FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

David Klement

Writing assistant based on large language models

Institute of Formal and Applied Linguistics

Supervisor of the bachelor thesis: Mgr. Jindřich Helcl, Ph.D.
Study programme: Computer Science

Prague 2024

I want to thank my supervisor, Jindřich Helcl, for his positive attitude and the countless ideas he had during the work on this thesis. I would also like to thank Milan Straka for sparking my interest in deep learning. Finally, I want to thank my family and friends for keeping my spirits up during the long months of work.

Computational resources were provided by the e-INFRA CZ project (ID:90254), supported by the Ministry of Education, Youth and Sports of the Czech Republic.

Title: Writing assistant based on large language models

Author: David Klement

Institute: Institute of Formal and Applied Linguistics

Supervisor: Mgr. Jindřich Helcl, Ph.D., Institute of Formal and Applied Linguistics

Abstract: A standard approach to many natural language processing tasks is to take an existing, pre-trained large language model and fine-tune it for the given task. Such an approach leads to having a separate model for each task; furthermore, the fine-tuning must be repeated when upgrading to a new pre-trained model. This thesis explores the possibilities of using a single off-the-shelf model for three different tasks without fine-tuning. We present *Preditor*, a writing assistant that supports rewriting a sentence after replacing one of its words, suggesting continuations, and suggesting words that fit into a sentence. We design the system in a model-agnostic way, making it possible to upgrade to a new model with little effort. We also provide an extension that integrates the assistant into the text editor.

Keywords: writing assistant, model-agnostic, large language models, natural language processing

Název práce: Asistent pro psaní textu založený na velkých jazykových modelech

Autor: David Klement

Ústav: Ústav formální a aplikované lingvistiky

Vedoucí bakalářské práce: Mgr. Jindřich Helcl, Ph.D., Ústav formální a aplikované lingvistiky

Abstrakt: Standardní přístup k mnoha úlohám zpracování přirozeného jazyka je vzít existující předtrénovaný velký jazykový model a dotrénovat jej pro danou úlohu. Tento přístup vede k tomu, že pro každou úlohu vznikne samostatný model, navíc je nutné dotrénování opakovat při přechodu na lepší předtrénovaný model. Tato práce zkoumá možnosti použití jediného veřejně dostupného modelu pro tři různé úlohy bez nutnosti dotrénování. Představujeme *Preditor*, asistenta pro psaní, který umí přepsat větu po nahrazení jednoho z jejích slov, navrhuje pokračování věty a navrhuje slova, která se hodí doprostřed věty. Systém navrhujeme nezávisle na konkrétním modelu, což umožňuje jednoduchý přechod na nový model. Poskytujeme také rozšíření, které integruje asistenta do textového editoru.

Klíčová slova: asistent pro psaní, nezávislost na modelu, velké jazykové modely, zpracování přirozeného jazyka

Contents

Introduction

This thesis aims to create a *writing assistant*, a tool that helps write text in a natural language. The goal is to speed up typing by offering text suggestions and to improve the editing experience by automating tedious word replacements.

Recent advances in natural language processing have led to the development of large language models, which can generate text that approaches human quality. Such models make great tools that suggest continuations of a sentence. It is also possible to train these models for other tasks; we could create a writing assistant with many features if we trained enough such models.

However, an assistant that uses several large language models is inefficient. First, many models need to be loaded into memory, which leads to a high memory footprint, as the memory requirements of language models are high. Second, state-of-the-art language models need a lot of computational resources to train or fine-tune for a specific task. At the same time, new models emerge frequently; if we wanted to keep up with the latest models, we would need to retrain our model often.

To address these issues, we introduce *Preditor*, a writing assistant that accomplishes three different tasks using a single language model. Furthermore, we use an off-the-shelf model without the need for further training. As such, updating to a newer model should be possible without too much effort.

Our goal is not to find a state-of-the-art solution to the tasks; fine-tuned models or models trained specifically for the tasks would perform better. Instead, we aim to explore the possibilities of adapting existing models to several tasks.

Tasks

Preditor helps the user with three tasks: substitution, prediction, and infilling.

Substitution The main focus of this thesis is sentence editing. Quite often, a sentence needs to be modified, whether it is to shift the meaning or to improve the style. In any given sentence, some parts carry the semantic meaning, while others are there just to make it grammatical. Depending on the language, these parts can be entire words or just parts of words. Whenever we modify a sentence, even slightly, we may need to change many words to keep the sentence grammatically correct.

An example is worth a thousand words, so let us look at one. Take the sentence, ”My friend exercises regularly because he finds it important for his health.” If we were to change the subject from ”friend” to ”friends”, we would need to change four additional words: ”My *friends exercise* regularly because *they find* it important for *their* health.”

This effect is even more pronounced in languages with rich morphology; we focus on Czech in this thesis. For example, the form of an adjective depends on the gender of the respective noun. Combined with the fact that Czech has grammatical gender, the need to adjust many words arises frequently. Our assistant performs these edits automatically; we call this task substitution.

Prediction Another task that our assistant offers is prediction. In this task, the goal is to suggest the next few words in a sentence. Reusing the example from before, given the beginning of the sentence ”My friend exercises regularly because he finds”, the assistant may suggest, ”it important for his health.” But it may also suggest another continuation. There are many ways to complete this sentence, and the assistant wants to choose one with a high probability of being right. The length of the suggestion plays a role, too. Longer suggestions may save more typing, but they are more likely to diverge from what the user wants to say.

Infilling The final task is infilling, where the goal is to suggest a few words that fit into a sentence. This requires using both the left and right context, i.e., the words before and after the blank. Take the following example: ”My friend exercises regularly because he ____ a marathon.” The previous continuation no longer makes sense since it does not match the right context. In this case, a plausible suggestion would be ”is training for”.

Previous Approaches

Here, we briefly overview the existing approaches to the individual tasks. Details for the mentioned works follow in Chapter 2.

No dedicated tools exist to perform substitution, as the task is quite specific. It is possible to achieve some success with chat models, such as ChatGPT, that can reformulate sentences given a suitable prompt. However, they are not very consistent.

Prediction is a common task, and many models exist for it, ranging from simple statistical models to large language models. However, there are only a few techniques to find the best place to end the prediction. One approach is to fine-tune the model to generate an end-of-sentence token at the place where the suggestion should end; some writing assistants use this approach. We were not able to find any techniques that avoid fine-tuning, although it is likely that some exist.

Infilling is also a common task. Masked modes, such as BERT [Devlin et al., 2019], can solve it directly, as they are trained to predict the missing words; however, they need to know the length of the missing part beforehand. Other models can be fine-tuned to perform infilling. However, there are not many models that handle both prediction and infilling well; one approach is to use an encoder-decoder model, as explored by Ippolito et al. [2022].

1. Background

This chapter introduces the fundamental concepts that *Preditor* builds upon.

1.1 Language Models

Language models are a fundamental part of natural language processing. They are statistical models designed to capture the structure of language. By analyzing the patterns and structures in a large corpus of text, language models learn to estimate the probability of a sequence of words appearing in a sentence.

Various language models differ in the way that they estimate this probability. Some are unidirectional and only consider the preceding words in the sequence. In other words, in some models, the probability of a word appearing in a sentence does not depend on the words that come after it. Other models are bidirectional and simultaneously consider the preceding and following words in the sequence. Each approach has its advantages and disadvantages and is suitable for different tasks.

Estimating the probabilities is closely related to predicting what words fit in a sentence. The model can calculate the probability of each possible word based on the words it has seen. Then, it can choose the word with the highest probability.

1.1.1 Tokenization

Language models do not grasp the entire sentence at once. They split the sentence into smaller units that they further process. These units are called tokens, and the tool that performs this splitting is called a tokenizer. Language models learn the meaning of each token, and they combine the meanings of the tokens to understand the sentence as a whole. It seems natural to split the text into words; however, it is impossible for these models to learn every possible word because the number of words in a language is too large. Even if the model learned all the words, it would struggle if it encountered a word it had never seen before.

So, instead of splitting the text into words, we split it into smaller units called subwords; these subwords will be the tokens that the model works with. Subwords can range from individual characters to entire words, and the model can learn the meaning of each subword. When the tokenizer encounters a new word, it can break it down into tokens it already knows; an example of this is shown in Figure 1.1. The tokenizer also assigns a unique numerical ID to each token, so the model only works with these numbers.

We can choose the set of tokens in any way we like as long as we can construct any word from them. There are several ways to do this; one such strategy is based on Byte Pair Encoding (BPE), as introduced by Sennrich et al. [2016]. BPE takes a large corpus of text and creates a token for each unique character that occurs in it. Then, it counts the frequency of each pair of tokens in the text and replaces the most frequent pair with a new token. This step is repeated until the desired vocabulary size is reached.

