Capgemini

Sonderdruck aus
BI-SPEKTRUM 4/2018

Bild: Shutterstock

Bestimmung von Assoziationsregeln zur statischen Codeanalyse

Maschinelles Lernen in der Codeanalyse bei der Bundesagentur für Arbeit

Ein Beitrag von
Eldar Sultanow,
Stefan Konopik und
Artur Biedulski

Die Assoziationsanalyse ist ein maschinelles Lernverfahren aus dem Bereich des unüberwachten Lernens, das ursprünglich durch Anwendungen aus dem Handelsumfeld motiviert ist. Ihr liegt das Prinzip zugrunde, bedingte Wahrscheinlichkeiten des Erwerbs eines Produkts A unter der Bedingung des Erwerbs eines anderen Produkts B zu ermitteln. Das Verfahren wird hier auf die statische Codeanalyse (SCA) zur Identifizierung potenzieller Fehler eines missionskritischen Großsystems im öffentlichen Sektor angewendet. Verzögert sich dort die Fehlerbereinigung, kann es je nach Schwere des Fehlers teuer werden – ein Hotfix ist dann unabdingbar. Welches Vorgehen sich für die statische Codeanalyse in einem derart kritischen Umfeld besonders eignet, beleuchtet der Artikel anhand eines eigens entwickelten neuen Ansatzes einschließlich eines Prototypen und gibt einen Ausblick auf die weiteren Potenziale maschinellen Lernens in der Codeanalyse.

In großen kritischen Projekten sind die Qualitätsanforderungen immens. Die klassische SCA stößt hier schnell an ihre Grenzen. Mit Machine Learning (ML) erschließen sich neue Wege, darüber hinauszugehen. Dies zeigt das Beispiel „ALLEGRO“: Das IT-Verfahren ALLEGRO ist mit einem monatlichen Auszahlungsvolumen von circa zwei Milliarden Euro (jährlich ~25 Milliarden Euro) an insgesamt 10 Millionen Haushalte beziehungsweise 20 Millionen leistungsberechtigte Personen ein kritisches Großverfahren der Bundesagentur für Arbeit. Das System ALLEGRO hat etwa 50.000 parallele Power-User, wird von rund 80 Entwicklern umgesetzt beziehungsweise weiterentwickelt und umfasst mittlerweile 800.000 Lines of Code (LoC).

Problemstellung: Fehlersuche in 800.000 Code-Zeilen

Die Softwareentwicklung unterliegt bereits einem stark ausgeprägten qualitätsgesicherten Vorgehen und setzt das etablierte SCA-Werkzeug SonarQube ein. Trotzdem lassen sich nicht alle Fehler vor dem Produktivbetrieb abfangen. Je nach Schwere eines

DR. ELDAR SULTANOW ist Architekt bei Capgemini. Er befasst sich mit Unternehmensarchitekturen, Qualitätssicherung und maschinellem Lernen.
E-Mail: eldar.sultanow@capgemini.com

STEFAN KONOPIK arbeitet im IT-Systemhaus der Bundesagentur für Arbeit innerhalb des IT-Verfahrens ALLEGRO. Seine Interessen liegen im Bereich der populationsbasierten Algorithmen und des maschinellen Lernens.
E-Mail: stefan.konopik@arbeitsagentur.de

ARTUR BIEDULSKI ist Masterstudent am Lehrstuhl für Wirtschaftsinformatik, Prozesse und Systeme der Universität Potsdam. Seine Schwerpunkte sind Unternehmensarchitekturen, Qualitätssicherung und Maschinelles Lernen.
E-Mail: biedulski@uni-potsdam.de

Fehlers in der Produktion hat dieser zur Folge, dass ein Hotfix geliefert werden muss.

Bei einem Hotfix im letzten Jahr fiel in den Flurgesprächen der Satz: „Zu dieser fehlerhaften Stelle im Code habe ich ein Déjà-vu – der Fehler kommt mir bekannt vor, einen ähnlichen hatten wir bereits in der Vergangenheit. An vier anderen, ähnlichen Stellen im ALLEGRO-Code ist es mittlerweile konsistent richtig gemacht.“ Es liegt auf der Hand, dass ein Entwickler in 800.000 Zeilen Code es kaum überblicken kann, wenn vier nahezu identische Stellen im System existieren, von denen sein neu hinzugefügter Code nur minimal abweicht. Im Fall dieses Hotfix war es eine fehlende Nullprüfung einer Variablen eines bestimmten Typs, die direkt auf eine bestimmte Variablendeklaration folgt.

