HotCat: Green and Effective Feature Selection toward Hotfix Bug Taxonomy

Luis de la Cal[1], Yazhuo Cao[2], Ayse Irmak Ercevik[2], Giovanni Pinna[3], Lukas Twist[2], David Williams[4], Karine Even-Mendoza[2], W.B. Langdon[4], Hector D. Menendez[2], and Federica Sarro[4]

[1] Universidad Politécnica de Madrid `l.delacal@upm.es`
[2] King's College London `{yazhuo.cao, ayse.ercevik, lukas.twist, karine.even_mendoza, hector.menendez}@kcl.ac.uk`
[3] University of Trieste `giovanni.pinna@phd.units.it`
[4] University College London `{david.williams.22, w.langdon, f.sarro}@ucl.ac.uk` (corresponding author: f.sarro@ucl.ac.uk)

Abstract. HotBugs.jar is a novel benchmark targeting time-critical (a.k.a. hot) fixes. We propose an approach to analyze the taxonomy of the bugs in HotBugs.jar by extending *PatchCat* into *HotCat*, integrating hotfix metadata with multi-objective optimization. Using NSGA-II, we evolve bitmask-based feature subsets that balance accuracy, Normalized Mutual Information (NMI), and runtime. On 88 records across 17 categories, *HotCat* achieved 0.59 accuracy and 0.58 NMI in 129 seconds, with maximum accuracy of 0.63 in 132 s, demonstrating accuracy improvements without additional resource use, thus supporting sustainability. Future work will expand and augment the dataset, refine optimization objectives, and improve semantic categorization, robustness, and cluster balance.

Keywords: Genetic Algorithm · LLM · SSBSE · Time-critical Apache Java Software Errors · Hotfix · HotBugs-dot-jar · PatchCat · Jira issue

1 Introduction

Hotfixes represent a critical category of software patches that address time-sensitive issues requiring immediate deployment to production systems [9]. These patches aim to minimize system downtime while fixing urgent vulnerabilities.

Recently, Even-Mendoza et al. [6] introduced *PatchCat*, which leverages Sentence-BERT embeddings with short-text clustering [13,12] as a lightweight machine learning approach to approximate patch edits based on descriptions synthesized by *Large Language Models* (LLMs). *PatchCat* demonstrated the feasibility of semantic-aware patch classification, capturing edit intent (e.g., "added dead code", "duplicated code") and integration with issue-tracking systems for analysis.

We extend *PatchCat* from patch clustering to classifying and organizing hotfixes into coherent semantic categories, combining clustering results with project metadata, such as Jira issue reports, to derive a taxonomy that reflects real-world maintenance practices. This taxonomy enables systematic analysis of urgent patches and opportunities for automated reasoning in software maintenance

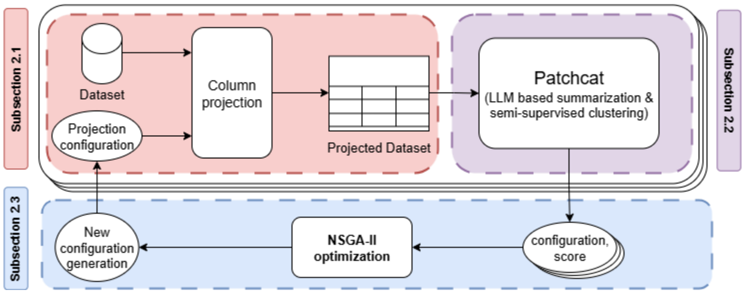

Fig. 1. Architecture diagram of *HotCat*.

while targeting challenges: 1) Hotfix data is typically sparse and imbalanced, limiting the effectiveness of traditional classification approaches [4]; and 2) balancing accuracy with the high computational cost of large-scale LLM-based analyses is crucial, motivating the integration of *Search-Based Software Engineering* (SBSE) techniques, where multi-objective optimization has proven effective in feature selection under competing objectives.

Our Contribution. We propose an automated pipeline that adapts and extends *PatchCat* [6] for hotfix analysis of the HotBugs.jar [7] SSBSE'25 Challenge Case:

- We adapt *PatchCat* to operate directly on hotfix patches from the HotBugs.jar dataset [7], integrating project metadata and LLM-based summarization.
- We address sparsity and imbalance by combining centroid-based description, LLM-generated augmentation, and semantic enrichment [4,9].
- We use NSGA-II for feature selection, balancing quality with computational efficiency, and green responsible resource usage.
- We empirically demonstrate that extending *PatchCat* with search-based optimization is a step toward an automatically extracted hotfix bug taxonomy.

