Day 4 - Guidelines

> Today, it's all about tooling for Features Engineering...

This is mainly a pretext to make you practice with Advanced Python on interesting topics!

✔️ Prerequisites

- Be sure that the `day_3` directory is complete (you can copy-paste from our yesterday's suggestions if necessary)

- **Copy all the content of `day_3` directory to a new `day_4` directory: today, we will work exclusively in `day_4`**

- Update your package installation by running `pip install -e .` **from the `day_4` directory.**

💡 General tips 🚨

- Some exercises, like #2 and #4 are quite challenging... You **absolutely need to discuss with your teammates** to make good progress.

---

1. Historical features

`Duration: 15-20 min`

If you remember, at the end of Day 2, we had to remove some of our features because they needed the target to be computed. At the time, we did not have a clean way to add/remove them depending on the execution context.

With today's feature engineering pipeline, we should be able to use them again!

Before anything, we need to create "atomic" functions that could then be used as tasks in our graph.

1.1 Create the base functions

In `training/data.py` implement 2 functions:

- `_lagged()` to compute the offset
- `_moving_average()` to compute the average on the 3 previous weeks

```
# src/training/data.py

...

def _lagged(data, *, lag):
    # TODO: return a pd.Series with lagged data,
    # based on lag's value

def _moving_average(data, *, window):
    # TODO: return a pd.Series with moving average,
    # based on window
```

1.2 Create the historical function

In the same module, create a `_compute_historical` function whose only job is to take the `transactions` dataset as input, and to build a dataframe containing 9 columns:

- `product_id`
- `nb_sold_pieces` (the target)
- the period (`period` column from our intermediate dataset)
- the 3 lagged columns (cf. config)
- the 3 moving average columns (cf. config)

We also need a function to store this data on disk.

```
# src/training/data.py

...

def _compute_historical(transactions):
    # TODO: return a pd.DataFrame containing 9 columns

def _save_historical(historical):
    # TODO: store this dataset in `data/intermediate/historical.csv`
    # Remember to use your existing function for saving a dataset on disk!
```

1.3 Update the `build_dataset` pipeline

Now that we are able to build a second dataset only containing historical data, we can update our dataset pipeline so it can use these new tasks.

First, rename your existing `build_dataset` function to `build_datasets` (plural) as it will now build 2 datasets, instead of one. Adapt your code base to this minor refactoring,

or use the refactoring tools built-in with your favorite IDE.

Then, add 2 new edges to your existing pipeline:

```
# src/training/data.py

...

def build_datasets():
    pipeline = Graph()
    pipeline.add_edge(
        load_transactions,
        _compute_daily_transactions
    )
    pipeline.add_edge(
        _compute_daily_transactions,
        _compute_weekly_transactions
    )
    pipeline.add_edge(
        _compute_weekly_transactions,
        _merge_transactions_with_products
    )
    pipeline.add_edge(
        load_products,
        _merge_transactions_with_products
    )
    pipeline.add_edge(
        _merge_transactions_with_products,
        _save_dataset
    )
    # TODO: Add 2 new edges required to build the historical data
    # and save it on disk. Those edges must use the functions you
    # created in 1.2, and probably re-use some existing too...
    return pipeline
```

1.4 Run your new dataset pipeline

In your terminal, run `python -m src dataset` then check if the `historical.csv` data is stored in your intermediate data directory.

> Now, take a deep breath...

---

2. Build a features registry

`Duration: 45 min`

We are going to build a DAG to register the features we used in our modelization, as well as the dependencies between them.

Again, implementing a graph structure for such a simple example is overkill, but this a great pretext for us to dig deeper into Python possibilities.

2.1 Implement a new custom error

In `src/libs/features/exceptions.py`, create a new `FeatureNotFoundError`, derived for our base custom error. As its name suggests, this is the error the graph will raise in case a feature is being accessed before its registration.

