CustomPaperPipeline: Advanced AI for Scientific Paper Generation

by Nigel van der Laan, AI Researcher and Developer, ARQNXS

Introduction

In the rapidly evolving landscape of artificial intelligence and natural language processing, we've developed a cutting-edge AI pipeline that pushes the boundaries of automated scientific paper generation. This innovative system, which we call the CustomPaperPipeline, combines advanced machine learning techniques with natural language processing to create a powerful tool for researchers and academics.

The Core Components

Our CustomPaperPipeline is built on several key components:

1. **Data Collection**: The system begins by fetching relevant scientific papers from arXiv, a popular repository for scientific preprints. This ensures that our model is trained on the latest research in the specified field.

2. **Preprocessing**: Raw text data is cleaned and structured, extracting key sections such as abstracts, introductions, methods, results, and discussions. This step is crucial for maintaining the logical flow and structure of scientific papers.

3. **Custom Transformer Model**: At the heart of our pipeline is a custom-built transformer model. This neural network architecture, inspired by state-of-the-art language models like GPT, is specifically tailored for scientific writing.

4. **Training Process**: The model is trained on the preprocessed scientific papers, learning the patterns, structures, and language specific to academic writing in the chosen field.

5. **Paper Generation**: Once trained, the model can generate new scientific papers based on given prompts, complete with proper section structure and academic language.

6. **Refinement**: As an optional step, the generated papers can be refined using OpenAI's GPT model, adding an extra layer of polish and coherence.

Technical Innovations

Several technical innovations make our CustomPaperPipeline stand out:

Custom Transformer Encoder

We've implemented a custom transformer encoder layer that allows for more efficient processing of scientific text. The key innovation lies in our modification of the standard transformer architecture to better handle the unique characteristics of scientific writing.

The core of our custom transformer encoder is defined as follows:

```
class CustomTransformerEncoderLayer(OriginalTransformerEncoderLayer):
    def __init__(self, d_model, nhead, dim_feedforward=2048,
dropout=0.1, activation=F.relu,
                 layer_norm_eps=1e-5, norm_first=False,
                 device=None, dtype=None):
        super().__init__(
            d_model, nhead, dim_feedforward, dropout, activation,
            layer_norm_eps, batch_first=True, norm_first=norm_first,
            device=device, dtype=dtype
        )
        self.linear1 = nn.Linear(d_model, dim_feedforward)
        self.linear2 = nn.Linear(dim_feedforward, d_model)
```

This custom layer incorporates batch-first processing and allows for more flexible handling of scientific text structures.

Adaptive Tokenization

Our system uses the GPT-2 tokenizer but adapts it for scientific vocabulary, ensuring better representation of domain-specific terms. The tokenization process can be represented by the following function:

$$T(x)=\{t_1,t_2,\ldots,t_n\}$$

Where T is the tokenization function, x is the input text, and $\{t_1,t_2,\ldots,t_n\}$ is the sequence of tokens.

Dynamic Mask Handling

The model intelligently handles attention masks, allowing it to focus on relevant parts of the input during both training and generation. The attention mechanism can be described by the following equation:

$$\text{Attention}(Q,K,V)=\text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

Where Q, K, and V are the query, key, and value matrices respectively, and d_k is the dimension of the key vectors.

Flexible Paper Structure

Our paper formatting algorithm can adapt to various scientific paper structures, ensuring that generated papers follow standard academic conventions. The structure is maintained through a series of regex-based extractors:

```
def extract_section(text: str, section_name: str) -> str:
    pattern = f"{section_name}[:.\n](.*?)(?:\n\n|\n(?=[0-9]+\.?\s+[A-Z]))"
    match = re.search(pattern, text, re.DOTALL | re.IGNORECASE)
    return match.group(1).strip() if match else ""
```

