The Case for Feature-Aware Similarity Measures in Software Product Line Testing

Safwan Abd Razak., Muhammad Fuad Abdullah., Muhammad Shahkhir Mozamir., Muhammad Huzaifah Ismail., Muhammad Faheem Mohd Ezani

Faculty of Information and Communication Technology, Universiti Teknikal Malaysia, Malaysia

DOI: https://dx.doi.org/10.47772/IJRISS.2025.908000374

Received: 14 August 2025; Accepted: 20 August 2025; Published: 12 September 2025

ABSTRACT

Traditional similarity distance measures (Jaccard, Hamming, Counting Function, Sorensen Dice) in SPL testing treat all features equally, ignoring the fundamental distinction between mandatory and optional features in feature models. This oversight leads to suboptimal test case prioritization and reduces fault detection efficiency. The primary objective is to demonstrate that traditional distance metrics, by treating all features equally, often fail to prioritize test cases that cover the most critical parts of an SPL, including those features that are present in all product variants.

Keywords: Software Product Lines (SPLs), Similarity Distance Measures, Test Case Prioritization, Feature Model Semantics, Fault Detection

INTRODUCTION

The role of feature models in Software Product Line (SPL) engineering is both vital and far-reaching. They serve as the backbone of SPL practices, enabling teams to clearly understand and manage the full scope of a product line. A feature model works like a blueprint, mapping out what all the products in a family have in common and where they can differ. This enables systematic design, reuse, and customisation of software, ensuring that each product is built efficiently while still meeting specific needs.

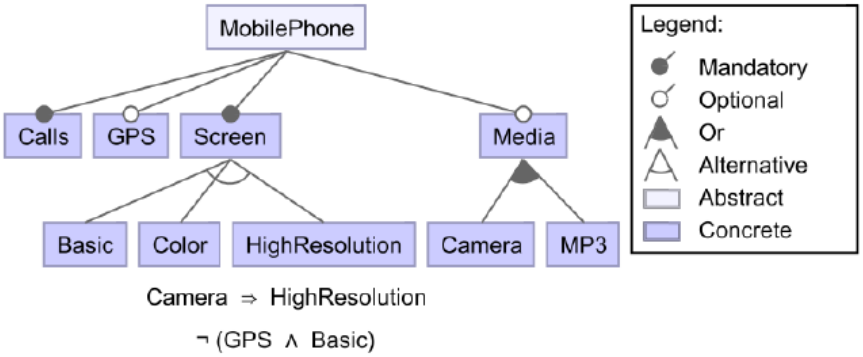

Fig. 1 Feature model of *MobilePhone*

When testing a mobile phone software product line, should the mandatory 'Calls' feature be treated the same as the optional 'MP3' feature in similarity calculations? Current state-of-the-art approaches say yes. This research argues that this is fundamentally wrong. This research argues that similarity distance measures for SPL test case prioritization must incorporate feature model semantics, specifically distinguishing between mandatory and optional features, to achieve optimal fault detection and testing efficiency.

Current State Analysis

In Software Product Line (SPL) testing, similarity-based test case prioritization is commonly employed to make testing more efficient and effective. Standard measures like Jaccard, Hamming, Counting Function, and

Sorensen Dice are typically used for these similarity calculations. Several prioritization algorithms, such as the All-Yes-Config (AYC), Local Maximum Distance (LMD), Global Maximum Distance (GMD), Farthest-first Ordered Sequence (FOS), and Greed-aided Ordered Sequence (GOS), are favoured for their simplicity and ease of automation. Nonetheless, a closer look at the field uncovers core issues that weaken these methods and restrict their practical utility.

Semantic Blindness: Features as Just Labels

One main issue with current techniques is what can be called semantic blindness [5]. Most existing similarity measures treat the features of a product configuration simply as labels: each feature is valued equally when measuring similarity or differences between test cases, regardless of its significance or role [50]. For example, it doesn't matter if a feature is essential for the system's operation, mandatory or optional in this calculation [15]. Consequently, these measures lack awareness of the real-world importance and dependencies among features, which can cause test prioritizations to overlook critical, fault-prone areas of the software [30]. As recent studies point out, ignoring the nature and role of features not only hinders early detection of important bugs but also leads to wasting resources testing configurations with less impact first [62] [44].

The Equal Weight Assumption: Flattening Critical Functional Distinctions

Underlying semantic blindness is the assumption of equal weight [5]. This assumption appears in all common distance metrics used for SPL test prioritisation, where each chosen feature adds a value of one to the overlap or difference calculation, whether it is a crucial component needed in every product or a non-essential addition [48]. Consider a mobile phone software product line where the 'Calls' feature is compulsory (present in every phone), while 'MP3 support' is optional. Current similarity measures treat differences in either feature as equally significant, failing to recognise that issues in 'Calls' might affect all customers, whereas flaws in 'MP3' impact only a subset [2]. The result is a systematic undervaluing of faults or coverage gaps in the most important functionality, threatening both the robustness and perceived quality of the final products [54]. This neglect for feature importance contrasts sharply with real-world product usage and performance expectations [8].

