Deep Learning Powered Python Autocomplete

Authors: Niki Agrawal*, Kevin Tran*, Mauricio Wulfovich*

**All authors contributed equally*

Problem Definition

The population of professional software developers has been rapidly growing in recent years. According to Evans Data Corporation, the number of professional software developers is expected to reach 28.7 million by 2024. To increase developer productivity, our team built a code autocomplete tool to reduce careless mistakes and to speed up development time. Our tool is currently developed for Python users since Python is one of the most commonly used programming languages, but we hope to extend it to other languages.

Our tool is an extension that is directly integrated with Visual Studio Code (VS Code) and while VS Code has built-in autocomplete tools itself that can autocomplete variable and function names, these tools have yet to exploit common patterns in Python code (i.e. if you type “import pandas as”, you will not get an autocomplete recommendation of “pd” or if you type “import torch as” you will not get an autocomplete recommendation of “nn”). To take advantage of these common patterns our team built a machine-learning powered autocomplete trained on large corpuses of Python code.

In the next section, we will take a deep dive into the system architecture of our autocomplete tool and introduce its various components and algorithms.

System Design

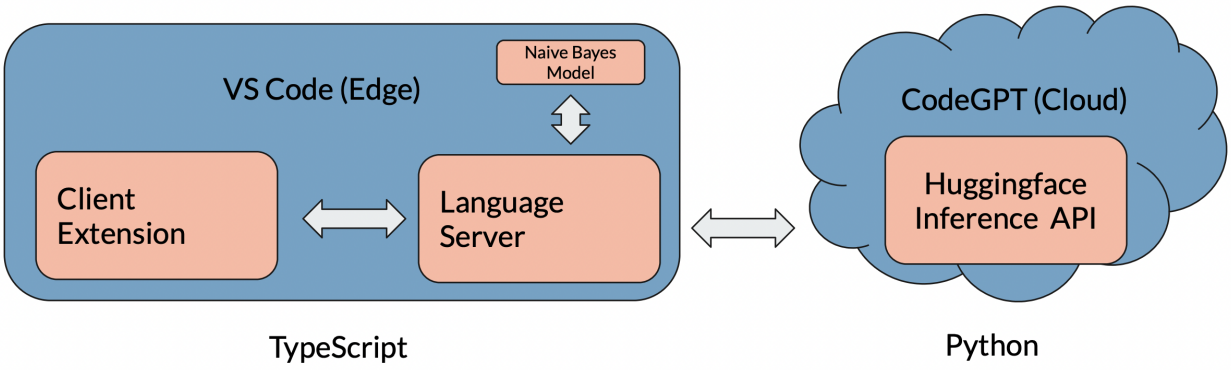


Figure 1. System Design Diagram

The first component in our system is the client extension. Whenever a user changes a file on the client extension, the client extension will make a GET request to the second component in our system, the language server. The language server's job is to compile a list of autocomplete items and send it back to the client.
When the language server gets a request from the client extension, it first makes a call to our third component, the CodeGPT model. The CodeGPT model is hosted on the cloud by Huggingface and is called using the Huggingface Inference API. The language server sends the last two lines of prior context to CodeGPT. The CodeGPT model then returns a list of word sequences (Python code line completion) to the language server. The language server loops through this list of word sequences and only keeps the next immediate word of each word sequence predicted by the CodeGPT model. We made this decision because if we autocomplete a sequence, and one of the words in the sequence is incorrect, the user will have to delete the incorrect word which could lead to an unpleasant user experience. As an example, imagine that the language-server passed in "from pandas import" to the CodeGPT model. A possible list of word sequences returned by CodeGPT are ["from pandas import DataFrame, to_csv", "from pandas import Series, panel", "from pandas import to_csv, panel"]. When the language-server preprocesses this list of word sequences, it will only keep the next immediate word of each word sequence, namely "DataFrame", "Series", and "to_csv". These next immediate words are the first items to be added to the list of autocomplete items.

