Tamás Gyenis

hupunct – Finetuning BERT for Automatic Punctuation Restoration in Hungarian

Diploma Thesis

Supervisor:
András Lukács

Budapest, 2023

TABLE OF CONTENTS

1 Introduction 1
2 Related Work and Proposed Solution 2
2.1 Solutions in English 2
2.2 Solution In Hungarian 3
2.3 Proposed Solution 3
3 Methodology 5
3.1 Transformers and BERT 5
3.1.1 Transformers 5
3.1.2 BERT 8
3.2 Dataset Preparation 9
3.3 Data Pipeline 10
3.4 Experiment Setup 13
3.5 Initial Experiments 15
3.5.1 First Experiment 15
3.5.2 Hyperparameter Tuning 18
3.6 Final Training 20
4 Results 22
4.1 Evaluation of The Results 22
4.2 Conclusion 23
5 References 24
6 Appendix 26
6.1 Bad Examples 26
6.2 Good Examples 27

1 INTRODUCTION

NLP or Natural Language Processing is a rapidly evolving field, with huge amounts of research and investment. With the internet becoming a standard for day-to-day communication for individuals and businesses alike, vast amounts of text data is being generated. NLP models aim to extract meaningful information from this data. Computers can analyze text and execute tasks ranging from basic spell-checking and machine translation to semantic tasks, like summarization, sentiment analysis or even chatbots. Large Language Models (LLM) are even capable of performing a multitude of the above-mentioned NLP tasks, think GPT4 [1] for example. These advances in the field of NLP allow us to automate repetitive tasks and processes, or to enhance and aid many creative processes.

Besides written communication, the digestion and interpretation of the spoken language is also starting to become mainstream for machines. Speech2Text or Automatic Speech Recognition (ASR) is new way for humans to interact with technology. ASR systems have become increasingly sophisticated, employing advanced algorithms and deep learning. They can provide input to a variety of NLP pipelines allowing for applications in various domains, including transcription services, voice assistants, call center automation, language learning, and hearing aid accessibility solutions, just to mention a few.

Despite the huge advancements of the mentioned fields some gaps of functionality still remain today. One shortcoming of many ASR systems, is that they are only able to output a stream of text without punctuation, showing, it can be very hard to extract the intricate or subtle information about punctuation placement in spoken language. This can leave the usage of ASR undesirable for some applications. The motivation for this work originated from an attempt to transcribe an interview discussion for a Budapest-based program magazine that has the interview as one of its more prominent formats. Currently, a writer or editor needs to listen to a recorded interview and manually type in the – often very lengthy – conversation, which can take multiple hours of tedious work. When asked about the possible usage of ASR, the feedback was that it is even more work to first transcribe the interview using a transcription service and then manually restore punctuations and correct misunderstood words, before proceeding with the editing of the text. We can look at an actual 30 second interview segment that was transcribed by Google Cloud [2], a presumably state of the art ASR service:

```
'egy ilyen nagyon névleges összegért kellene viszonylag értékesebb
átválogatott könyveket az emberek kezébe adni ugye az olvasási
programnak az lenne a lényege hogy egyrészt a fiatalok kezébe könyvet
adjunk mert ugye ebben az online világban akkor úgy látszott 2014 ben
kezd elmenni egy kicsit ez a hangoskönyvet könyvek irányába a dolog és
ez nagyon sajnáltuk volna most azt kell már hogy mondjam nem egészen
így van tehát a könyvnek mindig megvan a maga varázsa megvan az illata
kézbe lehet fogni magamon látom ha sokat olvasom a számítógéppel
megjelenített leveleket és a telefont okoseszközöket az ember szeme
elfárad már estére'
```

We can see that the text is missing punctuations entirely, including hyphens, making it harder to read and interpret. This is not only a drawback for human readers, who try to format this into a sensible text, but also to other possible downstream NLP tasks trained on punctuated text, which is the majority. In this project, I aim to fill the gap mentioned above by developing a model for Automatic Punctuation Restoration (APR) in Hungarian called '**hupunct'**, that has raw unpunctuated lower-cased text as its input, and has the corrected, punctuated text as its output. The solution is based on a widely used NLP technique, which involves the finetuning of a pretrained special deep neural network, a Transformer.
In the next chapter, I explore previous research and available literature, mentioning solutions in English and in Hungarian. In Chapter 3, after briefly discussing the used base model and its architecture and intuitions, I talk about my methodology to solve the APR problem, available datasets, the constructed data and training pipeline and my experiment setup. Finally, in Chapter 4, I present my results and draw a conclusion.

2 RELATED WORK AND PROPOSED SOLUTION

2.1 SOLUTIONS IN ENGLISH

There have been many models proposed for the task of APR. They mostly use lexical features of text datasets as reference or training data, which can be easily gathered from the internet (Wikipedia, web articles and so on). For the task of APR in non-Hungarian languages solutions range from early models using n-grams [3] and conditional random fields (CRF) [4] to more advanced deep learning based models, like Long Short-Term Memory (LSTM) [5], Convolutional Neural Network (CNN) [6] and finally the state of the art: Transformers [7]. It's hard to compare vanilla models' performances, because of the difference in their test dataset, but generally deep learning-based approaches outperform vanilla models because of their ability to learn more complex features corresponding to wider context that is more sparsely distributed in the text. On the other hand, most modern models are benchmarked on the popular IWSLT2011 [8] dataset, and so their performances can be compared. Some of the state-of-the-art model benchmarks can be seen on *Figure 1.* We can see a wide range of scores (Overall - F1 is the most relevant), ranging from 65.1 % for Recurrent Neural Network based (RNN) models up to 80+% for transformer-based models. The current state-of-the-art model, codenamed FF2 for Feature Fusion two-stream, uses two main models that operate on the same input token stream: one larger pretrained transformer encoder to capture semantic features and one smaller non-pretrained transformer encoder-decoder to capture the task specific features. Then, the two streams are combined in a fusion layer before the classification head. The approach proposedly mitigates the bias of the large pretrained transformer encoder towards the punctuated training dataset. We can see on *Figure 1* that this approach brings around 2% improvement over purely pretrained-transformer-based models. We must note though, that these scores only hold for the simplest

APR task consisting of only 3 punctuation classes: Comma, Perion and Question-mark. Furthermore, although the approaches are essentially the same in most languages which have punctuation, the above-mentioned approaches focus on English language APR.

Model	Comma			Period			Question			Overall		
	P	R	F1	P	R	F1	P	R	F1	P	R	F1
BLSTM-CRF	58.9	59.1	59.0	68.9	72.1	70.5	71.8	60.6	65.7	66.5	63.9	65.1
Teacher-Ensemble	66.2	59.9	62.9	75.1	73.7	74.4	72.3	63.8	67.8	71.2	65.8	68.4
DRNN-LWMA-pre	62.9	60.8	61.9	77.3	73.7	75.5	69.6	69.6	69.6	69.9	67.2	68.6
Self-attention-word-speech	67.4	61.1	64.1	82.5	77.4	79.9	80.1	70.2	74.8	76.7	69.6	72.9
CT-Transformer	68.8	69.8	69.3	78.4	82.1	80.2	76.0	82.6	79.2	73.7	76.0	74.9
SAPR	57.2	50.8	55.9	**96.7**	**97.3**	**96.8**	70.6	69.2	70.3	78.2	74.4	77.4
BERT-base+Adversarial	76.2	71.2	73.6	87.3	81.1	84.1	79.1	72.7	75.8	80.9	75.0	77.8
BERT-large+Transfer	70.8	74.3	72.5	84.9	83.3	84.1	82.7	93.5	87.8	79.5	83.7	81.4
BERT-base+FocalLoss	74.4	77.1	75.7	87.9	88.2	88.1	74.2	88.5	80.7	78.8	84.6	81.6
RoBERTa-large+augmentation	76.8	76.6	76.7	88.6	89.2	88.9	82.7	**93.5**	87.8	82.6	83.1	82.9
RoBERTa-base	76.9	75.4	76.2	86.1	89.3	87.7	88.9	87.0	87.9	84.0	83.9	83.9
RoBERTa-large+SCL	78.4	73.1	75.7	86.9	87.2	87.0	**89.1**	89.1	89.1	**84.8**	83.1	83.9
FT+POS	**78.9**	78.0	78.4	86.5	93.4	89.8	87.5	91.3	**89.4**	82.9	85.7	84.3
Disc-ST	78.0	82.4	80.1	89.9	90.8	90.4	79.6	93.5	86.0	83.6	86.7	85.2
FF2 - w/o TNP	76.3	81.9	79.0	89.3	90.8	90.0	79.6	93.4	85.9	82.4	86.5	84.4
FF2 - w/o Interaction	78.2	83.6	80.8	89.7	89.5	89.6	78.1	93.5	85.1	83.5	86.7	85.0
FF2	78.8	**82.7**	**80.7**	89.3	90.7	90.0	79.6	**93.5**	86.0	83.8	**86.8**	**85.3**

***Figure 1**: Results in terms of precision (P %), recall (R %), and F1-score (F1 %) on the English IWSLT2011 test set. [7]*

2.2 Solution In Hungarian

For APR models trained on Hungarian corpora specifically, the literature focusing on deep learning-based solutions is very sparse. In the only relevant paper [9], the authors use the transformer finetuning approach to train models for both English and Hungarian APR. They don't benchmark their English model on the IWSLT2011 dataset, but they report similar results compared to the then state-of-the-art transformer models, based on the newer IWSLT Ted Talks dataset [10] benchmark. For the Hungarian models, the authors used the Szeged Treebank [11] dataset to train and test a multilingual BERT (mBERT) model and a Hungarian BERT model, huBERT [12], that was pretrained only on Hungarian corpora, and was supplemented with a fully connected classification head to produce a probability distribution over the punctuation classes. They found that in all cases the huBERT based model proved superior, reaching a macro F1 score of 82.2 on the test data averaging individual scores of the three classes, surpassing the mBERT based model by 12.2 percentage points.

