Hands-on Machine Learning

ML Projects Checklists

Marc Evrard

2022-2023

université
PARIS-SACLAY

Section 1

Checklists

Global structure

There are 7 main steps:[†]

1. **Frame the problem** and look at the big picture
2. **Collect** the data
3. **Explore** the data to gain insights
4. **Prepare** the data for Machine Learning algorithms
5. Select a **model** and train it
6. **Fine-tune** your model
7. **Present** your solution

[†]A large part of these slides refer to Géron (2019)

Define the objective in business terms

- You imagine yourself **working in a company** as a data scientist
- You get assigned to **build a new model** for an ML project
- The 1^{st} thing you do is to reach your **ML project checklist**
- In this part, we will go through several checklist items

Section 2

The problem

Checklist 1: Frame the Problem

1. Define the **objective** in business terms
2. **How** will your solution be used?
3. What are the **current solutions**/workarounds?
   - What are **comparable** problems? Can you reuse experience or tools?
   - Is **human expertise** available?
   - How would you solve the problem **manually**?
4. How should you **frame this problem**? (supervised/unsupervised, online/offline, etc.)
5. How should **performance be measured**?
   - Is the performance **measure aligned** with the **business objective**?
   - What would be the **minimum performance** needed to reach the business objective?
6. List the **assumptions** you (or others) have made so far
7. **Verify** assumptions if possible

Frame the business problem

- Ask yourself (or your manager) the exact **business objective**:
  - What will be the **usage** of the model?
  - Your project may **not** be the **final goal**
- This will determine:
  - What **performance measure** you'll use
  - What **algorithm** you will select
  - How much should it be **optimized**?

Frame the problem in ML terms

- Is it a **supervised** or an **unsupervised** problem?
- Is it a **reinforcement** learning problem?
- Is it a **classification** task, **regression**, or something else?
- Select a **performance measure**
  - E.g., for **classification**: Is **recall** more, less, or equally important than **precision**?
  - E.g., for **regression**: How much must outliers be penalized? (Root Mean Square Error vs Mean Absolute Error)
- Should we use **batch** learning, **online** learning?
- It's important to take a **step back** and take the time to think, before starting to try implementing a solution

Current solutions

- Before starting the implementation of any system, it's very important to learn about **existing solutions**
- It's also a good idea to start from an existing solution and try to **improve it**
- Starting from scratch on each system is usually a waste of time
- Analyzing other people's work is also a source of **inspiration**
- You should also ask about current solutions implemented by **experts** in the **field**, who might also use completely **manual workarounds** that could give precious **insights**

Check the assumptions

- Good practice is to **check the assumptions** that were made so far
  - Both by you and by others
- It could be any of the assumptions made during the review of the various points in the **checklist**
- The best way to handle an assumption is to **test it**
- Test as much as possible, since working with statistical models implies coming across several **counterintuitive situations**

Section 3

Data

Checklist 2: Collect the data

1. List the data you need (e.g., labels or not) and **how much**
2. Find and document **where** you can get that data
3. Check how much space it will take
   - Create a workspace (with enough **storage space**)
   - Check the size and type of data (time series, sample, geographical, etc.)
4. Check legal obligations, and get **authorization** if necessary
   - Get access authorizations
5. Get the data
6. Convert data to **practical formats** (without altering them)
7. Ensure **sensitive information** is deleted or protected (e.g., anonymized, pseudonymized)
8. Sample a **test set**, put it aside, and **don't look at it!**

Collect the data

Bottom line:

- **Automate** as much as possible
  - To get **fresh data** easily
  - To **avoid mistake**
  - To save time for your next project (**investment**)

Checklist 3: Explore the data

Note: Try to get insights from a field expert for these steps

1. Create a copy of the data (possibly sampling it down)
2. Use Jupyter notebook to keep a record of your exploration
3. Study each attribute and its characteristics:
   - Name, Type (e.g., categorical, int/float, un/bounded)
   - Ratio of missing values
   - Noise level and type of noise (e.g., stochastic, outliers)
   - Relevance for the task
   - Distribution (e.g., normal, uniform)
4. For supervised learning tasks, identify the target attribute(s)
5. Visualize the data
6. Study the correlations between attributes
7. Study how you would solve the problem manually
8. Identify the promising transformations you may apply
9. Identify extra data that would be useful
10. Document what you have learned

