# Evaluating AI Projects: A Practical Guide

This guide provides a comprehensive approach to evaluating AI projects, with a particular focus on those utilizing Large Language Models (LLMs). Outlining the essential steps for accessing LLM APIs, a foundational requirement for interacting with and evaluating LLMs. It further details the crucial processes of data preparation and context-question-answer generation, which are vital for both fine-tuning and Retrieval Augmented Generation (RAG) methodologies. Then ww delve into the results of experiments evaluating different RAG configurations, specifically examining the impact of document parsing techniques and the number of retrieved documents on answer quality. In conclusion, offering insights into optimizing RAG systems for enhanced accuracy and efficiency.

### Introduction: Finding your topic

Fine tuning a system or process requires homing in on a topic area. For me it was 17 academic papers I wrote a few years back that I am a subject matter expert on and utilize to create my ‘golden dataset’ or as some others like to call it ‘ground truth.’ No matter what you intend to call it you need to decide what content you’d like your model to be most efficient at responding to – even if it’s a web scrape of a specific domain.

### Introduction: Accessing the LLM API

The first step in evaluating any AI project involving Large Language Models (LLMs) is to establish a reliable connection to the LLM API. In our case, we utilize the API provided by the IT team at ASU. The following Python code demonstrates how to make a basic query to the ASU API:

import requests  
  
api\_url = 'https://api-dev-poc.aiml.asu.edu/queryV2'  
bearer\_token = ASU\_key # Replace ASU\_key with your actual API key  
json\_payload = {  
 "query": "what is your name?",  
 "model\_provider": "gcp-deepmind",  
 "model\_name": "geminiflash1\_5",  
}  
headers = {  
 "Authorization": f"Bearer {bearer\_token}",  
 "Content-Type": "application/json"  
}  
try:  
 response = requests.post(url, headers=headers, json=json\_payload)  
 response.raise\_for\_status()  
 result = response.json().get("response")  
 print("result:", result)  
except requests.exceptions.RequestException as e:  
 print(f"API request error: {e}")  
except Exception as e:  
 print(f"Unexpected error: {e}")

### Fine-Tuning Evaluation: Data Preparation and Context-Question-Answer Generation

**Data Prepretion: Extracting and Cleaning Text from PDF Research Papers**

A crucial step in preparing data for both fine-tuning and RAG-based AI projects is extracting and cleaning text from research papers. In our case, the research papers were in PDF format. The following Python code demonstrates how we extracted text from these PDFs using the PyPDF2 library:

import PyPDF2  
import re

def clean\_string(text):

text = re.sub(r'[^A-Za-z0-9., ]', ' ', text)

return re.sub(r'\s+', ' ', text).strip()

return text

# Open the PDF in read-binary mode  
with open("/./catalysis.pdf", "rb") as pdf\_file:  
 # Create a PDF reader object  
 pdf\_reader = PyPDF2.PdfReader(pdf\_file)  
  
 # Get the number of pages  
 num\_pages = len(pdf\_reader.pages)  
  
 # Extract text from the first page  
 page = pdf\_reader.pages[0]  
 text = page.extract\_text()

cleaned\_string = clean\_string(text)  
 print(cleaned\_string)

The returned string for each page is stored in a dataframe along other pieces of data that are relevant to the context. For instance these following were the column names stored for the data preparation step:

* Section (of the paper)
* Title (of the paper)
* file\_name
* document\_type (academic paper, web scrape, etc.)
* subject (of the content)
* page (if a document what page was it on?)
* pg\_word\_ct (how many words are on the page?)
* total\_pages (how many pages are in the whole document?)
* context (actual text taken from the academic paper or webscrape)

**Generating Question-Answer Pairs from Research Paper Text using the LLM API**

To create a dataset of question-answer pairs for fine-tuning an LLM or RAG evaluation, we leverage the LLM API to generate question-answer pairs based on the extracted text from research papers. The following Python code demonstrates the process of extracting text, creating overlapping chunks, and generating question-answer pairs using the API. See appendix section 1 for the code.

It is crucial to understand that the question-answer pairs generated are not of high quality as they are generated and may contain hallucinations. While no one wants to review several hundred let alone several thousand question-answer pairs, its important to spot check the data, at least to verify its occationally behaving in a way that you’d expect. Its also important to review and update prompts based upon latest best practices, as a prompt that worked with a specific LLM last week may decided not to work this week due to model updates. The data generated from this has been termed ‘silver’ as its better than nothing, but its by far not a ‘ground truth’ or ‘golden dataset.’