After processing the output of the CodeGPT model, the language server makes a call to our fourth component, the Naive Bayes model, which is hosted locally. The Naive Bayes model returns a list of up to 500 of the most likely Python words, independent of context. We train the Naive Bayes model on the Py150 Corpus. The language server finishes up the list of autocomplete items by adding in this list of frequently used Python words. It then sends this list to our client extension. The client extension then performs one final preprocessing step to filter out suggestions that don't contain the current word being typed as a substring. Once this preprocessing is done, the client extension will display the autocomplete items ranked by relevance. The user can mouse click or hit Enter to autocomplete.

Before we move on to further explaining the machine learning components of our system, we want to briefly explain the latency of the CodeGPT model. We experimented with different approaches for querying our CodeGPT model, timing these approaches to optimize for latency. We found that using the Huggingface Inference API (cloud-based approach) was twice as fast as obtaining inference results locally on the edge with CPU. This API has built in optimizations for its models, such as reducing the amount of computation in each forward pass. It also caches responses for previously seen inputs. Since our class is on the Startup plan, we used Accelerated GPU inference on Huggingface's server, which was generally about 0.2 seconds faster than Huggingface's CPU inference. We did not consider running the model on the Edge with GPU since many users might not have one.

Machine Learning Component

Our baseline is a Naive Bayes model. Naive Bayes assumes strong conditional independence between features. That is, the probability of every word occuring is independent of its context. We built the naive bayes model by first creating a term frequency table from a large corpus of Python git repos. We preprocessed the corpus by removing all comments. We then normalized the table into probabilities by dividing all counts by the sum total of words in the corpus. For numeric stability, we stored log probabilities. We then rank autocomplete suggestions in order of most to least probable words. While this model had very low latency, since it ran on the edge, its suggestions were limited and did not account for context.

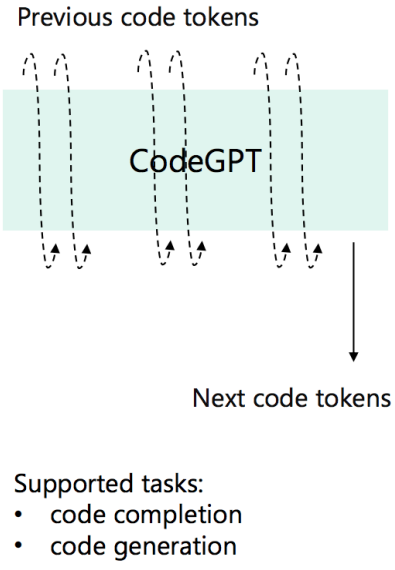


Image from CodeXGlue paper

After building a full end-to-end autocomplete system using the Naive Bayes Model, we decided to incorporate a better model into our system. We decided on Microsoft's CodeXGlue's CodeGPT model. This model enables autocomplete suggestions to change based on what the

user has already typed, unlike Naive Bayes. We experimented with variations of this model, obtaining the best results with fine-tuned CodeGPT-small-py. This model has the same architecture and training object as the GPT-2 language generation model, containing 12 layers of Transformer decoders. However, CodeGPT is more appropriate for our task than GPT-2 since CodeGPT is pre-trained for the code completion and text-to-code generation tasks on the CodeSearchNet Python corpus, which includes 150K Python source files, 1.14M Python functions, and 119M tokens. 100K of these files were used in the train dataset and 50K in the test dataset. The CodeGPT variation we use is fine-tuned only on the code completion task of next token prediction for 5 epochs, again using the CodeSearchNet corpus. We believe the fine-tuned model performs better than the pre-trained model because it is finetuned specifically for our task, which is prediction of the next token given prior context.

Table 4: Parameters of CodeBERT and CodeGPT models.

