Knowledge-Enhanced Program Repair for Data Science Code

Shuyin Ouyang
King's College London
London, UK
shuyin.ouyang@kcl.ac.uk

Jie M. Zhang
King's College London
London, UK
jie.zhang@kcl.ac.uk

Zeyu Sun
Institute of Software, Chinese Academy of Sciences, Beijing, China
zeyu.zys@gmail.com

Albert Merono Penuela
King's College London
London, UK
albert.merono@kcl.ac.uk

***Abstract*—This paper introduces DSrepair, a knowledge-enhanced program repair approach designed to repair the buggy code generated by LLMs in the data science domain. DSrepair uses knowledge graph based RAG for API knowledge retrieval and bug knowledge enrichment to construct repair prompts for LLMs. Specifically, to enable knowledge graph-based API retrieval, we construct DS-KG (Data Science Knowledge Graph) for widely used data science libraries. For bug knowledge enrichment, we employ an abstract syntax tree (AST) to localize errors at the AST node level. We evaluate DSrepair's effectiveness against five state-of-the-art LLM-based repair baselines using four advanced LLMs on the DS-1000 dataset. The results show that DSrepair outperforms all five baselines. Specifically, when compared to the second-best baseline, DSrepair achieves substantial improvements, fixing 44.4%, 14.2%, 20.6%, and 32.1% more buggy code snippets for each of the four evaluated LLMs, respectively. Additionally, it achieves greater efficiency, reducing the number of tokens required per code task by 17.49%, 34.24%, 24.71%, and 17.59%, respectively.**

***Index Terms*—Code Repair, Large Language Model, Knowledge Graph, Data Science**

I. Introduction

Data science is crucial in driving innovation and decision-making across various domains [1], [2], leveraging data to uncover insights and inform strategic actions. Nevertheless, the complexity of data science libraries and the expertise required to use them can pose significant barriers to lay users. Large Language Models (LLMs) have emerged as powerful tools to generate data science code automatically [3]–[5], democratizing access and accelerating development processes. Despite their potential, the widely acknowledged shortcomings with LLMs, such as hallucination and the lack of specialized knowledge of certain domains (e.g. long-tail API usage) [6], [7] and specific code context [8], remain significant obstacles. These issues are particularly critical in the data science domain, where the code heavily relies on libraries for accurate and efficient data processing and analysis, making precision and contextual accuracy essential for robust outcomes. Existing studies have applied feedback-based iterative self repair [9]–[13] to improve the reliability of LLM-generated code. Nevertheless, these approaches are not designed for data science code.

Recently, Retrieval-Augmented Generation (RAG) [14] has also emerged as a widely-adopted technique to inject external knowledge into LLMs to facilitate more coherent code generation. RAG combines the strengths of information retrieval and LLMs to enhance code generation. Existing RAG-based code generation studies [15]–[18] commonly follow a standard RAG architecture, where the "retriever" component retrieves relevant **plain text** from a vast corpus or database using a vector similarity search. This retrieved information is then fed into an LLM, which uses this context to produce more accurate and relevant code [19]–[22]. However, these text-based RAG approaches are not well-suited for code generation tasks, as they rely on unstructured or semi-structured plain text, which lacks the semantic relationships and structured representation needed for complex code understanding and generation. Specifically, 1) text-based retrieval relies on vector similarity search, which often retrieves irrelevant or loosely related information due to ambiguities in natural language; 2) plain text does not explicitly represent the relationships between APIs, their dependencies, or their attributes (e.g., parameters, return types). As a result, text-based RAG approaches may fail to provide the comprehensive contextual knowledge required for resolving an issue; 3) retrieved plain text often contains redundant descriptions or ambiguities, which can confuse large language models (LLMs) or lead to suboptimal code generation.

This paper introduces DSrepair, a knowledge-enhanced approach for repairing incorrect data science code produced by LLMs through knowledge graph-based RAG and bug information enrichment. We construct DS-KG (**D**ata **S**cience **K**nowledge **G**raph), a set of knowledge graphs for the seven most widely adopted data science libraries (i.e., NumPy, Pandas, SciPy, Scikit-learn, Matplotlib, PyTorch, and TensorFlow) [3]–[5], [23], [24]. For the buggy code generated by an LLM, DSrepair uses the API name appeared in the code as the query, to obtain the correct usage of corresponding API functions by accessing DS-KG. It then uses the returned result to guide the LLM in repairing code.

Compared to text-based RAG, our KG-based RAG naturally captures more complex relationships and dependencies within API documents, which is essential in the data science domain. The rich semantic relationships stored in knowledge graphs enhance the reliability and efficiency of code generation and repair. For instance, in Matplotlib's API document, an API named *matplotlib.pyplot.subplots*, has a parameter called *gridspec_kw*. The value of *gridspec_kw* must be passed to another API object, called *matplotlib.gridspec.GridSpec*. If an error occurs in *gridspec_kw*, querying knowledge from *matplotlib.gridspec.GridSpec* is more helpful than querying only from *matplotlib.pyplot.subplots*. A well-designed KG can

1558-1225/25/$31.00 ©2025 IEEE
DOI 10.1109/ICSE55347.2025.00246

infer such dependency naturally.

DSrepair also uses enhanced bug information to improve the program repair effectiveness. Data science code often contains multiple function calls and data operations within a single line. Therefore, to obtain fine-grained bug information, DSrepair uses Abstract Syntax Tree (AST) and test case execution information to localize errors at the AST node level.

We evaluate DSrepair on two widely used general-purpose LLM (i.e., GPT-3.5-turbo and GPT-4o-mini [25]) and two state-of-the-art coding LLMs (i.e., DeepSeek-Coder [26] and Codestral [27]) based on DS-1000 [5], the data science code generation benchmark spanning seven data science libraries. Our results show that DSrepair outperforms all the five baseline LLM-based repairing approaches. In particular, compared to the second-best baseline, DSrepair correctly fixes 44.4%, 14.2%, 20.6%, and 32.1% more buggy code snippets for the four LLMs, respectively, while reducing token usage per code task by 17.49%, 34.24%, 24.71%, and 17.59%.

To summarize, this paper makes the following contributions:

- We present DSrepair, a novel LLM-based program repair approach for data science code, leveraging knowledge graph-based RAG and enriched bug information.
- We construct and release DS-KG (Data Science Knowledge Graph), a comprehensive set of knowledge graphs tailored to the seven most widely used data science libraries.
- We propose an AST-based bug information enriching approach that can pinpoint errors at the AST node level.
- We conduct an empirical study using four LLMs and five baselines, demonstrating that DSrepair significantly outperforms all baselines in repairing data science code.

We release our data, code, KG data dump, and results at our homepage [28]. The rest of the paper is organized as follows. Section II outlines our methodology. Section III describes the design of the experiments, including research questions, benchmarks, baselines, selected models, and measurements. Section IV presents the results and highlights notable findings based on our empirical results. Section V discusses the threat to validity, the limitation, and the generalizability of our work. Section VI introduces the related work of our study. Section VII concludes.

II. Method

Fig 1 shows an overview of DSrepair. Given a code problem description, we first let LLM (i.e. GPT-3.5-turbo) generate code. If the code fails to pass the test cases, DSrepair constructs a repair prompt and requests LLM to regenerate the code (see Section III-B for details). As shown in the figure, DSrepair involves four main steps: API KG Construction, where a knowledge graph (DS-KG) is built for popular data science libraries (e.g., NumPy and Pandas) to capture detailed API usage and relationships; API Knowledge Retrieval, where API calls are extracted from the buggy code and queried from DS-KG, with the results verbalized into natural language for LLM prompts; Bug Knowledge Enrichment, which localizes errors at the AST node level using test case execution to provide fine-grained bug information; and Prompt Construction,

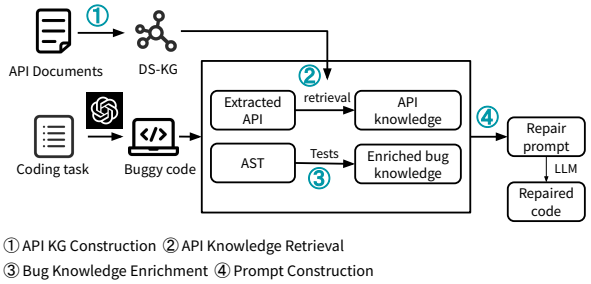


Fig. 1: Overview of DSrepair.

where all gathered information is structured into a detailed prompt to guide the LLM in generating effective repairs.

A. API KG Construction

We develop DS-KG, a knowledge graph tailored to widely adopted data science libraries such as NumPy, Pandas, SciPy, Scikit-learn, Matplotlib, PyTorch, and TensorFlow. Its primary purpose is to assist LLMs in repairing buggy code by providing structured information about the correct usage of APIs. Following standard KG construction procedures, we begin by creating an ontology to define the schema for DS-KG [29], [30]. Existing ontologies are unsuitable for representing API documentation in the context of data science code repair. To address this gap, we manually design a domain-specific ontology schema, drawing insights from the structure of API documentation. API documentation typically provides details such as an API's name, expression, explanation, parameters, and return types. Our ontology captures these attributes for individual API functions, enabling precise and structured queries based on error information extracted from buggy code. Inspired by prior work in code ontology design [31], [32], we represent each API function as a unique entity within DS-KG. The ontology includes two types of relations: (1) Attribute Relations, describe links between API entities and their attributes, such as: '*has_name*', '*has_expression*', and '*has_explanation*'[1]. (2) Dependency Relations, capture the hierarchical structure and dependencies of APIs, such as '*belongsToLibrary*' and '*belongsToModule*'.

