Project: Bug prediction

Information Modeling & Analysis

Paolo Tonella

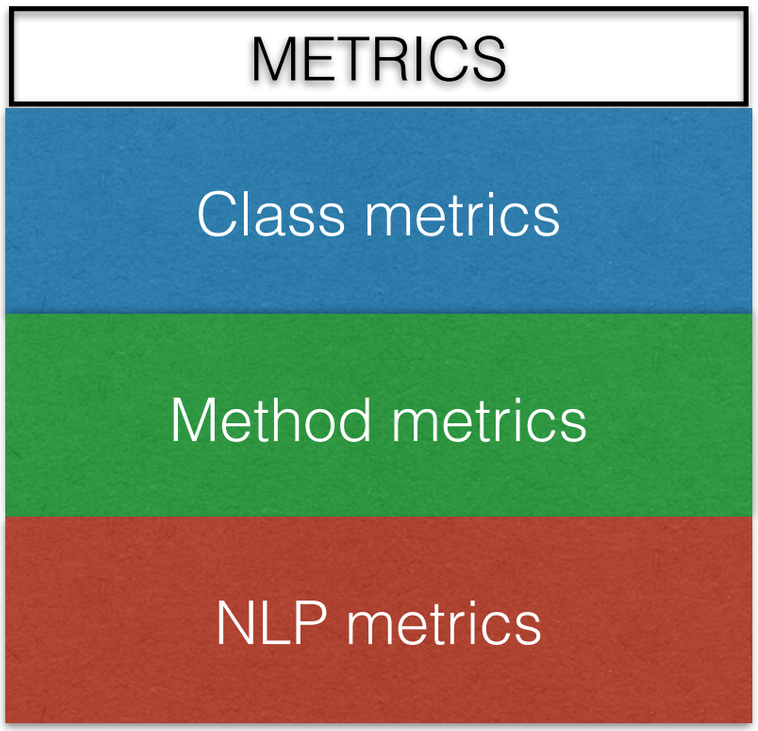
paolo.tonella@usi.ch

Goal of the project

Predict bug proneness of classes from code and NLP metrics

A ***bug prone class*** is a class that deserves intense verification and validation because its features point to a high chance that it may contain bugs:

- it contains complex/long methods
- it contains a high number of fields/methods
- it is documented poorly

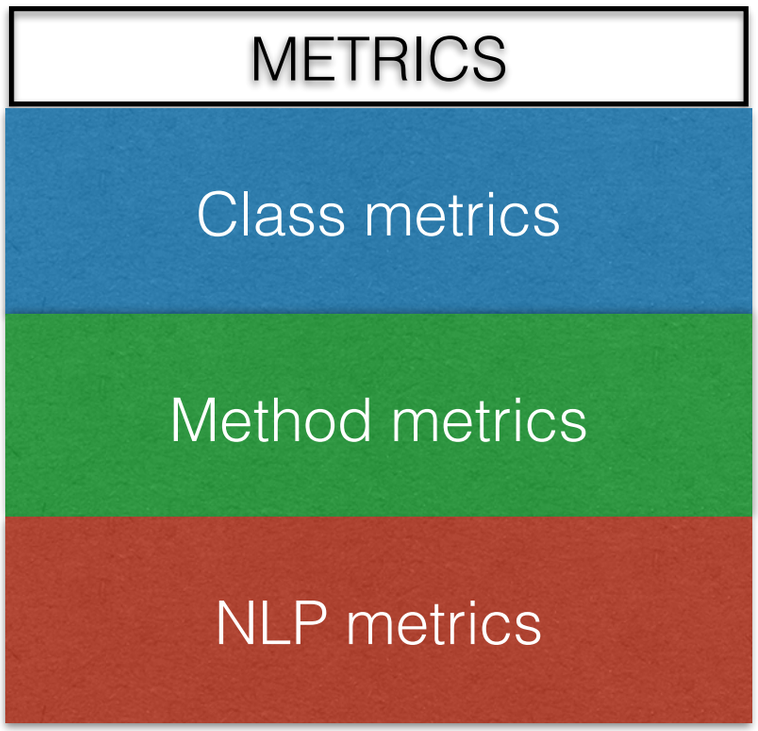


Goal of the project

Predict bug proneness of classes from code and NLP metrics

A ***bug prone class*** is a class that deserves intense verification and validation because its features point to a high chance that it may contain bugs:

- it contains complex/long methods
- it contains a high number of fields/methods
- it is documented poorly



Class metrics	
MTH	#Methods
FLD	#Fields
RFC	#Public methods + #Method invocations
INT	#Implemented interfaces

