# Co-Change Graph Entropy: A New Process Metric for Defect Prediction

Ethari Hrishikesh
Indian Institute of Information Technology, Manipur
Mantripukhri, Imphal, India
hrishikeshethari@gmail.com

Amit Kumar
Indian Institute of Information Technology Allahabad
Prayagraj, India
amitchandramunityagi@gmail.com

Meher Bhardwaj
Indian Institute of Information Technology, Manipur
Mantripukhri, Imphal, India
bhardwajmeher01@gmail.com

Sonali Agarwal
Indian Institute of Information Technology Allahabad
Prayagraj, India
sonali@iiita.ac.in



## Abstract

Process metrics, valued for their language independence and ease of collection, have been shown to outperform product metrics in defect prediction. Among these, change entropy (Hassan, 2009) is widely used at the file level and has proven highly effective. Additionally, past research suggests that co-change patterns provide valuable insights into software quality. Building on these findings, we introduce Co-Change Graph Entropy, a novel metric that models co-changes as a graph to quantify co-change scattering.

Experiments on eight Apache projects reveal a significant correlation between co-change entropy and defect counts at the file level, with a Pearson correlation coefficient of up to 0.54. In file-level defect classification, replacing change entropy with co-change entropy improves AUROC in 72.5% of cases and MCC in 62.5% across 40 experimental settings (five machine learning classifiers and eight projects), though these improvements are not statistically significant. However, when co-change entropy is combined with change entropy, AUROC improves in 82.5% of cases and MCC in 65%, with statistically significant gains confirmed via the Friedman test followed by the post-hoc Nemenyi test.

These results indicate that co-change entropy complements change entropy, significantly enhancing defect classification performance and underscoring its practical importance in defect prediction.

## CCS Concepts

• **Software and its engineering → Software defect analysis**.

## Keywords

Defect Prediction, Change Entropy, Co-change Entropy, Software Quality, Graph-based Metrics, Machine Learning


*EASE 2025, Istanbul, Turkey*

ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

**ACM Reference Format:**
Ethari Hrishikesh, Amit Kumar, Meher Bhardwaj, and Sonali Agarwal. 2025. Co-Change Graph Entropy: A New Process Metric for Defect Prediction. In *Proceedings of The 29th International Conference on Evaluation and Assessment in Software Engineering (EASE 2025).* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

## 1 Introduction and Motivation

Software defect prediction has remained a central topic in software engineering for decades, as it enables the identification of potentially faulty modules, thereby guiding the efficient allocation of testing resources to enhance software quality. While a wide range of product and process metrics have been explored to build predictive models, the pursuit of more effective and reliable metrics remains ongoing. Notably, prior studies suggest that process metrics not only outperform product metrics in predictive performance [1][2][20], but also offer practical advantages—they are easier to collect and are largely language-independent, thus improving their generalizability across diverse software projects [17].

The scattering of changes, also referred to as change entropy [10], has been widely used as a process metric in defect prediction studies. Change entropy describes how dispersed changes are across the modules or files in a system during a specific time interval. Systems where changes are scattered across many files during a given period have been found to be more prone to defects in the future. Apart from the amount and patterns of changes made to a system and their dispersion, the patterns of co-changes—where two or more software entities are modified together in the same commit—have also been found to be associated with software quality [21][22][5]. However, despite the significant impact of co-change patterns on software quality and modularity, the effect of co-change scattering on defects has not been studied in depth, to the best of our knowledge.

In this paper, similar to previous studies [14][22], we model the co-changes occurring in a software system as a graph and propose co-change graph entropy as a process metric to study its impact on defects in the system. Although co-change graph entropy may seem similar to change entropy, it is fundamentally different. To demonstrate how co-change graph entropy is computed and how it differs from change entropy, we consider a toy example where 12 commits ($C_1, C_2, \ldots, C_{12}$) have been made in a system with four source code files (A, B, C, and D) during some time interval $T$. In the first 9 commits ($C_1, C_2, \ldots, C_9$), files A and C are changed, while in commit $C_{10}$, files A and D are changed; in commit $C_{11}$, files C

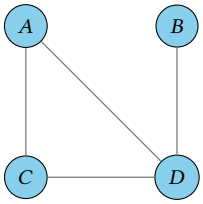



**Figure 1: Co-change graph based on the toy example**

and D are changed; and in commit $C_{12}$, files B and D are changed. A corresponding co-change graph capturing the co-change relationships among the files in the above example is shown in Figure 1.

Change entropy, as defined by Hassan [10], first computes the probability of a file being changed in the time interval $T$, which is computed as the number of times a file $f$ has been changed during $T$ divided by the total number of file changes occurring in the system during the time period $T$. In the above example, the probability of each file being changed in the time period $T$ is as follows: $p_A = 10/24 = 0.416, p_B = 1/24 = 0.0416, p_C = 10/24 = 0.416, p_D = 3/24 = 0.125$. The change entropy of the system is computed using Shannon's entropy formula as follows:.

$$H(S) = -\sum_{k} p_k \log p_k$$

Where $p_k$ is the change probability of a file. Simple arithmetic can be used to compute the change entropy of the system in the toy example, which is 1.68551. Since defect proneness is to be investigated at the file level, the system's entropy in Hassan's model is attributed to the file level. As observed by Hassan, one of the best ways to assign the entropy measure at the file level is to multiply the system's entropy by the change probability of the file. In this way, each file is assigned a change complexity value as follows: $H_A = p_A * H(s) = 0.673, H_B = p_B * H(s) = 0.0673, H_C = p_C * H(s) = 0.673, H_D = p_D * H(s) = 0.202$.

