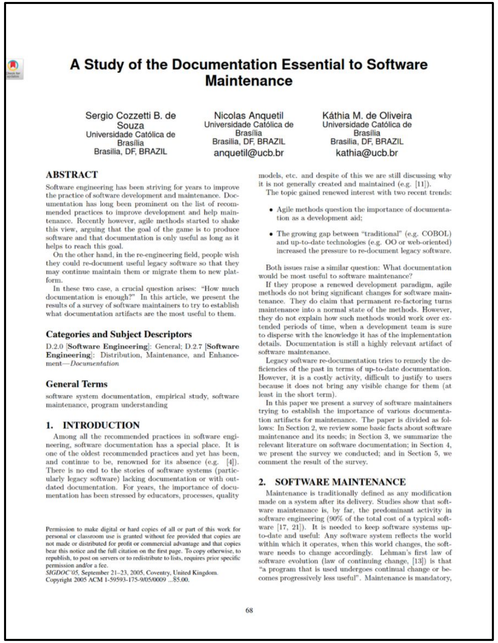
Optimizing Datasets for Code Summarization: Is Code-Comment Coherence Enough?

Antonio Vitale, Antonio Mastropaolo,
Rocco Oliveto, Massimiliano Di Penta, Simone Scalabrino

ICPC 2025
The 33rd IEEE/ACM International Conference on Program Comprehension
Ottawa, ON, Canada – April 27 – 28, 2025

BACKGROUND



A Study of the Documentation Essential to Software Maintenance

Sergio Cozzetti B. de Souza
Universidade Católica de Brasília
Brasilia, DF, BRAZIL

Nicolas Anquetil
Universidade Católica de Brasília
Brasilia, DF, BRAZIL
anquetil@ucb.br

Káthia M. de Oliveira
Universidade Católica de Brasília
Brasilia, DF, BRAZIL
kathia@ucb.br

ABSTRACT

Software engineering has been striving for years to improve the practice of software development and maintenance. Documentation has long been prominent on the list of recommended practices to improve development and help maintenance. Recently however, agile methods started to shake this view, arguing that the goal of the game is to produce software and that documentation is only useful as long as it helps to reach this goal.

On the other hand, in the re-engineering field, people wish they could re-document useful legacy software so that they may continue maintain them or migrate them to new platform.

In these two case, a crucial question arises: "How much documentation is enough?" In this article, we present the results of a survey of software maintainers to try to establish what documentation artifacts are the most useful to them.

Categories and Subject Descriptors

D.2.0 [**Software Engineering**]: General; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Documentation*

General Terms

software system documentation, empirical study, software maintenance, program understanding

1. INTRODUCTION

Among all the recommended practices in software engineering, software documentation has a special place. It is one of the oldest recommended practices and yet has been, and continue to be, renowned for its absence (e.g. [4]). There is no end to the stories of software systems (particularly legacy software) lacking documentation or with outdated documentation. For years, the importance of documentation has been stressed by educators, processes, quality models, etc. and despite of this we are still discussing why it is not generally created and maintained (e.g. [11]).

The topic gained renewed interest with two recent trends:

- Agile methods question the importance of documentation as a development aid;
- The growing gap between "traditional" (e.g. COBOL) and up-to-date technologies (e.g. OO or web-oriented) increased the pressure to re-document legacy software.

Both issues raise a similar question: What documentation would be most useful to software maintenance?

If they propose a renewed development paradigm, agile methods do not bring significant changes for software maintenance. They do claim that permanent re-factoring turns maintenance into a normal state of the methods. However, they do not explain how such methods would work over extended periods of time, when a development team is sure to disperse with the knowledge it has of the implementation details. Documentation is still a highly relevant artifact of software maintenance.

Legacy software re-documentation tries to remedy the deficiencies of the past in terms of up-to-date documentation. However, it is a costly activity, difficult to justify to users because it does not bring any visible change for them (at least in the short term).

In this paper we present a survey of software maintainers trying to establish the importance of various documentation artifacts for maintenance. The paper is divided as follows: In Section 2, we review some basic facts about software maintenance and its needs; in Section 3, we summarize the relevant literature on software documentation; in Section 4, we present the survey we conducted; and in Section 5, we comment the result of the survey.

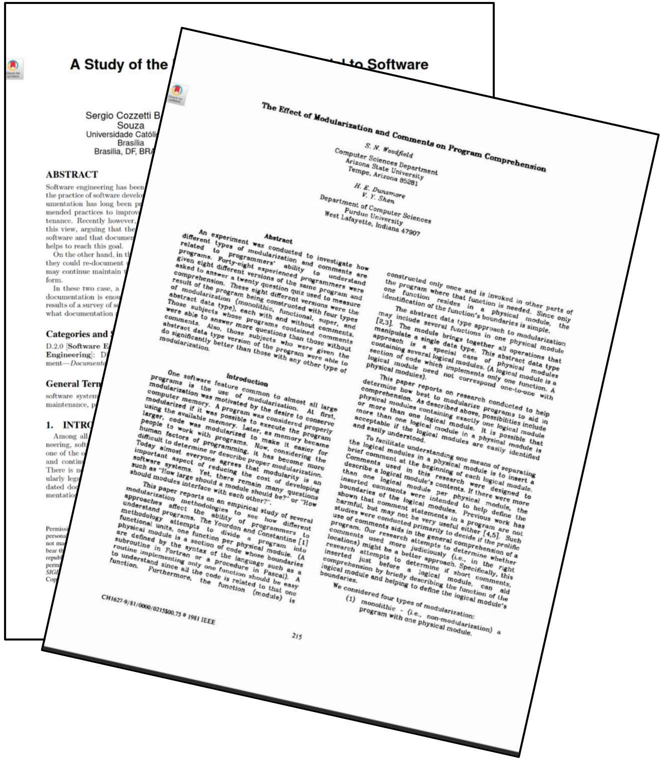
2. SOFTWARE MAINTENANCE

Maintenance is traditionally defined as any modification made on a system after its delivery. Studies show that software maintenance is, by far, the predominant activity in software engineering (90% of the total cost of a typical software [17, 21]). It is needed to keep software systems up-to-date and useful: Any software system reflects the world within which it operates, when this world changes, the software needs to change accordingly. Lehman's first law of software evolution (law of continuing change, [13]) is that "a program that is used undergoes continual change or becomes progressively less useful". Maintenance is mandatory,

SIGDOC'05, September 21–23, 2005, Coventry, United Kingdom.

Documenting code is **crucially** important.

BACKGROUND



Documenting code is **crucially** important.

It allows developers to better **understand** code.

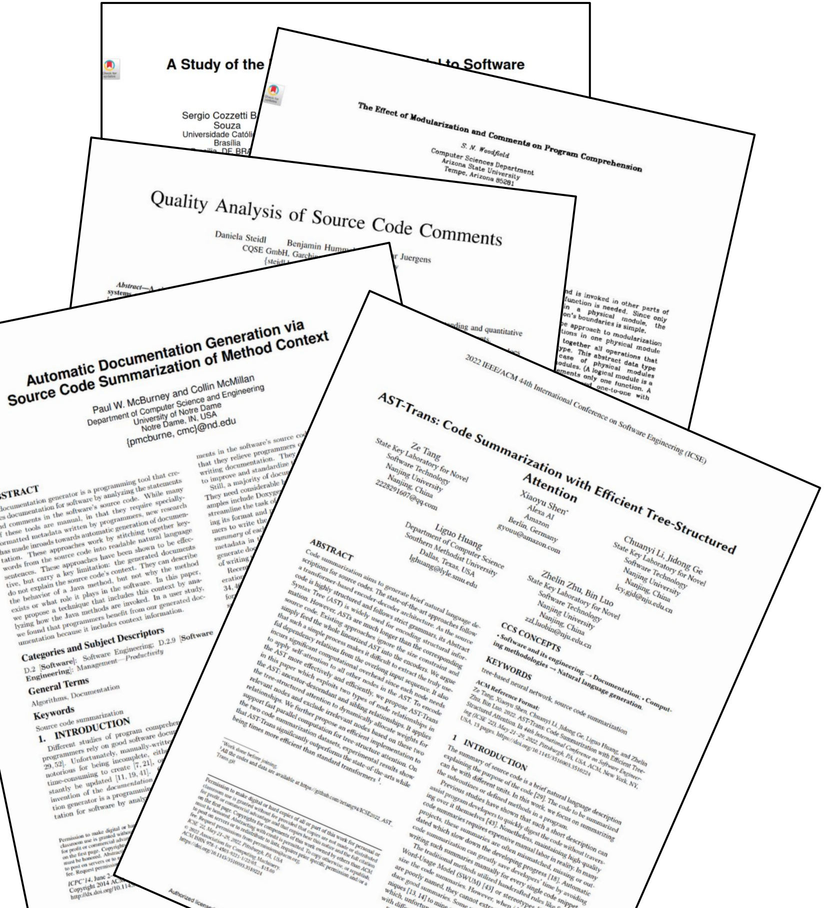
BACKGROUND

Documenting code is **crucially** important.

It allows developers to better **understand** code.

Documenting code is both **labor-intensive** and frequently **neglected**.

BACKGROUND



Documenting code is **crucially** important.

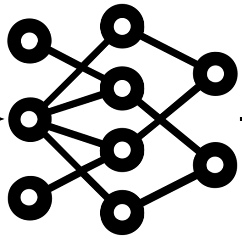
It allows developers to better **understand** code.

Documenting code is both **labor-intensive** and frequently **neglected**.

Automated **code summarization** has emerged as a promising solution.

