Code for generating long document datasets

Version 0.2

Main goal
Develop a Python code to automatically extract plan and content embeddings from 3 different document types, and evaluate divergence between 2 documents' plan & content. The goal is to use it as evaluation datasets.

Python code main roles
- automatically extract plan (each section length should be <= 512 tokens) and title/abstract from 3 types of document (a pdf survey, a wikipedia article, a patent), then create an embedding for each section of the plan (ada v2 and e5-base-v2). This code will create JSON datasets (see format below) to be used for training on 2 types of task: predict a plan given a subject, generate expert content based on a given a subject and plan.
- compare 2 generated plans based on embedding (cosine similarity or MAUVE) and Rouge-L on text.
- compare 2 generated documents with the same plan on average similarity on each section the document based on embedding (cosine similarity or MAUVE) and Rouge-L based on text.

Code should be validated on automatic tests on these documents set:

1. 10 Arxiv research survey papers in PDF (selection will be provided) + 10 similar documents we will generate based on plans you will have extracted
2. 10 Wikipedia articles from URL (selection will be provided) + 10 similar documents we will generate based on plans you will have extracted
3. 10 Patents in XML (selection will be provided from 10 patents EPO) + 10 similar documents we will generate based on plans you will have extracted - IMPORTANT: DESCR and CLAIM section should be splitted in meaningful section titles (not CLAIM1, CLAIM2, DESCR1, DESCR2…).

JSON format of generated plan and content embeddings should be:


```
{
  "id": 123,
  "title": "Paper Title", "abstract": "Abstract content...",
  "title_embedding1": [0.3, 0.6, 0.9, ...],
  "title_embedding2": [0.2, 0.7, 0.8, ...],
  "plan": [
{"section_id": 1, "section": "Introduction", "content": "Introduction content...", "section_embedding1": [0.1, 0.2, 0.3, ...],, "section_embedding2": [0.1, 0.2, 0.3, ...], "content_embedding1": [0.1, 0.2, 0.3, ...]},"content_embedding2": [0.1, 0.2, 0.3, ...]},
{"section_id": 2, "section": "Related Work", "content": "Related work content...", "section_embedding1": [0.4, 0.5, 0.6, ...],, "section_embedding2": [0.4, 0.5, 0.6, ...], "content_embedding1": [0.2, 0.2, 0.3, ...]}, "content_embedding2": [0.2, 0.2, 0.3, ...]},
...
   ],
  "plan_embedding1": [0.3, 0.6, 0.9, ...],
  "plan_embedding2": [0.4, 0.2, 0.7, ...],
  "embedding1_model": "text-embedding-ada-002",
  "embedding2_model": "e5-base-v2",
  "success": true,
  "error": null
  # if extracted: "author": xxx, "keywords": xxx, "references": xxx
}
```

JSON format of comparison results

```
# Plan comparison
{
  "document_id": 123,
  "prediction_id": 456,
  "plan_similarity": {
    "embedding1_cosine_similarity": 0.85,
    "embedding1_mauve_similarity": 0.92,
    "text_rouge_l_similarity": 0.75
  }
}
# Content comparison
{
  "document_id": 123,
  "prediction_id": 456,
  "content_total_similarity": {
    "embedding1_cosine_similarity": 0.75,
    "embedding1_mauve_similarity": 0.82,
    "text_rouge_l_similarity": 0.68
  },
  "content_bysection_similarity": {
    {"section_id": 1, "embedding1_cosine_similarity": 0.75, "embedding1_mauve_similarity": 0.82,
"text_rouge_l_similarity": 0.68},
    {"section_id": 2, "embedding1_cosine_similarity": 0.55, "embedding1_mauve_similarity": 0.62,
"text_rouge_l_similarity": 0.58},
    ....
  }
}
```

