# *Applying Social Network Analysis Techniques to Architectural Smell Prediction*

**Dr. Antonela Tommasel**

ISISTAN, CONICET-UNICEN, Argentina

antonela.tommasel@isistan.unicen.edu.ar

CONICET

ISISTAN

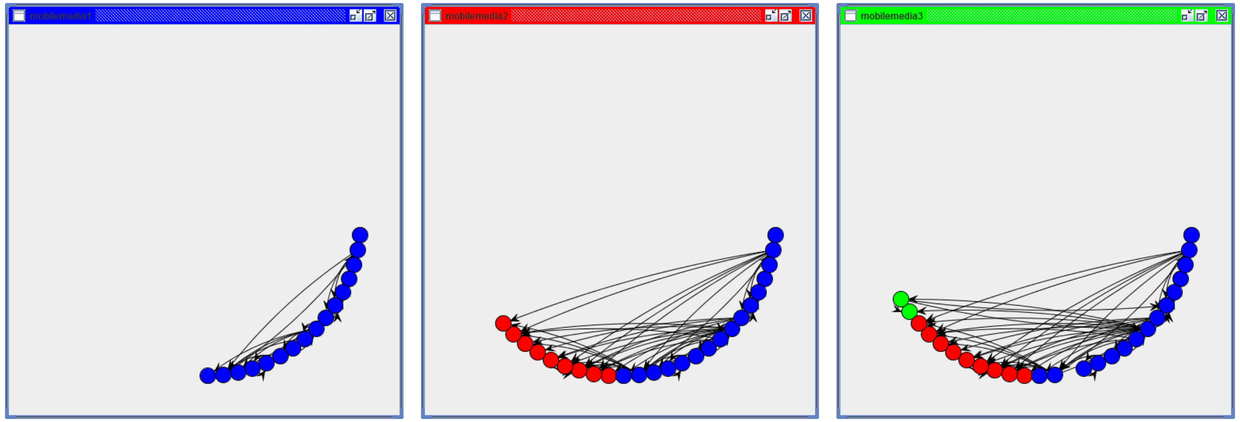
# Table of Contents

# Software Evolution & Dependencies

- As software systems evolve, the **amount and complexity of the interactions** amongst their components often increases.
  - More coupling.
  - **“*Undesired*” dependencies** amongst certain components (e.g., layer bridging, direct access to databases, cycles).
  - Degradation of intended design.

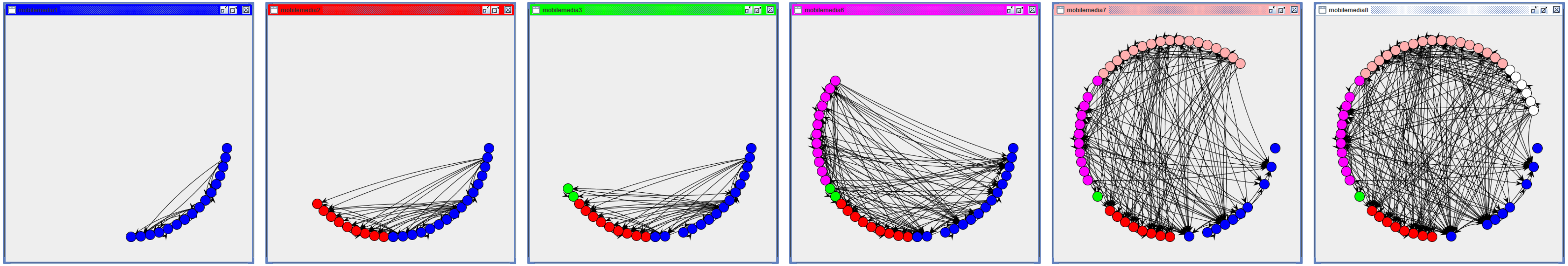
# Software Evolution & Dependencies

- As software systems evolve, the **amount and complexity of the interactions** amongst their components often increases.
  - More coupling.
  - **“*Undesired*” dependencies** amongst certain components (e.g., layer bridging, direct access to databases, cycles).
  - <u>Degradation</u> of intended design.

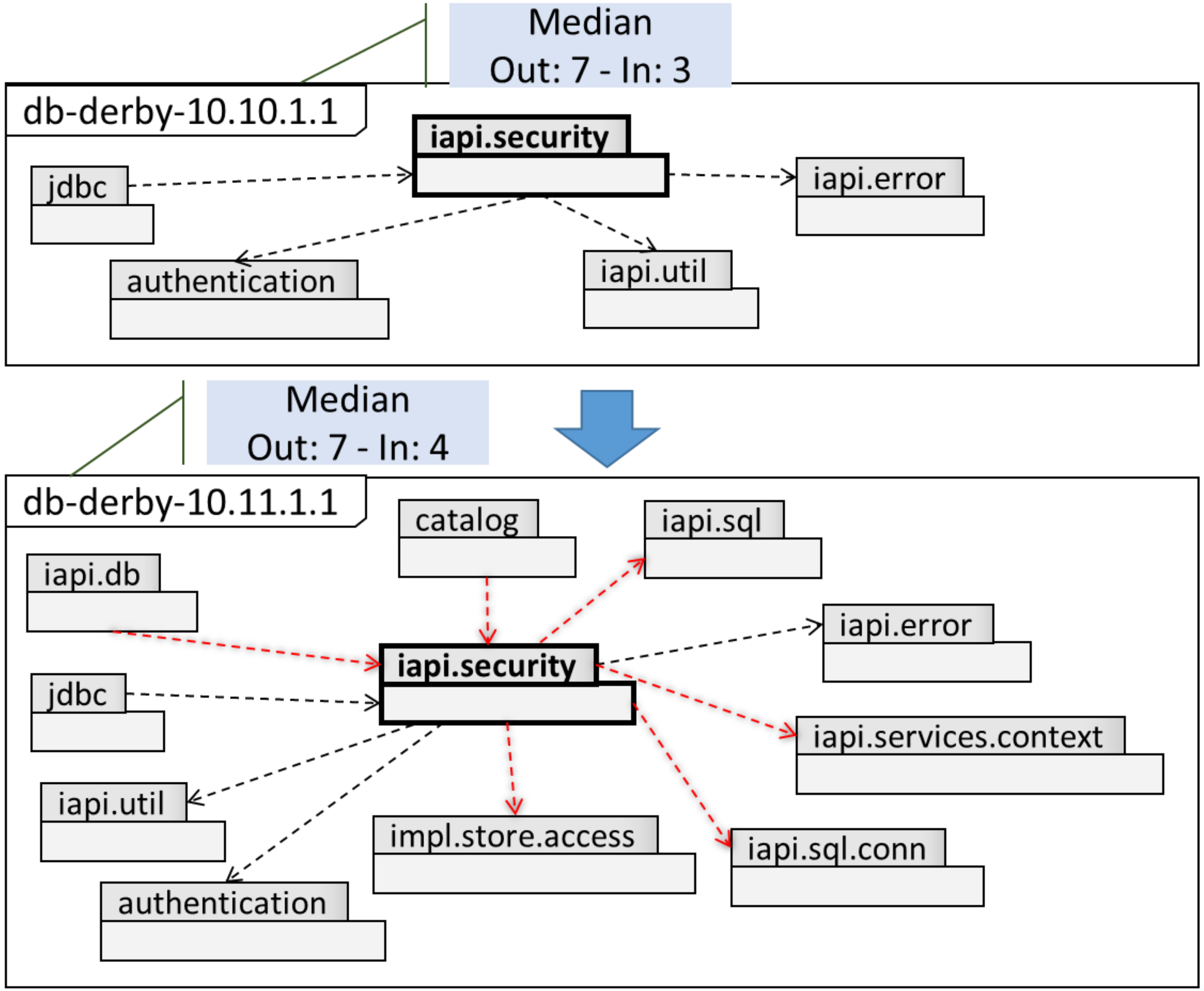


# Software Evolution & Dependencies

- As software systems evolve, the **amount and complexity of the interactions** amongst their components often increases.
  - More coupling.
  - **“*Undesired*” dependencies** amongst certain components (e.g., layer bridging, direct access to databases, cycles).
  - <u>Degradation</u> of intended design.