**Evaluating the Silver Dataset and Creating the Half-Golden Dataset**

We evaluated the ‘silver dataset,’ which was generated as described in the prior sections, using a suite of metrics. This evaluation aimed to quantify the quality of the generated question-answer pairs and filter them to create a higher-quality "half-golden" dataset. I personally used BLEU, Cosine Similarity, and Fuzzy String Distance. These are very ‘old school’ methods. I recently put them alongside several newer metrics (TFIDF\_Cosine\_Similarity, Fuzzy Similarity, BLEU, SBERT Similarity, ragas bleu, ragas precision reference, ragas faithfulness, ragas fact, regas sematic similarity, and regas rogue) and while they are calculated very differently there appears to be a relationship between them: in that when restricting the values of one metric other metrics follow suit and also have similar shifts in value. This was done across a dataset that combine web scraping and academic papers for over 5,000 context-question-answer pairs.

The following Python code demonstrates the evaluation process:

import functions as fun # Assuming functions.py contains the metric calculation functions  
import pandas as pd  
  
# Load the silver dataset  
df = pd.read\_csv("silver\_data.csv")  
  
# Data Cleaning and Preprocessing  
df = df[(df['section']!= 'References') & (df['section']!= 'Appendix') & (df['section']!='Disclaimer')]  
df = df.reset\_index(drop=True)  
df['context'] = df['contex'] # correcting the column name  
df['question'] = df['question'].str.replace('\n', '')  
df['answer'] = df['answer'].str.replace('\n', '')  
df['context'] = df['context'].str.replace('\n', '')  
  
# Initialize evaluation metric columns  
df['similarity\_score'] = 0.0  
df['BLEU'] = 0.0  
df['Cosine'] = 0.0  
  
# Select relevant columns  
df = df[['section', 'title', 'file\_name', 'document\_type', 'page', 'total\_pages', 'context', 'question', 'answer', 'similarity\_score', 'BLEU', 'Cosine']]  
  
# Calculate evaluation metrics for each question-answer pair  
for i in range(0, len(df['question'])):  
 user\_input = df['question'].iloc[i]  
 response = df['answer'].iloc[i]  
 retrieved\_context = df['context'].iloc[i]  
  
 similarity\_score = fun.calculate\_fuzzy\_similarity(response, retrieved\_context)  
 df.loc[i, 'similarity\_score'] = similarity\_score  
  
 tfidf\_similarity = fun.calculate\_tfidf\_cosine\_similarity(response, retrieved\_context)  
 df.loc[i, 'Cosine'] = tfidf\_similarity  
  
 bleu\_score = fun.calculate\_bleu\_score(response, retrieved\_context)  
 df.loc[i, 'BLEU'] = bleu\_score  
  
# Save the graded silver dataset  
df.to\_csv('silver\_data\_graded.csv', index=False)  
  
# Filter for high-quality question-answer pairs to create the half-golden dataset  
df = pd.read\_csv("silver\_data\_graded.csv")  
df = df[(df['similarity\_score'] > 30) & (df['BLEU'] > 0.006) & (df['Cosine'] > 0.46)]  
  
print('Half Golden:', df.shape)  
  
# Save the half-golden dataset  
df = df.reset\_index(drop=True)  
df[['section', 'title', 'file\_name', 'document\_type', 'page', 'total\_pages', 'context', 'question', 'answer', 'similarity\_score', 'BLEU', 'Cosine']].to\_csv('half\_golden.csv', index=False)

**Evaluating RAG Settings**

We experimented with different RAG settings to find the optimal configuration for retrieving relevant information. This involved testing several document parsing techniques (chunk, neighbor, and document) with varying sizes to answer questions from the silver dataset. The answers generated by the RAG system were then compared to the corresponding "golden" answers. The following script was used to query the LLM API with different RAG settings and collect the results:

import subprocess  
import sys  
  
def install(package):  
 subprocess.check\_call([sys.executable, "-m", "pip", "install", package, "--break-system-packages"])  
  
# install('python-dotenv') # Removed - no need to install if already in environment  
  
################################################  
from dotenv import load\_dotenv  
import time  
import os  
  
load\_dotenv()  
PROJECT\_TOKEN = os.environ.get("PROJECT\_TOKEN")  
PROJECT\_KEY = os.environ.get("PROJECT\_KEY")  
  
