From Examples to Patterns: LLM-Generated Regular Expressions for Entity Extraction in Czech Clinical Texts

Petr Zelina

Faculty of Informatics, Masaryk University, Brno, 601 77, Czechia
xzelina2@fi.muni.cz

Abstract. Entity extraction in clinical texts is essential for converting unstructured data in clinical notes into structured formats, facilitating large-scale analysis and clinical decision support. Traditional methods often rely on handcrafted regular expressions (regexes), which, while effective, demand significant time and specialized knowledge to create – resources that healthcare professionals may lack. We introduce a novel approach leveraging large language models (LLMs) to automate regex generation for clinical entity extraction. Our method involves prompting LLMs to generate regex patterns from examples, followed by iterative refinement using a feedback loop. Despite regex limitations, this approach is practical for extracting frequently patterned information common in clinical texts, such as dates, specific data about medical procedures or event detection. Our experiments on Czech clinical notes show this method outperforms current SOTA genetic-programming-based methods for generating regular expression patterns from examples, especially when there are few of them.

Keywords: NLP, LLM, regular expressions, text mining, clinical notes.

1 Introduction

Entity extraction in clinical texts is crucial for transforming unstructured information in clinical notes into structured data, enabling various applications such as clinical decision support, large-scale data analysis or patient cohort selection. One of the traditional methods for extracting entities are handcrafted regular expressions (regexes). Once developed, regular expressions offer several benefits, including speed, portability, and ease of use and deployment. However, creating regex patterns manually is time-consuming and complex, particularly in domains like medicine, where specialized knowledge is essential. Additionally, healthcare professionals may not have the technical expertise required to develop regular expressions, further complicating the manual creation process.

To address these challenges, we propose a novel method that leverages large language models (LLMs) to generate regular expressions automatically. In the most basic form, we show the LLM examples of the clinical extractions and ask it to generate regex patterns that match them. This provides a foundation

A. Horák, P. Rychlý, A. Rambousek (eds.): Proceedings of Recent Advances in Slavonic Natural Language Processing, RASLAN 2024, pp. 3–16, 2024. © Tribun EU 2024

upon which regex patterns can be iteratively improved by using a feedback mechanism similar to the Tree of Thought approach [10]. Through this iterative process, we expose the model to false positives and false negatives, improving precision and recall. We also branch the LLM conversations into a search tree to leverage the inherent parallelizability of LLMs, improve training speed and stabilize the overall performance of this method.

The limitations of regular expressions mean that this approach is useful for entities that follow a limited number of patterns; however, this is often the case in clinical notes. Examples include extraction of date or time (e.g. for anonymization or event timestamping), searching whether some defined event occurred, or extracting additional information about treatments or procedures missing from structured records.

Recent developments in open large language models (like llama, gemma or mistral) make them usable even for less frequent languages. They enable offline, on-premise use, more easily satisfying privacy requirements.

The source code is available on GitHub[1]

2 Related Work

We can broadly classify approaches for creating regex patterns into three categories – Generating from natural descriptions, generating from in-context examples (annotations) and generating grammars that distinguish between positive and negative examples.

Regex from natural descriptions Kushman and Barzilay (2013) [2] create a syntactic tree of the description and transform it into a regex. Locascio et al. (2016) [5] formulate it as a sequence translation task and apply recurrent neural networks to solve it. However, Zhong et al. (2018a) [13] suggests these approaches fail on real-world datasets.

Zhong et al. (2018b) [12] use expected semantic correctness as the optimization target to mitigate the fact that syntactically different regexes can have the same semantic meaning.

Regex from in-context examples Bartolio et al. (2016) [1] introduce a tool called RegexGenerator++, which uses genetic programming to evolve pattern candidates. As opposed to other methods, it learns from in-context examples, which is exactly the use case for our method as well. It is also one of the few solutions that have complete, working, and easily runnable source code. For these reasons, we compare our method with this one.

Regex from positive and negative matches These methods use a set of positive and a set of negative examples and try to differentiate between them. A prototypical example of this is RegexGolf[2].

[1] `https://github.com/ZepZep/regexulator`
[2] `https://alf.nu/RegexGolf`

Lee et al. (2016) [3] introduce AlphaRegex, which enumerates over possible regexes while using over- and under-approximations to prune the search space.

