Language Detector Web Application

Project Documentation

1. Overview

This project is a character-based language detection system exposed as a Flask web application. It downloads books in six languages from Project Gutenberg (French, German, English, Spanish, Chinese, and Japanese), cleans and tokenizes the text, builds character n-gram statistics for each language, and then uses these statistics to guess the language of an input paragraph.

The architecture has two main parts:

- An offline training pipeline that downloads data, preprocesses it, and builds language models stored as n-gram ranking files.
- An online Flask application that accepts user text, applies the language-detection logic, and displays the detected language in a web page.

2. Project Structure

On your local machine, you can organize the project with the following directory layout:

```
language_detector/
  app.py                  # Flask web server
  lang_detection.py       # Core language detection logic (training + testing)
  prepare_models.py       # Script to run the training pipeline once
  test.txt                # Temporary file used when testing user input
  templates/
    index.html            # Main page with form
    result.html           # Page that shows “Detected Language: ...”
  static/
    style.css             # CSS styling
  dataset/                # Raw or representative text per language
  tokenized/              # Tokenized and cleaned text (one token per line)
  nGrams/                 # Ranked n-gram lists per language
  processed/              # n-gram frequency files per language
```

The templates folder holds HTML templates rendered by Flask. The static folder holds CSS files. The dataset, tokenized, nGrams, and processed folders are created and filled by the training pipeline.

3. Installation and Setup

• Create the project folder and enter it:

```
mkdir language_detector
```

```
cd language_detector
```

• **Create a virtual environment (recommended):**

```
python3 -m venv venv
source venv/bin/activate
```

• **Install the required Python packages:**

```
pip install flask requests pycountry regex
```

• **Create the subfolders:** dataset, tokenized, nGrams, processed, templates, static.

• **Add the Python and HTML files** as described in the next sections.

4. Core Language-Detection Logic (lang_detection.py)

The lang_detection.py module contains the logic for preprocessing text, generating character n-grams, building language-specific models, and testing an input paragraph against those models.

4.1 Imports and Base Directory

The module imports os, requests, pycountry, regex, and collections.Counter. It also defines a BASE_DIR constant pointing to the directory where lang_detection.py is located. Using BASE_DIR instead of hard-coded paths makes your project portable.

4.2 Text Cleaning Helpers

Three helper functions standardize preprocessing for both training and testing:

- **split_and_pad(text)**: extracts word-like tokens using a Unicode-aware regular expression, lowercases them, and adds a newline to each token. The result is a list of tokens, one per line.
- **data_cleaning(raw_text)**: removes punctuation and digits using a regular expression, focusing the model on pure letter patterns.
- **skip_template_text(text)**: strips the Project Gutenberg header and footer using marker phrases like '*** START OF THE PROJECT GUTENBERG EBOOK' and '*** END OF THE PROJECT GUTENBERG EBOOK'. It then trims extra boilerplate and calls data_cleaning.

4.3 Downloading Books and Creating Token Files

The get_books_text function drives the training data download and preprocessing.

High-level steps:

- Define the list of language codes: ['fra', 'deu', 'eng', 'spa', 'zho', 'jpn'].
- For each language, define a list of Project Gutenberg URLs pointing to books in that language.
- For each (language, URL) pair, download the book using requests.get and decode it as UTF-8.
- Call skip_template_text to remove headers, footers, and boilerplate, then accumulate cleaned text for that language.
- After all URLs for a language are processed, save raw representative text to dataset/Language.txt and the tokens (from split_and_pad) to tokenized/Language.int1.txt.

5. N-Gram Generation and Counting

Once tokenized text exists for each language, the project builds models of character n-gram frequencies. These n-grams capture short sequences of characters that are typical in a language.

5.1 Character N-Gram Generation

The function generate_ngrams(line) iterates over a string and collects all substrings of length 1 to 5. For example, the word 'hello' produces n-grams such as 'h', 'he', 'hel', 'hell', 'hello', 'e', 'el', 'ell', and so on.

5.2 Counting and Sorting

count_ngram_frequency(ngrams) uses collections.Counter to count occurrences of each n-gram. sort_ngrams_by_frequency(counter) sorts the n-grams primarily by decreasing count and secondarily alphabetically, yielding a ranked list such as [('a', 1000), ('e', 950), ...].

