NLP Pipeline for Sherlock Holmes Dataset

Deep Learning 2024-2025

Nikoletta Teazi 30137927

Introduction

Natural Language Processing (NLP) plays a crucial role in enabling machines to understand and process human language, particularly in complex literary texts. This project focuses on designing and implementing an NLP pipeline to analyze Sherlock Holmes stories, addressing the challenges of text preprocessing, sequence-to-sequence summarization, semantic search, and topic modeling using LDA.

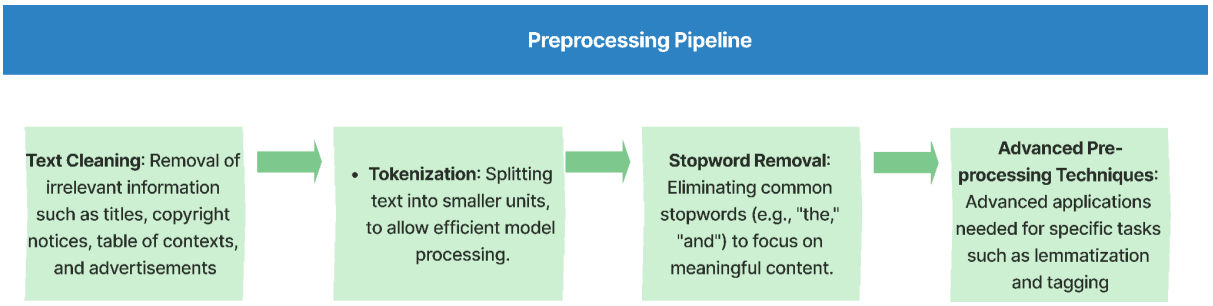
Unlike structured text, literary works often contain idiomatic expressions, intricate linguistic structures, and additional non-essential elements such as titles, summaries, and copyright notices. These elements make automated text processing more challenging, as they introduce variability that models must learn to handle effectively.

This study explores how preprocessing techniques along with further model tuning impacts NLP tasks.

Methodology

Data Preprocessing (Task 1)

The first task of this assignment was data preprocessing, addressing common challenges in literary text. Given its complexity, preprocessing is crucial for ensuring the effectiveness of deep learning models. As the saying goes, “garbage in, garbage out”, the quality of preprocessing directly impacts the performance of all the following tasks.



The preprocessing steps were implemented as functions with configurable parameters, allowing flexibility based on task requirements.

```
processed_stories = process_stories(stories, keep_special_chars=False)
processed_stories_wspecialchar = process_stories(stories, keep_special_chars=True)
```

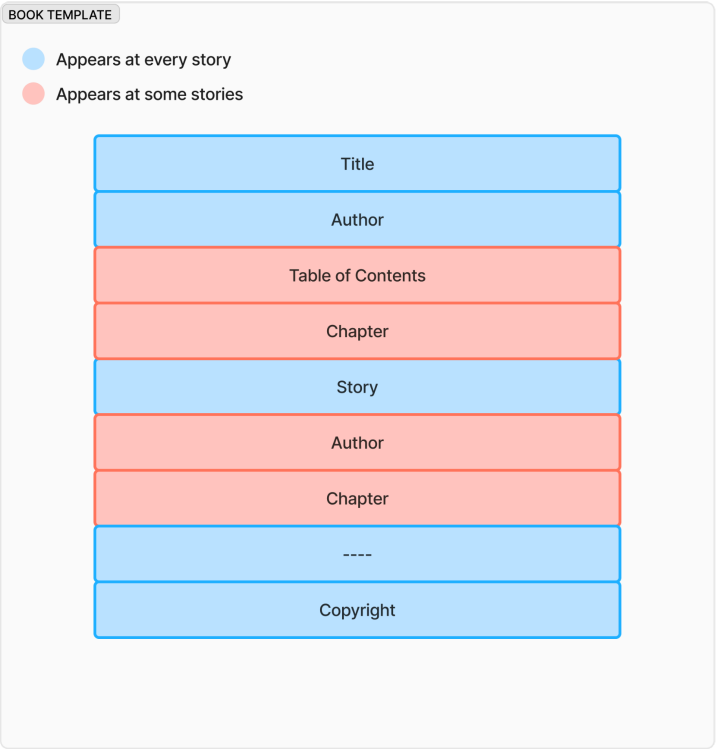
For example, the process_stories function cleans punctuation, titles, and copyright text while offering an option to retain or remove special characters, ensuring adaptability for different tasks.

Preprocessing Steps

Most preprocessing challenges involved cleaning the dataset, the first step in the pipeline. Since the text is in English, issues like tokenization errors from multiple languages were not a concern. Additionally, spelling errors which is a common preprocessing challenge were not expected, as the dataset consists of published literature that has been thoroughly edited.

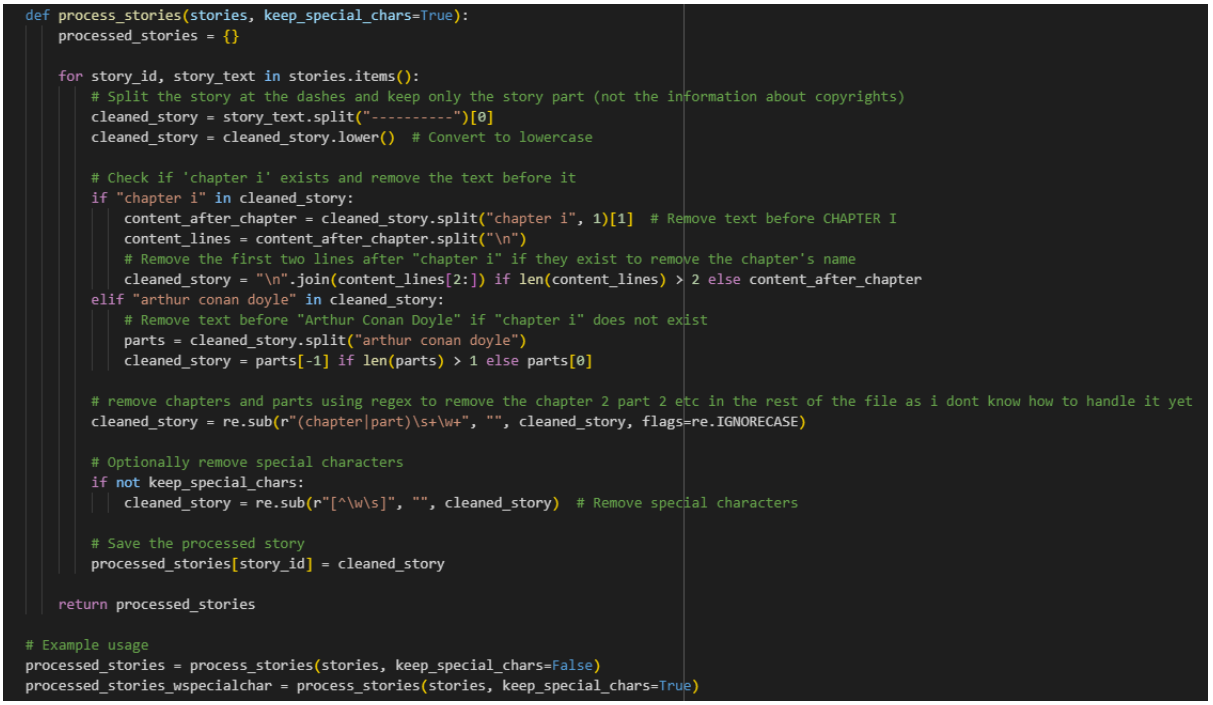
Text Cleaning

In the image below a simple template of a story is analysed. It is important to note that this can be repeated multiple times within a text file if it includes multiple stories or it can slightly change.