NOTE

We don't create it in the base `src/exceptions.py` just to emphasize that today's project is not supposed to be implemented among business code. It's supposed to be an independent library.

```
# src/libs/features/exceptions.py
...

# TODO: implement a new custom error, with docstring.
```

2.2 Create the `FeatureRegistry` object

In a new module in `src/libs/features/registry.py` create a `FeatureRegistry` class.

> This class will implement another graph data structure, with important differences compared to yesterday's DAG.

First and foremost: it will **not** add edges via a dedicated method. We shall see in section 2.4 how we are supposed to add nodes and edges.

For now, just create 3 basics methods, and a property, according to the following code snippet:

```
# src/libs/features/registry.py

class FeatureRegistry:  # or FeatureStore... as you prefer!

    def __init__(self):
        self._registry = {}

    @property
    def registry(self):  # avoid state mutation from outside
        return self._registry

    @property
    def adjacency(self):
        # TODO 0. return a dictionary, mapping feature names to
        #          list of dependencies names.
        # Merry christmas. This one is a gift :)
        return {k: v.depends for k, v in self._registry.items()}

    def get_feature_dependencies(self, name):
```

```
        deps = self.adjacency.get(name)
        if deps is None:
            raise FeatureNotFoundError  # your custom error
        return deps

    def topo_sorted(self):
        # TODO 1. this is the same logic than yesterday.
        # Leverage NetworkX to do the heavy lifting!

    def get(self, name):
    # Features data is stored in `self._registry`
    # The key is the feature name, the value is the feature data.
    # (more details later. You don't need more to implement this
    # function.
    # TODO 2. return the feature data



    # TODO 3.
    # make instances of this class ITERABLE.
    # When we iterate over an instance, we must get
    # all nodes in topological order.
```

2.3 Create atomic features

So we can use them with the expected flexibility in our pipeline, our feature **must be atomic**.

For today's use case, a feature will be a single function, taking *at least* the intermediate dataset as first argument. Depending on the feature, it can also take other arguments.

NOTE

Every feature will receive the intermediate dataset as first argument, **even if it doesn't need it to build its output**. In such case, a good practice would be to call the first function parameter `_` (a simple underscore) to explicitly indicate to the reader that this argument will not be used.

This is a shortcut we took to simplify the creation process that seems totally acceptable for the purpose of the training.

```
# src/training/features.py

def product_id(data):
    # TODO: return the corresponding pd.Series

def period(data):
    # TODO: return the corresponding pd.Series

def week(_, period):  # dependency here!
    # TODO: return the corresponding pd.Series

def year(_, period):  # and another dependency
    # TODO: return the corresponding pd.Series
```

2.4 Improving the `FeatureRegistry` object

How can we automatically register our atomic features in our registry object?

Knowing that these atomic functions are not meant to be executed from anywhere else than from our graph executor...

> ... a decorator seems like the perfect solution for this use case.

By using the `@decorator` syntax, we can automatically register the features in the registry (or feature store), and still keep them as independent, simple callables (*ie*, functions). Simple and readable: just the way we love it 😍.

The final result will look like this (read carefully!):

```
# Just an example - *Do not implement it*

registry = FeatureRegistry()

@registry.register(name='product_id')
def product_id(data):
    ...

@registry.register(name='year', depends=['period'])
def year(_, period):
    ...

@registry.register(resources=['historical'])
def lagged_target_1W(data, historical):
    ...  # we'll do that one later on.
```

As you saw earlier, the registry holds its state in a `_registry` attribute. By reading the above example, we clearly need to implement a `.register()` method that will **take care of writing into the registry's inner state.**

This method must be used as a decorator... **However**, by nature, a bound method cannot take anything else than the instance as a first argument. Hence, this method, when called, must return a function that will *in fine* be the decorator.

