LM-Powered: Bringing Neural Language Models to Your IDE

Hlib Babii
Free University of Bozen-Bolzano
Bolzano, Italy
hlibbabii@gmail.com

Moritz Griesser
Free University of Bozen-Bolzano
Bolzano, Italy
moritz.griesser@gmail.com

Andrea Janes
Free University of Bozen-Bolzano
Bolzano, Italy
ajanes@unibz.it

Romain Robbes
Free University of Bozen-Bolzano
Bolzano, Italy
rrobbes@unibz.it

ABSTRACT

We present **LM-Powered**, an extension to the Visual Studio Code IDE which uses the power and the versatility of Neural Language Models (NLM) to assist developers in their everyday routines on a number of tasks such as code completion, risky code visualization, natural language code search, and code folding. LM-Powered also provides features for researchers to debug and visualize the performance of NLMs. Thanks to its flexible architecture, the extension is not bound to a specific NLM and allows to seamlessly switch between NLMs under the hood. The video demonstration of LM-Powered is available at https://youtu.be/FnzzhJfZtIQ.

CCS CONCEPTS

• **Software and its engineering → Software maintenance tools**.

KEYWORDS

Naturalness of code, Neural Language Models, Software tools

ACM Reference Format:
Hlib Babii, Moritz Griesser, Andrea Janes, and Romain Robbes. 2022. LM-Powered: Bringing Neural Language Models to Your IDE. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The state of practice in IDEs is to use static code analysis to aid software development (e.g., style checking, code completion, detection of bugs and potential problems, etc.). Lately, a lot of research exploits the repetitive nature of code [9] to build machine learning models of source code [1].

As a result, a number of tools that facilitate software development have emerged. Cacheca [6] enhances code completion provided by the Eclipse IDE by utilizing an n-gram Language Model (LM) with a cache. More recently, TabNine [12] uses an Neural LM to provide code completion. Visual Studio IntelliCode [4] is another tool that provides AI-assisted context-aware code completion, as well as code style assistance. Hatari [18] identifies the riskiest locations in the code based on version control history. DeepCS [7] implements natural language code search by embedding a query and code snippets into high-dimensional vectors and finding the snippet which corresponds to the vector closest to the query vector. Tassal [5] uses a topic model to fold the code which it considers the least informative, in this way performing code summarization.

All these tools concentrate on a single task. In this demo, we show that a Neural Language Model (NLM) can support a variety of software engineering tasks. An LM is trained on a large corpus of text, and defines a probability distribution of a sequence of tokens. LMs can be used in two ways: 1) to predict the most likely next tokens, given a context, and 2) to measure the probability of a sequence of tokens. NLMs are trained in an unsupervised fashion by repeatedly predicting the next token in their training corpus. Research has shown that this training objective allow NLMs to be versatile: they can be used with little [11, 16] to no adaptation [14] to perform a number of different tasks.

We present **LM-Powered**, a tool implemented as an extension for the Visual Studio Code (VSC) IDE. LM-Powered uses an NLM to provide features such as code completion, risky code visualization, code search, and code folding. We discuss the features of LM-Powered (also for LM researchers and developers) in detail in Section 2. An important question in this context is also how to deploy and use LMs in production; often, LMs are not moved anywhere beyond the researcher's workstation. We describe the approach used in LM-Powered in Section 3. NLMs have a reputation of having high resource requirements (i.e., long training time, GPU acceleration). However, this mostly applies to training, which needs to be done only once and in advance. Inference, on the other hand, can be often done in real time without the use of a GPU. We discuss performance aspects in Section 4.

The script to install LM-Powered and the source code can be downloaded from https://github.com/giganticode/lm-powered. The demo video is available at https://youtu.be/FnzzhJfZtIQ.

