

Proceedings des KI-Praktikums der Veranstaltung
Management Support Systems III
an der Universität Osnabrück

Bugtracker Information Mining

Wintersemester 2015/2016

Herausgeber

Prof. Dr.-Ing. Bodo Rieger

Axel Benjamins

Inhaltsverzeichnis

Geleitwort.....	4
------------------------	----------

Automatische Generierung von Zuständigkeitsvorschlägen zwischen neuen Fehlerberichten und deren Lösungsanbietern (Christian Benz, Simon Hagen, Oksana Tur).....	5
--	----------

1 Business Understanding	5
2 Data Understanding	8
3 Data Preparation	10
4 Modeling	14
5 Evaluation.....	15
6 Deployment	16

Prototypische Implementierung zur systemseitigen Früherkennung von Dubletten beim Anlegen neuer Einträge im Eclipse Bugtracker (Katrin Baalman, Ulrike Hinrichs, Stephanie Klatte)	19
---	-----------

1 Business Understanding	19
2 Data Understanding	21
3 Data Preparation	23
4 Modeling	26
4.1 Naive Bayes.....	26
4.2 k-NN	29
5 Evaluation.....	30
5.1 Naive Bayes.....	30
5.2 k-NN	33
6 Deployment	35
7 Literatur.....	36

Automatische Zuordnung von Bugs zu Entwicklern mit Hilfe von Text Mining (Daniel Bender, Lukas Brenning, Henrik Kortum)	37
--	-----------

1 Business Understanding	37
2 Data Understanding	39

3	Data Preparation	43
4	Modeling	47
5	Evaluation.....	49
	5.1 Evaluation der Ergebnisse	49
	5.2 Kritische Würdigung und Limitation	50
6	Deployment	51
7	Literatur	53

Konzeption und prototypische Implementierung einer Bug Klassifizierung zur Realisierung innerbetrieblicher Einsparungspotenziale auf Basis des Eclipse Bugtrackers (Jens Keuter, Ines Wojtczyk, Denis Jakupovic).....54

1	Business Understanding	54
2	Data Understanding.....	56
3	Data Preparation	56
4	Modeling	58
	4.1 Support Vector Machines	58
	4.2 Das Klassifikationsmodell in RapidMiner	60
5	Evaluation.....	63
6	Deployment	64
7	Literatur	65

Prognose der Bearbeitungsdauer von Bugs mit Hilfe von Clustering und Entscheidungsbäumen (Michael Körfer, Victor Brinkhege, Kai Steenblock)67

1	Business Understanding	67
2	Data Understanding.....	69
	2.1 Daten Beschreibung.....	69
	2.2 Daten Qualität.....	70
3	Data Preparation	72
4	Modeling	77
5	Evaluation.....	80
6	Deployment	81

Zusammenfassung und Ausblick82

Geleitwort

In den vorliegenden Proceedings wird die Anwendung von Text Mining auf die Daten eines Bugtracker zur Optimierung des Software-Entwicklungsprozesses fokussiert. Bugtracker dienen als Plattform für das Qualitätsmanagement und werden in allen größeren Projekten eingesetzt. Sie werden zur Erfassung von Bugs sowie zur Dokumentation des Software-Entwicklungsstands verwendet. Viele Informationen liegen dabei in Form von Texten vor. Diese Informationen können schwer maschinell ausgelesen und verarbeitet werden, tragen jedoch oftmals zu einer Verbesserung von Analysen bei. Das Text Mining stellt eine aufstrebende Forschungsdomäne dar, welche die automatisierte Extraktion und Verarbeitung von Informationen aus Texten in den Fokus stellt.

Die nachfolgenden Beiträge entstanden durch Studierende im Rahmen des KI-Praktikums der Veranstaltung „Management Support Systems III – Künstliche Intelligenz“ an der Universität Osnabrück zum Thema „Bugtracker Information Mining“. Als Datensatz wurden dabei die öffentlich zugänglichen Daten des Eclipse Bugtrackers gewählt. Im ersten Beitrag werden neuen Fehlerberichten durch Zuständigkeitsvorschläge potenziell passende Entwickler zugeordnet. Der zweite Beitrag ermöglicht die systemseitige Früherkennung von Dubletten beim Anlegen eines Fehlerberichts. Die automatische Zuordnung von Entwicklern zu Bugs steht ebenfalls im dritten Beitrag im Fokus. Der vierte Beitrag soll innerbetriebliche Einsparungspotenziale durch die Klassifizierung von Bugs ermöglichen. Im abschließenden fünften Beitrag wird die Bearbeitungsdauer von Bugs prognostiziert.

Allen Beiträgen liegt eine prototypische Implementierung zugrunde, welche grob das Potenzial der jeweiligen Idee aufzeigt. Die prototypischen Umsetzungen und die Dokumentationen orientieren sich am international anerkannten CRISP-DM. Die Implementierungen wurden mit der Software RapidMiner 6 durchgeführt. Für die Bereitstellung der Lizenzen danken wir der Firma RapidMiner GmbH.

Osnabrück, im September 2016

Prof. Dr.-Ing. Bodo Rieger
Axel Benjamins

Automatische Generierung von Zuständigkeitsvorschlägen zwischen neuen Fehlerberichten und deren Lösungsanbietern

Christian Benz, Simon Hagen, Oksana Tur

Abstract. *Das Open Source Projekt rund um die Entwicklungsumgebung Eclipse nutzt zur Verwaltung ihrer Fehlerberichte (Bugs) den webbasierten Bugtracker bugzilla. Um die Zuordnung neu eintreffender Bugs zu den entsprechenden Entwicklern zu vereinfachen, werden in dieser Arbeit die Rohdaten des Bugtrackers genutzt, um entsprechende Vorschläge zu generieren. Dies geschieht mit Hilfe von künstlichen neuronalen Netzen (KNN) und Text Mining. Um ein strukturiertes Vorgehen zu ermöglichen, wird der Cross Industry Standard for Data Mining (CRISP-DM) angewandt. Nach der Auswertung der Ergebnisse werden Anregungen für die Implementierung einer gezielteren Zuordnung gegeben.*

1 Business Understanding

Die 2004 gegründete Eclipse Foundation mit Sitz in Ottawa, Kanada ist für die Leitung und Koordinierung der zugehörigen Open Source Gemeinschaft und derer Projekte zuständig. Zu den bekanntesten und erfolgreichsten Projekten zählen unter anderen die gleichnamige Entwicklungsumgebung „Eclipse“ oder die Java Development Tools (JDT). Darüber hinaus gibt es über 150 weitere Projekte unterschiedlicher Größe. Zum Erhalt der Produktqualität bietet die Eclipse Foundation ihren Nutzern und den Entwicklern die Möglichkeit, Fehlerberichte (sog. Bugs) innerhalb eines webbasierten Bugtrackers zu verwalten, wobei neue Bugs hinzugefügt, der aktuelle Status verfolgt oder Kommentare verfasst werden können. Passive Teilnehmer können außerdem bereits gelöste Fehlermeldungen nutzen um ihre Fehler eigenständig zu beheben oder die Gründe weiter einzugrenzen.

Aktuell sind weniger als 8% der Bugs mit aussagekräftigen Tags versehen. Da dadurch das Auffinden von bereits gelösten Fehlern nur eingeschränkt möglich ist, werden viele Problemfälle mehrfach eingetragen. Außerdem haben die Entwickler oder andere etwaige Lösungsanbieter es schwer, die Bugs zu finden, die möglicherweise von Ihnen gelöst

werden können. Somit gewährleistet das System keine effiziente Zusammenarbeit von Entwicklern und Bugerstellern.

Um einen effizienteren Prozess bei der Wahl eines Bugs für den Entwickler zu ermöglichen, besteht die Option der Erstellung von Entwicklerprofilen durch neuronale Netze, die auf dieser Grundlage dem Entwickler neue Bugs vorschlagen. Hierfür können Zwischenziele formuliert werden. Zuerst sollen Benutzerprofile erstellt werden. Zu diesem Zweck sollten zu jedem Entwickler die von ihm bearbeiteten Bugs zusammengeführt werden. Auf dieser Grundlage von Daten sollte ein Text Mining durchgeführt werden, um daraus Profile abzuleiten, die aus den Wörtern und deren Häufigkeiten des Vorkommens interpretiert werden. Für die nötigen Schritte gewährleistet das Programm Rapid-Miner ein Fundament von verschiedenen Operatoren, wie Text Mining, aber auch Neuronale Netze, die für die Erstellung der Entwicklerprofile notwendig sind.

Im Rahmen dieser Arbeit sind zwei Ansätze entstanden, um die zuvor vorgestellten Ziele zu erreichen. Diese unterscheiden sich hauptsächlich in der Filterung der Wörter, die zur Bildung der Nutzerprofile herangezogen werden. Beim ersten Ansatz wird hier die „Term Frequency“-Methode verwendet, bei der die gesamte Anzahl der Vorkommen der Wörter betrachtet wird. Der zweite Ansatz nutzt die TF-IDF (Term Frequency - Inverse Document Frequency) Methode, bei der die Relevanz eines Wortes im Gesamtzusammenhang anhand der Häufigkeit des Vorkommens gemessen wird.

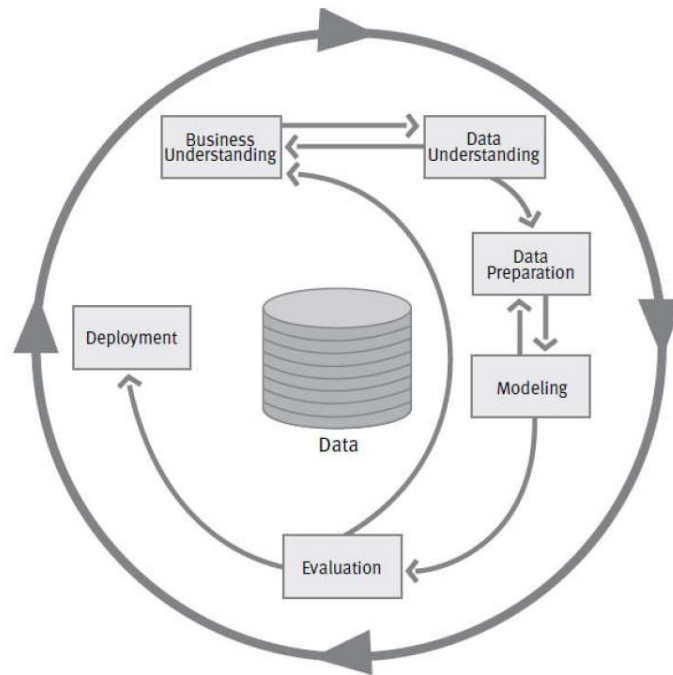


Abbildung 1: Schematische Darstellung des CRISP-DM

Das Vorgehen orientiert sich dabei am CRISP-DM (Cross Industry Standard for Data Mining), welches eine strukturierte Methode zur Durchführung von Data Mining-Projekten bietet. Der Projektzeitraum für die fünf Phasen erstreckt sich dabei auf ca. einen Monat, von Mitte Dezember 2015 bis Ende Januar 2016. Die nachfolgende **Fehler! Verweisquelle konnte nicht gefunden werden.** gibt einen Überblick über die grundlegenden Meilensteine im Projektverlauf.

Tabelle 1: Projektverlauf

#	Name	Ziele	Termin
1	Business Understanding	Zieldefinition und Projektplan	06.01.16
2	Data Understanding	Datenbeschreibung & -exploration	11.01.16
3	Data Preparation	Zusammenführen aller benötigter Daten in eine Tabelle zur schnelleren Abfrage	16.01.16
4	Modeling	Modell erstellen und beschreiben, Parameter festlegen	21.02.16
5	Evaluation / Deployment	Interpretation der Ergebnisse, Limitationen, möglicher späterer Einsatz	26.01.16

2 Data Understanding

Die Rohdaten des Bugtrackers wurden über eine MySQL-Schnittstelle zur Verfügung gestellt und können mit Hilfe des ReadDatabase-Connectors im RapidMiner Studio abgefragt werden. Da der bestehende Datenbankzugriff nur mit Lese-Berechtigungen möglich ist, wurde ein Abbild der Datenbank temporär in eine eigene Entwicklungsdatenbank geladen um vorbereitende Änderungen an den Daten vornehmen zu können und diese persistent speichern zu können.

Die Datenbank besteht aus 18 Tabellen, welche zum großen Teil durch entsprechende Constraints verbunden sind. Neben den eigentlichen Fehlermeldungen und deren Metadaten werden beispielsweise noch projektspezifische Daten zu sämtlichen Eclipse Produkten gespeichert. Dazu werden entsprechende Daten zu Versionsständen, Entwicklungserfolgen oder Komponenten der Produkte zugeordnet. Im folgenden relationalen Datenbankmodell (Abbildung 2) sind die Tabellendaten und deren Beziehungen zueinander dargestellt.

Die türkis hinterlegten Tabellen beinhalten dabei alle Daten, die für die Erfassung der verschiedenen Bugs verwendet werden. Die rot hinterlegten Tabellen enthalten hingegen Informationen zu den verschiedenen Produkten der Eclipse Foundation.

Die Tabellen am unteren Rand ohne farbige Hinterlegung gehören ebenfalls zu den Bugbezogenen Daten, werden jedoch für unsere Analyse benötigt und somit nicht gekennzeichnet.

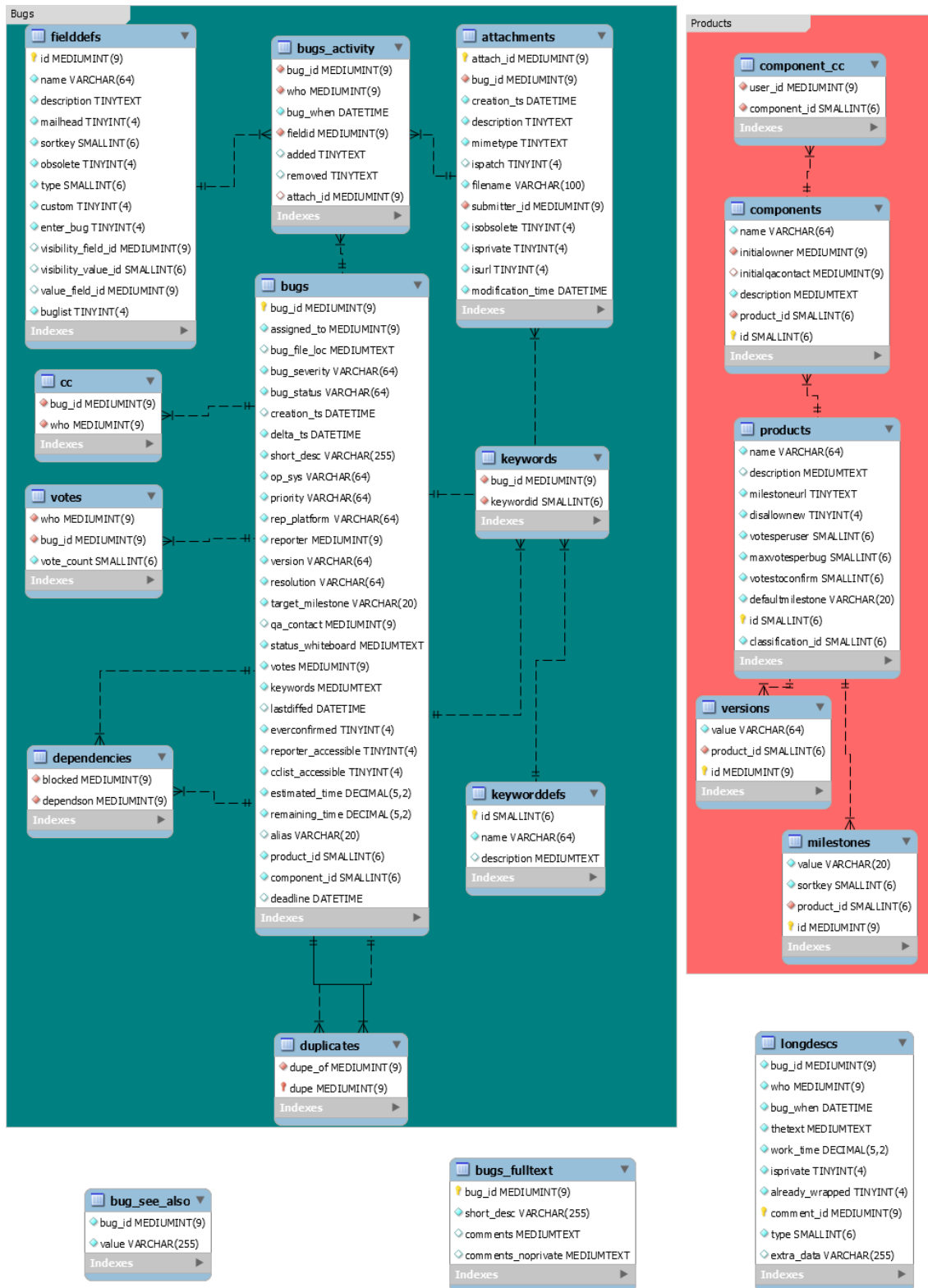


Abbildung 2: Entity-Relationship-Diagramm der Bugtracker-Datenbank

Die vorhandenen Daten waren zwar syntaktisch korrekt und logisch verknüpft, allerdings haben sich auf semantischer Ebene einige Probleme mit der Datenqualität ergeben. So

sind beispielsweise nur ca. 7% der gemeldeten Fehlerberichte mit entsprechenden Schlagwörtern versehen. Dies macht eine Kategorisierung der Fälle im Vorfeld nahezu unmöglich.

3 Data Preparation

Um eine Zuordnung von neuen Fehlermeldungen zu Entwicklern auf Basis von historischen Lösungen zu ermöglichen, kann die Datenbasis deutlich eingeschränkt werden. Die Auswahl wurde auf die Tabellen `bugtracker.bugs` und `bugtracker.longdescs` beschränkt, da nur Inhalte der Fehlerbeschreibungen und Informationen zum zugeordneten Entwickler relevant für die Erstellung der neuen Zuordnungen sind. Für ein entsprechendes Text Mining sind lediglich die Benutzer-ID (`bugtracker.longdescs.who`) sowie der vollständige Bugtext (`bugtracker.longdescs.thetext`) erforderlich. Um die Zuordnung nur auf Basis von erfolgreichen historischen Daten zu berechnen, wurden außerdem auch die Bugs, die nicht erfolgreich gelöst sind, ausgeschlossen.

Um die oben genannten Daten zu extrahieren wurden die Daten per SQL in eine neue, temporäre Tabelle geschrieben (SQL01). Dabei wurden der Text, die Benutzer-ID sowie das Datum des Textes übernommen. Im weiteren Schritt wurden sämtliche Texte, die denselben Autoren zugeordnet sind, verknüpft und wiederum in eine weitere temporäre Tabelle geschrieben (SQL02). Darüber hinaus wurden alle Großbuchstaben im gleichen SQL-Befehl (`lower()`) zu Kleinbuchstaben transformiert. Die so erstellte SQL-Tabelle wird dann als Grundlage für die weitere Verwendung in RapidMiner genutzt und abhängig vom Ansatz unterschiedlich aufbereitet.

SQL01:

```
INSERT INTO bugtracker.combined (  
    thetext,  
    authorid,  
    bugdate)  
(SELECT l.thetext as thetext, l.who as authorid, l.bug_when as  
bugdate  
FROM bugtracker.bugs b, bugtracker.longdescs l  
WHERE l.bug_id = b.bug_id  
AND b.assigned_to = l.who
```

SQL02:

```
INSERT INTO bugtracker.concatenated (  
    authorid,  
    alltext)  
(SELECT com.authorid, lower(group_concat(thetext SEPARATOR '  
'))) as alltext  
FROM bugtracker.combined com
```

Beim ersten Ansatz (Term Frequency) wird die ID des Benutzers und das verbundene Beschreibungsfeld aus der zuvor erstellten Tabelle `concatenated` geladen. Durch die insgesamt sehr große Datenmenge werden hier neben der oben genannten zeitlichen Selektion zusätzlich nur die ersten 20 Einträge verwendet, da der Export zu Excel und die Generierung des Neuronalen Netzes sonst zu zeitintensiv ist. Die Felder werden durch die Konvertierung in Polynomiale- und Text-Felder auf die weitere Verwendung in RapidMiner vorbereitet. Anschließend werden die verknüpften Beschreibungen mit Hilfe von Text Mining verarbeitet und in zwei Arten gespeichert: Zum einen wird in einem Repository in RapidMiner eine Tabelle mit allen Wörtern (Spalten) und Benutzern (Zeilen) und der jeweiligen Anzahl der Vorkommen in den Zellen gespeichert. Zum anderen wird die Liste mit allen gefundenen Wörtern in einer Excel-Tabelle abgelegt, da diese im nächsten Schritt manuell bearbeitet wird, was in RapidMiner nur mit großem Aufwand möglich wäre. In der Tabelle wird die Spalte „in document“ und alle Wörter mit weniger als 5

Vorkommen entfernt sowie ein Eintrag „authorid“ hinzugefügt (Schritt 2) (Abbildung 3). Diese Reduzierung der Daten erfolgt, um die durch selten vorkommende Wörter erzeugte Streuung des Ergebnisses zu verringern und die allgemeine Bearbeitungszeit zu verbessern. Die Spalte „in document“ wird für die weitere Verwendung schlicht nicht benötigt.

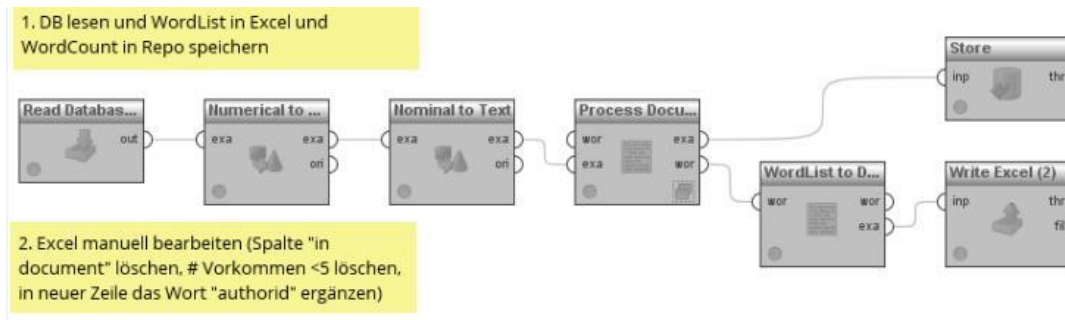


Abbildung 3: Ausschnitt aus RapidMiner: Text Mining

Die bearbeitete Tabelle wird im nächsten Schritt mit dem zuvor erstellten Repository ge-join, um lediglich Wörter mit fünf oder mehr Vorkommen und dem Vorkommen pro Entwickler zu behalten. Dieses Ergebnis wird wiederum in einer Excel-Datei abgelegt, um wieder manuell bearbeitet werden zu können (Schritt 3).

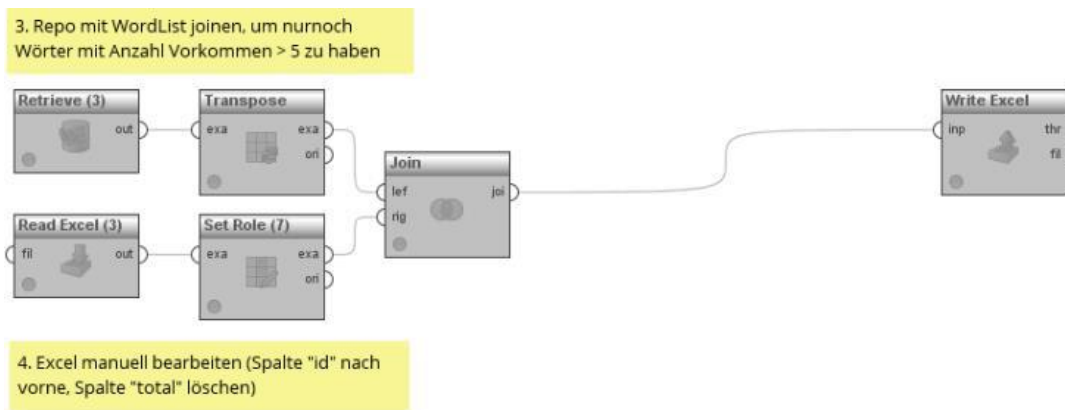
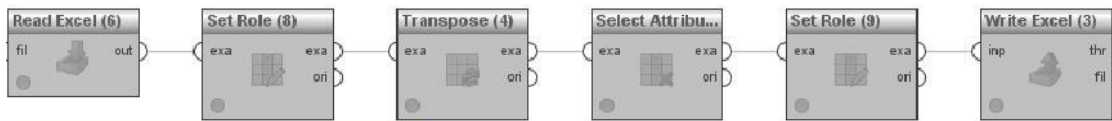


Abbildung 4: Ausschnitt aus RapidMiner: Join-Operation

Im 4. Schritt wird die Spalte „id“ an den Anfang der Tabelle sortiert und eine weitere Spalte „total“ entfernt. Die so neu entstandene Tabelle wird mit Hilfe von RapidMiner in Schritt 5. transponiert, um für die Erstellung eines Neuronalen Netzes verwendet werden zu können. Diese Tabelle wird wiederum kurz umsortiert (Schritt 6.), um die abschließende Verwendung in RapidMiner zu vereinfachen.

Die finale Excel-Tabelle hat somit die folgende Form (Abbildung 6), um im nächsten Schritt „Modeling“ verwendet zu werden.

