Gottfried Wilhelm
Leibniz Universität Hannover
Fakultät für Elektrotechnik und Informatik
Institut für Praktische Informatik
Fachgebiet Software Engineering

Entwicklung eines Tools zur Feedback-Visualisierung

Bachelorarbeit

im Studiengang Technische Informatik

von

Ben Luca Jacobsen

Prüfer: Prof. Dr. Kurt Schneider
Zweitprüfer: Dr. Jil Klünder
Betreuer: M. Sc. Larissa Chazette und
M. Sc. Wasja Brunotte

Hannover, 09.03.2021

Erklärung der Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbständig und ohne fremde Hilfe verfasst und keine anderen als die in der Arbeit angegebenen Quellen und Hilfsmittel verwendet habe. Die Arbeit hat in gleicher oder ähnlicher Form noch keinem anderen Prüfungsamt vorgelegen.

Hannover, den 09.03.2021

Ben Luca Jacobsen

Zusammenfassung

In der Softwareentwicklung ist das Feedback von Endnutzern eine wichtige Informationsquelle. Durch die Analyse von Nutzer-Feedback können Requirements Engineers Bedürfnisse von Nutzern in die Entwicklung einfließen lassen. Allerdings ist die Analyse großer Mengen an Feedback zu aufwendig, um sie manuell durchzuführen. Eine Automatisierung der Analyse ist daher sinnvoll.

Um Requirements Engineers zu unterstützen, habe ich ein grafisches Tool zur Analyse von Nutzer-Feedback entwickelt. Das Tool unterstützt dabei, Erklärungsbedarf bei Endnutzern zu identifizieren, der im Feedback enthalten ist. Das Nutzer-Feedback liegt hierbei in Form von App-Reviews vor. Dieses Feedback wird mithilfe von Techniken aus dem Bereich der künstlichen Intelligenz automatisch analysiert. Entsprechende Textsegmente, die Erklärungsbedarf enthalten, werden dabei identifiziert. Indem Requirements Engineers Textsegmente manuell kennzeichnen, kann die automatisierte Analyse unterstützt und weiter verbessert werden.

Gefundener Erklärungsbedarf wird im Tool durch Hervorhebung im Text visualisiert. Um die Bedienung des Tools zu erleichtern, standen Aspekte der Usability bei der Entwicklung mit im Vordergrund.

Durch den verwendeten Ansatz können viele Fälle von Erklärungsbedarf identifiziert werden. Das implementierte Tool kann Requirements Engineers unterstützen, für Nutzer besser verständliche Systeme zu entwickeln.

Abstract

End user feedback is an important source of information in software development. By analysing user feedback, requirements engineers can identify and incorporate users' needs into the development process. However, analysing large sets of user feedback manually is too time-consuming. Therefore, automating the feedback analysis is meaningful.

To support requirements engineers I developed a graphic-based application for analysing user feedback. The focus of this tool is to support the identification of users' need for explanation contained in feedback from app reviews. An automatic analysis is implemented by using AI techniques able to recognise explanation need in feedback texts through reoccurring phrases. Requirements engineers can improve classification by manually tagging further corresponding text parts, improving the analysis.

If an explanation need is detected, the identified text is highlighted for better visualization. In order to make the tool easier to use, usability aspects were a key focus during development.

Many cases of explanation need could be identified by using the approach presented in this work. The implemented application can support requirements engineers in developing more understandable systems.

Inhaltsverzeichnis

Kapitel 1

Einleitung

„Beim Programmieren geht es doch nur darum, das umzusetzen, was der Kunde möchte. Das ist ja ganz simpel.“ Jemand, der nicht viel Erfahrung mit Software-Entwicklung hat, mag diese Aussage für richtig halten. Doch jedem erfahrenen Software-Entwickler ist bewusst: Wenn es darum geht für einen Kunden ein Software-System aufzusetzen, dann ist eine der schwierigsten Fragen folgende: Was möchte der Kunde? Das liegt daran, dass eine besondere Herausforderung darin besteht, die Anforderungen des Kunden korrekt zu erfassen [15, S. 17-18].

Nehmen wir einmal an, ein Kunde möchte einen Online-Shop für sein Kleidungsgeschäft aufsetzen. Er hat eine gewisse Vorstellung, wie dieser Online-Shop auszusehen hat. Außerdem weiß er ganz genau, was sein Unternehmen bisher für Abläufe und Informationen gebraucht hat, um Ware im Laden verkaufen zu können. Allerdings hat er keinerlei Erfahrung mit den Möglichkeiten von Software, geschweige denn, wie man diese umsetzt.

Auf der anderen Seite steht ein Entwickler, der den Auftrag für einen Online-Shop bekommt. Er weiß genau, wie er mit Software das umsetzen kann, was er möchte. Allerdings hat er keine Erfahrung mit Kleidungsgeschäften und kennt deren Abläufe nicht, geschweige denn die speziellen, die der Kunde in seiner Firma verwendet.

Das Problem liegt auf der Hand: Die beiden Seiten werden aneinander vorbeireden. Es besteht eine Lücke zwischen Entwickler und Kunden. Es wird auch von einer „Symmetry of Ignorance“ (nach Fischer [2]) gesprochen. Ein Unwissen (engl. Ignorance) auf beiden Seiten führt zu Missverständnissen und Frustration. Um das Problem anzugehen, müssen zwei Fragen geklärt werden.

1. Wie kann der Kunde dem Entwickler verständlich machen, was er möchte?

2. Wie kann der Entwickler dem Kunden verständlich machen, was technisch möglich ist, und verstehen, was der Kunde möchte?

Um die Lücke zwischen Entwickler und Kunden zu verkleinern, hat sich das Gebiet des Requirements Engineering (siehe Abschnitt 2.1) gebildet. Hier werden systematisch Anforderungen erhoben und verwaltet, die das gewünschte Software-Produkt beschreiben. Somit versteht der Entwickler, was der Kunde möchte, und der Kunde bekommt sein gewünschtes Produkt.