CODE SUMMARIZATION

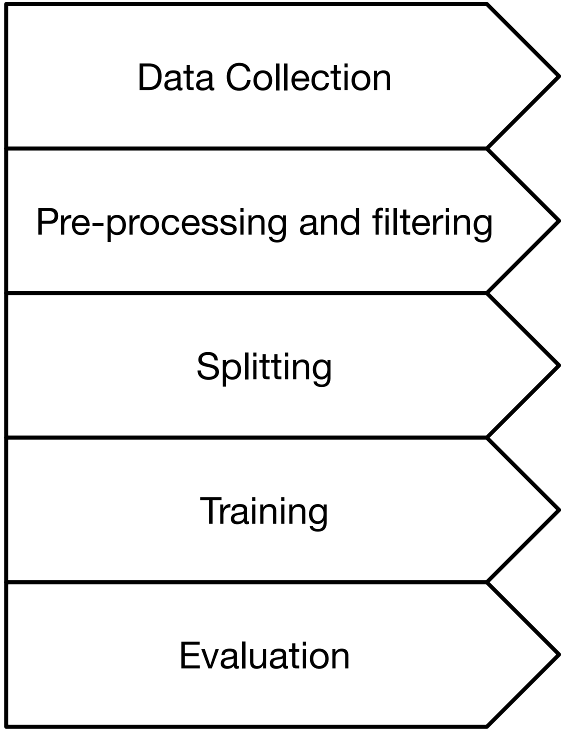
```
public String toString() {
      final StringBuffer s = new StringBuffer();
      final int size = size();
      for (int i = 0; i < size; i++)
          s.append(getInt(i));
      return s.toString();
}
```



DL-Model

```
Returns a string representation of this vector.
```

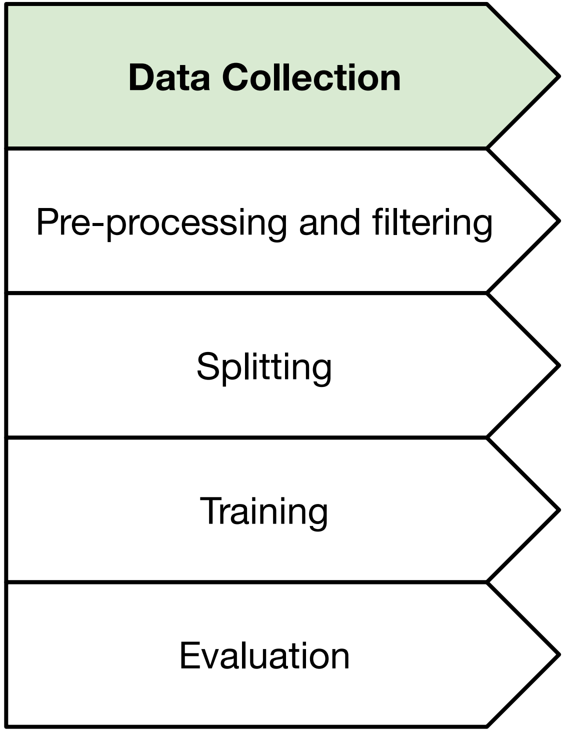
PIPELINE



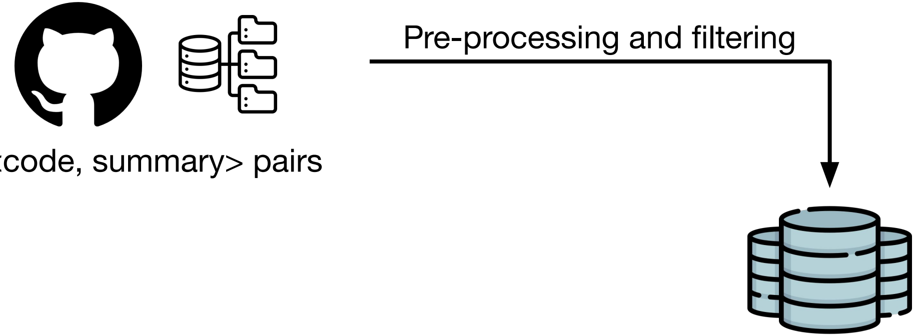
Data Collection
Pre-processing and filtering
Splitting
Training
Evaluation

PIPELINE

<code, summary> pairs



PIPELINE



Data Collection

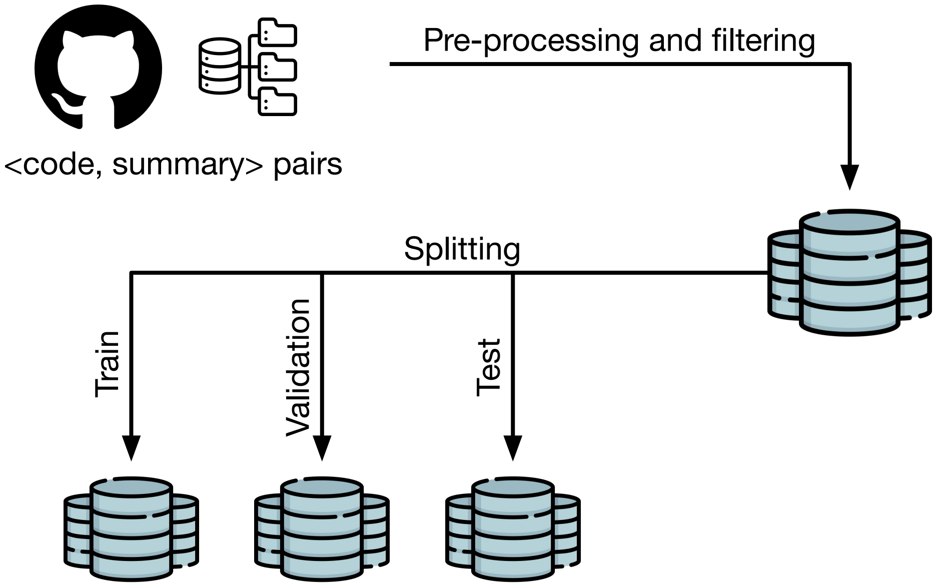
Pre-processing and filtering

Splitting

Training

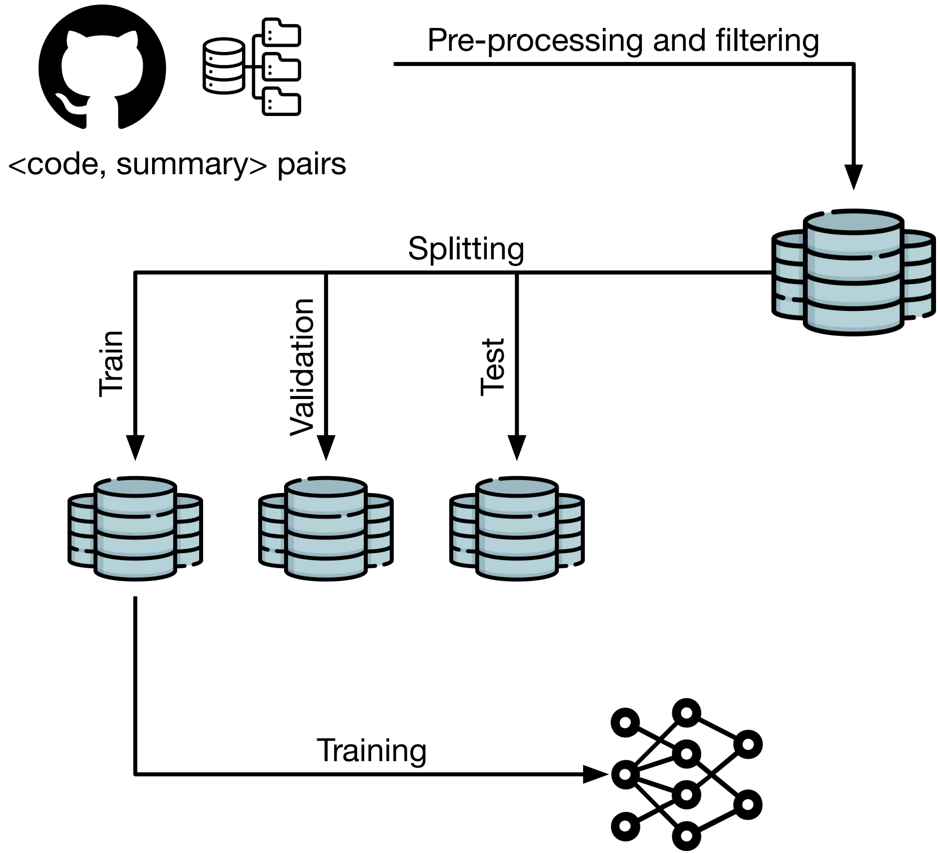
Evaluation

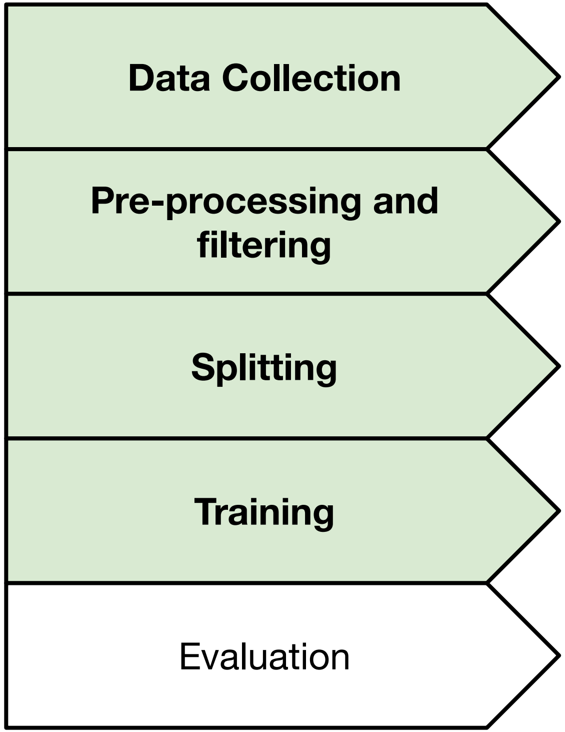
BACKGROUND



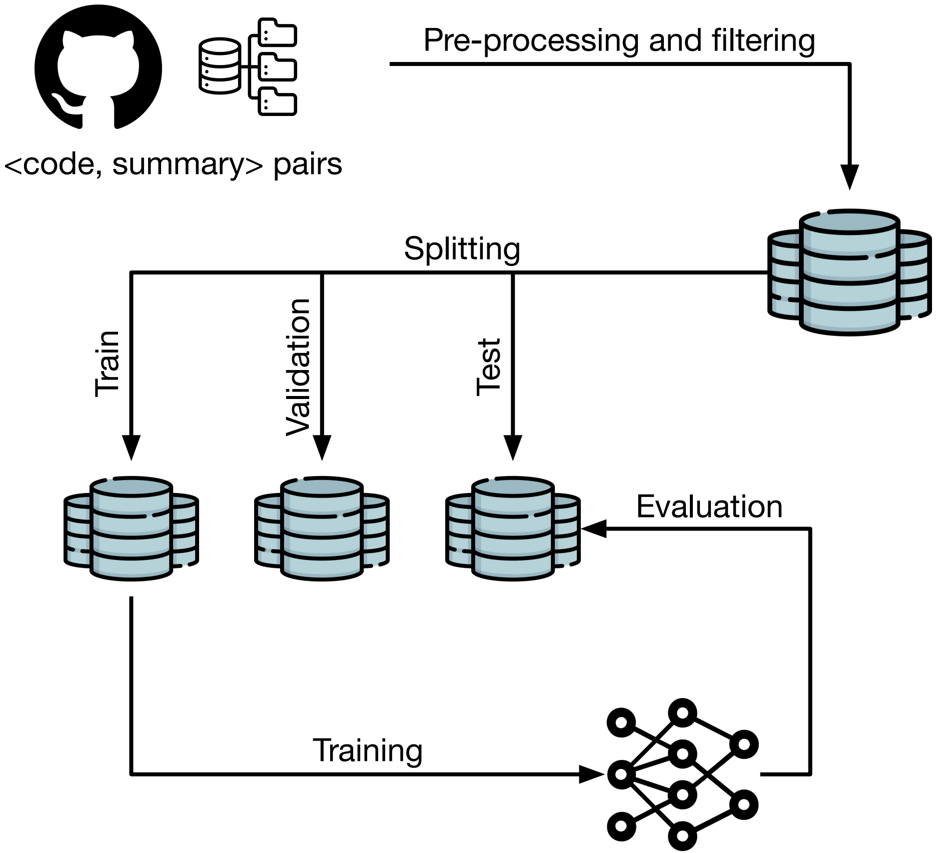
- **Data Collection**
- **Pre-processing and filtering**
- **Splitting**
- Training
- Evaluation

