CULTUREMAP-IR: Cultural Unification and Linguistic Textual Utilization for Regional Extraction and Mapping of All Iranian Provinces, IRan

Asal Meskin* (401106511), Alireza Mirrokni* (401106617)

Computer Engineering Department, Sharif University of Technology

*These authors contributed equally to this work.

https://github.com/CULTUREMAP-IR

Contents

1 Introduction

In this project, we present two complementary automated methods for extracting and structuring cultural, geographic, and environmental data for six distinct Iranian provinces: **Isfahan**, **Fars**, **Bushehr**, **Chaharmahal and Bakhtiari**, **Hormozgan**, and **Kohgiluyeh and Boyer-Ahmad**. These regions span central desert basins and mountain plateaus, fertile coastal plains on the Persian Gulf, and densely forested highlands, each embodying unique traditions, dialects, and ecological challenges.

- **Method 1** leverages a locally hosted large language model (Gemma-3) to parse PDF documents—ranging from provincial geography texts to scientific reports—directly into a strict JSON schema.
- **Method 2** builds on the same extraction utilities but instructs GPT-4 (via the OpenAI API) not only to map text to JSON, but also to autonomously query authoritative online sources (e.g. Wikipedia and official tourism sites) when necessary to fill gaps or update figures. Although early experiments were performed by manually pasting prompts into the OpenAI playground due to quota limits, the provided code will execute end-to-end once a valid `OPENAI_API_KEY` is configured.

2 Challenges and Motivations

Building a high-quality, culturally informed dataset for six diverse Iranian provinces entails a range of technical and conceptual challenges. At the PDF ingestion stage, we must contend with inconsistent formatting, embedded images, and non-standard fonts in provincial geography texts. Extracted text often contains noise—header/footer artifacts, page numbers, and stray line breaks—that must be normalized without losing semantic content. Splitting long documents into overlapping chunks requires careful tuning: too small, and we lose context; too large, and we exceed the model's input limits.

A core difficulty is the relative scarcity of high-quality Farsi training and reference data. Many regional sources are unpublished or available only in print, and online resources in Farsi often lack consistent metadata or structured markup. This necessitates web crawling, OCR, and manual validation steps to bootstrap a reliable corpus. Furthermore, dialectal variations across provinces introduce inconsistencies in spelling and terminology, complicating automatic extraction and normalization.

On the LLM side, guiding Gemma-3 or GPT-4 to produce strictly valid JSON demands a meticulously crafted prompt template and robust post-processing to strip out markdown fences, stray commas, and incidental commentary. Deduplication is non-trivial: cultural and geographical entities may appear across multiple chunks with minor naming variations, so we maintain per-section "seen" sets and apply normalization heuristics to avoid redundant entries.

Quota and latency constraints also shaped our design. While Method 2 leverages GPT-4's superior reasoning and live web search capabilities (e.g. Wikipedia lookups), limited API credits required manual prompt testing before automating. To hide network latency, we dispatch chunks in parallel and provide real-time progress feedback, ensuring the pipeline remains responsive even under heavy load.

Despite these difficulties, the project is motivated by a compelling vision: to enrich NLP models with deep, region-specific knowledge that reflects Iran's rich cultural tapestry. By converting scattered text sources—PDFs, web pages, and archival images—into a unified JSON schema, we lay the groundwork for language models that can generate, translate, and analyze content with genuine awareness of local landmarks, dialects, and environmental factors. This cultural grounding is essential for applications ranging from intelligent tutoring systems to heritage preservation and region-tailored information retrieval.

