TYPHON: AUTOMATICALLY RECOMMENDING RELEVANT CODE CELLS IN JUPYTER NOTEBOOKS

ไทฟ่อน ระบบแนะนำโค้ดที่เกี่ยวข้องแบบอัตโนมัติในจูปีเตอร์โน๊ตบุ๊ค

BY

MR. VEERAKIT PRASERTPOL 6288035
MR. NATANON RITTA 6288037
MR. PAPHON SAE-WONG 6288118

ADVISOR
DR. CHAIYONG RAGKHITWETSAGUL

CO-ADVISOR
ASST. PROF. THANWADEE SUNETNANTA

A Senior Project Submitted in Partial Fullfillment of the Requirement for

THE DEGREE OF BACHELOR OF SCIENCE (INFORMATION AND COMMUNICATION TECHNOLOGY)

Faculty of Information and Communication Technology
Mahidol University

2022

ACKNOWLEDGEMENTS

We would like to devote this part to presenting our utmost gratitude to all people who have been involved through all the processes from the start until the current status of this project. Firstly, we would like to give our appreciation to the most important person, Dr. Chaiwong Ragkhitwetsagul, our senior project advisor and Asst. Prof. Dr. Thanwadee Sunetnanta, our co-advisor. Furthermore, we would like to thank our honorable collaboration with advisors at the Software Engineering lab, Nara Institute of Science and Technology, Assist. Prof. Dr. Raula Gaikovina Kula, Assist. Prof. Dr. Bodin Chinthanet, Assoc. Prof. Dr. Takashi Ishio, and Professor Kenichi Matsumoto. This project is well-refined and completed with shared knowledge and resources from this collaboration.

Mr. Veerakit Prasertpol

Mr. Natanon Ritta

Mr. Paphon Sae-wong

Typhon: Automatically recommending relevant code cells in Jupyter notebooks

MR. VEERAKIT PRASERTPOL 6288035 ITCS/B
MR. NATANON RITTA 6288037 ITCS/B
MR. PAPHON SAE-WONG 6288118 ITCS/B

B.Sc.(INFORMATION AND COMMUNICATION TECHNOLOGY)

PROJECT ADVISOR: DR. CHAIYONG RAGKHITWETSAGUL

ABSTRACT

At present, code recommendation tools have gained greater importance to many software developers in various areas of expertise. Having code recommendation tools have enabled better productivity and performance in developing the code in software and made it easier for developers to find code examples and learn from them. GitHub Copilot is potentially one of the well-known code recommendation tools. Besides Copilot, there are also other related studies on code recommendation such as Aroma, Strathcona, Senatus, Example Overflow, Copilot, and Tabnine. However, our experiment with GitHub Copilot on Python Jupyter notebooks shows that Copilot has some limitations in recommending code cells.

Therefore, this project proposes Typhon, an approach to automatically recommend relevant code cells in Jupyter notebooks. Typhon tokenizes developers' Markdown description cells and looks for the most similar code cells from the database using text similarities such as the BM25 ranking function or CodeBERT, a machine-learning approach. Then, the algorithm computes the similarity distance between the tokenized query and Markdown cells to return the most relevant code cells to the developers. We evaluate the Typhon tool on Jupyter notebooks from Kaggle competitions and then let human experts evaluate the tool to judge their usefulness.

The final evaluation shows that the Typhon tool able to offer good and simple to use interface as promised. However, the qualitative aspect of recommendation results are still under satisfaction. Thus, the work will need further investigation in recommendation algorithm and improvement.

KEYWORDS: CODE RECOMMENDATION, JUPYTER NOTEBOOK 81page.81 P.

ไทฟ่อน ระบบแนะนำโค้ดที่เกี่ยวข้องแบบอัตโนมัติในจูปีเตอร์โน๊ตบุ๊ค

นาย วีระกิตติ์ ประเสริฐผล 6288035 ITCS/B
นาย ณฐนนท ฤทธิ์ตา 6288037 ITCS/B
นาย ปภน แซ่หว่อง 6288118 ITCS/B

วท.บ. (เทคโนโลยีสารสนเทศและการสื่อสาร)

อาจารย์ที่ปรึกษาโครงการ: ดร. ชัยยงค์ รักขิตเวชสกุล

บทคัดย่อ

ในปัจจุบัน เครื่องมือแนะนำโค้ดได้รับความสำคัญมากขึ้นสำหรับนักพัฒนาซอฟต์แวร์ในหลากหลายความเชี่ยวชาญ การมีเครื่องมือแนะนำโค้ดช่วยเพิ่มประสิทธิภาพและประสิทธิผลในการพัฒนาโค้ดในซอฟต์แวร์ และยังสามารถค้นหาตัวอย่างโค้ดและเรียนรู้จากโค้ดเหล่านั้นได้ง่ายขึ้น GitHub Copilot เป็นหนึ่งในเครื่องมือแนะนำโค้ดที่รู้จักกันเป็นอย่างดี นอกจากนั้นแล้ว ยังมีการศึกษาอื่น ๆ ที่เกี่ยวข้องกับการแนะนำโค้ด เช่น Aroma, Strathcona, Senatus, Example Overflow, Copilot, และ Tabnine. อย่างไรก็ตาม หลังการทดลอง GitHub Copilot บน Jupyter Notebook ด้วย Python เราได้เห็นว่า Copilot ยังคงมีข้อจำกัดบางอย่างในการแนะนำเซลล์โค้ด ความเป็นไปได้ที่ส่งผลให้Copilot แนะนำโค้ดได้ด้อยประสิทธิภาพอาจเกิดจากความยาวและบริบทที่หลากหลายในคำอธิบายของ Markdown ที่ใช้เป็นอินพุตสำหรับค้นหา ดังนั้นโปรเจ็กต์นี้จึงเสนอ Typon ระบบแนะนำเซลล์โค้ดที่เกี่ยวข้องโดยอัตโนมัติใน Jupyter Notebook Typhon สร้างโทเค็นให้กับเซลล์คำอธิบาย Markdown ของนักพัฒนา และค้นหาเซลล์รหัสที่คล้ายกันที่สุดจากฐานข้อมูลโดยใช้ฟังก์ชันการจัดอันดับ BM25 หรือ CodeBERT สำหรับแนวทางการเรียนรู้ด้วยเครื่อง จากนั้นอัลกอริธึมจะคำนวณระยะห่างของความคล้ายคลึงกันระหว่างคิวรีโทเค็นและเซลล์ Markdown เพื่อให้ส่งคืนเซลล์โค้ดที่มีความเกี่ยวข้องมากที่สุดไปให้กับนักพัฒนาโปรเจ็กต์นี้ประเมินเครื่องมือ Typhon บนโน้ตบุ๊ก Jupyter จากการแข่งขัน Kaggle จากนั้นให้ผู้เชี่ยวชาญที่เป็นมนุษย์ประเมินโค้ดเซลล์ที่แนะนำเพื่อตัดสินว่ามีประโยชน์อย่างไร โครงการนี้ยังเปรียบเทียบประสิทธิภาพของคำแนะนำรหัส Typhon กับเครื่องมือที่เกี่ยวข้อง เช่น GitHub Copilot ในการประเมินประสิทธิภาพและการใช้งานเครื่องมือ Typhon นั้นรับรองได้ว่าเครื่องมือสามารถใช้งานได้ง่ายและนำโค้ดมาใช้ซ้ำได้เป็นอย่างดี อย่างไรก็ดีประสิทธิผลของผลลัพธ์การแนะนำโค้ดนั้นยังไม่พึงพอใจจากกลุ่มผู้ทดลองใช้งานเท่าที่ควร งานชิ้นนี้จึงจำเป็นต้องสำรวจเพิ่มเติมและปรับปรุงในด้านของกระบวนการแนะนำเซลล์โค้ด
81page.81 หน้า

CONTENTS

Page

LIST OF TABLES

Page

LIST OF FIGURES

Page

CHAPTER 1
INTRODUCTION

This chapter makes an introduction to the project including the project's motivation, objectives, scope, and benefits of the project.

1.1 Motivation

Code recommendation has been a great advantage to a lot of software developers for auto-completing the written code or suggesting the next tokens or statements to the developers. This function has helped improve developer productivity significantly. It works by taking the present writing code, if it is part of any known coding statements, an integrated development environment (IDE) or its extension application will suggest the complete command back to the user. This reduces the time the developers need to complete their code tremendously, and helps to minimize the typo errors to almost none.

At present, there are code recommendation tools with various kinds of functionalities. The common functionalities can be distinguished into code-to-code recommendation and text-to-code recommendation. Where a code-to-code recommendation is an approach where the recommendation tools understand the current context of developers' code and recommend the code accordingly. While the text-to-code recommendation will have developers inquire about the details and behaviors of the code they require from the recommendation tool using natural language text.

There have been multiple studies about the concept of code-to-code recommendation tools in past research, for instance, Aroma[2], Senatus[3], Strathcona[4], and Example Overflow[1] where each research study takes a different approach to achieve the common goal, which is recommending code for developers. In addition, the followings are examples of existing text-to-code code recommendation tools that are available; Tabnine and GitHub Copilot, which are some of the most popular recommendation tools capable of recommending fast and efficient code for improving the development workflow.

With the help of modern machine learning inventions, the code recommendation feature has made another big step in the software development industry. Recently, the code recommendation tools have been gaining a lot of attention, accompanied by an outstanding performance of GitHub Copilot, an AI-powered source code recommendation tool based on large language models [1]. GitHub Copilot offers highly adaptive recommendations from a comment string input and code in various languages and frameworks. Another tool worth mentioning is called Tabnine [2] [5], it is a code completion tool empowered by advanced AI and implemented neural network to learn the user's coding style to improve its performance according to each user. At the latest, Tabnine announces support for over thirty mainstream programming languages and twenty popular IDEs.

In the data science domain, there are many various kinds of IDE tools that let developers and researchers work on their projects. Nevertheless, the interfaces and functionalities are similar across every tool. The most commonly used platform is Jupyter Notebook, which enables a simplified and easy-to-navigate workflow, making Jupyter Notebook one of the most popular options in data Science project development[6].

Code reuse is widely spread in many fields of coding and programming, such as software development, data engineering, and data science. The preliminary study of Ritta et al. [7] shows that many data engineers and data scientists reuse code from an open-source platform like Kaggle [3]. Code reuse provides valuable benefits to both newcomer and experienced data scientists. They can learn how to write code and apply their knowledge to advanced work.

In this project, we are interested in code recommendations for the Jupyter Notebooks platform by reusing code snippets that are already written by data science developers. We want to study if we can give appropriate code recommendations based on a given Markdown text in Jupyter Notebooks. This can help reduce the developer's time by not writing the whole code by themselves or learning from the recommended code snippets and improving further upon them. We performed a preliminary experiment with Jupyter Notebooks using GitHub Copilot on Python programming language based on the content

[1] https://github.com/features/copilot

[2] https://aiterms.net/tabnine/

[3] https://www.kaggle.com

of 5 notebooks from the KGTorrent dataset [8] The result showed that the GitHub Copilot efficiency offers acceptable performance with a correctness percentage ranging from a minimum of 0.00 percent to a maximum of 51.16 percent, with an average of 21.83 percent. However, there was still room for improvement. We investigated this further and found that it is potentially due to the specific environment of the Jupyter Notebook, which has mixes of natural language (markdown text) and code. In general, Copilot, the code suggestion system, will take a partial comment, which is supposed to be relatively short, and the whole source code file as input. Then suggests back possible solutions the programmers may need. Many data science notebooks often have long descriptive text in markdown cells. We found that the Copilot sometimes performs source code suggestions poorly when the given input comment is large, and the whole file context contains much descriptive text. The given situation often produces non-reusable source code or a comment string related to the input comment instead of the source code.

Therefore, this project attempts to propose an alternative source code suggestion tool that provides recommended source code from a given descriptive markdown text as input, regardless of the size of the description. Moreover, our proposed technique will not base on code generation, but on code reuse. Our recommended code will be derived from the existing code cells within the Jupyter notebooks in the Kaggle competitions, written by experts in data science.

1.2 Problem Statements

As mentioned above, the existing code recommendation tools may not work well within the context of Jupyter notebooks. In this project, *we want to study a novel technique to recommend code cells in Jupyter notebooks using existing code in a large Jupyter notebook code base in Kaggle, the well-known machine learning and data science platform, using the information from the associated markdown text.*

1.3 Objectives of the Project

The objectives of this project are as follows.

1. *To study the similarity matching techniques specifically designed for Jupyter notebook code cells.* This project primarily focuses on developing a solution for find-

ing the similarity between a single code cell and others code cells in large-scale Jupyter Notebooks databases, such as Kaggle. The studied solution will be further implemented as an application.

2. *To develop a plug-in extension tool for Visual Studio Code.* The proposed solution for matching the similarity of code cells serves as underlying knowledge for the development of a recommendation system. The proposed application which applied the recommendation system will be developed as a plug-in extension tool in the Visual Studio Code environment for ease of use by data science developers.

3. *To evaluate the effectiveness usability of the developed tool.* With the application in place and ready for deployment, the application will undergo an evaluation to measure the usability and usefulness of the developed tool to verify if the developed tool can alleviate the problem proposed in this project.

1.4 Scope of the Project

The scopes of this project are as follows.

1. *The compatibility of programming language and framework.* The focus of this research is source code written inside Jupyter Notebooks. The programming language for the source code provided by the dataset is Python source code. Therefore, the programming language and framework which will comply with the recommendation system will strictly be Python and Jupyter Notebook accordingly.

2. *The compatibility of the dataset.* The dataset used to process within this project scope is derived from the KGTorrent dataset. This dataset contains a collection of Kaggle notebooks metadata and written source code. Therefore, the resulting suggested source code from the recommendation system will derive from the pool of source code in the dataset and provide them as a recommendation.

3. *Environment scope of the developed application.* This project goal is to develop an application for applying the recommendation system with and suggesting Python source code written in Jupyter Notebooks. The environment this project aims to

serve will be as a plug-in extension for Visual Studio Code. Considering that Visual Studio Code extension development already has a large community, the proposed plug-in extension should be beneficial to a large number of developers.

1.5 Expected Benefits

1. *Support the learning process for a developer.* The developers may expect to learn the practice of data science coding through the usage of the developed tool. Considering that the tool is supposed to retrieve the best-matched code cell. Therefore, it is expected that users will have an opportunity to learn and apply the existing solution to their knowledge.

2. *Improving the development performance.* In terms of the recommendation system behavior, it will recommend code cells based on the similarity of descriptive markdown cells. Consequently, the users are expected to obtain the suggested working code cell that has a markdown description that matches the developers' markdown description cell the most.

1.6 Organization of the Document

The document consists of 6 parts: Introduction (Chapter 1), Background (Chapter 2), Analysis and Design (Chapter 3), Implementation (Chapter 4), Evaluation (Chapter 5), and Conclusions (Chapter 6). Firstly, The Introduction chapter includes motivation, problem statements, objectives, scope, expected benefits, and organization of the document. Secondly, the Background chapter describes the background knowledge needed for completing the project, which includes fundamental concepts in code recommendation and code similarity and the related work. Thirdly, the Analysis and design chapter contains work procedures: methodology, system architecture, structure chart, and system analysis. Fourthly, the Implementation contains the technical details and system architecture being implemented in the project. Fifthly, describe the evaluation of experimentation and tests done from the project, and the result analysis from the final test. The final sixth chapter, Conclusions, describes the initial problems, future plan, and the final conclusion of the project.

CHAPTER 2
BACKGROUND

This chapter explains the background knowledge supporting this project and the related work that are similar to this project.

2.1 Jupyter Notebooks

Jupyter Notebook is a web-based interactive computing and documenting tool that functions in terms of a digital document containing cells of text, code, and computing results proposed by Kluyver et al. [6]. The concept of the Jupyter Notebook evolves from the interactive shell or REPL (Read-Evaluate-Print-Loop), which is the basis of interactive programming. Considering that many academic researchers require to document their research progress, they often need to write computer code to perform computative operations, statistical tests, and run simulations. Delivering the composed code in academic papers via human descriptive language is not precise and unreliable for reproducing the implementation code in that paper. Typical academic papers often supply the supplement code in a separate section and are often isolated from the written content. The readers must cross-reference the written content and the relevant code. Consequently, this reduces the readability of the papers and might cause inconsistencies in the code and the prose. Jupyter Notebook aims to improve readability and increase the document's reproducibility and computative results. Therefore, the Notebook is designed to support the scientific research documentation workflow by integrating the text, code, and computative results into a single view. Thus, reducing the effort to cross-reference between code and prose. This Notebook lets authors document their papers via an interactive text and code cells component, enabling later editing and re-executing of the code cells. Hence, increase the reproducibility of the interactive academic documents. Two major platforms provide hosting services and usage for Jupyter Notebook: Google Colab and Kaggle, both owned by Google, and Deepnote which has free and paid Jupyter Notebook services.

2.1.1 Kaggle

Kaggle[1] can be thought of as a large warehouse of open-source Jupyter Notebooks and a playground for data scientists and machine learning engineers. Kaggle is a website that allows users to find and publish data sets and provides a web-based Jupyter Notebooks with a data-science environment. Kaggle users can work together and participate in competitions to solve data science challenges. Due to the popularity of Kaggle Competitions, most users of Kaggle have participated in competitions to challenge each other. Kaggle also has a ranking system for users, and the rank has four categories, including Competitions, Datasets, Notebooks, and Discussions. The rank of each category will depend on the contribution and the prize of each category. According to a large number of competitions, numerous Jupyter Notebooks have been available on Kaggle as public data. Thus, Kaggle is an excellent place to learn from the works of various users, from beginners to experts.

2.1.2 Google Colab

Google Colab[2] is a platform providing services for hosting Jupyter notebooks without configuring the server. It lets users run and compute Jupyter notebook that requires no setup for free, providing users with computing resources and GPUs for zero expense. Google Colab delivers a Jupyter notebook Markdown and Python code cells, organized similarly to regular Jupyter notebooks. The notebooks are served with commonly-use built-in Python packages. It lets users access and manipulates content directly in their Google Drive storage. Also, it gives users the functionality to share the notebook and lets the guest edit or view the shared notebooks. Thus, increasing the reproducibility of the Jupyter Notebook.

2.1.3 Deepnote

Deepnote[3] is the platform that hosts Jupyter Notebook service, which allows less-technical users to comfortably work with Jupyter Notebook. To accommodate the

[1] https://www.kaggle.com/

[2] https://research.google.com/colaboratory/faq.html

[3] https://deepnote.com/

experience To accommodate users' experience to easily work with Jupyter Notebook, Deepnote offers better realtime-collaboration, versioning, and comments. Additionally, Deepnote allows for notebooks organization, which is similar to documentation structure, so the content is organized in an easy-to-find manner. Furthermore, it offers various features for collaboration with other tools, for instance, connectors with external data sources (e.g. Snowflake, PostgreSQL), and visual chart builders.

2.1.4 A Study of Code Reuse in Jupyter Notebooks

According to the preliminary study of Ritta et al. [7], they investigated if code reuse occurs in Jupyter Notebooks of Kaggle Competition. They selected three competitors that have high-level skills in writing and programming in Jupyter Notebooks. After that, they extracted all notebooks from all the competitions of three competitors. The code cells extracted from each competitor's latest competitions will be used as queries to find the reused code cells via NCDSearch[4]. NCDSearch is a grep-like tool. One command of a Unix-like operating system named grep is used for searching plain text from lines that match a regular expression in files. Thus, NCDSearch is created from the grep concept to find similar source code fragments in specific files. Any code cell with a similarity distance from 0 to 0.5 was counted as reused code. They also classified and found the common type of reused code to see which type is the most code that is reused. They discovered that competitors might have different most reused code types, but the most common type that everyone reused is Import Package, which is code used for importing Python's packages. Additionally, the authors suggest that their work has the potential to be fundamental for the automatic code-recommending tool because code recommendation can come from reusing the existing codes from open-source projects.

2.2 Code Recommendation

This section discusses the related work of code reuse in Jupyter Notebooks and various research papers in the code recommendation areas. This includes code-to-code recommendations (i.e., search by using code and receive code as results) such as Aroma, Strathcona, Senatus, and Example Overflow. Moreover, we also discuss a text-to-code

[4] https://github.com/takashi-ishio/NCDSearch

recommendation study (i.e., search by using natural text and receive code as results) such as CodeSearchNet.

2.2.1 Code-to-code Recommendations

We discuss some of the existing work in code-to-code recommendation tools and techniques below.

Aroma is a source code recommendation tool that is implemented with a code-to-code search concept from Luan et al. [2]. The core idea of Aroma is recommending any possible complete and working or extended code snippets, having the input as partially written source code. Aroma's purposes are to address the extensibility opportunity of the existing code snippet, such as error handling parts and setup of the code, and comparison with the common pattern other developers have made. It takes a code snippet as input, and then Aroma will index the given corpus of code and retrieve a small set of code snippets, where the retrieved code snippet will approximately contain the input code snippet. Then Aroma will take these retrieved snippets through a series of pruning, ranking, and clustering. After clustering, Aroma filters these clustered snippets with intersecting operations to derive a set of the most common snippets among the retrieved snippets. These intersected snippets are recommended to the user as a set of alternative extended code snippets. Although Aroma shares the same concept of recommending source code, the input of Aroma is taken as a snippet of code, whereas our project focuses on the input as Markdown cells.

Senatus is a code-to-code recommendation engine for developers that aim to enhance their productivity, efficiency, and the reliability and consistency of the code. The team builds Senatus by following the definition given by Luan et al. [2]. The authors propose a new method of code recommendation by indexing the code repository with Minhash-Locality Sensitive Hashing or Minhas-LSH. Minhash is generally used for maintaining the Jaccard similarity of sets by representing those sets as a lower dimensional similarity and preserving features of their vectors. This technique provides much better efficiency at the query time. Minhash can be applied in many applications such as web search, graph sampling, earthquake detection, and efficient clustering of a very large collection of DNA sequences. However, by using Minhash in the code-to-code

recommendations, they provide poor retrieval quality because of the length of skewness in the code snippet. Senatus fixes this issue by operating directly on the abstract syntax tree (AST) structure representation of the code which decreases the time to query and the effectiveness of the retrieval.

Strathcona: Another benefit of code recommendation is provided by Strathcona Example Recommendation Tool from Holmes et al. [4], a tool that helps developers understand and use API. Incomplete or outdated documentation about API usage causes problems for developers, so the tool aims to reduce the gap between source code and documentation. In the case of understanding API usage, developers must spend a lot of time manually searching the relevant source code showing how to use the API. With the example recommendation tool, developers can write code and get recommended examples for API usage. The authors of the work intended to make the developers use less effort to get examples. The code written by developers will be a query for searching. Then the tool will locate structurally relevant examples by comparing the context of the structure between the query and the existing repository. Moreover, after the tool finds the relevant examples and returns them, developers are able to navigate the example to inspect it in more detail as desired. Returned examples can be navigated visually and textually.

Example Overflow: Using the idea of social media or wisdom of crowd recommendations, Zagalsky et al. [1] propose Example Overflow, a code search, and recommendation tool. Example Overflow is a generic recommendation system that utilizes the code snippet and textual data on Stack Overflow. The approach's core idea is to retrieve the relevant code snippets based on human evaluation. Other existing search tools such as Strathcona, Aroma, and Senatus recommend relevant code snippets using the machine processing power to compute the rank and similarity. On the other hand, Example Overflow leverages the crowd wisdom from answers to questions from the Stack Overflow website. Intuitively, many programmers of any experience often learn and code by following examples of code snippets in the tool documentation or code example on the Internet. Therefore, Example Overflow offers two core practices by the system. The first practice is comparing multiple examples simultaneously without switching developing context, either by switching the browser tabs or searching for code examples. The

system aims to serve this key point by presenting the top five most relevant results, all in the same view. In the second practice, the system takes minimal context switching, according to the first practice, where the programmers are presented with the top five results. To support the second practice, Example Overflow has a user interface design with a single search window which can be seen in Figure 2.1.

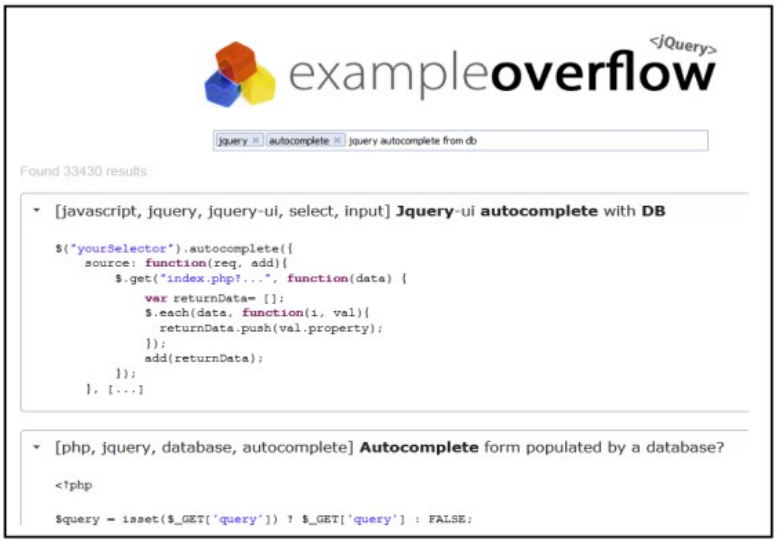


Figure 2.1: Example Overflow Web Interface from the Example Overflow paper [1]

2.2.2 Text-to-code Recommendations

We discuss some of the existing work in text-to-code recommendation tools and techniques below.

CodeSearchNet is a collaboration between GitHub and the Deep Program Understanding group at Microsoft Research - Cambridge [9]. It is a dataset and benchmark for code retrieval techniques. The goal of CodeSearchNet is to allow developers and researchers to develop and improve code retrieval techniques which are essential for improving software development productivity and quality.

CodeSearchNet dataset is collected from publicly available open-source repositories on GitHub. The dataset includes over 6 million methods consisting of GO, Java, JavaScript, Python, PHP, and Ruby programming languages. Of these, 2 million methods have associated documentation. This results in a set of pairs between the method of code and its documentation.

CodeSearchNet also provides a benchmark framework to measure and compare the performance of different code retrieval techniques. Its purpose is to encourage researchers and practitioners to further study semantic code search tasks, which involve retrieving relevant code given a natural language query. The process is generally as follows: 99 queries from the natural language with its corresponding possible programming language results in each considered language (Go, Java, JavaScript, PHP, Python, and Ruby). Then, these query/results pairs are labeled by human experts to indicate the relevance of the result and the query. Finally, the experts annotate the result for relevance on a zero to three scale. The framework supports exact match retrieval, semantic retrieval, and cross-lingual retrieval [9].