Details

- Python functions raise an error if a PDF file is not accessible or the extraction fails and return error message in final JSON returned, it continues to next document. Any other error should be handled the same way.
- you should use most used libraries (e.g; huggingface, openai, layoutlmv3/donut, pypdf2/pdfminer, beautifulsoap, sklearn). It is recommended to use LayoutLMv3 or similar for PDF plan/content extraction to get a coherent extraction except if you have a better solution.
- function does not need to handle multiple PDF files at once. Parallel processing of embedding generation should be an activatable option.
- function just extract text content, it does not extract diagrams, images, and tables.
- every document should always has an abstract section. For example, if the text after the title of the document has no section named "abstract" in the plan, classify it as the "abstract" section.
- if some other metadata about the papers are extracted, such as authors are extracted they can be stored like GROBID JSON output format but should comply with JSON format output defined above.
- the ordering of the sections in the plan should be kept in the original order, ID should be incremental starting from 0.
- no summary of the paper is expected in the output but an embedding of the full plan "plan_embedding1/2".
- for embeddings, depending on option, it can either use OpenAI text-embedding-ada-002 model or a specified pre-trained Huggingface model (e5-base-v2 recommended), or “both”.
- library/installation dependencies are handled through a requirements.txt file.
- coding style is PEP8.
- solution can be a single file with functions and main.

- for large PDF files, it must process step by step to avoid memory issues.
- function should output logs all along the process if option is activated.

Code structure should look like:

0. `get_embeddings(texts: List[str], model: str) -> List[List[float]]` - This function takes one string or a list of strings and a model name, generates an embedding for each string in the list using the specified model, and returns a list of embeddings, each represented as a list of floating point numbers. This function could leverage parallel processing if it's activated to improve the efficiency of generating embeddings for a large number of texts.

1. `extract_plan_and_sectionscontent_from_arxiv_pdf(pdf_path: str) -> Dict[str, Any]` - This function takes the path to a PDF file from Arxiv, extracts the plan and sections content, and returns a dictionary containing the plan, sections, and their corresponding content.

2. `extract_plan_and_sectionscontent_from_wikipedia_page(url: str) -> Dict[str, Any]` - This function takes a URL to a Wikipedia page, extracts the plan and sections content, and returns a dictionary containing the plan, sections, and their corresponding content.

3. `extract_plan_and_sectionscontent_from_EPOpatents_xml(xml_path: str) -> Dict[str, Any]` - This function takes the path to an EPO patent XML file, extracts the plan and sections content (with meaningful section title, not CLAIM1, CLAIM2…), and returns a dictionary containing the plan, sections, and their corresponding content.

4. `divide_sections_if_too_large_in_plan_and_sectionscontent(plan_and_content: Dict[str, Any]) -> Dict[str, Any]` - This function takes an existing dictionary containing the plan and sections content (from above functions), checks if any section is too large (i.e., more than 512 tokens), divides such sections into smaller sections, generate new title (use keyword generator), and returns the updated dictionary.

5. `generate_embeddings_for_plan_and_sectionscontent(plan_and_content: Dict[str, Any], model1: str, model2: str) -> Dict[str, Any]` - This function takes the dictionary containing the plan and sections content and the model names, generates embeddings for the plan and each section's content using the specified models, and returns the dictionary updated with the embeddings.

6. `document_to_json_dataset(document_path_in: str, document_type: Str, json_path_out: str) -> JSON` - This function takes the document path in, type, and a an output path, generate the JSON using all functions above, save it to an output if not null and return JSON.

7. `documents_to_json_datasets(documents: List[Dict[str, Any]], json_path_out: str) -> JSNO` - Convert a list of documents to JSON datasets using functions above , saves it as an aggregated JSON file.

8. `compare_documents_plans(document1: Dict[str, Any], document2: Dict[str, Any], method: str) -> Dict[str, float]` - This function takes two documents, a comparison method, compares the plans of the documents using the specified method, and returns a dictionary containing the similarity scores.

9. `compare_documents_sections(document1: Dict[str, Any], document2: Dict[str, Any], method: str) -> Dict[str, Dict[str, float]]` - This function takes two documents, a comparison method, compares the sections of the documents using the specified method, and returns a dictionary containing the section-wise similarity scores.

10. `run_tests() -> None` - This function runs a series of automated tests to verify the functionality of the code.