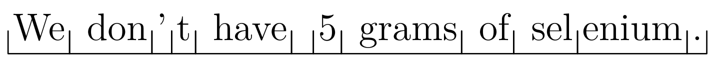

Figure 1.1: Tokenization of a sentence.

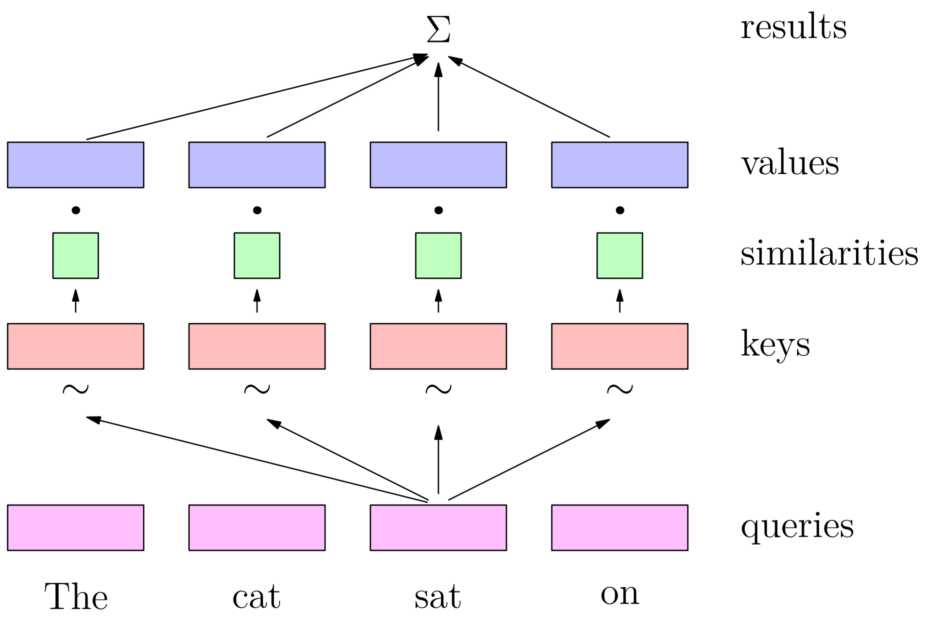

Figure 1.2: One self-attention layer. Only the calculation for the third token is shown. The queries and keys are compared to calculate similarities, and these similarities are used as weights for the weighted sum of values.

However, such an approach could create tokens consisting of multiple words, which is undesirable. To avoid this, we first run pre-tokenization, which creates boundaries in the text that the tokenizer cannot cross. For example, we can create boundaries at the end of each word. This behavior is visible in Figure 1.1, where no token contains a letter followed by a space.

1.1.2 Transformer

The models we explore in Preditor are based on the Transformer architecture introduced by Vaswani et al. [2017]. Transformer models have been highly successful in a wide range of tasks and are today's state-of-the-art models for text generation.

The Transformer model captures the relationships between tokens in a sequence using a technique called self-attention, depicted in Figure 1.2. In essence, each token is represented as a vector of numbers. The model uses its trained parameters to calculate other vectors from these representations; these vectors are called queries, keys, and values. Let's focus on position P in the sequence. The query at the position P is compared to all the keys to calculate the similarity, which is a score that represents how related the two positions are. The new representation of the position P will be a weighted sum of the values. The similarities determine the weights, so more related positions have a larger impact. Afterwards, the representation is passed through a feed-forward neural network. The entire process is repeated multiple times, allowing the model to capture complex relationships between tokens.

The original Transformer model consists of an encoder and a decoder, as shown in Figure 1.3. The encoder processes the input sequence while the decoder generates the output sequence. The decoder differs from the encoder in that it uses *masked-attention*: The calculation for position P only uses tokens to the left of position P. This is necessary, as the model cannot look into the future when generating the output sequence.

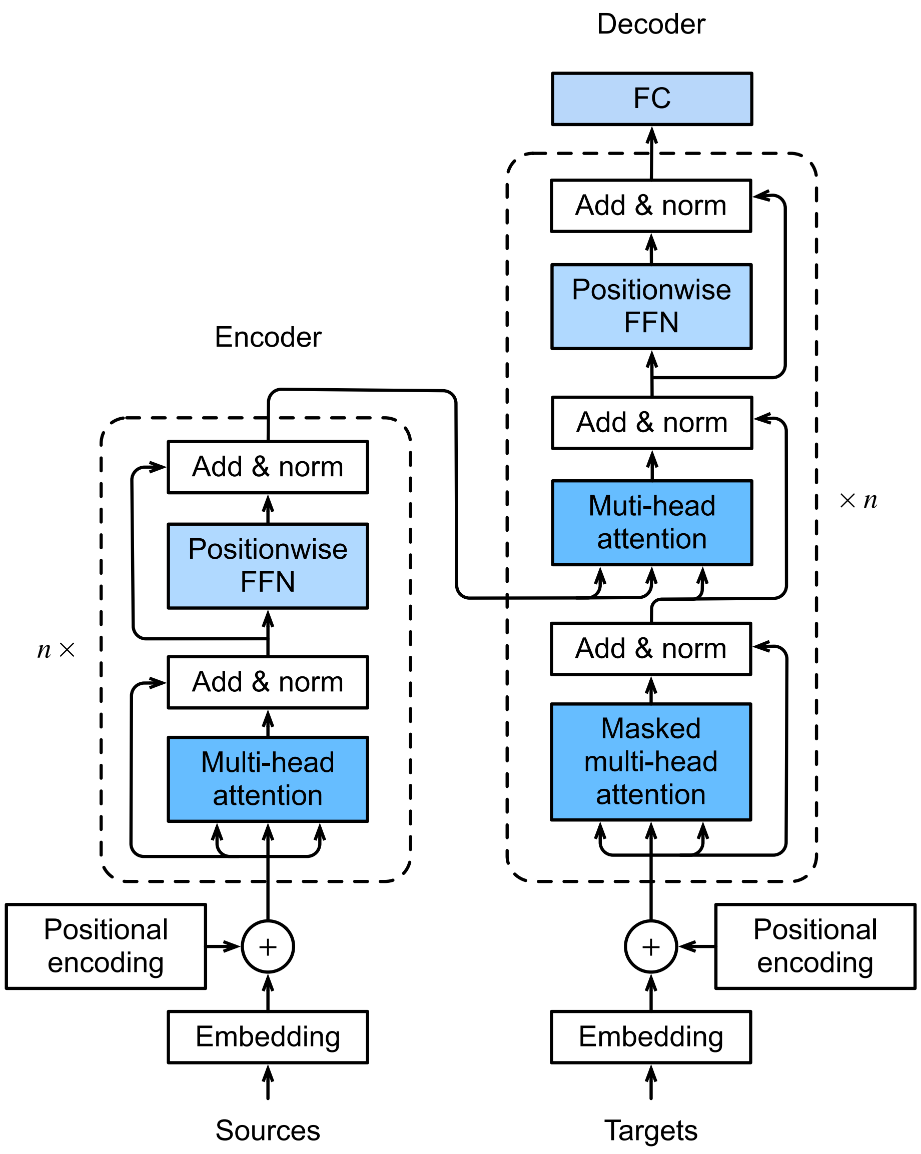

Figure 1.3: The transformer architecture with the encoder and decoder parts. Source: `https://d2l.ai/_images/transformer.svg`

The encoder-decoder architecture is useful for tasks like machine translation, where the model receives a sentence in one language and outputs the translation in another language. However, it is possible to only use the encoder part of the model; such models are suited for tasks like text classification [Devlin et al., 2019]. Similarly, it is possible to train a decoder-only model, as first demonstrated by Radford et al. [2018] in the Generative Pre-trained Transformer (GPT) model; this architecture is more suitable for text-generation tasks.

1.1.3 Generation & Scoring

Preditor uses a decoder-only Transformer, so we focus on this architecture. A decoder-only model takes a sequence of tokens as input, and it outputs the probability distribution for the next token in the sequence. The model is unidirectional, meaning it can only consider the preceding tokens when predicting the next token.

Text generation begins with the model receiving an initial input sequence, often just a start-of-sequence token. The model then generates a probability distribution for the next token in the sequence. We select the token with the highest probability and append it to the input sequence.[1] The resulting sequence is then fed back into the model, which predicts the next token based on the updated sequence. This process is repeated until a stop condition is met, such as reaching the desired sequence length. While it allows us to generate a sequence of any length, it also means that the model can only generate tokens one by one.

It is also possible to take an existing sentence and score it, i.e., calculate the probability of the sentence according to the model. This can be useful for comparing different sentences and selecting the best one; a higher probability indicates that the model considers the sequence more likely. We let the model generate a probability distribution at each position in the sequence. Then, we select the probability corresponding to the actual token at the next position. We multiply these probabilities together to get the probability p_s of the entire sequence.