1.1 Problemstellung

Allerdings ist es nicht immer ein Kunde, dessen Wünsche berücksichtigt werden müssen. Auch die Wünsche von Endnutzern sind wichtig. Um die Bedürfnisse der Endnutzer zu verstehen, wird häufig Nutzer-Feedback gesammelt. Da die Mengen von Nutzer-Feedback sehr groß werden können, ist es kaum möglich, das gesamte Feedback manuell in Betracht zu ziehen. Daher ist es nützlich die Analyse weitestgehend zu automatisieren.

Ein Teil des Requirements Engineering ist das Klassifizieren von Anforderungen (siehe Abschnitt 2.1). Durch eine Aufteilung in Klassen können gegebene Anforderungen besser bearbeitet werden, denn Entwickler können sich auf bestimmte Klassen spezialisieren (Sicherheit, Performance, ...). Außerdem ist so besser ersichtlich, worauf der Kunde einen Schwerpunkt legt. Möchte man nun automatisch Nutzer-Feedback analysieren, ist es sinnvoll, dass auch die Klassifizierung automatisch abläuft. Somit kann ein Requirements Engineer eine Klasse auswählen und sich rein mit Nutzer-Feedback dieser Klasse beschäftigen. Doch wie kann ein Algorithmus die oft sehr nuancierten Anforderungsklassen voneinander abgrenzen?

1.2 Lösungsansatz und Zielsetzung

Das Nutzer-Feedback liegt oft im Textformat vor, so zum Beispiel Reviews in App-Stores. Um Nutzer-Feedback automatisch im Kontext von Anforderungsklassen zu analysieren, braucht es eine Möglichkeit, Sprache automatisch zu verarbeiten. Da die Anforderungsklassen sich oft nur in Nuancen unterscheiden, sollte diese Sprachverarbeitung hochwertig sein. Es sollte ihr möglich sein, Sprache menschenähnlich zu handhaben.

Aus dem Forschungsgebiet des Machine Learning kommt der Ansatz des Natural Language Processing (siehe Abschnitt 2.3.3). Damit ist es möglich, Sprache in Kategorien einzuordnen, die aus der Linguistik bekannt sind. So kann man Satzteile in Sätzen erkennen und auch Worte grammatikalisch analysieren. Darauf aufbauend kann man nun aufgrund von manuell klassifizierten Daten ein System trainieren. Man füttert es pro Anforderungsklasse mit Feedback, das auf eben diese Klasse hindeutet. Daraus lernt das System. Das System weiß nun zu bewerten, ob bestimmtes Nutzer-Feedback in eine Anforderungsklasse fällt.

In dieser Arbeit wird versucht, das Problem anhand einer bestimmten Anforderungsklasse anzugehen. Das Ziel liegt darin, ein System aufzusetzen, das Nutzer-Feedback zu einer gewissen Güte einordnen kann. Dabei liegt der Fokus auf der Anforderungsklasse Erklärbarkeit (siehe Abschnitt 2.2). Requirements Engineers wird die Möglichkeit gegeben, zu Apps aus zwei bekannten App Stores Nutzer-Feedback zu sammeln. Hierbei liegt der Fokus auf Nutzer-Feedback aus englischsprachigen App Stores. Dieses gesammelte Feedback kann daraufhin automatisch analysiert werden. Die Requirements Engineers können allerdings auch selbst auf das System Einfluss ausüben und es trainieren. Dadurch kann es sich iterativ verbessern und zunehmend genauer analysieren.

Für einen effizienten Workflow ist es sinnvoll, wenn gefundene Stellen im Text visualisiert werden. So können Requirements Engineers auf einen Blick sehen, wo im Text die Anforderung gegeben worden ist. Um dieses Ziel zu erreichen, wird ein grafisches Tool entwickelt.

1.3 Struktur der Arbeit

In den nachfolgenden Kapiteln wird beschrieben, wie ich das Problem konkret angegangen habe, auf welche Probleme ich gestoßen bin und welche Erkenntnisse für weiterführende Arbeiten wichtig sein könnten.

In Kapitel 2 werden die wissenschaftlichen Grundlagen erklärt, auf die in dieser Arbeit Bezug genommen werden. Kapitel 3 führt andere wissenschaftliche Arbeiten an, die sich mit dem gleichen Thema beschäftigen. Auf ihren Nachforschungen baut meine Arbeit auf. In den Kapiteln danach gehe ich auf meine grundlegenden Ansätze 4 ein sowie auf meine konkrete Implementierung des Systems 5. Es wird außerdem beschrieben, welche Probleme bei der Analyse aufgetreten sind, und bewertet, wie gut das implementierte System funktioniert 6. In Kapitel 7 wird ein Ausblick für weiterführende Arbeiten gegeben.

Kapitel 2

Grundlagen

In diesem Kapitel werden die Grundlagen vermittelt, die zum weiteren Verständnis der Arbeit notwendig sind. Es wird dabei zunächst auf das Requirements Engineering eingegangen, danach auf die Anforderungsklasse Erklärbarkeit und zuletzt werden einige Techniken der künstlichen Intelligenz erläutert, insbesondere das Natural Language Processing.

2.1 Requirements Engineering

Das Requirements Engineering spielt sowohl vor als auch nach der Entwurfsphase von Software eine große Rolle. Stanik schreibt, das Requirements Engineering sei der Prozess zu entscheiden, was tatsächlich entwickelt werden soll [15, S. 16-17]. Einer formelleren Definition nach ist es der Prozess des Formulierens, Dokumentierens und systematischen Wartens von Anforderungen bezüglich Software [15, S. 16-17]. Das Requirements Engineering ist unter anderem verantwortlich dafür, Anforderungen systematisch zu erheben und nachzuverfolgen. Dabei geht es sowohl um die Zeit des Entwurfs vor Inbetriebnahme der Software als auch um die Zeit danach.

Die Anforderungen stammen von Stakeholdern der Software. Als Stakeholder bezeichnet man eine Gruppe von Individuen, die dabei mitwirkt, die Anforderungen eines Systems festzulegen [15, S. 16]. Das wären beispielsweise Kunden, Mitarbeiter oder auch Endnutzer der Software.

