Philipp de Sombre (3502136)

Automatische Erkennung von Entscheidungswissen in der Beschreibung und den Kommentaren von JIRA-Issues und Commit-Nachrichten

Fortgeschrittenenpraktikum (8LP)

Sommersemester 2019
Betreuung: Prof. Dr. Barbara Paech, Anja Kleebaum
Institut für Informatik
Ruprecht-Karls-Universität Heidelberg

19. Dezember 2019

Abstract

Kontext & Motivation Während des Softwareenwicklungsprozesses müssen EntwicklerInnen eine Vielzahl an Entscheidungen treffen. Diese Entscheidungen können sich auf verschiedenste Teile des Prozesses beziehen. Das hierbei erzeugte Wissen wird *Entscheidungswissen* genannt. Neben der Entscheidung ist es für EntwicklerInnen auch wichtig das Entscheidungsproblem, das hinter der Entscheidung steht, sowie alternative Lösungsmöglichkeiten und Argumente bzw. Begründungen zu kennen.

Studien haben gezeigt, dass EntwicklerInnen Entscheidungswissen häufig nicht explizit dokumentieren. Es ist stattdessen nur implizit in Kommunikationsartefakten vorhanden. Als Grund für dieses Phänomen gilt das *capture problem*. Das Problem beschreibt, dass der kurzfristige Nutzen, den einE EntwicklerIn aus der expliziten Dokumentation zieht gering, der Aufwand jedoch relativ dazu gesehen hoch ist.

Beiträge Um implizites Entscheidungswissen explizit zu machen wird maschinelles Lernen genutzt. Durch eine Literaturrecherche werden verschiedene Ansätze der automatischen Klassifizierung von Entscheidungswissen verglichen. Die Erkenntnisse werden dann in Experimenten geprüft und erweitert. Die Erkenntnisse aus den Experimenten fließen dann in die programmatische Umsetzung in dem Entscheidungsdokumentationstool ConDec für JIRA ein. Zusätzlich werden Git-Commit-Nachricht als weiterer mögliche Quelle für Entscheidungswissen in das Tool integriert. Die Umsetzung wird nach dem Technology-Acceptance-Model durch ConDec-EntwicklerInnen evaluiert.

Schlussfolgerung Die Entscheidungsdokumentation kann durch die automatische Klassifizierung von Entscheidungselementen verbessert werden. Implizites Wissen wird explizit und kann somit genutzt werden. Die Integration von Git-Commit-Nachrichten als weitere Wissensquelle erlaubt EntwicklerInnen die Dokumentation von Entscheidungswissen ohne einen Kontextwechsel und senkt somit den Aufwand der Dokumentation.

Abkürzungsverzeichnis

(A-)REACT (Automatic) RationalE Annotations in ChaT message

CI Continuous Integration

ConDec Continuous Decision

CURES Continuous Usage- and Rationale-based Evolution Decision Support

DRL Decision Representation Language

GloVe Global Vectors for Word Representation

IBIS Issue-Based Information System

IDE Integrated development environment

JSON JavaScript Object Notation

ML maschinelles Lernen

NFR Non-functional Requirement

QOC Questions, Options, and Criteria

REST REpresentational State Transfer

SDK Software Development Kit

SMILE Statistical Machine Intelligence & Learning Engine

SMOTE Synthetic minority over-sampling technique

SVM Support Vector Machine

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

In diesem Kapitel wird zuerst im Abschnitt *Motivation* die Relevanz von Entscheidungswissen und die Probleme bei dessen Dokumentation dargelegt. Anschließend werden im folgenden Abschnitt 1.2 die Ziele dieser Arbeit erläutert. Schließlich wird im Abschnitt Aufbau die Struktur der Ausarbeitung und der Inhalt der einzelnen Kapitel vorgestellt.

1.1 Motivation

Während des Softwareenwicklungsprozesses müssen EntwicklerInnen eine Vielzahl an Entscheidungen treffen. Diese Entscheidungen können sich auf verschiedenste Teile des Prozesses, zum Beispiel: Anforderungen oder Architektur, beziehen. Das hierbei erzeugte Wissen wird *Entscheidungswissen* genannt. [9]

Entscheidungswissen hilft dabei das Architektur-Design oder die Implementierung, in Form von Quellcode, nachzuvollziehen. Neben der Entscheidung ist es für EntwicklerInnen auch wichtig das Entscheidungsproblem, das hinter der Entscheidung steht, sowie alternative Lösungsmöglichkeiten und Argumente bzw. Begründungen zu kennen. Die Entscheidungsprobleme in der Softwareenwicklung sind häufig sogenannt *wicked problems*, die sich nicht mathematisch, sondern nur argumentativ lösen lassen. Das Entscheidungswissen ist vor allem bei lange andauernden Softwareenwicklung-Projekten, in denen Entscheidung und Umsetzung weit auseinander liegen können, von hohem Wert. [9]

Studien haben gezeigt, dass EntwicklerInnen Entscheidungswissen häufig nicht explizit dokumentieren. Es ist stattdessen implizit in Kommunikationsartefakten, wie beispielsweise Chat-Nachrichten, JIRA-Issue-Kommentar oder Git-Commit-Nachricht, vorhanden [2]. Als Grund für dieses Phänomen gilt das *capture problem*. Das Problem beschreibt, dass der kurzfristige Nutzen, den einE EntwicklerIn aus der expliziten Dokumentation zieht gering, der Aufwand jedoch relativ dazu gesehen hoch ist. Da die Dokumentation in externen Tools stattfindet, unterbricht dies den Entwicklungsprozess durch einen Kontextwechsel. Kurzfristig kann die explizite Dokumentation aber

auch Vorteile mit sich bringen. Dadurch werden die Kriterien und Argumente für und gegen Alternativen klarer und implizite Annahmen der EntwicklerInnen offengelegt. Langfristig überwiegt der Nutzen durch die Dokumentation den Aufwand. Da implizit dokumentiertes Entscheidungswissen schwer auffindbar ist, hat es nur ein geringes Nutzen für die Stakeholder des jeweiligen Projektes. Außerdem ist es schwer retrospektiv Entscheidungswissen, dass informell und an verschiedenen Stellen dokumentiert ist gut zu nutzen. Das implizite Wissen ist eventuell veraltet, inkonsistent, unvollständig oder schwer auffindbar ist.

1.2 Ziele

Ziel ist es die in der Motivation dargelegten Probleme zu lösen. Dafür müssen drei Forschungsfragen (FF) beantwortet werden. Zuerst, muss untersucht werden, wo, das bedeutet in welchen Artefakten, implizites Entscheidungswissen enthalten ist (FF1). Zusätzlich muss festgestellt werden, welche Typen von Entscheidungswissen unterschieden werden können (FF2). Darauf aufbauend soll schließlich eine Methode gefunden werden, mithilfe derer es möglich sein soll implizites Entscheidungswissen automatisch aus den Artefakten zu extrahieren und den Typen entsprechend zu klassifizieren (FF3). Der Fokus dieser Arbeit liegt auf der dritten Forschungsfrage. In dieser Arbeit wird eine automatisierte Lösung dieses Problems, mithilfe von Software, entwickelt.

1.3 Aufbau

Die Fragen FF1 und FF2 wurden bereits ausführlich untersucht und werden in Kapitel 2 beantwortet. Zusätzlich werden in diesem Kapitel, in Abschnitt 2.3, Grundlagen zum maschinellen Lernen und, in Abschnitt 2.2, das Projekt, in dessen Kontext diese Arbeit stattfindet, erklärt. Zur Beantwortung von FF3 werden verwandte Arbeiten analysiert und verglichen (siehe Kapitel 3).

Anschließend werden in Kapitel 4 systematisch funktionale und nicht-funktionale Anforderungen erhoben. Darauf aufbauend wird eine Software-Lösung entworfen. Die Erkenntnisse aus der Literaturrecherche werden als Grundlage genutzt, um eigene Experimente durchzuführen (Kapitel 5). Die Anforderungen werden darauf folgend umgesetzt (Kapitel 6). Um die Qualität dieser Lösung sicherzustellen wird im Kapitel Qualitätssicherung die Planung und die Ergebnisse von qualitätssichernden Maßnahmen, Komponenten- und Systemtests, erläutert. Anschließend wird die umgesetzte Software-

Lösung mithilfe einer Umfrage unter EntwicklerInnen validiert (Kapitel 8).

Abschließend werden im Kapitel Schlussfolgerung die Erkenntnisse zusammengefasst, diskutiert und ein Ausblick auf mögliche zukünftige Arbeiten gegeben.

2 Grundlagen

Dieses Kapitel erklärt die zum Verständnis dieser Arbeit notwendigen Grundlagen. Hierbei wird zuerst auf die *Dokumentation von Entscheidungswissen* eingegangen. Anschließend wird in Abschnitt 2.2 das Projekt, in dessen Kontext diese Arbeit durchgeführt wird, erläutert. Abschnitt 2.3 legt Techniken zur automatischen Klassifikation von Daten dar. Im Unterabschnitt Evaluation von Modellen wird erläutert wie die Qualität von Modellen gemessen und verglichen werden kann. In den weiteren Unterabschnitten *Online und Batch Learning* und *Natürlichsprachliche Texte* wird auf Details eingegangen, die für diese Arbeit besonders relevant sind.

2.1 Dokumentation von Entscheidungswissen

Im Laufe von Softwareprojekten fallen eine Vielzahl von Entscheidungen an [31]. Diese haben Auswirkungen auf die Eigenschaften der Software. Bei den dahinter liegenden Entscheidungsproblemen handelt es sich häufig um sogenannte *wicked problems*. Diese können nicht mathematisch gelöst werden. Stattdessen muss durch das Abwägen von Möglichkeiten und Diskussion die beste mögliche Lösung gefunden werden. Diese Motivation hinter Entscheidungen wird als *Rationale* bezeichnet. [5]

Dokumentationsmodelle Um Entscheidungen und vor allem die Hintergründe für diese festzuhalten wurde eine Vielzahl an Modellen entwickelt. Die Modelle sollen EntwicklerInnen unterstützen das Entscheidungswissen dokumentieren, zu strukturieren, sowie wieder darauf zurückgreifen zu können, wenn nötig.

Das erste dieser Modelle ist das 1970 von Kunz und Rittel vorgeschlagene *Issue-Based Information System* - Modell, kurz IBIS. Dieses unterscheidet drei Arten von Wissen: *Issue*, *Position* und *Argument*. Ein *Issue* beschreibt ein zu lösendes Problem und sollte in Form einer Frage dargestellt werden. Eine *Position* bezeichnet eine vorgeschlagene Lösung. In anderen Dokumentationsmodellen wird eine *Position* auch als *Alternative*

bezeichnet. Ein *Argument*-Element kann eine *Position* unterstützen oder ihr widersprechen. [35, 5, 9] Abbildung 2.1 stellt die Beziehungen der Elemente zueinander dar.

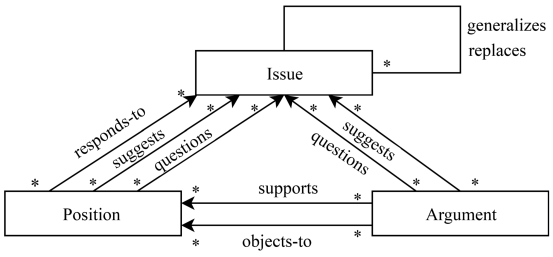

Abbildung 2.1: Beziehungen der Elemente des IBIS Dokumentationsmodells nach Kunz und Rittel [1]

Weitere Modelle zur Dokumentation sind *Decision Representation Language* (DRL), *Questions, Options, and Criteria* (QOC) und das *Non Functional Requirement Framework* (NFR Framwork) [5].

DRL ist eine Sprache, die entwickelt wurde um die qualitativen Aspekte des Entscheidungsfindungsprozesses repräsentieren zu können [19].

QOC besteht aus den drei Bestandteilen: *Question*, *Option* und *Criteria*. Die *Question* stellt das Problem dar. Die *Option* ist eine mögliche Antwort auf die *Question*. Die verschiedenen *Options* werden mithilfe der *Criteria* bewertet und verglichen. [21]

Beim NFR Framework wird von einem *Softgoal* ausgegangen. Dieses stellt eine allgemeine, und somit unkonkrete, nicht-funktionale Anforderung dar. Diese wird dann immer weiter verfeinert, bis konkrete Anforderungen gefunden wurden. [25]

Ein verfeinertes Modell wurde von Hesse, Tom-Michael und Kuehlwein, Arthur und Paech, Barbara und Roehm, Tobias und Bruegge, Bernd erarbeitet [12, 13]. Bei diesem steht die Entscheidung (*Decision*) im Mittelpunkt (siehe Abbildung 2.2). Eine Entscheidung wird von einer *Person* getroffen. Die Decision enthält Entscheidungskomponenten (*DecisionComponent*) und dient als Behälter für ebendiese. Eine Entscheidungskomponente wird von einer Person identifiziert. Sie kann außerdem auch andere Entscheidungskomponenten enthalten.

Die von Hesse, Tom-Michael und Kuehlwein, Arthur und Paech, Barbara und Roehm, Tobias und Bruegge, Bernd identifizierten Elemente des Entscheidungsfindungsprozesses sind als Unterklassen der DecisionComponent modelliert und aufgeteilt in:

Das *Problem* beschreibt das eigentliche Entscheidungsproblem und beschreibt einen negativen Zustand. Dieses kann entweder eine Problemfrage (*Issue*) oder ein Ziel (*Goal*),

das es zu erreichen gilt, sein. Das Problem gilt es zu beheben. Die Lösung (Englisch: *Lösung*) bezieht sich auf ein bestimmtes Problem und wird als mögliche *Alternative* ausgedrückt. Ein *Claim* beschreibt zusätzliche Forderungen an eine Lösung. Mit dem *Context* werden Rahmenbedingungen des Entscheidungsproblems in Form von: *Assumption*, *Constraint* und *Implication* beschrieben. Die *Rationale* sind die Gründe für oder gegen eine mögliche Lösung. Hierbei werden bei Hesse, Tom-Michael und Paech, Barbara *Argument*, ein Pro- oder Kontra-Argument, und *Assessment*, eine Einschätzung, unterschieden. [12, 13]

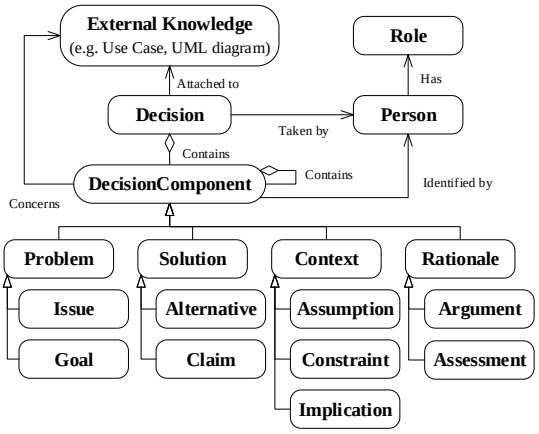

Abbildung 2.2: Beziehungen der Elemente des Dokumentationsmodells von Hesse et al. [12]

Orte für Entscheidungen In der Praxis wird Entscheidungswissen häufig unzureichend oder nur implizit dokumentiert [31]. Anstatt, dass das Wissen explizit dokumentiert wird, findet es sich implizit in Entwicklungsartefakten wieder. Artefakte und Tools, die Entscheidungswissen enthalten, sind: E-Mails, PowerPoint Präsentationen, Projekt Management -, Issue Management -, Source Code Version Management -, Meeting Recording Systeme oder Source Code [4, 12].

2.2 Das ConDec-Projekt

Das Continuous Decision Knowledge Management Projekt, kurz ConDec-Projekt, ist Teil des CURES-Forschungsprojektes [1]. Issue Management – und Source Code Version Management Systeme finden in der Softwareentwicklung eine häufige Anwendung. Sie unterstützen jedoch nicht das strukturierte Erfassen von Entscheidungswissen. Das Ziel des Projektes ist es diesen Missstand durch Tool-Unterstützung zu beseitigen. Dabei

[1] http://www.dfg-spp1593.de/cures/ (Zuletzt abgerufen am: 03.06.2019)

werden von Kleebaum, Anja und Johanssen, Jan Ole und Paech, Barbara und Alkadhi, Rana und Bruegge, Bernd die Anforderungen an solche Tools gesammelt und diese darauf aufbauend entworfen, umgesetzt und getestet. In Kleebaum, Anja und Johanssen, Jan Ole und Paech, Barbara und Bruegge, Bernd [17] wurden sieben Anforderungen an ein solches Tool zur Verwaltung von Entscheidungswissen erhoben.

R1 *Developers are supported in explicitly capturing decision knowledge consistent with artifacts in the tools they work with.*

R2 *Developers are presented with distributed decision knowledge when performing a finish practice.*

R3 *Developers are presented with summarized artifact changes when performing a finish practice.*

R4 *Developers are presented with relevant knowledge and artifacts when performing a start practice.*

R5 *Developers are supported in accessing knowledge by filtering and searching from within the tools they use and from a dashboard, not bound to start and finish practices, also from artifacts such as code and features.* [17]

Die implementierten ConDec-Tools setzen diese Anforderungen um und ermöglichen somit eine inkrementelle und kooperative Dokumentation von Entscheidungswissen. [16, 17] Das genutzte Datenmodell basiert auf dem IBIS-Modell Modell für Entscheidungswissen, kann jedoch durch zusätzliche Elemente erweitert werden.

2.3 Automatische Klassifikation

Beim maschinellen Lernen (ML) wird versucht mithilfe von Erfahrungen Muster zu finden. Diese Erfahrungen liegen bei Maschinen in Form von Daten vor. Mithilfe von Algorithmen werden dann Muster aus den Daten extrahiert. [22] Anstatt ein Problem durch explizite Anweisungen zu lösen, kann das implizite in den Daten vorhandene Wissen genutzt werden. Beispielsweise könnte eine Empfehlungsfunktion, wie sie bei Online-Händlern häufig vorkommt, einerseits durch eine Reihe von EntwicklerInnen festgelegten Bedingungen programmiert werden, die mögliche neue Empfehlungen berechnen. Dies ist der explizite Ansatz. Andererseits kann das Kaufverhalten anderer Kunden genutzt werden, um neue Empfehlungen zu berechnen. Dies ist der implizite Ansatz bei dem Muster, die in Daten vorhanden ist, genutzt werden, um das Problem zu lösen.