However, multiplying many probabilities can lead to numerical underflow, especially for long sequences. To avoid this, we often work with log probabilities instead, which are more numerically stable. The log probability of the entire sequence is the sum of the log probabilities of each token in the sequence.

$$p_s = \prod_{i=1}^{n} p_i$$

$$\log p_s = \sum_{i=1}^{n} \log p_i$$

1.1.4 Attention Cache

Calculating the queries, keys, and values for each token in the sequence is computationally expensive. This is especially problematic during text generation, where we repeatedly process the same sequence with only one token appended at each step. Using optimization techniques such as attention cache substantially reduces the computational load and speeds up the text generation process.

The attention cache stores the keys and values after they are computed. This is possible because these states do not change when we append a new token to the

[1] There are also other methods for selecting the token, such as sampling from the distribution.

sequence; they only depend on the states in the previous layer at the current position and to the left. When we add a new token to the sequence, we only need to compute the states for that specific position. The newly calculated query is compared to the cached keys, and the weighted sum uses the cached values. There is no need to cache the queries as they are used only in the current position.

1.2 Tagging

For the substitution task, it is useful to be able to generate other grammatical forms of a word. We achieve this using tagging.

Tagging is a task in natural language processing that involves labeling the words in a text according to their grammatical role. In the most basic form, the tagger determines the part of speech (POS) of each word. It may also identify other linguistic features, such as the person, tense, number, etc.

Tagging provides a deeper understanding of the syntax within a text. For instance, it can help disambiguate words with different meanings but the same spelling. The English word ”well” can be a noun, an adjective, an adverb, or an interjection. By identifying the tag of a word, we can better understand the context and meaning of a sentence. Because of this, tagging used to be the first step in many classical NLP approaches.

There are various techniques for tagging, ranging from rule-based methods to machine-learning approaches. Rule-based methods use hand-written rules and dictionaries, while machine-learning approaches learn from annotated corpora.

1.2.1 MorphoDiTa

MorphoDiTa is a morphological analyzer and tagger developed by Straková et al. [2014]. It is mainly aimed at Czech but can be used for other languages as well.

The tagger receives a sentence as input and outputs a list of *tagged lemmas*. A lemma is the base form of a word without any inflections. The lemma captures the meaning of the word without any grammatical information.

Each lemma has a tag associated with it, which describes the grammatical features of the word form. The features for the Czech tagger are encoded using positional tags as described by Hajič [2004]. The tag uses 15 features; thus, the resulting tag is a string of 15 characters. Words usually do not use all features; in that case, the unused position is filled with a dash.

For example, the word ”chodím” (”I walk”) has the lemma `chodit` and the tag `VB-S---1P-AAI--`. The individual features are verb (`V`), indicative (`B`), singular (`S`), first person (`1`), present (`P`), affirmative (`A`), active (`A`), imperfect (`I`).

MorphoDiTa can also generate word forms. If we provide a lemma and the desired tag, it will output the word form they describe. It is also possible to allow multiple symbols at some positions in the tag by creating a tag wildcard. For example, the tag wildcard `VB-?---[12]P-AAI--` allows the word form to be in any number (singular or plural) and in the first or second person. Generating word forms with this tag wildcard and the lemma ”chodit” yields `chodím`, `chodíme`, `chodíš`, `chodíte`.

2. Related Work

This chapter reviews the existing work that focuses on a subset of our tasks (substitution, prediction, and infilling). We mention one approach for each of the Transformer architectures: Decoder-only models, encoder-only models, and models with both encoder and decoder.

2.1 Chat Models

Chat models, also known as conversational agents, are a type of language model designed to simulate human-like conversation. Their input is structured as a dialogue with delimited sections for the user's messages and the model's responses. The user can ask a question, and the model generates a response, leaving space for the user to continue the conversation. These models typically stem from a pre-trained language model that has been fine-tuned for conversations. One prominent example of a chat model is ChatGPT,[1] developed by OpenAI, which is based on the GPT-3 model.

The underlying pre-trained language models already show the ability to solve a wide variety of tasks; they only need well-crafted instructions to guide them [Brown et al., 2020]. Chat models further extend this capability because they are trained to follow the user's instructions. Another advantage is that the model ends its response in a predictable way, which makes it easier to extract the desired output from its response.

Chat models are usually decoder-only models, and as such, they can perform prediction easily. They can also be effectively used for infilling tasks, where the model fills in missing information in a given context. The user can prompt the model by providing instructions and a sentence with a blank marked by underscores or a similar placeholder. The model should then generate the missing information. However, the form of its answer differs from sentence to sentence. Sometimes, it only includes the missing information; sometimes, it repeats the context, as seen in Table 2.1. It may also happen that the model ignores the instruction and leaves the blank empty. In rare cases, the model may even change the original sentence to make it easier to fill in the blank.

Chat models can perform substitution, too. Table 2.2 shows an example where we want to replace one noun in the sentence with a different noun and adjust the forms of other words to reflect that the grammatical gender has changed. The prompt once again includes an instruction and the sentence to modify. Here, the instruction tells the model what word to replace and what the replacement should be. In this case, the

[1] https://chat.openai.com

User	Fill in the blank marked by ___. I woke up in the ___ the night.
ChatGPT	I woke up in the middle of the night.
User	Fill in the blank marked by ___. My friend exercises regularly because he ___ a marathon.
ChatGPT	is training for a marathon.

Table 2.1: Chat model performing the infilling task. There is a difference between the two examples in the form of the answer.

User	Přepiš větu tak, aby místo slova ”kolo” bylo slovo ”barvu”. Mám modré kolo, které se mi líbí.
ChatGPT	Mám **rád** modrou barvu, která se mi líbí.
User	Přepiš větu tak, aby místo slova ”kamarád” bylo slovo ”kamarádi”. Můj kamarád rád sportuje, protože to považuje za důležité.
ChatGPT	Mí kamarádi rádi sportují, protože to považují za důležité.

Table 2.2: Chat model performing the substitution task. In the first example, the model added an extra word. The second answer is correct.

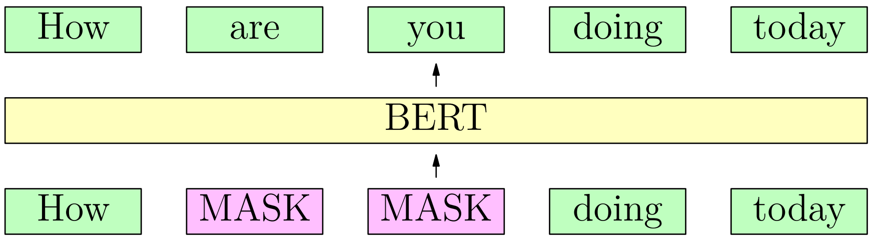

Figure 2.1: BERT predicting the masked words.

model correctly adjusted all the words, but it also added one extra word ”rád” to the sentence, which we did not ask for.

In summary, chat models are capable of performing both infilling and substitution tasks, but their answers only sometimes match the instructions.

2.2 Masked Models

Masked models are encoder-only models that are trained in a fill-in-the-blank fashion. The training process involves masking a portion of the input text and then asking the model to predict the masked words based on the context provided by the unmasked words. This training methodology is used in models like BERT [Devlin et al., 2019].

This training process allows the model to learn a bidirectional understanding of the language, as it has to consider the context from both before and after the masked word to make an accurate prediction. As such, masked models are perfectly suited for the infilling task. We create a mask at the position where we want the model to generate the missing information, and the model will suggest the most likely word to fill in the blank. However, creating a mask is more complex in practice. The model uses special mask tokens to indicate the position of the mask, and it replaces these tokens with the predicted words, as depicted in Figure 2.1. Because of that, the number of mask tokens in the input must be the same as the number of tokens of the prediction. We do not know the length of the prediction in advance, so we create multiple inputs, each with a different number of mask tokens, pass all of them to the model, and then choose the best output.