2.3 Proposed Solution

Although some models outperformed finetuned pretrained transformer models recently, still, the finetuning approach offers considerable benefits, like transfer-learning, reduced data requirements, and ease of implementation and deployment,

thanks to many common interfaces through unified APIs, like HuggingFace [13]. For this reason, I chose to solve the problem of punctuation restoration in Hungarian with the pretrained transformer finetuning technique. Since the state-of-the-art model for Hungarian semantic language understanding is still huBERT, I as well, chose it as my base model. The prior Hungarian work is a good baseline, but I aim to improve on it in many aspects. The most notable improvements are as follow:

1. **Different dataset**: Instead of the Szeged Tree Bank, I use a dataset generated from the Hungarian Web Corpus 2.0 [14] that contains over 9 billion words, which is ~7500 times more than in the Szeged Tree Bank. This opens up many possibilities.

2. **More punctuations**: I am extending the predictable punctuations with *exclamation* (!), *hyphen* (-), *colon* (:) and *quote* (“). I am also adding the upper cased versions of all these, including *empty* ( ), meaning the model will be able to automatically capitalize words as well. This is now possible due to the considerable number of examples in the new dataset, increasing the number of classes from 3+1 to 15+1.

3. **Practical high level data representation**: For the Hungarian APR model, they used a sentence based high representation and they shuffled sentences in the dataset before splitting. This can result in training and testing examples with unrelated subsequent sentences, which is very impractical. First, it creates a bias towards easy-to-predict sentence boundaries, since the context of the sentences can be very different. Second, it contradicts real-world application, where the context hardly ever changes from sentence to sentence. To address this, I use a higher-level representation of ‘articles’, which I discuss in a later chapter.

4. **One-shot prediction**: In the paper the authors mention a multi-shot prediction approach, where they improve prediction performance by sliding a window over the example sequence using various overlaps and performing prediction on the sequence slice inside the window, essentially creating multiple predictions for the given token. For training this can increase the number of possible examples generated from the dataset (because of the overlap) and for prediction it allows for an ensemble-like method of producing a final prediction. There is no need for this with a large enough dataset, since it has the drawback of drastically reducing inference speeds. I aim to reach a good performance using only one prediction per token during training and inference.

3 METHODOLOGY

In this following chapter, I discuss the important aspects of the used base model, BERT, then I continue with the presentation of the dataset and data preprocessing, after which I talk about the training pipeline and experiment setup. I aim to provide a practical solution and to have a working, high performance model in the end, which can be shared with the community.

3.1 TRANSFORMERS AND BERT

This summary is based on chapter 10 and 11 of the book *Speech and Language Processing* [15], and the paper *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding* [16].
For many years the predominant choice for NLP problems were RNNs. RNNs were specifically designed to handle sequential data and were widely adopted for various NLP tasks. One of the key advantages of RNNs compared to feed forward models like fully connected networks (FCN) or convolutional networks (CNN) is their ability to capture dependencies and relationships between elements of a sequence, maintaining hidden states that carry information from previous steps and allow them to model context and temporal dependencies. However, RNNs come with certain limitations. One major challenge is the sequential nature of their computation, which reduces parallelization capabilities and can lead to slower training and inference. This limitation becomes more pronounced as the length of the input sequence increases, since all the states must be kept in memory. Additionally, RNNs may struggle to capture long-range dependencies due to the difficulty of information propagation over many time steps, even with the presence of LSTM or GRU cells.

3.1.1 TRANSFORMERS

The introduction of the Transformer [17] architecture, was a major step forward in the field of NLP. The original Transformers employed an encoder-decoder architecture, mapping sequences of input embedding vectors $(\mathbf{x}_i \dots \mathbf{x}_j)$, to sequences of output vectors $(\mathbf{y}_i \dots \mathbf{y}_j)$ using transformer blocks. The main innovation of transformers is the usage of **self-attention** layers. It allows the transformer to process input sequences in parallel, meaning the model has access to all the inputs at once, and the computation for each item is carried out independently of all the other computations. This enables digesting longer input sequences and learning complex dependencies, even between distant tokens. Self-attention essentially tries to estimate the relevance of the given token embedding in the current context. One form of self-attention is **scaled dot-product self-attention**, where the importance score of a given token in the current context is derived using the dot-product of query and key vectors, that represent the current focus of attention and the

preceding input. The calculation of the importance score for a single token with index *i* is:

$$\text{score}(\mathbf{x}_i, \mathbf{x}_j) = \frac{\mathbf{q}_i \cdot \mathbf{k}_j}{\sqrt{d_k}}$$

, where $\mathbf{q}_i = \mathbf{W}^{\mathbf{Q}}\mathbf{x}_i$, $\mathbf{k}_i = \mathbf{W}^{\mathbf{K}}\mathbf{x}_i$ are the query (q) and key (k) vectors respectively and d_k is the dimension of the key vectors. The $\mathbf{W}^{\mathbf{Q}}$ and $\mathbf{W}^{\mathbf{K}}$ matrices are transformation matrices that project the input vectors to the query and key vectors. They are of shape $(d_{\text{model}}, d_{model})$, where d_{model} is the maximum embedding dimension. The final output is:

$$\mathbf{y}_i = \sum_{j=0}^{i} softmax(\text{score}(\mathbf{x}_i, \mathbf{x}_j))\, \mathbf{v}_j$$

, where $\mathbf{v}_j = \mathbf{W}^{\mathbf{V}}\mathbf{x}_i$. The softmax is used for normalization and the division with d_k in the score is used for preventing extremely large values of the otherwise uncapped dot product to cause numerical problems and loss of gradients during training. The calculation is illustrated on *Figure 2*.

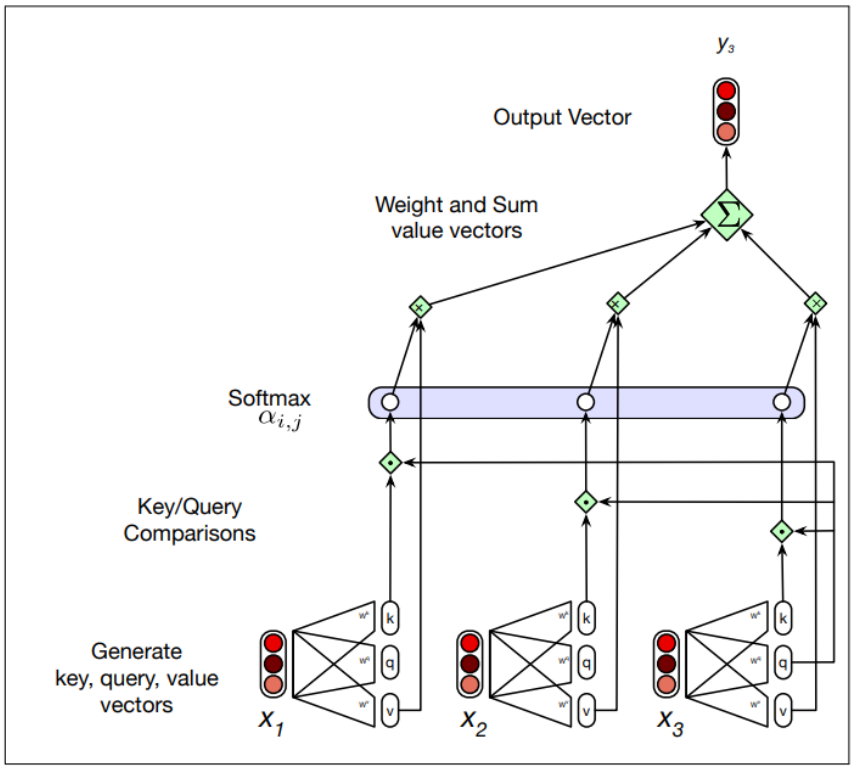

***Figure 2**: Calculating the value of y3, the third element of a sequence using causal (leftto-right) self-attention. [15]*

Since each output $\mathbf{y}_i$ can be computed independently, we can calculate query and key matrices instead of vectors by packing *n* input embeddings together into input matrix $\mathbf{X} \in \mathbb{R}^{n \times d}$ and using matrix multiplication instead of matrix-vector multiplication, thus parallelizing the calculation:

$$\mathbf{Q} = \mathbf{X}\mathbf{W}^{\mathbf{Q}}; \mathbf{K} = \mathbf{X}\mathbf{W}^{\mathbf{K}}$$

We also calculate a value matrix $\mathbf{V} \in \mathbb{R}^{n \times d} = \mathbf{X}\mathbf{W}^{\mathbf{V}}$. Finally, the self-attention layer becomes:

$$SelfAttention(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = softmax\left(\frac{\mathbf{Q}\mathbf{K}^{\mathbf{T}}}{\sqrt{d_k}}\right)\mathbf{V}$$

This way the score for every key and value is included. Depending on the training objective, we have to mask the corresponding elements of the $\mathbf{Q}\mathbf{K}^{\mathbf{T}} \in \mathbb{R}^{n \times n}$ importance matrix that we don't want the model to 'pay attention' to. For example, if our training objective is to predict the word(s) following the query, then we have to mask the upper-triangle portion of the matrix corresponding the input tokens following the query. Therefore, we must pass an attention mask along with the input tokens to the Transformer models when specifying the training objective.
Besides the SelfAttention, the transformer block is composed of FeedForward and Normalization layers. FeedForward layers are made of densely connected neurons adding to the network capacity, while Normalization layers improve training performance by normalizing the tensor to which they are applied to, keeping extreme values closer to zero mean and with unit variance. This combats large gradient changes, which speeds up and stabilizes training. Additional residual connections help with propagating useful information to deeper parts of the network. They were introduced in deep CNNs used for image processing and were proven very powerful. The schematic of a transformer block can be seen on *Figure 3*.

