ChatGPT

Patient Data Extraction Pipeline from a Scientific Article

Overview

We design a **minimal working prototype** that reads a scientific article (PMID: 32679198) in PDF form and extracts structured patient-level data as JSON records. The goal is to replicate a detailed table of patient information using **Large Language Model (LLM)** extraction (DeepSeek V3 0324 via OpenRouter API) along with programmatic validation and normalization. The solution is organized into modular components with an emphasis on **accuracy and schema compliance** over raw speed. We also outline how to extend this pipeline to handle other document formats (HTML, XML, OCR for scans) and new document types (patents, clinical trial reports) in the future. All code and comments are provided in English for clarity.

Pipeline Components and Design

Our pipeline consists of the following stages:

1. **PDF Parsing** – Reading and segmenting the PDF into structured text (sections, paragraphs, tables).
2. **Context Retrieval** – Finding relevant text fragments for each data field (using semantic search with embeddings).
3. **LLM-based Extraction** – Prompting the DeepSeek model (via OpenRouter) with context to produce a **strict JSON** object per patient, following a given schema.
4. **Normalization** – Cleaning and standardizing the LLM output (gene symbols to HGNC standard, controlled vocabulary for categorical fields, uniform formats for ages/dates, etc.).
5. **Validation** – Enforcing the JSON Schema and custom rules to ensure each patient record is complete and consistent [1] .
6. **Alignment with Ground Truth** – Comparing the extracted records to a manually curated CSV (ground truth) and generating a **diff report** highlighting missing, mismatched, or extra fields. Where possible, we link each extracted value to its source text snippet in the article for traceability.

Each component is implemented as an independent module (akin to an "agent" for that task), making the codebase extensible. Below, we discuss each component and provide code snippets. Finally, we demonstrate how the pieces integrate into an end-to-end pipeline.

1. PDF Parsing and Text Segmentation

The PDF is first parsed to extract raw text content. We use **PyMuPDF** (`fitz` library) to load the PDF and iterate over pages. To preserve structure, we segment the text by sections and paragraphs. In this article, each patient is described under a section (e.g., "3.2 Patient 1") and there are summary tables (Table 1: clinical features, Table 2: gene mutations). We will:

- **Extract plain text** from the PDF.
- **Clean** the text: remove page headers/footers, line numbers and artifacts introduced by the PDF format (for example, the pre-proof PDF shows line numbers like 137, 154 which we strip out). We also join broken lines into coherent paragraphs.
- **Identify sections** for each patient. We detect headings like "Patient 1", "Patient 2", etc., and use them to split the text so that we have an isolated block of text for each patient's case description. This helps focus the LLM on one patient at a time and later allows aligning each JSON to a patient ID.
- Optionally, **extract tables** if needed. In this case, much of the patient data is also summarized in Table 1 (clinical features) and Table 2 (mutations). We can parse these by locating the table in text and splitting columns. However, for simplicity, we may rely on the narrative text and the LLM to capture those details. (For a robust solution, one could integrate a PDF table parser like Camelot or extract table text and feed it to the LLM as additional context.)

Code – PDF Parsing and Sectioning: Below is a code snippet using PyMuPDF to read the PDF and segment it by patient. We compile a dictionary `patient_texts` mapping patient IDs to their textual description. We also capture any relevant global context (like the tables or intro) that might aid extraction.

```
import fitz
import re

class PDFParser:
    def __init__(self, pdf_path: str):
        self.pdf_path = pdf_path

    def extract_text(self) -> str:
        """Extract raw text from PDF."""
        doc = fitz.open(self.pdf_path)
        text = ""
        for page in doc:
            text += page.get_text()  # Extract text from each page
        doc.close()
        return text

    def clean_text(self, text: str) -> str:
        """Clean PDF text by removing artifacts (line numbers, excessive
whitespace, hyphenation)."""
        # Remove standalone line numbers (e.g., lines that contain only digits)
        text = re.sub(r'(?m)^[0-9]+\s*$', '', text)
        # Remove hyphenation at line breaks (e.g., "develop-\nment" ->
"development")
        text = text.replace("-\n", "")
        # Replace newline+spaces with space, and multiple newlines with two
newlines (paragraph breaks)
        text = re.sub(r'\n\s*', ' ', text)
        text = re.sub(r'\n{2,}', '\n\n', text)
        return text.strip()

    def segment_by_patient(self, text: str) -> dict:
        """Split the cleaned text by patient sections, returning {patient_id:
text}."""
        patient_sections = {}
        # Use a regex to find "Patient X" headings (assuming "Patient 1",
"Patient 2", etc. in text)
        pattern = re.compile(r'(Patient\s+(\d+))', flags=re.IGNORECASE)
        matches = list(pattern.finditer(text))
        for i, match in enumerate(matches):
            pid = match.group(2)  # capture the patient number
            start_idx = match.end()  # end of "Patient X" match
            # end at the start of the next patient or end of text
            end_idx = matches[i+1].start() if i+1 < len(matches) else len(text)
            section_text = text[start_idx:end_idx].strip()
            # Prepend the "Patient X" heading back to section (for clarity)
            section_text = f"{match.group(1)}\n{section_text}"
            patient_sections[pid] = section_text
        return patient_sections

    def parse(self):
        raw_text = self.extract_text()
        clean = self.clean_text(raw_text)
        sections = self.segment_by_patient(clean)
        return sections

# Usage:
pdf_path = "PMID32679198.pdf"
parser = PDFParser(pdf_path)
patient_texts = parser.parse()
print(f"Extracted sections for {len(patient_texts)} patients.")
# patient_texts["1"] would contain the text description for Patient 1, etc.
```