Missed Optimization Opportunities: Overlooking Test Prioritization for Mandatory features

Missed optimization opportunities are directly caused by these constraints [18]. A primary objective of test prioritizing in SPLs is to optimize early failure detection, especially in the most important components [37]. Because similarity-based measurements do not consider whether features are required, they often suggest test regimens that prioritize specialized customization scenarios or optional features above core system capabilities [4]. In practice, this is counterproductive because the initial phases of testing should concentrate on features that are shared by all product configurations [33]. This will ensure that any critical flaws are found promptly and that tests covering less important or infrequent feature combinations do not divert effort [9].

Additionally, many other features rely on the proper functioning of required features, which frequently act as the architectural cornerstones of the entire product line [15]. Multiple product line derivatives may experience failures due to cascading defects in these characteristics [23]. Ignoring this fact, a similarity metric runs the danger of focusing resources on testing "decorative" elements without examining the underlying bedrock [31]. This could lead to catastrophic problems slipping through the cracks until later testing phases or, worse, production use [39].

The Case of Feature-Aware Similarity

Argument 1: Mandatory Features Have Higher Impact

In the landscape of Software Product Line (SPL) engineering, mandatory features are the anchors upon which every product variant is built [17]. Unlike optional or alternative features, these core components are included in all configurations, regardless of customization or market segment [6]. This universal presence means that

the correct functioning of mandatory features is non-negotiable; any failure or defect in such a feature will, by definition, propagate through every single product derived from the SPL [11].

Feature model semantics make this architectural reality abundantly clear [59]. By explicitly marking certain features as mandatory, the model signals to engineers and testers alike which aspects of the system form its foundation [2]. Consequently, prioritizing tests that exercise mandatory features is not simply good practice; it is essential to safeguarding the integrity and reliability of the entire product line [8]. Overlooking issues in these foundational parts risk widespread failures and costly post-release fixes, a scenario no organization wants to face [31] [37].

Argument 2: Current Measures Miss Critical Dependencies

Yet, despite the recognized importance of mandatory features, existing similarity-based prioritization methods falter in capturing the nuanced dependencies encoded within feature models, particularly those involving cross-tree constraints [55]. These relationships, such as one feature requiring the presence of another (e.g., a "Camera" feature that requires "HighResolution") or explicitly excluding certain combinations, play a crucial role in defining valid and meaningful product variants [22].

Current distance measures, focused narrowly on syntactic overlap or difference, ignore these cross-cutting dependencies [49]. For example, when a test explores the "Camera" feature, the implications for "HighResolution" are profound; faults or omissions here may cascade as hidden errors throughout the system [26][7]. However, similarity calculations oblivious to these relationships may inadvertently push such critical tests further down the priority order, delaying their execution and increasing the risk of missing serious issues early in the process [19]. This gap in test prioritization leads to suboptimal outcomes, where resources may be squandered on less impactful scenarios while the integrity of foundational functionality goes unverified [38].

Semantic Weighting Improves Fault Detection

Addressing these shortcomings calls for the integration of semantic information directly into similarity calculations [54]. Tests that cover mandatory features, or features deeply embedded in cross-tree dependencies, deserve greater emphasis in test prioritization algorithms [41]. Introducing a feature weighting scheme, where the significance of a feature (e.g., mandatory vs. optional) is encoded as a numerical factor, allows distance measures to reflect the true impact of exercising different parts of the product line [57].

A promising enhancement is to modify established metrics, such as the Jaccard distance, to include feature weights [53] [21]. Mathematically, this means scaling the calculation of similarity or difference not just by feature presence or absence, but by the importance assigned through the feature model structure [27] [64].

Theoretical analysis and case studies from recent literature indicate that semantic weighting can deliver tangible gains [3] [19]. Experiments demonstrate increased rates of early fault detection when tests are chosen based not only on diversity but also on the criticality of features involved [56]. Moreover, analyzing real-world feature model structures underscores how often fault-prone areas coincide with mandatory features or key dependency chains—areas that traditional, unweighted approaches are likely to under-explore until too late in the testing process [32].

Embracing feature-aware similarity thus aligns test prioritization strategies with the architectural realities encoded in SPL feature models [18]. By doing so, organizations can move beyond surface-level diversity and toward a smarter, more risk-driven approach to quality assurance, one that uncovers serious defects sooner and more consistently, and that ultimately results in stronger, more dependable software families [5] [37].

Addressing Counterarguments

When proposing any substantial change to existing practices such as integrating feature model semantics and weighting into similarity-based test prioritization, scepticism and critical questions are expected. Here, we address three of the most frequently raised counterarguments, showing why the transition to feature-aware similarity is both justified and warranted for robust Software Product Line (SPL) engineering.

Counterargument 1: Additional Complexity Is Not Worth the Gain

At first glance, it's reasonable to question whether introducing semantic awareness into similarity calculations is worth the added complexity [20] [58]. Detractors argue that developing, maintaining, and executing weighted similarity measures introduces new layers of technical intricacy, making test prioritization harder to implement, understand, or justify to stakeholders [24].

