AI-based Fault-proneness Metrics for Source Code Changes

F. Altiero, A. Corazza, S. Di Martino, A. Peron, and
Luigi Libero Lucio Starace

Università degli Studi di Napoli Federico II, Naples, Italy
Dept. of Electrical Engineering and Information Technology

IWSM MENSURA
September 15, 2023
Rome, Italy

luigiliberolucio.starace@unina.it

https://luistar.github.io

Software Evolution

- The lifecycle of a Software Project does not end with its initial release
- Software systems indeed typically **evolve** over time
  - To **fix bugs**
  - To **adapt** to changing environment and requirements
  - To introduce **new features**
  - To **improve design** and **performance**

Software Evolution and Regressions

- During software evolution, **changes** are made to the **codebase**
- Software evolution presents many **challenges**:
  - Keep **documentation aligned** with the code
  - Track evolving requirements
  - Refactor code to maintain adequate levels of **Software Quality**
  - **Ensure** no **regression fault** is introduced with the changes
- Not all changes are equal in terms of **fault-proneness**

Not all changes are equal

Version *i*

```
public int method(){
  int x, y;
  y = 3;
  x = y - 1;
  for(int i = 0; i < x; i++){
    x += i;
  }
  return x;
}
```

Version *i+1*

```
public int method(){
  int value, threshold;
  threshold = 3;
  value = threshold - 1;
  for(int i = 0; i < value; i++){
    value += i;
  }
  return value;
}
```

Not all changes are equal

Version *i*

```
public int method(){
  int x, y;
  y = 3;
  x = y - 1;
  for(int i = 0; i < x; i++){
    x += i;
  }
  return x;
}
```

Version *i+1*

```
public int method(){
  int x, y, i = 0;
  y = 3;
  x = y - 1;
  while(++i < x){
    x += i;
  }
  return x;
}
```

Assessing the Fault-proneness of changes

Effectively **estimating** the **fault-proneness** of codebase changes can provide several benefits:

- Allow more effective allocation of limited resources
  - **Focus testing** and **inspection** efforts towards the **most critical changes**
- Guide **fault localization** efforts

Related Works

- Many works investigated metrics to predict fault-proneness of a software system [1]
  - Detecting the most fault-prone components (classes, methods, modules)
  - Often use **historical data** to train project-specific models
- Fewer works have focused on the fault-proneness of **codebase changes**, and evaluated fault-proneness with respect to **human assessments**

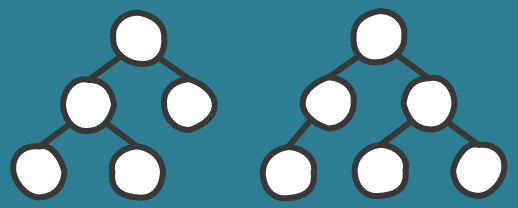
[1] A. Ouellet, M. Badri, Combining object-oriented metrics and centrality measures to predict faults in object-oriented software: An empirical validation, Journal of Software: Evolution and Process (2023)

Goals

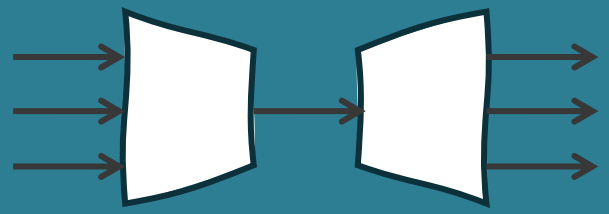
- We present a set of **AI-based metrics** for **estimating fault-proneness** of codebase **changes**, in a **project-agnostic way**
- We **assess** their **effectiveness** by comparing them with fault-proneness scores defined manually by a Software Engineer

