Appendix

This file is the appendix of the paper "Boosting Code-line-level Defect Prediction with Spectrum Information and Causality Analysis". Due to the space limitation of the submitted paper, we present more detailed experimental results here. The appendix can be divided into seven parts: performance under more token selection, a real example with the ranking list of code lines in a file, IFA improvements on non-effective projects, statistical tests, performance under other prediction settings, performance under other classification metrics, and performance under other effort-aware metrics. The appendix is online at https://github.com/SOUNDLineDP/SOUND/blob/main/Appendix.pdf.

A. Performance under more token selection

Motivation and Approach. Our selection of the top 100 tokens (see Section **III.B**, Page 5 of the paper) is based on the characteristics of causality analysis, which are time-consuming and with high hardware requirements. Besides, we consider the top 100 tokens the primary representative information across releases. If we extend the token number, the potential overfitting would increase. To alleviate concerns about the impact of limited causal analysis on experimental results, we selected the top 200 tokens in the Causal Analysis to conduct a controlled experiment.

Results. Table I shows the comparison results between the top 100 and the top 200 token settings. As can be seen in Table I, in two cases under IFA and two cases under Effort@20%, performance with the top 100 is better than that of the top 200; in contrast, in five cases under IFA, performance with the top 200 is better than the top 100. In the remaining 11 cases, their performances are the same. We conclude that the performances of the top 100 and top 200 are almost the same.

TABLE I: Comparison between our method SOUND under the top 100 and 200 tokens. Gray background indicates better.

		Top 100		Top 200	
		Mean	Median	Mean	Median
IFA	Barinel	13	0	14	0
	Dstar	60	8	52	8
	Ochiai	45	2	44	2
	Op2	69	11	70	8
	Tarantula	21	3	18	1
Effort@20%	Barinel	0.039	0.007	0.039	0.007
	Dstar	0.072	0.044	0.072	0.044
	Ochiai	0.054	0.028	0.054	0.028
	Op2	0.077	0.047	0.077	0.048
	Tarantula	0.052	0.021	0.052	0.022

Conclusion. Extending the number of tokens in the causal analysis does not obviously affect the performance of our methods.

B. A real example with the ranking list of the code lines in a file

Motivation and Approach. To demonstrate the rationale of our method in a real scenario, we present a detailed example along with the ranking list of the code lines in a file that is easy to check.

We select the file NetworkBridgeFactory.java in the project amq-5.3.0, which contains fewer than 100 lines, making it easy for manual checks. We construct the prediction model using the historical data from the previous release amq-5.2.0. We conduct an experiment in this file to compare our method (with `Barinel`) with the baseline method GLANCE-LR[1].

Results. We present the results of ranking all lines (excluding meaningless lines, e.g., lines just containing } or {, and comment lines) in NetworkBridgeFactory.java in Figure 1. From Figure 1, we have the following observatoins:

- For our method with `Barinel`, based on the historical data information in amq-5.2.0, the suspiciousness score of `remoteTransport` is approximately 0.03. As a result, lines 59 and 61 that contain `remoteTransport` are ranked with higher priority (they are also actual buggy lines).
- For GLANCE-LR, the score is evaluated based on NT (the number of tokens) and NFC (the number of function calls). In this example, line 41, which is ranked first, is with NT = 6 and NFC = 1. So the line score is NT*(NFC+1) = 12. Moreover, the lines containing control elements are given higher priority, including `return`, `if`, and `else`.

We conclude that (a) from the IFA view, our method achieves 1, and GLANCE-LR achieves 2, and (b) for the Recall@20% view, since the total number of lines is 20 here, we check the performance within the top 4 ($20 \times 20\%$) lines. Our method identified two buggy lines, while GLANCE-LR identified one. By comparing the results of the methods, we can see that our method, based on historical data, performs better than the baseline GLANCE-LR.

Conclusion. This real example shows that our method based on historical data performs better than baselines.

