https://doi.org/10.1142/9789811239922_0006

Chapter 6

Machine Learning to Support Code Reviews in Continuous Integration

Miroslaw Staron[a], Miroslaw Ochodek[b], Wilhelm Meding[c],
Ola Söder[d] and Emil Rosenberg[e]
[a]*Chalmers University of Gothenburg*
[b]*Poznan University of Technology*
[c]*Ericsson AB*
[d]*Axis Communications*
[e]*Saab AB*

6.1 Introduction

The paradigm shift from simple Agile practices to Continuous Integration (CI) brought a different dynamics into software development [1, 2]. 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. This puts a lot of pressure on the organization using the CI flow. For instance, testing must be automated and cover all possible aspects [3], and the code reviews have to be frequent and effort-consuming.

Keeping high quality of the code base entails the use of a number of quality assurance steps, between the check-ins of the code by the designers and its integration into the main branch [4]. These steps include static analysis of the code (e.g. using Lint, [5]) and manual code reviews [6]. 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 [7]. Therefore, automation of this process is called for by the software engineering industry in general, and by our industrial case companies in particular.

In our previous work (Ochodek *et al.* [8]) we studied how to find arbitrary, company-specific, coding violations and using examples to train machine learning classifiers to find similar code fragments. The scenario was to find examples of good and smelly code, use these examples to train a machine learning classifier and look for more examples of the smelly code. The limitation, however, was the need to create a sufficient number of examples for each type of smell in the code and to maintain the examples as the organization evolved.

Therefore, in this chapter, we present a method that addresses this limitation. Instead of manually creating the examples and label the code fragments based on whether the example shows good code or a code smell, we use review comments from code review tools to label the code. In particular, we address the research problem of how to automatically process the large number of code patches submitted in CI flows, extract the rules for coding guidelines automatically and recommend code fragments/lines for manual reviews from the perspective of software designers. Mostly, these are code reviews done by experienced designers and reviewers in tools like Gerrit [9].

This chapter is based on our experiences from a number of research projects, conducted according to the action research methodology [10].

During the project we had the unique opportunity to work with two organizations, where we studied their code reviews and suggested changes. The chapter uses the same techniques and methods, but is based on the data from open source repositories to provide the readers with the opportunity to replicate the study and to reuse the tools and the data.

The main contribution of the chapter is the hands-on, step-by-step presentation of the method and the demonstration of its potential. It shows how the modularity of the method can be utilized to develop even more complex code analysis tools and how to use the method in reader's own context.

The remaining of the chapter is structured as follows. Firstly, Sec. 6.2 presents how code review processes work in modern CI toolchains. Section 6.3 describes the workflow for automated review analysis and recommendation. Sections 6.4, 6.5, 6.6 and 6.7 present the details of our approach. Section 6.9 presents a full example from the open source wireshark protocol implementation. Section 6.10 discusses the customization of the workflow with more classifiers, different source systems and different feature extraction techniques. Section 6.11 presents further reading in this area, for the readers who would like to get more in-depth understanding of the techniques and methods presented in the study. Finally Sec. 6.12 presents the conclusions and further work from our study.

6.2 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 [11]. Figure 6.1 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.

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.

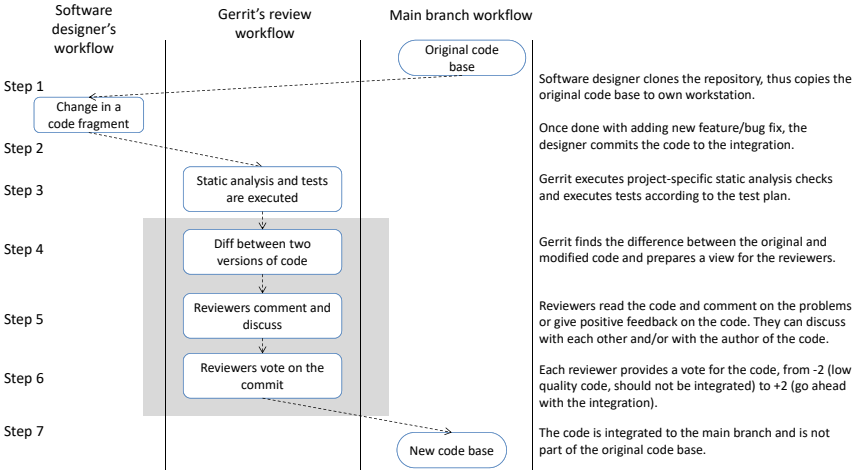


Fig. 6.1: A typical code review workflow in continuous integration projects.

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.

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

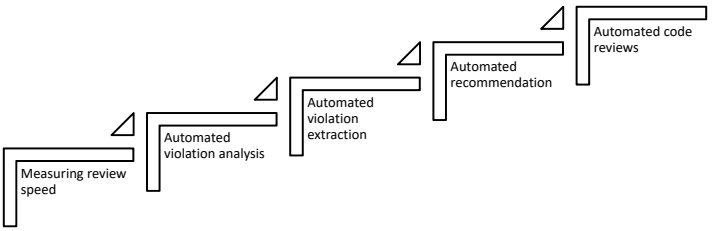
Steps 3–6 are often repeated several times until the reviewers are satisfied with the changes. In practice this can mean that these steps can take over four iterations. So, even small improvements in that loop can bring significant savings. Our focus on the three greyed steps is also dictated by the fact that these activities are dependent on human reviewers and therefore 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 [12].

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.
(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 staiway is inspired by the organizational performance stairway [13] and symbolizes how an organization can elevate its competence in code reviewing without jeopardizing the organizational learning process.



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

Fig. 6.2: Stairway in Automation of Code Review Processes in studied companies.

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 [12].

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.

6.3 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

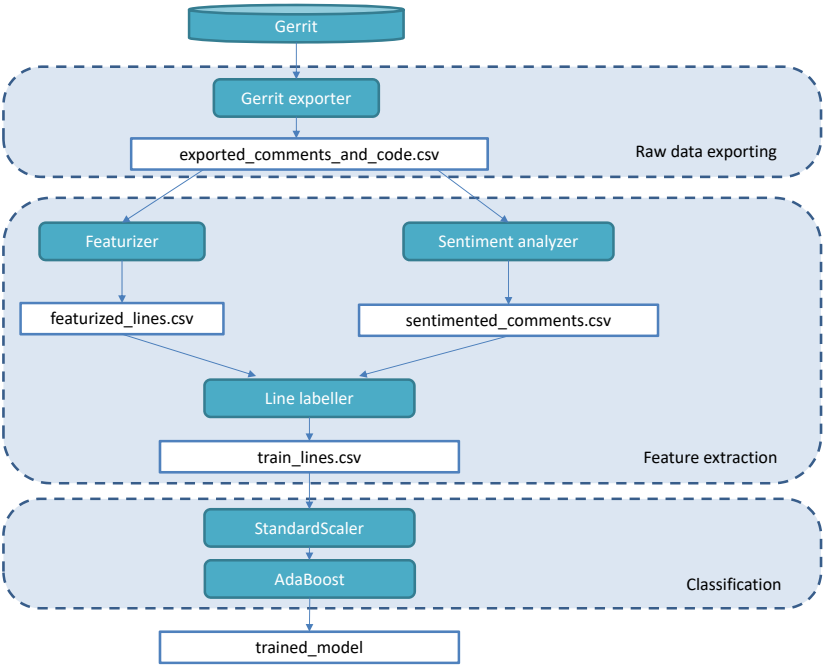


Fig. 6.3: Code analysis flow — training the classifier.

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.

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

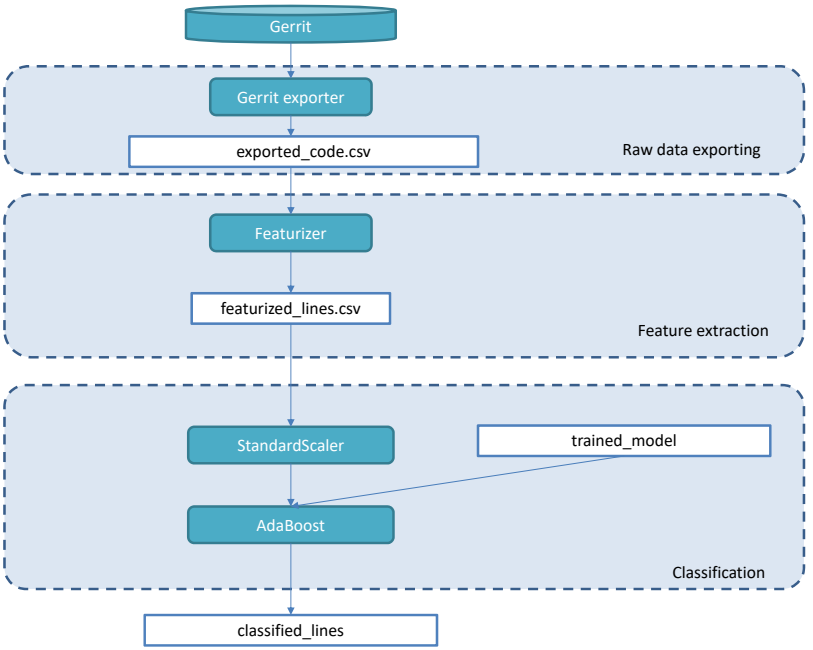


Fig. 6.4: Code analysis flow — recognizing violations on new code.

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.

6.4 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.

The figure illustrates an important design consideration — which lines are labelled as “good” and which are labelled as “bad”. In our studies, we

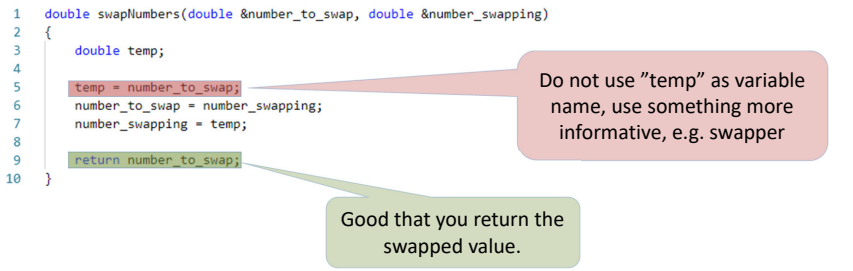


Fig. 6.5: An example of a review comment.

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.

```
# 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.

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.[1]

[1]For the sake of the simplicity of the example, we omit error handling, e.g. timeouts and missing elements of the JSON string.

