# MACH

## 2. End-to-End Machine Learning Project

# A roadmap for end-to-end ML projects


**Illustration through the California housing prices dataset**

1. **Frame the problem and look at the big picture.**
2. **Get the data.**
3. **Explore and visualize the data to gain insights.**
4. **Prepare the data for ML algorithms.**
5. **Select a model and train it.**
6. **Fine-tune your model.**
7. **Present your solution.**
8. **Launch, monitor, and maintain your system.**

EPITA
ÉCOLE D'INGÉNIEURS EN INFORMATIQUE

# Frame the problem and look at the big picture

01. Define the objective in business terms.

02. How will your solution be used?

03. What are the current solutions/workarounds (if any)?

04. How should you frame this problem (supervised/unsupervised, online/offline, etc.)

05. How should performance be measured?

06. Is the performance measure aligned with the business objective?

07. What would be the minimum performance needed to reach the business objective?

08. What are comparable problems? Can you reuse experience or tools?

09. Is human expertise available?

10. How would you solve the problem manually?

11. List the assumptions you or others have made so far.

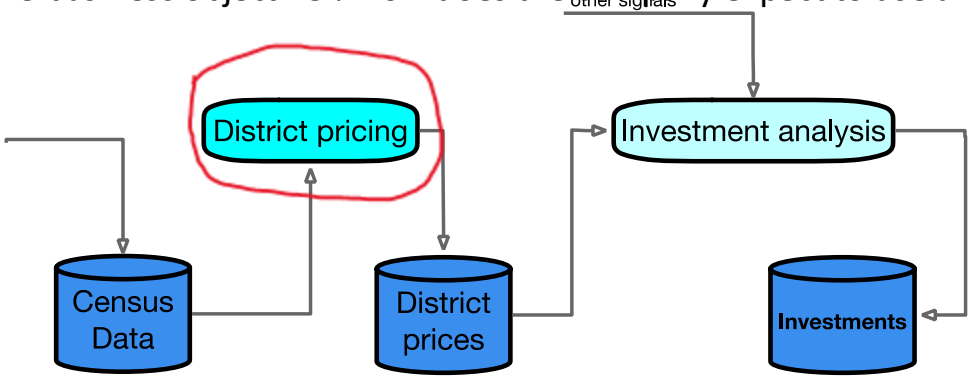
12. Verify assumptions if possible.

EPITA

# Look at the big picture

Use census data to build a model of housing prices in California.

Why ? ⇔ what is exactly the business objective ? How does the company expect to use and benefit from the model ?




Answering to these questions is important: it will determine how to frame the model, which **algorithms** to use, which **performance measure** to **evaluate** the model, and how **much effort** to spend tweaking the model.

Next Question: what does the current solution look like? (if any) ➔ prices are currently manually estimated by experts (costly and time-consumming, estimates off by more than 30%).

EPITA

# Frame the pb

- How should you frame this problem (supervised/unsupervised, online/offline, etc.) ?
  - Supervised learning task (we have labeled examples)
  - Regression task (predict a value)
    - Multiple regression (multiple features)
    - Univariate regression (predict a single value)
  - No continuous flow of data, no no need to adjust change rapidly ➔ batch learning

# Select a performance measure

Root Mean Square Error (RMSE): a higher weight is given to large errors.

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

# Notations

- *m* is the number of instances
- $\mathbf{x}^{(i)}$ is the vector of all the features value of the $i^{th}$ instance, $y^{(i)}$ is the label (the desired output) for that instance
- **X** is a matrix containing all the feature values:
  - One row per instance
  - One column per feature
- *h* is the system prediction function (aka hypothesis). It outputs a predicted value $\hat{y}^{(i)}$
- *Lowercase italic* for scalar values and ($m$, $y^{(i)}$) function name ($h$), **lowercase bold** for vectors, and **UPPERCASE BOLD** for matrices

$$\mathbf{x}^{(1)} = \begin{pmatrix} -118.9 \\ 33.91 \\ 1,415 \\ 38,372 \end{pmatrix}$$

Long.
Lat.
#inhabitants
Median income

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

Median house price

$$\mathbf{X} = \begin{pmatrix} \left(\mathbf{x}^{(1)}\right)^\top \\ \left(\mathbf{x}^{(2)}\right)^\top \\ \vdots \\ \left(\mathbf{x}^{(m)}\right)^\top \end{pmatrix} \quad \mathbf{X} = \begin{pmatrix} -118.9 & 33.91 & 1,415 & 38,372 \\ \vdots & \vdots & \vdots & \vdots \end{pmatrix}$$