Das war der Auslöser der Idee, die dem hier vorgestellten Ansatz zugrunde liegt: Muster, die ein Mensch in einem äußerst umfangreichen Quellcode nicht finden oder überblicken kann, findet ein maschinelles Lernsystem.

Die Idee: Mustererkennung mittels Machine Learning

Wie können Methoden der Mustererkennung bei der Codeanalyse angewendet werden? Die Idee besteht darin, das Prinzip der Warenkorbanalyse auf die statische Codeanalyse zu übertragen. Die Warenkorbanalyse geht von einer Menge O von Items, etwa Objektmengen oder Kaufgegenständen, und einer Menge F aller Transaktionen aus. Jede Transaktion $T = (TID, I)$ umfasst eine Teilmenge der Item-Menge $TID \in F, I \subseteq O$, wie in Abbildung 1 dargestellt.

Ziel dabei ist die Lösung der Aufgabe: Finde Item-Mengen wie etwa „Fisch“ und „Zitrone“, die Bestandteil solcher Transaktionen sind, welche einen gewissen prozentualen Minimalanteil aller Transaktionen ausmachen. Diese Item-Mengen bezeichnet man als sogenannte „häufige Muster“. Diese häufigen Muster sind der Ausgangspunkt für das Finden von Assoziationsregeln wie „auf Fisch folgt Zitrone“ oder gegenständlich formuliert: „wer Fisch kauft, der kauft auch Zitrone“. $R : X \rightarrow Y$ wird dabei als Assoziationsregel mit $X, Y \subseteq O,\ X \cap Y = \emptyset$ bezeichnet. Weiterhin erfüllt eine Transaktion $T = (TID, I)$ die Regel R, wenn die disjunkten Item-Mengen X und Y in ihr vorkommen $(X \cup Y) \subseteq I$.

In Bezug auf den prozentualen Minimalanteil und um das definitorische Gerüst zu vervollständigen, sind drei Kenngrößen zu berücksichtigen [Alp 14; Liu07]:

Der Unterstützungsgrad $support_F(X)$ einer Item-Menge X ist der an der Gesamtzahl von Transaktionen gemessene Anteil jener Transaktionen, die X enthalten.	$support_F(X) = \frac{\lvert\{T \in F \mid T = (TID, I), X \subseteq I\}\rvert}{\lvert F \rvert}$
Der Unterstützungsgrad $support_F(X \rightarrow Y)$ einer Assoziationsregel $R : X \rightarrow Y$ repräsentiert eine statistische Signifikanz der Regel R, ermittelt anhand des an der Gesamtzahl von Transaktionen gemessenen Anteils jener Transaktionen, die $X \cup Y$ enthalten.	$support_F(X \rightarrow Y) = \frac{\lvert\{T \in F \mid T = (TID, I), X \cup Y \subseteq I\}\rvert}{\lvert F \rvert} = support_F(X \cup Y)$
Die Konfidenz $confidence_F(X \rightarrow Y)$ einer Assoziationsregel beschreibt einen Vertrauensgrad für diese Regel. Berechnet wird er anhand des Anteils der $X \cup Y$ enthaltenen Transaktionen, gemessen an der Menge von Transaktionen, die die Item-Menge X enthalten.	$confidence_F(X \rightarrow Y) = \frac{support_F(X \rightarrow Y)}{support_F(X)}$

Was bedeutet dieses Konzept für den Warenkauf nun in Hinblick auf Programmcodes? Kern der Idee ist es, Codeanweisungen, etwa Methodenaufrufe, Variablendeklarationen oder Nullzeiger-Prüfungen, als Items zu betrachten. Transaktionen sind durch Java-Methoden somit umgrenzte Item-Mengen. Basierend auf der Häufigkeit des Vorkommens von