Related Work. Hotfixes require immediate deployment and must balance urgency with system stability [9]. This urgency has motivated advances in semantic patch analysis, where approaches such as SemFix [10] and CapGen [15] showed that capturing edit intent yields better results than syntactic methods. In parallel, bug taxonomies have progressed from manual schemes like Orthogonal Defect Classification [3] to machine learning approaches [2]. However, a key challenge lies in the sparse and imbalanced nature of hotfix data. To tackle this, data augmentation has shown promise, improving classification accuracy to 88% in imbalanced settings [4]. Yet, despite advances with *PatchCat* [6] and *Gin* [11], the combination of semantic clustering and multi-objective optimization for hotfix taxonomy construction remains unexplored.

Availability. *HotCat* and all artifacts are available at DOI 10.5281/zenodo.17170205.

2 Methodology

Our objective is to leverage *PatchCat* [6] to classify Jira issues based on the features in the SSBSE hotfix dataset [7]. *PatchCat* [6] was originally designed to classify software patches in *Gin* : given a code diff, it generates a short natural language summary via an LLM, applies semi-supervised clustering via K-Means

Table 1. Bug taxonomy description & data sources for categories 1–17.

#	Description	Hotfix	Centroids	Le Chat	PatchCat
1	Test suite, tests, test folder	✓	✓		
2	Crash or Hang	✓	✓		
3	Missing Code or Components, or Incomplete	✓	✓	✓	14,3
4	Start, Access, or Availability of Service Issues	✓	✓		
5	Security Vulnerability or Permission Issues	✓	✓		
6	Configuration Dependency, Versioning or Deprecation	✓	✓	✓	
7	Configuration Build or CI Failures	✓	✓	✓	
8	Buggy Configuration or Broken Config Files	✓	✓		
9	Database	✓	✓	✓	
10	API / Parsing / Syntax errors	✓	✓		
11	Exceptions, Error Handling, or Missing Checks	✓✓	✓		
12	unsupported, Undefined or unspecified behavior	✓	✓	✓	18
13	Network	✓	✓		
14	Performance	✓	✓	✓	11,14
15	Permission Deprecation, Access Control or Policy Issues	✓	✓	✓	
16	Functionality issue (Logical Bugs)	✓✓✓	✓		
17	Concurrency or Race Conditions	✓	✓	✓	11

and Sentence-BERT embeddings [13] refined by iterative classification [12] to assign patches to semantic categories, and outputs accuracy and *Normalized Mutual Information* (NMI) scores, indicating clustering performance.

We adapt *PatchCat* to operate not only on patch diffs but on selected subsets of attributes from the hotfix dataset, thereby extending its scope from categorizing code-level edits to classifying hotfix issues using a richer representation of metadata. Using all features is neither sustainable nor effective. Larger prompts increase energy use and costs [14], while some features add more noise than value. We employ NSGA-II to identify feature subsets that strike a balance between predictive accuracy and efficiency. The overall process is illustrated in Figure 1: subset selection produces a projected dataset (§2.1), each projection is evaluated with *PatchCat* to assess performance (§2.2), and resulting fitness scores are fed into NSGA-II to evolve improved generations of feature subsets (§2.3).

2.1 Dataset, Projected Dataset and Configuration Definition

The HotBugs.jar dataset [7] contains **88 entries** and provides metadata per hotfix instance, like project name, Jira reference, build configuration, assigned bug category, and rationale for classifying it as a hotfix. We further enrich the dataset with code diffs and additional information, such as time-to-fix and the number of participants, gathering information from multiple sources listed in Table 1. The **enriched and augmented dataset** comprises **155 records**, each with **18 features** (i.e., bitmask vector size) across **17 categories** (Table 1). We applied two-stage **data augmentation**: 1) balancing category sizes so no category contained fewer than three records (yielding 155 records), and 2) adding 50 records per category *post-optimization* to evaluate if augmentation improves quality (yielding an additional 17x50 records, used in RQ2 only). Features span hotfix data, bug details, and code commit diffs detailed at [8].

We load the dataset and apply a *bitmask* to project onto selected features, where, per feature, 1 denotes inclusion and 0 exclusion. The projected dataset (referred to as "hotfix"), together with its *bitmask*, is then passed to *PatchCat* for fitness evaluation. In this setting, the *configuration* is the *bitmask*.