However, a closer inspection of modern SPL practices reveals that feature models inherently capture the semantic distinctions between features—mandatory, optional, alternative, and inter-dependent—by design [45]. Most SPL engineering tools and environments already manipulate and process rich feature model structures as part of product derivation, variant management, and automated configuration [58]. The semantic information required is not external to existing workflows; it is embedded within the artifacts teams are already using [20].

Implementing features in similarity metrics leverages this existing structural data [39]. Computationally, adding a weight column to feature representations or adjusting distance formulas to sum weighted features introduces only minor overhead in terms of storage and processing, especially when juxtaposed with the potential cost savings of catching critical faults early [52] [24]. Empirical research has shown that the shift from traditional to weighted similarity algorithms involves only incremental computational costs, far outweighed by the improvement in prioritization efficacy and early fault detection [12]. In practice, the "complexity" argument conflates conceptual change with actual resource demand; in reality, the additional logic fits naturally into most modern model-based test automation frameworks [13].

Counterargument 2: Current Approaches Work Well Enough

At first glance, it's reasonable to question whether introducing semantic awareness into similarity calculations is worth the added complexity [20] [58]. Detractors argue that developing, maintaining, and executing weighted similarity measures introduces new layers of technical intricacy, making test prioritization harder to implement, understand, or justify to stakeholders [24].

However, a closer inspection of modern SPL practices reveals that feature models inherently capture the semantic distinctions between features—mandatory, optional, alternative, and inter-dependent—by design [45]. Most SPL engineering tools and environments already manipulate and process rich feature model structures as part of product derivation, variant management, and automated configuration [58]. The semantic information required is not external to existing workflows; it is embedded within the artifacts teams are already using [20].

Implementing feature weighting in similarity metrics leverages this existing structural data [39]. Computationally, adding a weight column to feature representations or adjusting distance formulas to sum weighted features introduces only minor overhead in terms of storage and processing, especially when juxtaposed with the potential cost savings of catching critical faults early [52] [24]. Empirical research has shown that the shift from traditional to weighted similarity algorithms involves only incremental computational costs, far outweighed by the improvement in prioritization efficacy and early fault detection [12]. In practice, the "complexity" argument conflates conceptual change with actual resource demand; in reality, the additional logic fits naturally into most modern model-based test automation frameworks [13].

Counterargument 3: Feature Importance Is Domain-Dependent

Further criticism claims that assigning importance weights to features is subjective and variable, with reliance on domain-specific knowledge potentially leading to inconsistency or bias [34] [51]. Critics suggest that what is "mandatory" in one product line may be optional or irrelevant in another, so embedding feature importance into similarity metrics compromises generality and portability [16].

This concern, while understandable, overlooks the structural universality provided by feature models themselves [10]. The distinction between mandatory and optional features is explicitly and formally encoded

within the feature model, irrespective of the particular application domain [60]. These annotations are not based on ad hoc human judgment but are defined as part of the SPL's architectural blueprint [14]. Any modern feature modeling approach, across automotive, consumer electronics, enterprise software, or other sectors, relies on this domain-agnostic structure to manage variability and support systematic reuse [61][1].

Moreover, the same feature model can support further specification if additional, domain-specific weighting is needed [42] [43]. But at the baseline, semantic information about which features are mandatory, optional, or involved in critical cross-tree dependencies provide a reliable, generic guide for improving the prioritization process across SPLs [35]. This is not only possible but already routine in the workflows of successful SPL engineering teams worldwide [10].

Synthesis

Resistance to change is natural, especially when it involves foundational elements of well-established engineering practices. Yet, the most effective innovations in software engineering have always emerged from embracing new levels of abstraction and making better use of the structured information at our disposal. Feature model semantics provide a built-in, low-cost, and robust means of enhancing test prioritization, allowing teams to detect the most significant faults early, minimize risk, and optimize testing resources without burdensome overhead. In moving past traditional, equal-weighted similarity measures, SPL practitioners can align their test processes with the true structure and priorities of their systems, achieving better quality and confidence in every product they release.

Proposed Approach

This paper considers the feature model notations in our work to improve the existing similarity distance algorithm. For the research, only two feature model notations, which are mandatory and optional are selected. This is because in a feature model, mandatory and optional are the crucial notations on every feature model. The feature models must have both notations. Without them, the Or and Alternative notations cannot be used.