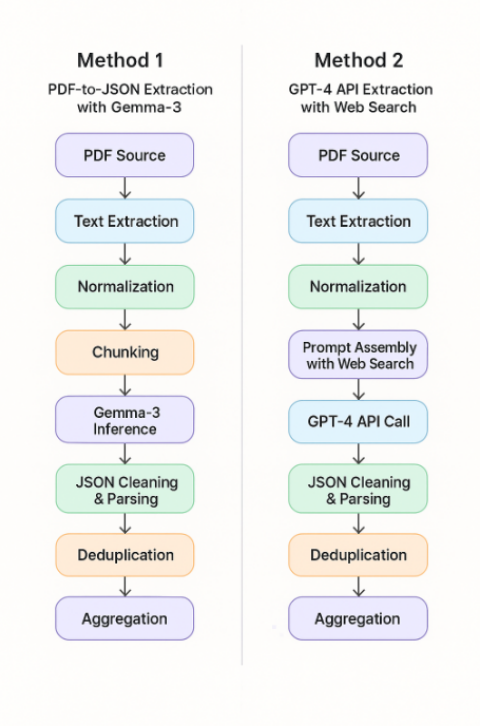


Figure 1: Side-by-side overview of Method 1 (PDF-to-JSON with Gemma-3) and Method 2 (GPT-4 API with web search) pipelines.

3 Method 1: PDF-based JSON Extraction with Gemma-3

3.1 Overview

The first approach automates the conversion of PDF text into a strict JSON format by orchestrating text extraction, cleaning, chunking, prompt assembly, model inference, and result aggregation. This method is fully configurable through a single dictionary and is designed to handle long documents while maintaining context.

3.2 Code and Explanations

3.2.1 Dependency Imports

In this segment, we import only the essential modules required for the pipeline. Standard libraries include `os`, `re`, `json`, and `logging` for filesystem interaction, pattern matching, JSON handling, and logging, respectively. The `time` module records execution times. We use `Path` from `pathlib` for path manipulations. For model operations, `torch` is imported, while `FastModel` and `get_chat_template` from `unsloth` provide an interface to the Gemma-3 model. `PdfReader` from `PyPDF2` enables PDF text extraction. Progress feedback is achieved through `tqdm` and `rich`, which also handle console messages. Finally, `gc` is imported to allow explicit memory cleanup between inference calls.

```
import os, re, json, logging, time, gc
from pathlib import Path
from typing import Optional, List, Dict, Any, Set

import torch
from PyPDF2 import PdfReader
from unsloth import FastModel
from unsloth.chat_templates import get_chat_template
```

```
from rich.console import Console
from rich.progress import (
    Progress, SpinnerColumn, BarColumn,
    TextColumn, TimeElapsedColumn, TimeRemainingColumn
)
from rich.table import Table
```

Listing 1: Essential imports

3.2.2 Configuration and Logging

All configurable parameters have been centralized in a single `CONFIG` dictionary. This includes PDF path, province name, page range, chunk sizes, model identifier, and device selection. We set up a standard Python logger to emit INFO-level messages to both a file and the console, ensuring reproducibility and traceability.

```
CONFIG = {
    "pdf_path": Path(os.getenv("PDF_PATH", "./books/kohgiloye.pdf")),
    "province": os.getenv("PROVINCE", "Kohgiluyeh and Boyer-Ahmad"),
    "start_page": 10,
    "end_page": 103,
    "chunk_size": 2000,
    "overlap_size": 50,
    "model_name": "unsloth/gemma-3-4b-it-unsloth-bnb-4bit",
    "max_seq_length": 2048,
    "device": "cuda" if torch.cuda.is_available() else "cpu",
    "log_file": "extraction.log"
}
logging.basicConfig(
    level=logging.INFO,
    format="%(asctime)s [%(levelname)s] %(message)s",
    handlers=[
        logging.FileHandler(CONFIG["log_file"]),
        logging.StreamHandler()
    ]
)
console = Console()
```

Listing 2: Centralized configuration and logging

3.2.3 JSON Cleaning Utilities

To robustly extract valid JSON from the model's free-form output, we define three helper functions. `clean_json_string` strips JavaScript-style comments and trailing commas. `extract_json_block` scans for the longest balanced brace-matched substring. `clean_json_block` removes any Markdown code fences.

```
def clean_json_string(s: str) -> str:
    s = re.sub(r"//.*", "", s)
    s = re.sub(r"/\*.*?\*/", "", s, flags=re.DOTALL)
    return re.sub(r",(\s*[}\]])", r"\1", s)

def extract_json_block(text: str) -> Optional[str]:
    blocks, start, depth = [], None, 0
    for i, ch in enumerate(text):
        if ch == "{":
            if depth == 0: start = i
```

```
            depth += 1
        elif ch == "}" and depth > 0:
            depth -= 1
            if depth == 0 and start is not None:
                blocks.append(text[start : i+1])
                start = None
    return max(blocks, key=len) if blocks else None

def clean_json_block(raw: str) -> str:
    txt = raw.strip()
    if txt.startswith("```"):
        lines = txt.splitlines()[1:-1]
        txt = "\n".join(lines)
    return txt
```

