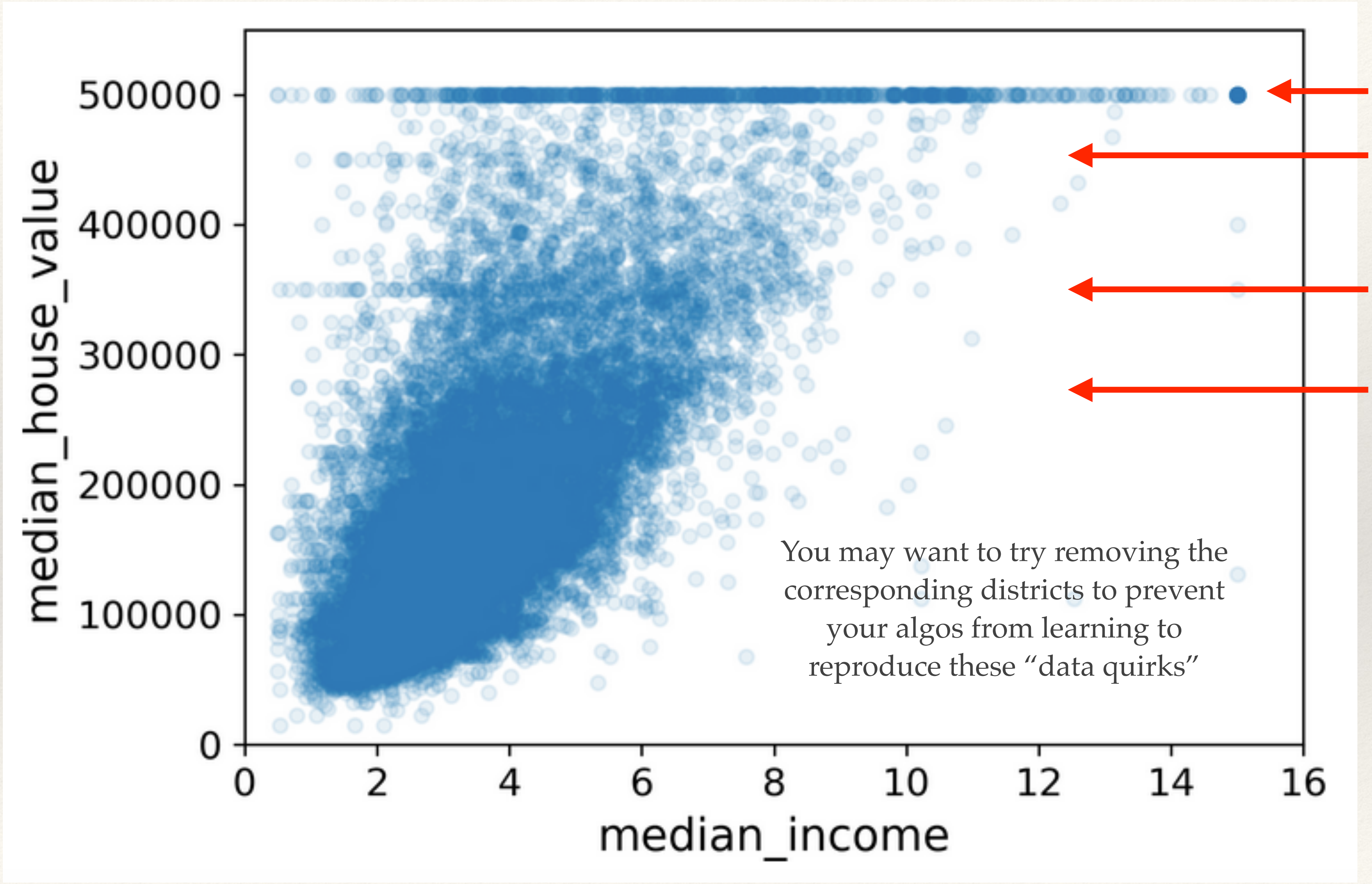
11 attributes → 11^2 plots,
here focussing on just 4 → 16 plots



Correlation is strong..

We see the cap at 500k

And we see more if we zoom..
(next)



You may want to try removing the corresponding districts to prevent your algos from learning to reproduce these “data quirks”

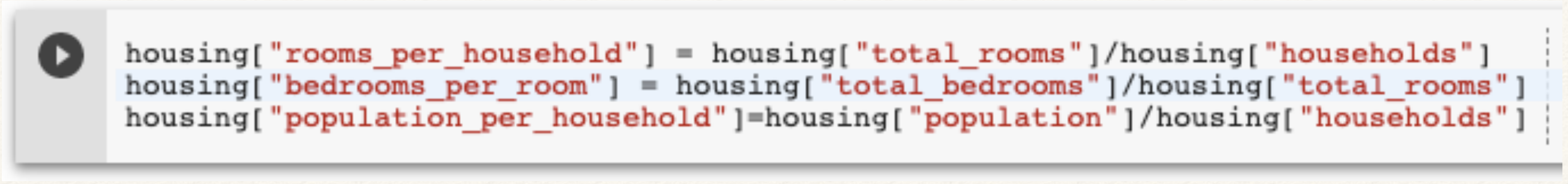
Explore attribute combinations

You need to clean

- here you can follow on slides and code

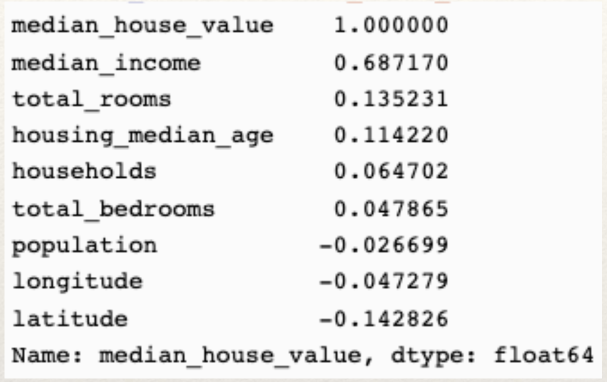
Practice R7 | 5 minutes | Time to code!

Explore attribute combinations



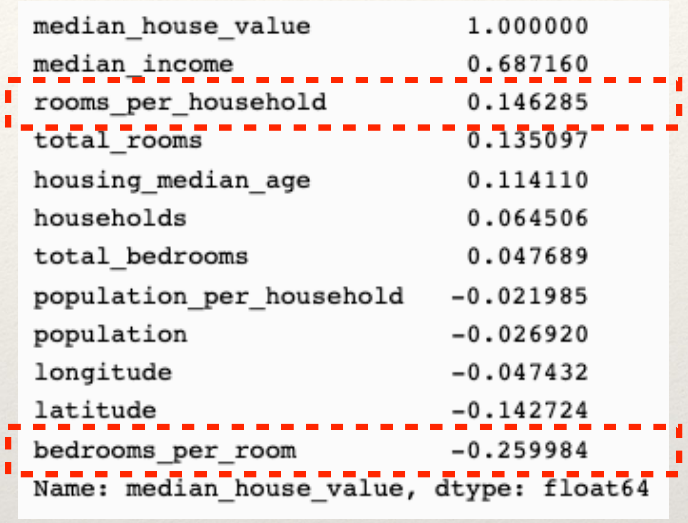
```
housing["rooms_per_household"] = housing["total_rooms"]/housing["households"]
housing["bedrooms_per_room"] = housing["total_bedrooms"]/housing["total_rooms"]
housing["population_per_household"]=housing["population"]/housing["households"]
```

Before:



```
median_house_value    1.000000
median_income         0.687170
total_rooms           0.135231
housing_median_age    0.114220
households            0.064702
total_bedrooms        0.047865
population           -0.026699
longitude            -0.047279
latitude             -0.142826
Name: median_house_value, dtype: float64
```

After:



```
median_house_value          1.000000
median_income               0.687160
rooms_per_household         0.146285
total_rooms                 0.135097
housing_median_age          0.114110
households                  0.064506
total_bedrooms              0.047689
population_per_household   -0.021985
population                 -0.026920
longitude                  -0.047432
latitude                   -0.142724
bedrooms_per_room          -0.259984
Name: median_house_value, dtype: float64
```

Create and add more meaningful attributes. And re-check correlation matrix.

Not bad:

- one of the new variables (bedrooms per room) is more anti-correlated to median house value than other old variables like the total number of rooms or bedrooms. Apparently houses with a lower bedroom/room ratio tend to be more expensive.
- The number of rooms per household is also more informative than the total number of rooms in a district - obviously the larger the houses, the more expensive they are.

Data preparation for ML algos: **missing features**

You need to prepare the data for ML algos, and one action is to attack the issues of attributes that miss the corresponding feature

- here you can follow on slides and code

Practice R8 | 7 minutes | Time to code!

Data preparation for ML algos: **missing features**

Deal with **missing features**

- remember total_bedrooms, that had missing entries?

