NeSy is alive and well: A LLM-driven symbolic approach for better code comment data generation and classification

Hanna Abi Akl[1,2*]

[1*]Data ScienceTech Institute (DSTI), 4 Rue de la Collégiale, 75005, Paris, France.
[2]Université Côte d'Azur, Inria, CNRS, I3S.

Corresponding author(s). E-mail(s): hanna.abi-akl@dsti.institute;

Abstract

We present a neuro-symbolic (NeSy) workflow combining a symbolic-based learning technique with a large language model (LLM) agent to generate synthetic data for code comment classification in the C programming language. We also show how generating controlled synthetic data using this workflow fixes some of the notable weaknesses of LLM-based generation and increases the performance of classical machine learning models on the code comment classification task. Our best model, a Neural Network, achieves a Macro-F1 score of 91.412% with an increase of 1.033% after data augmentation.

Keywords: Neuro-symbolic AI, Natural Language Processing, Machine Learning, Large Language Models, Code Generation, Synthetic Data

Contents

1 Introduction

The era of Large Language Models (LLMs) has introduced agents capable of handling different tasks and performing well on them in domains such as text, image and audio [1]. A popular extension to the use of LLMs is in applying them to other data formats often used by humans in their daily activities. One such data source is code which circulates heavily and makes up a crucial block of technological projects [2].

The public availability of code-hosting repositories like GitHub on the web makes code an accessible data source and a valuable input for LLMs to tackle code-related challenges [2]. These tasks can range from identifying correct code to generating entirely new source code [2]. This has made source code datasets an invaluable part of the pre-training of modern LLM agents [2]. However, a persistent requirement and problem for these models is that they are both data-hungry and resource-hungry [1]. This is tied to the question of scale that dictates that in order to keep performing well on tasks and adapt to new tasks, LLMs have to be fed more data [1]. A consequence of this demand is data scarcity, a pitfall for all LLM agents today. Data scarcity is still an open problem that is becoming a pressing issue in the face of the advancement and improvement of LLM technologies since it directly affects their greatest source of power: data. Research is ongoing to actively tackle and solve the problem of data scarcity [3–5] but, to our knowledge, no wide-scale solution exists as of yet at the time of writing of this work.

The Information Retrieval in Software Engineering (IRSE)[1] at the Forum for Information Retrieval Evaluation (FIRE)[2] 2023 shared task is one challenge that addresses the problem of data scarcity. It sets out to measure the effects of leveraging LLMs to generate new data and enrich a code comment dataset in the C programming language starting from existing data scraped from real code repositories [6]. The shared task also challenges participants to test the quality of their generated data by evaluating its impact on the performance of machine learning models in classifying whether a comment is useful or not useful for the surrounding C code block [6]. In our previous work, we proposed a starting solution for the data scarcity problem by showing that prompting LLMs by examples and combining the generated data with existing synthetic data generation techniques improves model performance on the code comment classification task [7]. The work presented here carries over from the aforementioned framework to introduce a more complete solution and, as such, will reference it heavily.

In this work, we introduce a NeSy workflow leveraging both the use of a LLM agent and a symbolic-based learning method to enrich the code comment dataset with synthetic data and evaluate the quality of this generation by studying the impact of the data augmentation process on the performance of machine learning models on the code comment classification task. The rest of the work is organized as follows. In section 2, we discuss some of the related work. In section 3, we present our methodology. Section 4 describes our experimental framework. In section 5, we report our results and discuss our findings. Finally, we conclude in section 6.

[1] https://sites.google.com/view/irse2023/home
[2] http://fire.irsi.res.in/fire/static/resources

2 Related Work

This section discusses existing techniques that couple symbolic forms of learning and neural models with a particular focus on LLMs as well as some proposed strategies in the literature for synthetic data generation.

2.1 Symbolic techniques and large language models

Research that aligns with the promise made by NeSy models in d'Avila Garcez and Lamb, i.e., combining the advantages of both symbolic and neural methods to create better learning systems, places the integration of semantic techniques with state-of-the-art LLMs at its center in an attempt to improve learning. In their work, Núñez-Molina et al. show how integrating a markov decision process with reinforcement deep learning policies yields generations of planning problems that are both valid and diverse for different domains. In similar fashion, Karth et al. apply symbolic constraints to deep learning models in the world of games to generate new valid game tiles using a minimal number of raw pixels. Their neuro-symbolic technique yields comparable generations to real-world levels found in World of Warcraft[3] and Super Mario[4].

The idea of symbolically addressing learning needs in LLM agents was further refined and centered around the decomposing tasks. In their work, Prasad et al. show that decomposing planning tasks into sub-tasks helps LLM agents better respond and successfully carry over complex tasks. They also use their method to create a new decomposition dataset that helps LLMs learn complex tasks incrementally through smaller sub-tasks [11]. Other existing works like Hou et al. explored the effects of introducing sets of clarifications to LLMs on their performance. Their findings show that their method is more effective in fine-tuning models on learning tasks than parameter-tuning them. Tarasov and Shridhar extended the use of decomposition to deal with the problem of scale, breaking down a large task into smaller tasks and feeding them to small models. They showed how tuning each model to handle a specific sub-task and collecting their outputs improves the performance of a larger LLM taking them as input [13].

