1. Name: Vitaliy Alifanov
2. Email: v.alifanov@innopolis.university
3. Nickname: ADIS
4. GitHub: https://github.com/1ADIS1/Inno-NLP

Baseline solution - vocabulary

```
if text in vocab:
    label = vocab[text].most_common(1)[0][0]
    ners.append([start_i, end_i, label])
```

The provided solution demonstrates a classification of NERs by utilising the SpaCy library, specifically the Russian language model (`ru_core_news_lg`), to process a dataset of test sentences and identify named entities based on a vocabulary dictionary built from training data. This vocabulary dictionary counts all labels for each token from the training data, allowing the model to extract and classify entities based on the most common label for each token.

This approach has a 0.34 f1 score. Why is it so low?

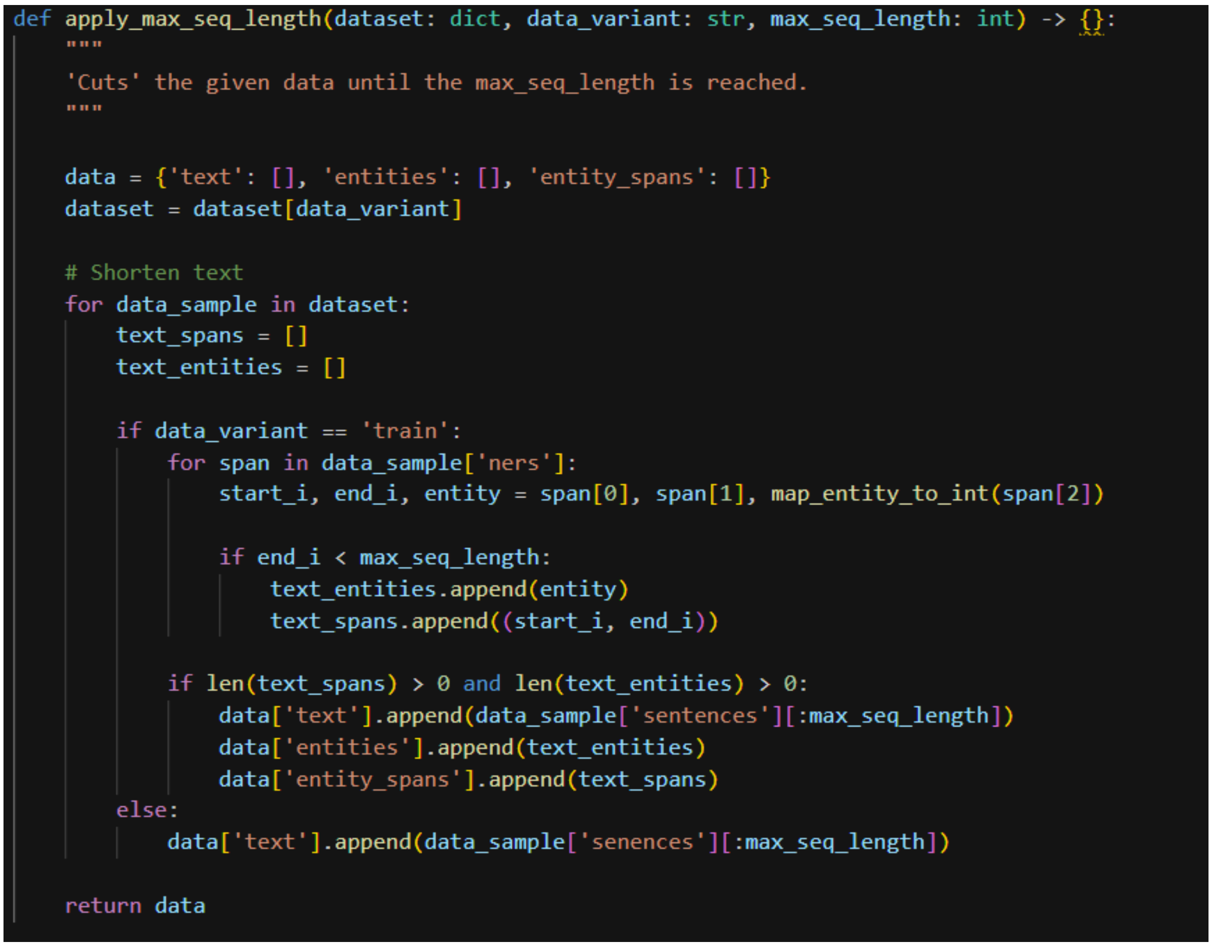
1. Train data provided for the assignment has around 500 samples, which is very low to build such a vocabulary model.
2. Choice of tokenizer does matter, there could be room for a grid search of the best russian text tokenizer.
3. Vocabulary solution is rigid by definition, and cannot capture nor nested entities, nor more complex abstractions.
4. As a baseline it is very simple and achieves good results, in my opinion.

