1 Objectives

In this lab, students will learn how to employ the knowledge about data scraping (or crawling) obtained from theoretical lectures to scrape an online repository. This practice is expected to help you convert that knowledge into useful skills about engineering aspect during your data science journey. However, do not think that this lab as well as others will cover all the necessary skills you should know before joining realistic projects, hence you must continue learning even after the completion of this course!

2 Description

This year, you will scrape the open repository of open-access papers that is popular known as ARXIV. It is the most popular and the largest corpus of open-access scientific resources that can be compared to WIKIPEDIA of the common knowledge domain. Scraping the papers from arXiv has already been a known practice for data engineering courses, though most of them exclusively focus on metadata rather than the full-text resources. Therefore, in this milestone, you will get ahead of them in several steps where the scraped data will focus on the full-texts of those papers rather than metadata-only parts, and you should crawl LATEX sources rather than the difficult-to-process PDFs.

2.1 Entry Discovery

The first step in any arXiv crawling workflow is to identify which papers you want to collect. This section introduces three common discovery methods that are used in practice to find relevant paper entries efficiently.

2.1.1 arXiv API for Targeted Discovery

The arXiv API (`http://export.arxiv.org/api/query`) provides an HTTP interface for retrieving information on papers based on search keywords, authors, or categories. It returns results in Atom XML format, which can be parsed programmatically. This method is particularly convenient for targeted queries or daily updates of new submissions.

Key Parameters and Usage The arXiv API accepts several useful parameters:

- `search_query`: Field-specific search (for instance `ti:neural AND cat:cs.AI`)
- `start`, `max_results`: For pagination control
- `sortBy`, `sortOrder`: Sorting criteria such as submission date

The library `arxiv.py` allows flexible interaction with this API. The following short function demonstrates the core concept of querying papers and listing titles:

```python
import arxiv

def find_papers(keyword, limit=5):
    search = arxiv.Search(query=f"all:{keyword}",
                          max_results=limit,
                          sort_by=arxiv.SortCriterion.SubmittedDate)
    return [paper.title for paper in arxiv.Client().results(search)]

# Example usage
titles = find_papers("quantum computing")
print(titles)
```

Listing 1: Example arXiv API querying function

Results are returned as structured objects containing metadata such as title, authors, abstract, and arXiv ID.

2.1.2 OAI-PMH for Metadata Harvesting

The Open Archives Initiative Protocol for Metadata Harvesting (OAI-PMH) (`https://export.arxiv.org/oai2`) enables systematic retrieval of metadata from arXiv. It is the officially supported bulk-access protocol for academic repositories and is well suited for automated metadata gathering.

Main Concepts OAI-PMH uses a set of "verbs" (commands) to request data from the repository:

- `Identify`: Get information about the repository
- `ListRecords`: Retrieve multiple records including metadata
- `ListIdentifiers`: Retrieve identifiers (IDs) only

Practical Example With the `Sickle` package in Python, one can harvest records efficiently through a single query function:

```python
from sickle import Sickle

def harvest_arxiv_records(date_from, date_to):
    sickle = Sickle("https://export.arxiv.org/oai2")
    records = sickle.ListRecords(metadataPrefix='arXiv',
                                 from_=date_from, until=date_to)
    return [r.metadata for r in records]
```

Listing 2: Basic OAI-PMH retrieval function

This example shows how to obtain a structured list of metadata dictionaries that can later be stored as JSON or a database.

2.1.3 Kaggle Dataset for Pre-Aggregated Metadata

The Cornell arXiv dataset available on Kaggle (`https://www.kaggle.com/datasets/Cornell-University/arxiv`) provides pre-collected metadata for over 1.7 million papers. It can be used directly without any API calls and is ideal for local data processing or exploratory searches.

Data Handling and Example The dataset is provided in JSON lines format, where each line represents one paper's metadata. It can easily be loaded into Python for filtering and analysis using `pandas`:

```python
import pandas as pd
import json

def load_arxiv_metadata(file_path, keyword):
    with open(file_path, 'r') as f:
        records = [json.loads(line) for line in f]
    df = pd.DataFrame(records)
    return df[df['title'].str.contains(keyword, case=False)]
```

Listing 3: Reading and filtering arXiv metadata via Kaggle dataset

After loading, students may extract the fields they need such as titles, abstracts, authors, or categories for the selected subset of papers.

2.1.4 Summary

Method	Interface Type	Example Tool or Library
arXiv API	HTTP/XML Endpoint	arxiv.py
OAI-PMH	Metadata Protocol	Sickle
Kaggle Dataset	Static JSON Dataset	pandas

Tabelle 1: Summary of entry discovery methods

2.2 Full-Text Source Download

Once you have identified the list of papers to collect, the next step is to obtain their full source files. The goal here is only retrieving the original submission archives in `.tar.gz` format, not extracting or modifying their contents. arXiv provides several supported methods to do this, each suitable for different programming environments.