Model Architecture

The core of our system is the `TransformerModel` class, which implements a custom transformer architecture:

```
class TransformerModel(nn.Module):
    def __init__(self, vocab_size, hidden_size, num_layers, num_heads, dropout=0.1, device=None):
        super(TransformerModel, self).__init__()
        self.device = device
        self.embedding = nn.Embedding(vocab_size, hidden_size)
        encoder_layers = CustomTransformerEncoderLayer(hidden_size, num_heads, dropout=dropout)
        self.transformer_encoder = TransformerEncoder(encoder_layers, num_layers)
        self.fc = nn.Linear(hidden_size, vocab_size)

    def forward(self, x, mask=None):
        embedded = self.embedding(x)
        if mask is not None:
            mask = mask.bool()
            mask = ~mask
        output = self.transformer_encoder(embedded, src_key_padding_mask=mask)
        output = self.fc(output)
        return output
```

The model uses an embedding layer, followed by multiple transformer encoder layers, and a final linear layer for output generation.

Training Process

The training process involves minimizing the cross-entropy loss between the model's predictions and the actual tokens in the scientific papers. The loss function can be expressed as:

$$L=-\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{V} y_{ij}\log\left(p_{ij}\right)$$

Where N is the number of samples, V is the vocabulary size, y_{ij} is the true distribution, and p_{ij} is the predicted probability distribution.

Paper Generation

The paper generation process uses a combination of top-k and top-p (nucleus) sampling to produce diverse and coherent text. The sampling process can be described by the following algorithm:

1. Compute the probability distribution over the vocabulary: $P\left(x_t \vee x_{¿t}\right)$
2. Sort the probabilities in descending order
3. Keep top-k tokens or tokens that cumulatively exceed probability p
4. Renormalize the probabilities
5. Sample from the reduced vocabulary

This process is implemented in the `top_k_top_p_filtering` function:

```
def top_k_top_p_filtering(logits, top_k=0, top_p=1.0, filter_value=-
float('Inf')):
    top_k = min(top_k, logits.size(-1))
    if top_k > 0:
        indices_to_remove = logits < torch.topk(logits, top_k)[0][...,
-1, None]
        logits[indices_to_remove] = filter_value
    if top_p < 1.0:
        sorted_logits, sorted_indices = torch.sort(logits,
descending=True)
        cumulative_probs = torch.cumsum(F.softmax(sorted_logits, dim=-
1), dim=-1)
        sorted_indices_to_remove = cumulative_probs > top_p
        sorted_indices_to_remove[..., 1:] =
sorted_indices_to_remove[..., :-1].clone()
        sorted_indices_to_remove[..., 0] = 0
        indices_to_remove = sorted_indices[sorted_indices_to_remove]
        logits[indices_to_remove] = filter_value
    return logits
```

Potential Applications

The CustomPaperPipeline has numerous potential applications in academia and research:

1. **Literature Review Assistance**: Researchers can use the system to generate initial drafts of literature reviews, saving time in the early stages of research.
2. **Hypothesis Generation**: By analyzing patterns in existing research, the system could suggest novel hypotheses for further investigation.

3. **Writing Support**: The pipeline can assist researchers in overcoming writer's block by generating initial drafts or suggesting content for specific sections.
4. **Educational Tool**: Students can use the system to understand the structure and language of scientific papers in their field of study.

Ethical Considerations

While the CustomPaperPipeline represents a significant advancement in AI-assisted scientific writing, it's crucial to approach its use ethically. The system is designed as a tool to augment human researchers, not replace them. All generated content should be thoroughly reviewed, fact-checked, and appropriately edited by human experts before any form of publication or submission.

Future Directions

As we continue to develop and refine the CustomPaperPipeline, we're exploring several exciting avenues for improvement:

- Integration with citation databases for automatic reference generation
- Expansion to cover a wider range of scientific disciplines
- Implementation of more sophisticated coherence and factual consistency checks
- Development of a user-friendly interface for easier adoption by researchers

Conclusion

The CustomPaperPipeline represents a significant step forward in the application of AI to scientific research and writing. By automating certain aspects of the paper writing process, we aim to free up researchers' time for more creative and analytical tasks. As we continue to refine and expand this technology, we're excited about its potential to accelerate scientific discovery and enhance the productivity of researchers worldwide.

