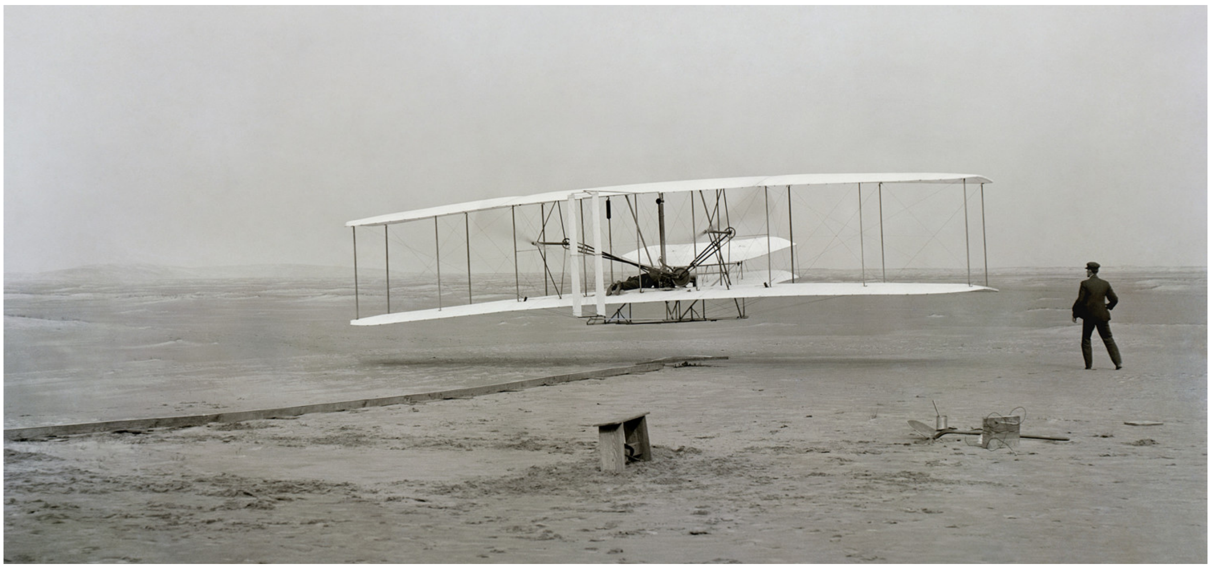

Day 1 - Guidelines

Forewords

Today's main goal is to start building a real software project, based on the messy notebook in which we did our R&D work.

We want to work step-by-step, and make things as clean as possible.

✔️ Prerequisites

- Clone the Git repo on your machine
- Create a `day_1` directory at the root of your local repo. All your work from today's session will be saved in this directory.

💡 General tips

- Every exercise has a suggested duration. Use them as an indicator as to when you should move on to the next one.
- All code snippets provided in the exercises guidelines are **nothing more than suggestions.**
- Talk with your teammates during the workshops.

- You can quickly re-write some code in the notebook if it helps you be more comfortable. Keep in mind that **this notebook is just used as a disposable source of knowledge**. The real deliverables are the Python modules.
- Use an IPython shell or a notebook to debug your work "live". You can use the autoreload extension. The following lines will automatically reload all the Python modules before executing a cell, which is very convenient during development, for debugging:

  ```
  %load_ext autoreload
  %autoreload 2
  ```

---

0. Creating a Python project

`Duration: 10 min`

All the following files and directories must be created **inside** the `day_1` directory. At the end of this exercise, your file tree must look something like this:

- ▼ `day_1` *toggle to see the tree structure.*
  - ▼ `data`
    - ▼ `raw`
      - `products.csv`
      - `stores.csv`
      - `transactions.csv`
  - ▼ `src`
    - `__init__.py`
    - `__main__.py`
  - `README.md`
  - `requirements.txt`
  - `setup.py`

Detailed steps

- Copy the 3 CSV files into a `data/raw` directory

- Create a `src` directory, holding two empty files: `__init__.py` and `__main__.py`

- Create a `README.md` file, with the following content

  ```
  # ACME - Machine Learning for retail sales forecasting

  A Machine Learning project to predict future sales at ACME Inc.

  This project uses Python > 3.7
  ```