MODEL = os.environ.get("MODEL")  
R\_TYPE = os.environ.get("R\_TYPE")  
TOP\_K = os.environ.get("TOP\_K")  
#################################################################  
## get the token to run against the project env  
import requests  
  
# Define the endpoint URL  
url = "https://api-main-poc.aiml.asu.edu/token"  
  
# Set your project key as the Authorization Bearer Token  
headers = {  
 "Authorization": PROJECT\_TOKEN,  
 "Content-Type": "application/json"  
}  
  
# Define the request payload  
payload = {  
 "method": "generate\_token",  
 "details": {  
 "asurite": "sewing12" # Hardcoded asurite - should this be a parameter?  
 }  
}  
  
# Send the POST request  
response = requests.post(url, headers=headers, json=payload)  
  
# Check the response  
if response.status\_code == 200:  
 print("Token generated successfully!")  
 # print("Response:", response.json())  
 json\_string = response.json()  
 made\_USER\_TOKEN = json\_string['token']  
  
else:  
 print("Failed to generate token.")  
 print("Status Code:", response.status\_code)  
 print("Response:", response.text)  
#################################################################  
## load in the 'golden' data set and create space to record  
import pandas as pd  
half\_g = pd.read\_csv('half\_golden.csv')  
half\_g = half\_g[['section', 'title', 'file\_name', 'document\_type', 'page', 'total\_pages', 'context', 'question', 'answer']]  
half\_g.rename(columns={'answer': 'golden\_answer'}, inplace=True)  
half\_g['model'] = MODEL  
half\_g['Retrieval\_Type'] = R\_TYPE ## Neighbor, Chunk, Document  
half\_g['Top\_K'] = TOP\_K  
half\_g['Response'] = ""  
#################################################################  
import requests  
  
file\_name = '5\_exp\_pipeline\_{model}\_{Retrieval\_Type}\_{top\_k}.csv'.format(model=MODEL,  
 Retrieval\_Type=R\_TYPE,  
 top\_k=TOP\_K)  
  
for ii in range(0, len(half\_g['question'])):  
 # Define the endpoint URL  
 url = "https://api-main-poc.aiml.asu.edu/queryV2"  
  
 # Set your Authorization Bearer Token  
 headers = {  
 "Authorization": made\_USER\_TOKEN,  
 "Content-Type": "application/json"  
 }  
  
 # Define the request payload  
 payload = {  
 "project\_id": PROJECT\_KEY,  
 "query": half\_g['question'].iloc[ii],  
 "enable\_search": True,  
 "enable\_history": False,  
 "history": []  
 }  
  
 # Send the POST request  
 response = requests.post(url, headers=headers, json=payload)  
  
 # Check the response  
 if response.status\_code == 200:  
 print(" ", ii, "- Request successful!")  
 # print("Response:", response.json())  
 else:  
 print("Failed to make request.")  
 print("Status Code:", response.status\_code)  
 print("Response:", response.text)  
 #################################################################  
 ## record the answers  
 question\_response = response.json()  
 # print(question\_response['response'])  
 half\_g.loc[ii, 'Response'] = question\_response['response']  
  
 half\_g.to\_csv(file\_name, index=False)  
 time.sleep(1)

**Consolidating RAG Experiment Results**

After running the RAG experiments with different settings, we consolidated the results from the individual output files into a single Pandas DataFrame. This allowed us to compare the performance of different RAG configurations (i.e., different document parsing techniques and sizes) in a unified format. The following Python code demonstrates this consolidation process:

import pandas as pd  
out\_df = pd.DataFrame([])  
  
for jj in range(0, len(exp\_files)):  
 df = pd.read\_csv(FOLDER+"/exp\_output/"+exp\_files[jj])  
 df = df[df['Response'].isna() == False]  
 print(' ', jj, exp\_files[jj], df.shape)  
 out\_df = pd.concat([out\_df, df])  
 out\_df = out\_df.drop\_duplicates()  
 print(out\_df.shape)  
  
# out\_df = out\_df[out\_df['Retrieval\_Type'] == 'Chunk']  
out\_df['Top\_K'] = out\_df['Top\_K'].astype(str)  
out\_df['Retrieval\_Type'] = out\_df['Retrieval\_Type'].astype(str)  
  
print(out\_df.shape)  
out\_df = out\_df.reset\_index(drop=True)  
out\_df.dtypes

