Ein paralleler Algorithmus für API Mining von C# Code

Robert Horkovics-Kovats (M. Sc.)
Capgemini SE
Bahnhofstraße 30, 90402 Nürnberg
robert.horkovics-kovats@capgemini.com

Dr. Eldar Sultanow
Capgemini SE
Bahnhofstraße 30, 90402 Nürnberg
eldar.sultanow@capgemini.com

Prof. Dr.-Ing. Frank Herrmann
OTH Regensburg
Innovationszentrum für Produktionslogistik
und Fabrikplanung (IPF)
Galgenbergstraße 32, 93053 Regensburg
frank.herrmann@oth-regensburg.de

Zusammenfassung

Die Konformitätsanalyse ist eine Technik der statischen Code-Analyse (SCA) zur Software-Qualitätssicherung. Ihr Kernproblem ist, dass Werkzeuge nicht aus bereits eingetretenen Fehlern automatisiert dazulernen. Zur Lösung wurde in dieser Arbeit das maschinelle Lernen (ML) evaluiert, indem ein wissenschaftlich fundierter und praktisch erprobter Ansatz zur unüberwachten Lerntechnik angewandt und das Ergebnis analysiert wurde. Es wurde festgestellt, dass zur Anwendung auf verschiedene Programmiersprachen nur ein sprachspezifisches API Mining-Tool notwendig ist. Ein derartiges Tool durchsucht in parallelisierter Form Codezeilen und normalisiert sie für maschinelle Lernprozesse. Dieses System wurde für die Programmiersprache C# implementiert, da viele Industrieprojekte in dieser Sprache entwickelt werden. Zur funktionalen Validierung wurde in einer Fallstudie gezeigt, dass Regeln mit einem positiven Effekt auf Software-Qualität gelernt wurden. Konkret wurde der Wartungsaufwand eines Code-Smells in einem Beispielprojekt durch das Auslagern einer gelernten Assoziation in eine gemeinsame Methode um den Faktor 30 reduziert. Die Laufzeit des Algorithmus wurde empirisch in acht open-source Repositorys evaluiert. Durch Parallelisierung kann eine durchschnittliche Laufzeitverbesserung von 45,16% erwartet werden. Allerdings wurden bei der Anwendung auch Grenzen deutlich: Viele Assoziationen sind nutzlos, die Regelbewertung ist von einem subjektiven Faktor abhängig und die Wirtschaftlichkeit des Tools ist deshalb nicht transparent. Dennoch belegt diese Arbeit, dass ein ML-basiertes SCA-Tool als ergänzende Qualitätssicherungsmaßnahme im Software-Engineering möglich ist.

I. Einleitung

Eine Methode zur Qualitätssicherung im Software-Engineering ist die statische Code-Analyse (SCA). Dabei wird der Code durch Sichtung der Quelltexte untersucht, ohne diese in ein ausführbares Programm zu übersetzen. Eine konkrete Technik ist die Konformitätsanalyse, wo der Code auf vordefinierte Syntax-, Grammatik- und Semantikregeln auf Konformität geprüft wird. Ihr Kernproblem ist, dass sie nicht aus bereits eingetretenen Fehlern automatisiert dazulernt: Fehler müssen erst im Produktivbetrieb eintreten, bevor daraus manuell neue Regeln identifiziert werden können [Sult18]. Zur Lösung wurde in dieser Arbeit das maschinelle Lernen (ML) evaluiert. Ein wesentlicher Bestandteil besteht darin, die Codebasis in eine für ML-Prozesse verwertbare Datengrundlage zu normalisieren.

Aufbau des Manuskripts

Der Artikel ist wie folgt strukturiert: Zuerst wird das Problem anhand der Situation in einem Unternehmen präzisiert. Daraufhin werden die relevanten theoretischen Grundlagen zur Assoziationsanalyse dargelegt. Damit wird ein wissenschaftlich fundierter und praktisch erprobter ML-basierter SCA-Ansatz angewandt und das Ergebnis analysiert. Das Konzept eines parallelisierten API Miners für die Programmiersprache C# wird vorgestellt. Seine Funktionalität wird in einer Fallstudie validiert und die Laufzeit unter Parallelisierung empirisch untersucht. Die Ergebnisse werden als Nächstes kritisch hinterfragt und diskutiert. Abschließend wird ein Fazit mit Ausblick gezogen, dass ein ML-basiertes SCA-Tool als ergänzende Qualitätssicherungsmaßnahme im Software-Engineering möglich ist.