On the leaderboard of the CodeSearchNet benchmark 4 out of the top 5 results use the NeuralBoW (Neural Bag-of-Words). The basic idea behind NeuralBoW is to first represent each word in a document using a one-hot encoding to create a vector of zeros except for a single 1 in the position corresponding to the index of the word in a predefined vocabulary. These vectors are then averaged together to produce a fixed-length vector representation of the entire document. This document vector can then be used as input to a neural network classifier to predict the document's class label.

2.3 Techniques to Measure Code and Text Similarity

Since there are both code-to-code recommendations and text-to-code recommendations, we explain the technique to measure code and text similarity below.

2.3.1 Code Similarity Detection

In this era, code similarity and plagiarism have gained a lot of attention in software engineering, with many contributions from developers and researchers. Thus, many code similarity analyzers were developed to be tools for use in different aspects, according to Ragkhitwetsagul et al.'s empirical study of a wide-range comparison of code similarity analyzers [10]. In the study, the authors do a broad performance-based evaluation of 30 code similarity analyzers and also techniques for code similarity detection. They show that different tools can have different performances depending on the technique used by the tool. The tools used specialized source code similarity detection techniques

has a better performance compared to general textual similarity measures.

Furthermore, according to the same work, the authors mention different types of code similarity (i.e., cloned code). The types of code similarity can be categorized into four types including

1. *Type I clones* are source code cloning with lexical changes of formatting and layout modifications.

2. *Type II clones* are source code cloning with lexical changes of formatting, similar to Type I but with identifier renaming (e.g., `a=5` to `b=5`) and different literals (such as string or numbers).

3. *Type III clones* are source code cloning with structural changes (e.g. `if` to `switch` or `while` to `for`). Code with insertions, deletions, and modifications of statements is also categorized as Type III clones.

4. *Type IV clones* are source code cloning that shares the same semantics but with different implementations.

2.3.2 Vector Representation Models for Text

Multiple techniques in the Information Retrieval methods are used to recommend similar text in the collection of documents.

Best Matching 25: BM25, essentially, the BM25 algorithm is a bag-of-words retrieval function, which ranks a set of documents regarding the query terms appearing in each document regardless of the proximity within the document. The below formula is shown to demonstrate how to compute the similarity between the query terms and the comparison documents.

$$score(Doc, Query) = \sum_{i=1}^{n} IDF(q_i) \cdot \frac{f(q_i, D) \cdot (k_1 + 1)}{f(q_i, D) + k_1 \cdot (1 - b + b \cdot \frac{fieldLen}{avgFieldLen})}$$

Each component in the formula is briefly explained in the following points.

1. q_i : The ith query term

2. $IDF(q_i)$: The inverse document frequency of the ith query term, which measures how often a term occurs in all documents and penalizes terms that are common.

3. $f(q_i, D)$: The number of times the ith query term occurs in document D.

4. k_1 : A variable that limits how much a single query term can affect the score of a document.

5. b : A variable that amplifies the effect of the length of a document relative to the average document length.

6. $fieldLen$: The length of the field (in terms) of the document.

7. $avgFieldLen$: The average length of the field (in terms) of all documents.

2.3.3 Vector Representation Models for Code

Moreover, recently there have been studies about creating code models, which are machine-learning models that derive vectors (i.e., code embeddings) from code snippets. We discuss some of the recent code models below.

CodeBERT is a natural language processing (NLP) pre-trained model developed by Microsoft that can understand both programming language and natural language. The paper presents a hybrid objective function that incorporates the pre-training task of replaced token detection to train the model. CodeBERT can be used for various tasks, such as natural language code search and code documentation generation. The model achieves state-of-the-art performance on two natural language-programming language (NL-PL) applications and performs better than previous pre-trained models on NL-PL probing. In summary, CodeBERT is a powerful tool for understanding and generating both programming language and natural language, which can be useful for various applications in software engineering.

UniXcoder, also developed by Microsoft, is a unified cross-modal pre-trained model for programming language[11], which specialize in code understanding and code generation tasks. UniXcoder is a pre-trained NLP cross-modal model that facilitates the communication between natural language and programming languages for code-related tasks. Additionally, UniXcoder is a pre-trained model in the CodeBERT series from

Microsoft, similarly as CodeBERT mentioned previously. It is designed to understand and generate code from natural language queries or descriptions. UniXcoder is a unified cross-modal model that bridges the gap between natural language and programming language by leveraging advanced machine learning techniques. It is pre-trained on a large corpus of natural language text and programming languages, making it more efficient and effective in code-related tasks.

Additionally, UniXcoder uses a mask attention mechanism with a prefix adapter to control the encode-decode behavior. This allows the model to adjust its output based on the input, improving its accuracy and relevance to the task at hand. Additionally, the model leverages cross-modal content, such as abstract syntax trees and code comments, to enhance code representation. This improves the quality of the code generated by the model and makes it more natural and readable.

To conclude, UniXcoder is capable of both code-related understanding and generation tasks. This means that it can understand natural language queries or descriptions and generate the corresponding code, or it can take code as input and generate natural language descriptions or comments about the code. This versatility makes it a powerful tool for developers, researchers, and other professionals working with natural language and programming languages.

2.3.4 Existing Commercial Tools for Code Recommendation

GitHub Copilot: Resulting of collaboration between GitHub and OpenAI [12], Github Copilot is a cloud-based artificial intelligence tool powered by a modern AI named Codex which itself is based on GPT-3 model (third-generation pre-trained transformer) [13]. Copilot is trained with billion lines of publicly available code on GitHub. Copilot works with Visual Studio code, Neovim, and JetBrains IDE in which the extension will suggest or auto-complete the code based on the context of the comment and existing code (see Figure 2.2 for an example of code recommended by GitHub Copilot).

GitHub Copilot bases its suggestion on the context of the file that the user is editing. Copilot will mainly look at a comment and developing environment to provide suggestion code which collaborative with the written comment and environment.

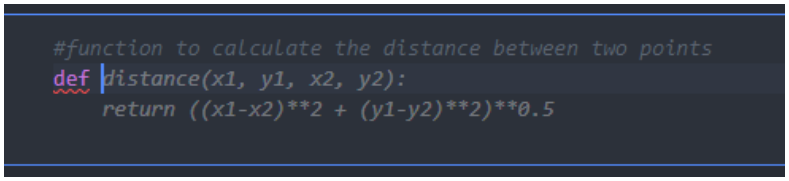


Figure 2.2: The image shows that by writing a comment, Copilot will suggest code based on that comment and the current developing environment

Tabnine[5]**:** is another AI-powered code recommendation program. Tabnine's AI model is trained on a dataset of code and has specified optimized for each programming language and domain. Tabnine will adapt to each person's coding standards and best practices. By connecting to each person's repository Tabnine can learn and train its model on that specific code and tailor code completions based on context and past code.

ChatGPT[6]**:** is a large language model developed by OpenAI, based on the GPT-3.5 architecture. It is one of the most advanced AI language models ever created, with a staggering 175 billion parameters. This allows it to understand and generate text in an incredibly natural and human-like way, making it a powerful tool for a wide range of applications.

ChatGPT has become incredibly popular recently, with many individuals and organizations using it for a variety of purposes. Some of the most common uses of ChatGPT include language translation, text summarization, and conversational AI. Additionally, many people use ChatGPT for creative writing, such as generating poems or short stories. Its popularity has led to a wide range of applications, and its advanced capabilities have made it an invaluable tool for anyone looking to work with text data in a powerful and efficient way [14].

2.4 Chapter Summary

The second chapter of this project discusses the fundamental concept of Jupyter Notebook, a web-based tool for programming in python and documentation. There are many services that provide for hosting Jupyter Notebook servers. One is Kaggle, an open-source playground for data science and beginning machine learning engineers.

[5] https://www.tabnine.com/

[6] https://openai.com/blog/chatgpt

Google also provides a service to host a Jupyter Notebook server for free. This chapter explored existing tools and technology on the topic of code reuse and code recommendation, such as Aroma a source code recommendation tool, Senatus a code-to-code recommendation engine, Strathcona recommendation tool, and Example Overflow a code search and recommendation tool. Another topic in this chapter is Code Similarity, a study on code similarity analyzers and types of code similarity. Lastly, this chapter covers modern code medals that are based on the machine-learning model to provide code recommendations, such as CodeBERT, Github Copilot, and Tabnine.

CHAPTER 3
ANALYSIS AND DESIGN

This chapter explains the initial experiment with GitHub Copilot, the overview of the Typhon framework, the proposed methodology, and the comparison of related works.

3.1 Evaluation of GitHub Copilot on Jupyter Notebooks

Since GitHub Copilot can recommend relevant code snippets in many programming languages, we evaluated GitHub Copilot in the setting of Jupyter Notebook. This is to assess the usefulness of the generated code from Copilot given an associated markdown text.

3.1.1 Methodology

We set up our study as follows. First, we selected five Jupyter notebooks from the Kaggle dataset. Second, we used Copilot to generate a code cell by giving its associated markdown text. Third, we manually evaluated the suggested code. We explain each step in detail below.

Step 1: Jupyter Notebook selection

The preliminary experiment of GitHub Copilot code recommendation uses the data from the Kaggle website. First, five Kaggle competitions were randomly selected without any consideration. Then, the participants' notebook is sorted using **Most Vote** as the criteria. We selected five Jupyter notebooks based on the number of upvotes that each notebook in each competition has received up until the present. The five selected notebooks are described in Table 3.1. The notebooks were extracted on September 26, 2022.

Step 2: Code suggestion generation

Technically, Copilot will generate the recommended code when the users enter a new line, users can choose to take the recommended code by tapping the **Tab** button

Table 3.1: Five selected Jupyter notebooks for evaluating GitHub Copilot suggestions

Notebook	No. of votes
Titanic Data Science Solutions	9,365
Start Here: A Gentle Introduction	3,372
Time series Basics: Exploring traditional TS	1,764
Santander EDA and Prediction	1,215
EDA and models	1,096

on the keyboard, or entering another new line to make Copilot recommended new code. Therefore, in the project's preliminary experiment, the first recommended code from Copilot will be taken into the experimental evaluation. In the experiment, a new Jupyter notebook was created for each one of the competitions for testing with code generation from Copilot. A Markdown cell, which is on top of the code cell, will be copied from the original notebook, and pasted into the new Jupyter notebook. The next step is to have the Copilot generate the code base on the context of pasted Markdown description. In the case that Copilot does not generate code upon entering a newline, another newline will be added to try to trigger the Copilot code generation. After getting the recommended code for the copied Markdown cells, the same steps will be repeated until the last Markdown cell from the original notebook is copied over and recommended code is documented.

Step 3: Evaluation of code suggestions

The evaluation of the generated code snippet in the Python code cell was conducted with three evaluators, and each of them independently assess the efficiency of the recommended code snippet, at the end the final evaluation of the generated code snippet is determined from the consensus of the three evaluators. When all new notebooks for each competition are filled with recommended code cells from Copilot, all evaluators will take these notebooks and evaluate the generated code cells recommended from the Markdown cell. Each evaluator will evaluate by deciding whether the recommended code can satisfy two criteria, the generated code can run, and the produced result is similar to the original code cell. The evaluation was conducted on Google Sheets, where each sheet represents one competition that was randomly selected, and each row in the competition sheet represents the Python code cell which generates from GitHub Copilot. Each row contains three checkboxes, one for each evaluator, for ticking to decide that

the generated code cell can satisfy the two given criteria. During the experiment, evaluators will evaluate different notebooks asynchronously, and hide their checkbox columns when done with one notebook.

3.1.2 Results

After completing the evaluation, the ratio of accepted code cells overall code cells is computed for each notebook. These percentage ratios are shown in the Figure 3.1. The notebook that Copilot can perform code generation the best is *Start Here: A Gentle Introductio* [1] that takes part in *Home Credit Default Risk* competition, with the acceptance ratio over all generated code cells of 51.16 percent. The worst notebook that Copilot can achieve is *EDA and models* [2] that takes part in *IEEE-CIS Fraud Detection* competition, with the acceptance ratio of 0.00 percent. The average ratio of all notebooks of 21.83 percent. There is a significant sign that indicates Copilot underperformance in an *EDA and models* notebook. The possible reason to explain this underperformance could trace to the content of the input Markdown cell description. Typically, Markdown cells description commonly found in Kaggle notebooks are varied in length and content. The assumption for Copilot underperformance is from this variety in Markdown cells, there are possibilities that the provided description may be too little to provide enough context, or maybe too much that it overflows the context that Copilot takes to process and delivers the recommended code. The result of the experiment can be concluded that the reliability of the recommended code cells primarily depends on the quality and context that the input Markdown cell can provide.

[1] https://www.kaggle.com/code/willkoehrsen/start-here-a-gentle-introduction

[2] https://www.kaggle.com/code/artgor/eda-and-models

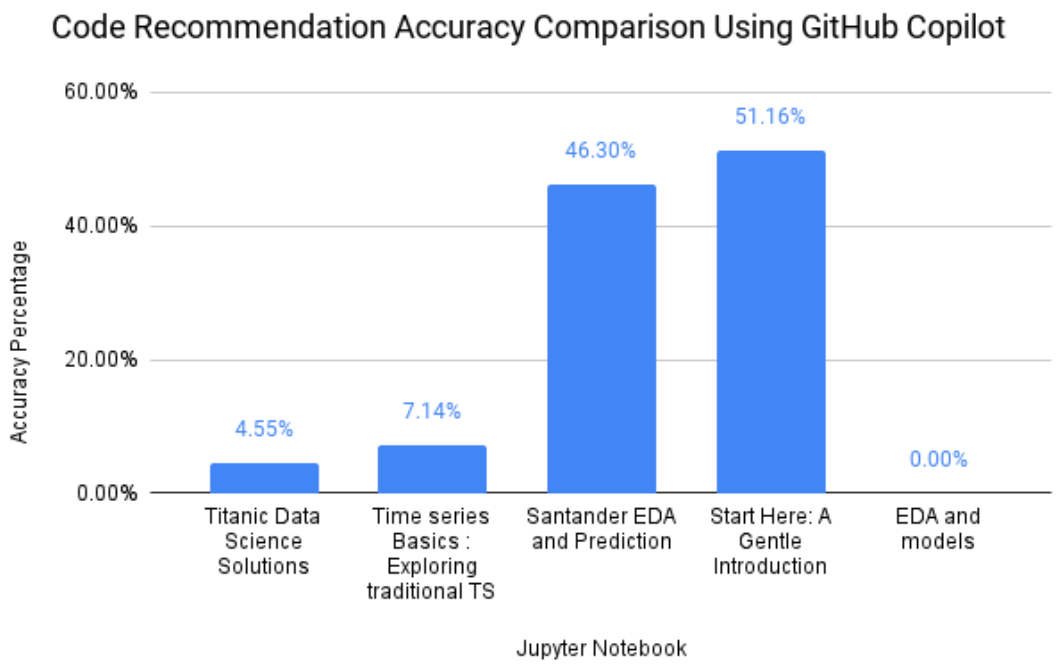


Figure 3.1: The preliminary testing result from GitHub Copilot code recommendation for 5 Jupyter notebooks from Kaggle.

3.2 Proposed Methodology: Typhon

After studying the use of Copilot on Jupyter notebooks, we found that there is still some room for improvement. Thus, we decided to propose a new methodology, called **Typhon**, for recommending code cells based on existing code cells from Jupyter notebooks in the Kaggle dataset.

3.2.1 Overview of the Framework

As mentioned in Chapter 1, existing code recommendation tools may not work well with the Jupyter Notebook environment, specifically in a data science context. Where the Markdown description cells are varied in terms of length and detail. The recommendation tool often cannot recommend reliable code cells for users because the code and text in the local file's context sometimes provide vague and unclear context. Therefore, our proposed tool offers a capability that focuses on suggesting code cell(s) based on a Markdown cell's content, which concentrates on the data science practice.

Afterward, the tool will search through the code corpus for the Markdown cell with the most similarity with the input Markdown cell, then suggest the code cell associ-

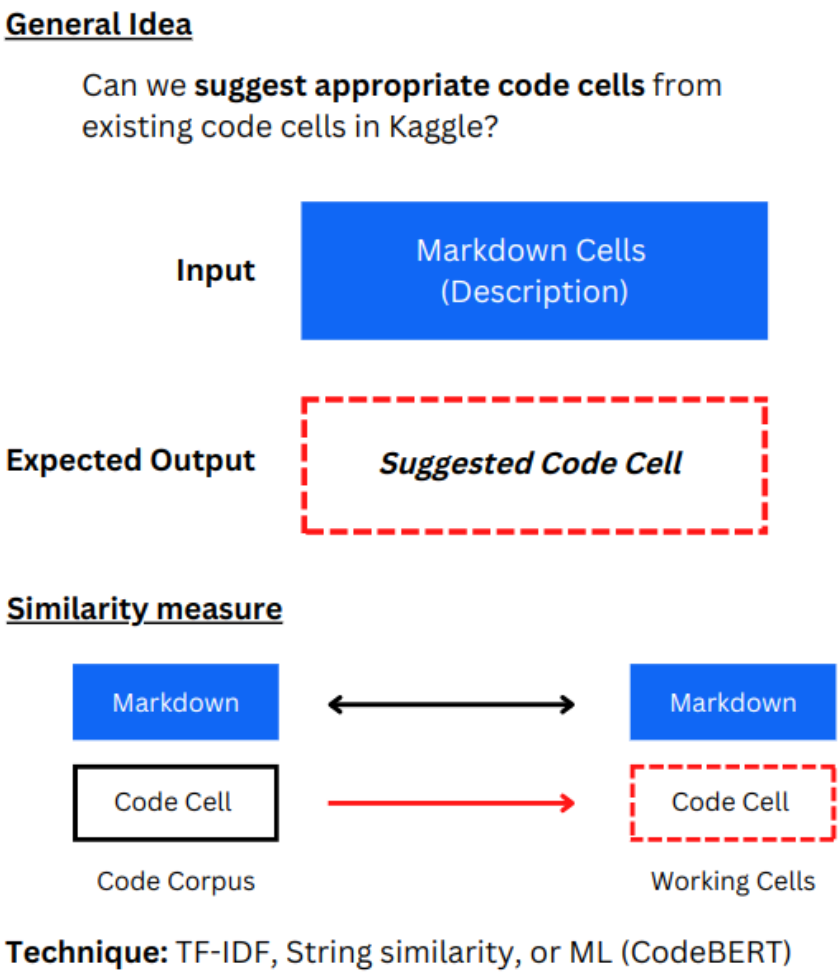


Figure 3.2: The general idea of Typhon's recommendation system framework

ated with the Markdown cell with the most similarity to the input to the user. The concept of the code cell recommendation by the tool is presented in the Figure 3.2. We plan to evaluate a few techniques to find similar Markdown cells including information retrieval such as BM25, code embedding such as UniXcoder, and other potential techniques.

Similar to GitHub Copilot, the tool also functions as Visual Studio Code Extension, in which users will continue writing their code and seamlessly recommend code cells from the tool. Also, the proposed tool offers code cell generation after finishing writing the Markdown cell, which is similar to GitHub Copilot, where the users get recommended code after entering a new line to minimize the overhead of context switching, as much as Copilot did as possible.

3.2.2 Dataset: KGTorrent

KGTorrent is an extensive dataset of computational notebooks with dense metadata retrieved from the Google-owned platform Kaggle. The collection process covers the time period of November 2015 to October 2020. The data is collected from around 5,598,921 users and 2,910 competitions. KGTorrent has a size of 175 GB and compress to the size of 76 GB and consists of 248,761 Jupyter notebooks in the Python programming language.

3.2.3 Techniques for Locating Recommended Code Cells

The followings are the potential techniques to find text similarity for locating the most similar code cells.

1) Information Retrieval Technique: BM25

One possible solution for finding the similarity of Markdown text in the proposed tool is by integrating the BM25 as the similarity-finding method. BM25 is a ranking function that can rank the similarity weight of the documents and query terms, which is sufficient to solve the proposed problem of finding the similarity among Markdown cells. The general flow of the system with BM25 starts with the system taking a query Markdown cell as input, then computing the similarity score with each document in the database. The more similar the query Markdown cell has with the document, the higher the similarity score. After that, the system delivers the code cells corresponding to the ranked Markdown cells with the highest similarity.

2) Machine Learning Technique: UniXcoder

One possible approach is to use UniXcoder to assist in the similarity-finding method. UniXcoder is a pre-trained cross-modal model specialized in natural language (NL) and programming language (PL), and it supports various downstream NL-PL tasks. UniXcoder is based on the transformer architecture and is specifically designed for handling code-related tasks. The model can take natural language text and/or code as input and generate an embedding vector that represents the input. In the typical workflow, the input is taken into a UniXcoder model to produce an embedding vector representing the input. In addition, Typhon system utilizes the Weaviate, a vector search engine database

for accommodating vectorizing process and storing a collection of code dataset. Both the similarity finding method, and returning such results to middleware system are happening within the Weaviate system. Then, the middleware system will take care of refining the relevance code and returning recommended code cells to users.

3.2.4 Recommend Relevant Code Cells in Visual Studio Code

The proposed methodology is planned to be implemented as an extension of Visual Studio Code. This is to help the data science developers to make use of the Typhon recommendation in an easy and intuitive way. We analyze the use cases for the Typhon tool as follows.

Use Case Analysis

Use cases of Typhon's users are displayed in the Figure 3.3. Detail of each use case is shown in the Table 3.2

1. *Get top-ranked code cells* The system recommends and appends the top-ranked recommended code cell to the user's notebook environment after the Markdown description cell.

2. *Get a list of similar code cells ordered by rank* The system opens a new tab in Visual Studio Code to display the list of recommended code cells ordered by the similarity score.

3. *Manage suggested code cells* The user interact with the configuration of Typhon extension to choose their recommendation approach of choice.

Data Flow Diagram

Typhon receives input from the developers and outputs the code recommendations to the developers via the Visual Studio Code IDE. We explain its processes using the data flow diagram. The followings are descriptions for Typhon's data flow diagrams in levels 0 and 1.

1. **Data Flow Diagram Level 0:** In Figure 3.4, the users start interacting with the extension by pressing the trigger button, and the extension brings forward the Mark-

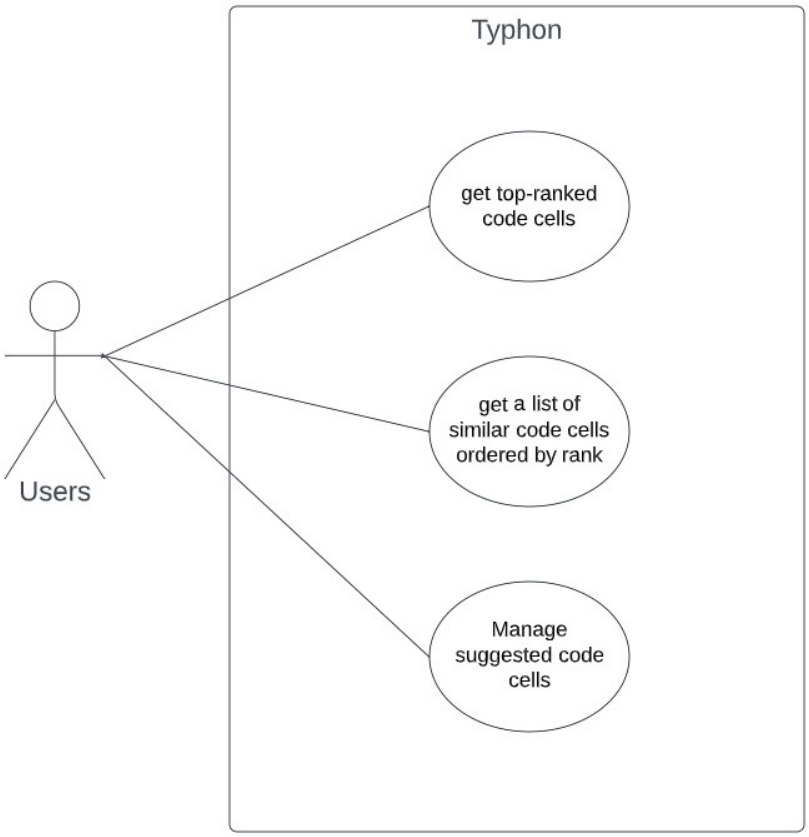


Figure 3.3: Use case analysis diagram for Typhon code cells recommendation tool.

down description content to the system. Then, the top-ranked code cells or list of ranked code cells are returned to the users.

2. **Data Flow Diagram Level 1:** In Figure 3.5, the users trigger the extension interaction and send forward the Jupyter notebook content. Then, Typhon takes Markdown cell input where the Typhon is triggered, tokenizes the input, and then finds the similarity of the tokenized description with the collection of the document in the corpus. After that, a list of code cells ranked by the similarity score is returned to the system and then delivered back to the users.

Table 3.2: Use cases of Typhon

Number	Actor	Use Cases	Description
1	Users	Get top-ranked code cells	Users request code cells that have the most similarity to the given Markdown cell.
2	Users	Get a list of similar code cells ordered by rank.	Users request a list of code cells that are ranked by the similarity score to the given Markdown cell.
3	Users	Manage suggested code cells	Users configure the recommendation approach

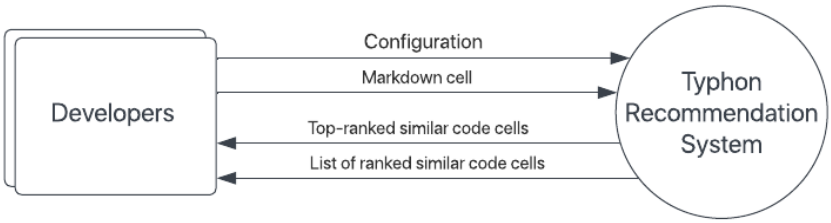


Figure 3.4: Data flow diagram level 0 for Typhon recommendation system.