> Does it sound obscure? **Good!** This is definitely some advanced Python we're doing here 🚀

💡 Some suggestions about the `register` method

- It takes 3 optional arguments:
  - `name`: the name of the feature. If not provided, default to the feature's function's name (*eg* `product_id`)
  - `depends`: the list of the feature names on which this feature depends. Default to an empty list.
  - `resources`: the list of external data that this feature uses. Default to an empty list.
- It returns a function. Otherwise, how could we use its return value as a decorator?
- The returned function must probably be a closure... Otherwise, how can it keep the state of the method when called (*ie*, name or dependencies)?

```
# src/libs/features/registry.py
from typing import Callable, List


@dataclass
class FeatureRecord:
    name: str
    func: Callable
    depends: List[str]  # list of required dependency names
    resources: List[str]  # same with resources


class FeatureRegistry:
    ...

    def register(  # TODO 1. : define the method's signature):

        def do_register(  # TODO 2. : define the closure's signature):
        # TODO 3. : build a record to store in `self._registry`

        # A record is an instance of the FeatureRecord dataclass.
        #  - name: the name of the feature
        #  - func: the callable function
        #  - depends: the list of required dependencies
        #  - resources: the list of required external resources

        # TODO 4. : store the record in the instance state,
        # ie, in `self._registry` (a dict)
        # Records must be accessible by their names

        return  # TODO 5. : what should the `register` method return?
```

2.5 Decorate the feature functions

Now that you're done with building the `register` method, you can decorate the 4 features functions you created earlier.

Reminder: this is how the decorator should be used on your atomic features:

```
# src/training/features.py
from src.features.registry import FeatureRegistry

registry = FeatureRegistry()

# Just an example.
# Do the same with the other features

@registry.register(name='year', depends=['period'])
def year(_, period):
    ...
```

When you're done, **open a Jupyter notebook or a Python shell**. Check that you get the following output:

```
# in a notebook or a Python shell

################################################################

# Just some util calls to instantiate our context
# Don't forget to run it every time you're starting
# a new Python interpreter!

from src.config.config import get_config
from src.context import context

# or 'dev', or whatev'... Adapt to your actual config file name
context.environment = 'development'
context.config = get_config(context)

################################################################

from src.training.features import registry

registry.adjacency
```

```
# must output the following dictionary:
{'product_id': [], 'period': [], 'week': ['period'], 'year': ['period']}
```

🚦 **Checkpoint**

If you don't have the previous output, go back to 2.4 🛂

If you do have the previous output, **congratulations**. Ping me on Slack, or ask me to join you in your room. Can you tell me why are the features successfully registered whereas we **did not call them?**

3. Some visualization

`Duration: 15 min`

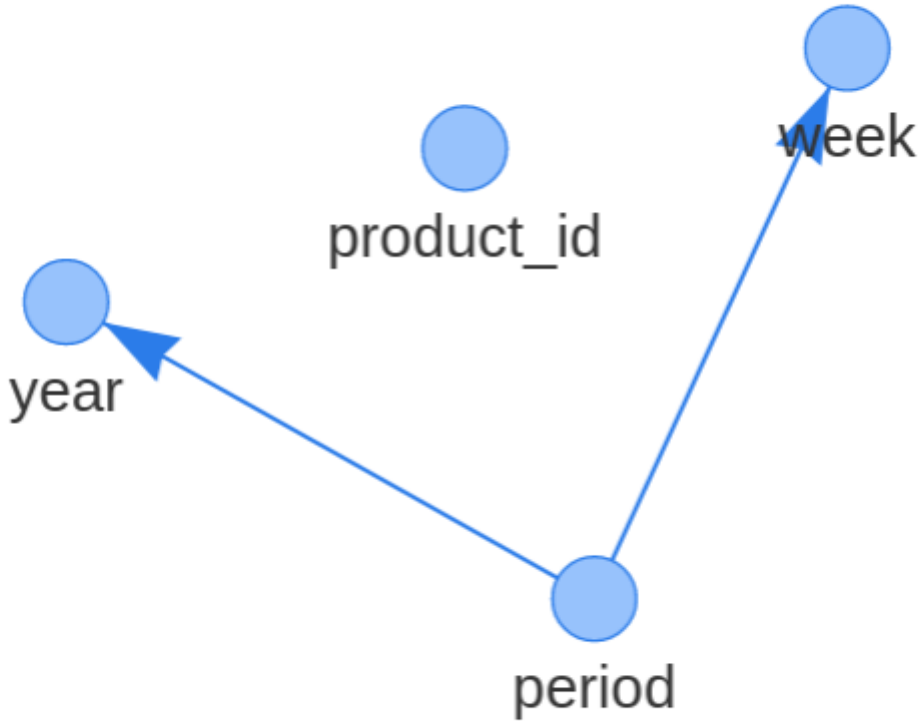
If you need an easy step before proceeding to the next exercise, check the visualization of your current feature store, or registry.