BACKGROUND


<code, summary> pairs
Pre-processing and filtering
Splitting
Train
Validation
Test
Training

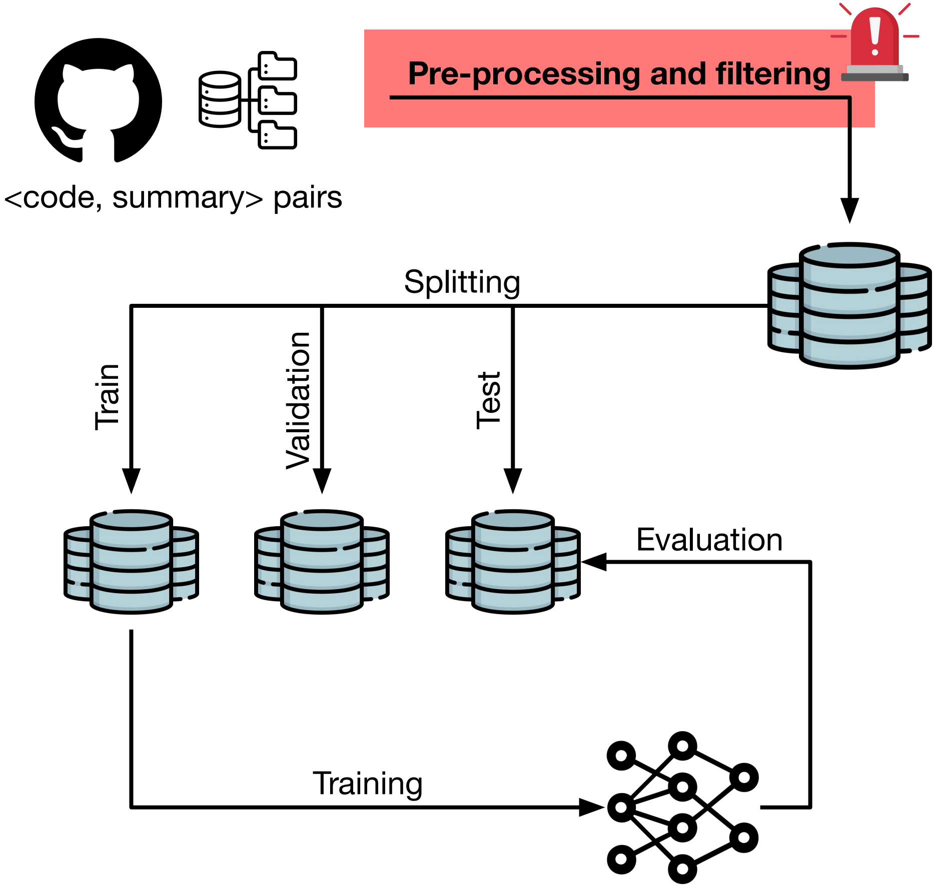

Data Collection
Pre-processing and filtering
Splitting
Training
Evaluation

PIPELINE



- **Data Collection**
- **Pre-processing and filtering**
- **Splitting**
- **Training**
- **Evaluation**

PIPELINE



- **Data Collection**
- **Pre-processing and filtering**
- **Splitting**
- **Training**
- **Evaluation**

LOW QUALITY INSTANCES

```
/* Returns the high-value
 * for an item within a series. */
```

LOW QUALITY INSTANCES

```
/* Returns the high-value
 * for an item within a series. */
```

```
returns the high value
```

LOW QUALITY INSTANCES

```
/* Returns the high-value
 * for an item within a series. */
```

returns the high value

```
/* <p> Builds the JASPIC application context. </p> */
```

p builds the jaspic application context p

LOW QUALITY INSTANCES

```
/* Returns the high-value
 * for an item within a series. */
```

→

```
returns the high value
```

```
/* <p> Builds the JASPIC application context. </p> */
```

→

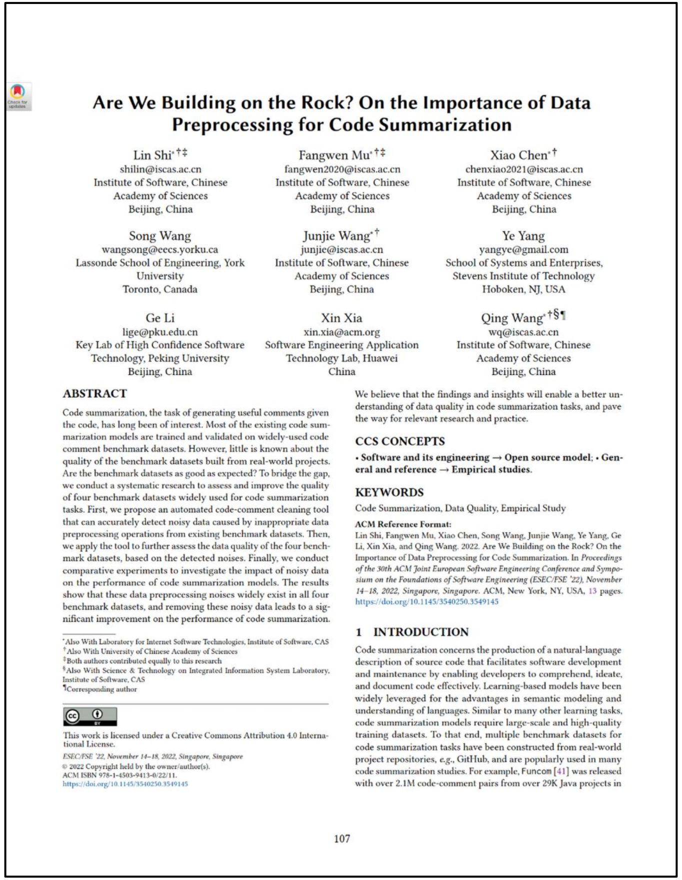
```
p builds the jaspic application context p
```

```
public void testConstructor() {
      System TestResult str;
      System TestID testID1;
      ...
}
```

→

```
test the constructor
```

LOW QUALITY INSTANCES



Are We Building on the Rock? On the Importance of Data Preprocessing for Code Summarization

Lin Shi[*,†,‡] shilin@iscas.ac.cn Institute of Software, Chinese Academy of Sciences Beijing, China

Fangwen Mu[*,†,‡] fangwen2020@iscas.ac.cn Institute of Software, Chinese Academy of Sciences Beijing, China

Xiao Chen[*,†] chenxiao2021@iscas.ac.cn Institute of Software, Chinese Academy of Sciences Beijing, China

Song Wang wangsong@eecs.yorku.ca Lassonde School of Engineering, York University Toronto, Canada

Junjie Wang[*,†] junjie@iscas.ac.cn Institute of Software, Chinese Academy of Sciences Beijing, China

Ye Yang yangye@gmail.com School of Systems and Enterprises, Stevens Institute of Technology Hoboken, NJ, USA

Ge Li lige@pku.edu.cn Key Lab of High Confidence Software Technology, Peking University Beijing, China

Xin Xia xin.xia@acm.org Software Engineering Application Technology Lab, Huawei China

Qing Wang[*,†,§,¶] wq@iscas.ac.cn Institute of Software, Chinese Academy of Sciences Beijing, China

ABSTRACT

Code summarization, the task of generating useful comments given the code, has long been of interest. Most of the existing code summarization models are trained and validated on widely-used code comment benchmark datasets. However, little is known about the quality of the benchmark datasets built from real-world projects. Are the benchmark datasets as good as expected? To bridge the gap, we conduct a systematic research to assess and improve the quality of four benchmark datasets widely used for code summarization tasks. First, we propose an automated code-comment cleaning tool that can accurately detect noisy data caused by inappropriate data preprocessing operations from existing benchmark datasets. Then, we apply the tool to further assess the data quality of the four benchmark datasets, based on the detected noises. Finally, we conduct comparative experiments to investigate the impact of noisy data on the performance of code summarization models. The results show that these data preprocessing noises widely exist in all four benchmark datasets, and removing these noisy data leads to a significant improvement on the performance of code summarization. We believe that the findings and insights will enable a better understanding of data quality in code summarization tasks, and pave the way for relevant research and practice.

CCS CONCEPTS

• Software and its engineering → Open source model; • General and reference → Empirical studies.

