# Programming Task 1. Text Preprocessing, Representation

Submit your solutions **until 18.05.2025, 23:59 pm** on Moodle. The submission should include a single `zip` file with your group's solutions to the exercises.

**General Task:** The main goal of this assignment is to learn how to preprocess, tokenize and represent text data. You will also learn how to implement a Byte Pair Encoding (BPE) tokenizer (task 1) and train a SkipGram (sub-)word embedding model on the tokenized text data (task 2).

**Dataset:** For both tasks, you will be using the IMDB dataset, which contains 5000 polar movie reviews. You can find the data in the `imdb.txt` file. Each review is a single line in the file.
*Hint: It is recommended to look through the data before starting the implementation.*

**Suggestions and hints:**

- For both tasks, you are provided with function stubs in the `bpe.py` and `skipgram.py` files. Each function stub contains a docstring that describes the expected input and output. The functions that you need to implement are marked with the `TODO` comment.
- Your implementation should follow the logic described in the referenced lecture slides (NLP:III). To get points for the tasks, your implementation must pass the provided pytest test cases.
- **Start early:** This assignment involves implementing a non-trivial algorithms. Don't leave it until the last minute.
- **Understand BPE and SkipGram:** Before writing any code, make sure you understand the algorithms conceptually. Review the lecture slides and read the recommended literature (see below).
- **Use the provided test functions:** The provided test functions are designed to help you test your implementation. Use them to verify that your code works as expected.
- **PyTorch fundamentals:** This assignment requires understanding `torch.Tensor` operations, `nn.Module`, `nn.Linear`, `nn.LogSoftmax`, `nn.NLLLoss`, `torch.optim`, and the basic training loop (forward pass, loss calculation, backward pass, optimizer step). Refer to the official PyTorch tutorials and documentation if needed.
- **Start small:** Begin with a small subset of the data to test your implementation (The function `load_imdb_dataset` has a parameter `small_dataset` that allows you to load a smaller dataset for testing purposes.). Once you have a working version, scale it up to the full dataset.
- **Debugging:** If you encounter issues, use print statements or a debugger to inspect the values of variables at different stages of your code. This will help you identify where things might be going wrong. Take advantage of the provided test functions to check your implementation step by step.
- **Consult resources (but implement yourself!):** Besides the suggested readings, you can look up explanations of the algorithms online. However, the goal is to implement it yourself, so avoid directly copying code from external sources. Understanding the algorithms will help you in the long run, especially for the exam.
- **Questions:** If you have any questions or need clarification, please post your questions on Moodle.

**Task 1 (10 + 3 Points):** Implement the Byte Pair Encoding (BPE) algorithm from scratch (`bpe.py`). You need to implement the following methods:

(a) `pre_tokenize`: for normalizing and preprocessing a raw input string. The function should lowercase the input string, split it by whitespace and return a list of tokens.

(b) `preprocess`: for splitting the pretokenized text into characters.

(c) `_get_stats`: for counting the frequency of all pairs of adjacent sub-words in the text.

(d) `_merge_pair`: for merging the most frequent pair of adjacent sub-words in the text.

(e) `train`: for training the BPE tokenizer. The function should take a list of sentences as input and update the list of merge rules. `_tokenize_string`: for tokenizing a string using the learned merge rules. The function should take a single string as input and return a list of tokens.

(f) `tokenize`: for tokenizing a list of sentences using the learned merge rules. The function should take a list of sentences as input and return a list of lists of tokens.

(g) After implementing the methods, train the BPE tokenizer and save the learned merge rules by running: `python3 bpe.py`. Adjust the hyperparameters (number of merge rules, vocabulary size) to optimize the performance.

(h) In your submission, include your BPE implementation `bpe.py`, trained tokenizer vocabulary and merge rules `bpe_trained_group-<number>.json`, and test case results `bpe_test_group-<number>.txt`.