Fig 2 illustrates an example of DS-KG construction from the NumPy API document. Each API object introduced on a webpage, such as *numpy.flipud* and *numpy.array_split*, is treated as an entity. Detailed information about an API object, such as its name, expression, explanation, and parameters & returns, is used to build RDF triples with attribute relations. For example, the API object *numpy.flipud* has such an RDF triple, <*numpy.flipud*, *has_expression*, "*numpy.flipud(m)*">. New entities are created for the parameters and return values of each API object, each with attributes like argument position, data type, and explanation. For example, the parameter *m* in "*numpy.flipud(m)*" has the following RDF triple: <*numpy.flipud_parameter_m*, *hasType*, "*array_like*">. Using the prefix of the API entity (derived from the name and

[1]In this paper, we ignore the OWL prefixes in RDF triples (<subject, predicate, object>) to make the article more concise.

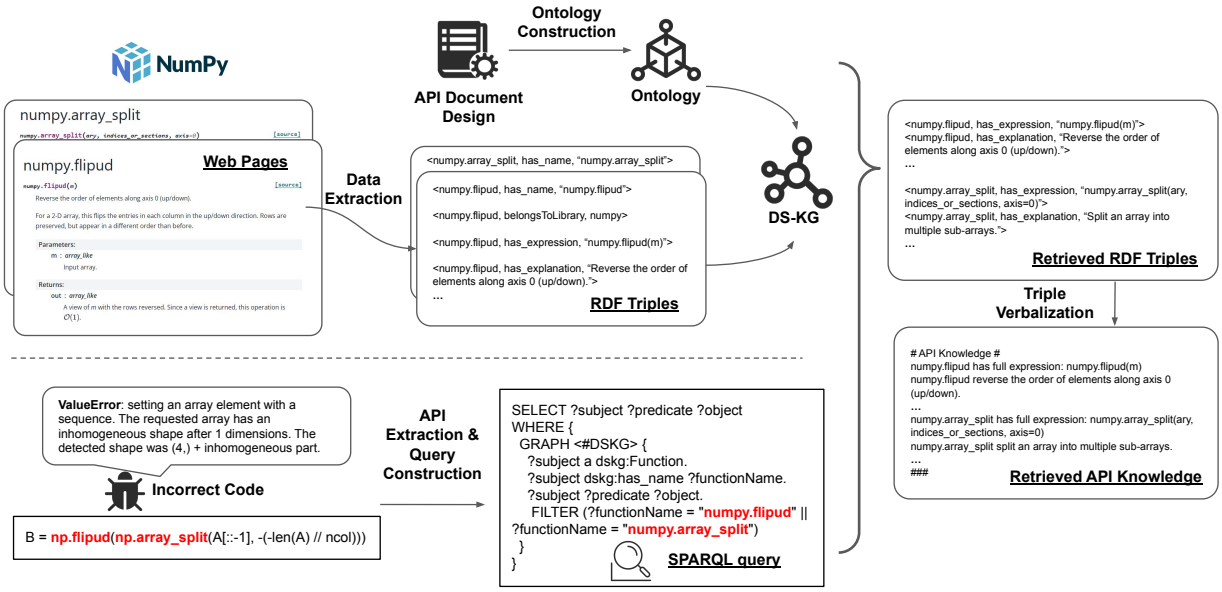


Fig. 2: Details on API KG construction (Step 1) and API Knowledge Retrieval (Step 2) in DSrepair. The code raises an error because of the mismatched array shape between *np.flipud*'s required input and *np.array_split*'s output. DSrepair extracts the API call in the error line and builds a SPARQL query to search the relevant RDF triples in the DS-KG, which is constructed from API documents and guided by the ontology. Finally, DSrepair maps the returned RDF triples to natural language, which will be used as a part of the repair prompt.

webpage URL), we construct RDF triples with dependency relations. For instance, the API object *numpy.flipud* is linked to its library through the RDF triple: <*numpy.flipud*, *belongsToLibrary*, *numpy*>.

B. API Knowledge Retrieval

DSrepair integrates DS-KG to enhance the repair of buggy code by retrieving relevant API knowledge and incorporating it into the repair process. DSrepair extracts all API invocations in the buggy code snippet using regular expressions (e.g., identifying '*np.flipud*' or '*np.array_split*'). It resolves API prefixes (e.g., mapping '*np*' to '*numpy*') and uses the full API name for queries, accounting for the common use of abbreviations in data science libraries. Using the resolved API name, DSrepair constructs SPARQL queries [33] to retrieve RDF triples from DS-KG. These triples encapsulate knowledge specific to the queried API, such as its attributes, dependencies, and parameter details. To ensure compatibility with LLMs, we transform the retrieved RDF triples into natural language sentences using triple verbalization techniques [34]. These sentences provide human-readable explanations, including a description of the API's purpose and syntax, details about parameters and returns together with their data types and explanations. The retrieved API knowledge is concatenated and included in the "API Knowledge" section of the repair prompt provided to the LLM.

In Section IV-F, we demonstrate that incorporating only the full API expression as knowledge yields the best performance for data science code repair. Thus, by default, DSrepair includes the full API expression in the prompt. This approach balances the richness of information and efficiency, ensuring LLMs receive sufficient contextual guidance without overwhelming them with unnecessary details.

C. Bug Knowledge Enrichment

Bug knowledge enrichment aims to provide LLMs with extra bug information to help LLMs better repair the bug without requesting extra tests. We use only the example tests provided in the coding task description. Traditional fault localization approaches such as spectrum-based fault localization [35] and mutation-based fault localization [36] are not applicable here for two reasons. First, data science code generation benchmarks usually provide a very limited number of tests (e.g., 1.6 tests on average per problem in DS-1000) since the annotators need to define program inputs with complex objects such as square matrices, classifiers, or dataframes [5]; second, traditional approaches are often file-level or line-level fault localization, while data science code tends to contain multiple function calls and data operations in one line. Therefore, different from traditional approaches, DSrepair uses AST-node level bug information to provide LLMs with more fine-grained bug information. Fig. 3 shows a specific example of our bug information enrichment procedure.

Firstly, test cases are extracted from the coding task description provided. These tests are essential for validating the correctness of the code and are used later in the bug knowledge enrichment process. We then transform the incorrect code snippet into its AST representation. Once the AST is generated, DSrepair iterates within a namespace that includes all necessary libraries and the extracted test cases. This iteration involves executing nodes in the AST sequentially.

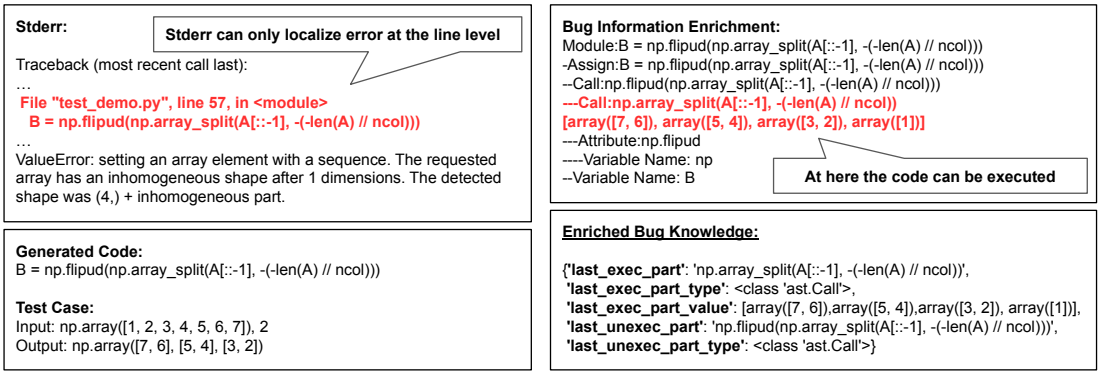


Fig. 3: Bug knowledge enrichment example. Stderr (standard error information) can only localize the bug at the line level, while our bug knowledge enrichment could enrich the error information to the AST node level.

To gain detailed bug information, the system identifies the last unexecuted node in the AST. We classify all the bugs into two categories: 1) Runtime Errors. If the code contains bugs that prevent it from being executed, the system will run each AST node until it encounters an error. The AST node that was executable before the failure occurred is noted as the last executed node. The node immediately following this, which causes the failure, is where the bug is likely located. The error is between these two nodes: the last executable node and the first unexecutable node. 2) Assertion Errors. If the entire code can be executed but the results do not match the expected output, such an issue can be due to an assertion error. In this case, the system captures the final value returned by the code execution. By comparing this actual output with the desired result, the system can provide information to LLMs about why the code is incorrect. The comparison highlights discrepancies, offering insights into potential logical errors in the code.

D. Prompt Construction

This step uses information obtained from the previous steps and organizes it into a structured prompt [37], which is then fed to the LLM for code repair. As shown in Fig 4, the final prompt includes the following components: problem description, incorrect code, stderr information, API knowledge, bug knowledge, fact-checking, and response format.

We first put the problem description and LLM generated incorrect code in the prompt. Error messages are cleaned by removing local file paths and deleting warnings. To some extent, this action protects the privacy of users' operation system environment and ensures that only relevant error information is included, focusing on critical errors that hinder code execution.

