

A Visual Studio Code Plugin for Experimenting Code Completion Models

Matteo Omenetti

Abstract

Code completion is one of the main features of modern Integrated Development Environments (IDEs). Its aim is to assist and facilitate the developer in code writing, by predicting the next code token(s) the developer is likely to write, therefore speeding up the development process. Currently, there are no tools that can be easily adapted to support different prediction models, therefore allowing researchers and practitioners to intuitively and easily experiment with different approaches. In this work, we take a step further in this direction by presenting a newly developed plugin for on the most popular IDEs currently on the market, Visual Studio Code (VSCode). The plugin presented in this work seamlessly integrates with the user interface of VSCode by showing the predictions directly in the text editor in a light grey color. The developer can then accept or reject the predictions coming from the model in different, but all very intuitive ways. For accepting, it is sufficient to manually write the suggestion or use a keyboard shortcut. For rejecting, writing something else will make the suggestion gently fade away. By supporting the simple API of the plugin, researchers can experiment with their own prediction models. A further endpoint for feedback storing can be integrated, this allows researchers to collect new data points directly from the developers that on a daily basis accept and reject hundreds of suggestions coming from the model.

Advisor

Prof. Gabriele Bavota

Assistants

Dr. Emad Aghajani, Dr. Antonio Mastropaolo

Advisor's approval (Prof. Gabriele Bavota):

Date:

Contents

1	Introduction	2
2	Related Work	3
2.1	Code Completion Approaches	3
2.2	Comment Completion Approaches	4
2.2.1	Code Summarization	4
2.2.2	Mining Crowd Documentation	4
2.2.3	Supervised Learning Techniques	4
2.3	Current Code Completion Tools & UI	5
2.4	Summing Up	6
3	S.I.M.O.N.	6
3.1	The (default) Models Used by the Plugin	6
3.2	Design	7
3.2.1	Visual Studio Code Plugin	7
3.2.1.1	Parse the Java File	7
3.2.1.2	Capture User Input	7
3.2.1.3	Show Recommendations	8
3.2.1.4	Keyboard shortcuts	8
3.2.1.5	Settings	8
3.2.2	Code/Comment Completion Web Service	9
3.2.3	Feedback Web Service	9
3.3	Usage Scenarios	10
4	Conclusions and Future Work	11
4.1	Future Work	12

1 Introduction

Today's world is evolving faster than has ever done before. Computer Science is trying to keep up with the constant need of always larger and more complex software, this makes programming languages, libraries and frameworks to get constantly updated with new features. As software developers, we are often required to rapidly and efficiently switch between different programming languages and sometimes it might be challenging to keep up with this continuous evolution. As a result, code completion was invented and has now become a fundamental feature of modern IDEs. This feature can provide recommendations about the next code token(s) that the developer is likely to write given the code already written, therefore improving the speed and accuracy of software development. The invention of code completion dates back to 1996, when Microsoft first introduced it to the wide public in Visual Basic 5.0 Control Creation Edition. Over the decades, this feature went from merely suggesting alphabetical lists of the next token to write, to "intelligently" taking into consideration the context surrounding the code and typical code patterns, therefore drastically improving the performance of code prediction. In recent years, with the rapid development of artificial intelligence, a new way of performing code completion has started taking place. This new way uses the most recent and advanced discoveries in neural networks to set up new standards in code completion performance. Only a few recent studies apply the concept of neural networks to the prediction of multiple contiguous tokens [1] [2]. To solve this complex problem, different models have been proposed, each one with its own strengths and weaknesses.

Currently, there are no tools that can be easily adapted to support different prediction models, therefore allowing researchers and practitioners to intuitively and easily experiment with different approaches. In this work we present a tool to overcome this limitation, by presenting a plugin for Visual Studio Code (VSCode), one of the most popular IDE currently on the market. For the first time, this seamlessly integrated tool named S.I.M.O.N. (Software Institute Matteo Omenetti Neural Network), allows experimenters to test their own Java code prediction neural networks in real world scenarios. The functioning of this plugin is very intuitive for the final user. The proposed neural network must be uploaded on a server that respects the simple API of the plugin. In the plugin settings, two addresses can be specified, one for the code completion neural network and one for the comment completion neural network. Once an address has been specified, it is sufficient to normally start programming and recommendations will be displayed in a non-intrusive light grey color directly in the text-editor. At this point, if the recommendation is not relevant to the context, it is sufficient to keep writing to make the suggestion gently fade away. If, on the other hand, the suggestion is relevant to the context, the user can manually write it down or accept it with a keyboard shortcut. The plugin will take care of the business of parsing the java file, understanding the position of the caret (code or comment), triggering recommendations in the correct situations and understanding when recommendations are not needed anymore.

Programming is a huge discipline and it can be carried out in different ways. Sometimes plain java code is sufficient to solve a problem, but most of the times the use of external libraries and frameworks is required. Solving such a complex task as token(s) prediction requires lots of data, since a neural network, in order to be effective, must be trained in the largest amount of scenarios. Collecting huge amount of data that spans the greatest amount of use cases is one of the most challenging aspects of building such a system. For this reason, there is a feedback storing feature integrated in the plugin. An additional address can be specified in the settings. This address must lead to a server that respects the simple API of the plugin. Whenever a user accepts or rejects a suggestion, a feedback will be sent to the server. By doing so, it is possible to collect several additional data points in order to further train the network and therefore enhance its capabilities.

This plugin is currently available in the Visual Studio Plugin Marketplace and it can be downloaded for free either by typing "S.I.M.O.N." in the VSCode extension panel or at the following link <https://marketplace.visualstudio.com/items?itemName=Matteo-Omenetti.simon>. In order to experience the beauty of intelligent code prediction, there is no need to build your own neural network. By default the system uses an implementation of the T5 model [1] [2], specifically tuned and trained by the USI Software Institute to perform token(s) prediction. T5 models represent a viable solution for code completion, outperforming even the powerful RoBERTa models, with perfect predictions ranging from ~30%, obtained when asking the model to guess entire blocks, up to ~69%, reached in the simpler scenario of few tokens masked from the same code statement.

