RV College of Engineering®
Mysore Road, RV Vidyaniketan Post,
Bengaluru - 560059, Karnataka, India

AI Integrated Software Engineering
UNIT-IV

By
Prof. Narasimha Swamy S
Department of AIML
R V College of Engineering®, Bengaluru-59

Go, change the world®

Outline

- → Introduction
- → Machine Learning to Support Code Reviews in Continuous Integration
  - > Code review in CI
  - > Code analysis toolchain,
  - > Code extraction,
  - > Feature extraction
  - > Model development
  - > Making a recommendations
  - > Visualization of the results
  - > Full example.
- → Using Artificial Intelligence for Auto-Generating Software for Cyber-Physical Applications
  - > Introduction
  - > Model-Based Methods
  - > Learning-Based Methods
  - > Fault Trees
  - > Model-Based Software Engineering
  - > Running Example
  - > AI-Based Framework for MBSE Task
  - > AI-based MBSE Model Construction Methods
  - > MBSE Trade-Off Framework, Empirical Modeling Cost Comparison

Introduction

Introduction

- The paradigm shift from simple Agile practices to Continuous Integration (CI) brought a different dynamics into software development.
- Five to ten years ago, software companies focused heavily on CI, initiating, and monitoring, activities aimed at getting the CI machinery working.
- Activities involved e.g. efficient use of new tools, (e.g. SonarQube, Gerrit, Kubernetes), high focus on code reviews, and change of ways-of-working to adapt to the efficient set-up and working of the CI machinery.
- Agile teams, together with the CI team, program managers, releases managers, tools team and more, all coordinated their effort to enable their organization to become a CI efficient organization.
- The last few years the focus has widened to include CDs (i.e., continuous deliveries and continuous deployments).
- At the same time, (former) new technologies as cloud, 5G, containerization, microservices and more has put a whole new attention and focus on CI, since CI is the very prerequisite for CI/CD.
- For CI, being efficient and having good quality, is far from enough. Reality today demands that the main branch holds such quality, that the main branch is always open for deliveries from the agile teams, and it is always available for deliveries to customers.

Introduction (Contd.)