In contrast to change entropy, co-change graph entropy is based on the co-change probability of a node. The co-change probability of a node in a co-change graph is computed as the degree of the node divided by twice the number of edges in the graph. In this way, the co-change probability of the files, based on the co-change graph shown in Figure 1, can be calculated as follows:$p'_A = 2/8 = 0.25, p'_B = 1/8 = 0.125, p'_C = 2/8 = 0.25, p'_D = 3/8 = 0.375$. Corresponding co-change graph entropy $H'(S)$ for the system during time period $T$ can be computed as 1.90584. The co-change entropy assigned to each of the files can easily be computed as follows[1]: $H'_A = p'_A * H'(s) = 0.476, H'_B = p'_B * H'(s) = 0.238, H'_C = p'_C * H'(s) = 0.476, H'_D = p'_D * H'(s) = 0.715$.

It is evident that files A and C have changed 10 times, resulting in significantly higher change entropy than files B and D. However, file D has a higher co-change entropy than A and B, indicating its greater co-change proneness. This highlights the fundamental difference between the two measures. Motivated by past studies highlighting the effectiveness of co-change patterns in assessing software quality and the importance of change entropy in defect prediction, we pose our first research questions as follows:

**RQ1:** *To what extent does co-change entropy correlate with the defect count of software modules?*

In a typical defect prediction scenario, machine learning classifiers are trained on past data (e.g., process metrics of the previous release) and are used to predict defects in the future release of the software. Various process metrics have been proposed and found effective in predicting the defect proneness of modules and the number of defects in them [20][17]. Change entropy has been found to effectively complement other process metrics and improve the performance of defect prediction. However, to investigate how our proposed co-change graph entropy complements other process metrics and how it performs in comparison to change entropy, we ask our second research question as follows:

**RQ2:** *Does co-change entropy complement standard process metrics and change entropy in improving defect prediction accuracy?*

To address our research questions, we conducted experiments on eight popular Apache projects from the SmartSHARK Dataset[25]. Our findings indicate that co-change entropy exhibits a significant correlation with defect count in source code files. Furthermore, co-change entropy proved highly effective in predicting defect proneness across various classifiers. In the majority of dataset and classifier combinations, co-change entropy effectively complemented standard process and change metrics, resulting in improved overall defect prediction performance.

## 2 Related Work

Software defect prediction relies heavily on metrics that capture different aspects of the development process. The two primary categories are product metrics, which assess code quality through measures like cyclomatic complexity, and process metrics, which leverage historical change data, such as lines of code added or deleted. Past research demonstrates the superior predictive power of process metrics over product metrics. Rahman and Devanbu [20] provided influential insights into this trend, which were later validated by large-scale studies such as those by Majumder et al. [17]. Furthermore, Majumder et al. [16] showed that process metrics remain effective even in semi-supervised learning scenarios with limited labeled data. Within process metrics, change-based measures—particularly those capturing patterns of past software version changes—have proven highly effective. Hassan [10] pioneered the use of change entropy at the file level to predict defects. Nagappan et al. [18] identified "change bursts", or frequent changes within short intervals, as strong indicators of defect-prone modules. More recently, Wen et al. [27] emphasized the importance of change sequences, employing Recurrent Neural Networks (RNNs) to model these sequences and significantly improve prediction accuracy.

The concept of co-change, or simultaneous modifications to multiple files, has been recognized as a valuable predictor of software defects. D'Ambros et al. [5] established foundational work by quantifying the impact of co-change frequency and defining metrics that

[1]Throughout this paper, "co-change entropy" and "change entropy" refer to the entropy assigned to a file, as our study focuses on defect prediction at the file level. Additionally, "module" and "source code file" are used interchangeably, as defect proneness is assessed at this level.

demonstrated its utility in defect prediction. While D'Ambros et al. utilized numerous measures based on the raw count of co-changes per file, our approach diverges by first assessing the dispersion of co-changes at the system level. Subsequently, we derive a file-level metric that quantifies each file's contribution to the overall system's co-change entropy. Notably, co-change dispersion and its attribution at the file level have not been used in prior studies, to the best of our knowledge. Kouroshfar [13] further explored co-change dispersion, identifying inter-subsystem co-changes as particularly defect-prone. However, Kouroshfar's analysis relied on simple count-based measures of co-changes within and across subsystems, lacking the nuanced, information-theoretic perspective offered by our graph entropy approach. In contrast to prior work, our study introduces a novel approach using co-change graph entropy. This measure, based on information-theoretic entropy, quantifies co-change dispersion at the system level, moving beyond simple counts. We not only demonstrate the correlation between our proposed co-change entropy and software defects but also explore its ability to enhance the predictive power of other established process metrics. This comprehensive analysis highlights the unique contribution of our co-change entropy metric to improving defect prediction accuracy.

## 3 Experimental setup

This section outlines the datasets and experimental methodology utilized to answer our research questions.

### 3.1 Dataset Description

We conducted our experiments using the SmartSHARK dataset [25], which is widely recognized and has been extensively used in prior research [11] [6]. This dataset undergoes regular updates and is integrated into a broader, well-structured data collection infrastructure[2]. Its usability and effectiveness have been demonstrated in numerous influential defect prediction studies [12][24].

We selected only the projects that have more than 2,000 defects and over 5,000 commits, resulting in a total of nine projects. As the release information for project Phoenix was ambiguous, pre-release changes could not be determined. Consequently, this project was excluded from the study. After this exclusion, our dataset consists of eight projects and 25 releases for defect analysis. The details of the dataset are provided in Table 1. Our dataset complies with the guidelines laid down by Yatish et al. [29], ensuring that it includes a large number of fixed or closed issues, with a significant fraction of issue reports traced to the commits that resolve them. Furthermore, the dataset comprises projects of diverse nature and size, and the defect ratio across the studied releases also exhibits diversity.