In the code above, `PDFParser.parse()` returns a dictionary like `{"1": "Patient 1\nP1 is female and had normal development until ...", "2": "Patient 2\nP2 is female ...", ...}`. We detect patient sections by regex. We removed line artifacts and joined broken lines so that each section's narrative reads naturally, which will help the LLM interpret it correctly. We can also separately extract the table text (for example, by searching for "Table 1" in `raw_text`), but in this prototype we rely on the narrative for each patient.

2. Context Retrieval for Relevant Fragments

To focus the LLM on specific information, we implement a **Retriever** that can fetch the most relevant text fragments for a given patient and field. This is especially useful for long documents or when the schema has many fields, ensuring the prompt to the LLM stays within context limits and contains only pertinent information.

We use a **semantic similarity search** approach with embeddings: we split the text into chunks (such as sentences or small paragraphs), embed them using a model from **Sentence-Transformers**, and index them with **FAISS** for fast similarity lookup [2]. This allows us to retrieve, for example, the snippet mentioning the "age of onset" or the "mutation identified" for a patient, even if exact keywords differ (semantic search can handle synonyms and paraphrasing). Semantic search is more robust than simple keyword search for this task, since medical descriptions might use varied phrasing [3] [2].

Design:
- We prepare a list of text fragments, each labeled with the patient ID it refers to. Fragments could be individual sentences or a few sentences together. We'll mostly derive these from the `patient_texts` obtained above by further splitting on sentence boundaries or bullet points. - We compute embeddings for all fragments using a pre-trained model (e.g., `sentence-transformers/all-MiniLM-L6-v2` for speed). - For each query (like *"Patient 1 age of onset"* or *"Patient 1 mutation"*), we also embed the query and use FAISS to find the top-N most similar fragments in vector space. We can then include those fragments in the prompt for the LLM. - In our case, since each patient section is relatively short (a few paragraphs) we might simplify by retrieving the entire patient section plus perhaps specific lines from the tables. But the retriever is designed so that if needed, it can target finer details. This modular retriever becomes more crucial with larger documents or for future extensions where data might be scattered.

Code – Building and Querying the Retriever:

```python
import faiss
from sentence_transformers import SentenceTransformer, util

class Retriever:
    def __init__(self, patient_texts: dict):
        self.patient_texts = patient_texts
        # Split each patient's text into smaller fragments (sentences or pairs
of sentences)
        self.fragments = []       # list of fragment texts
        self.fragment_meta = []   # list of (patient_id, fragment_text)
        for pid, text in patient_texts.items():
            # Simple sentence splitting by period (could use nltk or spacy for
more accuracy)
            sentences = [s.strip() for s in re.split(r'(?<=\.)\s', text) if s]
            for sent in sentences:
                self.fragments.append(sent)
                self.fragment_meta.append((pid, sent))
        # Compute embeddings for all fragments
        self.model = SentenceTransformer("sentence-transformers/all-MiniLM-L6-
v2")
        self.embeddings = self.model.encode(self.fragments)
        # Build FAISS index for quick similarity search
        d = self.embeddings.shape[1]
        self.index = faiss.IndexFlatIP(d)  # cosine similarity can use inner
product on normalized vectors
        # Normalize embeddings for cosine similarity
        faiss.normalize_L2(self.embeddings)
        self.index.add(self.embeddings)

    def query(self, patient_id: str, field_query: str, top_k: int = 5) -> list:
        """Retrieve top-K relevant fragments for a given patient and field
query."""
        # Form a combined query like "Patient 1 <field_query>"
        query_text = f"Patient {patient_id} {field_query}"
        # Embed the query and search in index
        q_vec = self.model.encode([query_text])
        faiss.normalize_L2(q_vec)
        D, I = self.index.search(q_vec, top_k)
        results = []
        for idx in I[0]:
            pid, fragment = self.fragment_meta[idx]
            if pid == patient_id:  # ensure the fragment is from the same
patient
                results.append(fragment)
        return results

# Usage:
retriever = Retriever(patient_texts)
fragments = retriever.query("1", "age of onset", top_k=3)
print("Top fragments for Patient 1 'age of onset':", fragments)
```