[1]As GLANCE-LR obtains the best IFA values among all baselines, we consider GLANCE-LR as the main baseline.

predicted_buggy_lines	predicted_score	src
activemq-core/src/main/java/org/apache/activemq/network/NetworkBridgeFactory.java:55	0.083976627	final NetworkBridgeListener listener) {
activemq-core/src/main/java/org/apache/activemq/network/NetworkBridgeFactory.java:59	0.03732655	result = new ConduitBridge(configuration, localTransport, remoteTransport);
activemq-core/src/main/java/org/apache/activemq/network/NetworkBridgeFactory.java:61	0.03732655	result = new DurableConduitBridge(configuration, localTransport, remoteTransport);
activemq-core/src/main/java/org/apache/activemq/network/NetworkBridgeFactory.java:64	0.03732655	result = new DemandForwardingBridge(configuration, localTransport, remoteTransport);
activemq-core/src/main/java/org/apache/activemq/network/NetworkBridgeFactory.java:40	0.033962264	Transport localTransport, Transport remoteTransport) {
activemq-core/src/main/java/org/apache/activemq/network/NetworkBridgeFactory.java:54	0.033962264	Transport localTransport, Transport remoteTransport,
activemq-core/src/main/java/org/apache/activemq/network/NetworkBridgeFactory.java:41	0.031839765	return createBridge(config, localTransport, remoteTransport, null);
activemq-core/src/main/java/org/apache/activemq/network/NetworkBridgeFactory.java:17	0.010383388	package org.apache.activemq.network;
activemq-core/src/main/java/org/apache/activemq/network/NetworkBridgeFactory.java:53	0.009309478	public static DemandForwardingBridge createBridge(NetworkBridgeConfiguration configuration,
activemq-core/src/main/java/org/apache/activemq/network/NetworkBridgeFactory.java:57	0.008696755	if (configuration.isConduitSubscriptions()) {
activemq-core/src/main/java/org/apache/activemq/network/NetworkBridgeFactory.java:58	0.008696755	if (configuration.isDynamicOnly()) {
activemq-core/src/main/java/org/apache/activemq/network/NetworkBridgeFactory.java:66	0.004166802	if (listener != null) {
activemq-core/src/main/java/org/apache/activemq/network/NetworkBridgeFactory.java:19	0.004069078	import org.apache.activemq.transport.Transport;
activemq-core/src/main/java/org/apache/activemq/network/NetworkBridgeFactory.java:56	0.002463054	DemandForwardingBridge result = null;
activemq-core/src/main/java/org/apache/activemq/network/NetworkBridgeFactory.java:26	0.002405083	public final class NetworkBridgeFactory {
activemq-core/src/main/java/org/apache/activemq/network/NetworkBridgeFactory.java:39	0.002316471	public static DemandForwardingBridge createBridge(NetworkBridgeConfiguration config,
activemq-core/src/main/java/org/apache/activemq/network/NetworkBridgeFactory.java:28	0.001432665	private NetworkBridgeFactory() {
activemq-core/src/main/java/org/apache/activemq/network/NetworkBridgeFactory.java:69	0.000805283	return result;
activemq-core/src/main/java/org/apache/activemq/network/NetworkBridgeFactory.java:60	0.00049975	} else {
activemq-core/src/main/java/org/apache/activemq/network/NetworkBridgeFactory.java:63	0.00049975	} else {

(a) The ranking list sorted by our method with Barinel. The red background indicates the actual buggy lines.