We extract useful API knowledge from the DS-KG query results, specifically the API expression or signature. This expression includes all parameters both compulsory and optional highlighting potential errors related to parameter usage and function calls. This comprehensive parameter information is crucial as it often points to the source of errors in the code.

For bug knowledge, we leverage the results from bug enrichment. This involves providing the test case, the last unexecutable AST node, and the last executable AST node along with the executed result value. By comparing the actual output with the desired output, we can pinpoint the exact location and nature of the error. This detailed local context helps in understanding the specific issues within the code.

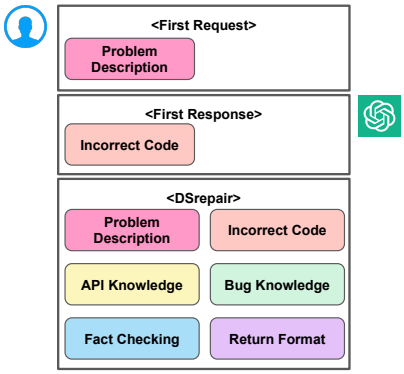


Fig. 4: DSrepair prompt example. The prompt contains structural and rich information to guide LLMs for code generation.

The fact-checking component identifies where the existing code logic violates the corresponding requirements outlined in the problem description. This step is essential to redirect the LLM's attention back to the problem description, ensuring that the solution aligns with the original requirements and constraints. The complete prompts that we use are available on our homepage [28].

III. Experimental Design

A. Research Questions

Our evaluation answers the following questions:

RQ1: How effective is DSrepair in repairing LLM-generated data science code? This RQ investigates the buggy code fix rate of DSrepair compared with other state-of-the-art program repair approaches.

RQ2: How do DSrepair's bug fixes overlap with the five baselines? This RQ investigates whether DSrepair could fix unique bugs that the baselines fail to address.

RQ3: What is the cost of DSrepair? This RQ investigates token usage and money spent on DSrepair and our baselines.

RQ4: How is DSrepair's performance affected by different prompt designs? To understand how different prompt designs

affect DSrepair, we conduct an ablation study to analyze each key prompt component's contribution to DSrepair.

RQ5: How do different knowledge retrieval approaches affect DSrepair? This RQ aims to explore the advantage of knowledge graph-based RAG against plain text-based RAG.

RQ6: How does the richness of API knowledge affect DSrepair? This RQ studies whether different types of API knowledge (e.g., whether the knowledge contains explanation or parameters) given in DSrepair will affect its performance.

RQ7: How does the non-determinism of LLM affect our experiment results? This RQ studies the influence of LLM's inherent randomness on our experiment results.

B. Data Science Benchmark

Our evaluation uses DS-1000 [5], the state-of-the-art benchmark specifically designed for benchmarking LLMs in data science code generation. The DS-1000 benchmark was specifically constructed to mitigate concerns about data leakage. In particular, the dataset applies perturbation (e.g., text rephrasing and semantic perturbation), so that models cannot answer them correctly by memorizing the solutions from pre-training [5].

Other data science code generation benchmarks [23], [38] are not applicable because they are not based on realistic problems and have no dedicated test cases to evaluate the correctness of the code (they use Exact Match [39] or BLEU score [40]). DS-1000 comprises one thousand diverse and practical data science problems sourced from StackOverflow, covering seven essential Python libraries: Numpy [41], Pandas [42], Scipy [43], Sklearn [44], Matplotlib [45], PyTorch [46], and TensorFlow [47]. The version of each library can be found on our homepage [28].

In our experiments, we let GPT-3.5-turbo generates code for each of the 1000 coding tasks in DS-1000, 562 of the generated programs fail to pass the test cases, and are regarded as repair targets of DSrepair.

C. Baseline

As far as we know, there are no existing repair approaches that are specifically designed for data science code generation. Therefore, in our evaluation, we compare DSrepair against the following state-of-the-art LLM-based approaches that are capable of repairing general types of code. While these general-purpose approaches are effective in many scenarios, they are not tailored to address the unique challenges posed by data science-specific bugs. By addressing the distinct requirements of data science code, we aim to demonstrate DSrepair's enhanced ability to handle data science-specific repair tasks in comparison to these baselines.

Code-Search [48]: Code-Search guides code repair by searching for similar code in the code base and adding the search result as a suggestion to the prompt. Following the practice in the paper, we use the code problem description as the query, and Lucene [49] as searching engine to conduct code search in the code base PyTorrent [50].

Chat-Repair [9]: Chat-Repair leverages the code execution result to check code correctness. If the code cannot pass the test cases, Chat-Repair incorporates the execution results in the prompt, to provide richer information for code debugging.

Self-Debugging-S [11]: Self-Debugging-S (S represents Simple) enriches the prompt with the simplest information, a sentence that indicates the code's correctness without more detailed information. For instance, "The generated code is incorrect. Please fix the code."

Self-Debugging-E [11]: Self-Debugging-E (E represents Explanation) first requests LLM to generate a line-by-line explanation about intermediate execution steps of the generated code. Then, it requests LLM again to generate code, based on the line-by-line explanation of the incorrect code.

Self-Repair [51]: Self-Repair first leverages error information produced by test execution to make LLM produce a short explanation of why the code failed. Then, it uses the explanation as part of the prompt to request LLM to improve the incorrectly generated code.

D. LLMs

We use two widely used general-purpose LLMs (GPT-3.5-turbo and GPT-4o-mini [25]) and two state-of-the-art coding LLMs (DeepSeek-Coder [26] and Codestral [27])[2]. We access all the LLMs by using their commercial APIs. The details of the four LLMs are shown in Table 1. We choose Python as our programming language for the code generation tasks because DS-1000 is based on Python.

To control the randomness, we set the temperature of all the LLMs to 0. For each approach, we let the LLMs generate code for each coding task ten times[3]. We select the result with median overall performance as the final result [52]. Our RQ7 in Section IV-G is about the influence of LLM's inherent randomness on our experiment results.

Table 1: LLMs used in the evaluation.

LLM	Version	Input Token Price	Output Token Price
GPT-3.5-turbo	GPT-3.5-turbo-0125	\$0.50/1M tokens	\$1.50/1M tokens
GPT-4o-mini	GPT-4o-mini-2024-07-18	\$0.15/1M tokens	\$0.60/1M tokens
DeepSeek-Coder	DeepSeek-Coder-V2	\$0.14/1M tokens	\$0.28/1M tokens
Codestral	Codestral-2405	\$1.00/1M tokens	\$3.00/1M tokens

E. Measurement

We introduce the following metrics for measuring the performance of DSrepair and baselines.

Effectiveness: We measure the effectiveness of different approaches by checking their capability in fixing incorrectly generated code, including the Absolute Number of Fixes (ANF) and Fix Rate (FR). The former is the absolute number of coding tasks whose code is successfully fixed. The latter is the ratio of ANF against all the buggy code snippets. For ANF, two of the authors conduct manual verification on the correctness of the patches to make sure that the reported fixes are not overfitted.

Cost: We measure the cost by Token Usage (TU) and Money Spent (MS), which are the most widely used metrics

[2]We do not use GPT-4 because it comes at a significantly higher cost.

[3]We repeat experiments for Deepseek-Coder-V2 three times only, because the API is no longer available after 2024/09/05.

for measuring cost for LLM-based approaches [9]. TU refers to the total token usage when using LLM to finish one complete request on average, including input token usage and output token usage. MS refers to the money cost for LLM to receive and return those tokens. Below is the formula for the MS:

$$MS = \sum_{n=1}^{N} (Token_{i,n} \times P_i + Token_{o,n} \times P_o)$$

where P_i and P_o refer to the input and output token price, $Token_{i,n}$ and $Token_{o,n}$ refer to the input token usage and output token usage at certain request n, and N refers to the total number of LLM requests.

IV. Results

This section introduces the experimental results as well as the analysis and findings for each RQ.

A. RQ1: Effectiveness of DSrepair

To answer RQ1, we report the results of Absolute Number of Fixes (ANF) and Fix Rate (FR) for DSrepair and all the baselines with each LLM. DSrepair initially generates 555 patches that successfully pass the tests from all the LLMs. After manual checking, two of the patches generated by GPT-4o-mini are overfitted[4] and have been removed from the repaired set. Table 2 shows the ultimate results.

We can observe that DSrepair significantly outperforms all the baselines in terms of ANF and FR across all four LLMs we study. Specifically, DSrepair can fix the buggy code for 104, 145, 164, and 140 coding tasks for the four LLMs, respectively, while the second-best results are 72, 127, 136, and 106, respectively.

For specific data science libraries, DSrepair outperforms the baselines for most libraries. For example, for GPT-3.5-turbo, DSrepair has the highest fix rate in Numpy, Scipy, Sklearn, Matplotlib, and PyTorch. For Codestral, DSrepair performs the best on Numpy, Pandas, Sklearn, Matplotlib, and PyTorch.