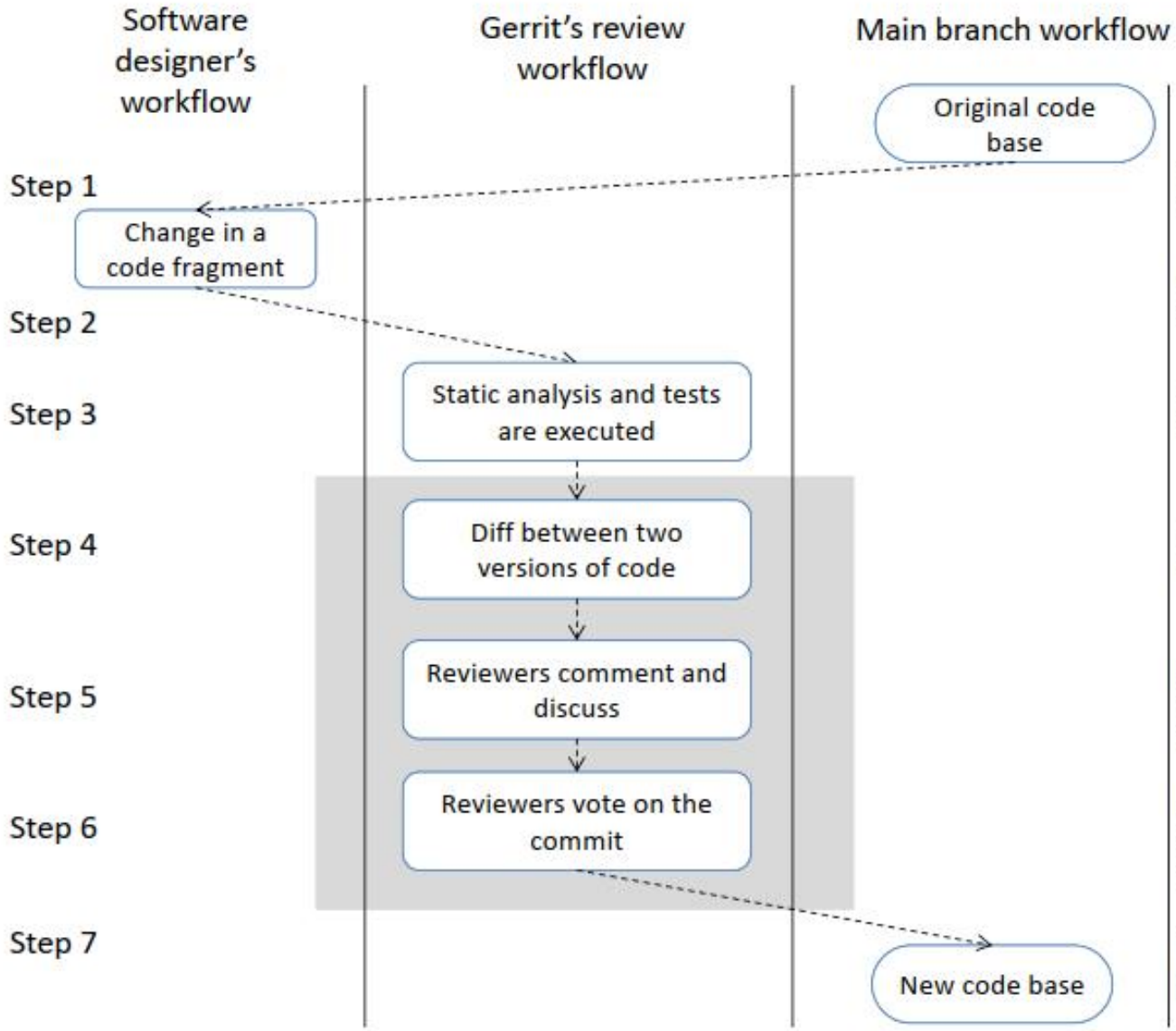
- This puts a lot of pressure on the organization using the CI flow. For instance, testing must be automated and cover all possible aspects, and the code reviews must be frequent and effort-consuming.
- Keeping high quality of the code base entails the use of several quality assurance steps, between the check-ins of the code by the designers and its integration into the main branch. These steps include static analysis of the code and manual code reviews. Despite their obvious benefits, these methods have several downsides.
- Manual reviews are time consuming, effort intensive and often reviewer-dependent. Even Linus Torvalds, the creator of Linux, identifies the need for coding review as one of the crucial activities, at the same time acknowledging that it is a time and effort consuming activity.
- Therefore, automation of this process is called for by the software engineering industry in general, and by our industrial case companies in particular.

Machine Learning to Support Code Reviews in Continuous Integration

Code Review in CI

- Modern code reviews have evolved from being a physical meeting between the reviewer and the author to a collaborative activity supported by dedicated tools
- Figure presents a typical code review workflow in a continuous integration context. The grey background in the figure shows which activities were in the scope of the automated code review support described in this chapter.
- The grey background in the figure shows which activities were in the scope of the automated code review support described in this chapter.
  - Step 1: Software designer clones the repository, thus copies the original code base to own workstation.
  - Step 2: Once done with adding new feature/bug fix, the designer commits the code to the integration.
  - Step 3: Gerrit executes project-specific static analysis checks and executes tests according to the test plan.

Code Review in CI (Contd.)



Software designer clones the repository, thus copies the original code base to own workstation.

Once done with adding new feature/bug fix, the designer commits the code to the integration.

Gerrit executes project-specific static analysis checks and executes tests according to the test plan.

Gerrit finds the difference between the original and modified code and prepares a view for the reviewers.

Reviewers read the code and comment on the problems or give positive feedback on the code. They can discuss with each other and/or with the author of the code.

Each reviewer provides a vote for the code, from -2 (low quality code, should not be integrated) to +2 (go ahead with the integration).

The code is integrated to the main branch and is not part of the original code base.

Code Review in CI (Contd.)