3 options:

1. Get rid of the corresponding districts → drop rows
2. Get rid of the whole attribute → drop a single column
3. Set the values to some value (zero, the mean, the median, etc.) → save it as you will need it for the test set too, or if/when the system goes live to replace new missing values

Data prep for ML: **Handling Text and Categorical Attributes**

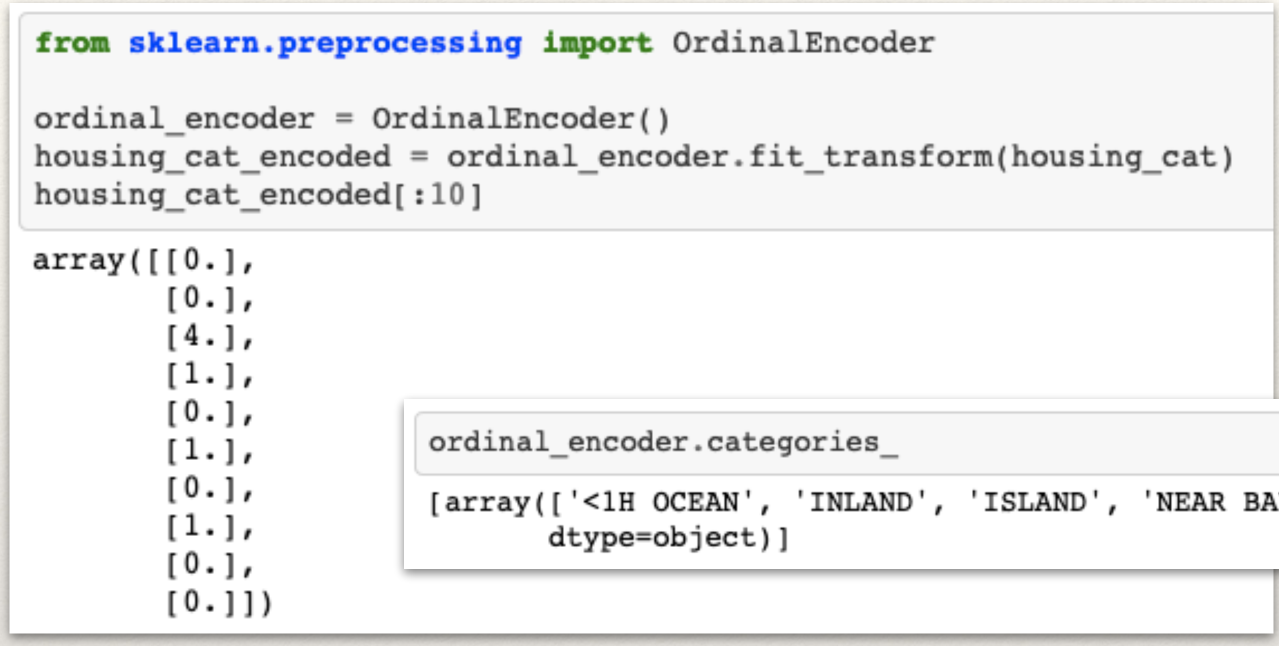
You need to prepare the data for ML algos, and another action is to **attack categorical attributes**

- here you can follow on slides and code

Practice R9 | 5 minutes | Time to code!

Data prep for ML: **Handling Text and Categorical Attributes**

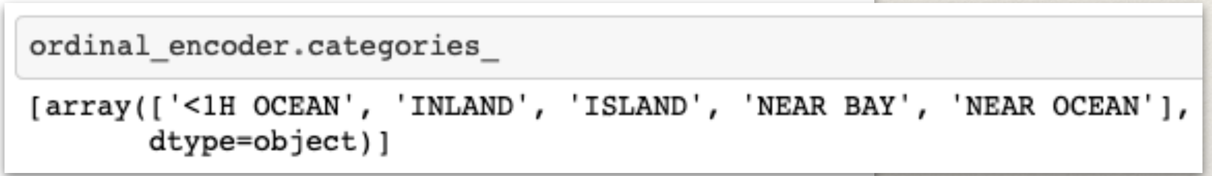
Need to convert categories from text to numbers.



```
from sklearn.preprocessing import OrdinalEncoder

ordinal_encoder = OrdinalEncoder()
housing_cat_encoded = ordinal_encoder.fit_transform(housing_cat)
housing_cat_encoded[:10]
```

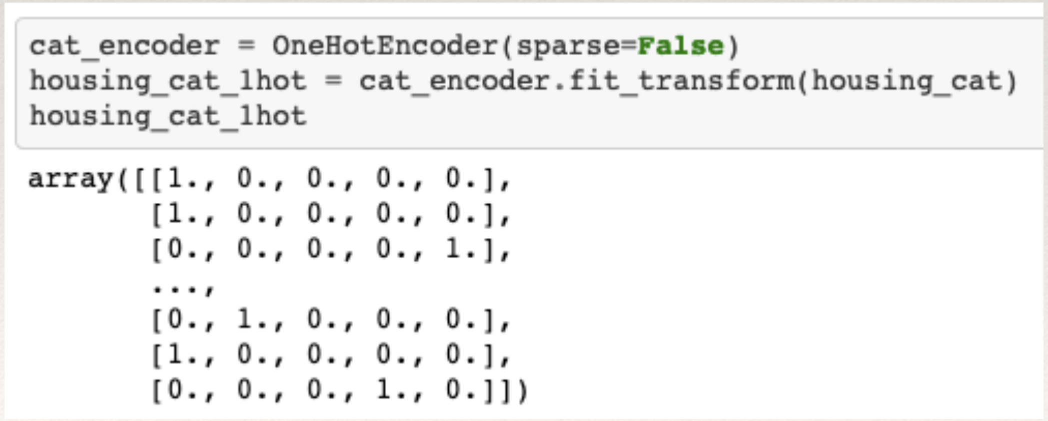
```
array([[0.],
       [0.],
       [4.],
       [1.],
       [0.],
       [1.],
       [0.],
       [1.],
       [0.],
       [0.]])
```



```
ordinal_encoder.categories_
```

```
[array(['<1H OCEAN', 'INLAND', 'ISLAND', 'NEAR BAY', 'NEAR OCEAN'],
       dtype=object)]
```

This representation has the issue that ML algos will assume that 2 nearby values are more similar than 2 distant values.



```
cat_encoder = OneHotEncoder(sparse=False)
housing_cat_1hot = cat_encoder.fit_transform(housing_cat)
housing_cat_1hot
```

```
array([[1., 0., 0., 0., 0.],
       [1., 0., 0., 0., 0.],
       [0., 0., 0., 0., 1.],
       ...,
       [0., 1., 0., 0., 0.],
       [1., 0., 0., 0., 0.],
       [0., 0., 0., 1., 0.]])
```

A common solution is to create one binary attribute per category. This is called **one-hot encoding**, because only one attribute will be equal to 1 (hot), while the others will be 0 (cold).

Data preparation for ML algos: **Feature scaling**

Additional code available also to **add new attributes**, make a **pipeline**, **transformations**, etc..

- check the code (it is abundant!)

Practice R10 | 5-10 minutes | Time to code!

Data preparation for ML algos: **Feature scaling**

ML algos don’t perform well when the input numerical attributes have very different scales

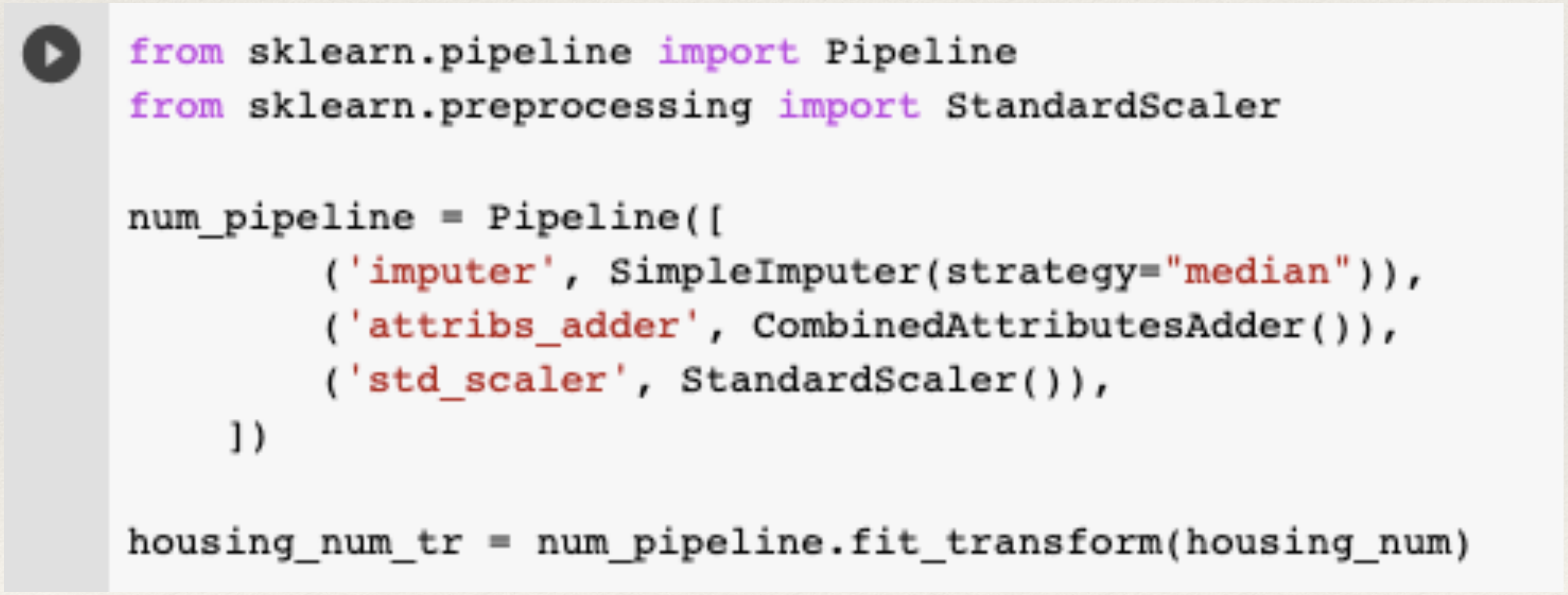
There are 2 common ways to get all attributes to have the same scale:

- **min-max scaling** (aka **normalisation**): values are shifted and rescaled so that they end up all ranging from 0 to 1 (or any other similar range)
  - useful e.g. for NN
- **standardisation**: it subtracts the mean value (so standardised values always have a zero mean), and then it divides by the standard deviation so that the resulting distribution has unit variance
  - much less affected by outliers w.r.t. normalisation

They can be used altogether.

