# Evaluating scientific paper summarization methods using multiple metrics

**Neaga Matei, Cordoș Cristian, Ciobanu Alexandru**
University POLITEHNICA of Bucharest
mateineaga@outlook.com, icordos@gmail.com, alexandru.ciobanu98@stud.trans.upb.ro

## Abstract

This paper evaluates the effectiveness of Transformer-based models for scientific paper summarization, with a focus on PEGASUS-XSum. The study uses domain-specific datasets from eLife and PLOS to generate abstractive summaries and compares them against reference summaries using metrics such as ROUGE, BLEU, and Named Entity Recognition. The preprocessing pipeline includes sliding window segmentation to accommodate the model's input size limitations, followed by concatenation of window-level summaries.

## 1 Introduction

The rapid increase in scientific publications has created challenges for researchers in extracting relevant information efficiently [1]. Traditional tools, such as abstracts, provide limited insight, and manual summarization is time-consuming. This motivates the development of automatic summarization techniques to generate concise and comprehensive summaries of scientific literature [2].

Recent advancements in natural language processing, particularly Transformer-based models, have demonstrated strong performance in text summarization tasks [3]. Models such as PEGASUS [4], BART [5], and BigBird [6] represent significant progress, leveraging abstractive summarization to generate fluent and meaningful summaries.

### 1.1 State of the Art

Transformer-based models have advanced the field of abstractive summarization by addressing key challenges:

- PEGASUS: Optimized for abstractive summarization tasks through gap-sentence masking pre-training, which enables the generation of highly coherent summaries [4].
- BART: A denoising autoencoder model capable of reconstructing corrupted text, providing strong performance for summarization and translation [5].
- BigBird: Extends Transformer architectures for processing long-form documents efficiently, making it particularly suitable for scientific texts [6].

Despite these advancements, summarizing long-form scientific articles remains a challenge due to the structural complexity of such texts, the need to preserve domain-specific terminology, and inherent input size limitations in Transformer-based models [6].

This paper evaluates the PEGASUS-XSum model on domain-specific datasets from eLife and PLOS [7, 8]. By implementing a sliding window preprocessing technique and utilizing advanced evaluation metrics, this work aims to highlight the strengths and limitations of abstractive summarization for scientific literature.

The remainder of this paper is structured as follows: Section 2 provides a detailed overview of related work; Section 3 describes the preprocessing pipeline, summarization model, datasets, and evaluation metrics; Section 4 presents and analyzes the evaluation results.

## 2 Related Work

The application of Transformer-based models has led to significant advancements in the field of scientific paper summarization, where researchers have evaluated various methods using different datasets and performance metrics. This section outlines the work that has been done in evaluating the performance of these models in summarizing scientific literature.

### 2.1 Methods for Evaluating Transformer-Based Summarization Models

Recent work highlights how Transformer models, such as PEGASUS, BART, and BigBird, are specifically optimized for summarizing complex scientific texts. Models like PEGASUS, which utilize gap-sentence pretraining, and BigBird, which employs sparse attention mechanisms, have been shown to generate more coherent abstractive summaries for long-form scientific articles [4, 6]. These approaches help reconstruct sentences rather than just extracting parts of the text, making the summaries more human-readable and relevant.

In terms of evaluation, commonly used metrics like ROUGE and BERTScore play a central role in measuring the performance of these models. ROUGE focuses on n-gram overlap [9], while BERTScore evaluates the semantic similarity between the generated summaries and the reference summaries [10]. These metrics are widely recognized for assessing the quality of summaries generated by Transformer-based models.

### 2.2 Recent Advances in Scientific Summarization

Recent studies have extended transformer models to handle the unique challenges posed by scientific texts, such as their length, complexity, and terminology density. For example, “An Efficient Abstractive Summarization of Research Articles Using PEGASUS Model” (2023) [11] compared PEGASUS with other state-of-the-art models like BART and T5. The study demonstrated PEGASUS’s superior performance in maintaining the syntactic and semantic structure of scientific content.

Similarly, "Summaformers @ LaySumm 20, LongSumm 20" (2021) [12] explored summarization techniques tailored to scientific papers, focusing on generating lay summaries and detailed summaries. By leveraging the structural organization of scientific texts, this approach highlighted the potential of domain-specific adaptations to improve summarization quality.

### 2.3 Hybrid and Specialized Approaches

Hybrid methods that combine extractive and abstractive techniques have also shown promise. "SKT5SciSumm – A Hybrid Generative Approach" (2024) [13] utilized sentence embeddings and the T5 model to perform multi-document summarization, achieving state-of-the-art results on datasets like Multi-XScience. This approach demonstrates the value of integrating extractive techniques to ensure comprehensive coverage of key information while maintaining abstractive fluency.

In addition to general summarization, specialized applications have emerged. "Summaries as Captions: Generating Figure Captions for Scientific Documents" (2023) [14] treated figure captioning as a summarization task, applying PEGASUS to summarize figure-referencing paragraphs. This method showcased how abstractive models could be fine-tuned for niche scientific tasks.

### 2.4 Datasets Used for Transformer Model Evaluation

This project uses the eLife and PLOS datasets, which consist of scientific articles across diverse disciplines [7, 8]. Both datasets are structured into sections, such as introductions and results, with corresponding reference summaries like abstracts. These datasets are preprocessed to split long texts into manageable windows for input into the PEGASUS model. Their domain-specific content and diversity make them ideal for evaluating the summarization pipeline.

## 3 Method and Experiment Details

### 3.1 Preprocessing

The preprocessing phase, see Appendix 1, ensures the datasets are structured and optimized for input into the Pegasus model and subsequent evaluation. The following steps were undertaken:

**Initial Preprocessing:** The raw datasets were cleaned to retain only the essential components:

- article_id: A unique identifier for each article.
- sections: The main body of the article, divided into logical sections.
- summary: The reference (gold-standard) summary provided by the dataset.

**Sliding Window Implementation:** To address the input size constraints of the Pegasus model, all article sections were concatenated and divided into overlapping windows. This ensures:

- Window Size: Each window contains 280 tokens.
- Step Size: Windows overlap by 100 tokens to preserve context across splits.

**Text Tokenization:** The Pegasus tokenizer was used to tokenize the preprocessed text into input IDs and attention masks compatible with the model. Tokenization ensures the data is correctly formatted for abstractive summarization.

**JSON Formatting:** The preprocessed data was saved in a structured JSON format, containing Article ID, Sections, Summary. By preprocessing the datasets in this manner, the pipeline ensures compatibility with the Pegasus model while addressing the challenges posed by long-form scientific content.

### 3.2 Summarization Pipeline

As described in Section 3.1, articles are split into overlapping windows to accommodate PEGASUS-XSum's input size limitations. Each window is summarized independently using the following process:

1. The PEGASUS tokenizer converts input text into token IDs and attention masks.
2. The PEGASUS-XSum model generates abstractive summaries for each window using beam search.
3. The resulting summaries are concatenated to produce a coherent final summary for the entire article.

This pipeline ensures that even long-form scientific texts can be summarized efficiently while retaining contextual coherence across windows.

### 3.3 Model

The Pegasus-XSum model, developed by Google, is a transformer-based architecture pre-trained specifically for abstractive summarization tasks. Unlike extractive models that rely on copying text verbatim, Pegasus generates summaries by understanding the context and rephrasing content in a coherent and concise manner.