EPITA

# Other measures and norms

- RMSE the most common
- Mean Absolute Error (MAE) may be used if there are many outliers
- Both are ways to measure the distance between two vectors. Various distances are possible:
  - RMSE ⇔ Euclidean norm. Also called $\ell_2$ norm noted $||.||_2$ or just $||.||$
  - MAE ⇔ $\ell_1$ norm noted $||.||_1$ aka Manhattan norm.
  - $\ell_k$ norm: $$||\mathbf{v}||_k = \left(|v_1|^k + |v_2|^k + \ldots + |v_n|^k\right)^{1/k}$$
    - $\ell_0$ norm gives the number of non-zero elements
    - $\ell_\infty$ norm gives the maximum abolute value in the vector.

# Check the assumptions

It is a good practice to list and verify assumptions that have been made so far (by anyone).

- This can help catch serious issue early on.
- E.g., prediction of median prices but predictions are not used as such in the downstream taks:
  - Category vs prices ➔ Classification task not an useless regression task!

# Get the data

automate as much as possible so you can easily get fresh data

1. List the data you need and how much you need.
2. Find and document where you can get that data.
3. Check how much space it will take.
4. Check legal obligations, and get the authorization if necessary.
5. Get access authorizations.
6. Create a workspace (with enough storage space).
7. Get the data.
8. Convert the data to a format you can easily manipulate (without changing the data itself).
9. Ensure sensitive information is deleted or protected (e.g., anonymized).
10. Check the size and type of data (time series, sample, geographical, etc.).
11. Sample a test set, put it aside, and never look at it (no data snooping!).

# Take a quick look at the data structure

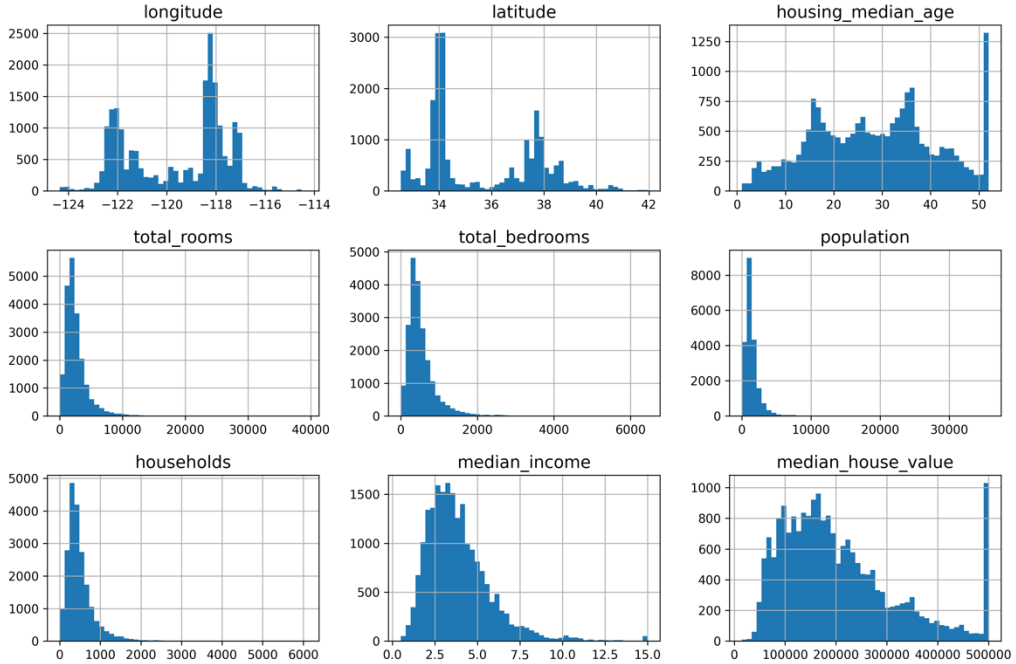
# instances
#attributes
Types of attributes
Missing values?
Capped values?

- Useful methods:
  - Info() to get a quick description of the data (#instances, attribute types, #non null values)
  - describe() to get a summary of the numerical attributes.


