**This technical report serves to help reader understand the code and share insightful findings.   
  
1. Introduction**

This project aims to develop a comprehensive pipeline for extracting, processing, and evaluating information from insurance policy documents. The pipeline leverages various tools and techniques including PDF to Markdown conversion, embedding models, vector stores, and JSON extraction to create structured datasets from unstructured data sources.

**2. Objectives**

1. **Standardize** the extracted data into a comprehensive schema.
2. **Extract** structured data from unstructured insurance policy documents.
3. **Evaluate** the extracted data against ground truth for accuracy and completeness.
4. **Create** cleaned and structured CSV files from the JSON data.

**3. Structure**

**3.1 Environment Setup**

Environment variables for API keys and URLs, managed through the .env file. The following keys are required:

* LLAMA\_CLOUD\_API\_KEY #Need to set up LLAMACLOUD
* OPENAI\_API\_KEY
* QDRANT\_API\_KEY #Need to set up QDRANT
* QDRANT\_URL

**3.2 Directory Structure**

* raw: Contains the raw PDF and Markdown files taken from Income’s website.
* config: Stores the YAML configuration files for different policy types.
* parsed\_documents: Holds the parsed Markdown documents created during the process of embedding.
* jsons: Contains the all extracted JSON files across all configs
* jsons/<specific policy type>: Contains the extracted JSON files based on selected config. These JSON files are used in creating CSV files
* structured\_data: Stores the all CSV files – uncleaned CSV files, cleaned CSV files and cleaned CSV files (Key Benefits)
* evaluation\_data: Contains JSON files serving as the ground truth to be evaluated against
* scripts: Contains scripts for running automated pipelines.
* notebook: Used for exploration, prototyping, and data analysis. Find develop.ipynb for final codes and analysis.

**4. Functions**

def load\_config(config\_file): # Loads a YAML configuration file

**4.1 PDF to Markdown Conversion**

PDF files are converted to Markdown format using pymupdf4llm and LlamaParse based on the file type and content.

def convert\_pdf\_to\_markdown(file\_path):

return pymupdf4llm.to\_markdown(file\_path)

def convert\_pdf\_to\_markdown\_llamaparse(file\_path, api\_key):

llama\_parse = LlamaParse(api\_key=api\_key, result\_type="markdown")

parsed\_document = llama\_parse.load\_data([file\_path])

return parsed\_document[0].text if parsed\_document else ""

def convert\_pdfs\_to\_markdown(directory):

# Implementation to convert PDFs to Markdown documents

def convert\_pdfs\_to\_markdown(directory):

# Implementation to convert PDFs to Markdown documents

def clean\_markdown(content):

# Implementation to clean and formats the Markdown content for better

Embeddings

**4.2 Chunking**

def custom\_chunk\_by\_questions(content):

# Chunks Markdown content by questions, ensuring each chunk does not exceed a specified maximum size This is only applicable to and applied on faq’s markdown for cleaner embeddings

Note: For the other markdowns, they are chunked

**4.2 Document Processing and Indexing**

Markdown documents are parsed and indexed using the Qdrant vector store for efficient querying and similarity search.

def create\_or\_get\_vector\_store\_and\_index(documents, collection\_name):

# Implementation for vector store and index creation

**4.3 JSON Extraction**

Extract JSON objects from response texts and save them in specified directories.

def apply\_prompt\_chaining(prompt\_steps, query\_engine):

# Applies a series of prompts to a query engine and accumulates

Responses This is because prompts are broken down into smaller ones for more accurate response, especially so for prompts on benefits

def extract\_json\_objects(text):

# Implementation for JSON object extraction from text

def extract\_and\_save\_json(response\_text, output\_file, specific\_output\_file):

# Implementation for saving extracted JSON objects

**4.4 Data Standardization and Evaluation**

Standardize benefits extracted from documents and evaluate the generated JSON against ground truth data.

def extract\_benefits\_from\_document(index):

# Extracts benefit names from a document using a query engine

def standardize\_benefits\_schema(index, benefits\_list):

# Implementation for standardizing benefits schema

def evaluate(generated, ground\_truth):

