Identifying Multi-parameter Constraint Errors in Python Data Science Library API Documentation

Xiufeng Xu[1] Fuman Xie[2] Chenguang Zhu[3] Guangdong Bai[2] Sarfraz Khurshid[3] **Yi Li**[1]

1. Nanyang Technological University

2. University of Queensland

3. University of Texas at Austin

ISSTA '25 – Trondheim, Norway

June 25, 2025

Code-Documentation Inconsistencies in Data Science Libraries

- Confusion[1]: Can solver ***newton-cg***, ***sag***, *and* ***lbfgs*** work with no penalty?
- If users use the API in a biased way, it will lead to poor model training performance like underfitting

Old Doc of LogisticRegression

penalty : *{'l1', 'l2', 'elasticnet', 'none'}, default='l2'*

Used to specify the norm used in the penalization. The 'newton-cg', 'sag' and 'lbfgs' solvers support only l2 penalties. 'elasticnet' is only supported by the 'saga' solver. If 'none' (not supported by the liblinear solver), no regularization is applied.

solver : *{'newton-cg', 'lbfgs', 'liblinear', 'sag', 'saga'}, default='lbfgs'*

Algorithm to use in the optimization problem.

- For small datasets, 'liblinear' is a good choice, whereas 'sag' and 'saga' are faster for large ones.
- For multiclass problems, only 'newton-cg', 'sag', 'saga' and 'lbfgs' handle multinomial loss; 'liblinear' is limited to one-versus-rest schemes.
- 'newton-cg', 'lbfgs', 'sag' and 'saga' handle L2 or no penalty

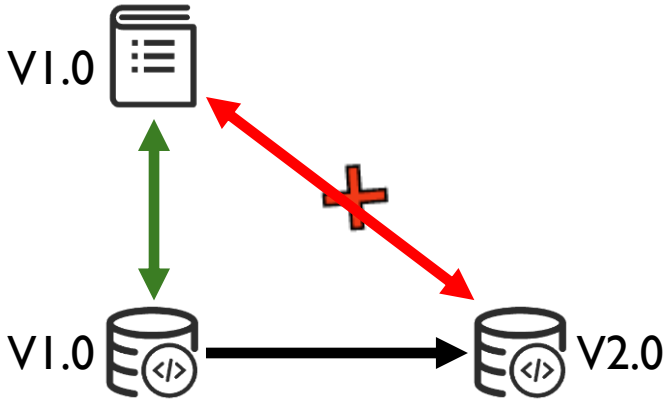
Modified Doc

```
Supported penalties by solver:
- 'newton-cg'   -   ['l2', 'none']
- 'lbfgs'       -   ['l2', 'none']
- 'liblinear'   -   ['l1', 'l2']
- 'sag'         -   ['l2', 'none']
- 'saga'        -   ['elasticnet', 'l1', 'l2', 'none']
```

[1] https://github.com/scikit-learn/scikit-learn/issues/19651

Code-Documentation Inconsistencies in Data Science Libraries

- API documentation and code evolve at different speeds.
- Lots of multi-parameter constraints in data science libraries.
  - No tool do a good job of checking it !!



Constraint description of `trend` and `seasonal` in class `AutoReg`

> **deterministic:** DeterministicProcess
> A deterministic process. If provided, trend and seasonal are ignored. A warning is raised if trend is not "n" and seasonal is not False.

Corresponding code snippet in class `AutoReg`

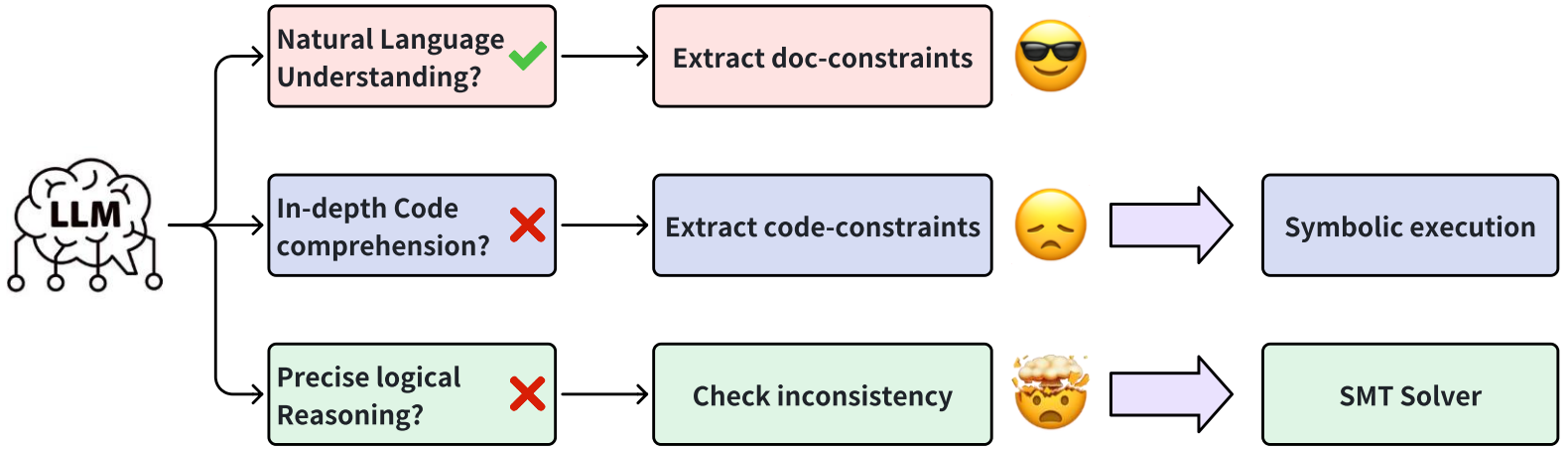
```
1 class AutoReg(tsa_model.TimeSeriesModel):
2     def __init__(...):
3         if deterministic is not None and (self.trend != "n" or self.seasonal):
4             warnings.warn('When using deterministic, trend must be "n"
5                 and seasonal must be False.', SpecificationWarning, stacklevel=2)
```

Why is it challenging?