### 3.2 Set up to answer RQ1

To assess whether co-change entropy can aid defect prediction, we analyze its correlation with post-release defects. We link defects to a release using the "affected version" field in the issue tracking system, as supported by past studies [4][29]. To collect the change data for a release $R_i$, we follow this approach: if the release time for $R_i$ is $T_i$ and for the previous release it is $T_{i-1}$, then all commits made during $T_{i-1}$ and $T_i$ are assigned to release $R_i$. This follows

[2]https://smartshark.github.io/dbreleases/

**Table 1: Dataset Description**

| Project | Release Type | Release | Edges | Nodes | Defects | Defect Ratio |
|---|---|---|---|---|---|---|
| *Derby* | Training | 10.3.1.4 | 159226 | 1550 | 309 | 0.1653 |
| | Test | 10.5.1.1 | 22589 | 1485 | 412 | 0.240909 |
| *Activemq* | Training | activemq-5.0.0 | 461396 | 1291 | 66 | 0.046404 |
| | | activemq-5.1.0 | 4348 | 294 | 83 | 0.248466 |
| | | activemq-5.2.0 | 4256 | 297 | 106 | 0.302181 |
| | | activemq-5.3.0 | 13672 | 539 | 158 | 0.227586 |
| | Test | activemq-5.5.0 | 1966322 | 3259 | 309 | 0.082263 |
| *Pdfbox* | Training | 1.5.0 | 1086 | 203 | 104 | 0.435897 |
| | | 1.7.0 | 9378 | 416 | 126 | 0.273563 |
| | | 1.8.0 | 159777 | 1005 | 76 | 0.072277 |
| | Test | 2.0.0 | 63155 | 1559 | 797 | 0.489308 |
| *Pig* | Training | release-0.6.0 | 45588 | 503 | 99 | 0.172962 |
| | | release-0.7.0 | 8853 | 347 | 51 | 0.116343 |
| | | release-0.8.0 | 149036 | 1034 | 146 | 0.117929 |
| | Test | release-0.9.0 | 13451 | 474 | 177 | 0.282258 |
| *Kafka* | Training | 0.10.0.0 | 28379 | 562 | 202 | 0.212544 |
| | Test | 0.11.0.0 | 292651 | 997 | 186 | 0.127617 |
| *Maven* | Training | maven-3.1.0 | 27025 | 403 | 178 | 0.337408 |
| | | maven-3.3.9 | 63580 | 538 | 150 | 0.188889 |
| | Test | maven-3.5.0 | 9970 | 320 | 114 | 0.222222 |
| *Struts* | Training | STRUTS_2_3_28 | 3504646 | 2724 | 38 | 0.012115 |
| | Test | STRUTS_2_3_32 | 459284 | 1034 | 59 | 0.04771 |
| *Nifi* | Training | nifi-0.5.0 | 106897 | 592 | 69 | 0.08769 |
| | | nifi-0.6.0 | 5244 | 197 | 50 | 0.200957 |
| | Test | nifi-0.7.0 | 97299 | 1397 | 97 | 0.057123 |

past studies [27]. We removed all commits that change more than 30 files (fatty commits) to ensure there is no noise in our data. Past research also suggests excluding such commits to avoid noise in the data [28].

For correlation analysis, we construct a co-change graph using change data for each release and compute co-change entropy at the file level. Specifically, for each source code file in a release, we obtain its co-change entropy and the number of post-release defects. We then aggregate this data across all studied releases of the project and compute the Spearman and Pearson correlation coefficients between these two variables.

One might question why we do not consider the number of times a pair of files has been co-changed and use a weighted co-change graph instead of an unweighted one. Upon closer examination, we find that as the frequency of co-changes among the same set of files increases, co-change entropy and change entropy tend to converge. This is evident in the toy example provided in the introduction. For a weighted graph, the co-change probability can be computed as $p_k = \frac{d(k)}{\sum_i d(i)}$, where $d(x)$ represents the weighted degree of node $x$ and $\sum_i d(i)$ is the weighted sum of the degrees of all nodes. However, a simple calculation shows that, in this case, the weighted co-change probability for each file effectively reduces to the file's change entropy, offering no additional benefit from a feature engineering perspective. Additionally, when edge weights are small, the co-change probability of the weighted graph approaches that of the unweighted graph. Recognizing these observations, we opted to use only the unweighted co-change graph in all our co-change entropy calculations.

### 3.3 Set up to answer RQ2

The answer to RQ1 determines whether co-change entropy is useful for defect prediction. To evaluate its practical utility, we integrate it into machine learning classifiers alongside other metrics and analyze its impact on defect prediction accuracy. In addition to change entropy and co-change entropy-based file measures, we use process metrics used by Rahman and Devanbu [20]. To compare change entropy with co-change graph entropy, we define the following

three sets of metrics for training machine learning classifiers on past releases and predicting defects in future releases.

- **P+C**: The metrics provided in Rahman & Devanbu's study, including change entropy (referred to as "changed code scattering" in their work).
- **P+Co**: The metrics provided in Rahman & Devanbu's study, except that change entropy is replaced with co-change entropy.
- **P+C+Co**: The metrics provided in Rahman & Devanbu's study, including change entropy + our proposed co-change entropy.

To investigate the effectiveness of co-change entropy in defect classification, we selected five widely used machine learning classifiers: Logistic Regression, Support Vector Machine, XGBoost, Random Forest, and Gradient Boosting. We used the SciKit-Learn [19] implementation of all these classifiers. The defect prediction dataset suffers from class imbalance. To address this, we apply SMOTE [3], a widely used class balancing technique in software engineering research [17][26].

