Google Issue Tracker Scraper

Codebase Explanation & Technical Documentation

Date: 2026-01-21

This document provides a detailed technical explanation of the Python script designed to scrape, analyze, and visualize data from the Google Issue Tracker.

Table of Contents

1. Project Overview

The Google Issue Tracker Scraper is a robust Python application that automates the extraction of issue details from Google's public issue tracker. It takes a CSV file containing issue IDs or URLs as input, scrapes detailed descriptions and labels for each issue, categorizes them based on keywords (e.g., Pixel models, bug types), and generates actionable insights through charts and summary reports.

2. Key Libraries & Dependencies

Library	Purpose
pandas	Data manipulation and analysis (DataFrames).
requests	Making HTTP requests to fetch web pages.
BeautifulSoup	Parsing HTML content to extract text.
concurrent.futures	Handling multithreading for fast parallel scraping.
matplotlib / seaborn	Generating bar charts and pie charts.
tqdm	Displaying progress bars during long operations.

3. Core Architecture

The application follows a linear pipeline architecture with distinct stages: Input -> Cleaning -> Scraping (Parallel) -> Analysis -> Visualization -> Output.

4. Detailed Code Breakdown

A. Dependency Management

The script includes a self-check mechanism (`check_and_install_dependencies`) that runs at startup. It verifies if all required packages are installed. If any are missing, it attempts to install them automatically using `pip` via `subprocess`.

B. Setup & Logging

To ensure robustness and traceability:
- **Directories:** Uses `pathlib` to create input/output folders automatically.
- **Logging:** Configures a dual-output logger (Console + File) that handles UTF-8 encoding, which is critical for Windows systems to avoid emoji/character errors.

C. Data Loading & Cleaning

The `load_csv` function handles file I/O. It attempts multiple file encodings (utf-8, latin-1, cp1252) to prevent crashes when reading user-provided CSVs.

The `remove_duplicates` function intelligently identifies unique issues based on 'ID' or 'Title' columns to ensure the analysis is accurate and efficient.

D. Multithreaded Web Scraping (The Core)

This is the most critical part of the application for performance.
- **Problem:** Scraping 1000 pages sequentially is too slow.
- **Solution:** The `scrape_all_issues` function uses `concurrent.futures.ThreadPoolExecutor`.
- **Implementation:** It spins up 20 worker threads that fetch pages in parallel. It uses `urllib3` to suppress SSL warnings (needed for some corporate/proxy environments) and `requests` to fetch the HTML. `BeautifulSoup` extracts the 'description' and 'labels' from the raw HTML.

```
with concurrent.futures.ThreadPoolExecutor(max_workers=20) as executor:
# Submit tasks...
# Process results as they complete...
```

E. Categorization Logic

Once text is scraped, the `apply_categorization` function runs.
- **Pixel Model Detection:** Regex patterns (e.g., `pixel\s*9`) find specific device models.
- **Severity Analysis:** Keywords like 'crash', 'dead', 'freeze' trigger 'High' severity.
- **Topic Categorization:** A keyword dictionary maps terms to categories like 'UI/Graphics Issues', 'Network Issues', etc.

F. Visualization & Reporting

Finally, `matplotlib` and `seaborn` generate charts:
- **Bar Chart:** Visualizes the count of issues per category.
- **Pie Chart:** Shows the percentage distribution.

The `export_results` function saves the cleaned dataset (`cleaned_data.csv`) and a summary table (`summary.csv`).

Conclusion

This project demonstrates a complete data engineering pipeline: ingesting raw data, enriching it via external sources (web scraping), transforming it (cleaning/categorizing), and loading it into consumable formats (CSV reports and visualizations).