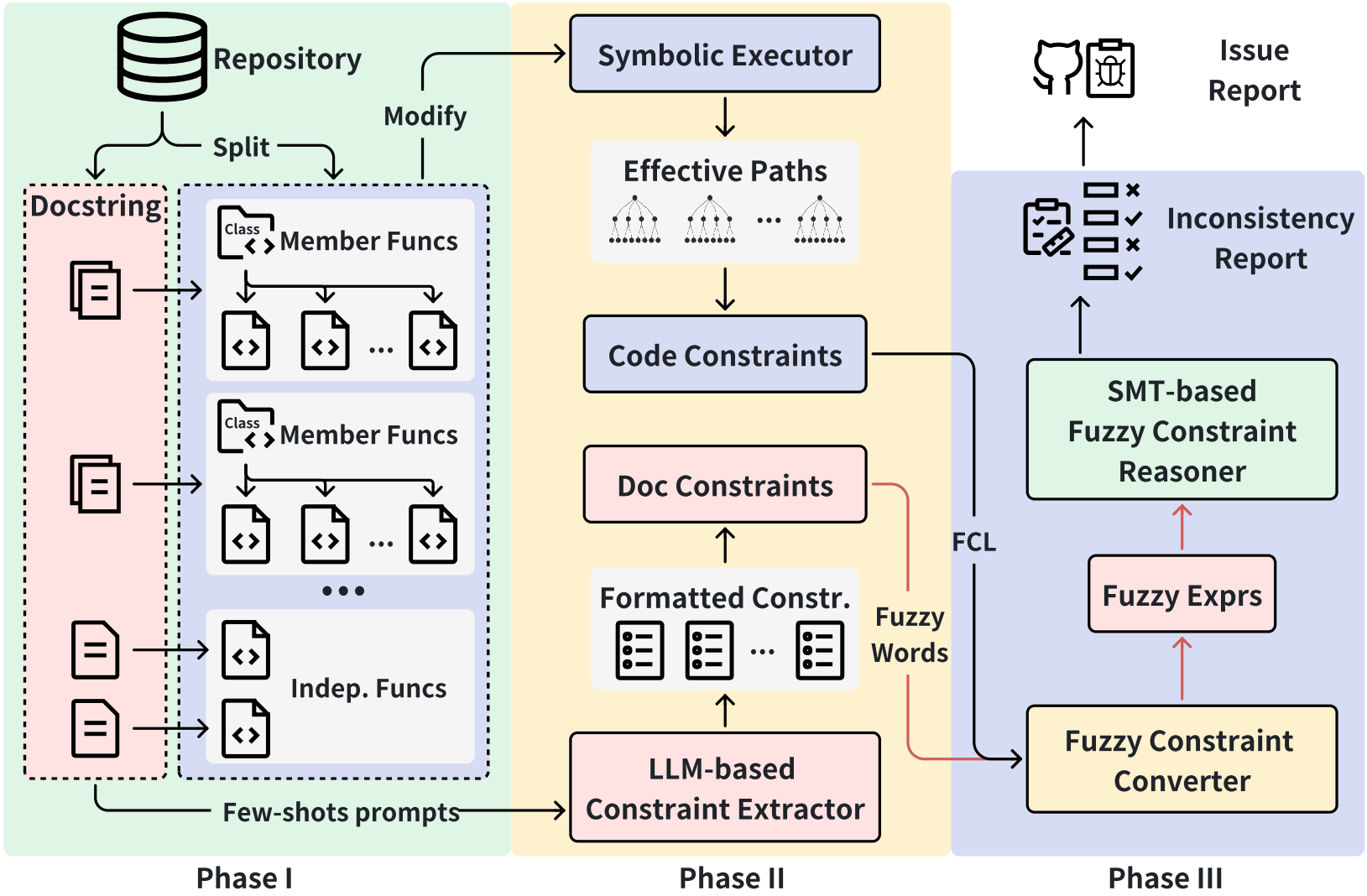
- Parameter-rich interfaces
  - Numerous parameters with complex dependencies
- Silent constraint violations
  - Unexpected behaviors without triggering explicit exceptions
- No fixed format for API documentation
  - Ambiguous descriptions and sometimes unclear/hidden constraints
- Implicit code-doc constraint correspondence
  - Hard to locate code segments and verify specific constraints

LLM may be a promising solution, but …

- LLM-only multi-parameter inconsistency checker? Probably not
  - Unavoidable stochastic behaviors and hallucination
  - Unreliable code comprehension and reasoning capability
- Our proposal: combine LLM strengths in natural language understanding with symbolic reasoning