In a typical cross-version defect classification task utilizing process metrics, software development process quality metrics for software modules are collected from previous releases and used to train a binary classifier that predicts defect proneness in a future release. Generally, for releases $R_1, R_2, R_3, R_4, \ldots, R_{n-1}$, metric values are collected, and the defect proneness of modules is predicted for release $R_n$.

For each classifier and each project, we train three models using the P+C, P+Co, and P+C+Co sets of metrics. The process metrics from training releases are used to evaluate predictive performance on the test release. To assess classifier performance, we use AUROC, F1-score, MCC (Matthews Correlation Coefficient), Precision, and Recall. These metrics are widely used for binary classification and have been extensively used in defect classification research [9][20][23].

**Table 2: Correlation analysis with Bug Count**

| | *Change Entropy* | | | | *Co_Change_Entropy* | | | |
|---|---|---|---|---|---|---|---|---|
| *Project* | *P_Corr* | *P_pval* | *S_Corr* | *S_pval* | *P_Corr* | *P_pval* | *S_Corr* | *S_pval* |
| ***Derby*** | 0.779 | 0.00E+00 | 0.567 | 1.78E-204 | 0.461 | 1.44E-126 | 0.25 | 2.05E-35 |
| ***Activemq*** | 0.548 | 3.60E-129 | 0.311 | 4.23E-38 | 0.516 | 3.94E-112 | 0.283 | 1.37E-31 |
| ***Pdfbox*** | 0.385 | 1.13E-71 | 0.411 | 1.55E-82 | 0.125 | 2.01E-08 | 0.196 | 7.31E-19 |
| ***Pig*** | 0.683 | 1.82E-151 | 0.445 | 2.32E-54 | 0.358 | 1.93E-34 | 0.145 | 1.41E-06 |
| ***Kafka*** | 0.729 | 4.58E-166 | 0.522 | 9.05E-71 | 0.314 | 2.93E-24 | 0.13 | 3.97E-05 |
| ***Maven*** | 0.637 | 1.50E-86 | 0.405 | 5.65E-31 | 0.343 | 3.63E-22 | 0.166 | 4.89E-06 |
| ***Struts*** | 0.467 | 1.27E-30 | 0.321 | 2.05E-14 | 0.542 | 1.47E-42 | 0.346 | 1.16E-16 |
| ***Nifi*** | 0.337 | 3.96E-28 | 0.359 | 5.78E-32 | 0.143 | 5.32E-06 | 0.006 | 8.54E-01 |

**Note:** P -> Pearson, S -> Spearman, Corr -> Correlation, pval -> p-value.

## 4 Results

Table 2 shows the correlation analysis to answer RQ1. It can easily be seen that co-change entropy is significantly correlated with the number of defects. For a better comparison, we also show the correlation results for change entropy with the post release defect count. It can easily be verified from the table that change entropy is better correlated with the defect count than the co-change graph entropy. However, both Pearson and Spearman correlation coefficients between co-change entropy and post-release defect counts are significant. It can also be seen that Pearson correlation coefficients are significantly more than Spearman coefficients, indicating the linear relationship between the independent variable (co-change graph entropy) and the dependent variable, i.e., post-release defect counts. The consistently low p-value (<0.05) across all projects indicates the statistical significance of our results. Based on this observation, we answer our first research question as follows:

**Answer to RQ1:** *Although the correlation between change entropy and post-release defects is higher than that of co-change entropy, co-change entropy remains significantly correlated with post-release defects, with a Pearson correlation coefficient exceeding 0.50 in some cases. Moreover, the higher Pearson correlation coefficient compared to Spearman's suggests that the relationship between co-change graph entropy and post-release defects tends toward linearity.*

In defect classification, a single metric is rarely used. Instead, multiple metrics are combined to train machine learning classifiers, which then predict defect proneness in future releases. Thus, a metric's high correlation with defect count alone is insufficient; it should complement existing metrics to enhance overall classification performance. To evaluate the effectiveness of our proposed co-change entropy in complementing other process metrics (as discussed in Section 3.3), we trained three classifier versions for each dataset-classifier pair: P+C, P+Co, and P+C+Co, and assessed their performance. Table 3 and Figure 2 summarize the results. Notably, replacing change entropy with co-change entropy in process metrics (P+Co) consistently outperforms the process metrics with change entropy (P+C). As shown in Figure 2a, across 40 dataset-classifier pairs (8 projects × 5 classifiers), co-change entropy improves AUROC in 72.5% of the cases, F1-score in 77.5%, MCC in 62.5%, precision in 60%, and recall in 75%.

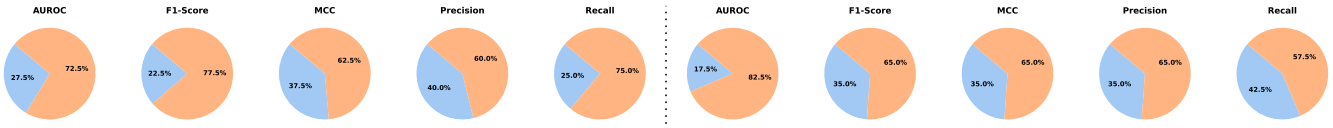
Interestingly, although change entropy exhibits a stronger correlation with defect count than co-change entropy, classifier performance in defect prediction improves when co-change entropy replaces change entropy in the process metrics. This indicates that co-change entropy better complements other process metrics than change entropy. To analyze their combined effect, we compare classifier performance between P+C (process metrics with change entropy) and P+C+Co (process metrics with both change entropy and co-change entropy). The summary in Figure 2b shows that in 82.5% of dataset-classifier pairs, AUROC is higher with P+C+Co than with P+C. A similar trend is observed across other performance measures, such as F1-score, precision, and MCC, where 65% of the cases show an improvement with P+C+Co, while recall is improved in only 57.5% of the cases. This demonstrates that co-change entropy complements change entropy in most cases, enhancing defect classification performance overall.

