Individual Final Report

Scholarly Topic Navigator

Data Ingestion, Preprocessing & Embeddings

Aditya Kanbargi

GWID : G40754750

Team LexiCore: Aditya Kanbargi, Trisha Singh, Pramod Krishnachari

George Washington University

December 2025

1. Introduction

The Scholarly Topic Navigator is a Natural Language Processing system we built to help researchers find and understand academic papers more easily. Our team, Team LexiCore, developed this as part of the Master's NLP course at George Washington University. The main problem we wanted to solve was pretty straightforward: there are way too many papers being published these days, especially in ML and NLP, and it's becoming really hard for researchers to keep up with everything relevant to their work.

We split the project into three weeks, with each week building on what came before. In Week 1 (my part), we focused on getting the data ready, which meant collecting papers from multiple sources, cleaning up the text, and generating embeddings. Week 2 handled the core functionality like search, classification, and topic modeling. Week 3 brought everything together with summarization, explainability features, and a nice dashboard interface.

My work for Week 1 covered three main areas. First, I built the data ingestion system that pulls papers from ArXiv, ACL Anthology, and Semantic Scholar. Second, I developed the preprocessing pipeline that cleans and normalizes the text. Third, I generated embeddings using Word2Vec, SBERT, and SciBERT so that downstream tasks would have good vector representations to work with. I also ended up helping Pramod debug some issues in his Week 2 code, which I'll talk about later in this report.

2. Description of Individual Work

2.1 Data Ingestion Pipeline

The goal of data ingestion was to collect as many relevant NLP and ML papers as possible from different sources. Each source has its own strengths, so using multiple sources gives us better coverage. ArXiv is great for recent preprints since papers show up there before they're officially published anywhere else. ACL Anthology is the go-to source for computational linguistics papers and has decades of peer reviewed research. Semantic Scholar adds citation data which helps us understand how papers relate to each other.

ArXiv Integration: For ArXiv, I used the official Python library to query five categories: cs.CL for language stuff, cs.LG and stat.ML for machine learning, cs.AI for general AI, and cs.IR for information retrieval. I grabbed about 5,000 papers from each category, sorted by submission date so we'd get the newest research first. One thing I had to be careful about was duplicate papers. A lot of papers get tagged with multiple categories, like a paper on attention mechanisms might be in both cs.CL and cs.LG. So I added a seen_ids set to track which papers we'd already collected and skip duplicates. This ended up preventing about 15% redundant collection.

ACL Anthology Integration: ACL provides a BibTeX dump of all their entries, which sounds convenient but turned out to be a pain. The standard pybtex library kept crashing on weird entries in the file, things like missing braces or encoding problems. I ended up writing my own parser that goes through line by line and handles errors gracefully instead of crashing. It worked pretty well and parsed all 118,000 entries in about a second and a half. The big surprise was that none of the ACL entries had abstracts. The BibTeX format only stores metadata, not the actual content. I documented this finding since it's important for anyone else working with this data.

Semantic Scholar Integration: The Semantic Scholar API was tricky because of rate limiting. Initially I was getting HTTP 429 errors (too many requests) on more than half my queries, which meant I was only getting around 500 papers. I fixed this by adding exponential backoff, where if a request fails, we wait longer before trying again. I also increased the delay between queries to 3 to 5 seconds. After these changes, the success rate went up to 95% and I was able to collect close to 2,000 papers.

For deduplication across sources, I used a priority system where ACL papers take precedence over Semantic Scholar, which takes precedence over ArXiv. The logic is that if the same paper appears in multiple places, we want to keep the version from the most authoritative source. After filtering out papers with missing abstracts and removing duplicates, we ended up with 21,422 clean papers.

2.2 Text Preprocessing Pipeline

Academic text needs quite a bit of cleaning before you can do anything useful with it. Abstracts often contain URLs, LaTeX commands, weird unicode characters, and other noise that doesn't help with semantic understanding. The preprocessing pipeline I built has nine steps that each handle a different type of cleaning.

The first few steps handle basic cleaning. We remove URLs and email addresses using regex patterns since these don't add meaning. Special characters get stripped out, keeping only letters, numbers, and basic punctuation. I also added language detection using langdetect because our embedding models are trained on English, so non-English papers would produce bad representations. About 0.6% of papers turned out to be in other languages and got filtered out.

For the linguistic processing, I went with spaCy over NLTK for tokenization and lemmatization. spaCy is faster since it's implemented in Cython, and it handles edge cases better, like keeping contractions and hyphenated terms together properly. Stopword removal uses NLTK's list of 179 common English words. Lemmatization reduces words to their base form, so 'transformers' becomes 'transformer' and 'running' becomes 'run'. This helps reduce vocabulary size without losing meaning.

One optimization I made was combining tokenization and lemmatization into a single pass through the spaCy model. Originally I was running spaCy twice, once for tokens and once for lemmas, which was wasteful. The new single pass approach cut processing time roughly in half, from around 20 minutes to under 10 minutes for the full corpus.