Fig 5 shows an example from Codestral where the error can be solved by DSrepair, but cannot be solved by Self-Repair. The purpose of this code problem is to only turn on minor ticks on the x-axis. Self-Repair generates an incorrect fix, while DSrepair generates the correct fix. In this problem, the buggy code uses the function *plt.minorticks_on* (short for *matplotlib.pyplot.minorticks_on*), with parameter *axis='x'*. However, as stated in the Matplotlib official document, the full expression of *plt.minorticks_on* is *matplotlib.pyplot.minorticks_on()* with no parameters, which means that *plt.minorticks_on* can control the display of minor ticks on both x-axis and y-axis, but there is no optional parameter to control the display on x-axis or y-axis only. With DSrepair, by enriching the prompt with knowledge of how to use *plt.minorticks_on* correctly, LLM is more likely to realize that putting parameters in function *plt.minorticks_on* is incorrect. The correct solution uses *plt.gca().xaxis.set_minor_locator()* instead to reach the goal of the code problem.

[4]An overfitted patch passes the test cases but is actually incorrect.

Problem No. 514 (Library category: Matplotlib):

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
x = np.random.rand(10)
y = np.random.rand(10)
plt.scatter(x, y)
# how to turn on minor ticks on x axis only
# SOLUTION START
```

-------Incorrect Solution----------

```
plt.minorticks_on(axis='x')
plt.savefig('output.png', bbox_inches
='tight')
```

-------Correct Solution----------

