**Problem**

Artificial Intelligence (AI) has revolutionised various aspects of our daily lives, from virtual assistants like Siri and Alexa to self-driving cars and personalised recommendations. However, the latest breakthrough in AI research has been the development of Large Language Models (LLMs) which use deep learning algorithms to comprehend and generate human-like text. As opposed to previous recurrent neural networks and long-short term memory cells, LLMs utilise the transformer architecture and contain billions of parameters. Some of the most popular LLMs, such as the Generative Pre-trained Transformer (GPT) family of models, have newly redefined the state of the art, especially GPT-3 [[1](https://arxiv.org/abs/2005.14165)]. These models have the capability to perform a wide range of tasks, including text generation, translation, and question answering. While these possibilities present exciting opportunities for automation and innovation, they also raise concerns about the impact of this technology on society.

OpenAI's research agenda emphasises the need for further research into the impact of AI models on employment. The agenda cites a survey that estimates a 50% chance of AI automating 90% of human tasks by 2045 [[Source](https://www.sciencedirect.com/science/article/pii/S0268401223000233)]. Older assessments and surveys indicate that 47% of jobs in the United States are vulnerable to computer automation [[Source](https://www.sciencedirect.com/science/article/abs/pii/S0040162516302244?via%3Dihub)], and there is a 50% chance that AI will surpass humans in all tasks by 2063 [[Source](https://jair.org/index.php/jair/article/view/11222)]. One of the most promising applications of LLMs is their ability to generate code. However, given the aforementioned surveys, it is possible that human programmers may become replaceable. There are already AI tools available, such as Github Copilot [[Source](https://github.com/features/copilot)] that have been trained on enormous amounts of open source text data and are still used by current developers. For this reason, human employment, especially regarding human-coding labour, is one of the primary research areas of OpenAI’s agenda.

**Research Aim**

To assess the potential of language models replacing programmers, research was conducted to test current AI products and evaluate their ability to create a basic coding project. Other relevant work and literature were also examined to measure the current intelligence of AI in generative programming. Overall, the culmination of this study should answer the following:

**"Can modern language models replace human programmers, or are they simply tools?"**

**Method**

**Describing a Software Development Workflow**

Human programmers and software developers all possess a similar workflow. This acts as a guide from the conception of an idea to the creation of a product that is ready for deployment. The following briefly describes this step by step process:

1. **Design and planning**

This is where a project idea is drafted, refined and reflected upon. Brainstorming and planning heavily occurs at this stage

1. **Creation and coding**

Where the project is implemented, created and coded

1. **Testing and verifying**

Where the project is subject to numerous tests to ensure that it runs smoothly without inhibiting errors

1. **Documentation**

Where the project is explained in detail and recorded for future reference to maintain it

To assess the potential of generative AI replacing programmers, models should be tested at all 4 stages -- ideally, an AI model should be able to design a project idea, generate the code, perform tests and write appropriate documentation. However, to test models at all 4 stages would require extensive time and resources. To simplify this study, I decided to take an output-based approach that will test AI models on their ability to generate code and write documentation.

**Types of AI products Used**

There already exists a variety of language models like CodeBERT [[Source](https://arxiv.org/abs/2002.08155)], CodeT5 [[Source](https://arxiv.org/abs/2109.00859)] and Codex [[Source](https://arxiv.org/abs/2107.03374)] that can perform code generation. To evaluate all of them is unfeasible. Therefore I selected new and relevant products that are still actively being used by developers. The results of my selection were 4 products: Github Copilot (free-trial version), Tabnine (free version), ChatGPT-3 and the recent Bing Chat. However, these products can be classified into two types:

**Plug-in products -- includes Github Copilot and Tabnine**

It is important to note that although these 4 products generate code, Github Copilot and Tabnine are utilised in the IDE and will be referred to as plug-ins. These tools assist developers by analysing previous lines of code in their working directory and predicting the next lines of code. In this case, documentation will not be examined, only the final generated code.

**Chat-based products -- includes ChatGPT-3 and Bing Chat**

Unlike plug-ins, chat-based models require a user to input a question or an instruction. These models then produce a response relevant to the question. For code-related questions, code can possibly be embedded into the responses. A developer can leverage this feature to obtain specific code needed for a function or solutions to a coding problem. Additionally, because it is chat-based, a specific inquiry could be further explored by asking more questions. These models will consider the context from previous questions and the current one to generate appropriate responses. For these models, documentation and generated code will be evaluated.

**Provided Task and Intended Output**

To avoid complications, I decided to choose a simple coding task to test the 4 products. For each test case, I wanted to observe how each product can create a tic tac toe game based on the terminal command line and another based on the pygame library. In this case, the command line-based game would be considered as a “simple” task and the pygame based game would be considered as a “complex task”.

For plug-ins, the intended output is code that runs smoothly and allows a user to play tic tac toe. For chat-based products, the intended output is code that runs smoothly in addition to well-written documentation. The format of this documentation should be a README file that adheres to the following:

1. Includes a brief summary/description
2. How to install/run the project
3. Dependencies ( libraries and language versions)
4. Usage examples could include diagrams or code snippets of actual usage

**Measuring a Product’s Intelligence**

Because all language models require context (as either input words or input code), I measured each product’s overall intelligence by noting how much context was provided to it. Ideally, a model should be able to generate the intended output using minimal input/context. For humans, when given a task like creating a drawing application, humans assume that the README file should contain diagrams or example images of the project because of the nature of the project. Additionally, we know a drawing application should make use of specific frameworks like tkinter or pygame -- these are types of assumptions that individuals make because of the provided context. Humans do not need extra information and specificity in the project task because of this understanding.

Likewise, an AI model should be able to “generalise” to the task and make any necessary assumptions just how humans do. A high performing modelwould require less information to produce executable code and coherent documentation; the less human intervention to correct a model equates to a more intelligent model.

The type of input and measurement of intelligence can be summarised below:

**Table 1: Summary of inputs types and measurements of intelligence for each product**

| **Product Type** | **Chat-based** | **Plug-ins** |
| --- | --- | --- |
| **Input Type** | In the form of questions. | In the form of comments and code provided in the IDE script. |
| **Example Inputs** | * “Can you create me a tic tac toe game in python?” * “Can you create the appropriate README file for this project?” | * #python tic tac toe game * #tic tac toe game * #tic tac toe game in python * Tic tac toe code snippets from internet articles |
| **Measuring of Intelligence** | Measured qualitatively by how many questions asked and types of questions asked to the product. | Measured qualitatively by the amount of comments and code provided to the product. |

**Research**

Before proceeding it is important to understand the underlying technology behind modern language modelling. The first section of this research highlights

**A Brief Review of Generative Language Models**

**N-grams**

A very simplified description of how language modelling works is: “given the previous words (context), what is the probability of the next word?” [[Source](https://builtin.com/data-science/beginners-guide-language-models)]. This can be done numerous ways. Historically, this began with the N-gram model that calculates a probability distribution of words based on the n-1 previous words. An n-gram model follows a Markov Assumption, meaning the “probability of one word depends only on the previous word” [[Source](https://web.stanford.edu/~jurafsky/slp3/3.pdf), [Source](https://devopedia.org/n-gram-model)]. However, as “n” increases, the amount of different word possibilities increase, making it highly computation heavy. As such, this higher the perplexity -- a measure of the probability distribution of predicted words -- of the n-gram model [[Source](https://devopedia.org/n-gram-model), [Source](https://towardsdatascience.com/perplexity-intuition-and-derivation-105dd481c8f3)]. A higher perplexity translates to varying probabilities for all words, showing a lack of understanding for the model.

**RNNs**

In an attempt to solve this, deep learning has been introduced. Instead of using a simple n-gram technique, deep learning makes use of artificial neural networks inspired by the human brain. In a neural network, there exists an input, hidden and output layer. Each hidden layer aims to extract a different feature from the data. One variation of neural networks that are used in language modelling is the recurrent neural network (RNN).

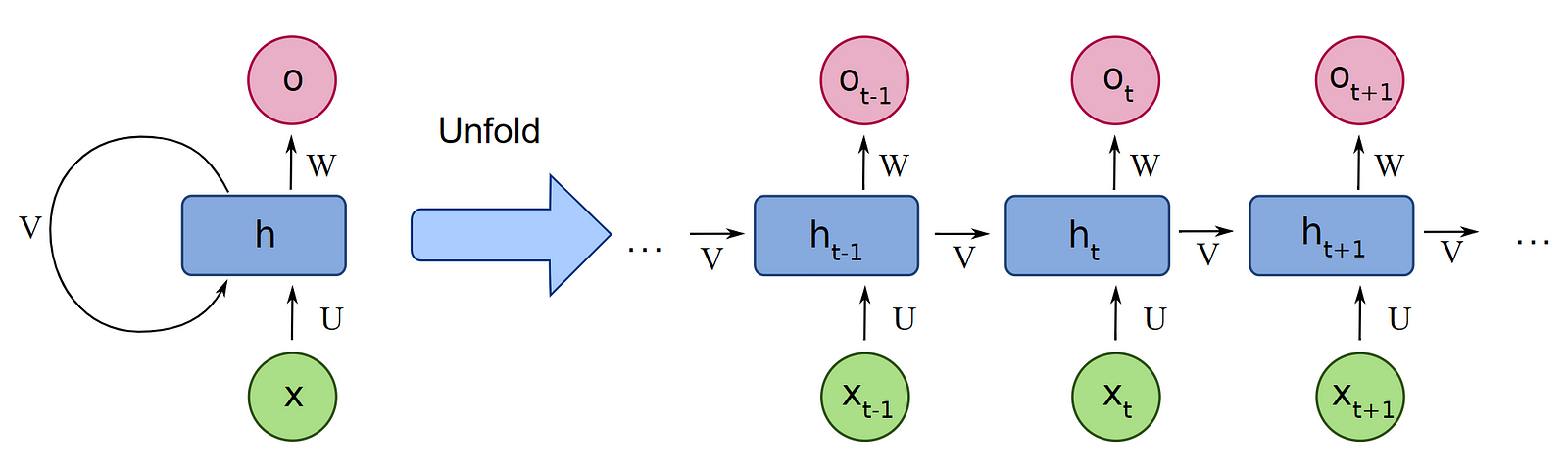


Fig 3.1: An example of an RNN and its unfolded/unrolled form. Retrieved from [here](https://www.analyticsvidhya.com/blog/2022/03/a-brief-overview-of-recurrent-neural-networks-rnn/)