Li et al. (2020) [4] focus on creating linear-time regex patterns only to avoid Regex Denial of Service attacks.

The fAST approach – Raynal et al. (2023) [7] – infers a grammar based on positive examples only (without context). They offer an easily installable Python module. However, it crashes with an import error as of writing of this paper.

Ye et al. (2020) [11] use both natural descriptions and examples and combine a semantic parser with a program synthesizer. They first let the parser create an incomplete sketch with holes and then use the synthesizer to fill them in so that the pattern agrees with the examples.

3 Proposed Method

The task is to create a regular expression pattern based on example matches.

3.1 Overview

The main idea of the proposed method is to show examples of matches (snippets) to a generative large language model and ask it to develop a regular expression pattern.

To improve this zero-shot scheme, we use an idea similar to Tree of Thought [10] – we iterate on the pattern discovered so far, showing relevant examples (false positives and false negatives) and branching into different paths to achieve more stable performance.

Even though LLM text generation is relatively slow, it can often be efficiently parallelized. We can use this to expand multiple nodes in the branching graph simultaneously. The graph search can be limited by the maximum graph size or a time limit.

To facilitate this, we use two types of nodes – *start* nodes, which are used at the beginning to find initial patterns, and *improve* nodes, which take the current pattern and iterate on it.

We select which nodes to evaluate next based on the validation performance and position in the tree, allowing for more efficient use of resources.

A more rigorous description is available in the following sections.

3.2 Training data

Example set Let E be the Example set. Each example $e_i \in E$ is a pair (d_i, M_i) where d_i is document text and M_i is a set of snippets (matches, extractions). Each $m_{i,j} \in M_i$ is a pair $(l_{i,j}, r_{i,j})$ of indexes into the document d_i, where $l_{i,j}$ points to the first character of the extraction and $r_{i,j}$ points to the last character of the extraction plus one.

Dataset splits In order to improve generalization and maintain good machine learning practices, we can use multiple splits of the example dataset.

- **Training set** T is used during training – the model can use the examples in it directly.
- **Validation set** V is used for comparing different pattern candidates during training. The model does not see its examples directly but has access to the evaluation metrics it provides.
- **Test set** F is used only once after the training to get the final performance for the predicted pattern.

If no validation set is provided, the method uses the training set for both training and comparing nodes.

3.3 Building the exploration tree

We maintain two data structures: a *tree* of nodes, which represents the exploration and dependency of patterns discovered so far, and a priority queue of nodes from the tree that are awaiting evaluation (*task queue*). An example of a small exploration tree is depicted in Figure 1.

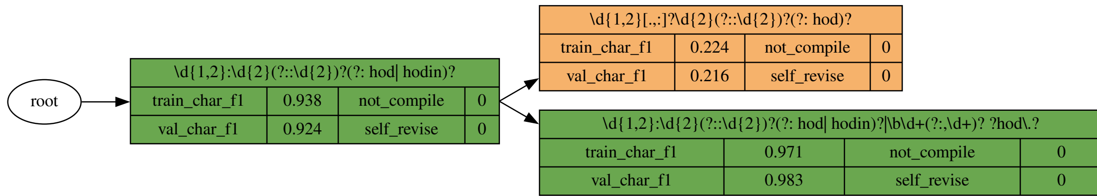

Fig. 1: A small exploration tree with `initial_nodes` = 1, `maximum_depth` = 2 and `branching_factor` = 2. The rectangle records show the resulting pattern of each node, primary training and validation metric and number of internal checks performed. We can see that the bottom *improve node* in the second level led to an improvement, but the top one did not.

We start by initializing several (`initial_splits`) `start` nodes connected to the root of the exploration *tree*. We also add them to the *task queue* with a high priority.

To evaluate nodes from the *task queue*, we use a pool of concurrent workers, which take nodes from the queue and execute them. The execution entails selecting examples to show to the LLM, the conversation with the LLM itself, and extracting the resulting pattern from the LLM's answer. (more details in section 3.4)

After a node is executed, the resulting pattern is evaluated on the validation dataset V. Based on the `branching_factor`, new *improve* nodes are created, initialized with the resulting pattern and appended to the tree as children of the executed node. They are also added to the task queue with priority

$$\texttt{validaton_score} \times \texttt{depth_base}^{\texttt{depth}} \times \texttt{sibling_base}^{\texttt{sibling_index}}$$

where `depth_base` and `sibling_base` are configurable parameters. The `validation score` can be any metric calculated on the validation dataset. By default it is the mean of `character_f1` and `match_f1` scores (`mean_f1`; see 5.1).

