Identifying Emerging Smells in Software Designs based on Predicting Package Dependencies

Antonela Tommasel[a,*], J. Andres Diaz-Pace[a]

[a]*ISISTAN, CONICET-UNICEN, Argentina*

Abstract

Software systems naturally evolve, and this evolution often brings design problems that contribute to system degradation. Architectural smells are typical symptoms of such problems, and several of these smells are related to undesired dependencies among packages. The early detection of smells is essential for software engineers to plan ahead for maintenance or refactoring efforts. Although tools for identifying smells exist, they detect the smells once they exist in the source code when their undesired dependencies are already created. In this work, we explore a forward-looking approach for identifying smells that can emerge in the next system version based on inferring package dependencies that are likely to appear in the system. Our approach takes the current design structure of the system as a network, along with information from previous versions, and applies link prediction techniques from the field of social network analysis. In particular, we consider a group of smells known as instability smells (cyclic dependency, hub-like dependency, and unstable dependency), which fit well with the link prediction model. The approach includes a feedback mechanism to progressively reduce false positives in predictions. An evaluation based on six open-source projects showed that, under certain considerations, the proposed approach can satisfactorily predict missing dependencies and smell configurations thereof. The feedback mechanism led to improvements of up to three times the initial precision values. Furthermore, we have developed a tool for practitioners to apply the approach in their projects.

*Corresponding author

Email addresses: antonela.tommasel@isistan.unicen.edu.ar (Antonela Tommasel), andres.diazpace@isistan.unicen.edu.ar (J. Andres Diaz-Pace)

Keywords: Package Dependencies, Architectural Smells, Link Prediction, Cycles, Machine Learning

1. Introduction

Software systems naturally evolve due to changes in their requirements and operating environment, which leads to new design decisions and source code modifications. In light of these changes, engineers need to ensure a graceful system evolution. Nonetheless, as a system evolves, the amount and complexity of the interactions among its software elements are likely to increase, affecting the system design structure (Hochstein and Lindvall, 2005; de Silva and Balasubramaniam, 2012). Although functional dependencies are necessary, evolution often brings undesired dependencies among modules (e.g., system packages) (Hochstein and Lindvall, 2005; de Silva and Balasubramaniam, 2012). Architectural smells (ASs) (Garcia et al., 2009) are system configurations that usually hint at degradation problems at the design level. A typical example are cycles (Melton and Tempero, 2007), which involve a particular pattern of undesired dependencies among packages. This work focuses on the so-called instability smells (Sas et al., 2019), which describe dependency-based configurations that are likely to produce rippling effects upon changes, compromising the system's maintainability and testability. The early detection of such smells is important for engineers to spot degradation trends in a system and evaluate repairing actions.

There are several tools for managing system dependencies and detecting some types of smells. These tools typically extract information about software elements and their dependencies from source code and compute certain metrics, which altogether serve to identify candidate ASs in the system. However, a limitation of existing tools is that they are reactive, as they can detect a smell once it exists in the code. However, developers might be reluctant to fix problems once they are already in the code. In the case of instability smells, preventing them often requires a global design assessment to detect package configurations that might become smelly in the near future. This is a tedious and error-prone activity if performed manually. Thus, we argue for the anticipation (i.e., prediction) of likely ASs to provide engineers with insights into the health of the current software architecture and the places where degradation is expected to appear. Indeed, the predicted ASs should be interpreted as system trends and not as definite problems.

In this work, we present a proactive approach that leverages link prediction (LP) techniques (Liben-Nowell and Kleinberg, 2007) for inferring (future) package dependencies for a given system version and then predicting possible ASs configurations in the inferred design structure. This approach is a continuation of prior work (Díaz-Pace et al., 2018), in which we investigated the usage of supervised classification techniques for predicting package dependencies for the next system version. We have now evolved our approach to include a feedback mechanism that considers the evolution of package dependencies from one system version to the next one and consequently adjusts the ranking of ASs presented to engineers. An experimental evaluation with three types of smells and six open-source projects from the literature shows that this mechanism reduces the number of false positives over time, leading to higher precision in the prediction of ASs. The main contribution of the approach lies in the feasibility of predicting relevant smells that, if left unattended by engineers, might lead to high maintenance costs or system degradation in the future. We have also built a tool, called `ASPredictor`, to support the approach and make it practical for engineers and researchers.

The rest of the article is organized into six sections. Section 2 gives background information about ASs and motivates the prediction of dependencies for smell configurations. Section 3 discusses how the prediction of package dependencies can be cast as a link prediction problem. Section 4 presents the main building blocks of our prediction approach. Section 5 describes a series of experiments with six Java open-source systems for evaluating the approach and discusses the main results and lessons learned. Section 6 covers related work. Finally, Section 7 presents the conclusions and outlines future work.

2. Dependencies and Architectural Smells

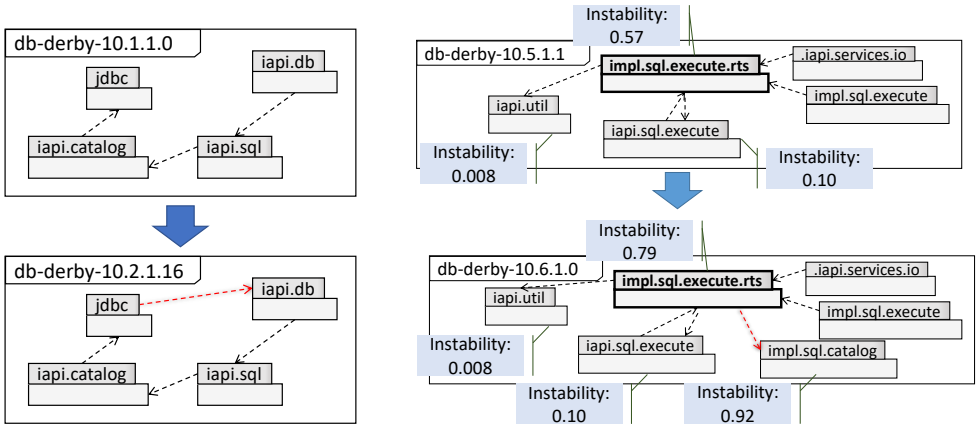
Software systems often exhibit design problems, which can be introduced either during development or along with their evolution. These problems harm the system's quality and make maintenance difficult, as they degrade its design structure. Architectural smells (ASs) can be defined as a combination of design constructs that often indicate modifiability problems (Garcia et al., 2009). Different ASs have been cataloged in the literature (Garcia et al., 2009; Tracz, 2015; Marinescu, 2012a). Of particular relevance to this work are the so-called dependency-based smells (Garcia et al., 2009), which involve interactions among system elements (e.g., packages). These smells

occur when one or more elements violate design principles or rules, often manifesting as undesired dependencies in the module structure (Hochstein and Lindvall, 2005).

In this work, we tackle a group of dependency-based ASs known as instability smells (Fontana et al., 2016), namely: Cyclic Dependency (CD), Hub-like Dependency (HLD) (Tracz, 2015), and Unstable Dependency (UD) (Marinescu, 2012a). These smells can considerably impact on maintenance efforts (Sas et al., 2019). We focus on packages as the unit of analysis, as the package structure of a system provides an approximate view of its modules, which often reflects the decomposition criteria followed by the developers. Furthermore, instability smells among packages tend to persist longer across system versions than smells among classes (Sas et al., 2019) and might move to central parts of the system design. For this reason, we argue that these smells at the package level are a good target for degradation trends. Next, we discuss the characteristics of the smells and their detection strategies for Java packages.

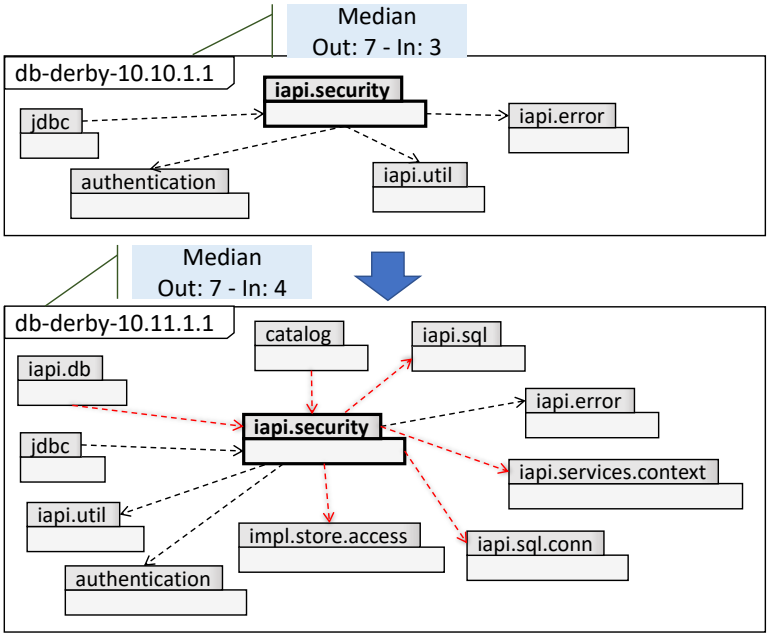
Cyclic Dependency. In this smell, various elements directly or indirectly depend on each other to function correctly. For example, Figure 1a depicts a cycle among four packages in Apache Derby, which appeared in version `10.2.1.16` due to the dependency between packages `jdbc` and `iapi.db`. The packages are connected through usage relations (dependencies). In general, the chain of dependencies among packages breaks the desirable acyclic nature of a sub-system's dependency structure. Thus, the elements involved in a cycle can be hard to maintain, test or reuse in isolation. The typical strategies for detecting cycles are based on the DFS algorithm for graphs (Fontana et al., 2017).

Hub-like Dependency. This smell arises when an element has a large number of incoming and outgoing dependencies. The central element in the smell is called a hub. The strategy for detecting hubs (Tracz, 2015) first computes the median of all packages' incoming and outgoing dependencies. Then, for each package, it checks if both its incoming and outgoing dependencies are greater than the incoming and outgoing medians, respectively, and finally checks whether the difference between incoming and outgoing dependencies is smaller than a fraction of the total dependencies of that package. This difference is referred to as the "hubness" of the package. For example, Figure 1c shows that package `org.apache.derby.catalog.types` uses 7 packages and



(a) Cyclic Dependency

(b) Unstable Dependency



(c) Hub-like Dependency

Figure 1: Examples of Architectural Smells (Apache Derby)

is used by 5 other packages in version `10.11.1.1`, which turns the package into a hub. The emergence of this hub is due to additional package dependencies with respect to version `10.10.1.1`.

Unstable Dependency. Based on the well-known Instability metric (Marinescu, 2012b), this smell refers to an element that depends on other elements that are less stable than itself. Thus, the affected element might cause a ripple effect of changes in the system. The strategy for detecting unstable elements relies on the computation of the Instability metric, which was originally defined for dependencies among classes. When extending this definition to packages, we need to consider that packages might be coupled due to dependencies with multiple classes and include the number of dependencies in the metric. Once the metric is computed, a package is deemed unstable if it depends upon other packages with higher Instability values. The more dependencies a package (affected by the smell) has with other unstable packages, the stronger the smell is (Fontana et al., 2016). For example, Figure 1b shows that `org.apache.derby.impl.sql.execute.rts` becomes unstable in version `10.6.1.0`, due to changes in its dependencies, which alter its Instability score. As a result, the package ends up depending on a package (`org.apache.derby.impl.sql.catalog`) with a higher Instability score.

2.1. Evolution of smells

Architectural smells present different behaviors in the evolution of a system. Some ASs appear in the system from the initial versions, while others emerge at specific points in time. ASs can also grow, in terms of their severity (i.e., number of affected elements or number of attached dependencies), from one version to another. For instance, one might consider that a CD smell worsens when the number of packages in the cycle increases. Figure 2 illustrates the evolution of CD smells for 11 versions of Apache Camel, in which each row represents a different cycle. Note that several cycles emerge on intermediate versions rather than on the initial version. The smell can also disappear, either due to a functional change in the system or a refactoring action.