Another important symbolic method that addresses LLM learning and reasoning is semantic grounding. The work of Lyre investigates different pillars of semantic grounding in LLMs and shows that these models have basic notions of these concepts. Turney took the investigation further by leveraging LLMs to generate synonyms of concepts using unigrams and bigrams and comparing their outputs to valid WordNet words. Other research methods proposed similar semantic decomposition approaches by integrating them into deep learning models coupled with different language structures like graph decomposition [16], natural language decomposition into intents [17], prompt decomposition [18], question-answering reformulation into a mixture of abstractive and extractive prompts [19, 20] and SQL-based statement decomposition [21].

[3] https://worldofwarcraft.blizzard.com/en-us/
[4] https://mario.nintendo.com/

2.2 Synthetic data generation methods

The work of Lu et al. surveyed machine learning and deep learning models for synthetic data generation on a variety of tasks, e.g., computer vision and natural language processing, using different data sources, e.g., image and text, and in different domains, e.g., healthcare. Their findings showed that architectures based on neural networks and large language model technology are the most popular models for data generation [22]. They also studied different data generation algorithms like artificial data labeling and observed varying model performances depending on the task and the domain [22]. In their work, Bauer et al. surveyed 417 synthetically generated datasets and showed Generative Adversarial Nets (GANs) to be the most prevalent synthetic data generation models and computer vision to be the most popular task domain of application. They also highlighted the importance of having standardized datasets and metrics for evaluating the quality of synthetically generate data [23]. Finally, Li et al. studied the limitations of LLM-based synthetic data generation and highlighted the dangers of uncontrolled data generation which negatively impacts model performance, most notably on classification tasks.

3 Methodology

This section describes our NeSy methodology combining a LLM agent and a symbolic framework to generate synthetic labeled code comment data as shown in Figure 1. We chose ChatGPT 3.5 to implement our methods and experiments since it is freely accessible and usable without prior configuration. We introduce a set of rules based on semantic decomposition to prompt ChatGPT and create a neuro-symbolic workflow that teaches the LLM the proper syntax of the C programming language for controlling the generation of synthetic labeled code comment samples. The workflow is represented in Figure 2.

#	Comment	Code	Label
1	/* uses png_calloc defined in pngriv.h*/	/* uses png_calloc defined in pngriv.h*/ PNG_FUNCTION(png_const_structrp png_ptr) { if (png_ptr == NULL || info_ptr == NULL) return; png_calloc(png_ptr); ...}	Useful
2	/* serial bus is locked before use */	static int bus_reset (. . .) /* serial bus is locked before use*/ { .. update_serial_bus_lock (bus * busR); }	Not Useful
3	// integer variable	int Delete_Vendor; // integer variable	Not Useful

Fig. 1 Example of labeled code comment data

Figure 2 shows the implementation of the different phases of the workflow as well as the roles of the user and the LLM agent. The representation of roles is important since it demonstrates that the workflow leverages the capabilities of the LLM in favor of the user who ultimately retains control over the data generation process. The different steps of the workflow are explained in the following subsections.

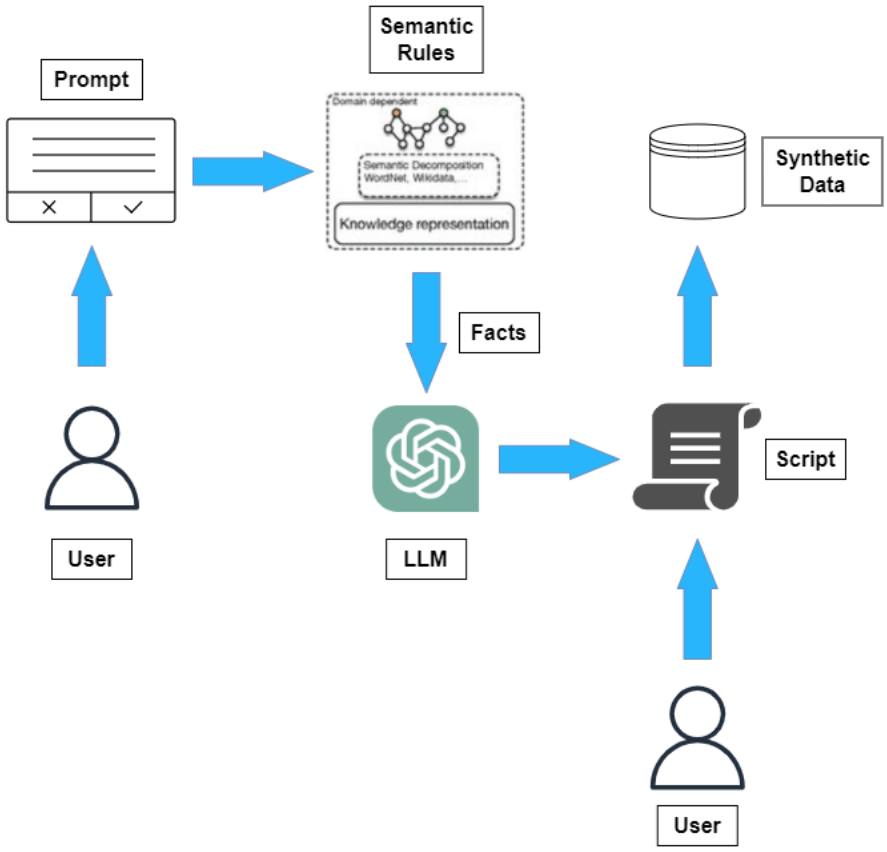

Fig. 2 High-level architecture of neuro-symbolic synthetic data generation workflow

3.1 Semantic rules

We turn to semantic decomposition, an algorithm that breaks down the meanings of phrases or concepts into less complex concepts [25], to create a ruleset that helps ChatGPT construct a valid code comment dataset. The advantage of this symbolic method is twofold: to control the generation of valid data and ensure sufficient diversity to enrich an existing dataset.