2 Related Work

In this section an overview of the literature related to code and comment completion techniques is given. Furthermore, the approaches aimed at automating code writing are highlighted. Lastly the most recent approaches to show model predictions are presented and compared to the approach presented in this work.

N.B. This section has been written by taking inspiration from the Related Work sections of the following papers, shared with me by the respective authors (these papers are currently under review at TSE):

- Matteo Ciniselli*, Nathan Cooper†, Luca Pascarella*, Denys Poshyvanyk‡, Massimiliano Di Penta‡, Gabriele Bavota* *“An Empirical Study on the Usage of BERT Models for Code Completion”*
*SEART @ Software Institute, Università della Svizzera italiana (USI), Switzerland
†SEMERU @ Computer Science Department, William and Mary, USA
‡Department of Engineering, University of Sannio, Italy
- A. Mastropaolo, S. Scalabrino, N. Cooper, D. Palacio, D. Poshyvanyk, R. Oliveto, and G. Bavota, *“Studying the usage of text-to-text transfer transformer to support code-related tasks,”* in 43rd International Conference on Software Engineering (ICSE 2021), 2021, p. <https://arxiv.org/pdf/2102.02017.pdf>.

The Related Work sections of these papers have been summarized and paraphrased to the best of my abilities.

2.1 Code Completion Approaches

Computer Science literature is rich in code completion studies. Throughout the years many tools and approaches to perform an always more accurate code completion have been studied and some of these tools are currently in use in the most popular IDEs. In this section, the most popular ones are presented in chronological order.

The Prospector tool by Mandelin et al. [14] paved the way for all the following studies in the research field of code completion. This first basic tool was able to suggest variables and method calls from the user’s code base. This tool was then improved by other independent studies such as the InSynth tool by Gvero et al. [9], which took into consideration also the type of the expression needed at the given point.

Bruch et al. [8] introduced an intelligent code completion system that learns from existing code repositories, and it is capable of filtering out from the list of method calls recommended by the IDE those that are more relevant to the current working context. The results of this work show an 82% accuracy in predicting method class needed by the user and 72% accuracy in predicting method calls that are relevant to the current development context. This approach was then improved by Proksch et al. [16], by adding further contextual information and by proposing a Pattern-based Bayesian Networks approach. This approach showed that the additional context information collected improves prediction quality, especially for queries that do not contain method calls.

Robbes and Lanza [3] used information extracted from the change history of software systems to support the code completion of method calls and class names.

Asaduzzaman et al. [5] proposed a technique named CSCC (Context Sensitive Code Completion). By collecting examples from online software repositories, this approach is based on the assumption that similar contexts require similar method calls. In particular, each method call is assigned a context made up of methods, keywords, class and interface names appearing within four lines of code. This approach outperforms all the tools discussed so far by achieving 86% precision and 99% recall.

Hindle et al. [11] pioneered the work on statistical language models. This approach is based on the naturalness of software, that just like natural language, it is also likely to be repetitive and predictable. This work paved the way for similar work aimed at solving the problem of code completion through the use of statistical models. One fundamental work in this field is represented by Raychev et al. [17]. The main idea behind this work is to reduce the problem of code completion to a natural-language processing problem of predicting probabilities of sentences. In particular, this method extracts sequences of method calls from a large code base, and uses this dataset to train a language model able to predict API calls. Their model achieves a 90% accuracy in the top-3 recommendations.

Nguyen et al. [15] proposed GraPacc, a context-sensitive code completion model trained on a database of API usage patterns. These patterns are then matched to a given code under development to support code completion. GraPacc achieves up to 95% precision and 92% recall.

Tu et al. [57] introduced a cache component to exploit the “localness of code” in the n-gram model. Given the local repetitiveness of code, localized information can be used to improve performance. Data shows that this new model outperforms standard n-gram completion by up to 45% in accuracy. The proposed model’s suggestion accuracy is comparable to a state-of-the-art, semantically augmented language model; but it is simpler and easier to implement. This model requires nothing beyond lexicalization, and thus is applicable to all programming languages.

Hellendoorn and Devanbu [10] proposed further improvements to the cached models by also considering specific characteristics of code (e.g., unlimited, nested and scoped vocabulary). They showed that their model can be combined together with DL-model, leading to an unprecedented 1.25 bits of entropy per token.

Karampatsis et al. [12], finally proposed neural networks as the best models that can be used for code completion tasks. The proposed neural network was not only able to dynamically adapt to different projects, but also outperforms the best n-gram models, achieving an entropy of 1.03 bits.

Kim et al. [13] leveraged the Transformers neural network architecture for code completion. They provide the syntactic structure of code to the network by using information from the Abstract Syntax Tree to fortify the self-attention mechanism. The best model they were able to find reached a MRR up to 74.1% in predicting the next token.

Alon et al. [4] proposed a language agnostic approach named Structural Language Model. Given a partial AST, the model, based on LSTMs and Transformers, reached state-of-the-art performance with an exact match accuracy for the top prediction of 18.04%.

Svyatkovskiy et al. [55] introduced IntelliCode Compose. This tool is able to predict code sequences of arbitrary token types, independently of the programming language. Their model can recommend an entire statement, and achieves a perplexity of 1.82 for the Python programming language. Svyatkovskiy et al. [56] also proposed a different approach on code completion with neural networks. This new approach proposed shifting from a generative task to a learning-to-rank task. Their model is used to rerank the recommendations provided via static analysis, being cheaper in terms of memory footprint than generative models. To this aim, Avishkar et al. [7] proposed a neural language model for code suggestion in Python, aiming to capture long-range relationships among identifiers exploiting a sparse pointer network.

