
CHALMERS
UNIVERSITY OF TECHNOLOGY

# Recognizing lines of code violating company-specific coding guidelines using machine learning: A Method and Its Evaluation

Empirical Software Engineering (2020) 25:220–265
https://doi.org/10.1007/s10664-019-09769-8

# Recognizing lines of code violating company-specific coding guidelines using machine learning

## A Method and Its Evaluation

**Miroslaw Ochodek[1,2] · Regina Hebig[3] · Wilhelm Meding[4] · Gert Frost[5] · Miroslaw Staron[3]**

Published online: 14 November 2019


**Abstract**

Software developers in big and medium-size companies are working with millions of lines of code in their codebases. Assuring the quality of this code has shifted from simple defect management to proactive assurance of internal code quality. Although static code analysis and code reviews have been at the forefront of research and practice in this area, code reviews are still an effort-intensive and interpretation-prone activity. The aim of this research is to support code reviews by automatically recognizing company-specific code guidelines violations in large-scale, industrial source code. In our action research project, we constructed a machine-learning-based tool for code analysis where software developers and architects in big and medium-sized companies can use a few examples of source code lines violating code/design guidelines (up to 700 lines of code) to train decision-tree classifiers to find similar violations in their codebases (up to 3 million lines of code). Our action research project consisted of (i) understanding the challenges of two large software development companies, (ii) applying the machine-learning-based tool to detect violations of Sun's and Google's coding conventions in the code of three large open source projects implemented in Java, (iii) evaluating the tool on evolving industrial codebase, and (iv) finding the best learning strategies to reduce the cost of training the classifiers. We were able to achieve the average accuracy of over 99% and the average F-score of 0.80 for open source projects when using ca. 40K lines for training the tool. We obtained a similar average F-score of 0.78 for the industrial code but this time using only up to 700 lines of code as a training dataset. Finally, we observed the tool performed visibly better for the rules requiring to understand a single line of code or the context of a few lines (often allowing to reach the F-score of 0.90 or higher). Based on these results, we could observe that this approach can provide modern software development companies with the ability to use examples to teach an algorithm to recognize violations of code/design guidelines and thus increase the number of reviews conducted before the product release. This, in turn, leads to the increased quality of the final software.

Communicated by: Lin Tan

✉ Miroslaw Ochodek
miroslaw.ochodek@cs.put.poznan.pl

Extended author information available on the last page of the article.

**Keywords** Measurement · Machine learning · Action research · Code reviews

## 1 Introduction

Software developers in big and medium-size software development companies and organizations are working with codebases that are usually of several millions of lines of code. To cope with this code size and at the same time comply with the agile and lean paradigms, e.g. continuous delivery, developers have access to tools that help them with the code review management. Code reviews tools (e.g. static, dynamic, automatic, manual, purchased/own developed review tools) are important for maximizing the quality of the software product that is under development.

Violations in coding guidelines lead to a double negative effect. On the one hand, having to check for inconsistent styles and violated guidelines requires time and slows down the reviewers—taking away time for finding serious mistakes or bad smells. On the other hand, it is known that inconsistent styles make it harder for reviewers to read and understand the code (Maruping et al. 2009; Smit et al. 2011).

Many companies adopted fast-feedback loops where the source code is reviewed per commit, in small chunks and often outside of the compiler environment (McIntosh et al. 2014). Tools like Gerrit are used for managing these reviews, but they need integration with tools checking for violations of code/design guidelines.

These code analysis tools and style checkers have the potential to support code reviews. These techniques have advanced significantly during the recent years and can provide over 90% accuracy in finding potential code guideline violations for pre-defined rules. Also, they have been shown to trigger the right discussions during the code review. Singh et al. (2017) showed that 73% of the suggestions from the static analysis led to discussions and, in turn, improvements of the source code quality.

However, continuous integration with its fast-feedback loops poses a challenge for the adoption of these tools in the industry. Source code is often too large to be compiled completely for each commit, and tools like Gerrit even promote the review at the level of code commit differences. In addition, coding styles, and thus code/design guidelines, are often company or even project-specific (Torunski et al. 2017), and evolve as the product matures. For example, some conventions used by our industry partners can be found in widely accepted standards or coding conventions, such as camel case variable naming. Others can become very specific to the companies programming environment, concerning aspects such as build time variants. Thus, static code analysis tools and style checkers need to be constantly extended to capture the new rules[1] and the evolution of the existing rules. Both the need for analyzing incomplete code and flexibility in evolution are not well supported by existing tools. This construction can require significant effort as it requires practitioners to learn specific APIs (Ochodek et al. 2017). Finally, our industrial partners develop and use their custom Domain Specific Languages (DSLs) for which static code analysis tools do not exist and since they are mostly proprietary and company-specific there are not enough resources to develop and maintain dedicated static code analysis tools.

[1]We will use the term "rule" interchangeably with the term "guideline" since the guidebooks of our industrial partners and most of the commonly adhered coding conventions are composed as sets of rules.

In this paper, we address the problem of *How to support new ways of reviewing code in continuous integration in large and medium-size software companies?* We present an action research study (Susman and Evered 1978; Baskerville and Wood-Harper 1996) that aimed at investigating the possibilities of supporting code reviews by automatically recognizing company-specific code guidelines violations in large and medium scale, industrial source code, without the need to learn new ways of specifying the rules – i.e. by example.

Our action research study was designed in four cycles, which gradually progressed from theoretical studies and theory-building into industrial practice and action-taking, following the organizational change model by Goodman et al. (1980). In the first cycles, we follow the "research as action" principle and we gradually shift to "change as action" in the last cycles, as practiced often in collaborative action research as prescribed by Masters (1995). We worked with two companies which develop embedded software in Scandinavia: a large infrastructure provider and provider of consumer products in the embedded software domain. Both companies allowed us to analyze the source code of their products and provided access to architects and designers—all with over 10 years (partially over 20 years) of experience in the domains.

As a result, we developed a method and implemented a machine learning-based (ML-based) tool to recognize violations of code/design guidelines that is trained on a small number of examples, is language-agnostic, and does not require the code being analyzed to compile. In the method developed together with our industrial partners, instead of defining and developing rules, the software architects recognize violations in the code, mark the code and teach the tool to recognize similar patterns. Over time, the examples evolve as the codebase evolves, which reduces the need for manual management of rules, and in the end, leads to automation of code reviews using company-specific coding guidelines. The opinion of the practitioners from the cooperating companies is the tool in its current version can complement static code analysis tools they use (e.g., to perform quick or partial codebase quality checks before the complex static code analysis tools are run to analyze the whole codebases) or fill the existing gap by enabling analyzing code written in programming language for which such static code analyzers do not exist or code that does not compile. Ultimately, the example-based approach could potentially evolve to automatically learn to recognize poor quality code from the automatic code reviews. However, at this stage, we focus on training the tool to detect violations of existing company-specific guidelines.

The contributions from the study are:

- we found that by using machine learning (ML) to perform a line-level code analysis it is possible to match the accuracy of static code analysis when identifying violations of guidelines requiring to understand a limited code context (i.e., a single line or a few lines of code),
- we show that using active learning for sampling training data provides the most accurate results in recognizing violations of company-specific coding guidelines and allows reducing the effort required to train the ML-based tool (a smaller number of training examples was needed to achieve the same (or higher) prediction quality as in the case of manual selection of examples),
- we show that the frequencies of tokens are a valuable source of information while recognizing code violations and allow to perform this task without the need for parsing or compiling the code,
- we show that the approach works both on industry-wide standards applied to open source and on the company-specific, proprietary guidelines applied to professionally

developed code from two large companies. Therefore, using the ML-based tool can help to reduce the effort of manual code review, and
– we report observations from the *in situ* application of ML to analyze code in industrial environments that could help practitioners to adopt ML-based approaches in their companies (i.e., the strategies to minimize the effort of labeling data, the effect that the guidelines and code evolution can have on the accuracy of an ML-based tool).

The rest of the paper is structured as follows. Section 2 outlines the most important related research in the area of code reviews and applying machine learning for this task. In Section 3, we present a machine-learning based code analyzer. Section 4 describes the research methodology applied and our research design. The results of each of the four action-research cycles are presented in Section 5. Section 6 discusses threats to validity of our study. Finally, we summarize the findings in Section 7.

## 2 Related Work

Code reviews have been studied extensively and we started first with the overview of the existing works in this area. We review the related work from two perspectives—comparison between static code analysis/style checker tools and from the perspective of academic and using machine learning for static code analysis.

### 2.1 Comparison Between Tools

Static code analysis tools and style checkers are widely used development tools with over hundreds of tools listed alone in Wikipedia. The compare a sample of these tools for this paper, we decided on a set of comparison criteria inspired by the taxonomy of Novak et al. (2010):

**Supported Languages:** Due to the needs of our industry partners, we will focus on tools for C and C++.

**Compilation Requirements:** This criterion is inspired by Novak's *technology* and *input* dimensions. The focus is on the tool's requirements on the syntactic correctness and completeness of the code. Possible values are: "parsing" (code needs to be syntactically correct)", "linking" (code needs to be complete to run the analysis), and "robust" (robust against most syntactical errors, no linking required).

**Extensibility:** The criterion is a refined dimension from Novak et al. (2010) and describes whether the tool can be extended with additional rules/checks. Possible values are: "no" (extensible only by feature request), "imperative" (extensible by writing imperative code, e.g. in plug-ins), "declarative by rule" (extensible by declarative specification of rules), and "declarative by example" (extensible by declaration of valid and invalid code examples).

**Configurability:** The category refines Novak's dimension with the focus on the subject of the configuration. Possible values are: "no" (rules cannot be configured), "rules by parameter" (single rules can be configured via parameters), "rules by example" (single rules can be configured via examples), "ruleset" (Selection of what rules should be applied - single rules or sets of rules), and "other" (other configuration options).

**Interoperability:** The category is inspired by Novak's *user experience* dimension and focuses on how the tool can be used in a tool environment. Possible values are: "stand-alone" (the tool has a stand-alone version), "IDE" (there are IDE/Editor plug-ins available), and "collab. tools" (Integration to collaborative development tools, such as Github and Gerrit is available).

**Access to Results:** The category captures, inspired by Novak's *output* dimension, whether the tool's results can be accessed externally or exported. Possible values are: "no external" (the results cannot be exported easily), "API" (the results can be accessed via an API, e.g. via web services), and "file-export" (Results can be exported to file formats such as HTML or text).

**Static Code Analysis Tools for C/C++** In the following paragraphs, we discuss a sample of static code analysis tools for C and C++ with regards to the criteria above (see Table 1). We selected the sample by starting with the tools that are discussed in known comparative studies on static code analysis tools, namely the works of Fatima et al. (2018), Emanuelsson and Nilsson (2008), Mantere et al. (2009), and Shaukat et al. (2018), and (Brar and Kaur 2015).

In the end we chose to select the 6 tools addressed by most of these studies: Coverity Scan[2], KlocWork[3], PolySpace[4], Splint[5], CPPcheck[6] and Flawfinder[7]. All of these tools are mostly targeting the detection of errors and security issues in the code.

Most of the static code analysis tools require the code to parse and link, due to their analysis methods: Coverity Scan and KlocWork perform data-flow analysis, PolySpace uses formal methods, and Splint uses theorem proving. Some of these tools can run on a single source file but perform a shallow analysis, based on assumptions on the missing code, only. The exception is Flawfinder, which can handle code that is incomplete and does not necessarily parse, due to the token-based analysis.

Coverity Scan (according to Emanuelsson and Nilsson (2008)) and KlocWork can both be extended with new checkers imperatively. Splint, CPPCheck, and Flawfinder can be extended as well by the declaration of new rules. This happens in the form of regular expressions in CPPCheck, which is not easy as internal variables need to be known. Flawfinder allows exchanging the database with new pattern specifying invalid situations. None of these extension mechanisms is trivial to use.

KlocWork allows to configure single rules, e.g. by defining metric thresholds, and to define rulesets (called "taxonomies") and PolySpace allows a selection of the rules to be applied. CPPcheck allows to configure the used language version and Flawfinder allows to exclude code-lines from the analysis.

With the exception of Splint, each tool can be integrated into at least one IDE or code editor, such as Eclipse, Rhapsody UML, VIM or emacs, or offer integration to collaboration tools, such as Hudson or Jenkins. CPPcheck and Flawfinder allow a. export of results to HTML and Coverity Scan and KlocWork provide APIs.

[2]Coverity Scan https://scan.coverity.com/users/sign_in

[3]KlocWork https://support.roguewave.com/documentation/klocwork/en/10-x/whichtypeofcheckertocreatekastorpath/

[4]PolySpace https://se.mathworks.com/discovery/static-code-analysis.html

[5]Splint http://www.splint.org/

[6]CPPcheck http://cppcheck.sourceforge.net/

[7]Flawfinder https://www.dwheeler.com/flawfinder/

**Table 1** Comparison between popular static analysis tools and code style checkers

| Tool | Supported Languages | Compilation Requirements | Extensibility | Configurability | Interoperability | Access to Results |
|---|---|---|---|---|---|---|
| Static Code Analysis Tools | | | | | | |
| *Coverity Scan* | C/C++ | parsing/ (linking) | imperative | N/A | stand-alone/ IDE/ collab. tools | API |
| *KlocWork* | C/C++ | parsing/ (linking) | imperative | rules by parameter/ ruleset | stand-alone/ IDE/ collab. tools | API |
| *PolySpace* | C/C++ | parsing/ (linking) | N/A | ruleset | stand-alone/ IDE | N/A |
| *Splint* | C | parsing/linking | declarative by rule | N/A | stand-alone | no external |
| *CPPcheck* | C/C++ | parsing/linking | declarative by rule | other | stand-alone/ IDE/ collab. tools | file-export |
| *Flawfinder* | C/C++ | robust | declarative by rule | other | stand-alone/ IDE | file-export |
| Style Checkers | | | | | | |
| *CodeCheck* | C/C++ | parsing | imperative | N/A | stand-alone | file-export |
| *Uncrustify* | C/C++ | parsing | no | rules by parameter | stand-alone/ IDE | file-export (config.) |
| *KWStyle* | C/C++ | parsing/linking | no | rules by parameter | stand-alone/ collab. tool | N/A |
| *C++ Style Checker* | C++ | parsing/linking | N/A | ruleset | stand-alone | file-export |
| Learning/Example-Based Approaches | | | | | | |
| *Naturalize* | C/C++ | parsing/linking | no | rules by example | stand-alone/ IDE/ collab. tools | N/A |
| *Code Style Analytics* | N/A | parsing/linking | no | rules by example | N/A | N/A |
| *CCFlex* | C/C++ | robust | declarative by example | ruleset | stand-alone / collab. tools | file-export |

**Style Checkers for C/C++** In contrast to static code analysis tools, style checkers mainly focus on aspects of the code that do not directly impact the behavior of the system. Nonetheless, having a consistent style can significantly improve readability of the code and with that maintainability and quality. In the following, we discuss for well known examples for Style Checkers that target C or C++: CodeCheck[8], Uncrustify[9], KWStyle[10], and C++ Style Checker Tool[11].