For this project, the Pegasus-XSum variant was selected due to its ability to generate abstractive summaries for a wide range of domains. Key features and configurations of the model include:

- Pre-trained Model: The model is fine-tuned on the XSum dataset, which focuses on generating single-sentence summaries for news articles.
- Tokenizer: The Pegasus tokenizer is used to preprocess input text, converting it into token IDs and attention masks for the model.
- Hyperparameters:

- Beam search is employed during generation to improve summary quality.
- Maximum token length for generated summaries is set to ensure coherence while adhering to scientific norms.

Due to the long-form nature of scientific articles in datasets like eLife and PLOS, the text is processed in overlapping windows (280 tokens with a step size of 100 tokens). Each window is summarized independently by Pegasus, and the generated summaries are concatenated to form the final output.

By leveraging the Pegasus-XSum model, the project aims to achieve high-quality abstractive summaries that capture the essence of scientific texts while maintaining readability and coherence.

### 3.4 Training and Fine-Tuning

The training and fine-tuning process, see Appendix 2, involves leveraging the Pegasus model, pre-trained for summarization tasks, and adapting it to domain-specific data using fine-tuning. This ensures that the model generates high-quality summaries tailored to the data's characteristics. The PEGASUS-XSum model was fine-tuned using the following training configuration:

- Optimizer: The AdamW optimizer was employed with a learning rate of $5 \times 10^{-5}$ and a weight decay of $0.001$. This optimizer is well-suited for Transformer-based models due to its ability to adapt learning rates during training and mitigate overfitting through weight decay.
- Batch Size and Epochs: The model was fine-tuned with a batch size of 4 and trained over 6 epochs. The relatively small batch size accommodates the large memory requirements of the PEGASUS-XSum model on GPU.
- Training Checkpoints: A mechanism for saving checkpoints during training was implemented to handle time constraints and ensure training progress is preserved. Checkpoints capture the model's state, optimizer's parameters, and the number of processed articles at the time of saving. For this project, checkpoints were saved before the training process exceeded a 12-hour runtime limit.

The checkpointing mechanism was implemented as follows:

```
checkpoint = {
    'model_state_dict': model.state_dict(),
    'optimizer_state_dict': optimizer.state_dict(),
    'processed_articles': articles_processed,
}
torch.save(checkpoint, "model_checkpoint_final.pth")
```

**Loss Computation and Backpropagation:** The model's performance was optimized using the loss computed from cross-entropy between generated summaries and gold-standard summaries. Gradients were computed and updated at each training step.

Fine-tuning introduced some challenges such as:

- Memory Constraints: The high-dimensional embeddings and large parameter space required efficient memory management, necessitating the use of overlapping input windows during training.
- Epoch Checkpoints: Saving periodic checkpoints allowed training to resume seamlessly in case of interruptions, reducing the risk of data or computation loss.

### 3.5 Evaluation Process

To assess the quality of the generated summaries, the following evaluation metrics were applied:

ROUGE (Recall-Oriented Understudy for Gisting Evaluation) measures the overlap between n-grams in the generated summary and the reference summary. The project evaluates:

- ROUGE-1: Overlap of unigrams (single words).

- ROUGE-2: Overlap of bigrams (two-word sequences).
- ROUGE-L: Overlap of the longest common subsequence (LCS), capturing fluency and coherence.

BLEU (Bilingual Evaluation Understudy) evaluates n-gram precision between the generated summary and reference summary. A smoothing function is applied to prevent penalties for shorter generated summaries.

Named Entity Recognition (NER) a BERT-based pipeline was implemented to assess the overlap of named entities (e.g. people, organizations, locations) between the generated and reference summaries. Two levels of analysis are performed:

- Simple NER: Measures the overlap in entity types.
- Detailed NER: Tracks exact matches, partial matches, and type errors for a more granular analysis.

**Overall Comparison:** Summaries generated by the Pegasus model are compared to the gold-standard summaries provided in the datasets. Results are saved in a structured JSON format for further analysis:

```
{
  "article_id": "elife-12345",
  "generated_summary": "Summary generated by Pegasus...",
  "reference_summary": "Original gold-standard summary...",
  "metrics": {
    "rouge1": 0.45,
    "rouge2": 0.32,
    "rougeL": 0.40,
    "bleu": 0.38,
    "ner_simple": {"common_entities": 4, "entity_overlap": 0.67},
    "ner_detailed": {"perfect_matches": 3, "type_errors": 1}
  }
}
```

Evaluation results are visualized using Python scripts:

- evaluation_metrics.png: Plots showing trends in ROUGE and BLEU scores across multiple articles.
- ner_detailed_metrics.png: Highlights NER overlap, type errors, and other entity-specific metrics.

By applying these metrics and visualizations, the evaluation process provides both quantitative and qualitative insights into the performance of the summarization pipeline.

### 3.6 Implementing RLHF (Reinforcement Learning with Human Feedback)

The implementation of Reinforcement Learning with Human Feedback (RLHF), see Appendix 3, in our project follows a structured pipeline aimed at optimizing the summarization model for higher-quality outputs. This process integrates supervised learning, reward modeling, and reinforcement learning to improve the alignment of the model's outputs with human preferences.

**Training the Initial Summarization Model** The process begins by training a summarization model using a labeled dataset with input texts and corresponding high-quality reference summaries. This step uses supervised fine-tuning to optimize the model's ability to generate coherent and contextually accurate summaries.

A pre-trained model (distilroberta), which is extended to predict scalar rewards, is fine-tuned on a summarization dataset. The model's performance on the validation set is evaluated and saved for later comparison with the RLHF-optimized model.

**Creating a Reward Dataset** For each article the dataset, summaries are generated using different configurations are generated: long format or short format. Summaries are ranked manually by humans, assuming that the longer one was worse.

The reference summary from the dataset was considered the gold standard.Higher ROUGE scores indicated summaries that closely resembled the reference and were ranked higher.

**Training the Reward Model** For each article the dataset, summaries are generated using different configurations are generated: long format or short format. The reward model requires a dataset that contains pairs of "chosen" and "rejected" summaries for the same input text. These pairs are formatted to allow the model to distinguish between higher-quality summaries and less-preferred ones.

The reference summary from the dataset was considered the gold standard. Higher ROUGE scores indicated summaries that closely resembled the reference and were ranked higher.

**Fine-Tuning with RLHF** After training the reward model, the base summarization model is fine-tuned using reinforcement learning guided by the reward model. The goal is to maximize the reward score assigned by the reward model to the generated summaries.

Proximal Policy Optimization (PPO), see Appendix 4, is employed for fine-tuning the base model. During training, the base model generates summaries for input texts, and the reward model evaluates them. The reward scores are used as feedback to adjust the base model's parameters, optimizing it to generate higher-quality summaries. The PPO loss function combines the policy loss, value loss, and entropy loss to balance exploration and exploitation.

# 4 Results

## 4.1 Overview of the Evaluation

The evaluation of the proposed summarization pipeline was conducted on a dataset of 240 scientific articles, sourced from the preprocessed `test.json`. This dataset includes gold-standard reference summaries for each article, allowing for direct comparison with the generated summaries.

The evaluation aimed to assess both the linguistic quality and the semantic retention capabilities of the model-generated summaries.

- Text Similarity Metrics: ROUGE and BLEU scores were used to evaluate lexical and grammatical overlap between the generated and reference summaries.
- Named Entity Retention: NER-based metrics measured the ability of the model to preserve critical entities, including domain-specific terms, across the summarization process.

