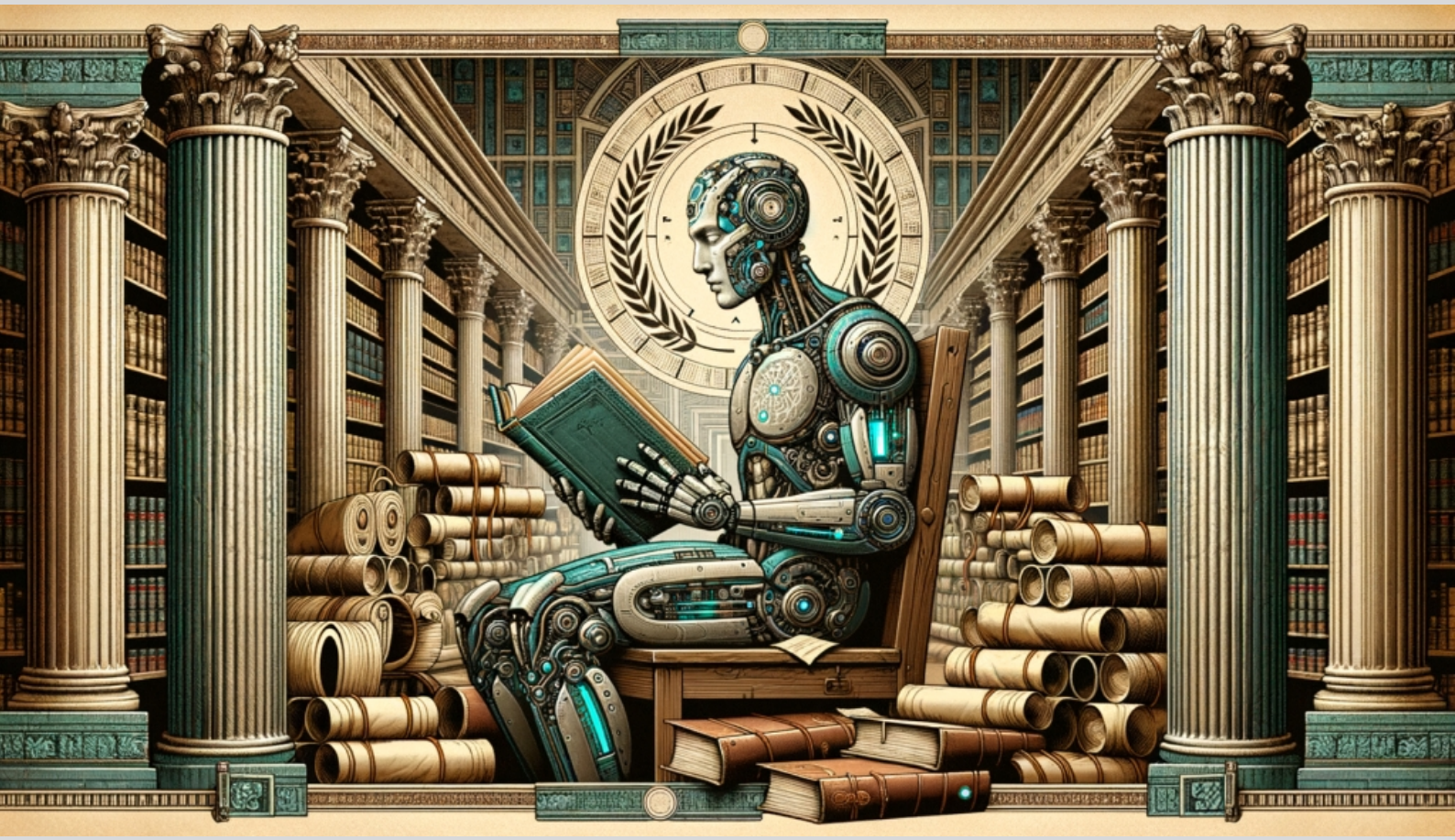


Smart Structuring

A Guide to Efficiently Managing Data Science Projects in Python

Tomasz Guściora

2024

Smart Structuring: A Guide to Efficiently Managing Data Science Projects in Python

Tomasz Guściora

June 2024

Images & book cover created by Tomasz Guściora with OpenAI DALL·E.
Text corrections by Tomasz Guściora with OpenAI ChatGPT.

Published by Tomasz Guściora
2024, Warsaw, Poland
ISBN: 978-83-972422-0-3

Tomasz Guściora

About the Author

With an analytical mind, a knack for writing, and a good sense of humor, I've spent over a decade in the data science field, leading projects across multiple continents and industries. In January 2024, I started DemystifAI.blog to document and share my learning experiences, aiming to make AI and automation more understandable and accessible.

Notable projects I've led include developing predictive models for banking and insurance, aiding in anomaly detection and streamlining daily processes. These experiences sparked my passion for integrating AI into daily life and leveraging technology for personal growth. If big organizations can harness AI for their benefit, why can't we?

You can explore my work and connect with me through the following platforms:

- Homepage: https://demystifAI.blog/
- LinkedIn: https://www.linkedin.com/in/tgusciora/
- Github: https://github.com/TGusciora
- Substack: https://demystifAI.substack.com
- Twitter/X: https://twitter.com/TomaszGusciora
- Contact me: tomasz@demystifAI.blog

Contents

Chapter 1: Preface

Since the beginning of ages, people have had problems with ordering files on their desktops. You could easily imagine an honest day of work in ancient Egypt, opening your *Pyramid_resources* folder and seeing there something like this:

- *limestone.jpg*
- *limestone_v1.jpg*
- *limestone_v4.jpg*
- *limestone_v4_but_better.jpg*
- *limestone_v3_sent_from_Anubis_Final.jpg*

What the hell are you supposed to do with this?! Which limestone should be used for today's construction batch? There is an enhanced version 4 at hand, but I am not sure if Anubis isn't someone who has the final say around here...

Imagine how much easier it would be to just have it formatted as *yyyymmdd_limestone.jpg*? For example:

- *20240401_limestone.jpg*
- *20240324_limestone.jpg*

Now, if you sort your files by name in descending order - you immediately know the last version you worked on.

You have multiple versions per day? Not a problem - add a *vxx* component for version. So together it will be *yyyymmdd_vxx_limestone.jpg*. Like:

- *20240410_v05_limestone.jpg*
- *20240410_v04_limestone.jpg*
- *(...)*
- *20240324_limestone.jpg*

Benefits?

1. Again, you can just find the current version by sorting files by their names.
2. There's no need to implement changes retroactively—all older items with only dates will still sort correctly.

Figure 1.1: "Listen, Garry, we have to talk about your last quarter's performance.". Robotic Anubis scolding somebody at the office. Image generated with DALL·E

Phew! That was a close one! Thankfully, you avoided a scolding.

Are you the only one who struggles with keeping order in their projects? Of course not! There are legions of us!

Even kind souls at prestigious universities like Harvard and the Massachusetts Institute of Technology (MIT) have faced similar challenges and have developed guidelines on how to organize your project files. I'll do my best to combine those with my data science experience and share a recipe for a project structure that features:

- usability - easy to understand

- maintainability - easy to change and update
- reusability - easy to use in other projects or applications
- portability - easy to shift to other systems
- modularity - can be easily decomposed into smaller, independent pieces

Although it's past Dry January, I will also cover the DRY (Don't Repeat Yourself) principle in programming.

All of the above should make your projects more standardized, easier to kick off, and over time - bring more joy to your work. I don't recommend reinventing the wheel each time; this allows you to save precious brain energy for thinking about things like flying saucers, a cure for cancer, or where you left your car keys.

There is a reason why Barack Obama, during his presidency, wore only gray or blue suits daily. It was one of his strategies to limit decision fatigue, saving energy for more important decisions. Be like Barack. Hop on! There's a lot to explain!

Figure 1.2: "Woof, woof!". Robo-dog inviting you for a ride. Image generated with DALL·E

P.S.: View project codes in Github repository. You can contact me at tomasz@demystifAI.blog.

Chapter 2: Law & Order - Files Episode

Below are some real-life potential problem scenarios that you might want to avoid (and don't worry - I am no stranger to these mortal sins. We are all humans, after all. The aim is to improve every day, not to be perfect):

You have a potential problem if you have your project files:

- among other files on your desktop "just to quickly check something" or "just temporarily"
- scattered among different folders/paths on your PC/server
- with "final" and "final_final" suffixes or prefixes
- for the same task without clear versioning (e.g., *clean_stuff.py*, *clean_stuff_but_better.py*)
- with confusing names (e.g., *credit_score_dont_be_angry_mom.py*)
- with spaces in path or name (e.g., */hey honey/why is this file not working.py*)
- with manually corrected raw data (I'll describe in more detail)

With time, I see this less and less, but here are a couple of examples from my career that were directly a result of suboptimal file management.

2.1 Example 1 - Sudden Migration

Once, I worked on re-estimating a propensity model, and during that time, the data scientist who had produced 90% of the code was moved to another project (and his access was restricted; yes, this can actually happen in real-life!).

Soon afterward, it turned out that the model's performance on the test group was not as good as expected by the client. I decided that we needed to understand more about the code that handled the target variable calculation, preliminary exploratory data analysis (EDA), and some cleaning.

It should have been easy, and I should have been able to simply run some code and

achieve the same result. However, due to a lack of clear documentation, disorganized files, and numerous hidden dependencies among them, it took me over a month to do so. Over a month! It might have been easier to just redo the modeling from scratch (or maybe not; I guess I'll never know.).

2.2 Example 2 - It Works, But I Don't Know Why

In a different case, I developed a very effective model, but one variable connected to the date of email address registration was critically important. Excluding this variable resulted in a 20% drop in model performance KPIs.

The issue arose when I discovered that the values of this variable in the source tables were completely different from those in the modeling sample. What happened?

After two months of investigation, we determined that due to human error, the variable was initially calculated as the number of days between email registration and a specific date, then divided by some constant.

It was later corrected to days between email registration and the date of the monthly data snapshot, but unfortunately, this update was not communicated to the data science team.

After the correction, it remained a strong predictor but only accounted for about 2% of the model performance KPI. Why it was such a strong predictor initially is still unclear to me; perhaps it was connected to a seasonal spike in the target variable.

One might wonder why we didn't maintain the incorrect logic if it was such a strong predictor. There are two main reasons:

1. It was a completely spurious relationship.
2. The specific date was in the past. With new data, the results would vary significantly, causing model scores to become unstable.

2.3 Example 3 - What's This Git Stuff Again?

Ah, the humble beginnings.

I was producing multiple versions of pipeline code for using a model in production. What do I mean by pipeline code? Since the model was for online scoring, it consisted of:

1. Extracting variables from online databases.
2. Transforming variables into the forms expected by the model.
3. Calling the scoring code to obtain the model score.
4. Offloading data about input variables, transformed variables, model scores, and possible errors to audit tables for future reference.

In this particular project, I did not have access to customer data due to the General Data Protection Regulation, California Consumer Privacy Act, Personal Information Protection and Electronic Documents Act, or the New Zealand Privacy Act 2020 - whichever applies in your country, dear Reader. Therefore, each time I finished the code, I sent it to my business counterpart to test it on "true" data.

Why didn't I have access to anonymized data? Well, sometimes system designs have flaws in the real world. If this is your scenario, do all within your influence to secure a steady flow of anonymized data or fix whatever issue you are facing.

Each time I did this, I incremented the version of the code to something like "M5_scoring_v1", "M5_scoring_v2", etc. (I know, I know...)

Long story short: I had so many versions of pipelines that on multiple occasions, I opened the wrong code and started working on an old version. This obviously wasn't a showcase of productivity mastery.

I did one more thing to supposedly make it "easier" for everyone. I started to add comments in the code in sections that I updated, like:

```
# 2024-04-20 L54 - changed missing data to 0
(...happy chunks of code...)
```

```
# 2024-04-21 L56 - added transformed variable
(...sad chunks of code...)
```

In the next version of the file, I either removed old comments and inserted new ones, or I didn't, leaving a lot of unnecessary comments referring to some Excel files (LXX was a remark number from an Excel file regarding testing that model) from the past.

It was good, hard and completely unnecessary work.

What I didn't realize back then is that there are possibly infinite tools for text file comparison, such as:

- notepad++ with the Compare plugin,
- Powershell (Windows)

```
compare-object (get-content new1.txt) (get-content new2.txt)
# or
diff (cat new1.txt) (cat new2.txt)
```

- Bash (Unix)

```
diff new1.txt new2.txt
```

- whole Git universe for version control (well, that actually I was aware of, we just didn't use git on the project. Yes, I know it's a sin.)

Each time I sent my business counterpart a changed file with a lot of overhead text, it just made him more frustrated.

Golden times (wiping away tears).

I don't even remember how long we played email ping-pong, but in the end, we were both very frustrated.

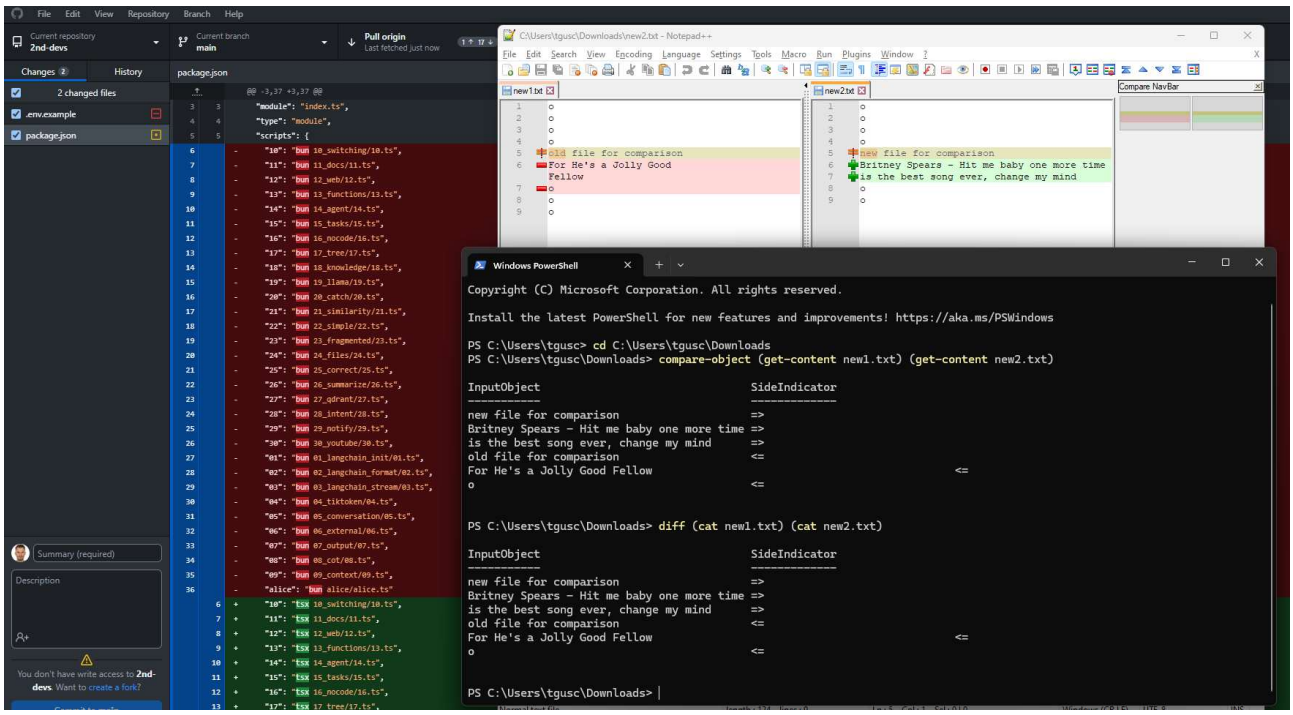


Figure 2.1: notepad++, PowerShell and Github Desktop. Screenshot from yours truly.

Chapter 3: Folder Structure

You know the benefits of keeping your files tidy and some potential problems if you don't. Now it's time to get to the meat.

I would very much like to take credit for figuring out all of this, but the truth is that all of this was already done by cookiecutter github and its spin-off by jvelezmagic github.

My example will be more basic, but I will take you by hand and get you through those dark beginner woods.

For more details regarding tests, auto-documentation, and make-files, visit the above GitHub repositories. They both serve as cookiecutter package templates, so you can easily reproduce them.

My github repo is available here. Although this is not a template, I won't stop you from making it one or using its parts to enrich your template. Take it and use it for a better tomorrow.

3.1 Folder Tree

I believe the structure below will be sufficient for 80% of projects, following the Pareto principle and paying homage to it.

The semi-visual approach to project folder tree looks like this:

```
LICENSE ..... Specifying terms on which people can use and modify your code
README.md ..... The top-level README for developers using this project
data
    external ..... Data from third party sources
    interim ..... Intermediate data that has been transformed
    processed ..... The final, canonical data sets for modeling
    raw ..... The original, immutable data dump
docs ..... Folder for documentation
models ..... Trained and serialized models, model predictions, or model summaries
notebooks ..... Jupyter notebooks. Naming convention is a number (for ordering), the
    creator's initials, and a short _ delimited description, e.g. 001_tg_data_exploration
references ..... Data dictionaries, manuals, and all other explanatory materials
reports ..... Generated analysis as HTML, PDF, LaTeX, etc
    figures ..... Generated graphics and figures to be used in reporting
requirements.txt ... The requirements file for reproducing the analysis environment, e.g.
    can be generated with pip freeze > requirements.txt
setup.py ..... Makes project pip installable (pip install -e .) so src can be imported
src ..... Source code for use in this project
    __init__.py ..... Makes src a Python module
    data ..... Scripts to download or generate data
        make_dataset.py ..... Sample script
        __init__.py
    features ..... Scripts to turn raw data into features for modeling
        build_features.py
        __init__.py
    models ..... Scripts to train models and then use trained models to make predictions
        predict_model.py ..... Sample script for prediction on new data
        train_model.py ..... Sample script for model training
        __init__.py
    visualization ..... Scripts to create exploratory and results oriented visualizations
        visualize.py ..... Sample script
        __init__.py
```

With this structure, we assume that the analysis will be a straightforward, logical process with clearly defined steps. This is rarely the case in data science, as projects often undergo many unforeseen changes.

Nevertheless, managing these changes will be much easier if I try to fit them into the above structure rather than, for example, generating new features in 15 different places in the code.

So, how would I apply this to a simple project? Let's find out.

Chapter 4: Let's Win the Stock Market

Money is usually a good short-term incentive to do things, and in data science, stock price prediction is an always enticing topic. Let's explore this idea for a while.

You will step into the role of a stock market analyst tasked with providing a short-term recommendation for Apple Inc. stock. To do this, we obviously need to acquire stock price data.

But before that, let's dive in. First things first.

4.1 Prepare Folder Structure

To begin, you need to create a folder structure for our project. There are a couple of ways to do this:

1. Manually create each folder and file to resemble the previously mentioned folder structure (not ideal).
2. Utilize tools like cookiecutter repository or jvelezmagic repository to replicate their structure:
    - cookiecutter is the most advanced option (which both I and jvelezmagic have adapted from). However, it includes many elements you might not be familiar with, such as Sphinx auto-documentation, Amazon S3 cloud object storage usage, makefiles, and tests.
    - jvelezmagic version is simpler, with many advanced features removed, and it allows for a customizable module name (I use the standard "src" for reasons I will explain later).
3. Fork (copy) my github repository for this e-book. The file structure and everything that I'll describe here are all ready there—easy peasy lemon squeezy. Of course, the best course for learning is by doing, so it's best if you play around with the files or build a similar project concurrently.

Et voila! Your folders are ready to go.

4.2 Start Your Code Development Environment

I am a fan of isolated environments (a separate Docker for each project—neat and tidy) and VS Code, which is why I advocate using them for your development environment.