Capgemini

Capgemini

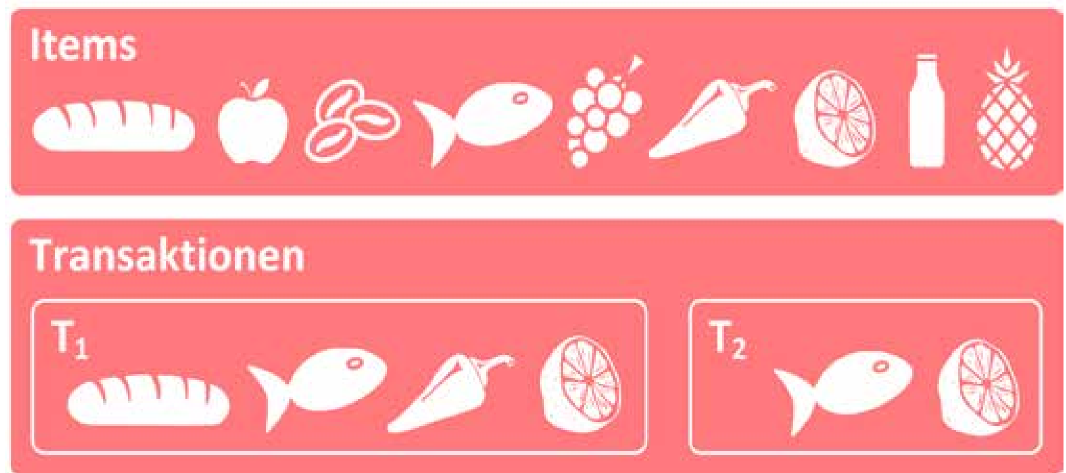

Abb. 1: Transaktionen und Objekte in der Warenkorbanalyse

Codeanweisungen innerhalb der Methoden lassen sich Assoziationsregeln ableiten. Zudem kann ein Mindest-Unterstützungsgrad festgelegt werden, beispielsweise 50 Prozent, sodass Items nur berücksichtigt werden, wenn sie in jeder zweiten Transaktion vorkommen.

Bevor jedoch überhaupt eine Mustererkennung im Code durchgeführt wird, muss die Bemessungsgrundlage festgelegt werden, wann überhaupt ein Code als „ähnlich" betrachtet wird. So ist es weniger sinnvoll, zwei Variablendeklarationen aufgrund des gleichen Namens der Variablen als ähnlich anzusehen. Empfehlenswerter ist es, einen Vergleich anhand ihres Typs vorzunehmen. Um Vergleiche jedoch effizient durchführen zu können, muss der Code entsprechend aufbereitet werden. Jede Variablendeklaration und jeder Methodenaufruf muss demnach vollqualifiziert angegeben sein. Diese Aufbereitung erfolgt bereits zu Beginn. Die gesamte verifizierte Codebasis dient dabei als Input, als Output erhält man eine ARFF-Datei (Attribute-Relation File Format), die den transformierten Code mit vollqualifizierten Angaben enthält. Eine ARFF-Datei ist eine Textdatei mit einer Liste von Datensätzen, deren Attribute der Dateikopf spezifiziert – man kann eine ARFF-Datei also als eine Art strukturierte CSV-Datei auffassen. Das Format wurde an der University of Waikato in Neuseeland für den Einsatz mit der ML-Software Weka entwickelt und ist im Machine-Learning-Umfeld gängig. In dem hier vorgestellten, eigens entwickelten System definiert die ARFF-Datei zwei Attribute, die Transaktions-ID und die Transaktion:

```
@relation example

@attribute transaktions-id string
@attribute transaktion string

@data
,codemining.example.test.Example.main(java.lang.String[])','java.
lang.System.exit
java.lang.System.out.println'
,codemining.example.test.Example.test()','java.lang.System.
currentTimeMillis
java.lang.System.gc java.lang.System.out.println'
```

Die Transaktions-ID ist der vollqualifizierte Name einer Java-Methode, die Transaktion umfasst ebenfalls vollqualifizierte Codeanweisungen innerhalb der Methode, also Items. In der ARFF-Datei sind die Items durch Leerzeichen getrennt. Die ARFF-Datei ermöglicht es, ähnlich einer CSV-Datei, Tabellen mit geringem Speicherbedarf darzustellen. In einem Zwischenschritt werden den in der ARFF-Datei gespeicherten Methodenaufrufen Zahlen zugewiesen, um weiteren Arbeitsspeicher für den Prozess der Ermittlung von Assoziationsregeln (den sogenannten Rule-Mining-Prozess) zu sparen. Auf Basis dieser ARFF-Datei erfolgt das eigentliche Analysieren – das „Code Mining".