Proposed Metrics

The Considered AI-based Metrics



Tree Kernel Functions

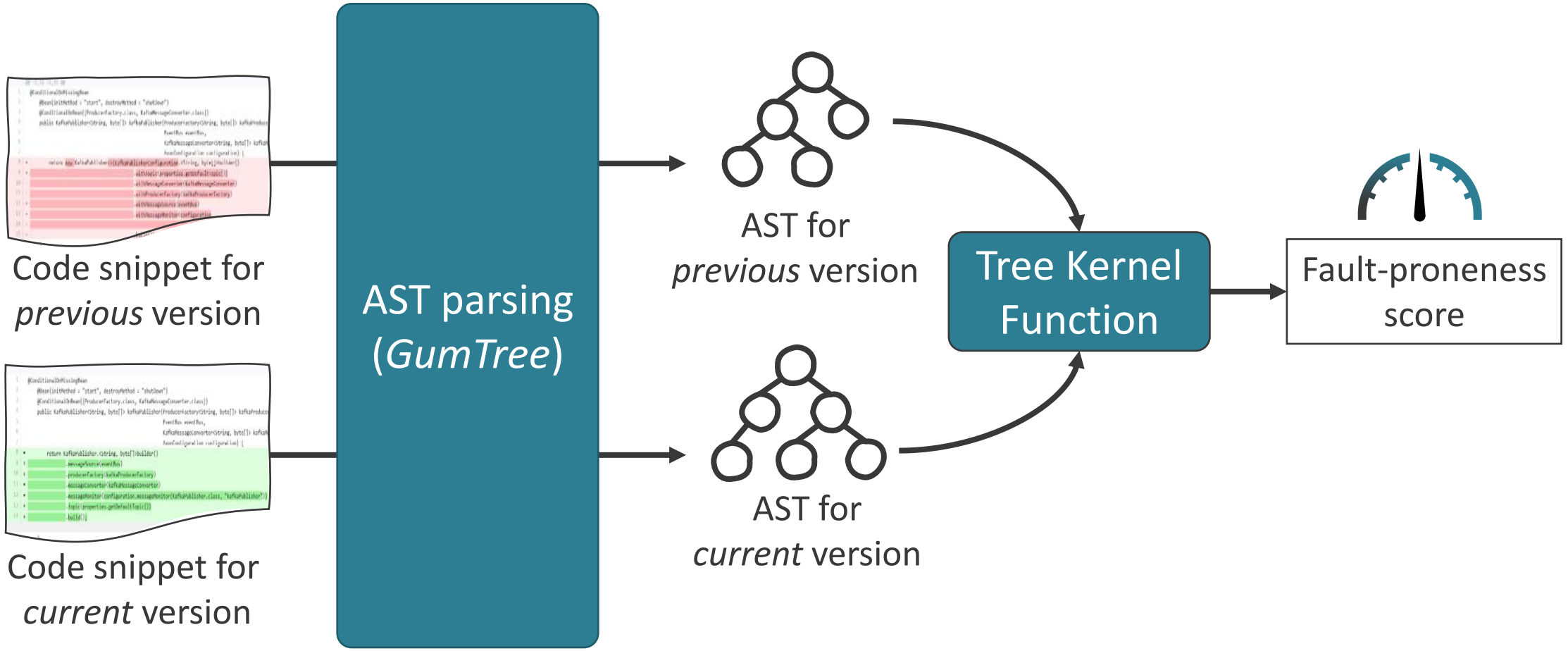


Transformer Models

Tree Kernel (TK) Functions

- Largely and effectively used in **NLP**
- **Idea**: similarity between two trees depends on the number of **fragments** *(subsets of nodes and edges)* they share
- Different definitions of «fragments» lead to different TKs
- We considered 3 TK functions from the literature:
  - Subtree Kernels
  - Subset-Tree Kernels
  - Partial Tree Kernels