In this retriever implementation, we embedded all patient sentences. The `query` method takes a patient ID and a field description, and returns the most semantically relevant fragments from that patient's text. We ensure we only return fragments from the *same* patient (so Patient 1's query won't return Patient 2's text even if semantically similar). This step uses FAISS to efficiently find nearest neighbors in the embedding space. In practice, we will call `retriever.query` for each field or group of fields we want context for, and aggregate these snippets for the prompt.

Note: For this small dataset, one could also fetch context by simple string matching (e.g., find the line containing "onset" in patient's text). However, the semantic approach is more scalable and can handle cases where wording differs (e.g., “symptoms began at 4 months” vs “onset at age 0.33 years”). We included it here to future-proof the pipeline for more complex scenarios or longer texts.

3. LLM Extraction with Few-Shot Prompting

We use the **DeepSeek V3 0324** LLM (via OpenRouter API) to extract a structured JSON object for each patient. The model will be prompted with the context (text fragments) from the previous steps and instructed to output a JSON **strictly following our schema**. Few-shot examples and rules are embedded in the prompt to guide the model towards correct formatting and content. Accuracy is paramount: we want the model to **quote or closely paraphrase** the source for each field rather than hallucinate, and to include all relevant details.

Prompt Design:
- **System Instruction:** We set the stage by telling the model its task (e.g., *"You are an assistant extracting structured data for patients. Output only valid JSON."*). We can also provide the JSON schema or a description of the fields as part of the system message, so the model knows exactly what keys to include.
- **Few-Shot Examples:** We include one or two examples of a patient description and the desired JSON output. For instance, we might create a synthetic mini-example like a Patient 0 with a short description and a JSON object filled accordingly. This teaches the model the pattern of output (especially important to enforce things like enumeration values: e.g., "sex": "f" not "female", or alive/dead as 0/1).
- **User Prompt (for each patient):** We then provide the actual context for that patient, which could be the whole patient section or the concatenated relevant fragments retrieved. We ask the model to produce the JSON for this patient. We also explicitly instruct: *"Only output a JSON object with the following fields…"* to prevent any extra prose. Using a **structured output format** like JSON ensures the result is easily machine-readable and consistent [4], which is crucial for downstream processing (no tolerance for free-form variations in date formats, etc.).

Because the model must adhere to a schema, we might leverage OpenAI-compatible **function calling** or a structured output parser in LangChain. However, since OpenRouter's interface is OpenAI-like, we'll proceed by manual prompt construction for clarity. (If using LangChain, one could use a `PydanticOutputParser` to auto-generate the expected format from the schema [5], which can help validate the LLM's output.)