Beim maschinellen Lernen mit Daten, bei denen das gewünschte Resultat schon bekannt ist, spricht man vom überwachten Lernen. Hierbei wird zwischen zwei Arten unterschieden. Bei Regressions-Algorithmen handelt es sich beim zu erlernenden Ziel-

wert um einen numerischen Wert. Bei der Klassifikation hingegen sind die Zielwerte endlich, diese werden in der Literatur als *Labels* oder auch *Klassen* bezeichnet. [33]

Im Gegensatz zum überwachten Lernen sind beim unüberwachten Lernen keine Klassen bei den Trainingsdaten vorhanden. Hierbei ist es nicht das Ziel der Algorithmen einzelne Zielwerte zu berechnen, sondern die Daten beispielsweise zu gruppieren oder eine Dimensionsreduktion durchzuführen. [22]

Abbildung 2.3 stellt das Vorgehen beim überwachten maschinellen Lernen grafisch dar. Zuerst werden die Daten in Trainings- und Testdaten geteilt. Mithilfe der Trainingsdaten werden Modelle trainiert. Um die Fähigkeiten der Modelle zu testen, werden Testdaten genommen, die nicht bereits zum Trainieren verwendet wurden. Somit soll sichergestellt werden, dass das Modell gut verallgemeinert und nicht nur die Daten wiedergibt, dieses ungewollte Verhalten wird in der Literatur als *overfitting* bezeichnet. Das am besten abschneidende Modell wird dann ausgewählt und eingesetzt.

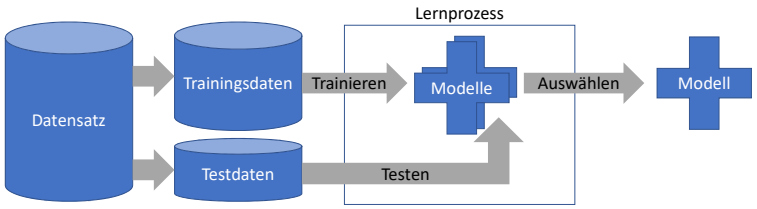

Abbildung 2.3: Schematische Darstellung des überwachten maschinellen Lernens (eigene Darstellung nach [7])

2.3.1 Evaluation von Modellen

Um das Abschneiden der Modelle bewerten zu können müssen Kriterien festgelegt, gemessen und die Ergebnisse der verschiedenen Modelle verglichen werden. Nehmen wir einen Klassifizierer an der nur zwei Klassen unterscheidet. Wenn bei diesem binäre Klassifizierer die vorausgesagten mit den tatsächlichen Klassen in Beziehung gesetzt werden, ergibt sich eine 2×2 – Tabelle (siehe Abbildung 2.1).

Tabelle 2.1: Ergebnistabelle eines binären Klassifizierers

		Wahr	
		positiv	negativ
Vorhergesagt	positiv	True positive (TP)	False positive (FP)
	negativ	False negative (FN)	True negative (TN)

Mithilfe dieser Tabelle lassen sich die mehrere Kriterien berechnen. Im folgenden werden *Genauigkeit (Precision)*, *Trefferquote (Recall)*, *Korrektklassifikationsrate (Accuracy)*

und der *F1-Wert* vorgestellt.

Die Genauigkeit ist das Verhältnis der korrekt als positiv klassifizierten Daten zu allen als positiv klassifizierten Daten: $Przision := \frac{TP}{TP+FP}$.

Der Trefferquote wiederum ist das Verhältnis der korrekt als positiv klassifizierten Daten zu allen tatsächlich positiven Daten: $Trefferquote := \frac{TP}{TP+FN}$.

Die Korrektklassifikationsrate beschreibt das Verhältnis aller korrekt klassifizierten Elemente zu allen Elementen: $Korrektklassifikationsrate := \frac{TP+TN}{GESAMTPOPULATION}$.

Beide Kriterien für sich alleine sind anfällig für extreme Situationen. Wenn ein Klassifizierer zum Beispiel alle Daten als positiv klassifizieren würde, hätte der Trefferquote einen Wert von 1 und würde somit als perfekt gelten. Um diesen Effekt zu verhindern wird der sogenannte F_1-Wert eingeführt. Dieser ist das harmonische Mittel von Genauigkeit und Trefferquote: $F_1 := \frac{2 \times Genauigkeit \times Trefferquote}{Genauigkeit+Trefferquote}$ [33]

Bei Klassifikationsproblemen mit mehr als zwei Klassen kann die sogenannte *one-versus-all* Methode genutzt werden. Bei dieser wird jedes Label wie ein binäreres Klassifizierungsproblem evaluiert. [22] Somit können die gleichen Kriterien wie bei einem binären Klassifizierer genutzt werden.

2.3.2 Ungleichmäßig verteilte Daten

In Datensätzen sind die Klassen häufig ungleichmäßig verteilt. Dabei sind die normalen Klassen öfters vorhanden als die abnormalen Klassen. Die abnormalen Klassen sind jedoch häufig die interessanteren, zum Beispiel das Bild einer Krebszelle oder Kreditkartenbetrug. Ein Modell, das das Ziel hat die Genauigkeit zu erhöhen, wird die seltenen Klassen deutlich schlechter erkennen als die häufig vorkommenden. Die Kosten oder negativen Folgen, die beim Nicht-Erkennen der abnormalen Klassen anfallen, sind jedoch oft höher als beim Nicht-Erkennen der normalen Klassen. Deshalb wurden verschiedene Methoden entwickelt, um diesem Phänomen zu begegnen. [26]

Beim *Undersampling* werden die Anzahl der Beispiele aus der normalen Mehrheitsklasse reduziert. *Oversampling* verhält sich gegenteilig zum Undersampling. Hierbei werden Beispiele aus der abnormalen Minderheitsklasse mehrfach beim Training verwendet, um einen ausgeglichenen Datensatz zu schaffen. [10] *SMOTE* steht für „Synthetic Minority Over-sampling Technique“ und beschreibt eine spezielle Oversampling-Methode. Anstatt existierende Beispiele mehrfach zu nutzen, werden neue synthetische Beispiele

erzeugt. Bei dieser Methode werden die vorhandenen Beispiele zu neuen kombiniert. [26]

Die vorgestellten Methoden können miteinander kombiniert werden.

2.3.3 On-line und Batch Learning

Wenn beim maschinellen Lernen so vorgegangen wird wie in Abbildung 2.3 dargestellt, müssen die Trainings- und Testdaten die gleiche Verteilung aufweisen, wie die später zu klassifizierenden Daten. Dieses Vorgehen wird in der Literatur als *Batch Learning* bezeichnet. Ändern sich die Daten im Laufe der Zeit oder weil das trainierte Modell in einer anderen Umgebung zum Einsatz kommt, funktionieren die Voraussagen nicht mehr gut. Im Gegensatz dazu gibt es den Ansatz des *On-line Learning*s, auch inkrementelles Lernen genannt. Hierbei wird das Modell immer weiter angepasst und somit verbessert. Neue Daten werden genutzt, um das Modell anzupassen. Es lernt also dazu. Dadurch kann auf Veränderungen in der Verteilung der Daten reagiert werden. [22]

Diese Methode des On-line Learning ist besonders sinnvoll, wenn bei der Nutzung des Modells immer neue Daten generiert werden. Stellen wir uns zum Beispiel folgendes Szenario vor. Die Reihenfolge der Ergebnisse einer Suche wird durch ein Modell festgelegt. Sobald ein Nutzer sucht und ein Ergebnis auswählt, kann das Modell verbessert werden, indem es mit dem ausgewählten und nicht ausgewählten Ergebnissen weiter trainiert wird.

2.3.4 Natürlichsprachliche Texte

Texte in natürlicher Sprache sind eine mögliche Art der Daten, die für maschinelles Lernen genutzt werden können. Die natürliche Sprache ist nicht immer eindeutig und kann sich im Laufe der Zeit ändern. Wenn solche Texte effektiv klassifiziert werden sollen, müssen sie zuerst vorbehandelt werden. [15, 33]

Bei solche Vorbehandlungs-Methoden können drei Arten unterschieden werden. Als Beispiel zur Veranschaulichung der Effekte der Methoden auf einen Text, dient der Kommentar *' But please keep looking for simplifications! '*. Dieser Kommentar stammt aus einem Datensatz von Kommentaren aus dem LUCENE Projekt [2]. Die drei Arten werden bezeichnet als:

[2]LUCENE JIRA: https://issues.apache.org/jira/projects/LUCENE/issues (Zuletzt abgerufen am: 03.06.2019)

- *Tokenization*: Das Zerlegen eines Textes in seine einzelnen Elemente. Beispiel: * , But, please, keep, looking, for, simplifications, !, *
- *Stop Word Removal*: Das Entfernen von bedeutungslosen Füllwörtern. Beispiel: *But, please, looking, simplifications, !*
- *Stemming*: Das Zusammenführen Wörtervarianten zu einem gemeinsamen Stamm. Beispiel: *but, please, look, simple, !*

[11, 32].

Je nach konkretem Algorithmus können sich die Ergebnisse der jeweiligen Verarbeitungsschritte unterscheiden.

3 Verwandte Arbeiten

In diesem Kapitel werden verwandte Arbeiten betrachtet. Um als verwandte Arbeit betrachtet zu werden, muss diese sich mit der Klassifizierung von Textdaten in Softwareentwicklungsartefakten beschäftigen. Arbeiten die sich in einer anderen Form mit Artefakten oder Entscheidungen in der Softwareentwicklung beschäftigen werden nicht untersucht.

Zuerst werden im Abschnitt 3.1 die verschiedenen Arbeiten erläutert, wobei jeweils besonders auf die Relevanz für dieses Praktikum eingegangen wird. In der anschließenden Synthese werden die Arbeiten dann untereinander verglichen. Ausgegangen bei der Auswahl der Literatur wird hierbei von der Praktikumsbeschreibung. Von dort aus wird Forward- und Backward-Snowballing durchgeführt, um weitere relevante Quellen zu finden.

3.1 Beschreibung

Automatic Extraction of Design Decisions from Issue Management Systems: A Machine Learning Based Approach In [4] untersuchen die Autoren implizit dokumentiertes Entscheidungswissen. Dieses soll mithilfe von überwachtem maschinellem Lernen aus JIRA-Issues extrahiert werden. Die Daten werden zuerst per Hand annotiert, um das Modell zu trainieren und testen zu können. Als Dokumentationsmodell nutzen die Autoren eine von Kruchten vorgeschlagene Ontologie [18]. Die Textdaten aus den JIRA-Issues werden dann vorverarbeitet (siehe Abschnitt 2.3.4). Anschließend wird mit diesen Daten ein Multi-Klassen Klassifizierer trainiert. In ihrer Arbeit werden fünf verschiedene Machine-Learning-Algorithmen getestet und verglichen.

DecXtract: Dokumentation und Nutzung von Entscheidungswissen in JIRA-Issue-Kommentaren In dieser Arbeit ([6]) untersucht der Autor ebenfalls verschiedene Möglichkeiten der automatischen Extraktion von Wissen aus JIRA-Issue-Kommentar. Der Fokus ist jedoch ein anderer. Anstatt verschiedene Algorithmen miteinander zu verglei-

chen, führt der Autor eine ausführliche Literaturrecherche durch und legt den Fokus auf die Integration der Textklassifizierung in Elemente des Entscheidungswissens in den Softwareenwicklungsprozess. Somit, wird eine Architektur erarbeitet, implementiert und getestet. Die Arbeit fand im Rahmen des ConDec-Projektes statt.

Exploring techniques for rationale extraction from existing documents In dieser Arbeit ([30]) liegt der Fokus der Autoren nicht darauf eine einzelne Methode zu verfeinern oder eine Software zu entwickeln, sondern darauf eine breite Masse an Algorithmen miteinander zu vergleichen. Um diesen Vergleich durchzuführen werden 100 zufällig ausgewählte Fehlerberichte des Chrome Webbrowsers[1] als Datenbasis genutzt. Zuerst werden die Daten in Text umgewandelt und manuell annotiert, sodass ein Goldstandard erzeugt wird. Die Autoren benutzen Vorbehandlungs-Methoden Stemming und Stop Word Removal, um ihre Ergebnisse zu verbessern. Insgesamt werden von den Autoren 47 Klassifizierer untereinander verglichen.

Using Text Mining Techniques to Extract Rationale from Existing Documentation Der Fokus dieser Arbeit ([31]) liegt auf den Text-Mining-Techniken. Die Autoren, die zum Teil auch an [30] mitgearbeitet haben, benutzen für die Evaluation der Algorithmen Chrome Fehlerberichte als Datengrundlage. Diese werden zuerst manuell annotiert, das heißt mit Klassenlabels versehen. Dabei wurden acht verschiedene Arten von Rationalen unterschieden: *Requirements, Decisions, Alternatives, Arguments, Assumptions, Questions, Answers* und *Procedures*. Die Texte werden dann mithilfe von verschiedenen Preprocessing-Techniken aufbereitet. Bei diesen Techniken vergleichen die Autoren die Resultate untereinander.

Beim Klassifizieren der Texte gehen die Autoren in mehreren Schritten vor. Zuerst, klassifiziert ein binärer Klassifizierer, ob es sich um Rationale handelt. Danach, wird erneut eine binäre Klassifizierung durchgeführt, um Argumentation von Nicht-Argumentation zu separieren. Schließlich werden noch die unterschiedlich Argumentationstypen unterschieden. Auch hier werden Versuche mit verschiedenen Algorithmen durchgeführt und diese untereinander verglichen.

Rationale in Developers' Communication In dieser Dissertation von Alkadhi, Rana Mohammed A werden Rationale in Kommunikationstools von EntwicklerInnen untersucht. Die Autorin präsentiert eine manuelle Methode, genannt *REACT*, zum Erfassen von Rationalen in der Kommunikation der EntwicklerInnen. Darauf aufbauend wird *A-*

[1] http://2011.msrconf.org/msr-challenge.html

REACT konzipiert, da der Aufwand der manuellen Annotation für EntwicklerInnen als zu hoch eingeschätzt wird. A-REACT extrahiert Rationale automatisch. Zuerst werden die Texte mit Preprocessing-Methoden behandelt. Der gewählte Ansatz zum Klassifizieren ist zweistufig angelegt. Zuerst werden die Texte binär klassifiziert in *mit Rationale* und *ohne Rationale*. Die Texte, die Rationale enthalten, werden anschließend genauer Klassifiziert in die Klassen: *Issues*, *Alternatives*, *Pro-arguments*, *Con-arguments* und *Decisions*. Bei der Klassifizierung werden nicht einzelne Sätze, sondern ganze Nachrichten und Kommentare betrachtet, um den Kontext besser erfassen zu können. Ein Problem, auf das Alkadhi, Rana Mohammed A gestoßen ist, ist das ungleichmäßig häufige Auftreten der verschiedenen Wissenstypen. Mithilfe der Data-Balancing-Methoden *Undersampling* und *SMOTE* wurden die Klassen ausbalanciert. Da in dieser Arbeit Artefakte aus unterschiedlichen Teams zur verfügung stehen, führt die Autorin eine Projekt-Kreuzvalidierung durch.

3.2 Synthese

Die Synthese der vorgestellten Arbeiten ist in den Tabellen 3.1 und 3.2 systematisch dargestellt. Die verschiedenen Arbeiten nutzen eine Vielzahl an unterschiedlichen Preprocessing-Methoden und ML-Algorithmen, um die besten Kombinationen für das jeweilige Klassifizierungs-Problem zu finden. Die Größe der Datensätze fällt zum Teil sehr klein aus ([6, 30, 31]), jedoch enthält ein JIRA-Issue mehrere Sätze, die jeweils einzeln klassifiziert werden.

Die Ergebnisse der verwandten Arbeiten zeigen vor allem drei Dinge. Erstens ist das Vorverarbeiten von Textdaten wichtig, um gute Ergebnisse mit einem ML-Algorithmus zu erlangen. Zweitens ist die Wahl des Algorithmus offen. Das heißt, es gibt keinen Algorithmus, der immer am besten abschneidet. Drittens ist die Performanz eines trainierten ML-Modells auf Daten fremder Projekte oder Teams deutlich schlechter als auf den eigenen [1]. Die Autoren in [31, 34] stellen diesen Effekt als Vermutung an, prüfen ihn aber nicht explizit.

Des Weiteren werden von den Autoren nur Algorithmen betrachtet, die nach dem Batch-Prinzip trainiert werden. Es liegen also keine Daten vor wie on-line ML-Algorithmen im Vergleich abschneiden oder wie sie auf fremden Daten weiter lernen.

Tabelle 3.1: Synthese der verwandten Arbeiten in Bezug auf die Voraussetzungen für das Maschinelle Lernen.

Arbeit	**Datenmodell**	**Artefakte**	**Datensatz**	**Preprocessing**	**Machine Learning**
[4]: Bhat, Manoj und Shumaiev, Klym und Biesdorf, Andreas und Hohenstein, Uwe und Matthes, Florian	Architectural Design Decisions (ADD) aus [18]	Issues	1,571 aus *Apache Spark* und *Apache Hadoop Common* Projekten	Stemming, Stop Word Removal und N-Grams, dann Umwandlung in eine *term frequency representation*	5 Algorithmen, jeweils mit verschiedenen Hyperparametern[a]
[6]: Clormann, Jochen	Angelehnt an IBIS	Issues	100 aus dem LUCENE Projekt	Stemming, Stop Word Removal, N-Grams und term frequency representation	Zuerst binär mit 2, dann 5 Klassen mit 4 verschiedenen Algorithmen getestet
[30]: Rogers, Benjamin und Gung, James und Qiao, Yechen und E. Burge, Janet	Binär: *contains rationale*	Fehlerberichte	100, Chrome Webbrowser	Stemming und Stop Word Removal	47 Klassifizierer
[31]: Rogers, Benjamin und Qiao, Yechen und Gung, James und Mathur, Tanmay und E. Burge, Janet	8 verschiedene Arten von Rationalen	Fehlerberichte	200, Chrome Webbrowser	part-of-speech tagging, sentence splitting, verb group chunking, Stemming, Contextual Information Annotation und Sentence Length Annotation	3-stufig: binär Rationale, binär Argumentation, Argumentations-Typen; getestet: beste Algorithmen aus [30]
[1]: Alkadhi, Rana Mohammed A	5 verschiedene Arten von Rationalen	Chat-Nachrichten	8.702 Nachrichten zwischen drei externen Entwicklerteams	Lowercase, Stemming, N-Gram-Tokenization und Umwandlung in eine *term frequency representation*	Data-Balancing, 2-stufig: binär Rationale, dann Rationale-Typ
		Chat-Nachrichten	7.500 Nachrichten von drei verteilten Entwicklungsteams		
		Issue-Kommentare	3.007 Kommentare von drei verteilten Entwicklungsteams		

Tabelle 3.2: Synthese der verwandten Arbeiten in Bezug auf die Resultate für das maschinelle Lernen.