# Let’s look at the notebook

# Visualizing data to get insights/warning

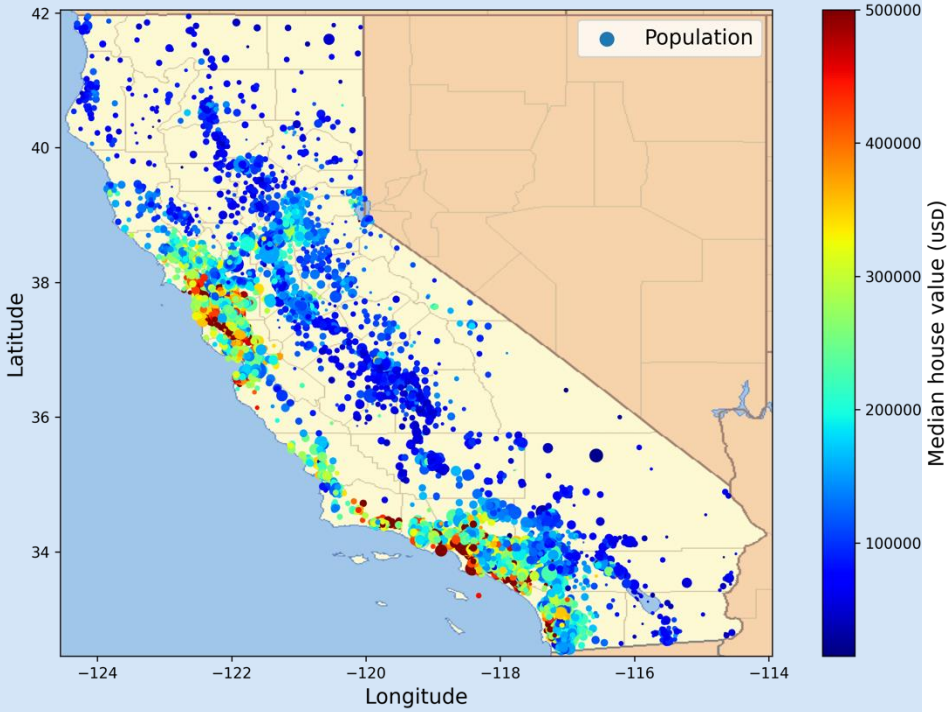
- Median incomes not in US $:
  - x 10,000 US $
  - Capped at 15 for higher incomes.
- The housing median age also capped at 500k$.
  - It is a problem since it is the label to predict.
    - Collect proper labels for capped ones.
    - Remove those districts from the training (and the test).
- Attributes with very different scales ➔ feature scaling.
- Many histograms are skewed right: they extend much farther to the right of the median than to the left.
  - A bit harder for some ML algorithms to detect patterns ➔ transformation.



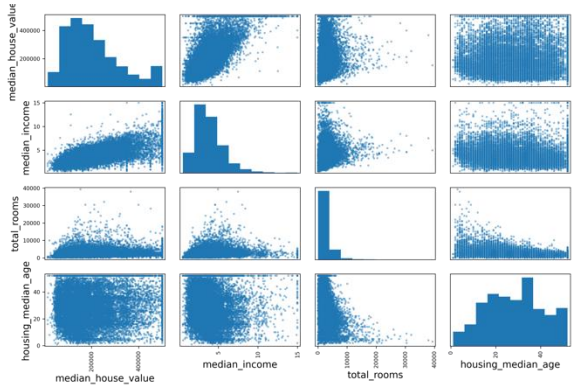
# Creating a test set

- To avoid overfitting
- Pick some instances randomly:
  - 20% of the dataset (or less if your dataset is very large) and set them aside.
  - Make sure to have the  same test set at each run
    - You can set the random see to 42 :D.
- **Stratified sampling** to assure representativeness.