The preprocessing involved cleaning and structuring the text for analysis. Copyright text was removed first, as it consistently appeared at the end and was easy to identify. Next, lowercasing was applied for consistency, another straightforward step in standardizing the text.
Titles, author names, and tables of contents were removed, with a conditional approach for stories with chapters, keeping text after the chapter i marker or, if

absent, after the author's name. Furthermore, "chapter," "part," and chapter titles were also removed. Finally, two dataset versions were created: one without special characters and another retaining punctuation for specific tasks.



```python
def process_stories(stories, keep_special_chars=True):
    processed_stories = {}

    for story_id, story_text in stories.items():
        # Split the story at the dashes and keep only the story part (not the information about copyrights)
        cleaned_story = story_text.split("----------")[0]
        cleaned_story = cleaned_story.lower()  # Convert to lowercase

        # Check if 'chapter i' exists and remove the text before it
        if "chapter i" in cleaned_story:
            content_after_chapter = cleaned_story.split("chapter i", 1)[1]  # Remove text before CHAPTER I
            content_lines = content_after_chapter.split("\n")
            # Remove the first two lines after "chapter i" if they exist to remove the chapter's name
            cleaned_story = "\n".join(content_lines[2:]) if len(content_lines) > 2 else content_after_chapter
        elif "arthur conan doyle" in cleaned_story:
            # Remove text before "Arthur Conan Doyle" if "chapter i" does not exist
            parts = cleaned_story.split("arthur conan doyle")
            cleaned_story = parts[-1] if len(parts) > 1 else parts[0]

        # remove chapters and parts using regex to remove the chapter 2 part 2 etc in the rest of the file as i dont know how to handle it yet
        cleaned_story = re.sub(r"(chapter|part)\s+\w+", "", cleaned_story, flags=re.IGNORECASE)

        # Optionally remove special characters
        if not keep_special_chars:
            cleaned_story = re.sub(r"[^\w\s]", "", cleaned_story)  # Remove special characters

        # Save the processed story
        processed_stories[story_id] = cleaned_story

    return processed_stories

# Example usage
processed_stories = process_stories(stories, keep_special_chars=False)
processed_stories_wspecialchar = process_stories(stories, keep_special_chars=True)
```

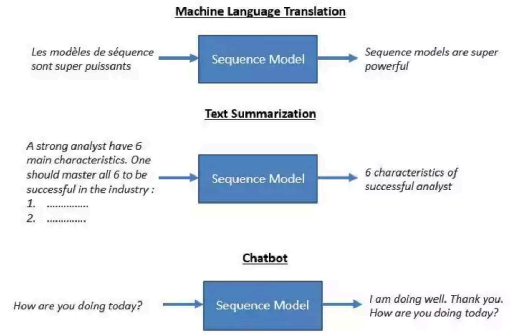
Tokenization, Stop Word Removal and Stemming were applied in a similar way without any unique challenge for the reasons specified above.

Seq2Seq model implementation (Task 2)

Terminology

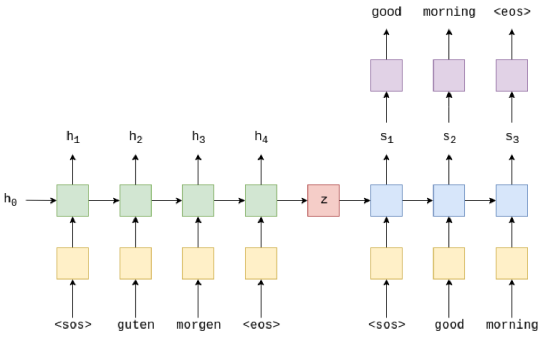
Attention is the key!
Seq2Seq models terminology

What is a Seq2Seq Model?
- Aims to map a fixed length input with a fixed length output even when their length differs.
- As an example translating the English phrase "Good Afternoon" to the Greek phrase "Καλησπέρα".
- In our case, the entire story and the summary have different lengths as the one is significantly shorter than the other.



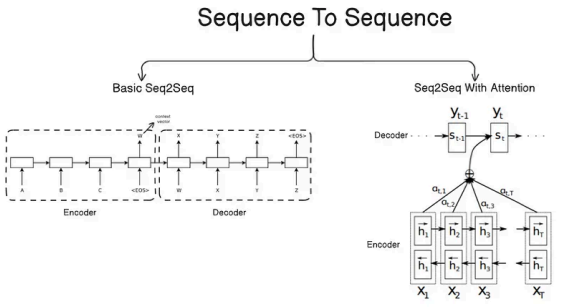
Use Cases of Seq2Seq Models across various applications https://www.analyticsvidhya.com/blog/2020/08/a-simple-introduction-to-sequence-to-sequence-models/

What is Encoder-Decoder LSTM?
- A recurrent neural network designed to address sequence-to-sequence problems, that has proven very effective.
- This architecture is comprised of two models:
  - one for reading the input sequence and encoding it into a fixed-length vector (Encoder)
  - a second for decoding the fixed-length vector and outputting the predicted sequence (Decoder).



The input sentence, "guten morgen," is processed by appending a start-of-sequence token (<sos>) at the beginning and an end-of-sequence token (<eos>) at the end. This modified sequence is then passed through the embedding layer (yellow) before being fed into the encoder (green). https://github.com/bentrevett/pytorch-seq2seq/blob/main/1%20-%20Sequence%20to%20Sequence%20Learning%20with%20Neural%20Networks.ipynb

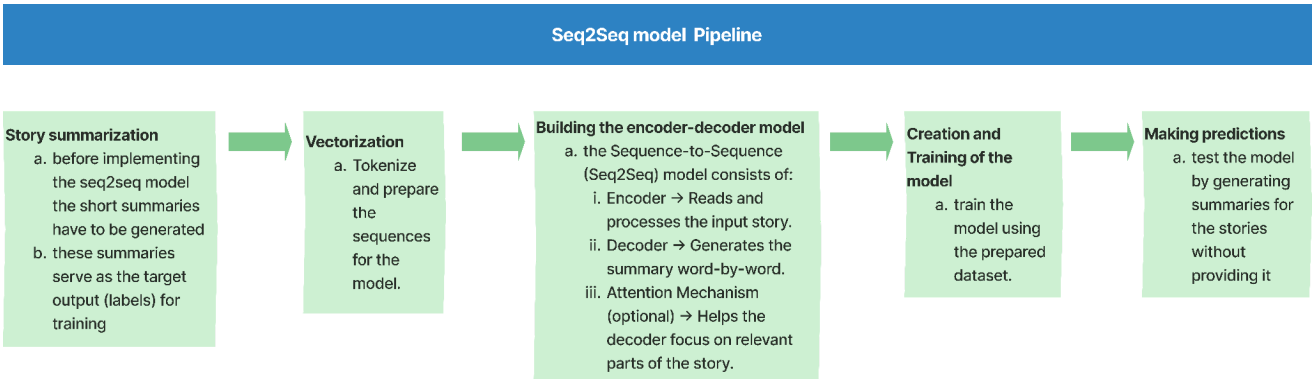
What about attention mechanism?
- Attention solves a problem in traditional Seq2Seq models where the decoder must rely on the final encoder state. This loses important information from earlier parts of long sequences. The attention mechanism allows the model to focus on different parts of the input sequence at each step of the output generation.
- Attention allows the decoder to focus on specific words at each step.
- Improves performance on long text summaries.
- Helps generate more meaningful summaries.



https://medium.com/@fraidoonomarzai99/seq2seq-encoder-decoder-and-attention-mechanism-in-depth-417ef88037f1

Pipeline

The steps that are going to be followed to implement a seq2seq model are the following:



Story Summarisation

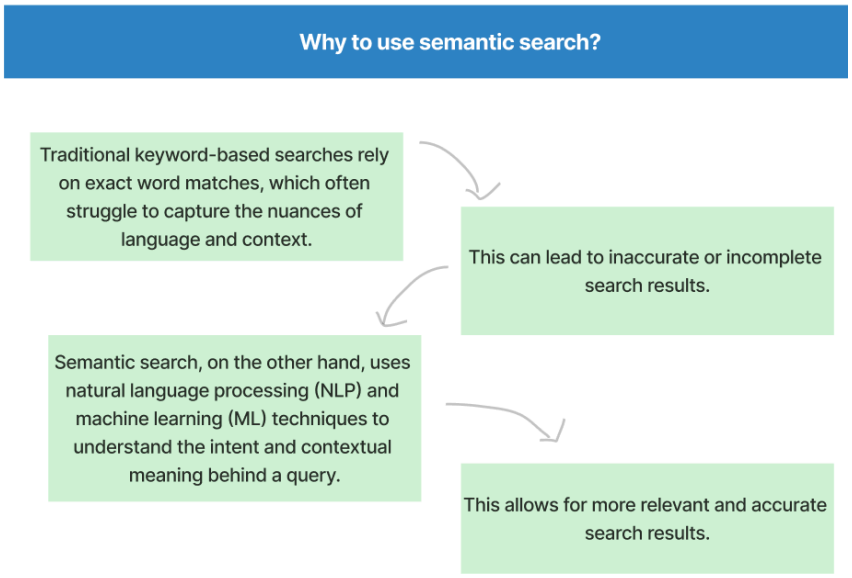
One task in this assignment is to implement a Seq2Seq model to summarize Sherlock Holmes stories into 50-100 word summaries. Since generating summaries is not the main focus, pre-existing summaries will be used instead. While models like BART or T5-small would provide high-quality summaries, they are computationally expensive and outside the project scope. Extractive methods based on frequent sentences were tested but proved not to be good enough. Using pre-made summaries results in a smaller training set, leading to worse performance, which should be addressed in future work.