5. Excel transponieren



6. Excel manuell bearbeiten (Spalte "authorid" nach vorne ziehen)

Abbildung 5: Ausschnitt aus RapidMiner: Transponieren der Daten

	A	B	C	D	E	F	G	H	I	J
1	authorid	ability	able	abort	absolute	absolutely	abstract	abstractimag	abstractmetl	abstractreco
2	6	0,0000	0,0000	0,0000	0,0000	0,0000	0,0000	0,0000	0,0000	0,0000
3	8	0,0016	0,0094	0,0000	0,0000	0,0000	0,0023	0,0000	0,0000	0,0000
4	9	0,0026	0,0188	0,0000	0,0032	0,0000	0,0000	0,0000	0,0000	0,0000
5	10	0,0000	0,0108	0,0000	0,0000	0,0015	0,0015	0,0000	0,0000	0,0000
6	11	0,0000	0,0122	0,0024	0,0000	0,0024	0,0012	0,0000	0,0000	0,0012
7	12	0,0000	0,0093	0,0000	0,0000	0,0000	0,0000	0,0000	0,0000	0,0000
8	13	0,0000	0,0115	0,0016	0,0000	0,0008	0,0074	0,0000	0,0008	0,0000
9	14	0,0092	0,0257	0,0000	0,0000	0,0055	0,0018	0,0000	0,0000	0,0000
10	15	0,0000	0,0000	0,0000	0,0000	0,0000	0,0000	0,0000	0,0000	0,0000
11	16	0,0072	0,0000	0,0000	0,0000	0,0000	0,0000	0,0000	0,0000	0,0000
12	17	0,0000	0,0000	0,0000	0,0000	0,0000	0,0000	0,0000	0,0000	0,0000
13	18	0,0004	0,0306	0,0000	0,0000	0,0000	0,0000	0,0000	0,0000	0,0000
14	20	0,0000	0,0096	0,0000	0,0000	0,0000	0,0145	0,0000	0,0000	0,0000
15	21	0,0000	0,0028	0,0014	0,0014	0,0000	0,0000	0,0000	0,0000	0,0000
16	22	0,0000	0,0000	0,0000	0,0000	0,0000	0,0000	0,0000	0,0000	0,0000
17	23	0,0000	0,0192	0,0000	0,0000	0,0000	0,0000	0,0000	0,0000	0,0000
18	24	0,0007	0,0069	0,0000	0,0007	0,0000	0,0131	0,0083	0,0041	0,0069
19	25	0,0023	0,0093	0,0000	0,0000	0,0000	0,0023	0,0000	0,0000	0,0000
20	27	0,0000	0,0000	0,0000	0,0000	0,0000	0,0000	0,0000	0,0000	0,0000
21	28	0,0000	0,0548	0,0000	0,0000	0,0000	0,0055	0,0000	0,0000	0,0000
??										

Abbildung 6: Ausschnitt aus der Ziel Excel-Tabelle für das Training des KNN

Für den zweiten Ansatz (TF-IDF) ist eine weniger umfangreiche Transformation der Daten notwendig. Im 1. Schritt werden hier ebenfalls die Autor-ID sowie die verknüpften Beschreibungen der Bugs mit einer Limitierung auf die ersten 32 Einträge abgerufen. Die Limitierung auf 32 Einträge entsteht durch die Begrenzung von Excel auf ca. 16.000 Attribute bzw. Spalten, welche durch 33 Einträge überschritten wird. Wie auch im ersten Ansatz werden die Daten für die Verwendung in RapidMiner in andere Datenformate konvertiert und abschließend als Excel-Tabelle gespeichert. In dieser Tabelle werden dann die Spaltensummen für jedes Wort gebildet und alle Wörter mit einer Spaltensumme von weniger als 0,05 entfernt, um den Datensatz für die Verwendung in RapidMiner deutlich zu reduzieren (Schritt 2.). Diese Tabelle hat nun die gleiche Struktur wie die in Abbildung 6 zeigte.

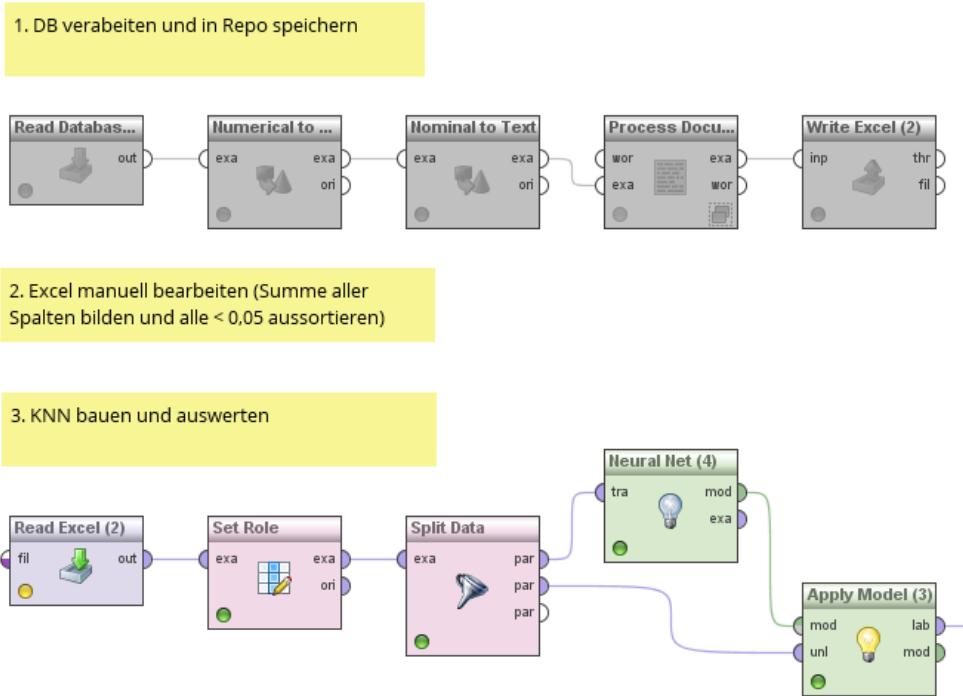


Abbildung 7: RapidMiner Text Mining Process des 2. Ansatzes

4 Modeling

Wie in den Kapiteln zuvor bereits erwähnt, wird in dieser Arbeit die Methode des Neuronalen Netzes verwendet, um die Vorhersage eines passenden Entwicklers anhand der Bugbeschreibung zu erhalten. Es wurde sich für die Methode des Neuronalen Netzes entschieden, da hier jedes Wort und seine Häufigkeit des Vorkommens mit denen aus den Entwicklerprofilen verglichen werden kann und so die Überführung in eine eindeutige Entwickler-ID möglich scheint. Auch wenn berücksichtigt werden muss, dass durch die Reduzierung der Daten in den vorherigen Schritten, kein klares Ergebnis zu erwarten ist, da die Lernmenge für ein deutliches Ergebnis höher sein muss. Doch scheinen andere Modellierungsmethoden wie z.B. das Clustering eher ungeeignet, da die Ähnlichkeit von Begriffen lediglich anhand der Anzahl der Vorkommen bewertet werden kann und nicht inhaltlich. Dies müsste manuell geschehen, was bei einer Anzahl von Begriffen von ca. 15.000 (mit den im vorherigen Kapitel beschriebenen Limitierungen) zu umfangreich ist. Durch die Vorverarbeitung der Daten (vgl. Kapitel 3) können die finalen Excel-Dateien in RapidMiner importiert und nach festlegen des Label-Attributs in ein Neuronales Netz überführt werden. Die Attribute (Spaltennamen) werden dann automatisch als Input-Knoten modelliert und die Autor-ID, die zuvor als Label definiert wurde, wird als Output-Knoten verwendet. Durch die Konvertierung der Autor-ID in ein Text-Format werden die

Einträge von RapidMiner als einzigartig behandelt und es werden keine „Berechnungen“ durchgeführt, welche zu Kommazahlen bei der Vorhersage der Autor-ID geführt hätte. Da hier kein kausaler Zusammenhang zwischen den IDs besteht, führt diese Vorgehensweise zu keinem sinnvollen Ergebnis.

Die hohe Anzahl an Attributen (Spalten), ca. 3.500 beim 1. und 3000 beim 2. Ansatz, führt entsprechend zu einer sehr hohen Anzahl an Input-Knoten des Neuronalen Netzes. Bei den Ausgabeknoten beschränkt sich die Anzahl auf lediglich ca. 20, da diese nur die betrachteten Entwickler darstellen.

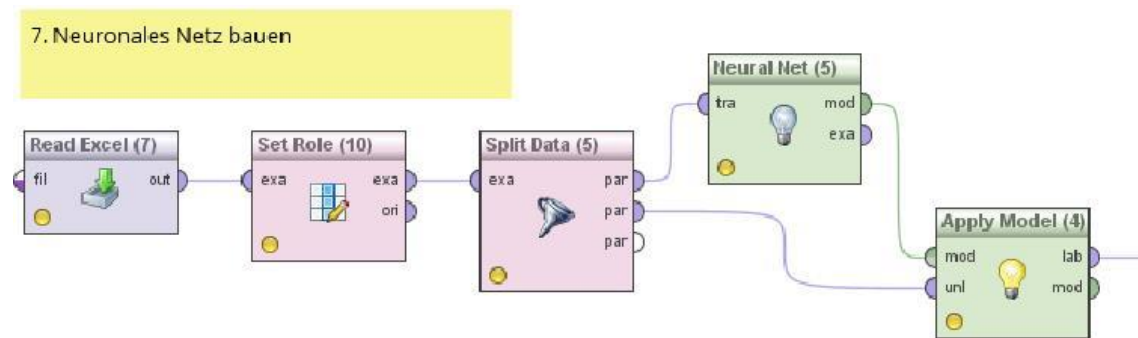


Abbildung 8: Darstellung der Generierung des KNN im ersten Ansatz

5 Evaluation

Das Ergebnis der beiden verwendeten Ansätze ist den nachfolgenden Grafiken zu entnehmen:

ExampleSet (2 examples, 22 special attributes, 3543 regular attributes)															
Row No.	authorid	prediction(a...	confidence(6)	confidence(8)	confidence(9)	confidence(...	confidence(...	confidence(...	confidence(...	confidence(...	confidence(...	confidence(...	confidence(...	confidence(...	confidence(...
1	9	10	0	0.037	0	0.962	0.000	0	0.001	0.000	0.000	0.000	0.000	0.000	0.000
2	23	10	0	0.020	0	0.979	0.000	0	0.000	0.000	0	0.000	0.000	0.000	0.000

Abbildung 9: Vorhersage des 1. Ansatzes

ExampleSet (3 examples, 34 special attributes, 2975 regular attributes)															
R...	authorid	pred...	confidence(6)	confidence(8)	confidence(9)	confidence(10)	confidence(...	confidence(...	confidence(...	confidence(...	confidence(...	confidence(...	confidence(...	confidence(...	confidence(...
1	39	8	0	1.000	0.000	0.000	0	0	0	0	0	0	0	0	0
2	41	8	0	1.000	0.000	0.000	0	0	0	0	0	0	0	0	0
3	43	8	0	1.000	0.000	0.000	0	0	0	0	0	0	0	0	0

Abbildung 10: Vorhersage des 2. Ansatzes

Die Grafiken zeigen, dass die Vorhersage beider Modelle immer den gleichen Entwickler vorschlägt und dies mit nahezu (Ansatz 1) bzw. 100%iger (Ansatz 2) Sicherheit. Beim Lesen der Bugbeschreibungen der betroffenen Entwickler (9 und 23 im Vergleich zu 10

bzw. 39, 41 und 43 im Vergleich zu 8) ist dieses Ergebnis nicht nachvollziehbar. Besonders die sehr hohe bzw. vollständige Sicherheit der Vorhersage lässt Zweifel an beiden Modellen zu.

Die Ursache dieser Fehleinschätzungen liegt vermutlich in der zu geringen Datenmengen. Es müssten weit mehr Entwickler und Bugs betrachtet werden, um ein annähernd brauchbares Ergebnis zu erhalten. Durch die sehr hohe Anzahl an Daten ist dieses Vorgehen, zumindest im Universitären Umfeld, nicht möglich, da die entsprechenden Rechenkapazitäten und Tools (Excel ist wie bereits erwähnt nicht in der Lage, große Datenmengen zu Händeln) nicht zur Verfügung stehen. Ein weiterer Grund kann die breite Streuung der durch Text Mining erhaltenen Begriffe sein. Da diese nicht ausschließlich den technischen Sachverhalt beschreiben, sondern auch Wörter des täglichen Gebrauchs sein können, müssten diese erst mit einer Bibliothek verglichen werden, um eine Art Keywords zu erhalten. Auch die Betrachtung von lediglich 20 Benutzern führt nicht zu ausreichend vielen und aussagekräftigen Entwicklerprofilen, mit denen die neuen Bugs verglichen werden können.

Eine weitere Möglichkeit zur Lösungsverbesserung könnte die Reduzierung der Daten auf die wichtigsten Keywords sein. Hierfür verfügt RapidMiner nicht über dafür nötige Operatoren, denn auch durch Ausschließung von Wörtern, die selten genannt werden, ist die Datenmenge zu groß. Dieser Ansatz würde lediglich durch eine manuelle Wahl von Keywords ermöglicht werden. Die Keywords müssten spezifische Probleme widerspiegeln und verpflichtend bei der Erstellung eines Bugs verwendet werden. Somit müssten zunächst im Bugtracker Veränderungen vorgenommen werden. Diese hätten zur Folge, dass die verschiedenen Bugs durch die Keywords eine Ordnung bekämen und somit eine bessere Grundlage bieten würden ein neuronales Netz zu verwenden.

6 Deployment

Die Ergebnisse der Datenanalyse soll dazu genutzt werden, den Entwicklern Vorschläge zu unterbreiten, welche neuen Fehlermeldungen mit hoher Wahrscheinlichkeit von Ihnen gelöst oder zumindest bearbeitet werden können. Dazu ist es notwendig, die Entwickler auf das Eintreffen neuer, zu ihnen passender Fehlermeldungen aufmerksam zu machen. Das kann entweder durch einen passiven oder einen aktiven Ansatz gelöst werden.

Den passiven Ansatz könnte beispielsweise eine Information darstellen, die dem Entwickler bei seinem nächsten Besuch des Bugtrackers angezeigt wird. Dazu könnte man

eine Tabelle generieren, die notwendigen Informationen wie den Titel des Bugs, das Datum, die Konfidenz der Zuordnung sowie einen entsprechenden Link zum Bug enthält (siehe Abbildung 11). In dieser Tabelle kann nun der Entwickler direkt zu einem vorgeschlagenen Bug springen und mit der Bearbeitung beginnen. Alternativ könnte zusätzlich die Möglichkeit eingebunden werden, die Zuordnung von Entwicklerseite als „nicht zutreffend“ zu klassifizieren. Diese Daten könnten dann zur Qualitätskontrolle und Optimierung des Modellierungsprozesses genutzt werden.

Before reporting a bug, please read the [bug writing guidelines](#), please look at the list of [most frequently reported bugs](#), and please search for the bug.

Show Advanced Fields (* = Required Field)

* Product: Platform Reporter: cbenz@uos.de

Component: **Ant** Component Description: Eclipse Platform Debug framework
 CVS
 Debug
 Doc
 IDE
 Incubator

* Version: 4.4.1 ^
 4.4.2
 4.5
 4.5.1
 4.6 v

Severity: minor
 Hardware: PC
 OS: Windows NT

We've made a guess at your operating system and platform. Please check them and make any corrections if necessary.

* Summary: crash on startup

Bug ID	Summary	Status	
25628	startup - crash	RESOLVED WONTFIX	Add Me to the CC List
34175	Crash on startup	RESOLVED FIXED	Add Me to the CC List
47296	[KeyBindings] Fails on Startup	RESOLVED INVALID	Add Me to the CC List
71619	Crash at startup.	RESOLVED WORKSFORME	Add Me to the CC List
135763	Crash on startup	RESOLVED FIXED	Add Me to the CC List
169711	Crash on startup	RESOLVED INVALID	Add Me to the CC List
430736	Crash at Eclipse startup under Debian Wheezy stable - Problematic frame: C [[libgdk-x11-2.0.so.0+0x5173f] gdk_display_open+0x3f	RESOLVED WONTFIX	Add Me to the CC List

Description:

Abbildung 11: Beispielhafte Integration der Ergebnisse in Bugzilla

Der aktive Ansatz ist durch eine direkte Benachrichtigung des Entwicklers umsetzbar. Da im realen System bereits die E-Mail-Adressen vorhanden sind, können diese im Rahmen der existierenden Benachrichtigungsfunktionen genutzt werden um bei eintreffenden Fehlermeldungen und einem ausreichenden Konfidenzniveau eine E-Mail an den Entwickler zu schicken. Darin können dann sowohl die Informationen analog zum passiven Ansatz als auch initiale Informationen zur Fehlermeldung enthalten sein. Über Links in der E-Mail kann der Entwickler dann entweder direkt den Bug oder die oben genannte Tabelle erreichen. Parameter, wie die Anzahl der Benachrichtigungen pro Tag oder das mindeste Konfidenzniveau könnten dabei vom Benutzer festgelegt werden.

Um die erwähnten Funktionen im produktiven Umfeld einzusetzen, sind sowohl Änderungen am Datenbankaufbau als auch in der Weboberfläche des Bugtrackers notwendig. Gegebenenfalls müsste bei einer aktiven Variante auch geprüft werden, ob der Entwickler sein Einverständnis zum Erhalt der E-Mail-Benachrichtigungen geben muss.

Da der aktive Ansatz den passiven Ansatz bedingt, muss die Datenbank in jedem Fall um eine neue Tabelle ergänzt werden, welche die Zuordnungen sowie die Konfidenzinformationen als auch ein mögliches Feedback des Entwicklers enthält. Auf Änderungen in Bezug auf die Darstellung auf der Webseite des Bugtrackers und die Benachrichtigungen per E-Mail wird nicht im Detail eingegangen, da dies tiefgreifende Kenntnisse zur Funktionsweise von Bugzilla erfordert und den Rahmen bei entsprechender Würdigung somit sprengen würde.

Prototypische Implementierung zur systemseitigen Früherkennung von Dubletten beim Anlegen neuer Einträge im Eclipse Bugtracker

Katrin Baalman, Ulrike Hinrichs und Stephanie Klatte

Abstract. Ziel dieser Ausarbeitung ist die frühzeitige Erkennung von doppelten Bugmeldungen, um den Datenbestand des „Eclipse Bugtrackers“ möglichst übersichtlich und gering zu halten. In RapidMiner, Excel und verschiedenen SQL-Programmen wurden die Daten analysiert, aufbereitet und mithilfe von Data- und Text-Mining-Verfahren bearbeitet. Der Fokus lag hierbei auf der Naive Bayes- und der k-NN-Methode, wobei sich letztere bewährt hat.

1 Business Understanding

Das Ziel dieser Ausarbeitung besteht darin, doppelte bzw. mehrfache Bugmeldungen, sog. Dubletten, bereits bei einer potenziellen Neuanlage eines Bugs zu vermeiden. Derzeit können neue Bugmeldungen von Usern angelegt werden, ohne dass zuerst systemseitig geprüft wird, ob dieser Sachverhalt bereits als Bug gemeldet wurde. Somit kann eine fehlende oder fehlerhafte Suche durch den User zu der Anlage von einer Bugmeldung führen, die bereits im Bestand vorhanden ist und ggf. schon durch andere Nutzer oder Mitarbeiter gelöst wurde. Der Datenbestand wird durch das bisherige Vorgehen unübersichtlich und künstlich vergrößert. Von allen erstellten Bugs im Eclipse Bugtracker ist davon auszugehen, dass ungefähr 20 % Dubletten eines bereits erstellten Bugs sind.¹

Findet ein User bzw. ein Mitarbeiter eine Dublette, wird diese manuell gemeldet und nachträglich im System geschlossen. Zusätzlich gibt es ein Programm von Eclipse, welches die Bugmeldungen analysiert und eventuelle Dubletten versucht zu identifizieren. Zu erkennen ist dieses an den Daten der Bearbeiter, von denen die Bugmeldung als Dublette gemeldet wurde. Zudem erfolgt eine Verlinkung zwischen der Dublette und der zuerst angelegten Bugmeldung. Durch diesen Prozess werden Mitarbeiterressourcen zeitweilig unnötig gebunden.

¹ Vgl. Anvik, John et al. (2005).

Der Projektplan in Tabelle 2 stellt eine grob definierte Aufgabenreihenfolge vor.

Tabelle 2: Projektplan ²

Aufgabe	Tool	Ziel
Data Understanding	RapidMiner, Excel	Daten verstehen und mögliche Probleme erkennen.
Festlegung der Aufgabenstellung	Stift & Papier, Word, PowerPoint	Eines der gefundenen Probleme auswählen und Lösungsansätze finden.
Data Preparation	Excel, HeidiSQL, MySQL Workbench	Vorbereitung der Daten zur Analyse sowie Erstellung eines Testdatensatzes.
Keywords in einen Prozess einbauen	RapidMiner	Erste Vorsortierung, um Dubletten zu erkennen.
Text Mining	RapidMiner	Anwendung eines Verfahrens, um Dubletten zu erkennen.
Naive Bayes	Excel, HeidiSQL, MySQL Workbench, RapidMiner	Anwendung eines Klassifikationsverfahrens zur Dublettenerkennung.
k-NN	Excel, HeidiSQL, MySQL Workbench, RapidMiner	Anwendung eines Algorithmus zur Dublettenerkennung.
Evaluierung	Excel, RapidMiner	Auswertung der unterschiedlichen Verfahren.

Wie dem Projektplan zu entnehmen ist, werden im Rahmen dieser Arbeit verschiedene Verfahren zur Dublettenerkennung zur Anwendung kommen. Das Ziel dieses Projektes ist letztendlich, die angewandten Verfahren miteinander zu vergleichen und zu ermitteln,

² Eigene Tabelle.

welches der Problemlösung am nächsten kommt, d. h. welches Verfahren Dubletten am besten automatisiert erkennen kann.

2 Data Understanding

Im Eclipse Bugtracker können die Bugmeldungen mit den zugehörigen Erläuterungen und Eigenschaften eingesehen werden. Mittels des Programmes RapidMiner war es möglich, die Daten in Excel ausgeben zu lassen, um eine bessere Analyse durchführen zu können.

Zur Abgrenzung der einzelnen Bugmeldungen untereinander hat jeder Bug seine eigene *bug_id* und wird einer Person zugeteilt (*assigned_to*), die den Bug bearbeitet. Zu jedem Bug existiert zudem eine kurze Beschreibung (*short description*). Diese sind qualitativ jedoch nicht in jedem Fall hilfreich, da es sich hierbei nur um eine Art Überschrift für den Bug handelt und somit nicht viele Informationen zu dem speziellen Problem beinhaltet. Zudem sind die sog. Überschriften teilweise gar nicht oder nur mit einem Wort sowie mit Testdatensätzen gefüllt. Es könnte folglich dazu kommen, dass zwei Bugs die gleiche oder sehr ähnliche kurze Beschreibung haben, es sich jedoch um zwei unterschiedliche Probleme handelt.

Der *bug_status* gibt dem User Auskunft über den aktuellen Status der Bugmeldung. Mittels der *resolution* ist zu erkennen, ob das Problem der Bugmeldung bereits gelöst werden konnte oder es sich bspw. um eine Dublette handelt.

Für unser Problem ebenfalls interessant sind die während der Anlage der Bugmeldung bereits zugeteilten *keywords*. Das Problem hierbei ist, dass es lediglich 28 unterschiedliche Angaben gibt und erneut nicht jeder Bug einen Eintrag besitzt. Lediglich 7 % der Bugs ist ein Keyword zugeordnet, nur 16 % von diesen mehr als ein einziges.

Dubletten könnten durch die angelegten Keywords schneller erkannt werden, sofern den Bugmeldungen mehrere Keywords zugeordnet wären. Da derzeit im System bei den Bugmeldungen sehr wenig Keywords hinterlegt sind, kann die gegebene Bugtabelle nicht zur Lösung der Früherkennung von Dubletten beitragen.

Abbildung 12 zeigt einen beispielhaften Eintrag aus dem Eclipse Bugtracker. Anhand dieser Bugmeldung mit der *bug_id* 444551 ist deutlich zu sehen, dass die Erkennung von Dubletten ein großes Problem darstellt. Allein dieser Bugmeldung sind bisher 30 andere, später angelegte Meldungen als Duplicates (Dubletten) zugeordnet.

Bug 444551 - [egit] Internal Error on switching to a different branch (err_grp: 89d917ef)

Status: UNCONFIRMED

Product: EGit

Component: Core

Version: unspecified

Hardware: All All

Importance: P3 critical ([vote](#))

Target Milestone: ---

Assigned To: Project Inbox CLA

QA Contact:

URL:

Whiteboard:

Keywords:

Duplicates: [444552](#) [449397](#) [449808](#) [449809](#) [449814](#) [449815](#) [449816](#) [450197](#) [450198](#) [450199](#) [450200](#) [450201](#) [450429](#) [450614](#) [451085](#) [451106](#) [451137](#) [451138](#) [451528](#) [451545](#) [451548](#) [451569](#) [451570](#) [451718](#) [451720](#) [452067](#) [452068](#) [452103](#) [452104](#) [452747](#) ([view as bug list](#))

Reported: 2014-09-19 01:24 EDT by Marcel Bruch CLA

Modified: 2015-06-25 00:39 EDT ([History](#))

CC List: 5 users ([show](#))

See Also:

Abbildung 12: Beispielhafter Auszug eines Bugs aus dem Eclipse Bugtracker³

Die Ursache für das Nichterkennen von Dubletten liegt darin, dass jeder User identische Bugs mit sehr unterschiedlichen Beschreibungen anlegen kann. Dabei sind verschieden umfangreiche Ausprägungen vorhanden, von einer Fehlerbeschreibung mit lediglich ein paar Wörtern bis zu einer vollständigen Beschreibung des Ablaufs. Somit ist die Qualität der Beschreibungen der Bugs sehr unterschiedlich, da diese nicht nach einem vorgegebenen Protokoll durchgeführt werden. Das gleiche gilt für die Kommentare, die die User bei einem Bug hinterlassen können. Wie zuvor bei den *short descriptions* der Bugmeldung kann es bei der Volltextausgabe mit den Beschreibungen und Kommentaren aufgrund der sehr diversifizierten Qualität ebenfalls zu ähnlichen Beschreibungen kommen, die jedoch keine Dubletten darstellen.

³ Vgl. Bruch, Marcel (2014).