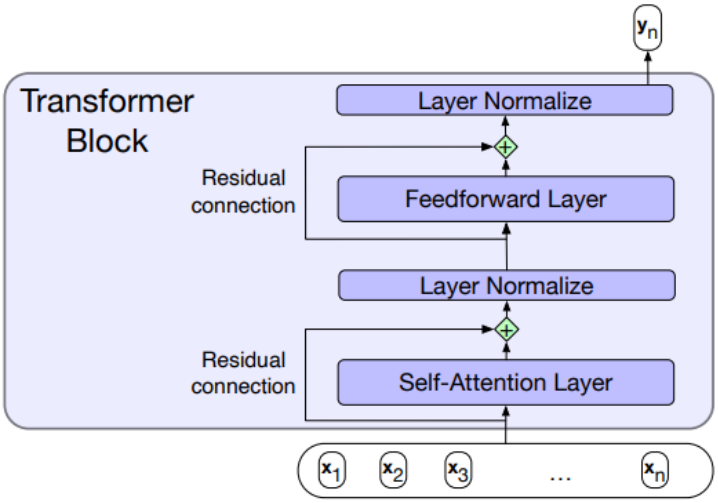

***Figure 3**: A transformer block showing all the layers. [15]*

Within a sentence, multiple words can establish various simultaneous relationships with one another. Verbs and their arguments, for instance, can exhibit distinct syntactic, semantic, and discourse connections. For a single transformer block to effectively capture all these diverse parallel relations among its inputs would be

challenging. To address this, instead of simple SelfAttention layers, transformers employ **multihead-attention**. Multihed-attention works by employing SelfAttention layers in parallel, each with their own keys, queries and values, and then a projection is used to transform their outputs to the original input shape. These layers allow for the exploration and learning of different kinds of relationships in parallel, enhancing the model's ability to comprehend the intricate connections within the sentence. The last consideration is, that unlike RNNs, transformers have no inherit information about word positions. This is addressed by passing another set of embeddings to the model, along with the word embeddings, which are the positional embedding. Both are randomly initialized and can be learned along with the other learnable parameters of the model.

3.1.2 BERT

BERT, or Bidirectional Encoder Representations from Transformers, builds on the original transformer with some structural differences and a different training objective. The model is bidirectional, because it receives all the inputs at once from both left and right of a given index, allowing to contextualize from any part of the input sequence, compared to only the left side in case of the original transformer. BERT is specifically designed for fine-tuning. Fine tuning is a form of transfer learning, where a high-capacity, powerful language representation model is used as a base for various NLP tasks after pretraining. In the case of BERT, the language model, which is a transformer encoder with bidirectional attention is pretrained using the Masked Language Model (MLM) training objective. Instead of predicting the next word, as in the case for the original transformers, MLMs are tasked to correctly predict a number of randomly masked tokens, that can be anywhere in the input sequence, essentially filling in the blank spaces. In case of BERT, 15% of the input tokens are changed by masking the majority using the [MASK] token, and also changing some tokens to a random different token in the dictionary. The idea behind fine-tuning, is that the representations learned during the pretraining, are still useful for the downstream task. For example, the learned embeddings are very meaningful and maybe the weights of the model don’t have to change that much when we are training on the finetuning task. This help drive down training time considerably, since pretrained model have a ‘head start’ on the finetuning task. This idea is supported by the literature, since BERT achieved state-of-the-art performance in many NLP benchmarks for question answering, sentiment analysis, natural language inference, and named entity recognition, among others. The Hungarian BERT model huBERT was trained on the Hungarian Webcorpus 2.0 using the same MLM technique, and it surpasses other multilingual BERT models according to the huBERT paper. The model is also made available on the HuggingFace Model-Hub. In the following chapters I present my solution to the APR problem using this model as a base.

3.2 Dataset Preparation

The dataset used for training the **hupunct** model is generated by filtering and transforming the new Hungarian Webcorpus. The corpus was built from the Common Crawl [18] and consist of over 9 billion words. It has 3 versions: text, clean, and analyzed, where the text version contains the raw documents in text files, the clean version has the words tokenized with lemma and POS tag assigned and is in tsv format, while the analyzed version has all morphological analyses. For my project, I selected the text corpus, because that contains no additional information that is not needed for the task, while having the smallest size (25GB vs the 511GB analyzed corpus) and should be easily converted to my target data representation. The corpus has ordinary web data from 2017 to 2019 and has data from the Hungarian Wikipedia as well.

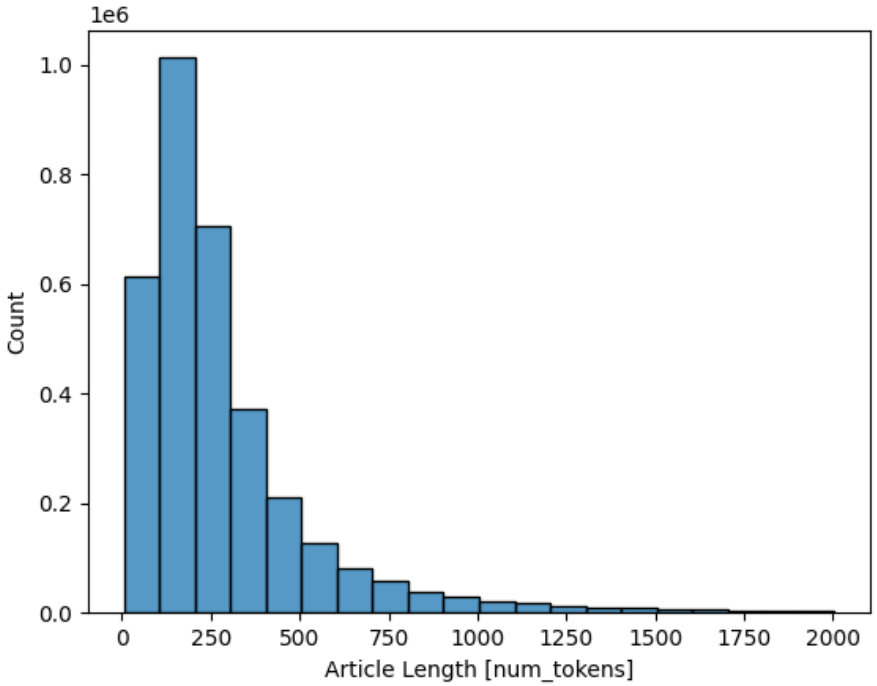

***Figure 4**: Distribution of article lengths in the full dataset.*

In the raw text files, each line contains a separate sentence and documents are separated by additional line break ("\n"). We generate the dataset for APR by iterating though the text files in random order (for an initial shuffle) and extracting articles based on some filtering criteria. We check each line in an article candidate by filtering every element of the string for allowed characters, while also converting non-standard but interpretable characters and punctuations to standard ones. After this general filtering of the string, we only add it to the article if it ends with one of the predefined EoL (End of Line) symbols. This is to filter out any lines that aren't part of a sentence, like dates of web posts, headers, document ID-s and so on. It can still happen, that the line has some unrelated content at the start. We filter this out as well by deleting the first character until we find a valid one. Our articles are strings made from valid subsequent sentences with the minimum character count

of 500. This ensures that an article has only context-related sentences originating from the same document **in order**. Our training examples are going to be generated from every article separately, which is important because we want our model to find sentence borders within the same context in practice. The chosen file format to store the dataset is also txt, because it has a nice property of allowing reading line by line, while also being human-readable, in contrast to binary formats like parquet or hdf.
We write each article as a separate line to one output txt-s of "train", "eval" and "test" based on a probability distribution set by the *splits* variable. We create two datasets: a full one with all the articles selected and a smaller one, which is around 2% of the full one in article number. This set is going to be used for testing the data pipeline and the initial experiments. Splits in the full dataset are (99%, 0.5%, 0.5%) and (80%,10%, 10%) in the mini dataset. The total number of articles in the full dataset is 3.4 million. The article-length distribution of the full dataset can be seen on *Figure 4.*

3.3 Data Pipeline

Efficient data loading is handled with the help of the HuggingFace (HF) Datasets API. In order to create examples with our training features and targets we use the generator support to create our HF Dataset object. For this, we must define our training objective.
We can interpret APR as a seq-to-seq type task (like machine translation) or as a token classification task. For a seq-to-seq approach we would have our inputs mapped to an arbitrarily long sequence of outputs (length would be num_input_tokens + num_predicted_punctuations in theory), where the outputs have the correct punctuations restore. In case of the token classification approach, we would predict a punctuation class for every input token. E.g., if we have classes NONE, COMMA and QUES the following classes should be predicted to restore the input to "Hello, how are you?":

```
Input:
['hello', 'how', 'are', 'you']

Output labels:
['COMMA', 'NONE', NONE, 'QUES']
```

In the case of token classification, we need to collapse all classes to their respective input tokens. This means, for example, if we would also like to handle capitalization in one model, we need to create hybrid classes. In the above case the COMMA would have to become UPPER+COMMA. Consequently, the number of possible class combinations can get exponentially large and some of those classes would hardly occur in the dataset making a very unbalanced classification problem. On the other hand, the seq-to-seq approach has its own drawbacks, like the possibility

of ungrammatical output in case of harder inputs, which can be a challenge to notice or correct. We also lose the nice property of having a fixed-size output. Also, a seq-to-seq model would require a separate decoder network which can take more effort to train than only the BERT model with a simple classification head. Based on these considerations, we solve APR as a token classification problem.
We limit the number of punctuation classes to 15+1 (see *Table 1*) including only the most frequent ones. The rarest class of UPQUES still has 1000+ occurrences in the test set. The class labels are supplemented with BIO tags [19], traditionally used for Named Entity Recognition (NER), the most popular example of token classification. BIO tags allow us to use some convenient built-in functionality of HF later. Since we don't have labels that span multiple input tokens (unlike in NER), we only use the Beginning and Out tags.