You can check my first blog post - it's a step-by-step guide that should help you prepare such an environment (at least in the first half of 2024, which is when I am writing these words). It's designed for Windows users, but I am sure macOS or Unix won't differ much. It's available here.

One thing I didn't cover in the post, which might come in handy, is what a container actually is and the difference between a container and a virtual machine. I'll be honest - I had trouble figuring out how to explain this well, so I asked ChatGPT-4. After reviewing some answers, I think the best one is below:

> " Docker containers are like apartments in a building. They share the same building structure (the host operating system) but keep their own rooms (software, libraries, and files) separate. They're lightweight because they don't need their own operating system each time.
>
> Virtual machines (VMs) are more like separate houses. Each has its own foundations (a complete operating system) on top of the land (the physical server). They're heavier because they include the full package - an operating system for each VM - which takes up more resources."

Which is well supplemented by an image from Docker's article on what is a container:

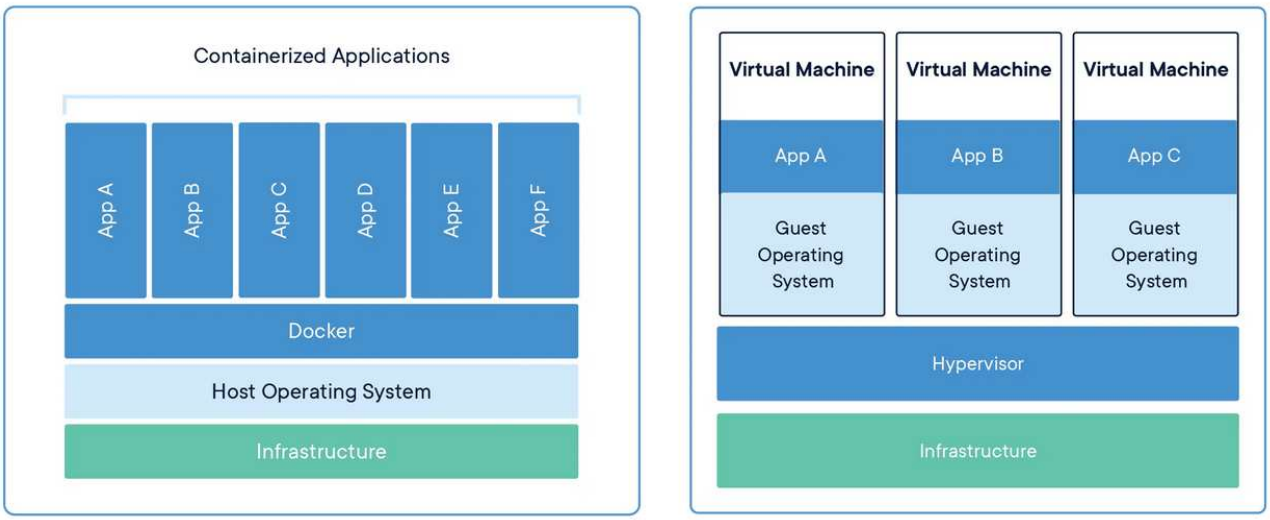


Figure 4.1: What is a container vs what is a virtual machine. Source: https://www.docker.com/resources/what-container/

4.3 Project Folder - File by File

Now let me take you to the main folder city, where the grass is green and the files are pretty!

What can you find here?

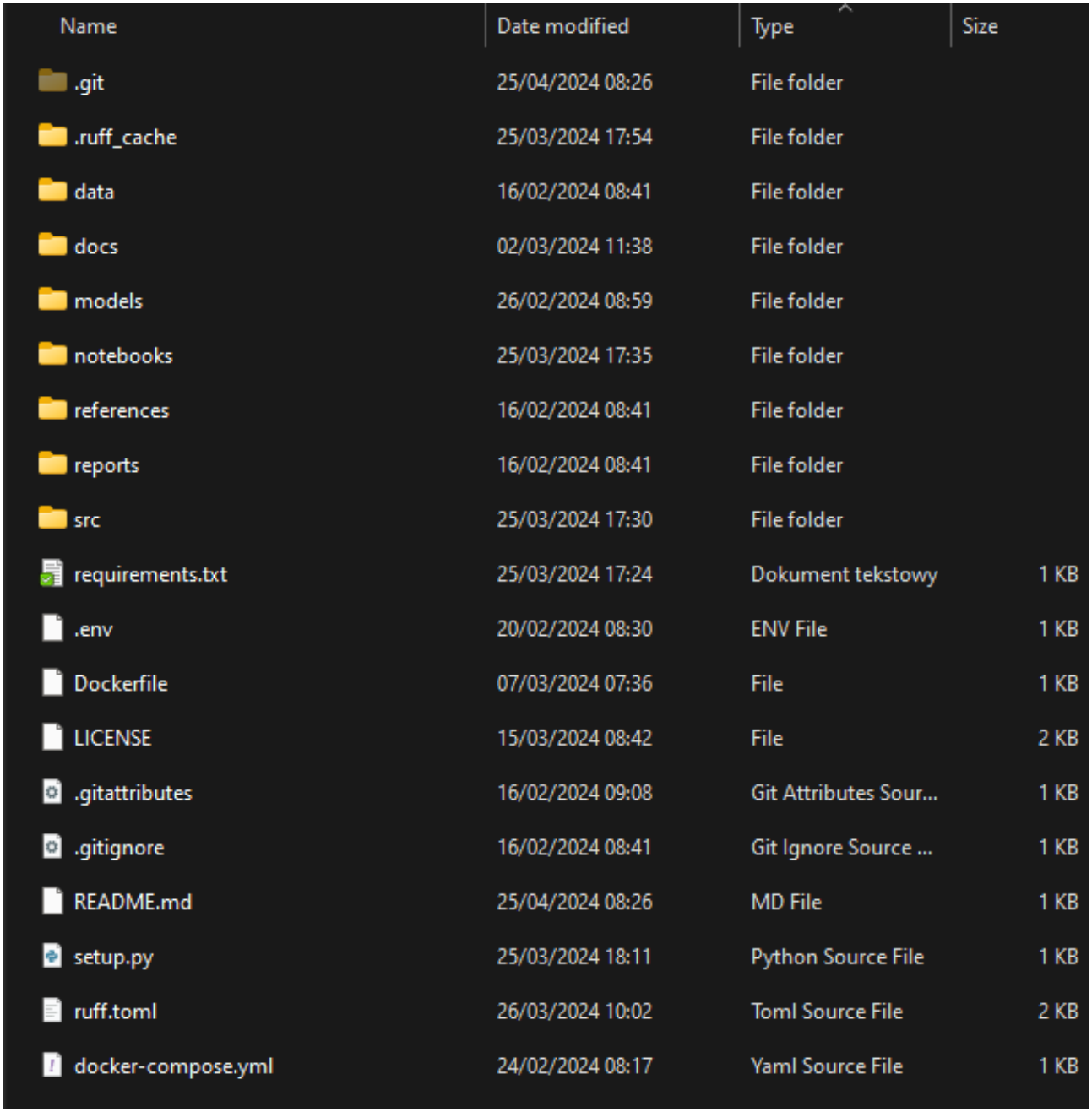


Figure 4.2: List of files in main project folder. Screenshot from yours truly.

A sharp sniper eye (yes, I'm talking about you) will spot that, besides the previously

mentioned folder structure, there are some mysterious files there. Let me *demystifAI* (don't click here! too late...) them for you in today's episode of "What's in My File?!".

4.3.1 .env File

".env" stores (or should store) environment variables for software applications. All sensitive information, like your API keys, passwords, whatever else you wouldn't like to leave your computer.

You need to care about your environment - so if you use Git or Github, add .env file to your .gitignore files (I'll expand in the next section, but if you are impatient, ask Perplexity.ai, ChatGPT or google). This prevents it from being versioned, uploaded to the internet or displayed in front of unwanted eyes.

Your .env file might look like this (please don't use/share it; this is my main banking account password. I trust you. We've never met, but I have this strange feeling...):

```
secret_apiKey = S0_S3cR3T
```

Later, these variables can be read in Python environment via python-dotenv package (**Caution!** At least as of date when I'm writing this post: you install the package to your environment using "python-dotenv"). Sample code here:

```
# load module
import dotenv
# load .env file
dotenv.load_dotenv('/path/.env')

# apply value to existing variable
apiKey = os.environ["secret_apiKey"]
```

Again, this is to avoid declaring your passwords in the actual notebook or code with

entries like:

```
password = Doggo123WoofW0of
```

and then accidentally sharing them while uploading your projects to the internet. Humans are fallible by nature—we tend to forget. So, what's the best way to prevent forgetting? Eliminate the opportunity to face the consequences of forgetting. And of course, don't forget to eat nuts, read and engage in plenty of crosswords and Sudoku.

4.3.2 .gitattributes and .gitignore Files

So this is a section specifically for Git or Github users. If you don't use a versioning system (remember, that this is a mortal sin. Repent, repent!) this section is not applicable.

.gitattributes specifies properties for Git/Github in specified path. In our case we have very simple contents:

```
# Auto detect text files and perform LF normalization
* text=auto
```

Which basically means - whenever Git processes the end of text line it will:

- Mark the end of line as LF (Line Feed - escape `\n`) or CRLF (Carriage Return Line Feed - `\r\n`, originating from the way old printing machines worked) in the local repository copy, most commonly depending on your operating system (Linux based systems - LF, Windows/DOS systems - CRLF).
- In the pushed main repository location, it will consistently mark end of lines as LF.

This topic is a great conversation starter at any party—discuss the differences between LF and CRLF to engage your audience and win their hearts in no time!

Unified line endings allow us to correctly interpret line endings on our local computers (which can be crucial for interpreting Python indentations, for example), while in the reposi-

tory, we avoid conflicts and apparent differences. Without this, if two users were contributing - one using Unix and the other using Windows - each contribution could make entire files appear different & compute differently due to line ending changes.

Okay, if we were actually at a party, you might want to ignore me right now, so let's move on to .gitignore.

.gitignore specifies files that are not exchanged with the repository (whatever you edit in them, the changes remain only on your machine). This should include all temporary files that you don't want stored in the main repository, as well as your secrets. A good example of .gitignore content is:

```
# DotEnv configuration
.env

# Database
*.db
*.rdb
```

Which essentially means:

- Don't share my passwords and API keys over Intranet / Internet.
- Don't upload whole databases to the repository each time they change (that can be a lot of transferred data).

Figure 4.3: "Shh, mate... I had too much electricity yesterday!". Robot asking another robot for silence. Image generated with DALL·E

4.3.3 Docker Files

Docker-compose.yml and Dockerfile files will allow us to launch a container using, for example, Docker Desktop.

For a deeper explanation, check my post here.

I'll show the contents of the files and then explain the subtle differences between the configuration below and the one mentioned in a previous post.

docker-compose.yml contents:

```
services:
    app:
        build: .
        container_name: python_dev_env
        env_file:
            - .env
```

```
        command: sleep infinity
        volumes:
            - .:/app

# build: . - specifies that you want to build from the local Dockerfile
# to run, you need to execute a single command: docker compose up
```

Dockerfile contents:

```
# Base image - typically Python for Python dev environments
FROM python:3.11-slim

# Set the working directory
WORKDIR /app

# Install dependencies and src
COPY ./requirements.txt ./
COPY ./setup.py ./
RUN pip install --no-cache-dir --upgrade -r requirements.txt

# Copy src package codes to the container
COPY ./src ./src

# Common pitfalls:
# 1) Filename has to be Dockerfile exactly. Otherwise it won't work.
# 2) You must copy ./setup.py to install the src package
```

First of all - you might notice that I changed the volume and working directory from *"/code"* to *"/app"*. Why is that? Well, one might argue that our analysis is not merely "code". It's an "application" with codes, data, images & analysis leading to conclusions.

In a production environment, you might have splits like *"/data"* and *"/app"* separating codes from underlying or produced data. That might be justified - those folders can point at

different servers, for example, a bigger and slower one for data, and a smaller, faster one for the codebase and applications.

However, in my case, I find structuring all files in a project-oriented manner very beneficial. When I think about a project, I only have to locate one project folder to find all necessary files - I don't need to search separately in app, data, and/or code folders. This way, it's neat and tidy (however, for production applications, stick to your machine learning engineers' guidelines).

There is also one more reason (a true lifehack): the "app" folder will often be your first folder when sorted alphabetically. That makes it quick to access when you manually navigate through folders (which I hope is rare).

Secondly, there is an additional COPY statement after copying the requirements, stating that we need to copy the "setup.py" file. This is because you will package all your project code located in the *src* folder and install it into your environment when you process the *requirements.txt* file. I will elaborate more on this in a later section, but note this: this setup allows you to access all your Python data classes and functions located in the *src* codes (if structured properly) very quickly.

This adheres to the principles of modularity, reusability, and portability.

4.3.4 Setup.py

In the setup.py file, you will instruct the interpreter to install application-specific packages, making them referable in codes and notebooks. In my case, all package files are located in a standardized *src* folder for source codes.

The word "standardized" is key here. While jvelezmagic cookiecutter allows you to choose the package name, I find that this increases complexity. In our Python notebooks or codes, we refer to packages and subpackages (more on this in the next section). Having one source package name across projects allows me to lift and shift code pieces and limits the need for manual corrections, while still preserving each project's individuality, as we keep them in separate folders.

Setup.py file contents:

```
from setuptools import find_packages, setup
setup(
    name='src',
    packages=find_packages(),
    version='1.0.0',
    description='How to structure your project? Explaining best practices for organizing data science projects in Python using Apple Inc. short-term price prediction as an example. Project goals: replicability, maintainability, portability, clarity.',
    author='Tomasz Gusciora',
    license='MIT',
)
```

Now, bear with me - we are essentially saying that we set up our project package as *src* with appropriate metadata: version, description, license (this creates a *LICENSE* file), and author credentials (this guy sounds familiar).

I'm not a lawyer (sometimes maybe a Devil's advocate), but I use the MIT license, which seems to be the most frequently used among freeware. You can read about license types here: choose a license.

The contents of the MIT License ('license' file) are as follows (more or less; I can't guarantee that I fully understand this ancient language of lawyers): you are sharing your codebase/software for free. However, if someone uses it to harm someone else's mother, you cannot be held accountable. It's akin to Alfred Nobel saying, "I did not know that dynamite would be used by bad people, please don't sue me."

```
MIT License

Copyright (c) 2024 Tomasz Gusciora

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "
```

```
    Software"), to deal in the Software without restriction, including
    without limitation the rights to use, copy, modify, merge, publish,
    distribute, sublicense, and/or sell copies of the Software, and to
    permit persons to whom the Software is furnished to do so, subject to
    the following conditions:

The above copyright notice and this permission notice shall be included
    in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS
    OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
    MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
     IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY
    CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT,
    TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
    SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

Figure 4.4: "Your Honor, I swear his wallet was granted to me based on the MIT license!". Robots arguing a case in a courtroom. Image generated with DALL·E

While legal matters are obviously very exciting, let's delve into the *packages = find_packages()* section. This command instructs pip to search the folder tree for packages and subpackages, which is extremely useful for adopting a modular approach in our project.

How does this work?! Let's explore an example.

4.3.5 Python Packages, Subpackages and Modules

To explore how Python handles packages, let's examine what's inside my *src* folder structure (project .py codes). To do this, I will open the project folder in the command line. (Note: Your location might be different, and the commands below might vary for non-Windows operating systems).

```
cd C:\GitHub\E001_smart_structuring\src
tree /f
```

The *tree /f* command should produce a tree-like representation of folders and files that are present in the *src* folder. In my case, there are numerous .gitkeep and __pycache__ files lying around (irrelevant for this section), so I have omitted them from the representation below. Here is the tree structure of our "src" folders and files:

```
__init__.py
data
    make_dataset.py
    __init__.py
features
    build_features.py
    __init__.py
models
    predict_model.py
    train_model.py
    __init__.py
utils
    df_inspect.py
    paths.py
    requirements_versions.py
    start_wrapper.py
    __init__.py
visualization
    visualize.py
    __init__.py
```

A keen observer will notice that there is an *__init__.py* file in each folder. This signals to the Python interpreter that Santa's home and he's brought presents (indicating that a module or package is present for import). If there is only one .py file to import, it's considered a module. If there is a collection of modules organized in a directory tree, as in our case, we are dealing with a package.

If I have a one-folder-deep structure, for example, I am in the *src* folder and there is only the *data* folder present and its contents, if my system path is correctly set up (as the *src* folder location), I can import *data* as a package with the below command.

```
1 import data
```

This allows me to call modules (.py files), granting access to all their code, functions, and objects.

```
import data.make_dataset
```

As long as I have *__init__.py* files in the folders and the system path set up properly, I need only the second command. I can then utilize whatever is in that code, such as specific functions.

```
import data.make_dataset

# Downloading data from Stooq
stock = 'aapl.us'
data_in = data.make_dataset.stooqData(stock)
```

So essentially, I have code for the *stooqData* function written in the *"data/make_dataset.py"* file. I call it in my main notebook and execute it without needing to copy the full code of the function.

In this particular case, it will download *aapl.us* stock (Apple Inc. - you might have heard of them; they have the worst apples ever, completely inedible.)

Now let's take it up a notch. My project folder structure doesn't start in the *src* folder; the main folder is one level higher. Therefore, *src* becomes my package, and *data* along with other folders become subpackages.

How to import subpackages? Simply by adding one dot:

```
import src  # importing package
import src.visualize  # importing subpackage
import src.data.make_dataset  # importing module
```

So now, let's return to those *__init__.py* files. You can leave them empty, but then you will need to specify each module you want to import, as in the example above.

Also, remember that you can have sub-sub, sub-sub-sub, $\sum_{i=1}^{inf}$ sub-packages and Python will deal with that. I'm not sure how readable that will be. So, as a rule of thumb, don't exceed a tree depth of three unless absolutely necessary. You can include modules throughout - within your main package, subpackages, and so on.

How can you make your project-specific package imports a bit smarter (or just lazier)?

You can nest subpackage and module imports in your *__init__.py* files, allowing you to import all subpackages and modules with a single command: *import src*. To achieve this, you must specify that you will import all subpackages in your package init - *"/src/__init__.py"*:

```
# Import subpackages
import src.data
import src.visualize
import src.features
import src.models
import src.utils
```

Then, in each subpackage init file, specify that you want to import modules. For example, see the *"/src/data/__init__.py"* file below:

```
# import modules
import src.data.make_dataset
```

Now, if you have it structured like this, it only takes one command to run them all in your notebook:

```
# import package, all subpackages and modules
import src

# use module
# Downloading data from Stooq
stock = 'aapl.us'
```

```
data_in = src.data.make_dataset.stooqData(stock)
```

Importing src will trigger the importing of subpackages, and importing a subpackage will trigger the importing of its modules.

The downside is that you must maintain the contents of the *__init__.py* files. If you call a module that doesn't exist, the code will throw an error. However, there are ways around this (I don't recommend this approach), such as ignoring errors with try/except:

```
# if do_something() fails, ignore and move on
try:
    do_something()
except Exception:
    pass
```

What's the alternative?

Instead of importing the entire package, you can selectively import only the components you need. This approach is common with large, publicly available packages such as sklearn. You load specific functionalities as required. For example, if you want to use model prediction error metrics like MSE (Mean Squared Error), you would import it separately.

This ensures that memory usage and load times are minimized, which can be crucial for performance in production environments.

Here's how you can do it:

```
# Will not work - import sklearn does not import all subpackages
import numpy as np
import sklearn
y_true = np.array([2, 2, 2])
y_pred = np.array([2.5, 3, 4])
MSE = sklearn.metrics.mean_squared_error(y_true, y_pred)
print(MSE)

```