Building the encoder-decoder model

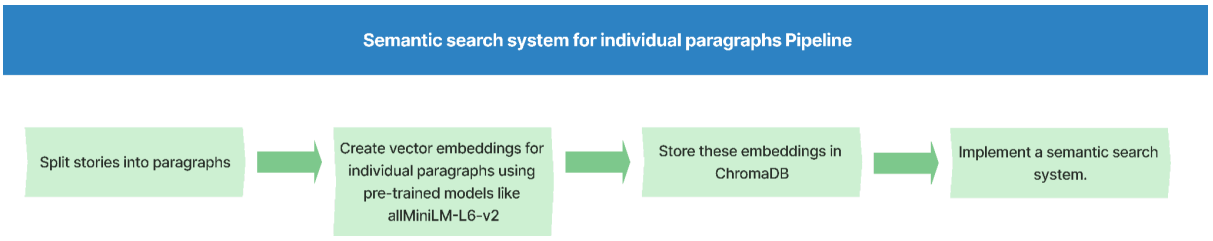
In this project two versions for the implementation of seq2seq models were implemented. The first version uses the simple encoder-decoder architecture while the second one is an encoder-decoder architecture with attention. The reason behind this choice is because of the way a traditional Seq2Seq model works, it loses important information from earlier parts of long sequences. The attention mechanism allows the model to focus on different parts of the input sequence at each step of the output generation. In a few words, attention is implemented with the goal of generating more meaningful summaries.

Implementation of a semantic search system for individual paragraphs (Task 3)

This task involves implementing a semantic search system for individual paragraphs. A common question that may arise is: Why choose semantic search over traditional keyword-based methods?



Pipeline



As it can be noticed from the diagram above, to implement a semantic search system, paragraphs must be converted into vector embeddings, which map text into a high-dimensional space for efficient similarity calculations (e.g., cosine similarity). This allows retrieval of semantically similar passages, even without exact word matches.
Since traditional databases struggle with high-dimensional vector searches due to indexing inefficiencies, ChromaDB is used for optimized storage and retrieval. The final step is implementing the search system and evaluating its performance across various test cases.

Splitting stories into paragraphs

Before splitting the text into paragraphs, an analysis identified that paragraphs are separated by two blank lines. To handle the dialogue-heavy nature of the text, sections were considered paragraphs only if they were longer than two lines, ensuring short dialogue exchanges were not mistakenly treated as standalone

paragraphs. The example in the image visually demonstrates how this approach maintains logical paragraph grouping.

```
I remember the date very well, for it was in the same month that
Holmes refused a knighthood for services which may perhaps some day
be described. I only refer to the matter in passing, for in my
position of partner and confidant I am obliged to be particularly
careful to avoid any indiscretion. I repeat, however, that this
enables me to fix the date, which was the latter end of June, 1902,
shortly after the conclusion of the South African War. Holmes had
spent several days in bed, as was his habit from time to time, but he
emerged that morning with a long foolscap document in his hand and a
twinkle of amusement in his austere gray eyes.

"There is a chance for you to make some money, friend Watson," said
he. "Have you ever heard the name of Garrideb?"

I admitted that I had not.
```

Example that depicts how the text includes conversations and that's the reason we can't split paragraphs solely on the number of empty lines.

Create vector embeddings

The pretrained model used for vector embedding is all-MiniLM-L6-v2, selected for its lightweight design and high accuracy in embedding tasks. Additionally, it is the default model for many tasks, such as query embedding, ensuring consistency and eliminating the need for further tuning to match the vector embeddings.
The code for creating the vector embeddings is pretty simple as it includes only the initialisation of the model and the generation of the embeddings for each paragraph.

```
[8]  from sentence_transformers import SentenceTransformer

     # Initialize the SentenceTransformer model
     model = SentenceTransformer('sentence-transformers/all-MiniLM-L6-v2')

     # Generate embeddings for each paragraph
     story_embeddings = {}

     for story_id, paragraphs_list in paragraphs.items():
         embeddings = model.encode(paragraphs_list)  # Generate embeddings for all paragraphs of the story
         story_embeddings[story_id] = embeddings
```

Code for creating embeddings.

Store embeddings to ChromaDB

Storing embeddings in ChromaDB is straightforward, involving client initialization and creating a database. The main complexity lies in using the collection.add() function to store text and embeddings together for future retrieval. ChromaDB automatically

downloads all-MiniLM-L6-v2 to generate embeddings, eliminating the need for separate computations.
It can be observed that there is no compelling justification for computing the embeddings separately in this case, as both the manual embedding and ChromaDB's automatic embedding use all-MiniLM-L6-v2, adding unnecessary steps. However, separating the embedding process allows flexibility to switch to a different embedding model in the future if all-MiniLM-L6-v2 doesn't perform well, ensuring the system isn't tied to a single model.

Perform Similarity Search

To perform a semantic similarity search, the query() function in ChromaDB is used, allowing natural language queries. This function converts the query into an embedding and retrieves the most relevant results using a prebuilt similarity computation. While efficient, the similarity metric can be customized if needed.
By default, ChromaDB uses cosine similarity, a widely used metric in NLP for comparing embeddings by measuring angular distance between vectors. The encoding process employs all-MiniLM-L6-v2, the same model used for embedding stored paragraphs. This ensures vector space consistency, eliminating the need to modify the embedding strategy. If a different model were used, embedding both queries and documents with the same model would be necessary for accurate similarity calculations.

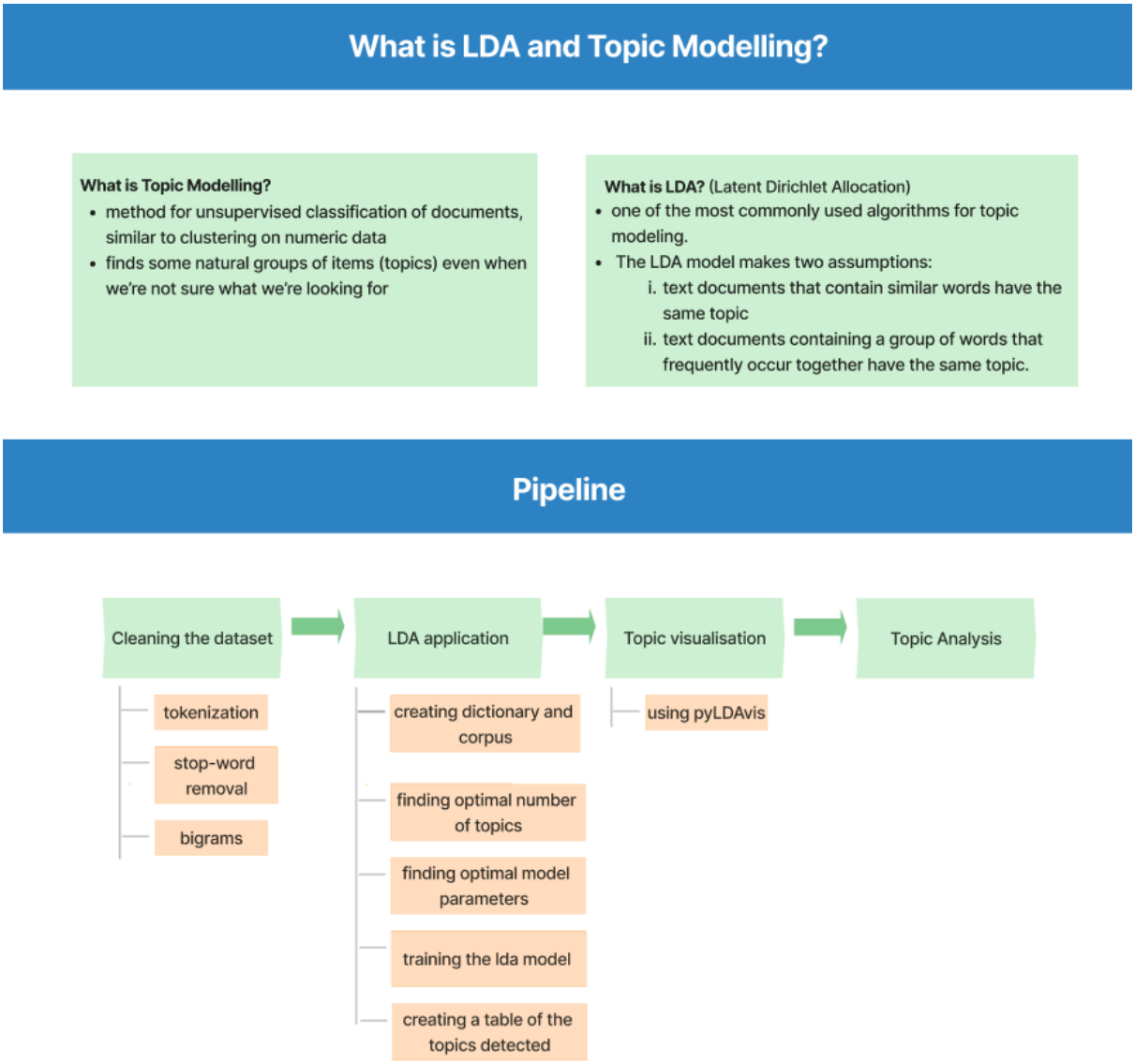
```
# Perform similarity query
results = collection.query(
    query_texts=["Find paragraphs where Sherlock is solving a case."],
    n_results=7  # Number of top similar results
)

results
```