Char	Class Label	Class ID	Train Count	Eval Count	Test Count
none	O	0	760 479 309	3 881 129	3 744 499
,	B-COMMA	1	89 645 306	454 259	438 984
.	B-DOT	2	53 933 996	276 029	263 820
!	B-EXCLAM	3	2 640 463	13 553	13 066
?	B-QUES	4	2 769 419	14 066	13 805
-	B-HYPHEN	5	16 354 614	83 774	81 188
:	B-COLON	6	4 299 129	22 358	20 939
"	B-QUOTE	7	4 427 826	21 978	22 418
+	B-UPPER	8	110 999 887	571 543	543 620
.+	B-UPDOT	9	8 513 335	42 671	42 063
,+	B-UPCOMMA	10	3 479 712	18 018	16 749
!+	B-UPEXCLAM	11	510 176	2 427	2 591
?+	B-UPQUES	12	234 527	1 114	1 076
-+	B-UPHYPEN	13	5 914 444	30 202	29 715
:+	B-UPCOLON	14	1 069 835	6 372	4 949
"+	B-UPQUOTE	15	609 640	3 058	3 101

***Table 1**: Summary of the punctuation classes and their distribution in the train eval and test sets.*

We create our HF Dataset by yielding training examples from a custom generator. The generator operates on a file pointer pointing to one of our dataset text files with the articles represented line by line. For each article we first word-tokenize the string using the simple nltk [20] tokenizer. This returns a list of word tokens including the punctuation characters. We iterate over this list and formulate our feature target pair by checking the subsequent word, and if that is in a mapping dictionary with our possible punctuations then we also check if the current word is capitalized and we add a target class ID based on this criterion and the mapping defined. Hyphenated words are not separated by nltk, so we need to manually split them and then recurse back to the same logic as above. For our feature tokens we always append the lower-case version of the current word. Using this method, we generated a label for every word in our article. Let's have an example:

```
Input article:
‘Szeretem a Deep-learninget és a nehéz feladatokat, ezért leszek
adattudós!’

Output feature tokens:
['szeretem', 'a', 'deep', 'learninget', 'és', 'a', 'nehéz',
'feladatokat', 'ezért', 'leszek', 'adattudós']

Output target labels:
[8, 0, 13, 0, 0, 0, 0, 1, 0, 0, 3]
```

The last step is to yield a dictionary with our training example. Since we are using a generator-based Dataset builder, the process is done lazily, so there are no memory concerns. Also, if set, the HF Dataset can be cached to an Apache Arrow binary, making subsequent loadings much-much faster. From the raw dataset we can generate 3+ million training examples with 300+ million token-label pairs. This representation is not-yet digestible by our model though. We have word-based tokens, but BERT uses subword tokens. Using a subword tokenizer we may need a larger vocabulary and some additional pre- and post-processing, but we get the benefit of having subword representations. This is particularly useful for Hungarian, since it is an agglutinative language, which uses many affixes and morphemes for a number of grammatic functions. With subword tokenizers we can have meaningful embeddings from the subword units, as well as the ability to split out-of-vocabulary words into subword tokens that are present in our vocabulary. An example output of the huBERT tokenizer:

```
Input:
‘Menjünk el Lacihoz a fűnyírójáért!’

Word tokens:
['menjünk', 'el', 'lacihoz', 'a', 'fűnyírójáért']

huBERT tokenizer subword tokens:
['[CLS]', 'menjünk', 'el', 'la', '##ci', '##hoz', 'a', 'fű', '##nyí',
'##ró', '##j', '##áért', '[SEP]']
```

The word ‘fűnyírójáért’ is not in the vocabulary of huBERT, but it can be split into some subwords that are. BERT tokenizers also have [CLS] and [SEP] tokens. These can be useful for other fine-tuning tasks, such as question answering or sequence classification, but in our case, these can be ignored and masked.

At this point we are presented with the problem of our subword tokens not matching in length with our class-ID labels. For this, we need a way to broadcast the labels to match the subwords. Thankfully, by using a tokenizer with the *fast* attribute, we can get access to the mapping of the subword tokens to the original word ID-s (or indices). This mapping can help us to create a function that can realign the labels. In actuality, the subword tokenization and label broadcasting are

encompassed in a custom pre-processing function *tokenize_and_align_labels,* which is then mapped to our Dataset. The function also assigns the class-ID of '-100' to the labels, that we don't want to contribute to the loss during training, which are the CLS and SEP tokens in this case. Broadcasting can be observed in the below example:

```
Original assigned class IDs:
[8, 0, 0, 8, 0, 3]

huBERT tokenizer subword tokens:
['[CLS]', 'menjünk', 'el', 'la', '##ci', '##hoz', 'a', 'fű', '##nyí',
'##ró', '##j', '##áért', '[SEP]']

Word ID mapping:
[None, 0, 1, 2, 3, 3, 3, 4, 5, 5, 5, 5, 5, None]

Broadcasted class IDs:
[-100, 8, 0, 0, 8, 8, 8, 0, 3, 3, 3, 3, 3, -100]
```

In case of BERT, the maximum number of input tokens is capped at 512, due to memory concerns. This means, we need to pad shorter examples and truncate longer ones to match the model input length. Shorter examples are padded with '-100' until the number of tokens reach 512, and longer ones are truncated in a way that all the chunks are added as new examples, so there is no loss of training data. This is a limitation of the original BERT and of huBERT consequently. The input size cap limits the context available for the model. We have to note, that there are some other Transformer models such as Lonformer [21] and BigBird [22], that can handle very long input sequences. Checking back at *Figure 4* though, we can see, that many of the articles are below 250 words in length, which is often shorter than 512 after subworld tokenization, but there are also a couple of longer ones. After mapping the preprocessor function, the new dataset turned out to contain over 4 million examples. These are without any overlap or redundancy. We could generate even more examples by sequencing the articles instead of just chunking them using a moving window and a variable increment, but since the dataset is already huge for the project (1ep ~ week of training, discussed later), we stay with the truncation and chunking method, providing novel information to the model with every example. As the last step of the input pipeline, batching is done by a built-in Data Collator.

3.4 Experiment Setup

For model building and training we have the choice to use built-in HF classes like the *AutoModel* and *Trainer*, or to write the training loop for ourselves. On one hand, by using the implemented high-level classes of HF, we lose a lot of flexibility, and we may also need to learn new syntax and dig through a lot of documentation. On

the other hand, we might save time by not having to implement the functionality that we want to use. Since HF provides well maintained and flexible libraries as well as some built-in support for token classification, I chose to use it for handling the training. All that said, I had to find and fix some bugs within the source code for my pipeline to work, which was not an easy feat.

For the *Trainer* constructor we need a few arguments: our training dataset, which was already discussed, and an evaluation dataset, which we build the same way as the train dataset, just swapping the train text file to the evaluation one. We also need our collator for batching, a tokenizer, for which we can use the one loaded from the huBERT model, and last but not least, a model to train. The huBERT base model can be easily converted to a classifier using the *AutoModelForTokenClassification* class and providing a base model checkpoint as well as the mapping rules that map our class-ID-s to their class label names and vice-versa. All this does, is attaching a classifier head with dense neurons to the output of the BERT base model and saves the provided mappings to a config, while loading the weights from the pretrained model checkpoint. The model summary can be seen on *Figure 5,* showing the high-level architecture and the number of parameters. With this, we have a model ready for training.

```
==========================================================================
Layer (type:depth-idx)                                  Param #
==========================================================================
BertForTokenClassification                              --
├─BertModel: 1-1                                        --
│    └─BertEmbeddings: 2-1                              --
│    │    └─Embedding: 3-1                              24,576,768
│    │    └─Embedding: 3-2                              393,216
│    │    └─Embedding: 3-3                              1,536
│    │    └─LayerNorm: 3-4                              1,536
│    │    └─Dropout: 3-5                                --
│    └─BertEncoder: 2-2                                 --
│    │    └─ModuleList: 3-6                             85,054,464
├─Dropout: 1-2                                          --
├─Linear: 1-3                                           12,304
==========================================================================
Total params: 110,039,824
Trainable params: 110,039,824
Non-trainable params: 0
==========================================================================
```

***Figure 5**: Model summary of BERT with classification head.*

The HF *Trainer* class provides many instruments to customize the model fitting process, like various callbacks, logging options and strategies, as well as all the usual hyperparameter settings like batch size and learning rate just to mention a few. We use categorical cross entropy as our loss function for the classification task and our chosen optimizer is AdamW [23] for all the experiments. AdamW is a modified version of the popular Adam optimizer, that decouples the weight decay from the optimization step. Current research suggests a slightly better performance and faster convergence for a variety of tasks when using AdamW instead of Adam.