Arbeit	**Beste Ergebnisse**	**Erkenntnisse**
[4]: Bhat, Manoj und Shumaiev, Klym und Biesdorf, Andreas und Hohenstein, Uwe und Matthes, Florian	SVM mit N-Gram (n >= 2) F1: 91.29%	Machbarkeit eines ML-Ansatzes gezeigt
[6]: Clormann, Jochen	Binär Sequential minimal optimization Algorithmus mit F1 = 0,862 und NaïveBayes mit F1 = 0,54	schlechte Ergebnisse auf fremden Daten
[30]: Rogers, Benjamin und Gung, James und Qiao, Yechen und E. Burge, Janet	PART-Algorithmus F1: 59,7 %	guter Einstiegspunkt für eigene Experimente, aber weitere Studien notwendig
[31]: Rogers, Benjamin und Qiao, Yechen und Gung, James und Mathur, Tanmay und E. Burge, Janet	für jede Preprocessing-Methode und Klassifizierungs-Problem individuell angegeben	Preprocessing sehr wichtig für ein gutes Ergebnis, bei den ML-Algorithmen keiner eindeutig am besten
[1]: Alkadhi, Rana Mohammed A	keine Algorithmus, der immer am Besten ist	Binäre Klassifikation vielversprechend, Feine Klassifikation schwankt je nach Typ, Projekt-Kreuzvalidierung: unterschiedliche Begriffe und Jargon → Kompromiss zwischen allgemeiner Anwendbarkeit und Performanz

4 Anforderungen

In diesem Kapitel werden die Anforderungen an die zu implementierende Software gestellt. Zuerst werden Personas definiert. Diese werden aus dem ConDec-Projekt übernommen, um die Konsistenz über das Projekt hinweg zu gewährleisten. In Abschnitt 4.2 werden die notwendigen Daten in einem Diagramm modelliert und erläutert. Anschließend werden im Abschnitt *Funktionale Anforderungen* die notwendigen Funktionen aus den Personas hergeleitet und systematisch erfasst. Zusätzlich zu den funktionalen müssen auch nicht-funktionale Anforderungen erhoben werden. Diese werden in Abschnitt 4.4 aufgeführt. Hierbei wird grundsätzlich auf die Messbarkeit und auf eine Rechtfertigung der Zielgrößen geachtet. Zuletzt werden die Workspaces, die von der neuen Software betroffen sind, erläutert, sowie notwendige Änderungen dargestellt.

Bei der Anforderungserhebung wird die Aufgabenbeschreibung des Praktikums als Ausgangsbasis genutzt.

4.1 Personae

Die Personae dienen dazu sich in einen potenzielle Stakeholder der Software besser vorstellen zu können. Damit soll erleichtert werden verschiedene Sichten auf die Software zu bekommen. Dabei werden neben Themen, die die Software direkt beeinflussen können, wie beispielsweise *Ideale Funktionen*, auch biografische Fakten erfasst. Dies soll es Entwicklern erleichtern sich in eine Persona hinein versetzen zu können. [20]

Die nachfolgenden Personae sind aus dem ConDec-Projekt übernommen worden. Dabei wurden sie aus dem Englischen in die deutsche Sprache übersetzt. Die Personae sind nach Relevanz absteigend sortiert in Rücksprache mit der Betreuerin[1] dieser Arbeit, die als Domänenexpertin agiert.

[1] Anja Kleebaum

Tabelle 4.1: Persona einer Softwareentwicklerin.

Name	Lori Bell
Biografie	22 Jahre alt, ausgebildete Software Entwicklerin. Arbeitet für eine große Softwarefirma die Anforderungen und Aufgaben seit 2005 in JIRA dokumentiert. Sie benutzt IntelliJ als IDE und ist sehr erfahren in Java und Java Script.
Wissen	Arbeitet seit 2 Monaten als Entwicklerin in dieser Firma. Hat vorher nie JIRA benutzt. Hat nie Entscheidungen dokumentiert oder Commit Nachrichten geschrieben.
Bedürfnisse	Einfach bedienbare Plugins. Plugins mit guter Dokumentation und Tutorials.
Frustrationen	Keine Hilfestellung zu Plugins.
Ideale Funktionen	Einscheidungen durch Markieren kenntlich machen. Sehen welche Entscheidungen zu einem JIRA-Issue bereits getroffen wurden.

Tabelle 4.2: Persona einer Rationale Managerin.

Name	Ralph Reed
Biografie	31 Jahre alt, Master in Computer Science, Bachelor in Kommunikationswissenschaften. Arbeitet für eine große Software Firma. Ist in viele verschiedene Projekte eingebunden. Benutzt JIRA Dashboards und Git Konsolen um alle neuen JIRA Issues und Entscheidungselemente im Blick zu behalten.
Wissen	3 Jahre JIRA Erfahrung. Betreut(e) viele Projekte als Rationale Manager.
Bedürfnisse	Möglichkeit Entscheidungswissen aus Issue Kommentaren mit expliziten Entscheidungswissen in Issues zu verbinden.
Frustrationen	Projektbeteiligte die ihre Entscheidungen nicht in Issues und Commits dokumentieren. Technik die nicht funktioniert, aber benutzt werden muss.
Ideale Funktionen	Gefundenes Entscheidungswissen aus Kommentaren und Commits in Issues exportieren zu können. Konfigurieren welches Wissen angezeigt wird. Konfigurieren können ob eine automatische Klassifizierung von Entscheidungswissen stattfinden soll.

4.2 Domänendatendiagramm

Das Domänendatendiagramm (siehe Abbildung 4.1) stellt die Daten dar, die mit dem Klassifizierungsproblem aus dieser Arbeit in Zusammenhang stehen.

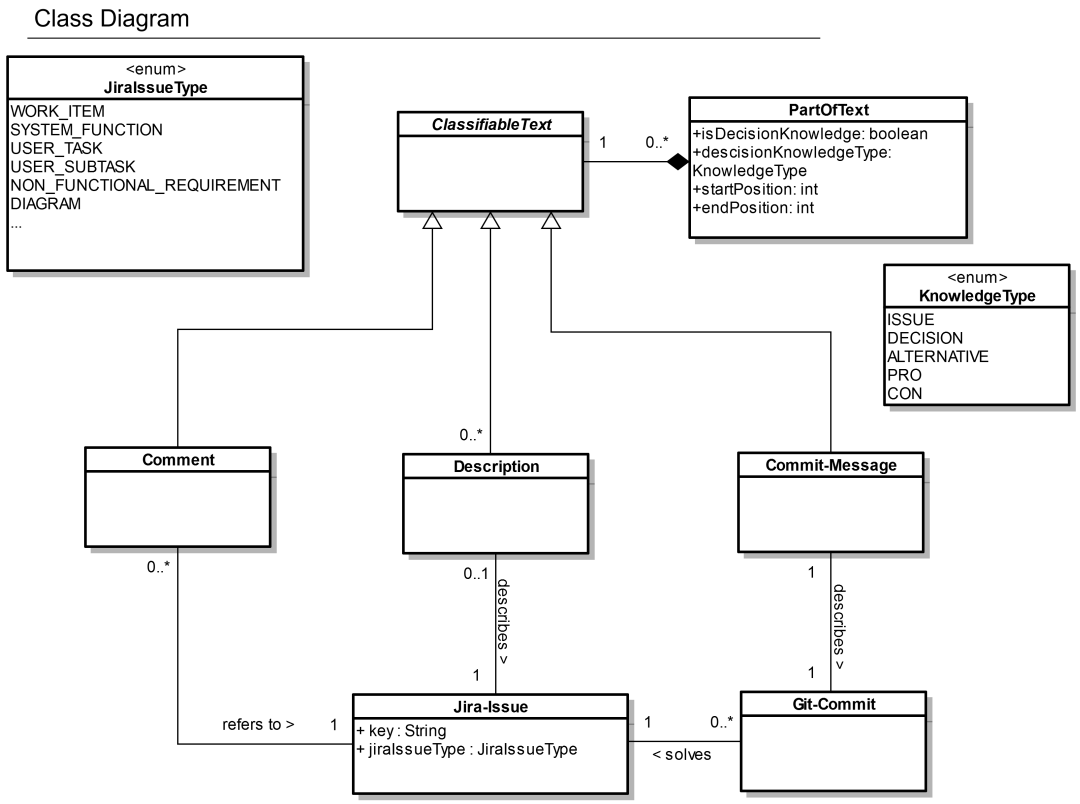

Abbildung 4.1: Domänendatendiagramm auf Basis das UML-Klassendiagramm Notation für zu klassifizierende Daten [2]

4.3 Funktionale Anforderungen

Zuerst werden in diesem Abschnitt die User Tasks (UT), die unterstützt werden sollen, beschrieben. Subtasks (ST) verfeinern dann diese User Tasks. Anschließend werden Systemfunktionen (SF) verfasst und mit User Stories, die aus den Personae abgeleitet sind, begründet.

4.3.1 User Tasks

Die User Tasks und Subtasks werden in Tabelle 4.3 beschrieben. Bei der beschriebenen User Task handelt es ich um das Entwickeln von Software.

Tabelle 4.3: *User Task 1 – Software entwickeln* mit den zugehörigen Subtasks.

UT1 – Software entwickeln	
Sub-Tasks und Variante	Beschreibung
SUBT1 – Software-Entwicklungsprozess für das Projekt definieren	Rationale ManagerInnen definieren den Software-Entwicklungsprozess für das Projekt. Sie entscheiden darüber, wie Entscheidungswissen im Softwareentwicklungsprojekt verwaltet wird.
SUBT2 – Anforderungen erheben, dokumentieren, ändern, verwalten	Rationale ManagerInnen ermitteln und dokumentieren Anforderungen an die Software. Dabei treffen sie Entscheidungen darüber ob Anforderungen aufgenommen werden und mit welcher Priorität sie behandelt werden.
SUBT3 – Code implementieren	EntwicklerInnen implementieren Quellcode anhand der definierten Anforderungen. Alle Design und Entwicklungsentscheidungen werden entsprechend des festgelegten Prozesses dokumentiert.
SUBT4 – Qualität des dokumentierten Entscheidungswissens analysieren	Die Rationale ManagerIn pflegt dokumentiertes Entscheidungswissen, sodass es konsistent und in hoher Qualität vorliegt.

4.3.2 Systemfunktionen

Die Reihenfolge der nachfolgenden Systemfunktionen hat keinen Bezug auf die jeweilige Priorität. Bei jeder Systemfunktion wird der Grad der Umsetzung aus [6] bewertet, sodass der Beitrag dieser Arbeit klar davon abgegrenzt werden kann. Der Grad der Umsetzung kann entweder *Noch nicht umgesetzt*, *Zum Teil umgesetzt* oder *Bereits umgesetzt* sein.

SF1: Klassifizierer aktivieren Als Rationale ManagerIn möchte ich in den Projekteinstellungen den Klassifizierer aktivieren und deaktivieren, sodass ich entscheiden kann, wann es für mein Projekt genutzt wird. Tabelle 4.4 beschreibt die Systemfunktion zu dieser Anforderung. Diese Systemfunktion unterstützt Subtask SUBT1.

SF2: Integration von Git-Commit-Nachricht aktivieren Als Rationale ManagerIn möchte ich in den Projekteinstellungen die Integration von Git-Commit-Nachricht aktivieren und deaktivieren können, sodass ich entscheiden kann, wann die Funktion für

Tabelle 4.4: SF1: Plugin aktivieren

Vorbedingung	JIRA-Projekt existiert, ConDec ist aktiviert.
Eingabe	Projekt Schlüssel, Ist Extraktion aktiviert?
Nachbedingung	Klassifizierer ist entweder aktiviert oder deaktiviert
Ausgabe	Erfolgs- oder Fehlermeldung
Ausnahmen	Projektschlüssel existiert nicht in der Datenbank.
Grad der Umsetzung	Bereits umgesetzt

mein Projekt genutzt wird. Tabelle 4.5 beschreibt die Systemfunktion zu dieser Anforderung. Diese Systemfunktion unterstützt Subtask SUBT1.

Tabelle 4.5: SF2: Integration von Git-Commit-Nachricht aktivieren

Vorbedingung	JIRA-Projekt existiert, ConDec ist aktiviert.
Eingabe	Projekt Schlüssel, Ist Git-Commit-Nachricht Integration aktiviert?
Nachbedingung	Integration ist entweder aktiviert oder deaktiviert
Ausgabe	Erfolgs- oder Fehlermeldung
Ausnahmen	Projektschlüssel existiert nicht in der Datenbank.
Grad der Umsetzung	Noch nicht umgesetzt

SF3: Entscheidungswissen in JIRA-Issue-Kommentaren automatisch klassifizieren
Als eine EntwicklerIn möchte ich Entscheidungswissen in JIRA-Issue-Kommentaren automatisiert klassifizieren lassen, um selbst keine Sätze in Wissenstypen klassifizieren zu müssen. Tabelle 4.6 beschreibt die Systemfunktion zu dieser Anforderung. Diese Systemfunktion unterstützt Subtask SUBT3.

Tabelle 4.6: SF3: Entscheidungswissen in JIRA-Issue-Kommentaren automatisch klassifizieren

Vorbedingung	JIRA-Issue existiert mit mindestens einem Kommentar. Plugin und Klassifizierer sind aktiviert.
Eingabe	JIRA-Issue
Nachbedingung	Klassifikation ist intern gespeichert.
Ausgabe	Text ist als relevantes Entscheidungselement mit bestimmten Wissenstypen oder als nicht relevant gekennzeichnet.
Regeln	Wird ein Satz als relevant klassifiziert, wird der Wissenstyp klassifiziert. Speziell formatierter Text wird nicht klassifiziert. Bereits klassifizierter Text wird nicht klassifiziert.
Grad der Umsetzung	Zum Teil umgesetzt: Klassifizierer muss angepasst werden

SF4: Entscheidungswissen in Git-Commit-Nachrichten automatisch klassifizieren
Als eine EntwicklerIn möchte ich Entscheidungswissen in Git-Commit-Nachrichten au-

tomatisiert klassifizieren lassen, um selbst keine Sätze in Wissenstypen klassifizieren zu müssen. Tabelle 4.7 beschreibt die Systemfunktion zu dieser Anforderung. Diese Systemfunktion unterstützt Subtask SUBT3.

Tabelle 4.7: SF4: Entscheidungswissen in Git-Commit-Nachrichten automatisch klassifizieren

Vorbedingung	Git-Commit existiert mit mindestens einem Kommentar
Eingabe	Git-Commit
Nachbedingung	Klassifikation ist intern gespeichert.
Ausgabe	Text ist als relevantes Entscheidungselement mit bestimmten Wissenstypen oder als nicht relevant gekennzeichnet.
Regeln	Wird ein Satz als relevant klassifiziert, wird der Wissenstyp klassifiziert. Speziell formatierter Text wird nicht klassifiziert. Bereits klassifizierter Text wird nicht klassifiziert.
Grad der Umsetzung	Noch nicht umgesetzt

SF5: Entscheidungswissen in JIRA-Issue-Kommentaren manuell klassifizieren Als eine EntwicklerIn möchte ich Entscheidungswissen in JIRA-Issue-Kommentaren markieren, um Sätze in entsprechende Wissenstypen zu klassifizieren. Tabelle 4.8 beschreibt die Systemfunktion zu dieser Anforderung. Diese Systemfunktion unterstützt Subtask SUBT3.

Tabelle 4.8: SF5: Entscheidungswissen in JIRA-Issue-Kommentaren manuell klassifizieren

Vorbedingung	Issue existiert mit mindestens einem Kommentar.
Eingabe	Satz bzw. Sätze/Teile aus JIRA-Issue-Kommentar und Wissenstyp.
Nachbedingung	Klassifikation ist intern gespeichert.
Ausgabe	Entscheidungswissen ist im Text gekennzeichnet.
Ausnahmen	Die Klassifikation ist unzulässig.
Regeln	Markierte Sätze werden nicht klassifiziert. Die Markierung muss nach der Vorschrift des Systems erfolgen.
Grad der Umsetzung	Bereits umgesetzt

SF6: Entscheidungswissen in Git-Commit-Nachrichten manuell klassifizieren Als eine EntwicklerInmöchte ich Entscheidungswissen in Git-Commit-Nachrichten markieren, um Sätze in entsprechende Wissenstypen zu klassifizieren. Tabelle 4.9 beschreibt die Systemfunktion zu dieser Anforderung. Diese Systemfunktion unterstützt Subtask SUBT3.

SF7: Entscheidungswissen in JIRA-Issue-Kommentaren anzeigen Als eine EntwicklerInoder Rationale-ManagerInmöchte ich Entscheidungswissen in JIRA-Issue-Kommen-

Tabelle 4.9: SF6: Entscheidungswissen in Git-Commit-Nachrichten manuell klassifizieren

Vorbedingung	Git-Commitexistiert mit Git-Commit-Nachricht.
Eingabe	Satz bzw. Sätze/Teile aus Git-Commit-Nachrichtund Wissenstyp.
Nachbedingung	Klassifikation ist intern gespeichert.
Ausgabe	Entscheidungswissen ist im Text gekennzeichnet.
Ausnahmen	Die Klassifikation ist unzulässig.
Regeln	Markierte Sätze werden nicht klassifiziert. Die Markierung muss nach der Vorschrift des Systems erfolgen.
Grad der Umsetzung	Noch nicht umgesetzt

taren einsehen, sodass ich einen Überblick über alle Entscheidungselemente zu diesem JIRA-Issue habe. Tabelle 4.10 beschreibt die Systemfunktion zu dieser Anforderung.

Tabelle 4.10: SF7: Entscheidungswissen in JIRA-Issue-Kommentaren anzeigen

Vorbedingung	JIRA-Issue existiert mit in Kommentaren klassifiziertem Entscheidungswissen.
Eingabe	JIRA-Issue
Nachbedingung	Entscheidungswissen ist angezeigt.
Ausgabe	Entscheidungswissen in übersichtlicher Darstellung
Ausnahmen	Projekt existiert nicht. Keine Kommentare vorhanden.
Regeln	Nur lesende Ansicht
Grad der Umsetzung	Bereits umgesetzt

SF8: Entscheidungswissen aus Git-Commit-Nachrichten anzeigen Als eine EntwicklerInoder Rationale-ManagerInmöchte ich Entscheidungswissen in Git-Commit-Nachrichten einsehen, sodass ich einen Überblick über alle Entscheidungselemente zu diesem JIRA-Issue habe. Tabelle 4.11 beschreibt die Systemfunktion zu dieser Anforderung.