3 Data Preparation

Bei den zur Verfügung gestellten Daten handelt es sich mit 316.911 eingetragenen Bugs um eine große Datenbasis, die von herkömmlichen Rechnern nicht in angemessener Zeit bearbeitet werden kann. Aus diesem Grund wird in einem vorbereitenden Schritt ein eingeschränkter Datensatz erstellt, der einen beispielhaften Ausschnitt der für die Bearbeitung der Aufgabenstellung relevanten Datensätze beinhaltet.

Um die Auswahl der Daten für die Bearbeitung zu beschränken, wurde zuerst unter Verwendung des Beispielprozesses die gesamte Datenbasis in Excel ausgegeben. Dazu wurden unter anderem mithilfe des *Tokenize* Operators die Tokens festgelegt. Des Weiteren wurden Stopp-Wörter herausgefiltert sowie mithilfe des *Filter Tokens* Operators alle Wörter eliminiert, die weniger als drei und mehr als 25 Buchstaben besitzen. Ausgeschlossen wurden zudem Wörter von der zuvor angelegten Liste der Keywords.

Auffällig ist, dass am Anfang des Datenzeitraumes das Anlegen und Bearbeiten von Bugmeldungen extensiv getestet wurde. Diverse Einträge enthalten keine sinnvollen Buchstabenkombinationen. Da im Laufe der Zeit aufgrund der steigenden Nutzung des Eclipse Bugtrackers mehr Daten zur Verfügung standen, werden im Rahmen dieser Arbeit nur Einträge aus dem Zeitraum Mai 2008 zur Analyse herangezogen. Hierbei sollte es sich um einen sinnvollen und repräsentativen Auszug der Daten handeln, da weder Testdaten aus der Anfangsphase noch wenig bearbeitete Daten aus dem Ende des Ausschnittszeitraumes enthalten sind. Die verbleibende Anzahl an 5.260 Datensätzen kann von herkömmlichen Computern in angemessener Zeit bearbeitet werden.

Nach der Begrenzung des Zeitraumes müssen im nächsten Schritt die relevanten Daten zusammengeführt werden. Hierfür wurden SQL-Abfragen in HeidiSQL und MySQL Workbench durchgeführt. Die relevanten Daten lauten: *bug_id*, *op_sys* (Betriebssystem), *prod_name* (Name des Produktes), *comp_name* (Name der Komponente), *short_desc* und *comments*. Bei letzterem handelt es sich entgegen des Namens der Spalte nicht um Kommentare zu einem Bug, sondern um die Langbeschreibung des Problems.

Abbildung 13 zeigt eine beispielhafte SQL-Abfrage zur Erstellung der verknüpften Daten in HeidiSQL. Hierbei ist anzumerken, dass es sich bei ‚*g02_bugs_may08*‘ um eine zuvor erstellte View handelt, die die Daten der umfangreichen *bugs*-Tabelle auf den zu Anfang ausgewählten Zeitraum Mai 2008 beschränkt.

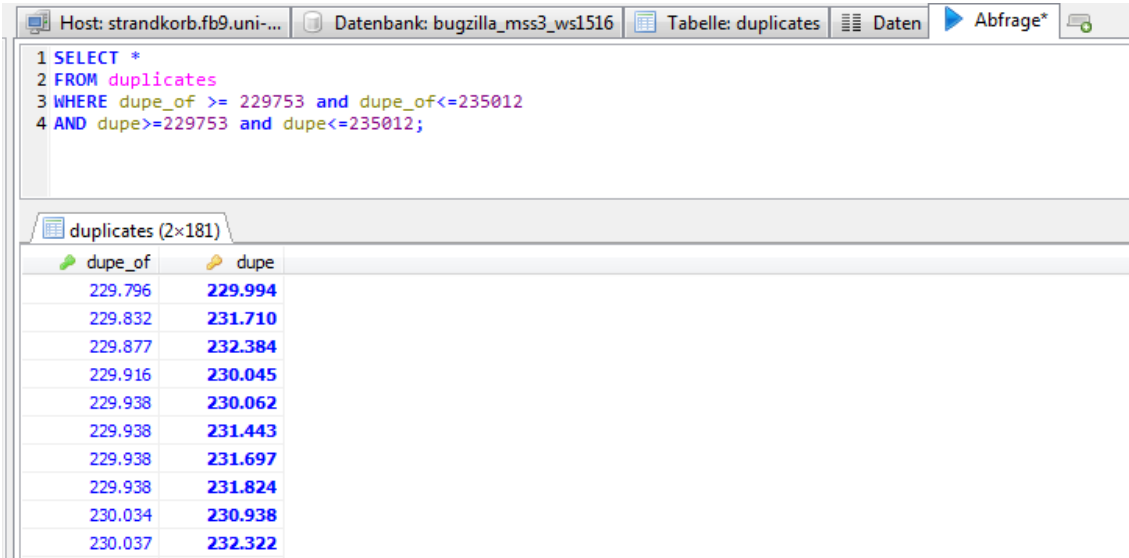
```

1 select `g02_bugs_may08`.`bug_id` AS `bug_id`,
2       `g02_bugs_may08`.`op_sys` AS `op_sys`,
3       `products`.`name` AS `prod_name`,
4       `components`.`name` AS `comp_name`,
5       `g02_bugs_may08`.`keywords` AS `keywords`,
6       `g02_bugs_may08`.`short_desc` AS `short_desc`,
7       `bugs_fulltext`.`comments` AS `comments`
8 from (((`g02_bugs_may08`
9  join `products` on((`g02_bugs_may08`.`product_id` = `products`.`id`)))
10 join `components` on((`g02_bugs_may08`.`component_id` = `components`.`id`)))
11 join `bugs_fulltext` on((`g02_bugs_may08`.`bug_id` = `bugs_fulltext`.`bug_id`)))

```

Abbildung 13: SQL-Anfrage in HeidiSQL zur Erstellung der verknüpften Daten⁴

Daneben wurde noch ein zweiter Datensatz zur Analyse der Dubletten erstellt. Mithilfe einer SQL-Abfrage wurden die Bugs aus dem Zeitraum Mai 2008 mit ihren ggf. vorhandenen Dubletten verknüpft. Abbildung 14 zeigt die SQL-Abfrage und einen Ausschnitt der verknüpften Daten.



```

1 SELECT *
2 FROM duplicates
3 WHERE dupe_of >= 229753 and dupe_of<=235012
4 AND dupe>=229753 and dupe<=235012;

```

dupe_of	dupe
229.796	229.994
229.832	231.710
229.877	232.384
229.916	230.045
229.938	230.062
229.938	231.443
229.938	231.697
229.938	231.824
230.034	230.938
230.037	232.322

Abbildung 14: SQL-Abfrage in HeidiSQL zur Verknüpfung der Bugs aus Mai 2008 mit ggf. vorhandenen Dubletten im gleichen Zeitraum⁵

Die Daten der Bugs und ihrer zugeordneten Dubletten wurden anschließend manuell in einer zuvor erstellten Excel-Tabelle mit den relevanten Angaben der Bugs ergänzt.

⁴ Eigene Abbildung.

⁵ Eigene Abbildung.

Darüber hinaus wurde ein Testdatensatz in Excel erstellt, der in Abbildung 15 ersichtlich ist. Dazu wurden Einträge aus dem Basisdatensatz kopiert, einige Einträge willkürlich erfunden und zusätzliche Einträge erstellt, die zum Teil aus vorhandenen Daten, zum Teil aus geänderten Daten bestehen. So soll ausgewertet werden, wie gut das letztendlich ausgewählte Verfahren mit unbekanntem und teils bekannten Daten umgeht.

bug_id	op_sys	prod_name	comp_name	short_desc	comments
234976	All	Higgins	STS	Use Apache Common Logging - sts.binding.a	change the component to
234676	Windows XP	CDT	cdt-core	Externalize empty project type template	The Empty Project project
	All			Kein Ergebnis.	Kein Ergebnis.
	Linux	Platform	UI	Hier soll es keinen Treffer geben.	
	Windows XP	GMF	Models - Graphical	Support SVG figures in gmfgraph models	
230490	All	IMP	LPG IDE	Doesn't work on Linux	The generator can't find tl
230491	All	IMP	LPG IDE	Doesn't work on Linux	The generator can't find tl
230816	Windows XP	ACTF	AI	bug_1	first bugHi.Please let us kr
233112	Windows XP	ACTF	AI	test	testnot a bug report.close
	Windows XP	ACTF	AI	test	first bugHi.Please let us kr
	Linux	Platform	UI	[infer type arguments] The fix 'Add type arg	What steps will reproduce
232735	All	JDT	UI	[infer type arguments] The fix 'Add type arg	What steps will reproduce

Abbildung 15: Ausschnitt der Tabelle mit den Testdatensätzen⁶

Idealerweise soll das Verfahren die abgeänderten Daten einem ähnlichen, bereits angelegten Datensatz zuordnen. Das würde bedeuten, dass Dubletten erkannt werden, auch wenn die Angaben der Bugs nicht identisch sind. Wie bereits zu Anfang dieser Ausarbeitung erwähnt, ist es in der Regel nicht der Fall, dass ein erneut angelegter Bug von einer zweiten Person genauso beschrieben wird wie von der ersten.

⁶ Eigene Abbildung.

4 Modeling

Im Rahmen dieses Projektes kamen verschiedene Data-Mining-Verfahren zur Anwendung. Während der Fokus zu Beginn auf rudimentären Text-Mining-Methoden und Entscheidungsbäumen lag, hat sich im Laufe der Bearbeitung gezeigt, dass damit alleine keine zufriedenstellenden Ergebnisse erreicht werden können. Daraufhin fiel die Wahl auf Naive Bayes und k-NN, auf welche im Folgenden genauer eingegangen wird.

4.1 Naive Bayes

Beim Naive Bayes handelt es sich um einen Operator, der potenzielle Dubletten den Bugs zuordnet, mit denen sie mit höchster Wahrscheinlichkeit übereinstimmen. Der Bayes-Klassifizierer arbeitet basierend auf dem Bayesschem Theorem. Dabei wird die Annahme starker Unabhängigkeit getroffen. Dieses bedeutet, dass die An- oder Abwesenheit bestimmter Faktoren nicht mit der An- oder Abwesenheit anderer Faktoren zusammenhängt. Vorteilhaft dabei ist, dass die Eigenschaften einzeln bewertet werden, sodass z. B. dasselbe Problem in Windows und Linux trotz des unterschiedlichen Betriebssystems als Dublette erkannt wird. Zudem benötigt der Naive Bayes nur relativ wenig Datensätze, um den Trainingsdatensatz zu schätzen und auf den Testdatensatz anzuwenden.⁷

Das Naive Bayes-Verfahren wird zuerst auf einen einheitlichen Trainings- und Testdatensatz angewendet, anschließend auf zwei unterschiedliche Datensätze. Der Grund dafür ist, dass verschiedene Fragestellungen zugrunde liegen, die die unterschiedlichen Datensätze sowie leichte Abweichungen in der Vorgehensweise begründen. Nachfolgend werden diese Ansätze im jeweiligen Modell erläutert.

Einheitlicher Trainings- und Testdatensatz

Der einheitliche Datensatz wurde aus der Excel-Datei eingelesen und bestand aus den 5.260 Bugs, die aus Mai 2008 stammen. Eingelesen wurden die Attribute *bug_id*, *op_sys*, *prod_name*, *comp_name*, *short_desc*, *comments* und die manuell in Excel hinzugefügte binomiale Information, ob es sich bei dem Bug um eine Dublette (*Duplicate*) handelt oder nicht. Als *label* wurde bei dem Operator *Set Role Duplicate* gewählt. Dadurch soll herausgefunden werden, ob die Bugs Dubletten sind oder nicht. Abbildung 16 zeigt diesen Prozess auf.

⁷ Vgl. RapidMiner (2016).

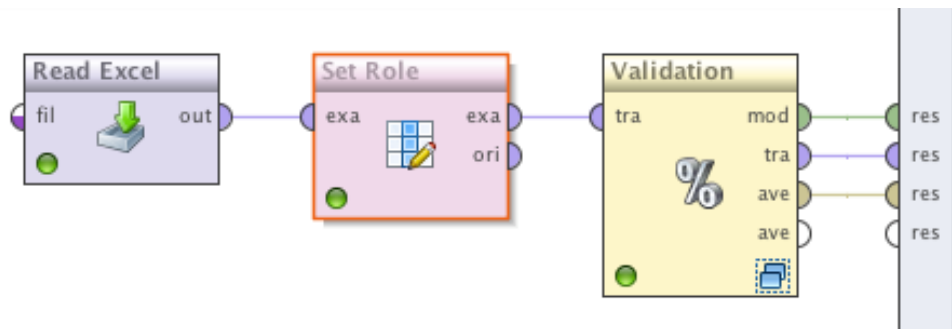


Abbildung 16: Naive Bayes-Prozess beim einheitlichen Datensatz in RapidMiner⁸

Der Operator *Validation* wird verwendet, um den Trainings- und Testdatensatz anzulegen und das Modell anzuwenden. Die Aufteilung in Trainings- und Testdatensatz beträgt 60 % zu 40 %, also 3.156 Bugs zum Trainieren und 2.104 Bugs zum Testen.

In Abbildung 17 ist die Unterordnung des Operators *Validation* zu sehen. Zuerst wird in einem eigenständigen Prozess anhand des Naive Bayes trainiert. Danach ist zu erkennen, dass das Model auf den Testdatensatz, also die restlichen 40 % des Datensatzes, angewendet wird.

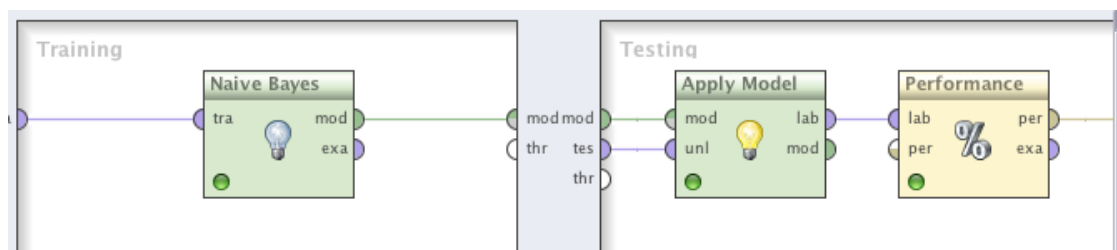


Abbildung 17: Unterordnung des Validation-Operators in RapidMiner⁹

Vorteilhaft ist beim Operator *Validation*, dass er eine direkte Evaluation des Modells durchführt, hier durch den Operator *Performance*. Die Evaluation bzw. die Performance-Messung wird in Kapitel 5.1 ausgewertet.

Unterschiedliche Trainings- und Testdatensätze

Bei der zweiten Schätzung mithilfe des Naive Bayes wird mit zwei unterschiedlichen Datensätzen gearbeitet, um zwei neue Fragestellungen anzugehen. Zum einen soll wie zuvor in Erfahrung gebracht werden, ob eine Dublette von einem anderen Bug vorliegt.

⁸ Eigene Abbildung.

⁹ Eigene Abbildung.

Zum anderen soll die *bug_id* von einer Dublette dem zugehörigen Bug zugeteilt werden. Darüber hinaus soll getestet werden, ob dem Zeitraum, in dem die Bugs vorliegen, passende Bugs aus dem gesamten Zeitraum zugeteilt werden können, d. h. nicht nur Bugs aus dem betrachteten Zeitraum. Aufgrund des nun größeren Zeitraumes und der gestiegenen Dublettenanzahl ist die Wahrscheinlichkeit höher, dass es mehrere Dubletten zu einem Bug gibt.

In Abbildung 18 ist der Modellierungsprozess abgebildet. Zuerst wird erneut die Excel-Tabelle mit den 5.260 Bugs aus Mai 2008 eingelesen. Dabei werden dieselben Attribute wie zuvor genutzt, bis auf *Duplicate*. Beim Einlesen wird die *bug_id* als *label* bestimmt, um die gewünschten *bug_id*-Zuteilungen zu erzielen. Anschließend wird auf Basis dieses Datensatzes der Naive Bayes trainiert. Somit hat der Naive Bayes die Möglichkeit, anhand von 181 bestehenden Dubletten seine Prognosen und Wahrscheinlichkeiten anzupassen.

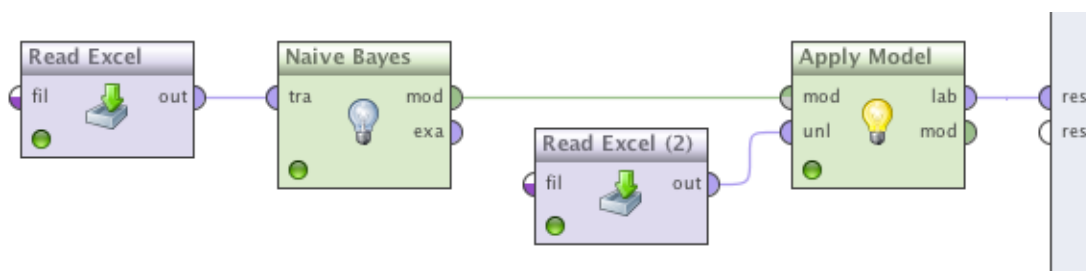


Abbildung 18: Naive Bayes-Prozess bei unterschiedlichen Datensätzen in RapidMiner¹⁰

Anschließend wird das Modell auf einen Datensatz aus einem größeren Zeitraum angewendet. Da anhand der Daten aus Mai 2008 trainiert wird und für diesen Zeitraum insgesamt 426 Dubletten aus dem gesamten Datenzeitraum existieren, werden genau diese Dubletten eingelesen. Im Idealfall werden alle 426 Dubletten richtig identifiziert und zugeteilt. Bei diesem erweiterten Datensatz (*Read Excel (2)*) werden ebenfalls die *bug_id* als *label* gewählt und die gleichen Attribute wie beim Trainingsdatensatz eingelesen. Mithilfe des Operators *Apply Model* wird das Modell auf den neuen Testdatensatz angewendet. Die Auswertung dieser Ergebnisse erfolgt ebenfalls in Kapitel 5.1.

¹⁰ Eigene Abbildung.

4.2 k-NN

Bei der k-NN-Methode handelt es sich um ein Klassifikationsverfahren, bei der Schätzungen und Vorhersagen auf Basis von Mustern in den existierenden Daten vorgenommen werden. Dabei werden für alle Objekte die nächsten Nachbarn (Nearest Neighbours, NN), also ähnliche Objekte, ermittelt. Das ‚k‘ steht hierbei für die Anzahl der Nachbarn, die in die Klassifikation einfließen. Ein Objekt wird in die Klasse eingeteilt, in der es die meisten der k Nachbarn hat bzw. erhält die gleiche Vorhersage wie seine nächsten Nachbarn.¹¹

Die Vorteile dieses Verfahrens liegen darin, dass es relativ simpel und schnell ist sowie über eine hohe Skalierbarkeit verfügt. Es sind außerdem keine linearen Beziehungen zwischen den Daten nötig.¹²

Allerdings ist bei diesem Algorithmus unklar, wie viele Daten zum Vergleich herangezogen werden und wie viele Nachbarn einen Durchschnitt bilden. Darüber hinaus erhält man keine Einsicht in die Effektivität der Merkmale, d. h. welche Eigenschaften am wichtigsten für eine gute Vorhersage sind.¹³

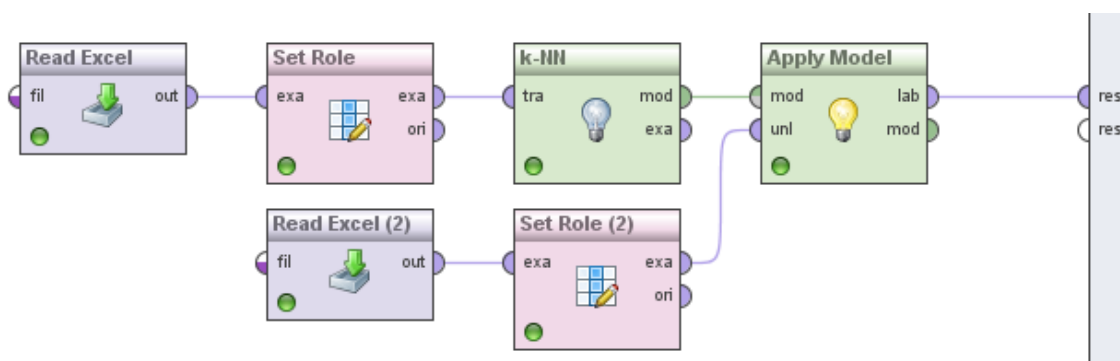


Abbildung 19: k-NN-Prozess in RapidMiner¹⁴

In RapidMiner ist die Anwendung des k-NN-Operators leicht verständlich. Abbildung 19 zeigt den im Projekt angewandten Prozess auf.

¹¹ Vgl. Miner et al. (2012, S. 899-901).

¹² Vgl. Miner et al. (2012, S. 899-901).

¹³ Vgl. Miner et al. (2012, S. 899-901).

¹⁴ Eigene Abbildung.

Bei der ersten eingelesenen Excel-Datei (*Read Excel*) handelt es sich um die in Kapitel 3 beschriebene selbsterstellte Datei mit allen für diese Aufgabe relevanten Daten zu den Bugs aus Mai 2008. Hier wird mithilfe des Operators *Set Role* das Attribut *bug_id* als *label* gesetzt. Anschließend wird der k-NN-Algorithmus angewandt.

Die Ergebnisse werden dem *Apply Model*-Operator zugeführt. Gleichzeitig braucht dieser Operator auch Testdaten. In *Read Excel (2)* wird die aus Kapitel 3 bekannte selbsterstellte Testdatei eingelesen. Auch hier wird mithilfe von *Set Role (2)* das Attribut *bug_id* als *label* gesetzt.

Zuletzt werden die Ergebnisse des angewandten Modells ausgegeben, welche im folgenden Kapitel 5.2 vorgestellt und interpretiert werden.

5 Evaluation

Bei der Evaluation wird erneut zwischen Naive Bayes und dem k-NN-Algorithmus unterschieden. Bei ersterem wurde zum Testen ein Teil der Ursprungsdaten verwendet, bei letzterem ein selbsterstellter Testdatensatz.

5.1 Naive Bayes

Auch bei der Evaluation wird zwischen den einheitlichen und den unterschiedlichen Datensätzen getrennt, da die einzelnen Ansprüche an die beiden Verfahren und zugrunde gelegten Daten deutlich voneinander abweichen.

Einheitlicher Trainings- und Testdatensatz

Wie in Kapitel 4.1 erläutert, wird die direkte Evaluation in dem Prozess durch den Operator *Performance* bei den Ergebnissen in RapidMiner durchgeführt und ausgegeben. In Abbildung 20 ist die Ergebnisausgabe des Schätzungsvorganges dargestellt.

accuracy: 44.96%			
	true no	true yes	class precision
pred. no	851	30	96.59%
pred. yes	1128	95	7.77%
class recall	43.00%	76.00%	

Abbildung 20: Performance Messung des *Validation* Operators in RapidMiner¹⁵

¹⁵ Eigene Abbildung.

Die Schätzung ist zu knapp 45 % richtig, zu sehen bei ‚*accuracy: 44.96%*‘. Dieses war unter Betrachtung unterschiedlicher Aufteilungen in Trainings- und Testdatensätze das bestmögliche Ergebnis bei einem Verhältnis von 60-40. Dabei wurde die Aufteilung nicht jedes Mal neu und per Zufall durchgeführt, sondern zuvor im *Validation Operator* als fest eingestellt, um vergleichbare Werte zu erzielen.

Es wird mit einer Genauigkeit von ca. 97 % richtig hervorgesagt, dass bei einem Bug keine Dublette besteht. Jedoch wird in 1.128 Fällen falsch vorhergesagt, dass eine Dublette besteht. Lediglich in 95 Fällen wird das Vorliegen einer Dublette richtig vorhergesagt, woraus sich beim Vorliegen einer Dublette eine niedrige richtige Prognose von ca. 8 % ergibt.

Abschließend kann festgehalten werden, dass die Aufgabe, eine Aussage über das Vorhandensein einer Dublette zu finden, richtig umgesetzt wurde. Dabei fallen die Ergebnisse allerdings sehr schlecht aus, sodass diese Methode nicht verlässlich ist. Ebenfalls weicht diese Prognose von dem Startziel ab, eine direkte Empfehlung für vorhandene Dubletten mit Angabe der *bug_id* zu erzielen. Außerdem ist dieses Vorgehen nicht auf neue Datensätze anwendbar. Folglich ist diese Vorgehensweise ungenügend.

Unterschiedliche Trainings- und Testdatensätze

Bei den zwei unterschiedlichen Datensätzen wird bei der Ergebnisausgabe in RapidMiner die *bug_id* des Testdatensatzes ausgegeben und einer geschätzten *bug_id* des zuerst eingelesenen Trainingsdatensatzes zugeteilt. Somit wird das Ziel einer direkten *bug_id*-Zuteilung von Dubletten erfüllt.

Als weiteres Ziel wurde definiert, dass aufgrund des Vorhandenseins mehrerer Dubletten eine mehrfache Zuteilung einer Dublette möglich sein sollte. Wie in Abbildung 21 anhand der roten Umrandung zu erkennen ist, wird dieses Zwischenziel ebenfalls umgesetzt.

	A	B	C	D	E	F	G	H	I
1	bug_id	prediction	correct						
2	35144	233106	231472	-		Korrekt:	11		2,58%
3	76873	231873	230307	-					
4	87368	234053	230854	-					
5	110533	229769	231112	-					
6	126471	229969	229969	MATCH					
7	131123	229769	234849	-					
8	132315	229769	231112	-					
9	147380	230109	233843	-					
10	160810	229866	231964	-					

Abbildung 21: Auswertung der Ergebnisse des Naive Bayes-Prozesses bei unterschiedlichen Datensätzen in Excel¹⁶

Als letztes erfolgt die Evaluation dieser Methode anhand der Korrektheit ihrer Prognose. Hierzu werden die Ergebnisse aus RapidMiner in Excel übertragen, in Abbildung 21 dargestellt als *bug_id* und die geschätzte *prediction(bug_id)*. In der Spalte *correct* wird die richtige *bug_id* angegeben, die zu dem Bug eine Dublette ist. Mithilfe einer Wenn-Dann-Funktion erfolgt die Ermittlung der korrekt hervorgesagten Dubletten, in der Abbildung dargestellt durch ‚MATCH‘.