2.3 Embedding Generation

Embeddings are how we turn text into numbers that capture meaning. I generated four different types of embeddings because different downstream tasks work better with different representations.

Word2Vec: This is the baseline embedding. I trained it directly on our corpus using gensim with the Skip-gram architecture. The advantage of training on our own data is that domain specific terms like 'transformer' and 'attention' get representations that reflect their NLP meanings rather than everyday English. I used 100 dimensions, a window size of 5, and trained for 10 epochs. For document embeddings, I just averaged all the word vectors together. It's simple and loses word order information, but it's fast and works okay as a baseline.

SBERT: Sentence BERT with the all-MiniLM-L6-v2 model ended up being our main embedding for most downstream tasks. It's a pretty small model with only 22 million parameters, but it produces good 384 dimensional embeddings and runs quickly. I generated embeddings for both abstracts and titles since they're useful for different search scenarios. Abstract embeddings capture the full content while title embeddings are good for quick lookups.

SciBERT: I also experimented with the AllenAI SciBERT model, which is trained specifically on scientific papers. Because it learns from citation links, papers that reference similar work tend to receive closely related embeddings. SciBERT produces 768-dimensional vectors and a much higher average pairwise similarity in our tests (0.74) compared to SBERT (0.36). This initially seemed like an improvement, but the higher similarity made clustering more difficult, when embeddings are too close together, algorithms struggle to separate documents into distinct topics. For this reason, we use SBERT for topic modeling and classification, where clearer separation is important, and reserve SciBERT for pure similarity-search tasks where tighter semantic grouping is desirable.

Table 1: Embedding Comparison

Method	Dimensions	Avg Similarity	Best Use Case
Word2Vec	100	0.47	Baseline, interpretability
SBERT	384	0.36	Clustering, classification
SciBERT	768	0.74	Similarity search

3. Detailed Work Description

3.1 Implementation Details

The ArXiv code is built around the arxiv Python library which handles most of the API complexity. For each category, I create a Search object and iterate through results. The tricky part was handling duplicates across categories. I maintain a seen_ids set throughout all the queries, and before adding any paper, I check if we've seen it before. The paper ID comes from the entry URL, which looks something like 'http://arxiv.org/abs/2401.12345', so I just grab the last part after the slash.

The BibTeX parser reads the file line by line. When it sees a line starting with '@', it knows a new entry is starting. Lines with '=' are field assignments like 'title = {Some Paper Title}'. I strip the braces and quotes from values and store everything in a dictionary. The key insight was using errors='ignore' when opening the file, which skips unreadable characters instead of crashing. Some entries in the ACL dump have encoding issues, probably from copy-paste errors in the original submissions.

For the Semantic Scholar API, each query has a try-except block with retry logic. If we get a 429 error, we wait and try again, doubling the wait time each attempt up to 5 retries. Between successful queries, there's a random delay of 3 to 5 seconds to stay under the rate limit. The randomness helps avoid patterns that might trigger additional limiting.

3.2 Collaborative Work with Pramod

While Pramod was working on Week 2, he ran into several issues that I helped debug. Some of these were related to my Week 1 code and others were general problems we solved together. Here's a summary of the fixes:

Table 2: Collaborative Fixes

Issue	Fix
NLTK punkt_tab missing	Added the download for punkt_tab
Duplicate ArXiv papers	Added seen_ids tracking
ACL abstracts confusion	Documented that BibTeX has no abstracts
S2ORC rate limiting	Added exponential backoff with 5 retries
Still hitting rate limits	Increased delays to 3-5 seconds
Unclear filtering stats	Added logging for abstract availability
Slow preprocessing	Combined spaCy passes for 2x speedup
Sentence count wrong	Fixed segmentation test
Old pipeline code	Switched to optimized function
Only 10k papers used	Changed to process all 21k papers
Non-English papers	Added filtering after language detection
Incomplete summary	Added sentences per paper and ACL note

The sentence segmentation issue was interesting. The code was reporting 1 sentence per paper on average, which was obviously wrong since abstracts typically have 8 to 10 sentences. It turned out that newer NLTK versions require a different data package called punkt_tab, and without it the segmentation just fails silently. After adding the right download, sentence counts looked normal.

Another big improvement was processing the full dataset. Originally the code was only using a 10k sample for speed during development, but that setting got left in. Changing it to use all 21k papers meant the downstream models had access to everything.

4. Results

4.1 Data Ingestion Results

Here's what we ended up with after running the full ingestion pipeline. ArXiv gave us 25,000 papers across the five categories, all with abstracts. ACL parsing worked fine and got 118,000 entries, but none had abstracts so they all got filtered out later. Semantic Scholar contributed about 2,000 papers after fixing the rate limiting issues. The raw total was around 145,000 entries, but after quality filtering and deduplication we ended up with 21,422 usable papers.