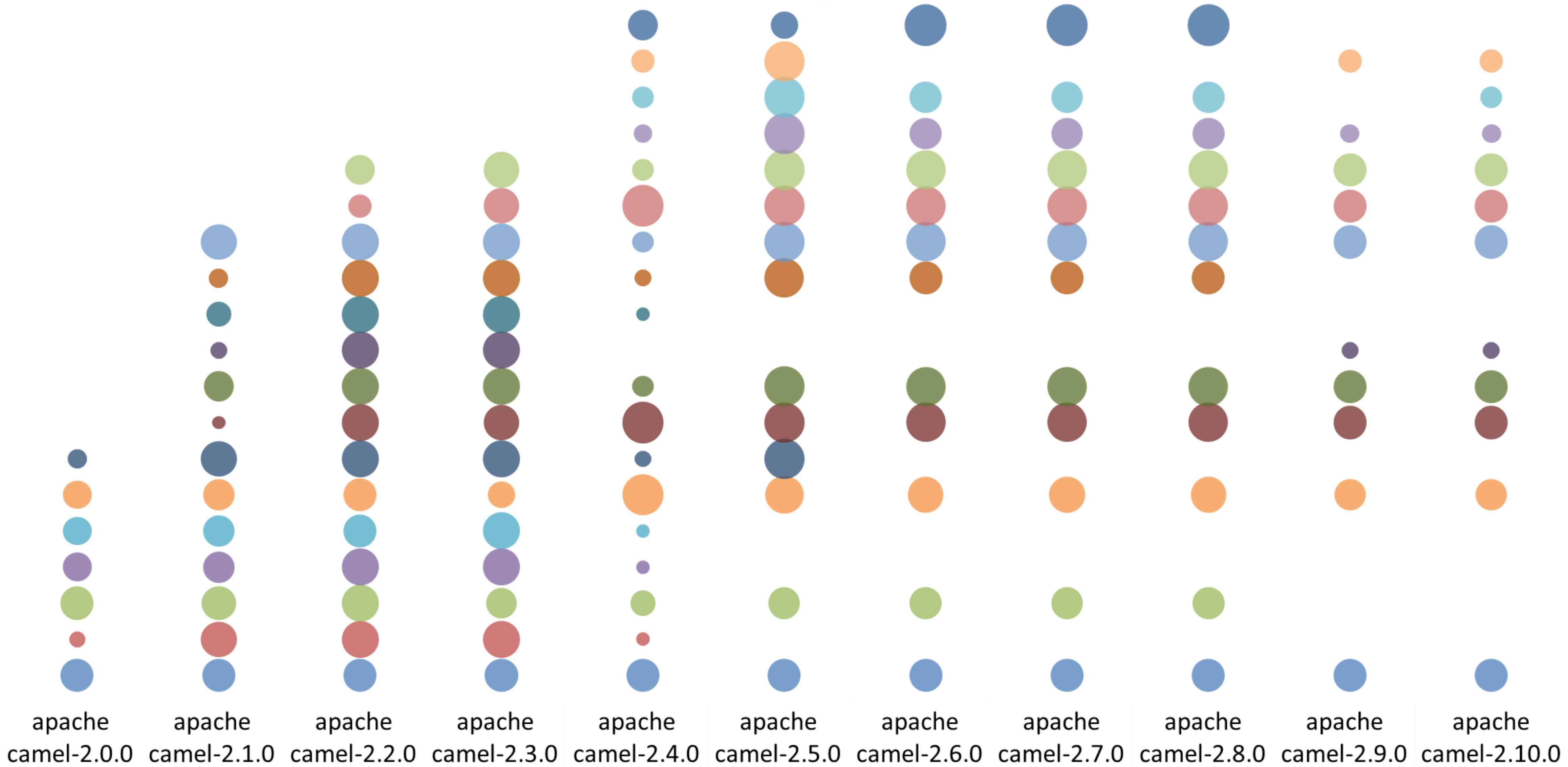


# Software Evolution & Dependencies




**iapi.security** becomes a hub in db-derby-10.11.1.1 due to the addition of new dependencies

# Software Evolution & Dependencies



# Software Evolution & Dependencies

- Conscious efforts must be made to stop (or alleviate) degradation.
  - Plan for corrective actions (e.g., refactoring).
  - Monitor system health (e.g., via metrics).
  - Conformance checks.

# Software Evolution & Dependencies

- Conscious efforts must be made to stop (or alleviate) degradation.
  - Plan for corrective actions (e.g., refactoring).
  - Monitor system health (e.g., via metrics).
  - Conformance checks.

The early detection of architectural smells is **important** for architects, so that they can **plan ahead** for actions that **preserve** the **quality** of the system.

# What can we do about it?

- Different tools available
  - LattixDSM, SonarQube, SonarGraph, JITTAC.

- Tools provide "**just-in-time**" detection capabilities.

# What can we do about it?

- Different tools available
  - LattixDSM, SonarQube, SonarGraph, JITTAC.

- Tools provide "**just-in-time**" detection capabilities.

- Identification of problems **once they occurred** in the system!
  - Might help developers to fix specific smells in a local design context.
  - Not always helpful when architects want to assess the different smells of the system with a global design perspective.

# What can we do about it?

- In a forward-looking scenario, **architects** would want to know:
    - Which modules are likely to be coupled in the near future.
    - Which smells are more harmful for the system.

# What can we do about it?

- In a forward-looking scenario, **architects** would want to know:
  - Which modules are likely to be coupled in the near future.
  - Which smells are more harmful for the system.

This architecture-level analysis requires **to anticipate dependency-related problems** in order **to proactively look for solutions**.

**Predict when a dependency-related problem is likely to manifest!**

# Social Network Analysis to the Rescue!

- In the last decade, research has been devoted to study:
  - How smells are **introduced**.
  - How smells **evolve**.
  - What their **effect** is on program comprehension.

- However, research on how to **predict the appearance** of architectural smells **has been scarce**.

# Social Network Analysis to the Rescue!

- In the last decade, research has been devoted to study:
  - How smells are **introduced**.
  - How smells **evolve**.
  - What their **effect** is on program comprehension.

- However, research on how to **predict the appearance** of architectural smells **has been scarce**.

- A particular graph-based approach is Social Networks Analysis (SNA), which has been used for modelling both nature and human phenomena.

*SNA techniques can predict links that yet do not exist between pairs of nodes in a network.*

# Social Network Analysis to the Rescue!

## ...and Software Engineering?

We hypothesise that **software systems** and
their underlying architectures **behave** as **social networks**.

- **Evidence** that the **topological** features of **dependency graphs** can reveal interesting properties of the software system under analysis.
- Nonetheless, ***SNA*** techniques has not yet greatly exploited in the Software Engineering community.

We argue that **Social Network Analysis** techniques need
to be revisited with respect to **software dependency prediction**!

# Social Network Analysis to the Rescue!

## …and Software Engineering?

- **RQ1.** How do architectural **smells evolve** over system versions, in terms of increasing or decreasing their dependency configurations?

- **RQ2.** What criteria are useful for **assessing similarity of design elements** with respect to link prediction?

- **RQ3.** Can **past system versions affect, and improve the predictions** of, the design structure of a future version?

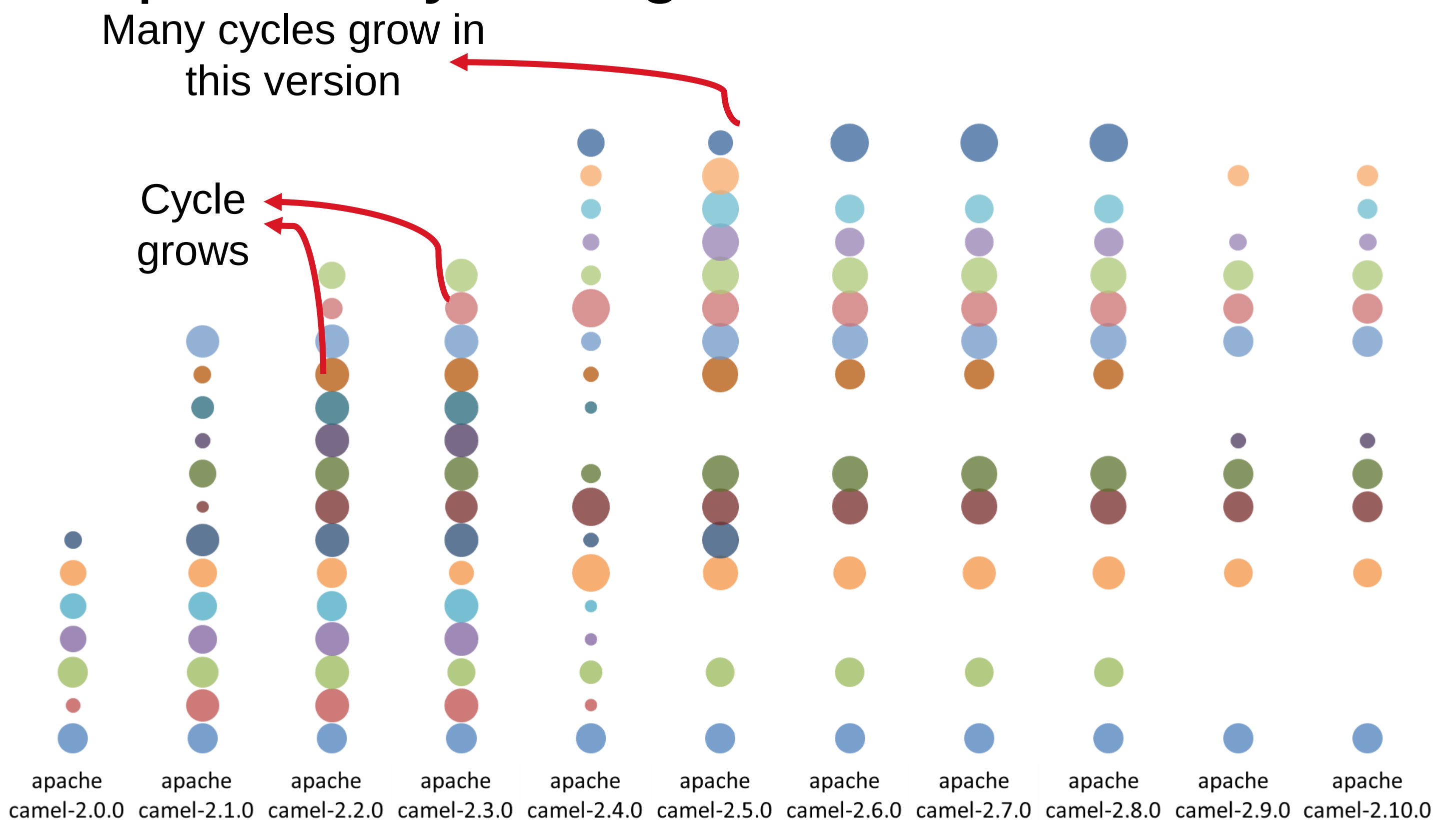
- **RQ4.** To what extent **Machine Learning** techniques can **aid** in the **prediction** of architectural smells?

# Social Network Analysis to the Rescue!

## …and Software Engineering?

- **RQ1.** How do architectural **smells evolve** over system versions, in terms of increasing or decreasing their dependency configurations?

  - For instance, given a cycle, it can be analysed whether the cycle will grow in successive versions.
  - Changes should be tracked.
  - A similar analysis can be performed for other architectural smells



# Social Network Analysis to the Rescue!

## …and Software Engineering?

- **RQ2.** What criteria are useful for **assessing similarity of design elements** with respect to link prediction?

- Traditional LP techniques are primary based on determining whether two elements are likely to interact in the future based on a similarity score.
- Similarity is supported by the homophily principle.
  - Similar elements are more likely to interact than dissimilar ones.
- In a software design context, homophily would mean, for example, that similar modules are more likely to establish dependencies than dissimilar modules.

# Social Network Analysis to the Rescue!

## …and Software Engineering?

- **RQ2.** What criteria are useful for **assessing similarity of design elements** with respect to link prediction?

