Getting started with CoAuthor

Goal: Download and read the CoAuthor dataset

Steps

1. Download CoAuthor
2. Read writing sessions
3. Examine events

1. Download CoAuthor

In [1]:
```
!wget https://cs.stanford.edu/~minalee/zip/chi2022-coauthor-v1.0.zip
!unzip -q chi2022-coauthor-v1.0.zip
!rm chi2022-coauthor-v1.0.zip
```

```
--2025-02-25 21:08:20--  https://cs.stanford.edu/~minalee/zip/chi2022-coauthor
-v1.0.zip
Resolving cs.stanford.edu (cs.stanford.edu)... 171.64.64.64
Connecting to cs.stanford.edu (cs.stanford.edu)|171.64.64.64|:443... connecte
d.
HTTP request sent, awaiting response... 200 OK
Length: 49956179 (48M) [application/zip]
Saving to: ‘chi2022-coauthor-v1.0.zip’

chi2022-coauthor-v1 100%[===================>]  47.64M  49.6MB/s    in 1.0s

2025-02-25 21:08:21 (49.6 MB/s) - ‘chi2022-coauthor-v1.0.zip’ saved [49956179/
49956179]
```

In [2]:
```
import os

dataset_dir = './coauthor-v1.0'
paths = [
    os.path.join(dataset_dir, path)
    for path in os.listdir(dataset_dir)
    if path.endswith('jsonl')
]

print(f'Successfully downloaded {len(paths)} writing sessions in CoAuthor!')
```

```
Successfully downloaded 1447 writing sessions in CoAuthor!
```

2. Read writing sessions

Each writing session is saved as a `.jsonl` file in CoAuthor. It is a very convenient format to store events occured in the writing session: each line is a JSON object, representing an *event*, and there are many lines in the file, representing *a sequence of events*.

Let's read one of the files and see what it looks like!

> `.jsonl` is the JSON Lines text format. It is convenient for storing structured data that may be processed one record at a time. For more information, please refer to https://jsonlines.org/.

In [3]:
```
import json

def read_writing_session(path):
    events = []
    with open(path, 'r') as f:
        for event in f:
            events.append(json.loads(event))
    print(f'Successfully read {len(events)} events in a writing session from {
    return events
```

In [4]:
```
events = read_writing_session(paths[2])
```

```
Successfully read 1657 events in a writing session from ./coauthor-v1.0/ba74a6
b6e52d42318d8ab8ab67651632.jsonl
```

3. Examine events

Whenever writers insert or delete text, move a cursor forward or backward, get suggestions from the system by pressing the tab key, or accept or dismiss suggestions, it is recorded as an event.

> Here is a list of all possible events in CoAuthor:
>
> - `system-initialize`
> - `text-insert`
> - `text-delete`
> - `cursor-forward`
> - `cursor-backward`
> - `cursor-select`
> - `suggestion-get`
> - `suggestion-open`
> - `suggestion-up`
> - `suggestion-down`
> - `suggestion-select`
> - `suggestion-close`

For more details, please check out our paper (Table 1): https://arxiv.org/pdf/2201.06796.pdf

In [5]:
```
event_names = [event['eventName'] for event in events]
event_names[:15]
```

Out[5]:
```
['system-initialize',
 'cursor-backward',
 'cursor-forward',
 'text-insert',
 'suggestion-get',
 'suggestion-open',
 'suggestion-hover',
 'suggestion-hover',
 'suggestion-hover',
 'suggestion-select',
 'suggestion-close',
 'text-insert',
 'text-insert',
 'text-insert',
 'text-insert']
```

Let's look at each *event* more closely now!

In the beginning of a writing session, you will see something like this for a system-initialize event:

In [6]:
```
events[106]
```

Out[6]:
```
{'eventName': 'text-insert',
 'eventSource': 'user',
 'eventTimestamp': 1630054490150,
 'textDelta': {'ops': [{'retain': 499}, {'insert': 't'}]},
 'cursorRange': '',
 'currentDoc': '',
 'currentCursor': 500,
 'currentSuggestions': [],
 'currentSuggestionIndex': 2,
 'currentHoverIndex': 2,
 'currentN': '5',
 'currentMaxToken': '30',
 'currentTemperature': '0.9',
 'currentTopP': '1',
 'currentPresencePenalty': '0',
 'currentFrequencyPenalty': '0.5',
 'eventNum': 106}
```

Concretely, an *event* is a tuple of event name, timestamp, and snapshot of the current editor. This is designed to preserve every detail about interactions at a keystroke-level, so it is quite detailed as you can see!