CodeCheck and Uncrustify both parse the code, while KWStyle and C++ Style Checker Tool also require the code to be linked. Only CodeCheck can be extended by writing new checkers. The checks of Uncrustify and KWstyle cannot be extended without extending the tools themselves. However, both tools allow configuring single rules, e.g. by specifying the allowed length of a code line. The C++ Style Checker Tool enables a selection of rules/rule-sets to be applied.

Uncrustify has a plug-in for the UniversalIndentGUI and exports rule configurations. KWStyle can be integrated into Git. Finally, CodeCheck and C++ Style Checker Tool both allow to export results, e.g. into HTML or XML.

### 2.2 Machine Learning for Static Code Analysis

There are numerous studies applying ML algorithms to evaluate code quality. However, according to our best knowledge, none of them solely focus on recognizing company-specific code guidelines violations in code using examples. Also, most of these tools are at the early stages of development and are not ready to be integrated into industrial code-review pipelines.

Among these studies, there are some that localize defects at the level of code lines or statements. For instance, Brun and Ernst (2004) proposed a fault invariant classifier that uses a dynamic (runtime) analysis to extract semantic properties of the program's computation as features describing the code.

Axelsson et al. (2009) proposed to use Normalised Compression Distance to find potential string overflows, null pointer references, memory leaks, and incorrect API usage. Chappelly et al. (2017) reported findings on using machine learning techniques to detect defects in C programs at Oracle. They used neural networks and a large corpus of programs to compare the prediction quality of ML-based classifier against four static program analysis tools, including Parfait used internally at Oracle. Their conclusion was that the ML-based tools were not suitable replacements for static program analysis tools due to the low precision of the results.

Allamanis et al. (2014) and (Torunski et al. 2017), present with Naturalize[12] and a code style analytics tool, two approaches that adapt style checking to a code-base by configuring rules according to what is typical for that codebase.

Torunski et al. (2017) bases the configuration on an assessment of metrics, such as method lengths, number of CamelCase variables, or number of opening braces on the same line for loops.

[8]CodeCheck http://www.abxsoft.com/

[9]Uncrustify http://uncrustify.sourceforge.net/

[10]KWStyle https://kitware.github.io/KWStyle/

[11]C++ Style Checker Tool http://www.semdesigns.com/Products/StyleChecker/CppStyleChecker.html?Home=StyleChecker

[12]Naturalize http://groups.inf.ed.ac.uk/naturalize/

Recently, Mi et al. (2018) studied the possibility of classifying source code depending on its readability. They used three Convolution Neural Networks operating on different levels of granularity (character-level, token-level, and AST-Tree nodes level). The accuracy of the proposed DeepCRM+ConvNets method was evaluated on the open source code since training the networks required large amounts of labeled data.

Finally, the tool we use in this study is called Flexible Code Counter/Classifier (CCFlex). It was developed by the authors of this paper and used to perform software size measurement in one of the previous studies (Ochodek et al. 2017). However, since the tool is a general-purpose code classifier, it could be adapted to detect violations of coding-style guidelines.

### 2.3 Machine Learning for Code-Smell Detection

Fontana et al. (2016) and Fontana et al. (2013) presented an experiment to compare how good different machine learning algorithms would be in learning to detect code smells. While Fontana et. al. report on reaching an accuracy of 95%, Di Nucci et al. (2018) challenged these results with a replication study. The replication shows that the selection of the dataset including the balance of violations with none-violations as well as an unrealistic characteristic of violation cases can lead to an up to 90% too high precision.

Although the aforementioned studies use the same classification algorithms as we do in our study (e.g., decision trees, random forest), they differ in the approach to feature extraction. The studies on code-smell detection use code metrics to describing units of code (e.g., metrics of size, complexity, cohesion, coupling, encapsulation, or inheritance), while in this study, we extract linguistic features (e.g. number of "if" statements, which words/variables/statements were used in a line) directly from the source code in a similar way as it is done in the field of natural language processing (e.g., by using the bag-of-words model).

Also, code smells are typically concerning architectural aspects and larger chunks of code than coding guideline violations. However, it is nonetheless interesting to compare progress in applying machine learning in the two fields. We especially expect the relevance of realistic data in training and evaluation to be valid for machine learning of coding guideline violations.

### 2.4 Summary

As shown in Table 1, most existing static code analysis tools and style checkers need to parse the code and often even link it to fully apply their analysis. This also holds for most of the ML-based approaches. The only exceptions are Flawfinder and *CCFlex*, which are both robust, due to token- or text-based analysis.

Extensibility is never trivial. Even when a declaration of rules is allowed, there is a steep learning curve to understand the declaration languages. CCFlex and DeepCRM+ConvNets provide an alternative approach, allowing developers to define extensions, by listing examples in the language they usually work in.

The other ML-based approaches are limited with regards to the type of violations that can be found. This is something that also holds for Flawfinder for which the specialization on security seems to limit the type of extensions possible.

Finally, most of the ML-based tools (e.g., DeepCRM+ConvNets) are experimental/research tools at the early stages of development. Therefore, they lack documentation and usability features what makes them hard to use by practitioners.

## 3 The CCFlex Tool

The CCFlex tool (Flexible Code Counter/Classifier) was initially designed to perform software size measurement (Ochodek et al. 2017). In this study, we adapted the tool to detect code guidelines violations in the code. Since the tool evolved over the course of the study, here, we describe its final design to give the reader a complete overview of how the tool operates. The tool is distributed as Open Source and available on GitHub.[13]

### 3.1 Architecture

The version of CCFlex used in the previous study was implemented as a monolithic Java application. In the course of this study, we decided to redesign the tool so it is possible to change the processing pipeline without the need for recompiling or redeploying the tool. We used the pipes-and-filters architecture style, where we use a number of independent components (filters) that could be organized into processing pipelines. We also switched to the Python ML technological stack to implement the tool. However, filters can be implemented in any programming languages as long as they accept input and produce output in the agreed format. The total size of the Python code is around 5K source lines of code (SLOC) with the average size of a filter of ca. 140 SLOC. The full documentation of the filters is available on the project's GitHub page.

A typical processing pipeline used in this study is presented in Fig. 1. Before the training begins, we need to prepare a training codebase with labels. The lines violating coding guidelines are labeled by adding a configurable prefix to each of them. For instance, if we configured the prefix to be "`@!`", then the line "`@! int MyVAR = 10;`" would be recognized as a line violating the coding guideline.

The first filter used in the pipeline presented in the figure is called *lines extractor*. It traverses through a codebase, extracts each line, its class label, and stores the results in a CSV file. The output file can be passed any of the available feature extractors. Each extractor adds column(s) (extracted features describing each of the lines) to the CSV file. The output files produced by feature extractors can be merged using specialized filters (they are omitted in Fig. 1 for the sake of brevity). On top of that, we can use *feature selection* filters to reduce the dimensionality of the feature space using the algorithms available in Python sklearn library. Features selection is performed on training codebase and then also applied to code that is going to be evaluated by dropping rejected features. Finally, the produced CSV file can be loaded by one of the filters that *train a classifier* and use it to classify the new lines. The results are stored in an output CSV file and can be passed to one of the report-generation filters.

In addition to the filters, CCFlex provides a suite of standalone tools allowing to perform independent tasks, like the selection of lines to label using Active Learning (AL) (Fu et al. 2013) or checking the consistency of line labels provided by a human (i.e., identifying lines having similar representation in the feature space but different labels).

### 3.2 Feature-Extraction Filters

The feature extractors implemented in CCFlex can be divided into three categories: predefined features extractors, vocabulary-based extractor, and block-feature extractors.

[13] CCFlex https://github.com/mochodek/py-ccflex

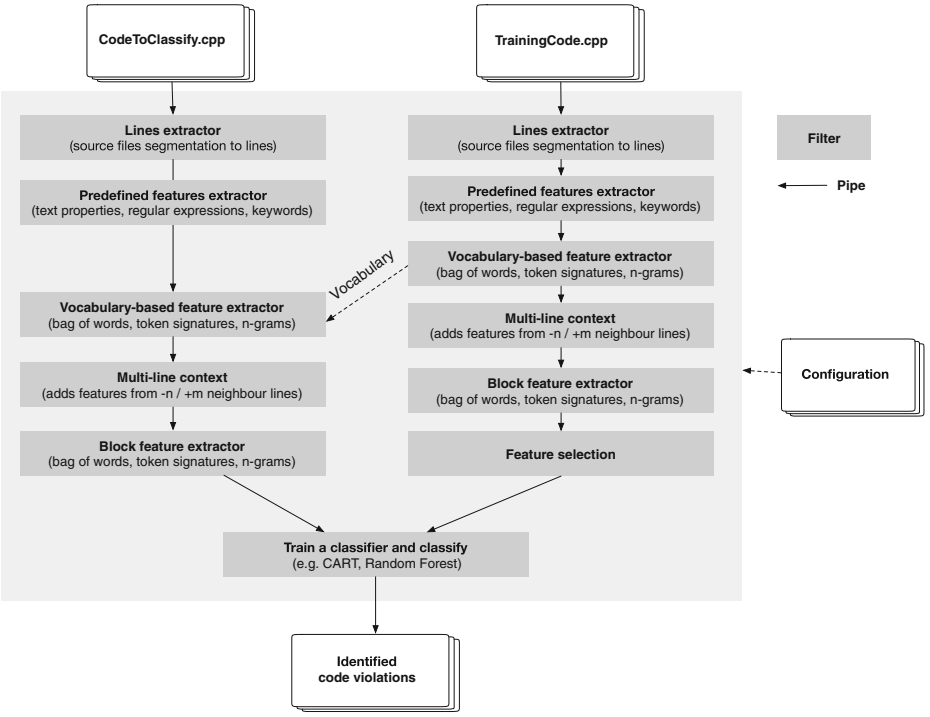



**Fig. 1** An example of a typical CCFlex processing pipeline

**Predefined features extractors** process lines of code and extract the features defined by the user, e.g., counts occurrences of a given substring in a line or number of characters or words. For instance, one could configure a filter to count the number of string "for" in a line and extract it as a feature.

**Vocabulary-based extractors** use the bag-of-words (BOW) model to extract features. Bag-of-words uses a vocabulary that can be either automatically extracted from the training examples or predefined (CCFlex supports both approaches). When the vocabulary is extracted from the training code it has to be passed as an input to the filter extracting features using BOW on the code to be evaluated (see Fig. 1). BOW counts the occurrences of tokens in the code that are in the vocabulary (the code is tokenized). Also, it can count occurrences of sequences of tokens called n-grams (e.g., bi-gram or tri-grams). N-grams are a valuable source of information for finding code guidelines violations since it is often important to understand the context in which a given token appears (e.g., `int a` vs. `class a`).

We introduced several modifications to a commonly used bag-of-words extraction algorithm. Firstly, we introduced an alternative tokenizer that split each line to tokens using not only white but also special characters: ()[]{}!@#$%^&*-=;:'"\|`~,.<>/?. The split strings are also preserved as tokens since the represent constructs used by many programming languages. Another difference is that we convert each token to what we call *token signature* before creating a vocabulary. This allows reducing the size of vocabulary, and consequently, the number of features and help in preventing overfitting to specific names of variables or methods while training.

To create a token signature, we firstly replace each uppercase letter with "A", each lowercase letter with "a", and each digit with "0", while special characters are left unchanged (e.g., "_"). Then, each subsequence of the same characters is shrunk to a single character only (e.g., *aaa* to *a* or __ to _). The same is repeated for pairs and triples of characters (e.g., *AaAa* is converted to *Aa*). An example of how a token signature and bag-of-words model are constructed for a line is presented in Fig. 2.

**Block-feature extractors** use heuristic approaches that allows identifying features spanning through multiple lines. We have two variants of such tools. The first one is based on the already extracted features. Each block feature is defined by providing the features that need to be present in a line (their values have to be greater than zero) to treat the line as *start* or *end* of the block. Similarly, it is possible to define forbidding features. If any of the forbidding features are present in a line (its value is greater than zero) the line cannot be

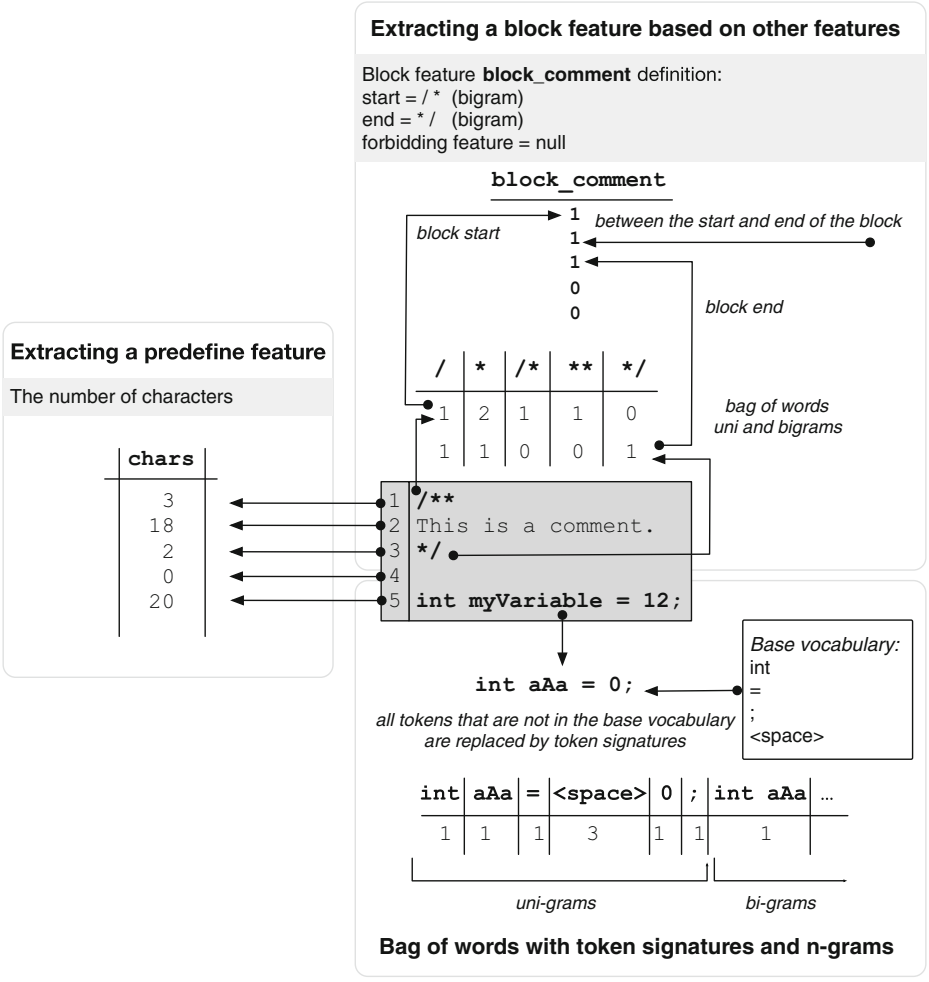



**Fig. 2** An example of extracting bag-of-words (with and example of generating token signature), block feature, and a predefined feature for a fragment of code

considered as a start or end of the block. All the lines between the start and end of a block belong to that block (the value of the block feature is set to one). An example of extracting a block-comment feature is presented in Fig. 2. The second variant of block-feature extraction is based on training and using a separate classifier to identify the starts and ends of block features.