2.2.1 arXiv API Download Using Python

The arXiv API endpoint (`https://export.arxiv.org/api/query`) can be used not only for metadata but also to programmatically download each paper's full source archive. The simplest and most common way is through the `arxiv.py` Python library, which wraps the API into user-friendly methods.

Basic Workflow

1. Query the API to find the target paper by its arXiv ID.
2. Use the `download_source()` method to retrieve the archive.
3. Optionally specify a directory and filename for saving.

Here is a short example function showing how to fetch a paper's LaTeX source using `arxiv.py`:

```python
import arxiv

def get_source(arxiv_id, save_dir="./sources"):
    paper = next(arxiv.Client()
                .results(arxiv.Search(id_list=[arxiv_id])))
    paper.download_source(dirpath=save_dir,
                          filename=f"{arxiv_id}.tar.gz")
```

Listing 4: Downloading an arXiv source file with arxiv.py

This function calls the official API endpoint internally, downloads the corresponding `.tar.gz` source archive, and saves it in the designated directory.

2.2.2 Command-Line Download with `arxiv-downloader`

For users preferring direct commands over scripting, the open-source command-line utility `arxiv-downloader` provides a quick way to get both PDFs and sources. Installed via `pip`, it downloads source files using only the paper ID or URL.

Example Usage

```
arxiv-downloader 2302.06544 -s -d ./downloads
```

The flag `-s` ensures that source files are fetched (not PDFs), and `-d` designates the output directory.

2.2.3 Amazon S3 Bulk Repository Access

arXiv maintains complete source archives within an Amazon Web Services (AWS) S3 requester-pays bucket at `s3://arxiv/src/`. This interface is intended for large-scale retrieval, allowing structured downloads of `.tar` bundles containing multiple monthly paper packages.

Setup and Example Command Before downloading, configure AWS credentials with `aws configure`. Then use:

```
aws s3 cp s3://arxiv/src/arXiv_src_2401_001.tar \
          ./bulk/ --request-payer requester
```

Each archive contains multiple submission source files grouped by upload date. You can loop this process to download archives for consecutive months as needed.
Alternatively, `s3cmd` provides a lightweight interface:

```
s3cmd get --recursive --requester-pays s3://arxiv/src/
```

This retrieves all available source archives without manual iteration.

2.2.4 Kaggle Mirror for Public Access

The Cornell arXiv dataset on Kaggle (`https://www.kaggle.com/datasets/Cornell-University/arxiv`) provides a mirror containing metadata entries referencing original arXiv source URLs. While the dataset itself does not contain full-texts, it can be utilized to automate requests for downloading original sources through the `/src/` endpoint.

Integration Example The below minimal code demonstrates how to read the Kaggle dataset and queue download requests for the corresponding arXiv IDs:

```python
import pandas as pd

def queue_sources_from_kaggle(metadata_file):
    data = pd.read_json(metadata_file, lines=True)
    ids = data['id'].dropna().head(3)  # first three IDs
    for arxiv_id in ids:
        print(f"Prepare download for {arxiv_id}")
```

Listing 5: Queueing source downloads using Kaggle metadata

This metadata-driven approach can serve as a scheduling mechanism before making individual API download calls.

2.2.5 Summary

Method	Access Interface	Example Tool
arXiv API	REST via Python client	arxiv.py
Command-Line	CLI utility	arxiv-downloader
Amazon S3	Bulk cloud storage	awscli or s3cmd
Kaggle Mirror	Pre-collected metadata	pandas + API call

Tabelle 2: Available methods for retrieving arXiv source files

2.3 Extracting References

When working with arXiv papers, extracting structured reference information is essential for understanding citation relationships. This subsection introduces the Semantic Scholar API as a method to retrieve references and associated identifiers like arXiv IDs without direct LaTeX parsing.

2.3.1 Semantic Scholar

Semantic Scholar, developed by the Allen Institute for AI, indexes over 214 million papers and provides structured access to citation data through a free API [1]. It excels in linking arXiv papers to broader academic graphs, offering fields such as titles, authors, years, and external identifiers. The API endpoint is `https://api.semanticscholar.org/graph/v1/paper/arXiv:{arxiv_id}`, where queries use the arXiv prefix for identification.

Request Fields and Response To obtain references, specify fields like `references`, `references.externalIds`, `references.title`, and `references.year` in the query parameters. The response is a JSON object containing an array of references, each with potential arXiv IDs in the `externalIds.ArXiv` field.

