Computional Linguistic Task 2 - report

Jerzy Boksa

November 2025

1 Introduction

The goal of this assignment is to analyze how different tokenization strategies influence the performance and behavior of a language model. Students will implement and compare three tokenizers and evaluate their effects on model efficiency and text representation quality.

2 Text Corpus

The text corpus used in this work is the `wolne_lektury_corpus` from Speakleash, comprising 6,619 Polish books, poems, and literary texts. The dataset was carefully preprocessed to remove non-essential elements such as author names, publication dates, and other metadata appearing at the beginning of poems or texts.

The corpus was divided into training and evaluation subsets. The training set consists of 6,453 texts (97.5% of the entire corpus), while the evaluation set contains 166 texts (2.5%).

3 Tokenizers

In this section I will introduce to three the tokenizers that were used in the task. Tokenizers training and preparation is included in the `tokenizers.ipynb` notebook. They were trained only on training dataset.

3.1 Pre-trained tokenizer

The pre-trained tokenizer used in this work is `radlab/polish-gpt2-small-v2` [1], featuring a vocabulary size of 52,000 tokens. It is the tokenizer originally employed in the Polish GPT-2 model developed by Radlab. This tokenizer was selected due to its optimization for the Polish language and its training on a large, diverse Polish text corpus, which is described in detail in the following section. Additional information about Radlab and their work can be found in [2].

Overall, this tokenizer provides a robust and linguistically appropriate foundation for processing Polish text.

3.2 Custom implementation of whitespace-tokenizer

A custom whitespace tokenizer [3] was implemented for this task. It splits the text based on whitespace characters, while treating all punctuation marks as separate tokens. The tokenizer is also case-sensitive, preserving the original capitalization of the text. The tokenizer was trained on the `wolne_lektury_corpus` dataset. Vocabulary size is the same as in the first tokenizer.

3.3 SentencePiece Tokenizer

A SentencePiece tokenizer was trained on the `wolne_lektury_corpus` dataset using a unigram model with a vocabulary size of 52,000.

4 Models and Training

For this task, a decoder-only transformer model was used. The model architecture and training procedure follow the implementation described by Andrej Karpathy [4]. The implementation can be found in `transformer.py` and `train.py`.

Three separate models were trained, one for each tokenizer introduced earlier. Details regarding each model and its training setup are provided in the following sections.

4.1 Datasets

For each tokenizer type, separate training and evaluation datasets were created. Only the training text was passed to the tokenizer to convert it into tensors. Due to differences in the tokenization strategies, the resulting datasets vary in size. These differences are summarized in the table below:

	GPT2-PL	**Whitespace**	**SentencePiece**
Training Tokens Count	62,935,237	44,551,053	55,392,422
Evaluation Tokens Count	2,837,691	1,999,054	2,490,498

Table 1: Number of training and evaluation tokens for models trained with different tokenizers.

As can be observed from Table 1, the GPT2-PL tokenizer produces the largest number of tokens, followed by SentencePiece, with the Whitespace tokenizer generating the fewest. The lower token count for the Whitespace tokenizer was expected, as it only splits text on spaces without further subword segmentation. These differences are significant and reflect how the choice of tokenizer affects the granularity and size of the resulting datasets.

Next, all tokens from each tokenizer were organized into sequences of length 100 and grouped into batches of size 196. Three separate models were then trained on these datasets, following the architecture described in the next subsection.

4.2 Model architecture and hardware

All models were trained using the same configuration; the only difference between them was the tokenizer. To prevent overfitting, a learning rate scheduler with `LROnPlateau` was employed, as in the previous task.

