Intelligente Werkzeuge im Software Engineering

Bastian Schneider
Fakultät Informatik
Hochschule Furtwangen
Furtwangen, Deutschland
bastian.schneider@hs-furtwangen.de

Marius Stuber
Fakultät Informatik
Hochschule Furtwangen
Furtwangen, Deutschland
marius.stuber@hs-furtwangen.de

***Abstract*—Weite Bereiche der Softwareentwicklung werden bereits heute durch diverse Werkzeuge unterstützt und teilweise automatisiert. Eine Verbesserung dieser Automatisierung soll dazu führen, dass die Entwicklung schneller und kosteneffizienter wird und eine höhere Qualität aufweist. Durch stetig steigende Forschungsarbeit im Bereich der künstlichen Intelligenz kann auch die werkzeugunterstützte Softwareentwicklung (engl. computer-aided software engineering, CASE) profitieren, indem bereits gängige Automatisierungsschritte durch den Einsatz neuer Methoden stark verbessert werden. Auch können sich durch diese Forschung neue Möglichkeiten ergeben, weitere Aufgaben während der Entwicklung zu automatisieren. Diese Verbesserungen und neuen Möglichkeiten sollen in dieser Ausarbeitung erklärt und gegebenenfalls genauer untersucht werden. In Tests, die im Rahmen dieser Ausarbeitung durchgeführt wurden, hat sich das Potential einiger Werkzeuge gezeigt, jedoch auch, dass sich die meisten Werkzeuge auf die KI-gestützte Verbesserung bereits verbreiteter Werkzeuge beschränken und damit nur wenig zu einer Automatisierung des kompletten Prozesses beitragen. Außerdem hat sich mit dem sog. Data Poisoning neues Gefahrenpotential herausgestellt.**

***Schlüsselworte*—Computer-Aided Software Engineering (CASE), Intelligente Werkzeuge, Künstliche Intelligenz, Maschinelles Lernen, Werkzeugunterstützung**

I. Einleitung

Die Entwicklung neuer Software ist ein langer, umfangreicher Prozess. Sie überspannt hierbei mehrere Themenbereiche mit unterschiedlichen, nötigen Kompetenzen und erfordert trotzdem einen engen Zusammenhalt zwischen den Ergebnissen dieser Schritte. Dies führt dazu, dass bei der Erstellung von Softwareprodukten nicht alle Ansprüche: eine schnelle Entwicklung und zeitnahe Berücksichtigung sich ändernder Anforderungen, Qualität und Kosteneffizienz gleichzeitig maximal gedeckt werden können.

In allen umfassten Themenbereichen – von der Analyse der Anforderungen bis zum fertigen Produkt – können den Entwicklern diverse Werkzeuge bei ihrer Arbeit unter die Arme greifen und so dabei helfen, den Erfüllungsgrad der oben genannten Ansprüche zu verbessern.

Mit der stetig wachsenden Forschung im Bereich der künstlichen Intelligenz (KI, engl. artificial intelligence, AI) bieten sich in vielen Arbeitsbereichen neue Möglichkeiten. Auch die Softwareentwicklung kann von diversen Durchbrüchen (wie durch künstliche Intelligenz verbesserten Vorschlagsalgorithmen) und aktuellen Forschungen im Bereich der künstlichen Intelligenz profitieren, indem bereits verbreitete Werkzeuge um neue Möglichkeiten erweitert oder abgelöst werden. Diese Ausarbeitung soll eine Übersicht über aktuelle Durchbrüche der künstlichen Intelligenz im Bereich der Werkzeugunterstützung des Software Engineerings bieten. Es wird sich mit der Frage beschäftigt, ob durch den Einsatz solcher Methoden eine Optimierung und ein höherer Automatisierungsgrad des Softwareentwicklungsprozesses erzielt werden kann. Außerdem werden Tests und Vergleiche einiger solcher Werkzeuge vorgestellt und entsprechend bewertet.

Aufgrund des Bedarfs, großer Mengen an Lerndaten im Bereich der künstliche Intelligenz, bedienen sich viele der untersuchten Werkzeuge bei öffentlichen Codequellen (Repositories). Dieser Ablauf vom Bezug der Lerndaten zum eingelernten Netzwerk ist schematisch in Abbildung 1 dargestellt. Es zeigt, wie auf Basis öffentlicher Daten das Training eines internen, neuronalen Netzes durchgeführt wird, welches schlussendlich für die dem Nutzer gegebenen Vorschläge verantwortlich ist. Diese Art des Datenbezugs birgt jedoch die Gefahr des sog. Data Poisonings, auf welches ebenfalls eingegangen wird.

II. Mögliche Einsatzgebiete für künstliche Intelligenz in der Softwareentwicklung

Software Engineering Werkzeuge finden in allen Bereichen des Entwicklungsprozesses Anwendung. Dieser Prozess lässt sich allgemein in fünf Phasen gliedern.

1) Anforderungsanalyse
   Hier werden die Wünsche des Kunden erfasst, die in der zu erstellenden Software berücksichtigt und erfüllt werden müssen. Das Ergebnis dieser Phase ist eine meist schriftliche Softwarespezifikation.

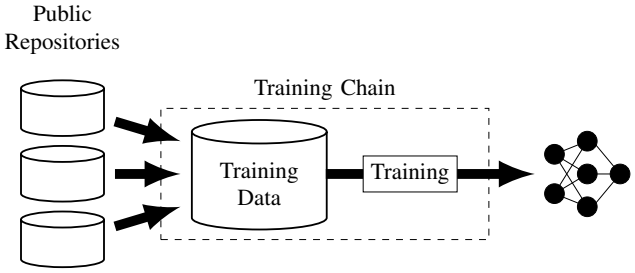

Abbildung 1. Bezug der Daten aus öffentlichen Quellen

2) Entwurf / Design
   Die genaue Struktur der Software wird, meist mithilfe von Modellierungssprachen wie UML, erstellt.
3) Implementierung
   Aus dem zuvor festgelegten Design wird eine konkrete Software erstellt.
4) Test
   Die erstellte Software wird diversen Tests unterzogen, um Merkmale wie Abdeckung der Anforderungen, Funktionalität und Sicherheit sicherzustellen.
5) Wartung
   Die Wartung bezieht sich auf die Instandhaltung der Software auch nach der Auslieferung an den Kunden, um zum Beispiel auf geänderte Anforderungen oder sich erst im Nachhinein herausstellende Fehler zu reagieren und nachzubessern.