To statistically validate whether classifier performance differs significantly, we applied the non-parametric Friedman test followed by the post-hoc Nemenyi test, as recommended in prior research [7][8][15]. These tests were conducted across five performance metrics: AUROC, F1-score, precision, recall, and MCC. As shown in Table 4, the Friedman test indicates that classifier performance differs significantly for AUROC, F1-score, precision, and MCC, but not for recall. The Nemenyi test revealed no significant difference between P+C and P+Co, but a statistically significant improvement with P+C+Co over P+C across all metrics except recall. These findings suggest that co-change entropy effectively complements process metrics, demonstrating its practical utility in defect prediction. In summary, the answer to our second research question is as

**Table 3: Defect Classification Results**

| *Classifier* | | *Logistic Regression* | | | *Support Vector Machine* | | | *Random Forest* | | | *XGBoost* | | | *Gradient Boosting* | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| *Project* | *Metrics* | *P + C* | *P + Co* | *P + C + Co* | *P + C* | *P + Co* | *P + C + Co* | *P + C* | *P + Co* | *P + C + Co* | *P + C* | *P + Co* | *P + C + Co* | *P + C* | *P + Co* | *P + C + Co* |
| *Derby* | *AUROC* | 0.77 | 0.763 | 0.77 | 0.731 | 0.718 | 0.724 | 0.789 | 0.799 | 0.792 | 0.785 | 0.773 | 0.773 | 0.787 | 0.797 | 0.8 |
| | *F1-Score* | 0.649 | 0.634 | 0.65 | 0.594 | 0.593 | 0.586 | 0.685 | 0.688 | 0.686 | 0.695 | 0.673 | 0.673 | 0.688 | 0.69 | 0.698 |
| | *MCC* | 0.447 | 0.419 | 0.451 | 0.389 | 0.394 | 0.387 | 0.414 | 0.449 | 0.431 | 0.409 | 0.371 | 0.371 | 0.386 | 0.431 | 0.417 |
| | *Precision* | 0.823 | 0.802 | 0.827 | 0.804 | 0.811 | 0.809 | 0.738 | 0.779 | 0.758 | 0.72 | 0.703 | 0.703 | 0.703 | 0.754 | 0.726 |
| | *Recall* | 0.535 | 0.524 | 0.535 | 0.471 | 0.468 | 0.46 | 0.639 | 0.616 | 0.627 | 0.672 | 0.646 | 0.646 | 0.674 | 0.636 | 0.672 |
| *Activemq* | *AUROC* | 0.771 | 0.777 | 0.778 | 0.691 | 0.708 | 0.709 | 0.763 | 0.747 | 0.775 | 0.759 | 0.772 | 0.759 | 0.717 | 0.675 | 0.687 |
| | *F1-Score* | 0.792 | 0.796 | 0.801 | 0.785 | 0.793 | 0.792 | 0.787 | 0.802 | 0.791 | 0.804 | 0.812 | 0.792 | 0.758 | 0.752 | 0.75 |
| | *MCC* | 0.323 | 0.339 | 0.355 | 0.272 | 0.291 | 0.292 | 0.358 | 0.353 | 0.358 | 0.381 | 0.391 | 0.331 | 0.263 | 0.2 | 0.213 |
| | *Precision* | 0.733 | 0.738 | 0.743 | 0.712 | 0.714 | 0.715 | 0.76 | 0.739 | 0.756 | 0.756 | 0.751 | 0.738 | 0.729 | 0.702 | 0.709 |
| | *Recall* | 0.861 | 0.865 | 0.868 | 0.875 | 0.891 | 0.888 | 0.815 | 0.878 | 0.828 | 0.858 | 0.884 | 0.855 | 0.789 | 0.809 | 0.795 |
| *Pdfbox* | *AUROC* | 0.719 | 0.725 | 0.72 | 0.54 | 0.53 | 0.539 | 0.774 | 0.753 | 0.792 | 0.739 | 0.751 | 0.749 | 0.725 | 0.777 | 0.709 |
| | *F1-Score* | 0.756 | 0.762 | 0.761 | 0.738 | 0.756 | 0.753 | 0.777 | 0.776 | 0.795 | 0.769 | 0.776 | 0.782 | 0.766 | 0.784 | 0.757 |
| | *MCC* | 0.217 | 0.231 | 0.229 | 0.202 | 0.224 | 0.224 | 0.28 | 0.272 | 0.378 | 0.231 | 0.266 | 0.295 | 0.249 | 0.3 | 0.265 |
| | *Precision* | 0.658 | 0.658 | 0.659 | 0.663 | 0.66 | 0.663 | 0.664 | 0.661 | 0.704 | 0.649 | 0.656 | 0.664 | 0.662 | 0.66 | 0.681 |
| | *Recall* | 0.888 | 0.907 | 0.9 | 0.833 | 0.884 | 0.871 | 0.934 | 0.938 | 0.912 | 0.943 | 0.95 | 0.95 | 0.909 | 0.965 | 0.852 |
| *Pig* | *AUROC* | 0.621 | 0.601 | 0.621 | 0.595 | 0.586 | 0.602 | 0.635 | 0.607 | 0.617 | 0.613 | 0.609 | 0.636 | 0.658 | 0.624 | 0.649 |
| | *F1-Score* | 0.68 | 0.682 | 0.68 | 0.686 | 0.69 | 0.688 | 0.696 | 0.687 | 0.689 | 0.725 | 0.708 | 0.712 | 0.681 | 0.691 | 0.694 |
| | *MCC* | 0.105 | 0.115 | 0.105 | 0.107 | 0.127 | 0.131 | 0.067 | 0.073 | 0.054 | 0.148 | 0.144 | 0.118 | 0.026 | 0.08 | 0.071 |