### 3.3 Classification Algorithms

While designing ML-based filters, we took into account two observations made while analyzing the coding guidelines and discussing them with our partners. Firstly, we had to accept the fact that the size of a training sample would be very small since the code guidelines used by our industrial partners included only a limited number of examples (additional examples would have to be provided by the users). Secondly, the traceability of the decision made would be welcomed. Therefore, we preferred to use ML-algorithms that provide explanations of how the decision is made. We decided to use decision trees since they are commonly considered as interpretable models (Freitas 2014). Consequently, we have integrated into CCFlex and used in this study a set of decision-tree-based algorithms, such as CART, C5.0 Decision Trees, and Random Forest.

### 3.4 Active Learning

Active learning (AL) aims at training an accurate prediction model with minimum cost by labeling most informative instances (Fu et al. 2013). In essence, AL is an iterative and interactive process of performing two steps: *measuring the uncertainty of classification* and using this information to *query* unlabeled instances.

There are many ways to measure uncertainty and strategies to query for labels. In our study, we use the query by committee (QBC) strategy (the implementation available in the modAL[14] Python library using committee of decision trees (CART), K-Nearest Neighbours (KNN), and Random Forests as base learners. In the QBC strategy, when an instance is queried, each member of the committee votes on the class label of the instance. The final predictions are the majority voting of the members. The most informative instance is the one with the most disagreement in the prediction of the committee classifiers.

The AL tool available in CCFlex allows for interactive labeling of code with the use of the selected querying strategy (either QBC or single classifier uncertainty sampling). The user is presented with a line of code in its context (a few proceeding and following lines) and asked to label the line (decide whether the line violates a coding guideline or not). Then, the line is added to the training codebase and another query is performed to select the next line to be labeled.

When using this tool in our study, we always started from creating an initial training dataset by adding positive and negative examples available in the companies' coding guidebooks and then used the QBC querying strategy to poll more lines for labeling from the codebase.

## 4 Research Methodology and Design

The study of the violations of the coding guidelines needs to be conducted in the industrial context, where the interaction between the researchers and the studied organization stim-

[14] modAL https://github.com/cosmic-cortex/modAL.

ulates learning for both sides. For the theory development (research), the learning of the practical side of constructing coding/design guidelines, recognizing the violations and the acceptance of accuracy are important. For the application (industry), the learning of how the limitations and their impact on how to check guidelines automatically are important. For this kind of context, action research is the most suitable research methodology, as it emphasizes the learning and theory development from empirical observations in the industrial context (Baskerville and Wood-Harper 1996).

We use the opportunity to work closely with two industrial partners, which provided the context of our research and allowed us to work on their premises, with their codebase and software engineers. Both companies were present in discussing the results of all action-research cycles and deeply involved in the cycles when the researchers worked on their premises.

Company A is a leading provider of consumer products in the embedded software domain. The company has a tradition of collaboration with software engineering researchers and being part of action research projects. The company allowed us to analyze their platform code used in a large number of their products and provided access to a software architect, and three designers which have formulated the rules which have been as learning input—all with over 10 years experience in the domain.

Company B is a leading provider of infrastructure products in the embedded software domain. The company has a long tradition of collaborating with software engineering researchers and over ten years of experience in running action research projects. The company provided us with the access to their source code and collaboration with two architects with over 20 years of experience with the company's products and with design/code quality.

Both companies are involved in the development of the tool as their operations moved towards continuous deployment and their intention is to increase the automation of software development by increasing the use of machine learning and autonomous computing techniques for software engineering tasks. The intention is to find methods to identify violations of their proprietary guidelines with as little manual work as possible.

In Table 2, we summarize each of the action research cycles. The details of the results are presented in Section 5.

As the essence of action research projects is that they combine the learning with the design of the study, thus being a flexible research design (Robson and McCartan 2016), we detail the choices we made for each cycle in the following section, including the results from each of the action research steps.

## 5 Execution and Results

In this section, we present the evaluating and learning activities from the action research cycles.

### 5.1 Action Research Cycle 1 – What Coding Guidelines are used by our Industrial Partners?

#### 5.1.1 Cycle Goal and Research Procedure

The goal of the cycle was to investigate the coding guidelines of our industrial partners. In particular, we wanted to:

**Table 2** Summary of action research cycles

| Cycle | Diagnosing | Action planning | Action taking / Executing | Evaluating | Learning |
|---|---|---|---|---|---|
| 1 | What coding guidelines are used in the industry? | We planned to conduct a document analysis at partner companies to understand how they design their coding guidelines and the content of these guidelines. | We changed the way of grouping of guidelines—from thematic to scope-based; we analyzed 45 and 66 guidelines and classified them according to this new grouping. | We found that some guidelines need quality improvement because of their ambiguity or difficulty to make the assessment. | Most of the coding guidelines are different between the companies, which means that the potential tool has to be adapted and tuned on a per-company basis. |
| 2 | Which of available tools can be adapted to recognize violations of code guidelines assuming that code might not parse or compile? | We planned to review the most popular tools for C/C++ code analysis and compare extendability, readability and usability of the tools to find a tool that is easy to extend, does not require parsing or compiling the code, and can recognize code guidelines violations in code. | We reviewed 13 tools for C/C++ code analysis. We refactored CCFlex (a machine-learning-based tool) and performed a benchmark study on a problem of finding violations of popular Sun and Google coding conventions for Java by comparing on 3 open source projects. | CCFlex could recognize 98.98%–99.93% of lines violating *any* of the Java Sun's and Google's coding style guidelines | Since we could recognize almost all of the lines with violations, we set off to check the guidelines from the industrial partners. |
| 3 | What is the accuracy of the recognition of the violations of guidelines in the industrial settings and how it depends on coding style? | We planned to assess different situations: old/legacy codebase (before the guidelines), modern codebase (currently under development) and in-between (code which was developed alongside the development of the guidelines). | We analyzed three codebases of Company A; we assessed 3 out of 45 guidelines; using different configurations of CCFlex and a limited number of iterations. | We could achieve satisfactory results of up to 87% Recall for the evaluated guidelines. | We needed to conduct a formal evaluation with Company B to first evaluate the generalizability of the findings and secondly investigate if we are able to reduce the number of false-positives. |

**Table 2** (continued)

| Cycle | Diagnosing | Action planning | Action taking / Executing | Evaluating | Learning |
|---|---|---|---|---|---|
| 4 | How much training of CCFlex is needed to minimize the percentage of false-positives? | We selected one or two guidelines per type of rule (taxonomy) and defined a procedure on how to sample code for training. | We conducted the assessment of seven rules at Company A and used two different strategies for selecting lines in the training set—manual selection and Active Learning; we assessed the code quality of a product. We evaluated the approach on a large codebase of over three million SLOC. The number of trials varied from three to seven and the F-score varied from 0.04 to 1.00 | We found that the identified violations were accurate and the architects used them in their quality improvement after the study. | Active learning allows to achieve higher F-score in fewer iterations and provides higher F-score for the entire codebase. |

– understand the types of rules that are in the companies' guidebooks,
– check the quality of the rules (whether they are unambiguous and their violations can be found by analyzing code).

We investigated rules in the Company A and B guidebooks and categorize them based on the information required to identify their violations. We assessed the quality of the rules and consulted our findings with software engineers from both companies.

### 5.1.2 Cycle Execution and Results

After reviewing the literature, we learned that there is no agreed taxonomy that could be used to categorize coding guidelines (and their violations). For instance, Novak et al. (2010) compared four static code analysis tools and showed that each of the tools uses a different set of categories, and even within a single taxonomy multiple criteria are often used to define these categories. For instance, some categories refer to syntax and programming concepts (e.g., naming conventions, code layout, exceptions handling) while others are defined by referring to quality attributes of software products that could be affected by certain violations (e.g., maintainability, security, or performance problems).

Since we wanted to automatically identify lines of code violating coding guidelines, we proposed a different taxonomy that categorizes violations based on the information that is required to recognize them in the code. The taxonomy is presented in Fig. 3. It groups guidelines into three main categories.

The first root category groups "semantical" coding guidelines. Finding violations of such rules requires understanding the meaning of a text in its context. Depending on the size of the context, we distinguish four sub-categories: a uni-line context—we need to understand the meaning of the tokens/words in a single line (e.g., the rule stating that there can

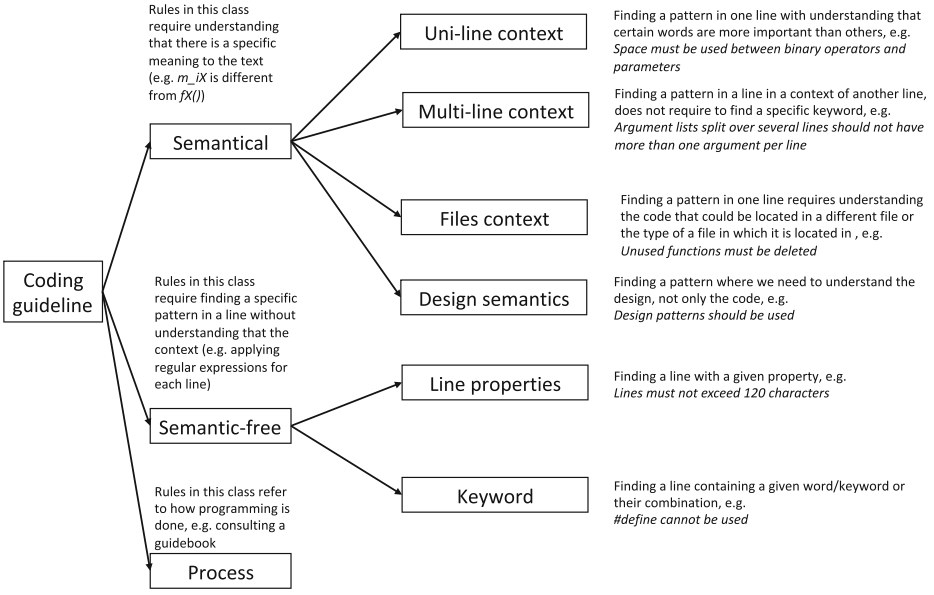



**Fig. 3** Taxonomy of code guidelines violations

be only one statement in a line), multi-line context—we need to understand the meaning of words/tokens in a sequence of lines (e.g., braces must be used for all compound statements), files context—we need to be able to relate the code in different files or understand file properties (e.g., unused functions must be deleted), and design context which goes beyond understanding code constructs and requiring recognizing the design intent (e.g., design patterns should be used). For uni-line context we do not take into account the possibility of intentionally breaking the line in the middle of statement.

The second root category groups "semantic-free" coding guidelines. Finding violations of such rules requires recognizing the presence of some patterns in a line of code (without the need for understanding the role or meaning of particular tokens or words). We distinguish two sub-categories: line properties—a guideline refers to a quantifiable property of text (e.g., the number of characters in a line shall not exceed 120 characters) and keyword-based guidelines—we need to find a keyword or combination of keywords (or lack of such) in the text (e.g., union types shall not be used—we need to find the keyword `union`).

The remaining root category is called "process" coding guidelines. The guidelines belonging to this category regards the development process (e.g., the necessity of consulting a guidebook).

We used the proposed taxonomy to categorize the coding guidelines of our industrial partners. The guidebook provided by Company A contained 45 coding guidelines while the one used by Company B included 66 rules. The distribution of the rules between the categories of our taxonomy is presented in Fig. 4.

When classifying the rules into the categories, we identified three groups of outlying rules:

– *rules as documentation*: no style-related coding guidelines, but rather hints about what libraries or interfaces to use or what protocols to follow when calling an interface,
– *optional rules*: either a whole rule or its part is optional to follow,

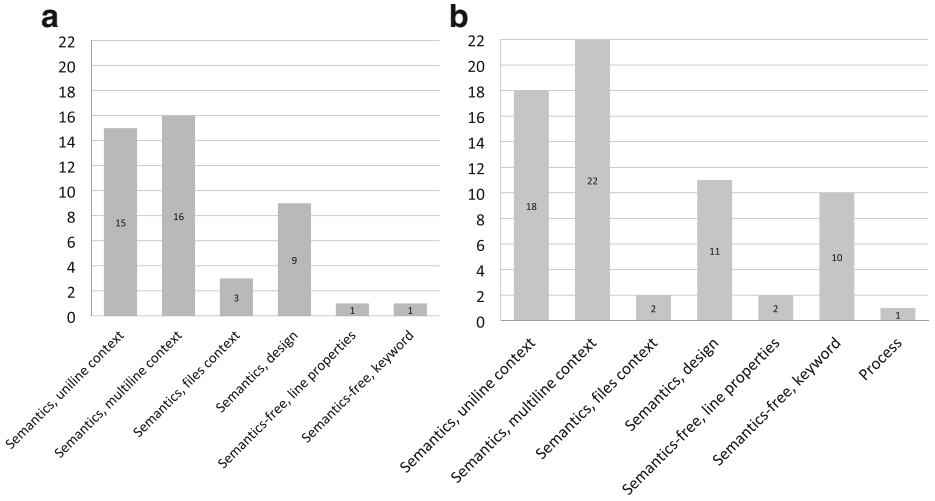



**Fig. 4** The number of coding-guideline rules identified for a) Company A and b) Company B

– *rules on external information*: rules that require information outside of the studied code, e.g. user requirements.

Rules that serve as documentation and rules on external information might be extremely difficult to identify and certainly have special needs to the static code analysis approach, such as consideration of multiple files at once. On the other hand, optional rules might as well be ignored by any approach. Therefore, we defined these three types of rules to be out of scope for the further investigation.

We also observed that some rules were imprecise and could have different interpretations. We consulted each of such rules with the companies, and as a consequence, some rules were indicated as to be improved.

The performed analysis resulted in narrowing our study to rules belonging to four categories: semantical uni-/multi-line context and semantic-free line properties and keywords. We observed that the rules belonging to the three remaining categories were either not related to code (process rules), violations could not be mapped to particular lines (design semantics, e.g., usage of design patterns), examples were not available in the guidebook and would be very difficult to get (design semantics) or span through multiple source files (semantical files context, e.g., unused functions should be deleted).

## 5.2 Action Research Cycle 2 – Selecting a Tool Capable of Recognizing Code Guidelines Violations of our Partners

### 5.2.1 Cycle Goal and Research Procedure

The goal of the second cycle of our action research study was to find a tool capable of recognizing violations of our partners' coding guidelines while also meeting the quality attribute requirements they perceived as important, namely:

– the proposed solution needs to be easy to extend or modify without the need of learning any API or having a deep understanding of static code analysis techniques,
– running the code analysis should not require parsing or compiling the code.