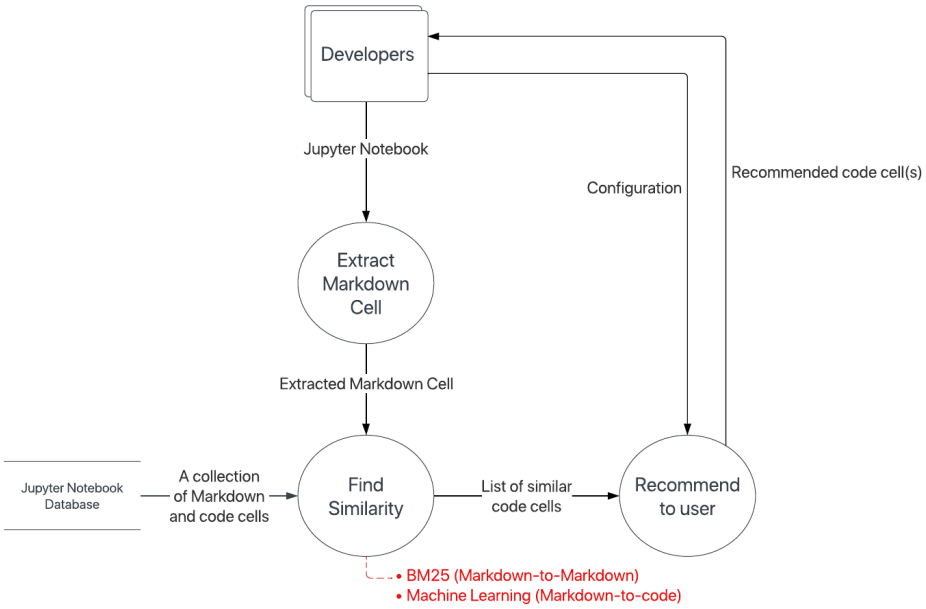


Figure 3.5: Data flow diagram level 1 for Typhon recommendation system.

3.3 Comparison To Related Works

The proposed tool shares several similar traits against the GitHub Copilot, which is the tool the project was trying to study.

3.3.1 Comparison of Typhon approach to other related works approaches

The comparison of Typhon to other related works approaches is described in Table 3.3. The seven techniques, including Typhon, have different ways of recommending code. Copilot and Tabnine use AI-based techniques to generate new code from scratch. The other five techniques; Aroma, Strathcona, Senatus, ExampleOverflow, and Typhon; are based on searching from existing code based by using the other code snippet or search keywords as queries. Aroma recommends clustering and ranking code snippets categories, Strathcona recommends from the similarity between the user's IDE structural details and code structural detail in repositories, Senatus recommends from the similarity of input description and database using Jaccard similarity, Example Overflow recommends from fine-tuned TF-IDF weight algorithm, and Typhon suggests code cells from the current studying methods: BM25 and Machine Learning.

3.3.2 Comparison of Typhon and Copilot in terms of Visual Studio Code extension

The followings are detailed comparisons of the proposed tool with GitHub Copilot as an extension of Visual Studio Code. The comparison topics are as follows: the installation process, the tool interaction to generate code, the cancellation of the suggested code, the referenced source in the recommended code, and the comparison of the recommendation results.

Installation

The installation process for the proposed tool and GitHub Copilot is the same, considering that GitHub Copilot is a Visual Studio Code extension, which is also the same for the proposed tool. It is designed to be an extension tool to reduce the effort of installing the tool and getting it to work.

Table 3.3: Comparison of code recommendation approaches

Name	Approach
Typhon	Search and recommend code cells based on the similarity of markdown text cells using BM25 or Machine Learning.
Copilot	Generate code cells using the OpenAI Codex model trained on large open-source projects in GitHub
Tabnine	Generate code snippets based on AI-based proprietary algorithm
Aroma	Search and recommend code snippets based on the clustered and ranked code snippet categories
Strathcona	Search and recommend code snippets based on the user's IDE structural details with relevant to structural details in code repositories
Senatus	Search and recommend code snippets based on Jaccard similarity using Minhash-LSH on the indexed code
ExampleOverflow	Search and recommend code snippets based on similarity of keyword search using fine-tuned TF-IDF weight

The tool interaction to generate code

The interaction between the tools that require triggering the recommendation function of GitHub Copilot and the proposed tool differs to a certain degree. In GitHub Copilot, users start by writing a comment or partial code; with a comment, entering a new line will trigger a code recommendation. With partial code, Copilot suggests code with the dark grey highlight color. As for the proposed tool, after the users execute a Markdown cell, Python code cells are generated after the executed the Markdown cell.

The cancellation of the suggested code

The cancellation of the suggested code between GitHub Copilot and the proposed tool is different. GitHub Copilot suggested code can be canceled by pressing any button apart from the **tab** button, and the editor will clear the suggestion. In the proposed, the suggested code cells will have to be removed manually by clicking at **Delete Cell** button, which is represented as a trash bin icon on the top left of the cell.

The referenced source in the recommended code

Both GitHub Copilot and the proposed tool can now show references to their generated code. In GitHub Copilot, If the generated code matches multiple references, it will approximately tell the number of matches and provide a button for viewing the references [15]. While the reference for recommended code cells in our Typhon exten-

sion is embedded into the code cells as a comment at the top, providing traceability to the original notebook from which the suggested code cells were taken.

3.4 Chapter Summary

This chapter discusses the preliminary experiment of code recommendation in the Jupyter notebook context with GitHub Copilot, the overview of the Typhon framework, and an explanation of how the system works, including the study of proposed techniques for recommending code cells. Additionally, this chapter includes the system architecture of Typhon, the use case analysis diagram, and the data flow diagram in levels 0 and 1. Furthermore, this chapter provides a comparison of the Typhon approach with other related works, which compares in terms of the recommended approach and extension detail.

CHAPTER 4
IMPLEMENTATION

In this chapter, we explain the development process of the Typhon software along with technical issues we encounter while in the development process and how we overcame them.

4.1 Software and System Environment

This section explains the software and system environment that we use to develop Typhon.

Figure 4.1 depicts how the services in the Typhon environment are connected and communicated the requests from users and responses from the servers are sent.

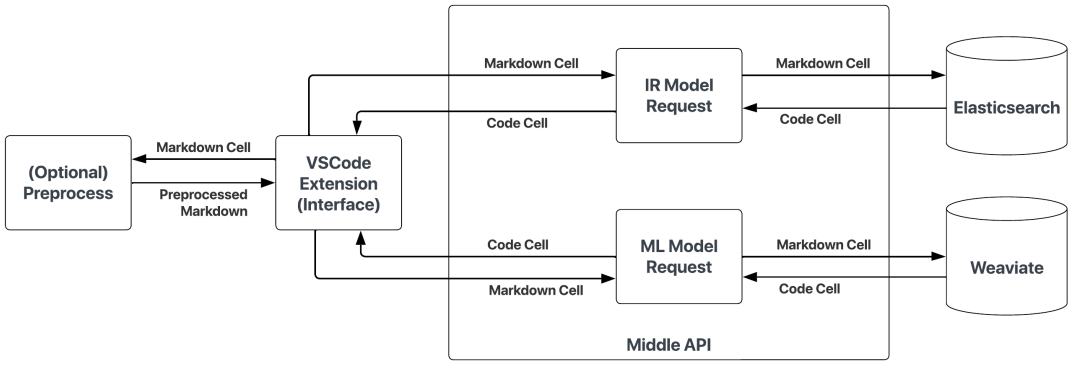


Figure 4.1: System Architecture of Typhon

4.1.1 System Setup and Hardware

The following section describes the hardware used within the Typhon system, where all services and devices are hosted on the PROEN[1] Cloud Service provider.

1. **Node.js Middleware API Service**

[1] https://www.proen.cloud/en/home/

- CPU: 6.4GHz
- Memory: 2GB
- OS: CentOS 7 Linux

2. **Text Processing Service**
   - CPU: 51.2 GHz
   - Memory: 16GB
   - OS: Alpine Linux
3. **Elasticsearch Engine Instance**
   - CPU: 51.2 GHz
   - Memory: 16GB
   - OS: Ubuntu 20.04.5 LTS
4. **Weaviate System Instance**
   - CPU: 51.2 GHz
   - Memory: 16GB
   - OS: Alpine Linux

4.2 Software and Tools used

The following section describes the tools and software being used in the project and briefly explains their purpose.

4.2.1 Development tools

The next section is a list of software that was used to develop and test the Typhon infrastructure and system.

1. **Microsoft Visual Studio Code** A free and open-source code editor that is used for building and debugging various software applications. It provides a range of features that support multiple programming languages, including syntax highlighting, code completion, debugging tools, and Git integration.

2. **Jupyter Notebook** A web application for creating and sharing interactive code, visualizations, and documents. It supports multiple programming languages and is widely used for data analysis, scientific computing, and education.

3. **Postman** A software application used for developing, testing, and documenting APIs. It provides a user-friendly interface, automation features, and collaboration tools for developers.

4. **Kibana** An open-source data visualization and exploration platform that allows users to analyze and understand large datasets. It is part of the Elastic Stack, which includes Elasticsearch and Logstash and is used primarily for visualizing data stored in Elasticsearch.

5. **Node.JS** A popular open-source runtime environment that allows developers to run JavaScript on the server side. It is known for its scalability, non-blocking I/O model, and a vast collection of libraries and frameworks.

6. **FastAPI** A modern, fast web framework for building APIs with Python 3.7+ based on the asynchronous programming paradigm. It provides a simple and intuitive interface for creating web applications while also allowing for highly performant and scalable code.

4.2.2 Databases

The following is a list of the database engine used to hold the code and Markdown data for a recommendation.

1. **Elasticsearch** An open-source, distributed search engine that allows users to store, search, and analyze data in real-time. It is widely used for its scalability, flexibility, and speed and is particularly useful for search applications and analytics.

2. **Weaviate** An open-source, cloud-native, real-time vector search engine that uses machine learning to provide a semantic search for unstructured and structured data. It allows users to index and search data in real-time and provides a natural language interface for searching data.

4.3 Software Architecture

Since our primary focus is research, we have to design the system architecture of the tool, Typhon, to have the most flexibility as much as possible. We use the concept of Microservices for the implementation of Typhon, which separates each service or module to have low coupling to support changes in tool usage. The system architecture of Typhon is divided into 4 primary services.

4.3.1 Visual Studio Code

Typhon is an extension of Visual Studio Code, one of the most popular IDE developed by Microsoft. If we apply the same logic of general software system implementation, an extension of Visual Studio Code is the interface of the system. The extension will send the markdown description that the user inputs to the preprocessing system or middle API, depending on the user-selected feature on the extension. The architectural diagram of the system is depicted on Figure 4.1

Typhon Extension Development

We developed the Typhon extension by using the provided Visual Studio Code API. Since Typhon supports only Jupyter Notebook, we designed Typhon to get an entire text from an active markdown cell using the specific API for the Notebook editor. After that, the text will send to our own created API as a query for related code searching.

In the part of the result presentation, the related codes are sent as the search response from the database, either Elasticsearch or Weaviate. When the search response is sent back to the Typhon extension, the extension will insert a new code cell below an active markdown. The inserted code cell will come with the top-rank (highest score) related code snippet if using the default function of Typhon. However, the blank code cell will also be inserted if the user uses the function that gets multiple related code snippets to decide the code usage later.

For the usage of Typhon as a Visual Studio Code extension, the user can write the description on the markdown cell and then click on the Typhon button to use the tool shown in Figure 4.2. Typhon will send back or insert the code related to the description in the markdown cell shown in Figure 4.3. The user can choose to let Typhon insert a

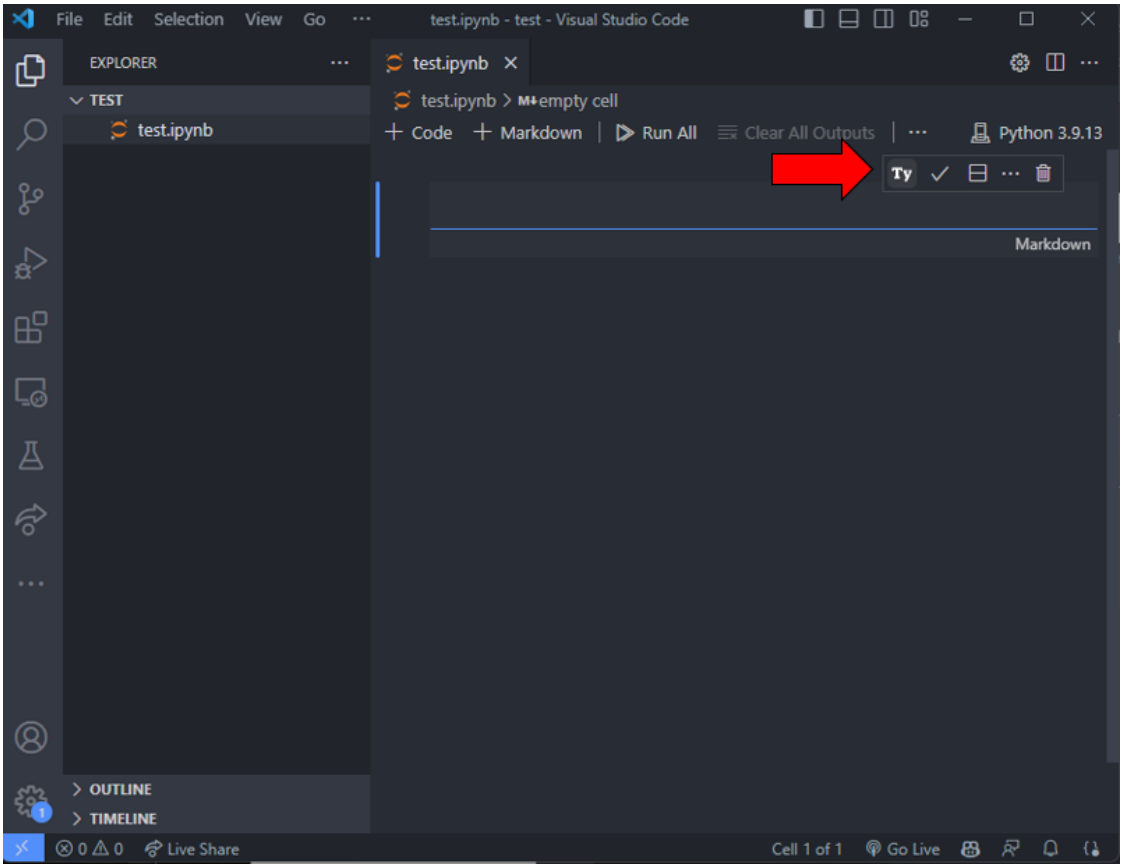


Figure 4.2: Typhon button in Visual Studio Code

new code cell with the code with the highest score or to show the list of multiple related codes as a side panel shown in Figure 4.4. Moreover, the user can select the data tier and model for searching in the extension setting. The data tier is the selection of notebooks dataset scope, which uses the same tier list as Kaggle, e.g., Grandmaster, Master, and Expert. Typhon comes with two models, Information Retrieval (BM25) and Machine Learning as shown in Figure 4.5.

4.3.2 Stemming and Lemmatization Service

Stemming and Lemmatization is the preprocessed module for enhancing BM25 written with Python and using FastAPI to provide this module as an external service. Stemming and Lemmatization service will be used when the user selects the “Run Typhon with Stemming and Lemmatization” feature on the extension.

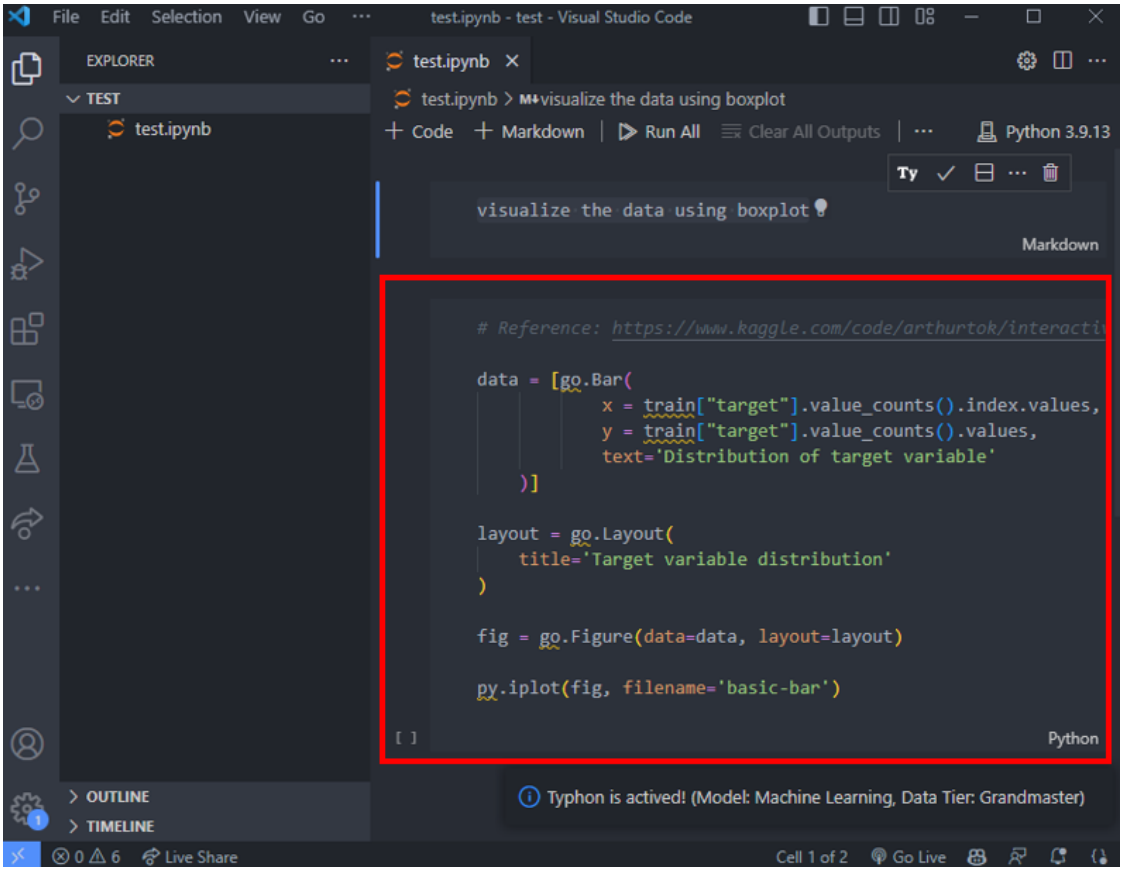


Figure 4.3: Recommended code cell from Typhon after clicking the Typhon button

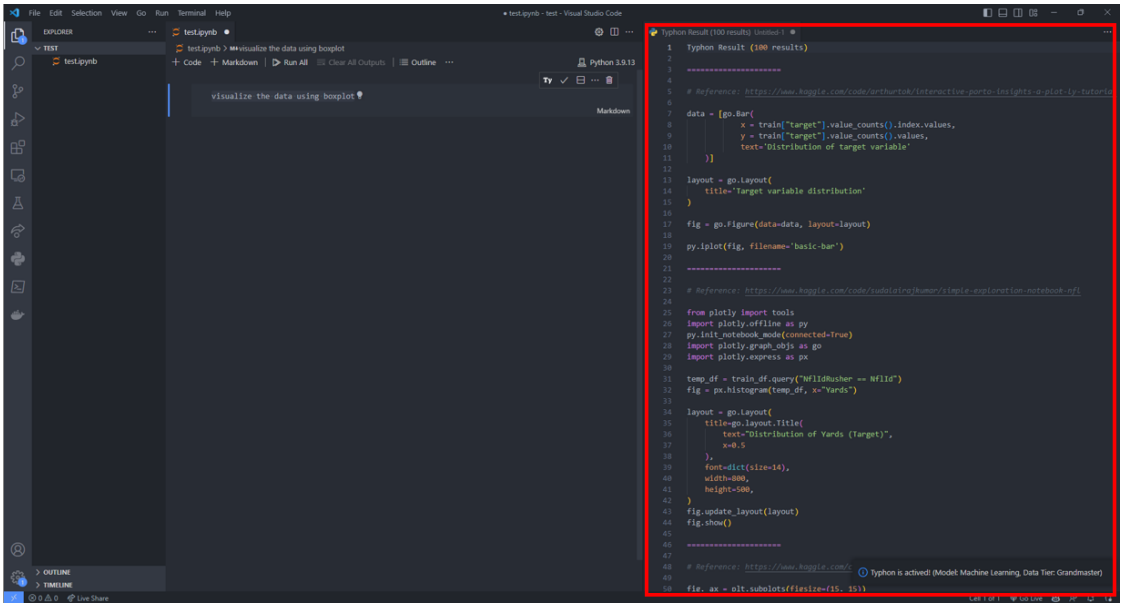


Figure 4.4: Side panel opened by Typhon

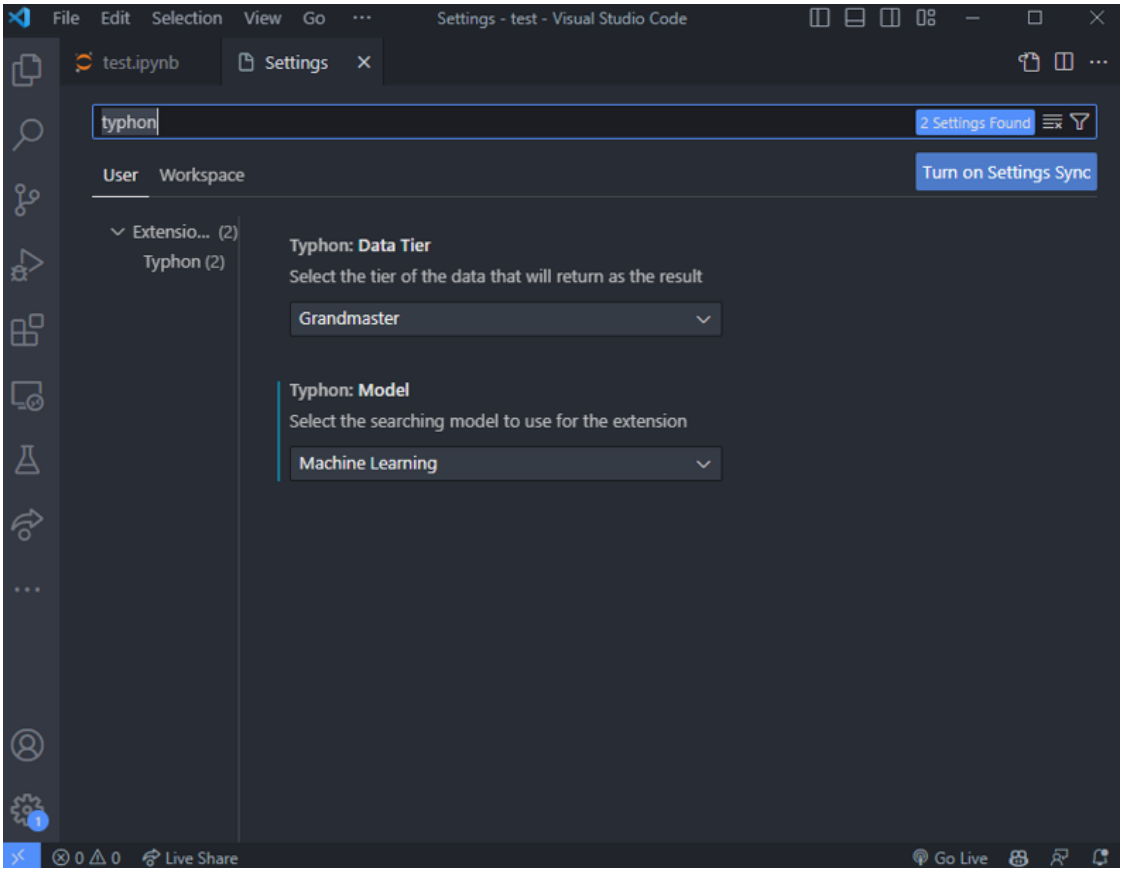


Figure 4.5: Setting window of Typhon

4.3.3 Middle API

We created a middle application programming interface (API) that acts as a router using Node.js. This API is mainly used as a switch for database selection according to the preferred model from the interface. Moreover, the minimum threshold, like hit score and certainty also set on this layer. This API is the middle layer and bridge between the interface (Visual Studio Code Extension) and databases.

4.3.4 Databases

Typhon has two main databases to support two models, Information Retrieval and Machine Learning. Two models use different techniques, so we need to handle them differently. We use Elasticsearch for the Information Retrieval technique. Elasticsearch uses BM25 as a default algorithm for searching indexed data. When the markdown description is sent and forwarded from the interface via the middle API, Elasticsearch will compare the sent markdown description with the indexed markdown description and then return the related code that pairs with the indexed markdown.

4.4 Markdown-Code Cell Matching and Recommendation Techniques

The following sections briefly explain the techniques being used for recommending code in the Typhon system.

4.4.1 Information Retrieval Technique: BM25

For Elasticsearch data indexing, the Python script was used to upload the data that we processed from the raw data of KGTorrent by mapping the markdown cell and code cell next to the markdown cell(s) to be a pair via Python script. The uploading Script requires the IP Address of Elasticsearch, the index name, and a JSON file containing the data. The data will upload to Elasticsearch in the same index as the index name that we input.

Subsequently, the Middle API employed the user's Markdown as a query to retrieve comparable Markdown pairs from Elasticsearch. The default setting utilized by Elasticsearch, which is the BM25 ranking function, facilitated the retrieval of a list of search outcomes ranked based on the significance of each Markdown pair, where the

flow of the process is depicted in Figure 4.6. The code snippet presented below depicts a segment of the code utilized in the Elasticsearch server.

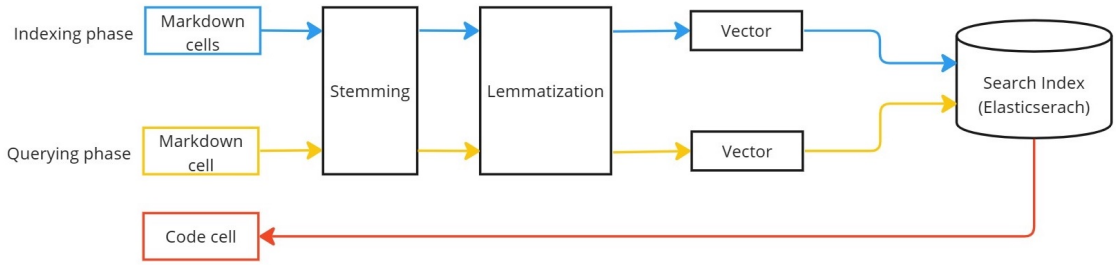


Figure 4.6: Operation flow of BM25 approach

The code snippet presented in Figure 4.8 depicts a segment of the code utilized in the Elasticsearch server.

4.4.2 Machine Learning: UniXcoder

UniXcoder is a machine learning model that is trained on both natural language and programming languages, which is developed by Microsoft. It is a pre-trained model developed in the CodeBERT series as a unified cross-modal model that specializes in code understanding tasks [11].