# Evaluates the generated JSON against the ground truth JSON,

calculating completeness, faithfulness, and accuracy scores

def calculate\_weighted\_score(generated, ground\_truth, completeness\_score, accuracy\_score):

# Calculates a weighted score based on completeness, accuracy,

product name, single premium, and regular premium Only weighted

score is reflected in evaluation\_results.csv

evaluation\_results.csv  
A table of text with black and white text

Description automatically generated

**4.5 CSV Creation**

Create CSV files from the extracted and cleaned JSON data, focusing on both key benefits and complete data.

def process\_all\_documents(directory):

# Processes all documents in a directory, extracting and standardizing benefits

def process\_policy(policy\_path, config\_path, llamaparse\_api\_key, openai\_api\_key, qdrant\_api\_key, qdrant\_url):

# Processes a single insurance policy document, generating JSON and

saving it to specified files

def create\_csv\_from\_jsons():

# Implementation for creating CSV files from JSON data

def create\_csv\_from\_jsons\_key\_benefits():

# Implementation for creating CSV files focusing on key benefits

def create\_csv\_from\_jsons\_uncleaned():

# Implementation for creating CSV files with raw data points

**5. Main Scripts**

**5.1 standardize\_schema.py**

Processes all documents to extract and standardize benefits into a comprehensive schema.

from functions import process\_all\_documents

if \_\_name\_\_ == "\_\_main\_\_":

process\_all\_documents('data/raw/termlife')

**5.2 evaluate.py**

Processes and evaluates insurance policies against configuration files, saving results in a CSV file.

import os

import logging

import pandas as pd

from dotenv import load\_dotenv

from functions import process\_all\_documents, process\_and\_evaluate\_policy

def main():

# Implementation for processing and evaluating policies

**5.3 cleaned\_csv.py**

Processes policies and creates cleaned CSV files from JSON data.

import os

import logging

import pandas as pd

from dotenv import load\_dotenv

from functions import process\_all\_documents, process\_policy, create\_csv\_from\_jsons

def main():

# Implementation for creating cleaned CSV files

**5.4 uncleaned\_csv.py**

Creates uncleaned CSV files from JSON data without additional cleaning steps.

import os

import logging

import pandas as pd

from dotenv import load\_dotenv

from functions import process\_all\_documents, process\_policy, create\_csv\_from\_jsons\_uncleaned

def main():

# Implementation for creating uncleaned CSV files

**5.5 cleaned\_key\_benefits\_csv.py**

Creates CSV files focusing on key benefits extracted from JSON data.

import os

import logging

import pandas as pd

from dotenv import load\_dotenv

from functions import process\_all\_documents, process\_policy, create\_csv\_from\_jsons\_key\_benefits

def main():

# Implementation for creating CSV files focusing on key benefits

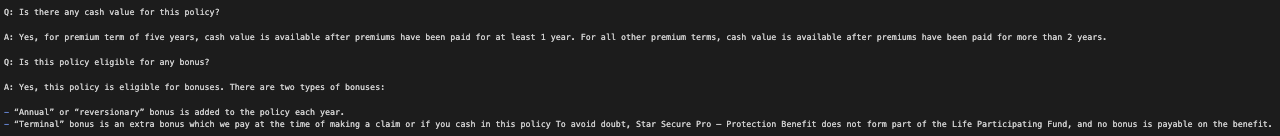
**6. Some of Considerations made:**

**6.1 Standardized Benefits**

The necessity of standardized benefits arises from the complexity involved when using a language model (LLM) to pick up and generate benefits directly from unstructured text. Without standardization, the LLM may struggle to accurately identify and categorize benefits due to the diverse terminologies and formats found in insurance policy documents. By directly specifying the benefits in a template, we ensure that the extracted data is consistent and accurately reflects the intended information.   
  
Having standardized schema also facilitates cleaner and more efficient convergence for the CSV output, ensuring that the data is uniform and ready for further analysis or integration.

**6.2 Chunking**

Initially, fixed-size chunking was experimented with as a method to divide the text into manageable parts for processing. However, it quickly became apparent that fixed-size chunking was not suitable for production use. This method often resulted in chunks that either split important contextual information across boundaries or included irrelevant data, leading to poor performance and inaccurate information retrieval.  
  