```
import matplotlib.ticker as ticker
# Turn on minor ticks on x-axis
plt.gca().xaxis.set_minor_locator(ticker.Auto
MinorLocator())
plt.savefig('output.png', bbox_inches
='tight')
```

Fig. 5: A code problem example from DS-1000. The incorrect solution is generated from Self-Repair, and the correct solution is generated from DSrepair. By incorporating knowledge of the invoked API, DSrepair can assist LLMs in generating solutions with correct API usage.

Looking deeper into the buggy code that DSrepair cannot fix, we identify two primary reasons. Firstly, the presence of multiple errors in the code poses a significant challenge. DSrepair is designed to address specific errors highlighted by standard error messages. However, when a code segment contains hidden bugs that come out only after fixing one bug, our approach struggles to resolve all issues in a single request. Secondly, the insufficiency of information provided from the test cases in the description limits the repair effectiveness. Some descriptions lack accompanying test cases, which are crucial for identifying and fixing errors. For instance, if the buggy code triggers an assertion error, the absence of concrete test cases impedes the LLM's ability to generate a precise fix. Even when generated code passes the given test cases, it may still fail during actual evaluation. Simply informing the LLM that the code is incorrect without detailed guidance is often inadequate for effective repair.

Answer to RQ1: DSrepair significantly outperforms all the baselines in fixing buggy data science code. Specifically, DSrepair demonstrates notable improvements across four LLMs by fixing 104, 145, 164, and 140 buggy programs respectively, with improvement rates of 44.4%, 14.2%, 20.6%, and 32.1% compared to the second-best baseline, respectively.

Please note that our comparison with baselines for detecting general software logic bugs is not meant to imply that DSrepair outperforms these baselines in all domains. Rather, we show that approaches not specifically designed for data science

Table 2: RQ1: Effectiveness of DSrepair against the baselines. Values are shown in the format ANF (FR). ANF is the Absolute Number of Fixes. FR is Fix Rate. The results indicate that DSrepair outperforms the baselines for the majority of the libraries.

Model	Approach	Numpy	Pandas	Scipy	Sklearn	Matplotlib	PyTorch	TensorFlow	Total
GPT-3.5-turbo	Code-Search	5 (4.63%)	4 (2.12%)	2 (3.33%)	11 (14.86%)	0 (0.00%)	2 (4.55%)	2 (7.41%)	26 (4.63%)
	Chat-Repair	21 (19.44%)	7 (3.70%)	5 (8.33%)	18 (24.32%)	4 (6.67%)	6 (13.64%)	2 (7.41%)	63 (11.21%)
	Self-Debugging-S	14 (12.96%)	5 (2.65%)	6 (10.00%)	13 (17.57%)	4 (6.67%)	7 (15.91%)	2 (7.41%)	51 (9.07%)
	Self-Debugging-E	20 (18.52%)	**19 (10.05%)**	2 (3.33%)	14 (18.92%)	6 (10.00%)	3 (6.82%)	**4 (14.81%)**	68 (12.10%)
	Self-Repair	17 (15.74%)	17 (8.99%)	5 (8.33%)	12 (16.22%)	8 (13.33%)	9 (20.45%)	**4 (14.81%)**	72 (12.81%)
	DSrepair	**24 (22.22%)**	17 (8.99%)	**15 (25.00%)**	**20 (27.03%)**	**10 (16.67%)**	**15 (34.09%)**	3 (11.11%)	**104 (18.51%)**
GPT-4o-mini	Code-Search	28 (25.93%)	21 (11.11%)	11 (18.33%)	11 (14.86%)	15 (25.00%)	14 (31.82%)	3 (11.11%)	103 (18.33%)
	Chat-Repair	29 (26.85%)	28 (14.81%)	14 (23.33%)	19 (25.68%)	14 (23.33%)	13 (29.55%)	**5 (18.52%)**	122 (21.71%)
	Self-Debugging-S	32 (29.63%)	25 (13.23%)	**16 (26.67%)**	20 (27.03%)	7 (11.67%)	14 (31.82%)	4 (14.81%)	118 (21.00%)
	Self-Debugging-E	**35 (32.41%)**	**33 (17.46%)**	12 (20.00%)	16 (21.62%)	10 (16.67%)	17 (38.64%)	4 (14.81%)	127 (22.60%)
	Self-Repair	34 (31.48%)	32 (16.93%)	13 (21.67%)	15 (20.27%)	11 (18.33%)	15 (34.09%)	**5 (18.52%)**	125 (22.24%)
	DSrepair	33 (30.56%)	20 (10.58%)	15 (25.00%)	**31 (41.89%)**	**22 (36.67%)**	**19 (43.18%)**	**5 (18.52%)**	**145 (25.80%)**
DeepSeek-Coder	Code-Search	23 (21.30%)	11 (5.82%)	2 (3.33%)	12 (16.22%)	17 (28.33%)	8 (18.18%)	4 (14.81%)	77 (13.70%)
	Chat-Repair	39 (36.11%)	22 (11.64%)	8 (13.33%)	18 (24.32%)	20 (33.33%)	14 (31.82%)	7 (25.93%)	128 (22.78%)
	Self-Debugging-S	33 (30.56%)	26 (13.76%)	9 (15.00%)	11 (14.86%)	13 (21.67%)	10 (22.73%)	7 (25.93%)	109 (19.40%)
	Self-Debugging-E	28 (25.93%)	23 (12.17%)	3 (5.00%)	18 (24.32%)	12 (20.00%)	10 (22.73%)	6 (22.22%)	100 (17.79%)
	Self-Repair	**40 (37.04%)**	22 (11.64%)	**12 (20.00%)**	24 (32.43%)	17 (28.33%)	11 (25.00%)	**10 (37.04%)**	136 (24.20%)
	DSrepair	38 (35.19%)	**28 (14.81%)**	10 (16.67%)	**31 (41.89%)**	**23 (38.33%)**	**24 (54.55%)**	**10 (37.04%)**	**164 (29.18%)**
Codestral	Code-Search	27 (25.00%)	13 (6.88%)	9 (15.00%)	24 (32.43%)	19 (31.67%)	8 (18.18%)	**6 (22.22%)**	106 (18.86%)
	Chat-Repair	28 (25.93%)	13 (6.88%)	**12 (20.00%)**	21 (28.38%)	16 (26.67%)	10 (22.73%)	5 (18.52%)	105 (18.68%)
	Self-Debugging-S	27 (25.00%)	19 (10.05%)	7 (11.67%)	13 (17.57%)	8 (13.33%)	9 (20.45%)	2 (7.41%)	85 (15.12%)
	Self-Debugging-E	26 (24.07%)	21 (11.11%)	8 (13.33%)	16 (21.62%)	10 (16.67%)	11 (25.00%)	4 (14.81%)	96 (17.08%)
	Self-Repair	**32 (29.63%)**	17 (8.99%)	6 (10.00%)	14 (18.92%)	10 (16.67%)	12 (27.27%)	5 (18.52%)	96 (17.08%)
	DSrepair	**32 (29.63%)**	**30 (15.87%)**	9 (15.00%)	**28 (37.84%)**	**21 (35.00%)**	**17 (38.64%)**	3 (11.11%)	**140 (24.91%)**

struggle with addressing data science bugs. By addressing the unique needs of data science code, DSrepair can significantly improve repair outcomes in the context of data science.

B. RQ2: Overlap with Baseline

In this RQ, we conduct an overlap analysis by comparing the solved buggy code snippets between DSrepair and the baselines. Fig 6 shows the upset plots [53] for different approaches and the intersection of their ANF.

We can observe that the fixed buggy code overlaps between DSrepair and baselines are overall less than 55% of the bug fixes from DSrepair. This means that about half of the code fixes from DSrepair could not be fixed by baselines. For example, in Fig 6(a), DSrepair can fix 104 buggy code snippets, while Self-Repair can only fix 72 buggy code snippets. The overlap between their fixed code snippets is only 34, which means that DSrepair has 70 (104-34=70) code snippets that Self-Repair cannot fix, and Self-Repair has 38 (72-34=38) code snippets that DSrepair cannot fix. The overlap among the six baselines is quite low (5 for GPT-3.5-turbo, 36 for GPT-4o-mini, 21 for DeepSeek-Coder, and 14 for Codestral). Overall, each baseline shows the uniqueness of buggy code repair.

Answer to RQ2: DSrepair uniquely fixes approximately 55% of buggy code snippets that baselines are unable to fix.

C. RQ3: Cost of DSrepair

To answer RQ3, we assess the financial costs associated with using DSrepair by quantifying the US dollar spent on interactions with using the APIs of the four LLMs. The cost of each request to these models depends directly on the number of tokens processed, including both the tokens used for input and those generated as output. We calculate the expenses incurred during these interactions by measuring the Token Usage (TU) of DSrepair and then converting this usage into actual Money Spent (MS), comparing these against the cost of our baselines.

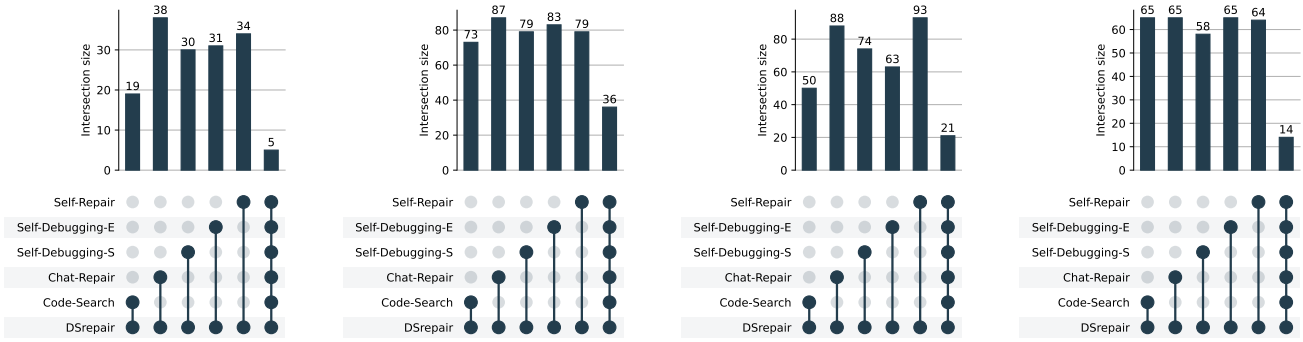
Table 3 shows the TU and MS for different approaches. Fig 7 shows a scatter plot of TU and FR. We observe that DSrepair costs less token usage than the second-best baseline. For example, DSrepair uses only 1262.14, 1584.74, 1453.96, and 1407.15 tokens per code problem, while Self-Repair needs 1529.63, 1944.56, 1931.20, and 1657.99 tokens per code problem. Based on the real-time price in Table 1, the money spent on each request is \$0.00073, \$0.00043, \$0.00025, and \$0.00185 for using GPT-3.5-turbo, GPT-4o-mini, DeepSeek-Coder, and Codestral as LLM respectively.

We can also observe that DSrepair's token usage with GPT-4o-mini, DeepSeek-Coder, and Codestral is higher than when using GPT-3.5-turbo. This is because the return of these code LLMs may not follow the prompt's output format instruction. The responses typically contain more information, such as line-by-line code comments, natural language explanation of the code, and an analysis of why the first generated code is incorrect, all of which contribute to extra costs.

Answer to RQ3: Compared to the second-best baseline, DSrepair uses fewer tokens (1262.14, 1584.74, 1453.96, and 1407.15), saving 17.49%, 34.24%, 24.71%, and 17.59% tokens per code task respectively across different LLMs.

D. RQ4: Influence of Prompt Design

To figure out how different prompt components influence DSrepair, we conduct an ablation study. In DSrepair, there are two key components, i.e., API knowledge and bug knowledge. In the ablation study, we compare DSrepair's performance



(a) GPT-3.5-turbo (b) GPT-4o-mini (c) DeepSeek-Coder (d) Codestral

Fig. 6: RQ2: Upset plots for overlap analysis. For example, in (a), the first column indicates that 19 buggy code snippets that can be fixed by both DSrepair and Code-Search.

Table 3: RQ3: Cost of different approaches. TU refers to Token Usage (input token usage + output token usage), and MS refers to Money Spent for LLM receiving the prompt and generating the response.

Approach	GPT-3.5-turbo		GPT-4o-mini		DeepSeek-Coder		Codestral	
	TU	MS	TU	MS	TU	MS	TU	MS
Code-Search	1829.66	$0.00124	1697.94	$0.00034	1804.85	$0.00030	1707.45	$0.00218
Chat-Repair	736.22	$0.00050	788.97	$0.00022	696.66	$0.00012	784.52	$0.00121
Self-Debugging-S	**546.13**	**$0.00041**	**605.48**	**$0.00018**	**585.90**	**$0.00011**	**634.84**	**$0.00108**
Self-Debugging-E	1695.55	$0.00120	2410.03	$0.00070	1738.98	$0.0003	1763.98	$0.00260
Self-Repair	1529.63	$0.00104	1944.56	$0.00053	1931.20	$0.00034	1657.99	$0.00232
DSrepair	1262.14	$0.00073	1584.74	$0.00043	1453.96	$0.00025	1407.15	$0.00185

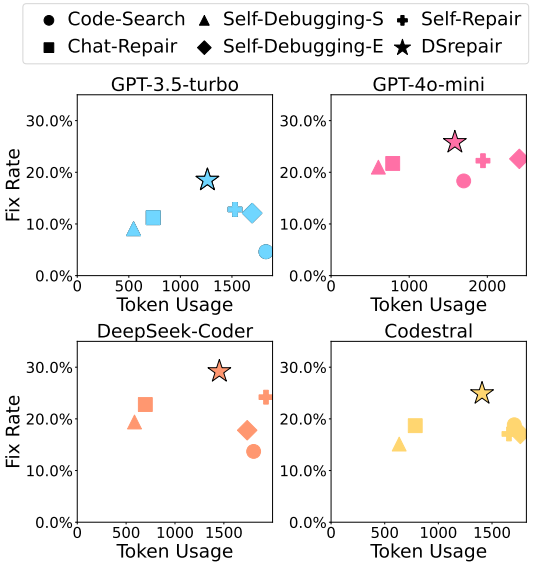


Fig. 7: RQ3: Scatter plot of TU (Token Usage) and FR (Fix Rate). DSrepair is the optimal approach (the star markers) compared with baselines.