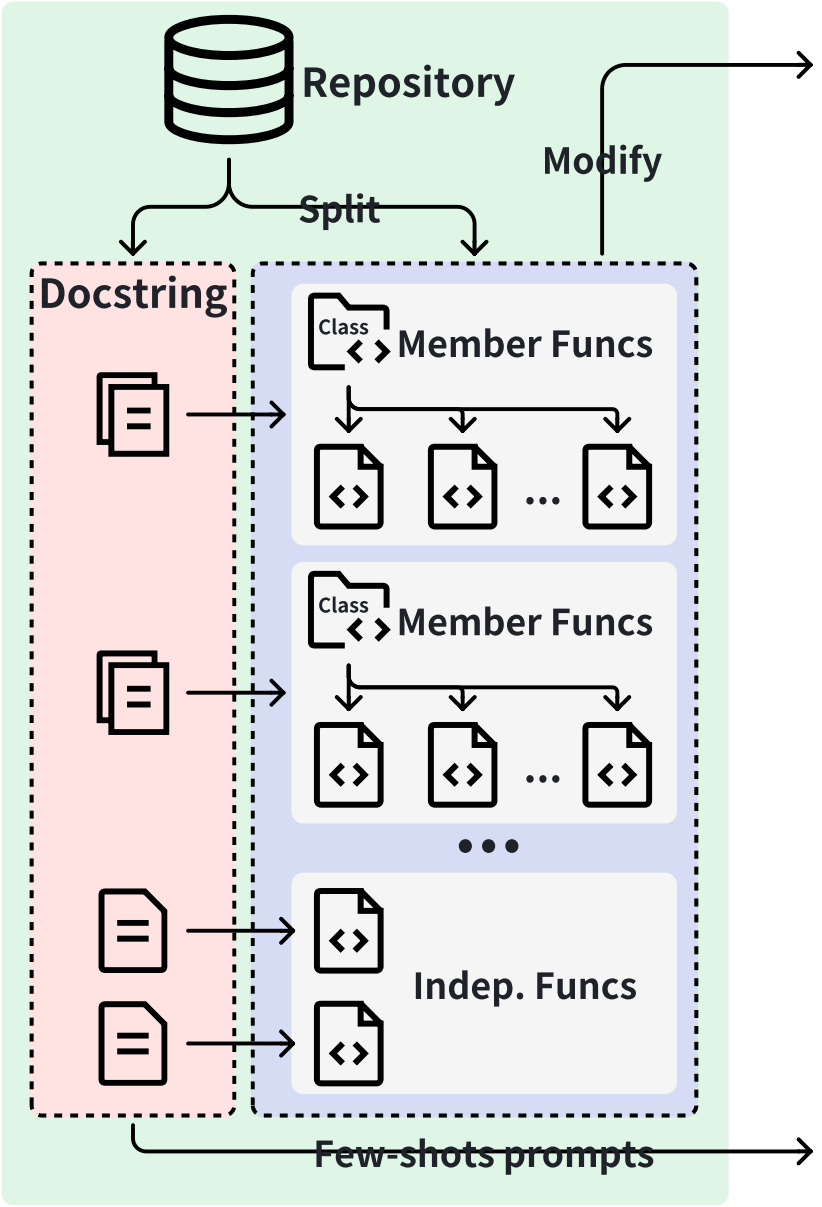


Overview of MPChecker



Phase I: Preprocessing

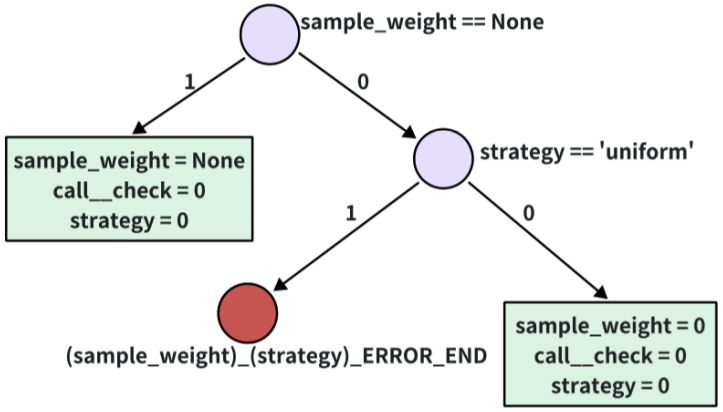
- Split code into analyzable function units
- Extract documentation texts
  - Following Google or Numpy style
- Equivalent Python code transformations to support dynamic symbolic execution
  - Get rid of unnecessary complications that do not affect path constraints
    - E.g., external calls, complex data structures
    - Translate exceptions to special symbols
  - Focus on visible API parameters



Phase II-A: Code-constraint extraction

Original source code

```
1 def fit(self, sample_weight):
2   if sample_weight is not None and self.
        strategy == "uniform":
3     raise ValueError("Warning Info")
4   if sample_weight is not None:
5     sample_weight = _check_sample_weight(
          sample_weight, X)
```

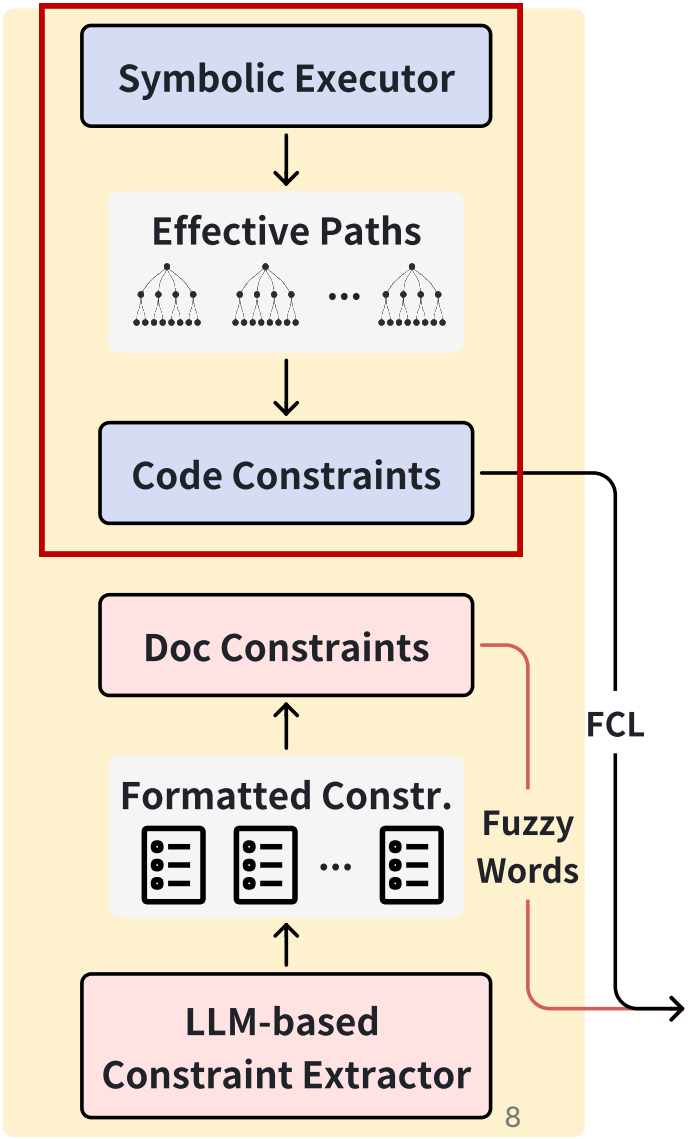


Modified source code

```
1 def fit(sample_weight, strategy, call__check_sample_weight):
2   if sample_weight != 'None' and strategy == 'uniform':
3     return '(sample_weight)_(strategy)_ERROR_END'
4   if sample_weight != 'None':
5     sample_weight = call__check_sample_weight
6   return (f'(sample_weight = {sample_weight}) ^ (call__check_sample_weight =
                    {call__check_sample_weight}) ^ (strategy = {strategy})')
```