Der Algorithmus liefert als Ergebnis häufige Item-Mengen, inklusive Unterstützungsgrad und Assoziationsregeln, einschließlich Unterstützungsgrad und Konfidenz. Diese Regelermittlung geschieht dabei auf Basis eines verifizierten, „sauberen" Codestatus. Neuer, nicht verifizierter Code wird gegen Verletzungen dieser Regeln geprüft. Nachdem sämtliche Quality Gates wie Unit-Tests, Integrationstests und Peer-Reviews durchlaufen wurden, wird er der Codebasis hinzugefügt. Mit der wachsenden Codebasis wird das Modell zunehmend besser, da die Transaktionsdatenbank sowie die Regeln – bei zunehmender Konfidenz – anwachsen. Der Ablauf wird in Abbildung 2 vereinfacht dargestellt. Die Vereinfachung besteht unter anderem darin, dass die Entwicklung von Releases in der Bundesagentur für Arbeit nicht streng sequenziell, sondern teils parallel verläuft – sprich gegen Ende der Entwicklung von Release X wird bereits mit der Entwicklung von Release X+1 begonnen. Zudem erfolgt die Aktuali-

sierung des Modells in der Realität öfter als einmal pro Release, und zwar künftig automatisiert per Jenkins Job.

Im Wesentlichen verdeutlicht Abbildung 2 das Prinzip, dass während der Entwicklung von Releases neuer oder geänderter Code gegen die vorhandenen Regeln geprüft wird. Finden Regelverletzungen statt, prüft der Entwickler diese, korrigiert gegebenenfalls seinen Code und fügt ihn in das Git Repository ein. Ungeachtet dessen unterliegt jeder Push einem Peer-Review durch andere erfahrene Entwickler.

Wie sieht die Lösung aus?

Das System nutzt drei Machine-Learning-Verfahren: das *API Mining* zur Erstellung der ARFF-Datei, das *Sequential Pattern Mining* [Liu07] und das *Frequent Itemset Mining* [RiC13] zur Mustererkennung im aufbereiteten, als ARFF-Datei vorliegenden Code. Verschiedene Open-Source-Frameworks werden dabei kombiniert eingesetzt.

Allerdings ist es deutlich anspruchsvoller, entsprechende Algorithmen umzusetzen, als es auf den ersten Blick scheint. Als praktikabel erweist sich ein Vorgehen, bei dem synthetischer Code als verifizierte Codebasis genutzt wird und – in Anlehnung an den Hotfix im erwähnten Beispiel – an vier ähnlichen Stellen eine Nullzeiger-Prüfung durchgeführt wurde. An einer fünften Stelle im zu verifizierenden neuen Code wurde die Nullzeiger-Prüfung bewusst ausgelassen. Dies war der Lackmustest, ob der Ansatz funktioniert und das System anschlägt. Bei der Umsetzung wurden die entsprechenden Erfahrungen gemacht, wie die algorithmischen Stellschrauben am besten zu bedienen sind. Eine der wichtigsten Erkenntnisse war, dass eine sehr niedrige Untergrenze für Unterstützungsgrad und Konfidenz erforderlich ist, um überhaupt die für den Hotfix relevante Regel zu finden. Über folgende Stellschrauben (näher erläutert in [Alp14; RiC13]) verfügt das System:

- **Minimum Support Training** gibt die Untergrenze der relativen Häufigkeit von Mustern an, die beim Pattern Mining berücksichtigt werden sollen. Es bewirkt, dass Muster mit niedrigerer Frequenz verworfen werden.
- **Minimum Confidence Training** gibt die Sicherheit einer Regel in Prozent an. Die Konfidenz einer Regel wird durch die Formel $conf\ (X \rightarrow Y) = sup\ (X \cup Y)\ /\ sup\ (X)$ berechnet.
- **Max Antecendents Training** gibt die maximale Anzahl an Itemsets innerhalb des Musters X an. Muster, die oberhalb der Grenze liegen, werden ausgeschlossen. Jede Teilmenge eines ausgeschlossenen Musters unterhalb der Grenze bleibt dennoch bestehen. Je höher der Wert, desto länger beträgt hierbei die Rechenzeit.
- **Max Consequents Training** gibt die maximale Anzahl an Itemsets innerhalb des Musters Y an. Muster oberhalb dieser Grenze werden ausgeschlossen. Jede Teilmenge eines ausgeschlossenen Musters unterhalb der Grenze bleibt dennoch bestehen. Auch hier bedeutet ein höherer Wert eine längere Rechenzeit.
- **Max Interprocedural Recursion** gibt die maximale Anzahl an rekursiven Schritten beim Auflösen von Methodenaufrufen an. Der Wert Null deaktiviert dieses Feature.
- **Training Set Directory** ist ein Basisverzeichnis, das .java-Klassen enthält, die für Muster- und Rule Mining verwendet werden.
- **Input Pattern Training** ist ein regulärer Ausdruck, der .java-Klassen aus dem Training Set Directory filtert. Hierbei wird der Ausdruck mit den absoluten Pfaden der .java-Dateien abgeglichen. Beim lokalen Training wird die Testdatei automatisch herausgefiltert. Ein möglicher Ausdruck ist zum Beispiel:

```
.*BProtErgSachlAbsetzungen
Ermittler.*|.*SubModelAbse
tzung.*|.*DAO.*|.*Constrai
nt.*|.*Helper.*
```

  Er gibt an, dass ausschließlich Dateien, in denen Dateinamen wie „DAO“, „Helper“ etc. als fachliche Datenzugriffsobjekt- und Hilfsklassen vorkommen, für das Training des Modells berücksichtigt werden sollen.
- **Caller Method Space** ist ein regulärer Ausdruck, der Methoden aus der Analyse herausfiltert. Hierbei wird der Ausdruck mit den vollaufgelösten Namen der Methoden abgeglichen.
- **Call Method Space** ist ein regulärer Ausdruck, der Methodenaufrufe aus der Analyse herausfiltert. Dabei wird der Ausdruck mit den vollaufgelösten Namen der Methodenaufrufe abgeglichen.

Abbildung 3 zeigt die eigens entwickelte Eclipse-Erweiterung im Einsatz bei der Bundesagentur für Arbeit zur Identifikation potenzieller fachlicher Fehler. Der Screenshot zeigt den im Beispiel aufgeführten Hotfix. Hier wurde in der Methode *ermittleDatenAusAuskunft* die Nullprüfung entfernt (deren Fehlen damals zum Hotfix führte) und das Machine-Learning-basierte System auf diese Methode angesetzt. Tatsächlich wurde die Regel gefunden, dass an der besagten Stelle eine Nullprüfung stattzufinden hat, und dies entsprechend vorgeschlagen. Konkret zeigte das System an, dass aus dem die Anweisung *getAbsetzungsrateBetragDetail* beinhaltenden Itemset *X* das Itemset *Y* mit der Anweisung *if(IAbsetzungsrateBetragDetailTO!=null)* folgt. Entsprechend wurde auf die fehlende Nullprüfung an der markierten Stelle in Zeile 299 hingewiesen.

Erste Erkenntnisse und Nutzen im konkreten Anwendungsfall der BA

Mit dem System wurde das ALLEGRO-Projekt analysiert und die für den Hotfix relevante Regel ermittelt. Darüber hinaus wurden weitere Regeln identifiziert.

Eine dieser Regeln besagt, dass eine per „TransactionalSection.enter“ eröffnete Transaktion auch per „TransactionalSection.markCommitable“ als

Capgemini

„committable“ markiert werden muss. Der Fehler trat hauptsächlich in Tests auf und ist zum Teil auf mangelnde Erfahrung zurückzuführen, da in Testfällen Transaktionen stets selbst zu öffnen sind. Im Produktivcode hingegen übernimmt dies ein Framework.

Ein anderes Beispiel ist die Regel, dass Entwickler beim Umgang mit bestimmten Geldbetrags-Objekten die Runden-Funktion aufrufen müssen. Darüber hinaus lieferte das System zudem Hinweise für die Benutzung von wiederverwendbaren, fachlich geeigneten Utility-/Helper-Methoden.

An dieser Stelle sei noch eine dritte gefundene Regel erwähnt. So muss in Testfällen eine Liste sortiert werden, die Summen bestimmter Beträge (hier Überzahlungen von Kranken- und Pflegeversicherungsbeiträgen) enthält. Dies muss genau dann geschehen, nachdem ihre Einträge, das heißt die Summen, in einem speziellen Verarbeitungsschritt zusammengefasst wurden. Die sortierte Reihenfolge ist für den Testfall wichtig, da sonst im schrittweisen Vergleich der erwartete vom tatsächlichen Wert abweichen würde.