II. Problemstellung, Ziel und Lösungsansatz

Ein globaler Technologiekonzern, der vor allem auf die Elektrifizierung, Automatisierung und Digitalisierung insbesondere in den Bereichen Stromerzeugung und -übertragung, Medizintechnik, Infrastruktur und industrielle Anwendungen ausgerichtet ist, verfügt über viele in der Programmiersprache C# implementierte Projekte. In den Projekten werden klassische SCA-Werkzeuge wie SonarQube zur Software-Qualitätssicherung eingesetzt. Dabei definiert ein Architekt oder Software-Engineer Prüfregeln vorab, das Werkzeug validiert gegen diese Regeln und die Entwickler beheben, falls vorhanden, Regelverstöße. Treten neue Fehler im Produktivbetrieb auf, ergänzt der Architekt/Software-Engineer neue Regeln, um diese Fehler in Zukunft verhindern zu können. Die eingesetzte Methode weist die zuvor beschriebene Schwäche der Konformitätsanalyse auf: Neue Regeln sind erst a-posteriori, also nach dem mit Kosten verbundenem Eintritt des dazugehörigen Fehlers, bekannt. Die genannte Frage wurde auf diese Projekte angewandt. Sie lässt sich wie folgt präzisieren: Wie lässt sich mittels API-Mining C# Code so normalisieren, dass es für ML-basierte SCA verwertbar ist?

Das Ziel dieser Arbeit ist es, ein API-Mining-Prototyp zu entwickeln, das in parallelisierter Form C# Codezeilen durchsuchen und für maschinelle Lernprozesse aufbereiten (normalisieren) kann. Die normalisierten Daten enthalten Transaktionen mit Items, die eine Regelidentifikation anhand der unüberwachten Lerntechnik *Assoziationsanalyse* ermöglichen. Das Prinzip ist wissenschaftlich fundiert und praktisch erprobt [Sult18] und soll auf den Daten anwendbar sein, die aus der, mit der vorliegenden Arbeit angestrebten, Normalisierung resultieren.

III. Unüberwachte Lerntechnik und strukturierte Datenrepräsentation

Die Assoziationsanalyse (engl. Assocation Rule Mining) ist eine unüberwachte Lerntechnik des ML und wurde zuerst 1993 von Agrawal et al. als Warenkorbanalyse vorgestellt [Agra93]. Unüberwachte Ansätze zeichnen sich dadurch aus, dass sie auf einer Datenmenge ohne Labels operieren. Es wird in protokollierten Verkaufsdaten analysiert, welches Produkt häufig mit einem Anderen gekauft wird. Es werden Assoziationsregeln mit der Aussage „Wer Produkt A kauft, kauft häufig auch Produkt B“ gesucht.

Das Prinzip wird an folgendem Beispiel erklärt:

Tabelle 1: Warenkorbanalyse Beispiel

Transaktion	Smartphone	PC	PC-Bildschirm
1	x	x	x
2	x	x	x
3		x	x
4		x	

Hier sind vier Einkäufe protokolliert. Sie werden als Transaktionen bezeichnet. Jede Transaktion T_i beinhaltet ihre eingekauften Produkte, sogenannte Items. Die Transaktionsdatenbank zeigt, dass gemeinsam mit einem PC in 75% auch ein PC-Bildschirm gekauft wurde. Dieser Zusammenhang kann als Assoziationsregel R formuliert werden. Wird Item(-menge) A gekauft, besteht eine Wahrscheinlichkeit von x%, dass auch Item(-menge) B

gekauft wird. Diese Regel trifft auf y% aller Transaktionen zu. Formal notiert:
Sei I die Menge aller Items i_1, i_2, ..., i_n. Es gilt:

$$R : A \rightarrow B \ \ [support : y\%] \ \ [confidence : x\%],$$
$$\text{wobei } \ A \subset I, B \subset I \ \text{ und } \ A \cap B = \varnothing$$

WEKA ist ein open-source Data-Mining-Tool, das an der University of Wakato in Neuseeland entwickelt wird. Der Name steht für Wakato Environment for Knowledge Analysis. Es bietet erprobte und getestete ML-Software, auf die über eine grafische Oberfläche, über Kommandozeilenapplikationen oder über eine Java API zugegriffen werden kann. Es wird zum Lehren, zur Forschung und in industriellen Anwendungen eingesetzt und enthält eingebaute Tools für Standardproblemstellungen aus dem maschinellen Lernen [Weka20].