The training hyperparameters were as follows:

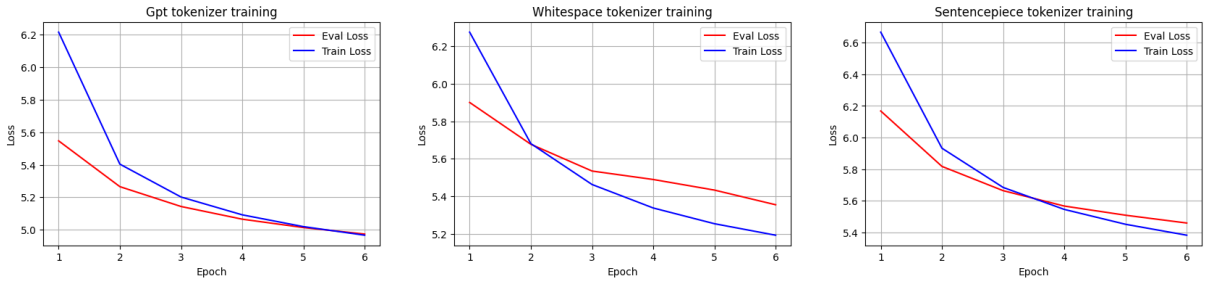
- **Sequence length** (`seq_len`): 100 tokens per input sequence.
- **Batch size** (`batch_size`): 196 sequences per batch.
- **Learning rate** (`lr`): 5×10^{-4}, used as the initial learning rate.
- **Dropout** (`dropout`): 0.3, applied to embeddings and attention layers to reduce overfitting.
- **Number of attention heads** (`heads_count`): 4, controlling the multi-head attention mechanism.
- **Embedding dimension** (`embedding_dim`): 144, defining the size of token embeddings.
- **Number of transformer blocks** (`blocks_count`): 4, determining the depth of the model.
- **Maximum epochs** (`max_epochs`): 6, setting the upper limit for training iterations.

This configuration ensures consistent training across all models while providing sufficient capacity for learning the patterns in Polish text.

For the training GPU Nvidia a100 was used.

4.3 Training

Training proceeded similarly for all models. In each case, both training and evaluation losses decreased steadily, as illustrated in the plots below.



GPT2-PL Whitespace SentencePiece

Figure 1: Training and evaluation losses for models trained with different tokenizers.

All models were able to capture meaningful patterns from the data. Naturally, the extent of learning was limited by the small model size and the relatively short training duration of six epochs. However, direct comparison based solely on evaluation loss is difficult due to differences in tokenizers. Perplexity provides a more appropriate metric for comparison and is presented in the next section.

5 Evaluation Metrics

In this section, a detailed comparison of the models and their respective tokenizers is presented.

5.1 Model Perplexity

Perplexity is a measure of how surprised a model is to encounter a given token, with lower values indicating better predictive performance. The table below presents both word-level and character-level perplexity for the trained models. Perplexity calculation is in the `results.ipynb` jupyter notebook.

Eval data is bounded to 10k words from 1 678 418. For character level it is bouned to 10k chars.

Word level perplexlity is calculated in following way: 1) split evaluation data (models were not trained on this) on words (by regex) 2) iterate over the words, tokenize each of them and calculta perplexity 3) calculate average perplexity per worda

	GPT2-PL	**SentencePiece**	**Whitespace**
Character-level Perplexity	3381.88	3406.16	1082.84
Word-level Perplexity	13191.68	1575.48	4510.76

Table 2: Character-level and word-level perplexities for models trained with different tokenizers.

Models trained on the same dataset (SentencePiece and Whitespace) achieved lower perplexity scores, indicating better adaptation to the corpus. In contrast, the GPT2-PL model, trained on general Polish data, performed worse, which is expected given the domain mismatch. This effect is particularly noticeable for the literary texts used here, which contain uncommon and archaic vocabulary.

5.2 Out-Of-Vocabulary

The out-of-vocabulary (OOV) rate was evaluated by tokenizing the entire evaluation dataset (11.6MB of text) with each of the three tokenizers. For each tokenizer, the proportion of tokens representing unknown or unseen words was computed, providing a measure of how well the tokenizer's vocabulary covers the evaluation corpus. The detailed implementation of this procedure is available in the `results.ipynb` notebook. Results are presented in table below.

Tokenizer	**Total Tokens**	**OOV Tokens**	**OOV %**
GPT2-PL	2,898,717	0	0.0 %
Whitespace	1,999,054	204,180	10.21 %
SentencePiece	2,429,333	0	0.0 %

Table 3: Out-of-vocabulary statistics for different tokenizers on the evaluation dataset.