with the performance of 'No knowledge' (prompt without API and bug knowledge), 'API Knowledge only' (prompt without bug knowledge provided by tests), and 'Bug Knowledge only' (prompt without API knowledge) We use 'DSrepair w/o API&Bug', 'DSrepair w/o Bug', and 'DSrepair w/o API' to represent 'No knowledge', 'API Knowledge only', and 'Bug Knowledge only' for short.

The results of the ablation study are shown in Table 4. When using GPT-3.5-turbo as LLM, the overall performance of DSrepair (18.51% FR) is better than DSrepair w/o Bug (14.77% FR) and DSrepair w/o API (16.73% FR). The overall performance for GPT-4o-mini of DSrepair (25.80% FR) is better than DSrepair w/o Bug (23.49% FR) and DSrepair w/o API (22.06% FR). When using DeepSeek-Coder as our LLM, DSrepair still stands for the best, with 29.18% total FR. However, DSrepair w/o Bug has better overall performance than DSrepair w/o API, where using DSrepair w/o Bug has 28.83% FR while using DSrepair w/o API only has 27.94% FR. Using Codestral as LLM, DSrepair has 24.91% FR, which is higher than both DSrepair w/o Bug (24.51%) and DSrepair w/o API (24.38%). Interestingly, we observe that the FR declines in DSrepair w/o Bug (GPT-3.5-turbo), DSrepair w/o Bug and w/o API (GPT-4o-mini) and DSrepair w/o API (DeepSeek-Coder) compared with DSrepair w/o API&Bug.

Answer to RQ4: Both enriched API knowledge and enriched bug knowledge in the prompt contribute to the final effectiveness of DSrepair.

E. RQ5: Comparison of Different Knowledge Retrieval Approaches

To answer RQ5, we examine the impact of various knowledge retrieval approaches on the performance of DSrepair. Specifically, we compare knowledge retrieval through KG (DSrepair) with knowledge retrieval through plain-text searching. For plain-text searching, we extract API knowledge using

Table 4: RQ4: Results of ablation study. 'DSrepair w/o API&Bug' is for prompt without API and bug knowledge, 'DSrepair w/o Bug' is for prompt without bug knowledge, and 'DSrepair w/o API' refers to prompt without API knowledge. ANF is the Absolute Number of Fixes. FR is Fix Rate.

Model	Prompt	ANF	FR
GPT-3.5-turbo	DSrepair w/o API	94	16.73%
	DSrepair w/o Bug	83	14.77%
	DSrepair w/o API&Bug	85	15.12%
	DSrepair	**104**	**18.51%**
GPT-4o-mini	DSrepair w/o API	124	22.06%
	DSrepair w/o Bug	132	23.49%
	DSrepair w/o API&Bug	133	23.67%
	DSrepair	**145**	**25.80%**
DeepSeek-Coder	DSrepair w/o API	157	27.94%
	DSrepair w/o Bug	162	28.83%
	DSrepair w/o API&Bug	160	28.47%
	DSrepair	**164**	**29.18%**
Codestral	DSrepair w/o API	137	24.38%
	DSrepair w/o Bug	139	24.51%
	DSrepair w/o API&Bug	123	21.89%
	DSrepair	**140**	**24.91%**

invoked API names as keywords. The API knowledge is retrieved as a window of text encompassing 50 tokens per keyword. This window length was chosen to match the average size of the retrieval results from DS-KG for each keyword, ensuring a fair comparison. All other experimental settings are kept consistent with those used in DSrepair.

Table 5 shows the results of different knowledge retrieval approaches for DSrepair. We can see that retrieving knowledge from plain text only has 14.77%, 22.60%, 25.44%, and 22.78% Fix Rate for four tested LLMs. Retrieving knowledge from plain text uses 1432.42, 1827.59, 1723.77, and 1636.46 tokens per buggy code, which is higher than retrieval from DS-KG, and thus has higher Money Spent on four LLMs.

Answer to RQ5: Knowledge graph-based retrieval outperforms plain text-based retrieval in fixing buggy data science code. The former's fix rate is 18.51%, 25.80%, 29.18%, and 24.91% for GPT-3.5-turbo, GPT-4o-mini, DeepSeek-Coder, and Codestral, respectively, compared to 14.77%, 22.60%, 25.44%, and 22.78% for the latter.

F. RQ6: Influence of API Richness

To address RQ6, we assess how varying the richness of API knowledge impacts the performance of DSrepair. In our DSrepair setup, we use only the full expressions of the invoked API to enrich the prompts. Our queries also yield additional information about correct API usage, including explanations of functions, and details about parameters and returns. To explore the potential benefits of this enriched API knowledge, we design experiments with different richness levels of API information: DSrepair+explanation, DSrepair+parameter&return, and DSrepair+explanation+parameter&return. DSrepair+explanation incorporates explanations of the invoked API into the API knowledge. DSrepair+parameter&return adds information about the function's parameters and returns. DSrepair+explanation+parameter&return combines both types of information into the API knowledge.

Table 5: RQ5: Knowledge retrieval comparison between plain text and KG. Retrieval from KG is better than from plain text. FR refers to Fix Rate, TU refers to Token Usage (input token usage + output token usage), and MS refers to Money Spent for LLM receiving the prompt and generating the response.

Model	Knowledge Retrieval	FR	TU	MS
GPT-3.5-turbo	Plain Text	14.77%	1432.42	$0.00082
	Knowledge Graph	**18.51%**	**1262.14**	**$0.00073**
GPT-4o-mini	Plain Text	22.60%	1827.59	$0.00046
	Knowledge Graph	**25.80%**	**1584.74**	**$0.00043**
DeepSeek-Coder	Plain Text	25.44%	1723.77	$0.00030
	Knowledge Graph	**29.18%**	**1453.96**	**$0.00025**
Codestral	Plain Text	22.78%	1636.46	$0.00209
	Knowledge Graph	**24.91%**	**1407.15**	**$0.00185**

Table 6 presents the performance results of these different levels of API knowledge richness. The results are evaluated in terms of effectiveness, as measured by the Fix Rate (FR), and cost, as quantified by Token Usage (TU) and Money Spent (MS). The data shows that DSrepair achieves the highest Fix Rate across all richness levels, with 18.51% on GPT-3.5-turbo, 25.80% on GPT-4o-mini, 29.18% on DeepSeek-Coder, and 24.91% on Codestral. This suggests that the additional information may complicate the prompt without necessarily improving the effectiveness of the repair.

In terms of cost, DSrepair generally exhibits lower token usage and monetary cost compared to its enriched counterparts. For example, on GPT-3.5-turbo, DSrepair uses 1262.14 tokens and incurs a cost of $0.00073, whereas DSrepair+explanation+parameter&return uses 1584.49 tokens and costs $0.00089. This pattern holds across the other models as well, indicating that increasing the complexity of the API knowledge may lead to higher money costs without a proportional gain in repair effectiveness.

Answer to RQ6: Using full expressions of invoked API from the retrieval results in DSrepair performs the best in fixing bugs.

G. RQ7: Influence of LLM Non-determinism

To investigate RQ7, we explore the effect of non-determinism in LLMs on our experimental results. As outlined in Section III, we conduct each experiment ten times to account for variability in LLM responses. From these iterations, we select the median performance result as our final data point for analysis. To further understand how LLM non-determinism might influence our results, we calculate the mean and standard deviation of the results across the ten trials.

Table 7 shows the mean FR (Fix Rate) and the standard deviation of code repair results. We observe that the standard

Table 6: RQ6: Influence of API knowledge richness on DSrepair. FR refers to Fix Rate, TU refers to Token Usage (input token usage + output token usage), and MS refers to Money Spent for LLM receiving the prompt and generating the response.

API Knowledge Richness	GPT-3.5-turbo			GPT-4o-mini			DeepSeek-Coder			Codestral		
	FR	TU	MS	FR	TU	MS	FR	TU	MS	FR	TU	MS
DSrepair+explanation	15.84%	1279.50	$0.00074	22.78%	1597.57	$0.00043	27.22%	1503.78	$0.00026	24.38%	1410.30	$0.00187
DSrepair+parameter&return	14.95%	1573.98	$0.00089	22.60%	1885.12	$0.00047	26.33%	1803.51	$0.00030	24.02%	1698.39	$0.00215
DSrepair+explanation+parameter&return	15.30%	1584.49	$0.00089	23.49%	1901.94	$0.00047	26.87%	1816.90	$0.00030	24.02%	1712.37	$0.00217
DSrepair	**18.51%**	**1262.14**	**$0.00073**	**25.80%**	**1584.74**	**$0.00043**	**29.18%**	**1453.96**	**$0.00025**	**24.91%**	**1407.15**	**$0.00185**

deviations of DSrepair are not big, which indicates the stability of our experiment results. Additionally, with the standard deviation, DSrepair still outperforms the baselines in its mean FR, with 101.80 $\pm$ 6.71, 142.90 $\pm$ 6.44, 163.67 $\pm$ 1.25, 137.80 $\pm$ 4.60 for using GPT-3.5-turbo, GPT-4o-mini, DeepSeek-Coder, and Codestral as LLM respectively.

Answer to RQ7: Despite the randomness of LLMs, DSrepair consistently outperforms the baselines with greater stability across multiple trials. It achieves mean Fix Rates of 101.80 $\pm$ 6.71, 142.90 $\pm$ 6.44, 163.67 $\pm$ 1.25, 137.80 $\pm$ 4.60 across GPT-3.5-turbo, GPT-4o-mini, DeepSeek-Coder, and Codestral respectively.

V. Discussion

In this section, we discuss the threats to validity, the limitations, and the generalizability of our research.

A. Threats to Validity

The threats to *internal* validity mainly lie in the implementation of our prompt design. To reduce this threat, we design DSrepair with the idea of 'Structuring Prompts', adapted from a handy template for structuring prompts, called CO-STAR framework [54]. Considering key aspects that influence the effectiveness and relevance of an LLM's response, DSrepair can lead LLMs to generate more optimal responses for code purposes. In addition, we design research questions, such as RQ4 and RQ6, to study the influence of the different prompts on our final performance.

The threats to *external* validity mainly lie in the datasets, and LLMs used in our study. To reduce the threat regarding datasets, we carefully choose to use DS-1000 as our experiment dataset, which is the state-of-the-art benchmark tailored to address data leakage concerns with realistic and diverse data science problems, with testing methods checking both execution semantics and surface-form constraints [5]. To reduce the threat regarding LLMs, we use four widely studied LLMs to mitigate the potential bias that certain LLMs can bring to the experiment results. In addition, to mitigate the inherent randomness of LLMs, we experiment ten times for each approach and choose one with the median overall performance as our final result, which could further mitigate the non-determinism of LLMs. Moreover, we exclusively design RQ7 to study whether the non-determinism of LLMs will affect our experiment findings.