Listing 3: JSON extraction and cleanup helpers

3.2.4 Text Extraction and Normalization

We read the specified page range from the PDF using `PdfReader`, concatenating each page's text. Any extraction errors are logged but do not halt execution. The normalized text collapses all whitespace to single spaces and strips out digits to remove page numbers or footnotes.

```
def extract_text_from_pdf(path: Path, start: int, end: int) -> str:
    reader = PdfReader(str(path))
    pages = reader.pages[start-1:end]
    texts = []
    for pg in pages:
        try:
            texts.append(pg.extract_text() or "")
        except Exception as e:
            logging.warning(f"Failed to extract page: {e}")
    return "\n".join(texts)

def normalize_text(text: str) -> str:
    text = re.sub(r"\s+", " ", text)
    return re.sub(r"\d+", "", text).strip()
```

Listing 4: PDF text extraction and normalization

3.2.5 Chunking Strategy

Because Gemma-3 has a maximum sequence length, we split the normalized text into overlapping chunks of up to 2 000 characters, preserving 50 characters of overlap at each boundary to maintain context.

```
def chunk_text(text: str, max_chars: int, overlap: int) -> List[str]:
    words = text.split()
    chunks, current, length = [], [], 0
    for w in words:
        if length + len(w) + 1 > max_chars:
            chunks.append(" ".join(current))
            carry = " ".join(current)[-overlap:].split() if overlap
                else []
            current, length = carry, sum(len(x)+1 for x in carry)
        current.append(w)
        length += len(w) + 1
```

```
    if current:
        chunks.append(" ".join(current))
    return chunks
```

Listing 5: Split text into overlapping chunks

3.2.6 Prompt Template

We provide a strict English-language JSON schema and a filled example, followed by the header `"-- Source Text --"`. This guides the model to output a single, valid JSON object per chunk without any fences or commentary.

```
JSON_SCHEMA = """{ ... }"""      # full schema in English
EXAMPLE_JSON = """{ ... }"""     # single example

PROMPT_TEMPLATE = (
    "You are Gemma-3. Produce one valid JSON object matching the schema below.\n"
    "Use double quotes for keys/strings. If uncertain, use \"\" or [].\n"
    "Do not wrap output in fences or add comments.\n\n"
    + JSON_SCHEMA + "\nExample:\n" + EXAMPLE_JSON
    + "\n--- Source Text ---\n"
)
```

Listing 6: Prompt template with schema and example

3.2.7 Model Initialization

We load Gemma-3 in 4-bit quantized mode to reduce GPU memory usage, automatically mapping layers to available devices. The tokenizer is wrapped in a chat template so that our prompt is treated as a user message.

```
console.log(f"Loading {CONFIG['model_name']} on {CONFIG['device']}")
model, tokenizer = FastModel.from_pretrained(
    CONFIG['model_name'],
    max_seq_length=CONFIG['max_seq_length'],
    load_in_4bit=True,
    device_map="auto"
)
tokenizer = get_chat_template(tokenizer, chat_template="gemma-3")
console.log("Model loaded successfully!")
```

Listing 7: Load and prepare Gemma-3

3.2.8 Deduplication and GPU Cleanup

To avoid duplicate entries across chunks, we maintain sets of seen section names and sub-items. After each inference we immediately free GPU memory by deleting tensors and calling both Python and CUDA garbage collectors.

```
def cleanup_gpu(tensors, output):
    del tensors, output
    gc.collect()
    if CONFIG['device'].startswith("cuda"):
        torch.cuda.empty_cache()