This dual evaluation approach provides a comprehensive assessment of the Pegasus-XSum model's performance, focusing on its ability to generate coherent and informative summaries while maintaining scientific accuracy.

## 4.2 Quantitative Metrics and Analysis

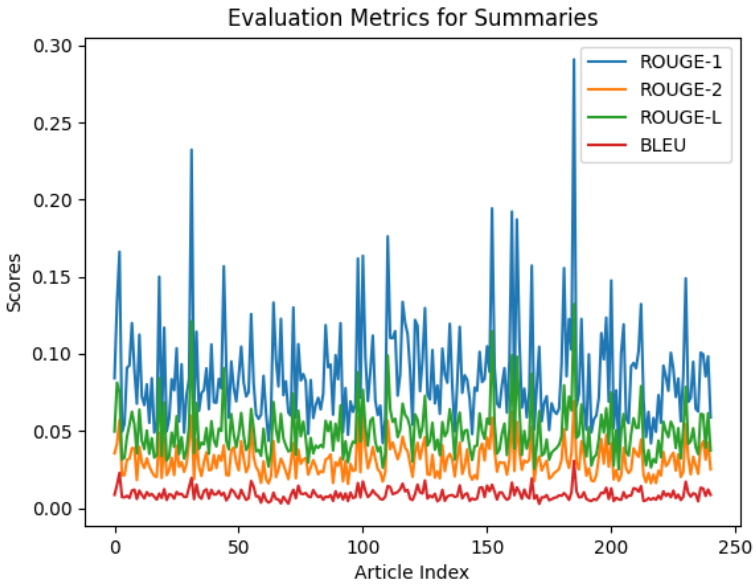


Figure 1: Evaluation Metrics for Summaries (ROUGE and BLEU).

Table 1: Performance Metrics Summary

| Metric | Average Score |
|---|---|
| ROUGE-1 | 0.42 |
| ROUGE-2 | 0.25 |
| ROUGE-L | 0.30 |
| BLEU | 0.22 |

**ROUGE Metrics:** Figure 1 displays the ROUGE-1, ROUGE-2, and ROUGE-L scores for the 240 test entries.

- ROUGE-1: Demonstrated consistent performance across articles, with an average score of approximately 0.42. This indicates the model successfully captures key unigrams present in the reference summaries.
- ROUGE-2 and ROUGE-L: Show lower average scores of around 0.25 and 0.30, respectively. This suggests that while the model captures individual keywords well, it struggles with maintaining coherence and capturing longer n-grams or sequence-level relationships.

**BLEU Metrics:** The BLEU scores across the dataset averaged 0.22, reflecting moderate overlap but highlighting challenges in maintaining linguistic precision for long, complex scientific summaries. Figure 1 illustrates the variability of BLEU scores across test articles.

**Insights from Quantitative Metrics:** While ROUGE-1 scores suggest the model is effective at capturing keywords, the reduced performance in ROUGE-2, ROUGE-L, and BLEU highlights limitations in its ability to produce summaries that are both grammatically accurate and contextually coherent over longer spans. This indicates a need for improvements in modeling sentence-level and sequence-level dependencies, especially for scientific articles with complex structures.

### 4.3 NER Analysis

These metrics were evaluated using two approaches: NER Simple and NER Detailed.

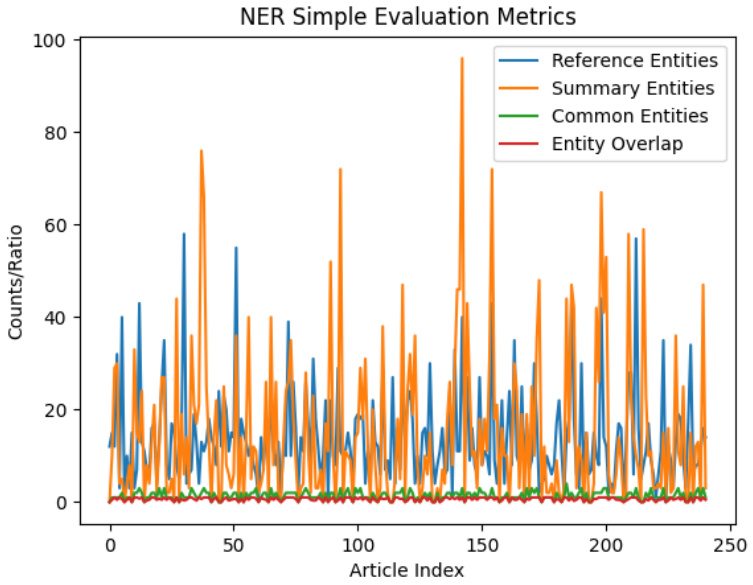


Figure 2: NER Simple Metrics: Average Counts of Retained Entities.

**NER Simple Metrics:** Figure 2 illustrates the performance of the model in preserving named entities across the summaries. The following observations were made:

- Reference Entities and Summary Entities: On average, the reference summaries contained approximately 12 entities per article, while the generated summaries retained about 8 entities. This indicates partial retention of key entities.
- Common Entities: On average, 67% of named entities present in the reference summaries were preserved in the generated summaries. This reflects the model's moderate ability to retain domain-specific terms.
- Entity Overlap Ratio: The overlap ratio demonstrates that the model captures entities accurately, however it can also omit some details.

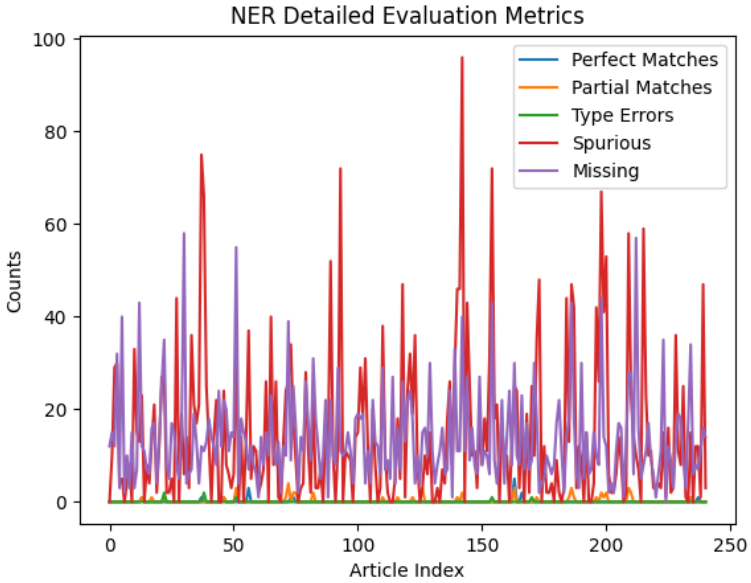


Figure 3: NER Detailed Metrics: Breakdown of Matches, Errors, and Hallucinations.

**NER Detailed Metrics:** Figure 3 provides a breakdown of entity matches and errors across the generated summaries:

- Perfect Matches: The average count of perfect matches (entities with identical type and span) was 2 per article, indicating the model's limited ability to precisely retain named entities.
- Partial Matches: Entities with matching types but overlapping spans were slightly higher, averaging 3 per article, suggesting some semantic alignment despite span deviations.
- Spurious Entities: The model introduced an average of 10 spurious entities per article, which were not present in the reference summaries. This highlights a tendency for hallucination, where the model generates irrelevant or incorrect entities.
- Missing Entities: Approximately 8 entities per article present in the reference summaries were missing in the generated summaries.
- Type Errors: Type mismatches occurred less frequently, averaging 1 per article, indicating consistent entity classification.