Im Requirements Engineering unterteilt man die Anforderungen der Stakeholder in die Klassen funktionale und nicht-funktionale Anforderungen [15, S. 16](siehe Abb. 2.1). So wären konkrete Funktionen der Software (z.B. „zeigt die aktuelle Uhrzeit an“) in die Klasse der funktionalen Anforderungen einzuordnen. Anforderungen an die Usability oder Erklärbarkeit (siehe Abschnitt 2.2) fallen in eine Unterklasse der sogenannten nicht-funktionalen Anforderungen, die Qualitätsanforderungen.

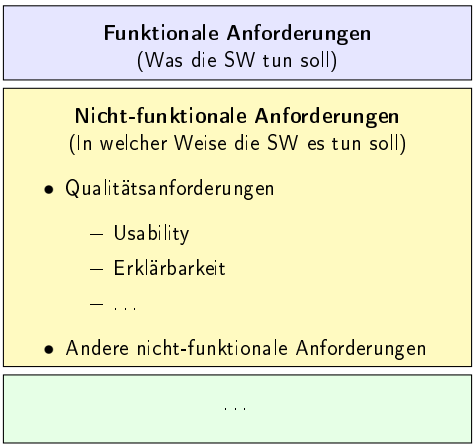

Abbildung 2.1: Klassen von Anforderungen (Abbildung in Anlehnung an [14, S. 90]). Hier werden lediglich zwei Klassen aufgezeigt. Es gibt je nach Definition noch mehr [3][14, S. 90].

2.1.1 Nicht-funktionale Anforderungen

Eine Anforderungsklasse im Requirements Engineering sind die nicht-funktionalen Anforderungen (engl. non-functional requirement, kurz NFR).

Die nicht-funktionalen Anforderungen haben, wie der Name schon sagt, nichts mit den eigentlichen Funktionen der Software zu tun. Glinz definiert sie folgendermaßen: Sie seien ein Attribut eines Systems oder eine Bedingung an ein System [3][15, S. 16]. Sie beschreiben nicht, was die Software kann, sondern in welcher Weise (schnell, flexibel, schön, ...) die Software etwas tun soll [3].

2.1.2 Nutzer-Feedback

„Ein Software-System muss die Bedürfnisse von Endnutzern erfüllen, um weitgehend akzeptiert und genutzt zu werden.“ (frei übersetzt nach Gärtner und Schneider [7]). Durch Feedback kommunizieren Nutzer ihre Bedürfnisse und bewerten, was an einem Software-System ihre Bedürfnisse erfüllt und was nicht. Nutzer-Feedback gibt Requirements Engineers die Möglichkeit, die Perspektive der Endnutzer zu sehen. So können sie ihre Bedürfnisse als Anforderungen mit in den Entwicklungsprozess einfließen lassen. Wenn das Software-System diese Anforderungen berücksichtigt, können die Bedürfnisse der Nutzer noch besser erfüllt werden.

Um an hilfreiches Nutzer-Feedback zu kommen, gibt es viele Ansätze. Mehr und mehr sehen Requirements Engineers Potenzial in Feedback aus Online-Plattformen wie App-Stores [6]. Diese Datenquellen bieten große Mengen an Nutzer-Feedback in Form von App-Reviews. Allerdings bilden diese Reviews kein umfassendes Feedback zu allen Aspekten der Software.

Groen et al. [6] erkennen, dass es in vielen Reviews stets um die gleichen Schwerpunkte geht. Viele Aspekte, die ein Software-System ausmachen, bekommen so kein Feedback und können daher auch nicht davon profitieren. Beispielsweise würden Reviews keine Auskunft über Aspekte einer Software geben, die der Nutzer nicht sehen kann [6]. Zusätzlich gibt es viele Reviews ohne hilfreiche Information. Stanik schreibt, dass mehr als 70% der Reviews nicht mit Requirements Engineering zusammenhängen [15, S. 96].

2.2 Erklärbarkeit

Erklärbarkeit gehört zu der Klasse der nicht-funktionalen Anforderungen. Es kann definiert werden als das Ausmaß, zu dem ein System Erklärungen für seine Entscheidungen oder Ausgaben liefern kann [1, 16]. Das bedeutet, dass Nutzer durch diese Erklärungen nachvollziehen können, warum etwas passiert ist oder nicht. Haben Nutzer beim Verwenden eines Systems Erklärungsbedarf, dann fehlen Erklärungen.

Das Konzept Erklärbarkeit überschneidet sich in einem Aspekt stark mit zwei anderen NFRs: Transparenz und Interpretierbarkeit. Transparenz hängt damit zusammen, dass der Nutzer die inneren Mechanismen eines Systems verstehen kann. Interpretierbarkeit besagt, zu welchem Grad ein Nutzer gegebene Informationen und Erklärungen verstehen kann [1]. Ein Nutzer kauft zum Beispiel in einem Online-Shop ein und es werden ihm Artikel vorgeschlagen. Wenn er nicht sehen kann, warum ihm eben diese Artikel vorgeschlagen werden, dann besteht hier Erklärungsbedarf.

Aber Erklärbarkeit hat auch noch einen anderen Aspekt. So tragen Erklärungen (beispielsweise Tutorials) dazu bei, dass ein Nutzer schnell mit Software umzugehen lernt. Außerdem bleiben ihm ohne Erklärungen möglicherweise Funktionen unbekannt. Diese Eigenschaft überschneidet sich wiederum mit der NFR Usability [1], welches sich darum dreht, wie gebrauchstauglich ein System ist. Im Beispiel des Online-Shops will der Nutzer nun einen Kauf tätigen. Wenn ihm nicht ersichtlich ist, wie er das tun kann und ihm keine Erklärung gegeben wird, dann besteht hier Erklärungsbedarf.

Zusammenfassend kann gesagt werden, dass Erklärbarkeit aus zwei Teilaspekten besteht. Der erste überschneidet sich mit Transparenz und Interpretierbarkeit, während der zweite sich mit Usability überschneidet.