Prediction with masked models is also possible; we put the mask tokens at the end of the sentence. Once again, we need to create many inputs with different numbers of mask tokens since we do not know how many words remain until the end of the sentence. Because of that, prediction with masked models is more complicated than

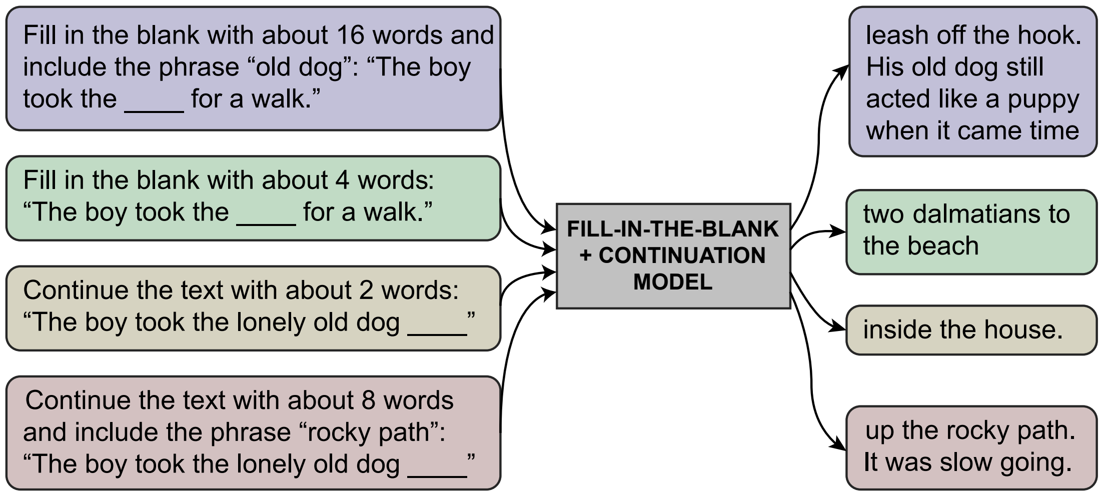
Fill in the blank with about 16 words and include the phrase “old dog”: “The boy took the ____ for a walk.”
Fill in the blank with about 4 words: “The boy took the ____ for a walk.”
Continue the text with about 2 words: “The boy took the lonely old dog ____”
Continue the text with about 8 words and include the phrase “rocky path”: “The boy took the lonely old dog ____”
FILL-IN-THE-BLANK + CONTINUATION MODEL
leash off the hook. His old dog still acted like a puppy when it came time
two dalmatians to the beach
inside the house.
up the rocky path. It was slow going.

Figure 2.2: Examples of tasks for the combined model.

with decoder-only models.

2.3 Combined Model

A recent study by Ippolito et al. [2022] proposed a model that could handle both infilling and substitution tasks. They fine-tune an encoder-decoder model to provide suggestions with fine-grained control over the output. For example, they can instruct the model with an approximate number of words to generate and specify a phrase that should be included in the output, as shown in Figure 2.2. The architecture makes it possible to include a single mask token in the input, as the decoder can generate any number of tokens in its place.

Unlike the first two works, this model is fine-tuned for these specific tasks. As explained in the thesis goal, we want to use a generic pre-trained model because fine-tuning requires many computational resources. Furthermore, it is difficult to adapt a fine-tuned model for additional tasks.

3. Tasks

In the preceding chapters, we have introduced the various tasks that constitute *Preditor*: Substitution, prediction, and infilling. We have also explored a range of existing approaches, each with its strengths and weaknesses.

This chapter describes our approaches to these tasks. We have experimented with multiple strategies to identify the most effective one. The results are discussed in Chapter 5.

As previously stated, our objective is to utilize an existing model without additional training.

3.1 Substitution

The substitution task aims to adjust a sentence to maintain grammatical correctness after the user has replaced one or more words in the sentence.

We narrow the task down to the following: The user selects a part of the sentence to replace. Then, they type a few words to use instead of them: the *replacement*. The assistant then adjusts the forms of the other words in the sentence to maintain grammatical correctness. The replacement is fixed and will not alter. We do not allow the assistant to reorder, add, remove, or change entire words. While such a constraint limits the possible use cases, it simplifies the task and makes it possible to reason about it. We also only consider single-sentence cases.

Our approach to this task can be summarized as follows: First, we generate all possible forms for all words in the sentence. Second, we assume all possible combinations and select the one with the highest score.

3.1.1 Variants Generation

To generate alternate morphological forms of all words in the sentence, we start by analyzing the original sentence (before substitution) using a tagger. We thus get the lemmas and tags for all words.

Next, we create a tag wildcard for each word. This wildcard specifies what grammatical features the word form can have. We do not allow all possible forms because some of them do not make sense under the constraints we put on the substitution task. For example, we do not allow a verb to change its tense because, in Czech, such a change usually adds an auxiliary verb to the sentence. We create the tag wildcard based on the tag we get from the tagger; we keep the original value for some features and allow any value for others. In particular, we allow the assistant to change the following features: Gender, number, person, possessor's gender, possesor's number.

Next, we generate word forms by altering the features listed above. Through this process, we obtain a set of possible forms for each word; an example is shown in Figure 3.1. The next step will be to choose the best combination of these forms.

Preditor employs MorphoDiTa, the state-of-the-art tagger for Czech. MorphoDiTa handles both of the tasks we need: It provides lemmatization and tagging, and it can also generate word forms.

During the generation of variants, we do not generate alternate forms for the replacement to ensure that we keep it in the form that the user provided.

Dnes	jdu	do	školy	.
	jde		škol	
	jdeme			
	jdete			
	jdeš			
	jdou			

Figure 3.1: A sentence and alternate forms of its words.

3.1.2 Candidate Scoring

The number of possible variants is exponential, making it impractical to score all of them. For this reason, we need to decide which variants to score and which to discard.

We observe that the probability of a sequence never increases when we add more words. Consequently, we can construct the variants incrementally, scoring them at each step and extending the ones with the highest probability.

Our approach resembles Dijkstra's algorithm to find the shortest path between two nodes in a graph [Dijkstra, 1959]. We start by generating all variants for the first word and scoring them. We then select the best variant, extend it with all the forms of the second word, and score these extensions again. However, unlike beam search, we do not discard the worst variants. At each step, we select the best variants overall, no matter their length.

It is possible to think of the algorithm as a search in the space of all possible variants. The monotonicity of probability ensures that we will find the optimal solution. However, the sentence with the best score may not necessarily be grammatically correct.

Although still exponential, this approach is significantly faster than scoring all possible variants. We present various optimizations in the next few sections.

Batching

To make the algorithm even faster, we can score multiple sentences simultaneously by creating a batch of inputs. The individual sentences within a batch may vary in length (i.e., they may have a different number of tokens). However, the model expects a rectangular input, so all sentences must have the same length. We use the standard solution: We right-pad the sentences to the length of the longest sentence in the batch. However, this solution comes at a cost, as the padding tokens are scored unnecessarily.

We can utilize batching in several ways. The most straightforward method is to parallelize the scoring of extensions: When selecting the best variant, we score all its extensions in a single batch. Consequently, inference only runs once for each relaxed variant, thereby saving time.

Another strategy is to generate longer extensions. Instead of generating the forms of only the next word, we generate the forms of the next few words. We then construct all possible combinations of these forms. Through this process, we obtain more extensions that can be scored in a single batch. Unlike the previous method, it is not guaranteed that this approach will be faster; many poor variants are scored that would otherwise not have been extended.

The final optimization focuses on relaxing multiple variants simultaneously: We do not only relax the best variant but also the second best, third best, and so on. The downside here is that the variants may have significantly different lengths. This was not too much of a problem in the previous methods, as the sentences had the same number of words, so the number of tokens could have only varied slightly.

We implement all of these optimizations in our solution. Generating longer extensions does not always lead to better results, so the default behavior for the assistant is to only use parallel scoring and batched relaxation.

Length Penalty Heuristic

Unfortunately, the Dijkstra search is exponential due to the need to consider all possible combinations of word forms at each position. The idea behind the algorithm is that incorrect sentences will receive poor scores and will not extend. While this is true, it only helps to a certain extent. The problem is that the probability of a sequence invariably decreases as more words are added. A sentence has many possible continuations, so even the correct one has a low probability. Consequently, correct sentences have scores similar to those of incorrect sentences that are a few words shorter.

Our goal, therefore, is to discount longer sentences. Could we adapt the A* algorithm, which discounts paths closer to the goal? Unfortunately, this method is not applicable to our problem; the distance to the goal could theoretically be zero if all subsequent tokens received a probability of 1.

We opt for a different heuristic, specifically the $lp(Y)$ discount as introduced by Wu et al. [2016]. The function $lp(Y)$, where Y represents the sentence, is defined as follows:

$$lp(Y) = \frac{(5 + |Y|)^\alpha}{(5 + 1)^\alpha}$$

The function increases with the length of the sentence. In our solution, we divide the score of a sentence by this function. With this heuristic, the score is no longer monotonic with length; hence, the algorithm is no longer guaranteed to find the optimal solution.

The hyperparameter α determines the strength of the discount. Higher values of α lead to faster runtime but may yield incorrect results.

Baseline Heuristic

We also considered another way to discount long sentences. The idea is to overcome the problem of the probability of a sentence decreasing as more words are added. We make use of the way we construct sentences: There is a fixed word at each position; only its form changes. We assume that the probability of that word will be similar across all of its forms. Hopefully, only the forms that do not match grammatically will have a lower probability.