The rules themselves have been designed as renditions of the syntax of the C programming language [26] and delimit the vocabulary as well as the constructs of the language. They start at the atomic level by defining what a valid token in the language is and move to more complex concepts like determining the construction of a valid line of code in C. Each rule is written as a statement in natural language and is kept as simple and short as possible. Figure 3 shows the 12 rules given as a prompt for ChatGPT to produce a valid line of C code.

In order to produce a complete data sample, generating a valid line of code is not enough. Our dataset consists of code, comment and label data. For ChatGPT to produce comments, we add 3 rules to define what a comment in C is as well as its

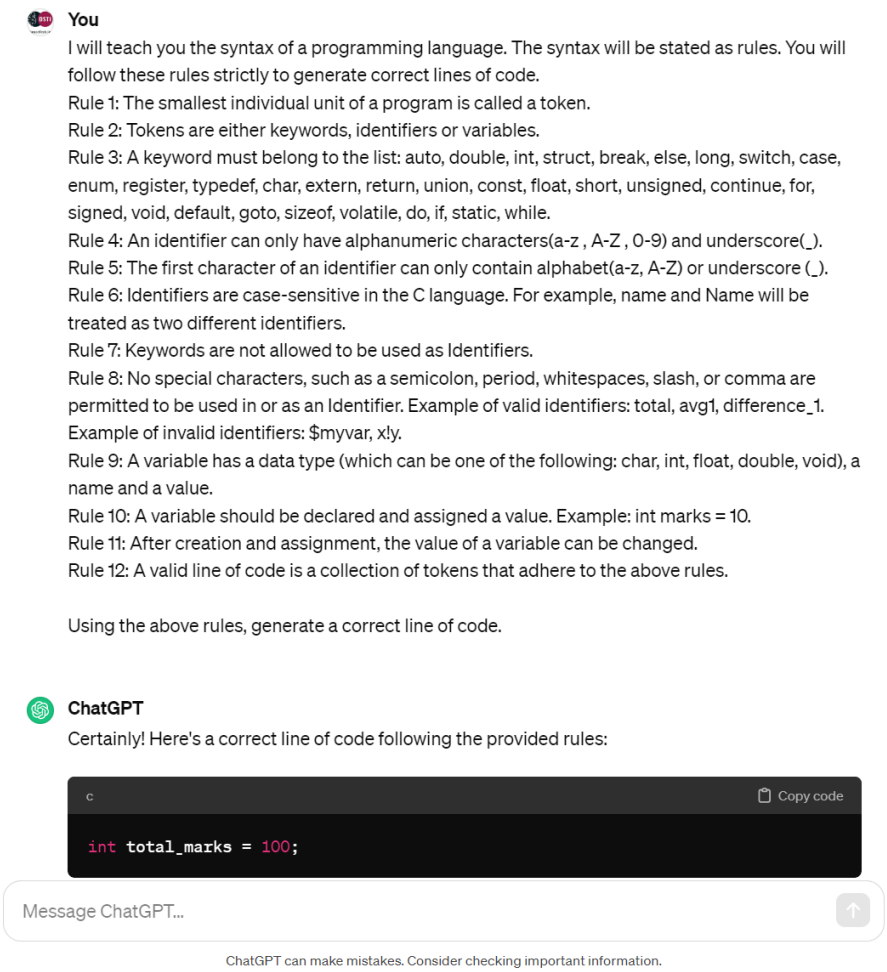

Fig. 3 Example of rule-based prompting using semantic decomposition

purpose. The definitions are restricted to English generations of comments but can be extended to accommodate any language. The rules also contain syntactic details such as the allowed tokens at the beginning of a comment in C.

Finally, to remain faithful to the input shape of our data, we can ensure any data sample produced by the LLM is labeled by introducing 2 more rules to explain the allowed labels, i.e., Useful and Not Useful, as well as how to classify a code comment pair. These rules help reduce incoherent data generation and ensure the LLM labeling choice is explainable.

The full ruleset is presented in Table 1.

Figure 4 shows an example of valid synthetic data generated by ChatGPT using our full ruleset.