2.2.1 Analyse von Erklärbarkeit

Die Entscheidung, ob die Aussage eines Stakeholders eine Anforderung enthält, die sich auf Erklärbarkeit bezieht, ist nicht einfach zu treffen. Erklärbarkeit überschneidet sich mit mehreren NFRs, die untereinander wenig gemeinsam haben. Es kann daher passieren, dass man irrtümlich Anforderungen als Erklärungsbedarf klassifiziert, die eigentlich nur zu

Usability oder nur zu Transparenz gehören (siehe hierzu auch das Beispiel in Abschnitt 4.4.1).

Außerdem kann Erklärungsbedarf explizit oder implizit in einer Aussage vorkommen [8, S. 25]. Im expliziten Fall ist direkt ersichtlich, dass es sich um Erklärungsbedarf handelt, denn es wird *explizit* darauf hingewiesen. Beispiele sind „Ich finde das nicht“ oder „Ich brauche eine Erklärung“. Im impliziten Fall verbirgt sich hinter der Aussage ein Unverständnis, welches durch Erklärungen aufgeklärt werden könnte, so zum Beispiel ein Review zu „Google Maps“: „Jedes Mal, wenn wir zum Tanken anhalten, ändert es die Strecke, wie es will.“ (Beispiel aus [8, S. 25]). Die Person benötigt eine Erklärung aber drückt dies *implizit* aus. Diese Art von Erklärungsbedarf ist deutlich schwieriger zu erkennen.

2.2.2 Sind Erklärungen immer sinnvoll?

Ob Erklärungen sinnvoll sind oder nicht, ist stark fallabhängig. In kritischen Systemen, beispielsweise einem autonomen Fahrzeug, ist Erklärbarkeit sehr wichtig, insbesondere der Teilaspekt von Erklärbarkeit, der sich mit Transparenz überschneidet. Durch Transparenz kann ein Mitfahrer den Entscheidungen des Fahrzeugs vertrauen lernen. Wenn das Fahrzeug plötzlich die Route ändert und keine Erklärung dazu liefert, kann es sein, dass der Mitfahrer dem Fahrzeug nicht mehr vertraut.

In anderen Systemen, zum Beispiel in der Navigations-App „Google Maps“, brauchen Nutzer nicht immer Erklärungen. Es verhält sich sogar so, dass zu viele Erklärungen stören und somit die Usability negativ beeinflussen können. Wenn ein Nutzer „Google Maps“ verwendet und zu viele Erklärungen auf seinem Smartphone angezeigt werden, kann er die Strecke möglicherweise nicht mehr gut erkennen. Chazette und Schneider bezeichnen dies als ein zweischneidiges Schwert [1].

Erklärbarkeit ist aber nicht nur optional. Durch Datenschutz-Gesetze wird es zunehmend notwendig, Erklärungen im Rahmen von Transparenz liefern zu können [1]. Goodman und Flaxman argumentieren, dass es bald zur Pflicht werden könnte, wenn es ein „Recht zu Erklärungen“ gibt [5].

2.3 Künstliche Intelligenz

Das Forschungsgebiet der künstlichen Intelligenz (KI) hat in den letzten Jahren viel Fortschritt erfahren und findet seither mehr und mehr Einsatz. Auch früher gab es schon Ansätze für KI. Hier hat man oft eine für Menschen schwierig zu lösende oder aufwendige Aufgabe durch Computer lösen lassen. „Die wahre Herausforderung an eine künstliche Intelligenz besteht jedoch darin, Aufgaben zu erledigen, mit denen Menschen keinerlei Probleme haben, obwohl sie schwierig in Worte zu fassen sind.“ (frei übersetzt nach Goodfellow et al. [4, S. 1])

Solange eine Aufgabe komplett definiert ist und in einer sterilen und formalen Umgebung stattfindet, kann ein Computer sie leicht lösen [4, S. 2-3]. Sie mag für Menschen schwierig sein, aber nicht für einen Computer. Sobald es aber um Aufgaben geht, die man nur mit informalem und nicht eindeutigen Wissen lösen kann, geht ein Computer in die Knie.

2.3.1 KI, Machine Learning und Deep Learning

Um sich subjektiven Problemen anzunähern, braucht man einen neuen Ansatz. Dieser neue Ansatz sollte kein wissensbasierter sein, bei dem Menschen per Hand alles formal definieren müssen. Der Ansatz des Machine Learning bietet diese Eigenschaft. Machine Learning bedeutet, dass ein System in der Lage ist, eigenes Wissen zu erwerben, indem es aus Rohdaten Muster extrahiert [4, S. 3]. Durch diesen Lernprozess kann das System scheinbar subjektive Entscheidungen treffen.

Einfache Machine-Learning-Algorithmen stützen sich auf vordefinierte Merkmale. Um eine Aufgabe zu lösen, wird untersucht, inwieweit die Merkmale untereinander korrelieren. Nehmen wir als Beispiel die Aufgabe, an der Stimme zu erkennen, um was für einen Sprecher es sich handelt (Beispiel angelehnt an [4, S. 4]). Hier könnte ein Merkmal das Volumen seines Stimmapparats sein. Anhand dieses Merkmals lässt sich einschätzen, ob der Sprecher ein Mann, eine Frau oder ein Kind ist.