The rationale behind the first requirement is that the code guidelines are specific to each company and can evolve or change in time. Therefore, the tool needs to be easy and cost-effective to maintain. The second requirement is motivated by the fact that some of the code of our partners might not compile when taken outside of the runtime environment. Also, at some stage, it is expected that the tool could be integrated into code-reviewing tools such as Gerrit and be able to recognize the violations in fragments of code being reviewed that might not parse as well (e.g., in form of Git diffs).

We planned to review the tools for C/C++ code analysis and ML-based tools. Based on the results of the analysis we wanted to select a tool to be used in the next cycles of the study.

We then planned to perform a simulation study to preliminarily evaluate the accuracy of the tool in finding lines violating two well-known coding standards—Google Java Style Guide[15] and Sun Java Code Conventions[16] in the code of open-source projects. Since it was supposed to be a feasibility study, we wanted to consider a simpler but similar problem to the

[15]Google Java Style Guide http://checkstyle.sourceforge.net/reports/google-java-style-20170228.html

[16]Code Conventions for the Java Programming Language: Contents http://www.oracle.com/technetwork/java/javase/documentation/codeconvtoc-136057.html

one of recognizing code guidelines violations in the code of our partners. Java programs are generally easier to parse than programs written in C++, which is considered an extremely difficult language to parse (Irwin and Churcher 2001). Secondly, both coding standards focus on coding style/formatting issues (similarly to many of the rules of our partners) and cover all semantical and semantic-free subcategories of our taxonomy (see Tables 3 and 4). We assumed that if the tool cannot effectively recognize violations of guidelines in such settings, its tuning would not likely lead us to achieve satisfactory results for our partners' code and rules.

### 5.2.2 Cycle Execution and Results

Our comparison of the existing tools in Section 2 showed, that there are only two tools that are robust with regards to the defined quality requirements, i.e., Flawfinder and CCFlex. However, Flawfinder's is specialized to recognize security-related problems and its extensibility is limited to similar types of issues (still, extending the tool requires understanding and modifying a built-in database of patterns). On contrary, CCFlex allows extending the tool to support a new rule by providing examples of lines following and violating the rule. Therefore, it does not require learning any API or implementing any algorithms. However, the original tool was designed to solve a different problem of recognizing lines to be counted while measuring the physical size of software products. Therefore, we decided to first investigate if the CCFlex tool can be effectively used to recognize code guidelines violations before making the final decision of presenting it to industrial partners and adapting it to recognize violations of their coding guidelines.

As a feasibility study, we investigated how effective the CCFlex tool in recognizing lines of code violating coding guidelines depending on the number of labeled lines used for training. We assumed that when using an ML-based tool, such as CCFlex, instead of a static code analysis tools there could be a trade-off between the accuracy of the tool and its extensibility. We took the CCFlex tool and used Checkstyle[17] to find lines violating Google Java Style Guide and Sun Java Code Conventions.

We performed validation on a dataset containing samples of code from three Java open-source projects: Eclipse Platform, Jasper Reports, and Spring Framework. For each project, we randomly sampled source code files that in total consisted of around 15K SLOC (Eclipse Platform 15,588 SLOC, Jasper Reports 14,845, and Spring Framework 14,869 SLOC). The whole datasets contained 45,302 SLOC. We perceived the code of Eclipse Platform and Spring Framework as being well documented and convergent with the coding standards for Java. Contrary, the code of Jasper Reports seemed to violate the basic rules of the conventions (e.g., placing an opening curly bracket of a compound statement in a new line).

The Checkstyle tool was used as an oracle to automatically label the lines that were violating code-style *any* of the rules proposed by each of these standards. As a result, we initially constructed eight datasets:

- All–Sun (Count: 10,821, Ignore: 34,481)
- Eclipse-Sun (Count: 3,003, Ignore: 12,585)
- Jasper-Sun (Count: 5,046, Ignore: 9,799)
- Spring-Sun (Count: 2,772, Ignore: 12,097)

[17] Checkstyle: http://checkstyle.sourceforge.net.

**Table 3** The percentage of lines violating Sun Java Coding Conventions (Checkstyle)

| Violation of the Sun's guidelines | All% | Eclipse% | Jasper% | Spring% |
|---|---|---|---|---|
| Line is longer than 80 characters (Line properties) | 40.70 | 55.04 | 20.13 | 62.59 |
| Parameter should be final (Uni-line context) | 19.41 | 26.37 | 11.79 | 25.72 |
| '{' should be on the previous line (Multi-line context) | 13.67 | | 29.31 | |
| Missing a Javadoc comment (Multi-line context) | 10.67 | 12.52 | 9.77 | 10.32 |
| Class designed for extension without Javadoc (Design semantics) | 9.03 | 6.93 | 12.88 | 4.29 |
| Line has trailing spaces (Uni-line context) | 8.78 | | 18.83 | |
| Expected @param tag (Multi-line context) | 5.73 | 9.26 | 2.93 | 7.00 |
| Hidden field (Files context) | 3.90 | 2.73 | 3.03 | 6.75 |
| Variable must be private and have accessor methods (Multi-line context) | 2.73 | 1.76 | 4.68 | 0.22 |
| Symbol is not followed by whitespace (Uni-line context) | 2.50 | 2.30 | 3.94 | 0.07 |
| File contains tab characters (this is the first instance) (Keyword) | 2.40 | 3.40 | 1.35 | 3.25 |
| Expected an @return tag (Multi-line context) | 2.27 | 0.07 | 1.45 | 6.17 |
| '}' should be on the same line as the next part of a multi-block statement (Multi-line context) | 1.72 | 0.33 | 1.01 | 4.51 |
| Avoid inline conditionals (Uni-line context) | 1.45 | 2.36 | 0.55 | 2.09 |
| 'if' construct must use '{}'s (Multi-line context) | 0.85 | 3.06 | | |
| Expected @throws tag (Multi-line context) | 0.69 | | 0.87 | 1.12 |
| First sentence should end with a period (Multi-line context) | 0.61 | 1.20 | 0.02 | 1.05 |
| Symbol should be on a new line (Multi-line context) | 0.52 | | 0.10 | 1.84 |
| Symbol is followed by whitespace (Uni-line context) | 0.50 | 0.87 | 0.55 | |
| Name must match pattern '^[A-Z][A-Z0-9]*(_[A-Z0-9]+)*$' (Uni-line context) | 0.46 | 0.67 | 0.42 | 0.32 |
| Redundant 'final' modifier (Multi-line context) | 0.34 | 1.23 | | |
| Symbol is not preceded with whitespace (Uni-line context) | 0.34 | 0.77 | 0.26 | 0.04 |
| Unused import (Multi-line context) | 0.34 | 0.20 | 0.10 | 0.94 |
| Redundant 'public' modifier (Multi-line context) | 0.30 | 0.60 | 0.18 | 0.18 |

**Table 3** (continued)

| Violation of the Sun's guidelines | All% | Eclipse% | Jasper% | Spring% |
|---|---|---|---|---|
| Symbol is preceded with whitespace (Uni-line context) | 0.29 | 0.03 | 0.57 | 0.04 |
| Magic number (Uni-line context) | 0.20 | 0.47 | 0.06 | 0.18 |
| 'for' construct must use '{}'s (Multi-line context) | 0.14 | 0.50 | | |
| 'static' modifier out of order with the JLS suggestions (Uni-line context) | 0.10 | 0.27 | 0.04 | 0.04 |
| File does not end with a newline (Multi-line context) | 0.10 | 0.30 | 0.04 | |
| Class should be declared as final (Uni-line context) | 0.08 | 0.30 | | |
| Avoid nested blocks (Multi-line context) | 0.07 | 0.03 | 0.14 | |
| Extra HTML tag found (Multi-line context) | 0.06 | 0.17 | | 0.07 |
| Utility classes should not have a public or default constructor (Design semantics) | 0.05 | 0.17 | | |
| Unused @param tag (Multi-line context) | 0.04 | 0.13 | | |
| Inner assignments should be avoided (Multi-line context) | 0.03 | 0.10 | | |
| Unclosed HTML tag found (Multi-line context) | 0.03 | 0.03 | | 0.07 |
| Unknown tag (Uni-line context) | 0.03 | 0.10 | | |
| 'else' construct must use '{}'s (Multi-line context) | 0.02 | 0.07 | | |
| Expression can be simplified (Multi-line context) | 0.02 | 0.07 | | |
| Method length greater than 150 (Multi-line context) | 0.02 | 0.07 | | |
| Unable to get class information for @throws tag (Checkstyle error) | 0.02 | | | 0.07 |
| 'protected' modifier out of order with the JLS suggestions (Uni-line context) | 0.01 | 0.03 | | |
| 'public' modifier out of order with the JLS suggestions (Uni-line context) | 0.01 | 0.03 | | |
| Array brackets at illegal position (Uni-line context) | 0.01 | 0.03 | | |
| Comment matches to-do format 'TODO:' (Multi-line context) | 0.01 | 0.03 | | |
| Redundant 'private' modifier (Multi-line context) | 0.01 | | 0.02 | |
| Switch without "default" clause (Multi-line context) | 0.01 | | 0.02 | |

**Table 4** The percentage of the lines violating Google Java Style Guide (Checkstyle)

| Violation of the Google Java style | All% | Eclipse% | Jasper% | Spring% |
|---|---|---|---|---|
| Line contains a tab character (Keyword) | 98.83 | 99.42 | 97.91 | 99.16 |
| Incorrect indentation level (Multi-line context) | 59.48 | 55.05 | 69.01 | 53.82 |
| '{' should be on the previous line (Multi-line context) | 4.81 | | 14.02 | |
| Line is longer than 100 characters (Line properties) | 3.83 | 3.47 | 2.66 | 5.66 |
| First sentence of Javadoc is incomplete (period is missing) or not present (Multi-line context) | 1.36 | 1.18 | 2.36 | 0.40 |
| At-clause should have a non-empty description (Uni-line context) | 1.03 | 2.43 | 0.41 | 0.01 |
| <p> tag should be preceded with an empty line (Multi-line context) | 0.75 | 0.18 | 0.08 | 2.24 |
| '}' should be on the same line as the next part of a multi-block statement (Multi-line context) | 0.61 | 0.09 | 0.48 | 1.39 |
| 'package' should be separated from previous statement (Uni-line context) | 0.31 | 0.25 | 0.63 | 0.01 |
| 'if' construct must use '{}'s (Multi-line context) | 0.30 | 0.82 | | |
| Whitespace around a symbol is not followed by whitespace (Uni-line context) | 0.28 | 0.46 | 0.33 | |
| Abbreviation in name must contain no more than '2' consecutive capital letters (Uni-line context) | 0.22 | 0.04 | 0.60 | |
| Missing a Javadoc comment (Multi-line context) | 0.18 | 0.11 | 0.40 | 0.02 |
| Symbol should be on a new line (Uni-line context) | 0.18 | | 0.05 | 0.55 |
| Wrong lexicographical order for import (Uni-line context) | 0.16 | 0.14 | 0.22 | 0.11 |
| Whitespace around a symbol is not preceded by whitespace (Uni-line context) | 0.12 | 0.21 | 0.12 | 0.01 |
| Member name must match pattern '^[a-z][a-z0-9][a-zA-Z0-9]*$' (Uni-line context) | 0.11 | 0.12 | 0.19 | |
| <p> tag should be placed immediately before the first word. with no space after (Uni-line context) | 0.10 | 0.20 | 0.09 | |
| ')' is preceded with whitespace (Uni-line context) | 0.10 | 0.01 | 0.27 | |
| '(' is followed by whitespace (Uni-line context) | 0.09 | 0.01 | 0.27 | |
| 'for' construct must use ''s (Multi-line context) | 0.05 | 0.13 | | |

**Table 4** (continued)

| Violation of the Google Java style | All% | Eclipse% | Jasper% | Spring% |
|---|---|---|---|---|
| 'static' modifier out of order with the JLS suggestions (Uni-line context) | 0.04 | 0.07 | 0.02 | 0.01 |
| Empty line should be followed by <p¿ tag on the next line (Multi-line context) | 0.04 | 0.04 | 0.04 | 0.03 |
| Overload methods should not be split (Multi-line context) | 0.04 | 0.01 | 0.08 | 0.02 |
| Empty catch block (Multi-line context) | 0.03 | 0.07 | 0.01 | |
| Javadoc comment has parse error (Multi-line context) | 0.02 | 0.04 | | 0.03 |
| At-clauses have to appear in the order '[@param. @return. @throws. @deprecated]' (Multi-line context) | 0.02 | 0.04 | | 0.01 |
| Distance between variable declaration and its first usage is more than '3' (Multi-line context) | 0.02 | 0.04 | | |
| 'METHOD_DEF' should be separated from previous statement (Multi-line context) | 0.01 | | | 0.04 |
| Single-line Javadoc comment should be multi-line (Uni-line context) | 0.01 | | | 0.04 |
| Local variable name must match pattern '^[a-z]([a-z0-9][a-zA-Z0-9]*)?$' (Multi-line context) | 0.01 | 0.03 | | |
| 'else' construct must use '{}'s (Multi-line context) | 0.01 | 0.02 | | |
| Each variable declaration must be in its own statement (Uni-line context) | 0.01 | 0.02 | | |
| Parameter must match pattern '^[a-z]([a-z0-9][a-zA-Z0-9]*)?$' (Uni-line context) | 0.01 | 0.02 | | |
| 'CTOR_DEF' should be separated from previous statement (Multi-line context) | 0.00 | 0.01 | | |
| 'protected' modifier out of order with the JLS suggestions (Uni-line context) | 0.00 | 0.01 | | |
| 'public' modifier out of order with the JLS suggestions (Uni-line context) | 0.00 | 0.01 | | |
| Array brackets at illegal position (Uni-line context) | 0.00 | 0.01 | | |
| Catch parameter name must match pattern '^[a-z]([a-z0-9][a-zA-Z0-9]*)?$' (Uni-line context) | 0.00 | 0.01 | | |

**Table 4** (continued)

| Violation of the Google Java style | All% | Eclipse% | Jasper% | Spring% |
|---|---|---|---|---|
| GenericWhitespace '>' is followed by whitespace (Uni-line context) | 0.00 | 0.01 | | |
| Redundant <p> tag (Multi-line context) | 0.00 | | | 0.01 |
| Top-level class BookmarkStack has to reside in its own source file (Files context) | 0.00 | | 0.01 | |
| Switch without "default" clause (Multi-line context) | 0.00 | | 0.01 | |

– All–Google (Count: 30,727, Ignore: 14,575)
– Eclipse-Google (Count: 11,157, Ignore: 4,431)
– Jasper-Google (Count: 10,546, Ignore: 4,299)
– Spring-Google (Count: 9,024, Ignore: 5,845)

All the datasets seemed unbalanced with respect to the number of lines belonging to the considered decision classes (violation and non-violation). The percentage of lines violating certain rules of both standards are presented in Tables 3 and 4. The data presented in the latter table shows that nearly all the lines violated the Google's rule stating that developers shall avoid using tabular characters (97.91–99.42%) and more than half of the lines used wrong indentation levels (53.82–69.01%). The frequency of the other rules violations was visibly lower. Therefore, we decided to introduce another variant of each data set by excluding the two over-represented types of violations to avoid making the problem too trivial (we denoted this sets as Google'):