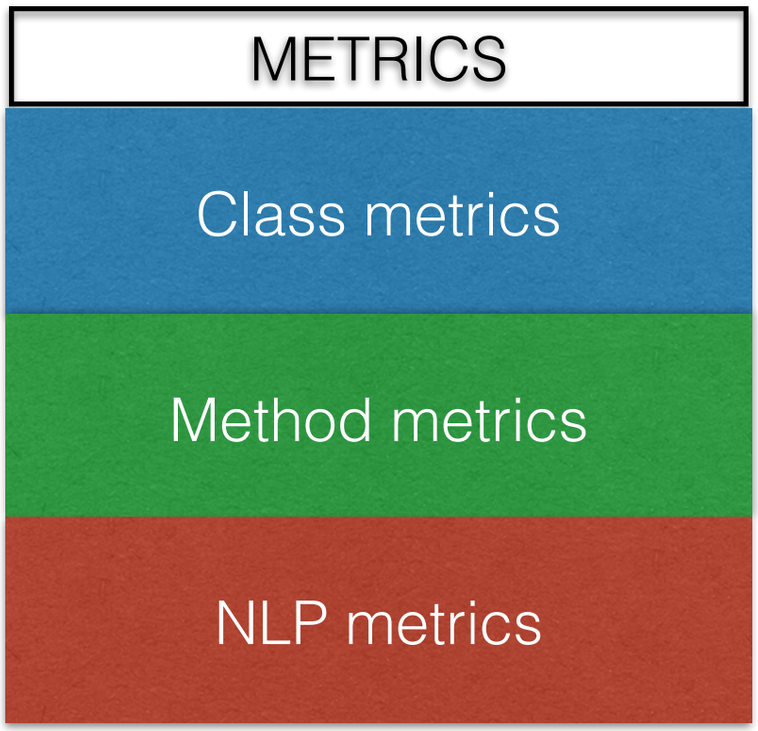
#Method invocations counts multiple invocations of the same methods multiple times

Goal of the project

Predict bug proneness of classes from code and NLP metrics

A ***bug prone class*** is a class that deserves intense verification and validation because its features point to a high chance that it may contain bugs:

- it contains complex/long methods
- it contains a high number of fields/methods
- it is documented poorly



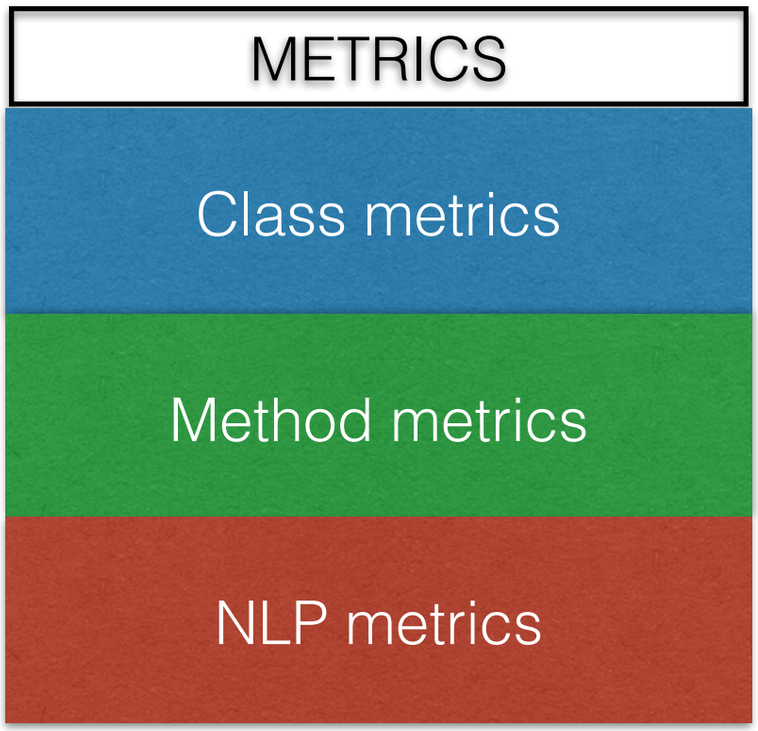
Method metrics (max over all methods)	
SZ	#Statements
CPX	#CONDITIONAL + #LOOP statements
EX	#Exceptions in *throws* clause
RET	#Return points

Goal of the project

Predict bug proneness of classes from code and NLP metrics

A ***bug prone class*** is a class that deserves intense verification and validation because its features point to a high chance that it may contain bugs:

- it contains complex/long methods
- it contains a high number of fields/methods
- it is documented poorly



NLP metrics	
BCM	#Block comments
NML	Average length of method names
WRD	#Words (longest alphanumeric substrings) in block comments
DCM	#Words in comments / #Statements

Steps of the project

1. [data pre-processing] Extract feature vectors
2. [data pre-processing] Label the data set
3. [training] Train the classifiers
4. [evaluation] Use statistical tests to compare the performance of classifiers

Github

Create your Github classroom repository at:
https://classroom.github.com/a/FbJHaDjU

Note: You must create your repository through the link provided above. Please do not create any custom repositories yourself!

Subject

Closure (defects4j): tool to parse and analyse JavaScript, remove dead code, rewrite left code and minimize it

Download the code of Closure using defects4j: https://github.com/rjust/defects4j
Defects4J is a collection of reproducible bugs

For instance:

```
defects4j checkout -p Closure -v 1f -w tmp/
defects4j checkout -p Closure -v 1b -w tmp/
defects4j info -p Closure -b 1
```

The project is focused on the analysis of version `1f`, Java files contained in:
`src/com/google/javascript/jscomp/`

Step 1: Extract feature vectors

1. Parse the source code using the Python library `javalang`
2. Compute the 12 source code metrics reported in previous slides

Step 1: Extract feature vectors

Hints:

- get all files in a directory by Python's walk: `path, dirs, files in os.walk(inputPath)`
- populate a data frame with columns `'class', 'MTH', 'FLD', 'RFC'`, etc.
- visit the AST of each class by Python's `for path, node in tree`
- consult Javalang's file `tree.py` to know the AST node types and/or print available attributes/values at each AST node by `print dir(node)` or `print node`
- recognize AST node type (e.g., class declaration) by `type(node)` (e.g., `is ClassDeclaration`), the parent type being `type(node).__base__`
- use `node.documentation` to get the block comment documenting `node`
- use `re.findall('\w+', s)` to get all alphanumeric substrings in `s`