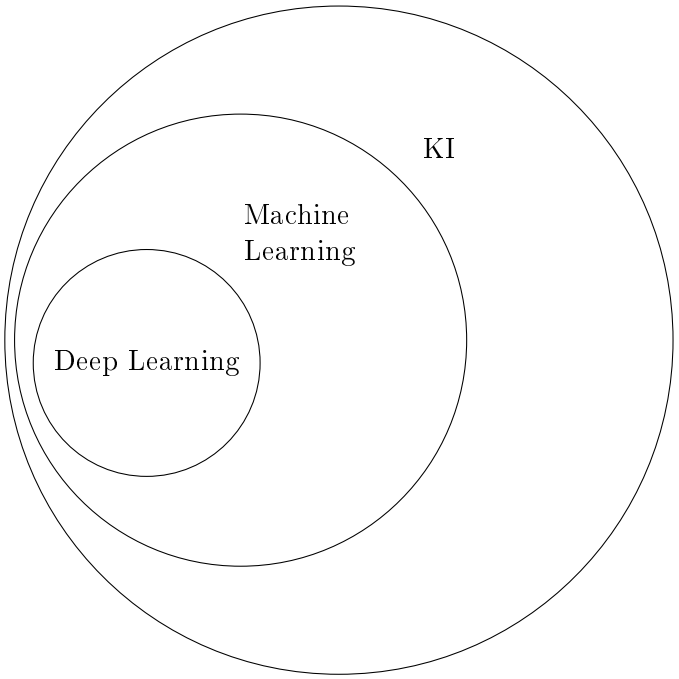

Abbildung 2.2: Unterbereiche von KI (Abbildung adaptiert von [4, S. 10])

Allerdings ist nicht immer klar, welche Merkmale wichtig sind. So ist in der Objekterkennung nicht immer klar, welche Merkmale ein Objekt ausmachen. Einen Schritt weiter geht das Deep Learning, welches ein Unterbereich des Machine Learning ist (siehe Abb. 2.2). Hier müssen nicht mehr per Hand Merkmale definiert werden, sondern der Algorithmus bestimmt seine Merkmale selbst. Der Ansatz des Deep Learning kann komplexe Konzepte erlernen, indem es diese aus einfacheren zusammensetzt [4, S. 2]. Das Konzept einer Person in einem Bild kann durch einen Deep-Learning-Algorithmus erkannt werden, indem es in geschachtelte einfachere Konzepte aufteilt wird. In einem Bild fängt es beispielsweise mit einfachen Pixeln an, dann Kanten, dahinter Umrisse von Objekten. Aus diesen einfachen Konzepten schließt der Algorithmus auf Objekte und gibt aus: Es handelt sich um eine Person.

Diese automatische Schachtelung des Deep Learning macht es möglich, viele subjektive Aufgaben zu lösen. Solange man genug Trainingsdaten hat, kann man den Algorithmus zum Lösen dieser Aufgaben trainieren. Seit 2016 gibt es eine Faustregel, dass ein solcher Algorithmus annehmbare Ergebnisse liefert, wenn er mit etwa 5000 Trainingsdaten gefüttert wurde [4, S. 24]. Je mehr Trainingsdaten es gibt, desto besser sind die Ergebnisse.

2.3.2 Metriken

Im Bereich KI tauchen ein Paar Metriken immer wieder auf. Diese Metriken werden verwendet, um die Güte einer KI-Implementation zu beschreiben. Hier wird auf 3 Metriken eingegangen: Precision, Recall und F1-Score. Die Informationen und Formeln stammen von Manning et al. [11, S. 155-156].

Um das abstrakte Konzept dieser Metriken zu verstehen, wird es in einem Beispiel veranschaulicht. Nehmen wir an, ein KI-Algorithmus soll Bilder erkennen, auf denen Personen zu sehen sind. Er erkennt nun aus einer Gesamtmenge G an Bildern eine Untermenge U, bei denen er glaubt, dort seien Personen vorhanden. Nun wollen wir aber untersuchen, wie gut die Personenerkennung ist. Wir analysieren händisch alle G Bilder und kommen auf eine tatsächliche Menge T an Bildern, auf denen Personen zu sehen sind.

Die Metrik *Precision* beschreibt das Verhältnis von richtig erkannten zu allen erkannten Fällen. Im Beispiel bestimmt die *Precision* das Verhältnis richtig erkannter Bilder zu allen Bildern der Menge U. Sie beantwortet also die Frage, wie rein das Ergebnis der KI ist. Dabei ist es egal, wie viele Fälle U enthält. Vielleicht hat sie nur ein Bild erkannt, aber dafür korrekt. Somit wäre die Precision 1 (Maximalwert).

$$Precision = \frac{\#_{relevante\ entnommene\ Elemente}}{\#_{entnommene\ Elemente}} \overset{\text{Bsp.}}{=} \frac{\#_{richtige\ Bilder\ in\ U}}{\#_{Bilder\ in\ U}} \tag{2.1}$$

Bei der Metrik *Recall* hingegen geht es um das Verhältnis richtig erkannter Fälle zu allen richtigen Fällen. Im Beispiel bestimmt der *Recall*

einen Wert, der aussagt, wie viele der T richtigen Bilder auch in U enthalten sind. Dabei ist es egal, ob U auch viele falsch erkannte Fälle enthält. Der Recall beschreibt also das Maß, zu dem alle T Bilder erkannt werden. Hat die KI beispielsweise alle G Bilder ausgegeben ($U = G$), dann wären auch alle T Bilder enthalten. Demnach wäre der Recall 1 (Maximalwert).

$$Recall = \frac{\#_{relevante\ entnommene\ Elemente}}{\#_{relevante\ Elemente}} \overset{\text{Bsp.}}{=} \frac{\#_{richtige\ Bilder\ in\ U}}{\#_{Bilder\ in\ T}} \tag{2.2}$$

Der *F1-Score* nimmt einen Mittelwert beider Metriken. So können beide Metriken in einem ausgewogenen Maße einbezogen werden.

$$F_1 = \frac{2 \times Precision \times Recall}{Precision + Recall} \tag{2.3}$$

2.3.3 Natural Language Processing

Die Verarbeitung natürlicher Sprache (engl. natural language processing, kurz: NLP) beschreibt den Einsatz menschlicher Sprache durch einen Computer [4, S. 513]. NLP bietet die Möglichkeit, dass Computer gewöhnliche Sprachen wie Deutsch oder Englisch verstehen und verwenden.