A considerable step forward, has been taken recently by Aye and Kaiser [6]. They proposed a novel language model capable of predicting the next top-k tokens while taking into consideration some real-world constraints such as (i) prediction latency, (ii) size of the model and its memory footprint, and (iii) validity of suggestions.

Finally, while the tool proposed in this work can be used with any model, it has been developed from the ground-up using the model proposed in [2]. This model, for the first time, attempts the automatic generation of entire code blocks.

2.2 Comment Completion Approaches

In order to generate comments three are the main techniques that can be used: Code Summarization, Mining Crowd Knowledge and Supervised Learning Techniques. In the following sections these techniques are briefly discussed, while more emphasis is put on the latter, given that is the approach used by the default model of the project.

2.2.1 Code Summarization

Code Summarization techniques can be used in three different ways (i) creating bag of words that represent the main responsibilities of the code [18], [19]-[21]; (ii) hiding lines that are not considered fundamental in understanding the aim of the code [22]; and (iii) describing code in natural language as a human would do [23], [24]. While the first two approaches are "extractive", since they just create a summary of the code by extracting the most important features from the code itself, the latter is an abstractive approach, since it generates new information that may not be already included in the documentation.

2.2.2 Mining Crowd Documentation

Mining Crowd Documentation is an approach that relies on mining and processing online project (written by humans) to aid program comprehension. These approaches have been used to recommend API usage examples [25]-[42] and relevant pieces of information in API reference documentation [33]. In particular, Rahman et al. [34] implemented CodeInsight a tool that can suggest improvements to a given snippet of code, using code samples taken from StackOverflow. Wong et al. [43], [45] uses code comments taken from existing projects to automatically generate comments for similar snippets of code. Similarly, Aghajani et al. [44] presented ADANA, a tool trained from StackOverflow and GitHub projects, to automatically inject code comments in Android projects.

2.2.3 Supervised Learning Techniques

In this section, the main works exploiting machine learning techniques to summarize and document code are presented.

Nazar et al. [46] recruited four students to extract, from a corpus of 127 code snippets, the lines of code needed to summarize each snippet. Then, ten developers inspected the selected lines and they extracted a total of 21 features that have been used to train a SVM classifier able to achieve a precision of 82% in selecting relevant code lines for a given code snippet.

Ying and Robilliard [47] used a supervised learning approach to summarize code examples. The supervised algorithm exploits features extracted from the statements composing the code examples. By comparing their model to manually written comments, they were able to achieve a 71% precision.

Iyer et al. [50] presented CODE-NN, a neural network model using LSTM to produce natural language summaries describing C# code snippets and SQL queries. Zheng et al. [48] proposed Code Attention, an attention module to translate code to comments.

Liang et al. [54] presented a Recursive Neural Network named Code-RNN to describe the structural information of source code and produce natural language comments. Their Code-RNN takes advantage of a novel GRU cell (Code-GRU) designed for code comments generation. Hu et al. [49] used a Deep Neural Network (DNN) to automatically generate comments for a given Java method. To train the DNN, the authors mined $\sim 9k$ Java projects hosted on GitHub by collecting pairs of method, comment, where "comment" is the first sentence of the Javadoc linked to the method.

LeClair et al. [53] presented a neural model combining the AST source code structure and words from code to generate coherent summaries of Java methods. The approach, tested on 2.1M methods, showed its superiority as compared to the previous works by Hu et al. [49] and Iyer et al. [50].

Finally, in a recent work, Mastropaolo et al. [51] showed that a T5 Model [52] properly pre-trained and fine-tuned can achieve better performance than the techniques presented in this section, generating comments "as humans would do" in $\sim 10\%$ of cases. This is the default model used by the tool presented in this work.

2.3 Current Code Completion Tools & UI

Every major IDE supports some sort of code completion feature. Some have very basic features, in which only functions available in the current scope are offered, sorted in decreasing number of usages or in alphabetical order (Atom). Some instead, have more sophisticated methods, for example Visual Studio Code with its IntelliSense framework. Every IDE has its own proprietary way of performing code completion and usually companies don't like to disclose their secret algorithms. A likely general approach used to perform this so called "intelligent" code completion could be the following:

1. An Abstract Syntax Tree (AST) is built from the project source code. This tree is usually built using the same techniques and algorithms used by compilers, and IDE doesn't need to generate or optimize code, but in order to provide code completion the rest must be there (i.e. lexing, parsing, semantic analysis, type inference, type checking, macro expansion, etc...). However, in this case we are usually trying to build a tree out of incomplete expressions, therefore the algorithms used in this scenario will have to take into account this possibility.
2. When a character is typed, it walks down a path in the AST. All of the descendants of a particular AST node are possible completions. The IDE then just needs to filter those out by the ones that make sense in the current context, but it only needs to compute as many as can be displayed in the tab-completion pop-up window.
3. When an expression is fully typed, the AST is partially recomputed to further enhance the "guessing" capabilities of the method.

Methods based on the above approach are already very powerful and enable software developers to speed up the coding process. On the other hand, these methods are also very basics since they are just based on some rules and heuristics and their limitations become visible when trying to predict multiple contiguous tokens. For this reason, only one token is suggested most of the times. Since multiple predictions are returned and their length is usually short, a pop-up window is a convenient choice to show recommendations to the user. 1