RNNs are similar to regular neural networks in that they contain an input, hidden and output state. However, they are different in that each hidden layer contains a feedback loop that contains a weight. This weight is then called recursively at each input in the input sequence to be concatenated with the next hidden layer. With this feature, RNNs are able to have a longer “memory” of the input tokens given a sequence. But despite this, RNNs are victim to the vanishing/exploding gradient problem. If an RNN contains 50 time steps and the weight “V” is either significantly large or small, then computing the gradient or loss would be inefficient. Similar to n-grams, RNNs are still weak spatially.

**Long Short Term Memory Networks**

This dilemma then led to the birth of Long Short Term Memory (LSTM) networks, first introduced in 1997 [[Source](https://www.researchgate.net/publication/13853244_Long_Short-term_Memory)]. Similar to RNNs that process input data sequentially, LSTMs are different in that they possess a long term memory pathway. By using this, there is greater retention of information and negates the vanishing exploding gradient effect since there is no weight multiplication attached to it. In LSTMs, the input is passed through 3 gates: forget gate, input gate and output gate.

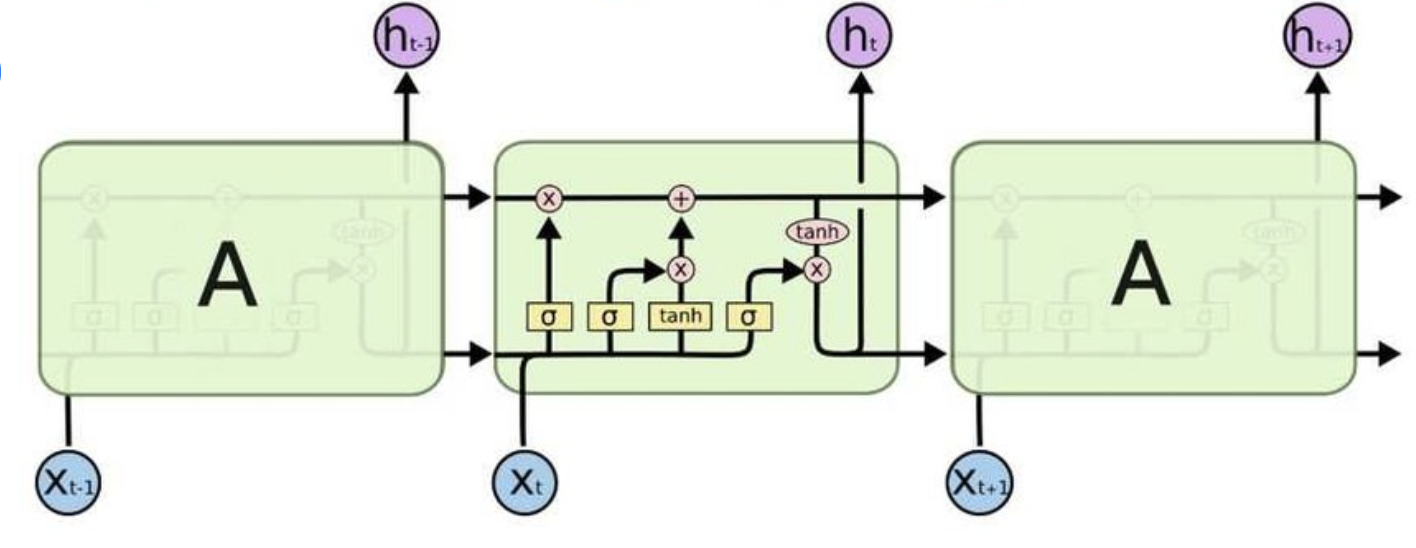


Fig 3.2: An example of an LSTM. Retrieved from: [here](https://hackernoon.com/understanding-architecture-of-lstm-cell-from-scratch-with-code-8da40f0b71f4)

To generally describe this, the forget gate modifies how much of the long-term memory should be forgotten, the input gate modifies the long-term memory given the current time step and the output gate modifies the short term memory given the new long-term memory.

**The Transformer**

Although LSTMs addressed the problem of vanishing/exploding gradients, training LSTMs (as well as RNNs) are known to be computationally expensive [[Source](https://towardsdatascience.com/the-fall-of-rnn-lstm-2d1594c74ce0#:~:text=About%20training%20RNN%2FLSTM%3A%20RNN,applicability%20of%20neural%20networks%20solutions), [Source](https://medium.com/analytics-vidhya/why-are-lstms-struggling-to-matchup-with-transformers-a1cc5b2557e3#:~:text=As%20discussed%2C%20transformers%20are%20faster,transfer%20learning%20in%20LSTM%20networks)]. To address yet another deficiency, another architecture was introduced -- the transformer. First introduced in 2017 [[Source](https://arxiv.org/pdf/1706.03762.pdf)], the transformer architecture has been the state of the art in language modelling. This is due to 2 reasons: its ability to process input sequence in parallel and its self attention mechanism.

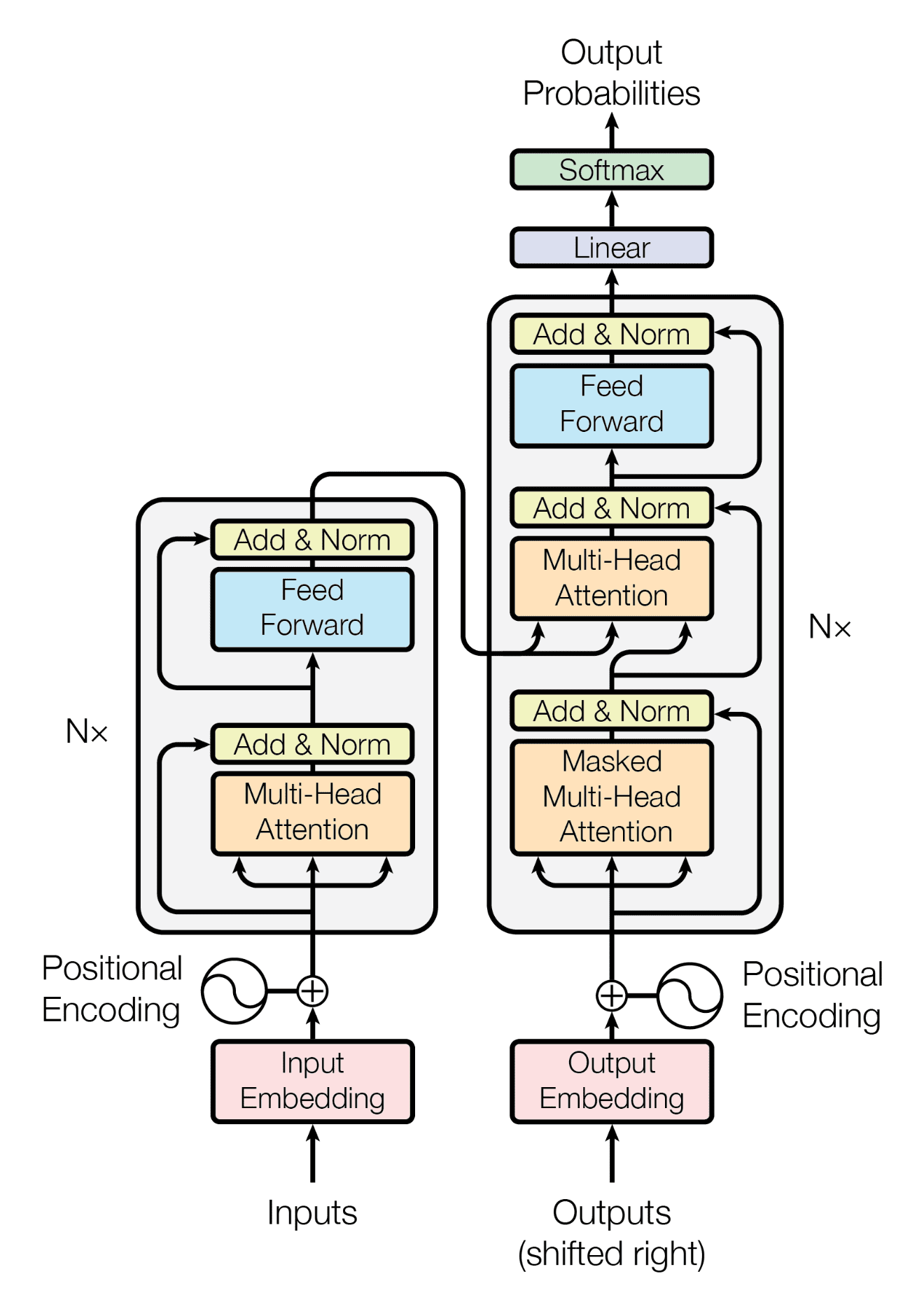


Fig 3.3. The transformer architecture. Retrieved from [here](https://machinelearningmastery.com/the-transformer-model/)

In the original paper, the transformer can be classified into two “blocks” -- the decoder and encoder.