2.2 PatchCat and Fitness Function

We modify *PatchCat* in two ways: 1) extending it to incorporate additional hotfix fields, specified through the *configuration* bitmask, as opposed to code diffs alone, and 2) automating its operation within the optimization loop to serve as the NSGA-II fitness function. We then, per configuration, employ modified *PatchCat* as a black box, taking the projected dataset as input and returning category-quality metrics that directly serve as fitness values.

Hotfix projected records are passed to an LLM (via Ollama or similar interfaces) to generate short, clean summaries of each hotfix. Prompting is altered to include two prompts in sequence. The first sets the LLM's role, and the second defines the concrete task. Prompts templates are at [8]. Bug category IDs are fixed: While projections yield different summaries, labels remain constant as defined by the original hotfix dataset's expert as ground truth.

2.3 Optimization

Determining the right balance between too much and too little information is an open question in the context of hotfixes and bug reports, which we begin to investigate in this work. We define the *bitmask* pattern as a configuration in our optimization setting: each bit encodes whether a dataset field is included or excluded (1 or 0, respectively). Searching for bitmasks corresponds to exploring different projections of the dataset, with optimization aiming to identify those projections that yield the most meaningful summaries and category performance.

NSGA-II Optimization. Using NSGA-II [5,1], we evolve new generations of candidate configurations, keep those that strike the best balance across objectives, promote diversity via binary crossover and occasional bit flip mutations, and discard duplicates. Since our problem is multi-objective, we will produce an approximation to the Pareto front, representing the set of non-dominated configurations that characterize the trade-offs between classification performance and computational efficiency for hotfix analysis.

3 Evaluation

Research Questions. We ask the following research questions (RQ):

RQ1: *What is the trade-off between classification performance and runtime when classifying hotfixes using HotCat?*

RQ2: *Do our predicted bug categories improve real maintenance workflows?*

Setup. We implemented *HotCat* in Bash and Python, with performance assessed by execution time. Configurations were evaluated using PatchCat. *Optimization* used the NSGA-II algorithm [5] from `pymoo` library v0.6.1.5 [1], with a population size of 20 over 20 generations, which balances exploration and exploitation. Objectives minimized runtime and maximized accuracy (label–ground truth alignment), and NMI, yielding a Pareto front of non-dominated configurations, capturing the best trade-offs. *Post-optimization*, Pareto-front configurations train PatchCat models to classify manually labeled unclassified hotfix entries.

`indiv.`	accuracy	NMI	seconds
`1.1`	0.59	**0.58**	129
`1.2`	0.55	0.56	125
`1.3`	0.54	0.50	119
`1.4`	0.56	0.56	125
`2`	0.59	0.55	125
`3`	**0.63**	0.57	132
`4.1`	0.58	0.53	121
`4.2`	0.54	0.52	120
`5.1`	0.52	0.48	**116**
`5.2`	0.54	0.49	119
`6`	0.57	0.54	123
`7`	0.51	0.49	118
`8.1`	0.57	0.53	121
`8.2`	0.55	0.55	124
`9`	0.52	0.48	118

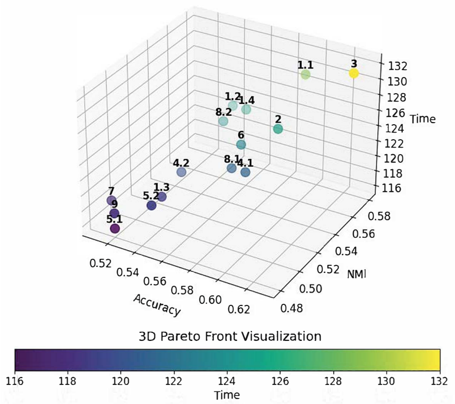

Fig. 2. Pareto Front individuals and visualization.

Results. Our optimization process focused on three specific objectives using the NSGA-II algorithm: accuracy, NMI, and execution time. It also accounts for the non-deterministic behavior of the bug categorization process, which may yield repeated solutions with different fitness values. These solutions are denoted as $X.Y$, where X is the individual number and Y is the repetition identifier. In Figure 2 (left side, table), we present the Pareto-optimal individuals obtained after running the algorithm with these objectives. Individual 3 is the slowest (132 seconds) and the fastest is 5.1 (116 s). Individual 3 achieves the highest accuracy (0.63), whereas individual 7 records the lowest (0.51). The best NMI value is obtained by individual 1.1 (0.58), while the lowest is found in individual 5.1 (0.48). Individual 6 combines solid accuracy (0.57) and NMI (0.54), with a competitive runtime (123 s), making it a reasonable candidate. The most *balanced solution* is individual 1.1.