```
# Will work
import sklearn.metrics
MSE = sklearn.metrics.mean_squared_error(y_true, y_pred)
print(MSE)
```

Proof (I know you trust me, but I don't trust myself):

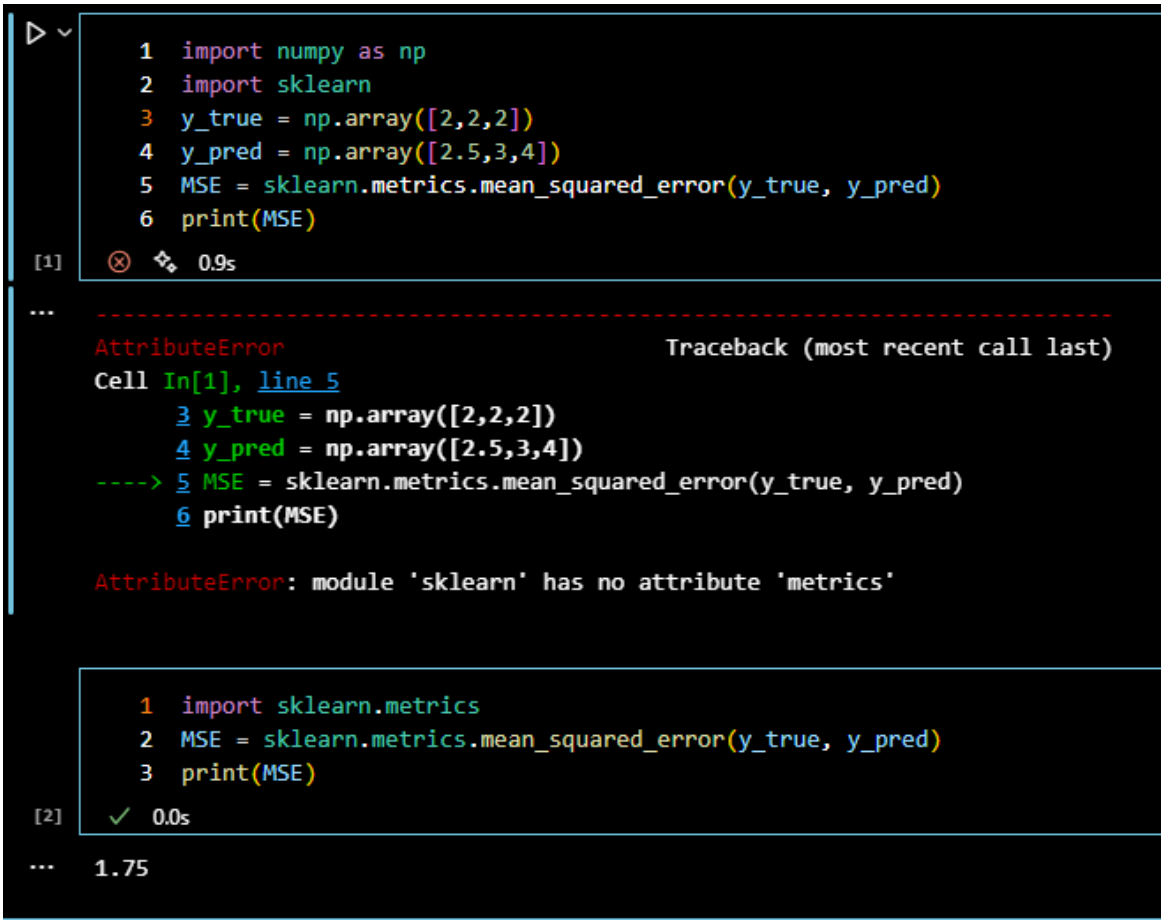


Figure 4.5: Sklearn MSE imports check. Screenshot from yours truly.

To sum up, from my perspective:

- Keep *__init__.py* files tidy and maintain them in your src project-specific package.
- Maintain your package to contain only stuff that you use - get rid of redundant code (limit complexity; increase readability).
- For publicly available packages - import only what you need. Save memory & reach for max performance.

So, if you value tidiness, maintain your inits meticulously. This approach offers several

advantages:

- A single command is needed to import the entire package in each notebook (*import src*).
- There is a single version of truth in your package files. If you need to change one function, you do not need to modify it in multiple files. A change in the package affects all codes/notebooks that call this function (adhering to the DRY principle—Don't Repeat Yourself).
- You can reuse packages, subpackages, and modules in different projects - simply copy codes & update their init files for correct imports.

4.3.6 Requirements.txt

Now let's review what I am installing for my project. The *requirements.txt* file should be updated with everything you use in your project, ensuring all dependencies are installed when the container is built.

It's common to access the terminal during the development phase to install additional packages. However, if you plan to use these packages regularly, add them to the *requirements.txt* file.

The contents of the basic *requirements.txt* for my project:

```
# Installing local src package in editable mode
-e .

# Basic packages
numpy
pandas
matplotlib
setuptools

# Clean-code
```

```
python-dotenv
ruff
pre-commit

# Packages for our analysis
# Hi Mark
prophet
scikit-learn

# For Prophet (Hi Mark) plots - not used in project
ipywidgets
plotly

# Packages for smooth Python development environment running in VS Code
    on Docker
IPython
jupyter_client
jupyter_core
jupyterlab
notebook
```

And explanation of what and why:

Package	**Explanation**
-e .	This command searches the current directory (denoted by ".") for a setup.py file (or pyproject.toml—you can read about it here) and installs it in editable mode. This is important, as it allows you to make changes on the fly during the development phase. **Warning**: Without a properly specified setup.py file, this operation will fail.

Package	Explanation
numpy	The most popular Python library for numerical operations is NumPy. You can find more information in the NumPy documentation.
pandas	The most popular Python library for data processing is Pandas. You can find more information in the Pandas documentation.
matplotllib	The most popular Python library for visualization is Matplotlib. You can find more information in the Matplotlib documentation.
setuptools	This allows you to set your "src" package as an installable package in a development environment. You can find more information in the Setuptools documentation.
python-dotenv	Corresponds to dotenv (import dotenv). It manages environment files (.env) and allows you to safely store secrets, which is a safety best practice. You can find more information in the Dotenv documentation.
ruff	Extremely fast Python linter and code formatter, Ruff beautifies your code and performs tasks like removing unused imports. Written in Rust, which contributes to its speed, it executes commands from the CLI (Command Line Interface, such as the VS Code terminal). For optimal use, set the ruff.toml file in your project. Using Ruff, you can format your code to PEP8 standards with just one or two commands, making it easier to read and understand for everyone. That's amazing. You can find more information in the Ruff documentation.
pre-commit	Automatic formatting checker that will not allow you to push your code to the main branch before ensuring it is readable and properly formatted. Just like Ruff, it is a game-changer when it comes to maintaining clean code. You can find more information in the Pre-commit documentation.

Package	Explanation
prophet	A powerful package developed by Facebook (Hi Mark, I know you are reading this) for data forecasting. You can find more information in the Prophet documentation.
scikit-learn	The go-to package for many machine learning tasks. You can find more information in the Scikit-learn documentation.
ipywidgets	This package enables interactive plots for Jupyter Notebooks. Prophet (Hi Mark) uses it for its plotting. I won't use it in this project, as I opted for simplicity with Matplotlib; however, I'm leaving it here for anyone who wants to explore further. You can find more information in the Ipywidgets documentation.
plotly	Another package for interactive plots, Plotly is also required by Prophet (Hi Mark) for plotting. The same explanation as above applies. You can find more information in the Plotly documentation.
IPython	Interactive shell for Python, used in Jupyter. You can find more information in the IPython documentation.
jupyter_client	One of the packages for using Jupyter Notebooks. You can find more information in the Jupyter documentation.
jupyter_core	One of the packages for using Jupyter Notebooks. You can find more information in the Jupyter documentation.
jupyterlab	Jupyter Lab is a package for using Jupyter Notebooks. You can find more information in the Jupyter documentation.
notebook	Classic Jupyter Notebook is a package used for interactive computing. You can find more information in the Jupyter documentation.

There is one more thing you can do to avoid potential version problems: add requirements specifiers (preferably strong "==" ; see pip requirement specifiers). With these added, your *requirements.txt* file will look more like:

```
# Installing local src package in editable mode
-e .

# Basic packages
numpy==1.26.4
pandas==2.2.1
matplotlib==3.8.3
setuptools==69.2.0

# Clean-code
python-dotenv==1.0.1
ruff==0.3.4
pre-commit==3.6.2

# Packages for our analysis
# Hi Mark
prophet==1.1.5
scikit-learn==1.4.1.post1

# For Prophet plots - not used in project
ipywidgets==8.1.2
plotly==5.20.0

# Packages for smooth Python dev environment run in VS Code on Docker
IPython==8.22.2
jupyter_client==8.6.1
jupyter_core==5.7.2
jupyterlab==4.1.5
notebook==7.1.2
```

You can create a similar file by running the following command in your notebook, where "!" calls pip to run a CLI script:

```
!pip freeze > 'requirements.txt'
```

However, in my experience, particularly when running this in a Docker container, you'll end up with a much longer list of packages, some of which you may have never heard of.

In the next section, I will introduce a powerful formatting tool that automates reading and formatting your code. Ok, ready to go!

Figure 4.6: "It was supposed to be around the block, Karen!". Robotic blast-off to space. Image generated with DALL·E

4.3.7 ruff.toml

Formatting code and adhering to clean-code rules can be a real pain in the a-hole.

But don't worry. I'm here to help.

Well, not me exactly, but Ruff can.

Meet Ruff - our superhero.

Ruff can format your code incredibly fast to adhere to the expected guidelines.

Simply go to the desired folder in the terminal and type the following:

- *ruff format* - to format all files in the current folder and subfolders
- *ruff check* - to check if files in the current folder and subfolders pass all rule checks (for example, rule E101 - Indentation contains mixed spaces and tabs. You can find Ruff's rules here)
- *ruff check –fix* - to not only check but immediately fix all broken rules (wherever possible)

And the sheer beauty of it - Ruff will do this in a blink of an eye!

But don't take my word for it. See for yourself in my screenshots.

In my dockerized Visual Code instance, I'll open a terminal and ask Ruff to format one folder (please, Ruff, please):

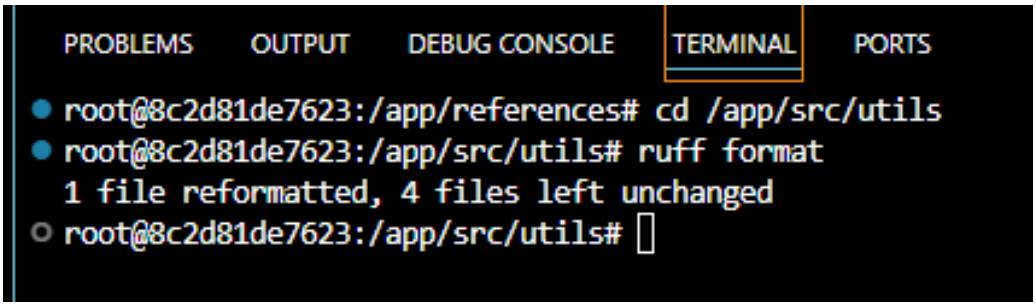


Figure 4.7: Ruff terminal format. Screenshot from yours truly.

Ok, one file formatted. Let me look for signs. Here's a tip: according to PEP 8 Python guidelines, a code line should not exceed 79 characters. Why 79? There are several reasons:

- Readability - shorter (but not too short) lines are easier to read.
- Ability to open multiple files side by side on wide screens. This aids in visual file comparison.
- Consistency - standardized settings across developers make it easier to read codes. After getting used to it, based on line length your mind will subconsciously switch to "code-read and review" mode (lifehack alert).
- Accessibility - allows people with smaller screens or visual impairments to read code without horizontal scrolling, making reading easier.

Now let's look at the pre vs post Ruff treatment on our patient file:

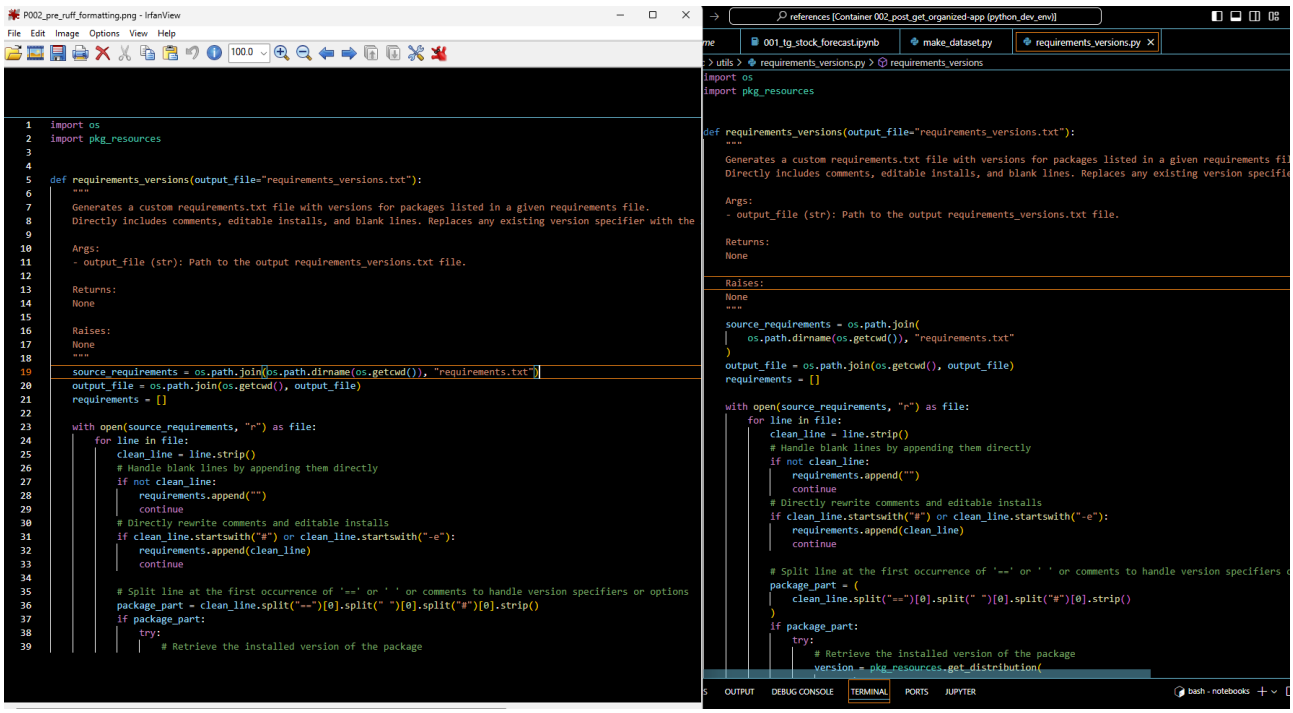


Figure 4.8: Pre and post Ruff code formatting. Screenshot from yours truly.

Now lines of code have a maximum of 79 characters.

However, I can still see that some comment sections and docstring parts exceed my 79-character limit. This will become apparent when I run the check & fix command.

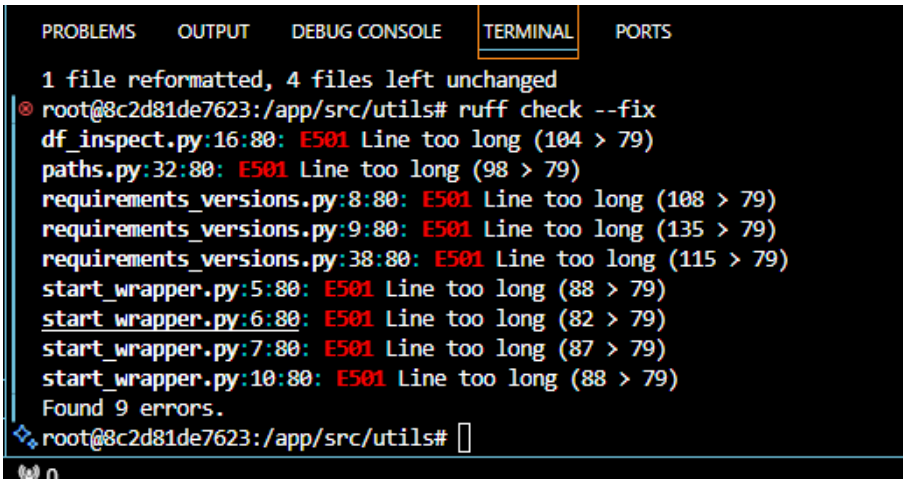


Figure 4.9: Ruff terminal check and fix. Screenshot from yours truly.

Ah! It shows me exactly what needs to be corrected in the following format:

- File name: requirements_versions.py
- Row number: 8
- Column number: 80

- Error explained: E501 Line too long (108 > 79 characters). So smart. Wow.

Beautiful! Now I know exactly what and where to correct to achieve perfectly clean code.

Another lifehack: when working with line length errors, correct them from the last one to the first. That way, you'll avoid having to figure out new line lengths after you split the first line (if you split line 8 into two lines, the second error will naturally move from line 9 to 10. Easy when they are next to each other, harder for your brain when they are set apart. Save energy, save brainpower.)

To make things even easier, you can come up with a standardized Ruff configuration file that you'll just copy to new project structures. That way, you don't need to spend time setting up Ruff each time. Time savings rise with the number of developers involved in the project, as it forces standardization of code quality.

Similar code across developers = less brainpower needed to work on the codebase. Smart.

A *ruff.toml* file can look like below:

```
# Exclude a variety of commonly ignored directories & file-types.
exclude = [
    ".bzr",
    ".direnv",
    ".eggs",
    ".git",
    ".git-rewrite",
    ".hg",
    ".ipynb_checkpoints",
    ".mypy_cache",
    ".nox",
    ".pants.d",
    ".pyenv",
    ".pytest_cache",
    ".pytype",
```

```
    ".ruff_cache",
    ".svn",
    ".tox",
    ".venv",
    ".vscode",
    "__pypackages__",
    "_build",
    "buck-out",
    "build",
    "dist",
    "node_modules",
    "site-packages",
    "venv",
    "setup.py"
]

# Adhering to Python PEP 8 guidelines
line-length = 79
indent-width = 4

# Formatting Jupyter notebooks as well
extend-include = ["*.ipynb"]

[lint]
# Enable Pyflakes (`F`) and a subset of the pycodestyle (`E`) codes by
    default.
# Unlike Flake8, Ruff doesn't enable pycodestyle warnings (`W`) or
# McCabe complexity (`C901`) by default.
select = ["E1", "E4", "E5", "E7", "E9", "F", "W", "I", "N"]
ignore = ["E402","F401"]