B. Limitation

The effectiveness of DSrepair depends largely on the quality and completeness of the knowledge it provides. Our approach demonstrates capability in addressing runtime errors by eliminating the initial error. However, this repair process can sometimes introduce or trigger new errors. This phenomenon is particularly evident when the repaired code successfully executes but subsequently results in assertion errors. The reliance on high-quality test cases in the problem description is crucial; in their absence, DSrepair may guide LLMs to generate code that closely mirrors the incorrect code. This occurs because the LLMs are provided with API knowledge that can inadvertently reinforce the use of incorrect or irrelevant APIs present in the original code. Despite this, there are instances where, upon receiving API knowledge, the LLMs deviate from the incorrect APIs, opting instead for alternative solutions, such as using different APIs or defining new functions.

Maintaining the DS-KG presents significant challenges. Our DS-KG only reflects the correct knowledge of APIs based on a specific version. The rapid pace at which online API documentation is updated complicates the task of ensuring the DS-KG remains up-to-date. Consequently, keeping the DS-KG up-to-date demands substantial effort and resources. This maintenance burden is a critical consideration, as outdated or incomplete knowledge can adversely affect the accuracy and reliability of the repairs generated by DSrepair. With the assistance of API document's release notes, we could manage the updating of DS-KG by leveraging library development logs to automate the process. These logs often document changes and updates made to API libraries, allowing us to efficiently identify and integrate the necessary modifications into the KG.

Another notable limitation of DSrepair is the time cost associated with knowledge retrieval. When compared to plain text searching, retrieval using the DS-KG incurs a significant time overhead, averaging 51.49% more time (approximately 0.06 seconds) per task. While this increase in retrieval time may seem marginal, it can accumulate and impact the overall efficiency of the repair process, particularly in scenarios requiring rapid iteration and testing.

C. Generalizability of DSrepair

In this paper, DSrepair is specifically designed to enhance the repair of data science code. Nevertheless, DSrepair's underlying methodology—leveraging knowledge-enhanced retrieval and structured bug information—can be generalized to broader coding tasks. The key innovation of DSrepair lies

Table 7: RQ7: Mean and standard deviation of the ANF (Absolute Number of Fix), expressed as mean ANF ± standard deviation. The small standard deviation indicates the reliability of our results, and our experimental conclusions in RQ1 remain valid within this range.

Approach	GPT-3.5-turbo	GPT-4o-mini	DeepSeek-Coder	Codestral
Code-Search	26.40 ± 2.46	102.5 ± 2.29	75.33 ± 3.09	104.3 ± 2.83
Chat-Repair	60.50 ± 3.75	122.0 ± 2.79	128.00 ± 0.82	104.8 ± 2.18
Self-Debugging-S	51.10 ± 4.44	117.4 ± 2.69	108.00 ± 2.16	81.70 ± 6.39
Self-Debugging-E	68.60 ± 6.18	124.2 ± 6.16	102.67 ± 3.77	94.60 ± 5.70
Self-Repair	71.20 ± 4.58	125.1 ± 3.59	134.33 ± 3.09	91.50 ± 8.42
DSrepair	**101.80 ± 6.71**	**142.90 ± 6.44**	**163.67 ± 1.25**	**137.80 ± 4.60**

in its use of knowledge graph-based Retrieval-Augmented Generation (RAG), which is not inherently limited to data science APIs. By replacing the domain-specific DS-KG with a knowledge graph covering general-purpose programming languages and software libraries, DSrepair could be adapted to repair a wide range of buggy code across different domains, as well as to improve other coding tasks other than repair, such as bug localization and code generation.

DSrepair also has the potential to be extended to support project-level code generation and repair, where understanding dependencies across multiple files and context knowledge with the same project are crucial. By constructing a project-specific KG that encodes function definitions, module dependencies, and code architecture, DSrepair can enhance code generation and repair in large-scale software development.

VI. Related Work

A. Code Repair

The goal of automated program repair is to automatically identify and fix bugs or defects in the software. Leveraging LLMs, such as BERT [55], CodeBERT [56], Codex [57]–[59], and GPT-series [9], [60]–[62], for code repair can achieve promising performance in generating patches for various kinds of bugs and defects. These models are adept at grasping the core meaning and relationships within code, resulting in the generation of precise and functional fixes without the need for compilation. Using LLMs for fixing code speeds up the identification and resolution of bugs, freeing software developers to tackle more intricate issues. This contributes to improved software reliability and upkeep. ChatGPT, in particular, stands out among LLMs because of its built-in interactive nature, which fosters an ongoing loop of feedback, producing patches that are more polished and appropriate to the context [9], [61].

B. Prompt Engineering

Prompt designing is an increasingly important skill set needed to leverage effectively with LLMs [63], such as ChatGPT. Similar to software design [64], the design of prompt aims at offering reusable solutions to specific problems, by providing a codified approach to customizing the output and interactions of LLMs. Abukhalaf et al. [65] conduct an empirical study on Object Constraint Language based constraint generation, by comparing the Codex generated constraints and humane-written constraints. Xia et al. [61] specifically examined prompts for automatic code repair. More specifically, White et al. [66] focus on combatting mistakes and improving generated code quality by designing prompt patterns. Borji et al. [67] examine the quality of generated answers and code from LLMs, and conclude the existing failures from the experiment. Our research work draws inspiration from these explorations and prompts that could be used to generate code candidates with better quality and fewer errors.

C. Retrieval-Augmented Generation

RAG aims to address the limitations of generative models, including issues related to outdated knowledge, a deficiency in long-tail knowledge [68], and the potential for private training data leakage [69]. Early research in code generation concentrated on code-to-code retrieval using dual encoder models, with the retrieved outputs subsequently inputted into autoregressive language models [20]. RepoCoder [22] enhances retrieval processes by employing iterative incremental generations [70]. KNM [21] leverages in-domain code databases and applies Bayesian inference to finalize the generated code. RAG also can be used to build prompts for transformer-based generative models with retrieved information, including similar examples [15], [16], relevant API details [6], [7], documentations [17], and imports [18].

VII. Conclusion

We propose DSrepair, a novel knowledge-enhanced approach for data science code repair. We perform experiments with four LLMs and five baselines in data science code repair and find that DSrepair significantly outperforms all the baselines in repairing data science code. By integrating API knowledge retrieval and bug information enrichment, we can guarantee better performance in code repair, and gain people's trust in using LLMs for coding. In future work, we also plan to explore a multi-agent framework with interactive feedback to enhance the performance of DSrepair while focusing on optimizing feedback steps and resource use to ensure scalability, cost efficiency and robust data science code repair.

VIII. Acknowledgement

This work was supported by the UKRI Centre for Doctoral Training in Safe and Trusted Artificial Intelligence (EP/S023356/1), the NSFC (62192732), and the National Natural Science Foundation of China (62402482).

REFERENCES

[1] E. Bolyen, J. R. Rideout, M. R. Dillon, N. A. Bokulich, C. C. Abnet, G. A. Al-Ghalith, H. Alexander, E. J. Alm, M. Arumugam, F. Asnicar *et al.*, "Reproducible, interactive, scalable and extensible microbiome data science using qiime 2," *Nature biotechnology*, vol. 37, no. 8, pp. 852–857, 2019.

[2] H. Hassani and E. S. Silva, "The role of chatgpt in data science: how ai-assisted conversational interfaces are revolutionizing the field," *Big data and cognitive computing*, vol. 7, no. 2, p. 62, 2023.

[3] S. Hong, Y. Lin, B. Liu, B. Wu, D. Li, J. Chen, J. Zhang, J. Wang, L. Zhang, M. Zhuge *et al.*, "Data interpreter: An llm agent for data science," *arXiv preprint arXiv:2402.18679*, 2024.

[4] M. Nejjar, L. Zacharias, F. Stiehle, and I. Weber, "Llms for science: Usage for code generation and data analysis," *arXiv preprint arXiv:2311.16733*, 2023.

[5] Y. Lai, C. Li, Y. Wang, T. Zhang, R. Zhong, L. Zettlemoyer, W.-t. Yih, D. Fried, S. Wang, and T. Yu, "Ds-1000: A natural and reliable benchmark for data science code generation," in *International Conference on Machine Learning*. PMLR, 2023, pp. 18 319–18 345.

[6] D. Zan, B. Chen, Y. Gong, J. Cao, F. Zhang, B. Wu, B. Guan, Y. Yin, and Y. Wang, "Private-library-oriented code generation with large language models," *arXiv preprint arXiv:2307.15370*, 2023.

[7] D. Zan, B. Chen, Z. Lin, B. Guan, Y. Wang, and J.-G. Lou, "When language model meets private library," *arXiv preprint arXiv:2210.17236*, 2022.

[8] Y. Ge, W. Hua, K. Mei, J. Tan, S. Xu, Z. Li, Y. Zhang *et al.*, "Openagi: When llm meets domain experts," *Advances in Neural Information Processing Systems*, vol. 36, 2024.