In jeder dieser Phasen kann, durch den Einsatz von auf die Phase bezogenen Werkzeugen, die Arbeit erleichtert und teilweise automatisiert werden [1]. Hierbei eignen sich einige Phasen jedoch besser für den Einsatz solcher Werkzeuge als andere. Aufgrund wesentlicher Unterschiede zwischen den einzelnen Phasen der Softwareentwicklung sind die meisten Werkzeuge auch auf einzelne Phasen des Entwicklungsprozesses spezialisiert, wobei sich der Grad der Unterstützung und die Breite der angebotenen Werkzeuge in den Phasen unterscheidet [1].

Die Arbeit der verschiedenen Phasen findet auf sehr unterschiedlichen Abstraktionsniveaus statt. Während die Anforderungen üblicherweise in natürlicher Sprache formuliert werden und Objekttechnologien noch nicht beachten, wird in der Entwurfsphase bereits ein feiner Entwurf des Softwaresystems unter Berücksichtigung entsprechender Technologien erstellt. Dieser sogenannte Medienbruch zwischen zwei aufeinanderfolgenden Phasen macht einen automatisierten Übergang – oder auch schon die Unterstützung durch Werkzeuge – schwieriger, er ist jedoch nicht zwischen allen Phasen gleich stark ausgeprägt. Aus diesem Grund finden sich gerade im Bereich des Übergangs von der Anforderungsanalyse zum Design kaum Automatisierungswerkzeuge [1]. Selbiges gilt auch bei der Generierung nützlicher Tests aus einer Softwareimplementierung.

Die unterschiedlich starke Ausprägung des Medienbruchs zeigt sich besonders im Übergang vom Entwurf zu der Implementierung. So herrscht hier ein wesentlich geringerer Abstand, vor allem, wenn man schon beim Entwurf sich auf entsprechende Objekttechnologien bezieht. Dieser Übergang kann einfacher automatisiert werden, was auch mit bereits bestehenden Codeerzeugungswerkzeugen gängige Praxis ist. Dies beschränkt sich aktuell jedoch meist auf die Erzeugung eines Grundgerüsts, auf welchem dann die konkrete Implementierung aufbaut [1]. Durch den Einsatz künstlicher Intelligenz / maschinellen Lernens kann diese Automatisierung möglicherweise verbessert werden, um die Praxistauglichkeit weiter zu steigern. Wie ein Entwurf konkret umgesetzt werden soll, ist oft von unterschiedlichen Faktoren, beispielsweise von Performanzansprüchen, abhängig [2]. Dies führt dazu, dass sich aus dem gleichen Design unterschiedliche, konkrete Implementierungen ergeben können, die alle für ihren jeweiligen Anspruch korrekt sind. Kann nun ein Werkzeug die Zusammenhänge zwischen Design und Code besser „verstehen und nachvollziehen“, führt dies zu besseren Rückmeldungen zur Einhaltung oder Missachtung des zugrundeliegenden Entwurfs [2]. So soll diesem Beispiel entsprechend das in Unterabschnitt III-B vorgestellte Werkzeug erkennen können, ob eine Implementierung von dem geplanten Entwurf abweicht und entsprechende Rückmeldung liefern.

Innerhalb einer Phase können intelligente Werkzeuge einfacher zum Einsatz gebracht werden, da hier kein Medienbruch überwunden werden muss und die Aufgaben der einzelnen Werkzeuge wesentlich konkreter sein können. So müssen zum Beispiel Coding-Assistenten nur innerhalb eines Abstraktionsniveaus arbeiten, indem sie dem Entwickler Vorschläge beim Quelltext schreiben liefern.

III. Vorstellung intelligenter Werkzeuge

Folgend soll ein Überblick über bereits vermarktete oder noch in der Forschung befindliche Werkzeuge geschaffen werden, welche Methoden der künstlichen Intelligenz nutzen. Die Auswahl der Werkzeuge ist nach der Phase ihrer Anwendung innerhalb des Softwareentwicklungsprozesses sortiert (dargestellt in Abbildung 2). In diesem ersten Überblick wird sich hauptsächlich auf von den Entwicklern der Werkzeuge gebotenen Informationen bezogen.

In den anschließenden Kapiteln werden einige Werkzeuge, welche als Studierende der Hochschule Furtwangen frei verfügbar sind, einem eigenen Test unterzogen. Daraus soll eine bessere Einschätzung getroffen werden, wie ernstzunehmen die durch künstliche Intelligenz bedingten Verbesserungen sind.

A. Anforderungsanalyse

1) IBM Requirements Quality Assistant: IBM RQA ist ein Werkzeug, das Entwickler bei der Erstellung von klaren Anforderungen innerhalb der Anforderungsanalyse unterstützt. Dabei werden definierte Anforderungen durch das Werkzeug mithilfe der Watson AI analysiert. Diese nutzt Natural Language Processing (NLP), um die Anforderungen in ihrem Kontext zu verstehen und zu beurteilen [3]. Die Beurteilung erfolgt

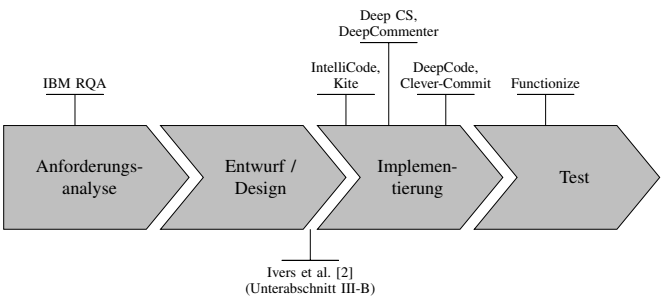

Abbildung 2. Einordnung betrachteter Werkzeuge in die Phasen des Software Engineering

auf Grundlage der INCOSE-Richtlinien, welche definieren wie eine unmissverständliche Anforderung aufgebaut werden soll [3]. Wörter die subjektiv, vage oder mehrdeutig interpretiert werden können, werden markiert, sodass der Entwickler die Befunde ausbessern kann. Das Werkzeug hilft somit dabei, Anforderungen klarer und weniger missverständlich zu gestalten. Außerdem kann es nach jeder Anwendung durch Feedback des Bedieners auf ein unerwünschtes Verhalten bei der Beurteilung hingewiesen werden. IBM RQA analysiert diese Hinweise dann wiederum mittels maschinellen Lernens und berücksichtigt sie bei der Analyse weiterer Anforderungen [3].