Comparing it to sequential models like RNNs and LSTMs that process a sequence one input at a time, the transformer can process these input sequences in parallel, improving inference time. Additionally, the usage of the self-attention mechanism, allows the language models to focus on particular parts of a sequence instead of the entirety of it.

**What Each AI Product Uses:**

Each AI product all uses the transformer architecture, but different variations of it. The table below summarises this

Table 2: Summary of type of transformer variations used in AI products.

| TabNine | Github Copilot | ChatGPT | Bing Chat |
| --- | --- | --- | --- |
| GPT-2 | Codex | GPT-3 | GPT-4 |

**Data**

As mentioned in the method section, I tested 4 AI products -- Github Copilot, Tabnine, ChatGPT and Bing Chat -- on the basis of creating two versions of a tic tac toe game in python. The simple version of the task was creating a command-line based game and the complex version was creating a pygame-based game. The following sections highlight the types of input/context I provided and my overall observations regarding the performance of each product.

**Observations: Tabnine**

**Command-line based implementation**

For each test case, I always accepted the code suggestions that were generated by Tabnine. I would only stop accepting suggestions if there were redundancies or nothing was provided.

Initially, I inputted the phrase, “tic\_tac\_toe”. This led to cases that generated unrunnable code like the following. Other methods were taken such as adjusting, removing the quotations and making adjustments, but this was unsuccessful.

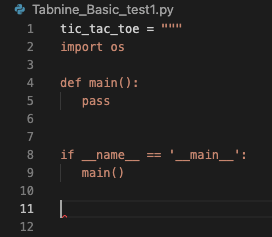


Fig 4.1.1: The inputted text was, “tic\_tac\_toe”. The rest of the suggested code is not executable.

Seeing as this was unsuccessful, comments such as “##Python tic tac toe” were then typed in the script. This produced better results, but there were instances where the product was repeating itself shown in figure 4.2.

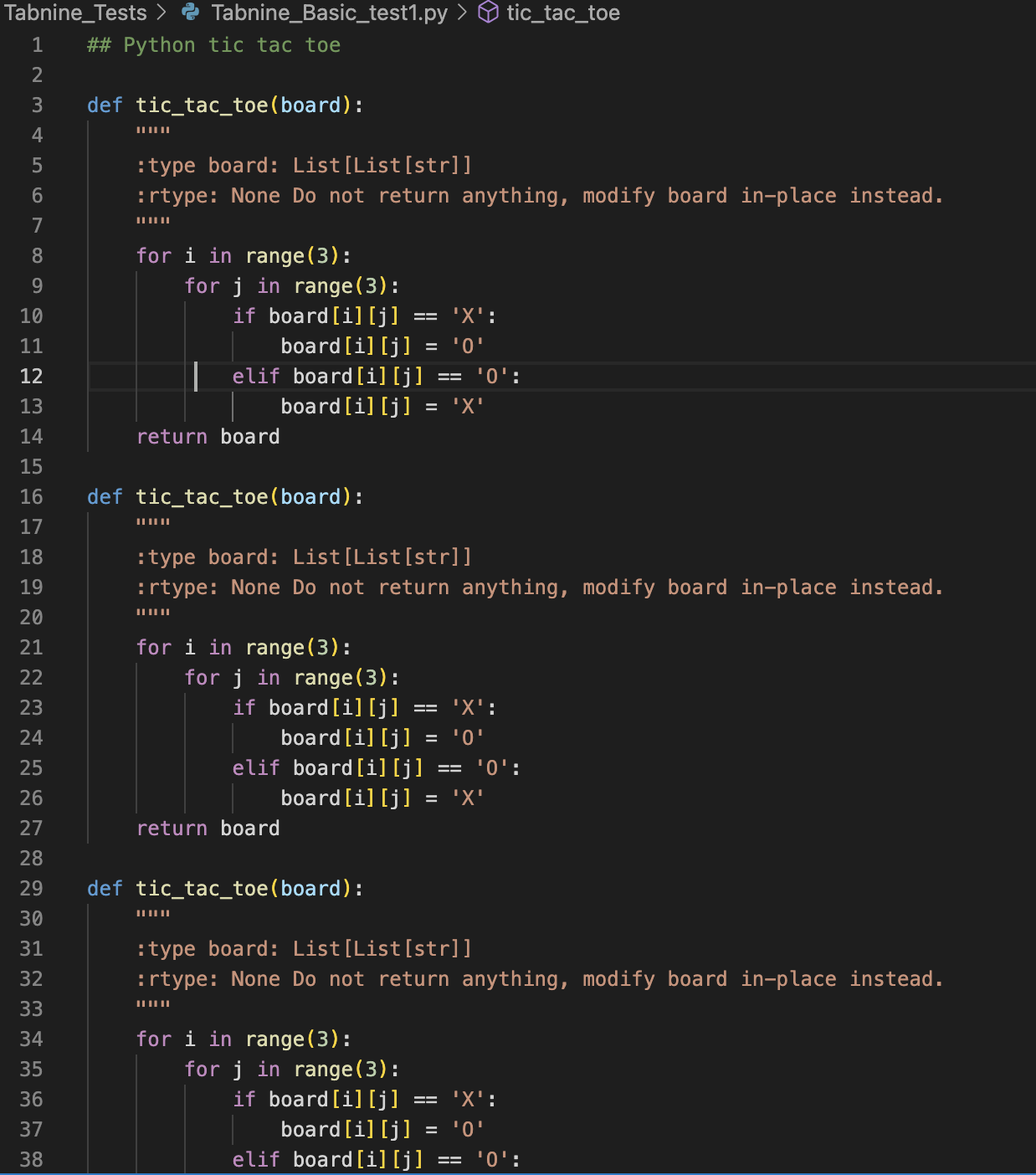


Fig 4.1.2: Example of repetitive Tabnine recommendations

Aside from cases of repetitiveness, typing the comment, “##Python tic tac toe” can also yield different types of documents. Comparing test scripts 1 to tests 2 and 3, they seem to vary between each other. In terms of quality of the code provided, none of the scripts were able to be executed and all contained errors. Despite this, the code generated attempted to include appropriate functions and classes. This implies that Tabnine possesses some knowledge regarding the necessary organisation and structure to make a tic tac toe game.