[9] C. S. Xia and L. Zhang, "Keep the conversation going: Fixing 162 out of 337 bugs for $0.42 each using chatgpt," *arXiv preprint arXiv:2304.00385*, 2023.

[10] K. Gupta, P. E. Christensen, X. Chen, and D. Song, "Synthesize, execute and debug: Learning to repair for neural program synthesis," *Advances in Neural Information Processing Systems*, vol. 33, pp. 17 685–17 695, 2020.

[11] X. Chen, M. Lin, N. Schärli, and D. Zhou, "Teaching large language models to self-debug," *arXiv preprint arXiv:2304.05128*, 2023.

[12] M. Fu, C. K. Tantithamthavorn, V. Nguyen, and T. Le, "Chatgpt for vulnerability detection, classification, and repair: How far are we?" in *2023 30th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2023, pp. 632–636.

[13] Q. Zhang, T. Zhang, J. Zhai, C. Fang, B. Yu, W. Sun, and Z. Chen, "A critical review of large language model on software engineering: An example from chatgpt and automated program repair," *arXiv preprint arXiv:2310.08879*, 2023.

[14] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel *et al.*, "Retrieval-augmented generation for knowledge-intensive nlp tasks," *Advances in Neural Information Processing Systems*, vol. 33, pp. 9459–9474, 2020.

[15] M. R. Parvez, W. U. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "Retrieval augmented code generation and summarization," *arXiv preprint arXiv:2108.11601*, 2021.

[16] J. Li, Y. Zhao, Y. Li, G. Li, and Z. Jin, "Acecoder: Utilizing existing code to enhance code generation," *arXiv preprint arXiv:2303.17780*, 2023.

[17] S. Zhou, U. Alon, F. F. Xu, Z. Wang, Z. Jiang, and G. Neubig, "Docprompting: Generating code by retrieving the docs," *arXiv preprint arXiv:2207.05987*, 2022.

[18] M. Liu, T. Yang, Y. Lou, X. Du, Y. Wang, and X. Peng, "Codegen4libs: A two-stage approach for library-oriented code generation," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023, pp. 434–445.

[19] T. Ahmed, K. S. Pai, P. Devanbu, and E. Barr, "Automatic semantic augmentation of language model prompts (for code summarization)," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.

[20] S. Lu, N. Duan, H. Han, D. Guo, S.-w. Hwang, and A. Svyatkovskiy, "Reacc: A retrieval-augmented code completion framework," *arXiv preprint arXiv:2203.07722*, 2022.

[21] Z. Tang, J. Ge, S. Liu, T. Zhu, T. Xu, L. Huang, and B. Luo, "Domain adaptive code completion via language models and decoupled domain databases," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023, pp. 421–433.

[22] F. Zhang, B. Chen, Y. Zhang, J. Keung, J. Liu, D. Zan, Y. Mao, J.-G. Lou, and W. Chen, "Repocoder: Repository-level code completion through iterative retrieval and generation," *arXiv preprint arXiv:2303.12570*, 2023.

[23] P. Yin, B. Deng, E. Chen, B. Vasilescu, and G. Neubig, "Learning to mine aligned code and natural language pairs from stack overflow," in *Proceedings of the 15th international conference on mining software repositories*, 2018, pp. 476–486.

[24] P. Yin, W.-D. Li, K. Xiao, A. Rao, Y. Wen, K. Shi, J. Howland, P. Bailey, M. Catasta, H. Michalewski *et al.*, "Natural language to code generation in interactive data science notebooks," *arXiv preprint arXiv:2212.09248*, 2022.

[25] https://platform.openai.com/docs/models.

[26] D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, W. Zhang, G. Chen, X. Bi, Y. Wu, Y. Li *et al.*, "Deepseek-coder: When the large language model meets programming–the rise of code intelligence," *arXiv preprint arXiv:2401.14196*, 2024.

[27] https://mistral.ai/news/codestral/.

[28] https://github.com/ShuyinOuyang/DSrepair.

[29] E. Simperl and M. Luczak-Rösch, "Collaborative ontology engineering: a survey," *The Knowledge Engineering Review*, vol. 29, no. 1, pp. 101–131, 2014.

[30] M. C. Suárez-Figueroa, A. Gómez-Pérez, and M. Fernández-López, "The neon methodology for ontology engineering," in *Ontology engineering in a networked world*. Springer, 2011, pp. 9–34.

[31] Q. Liang, Z. Kuai, Y. Zhang, Z. Zhang, L. Kuang, and L. Zhang, "Misusehint: A service for api misuse detection based on building knowledge graph from documentation and codebase," in *2022 IEEE International Conference on Web Services (ICWS)*. IEEE, 2022, pp. 246–255.

[32] I. Abdelaziz, J. Dolby, J. McCusker, and K. Srinivas, "A toolkit for generating code knowledge graphs," in *Proceedings of the 11th Knowledge Capture Conference*, 2021, pp. 137–144.

[33] E. Prudhommeaux, "Sparql query language for rdf," *http://www. w3. org/TR/rdf-sparql-query/*, 2008.

[34] P. Blinov, "Semantic triples verbalization with generative pre-training model," in *Proceedings of the 3rd International Workshop on Natural Language Generation from the Semantic Web (WebNLG+)*, 2020, pp. 154–158.

[35] R. Abreu, P. Zoeteweij, and A. J. Van Gemund, "On the accuracy of spectrum-based fault localization," in *Testing: Academic and industrial conference practice and research techniques-MUTATION (TAICPART-MUTATION 2007)*. IEEE, 2007, pp. 89–98.

[36] M. Papadakis and Y. Le Traon, "Metallaxis-fl: mutation-based fault localization," *Software Testing, Verification and Reliability*, vol. 25, no. 5-7, pp. 605–628, 2015.

[37] R. Brate, M.-H. Dang, F. Hoppe, Y. He, A. Meroño-Peñuela, and V. Sadashivaiah, "Improving language model predictions via prompts enriched with knowledge graphs," in *DL4KG@ ISWC2022*, 2022.

[38] R. Agashe, S. Iyer, and L. Zettlemoyer, "Juice: A large scale distantly supervised dataset for open domain context-based code generation," *arXiv preprint arXiv:1910.02216*, 2019.

[39] https://huggingface.co/spaces/evaluate-metric/exact_match.

[40] M. Post, "A call for clarity in reporting bleu scores," *arXiv preprint arXiv:1804.08771*, 2018.

[41] https://numpy.org/doc/stable/index.html.

[42] https://pandas.pydata.org/docs/index.html.

[43] https://docs.scipy.org/doc/scipy/index.html.

[44] https://scikit-learn.org/stable/.

[45] https://matplotlib.org/.

[46] https://PyTorch.org/.

[47] https://www.tensorflow.org/.

[48] J. Chen, X. Hu, Z. Li, C. Gao, X. Xia, and D. Lo, "Code search is all you need? improving code suggestions with code search," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.

[49] https://lucene.apache.org/.

[50] M. Bahrami, N. Shrikanth, S. Ruangwan, L. Liu, Y. Mizobuchi, M. Fukuyori, W.-P. Chen, K. Munakata, and T. Menzies, "Pytorrent: A python library corpus for large-scale language models," *arXiv preprint arXiv:2110.01710*, 2021.

[51] T. X. Olausson, J. P. Inala, C. Wang, J. Gao, and A. Solar-Lezama, "Demystifying gpt self-repair for code generation," *arXiv preprint arXiv:2306.09896*, 2023.

[52] S. Ouyang, J. M. Zhang, M. Harman, and M. Wang, "Llm is like a box of chocolates: the non-determinism of chatgpt in code generation," *arXiv preprint arXiv:2308.02828*, 2023.
[53] https://en.wikipedia.org/wiki/UpSet_plot.
[54] https://towardsdatascience.com/how-i-won-singapores-gpt-4-prompt-engineering-competition-34c195a93d41.
[55] Q. Zhang, C. Fang, W. Sun, Y. Liu, T. He, X. Hao, and Z. Chen, "Appt: Boosting automated patch correctness prediction via fine-tuning pre-trained models," *IEEE Transactions on Software Engineering*, 2024.
[56] T. Le-Cong, D.-M. Luong, X. B. D. Le, D. Lo, N.-H. Tran, B. Quang-Huy, and Q.-T. Huynh, "Invalidator: Automated patch correctness assessment via semantic and syntactic reasoning," *IEEE Transactions on Software Engineering*, 2023.
[57] Z. Fan, X. Gao, M. Mirchev, A. Roychoudhury, and S. H. Tan, "Automated repair of programs from large language models," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 1469–1481.
[58] M. Jin, S. Shahriar, M. Tufano, X. Shi, S. Lu, N. Sundaresan, and A. Svyatkovskiy, "Inferfix: End-to-end program repair with llms," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 1646–1656.
[59] Y. Wu, N. Jiang, H. V. Pham, T. Lutellier, J. Davis, L. Tan, P. Babkin, and S. Shah, "How effective are neural networks for fixing security vulnerabilities," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023, pp. 1282–1294.
[60] M. Lajkó, V. Csuvik, and L. Vidács, "Towards javascript program repair with generative pre-trained transformer (gpt-2)," in *Proceedings of the Third International Workshop on Automated Program Repair*, 2022, pp. 61–68.
[61] C. S. Xia and L. Zhang, "Conversational automated program repair," *arXiv preprint arXiv:2301.13246*, 2023.
[62] D. Sobania, M. Briesch, C. Hanna, and J. Petke, "An analysis of the automatic bug fixing performance of chatgpt," in *2023 IEEE/ACM International Workshop on Automated Program Repair (APR)*. IEEE, 2023, pp. 23–30.
[63] J. White, Q. Fu, S. Hays, M. Sandborn, C. Olea, H. Gilbert, A. Elnashar, J. Spencer-Smith, and D. C. Schmidt, "A prompt pattern catalog to enhance prompt engineering with chatgpt," *arXiv preprint arXiv:2302.11382*, 2023.
[64] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. Pearson Deutschland GmbH, 1995.
[65] S. Abukhalaf, M. Hamdaqa, and F. Khomh, "On codex prompt engineering for ocl generation: An empirical study," *arXiv preprint arXiv:2303.16244*, 2023.
[66] J. White, S. Hays, Q. Fu, J. Spencer-Smith, and D. C. Schmidt, "Chatgpt prompt patterns for improving code quality, refactoring, requirements elicitation, and software design," *arXiv preprint arXiv:2303.07839*, 2023.
[67] A. Borji, "A categorical archive of chatgpt failures," *arXiv preprint arXiv:2302.03494*, 2023.
[68] A. Mallen, A. Asai, V. Zhong, R. Das, D. Khashabi, and H. Hajishirzi, "When not to trust language models: Investigating effectiveness of parametric and non-parametric memories," *arXiv preprint arXiv:2212.10511*, 2022.
[69] N. Carlini, F. Tramer, E. Wallace, M. Jagielski, A. Herbert-Voss, K. Lee, A. Roberts, T. Brown, D. Song, U. Erlingsson *et al.*, "Extracting training data from large language models," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 2633–2650.
[70] W. Chu, L. Li, L. Reyzin, and R. Schapire, "Contextual bandits with linear payoff functions," in *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*. JMLR Workshop and Conference Proceedings, 2011, pp. 208–214.