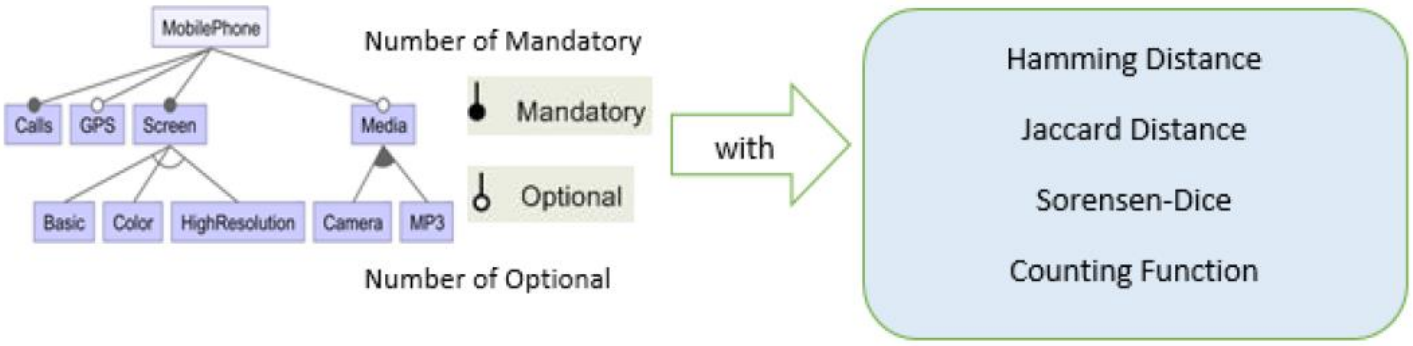

Fig. 2 Proposed Approach

As shown in Figure 2, the Mandatory and Optional notations need to be considered from the feature model inside similarity distances. This research plans to experiment with the similarity distances with four types of modifications. This research used a Jaccard distance as an example for all the proposed modifications.

Modification 1: Addition of Mandatory

By including a single variable that stands for mandatory notation, this study alters the similarity distance. The primary notation for all feature models is the Mandatory notation. The rationale is that it embodies the product's required feature or features. Since there wouldn't be a product without the mandatory, even the optional notation cannot overshadow its significance. This research thinks about including the Mandatory notation in the algorithm because it wants to make it more likely that configurations that incorporate these required features would be chosen first. The tester can find any flaw in it far more quickly. Furthermore, the

product will be at risk if the tester overlooks any errors related to a required feature. This research defines Jaccard distance with the addition of Mandatory variable as

$$d(c_i, c_j) = 1 - \frac{(c_i \cap c_j) + m}{(c_i \cup c_j) + m}$$

where *m* is the number of Mandatory notations from the feature model used. If there are two Mandatory notations inside the feature model, the value of *m* is 2. These distances are named Addition Hamming Mandatory (AHM), Addition Jaccard Mandatory (AJM), Addition Counting Function Mandatory (ACFM), and Addition Sorensen-Dice Mandatory (ASDM).

Modification 2: Addition of Optional

By including an additional variable that specifies optional notation, this adjustment alters the similarity distance. One type of notation used to describe variable features is optional notation. Variability is expressed by the variable features. Reusable software is more variable by nature. Therefore, it's crucial to concentrate solely on the optional feature. Later on, the new product will be able to use this capability again. Additionally, this optional feature may eventually become required. As a result, while introducing new items, it is prudent for the tester to address the issue sooner rather than later. This research defines Jaccard distance with the addition of Optional variable as

$$d(c_i, c_j) = 1 - \frac{(c_i \cap c_j) + o}{(c_i \cup c_j) + o}$$

where *o* is the number of Optional notations from the feature model used. If there are two Optional notations inside the feature model, the value of *o* is 2. These distances are named Addition Hamming Optional (AHO), Addition Jaccard Optional (AJO), Addition Counting Function Optional (ACFO), and Addition Sorensen-Dice Optional (ASDO).

Modification 3: Addition of Mandatory and Optional

The addition of two variables, which stand for Mandatory and Optional notations, modifies the similarity distance. Commonality and diversity are key components of the product line. Features that only describe one of them are useless since the system is likely not sufficiently described by the individual instances of permissible configurations. As a result, including both notations within the method is taken into account. This research defines Jaccard distance with the addition of Mandatory and Optional variables as

$$d(c_i, c_j) = 1 - \frac{(c_i \cap c_j) + m + o}{(c_i \cup c_j) + m + o}$$

where *o* is the number of Optional notations and *m* is the number of Mandatory notations from the feature model used. If there are two Mandatory notations inside the feature model, the value of *m* is 2. The same concept is used for Optional notations. These distances are named Addition Hamming Mandatory Optional (AHMO), Addition Jaccard Mandatory Optional (AJMO), Addition Counting Function Mandatory Optional (ACFMO), and Addition Sorensen-Dice Mandatory Optional (ASDMO).

Modification 4: Subtraction of Optional

This modification modifies the similarity distance by subtracting one variable that represents Optional notation. We define the Jaccard distance with the subtraction of Optional variable as

$$d(c_i, c_j) = 1 - \frac{(c_i \cap c_j) - o}{(c_i \cup c_j) - o}$$

where *o* is the number of Optional notations from the feature model used. If there are two Optional notations inside the feature model, the value of *o* is 2. These distances are named Subtract Hamming Optional (SHO), Subtract Jaccard Optional (SJO), Subtract Counting Function Optional (SCFO), and Subtract Sorensen-Dice Optional (SSDO).