# Allow fix for all enabled rules (when `--fix`) is provided.
```

```
47 fixable = ["ALL"]
48 unfixable = []
49
50 # Allow unused variables when underscore-prefixed.
51 dummy-variable-rgx = "^(_+|(_+[a-zA-Z0-9_]*[a-zA-Z0-9]+?))$"
52
53 [format]
54 # Like Black, use double quotes for strings.
55 quote-style = "double"
56
57 # Like Black, indent with spaces, rather than tabs.
58 indent-style = "space"
59
60 # Like Black, respect magic trailing commas.
61 skip-magic-trailing-comma = false
62
63 # Like Black, automatically detect the appropriate line ending.
64 line-ending = "auto"
65
66 # Enable auto-formatting of code examples in docstrings. Markdown,
67 # reStructuredText code/literal blocks and doctests are all supported.
68 #
69 # This is currently disabled by default, but it is planned for this
70 # to be opt-out in the future.
71 docstring-code-format = true
72
73 # Set the line length limit used when formatting code snippets in
74 # docstrings.
75 #
76 # This only has an effect when the `docstring-code-format` setting is
77 # enabled.
78 docstring-code-line-length = 79
```

I hope the configuration file is pretty self-explanatory. For more detailed information, you can find it on the Ruff documentation website with specific explanations of the rules used in *select* and *ignore* parameters.

Why is this important? Because decisions have consequences.

As I specified earlier, I am a fan of maintaining *__init__.py* files in such a way that I can import all packages and sub-packages for my project with a single import command. This causes those files to directly violate rule F401 - unused import statements, because imports are the only commands there.

That's why I put this rule in the ignore section and don't have to bother myself about it anymore (of course, now it will not flag unused imports in Jupyter notebooks and other .py files, and that's another consequence I have to deal with).

Another exception is adding *setup.py* to the list of excluded files. It contains a lengthy description of the project and frequently breaks the line length rule. I could either split the description into multiple concatenated strings (boring) or opt out of checking it. I vote in favor of the latter.

To finish this section on a high note: Unleash the power of Ruff. It's your secret weapon for coding with clarity and style.

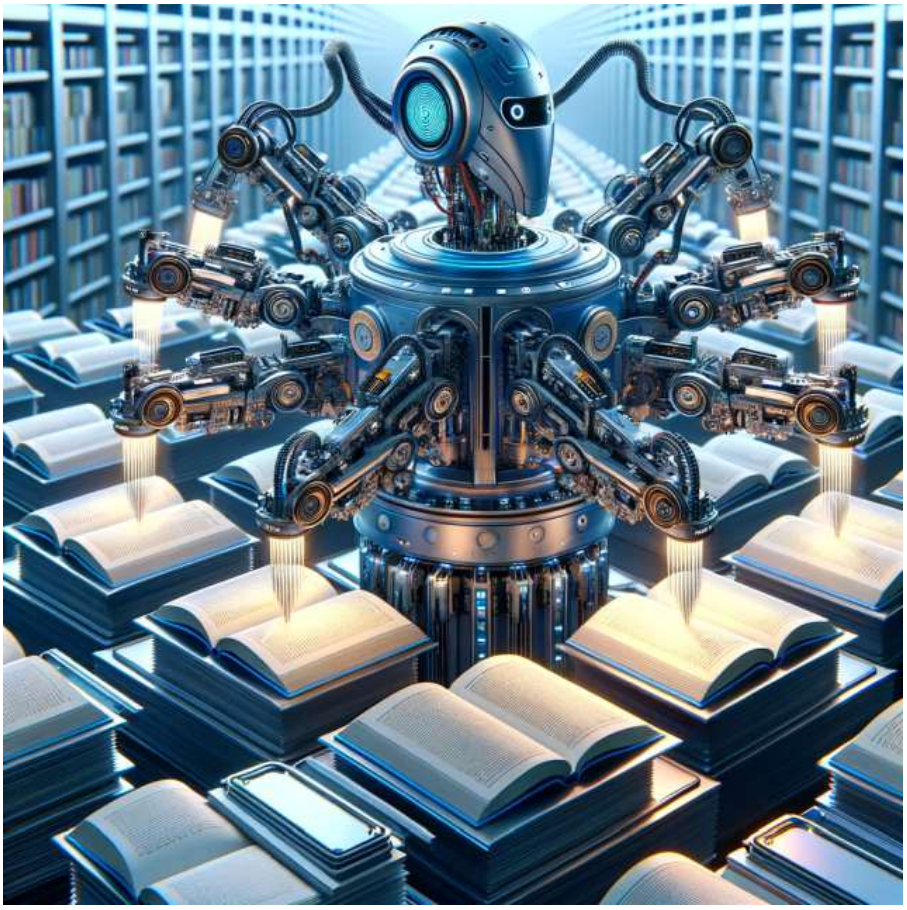


Figure 4.10: "Go to school, they said. You'll land a dream job..." . Robot writing many books at once. Image generated with DALL·E

4.4 Notebook With Analysis

The time has come to do some quad damage (a Quake reference). I must admit—I've always sucked at FPS games. RPGs like Fallout and Baldur's Gate were more my thing.

Let's start by creating a notebook in *"/app/notebooks"*. I'll call mine *001_tg_stock_forecast.ipynb* (tg - from my initials). You can do this from VS Code, which I explained in my first blog post. If you do read my first blog post - please do so while playing Madonna - Like A Virgin in the background.

4.4.1 Technical setup

4.4.1.1 Imports

In your first cell, you should have all your imported packages, possibly organized in a tidy way. Why? Because having your imports scattered around can make it trickier to debug the code, and there's a bigger chance you will import the same package multiple times. This also gives a clear overview of the tools used in the project. Of course, development time is development time - but after playtime, make it tidy (GIF for your entertainment - copyrights to the mighty web).

In my project, I will use a mix of standard packages that I believe make my life a tad easier, at least in my current perception of reality.

Here's my Jupyter notebook's first cell contents:

```
# Import project package, all subpackages and modules
import src

# Load dictionary of project paths
# Crawler from parent directory
paths = src.utils.paths.paths_dictionary()
print("Contents of paths dictionary:")
for key in paths:
    print(key, ": ", paths[key])

# Load environment variables - safe way to load sensitive data
import os
import dotenv
dotenv.load_dotenv(os.path.join(os.path.dirname(os.getcwd()), ".env"))

# Basic packages
```

```
import numpy as np

# Pandas and display settings
# Pandas.testing for dataframes comparison
import pandas as pd
import pandas.testing

pd.options.display.max_columns = 300
pd.options.display.max_rows = 100
pd.options.display.max_colwidth = 250
pd.options.display.float_format = "{:,.2f}".format

# FB (Hi Mark) module for forecasting
from prophet import Prophet
from prophet.serialize import model_from_json, model_to_json
```

As a natural consequence, I import the previously mentioned packages as well as my well-prepared *src* package (along with sub-packages, thanks to the import structures in my init files).

I also set up some *pandas* options for cleaner display. The float format *.2f* allows me to display all float variables as full numbers, with "," as the separator and 2 decimal points. The default is often scientific notation, which might not be the easiest to read, for example:

- 9.767604e+07 (scientific notation)
- 97,676,041.84 (*.2f* notation)

To keep all my secrets secure (shhhh... don't tell anybody) I load the *.env* file using below command:

```
dotenv.load_dotenv(os.path.join(os.path.dirname(os.getcwd()), '.env'))
```

This translates to:

1. *os.getcwd()* - get my current working directory (workbook directory)

2. *os.path.dirname(path)* - get the parent directory of the path
3. *os.path.join(X, Y)* - concatenate X and Y as a single path

That way, I am pointing directly to my *.env* file in my main project folder with sniper precision.

A more advanced topic is calling *src.utils.paths.paths_dictionary()*.

4.4.1.2 src.utils.paths.paths_dictionary()

First of all, it's not always the best solution to name your package or sub-package *utils*. Utility (quoting a couple of definitions from the Cambridge dictionary):

1. the usefulness of something, especially in a practical way
2. able to do several different things well
3. designed mainly to be useful rather than attractive or fashionable (that's hurtful)

That's a rather broad spectrum of possible uses. However, I couldn't find a more suitable name. If you can, drop me a message.

In this sub-package, we have a *paths* module with a defined function (the only one in that module) *"paths_dictionary"*. Let's take a peek:

```
import os

def paths_dictionary():
    """
    Creating dictionary with paths in the parent/project directory.
    Crawls through folders except names starting with '.' and '_'.
    Returns a dictionary with folder paths separated with "_" as
    keys and their true paths as values.

    Notes
    -------------------
    Required libraries: \n
    * import os
```

```
14
15      Returns
16      --------
17      data : dictionary
18             Dictionary with paths.
19      """
20
21      parent_path = os.path.dirname(os.getcwd())
22      paths_dict = {}
23      for root, dirs, files in os.walk(parent_path):
24          # Delete all folders starting with '.' and '_' and their
    subfolders
25          dirs[:] = [
26              d for d in dirs if not d.startswith(".") and not d.startswith
    ("_")
27          ]
28
29          for dir in dirs:
30              folder_path = os.path.join(root, dir) + "/"
31              # Adding dictionary keys, replacing separators to "_" and
32              # deleting paths before parent
33              dict_key = (
34                  folder_path.replace(parent_path, "")
35                  .strip(os.sep)
36                  .replace(os.sep, "_")
37              )
38              paths_dict[dict_key] = folder_path
39          paths_dict[parent_path.replace(os.sep, "")] = parent_path + "/"
40      return paths_dict
```

This is a crawler. It crawls from your main folder to your subfolders and creates a dictionary storing string objects pointing to particular folder paths.

Now let's add some AI magic to this:

1. Remember to always document your code - you can use AI for that. Check out GitHub Copilot for integration with your IDE (for example, Visual Studio Code) or Grimoire ChatGPT Assistant if you don't mind sending your data over the web. Always proof-read.
2. Want to explain the above function in a more visual-friendly way? Try Diagrams: Show Me GPT Assistant. Below is a sequence graph made by this GPT, which explains this function quite nicely. Here's a link to the actual conversation (can't promise that it still works).

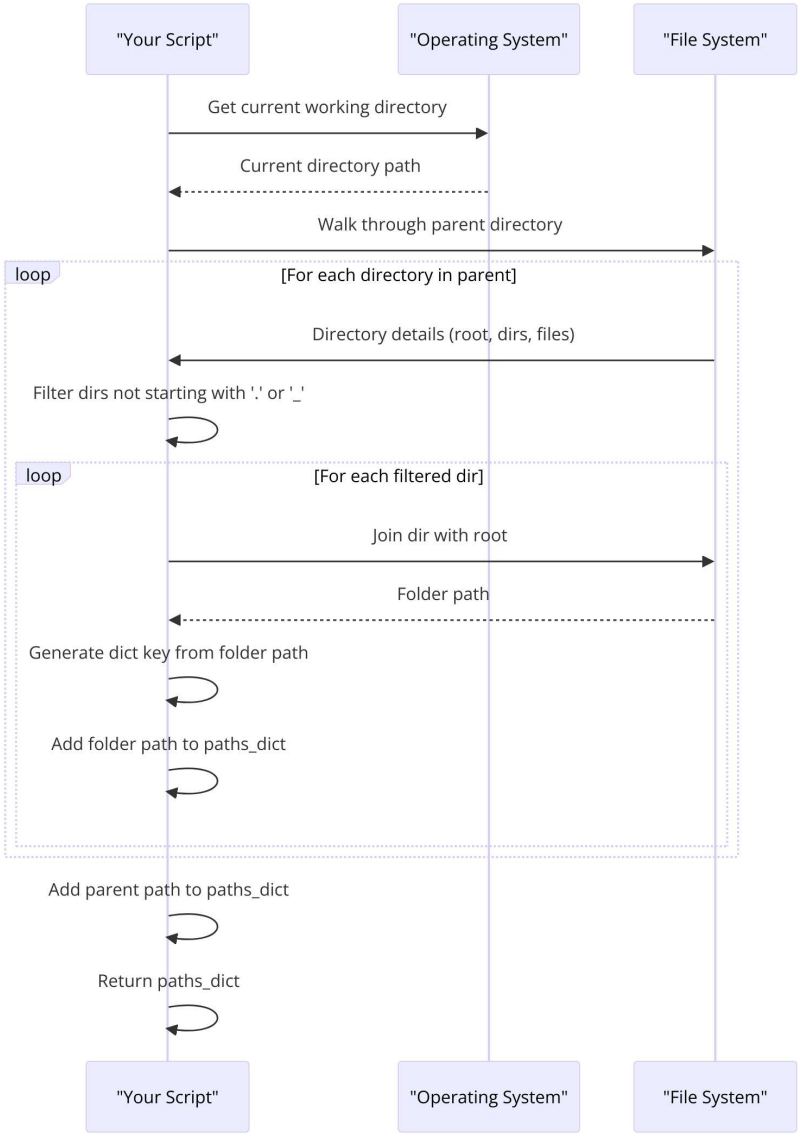


Figure 4.11: Sequence graph for *paths_dictionary* function. Image created with Grimoire ChatGPT Assistant

If you have your Docker container set up in the same way as I do, the command:

```
for key in paths:
    print(key, ": ", paths[key])
```

will nicely print the contents of this dictionary, and you will see something like:

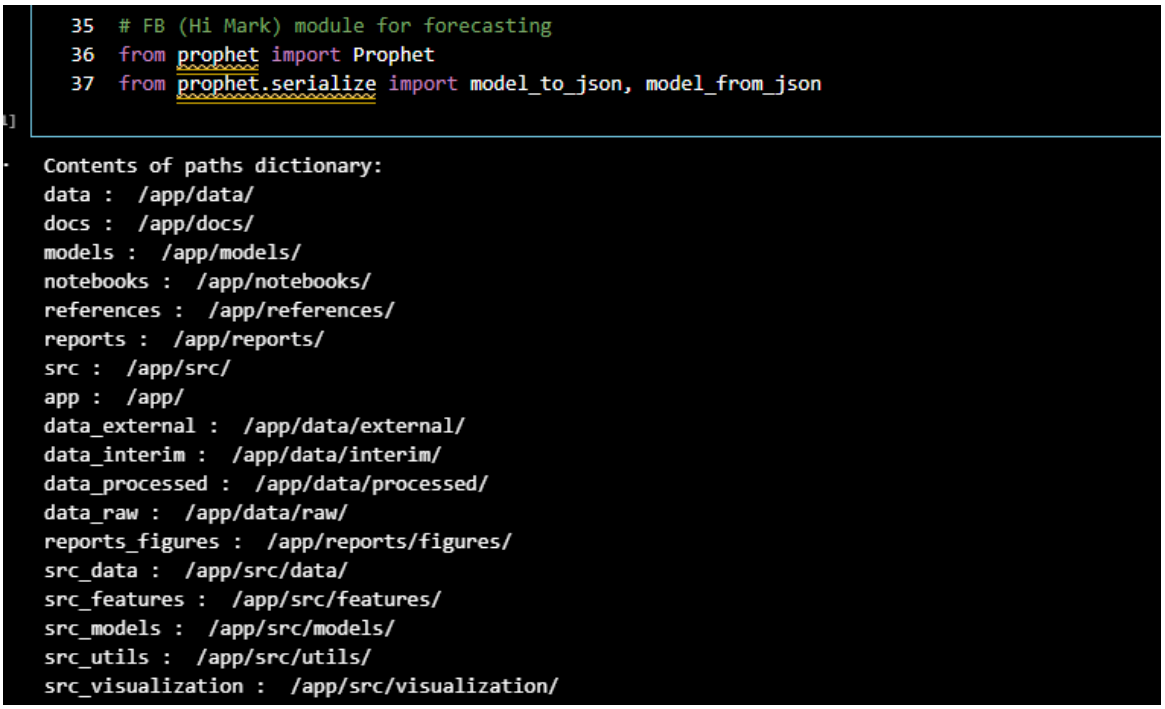


Figure 4.12: Path dictionary contents. Screenshot from yours truly.

Why is this useful?

- **Flexibility**: Paths become relative. You can copy your project to a different folder, run it without code changes (assuming the project structure stays the same), and it will work!
- **Maintainability**: The list of paths is kept in one place, making it easier to maintain.
- **Security**: Paths are not hard-coded, so their values are not shared unless you want them to be (I shared mine by printing the dictionary contents). If you keep all your projects at *"/secretpassword/admin123/yourmomma/files"*, you don't necessarily want to inform the whole GitHub community about it.

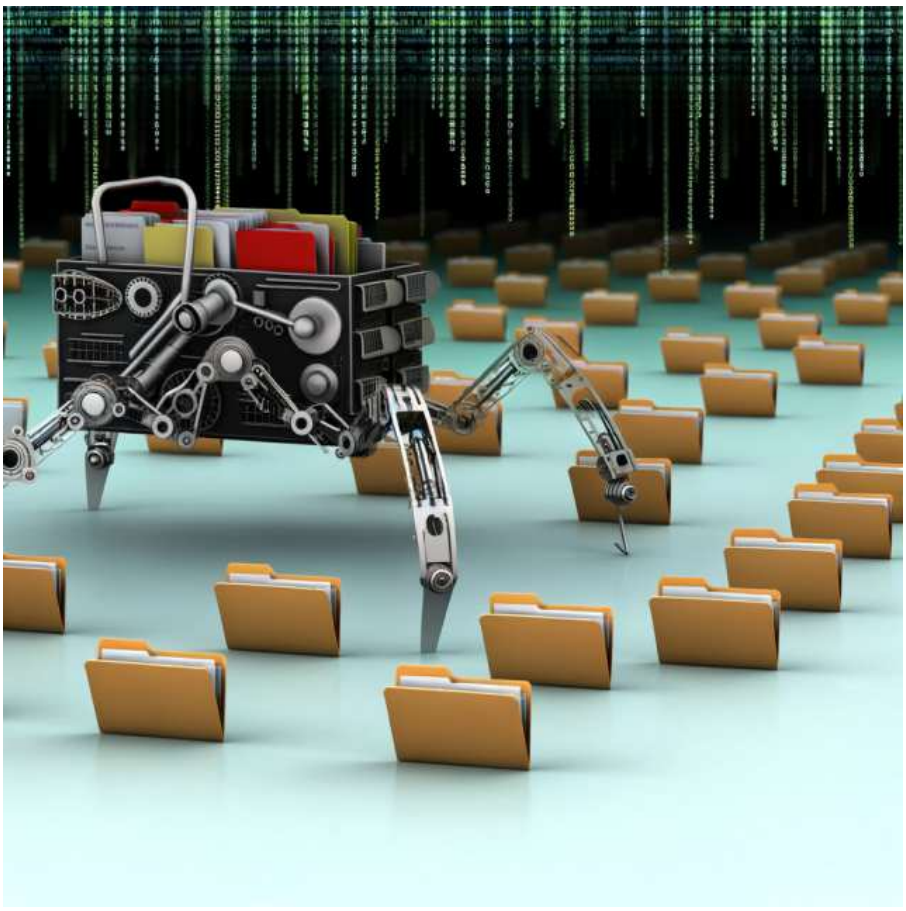


Figure 4.13: "One more folder named "temp_final" today and I'd have enough!". Robotic folder crawler. Image generated with DALL·E

4.4.1.3 Jupyter Notebook Settings

The next thing I do immediately after imports is set my Jupyter notebook to do two things:

1. Display matplotlib plots in the Jupyter notebook (for better exploration)
2. Reload each module automatically before execution (to allow me to modify modules/-packages without needing to re-import them)

   This is done by the function below:

```
# Enable autoreload - that way you don't have to reload modules, when you change them
# Enable matplotlib inline display

```

```
src.utils.start_wrapper.wrapper_notebook_settings()
```

Let's look at the source code:

```
def wrapper_notebook_settings():
    """
    Configures settings specific to IPython/Jupyter notebooks.

    This function checks if the code is running in an IPython/Jupyter
    notebooks session. If it is, it enables autoreloading of modules,
    sets matplotlib to inline mode. If it is not, it prints a message
    indicating that it is not in a notebooks session.

    Note:
        - This function requires the IPython, matplotlib,
        pandas module to be installed.

    Raises:
        None

    Returns:
        None
    """

    try:
        __IPYTHON__
        _in_ipython_session = True
    except NameError:
        _in_ipython_session = False
    if _in_ipython_session:
        from IPython import get_ipython
        get_ipython().run_line_magic("load_ext", "autoreload")
```

```
        get_ipython().run_line_magic("autoreload", "2")
        get_ipython().run_line_magic("matplotlib", "inline")
    else:
        print("Not in IPython/Jupyter notebooks session")
```

I guess the documentation makes it pretty easy to understand what's going on:

1. Checking if we are in IPython (Jupyter Interactive Shell). If not, Jupyter-specific commands will not run.
2. If we are in Jupyter, getting IPython to run plots inline and auto-reload modules.

Done!

4.4.1.4 src.utils.requirements_versions.requirements_versions()

I spoke before about the importance of version and dependency management if you want to get the same results each time you run your code.

This module is a little helper (thanks Grimoire) that creates a version of my *requirements.txt* file with strong specifiers.

In the project notebook, you can find both ways of generating the requirements file:

```
# Check what versions of packages are you using and save them
!pip freeze > './requirements_snapshot.txt'

# Call the function to generate requirements file with strong specifiers
# Needs: 1) all necessary imports done before that cell
# 2) all necessary packages mentioned in /project_folder/requirements.txt
src.utils.requirements_versions.requirements_versions()
```

I mentioned earlier the difference between the output of these two commands:

1. *pip freeze* will extract all packages currently in your environment. This is fine if you just want to look at them, but you cannot just copy and paste all of them into a *requirements.txt* file. This can result in errors. Test and try.

2. *requirements_versions()* will take the requirements.txt file from your main project folder and create a version of that file with strong specifiers (package == version, i.e., "install package EXACTLY in this version").

   So if we have, let's say, the following in (for example) *"app/requirements.txt"* file:

```
# Installing local src package in editable mode
-e .