```
1 # processing each code patch in the extracted JSON
2 for iIndex, change in enumerate(changes, start=1):
3     changeID = change['id']
4     revisions = change['revisions']
5 
6     for revID in list(revisions.keys()):
7         # JSON API call to get all comments for one revision in one patch
8         currentComment = rest.get("/changes/{}/revisions/{}/comments".format(changeID,revID),
9                                   headers={'Content-Type': 'application/json'})
10 
11        # not all revisions have comments, so we only look for those that have them
12        if len(currentComment) > 0:
13            for oneFile, oneComment in currentComment.items():
14                try:
15                    # this code extracts information about the comment
16                    # things like which file and which lines
17                    for oneCommentItem in oneComment:
18                        # if there is a specific line and characters as comments
19                        if 'range' in oneCommentItem:
20                            strStartLine = oneCommentItem['range']['start_line']
21                            strStartChar = oneCommentItem['range']['start_character']
22                            strEndLine = oneCommentItem['range']['end_line']
23                            strEndChar = oneCommentItem['range']['end_character']
24 
25                        # if the commented/reviewed line is not empty
26                        # then here is where we do it
27                        if strLine != '':
28                            # we need the line below to properly encode the filename as URL
29                            urlFileID = urllib.parse.quote_plus(oneFile)
30                            fileContentString = f'/changes/{changeID}/revisions/{revID}/files/{urlFileID}/content'
31                            fileContent = rest.get(fileContentString, headers={'Content-Type': 'application/json'})
32                            fileLines = fileContent.split("\n")
33 
34                            # if we have the lines delimitations (comment that is linked to lines)
35                            if strStartLine != '0':
36                                iStartLine = int(strStartLine) - 1
37                                iEndLine = int(strEndLine) - 1
38 
39                                for oneLine in fileLines[iStartLine:iEndLine]:
40                                    strToCSV = str(changeID) + ";" + \
41                                    str(revID) + ";" + \
42                                    oneFile + ";" + \
43                                    str(strLine) + ";" + \
44                                    str(strStartLine) + ";" + \
45                                    str(strEndLine) + ";" + \
46                                    fileHandle.write(strToCSV + "\n")
```

Fig. 6.7: Python code to process each patch — extract lines and comments.

Lines 8–9 call the Gerrit API and retrieve a comment for each revision in the current patch. The look starting in line 13 processes each comment and each file. Each comment has a start and end line, referenced as numbers, which we extract in lines 20–23. Each comment has a reference to the file which is commented, which we extract in line 29. We use the JSON API again to extract the content of the commented file in lines 30–31. In lines 35–37, we extract the source code lines to which the review comment belongs. In lines 39–46, we save the line and the comment to the .csv file for further processing — feature extraction.

6.5 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 [3, 8, 14]. For the analysis of comments we use a lexicon-based sentiment analysis [15].

6.5.1 *Bag-of-words analysis of source code*

The bag-of-words model (BoW) is a simplified representation of text frequently used in machine learning. It extracts features by counting occurrences of tokens or sequences of tokens (n-grams) in the passage of text (in our case, a line of code). The method requires a vocabulary that can be automatically derived from the text or explicitly provided by the user (we can also mix these two approaches).

In Fig. 6.8, we present a snippet of code in Python that can be to extract BoW features from lines of code. In lines 5–12, we define a custom tokenizer that split lines of code using characters that have special meaning in many programming languages. In contrast to the typical implementations of BoW used for natural language processing, these splitting characters are preserved and included in the vocabulary. In lines 14–18, we use the CountVectorized class available in the sklearn library to train the BoW model. The class offers multiple parameters that can be set to control how the model is created. In the provided example, we use automatic vocabulary creation and limit the size of vocabulary to 100 most frequently appearing features. We can also control the size of n-grams extracted from the text. W show how changing the ngram_range parameter affects the extracted features. Finally, in the lines 20–22 of the snippet, we show how to prepare a two-dimensional array storing the extracted features.

In our previous studies [3, 8, 14], we proposed some extensions to the standard BoW model that help applying it to code analysis. For instance, we proposed to convert tokens being out of the vocabulary to so-called token signatures. Such signatures are created by firstly replacing each uppercase letter with "A", each lowercase letter with "a", and each digit with "0" and then shrunking each subsequence of the same characters into a single character only (e.g., *aaa* to *a* or __ to _). Finally, the process is repeated for pairs and triples of characters (e.g., *AaAa* is converted to *Aa*). Using token signatures can help reducing the size of vocabulary and preventing

```
from sklearn.feature_extraction.text import CountVectorizer
import re
import pandas as pd

CODE_STOP_DELIM = "([\s\t\(\)\[\]{}!@#$%^&*\/\+\-=;:\\\\|`'\"~,.<>/?\n])"

def code_stop_words_tokenizer(line):
    global CODE_STOP_DELIM
    split_line = re.split(CODE_STOP_DELIM, line)
    split_line = list(filter(lambda a: a != '', split_line))
    split_line = ["0" if x.isdigit() else x for x in split_line]
    return split_line

count_vect = CountVectorizer(max_features=100,
                             tokenizer=code_stop_words_tokenizer,
                             ngram_range=(1,1))

bag_of_words = count_vect.fit_transform(lines).todense()

colnames = [x for x in sorted(count_vect.vocabulary_.keys())]

lines_bow = pd.DataFrame(bag_of_words, columns=colnames)

```

Fig. 6.8: Using the bag-of-words model to extract features from code.

from overfitting models to specific names of variables or methods in the training code.

6.5.2 *Sentiment analysis of review comments*

The first part of the sentiment analysis is the configuration of the algorithm. The configuration is a list of positive and negative sentiments, as shown in Fig. 6.9.

```
# lexicon for positive sentiments
keywordsComments_positive = ['good', 'idea', 'good idea',
                             'done', 'beside', 'improved',
                             'thank' 'yes', 'well',
                             'nice', 'positive', 'better',
                             'best', 'super', 'great',
                             'fantastic']

# lexicon for negative sentiments
keywordsComments_negative = ['not good', 'not improve', "don't"
                             'should', 'should not', '?',
                             'aside', 'tend', 'not done',
                             'bad', 'improve', 'remove',
                             'add', 'include', 'not include',
                             'defeat', 'no', 'do not',
                             'chaotic', 'negative', 'worse',
                             'worst']
```

Fig. 6.9: Definition of the lexicon for the analysis of code review comments.

In our case, we used a simple lexicon, which we derived by reading ca. 200 code review comments. The analysis of every review comment is presented in Fig. 6.10. Lines 10–11 and 13–14 are the main loops that calculate the number of words from the lexicon for positive and negative sentiments. Lines 16–17 calculate the quotinent, which is necessary as the number of lexicon words can differ between the positive and negative sentiments. Line 19 calculates the sentiment and lines 24–27 recalculate it to a class for the machine learning algorithm in the next step.

```
1 def comment2sentiment(strComment,
2                       keywordsComments_positive,
3                       keywordsComments_negative):
4     countPositive = 0
5     countNegative = 0
6
7     totalPositives = len(keywordsComments_positive)
8     totalNegatives = len(keywordsComments_negative)
9
10    for oneKeyword in keywordsComments_positive:
11        countPositive += strComment.lower().count(oneKeyword.lower())
12
13    for oneKeyword in keywordsComments_negative:
14        countNegative += strComment.lower().count(oneKeyword.lower())
15
16    quotinentPositive = countPositive / totalPositives
17    quotinentNegative = countNegative / totalNegatives
18
19    sentimentQuotinent = quotinentPositive - quotinentNegative
20
21    # once we have the quotinent, we change it into verdict
22    # anything that is positive becomes 1 and
23    # anything that is negative becomes 0
24    if sentimentQuotinent > 0:
25        return 1
26    else:
27        return 0
```

Fig. 6.10: Function to analyze the sentiment and turn this into a "class" that is used for the machine learning algorithms in the next step.

We chose this method for sentiment analysis due to its simplicity and our ability to control what is considered as positive and negative comment. By re-configuring the lexicon we can calibrate the method based on our needs. For example, we can calibrate the lexicon for the specific vocabulary used by the project team or by the entire company.

The result of the sentiment analysis provides us with the classification of each comment — positive (`1`) or negative (`0`). This is used in the labelling of the lines that are commented. Each line is labelled as `1` when the comment is positive and `0` when the comment is negative. In the next step, we train a machine learning classifier to recognize lines belonging to these classes.

More precisely, we train the classifier to assign a label to each line — `1` or `2`.

6.6 Model development

We use supervised learning classifiers to develop the model as we have the labelled data to train the model. In our studies, we experimented with two types of models — whitebox models based on decision trees and blackbox models based on convolutional neural networks. The innerworkings of the models are significantly different, but for the purpose of classifying lines, they are interchangeable, i.e. we can plug-in and out different models into the same workflow.

The features extracted from the source code are often many (over 1,000 features) as they need to capture the variability of the source code, but the classes are only two. This means that the classifier needs to be robust to multidimensional data and needs to be able to capture the variability. We use the AdaBoost classifier with multiple decision trees as the classifier which provides a good trade-off between the time needed for training, the data required and the resulting classification performance.

Figure 6.11 presents the code for the training of the classifier with its 10-fold cross-validation.

```
1 ab_pipeline = Pipeline([
2         ('std_scaler', StandardScaler()),
3         ('ab', AdaBoostClassifier(DecisionTreeClassifier(max_depth=20),
4                                   n_estimators=600,
5                                   algorithm="SAMME.R",
6                                   learning_rate=0.2)),
7     ])
8
9 cross_val_score(ab_pipeline, X, y, cv=10, scoring='f1')
```

Fig. 6.11: Code to train the AdaBoost classifier based on CART decision trees.

Lines 1–7 define the classifier, which is organized into a pipeline. The first step is to use the `StandardScaler`, which is a pre-processing algorithm that scales features by removing the mean and scaling to unit variance. The second step is the AdaBoost algorithm, which is based on the `DecisionTreeClassifier`. The `DecisionTreeClassifier` is a Python implementation of the CART decision tree algorithm [16]. The algorithm is well-known and robust, providing a good platform for the building the AdaBoost ensemble classifier [17].

The important parameter is the number of decision trees in the classifier, which is line 4. In this case, we have 600 trees. This is a large number and it depends on the number of features in the data set. For this code, we ended-up with over 2,000 features, which justified such a large number of trees. Based on our experiments, we recommend to set the parameter to be ca. 30% of the number of features.

Line 9 is the code that executes the training and validation process. It randomly splits the data set into train and test parts, then training the classifier 10 times (`cv` parameter). The result is the trained model, which is stored in the `ab_pipeline` variable.

The trained classifier is ready to help us to make predictions whether a specific line should be reviewed manually or not.

6.7 Making a recommendation

Making the recommendation is based on the application of the trained classifier, which is programmed using a single line of code — `y_pred_ab = ab_pipeline.predict(X_test)`. In this line, we provide the input to the classifier, `X_test`, which is a feature vector of one line. The result is the predicted class, `y_pred_ab`, which is either a `0` for the line that violates the programming practice or `1` for the line which does not.

Since our classifier was trained based on the classes obtained from the sentiment analysis, this predicted class `0` means, essentially, that "if this line was reviewed by a reviewer, the reviewer would react negatively to this line". Having this knowledge we can continue building systems on top of this, e.g. looking for similar lines in the training set, identifying their comments and providing the recommendation to the reviewers based on this identified comment.

In this workflow, however, we do not work with the semantics of the comments and therefore we should be careful with the automated recommendations. The comments can irrelevant for the context of the new line. We recommend to provide the context of the comment together with the suggested recommendation. The recommendation could be in the following form: "Previously, the reviewers identified a similar line <similar line from the training set> and provided the following comment <comment from the training set>."

6.8 Visualization of the results

In the visualization, the most important part is to show which lines of code the reviewer should focus on. This can be done in different ways — augmenting the code with comments, highlighting the code in Gerrit, highlighting the code directly in the IDE or creating a separate report.

When introducing this tool to the organization we recommend to check the way in which the team wants to set it up. In our case, we use the dashboard selection model for this purpose [**?**, 18]. The presentation of results is a kind of report, so augmenting the code in the Gerrit tool is the best visualization. However, this has the potential of cluttering the view, as this information will add more colors/annotations to the already cluttered user interface.

Therefore, a simple report with the lines recommended is often sufficient for the introductory stage. The code which is used for that is presented in Fig. 6.12.