Code – Calling the OpenRouter API for DeepSeek:

```
import os
import json
import requests

class Extractor:
    def __init__(self, api_key: str, schema: dict):
        self.api_key = api_key
        self.schema = schema  # JSON schema loaded as dict (from
table_schema.json)
        # Construct a template for the user prompt including an example
        self.few_shot_example = self._build_few_shot_example()

    def _build_few_shot_example(self) -> str:
        """Create a few-shot example: a dummy patient description and
corresponding JSON."""
        # Example narrative and JSON (this should be short but illustrative)
        example_text = (
            "Patient X is a male who had onset of symptoms at 6 months. "

"He was born from a normal pregnancy and had no dysmorphic features. "
```

```
            "Genetic testing revealed a mutation in GENE1. He is currently 2 
years old and alive."
        )
        example_json = {
            "pmid": 12345678,
            "patient_id": "Patient X",
            "sex": "m",
            "year_of_birth": None,
            "last_seen": 2.0,
            "age_of_death": None,
            "_0_alive_1_dead": 0,
            "age_of_onset": 0.5,
            "mt_oder_ndna": "n",
            "gene": "GENE1",
            "gene_1": "",
            "gene_2": "",
            "mutations": "GENE1 mutation",
            # ... (other fields as needed, or leave as null/empty if not in 
example narrative)
        }
        # Note: None in Python would be null in JSON output.
        example_json_str = json.dumps(example_json, ensure_ascii=False)
        # We'll place the example in the prompt as: [Example Text] -> [Example 
JSON]
        return f"{example_text}\n**Expected JSON Output:**\n{example_json_str}
\n\n"

    def extract_patient(self, patient_id: str, context_text: str) -> dict:
        """Call the LLM to extract patient info as JSON."""
        # Compose the prompt
        system_msg = (
            "You are a medical information extraction assistant. "
            "Extract the patient information as a JSON object following the 
schema. "
            "Use only the given context and do not invent data. If a field is 
not mentioned, use null or empty string."
        )
        user_msg = (
            f"{self.few_shot_example}"
            f"Context for {patient_id}:\n{context_text}\n\n"
            "Now extract the data for this patient and output **only** the JSON 
object."
        )
        # Prepare the API call
        url = "https://openrouter.ai/api/v1/chat/completions"
        headers = {
            "Authorization": f"Bearer {self.api_key}",
            "Content-Type": "application/json",
            # Specify the DeepSeek model:
            "X-Model": "deepseek/deepseek-chat-v3-0324:free"
        }
        payload = {
            "messages": [
                {"role": "system", "content": system_msg},
                {"role": "user", "content": user_msg}
            ],
            "temperature": 0.0,  # deterministic output
            "max_tokens": 1000   # enough to output the JSON
        }
        response = requests.post(url, headers=headers, json=payload)
        response.raise_for_status()
        result = response.json()
        # The response is expected to follow OpenAI's format
        model_reply = result["choices"][0]["message"]["content"]
        # Parse the JSON from the model's reply
        try:
            data = json.loads(model_reply)
        except json.JSONDecodeError:
            # If the model didn't produce valid JSON, we can try to fix or
return None
            data = None
        return data

# Usage:
API_KEY = os.getenv("OPENROUTER_API_KEY")  # ensure your OpenRouter API key is
set in env
schema = json.load(open("table_schema.json"))
extractor = Extractor(API_KEY, schema)
for pid, text in patient_texts.items():
    patient_data = retriever.query(pid, "")  # (Optionally retrieve context;
here just using whole text)
    context = " ".join(patient_data) if isinstance(patient_data, list) else text
    result = extractor.extract_patient(f"Patient {pid}", context)
    print(f"Extracted JSON for Patient {pid}: {result}")
```

In this code, we send a POST request to OpenRouter's API endpoint with the DeepSeek model specified in the `X-Model` header. The prompt includes: - A system message with general instructions/rules. - A user message containing a **few-shot example** (we constructed a hypothetical Patient X example and its JSON) and then the actual context for the target patient followed by the instruction to output JSON.

We set `temperature: 0` to minimize randomness, yielding consistent outputs. The model's reply is expected to be JSON text; we then `json.loads` it into a Python dict. If the model returns extra text or formatting issues, one might need to post-process or re-prompt (e.g., if `json.loads` fails, we could try to extract the JSON substring). In practice, careful prompt design (as above) plus validation will handle most issues.

Few-Shot and Format Enforcement: The few-shot example and explicit schema cues help the model stick to the output format. By providing an example JSON, we demonstrate the exact styling (e.g., use `null` for missing numeric fields, empty string for missing string fields, certain enumerations like "m"/"f"). This significantly reduces the chance of errors and "hallucinated" outputs, as the model knows *precisely* what we expect [4]. In more advanced scenarios, we could use tools like LangChain's `OutputParser` to automatically inject format instructions or even use function-calling with a schema, but our approach here keeps things straightforward.

4. Data Normalization

Once we receive the raw JSON output from the LLM, we apply a **Normalizer** module to clean and standardize the data fields. This step ensures that values conform to controlled vocabularies and formats required by our schema and downstream usage. Normalization rules include:

- **Gene Symbols:** Convert gene names to their official HGNC symbols. For example, if the model output a gene name in a variant form or lowercase, or an outdated synonym, we map it to the correct symbol (e.g., "Thiamine transporter 2" -> "SLC19A3"). In our case, genes like *SLC19A3, SLC25A19, TPK1* are already symbols, but if a discrepancy like "SLC25A20" appears (a potential model or manual error), we detect it. We could use an external service or a lookup table of known gene symbols to validate this.
- **Controlled Vocabulary for Sex:** The schema expects `"sex": "m"` or `"f"`. If the LLM output "male" or "female", we map those to the single-letter codes. Similarly for fields like `"mt_oder_ndna"` (should be `"m"` or `"n"` for mitochondrial vs nuclear DNA), or pregnancy/birth fields (`"n"` for normal, `"c"` for complicated, etc.). The normalizer will have dictionaries for these conversions.
- **Numeric Formats for Ages:** The schema uses numeric years (with decimals) for ages (e.g., `age_of_onset`, `last_seen`). We convert any textual age ("2 months", "2.5 years") into a float representing years. For example, "2 months" -> 0.17 (approximately), "2 years 6 months" -> 2.5. We decide on a consistent decimal rounding (perhaps two decimal places) to match the ground truth formatting. Dates, if any, could be normalized to a standard date format or age in years as required.
- **Boolean or Categorical Indicators:** The output has fields like `0_alive_1_dead` which should be 0 or 1. If the model mistakenly output True/False or "alive"/"dead", we map those to 0/1. Similarly, if any of the milestone fields or outcome fields have categorical values, we ensure they match expected terms (the schema enumerations or a standard list, possibly using the HPO terms listed in the CSV header).
- **Free-text Sanitization:** Remove any trailing spaces, unify units (if the LLM included units like "mmol/L" in values where they should be numeric only, we might strip the unit or put the value in a separate field if needed). But since the schema likely expects numeric values for lab results, the model should output them as numbers or we can strip out non-numeric characters.
- **Missing Data:** Ensure missing fields are properly represented (empty string or null). For example, in our schema, if `year_of_birth` is not provided in the article, we keep it `null`. The LLM might already do this if instructed; otherwise, the normalizer sets missing numeric fields to null and missing string fields to `""` (or vice versa as the schema defines).

Code – Normalization Functions:

```
import math

class Normalizer:
    def __init__(self):
        # Define mapping dictionaries for controlled vocabularies
        self.sex_map = {"male": "m", "female": "f", "m": "m", "f": "f"}
        self.alive_map = {"alive": 0, "dead": 1, "0": 0, "1": 1}
        self.mt_map = {"mtdna": "m", "mitochondrial": "m", "nuclear": "n",
"ndna": "n", "m": "m", "n": "n"}
        # ... (similar maps for pregnancy, birth, outcome codes if needed)

    def normalize_record(self, record: dict) -> dict:
        rec = record.copy()
        # Sex normalization
        if "sex" in rec and isinstance(rec["sex"], str):
            sex_val = rec["sex"].strip().lower()
            rec["sex"] = self.sex_map.get(sex_val, rec["sex"])
        # Alive/Dead normalization
        if "_0_alive_1_dead" in rec:
            val = rec["_0_alive_1_dead"]
            if isinstance(val, str):
                val = val.strip().lower()
            rec["_0_alive_1_dead"] = self.alive_map.get(val,
rec["_0_alive_1_dead"])
        # mt or nDNA normalization
        if "mt_oder_ndna" in rec and isinstance(rec["mt_oder_ndna"], str):
            val = rec["mt_oder_ndna"].strip().lower()
            rec["mt_oder_ndna"] = self.mt_map.get(val, rec["mt_oder_ndna"])
        # Age fields normalization (convert to number of years)
        for field in ["age_of_onset", "last_seen", "age_of_death"]:
            if field in rec and rec[field] is not None:
                rec[field] = self._parse_age(rec[field])
        # Gene symbols normalization: example check using a simple approach
        if "gene" in rec and rec["gene"]:
            rec["gene"] = rec["gene"].upper()  # upper-case it
            # If a gene symbol is known to be wrong (e.g., SLC25A20 instead of
SLC25A19), fix it:
            if rec["gene"] == "SLC25A20":
                rec["gene"] = "SLC25A19"  # (assuming a known correction or use
external lookup)
        # Mutations field: we might ensure it includes gene symbol at start or
standardized formatting.
        if "mutations" in rec and isinstance(rec["mutations"], str):
            rec["mutations"] = rec["mutations"].replace("; ", "; ")  # trivial
example: ensure consistent separator spacing
        # ... Additional normalization rules as needed ...
        return rec
    def _parse_age(self, age_val):
        """Parse age from text or number to a float (years)."""
        if age_val is None or age_val == "":
            return None
        # If already a number (int/float), return as float
        if isinstance(age_val, (int, float)):
            return float(age_val)
        # If it's a string like "2 y 6 m" or "8.5 m" etc.
        s = age_val.strip().lower().replace('months', 'm').replace('month', 'm')
\
                                   .replace('years', 'y').replace('year', 'y')
        # Extract any numbers
        years = 0.0
        # e.g., "2y6m", "1 m", "8.5 m"
        year_match = re.search(r'(\d+(\.\d+)?)\s*y', s)
        month_match = re.search(r'(\d+(\.\d+)?)\s*m', s)
        if year_match:
            years += float(year_match.group(1))
        if month_match:
            months = float(month_match.group(1))
            years += round(months / 12.0, 4)
# convert months to years (rounded to 4 decimal for precision)
        # If the string was something like "27 months", that would only match
month_match and be converted.

# If it was "6 days" or "15 d", we'll ignore as it’s negligible relative to
years for our data.
        return years if years != 0.0 else None

# Usage:
normalizer = Normalizer()
normalized_results = []
for pid, data in extracted_data.items():  # assume extracted_data is
{patient_id: raw_json}
    norm = normalizer.normalize_record(data)
    normalized_results.append(norm)
```

