Technical Documentation for Regulatory Gap Analysis Pipeline

Table of Contents

1. Introduction
2. File-by-File Documentation
   2.1) fetch_data.ipynb
   2.2) chunkdivide.ipynb
   2.3) fetch_citations.ipynb
   2.4) fetch_labels.ipynb
   2.5) opinion_analysis.ipynb
   2.6) thesis_main.ipynb
3. Usage and Workflow
4. Dependencies and Environment

1.Introduction

This document presents a comprehensive technical specification for an automated pipeline designed to identify, quantify, and visualize sector-specific regulatory gaps in U.S. federal court opinions concerning AI governance from 2010 through 2025. It is intended to guide developers and analysts through each stage of data acquisition, processing, analysis, and reporting, with sufficient depth to support replication, extension, or auditing of the methodology.

2. File-by-File Documentation

Each Jupyter notebook corresponds to a distinct stage of the pipeline. They must be executed in the prescribed order to ensure proper data dependencies and artifact generation.

2.1) fetch_data.ipynb

Description: This notebook orchestrates the retrieval of complete opinion metadata and full-text documents from the CourtListener API. It uses a predefined set of fifty AI-related keywords to filter opinions, applies robust retry and exponential backoff logic to handle transient API errors, and writes the cleaned results to a persistent file for later stages.

Implementation:

1. The notebook begins by applying nest_asyncio.apply() to allow nested asynchronous operations within the Jupyter environment.
2. It constructs a parameterized search URL combining the logical OR of AI keywords, date constraints (filed_after=2010-01-01, filed_before=2025-01-31), and a comprehensive list of federal courts.
3. The function fetch_page(session, url, headers, retries, backoff_factor) performs individual HTTP GET requests, catches exceptions, logs non-200 status codes, and implements an exponential backoff delay that grows by the specified factor with each retry.
4. The function fetch_all_pages_concurrently(api_url, headers, batch_size) leverages aiohttp.ClientSession to send concurrent batch requests, follows the API's pagination cursor via the next field, accumulates all JSON responses into a list, and estimates remaining time based on throughput. Any URLs that persistently fail after retries are captured in a failed_urls list for later manual or automated retry.
5. Once the fetch loop completes, retry_failed_urls(failed_urls, session, headers) attempts a single-pass re-fetch of any failed endpoints before finalizing the results.
6. The combined JSON results are loaded into a pandas.DataFrame. A nested field, recap_documents, is expanded into separate columns using json_normalize. A compiled regular expression removes illegal control characters from all string columns.
7. The cleaned DataFrame is saved first as an Excel workbook at the user-specified path. If the Excel engine is unavailable or an error occurs, the notebook gracefully falls back to writing a CSV file while notifying the user of the fallback.

Inputs and Outputs:

- Inputs include the full api_url, an authorization header dictionary, and optional parameters for retry count, backoff factor, concurrency batch size, and output file path.
- The primary output is a file named opinions_raw.xlsx or opinions_raw.csv, containing one row per court opinion with fields such as id, dateFiled, court, plain_text, and any expanded recap information.
- Secondary outputs include detailed console logs recording batch progress, elapsed time, estimated time remaining, and any errors or skipped URLs.

2.2) chunkdivide.ipynb

Description: This notebook addresses memory and performance constraints by partitioning a large JSON array of opinion records into smaller, uniformly sized JSON chunks. Each chunk can then be processed independently and in parallel by downstream notebooks.

Implementation:

1. The notebook verifies or creates the destination directory using os.makedirs(output_dir, exist_ok=True).
2. It loads the entire input JSON file into a Python list.
3. Using a for-loop over the range 0 to len(data) in steps of chunk_size, it slices the list into sublists of size up to chunk_size.
4. Each sublist is serialized with json.dump (using an indentation of 2 spaces) to a file named chunk_{index}.json in the output directory.
5. Immediately after writing, the notebook prints a log entry specifying the chunk index, the number of records written, and the index range covered.

Inputs and Outputs:

- Inputs: the path to the source JSON file, the desired output directory, and the integer chunk size.
- Outputs: one or more JSON files (e.g. chunk_0.json, chunk_1.json, …) and a console summary of the total chunks created.

2.3) fetch_citations.ipynb

Description: This notebook processes each JSON chunk of raw opinions to extract structured legal citations—statutes, case law references, and procedural rules—and compute essential text metrics (citation density, lexical complexity, hedging frequency, doctrinal counts) required for the Regulatory Gap Index.

Implementation:

1. It defines load_data(file_path), which attempts to open the file as JSON Lines and falls back to standard JSON if necessary, validating the presence of the plain_text column.
2. The function fast_clean_text(text) removes HTML markup and normalizes whitespace using BeautifulSoup with the lxml parser.
3. extract_citations(text) uses the Eyecite library to detect FullLawCitation (mapped to STATUTE) and FullCaseCitation (mapped to CASE_LAW), ignores IdCitation and SupraCitation, and applies custom regex rules to capture Federal Rules citations under the RULES group.
4. Normalization helpers—get_statute_codes, get_act_names_from_text, get_case_law_names_from_text, get_rule_citations—transform raw citation objects into human-readable strings.
5. Metric functions compute: citation density (total citations divided by sentence count), lexical complexity (ratio of unique tokens to total tokens), hedging frequency (modal verb occurrences ratio), and counts of doctrinal terms (“ultra vires” and “Chevron deference”).
6. The process_row function assembles all extracted and computed values into a dictionary.
7. process_all_rows(df) parallelizes process_row across CPU cores using multiprocessing.Pool and a tqdm progress bar to generate a completed DataFrame.
8. The final DataFrame is written to a CSV file named <chunk>_output.csv.