Similarity query performed with 7 results

Performance Topic Modelling using LDA and topic analysis (Task 4)

The fourth task of the project was to implement topic modelling using LDA. For a better understanding of their meaning and a brief pipeline of its implementation the following diagram was created:



Cleaning the dataset

The first step in LDA is data cleaning and preparation. Text needs to be tokenized and stopwords removed, as they add noise without contributing meaningful information. Additionally, character names, countries, and other frequently repeated terms are irrelevant for topic modeling and are included as extra_stopwords.
To enhance topic coherence, bigrams are identified (e.g., "baker street" becomes "baker_street"), treating commonly paired words as a single entity, which adds more meaning to the topics.

```python
extra_stopwords = {
    "holmes", "watson", "moriarty", "mycroft", "gregson", "hopkins", "carruthers",
    "rucastle", "garrideb", "mortimer", "baronet", "bannister", "hilton", "barclay",
    "pycroft", "brunton", "mcpherson", "stackhurst", "warren", "cadogan", "gennaro",
    "bennett", "robert", "jefferson", "garcia", "mccarthy", "sholto", "milverton",
    "arthur", "wilson", "oldacre", "hosmer", "stoner", "ferrier", "amberley",
    "tregennis", "barker", "colleague", "inspector", "colonel", "professor",
    "sergeant", "coroner", "cubitt", "godfrey", "douglas", "hudson", "emsworth",
    "stanley", "charlington", "hunter", "neville", "hatherley", "windibank",
    "stepfather", "society", "toller", "machine", "spaniel", "leonardo",
    "openshaw", "briony", "horsham", "roylott", "bohemia", "sutherland", "grimesby_roylott",
    "birmingham", "prague", "mountjames", "godolphin_street", "minister",
    "statesman", "sylvius", "trevelyan", "madame", "dunbar", "josiah", "overton",
    "lestrade", "mcmurdo", "charles", "france", "woking", "brixton", "shoscombe",
    "ferguson", "gibson", "murdoch", "phelps", "trevor", "cunningham", "maberley"
}

# Extend with standard NLTK stopwords
stop_words = set(stopwords.words('english')).union(extra_stopwords)

# Remove extra stopwords from preprocessed tokenized stories
stories_filtered = {
    story_id: [word for word in tokens if word not in stop_words]
    for story_id, tokens in stories_tokenized_nostopwords.items()
}

# Convert tokenized stories to a list of lists for Phrases training
tokenized_texts = list(stories_filtered.values())

# Train Gensim Phrases model to detect bigrams
bigram_phrases = Phrases(tokenized_texts, min_count=3, threshold=15)
bigram_model = Phraser(bigram_phrases)

# Apply the bigram model to tokenized texts (after extra stopword removal)
stories_with_phrases = {
    story_id: bigram_model[story_tokens]
    for story_id, story_tokens in stories_filtered.items()
}

# Example: Checking a story with bigrams applied
story_id_example = list(stories_with_phrases.keys())[0]  # Get first story ID
print(stories_with_phrases[story_id_example])
```

Cleaning code of LDA before implementation

LDA application

Lda application actually includes several steps which were mentioned on the diagram above. The steps above were performed multiple times until most meaningful topics were obtained.

```python
from gensim import corpora

# Create a Gensim dictionary from the processed stories with phrases
gensim_dict = corpora.Dictionary(stories_with_phrases.values())
gensim_dict.filter_extremes(no_below=3, no_above=0.3) # Filters out tokens that appear in fewer than
# Filters out tokens that appear in more than 30% of the documents.

# Create a Gensim corpus using doc2bow for each story with phrases
gensim_corpus = [gensim_dict.doc2bow(tokens) for tokens in stories_with_phrases.values()]

# Output example
print("Dictionary:", gensim_dict.token2id)
print("Corpus:", gensim_corpus)
```

When creating the Gensim dictionary and corpus, tokens were filtered using specific thresholds: words appearing in **fewer than 3 documents (no_below=3)** or in **more than 30% of documents (no_above=0.3)** were excluded. This ensures rare tokens (noise) and overly frequent tokens (less informative) are removed. These thresholds, determined through experimentation, provided optimal results based on:

- **Dataset Size**: Smaller datasets may require lower thresholds to retain rare but meaningful words.
- **Task Objective**: Filtering ensures topics are not dominated by generic or irrelevant terms.
- **Document Distribution**: Normalizes datasets with varying document lengths by removing skewed frequent terms.

Moreover, LDA contains several hyperparameters that require careful tuning to achieve optimal results. Manually adjusting these hyperparameters can be challenging, which is why the evaluate_lda function is used to automate the process. Below, the key hyperparameters are presented, along with an explanation of how adjusting them impacts the model's behavior.

```
[ ] from gensim.models import LdaModel
    from gensim.models import CoherenceModel
    import numpy as np

    # Define a function to evaluate coherence scores for various parameters
    def evaluate_lda(num_topics_range, alpha_vals, beta_vals, passes=20):
        best_model = None
        best_coherence = -1
        for num_topics in num_topics_range:
            for alpha in alpha_vals:
                for beta in beta_vals:
                    lda_model = LdaModel(
                        corpus=gensim_corpus,
                        id2word=gensim_dict,
                        num_topics=num_topics,
                        alpha=alpha,
                        eta=beta,
                        passes=passes,
                        random_state=42
                    )
                    coherence_model = CoherenceModel(
                        model=lda_model,
                        texts=stories_with_phrases.values(),
                        dictionary=gensim_dict,
                        coherence='c_v'
                    )
                    coherence_score = coherence_model.get_coherence()
                    print(f"Num Topics: {num_topics}, Alpha: {alpha}, Beta: {beta}, Coherence: {coherence_score}")
                    if coherence_score > best_coherence:
                        best_coherence = coherence_score
                        best_model = lda_model
        return best_model, best_coherence

    # Define ranges for hyperparameters
    num_topics_range = range(4,5)  # Test topics from 5 to 15
    alpha_vals = ['auto','symmetric', 'asymmetric', 0.01, 0.1]
    beta_vals = ['auto','symmetric', 0.01, 0.1]

    # Run the evaluation
    best_lda_model, best_score = evaluate_lda(num_topics_range, alpha_vals, beta_vals)
    print(f"Best Coherence Score: {best_score}")
```

evaluate_lda function

Hyperparameter	What it does	Possible Values	Effect on model
alpha	Controls how many topics each document is associated with.	'auto', 'symmetric', 'asymmetric', 0.01, 0.1	Each document focuses on fewer topics. Large values: Each document spreads across many topics.
eta	Controls how many words each topic contains.	'auto', 'symmetric', 0.01, 0.1	Small values: Each topic focuses on fewer, more specific words. Large values: Topics cover a broader range of words.
passes	The number of times the model goes over the dataset during training (like training epochs).	Integer (10, 20,etc)	Fewer passes: Faster training, but topics may not be well-separated. More passes: Better topics, but slower training.
number of topics	The number of topics to identify in the corpus.	integer(1,2,3,etc)	Fewer topics: Topics are broader and less specific. More topics: Topics are more specific but may overfit or fragment

hyperparameters table

Various hyperparameter combinations were tested to find the optimal configuration. Initially, auto was used for both alpha and eta, with the number of topics determined by the highest coherence score from the automated search. Subsequently, other combinations were explored through automated search and manual adjustments, providing insight into how each hyperparameter impacted the model's ability to generate meaningful and coherent topics.