Create the workspace (for Python)

- Use preferably Anaconda
- Create an isolated environment
  - It is strongly recommended to allow working on different projects without the risk of conflicting library versions
- Important libraries:

  JupyterLab, Matplotlib, NumPy, Pandas, SciPy, Scikit-Learn
- Use JupyterLab (or Jupyter Notebook)
  - ⚠ Pay attention to **global variables!**
  - Restart kernel regularly
  - When the project is mature, build a Python module out of the notebook

Data structure

- A great tool to investigate the data is Pandas
- After having downloaded the data, you can import `.csv` or other file formats through Pandas reading methods
- You can use the `df.head()` or `df.tail()` to view at the `DataFrame` top and bottom records
- The `df.info()` method is useful to view a quick description of the data
  - Total number of rows
  - Each attribute's type and number of non-null values
- The `df.describe()` method generates basic statistics
- The `df.memory_usage()` method returns the memory usage per column (useful if you work with large datasets)

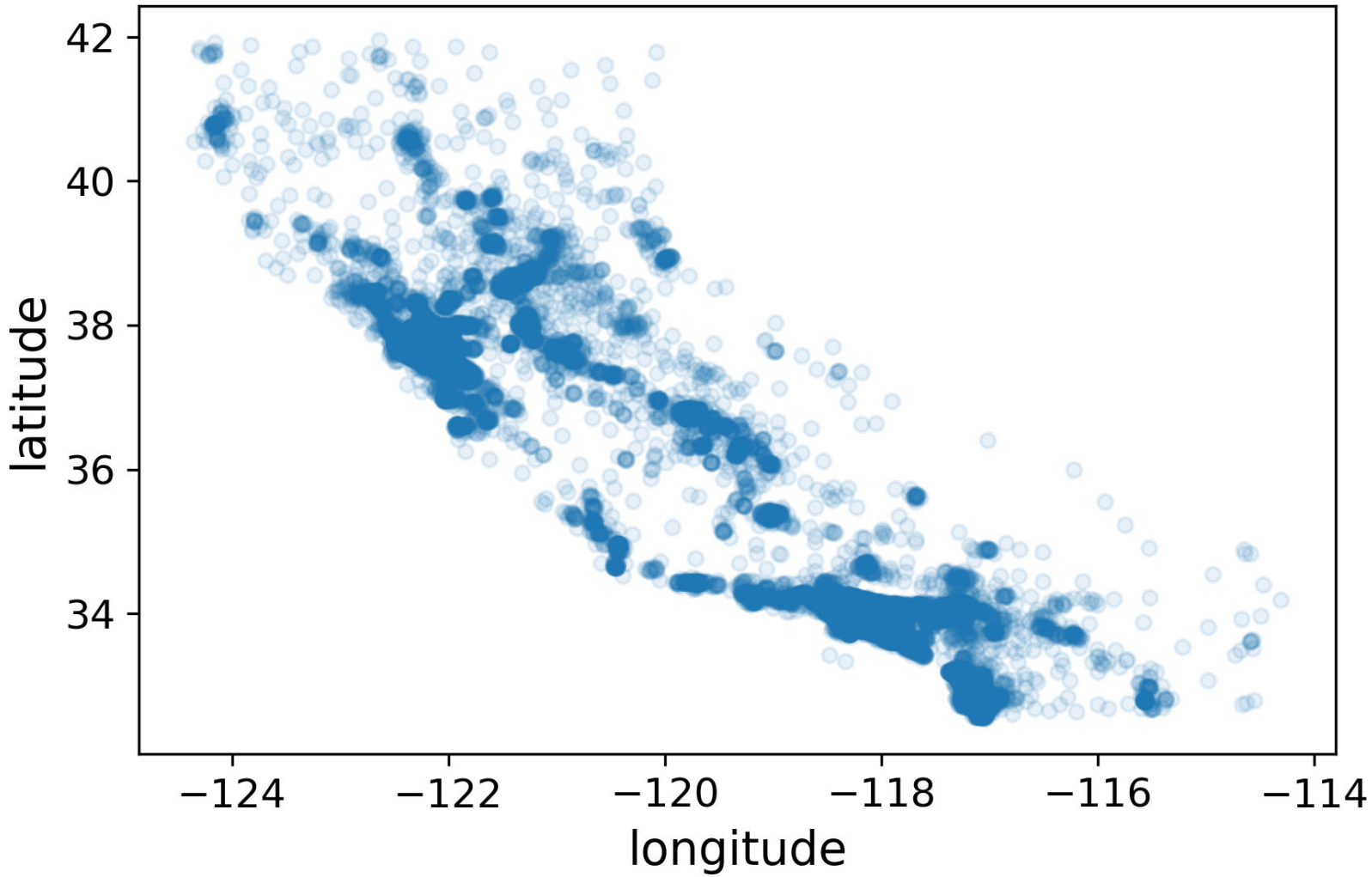
Visualize data I

- The Pandas library includes methods that call the Matplotlib API
- So that it's possible to plot very easily directly from `DataFrame` methods
- For example, you can very quickly plot geographical data using a scatter plot:

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

  - The `alpha` argument sets the points' transparency
  - Highlight places of a higher density of data points

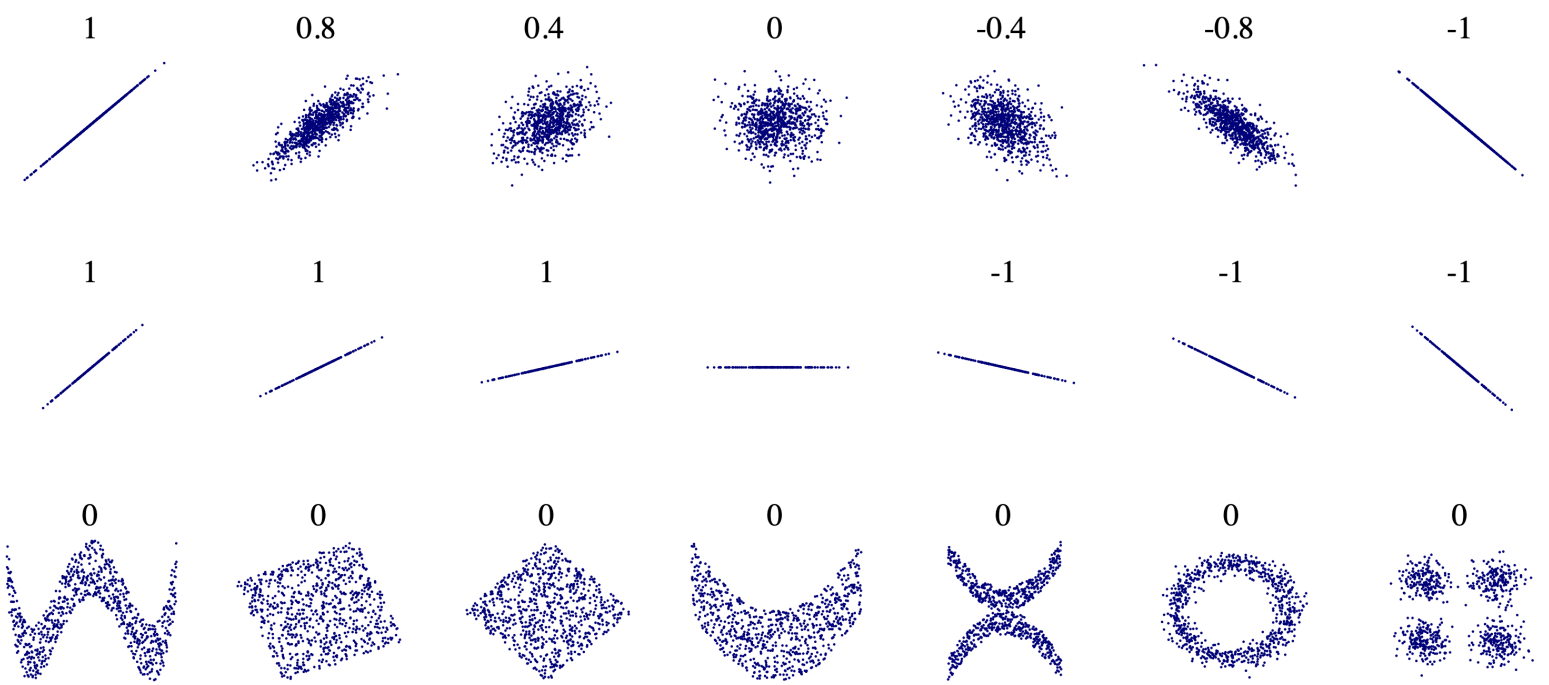
Visualize data† II



†Fig. from Géron (2019)