# Explore and visualize data



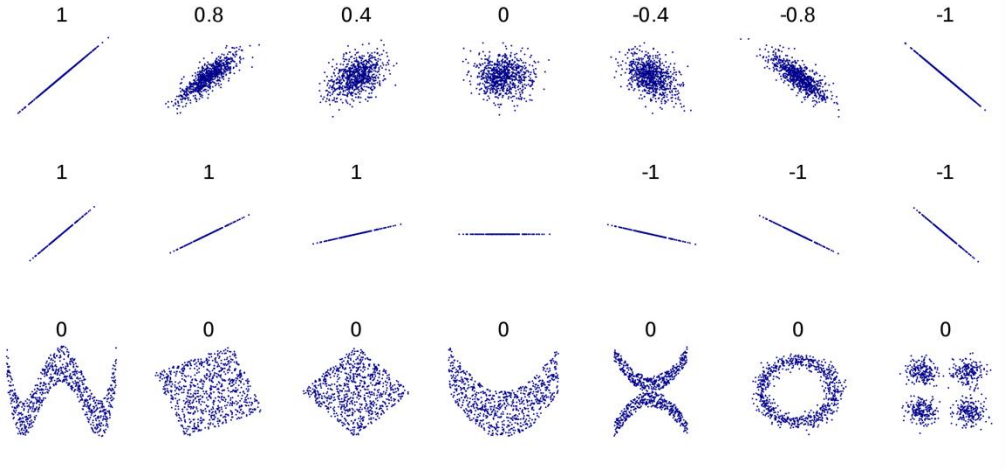

## Looking for correlations

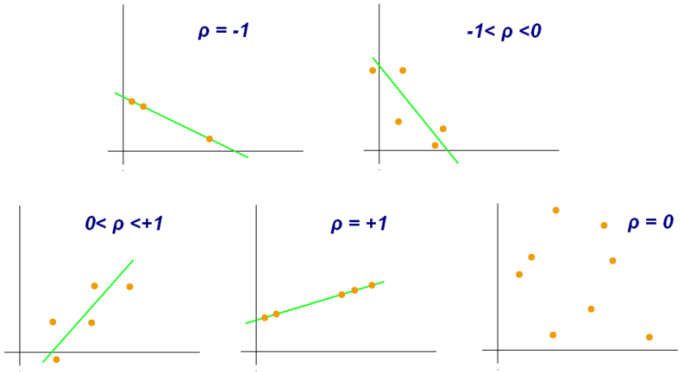


# Correlations

- Pearson's correlation coefficient is the covariance of the two variables divided by the product of their standard deviations.

$$\rho_{X,Y} = \frac{\operatorname{cov}(X,Y)}{\sigma_X \sigma_Y}$$




https://en.wikipedia.org/wiki/Pearson_correlation_coefficient


# Let’s look at the notebook

# Experimenting with attribute combinations

Try out various attribute combinations

- #rooms_per_house
- Bedrooms-ratio
- People_per_house
- …

# Prepare the data for ML algorithms

It's time to prepare the data for your ML algorithm.

- Do not do this manually, write functions instead!
  - **Reproducibility**: on any dataset, on a new version of the current dataset …
  - **Capitalisation**: you will gradually build a library of transformation function that you can reuse on future project and/or in your live system to transform the new data before feeding to the algorithm.
  - **Effectiveness**: make it possible to easily try various transformations and see which combination work best.

# Clean the data

- **Missing values**: Most ML algorithms cannot handle them!
- Several options to fix them:
  - **Get rid** of the **instances** containing missing values
  - **Get rid** of the **whole attribute** containing at least an instance with a missing value for this attribute.
  - **Inputation: set** the missing values to **some values**: (zero, the mean, the median, most frequent value, …)

# Scikit-learn Design

- Consistency: all objects share a consistent and simple interface.
  - **Estimators**: any object that can estimate some parameters on a dataset (e.g., SimpleImputer).
    - Estimation performed by the **fit()** method and it takes a **dataset** as a parameter, or two for supervised learning (the **labels**).
    - Any other parameter needed to guide the estimation process is considered as a **hyperparameter** (such a the SimpleImputer's strategy), and it must be set as an instance variable (generally via a constructor parameter).
  - Transformers:
  - Predictors


https://arxiv.org/pdf/1309.0238.pdf

# Scikit-learn Design (2)

- Consistency: all objects share a consistent and simple interface.
  - ✓ Estimators:
  - **Transformers**: estimators that can also **transform** a dataset
    - Transformation performed by the **transform()** method with the **dataset** to transform as a **parameter**. It returns the transformed dataset.
    - Transformation generally relies on the learned parameters (e.g., SimpleImputer).
    - All transformers have a convenience method **fit_transform()** ⇔ fit() then transform() (but sometimes fit_transform() is optimized and runs much faster).
  - Predictors

https://arxiv.org/pdf/1309.0238.pdf

# Scikit-learn Design (3)

- Consistency: all objects share a consistent and simple interface.
  - ✓ Estimators:
  - ✓ Transformers: estimators that can also transform a dataset
  - **Predictors**: estimators that are capable of making a prediction (e.g., LinearRegression model)
    - **predict()** method that takes a dataset of new instances and returns a dataset of corresponding predictions.
    - **score()** method that measures the **quality** of the predictions, given a **test set** in the case of supervised learning.