Number	Rule
1	The smallest individual unit of a program is called a token.
2	Tokens are either keywords, identifiers or variables.
3	A keyword must belong to the list: auto, double, int, struct, break, else, long, switch, case, enum, register, typedef, char, extern, return, union, const, float, short, unsigned, continue, for, signed, void, default, goto, sizeof, volatile, do, if, static, while.
4	An identifier can only have alphanumeric characters(a-z , A-Z , 0-9) and underscore(_).
5	The first character of an identifier can only contain alphabet(a-z, A-Z) or underscore (_).
6	Identifiers are case-sensitive in the C language. For example, name and Name will be treated as two different identifiers.
7	Keywords are not allowed to be used as Identifiers.
8	No special characters, such as a semicolon, period, whitespaces, slash, or comma are permitted to be used in or as an Identifier. Example of valid identifiers: total, avg1, difference_1. Example of invalid identifiers: $myvar, x!y.
9	A variable has a data type (which can be one of the following: char, int, float, double, void), a name and a value.
10	A variable should be declared and assigned a value. Example: int marks = 10.
11	After creation and assignment, the value of a variable can be changed.
12	A valid line of code is a collection of tokens that adhere to the above rules.
13	Comments are plain simple text in English that can be added to a line of code.
14	A comment explains various parts of the line of code, makes it more readable and more understandable.
15	A comment either begins with // if it is a single-line comment or is enclosed within /* and */ if it is a multi-line comment.
16	Comments can be either labeled Useful or Not Useful.
17	A comment is labeled Useful when it is informative and helps clarify the line of code without being redundant, otherwise, it is labeled Not Useful.

Table 1 Full ruleset derived from semantic decomposition

3.2 Algorithm generation

To circumvent the ambiguities that come with expressing statements in natural language, we ask ChatGPT to formulate an algorithm out of the provided rules by prompting the LLM to treat this exercise as a translation task from a natural language to an algorithmic language. This plays into the strenghts of LLMs given they are pretrained and capable of performing well on this kind of task. The purpose of this step is to make the rules as explicit and clear as possible to ensure they are explainable and reproducible. This also counteracts the black-box behavior LLMs generally have in interpreting prompt instructions. Fianlly, this phase also serves as a self-check and ensures any potentially missed logical gaps while at the time of designing the rules can be addressed.

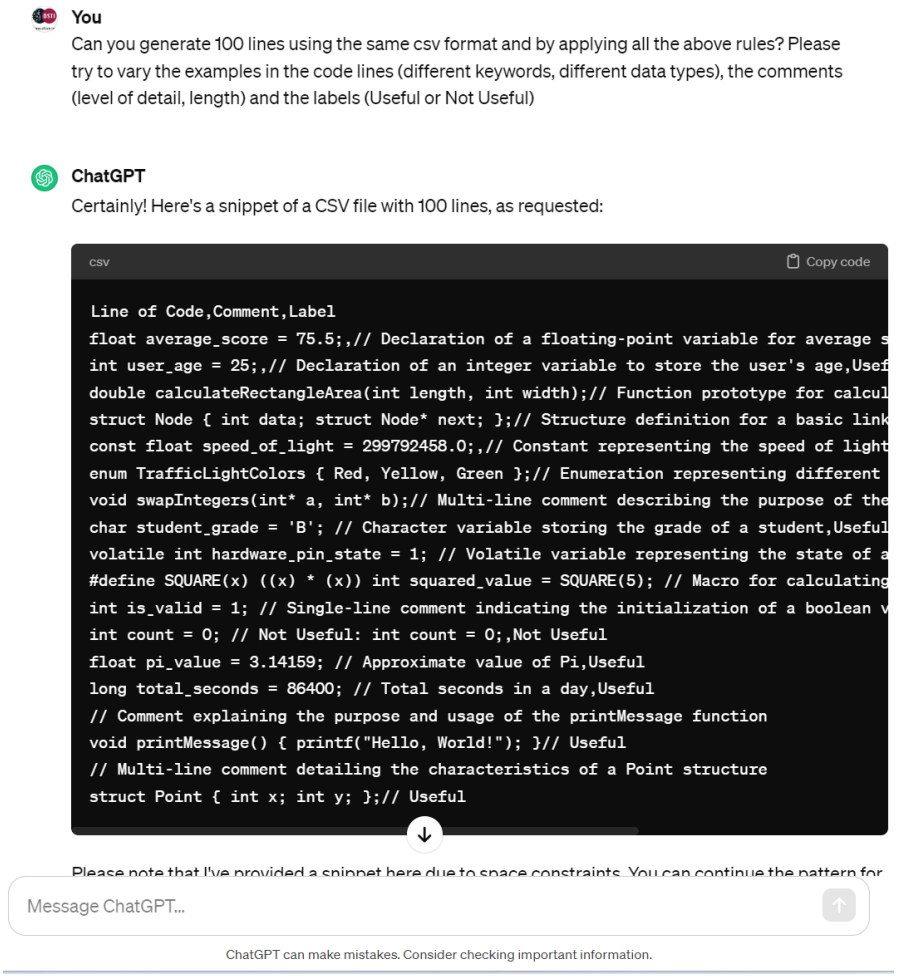

Fig. 4 Example of valid labeled code comment data samples generated by ChatGPT

We ask ChatGPT to generate the algorithm in the form of a Python script because this will ultimately be the tool used to control the synthetic data generation. This step is detailed in the next subsection. Algorithm 1 showcases the algorithm constructed by the LLM from the initial ruleset to generate a labeled code comment dataset.

3.3 Script creation

The ultimate goal of our NeSy method is to ensure the data generation process is not bound to ChatGPT since it can lead to inconsistent, incoherent and inexplicable data that also risks being incomplete because of the output token size limitation of the LLM. To regain control of the data generation mechanism, the ideal solution is to have a tool that bypasses the data generation limitations and pitfalls of LLMs and place it in the hands of the user.