WEKA erwartet Input-Dateien in einem standardisierten Format, dem sogenannten Attribute-Relation File Format (ARFF). Es beschreibt eine Liste von Instanzen, die eine Menge von Attributen teilen. Die Struktur besteht aus den Sektionen Kopf (Header) und Daten (Data). Im Kopf werden Attribute mit Datentypen deklariert. Im darunter folgenden Körper werden Dateninstanzen aufgelistet. Das Format ermöglicht die strukturierte Repräsentation von zuvor unstrukturierten oder semistrukturierten Daten, wodurch die Anwendung von ML-Algorithmen vereinfacht wird.

```
1  @RELATION weather
2
3  @ATTRIBUTE temperature NUMERIC
4  @ATTRIBUTE humidity {high, normal, low}
5  @ATTRIBUTE note STRING
6  @ATTRIBUTE timestamp DATE "yyyy/MM/dd"
7
8  @DATA
9  25.3,high,'slightly cloudy',"2019/05/22"
10 31.1,normal,'cloudless and hot',
       "2019/06/26"
```

Listing 1: ARFF Beispiel

IV. Analyse eines ML-basierten SCA-Ansatzes

Die Assoziationsanalyse untersucht Items innerhalb eines Gruppierungskriteriums. Dieses Prinzip muss zur statischen Code-Analyse auf den Quelltext übertragen werden. Um Assoziationen über Schnittstellen zur Programmierung von Anwendungen (APIs) zu lernen, schlagen Sultanow et al. in [Sult18] vor, Methodeninvokationen (Calls) als Items zu betrachten. Es werden Beziehungen zwischen Methoden gesucht, die mit einer bestimmten Wahrscheinlichkeit zusammen aufgerufen werden. Als Gruppierungskriterium wird die dazugehörige aufrufende Methodendeklaration (Caller) festgelegt. Der Zusammenhang wird im folgenden Codebeispiel verdeutlicht:

```
1  int caller()           // Transaction
2  {
3    call1();             // Item 1
4    int x = call2();     // Item 2
5    x += 1;
6    call3(5);            // Item 3
7    return call4() + x; // Item 4
8  }
```

Listing 2: Methodendeklaration mit Methodeninvokationen

Assoziationsregeln werden aus einer verifizierten Codebasis gelernt. Neuer Code wird daraufhin gegen diese Regeln auf Verletzungen geprüft. Eine Transaktion erfüllt eine Regel $R : A \rightarrow B$, wenn sie die disjunkten Itemmengen A und B vollständig enthält. Fehlt auch nur ein Item aus diesen Mengen, liegt eine Regelverletzung vor. Im Kontext der Code-Analyse muss also eine erwartete Invokation in einer gelernten Regel fehlen, damit im untersuchten Code eine Verletzung gemeldet wird.

Weiterhin stellen Sultanow et. al in [Sult18] eine theoretische Ablaufsequenz auf, wie das vorgestellte Konzept in eine Continuous Integration Pipeline integriert werden kann (s. Abbildung 1). Darin arbeiten drei Instanzen zusammen. Erst holt ein *API Miner* den Quellcode aus einem Versionskontrollsystem wie Git und extrahiert die Inputdaten in ein Dateiformat, das für den ML-Vorgang verwertbar