- Traditional LP techniques are primary based on determining whether two elements are likely to interact in the future based on a similarity score.
- Similarity is supported by the homophily principle.
  - Similar elements are more likely to interact than dissimilar ones.
- In a software design context, homophily would mean, for example, that similar modules are more likely to establish dependencies than dissimilar modules.

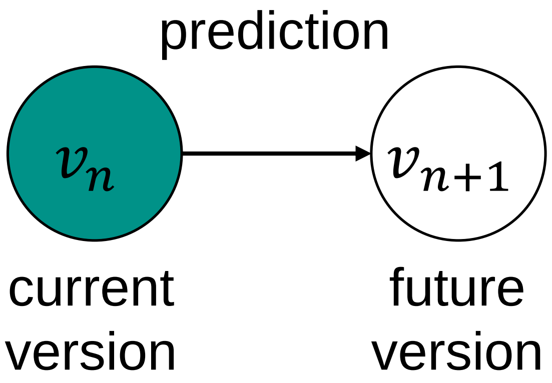
- **Standard Topological Similarity Metrics**
  - Common Neighbours
  - Adamic Adar
  - Katz
  - …
- **Source-code Similarity Metrics**
  - Kunczynsky
  - Relative Matching
  - …
- **Content-based Similarity**

# Social Network Analysis to the Rescue!

## …and Software Engineering?

- **RQ3.** Can **past system versions affect, and improve the predictions** of, the design structure of a future version?

Link Prediction can leverage on information from the **current version** to **predict** changes in the next version



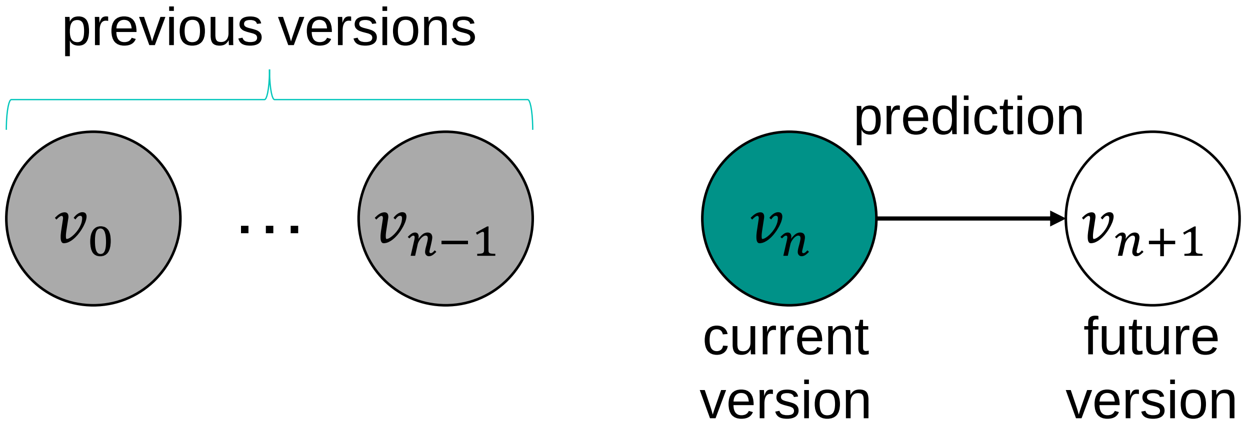

Use statistical techniques to give computer systems the ability to "learn" (on a specific task) with data, without being explicitly programmed

# Social Network Analysis to the Rescue!

## …and Software Engineering?

- **RQ3.** Can **past system versions affect, and improve the predictions** of, the design structure of a future version?

Link Prediction can leverage on information from the **past versions** to **predict** changes in the next version




Dynamic SNA (i.e., observations of the graph at different time periods) + features = Learn a robust ML model able to predict new links.

# Social Network Analysis to the Rescue!

## …and Software Engineering?

- **RQ4.** To what extent **Machine Learning** techniques can **aid** in the **prediction** of architectural smells?

- The simple detection of predicted dependencies might not be sufficient to determine how architectural smells will behave on the future.

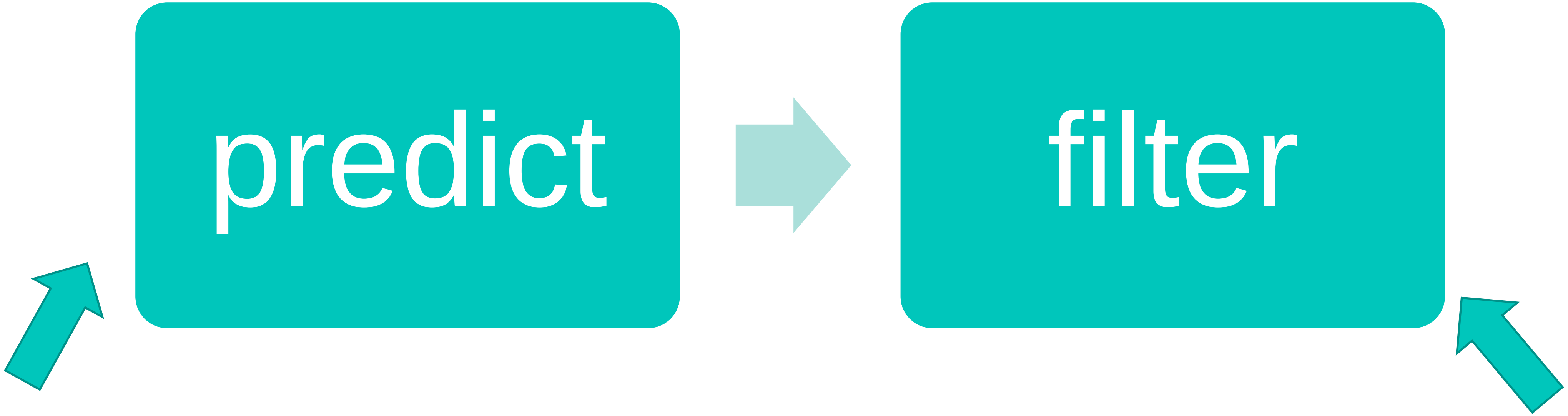
- If two usage relations for modules are to be predicted, it does necessarily mean that it will cause a new smell to appear.

- For this reason, each type of smell requires to consider the predicted dependencies in context.

# Social Network Analysis to the Rescue!

## …and Software Engineering?

- **RQ4.** To what extent **Machine Learning** techniques can **aid** in the **prediction** of architectural smells?

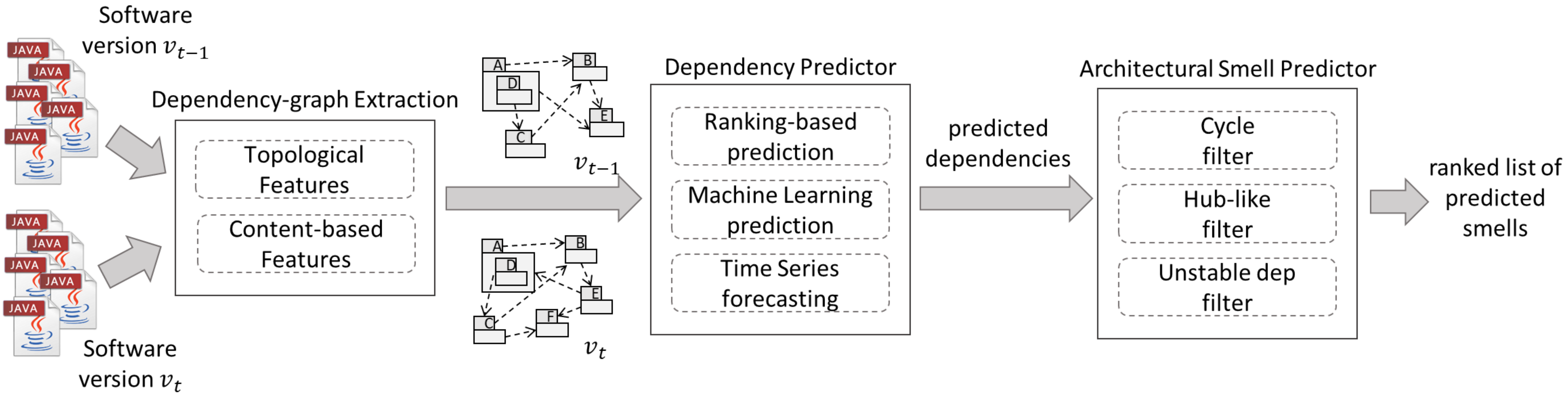



predict the appearance of new dependencies in the next system version

filter the predicted dependencies according to the characteristics of specific types of smells.