Data preparation for ML algos: **Transformation pipelines**

Many data transformation steps that need to be executed in sequence and in the right order: sklearn provides the **Pipeline** class to help with such sequences of transformations.



```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler

num_pipeline = Pipeline([
        ('imputer', SimpleImputer(strategy="median")),
        ('attribs_adder', CombinedAttributesAdder()),
        ('std_scaler', StandardScaler()),
    ])

housing_num_tr = num_pipeline.fit_transform(housing_num)
```

Select and Train a Model

Here we are!

Practice **R11** | 10 minutes | Time to code!

Select and Train a Model

Here we are!

- I framed the problem
- I got the data and explored it
- I sampled a training set and a test set
- I wrote transformation pipelines to clean up and prepare data for ML

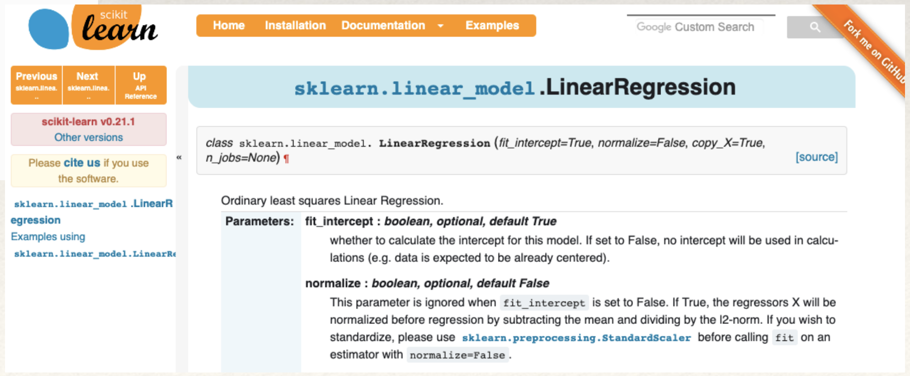
Now, data is ready, and next is to select and train a ML model

- start simple: a linear model

```
from sklearn.linear_model import LinearRegression

lin_reg = LinearRegression()
lin_reg.fit(housing_prepared, housing_labels)
```

Done! It is that simple!



sklearn.linear_model.LinearRegression

class `sklearn.linear_model.` **`LinearRegression`** (*fit_intercept=True, normalize=False, copy_X=True, n_jobs=None*) ¶ [source]

Ordinary least squares Linear Regression.

Parameters: **fit_intercept : *boolean, optional, default True***

whether to calculate the intercept for this model. If set to False, no intercept will be used in calculations (e.g. data is expected to be already centered).

normalize : *boolean, optional, default False*

This parameter is ignored when `fit_intercept` is set to False. If True, the regressors X will be normalized before regression by subtracting the mean and dividing by the l2-norm. If you wish to standardize, please use `sklearn.preprocessing.StandardScaler` before calling `fit` on an estimator with `normalize=False`.

I hope this gave you a feeling as of how much time to spend on

input data preprocessing

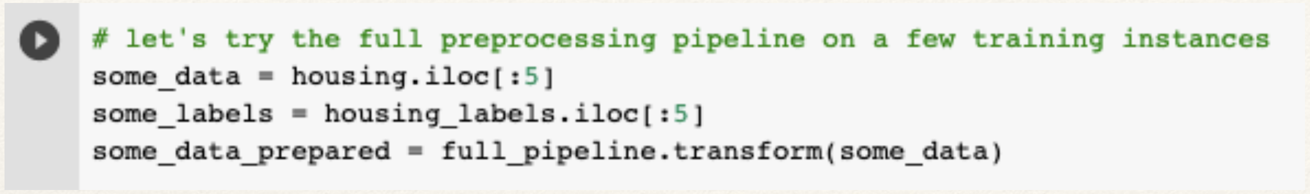
w.r.t

model selection

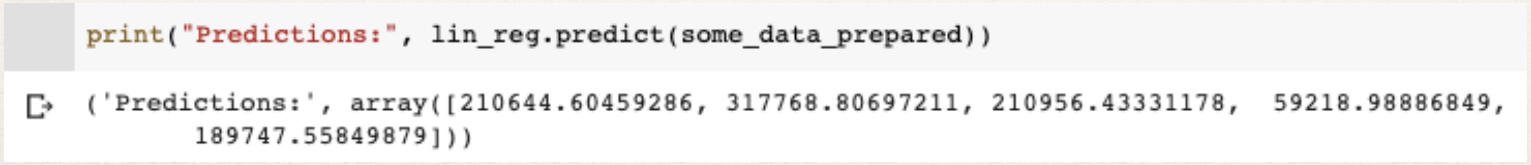
"90% of my time in careful data preparation for ML" is probably a decent guess..

Select and Train a Model

How is the model working? Let' s try some predictions!

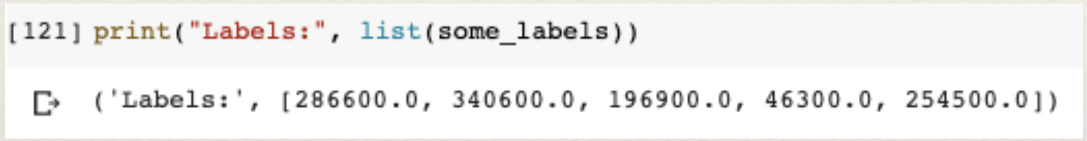


```
# let's try the full preprocessing pipeline on a few training instances
some_data = housing.iloc[:5]
some_labels = housing_labels.iloc[:5]
some_data_prepared = full_pipeline.transform(some_data)
```



```
print("Predictions:", lin_reg.predict(some_data_prepared))
```

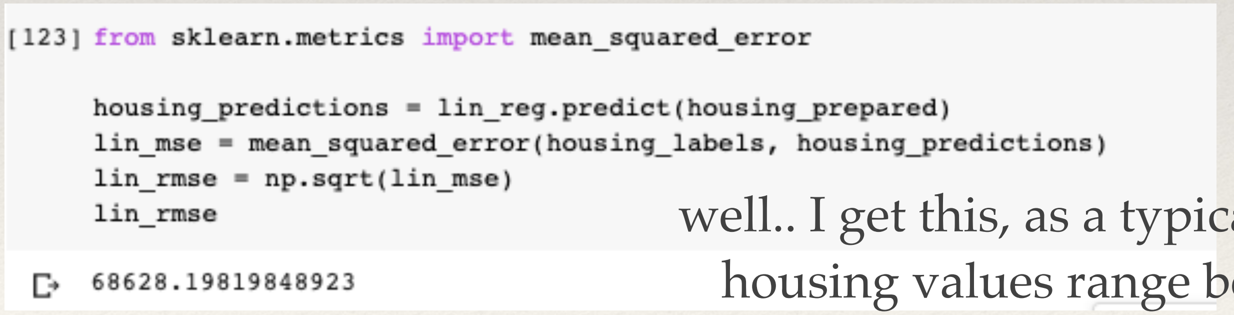
```
('Predictions:', array([210644.60459286, 317768.80697211, 210956.43331178,  59218.98886849,
        189747.55849879]))
```



```
[121] print("Labels:", list(some_labels))
```

```
('Labels:', [286600.0, 340600.0, 196900.0, 46300.0, 254500.0])
```

It works, although the predictions are not exactly accurate (e.g. the first is off by close to 40%!). Measure the RSME on the whole training set:



```
[123] from sklearn.metrics import mean_squared_error

      housing_predictions = lin_reg.predict(housing_prepared)
      lin_mse = mean_squared_error(housing_labels, housing_predictions)
      lin_rmse = np.sqrt(lin_mse)
      lin_rmse
```

```
68628.19819848923
```

well.. I get this, as a typical prediction error, when median housing values range between $120,000 and $265,000..

Clearly not a great score (underfitting) but it is a start! (and it was relatively quick!)

Select and Train a Model

This point is one of the most tricky you will encounter in project on real-world datasets

- i.e. "I need to make a choice. What do I try next?"
- data science is general is done via experienced trials.. no recipes written in stones on most aspects.. And all may largely be dependant on your dataset..

Common practices help you, though. This case:

- **symptoms of underfitting**: this is already A LOT to drive your next choice!
- main ways to fix underfitting are:
  - select a more powerful model
  - feed the training algorithm with better features
  - reduce the constraints on the model

What would you choose?

Select and Train a Model

This point is one of the most tricky you will encounter in project on real-world datasets

- i.e. "I need to make a choice. What do I try next?"
- data science is general is done via experienced trials.. no recipes written in stones on most aspects.. And all may largely be dependant on your dataset..