We welcome collaborations and feedback from the scientific community as we work towards shaping the future of AI-assisted academic writing.

For more information or collaboration opportunities, please contact:

Nigel van der Laan AI Researcher and Developer ARQNXS

Code

```
import os
import re
import json
import time
import logging
import requests
import lxml.etree as ET
```

```python
from typing import List, Dict, Union
from tqdm import tqdm
import nltk
from nltk.tokenize import word_tokenize, sent_tokenize
from nltk.corpus import stopwords
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.nn import TransformerEncoder
from torch.nn.modules.transformer import TransformerEncoderLayer as
OriginalTransformerEncoderLayer
from torch.utils.data import Dataset, DataLoader
import torch.optim as optim
import openai
from multiprocessing import Pool
from transformers import AutoTokenizer, AutoModelForCausalLM

# Set up logging
logging.basicConfig(level=logging.INFO, format='%(asctime)s - %
(levelname)s - %(message)s')

nltk.download('punkt', quiet=True)
nltk.download('stopwords', quiet=True)

openai_key = 'sk-proj-
xSYMrx4bYZr59ROqvcHpT3BlbkFJQyjjuhUfOmxdwJsPWJZu'

class CustomTransformerEncoderLayer(OriginalTransformerEncoderLayer):
    def __init__(self, d_model, nhead, dim_feedforward=2048,
dropout=0.1, activation=F.relu,
                 layer_norm_eps=1e-5, norm_first=False,
                 device=None, dtype=None):
        super().__init__(
            d_model, nhead, dim_feedforward, dropout, activation,
            layer_norm_eps, batch_first=True, norm_first=norm_first,
            device=device, dtype=dtype
        )
        self.linear1 = nn.Linear(d_model, dim_feedforward)
        self.linear2 = nn.Linear(dim_feedforward, d_model)


class PaperDataset(Dataset):
    def __init__(self, data_file, tokenizer, max_length):
        self.data = self.load_data(data_file)
        self.tokenizer = tokenizer
        self.max_length = max_length

    def load_data(self, data_file):
        with open(data_file, 'r', encoding='utf-8') as file:
            data = [json.loads(line) for line in file]
        return data

    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        paper = self.data[idx]
        # Convert all values to strings, joining lists if necessary
        paper_text = ' '.join(str(value) if isinstance(value, str)
else ' '.join(value) for value in paper.values())
        encoded = self.tokenizer(
            paper_text,
            max_length=self.max_length,
            padding='max_length',
            truncation=True,
            return_tensors='pt'
        )
        return encoded['input_ids'].squeeze(),
encoded['attention_mask'].squeeze()

class TransformerModel(nn.Module):
    def __init__(self, vocab_size, hidden_size, num_layers, num_heads,
dropout=0.1, device=None):
        super(TransformerModel, self).__init__()
        self.device = device
        self.embedding = nn.Embedding(vocab_size, hidden_size)
        encoder_layers = CustomTransformerEncoderLayer(hidden_size,
num_heads, dropout=dropout)
        self.transformer_encoder = TransformerEncoder(encoder_layers,
num_layers)
        self.fc = nn.Linear(hidden_size, vocab_size)

    def forward(self, x, mask=None):
        embedded = self.embedding(x)
        if mask is not None:
            # Convert mask to boolean
            mask = mask.bool()
            # Invert the mask as per PyTorch convention
            mask = ~mask
        output = self.transformer_encoder(embedded,
src_key_padding_mask=mask)
        output = self.fc(output)
        return output

    def generate(self, input_ids, attention_mask=None, max_length=100,
num_return_sequences=1, temperature=1.0, no_repeat_ngram_size=2,
pad_token_id=None):
        batch_size = input_ids.shape[0]
        current_length = input_ids.shape[1]
        for _ in range(max_length - current_length):
            outputs = self(input_ids, mask=attention_mask)
            next_token_logits = outputs[:, -1, :]
            next_token_logits = next_token_logits / temperature