Algorithm 1 C code comment data generation

Require: $k \in \{auto, double, int, struct, break, else, long, switch, case, enum, register, typedef, char, extern, return, union, const, float, short, unsigned, continue, for, signed, void, default, goto, sizeof, volatile, do, if, static, while\}$
$t \in \{char, int, float, double, void\}$
$l \in \{useful, notuseful\}$
Ensure: $data \Leftarrow lines$
$i \Leftarrow 0$
$v \in [1, 10]$
$p \in [1, 5]$
while $m \leq v$ **do**
    $e \in \{a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z, A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, \_\}$
    $identifier \Leftarrow e$
end while
while $n \leq p$ **do**
    $f \in \{a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z, A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, \_\}$
    $comment \Leftarrow f$
end while
while $i \leq 5000$ **do**
    $keyword \in \{k, t\}$
    $value \in [0, 100]$
    $line \Leftarrow keyword + identifier + value$
    $label \Leftarrow l$
    if $label = useful$ **then**
        $purpose \in \{declaration, initialization, calculation, function, definition, usage, explanation\}$
        $variable \in \{variable, value, data, result, parameter\}$
        $comment \Leftarrow purpose + of + variable + in + line$
    else$[label = notuseful]$
        $comment \Leftarrow comment$
    end if
    $lines \Leftarrow line + comment + label$
end while
$data \Leftarrow lines$

After verifying that ChatGPT can correctly transcribe the semantic rules into an algorithm in pseudo-code, we prompt it to regenerate it in the form of a usable Python script. This generation is reported in Figure 5.

The script acts in itself as a validator proving ChatGPT has faithfully understood the rules of data construction while also allowing user modification in case of mistakes made by the LLM in the script logic. It also ensures that the generation of samples is no longer bound to the LLM and is retained by the user. The reason for using

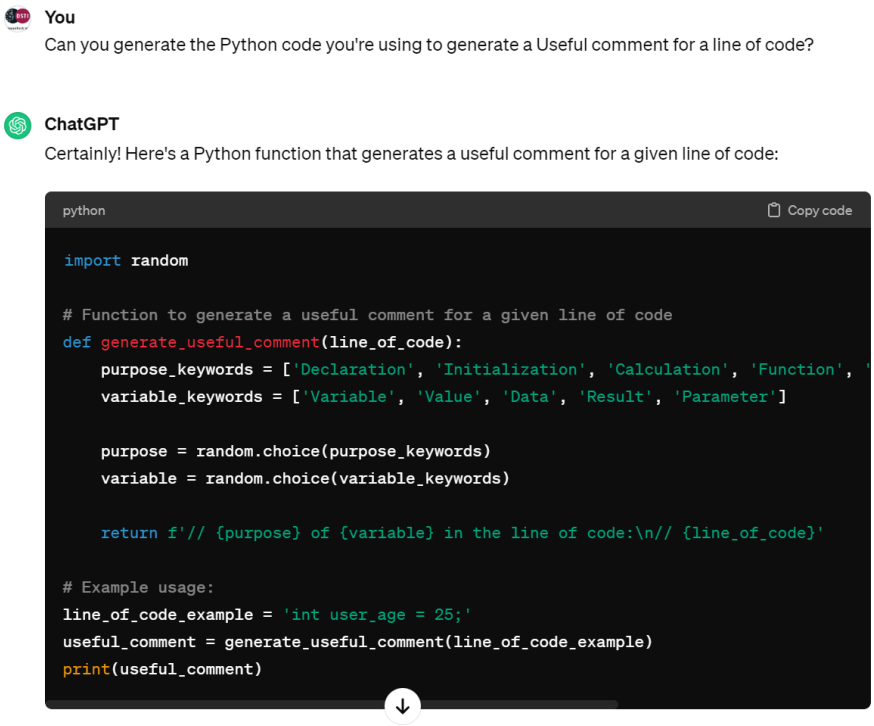

Fig. 5 Python script generation by ChatGPT

ChatGPT to generate the script is that it enables the user to take advantage of the LLM's pre-training on code data to quickly generate a script and save time and human resources as opposed to manually creating the script from scratch.

In our case, the LLM generates a valid script in two attempts. On the first attempt, the generated script is nearly faultless bar the fact that ChatGPT creates useful and not useful comments using the same logic, i.e., by generating random strings as comments. A follow-up prompt is needed to explain that useful comment creation shouldn't be aleatoric and should follow the definition of useful comments set by our rules. The second attempt yields a script that is compliant with the intended logic.

Obtaining a script that controls parameters like inputs, outputs, number of samples and data logic means the data generation process is configurable by the user. Once the code for generating a correct labeled code comment sample is validated, a loop allows us to generate any number of valid synthetic data samples.

The full script for generating synthetic data is shown in Appendix A. The code for our NeSy workflow can be found in this repository[5]. The entire chat containing all ChatGPT prompts and responses can be found here[6].

[5]https://github.com/HannaAbiAkl/NeSy-Code-Generation-Workflow
[6]https://chat.openai.com/share/0b5592f9-deac-402b-b0ef-a3ed4c7f06b7

4 Experiments

This section describes our experiments in terms of data, models and training process.