This ensures that the most promising candidates are evaluated first (exploitation) while also maintaining a broader exploration front.

There are two ways for the exploration to stop

- All nodes in the tree are executed. The size of the tree depends on the `initial_nodes`, `maximum_depth` and the `branching_factor` parameters.
- The `time_limit` is reached. From this point, no new nodes start execution, and the currently running ones are awaited.

3.4 Nodes

A node encapsulates one conversation with the LLM. Its execution should result in a new pattern.

Types of nodes We implement two types of nodes

- The *start node* is used at the beginning, where we do not have a candidate pattern. It only shows positive examples to the LLM.
- *improve node* already has a candidate pattern and can use it to select more relevant examples to show – *true positives* (to ensure the new patterns continue to match them), *false negatives* (to show which additional examples should be matched) and *false positives* (to show which examples the pattern currently matches but should not match)

Node workflow The nodes execute the following steps

1. select examples
   - *start node* selects randomly
   - *improve node* selects based on the initial pattern
2. create prompt
3. execute the conversation with the LLM
4. execute internal checks (see below)
5. calculate train metrics

Parts of the prompt Figure 2 shows an example. Definitions of all prompts are available in the code[3]

- Regex Cheatsheet[4] [optional]
- Task description
- Natural description of the matched entity [optional]
- Examples
- Think step by step [9] (encourage the model first to analyze the examples, write down common patterns, and only then create the regex)
- Instructions on how to mark the final regex so it is easy to extract it.

[3] The `prompt_templates.py` file contains all prompt definitions.

[4] `https://regexr.com` → Cheatsheet

```
Write a regular expression that matches all the following parts of
text wrapped in \` and \` as close as possible (but does not have to
be exact). The \` and `\ only highlight the desired extraction in the
examples, they do not appear in the original text and must not be
part of the final regex.

## Examples with added \` \` highlights:
- `12/10/2038`- Mastectomia partialis l. sin. et TAD axillae l
l.sin. Plánování radioterapie, CT simulátor `15.7.2045` v 12.00 hod.

## Only extractions:
12/10/2038
15.7.2045

First, write down common patterns in the extraction. Use them to
construct the regex. Mind the order of the patterns.
Make sure all parentheses match.
Write the final regex on the last line with a heading of FINAL REGEX: ...
Make sure it is on the same line as the heading and is not inside a code
block. Do not write anything else after that.
```

Fig. 2: An example of a *start* prompt with two training snippets (examples data have been altered to preserve privacy)

Formatting the examples The training snippets are shown to the LLM in two ways

- As *highlights with context*. For each example snippet, we create a context string containing it. We use 20–50 context characters, stopping at linebreaks. We use the backtick` character to highlight the snippet (the part that should be matched). This character is also used to highlight inline code blocks in the markdown-style format most LLMs use. An alternative could be to use XML tags (for example, `<match>`). Both of these options could suffer if the text of the examples already contains similar markings, so it might be beneficial to choose dynamically.

  In the prompt, it is also important to emphasize that the highlight markings are not part of the text and should not be used in the regular expression.

  This allows the LLM to use context clues, like positive lookaheads, in the regular expression pattern.
- As *extractions*. We also provide the contents of the snippets on their own in the prompt to improve pattern recognition inside the snippet.

The number of training snippets shown at the *start* and during *improvement* is configurable. We use 10 and 5, respectively, as the default.

Internal checks Both node types also come with two internal checks

1. *compile check* – checks if the generated pattern is a valid regular expression. If not, it continues the conversation with the LLM, shows it the compilation error and asks it to provide a fixed pattern.
2. *self validation* – continues the conversation with the LLM and asks it if the provided reasoning makes sense. If not, analyze the error and improve it. It has been shown that self-checking the generated response can improve performance [6].

Both of these checks are optional, and it can be configured how many of them to perform at most.

Node randomness To increase the likelihood that two initial *start nodes* or two *improve nodes* branching from the same pattern come up with different solutions and the branching is not redundant, we provide two ways of *disturbing* the LLM (changing the generated response). First, the sampling and order of examples shown to the LLM are based on a different seed in each node. Second, we can set the LLM sampling temperature to be greater than zero.

