UNIVERSITY OF SCIENCE - VNUHCM

FACULTY OF INFORMATION TECHNOLOGY

COURSE MILESTONE 1

CSC14119 – Introduction to Data Science

Student in charge:

Cao Tấn Hoàng Huy - 23127051

Nguyễn Hữu Anh Trí - 23127130

Nguyễn Huy Quân - 23127107

Instructor:

Huỳnh Lâm Hải Đăng

Mục lục

1 Introduction

This report covers the design and performance of a data scraper built for the "Introduction to Data Science" course milestone. The main goal was to practice web scraping by building a functional tool that collects scientific paper data.

The project focuses on **arXiv**, a massive public repository for research papers. Unlike typical scraping projects that only collect basic data (metadata), our system was uniquely designed to get the **full-text LaTeX source files** for all paper versions. This provides a much richer dataset than using PDFs, which is highly valuable for advanced research tasks like text analysis and studying citation networks.

The scraper operates in multiple stages: (1) finding all the assigned papers (Entry Discovery), (2) downloading and cleaning the raw LaTeX source code (Full-Text Source Download and processing), and (3) gathering citation details using the Semantic Scholar API (Reference Extraction).

The rest of this report will detail the tools used, present key scraping statistics, and provide a full analysis of the scraper's speed and resource.

2 Implementation Details

This section details the technical implementation of the scraper, outlining the tools used and the sequential workflow for processing each arXiv paper.

2.1 Tools and Libraries

The scraper is built in Python and relies on several key external libraries and APIs:

- **Libraries:**
  - `arxiv`: The primary library used to query the arXiv API for paper discovery, metadata retrieval, and downloading source files.

  - `requests`: Used to perform HTTP GET requests to the Semantic Scholar API for reference extraction and for downloading files or checking links.
  - `psutil`: Monitors system resources such as memory and CPU usage, enabling performance tracking and optimization.
  - `threading`: Enables concurrent execution of tasks, allowing multiple papers to be processed in parallel for improved efficiency.
  - `queue`: Provides thread-safe task queues to coordinate work between producer and consumer threads.
  - `tarfile`: Extracts `.tar.gz` or `.tgz` source archives downloaded from arXiv, preserving the original directory structure.
  - `re` (Regular Expressions): Used to parse TeX files, remove figure-related commands, and sanitize or extract information from text.
  - `json`: Handles reading and writing of JSON files for metadata and reference storage.
  - `os`: Manages file paths, directory creation, and file system operations.
  - `time`: Used for benchmarking, sleeping between requests, and measuring process durations.
  - `random`: Supports random selection or sampling of papers, and introduces jitter in retry logic to avoid API rate limits.
  - `string`: Assists in sanitizing filenames and dictionary keys to ensure compatibility with file systems.
  - `logging`: Used to suppress or manage log output, especially from verbose libraries such as `arxiv`.

- **APIs:**
  - **arXiv API** [1, 2] : The backend for the `arxiv` library, providing programmatic access to paper metadata and source files.
  - **Semantic Scholar API** [3]: Used to fetch citation data and identify arXiv IDs within a paper's reference list.

2.2 Overall Pipeline

The scraping process is arranged by `main.py`, which executes a sequential pipeline for each assigned arXiv ID. The workflow is modularized into distinct steps, handled by the different Python scripts:

1. **ID Discovery and Validation**

   The pipeline begins by determining the exact set of arXiv paper IDs to process. Since the last valid ID for each month is not known in advance, a binary search strategy is used to efficiently find the first and last available IDs within the assigned range. This ensures that all relevant papers are included and none are missed or duplicated. The resulting list of IDs is then formatted and prepared for the next stages.

2. **Metadata and BibTEX Collection**

   For each paper ID, the system queries the arXiv API to gather all available metadata, including the paper's title, list of authors, submission date, revision history, subject categories, and a cleaned abstract. If available, publication venue and BibTEX entries are also collected. All this information is organized and saved as `metadata.json` (and `references.bib` if present) in the paper's designated subfolder.