# Basic packages
numpy
```

Beautifully, *requirements_versions()* will turn this into an *app/notebooks/requirements_versions.txt* file:

```
# Installing local src package in editable mode
-e .

# Basic packages
numpy==1.26.4
```

It will re-write your comments, check your strong specifiers, and keep the blank line. Beautiful.

Why is that useful?

1. There are many versions of packages in a single Python version (for example, 3.12). You want to be sure that you are not operating on two different pandas package versions in the same project - that creates a risk that some underlying functions will work differently. Risk is bad (you can hear those words often in casinos at 5 AM on a Tuesday morning).
2. You don't need to pick and choose packages that you care about from the *pip freeze* output.
3. Semi-automated function saves *requirements_versions.txt* file at the end. No biggie.

You could do it manually yourself. However this takes time & I don't trust myself (Why do I know this stuff about casinos? I can't be a good person!). Last step is to just switch the original *requirements.txt* file with the new one, which you can also automate (if you trust yourself enough - this can be your homework).

The exact code is here and, of course, also on the GitHub repository. I'll skip the explaining part as I did it already above, and there is a nice AI-generated explanation in the function docstring.

src.utils.requirements_versions.requirements_versions module code:

```
import os
import pkg_resources

def requirements_versions(output_file="requirements_versions.txt"):
    """
    Generates a custom requirements.txt file with versions for packages
    listed in a given requirements file. Directly includes comments,
    editable installs, and blank lines. Replaces any existing version
    specifier with the installed version.

    Args:
    - output_file (str): Path to the output requirements_versions.txt
    file.

    Returns:
    None

    Raises:
    None
    """

    source_requirements = os.path.join(
        os.path.dirname(os.getcwd()), "requirements.txt"
```

```
    )
    output_file = os.path.join(os.getcwd(), output_file)
    requirements = []

    with open(source_requirements, "r") as file:
        for line in file:
            clean_line = line.strip()
            # Handle blank lines by appending them directly
            if not clean_line:
                requirements.append("")
                continue
            # Directly rewrite comments and editable installs
            if clean_line.startswith("#") or clean_line.startswith("-e"):
                requirements.append(clean_line)
                continue

            # Split line at the first occurrence of '==' or ' ' or
            # comments to handle version specifiers or options
            package_part = (
                clean_line.split("==")[0].split(" ")[0].split("#")[0].
strip()
            )
            if package_part:
                try:
                    # Retrieve the installed version of the package
                    version = pkg_resources.get_distribution(
                        package_part
                    ).version
                    # Append the package and its installed version
                    requirements.append(f"{package_part}=={version}")
                except pkg_resources.DistributionNotFound:
                    print(f"Package {package_part} not found. Skipping...
```

```
")
54          else:
55              # This handles the case where there's a comment after an
    empty
56              # package name
57              if clean_line.endswith("#"):
58                  requirements.append(clean_line)
59
60  # Write the requirements to the output file, including comments,
    editable
61  # installs, and blank lines
62  with open(output_file, "w") as f:
63      for requirement in requirements:
64          # Preserve blank lines in the output file
65          if requirement == "":
66              f.write("\n")
67          else:
68              f.write(f"{requirement}\n")
69  print(f"Custom requirements.txt generated successfully at {
    output_file}")
```

4.4.2 Analytical Setup - Analysis Parameters

Figure 4.14: "The same parameter is declared in 15 places in the notebook with different values, yummy!". Robo-cat untangling a ball of string. Image generated with DALL·E

Time to untangle this ball of string.

From my experience in the data science world, there is a certain rush to deliver projects quickly and it seems that you have to adjust your code on the fly, have a lot of ad-hoc coding, etc.

You are right. It only seems like that.

More often than not, when I sacrificed strategy and thoughtful implementation for speed of execution, it backfired.

Backfires can come in many shapes and forms:

- You go back to your code a month later and have no idea what you did there.
- You are not able to recreate the same results on the same training data.
- Knowledge gathered during project execution is lost and has to be redeveloped.
- If your model has to be overseen by a model validation department for regulatory reasons, the chances of passing decrease.

So my advice to you, Dear Reader, is: even if they force your hand, try your best to do things right. Push back. Fight the power. It's in your best interest.

To achieve that, plan carefully for your parameters section. This should consider all variables that are important for your analysis and allow you to perform similar analyses without diving into the vast sea of code.

My example of parameters section:

```
# Stock name in stooq.com required format
stock = "aapl.us"
start_date = "1984-09-07"
end_date = "2024-03-06"
# Number of days to forecast
forecast_days = 7

# Prophet (Hi Mark) forecast model name
model_name = "m001_tg_appl"
# Random seed for reproducibility of results in stochastic processes
seed = 123
np.random.seed(seed)
```

And this tells me a story. I will take Apple Inc. stock data (hi Tim) from a start date nearly 4 years after the December 12, 1980 IPO date, as per Apple's website (since stooq.com, from which I am downloading the data, doesn't have full historical data), until some point in March 2024. I will try to forecast the next 7 days of stock prices and will save my forecasting model as the *m001_tg_appl* object.

Why do I specify both the start date and end date? Because if, for whatever reason, I

were to receive more historical data, it might change the results. So both starting and ending points are crucial for the analysis. Data change = results change.

And then I will plant a seed of plant 123. No, it's actually not connected to agriculture.

Seed controls processes involving randomness like sample selection and confidence threshold estimation. If I specify the seed as a constant number, it will allow me to, for example, generate a random sample that will be the same each time.

Random results, every time the same. Makes sense, right?

It's a very important concept that allows us to calculate uncertainty in a predictable way. If you select a sample differently each time for model estimation, you could get slightly different results each time. This would also lead to slightly different performance on the test group, making it impossible to reproduce the results. Not the perfect scenario.

Where is the uncertainty in our case? Well, for forecasting you don't shuffle the sample, but for prediction, you involve statistics to calculate upper and lower uncertainty intervals. This has a random component to it.

And because I will try to prove that the model we estimate and later save and reload will be the same model with the same performance, I need to control this randomness.

Tame the power of randomness.

Also, this is a good place to assign your secret variables from the .env file. In particular instances, you can choose to call them directly from the .env file. Either way, it's better than having passwords and API keys in plain sight.

Here's me, violating what I just said and printing the environment file to the IDE(Integrated Development Environment):

```
api_key = os.environ['secret_apiKey']

# Kids, don't do this at home
os.environ['secret_apiKey']
```

And here's the result (please have some human decency and don't use this password together with my login tomasz@strongpasswords.com. I'm using this one for all my financial, email, healthcare, and social media services).

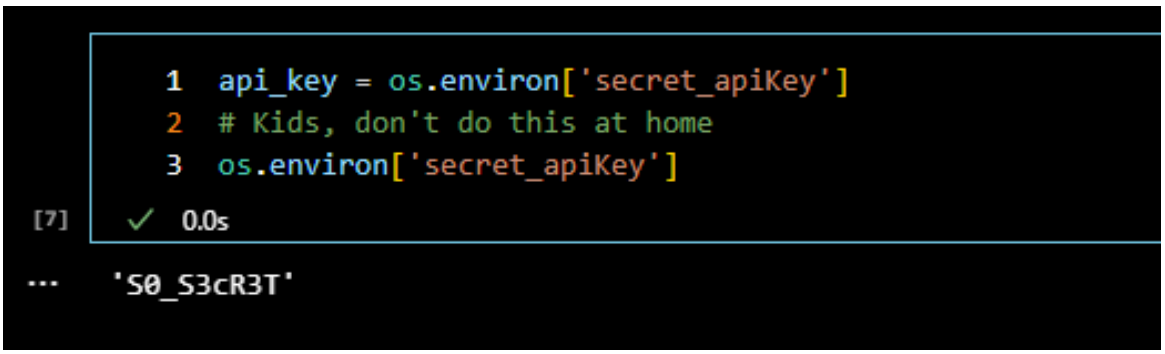


Figure 4.15: Printing environment variables in Visual Studio Code. Screenshot from yours truly.

Tip time: If you want to create unique API keys or passwords, try Nano ID CC. Outside the coding world, you can also use password managers like Norton Password Manager or KeePass.

Ok, your project is set up. Now let's jump to the execution part.

4.4.3 Data Import & Inspection

As "data science" indicates by its name, you will need some "data" to actually do the "science." So let's get some.

4.4.3.1 Import

Let's obtain raw data from Stooq.com and save it in the *"app/data/raw"* folder.

```
# Load data from Stooq using user-written function
data_in = src.data.make_dataset.stooq_import(stock, paths["data_raw"])

# Save data to csv
data_in.to_csv(paths["data_raw"] + stock + ".csv", index=False)
```

Behind the scenes, we have the *stooq_import* function, which downloads data from Stooq.com (what a coincidence):

```python
import datetime as dt
import os
import pandas as pd

def stooq_import(ticker: str, path: str):
    """
    Fetches historical stock data from Stooq for a given ticker. If data is
    not present locally, it downloads the data and saves it as a CSV file in
    the specified folder. If the data is already present, it loads the data
    from the CSV file.

    Args:
        ticker (str): Ticker symbol of the stock.
        paths (dict): Dictionary containing the path to save the raw data.

    Returns:
        pandas.DataFrame: DataFrame containing the historical stock data.
    """
    # Generate the file path
    file_path = os.path.join(path, ticker + ".csv")

    # Check if the file exists
    if os.path.isfile(file_path):
        print(f"Loading data from {file_path}")
        data = pd.read_csv(file_path)  # Load data from CSV file

    else:
        print(f"Fetching and saving data for {ticker}")
```

```
        url = f"https://stooq.com/q/d/l/?s={ticker}&i=d"
        data = pd.read_csv(url)  # Fetch data
        # if you're reading this carved out in stone in front of a cave,
        # I am sorry.
        # You have to be connected to the Internet to download data.
        data.columns = [col.lower() for col in data.columns]
        data.to_csv(file_path, index=False)  # Save data as CSV file

    data["date"] = pd.to_datetime(data["date"])
    return data
```

My ticker is mentioned before as *aapl.us*, which means I am downloading Apple data and saving it in the raw data folder. I have the stock specified earlier as a parameter, so I don't need to retype *aapl.us* here; my program will call the value of this key from the parameters section.

A couple of important things here:

1. *f"https://stooq.com/q/d/l/?s={ticker}&i=d"* is called an f-string. It allows me to pass a variable value (ticker) to the string. You can read more about it here.
2. I'm changing all column names in the data to lowercase for tidiness (refer to PEP 8 - Styling Guide for Python).
3. Changing date to datetime format is crucial for date-based analysis (like forecasting).
4. Never modify raw data; you'll forget about it and won't be able to recreate the same results afterward.
5. Document all your data changes in some way (code). This is very important for modeling. If your model is supposed to be deployed in production, all variable changes that you applied to the raw data will also have to happen there before scoring.

Now that we have data loaded to our IDE, we can check what's inside.

4.4.3.2 Inspect

Basic dataset information contains:

- Number of rows (observations) and columns (variables)
- Variable types
- Primary key information (level of uniqueness)
- Foreign key information (how to connect the dataset to other available datasets)

While foreign key information might be difficult to obtain without documentation or exploratory analysis, the first three points plus a sneak peek at the first and last 5 observations can be done pretty easily in pandas package.

I have a small helper class for that: *src.utils.df_inspect.DataFrameInspector()*.

Now, you don't have to use classes for this, but I wanted to showcase class utilization with a simple example.

What is a class? A class is something in school that you have to sit through.

No, no, I got it all mixed up.

A class is a blueprint on how to handle an object. You can imagine that a human (object) is a class with different characteristics (oh yeah, RPG references). You can have different skills, quantifiable parameters like strength, or different traits such as eye color or caffeine tolerance.

Our objects also share some common functions like eating, sleeping, and walking.

So if there is a group of objects that you want to treat and/or describe similarly, you can group them into a class.

In my case, my class *DataFrameInspector* specializes in treating *DataFrame* objects. It first assigns the *DataFrame* object to the *DataFrameInspector* class instance and then runs a series of private methods. These are set as private because I don't want to invoke them directly. I just execute all of them in the same step when I create the class instance. The methods (functions) called are:

1. *dataframe.info()* - Basic information about the *DataFrame*.
2. *dataframe.head()* - Prints the first 5 rows of the *DataFrame*.
3. *dataframe.tail()* - Prints the last 5 rows of the *DataFrame*.

4. *dataframe.shape* - Prints a formatted string confirming the shape (rows, columns) of the *DataFrame* (**caution:** this is information repetition, as this information is also displayed by the *dataframe.info()* function. Feel free to modify :)).
5. Identify primary key - Checks if there is any unique column in the *DataFrame*.

Why am I doing this? Because I'm lazy and I want to have a one-liner to inspect *DataFrames* properly. I'm leaving this concept here for you to digest and think over how you can apply it to your tasks.

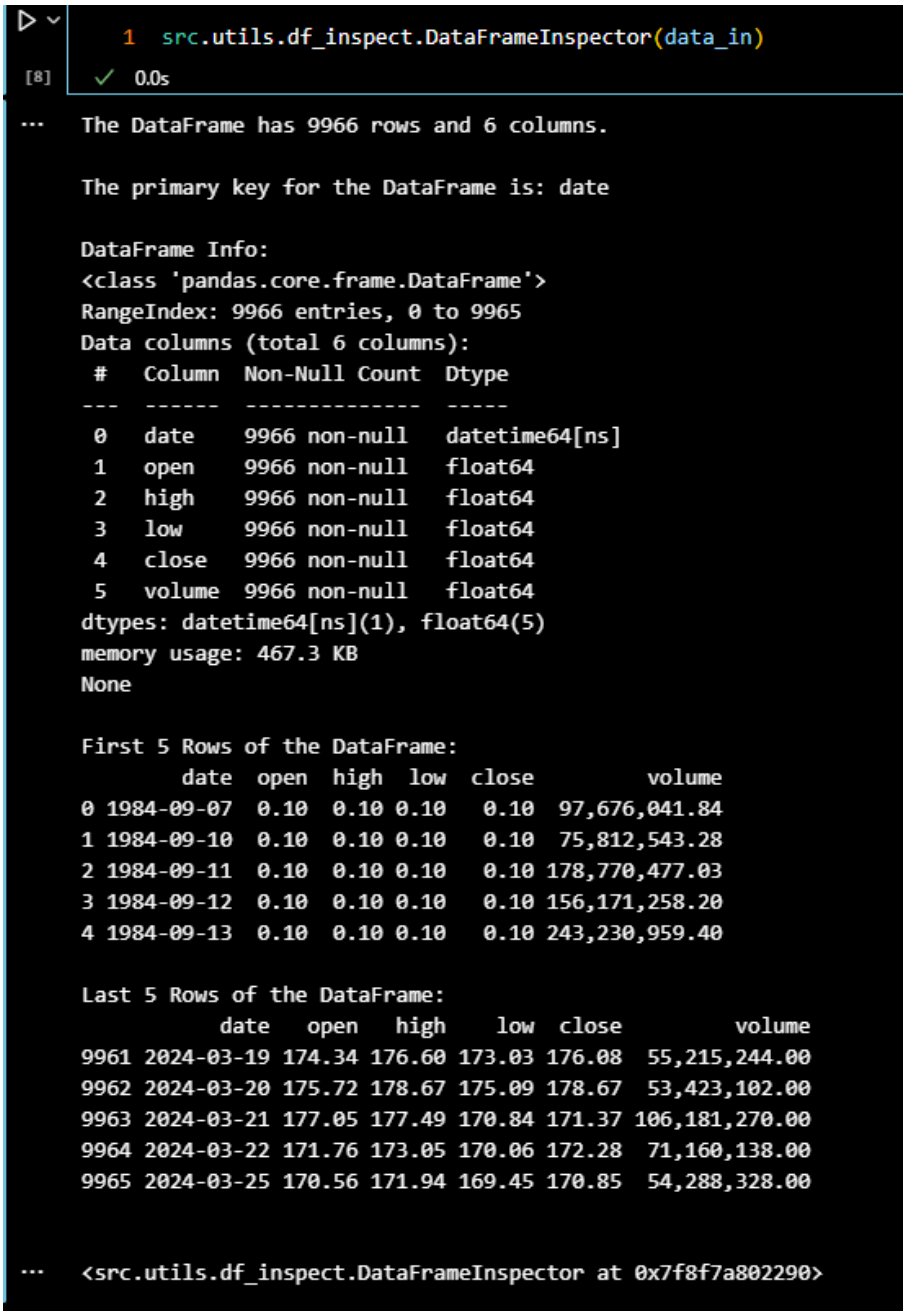
Full code here:

```
class DataFrameInspector:
    """
    A utility class for inspecting and analyzing pandas DataFrames.
    """

    def __init__(self, df: object):
        self.df = df
        self.__get_dataframe_shape()
        self.__identify_primary_key()
        self.__get_dataframe_info()
        self.__get_dataframe_head()
        self.__get_dataframe_tail()

    def __get_dataframe_info(self):
        """
        Prints information about the DataFrame, including column names,
        non-null counts, and data types.
        """
        print("DataFrame Info:")
        print(self.df.info(), "\n")
        return None

    def __get_dataframe_head(self, n=5):
```

```
        """
        Prints the first n rows of the DataFrame.
        """
        print(f"First {n} Rows of the DataFrame:")
        print(self.df.head(n), "\n")
        return None

    def __get_dataframe_tail(self, n=5):
        """
        Prints the last n rows of the DataFrame.
        """
        print(f"Last {n} Rows of the DataFrame:")
        print(self.df.tail(n), "\n")
        return None

    def __get_dataframe_shape(self):
        """
        Prints the number of rows and columns in the DataFrame.
        """
        rows, cols = self.df.shape
        print(f"The DataFrame has {rows} rows and {cols} columns.\n")
        return None

    def __identify_primary_key(self):
        """
        Identifies the primary key column(s) in the DataFrame, if any.
        """
        for col in self.df.columns:
            if self.df[col].is_unique:
                print(f"The primary key for the DataFrame is: {col}\n")
                return col
        print("No primary key found.\n")
```

```
56          return None
```

And this is how it works on our downloaded dataset.



```
[8]   1  src.utils.df_inspect.DataFrameInspector(data_in)
      ✓ 0.0s

...   The DataFrame has 9966 rows and 6 columns.