We implement this concept with a baseline: We maintain the best log probability for each position in the sentence. The difference between the log probability of a variant and the baseline provides an estimate of the grammatical correctness of the variant. The final score of a sentence is the sum of these differences.

Unfortunately, this heuristic does not perform well. One issue is that the baseline can decrease, which alters the score of the sentence. This violates the condition needed for the Dijkstra search to work. As a result, we do not know when to stop the search. We need to construct more sentences, recalculate the scores and select the best one.

Another issue is that we spend excessive time processing long sentences that are incorrect. The baseline does not allow us to extend short sentences that initially received a poor score. In a sense, it is actually necessary for short sentences to still be extendable.

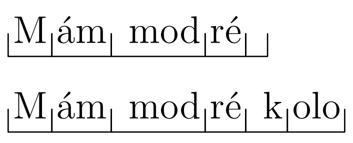

Figure 3.2: Change in tokenization when extending a sentence. The last space changes to the token " k".

Cache

The Transformer computes attention vectors for each token in the input. When we score the extension of a sequence that has already been scored, it is not necessary to recalculate these vectors. We thus adapt the attention cache optimization that is otherwise used during text generation. Specifically, we cache the attention weights for each scored variant. When its extensions are scored, we pass the cache to the model.

We still use batching to speed up the scoring process. This means we need to take the caches for individual sentences and merge them into a single cache for the entire batch. Here, we encounter a similar problem as with the batching of the inputs: The caches may vary in length, and the model expects a rectangular input. Consequently, we trim the caches to the length of the shortest cache in the batch at the cost of recalculating the attention weights for the trimmed tokens.

To minimize this cost, we improve the algorithm that selects the variants to relax. We always select the best variant to ensure that every variant is eventually scored. Subsequently, we consider a *pool* of the best variants and select a subset of the pool with the minimum number of trimmed tokens.

Finding the optimal subset can be achieved algorithmically. We sort the variants based on the length of their cache. We then use a sliding window with a size equal to the number of variants that we want to relax. For each window, we compute the number of trimmed tokens and select the window with the minimum number.

When we pass the cache to the model, it expects that we already have the logits for the old tokens, so it only calculates logits for the new ones. Unlike before, we cannot calculate the negative log probability for the entire sentence at once. We, therefore, calculate it for the extension and add it to the previous variant. However, a problem may arise due to the way that the text is tokenized: The extension of the text may not be an extension in terms of tokens. Specifically, a different token can form at the boundary between the old text and the extension. An example of this can be seen in Figure 3.2.

Fortunately, we can rely on the fact that tokens created by BPE follow some basic rules. Specifically, a token containing letters can only have whitespace at the beginning, never at the end. As long as we ensure that variants do not end with whitespace, tokenization will not change when we extend them.

3.2 Prediction

The goal of prediction is to suggest how a sentence will continue. Achieving this task is simple, as language models are trained precisely for this purpose: We pass the input to the model, and it returns the most probable next token. We append the generated token to the input and repeat the process until we reach the desired length.

But what is the desired length? How long should the prediction be? A single-token prediction is not very useful since the user could type it themselves, and it would be faster than waiting for the model to generate it. A very long prediction will likely diverge from what the user wants to say, which means the user will not accept it. Our goal is to find the optimal length that balances these two extremes. The optimal length differs for each sentence, as some sentences are more predictable than others. For example, if the user starts typing the first few words of a famous quote, the model can finish the entire quote, even if it is long.

We will use the probability of the generated suggestion to find the optimal place to terminate the prediction. The first idea is to choose a probability threshold: If the probability of the suggestion drops below this threshold, we stop generating. This rule ensures that predictable sentences are generated in full while unpredictable sentences are cut short. The user may even choose the threshold to tune the assistant to their preferences.

However, there is one drawback: The threshold may cut off the suggestion in an unnatural place, such as in the middle of a word or in the middle of a common phrase that would become incomplete. Although the probability of such a phrase is high, it is enough to lower the probability of the entire suggestion below the threshold. To address this issue, we introduce a different approach: We find the place in the suggestion where the token probabilities suddenly drop, and we terminate the suggestion there.

3.2.1 Usefulness

We base the approach on the concept of expected value. We aim to maximize the *usefulness* of the suggestion: If the user accepts the suggestion, the usefulness u is equal to the length ℓ of the suggestion. Otherwise, the usefulness is zero:

$$u = \begin{cases} \ell & \text{if the suggestion is accepted} \\ 0 & \text{otherwise} \end{cases}$$

We can calculate the expected usefulness as the product of the length of the suggestion and the probability of acceptance p_a:

$$\mathsf{E}[u] = \ell \cdot p_a + 0 \cdot (1 - p_a) = \ell \cdot p_a$$

We could model the probability of acceptance with the probability p_s of the suggestion; however, the probability usually decreases too fast, so we introduce a hyperparameter called *confidence* that controls the rate of decrease. We raise the probability to the power of the inverse of the confidence; since we usually work with log probabilities, we divide the log probability by the confidence. In the end, the expected value is a function of the probability p_s of the suggestion, its length ℓ, and the confidence c:

$$\begin{aligned} \mathsf{E}[u] &= \ell \cdot p_a \\ \mathsf{E}[u] &= \ell \cdot p_s^{1/c} \\ \mathsf{E}[u] &= \ell \cdot e^{(\log p_s)/c} \end{aligned}$$

Tweaking the value of confidence allows the user to adjust the length of the prediction, similar to the threshold approach. Higher confidence values result in longer predictions. In fact, the usefulness approach also favors predictable sentences, so it fully replaces the threshold approach.

```
Fill in the blank marked by ___.
My friend exercises regularly because he ___ a marathon.
My friend exercises regularly because he
```

Figure 3.3: Infilling prompt using a blank marker.

```
Write a sentence such that it ends with: a marathon.
My friend exercises regularly because he
```

Figure 3.4: Infilling prompt using the sentence end.

3.3 Infilling

Infilling aims to suggest a few words that fit into a sentence. Our primary challenge during this task is to consider both the left and right context. Unfortunately, decoder-only models cannot consider the right context, as they process the input from left to right. To address this issue, we generate many possible continuations of the left context and select the one that best aligns with the right context.

3.3.1 Generation

We use the beam search algorithm to generate multiple distinct continuations. We also employ a similarity penalty to ensure that the continuations are diverse. However, even with many continuations, it's still a matter of luck that the right context will match. So, we add a hint about the right context to the input. We create a prompt that includes both the left and right context and an instruction to generate the missing part.

Blank marker Our first attempt at such a prompt includes the sentence with a blank at the missing part and an instruction to fill in the blank. We also include the start of the sentence, as the model would need to generate it anyway. The resulting prompt can be seen in Figure 3.3. We let the model finish the output, generating the missing part. However, the model often ignores the instruction and repeats the blank marker in the output. To solve this, we suppress such tokens during generation.

Sentence end Our second approach avoids blank markers altogether. The left context is already a part of the final sentence, so it does not have to be a part of the instruction. Hence, we only include the right context and instruct the model to use it at the end of the sentence. The resulting prompt is shown in Figure 3.4.

Both approaches add special instructions to the input. When running the assistant, we need to choose the language of these instructions: If the infilled sentence is in Czech, it also makes sense to write the instruction in Czech. This also helps to clarify the language of short inputs. So, we detect the language of the input using FastText [Joulin et al., 2016] and decide which language to use for the instruction.

Another problem arises due to tokenization. The last word of the left context is not terminated, so the model may extend the word with another token. For example, the word "he" may become "health". Adding a space to the prompt does not work well either, as it creates a separate space token; this behavior is shown in Figure 3.5. In contrast, the model usually generates a space combined with a word as a single token. Consequently, seeing a single space makes the model generate special tokens

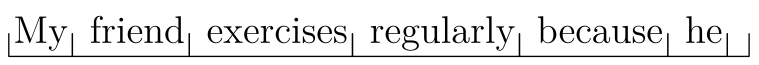

Figure 3.5: A trailing space creating a separate space token.

that usually occur after a space. To solve this, we enforce the first token to start with a space during generation. Unfortunately, we need to assume the tokenization strategy of the model, which limits the models our assistant is compatible with.

3.3.2 Selection

After generating multiple sentence variants, we need to select the most suitable one. Generally, it makes sense to select the sentence with the highest probability. However, the generated sentences are not guaranteed to adhere to the given instructions. Sometimes, the model generates a sentence that ends differently. To mitigate this, we try to find the sentences that best match the right context. Initially, we only consider sentences that are an exact match. We select the one with the highest probability if there are such sentences. Otherwise, we look for sentences that contain the first word of the right context.