```
def formatColor(row):
    color = 'background-color: {}'.format('lightgrey' if row.Review == 'No' else 'lightcoral')
    return (color, color, color)

styled = classifiedLines.style.apply(formatColor, axis=1)
styled.hide_index()

with open('./gerrit_review_comments_visualized_2.html', 'w') as f:
    f.write('<H1 style="text-align:center;">Results of identifying code fragments for manual review<H1/>')
    f.write('</BR>')
    f.write(styled.render())
```

Fig. 6.12: Code to visualize the results as an HTML report.

We also recommend to conduct an assessment of which kind of algorithms and types of machine learning should be adopted for the particular company [19].

6.9 Full example

In the remaining of this chapter, let us explore one example of classification and discuss the results. The code for our tool can be found at `https://github.com/miroslawstaron/auto_code_reviewer` on GitHub. In this example, we use the open source project Wireshark. The repository is available at `https://code.wireshark.org/review`.

The goal is to illustrate the output of each of the steps of the analysis and discuss them, not to obtain a perfect classification. We discuss the design decisions that can affect the performance of the classifiers as we proceed with the example.

The first step is the export. By executing the code exporting the data from the wireshark repository, we obtain the following output, as a `.csv` file. The head of the file is presented in Table 6.1.

The table contains lines of code that were commented and their corresponding comments. The first and the third comments are asking discussing the solution, whereas the second one is a confirmation that something has been done as a response to the comment.

Once we exported the raw data, we need to process it — use sentiment analysis on the comment messages and bag of words for source code lines. The application of the sentiment analysis for the comments results in the following sentiment results, presented in Table 6.2.

In order to validate the sentiment analysis, we manually labelled comments and compared that to the sentiment analysis. Figure 6.13 presents a confusion matrix and the accuracy scores for the validation. The high accuracy score (0.91) shows that even the simplistic, keyword-based sentiment analysis is rather good for our purposes.

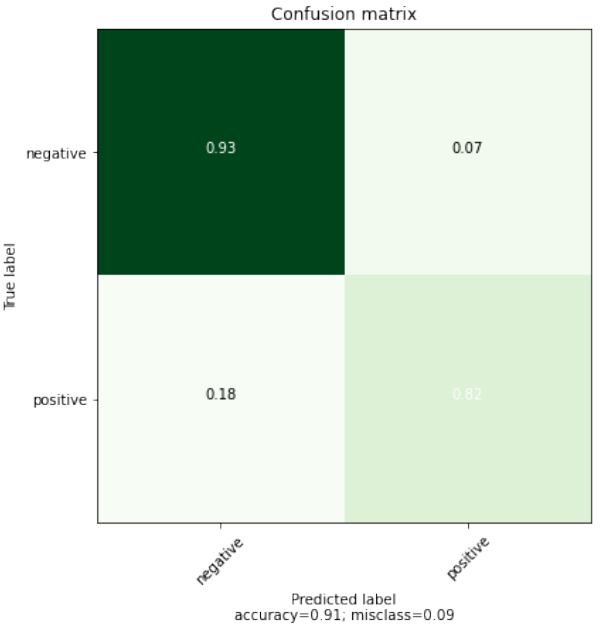


Fig. 6.13: Confusion matrix for the validation of sentiment analysis.

Table 6.1: Raw data exported from gerrit.

ch–id	rev–id	filename	line	startL	endL	LOC	message
1	r1	epan/dissectors packet-tls-utils.c	6061	0	0	tvb, offset, next_offset – offset, [truncated])	since you are not adding any extra information here, perhaps drop the [truncated] text here and rely on the label of the field in packet-tls-utils.h. And use proto_item_set_generated to get the [and] effect.
2	r2	epan/dissectors packet-tls-utils.c	6061	0	0	tvb, offset, next_offset – offset, "[truncated]")	Done
3	r3	epan/dissectors/ packet-afs.c	420	0	0	" char *version_type_"	const char * _ _ Can you also squash some trivial patches? Changing a typo in a comment can probably be done while you are modifying other code.

Table 6.2: Results of the sentiment analysis.

Message	Sentiment
since you are not adding any extra information here, perhaps drop the [truncated] text here and rely on the label of the field in packet-tls-utils.h. And use proto_item_set_generated to get the [and] effect.	0
Done	1
const char * _ _ Can you also squash some trivial patches? Changing a typo in a comment can probably be done while you are modifying other code.	1

Table 6.3: Bag of words features per line.

line	(tab)	(space)	!	“	#	$	%	...	a	able
tvb, offset, next_offset – offset, ”[truncated]”)	0	25	0	2	0	0	0	...	0	0
tvb, offset, next_offset – offset, ”[truncated]”)	0	25	0	2	0	0	0	...	0	0
char *version_type_”	1	1	0	0	0	0	0	...	0	0

Table 6.4: Raw data exported from gerrit.

line	(tab)	(space)	!	“	#	$	%	...	class_value
tvb, offset, next_offset – offset, ”[truncated]”)	0	25	0	2	0	0	0	...	0
tvb, offset, next_offset – offset, ”[truncated]”)	0	25	0	2	0	0	0	...	1
char *version_type_”	1	1	0	0	0	0	0	...	1

In parallel to the sentiment analysis, we also analyze the code and create the feature vector based on the bag of words analysis. The results are presented in Table 6.3. The table contains an excerpt of all the features identified, as the algorithm has identified 633 features.

Once we have both the sentiment for each comment and the feature vector for the line, we can merge them and create a matrix which we can use in the next step to train a machine learning model. The results are presented in Table 6.4.

Now that we have the lines labelled, we can use the AdaBoost algorithm to classify the lines. However, let us first create a diagram where we explore whether the classes are balanced, i.e. whether there is the same number of positive and negative instances. Figure 6.14 presents the results.

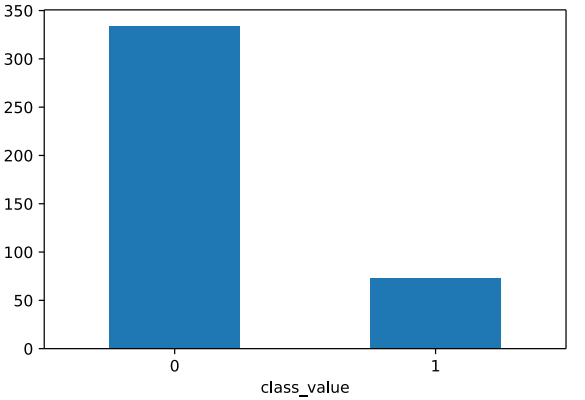


Fig. 6.14: Number of instances per class.

The diagram shows that the positive comments are fewer than the negative ones (which is quite common in code reviews). Therefore, we need to balance the classes, which is part of the toolchain described in this chapter.

Finally, we can train the machine learning classifier. The results of the trained classifier can be presented in different ways, here as a confusion matrix in Fig. 6.15.

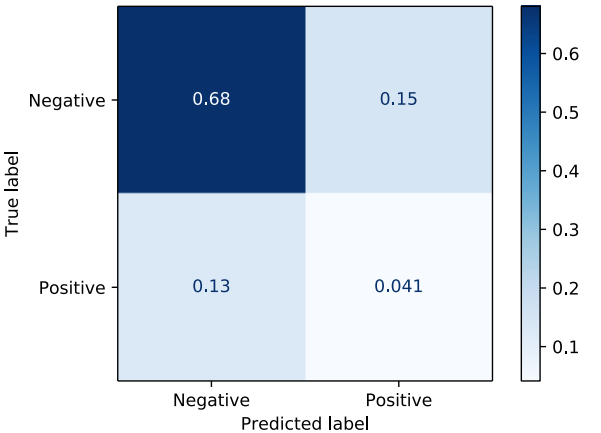


Fig. 6.15: Confusion matrix.

The results show that the classifier predicts most lines to be negative,

i.e. that they require manual reviews. This is caused by the example data set. We used a small data set for this example — ca. 400 lines — which is quite small for the classification. Since the code base of Wireshark significantly larger, over 2.8 million lines in different languages, using 400 lines to train the classifier is too little in practice, but ok for the illustration of the problem. To make it work in practice, the readers should modify the code provided in `https://github.com/miroslawstaron/auto_code_reviewer` to export all comments and then train the classifier. The results will be much better, i.e. more accurate.

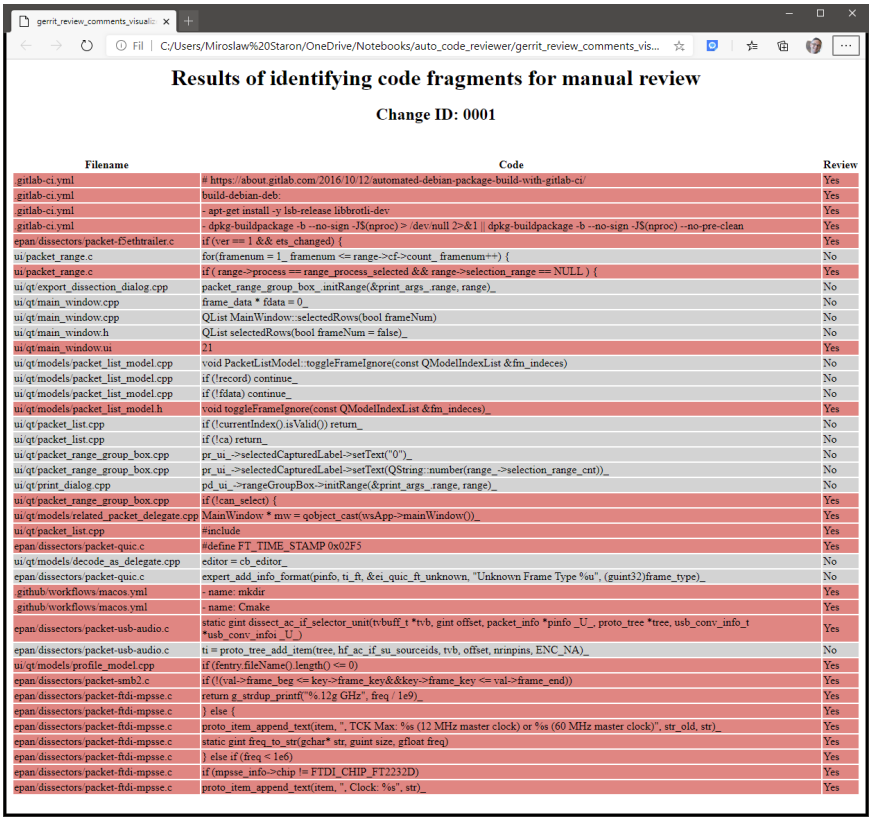
We can present the results as shown in Fig. 6.16. The figure presents an example of how this is visualized in an off-line manner, i.e. outside of the Gerrit review tool. Our future work is to create a plug-in to Gerrit to be able to highlight the lines for review in the Gerrit's user interface.

Presenting the results outside of the Gerrit's user interface is useful for the companies as this provides them with the possibility to review the entire code base, not only focus on a single patch. This helps the presented technology to get wider applicability than CI.

6.10 Using other techniques in the workflow

The workflow presented in this chapter, and depicted in Fig. 6.3, can be customized. We can exchange the source system, the feature extraction techniques and the classifiers. In our studies, we analyzed the following customizations:

(1) Exchanging the source tools with GitLab — we modified the export raw data script to use the GitLab API. The sentiment classification, bag-of-words and the classifier were not modified.
(2) Exchanging the feature extraction with own feature extraction techniques. We can exchange the bag-of-words with extractors that base on the keywords (e.g. `if`), tokens (e.g. `a0` instead of all variables that are named with small letters and numbers — `variable1` or `temp3`). We can even use more advanced techniques like word embeddings, but then we need to change the classifier to be a neural network.
(3) Exchanging the classifiers with neural networks. In the presented workflow we used the decision trees, but we can use more advanced algorithms. We can, for example, use convolutional neural networks. The advantage with deep learning techniques is that we can scale up the



Results of identifying code fragments for manual review

Change ID: 0001

Filename	Code	Review
gitlab-ci.yml	# https://about.gitlab.com/2016/10/12/automated-debian-package-build-with-gitlab-ci/	Yes
gitlab-ci.yml	build-debian-deb:	Yes
gitlab-ci.yml	- apt-get install -y lsb-release libbrotli-dev	Yes
gitlab-ci.yml	- dpkg-buildpackage -b --no-sign -J$(nproc) > /dev/null 2>&1 \|\| dpkg-buildpackage -b --no-sign -J$(nproc) --no-pre-clean	Yes
epan/dissectors/packet-f5ethtrailer.c	if (ver == 1 && ets_changed) {	Yes
ui/packet_range.c	for(framenum = 1_ framenum <= range->cf->count_ framenum++) {	No
ui/packet_range.c	if (range->process == range_process_selected && range->selection_range == NULL) {	Yes
ui/qt/export_dissection_dialog.cpp	packet_range_group_box_.initRange(&print_args_.range, range)_	No
ui/qt/main_window.cpp	frame_data * fdata = 0_	No
ui/qt/main_window.cpp	QList MainWindow::selectedRows(bool frameNum)	No
ui/qt/main_window.h	QList selectedRows(bool frameNum = false)_	No
ui/qt/main_window.ui	21	Yes
ui/qt/models/packet_list_model.cpp	void PacketListModel::toggleFrameIgnore(const QModelIndexList &fm_indeces)	No
ui/qt/models/packet_list_model.cpp	if (!record) continue_	No
ui/qt/models/packet_list_model.cpp	if (!fdata) continue_	No
ui/qt/models/packet_list_model.h	void toggleFrameIgnore(const QModelIndexList &fm_indeces)_	Yes
ui/qt/packet_list.cpp	if (!currentIndex().isValid()) return_	No
ui/qt/packet_list.cpp	if (!ca) return_	No
ui/qt/packet_range_group_box.cpp	pr_ui_->selectedCapturedLabel->setText("0")_	No
ui/qt/packet_range_group_box.cpp	pr_ui_->selectedCapturedLabel->setText(QString::number(range_->selection_range_cnt))_	No
ui/qt/print_dialog.cpp	pd_ui_->rangeGroupBox->initRange(&print_args_.range, range)_	No
ui/qt/packet_range_group_box.cpp	if (!can_select) {	Yes
ui/qt/models/related_packet_delegate.cpp	MainWindow * mw = qobject_cast(wsApp->mainWindow())_	Yes
ui/qt/packet_list.cpp	#include	Yes
epan/dissectors/packet-quic.c	#define FT_TIME_STAMP 0x02F5	Yes
ui/qt/models/decode_as_delegate.cpp	editor = cb_editor_	No
epan/dissectors/packet-quic.c	expert_add_info_format(pinfo, ti_ft, &ei_quic_ft_unknown, "Unknown Frame Type %u", (guint32)frame_type)_	No
.github/workflows/macos.yml	- name: mkdir	Yes
.github/workflows/macos.yml	- name: Cmake	Yes
epan/dissectors/packet-usb-audio.c	static gint dissect_ac_if_selector_unit(tvbuff_t *tvb, gint offset, packet_info *pinfo _U_, proto_tree *tree, usb_conv_info_t *usb_conv_info _U_)	Yes
epan/dissectors/packet-usb-audio.c	ti = proto_tree_add_item(tree, hf_ac_if_su_sourceids, tvb, offset, nrinpins, ENC_NA)_	No
ui/qt/models/profile_model.cpp	if (fentry.fileName().length() <= 0)	Yes
epan/dissectors/packet-smb2.c	if (!(val->frame_beg <= key->frame_key&&key->frame_key <= val->frame_end))	Yes
epan/dissectors/packet-ftdi-mpsse.c	return g_strdup_printf("%.12g GHz", freq / 1e9)_	Yes
epan/dissectors/packet-ftdi-mpsse.c	} else {	Yes
epan/dissectors/packet-ftdi-mpsse.c	proto_item_append_text(item, ", TCK Max: %s (12 MHz master clock) or %s (60 MHz master clock)", str_old, str)_	Yes
epan/dissectors/packet-ftdi-mpsse.c	static gint freq_to_str(gchar* str, guint size, gfloat freq)	Yes
epan/dissectors/packet-ftdi-mpsse.c	} else if (freq < 1e6)	Yes
epan/dissectors/packet-ftdi-mpsse.c	if (mpsse_info->chip != FTDI_CHIP_FT2232D)	Yes
epan/dissectors/packet-ftdi-mpsse.c	proto_item_append_text(item, ", Clock: %s", str)_	Yes

Fig. 6.16: Number of instances per class.

feature vectors to very large, we can also process much larger data sets (we experimented with products over 10 million LOC).

(4) Exchanging the sentiment analysis method. We used a simplistic sentiment analysis method, which is 91% accurate. However, it has its limitations and cannot capture sophisticated comments. To improve that, we can use natural networks for sentiment analysis and capture the details of the comments.