We hypothesize that ASs can emerge (in a future version) due to specific dependency patterns in the package structure. We refer to these likely ASs as "*quasi-smells*", as their appearance is conditional to the evolution of certain undesirable dependencies among packages. When these dependencies inadvertently appear in the source code (from one system version to another),

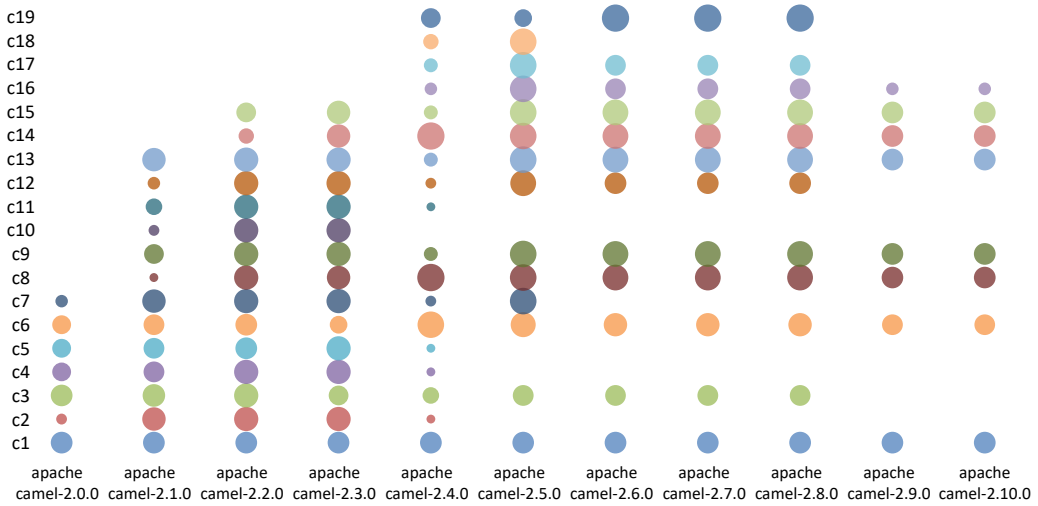


Figure 2: Evolution of cycles in Apache Camel (the circle diameter is proportional to the cycle size).

the smelliness of some packages intensifies, and they turn into actual smells. In other words, the difference between a quasi-smell and a smell lies in the number and configuration of package dependencies, which enables the detection strategies to flag the smells. This emergence of ASs is exemplified in Figure 1. Since quasi-smells can be seen as "seeds" for design problems in the system, forecasting the emergence of the corresponding smells is helpful for engineers as early quality warnings. For instance, engineers can decide to prevent specific ASs from happening (when they are still quasi-smells) or limit their adverse effects on the system.

Several tools currently provide detection capabilities for dependency-based ASs, such as Sonargraph, LattixDSM, HotspotDetector (Mo et al., 2015), or Arcan (Fontana et al., 2017). However, these tools are reactive, as they work when the smells are already realized in the code. In this scenario, removing a detected smell (e.g., via refactoring) might not be straightforward for developers because they need to invest additional efforts and guarantee that the system will continue to function correctly. Our approach aims to develop a proactive tool for spotting quasi-smells in the package structure, which have high chances of becoming ASs due to the addition of a few (undesired) dependencies in upcoming system versions. The prediction of such ASs is based on inferring likely package dependencies.

3. Link Prediction Techniques

A key observation about the appearance of dependencies contributing to the formation of ASs is that, despite changes occurring at the class level, the package structure remains more or less stable across system versions while dependencies among packages keep being added. There are, of course, exceptions in which the number of packages varies from one version to another, for example, across initial systems versions where the main structure and functionality are still fleshed out or versions in which refactoring takes place. Thus, we argue that the package structure and its evolution over time give the basis for predicting which dependencies are likely to appear between pairs of packages in the future.

Link Prediction (LP) adapts Social Network Analysis (SNA) techniques for studying to what extent the evolution of a network can be modeled using its intrinsic *features* (Liben-Nowell and Kleinberg, 2007). This task involves inferring "missing" links between pairs of nodes in a network based on the observable interactions (or links) among nodes and node attributes. As software systems comprise elements that interact with each other, it is natural to model systems (or their views) as graphs. However, software design networks could have a different dynamic than traditional social networks. A prerequisite for applying LP in our approach is transforming the system under analysis into a dependency graph. More formally, a dependency graph is a graph $DG\,(V, E)$, where each node $v \in V$ represents a module, and each edge (or link) $e\,(v, v') \in E$ represents a dependency from node v to v'. Since we deal with Java systems, nodes correspond to packages while edges represent the usage relations between those packages. In principle, each system package can be regarded as a different module. Nonetheless, this assumption might not hold in all systems (e.g., different sub-packages might belong to the same conceptual module).

The LP task takes a $DG_n\,(V, E)$ at time n, and then infers the edges that will be added to $DG_{n+1}\,(V, E)$ at time $n + 1$. Let U be the set of all possible edges among nodes in $DG_n\,(V, E)$. The LP task generates a list R of all possible edges in $U - E$, and then indicates whether each edge (in R) will be present in $DG_{n+1}\,(V, E)$.

Link Prediction in social networks is based on the principle of *homophily* (Liben-Nowell and Kleinberg, 2007), which states that interactions between similar individuals occur at a higher rate than those among dissimilar ones. In our context, this principle implies that similar packages (according

to some criteria) have a higher chance of establishing dependencies than dissimilar packages. In general, LP techniques assess node similarity based on topological and content-based features.

3.1. Applying Classification Techniques

Although intuitive, the homophily principle does not always hold for complex networks, such as those based on software-related dependencies (Zhou et al., 2014). For instance, two similar packages can intentionally be designed to not become dependent on each other based on business logic or modularity reasons. Moreover, dependencies might still appear between dissimilar packages. Thus, we argue that the LP task applied to software networks should be able to learn "exceptions" to the homophily principle. These exceptions should tell whether two packages would connect at version $n+1$. In particular, we cast LP as a binary classification problem and use a Machine Learning (ML) approach to learn from both the existence and absence of relations between the different pairs of packages in the dependency graph. Furthermore, this approach allows taking the history of the dependency graph into account (i.e., the graphs corresponding to previous system versions) to enhance the classification model.

The classifier to be trained receives as input a *dataset* consisting of a set of instances that are derived from the package dependency graph. Each instance consists of a given pair of nodes, a list of features characterizing it, and a label (or class) that indicates whether a dependency exists between the given pair of nodes. The pairs of connected nodes belong to the *positive class*, while the pairs of unconnected nodes to the *negative class*. This tabular representation is the input to build the ML models.

In previous work (Díaz-Pace et al., 2018), we assessed the predictive power of LP techniques for inferring package dependencies. The initial results using only topological features showed that classification models could provide reasonable predictions, which improved when including content-based features. For this reason, instances in this work are characterized by both topological and content-based features[1].

[1]The complete list of features is listed in the companion repository: `https://github.com/tommantonela/ASPredictor/wiki/Appendix:-Similarity-Metrics-Included`

3.2. Topological and Content-based Features

Topological features are related to the graph structure and the role that nodes and edges play in that structure (Liben-Nowell and Kleinberg, 2007). For instance, *Common Neighbors* is defined as the number of common adjacent nodes (i.e., neighbors) that two nodes have in common, aiming at capturing the notion that two unrelated elements sharing neighbors would be "introduced" to each other at a posterior time. In this context, we selected a number of topological similarity metrics (Deza and Deza, 2006), namely: Common Neighbors, Salton, Sorensen, Adamic-Adar, Katz, SimRank, Russell-Rao, and Resource Allocation. The selected metrics leverage either on local or global characteristics of the dependency graph (Lu and Zhou, 2011). On one hand, local similarity metrics (such as Common Neighbors or Adamic-Adar) are concerned with the neighborhood of nodes (i.e., packages). On the other hand, global metrics (such as Katz) are concerned with the whole network structure. As a result, global metrics can provide more accurate predictions than local ones, at the expense of higher computational complexity. The selected metrics have already been used for assessing the structural similarity of source code entities with satisfactory results (Terra et al., 2013).

Palomba et al. (2014) have indicated that content-based metrics are orthogonal to structural ones. In other words, using content-based information allows identifying specific properties of software elements that would be missed if only structural or topological information were considered. In this sense, content-based features are often used to complement topological features. For instance, we can leverage features derived from source code elements, such as classes or packages, which can be seen as textual artifacts containing identifiers, comments, parameters, and method names. Hence, they can be considered textual documents, and Natural Language Processing techniques can be applied to extract lexical properties from source code (Palomba et al., 2014). For example, a simple strategy is to compute the lexical overlap between the texts. To this end, texts are transformed into their bag-of-words (BoW) representations (Salton and McGill, 1986), which can be built by considering different aspects of the original texts.

In our domain, we can think of each Java class c as a BoW containing the most representative tokens that characterize its source code, for example the names of the field attributes of the classes, the names of the declared methods or the class comments and documentation, which could all lead to different degrees of similarity. This way, BoW can provide *content-based*

representations for assessing the similarity among classes. Furthermore, class representations can be easily extended to packages. The representation of a package p is defined as the (recursive) union of the BoW representations of all the classes contained by p. In our evaluation, packages were textually represented by the names of the field attributes, the name of the methods, the method invocations, the name of parameters, the comments and the Javadoc documentation. Then, similarity is assessed using the *Cosine Similarity*.

4. Approach

The LP task for predicting ASs is achieved in three phases, as depicted in Figure 3. First, a prediction phase seeks to infer the appearance of new dependencies in the following system version. Second, emerging smells of different types are identified in a filtering phase. These smells are ranked according to their characteristics and the confidence scores of the predicted dependencies (from the first phase). Third, once the next version is known, the confidence of the predicted dependencies is updated based on a learning automaton (LA) (Moradabadi and Meybodi, 2017). A round of prediction for ASs is implemented by two procedures, as depicted in Algorithms 1 and 2. Each phase is described in detail in the following sub-sections.

In addition, we have developed a prototype called `ASPredictor`[2] that supports the prediction of instability smells and evaluates those predictions using a sequence of system versions.

4.1. Phase 1: Prediction

Initially, individual dependencies are inferred based on training a binary classification model. To do so, the dependency graphs corresponding to the current (G_n) and previous (G_{n-1}) versions are required as inputs. The output of this phase is the set of dependencies that are likely to appear in the next version G^*_{n+1} and their confidence (lines $1-3$ of Algorithm 1). Note that this phase is smell-independent, as we only identify dependencies that might prefigure different ASs in the second phase. For instance, in Figure 3, dependencies B-D, F-D, F-A, and C-E were predicted for G^*_{n+1} by looking at graph information from G_{n-1} and G_n.

This phase involves three steps. In step 1, an instance-based representation of each dependency graph is constructed (as presented in Section 3.1)

[2]Available at: `https://github.com/tommantonela/ASPredictor`

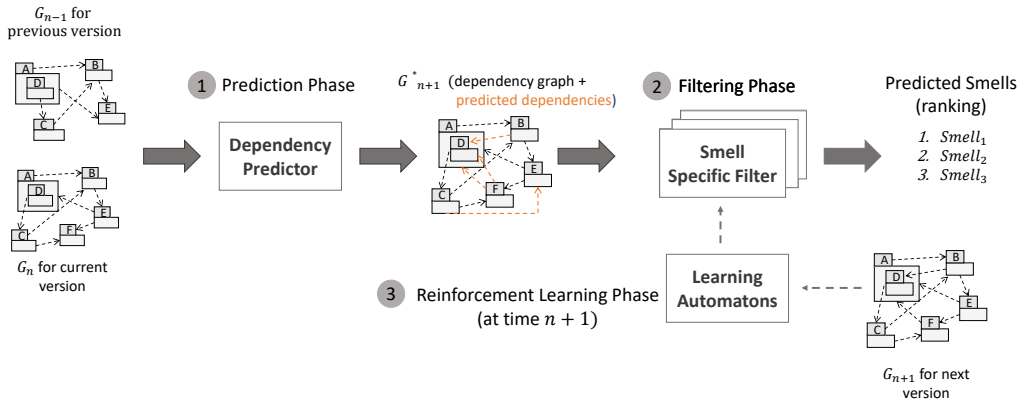


Figure 3: Overview of the Approach

Algorithm 1 Prediction of smells for the next version

Inputs: $G_n(V_n, E_n)$: graph for current version n
$G_{\text{n-1}}(V_{\text{n-1}}, E_{\text{n-1}})$: graph for previous version $n-1$
DL : list of dependencies (edges) from previous versions, including confidence scores

Outputs: $smellsCD$: ranking of predicted CD smells
$smellsHLD$: ranking of predicted HLD smells
$smellsUD$: ranking of predicted UD smells

1: $classifier \leftarrow buildClassificationModel(G_{\text{n-1}}, G_n)$ {The classifier is also configured with topological and content-based features}
2: $DL \leftarrow predictDependencies(classifier, G_n, DL)$ {DL contains the predicted dependencies with adjusted confidence scores, according to FL}
3: $G^*_{\text{n+1}} \leftarrow (V_n, E_n \cup DL)$ {The predicted graph at version $n+1$}
4: $smellsCD \leftarrow filterCD(G^*_{\text{n+1}}, DL)$
5: $smellsHLD \leftarrow filterHLD(G^*_{\text{n+1}}, DL)$
6: $smellsUD \leftarrow filterUD(G^*_{\text{n+1}}, DL)$
7: **return** $rank(smellsCD, smellsHLD, smellsUD)$

(line 1). The classification model to predict likely dependencies is built in step 2. Training a classifier for LP using graph data is challenging because data is usually unbalanced, in the sense that there are fewer instances of the positive class (i.e., dependencies between nodes) than instances of the negative class (all missing dependencies between nodes). Furthermore, the training and testing sets should be chosen judiciously by sampling the graph structures (de Bruin et al., 2020). In our case, the training set comprises instances belonging to two versions: i) existing dependencies in G_{n-1} (as instances of the positive class), ii) missing dependencies in G_{n-1} (as instances of the negative class), and iii) existing dependencies in G_n. This information serves to train the classifier for properly learning instances of both the positive and negative classes. That is, we include information of dependencies in G_{n-1} that are not going to appear in G_n. For example, suppose only one version was to be considered. In such case, no information regarding the negative class could be included for model training, as it would not be possible to guarantee that those dependencies will not appear in G_n.