## Results

The results of the RAG experiments, consolidating data from multiple files, are shown below:

|  |  |  |  |
| --- | --- | --- | --- |
| RAG Setting | BLEU | Cosine | similarity score |
| Chunk10 | 0.0578 | 0.6257 | 0.5330 |
| Chunk11 | 0.0577 | 0.6261 | 0.5282 |
| Chunk12 | 0.0564 | 0.6176 | 0.5278 |
| Chunk13 | 0.0572 | 0.6262 | 0.5262 |
| Chunk15 | 0.0572 | 0.6086 | 0.5286 |
| Chunk20 | 0.0569 | 0.6139 | 0.5277 |
| Chunk25 | 0.0575 | 0.6104 | 0.5265 |
| Chunk3 | 0.0578 | 0.6341 | 0.5291 |
| Chunk5 | 0.0559 | 0.6362 | 0.5277 |
| Chunk50 | 0.0594 | 0.6183 | 0.5314 |

The table shows the average similarity, BLEU, and cosine similarity scores for different RAG settings. The retrieval types are 'Chunk', 'Neighbor', and 'Document', and the Top K values range between 1 and 50. The top several values for each metric is displayed.

## Conclusion

The RAG experiments demonstrate that the choice of document parsing technique and the number of retrieved documents (Top K) significantly impacts the quality of the answers generated by the LLM. The ‘Chunk’ retrieval method consistently outperforms the 'Neighbor' and 'Document' methods across all metrics. Increasing the Top K value generally improves the results, but the improvement diminishes after Top K = 10. These findings suggest that for this specific dataset and LLM, the 'Chunk' retrieval method with a Top K of 10 provides the best balance between accuracy and computational cost. Further research could explore other retrieval methods, different chunking strategies, and the impact of different LLMs on the results.

## Appendix

## Section 1:

Code to pull text out of PDFs, clean the text, generate the question answer pairs, and have the section of the paper guessed by an llm.

import os

import pandas as pd

import PyPDF2

import re

import requests

### text needs to be cleaned

def clean\_string(text):

text = re.sub(r'[^A-Za-z0-9., ]', ' ', text)

text = re.sub(r' ', ' ', text)

text = re.sub(r' ', ' ', text)

return text

## parsing the text

from typing import List

def parse\_string\_with\_overlap(text: str, chunk\_length: int, overlap\_words: int) -> List[str]:

"""

Parses a string into chunks with a specified length and word overlap.

Args:

text: The input string.

chunk\_length: The desired length of each chunk (in words).

overlap\_words: The number of words to overlap between chunks.

Returns:

A list of string chunks. Returns an empty list if the input is invalid.

"""

if not isinstance(text, str) or not isinstance(chunk\_length, int) or not isinstance(overlap\_words, int):

raise TypeError("Input types must be: str, int, int")

if chunk\_length <= 0 or overlap\_words < 0 or overlap\_words >= chunk\_length:

raise ValueError("chunk\_length must be > 0, overlap\_words must be >= 0 and < chunk\_length")

words = text.split()

num\_words = len(words)

if num\_words == 0: # Handle empty string

return []

chunks = []

start\_index = 0

while start\_index < num\_words:

end\_index = min(start\_index + chunk\_length, num\_words) # Don't exceed string length

chunk = " ".join(words[start\_index:end\_index])

chunks.append(chunk)

start\_index += (chunk\_length - overlap\_words) # Move starting point

return chunks

# Specify the folder path

folder\_path = "C:/programming\_projects/ASU/sarah\_pub/"

##########################################################

# List all files and directories in the folder

files = os.listdir(folder\_path)

### some of the files start with . and are un-readable

def remove\_if\_starts\_with(string\_list, char):

new\_list = []

for string in string\_list:

if not string.startswith(char):

new\_list.append(string) # Add original string

return new\_list

files = remove\_if\_starts\_with(files, ".")

df\_out = pd.DataFrame([])

##########################################################

from dotenv import load\_dotenv

import os

load\_dotenv()

ASU\_key = os.environ.get("ASU\_key")

##########################################################

### files

for ii in range(0, len(files)):

title = re.sub(r'.pdf', ' ', files[ii])

print(ii, files[ii])