We recommend the companies and researchers to experiment with own techniques to get the most out of the presented work. The code base is provided in GitHub as open source under the GPL v.3 license.

6.11 Related work

The ideas of mining software data from existing repositories was populated by the seminal work of Zimmermann *et al.* [20]. The authors demonstrated how using past software design data, mined from software repositories, could help to improve the quality of software.

Liang *et al.* [21] expanded the work of Zimmermann et al. and analyzed patterns of involvement of designers in code reviews. The work opened up important alleys, but was not done in the context of continuous integration, which is the focus of our work. Bernhard *et al.* [22, 23] indicated that the Agile software development introduces changes in the way that software reviews work (compared to the non-Agile and non-CI ways of working).

Chatley and Jones [24] developed a tool for generating review comments, which is similar to what we wanted to achieve. However, the major difference is that we provide the designers with the freedom to interpret the code fragments, not generating the reviews. The important difference is that we provide the possibility for the organizations to learn, i.e. taking into consideration the human factors [25].

These human aspects are important when considering code reviews in larger industrial contexts [26]. There, learning and understanding code is important, as well as the need to discussions about the design solutions in the code. In such contexts even the softer aspects are important, e.g. code ownership [27].

Our work on data management is follows similar principles as the work of Menzies *et al.* [28]. Menzies *et al.* provided a comprehensive work on the use of big data in software engineering. We complement their work by contributing with a modern method for extracting code reviews from automated tools like Gerrit.

Another important position, which our research complement, is the work by Bird *et al.* [29], which focuses on mining software repositories. We can consider Gerrit to be one type of software repository and our methods help to mine the data, analyze it automatically, and use the new knowledge to help software engineers in their work.

In the area of code review, Pascarella *et al.* [30] studied what kind of needs the designers have in modern code reviews. The results show that the designers need to understand the usage of methods and design guidelines. Our work contributes to the automation of the review process and therefore complements this work, but opening up for automated matching between code fragments and comments.

Ebert *et al.* [31] studied the code reviews in more depth and observed a similar trend as we — that the comments often can be misinterpreted. The work of Ebert can be used to reduce the ambiguity in the sentiment analysis and therefore lead to more accurate results of the code classification, thus complementing our work.

Finally, a recent systematic mapping from Badampudi *et al.* [32] indicate that the current state-of-the-art in this area is focused on the tool support for the review process, but not in the automation of the code review, where our work fits.

6.12 Conclusions

Machine learning and artificial intelligence has risen in popularity for the past few years. These methods, which allow for automated handling of large quantities of data, seem to be a perfect match for the modern challenges of software engineering. In this chapter, we addressed one of the challenges in modern software engineering industry — achieving high quality software in the high velocity continuous integration pipelines.

By using machine learning, we designed a system that can learn from previous code reviews by abstracting both the reviewed code and the comments. The system illustrates how convenient machine learning is to solve software engineering problems. It also illustrates how to work with the data management workflows in this context.

Out further work is based on the defining a specific language model for comments in software engineering, which can provide the ability to automatically analyze the reviews with more accuracy.

References

[1] P. M. Duvall, S. Matyas and A. Glover, *Continuous integration: improving software quality and reducing risk*. Pearson Education (2007).

[2] M. Meyer, Continuous integration and its tools, *IEEE software* **31**, 3, pp. 14–16 (2014).

[3] K. Al-Sabbagh, M. Staron, R. Hebig and W. Meding, Predicting test case verdicts using textual analysis of commited code churns, (2019).

[4] A. Debbiche, M. Dienér and R. B. Svensson, Challenges when adopting continuous integration: A case study, in *International Conference on Product-Focused Software Process Improvement*. Springer, pp. 17–32 (2014).

[5] C. Sadowski, E. Aftandilian, A. Eagle, L. Miller-Cushon and C. Jaspan, Lessons from building static analysis tools at google, (2018).

https://doi.org/10.1142/9789811239922_0008

Chapter 8

Using Artificial Intelligence for Auto-Generating Software for Cyber-Physical Applications

Gregory Provan

School of Computer Science and IT, Western Gateway Building, University College Cork, Cork, Ireland T12 WT72
g.provan@cs.ucc.ie

8.1 Introduction

Model-based software engineering (MBSE) is the process of creating and exploiting domain models to generate software that can be verified to satisfy particular requirements R [1]. A model Φ is an abstract representation of the knowledge and activities that govern a particular application domain. The MBSE approach aims to increase productivity by (a) simplifying the design process through using models of recurring design patterns in the application domain; and (b) maximizing compatibility between systems through reuse of these standardized models.

Generating software that guarantees requirements $\mathcal{R}$ is an important task that traditionally has followed a manual process from requirements through to software [1]. For some cyber-physical systems (CPSs) for which models of nominal and faulty performance exist, this manual process can be augmented with the use of pre-defined system models. For example, this can be done for control systems using MATLAB/Simulink model libraries (www.mathworks.com/) or for other systems using languages like Modelica (e.g., www.modelica.org/libraries).

Traditional MBSE uses entirely human-defined artefacts, e.g., physical model components and fault trees, so it produces explainable outputs that meet particular safety requirements. However, a drawback of traditional model-based development is the need to manually generate the model

libraries and to assemble to pre-defined components into complete systems (e.g., [1]). A second drawback is that the models generated from "generic" components must have their parameters tuned to the specific application (e.g., [2]).

To circumvent these drawbacks, artificial intelligence (AI) techniques have been increasingly been proposed for generating software that guarantees we satisfy requirements $\mathcal{R}$ (e.g., [3–5]). These proposals range from using theorem provers throughout the process, replacing the entire process with machine learning [6] to applying learning to specific sub-sets of the process [7]. For example, Moitra *et al.* [8] discuss a tool that enables users to write requirements that are clear, unambiguous, conflict-free and complete. This tool creates requirements in a structured natural language that is both human- and machine-readable, and uses an automated theorem prover to formally verify the requirements and identify errors. Methods that use learning-based automation reduce costs, but may also reduce the explainability and trustworthiness of the generated systems. Automated MBSE systems don't incorporate AI/learning, although there are some recent proposals to do so [9].

At present, little focus is placed on the costs of the different phases of MBSE, and in particular the costs of model-generation. Given a new framework in which we can define different levels of automation in model-generation, it is important to explicitly examine the trade-offs associated with different model construction methods and the resulting models.

This article first reviews the most prominent proposals for applying AI to generating verified software and proposes an approach that integrates learning withing the model-based development process. We compare several different AI-based methods in terms of accuracy, and being explainable and trustworthy. We empirically show how this approach works for a CPS application. We focus on the model-generation phase, where we create a system model that guarantees properties defined with requirements $\mathcal{R}$.

Our contributions are as follows:

- We compare and contrast traditional MBSE with AI-based MBSE to highlight their strengths and weaknesses.
- We introduce a new framework for the model-development phase of MBSE, using an optimization framework to formalize both the traditional and AI approaches.
- We illustrate these differences using a CPS application of a chemical process system with fault-tolerance guarantees.

This article is organized as follows. Section 8.2 reviews the state-of-the-art in MBSE and learning-based model construction. Section 8.3 introduces the area of MBSE. Next, Sec. 8.4 introduces the phases of MBSE, and illustrates them with a well-known process-control example. Section 8.5 describes the new optimization-based framework for MBSE. Section 8.6 describes the technical details of the proposed new framework. Section 8.8 describes the different model-generation methodologies that can be used for semi- or fully-automated model generation within this new framework.

Section 8.9 presents an empirical comparison several examples of the new MBSE process. Section 8.10 summarizes our conclusions.

8.2 Related Work

Researchers have proposed two main methods for performing MBSE tasks:

(1) **Model-Based (formal methods)**: Manually build models, which can be used to formally verify if the requirements are satisfied [2, 10].
(2) **Machine Learning**: Learn representations from data that can be used to validate the requirements.

In addition, for generating software satisfying requirements on fault occurrence or fault-tolerance, **fault-tree** approaches are used to manually construct fault-trees (or equivalent representations) and, through simulation, test coverage of requirements by the generated fault-trees [11].

8.2.1 *Model-Based Methods*

Model-based methods typically use a collection of languages to model various aspects of a CPS. Examples of integrated tools for MBSE are general tools like [12, 13], or [14], which focuses on model based functional safety analysis. One integrated system, COMPASS, has been developed for critical systems such as aerospace and automotive systems [12]. COMPASS input models use the language SLIM, which is a version of the AADL language that has been extended to incorporate behavior and error specifications. The semantics and syntax of SLIM are summarized in [12]. Models are described in terms of a component hierarchy. Components interact with each other by means of ports, which send either discrete events or data values. COMPASS provides a declarative language for fault specification and then tools for safety verification.

8.2.2 *Learning-Based Methods*

Researchers have applied learning to several of the steps of MBSE, with the learning target being based on the MBSE sub-sequence addressed. Two of the most popular targets are the behaviour models and the fault-trees. In general, these purely data-driven approaches still need refinement to improve their performance with respect to manual techniques, and to improve the robustness of the learning to noise and missing data. To date, limited tools are available for design automation in AI-based systems.