Inputs and Outputs:

- Input: the path to a chunk JSON file.
- Output: a CSV file with one row per opinion containing normalized citation fields and all computed text metrics.

2.4) fetch_labels.ipynb

Description: This notebook applies Google Gemini to classify each opinion according to its tertiary industry sector and judicial outcome, merges these labels with citation metrics and case metadata, and outputs a unified dataset ready for index computation and statistical analysis.

Implementation:

1. It defines Python Literal types SectorType (covering 18 industry categories) and JudgmentOutcomeType (Plaintiff, Defendant, Undecided) to enforce schema compliance.
2. A detailed system_prompt instructs the Gemini model to output JSON strictly matching the Pydantic model LegalAnalysisResult.
3. In extract_sector_and_outcome, the notebook calls the Gemini model at deterministic settings (temperature=0.0, top_p=0.0), parses the raw JSON response into the Pydantic schema, and handles RESOURCE_EXHAUSTED errors via a 30-second recursive sleep-and-retry.
4. The function write_result_to_excel initializes or appends [id, sector, judgment_outcome] rows into analysis(remaining).xlsx, saving after each write to preserve progress in case of interruptions.
5. For data merging, it reads caselaw_meta.xlsx (renaming opinion_id to id), citations_analysis_final.xlsx, and label_analysis_final.xlsx. It performs sequential outer joins on id, computes unique counts for STATUTE_codes, CASE_LAW_names, and RULES_codes using pandas.unique(), and finally merges in Act names from output1_acts_updated.xlsx before dropping redundant columns.
6. The consolidated DataFrame is saved as final_df.xlsx on the user's Drive or local data folder.

Inputs and Outputs:

- Inputs: intermediate CSV/Excel artifacts and the GOOGLE_API_KEY environment variable.
- Outputs: analysis(remaining).xlsx of raw labels and final_df.xlsx containing the full feature set for subsequent analysis.

2.5) opinion_analysis.ipynb

Description: This notebook serves as a catch-all to ensure that any chunks missed in the first citation-extraction pass are processed. It ultimately consolidates all per-chunk metrics into final combined CSV files.

Implementation:

1. The notebook reuses all loading, cleaning, extraction, and metric functions from fetch_citations.
2. It executes process_all_rows for each remaining chunk, writing per-chunk <chunk>_analysis.csv files.
3. It concatenates these into citation_analysis_final.csv and text_metrics_final.csv to guarantee completeness before RGI computation.

Inputs and Outputs:

- Input: any chunk files not previously processed.
- Outputs: final merged CSVs of citation and text metrics for use in thesis_main.

2.6) thesis_main.ipynb

Description: This notebook brings together all prior outputs to compute the Regulatory Gap Index (RGI), create descriptive and geospatial visualizations, and perform comprehensive statistical and causal inference analyses.

Implementation:

1. It loads final_df.xlsx, converts dateFiled to datetime objects, increments nonzero citation_diversity entries, and computes an opinion_count feature.
2. The function compute_rgi(df) implements a four-step scaling pipeline: winsorization at the 1st/99th percentiles; log-transform of citation diversity; robust scaling (median & IQR); min–max normalization; centering and directional inversion; and finally averages the transformed features into df['RGI'].
3. The notebook produces a suite of visual outputs: a histogram of RGI values; bar charts comparing mean RGI by sector and by court

(highlighting top and bottom performers); a year-by-year line plot of median RGI; and a sector–year heatmap pivot table.

4. For geospatial mapping, it reads the U.S. states shapefile with GeoPandas, merges state-level median RGI or RGP values on the STUSPS code, computes polygon centroids for labeling, and exports both interactive GeoJSON and static choropleth maps.
5. It executes statistical validation and causal models: PCA loadings and Cronbach's alpha via factor_analyzer and pingouin for RGI construct validation; Inverse-Probability Weighting (IPW) to estimate the causal effect of high RGI on case resolution; and mediation analysis (sum_citations → RGI → outcome) using statsmodels formulas.
6. All plots are saved as static images in outputs/figures, and GeoJSON/PNG map files are written to outputs/maps for easy inclusion in the thesis report.

Inputs and Outputs:

- Input: final_df.xlsx and the tl_2024_us_state.shp shapefile directory.
- Outputs: embedded notebook visuals plus exported PNG and GeoJSON assets.

3. Usage and Workflow

Clone the GitHub repository and checkout the main branch.

Install dependencies via pip install -r requirements.txt or conda env create -f environment.yml.

Export required environment variables: export COURTLISTENER_API_TOKEN="" export GOOGLE_API_KEY=""

Execute each notebook in sequence

jupyter nbconvert --execute 2.1_fetch_data.ipynb

jupyter nbconvert --execute 2.6_thesis_main.ipynb

Inspect intermediate files in data/ and final artifacts in outputs/.

Embed exported figures and maps into the Google Doc thesis.

4. Dependencies and Environment

Python 3.8 or higher.

Key packages:

aiohttp, nest_asyncio, tqdm, pandas, numpy, jsonlines, openpyxl, nltk, bs4, eyecite, scikit-learn, statsmodels, matplotlib, seaborn, plotly, geopandas, contextily, shapely, factor_analyzer, pingouin.

External services: CourtListener API token and Google Gemini API key; optional Google Drive mount for persistent storage.

Data files: U.S. states shapefile (tl_2024_us_state) and auxiliary Excel metadata spreadsheets.