Once the model is trained, step 3 predicts likely dependencies for ${G^*}_{\mathrm{n+1}}$ (line 2). The prediction might face the case of dependencies in ${G^*}_{\mathrm{n+1}}$ arising between packages added in G_{n+1}, and thus did not initially exist in G_n. In this regard, only potential dependencies for the packages already existing in G_n are considered.

At last, each predicted dependency is associated with a score, which indicates the confidence of the prediction (DL in line 2). Confidence is initially defined as the probability of the dependency to appear in the next version, as determined by the trained model. The higher the confidence score, the higher the chances of the dependency appearing in the next version. Posterior updates of the dependency scores result from the LA mechanism in Phase 3.

4.2. Phase 2: Filtering

The fact that the classifier predicts whether an individual dependency is likely to appear is not enough to predict the appearance of a smell since not every predicted dependency might cause a smell to emerge. Thereby, the graph ${G^*}_{\mathrm{n+1}}$, which includes the predicted dependencies, undergoes a filtering process that depends on the type of smell at hand (lines $4-6$). This *filtering phase* requires the provision of smell-specific filters for the CD, HLD, and UD smell types. For instance, in the case of the cycle filter, based on the newly predicted dependencies should analyze whether they contribute to

closing a cycle. In this case, Figure 3 shows that the addition of the predicted C-E dependency closes a cycle involving packages D, C, and E in G^*_{n+1}.

Cycle Filter. It considers predicted dependencies leading quasi-cycles to close and become actual cycles in G^*_{n+1} (line 4). To do so, all predicted dependencies are simultaneously added to the current graph, and then this "expanded" graph is traversed in search of cycles involving such newly-added dependencies. Considering the large number of cycles that could traverse a given dependency, the filter outputs the dependencies closing new cycles instead of each newly discovered cycle. Dependencies are then prioritized according to their confidence and the number of cycles they complete.

Hub Filter. It considers the nodes incidental to the predicted dependencies that fit the hub characterization for G^*_{n+1} (line 5). This description relies only on the number of dependencies and not on the particularities of such dependencies. The filtering takes all predicted dependencies referring to a node altogether (Díaz-Pace et al., 2018). Hence, it considers the possibility of a quasi-hub node needing multiple dependencies to become a hub. Dependencies might be unnecessarily added to the graph, as not every incidental dependency might be required for the node to become a hub. In this context, we first rank dependencies according to their confidence scores. Then, dependencies are iteratively added, one by one, to G^*_{n+1}, and the hub detection strategy is checked at each step. Iterations stop when the node transforms into a hub, or there are no additional dependencies to be added. At last, the relevance score of a discovered hub is based on the "hubness" of the node (i.e., the ratio between the metrics involved in the hub characterization) multiplied by the average confidence of the dependencies involved.

Unstable Dependency Filter. Analogously to the hub filter, it considers the nodes incidental to the predicted dependencies that fit the characterization of the UD smell for G^*_{n+1} (line 6). However, unlike the hub filter, all predicted dependencies are simultaneously considered in the analysis. This is due to the global nature of the UD smell, in which changes to one node can trigger changes in the instability condition of an unrelated node. The relevance score of unstable nodes is determined by the strength of the smell and the average confidence of its incidental dependencies.

4.3. Ranking Predicted Smells

In (Díaz-Pace et al., 2018), smell filtering was solely based on the simple appearance of the smell, regardless of its characteristics or those of its predicted dependencies. This naive filtering implied that every AS was reported as output, which increased the engineer's efforts for (manual) inspection of the predicted ASs to discard false positives. In this work, filtering is enhanced to consider the computation of relevance scores for the predicted ASs (DL in lines $4-6$), which results in a prioritization (or ranking) of the ASs.

Determining the number of ASs to include in a ranking is not straightforward. Traditionally, the number of recommended elements is defined as a fixed threshold based on the number (or a percentage) of elements (Peker and Kocyigit, 2016). However, in a dynamic environment, as evolving software systems are, such a strategy might affect recommendations' quality. In this context, we adjust the number of ASs to select based on the history of discoverable smells and predictions to account for changes in the number of predictable ASs across system versions. Particularly, the number of smells is set to the average number of predictable ASs in the previous system versions plus its standard deviation. This dependence on the previous predictable smells might affect the quality of recommendations in particular cases, such as sudden changes in the system structure (e.g., a major refactoring), in which none smell could be predicted. Nonetheless, as the system continues to evolve, it is expected that the number of elements to predict will stabilize. Additionally, we include a correction factor based on the nDCG (Normalized Discounted Cumulative Gain) value achieved on previous recommendations. This factor adds a margin error based on the history of missed predictions and allows accounting for unexpected changes in the history of predictions. This factor also accounts for mistakes in the previous recommendations, in which not every smell was included in the defined ranking.

Algorithm 2 Update of dependency confidence scores for the next version

Inputs: $G_{n+1}(V_{n+1}, E_{n+1})$: actual graph for next version $n+1$
$G^*_{n+1}(V^*_{n+1}, E^*_{n+1})$: predicted graph for next version $n+1$
DL_n : list of dependencies (edges) from previous versions, including confidence scores
Outputs: DL_{n+1} : updated list of dependencies, including confidence scores

{DL_n is assumed to be already initialized}
$DL_{n+1} \leftarrow \emptyset$
{Existing predicted dependencies are monitored}
foreach $< d^i_n, s^i_n > \in DL_n$ **do**
  if $d^i_n \in E^*_{n+1}$ **then** {The dependency was predicted}
    if $d^i_n \notin E_{n+1}$ **then** {It did not appear in the actual next version}
      $s^i_{n+1} \leftarrow 1 - \beta * s^i_n$ {Penalize the dependency score}
      {Keep the updated dependency in the list}
      $DL_{n+1} \leftarrow DL_{n+1} \cup < d^i_n, s^i_{n+1} >$
    end if
    {If the dependency appeared, do not copy it to DL_{n+1}}
  end if
end for
{LA created for every new predicted dependency}
foreach $d^j_{n+1} \in E^*_{n+1}$ **do**
  if $d^j_{n+1} \notin DL_n$ **then** {If the predicted dependency is actually new}
    $s^j_{n+1} \leftarrow classifierProb(d^j_{n+1})$ {Initialize probability of appearance for the dependency, as determined by the classifier}
    $DL_{n+1} \leftarrow DL_{n+1} \cup < d^j_{n+1}, s^j_{n+1} >$ {Update list of dependencies}
  end if
end for
return DL_{n+1}

4.4. Phase 3: Feedback

Phases 1 and 2 correspond to a traditional LP approach in which predictions are based on "static" snapshots of the network. Thus, the appearance of each dependency is represented by a one-time event. However, in many applications, the structure and parameters of the network change over time. Thus, richer information could be extracted from the history of network evolution and predictions. For instance, if a dependency was mistakenly predicted for a given version, the knowledge that such prediction was incorrect is not carried to inform future predictions. To account for this situation, after predictions are made and the next system version is known, our approach includes information regarding both added and deleted dependencies on the newest system version. This information could be alternatively provided by the engineer when inspecting the outputs of the previous phases.

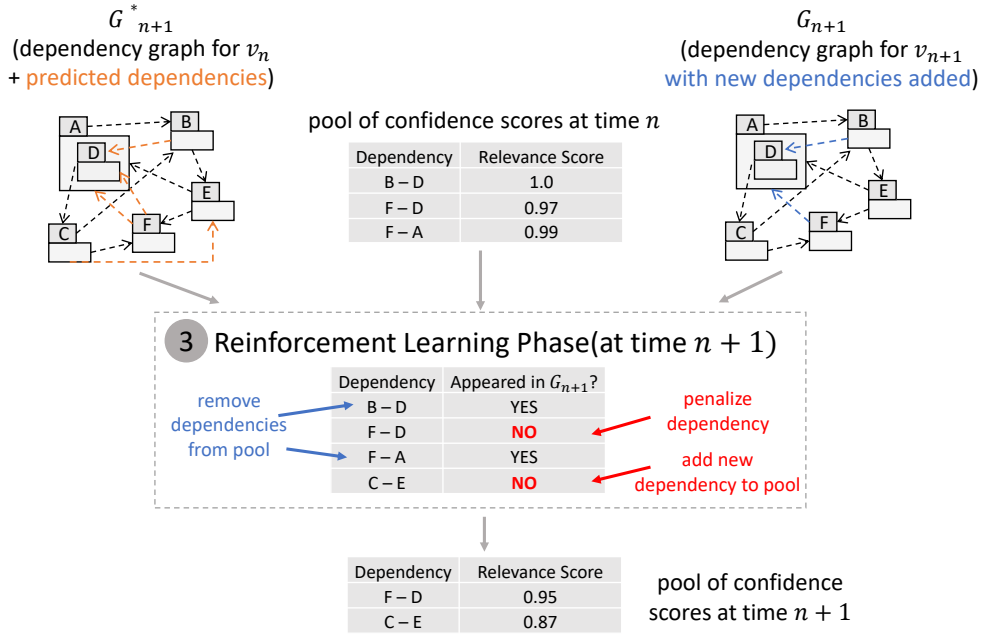


Figure 4: Overview of the Reinforcement Learning phase of the approach

This phase relies on a learning automaton (LA) (Moradabadi and Meybodi, 2017) as a schema for adjusting the probability of occurrence of a phenomenon according to a signal being generated from the environment. In our case, the phenomenon refers to the appearance of a dependency in a graph. An LA is created (and maintained) for every predicted dependency. Each LA starts assigning a confidence score equal to the probability of appearance of the dependency, as determined in Phase 1. The LA is activated as soon as the new version is known to monitor its corresponding dependency. The LA behavior for a dependency $< d^{\mathrm{i}}_{\mathrm{n}}, s^{\mathrm{i}}_{\mathrm{n}} >$ is defined by lines $3-7$ of Algorithm 2. If the dependency was predicted but did not appear in the new system version (lines $4-6$), the LA decreases its confidence score to penalize the incorrect prediction. Otherwise, when the dependency appears in the next version (i.e., the prediction was fulfilled), the LA is discarded as the dependency does not need further monitoring.

In the case of penalization, confidence is updated according to the rule $s^{\mathrm{i}}{}_{\mathrm{n+1}} \leftarrow 1-\beta * s^{\mathrm{i}}{}_{\mathrm{n}}$ (line 5), where $s^{\mathrm{i}}{}_{\mathrm{n}}$ represents the confidence of dependency $d^{\mathrm{i}}{}_{\mathrm{n}}$ at time n, and β is a parameter for measuring the confidence adjustment.

For example, let us assume that dependencies B-D, F-D, and F-A were

predicted for G^*_{n+1}, as schematized in Figure 4. Following Algorithm 2, dependencies should be checked once G_{n+1} is known. As B-D and F-A appeared in G_{n+1}, their LAs are discarded. Since F-D did not appear, its confidence score is penalized from 0.97 to 0.95 for a β of 0.01. In the case of C-E, which was firstly predicted for this version, a new LA is created with an initial confidence of 0.87, as determined by the classifier.

5. Evaluation

To assess the performance of the proposed approach, we selected six Java open-source systems (Apache Camel, Apache Ant, Apache Derby, Apache Cxf, Hibernate and Weka) that have also been used in the literature (Fontana et al., 2016; Le et al., 2018), and whose characteristics are summarized in Table 1[3]. Four of the selected systems belong to the Apache ecosystem, as it is one of the largest open-source organizations that has produced well-maintained code repositories. In turn, two non-Apache system were also included. The six systems have different size and come from different domains and organizations to ensure broad applicability of our results.

[3]The evaluations were performed only on the core libraries of the system distributions. Thus, the reported KLOC refers only such core library.