predicted_buggy_lines	predicted_score	src
activemq-core/src/main/java/org/apache/activemq/network/NetworkBridgeFactory.java:41	12	return createBridge(config, localTransport, remoteTransport, null);
activemq-core/src/main/java/org/apache/activemq/network/NetworkBridgeFactory.java:57	9	if (configuration.isConduitSubscriptions()) {
activemq-core/src/main/java/org/apache/activemq/network/NetworkBridgeFactory.java:58	9	if (configuration.isDynamicOnly()) {
activemq-core/src/main/java/org/apache/activemq/network/NetworkBridgeFactory.java:66	6	if (listener != null) {
activemq-core/src/main/java/org/apache/activemq/network/NetworkBridgeFactory.java:69	2	return result;
activemq-core/src/main/java/org/apache/activemq/network/NetworkBridgeFactory.java:63	1	} else {
activemq-core/src/main/java/org/apache/activemq/network/NetworkBridgeFactory.java:60	1	} else {
activemq-core/src/main/java/org/apache/activemq/network/NetworkBridgeFactory.java:39	12	public static DemandForwardingBridge createBridge(NetworkBridgeConfiguration config,
activemq-core/src/main/java/org/apache/activemq/network/NetworkBridgeFactory.java:53	12	public static DemandForwardingBridge createBridge(NetworkBridgeConfiguration configuration,
activemq-core/src/main/java/org/apache/activemq/network/NetworkBridgeFactory.java:64	12	result = new DemandForwardingBridge(configuration, localTransport, remoteTransport);
activemq-core/src/main/java/org/apache/activemq/network/NetworkBridgeFactory.java:59	12	result = new ConduitBridge(configuration, localTransport, remoteTransport);
activemq-core/src/main/java/org/apache/activemq/network/NetworkBridgeFactory.java:61	12	result = new DurableConduitBridge(configuration, localTransport, remoteTransport);
activemq-core/src/main/java/org/apache/activemq/network/NetworkBridgeFactory.java:19	6	import org.apache.activemq.transport.Transport;
activemq-core/src/main/java/org/apache/activemq/network/NetworkBridgeFactory.java:67	6	result.setNetworkBridgeListener(listener);
activemq-core/src/main/java/org/apache/activemq/network/NetworkBridgeFactory.java:17	5	package org.apache.activemq.network;
activemq-core/src/main/java/org/apache/activemq/network/NetworkBridgeFactory.java:26	4	public final class NetworkBridgeFactory {
activemq-core/src/main/java/org/apache/activemq/network/NetworkBridgeFactory.java:28	4	private NetworkBridgeFactory() {
activemq-core/src/main/java/org/apache/activemq/network/NetworkBridgeFactory.java:40	4	Transport localTransport, Transport remoteTransport) {
activemq-core/src/main/java/org/apache/activemq/network/NetworkBridgeFactory.java:54	4	Transport localTransport, Transport remoteTransport,
activemq-core/src/main/java/org/apache/activemq/network/NetworkBridgeFactory.java:55	3	final NetworkBridgeListener listener) {
activemq-core/src/main/java/org/apache/activemq/network/NetworkBridgeFactory.java:56	3	DemandForwardingBridge result = null;

(b) The ranking list sorted by GLANCE-LR. The red background indicates the actual buggy lines.

Fig. 1: An real example to show the results of ranking all lines in NetworkBridgeFactory.java

C. IFA improvement on Non-Effective projects

Motivation and Approach. We further analyze IFA results with and without Causal Analysis. As shown in Table VI of RQ4 (Page 9 of the paper), for the top/bottom 25% releases with the best/worst IFA, we observe (a) for the top 25%, IFA performance without Causal Analysis has already been pretty good, e.g., mean values are less than 2 for all formulas, which cause adding Causal Analysis does not improve IFA values; (b) for the bottom 25%, IFA performance essentially increases by adding Causal Analysis, e.g., median values of Tarantula is from 178 to 17. In this section, we extend the results listed in Table VI of RQ4 (Page 9 of the paper) by calculating the increasing percentage after adding Causality Analysis on Non-Effective projects (i.e., the bottom 25% projects with worse IFAs).

Results. Table II shows the results of the increasing percent after adding Causality Analysis on Non-Effective projects. As can be seen in Table II, IFA performance essentially increases by adding Causal Analysis on Non-Effective projects, e.g., median values of Tarantula is from 178 to 17, i.e., increase 90.4% ((178-17)/178).

TABLE II: The increasing percent of IFA On Non-effective set after adding Causality Analysis (CA).

	Formula	Mean	Median
Non-Effective	Barinel	1.7%	30.8%
	Dstar	10.3%	27.1%
	Ochiai	17.0%	67.2%
	Op2	18.8%	49.7%
	Tarantula	-5.1%	90.4%

Conclusion. IFA performance essentially increases by adding Causal Analysis on Non-Effective projects.

D. Statistical tests

Motivation and Approach. We follow the statistical test applied in GLANCE [11], i.e., employ the Wilcoxon-signed Test and Cliff's Delta to compare our method with the baseline methods under all studied projects (similar to Table 7 in [11]). We also calculate the W/T/L numbers for comparison. If **BH-corrected** p values of the Wilcoxon-signed Test are smaller than 0.05 and the Cliff's Delta values are non-negligibly better

TABLE III: The test results from the statistical analysis between SOUND and the baselines under the cross-release scenario. By convention, the magnitude of the difference (Cliff's Delta) is considered negligible (short for N, $|\delta| < 0.147$), small (S, $0.147 \leq |\delta| < 0.33$), median (M, $0.33 \leq |\delta| < 0.474$), or large (L, $|\delta| > 0.474$).