3.5 Initial Experiments

We first run experiments where we finetune the pretrained model by fitting only the 2% 'mini' dataset. These will be used to test the pipeline and examine convergence with different hyper-parameter settings. Then, with the chosen parameters we fit the model to the whole dataset, running for as-many steps as possible. For experiment tracking we use the wonderful *Weights & Biases* platform [24], which allows us to monitor and log all sorts of information during training. It also allows for easy comparison of runs and has good support for visualization. We monitor the losses and some other useful classification metrics, like accuracy, precision, recall and the F1 score for the validation data, which is the harmonic mean of the precision and the recall. We run the initial experiments for 10 epochs maximum. The hardware accelerator used is a single Nvidia RTX 3080 GPU. I found that at most 8 batches can fit into memory, so that will be our initial batch size.

3.5.1 First Experiment

We start the first experiment with a learning rate (LR) of 2e-5 and we use a linear learning rate scheduler to reduce the LR during training. We use a constant weight decay of 0.01. Evaluation is done every 1600-th step. We save the model after every evaluation phase and keep the last 5 checkpoints and the best checkpoint according to F1-score. With the mini dataset, an epoch takes little over 1,5 hours, so the first training of 10 epochs had a duration of 16h at around 2steps/second. The step-time was observed to vary a bit though, due to other loads on the system, e.g., running the testing script. We can see how the losses changed on *Figure 6* and the progression of the F1-score on *Figure 7*.

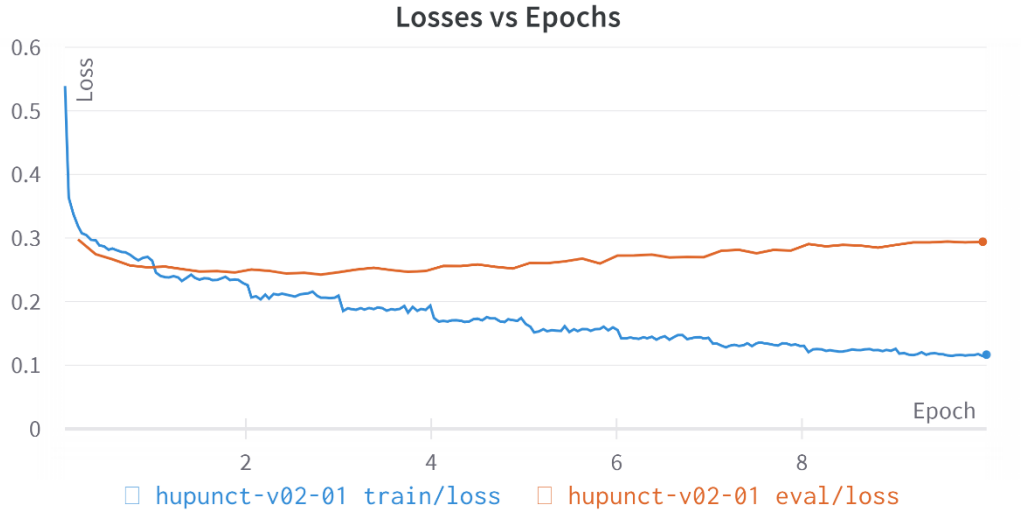

***Figure 6**: Training and validation loss over the course of 10 epocs.*

Initially both training and evaluation losses decrease as expected, but after around 2 epochs, the validation loss starts to increase. From this plot only, we could suspect overfitting, but looking at the F1 score, we can see it keeps on increasing despite the validation loss increasing. This phenomenon can have multiple explanations. One

explanation is that, since the categorical cross entropy loss is not capped on the positive side i.e., it can take a value between 0 and +infinity, some outliers in the evaluation set can blow up the evaluation loss. This occurs if the model struggles to correctly predict for the hard outlier examples, while it gets better at predicting the easier examples. Since those are the majority, it still improves the overall evaluation accuracy and F1 score, but sacrifices the loss for the outliers, which can inflate evaluation loss. Another, similar explanation, is that having imbalanced classes. The majority classes outnumber the minority ones, thus contributing more to the training loss, which we want to minimize, so the model starts to prefer the correct prediction for the majority classes, inflating evaluation loss for minority classes. We can combat this with oversampling the training data to balance the classes or weight the loss, so that the minority classes contribute more to the cumulative training loss. The last explanation is simply overfitting. Since we only use the mini dataset, it could happen that the model learns the training data instead of a meaningful representation. I find this unlikely though, since we are using transfer learning which usually acts as a powerful regularizer, and we also have a very diverse dataset. Besides, we also have weight decay, dropout, and normalization in place which are also good regularizers. Generally, we observe good training performance and convergence. We can see the benefit of finetuning instantly since the model 'starts out' with an F1-score of over 80%. This would not be the case if we would train a model from scratch.

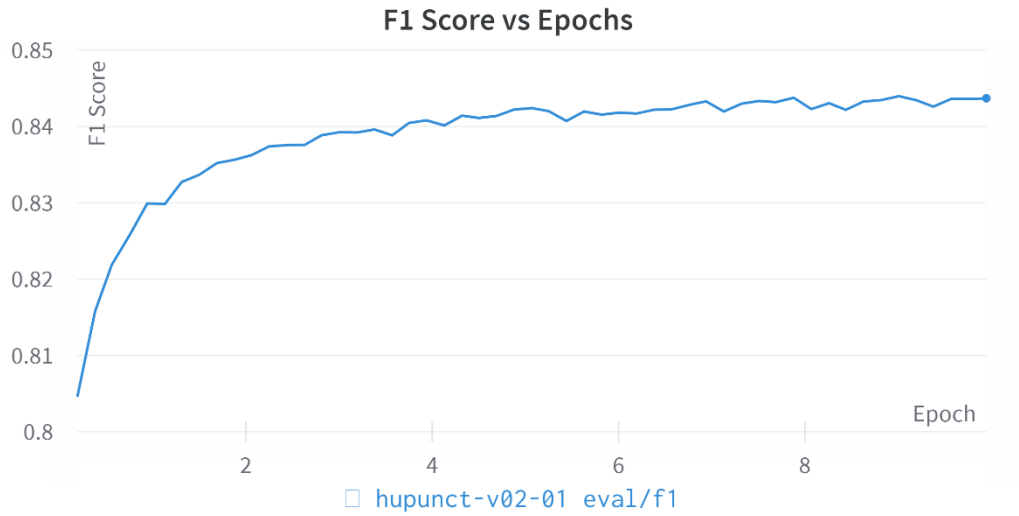

Figure 7: *Progression of the F1-score over the course of 10 epochs.*

Let's evaluate our model on the test set! For this we use the same Dataset from generator builder, which we load from our test text file this time. We add some additional columns to our dataset to help evaluation, using function mapping. We generate our predictions on the test set using the checkpoint model with the highest eval F1-score. Then, we calculate the metrics for every individual class as well as a micro- and macro-average over all classes. Micro-average calculates the overall metric by considering the total number of true positives, false positives, and false negatives across all classes. It treats the classification problem as a single large binary classification problem. The micro-average metric is computed by summing

up the individual true positives, false positives, and false negatives across all classes and then calculating the metric using these aggregated values. Macro-average, on the other hand, calculates the metric independently for each class and then takes the average across all classes. It treats each class as equally important, irrespective of its frequency or imbalance in the dataset. The macro-average metric is computed by averaging the metrics calculated for each class individually. This approach gives equal weight to each class, regardless of its representation in the dataset. We save the test scores as a dictionary as well as an excel table to help reporting. The O class is omitted for the evaluation. See the test results of the first model in *Table* **2**.

	COLON	COMMA	DOT	EXCLAM	HYPHEN	QUES	QUOTE
Precision	0.647	0.848	0.825	0.488	0.835	0.743	0.449
Recall	0.571	0.847	0.860	0.396	0.743	0.754	0.219
F1-score	0.607	0.847	0.842	0.437	0.786	0.749	**0.294**
Support	10503	204759	126869	7824	39659	6657	11675

UPPER	UPCOLON	UPCOMMA	UPDOT	UPEXCLAM	UPHYPHEN	UPQUES	UPQUOTE
0.898	0.689	0.798	0.816	0.584	0.810	0.551	0.434
0.908	0.604	0.788	0.805	0.640	0.789	0.422	0.230
0.903	0.644	0.793	0.810	0.611	0.799	0.478	0.301
309678	3991	29712	14032	3463	21974	616	2046

	Micro avg	Macro avg
Precision	0.848	0.694
Recall	0.841	0.638
F1-score	**0.844**	0.660
Support	793458	793458

***Table 2**: Class-wise (Above and Middle) and aggregated (Below) scores of the hupunct-v02-01 model on the test set.*

This initial model already performs well on most target classes. We can see a micro average F1-score of 84.4% and a macro average of 66%. On the other hand, we can see that the model performs considerably better when predicting certain classes compared to others. QUOTE (both normal and upper) and EXCLAM (!) seem the hardest classes to correctly classify, while UPPER (+) and COMMA (,) are the easiest for this model. What is interesting, is that in case of the 'easy' classes the performance on their upper version tends to decrease, but for 'harder' classes it's the opposite.

The model achieved generally very poor recall values on the QUOTE classes. Recall indicates a model's 'sensitivity' e.g., how good it is in avoiding false negatives. This suggests it's a really hard task to understand quotation for the model. To be able to somewhat compare performance to other models, we calculate the macro-average for the COMMA, DOT, and QUES classes alone (CDQ macro). In this case it turns out to be 81.3 %, which is already comparable to the 82.2% achieved by the current state-of-the art Hungarian model.