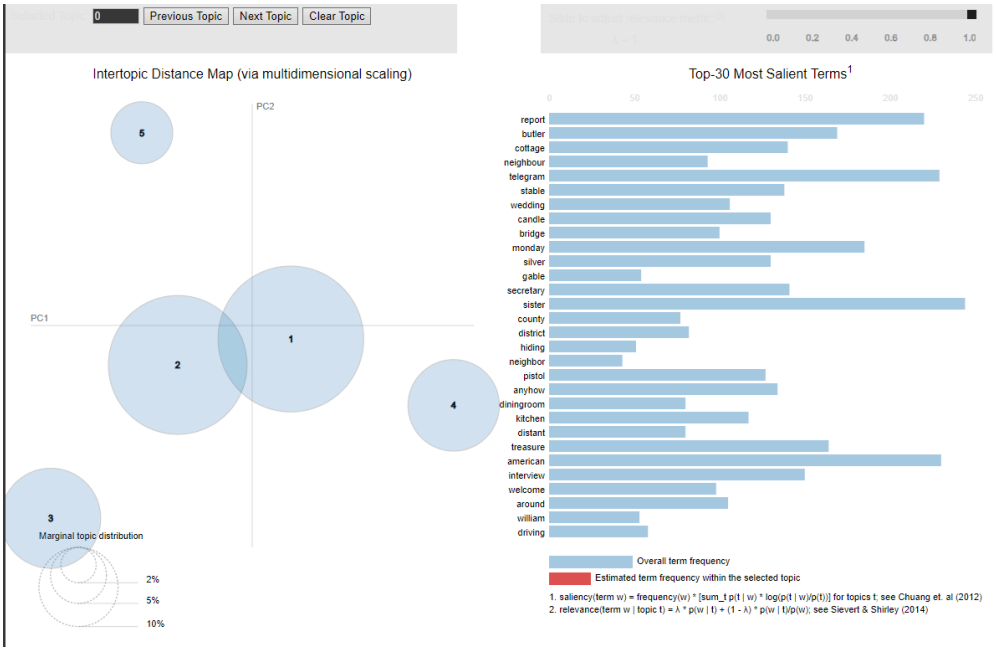
Visualisation

The final step in LDA is **visualization**, which helps to understand the relationships between topics and their distributions within the dataset. Visualizing the topics allows us to see how distinct or closely related they are. For example:

- **Circles in the visualization represent topics**, and their **sizes reflect their prominence** (how much of the corpus they explain).
- **Overlapping circles** indicate that topics are closely related, meaning the model has not clearly distinguished them. This could signal that:
    - The **number of topics** is too high, causing similar topics to overlap.
    - The **alpha or beta values** may need adjustment to ensure better differentiation between topics.

A clear and meaningful topic model should have well-separated circles with minimal overlap, showing that topics are distinct and interpretable. If the circles overlap too much, tuning the hyperparameters (e.g., reducing the number of topics or adjusting alpha and beta) is necessary to improve topic separation.

For visualization pyLDAvis was used:



Topics analysis

Topic analysis is a crucial step in LDA. The reason is that the way a computer identifies and differentiates topics often differs from how a human perceives them. At first glance, the topics generated by the model might not seem meaningful, even if the coherence score is high *(A high coherence score means the words within a topic are closely related and often co-occur in the dataset)*. This is why human interpretation is essential. Analyzing the topics helps us understand the logic behind the splits, interpret the meaning of each topic, and determine how well the topics correspond to the dataset. It also allows us to evaluate whether the generated topics are meaningful and useful or if they are irrelevant and require further refinement.

Results & Discussion

As mentioned, preprocessing can be considered the most important task as the quality of the data directly affects the quality of every output. While preprocessing solved many of expected challenges, several issues require further future work and moreover unexpected issues arise.

For example, a major challenge was text files containing multiple stories, often leading to duplicate stories (a story might appear both as part of a book and as a standalone file). This issue became evident in Task 3, where duplicate paragraphs appeared during searches, requiring a filtering approach. This highlights the need for improved segmentation to split books into separate stories and eliminate duplicates in future iterations.

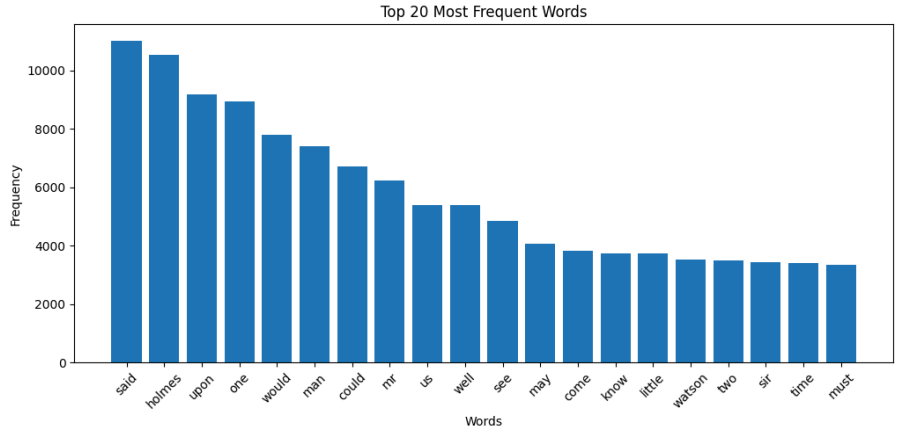
```
 **Top Matching Paragraphs:**

**Result 1:**
**Story Number:** 13
**Paragraph Number:** 240
**Document (Paragraph):**
i have only one further note of this case. it is the letter which
     holmes wrote in final answer to that with which the narrative begins.
     it ran thus:
**Similarity Score:** 0.8144
**Metadata:** {'paragraph': 240, 'source': 13}
--------------------------------------------------
**Result 2:**
**Story Number:** 11
**Paragraph Number:** 4068
**Document (Paragraph):**
i have only one further note of this case. it is the letter which
     holmes wrote in final answer to that with which the narrative begins.
     it ran thus:
**Similarity Score:** 0.8144
**Metadata:** {'paragraph': 4068, 'source': 11}
--------------------------------------------------
**Result 3:**
**Story Number:** 60
**Paragraph Number:** 38
**Document (Paragraph):**
i have only one further note of this case. it is the letter which
     holmes wrote in final answer to that with which the narrative begins.
     it ran thus:
**Similarity Score:** 0.8144
**Metadata:** {'paragraph': 38, 'source': 60}
--------------------------------------------------
```

Screenshot showing how the same paragraph across 3 different text files is the top 3 matching results of semantic search before filtering out the results.

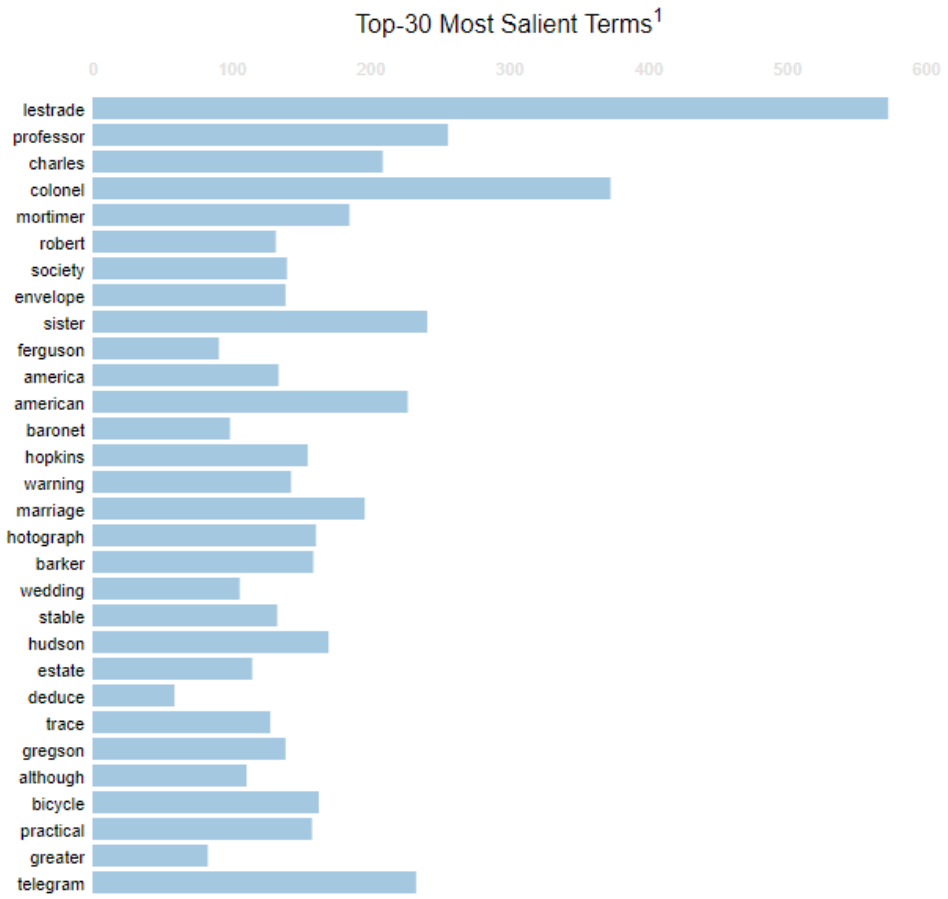
Another task that highlights the importance of preprocessing is analyzing **word distribution** and how removing stopwords and unnecessary high-frequency words improves results. By focusing on meaningful content, preprocessing ensures the data is more relevant and interpretable.