Tabelle 4.11: SF8: Entscheidungswissen aus Git-Commit-Nachrichten anzeigen

Vorbedingung	Git-Commitexistiert mit in Git-Commit-Nachrichtklassifiziertem Entscheidungswissen.
Eingabe	Git-Commit
Nachbedingung	Entscheidungswissen ist angezeigt.
Ausgabe	Entscheidungswissen in übersichtlicher Darstellung
Ausnahmen	Projekt existiert nicht. Keine Git-Commit-Nachrichtvorhanden.
Regeln	Nur lesende Ansicht
Grad der Umsetzung	Noch nicht umgesetzt

SF9: Entscheidungswissen in JIRA-Issue-Kommentaren im Kontext von weiterem Entscheidungswissen, Projektwissen und Systemwissen anzeigen Als eine EntwicklerInoder Rationale-ManagerInmöchte ich Entscheidungswissen in JIRA-Issue-Kommentaren im Kontext von weiterem Wissen einsehen, sodass ich alle vorhanden Entscheidungselemente ansehen und verwalten kann. Tabelle 4.12 beschreibt die Systemfunktion zu dieser Anforderung.

Tabelle 4.12: SF9: Entscheidungswissen in JIRA-Issue-Kommentaren im Kontext von weiterem Entscheidungswissen, Projektwissen und Systemwissen anzeigen

Vorbedingung	Projekt existiert, Entscheidungswissen ist vorhanden
Eingabe	JIRA-Projekt
Nachbedingung	Entscheidungswissen ist angezeigt
Ausgabe	Entscheidungswissen in übersichtlicher Darstellung
Ausnahmen	Projekt existiert nicht. Keine Entscheidungselemente vorhanden.
Grad der Umsetzung	Bereits umgesetzt

SF10: Automatisch sowie manuell klassifiziertes Entscheidungswissen bearbeiten Als eine EntwicklerInmöchte ich klassifiziertes Entscheidungswissen bearbeiten, um Fehler der automatischen und manuellen Klassifikation zu korrigieren. Damit erzeuge ich textuell korrektes Entscheidungswissen, das vollständig dokumentiert und verlinkt ist. Der Wissenstyp gibt die Aussage des Textes wieder. Tabelle 4.13 beschreibt die Systemfunktion zu dieser Anforderung. Diese Systemfunktion unterstützt Subtask SUBT4.

Tabelle 4.13: SF10: Automatisch sowie manuell klassifiziertes Entscheidungswissen bearbeiten

Vorbedingung	JIRA-Issue existiert mit in Kommentaren klassifiziertem Entscheidungswissen
Eingabe	Ausgewähltes Entscheidungselement, gewünschte Änderung
Nachbedingung	Klassifikation ist intern gespeichert. Es wird vermerkt, wenn Entscheidungswissen durch die NutzerIn markiert wurde.
Ausgabe	Erfolgs oder Fehlermeldung.
Grad der Umsetzung	Bereits umgesetzt

SF11: Entscheidungswissen in JIRA-Issue-Kommentaren verlinken, Verlinkung lösen Als eine EntwicklerInmöchte ich Entscheidungswissen in JIRA-Issue-Kommentaren mit anderem Wissen verlinken, um eine Entscheidung vollständig und korrekt zu dokumentieren. Tabelle 4.14 beschreibt die Systemfunktion zu dieser Anforderung. Diese Systemfunktion unterstützt Subtask SUBT4.

Tabelle 4.14: SF11: Entscheidungswissen in JIRA-Issue-Kommentaren verlinken, Verlinkung lösen

Vorbedingung	JIRA-Issue existiert mit in Kommentaren klassifiziertem Entscheidungswissen
Eingabe	Startelement mit Dokumentationsort, Zielelement mit Dokumentationsort.
Nachbedingung	Verlinkung ist intern gespeichert.
Ausgabe	Verlinkung ist auf der Oberfläche sichtbar.
Ausnahmen	Start oder Ziel Element existiert nicht.
Grad der Umsetzung	Bereits umgesetzt

SF12: Explizites Entscheidungswissen zu JIRA-Issue hinzufügen Als EntwicklerInmöchte ich existierendes Entscheidungswissen mit neuen Entscheidungselementen verknüpfen. Dabei möchte ich bestimmen, ob neues Entscheidungswissen als JIRA-Issue oder als JIRA-Issue-Kommentar gespeichert wird, sodass neue Entscheidungselemente entsprechend ihres Kontextes verfügbar sind. Tabelle 4.15 beschreibt die Systemfunktion zu dieser Anforderung. Diese Systemfunktion unterstützt Subtask SUBT2.

Tabelle 4.15: SF12: Explizites Entscheidungswissen zu JIRA-Issue hinzufügen

Vorbedingung	JIRA-Issue existiert.
Eingabe	JIRA-Issue, Beschreibung, Wissenstyp, Dokumentationsort
Nachbedingung	Neues Entscheidungselement ist erstellt und verlinkt.
Ausgabe	Neues Entscheidungselement, entsprechend gewähltem Dokumentationsort.
Ausnahmen	JIRA-Issue existiert nicht.
Grad der Umsetzung	Bereits umgesetzt

SF13: Metriken zur Beurteilung der Vollständigkeit des Entscheidungswissens berechnen Als Rationale-ManagerInmöchte ich Metriken zum Entscheidungswissens einsehen, um zu beurteilen, wie vollständig die getroffenen Entscheidungen in meinem Projekt dokumentiert sind. Tabelle 4.16 beschreibt die Systemfunktion zu dieser Anforderung. Diese Systemfunktion unterstützt Subtask SUBT4.

Tabelle 4.16: SF13: Metriken zur Beurteilung der Vollständigkeit des Entscheidungswissens berechnen

Vorbedingung	Projekt existiert, Entscheidungswissen ist vorhanden
Eingabe	Alle Entscheidungselemente
Nachbedingung	Metriken sind berechnet.
Ausgabe	Metriken in geeigneter Form
Ausnahmen	Keine Entscheidungselemente vorhanden.
Grad der Umsetzung	Bereits umgesetzt

SF14: Klassifizierer mit neuem oder geändertem Entscheidungswissen aktualisieren Als Rationale-ManagerInmöchte ich, dass der Klassifizierer dazu lernt, um falsch klassifiziertes Entscheidungswissen nicht immer wieder manuell verbessern zu müssen. Tabelle 4.17 beschreibt die Systemfunktion zu dieser Anforderung. Diese Systemfunktion unterstützt Subtask SUBT3.

Tabelle 4.17: SF14: Klassifizierer mit neuem oder geändertem Entscheidungswissen aktualisieren

Vorbedingung	neues/geändertes manuell klassifiziertes Entscheidungswissen existiert
Eingabe	Mit Entscheidungswissen annotierter Text
Nachbedingung	Klassifizierer ist aktualisiert
Ausgabe	Erfolg oder Fehlermeldung
Ausnahmen	Klassifizierer ist deaktiviert
Grad der Umsetzung	Noch nicht umgesetzt

SF15: Güte des Klassifizierers berechnen Als Rationale-ManagerInmöchte ich die Güte der Klassifizierer wissen, um feststellen zu können, ob er für das jeweilige Projekt geeignet ist. Tabelle 4.18 beschreibt die Systemfunktion zu dieser Anforderung. Diese Systemfunktion unterstützt Subtask SUBT3.

Tabelle 4.18: SF15: Güte des Klassifizierers berechnen

Vorbedingung	validiertes Entscheidungswissen existiert im Projekt
Eingabe	validiertes Entscheidungswissen
Nachbedingung	-
Ausgabe	berechnete Metriken über die Güte des Klassifizierers
Ausnahmen	Klassifizierer ist deaktiviert
Grad der Umsetzung	Noch nicht umgesetzt

4.4 Nicht-funktionale Anforderungen

Nicht-funktionale Anforderungen (NFR) beschreiben nicht *was* ein System macht, sondern mit welcher Qualität. Die Kategorien der NFR sind aus [28] übernommen. In den nachfolgenden Tabellen sind die NFR aufgelistet. Hierbei gehört zu jeder Anforderung eine Beschreibung, die Begründung (Rationale), wie die Erfüllung gemessen wird, welche konkreten Messwerte als *erfolgreich* zählen und die Phase des Projektes, in welcher diese Anforderung beachtet werden muss. Wie auch bei den Systemfunktionen sagt die Reihenfolge nichts über die Priorität der Anforderungen aus.

Tabelle 4.19: NFR1: Funktionale Vollständigkeit

Kategorie	Benutzbarkeit
Beschreibung	Das entwickelte System bildet alle dokumentierten Anforderungen ab. Die Anforderungen decken alle Aspekte des Ausschreibungsdokumentes ab.
Rationale	Nicht umgesetzte funktionale Anforderungen bedeuten, dass das System unvollständig ist. Dadurch kann es möglicherweise nicht wie vorhergesehen genutzt werden.
Messung	Manuelle Systemtests überprüfen, ob die Systemfunktionen vollständig vorhanden sind.
Zielgröße	Alle Systemfunktionen sind vorhanden.
Zeitpunkt	Entwurfs- und Entwicklungsphase

Tabelle 4.20: NFR2: Funktionale Korrektheit

Kategorie	Benutzbarkeit
Beschreibung	Das entwickelte System wird auf seine funktionale Korrektheit geprüft.
Rationale	Inkorrekt implementierte Funktionalität macht das System schwer bis unbenutzbar.
Messung	Durch Komponenten- und Systemtests werden die umgesetzten Funktionen getestet.
Zielgröße	Die Abdeckung der Komponententests soll besser als die bisherigen Durchschnittswerte. Bei den Systemtests darf kein Test fehlschlagen.
Zeitpunkt	Entwurfs- und Testphase

Tabelle 4.21: NFR3: Zeitverhalten

Kategorie	Performanz
Beschreibung	Das Klassifizieren von Entscheidungswissen in JIRA-Issue-Kommentaren und Git-Commit-Nachrichten benötigt Rechenaufwand und nimmt somit aus Benutzersicht Zeit in Anspruch.
Rationale	Zu langes Warten auf die automatische Klassifikation der Texte kann zu Frustrationen bei Nutzern führen.
Messung	Dauer des Klassifizierens eines Satzes durch den Klassifizierer.
Zielgröße	Der neue Klassifizierer darf bei der Berechnung nicht länger als 100ms benötigen. (Zielgröße nach [24])
Zeitpunkt	Entwicklungsphase

4.5 Arbeitsbereiche (Workspaces)

Workspaces modellieren die Struktur der Benutzeroberfläche. Die erstellten Diagramme stellen dar, in welchem Kontext der Oberfläche bestimmte Systemfunktionen aufrufbar sind. [29] Das entsprechende Diagramm ist in Abbildung 4.2 dargestellt.

Tabelle 4.22: NFR4: Interoperabilität

Kategorie	Übertragbarkeit
Beschreibung	Die entwickelte Komponente fügt sich in ConDec ein.
Rationale	Zu viele benötigte Schritte führen zu Frustration bei RationaleManagerInn. Diese führen dann möglicherweise die neuen Funktionen von ConDec nicht ein.
Messung	Komplexität der Aktivierung des Plugins.
Zielgröße	Die Integration von Git-Commit-Nachrichten benötigt weniger als 5 Aktionen, ab dem Login in JIRA.
Zeitpunkt	Entwurfs- und Entwicklungsphase

Tabelle 4.23: NFR5: Ästhetik der Benutzeroberfläche

Kategorie	Benutzbarkeit
Beschreibung	Farbliche Kennzeichnung verschiedener Wissenstypen.
Rationale	Eine farbliche Kennzeichnung erlaubt den Nutzern einen schnellen Überblick zu erlangen.
Messung	Manuelles Prüfen, ob das Entscheidungswissenin einem zufällig ausgewählten JIRA-Issue farblich markiert ist.
Zielgröße	Alle Entscheidungswissen-Elemente sind farblich gekennzeichnet.
Zeitpunkt	Entwicklungsphase

Tabelle 4.24: NFR6: Fehlertoleranz

Kategorie	Zuverlässigkeit
Beschreibung	Das System verhält sich stabil gegenüber fehlerhaften Nutzereingaben.
Rationale	Nutzer können durch Unverständnis oder Unachtsamkeit das System anders bedienen als es geplant war. Wenn das System infolgedessen abstürzt werden die Nutzer gefrustet und hören möglicherweise auf das Tool zu benutzen.
Messung	Durch Komponententests werden einzelne Module auf ihr Verhalten bei Fehleingaben getestet.
Zielgröße	Die Abdeckung der Komponententests soll besser als der im Projekt vorgeschriebene Wert von 75% Zeilenabdeckung. Das Einhalten wird toolgestützt automatisch geprüft.
Zeitpunkt	Entwicklungs- und Testphase

Tabelle 4.25: NFR7: Statische Codequalität

Kategorie	Wartbarkeit
Beschreibung	Der implementierte Code soll den Qualitätsstandards des Projektes entsprechen. Statische Codequalität prüft unter Anderem die Formatierung des Quelltextes und die Testabdeckung.
Rationale	Die Einhaltung erleichtert das Warten des Codes. Dies ist besonders wichtig, da Dritte an dem Projekt in der Zukunft weiterarbeiten.
Messung	Einhalten der im ConDec-Projekt spezifizierten Metriken wird toolgestützt automatisch geprüft.
Zielgröße	Besser als die bisherigen Durchschnittswerte.
Zeitpunkt	Entwicklungs- und Testphase

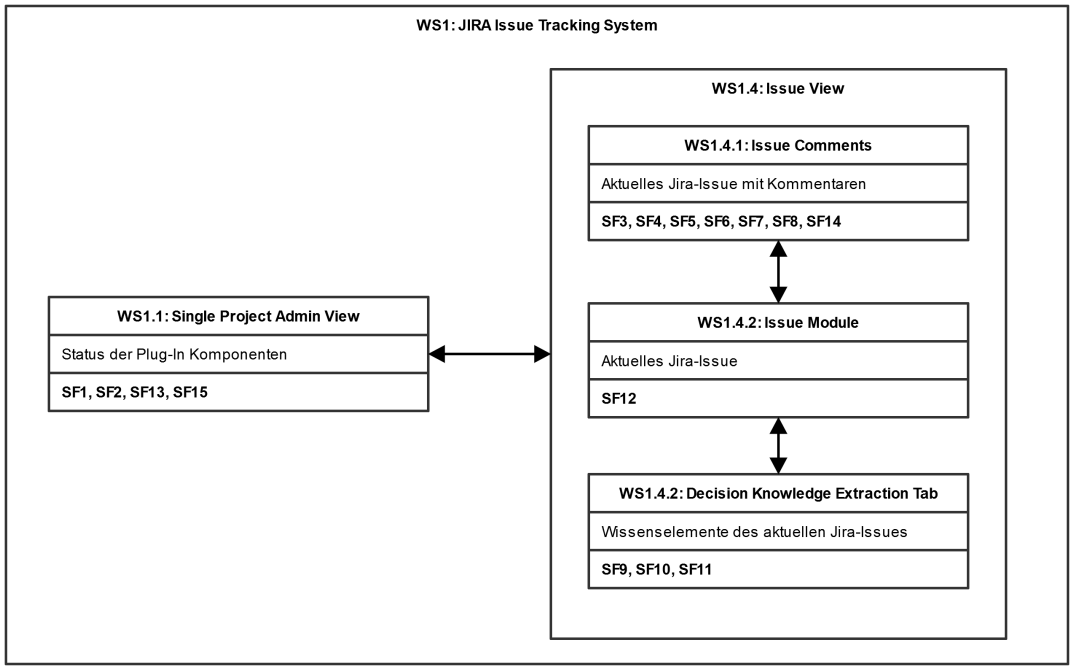

Abbildung 4.2: UI Strukturdiagramm zu ConDec mit allen relevanten Systemfunktionen.

5 Auswahl des Klassifizierers

In diesem Kapitel werden verschiedene Klassifizierungs-Algorithmen und Preprocessing-Verfahren getestet und verglichen. Der am besten geeignete Algorithmus wird dann bei der Implementierung umgesetzt und in das existierende Plugin integriert.

5.1 Ziele

Ziel ist es einen Algorithmus zu implementieren, der mit neuen Daten dazu lernt. Dies ist notwendig, da in verschiedenen Projekten und von verschiedenen Beteiligten in diesen Projekten unterschiedliche Begriffe und Schreibstile benutzt werden. Die Eignung eines Klassifizierers für Entscheidungswissen auf Daten von einem anderen Projekt, als das auf dem er ursprünglich trainiert wurde, ist wesentlich schlechter als auf dem Trainings-Projekt [1]. Welcher Klassifizierer dafür am besten geeignet ist, soll in diesem Kapitel experimentell ermittelt werden. Um die Eignung zu messen werden die F1-Werte betrachtet.

5.2 Datensätze

Um die Performanz der Algorithmen messen zu können müssen sie mithilfe von Daten evaluiert werden. Hierbei kommen zwei Datensätze zum Einsatz. Einerseits JIRA-Issue-Kommentare des LUCENE-Projektes und andererseits JIRA-Issue-Kommentare des ConDec-Projektes. Dadurch, dass die Datensätze aus verschiedenen Projekten stammen, kann die Cross-Projekt-Validation aus [1] angewendet werden, um zu testen, wie gut die Algorithmen verallgemeinern können. In beiden Projekten sind die Kommentare in der englischen Sprache verfasst.

JIRA-Issue-Kommentar des LUCENE-Projektes Dieser Datensatz besteht aus 2446 Sätzen aus JIRA-Issue-Kommentaren. Davon sind 1780 Sätze relevant im Bezug auf

Entscheidungswissen. Bei diesem Datensatz kann ein relevanter Satz mehreren Klassen zugeordnet sein. Zum Beispiel wurde der Satz " *Yea, I like having them listed because it makes it easier to go back and look at them and decide which ones to add.*äls Alternative und Pro-Argument gekennzeichnet. Eine Auflistung der Verteilung des Entscheidungswissens ist in Tabelle 5.1 dargestellt.

JIRA-Issue-Kommentare des ConDec-Projektes Der Datensatz der JIRA-Issue-Kommentare des ConDec-Projektes besteht aus insgesamt 2546 Sätzen. Davon beinhalten 590 Sätze relevantes Entscheidungswissen. Aus dem Datensatz ist von 600 Sätzen die Klassifikation manuell verifiziert worden. Bei den verifizierten Sätzen handelt es sich fast ausschließlich um Sätze mit relevantem Entscheidungswissen.