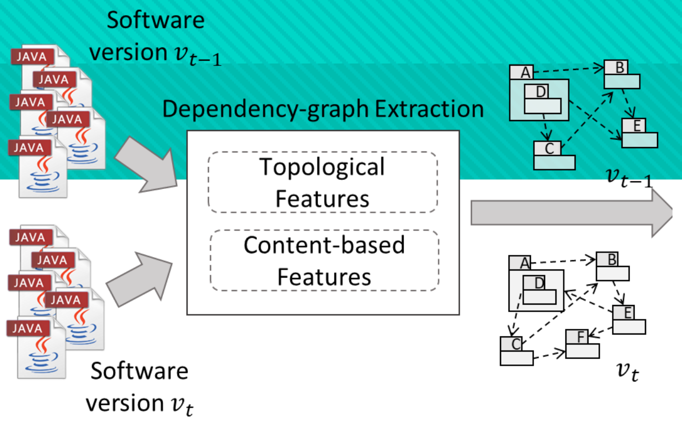
# Table of Contents

# Prediction Overview



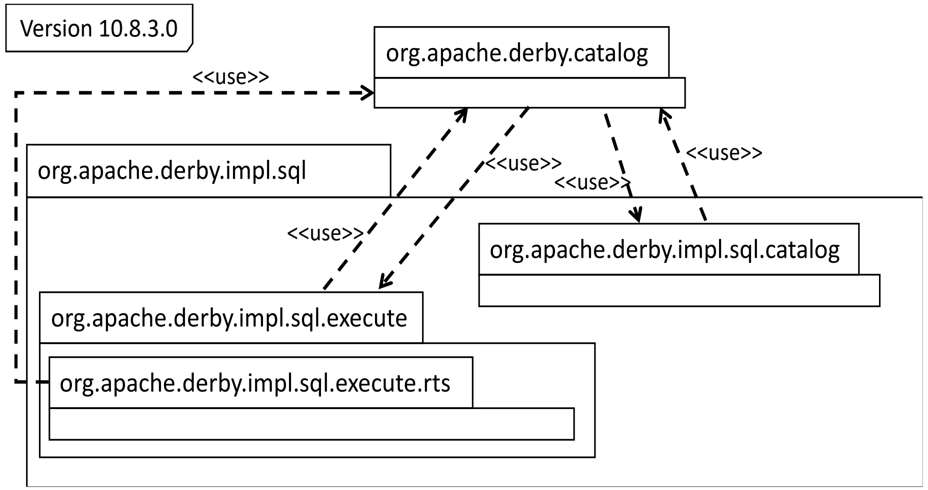
# Prediction Overview

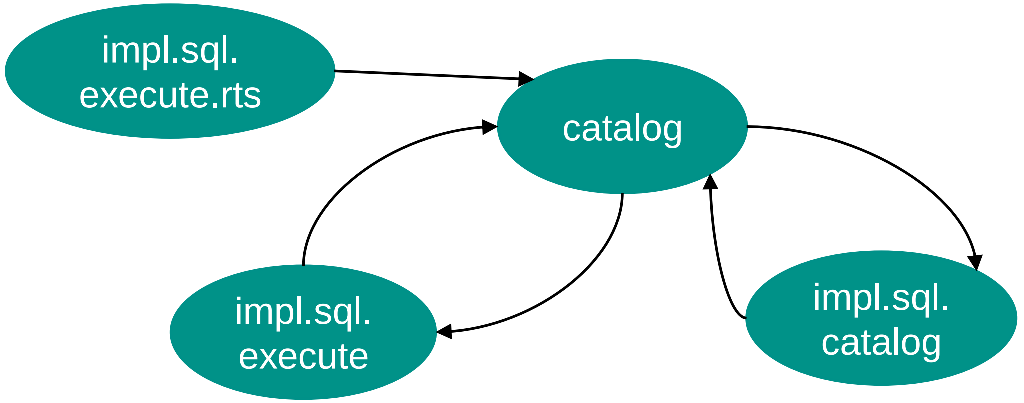
## Dependency-graph Extraction




- Need to transform the system under analysis into a dependency graph that captures the architecture view under analysis.

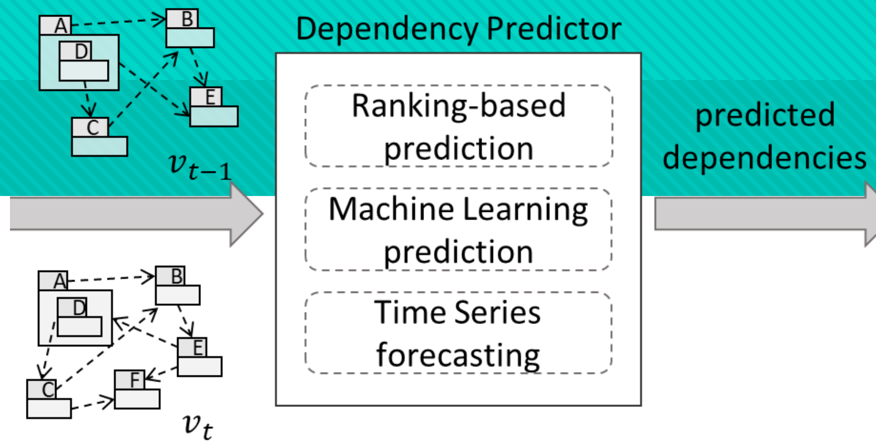
- Build a graph $DG(V, E)$ for system version $n$, where:
  - Each node $v$ in $V$ is Java package, and each edge *e* in *E* is a usage relationship between a pair of packages $v_1$ and $v_2$.
  - Compute $score(v_1, v_2)$ to assess similarity.



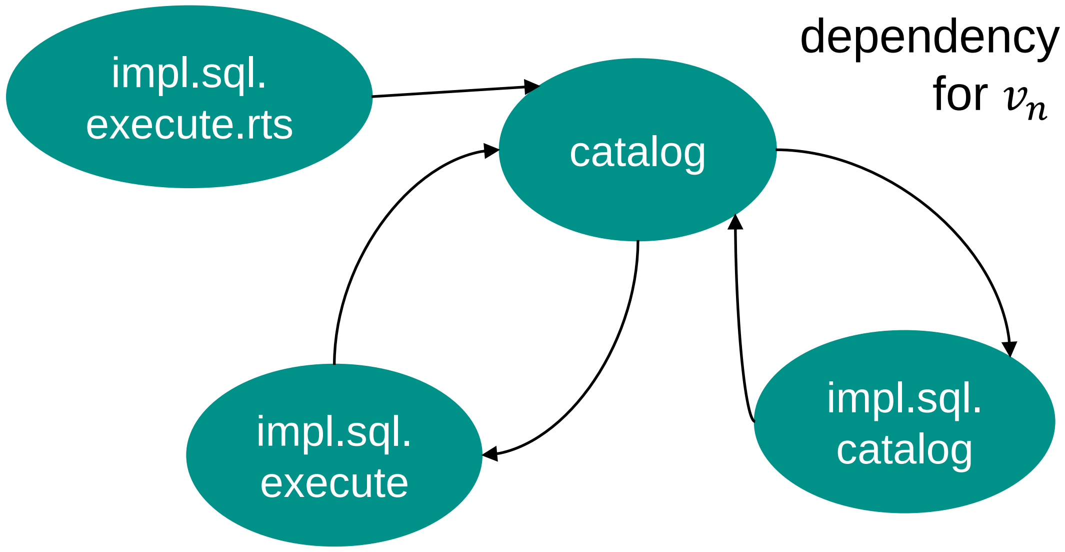



# Prediction Overview

## Dependency Predictor




dependency graph for $v_n$

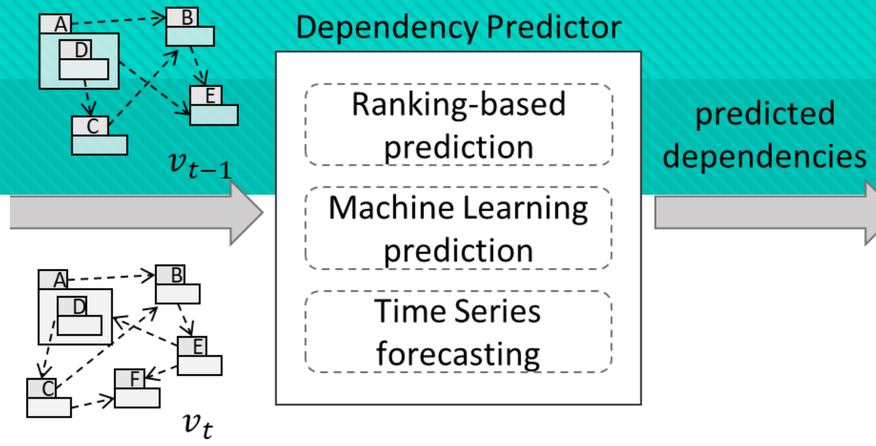



dependency graph for $v_n$

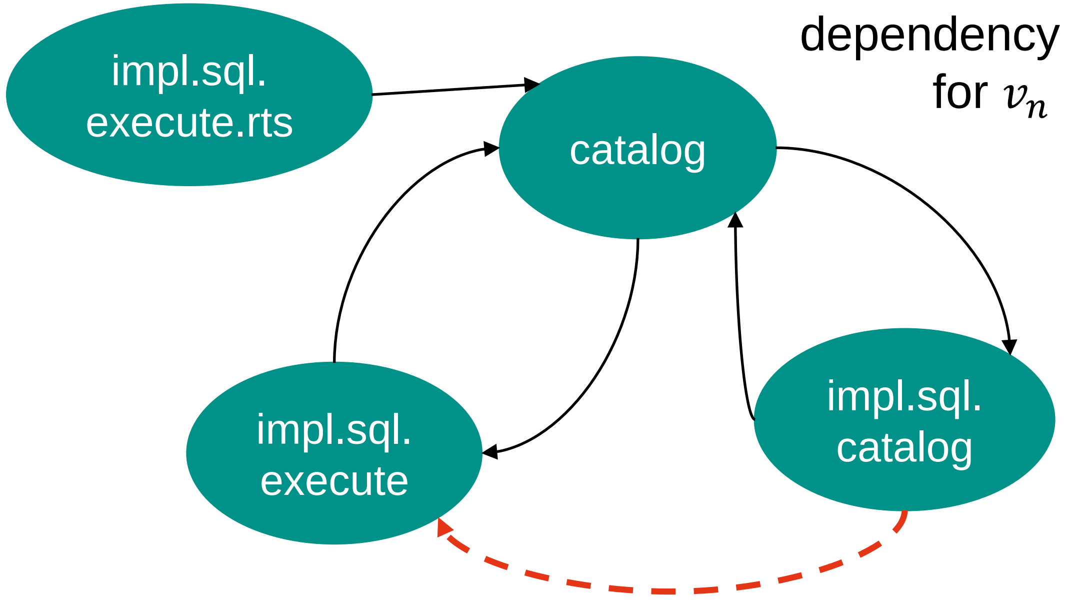
*To what extent we can leverage on information from software versions to predict likely dependencies in the next version, for those pairs of modules that exist in the analysed versions.*

# Prediction Overview

## Dependency Predictor




dependency graph for $v_n$



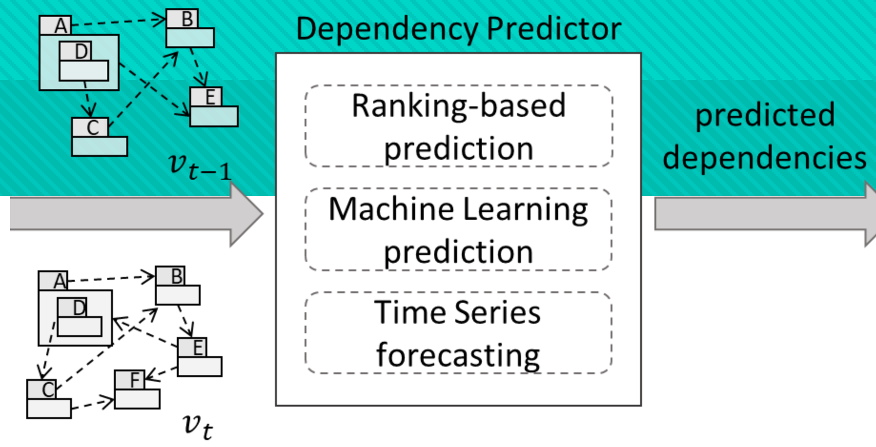

dependency graph for $v_n$

Will this dependency appear on $v_{n+1}$?