Common practices help you, though. This case:

- **symptoms of underfitting**: this is already A LOT to drive your next hoice!
- main ways to fix underfitting are:
  - select a more powerful model → you can try, and much cheaper
  - feed the training algorithm with better features → you can try, but expensive..
  - reduce the constraints on the model → non regularized, so this is ruled out

Try another model

Are you able to try a DecisionTreeRegressor yourself?

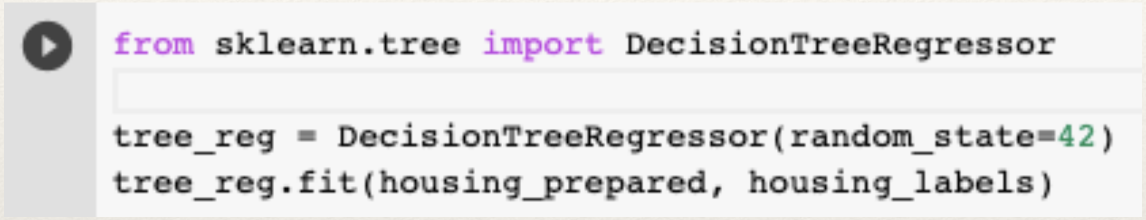
- HINT: sklearn documentation, StackOverflow, …

Practice R12 | 10 minutes | Time to code!

Select and Train a Model

Try a decision tree.

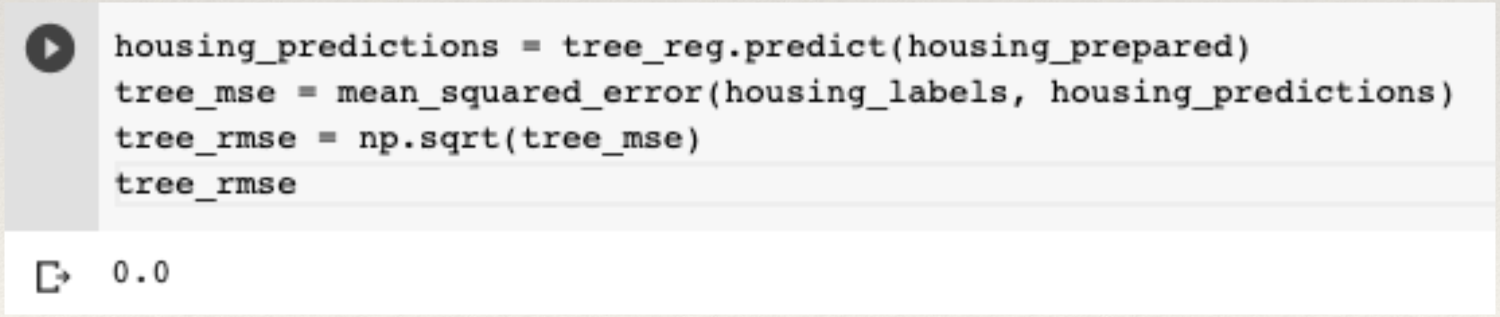
- Because it is a powerful model, capable of finding complex nonlinear relationships in the data



```
from sklearn.tree import DecisionTreeRegressor

tree_reg = DecisionTreeRegressor(random_state=42)
tree_reg.fit(housing_prepared, housing_labels)
```

Now that the model is trained, evaluate it on the training set:

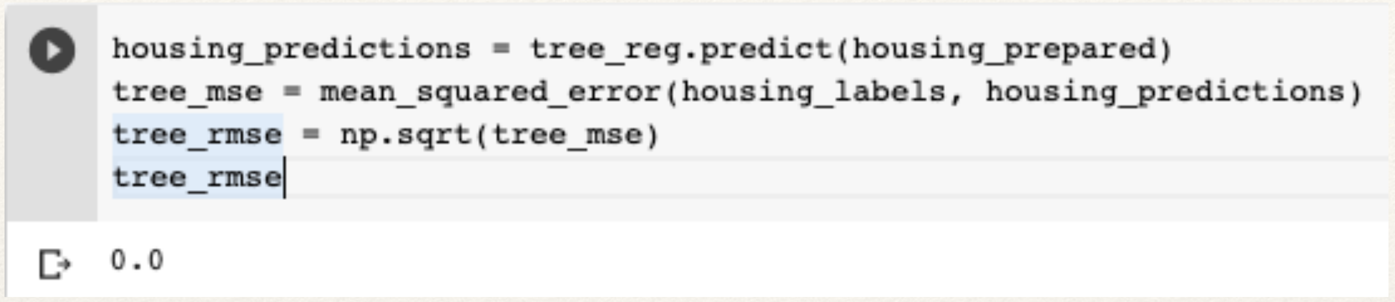


```
housing_predictions = tree_reg.predict(housing_prepared)
tree_mse = mean_squared_error(housing_labels, housing_predictions)
tree_rmse = np.sqrt(tree_mse)
tree_rmse
```

```
0.0
```

WHAT?!? (see next)

Select and Train a Model



```
housing_predictions = tree_reg.predict(housing_prepared)
tree_mse = mean_squared_error(housing_labels, housing_predictions)
tree_rmse = np.sqrt(tree_mse)
tree_rmse
```

```
0.0
```

Wait, **what!?** No prediction error AT ALL!? Is it PERFECT!?

- much more likely that the model has badly overfit the data.. I need to be sure though.. How?

I can't touch the **test set** (until I am ready to test a final model I am decently confident about, and see how it may perform in production). So I need to only **use a sub-part of the training set for training**, and **use the remaining data for.. <u>model validation</u>**

Cross-validation

Check how it is done in the code.

Practice R13 | 2 minutes | Time to code!

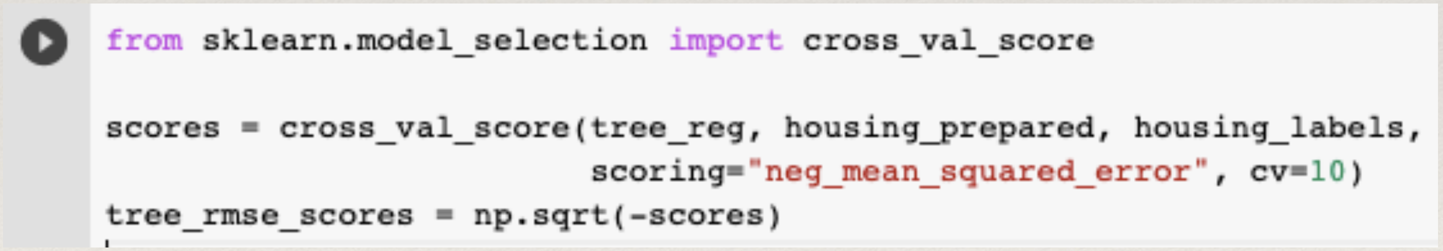
Model Evaluation w/ **k-fold Cross-Validation**

You can (statically):

- split the original training set into train and validation sets, train your model on the (smaller) training sub-set and evaluate it against the validation sub-set

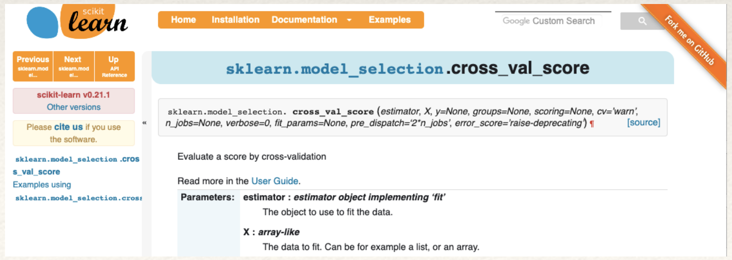
Or (more dynamically)…

- randomly split the training set into k distinct subsets called "**folds**"
  - or think of strata if you think it is the case..
- permute and pick k-1 fields for training and evaluate on the remaining 1, i.e. train and evaluate your model **k times**
- the result is an array containing the **k evaluation scores** (you will average..)



```python
from sklearn.model_selection import cross_val_score

scores = cross_val_score(tree_reg, housing_prepared, housing_labels,
                         scoring="neg_mean_squared_error", cv=10)
tree_rmse_scores = np.sqrt(-scores)
```



sklearn.model_selection.cross_val_score

`sklearn.model_selection.` **`cross_val_score`** (*estimator, X, y=None, groups=None, scoring=None, cv='warn', n_jobs=None, verbose=0, fit_params=None, pre_dispatch='2*n_jobs', error_score='raise-deprecating'*) ¶ [source]

Evaluate a score by cross-validation

Read more in the User Guide.

Parameters: **estimator :** ***estimator object implementing 'fit'***

The object to use to fit the data.