	# versions	Domain	Release Span Date	KLOC
Apache Ant	10	Software tool for automating software build processes. The primary known usage of Ant is the build of Java applications.	2003 2017	≈ 44
Apache Camel	16	Framework for message-oriented middleware with a rule-based routing and mediation engine that provides a Java object-based implementation of the Enterprise Integration Patterns using an Application Programming Interface to configure routing and mediation rules.	2009 2018	≈ 36
Apache Cxf	8	It is a fully-featured Web services framework.	2009 2014	≈ 39
Apache Derby	13	It is a relational database management system that can be embedded in Java programs and used for online transaction processing.	2005 2018	≈ 39
Hibernate	4	It is an object-relational mapping tool for Java that provides a framework for mapping an object-oriented domain model to a relational database.	2017 2019	≈ 37
Weka	7	It is a collection of machine learning algorithms for data mining tasks. It contains tools for data preparation, classification, regression, clustering, association rules mining, and visualization.	2002 2022	≈ 12

Table 1: Summary of the analyzed systems

We selected systems with more than 10 releases and more than 10 contributors each. The availability of multiple versions through the lifetime of a system is related to the maturity of the projects, as a long story of releases ensures that the evolution of smells can be tracked over an extensive period. It is in the long term that the variations or trends of smells can be noticed. Finally, regarding the number of developers, the goal was to select systems involving experienced developers who know the principles of software design and might be aware of the consequences of breaking them.

We addressed the following research questions:

- ***RQ1.*** *Are the ASs considered in the study correlated with design degradation symptoms?*

To be of interest for predictions, the ASs detected in the systems should be somehow related to quality problems in the system. We analyze the effect of ASs on package structure and the prevalence of ASs in the system evolution (Arvanitou et al., 2017). These two aspects might be indicators of design degradation if the smells are left unattended in the systems.

- ***RQ2.*** *Is the detection of the AS types acceptable in terms of precision and recall?*

Here, we analyze the precision and recall of the approach when combining phases 1 and 2. Note that the LA mechanism is not yet included.

- ***RQ3.*** *Does the prediction of smells improve when considering the LA feedback mechanism?*

We had initial evidence (Díaz-Pace et al., 2018) about low precision values when using only phases 1 and 2 of the approach. Thus, we analyze the precision and recall of the whole approach compared to the results of RQ2. The goal is to assess the possible benefits of the LA mechanism (phase 3).

5.1. Data Collection

As the AS prediction involves multiple system versions, it is necessary to select the sequence of versions to analyze. In this sense, an "*evolution path*" is a sequence of version pairs in which each pair (s, t) represents an ordered pair, where t is the target version that evolved directly from a source version s. Following Behnamghader et al. (2017), we define an evolution path as the sequence of all minor versions between two subsequent major versions of a system to capture the total changes within a single major version. Patches and pre-releases were ignored, as they mainly involve bug fixes, while pre-releases are merged into a posterior official version (either major or minor ones). To illustrate the selection of versions and evolution paths, let us assume the following set of versions for a system: `1.0.0`, `1.1.0`, `1.1.1`, `1.2.0`, `1.2.1`, `1.2.1-beta1`, `1.3.0` and `2.0.0`. As previously defined, the selected evolution path only considers minor versions. Hence, the evolution path for this system will include versions `1.0.0`, `1.1.0`, `1.2.0` and `1.3.0`. In this case, versions `1.1.1` and `1.2.1` are discarded, as they are patches for versions `1.1.0` and `1.2.0`. Version `1.2.1-beta1` represents a pre-release, which is also discarded. Finally, `2.0.0` is discarded as it represents a different major version.

Different systems follow different release evolution paths, so determining the correct evolution path for each system is not straightforward. To determine the correct version sequences and, thus, the evolution paths, we

analyzed the Git log of the selected systems[4].

For most systems, the package structure remains relatively stable. Moreover, in all pairs of consecutive versions (except for the last Derby version), edges between already existing packages were added, which gives good chances of predicting those edges. Regarding the number of ASs detected for each version, as the problem of finding cycles is NP, it might be time-consuming for systems to have a high number of packages and edges connecting them. Hence, we limit the prediction of cycles to those comprising between 3 and 10 packages, or 6 in the case of Hibernate. The reported cycles include every possible cycle in the system, disregarding whether the cycles have edges between sub-packages, which accounts for the high number of discovered cycles.

Cycle predictability was assessed considering that every package in the cycle was already present in the system, and at least one of the involved dependencies does not currently exist. On the other hand, in the case of HLD and UD smells, it was only checked whether the package affected by the smell was already present in the system. Given that this is a global phenomenon, it might occur that although a package already exists, it might need dependencies with other unknown packages to become an HLD or UD instance. As another observation, we should mention that not every pair of versions in the selected evolution path is helpful for predictions, given that there might not be smells to predict for every pair of them.

When transforming the package dependency graphs into datasets for the ML task, we computed all features described in Section 3.2. To obtain meaningful predictions, the pairs of consecutive versions to analyze (v_{n-1}, v_n) were required to present changes regarding the existence of ASs, which implies the addition of new dependencies and smells between existing packages. The different types of ASs were identified according to the detection rules presented in Section 2.

An SVM algorithm (Support Vector Machines) parameterized with an RBF (Radial Basis Function) kernel was selected for the classification phase, which is helpful for problems involving datasets with an unbalanced class distribution (Schölkopf et al., 2001). Basically, SVM aims to create a hy-

[4]The characteristics of the versions analyzed in this study for the six selected systems are listed in the Appendix at: `https://github.com/tommantonela/ASPredictor/wiki/Appendix:-Selected-System-Versions`

perplane that separates the data into classes by means of a kernel function that previously transforms the data. Traditional linear functions work well when the original data to classify is effectively linearly separable, which might not be always the case. Additionally, linear functions do not allow to work with outliers or highly skewed classes (Schölkopf et al., 2001). On the other hand, non-linear kernels, such as RBF, allow mapping the data into a high-dimensional feature space that increases the chances of finding an adequate separation in cases with highly skewed classes or outliers, when compared to linear or even polynomial kernels.

As explained in Section 4.1, system versions were temporally split into training and test sets (Yang et al., 2015). For the training set, it comprised existing dependencies in versions G_{n-1} and G_n (positive class), and missing dependencies in G_{n-1} that did not appear in G_n (negative class). The test set comprises the dependencies in G_{n+1} between packages already existing in G_n, as it would not be possible for the model to directly predict dependencies between new and unknown packages.

5.2. Performance Metrics

We compare the predicted results (for a given version) against the next system version. Predictions are made over v_{t-1} and v_t, and their predictive performance is then evaluated by considering the predictable elements (e.g., dependencies or ASs) in v_{t+1}.

Performance was assessed using traditional ML metrics, namely: precision and recall. In principle, both recall and precision are relevant metrics; however, we are more interested in recall than in precision because a high recall would indicate that most new real smells are predicted, regardless of the number of actually predicted smells. On the other hand, a high precision would suggest that most of the predicted smells appear in the system, regardless of the actual number of predictable smells. Eventually, a high recall might involve incorrectly predicting some smells, negatively affecting precision. If those smells are a small fraction, they could be discarded with a manual analysis of the tool outputs by engineers. Furthermore, the ranking of smells obtained from the LA feedback mechanism was evaluated using the nDCG metric (normalized Discounted Cumulative Gain).

In the following sub-sections, we separately discuss the predictions for the three phases of the approach as a series of 7 experiments. Experiments are organized according to the research question they are expected to answer. All evaluations were run on a PC i7-4500U 1.8 GHz. with 8GB RAM - Windows

10 and Java 8. In general, the computation time is affected by factors such as the number of features and the size of the dependency graphs.

5.3. Answers to RQ1: Importance of the Smells

Experiment #1: Smelly Packages and Smell Survival Rates

In a technical debt framework (Ramasubbu and Kemerer, 2014), we can think of ASs as symptoms of architecture degradation, particularly if the ASs persist over several system versions. One indicator of debt accumulation is change proneness (CP) (Arvanitou et al., 2017), which measures the susceptibility of an artifact (e.g., a class or a package) to changes in upcoming system versions. For a given package, the CP computation relies on two parameters: the frequency of changes and incoming/outgoing dependencies for the package. We analyze the relationships between the CP of the packages affected by smells (called "smelly" packages) and the CP of the non-smelly packages. As smelly packages indicate potential quality problems, we hypothesize that they should have higher CP than non-smelly packages, which will eventually compromise the maintainability of the system.

For this experiment, we computed the CP metric for all the packages and versions of each system, and marked those packages affected by at least one smell instance. Then, we performed a statistical test, as sketched in (Le et al., 2018), comparing the smelly and non-smelly CP distributions. Since these distributions were not statistically normal, we ran a Mann-Whitney test for each system with confidence levels of 0.05 and 0.01, respectively. Cliff's Delta was used to quantify the effect size between the tested distributions. Results showed that, in most cases, the CP distribution for smelly packages was significantly higher than that of the non-smelly packages, with medium to large effect sizes. Thus, we can infer that the ASs considered in our study were representative of design problems.

In addition, a recent study has provided evidence about the evolution of instability smells over time (Sas et al., 2019), showing that package-level smells often stay longer and tend to move to central parts of the system. To assess these presumptions, we studied the survival rates of the ASs using the Kaplan-Meier estimator, as depicted in Figure 5. Except for the cases of Cxf (CD) and Ant (HLD and UD), we observed that the trends for the three smell types are consistent and show a prevalence of the ASs over time.

Overall, we can answer **RQ1** by saying that the analyzed smells play a potential role in the design problems of the systems in terms of contributions

to CP and survivability across versions. These factors make the ASs relevant for engineers and a good target for our predictive approach.

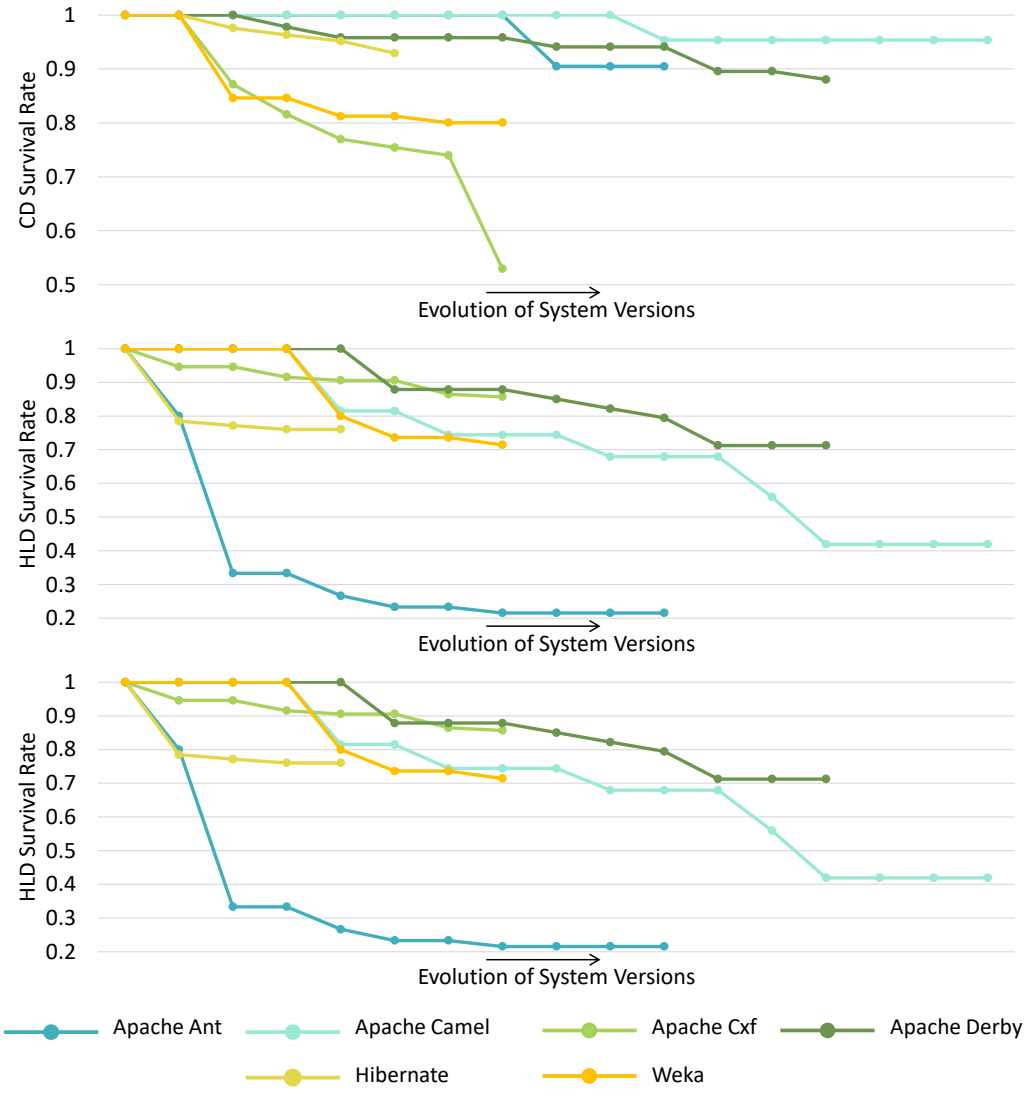
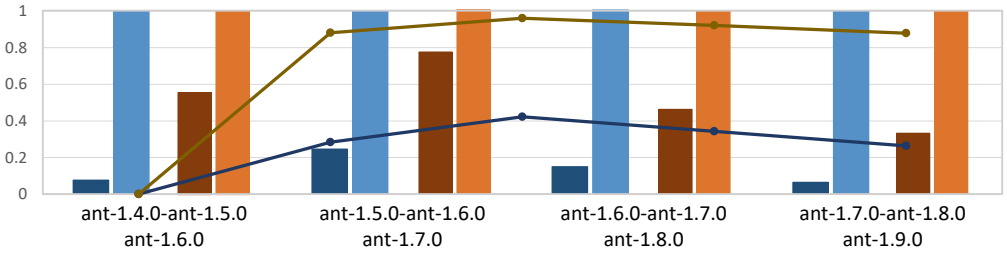