4 Experiments

4.1 Datasets

Anonymization in clinical notes The main evaluation dataset we used is a clinical note anonymization dataset [8]. It consists of 80 Czech clinical notes of cancer patients from Masaryk Memorial Cancer Institute. The notes were manually annotated with the label *anonymize*, which should contain all sensitive information. In total, there are 1031 annotations.

To obtain more granular labels, we have classified the extractions into several categories using regular expressions developed in [8] for automatic anonymization. We were left with 157 (15 %) unclassified entries, which we classified manually. Table 1 shows the distribution of labels.

We have chosen to evaluate our method on the *date* and *time* labels as they are suitable for regular expression extraction and have the most annotations.

Table 1: Distribution of labels in the Anonymization dataset [8].

label	date	time	name	hospital	place	job	phone	genetic	multi-date
count	621	146	121	52	40	19	17	8	7

Clinical entity extraction As part of the IDEA4RC[5] project, we are trying to automatically extract different medical entities from clinical notes of cancer

[5] Intelligent ecosystem to improve the governance, the sharing, and the re-use of health data for rare cancers `https://www.idea4rc.eu/`

patients, such as *biopsy type*, *mitotic activity count* or *invasion into fascia*. We are in the middle of an annotation campaign to gather example extractions of these entities. So far, we only have a few (less than 10) examples of each entity, so we cannot offer a full evaluation. Still, we can use this data for qualitative evaluation and discussion.

4.2 Comparison experiment

We compare our approach with RegexGenerator++ [1] as it tackles the same problem and the implementation is available and working. It uses genetic algorithms to find suitable regular expressions.

We create the *evaluation dataset* from the clinical note anonymization dataset (*date* and *time*) with the following steps For each label $l \in \{\texttt{date}, \texttt{time}\}$, for each split $s \in \{1..5\}$ sample half of all examples (notes) as a test set F^l_s. For each number of training examples (notes) $\in \{1, 2, 4, 8, 16, 24\}$ sample a training set $Tl_{s,d}$ from the remaining examples $E \setminus F^l_s$ such that $Tl_{s,d}$ contains d examples (notes) (if possible)

We end up with $2 \times 5 \times 6 = 60$ training sets corresponding to $2 \times 5 = 10$ test sets. The number of snippets (individual matches) in the training sets ranges from 1 to 224.

Configurations

- RegexGenerator++ The main parameters are jobs (j, how many simulations to run), population (p, how many individuals are in each simulation) and generations (g, how many generations each simulation runs). We use three configurations
  - `RG++ small` $j = 8$ $p = 100$ $g = 200$
  - `RG++ medium` $j = 8$ $p = 250$ $g = 500$
  - `RG++ large` $j = 8$ $p = 500$ $g = 1000$
- Our approach. We did not perform any hyperparameter optimizations prior to this test. We do not use `cheatsheet` or `natural_description`.
  The parameters we vary are: number of parallel `workers` w, `initial_splits` i, `max_depth` d, `branching_factor` b, and `time_limit` t (actual time it takes in parenthesis). We use the following parameters
  - `Our` $w = 4$ $i = 4$ $d = 3$ $b = 2$ $t = 60s$ $(80s)$

  As the LLM, we use `llama3.1:70b` available in Ollama with the default `Q4_0` quantization.

Experimental setup The clinical notes contain private information about the patients and doctors. For this reason, we are conducting all our experiments on the Czech CERIT-SC (SensitiveCloud) infrastructure.[6] We have access to an AMD EPYC 9454 CPU and an NVIDIA H100 GPU.

[6] `https://www.cerit-sc.cz/`

We run the RegexGenerator++ with 8 jobs on 8 threads. (more threads would not lead to a speedup as one job cannot run faster, and we already run all of them in parallel)

We use Ollama[7] on the H100 GPU to run the LLMs. We use 4 workers to run 4 conversations in parallel.