*To what extent we can leverage on information from software versions to predict likely dependencies in the next version, for those pairs of modules that exist in the analysed versions.*

# Prediction Overview

## Dependency Predictor

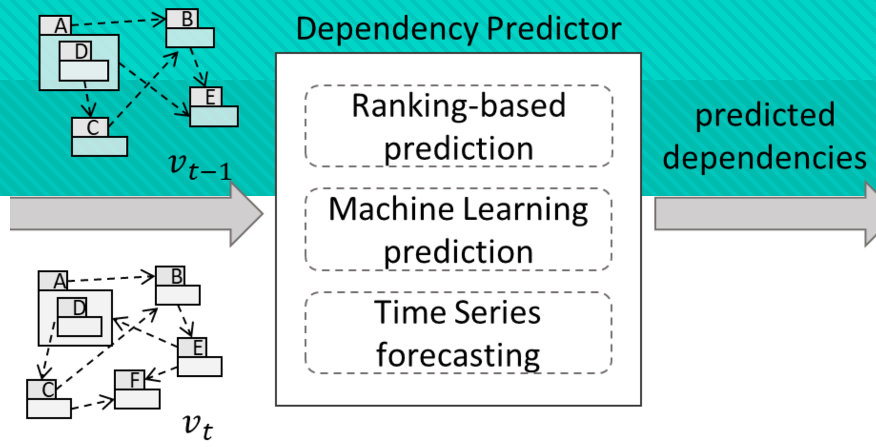



**Ranking-based prediction**

- For a package $p$, a **ranking** of packages is built, based on their chance of having a future dependency with $p$, according to a **similarity** metric.

- Most techniques rely on graph topological features that **assess similarity between pairs of nodes.**

# Prediction Overview

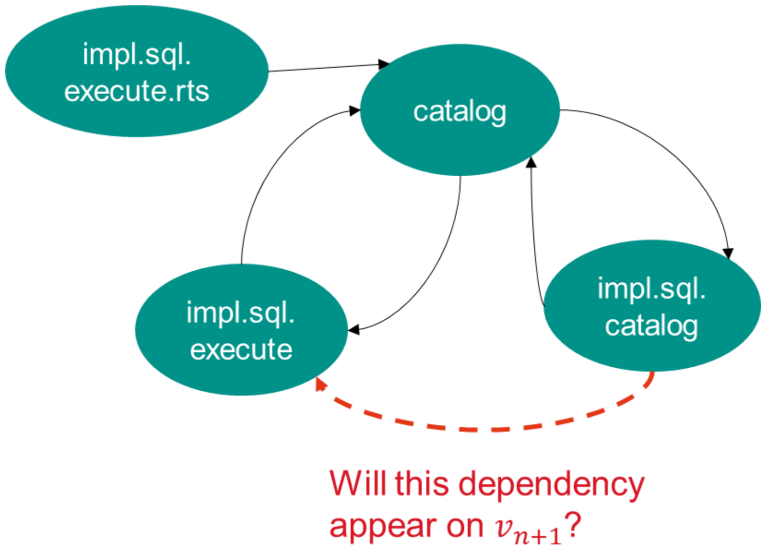



## Dependency Predictor

**Ranking-based prediction**

- For a package $p$, a **ranking** of packages is built, based on their chance of having a future dependency with $p$, according to a **similarity** metric.

- Most techniques rely on graph topological features that **assess similarity between pairs of nodes.**

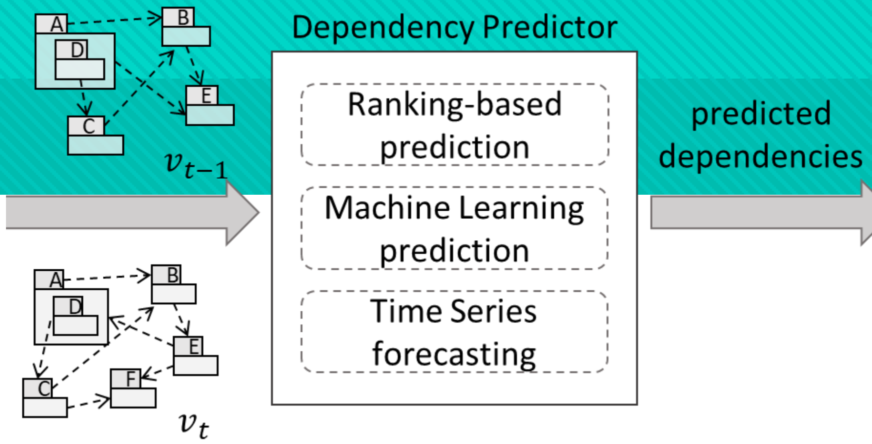



**Output for $v_{n+1}$**

| Ranking | Common Neighbours | Adamic-Adar |
|---|---|---|
| 1 | impl.sql | **impl.sql.execute** |
| 2 | **impl.sql.execute** | impl.sql |
| 3 | impl.sql.conn | impl.sql.con |
| 4 | impl.db | impl.db |
| 5 | impl.store.raw.data | impl.jdbc |

# Prediction Overview

## Dependency Predictor

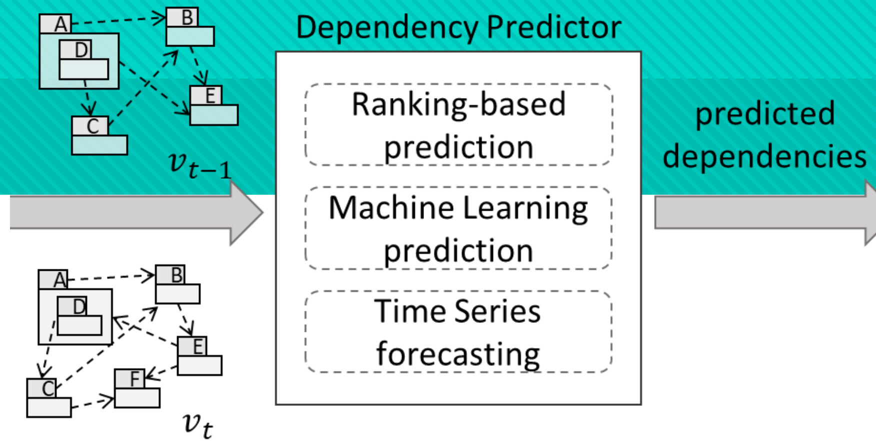



**Machine Learning prediction**

A **binary classifier** is trained using the topological information provided by a given graph version.

- An instance for the classifier consists of:
  - A pair of **nodes**.
  - A list of **features** (e.g., structural metrics) for the pair.
  - A **label** indicating if the nodes are linked (positive class) or not (negative class).

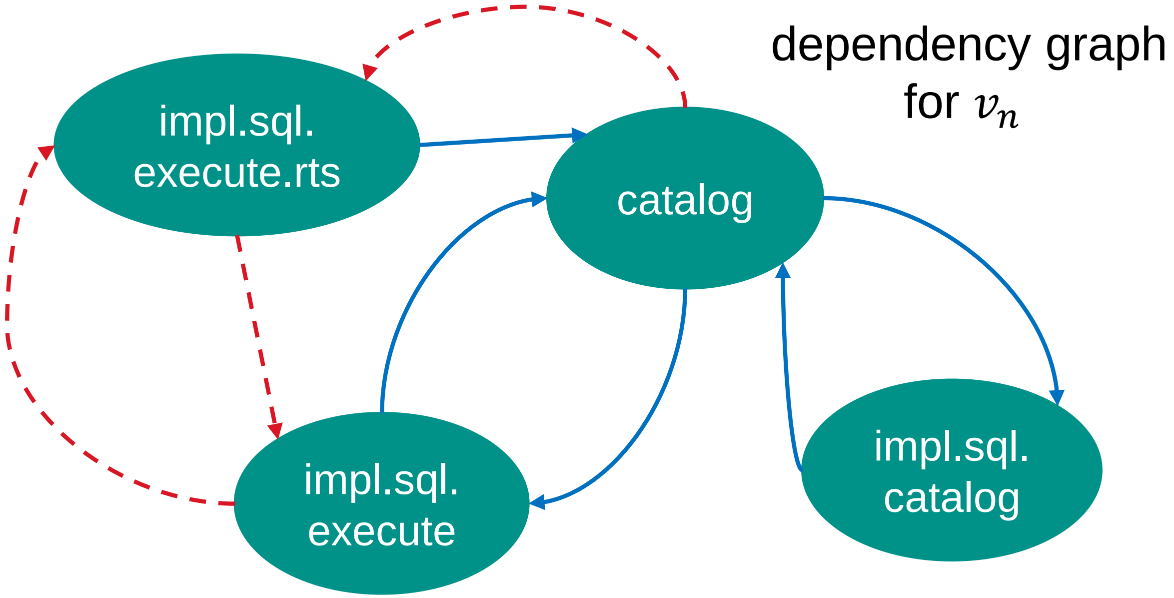
# Dependency Predictor



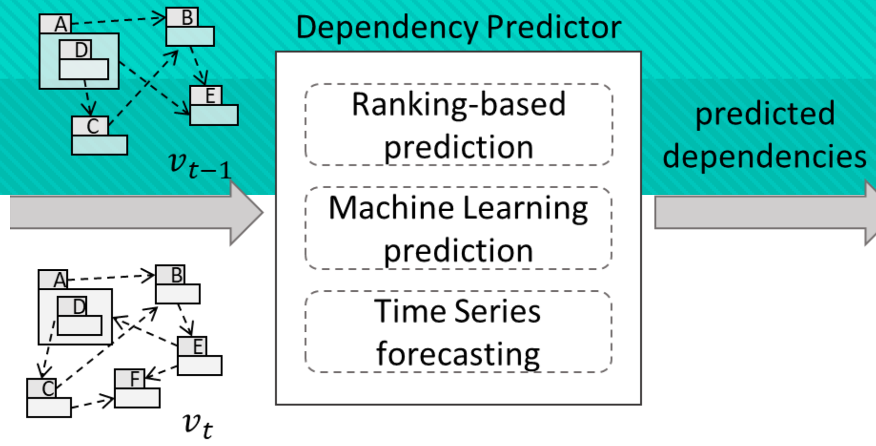

**Machine Learning prediction**

- **Existing dependencies** are used to compute features for instances of the positive class.
- **Missing dependencies** are used to compute features for instances of the negative class.