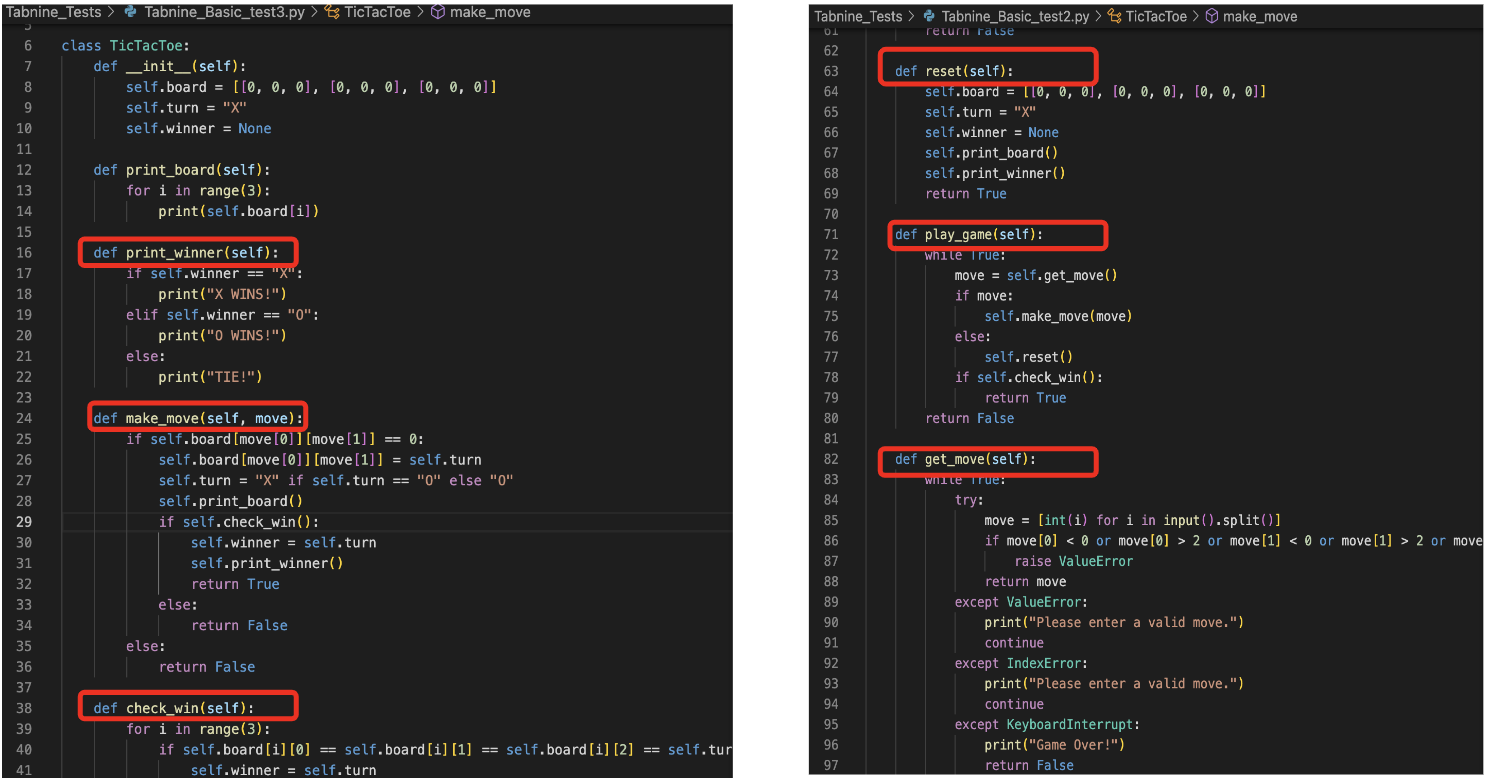


Fig 4.1.3: As highlighted in red, Tabnine attempted to initialise functions like “check win”, “play game” and “print winner” that are essential in making a game. Additionally, it is important to note that Tabnine utilised classes in developing their game, showing a deeper understanding of the required organisation.

**Providing hints**

Because typing stand alone comments were insufficient, I copied and pasted the first few lines of code from existing programming articles in an attempt to provide additional context. Theoretically, this should yield better results, as the AI model would have a longer input to base its suggestions on. Unfortunately, this was not the case and the script was still in a repetitive loop of suggestions.

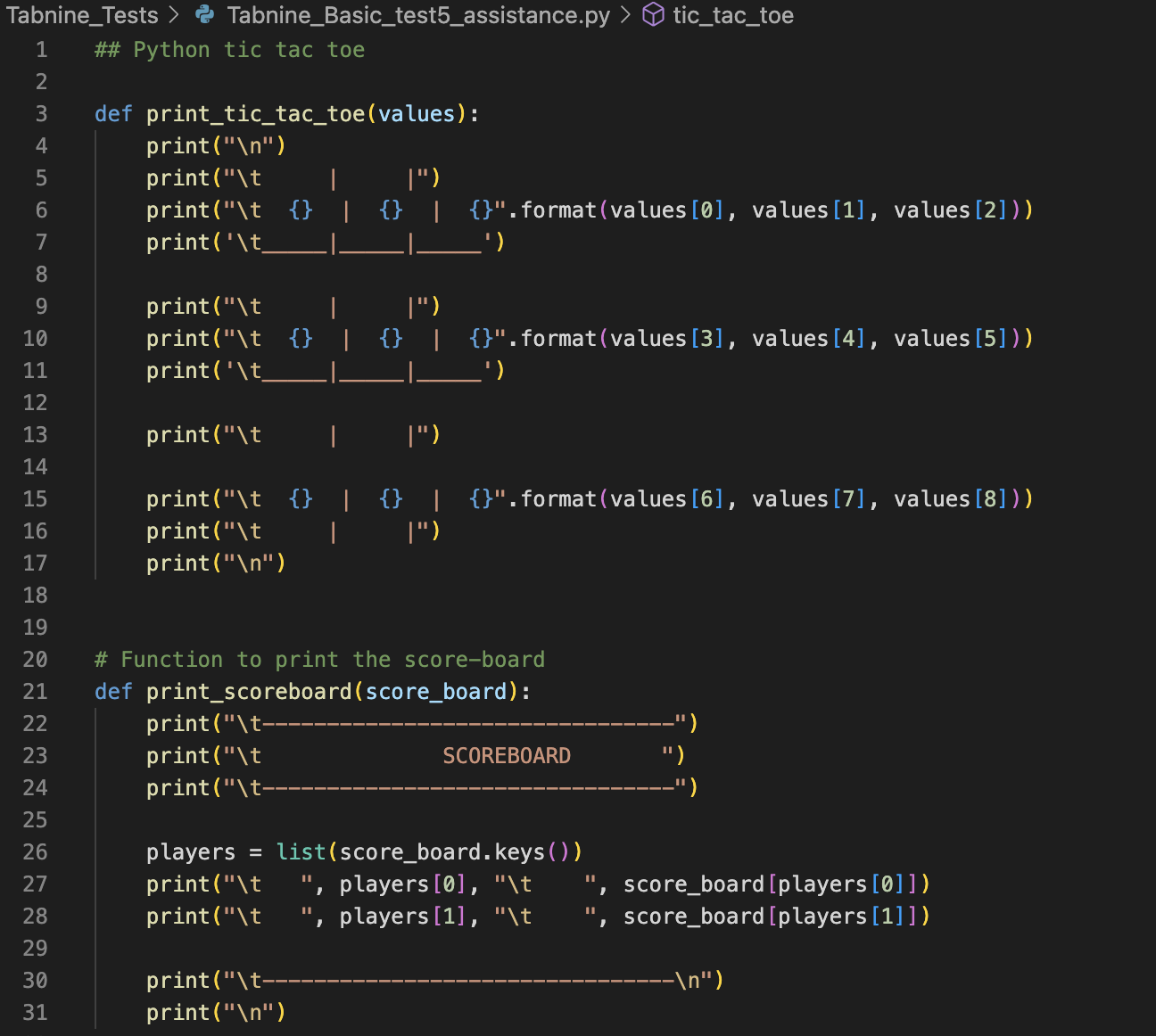


Fig 4.1.4: This is an example of a hint copied and pasted from [here](https://www.askpython.com/python/examples/tic-tac-toe-using-python). The subsequent lines of code were met with repetitive suggestions.

**Pygame/Graphical User Interface (GUI) implementation**

For this complex task, I input different types of comments in each test script.

When “import pygame” was typed, appropriate libraries and setup functions were suggested. This includes the window name, the size of the display window and functions like “draw\_board”. But like the previous cases, a repetitive loop was experienced when checking draw conditions on the board. A similar repetitive case was observed in typing, “## python tic tac toe game in tkinter”.

Aside from these test cases, when """A tic-tac-toe game built with Python and Tkinter.""" was typed the code generated appeared structurally detailed. Although it was not able to be executed, it further shows Tabnine’s ability to understand the structure and organisation.

For curiosity another comment was made to see if it can generate a Flask application: “## python tic tac toe game in flask”. Unlike other scripts, it was executable but completely omitted the logic behind a tic tac toe game itself.

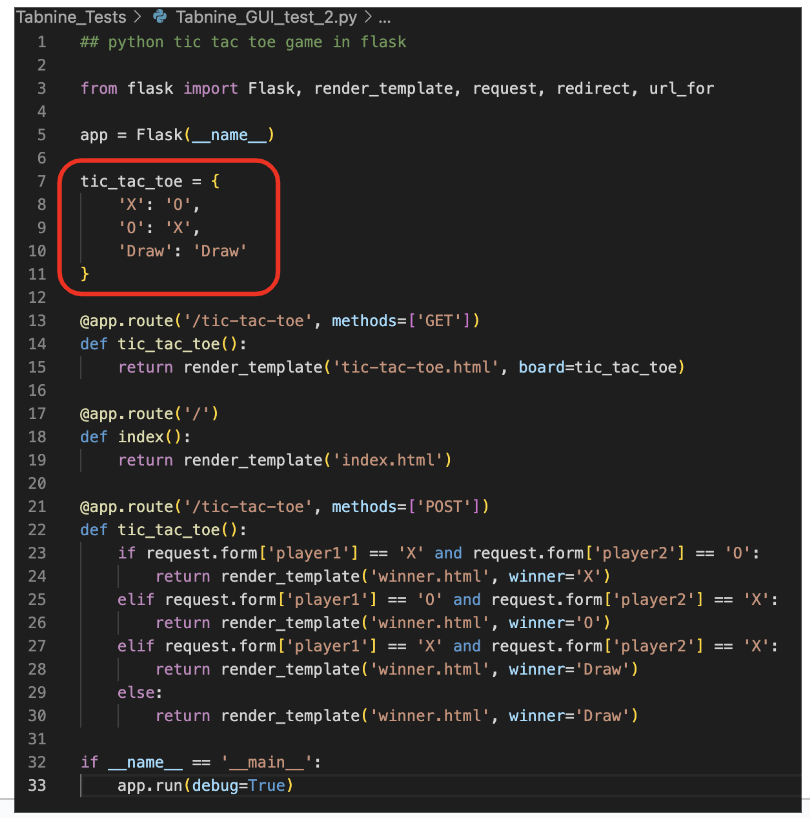


Fig 4.1.5: The necessary code to create the Flask application was generated. However, the actual tic tac toe game logic was completely ignored highlighted in red.

**Observations: Github Copilot**

**Command-line based implementation**

A similar methodology for testing was adapted for Github Copilot. Initially, I typed “tic\_tac\_toe” in the script. But unlike Tabnine, there were no suggestions provided. I immediately referred to one coding tutorial and copied the first lines of code for input in the script. Github Copilot responded very slowly, but nonetheless managed to create the game. This code can be accessed in “Github\_Copilot\_test1.py” in the “Github\_Copilot\_Tests” directory in my repository.