At this point, I would like to demonstrate another solution in the next page.

LUKE and BERT solution

1. Preparing the Data

The first step is to prepare the data for training. This involves tokenizing the text and mapping entity spans to their corresponding entity types. The provided code snippet demonstrates how to do this for a dataset that includes named entity recognition (NER) data.



```python
def apply_max_seq_length(dataset: dict, data_variant: str, max_seq_length: int) -> {}:
    """
    'Cuts' the given data until the max_seq_length is reached.
    """

    data = {'text': [], 'entities': [], 'entity_spans': []}
    dataset = dataset[data_variant]

    # Shorten text
    for data_sample in dataset:
        text_spans = []
        text_entities = []

        if data_variant == 'train':
            for span in data_sample['ners']:
                start_i, end_i, entity = span[0], span[1], map_entity_to_int(span[2])

                if end_i < max_seq_length:
                    text_entities.append(entity)
                    text_spans.append((start_i, end_i))

            if len(text_spans) > 0 and len(text_entities) > 0:
                data['text'].append(data_sample['sentences'][:max_seq_length])
                data['entities'].append(text_entities)
                data['entity_spans'].append(text_spans)
        else:
            data['text'].append(data_sample['senences'][:max_seq_length])

    return data
```

- **Tokenization**: The `AutoTokenizer` from the `transformers` library is used to tokenize the text. This is crucial for converting the raw text into a format that the model can understand.
- **Mapping Entities**: The `map_entity_to_int` and `map_int_to_entity` functions are used to map entity types to integers and vice versa. This is necessary because the model requires numerical inputs for training.
- **Applying Max Sequence Length**: The `apply_max_seq_length` function is used to ensure that all text samples are within a specified maximum sequence length. This is important for maintaining consistency in the input data and for computational efficiency.

2. Setting Up the Model

The LUKE model is set up for entity span classification. This involves loading the pre-trained LUKE model and configuring it for the specific task of NER.

- **Loading the Model**: The `LukeForEntitySpanClassification` class is used to load the pre-trained LUKE model. The `num_labels` parameter is set to the number of entity types in the dataset, and `ignore_mismatched_sizes` is set to `True` to allow for different sequence lengths.
- **Device Allocation**: The model is moved to the appropriate device (CPU or GPU) for training.

3. Training the Model

The training process involves iterating over the dataset, feeding the tokenized text and entity spans into the model, and updating the model's weights based on the loss.

- **Loss Function**: The `CrossEntropyLoss` is used as the loss function, which is suitable for classification tasks.
- **Optimizer**: The `AdamW` optimizer is used for updating the model's weights. The learning rate is set to `1e-5`.
- **Training Loop**: The training loop iterates over the dataset, tokenizes the text, and feeds it into the model. The model's outputs are compared with the true labels to compute the loss, which is then used to update the model's weights.

Conclusion

The combination of LUKE for classification and BERT for entity extraction is more flexible in terms of learning underlying hidden dependencies in the text, which the vocabulary solution lacks.