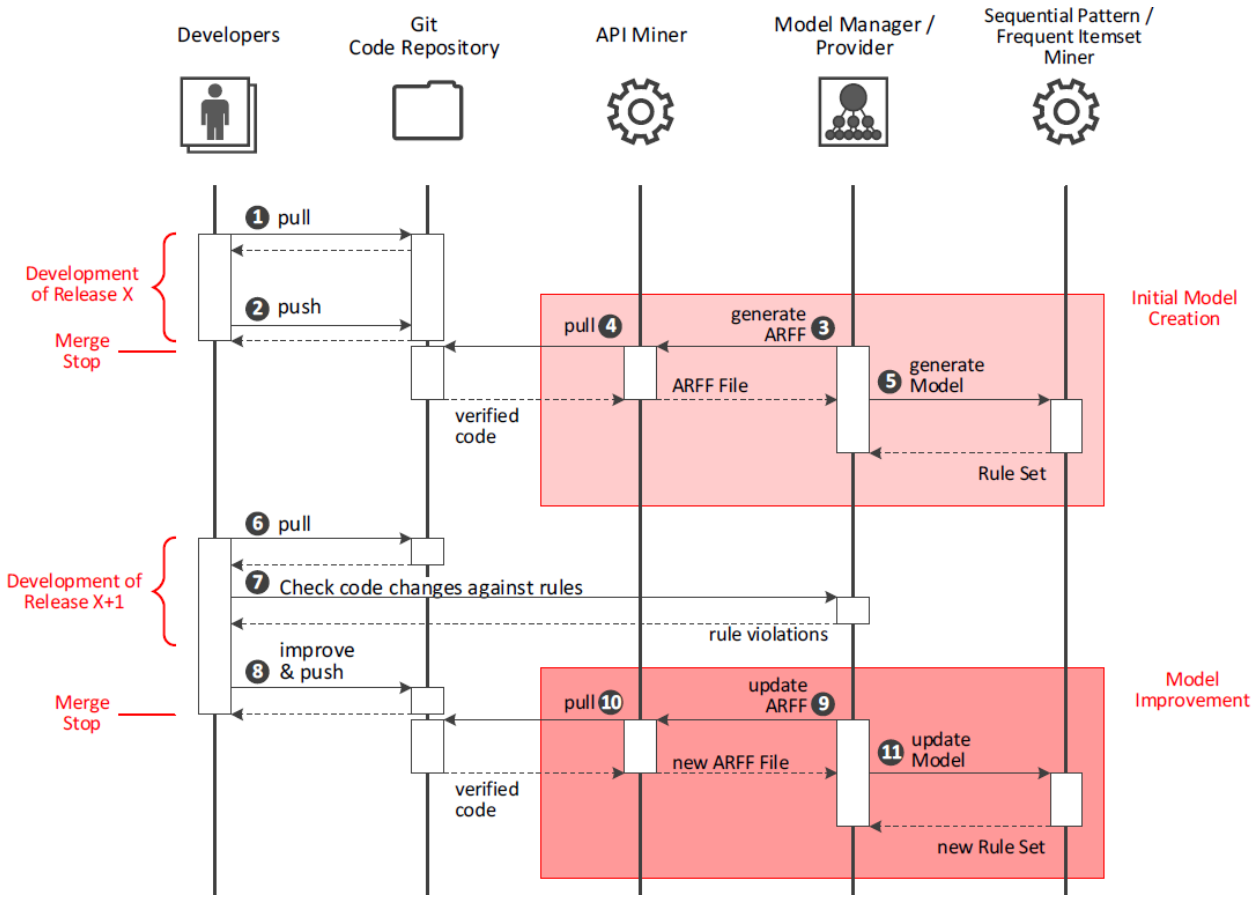

Abbildung 1: Theoretische Ablaufsequenz des ML-basierten SCA-Prototyps [Sult18]

ist. Ein *Model Manager* verwaltet das ML-Modell, indem der Manager es durch Aufruf der Algorithmen trainiert und Regeln auf Verletzungen prüft. Der *Pattern Miner* implementiert die Algorithmen zur Assoziationsanalyse. Da sowohl zweite als auch dritte Instanz mit Daten arbeiten, die unabhängig von der zugrundeliegenden Programmiersprache sind, ist nur der API Miner sprachspezifisch. Damit also eine statische Code-Analyse nach diesem Prinzip für C# möglich wird, muss nur ein API Miner entworfen werden, der C# Code extrahiert und in ein standardisiertes Datenformat transformiert.

V. Ein paralleler Algorithmus für API Mining von C# Code

Der C# API Miner wird als Konsolenapplikation entwickelt. Es verarbeitet Userinput und transformiert die geforderten Daten einer übergebenen API in eine ARFF-Datei.

Algorithmus 1: API Mining in C#

Eingabe:	C# API in Form einer Projektmappe (Solution), eines Projekts (Project), eines Unterverzeichnisses oder einer konkreten Quelldatei
Anweisungen:	Eine API wird zur weiteren Verarbeitung geladen. Es wird über ihre Elemente iteriert und für jedes Dokument der Syntaxbaum analysiert.

Aus diesem Objektmodell werden alle Methodendeklarationen des Dokuments gesammelt. Anschließend werden die Invokationen pro Deklaration gesucht und ihre Namen vollqualifiziert aufgelöst. Fehlerhaft aufgelöste Einträge werden mit „UNRESOLVED“ gekennzeichnet.