Der wesentliche Vorteil ist, dass der Ansatz verstecktes Fehlerpotenzial findet, das eine klassische SCA nicht feststellt. Vielmehr ist es dort nötig, alle zu prüfenden Regeln im Vorfeld festzulegen. Erst nach Auftreten eines Fehlers können diese Regeln wiederum angepasst werden. Machine Learning bietet dagegen den Vorteil, dass Regeln via Mustererkennung „intelligent“ gefunden werden – und zwar bevor der Fehler im Produktivbetrieb auftritt.

Limitationen und weitere Potenziale

Die Praxis hat gezeigt, dass Regeln mit einer sehr hohen Konfidenz zwar relevant sind und ihre Verletzung zu Fehlern führt. Interessanter sind jedoch Regeln, die eine sehr niedrige Konfidenz haben, da weniger Mustervorkommnisse in einer großen Codemasse umso schwerer von Menschen festzustellen sind. Hier muss für die Modellerstellung eine sehr niedrige untere Grenze für den Unterstützungsgrad und für die Konfidenz eingestellt sein, was mehr Ressourcen in Anspruch nimmt.

Deshalb läuft das System parallelisiert in einem Spark-Cluster. Daraus ergibt sich voraussichtlich das Potenzial, bestimmte Muster auf eine Blacklist zu setzen, um das Grundrauschen zu minimieren. Auch die Integration des Systems als Plug-In in Eclipse öffnet neue Optionen. Denkbar ist beispielsweise, mittels Eclipse Code Recommenders eine „intelligente“ Autovervollständigung zu ermöglichen. Zudem könnten auch Regeln in eine Sperrliste aufgenommen werden, damit sie nicht weiterhin angezeigt werden.

Das hier vorgestellte Konzept umfasst einen zentralen Teil der Möglichkeiten, die maschinel-