- Create a `requirements.txt` listing all project's dependencies. **Always be explicit** about versions!

  ```
  numpy==1.17.2
  pandas==1.2.3
  scikit-learn==0.24.1
  ```

- Create a `setup.py` file, so your project can be easily installed and imported as a package

  ```
  from setuptools import setup

  with open("./README.md") as fp:
      long_description = fp.read()

  with open("./requirements.txt") as fp:
      dependencies = [line.strip() for line in fp.readlines()]

  setup(
      name="ACME Sales Forecast",
      version="0.1",
      description="Machine Learning for sales forecasting at ACME Inc.",
      long_description=long_description,
      author="YOUR NAME",
      author_email="YOUR EMAIL",
  ```

```
    packages=["src"],
    install_requires=dependencies,
)
```

- Install your package by running the following command (don't forget the `.` at the end)

  ```
  pip install -e .
  ```

  The `-e` flag indicates that we want an *editable* installation. All future updates in `src` will be automatically accessible.

  The `.` indicates that we want to use the package described in the current directory.

⏱️ Now seems like a good time to `git commit` your work.

---

1. Project configuration

`Duration: 15-20 min`

1.1 Create a `config` object

Holding all the configuration within a single object is a simple way to avoid code duplication, and to ensure that static configuration data ban be easily accessed from anywhere.

The `config` object must have the following attributes:

- `root_dir` : a `Path` object pointing to the `day_1` directory
- `package_dir` points to `src`
- `data_dir` points to `data`
- `raw_data_dir` points to `data/raw`
- `target` a constant string, holding our regression target column name

```
# src/config.py
from pathlib import Path

class _Config:

    def __init__(self):
        self.root_dir = ...
  # TODO

config = _Config()
```

Do you see something missing? Add anything that feels like configuration data to you.

Before proceeding, check that your `config` object can be imported and everything works as expected:

```
# src/__main__.py
from src.config import config

if __name__ == '__main__':
    print(config.root_dir)  # just a sanity check
```

Then, in your terminal, run the following:

```
python -m src
```

and check that the output is what is expected.

1.2 Create a `constants.py` module

The notebook uses repeatedly the same values... It's a bad practice, as it makes our code harder to refactor. A good idea would be to store all the constants as real Python constants.

Create a `constants.py` module to hold all constants used throughout the code (*eg* `year`, `product_id` etc...)

```
# src/constants.py
YEAR = 'year'
# your other constants go here
# ...
```

Some constants, such as model parameters or features parameters are probably better stored within the `config` object... Use your brain, and update your config if it feels better 😏

⏱️ A good milestone was just reached... Perfect timing for a new `git commit`

---

2. A basic logging config

`Duration: 5-10 min`

Well formatted log messages will help you debugging your code.

For now, we simply want our logs to be printed in the terminal, which is the default behaviour. Implement a basic logging configuration so you can use loggers in your app.

```
# src/__init__.py
import logging
```

```
FORMAT = '[%(asctime)s | %(levelname)s]\t%(message)s'

# TODO - THIS IS JUST ONE LINE OF CODE!
#
# Implement a basic logging config using the given formatter.
# The log level must be set to 'DEBUG'.
```

💡 *We want to be sure that Python parses our logging config* ***before anything else****. This is the reason why we store the code in `__init__.py`. Init files are always parsed by the interpreter when importing or running any object from the package (eg, `src`)*

Before proceeding, check that your logging configuration works as expected:

```
# src/__main__.py
import logging

logger = logging.getLogger(__name__)

if __name__ == '__main__':
  logger.debug("I'm testing the logging configuration.")
```

If you run `python -m src` in a terminal, you should see your `DEBUG` log. If it's not the case, fix your config, then move to the next exercise.

⏱️ It's `git commit` O'clock.

---

3. Data and I/O

`Duration: 30 min`

💡 *You just created a `config` object, as well as several constants...* ***Use them!*** *You also created a logger config...* ***Use it!***

You will need them both.