Ausgabe: ARFF-Datei mit extrahierten API-Daten

Zur Implementierung müssen Projektmappen mit ihren Abhängigkeiten kompiliert, Quelltexte in abstrakte Syntaxbäume geparst und Methodennamen zur eindeutigen Identifikation vollqualifiziert aufgelöst werden. Derartige Syntax- und Semantikanalysen sind in C# mithilfe des .NET Compiler Platform-SDK (Roslyn APIs) möglich [Wagn20]. Die extrahierten Daten müssen die aufgelösten Namen von Methodendeklarationen mit dazugehörigen Methodeninvokationen enthalten. Die Struktur der ARFF-Datei enthält somit zwei Spalten, die diese Caller und Calls als Zeichenketten halten. ARFF unterstützt bis dato keine Datenkollektionen [Univ08], weshalb die Auflistung der Invokationen als Text angegeben wird. Als Trennzeichen wird das Leerzeichen eingesetzt:

```
1 @RELATION myProject
2
3 @ATTRIBUTE caller STRING
4 @ATTRIBUTE calls STRING
5
6 @DATA
7 'dummy.MyClass.caller1()','dummy.MyClass.x
      dummy.MyClass.y'
8 'dummy.MyClass.caller2()','System.out.
      println'
```

Listing 3: ARFF-Inhalt bei ML-basierter SCA

Da die Genauigkeit bzw. Aussagekraft von ML-Algorithmen höher ist, wenn sie auf hohen Datenmengen angewandt wird [Hurw18, S. 8], wird die Parallelisierung des Algorithmus gefordert. Dabei nutzt das Programm die Multikernarchitektur von modernen Rechnern aus, um Laufzeitverbesserungen bei zunehmender Input-Größe zu erzielen. Im Algorithmus liegen dafür drei verschachtelte Iterationsstufen vor:

- Stufe 1: Iteration über Projekte einer Projektmappe
- Stufe 2: Iteration über Dokumente eines Projekts
- Stufe 3: Iteration über Methoden eines Dokuments

Jede dieser Stufen kann von Parallelisierung profitieren. Jedoch ist unklar, welche Ebenen zur höchsten Beschleunigung führen [Toub10, S. 28–29]. Deshalb kann die Parallelisierung über optionale Input-Parameter gesteuert werden, wodurch die Konstellation mit der höchsten Laufzeitverbesserung abhängig von der Eingabe erwählt werden kann. Um den generellen Effekt der Parallelisierung zu schätzen und Konfigurationsempfehlungen vorzuschlagen, wurde die Performance des API Miners empirisch untersucht.

VI. Funktionale Validierung

Zur funktionalen Validierung wird der C# API Miner dem ML-Prozess von Sultanow et al. aus [Sult18] (s. Abbildung 1) unterzogen. Der API Miner gilt als validiert, wenn damit mindestens eine Assoziationsregel in C# gelernt werden kann, die einen nachweisbar positiven Effekt auf die Software-Qualität der untersuchten Projektmappe hat. Um den abstrakten Begriff *Software-Qualität* zu präzisieren, wird mithilfe der Qualitätseigenschaften des standardisierten ISO-25010-Modells der Nutzen einer Regel begründet. Die Qualitätsmerkmale umfassen funktionelle Eignung, Performance Effizienz, Kompatibilität, Benutzbarkeit, Zuverlässigkeit, Sicherheit, Wartbarkeit und Portabilität [ISO20].

Als Fallbeispiel wird das open-source Repository mit der ID 4 (AI-Programmer) betrachtet.

Tabelle 2: Regelbefund aus dem Repository 4 mit positivem Effekt auf Software-Qualität

Repo	Rule	Supp.	Conf.
4	Math.Abs → Fitness-Base.IsFitnessAchieved	31	100%

Der Regelbefund weist einen absoluten Support-

Wert von 31 und eine Konfidenz von 100% auf. Das heißt, dass diese Invokationsreihe insgesamt 31-mal im Code auftritt und Antezedenz- und Konsequenz-Invokation stets gemeinsam als Paar vorkommen. Die beiden Methoden können deshalb in eine gemeinsame Methode ausgelagert werden, da sie einer logischen Einheit entsprechen. Dadurch konnten Wartungsaufwände an dieser Codestelle um den Faktor 30 reduziert werden.

Das kurze Beispiel beweist die funktionale Eignung des C# API Miners: Relevante API-Daten des Repositorys *AI-Programmer* wurden extrahiert, sodass eine Assoziationsanalyse darauf ausgeführt werden konnte. Dadurch wurden Assoziationsregeln gelernt, die der statischen Code-Analyse zur Qualitätssicherung dienen.

VII. Empirische Untersuchung der Performance

In der empirischen Laufzeituntersuchung wurde der API Miner neben sieben weiteren open-source Repositorys betrachtet (s. Tabelle 3). Insgesamt umfasst die Analyse knapp 400.000 Zeilen Code. Der Algorithmus wurde in jeder Konfiguration zehnmal auf einem sonst stillen System ohne Hintergrundprozesse ausgeführt, um mittlere Laufzeitwerte mit Standardabweichungen zu erhalten.