Tabelle 5.1: Übersicht über die Verteilung der Typen des Entscheidungswissens in den Datensätzen.

	Lucene	**ConDec**	**ConDec (verifiziert)**
Gesamt	2446	2546	600
Relevant	1780	590	561
Issue	259	129	122
Alternative	828	103	96
Pro-Argument	543	119	109
Contra-Argument	284	98	95
Entscheidung	207	141	139

5.3 Durchführung

Zuerst werden die Daten geladen und in eine gleiche Form gebracht. Im ConDec-Datensatz sind die Sätze nicht explizit als relevant gekennzeichnet. Dieses Feld wird deshalb aus den Feldern für die Entscheidungswissenstypen berechnet. Ist ein Entscheidungswissenstyp vorhanden, so ist der Satz relevant.

Nachdem die Datensätze einen identischen Aufbau haben, können die Sätze vorbehandelt werden. Dieses Preprocessing orientiert sich an den vielversprechendsten Ergebnissen aus der Synthese (Kapitel 3.2).

Zuerst werden alle Zeichen in Kleinbuchstaben umgewandelt. Durch einen Tokenizer werden die Sätze anschließend in Listen von Wörtern und Zeichen umgewandelt. Aus diesen werden die Stoppwörter entfernt. Danach werden alle Links, Zahlen, lange Wörter und Wörter mit Unterstrichen mithilfe von regulären Ausdrücken durch jeweils feste Werte ersetzt. Anschließend werden die Wörter durch Lemmatisieren in ihre Grundform

übertragen. Dabei kommt der WordNet Lemmatisierer zum Einsatz. Die Stoppwörter, der Lemmatisierer und der Stemmer die zum Einsatz kommen, stammen aus dem nltk-Modul[1] für Python.

Ziel des Preprocessing ist es, dass Tokens mit der gleichen Bedeutung unabhängig von ihrer ursprünglichen Schreibweise als identisch erkannt werden. Deswegen werden den Wort-Tokens anschließend numerische Werte zugeordnet. Gleiche Tokens erhalten denselben Wert. Diese Werte werden mithilfe von GloVe berechnet [27]. Um eine gute Performanz bei den SVM-Klassifizierern zu erreichen sind die Werte auf den Bereich $[-1, 1]$ skaliert [14].

Abschließend werden aus den vorbereiteten Wortwerte-Listen N-Gramme, mit $N = 3$, erzeugt. Die N-Gramme erlauben eine Einbettung der Wörter in ihre Umgebung, was es erlaubt den Kontext bei der Klassifizierung zu beachten. Diese N-Gramme nutzen wir zum Trainieren und Testen der Klassifizierungs-Algorithmen.

Listing 5.1: Preprocessing Schritte als Pseudocode

```
1 def preprocess_sentence(sentence):
2  sentence.lower()
3  tokenize(sentence)
4  remove_stopwords(sentence)
5  replace_using_regex(sentence, regex)
6  lemmatize(sentence)
7  words_to_numbers(sentence)
8  generate_n_grams(sentence, n=3)
9  return sentence
```

Bei den getesteten Algorithmen handelt es sich um: linearer Support Vector Klassifizierer mit Stochastic Gradient Descent lernen (SGDClassifier), Support Vector Klassifizierer mit Radial basis function -, Polynomia - und Sigmoid-Kernel (SVC), gaußscher Naive Bayes (GaussianNB) und Passive-Aggressive-Klassifizierer (PassiveAggressiveClassifier) [2]. Diese Algorithmen, die untereinander verglichen werden, müssen inkrementell lernen können. Um die Güte der einzelnen Algorithmen zu bewerten, wird eine 10-fache Kreuzvalidierung durchgeführt und der durchschnittliche F1-Wert betrachtet. Gelernt werden die Klassen der vorverarbeiteten 3-Gramme. Nachdem der beste Klassifizierer ausgewählt wurde, wird dieser für das inkrementelle Lernen angepasst. Abschließend werden die Resultate der inkrementellen Implementierung, mit und ohne Vor-Trainieren, mit

[1] Natural Language Toolkit: https://www.nltk.org/ (Zuletzt abgerufen am: 08.10.2019)
[2] Die Implementierung in sklearn wurde genutzt: https://scikit-learn.org (Zuletzt abgerufen am: 12.10.2019)

denen der nicht-inkrementellen Version durch Kreuzvalidierung zwischen verschiedenen Projekten (siehe [1]) verglichen.

5.4 Ergebnisse

Aufgrund der unausgeglichenen Daten wird SMOTE genutzt, um die Aussagekraft der Ergebnisse zu erhöhen. Der ConDec Datensatz mit unverifizierten Daten wurde aus der Analyse ausgeschlossen, da hier falsch klassifiziertes Entscheidungswissen die Resultate der Klassifizierer negativ beeinflusst.

Grob-körniger Klassifizierer Der grob-körnige Klassifizierer, in der Literatur auch binärer Klassifizierer genannt, unterscheidet zwischen relevantem und irrelevantem Texten. Die Ergebnisse sind in Tabelle 5.2 dargestellt, mit den besten Werten jeweils fettgedruckt. Wie sich in der Literaturrecherche bereits angedeutet hat, ist eine Support Vector Machine der beste Klassifizierer. Dieser Klassifizierer erreicht auf dem Lucene Datensatz einen F1-Wert von 0.81 und auf dem ConDec Datensatz mit den verifizierten Daten einen Wert von 0.91.

Tabelle 5.2: Ergebnisse der Experimente zur Eignung des groben Klassifizierers.

	Lucene	ConDec (verifiziert)
SGDClassifier(loss=hinge)	0.74	0.77
SVC(kernel=rbf)	**0.81**	**0.91**
SVC(kernel=poly)	0.77	0.82
SVC(kernel=sigmoid)	0.69	0.76
GaussianNB	0.74	0.79
PassiveAggressiveClassifier	0.66	0.67

Fein-körniger Klassifizierer Die Ergebnisse des feinkörnigen Klassifizierers (Tabelle 5.3) sind jeweils die arithmetischen Mittelwerte über alle Klassen hinweg. Auch hier ist, wie auch schon beim groben Klassifizierer, eine Support Vector Machine der am besten abschneidende Klassifizierer.

In Abbildung 5.1 ist die Streuung der Resultate des fein-körnigen Support Vector Klassifizierer dargestellt. Dadurch soll die Güte des am besten abgeschnittenen Klassifizierers den einzelnen Klassen nach verdeutlicht werden. Die negativen Ausreißer der Klasse *Issue* gehören dem Lucene Datensatz an.

Tabelle 5.3: Ergebnisse der Experimente zur Eignung des fein-körnigen Klassifizierers.

	Lucene	ConDec (verifiziert)
SGDClassifier(loss=hinge)	0.66	0.59
SVC(kernel=rbf)	**0.76**	**0.78**
SVC(kernel=poly)	0.71	0.55
SVC(kernel=sigmoid)	0.61	0.62
GaussianNB	0.67	0.65
PassiveAggressiveClassifier	0.58	0.51

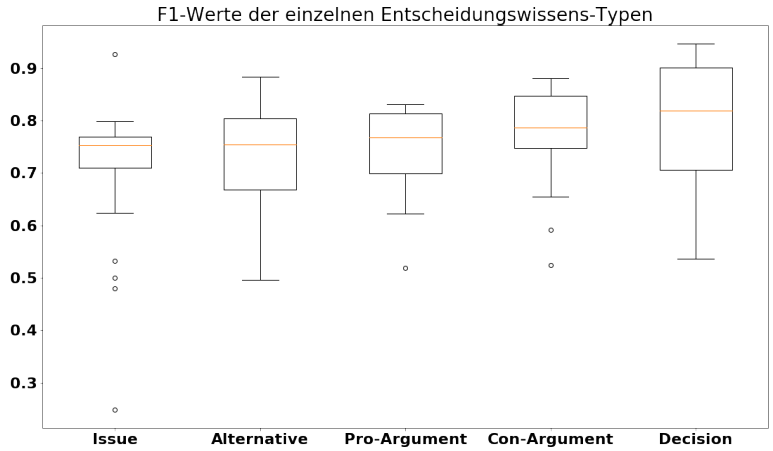

Abbildung 5.1: Boxplot der F1-Werte des am besten abgeschnittenen Klassifizierers

On-line Klassifizierer Um die Vorteile des inkrementellen Lernens zu messen, wird eine Kreuzvalidierung zwischen verschiedenen Projekten nach [1] durchgeführt. Unterschiedliche Domänensprache und Sprachgewohnheiten von individuellen EntwicklerInnen können die Klassifizierungsresultate auf fremden Projektdaten negativ beeinflussen. Im Folgenden wird der grob-körnige Klassifizierer mit durch SMOTE angereicherte Trainingsdaten vorgestellt. Jedes Experiment wurde 100 Mal wiederholt. Und die jeweiligen Medianwerte sind in Abbildung 5.2 dargestellt. Der ConDec-Datensatz wurde in 60 circa gleich große Bündel aufgeteilt. Die Schritte auf der Abszisse geben an nach wie vielen Bündeln die Evaluation auf den verbleibenden, noch nicht zum Weiter-Trainieren genutzten, Daten des ConDec-Datensatzes durchgeführt wurde. Des Weiteren wurden drei verschiedene Testszenarien entworfen.

Zuerst wurde *Lernen nur auf Lucene*, dargestellt in Blau, getestet. Diese Gerade zeigt den durchschnittlichen F1-Wert eines Klassifizierer, der nur auf dem Lucene-Datensatz trainiert und anschließend auf dem verifizierten ConDec-Datensatz getestet wurde. Als nächstes wurde *Vorlernen auf Lucene, inkrementell lernen auf ConDec* geprüft. Dabei ist zu sehen, dass der F1-Wert mit den Testschritten leicht steigt. Dies wird durch das inkrementelle Lernen erreicht. Dadurch, dass der Klassifizierer bereits auf dem Lucene-

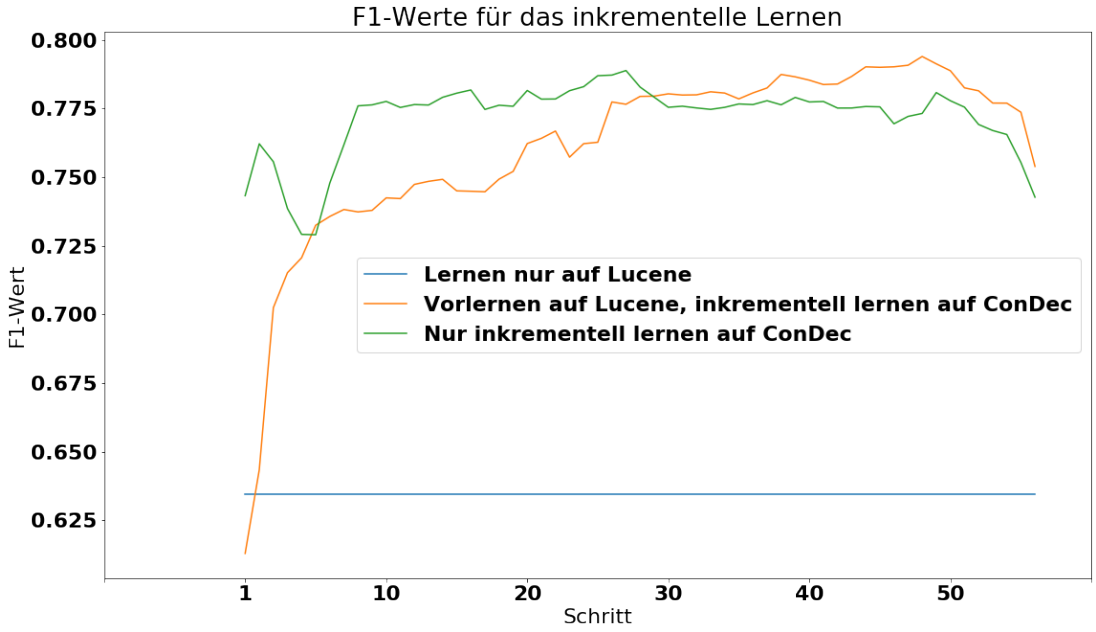

Abbildung 5.2: Entwicklung der F1-Werte des grob-körnigen Klassifizierers beim inkrementellen Lernen.

Datensatz vor-trainiert wurde, ist die anfängliche Erkennungsrate mit einem F1-Wert von circa 0.75 deutlich besser als der Wert von *Nur inkrementell Lernen auf ConDec*. Der letzte getestete Klassifizierer wurde nicht vor-trainiert. Die Klassifizierungsgüte ist deshalb anfangs schlechter als bei den beiden anderen Testszenarien.

Aus diesem Diagramm geht hervor, dass bei genug Trainingsdaten auf dem neuen Projekt der zweite und dritte Testfall keinen großen Unterschied aufweisen. Jedoch ist die Güte des vor-trainierten Klassifizierers Anfangs deutlich besser als die des nicht Vor-Trainierten.

5.5 Diskussion

Wie anhand der Ergebnisse der Experimente gezeigt werden kann, ist die Güte des Klassifizierers verbessert worden. Der alte beste Klassifizierer aus [6] hatte einen F1-Wert von 0.86, was in etwa dem Mittelwert der Resultate des besten groben Klassifizierers in den Experimenten in dieser Arbeit entspricht. Die Güte des fein-körnigen Klassifizierers hingegen wurde jedoch deutlich gesteigert. Der zu vorige F1-Wert betrug lediglich 0.56.

Aufgrund der Ergebnisse der Experimente zum inkrementellen Lernen, dargestellt Grafik 5.2, wird bei der Implementierung ein vor-trainierter Klassifizierer umgesetzt. Dieser

wird so implementiert, dass er durch inkrementelles Lernen auf den Daten des jeweiligen Projektes, auf dem er zum Einsatz kommt, weiterlernt.

Die Erweiterung des Klassifizierers um die Möglichkeit dazu lernen zu können, war zuvor noch nicht gegeben. Diese ist aber notwendig, um in fremden Projekten ebenfalls gute Klassifizierungsresultate zu erzielen.

6 Entwurf und Implementierung

In diesem Kapitel werden die wichtigsten Entscheidungen zur Umsetzung der Anforderungen aus Kapitel 4 präsentiert. Die Folgen der Entscheidungen für die Architektur werden in Form von Diagrammen dargestellt. Es wird nur auf Systemfunktionen und damit verbundene Entscheidungen eingegangen, falls sie nicht aus [6] übernommen wurden (siehe *Grad der Umsetzung* in Abschnitt 4.3). Zuerst wird im Abschnitt Architektur des ConDec JIRA-Plug-Ins der Aufbau des ConDec-Plugins erklärt. Anschließend werden in Abschnitt 6.2 die Umsetzung der einzelnen Systemfunktionen mit den hierbei getroffenen Entscheidungen dargelegt. Im Abschnitt 6.3 werden die Ergebnisse und die schlussendliche Architektur präsentiert.

6.1 Architektur des ConDec JIRA-Plug-Ins

Mithilfe des ConDec-Plug-Ins für Jira kann Entscheidungswissen erfasst und verwaltet werden. Das Projekt ist quelloffen und der Quellcode kann über GitHub abgerufen werden [1].

Das Plug-In besteht aus verschiedenen Komponenten, die je nach Vorlieben aktiviert oder deaktiviert werden können (siehe Abbildung 6.1). Diese einzelnen Komponenten verwalten Entscheidungswissen aus unterschiedlichen Quellen, zum Beispiel Git-Commit oder JIRA-Issue-Kommentar. Die hieraus gewonnen Daten werden in einer Datenbank gespeichert und können, je nach Komponente, in Jira betrachtet und analysiert werden. Zusätzlich werden Daten über REST-Schnittstellen nach außen zur Verfügung gestellt.

Wie im Abschnitt *Funktionale Anforderungen* in den Systemfunktionen beschrieben, sind die Anforderungen zum Teil bereits umgesetzt. Die neuen Funktionen sollen die alten Funktionen möglichst erweitern, anstatt diese zu ersetzen gemäß dem open-closed Prinzip [23].

[1] ConDec-Plugin auf GitHub: https://github.com/cures-hub/cures-condec-jira (Zuletzt abgerufen am: 08.11.2019)

Die bereits vorhandene Architektur und Technologien des Plug-Ins haben zur Folge, dass einige Entscheidungsprobleme keine eigenen Alternativen zulassen.

Tabelle 6.1: Architektur Entscheidungsprobleme

Entscheidungsproblem	Entscheidung
Welche Programmiersprachen sollen genutzt werden?	Java, JavaScript und Apache Velocity, da diese bereits genutzt werden.
Wie sollen die neuen Funktionen mit den vorhandenen ConDec Funktionalitäten verknüpft werden?	Integrieren in das ConDec Plug-In.

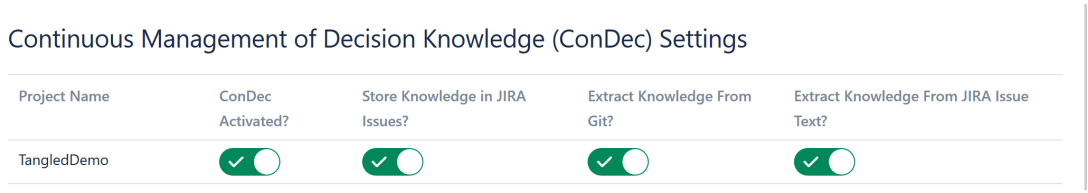

Abbildung 6.1: Bildschirmaufnahme der ConDec-Einstellungen, um Komponenten für einzelne Projekte zu aktivieren oder deaktivieren.

6.2 Umsetzung der Systemfunktionen

6.2.1 SF2: Integration von Git-Commit-Nachricht aktivieren

Die Systemfunktion *SF2: Integration von Git-Commit-Nachricht aktivieren* wird in dem Arbeitsbereich WS1.1: Single Project Admin View umgesetzt, dargestellt in Abbildung 4.2 in Abschnitt 4.5. Bevor die Systemfunktion umgesetzt wird, muss das Entscheidungsproblem aus Abbildung 6.2 gelöst werden.