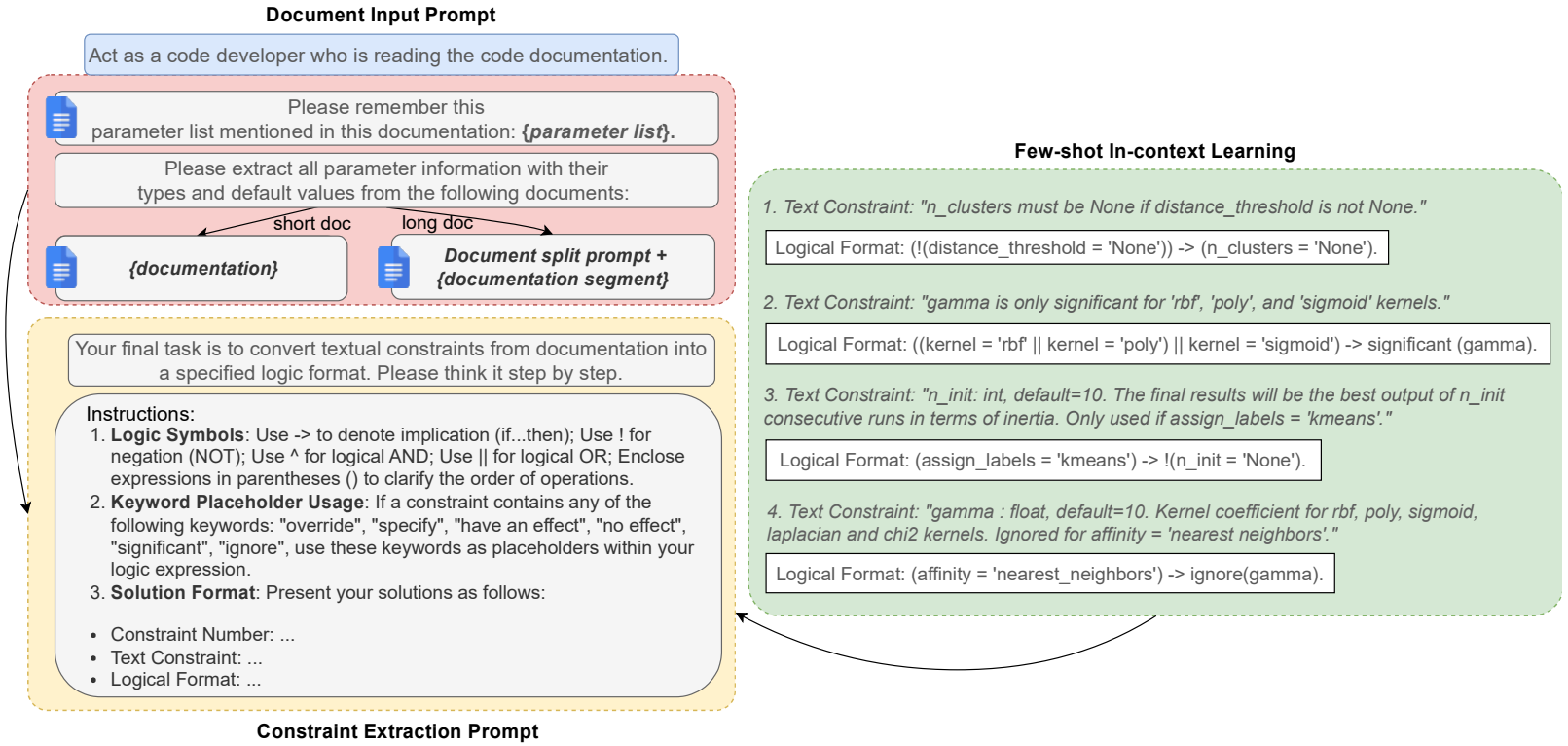
- Example code-constraint:

  `(sample_weight != None) ∧ (strategy = uniform) → ERROR`



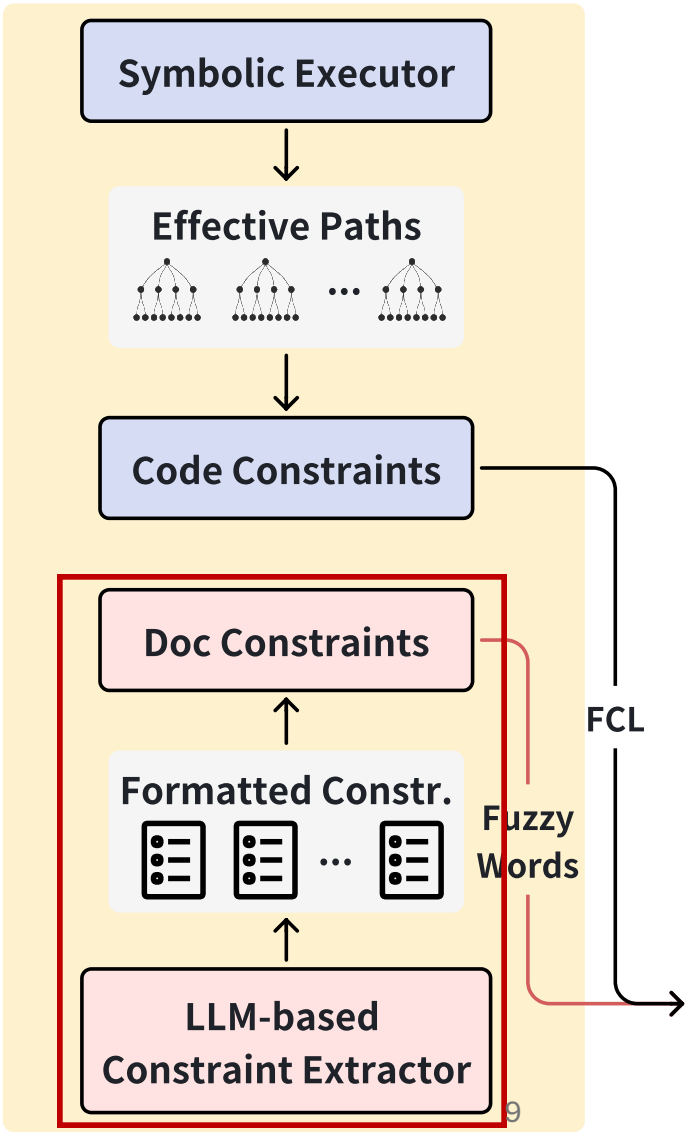
Phase II-B: Doc-constraint extraction

- Parameter list + Few-shots+ CoT



- Example doc-constraint:

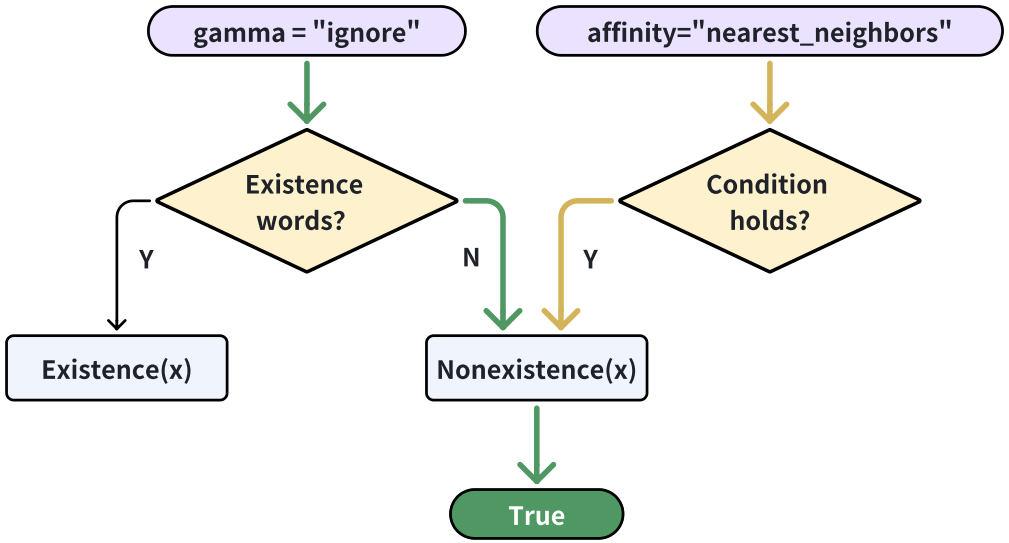
  `(affinity = nearest_neighbors) → ignore(gamma)`



Implicit constraints and fuzzy words

Existence	Non-existence
specify, have an effect, exist, significant, etc.	ignore, have no effect, unused, override, etc.

(affinity = nearest_neighbors) → ignore(gamma)



Constraint description of gamma and affinity in class SpectralClustering

> **gamma**: float, default=10
> Kernel coefficient for rbf, poly, sigmoid, laplacian and chi2 kernels. **Ignored for affinity="nearest_neighbors".**

Corresponding code snippet in class SpectralClustering

```
class SpectralClustering(ClusterMixin, BaseEstimator):
    def fit(self, X, y=None):
        if self.affinity == "nearest_neighbors":
            ...
        elif self.affinity == "precomputed_nearest_neighbors":
            ...
        elif self.affinity == "precomputed":
            ...
        else:
            params = self.kernel_params
            if params is None:
                params = {}
            if not callable(self.affinity):
                params["gamma"] = self.gamma
                params["degree"] = self.degree
                params["coef0"] = self.coef0
```