```

```
seen_sections = {sec: set() for sec in [
    "geographical_features","natural_resources",
    "vegetation","topography","tourist_attractions","climate_impacts"
]}
seen_items = set()
```

Listing 8: Cleanup and deduplication helpers

3.2.9 Chunk Processing

Each chunk is wrapped in the prompt, tokenized, and sent through the model with controlled decoding parameters. We extract the JSON block, clean it, parse into a dictionary, then apply our deduplication logic before returning the result.

```
def process_chunk(chunk: str, idx: int) -> Optional[Dict[str, Any]]:
    prompt = PROMPT_TEMPLATE + chunk + "\n"
    inputs = tokenizer.apply_chat_template(
        [{"role":"user","content":[{"type":"text","text":prompt}]}],
        add_generation_prompt=True
    )
    inputs = tokenizer([inputs], return_tensors="pt").to(CONFIG['device'])
    with torch.no_grad():
        out = model.generate(**inputs, max_new_tokens=512, temperature=1.0, top_p=0.9)
    decoded = tokenizer.batch_decode(out)[0]

    raw = extract_json_block(decoded)
    if not raw:
        cleanup_gpu(inputs, out)
        return None
    raw = clean_json_block(raw)
    raw = clean_json_string(raw)

    try:
        parsed = json.loads(raw)
    except json.JSONDecodeError:
        cleanup_gpu(inputs, out)
        return None

    # Deduplicate entries
    for sec in seen_sections:
        unique = []
        for item in parsed.get(sec, []):
            key = item.get("name") or json.dumps(item, sort_keys=True)
            if key not in seen_sections[sec]:
                seen_sections[sec].add(key)
                unique.append(item)
        parsed[sec] = unique

    cleanup_gpu(inputs, out)
    return parsed
```

Listing 9: Process a single chunk end-to-end

3.2.10 Orchestration and Output

Finally, the `main()` function orchestrates text extraction, normalization, chunking, parallel processing, and aggregation. Results are merged and written to `{}'_dataset.json`, and a summary table reports total items, chunks processed, and elapsed time.

```
def main():
    start = time.time()
    console.rule(f"Starting extraction for {CONFIG['province']}")
    text = normalize_text(extract_text_from_pdf(
        CONFIG['pdf_path'], CONFIG['start_page'], CONFIG['end_page']
    ))
    chunks = chunk_text(text, CONFIG['chunk_size'], CONFIG['
        overlap_size'])

    combined = {"province": CONFIG['province']}
    results = []

    with Progress(SpinnerColumn(), TextColumn("{task.description}"),
                  BarColumn(), TextColumn("{task.completed}/{task.total
                      }"),
                  TimeElapsedColumn(), TimeRemainingColumn(),
                  console=console) as progress:
        task = progress.add_task("Extracting chunks", total=len(chunks)
            )
        for idx, chunk in enumerate(chunks, 1):
            res = process_chunk(chunk, idx)
            if res:
                results.append(res)
            progress.advance(task)

    # Merge results
    for part in results:
        for k, v in part.items():
            if k != "province":
                combined.setdefault(k, []).extend(v if isinstance(v,
                    list) else [v])

    out_path = Path(f"./{CONFIG['province']}_dataset.json")
    with open(out_path, "w", encoding="utf-8") as f:
        json.dump(combined, f, ensure_ascii=False, indent=2)

    elapsed = time.time() - start
    total = sum(len(v) for k,v in combined.items() if k!="province")
    console.rule("Extraction complete")
    console.print(f"Saved to {out_path}     {total} items in {elapsed
        :.2f}s")
```

Listing 10: Main orchestration function

4 Method 2: GPT-4 with Autonomous Web Search

4.1 Overview

This method reuses the extraction, normalization, chunking, and JSON-cleaning utilities from Method 1, but replaces the local LLM inference with OpenAI's GPT-4 ChatCompletion API. Although our initial experiments were performed by manually pasting prompts into the OpenAI

playground (due to limited quota), the code below will run end-to-end automatically once a valid `OPENAI_API_KEY` is set.