Das Ergebnis auf der Oberfläche ist in Abbildung 6.3 dargestellt. Der Nutzer kann einstellen welche Art von Git-Commit integriert werden sollen, Squashed oder Feature-Branch Commits. Feature-Branch Commits sind einzelne Commits die einem Entwicklungszweig neue Änderungen hinzufügen. Squashed Commits fassen mehrere Commits zusammen bevor sie einem anderen Entwicklungszweig hinzugefügt werden. Die Schalter werden deaktiviert, falls die *Extract from Git?*-Option ausgeschaltet wird. Die Einstellungen werden dann auch auf der Serverseite deaktiviert. Alle Einstellungen auf der Benutzeroberfläche werden über eine REST-Schnittstelle an das Plug-In gesendet und dort persistiert.

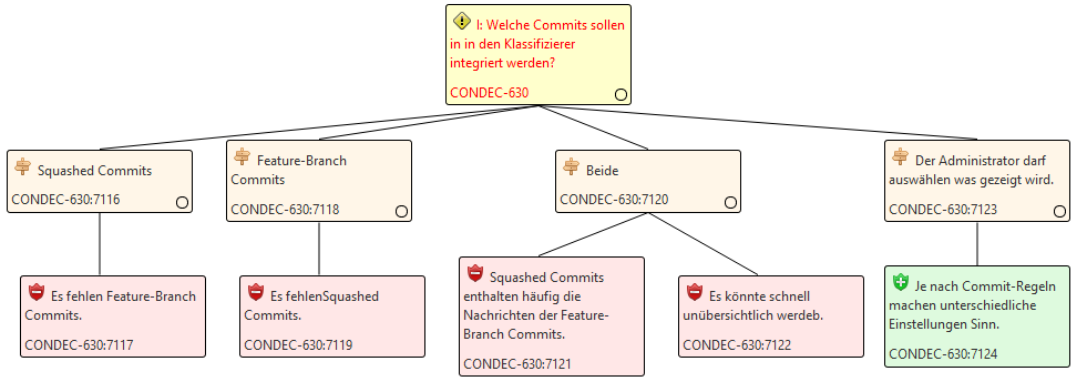

Abbildung 6.2: Entscheidungsproblem zu der Frage *Welche Commits sollen in in den Klassifizierer integriert werden?*, die für Systemfunktion SF2: Integration von Git-Commit-Nachricht aktivieren gelöst werden muss.

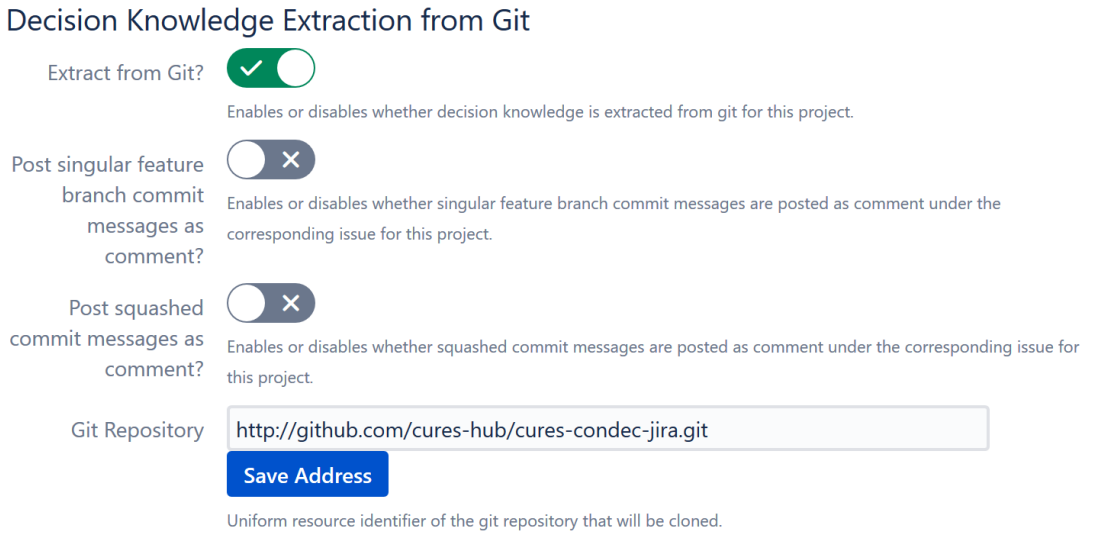

Abbildung 6.3: Bildschirmaufnahme der ConDec-Einstellungen zur Aktivierung und Deaktivierung der Git-Commit-Nachricht-Ingetration aus WS1.1.

6.2.2 SF3: Entscheidungswissen in JIRA-Issue-Kommentaren automatisch klassifizieren

Um Systemfunktion SF3 umzusetzen, wurden die Ergebnisse aus den Kapiteln Verwandte Arbeiten und *Auswahl des Klassifizierers* umgesetzt. Der Klassifizierer ist zweistufig aufgebaut, zuerst binär und danach feingranular. Als Klassifizierer kommen jeweils Support-Vector-Machines zu Einsatz. Dieser kann mithilfe von ARFF-Dateien vortrainiert werden. Ein Standard-Trainingsdatei wird mit dem Plug-In ausgeliefert. Zur Umsetzung wurde die SMILE (Statistical Machine Intelligence and Learning Engi-

ne) Bibliothek verwendet [2]. Die Rechtfertigung für die Entscheidung SMILE zu nutzen ist in Abbildung 6.4 dargestellt. Der Klassifizierer klassifiziert Sätze aus JIRA-Issue-Kommentaren. Anschließend werden die Kommentare durch Makros angereichert, um die Klassifizierungsergebnisse für Nutzer sichtbar zu machen. Die Ergebnisse werden außerdem persistiert.

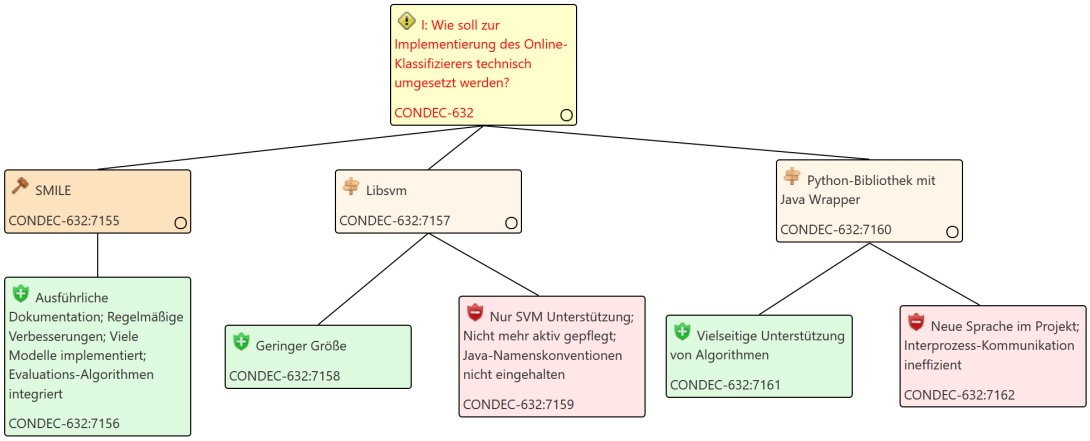

Abbildung 6.4: Entscheidung zur Problemstellung wie der Klassifizieres umgesetzt werden soll.

6.2.3 SF4: Entscheidungswissen in Git-Commit-Nachrichten automatisch klassifizieren

Bei der Umsetzung von Systemfunktion SF4 muss das Entscheidungsproblem *Wie sollen die Git-Commit-Nachricht klassifiziert werden?* geklärt werden (siehe Abbildung 6.5). Die in Abschnitt 6.2.5 getroffene Entscheidung die klassifizierte Git-Commit-Nachricht in den JIRA-Issue-Kommentar anzuzeigen, unterstützt die Umsetzung dieser Funktion. Die Git-Commit-Nachrichten werden als Kommentar unter einem JIRA-Issue hinzugefügt und anschließend wie normale Kommentare verarbeitet. Sie enthalten zusätzliche Metainformation, um Duplikate zu verhindern und die Nachvollziehbarkeit zu erleichtern. Die Metainformation beinhalten den Namen des Autors des Git-Commit, den Branch zu dem der Commit gehört und den Hashwert des Commits.

[2] Webseite von SMILE: http://haifengl.github.io/smile/ (Zuletzt abgerufen am: 09.11.2019)

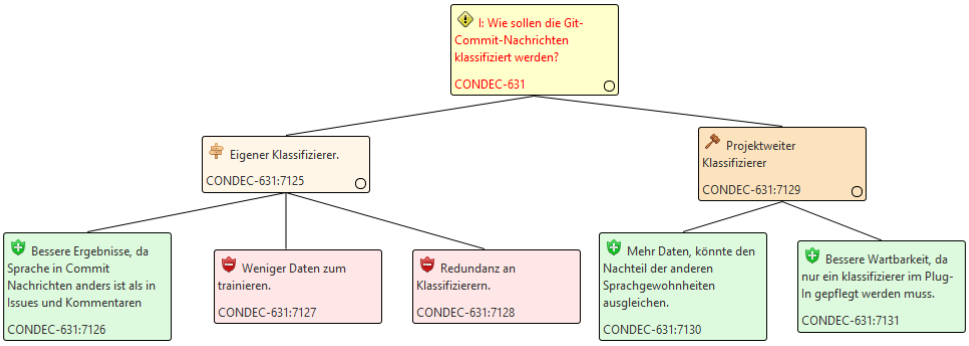

Abbildung 6.5: Entscheidung zur Problemstellung wie Git-Commit-Nachrichten klassifiziert werden sollen.

6.2.4 SF6: Entscheidungswissen in Git-Commit-Nachrichten manuell klassifizieren

Die in der Umsetzung von Systemfunktion SF8: Entscheidungswissen aus Git-Commit-Nachrichten anzeigen getroffene Entscheidung, klassifizierte Git-Commit-Nachrichten in den JIRA-Issue-Kommentaren anzuzeigen (siehe Abschnitt 6.2.5), unterstützt die Umsetzung dieser Systemfunktion. Das manuelle Klassifizieren von Entscheidungswissen in JIRA-Issue-Kommentar wurde von SF10: Automatisch sowie manuell klassifiziertes Entscheidungswissen bearbeiten bereits umgesetzt. Diese Funktionalität wird hier wiederverwendet. Dies erleichtert die Wartung des Quellcodes, da Redundanz vermieden wird.

6.2.5 SF8: Entscheidungswissen aus Git-Commit-Nachrichten anzeigen

Um diese Systemfunktion umzusetzen, müssen mehrere Entscheidungsprobleme gelöst werden. Zuerst muss entschieden werden, an welcher Stelle das klassifizierte Wissen aus den Git-Commit-Nachrichten angezeigt werden soll (siehe Abbildung 6.6). Die Entscheidung fiel darauf, sie in den JIRA-Issue-Kommentar des JIRA-Issue anzuzeigen, welches zu dem Entwicklungszweig gehört. Das wichtigste Pro-Argument war, dass die bereits implementierte Funktionalität für Entscheidungswissen in JIRA-Issue-Kommentar somit wiederverwendet werden kann.

Zusätzlich musste darauf aufbauend entschieden werden, unter welchem Nutzer die Kommentare gepostet werden. Die Entscheidung fiel darauf ein eigenes Konto programmatisch zu erstellen, den *GIT-COMMIT-COMMENTATOR*. Die Alternativen und Argumente zu dieser Entscheidung sind in Abbildung 6.7 dargestellt.

Das Ergebnis der Implementierung aus Nutzersicht ist in Abbildung 6.8 dargestellt. Hierbei handelt es sich um eine Bildschirmaufnahme einer Git-Commit-Nachricht die, als JIRA-Issue-Kommentar unter dem verbundenen JIRA-Issue, gepostet wurde.

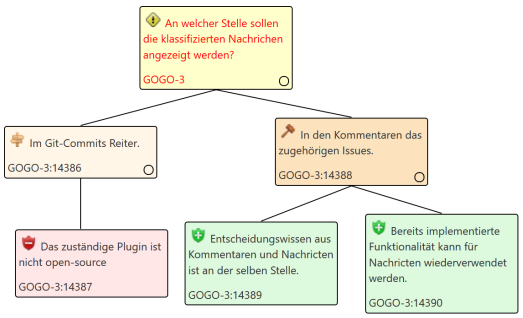

Abbildung 6.6: Entscheidung zur Problemstellung wo die Git-Commit-Nachricht angezeigt werden sollen.

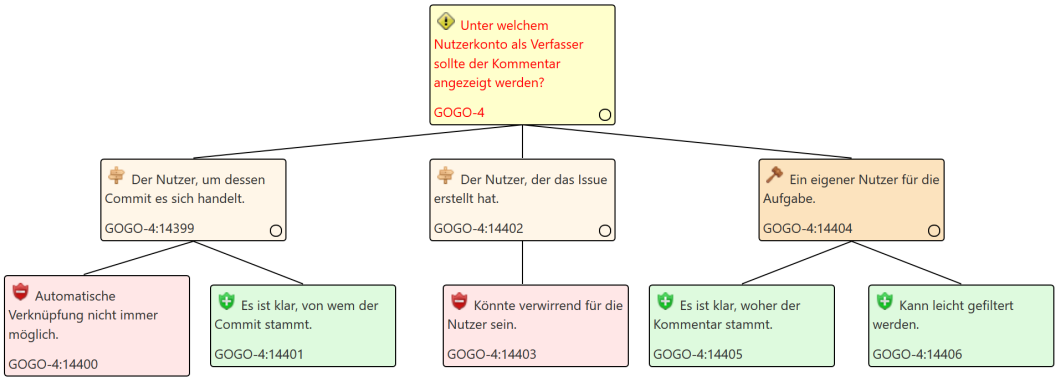

Abbildung 6.7: Entscheidung zur Problemstellung welcher Nutzer die Git-Commit-Nachricht posten soll.

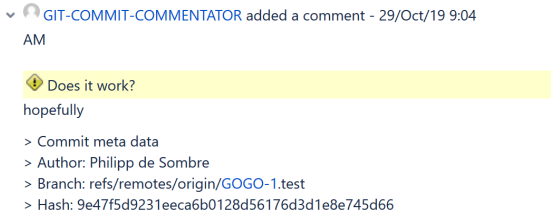

Abbildung 6.8: Bildschirmaufnahme eines gepostеten Git-Commits.

6.2.6 SF14: Klassifizierer mit neuem oder geändertem Entscheidungswissen aktualisieren

Der Klassifizierer für das Entscheidungswissen soll mit neuen Daten dazulernen (siehe On-line und Batch Learning). Hierbei tritt die Frage *Welche Daten sollen zum Dazu-*

lernen genutzt werden? auf.

Die Daten müssen aus korrekten Paaren von Sätzen und den Klassen, *ist relevant* und *Entscheidungswissenstyp*, bestehen. Mit automatisch klassifizierten Sätzen weiterzuarbeiten ist nicht zielführend, da dadurch auch falsch klassifizierte Sätze als neue Trainingsdaten genutzte werden. Deshalb werden nur manuell von Nutzern klassifizierte Sätze zum weiterlernen genutzt. Die Systemfunktion wird aufgerufen, sobald eine manuelle Klassifizierung stattfindet.

6.2.7 SF15: Güte des Klassifizierers berechnen

Diese Systemfunktion ist aus WS1.1 aufrufbar. Der Aufruf der Evaluation erfolgt über einen Knopf, dargestellt in Abbildung 6.10, der eine asynchrone Anfrage an das Jira-Plugin durchführt. Mithilfe aller im Projekt verifizierten Entscheidungswissenselementen werden die Metriken berechnet. Die berechneten Metriken sind: Korrektklassifikationsrate, Genauigkeit, Trefferquote und F1-Wert (siehe Abschnitt 2.3.1). Diese werden sowohl für die binäre, als auch für die feingranulare Klassifikation genutzt. Diese werden als JSON-Datei zum Download zur Verfügung gestellt. Die formatierten Ergebnisse erlauben eine einfache weitere Nutzung der Daten, beispielsweise zu Analysezwecken. Das Entscheidungsproblem ist in Abbildung 6.9 dargestellt.

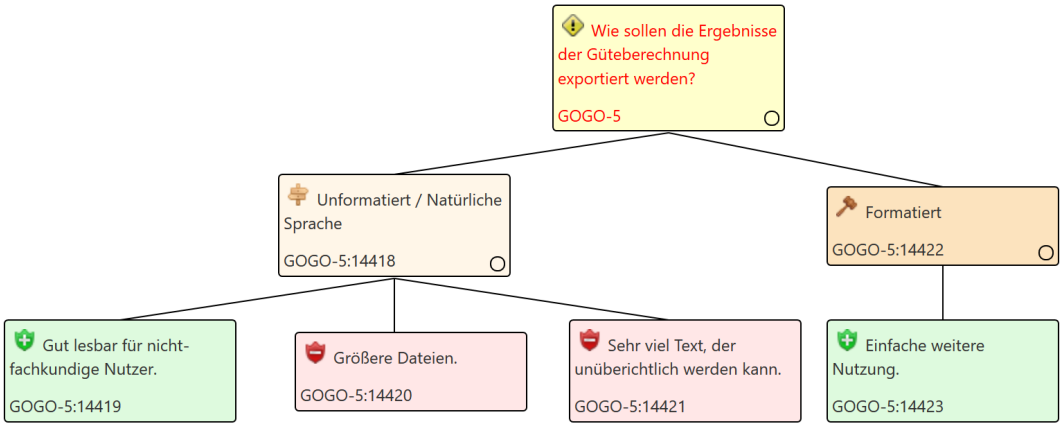

Abbildung 6.9: Entscheidung zur Problemstellung wie die Güte des Klassifizieres exportiert werden soll.

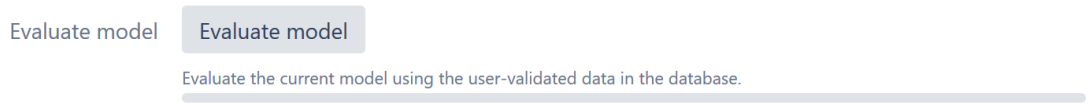

Abbildung 6.10: Bildschirmaufnahme der Option zur Evaluation des aktuell genutzten Modells.

6.3 Ergebnisse der Umsetzung

Die Architektur der implementierten Software wird in den zwei Abbildungen 6.11 und 6.12 dargestellt.

Abbildung 6.11 stellt den implementierten Klassifizierer und die damit in Verbindung stehenden Klassen dar.