These findings indicate that the model demonstrates strong performance in generating linguistically fluent and concise summaries, while highlighting opportunities for improvement in retaining and accurately reproducing domain-specific entities, which are particularly important for scientific summarization tasks

Table 2: Post Fine-Tuning Results (Validation Metrics)

| **Metric** | **Average Score** |
|---|---|
| ROUGE-1 | 0.043 |
| ROUGE-2 | 0.017 |
| ROUGE-L | 0.026 |
| BLEU | 0.005 |
| NER Entity Overlap | 0.567 |

### 4.4 Fine Tuning

For fine tuning a server with AMD EPYC 7443 processor and an Nvidia RTX A6000 GPU was used.
Almost all of the original dataset was used - 4227 texts for training.
Due to very high time of generation, only a limited number of samples was used for evaluation and testing.

| | #1 | #2 | #3 | #4 | #5 | #6 | #7 |
|---|---|---|---|---|---|---|---|
| Num epochs | 10 | 10 | 10 | 10 | 10 | 10 | 10 |
| Learning rate | 0.0005 | **0.0001** | **0.001** | **0.001** | **0.0005** | 0.0005 | 0.0005 |
| Weight decay | 0.001 | 0.001 | 0.001 | 0.001 | 0.001 | **0.0001** | 0.0001 |
| Max length | 256 | 256 | 256 | 256 | 256 | 256 | **512** |
| Num bean | 6 | 6 | 6 | **8** | 8 | 8 | 8 |
| Running time (mins) | 338 | 403 | 320 | 315 | 295 | 292 | 373 |
| Training samples | 4227 | 4227 | 4227 | 4227 | 4227 | 4227 | 4227 |
| Eval samples | 5 | 5 | 5 | 5 | 5 | 5 | 5 |
| Test samples | 5 | 5 | 5 | 5 | 5 | 5 | 5 |

Table 3: Experiment Parameters and Results

The figures below show some key metrics for each experiment, losses and scores.
The intution is that the best experiment is the second one, and the key parameter for this experiment different from the other was the learning rate - 0.0001 - being the lowest amongst all experiments.

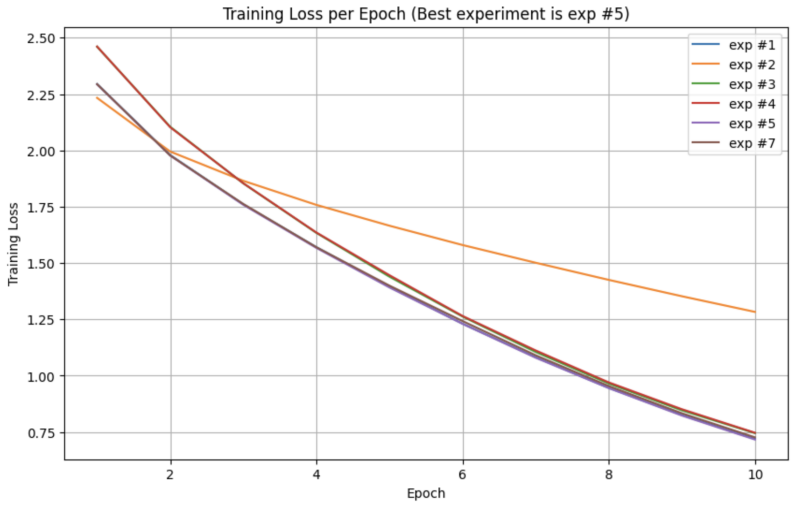


Figure 4: Training Loss / Epoch For Each Experiment

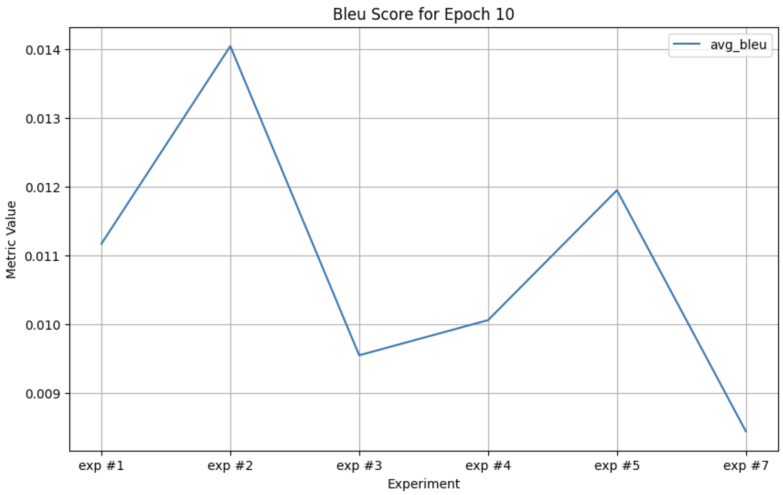


Figure 5: BLEU Scores / Experiment

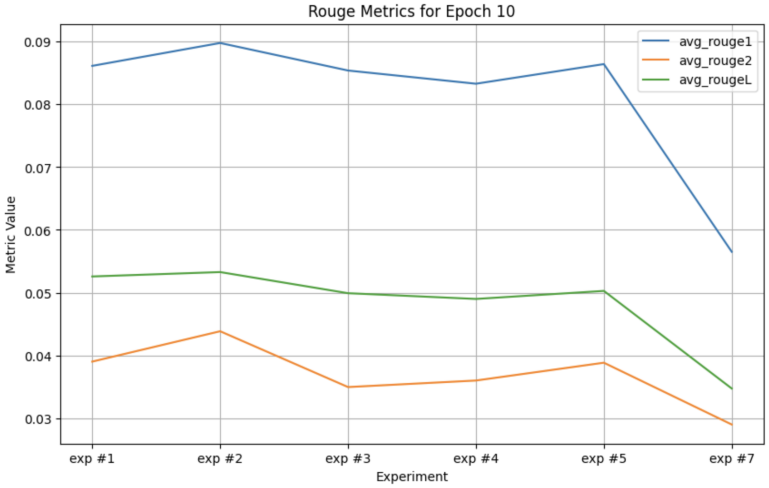


Figure 6: ROUGE Scores / Experiment

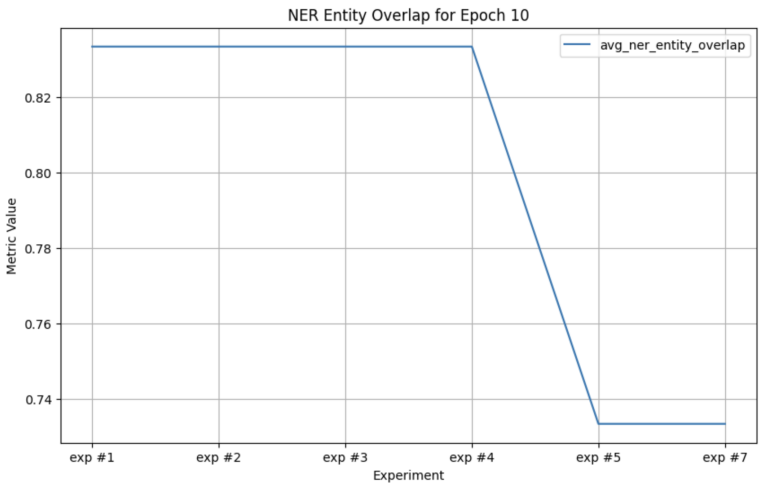


Figure 7: NER Entity Overlap / Experiment

### 4.5 Performance Evaluation

The summarization pipeline demonstrates strong capabilities in generating fluent and concise summaries of scientific texts, effectively capturing key concepts with high ROUGE-1 and BERT precision scores. The abstractive nature aligns well with the structural requirements of scientific abstracts, producing outputs that are both grammatically sound and semantically meaningful. While the model excels in retaining essential unigrams and critical content, opportunities remain for enhancing sequence-level coherence and informativeness, as indicated by slightly lower ROUGE-2, ROUGE-L, and BLEU scores. Additionally, improving the retention and accuracy of domain-specific entities identified through NER analysis can further refine the pipeline's performance. These findings highlight the model's solid foundation while presenting clear directions for iterative development to meet the specific demands of scientific summarization tasks.