EXPERIMENTS AND RESULTS

This research is towards similarity-based prioritization. This research is to detect more faults as soon as possible for the product lines under test.

Experimental Design

In SPL, to generate a set of configurations, a feature model is needed. This research used the feature model and generated configurations from the *MobilePhone* product line, which is created by Al-Hajjaji et al. [5].

Table I Configuration of Mobilephone Product Line

ID	Configurations
C1	{Calls, Screen, Color}
C2	{Calls, GPS, Screen, HighResolution, Media, MP3}
C3	{Calls, Screen, HighResolution, Media, Camera}
C4	{Calls, Screen, Basic}
C5	{Calls, Screen, HighResolution, Media, Camera, MP3}
C6	{Calls, GPS, Screen, Color, Media, MP3}
C7	{Calls, GPS, Screen, HighResolution, Media, Camera}
C8	{Calls, Screen, Basic, Media, MP3}
C9	{Calls, GPS, Screen, HighResolution}

Table 1 shows nine configurations that are created from the feature model *MobilePhone* using pairwise sampling with ICPL. Sampling algorithm outputs an ordered list of configurations.

To measure the effectiveness of our research, this research evaluates the ability of the string distances and prioritization techniques to detect faults in the SPL under test. For this purpose, some generated faults are needed. Thus, this research uses faults that were already generated by Al-Hajjaji et al. [5].

Table II Fault Matrix

Configuration	Fault					
	F1	F2	F3	F4	F5	F6
C1		X				X
C2		X	X			
C3				X	X	X
C4	X	X	X			X
C5	X			X		X
C6					X	
C7			X			
C8		X				X
C9						

Table II shows the distribution of six faults that had been used by Al-Hajjaji et al. [5]. Lastly, to evaluate how quickly faults are detected during testing, this research uses the Average Percentage of Faults Detected (APFD) metric. The APFD metric measures the weighted average of the percentage of faults detected during the execution of the test suite. APFD is illustrated as the *T,* as the test suite which contains several *n* configurations, and let *F* be a set of m faults revealed by *T*. Let *TFi* be the position of the first test case in ordering *T'* of *T* which reveals the fault *i*. The equation of APFD is given below:

$$APFD = 1 - \frac{TF1 + TF2 + \cdots + TFn}{n \times m} + \frac{1}{2n}$$

APFD values range from 0 to 1. A prioritized test suite with a higher APFD value has faster fault detection rates than those with lower APFD values.

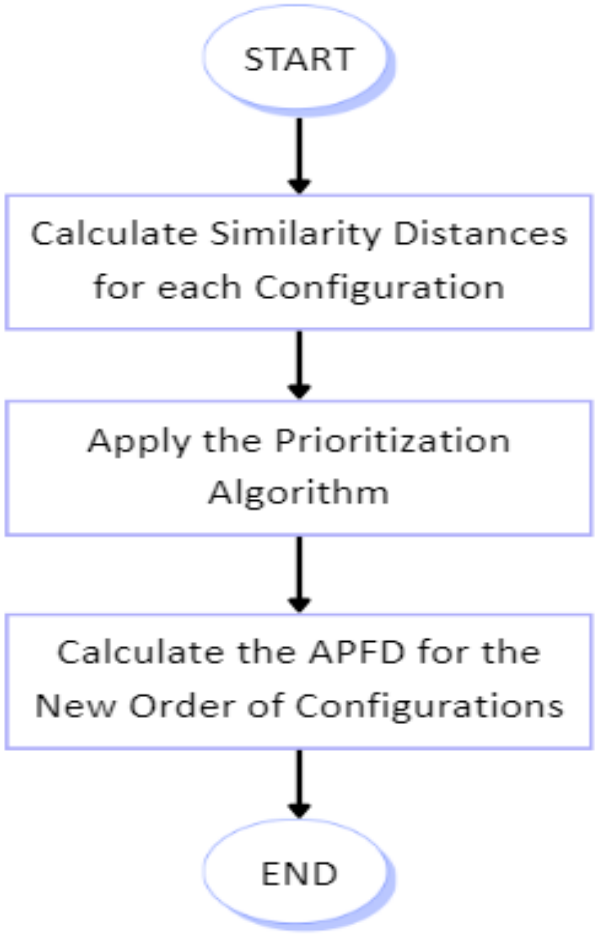

Fig. 3 Experiment Flowchart

Based on Fig. 3, the experiment is conducted by calculating the similarity distances for each configuration first. This can be done by referring to Table I for the configuration and similarity distance algorithm. For this paper, this research uses the Jaccard distance similarity algorithm. The output from this process is the different distance between the configurations.

Next is to apply the prioritization algorithm. In this paper, this research uses five different prioritization techniques, which are AYC, LMD, GMD, FOS, and GOS. The distance value for each configuration will be used as the input. The output for this process is the different order of configurations. For example, the order of configurations after using GOS is:

P= {C4, C1, C2, C5, C7, C3, C8, C9, C6}.