– All–Google' (Count: 4,423, Ignore: 40,879)
– Eclipse-Google' (Count: 1,066, Ignore: 14,522)
– Jasper-Google' (Count: 2,365, Ignore: 12,480)

**Table 5** The results of the prediction quality evaluation — lines violating any of the coding convention's rules (averages and standard deviations)

| Guidelines | Dataset | Accuracy % | Precision | Recall | F-score |
|---|---|---|---|---|---|
| Sun | All | 99.54±0.11 | 1.00±0.00 | 1.00±0.00 | 1.00±0.00 |
| Sun | Eclipse | 99.05±0.27 | 0.99±0.00 | 0.99±0.00 | 0.99±0.00 |
| Sun | Jasper | 99.55±0.20 | 1.00±0.00 | 1.00±0.00 | 1.00±0.00 |
| Sun | Spring | 99.15±0.22 | 0.99±0.00 | 0.99±0.00 | 0.99±0.00 |
| Google | All | 99.87±0.06 | 1.00±0.00 | 1.00±0.00 | 1.00±0.00 |
| Google | Eclipse | 99.93±0.06 | 1.00±0.00 | 1.00±0.00 | 1.00±0.00 |
| Google | Jasper | 99.70±0.14 | 1.00±0.00 | 1.00±0.00 | 1.00±0.00 |
| Google | Spring | 99.88±0.11 | 1.00±0.00 | 1.00±0.00 | 1.00±0.00 |
| Google' | All | 98.98±0.17 | 0.99±0.00 | 0.99±0.00 | 0.99±0.00 |
| Google' | Eclipse | 99.26±0.29 | 0.99±0.00 | 0.99±0.00 | 0.99±0.00 |
| Google' | Jasper | 99.03±0.26 | 0.99±0.00 | 0.99±0.00 | 0.99±0.00 |
| Google' | Spring | 98.91±0.29 | 0.99±0.00 | 0.99±0.00 | 0.99±0.00 |

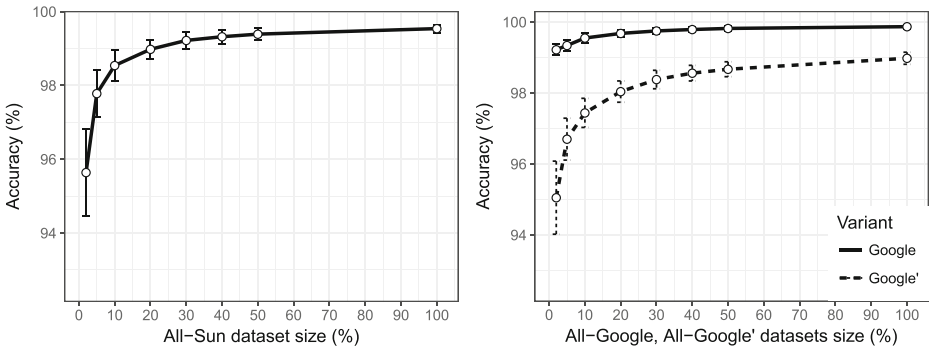



**Fig. 5** Learning curves showing impact of the dataset size on Accuracy

– Spring-Google' (Count: 992, Ignore: 13,877)

We performed ten runs of 10-fold cross-validation procedure. We evaluated the prediction quality of the classifier based on a set of commonly used measures, such as Accuracy, Precision, Recall, and F-score.

The results of the cross-validation are presented in Table 5. The accuracy for the Sun's coding standard ranged between 99.05% and 99.55%. For the Google and Google' coding standards, the accuracy ranged between 99.70–99.93% and 98.91–99.26%, respectively. The observed accuracy seemed stable between the runs of the cross-validation procedure (the maximum standard deviation for accuracy was equal to 0.29%). We also did not observe any visible differences between Precision and Recall, which ranged between 0.99 and 1.00.

The learning curves for the accuracy are presented in Fig. 5. We observed that even for a small dataset consisting of 2% of the whole dataset, the observed Accuracy was high—95.64% for Sun Java Coding Conventions, and 99.22% / 95.05% for the Google / Google' Java Style Guides. We also noticed that the variability the results was visibly higher for the smaller datasets.

After finishing the last cycle of our study, we decided to revisit this analysis and extend it to learn more about the accuracy of CCFlex when detecting violations of different types of coding rules.[18] This time, we trained separate binary classifiers (decision trees) for each of the rules in the Sun's and Google's coding conventions and evaluated their accuracy by performing 10-runs of 10-fold cross-validation. We used stratified sampling to create folds to avoid the situation when some of the training datasets would not contain any violations. Since this part of the analysis was performed after the last cycle of our study, we used the final version of the CCFlex tool with all the features presented in Section 3.

The prediction quality for the rules that have at least 10 violations in the dataset are presented in the Table 6 (Sun Java Coding Conventions) and Table 7 (Google Java Style Guide). The observed prediction quality was high with the average F-score around 0.80 (for 50% of the rules, F-score was 0.90 or higher). It seems that the easiest to detect were violations of the rule belonging to the keyword category. Also, a high prediction quality was observed for the rules belonging to the line-properties category (F-score equal to 0.947 and 0.996). These rules regarded the maximum number of characters in a line. The tool was not able to achieve the perfect F-score although there was a feature 'number of characters' that was

[18]The replication package for this analysis is available at https://github.com/mochodek/py-ccflex-java-sun-google.

**Table 6** The results of the prediction quality evaluation for lines violating particular Sun Java Coding Conventions rules (45,302 SLOC; rules with at least 10 violations)

| Rule | N | Accuracy % | Precision | Recall | F-score |
|---|---|---|---|---|---|
| '{' should be on the previous line. (Multi-line) | 1,479 | 99.98 | 0.998 | 0.997 | 0.997 |
| Line is longer than 80 characters (Line properties) | 4,404 | 99.93 | 0.997 | 0.996 | 0.996 |
| Avoid inline conditionals. (Uni-line) | 153 | 100.00 | 0.997 | 0.991 | 0.994 |
| 'for' construct must use '{}'s. (Multi-line) | 15 | 100.00 | 1.000 | 0.987 | 0.993 |
| Parameter should be final (Uni-line) | 1,597 | 99.90 | 0.983 | 0.990 | 0.987 |
| Missing a Javadoc comment. (Multi-line) | 1,153 | 99.88 | 0.979 | 0.975 | 0.977 |
| Variable must be private and have accessor methods (Multi-line) | 295 | 99.95 | 0.971 | 0.949 | 0.960 |
| Line has trailing spaces. (Uni-line) | 950 | 99.82 | 0.958 | 0.954 | 0.956 |
| 'static' modifier out of order with the JLS suggestions. (Uni-line) | 11 | 100.00 | 1.000 | 0.909 | 0.952 |
| Class designed for extension without Javadoc (Design semantics) | 977 | 99.79 | 0.938 | 0.965 | 0.951 |
| 'if' construct must use '{}'s. (Multi-line) | 92 | 99.98 | 0.906 | 0.983 | 0.943 |
| Symbol is preceded with whitespace. (Uni-line) | 31 | 99.99 | 0.966 | 0.919 | 0.942 |
| '}' should be on the same line as the next part of a multi-block statement (Multi-line) | 186 | 99.93 | 0.884 | 0.949 | 0.915 |
| Name must match pattern '^[A-Z][A-Z0-9]*(_[A-Z0-9]+)*$' (Uni-line) | 50 | 99.97 | 0.810 | 0.922 | 0.862 |
| Expected @throws tag (Multi-line) | 74 | 99.95 | 0.838 | 0.866 | 0.852 |

**Table 6** (continued)

| Rule | N | Accuracy % | Precision | Recall | F-score |
|---|---|---|---|---|---|
| Symbol is followed by whitespace. (Uni-line) | 48 | 99.97 | 0.822 | 0.871 | 0.845 |
| Symbol is not followed by whitespace. (Uni-line) | 253 | 99.81 | 0.823 | 0.831 | 0.827 |
| Expected an @return tag. (Multi-line) | 246 | 99.79 | 0.814 | 0.802 | 0.808 |
| Expected @param tag (Multi-line) | 508 | 99.55 | 0.796 | 0.801 | 0.799 |
| Redundant 'final' modifier. (Multi-line) | 37 | 99.96 | 0.760 | 0.805 | 0.781 |
| Hidden field (Files context) | 388 | 99.57 | 0.752 | 0.744 | 0.748 |
| Symbol should be on a new line. (Multi-line) | 56 | 99.94 | 0.782 | 0.709 | 0.743 |
| Symbol is not preceded with whitespace. (Uni-line) | 35 | 99.95 | 0.699 | 0.683 | 0.690 |
| First sentence should end with a period. (Multi-line) | 66 | 99.91 | 0.686 | 0.679 | 0.682 |
| Redundant 'public' modifier. (Multi-line) | 32 | 99.95 | 0.623 | 0.606 | 0.614 |
| Magic number (Uni-line) | 22 | 99.92 | 0.268 | 0.382 | 0.315 |
| Unused import (Multi-line) | 37 | 99.85 | 0.034 | 0.032 | 0.033 |

**Table 7** The results of the prediction quality evaluation for lines violating particular Google Java Style Guides rules (45,302 SLOC; rules with at least 10 violations)

| Rule | N | Accuracy % | Precision | Recall | F-score |
|---|---|---|---|---|---|
| Line contains a tab character. (Keyword) | 30,366 | 100.00 | 1.000 | 1.000 | 1.000 |
| '(' is followed by whitespace. (Uni-line) | 29 | 100.00 | 1.000 | 1.000 | 1.000 |
| '{' should be on the previous line (Multi-line) | 1,479 | 99.98 | 0.997 | 0.998 | 0.997 |
| Incorrect indentation level (Multi-line) | 18,277 | 99.51 | 0.993 | 0.995 | 0.994 |
| At-clause should have a non-empty description. (Uni-line) | 315 | 99.99 | 0.990 | 0.995 | 0.992 |
| 'package' should be separated from previous statement. (Uni-line) | 95 | 99.99 | 0.979 | 0.989 | 0.984 |
| 'for' construct must use '{}'s. (Multi-line) | 15 | 100.00 | 1.000 | 0.940 | 0.969 |
| ')' is preceded with whitespace. (Uni-line) | 30 | 100.00 | 0.967 | 0.963 | 0.965 |
| 'if' construct must use '{}'s. (Multi-line) | 92 | 99.98 | 0.929 | 0.977 | 0.952 |
| Line is longer than 100 characters (Line properties) | 1,178 | 99.73 | 0.954 | 0.940 | 0.947 |
| First sentence of Javadoc is incomplete (period is missing) or not present. (Multi-line) | 417 | 99.88 | 0.909 | 0.963 | 0.935 |
| '}' should be on the same line as the next part of a multi-block statement (Multi-line) | 186 | 99.92 | 0.867 | 0.957 | 0.910 |
| <p> tag should be preceded with an empty line. (Multi-line) | 230 | 99.91 | 0.918 | 0.900 | 0.909 |
| 'static' modifier out of order with the JLS suggestions. (Uni-line) | 11 | 100.00 | 1.000 | 0.818 | 0.900 |
| <p> tag should be placed immediately before the first word, with no space after. (Uni-line) | 31 | 99.97 | 0.791 | 0.790 | 0.790 |
| Abbreviation in name must contain no more than '2' consecutive capital letters (Uni-line) | 68 | 99.93 | 0.769 | 0.794 | 0.782 |

**Table 7** (continued)

| Rule | N | Accuracy % | Precision | Recall | F-score |
|---|---|---|---|---|---|
| Member name must match pattern '^[a-z][a-z0-9][a-zA-Z0-9]*$' (Uni-line) | 33 | 99.97 | 0.775 | 0.758 | 0.766 |
| Missing a Javadoc comment. (Multi-line) | 56 | 99.93 | 0.718 | 0.721 | 0.719 |
| Symbol should be on a new line (Uni-line) | 55 | 99.92 | 0.702 | 0.658 | 0.679 |
| Whitespace around a symbol is not followed by whitespace (Uni-line) | 73 | 99.88 | 0.632 | 0.641 | 0.636 |
| Whitespace around a symbol is not preceded by whitespace (Uni-line) | 35 | 99.94 | 0.620 | 0.643 | 0.631 |
| Empty line should be followed by <p> tag on the next line. (Multi-line) | 11 | 99.98 | 0.651 | 0.545 | 0.592 |
| Wrong lexicographical order for import (Uni-line) | 49 | 99.85 | 0.310 | 0.306 | 0.308 |
| Overload methods should not be split (Multi-line) | 11 | 99.96 | 0.000 | 0.000 | 0.000 |

solely sufficient to recognize violations of these rules. However, after further investigation, we have learned that Checkstyle ignores lines with package and import statements while recognizing violations of these rules what increases the complexity of the rules and explains the difficulties in learning to recognize violations of these rules.

For the multi-line-context rules, we observed that the tool was effective in recognizing their violations as long as the rules required to "understand" the context of a few lines (e.g. '{' should be on the previous line, 'for' construct must use '{}'s, 'Missing a Javadoc comment'). At the same time, it had difficulties in learning to recognize violations of multi-line rules that required capturing relationships between the lines in larger chunks of code (e.g., 'Expected @param tag', or 'Expected an @return tag'). The worst accuracy was observed for the rule requiring to recognize "unused imports" (F-score = 0.03) and the rule stating that the overloaded methods should be kept together (F-score = 0.00). However, this was not a surprising result taking into account that CCFlex does not extract features allowing to capture such dependencies in the code.

For the uni-line-context rules, one of the two rules that were most difficult to learn was the rule disallowing using so-called 'magic numbers' (F-score = 0.32). Magic numbers are numeric literals that are not defined as constants, however, Checkstyle does not consider the numbers -1, 0, 1, and 2 to be magic numbers. Unfortunately, since we used token signatures in our analysis, all of the numbers in the code were simplified to '0' making it difficult to learn the rule.

There was only one rule belonging to the design-semantics category in the considered coding conventions ('Class designed for extension without Javadoc'). We observed a high prediction quality for this rule (F-score = 0.95), however, this could be caused by the fact that it is a specific case of the 'Missing a Javadoc comment' rule.

We observed a moderately high prediction quality for the rule 'Hidden field' that belongs to the files-context category (F-score = 0.75). The rule states that a local variable or a parameter shall not shadow a field that is defined in the same class. Achieving such a high accuracy was a surprising result since we were not able to find any features extracted by CCFlex that could allow us to detect shadowing of fields. However, there might some correlations or coexistence of code structures in the considered source code that allowed the tool to indirectly learn to recognize lines violating this rule.

Finally, we repeated the analysis for different sub-samples of the original dataset to observe how the size of the training dataset affects the prediction quality of the classifiers. We used stratified sampling to create sub-samples containing 5% (2,265 SLOC), 10% (4,530 SLOC), 20% (9,060 SLOC), 30% (13,591 SLOC), 40% (18,121 SLOC), and 50% (22,651 SLOC) of the lines from the original dataset. The obtained learning curves for different categories of coding guidelines are presented in Fig. 6.[19] The observations are similar to those from the previous analysis. In most cases, the accuracy increases while variance decreases with the increase of the dataset size.