RQ1 Answer. The best trade-off (individual 1.1) identified by NSGA-II achieved high accuracy (0.59), the best NMI (0.58), and a runtime close to the median (129 s), representing a strong balance across the three objectives.

To evaluate generalization, we manually annotated 51 challenging bug reports that were tagged in the dataset but excluded from `HotBugs.jar` [7] to test our Pareto-optimal models with five repeats per configuration, obtaining average accuracy/NMI of 0.55/0.52 on the original dataset and 0.72/0.75 on the augmented dataset. Each model was provided with the corresponding Jira ticket (including description and metadata), and final labels were assigned by a majority vote across the ensemble. The ensemble underperformed relative to training results, likely due to the small, imbalanced training set (88 records across 17 classes) and vocabulary overlap that blurs fine-grained boundaries, tending to collapse predictions into a few dominant categories (notably 13 and 16).

RQ2 Answer. On 51 specially hard hotfixes, the ensemble underperformed. Labels collapsed into a few classes due to a tiny, imbalanced training dataset of 88 records across 17 categories and a strong vocabulary overlap. Improvements are likely with data augmentation techniques and class balancing.

4 Conclusion

We propose an initial taxonomy for classifying hot fixes. We showed that multi-objective optimization improves accuracy while maintaining sustainability: the best-balanced solution (1.1) reached 0.59 accuracy and 0.58 NMI at 129 seconds runtime, while the maximum accuracy (0.63) required only slightly longer runtime (132 seconds). These results demonstrate that higher accuracy can be achieved without increasing resource use.

During optimization, we used runtime as a proxy for efficiency and measured energy and emissions post hoc; future work should make these first-class objectives and mitigate LLM run-to-run variance. Our results are limited by a small, imbalanced dataset (88 records across 17 categories) and overlapping vocabularies that blur class boundaries (e.g., memory leaks vs. performance vs. concurrency), which led to overprediction of a few bug categories (13, 16). We also evaluated on harder, previously excluded records. Expanding the dataset and refining the objectives should improve robustness, avoid local optima, and yield better accuracy-efficiency trade-offs.

References

1. Blank, J., Deb, K.: Pymoo: Multi-objective optimization in python. IEEE access **8**, 89497–89509 (2020)
2. Catolino, G., et al.: Not all bugs are the same: Understanding, characterizing, and classifying bug types. J. Syst. Softw. **152**, 165–181 (2019)
3. Chillarege, R., et al.: Orthogonal defect classification-a concept for in-process measurements. IEEE TSE **18**(11), 943–956 (1992)
4. Ciborowska, A., Damevski, K.: Too few bug reports? exploring data augmentation for improved changeset-based bug localization. arXiv:2305.16430 (2023)
5. Deb, K., Pratap, A., Agarwal, S., Meyarivan, T.: A fast and elitist multiobjective genetic algorithm: Nsga-ii. IEEE TEVC **6**(2), 182–197 (2002)
6. Even-Mendoza, K., et al.: LLM-guided genetic improvement: Envisioning semantic aware automated software evolution. In: ASE NIER (2025)
7. Hanna, C., Sarro, F., Harman, M., Petke, J.: Hotbugs.jar: A benchmark of hot fixes for time-critical bugs (2025), `https://arxiv.org/abs/2510.07529`
8. HotCat: The paper's artifact (2025). `https://doi.org/10.5281/zenodo.17170205`
9. Islam, C., Prokhorenko, V., Babar, M.A.: Runtime software patching: Taxonomy, survey and future directions. J. Syst. Softw. **200**, 111652 (2023)
10. Nguyen, H.D.T., Qi, D., Roychoudhury, A., Chandra, S.: Semfix: Program repair via semantic analysis. In: ICSE. pp. 772–781. IEEE (2013)
11. Petke, J., et al.: Program trans... using Gin. ESE **28**, article no: 104 (2023)
12. Rakib, M.R.H., Zeh, N., Jankowska, M., Milios, E.: Enhancement of short text clustering by iterative classification. In: NLDB 2020. pp. 105–117. Springer (2020)
13. Reimers, N., Gurevych, I.: Sentence-BERT: Sentence embeddings using siamese BERT-networks (2019), `https://arxiv.org/abs/1908.10084`
14. Robinson, J., Kummerfeld, J.K.: Simple and effective baselines for code summarisation evaluation. arXiv:2505.19392 (2025)
15. Wen, M., Chen, J., Wu, R., Hao, D., Cheung, S.C.: Context-aware patch generation for better automated program repair. In: ICSE (2018)