4.2 Code and Explanations

4.2.1 Configuration and API Key Verification

We extend the `CONFIG` dictionary to include GPT-4 parameters and enforce that an `OPENAI_API_KEY` is present. In our experiments, limited quota compelled manual prompt submission at `https://chatgpt.com`, but the code below will perform calls programmatically once a key is provided.

```
CONFIG.update({
    "openai_model":  os.getenv("GPT_MODEL", "gpt-4"),
    "max_tokens":    int(os.getenv("MAX_TOKENS", 1024)),
    "temperature":   float(os.getenv("TEMPERATURE", 1.0)),
    "workers":       int(os.getenv("WORKERS", 1)),
})
OPENAI_API_KEY = os.getenv("OPENAI_API_KEY")
if not OPENAI_API_KEY:
    raise RuntimeError("OPENAI_API_KEY is not set.")
openai.api_key = OPENAI_API_KEY
```

Listing 11: Extend CONFIG and verify API key

4.2.2 Prompt Template with Search Instruction

The prompt template has been enhanced to instruct GPT-4 to consult reliable online sources when the extracted PDF text lacks sufficient detail.

```
PROMPT_TEMPLATE = f"""
You are GPT-4. Produce exactly one valid JSON object matching the
   schema below.
1. Use double quotes (") for all keys and string values.
2. If data is missing or uncertain (<90% confidence), use "" or []
     never   {}.
3. Always include at least one real example per list.
4. Do not wrap JSON in code fences or markdown, and do not add comments
5. If the source text does not provide enough information, autonomously
    search Wikipedia
   or other authoritative web resources to supplement and update the
      data.

{JSON_SCHEMA_EXACT}

Example:
{EXAMPLE_JSON}

--- Source Text ---
"""
```

Listing 12: Revised `PROMPT_TEMPLATE`

4.2.3 Chunk Processing via ChatCompletion

The `process_chunk` function sends the enhanced prompt to GPT-4, then extracts and cleans the JSON block using the same helpers as in Method 1.

```python
def process_chunk(chunk: str, idx: int) -> Optional[Dict[str, Any]]:
    prompt = PROMPT_TEMPLATE + chunk + "\n"
    try:
        resp = openai.ChatCompletion.create(
            model=CONFIG['openai_model'],
            messages=[{'role':'user','content':prompt}],
            max_tokens=CONFIG['max_tokens'],
            temperature=CONFIG['temperature'],
            n=1
        )
        content = resp.choices[0].message.content.strip()
    except Exception:
        return None

    raw = extract_json_block(content)
    if not raw:
        return None
    raw = clean_json_block(raw)
    raw = clean_json_string(raw)
    try:
        return json.loads(raw)
    except json.JSONDecodeError:
        return None
```

Listing 13: API-based `process_chunk`

4.2.4 Concurrent Execution

We dispatch chunk processing across a `ThreadPoolExecutor` to hide network latency. Results are merged as in Method 1 and saved to `{province}_dataset.json`, with progress bars and a final summary table reporting total items, chunks, and elapsed time.

```python
with ThreadPoolExecutor(max_workers=CONFIG['workers']) as executor:
    futures = {
        executor.submit(process_chunk, chunk, i+1): i+1
        for i, chunk in enumerate(chunks)
    }
    for fut in as_completed(futures):
        res = fut.result()
        if res:
            partials.append(res)
        progress.advance(task)
```

Listing 14: Parallel processing in `main()`

4.2.5 Aggregation and Output

After all chunks complete, we merge the per-chunk dictionaries exactly as in Method 1 and write the combined JSON to disk. Progress bars and a final summary table (via `rich.table`) report the total items extracted, number of chunks processed, and elapsed time.

5 Labeling Policy and Dataset Refinement

To ensure high quality and consistency in our extracted datasets, we employed LabelStudio as our annotation platform. All JSON records for each province were imported into LabelStudio and assigned one of three categorical labels:

- **Acceptable**: the record is complete, accurate, and ready for inclusion without further edits.
- **Needs Revision**: the record contains minor errors or omissions (e.g. typos, missing fields, slight formatting issues) that we can correct with minimal effort.
- **Unacceptable**: the record is fundamentally flawed (e.g. malformed JSON, grossly incorrect content, or irrelevant data) and must be discarded.