It is evident that the Whitespace tokenizer performed the worst, with over 10% of tokens being out-of-vocabulary. In contrast, both GPT2-PL and SentencePiece successfully covered all tokens in the evaluation dataset. It is also worth noting that the Whitespace

tokenizer has very few special tokens, which further limits its ability to handle unseen or rare words.

5.3 Efficiency Metrics

Efficiency metrics for the tokenizers were evaluated by processing a full batch of 50 sequences, each of length 100, during training. The number of next tokens processed per second was measured on a dataset of 11.6 MB.

5.3.1 Training Performance

The training performance of the different tokenizers was evaluated by measuring the average processing time per batch of 50 sequences (sequence length = 100). The results are summarized in Table 4.

Tokenizer	**Average Time per 50 Batches (s)**
GPT2-PL	11.02
SentencePiece	10.35
Whitespace	9.38

Table 4: Average processing time per batch during training for different tokenizers.

It is evident that the Whitespace tokenizer required noticeably less time per batch compared to the other tokenizers. SentencePiece performed slightly slower, while GPT2-PL had the longest processing time.

Interestingly, this result is somewhat surprising. Since the datasets for each tokenizer were prepared in advance, one would expect that the time difference primarily reflects model training rather than tokenization. Therefore, the noticeably faster processing of the Whitespace tokenizer is unexpected and may indicate that tokenization overhead was not fully accounted for during the measurement.

5.3.2 Tokens per second

Tokens per second calculation can be found in `reuslts.ipynb`.

Tokenizer	**Tokens per Second**
GPT2-PL	228 857
SentencePiece	160 522
Whitespace	585 611

Table 5: Average number of tokens processed per second by different tokenizers.

The fastest tokenizer in terms of tokens processed per second is Whitespace, followed by GPT2-PL. SentencePiece exhibits the slowest processing speed. This difference is likely due to the simplicity of the Whitespace tokenizer, which generates fewer tokens and requires minimal processing overhead, whereas SentencePiece uses more complex subword tokenization that increases computational cost.

The fastest tokenizer is not necessarily the best. Although the simplest tokenizer processes tokens more quickly, its performance is lower compared to the others, as reflected

in the OOV statistics. However, it still achieves reasonably good character-level perplexity. Overall, the trained SentencePiece tokenizer appears to offer the best trade-off between efficiency and quality.

6 Qualitative Analysis

In this section, tokenizers will be evaluated on manualy selected very hard polish text. We provide sample texts, show their tokenized outputs for each tokenizer, and analyze differences in tokenization granularity, average tokens per word, and out-of-vocabulary handling.

6.1 Tokenization

In this section, we illustrate how different tokenizers process sample texts. For each example, we show the original text followed by the corresponding token sequences produced by GPT2-PL, SentencePiece, and Whitespace tokenizers.

Text to tokenize:

> Chrząszcz brzmi w trzcinie w Szczebrzeszynie, W szczękach chrząszcza trzeszczy miąższ, Czcza szczypawka czka w Szczecinie, Chrząszcza szczudłem przechrzcił wąż, Strząsa skrzydła z dżdżu, A trzmiel w puszczy, tuż przy Pszczynie, Straszny wszczyna szum...

GPT2-PL Tokens:

```
['Ch', 'rzą', 'szcz', ' brzmi', ' w', ' trz', 'cinie', ' w', ' Szcze',
'brze', 'szynie', ',', ' W', ' szczę', 'kach', ' chrząszcza', ' trze',
'szczy', ' miąż', 'sz', ',', ' Cz', 'cza', ' szczy', 'pa', 'wka',
' cz', 'ka', ' w', ' Szczecinie', ',', ' Ch', 'rzą', 'szcza', ' szcz',
'ud', 'łem', ' przech', 'rz', 'cił', ' wąż', ',', ' S', 'trz', 'ą',
'sa', ' skrzydła', ' z', ' d', 'żd', 'żu', ',', ' A', ' trz', 'mie',
'l', ' w', ' p', 'uszczy', ',', ' tuż', ' przy', ' P', 'szczy', 'nie',
',', ' Stra', 'szny', ' wszczyna', ' sz', 'um', '...']
```