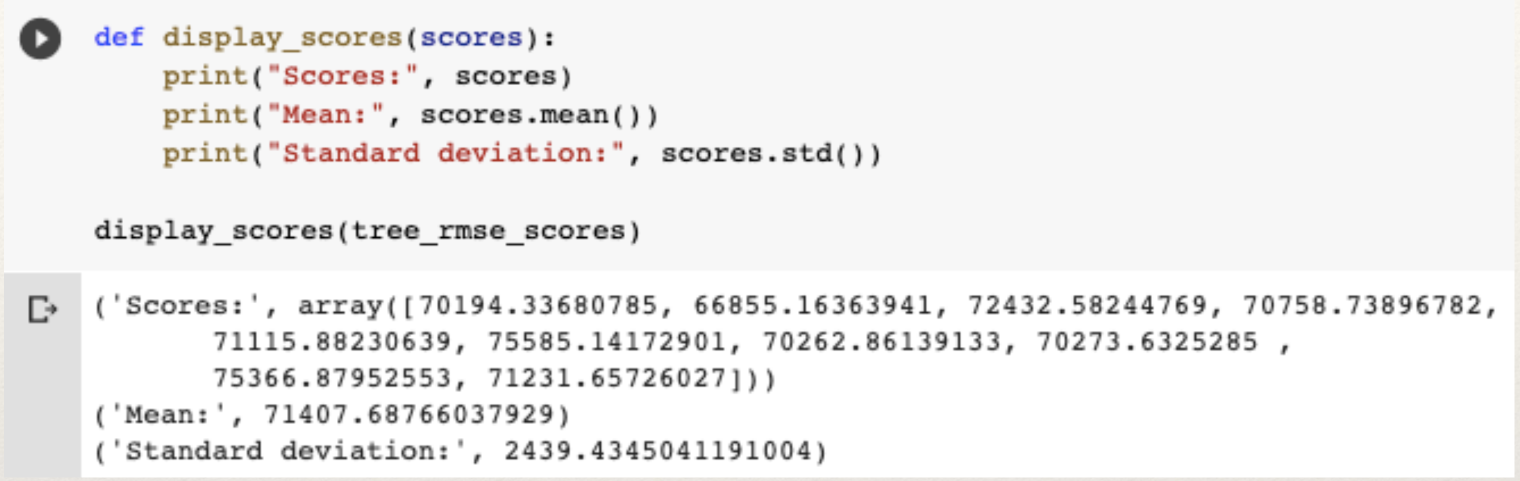
X : ***array-like***

The data to fit. Can be for example a list, or an array.

Practical note: sklearn CV approach expects a *utility* function (greater is better) rather than a *cost* function (lower is better), so the scoring function is actually the opposite of the MSE (i.e. a negative value), which is why the code above computes -scores before calculating the square root.

Model Evaluation w/ **k-fold Cross-Validation**

Lets look at results:



```
def display_scores(scores):
    print("Scores:", scores)
    print("Mean:", scores.mean())
    print("Standard deviation:", scores.std())

display_scores(tree_rmse_scores)
```

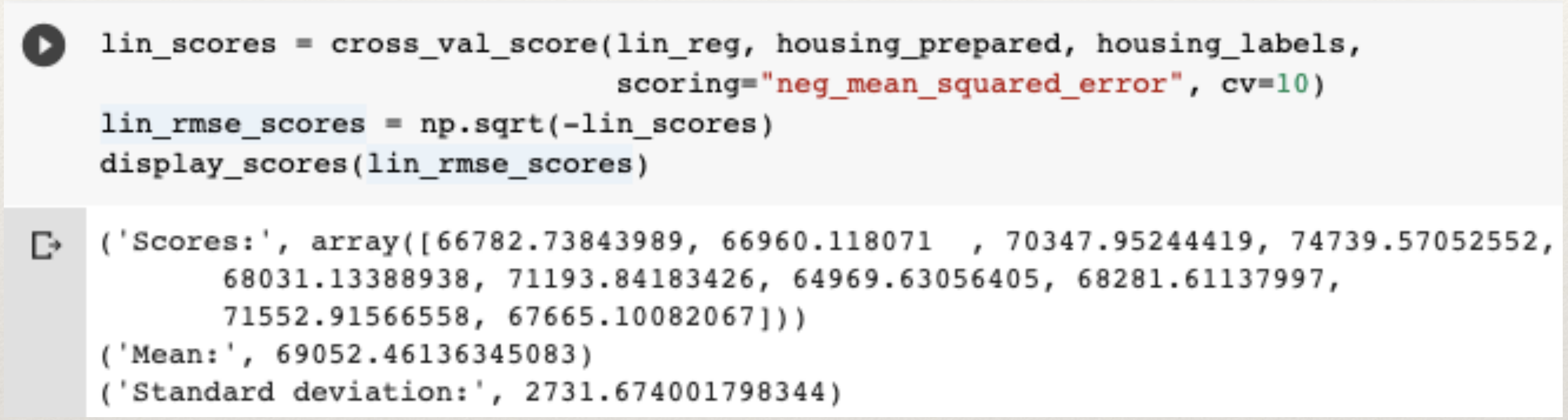
```
('Scores:', array([70194.33680785, 66855.16363941, 72432.58244769, 70758.73896782,
       71115.88230639, 75585.14172901, 70262.86139133, 70273.6325285 ,
       75366.87952553, 71231.65726027]))
('Mean:', 71407.68766037929)
('Standard deviation:', 2439.4345041191004)
```

Observation:

- the good: I have k models, so this techniques gives me a standard deviation too
    - it came at the cost of multiple trainings, you cannot afford it always..
- the bad: the score with a more complex model is **worse** than that with a simpler one.. no progress?
    - wait - we are comparing LinearRegression w/o CV (RMSE 68628) with DecisionTreeRegressor w CV (RMSE 71407). For a fair comparison, to be sure, run CV also for LinearRegression.. you get 69052 with a std deviation of 2731.. (see next for reference)

Not getting better: it is comparably bad, even with a more complex model and with proper CV..

(the right comparison discussed in previous slide)



```
lin_scores = cross_val_score(lin_reg, housing_prepared, housing_labels,
                             scoring="neg_mean_squared_error", cv=10)
lin_rmse_scores = np.sqrt(-lin_scores)
display_scores(lin_rmse_scores)
```

```
('Scores:', array([66782.73843989, 66960.118071  , 70347.95244419, 74739.57052552,
       68031.13388938, 71193.84183426, 64969.63056405, 68281.61137997,
       71552.91566558, 67665.10082067]))
('Mean:', 69052.46136345083)
('Standard deviation:', 2731.674001798344)
```

Try again another model

Are you able to try a RandomForestRegressor yourself?

- HINT: sklearn documentation, StackOverflow, …

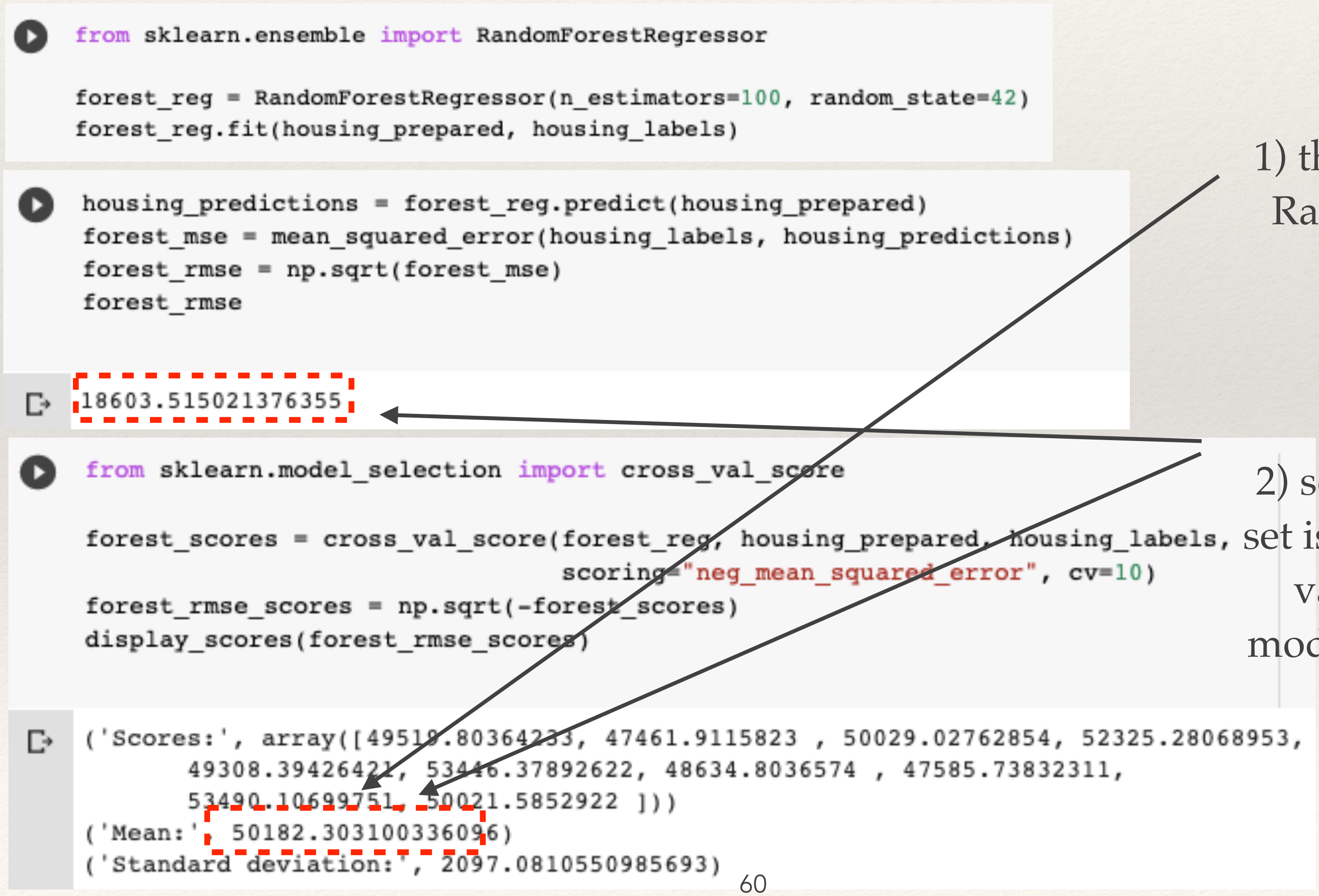
Practice R14 | 5 minutes | Time to code!