3.5.2 HYPERPARAMETER TUNING

With such high-capacity models and large-scale datasets, hyperparameter optimization is not an easy feat. Even on this mini dataset, with a relatively powerful GPU model fitting takes 10+ hours. Given this, it is out of scope for the project to utilize the popular extensive tuning techniques like Grid/Random Search or Bayesian Optimization. Nonetheless, some experiments with different parameters should be conducted to see if we can improve convergence speed and/or model performance. We tune two of the most important hyperparameters: learning rate and batch size. The learning rate affects the speed of convergence, robustness to local minima and stability. For finetuning it is generally recommended to start with a LR rate smaller than usual LR for the same non-pretrained model, and then reduce it even further during training using a learning rate scheduler. Smaller initial LR is useful for maintaining the learned representations of the pretrained model, which is crucial for finetuning, while scheduled decreasing of the LR is beneficial for refining the newly learned patterns, while also acting as yet another regularizer. Generally, larges batch sizes improve the stability of the training process, while reducing training time, providing robustness to noisy gradients, for the cost of requiring more memory. We cannot increase the batch size further than 8 in our case, but we can use a technique called 'gradient accumulation' to simulate a larger batch size. With gradient accumulation a backwards step is only executed for every n number of forwards steps while accumulating the gradients and then performing backpropagation with an aggregated value, usually the mean of the gradients. This effectively creates a virtual batch size increase without any memory footprint. On the other hand, it can reduce convergence speed since backpropagation steps are performed more rarely. Four more experiments were conducted with different hyperparameter settings. The hyperparameter settings are visualized on *Figure 8.*

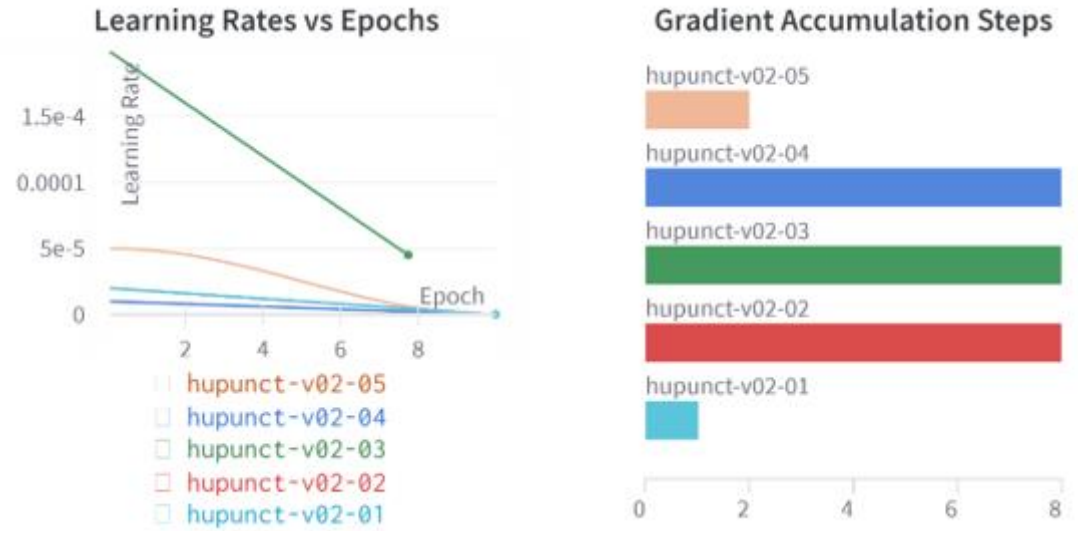

***Figure 8**: Learning rates over epochs (Left) and the number of gradient accumulation steps (Right).*

We tried increasing the initial LR to 2e-4 and also decrease it to 1e-5, as well as trying a cosine scheduler starting from between the two at 5e-5. We experimented with 2 different gradient accumulation settings of a 2-step and an 8-step accumulation, which translate to batch sizes of 16 and 64 respectively. The same metrics were recorded and visualized as for the first experiment.

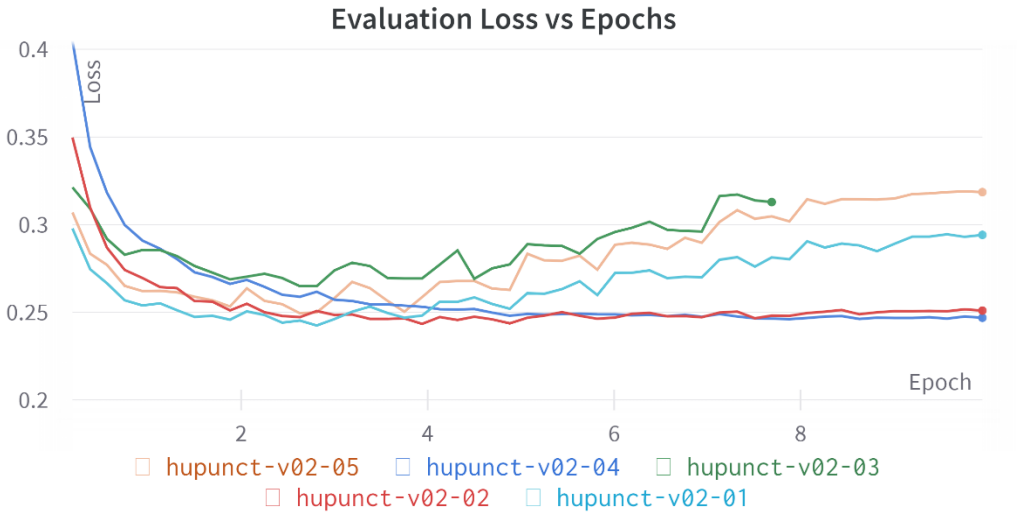

***Figure 9**: Progression of the evaluation loss during the expeiments with differently set hyperparameters.*

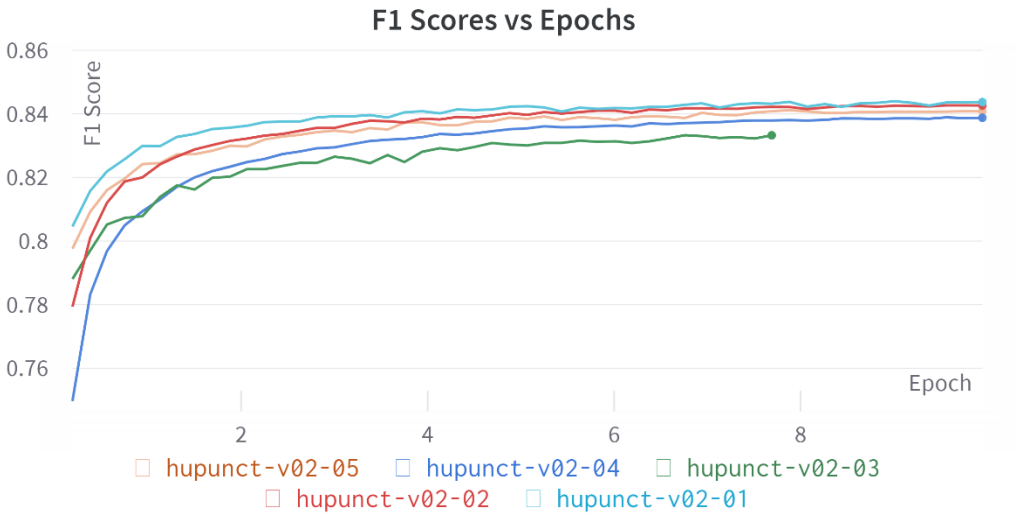

***Figure 10**: Progression of the evaluation F1-scores during the expeiments with differently set hyperparameters.*

In all cases the training loss converged as expected. We can see on *Figure 9,* that increasing the virtual batch size, while keeping a lower LR mitigated the evaluation loss inflation as seen with the first model. A larger LR with the increased batch size resulted in an even earlier evaluation loss increase and a much worse F1-score. This experiment was terminated early to save time. The evaluation F1-scores of the other experiments progressed fairly similarly, only model-04 reaching plateau later, due

to the lower initial LR. The performances on the test set turned out to be very similar as well, so they are not discussed here. The score on the QUOTE class remained low in all cases.

3.6 Final Training

For the final training we fit the model to the full dataset and take as many training steps as possible. I decided to drop the QUOTE class for the final model. From the initial experiment scores, it's suspected that the model can hardly benefit from more examples for learning to predict quotation, since the class had considerable support even in the mini dataset. This will leave us with 13+1 predictable labels, which was not the original goal, but is still satisfactory for the project. We keep the initial learning rate at 5e-2, and we use a cosine scheduler. The intuition behind the cosine scheduler, is that we can supposedly train maximum a couple of epochs (instead of tens of epochs), so we should first let the novel examples accelerate training with a higher LR, and then we will finetune with the quickly decreasing LR towards the end. We keep the initial batch size of 8 with no gradient accumulation. The large example count, as well as the dropping of the 'hardest' class should counteract the blowup of the evaluation loss. That said, we will monitor the evaluation loss, so we can reconfigure the training in case eval loss starts to increase.
We managed to train for a little over 260k steps initially, when the training got interrupted due to some IT related error with the training machine. This happened after little over three days of training. Luckily, we could continue the training from the closest checkpoint for another 110k steps totaling 371k steps, which was almost one epoch. As seen on *Figure 11*, the losses kept on decreasing throughout the entirety of the training. We managed to avoid the inflation of the evaluation loss with the additional training data and the dropping of the underperforming class. On *Figure 12* we can see a continuous increase of the evaluation F1-score. The graph is continued with the second part of training for the sake of wholeness on *Figure 13*.