| | *Precision* | 0.682 | 0.686 | 0.682 | 0.682 | 0.69 | 0.692 | 0.665 | 0.668 | 0.661 | 0.689 | 0.692 | 0.681 | 0.653 | 0.67 | 0.667 |
| | *Recall* | 0.678 | 0.678 | 0.678 | 0.69 | 0.69 | 0.684 | 0.73 | 0.707 | 0.718 | 0.764 | 0.724 | 0.747 | 0.713 | 0.713 | 0.724 |
| *Kafka* | *AUROC* | 0.674 | 0.678 | 0.679 | 0.632 | 0.632 | 0.634 | 0.677 | 0.71 | 0.734 | 0.625 | 0.682 | 0.682 | 0.646 | 0.683 | 0.681 |
| | *F1-Score* | 0.544 | 0.536 | 0.55 | 0.577 | 0.587 | 0.573 | 0.613 | 0.632 | 0.634 | 0.619 | 0.663 | 0.663 | 0.623 | 0.662 | 0.65 |
| | *MCC* | 0.211 | 0.199 | 0.224 | 0.162 | 0.178 | 0.15 | 0.227 | 0.283 | 0.275 | 0.145 | 0.257 | 0.257 | 0.19 | 0.273 | 0.259 |
| | *Precision* | 0.632 | 0.625 | 0.641 | 0.578 | 0.585 | 0.571 | 0.609 | 0.644 | 0.635 | 0.551 | 0.596 | 0.596 | 0.576 | 0.61 | 0.609 |
| | *Recall* | 0.478 | 0.469 | 0.482 | 0.576 | 0.588 | 0.576 | 0.616 | 0.62 | 0.633 | 0.706 | 0.747 | 0.747 | 0.678 | 0.722 | 0.698 |
| *Maven* | *AUROC* | 0.66 | 0.648 | 0.668 | 0.622 | 0.597 | 0.611 | 0.646 | 0.658 | 0.668 | 0.599 | 0.616 | 0.638 | 0.636 | 0.62 | 0.641 |
| | *F1-Score* | 0.54 | 0.475 | 0.5 | 0.442 | 0.442 | 0.462 | 0.424 | 0.293 | 0.4 | 0.4 | 0.265 | 0.376 | 0.467 | 0.349 | 0.378 |
| | *MCC* | 0.298 | 0.226 | 0.244 | 0.199 | 0.199 | 0.211 | 0.243 | 0.193 | 0.235 | 0.175 | 0.023 | 0.163 | 0.207 | 0.081 | 0.115 |
| | *Precision* | 0.531 | 0.5 | 0.5 | 0.49 | 0.49 | 0.491 | 0.568 | 0.6 | 0.576 | 0.488 | 0.361 | 0.487 | 0.483 | 0.404 | 0.429 |
| | *Recall* | 0.548 | 0.452 | 0.5 | 0.403 | 0.403 | 0.435 | 0.339 | 0.194 | 0.306 | 0.339 | 0.21 | 0.306 | 0.452 | 0.306 | 0.339 |
| *Struts* | *AUROC* | 0.499 | 0.573 | 0.58 | 0.727 | 0.749 | 0.743 | 0.705 | 0.722 | 0.744 | 0.649 | 0.685 | 0.685 | 0.751 | 0.702 | 0.7 |
| | *F1-Score* | 0.703 | 0.739 | 0.734 | 0.684 | 0.688 | 0.688 | 0.613 | 0.634 | 0.607 | 0.613 | 0.741 | 0.741 | 0.711 | 0.724 | 0.708 |
| | *MCC* | -0.25 | -0.228 | -0.236 | 0.426 | 0.448 | 0.448 | 0.361 | 0.27 | 0.418 | 0.197 | 0.12 | 0.12 | 0.265 | 0.151 | 0.141 |
| | *Precision* | 0.639 | 0.654 | 0.651 | 0.931 | 0.947 | 0.947 | 0.92 | 0.836 | 0.978 | 0.794 | 0.724 | 0.724 | 0.8 | 0.74 | 0.739 |
| | *Recall* | 0.78 | 0.85 | 0.84 | 0.54 | 0.54 | 0.54 | 0.46 | 0.51 | 0.44 | 0.5 | 0.76 | 0.76 | 0.64 | 0.71 | 0.68 |
| *Nifi* | *AUROC* | 0.67 | 0.7 | 0.703 | 0.586 | 0.553 | 0.582 | 0.687 | 0.666 | 0.711 | 0.625 | 0.685 | 0.69 | 0.555 | 0.616 | 0.641 |
| | *F1-Score* | 0.449 | 0.5 | 0.498 | 0.479 | 0.471 | 0.483 | 0.544 | 0.515 | 0.55 | 0.469 | 0.49 | 0.498 | 0.466 | 0.494 | 0.473 |
| | *MCC* | 0.17 | 0.232 | 0.227 | 0.194 | 0.178 | 0.195 | 0.273 | 0.21 | 0.288 | 0.09 | 0.138 | 0.163 | 0.101 | 0.172 | 0.116 |
| | *Precision* | 0.411 | 0.435 | 0.432 | 0.411 | 0.402 | 0.408 | 0.404 | 0.379 | 0.42 | 0.333 | 0.343 | 0.353 | 0.341 | 0.371 | 0.346 |
| | *Recall* | 0.494 | 0.587 | 0.587 | 0.576 | 0.57 | 0.593 | 0.831 | 0.802 | 0.797 | 0.791 | 0.86 | 0.849 | 0.733 | 0.738 | 0.75 |
| *Average* | *AUROC* | 0.673 | **0.683** | **0.69** | 0.641 | 0.634 | **0.643** | 0.71 | 0.708 | **0.729** | 0.674 | **0.697** | **0.702** | 0.684 | **0.687** | **0.689** |
| | *F1-Score* | 0.639 | **0.641** | **0.647** | 0.623 | **0.628** | **0.628** | 0.642 | 0.628 | **0.644** | 0.637 | **0.641** | **0.655** | 0.645 | 0.643 | 0.639 |
| | *MCC* | 0.19 | **0.192** | **0.2** | 0.244 | **0.255** | **0.255** | 0.278 | 0.263 | **0.305** | 0.222 | 0.214 | **0.227** | 0.211 | **0.211** | 0.2 |
| | *Precision* | 0.639 | 0.637 | **0.642** | 0.659 | **0.662** | **0.662** | 0.666 | 0.663 | **0.686** | 0.623 | 0.603 | 0.618 | 0.618 | 0.614 | 0.613 |
| | *Recall* | 0.658 | **0.667** | **0.674** | 0.621 | **0.629** | **0.631** | 0.671 | 0.658 | 0.658 | 0.697 | **0.723** | **0.733** | 0.699 | **0.7** | 0.689 |