	CodeBERT	CodeGPT
Number of layers	12	12
Max length of position	512	1,024
Embedding size	768	768
Attention heads	12	12
Attention head size	64	64
Vocabulary size	50,265	50,000
Total number of parameters	125M	124M

Image from CodeXGlue Paper

We experimented with several CodeGPT model parameters in our iterations, including input preprocessing, input length, number of sequences returned, sampling methods, and diversity penalty. The quality of our predictions was poor if we passed an input containing an incomplete word, for example: “from pandas im”. We hypothesize that the model is designed to take complete words as input, and often partial words may not be recognizable tokens. Thus, we currently preprocess our inputs to only include complete words, so our input to the model will be “from pandas” if the user has typed “from pandas im” in order to get the suggestion “import”. In addition, we wanted CodeGPT to output multiple possible suggestions for the next token. By default, the model performs greedy decoding and only outputs one sequence of predicted words. By adding the num_return_sequences and num_beams parameters, we changed the sampling method to beam search and generated multiple suggested sequences. To generate diverse suggestions, we set the diversity_penalty parameter to 0.5 after experimentation, which penalizes the model for generating the same token as a different beam at a given time. These parameters enable CodeGPT to generate multiple, diverse suggestions for the next word given context. If the low confidence suggestions have poor quality, they are unlikely to impact user experience because a suggestion only appears if the user has typed a matching first letter.

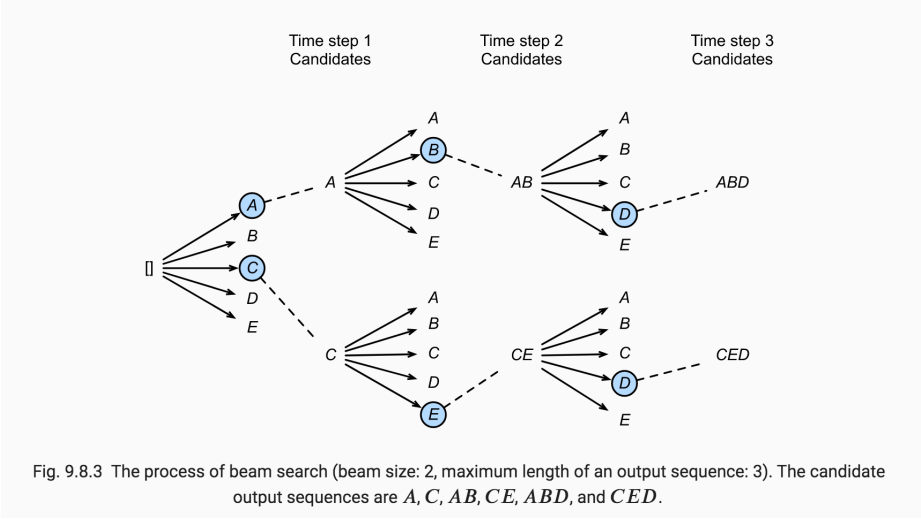
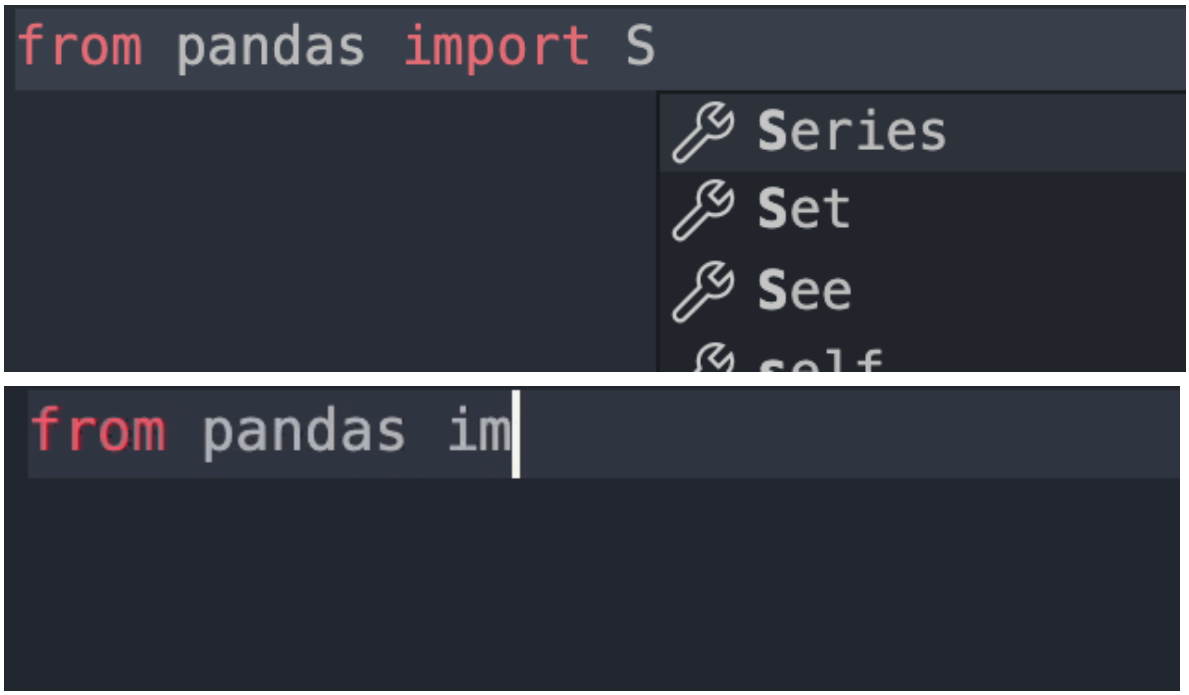