Because our team consists of two annotators, we adopted a collaborative adjudication procedure. For each record, both annotators independently inspected the JSON and assigned their label. If both annotators agreed—whether "Acceptable," "Needs Revision," or "Unacceptable"—that label was accepted as final. In the rare cases of disagreement, we automatically generated a concise summary of the record's content and fed it to a language model with the question:

> "Based on the schema and this JSON content, should this record be labeled Acceptable, Needs Revision, or Unacceptable?"

The model's response served as the tie-breaker, ensuring an objective third opinion without requiring a human arbitrator.

Once all records had a final label, we applied the following policy to craft our final datasets. Every record labeled "Unacceptable" was removed entirely from the collection. Records marked "Needs Revision" were exported back to a staging area for manual correction. After revision, those entries were re-validated through the same two-person + LLM adjudication loop. All records labeled "Acceptable" (either initially or after revision) were then merged into the finalized province-level JSON files.

This multi-stage labeling and refinement process combines human domain expertise, peer consensus, and automated model assistance to guarantee that our datasets are both accurate and consistent. By removing fundamentally flawed entries and improving borderline cases, we ensure that downstream tasks are built on a foundation of trustworthy, high-quality data.

6 Results and Analysis

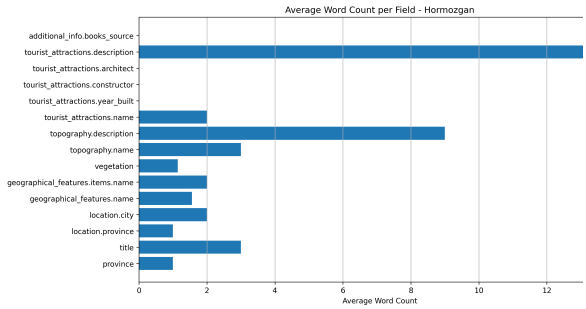
Method 2 (GPT-4 with autonomous web search) consistently enriches and deepens the extracted JSON compared to Method 1 (Gemma-3 on raw PDF text). Although Method 1 processes more raw chunks ("records"), Method 2 produces substantially more descriptive content per field by supplementing PDF data with up-to-date information from online sources.

6.1 Quantitative Comparison

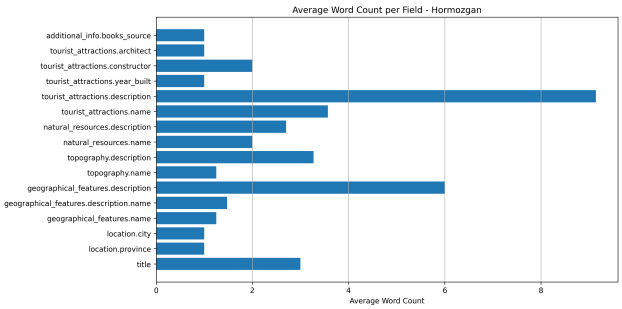
Province	Records M1	Records M2	Words M1	Words M2	% Increase
Hormozgan	76	37	218	236	+8.2%
Bushehr	57	31	170	299	+76.0%
Kohgiluyeh & Boyer-Ahmad	62	45	184	288	+56.5%
Fars	36	23	119	325	+173.9%
Chaharmahal & Bakhtiari	31	35	83	338	+307.2%
Isfahan	51	26	144	294	+104.2%

Table 1: Comparison of record counts and total word counts per province for Method 1 vs. Method 2.