KEYWORDS

Code Summarization, Data Quality, Empirical Study

ACM Reference Format:
Lin Shi, Fangwen Mu, Xiao Chen, Song Wang, Junjie Wang, Ye Yang, Ge Li, Xin Xia, and Qing Wang. 2022. Are We Building on the Rock? On the Importance of Data Preprocessing for Code Summarization. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '22), November 14–18, 2022, Singapore, Singapore.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3540250.3549145

1 INTRODUCTION

Code summarization concerns the production of a natural-language description of source code that facilitates software development and maintenance by enabling developers to comprehend, ideate, and document code effectively. Learning-based models have been widely leveraged for the advantages in semantic modeling and understanding of languages. Similar to many other learning tasks, code summarization models require large-scale and high-quality training datasets. To that end, multiple benchmark datasets for code summarization tasks have been constructed from real-world project repositories, *e.g.*, GitHub, and are popularly used in many code summarization studies. For example, Funcom [41] was released with over 2.1M code-comment pairs from over 29K Java projects in

[*] Also With Laboratory for Internet Software Technologies, Institute of Software, CAS
[†] Also With University of Chinese Academy of Sciences
[‡] Both authors contributed equally to this research
[§] Also With Science & Technology on Integrated Information System Laboratory, Institute of Software, CAS
[¶] Corresponding author

ESEC/FSE '22, November 14–18, 2022, Singapore, Singapore

ACM ISBN 978-1-4503-9413-0/22/11.
https://doi.org/10.1145/3540250.3549145

Propose **CAT** (**C**ode-comment cle**A**ning **T**ool), a **rule-based filtering tool** for automatically scanning and detecting the occurrences and distribution of data noises for a given dataset.

FSE 22’

LOW QUALITY INSTANCES

```
public HashSet getCommandResultsRootFeatures() {
      HashSet rootFeatureSet = new HashSet();
      Feature belowSplitRoot = null;
      Feature aboveSplitRoot = null;
      if (belowSplitTranscript != null) {
          belowSplitRoot = belowSplitTranscript.getRootFeature();
          rootFeatureSet.add(belowSplitRoot);
      }
      if (aboveSplitTranscript != null) {
            aboveSplitRoot = aboveSplitTranscript.getRootFeature();
            if (aboveSplitRoot != belowSplitRoot)
                  rootFeatureSet.add(aboveSplitRoot);
      }
      return rootFeatureSet;
}
```

Invoked AFTER the command is executed.

LOW QUALITY INSTANCES

Evaluating Code Summarization Techniques: A New Metric and an Empirical Characterization

Antonio Mastropaolo
antonio.mastropaolo@usi.ch
SEART @ Software Institute,
Università della Svizzera Italiana
Lugano, Switzerland, CH

Matteo Ciniselli
matteo.ciniselli@usi.ch
SEART @ Software Institute,
Università della Svizzera Italiana
Lugano, Switzerland, CH

Massimiliano Di Penta
dipenta@unisannio.it
Dept. of Engineering,
University of Sannio
Benevento, Italy, IT

Gabriele Bavota
gabriele.bavota@usi.ch
SEART @ Software Institute,
Università della Svizzera Italiana
Lugano, Switzerland, CH

ABSTRACT

Several code summarization techniques have been proposed in the literature to automatically document a code snippet or a function. Ideally, software developers should be involved in assessing the quality of the generated summaries. However, in most cases, researchers rely on automatic evaluation metrics such as BLEU, ROUGE, and METEOR. These metrics are all based on the same assumption: The higher the textual similarity between the generated summary and a reference summary written by developers, the higher its quality. However, there are two reasons for which this assumption falls short: (i) reference summaries, *e.g.*, code comments collected by mining software repositories, may be of low quality or even outdated; (ii) generated summaries, while using a different wording than a reference one, could be semantically equivalent to it, thus still being suitable to document the code snippet. In this paper, we perform a thorough empirical investigation on the complementarity of different types of metrics in capturing the quality of a generated summary. Also, we propose to address the limitations of existing metrics by considering a new dimension, capturing the extent to which the generated summary aligns with the semantics of the documented code snippet, independently from the reference summary. To this end, we present a new metric based on contrastive learning to capture said aspect. We empirically show that the inclusion of this novel dimension enables a more effective representation of developers' evaluations regarding the quality of automatically generated summaries.

CCS CONCEPTS

• **Software and its engineering → Documentation.**

ICSE '24, April 14–20, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0217-4/24/04…$15.00
https://doi.org/10.1145/3597503.3639174

KEYWORDS

Code Summarization, Contrastive Learning

ACM Reference Format:
Antonio Mastropaolo, Matteo Ciniselli, Massimiliano Di Penta, and Gabriele Bavota. 2024. Evaluating Code Summarization Techniques: A New Metric and an Empirical Characterization. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3597503.3639174

1 INTRODUCTION

Program comprehension can take up to 58% of developers' time [90]. Code comments are considered the most important form of documentation in this activity [16]. Despite the undisputed importance of code comments, developers do not always carefully comment code, or update existing comments in response to code changes [75]. This may result in a lack of documentation [16, 72] and/or in outdated code comments [18, 19, 46, 86]. To support developers in such a task, researchers proposed code summarization techniques [4, 6, 26, 29, 32, 37, 41, 52, 53, 64, 67, 73, 73, 87, 88, 93]. These approaches take as input a code component to document (*e.g.*, a code function, or an entire class) and provide as output a natural language summary describing the code. The underlying technique can range from pre-defined templates properly filled via code analysis to the most recent techniques exploiting deep learning models trained on $\langle C, S \rangle$ pairs mined from software repositories, where C represents the code to document and S the original summary (comment) written by developers.

Empirically evaluating the quality of code summaries generated by these approaches is far from trivial. Indeed, assessing the extent to which a natural language text represents a good summary for a code component would require human (developers) judgment. Given the difficulties of running large-scale evaluations with developers, the software engineering community borrowed evaluation metrics from the Natural Language Processing (NLP) field. These include (but are not limited to) BLEU [58], ROUGE [45], and METEOR [8]. These metrics have been originally designed to act as a proxy for the quality of automatically generated text (*e.g.*, a translation) by comparing it with a reference (expected) text: The higher the words' overlap between the generated and the reference text, the higher the assessed quality.

They propose **SIDE** (**S**ummary al**I**gnment to co**D**e s**E**mantics)**,** a new metric leveraging contrastive learning to model the characteristics of **suitable and unsuitable** code summaries for a given code.

ICSE 24'

LOW QUALITY INSTANCES

METHOD

```
public Element asElement() {
    return this.component.createCopy();
}
```

REFERENCE SUMMARY

not yet documented

GENERATED SUMMARY

returns a copy of this component as an element

SCORES

SOTA							OUR	HUMAN			
BLEU-1	BertScore-R	SentenceBERT_CS	InferNet_CS	ROUGE-1-P	Rouge-4-R	Rouge-W-R	SIDE	DA	Adequacy	Concise	Fluency
0.34	0.00	0.08	0.43	0.00	0.00	0.00	0.91	88	4	4	5

"**SIDE** is the metric that better describes humans' assessment of summary quality."

LOW QUALITY INSTANCES

METHOD

```java
public Element asElement() {
    return this.component.createCopy();
}
```

REFERENCE SUMMARY

not yet documented

GENERATED SUMMARY

returns a copy of this component as an element

SCORES

SOTA							OUR	HUMAN			
BLEU-1	BertScore-R	SentenceBERT_CS	InferNet_CS	ROUGE-1-P	Rouge-4-R	Rouge-W-R	SIDE	DA	Adequacy	Concise	Fluency
0.34	*0.00*	*0.08*	*0.43*	*0.00*	*0.00*	*0.00*	*0.91*	*88*	*4*	*4*	*5*

"**SIDE** is the metric that better describes humans' assessment of summary quality."

"Provides a continuous score ranging between [-1, 1]"

LOW QUALITY INSTANCES

METHOD

```java
public Element asElement() {
    return this.component.createCopy();
}
```

~~REFERENCE SUMMARY~~

~~not~~

GENERATED SUMMARY

returns a copy of this component as an element

SCORES

SOTA							OUR	HUMAN			
BLEU-1	BertScore-R	SentenceBERT_CS	InferNet_CS	ROUGE-1-P	Rouge-4-R	Rouge-W-R	SIDE	DA	Adequacy	Concise	Fluency
0.34	0.00	0.08	0.43	0.00	0.00	0.00	0.91	88	4	4	5

“**SIDE** is the metric that better describes humans’ assessment of summary quality.”

“Provides a continuous score ranging between [-1, 1]”

LOW QUALITY INSTANCES

```
public Element asElement() {
    return this.component.createCopy();
}
```

```
not yet documented
```

SIDE

LOW QUALITY INSTANCES

```
public Element asElement() {
    return this.component.createCopy();
}
```

```
not yet documented
```

SIDE → **-0.19**

LOW QUALITY INSTANCES

```
public Element asElement() {
    return this.component.createCopy();
}
```

not yet documented

SIDE

-0.19

```
public Element asElement() {
    return this.component.createCopy();
}
```

returns a copy of this component as an element

SIDE

0.91

LOW QUALITY INSTANCES

```
public Element asElement() {
    return this.component.createCopy();
}
```

```
not yet documented
```

SIDE → -0.19 → Incoherent

```
public Element asElement() {
    return this.component.createCopy();
}
```

```
returns a copy of this component as an
element
```

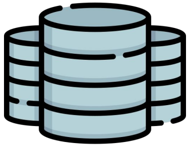
SIDE → 0.91 → Coherent

QUESTION

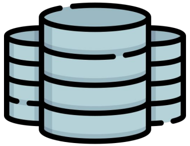
Can filtering out **incoherent code-comment pairs** serve as an effective strategy for **optimizing** code-summarization datasets**?**