            filtered_logits = top_k_top_p_filtering(next_token_logits,
top_k=50, top_p=1.0)
            next_token = torch.multinomial(F.softmax(filtered_logits,
dim=-1), num_samples=1)
            input_ids = torch.cat([input_ids, next_token], dim=-1)

            if attention_mask is not None:
                attention_mask = torch.cat([attention_mask,
attention_mask.new_ones((batch_size, 1))], dim=-1)

            if next_token.item() == pad_token_id:
                break

        return input_ids

def top_k_top_p_filtering(logits, top_k=0, top_p=1.0, filter_value=-
float('Inf')):
    top_k = min(top_k, logits.size(-1))
    if top_k > 0:
        indices_to_remove = logits < torch.topk(logits, top_k)[0][...,
-1, None]
        logits[indices_to_remove] = filter_value
    if top_p < 1.0:
        sorted_logits, sorted_indices = torch.sort(logits,
descending=True)
        cumulative_probs = torch.cumsum(F.softmax(sorted_logits, dim=-
1), dim=-1)
        sorted_indices_to_remove = cumulative_probs > top_p
        sorted_indices_to_remove[..., 1:] =
sorted_indices_to_remove[..., :-1].clone()
        sorted_indices_to_remove[..., 0] = 0
        indices_to_remove = sorted_indices[sorted_indices_to_remove]
        logits[indices_to_remove] = filter_value
    return logits

class CustomPaperPipeline:
    def __init__(self, vocab_size, hidden_size, num_layers, num_heads,
dropout=0.1, openai_api_key=None, cache_dir='cache'):
        self.device = torch.device('cuda' if torch.cuda.is_available()
else 'cpu')
        self.tokenizer = AutoTokenizer.from_pretrained("gpt2")

        # Set the padding token
        if self.tokenizer.pad_token is None:
            self.tokenizer.pad_token = self.tokenizer.eos_token
            self.tokenizer.pad_token_id = self.tokenizer.eos_token_id
        self.model = TransformerModel(len(self.tokenizer),
hidden_size, num_layers, num_heads, dropout,
device=self.device).to(self.device)
        self.cache_dir = cache_dir

        if openai_api_key:
            openai.api_key = openai_api_key

    def fetch_arxiv_paper(self, query: str, start: int, max_results:
int = 100) -> str:
        base_url = 'http://export.arxiv.org/api/query?'
        search_query =
f'search_query={query}&start={start}&max_results={max_results}'
        response = requests.get(base_url + search_query)
        return response.text

    def save_papers(self, query: str, dir_path: str, max_results: int
= 100, max_papers: int = 10000, num_processes: int = 4):
        if not os.path.exists(dir_path):
            os.makedirs(dir_path)

        with Pool(processes=num_processes) as pool:
            start_values = list(range(0, max_papers, max_results))
            results = pool.starmap(self.fetch_arxiv_paper, [(query,
start) for start in start_values])

        downloaded_papers = 0
        for i, data in enumerate(results):
            if '<entry>' not in data:
                logging.info(f"No more papers found after downloading
{downloaded_papers} papers.")
                break
            with open(f"{dir_path}/papers_{i*max_results}.xml", 'w',
encoding='utf-8') as f:
                f.write(data)
            downloaded_papers += max_results

        logging.info(f"Downloaded {downloaded_papers} papers.")

    def preprocess_text(self, text: str) -> str:
        text = re.sub(r'\s+', ' ', text)
        text = re.sub(r'\[.*?\]', '', text)
        text = re.sub(r'https?://\S+|www\.\S+', '', text)
        text = re.sub(r'<.*?>+', '', text)
        text = re.sub(r'[^\w\s]', '', text)
        tokens = word_tokenize(text)
        stop_words = set(stopwords.words('english'))
        tokens = [word.lower() for word in tokens if word.lower() not
in stop_words]
```

```
        return ' '.join(tokens)

    def extract_title(self, text: str) -> str:
        lines = text.split('\n')
        for line in lines:
            if line.strip() and not re.match(r'^(Abstract|Authors?|
Keywords):', line, re.IGNORECASE):
                return line.strip()
        return ""