Project report

Section 1: Data pre-processing

- describe the code written to extract feature vectors
- report/comment descriptive data on feature vectors extracted for Closure

Section 2: Classifiers

- report/comment the algorithm configurations defined after manual fine tuning (default vs specific values for optional parameters)
- comment any training error or error in the computation of precision and recall reported by the library functions

Section 3: Evaluation

- report and comment descriptive statistics and boxplots for the evaluation metrics; compare with biased classifier
- report and comment Wilcoxon's p-values
- discuss the practical usefulness of the obtained classifiers in a realistic bug prediction scenario

Project: Bug prediction

Information Modeling & Analysis

Paolo Tonella

paolo.tonella@usi.ch

Project: Bug prediction

Information Modeling & Analysis

Paolo Tonella

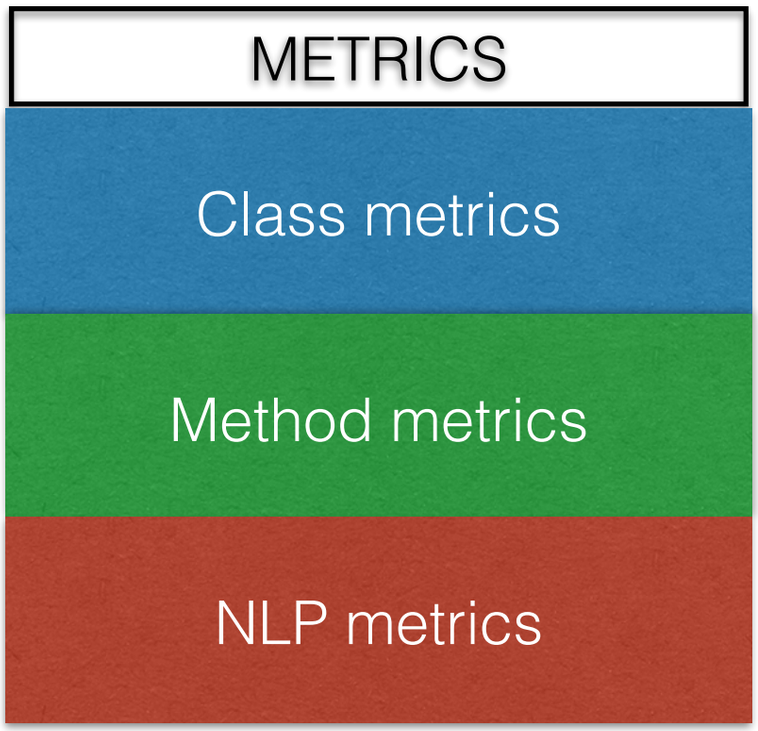
paolo.tonella@usi.ch

Goal of the project

Predict bug proneness of classes from code and NLP metrics

A ***bug prone class*** is a class that deserves intense verification and validation because its features point to a high chance that it may contain bugs:

- it contains complex/long methods
- it contains a high number of fields/methods
- it is documented poorly

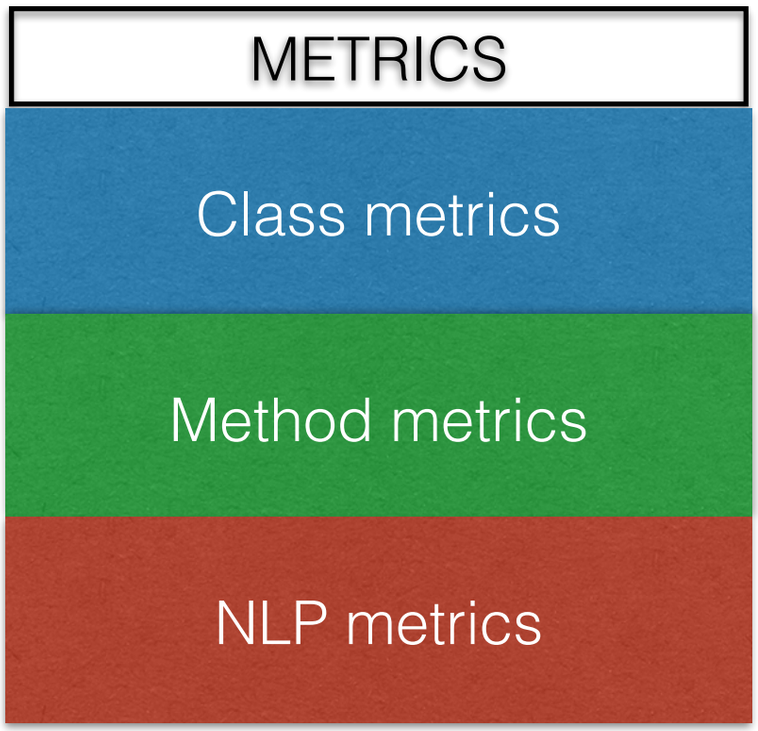


Goal of the project

Predict bug proneness of classes from code and NLP metrics

A ***bug prone class*** is a class that deserves intense verification and validation because its features point to a high chance that it may contain bugs:

- it contains complex/long methods
- it contains a high number of fields/methods
- it is documented poorly



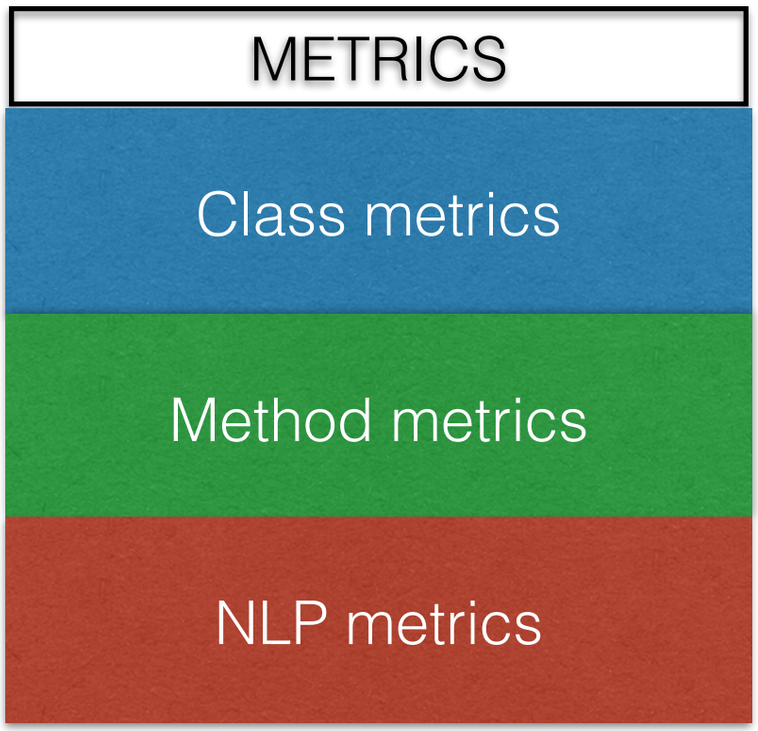
Class metrics	
MTH	#Methods
FLD	#Fields
RFC	#Public methods + #Called methods
INT	#Implemented interfaces

Goal of the project

Predict bug proneness of classes from code and NLP metrics

A ***bug prone class*** is a class that deserves intense verification and validation because its features point to a high chance that it may contain bugs:

- it contains complex/long methods
- it contains a high number of fields/methods
- it is documented poorly



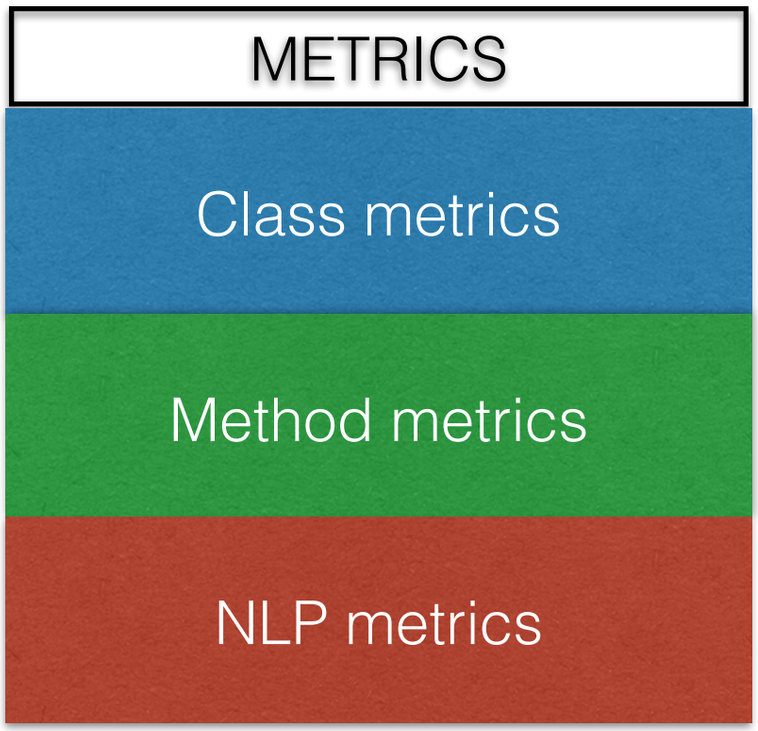
Method metrics (max over all methods)	
SZ	#Statements
CPX	#CONDITIONAL + #LOOP statements
EX	#Exceptions in *throws* clause
RET	#Return points

Goal of the project

Predict bug proneness of classes from code and NLP metrics

A ***bug prone class*** is a class that deserves intense verification and validation because its features point to a high chance that it may contain bugs:

- it contains complex/long methods
- it contains a high number of fields/methods
- it is documented poorly



NLP metrics	
BCM	#Block comments
NML	Average length of method names
WRD	#Words (longest alphanumeric substrings) in block comments
DCM	#Words in comments / #Statements

Steps of the project

1. [data pre-processing] Extract feature vectors (this task spans 2 labs)
2. [data pre-processing] Extract feature vectors (continued) and label the data set
3. [training] Train the classifiers
4. [evaluation] Use statistical tests to compare the performance of classifiers

Subject

Closure (defects4j): tool to parse and analyse JavaScript, remove dead code, rewrite left code and minimize it

Download the code of Closure using defects4j: https://github.com/rjust/defects4j
Defects4J is a collection of reproducible bugs