EMPIRICAL STUDY

Training Sets

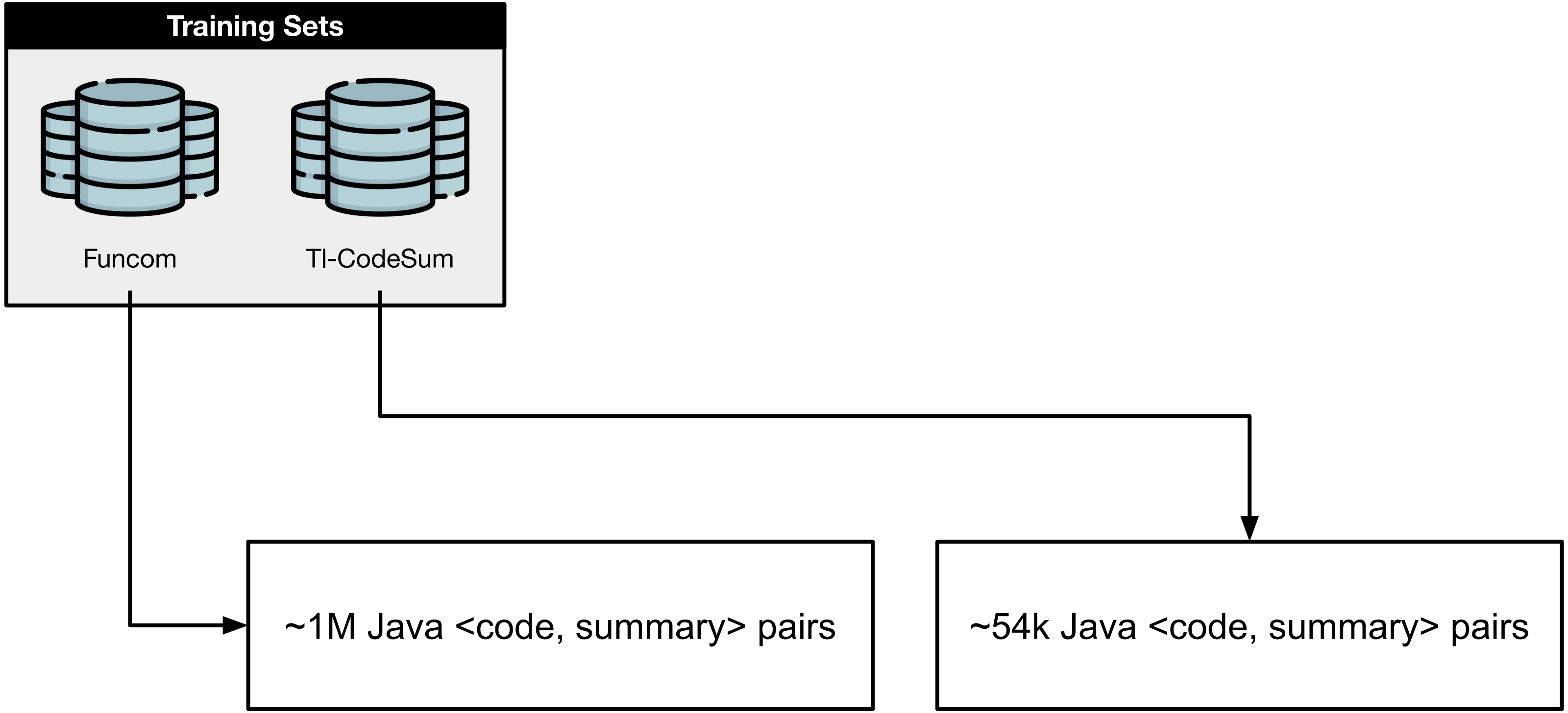


Funcom

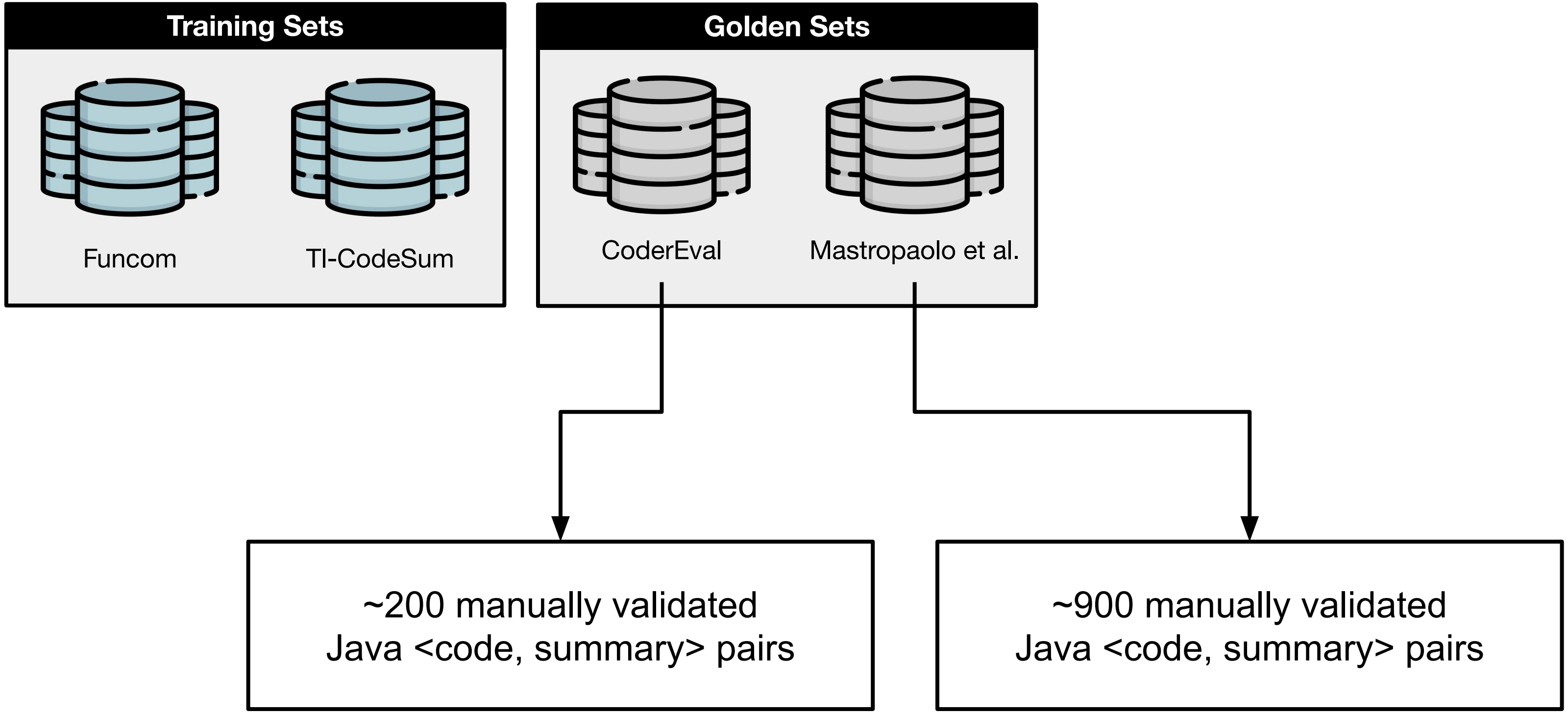


TI-CodeSum

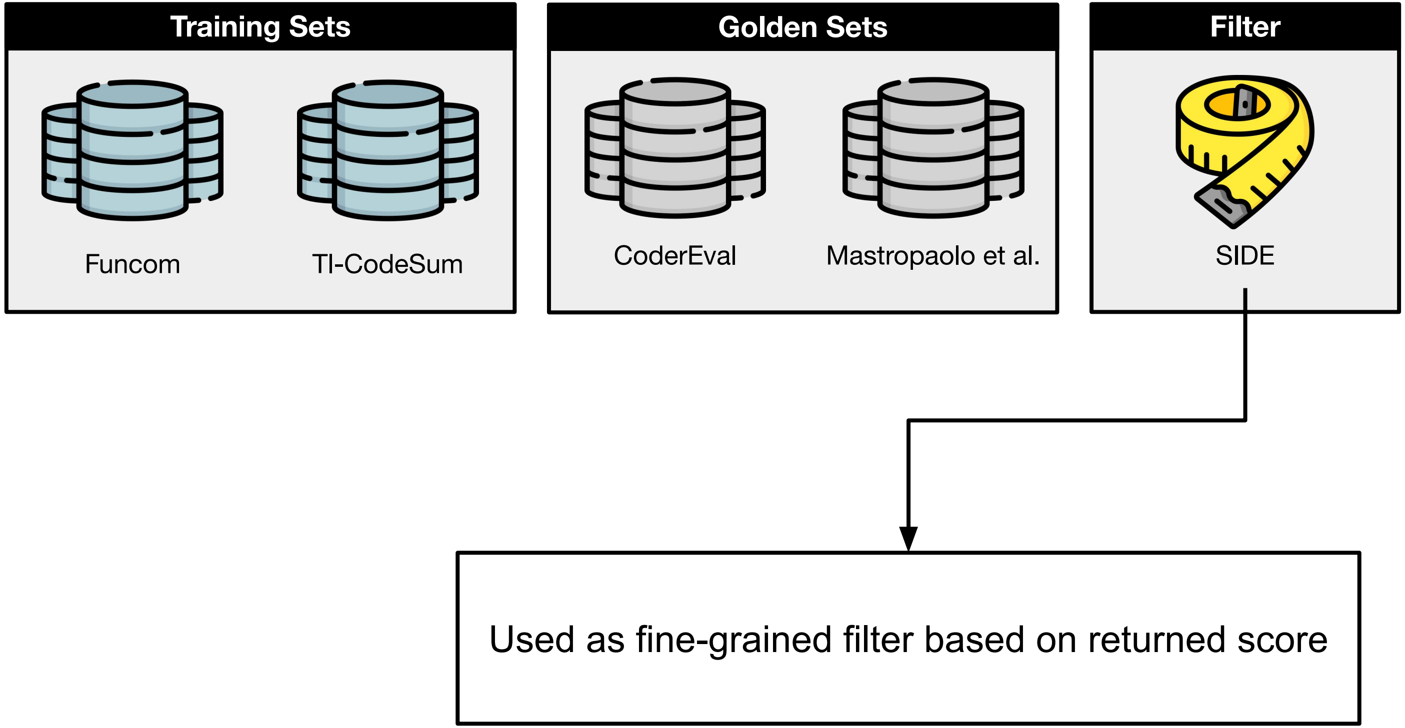
EMPIRICAL STUDY



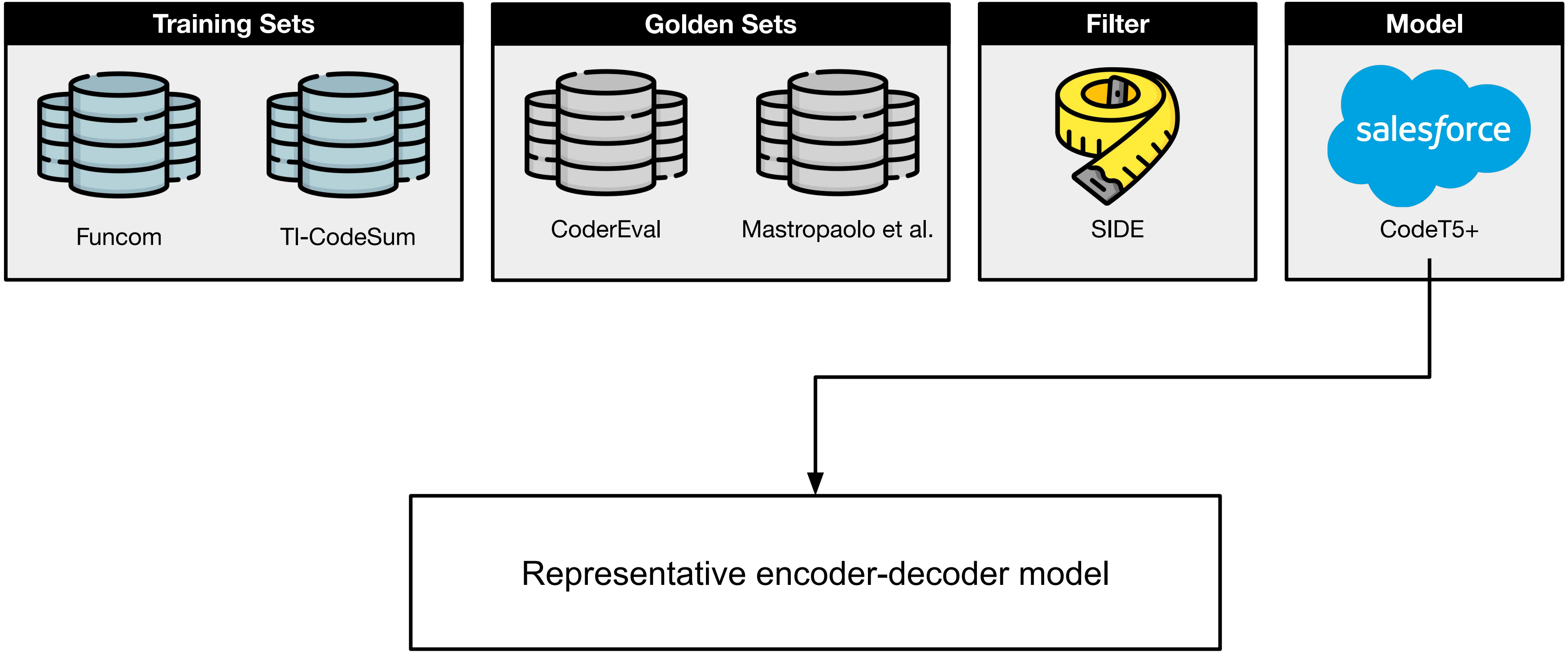
EMPIRICAL STUDY



EMPIRICAL STUDY



EMPIRICAL STUDY



RQ0

How do code summarization datasets measure up in terms of **code-comment coherence?**

RQ0

FUNCOM

0.81 mean SIDE score