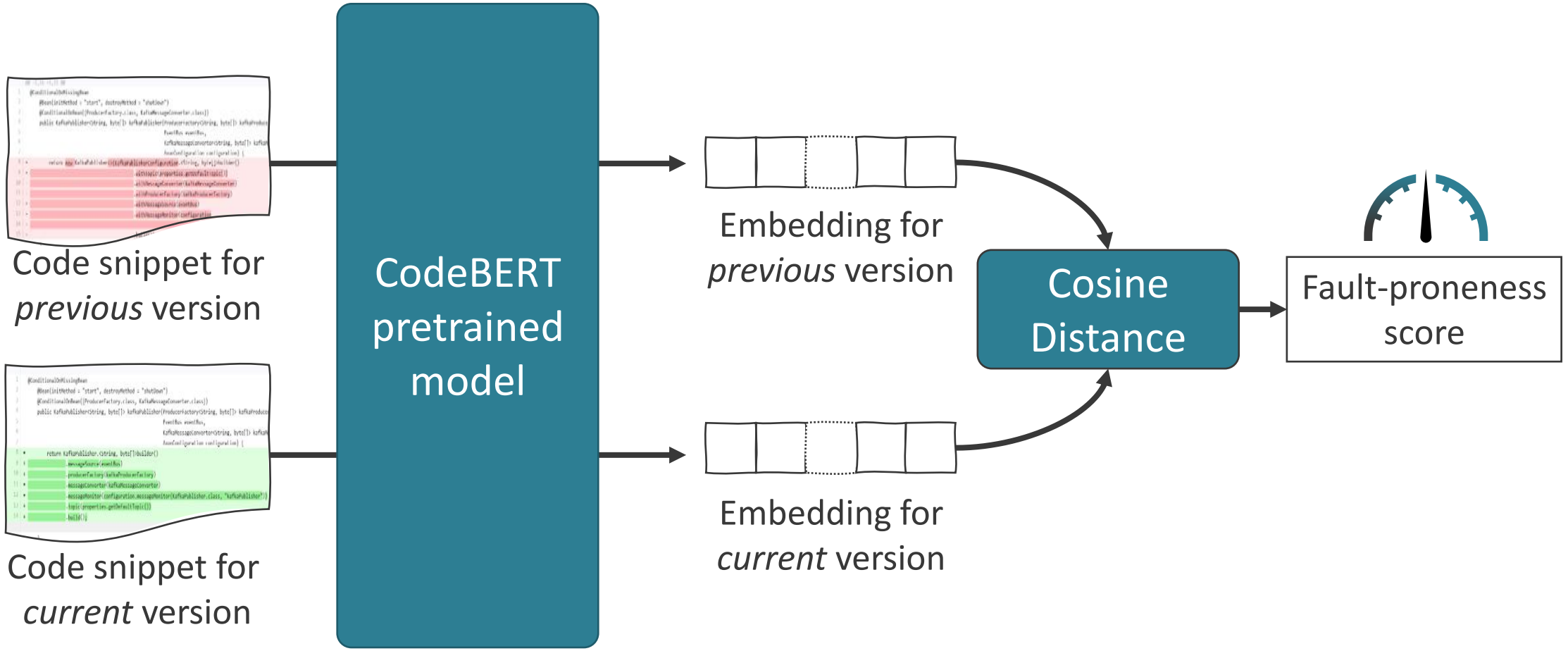
Tree Kernel-based Metrics



Transformer Models

- A class of **deep learning** models
- Trained on large corpora of data using unsupervised learning objectives (masked language modelling or next sentence prediction)
- Such models can be used to learn vector representations capturing the semantic and syntactic structure of the input
- We leverage a pre-trained **CodeBERT** model to map code snippets to vector representations in the latent vector space

Transformer-based Metric: CodeBERT-distance



Empirical Evaluation

Research Questions

- **RQ1**: To what extent do the considered metrics **correlate** with fault-proneness scores defined by a Software Engineer?