Machine learning will be utilized to find the similarity between query input and the collection of code where the similarity-finding operations in machine learning approach are encapsulated within the Weaviate system. Therefore, the Weaviate system is a vector database engine that accommodates all the store and gets operations. Once the Weaviate system is ready, the Typhon extension will utilize it to obtain the closest match code in the code base for a recommendation. The process is as depicted in Figure 4.9

The code snippet presented in Figure 4.10 depicts a segment of the code utilized in the UniXcoder Server API and the code snippet presented in Figure 4.11 depicts a segment of the code utilized in the Weaviate Server.

4.4.3 Preliminary Evaluation The Recommended Code Cells using BM25

The following section is the evaluation of the prototype of the recommendation algorithm, which implements the algorithm using the BM25 ranking function. This experimental setup is similar to the previous preliminary test with GitHub Copilot. Additionally, notebooks present in the initial experiment were removed from this experiment

```
from elasticsearch import Elasticsearch, helpers
import ndjson
import argparse
import uuid

es = Elasticsearch("<IP Address>")

parser = argparse.ArgumentParser()
parser.add_argument('--file')
parser.add_argument('--index')

args = parser.parse_args()

index = args.index
file = args.file


with open(file) as json_file:
    json_docs = ndjson.load(json_file)

def bulk_json_data(json_list, _index):
    for doc in json_list:

        if '{"index"' not in doc:
            yield {
                "_index": _index,
                "_id": uuid.uuid4(),
                "_source": doc
            }

try:
    # make the bulk call, and get a response
    response = helpers.bulk(es, bulk_json_data(json_docs, index))
    print ("\nRESPONSE:", response)
except Exception as e:
    print("\nERROR:", e)
```

Figure 4.7: Easticsearch Loader

corpus to ensure that the algorithm works without bias, suggesting its notebook. Also, it is important to mention that two notebooks, *Titanic Data Science Solutions* and *Time series Basics: Exploring traditional TS* were removed from evaluation subjects because those competitions did not exist in the KGTorrent dataset. The possible explanations were that the Time series Basics: Exploring traditional TS competition was organized too recently and therefore did not exist at the time KGTorrent gathers data. As for the Titanic Data Science Solutions competition, it was possible that the competition had not close yet, and therefore did not present in the dataset. Hence, the three notebooks used in the evaluation are shown in the Table 4.1

```
query: {
    match: {
        markdown: {
            query: <requested query>
        }
    }
}
```

Figure 4.8: JSON query of Elasticsearch

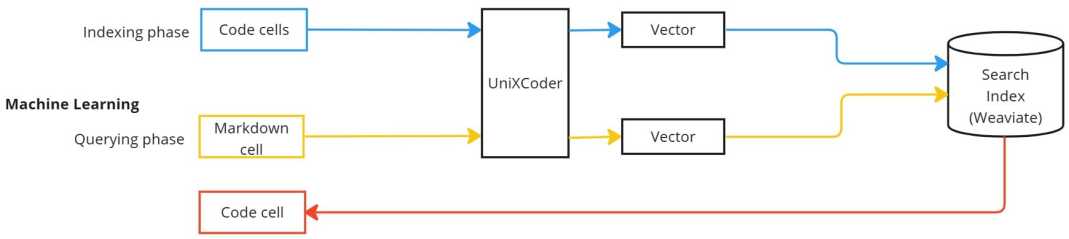


Figure 4.9: Operation flow of machine learning approach

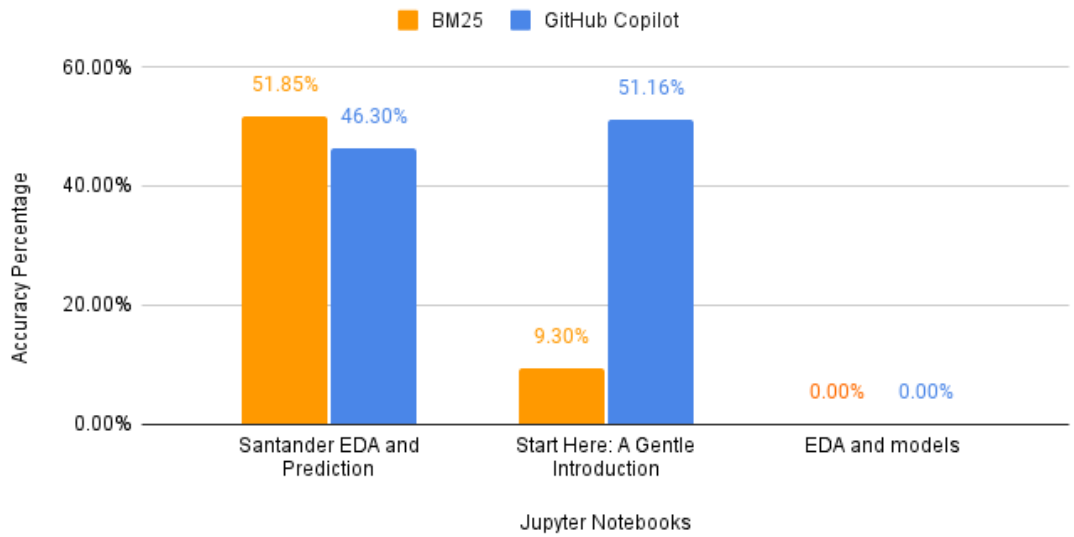


Figure 4.12: Experimental recommendation result from baseline BM25 algorithm in comparison with GitHub Copilot

Table 4.1: Three selected Jupyter notebooks for evaluating baseline BM25 algorithm suggestions

Notebook	No. of votes
Start Here: A Gentle Introduction	3,372
Santander EDA and Prediction	1,215
EDA and models	1,096

```python
from fastapi import FastAPI, Response, status
import os
from typhon_lite import Typhon
app = FastAPI()

model = Typhon()

from pydantic import BaseModel

class VectorInput(BaseModel):
    text: str


@app.get("/.well-known/live", response_class=Response)
@app.get("/.well-known/ready", response_class=Response)
def live_and_ready(response: Response):
    response.status_code = status.HTTP_204_NO_CONTENT


@app.get("/meta")
def meta():
    # return meta_info.get()
    return {
        "name": "Typhon",
        "lang": "en"
    }


@app.post("/vectors")
async def read_item(item: VectorInput, response: Response):
    try:
        vector = model.embedding([item.text])
        return {"text": item.text, "vector": vector, "dim": len(vector)}
    except Exception as e:
        response.status_code = status.HTTP_500_INTERNAL_SERVER_ERROR
        return {"error": str(e)}
```

Figure 4.10: UniXcoder Server API

```yaml
---
version: '3.4'
services:
  webserver:
    image: webserver
    build:
      context: .
      dockerfile: nginx.Dockerfile
    ports:
      - 81:80
  weaviate:
    command:
      - --host
      - 0.0.0.0
      - --port
      - '8088'
      - --scheme
      - http
    image: semitechnologies/weaviate:1.17.5
    hostname: weaviate
    restart: on-failure:0
    environment:
      TRANSFORMERS_INFERENCE_API: 'http://t2v-transformers:8088'
      QUERY_DEFAULTS_LIMIT: 25
      AUTHENTICATION_ANONYMOUS_ACCESS_ENABLED: 'true'
      PERSISTENCE_DATA_PATH: '/var/lib/weaviate'
      DEFAULT_VECTORIZER_MODULE: 'text2vec-transformers'
      ENABLE_MODULES: 'text2vec-transformers'
      CLUSTER_HOSTNAME: 'node1'
  t2v-transformers:
    image: text2vec-typhon
    build:
      context: .
      dockerfile: Dockerfile
    environment:
      ENABLE_CUDA: '0'
```

Figure 4.11: Weaviate configuration

After the experiment, the accuracy result from the recommendation system is shown inFigure 4.12, notebook from Santander Customer Transaction Prediction competition had an accuracy of 51.85 percent, the one from Home Credit Default Risk competition had an accuracy of 9.30 percent, and the one from IEEE-CIS Fraud Detection competition had 0.00 percent accuracy.

Although the notebook from the Santander Customer Transaction Prediction competition could achieve the highest score, after further investigation of the source notebook, it was found that the author of that notebook had duplicated the subject notebook to extend their code. However, the majority of the code cells and Markdown description cells remained the same; therefore, the corpus coincidently contained the notebook with cells mostly the same as the subject notebook.

While evaluating the other two notebooks, we found that the baseline BM25 ranking function algorithm could not provide the code cells to meet the satisfactory criteria, which were mentioned in the Evaluation of code suggestion part in the 3.1.1 Methodology section. The suggested code cells from the BM25 algorithm were almost irrelevant to the provided description. Even if the recommended code cells contained some similarity to the provided description, only a few lines of code were similar to the description, and the overall suggested cells could not provide satisfactory results most of the time.

With the previous duplication incident in mind, it could explain why the result from the other two notebook subjects was low. Therefore, it indicated that the highest similarity score from the BM25 retrieval did not indicate any relationship between the input Markdown and the result from retrieval. Furthermore, the input length showed that it did not increase the efficiency of BM25 in this experiment; thus, the result retrieval with the highest similarity score might not always have been the best answer. Therefore, the user could have chosen the alternative suggestion from the list of similar code cells instead of the code cell with the top similarity score.

CHAPTER 5
EVALUATION

In this chapter, we discuss the evaluation of Typhon in terms of usability and usefulness using online questionnaires and user interviews. The user study is conducted by following the Mahidol Central Institutional Review Board (IRB) guideline for performing system experimentation with a group of testing participants. The request for this evaluation was approved on 25 April 2023. However, due to a conflicting timeline, during development, the evaluation was required to perform prior approval of IRB.

5.1 Objectives of the Evaluation

The objectives of the evaluation are as follows.

1. To evaluate the accuracy of Typhon's recommended code cells.
2. To evaluate the usefulness of the Typhon extension.
3. To evaluate the usability of the Typhon extension.
4. To evaluate the user experience while experimenting with the Typhon extension.

5.2 Methodology

The following section describes the methodological detail of how the evaluation was conducted, the participant inclusion criteria, and the questionnaire in detail.

5.2.1 Preparation

Before starting the evaluation process, the codebase data of both models were prepared and organized into three groups based on their corresponding ranks. The approximate number of code-markdown pairs in each rank group were as follows: 2,670 pairs for Rank Grandmaster, 4,098 pairs for Rank Master, and 10,855 pairs for Rank Expert. These numbers are approximate because some pairs were discarded during the

indexing operation in the Machine Learning model. The data in the codebase was indexed from the KGTorrent dataset by mapping raw Markdown-code cells and filtering for only code-markdown pairs related to plotting and charts for reducing the scope of the evaluation. Furthermore, in the Typhon Visual Studio Code extension, the default configuration of the model is set to BM25 or Elasticsearch, with the default rank being Grand Master.

5.2.2 Accuracy of Typhon's recommended code cells

The methodology for evaluating the accuracy of Typhon involves recommendation Matplotlib plots. First, we studied the available plot types from the Matplotlib website. This is followed by creating a template query text for all the plot types, which is shown in Table 5.1. The next step is to start querying each suggested code and determining whether the suggested code is related to the query chart type.

One of the authors who has a background in Data Science is tasked with comparing the suggested code with the query chart type.

If the suggested code is related to the query chart type, then the accuracy of the experimenting chart is reported as true. On the other hand, if it is not related, the accuracy of the experiment chart is reported as false. This process is carried out for all the plot types on the Matplotlib website.

Finally, after completing querying recommendations for all plot types, the results are used for evaluating the accuracy of recommendations by conducting methods for accuracy testing. This process aims to determine the accuracy of recommendations given the context of plot types available in Matplotlib library, a popular visualization creation library, and ensure the system can reliably and accurately suggest the relevant code from the codebase.

5.2.3 Participants and Procedure

We follow the following steps to recruit the participants and perform the evaluation.

Table 5.1: Matplotlib visualization types and the associated query terms

Plot type		Query term
Basic	Scatter	plot data using scatter visualization
	Bar	plot data using bar visualization
	Stem	plot data using stem visualization
	Step	plot data using step visualization
	Fill_between	plot data using fill_between visualization
	Stackplot	plot data using stackplot visualization
Plots of Arrays and Fields	Imshow	plot data using imshow visualization
	Pcolormesh	plot data using pcolormesh visualization
	Contour	plot data using contour visualization
	Contourf	plot data using contourf visualization
	Barbs	plot data using barbs visualization
	Quiver	plot data using quiver visualization
	Streamplot	plot data using streamplot visualization
Statistics Plots	Hist	plot data using hist visualization
	Boxplot	plot data using boxplot visualization
	Errorbar	plot data using errorbar visualization
	Violinplot	plot data using violinplot visualization
	Eventplot	plot data using eventplot visualization
	Hist2d	plot data using hist2d visualization
	Hexbin	plot data using hexbin visualization
	Pie	plot data using pie visualization
Unstructured Co-ordinates	Tricontour	plot data using tricontour visualization
	Tricontourf	plot data using tricontourf visualization
	Tripcolor	plot data using tripcolor visualization
	Triplot	plot data using triplot visualization
3D	3D Scatterplot	plot data using 3D scatterplot visualization
	3D Surface	plot data using 3D surface visualization
	Triangular 3D Surface	plot data using triangular 3D surface visualization
	3D Voxel , Volumetric Plot	plot data using 3D voxel , volumetric plot visualization
	3D Wireframe Plot	plot data using 3D wireframe plot visualization

Participant Inclusion Criteria

The authors set up inclusion criteria for selecting the participant The authors select the participants for evaluating Typhon according to the following criteria.

1. The person must have experience in Python programming for at least a year.
2. The person must have experience developing a data science project using Jupyter Notebook for at least a year.

Evaluation Procedure

The following steps are a step-by-step methodology used to conduct the interview and experiment for evaluating Typhon's usefulness, usability, and user experience.

Table 5.2: Questions Asked in the Survey

No.	Question	Method	Minimum Value	Maximum Value
1	How easy is it to use the tool?	5-point Likert Scale	Very hard to use (0)	Very easy to use (5)
2	How satisfied are you with the speed of the tool?	5-point Likert Scale	Very dissatisfied (0)	Very satisfied (5)
3	How satisfied are you with the response of the tool?	5-point Likert Scale	Very dissatisfied (0)	Very satisfied (5)
4	Do you agree that the tool saves your time compared to other tools you have used?	5-point Likert Scale	Strongly disagree (0)	Strongly agree (5)
5	Overall, how satisfied are you with the tool?	5-point Likert Scale	Very dissatisfied (0)	Very satisfied (5)
6	Was there any confusion, struggle, or issue while using the tool?	Open-ended Question	None	None
7	Were there any features that slowed you down or hindered your progress?	Open-ended Question	None	None
8	How does this tool compare to other similar tools you have used?	Open-ended Question	None	None

1. The interviewer asks the participants to introduce their names and their current profession.

2. The interviewer tells the origin, purposes, expected outcomes, and goals of the project, as well as the limitation of Typhon.

3. The interviewer demonstrates the project installation and instructions, covering installation, program configuration, and usage.

4. Upon finishing the demonstration, the participant is assigned a task to experience the Typhon program on their own for no more than 10 minutes.

5. After the participant completes the task, the interviewer asks for comments on what the participant finds or feels during the experiment.

6. Finally, the participant completes a survey via Google Forms.

Questionnaire Detail

The following Table 5.2 describes the questions that were asked in during the interview process. In the table, each row describes the question, the scoring method, the minimum and maximum value.

Table 5.3: Sanity-check result

Rank	Type	Total Correct	Total Correct (%)	Total Items
Grand Master	UniXCoder	254	10.01	2517
	BM25	2399	95.31	2517
	BM25 + stemming and lemmatization	2132	84.72	2517
Master	UniXCoder	377	10.07	3744
	BM25	3391	90.57	3744
	BM25 + stemming and lemmatization	3007	80.32	3744
Expert	UniXCoder	605	6.33	9553
	BM25	8644	90.48	9553
	BM25 + stemming and lemmatization	7193	75.30	9553

5.3 Experimental Result

The next section describes the evaluation results from evaluating the accuracy and sanity testing. The authors also evaluate the usefulness, usability, and user experience of the questionnaire.

5.3.1 Accuracy of Typhon's recommended code cells

We evaluated the accuracy of Typhon's code cell recommendations as follows.

Sanity check

Prior to the accuracy testing evaluation, the sanity-checking test aims to prove that the Typhon model can provide recommendation functionality, at the very least, as correct as it should be. The checking steps are in the following order. Firstly, each code-Markdown pair in the code base is used to get the recommendation for each using the same codebase. Then, those recommended codes for each model, machine learning approach (UniXCoder), BM25, and BM25 with the pre-processed query using stemming and lemmatization are compared with the original code from the query code-Markdown pair.

During the sanity check, the data used for both the databases and queries were the same, therefore they are identical. This data consisted of code-markdown pairs specifically related to plotting and charts. In this process, the associated Markdown in each code-markdown pair was used as the query statement for the Typhon extension. Since the data used for querying was the same as the indexed data, there were 2,670 code-markdown pairs in Rank Grandmaster, 4,098 pairs in Rank Master, and 10,855 pairs in

Rank Expert.

Our assumption expects that querying with the query exists in the codebase; the recommendation result should return the exact copy of the original code in the query code-Markdown pair. Thus, the sanity-check result is shown in the Table 5.3. The result comprises 3 primary ranks, Grand Master, Master, and Expert. Each level contains 3 types of the recommendation model approach available for use in the Typhon tool, with the Total Correct, Total Correct (%), and Total Items columns, explaining the total number of correct recommendations, the total number of items in the codebase, and the percentage of correct items.

The following points analyze the result in each rank.

1. In the Grand Master rank, UniXCoder has the least correct recommendation, with 254 items or 10.01 percent. The correct recommendation is the BM25 model, with 2,399 items, or 95.31 percent. In comparison, BM25 with stemming and lemmatization has the correct recommendation of 2,132 items or 84.72 percent.

2. In the Master rank, the ranking are the same; the machine learning has the least correct recommendation items, the BM25 model is the highest, and BM25 with a pre-processed query in the middle. Where the UniXCoder has 377 items or 10.07 percent, BM25 has 3391 items or 90.57 percent, and BM25 with pre-processed query has 3,007 items or 80.32 percent.

3. In the Expert rank, the rank is also the same. UniXCoder has the least correct recommendation items, the BM25 model the highest, and BM25 with the pre-processed query in the middle. Machine learning has 605 items or 6.33 percent, Elasticsearch has 8,644 items or 90.48 percent, and Elasticsearch with pre-processed query has 7,193 items or 75.30 percent.

The low recommendation score in the UniXCoder, a machine learning model approach, could be explained by considering the underlying machine learning model, UniXcoder model, which was a cross-modal model that contained only code cells data in the codebase and returned recommendations by processing a natural language query with the code in the codebase. Unlike Elasticsearch, which have pairs of Markdown

cell and code cell for cross-referencing. Therefore, it is considered acceptable that the recommendation score in the machine learning model is noticeably low.

On the contrary, the score in BM25 and BM25 with pre-processed queries using stemming and lemmatization is significantly higher, which could explain by considering how this approach recommends the code by comparison of Markdown text. The BM25 approach suggests the code by comparing the natural language in the query with the natural language in the Markdown in the objects stored in the Elasticsearch codebase.

With further investigation of the BM25 recommendations that were wrong, the authors found that there were several issues that made the BM25 approach recommend different codes, even though the associated Markdown was used for querying already existed in the codebase. The following section discusses the topics found in the problem.

1. **General used words**

   The query Markdowns are not scoped into any particular topic or context. These Markdowns are short in length and convey very broad meaning. Examples of these Markdowns are ”Other” and ”Worldwide”.

2. **Few keywords or short phrase Markdown**

   The query Markdown consists of cases where the Markdown used for querying contains only a few keywords or short phrases. While the Markdowns in the recommendation are often longer and contain more of the keywords from the original Markdowns.

3. **Reused and duplicated Markdowns**

   There are many cases in which the author reuses and replicates the same Markdown multiple times, both in the same notebook and in different notebooks. There are also cases where different authors use the same Markdown in their own notebooks, which contain different codes. Often, the Markdown in such cases is short or has few keywords.

4. **General inaccurate recommendation**

   Various common inaccurate results can be noticed from the results, in which there were cases where the query Markdown and recommendation Markdown shared the

same keywords that are often unique among notebooks, or significantly common that can be found in any notebook.

From the test, it is clearly shown that clear and detailed instructions in the Markdown can directly improve code recommendation quality and improve the likelihood of generating relevant and accurate suggestions.

However, while the BM25 recommendation approach provides valuable insights and potential code recommendations, it is important to note that the algorithm may not always capture the specific context or subtle requirement in the Markdown query. Human experts are still required to manually intervene to verify and fine-tune the recommended code to ensure that actually relevant recommended codes are selected from the recommendation.

To enhance the BM25 approach and address the observed issues, further research and refinement of the recommendation algorithm, as well as incorporating additional contextual information from the Markdown, could potentially lead to more accurate and reliable code suggestions. We leave this as future work.

Accuracy Evaluation Result

The result from measuring the accuracy of Typhon, which aims to test the recommendation for various Matplolib chart visualizations, is shown in Table 5.4, where it included 14 charts types collected from the Matplotlib library [1] websites.

Table 5.4 contains the **Plot Type** column, 3 rank-level columns; **Grand Master, Master, and Expert**, and 2 sub-type for model columns; **Machine Learning and BM25**. At the Grand Master rank, the machine learning model has the capability to suggest six code snippets that are related to the charts and 3 code snippets for BM25. As for the Master rank, the Machine Learning approach can recommend 7 related code snippets, and the BM25 approach can recommend for 4 recommendations. Finally, for the Expert rank, the Machine Learning approach can recommend 9 related code snippets and 7 related code snippets for the BM25 approach. However, it is essential to mention that the number of code-Markdown pairs in the codebase is inconsistent due to the limited amount of available data in the dataset and filtering process. Where the Expert rank has

[1] https://matplotlib.org/stable/plot_types/index

the most data, with approximately 10,800 pairs of code and Markdown; the Master rank with approximately 4,090 pairs; and Grand Master rank with approximately 2,660 pairs. Therefore, the present imbalance could explain the noticeably slightly higher accuracy in Master and Expert, where they can potentially recommend a more variety of code due to more available data.

Furthermore, the charts presented in Table 5.4 show that there are some charts that rarely appeared in the recommendation results. For instance, **the contour, hist, errorbar, and triangular 3d surface** only appear once in the recommendation result. Additionally, considering the nature of the dataset, which is extracted from the competitive notebooks of Kaggle competition, there is a high possibility that the codebase contains some group of visualizations significantly more than other groups of visualization. Therefore, it explains why some charts appear more often than others appear in the recommendation.

Table 5.4: Typhon's first recommendation that contains the code cells related to the given visualization type.

	Grand Master		Master		Expert	
Plot Type	Machine Learning	BM25	Machine Learning	BM25	Machine Learning	BM25
scatter	✓	✓	✓	✓	✓	✓
bar	✓		✓	✓	✓	✓
step	✓				✓	✓
imshow	✓		✓			✓
contour					✓	
hist						✓
boxplot		✓			✓	✓
errorbar			✓			
violinplot				✓	✓	
pie	✓		✓	✓		
tripcolor	✓					
3d scatterplot		✓	✓		✓	✓
3d surface			✓		✓	
triangular 3d surface					✓	
Total Correct	6	3	7	4	9	7

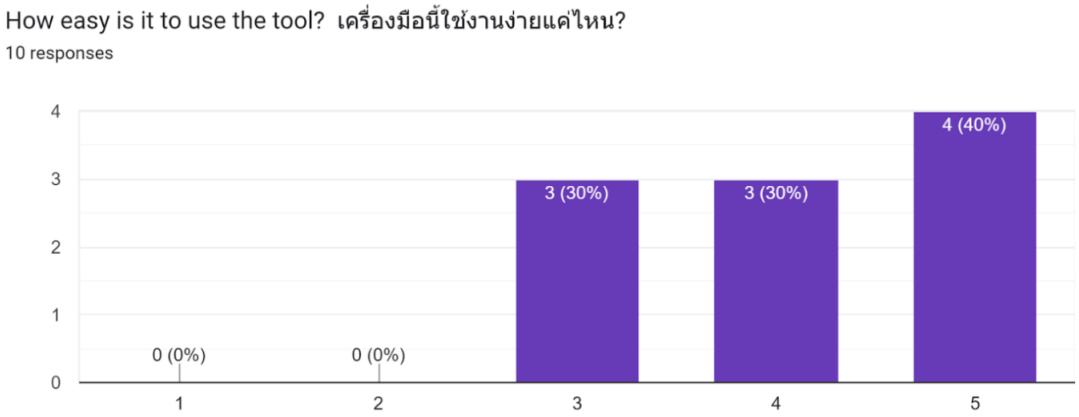


Figure 5.1: Ease of use

5.3.2 Usefulness, usability, and user experience of Typhon

The following sections explain the results of user evaluation in both 5-Likert scale questions and open questions. The evaluation was conducted on 10 participants who have worked with Python and Jupyter Notebook for at least 1 year of experience. The list of participants and their occupations are shown in Table 5.5.

Table 5.5: Participants in the interviews

Participant	Occupation	Experience (years)
P1	Master degree Student	3
P2	Master degree Student	4
P3	Product Owner	1
P4	Technology Lead	1
P5	Researcher	3
P6	DeFi Researcher	1
P7	Data Science Researcher/Lecturer	7
P8	Data Science Researcher/Lecturer	5
P9	Data Scientist/Data Engineer	8
P10	Data Scientist	3

1. **Question 1: How easy is it to use the tool?**

   From the evaluation and feedback, over 70 percent of participants agreed that the Typhon tool provided high usability and was easy to use, while the other 30 percent of participants thought that Typhon's usability and ease of use were around average as depicted on Figure 5.1.