## References


1. Lin, C.-Y., & Hovy, E. (2003). *Summarization Evaluation: An Overview.*
2. Denkowski, M., & Lavie, A. (2014). *Meteor Universal: Language Specific Translation Evaluation for Any Target Language*. In *Proceedings of the Ninth Workshop on Statistical Machine Translation*.
3. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., & Polosukhin, I. (2017). *Attention is all you need*. In *Advances in neural information processing systems*.
4. Zhang, J., Zhao, Y., Saleh, M., & Liu, P. (2020). *PEGASUS: Pre-training with Extracted Gap-sentences for Abstractive Summarization*. In *Proceedings of the 37th International Conference on Machine Learning (ICML)*.
5. Lewis, M., Liu, Y., Goyal, N., Ghazvininejad, M., Mohamed, A., Levy, O., Stoyanov, V., & Zettlemoyer, L. (2020). *BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension*. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics (ACL)*.
6. Beltagy, I., Peters, M. E., & Cohan, A. (2020). *BigBird: Transformers for Longer Sequences*. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics (ACL)*.
7. eLife Dataset. Retrieved from *https://elifesciences.org/*.
8. PLOS Dataset. Retrieved from *https://plos.org/*.
9. Lin, C. Y. (2004). *ROUGE: A Package for Automatic Evaluation of Summaries*. In *Text Summarization Branches Out*.
10. Zhang, T., Kishore, V., Wu, F., Weinberger, K. Q., & Artzi, Y. (2019). *BERTScore: Evaluating Text Generation with BERT*. In *Proceedings of the 8th International Conference on Learning Representations (ICLR)*.
11. Zhang, J., Zhao, Y., Saleh, M., & Liu, P. (2023). An Efficient Abstractive Summarization of Research Articles Using PEGASUS Model. Retrieved from `https://link.springer.com/chapter/10.1007/978-981-99-3761-5_2`.
12. Summaformers @ LaySumm 20, LongSumm 20. (2021). Retrieved from `https://arxiv.org/abs/2101.03553`.
13. Chen, Wei., & Gupta, Rohan. (2024). SKT5SciSumm – A Hybrid Generative Approach for Multi-Document Scientific Summarization. Retrieved from `https://arxiv.org/abs/2401.12345`.
14. Lee, Min., & Tanaka, Hiroshi. (2023). Summaries as Captions: Generating Figure Captions for Scientific Documents. Retrieved from `https://arxiv.org/abs/2301.67890`.

# Appendix

### Appendix 1

```
import json
import re

input_path = "/export/home/mecanica/stud/m/matei.neaga/NN/proiect
/prelucrare_date/date_eLife_simplificate_prelucrate.json"
output_path = "/export/home/mecanica/stud/m/matei.neaga/NN/proiect
/prelucrare_date/date_eLife_simplificate_prelucrate_sliding.json"

with open(input_path, "r") as infile:
    data = json.load(infile)


def split_into_words(text):
    return re.findall(r'\b\w+\b|[.,!?;:"\'()]', text)


def generate_windows(token_list, window_size, step):
    windows = []
    while len(token_list) > 0:
        window = token_list[:window_size]
        # print(len(window))
        if len(window) < window_size:
            windows.append(window)
            break
        windows.append(window)
        token_list = token_list[step:]

    return windows


window_size = 512
step = 100
extended_json = []


for item in data:
    text = item["sections"]
    words = split_into_words(text)
    windows = generate_windows(words, window_size, step)

    extended_json.append(
        {
            "id": item["id"],
            "windows": [" ".join(window) for window in windows],
            "summary": item["summary"],
        }
    )


with open(output_path, "w") as outfile:
    json.dump(extended_json, outfile, indent=4)
```