5 Results and Discussion

5.1 Metrics

We calculate many different metrics, but the most important are

- **Match F1 score** F1 score for exact matches. Averages precision and recall based on whole predictions. A prediction is classified as true positive only if it matches exactly with an annotation.
- **Character F1 score** F1 score for individual characters. Measures the F1 score as if we classified each character into binary classes. It is more granular than match F1 but may be slightly misleading (e.g. when the model predicts the entire text).
- **Mean F1 score** Arithmetic mean of match F1 and character F1. By default, it is used as the primary metric for selecting the best pattern (on the validation set) and calculating node priorities.

5.2 Comparison experiment

Figures 3 and 4 show the performance of our approach and RegexGenerator++[1]. We show two metrics (`match_f1` and `character_f1`) across two datasets (*time* and *date*). See Section 4.2 for details about the experiments.

Tables 2 and 3 show the means and standard deviations for the *time* dataset.

Table 2: Exact match F1 score ×100 for the *time* dataset. Columns show performance for different numbers of snippets (matches) in the training set. Aggregated from 5 runs.

training snippets	1	2-3	4-7	8-15	16-31	32-63	64-127
RG++ small	00 ± 00	09 ± 17	38 ± 33	76 ± 06	60 ± 22	49 ± 29	35 ± 32
RG++ medium	00 ± 00	33 ± 23	37 ± 32	75 ± 05	59 ± 17	64 ± 14	57 ± 03
RG++ large	00 ± 00	26 ± 18	37 ± 31	78 ± 07	62 ± 20	68 ± 09	60 ± 02
Our	$\mathbf{11} \pm 09$	$\mathbf{45} \pm 12$	$\mathbf{62} \pm 26$	$\mathbf{90} \pm 01$	$\mathbf{89} \pm 03$	$\mathbf{89} \pm 02$	$\mathbf{87} \pm 02$

We can see that our method outperforms RegexGenerator++, especially with low numbers of training snippets. It is also much more stable – the standard

[7] `https://ollama.com/`

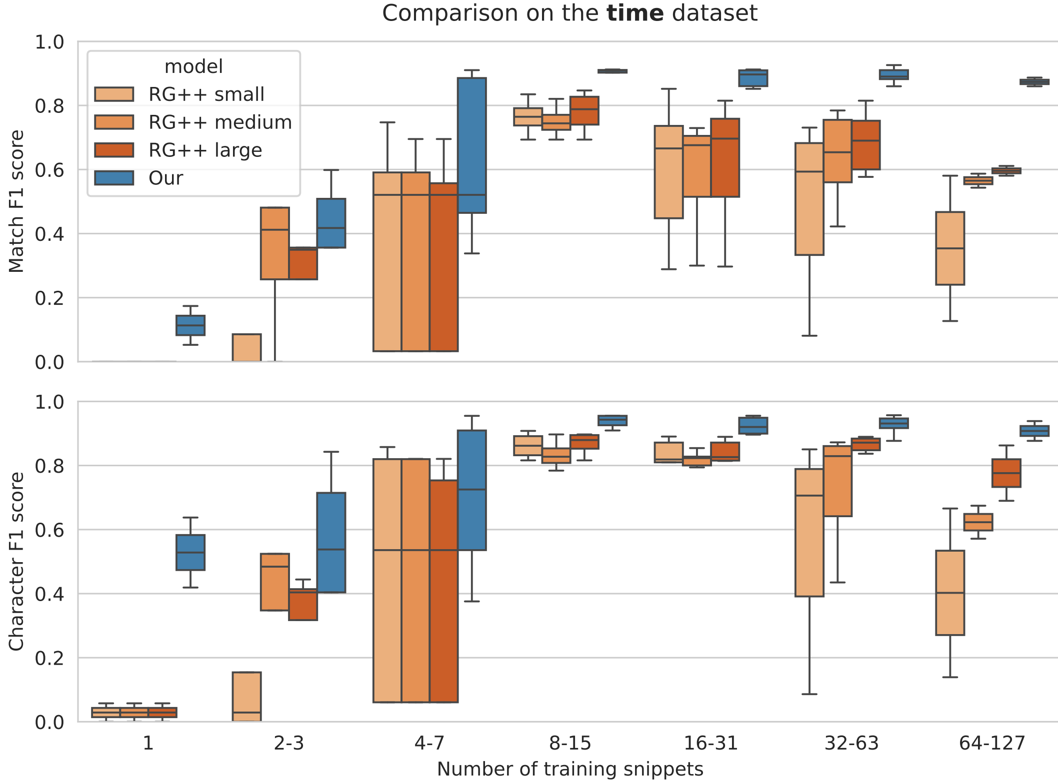

Fig. 3: Comparison of performance on the *time* dataset. The y axis shows two metrics – `match_f1` and `character_f1`. The x axis is stratified based on how many match snippets were in the training set. Aggregated from 5 runs.