3. **Full-Text Source Download and Extraction**

   The system downloads the full source files for every version of each paper (e.g., `v1`, `v2`, etc.), ensuring that all revisions are included. Each version's `.tar.gz` archive is downloaded and safely extracted, with special care taken to avoid problematic files or paths. Only the essential `.tex` and `.bib` files are retained, while all other files are removed to comply with requirements and save storage space.

4. **Citation Network Extraction**

   To build a comprehensive citation network, the system queries the Semantic Scholar API for each paper's arXiv ID. Only references that can be matched to another arXiv paper are included. For each matched reference, key metadata such as title, authors, submission date, and Semantic Scholar ID are collected and saved in a dedicated `references.json` file.

5. **Missing Papers Recovery**

   Throughout the process, the system continuously checks for missing or incomplete papers by comparing the expected list of IDs with the actual folders and files present. Any papers found to be missing or incomplete are automatically re-queued for download and processing, ensuring that the final dataset is as complete as possible.

6. **Benchmarking**
   The system monitors its own performance, recording statistics such as total runtime, time spent on each step, memory usage, and disk space consumed. This benchmarking information is used to optimize the workflow and provides a detailed report on the efficiency and reliability of the scraping process.

This six-step process is repeated for every paper in the assigned range, resulting in a complete, structured dataset that meets all assignment requirements.

3 Pipeline Detail

3.1 Step 1 - Entry Discovery

This initial step is responsible for generating the complete and accurate list of all arXiv paper IDs that the rest of the pipeline will scrape.

3.1.1 The Core Problem

The main challenge is that, although arXiv IDs (e.g., `YYMM.NNNNN`) are assigned sequentially within each month, arXiv does not provide a direct way to discover the ID of the last paper submitted in any given month. While the numeric part of the ID increases monotonically, there may be occasional gaps due to withdrawn submissions or administrative removals. Furthermore, since the arXiv API does not support listing all valid IDs for a month, it is not feasible to simply loop from `00001` to `99999` without making a large number of unnecessary or failed API requests. Therefore, the pipeline must efficiently and accurately determine the *actual* first and last valid ID for each month in the target range, ensuring that only real papers are included in the subsequent processing steps.

3.1.2 The Solution

The solution is a sophisticated search algorithm that efficiently finds the valid ID boundaries for each month. Instead of checking one by one, it uses a two-stage strategy:

1. **Exponential Search:** To quickly find an ID in the correct range, the search index is doubled (e.g., `1`, `2`, `4`, `8`, ...) until a valid (or invalid) paper is found.

2. **Binary Search:** Once a rough range is established, a binary search is used to pinpoint the *exact* first or last valid ID in that range.

3.1.3 Breakdown of the Process

- **`id_exists(paper_id)`:** This is the core "oracle" function. It queries the arXiv API with a single ID. It returns `True` if the paper exists and `False` if it doesn't (or if an error occurs).

- **`find_first_id(year, month)`:** This function finds the *first* valid ID. It uses an exponential search to find *any* valid ID, and then a binary search to find the lowest valid index.

- **`find_last_id(year, month)`:** This function finds the *last* valid ID. It uses an exponential search to find a *non-existent* ID, and then a binary search to find the highest valid index just before it.

- **`get_IDs_All(...)`:** This is the main orchestrator. It loops month by month from the start date to the end date. For the first and last months, it uses the user-provided start/end IDs. For all months in between, it uses `find_first_id` and `find_last_id` to get the complete list of IDs for that month.

3.1.4 Final Output

The result of this step is a single, complete list of all valid arXiv paper IDs within the specified date and ID range, ready to be passed to the next stage of the pipeline.

3.2 Step 2 - Metadata Collection

This step is responsible for fetching all descriptive information for a paper and saving it to a structured `metadata.json` file.

3.2.1 The Core Problem

The `arxiv.Result` object returned by the API is a Python object and not a persistent data format. This data must be parsed, cleaned, and structured according to the assignment's requirements, including extracting all authors, handling version history, and capturing submission/revision dates.