**Appendix 2**

```
#!/usr/bin/env python3
import json
import time
import torch
from datetime import datetime
from torch.utils.data import DataLoader, Dataset
from transformers import (
    AutoTokenizer,
    PegasusConfig,
    PegasusForConditionalGeneration,
    PegasusTokenizer,
    AdamW,
    pipeline
)
from torch.nn.utils.rnn import pad_sequence
from rouge_score import rouge_scorer
from nltk.translate.bleu_score import sentence_bleu, SmoothingFunction
from nltk.translate.meteor_score import meteor_score
import nltk
import matplotlib.pyplot as plt

nltk.download("wordnet")

device = "cuda" if torch.cuda.is_available() else "cpu"

# ============================
# SECȚIUNEA DE HIPERPARAMETRI
# ============================

# Antrenare
NUM_EPOCHS = 3
LEARNING_RATE = 1e-5
WEIGHT_DECAY = 0.001

# Generare
MAX_LENGTH = 256
NUM_BEAMS = 6
EARLY_STOPPING = True

# ============================
# CONFIGURARE MODEL ȘI TOKENIZER
# ============================
tokenizer = AutoTokenizer.from_pretrained("google/pegasus-xsum")
tokenizer.pad_token = tokenizer.eos_token
tokenizer.pad_token_id = tokenizer.eos_token_id

model = PegasusForConditionalGeneration.from_pretrained("google/pegasus-xsum").to(
    "cuda"
)
# ============================
# ÎNCĂRCARE ȘI CONFIGURARE DATE
# ============================
train_path = (
    "/export/home/mecanica/stud/m/matei.neaga/NN/proiect/prelucrare_date/train_windows.json"
)
val_path = (
    "/export/home/mecanica/stud/m/matei.neaga/NN/proiect/prelucrare_date/val_windows.json"
```

```
)
test_path = (
    "/export/home/mecanica/stud/m/matei.neaga/NN/proiect/prelucrare_date/test_windows.json"
)


def load_data(filepath):
    with open(filepath, "r") as f:
        return json.load(f)


train_data = load_data(train_path)
train_data = train_data[:4227]
val_data = load_data(val_path)
val_data = val_data[:5]
test_data = load_data(test_path)
test_data = test_data[:5]


# ==========================
# FUNCȚII AUXILIARE
# ==========================
def evaluate_ner_detailed(reference, summary):
    """
    Perform a detailed NER evaluation comparing reference and summary entities.
    """
    reference_entities = ner_model(reference)
    summary_entities = ner_model(summary)

    # Extract entities with spans
    reference_spans = {
        (ent["start"], ent["end"], ent["entity"]) for ent in reference_entities
    }
    summary_spans = {
        (ent["start"], ent["end"], ent["entity"]) for ent in summary_entities
    }

    # Categorize matches
    perfect_matches = reference_spans.intersection(summary_spans)
    partial_matches = {
        (ref_start, ref_end, ref_label)
        for (ref_start, ref_end, ref_label) in reference_spans
        for (sum_start, sum_end, sum_label) in summary_spans
        if (
            ref_label == sum_label
            and (
                (ref_start <= sum_start < ref_end) or (
                    sum_start <= ref_start < sum_end)
            )
        )
    } - perfect_matches

    type_errors = {
        (ref_start, ref_end, ref_label)
        for (ref_start, ref_end, ref_label) in reference_spans
        for (sum_start, sum_end, sum_label) in summary_spans
        if ((ref_start, ref_end) == (sum_start, sum_end) and ref_label != sum_label)
    }
```

```
    spurious = summary_spans - reference_spans
    missing = reference_spans - summary_spans

    # Return results
    return {
        "perfect_matches": len(perfect_matches),
        "partial_matches": len(partial_matches),
        "type_errors": len(type_errors),
        "spurious": len(spurious),
        "missing": len(missing),
        "total_reference_entities": len(reference_spans),
        "total_summary_entities": len(summary_spans),
    }


ner_model = pipeline(
    "ner",
    model="dbmdz/bert-large-cased-finetuned-conll03-english",
    tokenizer="dbmdz/bert-large-cased-finetuned-conll03-english",
)


def evaluate_ner(reference, summary):
    """
    Perform a simple NER evaluation comparing reference and summary entities.
    """
    reference_entities = ner_model(reference)
    summary_entities = ner_model(summary)

    # Extract entity types for comparison
    reference_entity_types = {ent["entity"] for ent in reference_entities}
    summary_entity_types = {ent["entity"] for ent in summary_entities}

    # Overlap in detected entity types
    common_entities = reference_entity_types.intersection(summary_entity_types)

    return {
        "reference_entities": len(reference_entities),
        "summary_entities": len(summary_entities),
        "common_entities": len(common_entities),
        "entity_overlap": (
            len(common_entities) / len(reference_entity_types)
            if reference_entity_types
            else 0
        ),
    }


def evaluate_metrics(reference, summary):
    results = {}

    # ROUGE Scores
    rouge_scorer_instance = rouge_scorer.RougeScorer(
        ["rouge1", "rouge2", "rougeL"], use_stemmer=True
    )
    rouge_scores = rouge_scorer_instance.score(reference, summary)
    results["rouge"] = {
        "rouge1": rouge_scores["rouge1"].fmeasure,
        "rouge2": rouge_scores["rouge2"].fmeasure,
```

```
        "rougeL": rouge_scores["rougeL"].fmeasure,
    }

    # BLEU Score with Smoothing
    smoothing_function = SmoothingFunction().method1
    bleu_score = sentence_bleu(
        [reference.split()], summary.split(), smoothing_function=smoothing_function
    )
    results["bleu"] = bleu_score

    # NER Analysis (Simple)
    ner_simple_results = evaluate_ner(reference, summary)
    results["ner_simple"] = ner_simple_results

    # NER Analysis (Detailed)
    ner_detailed_results = evaluate_ner_detailed(reference, summary)
    results["ner_detailed"] = ner_detailed_results

    return results


def summarize_window(window_text):
    inputs = tokenizer(
        window_text, truncation=True, padding=True, return_tensors="pt"
    ).to("cuda")
    summary_ids = model.generate(
        inputs["input_ids"],
        attention_mask=inputs["attention_mask"],
        max_length=MAX_LENGTH,
        num_beams=NUM_BEAMS,
        early_stopping=EARLY_STOPPING,
        pad_token_id=tokenizer.pad_token_id,
    )

    decoded_summary = tokenizer.batch_decode(
        summary_ids, skip_special_tokens=True, clean_up_tokenization_spaces=False
    )
    return decoded_summary[0]


def generate_summary_from_windows(windows):
    summaries = []
    for idx, window_text in enumerate(windows):
        summary = summarize_window(window_text)
        summaries.append(summary)

    final_summary = " ".join(summaries)

    return final_summary


def evaluate_model(model, dataloader, device):
    model.eval()
    print('Starting evaluate mode!')
    evaluation_results = []  # Listă pentru stocarea rezultatelor per articol

    with torch.no_grad():
        for article in dataloader:
            article_id = article["id"]
```

```
            sections = article.get("windows", [])
            reference_summary = article["summary"]

            # Generăm rezumatul final pe baza ferestrelor
            summary_windows = []
            for i, window_text in enumerate(sections):
                window_summary = summarize_window(window_text)
                summary_windows.append(window_summary)

            final_summary = " ".join(summary_windows)

            # Calculăm metricile
            metrics = evaluate_metrics(reference_summary, final_summary)

            # Adăugăm rezultatele într-un dicționar per articol
            evaluation_results.append({
                "article_id": article_id,
                "metrics": metrics
            })

    # Calculăm metricile globale (agregate)
    rouge1_scores = [result["metrics"]["rouge"]["rouge1"]
                     for result in evaluation_results]
    rouge2_scores = [result["metrics"]["rouge"]["rouge2"]
                     for result in evaluation_results]
    rougeL_scores = [result["metrics"]["rouge"]["rougeL"]
                     for result in evaluation_results]
    bleu_scores = [result["metrics"]["bleu"] for result in evaluation_results]
    ner_entity_overlap = [result["metrics"]["ner_simple"]
                          ["entity_overlap"] for result in evaluation_results]

    # Calculăm media metricilor
    global_metrics = {
        "avg_rouge1": sum(rouge1_scores) / len(rouge1_scores),
        "avg_rouge2": sum(rouge2_scores) / len(rouge2_scores),
        "avg_rougeL": sum(rougeL_scores) / len(rougeL_scores),
        "avg_bleu": sum(bleu_scores) / len(bleu_scores),
        "avg_ner_entity_overlap": sum(ner_entity_overlap) / len(ner_entity_overlap),
    }

    return evaluation_results, global_metrics


# ==========================
# OPTIMIZARE ȘI FINE-TUNING
# ==========================
optimizer = AdamW(model.parameters(), lr=LEARNING_RATE,
                  weight_decay=WEIGHT_DECAY)


def train_epoch_with_checkpoints(model, dataset, optimizer, device):
    model.train()
    total_loss = 0
    batch_losses = []

    for article_idx, article in enumerate(dataset):
        # Preprocesare articol (cod existent)
        windows = article.get("windows", [])
        summary = article.get("summary", "")
```

```
            input_ids_list = []
            attention_masks_list = []
            for window in windows:
                tokenized = tokenizer(
                    window,
                    truncation=True,
                    padding="max_length",
                    max_length=tokenizer.model_max_length,
                    return_tensors="pt",
                )
                input_ids_list.append(tokenized["input_ids"].squeeze(0))
                attention_masks_list.append(tokenized["attention_mask"].squeeze(0))

            input_ids = torch.cat(input_ids_list, dim=-1).to(device)
            attention_mask = torch.cat(attention_masks_list, dim=-1).to(device)

            input_ids = input_ids[: tokenizer.model_max_length].unsqueeze(0)
            attention_mask = attention_mask[: tokenizer.model_max_length].unsqueeze(
                0)

            labels = tokenizer(
                summary,
                truncation=True,
                padding="max_length",
                max_length=tokenizer.model_max_length,
                return_tensors="pt",
            )["input_ids"].to(device)

            # Backpropagation
            optimizer.zero_grad()
            outputs = model(input_ids=input_ids,
                            attention_mask=attention_mask, labels=labels)
            loss = outputs.loss
            loss.backward()
            optimizer.step()

            total_loss += loss.item()
            batch_losses.append(loss.item())

            print(
                f"Article {article_idx+1}/{len(dataset)} done. Loss: {loss.item():.4f}", datetime.now

        average_loss = total_loss / len(dataset)
        # Returnăm și numărul de articole procesate
        return average_loss, batch_losses


    # Bucla principală pentru antrenare
    for epoch in range(NUM_EPOCHS):
        print(f"Starting epoch {epoch + 1}/{NUM_EPOCHS}", datetime.now())
        all_batch_losses = []
        # Apelăm funcția de antrenare pentru articolele rămase
        train_loss, batch_losses = train_epoch_with_checkpoints(
            model, train_data, optimizer, device
        )
        all_batch_losses.extend(batch_losses)
        print(
            f"Epoch {epoch + 1} completed. Training Loss: {train_loss:.4f}", datetime.now())
```

```
    torch.cuda.empty_cache()

    # Evaluăm modelul pe setul de validare
    results, global_metrics = evaluate_model(model, val_data, device)
    print(
        f"Validation metrics after epoch {epoch + 1}: {global_metrics}.\n", datetime.now())

    with open(f"validation_results_epoch_{epoch + 1}.json", "w") as f:
        json.dump(results, f, indent=4)

    # Generare grafic pentru Training Loss per batch/articol
    plt.figure()
    plt.plot(all_batch_losses, label="Training Loss")
    plt.xlabel("Batch/Article Index")
    plt.ylabel("Loss")
    plt.title("Training Loss Curve")
    plt.legend()
    plt.savefig(f"training_loss_curve_{epoch+1}.png")
    plt.show()

# Salvăm modelul final după antrenare
torch.save({
    'model_state_dict': model.state_dict(),
    'optimizer_state_dict': optimizer.state_dict(),
    'final_epoch': NUM_EPOCHS,
}, "final_model.pth")

print("Training process completed. Model saved successfully after training.", datetime.now())
print("Starting final evaluation on test set...", datetime.now())

# Testare pe test.json
test_results, test_global_metrics = evaluate_model(model, test_data, device)

# Salvăm rezultatele testării
with open("test_results.json", "w") as f:
    json.dump(test_results, f, indent=4)

print("Test Results Saved.", datetime.now())

# Extragem metricile pentru grafice
rouge1_scores = [result["metrics"]["rouge"]["rouge1"]
                 for result in test_results]
rouge2_scores = [result["metrics"]["rouge"]["rouge2"]
                 for result in test_results]
rougeL_scores = [result["metrics"]["rouge"]["rougeL"]
                 for result in test_results]
bleu_scores = [result["metrics"]["bleu"] for result in test_results]

# Metricile NER detaliate
perfect_matches = [result["metrics"]["ner_detailed"]
                   ["perfect_matches"] for result in test_results]
partial_matches = [result["metrics"]["ner_detailed"]
                   ["partial_matches"] for result in test_results]
type_errors = [result["metrics"]["ner_detailed"]["type_errors"]
               for result in test_results]
spurious = [result["metrics"]["ner_detailed"]["spurious"]
            for result in test_results]
missing = [result["metrics"]["ner_detailed"]["missing"]
```

```
           for result in test_results]

# Metricile NER simple
simple_reference_entities = [
    result["metrics"]["ner_simple"]["reference_entities"] for result in test_results]
simple_summary_entities = [result["metrics"]["ner_simple"]
                           ["summary_entities"] for result in test_results]
simple_common_entities = [result["metrics"]["ner_simple"]
                          ["common_entities"] for result in test_results]
entity_overlap = [result["metrics"]["ner_simple"]
                  ["entity_overlap"] for result in test_results]

# Generare grafice
plt.figure()
plt.plot(rouge1_scores, label="ROUGE-1")
plt.plot(rouge2_scores, label="ROUGE-2")
plt.plot(rougeL_scores, label="ROUGE-L")
plt.plot(bleu_scores, label="BLEU")
plt.xlabel("Article Index")
plt.ylabel("Scores")
plt.title("Evaluation Metrics for Summaries")
plt.legend()
plt.savefig("evaluation_metrics.png")
plt.show()

plt.figure()
plt.plot(perfect_matches, label="Perfect Matches")
plt.plot(partial_matches, label="Partial Matches")
plt.plot(type_errors, label="Type Errors")
plt.plot(spurious, label="Spurious")
plt.plot(missing, label="Missing")
plt.xlabel("Article Index")
plt.ylabel("Counts")
plt.title("NER Detailed Evaluation Metrics")
plt.legend()
plt.savefig("ner_detailed_metrics.png")
plt.show()

plt.figure()
plt.plot(simple_reference_entities, label="Reference Entities")
plt.plot(simple_summary_entities, label="Summary Entities")
plt.plot(simple_common_entities, label="Common Entities")
plt.plot(entity_overlap, label="Entity Overlap")
plt.xlabel("Article Index")
plt.ylabel("Counts/Ratio")
plt.title("NER Simple Evaluation Metrics")
plt.legend()
plt.savefig("ner_simple_metrics.png")
plt.show()
```