For instance:

```
defects4j checkout -p Closure -v 1f -w tmp/
defects4j checkout -p Closure -v 1b -w tmp/
defects4j info -p Closure -b 1
```

The project is focused on the analysis of version `1f`, Java files contained in:
`src/com/google/javascript/jscomp/`

Step 1: Extract feature vectors

1. Parse the source code using the Python library `javalang`
2. Compute the 12 source code metrics reported in previous slides

Step 1: Extract feature vectors

Hints:

- get all files in a directory by Python's walk: `path, dirs, files in os.walk(inputPath)`
- populate a data frame with columns `'class', 'MTH', 'FLD', 'RFC'`, etc.
- visit the AST of each class by Python's `for path, node in tree`
- consult Javalang's file `tree.py` to know the AST node types and/or print available attributes/values at each AST node by `print dir(node)` or `print node`
- recognize AST node type (e.g., class declaration) by `type(node)` (e.g., `is ClassDeclaration`), the parent type being `type(node).__base__`
- use `node.documentation` to get the block comment documenting `node`
- use `re.findall('\w+', s)` to get all alphanumeric substrings in `s`

Step 2: Label the feature vectors

1. Add a column `'buggy'` and set it to 1 if the row is for a buggy class
2. Create a new CSV file with the additional column

The list of buggy classes is available from the defects4j installation in:
`framework/projects/Closure/modified_classes/`

Project report

Section 1: Data pre-processing

- describe the code written to extract feature vectors
- report/comment descriptive data on feature vectors extracted for Closure

Section 2: Classifiers

- report/comment the algorithm configurations defined after manual fine tuning (default vs specific values for optional parameters)
- comment any training error or error in the computation of precision and recall reported by the library functions

Section 3: Evaluation

- report and comment descriptive statistics and boxplots for the evaluation metrics; compare with biased classifier
- report and comment Wilcoxon's p-values
- discuss the practical usefulness of the obtained classifiers in a realistic bug prediction scenario

Project: Bug prediction

Information Modeling & Analysis

Paolo Tonella

paolo.tonella@usi.ch

Project: Bug prediction

Information Modeling & Analysis

Paolo Tonella

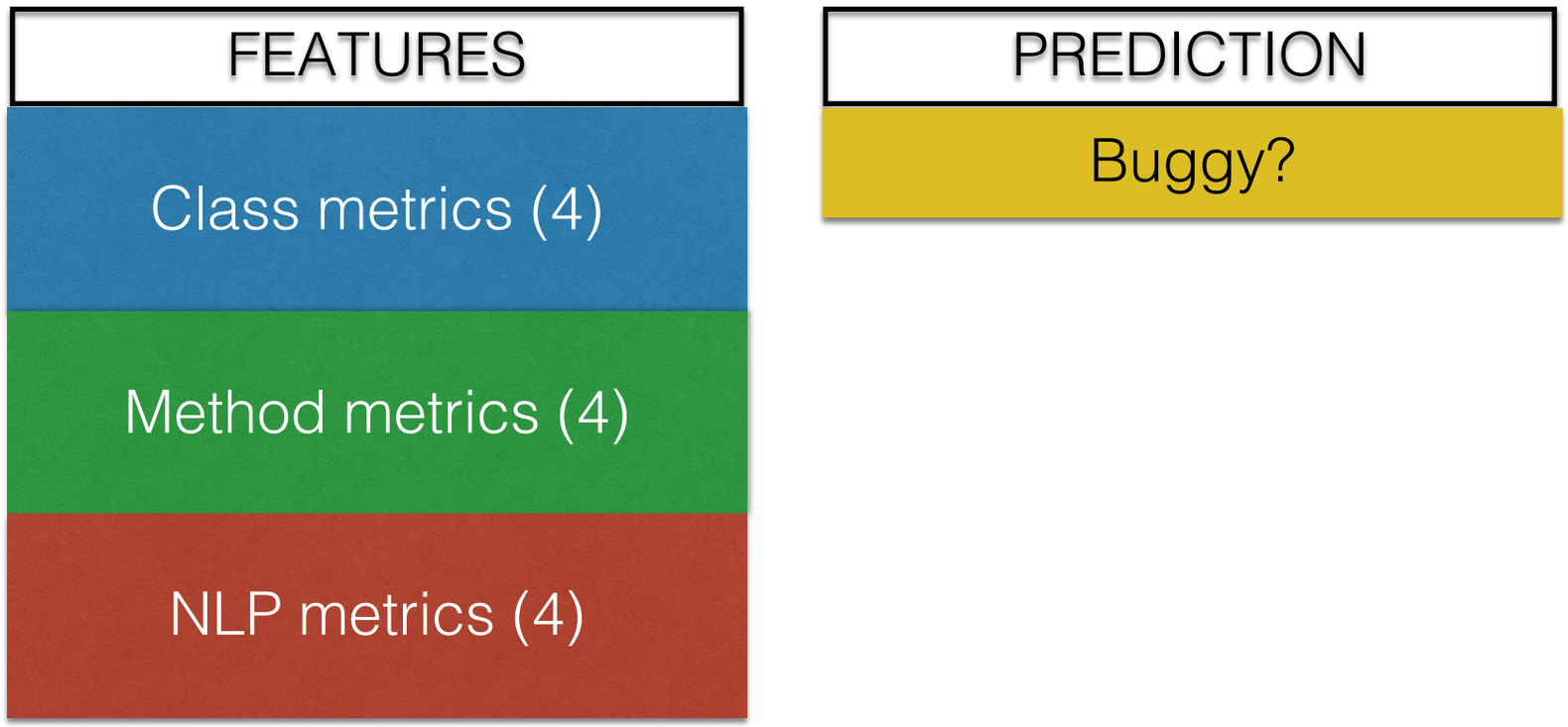
paolo.tonella@usi.ch

Goal of the project

Predict bug proneness of classes from code and NLP metrics

A ***bug prone class*** is a class that deserves intense verification and validation because its features point to a high chance that it may contain bugs:

- it contains complex/long methods
- it contains a high number of fields/methods
- it is documented poorly



Steps of the project

1. [data pre-processing] Extract feature vectors
2. [data pre-processing] Extract feature vectors and label the data set
3. [training] Train the classifiers
4. [evaluation] Use statistical tests to compare the performance of classifiers

Step 3: Hyperparameter Tuning

How to choose classifier hyperparameters (e.g., SVM kernel)?

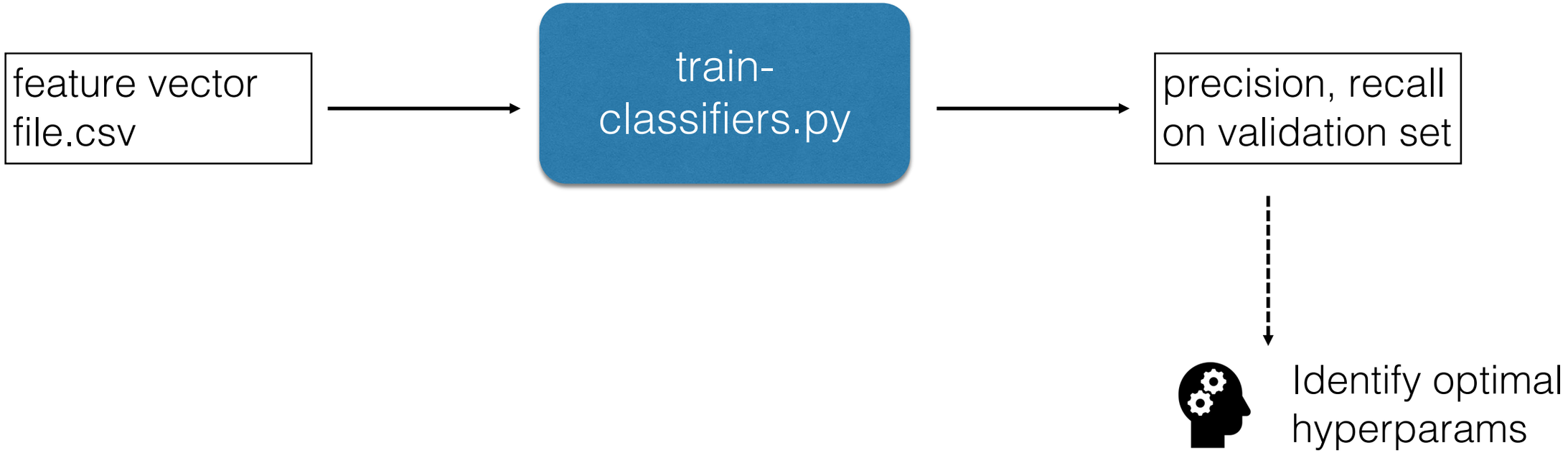
Repeat until you're happy*:

1. Select / refine hyperparameters
2. Train on training split of available data
3. Validate performance (precision, recall) on a validation set

* Various iteration approaches are possible and valid

Step 3: Train the classifiers

1. Decision Tree
2. Naive Bayes
3. Support Vector Machine
4. Multi-Layer Perceptron
5. Random Forest



Use 80% of dataset as train set and 20% as *validation* set

Step 3: Train the classifiers

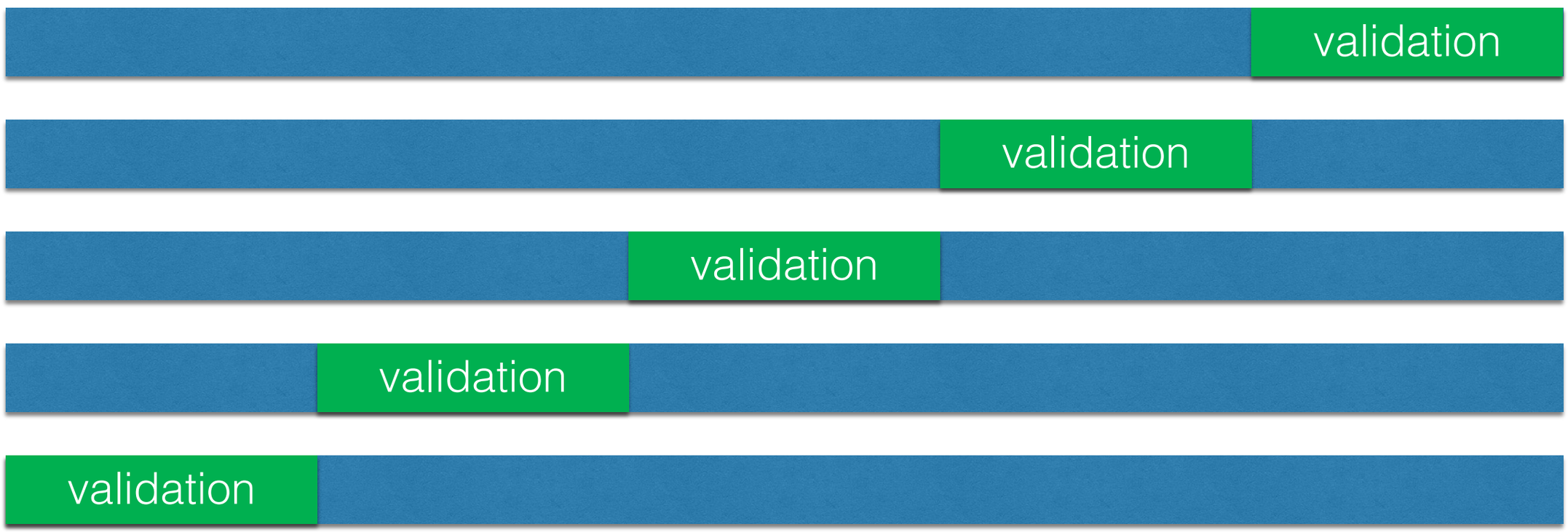
Hints:

- use library scikit-learn (`sklearn`): `DecisionTreeClassifier`, `GaussianNB`, `SVC` (linear), `MLPClassifier`, `RandomForestClassifier`
- use function `train_test_split` from `sklearn.model_selection`
- use function `precision_recall_fscore_support` from `sklearn.metrics` with `average = 'binary'`

Step 4 Preview: Evaluation

We do not have a dedicated test set! ➔ Let's test multiple times, by using parts of the available data as validation split

Example: 5-fold Cross-Validation:



Project report

Section 1: Data pre-processing

- describe the code written to extract feature vectors
- report/comment descriptive data on feature vectors extracted for Closure

Section 2: Classifiers

- report/comment the algorithm configurations defined after manual fine tuning (default vs specific values for optional parameters; size of hidden layer for MLP; etc.)
- comment any training error or error in the computation of precision and recall reported by the library functions

Section 3: Evaluation

- report and comment descriptive statistics and boxplots for the evaluation metrics; compare with biased classifier
- report and comment Wilcoxon's p-values
- discuss the practical usefulness of the obtained classifiers in a realistic bug prediction scenario

Project Report Template

IMA Project 2

FirstName LastName

May 2021

The following shows a minimal submission report for project 2. Please replace all template instructions with your own values. In addition, for any section, if **and only if** anything was unclear and you had to take assumptions about the correct implementation (e.g., about details of a metric), describe your assumptions in one or two sentences in the corresponding section.

0 Code Repository

The code and result files, part of this submission, can be found at

Repo: Add URL to your Repo
Commit: Commit ID of your submission.

1 Data Pre-Processing

I downloaded the Closure repository using " defects4J command " , and used the code in the following subfolder for the this project " Subfolder used ".

The resulting csv of extracted feature vectors can be found in the repository at the following path path .

1.1 Feature Vector Extraction

I extracted **number** feature vectors. Table Create and reference a table showing min, max and avg for every metric shows aggregate metrics about the extracted feature vectors.