3.1 Creating a data catalog

Create `src/io.py` file and write a `get_data_catalog` function.

```
# src/io.py
def get_data_catalog():
  # TODO: return a dictionary holding two keys
  #     'products': a pandas DataFrame holding the data from products.csv
  #     'transactions': same, from the file transactions.csv
  pass
```

⚠️ **WARNING**

If you are a PyCharm user, there is a known (and stupid!) bug preventing you to correctly use any file called `io.py` during debugging. It can be solved following these guidelines, or by simply renaming your Python module differently...

3.2 Data preparation

Create a `data.py` file. This module must hold several functions for data preparation.

⚠️ This module is all about data preparation, **not feature engineering.**

Here are some suggestions... **Do not hesitate to challenge the choices we've made**, and implement your own functions if you think you've come up with something better.

```
# src/data.py
import logging

from src.io import get_data_catalog
...

logger = logging.getLogger(__name__)

def _get_daily_transactions(transactions):
  # TODO: return a dataframe holding aggregated data.
  # We want the total sale volume for a given product
  # on a given day.
  pass

def _get_weekly_transactions(daily_transactions):
  # TODO: same as daily, except that we want aggregated
  # data per week, instead of day.
  pass

def _merge_transactions_with_products(weekly_transactions, products):
  # TODO: Merge the 2 dataframes on `product_id`.
  # We want the same results as in the notebook.
  pass

def build_dataset():
  # TODO: Return the merged dataframe
  # This function is the "orchestrator"
  catalog = get_data_calalog()
  ...
  daily_tx = _get_daily_transactions...
  weekly_tx = _get_weekly_transactions...
  dataset = _merge_transactions_with_products...
  ...
  return dataset
```

Don't forget to write docstrings!

3.3 Saving the dataset

As its name suggests, your `io` module must hold both input and output logic... Hence, we will implement our `save_dataset` function in this module.

```
# src/io.py
def save_dataset(???):  # what should be the function signature? 🤔
    # TODO: store the dataset as a CSV file
    # The file must be stored in data/intermediate/dataset.csv
    pass

def get_data_catalog():
    # This function was implemented in 3.1, remember? 😏
    pass
```

3.4 A simple data preparation pipeline

Create a `run_dataset.py` file. This module will act as a "runner" for our small data pipeline. Its only goal is to generate the dataset, and store it in its expected location. **And of course, it uses loggers to track the processing progression.**

```
# src/run_dataset.py
from src.data import build_dataset
from src.io import save_dataset

def main():
    # TODO: Generate the dataset, and store it on disk as a CSV file
    # Don't hard-code the file path in here... Use your `config` object!
    pass
```

3.5 Making the pipeline executable from command line

Update `src/__main__.py` so the execution of `python -m src` automatically generate the dataset, and store it on disk.

```
# src/__main__.py

# TODO: run the data preparation pipeline on invocation
```

You should see some logs popping up on your screen, so you can follow that the processing evolves as expected. When it's done, you should have a new file stored in `data/intermediate/dataset.csv`

⏰ Yes, you guessed it... A `git commit` would be welcome before going further.

---

4. Build feature set

`Duration: 30 min`

The notebook cell that implements the feature engineering is arguably a big mess!

Create a `src/features.py` module. The only requirement is that it **must** contain a `build_feature_set` function that takes the dataset as input (a `pd.DataFrame`) and returns another dataframe, enriched with all the features.

```
# src/features.py

def build_feature_set(data):
    # TODO: re-write a clean feature engineering process here.
    pass

# NOTE
# You may feel the need to implement helper functions...
# This is probably a good idea, please do!
```

```
def _my_helper_function1(...):
    ...
```

⏱️ You probably have already committed your work by now... This is just a friendly reminder.

---

5. Train model

`Duration: 30-40 min`

In order to train a model, we need 2 ingredients:

1. A **dataset**
2. A **model**

We are going to create them both in this exercise.

5.1 Loading the dataset

In 3.4, you implemented a pipeline to create and save the dataset. Here, we will load this dataset, previously saved by running the `run_dataset.py` pipeline.

Update `src/io.py` to create a `load_dataset` function. This function must return the dataset as a `pandas` dataframe.

If necessary, this `load_dataset` function can also be use to parse the date, and ensure that the returned dataframe is ready to be processed.

We'll have to manually delete and regenerate the dataset when we want to refresh it... Building automation scripts for this kind of tasks is beyond the scope of the training, but you probably have all the knowledge required to write one yourself.