- Step 4: Gerrit finds the difference between the original and modified code and prepares a view for the reviewers. The standard set-up of the tools identifies all changes (added, removed and modified code) and presents that the to reviewers. The modifications vary from small (a few lines modified in a single file) to quite extensive (multiple code fragments added, removed and modified in multiple files).
- Step 5: Reviewers read the code and comment on the problems or give positive feedback on the code. They can discuss with each other and/or with the author of the code. The process of reading the code, if done properly, requires the designers to read the commit messages to understand what was done (e.g. which feature was implemented or which defect was fixed). The extensive code commits are thus effort intensive and time consuming, which, together with scarce documentation, can lead to long review durations. Pinpointing “suspicious” code fragments could reduce the effort required and thus the duration
- Step 6: Each reviewer provides a vote for the code, from −2 (low quality code, should not be integrated) to +2 (go ahead with the integration). The voting is done for the entire commit but provides the basic view on the code quality — good quality code is up-voted and low-quality code is down-voted. As the voting is mandatory for all commits, this presents the opportunity to analyze and understand what good and bad quality means in terms of code constructs.

Code Review in CI (Contd.)

- Step 7: The code is integrated to the main branch and is not part of the original code base.

Steps 3–6 are often repeated several times until the reviewers are satisfied with the changes. In practice this can mean that these steps can take over four iterations. So, even small improvements in that loop can bring significant savings. Our focus on the three greyed steps is also dictated by the fact that these activities are dependent on human reviewers and there fore can be affected by external factors. For example, the reviewer can be unavailable due to his/her commitments to other projects, or the reviewers may not fully agree on the proposed change. These factors can play a significant role when the number of code commits is large and therefore the effort required from the reviewers is high. Reducing the number of manual reviews would help to optimize the process and therefore lead to the improvement of the overall speed and quality of software development.

Code Review in CI (Contd.)

- Our industrial partners identified the following challenges which need to be addressed in this process:

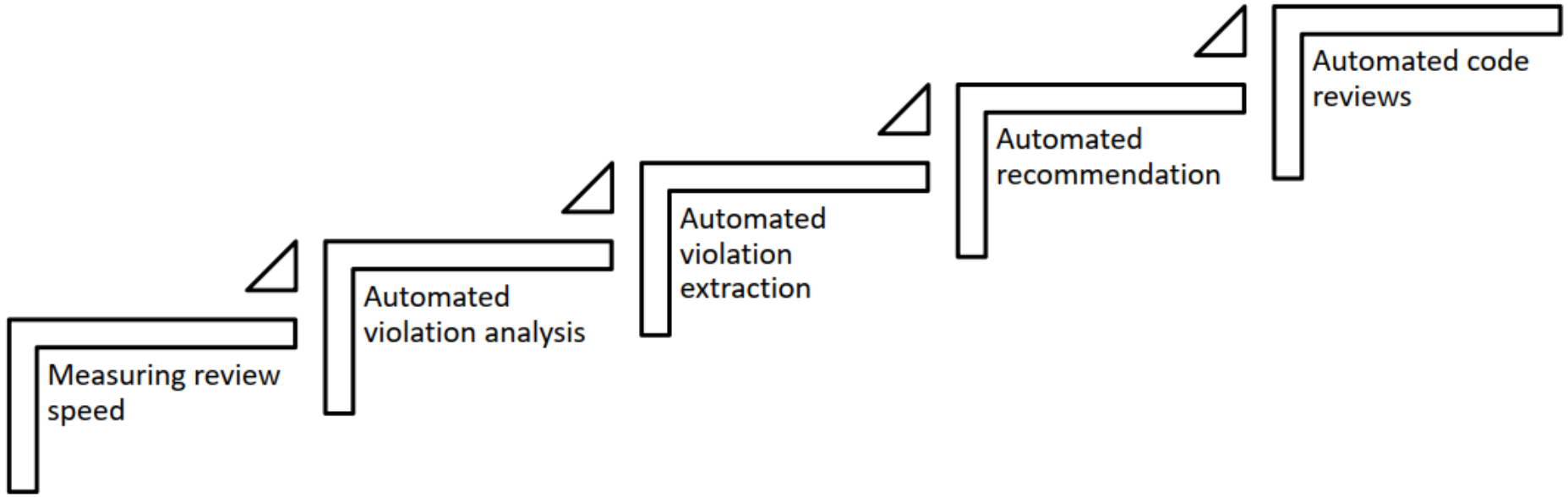
1) Human involvement from the beginning - not all commits require manual review, but involving human reviewers leads to, paradoxically, lower review quality; human reviewers often miss important code fragments in the constant inflow of the patches. Not enough focus on things that really matters, things that don't go away as soon as the compiler has done its job. Filtering out the commits that do not require manual review would have a positive effect, both on product quality and the spreading of knowledge from senior to junior developers.
2) Frustration in the iterative process — since steps 3–6 can be repeated several times, code authors and code reviewers can discuss for a long time with long breaks between each discussion comment, which slows down feature development and leads to frustration in the team.
3) Company specific coding rules are costly to maintain — the set of rules might be huge, they might change all the time. Even worse, some of the languages used in large scale, highly specialized software organizations, might be domain specific, or consist of an unholy mix of established languages that of the shelf tools can't handle.