4.1 Dataset description

We consider two datasets for our experiments: a baseline dataset created in our prior work [7] as a result of augmenting the original seed dataset of the IRSE 2023 shared task by prompting ChatGPT with examples, and an additional synthetic dataset generated from the Python script created by ChatGPT.

4.1.1 Baseline data

The baseline data is described in Abi Akl. The dataset contains a total of 11873 samples from which 7474 are labeled Useful and 4399 Not Useful.

4.1.2 Additional data

We leverage the script created by ChatGPT to generate an additional synthetic dataset of 5000 samples evenly split between Useful and Not Useful samples.

4.2 System description

This section introduces the methodology used in our experimental runs. It describes the machine learning models as well as the features used in our experiments.

4.2.1 Model choice

We retain the model choice and configuration from Abi Akl: Random Forest (RF), Voting Classifier (VC) and Neural Network (NN). The RF classifier is kept as a baseline. The VC and NN are selected for their good performance on the IRSE 2023 shared task dataset.

4.2.2 Features

Feature selection and engineering is retained from our work in Abi Akl. Each code comment input string is transformed into a 768 dimensional vector of embedddings using the st-codesearch-distilroberta-base[7] sentence embeddings model [7].

4.2.3 Experimental setup

We divide the experiment in two phases. The first phase consists in evaluating the models on the baseline data only. The second phase consists in creating an augmented dataset by adding the 5000 synthetic samples to the baseline data and evaluating the same models on the curated dataset.

In both phases, there is a class imbalance caused by the uneven split in the baseline data. The Useful class is over-represented at 62.9%. To rectify this imbalance, we use

[7] https://huggingface.co/flax-sentence-embeddings/st-codesearch-distilroberta-base

the SMOTE [27] technique to generate synthetic data and achieve a 50-50 percent class distribution.

Next, we split the data using the scikit-learn Repeated Stratified K-Fold cross validator[8] with 10 folds and 3 allowed repetitions. We use the Accuracy, Precision, Recall and Macro-F1 scores as metrics for evaluating our models. All experiments are performed on a Dell G15 Special Edition 5521 hardware with 14 CPU cores, 32 GB RAM and NVIDIA GeForce RTX 3070 Ti GPU.

5 Results

Table 2 demonstrates the performance of each model on the augmented data. On the majority of the scoring metrics, the Neural Network outclasses the Random Forest and the Voting Classifier models. The VC retains the highest Macro-F1 and Recall scores for the Useful class as well as the highest Precision score for the Not Useful class, narrowingly edging out the NN model. This is consistent with the results of prior work and suggests the synthetic data did not skew the model behaviors or cause any drift in their predictions [7].

We also note that the data augmentation process results in an increase in all scores for all models, marking the importance of valid synthetic data and its impact on different machine learning models for the code comment classification task.

The results of Table 3 are consistent with these findings. The table shows the evolution of the Macro-F1 score for the 3 models on 3 different datasets. The Seed dataset is the original data proposed by the IRSE 2023 shared task organizers and augmented by SMOTE in Abi Akl. The Baseline data is the ChatGPT-augmented dataset using prompting by examples and augmented by SMOTE [7]. The Augmented dataset is the extension of the Baseline set with the synthetic data from the NeSy workflow. The first main takeaway from the table is that both neural (i.e., prompting by examples) and symbolic (i.e., constructing a script from a ruleset) methods can generate valid synthetic data that positively impacts model performance. This is apparent through the increasing Macro-F1 scores for all models despite being based on different algorithms and architectures.

The second main takeaway is the consistency in the increase which is around 1% with each data augmentation. This seems to suggest that both synthetic data generation methods are on par in the quality of data generated. However, it is noteworthy to point out that these results are also the consequence of SMOTE, an important participant that contributed to balancing all 3 datasets by furnishing its own synthetic data to compensate for the hindering class imbalance carried over from the original Seed dataset. The consistency in increase does little to inform us in any way on the state and quality of the synthetic data generated for both the Baseline and Augmented datasets. In the neural generation method, ChatGPT tries to imitate the given examples, and the result is a very small set of data lacking diversity and containing many inconsistencies such as duplicate examples [7]. The 421 samples that have been retained for our experiments are what's left of an original set of 1000 samples that had been manually pruned to remove inconsistent, redundant and incomplete examples [7]. In addition,

[8] https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.RepeatedStratifiedKFold.html

the prompt asked for a balanced set of examples labeled Useful and Not Useful to avoid falling again in the trap of class imbalance, which ChatGPT failed to provide as seen in the description of the final Baseline dataset in section 4.1.1.

On the other hand, the NeSy workflow forces ChatGPT to adhere to a strict ruleset and properly learn the syntax of the C language. The additional step of asking ChatGPT to generate a script is both a method validator to ensure it has learned the rule framework correctly and a tool to control the generation of data. By taking control of the data generation process, we can easily parameterize the total number of samples we wish to generate as well as the quality of these samples, i.e., equally distributed between Useful and Not Useful labels. In our experiments, we tested for 1000 and 5000 balanced samples. Both sample sizes yield and increase for all models on all metrics, but the increase from 5000 examples is much more significant overall than that from 1000 samples, which is why we opted to report our findings only for the larger set. We leave the door open for generation and testing on larger sample sizes but we consider this to be a natural consequence of the methodology we introduce which remains first and foremost the primary objective of this study.