2 FEATURES

In this section we describe the features of LM-Powered in detail and illustrate them in Figure 1. We will refer to the different fragments of this figure in the following subsections. We note that our focus is to showcase the versatility of NLMs, rather than obtaining the best possible results for each task. State of the art results can be

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

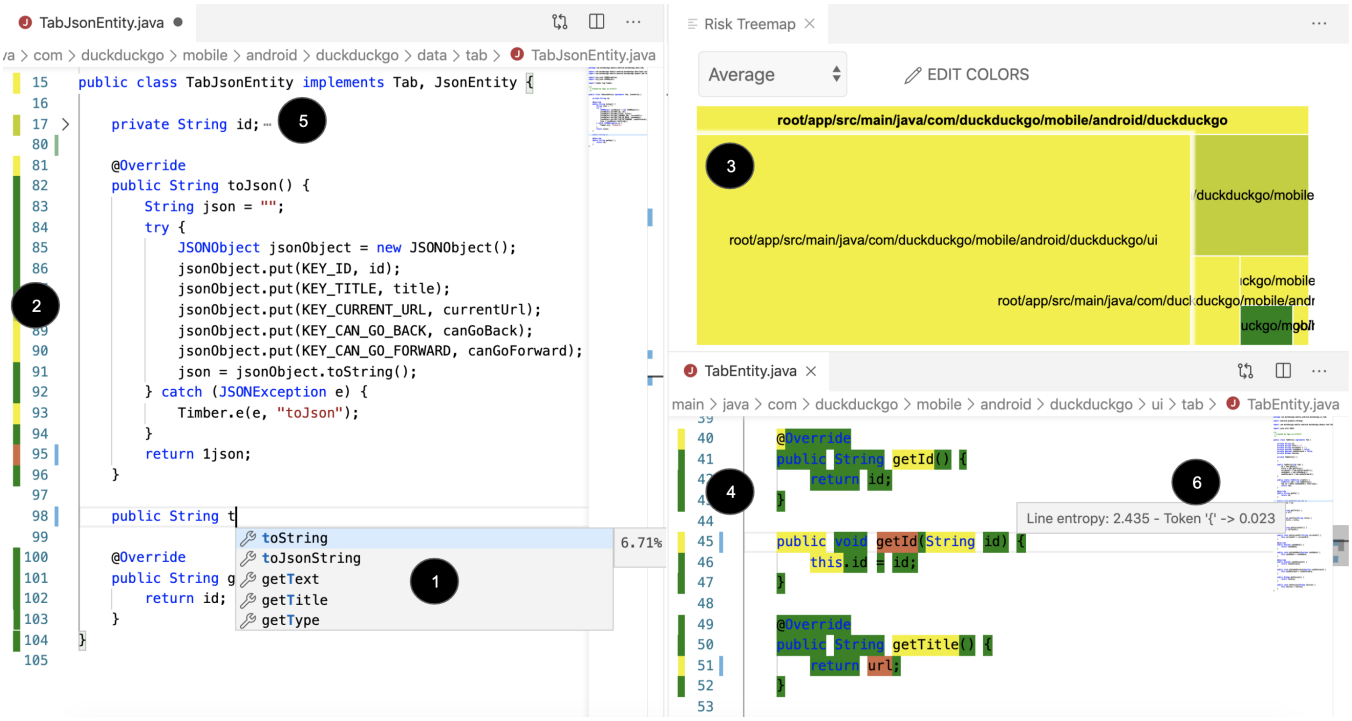

Figure 1: Overview of LM-Powered features

achieved by fine-tuning the NLM to a specific task, as our work on code completion shows [13], but exceeds the scope of this paper.

2.1 Code completion

The advantage of NLM-based code completion compared to static analysis is that NLMs are not limited to predicting information from the local scope: they can also use knowledge extracted from the thousands of projects they have been trained on. What makes code completion with NLMs even more appealing is that the prediction of the next token is an ability that they naturally possess. Therefore, the implementation of code completion is as simple as querying the next token from the model. To be more precise: sometimes we have to query not a single but multiple sub-tokens. The reason for this is that our NLMs operate on the sub-token level. For sub-token level models, the problem of predicting the next token becomes a problem of predicting multiple sub-tokens. We use beam search as a common approach to tackle this task [13].

LM-Powered queries the next token on the user's request (via keyboard shortcuts) or when some event occurs, e.g. '.' is entered. In Figure 1, Fragment 1 shows LM-Powered proposing a *new* method name. Notice how the proposals include generic names (`toString`), and others tailored to the context (`toJsonString`).