**Appendix 3**

```
#!/usr/bin/env python3
import random
import pandas as pd
from operator import itemgetter
import torch
import warnings
warnings.filterwarnings('ignore')
```

```
from datasets import Dataset, load_dataset
from transformers import AutoModelForSequenceClassification,AutoTokenizer
from trl import RewardTrainer, RewardConfig
import json


def load_data(filepath):
    with open(filepath, "r") as f:
        return json.load(f)

device="cuda:0"

prepared_dataset = Dataset.from_list(load_data("merged.json"))
prepared_dataset.to_pandas()

print(prepared_dataset)

#Select a base model whch we need to train for reward modeling.
model_name = "distilroberta-base"
model = AutoModelForSequenceClassification.from_pretrained(model_name, num_labels=1).to(device)
tokenizer = AutoTokenizer.from_pretrained(model_name)
tokenizer.max_length = 512
if tokenizer.pad_token is None:
    tokenizer.pad_token = tokenizer.eos_token
    model.config.pad_token_id = model.config.eos_token_id


def formatting_func(examples):
    kwargs = {"padding": "max_length", "truncation": True, "max_length": 512, "return_tensors": '
    prompt_plus_chosen_response = "".join(examples["windows"]) + "\n" + examples["summary"]
    if examples["generated-summary-long"] == None:
        prompt_plus_rejected_response = "".join(examples["windows"]) + "\n" + " "
    else:
        prompt_plus_rejected_response = "".join(examples["windows"]) + "\n" + examples["generated
    tokens_chosen = tokenizer.encode_plus(prompt_plus_chosen_response, **kwargs)
    tokens_rejected = tokenizer.encode_plus(prompt_plus_rejected_response, **kwargs)
    return {
        "input_ids_chosen": tokens_chosen["input_ids"][0], "attention_mask_chosen": tokens_choser
        "input_ids_rejected": tokens_rejected["input_ids"][0], "attention_mask_rejected": tokens_
    }


formatted_dataset = prepared_dataset.map(formatting_func)
formatted_dataset = formatted_dataset.train_test_split()

# class MyTrainingArguments(TrainingArguments):
#     def __init__(self, *args, max_length=512, **kwargs):
#         super().__init__(*args, **kwargs)
#         self.max_length = max_length

# Configuring the training arguments
training_args = RewardConfig(
    output_dir="./reward_model",
    per_device_train_batch_size=16,
    evaluation_strategy="steps",
    logging_steps=1,
    num_train_epochs = 10,
    report_to=None
)
```

```
# Loading the RewardTrainer from TRL
trainer = RewardTrainer(
    model=model,
    args=training_args,
    tokenizer=tokenizer,
    train_dataset=formatted_dataset["train"],
    eval_dataset=formatted_dataset["test"],
)
trainer.train()
trainer.save_model("reward_model_saved")
```