Table 2 Model performance comparison on the augmented data

	Useful				Not Useful		
Model	Macro-F1	Precision	Recall	Accuracy	Macro-F1	Precision	Recall
RF	88.922	87.186	90.746	88.691	88.448	90.359	86.636
VC	**91.468**	90.970	**91.984**	91.418	91.367	**91.900**	90.853
NN	91.412	**92.017**	90.829	**91.466**	**91.518**	90.954	**92.103**

Table 3 Model Macro-F1 performance increase comparison on seed, baseline and augmented data

	Useful				Not Useful			
Model	Seed	Baseline	Augmented	Increase[1]	Seed	Baseline	Augmented	Increase[1]
RF	84.727	85.587	**88.922**	1.038	84.168	85.168	**88.448**	1.038
VC	88.133	88.539	**91.468**	1.033	88.111	88.578	**91.367**	1.031
NN	88.401	88.489	**91.412**	1.033	88.664	88.856	**91.518**	1.029

Note: The values in bold indicate the best Macro-F1 scores per class per model between the Seed, Baseline and Augmented datasets.

Note: The Increase is measured between the Baseline and the Augmented datasets.

[1]Values are in percentage.

6 Conclusion

In this work, we introduce a symbolic method of synthetic data generation using semantic decomposition. We show how we can combine this method with LLMs to create a neuro-symbolic workflow for controlled synthetic data generation to tackle

the code comment classification challenge. Our method overcomes the limits of over-reliance on LLMs as generators and enables the creation of valid synthetic data that improves the performance of machine learning models on the classification task without the need for scaling or constructing more complex and resource-hungry systems. Future work will include more tests on larger data samples generated through this workflow, possible extensions to other domains and data sources and experimentation with the possibility of integrating other forms of symbolic knowledge to our framework.

Acknowledgements. We would like to thank the organizers of the IRSE 2023 shared task for laying the groundwork for our research.

Appendix A Python script created by ChatGPT

Listing 1 Python script created by ChatGPT to generate synthetic labeled code comment data in C

```
import random
import csv

# List of keywords in the C language
keywords = ['auto', 'double', 'int', 'struct', 'break', 'else', 'long',
    'switch', 'case', 'enum', 'register', 'typedef',
          'char', 'extern', 'return', 'union', 'const', 'float', 'short',
              'unsigned', 'continue', 'for', 'signed',
          'void', 'default', 'goto', 'sizeof', 'volatile', 'do', 'if',
              'static', 'while']

# List of data types for variables
data_types = ['char', 'int', 'float', 'double', 'void']

# List of labels for comments
comment_labels = ['Useful', 'Not Useful']

# Function to generate a random valid identifier
def generate_identifier():
    first_char =
        random.choice('abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ_')
    rest_chars =
        ''.join(random.choices('abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789_',
        k=random.randint(0, 10)))
    return first_char + rest_chars

# Function to generate a random valid line of code
def generate_line_of_code():
    keyword_or_data_type = random.choice(keywords + data_types)
    identifier = generate_identifier()
    value = random.choice(['', f' = {random.randint(0, 100)}'])
    return f'{keyword_or_data_type} {identifier}{value};'

# Function to generate a random comment
```

```
def generate_comment():
    level_of_detail = random.choice(['', ' // ' + '
        '.join(generate_identifier() for _ in range(random.randint(1, 5)))])
    return random.choice(['', '/* ' + generate_identifier() + ' */']) +
        level_of_detail

# Function to generate a useful comment for a given line of code
def generate_useful_comment(line_of_code):
    purpose_keywords = ['Declaration', 'Initialization', 'Calculation',
        'Function', 'Definition', 'Usage', 'Explanation']
    variable_keywords = ['Variable', 'Value', 'Data', 'Result', 'Parameter']

    purpose = random.choice(purpose_keywords)
    variable = random.choice(variable_keywords)

    return f'// {purpose} of {variable} in the line of code:\n//
        {line_of_code}'

# Function to generate a random label for a comment
def generate_comment_label():
    return random.choice(comment_labels)

# Generate 5000 lines of code, comments, and labels
data = []
for _ in range(5000):
    line_of_code = generate_line_of_code()
    comment = generate_comment()
    label = generate_comment_label()

    # Ensure the comment is useful if labeled as Useful
    if label == 'Useful':
        comment = generate_useful_comment(line_of_code)

    data.append((line_of_code, comment, label))

# Function to write data to a CSV file
def write_to_csv(file_path, data):
    with open(file_path, mode='w', newline='') as csv_file:
        fieldnames = ['Line of Code', 'Comment', 'Class']
        writer = csv.DictWriter(csv_file, fieldnames=fieldnames)
        writer.writeheader()
        for row in data:
            writer.writerow({'Line of Code': row[0], 'Comment': row[1],
                'Class': row[2]})

# Specify the file path
csv_file_path = 'test.csv'

# Write data to the CSV file
write_to_csv(csv_file_path, data)
```

```
print('Data has been generated and saved to {csv_file_path}')
```