**Note:** Bold highlights improvements over P with respect to the corresponding evaluation metric.




**(a) P+C (blue) vs. P+Co (orange)** **(b) P+C (blue) vs. P+C+Co (orange)**

**Figure 2: Comparison of Predictive-performance**

follows:

**Answer to RQ2:** *Substituting change entropy with co-change entropy in the process metrics set resulted in improved defect classification performance, with AUROC values increasing in 72.5% of cases. However, this improvement was not statistically significant, failing the post-hoc Nemenyi test following the Friedman test. In contrast, combining change entropy and co-change entropy significantly enhanced defect classification performance, leading to AUROC improvements in 82.5% of cases. This improvement was statistically significant, as confirmed by a successful post-hoc Nemenyi test after the Friedman test.*

**Table 4: Friedman and Post-hoc Nemenyi's Test**

| | *AUROC* | *F1* | *MCC* | *Precision* | *Recall* |
|---|---|---|---|---|---|
| *Friedman P+C,P+Co,P+C+Co* | ✓ | ✓ | ✓ | ✓ | x |
| *Nemenyi P+C vs P+Co* | x | x | x | x | x |
| *Nemenyi P+C vs P+C+Co* | ✓ | ✓ | ✓ | ✓ | x |

**Note:** '✓' = p-value $< 0.05$ (The difference between the groups is statistically significant); 'x' = p-value $> 0.05$ (The difference between the groups is not statistically significant).

## 5 Threats to validity

**External Validity:** A key limitation of this study is its reliance on the SmartSHARK dataset [25], which, although widely used, primarily consists of Java projects. This homogeneity limits the generalizability of our findings to software systems developed in other programming languages. While our project selection followed

established criteria [23], incorporating projects from more diverse datasets would strengthen the robustness and applicability of our results.

**Practical Validity:** Although the combined use of change and co-change entropy metrics resulted in statistically significant improvements in defect classification, the performance gain of 2–3% remains relatively modest. This may limit the practical adoption of these metrics in real-world defect prediction models. Nevertheless, our findings highlight the potential of entropy-based graph measures and encourage further research into more advanced techniques that could yield greater predictive accuracy.

## 6 Conclusion

This study introduced co-change entropy, a novel metric that quantifies co-change dispersion in software development using co-change graphs. Our analysis revealed a significant correlation between co-change entropy and post-release defects, with Pearson correlation coefficients exceeding 0.50 in several projects. While replacing traditional change metrics with co-change metrics in defect classification led to performance improvements, these gains were not statistically significant. However, combining change and co-change entropy significantly enhanced prediction accuracy, as confirmed by statistical testing. These findings highlight the value of incorporating co-change entropy into process metrics to improve defect prediction. Our research is easily verifiable and reproducible, as we publish our code and all other artifacts necessary to reproduce our results in a publicly available replication kit[3]

Software change history has been represented using more sophisticated graph constructs, such as hypergraphs and heterogeneous information networks, in previous studies. These advanced representations offer new opportunities for capturing software evolution patterns. For instance, heterogeneous graphs can model co-change dispersion at various types of node levels, providing a richer structural understanding. Extending this work by exploring heterogeneous graph entropy to capture such dispersions could be a promising direction for future research.

## References


[1] Christian Bird, Nachiappan Nagappan, Premkumar Devanbu, Harald Gall, and Brendan Murphy. 2009. Does distributed development affect software quality? an empirical case study of windows vista. *Commun. ACM* 52, 8 (2009), 85–93.

[2] Christian Bird, Nachiappan Nagappan, Brendan Murphy, Harald Gall, and Premkumar Devanbu. 2011. Don't touch my code! Examining the effects of ownership on software quality. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. Association for Computing Machinery, New York, NY, USA, 4–14.

[3] Nitesh V Chawla, Kevin W Bowyer, Lawrence O Hall, and W Philip Kegelmeyer. 2002. SMOTE: synthetic minority over-sampling technique. *Journal of artificial intelligence research* 16 (2002), 321–357.

[4] Daniel Alencar Da Costa, Shane McIntosh, Weiyi Shang, Uirá Kulesza, Roberta Coelho, and Ahmed E Hassan. 2016. A framework for evaluating the results of the szz approach for identifying bug-introducing changes. *IEEE Transactions on Software Engineering* 43, 7 (2016), 641–657.

[5] Marco D'Ambros, Michele Lanza, and Romain Robbes. 2009. On the relationship between change coupling and software defects. In *2009 16th Working Conference on Reverse Engineering*. IEEE, IEEE Computer Society, USA, 135–144.