In dem grünen Kasten in Abbildung 21 ist ersichtlich, dass von 426 Dubletten lediglich 11 richtig prognostiziert werden. Dieses entspricht einer Erfolgsquote von 2,58 %. Zur Ermittlung von Dubletten sollte dieses Vorgehen also nicht genutzt werden, weil die richtige Prognose auf Grundlage des Naive Bayes nicht mit einer ausreichenden Qualität und Verlässlichkeit garantiert werden kann.

Des Weiteren wird jedem eingelesenen Bug immer eine *bug_id* zugeteilt. Dieses ist hier sinnvoll, da es genau die 426 Dubletten für Mai 2008 sind, eine Zuteilung folglich immer möglich sein müsste. Wird der Datensatz jedoch erweitert, besitzt nicht mehr jeder Bug eine Dublette. Es müsste also mit einem Schwellenwert gearbeitet werden, bei dem erst ab einer gewissen Wahrscheinlichkeit ein Ergebnis bzw. eine Bug-Prognose ausgegeben werden darf. Diese Ergebnisse müssten im Anschluss noch manuell überprüft werden.

Außerdem werden nur *bug_ids* zugeteilt, anhand derer der Naive Bayes zuvor trainiert wurde. Somit müssten an beiden Stellen des Einlesens der Daten die vollständigen bis dahin vorhandenen Daten genutzt werden.

¹⁶ Eigene Abbildung.

Insgesamt lässt sich also festhalten, dass mithilfe des Naive Bayes nicht die gewünschten Ziele erreicht werden, zumindest nicht in einer akzeptablen Qualität. Das Verfahren müsste noch weiter angepasst und verändert werden. Ob dies jedoch möglich ist, um damit schlussendlich zufriedenstellende Ergebnisse zu erreichen, ist an dieser Stelle zu bezweifeln.

5.2 k-NN

Abbildung 22 zeigt die Auswertung der Testdatei in RapidMiner. In der Spalte *prediction(bug_id)* soll der Bug angegeben werden, der dem in der Testdatei am ähnlichsten ist. Blau markiert sind die Vorhersagen, die mit dem angegebenen Bug bzw. dem Bug, der leicht abgewandelt wurde, übereinstimmen. Rot markiert sind zwei Vorhersagen, die falsch sind.

Row No.	bug_id	prediction(bug_id)	op_sys	prod_name	comp_name	short_desc	comments
1	234976	234976	All	Higgins	STS	Use Apache Common Logging - s	change the compone
2	234676	234676	Windows XP	CDT	cdt-core	Externalize empty project type temp	The Empty Project pr
3	?	229762	All	?	?	Kein Ergebnis.	Kein Ergebnis.
4	?	229887	Linux	Platform	UI	Hier soll es keinen Treffer geben.	?
5	?	234668	Windows XP	GMF	Models - Graphical	Support SVG figures in gmfgraph r	?
6	230490	230490	All	IMP	LPG IDE	Doesn't work on Linux	The generator can't fi
7	230491	230491	All	IMP	LPG IDE	Doesn't work on Linux	The generator can't fi
8	230816	230816	Windows XP	ACTF	AI	bug_1	first bu
9	233112	233112	Windows XP	ACTF	AI	test	tes
10	?	230816	Windows XP	ACTF	AI	test	first bu
11	?	229887	Linux	Platform	UI	[infer type arguments] The fix 'Add t	What steps will repro
12	232735	230067	All	JDT	UI	[infer type arguments] The fix 'Add t	What steps will repro

Abbildung 22: Ergebnisse des k-NN-Algorithmus in RapidMiner ¹⁷

In Abbildung 23 werden die Ergebnisse aus RapidMiner mit der Testdatei zusammengeführt. Korrekte Vorhersagen werden mit ‚MATCH‘ verdeutlicht. Gelb markiert sind die Zellen, in denen etwas erfunden bzw. im Vergleich zum Original-Bug verändert wurde.

¹⁷ Eigene Abbildung.

expected	predicted	Match?	bug_id	op_sys	prod_name	comp_name	short_desc	comments
234976	234976	MATCH	234976	All	Higgins	STS	Use Apache Common Logging - sts.binding.a	change the component to
234676	234676	MATCH	234676	Windows XP	CDT	cdt-core	Externalize empty project type template	The Empty Project project
-	229762	-		All			Kein Ergebnis.	Kein Ergebnis.
			229762	All	TPTP Release	TPTP.Web	Resolve broken links (404s) on the build site.	Resolve broken links (404s)
-	229887	-		Linux	Platform	UI	Hier soll es keinen Treffer geben.	
			229887	Linux	Platform	UI	[MPE] MPE example disposes font too soon	I20080430-0100- observe
234668	234668	MATCH		Windows XP	GMF	Models - Graphical	Support SVG figures in gmfgraph models	
230490	230490	MATCH	230490	All	IMP	LPG IDE	Doesn't work on Linux	The generator can't find t
230491	230491	MATCH	230491	All	IMP	LPG IDE	Doesn't work on Linux	The generator can't find t
230816	230816	MATCH	230816	Windows XP	ACTF	AI	bug_1	first bugHi.Please let us kr
233112	233112	MATCH	233112	Windows XP	ACTF	AI	test	testnot a bug report.close
230816	230816	MATCH		Windows XP	ACTF	AI	test	first bugHi.Please let us kr
232735	229887	-		Linux	Platform	UI	[infer type arguments] The fix 'Add type arg	What steps will reproduce
			229887	Linux	Platform	UI	[MPE] MPE example disposes font too soon	I20080430-0100- observe
232735	230067	-	232735	All	JDT	UI	[infer type arguments] The fix 'Add type arg	What steps will reproduce
			230067	All	JDT	UI	[organize imports] Organize imports adds us	Sample, class:----package

Abbildung 23: Auswertung der Ergebnisse des k-NN-Algorithmus¹⁸

Der k-NN-Algorithmus trifft bei acht von zwölf Beispielen eine korrekte Vorhersage, was einer Trefferquote von 67 % entspricht.

Allerdings kann mithilfe der Auswertung ein Einblick gewonnen werden, wie der Algorithmus bei den falschen Prognosen zu seinem Ergebnis kommt. Bei den grau hinterlegten Zellen handelt es sich um die Daten des Bugs, der als Treffer angegeben wird. Die hellblau umrandeten Zellen zeigen, wo es eine Übereinstimmung zwischen den Testangaben und dem gefundenen Bug gibt.

Auffällig ist, dass das Betriebssystem sowie auch der Produkt- und Komponentename einen hohen Einfluss zu haben scheinen, die Textfelder hingegen weniger. Dieses zeigt klar eine Schwachstelle des Verfahrens, nämlich die fehlende Möglichkeit, die Gewichtung der Attribute vornehmen zu können. Wäre es möglich, den Feldern *short_desc* und *comments* eine größere Bedeutung beikommen zu lassen, sollte die Quote der korrekten Vorhersagen noch deutlich zu steigern sein.

Eine weitere Schwäche dieser Methode ist die Tatsache, dass jedem Eintrag zwanghaft ein Ergebnis zugewiesen wird. Dieses geschieht auch dann, wenn es sich um einen neuen Eintrag handelt und sich bisher kein ähnlicher in der Datenbank befindet. Folglich muss weiterhin manuell überprüft werden, ob der Treffer in der Tat Ähnlichkeiten mit dem neuen Eintrag aufweist.

¹⁸ Eigene Abbildung.

Ebenso kann es problematisch sein, dass bisher nur eine Dublette angegeben wird. Indem k erhöht wird, ist es möglich, mehr Ergebnisse zu erzwingen. Allerdings führt dieses erneut dazu, dass Resultate angezeigt werden, die nicht ideal sind.

Letztendlich lässt sich festhalten, dass die k -NN-Methode zwar zu den besten Ergebnissen führt, aber nach wie vor viele manuelle Überprüfungen notwendig sind.

6 Deployment

Dieses Projekt hat gezeigt, dass nicht ohne Grund zahlreiche Dubletten in der zur Verfügung gestellten Datenbank des Eclipse Bugtrackers vorhanden sind. Eine vollautomatisierte Suche nach Übereinstimmungen zwischen neuen und existierenden Einträgen ist nach aktuellem Kenntnisstand nicht oder zumindest nicht ohne Weiteres zu realisieren.

Trotzdem bieten moderne Text-Mining-Verfahren zahlreiche Möglichkeiten, den manuellen Überprüfungsvorgang zu vereinfachen. Von den hier vorgestellten Methoden verspricht das k -NN-Verfahren die besten Ergebnisse in der Praxis. Wäre es möglich, die einzelnen Attribute unterschiedlich stark zu gewichten, könnte der Fokus stärker auf die textuellen Inhalte der Bugs gelegt werden, welche zur Erkennung einer Dublette in der Praxis deutlich wichtiger sind als Betriebssystem, Produkt und Komponente. Darüber hinaus wäre es sinnvoll, einen Schwellenwert einzurichten, ab wann ein Bug als Dublette markiert werden soll. Im Idealfall würde so das zwanghafte Zuweisen eines Ergebnisses bei dem k -NN-Verfahren umgangen werden.

Nicht außer Acht zu lassen ist allerdings, dass für die Durchführung dieses Projektes nur ein kleiner Teil der zur Verfügung stehenden Daten verwendet wurde. Dieses liegt daran, dass herkömmliche, private Computer leistungstechnisch nicht in der Lage sind, die ausgewählten Verfahren auf einen großen Datenbestand anzuwenden. Das führt letztendlich dazu, dass eine systemseitige Früherkennung von Dubletten gerade dann, wenn sie besonders gebraucht wird, nämlich bei einem sehr großen Datenvolumen, nicht ohne Schwierigkeiten angewandt werden kann.

7 Literatur

- Anvik, J., Hiew, L., Murphy, G. C. (2005): *Coping with an open bug repository*, in: Proceedings of the 2005 OOPSLA Workshop on Eclipse Technology eXchange, New York, ACM, 2005, s. 35-39.
- Bruch, Marcel (2014): *Bug 444551 – [egit] Internal Error on switching to a different branch (err_grp: 89d917ef)*, unter: https://bugs.eclipse.org/bugs/show_bug.cgi?id=444551 (abgerufen am 18.01.2016).
- Miner, G., Delen, D., Elder, E., Fast, A., Hill, T., Nisbet, R. A. (2012): *Practical Text Mining and Statistical Analysis for Non-structured Text Data Applications*, Oxford: Elsevier / Academic Press, 2012.
- RapidMiner (2016): *Naïve Bayes (RapidMiner Studio Core)*, unter: http://docs.rapidminer.com/studio/operators/modeling/predictive/bayesian/naive_bayes.html (abgerufen am 27.06.2016).

Automatische Zuordnung von Bugs zu Entwicklern mit Hilfe von Text Mining

Daniel Bender, Lukas Brenning, Henrik Kortum

Abstract. *Die Bearbeitung von Bugs der Eclipse Foundation gestaltet sich derzeit sehr aufwändig, da Entwickler die von ihnen präferierten Bugs selbst finden und bearbeiten müssen. Um den Suchvorgang nach interessanten Bugs zu erleichtern wird in dieser Arbeit aufgezeigt, wie mit Hilfe von Text Mining Klassen von Bugs und Entwicklern gebildet und diese dann so miteinander verknüpft werden, dass neu angelegte Bugs direkt den entsprechenden Entwicklern zugeordnet werden. Mit diesem Ansatz soll eine effizientere und qualitativ hochwertigere Bearbeitung von Bugs ermöglicht werden.*

1 Business Understanding

Die Eclipse Foundation ist eine non-profit Organisation, die ihren Nutzern verschiedene Services zur Verfügung stellt. Unter anderem eine IT-Infrastruktur, IP Management und die Unterstützung von Entwicklern. Zu diesen Services gehört auch die Datenbank “Bugzilla”, in der Bugs gemeldet und bearbeitet werden können. Um die Optimierung der Bug-Bearbeitung geht es in diesem Projekt, das im Rahmen des KI-Praktikums durchgeführt wurde (The Eclipse Foundation, 2016). Die Idee, die Verteilung von Bugs innerhalb von Eclipse effizienter zu gestalten, entstand durch den Artikel von Alenezi und Magel (2013), die ebenfalls Entwicklern bestimmte Bugs zuordnen möchten.

Derzeit müssen Entwickler, die sich der Lösung von Bugs widmen möchten, diese selbstständig suchen. Es gibt also keinerlei Empfehlungen, welche Bugs von welchem Entwickler bearbeitet werden sollen. In dieser Arbeit wird ein möglicher Ansatz und die Umsetzung beschrieben, wie mit Hilfe von KDD eine Empfehlung ausgesprochen werden kann und damit Bugs direkt denjenigen Entwicklern zugeordnet werden, bei denen die Wahrscheinlichkeit einer qualitativ hochwertigen Lösung am größten ist. Aktuell gibt es lediglich eine Rangliste mit den am häufigsten frequentierten Bugs.

Die Bearbeitung von Bugs ist also zum aktuellen Zeitpunkt aufgrund der großen Menge an gemeldeten Bugs sehr unübersichtlich und zeitintensiv. Durch die Klassifizierung von

Bugs und der jeweiligen Zuordnung der Entwickler wird eine effizientere Bearbeitung von Bugs angestrebt.

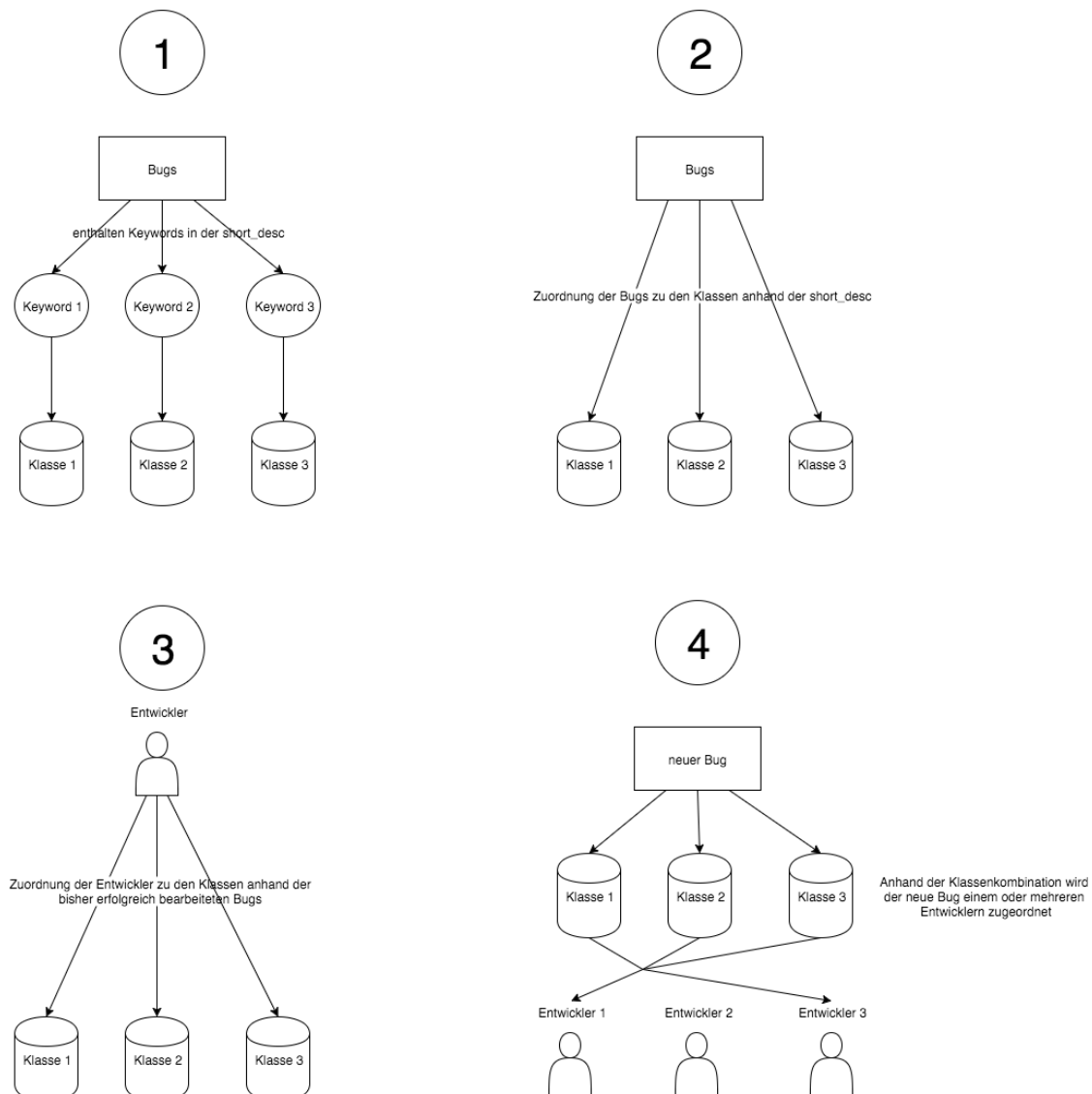


Abbildung 24: Vorgehen während des Projektes (eigene Darstellung)

Abbildung 24 zeigt das Vorgehen während des Projektes, der erste Schritt ist Klassen anhand der Keywords zu erstellen, die die Bugs in ihrer Kurzbeschreibung enthalten. Danach werden die Bugs in die neu erstellten Klassen eingeteilt. Das gleiche geschieht in Phase drei mit den Entwicklern, sie werden ebenfalls in die in Schritt eins erstellten Klassen eingeteilt. Die Entwickler werden anhand der Keywords der von ihnen erfolgreich bearbeiteten Bugs in die Klassen eingeteilt. Schritt vier umfasst die Zuordnung der neuen Bugs zu den am besten geeigneten Entwicklern. Dazu wird überprüft, welcher Entwickler mit dem neu angekommenen Bug die meiste Übereinstimmung hinsichtlich der

Keywords hat. Das Ziel ist, dass die am besten qualifizierten Entwickler direkt benachrichtigt werden, wenn ein Bug, der von ihnen bearbeitet werden soll, eintrifft. Damit soll die erwünschte Effizienzsteigerung umgesetzt werden.

Das KI-Praktikum wurde nach dem “Cross Industry Standard Process for Data Mining” (CRISP-DM) durchgeführt und damit in sechs Phasen aufgeteilt, die hier nacheinander bearbeitet werden. Zu diesen sechs Phasen gehören “Business Understanding”, “Data Understanding”, “Data Preparation”, “Modeling”, “Evaluation” und “Deployment” (Chapman et al., 2000, p. 10).

2 Data Understanding

Im Folgenden wird der Prozess des Data Understanding durchgeführt. Voraussetzung für valide Ergebnisse im Data Mining sind eine ausreichend große **Datenmenge** und eine **Datenqualität**.

Bei dem verwendeten Datensatz handelt es sich um einen Auszug aus der Datenbank des Eclipse Bugtrackers. Er umfasst Eintragungen im Zeitraum von Anfang Oktober 2001 bis ca. Ende Juni 2010.

Eine erste Analyse des Datensatzes erfolgte unter Verwendung des Datenbank-Modellierungswerkzeuges „MySQL Workbench“. Hierbei wurde zunächst das Kriterium einer ausreichend großen Datenbasis überprüft. Der Datensatz umfasst 18 Tabellen mit insgesamt 217 Attributen und 316.911 eingetragenen Bugs.

Der zugrundeliegende Datensatz weist ein ausreichend großes Volumen auf und ist somit für eine Text Mining-Analyse geeignet.

Anschließend wurde der Datensatz im Hinblick auf inhaltliche Aspekte näher betrachtet, hierzu wurde eine logische Gruppierung der Tabellen vorgenommen. Als Ergebnis wurden sieben Inhalts-Gruppen gebildet, die als Basis der Zuordnung der einzelnen Tabellen diente. Die o.g. Zuordnung ist im Folgenden Dargestellt.

Tabelle 3: Inhaltsgruppe „Bugs“

Tabelle	Kurzbeschreibung
Bugs	Zentrale Tabelle mit allen Bugs und relevanten Attributen

Tabelle 4: Inhaltsgruppe „Keywords“

Tabelle	Kurzbeschreibung
Keywords	Zuordnung von Keywords zu Bugs
Keywordsdef	Definition der Keywords und Kurzbeschreibung

Tabelle 5: Inhaltsgruppe „Historie der Änderung“

Tabelle	Kurzbeschreibung
Bugs_activity	Änderungshistorie: Zeitpunkt der Änderung und Inhalt der Änderung
fielddef	Definition von Feldern und Kurzbeschreibung

Tabelle 6: Inhaltsgruppe „Informationen über Anhänge“

Tabelle	Kurzbeschreibung
attachments	Beschreibungen, Dateityp und Dateinamen von Anhängen zu Bugs

Tabelle 7: Inhaltsgruppe „Produkte und Komponenten“

Tabelle	Kurzbeschreibung
Products	Tabelle mit allen Produkten der Eclipse Foundation und kurzer Produktbeschreibung
Components	Kurzbeschreibung der jeweiligen Produktkomponenten
Components_cc	Zuordnung von Benutzern zu Komponenten
Versions	Versionsnummern der Produkte
Milestones	Meilensteine der Produkte

Tabelle 8: Inhaltsgruppe „Volltexte“

Tabelle	Kurzbeschreibung
Longdescs	Alle Kommentare zu Bugs, inklusive der Bug-Beschreibung

Tabelle 9: Inhaltsgruppe „Weitere Informationen zu Bugs“

Tabelle	Kurzbeschreibung
dependencies	Abhängigkeiten in der Reihenfolge der Bugbearbeitung
duplicates	Dubletten von Bugs
cc	Benachrichtigung von Benutzern bei Änderungen des Bugs
votes	Anzahl der Votes pro Bug
bugs_fulltext	Titel des Bugs (wie in bugs-Tabelle) sowie die ausführliche Beschreibung des Bugs
bug_see_also	Verweise auf andere Bugs per Link

Nachdem die inhaltlichen Gruppen festgelegt wurden, erfolgte eine Auswahl derjenigen Einträge, welche für das in Kapitel 1 definierte Businessziel, der Kategorisierung von Bugs, relevant erschienen.

Hierbei konnten sieben relevante Attribute identifiziert werden. Zwei dienen potenziell als Labelattribut, eines zur Einschränkung des Analysebereichs und vier als potenzielle Ziele für den Text Miningprozess.

In der folgenden Tabelle sind die o. g. Attribute, die zugehörige Tabelle, eine kurze inhaltliche Beschreibung, sowie der jeweilige Nutzen für die Analyse aufgeführt.

Tabelle 10: Nutzen und Beschreibungen der Tabellen

Tabelle.Attribut	Beschreibung	Nutzen für Analyse
bugs.bug_id	ID des gemeldeten Bugs	Mögliches Labelattribut: ermöglicht Text Mining pro Bug
bugs.assigned_to	ID des mit Bugfixing betrauten Entwicklers	Mögliches Labelattribut: ermöglicht Text Mining pro Entwickler
bugs.product_id	ID des vom Bug betroffenen Softwareproduktes	Einschränkung des Analysebereichs
bugs.short_desc	Kurze Volltextbeschreibung des aufgetretenen Fehlers durch den meldenden User	Text Mining
bugs_fulltext.comments	Volltextkommentare zum gemeldeten Bug, durch User	Text Mining

bugs.keywords	Vorgegebene, die Art des Bugs beschreibende Keywords	Text Mining
longdescs.thetext	Alle zu einem bestimmten Bug verfassten Texte (Beschreibung + Kommentare)	Text Mining

Anschließend wurde eine Überprüfung der Datenqualität durchgeführt. In der Literatur werden verschiedene Kriterien zur Beurteilung von Datenqualität diskutiert. Im Folgenden wurde die Datenqualität auf Basis der Kriterien analysiert: Vollständigkeit, Redundanzfreiheit, Relevanz und Genauigkeit (Bayer, 2011). Da die Datenqualität der einzelnen Attribute teilweise stark voneinander abweicht wurde an dieser Stelle darauf verzichtet sie im Allgemeinen zu bewerten. Stattdessen findet eine Analyse der Datenqualität auf Ebene der o. g. als potenziell relevant eingestuften Attribute statt. Zur Redundanzfreiheit kann allgemein gesagt werden, dass der Datensatz Dubletten aufweist und das Kriterium somit nicht erfüllt ist. Allerdings hat diese Tatsache auf die Entwicklung eines Klassifizierungsverfahrens nur bedingt Einfluss, sodass es an dieser Stelle nicht weiter berücksichtigt wird.

Tabelle 11: Analyse der Datenqualität der Tabellen

Tabelle.Attribut	Vollständigkeit	Genauigkeit	Relevanz
bugs.bug_id	Hoch, jeder Bug hat eine ID	Hoch, exakte Zuordnung möglich	Hoch, zentrales Element der Zuordnung
bugs.assigned_to	Hoch, jeder Bug ist genau einem Entwickler zugeordnet	Hoch, exakte Zuordnung möglich	Hoch, zentrales Element der Zuordnung
bugs.product_id	Hoch, jeder Bug ist einem der 140 Produkte zugeordnet	Hoch, exakte Zuordnung möglich	Hoch, zentrales Element der Zuordnung
bugs.short_desc	Hoch, nur in drei Fällen kein Eintrag	Mittel, teilweise ungenaue Beschreibung des Bugs	Hoch, Beschreibung des Bugs ist Grundlage für Text Mining
bugs_full-text.comments	Hoch, nahezu jeder Bug hat Kommentare	Mittel, teilweise Beschreibungen zu ungenau um Klassifizierung zu ermöglichen	Mittel, teilweise beziehen sich Einträge nicht auf den gemeldeten Bug

bugs.keywords	Gering, nur in 7% der Einträge angegeben	Gering, nur geringe Anzahl an Keywords (27)	Gering, Keywords passen teilweise nicht zum Problem
longdescs.thetext	Hoch, nur in drei Fällen kein Eintrag	Mittel, teilweise Beschreibungen zu ungenau um Klassifizierung zu ermöglichen	Mittel, teilweise beziehen sich Einträge nicht auf den gemeldeten Bug

Abschließend kann festgestellt werden, dass sich die Attribute *bug_id*, *assigned_to*, und *product_id* im Hinblick auf die untersuchten Kriterien der Datenqualität sehr gut für das Text Mining eignen. Alle drei Kriterien werden als Selektionsmechanismus, bzw. Labelattribut bei der Modellerstellung berücksichtigt.