Hartsell *et al.* proposes a tool-suite for all aspects of developing CPSs that have Learning-Enabled Components (LECs) [3]. These aspects include architectural modeling, engineering and integration of LECs, including support for training data collection, LEC training, LEC evaluation and verification, and system software deployment. The tool suite focuses on safety modeling and analysis.

Meijer *et al.* focuses on learning finite-state automata to represent behaviour models, using requirements specified in Linear-time Temporal Logic (LTL) as inputs [15]. The LTL formulae are checked on intermediate automata, and potential counterexamples are validated on the actual system. Spurious counterexamples are used by the learner to refine these automata.

A recent trend has focused on using machine learning, and in particular deep learning, to generate CPS systems directly from data [5, 16]. This strand of research focuses on black-box (neural network) representations for models, in contrast to white-box models like automata, e.g., [15].

8.2.3 *Fault Trees*

Fault trees are a well-established and well-understood technique used for evaluating safety/dependability of a wide range of systems: see [11] for a survey. Fault trees typically require significant manual effort both for their generation and analysis, even where software tool support exists. Recent work has focused on automating fault tree synthesis from system models. For example, fault trees can be automatically generated from a range of model languages, such as MATLAB [17] or AADL [18].

[19] has developed an approach for learning fault trees from data. This approach produces results whose performance depends on the percentage of noise in the data: for example, the fault tress perform with around 65% accuracy given up to 3% noise in the data, at a significance level of 0.01. However, the learning algorithm has exponential time complexity since it does an exhaustive search, and it also cannot deal with hidden variables.

8.3 Model-Based Software Engineering

MBSE is a software development process that uses abstraction and automation to improve software-development speed and accuracy by tackling software development complexity [2]. Abstraction is achieved by employing suitable models of (parts of) a software system. Automation systematically transforms these models into executable source code [20].

8.3.1 *MBSE Languages*

We define a model that is used in MBSE as follows:

Definition 8.1. Model [21]. A model is an abstraction of an aspect of reality (as-is or to-be) that is built for a given purpose.

Specifying models can either be done using a general-purpose modeling language (GPML) or a domain-specific language (DSL).

Definition 8.2. Modeling Language [21]. A modeling language defines a set of models that can be used for modeling purposes. Its definition consists of (a) the syntax, (b) the semantics, and (c) its pragmatics.

While modeling languages are usually not tailored to a particular domain but rather address general-purpose concepts (e.g., the UML [22]), a DSL uses model-specific representations:

Definition 8.3. Domain-Specific Language [23]. A DSL is a language that is specifically dedicated to a domain of interest, using representations that enable communication between stakeholders.

A DSL aims to bridge the gap between problem and solution space [23] and consequently is more restrictive than a general-purpose modeling language. DSLs usually drop Turing-completeness, and often allow fully automated formal verification of the (domain-specific) properties of interest. This is hardly feasible using Turing-complete general-purpose programming languages. If a system is grounded on a well-defined theory such as physics, chemistry, or biology, then researchers develop models with syntax similar to the underlying mathematical theory, e.g., using differential equations in the language Modelica (www.modelica.org).

8.3.2 *Traditional MBSE Process*

MBSE typically consists of the following tasks:

(1) **Requirements generation**: Generate a formal representation of the task requirements.
(2) **Model construction**: A DSL is used to represent the entity (e.g., CPS) and its operation within a specific environment. The system structure (the system's components and their interconnections) is often represented using an architecture description language [24], which can be used in conjunction with the DSL to auto-generate a model that can be used for simulation and analysis.
(3) **Verification & validation**: Various properties, e.g., safety, can be verified on the model or generated code.
(4) **Code generation**: Once the CPS has been modeled and verified, automated methods for generating embedded code can be applied to the model.

The traditional approach to MBSE is shown in Fig. 8.1. Here, a manual process is used to construct a model, which is iteratively improved until the model Φ and requirements R are consistent, i.e., $\Phi \wedge R \not\models \bot$, after which it is fielded in the target application. We note that the model is fixed once it is fielded.

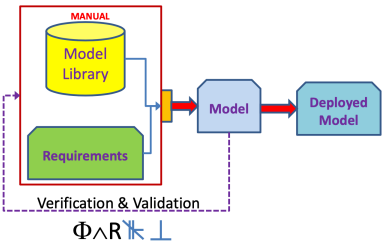


Fig. 8.1 Traditional approach to MBSE.

Definition: **MBSE Task** $[R, M, \mathcal{D}]$. Given a set R of requirements (for functionality/safety/privacy/security etc.), and component model library M, design a system model that can be guaranteed to meet the requirements R using model library M such that $\Phi \wedge R \not\models \bot$.

MBSE models a software system or parts of the system by abstract models, which are then used constructively for different aspects, e.g., functional demonstrations, safety analysis, etc. MBSE tools develop and employ code generators as well as model interpreters to reflect the models' meanings in a system. A code generator takes models as input and produces (parts

of) a software system [25]. Assuming correctness of the code generator, MBSE reduces manual development costs. However, since models typically omit certain details due to model abstractions, the generated code typically has to be manually complemented (e.g., with handwritten code). This can either be done on the generated source code level (e.g., [9]) or on the input model level (e.g., UML/P). Successfully applying constructive MBSE methods requires expertise in (a) application domain and the underlying modeling DSL; and (b) code generator or interpreter development.

8.3.3 *CPS Model Representation*

In this article, we frame our CPS as a dynamical system:

Definition: **CPS model**. We model the physical aspect of a CPS as a dynamical system as follows:

$$\dot{\boldsymbol{x}} = f(\boldsymbol{x}(t), u(t), \theta),$$
$$\boldsymbol{y} = h(\boldsymbol{x}(t), u(t), \theta),$$

where

- $\boldsymbol{x} \in \mathbb{R}^n$ is the state vector,
- $u \in \mathbb{R}^m$ are known process inputs (manipulated variables or measured disturbances taking arbitrary values independently of the rest of the variables),
- $\boldsymbol{y}$ is the system observation,
- $\theta \in \mathbb{R}^\rho$ are model parameters (assumed to be constant) and
- $f(\cdot) \in \mathbb{R}^n,\ h(\cdot) \in \mathbb{R}^l$ are nonlinear functions of their arguments.

A CPS also has a cyber element, which in this case is the set of output measurements y and the control settings Ω for the input flows and the valves.

We frame the measured data for our CPS as follows:

Definition: **Measured Data**. We assume that we have measured data $\mathcal{D}$ represented as a collection of pairs (ξ, c), where $\boldsymbol{\xi} \in \Xi$ consist of measured values from a set Ξ, and $\boldsymbol{c} \in \mathcal{C}$ consist of class labels for each measurement.

8.3.4 *Model Development and Validation*

The standard MBSE approach is to develop a model Φ and use verification techniques to show that the model satisfies the requirements R. In this approach, domain experts manually generate a model with parameter set

θ, and then estimate the parameters given data $\mathcal{D}$. The model is viewed as being "complete", in that it captures all necessary aspects of the application system.

In the model-based approach, the model Φ aims to predict the behaviour of the underlying system, which then is used for a top-level task, e.g., alarm generation. Hence the model aims to mimic the system's behaviour, i.e., to develop an accurate CPS model, as in Definition 8.3.3.

In traditional MBSE situations in which the model is validated using theorem proving, e.g., [26], one attempts to show that the model and requirements are consistent. Such model validation is typically performed without the use of data from real-world applications. In this case, if we have a model $\Phi(\theta)$ that is based on a set θ of parameters, we typically fix the parameters for model validation, and use these parameters during operation. The drawback of this approach is when the parameters do not match the real-world system, the software will never perform optimally. Further, if the operational system changes (due to natural degradation or changes in environmental conditions) the model also cannot adjust its performance by updating θ.

8.4 Running Example

This section introduces the phases of MBSE, and illustrates them with a well-known process-control example.

8.4.1 *Process-Control Example: Three Tank System*

To illustrate the concepts of this article, we use a running example of a well-known process-control example, a three-tank system. We show the 3-tank example in Fig. 8.2. The three-tank system is a prototype of many industrial applications in the process industry, such as chemical and petrochemical plants, oil and gas systems. The control of liquid level is a crucial problem in such process industries, and this is the control task on which we focus. While most articles on process control focus on the control issues, here we focus on the model-based methodology for generating the embeddable control and diagnosis software. We use the models developed for simulation and control in the software development process.

The system consists of three identical tanks, T_i, $i = 1, 2, 3$. (We use subscript i to denote tank i.) Pumps are used to provide input flows to tanks T_1 and T_2, with the flows denoted Q_1 and Q_2, respectively. Each

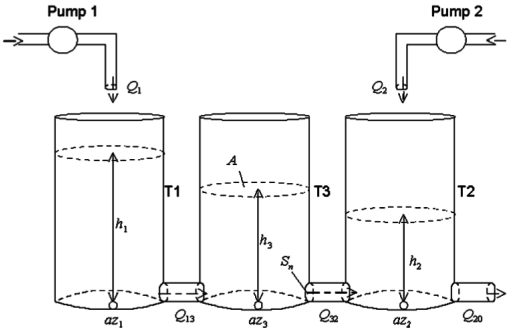


Fig. 8.2 3-tank system with inflows at the two outermost tanks.

input flow (Q_1 or Q_2) can be controlled to a level varying continuously between 0 and a maximum flow Q_{max}. We control flows between tanks using a switching valve for tank i, denoted V_i, $i = 1, 2, 3.$; each valve can be controlled with maximum values of open or closed. The liquid levels h_1, h_2, and h_3 in each tank can be measured with continuous-valued level sensors. The objective is to maintain set-point heights (h_1^*, h_2^*, h_3^*) in each of the tanks, by controlling the inflow rates and valve settings.

In the following, we will briefly illustrate the four steps of MBSE, but focus on model construction, which is the core topic of this article.

8.4.2 *Requirements*

Requirements generation is one of the most important MBSE steps, and generates a formal representation of the task requirements. Requirements generation for process control has been well studied, with a seminal paper being [27]. Recently, researchers have been developing methods to use ML for aspects of requirements generation, e.g., [8]. Reviews of this recent work on software requirements engineering using machine learning techniques are contained in [28, 29], noting that significant work is needed to create industrial-strength engineering tools.

We address the task of developing software for monitoring a hydraulic system for chemical process control. In this process-control system we frame the primary requirement as:

R1 pump two different chemicals into 3 tanks and ensure that the levels in the tanks are maintained at set-points for fixed time periods to ensure proper mixing of the chemicals (nominal operation)