3.2.2 The Solution

The notebook implements a two-function system. One function, `create_metadata`, is responsible for parsing the API object into a clean dictionary. The second function, `save_metadata`, handles the file I/O and saves the dictionary to disk.

3.2.3 Breakdown of the Process

1. **Data Parsing (`create_metadata(paper)`):** This function acts as the core parser. It takes the `arxiv.Result` object and extracts all required fields:
    - It splits the ID (e.g., `2305.00633v4`) into its base ID (`2305.00633`) and latest version (`4`).
    - It formats authors into a simple list of names and correctly formats the submission and revised dates.
    - It generates a list of PDF URLs for *all* versions of the paper, from `v1` to the latest.
    - It extracts the title, abstract, categories, and optional fields like DOI and publication venue (from the comments).
    - It returns a single, clean Python dictionary.
2. **Saving (`save_metadata(paper, folder)`):** This function orchestrates the saving process:
    - It first calls `create_metadata` to get the structured dictionary.
    - It ensures the target save folder (e.g., `.../2305-00633/`) exists.
    - It writes the dictionary to `metadata.json` using UTF-8 encoding and pretty-printing (`indent=4`) for human readability.

3.2.4 Final Output

The result of this step is a single, clean `metadata.json` file placed in the main directory for each paper, containing all of its metadata and version history.

3.3 Step 3 - Full-Text Source Download & Processing

This step fetches the actual LaTeXsource code for every version of a paper, extracts the archive, and cleans it to meet the project's requirements.

3.3.1 The Core Problem

The assignment requires the full LaTeXsource, not just the metadata. This involves two main challenges:

1. **Completeness:** The source code for *all* versions of the paper must be downloaded, not just the latest.
2. **Cleaning:** The raw `.tar.gz` source archives contain many non-text files, such as large images (`.png`, `.jpg`, `.pdf`). These must be removed to isolate the text source and reduce data size.

3.3.2 The Solution

The pipeline implements a robust "Download-Extract-Clean" workflow. This process is repeated for every version of every paper. It uses helper functions to ensure files are extracted safely and that all non-essential files are subsequently deleted.

3.3.3 Breakdown of the Process

1. **Orchestration (`download(...)`):** This main function iterates through the list of `arxiv.Result` objects for a paper (one for each version).
2. **URL Generation & Download:** For each version (e.g., `2304.07856v2`), it:
   - Creates a version-specific subfolder (e.g., `.../tex/2304.07856v2/`).
   - Constructs the direct download URL (e.g., `https://arxiv.org/src/2304.07856v2`).
   - Downloads the `.tar.gz` archive into that folder using the `download_url` helper.
3. **Validation & Safe Extraction:**
   - The script first validates the downloaded file is a genuine `.tar.gz` archive.

- It then uses `safe_extract_tar` to extract the contents. This function is critical for security and stability, as it sanitizes filenames and explicitly skips dangerous files like symbolic links or files with path traversal (`../`).

4. **Content Cleanup (`cleanup_non_tex_bib_files`):** After successful extraction, this function is called. It scans the new folder and **deletes every file** that does not have a `.tex` or `.bib` extension, effectively removing all images and other non-text content.

5. **Finalization:** The original `.tar.gz` file is deleted to conserve disk space.

3.3.4 Final Output

The result is a clean directory structure (e.g., `.../2304-07856/tex/`) containing a subfolder for each paper version. Each subfolder contains *only* the `.tex` and `.bib` source files for that version.

3.4 Step 4 - Citation Network Extraction

This final step builds the `references.json` file by identifying all of a paper's citations that are *also* available on arXiv.

3.4.1 The Core Problem

The arXiv API does not provide citation data. An external source is required. The challenge is to query this source (Semantic Scholar), parse its complex response, and then filter a large bibliography to find *only* the references that are explicitly linked to an arXiv ID, ignoring all other journal or conference citations.