Model Evaluation: cont’d

(… the karma: “I need to make a choice. What do I try next?” …)

Try one more model: Random Forest.

- it is an Ensemble Learning technique: build a model on top of many other models
  - it works by training many Decision Trees on random subsets of the features, then averaging out their predictions



```
from sklearn.ensemble import RandomForestRegressor

forest_reg = RandomForestRegressor(n_estimators=100, random_state=42)
forest_reg.fit(housing_prepared, housing_labels)
```

```
housing_predictions = forest_reg.predict(housing_prepared)
forest_mse = mean_squared_error(housing_labels, housing_predictions)
forest_rmse = np.sqrt(forest_mse)
forest_rmse
```

```
18603.515021376355
```

```
from sklearn.model_selection import cross_val_score

forest_scores = cross_val_score(forest_reg, housing_prepared, housing_labels,
                                scoring="neg_mean_squared_error", cv=10)
forest_rmse_scores = np.sqrt(-forest_scores)
display_scores(forest_rmse_scores)
```

```
('Scores:', array([49519.80364233, 47461.9115823 , 50029.02762854, 52325.28068953,
       49308.39426421, 53446.37892622, 48634.8036574 , 47585.73832311,
       53490.10699751, 50021.5852922 ]))
('Mean:', 50182.303100336096)
('Standard deviation:', 2097.0810550985693)
```

1) this is MUCH better: Random Forests look very promising.

2) score on the training set is still << than on the validation sets: the model is still overfitting the training set.

Have you noted the execution of last cells of code became much slower over last few steps?

Have you understood:

- which step precisely?
- why?

The timing is NOT going to get better.. note it down..

Model Evaluation w/ **k-fold Cross-Validation**

Possible solutions:

- simplify the model
- constrain it (i.e. regularize it)
- get a lot more training data

What would you choose?

Towards fine-tuning..

Possible solutions:

- simplify the model → we just moved to a more complex.. perhaps try others?
- constrain it (i.e. regularize it) → this is **fine tuning of hyperparameters**
- get a lot more training data → this (alone) might work only in some cases..

Towards fine-tuning..

Possible solutions:

- simplify the model → we just moved to a more complex.. perhaps try others?
- constrain it (i.e. regularize it) → this is **fine tuning of hyper-pameters**
- get a lot more training data → this (alone) might work only in some cases..

Try out many other models from various categories of ML algos

- e.g. Support Vector Machines with different kernels
- e.g. possibly a Neural Network..

w/o spending too much time (yet) tweaking the hyper-parameters

The goal is to **shortlist a few** (5ish?) **promising models**, and continue the work in parallel with them altogether

- yes, the work is self-organising in various branches.. to be constantly compared..

Hyper-parameters and **model fine-tuning**

Let's assume that I have now a shortlist of (few) promising models.

I need to fine-tune them.

One way to do that would be to fiddle with the hyper-parameters manually, until you find a great combination of their values that "magically" works

- this would be very tedious and time-consuming..

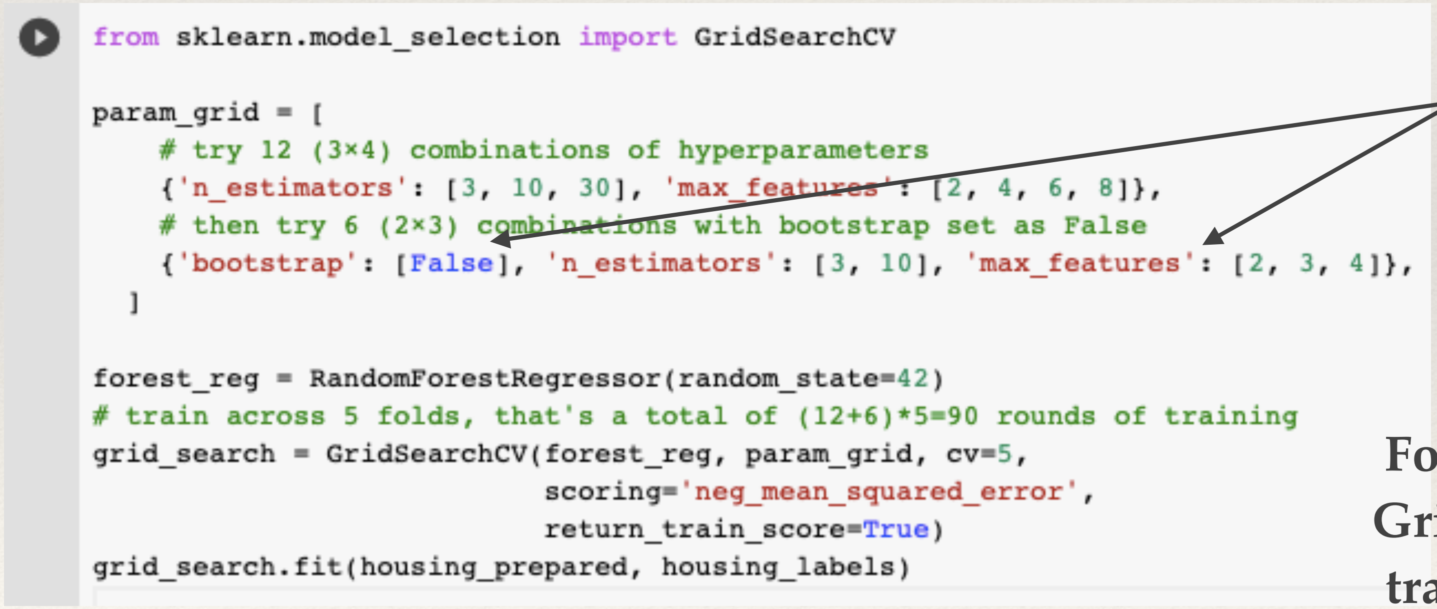
There are various ways to automatically do so:

- **GridSearch**
- **Randomised Search**
- **Ensemble Methods**

Grid Search

Grid Search does the **search** (via CV) of the best parameters across all permutations in the parameters' **grid**

- all you need to do is tell which hyper-parameters you want it to experiment with, and what values to try out



```
from sklearn.model_selection import GridSearchCV

param_grid = [
    # try 12 (3×4) combinations of hyperparameters
    {'n_estimators': [3, 10, 30], 'max_features': [2, 4, 6, 8]},
    # then try 6 (2×3) combinations with bootstrap set as False
    {'bootstrap': [False], 'n_estimators': [3, 10], 'max_features': [2, 3, 4]},
  ]

forest_reg = RandomForestRegressor(random_state=42)
# train across 5 folds, that's a total of (12+6)*5=90 rounds of training
grid_search = GridSearchCV(forest_reg, param_grid, cv=5,
                           scoring='neg_mean_squared_error',
                           return_train_score=True)
grid_search.fit(housing_prepared, housing_labels)
```

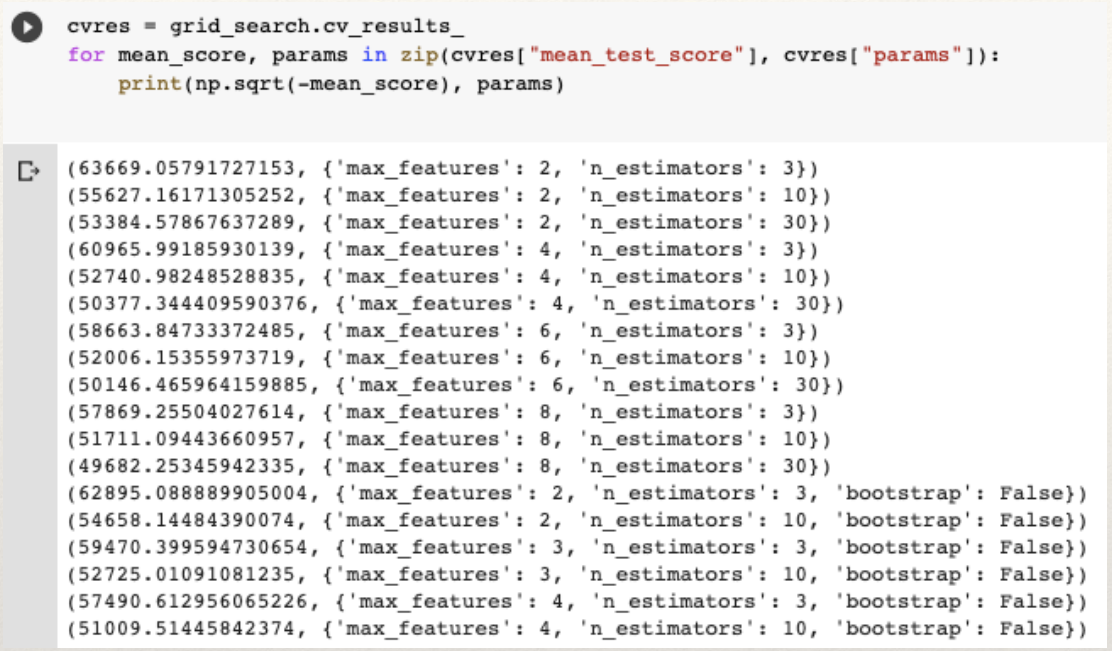
Note: do not worry about what these params mean: these are for RandomForest Regressor, others do have different ones..