			GLANCE-LR	GLANCE-EA	GLANCE-MD	LineDP	DeepLineDP	N-gram	ErrorProne
IFA (↓)	p-value	Barinel	0.000	0.000	0.000	0.000	0.000	0.000	0.000
		Dstar	0.314	0.009	0.057	0.000	0.000	0.279	0.000
		Ochiai	0.000	0.000	0.000	0.000	0.000	0.000	0.000
		Op2	0.019	0.301	0.401	0.026	0.000	0.766	0.000
		Tarantula	0.000	0.000	0.000	0.000	0.000	0.000	0.000
	Cliff-delta	Barinel	-0.588(L)	-0.691(L)	-0.679(L)	-0.709(L)	-0.853(L)	-0.759(L)	-0.846(L)
		Dstar	0.045(N)	-0.189(S)	-0.196(S)	-0.213(S)	-0.559(L)	-0.181(S)	-0.575(L)
		Ochiai	-0.354(M)	-0.513(L)	-0.502(L)	-0.515(L)	-0.753(L)	-0.551(L)	-0.752(L)
		Op2	0.114(N)	-0.113(N)	-0.131(N)	-0.152(S)	-0.474(L)	-0.108(N)	-0.502(L)
		Tarantula	-0.446(M)	-0.596(L)	-0.585(L)	-0.599(L)	-0.830(L)	-0.650(L)	-0.827(L)
Recall@20% (↑)	p-value	Barinel	0.000	0.000	0.000	0.000	0.000	0.000	0.000
		Dstar	0.000	0.000	0.000	0.000	0.000	0.000	0.000
		Ochiai	0.000	0.000	0.000	0.000	0.000	0.000	0.000
		Op2	0.000	0.000	0.000	0.000	0.000	0.000	0.000
		Tarantula	0.000	0.000	0.000	0.000	0.000	0.000	0.000
	Cliff-delta	Barinel	0.408(M)	0.799(L)	0.815(L)	0.647(L)	0.942(L)	0.932(L)	0.944(L)
		Dstar	0.081(N)	0.697(L)	0.748(L)	0.314(S)	0.938(L)	0.927(L)	0.936(L)
		Ochiai	0.263(S)	0.772(L)	0.798(L)	0.510(L)	0.940(L)	0.929(L)	0.941(L)
		Op2	0.072(N)	0.696(L)	0.752(L)	0.297(S)	0.941(L)	0.930(L)	0.939(L)
		Tarantula	0.326(S)	0.792(L)	0.813(L)	0.571(L)	0.942(L)	0.932(L)	0.943(L)
Effort@20% (↓)	p-value	Barinel	0.000	0.000	0.000	0.000	0.000	0.000	0.000
		Dstar	0.000	0.000	0.000	0.000	0.000	0.000	0.000
		Ochiai	0.000	0.000	0.000	0.000	0.000	0.000	0.000
		Op2	0.000	0.000	0.000	0.000	0.000	0.000	0.000
		Tarantula	0.000	0.000	0.000	0.000	0.000	0.000	0.000
	Cliff-delta	Barinel	-0.809(L)	-0.907(L)	-0.921(L)	-0.797(L)	-0.949(L)	-0.948(L)	-0.947(L)
		Dstar	-0.307(S)	-0.847(L)	-0.873(L)	-0.255(S)	-0.946(L)	-0.941(L)	-0.942(L)
		Ochiai	-0.638(L)	-0.896(L)	-0.897(L)	-0.588(L)	-0.947(L)	-0.945(L)	-0.945(L)
		Op2	-0.221(S)	-0.802(L)	-0.849(L)	-0.158(S)	-0.946(L)	-0.941(L)	-0.941(L)
		Tarantula	-0.707(L)	-0.908(L)	-0.912(L)	-0.662(L)	-0.949(L)	-0.948(L)	-0.947(L)

($\delta > 0.147$), we mark our methods as Win (W). Note that for IFA and Effort@20%, **smaller** values mean better, i.e., more **negative** Cliff's Delta means our method performs much better than baselines. The definitions of Tie (T) and Loss (L) are similar.

Results. Table III presents the results of the Wilcoxon-signed Test and Cliff's Delta. The gray background indicates that (a) BH-corrected p values of the Wilcoxon-signed Test are smaller than 0.05, and (b) the Cliff's Delta values are not negligible. Table IV summarizes the W/T/L results of our methods with five formulas compared with all baselines.

As can be seen from Tables III and IV, our methods with five formulas significantly overcome baselines in most cases, e.g., our methods with Barinel, Ochiai, and Tarantula, perform significantly better than all baselines (i.e., with 7 Wins) under three indicators.