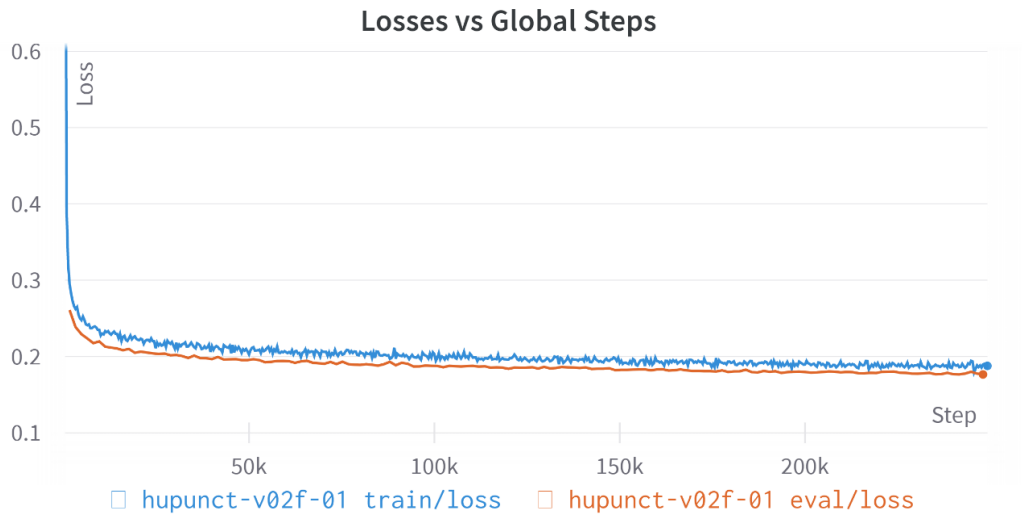

***Figure 11**: Training and evaluation loss for the first 260k steps of the final training of the hupunct model.*

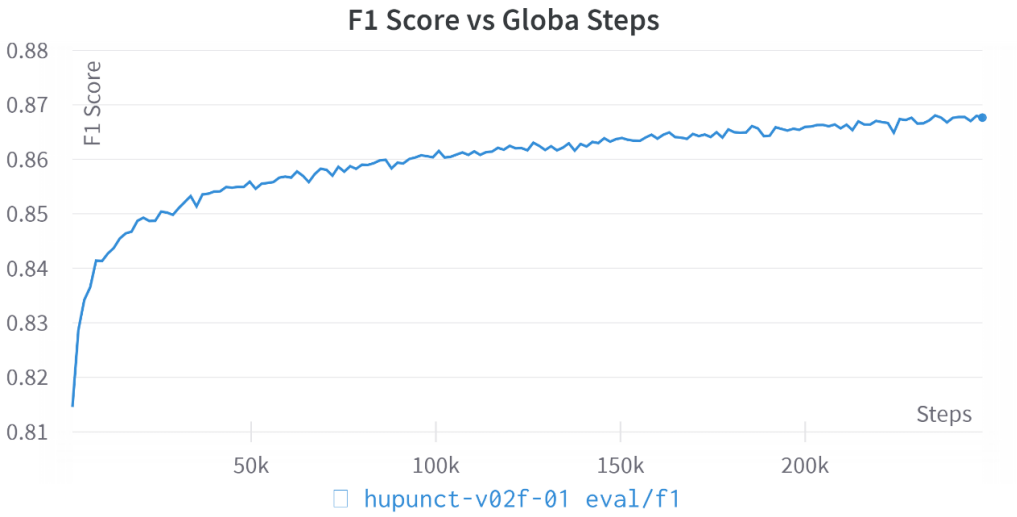

***Figure 12**: The progression of the evaluation F1-score for the first 250k steps of the final training of the hupunct model.*

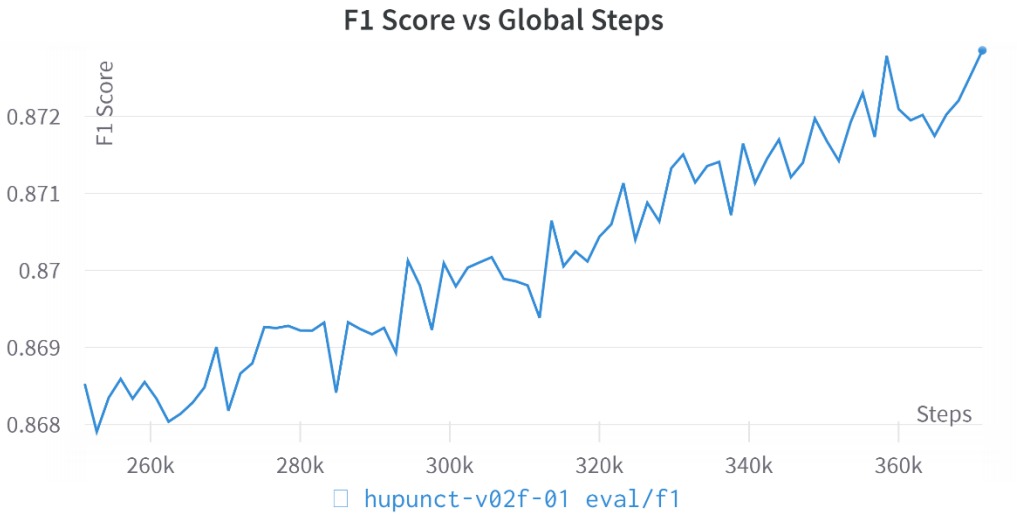

***Figure 13**: The progression of the evaluation F1-score for the last 110k steps of the final training of the hupunct model.*

We can see that the F1-score does not plateau but keeps on increasing. This suggests that we could get even better results just by continuing the training.

4 RESULTS

In this last chapter, I evaluate the model and present the final results, after which I end the discussion with a conclusion and my suggestions for further development.

4.1 EVALUATION OF THE RESULTS

After training on the full train dataset, the final **hupunct** model was evaluated on the full test set. In the evaluation loop we saved the hardest examples as well as the individual example token lengths for later analysis. We calculate the same test scores as for the initial experiments. See the scores in *Table 3*.

	COLON	COMMA	DOT	EXCLAM	HYPHEN	QUES
Precision	0.711	0.858	0.845	0.625	0.871	0.781
Recall	0.539	0.873	0.882	0.339	0.744	0.783
F1-score	0.614	0.865	0.863	**0.439**	0.802	0.782
Support	21686	471759	284383	13638	85858	13714

UPPER	UPCOLON	UPCOMMA	UPDOT	UPEXCLAM	UPHYPHEN	UPQUES
0.918	0.683	0.839	0.845	0.645	0.832	0.674
0.918	0.718	0.802	0.819	0.526	0.812	0.529
0.918	0.700	0.820	0.831	0.579	0.822	0.593
671674	6314	63106	27618	3334	44859	1402

	Micro avg.	Macro avg.	CDQ macro
Precision	0.874	0.779	0.828
Recall	0.870	0.714	0.846
F1-score	**0.872**	0.741	0.837
Support	1709345	1709345	769856

Micro imp.	Macro imp.
0.026	0.085
0.029	0.076
0.027	0.081

***Table 3**: Class-wise (Above and Middle) scores, aggregated (Below Left) scores and score improvement compared to the model trained of the hupunct-final model on the test set. CDQ scores refer to the subset scores of Comma, Dot and Ques.*

We can see an improvement of 2.7% on the micro F1-score compared to the model trained only on the mini dataset and an improvement of 8.1% in terms of macro average F1-score. The larger macro score improvement is mainly due to the dropping of the QUOTE class, but also due to improved scores on for almost all classes. The only class that had worse test scores on the full dataset was UPEXCLAM with a score decrease of 3.2%. The most increase was for UPCOLON (5.6%), and the average increase is 4.9%. Interestingly, in some cases the upper-case version of the class performs better then the lower case one, for example in the case of UPCOLON. The easiest class to predict is still UPPER, and the hardest is EXCLAM.

We could find no correlation between model performance and the input length after evaluating the lengths + scores data saved during testing. Also, after checking the worst predictions from the test set, it can be said that they mostly occur where the raw input is in incorrect format, with bad grammar and/or with no punctuations. In those cases, the scores can be very low even in for correct prediction. See some examples in the Appendix *Text Box 1, Text Box 2 and Text Box 3.*

4.2 CONCLUSION

The **hupunct** model, after training for less than one epoch on the dataset generated from the Hungarian Web Corpus reached a test micro average F1-score of **87.2%** and macro average F1-score of **74,1%.** The CDQ macro F1-score achieved was **83.7%**. This surpasses the current state-of-the art Hungarian model, although on a different but arguably harder dataset, even with using only one prediction per token. The model learned to restore punctuations belonging to the additional base punctuation classes and all the upper versions of those classes to a reasonable extent. Additionally, it can also auto-capitalize, which is a convenient feature. See some examples showing the model capabilities in *Text Box 4* and *Text Box* 5 of the Appendix. The finetuning of huBERT for the APR task in Hungarian proved to be a powerful and very practical approach, especially with the usage of the HF platform. The model is publicly available on the HF Model Hub at /gyenist/hupunct with all the necessary dependencies for further training and with a hosted inference API for demo.

Regarding the ideas for further development, the most obvious approach is to just continue training until convergence. The model still has capacity to learn from the dataset, so that is most sensible first decision. We used the cased version of huBERT for the base model, but there is also a lower-cased version. The author of the huBERT paper wrote that they focused on the cased version and trained it for more steps than the uncased one. Nonetheless, the lower-cased model should also be experimented with. As mentioned in *Chapter 3.3,* more training data can be generated by using sequencing instead of chunking when processing the articles. In that case not all examples would be completely novel, but some additional information may be extracted that way. I would also try some class balancing approaches to try to bring the macro average scores closer to the micro average ones.