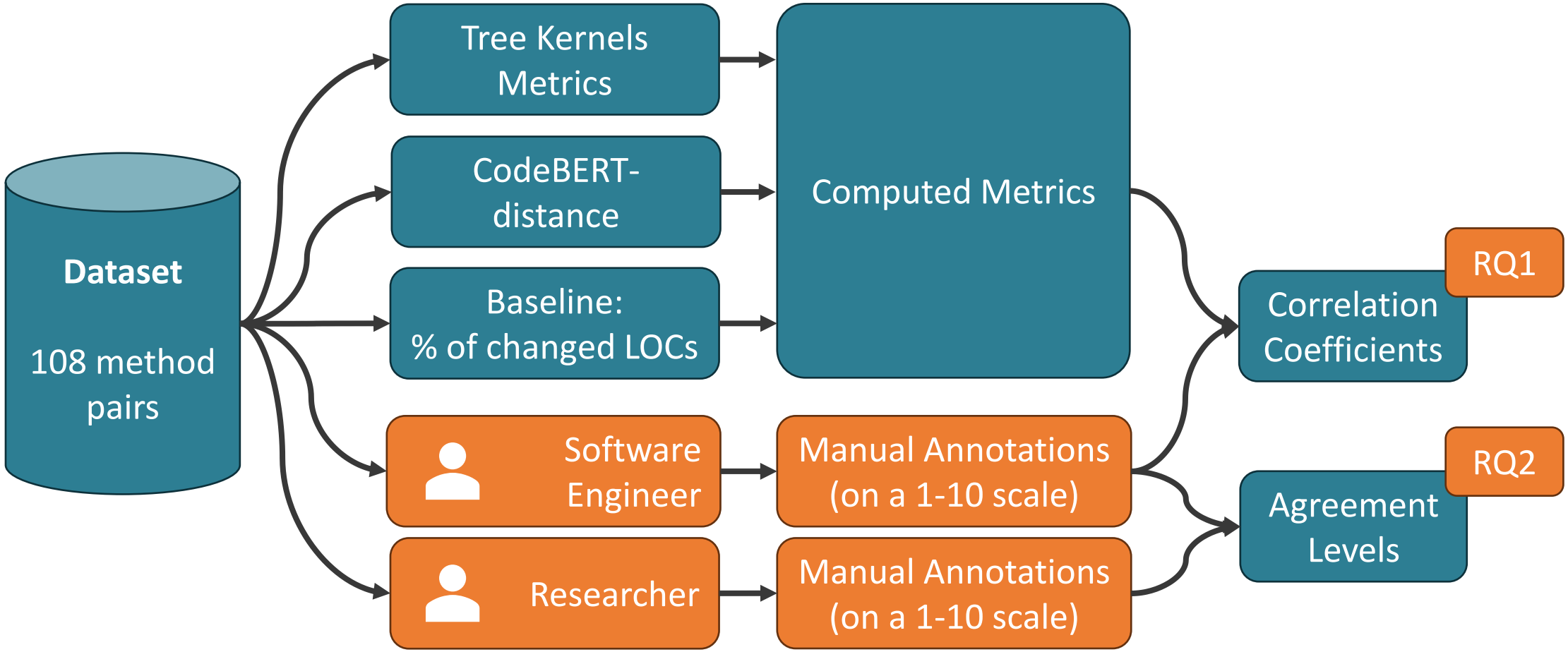
- **RQ2**: How **subjective** are manually-defined fault-proneness scores?

Dataset collection

- Started from a recent dataset for regression testing research [2]
- 104 subsequent version pairs from open source **Java projects**
- More than **1k method-level evolution scenarios**
  - Two subsequent versions of the same method **(m1, m2)**, where m2 has been affected by some changes
- Using **stratified sampling** w.r.t. projects, we sampled **108 method pairs** from **19 different projects**

[2] F. Altiero, A. Corazza, S. Di Martino, A. Peron, L. L. L. Starace, Recover: A curated dataset for regression testing research, in: Proceedings of the 19th International Conference on Mining Software Repositories, 2022.