Conclusion. According to the result of statistical tests, our methods with five formulas significantly overcome baselines in most cases.

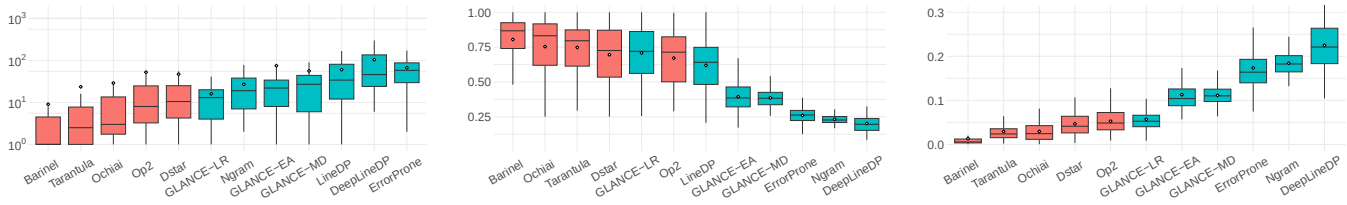
TABLE IV: The results of W/T/L

	IFA	Recall@20%LOC	Effort@20%Recall
Barinel	7/0/0	7/0/0	7/0/0
Dstar	4/3/0	6/1/0	7/0/0
Ochiai	7/0/0	7/0/0	7/0/0
Op2	3/4/0	6/1/0	7/0/0
Tarantula	7/0/0	7/0/0	7/0/0

E. Performance under other prediction settings.

Motivation and Approach. We use the **SPR** (Single Prior Release) before the predicted release as the training set since there may be **concept drift** [48,49] between too-old previous releases and the predicted release, i.e., the distribution of buggy lines in the old releases deviates from the prediction release. Therefore, introducing too old releases may not necessarily enhance prediction performance. In other words, using the single release before the predicted release would reduce the risk of concept drift.

In the discussion part (see Section **VI.C**, Page 10 of the paper), we have presented the result of our method under the **APR** (All Prior Releases) scenario, i.e., for release i in



(a) IFA (↓) boxplots in the increasing order (b) Recall@20%(↑)boxplots in the decreasing order (c) Effort@20% (↓) boxplots in the increasing order

Fig. 2: The Boxplots of our methods marked red and baseline methods marked green under **MW** (Moving Window) scenario, where ↑/↓ indicates that a higher/lower value of the metric is better.

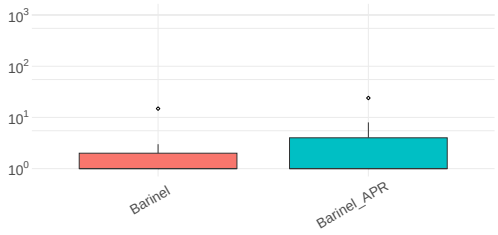
the studied projects, we employ all previous releases, releases $1, \ldots, i-1$, as the training set to construct the prediction models. In this section, we compare our method with baselines under the **MW** (Moving Window) scenario, i.e., for release i, we employ three previous releases, $i-3, i-2, i-1$, as the training set. Here, we choose the length of the **MW** as 3 ($l = 3$), which we consider proper. Among our studied dataset, one project has only two releases, and two projects have only three releases. These three projects are excluded due to the Moving Window with $l = 3$. If we choose a larger l value, more projects will be excluded, e.g., there are three projects with only four releases.

Results. Figure 2 shows the comparison results between our method and baselines under the **MW** scenario. As can be seen in Figure 2, our methods still perform better than baselines: for IFA, top-5 are all our methods; for Recall@20%, top-4 are our methods; and for Effort@20%, top-5 are all our methods.

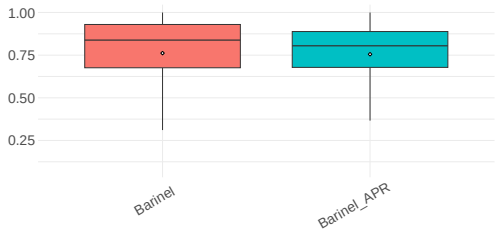
Figure 3 shows the comparison results between our `Barinel`-based method[2] with the default SPR (i.e., "using the single prior release") setting and the APR (i.e., "using all prior releases") setting. In Figure 3, the white diamond points represent the mean values. As can be seen in Figure 3, the performance under the default SPR scenario is slightly better than APR scenario: for IFA, the SPR scenario has better mean values and the same median values compared with the APR scenario; for Recall@20%, the SPR scenario has better mean and median values than the SPR scenario; for Effort@20%, the SPR scenario has worse mean values and better median values than APR scenario.