Table 3: Character F1 score ×100 for the *time* dataset. Columns show performance for different numbers of snippets (matches) in the training set. Aggregated from 5 runs.

training snippets	1	2-3	4-7	8-15	16-31	32-63	64-127
RG++ small	03 ± 04	13 ± 21	47 ± 39	86 ± 04	79 ± 13	57 ± 34	40 ± 37
RG++ medium	03 ± 04	39 ± 22	46 ± 38	83 ± 05	78 ± 12	73 ± 18	62 ± 07
RG++ large	03 ± 04	33 ± 18	45 ± 37	87 ± 04	81 ± 11	84 ± 07	78 ± 12
Our	$\mathbf{53} \pm 15$	$\mathbf{58} \pm 22$	$\mathbf{70} \pm 25$	$\mathbf{94} \pm 02$	$\mathbf{92} \pm 03$	$\mathbf{93} \pm 03$	$\mathbf{91} \pm 04$

deviation is lower. The instability with fewer training snippets for both methods is caused by insufficient coverage of the training set.

Our method achieves reasonable performance with as few as 4–7 training snippets. If an exact match is not required and the approximate location of the extracted entity is sufficient, the method starts to work even with 1-3 training snippets.

Table 4 shows how long it takes to calculate the predictions. We can see that the speed of RegexGenerator++ is dependent on the number of training examples, as it is used to calculate fitness for each individual many times.

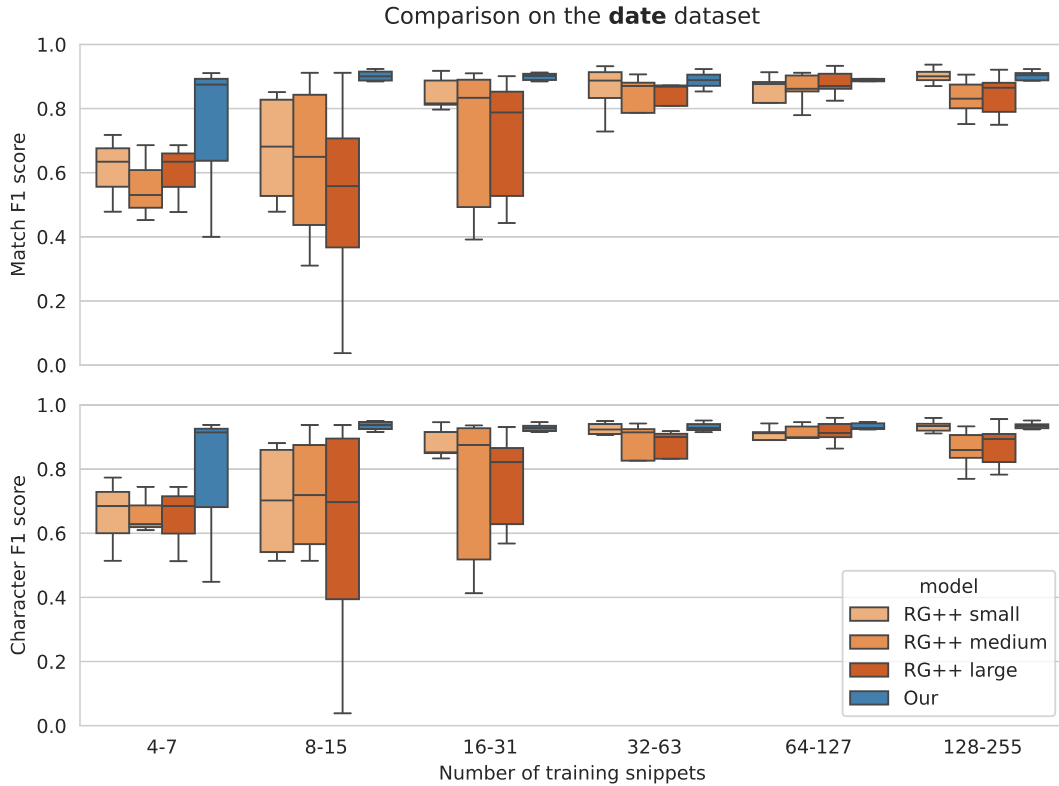

Fig. 4: Comparison of performance on the *date* dataset. The y axis shows two metrics – `match_f1` and `character_f1`. The x axis is stratified based on how many match snippets were in the training set. Aggregated from 5 runs.