R2 monitor the tank levels

R3 compute a residual in each of 3 tanks, computed as the difference between the measured and predicted levels, with a modelling accuracy level of q%

R4 if the residual exceeds a threshold, isolate the fault (in real time) causing that anomaly

R5 adjust the control system to tolerate any isolated fault, if possible (fault-tolerant operation)

This system is designed to have two main control regimes: (a) nominal operation and (b) fault-tolerant operation. We explicitly use a system model for developing these controllers.

Requirements engineering is a challenging task, and the mapping of requirements into system models is non-trivial. Also, requirements may change over time, for which machine learning is becoming increasingly useful [8]. We assume a complete set of requirements, and leave the application of AI/ML to both requirements and modelling of evolving systems to future work.

We can formally state requirement R_1 as minimizing the difference between the actual ($\boldsymbol{h}$) and set-point ($\boldsymbol{y}$) tank-height values, i.e.,

$$\min_{y(t), t \in [0,...,\tau]} \parallel \boldsymbol{h}(t) - \boldsymbol{y}(t) \parallel, \tag{8.1}$$

where $[0, ..., \tau]$ is the temporal window of interest.

To build a system to satisfy this requirement, we can define a system model Φ and show that the model satisfies requirement $R_1 - R_5$, i.e., the behaviour of the model does not lead to states that violate the level-control, safety and other constraints.

8.4.3 *Model Construction*

To build a system to satisfy these requirements, we can define a system model Φ and show that the model satisfies requirements $R_1 - R_5$, i.e., the behaviour of the model does not lead to states that violate the level-control, safety and other constraints. The most challenging model requirement is to balance the necessary accuracy level with real-time performance, e.g., fault isolation. Even for relatively simple non-linear systems like this tank example, using a first-principles ODE model may not lead to real-time fault isolation. Hence, in this article we focus on the model construction challenges of balancing accuracy with inference efficiency.

8.4.3.1 *Approach*

The classical approach to modelling a non-linear system is using ordinary differential equations (ODEs).

Generating models from data avoids manual model construction, but it still poses research challenges. Even state-of-the-art methods for inducing physics-based models from data still are limited to toy models, e.g., [33].

8.4.3.2 *Modeling Language: ODEs*

We model this non-linear system using ordinary differential equations (ODEs). Due to the difficulty of performing inference on this non-linear system, significant effort has been devoted to creating simplifications of non-linear systems, e.g., [30]. These simplifications include Mixed-Integer Linear Programs [31], smooth Metric Temporal Logics [32], linear approximations [30], among others. We use a linear approximation in this article. This section first describes our non-linear equations, and then a set of linear approximation of these equations.

Nominal System Description Φ_{nom}. For this 3-tank system (with inputs Q_1 and Q_2 at the outermost two tanks, and valves controlling the flows between adjacent tanks), the equations are given by

$$\begin{aligned}
\dot{h}_1 &= \frac{1}{A}[Q_1 - Q_{13} - Q_{1l}] \\
\dot{h}_2 &= \frac{1}{A}[Q_2 - Q_{32} - Q_{20} - Q_{2l}] \\
\dot{h}_3 &= \frac{1}{A}[Q_{13} - Q_{32} - Q_{3l}]
\end{aligned} \tag{8.2}$$

where

$$Q_{ij} = \eta_i V_i sign(h_i - h_j)\sqrt{2g|h_i - h_j|} \tag{8.3}$$

(through Bernoulli's law), η_i is a coefficient summarizing outflow parameters, A is tank area, and V_i is the [0,1] valve setting where 0 denotes closed and 1 open.

Fault System Description Φ_F. To create fault model Φ_F, we extend the equations in Φ_{nom} with fault equations covering possible faults in each state in Q. We assume that the impacts of the pump and valve faults are additive; as a consequence we model each fault independently in the fault model. Under this assumption, we can simulate the impact of multiple-fault scenarios by activating multiple faults at a time.

We assume that we can have faults in the pumps and valves. A pump fails off, such that the flow for pump i, $Q_i = 0$, for $i = 1, 2$. We model

a valve fault as an additive fault with (bounded) parameter $\zeta \in [-1, 1]$. When the valve setting is $V_i \in [0, 1]$, $i = 1, 2, 3$, the faulty setting is given by $(V_i + \zeta) \in [0, 1]$, $i = 1, 2, 3$. For example, if the tank is commanded to be open ($V = 1$) but is stuck shut then we model this using $\zeta = -1$ such that $V + \zeta = 0$.

8.5 AI-Based Framework for MBSE Task

We propose an AI-based framework as an alternative the the traditional MBSE approach, as shown in Fig. 8.3. Here, the V&V process involves the the model Φ, requirements R and *data* $\mathcal{D}$. We can no longer perform logical validation (since we have data), but instead perform an optimisation process $Opt(\Phi, R, \mathcal{D})$ so that we optimise an objective function over $(\Phi, R, \mathcal{D})$. The generated model Φ is updated continuously as the data generated from deployment changes over time.

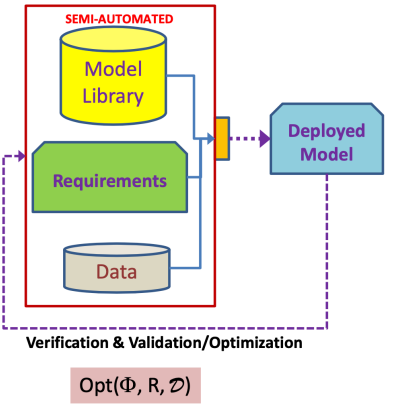


Fig. 8.3 Proposed approach to MBSE.

8.5.1 *Data-Driven MBSE*

Using AI and learning methods can take advantage of the recent availability of vast amounts of data, in order to reduce the onerous manual steps of MBSE. As a consequence, we re-define the MBSE process to incorporate data as follows:

Definition: [Data-driven MBSE Task $\langle R, M, \mathcal{D} \rangle$] Given a set R of requirements (for functionality/safety/privacy/security etc.), a component

model library M, and data $\mathcal{D}$, design a system model that can be guaranteed to meet the requirements R using model library M given data $\mathcal{D}$.

Given a system model Φ with faults, we can then use a range of techniques to verify if Φ satisfies R. In this article we focus on using AI-based methods to automate the model development process.

The MBSE approach of Definition 8.5.1 assumes that requirements are fully verifiable. In reality, requirements may be partially satisfiable. To address such situations, we generalize this definition to incorporate "soft verification" of requirements. To do this, we frame the software generation task as an *optimization problem.* In particular, we focus on a CPS application, where the underlying system has both cyber- and physical-aspects.

8.5.2 *Optimization-Based MBSE*

This section describes an MBSE framework that defines the overall MBSE task in terms of optimizing the outcomes of creating the embedded software, as represented in terms of a system model Φ.

Our aim is to define a CPS that optimizes an objective $\mathcal{J}$ subject to obeying:

- a set of physical constraints, which can be represented by a system model Φ;
- a set $\mathcal{R}$ of requirements;
- data $\mathcal{D}$ that can be used for model development (via learning) and/or model verification.

In this article we focus on the model-construction aspects of MBSE development. In contrast to traditional approaches that create a single model Φ that satisfies the requirements $\mathcal{R}$, we have a more general framework that aims to trade off three MBSE aspects:

- model accuracy–measured using a loss function $\mathcal{L}(\cdot)$;
- model (inference) complexity $\chi(\Phi)$;
- model development cost $\mathcal{C}(\cdot)$.

We presume that we have a model-development function $\varphi(M, \mathcal{D})$ that creates a model given a model library M and data $\mathcal{D}$. We denote the space of possible models as the powerset of φ, namely $\mathfrak{P}^{\varphi}$.

We select a model using a regularization framework, using the equation:

$$\Phi^* = \arg \min_{\Phi \in \mathfrak{P}^{\varphi}} \left[\mathcal{L}(\Phi, \mathcal{D}) \; + \; \lambda_1 \chi(\Phi) \; + \; \lambda_2 \mathcal{C}(\Phi) \right], \tag{8.4}$$

where λ_1 and λ_2 are the regularization weights for the complexity and development costs, respectively.

We now describe these aspects in more detail.

8.5.2.1 *Model Accuracy*

The loss function measures the degree to which a model Φ satisfies the requirements R, denoted as $\mathcal{L}(\Phi, R)$. This measure is a strict generalization of the traditional proof-theoretic notion of requirement validation. We can use a variety of measures, such as squared-error loss between model predictions and actual data.

8.5.2.2 *Model Complexity*

Model complexity measures the "size" of the model, e.g., [34]. We can use three complexity measures: (1) An explicit representation in terms of "degrees of freedom" of a model, e.g. effective number of parameters; (2) code length, a.k.a. "Kolmogorov complexity" (the longer the shortest model code, the higher its complexity, e.g., in bits); (3) information entropy of parametric or predictive uncertainty [35]. The literature contains several measures for each type, e.g., for degrees of freedom we can use AIC or BIC [36].

8.5.2.3 *Model Development Cost*

Another important aspect is the cost of developing a model. With manual construction from a model library, the cost must be estimated from the cost of the model components and the manual generation process. With automated construction, the cost is based on that of the data acquisition and learning process.

8.5.3 *System Verification*

Given a model whose parameters are optimized with respect to data, we then verify that the model does not violate our requirements. Verification can be decomposed into two steps: validating the model with respect to collected data, and then validating the run-time system with regards to the requirements and run-time data. We examine each in turn.

We address situations in which we use data from real-world applications for model generation and validation. To generate a model that is consistent

with regard to data (i.e., make the model best fit the data), we use $\mathcal{D}$ to estimate the parameters θ of Φ.

Given a model whose parameters are optimized with respect to data, we then verify that the model does not violate our requirements. Verification can take many forms; for example, for dynamical systems we may perform reachability analysis to verify the system trajectories do not end up in forbidden states [37]. This consists of using the model Φ together with initial conditions $\boldsymbol{x}_0$ to generate trajectories, which are checked for intersection with forbidden states.

8.6 AI-based MBSE Model Construction Methods

This section reviews approaches (that can be applied to MBSE) for generating models using both (a) physics and/or model-libraries and (b) data. These approaches span a range from purely data-driven approaches (generating black-box models) [38], hybrid approaches combining physics and data (generating grey-box models) [39], to model-driven approaches (white-box) [40]. In the following, we review the data-driven and hybrid approaches, since these are not well-known as MBSE approaches. Based on these modeling approaches, Sec. 8.9 presents an empirical comparison of these model classes.

8.6.1 *Data-Driven Approach*

This approach assumes that we have no manually-generated model, but we generate a model directly from data $\mathcal{D}$. In contrast to the model-based approach, which uses a model Φ aims for tasks like safety analysis or alarm generation, a learning-based approach trains a learning model to directly predict the necessary class labels for the top-level task; in other words, we need not represent the CPS model at all [38]. We generate a classifier Γ with parameters $\boldsymbol{w}$. To learn Γ, we frame our optimization task in quite a different way to the MBSE task: We want to maximize the classification accuracy over the data, in which case we define a loss function as the classification loss: i.e., if our classifier Γ outputs class $\hat{c}$ for data-pair (ξ, c), we minimize the loss as follows:

$$\boldsymbol{w}^* = \arg \min_{w \in \mathcal{P}^w} \sum_{D \in \mathcal{D}} \parallel \hat{\boldsymbol{c}} - \boldsymbol{c} \parallel \tag{8.5}$$

There are many different targets that can be learned, which include:

- **CPS models**: Significant work has been performed to learn CPS models from data, e.g., [41].
- **Fault Trees**: Examples of learning fault trees include [19, 42].

Many papers have applied learning to fault diagnosis of the multi-tank system, e.g., [43]. In order to learn a classifier, we need to generate data that covers nominal as well as all fault conditions.

One of the issues that must be addressed in learning for diagnostics is the *class imbalance problem*, where the data set has an unequal distribution between it's majority (nominal state) and minority (fault state) classes. This imbalance arises since most data that is measured corresponds to nominal cases, i.e., few fault cases are recorded given that faults are rare. Addressing class imbalance problems is important, as it leads to several problems [44], which include:

- learning a classifier to identify the minority class is very complex and challenging since a high-class imbalance introduces a bias in favour of the majority class;
- the predictive accuracy of the classifier (i.e., ratio of test samples for which we predicted the correct class) is not longer a good measure of the model performance, since predicting a nominal state always yields the proportion of nominal in the data set (e.g., 98%), thereby failing in the real task, i.e., correctly predicting fault cases.

In order to address class imbalance in training a classifier, we can use several techniques, such as:

- Using *data augmentation* to artificially increase the proportion of fault cases in the data by perturbing existing fault cases to create novel fault cases;
- Train using a loss function designed for achieving a *good trade-off between sensitivity and specificity*; this will correct "traditional" training methods, that create a classifier that never predicts faults (and hence has a sensitivity of 0 and a specificity of 1).[1]

[1] Sensitivity and specificity, though theoretically independent, typically trade off against each other and are inversely related [45].

8.6.2 *Hybrid Approach: Multi-Fidelity (Surrogate-Based) Optimization*

Multi-fidelity methods leverage models and/or data of multiple levels of fidelity in order to maximize the accuracy of model estimates, while minimizing the cost associated with parameterisation [46]. The model-based methods adopt a set of generative models, while the data-driven methods learn a model, e.g., regression-based model, physical model [47], or a reinforcement-learning model [48].

8.6.2.1 *Model-based Multi-Fidelity Approach*

In situations where a high-fidelity model Φ_H exists but is computationally too expensive to use for inference, surrogate low-fidelity models can be used instead through data-driven tuning [49–51]. A hybrid surrogate model takes advantage of the predictive ability of each individual surrogate model through a weighted-sum combination of the individual surrogate models. In other words, we approximate the system behavior (high-fidelity behavior) using a linear combination of the low-fidelity predictions and a discrepancy function (e.g., polynomial function). The key idea is to consider the low-fidelity model as a basis function in the multi-fidelity model with the scale factor as a regression coefficient. We can perform least-square estimation based on inputs of the low-fidelity model and the discrepancy function. We compute the scale factor and coefficients of the basis functions using linear regression, which guarantees the uniqueness of the fitting process. Besides enabling efficient estimation of the parameters, the proposed least-squares multi-fidelity surrogate can be applied to other tasks.

In this approach, if we have a high-fidelity model Φ_H with output y_H, and a low-fidelity model Φ_L with output y_L, then we can tune the system such that

$$y_H(x) = \boldsymbol{\alpha} y_L(x) \; + \; \delta(x), \tag{8.6}$$

where the regression scale factor $\boldsymbol{\alpha}$ and discrepancy function $\delta(x)$ are obtained through optimization, see [51].

Provan presents an approach that learns a polynomial function for the the scale factor $\boldsymbol{\alpha}$ in conjunction with multiple low-fidelity models $\Phi_{L_1}, \cdots, \Phi_{L_k}$ [52].[2] Feldman *et al.* takes a different approach given as input a collection of component models of differing fidelity: they perform

[2]This approach is similar to ensemble learning, where a weighted combination of multiple classifiers is generated.

an exhaustive combinatorial search over all component combinations to generate the system-level model that best fits the data $\mathcal{D}$ [53].

Various packages that build surrogate models have been developed using different programming languages, such as Scikit-learn in Python [1], SUMO in MATLAB [54], and Surrogate Modeling Toolbox (http://smt.readthedocs.io, [55]).

8.6.2.2 *Data-Driven Surrogate Approach*

Data-driven surrogate models (also known as meta-models, or response surface models) are constructed using *simulation data* from high-fidelity models [56]. Hence, they are approximate *black-box models* that provide black-box relationships between inputs and outputs of a system. We construct data-driven surrogate models by simulating the original (high-fidelity) model at a set of points, called training points, and using the corresponding evaluations to construct an approximate model based on mathematical functions (e.g., Kriging, Gaussian-Process functions, etc.).

The main challenge in data-driven surrogate modeling is how to obtain an approximate model from the simulation data that is as accurate as possible over some domain of interest while minimizing the simulation cost of the data generation. This challenge necessitates the appropriate selection of the structure and complexity of the surrogate model, the number and distribution of the simulation data used for learning the surrogate model, and the validation methods used for estimating the quality of the model. Several tool kits have been developed to enable this approach, e.g., [54].

8.6.3 *Hybrid Approach: Model-Constrained Optimization*

The hybrid approach uses a combination of models and learning for software generation [57]. In this case, we assume that we have an *incomplete model* $\breve{\Phi}$ and data $\mathcal{D}$. We represent the unknown part of the model in terms of a subset $z(t)$ of state variables.

Definition: **Incomplete CPS model** $\breve{\Phi}$. We model an incomplete CPS model as a dynamical system as follows:

$$\dot{x} = f(x(t), u(t), z(t), \theta),$$
$$y = h(x(t), u(t), z(t), \theta),$$

where

- $x \in \mathbb{R}^n$ is the state vector,

- $u \in \mathbb{R}^m$ are known process inputs (manipulated variables or measured disturbances taking arbitrary values independently of the rest of the variables),
- $z \in \mathbb{R}^q$ are algebraic variables (with undefined roles, e.g., representing arbitrary inputs or variables that are functions of other variables),
- $\theta \in \mathbb{R}^\rho$ are model parameters (assumed to be constant) and
- $f(\cdot) \in \mathbb{R}^n,\ h(\cdot) \in \mathbb{R}^l$ are nonlinear functions of their arguments.

Given the partial model, we define the problem as finding the sub-model $z(x, u)$ and parameters θ such that the response of $\breve{\Phi}$ optimally fits the experimental values $\mathcal{D}$. This approach has the potential to provide the best balance between physical models and empirical data, but is computationally intensive. The intractability arises since we are solving a semi-infinite programming problem (an optimization model that has finitely many variables and infinitely many constraints). Recent approaches have explored more tractable optimization algorithms, e.g., [57].

8.7 Running Example: Continued

This section uses our running example to illustrate and compare the different AI-based frameworks.

8.7.1 *Model-based Multi-Fidelity Approach*

To illustrate how surrogate models can be used for the tank system, we can replace the high-fidelity model Φ_H with a low-fidelity (but computationally simpler) model Φ_L as follows. We define a surrogate model Φ_L by replacing the non-linear model (Eqs. (8.2) and (8.3)) with a linear model of the nominal tank system using Eqs. (8.2), except that the outflow from tank i to tank j $(i = 1, 2 \quad j = 2, 3)$ is given by

$$Q_{ij} = \eta_i V_i sign(h_i - h_j) 2g(h_i - h_j). \tag{8.7}$$

For details of such a linearization process, see [58].

We then fit Φ_L to data $\boldsymbol{D}$. In this case, we use simulated data for this fitting. If we simulate the non-linear/linear systems under equivalent initial conditions x_0, we can obtain outputs y_H/y_L, respectively. For such a collection of non-linear/linear simulation outputs $\boldsymbol{D} = \{(\boldsymbol{y}_H, \boldsymbol{y}_L)\}$, we then compute parameters $\boldsymbol{\alpha}, \delta(x)$ to optimize the surrogate model's fit:

$$\boldsymbol{y}_H(x) = \boldsymbol{\alpha}\boldsymbol{y}_L(x) \ + \ \delta(x). \tag{8.8}$$

8.7.2 *Data-Driven Surrogate Approach*

A data-driven surrogate approach, rather than using physics-based surrogate models as just described, learns a low-fidelity model Φ_L from data, and then combines the model's outputs using optimization parameters (e.g., $\boldsymbol{\alpha}$, $\delta(x)$). We can create a neural network surrogate (simulation) model, e.g., as in [59,60], where the model class learned is a Radial Basis Function (RBF) network; alternatively, we can learn a Bayesian network, as in [61].

8.8 MBSE Trade-Off Framework

This section describes our experimental methods, focusing on the models compared and the relative model-construction costs that are key to our comparative analysis.

8.8.1 *Approaches Compared*

In our experiments we compared several approaches, as shown in Table 8.1. We compare the different approaches using the process-control model. We define a gold-standard (high-fidelity) model Φ_H that is manually developed within a traditional model-based framework. We then compare the fault-identification model cost of Φ_H with the that of models that use data for model-generation, using multi-fidelity surrogates, data-driven learning, and model-constrained optimization. We use as our cost metric a measure of manual effort necessary for model-construction.

Table 8.1 Code-generation approaches compared.

Approach	**Notation**	**Diagnostics inference**
Gold standard	Φ_H	Observer-based Engine $\mathcal{E}$
Data-driven (NN)	Φ_{NN}	Neural Network classifier
Model-based Surrogate	Φ_{L_m}	Observer-based Engine $\mathcal{E}$
Data-driven Surrogate	Φ_{L_d}	Observer-based Engine $\mathcal{E}$

We assign cost to the following operations:

- Manual model construction: components and complete models.
- Manual data preparation and class-label assignment.
- Data-driven model construction.

Table 8.2 summarises this comparative cost information. We note that we have assessed these costs based on the models we have generated, and

Table 8.2 Comparative costs for model construction among modeling approaches (all figures in person-hour estimates).

Approach		Library	Model	Data	Total
Manual	HF	100	30	-	130
	LF	60	30	-	90
Machine Learning	HF	-	-	50	50
MB-Surrogate	HF	100	-	-	100
	LF	60	-	-	60
DD-Surrogate	HF	100	-	10	110
	LF	60	-	10	70

we assign all operations (model construction, data preparation, etc.) to have identical cost. It is beyond the scope of this article to assess the costs in a more precise manner.

8.8.2 *Model Library Costs*