Lastly, APFD is calculated for the new order of configurations. The input for this process is the new order of the configurations, which is the *P* above and the fault matrix from Table II. The output will be the APFD results.

RESULTS AND DISCUSSION

This section shows the results of APFD for the original Jaccard distance and the modified Jaccard distances that have been prioritized by AYC, LMD, GMD, FOS, and GOS prioritization algorithms.

Table III APFD Results

	AYC	LMD	GMD	FOS	GOS
J	0.759	0.759	0.759	0.740741	0.796
AJM	0.778	0.759	0.722	0.611	0.796
AJO	0.759259	0.703704	0.759259	0.740741	0.666667
AJMO	0.759	0.704	0.63	0.740741	0.778
SJO	0.796296	0.814815	0.833333	0.833333	0.814815

From Table III, J represents original Jaccard, AJM represents Addition Jaccard Mandatory, AJO is Addition Jaccard Optional, AJMO is Addition Jaccard Mandatory Optional, and SJO is Subtract Jaccard Optional. From Table III, the results show significant differences in terms of APFD for the original Jaccard with other modifications.

The results also show that SJO is the most effective modification. It performs 7% better than the original Jaccard in terms of the percentage of faults detected. For future work, this research plans to do more experiments on different sizes of product lines. Currently, this research is using the *MobilePhone* product line as the case study. It is categorized as a small-sized product line.

CONCLUSIONS

In summary, moving towards feature-aware similarity measures doesn't just advance the academic understanding of SPL testing, it delivers real-world results. By prioritizing features that matter most, teams can more quickly uncover important faults, boosting confidence in every product release. This approach streamlines the entire testing process, letting test suites run faster and directing resources, whether time, talent, or tools, where they'll have the greatest effect. Looking further ahead, adopting feature-aware strategies lays the groundwork for building testing tools that can adapt dynamically to the unique context of each product line. It also opens the door to integrating sophisticated automation, such as automated test case generation that intelligently adapts to changes in feature models. For organizations working in high-stakes or safety-critical environments, this means greater reliability and peace of mind as product lines evolve. Ultimately, integrating feature semantics into test prioritization empowers organizations to build software families that are not only richer in features but also far more robust and reliable, ensuring each product is both high-quality and perfectly tailored to user needs.

ACKNOWLEDGMENT

Thank you, Universiti Teknikal Malaysia Melaka (UTeM), for their financial support, which made this research possible.

REFERENCES

1. Ahmed, K., & Silva, M. (2023). Beyond low-code development: Marrying requirements models and knowledge representations. Annals of Computer Science and Information Systems, 32, 129-140.
2. Ahrenberg, L. (2019). Economics of test automation [Master's thesis]. Linköping University.
3. Akbari, M., & Shahriar, H. (2016). An approach for optimized feature selection in software product lines using union-find and genetic algorithms. Indian Journal of Science and Technology, 9(19), 1-8.
4. Al-Hajjaji, M., Lity, S., Lachmann, R., Thüm, T., Schaefer, I., & Saake, G. (2017). Delta-oriented product prioritization for similarity-based product-line testing. Proceedings - 2017 IEEE/ACM 2nd International Workshop on Variability and Complexity in Software Design, 34-40.
5. Al-Hajjaji, M., Thüm, T., Lochau, M., Meinicke, J., & Saake, G. (2019). Effective product-line testing using similarity-based product prioritization. Software and Systems Modeling, 18(1), 499-521.
6. Alves, V., Niu, N., Alves, C., & Valença, G. (2015). Templates for textual use cases of software product lines: Results from a systematic mapping study and a controlled experiment. Journal of Software Engineering Research and Development, 3, Article 5.
7. Arrieta, A., Sagardui, G., Etxeberria, L., & Wang, S. (2021). A variability fault localization approach