A simple function to query and extract basic reference data is shown below:

```python
import requests

def get_paper_references(arxiv_id):
    url = f"https://api.semanticscholar.org/graph/v1/paper/arXiv:{arxiv_id}"
    params = {"fields": "references,references.externalIds,references.title"}
    response = requests.get(url, params=params)
    data = response.json()
    return data.get("references", [])
```

Listing 6: Querying Semantic Scholar for references

This retrieves the list of references for a given arXiv ID.

2.3.2 Extracting arXiv Identifiers

From the API response, arXiv IDs are found in the `externalIds` dictionary of each reference. The process involves iterating over the references array and checking for the presence of the `ArXiv` key.

Handling Missing Identifiers Not every reference includes an arXiv ID, as some cite journal articles, books, or non-arXiv sources. Coverage is higher in fields like computer science, where arXiv is prevalent.

2.3.3 Rate Limits and Usage

The API enforces limits of 1 request per second and 100 requests per 5-minute window for unauthenticated use [9]. Exceeding these triggers 429 errors, so implement delays and error handling.

Accuracy Considerations Semantic Scholar achieves over 90% precision in reference extraction through multi-source parsing, including PDFs and LaTeX when available [1]. However, recent uploads may have indexing delays of 1 to 7 days, and version differences might not be distinguished.

Limitations The API may lack data for very new papers, non-English content, or unconventional formats. In such cases, cross-verify with source files.

3 Requirement

In this lab, students will be assigned a specific range of arXiv papers, identified by their unique IDs, to scrape programmatically. For each assigned paper, all available versions must be crawled, including the initial submission and any subsequent revisions. arXiv version IDs follow the convention of appending a suffix "v<version number>" to the base ID (e.g., "2310.12345v1", "2310.12345v2"), and for papers that were submitted once you should suffice it with "v1". The scraping process should retrieve full-text source files (TeX), metadata, BibTeX references, and crawled reference information for each version, ensuring comprehensive coverage of the assigned papers. As the full size of each paper may be very large due to the size of contained figures, you are advised to REMOVE any figure from the scraped papers as we are not going to need it (for your future work :»»).

3.1 Submission Format

3.1.1 Data

For data submission, students must upload their scraped datasets to a designated Google Drive folder provided by the lab instructor. Prepare a local data folder named after your student ID (e.g., "23456789" for student ID

23456789). This folder should contain subfolders, each dedicated to a single arXiv paper from the assigned range, named using the arXiv ID in the “yymm-id” format (e.g., “2310-12345” for ID “2310.12345”). Within each paper’s subfolder, organize the scraped content as follows:

```
<yymm-id>
├── tex
│   └── <yymm-id>v<version>
│       ├── *.tex
│       ├── *.bib
│       └── <subfolders>
│           ├── *.tex
│           ├── *.bib
│           └── ...<recursively following the structure of original TeX sources>
├── metadata.json
└── references.json
```

Where each elements in the folder is explained below:

- A subfolder named `tex` containing all downloaded TeX source files as well as bibtex files for the paper’s versions with the below format. There is no need to modify any internal text content of any TeX source files, you just need to maintain the `.tex` and `.bib` files within the same directory structure as it appears in original TeX source. The bibtex files (extension `.bib`) are directly located from the TeX source as they contain the bib-entries originally uploaded by papers’ authors, thus you should use them without modification rather than construct it either from internal texts of the papers or from external sources such as SemanticScholar, which has been used to create the `references.json` as below. In case of non bibtex file found in the TeX source, it is likely the paper employs the in-line citations with bib-items instead of bib-entries, hence you are allowed to to not have bibtex file in such case.
- A `metadata.json` file storing the paper’s metadata, including at minimum: the paper title (as a string), authors (as a list of strings), submission date (as a string in ISO format), and revised dates (as a list of date strings in ISO format). Additionally, include the publication venue (e.g., journal or conference name) if applicable.
- A `references.json` file containing a dictionary where keys are arXiv IDs (in “yymm-id” format) of the crawled references, and values are their respective metadata dictionaries. Each metadata dictionary must include at minimum: the reference paper title (as a string), authors (as a list of strings), submission date (as a string in ISO format), and SemanticScholar ID (as provided by SemantcScholar APIs). For the papers that do not have associated arXiv IDs found, you should not put them in this file as the IDs are serving as keys.

After preparing the structure, zip the entire student ID-named data folder into a single ZIP file named after your student ID (e.g., “23456789.zip”). Upload this ZIP file to the instructor’s Google Drive folder. Ensure all files are complete and correctly formatted to facilitate evaluation. Of course, NO late uploading is permitted. An example directory for a single paper would be prepared for demonstration.