Looking for correlations

- If the dataset is not too large, you can compute the standard correlation coefficient (aka *Pearson's r*) between every pair of attributes using the `df.corr()` method
- It outputs a correlation matrix
  (to plot: `pd.plotting.scatter_matrix`)



Linear correlations

- Correlation coefficient only measures **linear correlations** (*if x goes up, then y generally goes up/down*)
- It may completely miss out on nonlinear relationships (e.g., *if x is close to 0 then y generally goes up*)
- On the previous slide's Fig., all bottom row plots present a **correlation coefficient of 0** although their axes are clearly **not independent**
- These are examples of **nonlinear** relationships
- The 2nd row shows examples where the correlation coefficient is equal to 1 or –1

  This has **nothing to do with the slope**

  - E.g., your height in meters has a correlation coefficient of 1 with your height expressed either in feet or millimeters

Checklist 4: Prepare data

- Work on copies of the data (keep original datasets intact)
- Write functions for all data transformations you apply:
  - To ease preprocessing for future dataset
    - Test set or new data instances for live systems
    - In future projects
  - To allow for treating these choices as hyperparameters

1. Data cleaning:
   - Fix or remove outliers (optional)
   - Fill in missing values or drop their rows (or columns)
2. Feature selection (optional):
   - Drop non relevant attributes for the task
3. Feature engineering, where appropriate:
   - Possibly discretize continuous features
   - Decompose features (e.g., categorical, date/time)
   - Add promising feature transformations (e.g., $\log(x)$, x^2)
   - Aggregate features into promising new features
4. Feature scaling: Standardize or normalize features

Data cleaning

- Most ML algorithms cannot work with missing features
- So you need to take care of them, as mentioned in the introductory class, 3 main options are available:
  - Discard the attribute altogether
  - Ignore these instances
  - Fill in the missing values
    (e.g., with 0, the mean, the median)
- If you choose the last option, Scikit-Learn provides a class to take care of missing values: `SimpleImputer`

Data cleaning with Scikit-Learn I

- First, you need to create a `SimpleImputer` instance, where the strategy to replace the missing value needs to be chosen

```
from sklearn.impute import SimpleImputer
imputer = SimpleImputer(strategy="median")
```

- Since the median can only be computed on numerical attributes, we need to create a copy of the data without any categorical values (e.g., text attribute)
- Now you can fit the `imputer` instance to the training data using the `fit()` method:

```
imputer.fit(df_num)
```

- The `imputer` has simply computed the median of each attribute and stored the result in its `statistics_` instance

Data cleaning with Scikit-Learn I

- Now you can use this *trained* `imputer` to transform the training set by replacing missing values with the learned medians

```
X = imputer.transform(df_num)
```

- The result is a plain NumPy array containing the transformed features
- To put it back into a Pandas DataFrame, you can simply use:

```
dt_transformed = pd.DataFrame(X, columns=df_num.columns)
```

Handling categorical attributes I

- Earlier we left out the categorical attribute since it is a text attribute, which has no median value
- Most ML algorithms prefer to work with numbers anyway, thus these categories should be converted to numbers
- For this, we can use Scikit-Learn's `OrdinalEncoder` class:

```
from sklearn.preprocessing import OrdinalEncoder
ordinal_encoder = OrdinalEncoder()
df_cat_encoded = ordinal_encoder.fit_transform(df_cat)
```

- You can get the list of categories using the `categories_` instance variable
- It is a list containing a 1-D array of categories for each attribute