Based on the results of this analysis we made the following observations:

– CCFlex was able to achieve high accuracy of identifying lines violating the coding guidelines for the simplified problem of recognizing lines violating *any* of the rules and for the problem of identifying violations of particular rules. However, for the latter problem, we observed rules for which the tool failed to learn to recognize their violations.

[19]To have at least two violations of the rules in each of the smallest 5% datasets (one line for training and the second one for the validation), we included only the rules that had at least 40 violations in the code.

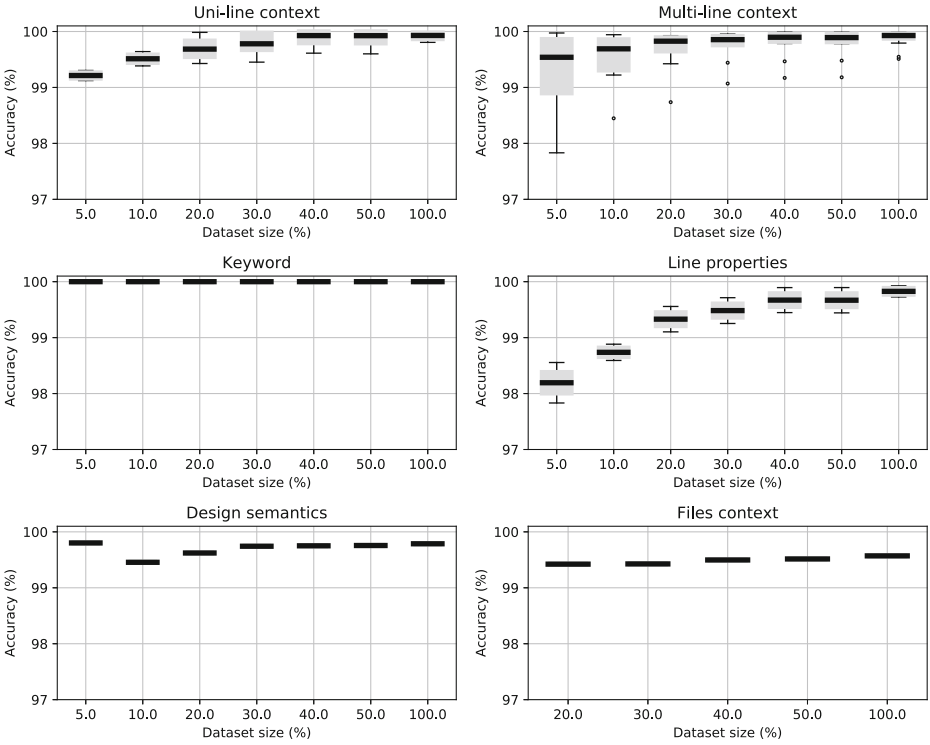



**Fig. 6** Learning curves showing impact of the dataset size on Accuracy of predicting certain violations types (only the rules having at least 40 violations are presented)

– CCFlex was able to achieve high accuracy even for the smallest datasets. Therefore, it seemed that the CCFlex could also be used to identify lines violating similar coding standards even if the lines had to be labeled manually.
– Sun's and Google's guidelines could be mapped to the proposed taxonomy. However, by comparing these guidelines and Java Open Source code to guidelines and C/C++ code of our industrial partners, the latter seemed to be more complicated and richer when it comes to syntax and language constructs being used. Therefore, we perceived the accuracy observed in this study as an upper bound of what we could expect for the study on the code of our partners.

## 5.3 Action Research Cycle 3 – how can we Recognize the Violations Provided by the Industrial Partners?

### 5.3.1 Cycle Goal and Research Procedure

When **diagnosing** the problem of how to recognize the violations provided by industrial partners, we studied their coding/design guidelines.

Once we understood that we can recognize the same violations as a popular style checking tool, we set off to recognize the violations of the coding/design guidelines at Company A. Company A has 45 coding guidelines and a codebase distributed over several source code repositories. Each of the repositories contains a dedicated part of the product and is

ca. 50,000 SLOC in size. The code is written in C. The goal of this cycle was to explore whether it is possible to recognize violations of different types of rules in the industrial context. In particular, we were interested in studying how different coding styles can influence the accuracy of the ML-based tool. Our constraints were that we set off to spend two working days at the company site. We obtained coding guidelines beforehand, but we were not able to obtain access to the source code until we arrived at the company site.

In Company A, we **planned** the evaluation of three coding guidelines belonging to different categories of our taxonomy. The company provided us with the unique possibility to study designated codebases developed at three different time periods — (i) long before the guidelines were defined, (ii) in the same period when guidelines were being defined, and (iii) long after the guidelines were defined. This provided us with the opportunity to study the question: *What is the influence of the difference between programming styles of the training and classified codebase?* In particular, we wanted to understand whether the evolution of the codebase requires the evolution of the examples used to train our machine learning and which configurations of CCFlex parameters provide best results in terms of finding violations.

In our plan, we decided to recognize the following guidelines:

- Pre-processor directives must be placed at the beginning of an empty line, and must never be indented (semantics, uni-line context). We chose this rule because it requires understanding the position of specific tokens in a line.
- For public enumerations, the members of enum should follow the pattern, i.e., the name of the component, underscore, and the value name, for example `ComponentName_ValueOne = 0` (semantics, multi-line context). We chose this rule because it requires to understand the context, i.e. recognition of the lines within ”enum” blocks. We expected that this type of recognition could be difficult for CCFlex as it is primarily designed for one-line rules.
- Names of the variables should follow the so-called camel case format — each word or abbreviation in the middle of the name begins with a capital letter (semantics, uni-line context). We expected that this type of recognition could be difficult for CCFlex as it requires to understand the concept of lower and upper cases and the fact that the token represents the variable name.

We planned to conduct as many exploratory trials as possible within the two days company visit. We planned to experiment with the following configurations of CCFlex (and their combinations):

- Bag of words — to explore whether this way of providing meaning to the constructs allows teaching the tool quicker.
- Active Learning — to explore the ease-of-use of providing examples line-by-line (suggested by active learning) rather than manually.
- Adding new features — to explore how important it is to have the right set of features for the decision tree (CART) algorithm used in CCFlex; in particular whether it is better to rely on bag-of-words or on adding the ability to recognize specific keywords.

During each trial we took three measurements: (i) number of lines in the training file, (ii) number of lines found as violations (true-positives + false-positives) and (iii) number of correct violations (true-positives).

Based on these trials, we wanted to learn how to prepare the evaluation strategy for the next cycle at Company B. We wanted to explore which strategies work best, what Recall we

can achieve and differences between these parameters, types or guidelines and the Recall measure.

Since we intended to use this action research cycle to learn the practical challenges of recognizing violations in the real product source code, we did not have an oracle in terms of tools to recognize these violations. The company used manual reviews as one of the quality assurance techniques, and therefore we followed the same principle—we studied the identified violations and focused on the ratio between the true-positives and false-positives since we would not be able to manually verify all the negatives.

### 5.3.2 Cycle Execution and Results

**Execution: Pre-processor macros** The results from the evaluation are presented in Fig. 7. It shows that the first trial resulted in the highest number of violations found and the highest number of correct violations (true-positives). The legacy code was developed before the coding guidelines were in place and designers could not follow them; the size of the legacy code during this evaluation was 40,010 LOC. For this trial, we used a sample from the legacy codebase for training. We classified 291 lines of code as the training set (based on one file from the legacy codebase, example of code from the coding guidelines, and variations of these).

In the second trial, we used the same training set and we applied the classifier to a new codebase, which was developed after the guidelines were in place; therefore the guidelines should be followed. The size of that code was 41,704 LOC. As Fig. 7 shows, the number of correctly classified violations was significantly lower. However, there were multiple cases where the classifier found violations incorrectly (false-positives). The incorrectly classified violations were caused by the change in the company's style for writing constants as preprocessor directives. In the legacy codebase, the preprocessor constants were often defined using small letters, while the enum values were defined using capital letters. In the new codebase, the preprocessor constants were defined using capital letters, which introduced false-positives in the evaluation. CCFlex recognized these pre-processor constants

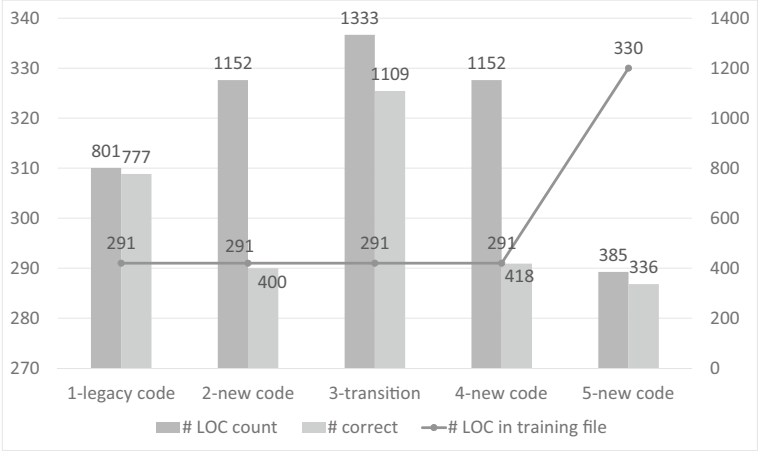



**Fig. 7** Size of the training set for each trial at Company A – recognizing preprocessor macros

as violations of the rule since their name did not have the corresponding context, e.g. no corresponding module name was present in the succeeding lines.

In the third trial, we used the same training set and applied the classifier to the transition codebase. It was the codebase which was developed at the same time when coding guidelines were defined, so it could contain violations. The size of this codebase was 55,026 LOC. The industrial partners provided us with this codebase with the motivation that "it resembles a situation when someone did not always follow the guidelines." As Fig. 7 shows, there were more violations and the classifier was more correct in finding the violations.

In the fourth trial, we used the same training set, but included the statistics of the most commonly used words as features (bag-of-words). When applied to the same new codebase as in trail 2, this decreased the number of false-positives.

In the fifth trial, we added 40 lines as examples from the new codebase, which resulted in a visible decrease in the number of false-positives. The percentage of correctly identified violations was 87%.

For the **evaluation** part, we conducted a workshop with four representatives at the company site, who are architects, designers, and managers. We presented them with the identified violations and noted their reflections. During the discussion with the architects from Company A, they assessed these results as valuable from the practical perspective.

From these five trials, we could **learn** that it is important to use the same programming style for the training and the validation set. We learned that, although from the same company, the coding style evolves significantly over time and this has an impact on the correctness of the identification (if the training set does not evolve together with the coding style). We have also learned that the size of the training set can be small: 291 – 331 SLOC to achieve a satisfactory prediction quality for a basic semantic, uni-line context guideline.

**Execution: Enums** For this guideline, we were provided the following example of a code that follows it.

```
typedef enum
{
  ComponentName_ValueOne = 0,
  ComponentName_ValueTwo,
  ComponentName_ValueThree
} ComponentName_MyEnum;
```

Based on this guideline we could provide the following example of a violation to bootstrap the training of CCFlex.

```
typedef enum
{ ValueOne = 0,
  VALUETWO,
  three
} ComponentName_MyEnum;
```

These guidelines require to understand the multi-line context of the line, i.e. that the guideline applies to the content of an enum and is not just a standard variable declaration. Therefore, in this trial, we tested the ability of CCFlex to recognize violations requiring to understand the context, both in terms of the name of the component (name of the enum) and the name of the enum constant.

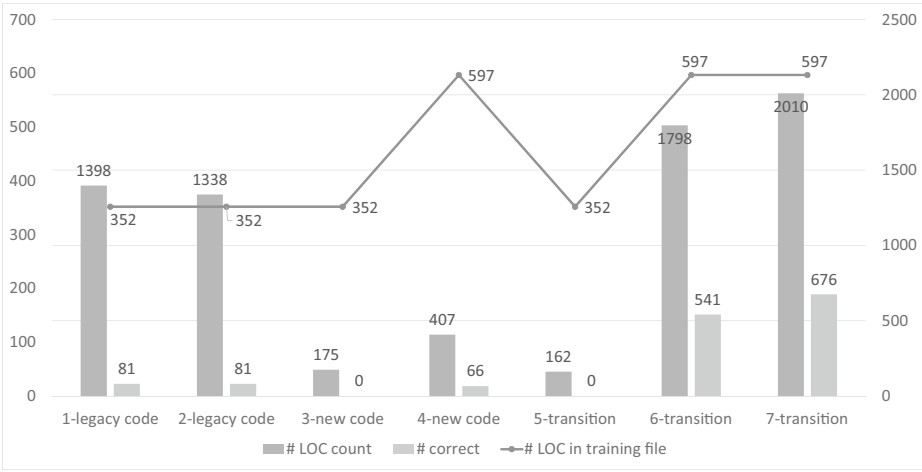



**Fig. 8** Size of the training set for each trial at Company A – recognizing enums

The results from recognizing violations of naming of enums is presented in Fig. 8. We use the same measurements as previously—the number of lines in the training set, the number of lines classified as violations and the number of lines correctly classified as violations.

For the first trial, we used the same training set as for the first trial of the recognizing pre-processor macros, removed the previous labeling and added the example and counterexample of the guideline for enums. The training set contained 352 lines. We used the bag-of-words parameter in the first trial and no bag-of-words in the second trial. The number of lines violating the guideline was 1398 and 1338 respectively, with 81 correct instances in both cases.

For the third trial, we changed the codebase for validation and the number of correctly identified dropped to 0, which was caused by the same issue as for pre-processor macros—the company changed the naming conventions. In order to reduce that problem, we used active learning, which resulted in adding 245 lines in the training set. The number of correctly identified violations increased to 66, and the number of all instances classified as violations increased to 407 lines.

For the fifth trial, we repeated the set-up of the third trial but changed the codebase to transition code. We did not capture any violations correctly, and 162 lines were falsely identified as violations. Using the same training set in the sixth trial as for the fourth trial, i.e. 597 lines mixed from the legacy code and the new code, we correctly identified 541 violations and 1798 lines in total as violations. For the last, the eight trial, we started with the 352 lines of the training set and used active learning to complement with 245 lines, we increased the number of identified violations to 2010 and the correctly identified violations to 676.

**Execution: Camel cases** For the last guideline, we used the same base training set and complemented it with 16 examples of correctly and incorrectly used camel cases. Also, we labeled the lines in the base training set where the rules were not followed.

The results are presented in Fig. 9, with the same measurements as for the previous two guidelines.