    def extract_abstract(self, text: str) -> str:
        abstract_pattern = r'Abstract[:.\n](.*?)(?:\n\n|\n(?=[0-
9]+\.?\s+[A-Z]))'
        abstract_match = re.search(abstract_pattern, text, re.DOTALL |
re.IGNORECASE)
        if abstract_match:
            return abstract_match.group(1).strip()
        return ""

    def extract_introduction(self, text: str) -> str:
        intro_pattern = r'(?:Introduction|Background)[:.\n](.*?)(?:\n\
n|\n(?=[0-9]+\.?\s+[A-Z]))'
        intro_match = re.search(intro_pattern, text, re.DOTALL |
re.IGNORECASE)
        if intro_match:
            return intro_match.group(1).strip()
        return ""

    def extract_methods(self, text: str) -> str:
        methods_pattern = r'(?:Methods|Methodology|Materials and
Methods)[:.\n](.*?)(?:\n\n|\n(?=[0-9]+\.?\s+[A-Z]))'
        methods_match = re.search(methods_pattern, text, re.DOTALL |
re.IGNORECASE)
        if methods_match:
            return methods_match.group(1).strip()
        return ""

    def extract_results(self, text: str) -> str:
        results_pattern = r'(?:Results|Findings)[:.\n](.*?)(?:\n\n|\
n(?=[0-9]+\.?\s+[A-Z]))'
        results_match = re.search(results_pattern, text, re.DOTALL |
re.IGNORECASE)
        if results_match:
            return results_match.group(1).strip()
        return ""

    def extract_discussion(self, text: str) -> str:
        discussion_pattern = r'(?:Discussion|Conclusion)[:.\n](.*?)
(?:\n\n|\n(?=[0-9]+\.?\s+[A-Z]))'
        discussion_match = re.search(discussion_pattern, text,
re.DOTALL | re.IGNORECASE)
        if discussion_match:
            return discussion_match.group(1).strip()
        return ""

    def extract_references(self, text: str) -> List[str]:
        ref_pattern = r'(?:References|Bibliography)(.*?)(?:\n\n|\Z)'
        ref_match = re.search(ref_pattern, text, re.DOTALL |
re.IGNORECASE)
        if ref_match:
            ref_text = ref_match.group(1)
            return [ref.strip() for ref in re.split(r'\n\s*\n|\[[0-
9]+\]', ref_text) if ref.strip()]
        return []

    def process_paper(self, raw_text: str) -> Dict[str, Union[str,
List[str]]]:
        paper = {
            'title': self.extract_title(raw_text),
            'abstract': self.extract_abstract(raw_text),
            'introduction': self.extract_introduction(raw_text),
            'methods': self.extract_methods(raw_text),
            'results': self.extract_results(raw_text),
            'discussion': self.extract_discussion(raw_text),
            'references': self.extract_references(raw_text)
        }

        for key in paper:
            if isinstance(paper[key], str):
                paper[key] = self.preprocess_text(paper[key])
            elif isinstance(paper[key], list):
                paper[key] = ' '.join([self.preprocess_text(item) for
item in paper[key]])

        return paper

    def extract_and_preprocess_papers(self, dir_path: str,
output_file: str = 'preprocessed_papers.txt'):
        cache_file = os.path.join(self.cache_dir, output_file)
        if os.path.exists(cache_file):
            logging.info(f"Using cached preprocessed papers from
{cache_file}")
            return cache_file

        with open(cache_file, 'w', encoding='utf-8') as out_file:
            for filename in tqdm(os.listdir(dir_path),
desc="Preprocessing papers"):
                if filename.endswith('.xml'):
                    try:
                        with open(os.path.join(dir_path, filename),
'r', encoding='utf-8') as file:
                            tree = ET.parse(file)
                            root = tree.getroot()
                            for entry in
root.findall('{http://www.w3.org/2005/Atom}entry'):
                                title =
entry.find('{http://www.w3.org/2005/Atom}title').text
                                summary =
entry.find('{http://www.w3.org/2005/Atom}summary').text
                                full_text = title + '\n\n' + summary
                                processed_paper =
self.process_paper(full_text)

out_file.write(json.dumps(processed_paper) + '\n')