Figure 5: Survival Rates for the different Smells by Type

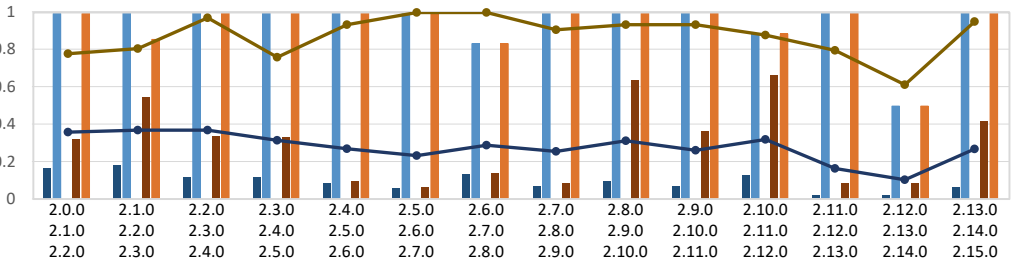
5.4. Answers to RQ2: First and Second Phases

Experiment #2: CD Prediction

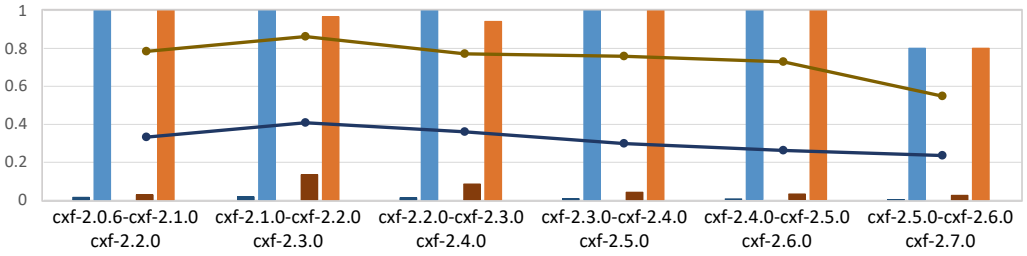
Figure 6 presents the results of the CD prediction, with and without the feedback mechanism for the Apache and non-Apache systems. As it can be observed, excepting for Weka, almost every new dependency closing a new cycle was found. Nonetheless, precision was generally low.



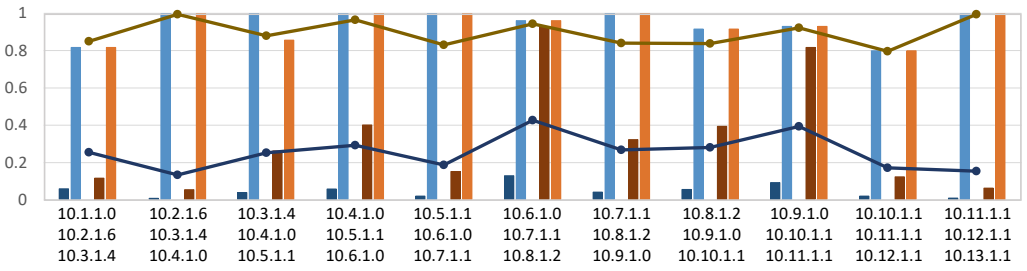
(a) Apache Ant



(b) Apache Camel



(c) Apache Cxf



(d) Apache Derby

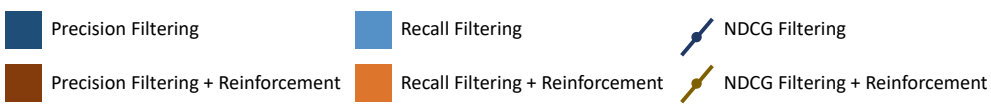
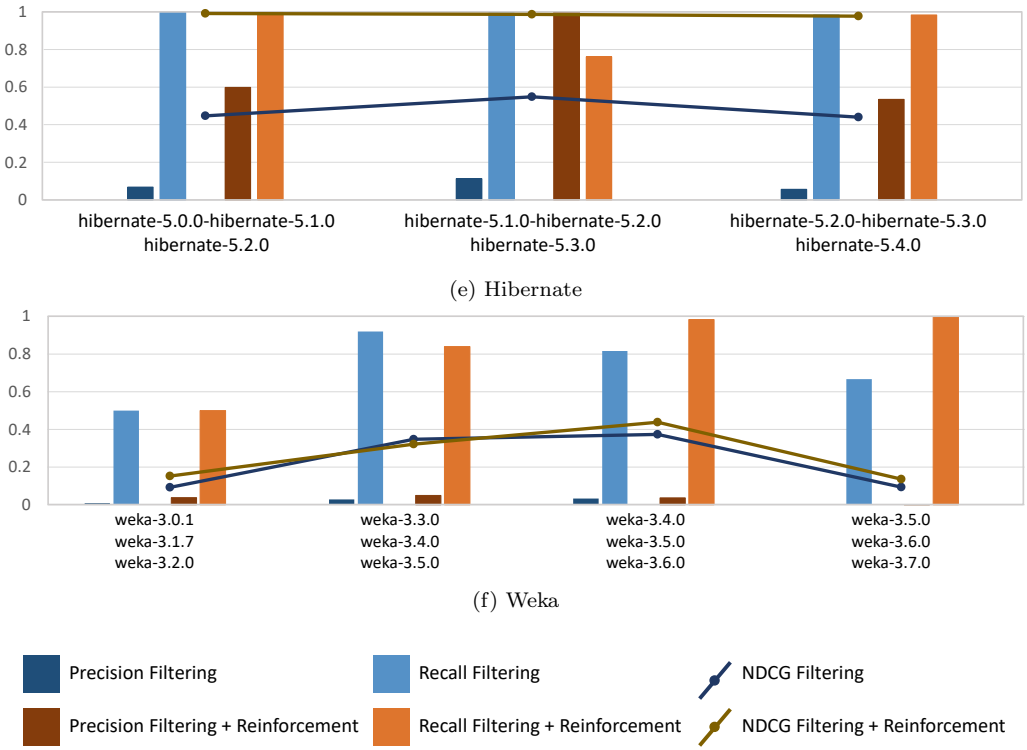


Figure 6: Cycle Prediction Results - Apache Systems



(e) Hibernate

(f) Weka

Figure 6: (cont.) Cycle Prediction Results - Non Apache Systems

As regards Ant, except for `ant-1.3-ant-1.4-ant-1.5`, the filter was able to discover the new cycles. This problem was caused by missed dependencies in the first phase. Even though at least one of the predicted dependencies was part of a cycle in version `ant-1.5`, cycles were not discovered, as closing them required additional dependencies that were not predicted. On the other hand, a perfect recall was achieved for the other pair of analyzed versions, meaning that all predictable dependencies involved in the closure of a new cycle were discovered.

Camel presented similar recall results as Ant, achieving high recall for all but one version pair. Similarly to Ant, the decrease in recall was due to mistakes in the dependency prediction phase. Cxf and Weka exhibited the lowest precision results derived from the low precision of the first phase.

In the case of Derby, precision was low while recall was perfect for three analyzed version pairs. In the remaining pairs, the missed predictions were due to mistakes in the first phase. In the case of Hibernate, recall was perfect in all cases, at the expense of making erroneous recommendations, as

evidenced by the low precision.

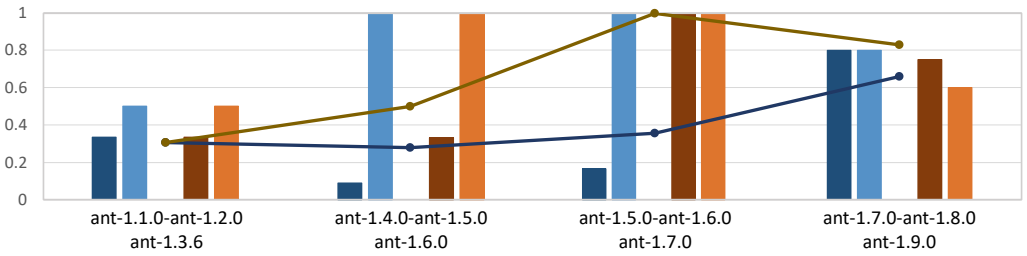
At last, Weka achieved the lowest results for the three metrics. Particularly, recall was not perfect in any case, showing that despite several false positive results (denoted by the low precision), not every newly closed cycle was found.

In summary, low precision was caused by the incorrect prediction of dependencies during the first phase that led to the closure of cycles. Nonetheless, low precision also accounts for missed predictions. For example, a predicted dependency that appears in the next version but needs another dependency for effectively closing a cycle would be counted as a mistake if the second dependency is not also predicted. In turn, this situation negatively affects recall.

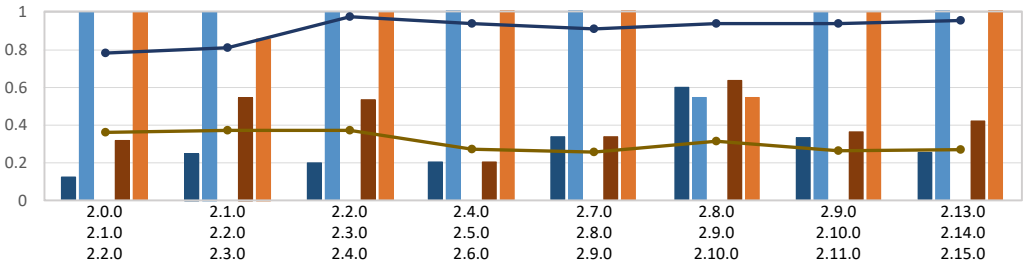
Experiment #3: HLD Prediction

Figure 7 presents the results for the HLD prediction. Ant, Cxf, and Hibernate achieved the lowest precision results. Both Ant and Hibernate achieved perfect recall, while Cxf achieved a maximum of 0.8. In the phase 1, Cxf achieved almost perfect recall, meaning that nearly every new dependency was correctly predicted. Hence, the low recall in phase 2 could account for the global nature of the HLD smell. On the other hand, Weka achieved perfect precision for two version pairs, at the expense of lower recall.

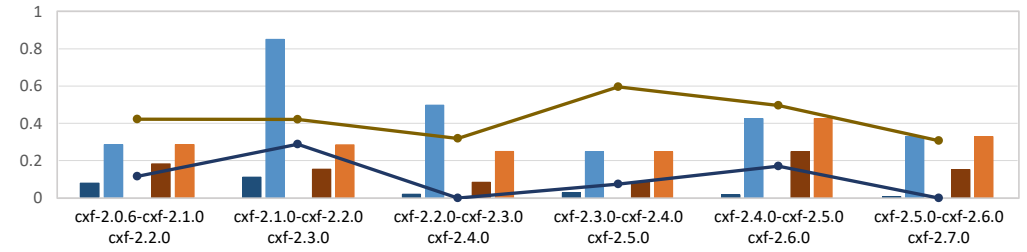
Camel achieved perfect recall for most analyzed sets of versions, implying that, unlike for Cxf and Hibernate, the global structure had a mild effect on predictions. This might also be related to the differences between the number of packages and edges added across versions. For example, considering Cxf versions `cxf-2.4.0-cxf-2.5.0-cxf-2.6.0` that achieved almost perfect recall in the dependency prediction but low recall in the HLD prediction, packages varied by 11%. For Camel, packages varied at most 19% when considering versions `camel-2.7.0-camel-2.8.0-camel-2.9.0`, which maintained similar recall levels. Similar observations can be made regarding the edges, which varied at most 8% and 13% for the same pairs of versions of Cxf and Camel, respectively. Finally, regarding precision, the highest score was observed for `camel-2.8.0-camel-2.9.0-camel-2.10.0`, which coincidentally also achieved the minimum recall.