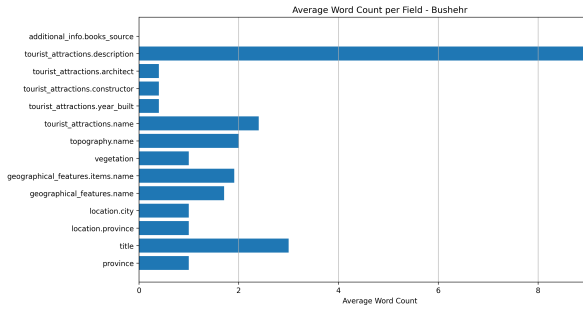
6.2 Field-Word Count Distributions



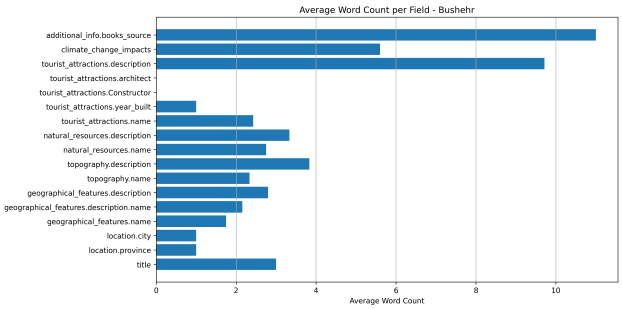
(a) Hormozgan – Method 1



(b) Hormozgan – Method 2



(c) Bushehr – Method 1



(d) Bushehr – Method 2

Figure 2: Field-word count distributions for Hormozgan and Bushehr.

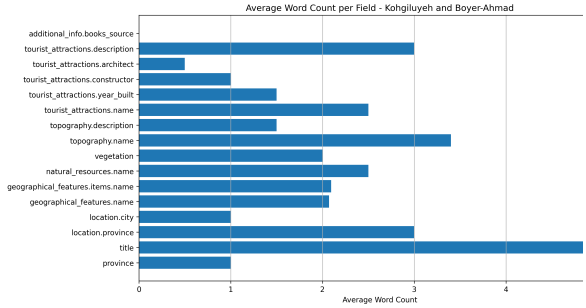
Method 2's plots show notably longer descriptions in key fields (e.g. 'tourist_attractions.description', 'natural_resources.description', and 'climate_change_impacts'), validating its effectiveness at filling gaps left by sparse PDF text in Method 1. Despite processing fewer raw chunks, Method 2 yields richer, more complete datasets suitable for downstream cultural and geographic analyses.

7 Conclusion

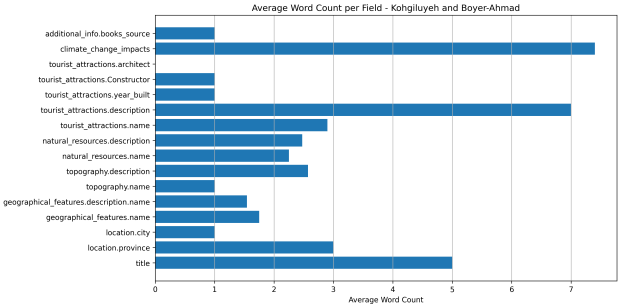
In this work, we have presented two complementary methodologies for the automated extraction and structuring of province-level cultural and geographic data from heterogeneous text sources. Method 1 employs a lightweight, quantized Gemma-3 model to parse raw PDF content into a rigid JSON schema, demonstrating how local inference can yield reliable, repeatable results even on modest hardware. Method 2 extends this pipeline by leveraging GPT-4's advanced reasoning and built-in web search capabilities to enrich sparse source text with real-time information from authoritative online repositories. Through a detailed comparison across six diverse Iranian provinces, we have shown that Method 2 substantially increases descriptive depth—particularly in fields such as natural resources, tourist attractions, and climate impacts—while maintaining schema fidelity.

Our labeling and refinement workflow, which integrates LabelStudio, dual human review, and LLM-based adjudication, further ensures that only high-quality, schema-compliant records enter the final datasets. This multi-stage process of extraction, annotation, and revision produces a corpus that is both rich in regional detail and rigorously validated for downstream use.

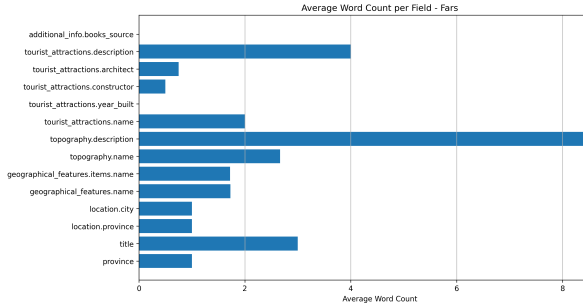
Looking ahead, these structured datasets can serve as a foundation for fine-tuning culturally aware language models, powering applications in regional information retrieval, digital heritage preservation, and targeted NLP services. Future work will explore semi-supervised approaches to reduce labeling overhead, cross-province transfer learning to generalize our schema to new languages and domains.