For FAQ documents, after converting all files to Markdown, a custom chunking technique was applied by splitting the content based on questions.

  
*Sample FAQ*

For other types of documents, LlamaIndex's MarkdownNodeParser was utilized.  
Pros: The parser can retain and understand the structure and hierarchy of the document, which helps in maintaining the context and relationships between different sections.

**A screenshot of a computer

Description automatically generated**  
*Sample Embedding on Qdrant (MarkdownNodeParser)*

**6.3 Prompting**

I have implemented prompt chaining - breaking down prompts into smaller, more manageable steps, in order to increase the accuracy of the extracted information. This technique allows for the retrieval of fewer top-k similar documents, thereby reducing the computational load while maintaining or even improving the precision of the responses. By applying a series of focused prompts, the model can generate more accurate and relevant outputs, especially for complex queries that require nuanced understanding and detailed information extraction.

Initially, few-shot prompts were also considered to guide the LLM in generating the desired output by providing examples. However, this approach proved to be less effective because the LLM sometimes picked up irrelevant details from the provided examples, leading to inaccuracies in the extracted data. Consequently, this approach was not suitable for our needs, as the objective was to achieve precise and reliable extraction of information from varied insurance policy documents.

**6.3 Weighted Score**

Completeness measures the extent to which the extracted data covers all the required information specified in the ground truth. It is calculated by comparing the number of data points present in the extracted JSON against the total number of expected data points in the ground truth JSON. The completeness score is given by:   
Completeness Score = (Number of Extracted Data Points) / (Total Number of Expected Data Points)

Accuracy measures how closely the extracted data matches the ground truth data in terms of both content and structure. This metric is evaluated by calculating the proportion of correctly extracted data points to the total data points. The accuracy score is given by:  
Accuracy Score = (Number of Correctly Extracted Data Points) / (Total Number of Data Points)

The weighted score combines completeness and accuracy to provide an overall measure of the extraction quality. It assigns weights to the completeness and accuracy scores based on their relative importance to the project goals. Additionally, important and easy-to-pck-up features such as product name, single premium, and regular premium are also considered in the weighted score. The weighted score is calculated as:  
Weighted Score = w\_c \* Completeness Score + w\_a \* Accuracy Score + w\_p \* Product Name Score + w\_sp \* Single Premium Score + w\_rp \* Regular Premium Score

### **6.4 Experimentation with Multi-column Condition Files**

For multicolumn condition files, I experimented with three different tools: LlamaParse, PyMuPDF4LLM, and Unstructured.io to determine the most effective method for extracting and processing the content.

LlamaParse and PyMuPDF4LLM

Both LlamaParse and PyMuPDF4LLM showed promising results in identifying and reading the columns within the documents. They were able to recognize the structure of the multicolumn format. However, when moving to the next page, the tools sometimes began reading from the right column when moving to the next page, leading to inaccuracies and loss of context in the extracted data.

Unstructured.io

Unstructured.io offered a quick approach to embedding and assigns parent and child nodes to the document elements, effectively drawing connections between related pieces of content.

However, I could not find a way to convert the structured output into Markdown format. This limitation hindered my ability to control chunking effectively, which is crucial given complexity of the documents.

### **7 References:**

<https://cloud.qdrant.io/>

<https://docs.llamaindex.ai/en/stable/llama_cloud/llama_parse/>

PyMuPDF4LLM is based on [PyMuPDF](https://pymupdf.readthedocs.io/)

This documentation explains how to use the PyMuPDF4LLM package as well as providing links to other related RAG & LLM resources for PyMuPDF:

<https://pymupdf4llm.readthedocs.io/en/latest/>

## Features

* Support for multi-column pages
* Support for image and vector graphics extraction (and inclusion of references in the MD text)
* Support for page chunking output.
* Direct support for output as [LlamaIndex Documents](https://pymupdf4llm.readthedocs.io/en/latest/#extracting-as-llamaindex).

<https://docs.llamaindex.ai/en/stable/api_reference/node_parsers/markdown/>

<https://pypi.org/project/deepdiff/>

Why I stop experimenting with fixed-size chunking in my config:  
<https://safjan.com/from-fixed-size-to-nlp-chunking-a-deep-dive-into-text-chunking-techniques/>