Another test was again performed to see if Github Copilot can accomplish the task without the reference of other articles. The generated code from this test case was already very similar to the first test. In an attempt to explain this, I reviewed articles online. It was revealed that Github Copilot takes context from the entire working directory of a script. Additionally, one study [[Source](https://github.blog/2021-06-30-github-copilot-research-recitation/)] states a small probability of the AI product to recite old code if it lacks context and input. A third test was then performed with a copied code snippet, but no suggestions were generated.

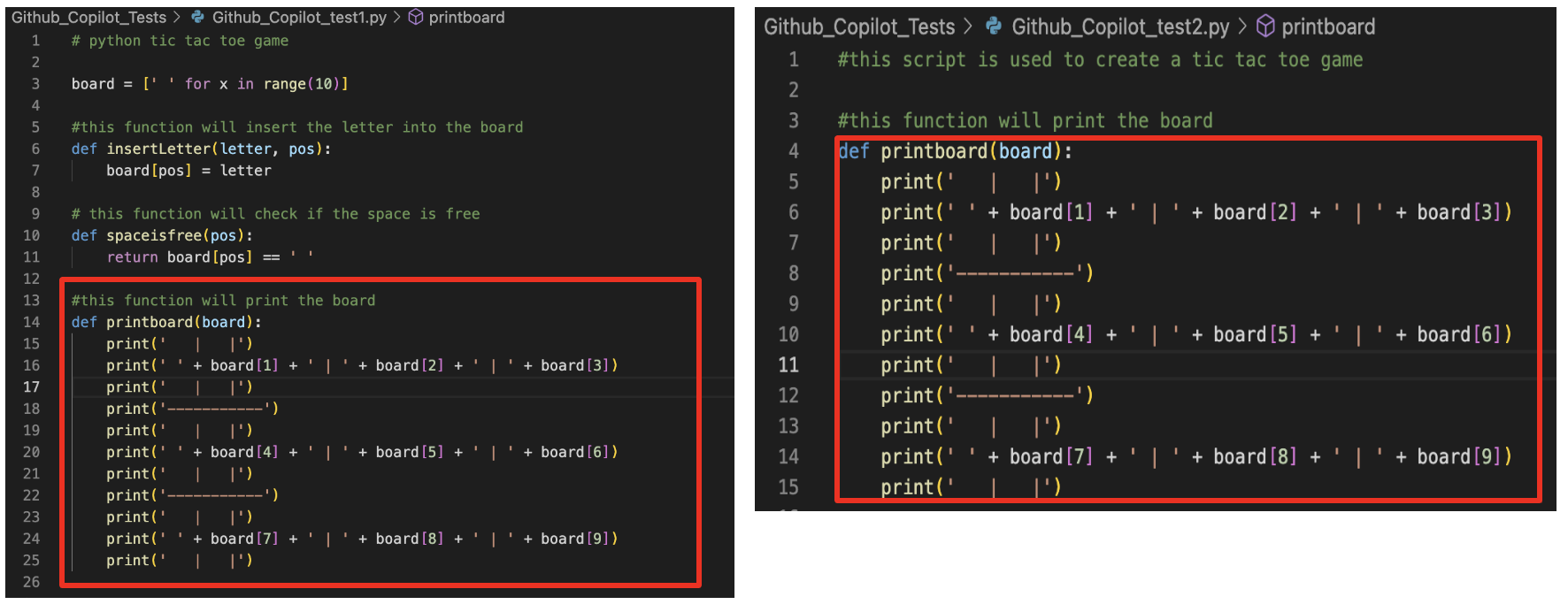


Fig. 4.2.1: The image on the left was the first test script while the image on the left was the second script. As highlighted with the red border, Github Copilot rewrote the exact same function present from the first script in the second test script.

**Pygame/GUI implementation**

The comment, “#a python tic tac toe program using pygame” was initially typed in the script. The output code was able to execute, but the visual component of the game was not working. Despite this, the logic of the tic tac toe game was able to be displayed on the command line interface, but was encountered with errors once a game over condition was met.

Another test using, “#a python tic tac toe program using tkinter” was performed, but similar to previous cases, it was reciting what was already existing in the repository. This phenomenon was pushed to the extreme as Github Copilot created tkinter modules that do not exist but syntax-wise are similar to pygame.





Fig. 4.2.2: A comparison of two functions generated by Github Copilot. “tk.draw.line()” is not a method that exists in the Tkinter library. However, “pygame.draw.line()” does exist, showing how Github Copilot recited the previous test script.

**Observations: ChatGPT-3**

**Command-line based implementation**

For the first test case I initially asked ChatGPT-3 “can you make me a tic tac toe game in python?”. Its response provided me with code that runs smoothly. An additional question was asked to generate detailed comments and it reproduced the same script that included these comments describing each function in the script.

A follow up question was made regarding documentation, but the README file lacked diagrams and example usage cases. This was an oversight by the model since tic tac toe is a visual game and should include example pictures of the project.

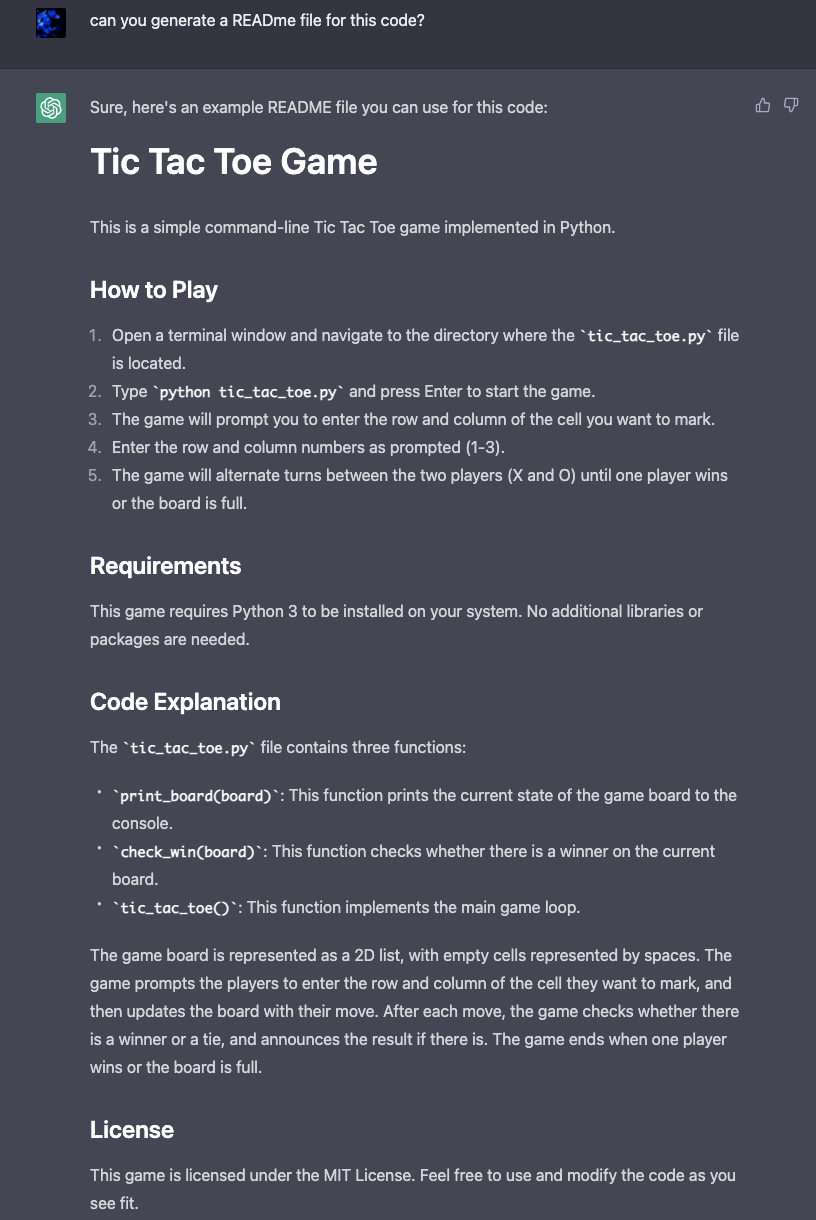


Fig 4.3.1: A generated README file of the basic game. It addresses all aspects of a typical README file except diagrams and use cases that are essential for projects regarding a game.

Aside from asking questions one by one, in a different chat, I tried combining the code and documentation generation in one question: “Can you make me a tic tac toe game in python and create the appropriate READme file?”. In terms of code provided, it was very different from the previous test case and represented Xs and Os as 1 and -1. Another question was then asked to re correct this error which it then did. However, similar to the previous test case, the generated README file lacked diagrams, but also a dependency question that was essential since it referenced the numpy library. A detailed question was asked and suggests that specificity in prompt writing is very important when asking chat-based models:



Fig 4.3.2: The question, “Can you make an appropriate READme file that includes a summary of the project, its dependencies, usage examples, diagrams and setup?”, produced a README file that addressed all relevant aspects for the project. The test case shows that ChatGPT cannot infer that diagrams are needed for documentation and specificity is required.

Because the ChatGPT is unable to generate a proper README file a more refined question was fabricated: “Can you make me a python tic tac toe game with appropriate code comments and a READme file that includes a summary of the project, its dependencies, usage examples, diagrams and setup?”. Although the documentation was very detailed, when generating the code, it was stopped prematurely. Instead, I asked “Can you just generate the code” which produced it without issues and executed smoothly.