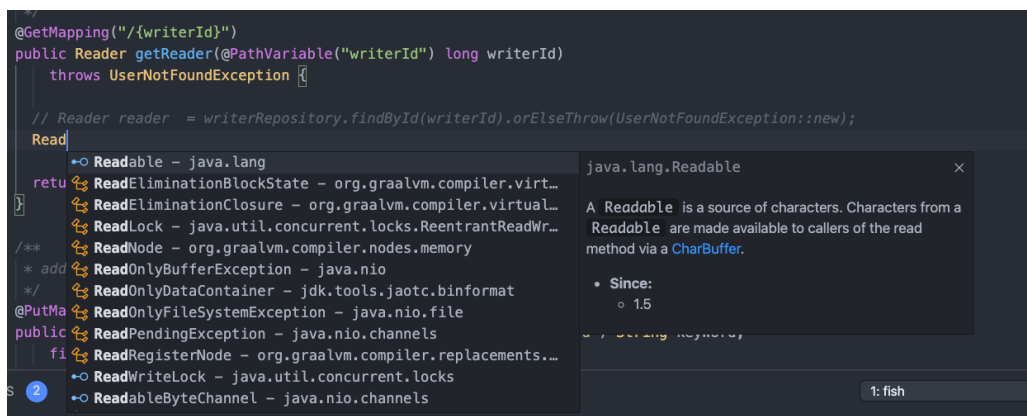


Figure 1. The Visual Studio Code pop-up window that provides suggestions

Regarding comment completion, current IDEs do not provide any tool for this task. In order to provide comment completion, the code surrounding the comment must be extracted and from that and the text already present in the

comment some text must generated. Therefore the model must not only have knowledge about the given programming language, but also about natural language. This peculiarity, makes comment completion a very different task from code completion and since comment completion can be also considered as not essential, this feature is usually not included in IDEs.

2.4 Summing Up

While many approaches have been proposed to improve code completion, there are no tools that can be easily used to experiment with these approaches and with real developers. The tool presented in this work solves this issue allowing to easily integrate code completion models just by specifying a service URL. The plugin also supports automatic feedback storing. Every time a developer accepts or rejects a recommendation, a feedback is saved on a server. These feedbacks can later be used to enhance the performance of the model.

Every major IDE already supports code completion by showing a pop-up window to the user. In recent years, with the rapid development of artificial intelligence a new way of performing code completion has started taking place. Not only neural networks are more accurate than standards code completion methods when asked to guess single tokens, but they are also capable of guessing multiple contiguous tokens and even entire code blocks. Since the length of suggestions is proportionally increasing with the "power" of neural networks, the standard pop-up list is not suitable anymore as a mean of showing recommendations to the user, because a big chunk of the suggestion would not be visible. In this work, we present a new intuitive and simple way of showing and interacting with long recommendations that neural networks are capable to provide.

3 S.I.M.O.N.

The tool proposed in this work, is a Visual Studio Code plugin that performs code and comment autocompletion through the use of neural networks. By default this plugin uses a very powerful implementation of a T5 neural network. However, the key feature of the system is the possibility for researchers to test their own models. The proposed models (one for code completion and one for comment completion) can be deployed on a server that respects the simple API of the plugin and by specifying the address of such server in the plugin settings the tool will make HTTP calls to the given neural network to retrieve recommendations. The suggestions coming from the model will then be shown in a non-intrusive (customizable) color directly in the text-editor. The plugin will take care of the business of parsing the java file, understanding the position of the caret (code or comment), triggering recommendations in the correct situations and understanding when recommendations are not needed anymore. An additional server can be specified in the settings. Every time the user accepts or rejects a recommendation, an HTTP request will be made to this server to store the feedback that can later be used to further enhance the model performance. In this section, the Visual Studio Plugin is presented to the reader, with a detailed explanation of its key features and components. Specifically a deep description is given regarding the VSCode API, the API the provided model must respect in order to interface with the plugin and the API for the feedback storing web service. Finally, through the use of images, usage scenarios are presented to the reader.



Figure 2. The User Interface of the plugin: the recommendation coming from the neural network is shown in a light-grey color directly in the text-editor

3.1 The (default) Models Used by the Plugin

The default model for code completion used in this project is the one proposed and developed by Ciniselli et al. [2] in a paper currently under review by TSE. It is a T5 model specifically built and tuned to perform code completion (another T5 model is used for comment completion).

The T5 is based on the transformer model architecture that allows handling a variable-sized input using stacks of self-attention layers. When an input sequence is provided, it is mapped into a sequence of embeddings passed into the encoder. In his work, Ciniselli compares the T5 model to RoBERTa, an instance of a BERT model, and the n-gram model in three different scenarios (i) token-level predictions, the model is used to guess the last n tokens in

a statement; (ii) construct-level predictions, the model is used to predict specific code constructs (e.g., the condition of an if statement); and (iii) block-level predictions, with the masked code spanning one or more entire statements composing a code block. The results of the work, showed that the T5 model outperforms by a wide margin the other two models in all the three scenarios. In particular, T5 was able to correctly guess all masked tokens in 66% to 69% of cases, with RoBERTa achieving 39% to 52% and the n-gram model 42% to 44%. In the most challenging prediction scenarios, in which entire blocks are masked, RoBERTa and the n-gram model show their limitations, being able to only correctly reconstruct the masked block in less than 9% of the cases, while the T5 achieves 30% of correct predictions.

A different T5 model is used for comment completion, this time specifically trained to perform comment completion. Also in this case, the T5 model outperforms the 5-gram model by a significant span for all the considered metrics.

3.2 Design

3.2.1 Visual Studio Code Plugin

The VSCode Api is meager and not well documented. Some of the feature of the API, are insufficiently explained or even not explained at all. Moreover the support of the community is scarce. Sometimes looking at the source code of other plugins with similar features, represents the best idea to rapidly move forward with the implementation. For instance, looking at the implementation of the famous GitLens plugin (<https://gitlens.amod.io>) has been fundamental in understanding how to show recommendations directly in the text editor in a non-intrusive way, since this feature is not mentioned anywhere. In the following sub-sections, the main components of the plugin are presented to the reader.