3.4.2 The Solution

The pipeline uses the Semantic Scholar Graph API, which indexes papers and their citations, including `externalIds` like arXiv IDs. The solution involves querying this API, then iterating over the results to filter and restructure the data.

3.4.3 Breakdown of the Process

1. **API Query (`get_paper_references(arxiv_id)`):** This function fetches the raw data.

- It constructs the Semantic Scholar API URL (e.g., `.../paper/arXiv:2304.07856`).
- It specifically requests the `references` field, along with sub-fields like `references.externalIds`, `references.title`, `references.authors`, and `references.year`.
- It includes robust error handling, especially for API rate limits (HTTP 429) and papers not found (HTTP 404).

2. **Filtering & Structuring (`convert_to_references_dict(...)`):** This is the core filtering logic.
   - It iterates through the full list of references returned by the API.
   - For each reference, it inspects the `externalIds` dictionary.
   - **If, and only if, a key named `'ArXiv'` exists** in `externalIds`, the reference is kept. All other references are discarded.
   - For the kept references, it extracts the title, author names, and publication date/year.
   - It formats this data into a new dictionary, where the **key is the cited paper's arXiv ID** (e.g., `2210.07675`).
3. **Saving (`save_references(...)`):** This function orchestrates the process. It calls the fetch and convert functions, then saves the final, structured dictionary to `references.json`. If no arXiv-to-arXiv citations are found, it correctly saves an empty dictionary (`{}`).

3.4.4 Final Output

The result of this step is a `references.json` file in the paper's main directory. This file contains a dictionary mapping the arXiv IDs of cited papers to their metadata, providing a clean list of the paper's arXiv-to-arXiv citations.

3.5 Step 5 - Missing Paper Recovery

This is a crucial data integrity step executed after the main multi-threaded download process is complete.

3.5.1 The Core Problem

Due to the concurrent nature of the pipeline and reliance on external APIs, transient errors such as network timeouts, API rate limits (HTTP 429), or other temporary disruptions can cause some papers to be skipped during the initial run. Without a verification step, the final dataset would be incomplete.

3.5.2 The Solution

The solution is a verification and recovery process. This process compares the initial list of *expected* paper IDs (from Step 1) against the list of paper IDs *actually* present in the data directory, then re-queues only the missing ones for processing.

3.5.3 Breakdown of the Process

1. **Collect Existing IDs (`collect_existing_ids`):** This function scans the base data directory (e.g., `../data/`). It reads the names of all subfolders (e.g., `2304-07856`, `2304-07857`) and uses a regular expression to parse them. It returns a dictionary mapping each month (e.g., `2304`) to a set of ID numbers (e.g., `{7856, 7857, ...}`) that have been successfully downloaded.

2. **Find Missing IDs (`find_missing_ids`):** This function takes a target month (e.g., `2304`), the set of existing IDs for that month, and the expected start and end ID numbers for that month. It compares the set of expected IDs against the set of existing IDs and returns a sorted list of the missing ID numbers.

3. **Recover Missing Papers (`recover_missing_papers`):** This is the main orchestration function for this step. It iterates through each month in the assignment's range. For each month, it calls `find_missing_ids` to get a list of what's missing. It then puts these missing paper IDs back into the download and reference queues to be processed by the worker threads, just like the initial batch.

3.5.4 Final Output

The output of this step is a complete dataset. It ensures that every paper within the target range has been successfully downloaded and processed, even if it failed on the first attempt.

3.6 Step 6 – Performance Monitoring (Benchmarking)

This final step is dedicated to monitoring and reporting the performance of the entire scraping pipeline, ensuring transparency, reproducibility, and providing actionable insights for future optimization.

3.6.1 The Core Problem

Large-scale data collection tasks can be resource-intensive and time-consuming. Without systematic monitoring, it is difficult to assess the efficiency of the pipeline, identify bottlenecks, or estimate the computational resources required for future runs. Accurate benchmarking is essential for both reproducibility and scalability.