However, the sentences may not match at all. Therefore, we develop another strategy to select the best sentence. The idea is to start with the generated infill, manually concatenate it with the correct context, and compute the probability of the resulting sentence. While this method ensures the correct ending, the infill and the right context may not connect well. Specifically, the infill may already contain a part of the right context, or the last word of the infill may be incomplete because only some of its tokens have been generated.

To address this, we assume all prefixes of the infill. For instance, if the generated infill was ”is training for a mara”, we create the prefixes ”is”, ”is training”, ”is training for,” and ”is training for a”. This process is repeated for every infill; the duplicates are removed, and the prefixes are concatenated with the right context. Subsequently, we compute the probability of each sentence and select the one with the highest probability.

4. Assistant

In the previous chapters, we described the tasks that *Preditor* performs and we explored various strategies to solve them. In this chapter, we focus on the assistant itself.

Our assistant has the form of an editor extension. Modern text editors make it possible to extend their functionality with plugins. These plugins can provide suggestions and refactor code, among other things. We can use this functionality to implement our tasks. In this thesis, we write an extension for Visual Studio Code.

The assistant needs a lot of computing power to perform its tasks, as it runs a large language model. We cannot expect users to run this model on their machines. Therefore, we design the assistant using the client-server architecture. The server is responsible for running the model and performing the computations for the tasks. We expect the server to run on a powerful machine with a GPU, perhaps in the cloud. The client is realized by the extension, and it is responsible for the user interface and communication with the editor. It forwards the user's requests to the server and displays the results.

Each editor needs a different extension, and this architecture makes it possible to develop many extensions. Therefore, we want to make the extensions as simple as possible. We move as much logic as possible to the server; each extension only needs to handle the user interface and communication with the server.

4.1 Server

The server part is implemented as a Python package and uses the Python package manager `pip` for installation. The repository is available on GitHub.[1]

The server has the following responsibilities: It loads the language model and the tagger and provides functions that expose their functionality. It listens for user requests with an API and dispatches them to the appropriate task. Finally, it contains the logic for the various task strategies, and it dispatches requests to these strategies.

4.1.1 Models

The server loads three models: the language model, the tagger model, and the FastText model. We need to specify where the server should find these models. We use environment variables to avoid hardcoding paths:

- `PREDITOR_MODEL_PATH`
- `PREDITOR_FASTTEXT_PATH`
- `PREDITOR_TAGGER_PATH`

Users can set the environment variables in the shell or with a `.env` file. We provide a utility script `download-models.sh` that downloads the models to the `models` directory and configures the environment variables in the `.env` file.

Language model The server needs a language model to generate text predictions and compute sentence scores. It can load this model from a local file or download it from HuggingFace.[2] In the latter case, the HuggingFace library caches the model locally

[1] https://github.com/kulisak12/preditor-model

[2] https://huggingface.co/models

so that it does not need to download it again. The model also includes a tokenizer, which is necessary to convert text to tokens and back.

The server defines a `Model` abstract class that holds the model and the tokenizer since these objects are always used together. The tasks use its subclass `HFModel`, which loads an actual model from HuggingFace. This distinction makes it possible to create mock models for testing. Both classes are defined in the `model/` directory.

Several other files define functionality related to the model. The `nlp.py` file contains functions that compute the sentence scores, and the `caching.py` file provides functions for manipulating the attention cache. The individual tasks define their own functions that use the model, a prominent example being the `infilling/generation.py` file that generates text for the infilling task with several constraints on allowed tokens.

Tagger model The MorphoDiTa tagger requires a tagger model to perform its tasks. The model can be downloaded from the LINDAT repository.[3] There are several variants available; we use the full variant, i.e., the variant that predicts all tags and uses diacritics.

All the tagger functionality is implemented in the `tag.py` file, which provides tokenization, tagging, and variant generation.

FastText model Finally, the server needs a FastText to determine the language of the input text. The FastText model is available from the FastText website.[4] We use the compressed model, as it fully suffices for our purposes. The language identification is implemented in the `language.py` file.

4.1.2 Flask API

The server handles requests using the Flask framework.[5] We use a Gunicorn[6] server to run the Flask application, as the built-in Flask server is only intended for development.

The application entry point is the file `server.py`. The API listens for POST requests on two endpoints:

- `/suggest` for the prediction and infilling tasks
- `/substitute` for the substitution task

Both the request and the response are in JSON format; the `README.md` file in the repository describes the API in detail. The request contains the task input data and the task-specific configuration. This configuration allows the user to tweak some task parameters, such as output length or the strength of the heuristic. By including the configuration in the request, we make it possible to adjust the task's behavior without restarting the server.

We parse the request using Pydantic[7] to validate the input and provide default values. We return an error response with an appropriate status code if the request is invalid. The error response format is the same for both endpoints.

We considered including user authentication in the API, such as an API token. In the end, we decided against it to simplify the setup. However, if the server is deployed in a public environment, we advise adding some kind of authentication, as the server's long processing times make it vulnerable to denial-of-service attacks.

[3] https://lindat.mff.cuni.cz/repository/xmlui/handle/11234/1-4794
[4] https://fasttext.cc/docs/en/language-identification.html
[5] https://flask.palletsprojects.com/en/3.0.x
[6] https://gunicorn.org
[7] https://docs.pydantic.dev/latest

4.1.3 Tasks

We discussed several strategies that the assistant can use to solve the tasks. These strategies are located in the directories `substitution/`, `prediction/`, and `infilling/`. Each directory contains a file with the same name as the directory that serves as the entry point for that task. This file handles configuration, prepares the input and calls the appropriate strategy. We provide a utility function that handles the entire request using the best strategy; however, we implement the logic so that it is easy to call a different strategy if needed.

Prediction & Infilling We decided to join the prediction and infilling tasks into one endpoint (`/suggest`) so that the editor extension does not need to distinguish between them. The extension sends the text around the cursor, and the server decides which task to perform based on the context. In particular, it trims the text to the current paragraph (a sequence of non-empty lines). If the cursor is at the end of the paragraph, the server performs the prediction task; otherwise, it performs the infilling task. We also join the lines in the paragraph with spaces to create a single line of text since the model is trained on continuous text. We pass this input to the task and return the result.

Substitution Our implementation of substitution expects to receive a single sentence. To keep the extension simple, we let it send the entire text around the cursor, and we leave it to the server to extract the sentence containing the replaced word. To this end, we utilize the tagger, which can split the text into sentences.

4.2 Language Model

We had the following requirements when choosing the language model:

- It is available in the HuggingFace model hub.
- It uses the decoder-only Transformer architecture.
- It supports Czech.
- It is fast enough to provide suggestions in a reasonable time.
- It fits on a single GPU to keep the resource consumption low.

For a long time, no such model could generate quality text in Czech. During the development, we have used the Falcon model[8] [Almazrouei et al., 2023], which is trained primarily on English data, so its capabilities in Czech are limited. Therefore, the results of the assistant were suboptimal. However, in March 2024, two new models were released: CSTinyLlama-1.2B[9] and csmpt7b[10] [Fajčík et al., 2024]; we chose the former because of its smaller size. It paid off that we designed the assistant to be model-agnostic; because of that, switching to a new model was straightforward.

We also considered using a conversational model. Such a model would likely perform better on the infilling task, where we give it the instruction to fill in the blank. However, no such model is available for Czech at the time of writing.

[8]https://huggingface.co/tiiuae/falcon-7b
[9]https://huggingface.co/BUT-FIT/CSTinyLlama-1.2B
[10]https://huggingface.co/BUT-FIT/csmpt7b

4.3 Editor Extension

We provide an extension for the popular editor Visual Studio Code. It is available in the Visual Studio Code Marketplace,[11] so users can install it directly from the editor. The source code is available in a separate GitHub repository.[12] The extension is written in TypeScript.

The extension defines a *completion provider*, which is how extensions can provide suggestions to the editor. Whenever the user types a trigger character, such as a space, the extension sends the text around the cursor to the server. Once the response comes, it displays the suggestion that the user can accept by pressing the `Tab` key.

The extension also defines a *rename provider*, which typically allows the user to rename variables in the code. We repurpose it to perform the substitution task. The user can use the rename keyboard shortcut (`F2`) to replace the word under the cursor. Afterwards, the extension sends the replaced word and the text before and after it to the server. The server returns the adjusted text, which the extension then replaces in the editor.

Lastly, the extension contributes several configuration settings. Using these settings, the user can configure the server's URL and the amount of context that the extension sends to the server. Apart from that, the settings contain the task-specific configuration that the user can adjust, such as the strength of the length penalty heuristic introduced in Section 3.1.2.

[11] https://marketplace.visualstudio.com/items?itemName=kulisak12.preditor
[12] https://github.com/kulisak12/preditor

5. Evaluation

In this chapter, we explore the accuracy and performance of the proposed strategies for *Preditor*. The results depend on the language model we use; we run the evaluation using the CSTinyLlama-1.2B model. We compare the individual strategies and highlight their strengths and weaknesses.