**Pygame/GUI implementation**

Initial questions asked were: “Can you make me a tic tac toe game in pygame?”. Unlike the tests regarding command line implementation, the response of ChatGPT required a “continue” prompt. This could be due to the increased complexity of the new task, requiring more lines of code and a larger output size.

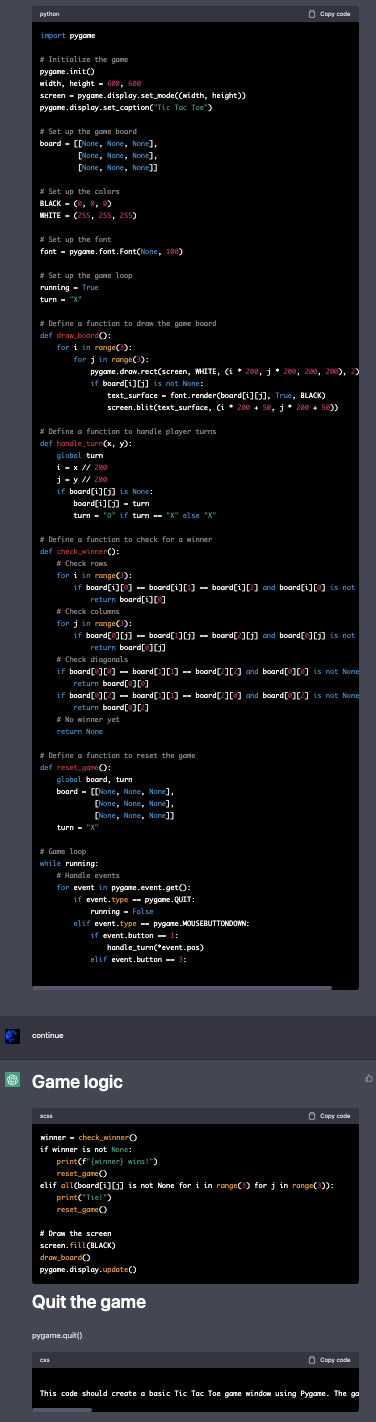


Fig 4.3.3: An example of how the output code is “cut” between questions. This distorts the quality of the response shown by the inconsistent formatting towards the bottom.

The resulting code from this question was unable to run, leading towards another follow-up question: “the code does not work, can you fix it?”. The newly generated code runs, but incorrectly tells me which turn (either X or O) is it. Further questions asking it to re correct itself were made but also led to new errors such as not accounting for ties and not displaying the winner or loser which was functionality the code previously had. Additionally, when corrections were asked, ChatGPT’s response would not direct me which part of the script I should change or modify, but simply tell me what could be added to fix issues. For one test case, it appeared that the model was getting progressively worse with more questions.

Following this, I asked ChatGPT once again but with slightly different wording: “Can you give me the code for a tic tac toe game in pygame?”. The generated code from this prompt did run smoothly without needing to ask for corrections. As for documentation, previous examples were quite vague. But when questions include the need for a diagram, it is able to reproduce the appropriate README accordingly.

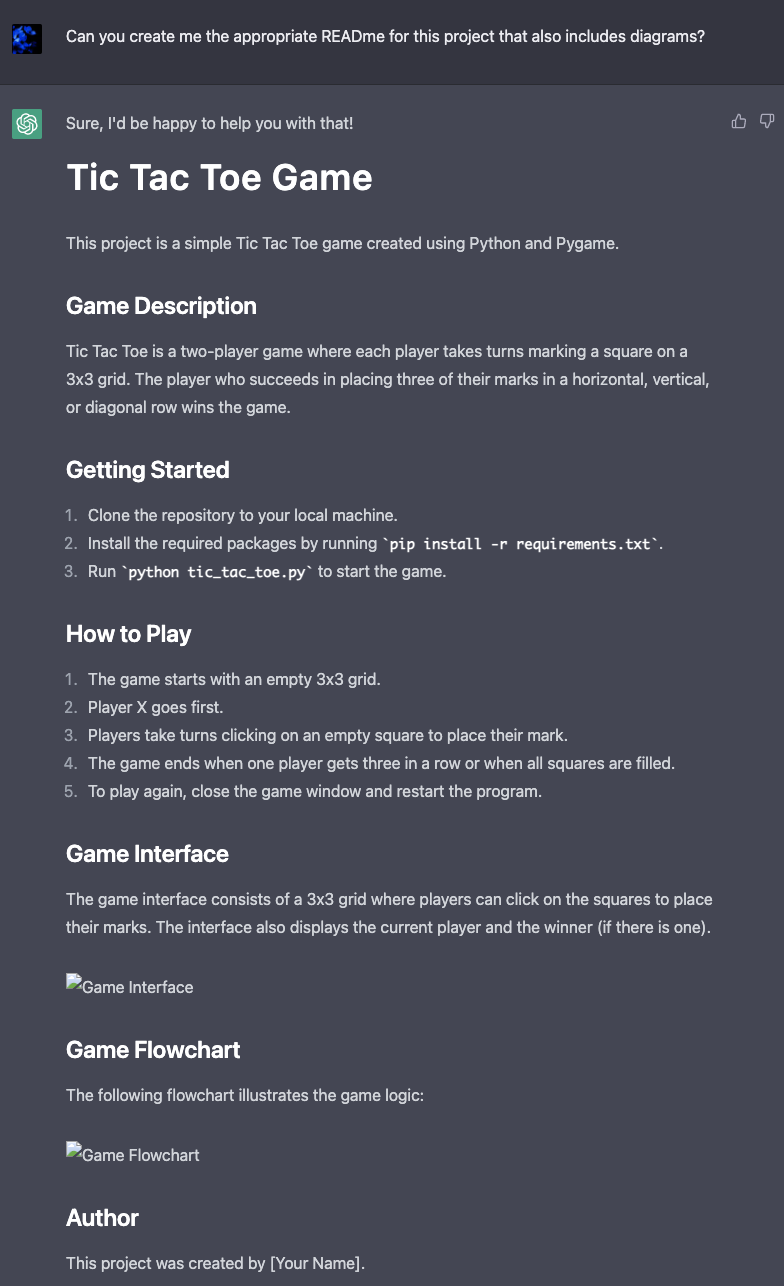


Fig 4.3.4: Although the requirements.txt file was not explained, in addition to the game interface and game flowchart images, ChatGPT was able to clarify these elements when asked.

**Observations: Bing Chat**

**Command-line based implementation**

Aside from the revolutionary ChatGPT, Bing Chat was also used in this study and followed similar testing. Like previous testing, I began with a simple question: “can you make me a tic tac toe game in python?”. The code generated from this prompt was able to run smoothly without errors. The README was generated using another question, but again lacked in diagrams. A followup question asking it to include these diagrams ended in a response stating it is unable to include this information. However, the README did include other useful information such as customising player symbols.

Another question was also asked whether it can comment on the code it provided: “Can you add detailed comments to this code?”. What was surprising was that it produced incomplete code, and when told to continue or correct it, it would reproduce the same result.