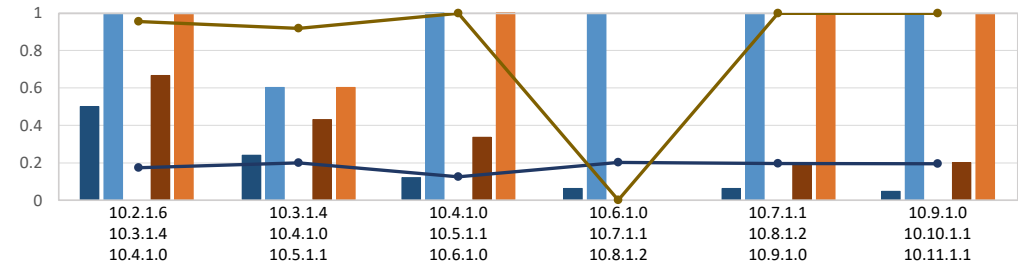
(a) Apache Ant



(b) Apache Camel



(c) Apache Cxf



(d) Apache Derby

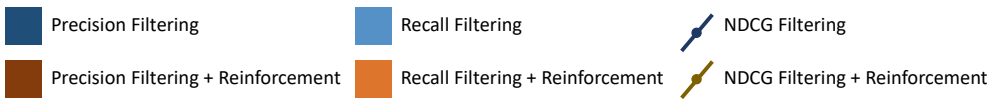
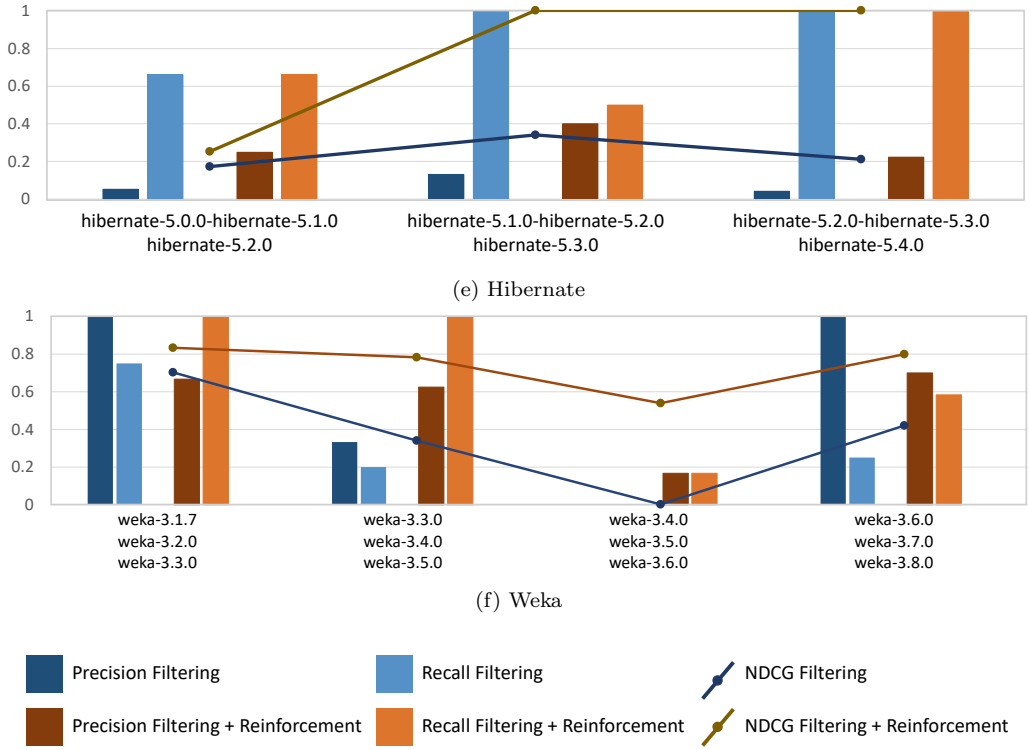


Figure 7: Hub Prediction Results - Apache Systems



(e) Hibernate

(f) Weka

Figure 7: (cont.) Hub Prediction Results - Non Apache Systems

Derby achieved perfect recall for some pairs of versions at the expense of low precision. Even though for `derby-10.1.1.0 - derby-10.2.1.6 - derby-10.3.1.4` there was one predictable hub, it could not be predicted due to either the global phenomenon of the hub or mistakes in dependency prediction.

For two version pairs, Weka achieved perfect precision but not perfect recall. This implies that all predictions were correct, but some correct predictions were missing. For one version pair (`weka-3.4.0 - weka-3.5.0 - weka-3.6.0`) both precision and recall were 0. This was caused by both missed dependencies on the first phase and the global nature of the smell.

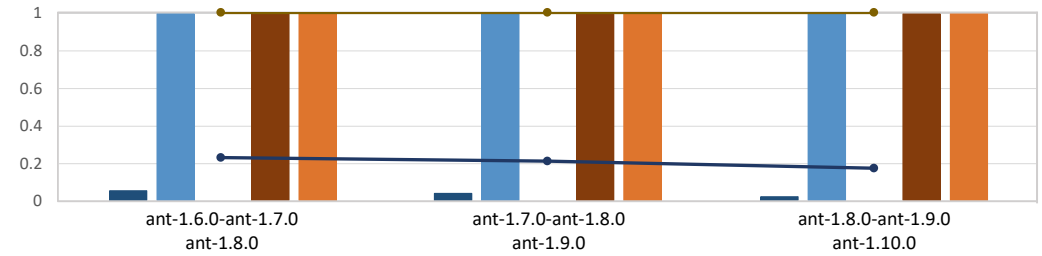
From the observed results, it can be stated that the appearance of a hub does not only depend on the already known structure of the graph (or specifically, only on the dependencies being added to a package), but also on the addition of new unknown packages to the system and their corresponding dependencies. Hence, the decrease in recall might not be only related to mistakes during the first phase.

Experiment #4: UD Prediction

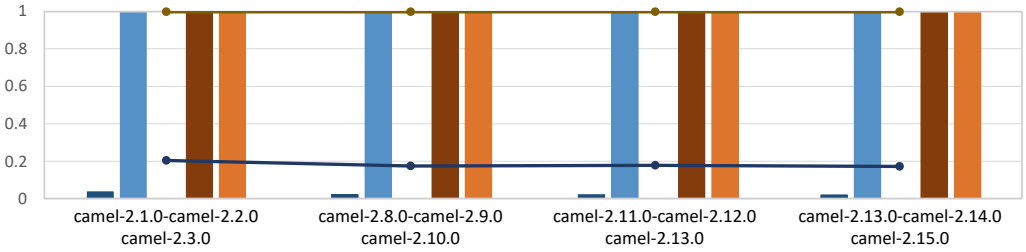
Figure 8 reports the prediction results for UD smells. As for the HLD prediction, the most variable results were observed for Cxf. For Derby, Hibernate and Weka not every new UD could be predicted. This situation highlights the effect of the low precision in the first phase since false positive dependencies affected the accurate prediction of UD smells. The cases in which UD predictions were not made differ from those for which no HLD predictions were made, emphasizing the differences between HLD and UD smells even though they respond to similar definitions and system configurations.

Prediction results were similar for Ant, Camel and Cxf. In all reported cases, the filtering correctly predicted the appearance of all new smells (perfect recall) at the expense of a high number of mistakenly predicted smells (low precision). Incorrect predictions ranged between 18 and 50 for Ant (representing between 60% and 83% of the total number of packages) and 28 and 55 for Camel (representing between 54% and 70% of the total number of packages). The low precision of predictions could be explain by the low number of predictable UDs. For example, in the case of Ant and Camel, only one UD was predictable in each version pair. Hibernate results showed a low precision, with variations in recall values, which was perfect for only one set of analyzed versions. Weka showed perfect recall for two version pairs and the highest precision among the three analyzed smells.

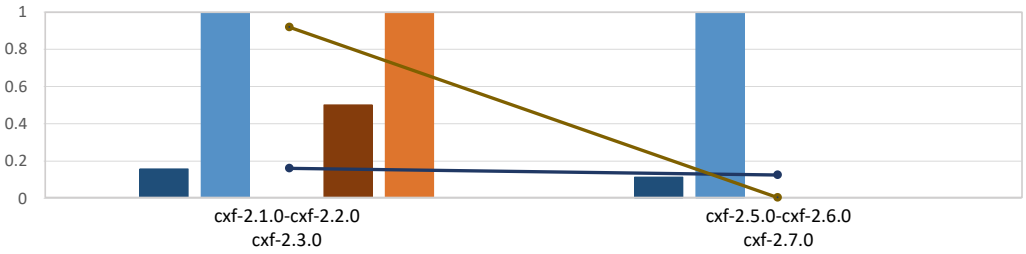
From the observed results, similarly to the trends for HLD smells, UDs appear not only due to the addition of new edges to specific nodes but also due to changes in the overall graph structure. Nonetheless, the variability of results showed differences in the characterizations of both types of smells.



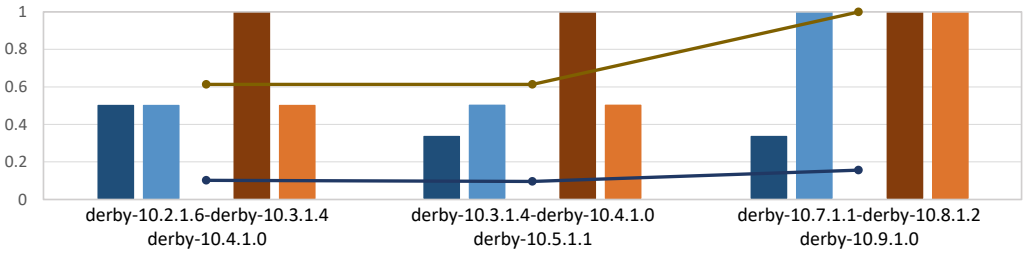
(a) Apache Ant



(b) Apache Camel



(c) Apache Cxf



(d) Apache Derby

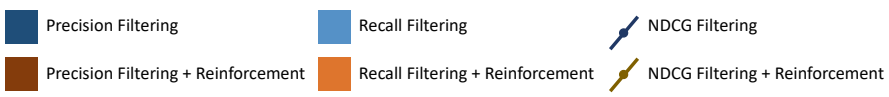


Figure 8: Unstable Dependency Prediction Results - Apache Systems

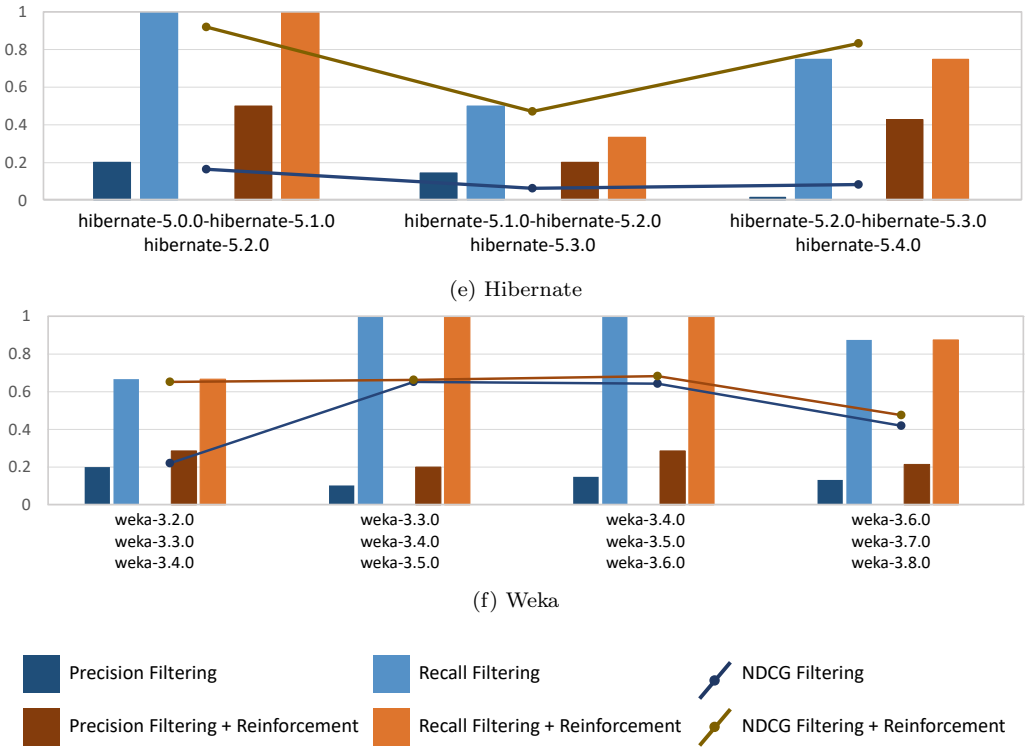


Figure 8: (cont.) Unstable Dependency Prediction Results - Non Apache systems

From Experiments #2, #3, and #4, we can answer **RQ2** by saying that the prediction of ASs exhibited an average precision of 0.15, 0.57, and 0.1 for the CD, HLD and UD types. The average recall was 0.75, 0.69, and 0.98 for the CD, HLD and UD types, respectively. For most pairs of versions, the CD filter predicted every new dependency leading to new cycles. As for HLD smells, hubs did not only appear due to new edges being added to specific nodes but also due to changes in the overall graph structure. The different systems had different sensitivity levels concerning those changes. The appearance of UD smells showed similar trends to those of HLD smells. Overall, the results motivate the need to improve the AS predictions' precision.