Project: Bug prediction

Information Modeling & Analysis

Paolo Tonella

paolo.tonella@usi.ch

Project: Bug prediction

Information Modeling & Analysis

Paolo Tonella

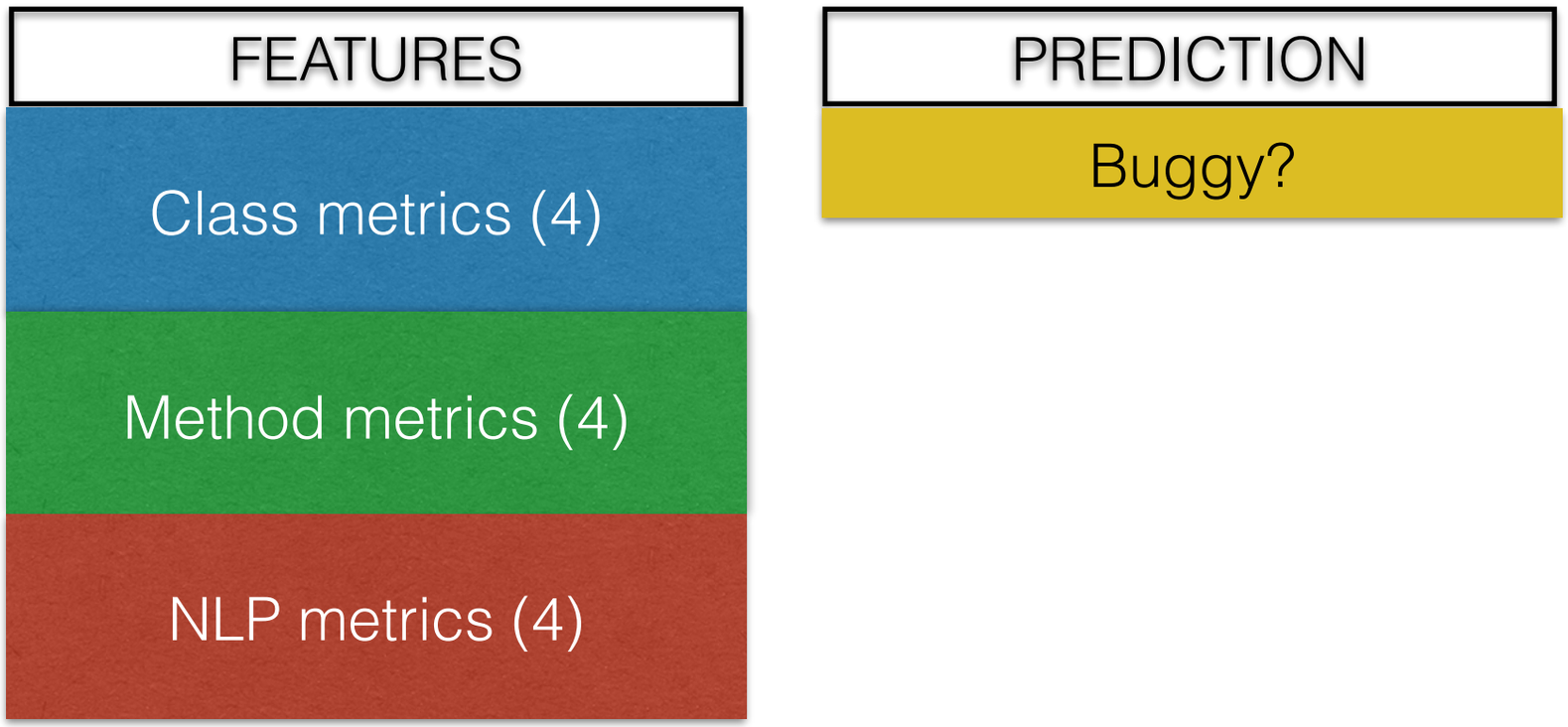
paolo.tonella@usi.ch

Goal of the project

Predict bug proneness of classes from code and NLP metrics

A ***bug prone class*** is a class that deserves intense verification and validation because its features point to a high chance that it may contain bugs:

- it contains complex/long methods
- it contains a high number of fields/methods
- it is documented poorly



Steps of the project

1. [data pre-processing] Extract feature vectors (this task spans 2 labs)
2. [data pre-processing] Extract feature vectors (continued) and label the data set
3. [training] Train the classifiers
4. [evaluation] Use statistical tests to compare the performance of classifiers

Step 4: Compare the classifiers

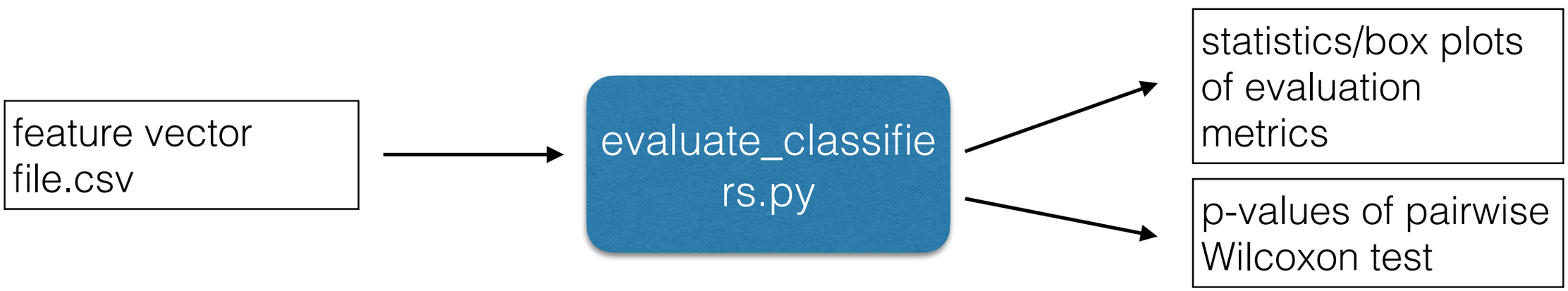
- Evaluate the classifier's performance using F-measure, as well as precision and recall
- Use 5-fold cross-validation repeated 20 times to obtain 100 evaluations (F-measures, precision/recall values) of the classifiers
- Use a non parametric statistical test (Wilcoxon) to assess the statistical significance of any observed evaluation metrics difference
- Compare the classifiers with a biased classifier that returns always 1

$$F1 = \frac{2\ \text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$$

Step 4: Compare the classifiers

Hints:

- use `cross_validate` with scoring = `{'f1': 'f1', 'precision': 'precision', 'recall': 'recall'}`
- use `make_pipeline` to use a scaler within cross validation (needed for `MLPClassifier`)
- use `wilcoxon` from `scipy.stats` to implement the Wilcoxon test
- use Numpy to compute `mean` and `std`
- use Pandas' DataFrame function `boxplot`



Project report

Section 1: Data pre-processing

- describe the code written to extract feature vectors
- report/comment descriptive data on feature vectors extracted for Closure

Section 2: Classifiers

- report/comment the algorithm configurations defined after manual fine tuning (default vs specific values for optional parameters)
- comment any training error or error in the computation of precision and recall reported by the library functions

Section 3: Evaluation

- report and comment descriptive statistics and boxplots for the evaluation metrics; compare with biased classifier
- report and comment Wilcoxon's p-values
- discuss the practical usefulness of the obtained classifiers in a realistic bug prediction scenario

Project: Bug prediction

Information Modeling & Analysis

Paolo Tonella

paolo.tonella@usi.ch