There are also existing tools, like Make.

5.2 Instantiating a model

Create a `src/models.py` file, holding 2 objects:

- A `_MODELS` or `_MODELS_REGISTRY` global variable: a dictionary that acts as a registry for all the available models (as of today, we only have a single `RandomForest` model). **As its name suggests, this dictionary will only be accessible from the `models.py` module.**
- A `get_model` function: this function will take the `config` object as its unique argument. Based on this config, it should return the expected model from the `_MODELS` registry. **This function is the public interface of our module.**

Even though we only have a single `RandomForestRegressor` today, it's highly probable that we will want to try different models in the future...

Defining a `get_model` function is a nice way to offer a durable interface. When we choose to add other models, we'll be able to use them without the need to refactor the callers. A simple config update will be enough.

You will also need to update your `config` object: it must now hold attributes related to the model (at the very least, its name or type, and the associated hyper-parameters).

```
# src/models.py
import ... # ?

_MODELS = ... # TODO

def get_model(config):
    # TODO: query the model registry depending on the `config` parameter.
    # Instantiate a model object, with the hyperparameters declared
    # in the config. Then, return the model object.
    pass
```

5.3 Implement training logic

We now have everything we need to create a training pipeline.

The main component will be a `train()` function implemented in `src/train.py`. Some helper functions will be required, to avoid overloading the `train()` logic. At least:

- `train_test_split()` to split your dataset at a given date. This is a much better approach than relying on `sklearn` which may shuffle our time-series data, or make the cut in the middle of a period (*ie* in the middle of a week)
- `evaluate_model()` to assess the performance on our fitted model (we'll keep `r2` and `WAPE`)
- *+ any other helper function you might find useful*

The core logic of `train()` is to:

1. Split the dataset into a training and a testing set (using your `train_test_split` helper function)
2. Build the feature set on each split (using the `build_feature_set` function you wrote earlier)
3. Split the target from the inputs on each split
4. Train the model with the training data
5. Evaluate the model performance (using your `evaluate_model` helper function)
6. Return the fitted model, along with its evaluation metrics. The returned value should be a dictionary with two keys:
    - `model` - the fitted model

- `metrics` - the evaluation metrics for both `train` and `test` sets.

```
# src/train.py

def train(model, dataset):
    # TODO: implement the training logic, as drafted in the notebook.
    # Don't forget to use some logging to ease your debugging process!
    # It may be a good idea to implement helper functions...
    pass
```

💡 Don't forget to use a logger to track the progression!

5.4 Creating a training pipeline

As we did for the dataset creation, we will now create a pipeline for the training step. We want an executable script `src/run_train.py` that can be invoked by a shell command. This script will simply:

1. Load the model
2. Load the dataset
3. Run the training process

🌟 **Bonus**

You can time the training process, and log some information about execution duration.

```
# src/run_train.py

def main():
    # TODO: run your train function
    # Make this module executable as a script.
```

Check that it runs correctly by running `python -m src.run_train` in a terminal. You should see your logs in the console.

⏱️ What about a quick `git commit` now?

---

6. Implement a basic CLI

`Duration: 20-30 min`

Python ships with a builtin module to implement a simple Command Line Interfaces: `argparse`.

During the previous exercices, we created 2 pipelines:

- `run_dataset` : to create a dataset based on the raw data
- `run_train`: to train a model on the dataset, and extract some evaluation metrics

The goal of this final exercise is use `argparse` to build a CLI so we can easily run those 2 pipelines from a terminal.

The final result must be something similar to this:

```
# From your terminal
$ python -m src run_dataset
[... some logs about data processing...]

$ python -m src run_train
[ ... some logs about training a model...]

$ _
```

6.1 Creating the CLI

Create a `src/cli.py` module to implement the CLI logic. Don't forget to check the slides and the official `argparse` docs.

6.2 Invoking the CLI

To make it executable by invoking `src` directly, you must import the CLI logic in `src/__main__.py`.

Run both `run_dataset` and `run_train` commands in your terminal to check if everything goes as expected.

⏰ And now it's time for the last `git commit` of the day.

That's it for today, congratulations!