The word distribution chart below illustrates the need to eliminate irrelevant terms, such as character names and generic words, to enhance topic modeling effectiveness in Task 4.



A good example of how important word distribution is the chart above

The bar chart shows the Top 20 Most Frequent Words in the dataset, with "said," "Holmes," and "upon" leading. These high-frequency words, including character names (e.g., "Holmes," "Watson") and common verbs (e.g., "said," "would"), dominate topic modeling, making it harder to extract meaningful themes like "mystery" or "investigation." This highlights the importance of tokenization and removing stopwords and frequent words for tasks like LDA.




```
(0, '0.026*"holmes" + 0.009*"watson" + 0.006*"nothing" + 0.006*"little" + 0.005*"friend" + 0.004*"window" + 0.004*"morning"')
(1, '0.010*"holmes" + 0.009*"mcmurdo" + 0.006*"little" + 0.005*"douglas" + 0.005*"thought" + 0.004*"nothing" + 0.004*"mcginty"')
(2, '0.006*"moriarty" + 0.006*"watson" + 0.005*"holmes" + 0.004*"professor" + 0.003*"friend" + 0.002*"career" + 0.002*"english"')
(3, '0.017*"holmes" + 0.010*"little" + 0.005*"matter" + 0.005*"however" + 0.005*"watson" + 0.005*"nothing" + 0.004*"friend"')
(4, '0.000*"holmes" + 0.000*"watson" + 0.000*"little" + 0.000*"street" + 0.000*"nothing" + 0.000*"thought" + 0.000*"window"')
(5, '0.000*"holmes" + 0.000*"little" + 0.000*"watson" + 0.000*"nothing" + 0.000*"matter" + 0.000*"window" + 0.000*"morning"')
(6, '0.029*"holmes" + 0.011*"watson" + 0.006*"little" + 0.006*"matter" + 0.005*"friend" + 0.005*"nothing" + 0.004*"seemed"')
(7, '0.014*"holmes" + 0.008*"little" + 0.007*"however" + 0.006*"treasure" + 0.006*"sholto" + 0.006*"morstan" + 0.005*"thought"')
(8, '0.013*"holmes" + 0.009*"baskerville" + 0.008*"watson" + 0.007*"charles" + 0.007*"stapleton" + 0.006*"mortimer" + 0.006*"barrymore"')
```

Examples of LDA where most of the top words are people before removing frequent words

After removing the unnecessary noise from task 4, a few topics were formed but they didn't really make any sense for human understanding. As presented below:

```
(0, '0.004*"monday" + 0.003*"mycroft" + 0.003*"office" + 0.003*"moriarty" + 0.002*"interpreter" + 0.002*"patient" + 0.002*"pinner"')
(1, '0.013*"douglas" + 0.010*"barker" + 0.006*"baldwin" + 0.006*"valley" + 0.004*"chicago" + 0.003*"cottage" + 0.003*"anyhow"')
(2, '0.011*"warren" + 0.010*"gregson" + 0.007*"lodger" + 0.006*"landlady" + 0.005*"signal" + 0.005*"italian" + 0.004*"american"')
(3, '0.004*"lestrade" + 0.002*"colonel" + 0.002*"charles" + 0.002*"mortimer" + 0.002*"telegram" + 0.001*"treasure" + 0.001*"professor"')
(4, '0.010*"colonel" + 0.005*"trevor" + 0.004*"hudson" + 0.004*"treaty" + 0.003*"joseph" + 0.003*"commissionaire" + 0.003*"charles"')
(5, '0.005*"robert" + 0.004*"professor" + 0.004*"ferguson" + 0.003*"american" + 0.003*"murdoch" + 0.003*"godfrey" + 0.002*"revolver"')
(6, '0.006*"lestrade" + 0.003*"office" + 0.003*"telegram" + 0.003*"mycroft" + 0.003*"garcia" + 0.003*"france" + 0.002*"gregson"')
(7, '0.024*"colonel" + 0.010*"barclay" + 0.010*"stable" + 0.007*"trainer" + 0.005*"silver" + 0.005*"horse" + 0.004*"simpson"')
(8, '0.003*"lestrade" + 0.003*"sister" + 0.002*"office" + 0.002*"coronet" + 0.002*"marriage" + 0.002*"colonel" + 0.002*"hunter"')
```

To improve results and make them more interpretable for humans, hyperparameter tuning was essential, as it significantly enhanced the clarity and relevance of topics.

Notably, a higher coherence score does not always equate to more interpretable topics. For instance, the first example achieved a coherence score of 0.53, while the second scored 0.46, yet the latter produced more meaningful and understandable topics for human analysis.

	Word 1	Word 2	Word 3	Word 4	Word 5	Word 6	Word 7	Word 8	Word 9	Word 10	Word 11	Word 12	Word 13
Topic 0	baskerville	stapleton	mcginty	barrymore	treasure	valley	morstan	baldwin	drebber	macdonald	stangerson	mountain	launch
Topic 1	stable	straker	blessington	coachman	silver	trainer	soldier	photograph	mawsons	coronet	morrison	ceased	thirty
Topic 2	baynes	soames	cigarette	glass	cushing	butler	secretary	bureau	kitchen	diningroom	turner	boscombe	height
Topic 3	secretary	lodger	bridge	signal	gable	masser	mistress	butler	ancient	ronder	quarrel	france	winter
Topic 4	bicycle	cottage	staunton	corridor	track	school	butler	captain	bullet	secretary	french	dancing	soames

Coherence score 0.53

	Word 1	Word 2	Word 3	Word 4	Word 5	Word 6	Word 7	Word 8	Word 9	Word 10	Word 11	Word 12	Wor
Topic 0	treasure	morstan	launch	thaddeus	photograph	coronet	holder	wedding	athelney	indian	jonathan	handkerchief	advertise
Topic 1	mcginty	valley	drebber	baldwin	macdonald	stangerson	butler	mountain	cottage	birlstone	chicago	straker	ver
Topic 2	bicycle	staunton	soames	secretary	glass	cigarette	track	bullet	disappearance	dancing	norwood	wilder	s
Topic 3	baskerville	stapleton	barrymore	secretary	signal	bridge	butler	grimpen	lodger	stapletons	material	melancholy	a

Coherence score 0.46

The second version of the topic model demonstrates significant improvements in thematic clarity and relevance. Topics 0 and 2 that are the most meaningful are analysed below:

Topic 0:
This topic centers around weddings, and relationships, with words like "treasure," morstan,haddeus,coronet, wedding,and holder. It suggests a marriage-related theme and family connections. The inclusion of terms such as "captain_morstan" and "photograph" hints at connections to family heirlooms or mysteries linked to past events.

Topic 2:
This topic focuses on mysteries involving disappearances and murder. Key words such as bicycle, disappearance and bullet indicate a narrative involving an unexplained vanishing or a crime. The presence of secretary, corridor and school suggests connections to professional or academic environments, possibly reflecting stories of intrigue in formal settings.

Further **hyperparameter tuning** and improved **stopword handling** are needed for better results. Preprocessing directly impacts performance, as changes like removing chapter titles throughout the story (rather than just the first chapter)

showed noticeable differences. This emphasizes the importance of robust preprocessing to ensure consistency and improve overall model accuracy.

Preprocessing was also proven crucial in Task 3, as proper paragraph separation was essential for accurate results. Some paragraphs were incorrectly split due to the dialogue-heavy nature of the stories, occasionally leading to irrelevant outputs. Although this issue was not very frequent, it became noticeable given the small size of the dataset, where even minor inconsistencies can have a significant impact. However, as the inconsistencies were relatively rare and not drastically affecting the model, they will not be addressed in this iteration.