Verarbeitung

In jeder NLP-Implementierung wird Sprache zuerst in eine Sequenz von Tokens unterteilt. Ein Token kann je nach Modell für ein Wort, ein Zeichen oder sogar ein Byte stehen [4, S. 514]. Den Vorgang des Aufteilens in Tokens nennt man Tokenisierung. Einige NLP-Bibliotheken bieten die Möglichkeit, die Tokens auf linguistische Eigenschaften zu untersuchen und beispielsweise als Satzteile zu erkennen. Dies geschieht häufig aufgrund von vortrainierten Deep Learning-Modellen.

Labeling

Das Labeling (dt. Klassifizierung) ist ein wichtiger Bestandteil im Prozess der Implementierung eines lernfähigen KI-Systems. Mit dem Begriff Labeling bezeichnet man die Aufgabe, zu definieren, in welche von k Kategorien eine Eingabe gehört [4, S. 109]. In einem KI-System kann es sein, dass es nur $k = 2$ Kategorien gibt. Dabei geht es darum, ob eine bestimmte Eingabe in das KI-System eine positive oder negative Ausgabe erzeugt. Im Fall eines NLP-Systems wäre die Eingabe Text und die Ausgabe die Information, ob dieser Text das Gesuchte enthält oder nicht. So wäre ein Beispiel ein NLP-System, das Früchte im Text erkennen kann. Der Satz „Anna isst einen Apfel“ enthält eine Frucht und würde somit eine positive Ausgabe erzeugen.

Bei KI-Systemen geht es dabei oft um sehr große Datensätze, die mit Labeling-Information versehen sind [4, S. 21-24]. Mithilfe dieser gelabelten Datensätze ist es möglich, einen Deep-Learning-Algorithmus zu trainieren.

Part-of-speech tags

Um die englische Sprache lexikographisch und grammatikalisch zu definieren, werden Satzteile (engl. parts of speech, kurz POS) gekennzeichnet. Das Projekt Universal Dependencies[1] hat die „Universal part-of-speech tags“ (kurz: UPOS tags) eingeführt [12]. Beispielsweise werden Adjektive mit dem UPOS tag „ADJ“ gekennzeichnet. Einige NLP-Bibliotheken gebrauchen diese Einordnung. Abbildung 2.3 zeigt eine vollständige Auflistung der vorhandenen UPOS tags.

ADJ	adjective
ADP	adposition
ADV	adverb
AUX	auxiliary verb
CONJ	coordinating conjunction
DET	determiner
INTJ	interjection
NOUN	noun
NUM	numeral
PART	particle
PRON	pronoun
PROPN	proper noun
PUNCT	punctuation
SCONJ	subordinating conjunction
SYM	symbol
VERB	verb
X	other

Abbildung 2.3: Liste aller UPOS tags [12]

[1] `https://universaldependencies.org/`

Dependency Relations

Im gleichen Sinne wie POS tags werden auch syntaktische Abhängigkeiten innerhalb von Sätzen gekennzeichnet. Abhängigkeiten können zum Beispiel grammatikalisch sein (Subjekt, Prädikat, ...). Das Projekt Universal Dependencies hat die „Universal Dependency Relations“ [12] definiert. Dort werden 37 syntaktische Abhängigkeiten genannt. Um einen kurzen Einblick zu geben, folgt in Abbildung 2.4 ein Beispiel.

Text	Dependency Rel.	Erklärung
I	nsubj	nominal subject $\hat{=}$ Subjekt
like	root	Wurzel des Satzes, hier Prädikat
artificial	amod	adjectival modifier $\hat{=}$ Adverb
intelligence	obj	object $\hat{=}$ Objekt
.	punct	punctuation $\hat{=}$ Satzzeichen

Abbildung 2.4: Einordnung des Satzes „I like artificial intelligence.“ in Universal Dependency Relations [12]

Kapitel 3

Verwandte Arbeiten

Diese Arbeit ist eine von vielen zum Thema der automatischen Feedback-Klassifizierung mit dem Schwerpunkt Erklärbarkeit. In den letzten Jahren hat die Bedeutung von App-Reviews und der NFR Erklärbarkeit zugenommen. Daher gibt es viel aktuelle Forschung zu diesem Thema.

Die Arbeit von Chazette und Schneider *„Explainability as a non-functional requirement: challenges and recommendations"* [1] definiert Erklärbarkeit als NFR und gibt Vorschläge zur Analyse. Kuhnke [8] beschäftigt sich mit der automatischen Identifizierung von Erklärungsbedarf, indem Nutzer-Feedback analysiert wird. Dabei beschränkt er sich auf Nutzer-Feedback aus dem Apple App Store und Google Play Store. Das Ergebnis ist, dass Erklärungsbedarf selten in Reviews vorkommt, nur in ca. 5.6% [8, S. 67].

Stanik [15] schlägt Konzepte vor, wie Requirements Engineering verbessert werden kann. Er definiert das System der Requirements Intelligence, das sich mit der Nutzung von Nutzer-Feedback aus Social-Media-Plattformen und App Stores auseinandersetzt. Durch automatische Klassifizierung von Nutzer-Feedback zu Anforderungsklassen wird Requirements Engineers geholfen. Mithilfe eines grafikbasierten Tools wird der Ansatz der Requirements Intelligence umgesetzt.

Maalej und Nabil [9] zeigen Herausforderungen bei der Analyse von Nutzer-Feedback auf. Sie beziehen sich beim Feedback speziell auf Reviews aus App Stores. Maalej und Nabil ziehen Schlüsse für das Design von zukünftigen Review-Analyse-Tools.

Diese Arbeit baut auf der Forschung einiger Arbeiten auf und verwendet zum Teil deren Programmteile oder Konzepte. So wurde das von Kuhnke entwickelte Tool zum Einlesen von Reviews aus App Stores in das Software-System integriert. Wie in der Arbeit von Stanik wurde auch hier ein Tool für Requirements Engineers [15, S. 49-52] entwickelt. Außerdem implementiert das entwickelte Programm einen Teil eines Nutzer-Feedback-Analyse-Frameworks und lehnt damit an Staniks Ansatz der Requirements Intelligence [15, S. 83].