5.3 Building Ranked N-Gram Files

The generate_and_count_ngrams(input_file_path, output_file_path, frequency_file_path) function:

- Reads each line from the tokenized language file.
- Generates character n-grams for each line and updates a Counter with their counts.
- Sorts the n-grams by frequency using sort_ngrams_by_frequency.
- Writes only the n-gram strings into output_file_path (one per line). This file defines the ranking of n-grams for that language.
- Writes 'ngram: count' lines into frequency_file_path, which is useful for inspection and debugging.

6. Training Pipeline Script (prepare_models.py)

The prepare_models.py script wraps the training steps so you can run them once from the command line instead of during every Flask request.

The script typically performs the following steps:

1. Call get_books_text() to download books, clean them, and generate token files for each language.
2. For each language code (fra, deu, eng, spa, zho, jpn):
- Build file paths to tokenized/Language.int1.txt, nGrams/Language.nGrams.txt, and processed/Language.nGramsFrequency.txt.
- Call generate_and_count_ngrams to create the ranked n-gram file and the corresponding frequency file.

After prepare_models.py completes, the nGrams folder contains a .nGrams.txt file for each language. These files are the language models used during detection.

7. Testing an Input Paragraph (test_language)

The test_language(file_name) function applies the trained models to a new paragraph of text. The file_name parameter typically points to test.txt, which holds the user's input paragraph.

- Collect all *.nGrams.txt language model files from the nGrams folder.
- Read the entire contents of file_name, then apply the same preprocessing as during training: data_cleaning followed by split_and_pad.
- Generate n-grams from the resulting tokens and build a frequency Counter.
- Sort the test n-grams by frequency to obtain a ranked list.
- For each language model file, build a rank table (ngram $\rightarrow$ rank) from its stored n-grams.
- Compute a distance score by summing the absolute differences between the test rank and the language-model rank for each n-gram, using a large penalty if an n-gram from the test is missing in that language model.
- Select the language model with the smallest distance score and return its language name.

A crucial detail is that preprocessing for test input must match preprocessing during training. If the test pipeline skipped data_cleaning or split_and_pad, n-grams would be inconsistent and the distance metric would become unreliable, often causing the same language to be chosen every time.

8. Flask Web Application (app.py)

The Flask web application exposes the language detector through a simple HTML form. It has two main routes:

- **GET /**: renders templates/index.html, which displays the title, instructions, text area, and 'Detect Language' button.
- **POST /detect_language**: reads the submitted paragraph from the form, writes it to test.txt, calls test_language(test.txt), and renders templates/result.html with the detected language injected as a template variable.

8.1 Templates and Static Files

index.html defines the overall layout and the form that posts to /detect_language. It also loads static/style.css using Flask's url_for function. result.html extends index.html and overrides the result block to show 'Detected Language: {{ language }}'.

style.css defines the visual appearance: a gradient background, centered container, styled textarea and buttons, and readable fonts and spacing.

9. Running the Application

- Ensure you have trained models by running prepare_models.py once.
- Start the Flask development server by running python app.py.

- Open a browser and navigate to http://127.0.0.1:5000.
- Enter a paragraph of text in one of the supported languages and click 'Detect Language'.
- Observe the detected language on the result page.

If the application always predicts the same language, verify that each language has a non-empty .nGrams.txt file and that the test preprocessing code matches the training preprocessing steps.

10. Possible Extensions

- Support additional languages by adding new Project Gutenberg URLs and updating the language code list.
- Experiment with different n-gram lengths or weighting schemes for the distance metric.
- Provide top-k language guesses with scores instead of a single best guess.
- Implement an API endpoint that returns JSON, allowing other applications to use the language detector programmatically.
- Replace the rank-based n-gram distance model with a more advanced machine learning model, such as a character-level neural network or a modern library like fastText.

This report summarizes the entire pipeline from data collection and preprocessing, through n-gram model training, to serving predictions via Flask. It is intended to help you understand the design so you can confidently modify, extend, or rebuild the project from scratch.