You can also skip this part if you prefer, and save your energy for the next exercise ⚡

```
# in a Jupyter notebook
######################################################################
%load_ext autoreload
%autoreload 2
######################################################################
%matplotlib inline
######################################################################

import matplotlib.pyplot as plt
import networkx as nx
from pyvis.network import Network

######################################################################
from src.config.config import get_config
from src.context import context
```

```
context.environment = 'development' # adapt to your config file names
context.config = get_config(context)

########################################################################

def draw_graph(adjacency, *, path=None, backend='pyvis', ax=None):
    nx_graph = nx.DiGraph(adjacency).reverse()

    if backend == 'matplotlib':
        if ax is None:
            ax = plt.axes()
        nx.draw_networkx(nx_graph, ax=ax)
        if path is not None:
            ax.figure.savefig(path)
        return ax

    elif backend == 'pyvis':
        assert path is not None, (
            "`draw_graph` needs a path if you're using pyvis backend."
        )
        net = Network(directed=True, notebook=True)
        net.from_nx(nx_graph)

        return net.show(str(path))
    else:
        raise NotImplementedError

########################################################################

from src.training.features import registry

########################################################################
draw_graph(registry.adjacency, path='graph.html')
```

The result must be similar to this:



4. Creating the executor

`Duration: 30 min`

Like yesterday, now that we have a graph, we need an object to execute its nodes. As this executor will be used to *generate* the features, we'll call this object `FeaturesGenerator`.

4.1 A simple utility function

The `itertools.chain` function is almost perfectly suited to our coming needs, but only almost...

We need to tweak it a little bit. Just copy/paste this code snippet.

```
# src/libs/features/utils.py
from types import GeneratorType

TYPES_TO_UNPACK = (tuple, list, GeneratorType)

def chain_from_iterables(iterables):
    """Like itertools.chain.from_iterable, but avoid unpacking np arrays."""
    for it in iterables:
        if isinstance(it, TYPES_TO_UNPACK):
            yield from it
        else:
            yield it
```

NOTE

Did you know about `yield from`? Now you do.

Defined by PEP 380

4.2 Executor implementation

The main behavior of this class will be implemented in a `.transform()` method: this method will sequentially transform all of our features, and store their output in the inner state of the generator.

About the transform method

As usual with a graph data structure, each feature must be generated only after all its dependencies have been themselves generated. The output of these dependencies will be used as the feature's input.

```
# src/libs/features/generator.py
from src.libs.features.utils import chain_from_iterables


class FeaturesGenerator:

    def __init__(self, registry, features=None, resources=None):
        self.registry = registry
        # Note: our naming is a bit confusing... There is room
        # for improvements... but later.

        self.features = features or list(registry.registry.keys())
        self.resources = resources or []
        # TODO 0. we miss something to keep track of the execution...
        #         Add it!

    def transform(self, data):
        for feature_name in self.registry:
            logger.debug(f"Generating feature '{feature_name}'...")

            record = self.registry.get(feature_name)

            deps =  # TODO 1. get the list of deps

            # unpack data from deps output. You're welcome!
            upstream_data = chain_from_iterables(
                self._state[d] for d in deps if d
            )

            output =  # TODO 2. run the transform process
            # on the feature, and fetch its output.

        # TODO 3. store the output in instance's state
        # It must be accessible by feature name.

        logger.debug("Feature generation done!")
        # TODO 4. return a pandas DataFrame whose columns are
        # the feature names, and values are the output of the
        # `transform` processing.
        return ...
```