Focus on the fact that this GridSearch launches 90ish training passes in one go!

The timing is NOT going to get better.. as expected..

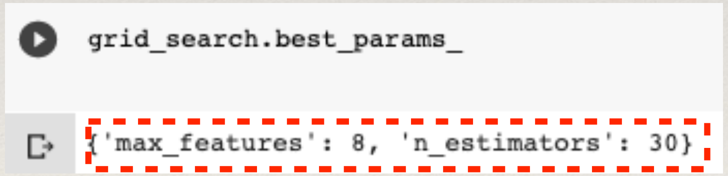
Grid Search

Let's look at the score of each hyperparameter combination tested during the grid search:



```
cvres = grid_search.cv_results_
for mean_score, params in zip(cvres["mean_test_score"], cvres["params"]):
    print(np.sqrt(-mean_score), params)
```

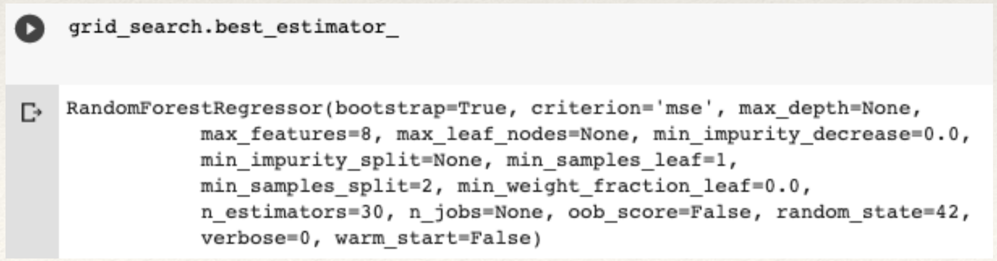
```
(63669.05791727153, {'max_features': 2, 'n_estimators': 3})
(55627.16171305252, {'max_features': 2, 'n_estimators': 10})
(53384.57867637289, {'max_features': 2, 'n_estimators': 30})
(60965.99185930139, {'max_features': 4, 'n_estimators': 3})
(52740.98248528835, {'max_features': 4, 'n_estimators': 10})
(50377.344409590376, {'max_features': 4, 'n_estimators': 30})
(58663.84733372485, {'max_features': 6, 'n_estimators': 3})
(52006.15355973719, {'max_features': 6, 'n_estimators': 10})
(50146.465964159885, {'max_features': 6, 'n_estimators': 30})
(57869.25504027614, {'max_features': 8, 'n_estimators': 3})
(51711.09443660957, {'max_features': 8, 'n_estimators': 10})
(49682.25345942335, {'max_features': 8, 'n_estimators': 30})
(62895.088889905004, {'max_features': 2, 'n_estimators': 3, 'bootstrap': False})
(54658.14484390074, {'max_features': 2, 'n_estimators': 10, 'bootstrap': False})
(59470.399594730654, {'max_features': 3, 'n_estimators': 3, 'bootstrap': False})
(52725.01091081235, {'max_features': 3, 'n_estimators': 10, 'bootstrap': False})
(57490.612956065226, {'max_features': 4, 'n_estimators': 3, 'bootstrap': False})
(51009.51445842374, {'max_features': 4, 'n_estimators': 10, 'bootstrap': False})
```



```
grid_search.best_params_
```

```
{'max_features': 8, 'n_estimators': 30}
```

The best hyper-parameters combination found is this one.



```
grid_search.best_estimator_
```

```
RandomForestRegressor(bootstrap=True, criterion='mse', max_depth=None,
           max_features=8, max_leaf_nodes=None, min_impurity_decrease=0.0,
           min_impurity_split=None, min_samples_leaf=1,
           min_samples_split=2, min_weight_fraction_leaf=0.0,
           n_estimators=30, n_jobs=None, oob_score=False, random_state=42,
           verbose=0, warm_start=False)
```

The RMSE score for this combination is 49,682 (code not shown): slightly better than the score you got earlier using the default hyper-parameter values (it was 50,182).

Some fine tuning worked!

Would you stop here!?

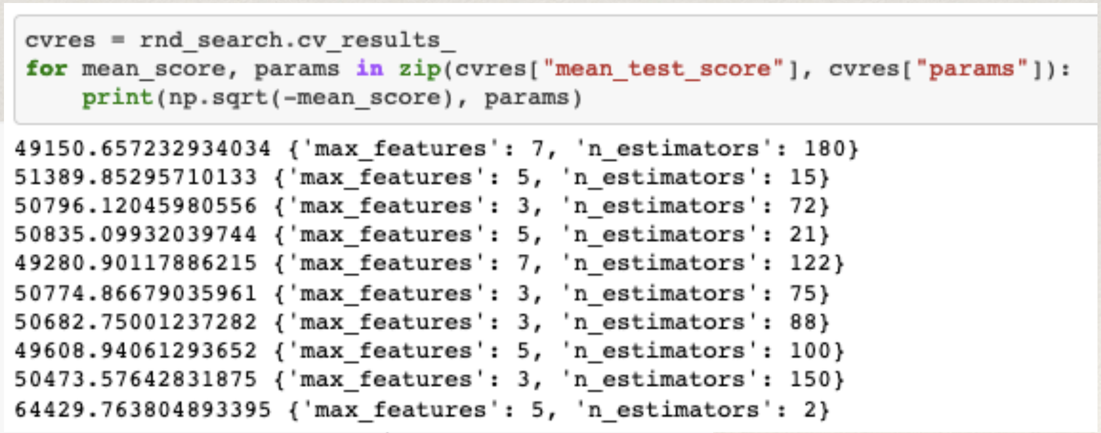
- *Hint: the chosen parameters happens to be the* ***maximum*** *values that were evaluated..*

Randomized Search

Grid Search is fine when exploring relatively few combinations. Move to **Randomized Search** if you want a larger hyperparameter search space

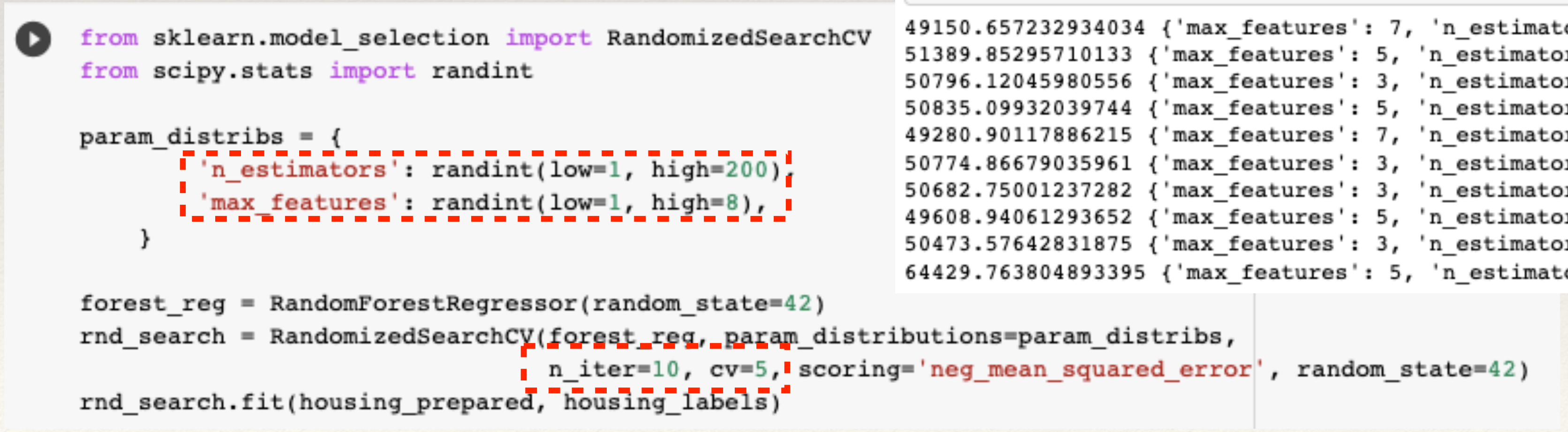
Instead of trying out all possible combinations, it evaluates a **given number of random combinations by selecting a random value for each hyperparameter at every iteration**.

- if you let the randomised search run for, say, 1000 iterations, this approach will explore 1,000 different values for <u>each</u> hyperparameter (instead of just a few values per hyperparameter as with GridSearch)



```
cvres = rnd_search.cv_results_
for mean_score, params in zip(cvres["mean_test_score"], cvres["params"]):
    print(np.sqrt(-mean_score), params)
```

```
49150.657232934034 {'max_features': 7, 'n_estimators': 180}
51389.85295710133 {'max_features': 5, 'n_estimators': 15}
50796.12045980556 {'max_features': 3, 'n_estimators': 72}
50835.09932039744 {'max_features': 5, 'n_estimators': 21}
49280.90117886215 {'max_features': 7, 'n_estimators': 122}
50774.86679035961 {'max_features': 3, 'n_estimators': 75}
50682.75001237282 {'max_features': 3, 'n_estimators': 88}
49608.94061293652 {'max_features': 5, 'n_estimators': 100}
50473.57642831875 {'max_features': 3, 'n_estimators': 150}
64429.763804893395 {'max_features': 5, 'n_estimators': 2}
```



```
from sklearn.model_selection import RandomizedSearchCV
from scipy.stats import randint

param_distribs = {
        'n_estimators': randint(low=1, high=200),
        'max_features': randint(low=1, high=8),
    }

forest_reg = RandomForestRegressor(random_state=42)
rnd_search = RandomizedSearchCV(forest_reg, param_distributions=param_distribs,
                                n_iter=10, cv=5, scoring='neg_mean_squared_error', random_state=42)
rnd_search.fit(housing_prepared, housing_labels)
```

Not bad at all..