Als Ziel für die Text Mininganalyse eignet sich das Attribut *short_desc* am besten, es weist eine hohe Vollständigkeit, bei einer gleichzeitig hohen Relevanz auf, was den Vorteil gegenüber den anderen Attributen darstellt.

Die Attribute *comments*, *keywords* und *thetext* werden aufgrund mangelnder Relevanz und Genauigkeit im weiteren Prozess nicht berücksichtigt.

3 Data Preparation

Da es sich jeweils um einen großen Prozess handelt, der einige Unterebenen besitzt, wird der Abschnitt Data Preparation wegen der besseren Nachvollziehbarkeit in drei gedankliche Bereiche aufgeteilt. Damit sollte ein besseres Verständnis für die erstellten Prozesse erreicht werden.

In- und Output des Prozesses

Nachdem im Data Understanding die Attribute für die Selektionsmechanismen und Text Mininganalyse definiert worden sind, folgt als nächster Schritt die Aufbereitung der Daten. Dabei sollen am Ende der Bearbeitung zwei Zielformate in Excel-Format entstehen, die weiter beim Modeling verwendet werden können.

Tabelle 12: User_ID bezogene Daten

user	Keyword
-------------	----------------

User_id	Keyword1, Keyword2, Keyword3, ... , Keyword n
---------	---

Tabelle 13: Bug_ID bezogene Daten

bug	Keyword
Bug_id	Keyword1, Keyword2, Keyword3, ... , Keyword n

Um dieses Format zu erreichen müssen als erster Schritt die dafür benötigten Daten geladen werden. Dies geschieht gleich zu Anfang in der *Prozessebene 1* (siehe Abbildung 26). Die Daten werden aus der Datenbank ausgelesen und anhand einer SQL-Query (siehe Abbildung 27) vorgefiltert. Am Beispiel der User_ID bezogenen Daten werden nur drei Spalten aus dem ganzen Datensatz von *Bugs* benötigt und auch nur die, denn Status *RESOLVED* besitzen. Aus Gründen des schnelleren Testens wurde eine Einschränkung eingeführt, indem nur Datensätze von dem Produkt mit der *product_id=4* aufgerufen werden.

```
- SQL Query -----
select assigned_to, keywords, short_desc from bugs
where bug_status = 'RESOLVED'
and product_id=4
```

Abbildung 25: SQL-Query (Screenshot)

Preparation um den Hauptprozess

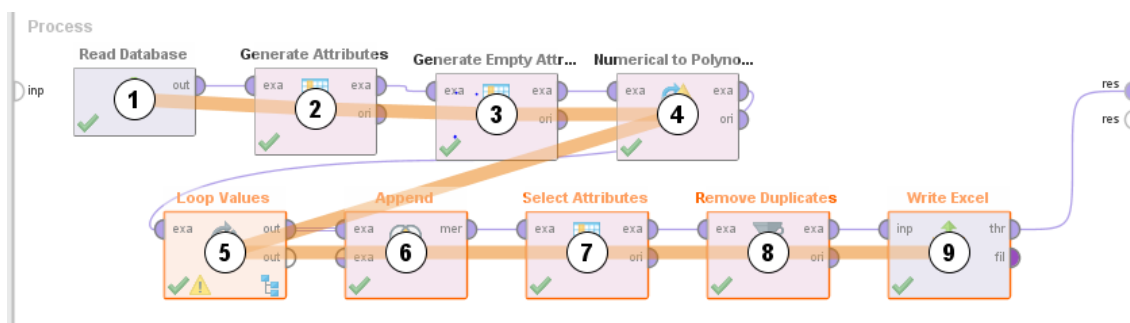


Abbildung 26: Prozessebene 1 (Screenshot)

Nach dem die Daten eingelesen werden, beginnt in RapidMiner der Bearbeitungsprozess. Vorab sei gesagt, dass der Hauptprozesse sich in dem Operator *Loop Value* als fünfter Schritt befindet. Vor dem Hauptprozess wird durch die Operatoren im Schritt zwei und

drei die Spaltenbezeichnung *assigned_to* zu „User“ geändert und es wird ein neues Attribut *keyword* erstellt, in den später die Keywords nach dem Text Mining eingefügt werden. Damit die Daten bei Schritt fünf in einer Schleife weiterbearbeitet werden können, wird bei *user* der Typ von numerical zu polynomial verändert. Hinter dem Hauptprozess werden die verarbeiteten Daten gebündelt, gefiltert und in einer Excel-Datei gespeichert.

Hauptprozess

Der vorliegende Hauptprozess (siehe Abbildung 27) beinhaltet zwei essentielle Funktionen. Die erste Funktion beschäftigt sich mit dem Text Mining und die zweite mit dem Problem der gewünschten Zieldarstellung des Outputs.

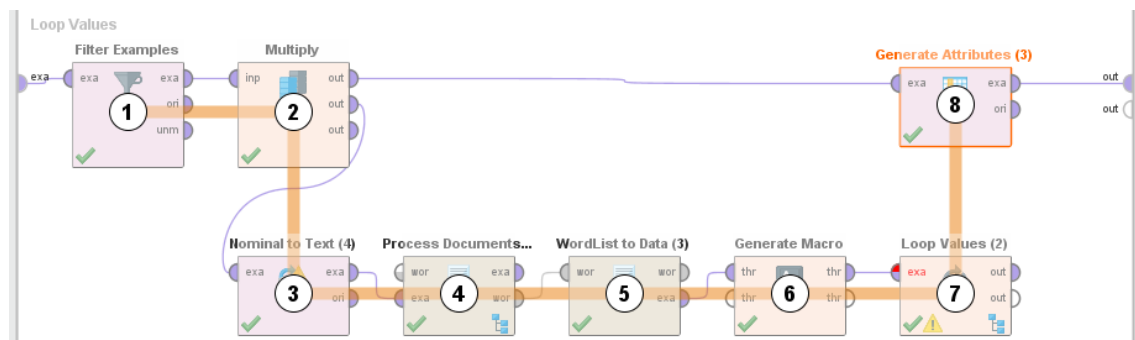


Abbildung 27: Prozessebene 2 (Screenshot)

Für die erste Funktion wird die *short_desc* bearbeitet. Dabei wird das Format in Text umgewandelt, damit im nächsten Operator *Prozess Documents* das eigentliche Text Mining durchgeführt werden kann. Wie in Abbildung 28 zuerkennen, werden auf den Text fünf verschiedene Operatoren angewandt:

- *Transorm Cases*: Darstellung des ganzen Texts in Kleinbuchstaben
- *Operator Tokenize*: Teilt den Text eines Dokuments in eine Folge von Zeichen.
- *Filter Stopword*: Filtert nach einer Stoppwörter-Liste die englischen Stoppwörter aus einem Dokument
- *Filter Tokens*: Filtert die Zeichen in Abhängigkeit von der Länge (min chars =4 und max chars= 25)
- *Stem*: Kürzt die engl. Wörter auf den Wortstamm

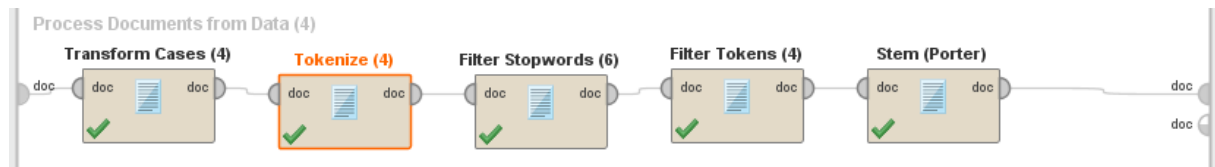


Abbildung 28: Übersicht der Funktion Process Documents (Screenshot)

Würden die Daten nach dem Durchführen des Text Mining ausgewertet werden, dann würde sich als Ergebnis eine Rangordnung der Wörter vorfinden. An dieser Stelle benötigt man die zweite Funktion mit den zwei Schleifen. Wie schon beschrieben befindet sich in der ersten der Text Miningprozess. In der ersten Schleife wird über *assigned_to* iteriert. Dies hat zur Folge, dass die erarbeiteten Keywords aus dem Text Mining zu einem user zugeordnet werden. Allerdings besteht das Problem, dass jedes Keyword in eine neue Zeile in der Spalte hinzugefügt wird. Für unsere Zieldarstellung müssen die keywords in eine Zelle der Spalte keyword aufgereiht werden. Diese Aufgabe übernimmt die zweite Schleife. In dieser werden die einzelnen Wörter nach dem Text Mining für einen user konkatinert und am Ende der zweiten Schleife an die erste übergeben. Damit wird das gewünschte Format erreicht (siehe Abbildung 29). Es liegen für jeden user die Keywords der von ihm bearbeiteten Bugs vor.

Row No.	user	keyword
1	448	alphanumeric, assist, chang, charact, clariti, code, command, condit, creat, creation, databas, declar, distinct, distinguish, doc...
2	455	action, alias, allow, architectur, attribut, behav, breakpoint, brows, build, button, carriag, close, column, command, commun, ...
3	460	activ, address, allow, applic, attach, break, breakpoint, brows, catch, chang, command, condit, configur, consol, daemon, de...
4	454	actio, applic, attach, caus, config, debugg, detach, dialog, emac, empti, enter, except, execut, fail, handl, incomplet, launch, l...
5	457	action, autoconf, autoconfmanag, autogen, automak, autoscan, built, case, clean, colon, command, configur, confirm, creat, ...
6	1906	abil, abl, abort, access, accumul, action, ad, adaptor, addinclud, addr, address, affect, allow, annot, anonym, anymor, appea...
7	5586	accept, accord, aco, ad, address, alia, alias, alignof, allow, altern, ambigu, annot, anonym, ansi, appear, appli, arg, argumen...
8	462	broken, browser, bugzilla, click, control, deselect, doc, doubl, effect, enter, extern, finish, gnome, help, info, inform, item, link, ...
9	456	action, ad, allow, arc, call, choos, filter, function, functioncheck, locat, multithread, prefer, program, properti, sourc, statist, su...

Abbildung 29: Ergebnis aus dem RapidMiner-Prozess (Screenshot)

Genau der gleiche Prozess wird auch für die Herstellung der Bug_ID bezogenen Daten benutzt, allerdings mit gewissen Anpassungen bzgl. der Attribute.

4 Modeling

Der Ansatz der ursprünglichen Idee bestand darin, dass aus den zwei gewonnenen Tabellen eine Klassifizierung von einem User zu einer bestimmten Bug-Sorte durch das Anwenden des Neuronalen Netzes erreicht werden könnte. Der Prozess wurde in RapidMiner (siehe Abbildung 30) erstellt und durchgeführt. Dabei wurden zuerst die User_ID bezogenen Daten an das Neuronale Netz zum Trainieren übergeben und das Ergebnis des Trainings sollten dann auf dem Bug_ID bezogenen Daten angewandt werden. Leider ließ sich dies aus syntaktischen Gründen nicht in RapidMiner umsetzen. Die beiden eingegebenen Tabellen müssen bei dieser RapidMiner-Funktion die gleiche Formatierung besitzen, was sich hier nicht umsetzen ließ.

Im zweiten Versuch wurden den Bug_ID bezogenen Daten auch noch die User_ID hinzugefügt. Dieses führte soweit zum Erfolg, dass der Prozess ohne Probleme durchlief und ein Ergebnis lieferte. Die Vorhersage bzgl. eines User zu einem Bug geschah jedoch nur mangelhaft. Als Vorhersagen entstanden kumulierte Durchschnittswerte der User_ID.

Dadurch wurden User_IDs vorhergesagt, die gar nicht existieren.

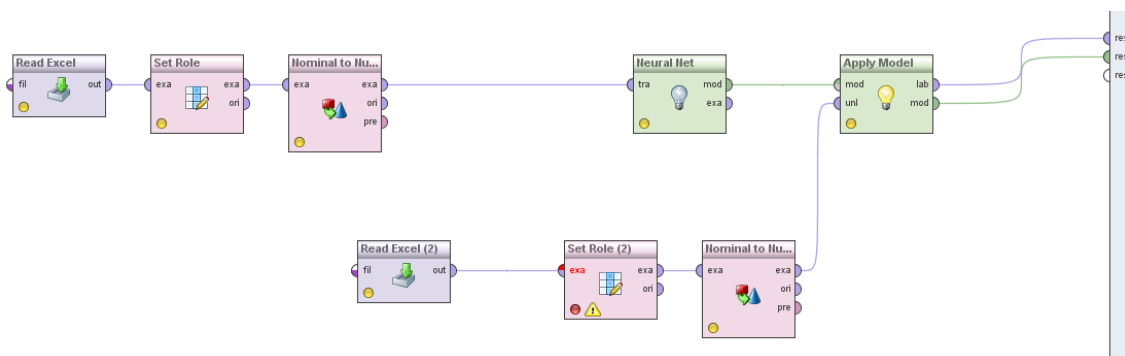


Abbildung 30: KNN Prozess (Screenshot)

Nachdem der Ansatz mit dem KNN nicht das gewünschte Ergebnis hervorgebracht hat, bedienen wir uns der Methode des Clustering. Als Clustering wird die Herangehensweise bezeichnet, die zur Erkennung von Ähnlichkeitsstrukturen in großen Datensätzen durchgeführt. Diese Methode ist ebenfalls im RapidMiner durchführbar. Nach dem der Clusteringprozess (siehe Abbildung 31) erstellt wurde, entstand das Ergebnis aus Abb. 8. Beim Betrachten des Ergebnisses ist zu erkennen, dass fünf Cluster durch die Voreinstellung gebildet wurden. Das Label bei diesem waren die User_IDs, da diese den Clustern zugeordnet werden sollten. Als nächstes erfolgte die Interpretation des Ergebnisses, dabei

sind die Werte gut, die nicht null sind und so nah wie möglich an dem Wert eins liegen. Insgesamt ist zu erkennen, dass nur 4 Werte vorhanden sind (0, 0.333, 0.5, 1). Diese Werte sagen aus, aus wie stark die User den bestimmten Attributen in denen zugeordneten Cluster passen. Aus diesen Werten lassen sich jedoch keine eindeutigen Cluster bilden.

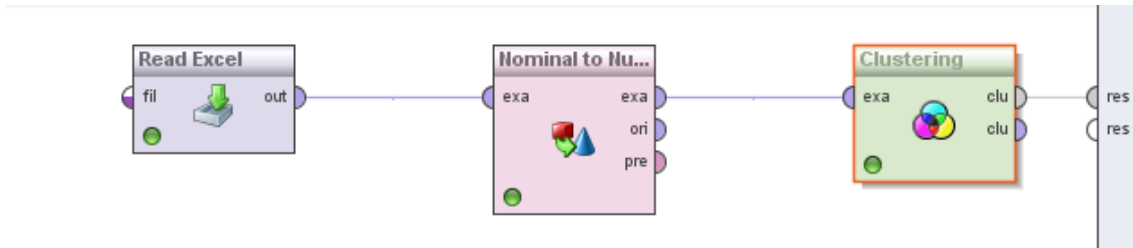


Abbildung 31: Clusteringprozess (Screenshot)

Attribute	cluster_0	cluster_1	cluster_2	cluster_3	cluster_4
keyword = alphanumer, assist, chang, charact, clariti, code, command, condit, creat, creation, databas, declar, d	1	0	0	0	0
keyword = action, alias, allow, architectur, attribut, behav, breakpoint, brows, build, button, carriag, close, column	0	0.333	0	0	0
keyword = activ, address, allow, applic, attach, break, breakpoint, brows, catch, chang, command, condit, configu	0	0	0	0	0.500
keyword = actio, applic, attach, caus, config, debugg, detach, dialog, emac, empti, enter, except, execut, fail, han	0	0.333	0	0	0
keyword = action, autoconf, autoconfmanag, autogen, automak, autoscan, built, case, clean, colon, command, c	0	0.333	0	0	0
keyword = abil, abl, abort, access, accumul, action, ad, adaptor, addinclud, addr, address, affect, allow, annot, ar	0	0	1	0	0
keyword = accept, accord, aco, ad, address, alia, alias, alignof, allow, altern, ambigu, annot, anonym, ansi, appe	0	0	0	1	0
keyword = broken, browser, bugzilla, click, control, deselect, doc, doubl, effect, enter, extern, finish, gnome, help,	0	0	0	0	0.500
user	448	455.333	1906	5586	461

Abbildung 32: Ergebnis aus dem Clusterprozess (Screenshot)

Nachdem sich aus den zwei vorangegangenen Methoden nicht das gewünschte Matching-konzept erarbeiten ließ, formulierten wir in der Gruppe ein grobes Matching so wie wir uns das Vorgelegt haben. Das Vorgehen lässt sich Anhand der Abb. 10 erklären. Beispielhaft wird ein neuer Bug eingegeben, dem nach dem Text Mining die Keywords (browse, commande, create und help) zugeordnet werden. Pro Keyword wird im nächsten Schritt ein Flag mit den Werten 0 (kein match) und 1 (match) für jeden User erstellt.

So wird am Ende ein Durchschnittswert errechnet für die Wörter, bei denen ein Match zustande gekommen ist. In dem Beispiel für den User 448 würde der Wert bei 0,5 liegen und bei dem User 455 bei 0,75. Da der User 445 einen höheren Wert der Übereinstimmung kriegt, wird diesem User der Bug als Vorschlag vermittelt.

Diese Vorgehensweise konnte lediglich manuell durchgeführt werden, weil keine Rapid-Miner Operatoren zur Verfügung stehen, die dieses durchführen konnten.

	A	B	C	D	E	F	G	H	I	J	K	L	M
1	user	keyword	keyword	keyword	keyword	keyword	keyword	keyword	keyword	keyword	keyword	keyword	keyword
2	448	alphanum	assist	changing	character	clarity	code	command	condition	created	database	declarati	distinct
3	455	aliases	architect	breakpoi	browse	button	carriage	closed	command	command	communi	connect	conrit
4	460	active	address	addresse	allow	break	breakpoi	breakpoi	catch	change	command	command	condit
5	454	actio	applicati	debugger	detach	emacs	handling	launch	process	projects	start	view	
6	457	action	actions	autoconf	automak	built	case	clean	colon	command	command	configure	confir
7	1906	cursor	emacs	indentati	mode	movement							
8	5586	directives	pragma										
9	462	browser	control	gnome	help	informati	remote	using					
10													
11	nB	browse	command	created	help								
12	448	0	1	1	0	0,5							
13	455	1	1	1	0	0,75							
14	460	0	1	0	0	0,25							
15	454	0	0	0	0	0							
16	457	0	1	0	0	0,25							
17	1906	0	0	0	0	0							
18	5586	0	0	0	0	0							
19	462	0	0	0	1	0,25							
20													

Abbildung 33: Manuelles Matching (Screenshot)

5 Evaluation

5.1 Evaluation der Ergebnisse

Im folgenden Abschnitt soll auf die Ergebnisse aus den vorherigen Kapiteln eingegangen werden.

Zunächst werden die Prozesse „*User_Keyword preparation*“ und „*Bug_Keyword preparation*“ im Hinblick auf das eingangs formulierte Businessziel: „*Die Klassifizierung von Bugs und Entwicklern soll eine effiziente Zuordnung von Entwicklern zu Bugs ermöglichen und so eine optimierte Bearbeitung sicherstellen*“, betrachtet. Zur Erreichung dieses Ziels wurde in Kapitel 1 ein vier stufiger Projektplan vorgestellt.

1. Finden von Keywords auf Basis der „*short_desc*“
2. Klassifizierung von Bugs auf Basis von Keywords in der Beschreibung „*short_desc*“
3. Klassifizierung von Entwicklern auf Basis der Historie der erfolgreich bearbeiteten Bugs
4. Prediction: Wenn neuer Bug der Kategorie x reported wird, werden Entwickler der Kategorie x benachrichtigt

Ergebnis der beiden Prozesse ist jeweils eine Excel-Datei, welche jedem Benutzer und jedem Bug, Keywords auf Basis von Text Mining der Shortdescription zuordnet.

Die ersten drei Stufen des Projektplans konnten erfolgreich umgesetzt werden.

Zur Erreichung des letzten Teilziels, einer automatischen Zuordnung von neuen Bugs zu geeigneten Nutzern, wurden im vorherigen Kapitel drei Ansätze vorgestellt, die nun evaluiert werden sollen.

Künstliches Neuronales Netz:

Als Ergebnis wurde hier ein Durchschnittswert über alle User_ID's ausgegeben.

Der KNN-Prozess scheint daher für eine Zuordnung ungeeignet.

Clustering:

Der Prozess wurde mehrfach mit einer wechselnden Anzahl an zu erstellenden Clustern durchgeführt. Eine Analyse der gebildeten Cluster lies dabei keine logischen Zusammenhänge von Usern und zugeordneten Bugs erkennen.

Der Cluster-Prozess scheint somit für eine Zuordnung ebenfalls nicht geeignet.

Matching:

Abschließend wurde in Excel ein einfacher Matching-Prozess modelliert.

Hierbei wird die Ähnlichkeit zwischen den Keywords eines neu gemeldeten Bugs und den Keywords aller Entwickler ermittelt. Ein Entwickler, der 8 von 10 Keywords eines neuen Bugs abdeckt, wäre folglich zu 80% geeignet den Bug zu bearbeiten.

Ein Matching-Prozess scheint für eine Zuordnung von Bugs zu Entwicklern am besten geeignet.

5.2 Kritische Würdigung und Limitation

Abschließend soll das Vorgehen kritisch hinterfragt und Limitationen aufgezeigt werden.

Zunächst ist festzustellen, dass an vielen Stellen im Prozess eine Zusammenarbeit mit Domänenexperten notwendig erscheint. So würde es sich anbieten die Stoplist mit Hilfe eines Experten zu erstellen um eine adäquate Qualität der ermittelten Keywords sicherzustellen.

Im Hinblick auf die Ergebnisqualität sollte mit Hilfe eines Domänenexperten außerdem überprüft werden, ob die Verwendung des Stemming-Operators einen positiven oder negativen Einfluss auf die ermittelten Keywords ausübt.

Der vorgestellte Matching-Prozess ist sehr rudimentär und sollte für eine Implementierung verfeinert werden. Ein wesentliches Problem der jetzigen Ausgestaltung des Prozesses stellt die Tatsache dar, dass Usern mit einer sehr hohen Anzahl an bearbeiteten Bugs auf lange Sicht gesehen, alle Keywords zugeordnet werden. Somit wäre die Eignung dieser User zur Bearbeitung von einem beliebigen neuen Bug immer 100%. Um diesem Problem vorzubeugen bietet es sich an einen Gewichtungsmechanismus und Schwellenwerte zu verwenden.

Beispielsweise würde ein User der nur einen Bug mit dem Keyword „Browser“ bearbeitet hat, nicht das Keyword „Browser“ zugeordnet werden, da er einen Schwellenwert von z.B. min 3 bearbeiteten Bugs mit diesem Keyword nicht übersteigt. Ebenso wäre ein User mit 8 bearbeiteten Bugs zum Keyword „Browser“ für die Bearbeitung besser geeignet, als ein User der lediglich 6 Bugs mit dem entsprechenden Keyword bearbeitet hat.

Eine weitere Limitation stellt die Tatsache dar, dass der Matching-Prozess nicht in RapidMiner integriert und automatisiert werden konnte. Für eine Implementierung der Lösung, sollte aus Konsistenzgründen an einer integrierten Lösung gearbeitet werden.

6 Deployment

Die sechste Phase des CRISP-DM befasst sich mit dem Deployment, also der Umsetzung der Idee. Um die Idee der automatischen Bugzuweisung umzusetzen, gibt es zwei verschiedene Möglichkeiten. Zum einen eine aktive Hilfe, z.B. mit Hilfe von Nachrichten, die dem Entwickler gesandt werden, wenn ein neuer für ihn relevanter Bug angelegt wurde. Zum anderen eine passive Umsetzung, die in Abbildung 34 dargestellt ist. Diese Abbildung zeigt die aktuelle Startseite von Bugzilla, erweitert um eine weitere Funktion in der für den aktuell angemeldeten User empfehlenswerte Bugs dargestellt werden. Diese Darstellung wird in Abbildung 35 ergänzt, die genauer zeigt, wie die für den Nutzer empfohlenen Bugs dargestellt werden. Die Abbildung ist ein Screenshot der am häufigsten frequentierten Bugs, hier erweitert um eine Spalte, in der die Übereinstimmung der Ent-

wickler mit den jeweiligen Bugs angezeigt wird. Der Bug, der am meisten mit dem Entwickler übereinstimmt ist in der ersten Zeile und fällt dem Entwickler damit als erstes auf.