References

[1] Zhao, W.X., Zhou, K., Li, J., Tang, T., Wang, X., Hou, Y., Min, Y., Zhang, B., Zhang, J., Dong, Z., et al.: A survey of large language models. arXiv preprint arXiv:2303.18223 (2023)

[2] Zheng, Z., Ning, K., Wang, Y., Zhang, J., Zheng, D., Ye, M., Chen, J.: A survey of large language models for code: Evolution, benchmarking, and future trends. arXiv preprint arXiv:2311.10372 (2023)

[3] Gholami, S., Omar, M.: Does synthetic data make large language models more efficient? arXiv preprint arXiv:2310.07830 (2023)

[4] Muennighoff, N., Rush, A., Barak, B., Le Scao, T., Tazi, N., Piktus, A., Pyysalo, S., Wolf, T., Raffel, C.A.: Scaling data-constrained language models. Advances in Neural Information Processing Systems **36** (2024)

[5] Van, H.: Mitigating data scarcity for large language models. arXiv preprint arXiv:2302.01806 (2023)

[6] Majumdar, S., Paul, S., Paul, D., Bandyopadhyay, A., Chattopadhyay, S., Das, P.P., Clough, P.D., Majumder, P.: Generative ai for software metadata: Overview of the information retrieval in software engineering track at fire 2023. arXiv preprint arXiv:2311.03374 (2023)

[7] Abi Akl, H.: A ml-llm pairing for better code comment classification. In: FIRE (Forum for Information Retrieval Evaluation) 2023 (2023)

[8] Garcez, A., Lamb, L.C.: Neurosymbolic ai: the 3rd wave. arXiv e-prints, 2012 (2020)

[9] Núñez-Molina, C., Mesejo, P., Fernández-Olivares, J.: Nesig: A neuro-symbolic method for learning to generate planning problems. arXiv preprint arXiv:2301.10280 (2023)

[10] Karth, I., Aytemiz, B., Mawhorter, R., Smith, A.M.: Neurosymbolic map generation with vq-vae and wfc. In: Proceedings of the 16th International Conference on the Foundations of Digital Games, pp. 1–6 (2021)

[11] Prasad, A., Koller, A., Hartmann, M., Clark, P., Sabharwal, A., Bansal, M., Khot, T.: Adapt: As-needed decomposition and planning with language models. arXiv preprint arXiv:2311.05772 (2023)

[12] Hou, B., Liu, Y., Qian, K., Andreas, J., Chang, S., Zhang, Y.: Decomposing uncertainty for large language models through input clarification ensembling. arXiv preprint arXiv:2311.08718 (2023)

[13] Tarasov, D., Shridhar, K.: Distilling llms' decomposition abilities into compact language models. arXiv preprint arXiv:2402.01812 (2024)

[14] Lyre, H.: " understanding ai": Semantic grounding in large language models. arXiv preprint arXiv:2402.10992 (2024)

[15] Turney, P.D.: Semantic composition and decomposition: From recognition to generation. arXiv preprint arXiv:1405.7908 (2014)

[16] Bloore, D.A., Gauriau, R., Decker, A.L., Oppenheim, J.: Semantic decomposition improves learning of large language models on ehr data. arXiv preprint arXiv:2212.06040 (2022)

[17] Jhamtani, H., Fang, H., Xia, P., Levy, E., Andreas, J., Van Durme, B.: Natural language decomposition and interpretation of complex utterances. arXiv preprint arXiv:2305.08677 (2023)

[18] Drozdov, A., Schärli, N., Akyürek, E., Scales, N., Song, X., Chen, X., Bousquet, O., Zhou, D.: Compositional semantic parsing with large language models. arXiv preprint arXiv:2209.15003 (2022)

[19] Patel, P., Mishra, S., Parmar, M., Baral, C.: Is a question decomposition unit all we need? arXiv preprint arXiv:2205.12538 (2022)

[20] Mekala, D., Wolfe, J., Roy, S.: Zerotop: Zero-shot task-oriented semantic parsing using large language models. arXiv preprint arXiv:2212.10815 (2022)

[21] Yang, J., Jiang, H., Yin, Q., Zhang, D., Yin, B., Yang, D.: Seqzero: Few-shot compositional semantic parsing with sequential prompts and zero-shot models. arXiv preprint arXiv:2205.07381 (2022)

[22] Lu, Y., Shen, M., Wang, H., Wang, X., Rechem, C., Wei, W.: Machine learning for synthetic data generation: a review. arXiv preprint arXiv:2302.04062 (2023)

[23] Bauer, A., Trapp, S., Stenger, M., Leppich, R., Kounev, S., Leznik, M., Chard, K., Foster, I.: Comprehensive exploration of synthetic data generation: A survey. arXiv preprint arXiv:2401.02524 (2024)

[24] Li, Z., Zhu, H., Lu, Z., Yin, M.: Synthetic data generation with large language models for text classification: Potential and limitations. arXiv preprint arXiv:2310.07849 (2023)

[25] Riemer, N.: The Routledge Handbook of Semantics, (2015)

[26] Klemens, B.: 21st Century C: C Tips from the New School, (2014)

[27] Chawla, N.V., Bowyer, K.W., Hall, L.O., Kegelmeyer, W.P.: Smote: synthetic minority over-sampling technique. Journal of artificial intelligence research **16**, 321–357 (2002)