In this normalization code, we handle a few key fields. The `_parse_age` function converts various textual representations of age into a numeric year value. We also handle mapping for sex and alive status. **Gene normalization** is shown in a simplified way; in a real scenario we might query an external service or have a list of valid gene symbols. For instance, using the `mygene` library or an offline list to validate "SLC25A20" and realize it's likely a mistake for "SLC25A19" given the context of THMD4.

The normalizer ensures that after this step, our patient records are **consistent** and **comparable** to the ground truth. This addresses the issue that LLMs might output semantically correct info in a slightly different form (e.g., "female" vs "f") – by normalizing, we bring it to the expected form before comparison. Having a consistent data format is crucial for robust data pipelines [1].

5. Validation and Schema Enforcement

After normalization, we run each record through a **validator** to ensure it complies with the JSON Schema (`table_schema.json`). We utilize Python's `jsonschema` library to validate each JSON object against the schema [6]. This catches any missing required fields, type mismatches, or value out-of-range issues. It essentially guarantees that our extracted data adheres to the contract we expect (for example, numeric fields are numbers, enumerated fields have allowed values, etc.).

Additionally, we can implement **custom rules** beyond the schema: for example, if a patient is marked dead (`_0_alive_1_dead = 1`), then `age_of_death` should not be null (and likely equals `last_seen`), and conversely for alive patients; or if `gene_1` and `gene_2` are present, maybe `gene` should be a primary gene etc. Such consistency checks can be done in code and any deviations flagged as errors or warnings.

Code – Schema Validation:

```
from jsonschema import validate, ValidationError

schema = json.load(open("table_schema.json"))

valid_records = []
for record in normalized_results:
    try:
        validate(instance=record, schema=schema)
        valid_records.append(record)
    except ValidationError as e:
        print(f"Validation error for patient {record.get('patient_id')}:
{e.message}")

# We could attempt to auto-fix certain errors here or mark the record as
invalid.
        # For now, just report it.
```

This will raise an error if, for example, the model omitted a field that the schema requires or put a string where a number is expected. By catching the exception, we can log it and decide how to handle it. In a minimal prototype, we may simply report it. In a production setting, we could attempt a second-pass extraction for missing info or have a human review.

Example: If the LLM failed to extract the APGAR scores (which are in the schema), the validator might complain those fields are missing (if marked required). We could then investigate that as a “missing field” in our diff report.

6. Alignment with Ground Truth and Diff Report

Finally, we compare our extracted, normalized records with the **manually curated CSV** provided (ground truth). We want to produce a **structured diff** that highlights:

- **Missing fields**: Data present in the ground truth but missing from our extraction.
- **Mismatched values**: Fields where our extracted value differs from the ground truth. This could be due to extraction errors or normalization differences. We should be tolerant to minor formatting differences (which normalization mostly handles), focusing on substantive discrepancies (e.g., gene name mismatch, an age that's off by more than minor rounding, etc.).
- **Extra fields**: Any field our output has that it shouldn't (unlikely if we follow the schema strictly, but if the LLM added an unexpected field, the validator would have caught it earlier). More relevant might be if the LLM hallucinated an extra symptom that wasn't in the source – but since we confine to schema fields, it would manifest as a mismatch (value present vs ground truth says none).

For each difference, we will try to **link to the source text** that could explain either value. This is important for transparency: if our extraction is right and the ground truth is wrong (or vice versa), the source text will back one of them. Because we have the `patient_texts` and possibly table text, we can search within those for the mention of the value. For example, if our output for "gene" is "SLC25A20" and ground truth is "SLC25A19", we search the article for "SLC25A20" (likely not found) and "SLC25A19" (likely found in Table 2 or discussion). We then include the snippet (e.g., *"... two patients, four novel mutations (c.194C>T... were identified in SLC25A19, supporting a diagnosis of THMD4."* [7]) as evidence.

Similarly, if an age or symptom is contested, we find where in text it appears. The retriever from earlier can be reused to find sentences containing certain keywords or values.