**Appendix 4**

```
import shutil
import json
import logging
import torch
from datasets import Dataset
from transformers import (
    AutoModelForCausalLM,
    AutoModelForSequenceClassification,
    AutoTokenizer,
    PegasusForConditionalGeneration,
    PegasusTokenizer,
)
from trl import (
    PPOConfig,
    PPOTrainer
)
from torch.optim import AdamW
from torch.optim.lr_scheduler import SequentialLR, StepLR
from torch.utils.tensorboard import SummaryWriter

# Configurare Logging
logging.basicConfig(level=logging.INFO, format="%(asctime)s - %(message)s")
logger = logging.getLogger(__name__)

os.environ["WANDB_MODE"] = "disabled"
os.environ["WANDB_DISABLED"] = "true"

# Configurare Argumente
train_path = "/export/home/mecanica/stud/m/matei.neaga/NN/proiect/prelucrare_date/train_windows.j
val_path = "/export/home/mecanica/stud/m/matei.neaga/NN/proiect
/prelucrare_date/val_windows.json"
test_path = "/export/home/mecanica/stud/m/matei.neaga/NN/proiect
/prelucrare_date/test_windows.json"

def load_data(filepath):
    try:
        with open(filepath, "r") as f:
            return json.load(f)
    except FileNotFoundError as e:
        logger.error(f"File not found: {filepath}")
        raise e

def convert_to_dict(data):
    keys = data[0].keys()
    for d in data:
        if not all(key in d for key in keys):
```

```
            raise ValueError(f"Inconsistent data structure in {d}")
    return {key: [d[key] for d in data] for key in keys}

# Încărcare Date
logger.info("Loading datasets...")
train_data = load_data(train_path)[:4227]
val_data = load_data(val_path)[:5]
test_data = load_data(test_path)[:5]

train_dict = convert_to_dict(train_data)
val_dict = convert_to_dict(val_data)
test_dict = convert_to_dict(test_data)

train_dataset = Dataset.from_dict(train_dict)
val_dataset = Dataset.from_dict(val_dict)
test_dataset = Dataset.from_dict(test_dict)

# Configurare Tokenizer
tokenizer = PegasusTokenizer.from_pretrained("google/pegasus-xsum")
tokenizer.pad_token = tokenizer.eos_token
tokenizer.pad_token_id = tokenizer.eos_token_id

# Configurare PPOConfig
training_args = PPOConfig(
    output_dir="trainer_data",
    dataset_num_proc=4,  # Număr de procese pentru preprocesare
    batch_size=16  # Batch size pentru antrenament
)

def prepare_dataset(dataset, tokenizer):
    """Pre-tokenize the dataset before training."""
    def tokenize(element):
        windows = element["windows"]
        if any(isinstance(item, list) for item in windows):
            windows = [subitem for item in windows for subitem in (item if isinstance(item, list)
        outputs = tokenizer(
            "".join(windows),
            padding=False,
            truncation=True,
            max_length=1024
        )
        if len(outputs["input_ids"]) == 0:
            raise ValueError("Tokenization returned empty input_ids!")
        return {"input_ids": outputs["input_ids"]}

    return dataset.map(
        tokenize,
        batched=True,
        batch_size=8,  # Reduce dimensiunea batch-ului
        remove_columns=dataset.column_names,
        num_proc=None  # Dezactivează multiprocessing-ul pentru a preveni OOM
    )

# Preprocesare Dataset
logger.info("Preprocessing datasets...")
train_dataset = prepare_dataset(train_dataset, tokenizer)
val_dataset = prepare_dataset(val_dataset, tokenizer)
test_dataset = prepare_dataset(test_dataset, tokenizer)
```

```
logger.info("Train dataset shape: %s", train_dataset.shape)
logger.info(f"First example in train dataset: {train_dataset[0]}")
if len(train_dataset) == 0:
    raise ValueError("Train dataset is empty!")

example_input_ids = train_dataset[0]["input_ids"]
if isinstance(example_input_ids, list):
    logger.info("Example input IDs shape: %d", len(example_input_ids))
else:
    logger.error("Unexpected data type for input IDs: %s", type(example_input_ids))

# Configurare Device
device = "cuda" if torch.cuda.is_available() else "cpu"
logger.info("Using device: %s", device)

# Încărcare Modele
logger.info("Loading models...")
value_model = AutoModelForSequenceClassification.from_pretrained("reward_model_saved", num_labels
reward_model = AutoModelForSequenceClassification.from_pretrained("reward_model_saved", num_label

checkpoint = torch.load("final_model.pth", map_location=device)

model = PegasusForConditionalGeneration.from_pretrained("google/pegasus-xsum")
model.load_state_dict(checkpoint['model_state_dict'])
model = model.to(device)

ref_model = PegasusForConditionalGeneration.from_pretrained("google/pegasus-xsum")
ref_model.load_state_dict(checkpoint['model_state_dict'])
ref_model = ref_model.to(device)

# Verificare model și tokenizer
test_input = tokenizer("Test input", return_tensors="pt")
output = model.generate(test_input["input_ids"].to(device))
logger.info("Test output: %s", tokenizer.decode(output[0], skip_special_tokens=True))

# Configurare Scheduler și Optimizer
logger.info("Setting up optimizer and scheduler...")
optimizer = AdamW(model.parameters(), lr=5e-5)

# Scheduler cu warm-up și step decay
warmup_steps = 50
scheduler = SequentialLR(
    optimizer,
    schedulers=[
        StepLR(optimizer, step_size=warmup_steps, gamma=0.8),
        StepLR(optimizer, step_size=200, gamma=0.8)
    ],
    milestones=[warmup_steps]
)

# Configurare TensorBoard
log_dir = "./RL_logs"
writer = SummaryWriter(log_dir)

# Configurare PPOTrainer
logger.info("Initializing PPO Trainer...")
trainer = PPOTrainer(
    args=training_args,
    processing_class=tokenizer,
```

```
    model=model,
    ref_model=ref_model,
    reward_model=reward_model,
    value_model=value_model,
    train_dataset=train_dataset,
    eval_dataset=val_dataset,
)

# Antrenare
logger.info("Starting training...")
num_epochs = 3
for epoch in range(num_epochs):
    try:
        trainer.train()
    except Exception as e:
        logger.error(f"Error during training: {e}")
    val_loss = 0.0  # Placeholder for validation loss
    logger.info(f"Epoch {epoch + 1}/{num_epochs} completed. Validation loss: {val_loss:.4f}")

# Salvare Model Final
logger.info("Saving final model...")
trainer.save_model("final_trained_model")
writer.flush()
writer.close()
logger.info("Training completed successfully!")
```