2.2 Risky code visualization

There are many situations when a developer wants to see the code sections that are likely to contain problems, for example, while doing code reviews, when working on technical debt reduction, etc. Moreover, getting this information in real-time at almost no price during actual code writing can be helpful as well.

LMs assign probabilities to sequences of tokens. Good LMs assign high probabilities to common code, and low ones to rarely seen code [15]. We use this property to spot suspicious code. We 'ask' the LM to estimate the probability of every line of code and based on that mark each line with a color shade between red and green (yellow being somewhere in between). The color is on the green side if the line is assigned a high probability, meaning it is of low risk. If a line gets a low probability value, it is colored with a shade of red.

A seen in Fragment 2, most of the code is marked with shades of green. There are some yellow lines e.g. `Timber.e(e, "toJson");`, which is according to the model not a common way to handle an exception. A typo `1json` in line 65 does not remain unnoticed and is colored in a shade of red.

The evaluation is done for the whole project after it is loaded into the IDE. Once the user changes a file, this file is reevaluated.

Treemap. It is possible to see the locations of suspicious code not only on the file, but also on the project level using the treemap visualization feature (see Fragment 3). It uses colors in the same way as explained above: files and folders are colored in red if they are considered to be on average risky, green otherwise.

Token-level visualization. If, on the contrary, a lower-level view is needed, LM-Powered can visualize risk levels for individual tokens as can be seen in Fragment 4. This can be useful if it is not clear to the developer which tokens in the line raise concerns. Notice how LM-Powered highlights the `getId`, that returns `void`, and `getTitle` that appears to return the wrong attribute (`url`).

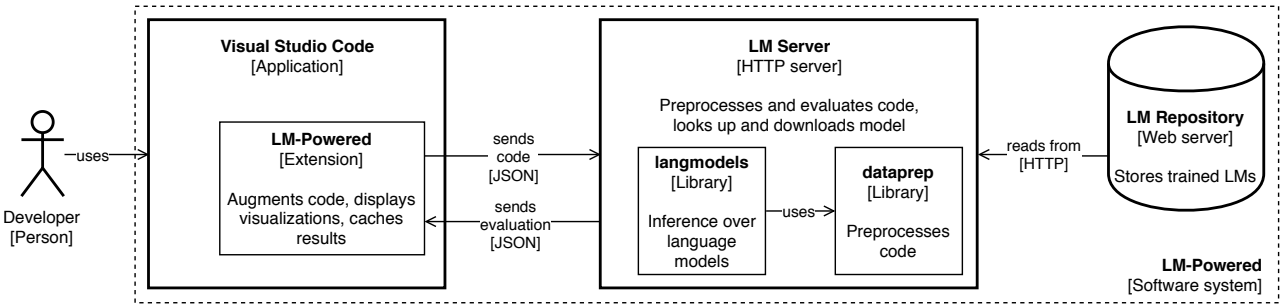

Figure 2: Architecture of LM-Powered

2.3 Natural language code search

Natural language code search allows looking for code snippets in the project that most closely correspond to a user defined query. The idea behind its implementation is to split the project into short code snippets and feed them one by one to the LM. After every snippet, we query the LM for the estimated probability of seeing the query in a form of a comment, given the code snippet as a context. For example, suppose that we are evaluating the relevance of a piece of UI code, when the query is about the database concerns; the NLM would estimate that the probability of the query, given this code snippet, is low. Those locations where the probability is above a certain threshold are considered a match.

2.4 Code folding

Code folding is the replacement of multiple lines of code with a single line. This line can contain a special token and/or the summarization of the folded code with the purpose of hiding 'uninteresting' pieces of code. It is common for IDEs to use this feature to hide boilerplate code such as getters/setter, and can be useful to find 'interesting' code faster, for example, if the file is large.

LM-Powered currently uses folding to hide the code that is not considered risky by the LM (see Fragment 5 where lines with class member fields are collapsed). Another feature (still under development) is a better visualization of the results of a search by folding all the code the query was not matched against.

2.5 Features for LM researchers

Since language modeling is a central concept on top of which LM-Powered is built, the availability of a toolkit for LM debugging and visualization is quite natural. While the target audience of the extension is developers, LM-Powered contains a few features LM researchers can also benefit from.