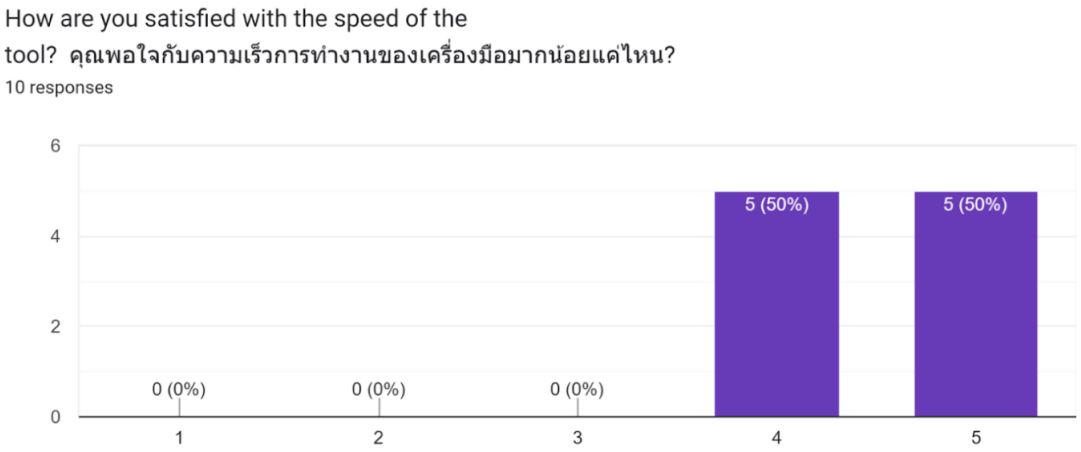


Figure 5.2: Speed of the tool

2. **Question 2: How are you satisfied with the speed of the tool?**

   The evaluation result on Figure 5.2 describes the satisfaction regarding the time for delivering the result from Typhon extension. Of all participants, 50 percent are satisfied, and 50 percent are very satisfied with the response time of the Typhon tool. Therefore, it indicates that the Typhon tool can offer a fast and quick response to users.

3. **Question 3: How are you satisfied with the response of the tool?**

   The Figure 5.3 shows satisfaction with the quality of the response from the tool. The chart explains that 40 percent are dissatisfied, 30 percent are moderate, and 30 percent are satisfied. Therefore, the chart indicates that the Typhon tool can somewhat satisfy the participants and still needs further improvement in its performance and efficiency for recommending code related to the user's query.

4. **Question 4: Do you agree that the tool saves your time compared to other tools you have used?**

   The resulting chart from the evaluation depicted on Figure 5.4 describes that 40 percent are dissatisfied, 30 percent are moderate, and 30 percent are satisfied. The result explains that the Typhon tool may not be able to save the participants time during their development as expected in the project criteria, which leads to similar thoughts on Question 3 that the Typhon tool will need further improvement to

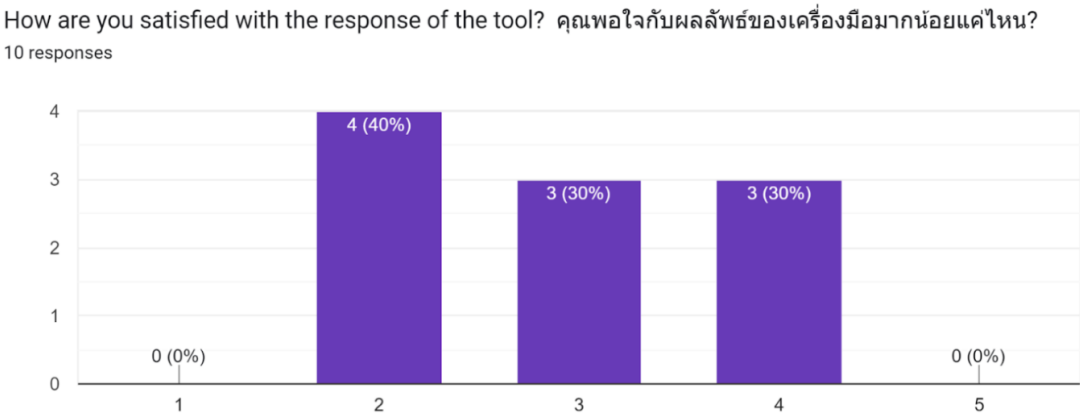


Figure 5.3: Satisfaction with the tool

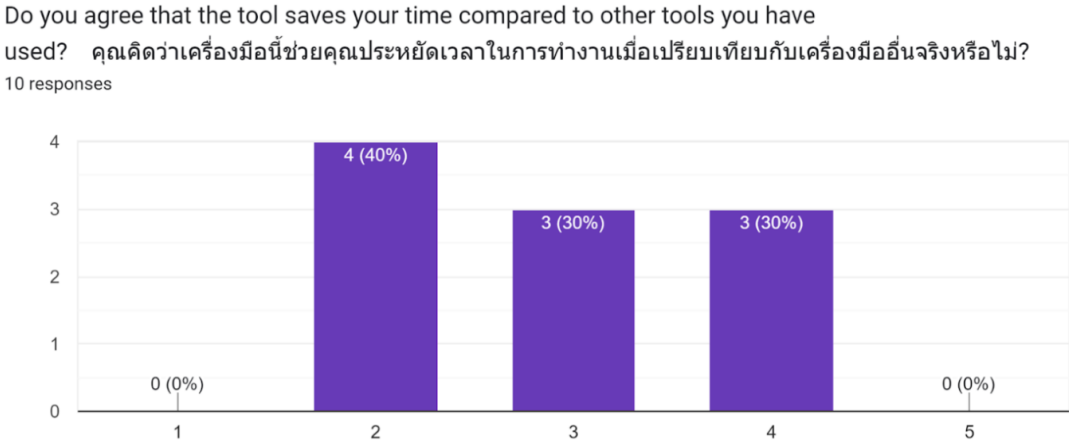


Figure 5.4: Saving time

enhance its capability to recommend a better code snippet for users to make use of.

5. **Question 5: Overall, how are you satisfied with the tool?**

The Figure 5.5 shows participants' overall satisfaction with the Typhon tool. The chart tells that 70 percent are moderate and 30 percent are satisfied. This can indicate that all participants are satisfied with the Typhon tool on a moderate score, and during interviews, the majority of participants find the core idea of Typhon interesting, and the delivery approach might be changed to satisfy their idea.

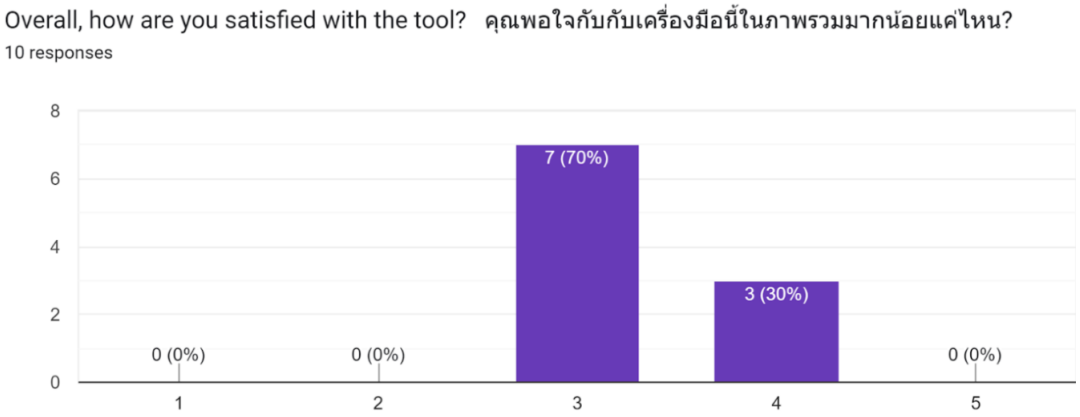


Figure 5.5: Overall satisfaction

5.3.3 Evaluation Result From Opened-question

6. **Question 6: Was there any confusion, struggle, or issue while using the tool?**
   The survey responses generally relate to the user experience of using a programming tool, with several issues reported, such as difficulty with installing and configuring the tool, struggling with context understanding, and suggested code not running or lacking awareness of context. However, some respondents found the tool to be user-friendly, with some suggestions for improvement, such as implementing a shortcut key for code suggestions to enhance workflow. Overall, while some users experienced issues, the general consensus was that the tool was easy to use, with some room for improvement.

7. **Question 7: Were there any features that slowed you down or hindered your progress?**
   The survey responses are generally related to features that slowed down or hindered progress while using a programming tool. One respondent reported difficulty with context switching between models, while others reported issues with suggested code that either recommended pre-defined functions they didn't have or code that couldn't run without finding the necessary library. Additionally, some users found that the suggested code was not useful, requiring extra effort to find the best code from numerous results, or was too long and contained irrelevant parts, making it time-consuming to read and understand. However, one user re-

ported that they didn't encounter any such issues. Overall, the survey responses suggest that some features within the tool can cause slowdowns or hinder progress, particularly when dealing with suggested code.

8. **Question 8: How does this tool compare to other similar tools you have used?** In comparing this tool to other similar tools, it has limitations in the suggestion of data as it is limited to the Kaggle database. However, it is quicker to search for code than using Google search and is useful for starting with Python visualization. Although it may not be as intuitive as other tools, it is still good for general use. Some users find it inferior to GitHub Copilot and ChatGPT, which may outperforms the Typhon tool with their understanding of the user's context. Additionally, the suggested code can be lengthy and may not be entirely relevant, making it time-consuming to find the best code for a task. Overall, this tool is convenient as it is integrated into VS Code, but the results may not be as accurate as ChatGPT or as customizable as GitHub Copilot.

5.4 Discussion

The Typhon tool received mixed feedback from users regarding its usability and ease of use, with participants agreeing that it provided high usability and was easy to use. However, the tool's ability to save time during development and the quality of the response needs further improvement. Some users experienced issues with suggested code, such as irrelevance and time-consuming search, and the tool has limitations in suggestion compared to GitHub Copilot and ChatGPT. Although Typhon offers fast response times, it may not be as intuitive or accurate as other tools. Overall, the tool is convenient but requires improvement, with reported issues including difficulties with installation and configuration, understanding context, and limitations in data suggestion and relevant code recommendations.

CHAPTER 6
CONCLUSIONS

This chapter summarizes the project and discusses the limitations and future directions of this project.

6.1 Problems and Limitations

During the project, we found the following problems and limitations.

1. In the beginning, the development team aimed to use clustering techniques to improve the accuracy of code recommendations. However, after the experimentation, the development team found this technique unsuitable for the current use case.

2. Development team encountered deployment issues when deploying the application on a cloud system due to a lack of familiarity with the system and network-related tasks.

3. Developing extensions for Visual Studio Code was challenging because the documentation on developing extensions for Viusal Studio Code was limited and incomplete.

6.2 Future Work

We plan the following items as our future work.

1. We plan to make the code recommendation extension readily available to users by publishing it on the Visual Studio Code extension marketplace.

2. We plan to incorporate the feature to set a minimum accuracy threshold for the machine learning search algorithm. This feature enables users to determine how closely the search results should match the query, thus providing greater control over the search results.

3. We plan to introduce additional customization settings to the extension to enable users to customize the tool to their specific needs

4. We plan to add more data to the extension, increasing the diversity and possibility of search results.

5. We plan to consider looking for a new method to process data and reduce noise in the recommended code.

6.3 Conclusion

In conclusion, code recommendation tools have become a big part of many software developers. Tools like GitHub Copilot and ChatGPT have become increasingly favored both in and outside the industry. There are also studies in software engineering research, such as Aroma, Strathcona, Senatus, and Example Overflow, which aim to improve further the quality and performance of the process of code recommendation.

Our proposed alternative experimental tool, Typhon, is an extension of Visual Studio Code created to tokenize Markdown cells for the developer to search similar Code cells using both BM25 (Information Retrieval Approach) and UniXcoder (Machine Learning Approach). Typhon is designed to improve the workflow of those in the Data Science field, primarily on Jupyer Notebook and Visual Studio Code IDE.

Currently, the Typhon extension offers benefits in terms of speed and a highly simplified interface. Although Typhon cannot perform code recommendations comparable to GitHub Copilot or ChatGPT, the evaluation shows that Typhon can still recommend multiple codes as references for developers. Nevertheless, there are notable issues exhibited in Typhon, such that the recommended code will not be aware of the existing libraries in the user's local machine, and the recommended code does not provide the libraries used within the recommended code snippet. Therefore, it shows that Typhon still has room for further improvement.

APPENDIX A
TYPHON: USER MANUAL

Typhon Manual

1

What is Typhon?

Typhon is an extension of Microsoft Visual Studio Code built for the data science field in Jupyter Notebook with Python programming language.

Video Demo

Click this Link > http://bit.ly/3ZWWusR

Typhon code recommendation method

- **BM25** is an information retrieval technique for finding similar Markdown cells for finding similar Code cells.
    - **Stemming and Lemmatization**

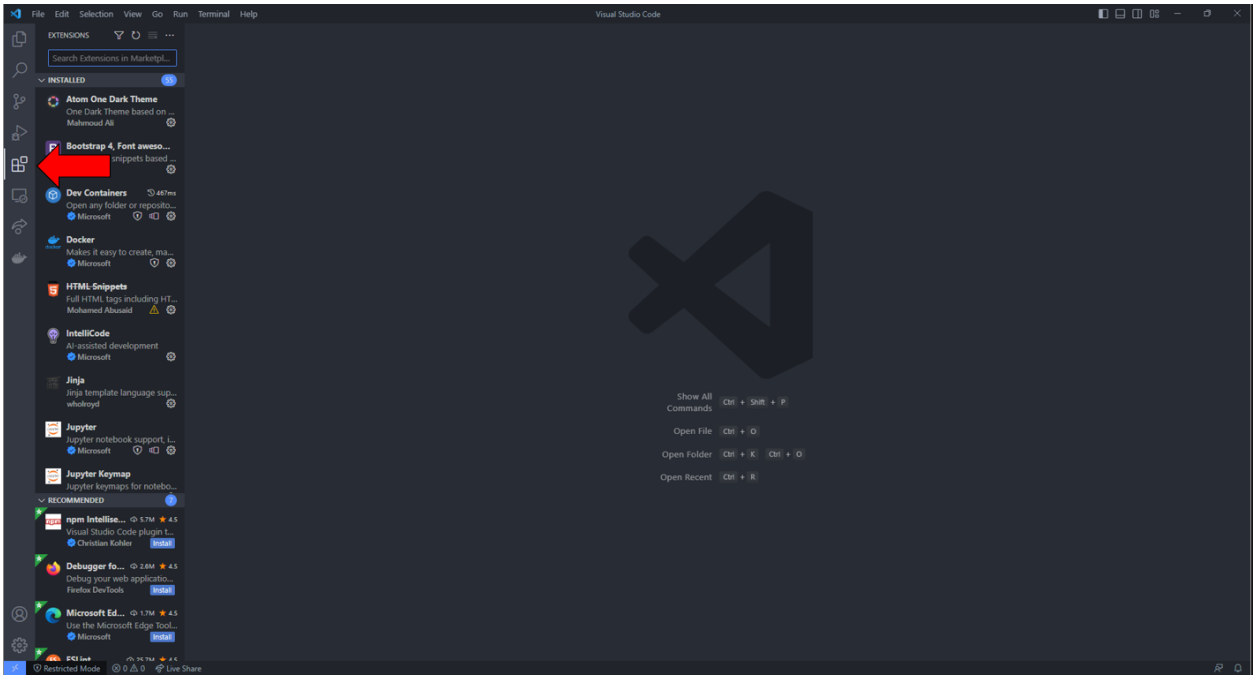
      Technique to reduce word form to a simple version

        - Stemming - reduce words down to its word stems
        - Lemmatization - reduce words to their root words
- Machine Learning - UniXcode is a machine learning technique that uses the UniXcoder model, which is trained to find programing language from natural language.

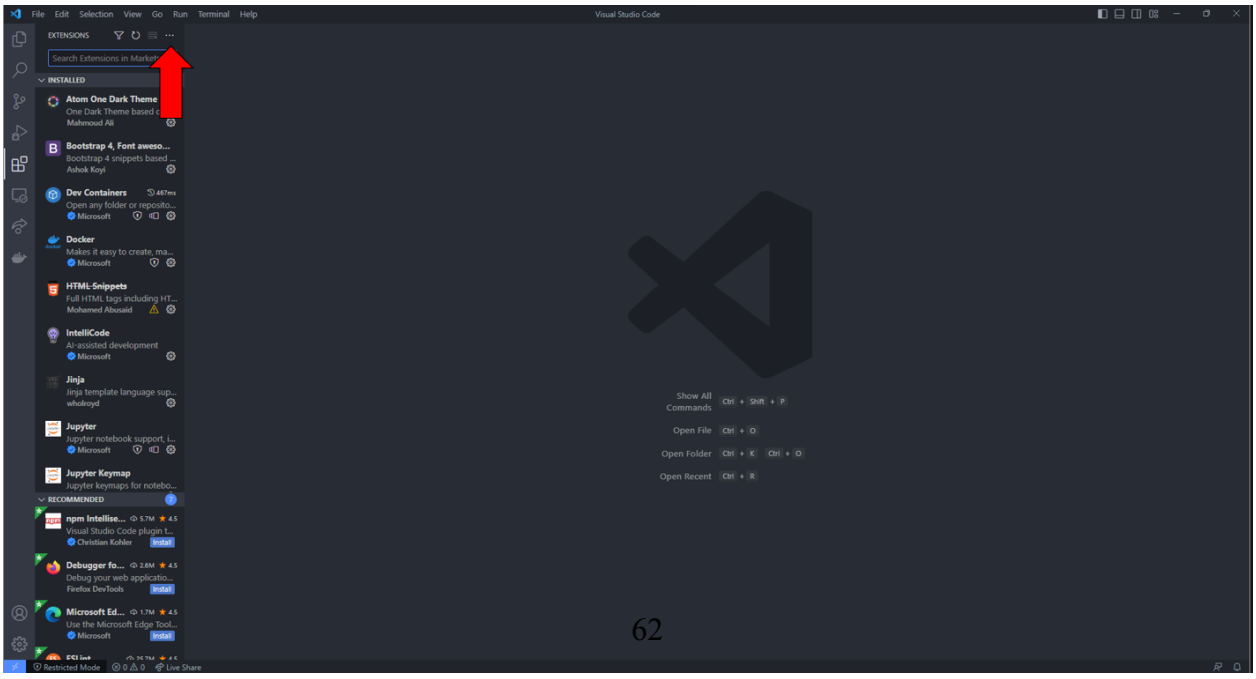
Installation Process

1. Download“typhon-0.1.0.vsix” from http://bit.ly/400kESL

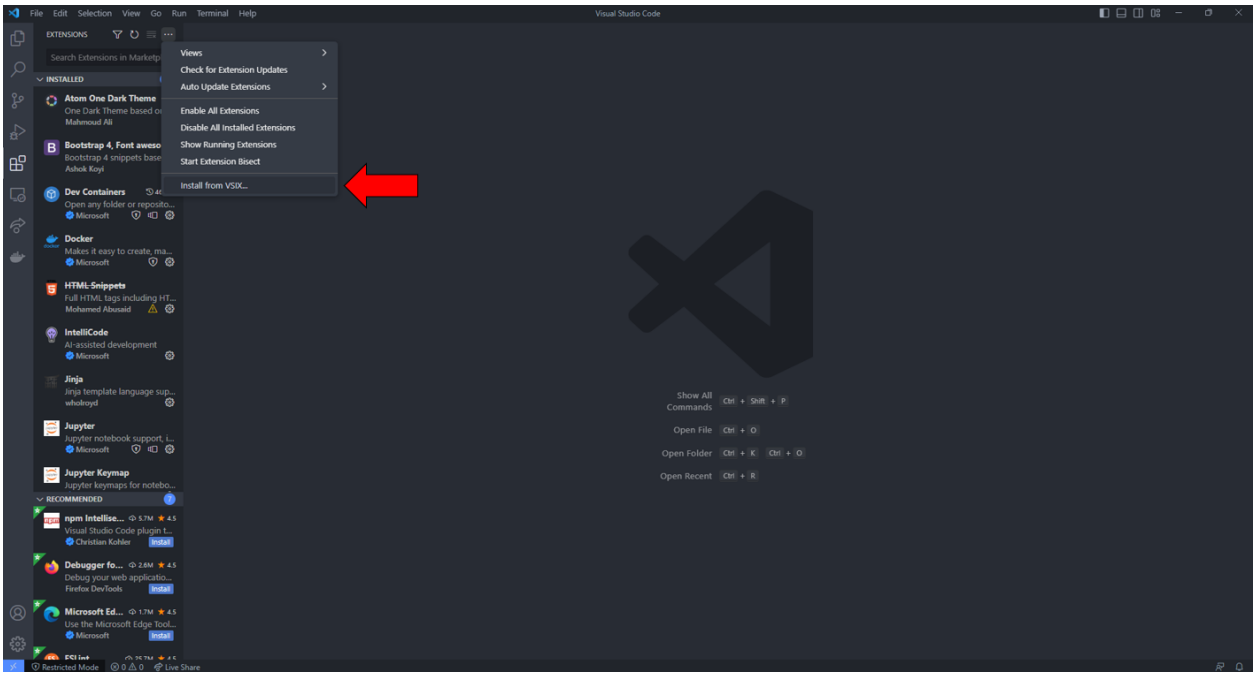
2. Select the extension section in Visual Studio Code



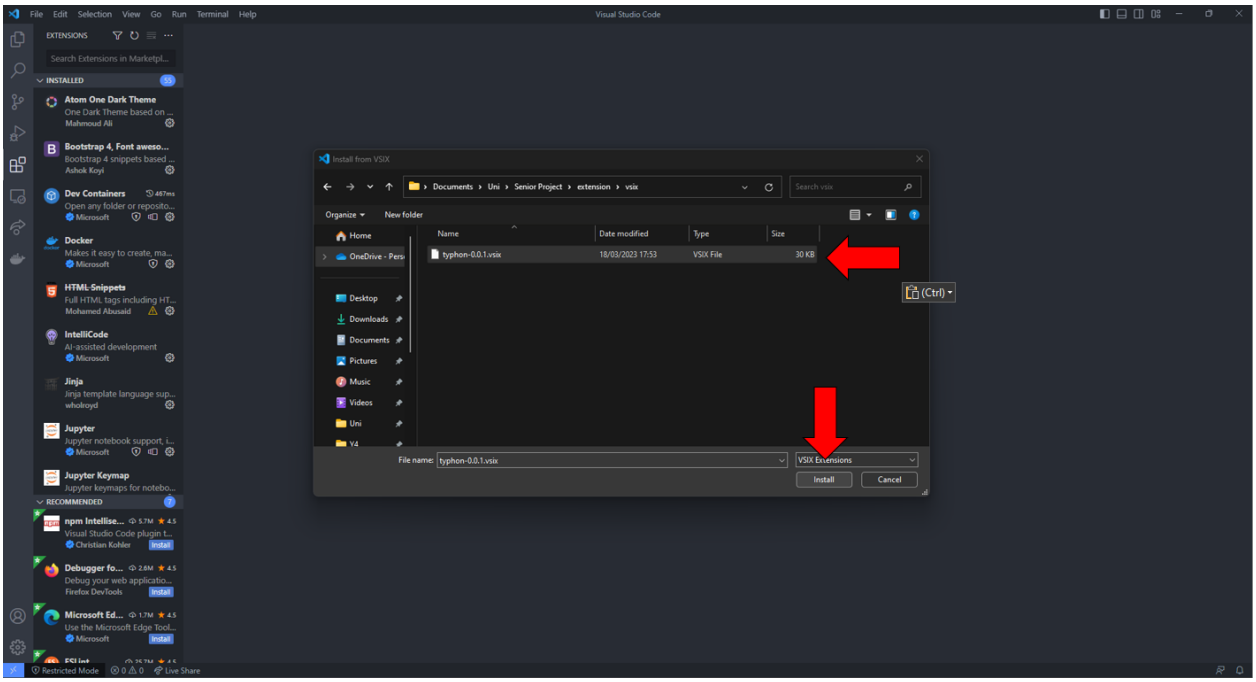
3. Click on the icon ···



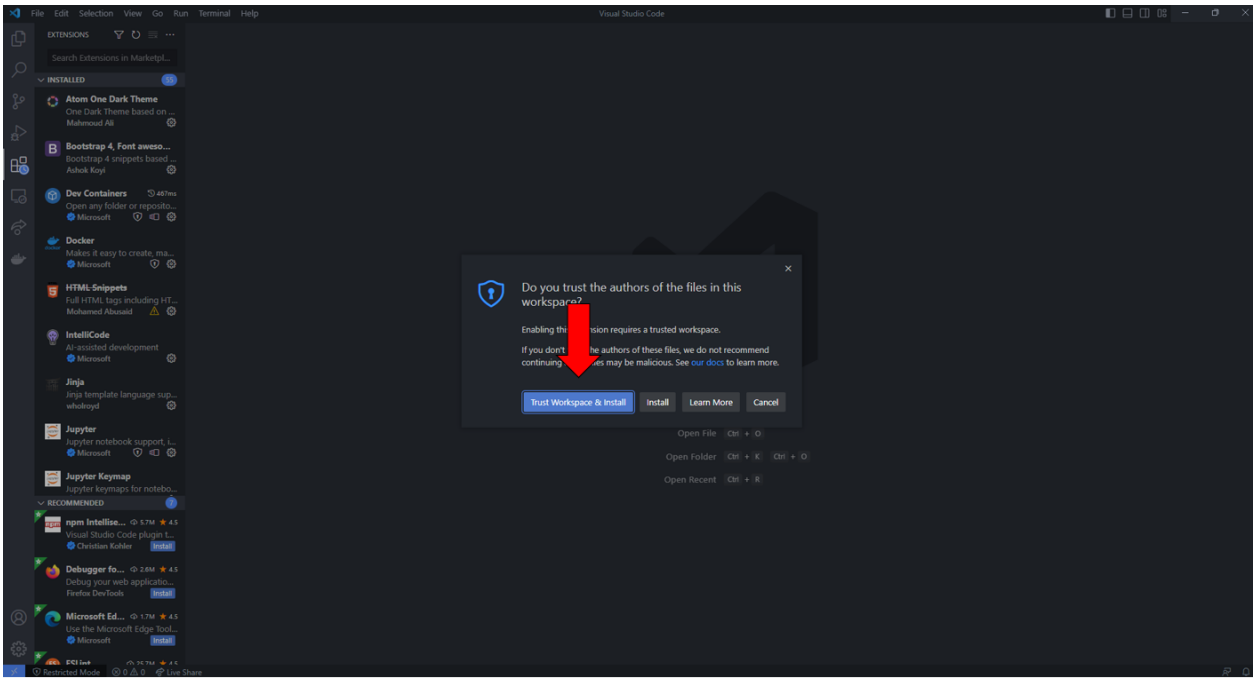
4. Select “Install from VSIX...”