Our method shows a constant number of snippets to the LLM in each node and spends almost all the time waiting for the LLM. In this configuration, the `time_limit` was set to 60 seconds. One LLM conversation with `llama3.1:70b` on the H100 usually takes 15–30 seconds. All running conversations are awaited and finished after the `time_limit` runs out, which explains the increased time. As we perform 4 conversations in parallel, each run manages to finish around 16 conversations.

Table 4: Average training time in minutes based on number of training snippets.

training snippets	1	2-3	4-7	8-15	16-31	32-63	64-127	128-255
RG++ small	0.1	0.1	0.2	0.4	1.0	2.3	2.4	1.6
RG++ medium	0.7	0.8	0.8	2.5	4.0	9.2	10.7	9.2
RG++ large	3.0	2.5	5.4	10.3	11.8	38.9	37.9	42.9
Our	1.4	1.2	1.2	1.3	1.3	1.3	1.3	1.4

5.3 Clinical entity extraction

We have access to prototype annotations of several entities in clinical notes that need to be extracted to facilitate federated large-scale data analysis in the IDEA4RC project[8]. As there are only a few annotations for each entity so far, there is not enough data to rigorously measure the performance. However, we offer a qualitative analysis of the behaviour of our approach on this data.

- *type of biopsy*, *invasion into fascia*, *type of necrosis*
  Our approach is able to join different formulations into a single pattern. It is also able to use positive/negative lookaheads.

  Pattern generated for *type of biopsy*:

  ```
  (punkční biopsii|Excidát|resekát|Exstirpát|Excize)|
  biopsie|Resekát|Resekce|excize|biospie(?= z| č\.| \d+)
  (?!biopsie|resekát)
  ```

  Pattern generated for *type of necrosis*:

  ```
  nekrotick[ý|é]|nekrotické|fokálně nekrotick[ý|é]|
  nekróz(y|ami)?|Bez (ložisek )?nekros
  ```

- *tumour size*
  Our approach successfully created a pattern for matching dimensions (e.g. `99x99x99 mm`). However, there are many different places where dimensions are used, and the desired extractions were only the primary tumour dimensions. More training examples would be needed to learn the context.

  Pattern generated for *tumour size*:

  ```
  \d+(?:\.\d+)?(?:×|x)\d+(?:\.\d+)? ?mm
  ```

- *mitotic activity count*
  Here, we have only 3 examples, each in a different format:
  `32 mf / 1,7 mm2     45/10 HPF     7-10/10HPF`
  The model matched two of them (the two `HPF`). The first (`mf/mm2`) was not matched only because the regex expects a decimal dot instead of a decimal comma.

  Pattern generated for *mitotic activity count*:

  ```
  \d+(?:-\d+\/\d+HPF|\s*\/\s*\d+\s*HPF|
  \s*mf\s*\/\s*[0-9]+(?:\.[0-9]+)?\s*mm2)
  ```

[8] Intelligent ecosystem to improve the governance, the sharing, and the re-use of health data for rare cancers https://www.idea4rc.eu/

6 Conclusion

We have introduced a new approach for generating regular expression patterns based on in-context examples using generative large language models (LLMs). We validate the effectiveness on date and time extractions from clinical notes and compare it with RegexGenerator++[1]. Our experiments show that our approach achieves better results with fewer examples and is more stable.

Finally, we conduct a qualitative study of performance on a small clinical entity extraction dataset. The results seem promising, as the model can merge different static patterns (without wildcards), use positive and negative lookaheads, and handle even more complex patterns with wildcards.

Future work We want to focus on generalizing the approach to be able to handle both positive and negative context-free examples and natural descriptions. Then, we want to compare it with other existing methods in these settings.

We also want to investigate the effects of different hyperparameters, like the LLM model, using the cheatsheet, including natural description, size of the search tree, number of snippets shown or the size of snippet context. For this, however, we will need a harder dataset so that it can properly differentiate the effects.