Handling categorical attributes II

- An issue with this representation is that ML algorithms will assume that 2 nearby values are more similar than 2 distant ones
- This may be fine in some cases (e.g., for ordered categories such as "bad", "average", "good") but not for all cases
- To fix this issue, a common solution is to create 1 binary attribute per category
- This is called one-hot encoding because only a single attribute will be equal to 1 (hot), while the others will be 0 (cold)
- The new attributes are sometimes called dummy attributes
- Scikit-Learn provides an `OneHotEncoder` class to convert categorical values into one-hot vectors

```
from sklearn.preprocessing import OneHotEncoder
cat_encoder = OneHotEncoder()
df_cat_1hot = cat_encoder.fit_transform(df_cat)
```

Feature scaling

- One of the most important transformations you need to apply to your data is feature scaling
- With few exceptions, ML algorithms don't perform well when numerical attributes have very different scales
- There are 2 common ways to get all attributes to have the same scale: **min-max scaling** and **standardization**
- **Min-max scaling** (aka **normalization**) is quite simple, Values are shifted and scaled to fit the range $[0, 1]$:

$$x' = \frac{x - \min(x)}{\max(x) - \min(x)}$$

- Scikit-Learn provides the transformer `MinMaxScaler`

Feature scaling: Standardization

- **Standardization** first subtracts the **mean** value (standardized values always have a 0 mean) and then divides by the **standard deviation** to obtain a distribution with unit variance

$$x' = \frac{x - \bar{x}}{\sigma}$$

- Unlike min-max scaling, standardization does not bound values to a specific range, which may be a problem for some algorithms (e.g., neural networks often expect an input value ranging from 0 to 1)
- However, standardization is much less affected by outliers
- Scikit-Learn provides a transformer called `StandardScaler` for standardization

Transformation pipelines I

- When many data transformation steps need to be executed in the right order, they could be grouped together
- Scikit-Learn provides the `Pipeline` class to help with such sequences of transformations
- Here is a small pipeline example:

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

num_pipeline = Pipeline(
    [
        ("imputer", SimpleImputer(strategy="median")),
        ("scaler", StandardScaler()),
    ]
)
df_transformed = num_pipeline.fit_transform(df_num)
```

Transformation pipelines II

- The Pipeline constructor takes a list of **name-estimator pairs** to define a sequence of steps
  - All but the last estimator must be ***transformers***
    - I.e., must have a `fit_transform()` method
    - The last estimator may be a transformer, classifier, etc.
  - If names are not given, the `make_pipeline` method sets them automatically based on the estimator's class name
  - These names are used for the hyperparameter tuning step
- When you call the pipeline's `fit()` method:
  - All transformers sequentially call their `fit_transform()` method
  - The output of each call is passed as a parameter to the next pipeline estimator
  - At last, the final estimator calls the `fit()` method

Section 4

Model

Checklist 5: Select a model

Note:

- In case of large data, smaller training sets could be sampled
  - Allow training several models in shorter time
    (may penalize complex models, e.g., DNN, Random Forests)

1. Train many different model types using default parameters (e.g., linear, naive Bayes, SVM, Random Forests, NN)
2. Compare their performance
   - For each, use N-fold cross-validation and compute the mean and standard deviation of the performance measures
3. Analyze the most significant variables for each algorithm
4. Analyze the types of errors the models make
   Ask: *What data would a human need to avoid them?*
5. Have a quick round of feature selection and engineering
6. Have a couple more quick iterations of the 5 previous steps
7. Short-list the top 3 to 5 most promising models, preferring models that make different types of errors

Training and evaluating on the training set

At last!

- After framing the problem
- Exploring the data
- Sampling a training set and a test set
- Implementing a transformation pipeline to clean up and prepare data
- Now it's time to select and train an ML model

Note: The hard part is behind

Improve evaluation through cross-validation

- Use the `train_test_split` function to create a validation set out of the training set
- Or use the N-fold cross-validation tool

```
from sklearn.model_selection import cross_val_score
scores = cross_val_score(
    tree_reg_model,
    housing_prepared,
    housing_labels,
    scoring="neg_mean_squared_error",
    cv=10,
)
tree_rmse_scores = np.sqrt(-scores)
```

⚠ Keep traces of all experiments

- Save every model you experiment with
- Make sure you save **both** the **hyperparameters** and the **trained parameters**, as well as the **cross-validation scores** and the actual **predictions**
  - Compare scores across model types
  - Compare the types of errors they make
- Use Python's pickle module or the Joblib module (which is more efficient at serializing large NumPy arrays)