Code Review in CI (Contd.)

4) The review process that's used for code is often used for other types of machine readable "text" as well — configuration files and so on. This is also something that's hard to manage with conventional of the shelf tools.

- However, the full automation of the review process is not desired. The process of reviewing code in CI is also a process of learning — knowledge from the experienced designers is transferred to the junior ones. The code and design decisions are discussed and therefore improved, or at least the tradeoffs are taken responsibly and after impact analyses. Therefore, we use the following metaphor of the review automation stairway in our work, Fig. 6.2.
- The first stage of is measuring review speed, where the organization automates the measurement of speed of the review process [12]. The other activities are manual:
  - Review data extraction: exporting the review comments, their metadata (e.g. timestamps) and the reviewed code fragments from the code review system to a database or a file which can be used in statistical analyses (e.g., using R).

Code Review in CI (Contd.)



Review speed measurement	Automated	Automated	Automated	Automated	Automated
Review data extraction	Manual	Automated	Automated	Automated	Automated
Rule definition	Manual	Manual	Assisted	Automated	Automated
Code analysis	Manual	Manual	Assisted	Automated	Automated
Feedback	Manual	Manual	Manual	Manual	Automated

Code Review in CI (Contd.)

- The first stage of is measuring review speed, where the organization automates the measurement of speed of the review process [12]. The other activities are manual:
  - Review data extraction: exporting the review comments, their metadata (e.g., timestamps) and the reviewed code fragments from the code review system to a database or a file which can be used in statistical analyses (e.g., using R).
  - Rule definition: defining which reviews should be considered as positive and negative, which review comments should be discarded (e.g., unambiguous) and which review comments should be considered as coding guidelines
  - Code analysis: analyzing the source code of new commits and providing the results to the code authors.
  - Feedback: providing the code reviewers with proposal for the comments for a given code fragment.
- The second stage of automated violation analysis is when the organization starts to codify their specific coding guidelines into automated tools and use these tools to analyze the code. For example, when organizations write their own static analysis rules or style checkers. The organization automatically analyze the code fragments, but the process of defining the rules and code analyses are still manual