RQ0

FUNCOM	TL-CODESUM
0.81 mean SIDE score	0.83 mean SIDE score

RQ0

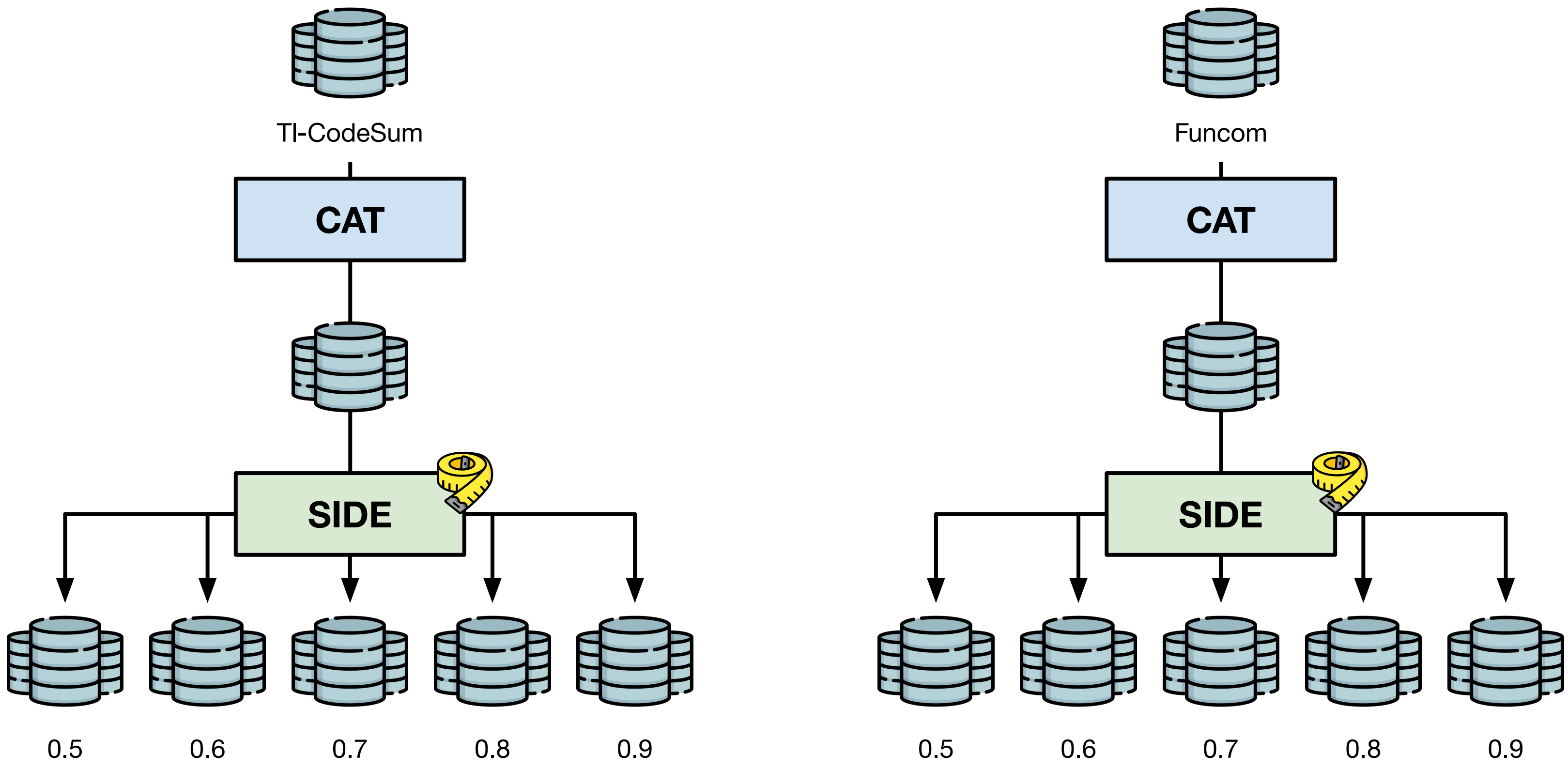
FUNCOM	TL-CODESUM
0.81 mean SIDE score	0.83 mean SIDE score

More than 50% of the instances
have sub-optimal SIDE scores below 0.9.

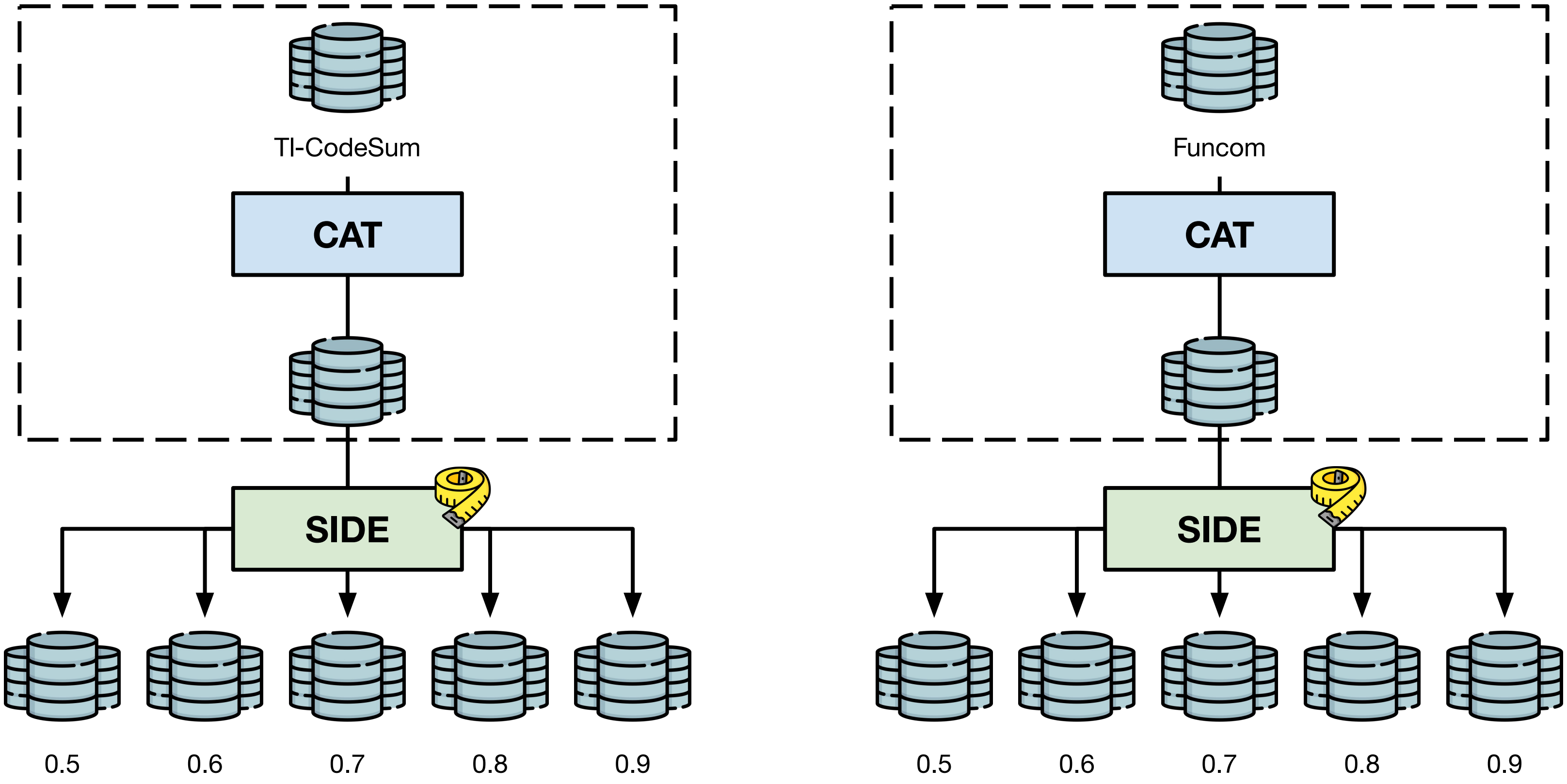
RQ1

How does a **coherence-aware strategy** selection **impact the performance** of neural code summarization models?