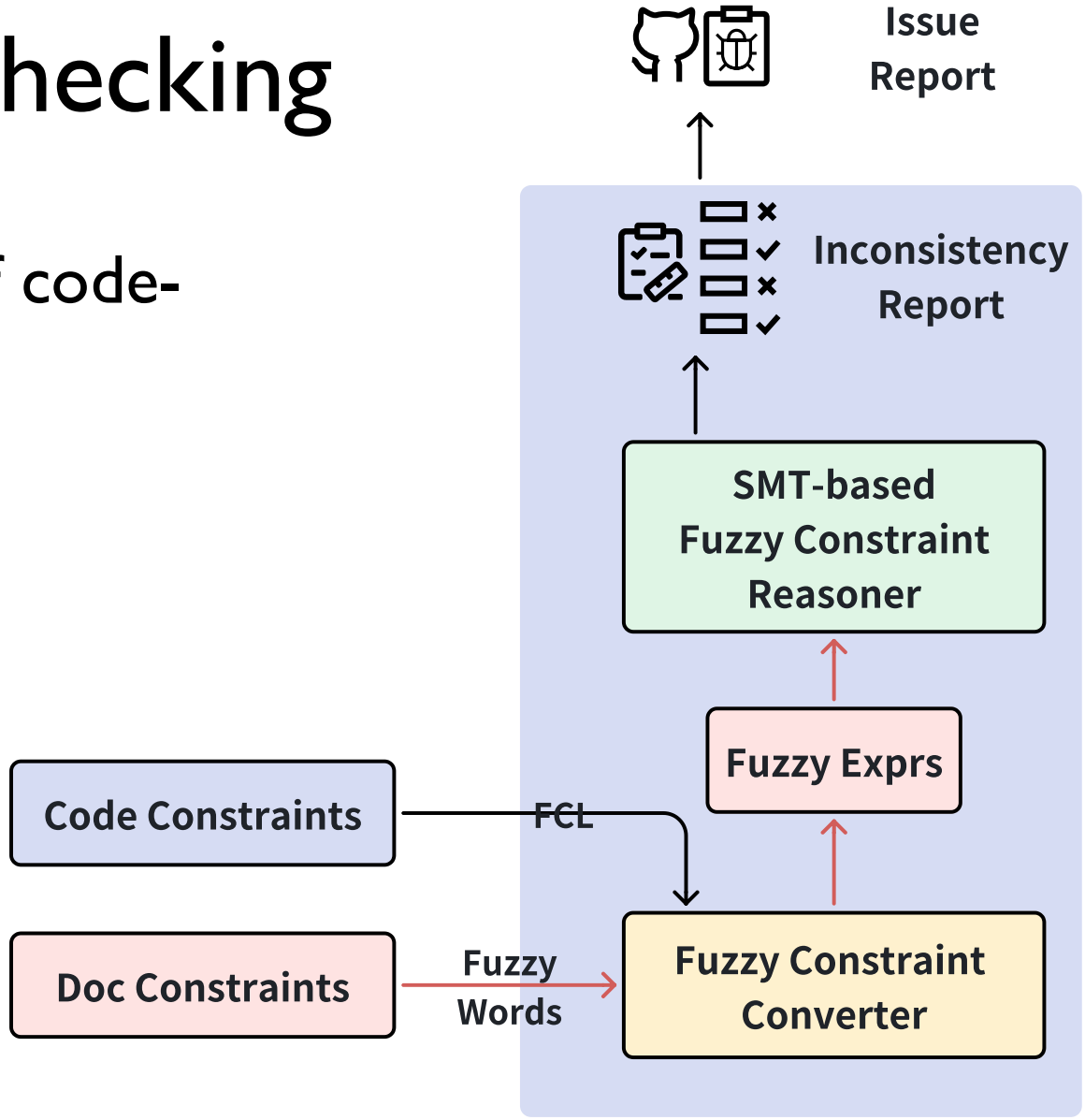
Phase III: Inconsistency checking

- Given a doc-constraint c and a set of code-constraints $P = \{p\}$
  - **Un-satisfiability:**

$$\forall p \in P, \neg(c \wedge p)$$

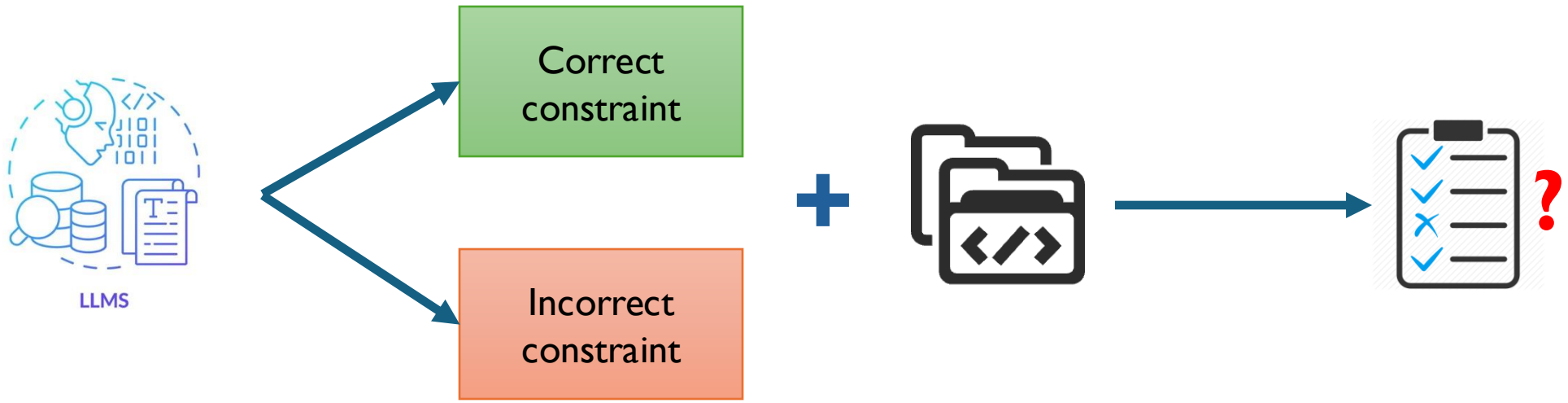
  - **Nonequivalence:**

$$\exists p \in P, \neg(c \Longleftrightarrow p)$$



Fuzzy constraint logic

- Hallucination ➔ Incorrect constraints ➔ fuzzy results
- LLM tends to make minor mistakes
  - Look-alike names, wrong symbols ($< \longrightarrow \leq$), etc.
  - Unfortunately, we can't tell whether the extracted doc-constraints are correct or not
- We could mitigate the impact of hallucination on consistency checking



FCL – Expression similarity

- Constraint in doc-constraint: `(samples < features) ∧ (dual = True)`
- Expression in code: `n_samples >= n_features, dual = True`

samples

Normalized Levenshtien Distance

$$\eta(s_1, s_2) = NLD = 1 - \frac{LD(s_1, s_2)}{max(|s_1|, |s_2|)}$$

n_samples

$$\eta = 1 - \frac{2}{9} \approx 0.78$$

<

features

>=

n_features

FCL – Expression similarity

- Constraint in doc-constraint: `(samples < features) ∧ (dual = True)`
- Expression in code: `n_samples >= n_features, dual = True`

samples → **n_samples**

> $\delta_{>} = (1,0,1,0,0)$

Cosine similarity

<= $\delta_{\leq} = (1,1,0,1,0)$

$cos\theta = 0.41$

Comparison, Equality, Greater than, Less than, Negativity

$\delta_{\bowtie} = (C, E, G, L, N)$, where $C, E, G, L, N \in \{0, 1\}$

$$cos\theta(\bowtie_1, \bowtie_2) = \frac{\delta_{\bowtie_1} \cdot \delta_{\bowtie_2}}{\|\delta_{\bowtie_1}\| \|\delta_{\bowtie_2}\|}$$

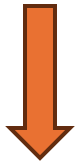
	<	>	<=	>=	=	!=
<	1	0.5	0.82	0.41	0	0
>	0.5	1	0.41	0.82	0	0
<=	0.82	0.41	1	0.67	0.58	0.41
>=	0.41	0.82	0.67	1	0.58	0.41
=	0	0	0.58	0.58	1	0.7
!=	0	0	0.41	0.41	0.7	1

