

Day 2: Chunking and Intelligent Processing for Data

Welcome to Day 2 of our [7-Day AI Agents Email Crash-Course](#).

In the first part of the course, we focus on **data preparation** – the process of properly preparing data before it can be used for AI agents.

Small and Large Documents

Yesterday (Day 1), we downloaded the data from a GitHub repository and processed it. For some use cases, like the FAQ database, this is sufficient. The questions and answers are small enough. We can put them directly into the search engine.

But it's different for the Evidently documentation. These documents are quite large. Let's take a look at this one:

<https://github.com/evidentlyai/docs/blob/main/docs/library/descriptors.mdx>.

We could use it as is, but we risk overwhelming our LLMs.

Why We Need to Prepare Large Documents Before Using Them

Large documents create several problems:

- Token limits: Most LLMs have maximum input token limits
- Cost: Longer prompts cost more money
- Performance: LLMs perform worse with very long contexts
- Relevance: Not all parts of a long document are relevant to a specific question

So we need to split documents into smaller subdocuments. For AI applications like RAG (which we will discuss tomorrow), this process is referred to as "chunking."

Today's Tasks (Day 2)

Today, we will cover multiple ways of chunking data:

- Simple character-based chunking
- Paragraph and section-based chunking
- Intelligent chunking with LLM

Just so you know, for the last section, you will need an OpenAI account or an account from an alternative LLM provider such as Groq.

1. Simple Chunking

Let's start with simple chunking. This will be sufficient for most cases.

We can continue with the notebook from Day 1. We already downloaded the data from Evidently docs. We put them into the `evidently_docs` list.

This is how the document at index `45` looks like:

Python

```
{'title': 'LLM regression testing',  
 'description': 'How to run regression testing for LLM outputs.',  
 'content': 'In this tutorial, you will learn...'  
}
```

The `content` field is 21,712 characters long. The simplest thing we can do is cut it into pieces of equal length. For example, for size of 2000 characters, we will have:

- Chunk 1: 0..2000
- Chunk 2: 2000..4000
- Chunk 3: 4000..6000

And so on.

However, this approach has disadvantages:

- Context loss: Important information might be split in the middle
- Incomplete sentences: Chunks might end mid-sentence
- Missing connections: Related information might end up in different chunks

That's why, in practice, we usually make sure there's overlap between chunks. For size 2000 and overlap 1000, we will have:

- Chunk 1: 0..2000
- Chunk 2: 1000..3000
- Chunk 3: 2000..4000
- ...

This is better for AI because:

- Continuity: Important information isn't lost at chunk boundaries
- Context preservation: Related sentences stay together in at least one chunk
- Better search: Queries can match information even if it spans chunk boundaries

This approach is known as the "sliding window" method. This is how we implement it in Python:

```
Python
def sliding_window(seq, size, step):
    if size <= 0 or step <= 0:
        raise ValueError("size and step must be positive")

    n = len(seq)
    result = []
    for i in range(0, n, step):
        chunk = seq[i:i+size]
        result.append({'start': i, 'chunk': chunk})
        if i + size >= n:
            break

    return result
```

Let's apply it for document 45. This gives us 21 chunks:

- 0..2000
- 1000..3000
- ...
- 19000..21000
- 20000..21712

Let's process all the documents:

```
Python
evidently_chunks = []

for doc in evidently_docs:
    doc_copy = doc.copy()
    doc_content = doc_copy.pop('content')
    chunks = sliding_window(doc_content, 2000, 1000)
    for chunk in chunks:
        chunk.update(doc_copy)
    evidently_chunks.extend(chunks)
```

Note that we use `copy()` and `pop()` operations:

- `doc.copy()` creates a shallow copy of the document dictionary
- `doc_copy.pop('content')` removes the 'content' key and returns its value
- This way we preserve the original dictionary keys that we can use later in the chunks.

This way, we obtain 575 chunks from 95 documents.

We can play with the parameters by including more or less content. 2000 characters is usually good enough for RAG applications.

There are some alternative approaches:

- Token-based chunking: You first tokenize the content (turn it into a sequence of words) and then do a sliding window over tokens
 - Advantages: More precise control over LLM input size
 - Disadvantages: Doesn't work well for documents with code
- Paragraph splitting: Split by paragraphs
- Section splitting: Split by sections
- AI-powered splitting: Let AI split the text intelligently

We won't cover token-based chunking here, as we're working with documents that contain code. But it's easy to implement - ask ChatGPT for help if you need it for text-only content.