5.1 Substitution

5.1.1 Dataset

No datasets were available for the substitution task, so we created our own. We picked five types of modifications that we identified as the most common use cases for the substitution task:

- change in number
- change in gender
- change in gender and number
- change in person
- change in person and number

To ensure the dataset is balanced, we created 20 examples for each type, resulting in 100 examples in total. Each example contains the original sentence, the replacement, and the expected sentence. We picked suitable sentences from the Czech NewsCrawl 2007 dataset[1] that had a mostly clear answer; we sometimes modified them to fit the task. We only used sentences shorter than about 100 characters, as longer sentences take too long to evaluate.

While creating the dataset, we spotted some limits of the task as we defined it. In particular, we did not allow the assistant to change the number of words, but that is sometimes necessary. For example, consider the sentence *Já jsem si dal pivo.* If we change it from the first person to the second person, we get *Ty sis dal pivo.* Our assistant cannot handle such cases, as it would need to combine two words into one.

5.1.2 Results

We evaluate four configurations. There are two strategies: *cache* uses the cache optimization, *simple* does not. For each of them, we evaluate three values of the hyperparameter α for the $lp(Y)$ heuristic. The value of $\alpha = 0$ is equivalent to not using the heuristic at all.

The metrics are as follows:

- *Time* is the average time it takes to generate the substitution.
- *Accuracy* is the percentage of fully correct substitutions.
- *Good* is the number of good word changes. A word change is good if it reduces the number of incorrect forms in the sentence — it is a change that the assistant should make.

[1] https://data.statmt.org/news-crawl/cs/

Strategy	α	Time	Accuracy	Good	Bad	Missed
simple	0.0	15.68 s	60 %	154	37	33
simple	0.5	4.06 s	60 %	154	37	33
simple	1.0	1.47 s	60 %	154	36	33
cache	0.0	6.12 s	61 %	156	34	31
cache	0.5	1.79 s	60 %	154	34	33
cache	1.0	0.74 s	61 %	154	32	33

Table 5.1: Evaluation of the substitution task.

Original	Od 18:00 bude hostem Impulsů Václava Moravce.
Replaced	Od 18:00 budou **hostem** Impulsů Václava Moravce.
Original	Je prvním novým členem Evropské unie, který se žezla ujímá.
Replaced	**Jsme** první novou zemí Evropské unie, která se žezla ujímá.
Original	Tyto domy jsou cenově příznivé.
Replaced	Tato chata je cenově příznivá.
Original	Mám z toho radost, ale beru to s rezervou.
Replaced	Má z toho radost, ale bere to s rezervou.
Original	Ani já jím nejsem.
Replaced	Ani vy jimi nejste.

Table 5.2: Outputs of the substitution task. The replacement is underlined; the wrong words are in bold.

- *Bad* is the number of bad word changes. A word change is bad if it increases the number of incorrect forms in the sentence — the assistant changes a word that should be left unchanged.
- *Missed* is the number of missed word changes. A word change is missed if it does not change the number of incorrect forms in the sentence — the assistant either does not change a word it should change or changes an incorrect word to another incorrect word.

Table 5.1 shows the results of the evaluation. Over half of the substitutions were entirely correct, and many of the remaining ones still produced grammatically correct sentences. The number of good changes outweighs the rest, which is also a good sign.

The simple and cache strategies achieve similar accuracy, which is expected since their algorithm is almost the same and only differs in implementation. The cache optimization reduces the time by more than half, a significant improvement. The heuristic helps, too; the higher the hyperparameter α, the larger the speedup. Surprisingly, the heuristic does not affect the accuracy.

Table 5.2 shows some outputs in detail. A common mistake is changing a word that does not depend on the replaced word; the second example in the table is one such case. That is one of the downsides of the approach we use; the assistant chooses the word forms that best fit the sentence, but it does not consider whether it depends on the replaced word or not.

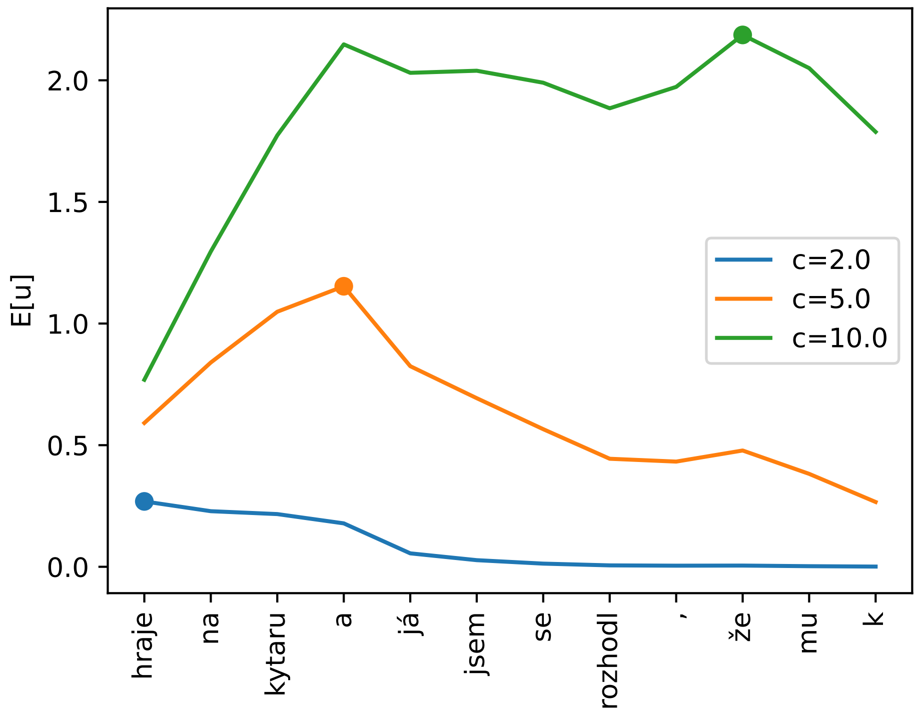

Figure 5.1: The continuation of the sentence ”Můj kamarád rád” and the expected usefulness of the suggestions for several confidence values. The highest score for each confidence value is highlighted. For $c = 5.0$, the assistant would suggest ”hraje na kytaru a”.

5.2 Prediction

We do not evaluate the quality of the predicted sentences because our assistant directly uses the prediction of the language model, and the quality of the language model is not the focus of this work.

Still, we can evaluate the strategy that chooses the cutoff point for the suggestion. However, this task is highly subjective, so we only provide a non-scientific, empirical evaluation.

Figure 5.1 shows the suggestion that the assistant would make for several confidence values; it selects the prefix that ends at the position with the highest score; this position is also included. All of the suggestions end at a reasonable place where it is no longer clear how the sentence should continue. For example, the phrase ”já jsem se rozhodl,” has an obvious continuation ”že”, so it makes sense to include that word in the suggestion. Afterward, the sentence could continue in many ways, so the assistant does not suggest any more words. Sometimes, the model generates a period and continues with another sentence. In such a case, the assistant almost always terminates the suggestion after the period, which is also a reasonable choice.

Generation	Selection	Time	Accuracy
predict (baseline)	match	0.35 s	7.1 %
predict (baseline)	score	0.51 s	9.8 %
blank	match	2.03 s	7.2 %
blank	score	2.61 s	11.2 %
end	match	1.62 s	6.2 %
end	score	2.27 s	**12.7 %**

Table 5.3: Evaluation of the infilling task.

5.3 Infilling

5.3.1 Datasets

We created two datasets for the infilling task. First, we created a large dataset with 1000 automatically generated examples to compare the strategies. Then, we manually created a tiny dataset with 30 examples that have a mostly clear answer to get an idea of the absolute performance.

For the first dataset, we take sentences from the Czech NewsCrawl dataset and randomly remove one to three consecutive words from them. We only keep sentences at least 8 words long to ensure some context is present. However, such automatic generation often generates sentences that have no clear answer. Therefore, the second dataset contains only examples that have a clear answer; in other words, a human would correctly fill in the blank in most cases. Moreover, the answer depends on both the left and right context.

5.3.2 Results

We evaluate our two strategies: *blank*, which uses a blank marker, and *end*, which instructs the assistant to generate a sentence with a given ending. Additionally, we use simple prediction without the right context as a baseline. We combine each of these strategies with the two selection strategies: *match* that selects the variant that most resembles the original sentence, and *score* that scores all variants using the language model and selects the best one.

We use the following metrics:

- *Time* is the average time it takes to generate the infilling.
- *Accuracy* is the percentage of fully correct infillings.