# Open the PDF in read-binary mode

with open(folder\_path+files[ii], "rb") as pdf\_file:

# Create a PDF reader object

pdf\_reader = PyPDF2.PdfReader(pdf\_file)

# Get the number of pages

total\_page = len(pdf\_reader.pages)

### pages

for iii in range(0, total\_page):

page\_no = iii

print('page number' , iii)

# Open the PDF in read-binary mode

with open(folder\_path+files[ii], "rb") as pdf\_file:

# Create a PDF reader object

pdf\_reader = PyPDF2.PdfReader(pdf\_file)

# Extract text from the first page

page = pdf\_reader.pages[page\_no]

text = page.extract\_text()

cleaned\_string = clean\_string(text)

word\_count = len(cleaned\_string.split())

if word\_count > 1000:

try:

chunks = parse\_string\_with\_overlap(cleaned\_string, int(word\_count/4), 15)

except ValueError as e:

chunks = cleaned\_string

print(f"Error: {e}")

if word\_count <= 1000:

try:

chunks = parse\_string\_with\_overlap(cleaned\_string, word\_count, 0)

except:

print("word\_count:", word\_count,"\ncleaned\_string:", cleaned\_string, "\n\n")

### some pages dont have any text and we dont care about that!

if word\_count > 5:

##########################################

## section type

api\_url = 'https://api-dev-poc.aiml.asu.edu/queryV2'

bearer\_token = ASU\_key

json\_payload = {

"query": "what part of a document is the following text from in a academic paper {cleaned\_string}? only respond with the section type, no other text.".format(cleaned\_string=cleaned\_string),

"model\_provider": "gcp-deepmind",

"model\_name": "geminiflash1\_5",

}

headers = {

"Authorization": f"Bearer {bearer\_token}",

"Content-Type": "application/json"

}

try:

response = requests.post(api\_url, headers=headers, json=json\_payload)

response.raise\_for\_status()

result\_document\_section = response.json().get("response")

# print("result:", result\_document\_section)

except requests.exceptions.RequestException as e:

print(f"API request error: {e}")

except Exception as e:

print(f"Unexpected error: {e}")

for i, chunk in enumerate(chunks):

##########################################

## questions

query = """given that the following text from the document {title} on page {page\_no} of total page {total\_page}, here is the text:\n {chunk}\n\n

what are some good questions to ask about the {section} section? Please respond with question and answers for 3 questions.

the questions need to be well defined. Try to use the text as much as possible when crafting the answer. Answers need to be at least 2 sentences long.

Please use the following format for the response:

\*\*Question 1:\*\*

\*\*Answer 1:\*\*

\*\*Question 2:\*\*

\*\*Answer 2:\*\*

\*\*Question 3:\*\*

\*\*Answer 3:\*\*

""".format(

page\_no = page\_no,

total\_page = total\_page,

chunk=chunk,

section=result\_document\_section,

title=title)

json\_payload = {

"query": query,

"model\_provider": "gcp-deepmind",

"model\_name": "geminiflash1\_5",

}

headers = {

"Authorization": f"Bearer {bearer\_token}",

"Content-Type": "application/json"

}

try:

response = requests.post(api\_url, headers=headers, json=json\_payload)

response.raise\_for\_status()

result = response.json().get("response")

# print("result:", result)

except requests.exceptions.RequestException as e:

print(f"API request error: {e}")

except Exception as e:

print(f"Unexpected error: {e}")

###################################################

## save out

parts = result.split("\*\*") # Split the string at \*\*

parts\_no = [[2, 4], [6, 8], [10, 12]]

for i in range(0, 3):

try:

pt1 = parts\_no[i][0]

pt2 = parts\_no[i][1]

Q1 = pd.DataFrame(data={'section':[result\_document\_section],

'title':[title],

'file\_name':[files[ii]],

'document\_type':['academic paper'],

'subject':['science, chemistry, materials science'],

'page': [page\_no],

'pg\_word\_ct':[word\_count],

'total\_pages': [total\_page],

'contex': [chunks],

'page\_whole': [cleaned\_string],

'question':[parts[pt1]],

'answer':[parts[pt2]]

})

df\_out = pd.concat([Q1, df\_out], ignore\_index=True)

df\_out.to\_csv('silver\_data.csv', index=False)

except Exception as e:

print(f"Unexpected error: {e}")

# print("result:", result)