Experimental Procedure



RQ1: Correlation with human-assigned scores

- All the considered metrics are **positively correlated** with human-defined fault-proneness scores
- SubTree Kernel and CodeBERT-distance exhibit a ***strong*** correlation
- The other metrics perform roughly as good as the baseline

Technique	Spearman's Coeff.	Grading
SubTree Kernel	0,61	Strong
CodeBERT-distance	0,52	Strong
% of changed LOCs (baseline)	0,43	Moderate

RQ2: Subjectivity of fault-proneness perception

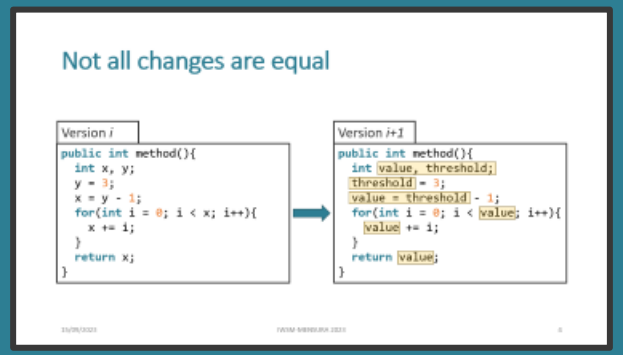
The **Software Engineer** and the **Researcher** have a **near-perfect** agreement on fault-proneness scores (**0,84 Weighted Cohen's Kappa**)

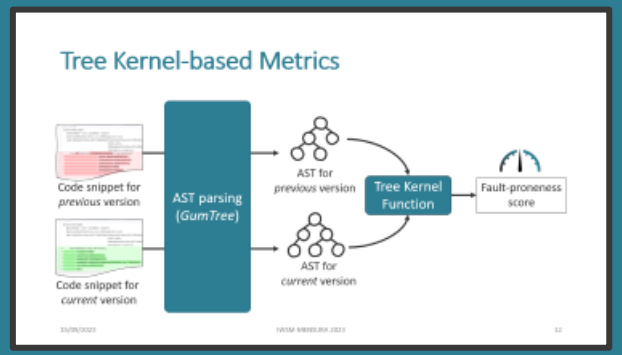
Entity of Disagreement	% of Occurrence	Cumul. % of Occurrence
0 (perfect agreement)	22	22
1	56	79
2	16	94
3	2	96
4	4	97
5	3	100

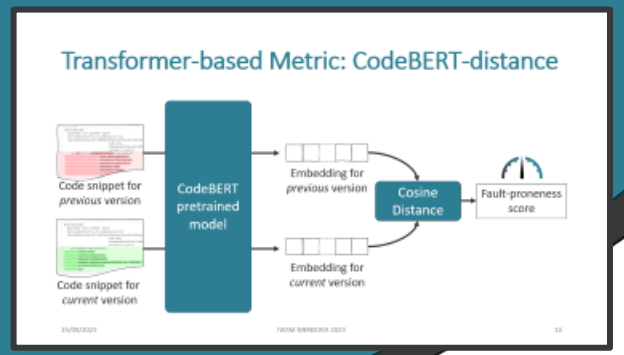
Conclusions and Future Works

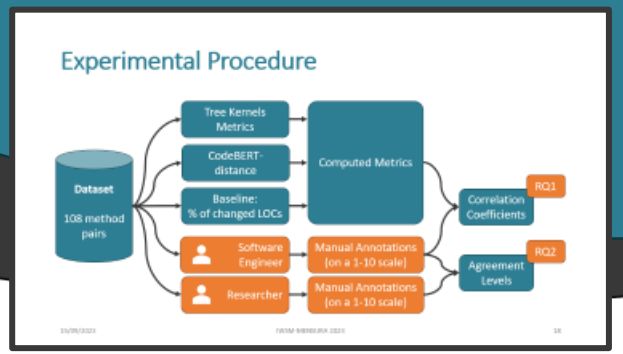
- Some of the proposed metrics are **strongly correlated** with time-consuming fault-proneness assessments performed by an expert
- In **future works**, we plan to:
  - **Further improve** the metrics, by defining ad-hoc Tree Kernels and fine-tuning the CodeBERT pre-trained model
  - Investigate correlation with the presence of **actual faults**
  - Apply the metrics in software engineering tasks such as **regression test optimization** or **fault localization**
  - Investigate the **factors** influencing human fault-proneness **perception** (i.e., **seniority**, **education**, **type of changes**, etc…)