Tabelle 3: Acht untersuchte Repositorys mit Lines of Code

ID	Repo	LoC
1	Unitiy Reference	340733
2	ScriptCS	18597
3	Algorithms-in-CSharp	17057
4	AI-Programmer	8124
5	Easy-http	4172
6	APIMiner	4007
7	Domain-Driven-Design-Example	3353
8	Design-Patterns	3045

Es wurden fünf Code-Metriken erfasst, um zu bewerten, wie repräsentativ die Repositorys hinsichtlich ihrer Merkmalsausprägungen sind:

Tabelle 4: Erhobene Code-Metriken von acht C#-Projektmappen

Repo	LoC	P	D	M	M/D
1	340733	4	2870	37085	12,92
2	18597	10	274	1155	4,22
3	17057	5	125	290	2,32
4	8124	10	76	170	2,24
5	4172	2	71	278	3,92
6	4007	4	33	186	5,64
7	3353	3	124	189	1,52
8	3045	17	177	269	1,52

Legende:
LoC Lines of Code
P Anzahl Projekte
D Anzahl C#-Dokumente
M Anzahl Methoden
M/D Anzahl Methoden pro Dokument

Aus den gemessenen Daten kommt hervor, dass eine Kategorisierung nach Anzahl von Lines-Of-Code-Ziffern heuristisch wertvolle Vergleichswerte zur Gesamtlaufzeit bietet. So dauert der Algorithmus beispielsweise für Projektmappen mit vierstelligen Codezeilenzahlen ca. 5 bis 6 Sekunden:

Tabelle 5: Sequenzielle Laufzeit des C# API Miners in Sekunden

Repo	Load	Mine	Write	Total
1	02,932	20,080	07,790	30,908
2	04,037	04,392	00,270	08,790
3	03,112	03,830	00,077	07,114
4	02,860	02,821	00,047	05,824
5	02,605	02,801	00,080	05,577
6	03,168	03,095	00,060	06,415
7	02,648	02,702	00,057	05,502
8	03,603	02,456	00,058	06,211

Das Öffnen bzw. Laden von Projektmappen dauert in der Stichprobe zwischen 2 und 4 Sekunden. Diese Werte resultieren je nach Projektgröße in stark unterschiedlichen Anteilen bzgl. der Gesamtlaufzeit. 2 bis 4 Sekunden entsprechen bei kleinen Projektmappen ca. 50% und bei der größten untersuchten Projektmappe knapp 10%. Demnach ist für alle Re-

positorys das Öffnen der Projektmappe und damit der Einsatz von Roslyn eine laufzeitintensive Aktion.

Die Laufzeitdaten der tiefsten Iterationsstufe zeigen für sieben von acht Repositorys eine Laufzeitverbesserung (s. Spalte M in Tabelle 6). Es wird in der tiefsten Stufe somit genug Arbeit parallelisiert, sodass sich die Kosten dafür amortisieren. Es wird kein Antipattern begangen. Die Ausnahme dieser Beobachtung ist die kleinste Projektmappe. Der Anwendungsfall bestätigt, dass die alleinige Parallelisierung dieser Stufe nicht genügt, um für beliebige Projekte mit unterschiedlichsten Merkmalen Laufzeitverbesserungen zu erzielen. Parallelisierungsoptionen für die höheren Iterationsstufen sind dafür notwendig.

Tabelle 6 zeigt die prozentuale Laufzeitverbesserung des Code-Abschnitts *Mine* in den acht Repositorys. Da die Parallelisierung über drei optionale boolesche Parameter gesteuert wird, liegen $2^3 - 1 = 7$ Optionen vor. Sie werden mit ihren Initialen beschriftet. So steht M für die Parallelisierung der Methoden, D für Dokumente, P für Projekte und beispielsweise der Ausdruck D+P für die Kombination dieser Parameter.