Figure 4 shows the comparison result between our `Barinel`-based method under the default SPR (i.e., "using the single prior release") setting and the MW (i.e., "moving window") setting. Since some releases are excluded under the MW setting (e.g., Flink has only two releases, which have been excluded), we compare the SPR and MW scenarios using the same set of releases as the testing set, which is a subset (**86 out of 123** predicted releases) of that utilized under the APR scenario, leading to the changes of the boxplots of `Barinel`-based results under SPR between Figures 3 (results of 123 releases) and 4 (results of 86 releases).

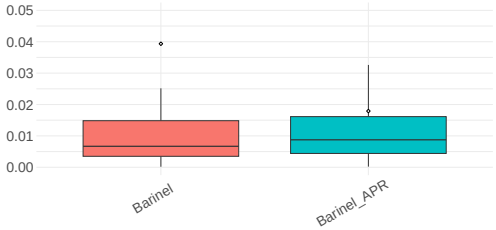
[2]As shown in RQ1 and Figure 4 in the paper, the `Barinel`-based method has performed the best. We consider `Barinel` as the default formula in our method.



(a) IFA (↓) boxplots in the increasing order



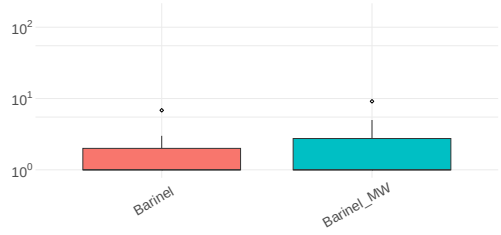
(b) Recall@20%LOC (↑) boxplots in the decreasing order



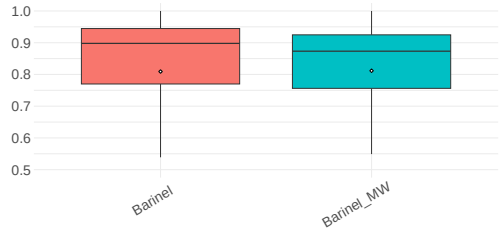
(c) Effort@20%Recall(↓) boxplots in the increasing order

Fig. 3: The Boxplots of our `Barinel`-based method under the **SPR** setting marked red and the **APR** setting marked green, where ↑/↓ indicates that a higher/lower value of the metric is better.

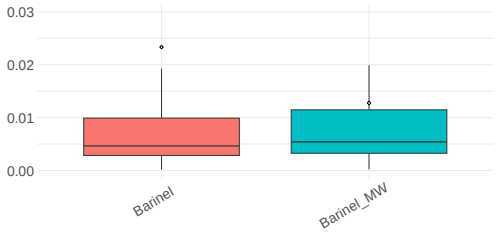
As can be seen in Figure 4, the performance under the default SPR scenario is also slightly better than MW scenario: for IFA, the SPR scenario has better mean values and the same median values; for Recall@20%, the SPR scenario has better median values and slightly worse mean values than the MW scenario; for Effort@20%, the SPR scenario has worse mean values and better median values than the MW scenario.



(a) IFA (↓) boxplots in the increasing order



(b) Recall@20%LOC (↑) boxplots in the decreasing order



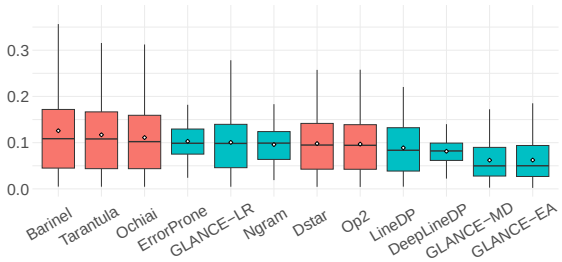
(c) Effort@20%Recall(↓) boxplots in the increasing order

Fig. 4: The Boxplots of our `Barinel`-based method under the **SPR** setting marked red and the **MW** setting marked green, where ↑/↓ indicates that a higher/lower value of the metric is better.