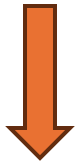
features → **n_features**

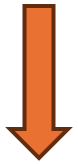
FCL – Expression similarity

- Constraint in doc-constraint: `(samples < features) ∧ (dual = True)`
- Expression in code: `n_samples >= n_features, dual = True`

samples **<** **features**







Normalized Levenshtien Distance

$$\eta(s_1, s_2) = NLD = 1 - \frac{LD(s_1, s_2)}{max(|s_1|, |s_2|)}$$

n_samples **>=** **n_features**

$$\eta = 1 - \frac{2}{10} = 0.8$$

FCL – Expression similarity

- α and β denotes the relative weights. Usually, we assign the same weights

$$\sigma(e_1, e_2) = \alpha * (1 - \frac{LD(p_1, p_2)}{\max(|p_1|, |p_2|)}) + \beta * (\frac{\delta_{\bowtie_1} \cdot \delta_{\bowtie_2}}{\|\delta_{\bowtie_1}\| \|\delta_{\bowtie_2}\|}) + \alpha * (1 - \frac{LD(v_1, v_2)}{\max(|v_1|, |v_2|)})$$

Expression 1: **samples** **<** **features**

Expression 2: **n_samples** **>=** **n_features**

$$\eta = 1 - \frac{2}{9} \approx 0.78 \qquad cos\theta = 0.41 \qquad \eta = 1 - \frac{2}{10} = 0.8$$

$$\sigma(e_1, e_2) = \frac{0.778 + 0.41 + 0.8}{3} \approx 0.66$$

FCL – Constraint similarity

- Constraint in doc-constraint: `(samples < features) ∧ (dual = True)`
- Expression in code: `n_samples >= n_features, dual = True`

Constraint similarity can be evaluated with the following formula:

$$\rho(c,\Phi) = \begin{cases} \arg\max_{e_i \in \Phi} \sigma(e, e_i), & \text{if } c \text{ is an expression } e \\ 1 - \sigma(c', \Phi), & \text{if } c = \neg c' \\ \min\{\sigma(c_1,\Phi), \sigma(c_2,\Phi)\}, & \text{if } c = c_1 \wedge c_2 \\ \max\{\sigma(c_1,\Phi), \sigma(c_2,\Phi)\}, & \text{if } c = c_1 \vee c_2 \end{cases}$$

$$\rho = \min\{0.66, 1\} = 0.66$$

FCL – Membership function

- Constraint similarity serves as the degree to which a given constraint is consistent with the code.

$$\mu_{\Omega}(\epsilon) = \rho(\epsilon, \Phi_{\Omega}) \cdot \epsilon[e \mapsto e_{\Phi_{\Omega}}]$$

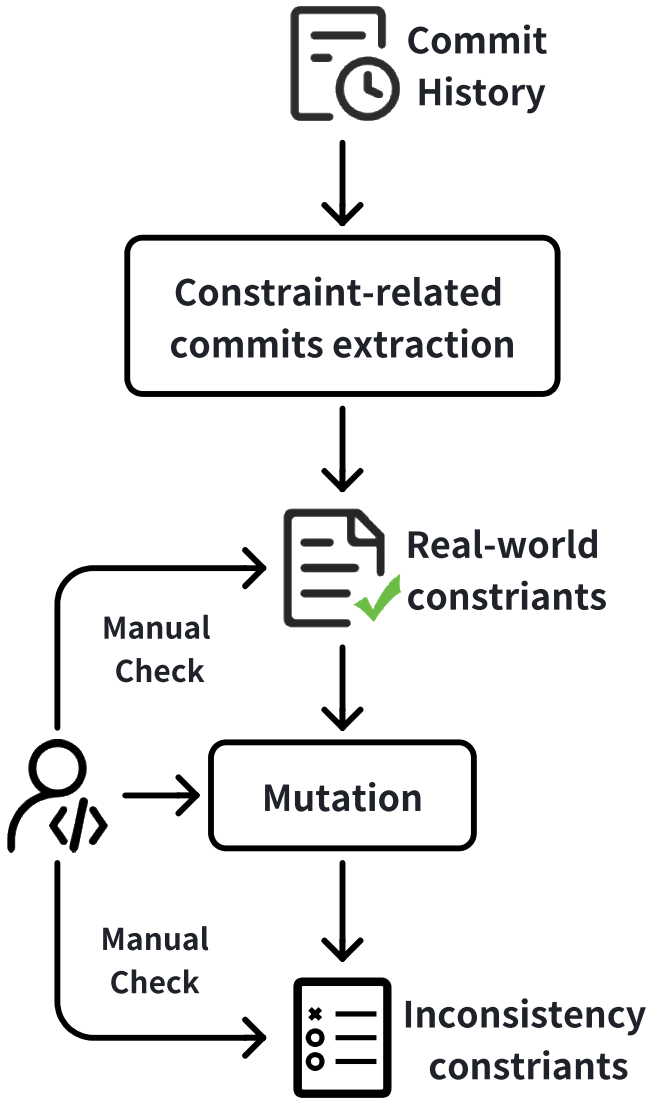
- After replacing each expression with its closest counterpart in path, the modified constraint is then evaluated by SMT solver to check whether the constraint is consistent with the code logic

$$0.66 \cdot True = 0.33 \cdot False$$

- To reduce false positives, we set a constraint similarity threshold of 0.85
  - >0.85, discard the result
  - <=0.85, accept the result

Evaluation

- Dataset[1]
  - 72 Real-world constraints from 4 popular libraries
    - scikit-learn, scipy, numpy, pandas
  - Mutation-based extended dataset (3X constraints)
    - 216 constraints (126 inconsistent + 90 consistent)
- Implementation
  - GPT-4, z3 SMT Solver
- Research Questions
  1. How accurate is MPChecker in extracting constraints from API documentation?
  2. How effective is MPChecker in detecting errors related to multi-parameter constraints in API documentation?
  3. How effective can MPChecker detect unknown inconsistency issues?



[1] https://doi.org/10.5281/zenodo.15202267

Evaluation: RQ1

- How accurate is MPChecker in extracting constraints from API documentation?