https://arxiv.org/pdf/1309.0238.pdf

# Scikit-learn Design (4)

- **Inspection**: all the estimator's hyperparameters are accessible directly via public instance variables (e.g., imputer.strategy) as well as the estimator's parameters (with an underscore suffix, e.g., imputer.statistics_).
- **Non proliferation of classes:** dataset are represented **as NumPy arrays** or **SciPy sparse matrices** instead of homemade classes. **Hyperparameters** are just regular Python strings or numbers.
- **Composition**: existing building blocks reused as much as possible. ➔ easy to create **Pipeline** estimator for an arbitrary sequence of transformers followed by a final estimator.
- **Sensible defaults**: Scikit-Learn provides reasonable default values for most parameters, making it easy to quickly create a baseline working system.


# Handling text and categorical attributes ➔ Let’s look at the notebook

# Handling text and categorical attributes

- So far, we only dealt with numerical attributes, what about text or categorical attributes ?
- Some ML algorithms require numbers:
  - Ordinal encoder
    - Issue: the similarity between values may have no sense.
  - One hot encoder

# Feature scaling and transformation

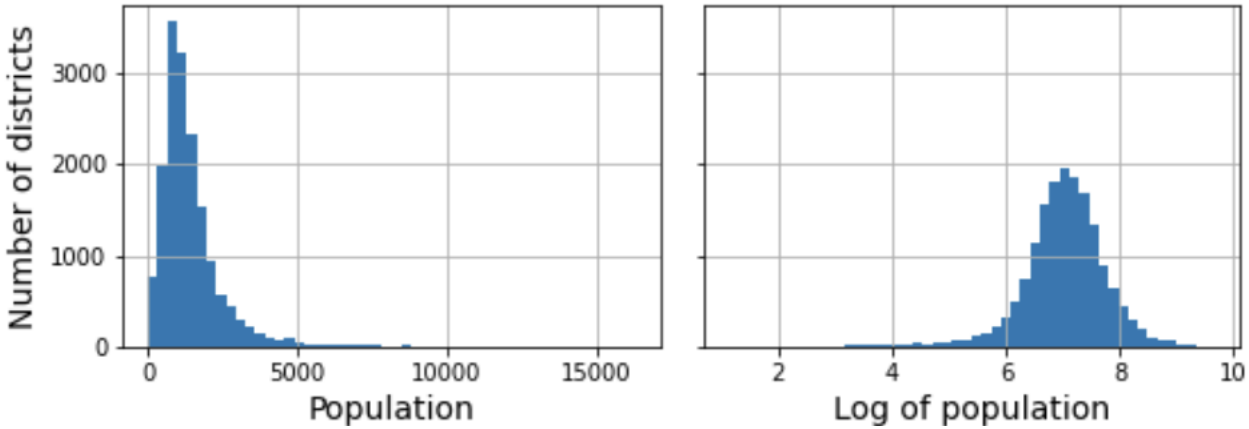
- Most of ML algorithms do not perform well when the input numerical attributes have very different scales.
- E.g.: total #rooms ranges from 6 to 39,320 while the median incomes only range from 0 to 15.
  - ➔ without any scaling, most models will be biased toward ignoring the median income and focusing more on the number of rooms.
- MinMax scaling
- Standardization

```python
from sklearn.preprocessing import MinMaxScaler

min_max_scaler = MinMaxScaler(feature_range=(-1, 1))
housing_num_min_max_scaled = min_max_scaler.fit_transform(housing_num)
```

```python
from sklearn.preprocessing import StandardScaler

std_scaler = StandardScaler()
housing_num_std_scaled = std_scaler.fit_transform(housing_num)
```