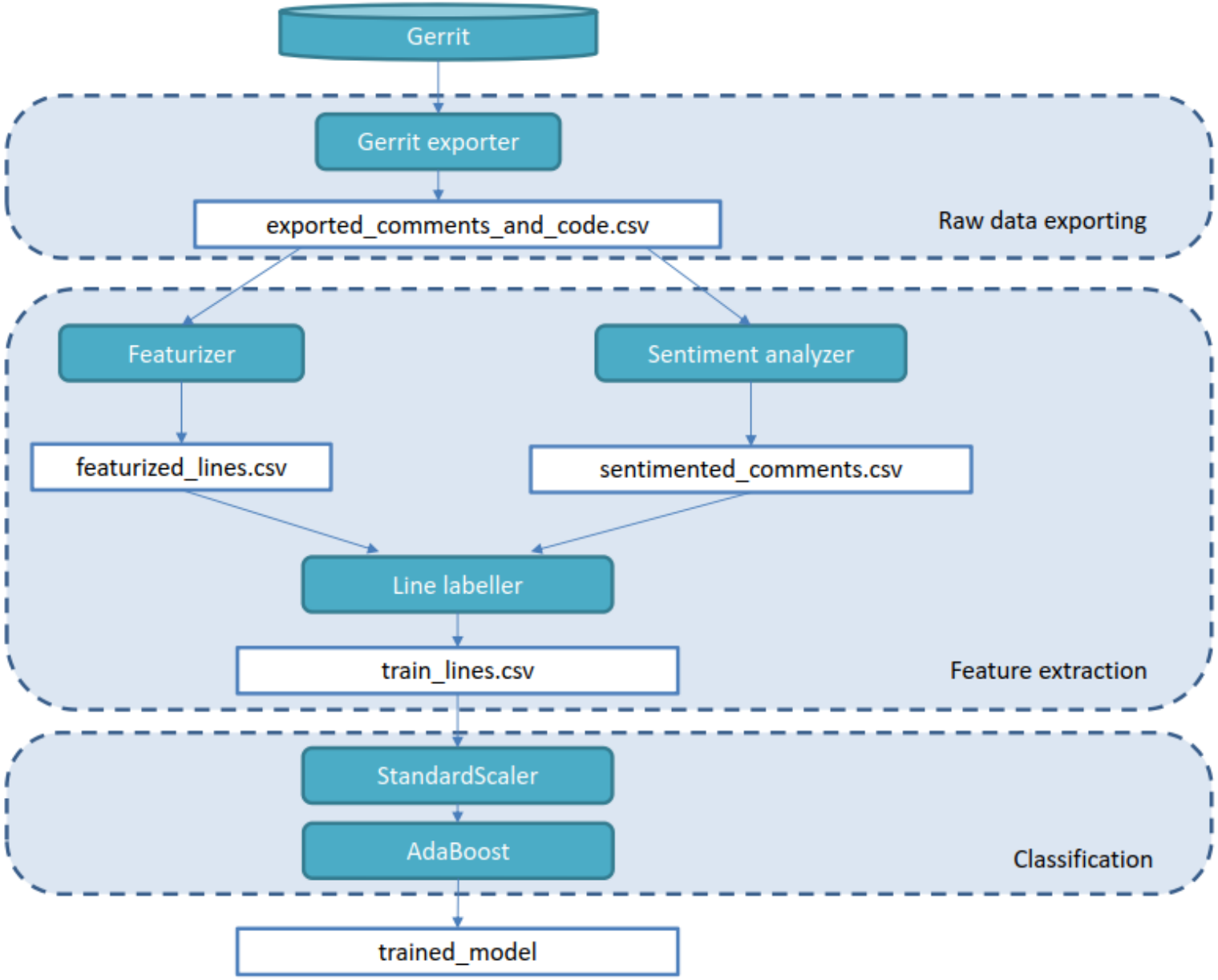
Code Review in CI (Contd.)

- In the third stage of automated violation extraction, the rule definition and code analysis is assisted. This means that the automated code review tool provides automated suggestions which review comments are repetitive and the designers can write a static analysis rule to be automatically checked. The tool can also provide examples of code fragments to which these review comments belong.
- Subsequently, in the stage of automated recommendations, the rule definition and the code analyses are automated. This means that the system can analyze the code and provide the code authors which an annotation whether a specific code fragment violates any rules or not.
- Finally, in the stage of automated code reviews, the system can also provide the insight which code review comments are most often used when commenting similar code fragments. Since this is the most interesting, and the most beneficial, approach, we focus on the automated code reviews in this paper.