	Equivalent	Non-Equivalent		Accuracy
	Correct extraction	Incorrect extraction	Missing constraints	
MPChecker w/o few-shot learning	45	20	7	62.5%
MPChecker w/o chain-of-thought	57	7	8	79.2%
MPChecker	66	2	4	**91.7%**

  - Few-shot learning and CoT can help LLM in this task
  - 66/72=91.7% demonstrates MPChecker is effective in extracting logical constraints from doc

Evaluation: RQ2

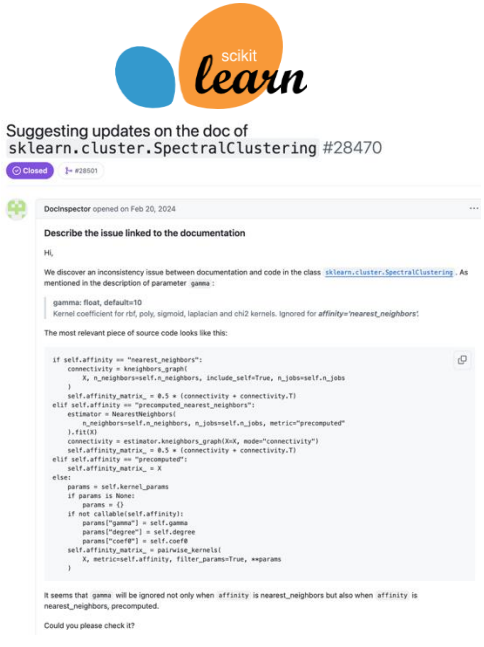
- How effective is MPChecker in detecting errors related to multi-parameter constraints in API documentation?

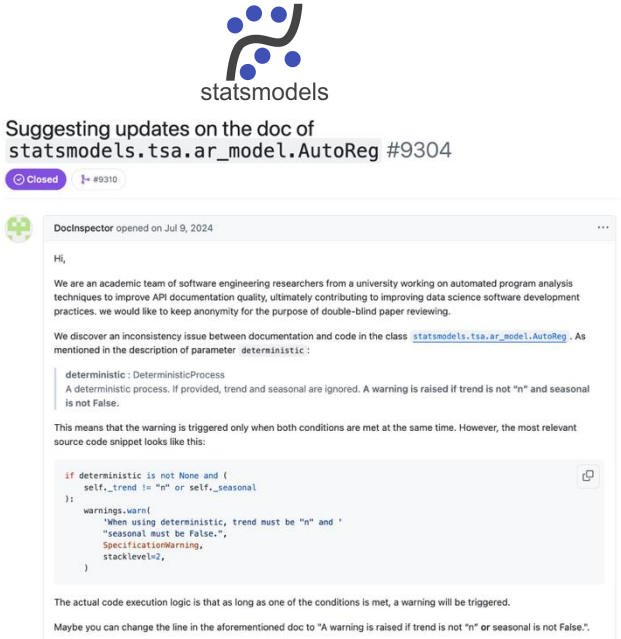
Checker		FN	TP	Recall
LLM	raw docs + code → LLM Checker	119	7	5.6%
LLM+C	doc-constrs + code → LLM Checker	74	52	41.3%
VANILLA MPCHECKER	doc-constrs + code-constrs → Z3	39	87	69.0%
FUZZY MPCHECKER	doc-constrs + code-constrs + fuzzy words + FCL → Z3	9	117	**92.8%**

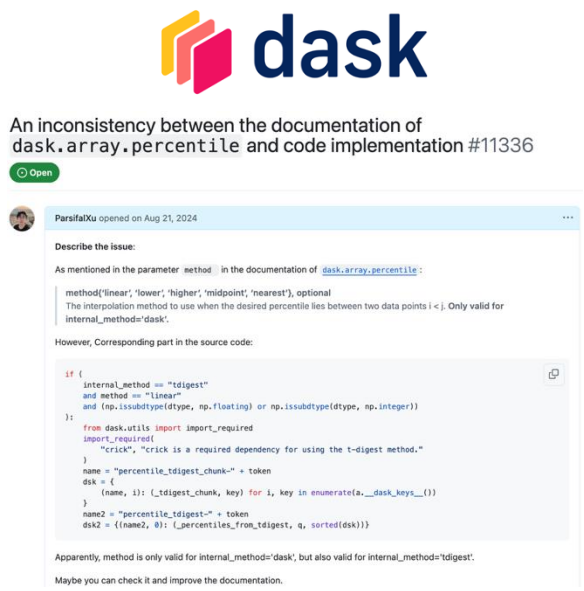
- LLM performs poorly as end-to-end inconsistency checker
- LLM Results: Correct conclusion + vague / incorrect reasons
- 117/126=92.8% shows MPChecker is effective in detecting inconsistent errors
- Fuzzy words and fuzzy constraint logic improve recall by 23.8%

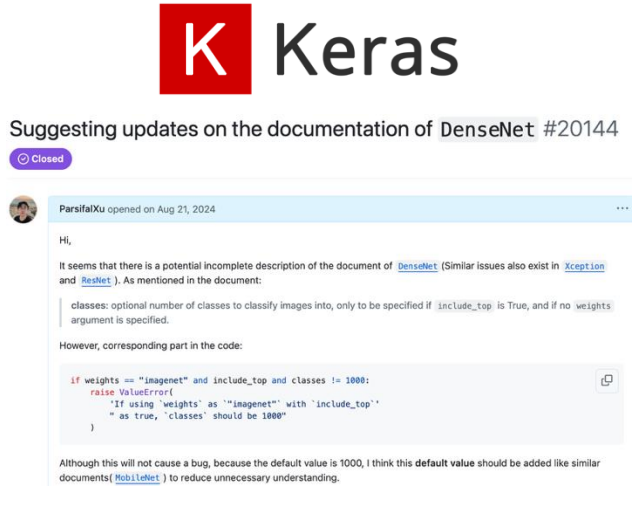
Evaluation: RQ3

- How effective can MPChecker detect unknown inconsistency issues?
  - Issue report confirmation rate: 11/14 = 78.6%
  - 10/11 have already been resolved. 7 fix doc / 3 fix doc+code
  - MPChecker can detect unknown API documentation errors
  - It works even in unseen libraries which highlights its generalization capability









Summary

10.5281/zenodo.15202267

https://github.com/ParsifalXu/MPChecker

xiufeng001@e.ntu.edu.sg

yi_li@ntu.edu.sg (Supervisor)