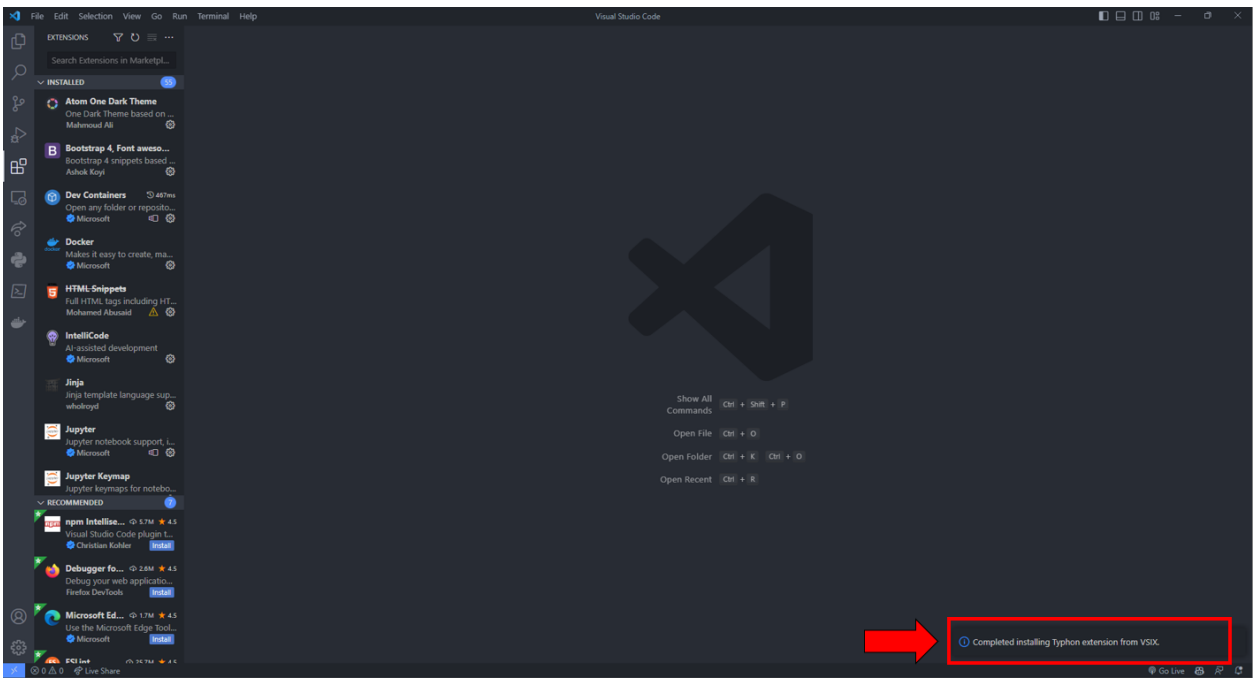
5. Choose “typhon-0.1.0.vsix” and click Install.



6. Click “Trust Workspace & Install”.



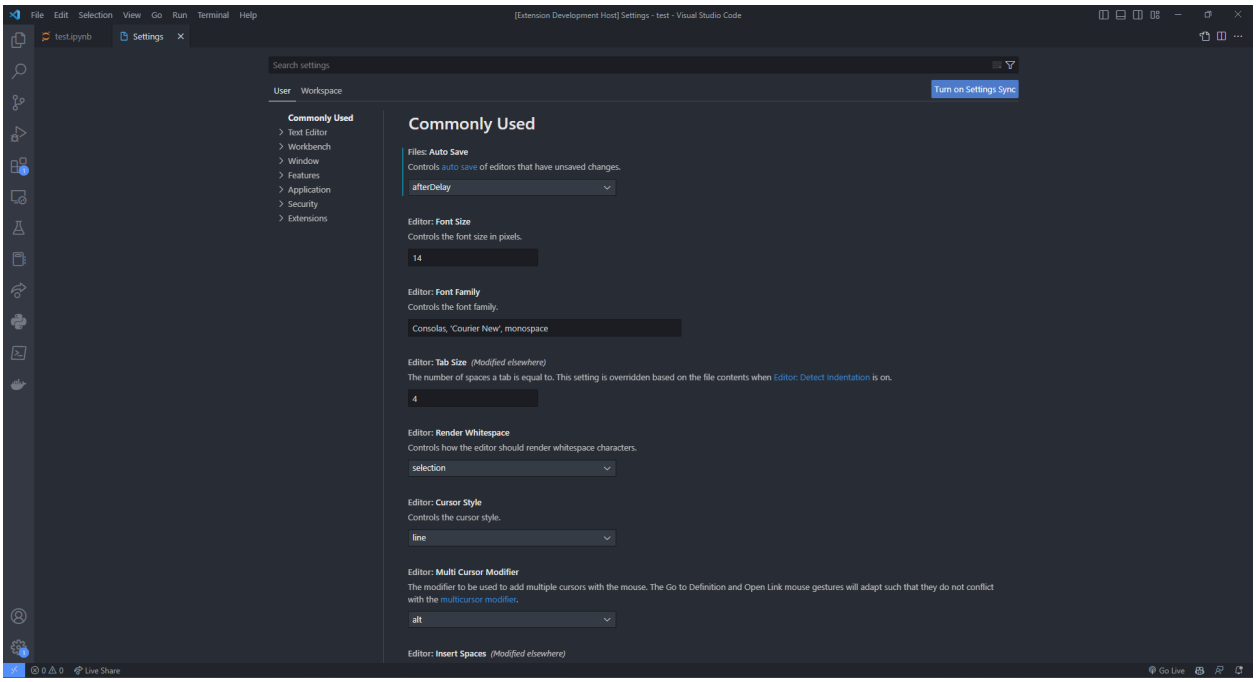
7. Installation is completed.



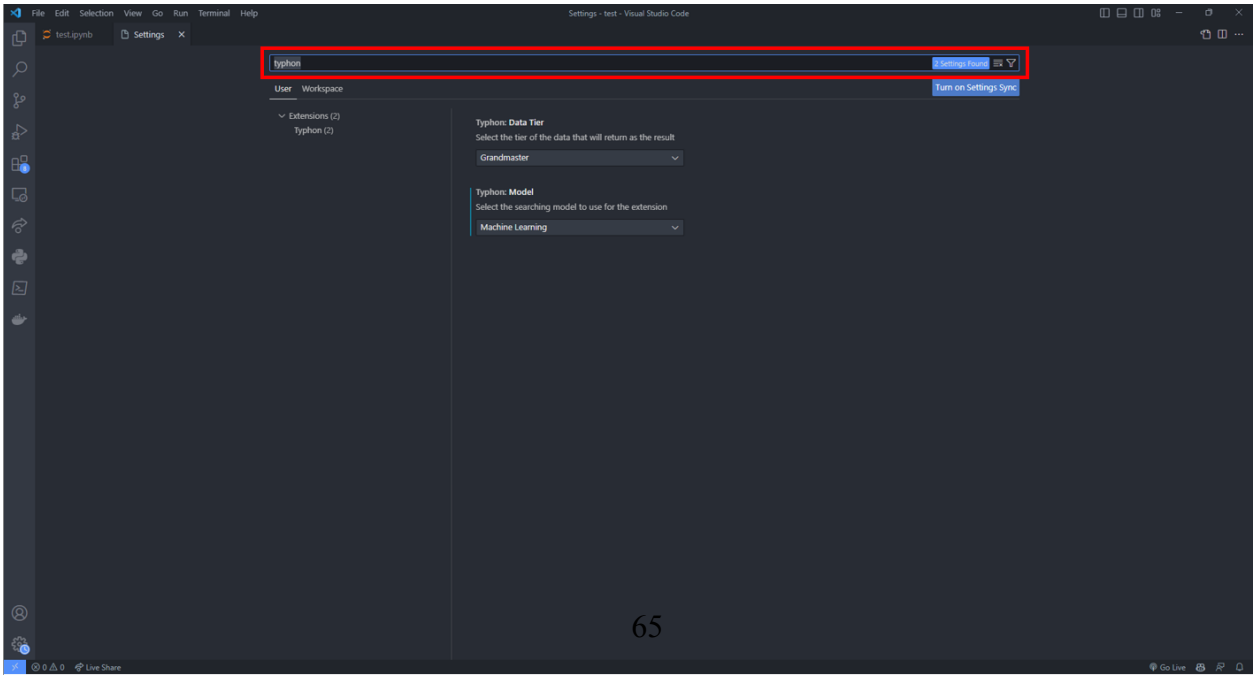
Manual

Setting option for Typhon

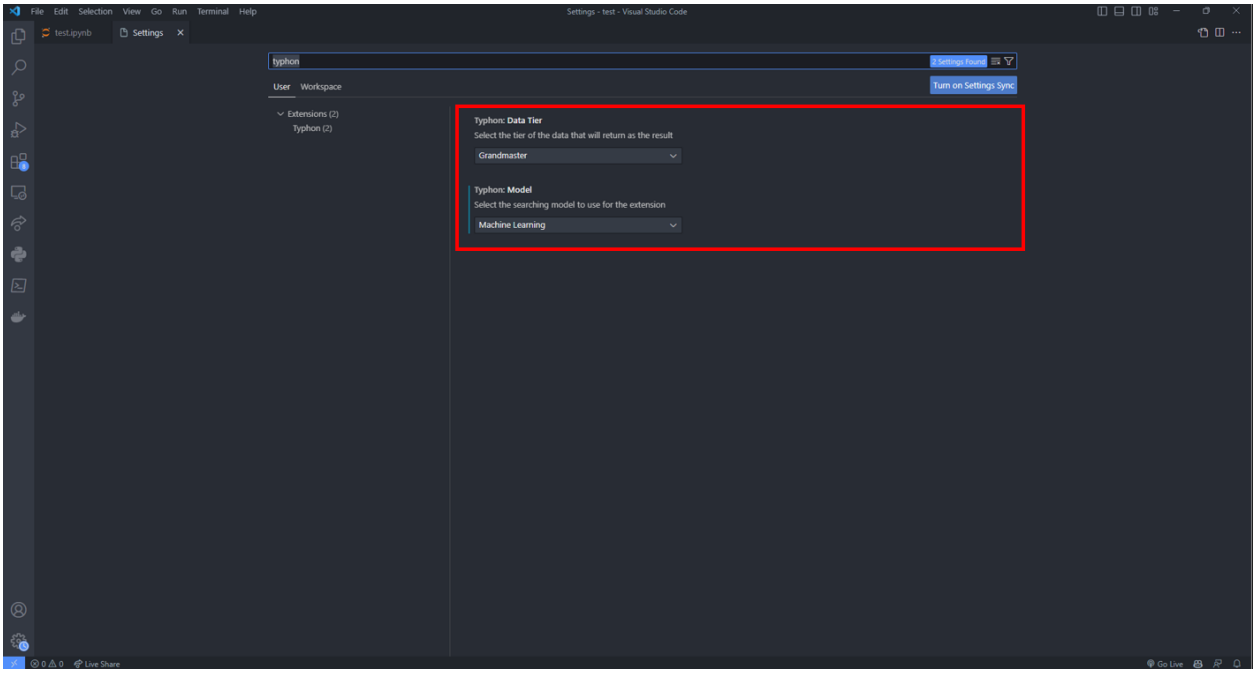
1. Press the “Ctrl” and “,” at the same time to open the setting window.



2. Type in “Typhon” in the search bar above.

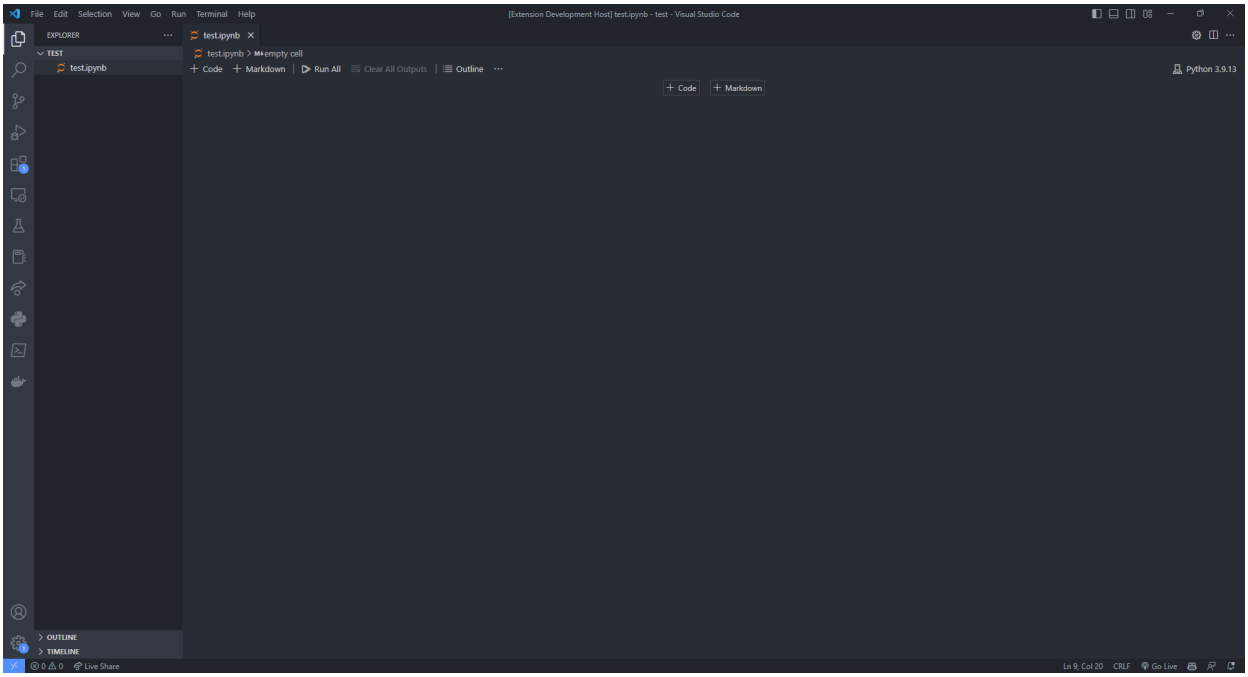


3. In this window, the user can set each of the options for Typhon.

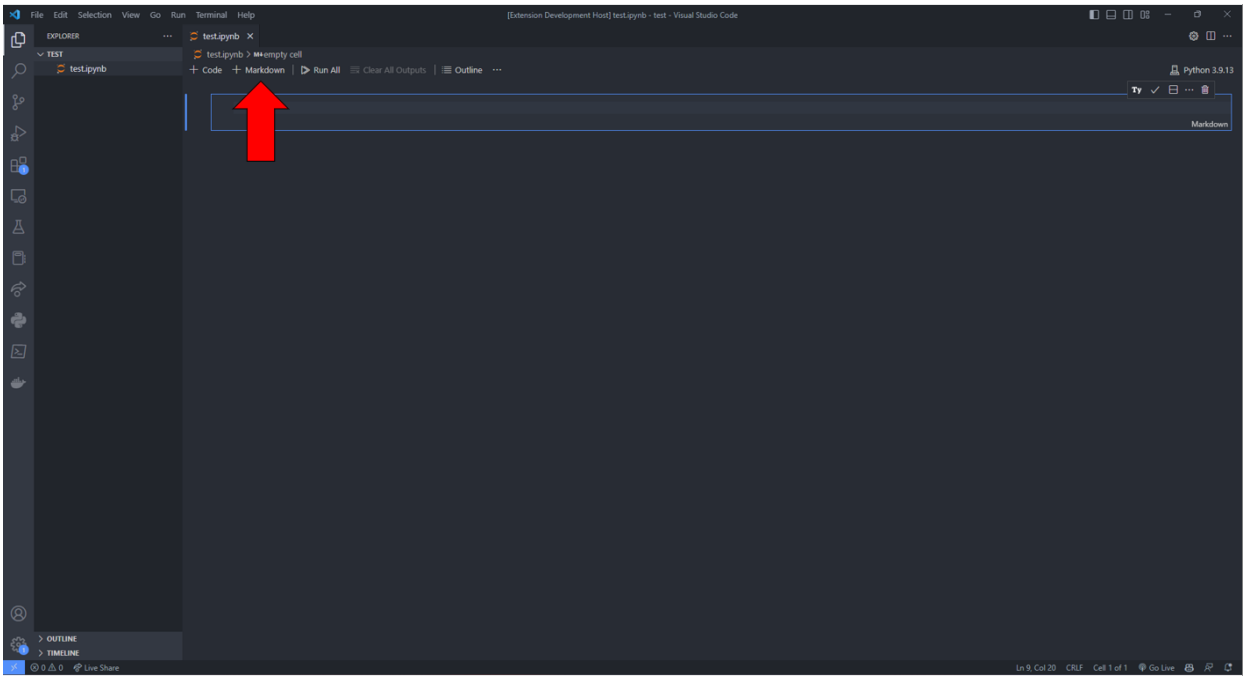


Code Recommendation Manual (BM25 - default setting)

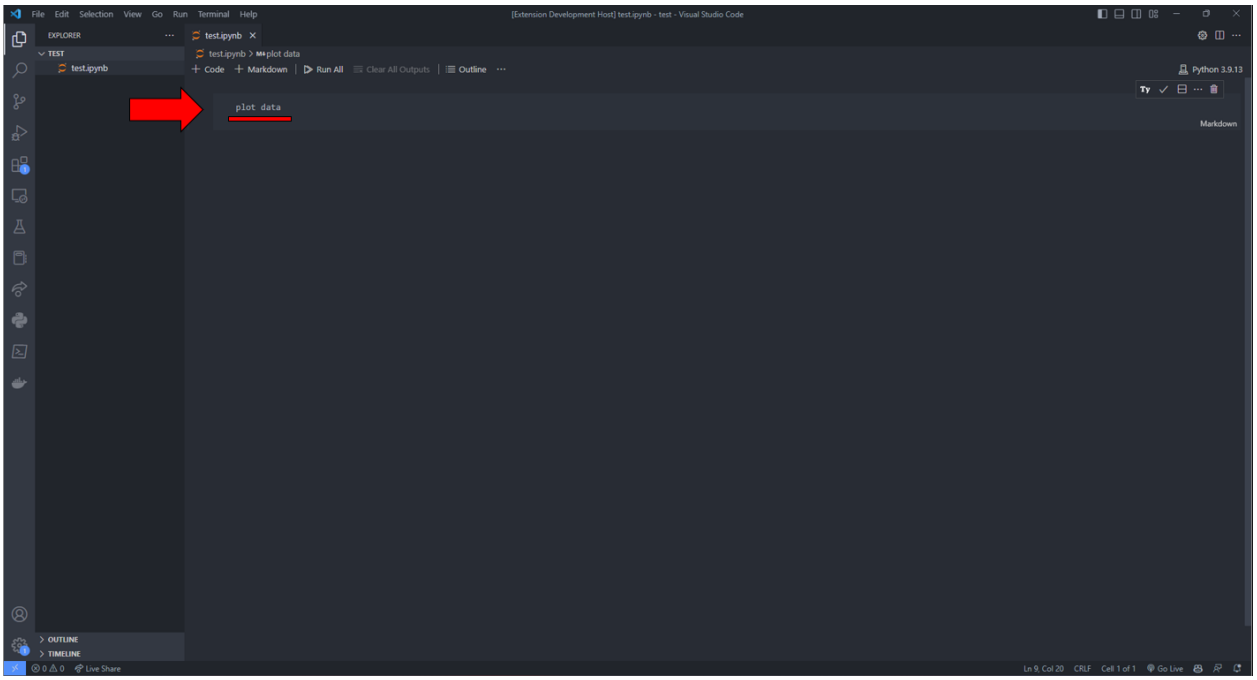
1. Create a Jupyter Notebook file (.ipynb).



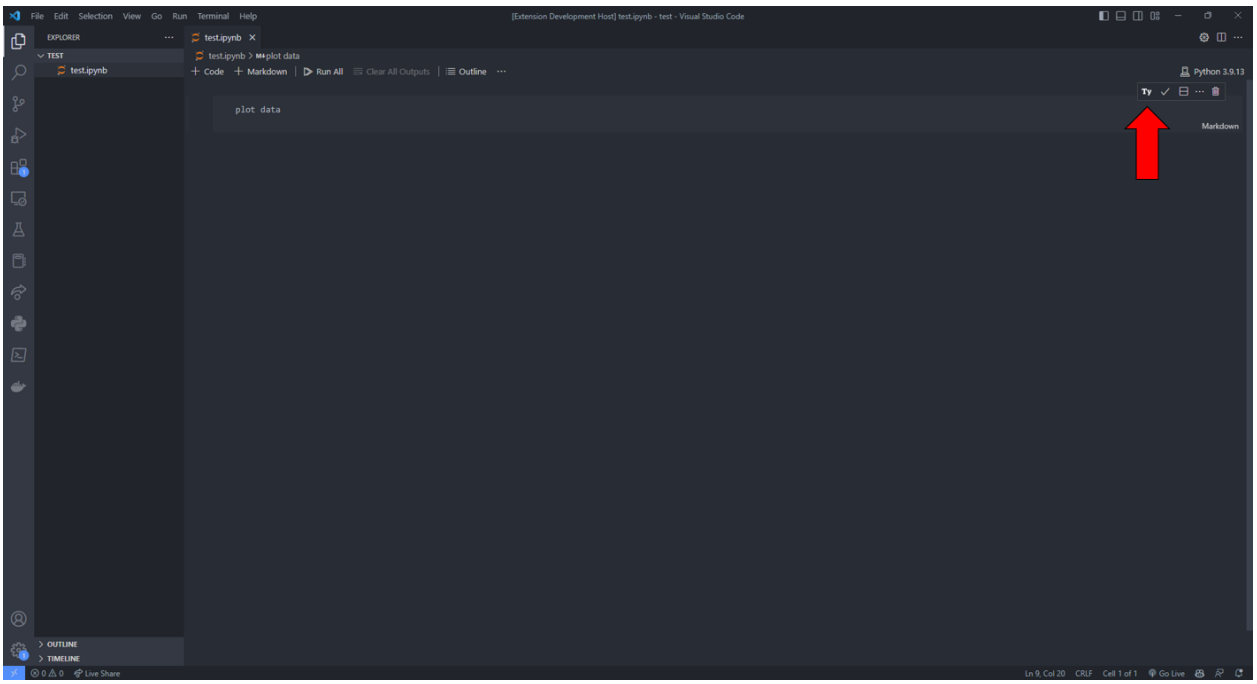
2. Create a Markdown Cell with the “+ Markdown” icon.



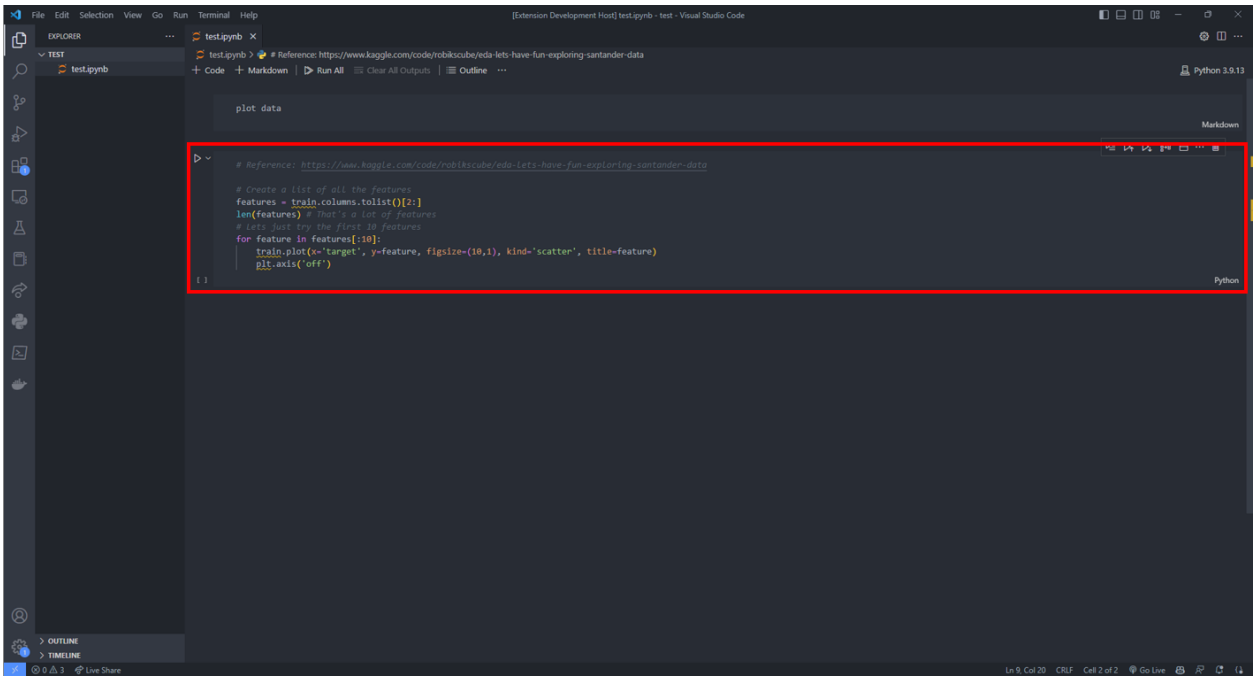
3. Input the Markdown.