3.1.2 Source code

For source code submission, include only source files such as Python files (`.py` or `.ipynb`) in the submission to reduce file size while excluding data files, compiled executables, or any other non-source materials. Students must also prepare a brief report detailing the implementation process, including how the scraping works, the tools and libraries used (e.g., arXiv API, Semantic Scholar API, requests library), and relevant statistics such as the number of papers scraped successfully, overall success rate, average paper size in bytes before and after removing figures, average number of references per paper, and average success rate for scraping reference metadata. Besides that, scraper's performance should also be illustrated through two main points: running time (such as total time to process one paper, average time to process each of the required papers, total time for entry discovery, .etc) and memory footprint (such as maximum RAM used during the scraping, average RAM consumption, maximum disk storage required, final output's storage size, .etc). If your system consists of several programs, the runtime is counted as the wall time taken to run the system in an end-to-end way while the memory footprint is considered as total memory consumed by the full system. The testbed for your final benchmarking is a Google Colab instance running on CPU-only mode.

This report must be separate from the README.md file, which should provide clear instructions on setting up the necessary environment (e.g., Python version, required packages via pip or conda) and running the submitted code with specific configurations (e.g., input parameters for arXiv ID ranges, scraping rate, parallelism).

In preparation for code submission, create a folder named after your student ID (e.g., "23456789") containing the following structure:

```
23456789/
|-- src # folder containing the source code
|   |-- *.{py/ipynb}
|   |-- requirements.txt # To re-produce the environment
|-- README.md # Instructions on environment setup and code execution
|-- Report.{docx/tex} # Use your preferred format but only single file
    permitted
```

Zip the entire student ID-named folder into a single ZIP file named after your student ID (e.g., "23456789.zip") and upload it to the Moodle system for evaluation. All the submission above should be in the official language of your program.

Furthermore, students should also make a Youtube video for demonstration purpose where its link is shared through the aforementioned report. Note that, students need to ensure that the video is made publicly viewable and it must be maintained as is for at least 01 month after the course's completion (marked by the announcement of your grades). It is students' responsibility to make sure that, before the end of this period, the video's last modification must not be later than the announced deadline of the source code submission. Violation of any requirements will result in your lab's zero grade. The video should be at most 120 seconds providing demonstration on how to run the submitted source code, its processing and outputs as well as relevant intermediate logs that the students consider significant. In parallel with such demonstration, students should provide voice-based explanation about what is going on in the running process as well as the reasoning behind the scraper's design (as the brief report will not cover this). This video is NOT a proof for your scraper's runtime and/or memory footprint, thus you will not need a very long video just to record the whole running of your program. The voice for this video can be in any language that the students feel comfortable.

4 Grading criteria

Your grade is determined upon five main criteria as listed below:

- Report: Does your report illustrate sufficient information as previously mentioned?
- Source code: Is your code clean, readable and runnable?
- Demo video: Does your video clear and meet the above requirements?
- Resulted data: To what extend has your scraper successfully crawled and how well is your submitted data formatted?
- Scraper's performance: How efficient is your scraper compared to others?

4.1 Regulations

Ensure your code is well-documented with clear comments.

You can use online documents for reference, but you must provide complete citations. You are free to discuss your topics with classmates, but your work must be implemented and interpreted according to your own understanding.

Your course's lab will be graded zero if there is any dishonest behaviour detected.

5 Contact

If you have any further questions for this project, contact the instructors at:

Huỳnh Lâm Hải Đăng: hlhdang@fit.hcmus.edu.vn

or through the previously posted ZALO group. Instructors will answer your questions as soon as possible.

Literatur

[1] Waleed Ammar u. a. *The Semantic Scholar Open Research Corpus*. `https://arxiv.org/abs/1805.02234`. Accessed: 2025-10-25. 2018.

[2] *arXiv API Basics*. `https://info.arxiv.org/help/api/basics.html`. Accessed: 2025-10-25.

[3] *arXiv API User's Manual*. `https://info.arxiv.org/help/api/user-manual.html`. Accessed: 2025-10-25.

[4] *arXiv Bulk Data Access on Amazon S3*. `https://info.arxiv.org/help/bulk_data_s3.html`. Accessed: 2025-10-25.

[5] *arXiv Dataset on Kaggle*. `https://www.kaggle.com/datasets/Cornell-University/arxiv`. Accessed: 2025-10-25.

[6] *arxiv-downloader: Command-Line Tool for arXiv*. `https://github.com/braun-steven/arxiv-downloader`. Accessed: 2025-10-25.

[7] *Open Archives Initiative Protocol for Metadata Harvesting (OAI-PMH)*. `https://info.arxiv.org/help/oa/index.html`. Accessed: 2025-10-25.

[8] Lukas Schwab. *arxiv.py: Python wrapper for the arXiv API*. https://github.com/lukasschwab/arxiv.py. Accessed: 2025-10-25. 2015.

[9] *Semantic Scholar Academic Graph API*. https://api.semanticscholar.org/api-docs/graph. Accessed: 2025-10-25.