      The primary key for the DataFrame is: date

      DataFrame Info:
      <class 'pandas.core.frame.DataFrame'>
      RangeIndex: 9966 entries, 0 to 9965
      Data columns (total 6 columns):
       #   Column  Non-Null Count  Dtype
      ---  ------  --------------  -----
       0   date    9966 non-null   datetime64[ns]
       1   open    9966 non-null   float64
       2   high    9966 non-null   float64
       3   low     9966 non-null   float64
       4   close   9966 non-null   float64
       5   volume  9966 non-null   float64
      dtypes: datetime64[ns](1), float64(5)
      memory usage: 467.3 KB
      None

      First 5 Rows of the DataFrame:
              date  open  high  low  close         volume
      0 1984-09-07  0.10  0.10 0.10   0.10  97,676,041.84
      1 1984-09-10  0.10  0.10 0.10   0.10  75,812,543.28
      2 1984-09-11  0.10  0.10 0.10   0.10 178,770,477.03
      3 1984-09-12  0.10  0.10 0.10   0.10 156,171,258.20
      4 1984-09-13  0.10  0.10 0.10   0.10 243,230,959.40

      Last 5 Rows of the DataFrame:
                 date   open   high    low  close         volume
      9961 2024-03-19 174.34 176.60 173.03 176.08  55,215,244.00
      9962 2024-03-20 175.72 178.67 175.09 178.67  53,423,102.00
      9963 2024-03-21 177.05 177.49 170.84 171.37 106,181,270.00
      9964 2024-03-22 171.76 173.05 170.06 172.28  71,160,138.00
      9965 2024-03-25 170.56 171.94 169.45 170.85  54,288,328.00


...   <src.utils.df_inspect.DataFrameInspector at 0x7f8f7a802290>
```

Figure 4.16: Imported dataset description. Screenshot from yours truly.

A couple of takeaways from results:

1. We have nearly 20 years of Apple Inc. stock data.
2. The adjusted close price (why adjusted? I'll come back to that later) on September 7, 1984, was 10 cents. Now it's around 170 dollars. Not a bad investment. I wish I had

put some money in the stock market when I was -5 years old (yes, less than zero).

3. The date is unique and transformed to the datetime type correctly. That's great information - it means I don't have two separate pricings for a single day (I would have to process this information somehow to get what, in our understanding, would be "the truth"), and the data is ready to be used for training a forecast model.
4. We have no null values in the data. That's also a big relief (missing data imputation can be an art by itself). Of course, we don't have data for literally "each day." Why is that? Because we only monitor stock prices on trading days - which in the US will be, on average, 252 days a year (in 2023, there were 250 trading days, for example).

Let's take a deeper look at the *close* variable. Why did I say it's the adjusted close price and not just the close price? On Apple Inc.'s website, you can read: "Apple went public on December 12, 1980, at $22.00 per share. The stock has split five times since the IPO, so on a split-adjusted basis, the IPO share price was $0.10."

Unfortunately, we don't have data from 1980, but from 1984. So let's go further. From the same website, we can read that Apple stock splits happened on the following dates:

- 4 to 1 basis (your 1 stock turns into 4 stocks with price/4) - August 28, 2020
- 7 to 1 basis - June 9, 2014
- 2 to 1 basis - February 28, 2005
- 2 to 1 basis - June 21, 2000
- 2 to 1 basis - June 16, 1987

So, let's see if we have split dates affecting our close variable. I'll look at the 2014 7-to-1 split as the difference in the close price should be the biggest.

```
# Checking whether close price is adjusted for stock splits or not
# Apple had 7:1 stock split in 2014-06-09
data_in[(data_in["date"].dt.year == 2014) & (data_in["date"].dt.month ==
    6)]
```

Let's look at the results:

```python
# Checking wheter close price is adjusted for stock splits or not
# Apple had 7:1 stock split in 2014-06-09
data_in[(data_in["date"].dt.year == 2014) & (data_in["date"].dt.month == 6)]

# Close price is adjusted
```

	date	open	high	low	close	volume
7495	2014-06-02	20.19	20.22	19.83	20.02	413,948,140.85
7496	2014-06-03	20.02	20.34	20.01	20.30	328,311,082.82
7497	2014-06-04	20.30	20.64	20.26	20.54	376,057,124.60
7498	2014-06-05	20.58	20.68	20.47	20.62	340,454,878.36
7499	2014-06-06	20.70	20.74	20.53	20.56	392,650,516.08
7500	2014-06-09	20.67	20.93	20.46	20.89	337,768,178.27
7501	2014-06-10	21.12	21.19	20.86	21.01	281,349,437.34
7502	2014-06-11	20.99	21.13	20.84	20.93	204,682,821.88
7503	2014-06-12	20.96	20.98	20.49	20.58	245,325,569.98
7504	2014-06-13	20.56	20.61	20.26	20.35	244,479,980.59
7505	2014-06-16	20.40	20.68	20.39	20.56	159,410,451.38
7506	2014-06-17	20.58	20.67	20.47	20.53	133,248,823.48
7507	2014-06-18	20.57	20.58	20.37	20.55	150,242,497.20
7508	2014-06-19	20.58	20.58	20.36	20.48	159,263,672.04
7509	2014-06-20	20.48	20.64	20.27	20.27	452,340,593.94
7510	2014-06-23	20.36	20.43	20.20	20.25	195,892,450.90
7511	2014-06-24	20.23	20.45	20.11	20.13	175,049,242.99
7512	2014-06-25	20.11	20.22	19.99	20.15	165,314,739.96
7513	2014-06-26	20.15	20.30	20.02	20.27	146,255,045.60
7514	2014-06-27	20.25	20.51	20.24	20.51	287,105,158.14
7515	2014-06-30	20.53	20.89	20.53	20.72	222,363,372.35

Figure 4.17: Apple 2014 June price listings. Screenshot from yours truly.

Price on the 9th? $20.89. Price on the 10th? $21.01. Ok, it's the adjusted price. Case closed.

4.4.4 Pre-processing

Again, I want my results to be as reproducible as possible. So first, I am adjusting my raw data to be limited to the time window that interests me.

I am using the start date and end date specified in the analytical setup section.

```
# Limit analysis to set-up dates in parameters
data_in = data_in[(data_in["date"] >= start_date) & (data_in["date"] <=
    end_date)]
data_in.shape
```

This action will result in *data_in* having 9953 observations. This should remain constant unless you specify different start/end dates or the underlying raw data changes.

Now I need to:

1. Split my data into two samples:
   (a) Train - the model will learn patterns based on this sample.
   (b) Test - the model will try to apply learned patterns to predict values here.
2. Rename columns for Prophet (the package expects the date to be named *ds* and the target variable to be named *y*).

As this is all historical data, I have access to the ground truth (adjusted close price) and can evaluate the actual model performance.

I am achieving this with the code below, again referring back to the analytical parameters setup.

```
# Train and test split - forecast_days last observations are used as test
    set
train = data_in[["date", "close"]][:-forecast_days].copy()
train.columns = ["ds", "y"]
display("train dataset size:", train.shape)

```

```
test = data_in[["date", "close"]][-forecast_days:].copy()
test.columns = ["ds", "y"]
display("test dataset size:", test.shape)
```

Result? The last 7 observations go to the test sample. The rest stay in the train sample.

```
# Train and test split - forecast_days last observations are used as test set

train = data_in[["date", "close"]][:-forecast_days].copy()
train.columns = ["ds", "y"]
display("train dataset size:", train.shape)

test = data_in[["date", "close"]][-forecast_days:].copy()
test.columns = ["ds", "y"]
display("test dataset size:", test.shape)
```

[11] ✓ 0.0s

```
'train dataset size:'
(9946, 2)
'test dataset size:'
(7, 2)
```

Figure 4.18: Data split to train and test. Screenshot from yours truly.

Done - ready to model.

Figure 4.19: "This econometric modelling stuff isn't so hard after all.". Robots on the catwalk during a fashion show. Image generated with DALL·E

4.4.5 Simple Forecasting Model With Prophet (Hi Mark)

Because I laid all the groundwork beforehand and because Prophet (Hi Mark) is easy to use, I am able to train my first model in a very simple way.

```
m = Prophet()
m.fit(train)
```

Basically, *m* declares an object from the Prophet (Hi Mark) class, and later I am calling the *fit* method on that object to adjust the model to the training dataset.

Then I can use Prophet's (Hi Mark) *make_future_dataframe* method to easily generate future days (test for my model) and adjust available days to only working days (**caution**: this does not necessarily mean trading days! If there is any US public holiday on a working day, I haven't taken this into account. An alternate way is to take dates from the test sample generated earlier.).

```
future = m.make_future_dataframe(periods=2 * forecast_days)
future = future[future["ds"].dt.dayofweek < 5]  # Exclude weekends
future.tail(10)
```

The whole execution looks like this:

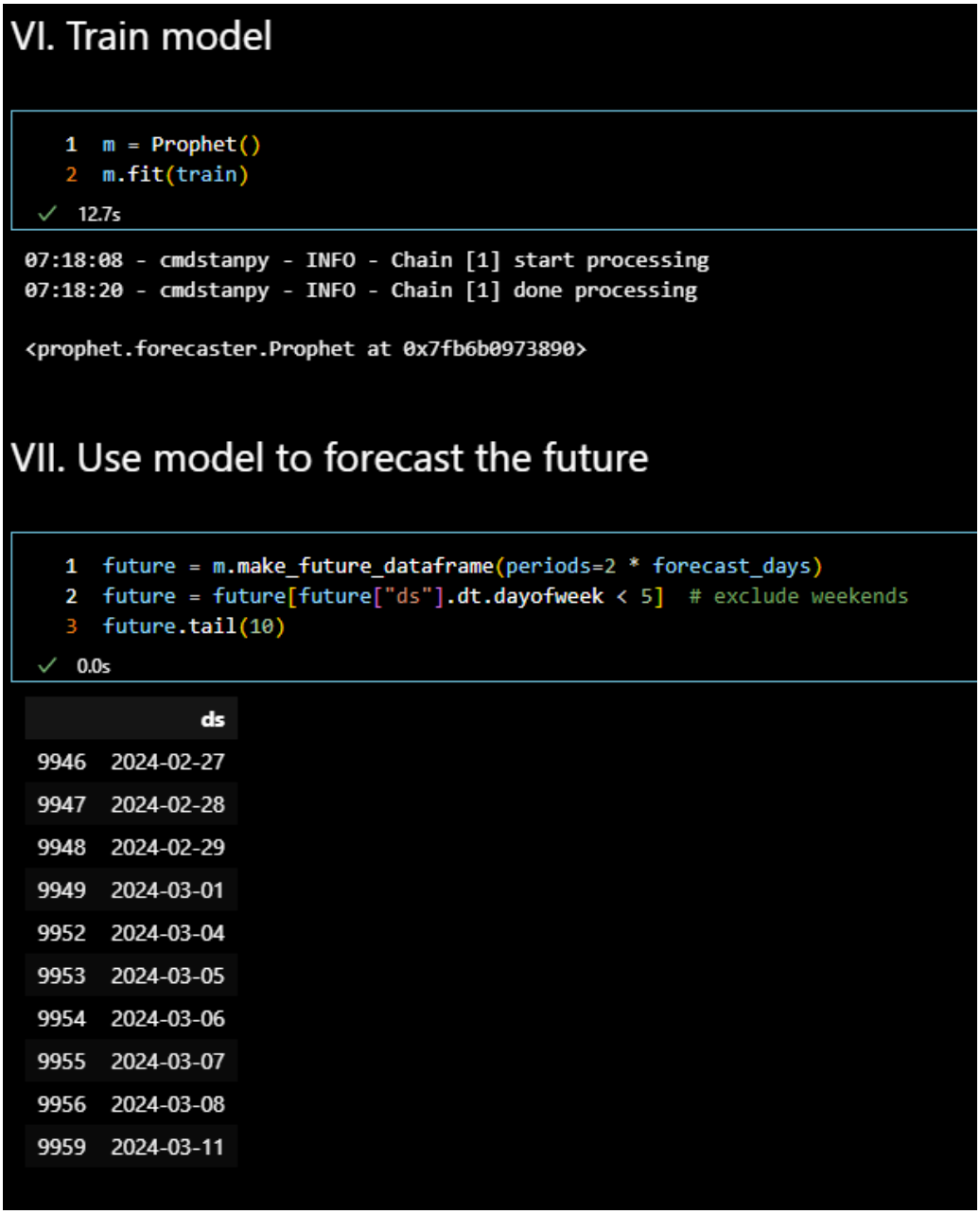
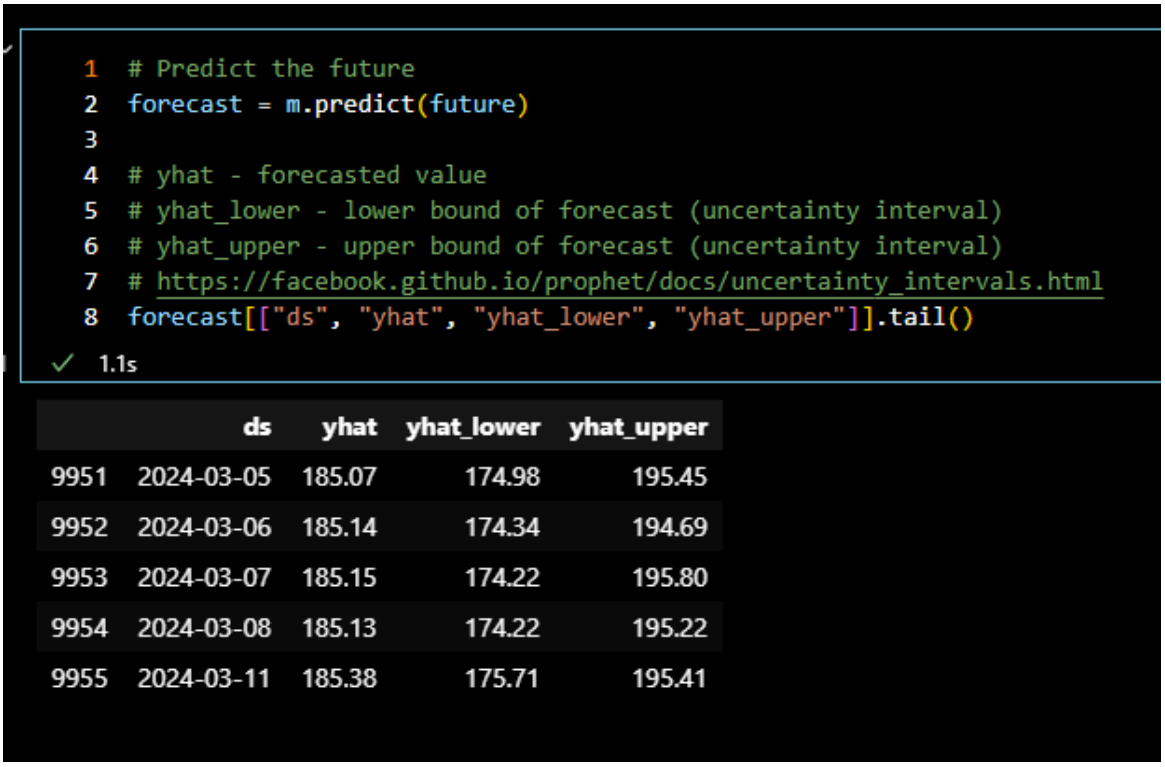


Figure 4.20: Prophet (Hi Mark) model train and generation of future dates. Screenshot from yours truly.

Now, using the *predict* method, I can predict the future!

```
# Predict the future
forecast = m.predict(future)

# yhat - forecasted value
# yhat_lower - lower bound of forecast (uncertainty interval)
```

```
# yhat_upper - upper bound of forecast (uncertainty interval)
# https://facebook.github.io/prophet/docs/uncertainty_intervals.html
forecast[["ds", "yhat", "yhat_lower", "yhat_upper"]].tail()
```



	ds	yhat	yhat_lower	yhat_upper
9951	2024-03-05	185.07	174.98	195.45
9952	2024-03-06	185.14	174.34	194.69
9953	2024-03-07	185.15	174.22	195.80
9954	2024-03-08	185.13	174.22	195.22
9955	2024-03-11	185.38	175.71	195.41

Figure 4.21: Prophet (Hi Mark) prediction and limits. Screenshot from yours truly.

You can look at the new variable definitions and comments from the snippet above.

How good was my prediction? I will actually have to compare the forecasted values against the ground truth (the actual historical observed adjusted closing price) to see if I did any good.

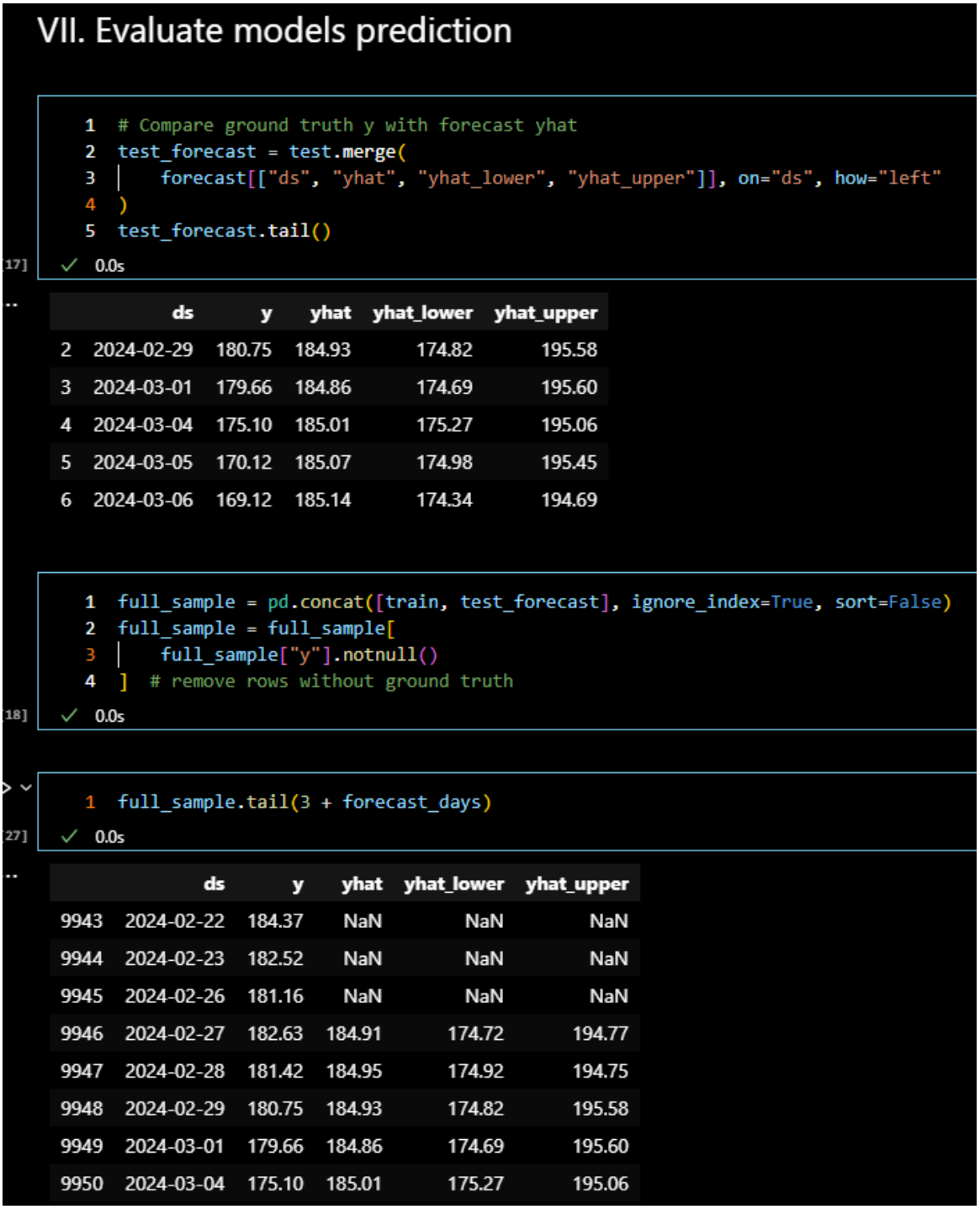
To achieve that, I will merge my forecast and test datasets and later add them to the train dataset (to have a train dataset with only ground truth values, and a test dataset with truth and forecast, as well as the uncertainty intervals).

This can be achieved with the code below:

```
# Compare ground truth y with forecast yhat
test_forecast = test.merge(
    forecast[["ds", "yhat", "yhat_lower", "yhat_upper"]], on="ds", how="
```

```
    left"
)
test_forecast.tail()

full_sample = pd.concat([train, test_forecast], ignore_index=True, sort=
    False)
full_sample = full_sample[
     full_sample["y"].notnull()
]  # Remove rows without ground truth

full_sample.tail(3 + forecast_days)
```

Which in my case produces below results:



	ds	y	yhat	yhat_lower	yhat_upper
2	2024-02-29	180.75	184.93	174.82	195.58
3	2024-03-01	179.66	184.86	174.69	195.60
4	2024-03-04	175.10	185.01	175.27	195.06
5	2024-03-05	170.12	185.07	174.98	195.45
6	2024-03-06	169.12	185.14	174.34	194.69

	ds	y	yhat	yhat_lower	yhat_upper
9943	2024-02-22	184.37	NaN	NaN	NaN
9944	2024-02-23	182.52	NaN	NaN	NaN
9945	2024-02-26	181.16	NaN	NaN	NaN
9946	2024-02-27	182.63	184.91	174.72	194.77
9947	2024-02-28	181.42	184.95	174.92	194.75
9948	2024-02-29	180.75	184.93	174.82	195.58
9949	2024-03-01	179.66	184.86	174.69	195.60
9950	2024-03-04	175.10	185.01	175.27	195.06

Figure 4.22: Prediction vs ground truth. Screenshot from yours truly.

This is not great. While Apple Inc's stock price is falling, my short-term prediction is stable at around $185 per share.

Figure 4.23: "But DontTrustMe667 wrote on the Internet that it's a sure bet!". Sad robotic stock broker. Image generated with DALL·E

Ok, so this time I didn't get it right and I would lose some money on the stock market. It's good that we are in the analysis phase and there are no actual buy orders going out based on prediction results.

If I look at the training data, I can see that the prediction price is revolving around the last 14 trading days (2 * forecast_days parameter value).