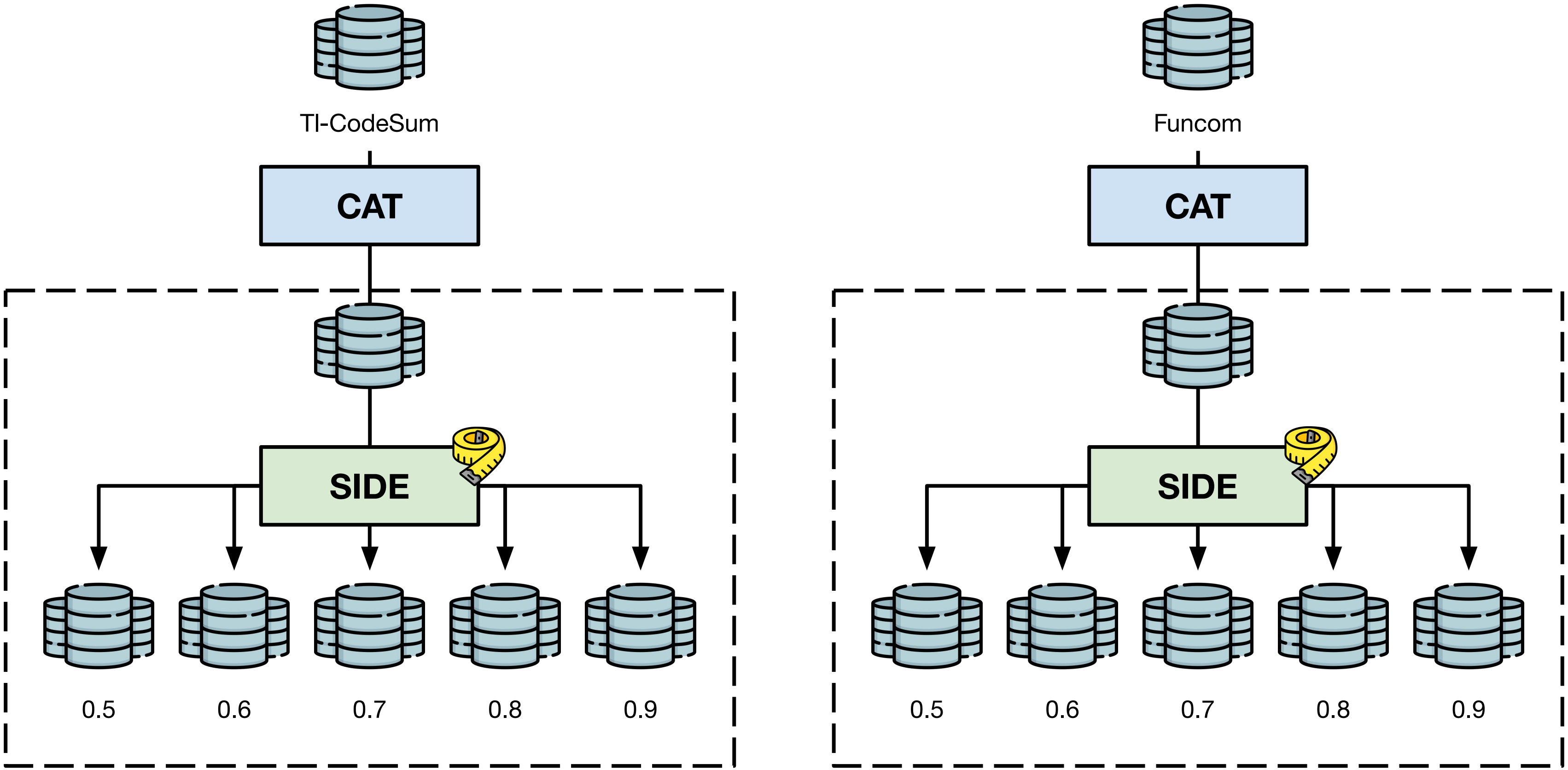
RQ1



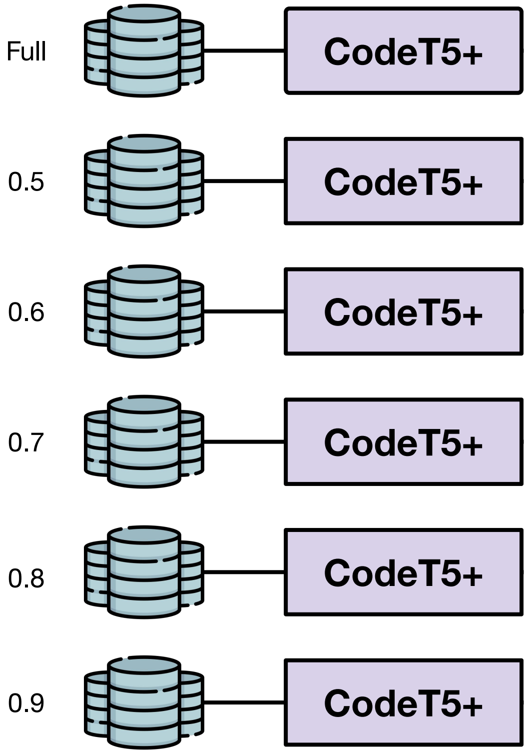
RQ1



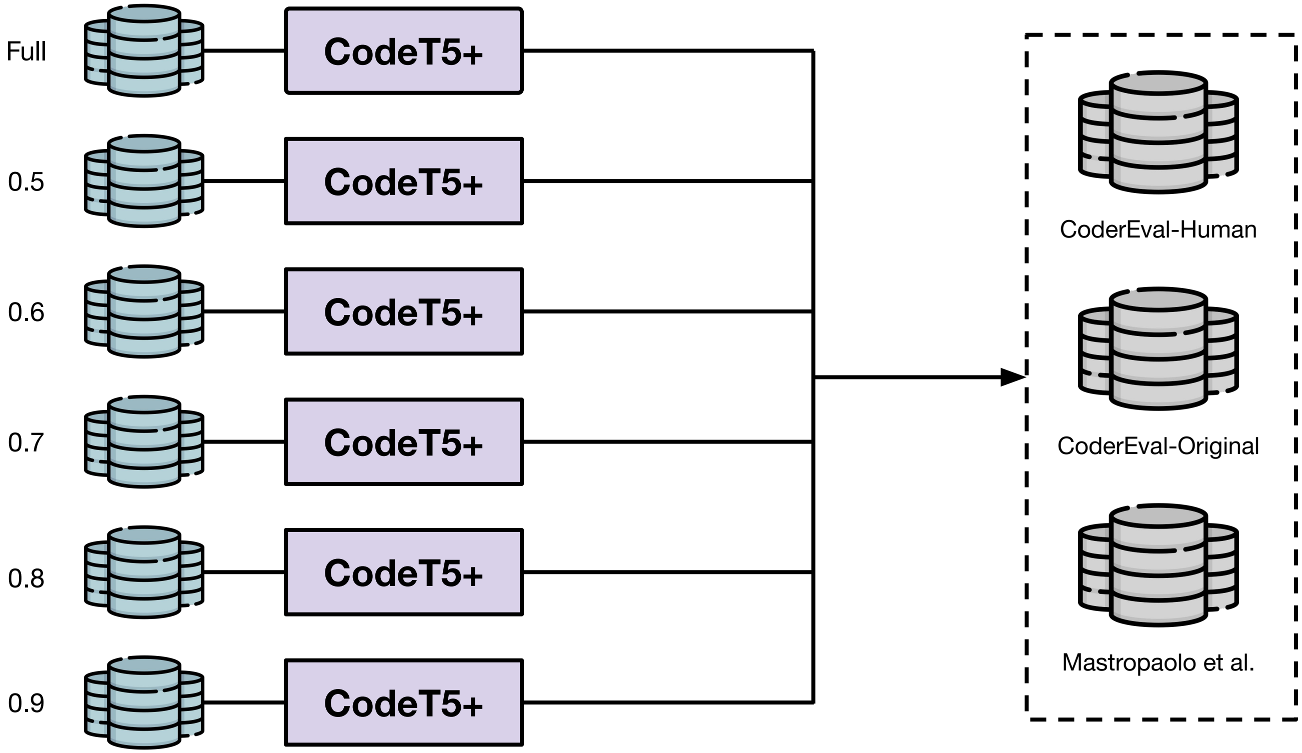
RQ1



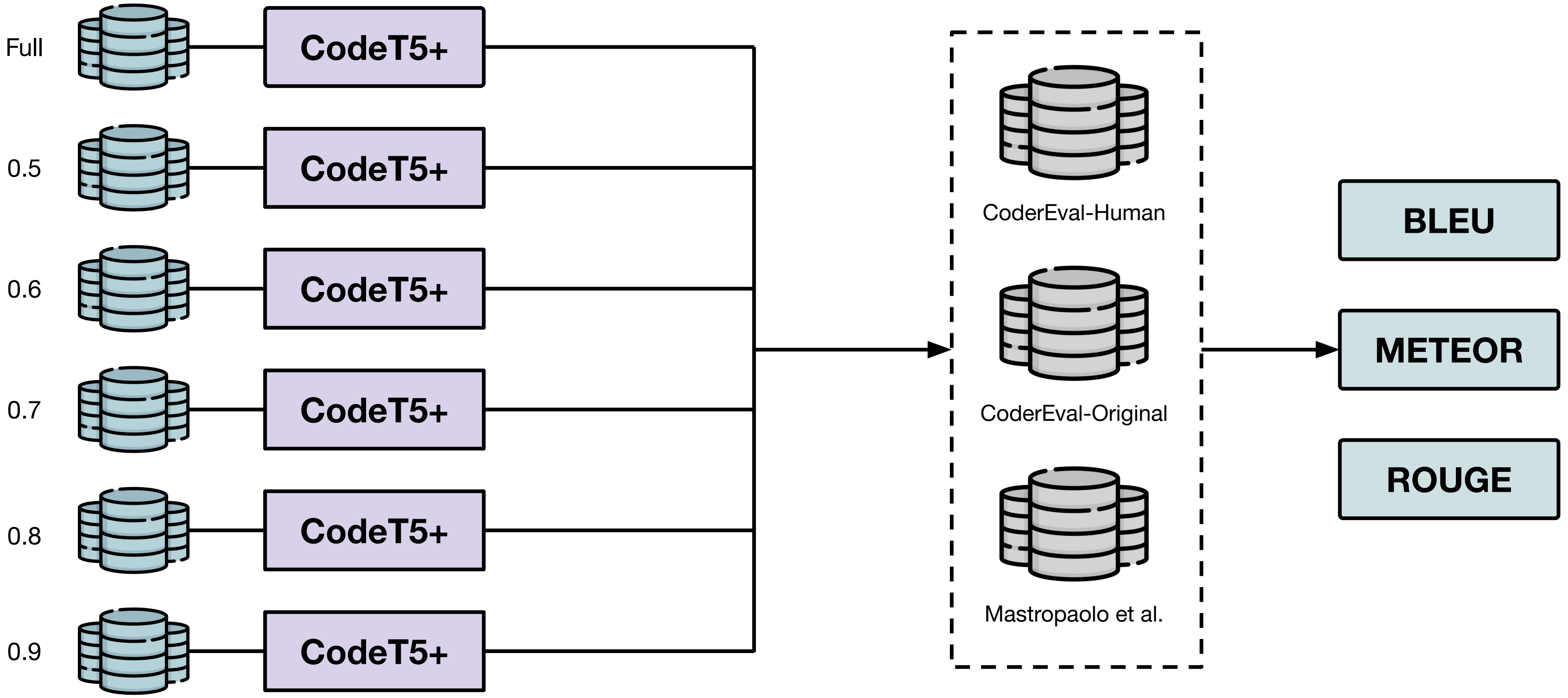
RQ1


Full
CodeT5+
0.5
CodeT5+
0.6
CodeT5+
0.7
CodeT5+
0.8
CodeT5+
0.9
CodeT5+

RQ1


Full
0.5
0.6
0.7
0.8
0.9
CodeT5+
CodeT5+
CodeT5+
CodeT5+
CodeT5+
CodeT5+
CoderEval-Human
CoderEval-Original
Mastropaolo et al.

RQ1



Full
0.5
0.6
0.7
0.8
0.9
CodeT5+
CodeT5+
CodeT5+
CodeT5+
CodeT5+
CodeT5+
CoderEval-Human
CoderEval-Original
Mastropaolo et al.
BLEU
METEOR
ROUGE

RQ1

Models **performances are comparable** to those obtained when fine-tuning the model on the **full** training sets.

RQ1

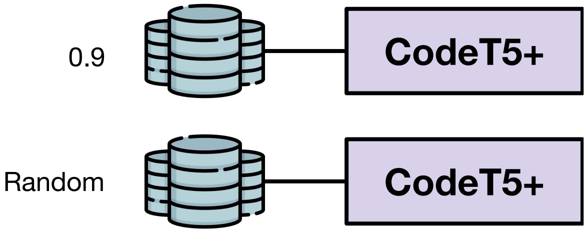
Models **performances are comparable** to those obtained when fine-tuning the model on the **full** training sets.

This happens **using only 50%** of the training set.

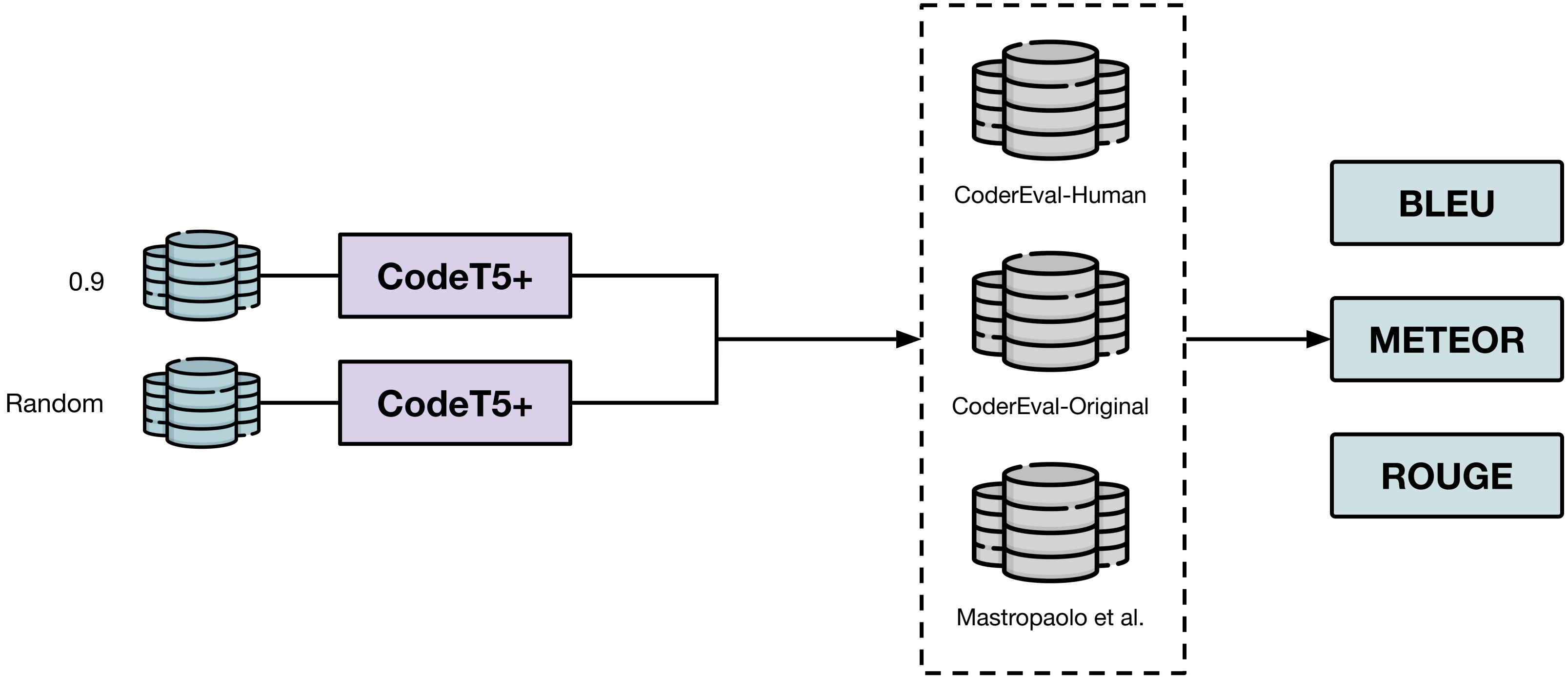
RQ2

How does the coherence-aware strategy selection **compare** with a **random baseline**?

RQ2



RQ2



0.9
Random
CodeT5+
CodeT5+
CoderEval-Human
CoderEval-Original
Mastropaolo et al.
BLEU
METEOR
ROUGE

RQ2

Filtering by code-comment coherence provides models with **comparable** effectiveness to those trained on **randomly selected** instances.

OUTCOMES

Code-comment coherence **might not be a problem** in state-of-the-art datasets.

The results clearly indicate that state-of-the-art datasets contain **instances that do not contribute** to improving the models' effectiveness.

Other quality aspects of code and comments that have not been explored yet (such as readability) may be important for smartly selecting the training instances.

Future work could explore different criteria for data selection that identify the **most informative** subsets for training.

OUTCOMES

Code-comment coherence **might not be a problem** in state-of-the-art datasets.

The results clearly indicate that state-of-the-art datasets contain **instances that do not contribute** to improving the models' effectiveness.

Other quality aspects of code and comments that have not been explored yet (such as readability) may be important for smartly selecting the training instances.