5.5. Answers to RQ3: Third Phase

The addition of the LA feedback mechanism incorporates the possibility of ranking the predicted ASs. As previously explained, we also adjusted the

number of potential smells to be presented to the engineer. When only using filtering, the nDCG represents the worst-case scenario in which the engineer is presented with the complete set of predicted ASs and has to analyze all of them to find the relevant ones.

Experiment #5: CD Prediction with Feedback

The overall trends showed in Figure 6 indicate differences in precision and nDCG values compared to the results for RQ2. That is, the LA mechanism reduces the number of mistakenly predicted CD smells and allows to rank the smells higher than the mistaken predictions.

For Ant, we found precision improvements (with the LA mechanism) leading to absolute differences of 0.5. The improvements also implied reducing the number of mistakenly predicted dependencies closing cycles from 60 to 8, representing less than 1% of all potential dependencies in the system. The fact that precision improved while recall was not affected shows that the ranking strategy could rank the actual smells higher than the predefined cut line. Nonetheless, further investigations are needed to determine the correct length of the ranking and better rank the predictions. As nDCG was not perfect, we infer that some false positives were still ranked higher than the correct predictions.

As for Camel, the LA mechanism achieved a lower recall for `camel-2.12.0-camel-2.13.0-camel-2.14.0`, which can be explained by mistakenly ranking a correct prediction below the cut line. Since precision was also low, mistaken predictions were also included in the ranking. The nDCG was close to 1 for several versions, showing that the ranking accurately had almost every correct dependency in the first positions, despite false positives. For this system, precision improvements showed absolute improvements of up to 0.54 and reduced the number of incorrectly predicted dependencies in half, representing approximately 1% of the potential dependencies in the system.

In Cxf, precision was improved in all cases, representing a reduction of 90% in the number of mistaken predictions. In the worst case, incorrectly predicted dependencies were only reduced in half, predicting at most 394 incorrect ones (representing approximately 1% of the potential dependencies). The LA mechanism led to two sets of versions `cxf-2.1.0-cxf-2.2.0-cxf-2.3.0` and `cxf-2.2.0-cxf-2.3.0-cxf-2.4.0`)) to a smaller recall than the filtering alternative, with differences up to 5%. In both cases, these differences corresponded to a correct dependency ranked below the cut line, thus highlighting the importance of determining an accurate ranking strategy.

For Derby, the precision and recall improvements due to the LA mechanism reached average absolute improvements of 0.28, with a maximum of 0.79. In light of these improvements, the number of incorrectly predicted dependencies decreased approximately one or two orders of magnitude, predicting at most 30 mistaken dependencies. In the best case, only two incorrect predictions were made.

For Hibernate, precision exhibited absolute improvements ranging between 0.47 and 0.86. Similar to Derby, the number of wrong predictions decreased approximately one or two orders of magnitude, with a maximum of 183 incorrectly predicted dependencies. Note that for filtering, at most 1400 incorrect dependencies were predicted. In the case of `hibernate-5.1.0-hibernate-5.2.0-hibernate-5.3.0`, precision was perfect, implying that all predictions were correct. Nonetheless, such perfect precision was not accompanied by a perfect recall, suggesting that some correct predictions were ranked below the cut line.

At last, for Weka, the LA mechanism achieved a lower recall for `weka-3.3.0-weka-3.4.0-weka-3.5.0`, which can be explained by ranking correct predictions below the cut line. In all cases, even though precision scores were low, they were improved in average by 115%. nDCG results were similar for all version pairs. This implies that both the filtering and the LA mechanism produce similar ranking of the correct predictions, while an adequate computation of the number of predictions to make helps to reduce the number of false smells.

Experiment #6: HLD Prediction with Feedback

As shown in Figure 7, the differences in precision, recall, and nDCG between the filtering and the LA mechanism depended on the system under analysis. As for CD prediction, the overall trend suggests that the LA mechanism allows discarding false smells, as shown by the precision improvements. Moreover, the nDCG values indicate that, in general, the LA mechanism helps to improve the position of the relevant smells in the ranking.

In the case of Ant, the LA mechanism outperformed filtering in terms of precision for all but one sets of versions. For such versions, recall also decreased due to the ranking of correct predictions below the cut line. However, the ranking did not seem to affect the nDCG values significantly. This could be related to the actual number of hubs to predict. Unlike the CD smells, in which there were dozens or hundreds of dependencies to predict, in this case, the number of predictable smells is lower than 10, representing

less than 8% of the total elements in the system.

For Camel, the LA mechanism achieved a lower recall for one set of versions than the filtering. This was caused by the fact that a missing hub was found (as the higher recall for the filtering results showed), but it was ranked below the cut line. The LA mechanism obtained the same precision and recall results as the filtering for two sets of versions. This situation was related to overestimating the number of smells to predict. Hence, the one smell that could be predicted was indeed predicted, but two additional incorrect predictions were made. In this case, precision showed average absolute improvements of 0.13, with a maximum difference of 0.33. The number of false positives was reduced in one order of magnitude, with at most 4 incorrect predictions, representing 5% of the total elements of the system.

In Cxf, for two sets of versions, even though the LA mechanism improved the filtering precision, it was at the expense of reducing the recall due to bad decisions in the ranking. Compared with the other systems, Cxf achieved the lowest nDCG results. These performance differences across systems show the sensitivity of the LA mechanism and ranking strategy to the underlying characteristics of the systems. Precision improvements had an average absolute value of 0.1, with a maximum improvement of 0.23. The number of incorrect predictions was reduced on average by 63%, making at most 12 incorrect predictions, representing 3% of all elements in the system. Note that this was the highest absolute number of incorrect predictions.

In Derby, for `derby-10.6.1.0-derby-10.7.1.1-derby-10.8.1.2`, the LA mechanism failed to discover the predictable hubs due to mistakes in the ranking positions. In all the other cases, recall was maintained, with average absolute precision improvements of 0.14 and a maximum of 0.21. Due to these improvements, the number of incorrectly predicted hubs decreased 85%, making at most 6 incorrect predictions. The nDCG results showed that the correct recommendations were consistently ranked above all the incorrect ones.

Similarly as for Cxf, for one set of Hibernate versions, even though precision was improved, recall decreased due to difficulties for producing an accurate ranking. For `hibernate-5.0.0-hibernate-5.1.0-hibernate-5.2.0`, precision improvements are proportionally higher than nDCG improvements showing that the ranking placed incorrect predictions in the top positions while correct predictions were ranked lower. Similar to the other systems, the highest improvements were observed for precision, with an average absolute improvement of 0.21.

In the case of Weka, precision decreased for the two version pairs that achieved perfect precision for the filtering mechanism, while recall greatly increased. These results showed that introducing the LA mechanism help improving the recall of the model by finding some or all of the missing dependencies, and thus the missing smells. This can also be observed for versions `weka-3.4.0-weka-3.5.0-weka-3.6.0`, in which the LA mechanism found one of the predictable smells. Unlike the other systems, the highest average improvements were observed for recall, with an average absolute improvement of 0.39, followed by nDCG with an average absolute improvement of 0.30.

With the exception of Weka, the combination of perfect precision with lower recall was not observed in any case. In this sense, it can be stated that, in general, the ranking length determined for the experiments did not underestimate the number of predictable smells. On the contrary, in some cases, such number was over-estimated with negative consequences in precision. The drop in recall, and especially the low results for Cxf and the minor nDCG differences for Weka, highlight the importance of adequately defining the ranking strategy.

Experiment #7: UD Prediction with Feedback

The effects of the LA feedback mechanism for predicting UD smells are shown in Figure 8. In the case of Ant and Camel, the LA mechanism reduced the number of false positives of the filtering. For both systems, perfect precision and recall were achieved. These results imply that correct predictions were ranked above the incorrect ones and that the number of smells to predict was accurately estimated.

For `cxf-2.5.0-cxf-2.6.0-cxf-2.7.0` in Cxf, the LA mechanism could not find the new UD smells, while the filtering found them. This was caused by the order of the elements in the ranking. In turn, for `cxf-2.1.0-cxf-2.2.0-cxf-2.3.0` the LA mechanism improved both precision and nDCG, while maintaining the recall.

Regarding Derby, the LA mechanism achieved perfect precision while maintaining the same recall level of the filtering. This implies that the ranking strategy placed the correct predictions above the incorrect ones. Nonetheless, as in some cases, recall was not perfect, neither for the filtering nor the LA mechanism, some smells could not be predicted due to mistakes in the first phase.

The results for Hibernate exhibited two different behaviors. First, for

`hibernate-5.0.0-hibernate-5.1.0-hibernate-5.2.0` and `hibernate5.2-.0-hibernate-5.3.0-hibernate-5.4.0`, the LA mechanism improved both the precision and nDCG values of the filtering variant. Proportionally, the increment in nDCG was higher than that of precision, showing that the correct predictions were ranked in higher positions despite false positives. Then, for `hibernate-5.1.0-hibernate5.2.0-hibernate-5.3.0`, precision was improved while recall decreased due to the ranking order.

For Weka, in all cases recall was maintained, while precision was improved in average 0.10, which represents an improvement of 72%. Similarly as for the CD smell, in general, nDCG was only slightly improved. These results showed that the LA mechanism mostly helped in reducing the number of incorrectly predicted smells.

Finally, from Experiments #5, #6, and #7, we can answer **RQ3** by saying that precision was consistently improved for every smell type due to reductions of the false positive rate when using the LA feedback mechanism. On one side, the differences between both variants were mainly in terms of precision and nDCG. Precision values went from an average of 0.2 to $0.3-0.7$ with the LA mechanism.

The significance of the observed differences was analyzed with the Wilcoxon test for related samples. For this test, the null and alternative hypotheses were defined. The null hypothesis stated that no difference existed between the results observed for filtering or the LA feedback mechanism, i.e., ranking and reductions in the number of smells presented to the engineer did not significantly affect prediction performance. On the contrary, the alternative hypothesis stated that the differences among the results observed for the different alternatives were significant and non-incidental. Results showed with a confidence of 0.01 that applying the LA feedback mechanism allowed to significantly improve the filtering performance for the three types of smells. As expected, the highest effect was observed for both precision and nDCG (with medium to large effects), while recall differences were negligible in most cases.

5.6. Threats to Validity

The performed study involves some threats to validity. Regarding internal validity, a first threat refers to extracting the static dependency graph and the smell identification. The rules for identifying ASs were based on guidelines from the literature. However, using different parameters for the ASs (e.g.,

the median value for HLD, the size limit for CD, or instability values for UD) could have led to other dependency graphs and a different number of AS instances in the versions. Furthermore, as several pre-releases or patches might have been disregarded between the selected minor versions, we might have missed some information in the pairs of analyzed versions.

Second, although we acknowledge that not all version ranges are suitable for having good predictions, the relations between them and the features included in the classification model might need further analysis. Features could have also favored predictions for some smell types. Third, the detection capabilities of the filters have some limitations. Filters rely on the newly predicted dependencies based on the existing packages of the system versions. This implies that there might be smells that the filters cannot discover, as they might also depend on changes caused by dependencies that cannot be predicted as they involve non-existing packages. Hence, we believe that alternative ASs filters should be investigated.

Fourth, we considered each system package to be a different module. However, this assumption might not hold in all systems due to code organization aspects. The criterion for identifying modules (from the code) or the granularity at which they are considered changes the graph structure, and consequently, it might affect the LP task. Constructing dependency graphs from systems written in different programming languages (e.g., C, C++ or Python) might also impact on the features of the datasets.

Regarding external validity, we performed an evaluation on a moderate sample of open-source systems. Nonetheless, we can identify a first threat related to user validation of both the detected ASs and the rankings of predicted ASs. Not all smell configurations in the dependency graphs might be "real" smells in the engineers' eyes. There are also cases where some smells are not harmful or might correspond to good design decisions. Second, we did not analyze in detail the issues, changes, and possible refactorings associated with the selected system versions. More experiments need to be conducted to generalize the trends observed in the experiments, particularly for commercial systems. Furthermore, these experiments should include systems using programming languages other than Java.

At last, the three types of ASs currently supported by the approach could be extended to other types of dependency-based smells or even to concern-based smells (Garcia et al., 2009).

6. Related Work

Several characterizations and catalogs of AS have been reported in the literature (Mo et al., 2015; Garcia et al., 2009). Most of these works categorize certain smells as related to undesired dependencies between software components.