Standardization= (value – mean) / std

# Heavy tail




- Min-max scaling and standardization will squash most values into a small range
- Before scaling the feature, transform it to shrink the heavy tail and if possible to make the distribution roughly symetrical.
- Another approach: bucketizing the feature
- Multimodal distribution ➔ add a feature per mode (peak)
    - Radial basis function (RBF)

EPITA

- Most of transformers have an inverse_transform() method making it easy to compute the inverse of their transformations.

```
from sklearn.linear_model import LinearRegression

target_scaler = StandardScaler()
scaled_labels = target_scaler.fit_transform(housing_labels.to_frame())

model = LinearRegression()
model.fit(housing[["median_income"]], scaled_labels)
some_new_data = housing[["median_income"]].iloc[:5]  # pretend this is new data

scaled_predictions = model.predict(some_new_data)
predictions = target_scaler.inverse_transform(scaled_predictions)
```

- A simpler solution is to use a TransformedTargetRegressor.
  - We just need to construct it. ..

```
from sklearn.compose import TransformedTargetRegressor

model = TransformedTargetRegressor(LinearRegression(),
                                   transformer=StandardScaler())
model.fit(housing[["median_income"]], housing_labels)
predictions = model.predict(some_new_data)
```

# Custom Transformers and pipelines

- Go back to the notebook.

# Select and train a model

Short-list promising models

Notes:

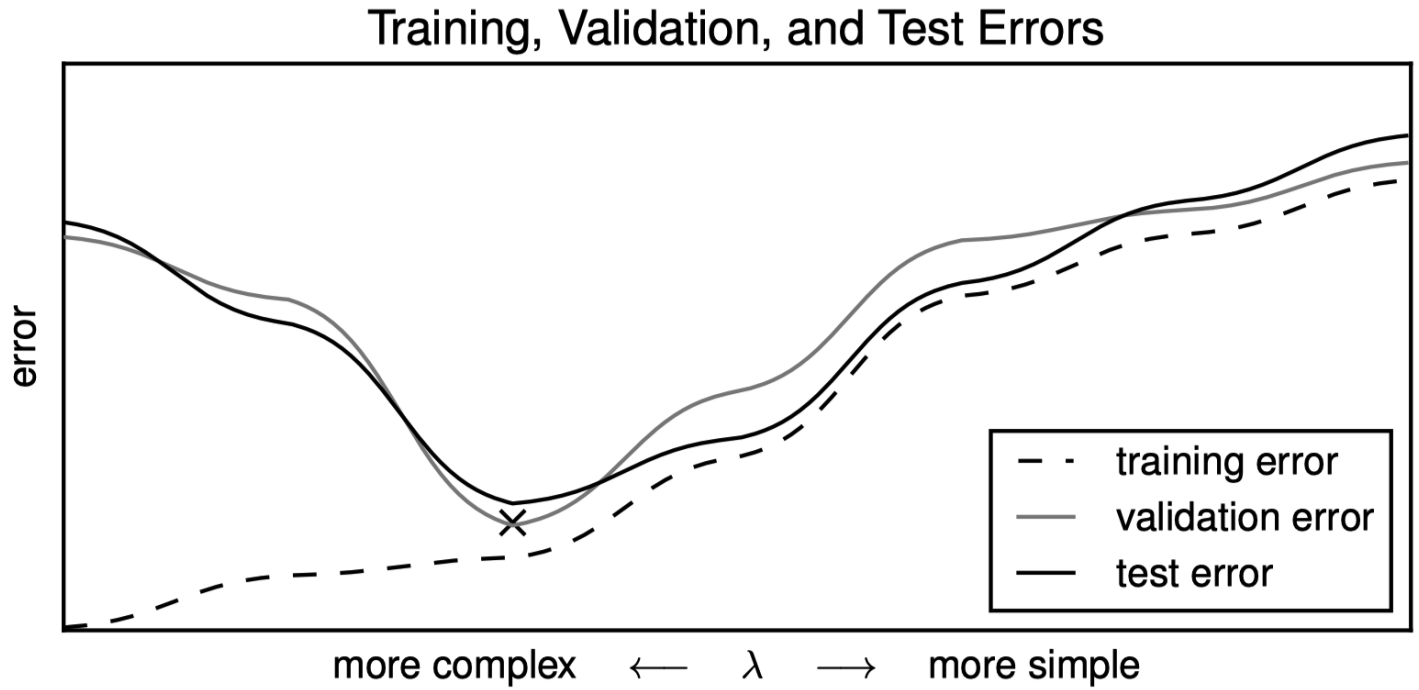
- If the data is huge, you may want to sample smaller training sets so you can train many different models in a reasonable time (be aware that this penalizes complex models such as large neural nets or Random Forests).
- Once again, try to automate these steps as much as possible.