We assess costs based on the person-hours required to develop the model libraries. Creating such libraries is labour-intensive, but the benefit is that the models can be re-used. In addition, many commercial libraries exist, both free (e.g., Modelica) and paid, so these costs could be reduced through the use of commercial libraries. We further note that the high-fidelity (HF) models are more expensive to create that the low–fidelity (LF) models.

8.8.3 *Data-Driven Model Costs*

The main costs for data-driven model generation are due to the cost of manual data preparation and class-label assignment (i.e., labeling the test-cases for the data). It is now well known that at least 50% of costs for data mining are accrued for data preparation.

8.8.4 *Comparative Analysis*

Table 8.2 summarizes the comparative costs of the different approaches. The Machine Learning approach is the cheapest, with the other approaches indicating the the low-fidelity model-based methods the next cheapest. Due to re-usability of models, this table may be somewhat misleading, since with new data an entirely new model must be learned using the Machine Learning approach; recent research in transfer learning (e.g., [62]) aims to use prior learned models to avoid this problem.

8.9 Empirical Modeling Cost Comparison

8.9.1 *Empirical Analysis*

We compare traditional and several AI-based (data-driven) MBSE approaches. In the proposed optimisation-focused MBSE process, our optimisation task incorporates both the model construction costs and model accuracy; however, at present these are computed independently. Future work is needed to be able to fold both into a single cost function and subsequently optimize the entire MBSE process.

This section describes our empirical analysis. We compare the different approaches using a process-control model. We define a gold-standard (high-fidelity) model Φ_H that is manually developed within a traditional model-based framework. We then compare the fault-identification performance of Φ_H with the performance of models that use data for model-generation, using multi-fidelity surrogates, data-driven learning, and model-constrained optimization. We develop and train all models using a MATLAB/Simulink platform.

We define an objective function for the experiments as follows:

$$\mathcal{J} = \lambda\Delta + (1 - \lambda)(\tau^* - \tau), \tag{8.9}$$

where

- Δ is the diagnostic accuracy, which we compute in terms of classification error (equation 7 of [63]);[3]
- τ is the diagnostic inference time for the total number of cases;
- τ^* is the target diagnostic inference time;
- λ is a regularization parameter that balances Δ and τ.

Here, we penalize an approach that exceeds the target diagnostic inference time using the $(\tau^* - \tau)$ term. Embedded diagnostics applications typically need timely diagnostics results, leading to application-specific target diagnostic inference times (τ^*). For example, aerospace systems have stringent values of τ^* so that fault-tolerant control (FTC) can be implemented in (near) real-time when a controller uses fault-isolation inputs as part of its control loop, e.g., [65]. In this article, we use a fixed value for τ^*, similar to the maximum fault-isolation requirements for FTC.[4]

[3]Several metrics for diagnostics accuracy have been proposed in the literature, e.g., [63, 64]; any of these metrics can be used for Δ.

[4]Fault isolation for real systems follows a distribution of times, since some faults are computationally harder to isolate than others. It is beyond the scope of this article to

8.9.2 *Data*

We simulate a data set $\mathcal{D}$ from the gold standard model Φ_H by sampling behaviours given nominal and faulty scenarios. We divide $\mathcal{D}$ into training $\mathcal{D}_{train}$ a test $\mathcal{D}_{test}$ subsets, with 90% used for training and 10% for testing.

We denote the system modes $\boldsymbol{\eta}$ as consisting of nominal (no faults, i.e., $\eta = \emptyset$), and faults in (a) pump P_1; (b) pump P_2; (c) valve V_{13}; (d) valve V_{32}; (e) valve V_{20}; and (f) combinations of the fault conditions. We sample modes according the a probability distribution over the fault occurrence, as follows: $P(\eta = \emptyset) = 0.93$; $P(\eta = P_1) = 0.01$; $P(\eta = P_2) = 0.01$; $P(\eta = V_{13}) = 0.01$; $P(\eta = V_{32}) = 0.01$; $P(\eta = V_{20}) = 0.01$.

For each test case, we simulate a behaviour B over $t = 100$ seconds using an initial condition x_0; we inject a fault into some scenarios (according to the mode distribution just described) at $t = 15$ seconds. We measure the tank heights (h_1, h_2, h_3) over $t = 100$ seconds, and our objective is to identify if a fault occurred during the simulation.

8.9.2.1 *Baseline: Model-Based Approach*

This approach uses the model Φ_H, which has had its parameters assigned using the training data $\mathcal{D}_{train}$. Φ_H provides a forecast of expected behaviour of the system, and we use an observer-based diagnosis algorithm [66] to isolate faults.

8.9.2.2 *Data-Driven Approach*

We learn a classifier given the input (B, η). For this application, we have chosen to learn a neural network. We divide the simulation period [0,100] into a sequence of intervals using a temporal window, and we learn the relationships between adjacent intervals to simplify the learning process. Note that this is supervised learning, in that each training case is labeled with the fault class. The output layer consists of the fault class values.

8.9.2.3 *Model-Based Hybrid Approach*

We selected 3 low-fidelity tank models, denoted Φ_{L1}, Φ_{L2}, Φ_{L3}, developed using the approach described in [52]. Then, using the training data $\mathcal{D}_{train}$,

explore such issues, but they could be incorporated into a regularisation task as described in Eq. (8.9).

we learned a polynomial function such that

$$y_H(x) \simeq \sum_{i=1}^{3} \boldsymbol{\alpha}_i y_{Li}(x) \ + \ \delta(x), \tag{8.10}$$

where $y_{Li}(x)$ is the output of Φ_{Li} given input x.

8.9.2.4 *Data-Driven Hybrid Approach*

We used training data $\mathcal{D}_{train}$ to learn a neural network function that generates output $\boldsymbol{y}(\boldsymbol{x})$ at time steps $t = 10, 20, \cdots, 100$.[5] We then used the trained network $\mathcal{D}_{train}$ as the "low-fidelity" model used to generate a polynomial function, such that Eq. (8.10) holds at time steps $t = 10, 20, \cdots, 100$.

8.9.3 *Results and Discussion*

We have averaged our results over a set of 100 test cases, which contained 92 nominal scenarios, 7 single- and 1 multiple-fault scenarios. Table 8.3 summarizes our results.

Table 8.3 Code-generation results, with the best results for each category shown in bold-face.

Approach	**Accuracy** Δ	**time** (s)	$\mathcal{J}: \ \lambda = 0.9$	$\mathcal{J}: \ \lambda = 0.6$
Φ_H	**96**	103.5	**81.1**	26.2
Φ_{NN}	64	**3.8**	62.2	56.9
Φ_{L_m}	85	41.3	77.4	54.5
Φ_{L_d}	78	73.1	21.4	**58.2**

8.9.3.1 *Accuracy*

- Φ_H: The fact that the gold-standard approach did not achieve 100% accuracy is due to the inference engine not being perfect in its diagnostics ability.
- Φ_N: We argue that the poor accuracy of the NN is due to issues with training: there were insufficient training cases ($n = 900$), and significantly more work is necessary to either pre-process the data so that training is simplified, or to tune the architecture. This shows that NN training shifts the burden from manual model generation to (at present) manual pre-processing and architecture tuning.

[5] We trained a deep LSTM network consisting of 6 densely-connected layers. The LSTM was found to capture the temporal aspects of this application and produce better results than atemporal networks.

- **Surrogates**: The model-driven surrogate approach performed the best of the non gold-standard approaches, with the data-driven being slightly worse due to issues with training.

8.9.3.2 *Time*

- Φ_H: The gold-standard approach was computationally the most expensive due to the complexity of ODE inference for diagnostics.
- Φ_{NN}: The NN shows very fast inference, the timings only hampered by the number of parameters for the classifier that need to be processed.
- **Surrogates**: The model-driven surrogate approach performed faster than the gold-standard approach, but still needed to execute costly ODE operations. The data-driven surrogate approach performed faster since it has the fewest parameters in it's pseudo-simulator that must be evaluated.

8.9.3.3 *Combined (Weighted) Objective*

- The optimal value of $\mathcal{J}$ depends on the different parameters used. For $\lambda = 0.9$ (accuracy is weighted highly) the gold-standard approach was optimal. However, if inference speed is also important ($\lambda = 0.6$) the data-driven surrogate approach is optimal.

8.9.4 *Discussion of Trade-Offs*

In this article we are ignoring the cost of parameter estimation or of training the different models.

Modeling Costs: The model-development costs for the NN are significantly higher than those for the other approaches. As a consequence, our results focus on the on-line inference costs and accuracy.

Explainability: The approaches differ considerably in terms of explainability. The gold-standard approach is best in this regard, since all equations are physically well-founded and understandable. The model-driven surrogate approach is also highly transparent, since it weights a collection of physics-based models. The data-driven models, which consist of just weights, do not have explanatory power since they have no physical equations to relate their results to.

8.10 Conclusion

This article has proposed an optimization-based framework for MBSE. Using this approach we have examined traditional (manual) and AI (data-driven) methods for embedded MBSE. We illustrated our framework using an embedded-systems application, in which the aim is to generate embedded code for monitoring a hydraulic tank benchmark such that we isolate faults with high accuracy and inference-time deadlines. The optimization-based framework enables developers to trade off a variety of system parameters in building the embedded code.

This work shows the benefits of AI-based methods for automating MBSE, in that manual model generation can be replaced by data-driven methods. Significant work remains, as this is just a preliminary evaluation of the trade-offs entailed in such automation. Purely data-driven methods are not explainable, and require significant data for training. For diagnostics applications, real-world data for faults in hard to acquire, which may limit the developed classifiers. However, for applications where physics-based models are expensive or impossible to develop, AI-based models show great promise if data is available.

References

[1] O. Lisagor, T. Kelly and R. Niu, Model-based safety assessment: Review of the discipline and its challenges, in *The Proceedings of 2011 9th International Conference on Reliability, Maintainability and Safety*. IEEE, pp. 625–632 (2011).

[2] M. Brambilla, J. Cabot and M. Wimmer, Model-driven software engineering in practice, *Synthesis Lectures on Software Engineering* **1**, 1, pp. 1–182 (2012).

[3] C. Hartsell, N. Mahadevan, S. Ramakrishna, A. Dubey, T. Bapty, T. Johnson, X. Koutsoukos, J. Sztipanovits and G. Karsai, Model-based design for cps with learning-enabled components, in *Proceedings of the Workshop on Design Automation for CPS and IoT*. ACM, pp. 1–9 (2019).

[4] L. E. Lwakatare, A. Raj, J. Bosch, H. H. Olsson and I. Crnkovic, A taxonomy of software engineering challenges for machine learning systems: An empirical investigation, in *International Conference on Agile Software Development*. Springer, pp. 227–243 (2019).

[5] J. Zhang and J. Li, Testing and verification of neural-network-based safety-critical control software: A systematic literature review, *arXiv preprint arXiv:1910.06715* (2019).

[6] S. L. Brunton, J. L. Proctor and J. N. Kutz, Discovering governing equations