Allerdings geht diese Arbeit bei der Implementation des NLP-Ansatzes einen anderen Weg als viele der genannten Arbeiten: Sie nutzt reines Machine Learning und kein Deep Learning. Außerdem wird zusätzlich zur KI-Implementierung eine intuitive GUI geboten.

Kapitel 4

Konzepte

In diesem Kapitel soll es um die grundlegenden Konzepte dieser Arbeit gehen. Sie bilden die Schwerpunkte der Systemimplementierung.

4.1 Unterstützung von Requirements Engineers durch hilfreichen Workflow des Tools

Das Ziel des zu implementierenden Tools ist die Unterstützung von Requirements Engineers bezüglich der Klassifizierung von Nutzer-Feedback. Sie werden die Hauptnutzer des Tools sein und werden im Nachfolgenden „Nutzer“ genannt. Um sie bestmöglich zu unterstützen, wird ein Ansatz gesucht, der ihnen einen möglichst hilfreichen Workflow bietet.

Dieser Workflow sollte es den Nutzern ermöglichen, fein festzulegen, welches Nutzer-Feedback analysiert wird. Außerdem sollte eine automatische Analyse möglich sein, deren Ergebnisse im Nachhinein durch die Nutzer verifiziert und verbessert werden können. Zur Verifikation und manuellen Analyse sollten eine visuelle Anzeige von Nutzer-Feeback und eine GUI mit hoher Usability zur Verfügung stehen. Zuletzt ist es wichtig, dass die automatische Analyse durch Nutzerinteraktion verbessert werden kann, damit sie zunehmend genauer wird.

Ich wähle folgenden Ansatz (siehe Abb. 4.1): Ein Nutzer kann im ersten Schritt Apps spezifizieren, von denen Reviews gesammelt werden sollen. Im zweiten Schritt kann er nun diese Reviews analysieren, sodass gekennzeichnet wird, wo sich Erklärungsbedarf (siehe Abschnitt 2.2) befindet. Daraufhin kann er in einem dritten Schritt die Ergebnisse der automatischen Analyse verifizieren, wodurch Trainingsdaten generiert werden. Zuletzt kann der Nutzer mithilfe der Trainingsdaten aus der Verifikation das System trainieren, sodass die automatische Analyse verbessert wird. Wird Schritt 2 nun mit der verbesserten Analyse wiederholt, liegen, verglichen mit der vorherigen Analyse, bessere Ergebnisse vor. In den folgenden Abschnitten geht es um Konzepte, die diesen Workflow ermöglichen.

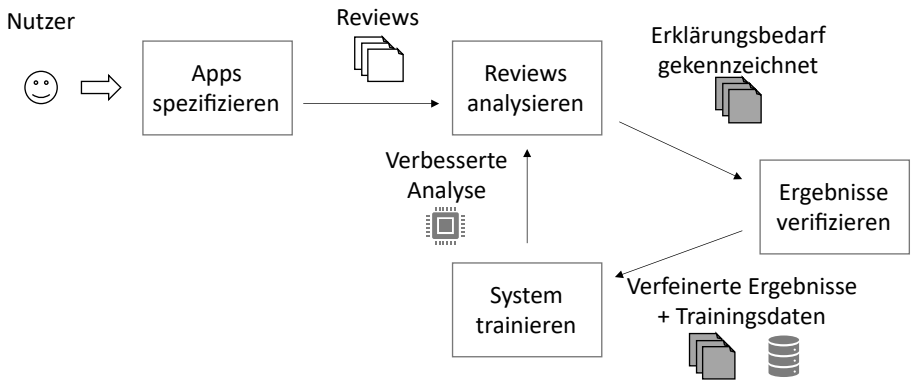

Abbildung 4.1: Schematischer Workflow

4.1.1 Spezifizierung von Apps

In diesem Abschnitt geht es um den ersten Schritt: Apps spezifizieren. Zuerst einmal ist es wichtig, dass die Nutzer auf einfachem Wege Feedback-Informationen einholen können. Diese Informationen müssen gespeichert werden können, damit Arbeit nicht verloren geht. Außerdem soll einstellbar sein, wie viel Feedback zu welcher App aus welchem App Store geholt wird, damit die Nutzer den Umfang ihrer Suche genau einstellen können.

Zu diesem Zweck werden „Projekte“ eingeführt. Ein Projekt stellt eine Suche dar, die beliebig viele sogenannte Suchregeln enthält. Nutzer können ein Projekt erstellen und darin die Suchregeln spezifizieren. Jede Suchregel enthält entweder eine bestimmte App oder eine Kategorie von Apps. Außerdem enthält eine Suchregel Informationen dazu, wie viel Feedback geholt werden soll. Die Projekte werden gespeichert und können auch gelöscht werden.

4.1.2 Analyse und Training

In diesem Abschnitt geht es um die Schritte 2-4: Die Analyse von Reviews, die Verifikation der Ergebnisse und zuletzt das Training.

Es ist wichtig, dass die Analyse automatisch erfolgt, damit die Nutzer nicht alle Reviews anschauen müssen, was sehr zeitintensiv ist. Allerdings ist es auch wichtig, dass sie einen Einfluss auf die Analyse haben können, da automatische Analyse fehleranfällig sein kann. Sie sollen daher einerseits die Möglichkeit haben, die Analyse-Information in analysiertem Feedback zu bearbeiten (Verifikation). Andererseits soll es auch möglich sein, auf die Analyse selbst Einfluss zu nehmen, indem das KI-System durch Nutzer trainiert werden kann.

Die Trainingsdaten ergeben sich aus der Verifikation, indem gespei-

chert wird, welche Textstellen der Nutzer nach der automatischen Analyse angepasst hat. Es wird sowohl gespeichert, welche Kennzeichnungen er hinzugefügt hat, als auch, bei welchen Textstellen er gekennzeichneten Erklärungsbedarf entfernt hat. Diese Trainingsdaten können in Schritt vier verwendet werden, um das System zu trainieren.