- Both training and test sets need to be defined.
  - The training set considers the known structure of $v_n$.
  - The test set considers the full graph of $v_{n+1}$.

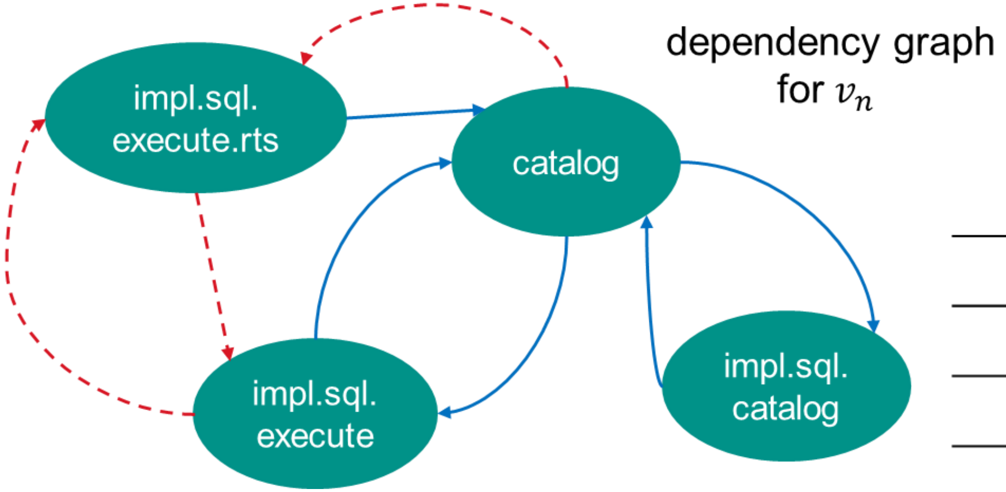


# Prediction Overview




## Dependency Predictor

**Machine Learning prediction**

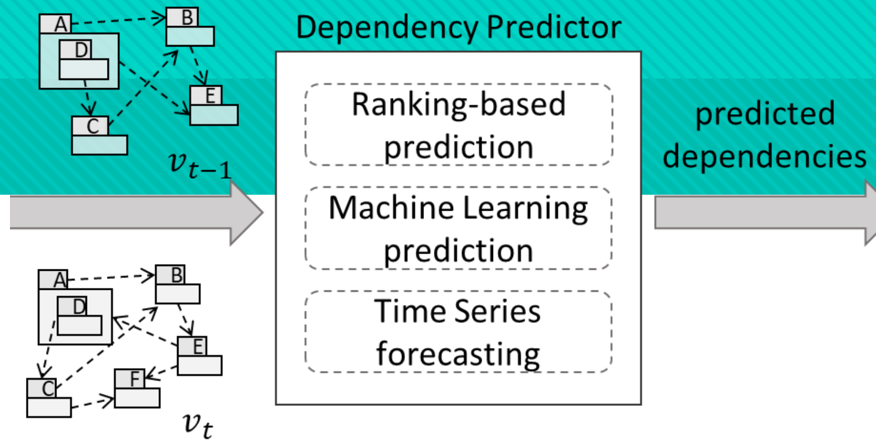



| source | target | source uses target? | Common Neighbours | Adamic Adar |
|---|---|---|---|---|
| catalog | impl.sql.execute | yes | 1 | 3.09 |
| catalog | impl.sql.execute.rts | no | 0.25 | 3.75 |
| impl.sql.catalog | catalog | yes | 0 | 1.66 |
| impl.sql.catalog | impl.sql.execute.rts | no | 0.33 | 3.09 |
| impl.sql.execute.rts | catalog | yes | 0.33 | 3.09 |

**Output**
$v_{n+1}$

**impl.sql.catalog uses impl.sql.execute.rts?**
**yes/no**

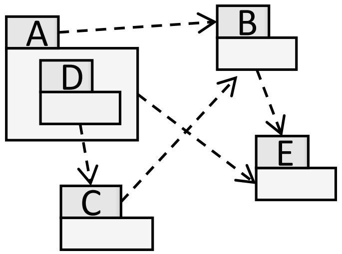
# Prediction Overview




## Dependency Predictor

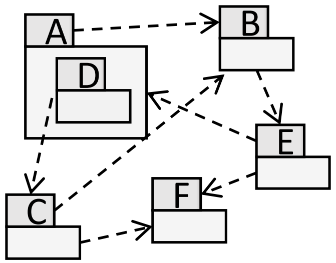
### Time Series forecasting

dependency graph for $v_{n-1}$




| source target | source uses target? | Common Neighbours |
|---|---|---|
| A – B | true | 0.233 |
| A – D | false | 0.518 |
| C – B | true | 0.289 |
| A – E | true | 0.235 |
| … | … | … |
| B – D | false | 0.505 |

dependency graph for $v_n$



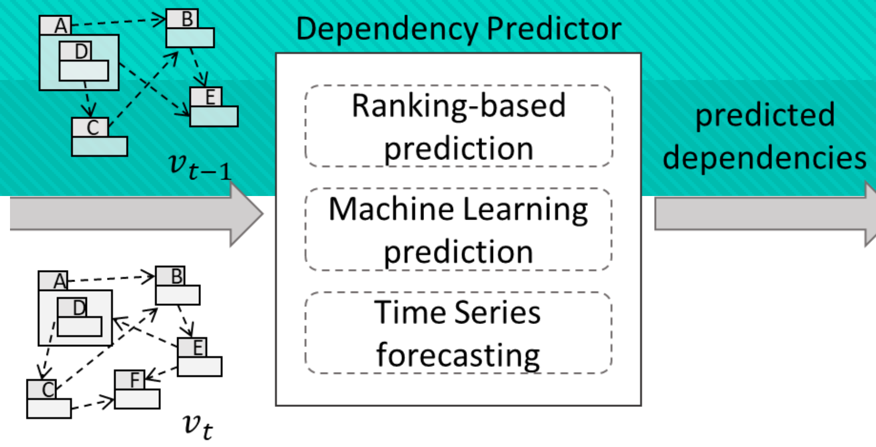

| source target | source uses target? | Common Neighbours |
|---|---|---|
| A – B | true | 0.353 |
| A – D | false | 0.618 |
| C – B | true | 0.389 |
| A – E | true | 0.385 |
| … | … | … |
| B – D | false | 0.605 |
| E - F | true | 0.171 |
| C - F | true | 0.1 |

estimation for $v_{n+1}$

| source target | source uses target? | Common Neighbours |
|---|---|---|
| A – B | ? | 0.453 |
| A – D | ? | 0.718 |
| C – B | ? | 0.289 |
| A – E | ? | 0.685 |
| … | … | … |
| B – D | ? | 0.805 |
| E - F | ? | 0.171 |
| C - F | ? | 0.11 |

We are **not yet predicting** new dependencies, but **estimating** the features' scores based on previous versions.

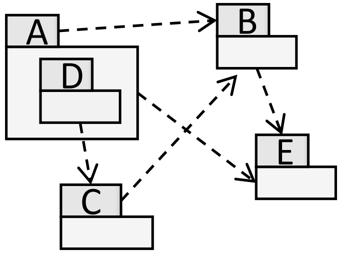
# Prediction Overview



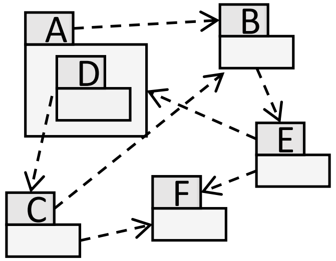

## Dependency Predictor

**Time Series forecasting**

dependency graph for $v_{n-1}$




dependency graph for $v_n$




estimation for $v_{n+1}$

| source target | source uses target? | Common Neighbours |
|---|---|---|
| A – B | true | 0.233 |
| A – D | false | 0.518 |
| C – B | true | 0.289 |
| A – E | true | 0.235 |
| … | … | … |
| B – D | false | 0.505 |

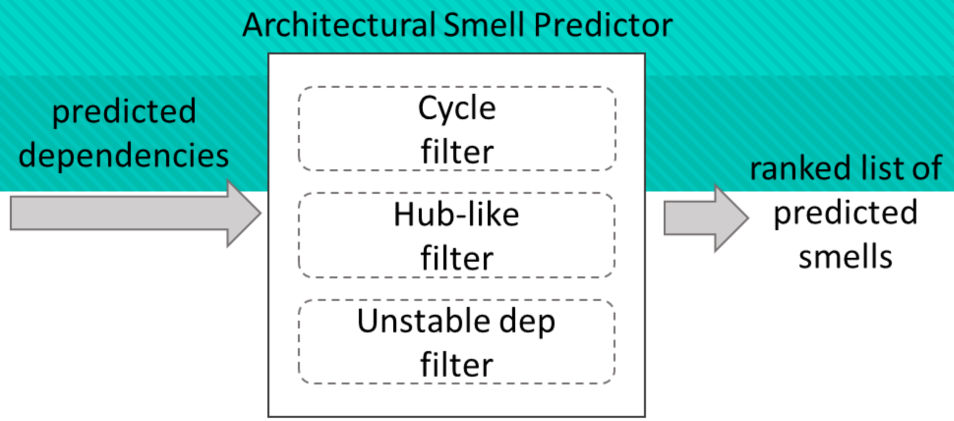
| source target | source uses target? | Common Neighbours |
|---|---|---|
| A – B | true | 0.353 |
| A – D | false | 0.618 |
| C – B | true | 0.389 |
| A – E | true | 0.385 |
| … | … | … |
| B – D | false | 0.605 |
| E - F | true | 0.171 |
| C - F | true | 0.1 |

| source target | source uses target? | Common Neighbours |
|---|---|---|
| A – B | ? | 0.453 |
| A – D | ? | 0.718 |
| C – B | ? | 0.289 |
| A – E | ? | 0.685 |
| … | … | … |
| B – D | ? | 0.805 |
| E - F | ? | 0.171 |
| C - F | ? | 0.11 |

- Prediction is based on a classifier **trained** with the **last known version of the system**, $v_n$.
- Predictions is based on **the estimated feature** scores for $v_{n+1}$.