3.2.1.1 Parse the Java File

In order to call the correct model (code or comment completion model), every time the user types a character, the plugin needs to understand whether the user is typing a line of code or a comment. Specifically, the caret can be in a line of code, in a single line comment, in a multi line comment or in a javadoc. Even though VSCode clearly knows this information, since the color of the text editor changes based on the above scenarios, it is not possible to access this information through an API call. The only solution left is to parse the java file manually. Several functions have been created to guess the correct type of the current line and they can be found in `src/parser.js`. Different heuristics are used to parse the file, some are more complicated than others. For instance, to understand the presence of a single line comment it is sufficient to look for two subsequent forward slashes in the same line of the caret. Understanding the presence of a multi line comment is a little bit more challenging. Starting from the current line, an opening comment token (i.e. `/*`) must be found before the token that signals the closing comment (i.e. `*/`). The same reasoning can be applied for parsing javadocs.

In order to generate a recommendation, the content of the method in which the suggestion is requested is needed. VSCode does not provide any API to parse the content of the file. Therefore, a regular expression has been used to find the content of the current method. Once the method has been found, some tokens must be injected in the code to tell the neural network where the suggestion is expected. Depending on the type of suggestion expected (code or comment) different tokens are needed. For the code completion neural network it is sufficient to insert the token `"extra_id_0"`, where the suggestion is expected. For comment completion, instead, more sophisticated heuristics must be used:


- Insert `"extra_id_0"` where the suggestion is expected.
- Replace the beginning and the end of the comment (i.e. `//`, `/* */`, `/** */`) with `"<sep>"`.
- From `"extra_id_0"` parse the file up until an empty line, a comment or the beginning of the method is found.
- From `"extra_id_0"` parse the file up until an empty line, a comment or `"}"` is found.

Again, this parsing must be carried out manually since VSCode does not provide any utility for parsing files.

Once the String to send to the neural network has been computed according to the above rules, the communication with the server can be established and a recommendation can finally be requested.

3.2.1.2 Capture User Input

After having computed the line type, the plugin must decide what is the right action to take. Also in this case, there is not an API call for that, therefore understanding the behaviour of the user (deleting, typing a new line, etc...) must be done completely manually and it has been very challenging. There are many conditions the plugin checks at each keystroke, here's the most important ones:

- When the user presses  to create a new line, a recommendation is shown.

- If a recommendation is present and the user types something different, the recommendation fades away.
- If a recommendation has just disappeared (because the user has typed something different), but the user corrects its mistake, the suggestion reappears.
- If the user manually types a recommendation, it progressively shrinks.
- If the user types one of the (customizable) trigger characters a recommendation is shown.

Suggestions come from a neural network hosted on a sever, therefore they are not instantaneous. On average, it takes around 1.3s for the suggestion to arrive. While a suggestion is pending, the user can of course still write in the editor and even trigger other suggestions. Therefore, these are the three possible scenarios that the plugin must be able to handle:





- While there is a pending suggestion, the user types another trigger character. A new suggestion is requested from the server and the result of the previous request is discarded. This behavior must hold with an infinite number of subsequent pending suggestions. No negative feedback is sent to the feedback server for the discarded suggestion(s).
- While there is a pending suggestion, the user types some text that matches the beginning of the suggestion. In this case, the suggestion is shown starting from the end of the match.
- While there is a pending suggestion, the user types some text that does not match the beginning of the suggestion. In this case, the suggestion is discarded and not shown to the user. No negative feedback is sent to the feedback server.




3.2.1.3 Show Recommendations

Once a recommendation arrives from the neural network it must be shown to the user (and perhaps even deleted if the user discards it). The plugin shows the recommendation in a non-intrusive way directly in the text editor, using the VSCode annotations API. A suggestion can span multiple lines and the presence of enough white lines below the current one is not guaranteed. Therefore, enough empty lines must be added below the current one according to the size of the suggestion. In order to show the suggestion, it is sufficient to use an API call. The position of the suggestion must be specified as well as the color. The color can be customized in the settings of the plugin to match any theme the user might use. Another API call can be used to remove the suggestion in case it is not needed anymore. When a suggestion is removed, the additional lines previously inserted must be removed as well. The code used to implement this feature can be found in `src/editor.js`.

3.2.1.4 Keyboard shortcuts

The behavior of the plugin can be controlled through a set of (customizable) keyboard shortcuts.

-  (tab) can be used to accept a recommendation.
-  (escape) can be used to manually discard a recommendation
-  +  (ctrl + spacebar) can be used to manually trigger a recommendation.

 and  change behavior depending on the context. For instance, it is still possible to indent code using , only when there is a recommendation this key changes its behavior.

3.2.1.5 Settings

Different settings can be tweaked to change the behavior of the plugin. This settings are defined in `/package.json`. Here's a complete list:

- "Url Code" can be used to specify the address of the code completion service.
- "Url Comment" can be used to specify the address of the comment completion service.
- "Url Feedback" can be used to specify the address of the feedback storing service.
- "Confidence Code" defines the confidence threshold for the code completion neural network under which recommendations are not shown to the user.
- "Confidence Comment" defines the confidence threshold for the comment completion neural network under which recommendations are not shown to the user.

- "Name" is an optional additional parameter inserted in the feedback schema that can be used to distinguish between feedbacks of different users/organizations.
- "Single Line Comment" can be activated to move single lines comments in the same line of some code to the line above when code is sent to the neural network. This option is present because of the way some neural networks are trained.
- "Suggestion Color" is the color in which the suggestion is shown. This color must be customizable, because depending on the theme used, the default color could not be visible.
- "Trigger Chars" is a set of characters for which recommendations are triggered.