Abbildung 34: Mögliche Umsetzung der Bugzuweisung (1) (eigene Darstellung)

Bug #	Match with User Profile	Dups. Count	Change in last 30 day(s)	Component	Severity	OS	Target Milestone	Summary
444551	99%	41	0	Core	critical	All	---	[egit] Internal Error on switching to a different branch (err_grp: 89d917ef)
441727	90%	32	0	wst.xml	normal	Mac OS X	---	Could not retrieve EHandlerService or ICommandService from context evaluation context ...
446978	90%	26	0	Diagram	normal	All	---	[sirius.gmf] Impossible to find an interpreter - Could not find a session for model elemen... (err_grp: 3d703af6)
344861	90%	26	0	UI	normal	Windows 7	---	[EditorHgmt] Error setting focus to : org.eclipse.e4.ui.model.application.ui.basic.impi.PartImpl apache_style.css
447011	90%	25	0	UI	normal	All	---	[Trim] NPE in TrimPanelLayout.setCursor
445916	90%	24	0	cdt-core	normal	All	---	NoClassDefFoundError for org.eclipse.cdt/core/settings/model/uti/ResourceChangeHandlerBase\$DeltaVisitor in resources
3188	90%	24	0	Debug	normal	All	---	[breakpoints] Breakpoints from one project, hit in another project
76936	88%	20	0	Debug	enhancement	All	---	[console] Eclipse Console window does not handle '\b, \f, and \r
15941	88%	19	0	UI	enhancement	Windows 2000	---	[WorkingSets] New resources and working sets
3109	88%	19	0	Launcher	normal	All	---	Product won't start if directory contains '%# (LGRLOO)
451797	88%	18	0	UI	normal	All	---	[platform] UI freezes in IOConsolePartitioner\$TrimJob.runInUIThread
73957	86%	18	0	UI	major	All	---	Correct plug-in dependencies result in invalid compile errors (was incomplete type hierarchy)
45423	85%	18	0	Text	enhancement	All	---	[formatting] Separate presentation from formatting
347558	85%	17	0	Team	normal	Linux-GTK	---	IllegalArgumentException from structuremergeviewer.DiffNode.removeCompareInputChangeListener
23406	84%	17	0	SWT	major	Windows All	---	StyledText/TextLayout render certain long lines as empty (was: Long lines are not displayed by editor)
41353	83%	16	0	Debug	enhancement	All	---	[launching] Launch config templates
27740	83%	16	0	UI	normal	All	---	[extract local] must not ignore value changes
20006	82%	15	0	SWT	normal	Linux-GTK	---	X Pointer grab not released
450710	81%	14	0	UI	normal	All	---	[Decorators] [platform] Plenty of Invalid registry object errors in LightweightDecoratorManager.decorate
115202	76%	14	0	UI	enhancement	Windows XP	---	[UX] [Markers] [Commands] Next/Previous problem key binding

Abbildung 35: Mögliche Umsetzung der Bugzuweisung (2) (eigene Darstellung)

Um diese Möglichkeiten umzusetzen, bedarf es noch einer automatischen Berechnung der Übereinstimmung zwischen Entwickler und Bug. Die derzeitige Lösung in Excel ist für diese Arten von Deployment nicht geeignet und bedarf einer Überarbeitung. Des Weiteren muss die Umsetzung wie hier in den Abbildungen dargestellt noch in die Eclipse Homepage eingebunden werden. Das KI-Praktikum befasst sich allerdings lediglich mit

der Beschaffung bzw. Modellierung der benötigten Daten, nicht aber mit der endgültigen Umsetzung, da hierfür benötigte Zugriffsrechte auf die Homepage der Eclipse Foundation fehlen.

7 Literatur

Alenezi, M., & Magel, K. (2013). Efficient Bug Triaging Using Text Mining. *Academic Publisher*, 2185–2190.

Bayer, M. (2011). Gute Daten - schlechte Daten. Retrieved February 15, 2016, from <http://www.computerwoche.de/a/gute-daten-schlechte-daten,1931857>

Chapman, P., Clinton, J., Kerber, R., Khabaza, T., Reinartz, T., Shearer, C., & Wirth, R. (2000). CRISP-DM 1.0: Step-by-step data mining guide. *The CRISP-DM Consortium*.

The Eclipse Foundation. (2016). About the Eclipse Foundation. Retrieved February 21, 2016, from <https://www.eclipse.org/org/>

Konzeption und prototypische Implementierung einer Bug Klassifizierung zur Realisierung innerbetrieblicher Einsparungs- potenziale auf Basis des Eclipse Bugtrackers

Jens Keuter, Ines Wojtczyk, Denis Jakupovic

Abstract. *Das Projekt behandelt die Datenanalyse, Verarbeitung und Schlussfolgerung mittels der kombinierten Methoden des Text Minings und der Support Vector Machines basierend auf einem Datenexport des Eclipse Bugtrackers zur Erstellung einer Konzeption und eines Prototyps mittels der OpenSource Lösung RapidMiner Studio. Die Ergebnisse dieses Projektes liefern verwertbare Ergebnisse für eine weitergehende Forschung.*

1 Business Understanding

Software- und Netzwerkfehler mit rund 50% sowie menschliches Fehlverhalten mit 43,5% gehörten im Jahr 2014 zu den am häufigsten auftretenden IT-Ausfällen in Unternehmen und Organisationen (vgl. DRP Council 2014). Einer Studie der Accso - Accelerated Solutions GmbH zufolge sollen sogar über 10 Mrd. Euro der deutschen Wirtschaft durch Qualitäts- und Effizienzmängel verloren gehen (vgl. Accso 2012). Zunehmend rücken Ansätze für den optimierten Ressourceneinsatz in den Fokus der Unternehmen. Hierbei machen sich nicht nur finanzielle Aspekte bemerkbar (32%), sondern auch die Anforderungen von Kunden und Anbietern (22%) gewinnen zunehmend an Gewicht (vgl. European Commission 2012). Ziel dieses Projektes war es mittels einer vorzunehmenden Konzeption und der damit verbundenen prototypischen Implementierung eine Klassifizierung von Softwarefehlern, sogenannte Bugs, anhand von textuellen Beschreibungen der Fehler zu ermöglichen. Das Prinzip der angestrebten Klassifizierung sieht vor, vorhandene wie auch neu auftretende Bugs mittels eines Data Mining Prozesses automatisch in die korrekte Klasse einzuteilen. Damit soll gewährleistet werden, dass sich bei einer Änderung des Status automatisiert und kontinuierlich die Resolution und somit auch die

Zuteilung zur Weiterverarbeitung anpassen kann. Die damit verbundene effizientere Nutzung von innerbetrieblichen Ressourcen soll nicht nur Kostenersparnis beitragen, sondern auch weitere Einsparungspotenziale ermöglichen. Denn insbesondere im IT-Umfeld bedeutet eine falsche Klassifizierung von Vorfällen, die beispielsweise nicht dem Standardverlauf eines Systems entsprechen, eine ineffiziente Ressourcenallokation.

Die zu erzielende Einsparung von Ressourcen bezieht sich nicht nur auf Aspekte monetärer Art, wie Personal- oder Projektkosten und schmalere IT Budgets, sondern auch auf die damit verbundenen Umstände der Bewältigung einer stetig anwachsenden Datenmenge. Nicht wenige, wie persönliche Überforderung, Zeitdruck, Stress oder Unzufriedenheit am Arbeitsplatz und fehlende Erfolgserlebnisse sind hier zu nennen. So befinden sich gerade Entwickler in einem Zwiespalt zwischen zunehmendem Druck und wachsender Verantwortung ihrer Tätigkeiten, was sich oftmals auch in der Qualität ihrer Arbeit niederschlägt. Es erscheint, dass kritische Bugs, die bereits über eine lange Historie verfügen nicht selten mit intensiven Arbeitsstunden verknüpft sind. Die Klassifizierungsmaßnahme des Data Mining Prozess soll an dieser Stelle ansetzen. Die nachfolgende Abbildung zeigt die wesentlichen Meilensteine mit einer Kurzbeschreibung der zugrunde gelegten Arbeitspakete in Anlehnung an das CRISP-DM Modell (vgl. Chapman 2000).



Abbildung 36: Wesentliche Meilensteine im Projektverlauf

2 Data Understanding

Die zur Verfügung gestellten Daten entspringen einem Ausschnitt der Datenbank des Eclipse Trackers aus einem Erhebungszeitraum von Oktober 2001 bis Juni 2010. Zum Zeitpunkt der Bearbeitung befinden sich 316.119 Bugs in diesem Datenexport, der für die weiterführende Analyse und anschließender Konzeption verwendet werden soll.

Die im Nachgang durchgeführte Selektion, basiert auf einer im Vorfeld vorgenommenen Diskussion betriebswirtschaftlich relevanter Aspekte. Die dadurch entstandene Transparenz soll für mehr Nachvollziehbarkeit bei der Interpretation der Ergebnisse sorgen und des Weiteren eine vereinfachte Weiterverarbeitung der Daten ermöglichen.

Um eine relativ hohe Qualität der weiter zu verarbeiteten Daten zu gewährleisten wurden Fehlzusammenhänge wie auch sog. Ausreißer kategorisch von der Untersuchung ausgeschlossen. Die Konsistenz der bereits in den Daten definierten Keywords konnte nicht gewährleistet werden, da viele Zeilen nicht befüllt waren. Auffällig war, dass innerhalb der Menge verschiedener Bug-Klassen auch die Klasse „Enhancement“ zu finden war. Per Definition sind Enhancements keine auftretenden Bugs oder Fehler eines Systems, sondern vielmehr durchzuführende Verbesserung eines Systems. Sie lassen sich somit vielmehr als mögliches Resultat aus den Bugs ableiten. Insbesondere ließe sich hier eine automatisierte Klassifikation ansetzen.

Nach ausreichender Begutachtung des zur Verfügung gestellten Datensatzes entfiel die Auswahl der Daten auf die Bugs selbst und die damit eng verknüpften Volltexte, die sowohl Beschreibungen als auch Kommentar, beinhalten. Im nachfolgenden Abschnitt soll die Aufbereitung und die Integration der Daten dargelegt werden.

3 Data Preparation

Der eigentliche Zugriff auf die Daten findet direkt in RapidMiner statt. Zuvor wurde über Excel Power Query und die MySQL Workbench ein Überblick über die Daten gegeben. In diesem Zuge wurden die wesentlichen Tabellen und Attribute für die Problemstellung identifiziert. Dabei bildet die Tabelle „bugs“ die Haupttabelle. In der Tabelle „bugs_fulltext“ finden sich in den Spalten „short_desc“ und „comments“ freitextuelle Inputfelder, die Informationen zu jeder Bug ID enthalten. Sie stellen den wesentlichen Input für das Text Mining beziehungsweise die spätere Klassifikation dar. In einem zuvor angedachten Datensetup sollte ebenfalls die Tabelle „longdesc“ eingebunden werden. Diese bildet

über die Spalte „thetext“ eine kommentierte Historie der Bugs ab. Dabei ergeben sich pro Bug ID mehrere historische Texteinträge, die jeweils in einer neuen Zeile gespeichert sind. Die erste Zeile entspricht einem für die entsprechende Bug ID hinterlegten Eintrag in der Tabelle „bugs_fulltext“. Für die spätere Verwendung sollten somit alle textuellen Spalten pro Bug ID aus der Tabelle „longdesc“, ausgenommen der jeweils ersten Zeile zur Vermeidung von Redundanzen, mit den Einträgen der Tabelle „bugs_fulltext“ pro Bug ID konkateniert werden. Derart sollte eine noch größere und unter Umständen aussagekräftigere textuelle Datenbasis realisiert werden. Um diesen Vorgang auszuführen fehlten jedoch die notwendigen Super User Rechte um den Merge in ein notwendiges anderes relationales Datenmodell zu überführen. Weitere Schritte der Datenaufbereitung wurden im Vorfeld nicht unternommen.

Per MySQL-Befehl wird in einem entsprechenden RapidMiner-Operator (vgl. Kapitel 4) auf die am Lehrstuhl hinterlegte Datenbank zugegriffen. Die Tabellen „bugs“ und „bugs_fulltext“ werden somit über die „bug_id“ miteinander verbunden. Die zu ladenden Werte der „bug_severity“, welche die Bedeutsamkeit des Bugs nach welcher klassifiziert werden soll abbildet, werden hier ebenso ausgewählt, wie die „product_id“. Diese wurde inkludiert um unterschiedliche Verteilungen der Severity-Klassen abbilden zu können (vgl. Kapitel 5). Um die Laufzeit zu verringern, wird der Datensatz auf insgesamt 1500 zu ladende Zeilen, unter den entsprechenden Bedingungen, begrenzt. Abbildung 37 zeigt den finalen SQL-Befehl der im RapidMiner-Load verwendet wird. Die Restriktion bestimmter Produkt ID's ist in diesem Beispielcode ausgenommen:

```
1 SELECT *
2 FROM bugs
3 INNER JOIN
4     bugs_fulltext ON bugs.bug_id=bugs_fulltext.bug_id
5 WHERE
6     bugs.bug_severity IN ('major','enhancement')
7     /* AND bugs.product_id BETWEEN '4' AND '5' */
8 GROUP BY bugs.bug_id
9 limit 1500
```

Abbildung 37: SQL-Befehl für den Dataload

Das finale Datenset setzt sich somit aus 1500 Zeilen einer aus den Einzeltabellen „bug“ und „bugs_fulltext“ verbundenen Tabelle zusammen.

4 Modeling

4.1 Support Vector Machines

Für die Klassifizierungsaufgabe der Bugs nach ihrer Bedeutsamkeit anhand ihrer textuellen Beschreibungen, wird in RapidMiner das Konstrukt einer Support Vector Machine (SVM) verwendet. SVM werden bereits höchst erfolgreich in der Bildklassifizierung eingesetzt (vgl. UCF CRCV 2012). Der ursprüngliche Ansatz, Künstliche Neuronale Netze (KNN) für die Klassifizierung zu nutzen, wurde nach anfänglichen Testdurchläufen mit unzureichenden Ergebnissen bei gleichzeitig enorm hohen Laufzeiten verworfen. Im Folgenden soll die Entscheidungsgrundlage pro SVM anhand ihrer Definition beschrieben und die Repräsentation in RapidMiner dargestellt werden.

Da es sich um eine klassische Problemstellung der Klassifizierung handelt, in der insbesondere eine Vielzahl von Attributen in Form einzelner Wörter in das Modell eingeht, bedarf es eines Modells, das auf diese Aufgabe ausgerichtet ist. Die SVM basieren auf dem Grundprinzip des maschinellen Lernens von (Klassifikations-) Zusammenhängen in binärer Form. Jedes Objekt, welches in das Modell eingeht, wird als gewichtete Kombination einzelner Attribute x_i in einem Vektorraum dargestellt. Dieser Vektorraum wird durch eine sogenannte Hyperebene in zwei Klassen separiert, beispielsweise in „positiv“ und „negativ“ (vgl. MIT OpenCourseWare 2010). Die innerhalb der jeweiligen Klasse der Hyperebene nächstgelegenen Vektoren werden als Stützvektoren (engl. „support vectors“) bezeichnet. Sie haben im Klassifikationsvorgang und für die Klassifikationsperformance des Modells einen elementaren Einfluss: Die Klassifizierung durch die Hyperebene erreicht für sich genommen noch keine optimale Leistung. Die Trennung der Klassen erfolgt lediglich unscharf. Wird nun jedoch der Abstand der Stützvektoren der beiden Klassen zueinander gemessen, ergibt sich eine Randzone um die Hyperebene herum. Dieser Abstand ist für die Neigung und Lage der Hyperebene im Vektorraum und ist damit für die Genauigkeit der Klassifizierung von großer Bedeutung. Im vereinfachten Modell bedeutet ein maximierter Abstand der Stützvektoren eine bessere Performanceleistung des Modells, da die Aufteilung in zwei Klassen durch eine größere Randzone klarer erfolgt. Verringert sich nun der Abstand führt dies zu einer Neigung der Hyperebene im Vektorraum und folglich einer schmalen Randzone. Eine hohe Klassifizierungsleistung ist also gleichbedeutend mit einer Maximierung des Abstandes der jeweiligen klassenbezogenen Stützvektoren (vgl. Tong & Koller 2001, S. 47 ff.).

Um auf einem gegebenen Modell die optimale Lage der Hyperebene zu bestimmen bedarf es im SVM-Modell eine Trainingsphase. In dieser Phase wird, zur Berechnung des maximalen Abstandes der Stützvektoren zueinander, anhand der Entscheidungsregel zuerst jene Vektoren beschrieben, welche die Hyperebene beschreiben. Diejenigen Vektoren \vec{x} , welche die Gleichung $\langle x, w \rangle + b = 0$ erfüllen, liegen dieser Regel zufolge auf der Hyperebene und bilden sie damit ab. Dabei bezeichnet b den konstanten Versatz der Hyperebene zum Ursprung und \vec{w} einen Richtungsvektor, der orthogonal zur Hyperebene steht (Normalvektor). Um nun die Randzone zu realisieren, welche durch die Stützvektoren beschrieben wird, wird zuerst festgelegt, unter welchen Bedingungen ein Vektor zu einer Klasse gehört. Für Objekte der positiven Klasse wird somit die Nebenbedingung $\langle x_+, w \rangle + b \geq 1$ und für Objekte der negativen Klasse die Ungleichung $\langle x_-, w \rangle + b \leq -1$ eingeführt. Die Festlegung auf das Intervall $[-1, 1]$ realisiert bereits die Randzone und gibt einen natürlichen Mindestabstand der Supportvektoren zueinander (Suykens, Vandewalle 1999). Das Maximum dieses derart definierten Abstandes wird nun durch eine zweite Nebenbedingung berechnet. Da bislang lediglich die formale Bedingung und der Mindestabstand der Stützvektoren durch die erste Nebenbedingung gegeben sind, ist im Einzelfall jener Abstand beziehungsweise die Breite der Randzone genau zu berechnen. Die Gleichung $\langle (x_- - x_+), \frac{w}{\|w\|} \rangle = \frac{2}{\|w\|}$ gibt diese errechnete Breite an. Daraus lässt sich schlussfolgern, dass das Maximierungsproblems der Breite der Randzone mit $\frac{2}{\|w\|}$ schlussendlich auf einer Minimierung der quadratischen Norm $\|w\|$ basiert. Derart wird jene Hyperebene auf dem gegebenen Modell berechnet, welche die Stützvektoren mit dem größten Abstand zueinander aufweist. Um das Modell nun in einem letzten Schritt robuster gegen mögliche Fehlklassifizierungen – solche Objekte, die innerhalb der Randzone oder gar in der falschen Klasse liegen – zu machen, werden für jeden Trainingsvektor \vec{x}_i Schlupfvariablen einbezogen. Diese multiplizieren den Abstand des fehlklassifizierten Vektors zur Hyperebene mit dem Fehlergewicht C . Derart wird das eingehende Modell erweitert, dass die Optimierung nun entsprechend einer möglichst großen Randzone bei gleichzeitig geringer Summe der Fehler verläuft (vgl. MIT OpenCourseWare 2010).

Eine weitere wichtige Eigenschaft des SVM-Modells ist die Behandlung nichtlinearer Datensätze, wie sie auch in dem vorliegenden Datenmodell auftreten können. Das bishe-

rige Modell bildet eine binäre und damit lineare Klassifizierung in einem zweidimensionalen Vektorraum ab. Um nichtlineare Daten verarbeiten zu können, kann in einem SVM-Modell das Theorem von Cover, dem „Kernel Trick“ herangezogen werden. Entsprechend des Theorems können nichtlineare Daten durch die Transformation in einen ausreichend höher dimensionierten Vektorraum in linear trennbare Daten überführt werden. In dieser Dimension wird durch die Kernel-Funktion unmittelbar das Skalarprodukt berechnet, wodurch die Laufzeit, aufgrund der ausbleibenden punktweisen Transformation in den neuen Vektorraum, deutlich verringert wird (vgl. MIT OpenCourseWare 2010).

Im nachfolgenden Abschnitt soll das Konzept der SVM in RapidMiner überführt und der Gesamtprozess zur Klassifikation dargestellt werden.

4.2 Das Klassifikationsmodell in RapidMiner

Das zuvor beschriebene Konstrukt der SVM liegt in RapidMiner als vordefinierter Klassifikationsoperator vor. Um die Daten jedoch für den Operator analysierbar zu machen, bedarf es weiterer Operatoren. Des Weiteren wird die Performance der SVM hinsichtlich der Klassifikationsgenauigkeit bewertet. Die nachfolgende Abbildung 38 veranschaulicht den Prozess in RapidMiner.

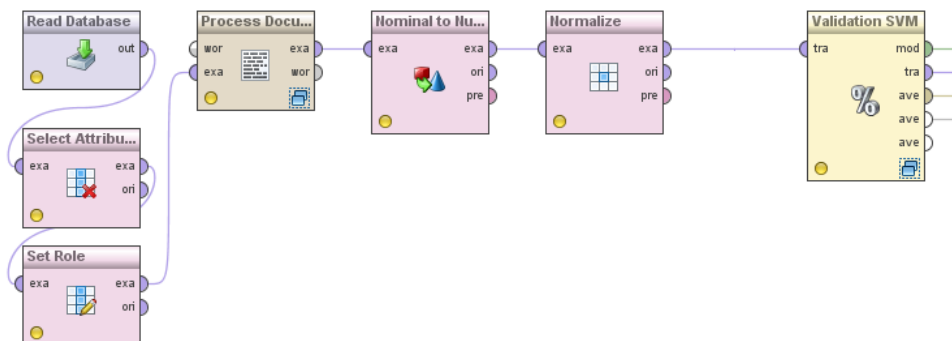


Abbildung 38: Der RapidMiner-Gesamtprozess

Über den Operator „Read Database“ werden die Daten aus der Datenbank in das Modell geladen (vgl. Kapitel 3). Um den Einfluss verschiedener Attribute auf die Performance des Modells überprüfen zu können, können im Operator „Select Attributes“ unterschiedliche Attributskombinationen ausgewählt werden. Entsprechend der gewünschten Klassifikation der Bugs, wird im Operator „Set Role“, das Label des Datensatzes auf „bug-severity“ gesetzt, was sich im Folgenden als essentiell für den späteren SVM-Operator erweist. Weiterhin wird in diesem Operator, für die spätere Nachvollziehbarkeit und Identifizierung der Bugs, die „bug_id“ als eindeutige ID des Datensatzes ausgewiesen.

Der nachfolgende Operator „Process Documents“ beinhaltet den Subprozess der Textvorverarbeitung. In diesem Schritt wird die Datenqualität erheblich verbessert, indem der textuelle Input der Spalten „short_desc_1“ und „comments“ bereinigt werden. Abbildung 39 stellt diesen Subprozess dar.



Abbildung 39: Der Subprozess der Textvorverarbeitung im Operator „Process Documents“

„Transform Cases“ überführt den gesamten Text in Kleinbuchstaben. Durch den Operator „Tokenize by non-letters“ wird der eingegebene Text in eine Sequenz von Tokens überführt. Sämtliche Zeichen ungleich Buchstaben fungieren in diesem Schritt als Spaltungsgrenzen, sodass der jeweilige Text nach Durchlauf des Operators lediglich aus Wörtern besteht. Diese Menge wird nun an den nachfolgenden Operator „Filter Stopwords“ weitergegeben. Dieser filtert sämtliche nicht-englischen Wörter, die als Stoppwörter fungieren. Durch einen weiteren Token-Filter im vorletzten Operator dieses Subprozesses, werden sämtliche Tokens gefiltert, die weniger als drei Zeichen beinhalten. In einem letzten Schritt wird für sämtliche Tokens eine Stammformreduktion vorgenommen, um die Daten weiter zu aggregieren. Das Ergebnis dieses Subprozesses und damit des gesamten Operators „Process Documents“ ist ein gewichteter Wordvektor, der sämtliche Tokens als Attribute beziehungsweise Features aufnimmt. Grundlage für diesen Wordvektor ist das durch den Subprozess realisierte Prinzip der „bag-of-words“. Nach diesem Konzept wird ein Text als ein Multimodell seiner Wörter dargestellt, ungeachtet der Grammatik und Wortreihenfolge. Vielmehr wird die Häufigkeit des Auftretens eines Wortes pro Dokument betrachtet – dargestellt in einem Wordvektor (vgl. UCF CRCV 2012). Die Gewichtung der Attribute des aus diesem Operator ausgehenden Wordvektors, erfolgt durch die Term-Frequency/ Inverse-Document-Frequency (TF/IDF). Diese stellt die Relation der Häufigkeit eines Attributs im Dokument zu der Anzahl der Dokumente, die dieses Attribut enthalten, her. Als unterschiedlich lassen sich in diesem Fall die verschiedenen Dimensionen des Klassifikationslabels „bug_severity“ betrachten. Um die Daten noch weiter zu bereinigen werden in diesem Wordvektor Wörter die mit einer geringeren Häufigkeit als 20 Vorkommnissen auftreten, aus dem Wordvektor gestrichen.

Im Rahmen des Operators „Nominal to Numerical“ und „Normalize“ (vgl. Abbildung 38) werden nicht-nominale Attribute in numerische Attribute transformiert und sämtliche Attribute durch eine Z-Transformation auf ein einheitliches Maß gebracht. In dieser Form liegen die Daten in einem bereinigten Maß für den eigentlichen Klassifikationsprozess durch den Operator „Validation SVM“ vor.

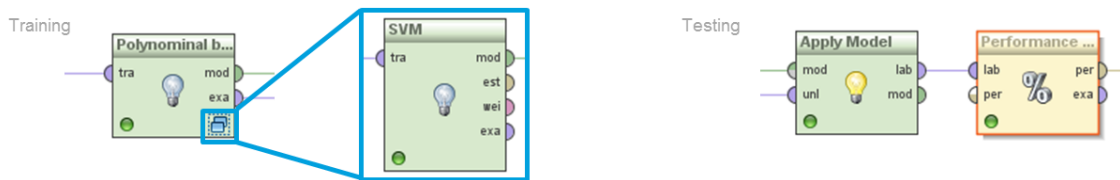


Abbildung 40: Doppelter Subprozess mit Trainings- und Testphase für das Modell der SVM

Abbildung 40 zeigt den Subprozess, der sich im Operator „Validation SVM“ findet. Dieser selbst stellt eine Kreuzvalidierung dar, die den vorliegenden Datensatz in neun gleich große Partitionen teilt und auf je acht der Partitionen trainiert sowie eine als Testpartitionen auswählt. Dieser Vorgang wird im vorliegenden Fall neun Mal wiederholt, sodass jede Partition einmal als Testdatensatz genutzt wird. Die jeweiligen Trainingspartitionen fließen in die Testphase des Subprozesses ein. Hier wird in dem Operator „Polynomial by Binomial“ die eigentliche SVM integriert. Dies ermöglicht den eigentlich binären Klassifizierer auf mehr als zwei Klassen auszuweiten. Die SVM erlernt in dieser Phase anhand der eingegebenen Daten die Zusammenhänge und die Einflüsse der einzelnen Attribute auf die Klassifizierung unter einem bestimmten Label. Nachdem die Trainingsphase durchlaufen ist, wird in der Testphase das zuvor erlernte Modell über den Operator „Apply Model“ auf die Testdaten angewendet. Der abschließende Operator „Performance“ misst den „Class Recall“¹⁹, die „Class Precision“²⁰ und die Klassifikationsgenauigkeit des Modells. Diese Werte gleichen die antizipierten Klassen des Modells mit den tatsächlichen Klassen ab.