for software product lines. IEEE Transactions on Software Engineering, 47(9), 1854-1871.
8. Arrieta, A., Segura, S., Markiegi, U., Sagardui, G., & Etxeberria, L. (2019). Spectrum-based fault localization in software product lines. Information and Software Technology, 108, 89-104.
9. Arrieta, A., Wang, S., Sagardui, G., & Etxeberria, L. (2019). Search-based test case prioritization for simulation-based testing of cyber-physical system product lines. Journal of Systems and Software, 149, 1-34.
10. Berger, T., Rublack, R., Nair, D., Atlee, J. M., Becker, M., Czarnecki, K., & Wąsowski, A. (2020). Software product-line evaluation in the large. Empirical Software Engineering, 25(4), 2993-3040.
11. Beuche, D., & Dalgarno, M. (2016). Software product line engineering with feature models. Pure-Systems Tutorial
12. Bürdek, J., Kehrer, T., Lochau, M., Reuling, D., Kelter, U., & Schürr, A. (2023). Reasoning on feature models: Compilation-based vs. direct approaches. arXiv preprint arXiv:2302.06867.
13. Bürdek, J., Kehrer, T., Lochau, M., Reuling, D., Kelter, U., & Schürr, A. (2016). Reasoning on feature models: Compilation-based vs. direct approaches. Software and Systems Modeling, 15(4), 1155-1188.
14. Buschmann, F., Henney, K., & Schmidt, D. C. (2024). Pattern-oriented software architecture: A system of patterns. Wiley.
15. Cabral, I., Cohen, M. B., & Rothermel, G. (2010). Improving the testing and testability of software product lines. In Proceedings of the 14th International Conference on Software Product Lines (pp. 241-255).
16. Castro, T., Teixeira, L., Alves, V., Apel, S., Cordy, M., & Gheyi, R. (2021). A formal framework of software product line analyses. ACM Computing Surveys, 54(7), 1-39.
17. Czarnecki, K., & Eisenecker, U. W. (2000). Generative programming: Methods, tools, and applications. Addison-Wesley.
18. Devroey, X., Perrouin, G., Cordy, M., Schobbens, P. Y., Legay, A., & Heymans, P. (2014). Towards statistical prioritization for software product lines testing. VaMoS '14 Proceedings of the Eighth International Workshop on Variability Modelling of Software-Intensive Systems, Article No. 10.
19. Engström, E., & Runeson, P. (2024). A comparative study on testing optimization techniques with combinatorial interaction testing for optimizing software product line testing. Applied Sciences and Engineering Technology, 15(3), 142-158.
20. Ferko, E. (2020). Automatic generation of configuration files for software product lines [Master's thesis]. Mälardalen University.
21. Friege, H., & Nasraoui, O. (2024). Minkowski metric, feature weighting and anomalous cluster initializing in K-Means clustering. Pattern Recognition, 127, 108621.
22. Gay, G. (2021). Feature modeling. Software Engineering Course Notes. Chalmers University of Technology.
23. Hajri, I., Goknil, A., Pastore, F., & Briand, L. C. (2020). Automating system test case classification and prioritization for use case-driven testing in product lines. Empirical Software Engineering, 25(5), 3711-3769.
24. Herbold, S. (2022). Exploring the relationship between performance metrics and cost saving potential of defect prediction models. Empirical Software Engineering, 27(6), 147.
25. Herlitz, A. (2022). Safety-critical software - test coverage vs remaining faults [Doctoral dissertation]. KTH Royal Institute of Technology.
26. Huang, Y., Lu, R., Yang, Y., Zhang, L., & Chen, L. (2025). A low-cost feature interaction fault localization approach for software product lines. ACM Transactions on Software Engineering and Methodology, 34(1), 1-32.
27. Huang, Z., & De Almeida, E. S. (2023). Local means-based fuzzy k-nearest neighbor classifier with feature weights and Minkowski distance. International Journal of Fuzzy Systems, 26(2), 496-512.
28. Johnson, C. W., & Weinstock, C. B. (2021). Towards a cross-domain software safety assurance process for embedded systems. arXiv preprint arXiv:2106.02140.
29. Joint Commission. (2024). Root cause analysis and medical error prevention. StatPearls Publishing.
30. Kumar, S., & Rajkumar. (2017). Test case prioritization techniques for software product line: A survey. Proceeding - IEEE International Conference on Computing, Communication and Automation, 884-889.
31. Kumar, S., Mittal, M., & Yadav, V. K. (2020). Cost-effective product prioritisation technique for software product line testing. Engineering Systems Modelling and Simulation, 10, 197-224.