                    except Exception as e:
                        logging.error(f"Error processing file
{filename}: {e}")
        return cache_file

    def train_model(self, train_file: str, output_dir: str =
'./custom_model',
                    num_epochs: int = 10, batch_size: int = 8,
                    learning_rate: float = 1e-4, max_length: int =
512):
        train_dataset = PaperDataset(train_file, self.tokenizer,
max_length)
        train_loader = DataLoader(train_dataset,
batch_size=batch_size, shuffle=True)

        criterion =
nn.CrossEntropyLoss(ignore_index=self.tokenizer.pad_token_id)
        optimizer = optim.Adam(self.model.parameters(),
lr=learning_rate)

        self.model.train()
        for epoch in range(num_epochs):
            epoch_loss = 0
            for batch in tqdm(train_loader, desc=f"Epoch {epoch+1}"):
                inputs, masks = batch
                inputs = inputs.to(self.device)
                masks = masks.to(self.device)

                optimizer.zero_grad()
                outputs = self.model(inputs, mask=masks)
                loss = criterion(outputs.view(-1, outputs.size(-1)),
inputs.view(-1))
                loss.backward()
                optimizer.step()

                epoch_loss += loss.item()
            logging.info(f"Epoch {epoch+1} - Loss:
{epoch_loss/len(train_loader)}")

        os.makedirs(output_dir, exist_ok=True)
        torch.save(self.model.state_dict(), os.path.join(output_dir,
'model_weights.pth'))
        logging.info(f"Model saved to {output_dir}")


    def generate_scientific_paper(self, prompt: str, max_length: int =
2000,
                                  num_return_sequences: int = 1,
temperature: float = 0.7) -> str:
        try:
            inputs = self.tokenizer(prompt, return_tensors="pt",
padding=True).to(self.device)
            outputs = self.model.generate(
                **inputs,
                max_length=max_length,
                num_return_sequences=num_return_sequences,
                temperature=temperature,
                no_repeat_ngram_size=2,
                pad_token_id=self.tokenizer.pad_token_id
            )
            return self.tokenizer.decode(outputs[0],
skip_special_tokens=True)
        except Exception as e:
            logging.error(f"Error generating scientific paper: {e}")
            return None

    def generate_sample_text(self, prompt: str, max_length: int = 100)
-> str:
        return self.generate_scientific_paper(prompt, max_length,
num_return_sequences=1, temperature=0.7)

    def refine_with_openai(self, draft: str) -> str:
        if not openai.api_key:
            raise ValueError("OpenAI API key is not set")

        try:
            response = openai.ChatCompletion.create(
                model="gpt-3.5-turbo",
                messages=[
                    {"role": "system", "content": "You are a
scientific paper editor. Refine and improve the following draft:"},
                    {"role": "user", "content": draft}
                ],
                max_tokens=1000,
                temperature=0.7
            )
            return response.choices[0].message['content']
        except Exception as e:
            logging.error(f"Error refining with OpenAI: {e}")
            return draft  # Return original draft if refinement fails

    def format_paper(self, text: str) -> str:
        sections = ['Abstract', 'Introduction', 'Methods', 'Results',
'Discussion', 'Conclusion']
        formatted_paper = ""
        for section in sections:
            pattern = f"{section}:?(.*?)(?
={sections[sections.index(section)+1]}|$)"
            match = re.search(pattern, text, re.DOTALL |
re.IGNORECASE)
            if match:
                formatted_paper += f"\n\n{section}\
n{match.group(1).strip()}"
        return formatted_paper.strip()

    def run_full_pipeline(self, query: str, num_papers: int = 10000,
train_epochs: int = 10, dry_run: bool = False) -> Union[str, None]:
        try:
            # Step 1: Data Collection