```
from sklearn.externals import joblib
joblib.dump(my_model, "my_model.pkl")

# And later:
my_model_loaded = joblib.load("my_model.pkl")
```

Checklist 6: Fine-tuning

Note: Largest data samples are needed for this step

1. Fine-tune the hyperparameters using cross-validation
   - Treat your data transformation choices as hyperparameters, especially when you are unsure of their impact (e.g., handling missing values through different strategies)
   - Unless there are very few hyperparameter values to explore, favor random search over grid search
     - If training time is large: Choose a Bayesian optimization approach (e.g., using Gaussian process priors)
2. Try Ensemble methods
   - Combining (best) models often perform better
3. Once confident with your final model: Estimate the generalization error (its performance) on the test set
   - ⚠ Don't tweak your model after measuring the generalization error (overfitting the test set)

Grid search I

- Let's assume that you now have a shortlist of promising models
- You now need to fine-tune them
- Grid search is a systematic way of changing 1 parameter at a time across all combinations of selected parameter values
- Use Scikit-Learn's `GridSearchCV`
  - Tell which hyperparameters and what values you want to experiment
  - It will evaluate all the possible combinations of hyperparameter values, using cross-validation

Grid Search II


```
from sklearn.model_selection import GridSearchCV
param_grid = [
    {"n_estimators": [3, 10, 30], "max_fts": [4, 8]},
    {"bootstrap": [False], "n_estimators": [3, 30]},
]
forest_reg = RandomForestRegressor()
grid_search = GridSearchCV(
    forest_reg_model,
    param_grid,
    cv=5,
    scoring="neg_mean_squared_error",
    return_train_score=True,
)
grid_search.fit(housing_prepared, housing_labels)
```

Grid search III

- A typical approach for a wide range of values is to use consecutive powers of 10
- You can give multiple grids as a list of dictionaries
- In this example, the 1^{st} dictionary will explore 3×2 combinations and the 2^{nd}, 1×2, thus $6 + 2$ in total
- On 5-fold cross-validation ($8 \times 5 = 40$ rounds of training)
- The `grid_search` object provides the hyperparameter values yielding the best result

```
>>> grid_search.best_params_
{"max_features": 8, "n_estimators": 30}
```

⚠ Since 8 and 30 are the maximum values that were evaluated, you should try searching again with higher values since performances may still improve

Randomized search

- Grid search approach is fine to explore few combinations
- Otherwise, `RandomizedSearchCV` may be preferable
- It is used similarly to `GridSearchCV`
- But instead of trying out all possible combinations, either a distribution over possible values or a list of discrete choices (which will be sampled uniformly) can be specified

```
param_grid = [{"gamma": scipy.stats.expon(scale=.1),
               "class_weight": ["balanced", None]}]
```

- This approach has 2 main benefits:
  - A budget can be chosen independent of the number of parameters and possible values
  - Adding parameters that do not influence the performance does not decrease efficiency

Ensemble methods

- Another fine-tuning operation is to experiment with different combinations of best performing models
- The *ensemble* will often perform better than the best individual model
    - Just like Random Forests perform better than the individual Decision Tree model
- The effect is even more pronounced if the individual models produce very different types of errors

Errors analysis I

- Some models offer the possibility to inspect their behaviors
- E.g., the `RandomForestRegressor` can indicate the relative importance of each attribute for making accurate predictions:

```
>>> feature_importances = (
    grid_search.best_estimator_.feature_importances_
)
>>> feature_importances

array([7.33e-02, 6.29e-02, 4.11e-02, 1.47e-02,
       ...
       1.65e-01, 6.03e-05, 1.96e-03, 2.86e-03])
```

Errors analysis II

- Knowing this information, some of the less useful features may be dropped
- You should also look at the specific errors that your system makes
- Try then to understand their root cause and how the system could be improved
  - Adding extra features
  - Or (on the contrary) getting rid of uninformative features
  - Cleaning up outliers
  - ...

Section 5

Bibliography

Bibliography

Géron, Aurélien. 2019. *Hands-on Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. 2nd ed. O'Reilly Media, Inc.