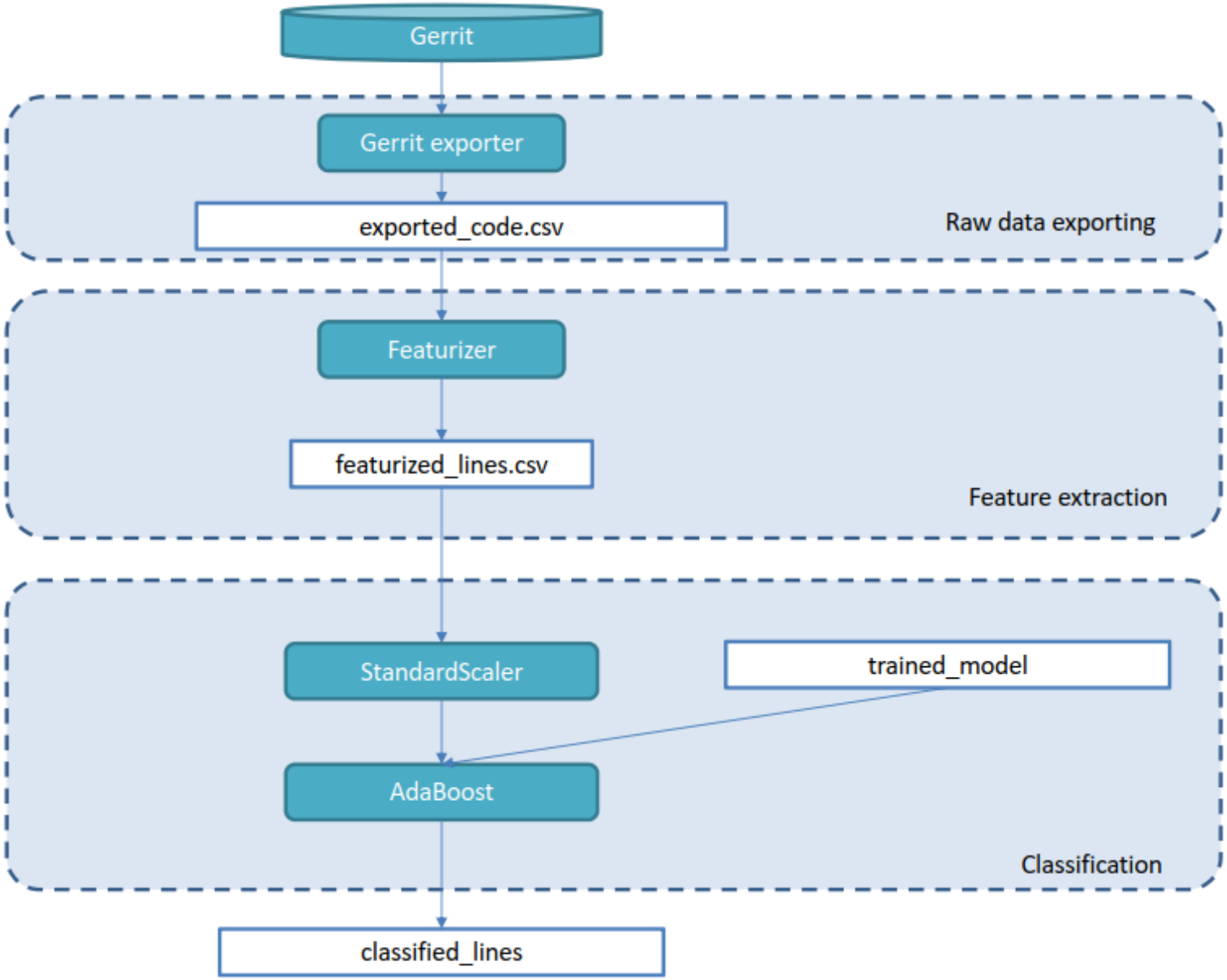
Code Analysis Toolchain

- Extracting code reviews and the code linked to these comments is a process which is organized into three parts:
  1. Exporting raw data from the source system.
  2. Extracting features from code and comments.
  3. Classification and recommendation.
- These parts are depicted in Fig. 6.3. The flow in the figure starts with the raw data export from the Gerrit review system, which is done using Python scripts and the JSON API to the system. Then, the flow continues to the feature extraction, which is based on the bag-of-words algorithm
- and finally it ends with the training of the classifier — the result is the trained ML model, which we can apply to recognize violations on new code fragments.
- Figure 6.4 presents how the trained model is applied on the new code base. The flow is similar to the analysis, except that there is no sentiment analysis (as there are not comments on the new code yet) and the classifier have the new input — the trained model.
- One observation to take from these diagrams is the change of complexity — in the training flow, the complexity is mostly around the concept of feature extraction and labelling of lines. In the recognition of the new code violation, the complexity is shifted toward the classification part the classifier needs to take the trained model as the input.

Code Analysis Toolchain (Contd.)



Code Analysis Toolchain (Contd.)



Code Analysis Toolchain (Contd.)

- Figure 6.4 presents how the trained model is applied on the new code base. The flow is like the analysis, except that there is no sentiment analysis (as there are not comments on the new code yet) and the classifier have the new input — the trained model.
- One observation to take from these diagrams is the change of complexity — in the training flow, the complexity is mostly around the concept of feature extraction and labelling of lines. In the recognition of the new code violation, the complexity is shifted toward the classification part the classifier needs to take the trained model as the input.
- In the figures we use one example of a data source — Gerrit code review system — which is one of the most popular tools. We can exchange this tool for others (e.g., GitLab, GitHub, Visual Studio Team System) leaving the other parts intact. The classifiers used in the figure — AdaBoost — are also an example and can be exchanged to neural networks or other types of classifiers.

Code Extraction

- Before we move to the description of the algorithm, let us look at an example of how a code review looks like in a typical code review tool — Fig. 6.5.