B. Design / Implementierung: Codegenerierung

Wie bereits zuvor beschrieben ist der automatisierte Übergang von Entwürfen zu Code meist auf bloße Grundgerüste beschränkt. Dies schränkt auch bei der Weiterentwicklung des Codes die Möglichkeit stark ein, die Einhaltung des Designs automatisiert zu gewährleisten und gegebene Abweichungen zu dokumentieren.

Ein sich noch in der Forschung befindendes Projekt von Ivers et al. [2] beschäftigt sich mit der Erkennung von Designstrukturen in Code und damit mit ebendiesem Problem. Hier werden mit Algorithmen des maschinellen Lernens strukturelle Zusammenhänge zwischen dem Entwurf und Strukturen im Code hergestellt. Ein erster Prototyp wurde dabei bereits mit 100 Open-Source Java Projekten getestet, mit dem Ziel, Model- und Controller-Strukturen des Model-View-Controller-Design Patterns zu identizifieren. Hierbei konnte für das Model eine Trefferquote (Recall) von 86% und eine Genauigkeit (Precision) von 91%, sowie für den Controller ein Recall von 87% und eine Precision von 96% erzielt werden [2]. Diese Forschungsarbeit wird jedoch noch weitergeführt, bevor hier ein anwendbares, intelligentes Werkzeug entsteht.

C. Implementierung: Coding-Assistenten

1) IntelliCode: IntelliCode ist ein IntelliSense Feature, das eine kontextbewusste Codevervollständigung mithilfe des Pythia Code Completion Systems liefert. Dabei versucht IntelliCode vorherzusagen, welche Methoden der Programmierer benutzen möchte und bietet sie dem Programmierer als Vorschlag an. Die dabei erhobenen Vorschläge haben durch Pythia eine deutlich höhere Trefferquote als Code-Verfollständigungssysteme, die häufigkeits- oder aufrufbasiert arbeiten [4].

Die Informationen aus denen IntelliCode lernt, um Vorschläge liefern zu können, bezieht es aus ausgewählten, öffentlichen GitHub Projekten. Zusätzlich dazu können auch eigene Git-Projekte als Trainingsdaten zur Verfügung gestellt werden, um die zu liefernden Vorschläge den eigenen Projekten anzupassen.

Der Vorschlagsfunktion von IntelliCode liegt der bei Microsoft von Svyatkovskiy et al. [4] entwickelte Pythia-Ansatz zugrunde. Hierbei wird der Code der Lerndaten anhand dessen abstrakten Syntaxbaums (engl. abstract syntax tree, AST) zerlegt und mittels der NLP-Methode Word2vec in Vektoren umgewandelt, aus welchen das neuronale Netz der Vorschlagsfunktion schließlich lernt [4]. Bei diesem Algorithmus wird jedes Wort einem Vektor zugewiesen, wobei Wörter mit ähnlichem Kontext nah beieinander liegen. So können Zusammenhänge zwischen verschiedenen Wörtern gelernt werden. Bei der Vervollständigung durch IntelliCode werden diese Zusammenhänge wieder erkannt und Vorschläge daraus abgeleitet.

IntelliCode unterstützt in Visual Studio die Sprachen C#, XAML, C++, JavaScript, TypeScript und Visual Basic. In Visual Studio Code wird Java, JavaScript, TypeScript, Python und SQL angeboten.

2) Kite: Ähnlich wie IntelliCode ist Kite ein Werkzeug, das mittels Deep Learning eine kontextbewusste Codevervollständigung liefert. Dabei ist Kite in der Lage, den Kontext einer ganzen Code-Zeile zu interpretieren und Vorschläge zu liefern, wie diese vervollständigt werden kann. Die Informationen aus denen Kite lernt, um diese Vorschläge liefern zu können, bezieht es ebenfalls aus ausgewählten, öffentlichen GitHub Projekten. Anders als IntelliCode kann Kite jedoch nicht auf eigenen Projekte trainiert werden.

Kite wird dabei für die Editoren IntelliJ, Jupiter Lab, Sublime, Webstorm, Atom, PhpStorm, RubyMine, GoLand, VS Code, PyCharm, Spyder, Vim, CLion, Rider, AppCode und Android Studio angeboten. Durch die breite Auswahl an Editoren werden die Sprachen Java, C, C++, C#, TypeScript, Kotlin, Less, Ruby, HTML/CSS, Go, Scala, Javascript, PHP und Bash unterstützt.

In Abschnitt IV werden die beiden Coding-Assistenten IntelliCode und Kite genauer verglichen.

D. Implementierung: Statische Codeanalyse

1) DeepCode: DeepCode ist ein Werkzeug für die statische Codeanalyse, das dabei hilft, Programmierfehler und Sicherheitslücken aufzuzeigen. Es wurde auf Basis von etwa 200 000 Git-Projekten und ihrer Änderungshistorie mittels maschinellen Lernens trainiert [5]. Dadurch ist es in der Lage, nicht nur Fehlerpotentiale aufzuspüren, sondern auch Lösungsvorschläge zu liefern, die andere Entwickler in anderen Projekten so schon einmal angewendet haben. Das Werkzeug ist sehr einfach in ein Projekt zu integrieren und kann bereits nach wenigen Minuten über die DeepCode-Webseite auf eigene Projekte in GitHub, Bitbucket oder GitLab angewendet werden. Alternativ gibt es DeepCode auch als Plugin für Visual Studio Code, SublimeText, IntelliJ und Atom, sowie für die Kommandozeile, um es automatisiert auszuführen. Die Analyse durch DeepCode verläuft zudem sehr schnell, sodass es sich für Echtzeit-Feedback eignet [5]. Für Teams mit bis zu 30 Entwicklern ist DeepCode außerdem gebührenfrei.

2) Clever-Commit: Clever-Commit / CLEVER (Combining Levels of Bug Prevention and Resolution techniques) ist ein von Ubisoft La Forge – einer Forschungsabteilung des französischen Videospielunternehmens Ubisoft – entwickeltes Analysewerkzeug, welches den Commit von Code überwacht. Die Entwicklung von Clever-Commit wird zusätzlich von Mozilla unterstützt.