Das Herzstück stellt die Klasse **DecisionKnowledgeClassifierImpl** dar, die das Interface **DecisionKnowledgeClassifier** implementiert. Diese Klasse enthält die Logik zur Klassifizierung des Entscheidungswissens. Hierbei wurde die zweistufige Architektur aus [1] übernommen. Die einzelnen Klassifizierer sind in den Klassen **BinaryClassifierImplementation** und **FineGrainedClassifierImpl** implementiert. Diese beiden Klassen erben von der abstrakten Klasse **Classifier**. Die Klassifizierer sind jeweils als SVMs umgesetzt (siehe Kapitel 5). Neben den Klassifizierern besitzt die Klasse **DecisionKnowledgeClassifierImpl** noch ein weiteres Attribut. Dabei handelt es sich um einen **Preprocessor**. Die Klasse **DecisionKnowledgeClassifierImpl** wurde nach dem Singleton-Entwurfsmuster umgesetzt. Diese Entscheidung wurde getroffen, da projektweit immer der gleiche Klassifizierer zum Einsatz kommen soll.

In Abbildung 6.12 ist die Klasse **CommitMessageToCommentTranscriber** dargestellt. Diese sorgt für die Integration der Git-Commit-Nachrichten in die JIRA-Issue-Kommentare. Die Klasse `ViewRest` enthält die Logik der REST-Schnittstelle. Von hier aus wird der **CommitMessageToCommentTranscriber** aufgerufen. Dieser Teil der Umsetzung fällt klein aus, da zur Darstellung die bereits vorhandener Logik für JIRA-Issue-Kommentare genutzt wird.

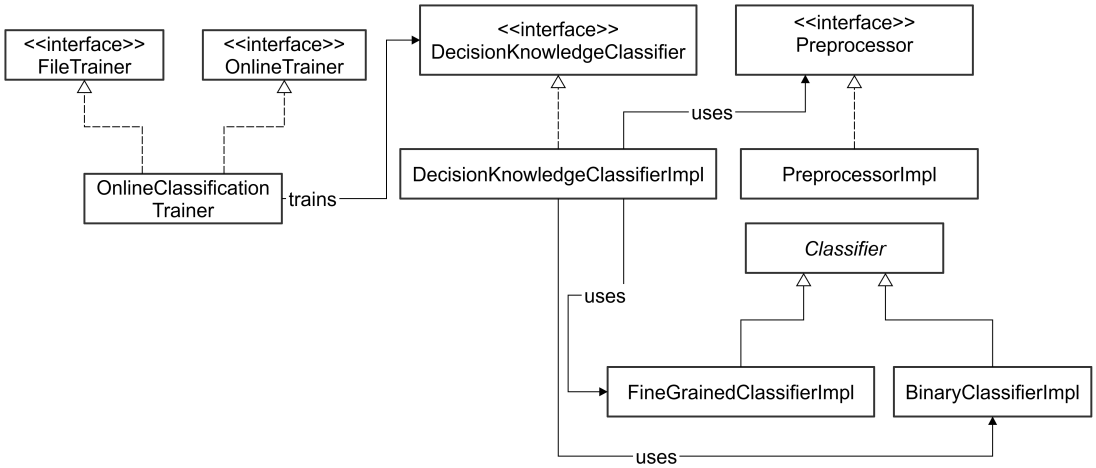

Abbildung 6.11: Vereinfachtes Klassendiagramm des Klassifizierers und der damit in Zusammenhang stehenden Klassen und Interfaces.

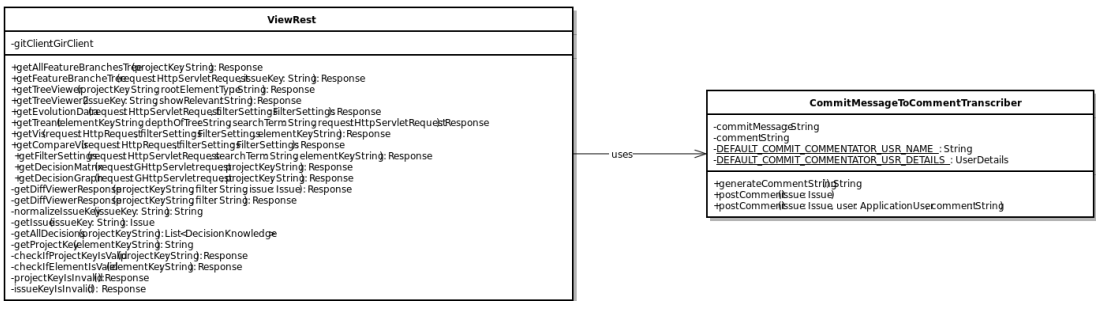

Abbildung 6.12: Klassendiagramm des CommitMessageToCommentTranscribers

7 Qualitätssicherung

In diesem Kapitel wird die Planung, Durchführung, sowie die Auswertung von Testfällen beschrieben. Falls dadurch Fehler gefunden werden, wird deren Lösung erläutert. Der Abschnitt *Planung der qualitätssichernden Maßnahmen* beschreibt, wie die Maßnahmen aufgebaut sind, um die Qualität möglichst vollständig sicherzustellen. Der darauffolgenden Abschnitte 7.2 und 7.3 beschreiben jeweils die Ergebnisse der Testdurchführung.

7.1 Planung der qualitätssichernden Maßnahmen

Die Sicherungsmaßnahmen werden für jede Systemfunktion in zwei Stufen durchgeführt: Komponententests und Systemtests. Diese Struktur der Tests in Stufen orientiert sich an Andreas Spillner [3].

Die Komponententests werden mir JUnit implementiert. Sie dienen zur Überprüfung einzelner Methoden. Die Tests werden sowohl lokal in der Intellij Entwicklungsumgebung und mit dem Befehl `atlas-package` der Jira SDK[1] ausgeführt. Zusätzlich werden die Tests automatisch auf einem Continuous-Integration-Server von TravisCI [2] ausgeführt. Die Komponententestfälle wurden mit dem Ziel entworfen eine möglichst hohe Codeabdeckung zu erreichen.

Systemtests überprüfen das Softwaresystem als Ganzes. Hierbei werden über die Benutzeroberfläche Testfälle durchgeführt. Dabei werden die Interaktionen der einzelnen Komponenten geprüft. Da die entwickelte Software ein Teil des ConDec-Plugins ist, wird auch die Integration in die gesamte Software implizit geprüft. Diese Tests werden auf einer externen Jira Instanz [3] getestet, um eine möglichst praxis-nahe Umgebung zu emulieren. Systemtests kommen hier nur dann zum Einsatz, wenn Komponententests nicht geeignet sind.

[1] Jira SDK: https://developer.atlassian.com/ Zuletzt aufgerufen am 24.10.2019
[2] TravisCI: https://travis-ci.org/ (Zuletzt abgerufen am: 24.10.2019)
[3] https://cures.ifi.uni-heidelberg.de/jira/

7.2 Tests funktionaler Anforderungen

Im folgenden Abschnitt werden die Tests für alle Systemfunktionen erläutert, die nicht bereits vollständig in [6] umgesetzt und getestet wurden. Alle beschriebenen Tests sind zum Projektabschluss fehlerfrei durchgeführt worden.

SF2: Integration von Git-Commit-Nachricht aktivieren Um diese Systemfunktion zu testen, kommen lediglich Systemtests und keine Komponententests zum Einsatz. Beim Aktivieren und Deaktivieren wird nur ein boolescher Wert in der Datenbank gespeichert, was einen Test, der rein auf der Serverseite stattfindet, überflüssig macht.

Tabelle 7.1: Systemtests für SF2: Integration von Git-Commit-Nachricht aktivieren

Id	Vorbedingungen	Schritte	Erwartetes Ergebnis
ST1	Extract from Git aktiviert, Integration deaktiviert	• Integration aktivieren	Integration aktiviert
ST2	Extract from Git aktiviert, Integration aktiviert	• Integration deaktivieren	Integration deaktiviert
ST3	Extract from Git aktiviert, Integration deaktiviert	• Extract from Git deaktivieren	Integration deaktiviert und gesperrt
ST4	Extract from Git aktiviert, Integration aktiviert	• Extract from Git deaktivieren	Integration deaktiviert und gesperrt

SF3: Entscheidungswissen in JIRA-Issue-Kommentaren automatisch klassifizieren Für Systemfunktion SF3 (SF3: Entscheidungswissen in JIRA-Issue-Kommentaren automatisch klassifizieren) wurden in [6] bereits Komponententests formuliert und durchgeführt. Eine Wiederholung dieser Tests ist notwendig, da der Klassifizierer im Rahmen dieser Arbeit stark modifiziert wurde.

SF4: Entscheidungswissen in Git-Commit-Nachrichten automatisch klassifizieren Um diese Systemfunktion zu testen, müssen sowohl die Git-Commit-Nachricht Integration als auch der Klassifizierer korrekt arbeiten. Die Software-Komponente, die die eigentliche Klassifizierung durchführt, ist dieselbe wie bei Systemfunktion SF3. Deswegen werden nur Systemtests durchgeführt, die die Integration von Git-Commit-Nachricht mit dem Klassifizierer testen.

Tabelle 7.2: Komponententests für SF3: Entscheidungswissen in JIRA-Issue-Kommentaren automatisch klassifizieren

Id	Testobjekt	Testdaten	Erwartetes Ergebnis
UT1	Binäre Klassifizierung	gefüllte Liste	klassifizierte Liste
UT2	Binäre Klassifizierung	klassifizierte Liste	unveränderte Liste
UT3	Binäre Klassifizierung	nicht klassifiziertes Element	klassifiziertes und unvalidiertes Element
UT4	Feinkörnige Klassifizierung	gefüllte Liste	klassifizierte Liste
UT5	Feinkörnige Klassifizierung	klassifizierte Liste	unveränderte Liste
UT6	Feinkörnige Klassifizierung	nicht klassifiziertes Element	klassifiziertes und unvalidiertes Element

Tabelle 7.3: Systemtests für SF4: Entscheidungswissen in Git-Commit-Nachrichten automatisch klassifizieren

Id	Vorbedingungen	Schritte	Erwartetes Ergebnis
ST5	Git-Commit-Nachricht Integration ist aktiviert, Branch für Issue existiert, Klassifizierer ist aktiviert	• Git-Commit auf Branch für Issue einchecken (Nachricht enthält kein Entscheidungswissen)	Git-Commit-Nachricht wird in JIRA-Issue-Kommentar angezeigt
ST6	Git-Commit-Nachricht Integration ist aktiviert, Branch für Issue existiert, Klassifizierer ist aktiviert	• Git-Commit auf Branch für Issue einchecken (Nachricht enthält Entscheidungswissen)	Git-Commit-Nachricht wird mit markiertem Entscheidungswissen in JIRA-Issue-Kommentar angezeigt
ST7	Git-Commit-Nachricht Integration ist deaktiviert, Branch für Issue existiert	• Git-Commit auf Branch für Issue einchecken	Git-Commit-Nachricht wird nicht angezeigt

SF6: Entscheidungswissen in Git-Commit-Nachrichten manuell klassifizieren EntwicklerInnen sollen manuell Entscheidungswissen in ihren Git-Commit-Nachricht annotieren können. Die Komponententests prüfen, ob die Anntationen in das korrekte Format zur Anzeige und Erkennung von Entscheidungswissen transkribiert werden.

SF8: Entscheidungswissen aus Git-Commit-Nachrichten anzeigen Das Anzeigen des Entscheidungswissens aus Git-Commit-Nachricht wird über die Kommentarfunktion von JIRA-Issue-Kommentar umgesetzt. Das Transkribieren für die korrekte Anzeige wird in den Komponententestfällen für Systemfunktion SF6: Entscheidungswissen in

Tabelle 7.4: Komponententests für SF6: Entscheidungswissen in Git-Commit-Nachrichten manuell klassifizieren

Id	Testobjekt	Testdaten	Erwartetes Ergebnis
UT7	Git-Commit-Nachricht Transkribierer	Leere Nachricht	leeres Element
UT8	Git-Commit-Nachricht Transkribierer	Nachricht mit kleingeschriebener Annotation	Element mit transkribierter kleingeschriebener Annotation
UT9	Git-Commit-Nachricht Transkribierer	Nachricht mit großgeschrieben Annotation	Element mit transkribierter kleingeschriebener Annotation
UT10	Git-Commit-Nachricht Transkribierer	Nachricht mit klein- und großgeschriebener Annotation	Element mit transkribierter kleingeschriebener Annotation
UT11	Git-Commit-Nachricht Transkribierer	Liste von Nachrichten mit kleingeschriebener Annotation	Elemente mit transkribierter kleingeschriebener Annotation als Kommentare in der Datenbank vorhanden

Git-Commit-Nachrichten manuell klassifizieren getestet. Die Darstellung des Entscheidungswissen aus Git-Commit-Nachricht wird bei den Systemtests für Systemfunktion SF4: Entscheidungswissen in Git-Commit-Nachrichten automatisch klassifizieren mit getestet, da hierbei auch überprüft wird, ob dieses angezeigt wird.

SF14: Klassifizierer mit neuem oder geändertem Entscheidungswissen aktualisieren

Diese Systemfunktion kann schwer mithilfe von Systemtests getestet werden, da die Verbesserung eines Klassifizierers nicht immer direkt sichtbar ist. Deshalb wird diese Funktion nur mit Komponententests überprüft.

Tabelle 7.5: Komponententests für SF14: Klassifizierer mit neuem oder geändertem Entscheidungswissen aktualisieren

Id	Testobjekt	Testdaten	Erwartetes Ergebnis
UT12	Klassifizierer	validiertes mit Entscheidungswissen annotiertes Element	fehlerfreie Durchführung
UT13	Klassifizierer	nicht validiertes mit Entscheidungswissen annotiertes Element	Durchführung wird übersprungen

SF15: Güte des Klassifizierers berechnen Bei den Testfällen für diese Systemfunktion wird lediglich die korrekt Durchführung, nicht aber die Güte des Klassifizierers an sich geprüft. Diese wurde mit den Experimenten aus Kapitel 5 geprüft.

Tabelle 7.6: Systemtests für SF15: Güte des Klassifizierers berechnen

Id	Vorbedingungen	Schritte	Erwartetes Ergebnis
ST8	Klassifizierer ist aktiviert	• Aufruf von *Evaluate Classifier* in den Einstellungen	JSON-Datei mit berechneten Metriken
ST9	Klassifizierer ist deaktiviert	• Aufruf des Reiters zur Text-Klassifizierung in den Einstellungen	Taste *Evaluate Classifier* ist deaktiviert

Tabelle 7.7: Komponententests für SF15: Güte des Klassifizierers berechnen

Id	Testobjekt	Testdaten	Erwartetes Ergebnis
UT14	Klassifizierer	List von validiertem Entscheidungswissen	fehlerfreie Durchführung
UT15	Klassifizierer	leere List	fehlerfreie Durchführung mit jeweils dem Wert 0 für die Metriken

Durch die umfassenden Komponententestfälle wurde in den neu implementierten Klassen der automatischen Klassifizierung eine durchschnittliche Testabdeckung von 86% der Codezeilen erreicht. Bei der Implementierung der Integration der Git-Commit-Nachrichten beträgt die durchschnittliche Testabdeckung der Codezeilen 85%.

7.3 Tests nicht-funktionaler Anforderungen

Die Einhaltung der nicht-funktionalen Anforderungen wird mithilfe der in den Tabellen aus Abschnitt 4.4 definierten Messungen geprüft. Die Messergebnisse werden mit den jeweiligen Zielgrößen abgeglichen.

NFR1: Funktionale Vollständigkeit Diese nicht-funktionale Anforderung wurde eingehalten. Alle Systemfunktionen, die in Abschnitt 4.3 beschrieben wurden sind vollständig implementiert worden.

NFR2: Funktionale Korrektheit Alle Komponenten- und Systemtests wurden erfolgreich durchgeführt. Diese Anforderung wurde somit vollständig umgesetzt.

NFR3: Zeitverhalten Der Klassifizierer wurde 100.000 Mal aufgerufen und dabei wurde die Gesamtzeit zur Berechnung der Klassen gemessen. Die binäre Klassifizierung benötigte insgesamt 11829ms, der fein-körnige Klassifizierer 15429ms. Insgesamt benötigt der Klassifizierer demzufolge ungefähr 0.273ms. Dies liegt deutlich unter dem Grenzwert von 100ms. Diese Anforderung wurde also erfolgreich umgesetzt.

NFR4: Interoperabilität Die Zielgröße dieser Anforderung war, dass die Git-Commit-Nachrichten-Integration mit weniger als 5 Aktionen durchgeführt werden kann, damit es nicht zu Frustrationen bei der jeweiligen NutzerIn kommt. Zur Aktivierung werden nach Login folgende Schritte benötigt:

1. Auswahl des Projektes
2. Auswahl der Projekteinstellungen
3. Auswahl des Reiters *Git connections*
4. Aktivieren der gewünschten Integration

. Diese Anforderung somit erfolgreich umgesetzt.

NFR5: Ästhetik der Benutzeroberfläche Für die Prüfung wird das neuste JIRA-Issue vom Typ *Issue* geprüft. Dieses enthält farblich und durch Symbole gekennzeichnetes Entscheidungswissen. Kein Entscheidungswissen-Element ist nicht gekennzeichnet. Diese Anforderung ist somit erfüllt.

NFR6: Fehlertoleranz Bei den Komponententestfällen wurde auch Tests mir nicht erwarteten Daten implementiert, zum Beispiel UT7. Die Zielgröße ist eine bessere Testabdeckung für zu erreichen, als bisher. Die Testabdeckung wird beim Einchecken neuer Git-Commit automatisch von Codecov[4] geprüft. Die Prüfung ergab vor dem Zusammenführen der einzelnen Entwicklungszweige mit dem Hauptentwicklungszweig eine ausreichend gute Abdeckung. Wobei bei dem neue hinzugefügten Quellcode ein Abdeckungsrate von 87,75% erreicht wurde. Diese Anforderung wurde vollständig erfüllt.

[4] https://codecov.io/ (Zuletzt abgerufen am: 21.11.2019)

NFR7: Statische Codequalität Weitere Metriken, neben der Testabdeckung, werden ebenfalls von Codecov automatisch geprüft. Diese Prüfungen wurden alle erfolgreich bestanden. Somit ist NFR7 erfüllt.

8 Evaluation

In diesem Kapitel wird eine Evaluation der implementierten Software vorgestellt und analysiert. Ziel der Evaluation ist festzustellen, inwieweit die implementierte Software als Lösung geeignet ist. Um dies zu erreichen wird ein Fragebogen nach dem Technology Acceptance Model (TAM) erstellt. Wie auch in den vorausgegangen Kapiteln wird nur auf noch nicht oder nur teilweise umgesetzte Funktionen eingegangen. Die bereits umgesetzten Funktionen wurden in [6] bereits evaluiert.