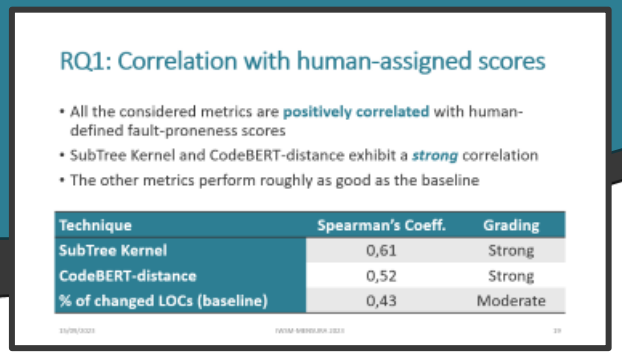
AI-based Fault-proneness Metrics for Source Code Changes

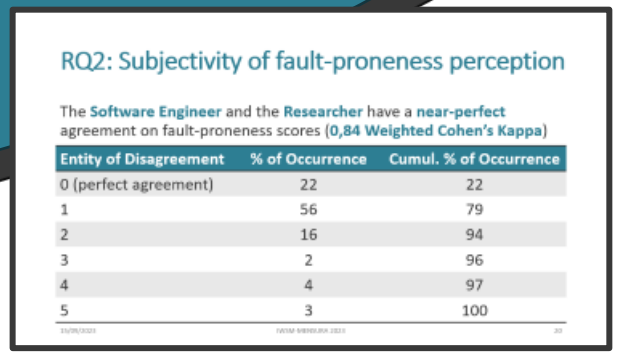












IWSM MENSURA
September 15, 2023
Rome, Italy

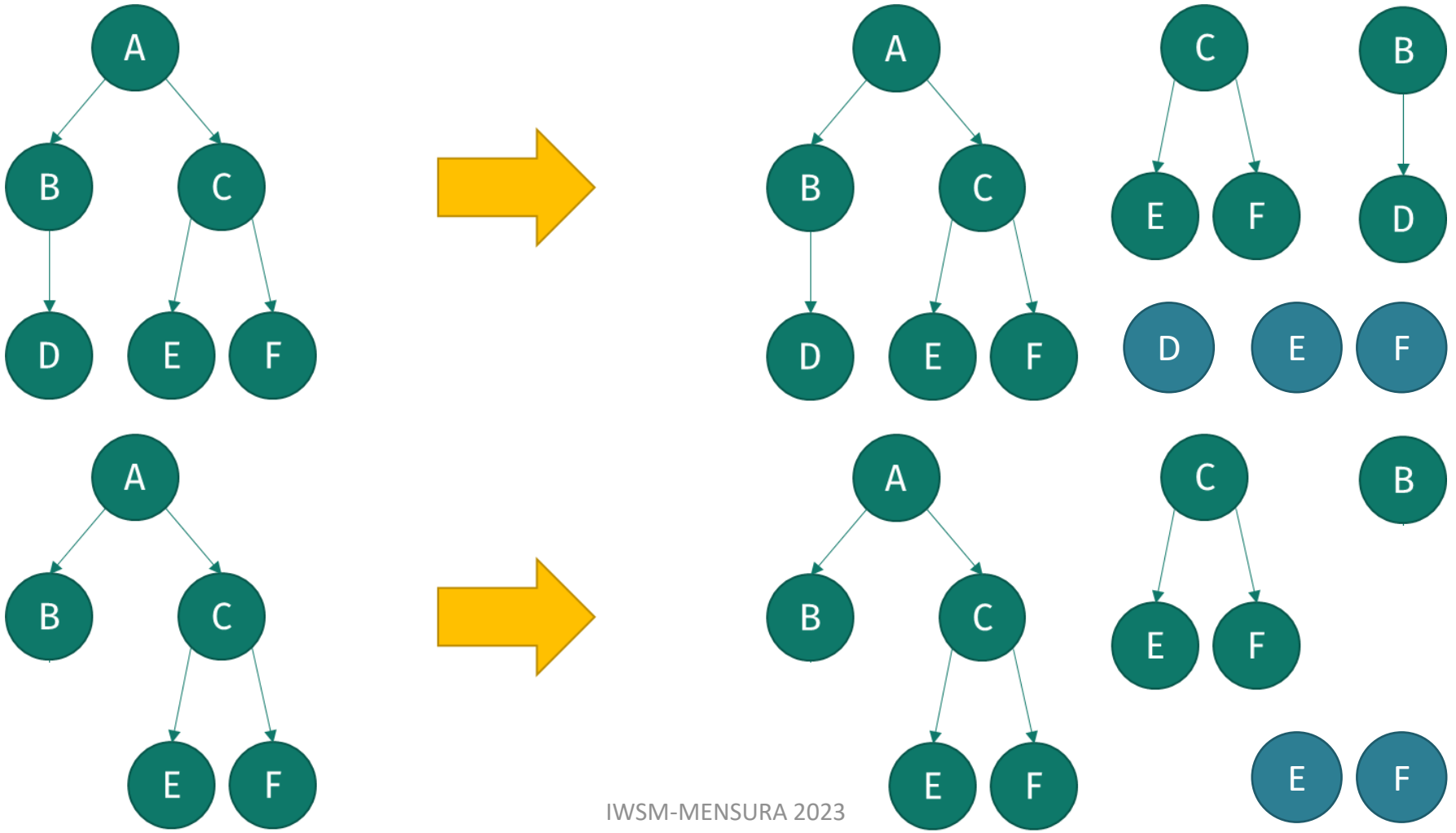
Luigi Libero Lucio Starace

luigiliberolucio.starace@unina.it

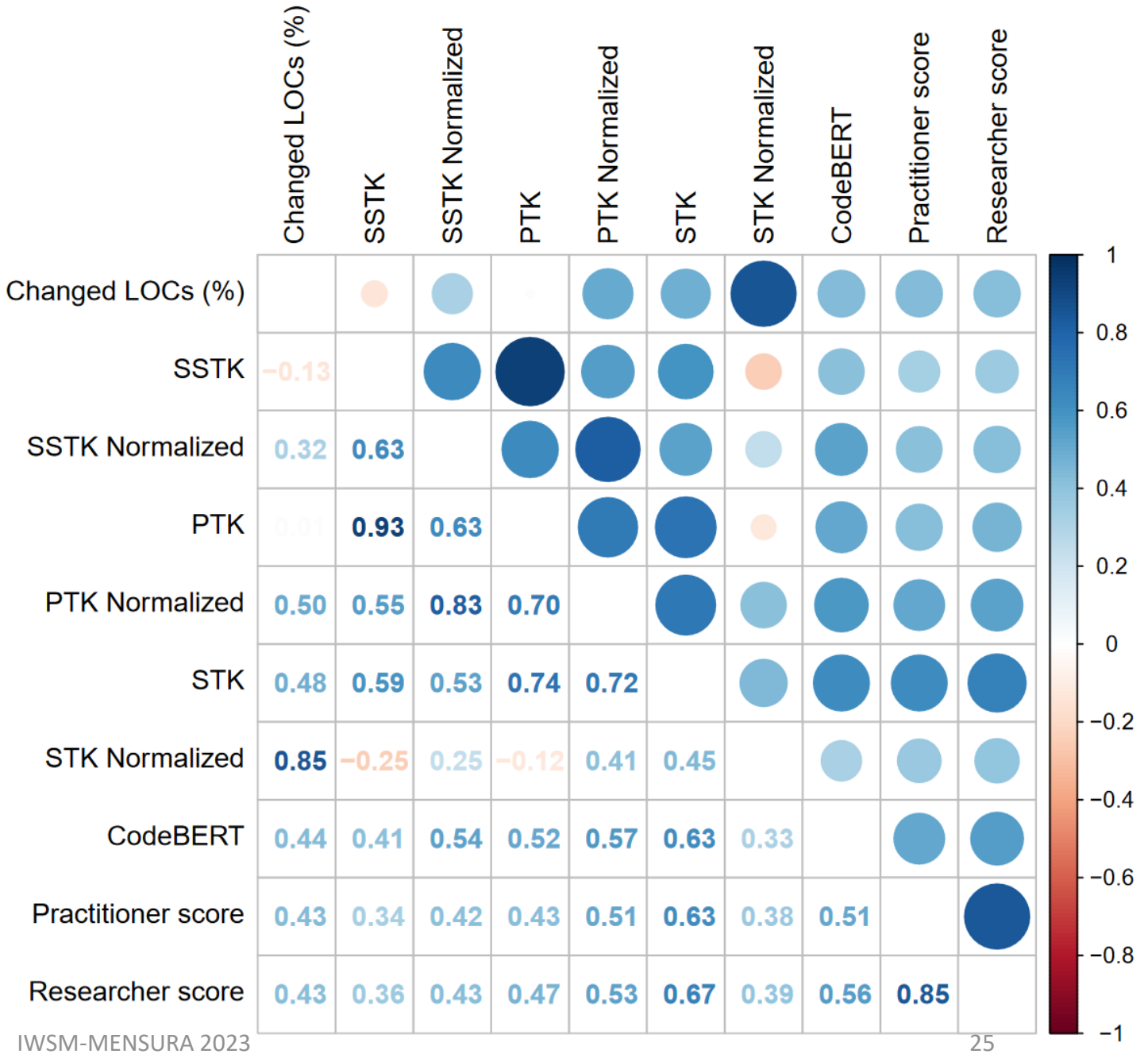
https://luistar.github.io

Backup Slides

Tree Kernels: Fragments Example



Correlation Analysis



Code Change View

Diff to HTML by rtfpessoa

Files changed (1) show

C://Users//luigi//Research//code-changes-mensura//dataset//code//3//{old.txt → new.txt} RENAMED

Viewed

```
@@ -1,37 +1,38 @@
 /**
      * Process a batch of events. The messages are processed in a new
      * the event processor creates an interceptor chain containing all
      * interceptors}.
      *
      * @param eventMessages The batch of messages that is to be proces
      * @param unitOfWork    The Unit of Work that has been prepared to
      * @param segment       The segment for which the events should be