- The example shows a comment related to the code in a patch that is committed to the main branch. The figure is drawn manually to emphasize the link between the comment and the code, and to abstract away the cluttering details of a code review tool. However, it is based on how Gerrit and Git present the review comments.

```
1   double swapNumbers(double &number_to_swap, double &number_swapping)
2   {
3       double temp;
4
5       temp = number_to_swap;
6       number_to_swap = number_swapping;
7       number_swapping = temp;
8
9       return number_to_swap;
10  }
```

Do not use ”temp” as variable name, use something more informative, e.g. swapper

Good that you return the swapped value.

Code Extraction

- The figure illustrates an important design consideration — which lines are labelled as “good”, and which are labelled as “bad”. In our studies, we found that we need to export all comments and label only the lines that are commented. We experimented with exporting all lines and labelling the lines that were not commented as “good”, but this is not accurate as:
  1) Reviewers unwillingly repeat their comments, instead the write comments like this “You use too many temp variables, I will not comment on every one instance, please fix it throughout the code.”
  2) In large commits, the reviewers often focus on “sensitive” code fragments and tend to comment on them. The rest of the lines is not commented, but this does not mean they are correct or proper, it could just mean that the reviewer was pressed on time.
- The script that extracts the lines and their comments uses the API of the code review tool. In our case, we use Gerrit, as it is a tool that is both popular and has a straightforward JSON API. Code in Fig. 6.6 presents a JSON API call to get the IDs of submitted and reviewed patches.
- The code returns a JSON string which we can process as a collection in the subsequent part of the script — processing each patch and extracting the comments. The code is presented in Fig. 6.7

Code Extraction

```
# getting the handle for the changes in a Gerrit instance
# the variable 'changes' stores the JSON string with the changes
changes = rest.get("/changes/?q=status:merged&o=ALL_FILES&o=ALL_REVISIONS&o=DETAILED_LABELS",
                   headers={'Content-Type': 'application/json'})
```

Fig. 6.6: JSON API call to retrieve a batch of patch information.

Feature Extraction

- The feature extraction step uses two algorithms — bag-of-words for the source code analysis and sentiment analysis for the comments. The bag-of-words extraction of features is based on the work we used in our previous studies. For the analysis of comments, we use a lexicon-based sentiment analysis.