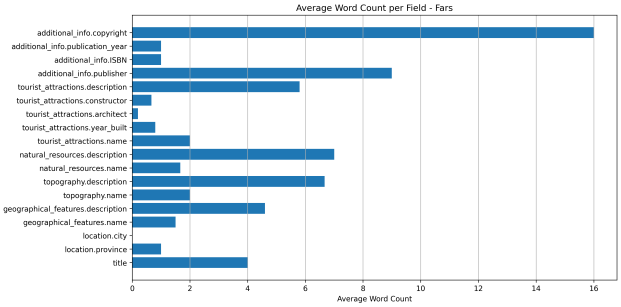
(a) Kohgiluyeh & Boyer-Ahmad – Method 1



(b) Kohgiluyeh & Boyer-Ahmad – Method 2

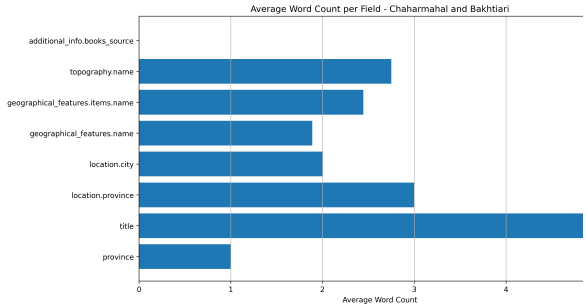


(c) Fars – Method 1

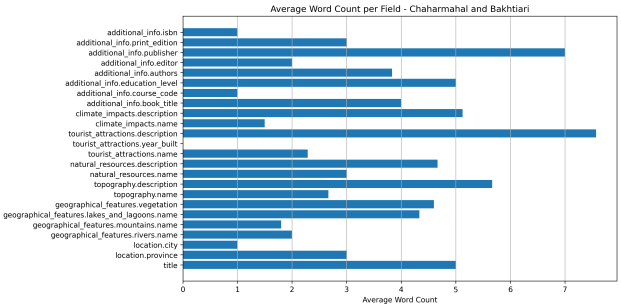


(d) Fars – Method 2

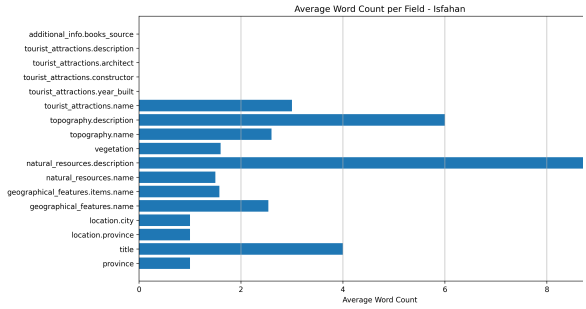
Figure 3: Field-word count distributions for Kohgiluyeh & Boyer-Ahmad and Fars.



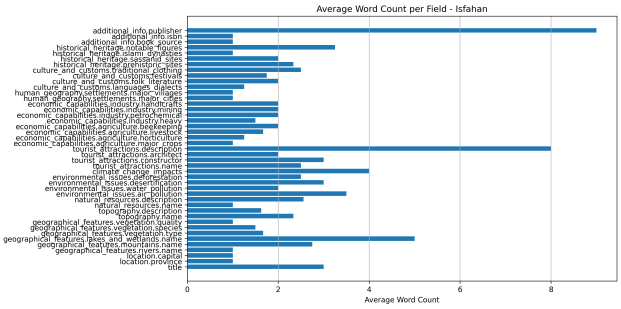
(a) Chaharmahal & Bakhtiari – Method 1



(b) Chaharmahal & Bakhtiari – Method 2



(c) Isfahan – Method 1



(d) Isfahan – Method 2

Figure 4: Field-word count distributions for Chaharmahal & Bakhtiari and Isfahan.