32. Kumar, V., Shukla, K. K., Sharma, D. K., & Singh, M. (2019). Collaborative filtering-based test case prioritization and reduction for software product-line testing. IEEE Access, 7, 149113-149127.
33. Lima, J. A. P., Mendonça, W. D. F., Vergilio, S. R., & Assunção, W. K. G. (2020). Learning-based prioritization of test cases in continuous integration of highly configurable software. Proceedings of the 24th ACM Conference on Systems and Software Product Line, 1-11.
34. Liu, S., & Zhang, W. (2024). Semantic-guided RL for interpretable feature engineering. Proceedings of the International Conference on Machine Learning, 41, 12847-12865.
35. Liu, Y., Wang, X., Chen, S., & Zhang, L. (2024). SPL-DB-Sync: Seamless database transformation during feature model evolution. Information and Software Technology, 170, 107428.
36. Lohmüller, P., & Bauer, B. (2019). Software product line engineering for safety-critical systems. Proceedings of the 15th International Conference on Software Technologies, 404-411.
37. Lopez-Herrejon, R. E., Ferrer, J., Chicano, F., Haslinger, E. N., Egyed, A., & Alba, E. (2014). A parallel evolutionary algorithm for prioritized pairwise testing of software product lines. Proceedings of the 2014 Genetic and Evolutionary Computation Conference, 1255-1262.
38. Machado, I. C., McGregor, J. D., & de Almeida, E. S. (2017). Software product line testing based on feature model mutation. International Journal of Software Engineering and Knowledge Engineering, 27(3), 309-338.
39. Mannion, M., & Kaindl, H. (2021). Using binary strings for comparing products from software-intensive systems product lines. Proceedings of the 15th International Conference on Software Technologies, 104-112.
40. Markiegi, U., Arrieta, A., Etxeberria, L., & Sagardui, G. (2021). Dynamic test prioritization of product lines: An application on configurable simulation models. Software Quality Journal, 30(2), 571-596.
41. Mohamed, H., Jawawi, D. N. A., & Halim, S. A. (2018). Enhancing similarity distances using mandatory and optional for early fault detection. Indonesian Journal of Electrical Engineering and Computer Science, 12(1), 89-97.
42. Mouheb, D., Abbas, A., & Kamel, M. (2025). Unified approaches in self-supervised event stream modeling: Progress and prospects. arXiv preprint arXiv:2502.04899.
43. Oliveira, P., & Costa, R. (2024). An aspect-oriented framework for weaving domain-specific concerns into component-based systems. Journal of Universal Computer Science, 30(8), 991-1019.
44. Oster, S., Zorcic, I., Markert, F., & Lochau, M. (2024). A systematic literature review of test case prioritization technique on software product line testing. ITIIS, 18(10), 4537-4560.
45. Perdek, J., & Vranić, V. (2023). Matrix based approach to structural and semantic analysis supporting software product line evolution. CEUR Workshop Proceedings, 3588, 1-15.
46. Pitchford, M. (2024). How AI impacts the qualification of safety-critical automotive software. LDRA Technical Articles.
47. Raza, M., Akbulut, F. P., & Catal, C. (2024). A virtual testing framework for real-time validation of automotive software systems based on hardware in the loop and fault injection. Sensors, 24(12), 3733.
48. Sahak, M., Jawawi, D. N. A., & Halim, S. A. (2017). An experiment of different similarity measures on test case prioritization for software product lines. Journal of Telecommunication, Electronic and Computer Engineering, 9(3-4), 177-183.
49. Sahak, M., Jawawi, D. N. A., & Halim, S. A. (2019). Similarity distance measure and prioritization algorithm for test case prioritization in software product line testing. Journal of Information and Communication Technology, 18(1), 77-104.
50. Sánchez, A. B., Segura, S., & Ruiz-Cortés, A. (2014). A comparison of test case prioritization criteria for software product lines. Proceedings - IEEE 7th International Conference on Software Testing, Verification and Validation, 41-50.
51. Sarigiannidis, A., Barna, P., Koufos, K., & Moschoglou, S. (2023). Increasing performance and sample efficiency with model-agnostic interactive feature attributions. arXiv preprint arXiv:2306.16431.
52. Shahpar, S., Bardsiri, V. K., & Bardsiri, A. K. (2025). Enhancing analogy-based software cost estimation using Grey Wolf optimization algorithm. Scientific Reports, 15, Article 2837.
53. Shi, Y., Zhang, L., & Wang, X. (2023). Jaccard index based similarity measure to compare transcription factor binding site models. Algorithms for Molecular Biology, 18, Article 23.
54. Sulaiman, R. A., Jawawi, D. N. A., & Halim, S. A. (2021). A dissimilarity with dice-jaro-winkler test case prioritization approach for model-based testing in software product line. KSII Transactions on

Internet and Information Systems, 15(3), 932-951.
55. Sun, Q., Li, X., Chen, W., & Wang, Z. (2024). Waffle: A novel feature modeling language for highly-configurable software systems. IEEE Transactions on Software Engineering, 50(6), 1247-1262.
56. Suwannaphong, S., Phumipat, C., & Thammano, A. (2023). The hybrid Jaro-Winkler and Manhattan distance using dissimilarity measure for test case prioritization approach. International Journal of Advanced Computer Science and Applications, 14(11), 1117-1124.
57. Theeren, A., Saeye, Y., & Cornelis, C. (2024). Feature subset weighting for distance-based supervised learning through Choquet integration. Machine Learning, 113(4), 2847-2885.
58. Thüm, T., Apel, S., Kästner, C., Schaefer, I., & Saake, G. (2024). FM-PRO: A feature modeling process. IEEE Transactions on Software Engineering, 51(1), 89-108.
59. Thüm, T., Kästner, C., Benduhn, F., Meinicke, J., Saake, G., & Leich, T. (2014). FeatureIDE: An extensible framework for feature-oriented software development. Journal of Science of Computer Programming, 79, 70-85.
60. Töçka, E. (2020). Complex variability modeling [Master's thesis]. University of Gothenburg.
61. Torres, R., & López, J. (2022). Towards an approach to pattern-based domain-specific requirements engineering. arXiv preprint arXiv:2404.17338.
62. Wahyudin, A., Harahap, E., Susilo, B., Rochimah, S., & Akbar, R. (2023). Test case prioritization for software product line: A systematic mapping study. Journal on Interactive Systems, 7(3), 150-165.
63. Wu, D., Kim, J., & Lee, S. (2022). A systematic literature review on prioritizing software test cases using coverage-based techniques. Information and Software Technology, 142, 106742.
64. Zilliz. (2025). Similarity metrics for vector search: Understanding Jaccard distance and weighted measures. Zilliz Technical Blog.