3.6.2 The Solution

The solution is to integrate a comprehensive benchmarking system that tracks key performance metrics throughout the pipeline's execution. This system records timing, memory usage, and disk usage at multiple points, and summarizes the results at the end of the process, providing a clear overview of resource consumption and efficiency.

3.6.3 Breakdown of the Process

1. **Initialization:**
   At the very start of the main script, a `Benchmark` object is created. Its initialization method records the current time as the `start_time` and prepares containers for tracking RAM, disk usage, and per-step timings.

2. **RAM Monitoring:**
   The pipeline periodically samples the memory usage of the Python process using `psutil`. Each sample is appended to a list, allowing calculation of both the maximum and average RAM usage over the entire run. Additionally, a system-wide RAM usage function is available for more comprehensive diagnostics, using either `/proc/meminfo` (on Linux) or `psutil.virtual_memory()` as a fallback.

3. **Disk Usage Tracking:**

The pipeline tracks the peak disk usage by recursively summing the sizes of all files in the output directory at various points during execution. This allows the benchmark to report both the maximum disk usage observed during runtime and the final disk usage at completion.

4. **Step Timing:**
   The benchmark records the time spent in key phases: entry discovery (`id_fetch_time`), downloading papers (`download_times`), and extracting references (`reference_times`). For downloads and reference extraction, per-paper timings are stored, enabling calculation of average times per paper for each step.

5. **Reporting:**
   After all pipeline steps, including any recovery operations, are complete, the `report` method is called. This method:
   - Calculates the total elapsed time (`total_time`) since initialization.
   - Reports the time spent on entry discovery, average download time per paper, and average reference extraction time per paper.
   - Reports the maximum and average RAM usage of the Python process.
   - Reports the peak disk usage observed during runtime and the final disk usage at completion.
   - Reports the total number of papers processed.

3.6.4 Final Output

The output of this step is a detailed performance report, including total runtime, per-step timings, peak and average memory usage, peak and final disk usage, and the total number of papers processed. This information is invaluable for evaluating the efficiency of the pipeline, planning future data collection efforts, and ensuring the reproducibility and scalability of results.

4 Performance Report (10 Papers)

- **Test batch size:** 10 papers
  The pipeline was evaluated on a batch of 10 arXiv papers to benchmark performance and

resource usage.

- **Total runtime:** 42.45 seconds

  The complete scraping process, from entry discovery to final data output, was completed in just over 42 seconds.

- **Time for entry discovery:** 1.94 seconds

  The initial identification of valid arXiv IDs was performed rapidly, demonstrating the efficiency of the ID discovery logic.

- **Average download time per paper:** 6.83 seconds

  Each paper, including all available versions, was downloaded in under 7 seconds on average, reflecting effective parallelization and network utilization.

- **Average reference extraction time per paper:** 1.03 seconds

  Reference metadata extraction, including API calls and parsing, averaged just over 1 second per paper.

- **Python process max RAM usage:** 113.66 MB

  The peak memory usage of the Python process remained low, indicating efficient memory management throughout the run.

- **Python process average RAM usage:** 112.68 MB

  The average memory footprint was stable, with minimal fluctuation during processing.