1. Train many quick and dirty models from different categories (e.g., linear, naive, Bayes, SVM, Random Forests, neural net, etc.) using standard parameters.
2. Measure and compare their performance.
   - For each model, use N-fold cross-validation and compute the mean and standard deviation of their performance.
3. Analyze the most significant variables for each algorithm.
4. Analyze the types of errors the models make.
   - What data would a human have used to avoid these errors?
5. Have a quick round of feature selection and engineering.
6. Have one or two more quick iterations of the five previous steps.
7. Short-list the top three to five most promising models, preferring models that make different types of errors.

EPITA
ÉCOLE D'INGÉNIEURS EN INFORMATIQUE


# Notebook: better evaluation using cross-validation

# Overfitting vs Underfitting



# Fine-Tune a model

- Grid Search
- Randomized Search
- Ensemble methods
- Analyzing the best models and their errors
- Evaluate a system on the test set

# Grid Search

- Fiddle with the hyperparameters manually, until finding the great combination of hyperparameter values.
  - Tedious work, exploration of many combinations.
- GridSearchCV class allow to test the hyperparameters using cross-validation to evaluate all the possible combinations.

➔ Notebook

# Randomized Search

- Grid Search is ok  when few combinations are explored.
- RandomizedSearchCV is often preferable, especially when the hyperparameter search space is large.
- Select a fixed number of combinations:
  - Selecting a random value for each hyperparameter at every  iteration.
- TODO: have a look at HalvingRandomSearchCV and HalvingGridSearchCV to use computational resources more efficiently, either to train faster or to explore a larger hyperparameter space.

# Ensemble Methods

- Combine the models that perform best.
- The « group » or ensemble will often perform better than the best individual model – especially if the models make very different types of errors.
- We will study later in the course.

# Analyzing the best models and their errors

- Gain good insights on the problem by inspecting the best models
- Ex: the relative importance of each attribute of each attribute for making accurate predictions.

➔ It may allow to drop the least useful features.

➔ Sklearn.feature_selection.SelectFromModel transformer can automatically drop the least useful features for you.

# Evaluate your system on the Test set

- One  can compute a 95% confidence interval for the test RMSE (**PBS2**)

# Model persistence using joblib

- Save the final model
- Now one can deploy this model to production. For example, the following code could be a script that would run in production
- You could use pickle instead, but joblib is mo efficient.

```
import joblib

joblib.dump(final_model, "my_california_housing_model.pkl")
```

```
import joblib

# extra code – excluded for conciseness
from sklearn.cluster import KMeans
from sklearn.base import BaseEstimator, TransformerMixin
from sklearn.metrics.pairwise import rbf_kernel

def column_ratio(X):
    return X[:, [0]] / X[:, [1]]

#class ClusterSimilarity(BaseEstimator, TransformerMixin):
#    [...]

final_model_reloaded = joblib.load("my_california_housing_model.pkl")

new_data = housing.iloc[:5]  # pretend these are new districts
predictions = final_model_reloaded.predict(new_data)
```

# Launch, monitor and maintain the system

1. Get your solution ready for production (plug into production data inputs, write unit tests, etc.).
2. Write monitoring code to check your system's live performance at regular intervals and trigger alerts when it drops.
    - Beware of slow degradation too: models tend to "rot" as data evolves.
    - Measuring performance may require a human pipeline (e.g., via a crowdsourcing service).
    - Also monitor your inputs' quality (e.g., a malfunctioning sensor sending random values, or another team's output becoming stale). This is particularly important for online learning systems.
3. Retrain your models on a regular basis on fresh data (automate as much as possible).

EPITA
ÉCOLE D'INGÉNIEURS EN INFORMATIQUE

# Summary

- End-to-end ML project
  - From the pb definition to the evaluation of the solution and its deployment
- Next courses:
  - We will pay a particular attention to each step and the learning models.


EPITA
ÉCOLE D'INGENIEURS EN INFORMATIQUE