3.2.2 Code/Comment Completion Web Service

When the plugin decides that a recommendation must be shown, a communication with the neural network server must be established. In order to do so, there is a strict, but simple API the server must be able to respect. In particular, the plugin is going to send to the server a POST request with application/json as header and with the following JSON object as payload:

```
data = {
    "input": methodContent,
    "beam": N,
    "javadoc": inJavaDoc
}
```

where methodContent is the content of the method where the suggestion has been requested, beam is the number of recommendations the neural network must return and javadoc is a boolean value that tells if the user is requesting a suggestion for a "standard" comment or for a javadoc. To test the correct functioning of the server, the following command can be used:

```
curl -X POST -H "Content-Type: application/json" -d '{"input": "this.s = 10;", "beam": 2, "javadoc": 0}' http://the_url_of_your_server
```

The plugin expects a JSON object as a response. This object must contain a field "text" being an object containing the different recommendations sorted in decreasing order of confidence. Here's an example for better understanding the structure of the response:

```
{
    "elapsed time": 1.4944343,
    "text": {
        "pred_0": ['Prediction number 0', 0.74],
        "pred_1": ['Prediction number 1', 0.67],
        "pred_2": ['Prediction number 2', 0.64]
    }
}
```

3.2.3 Feedback Web Service

An additional address can be specified in the settings. This URL must lead to a server capable of storing feedbacks coming from the plugin. Every time a suggestion is accepted or rejected a POST request containing the feedback is sent to such server. This server is allowed to use whatever technology as long as it respects the simple API of the plugin. The following is the payload of the POST request that the plugin makes to such server in order to store the feedback:

```
data = {
    "suggestion": suggestion_of_neural_network,
    "methodContent": content_of_the_method,
    "isPositive": suggestion_accepted_or_rejected,
    "name": name_defined_in_settings
}
```

where "suggestion" is the actual suggestion that the user has just accepted or rejected, "methodContent" is the context sent to the neural network in order to request the suggestion (see section 3.2.1.1) "isPositive" is a boolean set to True if the suggestion has been accepted or to False if the suggestion has been rejected and "name" is the optional variable defined in the settings (see section 3.2.1.5). The variable "name" will not be present in the request if it not set in the settings. The following two headers are also set as follows:

```
'Accept': 'application/json',  
'Content-Type': 'application/json'
```

The default server for feedback storing is a Python server built with Flask and a SQLAlchemy database to store the feedbacks. The server has been dockerized and deployed on the Software Institute server.

3.3 Usage Scenarios

In this section some usage scenarios are presented in the form of images, to let the reader better understand how the plugin works in practice.

```
8      private void calculateMean() {  
9          double sum = 0;  
10         Integer count = 0;  
11         for(int i=0; i<calibrationData.length; i++) {  
12             if (calibrationFlag[i]) {  
13                 sum += calibrationData[i]; count++; }  
14             }  
15         mean = sum / count.doubleValue();  
16     }  
17
```

Figure 3. When a trigger character ("return" in this case), a recommendation appears

```
8      private void calculateMean() {  
9          double sum = 0;  
10         Integer count = 0;  
11         for(int i=0; i<calibrationData.length; i++) {  
12             if (calibrationFlag[i]) {  
13                 sum += calibrationData[i]; count++; }  
14             }  
15         mean = sum / count.doubleValue();  
16     }  
17
```

Figure 4. The suggestion shrinks when typed

```
8      private void calculateMean() {  
9          double sum = 0;  
10         Integer count = 0;  
11         for(int i=0; i<calibrationData.length; i++) {  
12             if (calibrationFlag[i]) {  
13                 sum += calik  
14             }  
15         mean = sum / count.doubleValue();  
16     }  
17
```

Figure 5. The suggestion disappears if something else is typed

```

8      private void calculateMean() {
9          double sum = 0;
10         Integer count = 0;
11         for(int i=0; i<calibrationData.length; i++) {
12             if (calibrationFlag[i]) {
13                 sum += calibrationData[i]; count++; }
14             }
15         mean = sum / count.doubleValue();
16     }
17

```

Figure 6. The suggestion reappears if the user corrects his mistake

```

18     protected void onSizeChanged(int w, int h, int oldw, int oldh) {
19         super.onSizeChanged(w, h, oldw, oldh);
20
21         // Make sure scroll position is set correctly.
22         if (h != old) {
23             recomputeScrollPosition(h, oldh, mPageMargin, mPageMargin);
24         }
25     }

```

Figure 7. A different Neural Network can be used to perform comments' suggestions

```

7      /**
8       * This method computes the mean of the calibration.
9       */
10     private void calculateMean() {
11         double sum = 0;
12         Integer count = 0;
13         for(int i=0; i<calibrationData.length; i++) {
14             if (calibrationFlag[i]) {
15                 sum += calibrationData[i]; count++; }
16             }
17         mean = sum / count.doubleValue();
18     }

```

Figure 8. The comment neural network can also recommend javadocs

4 Conclusions and Future Work

In this work, the history of code prediction, performed with neural networks, has been briefly presented. Neural networks represent a new way of performing code prediction. These models outperform by a wide margin all the traditional models and algorithms currently used in IDEs. Neural networks are not only more accurate in predicting single tokens, but they are also able to predict multiple contiguous tokens and even entire code blocks. Given the longer average length of predictions produced by neural networks, the traditional list, used by every major IDE, is not suitable anymore to show recommendations, because part of the recommendation would not be visible. The tool presented in this work solves this problem by showing recommendations directly in the text editor. Moreover, this tool enables researches to easily integrate their java prediction models in VSCode, in order to let them assess their usefulness in real world scenarios. Finally, a feedback feature can be used to collect additional data points that can later be used to further enhance the guessing capabilities of the model.

4.1 Future Work

The VSCode API does not give access to the tree that the IDE creates during parsing. Therefore, the code must be parsed manually and different heuristics must be used for different programming languages. For example, the current line is parsed in order to understand whether a comment or some code is being written, however the tokens to create comments change depending on the programming language. For this reason, the current version of the plugin only works with java. Researchers might instead come up with neural networks for other programming languages. Future version of the plugin should focus on extending the support to other languages. In the future, Microsoft might also provide a new API that gives access to the AST that VSCode creates. This might enable the future developers of this tool to write some more general code that works with any programming language.