SentencePiece Tokens:

```
['Chrz', 'ą', 'szcz', 'brzmi', 'w', 'trzcin', 'ie', 'w', 'Szcze',
'brze', 'szy', 'nie', ',', 'W', 'szczęka', 'ch', 'chrząszcz', 'a',
'trzeszczy', 'miąższ', ',', 'Cz', 'cza', 'szczyp', 'a', 'wka', '',
'czka', 'w', 'Szcze', 'ci', 'nie', ',', 'Chrz', 'ą', 'szcza', 'szczu',
'dłem', 'prze', 'ch', 'rz', 'cił', 'wąż', ',', 'St', 'rzą', 'sa',
'skrzydła', 'z', 'dżdżu', ',', 'A', 'trzmiel', 'w', 'puszczy', ',',
'tuż', 'przy', 'P', 'szczy', 'nie', ',', 'Straszny', 'wszczyna', 'szum',
'...']
```

Whitespace Tokens:

```
['<UNK>', 'brzmi', 'w', '<UNK>', 'w', '<UNK>', ',', 'W', '<UNK>',
'<UNK>', 'trzeszczy', '<UNK>', ',', '<UNK>', '<UNK>', '<UNK>', 'w',
'<UNK>', ',', '<UNK>', '<UNK>', '<UNK>', 'wąż', ',', '<UNK>', 'skrzydła',
'z', 'dżdżu', ',', 'A', '<UNK>', 'w', 'puszczy', ',', 'tuż', 'przy',
'<UNK>', ',', 'Straszny', 'wszczyna', 'szum', '.', '.', '.']
```

The Whitespace tokenizer struggled on this unseen text: nearly every second token was <UNK>. In contrast, GPT2-PL and SentencePiece handled the text seamlessly, demonstrating strong generalization capabilities.

6.2 Tokens per word and words encoded directly

For these metrics, we again use 11.6 MB of evaluation data that was unseen during model training. The functions used to calculate both metrics are included in the `results.ipynb` notebook. After parsing the text into words, a total of 1 678 418 words were identified. The results are summarized in the table below.

Tokenizer	Direct Tokens	Total Tokens	Avg. Tokens per Word
GPT2-PL	679 167	3 151 169	1.88
SentencePiece	1 418 822	2 036 981	1.21
Whitespace	1 678 418	1 678 418	1.00

Table 6: Comparison of tokenization statistics for GPT2-PL, SentencePiece, and Whitespace tokenizers.

As expected, the Whitespace tokenizer achieved exactly 1.0 token per word, since it generates one token for each word. In contrast, GPT2-PL produces the highest number of tokens, reflecting its finer-grained subword tokenization.

7 Summary

The Whitespace tokenizer is extremely efficient in terms of token generation, achieving a perfect 1:1 mapping between words and tokens. However, its major drawback is the inability to generalize to unseen text or to handle novel words, resulting in a high number of out-of-vocabulary tokens. This is clearly visible both in the OOV statistics and in the tokenized sample text.

GPT2-PL performs reliably, successfully tokenizing all input without producing unknown tokens. Nevertheless, it does not achieve the best word-level perplexity, likely due to being a general-purpose model not specifically adapted to the unique language of school literature. Another limitation is that training on GPT2-PL tokens takes the longest time, even though the tokens-per-second metric is not the worst—this is probably because it produces a higher number of tokens overall.

SentencePiece shows strong performance, effectively capturing the structure of complex words and achieving good perplexity. It ranks second in training time and has the lowest tokens-per-second rate, which may result from generating fewer tokens than GPT2-PL, as shown in Section 6.2.

In summary, while Whitespace is highly efficient, its practical utility is limited. Subword tokenizers like GPT2-PL and SentencePiece offer better generalization to unseen text, maintaining reasonable training efficiency while producing higher-quality representations.

References

[1] "Radlab GPT-2 tokenizer". In: (). URL: https://huggingface.co/radlab/polish-gpt2-small-v2.

[2] "More about radlab". In: (). URL: https://radlab.dev/.

[3] Taku Kudo. "SentencePiece: A simple and language independent subword tokenizer and detokenizer for Neural Text Processing". In: *EMNLP* (2018).

[4] "Let's build GPT: from scratch, in code, spelled out". In: (). URL: https://www.youtube.com/watch?v=kCc8FmEb1nY.