We will implement the others.

2. Splitting by Paragraphs and Sections

Splitting by paragraphs is relatively easy:

```
Python
import re
text = evidently_docs[45]['content']
paragraphs = re.split(r"\n\s*\n", text.strip())
```

We use `\n\s*\n` regex pattern for splitting:

- `\n` matches a newline
- `\s*` matches zero or more whitespace characters
- `\n` matches another newline
- So `\n\s*\n` matches two newlines with optional whitespace between them

This works well for literature, but it doesn't work well for documents. Most paragraphs in technical documentation are very short.

You can combine sliding window and paragraph splitting for more intelligent processing. We won't do it here, but it's a good exercise to try.

Let's now look at section splitting. Here, we take advantage of the documents' structure. Markdown documents have this structure:

```
None
# Heading 1
## Heading 2
### Heading 3
```

What we can do is split by headers.

For that we will use regex too:

```

Python
import re

def split_markdown_by_level(text, level=2):
    """
    Split markdown text by a specific header level.

    :param text: Markdown text as a string
    :param level: Header level to split on
    :return: List of sections as strings
    """
    # This regex matches markdown headers
    # For level 2, it matches lines starting with "## "
    header_pattern = r'^#{' + str(level) + r'} (.+)$'
    pattern = re.compile(header_pattern, re.MULTILINE)

    # Split and keep the headers
    parts = pattern.split(text)

    sections = []
    for i in range(1, len(parts), 3):
        # We step by 3 because regex.split() with
        # capturing groups returns:
        # [before_match, group1, group2, after_match, ...]
        # here group1 is "## ", group2 is the header text
        header = parts[i] + parts[i+1] # "## " + "Title"
        header = header.strip()

        # Get the content after this header
        content = ""
        if i+2 < len(parts):
            content = parts[i+2].strip()

        if content:
            section = f'{header}\n\n{content}'
        else:
            section = header
        sections.append(section)

    return sections

```

Note: This code may not work perfectly if we want to split by level 1 headings and have Python code with `#` comments. But in general, this is not a big problem for documentation.

If we want to split by second-level headers, that's what we do:

Python

```
sections = split_markdown_by_level(text, level=2)
```

Now we iterate over all the docs to create the final result:

Python

```
evidently_chunks = []

for doc in evidently_docs:
    doc_copy = doc.copy()
    doc_content = doc_copy.pop('content')
    sections = split_markdown_by_level(doc_content, level=2)
    for section in sections:
        section_doc = doc_copy.copy()
        section_doc['section'] = section
        evidently_chunks.append(section_doc)
```

Like previously, `copy()` creates a copy of the document metadata. `pop('content')` removes and returns the content. This way, each section gets the same metadata (title, description) as the original document.

This was more intelligent processing, but we can go even further and use LLMs for that.

3. Intelligent Chunking with LLM

In some cases, we want to be more intelligent with chunking. Instead of doing simple splits, we delegate this work to AI.

This makes sense when:

- Complex structure: Documents have complex, non-standard structure
- Semantic coherence: You want chunks that are semantically meaningful
- Custom logic: You need domain-specific splitting rules
- Quality over cost: You prioritize quality over processing cost

This costs money. In most cases, we don't need intelligent chunking.

Simple approaches are sufficient. Use intelligent chunking only when

- You already evaluated simpler methods and you can confirm that they produce poor results
- You have complex, unstructured documents
- Quality is more important than cost
- You have the budget for LLM processing

Note: You can use any alternative LLM provider. One option is [Groq](#), which is free with rate limits. You can replace the OpenAI library with the Groq library and it should work.

To continue, you need to get the API key from <https://platform.openai.com/api-keys> (assuming you have an account).

Let's stop Jupyter and create an environment variable with your key:

```
Shell
export OPENAI_API_KEY='your-api-key'
```

Install the OpenAI SDK:

```
Shell
uv add openai
```

Then run jupyter notebook:

```
Shell
uv run jupyter notebook
```

It's cumbersome to set environment variables every time. I recommend using [direnv](#), which works for Linux, Mac and Windows.

Note: if you use direnv, don't forget to add `.envrc` to `.gitignore`.

Warning: Never commit your API keys to git! Others can use your API key and you'll pay for it.