Fig. 9.8.3 The process of beam search (beam size: 2, maximum length of an output sequence: 3). The candidate output sequences are A, C, AB, CE, ABD, and CED.

Image from Reference [10]

Demo GIFs



Our VSCode autocomplete is fully integrated into VSCode and uses the interface that its users are accustomed to. Currently, our autocomplete is not available on the VSCode marketplace, due to a lack of access to HuggingFace API after the quarter ends, and lack of sustained funding to purchase continued access. However, our code for the extension is available upon request.

Model Evaluation

Our model attains an accuracy of 72.2% on the PY-150 corpus in predicting the next token. The model has a perplexity of 2.50. Qualitatively, we see that the model correctly predicts the most intuitively likely next tokens:

Correct predictions:

Input	**Output**
def add(a, b):	return a + b
for i in	range
line = reader.	readline()

Error analysis

We noticed patterns in our model's errors:

1. Lack of context: oftentimes variables are initialized much earlier than the models input window. Without the context, the model must make predictions without the required information for the correct answer. For example, in error 1 below, masked is an array that was declared much earlier in the code.
2. Black box errors: sometimes the errors the model makes aren't understandable. More or better quality training data could probably help. See example error 2.

3. Lack of knowledge of library methods: while the model correctly learns valid arguments to common functions, for less common functions, the model can sometimes suggest methods that do not exist. In example 3, sns.PairGrid.grid does not exist.

Example errors:

	Input	**Model Output**	**Correct Output**
1	window[j] = constants.MASK <EOL> masked +=	1	window
2	raise	def test_get_all_by_id	NotImplementedError
3	import seaborn as sns sns.PairGrid	.grid	.map

User Evaluation

Iterative Evaluation

After integrating CodeGPT into our autocomplete, we decided to ask multiple users to evaluate our autocomplete system. Our users were undergraduate students in computer science who are proficient in Python and have used other Python autocomplete systems. We had a variety of directions we could take our autocomplete, and we wanted to get qualitative user feedback before deciding on features to prioritize. So we set up the extension, and asked the users to build a simple `Person` class with a `greet` method, and then provide us with feedback. Our users mentioned that latency and autocomplete after periods & parentheses were the most valuable features to build. We took the feedback to heart, and added the ability to query the HuggingFace API with a GPU, and built the autocomplete after periods and parentheses as they mentioned.

Final Evaluation

After our demo, we also decided to ask a few more users to evaluate our autocomplete system in order to have a sense of our current system's strengths and weaknesses, as well as features we could build in future iterations. We compile a subset of their feedback below:

Strengths:

1. Autocompleting previously entered variable names
2. Idiomatic python completions (a feature not available in Intellicode or Pycharm as of the time of writing)

    - For instance, after the users types ‘import pandas as…’ our autocomplete suggests ‘pd’

Future Areas of improvement:

1. Further decreasing latency: We can address this by distilling our model so it can run on the Edge and testing whether the results are still high quality.
2. Suggesting library methods: For instance, after typing ‘np.linalg.’ users would like to see suggestions for common methods in the submodule such as ‘dot’, ‘norm’ etc. We can address this by improving our Deep Learning model.