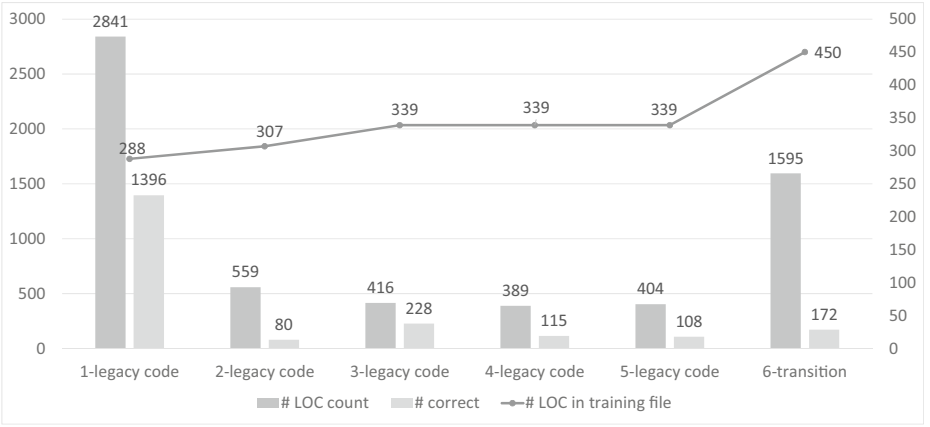



**Fig. 9** Size of the training set for each trial at Company A – recognizing camel cases

For the first trial, with 288 lines in the training set, we identified 2841 lines as violations, with 1396 correct instances. Increasing the number of lines in the training set decreased the number of correctly identified instances and the total number of instances to 80 and 559, respectively. This was a deterioration, as we had established that there are 1396 violations in the first trial. Therefore, in the third trial, we applied active learning and increased the number of lines in the training set to 339 from the original 288. This resulted in the increase of the correctly identified instances, but only to 228. In the fourth trial, we used the bag-of-words, which decreased the number of instances found, consistent with our previous observations for recognizing enum naming violations. In the fifth trial, we increased the number of lines in the training set, providing more examples from the false-positives found in the fourth trial. However, there was even a decrease in the number of instances classified correctly.

Finally, we applied the algorithm trained on the old code, with active learning using lines from the transitional code, on the transitional code. The training set contained 450 lines and we found 1595 violations, out of which 172 were correctly recognized as violations, the rest was false-positives. Due to the fact that the company visit was limited in time, that was the last trial we could perform.

We **learned** that the original set-up of CCFlex at that time did not allow to find violations of camel case guideline with a satisfactory Recall. We identified the need to use bi-grams, block features, and token signatures in the next version of CCFlex and applying it to the next company.

## 5.4 Action Research Cycle 4 – how much Training of CCFlex is Required to Reduce the Percentage of False-Positives?

### 5.4.1 Cycle Goal and Research Procedure

In the last cycle of our action research, we used Company B as the case, where we had the opportunity to study 66 coding guidelines. We focused on the understanding of how much training data and iterations are needed to train a classifier for different types of rules from our taxonomy. We chose seven guidelines to evaluate:

- 120 characters—line length must not exceed 120 characters (semantics-free, line properties).
- Braces in compound statements—braces must be used for all compound statements (semantics, multi-line context)—this rule helps to control the readability of the code and thus minimize programming mistakes.
- Do not use variants—software units must not have variants at build-time (semantics-free, keyword)—this rule helps to assure that `#ifdef` pre-processor statements are used scarcely to minimize the need for understanding which code is compiled during each build.
- Named constants—named constants must be used (semantics, uni-line context)—instead of using untyped `#define` pre-processor directive, this rule helps to enforce usage of constants, which are typed.
- One statement per line—only one statement per line of code is allowed (semantics, uni-line context)—this rule helps to enforce the simplicity of the code and reduce the cognitive burden when reading the code.
- Use Enum classes—C++11 Enum classes must be used instead of traditional `enum` types (semantics-free, keyword)—instead of `enum` types, the code should use `enum` classes, which can enforce constructors, typing, and destructors.
- Use constants instead of macros—C++ constructs must be used instead of pre-processor macros (semantics-free, keyword).

Based on the observations from the previous action-research cycle, we made two modifications to the research procedure. Firstly, we designed oracles for the research purposes, i.e. dedicated, heuristic scripts implemented to recognize violations of specific rules. We needed that script in order to be able to calculate the Precision, Recall and F-score measures. Secondly, we found a source codebase without the gradual progression of quality like it was in the case of Company A; all code in the codebase should follow the coding guidelines.

We set the following stop criteria for each trial: 90% in Precision or Recall (on the module from which we polled the training examples) or seven training iterations (but a minimum of three trials to verify that the observed prediction quality is stable). The second criterion was important as we wanted to limit our training set to less than 800 LOC in order to assure that it is not very time consuming for practitioners to classify the code.

### 5.4.2 Cycle Execution and Results

The summary of the Precision, Recall, and F-score for all rules for Company B is presented in Fig. 10.

The summary shows that for simple coding guidelines, like the guideline ”Line length must not exceed 120 characters”, the training took only three iterations. However, for the most complex guidelines, like ”Braces must be used for all compound statements”, even seven iterations did not result in high Precision, Recall and F-score.

The summary for the F-score, per rule and per training trial is presented in Fig. 11. Each line represents one rule and the number of trials differs per rule because not all rules required the same number of lines in the training set to reach the stop criteria. Although the goal was to add 100 lines to the training set for each trial, some rules required context and therefore we needed to add extra lines to close blocks (for example for `enum` or comment).

The figure shows that there is a big difference between the achieved F-score per rule, which is consistent with the results presented in Fig. 10. The F-score for the entire codebase (ca. 3M SLOC) is usually lower than the F-score for the last training trial on the module

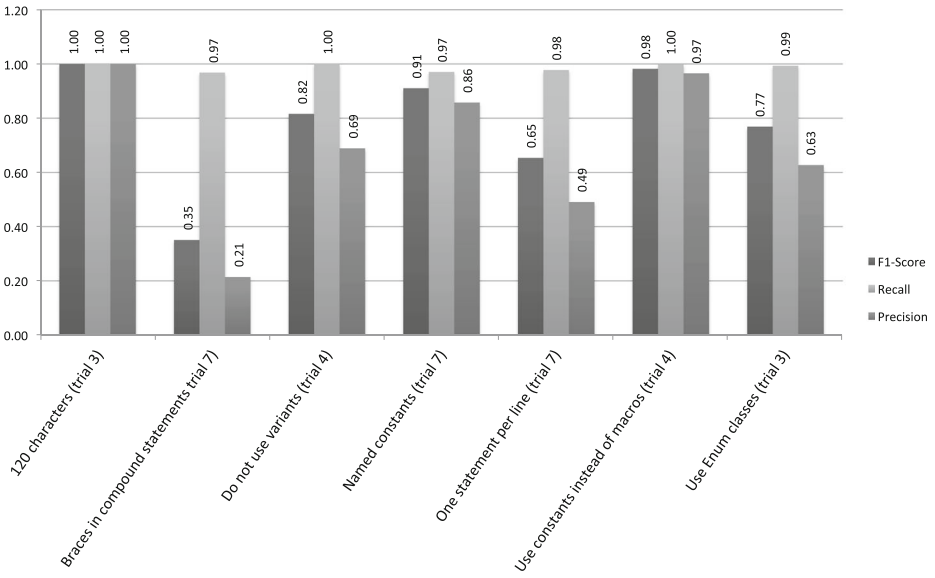



**Fig. 10** Precision, Recall and F-score for all trials at Company B

from which we polled the examples to the training set. The reason is that the code in the module used for training trials did not contain all possible language constructs which were present in the entire codebase.

In order to understand the impact of using active learning, we performed the same trials, adding the lines to the training set using pool-based active learning. We tried two querying strategies (i) uncertainty sampling and (ii) committee-based vote entropy sampling (Dagan and Engelson 1995) (a committee of random forest, CART, and k-nearest neighbors classifiers). However, the latter strategy turned out to be superior. The results for committee-based sampling are presented in Fig. 11.

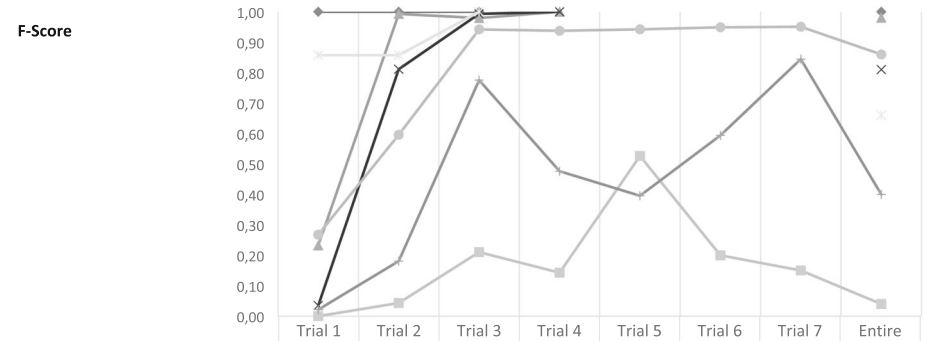
Figure 11 shows that the F-score is higher if we use active learning. Therefore, we recommend this strategy as the main strategy compared to the manual selection of lines for the training set.

The rule with the lowest F-score, "Brackets must be used for all compound statements," was the most difficult one for CCFlex. The lowest score is based on low Recall, which means that CCFlex provided too many false-positives. However, when developing the oracle for this rule, we also noticed that it was difficult to assess which line is indeed a false-positive and which is not even when writing a dedicated tool for this purpose—simply because of the way the code was written at Company B (e.g., using many of the advanced features of C++ 11 and local code optimizations).

### 5.5 Summary of the Results

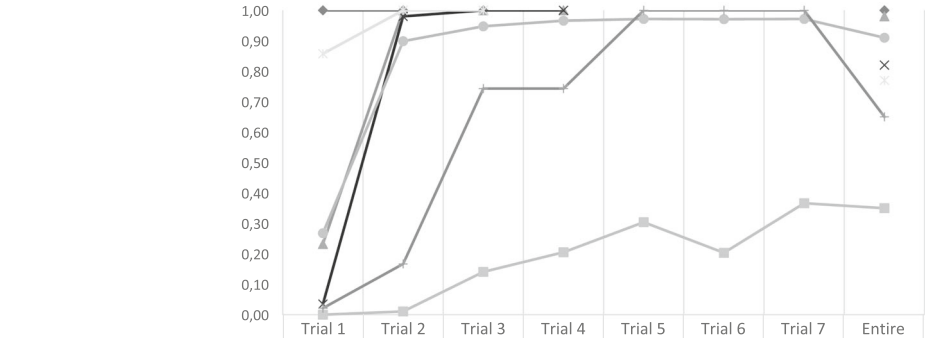
The outcomes of each of four action-research cycles provided us with valuable insights into using an ML-based tool to recognize company-specific code guidelines violations in code.

From the second cycle, we have learned that the ML-based tool, such as CCFlex, can achieve a high prediction quality for Java coding-style/formatting guidelines when enough examples are provided (CCFlex was able to detect lines violating Sun Coding Conventions




| | Trial 1 (ca. 100 LOC) | Trial 2 (ca. 200 LOC) | Trial 3 (ca. 300 LOC) | Trial 4 (ca. 400 LOC) | Trial 5 (ca. 500 LOC) | Trial 6 (ca. 600 LOC) | Trial 7 (ca. 700 LOC) | Entire code base |
|---|---|---|---|---|---|---|---|---|
| 120 Characters | 1,00 | 1,00 | 1,00 | | | | | 1,00 |
| Braces in compund statements | 0,00 | 0,04 | 0,21 | 0,14 | 0,53 | 0,20 | 0,15 | 0,04 |
| Use constants instead of macros | 0,23 | 0,99 | 0,98 | 1,00 | | | | 0,98 |
| Do not use variants | 0,03 | 0,81 | 0,99 | 1,00 | | | | 0,81 |
| Use Enum classes | 0,86 | 0,86 | 1,00 | | | | | 0,66 |
| Named constants | 0,27 | 0,60 | 0,94 | 0,94 | 0,94 | 0,95 | 0,95 | 0,86 |
| One statement per line | 0,02 | 0,18 | 0,78 | 0,48 | 0,40 | 0,59 | 0,84 | 0,40 |

**Manual selection**




| | Trial 1 (ca. 100 LOC) | Trial 2 (ca. 200 LOC) | Trial 3 (ca. 300 LOC) | Trial 4 (ca. 400 LOC) | Trial 5 (ca. 500 LOC) | Trial 6 (ca. 600 LOC) | Trial 7 (ca. 700 LOC) | Entire code base |
|---|---|---|---|---|---|---|---|---|
| 120 Characters | 1,00 | 1,00 | 1,00 | | | | | 1,00 |
| Braces in compund statements | 0,00 | 0,01 | 0,14 | 0,21 | 0,30 | 0,20 | 0,37 | 0,35 |
| Use constants instead of macros | 0,23 | 1,00 | 1,00 | 1,00 | | | | 0,98 |
| Do not use variants | 0,03 | 0,98 | 1,00 | 1,00 | | | | 0,82 |
| Use Enum classes | 0,86 | 1,00 | 1,00 | | | | | 0,77 |
| Named constants | 0,27 | 0,90 | 0,95 | 0,97 | 0,97 | 0,97 | 0,97 | 0,91 |
| One statement per line | 0,02 | 0,17 | 0,74 | 0,74 | 1,00 | 1,00 | 1,00 | 0,65 |

**Active Learning**

**Fig. 11** F-score per trial when selecting the lines to the training set manually and with active learning for a module from which training samples were polled and for entire codebase

and Google Java Style Guide, without determining the type of violation, with the Accuracy around 99% and with Recall and Precision ranging between 0.99 and 1.00).

The following third and fourth cycles revealed some pros and cons of using such a tool in an industrial environment. We trained the CCFlex tool to recognize violations of ten guidelines from the guidebooks of our partners and were able to achieve 0.97 and higher Recall

for all the rules we were able to calculate the measure for (Company B). Unfortunately, for some of the guidelines, high Recall was achieved by sacrificing Precision, which ranged from 0.35 to 1.00 depending on the rule.

When combining the results from the cycles two to four, we can state that a pleasant property of ML-based tools for recognizing code violations is that a single tool can be used to detect violations for different programming languages without the need of modifying the code of the tool (of course, it has to be trained for each of the applications).

When it comes to training strategies, we have observed that for most of the rules, we were able to achieve a high Recall even if the training dataset contained only around 300 SLOC. We initially tried to compose training sets by selecting examples from guidebooks and manually adding positive and negative examples from a sample of the codebase, however, we observed that using Active Learning resulted in better accuracy. The general observation was that the rules requiring understanding multi-line context were more difficult to train, while the rules based on text properties or keywords tend to be easier to tackle with. Although we were able to achieve a high Recall (0.97 and more) for all types of rules, Precision was visibly lower for the multi-line context rules (0.21 and 0.33). The rule regarding line properties was the easiest to train (even with a small training dataset of ca. 100 lines). However, this seems possible only when the property in question is used as a feature (in this case the number of characters in the line). Finally, we experienced that the difficulty of training a classifier can also be determined by other factors such as the inherent complexity of the code, i.e., the richness of the programming-language syntax, overloaded operators, local code optimizations, etc.