References

- [1] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, “Roberta: A robustly optimized BERT pretraining approach”, CoRR, vol. abs/1907.11692, 2019. [Online]. Available: <http://arxiv.org/abs/1907.11692>
- [2] Matteo Ciniselli*, Nathan Cooper†, Luca Pascarella*, Denys Poshyvanyk†, Massimiliano Di Penta‡, Gabriele Bavota* “An Empirical Study on the Usage of BERT Models for Code Completion”
*SEART @ Software Institute, Università della Svizzera italiana (USI), Switzerland
†SEMERU @ Computer Science Department, William and Mary, USA
‡Department of Engineering, University of Sannio, Italy
- [3] R. Robbes and M. Lanza, “Improving code completion with program history”, Automated Software Engineering, vol. 17, no. 2, pp. 181–212, 2010.
- [4] U. Alon, R. Sadaka, O. Levy, and E. Yahav, “Structural language models of code”, arXiv, pp. arXiv–1910, 2019.
- [5] M. Asaduzzaman, C. K. Roy, K. A. Schneider, and D. Hou, “Context- sensitive code completion tool for better api usability”, in 2014 IEEE International Conference on Software Maintenance and Evolution, 2014, pp. 621–624.
- [6] G. A. Aye and G. E. Kaiser, “Sequence model design for code completion in the modern ide”, arXiv preprint arXiv:2004.05249, 2020
- [7] A. Bhoopchand, T. Rocktäschel, E. Barr, and S. Riedel, “Learning python code suggestion with a sparse pointer network”, arXiv preprint arXiv:1611.08307, 2016
- [8] M. Bruch, M. Monperrus and M. Mezini, “Learning from examples to improve code completion systems”, in Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, ser. ESEC/FSE 2009, 2009, pp. 213–222.
- [9] T. Gvero, V. Kuncak, I. Kuraj, and R. Piskac, “Complete completion using types and weights”, in ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI ’13, Seattle, WA, USA, June 16–19, 2013, 2013, pp. 27–38.
- [10] V. J. Hellendoorn and P. Devanbu, “Are deep neural networks the best choice for modeling source code?”, in Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ser. ESEC/FSE 2017, 2017, p. 763–773.
- [11] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, “On the naturalness of software”, in Proceedings of the 34th International Conference on Software Engineering, ser. ICSE 2012. IEEE Press, 2012, pp. 837–847.
- [12] R. Karampatsis and C. A. Sutton, “Maybe deep neural networks are the best choice for modeling source code”, CoRR, vol. abs/1903.05734, 2019. [Online]. Available: <http://arxiv.org/abs/1903.05734>.
- [13] S. Kim, J. Zhao, Y. Tian, and S. Chandra, “Code prediction by feeding trees to transformers”, arXiv preprint arXiv:2003.13848, 2020.
- [14] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman, , “Jungloid mining: helping to navigate the API jungle”, in Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12–15, 2005, 2005, pp. 48–61.
- [15] A. T. Nguyen, T. T. Nguyen, H. A. Nguyen, A. Tamrawi, H. V. Nguyen, J. Al-Kofahi, and T. N. Nguyen, “Graph-based pattern-oriented, context-sensitive source code completion”, in 2012 34th International Conference on Software Engineering (ICSE), 2012, pp. 69–79.
- [16] S. Proksch, J. Lerch, and M. Mezini, “Intelligent code completion with bayesian networks”, ACM Trans. Softw. Eng. Methodol., vol. 25, no. 1, pp. 3:1–3:31, 2015.
- [17] V. Raychev, M. Vechev, and E. Yahav, “Code completion with statistical language models”, in Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, ser. PLDI 2014, 2014, pp. 419–428.
- [18] P. Rodeghero, C. McMillan, P. W. McBurney, N. Bosch, and S. D’Mello, “Improving automated source code summarization via an eye-tracking study of programmers”, in Proceedings of the 36th International Conference on Software Engineering, ser. ICSE 2014, 2014, pp. 390–401.