Dependency graphs are a typical representation of software elements and their relations, often built using static analysis of source code or other artifacts, which can be assimilated to social networks. Over the last years, SNA has developed its own set of techniques, algorithms, and metrics for modeling nature and human phenomena. Software Engineering researchers have started leveraging the power of SNA to understand different aspects of the software development process. For example, research has focused on studying how developers collaborate (Manteli et al., 2012), the interplay of dependencies among technical artifacts and their creators, or how decisions about organizational structure or the development process affect communities of developers.

Social Network Analysis techniques have been applied to study dependency graphs to predict software evolution (Terra et al., 2013), the appearance of defects and bugs (Zimmermann and Nagappan, 2008), and the existence of vulnerable components (Nguyen and Tran, 2010). Regarding software evolution and low-level design, Terra et al. (2013) used LP techniques to infer the package in which a particular class should be located during a refactoring process. At last, Zimmermann and Nagappan (2008) concluded that metrics assessing the local neighborhood of nodes contribute better than global metrics to understanding software systems. Additionally, they stated that network metrics are better descriptors of software defects than source code metrics. Finally, Nguyen and Tran (2010) derived metrics from the component dependency graph to predict vulnerable components. In all cases, the authors agreed that the topological analysis of dependency graphs could reveal (or even predict) interesting properties of software systems. However, these works are mostly descriptive, and there is no intent to predict future system states.

Several automated techniques for detecting smells have been proposed, which differ in the algorithm, the applied heuristics or rules, and the metrics considered. Smell detection tools have adopted ML approaches for predicting the existence of a given type of smell based on features describing the source code. The most notable advances were proposed in (Arcelli Fontana

et al., 2016), in which an evaluation of 32 ML algorithms for detecting four types of code smells on 74 systems was reported. Classifiers were trained based on object-oriented metrics at method, class, package, and project levels. For selecting the smells to train the classifiers, the authors used freely available tools, and the severity of each smell was manually defined. Experiments showed a high classification performance, guiding the authors to conclude that ML is a feasible approach for smell detection. However, the approach presented some drawbacks that might affect the generalization of results (Nucci et al., 2018). The classifiers seemed to neglect the structural characteristics of smelly and non-smelly components and were trained to detect smells in unrealistic conditions.

Our approach differs from the previously described works in two aspects. First, we are concerned with dependency-based ASs rather than with code smells. Second, our approach predicts relations among components and the appearance of new smells instead of inferring component characteristics (such as defects, bugs, or vulnerabilities). Moreover, whereas other techniques focused on identifying smells that already exist in the system, our approach tries to anticipate smells that do not exist yet but have high chances of affecting the system in future versions.

7. Conclusions

In this work, we develop an approach based on LP techniques and a classification model for predicting AS instances that are likely to appear in a system. This predictive capability is the main contribution of the work. The approach is designed as a pipeline with three phases, in which a dependency classification model works in tandem with a set of smell filters (per type of smell) and an LA mechanism.

An exploratory evaluation with three types of smells in six open-source systems showed average precision and recall values of 0.3 and 0.9 regarding the identification of the real smells. The performance effect of the first phase is due to the inclusion of content-based features in the classification model. As for the second phase, we found evidence that smell filters alone can lead to high recall but low precision values regarding the predicted smells. Although good recall is usually preferred over good precision, reducing the number of false positives is desirable. An LA mechanism was incorporated into the approach to improve the precision of smell prediction. The evaluation of this

LA mechanism showed improvements in the precision results of up to three times, with only minor (or none) drops in recall.

In future work, we plan to perform an evaluation with more systems (both commercial and open-source) and include user studies to corroborate the findings of this work. The current approach presents some limitations that should be addressed. First, predictions only work for existing packages between consecutive versions. However, we acknowledge that packages could be renamed or even re-grouped from one version to another. In this sense, we intend to reuse the content-based similarity criteria so that a predicted dependency for any given pair of packages could be extended to other packages being similar to those of the pair. Second, the approach supports predictions of a binary nature for a pair of packages, i.e., a dependency will or will not appear in the next version. However, there are situations in which two packages have a dependency, but they become more coupled over time. We believe the LP task can be adapted to this case by predicting whether a given dependency will strengthen in the next version. Third, we will investigate whether classification models for recognizing different smells can be devised to substitute for the detection rules and the filters. Furthermore, a classification model for ASs could also distinguish which smells are relevant to engineers. Finally, another line of future work is integrating the approach with existing tools or research prototypes, such as SonarQube or Arcan (Fontana et al., 2017).

Acknowledgements

This work was partially funded by project PICT-2016-2973 (ANPCyT-Argentina).

References

F. Arcelli Fontana, M. V. Mäntylä, M. Zanoni, and A. Marino. Comparing and experimenting machine learning techniques for code smell detection. *Empir Soft Eng*, 21(3):1143–1191, 2016. ISSN 1573-7616.

E. M. Arvanitou, A. Ampatzoglou, K. Tzouvalidis, A. Chatzigeorgiou, P. Avgeriou, and I. Deligiannis. Assessing change proneness at the architecture level: An empirical validation. In *2017 24th Asia-Pacific Software Engineering Conference Workshops (APSECW)*, pages 98–105, Dec 2017. doi: 10.1109/APSECW.2017.21.

P. Behnamghader, D. M. Le, J. Garcia, D. Link, A. Shahbazian, and N. Medvidovic. A large-scale study of architectural evolution in open-source software systems. *Empir Soft Eng*, 22(3):1146–1193, 2017. ISSN 1573-7616.

G. de Bruin, C. Veenman, H. van den Herik, and F. Takes. Experimental evaluation of train and test split strategies in link prediction. In R. Benito, C. Cherifi, H. Cherifi, E. Moro, L. Rocha, and M. Sales-Pardo, editors, *Complex Networks & Their Applications IX - Volume 2, Proceedings of the Ninth International Conference on Complex Networks and Their Applications, COMPLEX NETWORKS 2020, 1-3 December 2020, Madrid, Spain*, volume 944 of *Studies in Computational Intelligence*, pages 79–91. Springer, 2020. doi: 10.1007/978-3-030-65351-4_7.

Lakshitha de Silva and Dharini Balasubramaniam. Controlling software architecture erosion: A survey. *Journal of Systems and Software*, 85(1):132–151, 2012. doi: 10.1016/j.jss.2011.07.036.

Elena Deza and Michel-Marie Deza. Chapter 17 - distances and similarities in data analysis. In E. Deza and M. M. Deza, editors, *Dictionary of Distances*, pages 217 – 229. Elsevier, Amsterdam, 2006. ISBN 978-0-444-52087-6. doi: 10.1016/B978-044452087-6/50017-2.

J. A. Díaz-Pace, A. Tommasel, and D. Godoy. [research paper] Towards anticipation of architectural smells using link prediction techniques. In *2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 62–71, Sep. 2018. doi: 10.1109/SCAM.2018.00015.

F. Arcelli Fontana, I. Pigazzini, R. Roveda, D.A. Tamburri, M. Zanoni, and E. Di Nitto. Arcan: A tool for architectural smells detection. In *2017 IEEE ICSA Workshops 2017, Gothenburg, Sweden, April 5-7, 2017*, pages 282–285, 2017.

Francesca Arcelli Fontana, Ilaria Pigazzini, Riccardo Roveda, and Marco Zanoni. Automatic detection of instability architectural smells. In *ICSME*, pages 433–437. IEEE Computer Society, 2016.

J. Garcia, D. Popescu, G. Edwards, and N. Medvidovic. Toward a catalogue of architectural bad smells. In *QoSA*, volume 5581 of *LNCS*, pages 146–162. Springer, 2009. ISBN 978-3-642-02350-7.

L. Hochstein and M. Lindvall. Combating architectural degeneration: A survey. *Inf. Softw. Technol.*, 47(10):643–656, 2005. ISSN 0950-5849.

Duc Minh Le, Daniel Link, Arman Shahbazian, and Nenad Medvidovic. An empirical study of architectural decay in open-source software. In *IEEE International Conference on Software Architecture, ICSA 2018, Seattle, WA, USA, April 30 - May 4, 2018*, pages 176–185. IEEE Computer Society, 2018. doi: 10.1109/ICSA.2018.00027.

David Liben-Nowell and Jon Kleinberg. The link-prediction problem for social networks. *J. Am. Soc. Inf. Sci. Technol.*, 58(7):1019–1031, May 2007. ISSN 1532-2882.

Linyuan Lu and Tao Zhou. Link prediction in complex networks: A survey. *Physica A*, 390(6):11501170, 2011.

C. Manteli, H. v. Vliet, and B. v. d. Hooff. Adopting a social network perspective in global software development. In *2012 IEEE Seventh International Conference on Global Software Engineering*, pages 124–133, 2012.

Radu Marinescu. Assessing technical debt by identifying design flaws in software systems. *IBM Journal of Research and Development*, 56(5):9, 2012a.

Radu Marinescu. Assessing technical debt by identifying design flaws in software systems. *IBM Journal of Research and Development*, 56(5):9, 2012b.

H. Melton and E. Tempero. An empirical study of cycles among classes in java. *Empir Soft Eng*, 12(4):389–415, 2007. ISSN 1573-7616.

Ran Mo, Yuanfang Cai, Rick Kazman, and Lu Xiao. Hotspot patterns: The formal definition and automatic detection of architecture smells. In *Proceedings of the 2015 12th Working IEEE/IFIP Conference on Software Architecture*, WICSA '15, pages 51–60, Washington, DC, USA, 2015. IEEE Computer Society. ISBN 978-1-4799-1922-2. doi: 10.1109/WICSA.2015.12.

Behnaz Moradabadi and Mohammad Reza Meybodi. A novel time series link prediction method: Learning automata approach. *Physica A: Statistical Mechanics and its Applications*, 482:422 – 432, 2017. ISSN 0378-4371. doi: 10.1016/j.physa.2017.04.019.

Viet Hung Nguyen and Le Minh Sang Tran. Predicting vulnerable software components with dependency graphs. In *Proceedings of the 6th International Workshop on Security Measurements and Metrics*, MetriSec '10, pages 3:1–3:8, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0340-8.

D. Di Nucci, F. Palomba, D. A. Tamburri, A. Serebrenik, and A. De Lucia. Detecting code smells using machine learning techniques: Are we there yet? In *2018 IEEE 25th SANER*, pages 612–621, 2018.

F. Palomba, A. De Lucia, G. Bavota, and R. Oliveto. Chapter four - anti-pattern detection: Methods, challenges, and open issues. In Atif Memon, editor, *Advances in Computers*, volume 95 of *Advances in Computers*, pages 201 – 238. Elsevier, 2014.

Serhat Peker and Altan Kocyigit. An adjusted recommendation list size approach for users' multiple item preferences. In Christo Dichev and Gennady Agre, editors, *Artificial Intelligence: Methodology, Systems, and Applications*, pages 310–319, Cham, 2016. Springer International Publishing. ISBN 978-3-319-44748-3.

Narayan Ramasubbu and Chris F. Kemerer. Managing technical debt in enterprise software packages. *IEEE Transactions on Software Engineering*, 40(8):758–772, 2014. doi: 10.1109/TSE.2014.2327027.

G. Salton and M. J. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, Inc., New York, NY, USA, 1986.

Darius Sas, Paris Avgeriou, and Francesca Arcelli Fontana. Investigating instability architectural smells evolution: an exploratory case study. In *To appear in Proceedings of the 35th IEEE International Conference on Software Maintenance and Evolution*, ICSME 2019. IEEE, 2019.

Bernhard Schölkopf, John C. Platt, John C. Shawe-Taylor, Alex J. Smola, and Robert C. Williamson. Estimating the support of a high-dimensional distribution. *Neural Comput.*, 13(7):1443–1471, July 2001. ISSN 0899-7667. doi: 10.1162/089976601750264965.

Ricardo Terra, Joao Brunet, Luis Miranda, Marco Tulio Valente, Dalton Serey, Douglas Castilho, and Roberto Bigonha. Measuring the structural similarity between source code entities. In *25th International Conference on Software Engineering and Knowledge Engineering (SEKE)*, pages 753–758, 2013.

Will Tracz. Refactoring for software design smells: Managing technical debt by girish suryanarayana, ganesh samarthyam, and tushar sharma. *ACM SIGSOFT Software Engineering Notes*, 40(6):36, 2015.

Yang Yang, Ryan N. Lichtenwalter, and Nitesh V. Chawla. Evaluating link prediction methods. *Knowl. Inf. Syst.*, 45(3):751–782, December 2015. ISSN 0219-1377.

Bo Zhou, Xin Xia, David Lo, and Xinyu Wang. Build predictor: More accurate missed dependency prediction in build configuration files. In *Proceedings of the 2014 IEEE 38th Annual Computer Software and Applications Conference*, COMPSAC '14, pages 53–58, Washington, DC, USA, 2014. IEEE Computer Society. ISBN 978-1-4799-3575-8.

Thomas Zimmermann and Nachiappan Nagappan. Predicting defects using network analysis on dependency graphs. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 531–540, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-079-1.