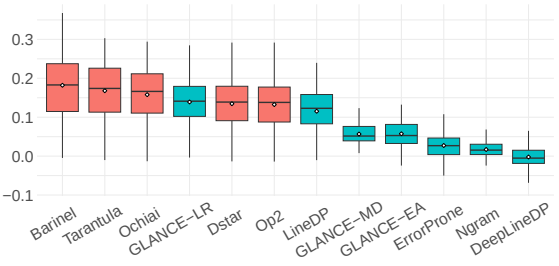
Conclusion. The performance under the **SPR** scenario is slightly better than new **APR** and **MW** scenarios.

F. Performance under other classification metrics

Motivation and Approach. In the paper, we formulate line-level prediction as a ranking task, and classification measures do not apply to our line-level defect prediction. We aim to check how our method, SOUND, performs under classification indicators. Since the results under three classification indicators, AUC, D2H, and FAR, have been added in the discussion part (see Section **VI.D**, Page 10 of the paper), we conduct experiments under **F1-score** and **MCC** here to check the performance of our method and baselines. As the metrics are classification indicators, we must set the threshold. Since 20%LOC is used in our paper as a cutoff point for ranking indicators, we continue to employ 20%LOC as the threshold for classification. Our experiment first ranks all lines in a release from the highest to lowest predicted values. Then, we classify the top 20% lines in the ranking list as **defective** and the remaining modules as **clean**.



(a) F1-score (↑) boxplots in the decreasing order



(b) MCC (↑) boxplots in the decreasing order

Fig. 5: The Boxplots of our methods marked red and baseline methods marked green, where ↑/↓ indicates that a higher/lower value of the metric is better.

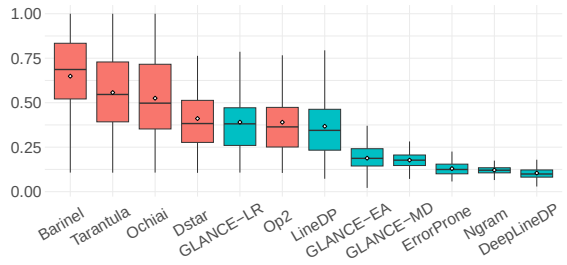
Results. Figure 5 shows the boxplots of the values of F1-score and MCC for our method and the baseline methods of prediction results of 123 release pairs, sorted by their *median* values. As seen in Figure 5, the top 3 are always our methods (`Barinel`, `Tarantula`, and `Ochiai`) under the two indicators.

Conclusion. Our methods perform better than baselines under the classification indicators.

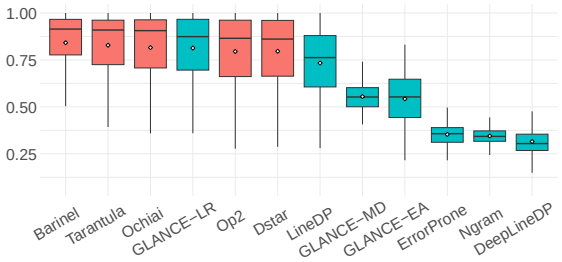
G. Performance under other effort-aware metrics.

Motivation and Approach. As 20% is a widely used inspection threshold, we choose Recall@20%LOC as one of our indicators. We aim to check the performance of our method under additional cost thresholds. Specifically, we conducted experiments to evaluate Recall@X%LOC (the percentage of total defective lines found within the first X% of code lines in a given release), where X is from 10 to 90 in intervals of 10. We present the results in Section **VI.E** (Page 10) of the paper with the line chart. In this section, we compare our methods and the baselines by evaluating the boxplots of Recall@10%LOC and Recall@30%LOC.

Results. Figure 6 presents the boxplots of the values of Recall@10%LOC and Recall@30%LOC for our method and the baseline methods of prediction results of 123 release pairs. As shown in Figure 6, four of our methods (`Barinel`, `Tarantula`, `Ochiai`, and `Dstar`) perform better than all the baselines under Recall@10%LOC, and three of our methods (`Barinel`, `Tarantula`, and `Ochiai`) outperform under Recall@30%LOC.



(a) Recall@10%LOC (↑) boxplots in the decreasing order



(b) Recall@30%LOC (↑) boxplots in the decreasing order

Fig. 6: The Boxplots of our methods marked red and baseline methods marked green, where ↑/↓ indicates that a higher/lower value of the metric is better.

Conclusion. Our method performs better than the baselines under other effort-aware metrics: Recall@10% and Recall@30%.