> **Event and its metadata**
>
> - `eventName` : event name (e.g. `system-initialize`)
> - `eventSource` : event source (e.g. `user` or `api`)
> - `textDelta` : text that has been changed compared to the previous event (if no change, empty)
> - `cursorRange` : cursor location or selection that has been changed compared to the previous event (if no change, empty)
>
> **Timestamp**

- `eventTimestamp` : timestamp of the event
- `eventNum` : index of the event

Snapshot of the current editor

- Editor
  - `currentDoc` : a writing prompt to start with (otherwise, empty)
  - `currentCursor` : cursor location
  - `currentSuggestions` : most recent suggestions that are stored and can be reopened
- Decoding parameters
  - `currentN` : the number of suggestions to generate per query (e.g. 5)
  - `currentMaxToken` : the maximum number of tokens to generate per suggestion
  - `currentTemperature` : sampling temperature to use for generation; higher values means the model will take more risks
  - `currentTopP` : nucleus sampling; the model considers the results of the tokens with top_p probability mass
  - `currentPresencePenalty` : positive values penalize new tokens based on whether they appear in the text so far, increasing the model's likelihood to talk about new topics
  - `currentFrequencyPenalty` : positive values penalize new tokens based on their existing frequency in the text so far, decreasing the model's likelihood to repeat the same line verbatim

For more details on decoding parameters, please refer to https://beta.openai.com/docs/api-reference/completions.

Helper Functions

Reconstruct the current document.

In [7]:
```
def reconstruct_current_doc(events):
    current_doc = events[0]['currentDoc']  # Initialize with the prompt
    i = 1

    for event in events[1:]:  # Skip the first event (system-initialize)
        if event['eventName'] == 'text-insert' or event['eventName'] == 'text-
            cursor_position = event['textDelta'].get('ops')[0].get('retain', '

            if any('delete' in d for d in event['textDelta'].get('ops')):
                nums = [d.get('delete') for d in event['textDelta'].get('ops')
                num_to_delete = 0
                for i in nums:
                    num_to_delete = num_to_delete + i
                current_doc = current_doc[:cursor_position] + current_doc[curs

            if any('insert' in d for d in event['textDelta'].get('ops')):
```

```
                text_to_insert = [d.get('insert') for d in event['textDelta'].
                retain_lst = [d.get('retain') for d in event['textDelta'].get(
                if len(retain_lst) > 1:
                    position_adj = 1
                else:
                    position_adj = 0
                current_doc = current_doc[:cursor_position+position_adj] + tex
    return current_doc
```

In [8]:
```
# Reconstruct the currentDoc
current_doc = reconstruct_current_doc(events)

# Print the final document
print("Final Document:")
print(current_doc)
```

```
Final Document:
What Are the Most Important Things Students Should Learn in School?

In your opinion, what are the most important things students should learn in s
chool? What is the most important thing you have learned in school? How has th
is knowledge affected your life? How do you think it will help your success in
the future?

---

In my opinion, the most important things students should learn in school are h
ow to get along with others, and how to be successful on the job. Few people g
o on to pursue a job that requires a highly specific degree. Most people just
finish high school and get a job. Because of that they really need to learn so
cial skills first. School forces you to be around people you may not like. You
are put into classes with people you may not know. The more you learn how to g
et along with others the better off you will be. That skill comes in handy whe
n it's time to work. You again will be surrounded by people you may not like a
nd you'll need social skills to navigate the situation.

Second there's the job market. School will show you what things you are intere
sted in and good at. You may learn that you are good at math so you feel comfo
rtable getting an office job where you do paperwork. On the other hand you may
learn that you are better at working with people and serving them or being dir
ectly around them, such as in a doctor's office. These things you learn in sch
ool show you what you'll be good at.

Obviously students need to learn the basic skills to get through life like lea
rning English and basic math skills. From there though it really comes down to
just finding out your social skills and the things you enjoy doing. School hel
ps you with those things, it really opens your eyes to many choices so that yo
u can choose the career that fits you and the one you will want to stick with.
Since most of us will not get a college degree it is important to realisticall
y view school for what it is. It's a chance to just learn about ourselves and
use that to guide us through our work life as an adult.
```

Extract the last three sentences from the document.