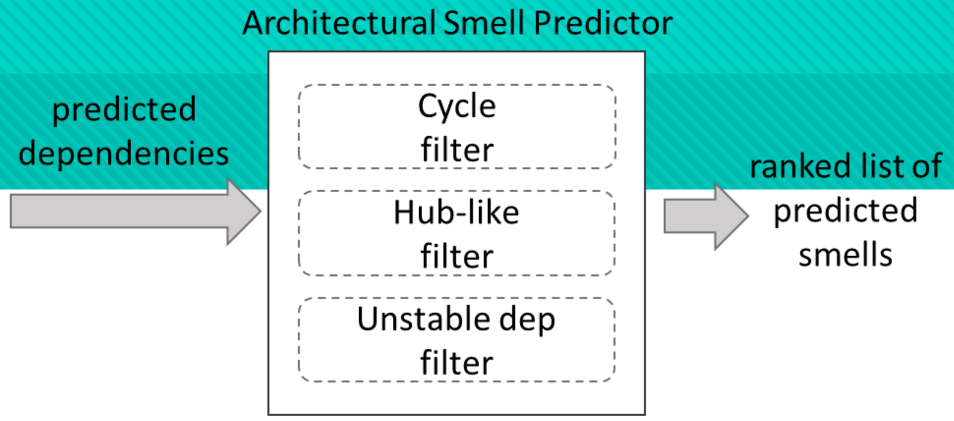
# Prediction Overview

## Architectural Smell Predictor




- The prediction of a dependency is not enough to predict the appearance of an architectural smell.
  - **Not every predicted dependency might cause an smell to emerge.**
- Predicted dependencies undergo a filtering process.
  - Filters are **smell-dependent**.

# Prediction Overview




## Architectural Smell Predictor

### Cycle Filter

- Considers only predicted dependencies that would lead to the closure of new cycles in $v_{n+1}$.

### Hub-like Filter

- Only the packages incidental to the predicted edges that fit with the hub definition are actually predicted.
  - Allow the detection of those nodes becoming hubs due to the addition of new dependencies.
  - Disregard nodes that might become hubs due to changes in the overall structure of the dependency graph.

### Unstable Dependency Filter

- Only the packages incidental to the predicted edges that fit with the unstable dependency definition are actually predicted.

# Table of Contents

# Expected Contributions

- We build on the advances of SNA and Machine Learning techniques to provide insights regarding how software design structures evolve, particularly in terms of degradation symptoms.

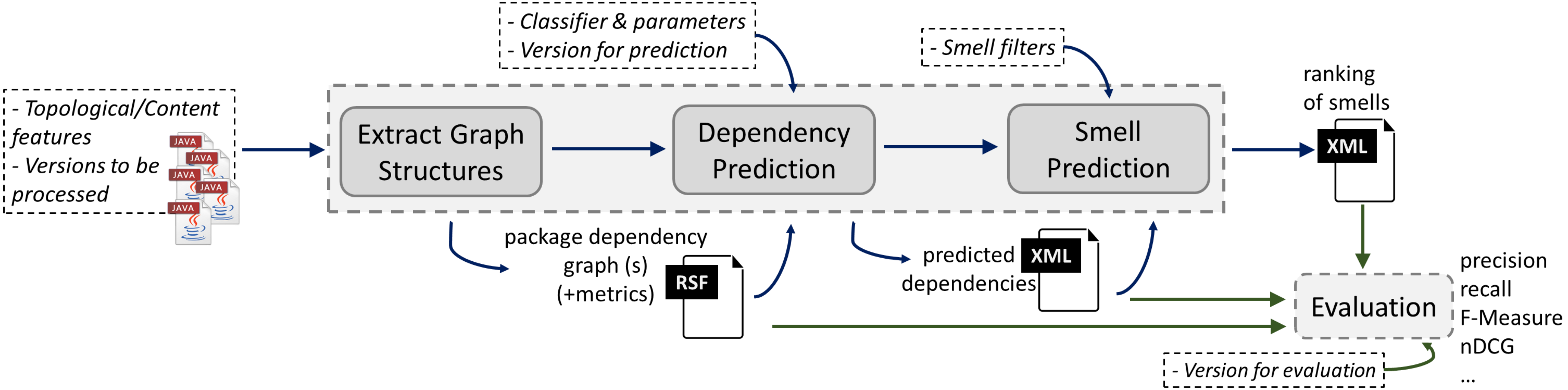
- We propose a predictive approach that would allow architects to spot a set of dependency-related problems that are likely to appear in a given system.

- The approach could also allow simulating dependency-related to observe their potential effects in future system versions.

- A tool !

# Expected Contributions

## The tool: ASPredictor



# Lessons Learned

## What have we done so far?

- Ongoing research has primary focused on the definition of the dependency graph and the evaluation of the dependency predictor.
- Complemented the definition of the dependency graph with a statistical analysis of software versions and the evolution of SNA metrics.
- Analysed how past decisions reflected in the software structure affect the future occurrence of dependencies, and smells thereof.
- Analysed the descriptive power of both topological and content-based features for defining the similarity of components.
- Smell prediction focused on cycles and hubs.

# Lessons Learned

## What do we do now?

- Perform a systematic study with more systems and other dependency-based smells.
- The prediction capabilities are sensitive to the prediction model.
  - Analyse and extend the set of features used.
  - Considering software specific-metrics?
- Smells might not be harmful.
  - How can we train a model to discard them?

# We are far from finished…

- Can communities help boost predictions?
- More features.
  - Design metrics? OO metrics? Global characteristics of smells?
- Analyse other dependency-based problems!
  - Analyse other types of smells?
- Can we predict the appearance of new nodes (e.g. new packages, classes)?
- Can we predict the disappearance of dependencies?

… and a lot more!

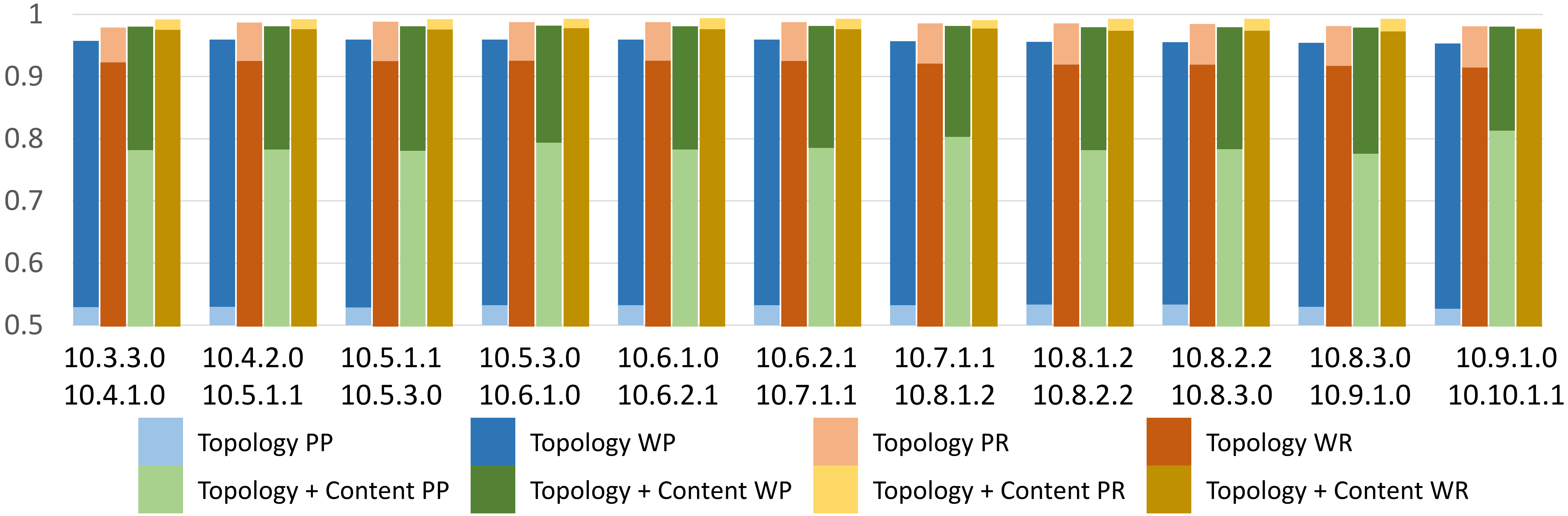
# Thanks!