Clever-Commit soll eine Möglichkeit bieten, sog. Risky Commits, welche Programmierfehler in das Gesamtsystem einführen, aufzudecken, bevor diese das Zentrale Code-Repository erreichen. Es baut hierbei auf Commit Guru von Rosen et al. [6] auf und erweitert dessen Analyseverfahren. Im ersten Schritt, wie auch bei Commit Guru, wird auf Basis von Metriken das Risiko, welches von einem Commit ausgeht, eingeschätzt. In der zweiten Phase wird eine Klonerkennung angewendet, die, wenn der Commit als ein solcher Risky Commit eingestuft wurde, diesen mithilfe künstlicher Intelligenz mit Mustern vergangener, nachweislich fehlerhafter Commits vergleicht [7], [8]. Zusätzlich bietet Clever-Commit dem Entwickler mögliche Fixes an, die das Risiko eliminieren können [7].

Laut eigenen Angaben liefert Clever-Commit beim Finden von Risky Commits einen Recall von 65% und eine Precision von 79%. Außerdem wurden 66,7% der vorgeschlagen Fixes angenommen [7].

Clever-Commit ist ein privates Werkzeug von Ubisoft, welches nicht öffentlich verfügbar ist.

E. Implementierung: Code-/Kommentargenerierung

1) Deep CS: Deep CS steht für Deep Code Search und ist ein experimentelles Werkzeug für Codevorschläge (sog. Code Search Tools) für Java-Code, das als Proof of Concept für die Implementierung des neuronalen Netzwerks CODEnn (Code-Description Embedding Neural Network) entwickelt wurde [9]. CODEnn weist Codebausteinen und deren Dokumentation den gleichen Vektor zu, wodurch eine semantische Beziehungen zwischen dem Code und seiner Dokumentation erzeugt wird. Suchanfragen werden über Deep CS in einen weiteren Vektor umgewandelt, der mit den Vektoren des Vektorraums aller gelernten Codebausteine verglichen wird. Die Vektoren, die dem Vektor der Suchanfrage am ähnlichsten sind, werden dann als Vorschläge zurückgegeben. Auf diese Weise können auch Funktionen vorgeschlagen werden, deren Funktionsnamen nur semantisch mit der Suchanfrage übereinstimmen. Irrelevante oder störende Wörter können ebenfalls besser gehandhabt werden als bei Code Search Tools, die mit Information Retrival arbeiten. So weist Deep CS, gelernt auf 9 950 GitHub Repositories mit mehr als 20 Sternen, für eine Suchanfrage eine Trefferquote von 76% unter den obersten 5 Treffern auf, während das mit Information Retrival arbeitende CodeHow bei den Versuchen von Gu et al. [9] nur 58% erreicht.

2) DeepCommenter: DeepCommenter ist ein Werkzeug zur automatisierten Kommentarerstellung für Java-Code. Es nutzt Hybrid-DeepCom, eine Methode, welche sich auf Neural Machine Translation (NMT) und einem sequenzbasierten Sprachmodel stützt. NMT wird häufig für die Übersetzung einer Sprache in eine andere genutzt und eignet sich eigentlich nicht für ein umfangreiches Vokabular wie das von Quelltext [10]. Dieses Problem wird durch die Kombination des NMT mit dem sequenzbasierten Sprachmodell gelöst [11]. DeepCommenter lernt aus den Daten von GitHub Repositories und baut die generierten Kommentare auf Basis des Codes und der Kommentare dieser Repositories auf. Es kann unter IntelliJ als Plugin genutzt werden [12]. Bei einem Test zur Beurteilung der Qualität generierter Kommentare schnitt das DeepCode zugrundeliegende Hybrid-DeepCom gegenüber der CODE-NN Methode – nicht zu verwechseln mit CODEnn – das eine vergleichbare Funktion bietet, deutlich besser ab [11]. Für den Test wurde DeepCode auf Basis von 9 714 GitHub Projekten mit jeweils über 10 Sternen eingelernt.

F. Tests

1) Functionize: Mit Functionize lassen sich funktionale Tests für Webanwendungen erzeugen und warten. Für die Testerzeugung stehen dabei zwei Möglichkeiten zur verfügung. Die Tests können in natürlicher Sprache geschrieben und dann durch Natural Language Processing in einen Testfall überführt werden oder über den Functionize Smart Recorder bei ausgeführter Webanwendung aufgenommen werden [13]. Dieser funktioniert ähnlich wie die Aufnahme eines Makros, wird aber von einer KI-gestützten Bilderkennung unterstützt, die Elemente der Oberfläche erkennt und dem Programmcode zuordnen kann [13]. Der Tester kann dadurch nach dem Start der Aufnahme die zu testende Oberfläche bedienen wie ein Benutzer, während das Werkzeug sich im Hintergrund die ausgeführten Aktionen merkt. Die erzeugten Testfälle können dann automatisiert auf verschiedenen Browsern für Desktop und Mobilgeräte ausgeführt werden.

Die Testwartung unterstützt Functionize durch intelligente Verbesserungsvorschläge und der adaptiven Event Analyse [13]. Hierdurch wird bei der Ausführung von Testfällen versucht, die Positionen von Objekten der Webseite zu ermitteln, wodurch zum Beispiel auch nach einer Neupositionierung einzelner Buttons diese auf der Webseite gefunden werden können. Nach dem Test kann ein Entwickler dann den Grund für die neue Positionierung des Objekts identifizeren oder Functionize den Test auf die neue Position abändern lassen.

Functionize bietet somit eine mit künstlicher Intelligenz gestützte Testfallerstellung und einen automatisierten Testprozess. Der Tester muss die Tests jedoch weiterhin selbst erstellen, Functionize kann diese nur an sich ändernde Rahmenbedingungen anpassen. Somit trägt Functionize nicht zu einem Übergang von der Implementierung zum Testen bei.

IV. Test ausgewählter Werkzeuge: Coding-Assistenten

Zum Vergleich intelligenter Coding-Assistenten wie Kite und IntelliCode wurde ein einfaches Machine-Learning Programm wiederholt unter folgenden Bedingungen geschrieben:

- ohne Unterstützung durch die Entwicklungsumgebung,
- mit einfacher, alphabetischer Vorschlagsfunktion,
- mithilfe von IntelliCode
- und mithilfe von Kite.