(i) For additional points you can improve the tokenizer by implementing the following:

(i1) Improve the `pre_tokenize` function: for normalizing and preprocessing a raw input string. You can implement one or more heuristics introduced in the lecture. The function should return a list of tokens. *Hint: Analyze the data and think about what kind of preprocessing might be useful.*

(i2) `_get_stats_with_threshold` (stub not provided): for counting the frequency of all pairs of adjacent sub-words in the text. The function should take a list of sentences as input and return a dictionary with the frequency of each pair of adjacent sub-words. The function should also take a threshold as input and only return pairs with a frequency above the threshold.

(i3) Improve the tokenizer by modifying the methods to include beginning-of-word (`<w>`), end-of-word (`<\w>`) tokens and/or subword prefixes (`##`) to distinguish between subwords and whole words.

**Task 2 (10 + 3 Points):** Implement the SkipGram (sub-)word embedding model from scratch (`skipgram.py`). You need to implement the following methods:

(a) `SkipGramTrainer` class:

(a1) `_generate_context_pairs`: for generating context pairs from a list of sentences. The function should take a list of sentences as input and return a list of tuples, where each tuple contains a target word and its context word within a given window size.

(a2) `cosine_similarity`: for calculating the cosine similarity between two vectors. The function should take two vectors as input and return the cosine similarity score.

(a3) `find_similar_tokens`: for finding the most similar tokens to a given token. The function should take a token and a `top_n` parameter and return a `top_n` list of most similar tokens from the vocabulary based on cosine similarity.

(b) `SkipGram` class:

(b1) `_to_one_hot`: for encoding a token as a one-hot vector. The function should take a token as input and return a one-hot encoded vector of vocabulary size.

(b2) `forward`: for performing a forward pass through the SkipGram model. The function should take a batch of token indices as input and return the probability distribution over the vocabulary for each token in the batch.

(b3) `get_token_embedding`: for looking up the embedding of a token. The function should take a token index as input and return the corresponding embedding vector.

(c) After implementing the methods, train the SkipGram model and save the learned embeddings by running: `python3 skipgram.py`. Adjust the hyperparameters (embedding size, window size, number of epochs, learning rate). You can use the provided test functions to test each method implementation.

(d) In your submission, include your SkipGram implementation `skipgram.py`, trained model `skipgram_group-<number>.pt`, and test case results `skipgram_test_group-<number>.txt`.

(e) For additional points you can improve the model by implementing the following:

(e1) Improve the `_generate_context_pairs` function to generate context pairs with a certain frequency threshold as a parameter. The function should also take a threshold as input and only return pairs with a frequency above the threshold.

(e2) Modify the model and trainer to support negative sampling. The function should take a list of sentences as input and return a list of tuples, where each tuple contains a target word and its context word within a given window size. The function should also take a threshold as input and only return pairs with a frequency above the threshold. Submit your implementation as a separate file `skipgram_negative.py`.

(f) *Optional:* Since the resulting embeddings are high-dimensional, you can visualize them using dimensionality reduction techniques such as PCA or t-SNE. You can use the `sklearn` library for this purpose. Or you can use an online visualization tool from Google: Embedding Projector [link]. To load your embeddings, you should first save them as two `tsv` files: one for the words and one for the vectors. Then, go to the Embedding Projector, click on "Load", and upload your files.

**Recommended Readings:**

- **BPE:**
  - Lecture slides. Text Models > Text Preprocessing
  - Rico Sennrich, Barry Haddow, and Alexandra Birch. 2016. Neural Machine Translation of Rare Words with Subword Units. In Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 1715–1725, Berlin, Germany. Association for Computational Linguistics.
  - Daniel Jurafsky, James H. Martin. 2025. Chapter 2: Regular Expressions, Tokenization, Edit Distance. Section 2.5 Word and Subword Tokenization. In Speech and Language Processing (3rd ed.).
- **SkipGram:**
  - Lecture slides. Text Models > Text Representation; Text Models > Text Similarity
  - Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient Estimation of Word Representations in Vector Space. In Proceedings of Workshop at ICLR.
  - Daniel Jurafsky, James H. Martin. 2025. Chapter 6: Vector Semantics and Embeddings. In Speech and Language Processing (3rd ed.).