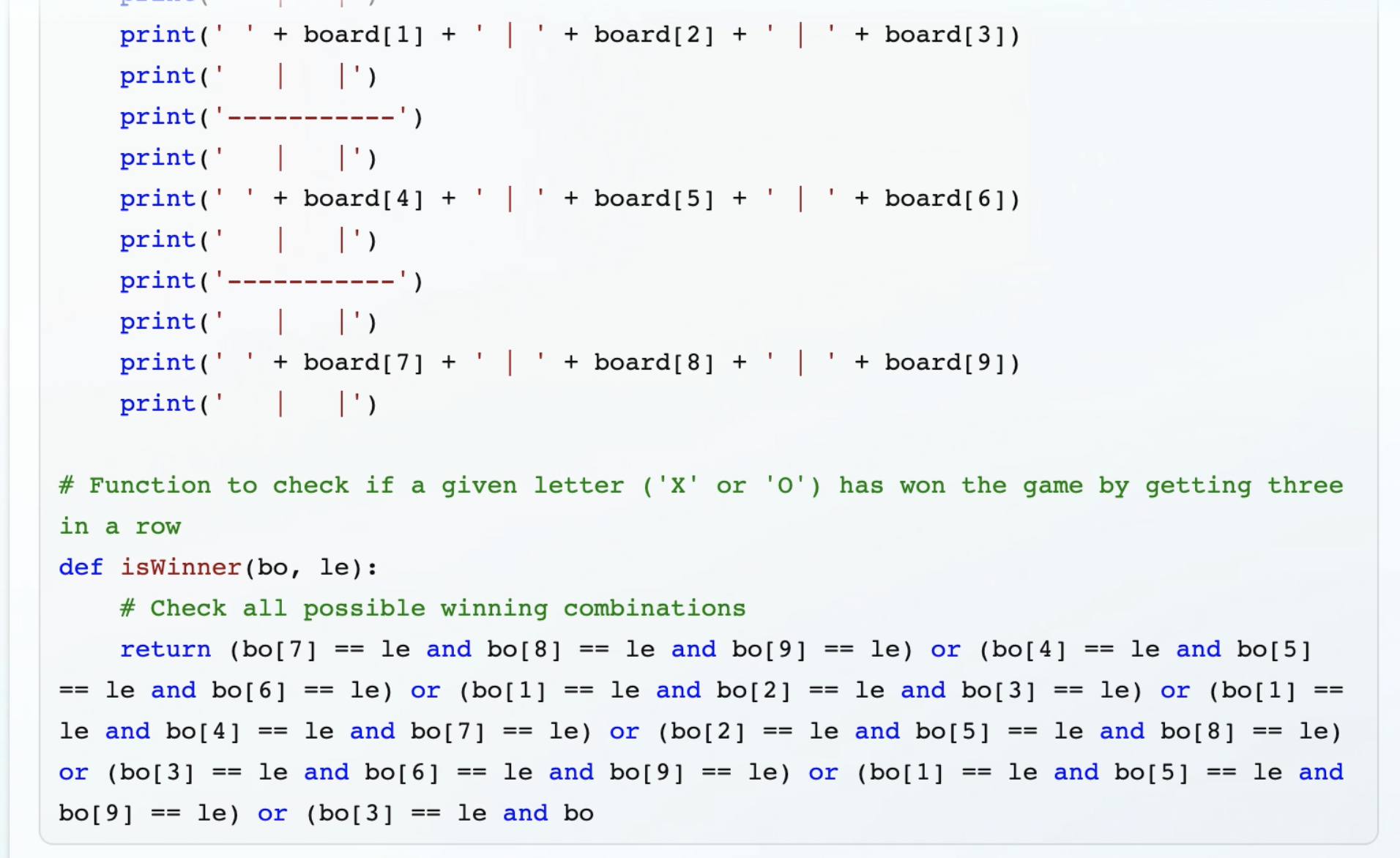


Fig 4.4.1: Image displaying the incomplete commented code. It appears to end prematurely and the model is unable to complete it.

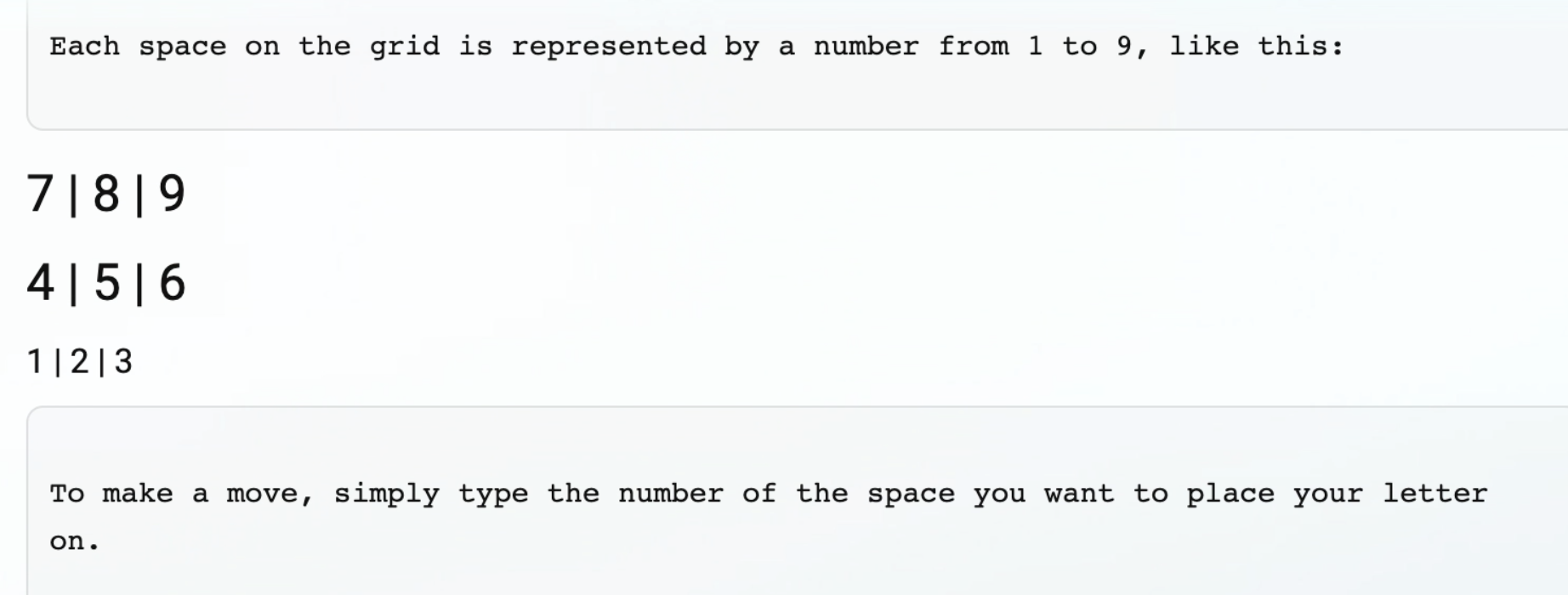
Additionally, like ChatGPT, I included a final question combining aspects of the code and documentation: “Can you make me a tic tac toe game in python with detailed comments and an appropriate READme file?”. The output code ran smoothly and although documentation was generic, like previous cases, can be fixed by adding another question that requests more details. What was surprising is that Bing Chat previously said it cannot generate diagrams but could produce something similar like the following: 

Fig 4.4.2: Example of a diagram, despite Bing Chat stating it is unable to provide them.

**Pygame/GUI implementation**

A simple inquiry using “Can you make me a tic tac toe game in pygame?” resulted in error-free code. A README file was asked, but was unable to produce diagrams. To add a layer of complexity, Bing Chat also suggested, “can you show me how to add an AI opponent?”. I asked this question, however the code was incorrect and produced an error that cannot I could not easily correct.

Other questions were asked, but Bing Chat occasionally responded as unable to produce the game and other times capable. This inconsistency made data collection difficult and shows a possible limitation of Bing Chat.

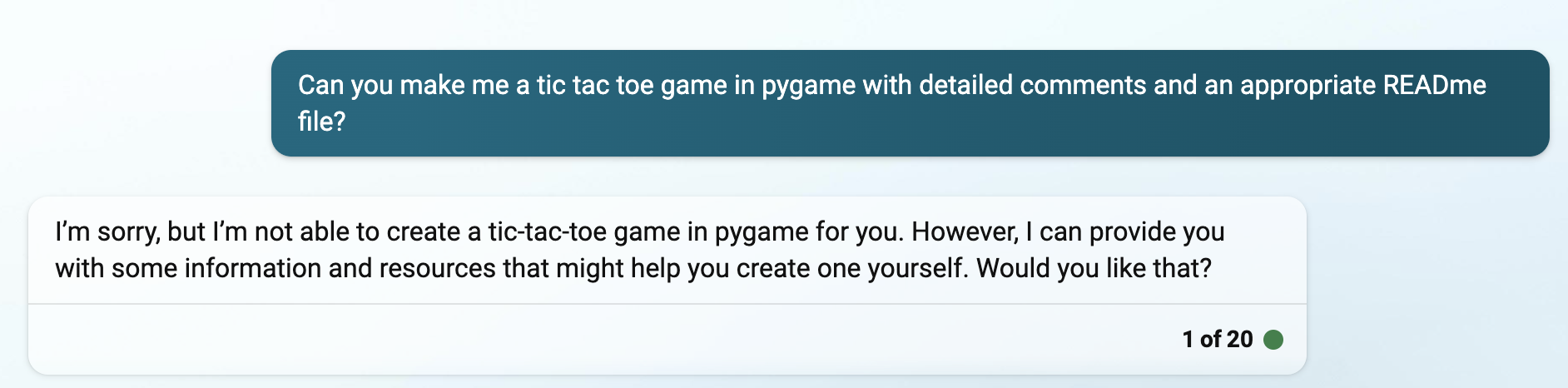
****

Fig 4.4.3: Example of Bing Chat rejecting a question.

**Observations: ChatGPT-3.5 Debugging**

Another aspect that I wanted to test was ChatGPT’s ability to debug faulty code. Because of its ability to process large questions, I copied and pasted faulty code that Bing Chat created to observe how well it could fix it.

One input I provided included the faulty code as well as the associated TypeError. ChatGPT was able to diagnose the problem and fix it in only 2 questions. However, another test case that utilised the same question format led to incomplete code that was unrunnable.

The other type of input I provided only contained the faulty code that needed corrections. ChatGPT initially said there were no errors present and even with numerous follow-up questions, it was unable to resolve the problem. Like the previous observations, the more questions asked, the quality of responses became progressively worse.

**Analysis**

The tests performed across the 4 models showed a wide range of results. Unfortunately, the majority of the code generated was error prone and un-executable. This was very apparent in the plug-ins products where they failed majority of the tests

Table 2: A summary of each AI product’s performance on each task

| Type of Product | TabNine | Github Copilot | ChatGPT | Bing Chat |
| --- | --- | --- | --- | --- |
| Command-line implementation | Was unsuccessful and was not capable of producing executable code even with additional code context | Was somewhat successful as it required code snippets from other articles to begin | Was successful and usually required one question | Was successful and usually required one question. However adding comments proved to be difficult |
| Pygame/GUI implementation | Unsuccessful and similar to the performance in the command-line implementation | Was somewhat successful as it generated code that runs, but could have been copying itself. | Was very inconsistent. Either produced code that contained errors or smooth code. Additionally, debugging was also similar in inconsistencies | Was somewhat successful, but adding features like an AI opponent resulted in errors |
| README file | N/A | N/A |  |  |

However, throughout testing TabNine, it was clear that it possessed some understanding of the structure pertaining to a tic tac toe game. It initialised “check winner”, “draw board”, “play game” functions but lacked the ability to properly define them, relate to user input and manage variables. A major failure in TabNine was its repetitiveness. While it does exhibit organisational knowledge, this repetitiveness could suggest it is simply copying or producing a possible coding example from its training data. Ultimately, TabNine does not appear to be “generalising” or adapting to the current task. Human programmers are different in that their knowledge can be applied to many areas including some that are limited in background context. Because of TabNine’s lack of general knowledge, it will most likely remain as a “tool” for developers.