Das Programm entstammt einer Praktikumsaufgabe aus dem Kurs „Maschinelles Lernen“ von Prof. Dr. Maja Temerinac-Ott der Hochschule Furtwangen. Der volle Programmtext findet sich im Anhang in Listing 4.

Die Ergebnisse dieses Versuchs sind in Abbildung 3 grafisch dargestellt. Beim Schreiben ohne Unterstützung der Entwicklungsumgebung wurden für den gesamten Quelltext 1 460 Tastenanschläge benötigt. Durch Verwendung der alphabetischen Vorschlagsfunktion in Visual Studio Code wurde die Anzahl benötigter Anschläge um 17,5% auf 1 205 reduziert. Gegenüber den alphabetischen Vorschlägen liefert IntelliCode eine Verbesserung von 5,9% und Kite 19,9%. Ignoriert man Strings wie Pfad- und Dateinamen sowie Kommentare, bei denen jeweils keine Vorschläge gemacht werden, steigt die Ersparnis von IntelliCode gegenüber den alphabetischen Vorschlägen auf 7,1% und die von Kite auf 23,8%.

A. *Kite*

Kite liefert im direkten Vergleich mit IntelliCode eine wesentlich höhere Ersparnis an reiner Schreibarbeit. Dies liegt daran, dass Kite, anders als IntelliCode, nicht nur häufig angewendete Methoden vorschlägt, sondern komplette Codekonstrukte.

Eine Ersparnis von rund 20% ist jedoch nicht gleichbedeutend mit einer Arbeitsersparnis von 20%. Die Unterstützung nimmt bei der Entwicklung hauptsächlich Routinearbeiten ab und ersetzt nicht die Denkarbeit des Entwicklers. Eine solche routinemäßige Vervollständigung ist beispielhaft in Abbildung 4 dargestellt, in dem ein sehr üblicher Import des Interface „pyplot“ der Bibliothek „matplotlib“ mit entsprechendem Alias vorgeschlagen wird. Die Kite-Vorschlagsfunktion kann jedoch durchaus auch unterstützend über reine Routinearbeit hinaus wirken, indem häufig verwendete Codekonstrukte weiter oben vorgeschlagen werden und so Denkanstöße, insbesondere beim Entwickeln mit zuvor selten verwendeten Bibliotheken, in eine richtige Richtung gegeben werden können.

Die Kite Vervollständigung passt sich über die Vorschläge der am häufigst genutzten Funktionen hinaus im Laufe der Entwicklung an diverse Faktoren an. Befindet man sich beispielweise mit dem Cursor am Anfang einer Datei, wo üblicherweise die Importe geregelt werden, bietet Kite beim Eintippen eines „f“ als ersten Vorschlag „from“ an. Befindet man sich jedoch nicht mehr in diesem Bereich und hat bereits Programmcode geschrieben, ändern sich die Vorschläge entsprechend. Dieses Verhalten ist in Abbildung 5 zu sehen. Ebenso werden zum Beispiel bereits verwendete Codeausschnitte in die Liste der Vorschläge aufgenommen, um zum Beispiel später auftretende, ähnliche Konstrukte vervollständigen zu können.

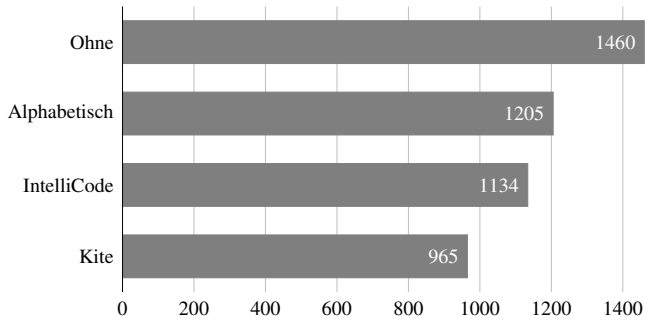

Abbildung 3. Tastenanschläge mit und ohne Coding-Assistent

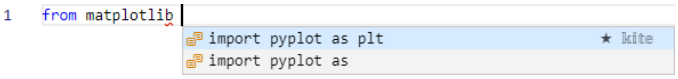

Abbildung 4. Kite-Autovervollständigung beim Import

B. *IntelliCode*

Anders als Kite bietet IntelliCode nur Vorschläge beim Aufruf von Methoden an. Für alle weiteren Vorschläge dient hier die bereits einzeln getestete alphabetische Vorschlagfunktion von Visual Studio Code.

Wie auch Kite ist die Vorschlagsfunktion von IntelliCode kontextsensitiv. Das heißt die vorgeschlagenen Methoden unterscheiden sich, abhängig von dem umliegenden Programmcode. So wurden beim Testen, wie in Abbildung 6 dargestellt, bei der Zuweisung von Variablen andere Methoden vorgeschlagen als bei einem späteren Aufruf.

Die vorgeschlagenen Methoden sind, wie im Vergleich zwischen IntelliCode und rein alphabetischer Vorschlagsfunktion in Abbildung 3 sichtbar, besser und sparen so bei der Entwicklung Zeit ein. Jedoch ist der Funktionsbereich wesentlich eingeschränkter und scheint auch hier noch nicht über das Wissen von Kite zu verfügen, da manche Methoden, wie `np.sqrt(...)` (siehe Listing 4, Zeile 42) von IntelliCode garnicht erkannt wurden. Es kann nach diesem Test nicht ausgeschlossen werden, dass IntelliCode eine bessere Abdeckung bei der Verwendung mit anderen Bibliotheken bietet, welche in diesem Test nicht überprüft wurden. Zumindest jedoch innerhalb der weit verbreiteten Bibliotheken „numpy“ und „matplotlib“ bietet Kite eine bessere Assistenz und ist IntelliCode vor allem bei Vorschlägen über Methodenaufrufe hinaus voraus, weshalb dieses Ergebnis durchaus Relevanz aufweist.