Code – Diff Report Generation:

```
import csv

# Load ground truth CSV into list of dicts for easy comparison
gt_records = []
with open("manually_processed.csv") as f:
    reader = csv.DictReader(f)
    for row in reader:
        if row["PMID"] == "32679198":
            gt_records.append(row)
# We assume patient ordering in CSV corresponds to "Patient 1", "Patient 2",
etc.
# Create a mapping from patient ID to ground truth record and to our extracted
record
gt_map = {row["patient ID"]: row for row in gt_records}
ext_map = {rec["patient_id"]: rec for rec in valid_records}

diff_report = []
for pid, gt in gt_map.items():
    ext = ext_map.get(pid)
    if not ext:
        diff_report.append({"patient_id": pid, "issue": "missing_all",
                             "message": f"No data extracted for {pid}"})
        continue
    # Compare each field
    for field, gt_val in gt.items():
        # Skip PMID and patient ID fields for diff (or convert types of GT
values if needed)
        if field.lower() in ("pmid", "patient id"):
            continue
        ext_val = ext.get(field_to_json_key(field), None)  # map CSV header to
JSON key if needed
        # Normalize types for comparison (e.g., strings vs numbers)
        if ext_val is None or ext_val == "" or (isinstance(ext_val, float) and
math.isnan(ext_val)):
            ext_present = False
        else:
            ext_present = True
        if gt_val in [None, "", "NA", "N/A"]:
            gt_present = False
        else:
            gt_present = True
        if not gt_present and not ext_present:
            continue  # both missing, that's fine
        if gt_present and not ext_present:
            diff_report.append({"patient_id": pid, "field": field, "issue":
"missing",
                                "expected": gt_val, "found": None})
        elif not gt_present and ext_present:
            diff_report.append({"patient_id": pid, "field": field, "issue":
"extra",
                                "expected": None, "found": ext_val})
        else:
            # Both have values, compare (with tolerance for numeric differences)
            if str(gt_val).strip() != str(ext_val).strip():
                # If numeric, allow small rounding differences
                try:
                    if abs(float(gt_val) - float(ext_val)) < 0.01:
                        continue  # consider essentially equal
                except:
                    pass
                # Log mismatch
                diff_report.append({"patient_id": pid, "field": field, "issue":
"mismatch",
                                    "expected": gt_val, "found": ext_val})
```

(The above is a simplified comparison – in practice, we'd refine type conversions and handle the many fields programmatically. Also, `field_to_json_key` *would convert CSV headers like "patient ID" or "Age of onset" to our JSON keys like* `"patient_id"` *or* `"age_of_onset"` *if they differ. We assume they are similar here.)*

We iterate through each patient. If a patient is completely missing in our extraction, we note that. Otherwise, for each field we categorize the discrepancy. Finally, we produce `diff_report`, which is a list of dictionaries or could be directly formatted to a human-readable report (e.g., markdown table or just printed statements). For example, a diff entry might look like:

```
{"patient_id": "Patient 6", "field": "gene", "issue": "mismatch", "expected":
"SLC25A19", "found": "SLC25A20"}
```

indicating a gene mismatch for Patient 6.

Linking to Source Text: For each diff item, we can retrieve a source snippet to justify the expected vs found values. We can leverage our earlier `patient_texts` and `retriever`. For instance, if `issue == "mismatch"`, we take `expected` value and `found` value, search the PDF text for each, and attach any snippet found. In the gene example, searching the PDF for "SLC25A20" likely yields nothing (since the article did not mention SLC25A20), whereas "SLC25A19" is found in multiple places (Table 2, and narrative). We could attach: *Source: "...identified in SLC25A19, supporting a diagnosis of THMD4."* [7] to show that the article supports SLC25A19. For numeric differences like ages, we search the context for that age (or corresponding textual age). E.g., if ground truth says onset 0.66 years (8 months) but our output was 0.75 (9 months), we find in text "8 months" or "9 months" in Patient 3's section to see which is correct. This part can be semi-automated: use regex or the retriever to find occurrences of the expected or found values (or their synonyms) in `patient_texts[pid]` or in the whole text. Then include a snippet around 50-100 chars as evidence.

We compile the diff report as needed (e.g., print out a list of issues per patient). This helps to pinpoint where the extraction needs improvement.

Future Extensions and Multi-Format Ingestion

Ingestion of HTML, TXT, PMC XML/JSON, OCR: The parsing module can be extended or new ones can be created for other input formats:
- **HTML:** Use an HTML parser (BeautifulSoup, lxml) to extract text from web pages or HTML-formatted papers. The structure (headings, paragraphs) can often be retained from HTML tags. For example, if an article is in HTML or XML (such as a PMC Open Access XML), we can directly parse sections (<sec>, <p>, <table> tags, etc.) to get clean text without needing OCR.
- **Plain Text:** If the input is already plain text (or a TXT file), parsing is trivial – we might only need to split it into sections by some known delimiters or headings.
- **PMC NLM XML/JSON:** Many scientific articles in PubMed Central have XML or JSON formats with semantic tagging. These can give us structured data directly (e.g., `<table>` elements, `<sec id="patient1">` etc.). We could write a parser that reads the XML, pulls out each patient case description and any table rows. This would likely be even more precise than PDF parsing, because the XML might label table columns or contain metadata. Our pipeline could detect if an XML/JSON is available and prefer that for input.

