UNI FREIBURG

Python Course

WT 24/25

Lecture: Marcus Rockel & Prof. Dr. Eva Lütkebohmert-Holtz, Tutorial: Felix Häschel
https://www.finance.uni-freiburg.de/

Tutorial 14

Exercise 1 (Remove code duplication)**.**
The macroeconomic exercises on the previous exercise sheets were each standalone but have overlapping functionality, causing repeated code for cleansing, aggregation, and plotting. To resolve this, you will create a **single** class in a new file `data_manager.py` to handle **all file reading** *only once* (including `income_levels.csv` and `oecd_countries.csv`).
Note: Whilst the file reading is still reasonably fast for these small datasets, in general reading and writing files is often the slowest part of data processing, so in general it can be highly beneficial to read each file not more often than necessary.

Once having the datasets in place, you'll distribute the other functionality (cleansing, aggregation, plotting) into their own modules. The final layout will be:

- `data_manager.py`
  Contains a **DataManager** class responsible for:
  - Loading each file *(Excel, CSV, etc.)* exactly once, storing its contents in-memory (e.g. `self.raw_data`).
  - Providing methods (or properties) that **return cleaned data** by calling the appropriate functions from `cleansers.py`.
  - Storing classification data (`income_levels.csv`, `oecd_countries.csv`) in a way that also **prevents re-reading** those files multiple times.
  - Optionally offering methods for merging or pivoting data, so aggregator/visualizer modules remain purely focused on *transforming* or *plotting*.
- `cleansers.py`
  Contains functions for **cleaning** and preparing raw data, such as:
  - Dropping unwanted rows/columns.
  - Renaming headers.
  - Pivoting tables and reindexing (e.g. setting "Year" as index).

  These functions receive `pandas.DataFrame` objects *already loaded* by the `DataManager` (instead of reading the files themselves).
- `aggregators.py`
  Contains functions for **aggregating** data:
  - Summation by region (requires classification info from `DataManager`).
  - Summation by decade or other groupings (time-based or category-based).

These do *not* need to be specific to the three datasets but can be generic functions that accept data (plus classification info in the case of aggregation by region).

- `visualizers.py`
  Contains functions for **plotting** or generating charts.
  - Line plots, bar plots, etc.
  - No file reading here—just direct plotting of the in-memory data returned by `DataManager` or aggregated by `aggregators.py`.

- `main.py`
  Creates a single **DataManager** instance (or more if multiple files). Then:
  - Calls `load_data` methods on the `DataManager` to read each file (e.g., `Carbon_Project_2023_Emiss` `World_Bank_2022_WDI.xls`, `income_levels.csv`, etc.) exactly once.
  - Uses the manager's methods/properties to get back *cleaned* data, then calls aggregator functions (by passing data plus classification info).
  - Calls the plot/visualizer functions to produce final charts.

Goal: By running `main.py`, you should produce the same transformations and final plots as before, but with **no** repeated file reading or duplicated code for cleansing, aggregating, or plotting. The **DataManager** class ensures each file is loaded only once, while each step (cleaning, aggregating, visualizing) occurs in its dedicated module.

Exercise 2 (Company Bankruptcy Classification)**.**
In this exercise, we will work with the Company Bankruptcy Prediction Dataset available on Kaggle. This dataset consists of financial information from various companies and aims to predict whether a company is at risk of bankruptcy.
The dataset includes 96 financial features, such as:

- `ROA(C) before interest and depreciation before interest`, `Current Ratio`, `Debt Ratio`, `Net Income`, etc.
- `Bankrupt?` (0 = not bankrupt, 1 = bankrupt).

The goal of this exercise is to predict whether a company will go bankrupt based on its financial data.

1. Download the dataset from Kaggle and load it into a Pandas DataFrame.
2. Clean the column names by removing leading spaces.
3. Check for and handle missing values if present.
4. Train a **LogisticRegression** model and compare it with a **RandomForestClassifier**.
5. Evaluate both models by printing:
   - Accuracy on the test set.
   - Confusion matrix (**TP, FP, TN, FN**).
   - Classification report (**precision, recall, f1-score**).
6. Interpret your results:
   - Which model performs better?
   - Do false positives or false negatives have a greater impact in a financial context?

Exercise 3 (Cross-Validation and Hyperparameter Tuning)**.**
This exercise builds upon the previous Company Bankruptcy Classification task. We will now focus on improving the performance of the logistic regression model using **cross-validation** and **hyperparameter tuning** techniques.
The dataset is the same as before, containing 96 financial features and the target variable. Optimize the logistic regression model by performing hyperparameter tuning and cross-validation to achieve better predictions.

1. Perform **k-fold cross-validation** to obtain a robust model evaluation.
2. Use **GridSearchCV** to fine-tune hyperparameters for logistic regression.
3. Compare the optimized model's performance to the baseline logistic regression model.
4. Interpret your results.

Exercise 4 (**Optional:** Machine setup)**.**

Consider the machine learning examples in our GitHub repository in the `ml` folder. Make sure they run on your machine including the Keras and Flask examples.
Note: Depending on your setup and machine, you may struggle getting the `tensorflow` library to work (which contains the `keras` library). Don't be discouraged by this but see it as a chance to learn how to handle such version issues ;-).
The `pyproject.toml` contains versions for `tensorflow` and `tensorflow-io-gcs-filesystem` that are known to work with Python 3.11. Consider using a virtual environment with a `requirements.txt` file to install the necessary packages. For the latter recall that you can easily extract the dependencies from `poetry.lock` to a `requirements.txt` file, see the "Project Setup" section in Lecture 9.

Exercise 5 (**Optional:** Flask Serving)**.**

Adjust the `keras_neural_network` example to save the model as a file. Take the `flask_serving` script from our GitHub repository and adjust it to load the model from the file and serve it via a Flask API. Once you start the Flask server, you should be able to view it in your browser (the link is provided in the terminal output when running).
Take the following curl command to send a POST request to the Flask server and receive a prediction. Paste it in your terminal and replace `<flask_server_url>` with the actual URL of your Flask server.

```
curl -X POST -H "Content-Type: application/json" -d "{\"input\": [3, 1, 22.0, 7.25]}"
↪  <your_flask_server_url>/predict
```

The exercise is successful if you receive a prediction as a response.
Note: This exercise mimics the real-world development process of deploying a machine learning model as a web service.