4. Click on the Typhon to receive Code Cell recommendations.

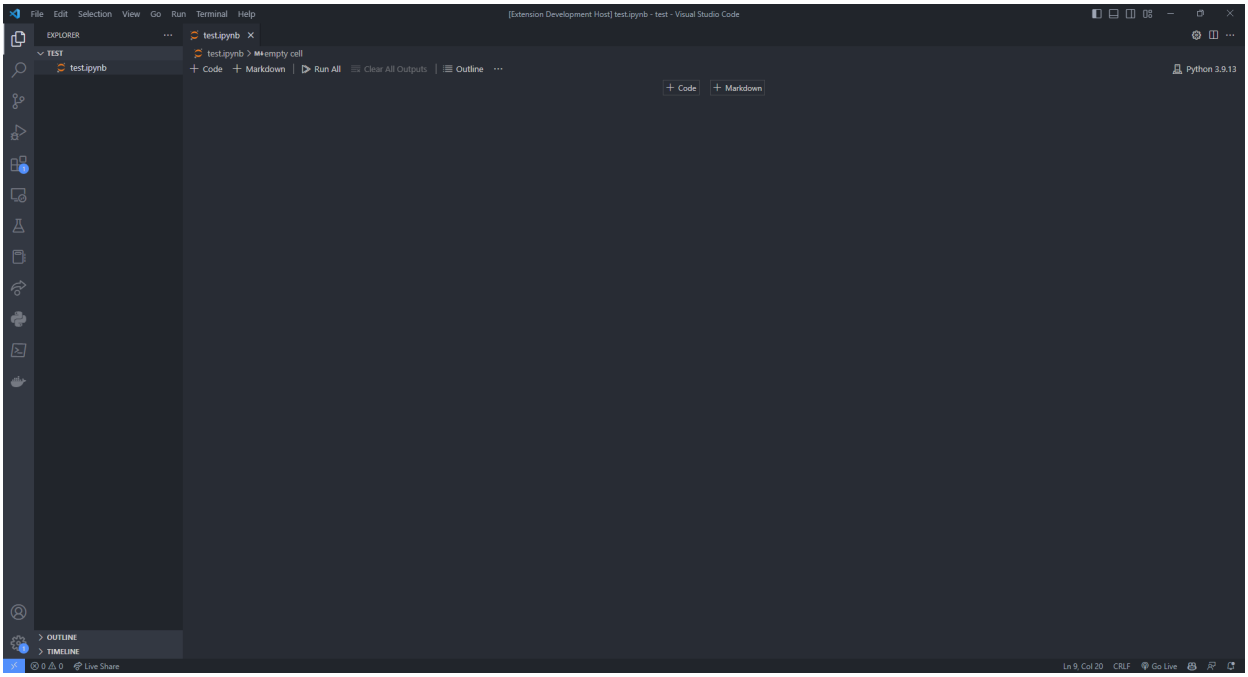


5. Recommended Code Cell will be below the selected Markdown Cell.

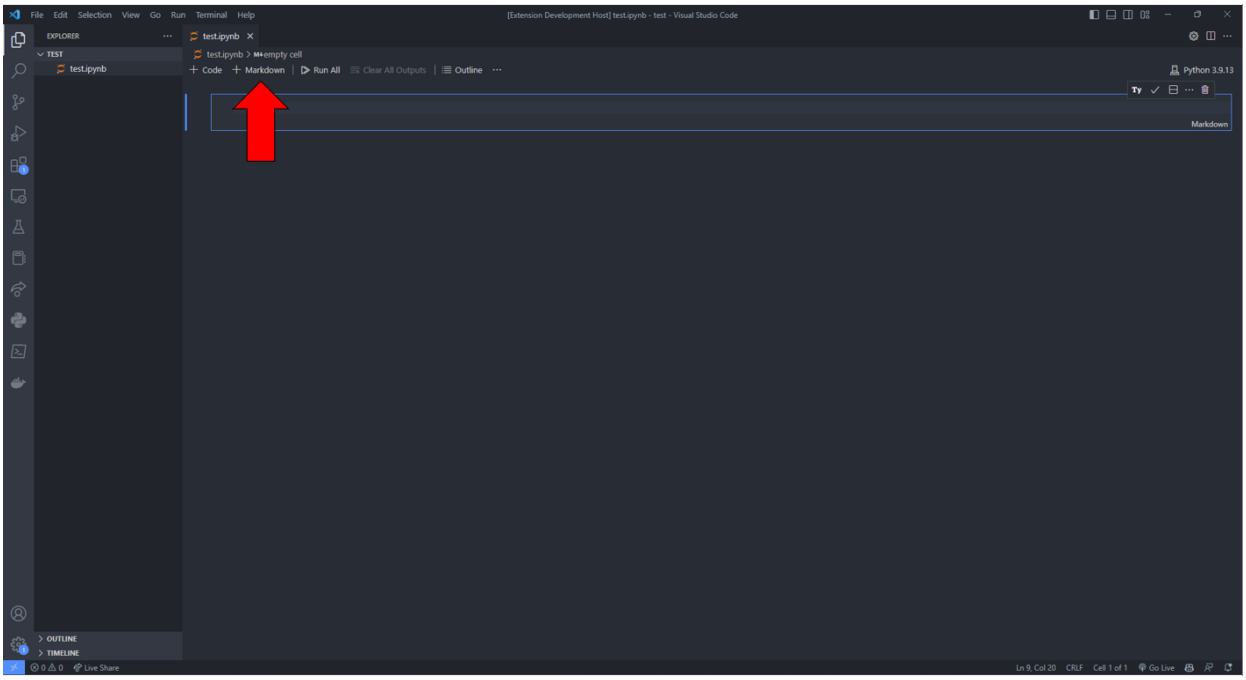


Code Recommendation Manual (BM25 with Stemming and Lemmatization)

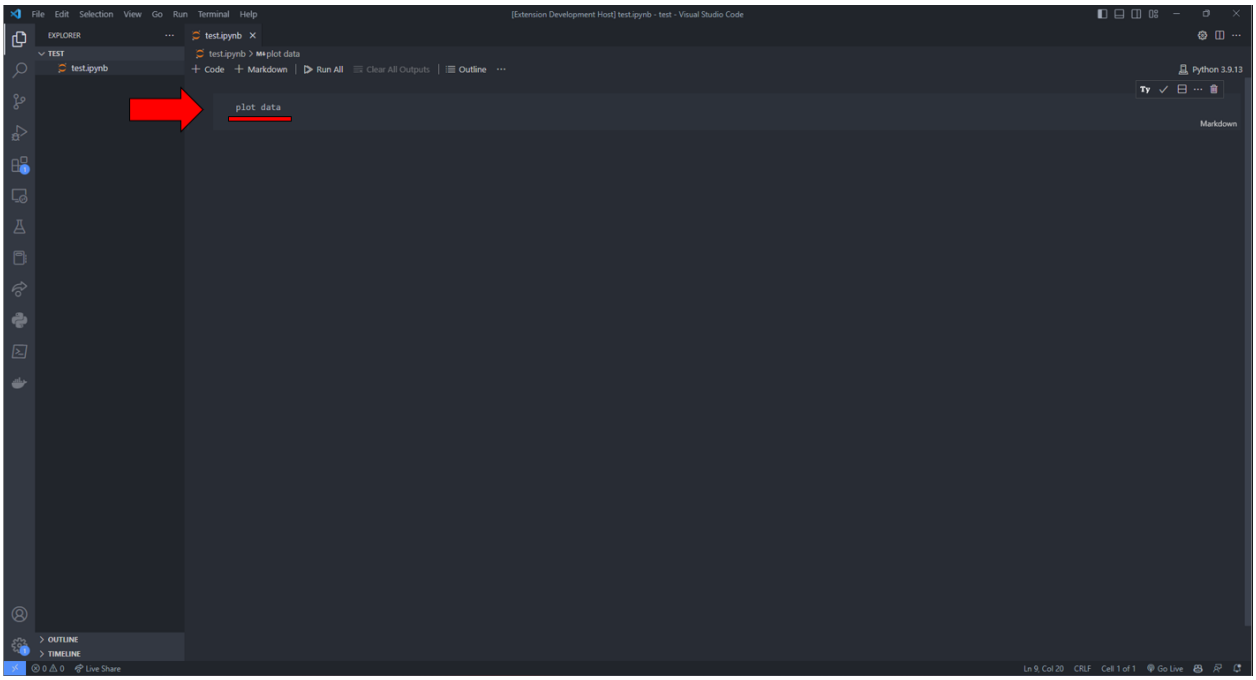
1. Create a Jupyter Notebook file (.ipynb).



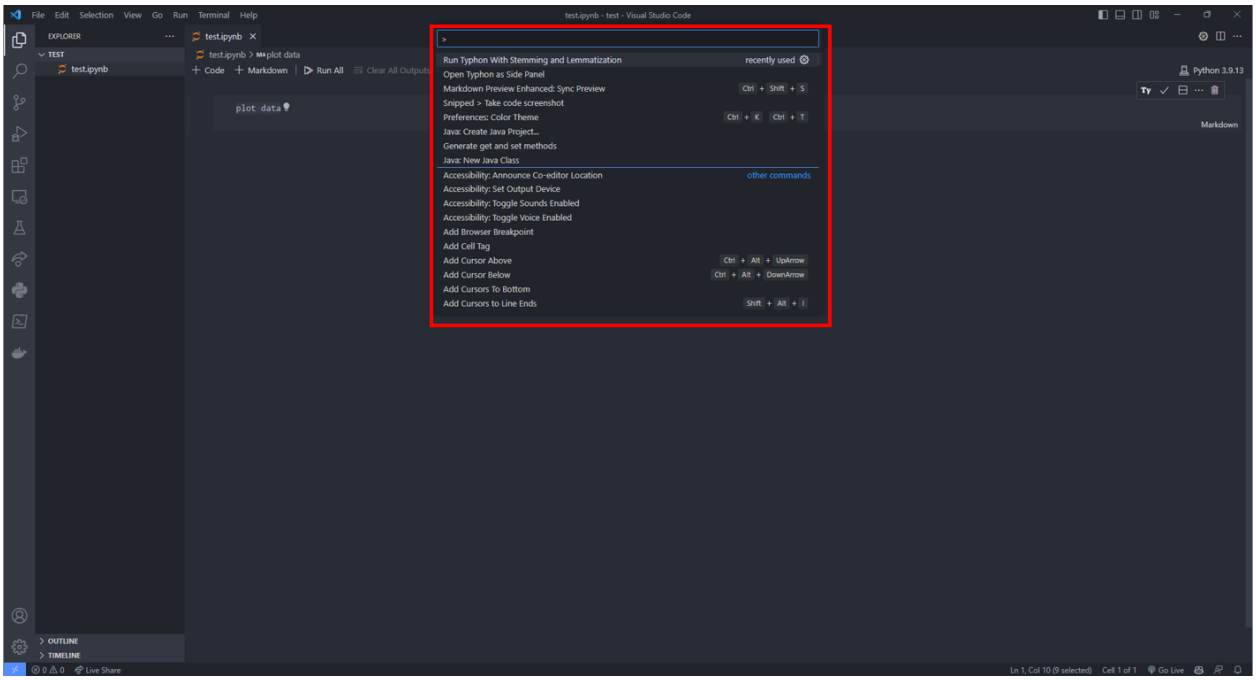
2. Create a Markdown Cell with the “+ Markdown” icon.



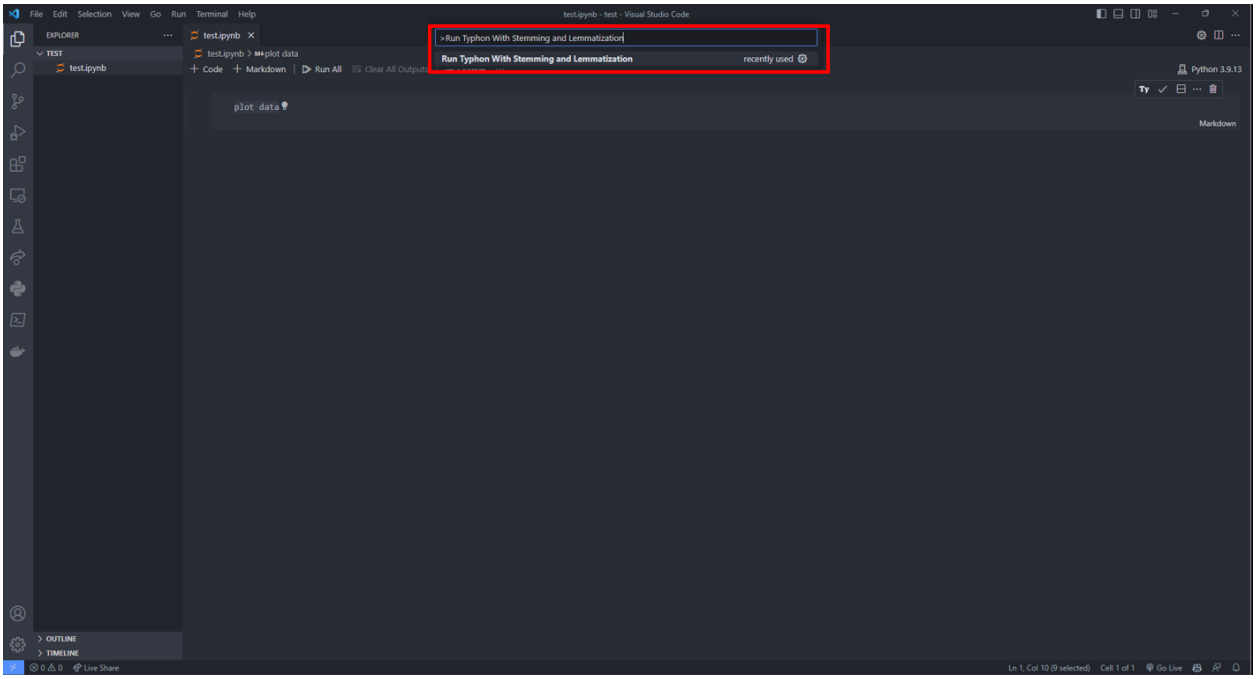
3. Input the Markdown.



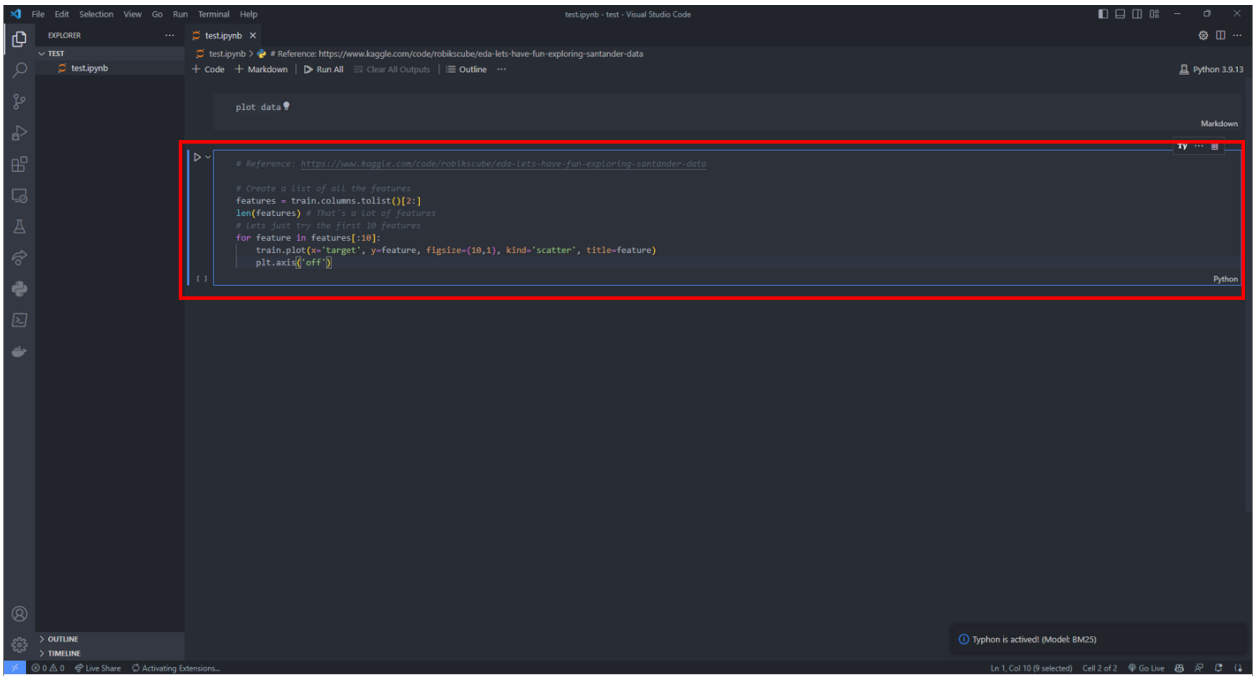
4. Press “Ctrl”, “Shift”, and ”P” at the same time to open the Command Palette.



5. Type “Run Typhon With Stemming and Lemmatization” and press “Enter"

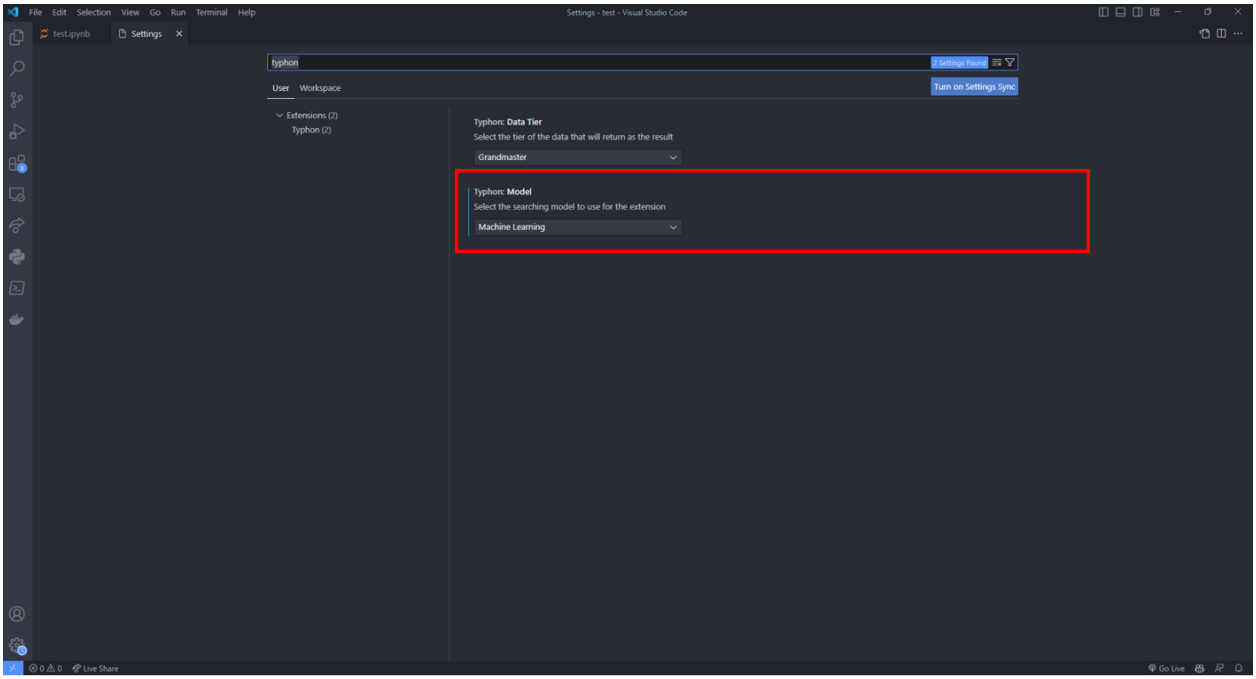


6. Recommended Code Cell will be below the selected Markdown Cell.

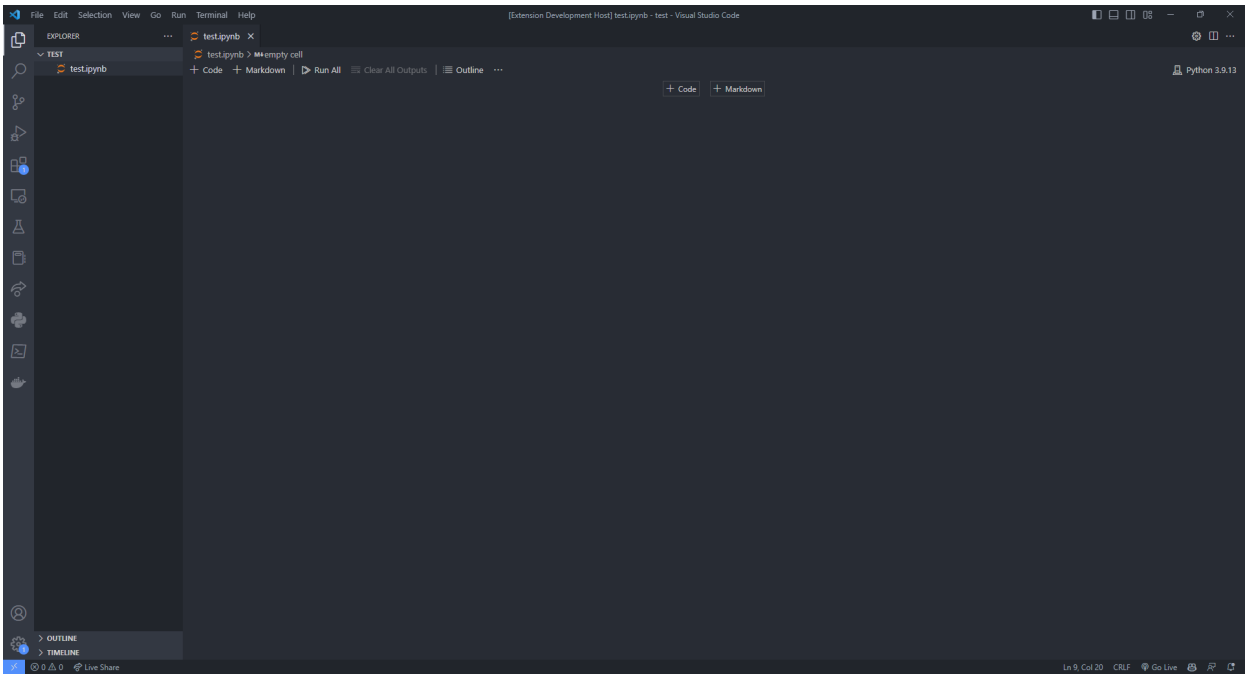


Code Recommendation Manual (Machine Learning)

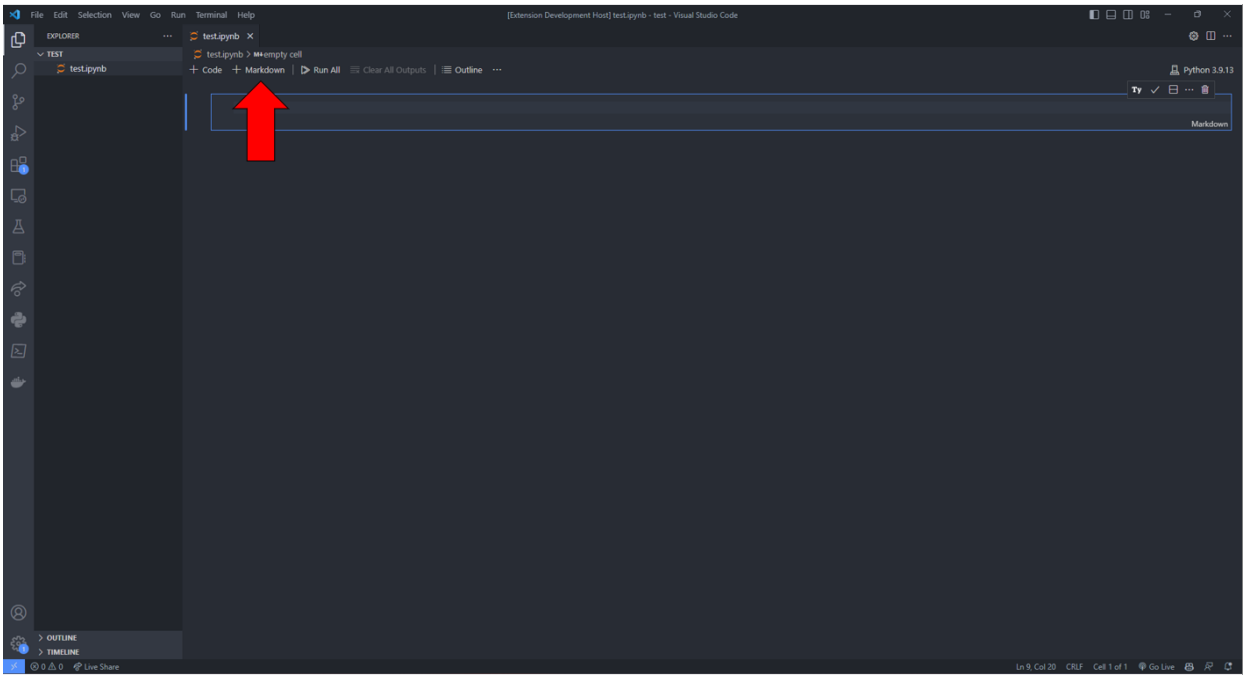
1. Set the option of the searching model to “Machine Learning”



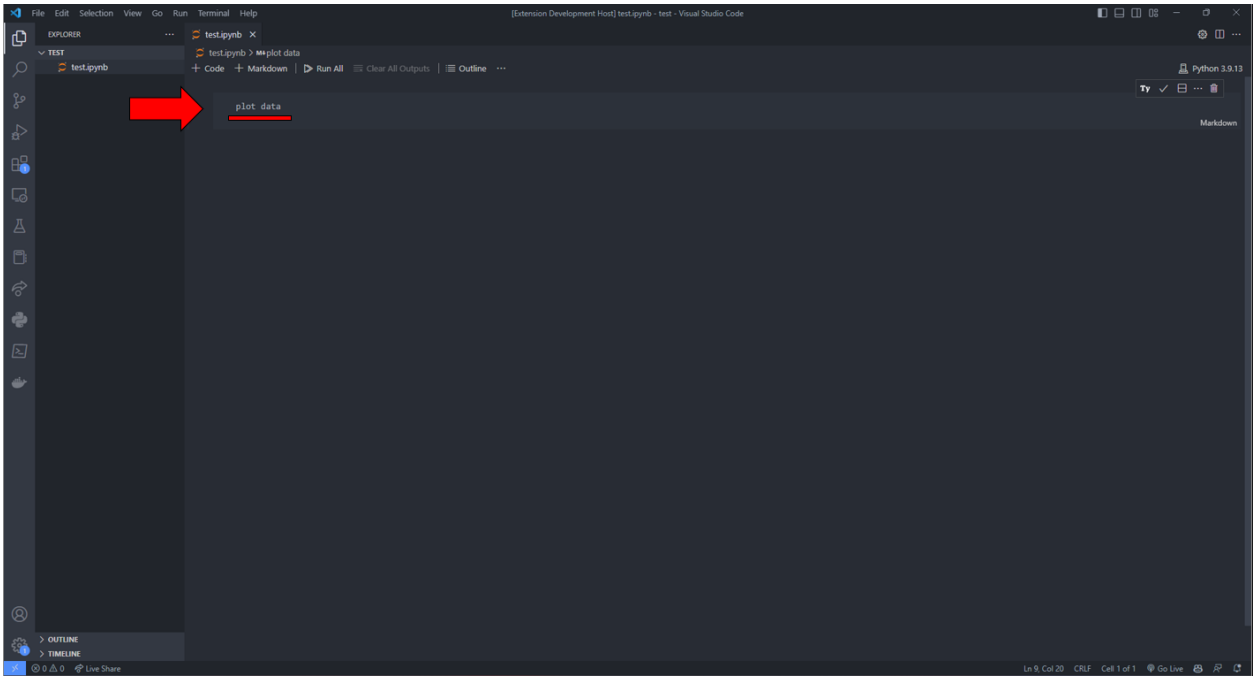
2. Create a Jupyter Notebook file (.ipynb).