¹⁹ Anzahl aller korrekt vorhergesagten Objekte einer Klasse / Anzahl aller tatsächlichen Objekte einer Klasse.

²⁰ Anzahl aller korrekt vorhergesagten Objekte einer Klasse / Anzahl aller vorhergesagten Objekte einer Klasse.

Die Ergebnisse des Klassifizierungsvorgangs werden im Folgenden in der Evaluation dargestellt.

5 Evaluation

Um die Aussagefähigkeit von SVM bewerten zu können, wird zunächst die Eignung des vorliegenden SVM evaluiert. Hierzu wird zunächst das Modell in seiner Grundform als binärer Klassifizierer genutzt. Dementsprechend wurde der Datensatz von 1.500 Zeilen auf die beiden Klassen „major“ und „enhancement“ sowie auf die Produkt ID's „4“ und „5“ verteilt. Auf diese Weise konnte eine ausgeglichene Verteilung des Datensatzes von 749 Enhancements zu 751 Major-Bugs erzielt werden. Nach mehreren Durchläufen zeigte sich, dass die SVM mit den Standardeinstellungen die beste Performance mit einer Klassifizierungsgenauigkeit von 85,13% erreichte. Dabei konnten sowohl der Class-Recall als auch die Class-Precision mit 86,42% beziehungsweise 84,29% für Major-Bugs sowie 83,85% beziehungsweise 86,03% für Enhancements als ausgeglichen bewertet werden.

In einem nachfolgenden Schritt wurde die Beschränkung der Produkt ID's aufgehoben und die Performance auf einem nicht balancierten Datensatz mit 1115 Enhancements und 385 Major-Bugs erprobt. Anhand dieser Evaluation zeigte sich eine problematische Eigenschaft einfacher SVM: Die in einem Datensatz stärker repräsentierte Klasse wird auch in der Klassifizierung stärker berücksichtigt (vgl. Akbani & Japkowicz 2004, S. 40). Als Ergebnis ist zwar die Klassifikationsgenauigkeit dieses Modells nur geringfügig auf 84,34% im Gegensatz zum vorherigen Fall gesunken, allerdings sind die Werte des Class-Recalls und der Class Precision für Enhancements und Major-Bugs deutlich verschieden. Ein Recall von 94,44% und eine Präzision von 95,89% für die Klasse Enhancements wurde infolgedessen erreicht, während für die Klasse Major ein Recall von 55,06% und eine Präzision von 77,37% auftritt. Die Werte der Major-Klasse sind somit nicht nur deutlich geringer gegenüber der Enhancement-Klasse, sondern weichen auch klassenintern stark voneinander ab.

Wird das Datenmodell durch die Hinzunahme des Labels „minor“ auf drei Klassen erweitert, zeigt sich die Eigenschaft der SVM, dass häufig auftretende Klassen in einem noch deutlicheren Ausmaß stärker berücksichtigt werden: Die größte Klasse „enhancement“ (934) erhält einen ähnlichen Recall wie zuvor. Für die Klasse „major“ (318) sinkt der Recall jedoch auf 48,11%, für Minor-Bugs werden gar nur 8,47% erreicht. Da die

Anteile der Enhancement- und Major-Klassen deutlich gesunken sind und der Anteil der Minor-Bugs lediglich bei 34,43% liegt, ist auch die durchschnittliche Genauigkeit des Modells mit 69,80% nicht ausreichend zu qualifizieren.

Diese Eigenschaft der SVM stellt ein wesentliches Problem dar: Im betrieblichen Umfeld werden selten derart ausgeglichene Daten in Bug- oder Ticketsystemen vorzufinden sein, sondern stets nicht balancierte Verteilungen mit dominierenden Klassen (vgl. Akbani & Japkowicz 2004, S. 39 ff.). Obgleich die Klassen im Datensatz unterschiedlich gewichtet sind, wird eine Fehlklassifikation in Form des Fehlergewichts C in beiden Klassen gleich bewertet. Daher kann der Effekt resultieren, dass die Fehlklassifikation eines Objekts der Minorität „in Kauf genommen“ wird, da der Abstand der Stützvektoren zueinander am größten und die Fehlerkosten der kleineren Klasse nicht im größeren Umfang gewichtet werden (vgl. Akbani & Japkowicz 2004, S. 42).

Um diese Problematik zu lösen, ist es angebracht, eine Gewichtung der Fehlklassifikation in das Modell zu integrieren, welche die Verteilung konterkariert. Somit werden Fehler in der Minorität stärker gewichtet als in der dominierenden Klasse. Um dies umzusetzen wurde der SVM-Operator in RapidMiner (vgl. Kapitel 4.2.) durch eine spezifischere SVM ersetzt – die LibSVM, welche diesen Gewichtungsanspruch erfüllt (vgl. Chang & Lin 2011). Erste Testdurchläufe des Modells mit diesen Ergänzungen auf dem oben genannten nicht balancierten Zwei-Dimensionen-Datensatz zeigen, dass sich zumindest ein Zuwachs des Class-Recalls, nach sukzessiver Adjustierung der Gewichtungsparameter und des Fehlergewichts C , von 6,56% erreichen lässt.

Da die Performance nach wie vor nicht ausreichend ist, bedarf es weiterer Schritte, zum Beispiel der Modifikation einzelner Parameter, wie einer weiteren Adjustierung der Gewichtung der Fehlklassifikation, oder noch tiefergehende Datenaufbereitungen und -bereinigungen im Vorfeld der Analyse

6 Deployment

Die vorausgegangenen Betrachtungen brachten mittels SVM anfänglich verwertbare und plausible Ergebnisse hervor. Bei tiefergehender Betrachtung empfehlen sich jedoch für zukünftige Untersuchung vermehrt ausgeglichene Datensätze. Der hier beschriebene Ansatz sollte insbesondere mit Blick auf die stärkere Integration von Fehlergewichten und einer noch prägnanteren und größeren Datenbasis weiterentwickelt werden. Hiervon aus-

gehend soll eine Handlungsempfehlung abgeleitet werden, die nicht nur für die Weiterverwendung von SVM spricht sondern auch vor dem Hintergrund heutiger und künftiger innerbetrieblichen Einsparungspotenziale realisiert werden kann. Ferner kommt hinzu, dass eine verständliche, ausführliche und vollständige Dokumentation unausweichlich für die Weitergabe und -verarbeitung von Softwareentwicklungen sind. Kundenorientierte Reports oder ansprechende Kurzpräsentationen, die wesentliche Themen wie Finals, Monitoring oder Maintenance beinhalten sind deshalb auch in Zukunft weiterhin unverzichtbar. Das Konzept der automatisierten Klassifizierung von Bugs in definierte Klassen und nach ihrer Bedeutsamkeit, lässt sich zudem ohne größere Umstände auch auf das Konzept des Service Desks transformieren. Hierbei kann durch die korrekte Klassifizierung von Incidents, Enhancements sowie weiteren Ticket-Arten enorme Einsparungspotenziale realisiert werden. Durch die weitere Erforschung, Erprobung und Weiterentwicklung des Ansatzes von SVM als Klassifizierungskonzept könnte sich ein Vorschlagswesen mit einem möglichst breiten Lösungsspektrum realisieren und zunehmend etablieren.

7 Literatur

Akbani, R., Kwek, S., Japkowicz, N. (2004). Applying support vector machines to imbalanced datasets. In Machine learning: ECML 2004 (pp. 39-50). Springer Berlin Heidelberg.

Accso: Accso-Studie zu Qualitäts- und Effizienzmängeln in Softwareprojekten im Jahr 2012, unter: <http://www.accso.de/index.php/news/70-accso-studie> (abgerufen am 24.02.2016).

Chang, C. C., Lin, C. J. (2011). LIBSVM: A library for support vector machines. ACM Transactions on Intelligent Systems and Technology (TIST),2(3), 27.

Chapman, P., et al., 2000, CRISP-DM 1.0, S. 10.

DRP Council: Hauptursachen von IT-Ausfällen oder Datenverlust in Unternehmen/Organisationen weltweit im Jahr 2014, unter: <http://de.statista.com/statistik/daten/studie/412753/umfrage/ursachen-von-it-ausfaellen-oder-datenverlust/> (abgerufen am 24.02.2016).

European Commission: Was sind die Hauptgründe, weshalb Ihr Unternehmen Maßnahmen zur Verbesserung der Ressourceneffizienz ergreift?, unter: <http://de.statista.com/statistik/daten/studie/255790/umfrage/gruende-von-unternehmen-fuer-massnahmen-zur-verbesserung-der-ressourceneffizienz/> (abgerufen am 24.02.2016).

MIT OpenCourseWare (2010): 16. Learning: Support Vector Machines, MIT 6.034 Artificial Intelligence [Youtube-Video], 10.01.2014, unter: https://www.youtube.com/watch?v=_PwhiWxHK8o (abgerufen am: 23.02.2016).

Suykens, J. A., & Vandewalle, J. (1999). Least squares support vector machine classifiers. *Neural processing letters*, 9(3), 293-300.

Tong, S., Koller, D. (2001): Support Vector Machine Active Learning with Applications for Text Classification, in: *Journal of Machine Learning Research*, Volume 2, 03.01.2002, S. 45-66, unter: <http://dl.acm.org/citation.cfm?id=944793> (abgerufen am: 23.02.2016).

UCF CRCV (2012): Lecture 17: Bag-of-Features (Bag-of-Words), UCF Computer Vision Video Lectures 2012 [Youtube-Video], unter: <https://www.youtube.com/watch?v=iGZpJZhqEME> (abgerufen am: 23.02.2016).

Prognose der Bearbeitungsdauer von Bugs mit Hilfe von Clustering und Entscheidungsbäumen

Michael Körfer, Victor Brinkhege, Kai Steenblock

Abstract. Wenn Nutzer einer Software im Eclipse Bugtracker einen Bug melden, sollen sie schon beim Meldezeitpunkt eine Prognose der Bearbeitungsdauer erhalten. Diese wird anhand gewisser Attribute (z.B.: Betriebssystem, Produkt, Komponente) des Bugs approximiert. Unter Verwendung einer Clusteranalyse werden zuerst bereits bearbeitete Bugs in Klassen mit entsprechender Bearbeitungsdauer segmentiert. Diese Klassen werden daraufhin wieder mit den ursprünglichen Daten verknüpft. Anschließend wird mit diesen Daten ein Entscheidungsbaum generiert. Der Entscheidungsbaum besitzt die Fähigkeit, neue Bugs anhand der genannten Attribute in die erstellten Klassen einzuordnen. Abschließend wird das Modell des Entscheidungsbaumes noch an einem Testdatensatz validiert. Dabei stellte sich heraus, dass kurze Bearbeitungszeiten deutlich zuverlässiger geschätzt werden konnten, als lange Bearbeitungszeiten.

1 Business Understanding

Die Eclipse Foundation ist eine gemeinnützige Gesellschaft, welche die Aufgabe hat, die Eclipse Open Source Gemeinschaft und ihre Projekte zu leiten.

Ziel dieser Gemeinschaft ist es, eine Plattform für die Entwicklung von Software bereitzustellen. Neben den Projekten ist auch die Entwicklung transparent. So kann durch den Eclipse Bugtracker praktisch jeder Benutzer Bugs melden.

Eine Problematik ergibt sich bei der Erstellung der Bugs. Ein User hat bei der Erstellung eines neuen Bug Tickets praktisch keinerlei Anhaltspunkte, wie lange es dauert, bis der Fehler behoben wird.

Ziel dieses Beitrages ist es, mithilfe von Methoden der künstlichen Intelligenz ein Modell zu entwickeln, das die Bearbeitungszeit von Bugs approximativ berechnen kann.

Im Folgenden soll kurzer ein Überblick über das weitere Vorgehen skizziert werden.

Zunächst müssen die „Key Attributes“ identifiziert werden, welche für das Data Mining von Bedeutung sind. Wichtig ist hierbei, dass die Datenqualität beachtet wird, sodass die

Validität des späteren Modells gewährleistet ist. Weiterhin ist das Datenformat von Bedeutung, da die entsprechenden Algorithmen nicht jedes Datenformat auswerten können. Die Daten müssen erst in ein entsprechendes Format umgewandelt werden.

Sind die „Key Attributes“ bekannt, so kann im nächsten Schritt das Modell entwickelt werden. Dies soll mithilfe des Programmes RapidMiner geschehen.

Ziel dieses Schrittes ist es, zunächst die Daten aufzubereiten und diese in ein verständliches Format umzuwandeln. Anschließend soll ein Clustering der Daten stattfinden. Im letzten Schritt wird ein Entscheidungsbaum entwickelt.

Tabelle 14: Projektphasen von CRISP-DM

Schritt	Phase	Ziele	Termin
1	Business Understanding	Zieldefinition und Projektplan	06.01.16
2	Data Understanding	Bestimmung der Datenqualität der vorliegenden Daten	13.01.16
3	Data Preparation	Bereinigtes Datenset	20.01.16
4	Modeling	Entscheidungsbaum und Clustering der Daten	23.01.16
5	Evaluation / Deployment	Interpretation der Ergebnisse und Beschränkung der Einsatzmöglichkeiten	27.01.16

Der Prozess der Analyse lehnt an den Cross Industry Standard Process for Data Mining (CRISP-DM) an.

Dabei werden folgende Phasen unterschieden: Business Understanding, Data Understanding, Data Preparation, Modeling, Evaluation und Deployment.

Jede Phase wird mit einem Zwischenziel abgeschlossen, welches das Ergebnis der Phase widerspiegelt und einem Meilenstein gleichzusetzen ist. Tabelle 14 bildet alle Phasen und Zwischenziele des Projektes ab.

2 Data Understanding

2.1 Daten Beschreibung

Die Ausgangsdaten des Eclipse Bugtrackers sind in einer SQL Datenbank abgebildet. Diese Datenbank umfasst einen Zeitraum von 2001 bis Ende Juni 2010 und besteht aus 18 Tabellen und unter Tabellen. Weiterhin hat die Datenbank einen Umfang von 316.911 eingetragenen Bugs. Neben Informationen zu den Bugs, wie Bug ID, Operating System oder dem Erstellungsdatum, enthält die Datenbank auch noch andere Bereiche, wie Informationen zu Produkten und Komponenten, Keywords, Historisierung von Änderungen, Informationen über Anhänge, Volltexte (Beschreibungen und Kommentare), sowie zusätzliche Informationen zu den Bugs, die im Bereich Bug selber nicht aufgenommen sind. Informationen zu den Usern enthält die Datenbank allerdings nicht.

Eine Erklärung zu den einzelnen Relationen der Datenbank des Eclipse Bugtrackers zeigt die Tabelle 15.

Tabelle 15: Aufbau der Eclipse Bugtracker Datenbank

Bereich	Tabelle	Beschreibung
Bugs	Bugs	Zentrale Tabelle mit allen Bugs
Keywords	Keywords	Verbindung von keywords mit Bugs
	Keywordsdefs	Inhalt der Keywords mit kurzer Beschreibung
Produkte und Komponenten	products	Produkte der Eclipse Foundation, die vom Bugtracker erfasst wurden
	versions	Version der Produkte
	milestones	Meilensteine der Produkte
	component_cc	Zuordnung von Benutzern zu Komponenten
	components	Zuordnung von Komponenten der einzelnen Produkten
Historisierung der Änderungen	Bugs_activity	Wann wurde welches Feld verändert
	fielddefs	Definition der Felder
Volltexte	longdescs	Beinhaltet alle Kommentare

(Beschreibungen und Kommentare)		
Informationen über Anhänge	attachments	Informationen zu den Anhängen aller Bugs
Weitere Informationen zu Bugs	dependencies	Reihenfolge der Bug Lösung
	duplicates	Informationen über doppelte Bugs
	cc	Benachrichtigungen von einem Benutzer in Abhängigkeit eines Bugs
	votes	Votes pro Bug
	bugs_fulltext	Titel und Beschreibung des bugs
	bug_see_also	Verweise auf andere Bugs per Link

2.2 Daten Qualität

Die recht umfangreiche Datenbank macht auf den ersten Blick einen guten Eindruck. Bei einigen Tabellen kommt es jedoch vor, dass es Attribute gibt, welche nicht gefüllt wurden. So besitzen beispielsweise nur rund 7% der Datensätze Keywords.

Bei den für das zu entwickelnde Modell kritischen Attributen, wie dem Creation Date eines Bugs oder dem Completion Date, besitzen hingegen alle geschlossenen Bug Meldungen die benötigten Daten, da diese vom System automatisch generiert werden und nicht von User Angaben abhängig sind.

Dementsprechend können für das zu entwickelnde Modell, soweit gewünscht, alle verfügbaren Daten ohne Einschränkung genutzt werden.

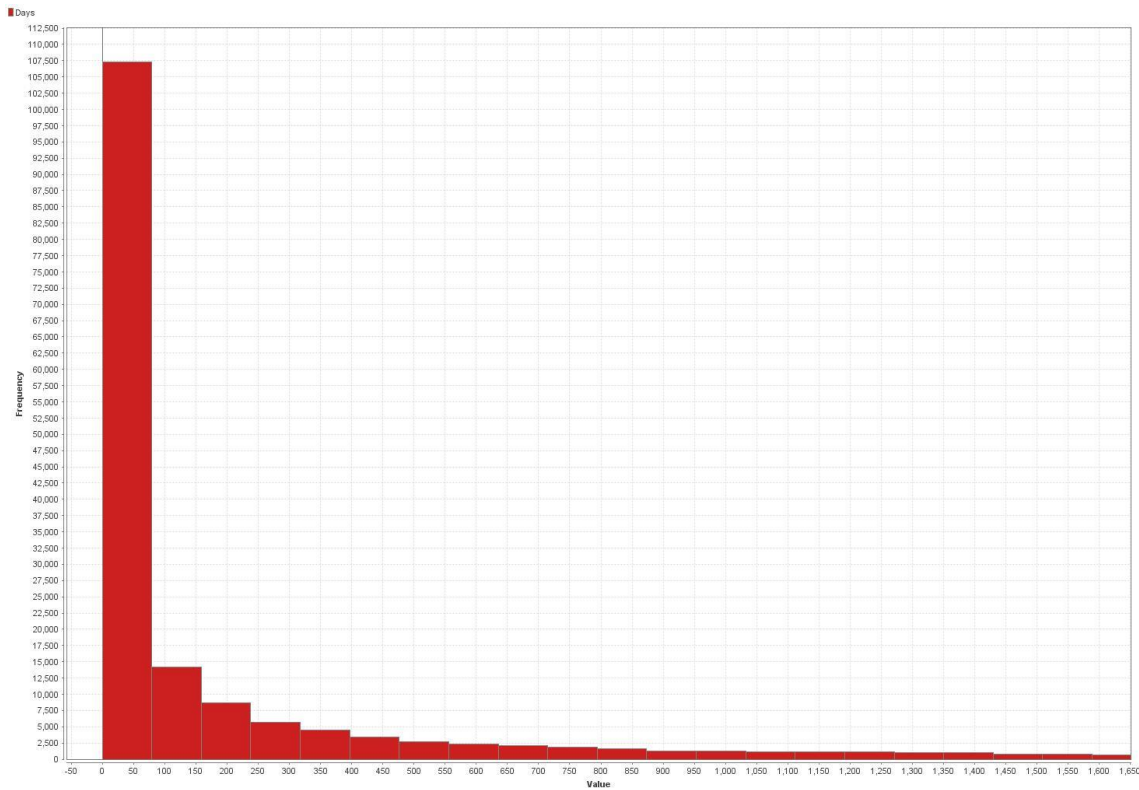


Abbildung 41: Histogramm der Daten

Eine Voruntersuchung der Zeit von Eröffnung bis Schließung eines Bugs, bestätigt diesen Eindruck. So wird ein Großteil der Bugs in einem recht kurzen Zeitraum von unter 100 Tage gelöst.

bug_id	Integer	0	Min 2	Max 318057	Average 146260.418	Deviation 95710.833
version	Nominal	0	Least 4.2.1.1 (1)	Most unspecified (28444)	Values unspecified (28444), 3.0 (22791), ... [247 more]	
op_sys	Nominal	0	Least Windows Server 2008 (1)	Most Windows XP (73646)	Values Windows XP (73646), All (33215), ... [32 more]	
product_id	Integer	0	Min 1	Max 176	Average 20.502	Deviation 33.965
component_id	Integer	0	Min 1	Max 1356	Average 220.196	Deviation 323.341
bug_severity	Nominal	0	Least trivial (2153)	Most normal (115749)	Values normal (115749), enhancement (23019), ... [5 more]	
priority	Nominal	0	Least P5 (1604)	Most P3 (150649)	Values P3 (150649), P2 (8063), ... [3 more]	
Creation	Date time	0	Earliest date Oct 10, 2001 9:34 PM	Latest date Jun 25, 2010 6:16 PM	Duration 31794 20h 41m 28s	
Diffed	Date time	2	Earliest date Oct 11, 2001 5:35 AM	Latest date Jun 26, 2010 2:07 AM	Duration 31794 20h 31m 41s	
Days	Real	2	Min 0	Max 3176.484	Average 234.356	Deviation 470.151

Abbildung 42: Statistik der Datensätze

Der Durchschnitt der Bearbeitungszeit liegt bei ungefähr 234 Tagen.

Das Maximum bei 3176 Tagen. Die Anzahl der Karteileichen ist im Vergleich zur gesamten Datenmenge relativ gering.

Die Datenqualität kann also als sehr gut beschrieben werden und für die Entwicklung des Modells herangezogen werden.

3 Data Preparation

Damit unsere Modelle die Daten verarbeiten können, ist es notwendig die rohen Daten entsprechend vorzubereiten. Konkret sind dazu folgende Fragen zu klären:

- Welche Daten sind für die Fragestellung relevant?
- Wie werden die relevanten Daten extrahiert?
- Was für Zwischenschritte sind dabei nötig?

Endgültiges Ziel des Entscheidungsbaumes ist es, die Bearbeitungszeit von Bugs zu schätzen. Man kann diese Aufgabe in zwei Teilaufgaben zerlegen:

1. die Bearbeitungszeit von bereits bearbeiteten Bugs berechnen können und
2. Herausfinden, welche Faktoren die Bearbeitungszeit beeinflussen.

Die Bearbeitungszeit ist die Differenz zwischen dem Meldedatum und dem Lösungsdatum. Das Meldedatum ist ein einfaches Attribut der Tabelle Bugs, `creation_ts`. Das Lösungsdatum kann aus den Attributen `bug_status` und `lastdiffed` berechnet werden. Dazu werden die Einträge herausgefiltert, bei denen der `bug_status` zuletzt auf "RESOLVED" geändert wurde. Von diesen "gelösten" Bugs kann nun das Feld `lastdiffed` als Lösungsdatum verwendet werden.

Herauszufinden, welche Faktoren die Bearbeitungszeit eines Bugs beeinflussen ist eine deutlich komplexere Aufgabe. Im Grunde ist die Aufgabe des Entscheidungsbaumes, zwischen den ausschlaggebenden und unwichtigen Attributen zu diskriminieren. Allerdings muss bereits eine Vorauswahl getroffen werden. Es gibt einige Attribute, die definitiv keinen Einfluss auf die Bearbeitungszeit haben, die aber trotzdem unter Umständen mit der Bearbeitungszeit korrelieren. Es wäre zum Beispiel denkbar, dass die Bearbeitungszeit mit höheren Bug IDs kürzer wird. Dies könnte daran liegen, dass eine höhere Bug ID bedeutet, dass mehr Menschen die Software nutzen und dadurch auch mehr Menschen an der Software arbeiten, da der Markt wächst. Aber daraus zu folgern, dass die Bug ID die Bearbeitungszeit beeinflusst wäre ein Trugschluss. Die eigentliche Ursache

ist die Anzahl der beteiligten Entwickler und diese lässt sich nicht direkt von der Bug ID ableiten.

Gleichzeitig gibt es viele Attribute, die nur in Verbindung mit anderen Attributen wirklich aussagekräftig sind. Ein Beispiel dafür wäre die Versionsnummer. Erfahrungsgemäß haben Bugs von reifen Produkten eine längere Bearbeitungszeit als Bugs in relativ neuen Produkten. Allerdings kann man Versionsnummern nicht pauschal vergleichen, sondern muss sie immer mit Bezug auf ein bestimmtes Produkt betrachten. Das liegt daran, dass verschiedene Produkte unterschiedlich lange Entwicklungszyklen haben. Bei manchen Softwareprodukten, wie zum Beispiel Smartphone Apps kommt jede Woche eine neue Version heraus. Bei Professioneller Office-Software gibt es vielleicht nur alle paar Jahre ein Update. In diesem Fall wäre es naiv, die Versionsnummer der beiden Produkte einfach miteinander zu vergleichen.

Weiterhin gibt es Attribute, die ziemlich sicher Einfluss auf die Bearbeitungszeit haben, die aber kurz nach der Meldung eines neuen Bugs meist noch nicht vorliegen. Ein Beispiel dafür wäre das Attribut `assigned_to`, das festhält, welcher Entwickler für einen Bug zuständig ist. Da es aber das Ziel ist, die Bearbeitungszeit schon beim Meldezeitpunkt vorherzusagen, müssen wir uns auf Attribute beschränken, die dann auch schon vorhanden sind.

Daher wurden folgende Attribute ausgewählt:

- Betriebssystem
- Produkt
- Komponente
- Schweregrad
- Versionsnummer

Bevor ein Entscheidungsbaum zur Schätzung der Bearbeitungszeit erstellt werden kann, werden grobe Kategorien erstellt, in die der Entscheidungsbaum die Bugs einteilt. Dadurch erhoffen wir uns, eine aussagekräftige Antwort zu bekommen. Um dies zu erreichen, wird zuerst ein Clustering durchgeführt.