Now we're ready to use OpenAI:

```
Python
from openai import OpenAI

openai_client = OpenAI()

def llm(prompt, model='gpt-4o-mini'):
    messages = [
        {"role": "user", "content": prompt}
    ]

    response = openai_client.responses.create(
        model='gpt-4o-mini',
        input=messages
    )

    return response.output_text
```

This code invokes an LLM (gpt-4o-mini) with the provided prompt and returns the results. We will explain in more detail what this code does in the next lessons.

Let's create a prompt:

```
Python
prompt_template = """
Split the provided document into logical sections
that make sense for a Q&A system.

Each section should be self-contained and cover
a specific topic or concept.

<DOCUMENT>
{document}
</DOCUMENT>

Use this format:

## Section Name
```

```
Section content with all relevant details
```

```
---
```

```
## Another Section Name
```

```
Another section content
```

```
---
```

```
"".strip()
```

The prompt asks the LLM to:

- Split the document logically (not just by length)
- Make sections self-contained
- Use a specific output format that's easy to parse

Let's create a function for intelligent chunking:

Python

```
def intelligent_chunking(text):  
    prompt = prompt_template.format(document=text)  
    response = llm(prompt)  
    sections = response.split('---')  
    sections = [s.strip() for s in sections if s.strip()]  
    return sections
```

Now we apply this to every document:

Python

```
from tqdm.auto import tqdm  
  
evidently_chunks = []  
  
for doc in tqdm(evidently_docs):  
    doc_copy = doc.copy()  
    doc_content = doc_copy.pop('content')
```

```
sections = intelligent_chunking(doc_content)
for section in sections:
    section_doc = doc_copy.copy()
    section_doc['section'] = section
    evidently_chunks.append(section_doc)
```

`tqdm` is a library that shows progress bars. It helps you track progress when processing a large number of documents.

Note: This process requires time and incurs costs. As mentioned before, use this only when really necessary. For most applications, you don't need intelligent chunking.

Bonus: you can use this approach for processing the code in your GitHub repository. You can use a variation of the following prompt:

"Summarize the code in plain English. Briefly describe each class and function/method (their purpose and role), then give a short overall summary of how they work together. Avoid low-level details.". Then add both the source code and the summary to your documents.

4. How to Choose a Chunking Approach

You may wonder - which chunking should I use? The answer: start with the simplest one and gradually increase complexity. Start with simple chunking with overlaps. We will later talk about evaluations. You can use evaluations to make informed decisions about chunking strategies.

Coming Up Tomorrow (Day 3)

Our data is ready. Now we can index it – insert it into a search engine and make it available for our (future) agent to use.


If you have suggestions about the course content or want to improve something, let me know! Answer to the email with this lesson.

Homework

- ☐ For the project you selected, apply chunking
- ☐ Experiment with simple chunking, paragraph chunking + sliding window, and section chunking
- ☐ Which approach makes sense for your application? Manually inspect the results and analyze what works best
- ☐ Make a post on social media about what you're building

Learning in Public

Example post for LinkedIn

 Day 2 of building my AI agent: Mastering document chunking!

Today I learned how to break down large documents into smaller, manageable pieces for my AI system.


I tried three approaches:

- • Simple sliding window chunking
- • Section-based splitting
- • AI-powered intelligent chunking

The key insight? Start simple! Most cases don't need complex chunking strategies.

My project focuses on [YOUR PROJECT DESCRIPTION]. Section-based chunking works perfectly because [YOUR REASONING].

Here's my repo: [YOUR_REPO_LINK]

Next up: Building the actual conversational agent! 




Following along with this amazing course - who else is building AI agents?

You can sign up here: <https://alexeygrigorev.com/aihero/>


Example post for Twitter/X:

Day 2: Document chunking for my AI agent 

3 chunking methods tested:

-  Simple sliding window
-  Section-based splitting
-  AI-powered chunking

Here's my repo: [YOUR_REPO_LINK]

Tomorrow: Building the conversational agent 

Join me: <https://alexeygrigorev.com/aihero/>

The Community

Have questions about this lesson or suggestions for improvement? You can find me and other learners in **DataTalks.Club Slack**:

- [Join DataTalks.Club](#)
- Find us in the [#course-ai-bootcamp channel](#)

In the community channel, you can:

- Ask questions about the course content
- Share your implementation and get feedback
- Show off your GitHub repositories
- Suggest improvements to the course materials
- Connect with other course participants

Don't hesitate to reach out - the community is here to help each other succeed!