```
# It seems that our prediction (yhat) is revolving around
# train dataset's last 14 days average value (y)
np.mean(train["y"][-2 * forecast_days :])
```

```
1  # It seem's that our predition (yhat) is revolving around train dataset's last 14 days average value (y)
2  np.mean(train["y"][-2*forecast_days:])
✓ 0.0s
185.02285714285713
```

Figure 4.24: Average price in last 14 days. Screenshot from yours truly.

I also prepared a simple code to show the prediction accuracy plot:

```
# Plotting
src.visualization.visualize.prophet_plot_forecast(
    data=full_sample, title=stock, obs=forecast_days, y_start_0=False
)
```

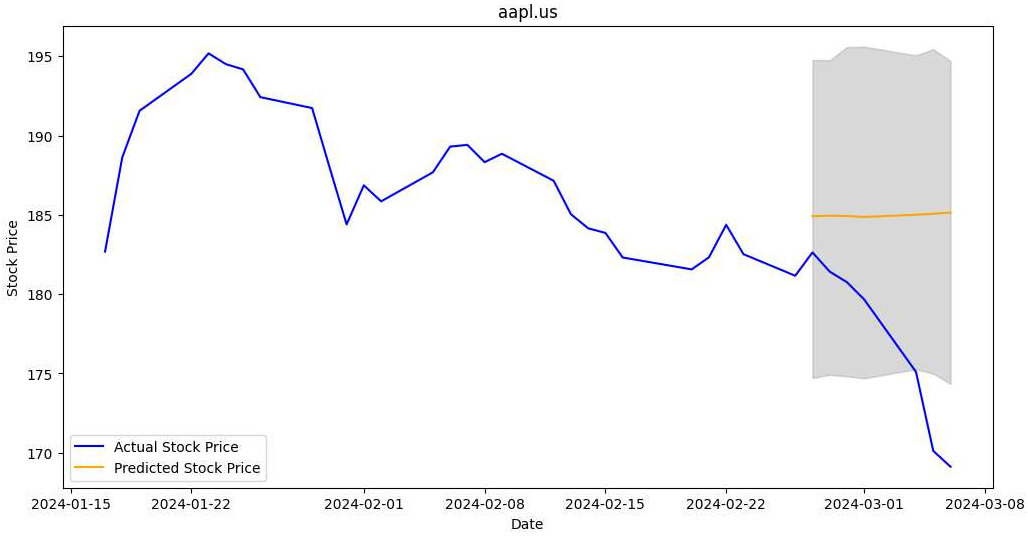


Figure 4.25: Prediction plot on a relative scale. Screenshot from yours truly.

Now this is a great plot to look at when you are doing forecasting because it:

1. Shows our ground truth vs forecast.
2. Distinguishes prediction (test) dates from the train sample.
3. Shows uncertainty intervals (grey area), which actually indicates that the true value went outside the assumed (80% set in the package by default) range of values.

I have only one issue with that plot. It doesn't show the true magnitude of our inaccuracy. Looking at this plot, you might think that the model is highly inaccurate; however, the price range here is only $25.

How would this look if we compared this plot against the true scale, starting from $0.00?

I'll call another function (and explain them both later):

```
# Is it so bad? Let's see true scale
src.visualization.visualize.prophet_plot_forecast_dual(
    data=full_sample, title=stock, obs=forecast_days
)
```

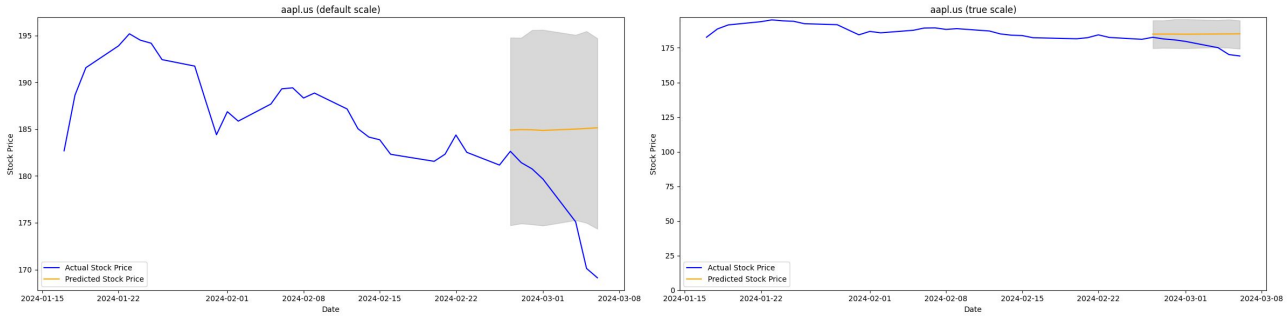


Figure 4.26: Prediction plot true scale vs default scale. Screenshot from yours truly.

Now, if you look at the true scale plot, it looks much less scary, right? I mean, I would still lose money placing a bet on my forecast prices, but overall, I probably wouldn't go broke.

What's the coding behind all of this? Welcome to the wrapper function.

A wrapper function is basically a function that uses other functions' results and builds on them.

In my case, the 'other' function on which I am building is the prediction plot function *src.visualization.visualize.prophet_plot_forecast*:

```
def prophet_plot_forecast(
    data: pd.DataFrame, title: str, obs: int = 5, y_start_0: int = 1, ax=
    None
):
    """
    Plots the actual and predicted stock prices using Prophet (Hi Mark)
    forecasting model.
```

```
7
8       Parameters:
9       - data (pandas.DataFrame): The dataframe containing the stock price
    data.
10      - title (str): The title of the plot.
11      - obs (int): The number of observations to consider for plotting.
12      - y_start_0 (int): If set to 1, Y axis starts with 0. If set to 0,
13      use default axis values.
14      - ax (matplotlib.axes.Axes, optional): The axes on which to plot.
15      If None, uses current.axes.
16
17      Returns:
18      None
19      """
20      if ax is None:
21          fig, ax = plt.subplots(figsize=(12, 6))
22
23      ax.plot(
24          data["ds"][-5 * obs :],
25          data["y"][-5 * obs :],
26          label="Actual Stock Price",
27          color="blue",
28      )
29
30      ax.plot(
31          data["ds"][-5 * obs :],
32          data["yhat"][-5 * obs :],
33          label="Predicted Stock Price",
34          color="orange",
35      )
36
37      ax.fill_between(
```

```
38          data["ds"][-obs:],
39          data["yhat_lower"][-obs:],
40          data["yhat_upper"][-obs:],
41          color="gray",
42          alpha=0.3,
43      )
44
45      ax.set_xlabel("Date")
46      ax.set_ylabel("Stock Price")
47      ax.set_title(title)
48      ax.legend(loc="lower left")
49
50      if y_start_0 == 1:
51          ax.set_ylim(bottom=0)
52
53      if ax is None:  # Show the plot only if this function created the
    figure
54          plt.show()
```

Which is a basic function:

- using the *matplotlib.pyplot* as *plt* package for plotting
- using the *pandas* as *pd* package to process *DataFrame* objects
- taking parameter values to decide how wide the graph should be (5*5 = 25 by default, 5 * *forecast_days* in our case; it shouldn't be too wide for readability purposes) and what the plot title should be
- filling the graph space between *yhat_lower* and *yhat_upper* (uncertainty intervals) with grey color
- based on the *y_start_0* parameter value, showing the true or default scale of the Y-axis (price)

With all this work done, I can have my wrapper function declared as simple:

```python
def prophet_plot_forecast_dual(data: pd.DataFrame, title: str, obs: int = 5):
    """
    Creates a figure with two subplots: the first in default scale, and the
    second with Y axis starting from 0.

    Parameters:
    - data (pandas.DataFrame): The dataframe containing the stock price data.
    - title (str): The title of the plot.
    - obs (int): The number of observations to consider for plotting.
    """
    fig, axs = plt.subplots(1, 2, figsize=(24, 6))

    # First subplot in default scale
    prophet_plot_forecast(
        data, title + " (default scale)", obs, y_start_0=0, ax=axs[0]
    )

    # Second subplot with Y axis starting from 0
    prophet_plot_forecast(
        data, title + " (true scale)", obs, y_start_0=1, ax=axs[1]
    )

    plt.tight_layout()
    plt.show()
```

Which says to the interpreter:

1. Call *prophet_plot_forecast* twice on the same *DataFrame* object.
2. Put them on one image next to each other.
3. Show the left one in default scale and the right one in true scale.

Simple, elegant, and time-saving. Both of these functions expect *DataFrames* adjusted to the Prophet (Hi Mark) forecast format, so for other purposes, adjustments will be needed, but you get the drill, soldier.

Figure 4.27: "Job's done, Sir. The code is beautiful and it caused the production environment to shut down.". Cyborg soldier salutes on the battlefield. Image generated with DALL·E

4.4.5.1 Did I Just Lose Money?

Yes Sir, you did.

But how much exactly? If you bought on 2024-02-27 (assuming you can't buy on the 25th of February, as you already have the data point in your training set, so the day had to pass) and sold 6 trading days later (2024-03-06), you would lose exactly... I won't tell now.

Let's calculate this in Python:

```
buy_price = (
    full_sample[full_sample["yhat"].notnull()]["y"][:1]
    .values.astype(float)
    .item()
)
sell_price = (
    full_sample[full_sample["yhat"].notnull()]["y"][-1:]
    .values.astype(float)
    .item()
)

print("Trade result ($ per share): ", round(sell_price - buy_price, 2))
print(
    "Trade result (pct per share): ",
    "{:.2%}".format((sell_price - buy_price) / buy_price),
)
```

Using some simple *DataFrame* operation and formatted string I can gather that my loss is:

```
1  buy_price = full_sample[full_sample["yhat"].notnull()]['y'][:1].values.astype(float).item()
2  sell_price = full_sample[full_sample["yhat"].notnull()]['y'][-1:].values.astype(float).item()
3
4  print('Trade result ($ per share): ', round(sell_price - buy_price,2))
5  print('Trade result (pct per share): ',"{:.2%}".format((sell_price - buy_price)/buy_price))
✓ 0.1s

Trade result ($ per share):  -13.51
Trade result (pct per share):  -7.40%
```

Figure 4.28: Trade result calculation. Screenshot from yours truly.

Whoa. I knew from the Bible that apples are trouble.

Above is a business interpretation of action based on forecasting results.

From an analytical perspective, for forecasting or regression tasks (trying to predict a continuous value), you would like to track prediction error throughout the whole test period and based on that draw conclusions about model choice (because usually, you would com-

pare a couple of models and choose the best). The most commonly encountered metrics in this space are:

1. MSE - Mean Squared Error
2. RMSE - Root Mean Squared Error - just the root of MSE
3. MAE - Mean Absolute Error

I wouldn't be myself if I didn't write a function for that in my visualize subpackage. By simply calling here:

```
# Prediction error test sample metrics
src.visualization.visualize.plot_forecast_error(
    data=full_sample, title=stock, path=paths["reports_figures"]
)
```

Result:

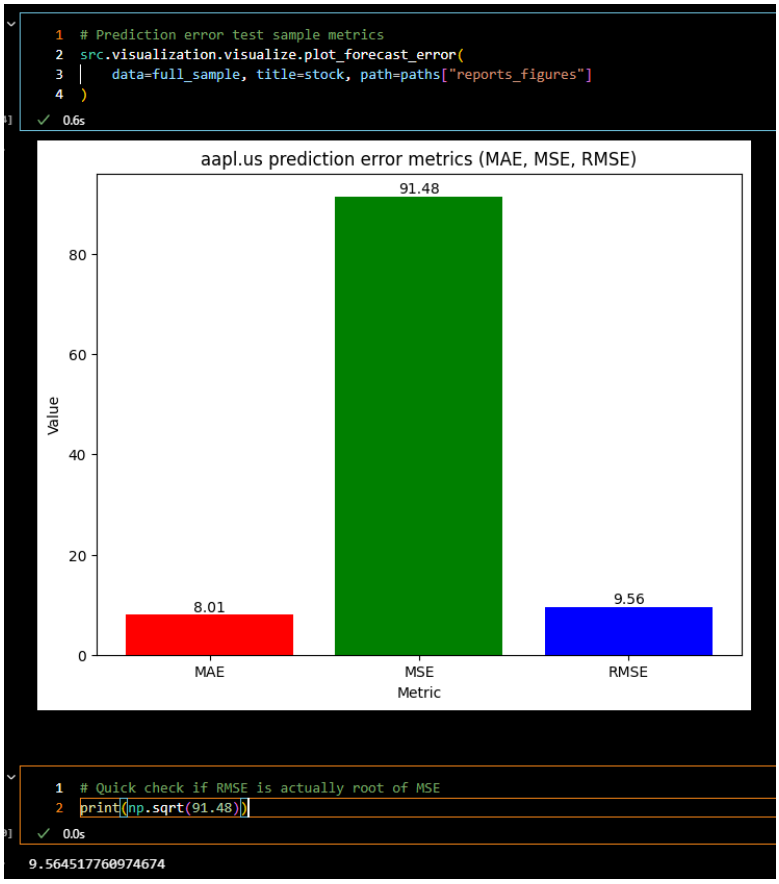


Figure 4.29: Prediction errors bar chart. Screenshot from yours truly.

Our mean absolute error is around $8, but the root mean squared error is around $9.56. RMSE, because of squaring (before rooting), applies more weight to observations with larger differences between prediction and estimation. MSE treats all observations as equal. That tells me that prediction error is unstable - and it's true.

I can see on the chart that Apple Inc.'s price is falling, while the prediction stays nearly the same. So, prediction error increases with time.

Treat this as additional feedback from reality that this time I was not a good prophet (Hi Mark).

Figure 4.30: "Guys, this stock is a sure thing. Trust me!". Robotic prophet on a street stand. Image generated with DALL·E

What's my code behind the function? Let's reveal the secret of *src.visualization.visualize.plot_forecast_error()*:

```python
def plot_forecast_error(
    data: object, title: str, savefig: bool = True, path: str = None
):
    """
    Plots the forecast error metrics (MAE, MSE, RMSE) for a given dataset.

    Parameters:
    data (object): The dataset containing the actual and predicted values.
    title (str): The title of the plot.
    savefig (bool, optional): Whether to save the plot as an image.
    Defaults to True.

    Returns:
    None
    """
    actual = data[data["yhat"].notnull()]["y"]
    predicted = data[data["yhat"].notnull()]["yhat"]

    # Calculate MAE, MSE, and RMSE
    mae = sk_metrics.mean_absolute_error(actual, predicted)
    mse = sk_metrics.mean_squared_error(actual, predicted)
    rmse = np.sqrt(mse)  # RMSE is the square root of MSE

    # Metrics and their names
    metrics = [mae, mse, rmse]
    metric_names = ["MAE", "MSE", "RMSE"]

    # Plotting
    plt.figure(figsize=(8, 6))
    bars = plt.bar(metric_names, metrics, color=["red", "green", "blue"])
```

```
31
32      # Adding the value on top of each bar
33      for bar in bars:
34          yval = bar.get_height()
35          plt.text(
36              bar.get_x() + bar.get_width() / 2,
37              yval,
38              round(yval, 2),
39              ha="center",
40              va="bottom",
41          )
42
43      # Adding labels and title
44      plt.xlabel("Metric")
45      plt.ylabel("Value")
46      plt.title(title + " prediction error metrics (MAE, MSE, RMSE)")
47
48      # Saving the plot
49      if savefig:
50          plt.savefig(path + title + "_error_metrics.png")
51      # Show the plot
52      plt.show()
```

Now quickly, before this knowledge evaporates from my feeble mind:

1. I'm taking from the *DataFrame* only those observations that have both true and predicted values (*y* and *yhat*). It's hard to compare against nothing - nothing always wins.
2. Based on *sklearn.metrics*, called as *sk_metrics* subpackage (you can check it in the subpackage code), I am calculating error metrics.
3. *Matplotlib.pyplot* as *plt* allows me to plot them on a bar plot.
4. I am saving the resulting plot for future reference in the pointed location.

And now, lookie-lookie, I'm saved:

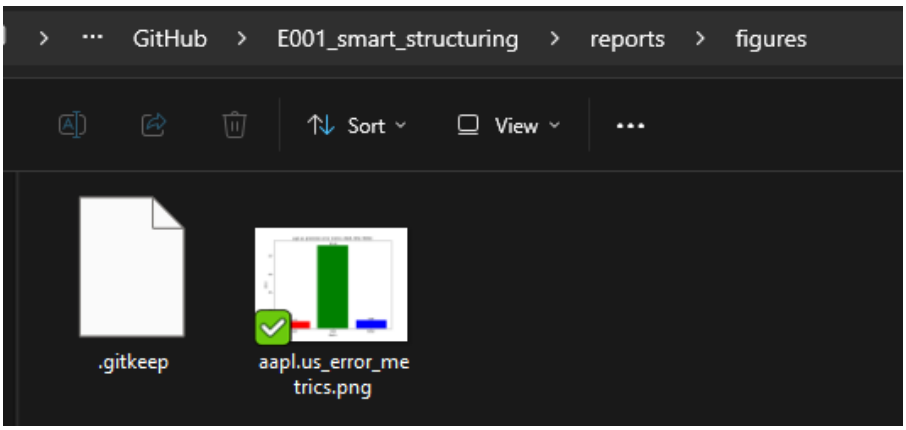


Figure 4.31: Error metrics file in figures folder. Screenshot from yours truly.

So not only will I automatically plot the graph, but I will also save it in the folder for further reference. Some other code can pick up the folder and, for example, put it in the documentation or presentation.

Another beautiful thing is that I don't need to specify many parameters. All of them are picked from previous parts of the code. And that's the mindset that I want to sell: try to plan out your projects so they will be controllable and steerable from the analytical parameters section.

In a more complex scenario, you could plan to launch multiple analyses for multiple stocks by launching this code in separate containers from external parameter .py files or just adjust this code to handle parameters list/dictionary/class.

Sky and your imagination are the limits ;)

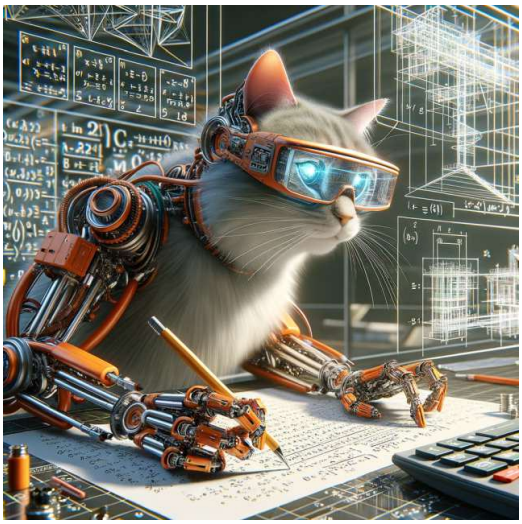


Figure 4.32: "Ok, if I reduce entropy in experiment 021B, I should receive more food on Caturday!". Robotic cat architect doing complex calculations. Image generated with DALL·E

4.5 Saving Model For Future Use

After all this effort, I don't want to re-estimate the model each time on the same data.

I want to save my results and deploy my model somewhere in production to score new data.

Here comes serialization.

Serialization is converting objects to files that can be easily stored and transmitted (like packing your suitcase for a trip).

This would usually be done in one of the following formats:

- CSV (tricky for objects with commas and end-line marks like LF/CRLF)
- JSON (can store unstructured data, human-readable format; popular in APIs)
- XML (can store unstructured data, human-readable format; popular in web services like SOAP or REST and configuration files for software/hardware)
- Pickle (Python-specific binary format; can only exchange information between Python codes)
- YAML (human-friendly standard for configuration files. Often spotted in places where you have to edit configuration files manually - for example, in Docker for docker-compose files)
- Protocol buffer (Google's language-neutral and platform-neutral serialization mechanism; you can send objects from Python code to Java/C/C++ code here)
- Other binary formats (readable by computers with specific software)

For Python, this is often a pickle object, but Prophet (Hi Mark) has its own serialize method that uses JSON.

So let's save my full dataset results somewhere and serialize the model:

```
# Export full dataset
full_sample.to_csv(
    paths["data_processed"] + "forecast_full_sample.csv", index=False
)

```

```
# Python serializing
with open(paths["models"] + model_name + ".json", "w") as fout:
    fout.write(model_to_json(m))  # Save model
```

As I am no longer planning to do anything with my forecast, the results end up in the *"app/data/processed"* directory.

Model on the other hand is saved in the *"/app/models/"* directory.

Files safely arrived. I can also look in the models JSON:

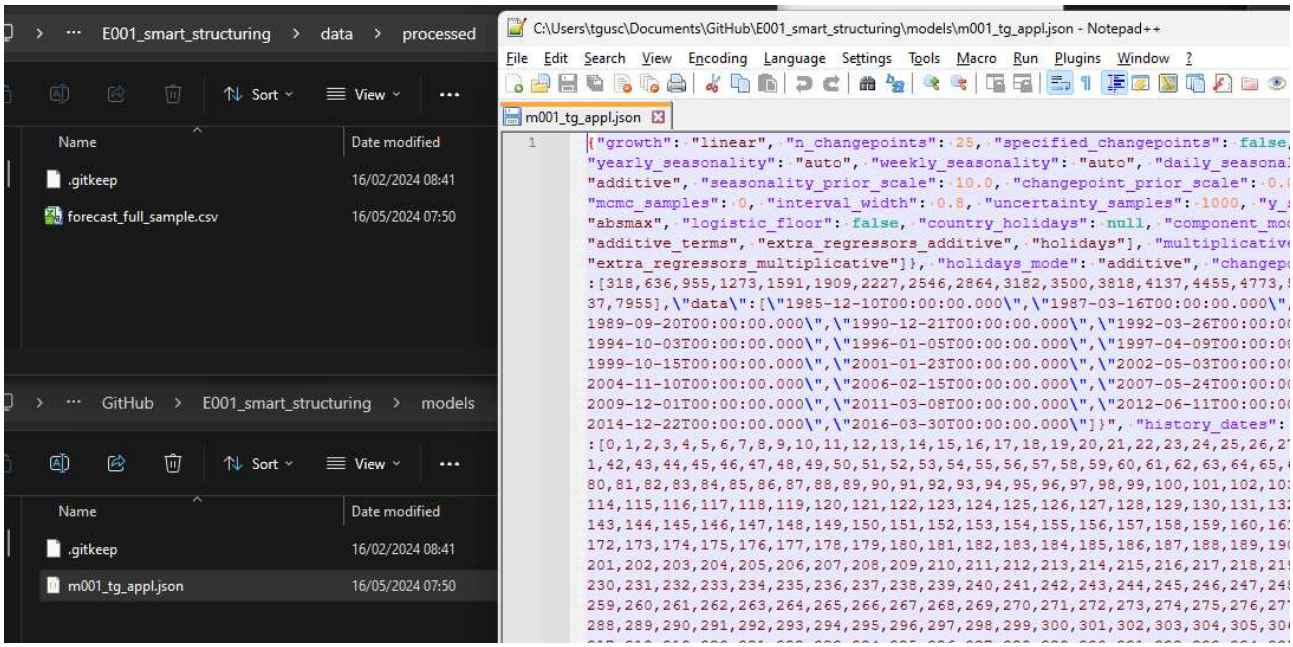


Figure 4.33: Exported files and model JSON content. Screenshot from yours truly.

Now:

- The model can be transported to other projects using Prophet (Hi Mark).
- I have my final dataset available at hand whenever I want to check something (don't you get those frequent thoughts like "I wonder what was Apple stock price on 12 March 1997?").
- If I'm working in a large multi-programming-language team, I might use protocol buffer to send my model to be deployed in some Java or C/C# pipeline.

4.6 Sanity Check - Testing

Last but not least - do you ever wonder if you did it the right way?

Of course not, because you don't make mistakes. You champ.

I don't make them either. I quit drinking 23 times in the last 5 years, and each time was a complete success.

Ok, maybe sometimes I put my meal in the microwave and then leave it there for several hours, completely convinced that I already ate it. Maybe.

But let's go back to the project. It's a good practice to double-check if my code works the way I expect. Because mistakes are bound to happen, companies introduce processes like:

- maker-checker (somebody checks your work after you)
- automated tests (not sure how to explain this to make it even more obvious)
- parallel programming (2 individuals working independently to solve the same problem)

Sometimes you can write correct code, but some underlying package bugs will cause your code to produce wrong results:

- Did you write wrong code? No.
- Are the results wrong? Yes.
- Does anybody care why the results are wrong? No.
- Are you accountable for the results? Yes.

You catch my drift. Even if it's "not your fault," your task is to take ownership and responsibility for the correctness of the results.

So what can I do here? I can actually import the exported JSON model back and check if I get the same results from the model estimated in the project and the exported model.

```
# load model from json
with open(paths["models"] + model_name + ".json", "r") as fin:
    m_test = model_from_json(fin.read())  # Load model

# this is not DRY - but why?
```

```
# you have to set random seed again before every Prophet
# (Hi Mark) prediction - checked empirically
np.random.seed(seed)
forecast_check = m_test.predict(future)

try:
    pandas.testing.assert_frame_equal(forecast, forecast_check)
    are_completely_same = True
except AssertionError:
    are_completely_same = False
print(f"Are the two DataFrames the same? {are_completely_same}")
# confidence intervals yhat_lower and yhat_uppe would be different
# if we wouldnt set np.random.seed(seed) to constant value
```

Exercise is as follows:

1. Read the JSON file and save it as the *m_test* object.
2. Declare a random seed for *numpy* (for results reproduction; for some reason, you have to declare it again before prediction - otherwise the *predict* method can apply different uncertainty intervals).
3. Do a similar future prediction as I did in subchapter 2.4.5, this time using my *m_test* object.
4. Using the *pandas.testing* package, I am checking if both *DataFrames* are equal: if they are, the boolean variable *are_completely_same* is set to *True*. Otherwise, the *assert_frame_equal* function will return an *AssertionError*, in which case the logic will move over to the except scenario and assign *False* to *are_completely_same*.
5. Print the result.

Presto!

```
# load model from json
with open(paths["models"] + model_name + ".json", "r") as fin:
    m_test = model_from_json(fin.read())  # Load model
```
✓ 0.1s

```
# this is not DRY - but why?
# you have to set random seed again before every Prophet prediction - checked empirically
np.random.seed(seed)
forecast_check = m_test.predict(future)

try:
    pandas.testing.assert_frame_equal(forecast, forecast_check)
    are_completely_same = True
except AssertionError:
    are_completely_same = False
print(f"Are the two DataFrames the same? {are_completely_same}")
# confidence intervals yhat_lower and yhat_uppe would be different if we wouldnt set np.random.seed(seed) to constant value
```
✓ 1.2s

```
Are the two DataFrames the same? True
```

Figure 4.34: *Dataframe* equality unit test. Screenshot from yours truly.

Above is an example of a basic unit test. While this term might be one of the least popular in data science, I believe it's one of the more important.

In the modern world, we are increasing the pace of development with AI and using more and more complex analytical tools. While these tools give us great power, without knowledge of how to use them, we can draw bad conclusions.

That's why I firmly believe that for your own satisfaction, sense of purpose, and obviously for proper compensation as a data scientist, you should:

- write good code (or structure your no-code/low-code projects well)
- understand what is happening under the hood
- be able to deliver reproducible results (which supports the notion that you understand how they were obtained in the first place)
- test your assumptions and correct your errors (maybe except me - I don't make erroer-ereus)

Figure 4.35: "Yes, everything went just as I planned! <evil laughter>" Fat robotcat enjoying himself with whisky and cigar. Image generated with DALL·E

Chapter 5: Final words

Farewell. I always loved you.

Oh, sorry, I mixed my notes.

In this lengthy e-book, that could have been an email, I did my best to share what I believe are good coding practices in data science projects:

1. Preparing your environment for your analysis.
2. Having a standardized structure for your projects.
3. Preparing reproducible functions and steps that you can later use in other projects.
4. Keeping your secrets (passwords) safe.
5. Managing dependencies.
6. Keeping your analysis parameters separate.
7. Controlling randomness in random processes.
8. Inspecting your datasets.
9. Keeping raw data untouched and separate from processed data.
10. Testing your code.
11. Using AI tools as support for your coding and documentation.

While the model itself wasn't successful, it was just a brute-force use of default Prophet (Hi Mark) estimation.

Proper time series forecasting is another big story, which I will maybe write about another time (emphasizing maybe). If you are looking for some good materials on this subject, take a look at Time Series Forecasting in Python by Marco Peixeiro.

This was a hell of a ride. I hope you enjoyed reading as much as I enjoyed writing.

And at the end, one more final piece of advice.

5.1 Stay DRY

Contrary to popular belief, it's not about chugging down 7 dry Martinis on a daily basis.

Figure 5.1: "Friday 5PM? It's time to deploy changes to production!". Robot pool party. Image generated with DALL·E

The DRY principle means "Don't Repeat Yourself." It's a software development concept that aims at:

- Reducing repetition of information.
- Minimizing duplication of code.
- Making the codebase easier to understand.
- Reducing errors.

That's exactly what we are trying to achieve by using modules and packages.

On the other hand, you might have noticed that we had to declare *np.random.seed* as *seed* twice. If I hadn't done that, the results of the forecasts would have been different (some method in between forecasts was probably updating *NumPy's* random seed).

And that's another lesson: there is always a fine balance you have to maintain in applying all principles and rules. You have to decide when it's worth it and when it's not.

Also, if I hadn't done *DataFrame* testing, I could have missed the fact that the uncertainty limits can differ.

Like Paracelsus said in 1538 (I remember that, I was standing next to him):

"All things are poison, and nothing is without poison; the dosage alone makes it so a thing is not a poison."

You have the power now. Use it wisely.

And that's all folks!

Figure 5.2: That's all folks! Image generated with DALL·E

P.S.: If you have any questions, comments, or improvement suggestions, please feel free to email me at tomasz@demystifAI.blog.

P.S. 2: Repository with all materials is available on Github.

Bibliography

[1] Adult Swim UK. "Pickle Rick Sewer Escape | Rick and Morty | Season 3 | Adult Swim". Retrieved from https://www.youtube.com/watch?v=Hj5z5AohDec

[2] Apple Inc. "Apple.com website". Retrieved from https://www.apple.com/

[3] Apple Inc. "Apple Inc. Investor Relations FAQ". Retrieved from https://investor.apple.com/faq/default.aspx

[4] Cambridge dictionary. "Utility". Retrieved from https://dictionary.cambridge.org/dictionary/english/utility

[5] Choosealicense.com. "Licenses". Retrieved from https://choosealicense.com/licenses/

[6] DemystifAI. "Create Python development environment in Docker - a step by step guide for Windows". Retrieved from https://demystifai.blog/2024/01/24/post.html

[7] Diagrams: Show Me GPT. "Diagrams: Show Me GPT Assistant". Retrieved from https://chat.openai.com/g/g-5QhhdsfDj-diagrams-show-me

[8] Diagrams: Show Me GPT. "Diagrams: Show Me GPT Assistant conversation". Retrieved from https://chat.openai.com/share/3548018f-46d3-47bb-92ff-bc7146f9c9f1

[9] Docker. "What is a container?". Retrieved from https://www.docker.com/resources/what-container/

[10] DrTypewriter. "Speed Typing Test (Halda Star Typewriter)". Retrieved from https://www.youtube.com/watch?v=oxN1C2QQUIE

[11] Drivendata Github repository. "Cookiecutter Data Science". Retrieved from https://github.com/drivendata/cookiecutter-data-science

[12] Fast Company. "Obama's Presidential Productivity Secrets". Retrieved from https://www.fastcompany.com/3026265/always-wear-the-same-suit-obamas-presidential-productivity-secrets

[13] Game of Thrones. "Game of Thrones Season 5: Episode #10 - Cersei's Walk of Atonement (HBO)". Retrieved from https://www.youtube.com/watch?v=M-9u6msqJNo

[14] GeeksforGeeks. "Formatted string literals (f-strings) in Python". Retrieved from https://www.geeksforgeeks.org/formatted-string-literals-f-strings-python/

[15] GeeksforGeeks. "Private Methods in Python". Retrieved from https://www.geeksforgeeks.org/private-methods-in-python/

[16] GitHub Copilot. "GitHub Copilot". Retrieved from https://github.com/features/copilot

[17] Grimoire GPT. "Grimoire ChatGPT Assistant". Retrieved from https://chat.openai.com/g/g-n7Rs0IK86-grimoire

[18] Guns N' Roses. "Paradise City". Retrieved from https://www.youtube.com/watch?v=Rbm6GXllBiw

[19] IPython. "Package documentation". Retrieved from https://ipython.readthedocs.io/en/stable/

[20] Jupyter. "Package documentation". Retrieved from https://docs.jupyter.org/en/latest/

[21] Jupyter Widgets. "Package documentation". Retrieved from https://ipywidgets.readthedocs.io/en/stable/index.html

[22] Jvelezmagic Github repository. "Cookiecutter Conda Data Science". Retrieved from https://github.com/jvelezmagic/cookiecutter-conda-data-science

[23] KeePass. "KeePass". Retrieved from https://keepass.info/

[24] King Production Studios. "Thats all folks! Looney Tunes". Retrieved from https://www.youtube.com/watch?v=0FHEeG_uq5Y

[25] Longwood Research Data Management - Harvard. "File Naming Conventions". Retrieved from https://datamanagement.hms.harvard.edu/plan-design/file-naming-conventions

[26] Madonna. "Like A Virgin". Retrieved from https://www.youtube.com/watch?v=s__rX_WL100

[27] Marco Peixeiro. "Time Series Forecasting in Python". Retrieved from https://www.manning.com/books/time-series-forecasting-in-python-book

[28] Matplotlib. "Package documentation". Retrieved from https://matplotlib.org/

[29] MIT Broad Research Communication Lab. "File Structure". Retrieved from https://mitcommlab.mit.edu/broad/commkit/file-structure/

[30] MontrealExpos1969. "Fucking, Magnets how do they work?". Retrieved from https://www.youtube.com/watch?v=SjgOnYytIEE

[31] Notepad++. "Downloads". Retrieved from https://notepad-plus-plus.org/downloads/

[32] Norton. "Norton Password Manager". Retrieved from https://support.norton.com/sp/en/us/home/current/solutions/v127051054

[33] Numpy. "Package documentation". Retrieved from https://numpy.org/

[34] OpenAI. "ChatGPT". Retrieved from https://chatgpt.com/

[35] OpenAI. "DALL·E". Retrieved from https://openai.com/index/dall-e-3/

[36] Package Installer for Python (pip). "Requirement Specifiers". Retrieved from https://pip.pypa.io/en/stable/reference/requirement-specifiers/

[37] Pandas. "Package documentation". Retrieved from https://pandas.pydata.org/

[38] Pandas. "Pandas.testing.assert_frame_equal documentation". Retrieved from https://pandas.pydata.org/docs/reference/api/pandas.testing.assert_frame_equal.html

[39] Perplexity.AI. "AI-powered search assistant". Retrieved from https://perplexity.ai

[40] Plotly. "Package documentation". Retrieved from https://plotly.com/python/

[41] Pre-commit. "Package documentation". Retrieved from https://pre-commit.com/

[42] Project GitHub repository. "E001 Smart Structuring". Retrieved from https://github.com/TGusciora/E001_smart_structuring

[43] Prophet (Hi Mark). "Package documentation". Retrieved from https://facebook.github.io/prophet/

[44] Python. "PEP 8 - Styling Guide for Python". Retrieved from https://peps.python.org/pep-0008/#function-and-variable-names

[45] Python-dotenv. "Package documentation". Retrieved from https://saurabh-kumar.com/python-dotenv/

[46] Python Package Index (PytPI). "ruff package index". Retrieved from https://pypi.org/project/ruff/

[47] Ruff. "Ruff documentation website". Retrieved from https://docs.astral.sh/ruff/

[48] Ruff. "Ruff rules documentation". Retrieved from https://docs.astral.sh/ruff/rules/

[49] Scikit-learn. "Package documentation". Retrieved from https://scikit-learn.org/stable/

[50] Setuptools. "Package documentation". Retrieved from https://setuptools.pypa.io/en/latest/index.html

[51] Statistics How To. "Root Mean Squared Error (RMSE)". Retrieved from https://www.statisticshowto.com/probability-and-statistics/regression-analysis/rmse-root-mean-square-error/

[52] Tomasz Guściora Blog. "DemystifAI". Retrieved from https://demystifAI.blog

[53] Tomasz Guściora Github. "Tomasz Guściora profile". Retrieved from https://github.com/TGusciora

[54] Tomasz Guściora LinkedIn. "Tomasz Guściora profile". Retrieved from https://www.linkedin.com/in/tgusciora/

[55] Tomasz Guściora Substack. "DemystifAI Substack". Retrieved from https://demystifAI.substack.com

[56] Tomasz Guściora X/Twitter. "Tomasz Guściora profile". Retrieved from https://twitter.com/TomaszGusciora

[57] Wikipedia. "Mean Squared Error". Retrieved from https://en.wikipedia.org/wiki/Mean_squared_error

[58] Wikipedia. "Mean Absolute Error". Retrieved from https://en.wikipedia.org/wiki/Mean_absolute_error

[59] Wikipedia. "Unit testing". Retrieved from https://en.wikipedia.org/wiki/Unit_testing

[60] Zelark's Github repository. "Nano ID Generator". Retrieved from https://zelark.github.io/nano-id-cc/

Thank You!

Thank you for reading Smart Structuring: A Guide to Efficiently Managing Data Science Projects in Python. If you have any suggestions, questions or want to discuss anything - feel free to connect with me via e-mail: tomasz@demystifAI.blog.

See you on DemystifAI.blog!

ISBN 978-83-972422-0-3