Future work could explore different criteria for data selection that identify the **most informative** subsets for training.

OUTCOMES

Code-comment coherence **might not be a problem** in state-of-the-art datasets.

The results clearly indicate that state-of-the-art datasets contain **instances that do not contribute** to improving the models' effectiveness.

Other quality aspects of code and comments that have not been explored yet (such as readability) may be important for smartly selecting the training instances.

Future work could explore different criteria for data selection that identify the **most informative** subsets for training.

OUTCOMES

Code-comment coherence **might not be a problem** in state-of-the-art datasets.

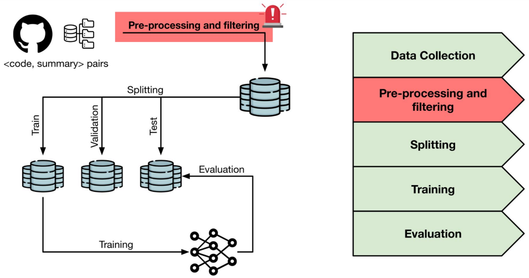
The results clearly indicate that state-of-the-art datasets contain **instances that do not contribute** to improving the models' effectiveness.

Other quality aspects of code and comments that have not been explored yet (such as readability) may be important for smartly selecting the training instances.

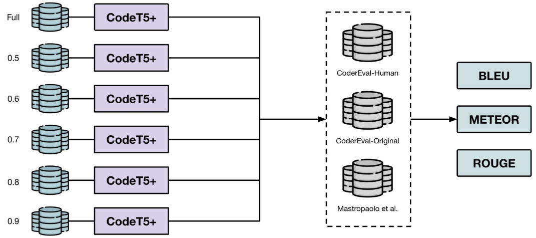
Future work could explore different criteria for data selection that identify the **most informative** subsets for training.

SUMMARY

PIPELINE



RQ1

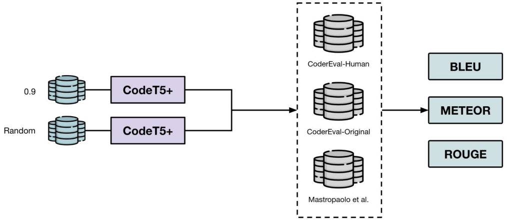


RQ1

Models **performances are comparable** to those obtained when fine-tuning the model on the **full** training sets.

This happens **using only 50%** of the training set.

RQ2



RQ2

Filtering by code-comment coherence provides models with **comparable** effectiveness to those trained on **randomly selected** instances.

OUTCOMES

Code-comment coherence **might not be a problem** in state-of-the-art datasets.

The results clearly indicate that state-of-the-art datasets contain **instances that do not contribute** to improving the models' effectiveness.

Other quality aspects of code and comments that have not been explored yet (such as readability) may be important for smartly selecting the training instances.

Future work could explore different criteria for data selection that identify the **most informative** subsets for training.