- **OCR for Scanned Documents:** If only a scanned PDF or an image is available, we incorporate an OCR step. We could use Tesseract via Python (`pytesseract`) or cloud OCR APIs to extract text. Additionally, specialized tools or libraries (like `layout-parser` or the Unstructured library) can maintain layout information – useful if we want to reconstruct tables from an image. This step would feed the extracted text into the same processing pipeline thereafter. Tools like **LLMWhisperer** have been noted to handle extraction from scans with high accuracy [8]; such a tool could be plugged in as an alternative text extractor for scanned PDFs.

To accommodate these, our codebase can have an **abstract DocumentParser** class with concrete subclasses: `PDFParser`, `HTMLParser`, `XMLParser`, `OCRParser`, etc. The orchestrator can select the parser based on file type (or try OCR if PDF text extraction yields poor results). Each parser outputs a standardized text structure (e.g., a dict of patient sections plus any global context). This design isolates format-specific logic in one place.

Extensible, Agent-like Modularity: We envision the pipeline as a set of agents or modular components that can be recombined or extended for new document types and schemas. For instance:

- If we want to process **patent documents** or **clinical trial reports**, the schema of information to extract will differ (patents might have inventors, claims, etc., clinical trials might have patient cohorts, outcomes, etc.). We can create new JSON schemas and adjust the LLM prompt (few-shot examples for that domain, different field queries for retrieval). The retrieval mechanism remains largely the same – we might retrain or select a domain-specific embedding model if needed (e.g., for legal text vs biomedical). The parsing module might change if the format is different (patents may be text or PDF, etc., but our architecture can plug in a suitable parser).
- The **Agent-like design** means each component (Parser, Retriever, Extractor, Normalizer, Validator, Aligner) could be orchestrated by a high-level controller. In a complex system, one could even employ an LLM agent to dynamically decide steps (for example, an agent could decide "the document is a scan, I should call the OCR tool first" or "I have extracted data, now I should validate it"). However, in our design we keep the flow deterministic and code-driven for reliability.
- We ensure **loose coupling**: each module communicates via simple data structures (e.g., passing text or JSON between them). This makes it easy to swap implementations. For example, if a new embedding technique comes out, we can replace the Retriever's internals without affecting the rest. Or if we want to use a different LLM or API, we just adjust the Extractor class (say, using OpenAI GPT-4 with function calling, or a local model via HuggingFace Transformers).
- The schema can be extended or changed, and only the prompt and perhaps normalizer need adjustments in tandem with it. If new fields are added (say we expand to capture lab test values in more detail), we update the JSON schema and add those to the prompt instructions. The model, if capable, will include them in output. The validator will enforce them, and the diff will naturally cover them.

Internal Documentation & Maintainability: The code is documented with docstrings and comments for clarity. We would maintain a README describing how to run the pipeline (e.g., how to set up the OpenRouter API key, which libraries to install: PyMuPDF, SentenceTransformers, FAISS, jsonschema, etc.). Logging can be added to each module for easier debugging (for example, logging what the LLM was asked vs answered, what normalization did, etc.). Accuracy is prioritized: every discrepancy is either handled or at least reported in the diff, so we can iteratively improve the prompt or logic.

In summary, the proposed system reads the PDF, isolates each patient's info, uses a powerful LLM with carefully crafted prompts to extract structured data, and then rigorously checks that data against schema and ground truth. It is built in a modular fashion that allows swapping in new data sources and expanding to new document types with minimal changes. By planning for extensibility and using controlled vocabularies and validation, we ensure the extracted data is reliable and ready for integration into databases or further analysis [9].

---

[1] [6] [9] How to Use JSON Schema to Validate JSON Documents in Python | Built In
https://builtin.com/software-engineering-perspectives/python-json-schema

[2] [3] Semantic Search — Sentence Transformers documentation
https://sbert.net/examples/sentence_transformer/applications/semantic-search/README.html

[4] [5] [8] LLMs for Structured Data Extraction from PDF | Comparing Approaches
https://unstract.com/blog/comparing-approaches-for-using-llms-for-structured-data-extraction-from-pdfs/

[7] PMID32679198.pdf
file://file-Ft7KegjfpASyCnmUDooJi1