Abb. 2: Vorgehensweise beim Trainieren und Anwenden des Modells

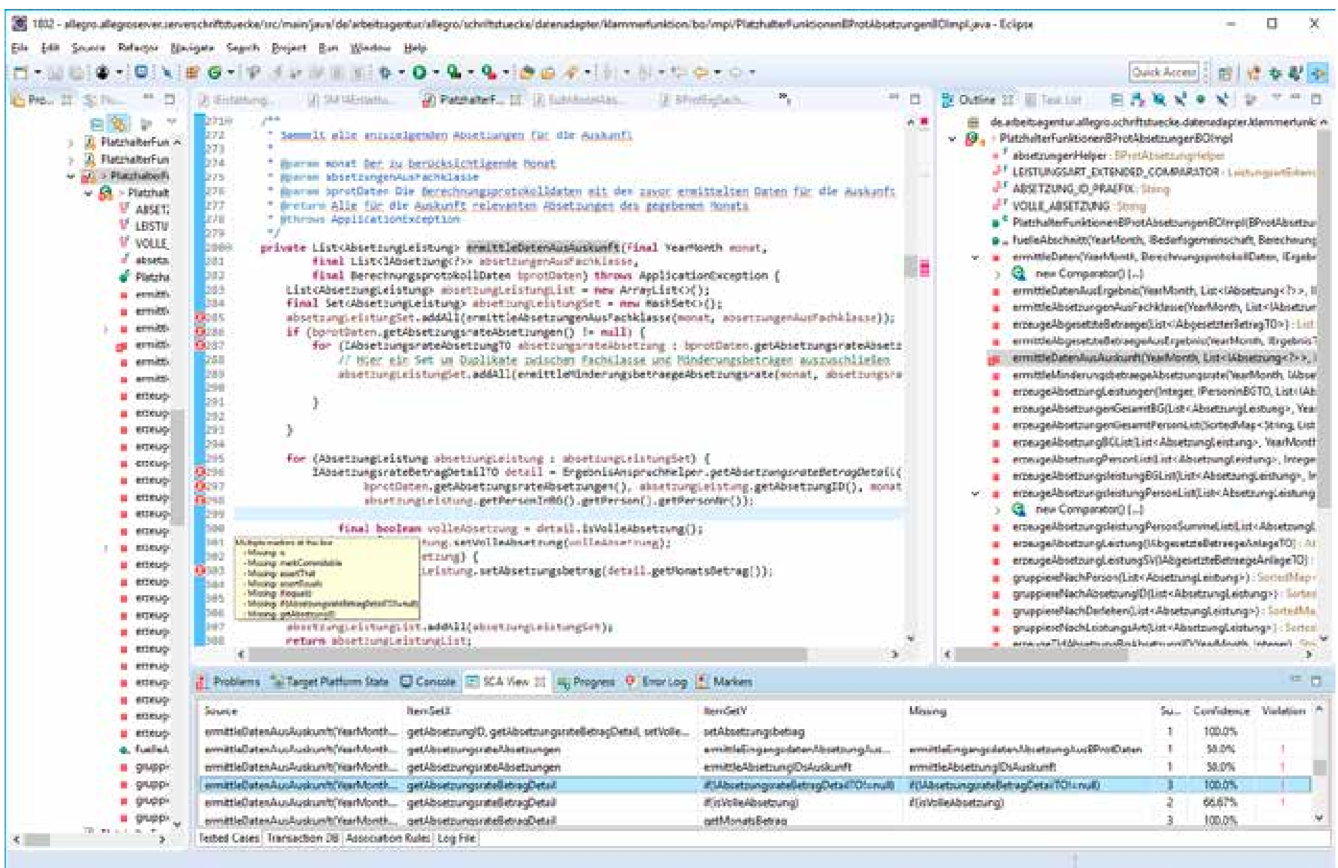

Abb. 3: Screenshot der Machine-Learning-basierten Eclipse-Erweiterung

les Lernen zur statischen Codeanalyse beitragen kann. Darüber hinaus zeichnen sich weitere Optionen ab, für die maschinelles Lernen gewinnbringend eingesetzt werden kann. Dazu zählt die *Code Clone Detection* [GaS16], [WeL17], womit sich doppelter Code auch dann erkennen lässt, wenn dieser abgewandelt oder anders geschrieben ist. *Vulnerability Scanning* zur Erkennung von Sicherheitsschwachstellen im Programmcode ist ein von Machine Learning adressierbares Problem [Mal06], [Git17], da sich Mustererkennungstechniken auch hierfür eignen. Es gibt hierzu sogar bereits Werkzeuge wie VDiscover (vdiscover.org) oder Sparrow (sparrow.fasoo.com), Letzteres sogar als quelloffene, akademische Version (ropas.snu.ac.kr/sparrow).

Ein weiteres Anwendungsgebiet sind *Naming & Documentation Suggestion* für Namens- sowie Dokumentationsvorschläge für Variablen- oder Methodennamen im Code [All16]. Hier bieten sich Text-Mining-Verfahren zur Kontexterschließung und -analyse aus dem Machine-Learning-Umfeld an.

Literatur

[All16] Allamanis, M.: Learning Natural Coding Conventions. Dissertation am Institute for Adaptive and Neural Computation, School of Informatics, University of Edinburgh 2016

[Alp14] Alpaydın, E.: Introduction to Machine Learning (3. Aufl.). Cambridge, MA: The MIT Press 2014

[GaS16] Gautam, P. / Saini, H.: Various Code Clone Detection Techniques and Tools: A Comprehensive Survey. In: Unal, A. et al. (Hrsg.): Smart Trends in Information Technology and Computer Communications. SmartCom 2016. Communications in Computer and Information Science, Vol. 628, Singapur: Springer 2016

[Git17] GitHub: Machine Learning for Cyber Security. Online verfügbar unter: https:// github.com/wtsxDev/Machine-Learning-for-Cyber-Security

[Liu07] Liu, B.: Web Data Mining – Exploring Hyperlinks, Contents, and Usage Data. Berlin 2007

[Mal06] Maloof, M. A. (Hrsg.): Machine Learning and Data Mining for Computer Security Methods and Applications. London 2006

[RiC13] Richert, W. / Coelho, L. P.: Building Machine Learning Systems with Python. Birmingham 2013

[WeL17] Wei, H.-H. / Li, M.: Supervised Deep Features for Software Functional Clone Detection by Exploiting Lexical and Syntactical Information in Source Code. Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence (IJCAI), 2017

Capgemini