4.3 Check it out!

Get back to your Python shell or notebook, and check the result.

```
# in a notebook or Python shell
```

```
from src.training.features import registry
# import anything you need

generator = FeaturesGenerator(registry=registry)

data = load_dataset(context.dirs.intermediate / 'dataset.csv')

generator.transform(data)
# and you should see your logs!
```

5. Adding resources

`Duration: 15-20 min`

At the very beginning of the workshop, we created an external dataset, called `historical`. This dataset can be used to build our features in some conditions, *eg* during training when we have access to the target.

Our new feature engineering system enables us to branch some feature so they can make use of these external resources.

This exercises is more guided: the goal is to show you what can be done, and maybe give you some feeling about what is missing, what could be improved, etc...

5.1 Create a loader function

We stored the historical dataset on disk, we now need a way to load it. You did it hundreds of times in the last few days, no surprise here!

```
# src/training/train.py
from src.io import load_dataset as _load_dataset

...

def load_historical_dataset():
    path = context.dirs.intermediate_data / 'historical.csv'
    raw_data = _load_dataset(path)
    data = raw_data.assign(**{
        PERIOD: lambda x: (
```

```
            pd.to_datetime(x[PERIOD].str.split('/').str[0])
            .dt.to_period('W')
        )
    })
    return data
```

5.2 Update the `transform` method

It must be able to deal with external data, or resources. It's alreay doable to instantiate a generator with a list of resources (cf. the `__init__` method) but we are not using it for the moment.

```
# src/libs/features/generator.py
from src.libs.features.utils import chain_from_iterables

class FeaturesGenerator:
    ...
    def transform(self, data):
        for feature_name in self.registry:
            ...
            deps = ... # same
            upstream_data = ... # same
            resources_data = (self.resources[r] for r in record.resources)

            output = # TODO: update, and pass the resources_data as well
            # the rest does not change
```

5.3 Create a feature with external resources

As an example, we'll use the `lagged_target_1W` but the process is the same for all the feature depending on external resources. As of now, we only have one external resource: the `historical` dataset, built in exercise #1.

```
# src/training/features.py
...

@registry.register(resources=['historical'])
```

```
def lagged_target_1W(data, historical):
    return pd.merge(
        data,
        historical[[PERIOD, PRODUCT_ID, LAG_TARGET_FEATURE.format(lag=1)]],
        on=[PERIOD, PRODUCT_ID],
        how='left'
    )[LAG_TARGET_FEATURE.format(lag=1)]
```

5.4 Final testing

And now, we can finally make use of our external historical data to build a feature set!

Open a notebook, a run the following experiment:

```
# Note: we use our constants here.

# We create a new generator, explicitely saying which features
# we want to extract... Any other features from the registry
# will be ignored (here, `period` is missing)
generator = FeaturesGenerator(
    features=[PRODUCT_ID, YEAR, WEEK, LAG_TARGET_FEATURE.format(lag=1)],
    registry=registry,
    resources={'historical': historical}
)

# Pretend to work only on a subset of the data
train_data = dataset.head()

generator.transform(train_data)
```

After some logging messages, you should see the following output:

	product_id	year	week	lag_target_1W
0	1	2016	22	123.0
1	1	2016	23	123.0
2	1	2016	24	151.0
3	1	2016	25	130.0
4	1	2016	26	116.0

What a ride... Congratulations!

Even though, as Data Scientists, your job is not to build such sophisticated engineering tools, knowing what they are, how they work, and how they can improve your workflow is very important.

Thank you 🌟