V. Test ausgewählter Werkzeuge: DeepCode

Um das Potential von DeepCode zu ermitteln, wurde dieses auf das bereits zuvor verwendete Machine-Learning Programm angewendet. Dabei wurden zwei Vorschläge für die Code-Optimierung durch PyLint geliefert, DeepCode selbst konnte

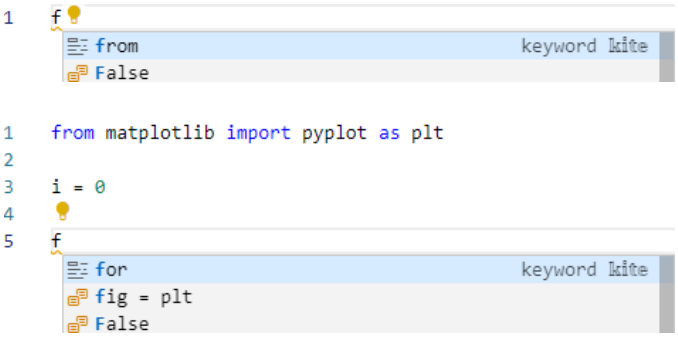

Abbildung 5. Verschiedene Vorschläge von Kite bei gleicher Eingabe

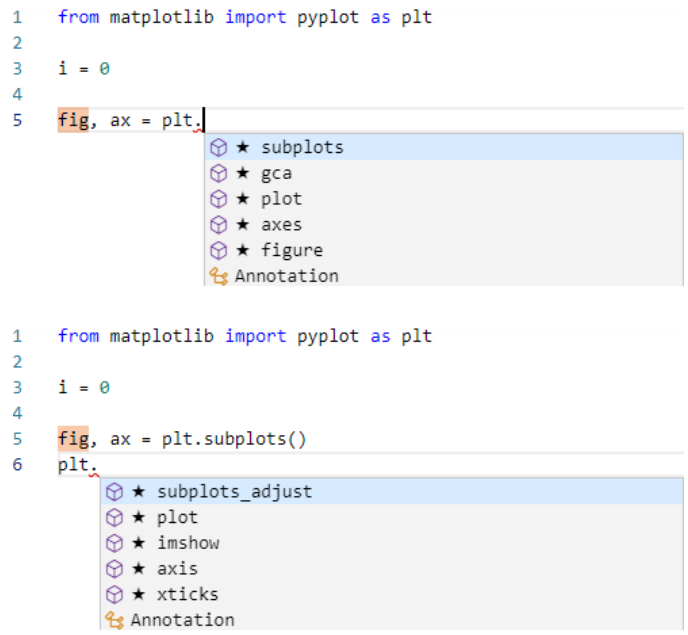

Abbildung 6. Verschiedene Vorschläge von IntelliCode bei gleicher Eingabe

jedoch keine weiteren Verbesserungspotentiale ausmachen. Daher wurde das Programm stellenweise manipuliert, um eine Reaktion von DeepCode zu provozieren. Mit den folgenden Fehlern wurde versucht, eine Reaktion von DeepCode zu erzielen:

- Dead Code, durch ein return vor weiteren Anweisungen,
- Dead Code durch eine if-Bedingung, die immer gleich verläuft (siehe Listing 1),
- eine Dekrementierung eines Wertes mit 0,
- ein direkt zu Beginn der Funktion überschriebener und damit ungenutzter Übergabewert (siehe Listing 2),
- eine Rekursion, die durch den überschriebenen Übergabewert zu einer Endlosrekursion wird (siehe ebenfalls Listing 2),
- eine versuchte Manipulation eines immutable Strings, die beim Ausführen zu einem Fehler führen wird
- und ein vergessener `len()` Aufruf, der zur Nutzung eines Strings als Argument für `range()` und damit zu einem Fehler führt.

DeepCode konnte unter diesen Fehlern den Dead Code durch return sowie die Manipulation des Strings ausmachen, einige offensichtlich wirkende Fehler wurden jedoch nicht erkannt. Für die gefundenen Fehler liefert DeepCode eine ausführliche Erklärung und Vorschläge, die zur Behebung hilfreich sind.

Bei den von uns getesteten Fehlerfällen ist das Ergebnis der DeepCode Analyse eher schlecht ausgefallen, zumal der entdeckte Dead Code Fehlerfall auch durch die Codeabdeckungsprüfung anderer Werkzeuge oder von verschiedenen IDEs ohne die Nutzung von KI erkannt werden kann. Weil im Rahmen dieses Tests nur wenige Fehlerfälle überprüft wurden, kann auf Basis dieses Ergebnisses kein abschließendes Urteil über die Qualität der DeepCode-Analyse gefällt werden Ebenso kann unter dieser Bedingung die Geschwindigkeit nicht ausreichend bewertet werden. Hierfür müsste DeepCode im Rahmen eines größeren Projekts getestet werden.

Aufgrund der Lernfähigkeit von DeepCode kann sich die Fehlerfindungsrate in Zukunft verbessern. Aktuell scheint DeepCode, unter Voraussetzung seiner vom Hersteller beworbenen Schnelligkeit, als Erweiterung in einem Projekt dienen zu können, nicht aber als Ersatz bestehender statischer Codeanalysewerkzeuge.

VI. Test ausgewählter Werkzeuge: DeepCommenter

Im Rahmen dieser Ausarbeitung wurde DeepCommenter stichprobenhaft getestet. Dazu wurde es auf mehrere Java-Methoden angewendet und die erzeugten Kommentare evaluiert. Die folgenden Punkte sind dabei aufgefallen:

- die Ausführungszeit des Werkzeugs auf Code war unabhängig von der Länge der Methode in einem akzeptablen Bereich von unter zwei Sekunden,
- eine Methode muss aktuell noch explizit mit einem Modifizierer wie private, public oder protected versehen sein, damit ein Kommentar generiert werden kann,
- Methoden dürfen nur eine maximale Länge von 150 Zeilen haben,
- bereits kleine Änderungen der Methode können den erzeugten Kommentar stark verändern (siehe Methoden isPrime in Listing 3),
- besonders für längere oder komplexere Methoden, sowie Methoden die zu großen Teilen aus eigenen Methodenaufrufen bestehen, wurden unbrauchbare Kommentare erzeugt
- und, wie in der helloWorld-Methode in Listing 3 zu sehen, ist die generelle Qualität der Kommentare noch nicht ausreichend.