Ensemble Methods

This is another way to fine-tune your system: try to **combine the models that perform best**, as the group (or "**ensemble**") will often perform better than the best individual model

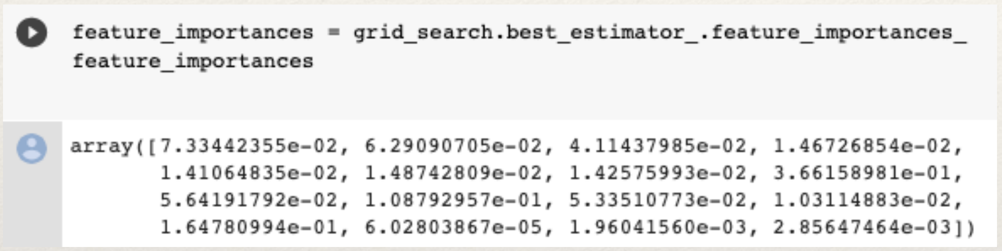
- we saw this already: RandomForest performed better than the individual DecisionTrees it relied on
- especially good if the individual models make very different types of prediction errors

Analyze the best models and their prediction errors

Crucial to understand **why a model is working better than others**

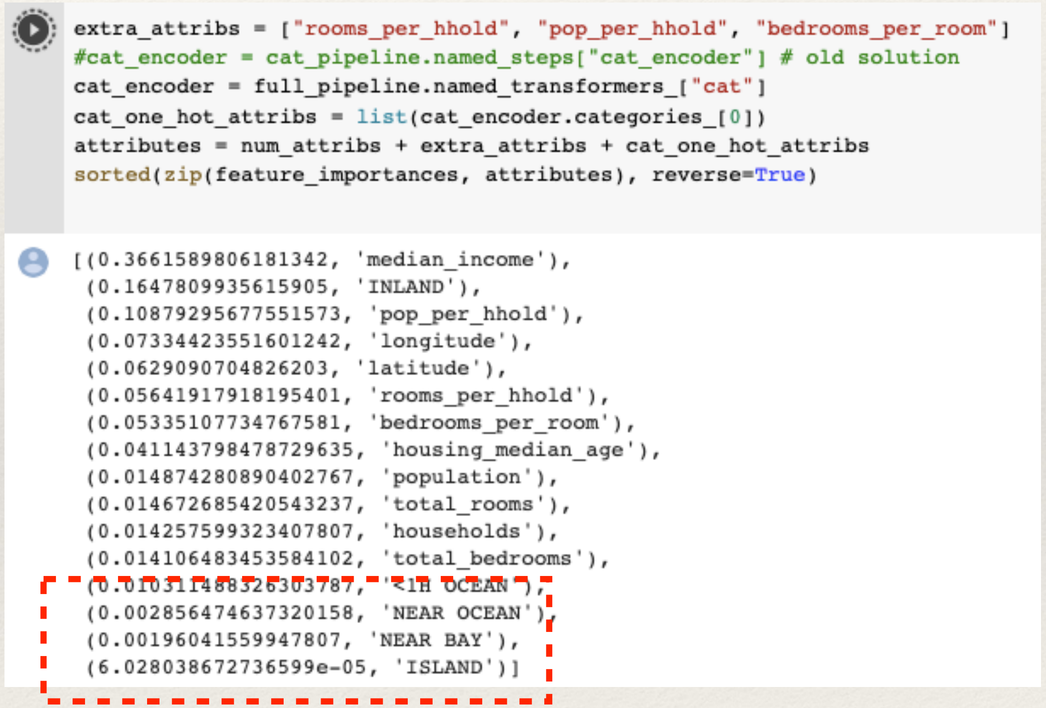
- "who drove this model to the point it performs the best?"

Feature importance: e.g. RandomForestRegressor can indicate the relative importance of each attribute for making accurate predictions



```
feature_importances = grid_search.best_estimator_.feature_importances_
feature_importances
```

```
array([7.33442355e-02, 6.29090705e-02, 4.11437985e-02, 1.46726854e-02,
       1.41064835e-02, 1.48742809e-02, 1.42575993e-02, 3.66158981e-01,
       5.64191792e-02, 1.08792957e-01, 5.33510773e-02, 1.03114883e-02,
       1.64780994e-01, 6.02803867e-05, 1.96041560e-03, 2.85647464e-03])
```



```
extra_attribs = ["rooms_per_hhold", "pop_per_hhold", "bedrooms_per_room"]
#cat_encoder = cat_pipeline.named_steps["cat_encoder"] # old solution
cat_encoder = full_pipeline.named_transformers_["cat"]
cat_one_hot_attribs = list(cat_encoder.categories_[0])
attributes = num_attribs + extra_attribs + cat_one_hot_attribs
sorted(zip(feature_importances, attributes), reverse=True)
```

```
[(0.3661589806181342, 'median_income'),
 (0.1647809935615905, 'INLAND'),
 (0.10879295677551573, 'pop_per_hhold'),
 (0.07334423551601242, 'longitude'),
 (0.0629090704826203, 'latitude'),
 (0.05641917918195401, 'rooms_per_hhold'),
 (0.05335107734767581, 'bedrooms_per_room'),
 (0.041143798478729635, 'housing_median_age'),
 (0.014874280890402767, 'population'),
 (0.014672685420543237, 'total_rooms'),
 (0.014257599323407807, 'households'),
 (0.014106483453584102, 'total_bedrooms'),
 (0.010311488326303787, '<1H OCEAN'),
 (0.002856474637320158, 'NEAR OCEAN'),
 (0.001960415599947807, 'NEAR BAY'),
 (6.028038672736599e-05, 'ISLAND')]
```

You may want to try dropping some of the less useful features

(e.g. apparently only one ocean_proximity category is really useful, so you could drop the others)

But do more!

- add extra features or, on the contrary, get rid of uninformative ones, cleaning up outliers, etc

Evaluate Your System on the Test Set

OK.

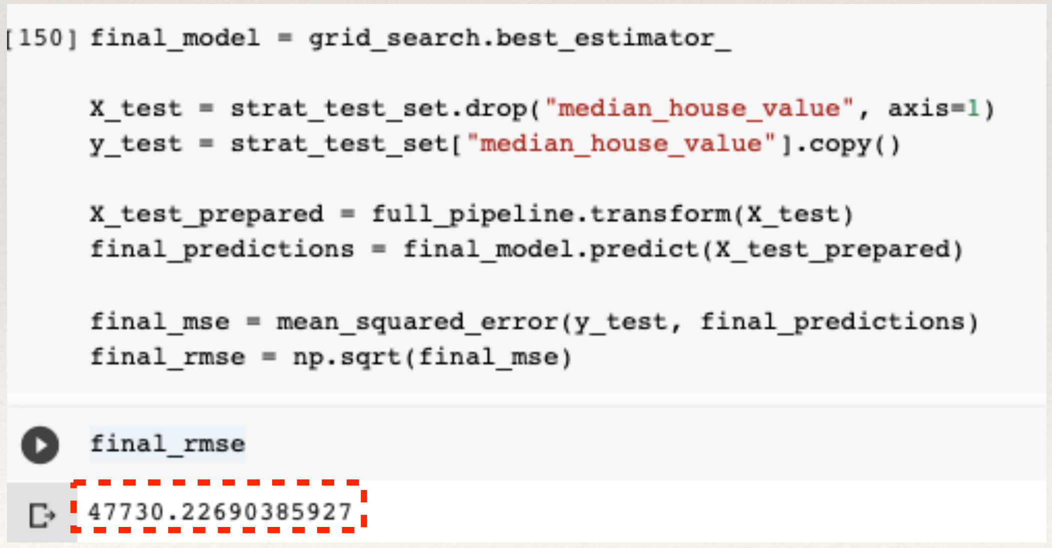
You eventually have a system that performs sufficiently well.

Now is the time to **evaluate the final model on the test set**.

Evaluate Your System on the Test Set

Easy, and nothing technical different wrt what we did already

- get features and labels from your test set, now
- run your full pipeline to transform the data (transform(), not fit_transform())
- evaluate the final model on the test set



```
[150] final_model = grid_search.best_estimator_

      X_test = strat_test_set.drop("median_house_value", axis=1)
      y_test = strat_test_set["median_house_value"].copy()

      X_test_prepared = full_pipeline.transform(X_test)
      final_predictions = final_model.predict(X_test_prepared)

      final_mse = mean_squared_error(y_test, final_predictions)
      final_rmse = np.sqrt(final_mse)
```

```
final_rmse
```

```
47730.22690385927
```

Not bad. Let's stop here.

Communicate out. You got an idea of features importances (median income as main predictor), you studied and excluded some features (e.g. some of the ocean vicinity ones), plenty of lesson learned (what worked and what not), you got a performance that can be compared with others

Depending on the case, (your boss will) **consider to switch the production system to this one**. (then, plenty of monitor, re-checks, etc..)

That's it,
for our Lab on **Regression**