In [9]:
```
import re

def get_last_sentences(paragraph):
    """
```

```
    Extracts the last sentence from a paragraph.

    Args:
        paragraph: The input paragraph as a string.

    Returns:
        The last sentence of the paragraph, or None if no sentence is found.
    """
    if not paragraph:
        return None

    sentences = re.split(r'(?<!\w\.\w)(?<![A-Z][a-z]\.)(?<=\.|\?|!)\s', paragra

    sentences = [s.strip() for s in sentences if s.strip()]
    last_three = sentences[-3:]
    return " ".join(last_three)

# Example usage
text = "This is the first sentence. Here is the second sentence.\n\nAnd this is
last_sentence = get_last_sentences(text)
print(last_sentence)

text_2 = "This is a paragraph without multiple sentences."
last_sentence_2 = get_last_sentences(text_2)
print(last_sentence_2)

text_3 = ""
last_sentence_3 = get_last_sentences(text_3)
print(last_sentence_3)
```

```
Here is the second sentence. And this is the last sentence! But what about thi
s one?
This is a paragraph without multiple sentences.
None
```

Compute Coherence Score

Use a pre-trained language model like BERT to generate embeddings for the AI suggestion and the context.

In [10]:
```
from sentence_transformers import SentenceTransformer, util

def compute_coherence(suggestion, context):
    model = SentenceTransformer('all-MiniLM-L6-v2')

    embeddings = model.encode([suggestion, context])

    similarity = util.cos_sim(embeddings[0], embeddings[1])[0][0]

    return similarity
```

In [ ]:
```
coherence_scores = []
for session_path in paths[:10]:
    events = read_writing_session(session_path)
    event_num = 0

    for event in events:
        if event['eventName'] == 'suggestion-open':
```

```
            if event['currentSuggestions']:  # Ensure there is at least one su
                context = get_last_sentences(reconstruct_current_doc(events[:e

                for suggestion in event['currentSuggestions']:
                    suggestion = suggestion.get('original')
                    coherence_score = compute_coherence(suggestion, context)
                    coherence_dict = {"suggestion": suggestion, "context": con
                    coherence_scores.append(coherence_dict)
                    # print(suggestion)
                    # print(context)
                    # print(coherence_score)
                    # print("-----------------------------------------------
            else:
                print(f"Skipping event with empty suggestions: {event}")
        event_num = event_num + 1
```

```
Successfully read 2301 events in a writing session from ./coauthor-v1.0/87b45d
27c2424504a98f49a29509480f.jsonl
```

```
/usr/local/lib/python3.11/dist-packages/huggingface_hub/utils/_auth.py:94: Use
rWarning:
The secret `HF_TOKEN` does not exist in your Colab secrets.
To authenticate with the Hugging Face Hub, create a token in your settings tab
(https://huggingface.co/settings/tokens), set it as secret in your Google Cola
b and restart your session.
You will be able to reuse this secret in all of your notebooks.
Please note that authentication is recommended but still optional to access pu
blic models or datasets.
  warnings.warn(
```

Calculate the average coherence score.

In [ ]:
```
sum = 0
for d in coherence_scores:
    sum += d['score']
average_coherence = sum / len(coherence_scores)

print([d['score'] for d in coherence_scores])
# print(coherence_scores)
print(f"Average coherence score: {average_coherence}")
```

In [ ]:
```
# Example usage
suggestion = "A dog stood on the table"
context = "The cat sat on the mat"
coherence_score = compute_coherence(suggestion, context)
print(f"Coherence score: {coherence_score}")
```