      * @throws Exception when an exception occurred during processing
      */
     protected void processInUnitOfWork(List<? extends EventMessage<?>>
                                        UnitOfWork<? extends EventMessa
                                        Segment segment) throws Excepti
-        try {
-            unitOfWork.executeWithResult(() -> {
-                MessageMonitor.MonitorCallback monitorCallback =
-                        messageMonitor.onMessageIngested(unitOfWork.ge
-                return new DefaultInterceptorChain<>(unitOfWork, inter
-                    try {
-                        eventHandlerInvoker.handle(m, segment);
-                        monitorCallback.reportSuccess();
-                        return null;
-                    } catch (Throwable throwable) {
-                        monitorCallback.reportFailure(throwable);
-                        throw throwable;
-                    }
-                }).proceed();
-            }, rollbackConfiguration);
-        } catch (Exception e) {
+        ResultMessage<?> resultMessage = unitOfWork.executeWithResult(
+            MessageMonitor.MonitorCallback monitorCallback =
+                    messageMonitor.onMessageIngested(unitOfWork.getMes
+            return new DefaultInterceptorChain<>(unitOfWork, intercept
+                try {
+                    eventHandlerInvoker.handle(m, segment);
+                    monitorCallback.reportSuccess();
+                    return null;
+                } catch (Throwable throwable) {
+                    monitorCallback.reportFailure(throwable);
+                    throw throwable;
+                }
+            }).proceed();
+        }, rollbackConfiguration);
+
+        if (resultMessage.isExceptional()) {
+            Throwable e = resultMessage.exceptionResult();
             if (unitOfWork.isRolledBack()) {
                 errorHandler.handleError(new ErrorContext(getName(), e
             } else {
                 logger.info("Exception occurred while processing a mes
                             e.getClass().getName());
             }
         }
     }
```