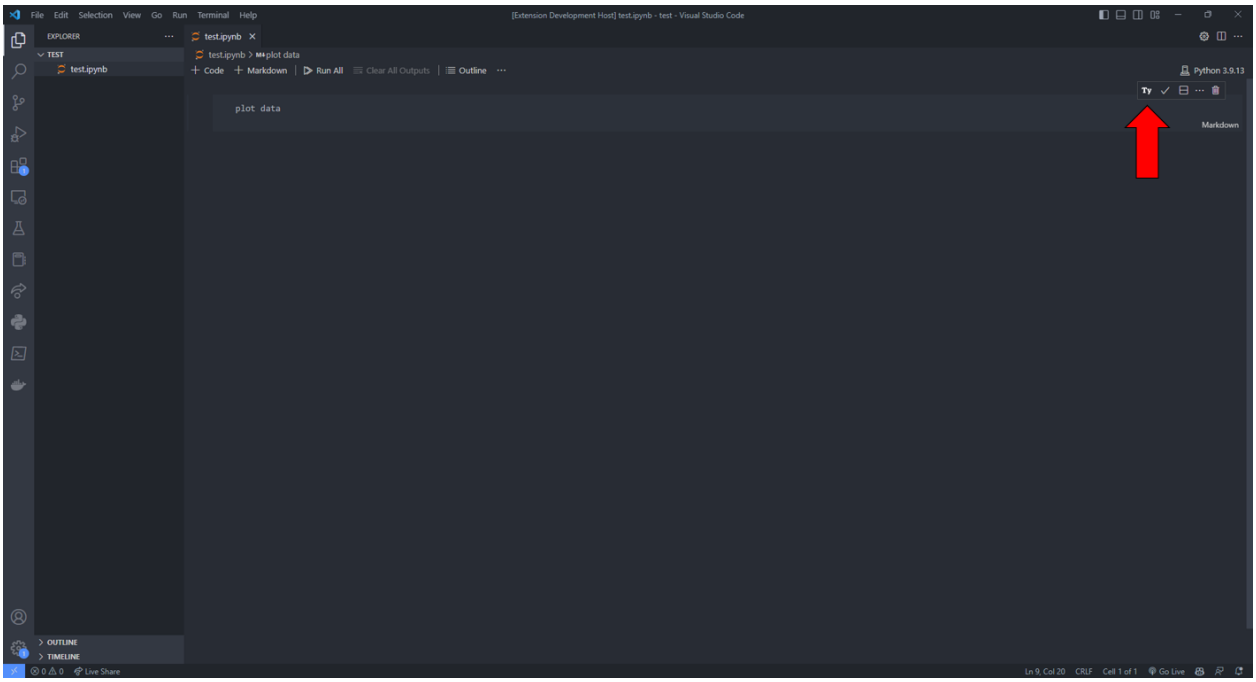
3. Create a Markdown Cell with the “+ Markdown” icon.



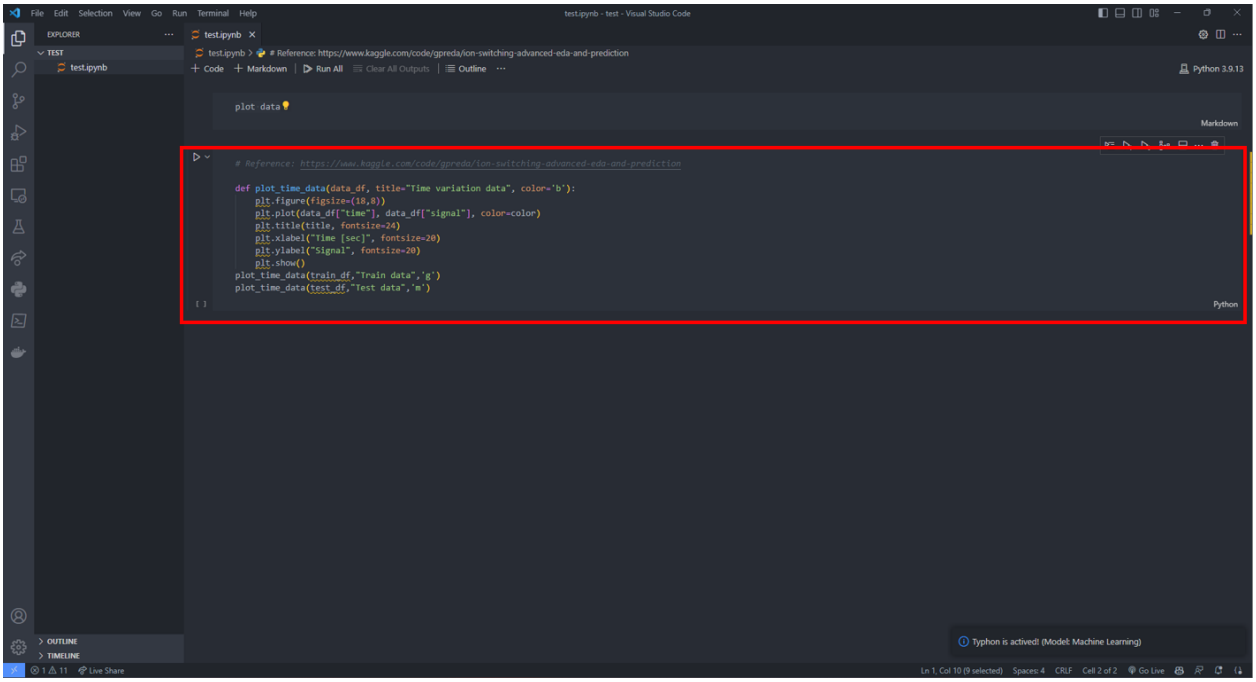
4. Input the Markdown.



5. Click on the Typhon to receive Code Cell recommendations.

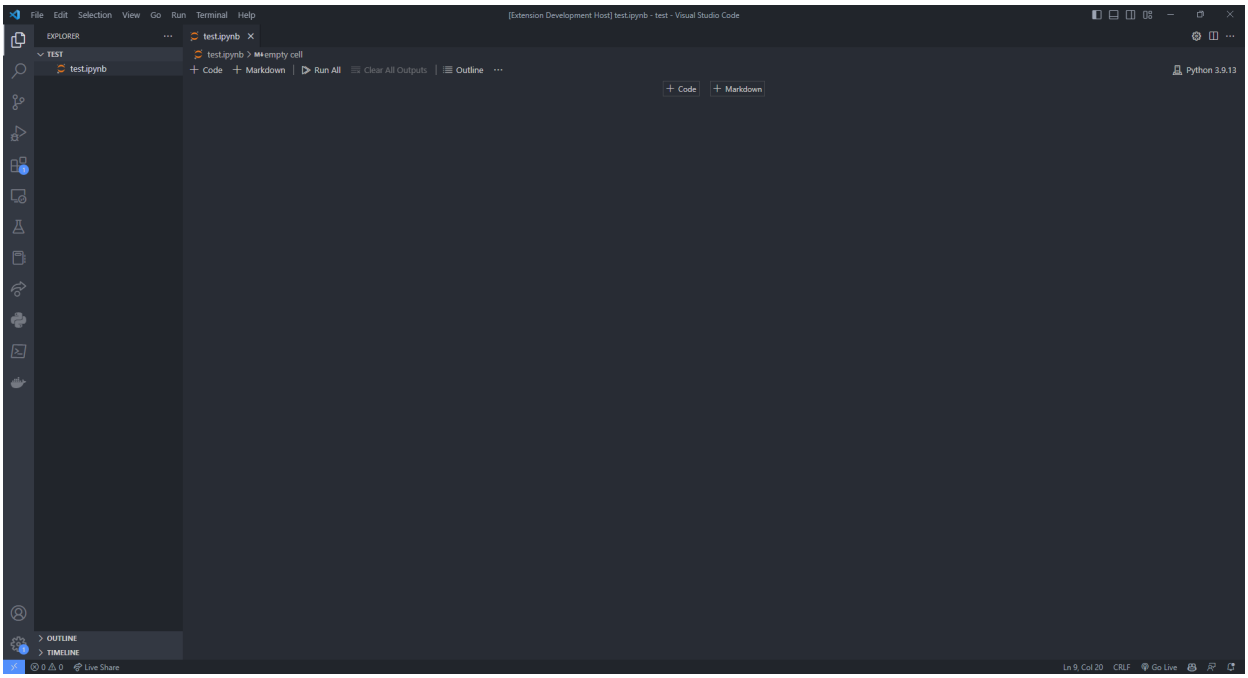


6. Recommended Code Cell will be below the selected Markdown Cell.

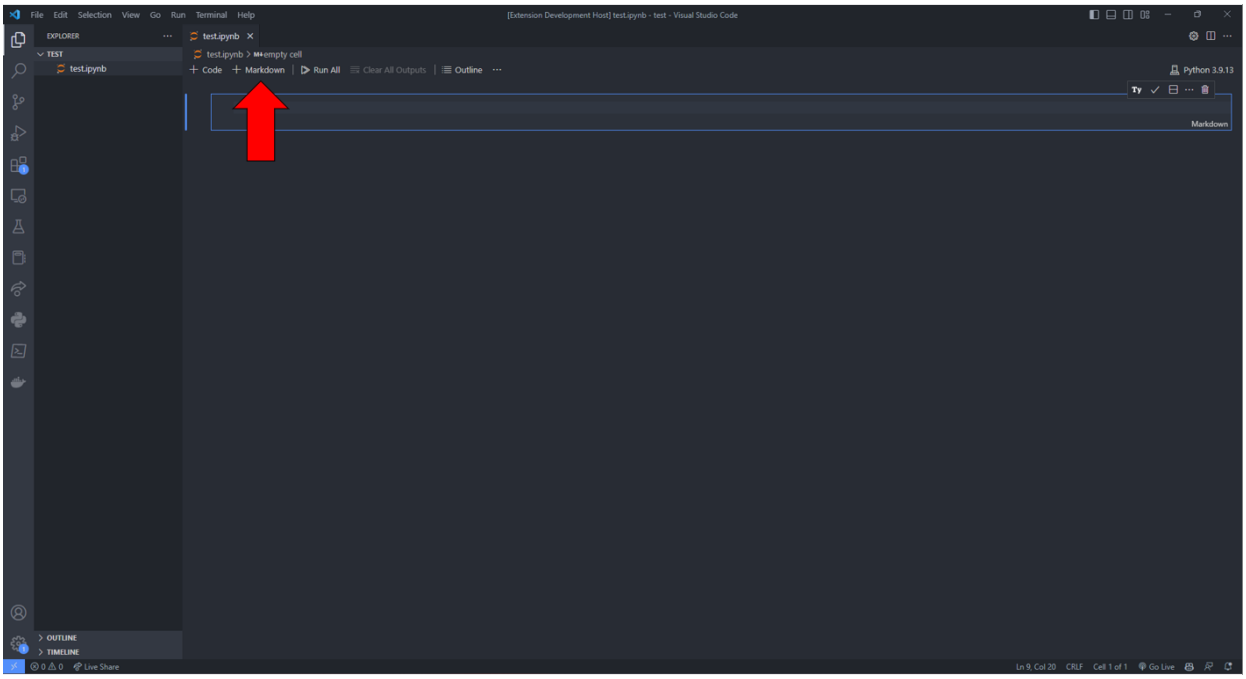


Open Typhon in a side panel

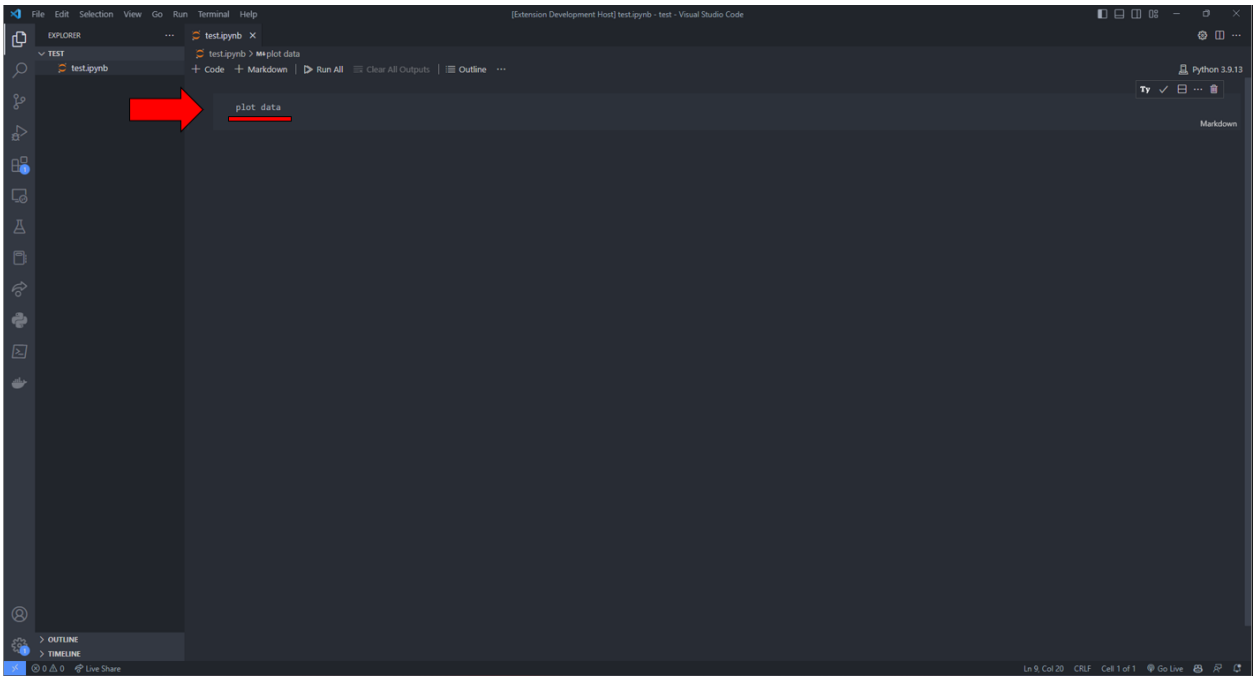
1. Create a Jupyter Notebook file (.ipynb).



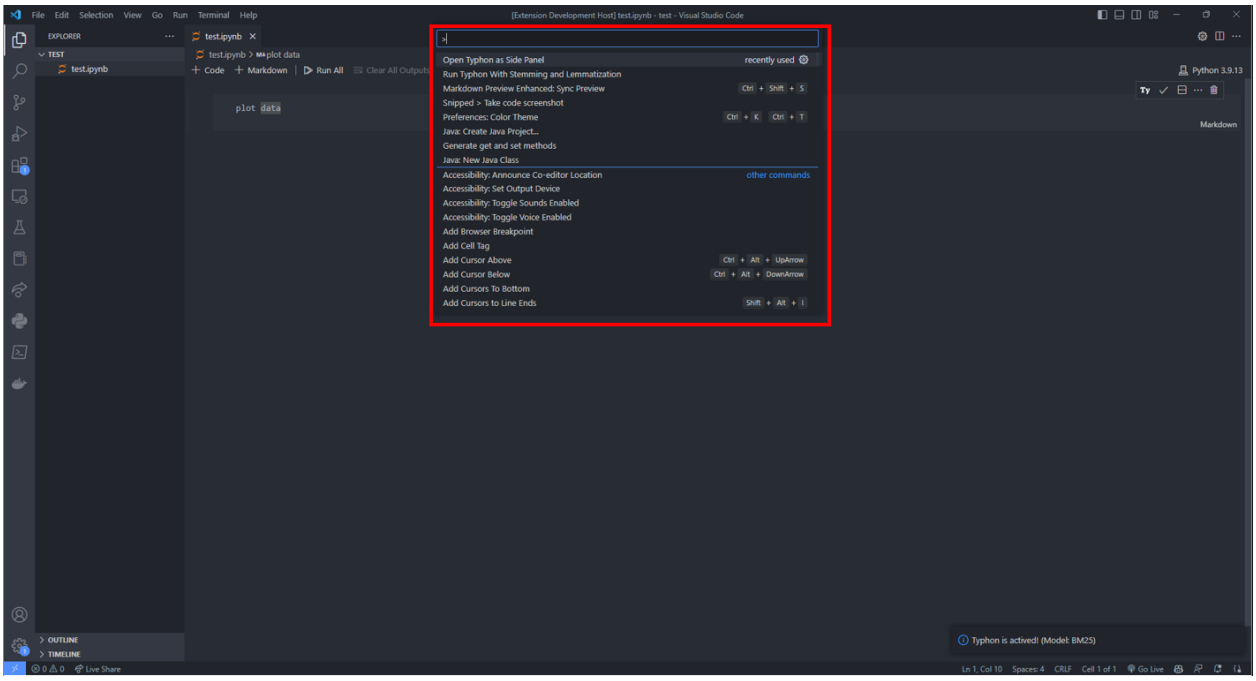
2. Create a Markdown Cell with the “+ Markdown” icon.



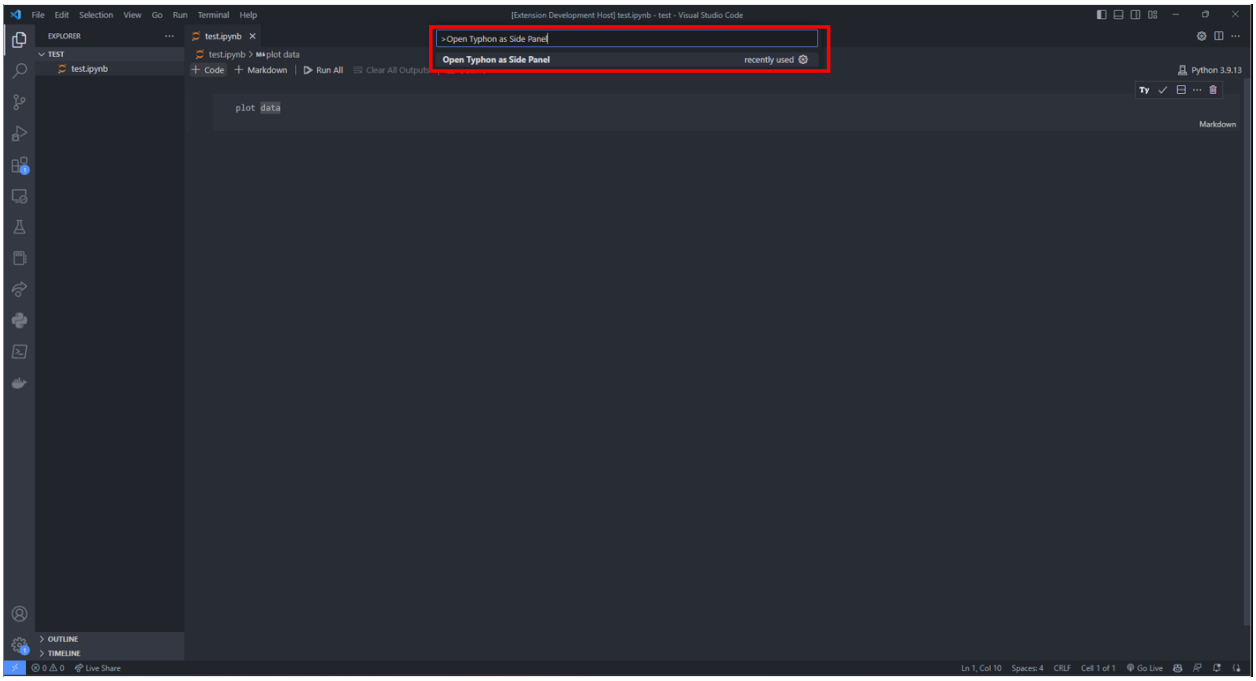
3. Input the Markdown.



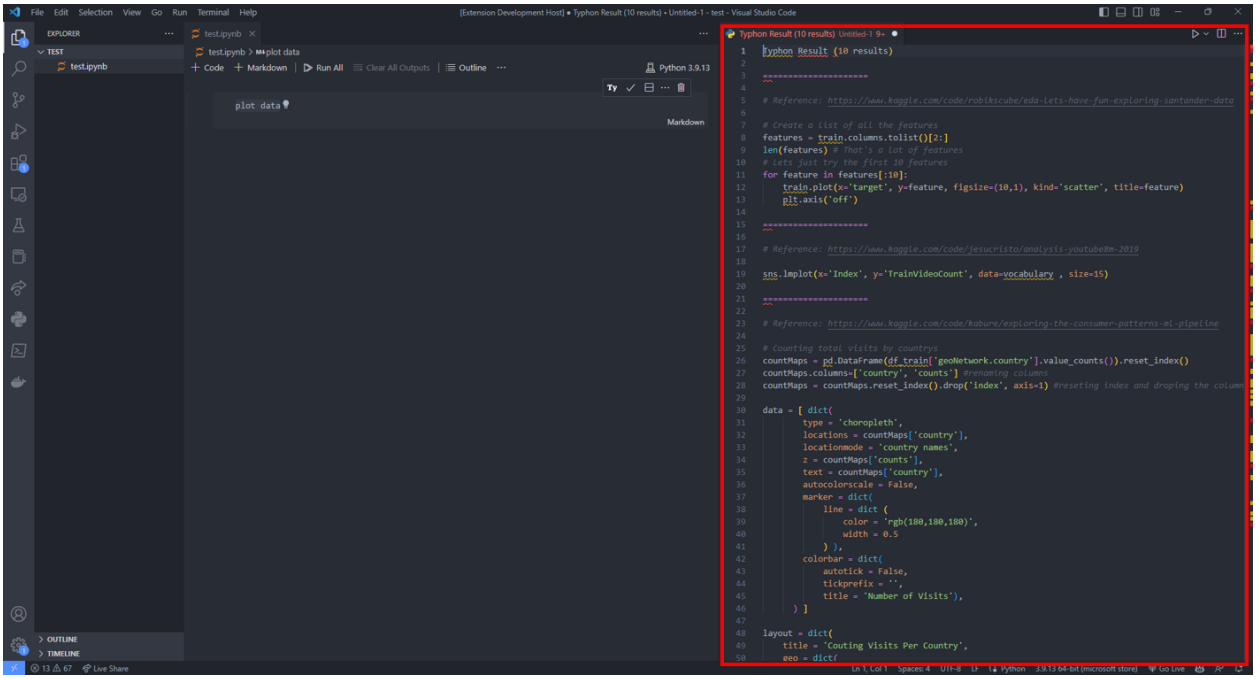
4. Press “Ctrl”, “Shift”, and ”P” at the same time to open the Command Palette.



5. Type “Open Typhon as Side Panel” and press” Enter”



6. Side Panel will be open on the right side of the window.



REFERENCES

[1] Zagalsky A., Barzilay O., Yehudai A., "Example overflow: Using social media for code recommendation"; 2012. p. 38–42.

[2] Luan S., Yang D., Barnaby C., Sen K., Chandra S., "Aroma: Code recommendation via structural code search", Proceedings of the ACM on Programming Languages. 10 2019;3.

[3] Silavong F., Moran S., Georgiadis A., Saphal R., Otter R., "Senatus - A Fast and Accurate Code-to-Code Recommendation Engine", Institute of Electrical and Electronics Engineers Inc.; 2022. p. 511–523.

[4] Holmes R., Walker RJ., Murphy GC., "Strathcona Example Recommendation Tool", ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering. 2005;p. 237–240.

[5] AI Terms. "What is TabNine?";, Online; accessed 4 November 2022, https://aiterms.net/tabnine/.

[6] Kluyver T., Ragan-Kelley B., Pérez F., Granger B., Bussonnier M., Frederic J., et al., "Jupyter Notebooks—a publishing format for reproducible computational workflows", IOS Press BV; 2016. p. 87–90.

[7] Ritta N., Settewong T., Kula RG., Ragkhitwetsagul C., Sunetnanta T., Matsumoto K., "Reusing My Own Code: Preliminary Results for Competitive Coding in Jupyter Notebooks", In: Proceedings of the 29th Asia-Pacific Software Engineering Conference (APSEC '22); 2022. .

[8] Quaranta L., Calefato F., Lanubile F., "KGTorrent: A Dataset of Python Jupyter Notebooks from Kaggle"; 3 2021. [Online]. Available: http://arxiv.org/abs/2103.10558 http://dx.doi.org/10.1109/MSR52588.2021.00072.

[9] Husain H., Wu HH., Gazit T., Allamanis M., Brockschmidt M., "CodeSearchNet Challenge: Evaluating the State of Semantic Code Search", arXiv. 9 2019;[Online]. Available: http://arxiv.org/abs/1909.09436.

[10] Ragkhitwetsagul C., Krinke J., Clark D., "A comparison of code similarity analysers", Empirical Software Engineering. aug 2018;23(4):2464–2519.

[11] Guo D., Lu S., Duan N., Wang Y., Zhou M., Yin J., "UniXcoder: Unified Cross-Modal Pre-training for Code Representation", arXiv. 3 2022;[Online]. Available: http://arxiv.org/abs/2203.03850.

[12] Nat Friedman. "Introducing GitHub Copilot: your AI pair programmer";, Online; accessed 3 April 2023, https://github.blog/2021-06-29-introducing-github-copilot-ai-pair-programmer/.

[13] Tilgner A.. "A Deep Dive Into GitHub Copilot";, Online; accessed 3 April 2023, https://betterprogramming.pub/ai-review-github-copilot-d43afde51a5a.

[14] Baruffati A.. "Chat GPT Statistics 2023: Trends And The Future Perspectives";, Online; accessed 8 April 2023, https://blog.gitnux.com/chat-gpt-statistics.

[15] Salva RJ.. "Preview: referencing public code in GitHub Copilot";, Online; accessed 4 April 2023, https://github.blog/2022-11-01-preview-referencing-public-code-in-github-copilot/.

BIOGRAPHIES

NAME	Mr. Veerakit Prasertpol
DATE OF BIRTH	10 June 2000
PLACE OF BIRTH	Bangkok, Thailand
INSTITUTIONS ATTENDED	A School, 2017: High School Diploma Mahidol University, 2023: Bachelor of Science (ICT)

NAME	Mr. Natanon Ritta
DATE OF BIRTH	13 September 2000
PLACE OF BIRTH	Khon Kaen, Thailand
INSTITUTIONS ATTENDED	A School, 2017: High School Diploma Mahidol University, 2023: Bachelor of Science (ICT)

NAME	Mr. Paphon Sae-wong
DATE OF BIRTH	29 August 2000
PLACE OF BIRTH	Bangkok, Thailand
INSTITUTIONS ATTENDED	A School, 2017: High School Diploma Mahidol University, 2023: Bachelor of Science (ICT)