            logging.info("Starting paper collection...")
            papers_dir = os.path.join(self.cache_dir, query.replace("
", "_") + "_papers")
            if not os.path.exists(papers_dir):
                self.save_papers(query, papers_dir,
max_papers=num_papers)
            else:
                logging.info(f"Using cached papers from {papers_dir}")

            # Step 2: Data Preprocessing
            logging.info("Starting data preprocessing...")
            train_file =
self.extract_and_preprocess_papers(papers_dir)

            # Step 3: Model Training
            logging.info("Starting model training...")
            self.train_model(train_file, num_epochs=train_epochs)

            if dry_run:
                logging.info("Dry run: Generating sample text...")
                sample_text = self.generate_sample_text(f"Write about
{query}")
                logging.info(f"Sample generated text: {sample_text}")
                return sample_text

            # Step 4: Generate Paper
            logging.info("Generating full paper...")
            prompt = f"""Write a comprehensive scientific paper about {query}.
            Include the following sections:
            1. Abstract
            2. Introduction
            3. Methods
            4. Results
            5. Discussion
            6. Conclusion
            Ensure each section is well-developed and follows scientific writing conventions."""

            paper = self.generate_scientific_paper(prompt, max_length=3000)
            if paper:
                try:
                    formatted_paper = self.format_paper(paper)
                except Exception as e:
                    logging.error(f"Error formatting paper: {e}")
                    return None
            else:
                logging.error("Failed to generate paper.")
                return None

            # Step 5: Refine Paper (if OpenAI API key is provided)
            if openai.api_key:
                try:
                    logging.info("Refining paper with OpenAI...")
                    formatted_paper = self.refine_with_openai(formatted_paper)
                except Exception as e:
                    logging.error(f"Error refining with OpenAI: {e}")
                    logging.info("Continuing with unrefined paper.")

            return formatted_paper
        except Exception as e:
            logging.error(f"Error in pipeline: {e}")
            return None

    def save_model(self, path: str) -> None:
        os.makedirs(path, exist_ok=True)
        torch.save(self.model.state_dict(), os.path.join(path, 'model_weights.pth'))
        logging.info(f"Model weights saved to {path}")

    def load_model(self, path: str) -> None:
        self.model.load_state_dict(torch.load(os.path.join(path, 'model_weights.pth')))
        self.model.to(self.device)
        logging.info(f"Model weights loaded from {path}")
"""
# Usage example:
if __name__ == "__main__":
    vocab_size = 10000
    hidden_size = 256
    num_layers = 4
    num_heads = 8
    dropout = 0.1

    pipeline = CustomPaperPipeline(vocab_size, hidden_size,
num_layers, num_heads, dropout, openai_api_key=openai_key)

    # Dry run to test the pipeline
    sample_text = pipeline.run_full_pipeline('machine learning in
healthcare', num_papers=1000, train_epochs=1, dry_run=True)
    print("Sample text:", sample_text)

    # Full run
    paper = pipeline.run_full_pipeline('machine learning in
healthcare', num_papers=10000, train_epochs=10)
    if paper:
        print("Generated paper:")
        print(paper)
    else:
        print("Failed to generate paper.")

    # Save the trained model
    pipeline.save_model('./custom_trained_model')

    # Load a previously trained model
    pipeline.load_model('./custom_trained_model')

    # Generate a new paper using the loaded model
    new_paper = pipeline.generate_scientific_paper("The future of AI
in healthcare")
    if new_paper:
        print("New generated paper:")
        print(new_paper)
    else:
        print("Failed to generate new paper.")
"""
```