Table 5.3 shows the evaluation results on the large dataset. The accuracy scores are pretty low, partly due to the sentences not having a clear answer but also because the assistant does not do well on this task.

The score selection strategy consistently outperforms the match strategy at the cost of a slightly longer suggestion time. Both the blank and the end strategy achieve higher accuracy than the baseline prediction strategy, but the difference is fairly small.

However, prediction is much faster than the other two strategies, which makes it more convenient for the user. Time-wise, the bad performance of the blank and end strategies is due to the beam search algorithm they use; its implementation in the Transformers library unnecessarily calculates attention values multiple times.[2] This issue may be fixed in the future, making the strategies almost as fast as prediction.

[2]https://github.com/huggingface/transformers/issues/27449

Generation	Selection	Accuracy
predict (baseline)	match	13.3 %
predict (baseline)	score	16.7 %
blank	match	13.3 %
blank	score	23.3 %
end	match	16.7 %
end	score	**30.0 %**

Table 5.4: Evaluation of the infilling task on the manually created dataset.

Prosíme která budou provoz blokovat, budou odtažena. *(Vozidla)*
Dodatek by měl směřovat k těžbě na těžbě lithia profitovala česká ekonomika. *(tomu, aby)*
Vyplývá to z předběžných výsledků, které dnes zveřejnil Český statistický úřad (ČSÚ).

Table 5.5: Outputs of the infilling task. The underline marks the infill. The first two examples show common mistakes; the correct answer is in italics. The last example shows a correct infill that depends on the right context.

Table 5.4 shows the evaluation results on the manually created dataset. The accuracy of our strategies is better than that of the large dataset, but it is still far from human performance.

The assistant struggles most when the blank is at the beginning of the sentence; it made a mistake in nearly all such cases, which is likely because the model is not trained to follow instructions. Another common mistake is repeating the word after the blank. Table 5.5 depicts these mistakes and one correct output.

6. Conclusion

In this thesis, we explored various ways in which a writing assistant can help users with writing. We focused on three tasks: rewriting a sentence after replacing one of its words, suggesting continuations of a sentence, and suggesting words that fit into a sentence. We developed several strategies for the tasks and evaluated them to choose the best ones. Finally, we integrated the strategies into our assistant, *Preditor*. We provided a server part that processes the tasks and exposes them through a REST API, which an editor extension can access. We developed one such extension for the Visual Studio Code editor.

We were able to handle all three tasks using a single pre-trained large language model, which minimizes the memory footprint and makes the assistant suitable for personal use. Moreover, we designed the system to be model-agnostic, which makes it possible to upgrade to a new model with little effort. We successfully performed such an upgrade during development when a better Czech language model came out. Initially, we aimed to make the assistant compatible with all decoder-only Transformer models. Due to implementation details, we had to restrict ourselves to models with a tokenization strategy that positions spaces at the start of tokens. Fortunately, most available models utilize this tokenization approach.

Substitution Our assistant can rewrite a sentence when the user replaces one of its words to make it grammatical. We restricted ourselves to a single sentence and only altered the forms of words; we did not rearrange, add or delete words. We introduced an algorithm that constructs variants for the new sentence by altering word forms and uses the probability the language model assigns to the sentences to choose the best one. This approach works well, correctly adjusting the sentence in a majority of cases. Unfortunately, its use case is limited due to the mentioned restrictions.

Prediction Next, our assistant suggests continuations of a sentence. A language model can generate continuations with ease; we introduce an approach that finds the best part of the continuation to suggest to the user. The approach considers the length of the suggestion and the probability assigned to it by the language model. We did not conduct a thorough evaluation, but empirically, this strategy leads to good suggestions.

Infilling The final task focuses on providing suggestions in the middle of a sentence. We designed an approach that prompts the model to generate words that fit into the sentence; we generate multiple such variants and then select the one that fits the best. We suggest two strategies for generation and two for selection. Our approach achieves better results than generating continuations of the text before the gap but still has room for improvement.

6.1 Future Work

The system we developed is a good starting point for a writing assistant. Still, the accuracy and performance of the strategies can be improved. We present several ideas for future work.

6.1.1 Substitution

Our algorithm for substitution works well in most cases but often fails because it changes a word that does not depend on the replaced word. One possible remedy is to favor the forms of the words that occurred in the original sentence. It remains unanswered how to balance the favoring so that it does not prevent the algorithm from making necessary changes.

We only used linguistic analysis to generate alternate word forms. Yet, syntactic analysis could provide more information about the sentence structure. We believe that a language model is still necessary to capture the semantics of the sentence. However, syntactic analysis can help decide which words to change and limit the number of possible forms.

We only focused on single-sentence scenarios. Our current approach has exponential complexity in the number of words that need adjusting, so directly extending it to multiple sentences seems infeasible. We believe that an iterative approach could work, where we adjust one sentence at a time but take into account the context of the other sentences.

The possible use cases of substitution would greatly expand if we allowed the reordering of words. A different approach would be necessary; one idea is to make small adjustments to the sentence iteratively. We could even consider adding or deleting words, but ensuring that the sentence's meaning remains the same becomes difficult.

6.1.2 Prediction

The current implementation of the prediction task first generates a continuation of a fixed length and then selects the part to suggest to the user. This approach is ineffective, as the model often generates a lot of text that we discard. Thus, it would be beneficial to adjust our approach to terminate the generation early when the continuation seems unlikely to be useful.

6.1.3 Infilling

Chat models show promising results when filling in gaps in a sentence but do not always follow the instructions. We believe that our approach would also work with a chat model; thus, it would solve this issue and achieve better results.

Our implementation does not use cache optimization when selecting the best variant. We omitted it because it is relatively fast, even without the cache. However, the optimization would speed up the selection process by a few tenths of a second.

6.1.4 Assistant

We designed the server to be stateless; it handles each request independently. Because of this, it processes each text snippet from scratch. Hence, the extension only sends a small part of the text to the server; otherwise, it will take a long time to process. Thus, one possible improvement is to make the server stateful. If the server could remember the attention states from previous requests, it could reuse them when processing a new request, making it possible to use a larger context.

Another possible improvement is to implement the cancelation of requests. Currently, the server processes each request to completion. In some situations, the user might want to cancel the request, for example, when the server takes too long to respond. Cancelation would free up resources for subsequent requests.

Bibliography

Ebtesam Almazrouei, Hamza Alobeidli, Abdulaziz Alshamsi, Alessandro Cappelli, Ruxandra Cojocaru, Mérouane Debbah, Étienne Goffinet, Daniel Hesslow, Julien Launay, Quentin Malartic, et al. The falcon series of open language models. *arXiv preprint arXiv:2311.16867*, 2023.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In Jill Burstein, Christy Doran, and Thamar Solorio, editors, *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics. doi: 10.18653/v1/N19-1423. URL https://aclanthology.org/N19-1423.

Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

Martin Fajčík, Martin Dočekal, Jan Doležal, Karel Beneš, and Michal Hradiš. Benczechmark: Machine language understanding benchmark for czech language. March 2024. URL https://huggingface.co/BUT-FIT/CSTinyLlama-1.2B.

J. Hajič. *Disambiguation of Rich Inflection: Computational Morphology of Czech*. Karolinum, 2004. ISBN 9788024602820. URL https://books.google.cz/books?id=sB63AAAACAAJ.

Daphne Ippolito, Liam Dugan, Emily Reif, Ann Yuan, Andy Coenen, and Chris Callison-Burch. The case for a single model that can both generate continuations and fill-in-the-blank. In Marine Carpuat, Marie-Catherine de Marneffe, and Ivan Vladimir Meza Ruiz, editors, *Findings of the Association for Computational Linguistics: NAACL 2022*, pages 2421–2432, Seattle, United States, July 2022. Association for Computational Linguistics. doi: 10.18653/v1/2022.findings-naacl.185. URL https://aclanthology.org/2022.findings-naacl.185.

Armand Joulin, Edouard Grave, Piotr Bojanowski, and Tomas Mikolov. Bag of tricks for efficient text classification. *arXiv preprint arXiv:1607.01759*, 2016.

Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. Improving language understanding by generative pre-training. 2018.

Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. In Katrin Erk and Noah A. Smith, editors, *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1715–1725, Berlin, Germany, August 2016. Association for Computational Linguistics. doi: 10.18653/v1/P16-1162. URL https://aclanthology.org/P16-1162.

Jana Straková, Milan Straka, and Jan Hajič. Open-source tools for morphology, lemmatization, POS tagging and named entity recognition. In Kalina Bontcheva and Jingbo Zhu, editors, *Proceedings of 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, pages 13–18, Baltimore, Maryland, June 2014. Association for Computational Linguistics. doi: 10.3115/v1/P14-5003. URL https://aclanthology.org/P14-5003.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.

List of Figures

List of Tables