Token-level debugging. To debug and improve the performance of an LM in the per-token risk visualization mode, the user can see how the LM performs on each token separately. In addition to color visualization, the cross-entropy values can be seen when hovering over the token of interest: see Fragment 6.

Switching between language models. The extension itself is decoupled from the specific LM being used. The LM researcher can easily change the LM currently being used, e.g. to a better performing one, to see the improvement of predictions, search, etc.

LM comparison. We have developed a stand-alone visualization that compares the performance of two LMs; we are currently integrating it in LM-Powered. The visualization also highlights tokens or lines with colors. A token or a line is green if the tested LM performs better than the baseline LM, and red if vice versa. The intensity of the color indicates the magnitude of the difference.

3 ARCHITECTURE

Figure 2 shows the three major components of LM-Powered: the extension itself, the Language Model Server (LM Server), and the Language Model Repository (LM Repository). We depict the architecture in the form of a container diagram according to the C4 model notation [3] also showing important components and libraries inside the containers.

The extension runs inside VSC and interacts with the user. The LM server processes requests from the extension and does most of the computations including inference on LMs. The LM repository stores trained LMs. All these components run as separate processes and communicate with each other via HTTP. In a production scenario, the LM Server can run in the cloud (possibly with the use of GPU if there is a need for exceptional performance). However, it can also run locally either on a GPU, or on a CPU.

When the extension is started, the LM Server loads an LM into memory. It first looks up the LM in its disk cache on the disk and, if absent, downloads it from the LM repository. For each client, an LM Server has to manage a separate LM since LMs are stateful and their state depends on the current context (the project loaded, the file opened, the line being edited at the moment).

When a project is loaded in VSC, LM-Powered sends a request to the LM Server to evaluate the project's code. For different tasks, the payload of the request varies. For search, the whole project is uploaded, whereas for code completion only a certain number of tokens before the cursor are sent within the request.

Apart from the HTTP server, which processes requests from the extension, the LM Server relies on the *dataprep* and *langmodels* libraries. The *dataprep* library preprocesses the code before it is used by the LM. The library makes sure that the code is preprocessed in the same way it was for training, ensuring that the LM will be able to 'understand' it correctly. The *langmodels* library is responsible for inference and returning information required by the tasks.

Once the extension receives the results from the server, it creates the visualization. Requests related to different tasks are continuously sent to the LM Server. Some are sent within some time

Table 1: Performance of the model with 27.7 M parameters.

Disk space	350 MB
RAM	108 MB
Risk calculation speed	56 LOC/s
Code completion time	340 ms

interval, some are sent after a file has been changed or after the user triggered it implicitly by a key combination.

The extension does not depend on the architecture of the underlying LM, on the way how it was trained, etc. The architecture allows to plug-in new LMs and switch between LMs. Once a better model is trained and uploaded to the LM Repository, the extension can seamlessly switch to it.

Several performance-related architectural decisions were made. For risky code visualization, the results of the queries are cached and saved on disk. The *langmodels* library does not need to be invoked unless the code changes, even after the IDE restarts. Also, many requests are done ahead of time in the background. For example, for the risky code visualization task, all the files in the project are evaluated as soon as the project is loaded. Another optimization exists for code completion: when the user changes the location of the cursor, the context before the cursor is sent to the LM Server ahead of time to suggest code completions with short delays. In situations where speed is much more important than the accuracy of the results, the context or the beam size that is used for code completion can be reduced.

4 PERFORMANCE

There are two important aspects of the performance of LM-Powered: the accuracy of the results and the responsiveness. The former aspect entirely depends on the accuracy of the underlying LM. Therefore it can be evaluated separately, outside of the context of extension. We refer the reader to our previous work [13] where we extensively evaluate our NLMs. The default NLM used in LM-Powered is an LSTM trained on more than 10k projects [2], pre-processed with BPE [17] with 10k merges. The model uses embedding vectors with 1024 dimensions, and 3 hidden layers of 1024 hidden units.

Notwithstanding the excellent predictive performance of the engine, the extension might be of little use if it could not handle user requests reasonably fast. For this reason, the runtime characteristics of the extension are crucial. We show that even using our largest model, that contains 27.7M trainable parameters, LM-Powered does not slow down the IDE. The evaluation is done on a consumer-grade laptop with 2.3 GHz Intel Core i5. Note that for the evaluation the LM Server was run locally, and a CPU was used for inference, to show the worst-case scenario (a GPU being 5-10X faster).