Dafür ist es notwendig, die rohen Daten zu bereinigen. Bei RapidMiner müssen zuerst die oben genannten Attribute aus der Datenbank gelesen und dann mit dem "Generate Attribute" Operator das neue Feld für die Bearbeitungszeit berechnet werden. Da das

Clustering nur anhand dieses Attributs geschehen soll, werden die “Select Attribute” und “Set Role” Operatoren angewendet, um dies sicherzustellen.

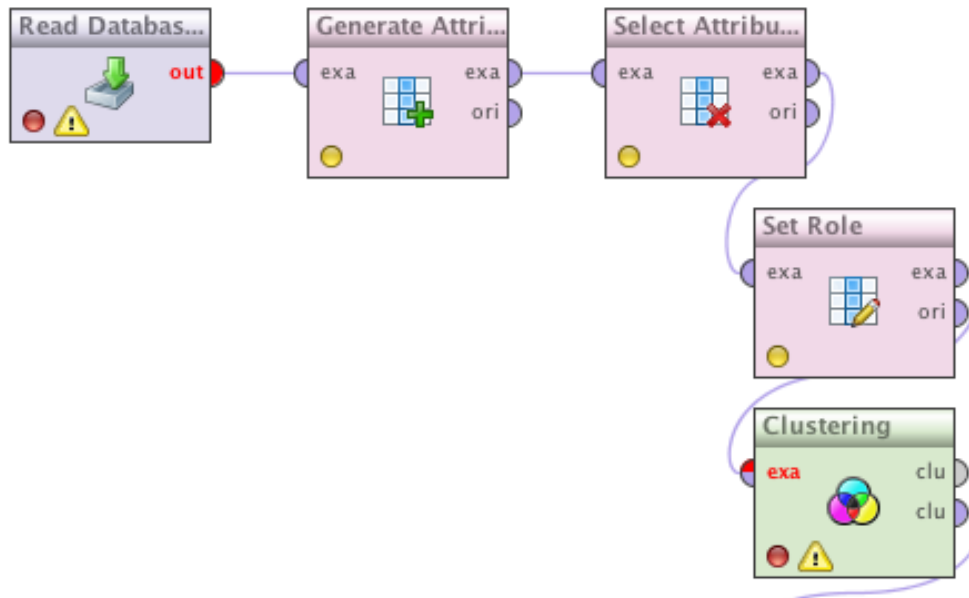


Abbildung 43: Datenreinigung & Clustering

Der “Clustering” Operator erzeugt eine neue Spalte, die jeden Bug einem Cluster zuordnet. Diese Spalte muss nun noch mit den relevanten Bug-Attributen verknüpft werden. Dies geschieht mit Hilfe des “Join” Operators.



Abbildung 44: Join Operator

Im nächsten Schritt wird nun der Datensatz für den “Decision Tree” Operator vorbereitet. Dazu werden die Daten in Test- und Trainingsdatensätze aufgeteilt. Dabei dient der Trainingsdatensatz der Erstellung des Entscheidungsbaums. Daraufhin validiert der “Performance” Operator die Leistung des Modells am Testdatensatz.

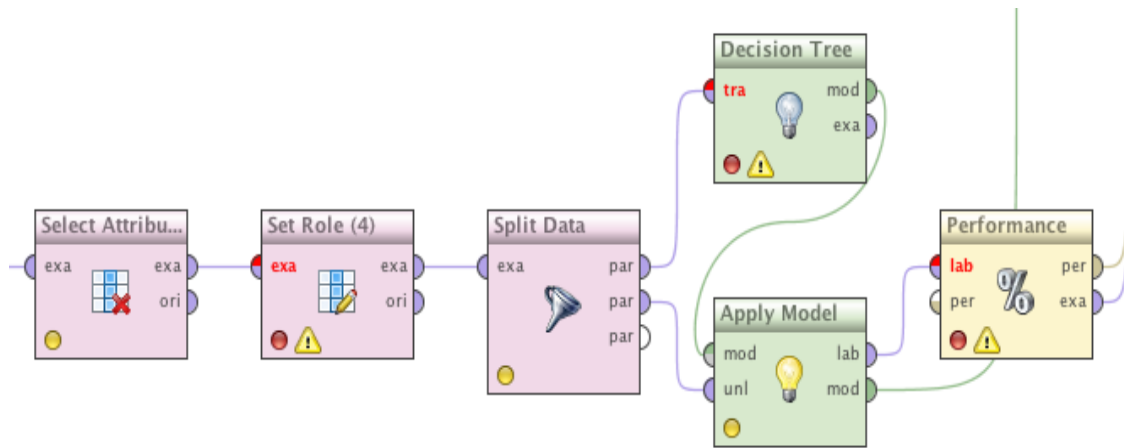


Abbildung 45: Entscheidungsbaum

Es entsteht eine Tabelle von gelösten Bugs mit relevanten Attributen, wobei das Cluster Attribut als Zielattribut für den Entscheidungsbaum festgelegt wird.

ExampleSet (10000 examples, 2 special attributes, 10 regular attributes)											Filter (10,000 / 10,000 examples):	
Row No.	bug_id	cluster	Days	id	version	op_sys	product_id	component..	bug_severity	priority	Creation	
1	2	cluster_2	208.541	1	2.0	All	1	6	normal	P5	Oct 10, 2001	
2	3	cluster_0	3130.538	2	2.0	Windows All	1	6	normal	P5	Oct 10, 2001	
3	4	cluster_2	141.828	3	2.0	All	1	6	normal	P5	Oct 10, 2001	
4	5	cluster_0	2500.437	4	2.0	Windows NT	1	6	normal	P3	Oct 10, 2001	
5	6	cluster_2	119.830	5	2.0	All	1	6	normal	P5	Oct 10, 2001	
6	7	cluster_1	1303.822	6	2.0	All	1	6	normal	P5	Oct 10, 2001	

Abbildung 46: Entstandene Tabelle

Es stellt sich die Frage, wie man die Bearbeitungsdauer eines Bugs definiert. Wie auf der Abbildung zu erkennen, bietet der Eclipse Bugtracker verschiedene Möglichkeiten, einen Bug als bearbeitet zu markieren. Abgeschlossene Bugs haben als Bug Status entweder den Eintrag "RESOLVED" (gelöst) oder "VERIFIED" (nachgeprüft). Gelöste Bugs können dann noch in 5 Unterschiedliche Kategorien eingestuft werden. Für unsere Fragestellung ist dies wichtig, denn man könnte behaupten, dass ein Bug der als "INVALID" oder "DUPLI-CATE" gelöst wurde, gar nicht richtig gelöst wurde. Der Bug wurde zwar behandelt, aber es wurden keine Veränderungen im Programmcode vorgenommen. Man könnte also behaupten, dass solche Pseudo-Bugs die Bearbeitungszeit verfälschen.

Closed Bugs	
<p>RESOLVED A resolution has been performed, and it is awaiting verification by QA. From here bugs are either reopened and given some open status, or are verified by QA and marked VERIFIED.</p> <p>VERIFIED QA has looked at the bug and the resolution and agrees that the appropriate resolution has been taken. This is the final status for bugs.</p>	<p>FIXED A fix for this bug is checked into the tree and tested.</p> <p>INVALID The problem described is not a bug.</p> <p>WONTFIX The problem described is a bug which will never be fixed.</p> <p>DUPLICATE The problem is a duplicate of an existing bug. When a bug is marked as a DUPLICATE, you will see which bug it is a duplicate of, next to the resolution.</p> <p>WORKSFORME All attempts at reproducing this bug were futile, and reading the code produces no clues as to why the described behavior would occur. If more information appears later, the bug can be reopened.</p>

Abbildung 47: Abgeschlossene Bugs

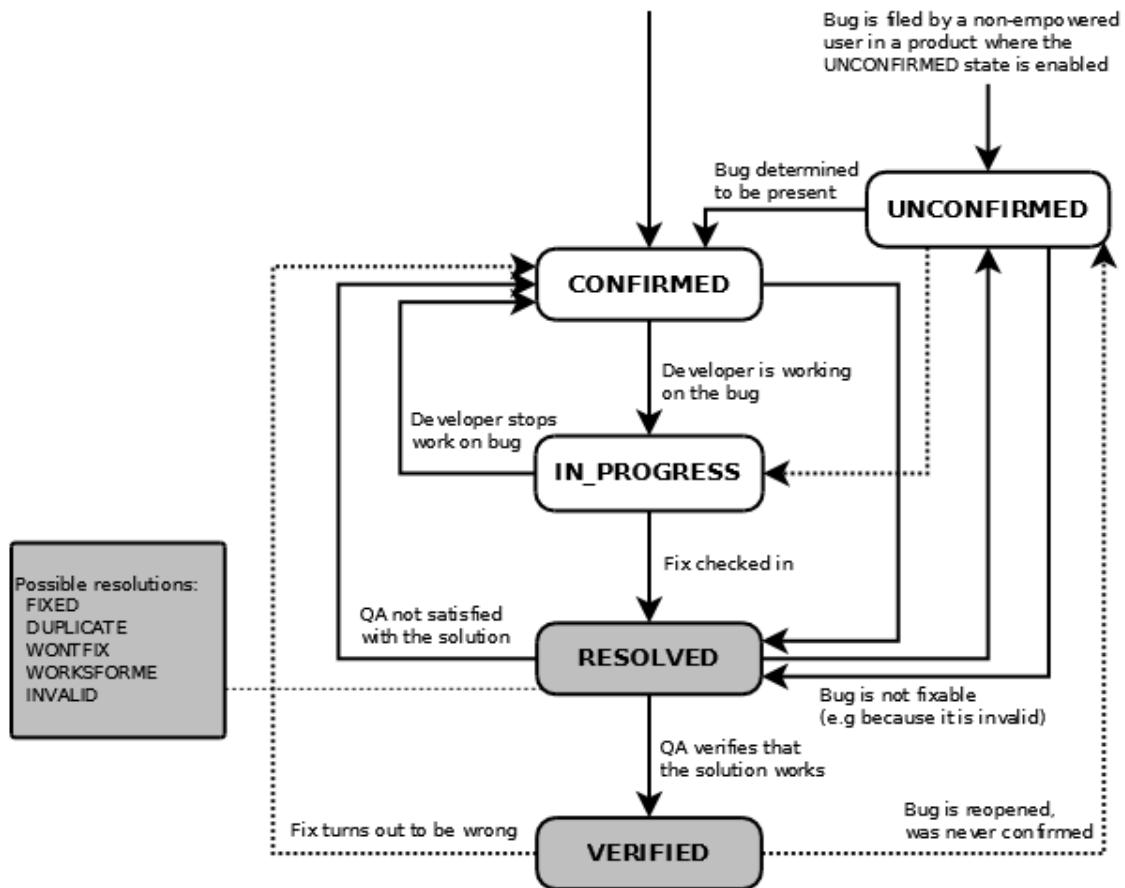
Figure 5.1. Lifecycle of a Bugzilla Bug

Abbildung 48: Lebenszyklus eines Bugs

Es wurde sich darauf geeinigt, auch diese Pseudo-Bugs zu berücksichtigen. Der Grund dafür ist, dass ein Benutzer beim Melden eines neuen Bugs noch nicht weiß, ob der Bug invalide oder etwa ein Duplikat ist. Um auch in solchen Fällen eine aussagekräftige Vorhersage zu treffen, muss das Modell diese Bugs in seine Berechnungen miteinbeziehen.

4 Modeling

Wie bereits in Kapitel 3 erwähnt, werden zur Schätzung der Bearbeitungszeit zwei verschiedene Modelle angewendet; zum einen das Clustering, zum anderen ein Entscheidungsbaum. Um diese zwei Models in den gesamten Workflow einordnen zu können, ist dieser in Abbildung 49 abgebildet.

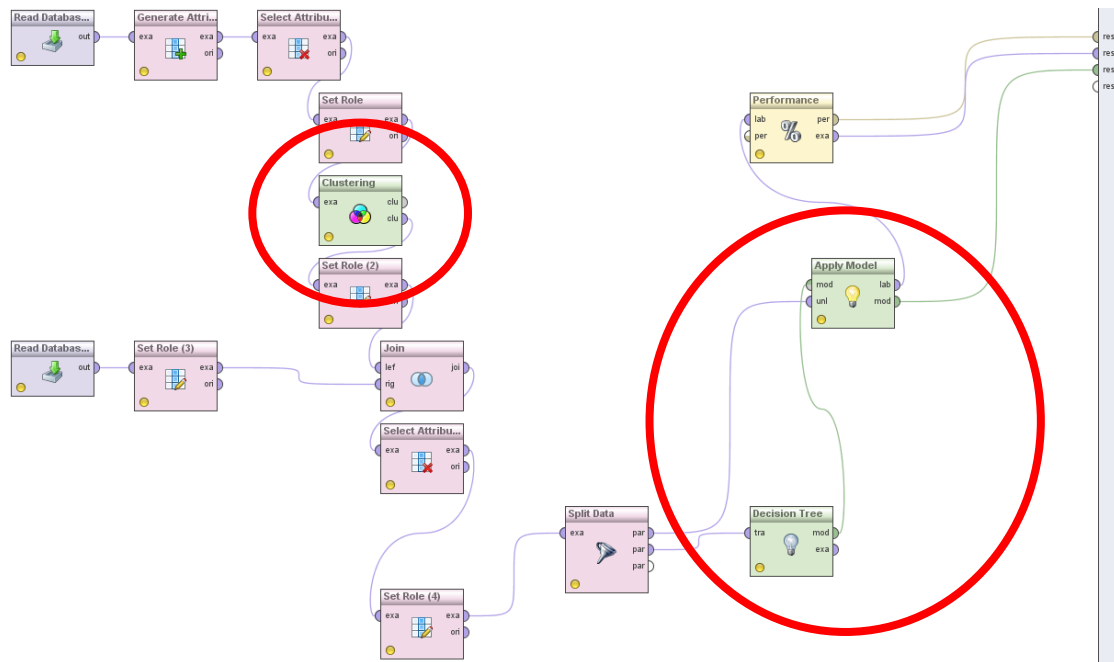
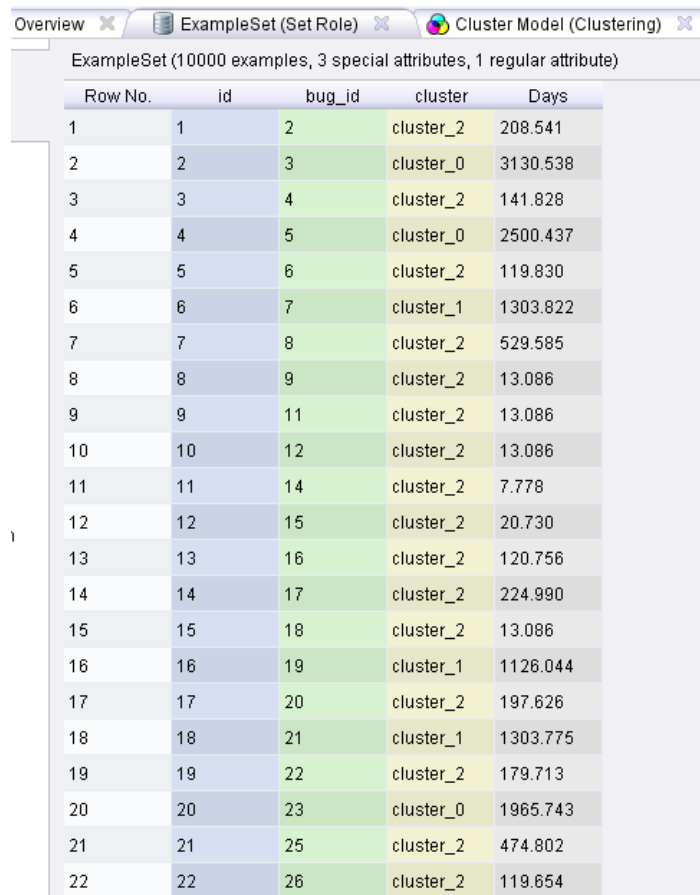


Abbildung 49: Lebenszyklus eines Bugs

Im Folgenden wird zuerst das Clustering Modell betrachtet. Hier wurde der k-Means Algorithmus verwendet. Dabei wird aus einer Menge von ähnlichen Objekten eine vorher bekannte Anzahl von Gruppen gebildet. Die Gruppierung der Datensätze soll logischerweise anhand der berechneten Zeitspanne in Tagen erfolgen. Die Entscheidung fiel in diesem Fall auf drei Gruppen n ($k = 3$). Dies soll später die Einteilung der Bugs in langfristige, mittelfristige und kurzfristige Bearbeitungszeit ermöglichen. In RapidMiner gibt es die Möglichkeit, das ermittelte Cluster dem jeweiligen Datensatz als Attribut zuzuordnen. Der Output „clustered set“ gibt nun diesen Datensatz aus. Dieser ist in Abbildung 50 dargestellt.

Der k-Means Algorithmus hat die Bugs in Cluster 0, Cluster 1 und Cluster 2 eingeteilt. Cluster 0 entspricht einer langen Bearbeitungszeit von durchschnittlich 7 Jahren, Cluster 1 enthält Bugs mit einer mittelfristigen Bearbeitungszeit von etwa 3 Jahren und Cluster 2 entspricht Bugs mit einer kurzen Bearbeitungszeit von etwa $\frac{1}{4}$ Jahr.



ExampleSet (10000 examples, 3 special attributes, 1 regular attribute)

Row No.	id	bug_id	cluster	Days
1	1	2	cluster_2	208.541
2	2	3	cluster_0	3130.538
3	3	4	cluster_2	141.828
4	4	5	cluster_0	2500.437
5	5	6	cluster_2	119.830
6	6	7	cluster_1	1303.822
7	7	8	cluster_2	529.585
8	8	9	cluster_2	13.086
9	9	11	cluster_2	13.086
10	10	12	cluster_2	13.086
11	11	14	cluster_2	7.778
12	12	15	cluster_2	20.730
13	13	16	cluster_2	120.756
14	14	17	cluster_2	224.990
15	15	18	cluster_2	13.086
16	16	19	cluster_1	1126.044
17	17	20	cluster_2	197.626
18	18	21	cluster_1	1303.775
19	19	22	cluster_2	179.713
20	20	23	cluster_0	1965.743
21	21	25	cluster_2	474.802
22	22	26	cluster_2	119.654

Abbildung 50: Datensatz mit ermitteltem Cluster

Um nun alle Daten dem Entscheidungsbaum übergeben zu können, wird nun der bereits näher erläuterte Inner-Join durchgeführt, um anhand der Bug-ID alle Attribute den jeweiligen Bugs wieder zuzuordnen. Es entsteht nun der Datensatz mit dem der Entscheidungsbaum Algorithmus, nach der Auswahl der relevanten Attribute durch den „Select Attributes“ Operator, den Baum berechnet. Diese Tabelle ist in Abbildung 51 dargestellt.

Row No.	bug_id	cluster	Days	id	version	op_sys	product_id	component...	bug_severity	priority	Creation
1	2	cluster_2	208.541	1	2.0	All	1	6	normal	P5	Oct 10, 2001
2	3	cluster_0	3130.538	2	2.0	Windows All	1	6	normal	P5	Oct 10, 2001
3	4	cluster_2	141.828	3	2.0	All	1	6	normal	P5	Oct 10, 2001
4	5	cluster_0	2500.437	4	2.0	Windows NT	1	6	normal	P3	Oct 10, 2001
5	6	cluster_2	119.830	5	2.0	All	1	6	normal	P5	Oct 10, 2001
6	7	cluster_1	1303.822	6	2.0	All	1	6	normal	P5	Oct 10, 2001
7	8	cluster_2	529.585	7	2.0	All	1	6	normal	P3	Oct 10, 2001
8	9	cluster_2	13.086	8	2.0	All	1	6	normal	P3	Oct 10, 2001
9	11	cluster_2	13.086	9	2.0	All	1	6	normal	P3	Oct 10, 2001
10	12	cluster_2	13.086	10	2.0	All	1	6	normal	P3	Oct 10, 2001
11	14	cluster_2	7.778	11	2.0	All	1	6	normal	P3	Oct 10, 2001
12	15	cluster_2	20.730	12	2.0	Windows 20	1	6	normal	P3	Oct 10, 2001
13	16	cluster_2	120.756	13	2.0	All	1	6	normal	P5	Oct 10, 2001
14	17	cluster_2	224.000	14	2.0	All	1	6	normal	P5	Oct 10, 2001

Abbildung 51: Datensatz zur Berechnung des Entscheidungsbaums

Nachdem das Cluster als Label ausgewählt wurde werden die Daten in Trainings- und Testdaten aufgeteilt. Durch mehrfaches Ausprobieren verschiedener Aufteilungen hat sich herausgestellt, dass bei 50% Trainings- und 50% Testdaten das beste Ergebnis erzielt wird.

Es folgt nun die Konfiguration des „Decision Tree“ Operators. Hierbei wurde eine maximale Baumtiefe von 4 Ebenen festgelegt, da sich ab dieser Tiefe die Performance des Baums nicht mehr verbessert. Trotz des aktivierten Prunings entsteht ein sehr komplexer Entscheidungsbaum, der zu umfassend ist um hier im Detail auf diesen einzugehen. Es ist jedoch zu erkennen, dass zuerst anhand des Attributs „bug-severity“ differenziert wird. Hat ein Datensatz den Wert „major“ oder „blocker“, so wird der Bug direkt dem Cluster 2 zugeordnet, das einer kurzen Bearbeitungszeit entspricht. Hat der Datensatz den Wert „critical“ oder „minor“ so wird nachfolgend anhand der Versionsnummer differenziert. Bei „enhancement“ oder „normal“ ist die Priorität ausschlaggebend. Ist der Wert „trivial“ so wird anhand des Betriebssystems und anschließend anhand der Priorität kategorisiert.

5 Evaluation

Die Evaluation erfolgt mithilfe des Performance-Operators. Der Output ist in Abbildung 52 dargestellt.

accuracy: 85.16%				
	true cluster_2	true cluster_0	true cluster_1	class precision
pred. cluster_2	4239	343	341	86.11%
pred. cluster_0	9	3	3	20.00%
pred. cluster_1	33	13	16	25.81%
class recall	99.02%	0.84%	4.44%	

Abbildung 52: Performance des Entscheidungsbaums

Es ist zu erkennen, dass das Cluster 2 mit sehr hoher Zuverlässigkeit von 86,11% vorhergesagt werden kann. Die Cluster 0 und 1 hingegen können nur in 20% bzw. in 25,81% der Fälle richtig kategorisiert werden. Daraus folgt, dass von diesem Modell kurze Bearbeitungszeiten relativ genau vorhergesagt werden können. Hierzu ist allerdings zu sagen, dass in Cluster 2 auch die meisten Datensätze enthalten waren. Somit kann das Modell zum einen deutlich besser lernen, wenn ein Bug dem Cluster 2 zuzuordnen ist, da hierzu mehr Trainingsdaten vorhanden sind. Zum anderen ist der Aspekt des Zufalls nicht zu vergessen. Aufgrund der deutlich höheren Anzahl an Bugs in Cluster 2 steigt auch die Wahrscheinlichkeit, dass ein Bug diesem Cluster durch Zufall richtig zugeordnet wurde.

6 Deployment

Dieser erarbeitete Prozess kann in der Praxis dafür eingesetzt werden, den Eclipse-Anwendern eine grobe Einschätzung dafür zu geben, wie lange und ob es sich überhaupt lohnt auf eine Lösung für den geposteten Bug zu warten. Zwar kann das Modell keine genaue Angabe über die voraussichtliche Bearbeitungsdauer machen, es kann jedoch voraussagen ob und mit welcher Wahrscheinlichkeit der Bug in den nächsten vier Monaten gelöst wird.

Zusammenfassung und Ausblick

Die Ergebnisse der Beiträge zeigen, dass durch *Bugtracker Information Mining* die Abläufe innerhalb der Softwareentwicklung durchaus optimiert werden können. Das Potenzial hängt vom konkreten Einsatzszenario ab. Die vorgestellten Prototypen stellen anschaulich Ansätze für mögliches Verbesserungspotenzial dar.

Neben dem inhaltlichen Ergebnis sollen nachfolgend vor allem die Vorteile der Veranstaltungsform hervorgehoben werden. Die Studierenden haben im Rahmen der Veranstaltung „Management Support Systeme III – Künstliche Intelligenz“ verschiedene Methoden kennengelernt, die im Themenbereich der Künstlichen Intelligenz eingeordnet sind. Zu den Verfahren zählen unter anderem das Fallbasierte Schließen, Künstliche Neuronale Netze oder Text Mining. Das Konzept jeder Methode wurde im Detail vorgestellt, sodass eine theoretische Fundierung vorhanden ist. Des Weiteren konnten die Studierenden das theoretische Konzept anhand von Übungsaufgaben mit einer aktuellen Software anwenden und somit zusätzliches Anwendungswissen aufbauen. In einem KI-Praktikum wurde das erlernte Wissen abschließend auf die Probe gestellt. Die Studierenden erhielten einen umfangreichen Datensatz, eine Software und eine grobe Aufgabenstellung. Eine genaue Aufgabenstellung wurde von den Studierenden selbst definiert, sodass ein kreativer Freiraum bereits in der Zielsetzung gegeben war. Dies wurde konsequent fortgeführt, sodass die Studierenden selbstständig die passenden Methoden zur Erreichung ihres Ziels auswählen und kombinieren mussten. Nach der sechswöchigen Bearbeitungszeit konnten alle Gruppen einen funktionstüchtigen Prototyp zur Demonstration ihrer Ideen präsentieren.

Diese Veranstaltungsform fordert und fördert die Studierenden bei der Vertiefung und Vernetzung ihres Wissens. Das Feedback der Studierenden ist durchweg positiv, wobei vor allem das Experimentieren mit einem Prototyp und somit die praktische Anwendung des theoretischen Wissens gelobt wird.

Erneut haben wir uns entschlossen, die guten bis sehr guten Ergebnisse des KI-Praktikums in Form dieses Sammelbandes zu veröffentlichen. Eine Fortführung der Veranstaltung sowie des Sammelbandes ist fest geplant.

Osnabrück, im September 2016

Prof. Dr.-Ing. Bodo Rieger

Axel Benjamins