Finally, once we have access to full annotations from IDEA4RC, we want to do a proper evaluation on this data.

Acknowledgements. Supported by the project SALVAGE (OP JAK; reg. no. CZ.02.01.01/00/22_008/0004644) – co-funded by the European Union and by the State Budget of the Czech Republic.

Supported by grant MUNI/A/1590/2023: Using artificial intelligence techniques for data processing, complex analysis and visualization of large-scale data (AI-for-data)

Supported by the Technology Agency of the Czech Republic project TQ12000018.

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.

Disclosure of Interests. No competing interests to declare.

References

1. Bartoli, A., Lorenzo, A.D., Medvet, E., Tarlao, F.: Inference of regular expressions for text extraction from examples. IEEE Transactions on Knowledge and Data Engineering **28**(5), 1217–1230 (May 2016). https://doi.org/10.1109/TKDE.2016.2515587

2. Kushman, N., Barzilay, R.: Using semantic unification to generate regular expressions from natural language. In: Vanderwende, L., Daumé III, H., Kirchhoff, K. (eds.) Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies. pp. 826–836. Association for Computational Linguistics, Atlanta, Georgia (Jun 2013), https://aclanthology.org/N13-1103
3. Lee, M., So, S., Oh, H.: Synthesizing regular expressions from examples for introductory automata assignments. SIGPLAN Not. **52**(3), 70–80 (Oct 2016). https://doi.org/10.1145/3093335.2993244
4. Li, Y., Xu, Z., Cao, J., Chen, H., Ge, T., Cheung, S.C., Zhao, H.: Flashregex: Deducing anti-redos regexes from examples. In: 2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 659–671 (2020)
5. Locascio, N., Narasimhan, K., DeLeon, E., Kushman, N., Barzilay, R.: Neural generation of regular expressions from natural language with minimal domain knowledge. In: Su, J., Duh, K., Carreras, X. (eds.) Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing. pp. 1918–1923. ACL, Austin, Texas (Nov 2016). https://doi.org/10.18653/v1/D16-1197
6. Miao, N., Teh, Y.W., Rainforth, T.: Selfcheck: Using llms to zero-shot check their own step-by-step reasoning (2023), https://arxiv.org/abs/2308.00436
7. Raynal, M., Buob, M.O., Quénot, G.: fast: regular expression inference from positive examples using abstract syntax trees. In: Coste, F., Ouardi, F., Rabusseau, G. (eds.) Proceedings of 16th edition of the International Conference on Grammatical Inference. Proceedings of Machine Learning Research, vol. 217, pp. 96–116. PMLR (10–13 Jul 2023), https://proceedings.mlr.press/v217/raynal23a.html
8. Rusnačková, K.: Anonymisation of clinical notes (2024), https://is.muni.cz/th/rf3hj/
9. Wei, J., Wang, X., Schuurmans, D., Bosma, M., Ichter, B., Xia, F., Chi, E., Le, Q., Zhou, D.: Chain-of-thought prompting elicits reasoning in large language models (2023), https://arxiv.org/abs/2201.11903
10. Yao, S., Yu, D., Zhao, J., Shafran, I., Griffiths, T.L., Cao, Y., Narasimhan, K.: Tree of thoughts: Deliberate problem solving with large language models (2023), https://arxiv.org/abs/2305.10601
11. Ye, X., Chen, Q., Wang, X., Dillig, I., Durrett, G.: Sketch-driven regular expression generation from natural language and examples (2020), https://arxiv.org/abs/1908.05848
12. Zhong, Z., Guo, J., Yang, W., Peng, J., Xie, T., Lou, J.G., Liu, T., Zhang, D.: SemRegex: A semantics-based approach for generating regular expressions from natural language specifications. In: Riloff, E., Chiang, D., Hockenmaier, J., Tsujii, J. (eds.) Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing. pp. 1608–1618. ACL, Brussels, Belgium (Oct-Nov 2018). https://doi.org/10.18653/v1/D18-1189
13. Zhong, Z., Guo, J., Yang, W., Xie, T., Lou, J.G., Liu, T., Zhang, D.: Generating regular expressions from natural language specifications: Are we there yet? In: AAAI Workshops (2018), https://api.semanticscholar.org/CorpusID:20221107