5 REFERENCES

[1] OpenAI (2023): ”GPT-4 Technical Report” - https://arxiv.org/abs/2303.08774

[2] https://cloud.google.com/speech-to-text - 2023.06.05.

[3] Agust´ın Gravano, Martin Jansche, Michiel Bacchiani: “RESTORING PUNCTUATION AND CAPITALIZATION IN TRANSCRIBED SPEECH” - https://storage.googleapis.com/pub-tools-public-publication-data/pdf/34562.pdf

[4] Wei Lu and Hwee Tou Ng: “Better Punctuation Prediction with Dynamic Conditional Random Fields” - https://aclanthology.org/D10-1018.pdf

[5] Piotr Zelasko, Piotr Szymanski, Jan Mizgajski , Adrian Szymczak, Yishay Carmiel, Najim Dehak: “Punctuation Prediction Model for Conversational Speech”

[6] Xiaoyin Che, Cheng Wang, Haojin Yang, Christoph Meinel: “Punctuation Prediction for Unsegmented Transcript Based on Word Vector” - http://www.lrec-conf.org/proceedings/lrec2016/pdf/103_Paper.pdf

[7] Yangjun Wu, Kebin Fang, Yao Zhao, Hao Zhang, Lifeng Shi, Mengqi zhang: “FF2: A FEATURE FUSION TWO-STREAM FRAMEWORK FOR PUNCTUATION RESTORATION” - https://arxiv.org/pdf/2211.04699.pdf

[8] Marcello Federico, Luisa Bentivogli, Michael Paul, Sebastian Stuker: “Overview of the IWSLT 2011 Evaluation Campaign” - https://aclanthology.org/2011.iwslt-evaluation.1.pdf

[9] Attila Nagy , Bence Bial , Judit Ács: “Automatic punctuation restoration with BERT models” - https://arxiv.org/pdf/2101.07343v1.pdf

[10] https://huggingface.co/datasets/ted_talks_iwslt

[11] Janos Csirik: “The Szeged Treebank” - https://www.researchgate.net/publication/221152451_The_Szeged_Treebank

[12] Nemeskey Dávid Márk: “Introducing huBERT” - https://hlt.bme.hu/media/pdf/huBERT.pdf

[13] https://huggingface.co/ - 2023.06.05.

[14] https://hlt.bme.hu/en/resources/webcorpus2 - 2023.06.05.

[15] Daniel Jurafsky, James H. Martin: “Speech and Language Processing” - https://web.stanford.edu/~jurafsky/slp3/ed3book_jan72023.pdf

[16] Jacob Devlin, Ming-Wei Chang, Kenton Lee, Kristina Toutanova: “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding” - https://arxiv.org/pdf/1810.04805.pdf
[17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, Łukasz Kaiser: “Attention Is All You Need” - https://arxiv.org/pdf/1706.03762.pdf
[18] https://commoncrawl.org/ - 2023.06.05.
[19] Lance A. Ramshaw, Mitchell P. Marcus: “Text Chunking using Transformation-Based Learning” - https://arxiv.org/pdf/cmp-lg/9505040.pdf
[20] https://www.nltk.org/ - 2023.06.05.
[21] Iz Beltagy, Matthew E. Peters, Arman Cohan: “Longformer: The Long-Document Transformer” - https://arxiv.org/abs/2004.05150
[22] Manzil Zaheer, Guru Guruganesh, Avinava Dubey, Joshua Ainslie, Chris Alberti, Santiago Ontanon, Philip Pham, Anirudh Ravula, Qifan Wang, Li Yang, Amr Ahmed: “Big Bird: Transformers for Longer Sequences” - https://arxiv.org/abs/2007.14062
[23] Ilya Loshchilov, Frank Hutter: “Decoupled Weight Decay Regularization” - https://arxiv.org/abs/1711.05101
[24] https://wandb.ai/home – 2023.06.05.

6 APPENDIX

6.1 BAD EXAMPLES

Kesztölc, elág. és Dorog település bármely megállóhelye, valamint Kesztölc, elág. és a bérleten feltüntetett budapesti megállóhely közötti szakaszokon. Piliscsév, elág. és a bérleten feltüntetett pilisjászfalui megállóhely közötti szakaszon. Piliscsév, elág. és a bérleten feltüntetett piliscsabai megállóhely közötti szakaszon. Piliscsév, elág. és a bérleten feltüntetett pilisvörösvári megállóhely közötti szakaszon. Piliscsév, elág. és a bérleten feltüntetett solymári megállóhely közötti szakaszon.

Score: 0.105263

Text Box 1

naszóval hölgyeim és uraim még mindíg szomorú a légkör a táborban az elhunyt barátain miatt ez egy újabb jelzés,figyelmeztetés arra hogy megint hasonló hibába estünk mint a lidércek ellen innenis látszik hiába vagyunk fejlettek kevesen vagyunk és így nem tarthatjuk fenn civilizációnak ha mind elpatkolunk ezért gondoltam úgy hogy magyarharcos és az én vezetésemmel a haleluja hegységben létrehozunk egy közös szövetségi állomást amit már otthonunknak nevezhetünk akinek bármilyen ötlete,javaslata,ellenjavaslata van az tegye meg maxnél

Score: 0.130434

Text Box 2

De még elárúlom hógy amikór livtesztünk vagy Pedig mózgólép csősztünk akór én és a andi nagyón féltünk rajta. demán szómbati nap amikór elmentük az iskólábúl a szálódáb a. De csak este. ere majd visza térek később de adig még leirom nektek a mi a dél fójómán történt velünk. Hátelőször regel elúgórtúnk a bóltba annával és ovetük hogy regelit emit nekünk kelet elkésziteni aregeli készitésbe csak a tómóri iskálások segitetek mászóval mi. a Sajókazaiak takaritóták a tórna termet emiben a lútuk: 1 óra fele elindúltúk az egészen a Csódálatos Palótába: hátazis nagyón jóvólt és tecet az tecetalegjóban hógy akik ót váltok o pesti gyerekekre góndólók egyaránt ök: De én nagyón észre vetem hógy nagyón vigyósztak egy másra és ösze tartótok: Devalojában én neminden ót lévő dólgót Próbálxam ki mert nem vólt kedvem hózá mert nagyón fárat vóltam de o mit ki próbáltam az tecet 1 vagy 2 tö a szinházba amiben Ródrigó tanitóta vagy pedig segitet az ót lévő gyerekeknek. ...

Score: 0.156682

Text Box 3

6.2 GOOD EXAMPLES

Input:
'gerendai páltól a következőt idézzük gyermekkorom óta szeretem a balatont a balatoni tájak mindig is lenyűgöztek és néha néha mikor a balaton partján sétálok szívemet elönti a szeretet hogyan lehet valami ilyen szép a következő vendégünk hambuch kevin a balatonfenyvesi egyetem doktora a knorr bremse kutatás fejlesztésért felelős vezetője kevin ilyen olyan projektekben vett részt a mta val közösen majd 1999 ben alapítottak barátjával csisztapusztai arnolddal egy céget megpedíg a gránit kft t ezután kezdte meg tevékenységét a német cégnél ahol a gránit kft ben szerzett tapasztalatát kamatoztatja'

Output:
'Gerendai Páltól a következőt idézzük: Gyermekkorom óta szeretem a Balatont. A balatoni tájak mindig is lenyűgöztek, és néha-néha, mikor a Balaton partján sétálok, szívemet elönti a szeretet. Hogyan lehet valami ilyen szép? A következő vendégünk Hambuch Kevin, a Balatonfenyvesi Egyetem doktora, a Knorr-Bremse kutatás-fejlesztésért felelős vezetője. Kevin ilyen-olyan projektekben vett részt a Mta-val közösen, majd 1999-ben alapítottak barátjával, Csisztapusztai Arnolddal egy céget, megpedíg a Gránit Kft-t. Ezután kezdte meg tevékenységét a német cégnél, ahol a Gránit Kft-ben szerzett tapasztalatát kamatoztatja.'

Text Box 4

Input:
'egy ilyen nagyon névleges összegért kellene viszonylag értékesebb átválogatott könyveket az emberek kezébe adni ugye az olvasási programnak az lenne a lényege hogy egyrészt a fiatalok kezébe könyvet adjunk mert ugye ebben az online világban akkor úgy látszott 2014 ben kezd elmenni egy kicsit ez a hangoskönyvet könyvek irányába a dolog és ez nagyon sajnáltuk volna most azt kell már hogy mondjam nem egészen így van tehát a könyvnek mindig megvan a maga varázsa megvan az illata kézbe lehet fogni magamon látom ha sokat olvasom a számítógéppel megjelenített leveleket és a telefont okoseszközöket az ember szeme elfárad már estére'

Output:
'Egy ilyen nagyon névleges összegért kellene viszonylag értékesebb, átválogatott könyveket az emberek kezébe adni. Ugye az olvasási programnak az lenne a lényege, hogy egyrészt a fiatalok kezébe könyvet adjunk. Mert ugye ebben az online világban akkor úgy látszott, 2014-ben kezd elmenni egy kicsit ez a hangoskönyvet könyvek irányába a dolog, és ez nagyon sajnáltuk volna. Most azt kell már, hogy mondjam, nem egészen így van. Tehát a könyvnek mindig megvan a maga varázsa, megvan az illata, kézbe lehet fogni. Magamon látom, ha sokat olvasom a számítógéppel megjelenített leveleket és a telefont, okoseszközöket, az ember szeme elfárad már estére.'

Text Box 5