[6] Carlos DA de Almeida, Diego N Feijó, and Lincoln S Rocha. 2022. Studying the impact of continuous delivery adoption on bug-fixing time in apache's open-source projects. In *Proceedings of the 19th International Conference on Mining Software Repositories*. Association for Computing Machinery, New York, NY, USA, 132–136.

[7] Janez Demšar. 2006. Statistical comparisons of classifiers over multiple data sets. *Journal of Machine learning research* 7, Jan (2006), 1–30.

[8] Lina Gong, Shujuan Jiang, Rongcun Wang, and Li Jiang. 2019. Empirical evaluation of the impact of class overlap on software defect prediction. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, IEEE Press, San Diego, California, 698–709.

[9] Lina Gong, Gopi Krishnan Rajbahadur, Ahmed E Hassan, and Shujuan Jiang. 2021. Revisiting the impact of dependency network metrics on software defect prediction. *IEEE Transactions on Software Engineering* 48, 12 (2021), 5030–5049.

[10] Ahmed E Hassan. 2009. Predicting faults using the complexity of code changes. In *2009 IEEE 31st international conference on software engineering*. IEEE, IEEE Computer Society, USA, 78–88.

[11] Steffen Herbold. 2019. On the costs and profit of software defect prediction. *IEEE Transactions on Software Engineering* 47, 11 (2019), 2617–2631.

[12] Steffen Herbold, Alexander Trautsch, Fabian Trautsch, and Benjamin Ledel. 2022. Problems with SZZ and features: An empirical study of the state of practice of defect prediction data collection. *Empirical Software Engineering* 27, 2 (2022), 42.

[13] Ehsan Kouroshfar. 2013. Studying the effect of co-change dispersion on software quality. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, IEEE Press, San Francisco, CA, USA, 1450–1452.

[14] Amit Kumar, Ethari Hrishikesh, Yugandhar Deasi, and Sonali Agarwal. 2024. Prevalence and Prediction of Unseen Co-Changes: A Graph-Based Approach. In *2024 IEEE 48th Annual Computers, Software, and Applications Conference (COMPSAC)*. IEEE, IEEE Computer Society, Los Alamitos, CA, USA, 1157–1167.

[15] Stefan Lessmann, Bart Baesens, Christophe Mues, and Swantje Pietsch. 2008. Benchmarking classification models for software defect prediction: A proposed framework and novel findings. *IEEE transactions on software engineering* 34, 4 (2008), 485–496.

[16] Suvodeep Majumder, Joymallya Chakraborty, and Tim Menzies. 2024. When less is more: on the value of "co-training" for semi-supervised software defect predictors. *Empirical Software Engineering* 29, 2 (2024), 51.

[17] Suvodeep Majumder, Pranav Mody, and Tim Menzies. 2022. Revisiting process versus product metrics: a large scale analysis. *Empirical Software Engineering* 27, 3 (2022), 60.

[18] Nachiappan Nagappan, Andreas Zeller, Thomas Zimmermann, Kim Herzig, and Brendan Murphy. 2010. Change bursts as defect predictors. In *2010 IEEE 21st international symposium on software reliability engineering*. IEEE, IEEE Computer Society, USA, 309–318.

[19] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. 2011. Scikit-learn: Machine learning in Python. *the Journal of machine Learning research* 12 (2011), 2825–2830.

[20] Foyzur Rahman and Premkumar Devanbu. 2013. How, and why, process metrics are better. In *2013 35th international conference on software engineering (ICSE)*. IEEE, IEEE Press, San Francisco, CA, USA, 432–441.

[21] Luciana L Silva, Marco Tulio Valente, and Marcelo A Maia. 2019. Co-change patterns: A large scale empirical study. *Journal of Systems and Software* 152 (2019), 196–214.

[22] Luciana Lourdes Silva, Marco Tulio Valente, and Marcelo de A Maia. 2014. Assessing modularity using co-change clusters. In *Proceedings of the 13th international conference on Modularity*. Association for Computing Machinery, New York, NY, USA, 49–60.

[23] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E Hassan, and Kenichi Matsumoto. 2018. The impact of automated parameter optimization on defect prediction models. *IEEE Transactions on Software Engineering* 45, 7 (2018), 683–711.

[24] Alexander Trautsch, Steffen Herbold, and Jens Grabowski. 2020. Static source code metrics and static analysis warnings for fine-grained just-in-time defect prediction. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, IEEE Computer Society, Los Alamitos, CA, USA, 127–138.

[25] Alexander Trautsch, Fabian Trautsch, and Steffen Herbold. 2021. MSR Mining Challenge: The SmartSHARK Repository Mining Data. *arXiv preprint arXiv:2102.11540* abs/2102.11540 (2021).

[26] Shuo Wang and Xin Yao. 2013. Using class imbalance learning for software defect prediction. *IEEE Transactions on Reliability* 62, 2 (2013), 434–443.

[27] Ming Wen, Rongxin Wu, and Shing-Chi Cheung. 2018. How well do change sequences predict defects? sequence learning from software changes. *IEEE Transactions on Software Engineering* 46, 11 (2018), 1155–1175.

[28] Lu Xiao, Yuanfang Cai, and Rick Kazman. 2014. Design rule spaces: A new form of architecture insight. In *Proceedings of the 36th International Conference on Software Engineering*. Association for Computing Machinery, New York, NY, USA, 967–977.

[29] Suraj Yatish, Jirayus Jiarpakdee, Patanamon Thongtanunam, and Chakkrit Tantithamthavorn. 2019. Mining software defects: Should we consider affected releases?. In *2019 IEEE/ACM 41st international conference on software engineering (ICSE)*. IEEE, IEEE Press, Montreal, Quebec, Canada, 654–665.


[3] https://github.com/hrishikeshethari/EASE2025