Grün-markierte Zellen zeigen die Konstellation mit der höchsten Verbesserung pro Zeile. Die Konstellation mit den meisten grünen Einträgen ist D+P. In fünf von acht Repositorys wurden hier die höchsten Verbesserungen erzielt. Weiterhin beinhalten die effizientesten Konstellationen stets die Parallelisierung von Projekten (P). Die Parallelisierung auf der höchsten hierarchischen Programmstufe bietet somit den größten Nutzen. Durchschnittlich kann für die schnellste Konstellation eine Verbesserung von 45,16% erwartet werden. Die rot-markierten Zellen zeigen Fälle mit Laufzeitverlangsamungen. Außer dem Wert -1,33% vom Repository 8(M), finden sich diese Negativfälle erneut bei der D+P-Konstellation. Werden diese Repositorys mit ihren erhobenen Code-Metriken (s. Tabelle 4) gegenübergestellt, fällt ein Zusammenhang zwischen hoher Projektanzahl (>10) und Laufzeitverschlechterung auf. Daraus kann abgeleitet werden, dass für Projektmappen mit vielen Projekten ein geringerer Parallelisierungsgrad empfohlen wird. In diesen Fällen bietet die alleinige Parallelisierung der Projektverarbeitung (P) den größten Nutzen.

VIII. Limitationen der vorgestellten Lösung

Zwar wurde anhand des Fallbeispiels Nutzpotenzial im ML-basierten Ansatz demonstriert, muss dennoch die Sinnhaftigkeit des Werkzeugs kritisch hinterfragt werden. In vier der acht untersuchten Repositorys wurden bis zu 99,35% der Regeln aus Testcode gelernt. Dieses dominante Vorkommen lässt sich über die Art, wie Software-Tests geschrieben werden, begründen: Bei Komponententests o.ä. ist es üblich, mithilfe von Invokationen Daten vorzubereiten, Platzhalterobjekte (Mocks) zu erstellen und Behauptungen (Asserts) zu prüfen. Das führt zu vielen Test-Invokationen, die gemeinsam mit dem eigentlichen Geschäftscode aufgerufen werden. In Folge wurden deshalb überwiegend Assoziationen

Tabelle 6: Prozentuale Laufzeitverbesserung der Parameterkonstellationen

Repo	M	D	P	M+D	M+P	D+P	M+D+P
1	26,58%	39,33%	27,28%	38,55%	41,90%	49,38%	46,82%
2	8,75%	30,42%	45,28%	32,27%	46,11%	21,27%	15,30%
3	22,20%	33,92%	45,87%	29,71%	47,33%	48,66%	47,97%
4	16,50%	5,09%	46,93%	22,62%	44,98%	-0,57%	-20,27%
5	11,61%	37,68%	32,95%	36,14%	37,70%	46,92%	41,99%
6	12,90%	33,14%	39,32%	31,17%	43,11%	45,39%	39,50%
7	1,23%	23,26%	35,25%	22,45%	37,71%	37,94%	33,92%
8	-1,33%	23,04%	39,95%	19,67%	37,74%	-32,62%	-23,01%

aus diesem Segment gelernt. Diese Regeln können jedoch nicht als sinnvoll eingestuft werden: Eine Regelverletzung im Geschäftscode, die ein fehlendes *Assert.Equals* meldet, ist stets falsch-negativ, da diese Invokation nur in Testklassen relevant ist. Derartige Regeln werden als Noise (Lärm) bezeichnet, die die Suche nach nützlichen Regeln erschweren. Darüber hinaus erweist sich die Bewertung von gelernten Assoziationsregeln als schwierig, da Interessantheit subjektiv ist. Beispielsweise liefert eine Regel $R : Console.WriteLine() \rightarrow string.Format()$, die einen Zusammenhang zwischen Konsolenausgabe und Textformatierung ausdrückt, kaum fachlichen Mehrwert, auch wenn ihr absolutes Vorkommen hoch ist (Support) und die Regel oft Anwendung findet (Confidence). Es gibt keinen nachgewiesenen Zusammenhang zwischen objektiver und subjektiver Interessantheit [Geng06], weshalb eine Regel trotz hohem Support-, Confidence- oder Korrelationswert (Lift) dennoch nutzlos sein kann. Regeln müssen also aufwändig, beispielsweise mithilfe von Black- und Whitelists, verwaltet werden. Es ist unklar, ob der nachgewiesene Nutzen diesem Aufwand überwiegt. Die Wirtschaftlichkeit des vorgestellten Ansatzes ist nicht transparent und muss zur Analyse der Marktreife in einer weiterführenden Forschung untersucht werden.

IX. Fazit und Ausblick

Die statische Code-Analyse ist in ihrer Natur sprachspezifisch. Allerdings verarbeiten die eingesetzten Algorithmen zur Assoziationsanalyse Daten in einem standardisierten Datenformat. Somit muss je Sprache nur ein sprachspezifischer API Miner entwickelt werden, um den Ansatz für die korrespondierende Sprache zu ermöglichen.