Reflection:

Overall, we are pleased with our work. We were able to build an end-to-end VSCode autocomplete extension with exciting features that are not present in the default VSCode. Through this project, we gained valuable experience in building an end-to-end system powered by ML. We gained both technical and project management experience -- ideating on a problem we were passionate about, finding the right platform to build it on, building an MVP, prioritizing features, researching and experimenting with models, evaluating our system, and iterating. For our MVP, we decided to build an end-to-end system with autocomplete suggestions as ranked by a normalized term frequency table. The term frequency approach is a rule-based, low-latency approach for autocompletion that is still backed by data. Thus, starting with this approach rather than trying to use a language model from the get-go enabled us to build an end-to-end system with a pleasant user interface early on. Then we could focus our energies on addressing the pain points of the MVP by integrating CodeGPT into it, which enabled context-based suggestions.

Next time, we would use additional methods for user evaluation. We could also perform user evaluation on more types of problems to get more holistic feedback. For example, if we had performed user evaluations on longer coding problems, we could have received more feedback related to how much context users would like autocomplete suggestions to take into account. Our current system will not be able to autocomplete a variable name that the user typed several lines away. In addition, we could have evaluated our system with both experienced coders and people just starting out. Finally, we could have done more iterations on our term frequency table, such as by adding more terms based on user interviews.

Given unlimited time and resources, we would start by addressing issues brought up by our users as stated above. In addition, we would improve our system through the following techniques:

- Supplement the system with more rule-based models, such as detecting variable names as words typed before the “=” sign or function names as words typed after “def” and store these for future suggestions. These rule-based techniques would likely improve our system with no latency cost.
- Feed in more context to the CodeGPT model, beyond the current and previous line.
- Train our own Deep Learning model.

- We could finetune GPT-3 on the CodeSearchNet Python corpus. Fine-tuning this model could have better performance than CodeGPT, which has the architecture of GPT-2.
- Our training dataset can include incomplete words, such as “from pandas im” to more accurately reflect the autocomplete input.
- We can experiment with hyperparameters like learning rate and number of epochs

Ultimately, our goal would be to publish our extension on the VSCode Marketplace, to integrate our autocomplete system into other developer tools such as Jupyter notebook and the IDLE IDE, and to enable autocomplete on even more programming languages.

Broader Impacts:

The intended use case of our application is for Python developers to improve their development productivity by helping them autocomplete their code. One possible unintended use case of our application is overreliance on our autocomplete tool. Our autocomplete tool is by no means perfect and in some cases, does not recommend the most appropriate autocomplete suggestion. If the user just auto-completed cases that had poor quality autocomplete suggestions without having considered their previous code, they may accidentally introduce a bug in their code that they would have to find later on. To mitigate the risk associated with overreliance on our autocomplete code, we made sure to only recommend autocomplete items one word at a time, so that it helps speed up development time but at the same time ensures that users are still completely aware of the flow of their code.

References:

[1] CodeGPT model in Model Hub: https://huggingface.co/mrm8488/CodeGPT-small-finetuned-python-token-completion
[2] CodeXGlue Paper including CodeGPT model: CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation (arxiv.org)
[3] CodeXGlue Code Completion Repo: CodeXGLUE/Code-Code at main · microsoft/CodeXGLUE · GitHub
[4] CodeXGlue blog post: https://www.microsoft.com/en-us/research/blog/codexglue-a-benchmark-dataset-and-open-challenge-for-code-intelligence/
[5] Term Frequency Table: https://anvaka.github.io/common-words/#?lang=py
[6] Term Frequency Table: https://github.com/anvaka/common-words
[7] Huggingface Text Generation Parameters: https://huggingface.co/blog/how-to-generate
[8] Huggingface Inference API: https://huggingface.co/blog/accelerated-inference
[9] VSCode Extensions Overview: Extension Guides | Visual Studio Code Extension API
[10] Beam Search Image: https://d2l.ai/chapter_recurrent-modern/beam-search.html

Link to Github:

Link to our github repo: https://github.com/mauriw/lsp/
Link to the Markdown: https://github.com/kevtran23/reports