Unter den generierten Kommentaren für die Test-Methoden befand sich noch keiner, der sich als brauchbar erwiesen hat. Die Nützlichkeit des aktuell verfügbaren IntelliJ-Plugins

Listing 1
DEAD CODE DURCH IF-BEDINGUNG

```
allowed_chars = []

if char not in allowed_chars:
  return False
```

Listing 2
ENDLOSREKURSION DURCH ÜBERSCHRIEBENEN PARAMETER

```
def rec(i):
  i = 1
  while i > 0:
    rec(i-1)
```

Listing 3
GENERIERTE KOMMENTARE MIT DEEPCOMMENTER

```
1  /**
2   * return true if the given long is a prime
       number min max number .
3   */
4  public static boolean isPrime(final long n) {
5    ...
6  }
7
8  /**
9   * return true if n is a prime number equals is
       true false otherwise .
10  */
11 public static boolean isPrime(final int n) {
12   ...
13 }
14
15 /**
16  * call this method to handle hello world book
17  */
18 public void helloWorld() {
19   System.out.println("hello World");
20 }
```

wird daher als gering eingestuft. Die erzeugten Kommentare weisen allerdings alle eine Beziehungen zu ihren Methoden auf. Es sind lediglich einzelne Wörter, die den Sinn eines Kommentars zerstören, wie anhand der Beispielkommentare aus Listing 3 zu sehen ist. Das Werkzeug hat Potential, in späteren Versionen eine hilfreiche Unterstützung bei der Erzeugung von besser dokumentiertem Code zu werden.

VII. DATA POISONING

Mit den neuen Möglichkeiten, die sich mit der Einführung künstlicher Intelligenz in den Softwareentwicklungsprozess ergeben, zeichnen sich jedoch auch neue Schwachstellen in diesem ab. Eine Gefahr der aktuellen Herangehensweise diverser Werkzeuge an das Training von Netzen stellt das sogenannte Data Poisoning (deutsch „vergiften von Daten“) dar.

Methoden des maschinellen Lernens bedürfen sehr großer Lerndatenmengen, um sich ordentlich einzulernen. Aus diesem Grund bedienen sich viele der vorgestellten Werkzeuge bei öffentlichen Repositories, um entsprechend große Datenmengen zu erhalten. Diese Praxis könnten sich Angreifer in Zukunft zunutze machen, indem sie angreifbare Schwachstellen in eigene Repositories einfügen, um so Lerndaten intelligenter Systeme mit diesen Schwachstellen zu versehen [14]. Die fehlerhaften Daten ziehen sich dann durch den Lernprozess bis zum finalen, neuronalen Netz durch, auf welchem dann gegebene Vorschläge oder Entscheidungen basieren (siehe Abbildung 7).

Entwickler von intelligenten Werkzeugen orientieren sich meist, um eine hohe Qualität an Trainingsdaten zu bekommen, bei den bestbewerteten Repositories. Um die Chance zu erhöhen, in die Trainingsdaten solcher Software aufgenommen zu werden, können Betreiber „bösartiger“ Repositories versuchen,

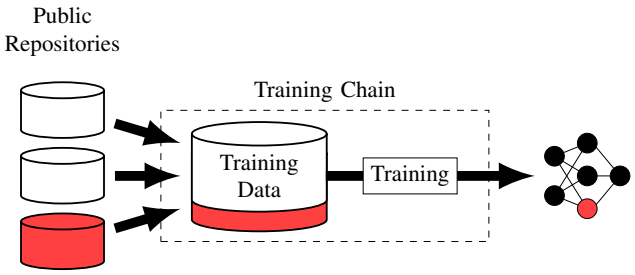

Abbildung 7. Data Poisoning durch öffentliche Repositories [14]

ihres künstlich besser dastehen zu lassen, durch wiederholte, gute Bewertungen mit neuen Accounts [14].

Es ist von besonderer Wichtigkeit für die Entwicklung intelligenter Software, solche Schwachstellen in den Trainingsdaten – egal ob absichtlich oder unabsichtlich eingebracht – zu identifizieren und zu entfernen, um dem Nutzer der Software keine fehlerhaften oder sicherheitskritischen Vorschläge zu machen.

Es zeichnet sich also auch eine Gefahr der gängigen Praxis beim Entwickeln intelligenter Software, das zu eventuellen Sicherheitsrisiken bei der Nutzung dieser Werkzeuge führen kann.

VIII. FAZIT

Die aktuell bereits marktfähigen, intelligenten Werkzeuge scheinen sich vorrangig innerhalb einzelner Phasen des Software Engineerings zu bewegen, in welchen auch bereits zuvor Werkzeugunterstützung geleistet werden konnte. Dieser Zusammenhang erscheint logisch, da es einfacher ist, bereits bestehende, funktionierende Konzepte mit Algorithmen des maschinellen Lernens zu unterstützten und so zu erweitern, also neue Durchbrüche zu machen.

Diese Unterstützung herkömmlicher Methoden durch solche des maschinellen Lernens erweist sich im Test als vielversprechend. Im Fall des Coding-Assistenten Kite konnte eine deutliche Senkung der reinen Codierarbeit festgestellt werden. Die Integration von künstlicher Intelligenz in bereit verbreiteten Werkzeugen liefert vielversprechende Ergebnisse und einen durchaus höheren Automatisierungsgrad.

Jedoch befinden sich auch diverse, neue Ansätze in der Forschung, welche bisher kaum abgedeckte Bereiche der Entwicklung im Fokus haben. Diese beziehen sich jedoch aktuell auch auf die Arbeit innerhalb einer Phase, wie bei Deep CS, oder, im Fall der Forschungsarbeit von Ivers et al. [2], auf einen Übergang mit einem geringeren Medienbruch. Zu den schwierigeren Aufgaben, wie dem Übergang von Anforderungen zu Design, wurden noch keine Ansätze gefunden.

Grenzen dieser zukünftigen Forschungen zu identifizieren – egal ob prinzipielle oder praktische Grenzen – erweist sich aktuell als schwierig, da im Bereich der künstlichen Intelligenz stetig weiter geforscht wird und so sich Grenzen immer weiter verschieben können. Es gibt hierbei jedoch auch harte Grenzen, wie die Prüfung, ob ein Programm terminiert, welche, unabhängig von der Qualität des Werkzeugs, nicht

überwunden werden können. Es kann anhand des aktuellen Forschungsstandes davon ausgegangen werden, dass Schritte, welche die Kreativität eines Entwicklers erfordern, sei es die konkrete Planung eines Entwurfs oder der Denkprozess bei der Implementierung, nicht ersetzt werden können. Selbes gilt für Übergänge zwischen den Phasen, vor allem je größer der zu überwindende Medienbruch ist. Somit ist nicht von einer Vollautomatisierung des Entwicklungsprozesses in greifbarer Zukunft auszugehen, hier kann also eine praktische Grenzen gezogen werden. Es ist anzunehmen, dass intelligente Werkzeuge weiterhin in einer unterstützenden Funktion bleiben. Mit einem Durchbruch im Bereich der künstlichen Intelligenz muss diese Grenze jedoch neu evaluiert werden.