- **Peak disk usage during runtime:** 24.60 MB

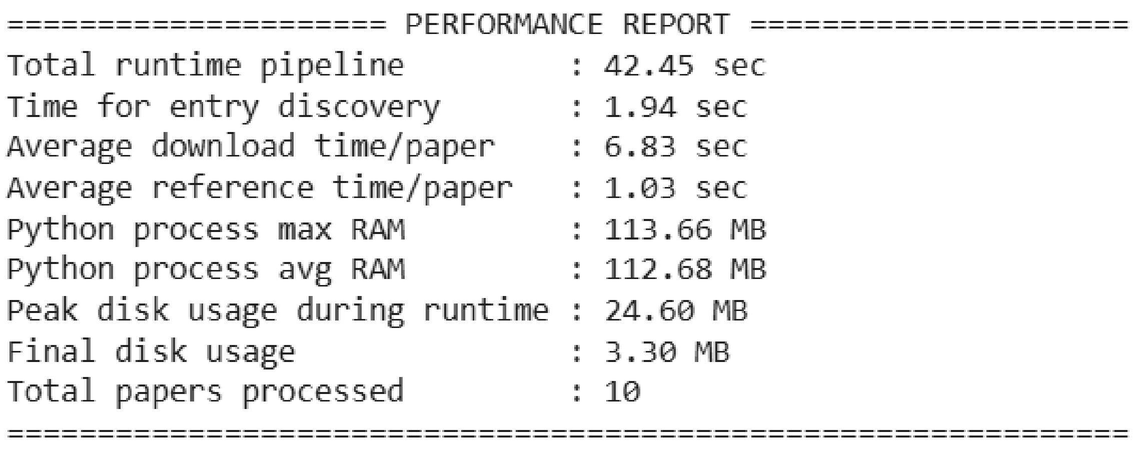
  The maximum disk space required during the scraping process, including temporary and intermediate files, was under 25 MB.

- **Final disk usage:** 3.30 MB

  After completion and cleanup, the total disk space occupied by the final dataset was only 3.3 MB, reflecting efficient storage and removal of temporary files.

- **Total papers processed:** 10

  All 10 papers in the test batch were successfully processed, with complete metadata and references extracted.


```
==================== PERFORMANCE REPORT ====================
Total runtime pipeline         : 42.45 sec
Time for entry discovery       : 1.94 sec
Average download time/paper    : 6.83 sec
Average reference time/paper   : 1.03 sec
Python process max RAM         : 113.66 MB
Python process avg RAM         : 112.68 MB
Peak disk usage during runtime : 24.60 MB
Final disk usage               : 3.30 MB
Total papers processed         : 10
============================================================
```

Hình 1: Performance with 10 papers

5 Task distribution

MSSV	Name	Tasks	Progress
23127130	Nguyễn Hữu Anh Trí	Handling the format and collecting ArXiv papers' IDs in given range	**100%**
		Downloading and extracting Source of the papers	**100%**
		Writing READ.ME for project overview and instructions on running the program	**100%**
23127051	Cao Tấn Hoàng Huy	Handing the metadata's structure and collecting metadata for papers	**100%**
		Building a function that recovers the missing papers after finishing process.	**100%**
		Writing Report for explaining pipeline and evaluating the result	**100%**
23127107	Nguyễn Huy Quân	Handling the Semantic Scholar requests to download references	**100%**
		Building a class for calculating the RAM and time of the process	**100%**
		Making a demo Video that explains the pipeline and tests the source code	**100%**

Bảng 1: Table of work distribution.

6 Pipeline Resources and Demonstration

6.1 Source Code Repository

The complete source code for the arXiv scraping pipeline, including all modules, utilities, and documentation, is publicly available on GitHub:

- **GitHub Repository:** https://github.com/AnhTtis/Data_Science_Project

The repository contains detailed instructions for installation, configuration, and usage, as well as example scripts and test cases.

6.2 Demonstration Video

A step-by-step demonstration of the pipeline in action, including setup, execution, and key features, is available on YouTube:

- **YouTube Video:** https://youtu.be/MWt4aQyfHkc

The video walkthrough highlights the main components of the system, showcases performance monitoring, and provides practical tips for running large-scale data collection tasks.

6.3 Summary

These resources provide both the technical foundation and a visual overview of the pipeline, enabling reproducibility and facilitating further development or adaptation by the research community.

7 Reference

[1] arXiv. arXiv API Basics. https://info.arxiv.org/help/api/basics.html, 2025. Accessed: 2025-11-17.

[2] arXiv. arXiv API User's Manual. https://info.arxiv.org/help/api/user-manual.html, 2025. Accessed: 2025-11-17.

[3] Semantic Scholar. Semantic scholar api — snippet text. https://api.semanticscholar.org/api-docs/graph#tag/Snippet-Text, 2025. Accessed: 2025-11-17.