The evaluation summary can be seen in table 1. One instance of the model consumes 108MB of RAM; multiple models can coexist if necessary. The code completion request is performed on average in 340ms, which is an acceptable delay for an interactive request. For risky code visualization in one second probabilities for 56 lines of code are calculated. This is acceptable if requests are done in the background and cached. If the user chose to run LM Server locally, the model would require 350MB of storage on their computer.

There are approaches to make models smaller and faster while keeping their performance at the similar levels than larger models, such as distilling a large model in a smaller one [10], quantization of floating points weights to discrete integers, and pruning of small weights [8]. We plan to investigate these approaches in future work.

5 CONCLUSIONS

We show that a trained-once NLM can be used to solve several tasks, such as risky code visualization, code completion, natural language code search, and code folding. We implement these tasks in LM-Powered, an extension for Visual Studio Code. We believe that LM-Powered can increase developer's productivity without being a significant burden on the IDE. Code completion can be done in a fraction of a second. For risk visualization, the pre-processing of a file with 100 LOC takes less than two seconds. The extension itself is decoupled from the LM being used which allows plugging in at runtime a new, better-performing LM, once one has been trained. With developers being a primary target audience of the extension, LM researchers can also benefit from features that simplify the debugging of LMs and visualization of their performance.

ACKNOWLEDGEMENTS

We thank Ivica Stanimirovic for his help with creating the video demonstration and giving valuable feedback.

REFERENCES

[1] Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. 2018. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)* 51, 4 (2018).
[2] Miltiadis Allamanis and Charles Sutton. 2013. Mining source code repositories at massive scale using language modeling. In *Proceedings of MSR 2013.*
[3] Simon Brown. 2015. *The Art of Visualising Software Architecture.* leanpub.com.
[4] Microsoft Corporation. 2019. IntelliCode. https://docs.microsoft.com/en-gb/visualstudio/intellicode
[5] Jaroslav Fowkes, Pankajan Chanthirasegaran, Razvan Ranca, Miltiadis Allamanis, Mirella Lapata, and Charles Sutton. 2016. TASSAL: Autofolding for source code summarization. In *Proceedings of ICSE 2016.*
[6] Christine Franks, Zhaopeng Tu, Premkumar Devanbu, and Vincent Hellendoorn. 2015. Cacheca: A cache language model based code suggestion tool. In *Proceedings of ICSE 2015-Volume 2.*
[7] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018. Deep code search. In *Proceedings of ICSE 2018.*
[8] Song Han, Huizi Mao, and William J Dally. 2015. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149* (2015).
[9] Abram Hindle, Earl T Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. 2012. On the naturalness of software. In *Proceedings of ICSE 2012.*
[10] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. 2015. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531* (2015).
[11] Jeremy Howard and Sebastian Ruder. 2018. Universal language model fine-tuning for text classification. *arXiv preprint arXiv:1801.06146* (2018).
[12] TabNine Inc. 2019. TabNine. https://tabnine.com
[13] Rafael Karampatsis, Hlib Babii, Andrea Janes, Charles Sutton, and Romain Robbes. 2020. Big Code != Big Vocabulary: Open-Vocabulary Models for Source Code. In *Proceedings of ICSE 2020.* in press.
[14] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language models are unsupervised multitask learners. *OpenAI Blog* 1, 8 (2019).
[15] Baishakhi Ray, Vincent Hellendoorn, Saheel Godhane, Zhaopeng Tu, Alberto Bacchelli, and Premkumar Devanbu. 2016. On the" naturalness" of buggy code. In *Proceedings of ICSE 2016.*
[16] Romain Robbes and Andrea Janes. 2019. Leveraging small software engineering data sets with pre-trained neural networks. In *Proceedings of ICSE (NIER) 2019.*
[17] Rico Sennrich, Barry Haddow, and Alexandra Birch. 2015. Neural machine translation of rare words with subword units. In *Proceedings of ACL 2016.*
[18] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. 2005. Hatari: raising risk awareness. In *ACM SIGSOFT Software Engineering Notes*, Vol. 30.