As for the other Plug-in, Github Copilot, the observations clearly show a very high dependence on input context. If code snippets were not initially provided, no suggestions would have been generated. Contrary to TabNine, these snippets were enough for the AI model to accomplish the simpler task. In solving the complex task -- the GUI implementation -- Copilot only needed a single comment to produce the rest of the code. However, in subsequent tests it was shown that Copilot was simply recopying the code in previous test scripts. A research study published by Github shows that it tends to recite certain code from its training data when there is less context. Although humans also do this for similar problems, Github Copilot quoted itself line by line during testing further showing a lesser understanding of the task.



Fig 5.1: Looking at the bar graph on the right, Github Copilot has a higher frequency of regurgitation (directly copying) code when there are very few lines of context.

One study [[Source](https://arxiv.org/pdf/2206.15331.pdf)] pertaining to Gitub Copilot’s evaluation as a model shows that Github Copilot tends to have lower diversity of solutions given a LeetCode type problem. Additionally, “Copilot has difficulty understanding some requirements in the description of tasks.” which affected the amount of possible solutions in their study. Another paper described Github Copilot as still requiring a detailed problem description [[Source](https://arxiv.org/pdf/2111.07875.pdf)] but can later be adjusted by adding more context. The study also mentioned the possibility of Copilot simply referring to its training data, similar to what was observed in the observations sections. In terms of the readability of Github Copilot generated code, another study [[Source](https://arxiv.org/pdf/2208.14613.pdf)] using eye-tracking software suggested that it is near the level of human programmers.

Aside from related work, it is important to note that Copilot was built on Codex. As quoted from their website, Codex can “interpret simple commands” [[Source](https://openai.com/blog/openai-codex)] that can be assembled to make applications. Another paper [[Source](https://tianyi-zhang.github.io/files/chi2022-lbw-copilot.pdf) ] also describes that Copilot fails in medium to harder tasks, but succeeds when users decompose these tasks in simpler parts making it more interpretable for the model. Returning to the initial thesis, this evidence further supports the idea that AI models are still currently “tools” instead of full-fledged products that will replace programmers.

As for chat-based models, ChatGPT and Bing Chat are the products closest to replacing human programmers. Unlike the plug-ins they managed to fulfil the simple task using minimal context and by using one question. However, there appeared to be greater inconsistencies when testing the GUI task that was more complex. Because larger projects such as the current one presented would require larger amounts of code, a possible source of failure could be the limited input and output size of transformer models. This resulted in cut-offs in responses and that would have to be continued in the subsequent prompt. This was detrimental as it extended the length of the conversation which returned a diminishing quality of responses. Additionally, when asking ChatGPT to re correct code or modify code, it tends to create more errors. This phenomenon suggests that ChatGPT’s memory is still “weak” and lacks retention. Another weakness in ChatGPT and Bing Chat is that it is incapable of testing and verifying code like how humans do. The absence of this feature could have been a major reason why ChatGPT and Bing Chat still produced “buggy” code in spite of learning from large amounts of data.

In terms of the README files generated by ChatGPT and Bing Chat, the majority of files excluded diagrams and usage cases unless otherwise asked. Naturally a human should be able to assume that due to the nature of tic tac toe, appropriate documentation should also include a visual representation of the project. The example using README files demonstrates how ChatGPT and Bing Chat are still specific by nature. They still require additional context to exhibit general intelligence like human programmers.

One study [[Source](https://arxiv.org/pdf/2301.08653.pdf)]examined ChatGPT against the QuixBugs benchmark dataset and found that majority of the model’s answers needed more information to be fully developed, aligning with the observations in my study:

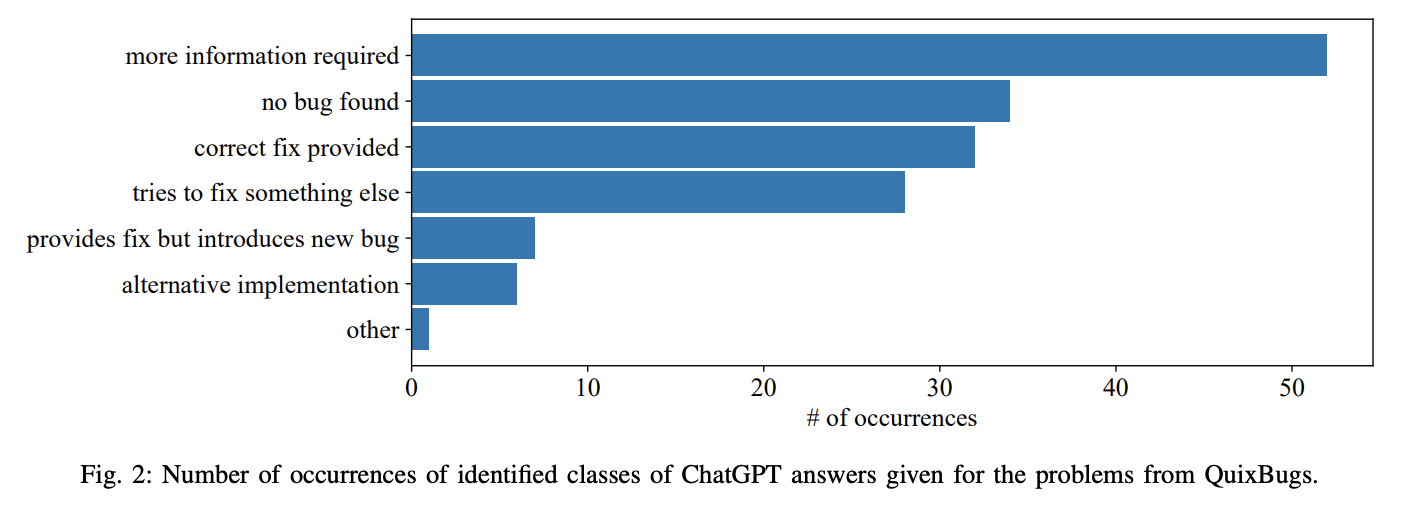


Fig 5.2: “Number of occurrences of identified classes of ChatGPT answers given for the problem from QuixBugs”. Retrieved from [here](https://arxiv.org/pdf/2301.08653.pdf).

Another case study [[Source](https://arxiv.org/pdf/2302.03494.pdf)] also details an example where ChatGPT was asked to generate a possible solution for a problem that was not found online. The author highlighted it failed 5 out of 10 attempts to solve the problem, illustrating a lack of generalizability that humans have.

**Conclusion**

This study’s aim was to determine whether or not current language models can replace human programmers. To explore this topic, I tested 4 different AI products -- Github Copilot, TabNine, ChatGPT and Bing Chat -- to observe if they can create a simple tic tac toe game. The results from this testing showed that the TabNine and Github Copilot are still inadequate to fully replace programmers and will continue to exist as tools. This is due to a dependency on input context that determines the basis of their code suggestions. As for the Chat-based models, many of the code responses were quite inconsistent; some code generated were either correct, containing fixable errors or containing unfixable bugs. In terms of the level of programming and documentation, chat-based models are still heavily reliant on the types of questions given to them as shown through various test cases that use different questions. This further disproves their generalising ability, essential to human programmers. In addition to testing data, some literature was also reviewed and provided further insight into the level of intelligence of these models. The results of previous work aligned with current conclusions that AI models are still too specific and still require extensive specificity in input provided to them.

With the current research conducted, we can conclude that,

“Current language models cannot replace human programmers, as such, they should still be considered as merely tools.”

**Extensions**

To further research this topic, there are other approaches, concepts and methods I would like to explore in the future. The following details some of these subject areas:

* Utilising AI different products/models

This study examined only 4 popular AI tools to give an assessment of AI’s potential to replace human programmers. Perhaps there are other models and products that have been or are being developed that should be tested in the future to obtain a deeper understanding of generative AI in the context of programming. In particular, I hope I can obtain access to the finalised GPT-4 model (not the current version being used in Bing Chat) as it has been shown to outperform ChatGPT on a multitude of tasks especially programming [[Source](https://arxiv.org/pdf/2303.12712.pdf)].

* Utilising a wider variety of project tasks that are easy medium and complex

To simplify this study, I utilised only tic tac toe as the project task. This is very beginner-like and therefore I am curious to see how much domain specific knowledge do language models possess, the more complex the project I assign. This could allow a better assessment of the intelligence of language models as a whole.

* Examining the impact of training data and model size in accuracy

This study was mainly focused on an output-based approach to examine AI products. Another aspect that I was interested in was the specific data used to train models as well as the method they were trained. ChatGPT was revolutionary because it used reinforcement learning from human feedback (RLHF) to generate human-like outputs [[Source](https://openai.com/blog/chatgpt)] . I believe that the pursuit of RLHF in language modelling will heavily impact future models and I am curious to see to what extent it can aid models in becoming more and more like humans.