8.1 Vorstellung des Fragebogens

TAM definiert zwei wesentliche Faktoren, die Akzeptanz von Nutzern in Bezug auf Software bestimmen: *perceived usefulness* und *perceived ease of use*. Die *perceived usefulness* ist definiert, als das Subjektive empfinden darüber wie Wahrscheinlich es ist, dass eine Software die Produktivität erhöhen kann. Die *perceived ease of use* beschreibt hingegen, was Nutzer darüber denken, wie einfach die Benutzung der Software ist. [8]

Der Fragebogen ist in drei Aspekte unterteilt. Jede Kategorie entspricht aus Nutzersicht einem Aspekt. Diese Aspekte werden definiert um Systemfunktionen, zwischen denen Nutzer nicht eindeutig unterscheiden können, zusammen abzufragen. Für jeden Aspekt sind die damit in Zusammenhang stehenden Systemfunktionen in der Spalte SF genannt. Für jede Kategorie wird je eine Frage nach TAM gestellt.

Mit Frage EF1 wird evaluiert, wie einfach die Nutzung der manuellen Klassifikation von Git-Commit-Nachrichten ist. Frage EF2 soll evaluieren wie viel Mehrwert darin gesehen wird, dass Codeänderungen und Entscheidungswissen zusammen erfasst werden können.

Frage EF3 soll überprüfen, ob die Nutzung der automatischen Klassifikation von Git-Commit-Nachrichten einfach zu nutzen ist. Mit EF4 soll der wahrgenommene Mehrwert aus Nutzersicht überprüft werden.

Schließlich wird mit den Fragen EF5 und EF6 der Aspekt der Präsentation überprüft.

Tabelle 8.1: Kategorien des Fragebogens zur Feststellung der Eignung der implementierten Software.

Kat.	Aspekt	SF	Aufgabe
K1	Manuelle Klassifikation	SF6	Durchführung der manuellen Klassifikation von Entscheidungswissen in Git-Commit-Nachrichten.
K2	Automatische Klassifikation	SF3, SF4, SF15, SF16	Durchführung und Bewertung der automatischen Klassifikation von Entscheidungswissen in Git-Commit-Nachrichten.
K3	Präsentation	SF2, SF8	Gemeinsame Diskussion und Lösungsfindung eines Entscheidungsproblems.

Hier soll festgestellt werden, ob und wie gut diese die Kollaboration in den Teams unterstützt.

Jede Aussage soll von Teilnehmern auf einer Skala von eins (Ich stimme gar nicht zu.) bis fünf (Ich stimme voll und ganz zu.) bewertet werden. Zusätzlich kann zu jeder Aussage ein Freitext als Begründung ausgefüllt werden.

Tabelle 8.2: Fragebogen zur Eignung der implementierten Software nach TAM [8].

Nr.	Kat.	Variable	Frage
EF1	K1	perceived ease of use	Es ist einfach Entscheidungswissen in Git-Commit-Nachrichten manuell zu dokumentieren.
EF2	K1	perceived usefulness	Es ist nützlich Git-Commit-Nachrichten manuell zu klassifizieren, um diese zeitnah und mit den in Zusammenhang stehenden Codeänderungen zu dokumentieren.
EF3	K2	perceived ease of use	Es ist einfach Entscheidungswissen in Git-Commit-Nachrichten automatisch klassifizieren zu lassen.
EF4	K2	perceived usefulness	Es ist nützlich Git-Commit-Nachrichten automatisch klassifizieren zu lassen, um Entscheidungen besser reflektieren und Entscheidungswissen verständlicher dokumentieren zu können.
EF5	K3	perceived ease of use	Es ist einfach Git-Commit-Nachricht zur Entscheidungsdokumentation im Team zu nutzen.
EF6	K3	perceived usefulness	Es nutzt der kollaborativen Entscheidungsfindung Entscheidungswissen aus Git-Commit-Nachrichten in Jira einsehen zu können.

8.2 Ergebnisse der Umfrage

Die Teilnahme an der Umfrage war sehr gering. Lediglich drei Fragebögen wurden ausgefüllt[1]. Eine aussagekräftige statistische Analyse und Auswertung ist somit nicht möglich. Deswegen wird im Folgenden der Fokus auf die Freitext-Antworten gelegt.

EF1 Diese Aussage wurde mit 2, 4 und 5 bewertet. Die Dokumentation wurde als „*leicht und intuitiv*“ bezeichnet. Als sehr gut wurde im Freitext auch bewertet, „*dass die Dokumentation ohne Kontextwechsel und zeitnah zu den Codeänderungen erfolgen kann*“. Als negativ wurde angemerkt, dass keine Syntaxunterstützung bei der manuellen Eingabe vorhanden ist.

EF2 Die Aussage EF2 wurde drei Male mit 5 bewertet. Als Mehrwert wurde von den Teilnehmern gesehen, dass dem Wissensverlust entgegenwirkt wird. EinE TeilnehmerIn merkte jedoch an, dass es zwar „*nützlich, aber aufwändig*“ sei.

EF3 Diese Aussage wurde einmal mit 4 und zweimal mit 5 bewertet. Die Bedienung wurde als sehr einfach eingestuft, da „*die Commit-Nachrichten automatisch importiert und klassifiziert*“ werden. Zusätzlich wurde noch erwähnt, dass eine „*Änderung [...] auch sehr einfach möglich*“ ist.

EF4 Aussage EF4 wurde ebenso wie EF3 einmal mit 4 und zweimal mit 5 bewertet. In der Freitext-Antwort wurde betont, dass die Entscheidungen nun eher von EntwicklerInnen reflektiert werden. Zusätzlich wurde angemerkt, dass „*[s]elbst wenn der Klassifikator falsch liegt, werden EntwicklerInnen angeregt die Dokumentation zu überarbeiten.*“.

EF5 EF5 wurde von den Teilnehmern der Umfrage jeweils mit 4 bewertet. In den Antworten der Teilnehmer wurde angemerkt, dass ohne die Integration in Jira Git-Commit-Nachrichten „*an sich kein guter Ort, um kollaborativ Entscheidungen zu treffen*“, da sie „*von einzelnen EntwicklerInnen geschrieben*“ werden. Zusätzlich wurde einer weiteren TeilnehmerIn angemerkt, dass Git-Commit-Nachrichten zur Entscheidungsdokumentation einfach zu nutzen sind, dies aber „*unabhängig vom Team*“ ist.

[1] Stand 18.12.2019

EF6 Aussage EF6 wurde jeweils einmal mit 3, 4 und 5 bewertet. Als nützlich wurde die grafische Visualisierung bezeichnet, da sie nicht nur die Git-Commit-Nachricht untereinander auflistet.

8.3 Fazit

Die bisherigen Teilnehmer der Umfrage haben beide sehr positive Rückmeldungen gegeben. Die implementierten Funktionen scheinen demzufolge gut geeignete Lösungen zu sein. Eine aussagekräftige Analyse ist jedoch aufgrund der wenigen Teilnehmer nicht möglich. Die Umfrage sollte erneut, beziehungsweise mit mehr Teilnehmern durchgeführt werden.

9 Schlussfolgerung

9.1 Zusammenfassung

In diesem Bericht wurden die drei Forschungsfragen FF1: *In welchen Artefakten ist implizites Entscheidungswissen enthalten?*, FF2: *Welche Typen von Entscheidungswissen können unterschieden werden?* und FF3 *Mit welchen Methoden ist es möglich implizites Entscheidungswissen automatischnaus den Artefakten zu extrahieren und den Entscheidungswissen-Typen zuzuordnen?* behandelt. Durch eine Literaturrecherche (Kapitel 3 und 2) und eigenständige Versuche (Kapitel 5) diese schließlich beantwortet. Die Antworten der Forschungsfragen wurden anschließend genutzt, um die Anforderungen (Kapitel 4) umzusetzen. Alle Anforderungen wurden umgesetzt, wobei Entscheidungen bei der Umsetzung und die Ergebnisse explizit dargestellt wurden. Die umgesetzten Funktionen wurden anschließend verifiziert (Kapitel 7) und validiert (Kapitel 8). Dadurch konnte gezeigt werden, dass die umgesetzten Funktionen sich korrekt ausführen lassen und dass diese auch die Probleme bei der Entscheidungsdokumentation sinnvoll lösen.

9.2 Ausblick

Die Klassifizierer zeigen zwar ziemlich gute Ergebnisse auf fremden Daten, da es sich um dazuzulernende Algorithmen handelt, jedoch wurden bei weitem nicht alle Algorithmen getestet. Neuronale Netze können beispielsweise dazulernen, wurden aber in den Experimenten in diesem Bericht nicht beachtet.

In Kapitel 5 wurde die geringe Anzahl an verifizierten irrelevanten Entscheidungswissen-Elementen erwähnt. Diese ist darauf zurückzuführen, dass die Werkzeug-Unterstützung zur Markierung von irrelevanten Elementen nicht einfach zu Bedienen ist. Das entsprechende Element muss zuerst als relevant markiert werden, um danach als irrelevant markiert zu werden, damit es als verifiziertes irrelevantes Element gilt. Dieser Aspekt könnte verbessert werden.

Des Weiteren können noch weitere Quellen für Entscheidungswissen angebunden werden. In Kapitel 2.1 im Abschnitt *Orte für Entscheidungen* sind eine Reihe weiterer Wissensquellen aus der Literatur aufgezählt.

Daneben könnte die Anbindung an Git verbessert werden. Die Zuordnung von JIRA-Issue zu einem Entwicklungszweig erfolgt über String-Matching. Dies könnte beispielsweise über eine manuelle Zuordnung erweitern werden.

10 Literaturverzeichnis

[1] Alkadhi, Rana Mohammed A. *Rationale in Developers' Communication.* Dissertation, Technische Universität München, München, 2018.

[2] Alkadhi, Rana und Laţa, Teodora und Guzman, Emitza und Bruegge, Bernd. Rationale in Development Chat Messages: An Exploratory Study. In *Proceedings of the 14th International Conference on Mining Software Repositories*, MSR '17, pages 436–446, Piscataway, NJ, USA, 2017. IEEE Press. ISBN 978-1-5386-1544-7. doi: {10.1109/MSR.2017.43}. URL https://doi.org/10.1109/MSR.2017.43.

[3] Tilo Linz Andreas Spillner. *Basiswissen Softwaretest: Aus- und Weiterbildung zum Certified Tester - Foundation Level nach ISTQB-Standard.* dpunkt.verlag, 5. auflage edition, 2012. ISBN 3864900247,9783864900242.

[4] Bhat, Manoj und Shumaiev, Klym und Biesdorf, Andreas und Hohenstein, Uwe und Matthes, Florian. Automatic Extraction of Design Decisions from Issue Management Systems: A Machine Learning Based Approach. In Lopes, Antónia und de Lemos, Rogério, editor, *Software Architecture*, pages 138–154, Cham, 2017. Springer International Publishing. ISBN 978-3-319-65831-5.

[5] Bruegge, Bernd und Dutoit, Allen H. *Object-Oriented Software Engineering Using UML, Patterns, und Java.* Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2009. ISBN 0136061257, 9780136061250.

[6] Clormann, Jochen. *DecXtract: Dokumentation und Nutzung von Entscheidungswissen in JIRA-Issue-Kommentaren.* PhD thesis, 02 2019.

[7] David Page. Lecture notes in CCS760 - Evaluating Machine Learning Methods, 2017. URL http://pages.cs.wisc.edu/~dpage/cs760/.

[8] Fred Davis, Richard Bagozzi, and Paul Warshaw. User acceptance of computer technology: A comparison of two theoretical models. *Management Science*, 35: 982–1003, 08 1989. doi: 10.1287/mnsc.35.8.982.

[9] Dutoit, Allen und McCall, Ray und Mistrík, Ivan und Paech, Barbara. *Rationale Management in Software Engineering* - . Springer Science & Business Media, Berlin Heidelberg, edition, 2007. ISBN 978-3-540-30998-7.

[10] Guillaume Lemaitre and Fernando Nogueira and Christos K. Aridas. Imbalanced-learn: A python toolbox to tackle the curse of imbalanced datasets in machine learning. *CoRR*, abs/1609.06570, 2016. URL http://arxiv.org/abs/1609.06570.

[11] Gurusamy, Vairaprakash und Kannan, Subbu. Preprocessing Techniques for Text Mining. page , 10 2014.

[12] Hesse, Tom-Michael und Kuehlwein, Arthur und Paech, Barbara und Roehm, Tobias und Bruegge, Bernd. Documenting Implementation Decisions with Code Annotations. page , 07 2015. doi: {10.18293/SEKE2015-084}.

[13] Hesse, Tom-Michael und Paech, Barbara. Supporting the collaborative development of requirements and architecture documentation. pages 22–26, 07 2013. doi: {10.1109/TwinPeaks-2.2013.6617355}.

[14] Chih-wei Hsu, Chih-chung Chang, and Chih-Jen Lin. A practical guide to support vector classification chih-wei hsu, chih-chung chang, and chih-jen lin. 11 2003.

[15] Joachims, Thorsten. Text categorization with Support Vector Machines: Learning with many relevant features. In Nédellec, Claire und Rouveirol, Céline, editor, *Machine Learning: ECML-98*, pages 137–142, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg. ISBN 978-3-540-69781-7.

[16] Kleebaum, Anja und Johanssen, Jan Ole und Paech, Barbara und Alkadhi, Rana und Bruegge, Bernd. Decision Knowledge Triggers in Continuous Software Engineering. In *Proceedings of the 4th International Workshop on Rapid Continuous Software Engineering*, RCoSE '18, pages 23–26, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5745-6. doi: {10.1145/3194760.3194765}. URL http://doi.acm.org/10.1145/3194760.3194765.

[17] Kleebaum, Anja und Johanssen, Jan Ole und Paech, Barbara und Bruegge, Bernd. Tool Support for Decision und Usage Knowledge in Continuous Software Engineering. 2018. doi: {10.11588/heidok.00024186}. URL http://archiv.ub.uni-heidelberg.de/volltextserver/id/eprint/24186.

[18] Kruchten, Philippe. An Ontology of Architectural Design Decisions in Software-Intensive Systems. *2nd Groningen Workshop on Software Variability*, page 54–61,

01 2004.

[19] Lee, Jintae. Decision Representation Language (DRL) and Its Support Environment. page , 08 1989.

[20] Jochen Lichter, Horst; Ludewig. *Software Engineering: Grundlagen, Menschen, Prozesse, Techniken*. Dpunkt.Verlag GmbH, 3., korr. aufl edition, 2013. ISBN 3864900921,978-3-86490-092-1,978-3-86491-298-6,978-3-86491-299-3,610-12-1990-9.

[21] MacLean, Allan und M. Young, Richard und Bellotti, Victoria und P. Moran, Thomas. Questions, Options, and Criteria: Elements of Design Space Analysis. *Human-Computer Interaction*, 6:201–250, 09 1991. doi: {10.1080/07370024.1991.9667168}.

[22] Mehryar Mohri, Afshin Rostamizadeh, Ameet Talwalkar. *Foundations of Machine Learning*, volume of *Adaptive Computation and Machine Learning*. The MIT Press, edition, 2012. ISBN 026201825X, 978-0262018258.

[23] Bertrand Meyer. *Object-oriented software construction*. Prentice-Hall, New York, 1988. ISBN 0-13-629049-3.

[24] Robert B. Miller. Response time in man-computer conversational transactions. In *Proceedings of the December 9-11, 1968, Fall Joint Computer Conference, Part I*, AFIPS '68 (Fall, part I), pages 267–277, New York, NY, USA, 1968. ACM. doi: 10.1145/1476589.1476628. URL http://doi.acm.org/10.1145/1476589.1476628.

[25] Mylopoulos, John und Chung, Lawrence und Yu, Eric. From Object-oriented to Goal-oriented Requirements Analysis. *Commun. ACM*, 42(1):31–37, January 1999. ISSN 0001-0782. doi: {10.1145/291469.293165}. URL http://doi.acm.org/10.1145/291469.293165.

[26] N. V. Chawla and K. W. Bowyer and L. O. Hall and W. P. Kegelmeyer. SMOTE: Synthetic minority over-sampling technique. *Journal of Artificial Intelligence Research*, 16:321–357, 06 2002. doi: {10.1613/jair.953}. URL https://doi.org/10.1613/jair.953.

[27] Jeffrey Pennington, Richard Socher, and Christopher Manning. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 1532–1543, 2014.

[28] Klaus Pohl and Chris Rupp. *Basiswissen Requirements Engineering - Aus- und Weiterbildung nach IREB-Standard zum Certified Professional for Requirements Engineering Foundation Level.* dpunkt.verlag, Heidelberg, 4. aufl. edition, 2015. ISBN 978-3-864-91674-8.

[29] Prof. Dr. Barbara Paech. Vorlesungsfolien: Software Engineering, September 2018.

[30] Rogers, Benjamin und Gung, James und Qiao, Yechen und E. Burge, Janet. Exploring techniques for rationale extraction from existing documents. *Proceedings - International Conference on Software Engineering*, pages 1313–1316, 06 2012. doi: 10.1109/ICSE.2012.6227091.

[31] Rogers, Benjamin und Qiao, Yechen und Gung, James und Mathur, Tanmay und E. Burge, Janet. *Using Text Mining Techniques to Extract Rationale from Existing Documentation*, pages 457–474. 01 2015. ISBN 978-3-319-14955-4. doi: {10.1007/978-3-319-14956-1_26}.

[32] S. Vijayarani und M. Ilamathi und Ms. Nithya. Preprocessing Techniques for Text Mining-An Overview. volume Vol 5(1),7-16, 2015.

[33] Stuart Russell, Peter Norvig. *Artificial Intelligence: A Modern Approach*, volume of *Prentice Hall Series in Artificial Intelligence*. Prentice Hall, 3rd edition, 2010. ISBN 0136042597, 9780136042594.

[34] Tom-Michael Hesse und Veronika Lerche und Marcus Seiler und Konstantin Knoess und Barbara Paech. Documented decision-making strategies and decision knowledge in open source projects: An empirical study on Firefox issue reports. *Information and Software Technology*, 79:36 – 51, 2016. ISSN 0950-5849. doi: {https://doi.org/10.1016/j.infsof.2016.06.003}. URL http://www.sciencedirect.com/science/article/pii/S0950584916301148.

[35] Werner Kunz und Horst W. J. Rittel und We Messrs und H. Dehlinger und T. Mann und J. J. Protzen. Issues as elements of information systems. 1970.