Außerdem gilt für alle sich bei öffentlichen Repositories bedienenden Werkzeuge die Problematik des Data Poisoning genau im Blick zu behalten, weshalb man sich nicht vollends auf die vorgeschlagenen Bibliotheken oder Methoden verlassen sollte.

A. Ausblick

Da die meisten Ansätze, welche sich auf neue, bisher noch nicht automatisierte Phasen der Softwareentwicklung beziehen, noch sehr jung sind, gilt es, diese Forschung eng im Auge zu behalten. Bei Erfolg betreffender Projekte kann sich ein wesentlich höherer Automatisierungsgrad des gesamten Prozesses ergeben.

Ein besonders interessanter Ausblick bietet sich in der Zusammenarbeit zwischen den Werkzeugen DeepCommenter und Deep CS. Einerseits gilt es hier zukünftig zu prüfen, ob diese beiden Werkzeuge sich gegenseitig negativ beeinflussen, da DeepCommenter automatisch Kommentare generiert und Deep CS auf Basis von Kommentaren Codevorschläge sucht. Andererseits kann diese Interaktion jedoch auch positiv ausfallen, indem das Codeverständnis und eventuell objektivere Kommentare von DeepCommenter zu noch besseren Suchen seitens Deep CS führen.

LITERATUR

[1] I. Sommerville, *Software Engineering*, 9. Aufl. USA: Addison-Wesley Publishing Company, 2012.

[2] J. Ivers, I. Ozkaya, und R. L. Nord, „Can AI Close the Design-Code Abstraction Gap?" in *2019 34th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW)*, 2019.

[3] IBM. IBM Engineering Requirements Quality Assistant. [Online]. Abrufbar: https://www.ibm.com/uk-en/products/requirements-quality-assistant [Abgerufen am 11.01.2021]

[4] A. Svyatkovskiy, Y. Zhao, S. Fu, und N. Sundaresan, „Pythia: AI-Assisted Code Completion System," in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, Serie KDD '19. New York, NY: Association for Computing Machinery, 2019.

[5] DeepCode. (2020) What makes DeepCode different to other tools? [Online]. Abrufbar: https://deepcode.freshdesk.com/support/solutions/articles/60000651789 [Abgerufen am 04.01.2021]

[6] C. Rosen, B. Grawi, und E. Shihab, „Commit Guru: Analytics and Risk Prediction of Software Commits," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. New York, NY: Association for Computing Machinery, 2015.

[7] M. Nayrolles und A. Hamou-Lhadj, „CLEVER: Combining Code Metrics with Clone Detection for Just-in-Time Fault Prevention and Resolution in Large Industrial Projects," in *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, 2018.

[8] Mozilla. (2019) Clever-Commit von Ubisoft macht die Entwicklung von Firefox noch schneller. [Online]. Abrufbar: https://blog.mozilla.org/press-de/2019/02/13/clever-commit-von-ubisoft-macht-die-entwicklung-von-firefox-noch-schneller/ [Abgerufen am 07.01.2021]

[9] X. Gu, H. Zhang, und S. Kim, „Deep Code Search," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, 2018.

[10] V. J. Hellendoorn und D. Premkumar, „Are deep neural networks the best choice for modeling source code?" in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. New York, NY: Association for Computing Machinery, 2017.

[11] X. Hu, G. Li, X. Xia, D. Lo, und Z. Jin, „Deep code comment generation with hybrid lexical and syntactical information," *Empirical Software Engineering*, Vol. 25, 2020.

[12] B. Li, M. Yan, X. Xia, X. Hu, G. Li, und D. Lo, *DeepCommenter: A Deep Code Comment Generation Tool with Hybrid Lexical and Syntactical Information*. New York, NY: Association for Computing Machinery, 2020.

[13] Functionize. (2018) Intelligent Functional Testing.

[14] R. Schuster, C. Song, E. Tromer, und V. Shmatikov, „You Autocomplete Me: Poisoning Vulnerabilities in Neural Code Completion," *ArXiv*, Vol. abs/2007.02220, 2020.

Listing 4
POINTS OF INTEREST

```
import numpy as np
from matplotlib import patches
from matplotlib import pyplot as plt
from skimage import data, io
from skimage.feature import blob_doh, blob_log, canny
from skimage.color import rgb2gray
from skimage.util import img_as_float, invert

img_path = './sunflower.jpg'
det_path = './img_determinant.jpg'
lap_path = './img_laplacian.jpg'

canny_sigma = 2
canny_low_threshold = 0.15
blob_overlap = 0

img = io.imread(img_path, True)
canny_img = canny(img, sigma=canny_sigma,
    low_threshold=canny_low_threshold)

fig, ax = plt.subplots(1, 2, figsize=(15, 10))
ax[0].imshow(img, cmap=plt.cm.gray)
ax[1].imshow(canny_img)

plt.show()

### Determinant of Hessian

img_blobs = blob_doh(img, overlap=blob_overlap)

fig, ax = plt.subplots()

# Set aspect ratio
ax = plt.Axes(fig, [0., 0., 1., 1.])
fig.add_axes(ax)

# Add image and blobs
ax.imshow(img, cmap=plt.cm.gray, aspect='auto')
ax.set_axis_off()
for blob in img_blobs:
  ax.add_patch(patches.Circle((blob[1], blob[0]),
        blob[2]*np.sqrt(2), color='r', fill=False))

plt.savefig(det_path)
plt.show()

### Laplacian of Gaussian

img_blobs = blob_log(invert(img))

fig, ax = plt.subplots()

# Set aspect ratio
ax = plt.Axes(fig, [0., 0., 1., 1.])
fig.add_axes(ax)

# Add image and blobs
ax.imshow(img, cmap=plt.cm.gray, aspect='auto')
ax.set_axis_off()
for blob in img_blobs:
  ax.add_patch(patches.Circle((blob[1], blob[0]),
        blob[2]*np.sqrt(2), color='r', fill=False))

plt.savefig(lap_path)
plt.show()
```