Durch Analyse eines erprobten ML-basierten SCA-Ansatzes wurden die Anforderungen an ein API Mining Tool verstanden, das Regeln lernt, die der statischen Code-Analyse in C# dienen. Es normalisiert Methodendeklarationen zu Transaktionen und dazugehörige Methodeninvokationen zu Items und setzt das Datenformat ARFF zur Abbildung der Transaktionsdatenbank ein. Moderne Programmiertechniken zur Ausnutzung einer Multikernarchitektur und zum Zugriff auf Kompilierinformationen über Roslyn wurden eingesetzt, um die gestellten Anforderungen zu implementieren. Bei der Güteevaluation wurden Funktionalität und Performance der Lösung untersucht. Dabei wurde der Algorithmus auf acht Repositorys mit knapp 400.000 Codezeilen jeweils zehnfach ausgeführt, um mittlere Laufzeitergebnisse zu erhalten. Mithilfe des vorgestellten API Miners ist es möglich, Regeln zu lernen, die einen positiven Effekt auf die Software-Qualität haben. Die höchsten Performancegewinne können bei der Parallelisierung der obersten Ebene entlang der Codehierarchie (Parallelisierung der Projektverarbeitung) erwartet werden. Aus der Stichprobe kommt eine durchschnittliche Laufzeitverbesserung von ca. 45% in der optimalen Parameterkonstellation hervor. Aus einer kritischen Perspektive betrachtet, liefert die Assoziationsanalyse viel Noise und die Bewertung von gelernten Regeln ist aufgrund von subjektiver Interessantheit schwierig. Die Wirtschaftlichkeit der Gesamtlösung ist deshalb nicht transparent.

Zusammengefasst wurde in dieser Kurzpublikation unter anderem ein Ansatz vorgestellt, wie maschinelles Lernen mit statischer Code-Analyse zur Verbesserung von Software-Qualität kombiniert werden kann. Im Tool wurde Nutzpotenzial nachgewiesen, gleichzeitig wurden dabei seine Grenzen deutlich. Daraus kann abgeleitet werden, dass die innovativen ML-basierten Ansätze die klassische SCA nicht ersetzen werden, sondern als ergänzende Qualitätssicherungsmaßnahme in Erwägung gezogen werden können. Für die Zukunft wird ausgehend von den Ergebnissen dieser Arbeit prognostiziert, dass sowohl klassische als auch intelligente SCA gemeinsam genutzt werden, mit dem Ziel, die steigenden Ansprüche an Software-Qualität zu erfüllen und daraus einen Wettbewerbsvorteil zu generieren. Die intensive Forschung im maschinellen Lernen lässt vermuten, dass die genannten Schwierigkeiten überwunden werden können.

Literatur

[Agra93] Rakesh Agrawal, Tomasz Imieliński und Arun Swami: "Mining associati-

on rules between sets of items in large databases". In: *ACM SIGMOD Record* 22.2 (1993), S. 207–216.

[Geng06] Liqiang Geng und Howard J. Hamilton: "Interestingness measures for data mining". In: *ACM Computing Surveys* 38.3 (2006), 9–es.

[Hurw18] Judith Hurwitz und Daniel Kirsch: *Machine Learning: for dummies*. John Wiley & Sons, Inc., 2018.

[ISO20] *ISO 25010*. Zugriff am 04.03.2020. 4.3.2020. URL: https://iso25000.com/index.php/en/iso-25000-standards/iso-25010.

[Sult18] Eldar Sultanow, Stefan Konopik, André Ullrich und Gergana Vladova: "Machine Learning based Static Code Analysis for Software Quality Assurance". In: (2018). Hrsg. von IEEE.

[Toub10] Stephen Toub: *Patterns of Parallel Programming: Understanding and Applying Parallel Patterns with the .NET Framework 4 and Visual C#*. 2010.

[Univ08] University of Waikato: *Attribute-Relation File Format (ARFF)*. Zugriff am 18.02.2020. 1.11.2008. URL: https://www.cs.waikato.ac.nz/ml/weka/arff.html.

[Wagn20] Bill Wagner: *Das .NET Compiler Platform SDK (Roslyn APIs) | Microsoft Docs*. Zugriff am 28.02.2020. 28.2.2020. URL: https://docs.microsoft.com/de-de/dotnet/csharp/roslyn-sdk/.

[Weka20] *Weka 3 - Data Mining with Open Source Machine Learning Software in Java*. Zugriff am 18.02.2020. 2020. URL: https://www.cs.waikato.ac.nz/ml/weka/.