From a practical standpoint, initializing ChromaDB and creating the dataset was straightforward. The challenge lies in storing both the vectors and embeddings together in a structured format for easy access during queries.
Another tricky aspect is how to effectively phrase the query to retrieve relevant results. To illustrate this, the following query example is presented:

"Find paragraphs where Sherlock is solving a case."

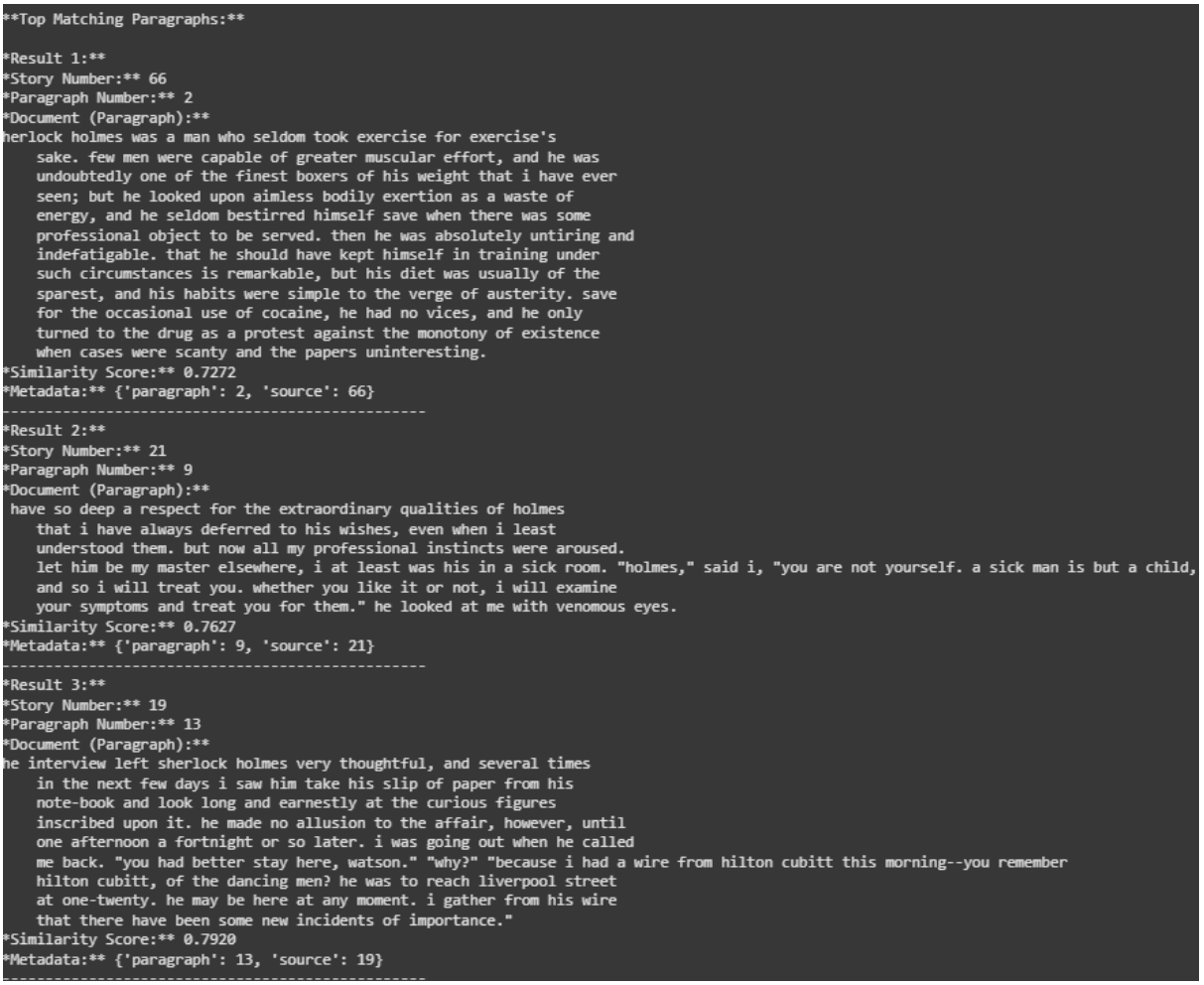
This query not only retrieves results about crime cases but some of the results may instead relate to phrases like "in this case scenario". To fix this problem the question can be rephrased to be more specific or include categories through metadata which is not possible at the moment as they are not available for this dataset.

```
**Top Matching Paragraphs:**

**Result 1:**
**Story Number:** 13
**Paragraph Number:** 240
**Document (Paragraph):**
 have only one further note of this case. it is the letter which
    holmes wrote in final answer to that with which the narrative begins.
    it ran thus:
**Similarity Score:** 0.8144
**Metadata:** {'paragraph': 240, 'source': 13}
--------------------------------------------------
**Result 2:**
**Story Number:** 11
**Paragraph Number:** 4068
**Document (Paragraph):**
 have only one further note of this case. it is the letter which
    holmes wrote in final answer to that with which the narrative begins.
    it ran thus:
**Similarity Score:** 0.8144
**Metadata:** {'paragraph': 4068, 'source': 11}
--------------------------------------------------
**Result 3:**
**Story Number:** 60
**Paragraph Number:** 38
**Document (Paragraph):**
 have only one further note of this case. it is the letter which
    holmes wrote in final answer to that with which the narrative begins.
    it ran thus:
**Similarity Score:** 0.8144
**Metadata:** {'paragraph': 38, 'source': 60}
```

“Find paragraphs where Sherlock is solving a case.” results

The duplicate entries problem, shown above, was addressed by calculating cosine similarity between paragraphs and removing files with a similarity score above 0.90. Initially, standalone text files with individual stories were removed, but undetected duplicates remained. To resolve this, text files containing multiple stories were deleted instead.
While this method significantly reduced duplicates, it did not remove all of them. However, the number of undetected duplicates is small, and meaningful results were still achieved. Further refinements could improve this approach, but they are not critical for reliable outcomes.



```
**Top Matching Paragraphs:**

*Result 1:**
*Story Number:** 66
*Paragraph Number:** 2
*Document (Paragraph):**
herlock holmes was a man who seldom took exercise for exercise's
    sake. few men were capable of greater muscular effort, and he was
    undoubtedly one of the finest boxers of his weight that i have ever
    seen; but he looked upon aimless bodily exertion as a waste of
    energy, and he seldom bestirred himself save when there was some
    professional object to be served. then he was absolutely untiring and
    indefatigable. that he should have kept himself in training under
    such circumstances is remarkable, but his diet was usually of the
    sparest, and his habits were simple to the verge of austerity. save
    for the occasional use of cocaine, he had no vices, and he only
    turned to the drug as a protest against the monotony of existence
    when cases were scanty and the papers uninteresting.
*Similarity Score:** 0.7272
*Metadata:** {'paragraph': 2, 'source': 66}
--------------------------------------------------
*Result 2:**
*Story Number:** 21
*Paragraph Number:** 9
*Document (Paragraph):**
 have so deep a respect for the extraordinary qualities of holmes
    that i have always deferred to his wishes, even when i least
    understood them. but now all my professional instincts were aroused.
    let him be my master elsewhere, i at least was his in a sick room. "holmes," said i, "you are not yourself. a sick man is but a child,
    and so i will treat you. whether you like it or not, i will examine
    your symptoms and treat you for them." he looked at me with venomous eyes.
*Similarity Score:** 0.7627
*Metadata:** {'paragraph': 9, 'source': 21}
--------------------------------------------------
*Result 3:**
*Story Number:** 19
*Paragraph Number:** 13
*Document (Paragraph):**
he interview left sherlock holmes very thoughtful, and several times
    in the next few days i saw him take his slip of paper from his
    note-book and look long and earnestly at the curious figures
    inscribed upon it. he made no allusion to the affair, however, until
    one afternoon a fortnight or so later. i was going out when he called
    me back. "you had better stay here, watson." "why?" "because i had a wire from hilton cubitt this morning--you remember
    hilton cubitt, of the dancing men? he was to reach liverpool street
    at one-twenty. he may be here at any moment. i gather from his wire
    that there have been some new incidents of importance."
*Similarity Score:** 0.7920
*Metadata:** {'paragraph': 13, 'source': 19}
--------------------------------------------------
```

The results after further adjustments which seem to be improved.

Initial improvements showed progress, but further query experimentation led to no significant advancements. This raised a key question: **Is the issue with semantic search or the nature of the queries?** Since paragraphs exist within a broader text, **semantic search may struggle with general questions** requiring combined

information from multiple paragraphs but should perform well for **direct questions** answerable within a single paragraph.