- [19] B. P. Eddy, J. A. Robinson, N. A. Kraft, and J. C. Carver, "Evaluating source code summarization techniques: Replication and expansion", in International Conference on Program Comprehension (ICPC), 2013, pp. 13–22.
- [20] S. Haiduc, J. Aponte, and A. Marcus, "Supporting program comprehension with source code summarization" in International Conference on Software Engineering - Volume 2 (ICSE'10), vol. 2. IEEE, 2010, pp. 223–226.
- [21] S. L. Abebe and P. Tonella, "Extraction of domain concepts from the source code", Science of Computer Programming, vol. 98, pp. 680–706, 2015.
- [22] J. Fowkes, P. Chanthirasegaran, R. Ranca, M. Allamanis, M. Lapata, and C. Sutton, "Autofolding for source code summarization," IEEE Transactions on Software Engineering, vol. 43, no. 12, pp. 1095–1109, 2017.
- [23] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker, "Towards automatically generating summary comments for java methods," ser. ASE '10, 2010, pp. 43–52.
- [24] S. Rastkar, G. C. Murphy, and A. W. J. Bradley, "Generating natural language summaries for crosscutting source code concerns," in International Conference on Software Maintenance (ICSM), 2011, pp. 103–112.
- [25] J. Li, A. Sun, and Z. Xing, "Learning to answer programming questions with software documentation through social context embedding," Information Sciences, vol. 448, pp. 36–52, 2018.
- [26] R. Holmes and G. C. Murphy "Using structural context to recommend source code examples," in Proceedings of the 27th International Conference on Software Engineering, ser. ICSE '05, pp. 117–125.
- [27] J. Stylos and B. A. Myers, "Mica: A web-search tool for finding api components and examples," in VLHCC 2006), pp. 195–202.
- [28] G. C. Murphy, R. J. Walker, and R. Holmes, "Approximate structural context matching: An approach to recommend relevant examples," IEEE Transactions on Software Engineering, vol. 32, pp. 952–970, 12 2006.
- [29] L. Moreno, G. Bavota, M. Di Penta, R. Oliveto, and A. Marcus, "How can i use this method?" ser. ICSE '15, pp. 880–890
- [30] S. Subramanian, L. Inozemtseva, and R. Holmes, "Live api documentation," in ICSE 2014, pp. 643–652.
- [31] S. Jiang, A. Armaly, C. McMillan, Q. Zhi, and R. Metoyer, "Docio: Documenting api input/output examples," ser. ICPC '17, pp. 364–367.
- [32] C. Treude and M. P. Robillard, "Augmenting api documentation with insights from stack overflow," ser. ICSE '16, pp. 392–403.
- [33] M. P. Robillard and Y. B. Chhetri, "Recommending reference api documentation," Empirical Softw. Engg., vol. 20, no. 6, pp. 1558–1586, 2015.
- [34] M. M. Rahman, C. K. Roy, and I. Keivanloo, "Recommending insightful comments for source code using crowd-sourced knowledge," in 2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM), 2015, pp. 81–90.
- [35] R. P. L. Buse and W. Weimer, "Synthesizing api usage examples," in Proceedings of the 34th International Conference on Software Engineering, ser. ICSE '12. IEEE Press, 2012, pp. 782–792.
- [36] S. K. Bajracharya, J. Ossher, and C. V. Lopes, "Leveraging usage similarity for effective retrieval of examples in code repositories," in Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering, ser. FSE '10, pp. 157–166.
- [37] C. Mcmillan, D. Poshyvanyk, M. Grechanik, Q. Xie, and C. Fu, "Portfolio: Searching for relevant functions and their usages in millions of lines of code," ACM Trans. Softw. Eng. Methodol., vol. 22, no. 4, pp.37:1–37:30, 2013.
- [38] L. Ponzanelli, G. Bavota, M. Di Penta, R. Oliveto, and M. Lanza "Mining stackoverflow to turn the ide into a self-confident programming prompter," in Proceedings of the 11th Working Conference on Mining Software Repositories, 2014, pp. 102–111.
- [39] L. Ponzanelli, S. Scalabrino, G. Bavota, A. Mocci, R. Oliveto, M. Di Penta, and M. Lanza, "Supporting software developers with a holistic recommender system," ser. ICSE '17, 2017, pp. 94–105.
- [40] T. Xie and J. Pei, "MAPO: Mining API usages from open source repositories," in Proc. of the 2006 Int. Workshop on Mining Software Repositories, 2006, pp. 54–57.

- [41] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei, “*Mapo: Mining and recommending api usage patterns*,” in ECOOP 2009–Object-Oriented Programming. Springer, 2009, pp. 318–343.
- [42] J. Wang, Y. Dang, H. Zhang, K. Chen, T. Xie, and D. Zhang, “*Mining succinct and high-coverage API usage patterns from source code*,” ser. MSR ’13, 2013, pp. 319–328.
- [43] E. Wong, T. Liu, and L. Tan, “*CloCom: Mining existing source code for automatic comment generation*,” in Software Analysis, Evolution and Reengineering (SANER), 2015, pp. 380–389.
- [44] E. Aghajani, G. Bavota, M. Linares-Vasquez, and M. Lanza, “*Auto- mated documentation of android apps*,” IEEE Transactions on Software Engineering, vol. 47, no. 1, pp. 204–220, 2021.
- [45] E. Wong, J. Yang, and L. Tan, “*Autocomment: Mining question and answer sites for automatic comment generation*,” in International Conference on Automated Software Engineering (ASE), 2013, pp. 562–567.
- [46] N. Nazar, H. Jiang, G. Gao, T. Zhang, X. Li, and Z. Ren, “*Source code fragment summarization with small-scale crowdsourcing based features*,” Frontiers of Computer Science, vol. 10, no. 3, pp. 504–517, 2016.
- [47] A. T. T. Ying and M. P. Robillard, “*Code fragment summarization*,” in Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, ser. ESEC/FSE 2013, pp. 655–658.
- [48] W. Zheng, H.-Y. Zhou, M. Li, and J. Wu, “*Code attention: Translating code to comments by exploiting domain features*,” 2017.
- [49] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, “*Deep code comment generation*,” ser. ICPC ’18. Association for Computing Machinery, 2018, pp. 200–210.
- [50] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, “*Summarizing source code using a neural attention model*,” in Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), 2016, pp. 2073–2083.
- [51] A. Mastropaolo, S. Scalabrino, N. Cooper, D. Palacio, D. Poshy- vanyk, R. Oliveto, and G. Bavota, “*Studying the usage of text-to- text transfer transformer to support code-related tasks*,” in 43rd Inter- national Conference on Software Engineering (ICSE 2021), 2021, p. <https://arxiv.org/pdf/2102.02017.pdf>.
- [52] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, “*Exploring the limits of transfer learning with a unified text-to-text transformer*,” 2019.
- [53] A. LeClair, S. Jiang, and C. McMillan, “*A neural model for generating natural language summaries of program subroutines*,” ser. ICSE ’19, 2019, pp. 795–806.
- [54] Y. Liang and K. Q. Zhu, “*Automatic generation of text descriptive comments for code blocks*,” CoRR, vol. abs/1808.06880, 2018.
- [55] A. Svyatkovskiy, S. K. Deng, S. Fu, and N. Sundaresan, “*Intellicode compose: Code generation using transformer*,” arXiv preprint arXiv:2005.08025, 2020.
- [56] A. Svyatkovskiy, S. Lee, A. Hadjitofi, M. Riechert, J. Franco, and M. Allamanis, “*Fast and memory-efficient neural code completion*,” 2020.
- [57] Z. Tu, Z. Su, and P. Devanbu, “*On the localness of software*,” in Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, ser. FSE 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 269–280. [Online]. Available: <https://doi.org/10.1145/2635868.2635875>