Finally, we made some observations regarding the maintainability of ML-based tools for code analysis. Firstly, we have learned that designing custom, specialized features can help in recognizing violations when the considered training sample is small. For instance, we were successful in using our token signatures or predefined, keyword or short-pattern based features. We have also learned that the coding style can evolve over time and this could have an impact on the correctness of the violations recognition (if the training set does not evolve together with the coding style). Therefore, when maintaining an ML-based code detector, one has to maintain a list of examples and re-train the tool over time instead of maintaining the source code of the detector, as it is in the case of static-code-analysis tools.

## 6 Validity Evaluation

In our action research study, we evaluated a number of validity threats, based on the framework provided by Wohlin et al. (2000).

The main *construct validity* threats which we identified are related to the choice of coding/design rules and the stop criteria. To mitigate the risk that we bias the evaluation by selecting a very similar, narrow group of coding guidelines to train CCFlex, we based our choices on the taxonomy of coding guidelines that group guidelines based on the context that need be understood to find violations of guidelines. We covered four categories of coding guidelines: semantic uni-/multi-line context and semantic-free line properties and keywords. The remaining three categories were either outside of our interests for this study (process-level rules) or we were not able to describe the contextual information at the level of a single line of code (multiple files context or design decisions). The choice of the stop criteria was arbitrary in cycle 4, which was a deliberate choice as we wanted to understand whether it is possible to achieve satisfactory results with a low number of training instances (lines of code). The alternative would be to continue the Active Learning trials beyond the

seven trials to achieve 90% F-score, but we left this as an open issue for our further work aimed at studying the minimum sufficient set of training examples balanced with the size of the feature set.

Also, it is important to emphasize that the coding style violations do not have to strongly correlate with code faults or product quality. When analyzing the coding guidelines of our industrial partners, we observed that most of the rules in their guidebooks were there to improve code readability and communication between software developers. However, there were also examples of guidelines that forced (or forbade) using some code constructs that could affect the performance of the systems or cause memory management problems (and consequently lead to defects).

The main *internal validity* threat is the fact that in many cases it was not obvious whether a violation found by CCFlex was indeed a violation. For example, for recognizing camel case styled functions, it was not easy to find which of the following two is correct: (i) `isECUPresent()` or (ii) `isEcuPresent()`. Consulting the practitioners led to diverse opinions. For these cases, we chose to include the second version as correct. For all cases like this, we consulted the practitioners and discussed them in our research team to minimize the researcher bias.

The main *conclusion validity* threat is the fact that for we created the oracles in cycle 4 ourselves. The oracles are mainly based on pattern matching and simple parsers. Therefore, they should be treated as heuristics (the scripts are available in CCFlex Github repository). Although we tested the tools manually, we did not provide validation against any industrial tool, this is a threat we need to accept. However, based on the cycle 2, where we evaluated the ability of CCFlex to find violations similar to industry-grade tools, we believe that this threat does not bias our conclusions. Since CCFlex's ability to "mimic" another tool was above 95%, we believe that the accuracy differs by as much as 5% and therefore can be neglected.

Finally, in order to minimize *external validity* threats of being unrepresentative, we diversified our study by including both open source and proprietary codebases. We chose to use oracles that are used in industry (Checkstyle) and own oracles. We evaluated the tool at two different companies, with different coding/design guidelines and different contexts.

Although the results found in the initial study on open source Java projects were slightly better than those in the industrial C/C++ projects, this cannot be taken as a prove that there is a difference in the accuracy of the tool between programming languages because of the visible differences in the study setup, e.g., such as the size of the datasets, the set of guidelines used, detecting lines violating the rules vs. determining the types of violations, or differences in the character of open source and closed source systems.

Finally, we compared various static analysis and style checkers to CCFlex. The observed promising results for C, C++, and Java indicate that similar results might be expected for other imperative languages as well as other companies and other codebases.

## 7 Conclusions

In this paper, we presented the results of an action research study that aimed to support code reviews by automatically recognizing company-specific code guidelines violations in large-scale, industrial source code. The study was performed in the collaboration with two large companies located in Scandinavia developing software-intensive products where we worked on premises of these companies to analyze their code and code/design guidelines.

The study was divided into four action-research-study cycles. In the first cycle, we analyzed the coding guidelines of our partners and proposed a taxonomy to categorize coding

rules depending on the information required to automatically recognize their violations in code. We also identified some requirements and constraints for violation-detecting software tools, which were the ease of adapting the tool to handle new or altered rules, as well as the possibility of processing the code without the need of parsing or compiling it. Based on the conducted literature review study, we learned that nearly all of the existing tools do not meet these requirements. The most promising tools were machine-learning-based ones as they allow for flexible evolution as requested by our industrial partners.

In the second cycle, we performed a preliminary validation of one of the machine-learning-based tools called CCFlex on the codebase of three Java open source products. We looked for violations of Google Java Style Guide and Sun Java Code Conventions, which are two widely accepted coding style/formatting guidelines for Java to learn that the tool was able to detect the lines violating *any* of the guidelines as well as the lines violating *specific* rules with the average accuracy of ca. 99% and and the average F-score of ca. 0.80. Although the overall accuracy was high, we observed that the tool had difficulties in detecting violations of the rules that required to understand the context of multiple lines (e.g., finding unused imports).

In the following two cycles, we investigated the possibility of training the tool to recognize lines violating 10 rules from our industrial partners' coding guidebooks. As a result of these studies, we found that:

- we were able to train the ML-based tool by using the maximum of around 700 SLOC to achieve the average F-score of 0.78. Although we obtained a high Recall (0.97 or higher) for all of the rules (often by using only 300 SLOC), it was usually at the cost of high false-positive rates (Precision ranged from 0.21 to 1.00 depending on the rule). The best results were obtained for the rules requiring understanding the context of a single line (semantical uni-line context, semantic-free line properties, and keywords) while the rules requiring to understand the context of multiple lines were far more difficult to train.
- the ML-based tool was able to recognize code guidelines violations by using features extracted directly from the text (e.g., frequencies of tokens) without the need for parsing or compiling the code,
- we observed that the best strategy for training the tool to recognize violations of company-specific guidelines was to start with the examples provided in the companies' code guidebook and then use Active Learning to poll lines from a sample of the codebase to label.
- we have learned that using ML-based code analysis tools bring new challenges when it comes to maintenance in comparison to static-code-analysis tools (that require source code modification) which is maintaining examples in training codebase.

Finally, our study showed that the same ML-based tool can be trained to recognize violations of different coding guidelines and even for different programming languages (C, C++, and Java).

**Future Work** Our further work includes the integration of our tool with Gerrit—a modern code review tool used in industry (and train the tool on the commit-level) and further studies of industry-wide standard rules like MISRA C++.

We also want to further investigate how *rules that serve as documentation*, *rules on external information*, and *optional rules* (see Section 5.1) can be captured with our approach in future.

Furthermore, many static code analysis tools, e.g. Splint and CPPCheck, style checkers, e.g. CodeCheck, and approaches for code smell detection, e.g. DECOR by Moha et al.

(2010) and BOA by Dyer et al. (2013), offer the option to formulate new rules, often using their own languages. What motivated the use of machine learning in this paper is that it enables the industrial users to formulate new rules without having to learn a new language and not having to verify whether the newly written rules are doing what they are supposed to do. In future work, we plan to further investigate whether specifying rules with our approach is really easier and/or less time consuming for practitioners than writing rules with DECOR, BOA, Splint, or CodeCheck.

**Acknowledgements** The research was conducted in Software Center, Chalmers, University of Gothenburg, Ericsson, and Grundfos. The authors would like to thank all architects, designers, and managers for help and support for this work.



## References

Allamanis M, Barr ET, Bird C, Sutton C (2014) Learning natural coding conventions. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. ACM, pp 281–293

Axelsson S, Baca D, Feldt R, Sidlauskas D, Kacan D (2009) Detecting defects with an interactive code review tool based on visualisation and machine learning. In: The 21st international conference on software engineering and knowledge engineering (SEKE)

Baskerville R, Wood-Harper AT (1996) A critical perspective on action research as a method for information systems research. J Inf Technol 11(2):235–246

Brar HK, Kaur PJ (2015) Static analysis tools for security: a comparative evaluation. International Journal 5(7):1085–1089

Brun Y, Ernst MD (2004) Finding latent code errors via machine learning over program executions. In: Proceedings of the 26th International Conference on Software Engineering, ICSE '04. IEEE Computer Society, Washington, pp 480–490. http://dl.acm.org/citation.cfm?id=998675.999452

Chappelly T, Cifuentes C, Krishnan P, Gevay S (2017) Machine Learning for finding bugs: An initial report. In: IEEE Workshop on Machine learning techniques for software quality evaluation (maLTeSQue). IEEE, pp 21–26

Dagan I, Engelson SP (1995) Committee-based sampling for training probabilistic classifiers. In: Machine Learning Proceedings 1995. Elsevier, pp 150–157

Di Nucci D, Palomba F, Tamburri DA, Serebrenik A, De Lucia A (2018) Detecting code smells using machine learning techniques: are we there yet?. In: 2018 IEEE 25Th international conference on software analysis, evolution and reengineering, SANER. IEEE, pp 612–621

Dyer R, Nguyen HA, Rajan H, Nguyen TN (2013) Boa: a language and infrastructure for analyzing ultra-large-scale software repositories. In: Proceedings of the 2013 International Conference on Software Engineering. IEEE Press, pp 422–431

Emanuelsson P, Nilsson U (2008) A comparative study of industrial static analysis tools. Electron Notes Theor Comput Sci 217:5–21

Fatima A, Bibi S, Hanif R (2018) Comparative study on static code analysis tools for c/c++. In: 2018 15th International Bhurban Conference on Applied Sciences and Technology (IBCAST). IEEE, pp 465–469

Fontana FA, Zanoni M, Marino A, Mantyla MV (2013) Code smell detection: Towards a machine learning-based approach. In: 2013 29th IEEE International Conference on Software Maintenance (ICSM). IEEE, pp 396–399

Fontana FA, Mäntylä MV, Zanoni M, Marino A (2016) Comparing and experimenting machine learning techniques for code smell detection. Empir Softw Eng 21(3):1143–1191

Freitas AA (2014) Comprehensible classification models: a position paper. ACM SIGKDD Explor Newslett 15(1):1–10

Fu Y, Zhu X, Li B (2013) A survey on instance selection for active learning. Knowl Inf Syst 35(2):249–283

Goodman PS, Bazerman M, Conlon E (1980) Institutionalization of planned organizational change. In: Research in Organizational Behavior, JAI Press,Greenwich, pp 215–246
Irwin W, Churcher N (2001) A generated parser of c++. NZ J Comput 8(3):26–37
Mantere M, Uusitalo I, Roning J (2009) Comparison of static code analysis tools. In: 2009. SECURWARE'09, Third International Conference on Emerging security information, systems and technologies. IEEE, pp 15–22
Maruping LM, Zhang X, Venkatesh V (2009) Role of collective ownership and coding standards in coordinating expertise in software project teams. Eur J Inf Syst 18(4):355–371
Masters J (1995) The history of action research. Action Res Electron Read 22:2005
McIntosh S, Kamei Y, Adams B, Hassan AE (2014) The impact of code review coverage and code review participation on software quality: A case study of the qt, vtk, and itk projects. In: Proceedings of the 11th Working Conference on Mining Software Repositories. ACM, pp 192–201
Mi Q, Keung J, Xiao Y, Mensah S, Gao Y (2018) Improving code readability classification using convolutional neural networks. Inf Softw Technol 104:60–71
Moha N, Gueheneuc YG, Duchien AF et al (2010) Decor: a method for the specification and detection of code and design smells. IEEE Trans Softw Eng (TSE) 36(1):20–36
Novak J, Krajnc A, Ontar R (2010) Taxonomy of static code analysis tools. In: MIPRO, 2010 Proceedings of the 33rd International Convention. IEEE, pp 418–422
Ochodek M, Staron M, Bargowski D, Meding W, Hebig R (2017) Using machine learning to design a flexible loc counter. In: IEEE Workshop on Machine learning techniques for software quality evaluation (maLTeSQue). IEEE, pp 14–20
Robson C, McCartan K (2016) Real world research. Wiley, New York
Shaukat R, Shahoor A, Urooj A (2018) Probing into code analysis tools: A comparison of c# supporting static code analyzers. In: 2018 15th International Bhurban Conference on Applied Sciences and Technology (IBCAST). IEEE, pp 455–464
Singh D, Sekar VR, Stolee KT, Johnson B (2017) Evaluating how static analysis tools can reduce code review effort. In: 2017 IEEE Symposium on Visual languages and human-centric computing (VL/HCC). IEEE, pp 101–105
Smit M, Gergel B, Hoover HJ, Stroulia E (2011) Maintainability and source code conventions: An analysis of open source projects. University of Alberta, Department of Computing Science, Tech Rep TR11-06
Susman G, Evered R (1978) An assessment of the scientific merits of action research. J Admin Sci Q 23(4):582–603
Torunski E, Shafiq MO, Whitehead A (2017) Code style analytics for the automatic setting of formatting rules in ides: a solution to the tabs vs. spaces debate. In: 2017 Twelfth International Conference on Digital information management (ICDIM). IEEE, pp 6–14
Wohlin C, Runeson P, Host M, Ohlsson MC, Regnell B, Wesslèn A (2000) Experimentation in software engineering: an introduction. Kluwer Academic Publisher, Boston



**Miroslaw Ochodek** is an assistant professor at Poznan University of Technology. His research interests are measurement and predictions, requirements engineering, and empirical software engineering.

**Regina Hebig** is an Associate Professor in Software Engineering at Chalmers and the University of Gothenburg. She did her PhD at the University of Potsdam, Germany in 2014. Her research interests include software evolution, software modelling, and software processes. Currently, Regina is the director of the master education in software engineering at Chalmers and the University of Gothenburg.

**Wilhelm Meding** is a senior measurement program leader. Leads a metrics team and an analytics team. 20% of his time is spent on software metrics research. Has published one book and more than 50 papers.

**Gert Frost** is a DevOps Project Manager with a long record of working with Software in the mechanical and industrial engineering industry. Working areas are Agile SW Development, Software Product Line Engineering, Continuous Integration/Continuous Delivery, Automatic Testing, Metrics and Data Visualization preferably in collaborating with external partners and academia both in DK and abroad. I prefer to work in areas where leadership and management goes hand in hand with strategy and execution, for driving the development and changes to always become smarter and better in what we deliver for our customers.

**Miroslaw Staron** is Professor in the Department of Computer Science and Engineering at the University of Gothenburg, Sweden. He has published extensively on software metrics, model-driven software development and empirical software engineering and cooperates with Ericsson, Volvo and other telecom companies and car manufacturers. He has written two books about automotive software development and about measurement.

## Affiliations

**Miroslaw Ochodek[1,2] · Regina Hebig[3] · Wilhelm Meding[4] · Gert Frost[5] · Miroslaw Staron[3]**

1 Poznan University of Technology, Poznan, Poland

2 University of Gothenburg, Gothenburg, Sweden

3 Computer Science and Engineering Department, Chalmers | University of Gothenburg, Gothenburg, Sweden

4 Ericsson AB, Gothenburg, Sweden

5 Grundfos, Bjerringbro, Denmark