To test this hypothesis, the following direct question was asked:

"Who is Sherlock Holmes' landlady?"

The top answer to the question above was

```
 **Top Matching Paragraphs:**

**Result 1:**
**Story Number:** 21
**Paragraph Number:** 1
**Document (Paragraph):**
mrs. hudson, the landlady of sherlock holmes, was a long-suffering
     woman. not only was her first-floor flat invaded at all hours by
     throngs of singular and often undesirable characters but her
     remarkable lodger showed an eccentricity and irregularity in his life
     which must have sorely tried her patience. his incredible untidiness,
     his addiction to music at strange hours, his occasional revolver
     practice within doors, his weird and often malodorous scientific
     experiments, and the atmosphere of violence and danger which hung
     around him made him the very worst tenant in london. on the other
     hand, his payments were princely. i have no doubt that the house
     might have been purchased at the price which holmes paid for his
     rooms during the years that i was with him.
**Similarity Score:** 0.5736
**Metadata:** {'paragraph': 1, 'source': 21}
```

In this case result 1 is the perfect answer as it starts with "mrs. hudson, the landlady of sherlock holmes".

Somehow, the correct answer, "Mrs. Hudson is the landlady of Sherlock Holmes," had a lower similarity score than irrelevant results. This suggests that semantic search prioritizes contextual similarity over exact matches, sometimes ranking loosely related paragraphs higher. This limitation highlights the need for fine-tuning or post-processing to improve ranking accuracy.
Dataset size also impacted Task 2, as only 11 summaries were initially available, later increased to 19. While this expansion improved results slightly, the limited training data remained a key factor in the model's underperformance.

Another significant factor influencing the results was the attention mechanism which drastically improved the results. It transformed repetitive, meaningless word lists that were the same for each story into more varied and relevant summaries for each story. While the improvement wasn't drastic as it simply repeated a few more words related to the story as a "summary", it demonstrated the importance of attention in enhancing model performance.
Additionally, tuning model parameters such as embedding_dim, latent_dim, batch_size, and epochs led to some improvements. Among these, training epochs and early stopping had the most noticeable impact, suggesting the model struggles to generalize effectively. This highlights the need for a larger dataset in future iterations to enhance performance.

A potential solution is to use BART to generate abstract summaries and then use these summaries to train the model.
Alternatively, a Seq2Seq model could be employed to create abstract summaries via reinforcement learning.
Creating summaries using TF-IDF sentence ranking was also attempted but was not successful, as it is incapable of producing abstract summaries and is better suited for extractive summarization.

```
Summary of Story 1:
it was twilight of a lovely spring evening, and even little ryder
     street, one of the smaller offshoots from the edgware road, within a
     stone-cast of old tyburn tree of evil memory, looked golden and
     wonderful in the slanting rays of the setting sun. our eyes fell upon
     a mass of rusted machinery, great rolls of paper, a litter of
     bottles, and, neatly arranged upon a small table, a number of neat
     little bundles. the particular
     house to which we were directed was a large, old-fashioned, early
     georgian edifice, with a flat brick face broken only by two deep bay
     windows on the ground floor. he made his money in real estate, and afterwards in the wheat pit at
     chicago, but he spent it in buying up as much land as would make one
     of your counties, lying along the arkansas river, west of fort dodge. they
     would willingly have subscribed to that soup-plate medal of which the
     criminal had spoken, but an unappreciative bench took a less
     favourable view, and the killer returned to those shades from which
     he had just emerged.
```

Example summary of using TF-IDF

Below a few predicted summaries examples are provided below highlighting what it was analysed above:

Simple encoder-decoder, batch_size = 8, 100 epochs no early stopping:
Predicted Summary 1: holmes holmes holmes holmes holmes holmes holmes holmes holmes holmes holmes holmes holmes holmes holmes holmes holmes holmes sherlock sherlock sherlock sherlock sherlock sherlock sherlock sherlock sherlock sherlock sherlock sherlock sherlock sherlock sherlock sherlock sherlock sherlock

Simple encoder-decoder, batch_size = 8, 100 epochs with early stopping:
Predicted Summary 1: sherlock sherlock sherlock sherlock sherlock sherlock sherlock sherlock sherlock holmes holmes holmes holmes holmes holmes holmes holmes holmes holmes holmes holmes holmes holmes holmes holmes holmes a a a a a a man man man man man man man investigates investigates investigates investigates investigates is is enlisted

Simple encoder-decoder, batch_size = 32, 100 epochs with early stopping:
Predicted Summary 1: holmes holmes sherlock sherlock sherlock sherlock sherlock sherlock sherlock sherlock sherlock sherlock sherlock sherlock holmes holmes holmes holmes holmes holmes holmes holmes a a a a a a a a a a by by by by by by by by a enlisted enlisted its when enlisted evidence watson grace grace mysteries mysteries enlisted enlisted fearing involving involving involving involving involving involving involving exposure involving leads shot words governess grace grace unresolved unresolved governess governess governess grace grace unresolved unresolved watson watson governess grace grace adder highlights governess grace highlights grace highlights highlights governess grace grace unresolved unresolved murder murder fearing

Encoder-decoder with attention mechanism, batch_size = 32, 100 epochs with early stopping:

Predicted Summary 1: tutor's tutor's tutor's convulsed convulsed cab cab cover brother convulsed cab cab cover brother dishonoured convulsed trophy dies cab cab cab cab convulsed cab cab
----
Predicted Summary 10: panelled panelled panelled panelled familiar familiar concussion concussion shrug guy outs adding guy stable outs humour humour declared stable envelope stable incriminate incriminate amoy amoy oath convulse helps creature creature flitted creature creature refreshed refreshed refreshed printed our presume leaders leaders anyone rotten rotten rotten authority classes endless hurry hurry glad varieties gravesend eyelashes familiar 'tragedy estimate estimate crown crown weird croaking humour humour threatened threatened peter's music corroborated ladies ladies ladies excite adding adding haired haired register' register' faintest faintest longed dene tropical bradstreet expressed lawn stable amoy amoy rested burning ballarat randall ballarat come hadn't satin system intimidation
...

Conclusion

This project highlights the importance of a well-structured NLP pipeline in handling complex literary texts, specifically Sherlock Holmes stories. Through a series of tasks such as data preprocessing, sequence-to-sequence summarization, semantic search, and topic modeling using LDA, various challenges and solutions were explored to improve text processing and model performance.
One of the most significant findings is the critical role of preprocessing in any NLP task. Cleaning the dataset had a direct impact on the tasks following, with the

removal of titles, chapter headings, table of contents, and copyright text improving model performance. Additionally, stopword removal and lemmatization were essential for meaningful topic modeling, preventing high-frequency but uninformative words from dominating LDA results. Similarly, handling duplicate stories required custom filtering and improper paragraph splitting required cosine similarity-based duplicate removal to enhance semantic search accuracy. These observations pinpoint that while each task requires a tailored preprocessing pipeline, preprocessing is a fundamental requirement across all NLP tasks. In essence, preprocessing is what separates an average model from a well-performing one. Future work should focus on further refining preprocessing techniques, such as separating files containing multiple stories and integrating more advanced methods like POS tagging to enhance future results.

Another challenge that requires attention is the Seq2Seq summarization model's performance, which was constrained by the limited training data. While adding an attention mechanism led to some improvements, the model still struggled to generate distinct summaries for different stories. Future work could explore larger datasets and take advantage of pretrained models like BART or T5 to create more meaningful summaries while calculating a larger amount of summaries than manually. Additionally, reinforcement learning could be an exciting alternative, especially considering recent advancements in DeepSeek and other data-efficient training methods, which demonstrate that large datasets are not always necessary for effective learning.A key observation in semantic search is that query phrasing has a significant impact on search results. While storing vector embeddings using ChromaDB is effective and straightforward, correct query phrasing is more complicated. Direct questions tend to perform well, whereas broader queries that require combining information from multiple paragraphs result in less relevant answers. Future improvements should focus on fine-tuning similarity ranking, implementing metadata-based filtering, and exploring alternative semantic search techniques.

Finally, LDA topic modeling proves the importance of hyperparameter tuning, demonstrating that coherence scores alone do not always align with human interpretability. Bigram detection and refined stopword removal significantly improved topic clarity, resulting in more distinct and interpretable themes.

Overall, this project provided valuable insights into NLP best practices, emphasizing the importance of preprocessing, high-quality data, and effective hyperparameter tuning. Additionally, arising technologies such as reinforcement learning present promising directions for future research. The lessons learned here will serve as essential tools for optimizing NLP models in future projects.