Table 3: Ingestion Summary

Source	Raw Count	After Filtering
ArXiv	25,000	20,980
ACL Anthology	118,461	0 (no abstracts)
Semantic Scholar	~2,000	442
Total	~145,000	21,422
Duplicates removed	~3,500	-

The final dataset is heavily weighted toward ArXiv (about 98%) because ACL got filtered and S2ORC had rate limiting. This isn't ideal for diversity, but ArXiv papers are high quality and cover the topics we care about.

4.2 Preprocessing Results

After preprocessing, the vocabulary has about 30,000 unique tokens. The total token count is around 2.6 million, averaging 122 tokens per abstract. Sentence segmentation now correctly finds 8 to 10 sentences per abstract on average.

The most common words are what you'd expect for NLP/ML papers: 'model' appears over 20,000 times, followed by 'method', 'llm', 'data', 'language', and 'task'. The high frequency of 'llm' shows how much recent research focuses on large language models.

4.3 Embedding Results

Word2Vec training took about 3 minutes and produced a vocabulary of 18,269 words. Quick sanity check on word similarities showed reasonable results, with 'model' being most similar to 'llm', 'classifier', and 'transformer'.

SBERT encoding was fast, about 8 seconds for 10,000 abstracts. SciBERT took longer at around 24 seconds for the same number due to the larger model size. The total embedding storage is about 62 MB across all four embedding types.

I tested the embeddings with a similarity search. Using the query 'scipy.spatial.transform: Differentiable Framework-Agnostic 3D Transformations', the top results were papers about mesh simulation, fine grained control, and articulated objects. All semantically related, which is a good sign the embeddings are working correctly.

5. Summary and Conclusions

5.1 Summary

Week 1 was all about building the foundation that the rest of the project depends on. The main outputs were a clean dataset of 21,422 papers with normalized text, and four types of embeddings ready for downstream tasks. The preprocessing runs about twice as fast as the original version thanks to the single pass optimization.

5.2 What I Learned

A few things stood out from this work. First, data quality varies a lot between sources. Finding out that ACL BibTeX has no abstracts was a surprise, and it would have been a bigger problem if I hadn't validated the data early on. Second, standard libraries don't always work on real world data. The pybtex crashes forced me to write custom parsing code, which ended up being more robust anyway. Third, embedding choice really depends on the task. Higher similarity isn't always better, as we saw with SciBERT causing issues for clustering. Finally, API rate limiting is something you have to plan for. The exponential backoff pattern is pretty standard but I hadn't implemented it myself before.

5.3 Future Work

If we had more time, there are a few things I'd want to add. Full text processing would capture way more information than just abstracts, though it requires PDF parsing which is messy. We could add support for non-English papers with multilingual embeddings. An incremental update system would let us keep adding new papers as they're published. And we could try semantic deduplication using embeddings to catch papers that are the same content but with different titles.

6. Code Percentage Calculation

Using the formula from the assignment:

Percentage = (Internet Lines - Modified Lines) / (Internet Lines + Original Lines) x 100

Table 4: Code Breakdown

Component	Lines	Source
ArXiv API code	40	~28 from arxiv docs
BibTeX parser	50	~18 from Stack Overflow
S2ORC with retry	35	~12 from requests tutorials
Preprocessing	95	~50 from spaCy/NLTK docs
Word2Vec	30	~22 from gensim examples

SBERT/SciBERT	40	~32 from sentence-transformers
Pandas/validation	55	~22 from pandas docs
Custom integration	55	Original

Most of the adapted code came from library documentation and tutorials. The arxiv library, gensim, sentence transformers, spaCy, and NLTK all have good examples that I used as starting points. Stack Overflow helped with the BibTeX edge cases and retry logic patterns. I modified about 50 lines substantially to fit our specific needs.

Total lines from internet: 184

Lines modified significantly: 50

Original lines: 216

Calculation: (184 - 50) / (184 + 216) x 100 = 134 / 400 x 100 = **33.5%**

After accounting for the modifications, I made to adapt the code, the effective percentage is around 30% from internet sources. The other 70% is original code or heavily modified versions.

7. References

[1] ArXiv API Documentation. https://arxiv.org/help/api

[2] ACL Anthology. https://aclanthology.org/

[3] Semantic Scholar API. https://api.semanticscholar.org/

[4] Mikolov et al. (2013). Efficient Estimation of Word Representations in Vector Space. arXiv:1301.3781.

[5] Reimers and Gurevych (2019). Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. EMNLP 2019.

[6] Cohan et al. (2020). SPECTER: Document-level Representation Learning using Citation-informed Transformers. ACL 2020.

[7] spaCy Documentation. https://spacy.io/

[8] Bird, Klein, and Loper (2009). Natural Language Processing with Python. O'Reilly.

[9] Gensim Documentation. https://radimrehurek.com/gensim/

[10] Sentence-Transformers. https://www.sbert.net/