## Questions?

Contact me at: antonela.tommasel@isistan.unicen.edu.ar

# Preliminary Results

## Dependency Prediction – Comparing Feature Types

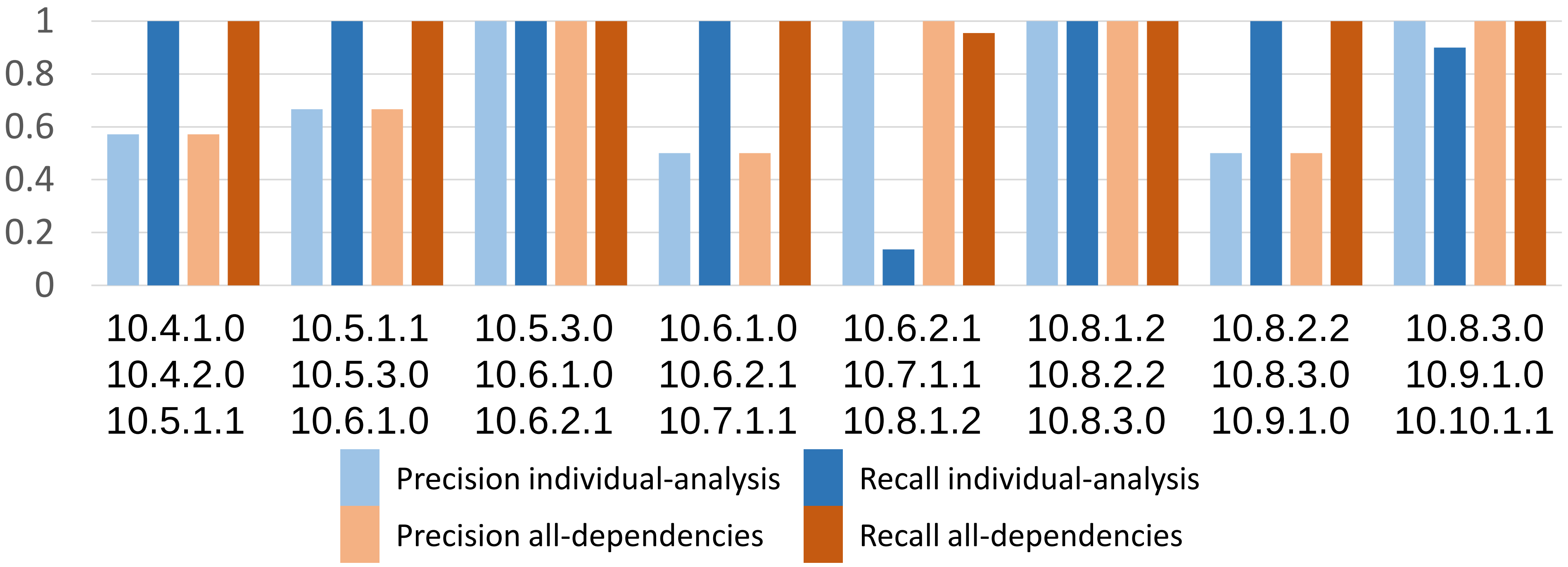



- Adding content-based features increased the quality of the predicted dependencies.
  - Average improvements of 27%.
- Finding most future dependencies, but also finds false ones.

# Preliminary Results

## Smell Prediction - Cycles

Apache Derby

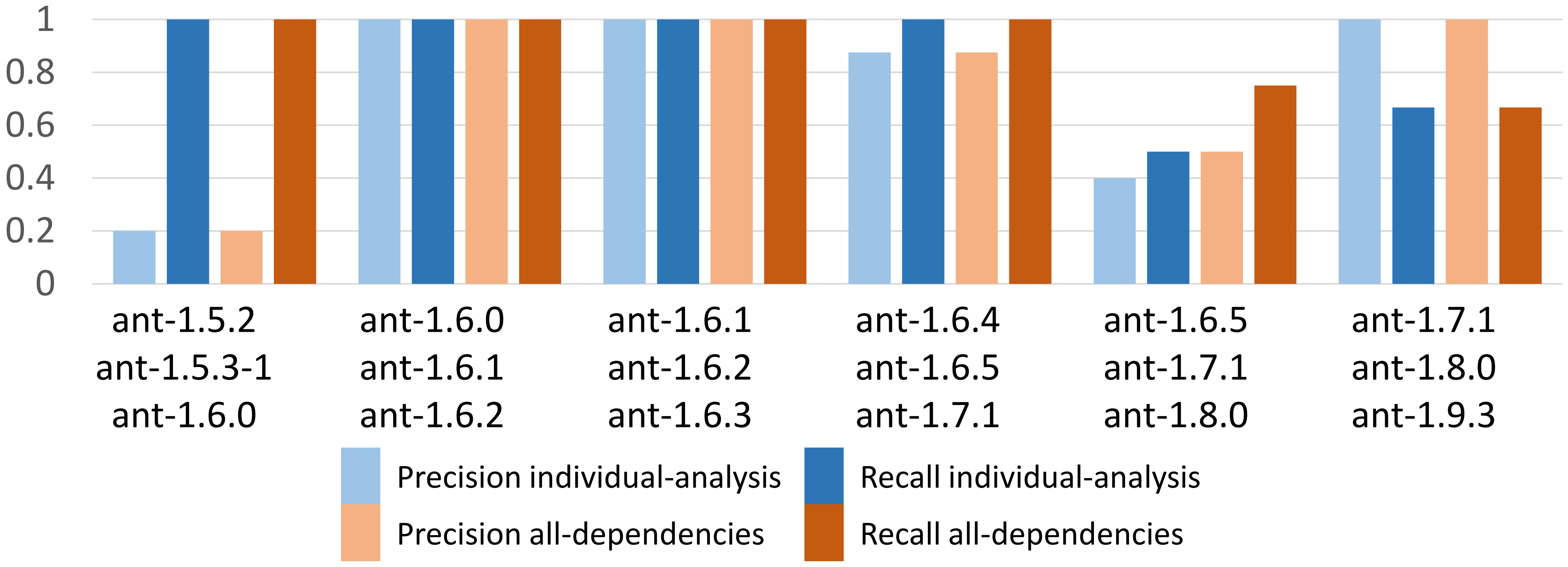



- In most cases recall is almost perfect (almost every new dependency leading to the closure of a quasi-cycle was found).
- Precision indicates that some mistaken dependencies are also predicted.
  - At most 5 mistaken predictions (0.06% of total dependencies).

# Preliminary Results

## Smell Prediction - Cycles

Apache Ant

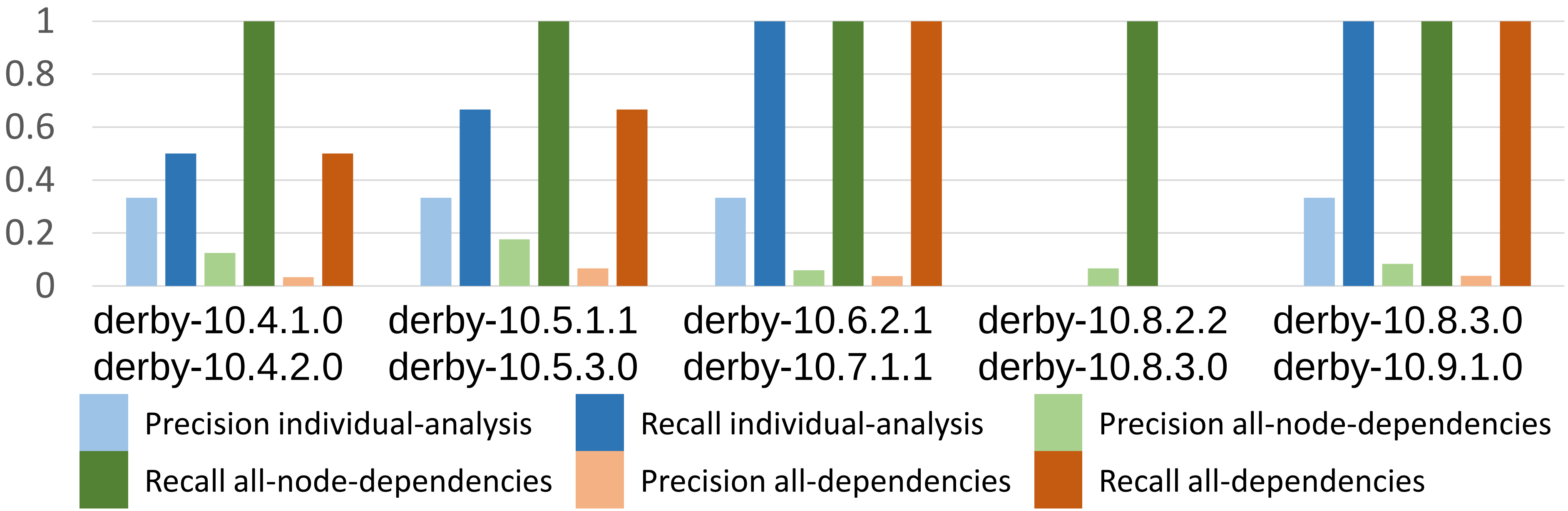



- Differences between the variants could be explained by the existence of quasi-cycles needing +1 dependency to be closed.
  - Precision of individual-analysis is not affected, but recall decreases.

# Preliminary Results

## Smell Prediction - Hubs

Apache Derby




- The performance of the variants differ.
  - individual-analysis. ↓recall (highest number of missed nodes) ↑ precision (fewest mistaken predictions)
  - all-node. ↑ recall → precision (mistaken predictions) → neighbourhood more important than overall structure
  - all-dependencies. ↓recall ↓precision

# Two and a Half Papers

- Diaz-Pace, J.A., Tommasel, A., and Godoy, D. "**Can Network Analysis Techniques help to Predict Design Dependencies? An Initial Study**". In Proceedings of the IEEE INTERNATIONAL CONFERENCE ON SOFTWARE ARCHITECTURE (ICSA 2018). Seattle USA. April, 2018. https://arxiv.org/abs/1808.02776v1

- Diaz-Pace, J.A., Tommasel, A., and Godoy, D. "**Towards Anticipation of Architectural Smells using Link Prediction Techniques**". In Proceedings of the 18th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2018). Madrid, Spain. September, 2018. http://arxiv.org/abs/1808.06362

# *Applying Social Network Analysis Techniques to Architectural Smell Prediction*

**Dr. Antonela Tommasel**

ISISTAN, CONICET-UNICEN, Argentina

antonela.tommasel@isistan.unicen.edu.ar

CONICET

I S I S T A N