Allerdings kann es sein, dass der Nutzer nicht alle der erfassten Trainingsdaten für das Training verwenden möchte. Daher wird ihm die Möglichkeit gegeben, aus den gespeicherten Trainingsdaten die Daten auszuwählen, welche in Schritt 4 zum Training verwendet werden sollen.

4.1.3 Anzeige von Erklärungsbedarf und Interaktion

Für die Anzeige von Erklärungsbedarf (und anderen NFRs, siehe Abschnitt 4.3) eignet es sich, die entsprechenden Stellen im Text durch Markierung hervorzuheben (siehe Abb. 4.2). So ist es leicht ersichtlich, wo Erklärungsbedarf gefunden wurde. Außerdem kann der Nutzer Text selbst markieren und somit nicht automatisch gefundenen Erklärungsbedarf kennzeichnen. Das ist eine intuitive Lösung für die Nutzer. Die durch den Nutzer gekennzeichneten Textstellen können im nächsten Schritt zum Trainieren des KI-Systems verwendet werden.

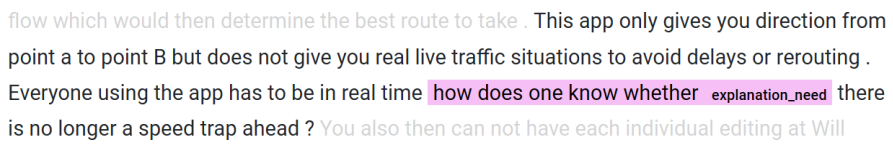

Abbildung 4.2: Hervorhebung von Erklärungsbedarf am Beispiel eines echten App-Reviews zur Navigationsapp „Waze“

4.2 Entscheidungen zur Plattform

Auf welcher Plattform soll einem Nutzer das Programm zugänglich sein - als native Desktop-App auf Windows/Mac/Linux, als Smartphone-App oder als Web-App im Browser? Anhand mehrerer Kriterien wurde die Entscheidung getroffen, eine Web-App zu implementieren. Diese Kriterien werden im Folgenden erläutert.

Da die Nutzer nicht nur ein paar Minuten am Stück, sondern über längere Zeiträume die App verwenden werden, scheidet eine reine Smartphone-App aus. Aufgrund des kleinen Bildschirms sind lange Arbeitsphasen auf einem Smartphone nicht praktikabel. Für den Bau einer nativen Desktop-App müsste ich mich auf ein Betriebssystem beschränken, da der mir gegebene Zeitrahmen nicht für die Anpassung auf mehrere Betriebssysteme ausreicht.

Aufgrund dessen wäre auch die Kompatibilität eingeschränkt. Eine Web-App dagegen hat den Vorteil, dass sofort eine hohe Kompatibilität erreicht wird, da annähernd jedes Gerät einen unterstützten Web-Browser besitzt. Außerdem ist es mit Web-Techniken schnell möglich, eine ansprechende GUI zu implementieren. Ein Contra-Argument ist die Anpassung auf den Nutzer. Ohne eine Anmelde-Möglichkeit ist es per Browser nicht möglich Nutzer zu unterscheiden. Das bedeutet, dass alle Informationen global für alle sichtbar und veränderbar sind, die die Website aufrufen. Bei einer nativen Desktop-App werden Daten lokal auf dem Rechner gespeichert, weswegen automatisch alle Daten benutzerbezogen sind.

Hinsichtlich der Kompatibilität und der schnellen Umsetzungsmöglichkeit habe ich mich für die Umsetzung als Web-App entschieden. Aufgrund des gegebenen Zeitrahmens wird dieser Prototyp ohne Anmeldemöglichkeit umgesetzt. In nachfolgenden Projekten könnte man dieses Feature umsetzen.

4.3 Modularität

In diesem Abschnitt geht es um das Konzept der Modularität, warum es für diese Arbeit wichtig ist und was es bedeutet.

Durch den eingeschränkten Zeitrahmen ist es nur möglich eine begrenzte Anzahl an Features zu implementieren. Allerdings wäre es für nachfolgende Projekte möglich, auf meiner Arbeit aufzubauen. Dort könnten dann weiterführende Features hinzugefügt werden. Um das möglich zu machen, braucht es eine Implementierung, die einfach erweiterbar ist. Daher ist eine wichtige Anforderung Modularität. Das bedeutet, dass das System nicht aus einem großen Modul besteht, sondern aus mehreren unabhängigen kleinen. Durch einen modularen Aufbau können Module einfach verändert oder ausgetauscht werden, ohne die Funktion des Systems zu beeinträchtigen.

Konkret wird es beispielsweise bei der Unterstützung der Analyse weiterer NFRs. Innerhalb meiner Arbeit wird nur die Analyse von Erklärbarkeit umgesetzt. Aber das System wird so implementiert, dass es einfach um weitere NFRs erweitert werden kann.

4.4 NLP-Ansatz

In diesem Abschnitt geht es um den Analyse-Ansatz und wie dieser mithilfe von NLP-Techniken (siehe Abschnitt 2.3.3) umgesetzt werden soll.

Die Anforderung an mein System ist nicht nur eine gute initiale Analyse, sondern ein iterativer Lernprozess. Indem ein Nutzer die Analyse des Systems „korrigiert“, kann es dazulernen und im nächsten Schritt besser analysieren. Zunächst geht es darum, wie Erklärungsbedarf überhaupt erkannt werden kann.

4.4.1 Erklärungsbedarf identifizieren

Um einer Lösung näherzukommen, muss ich das Problem zunächst besser verstehen. Ich erstelle mir einen Datensatz mit 5360 Reviews (mehr dazu siehe Abschnitt 5.4.4) und fange an, manuell nach Erklärungsbedarf zu suchen. Ich bemerke, dass mir die Entscheidung, ob ein Textsegment (im Folgenden auch „Phrase" genannt) auf Erklärungsbedarf hindeutet, oft schwerfällt. Häufig ist es wichtig, sich in den Absender hineinzuversetzen, um zu verstehen, was er benötigt. In diesem Beispiel eines Reviews zur Navigationsapp „Lyft" wird deutlich, dass die Abgrenzung zwischen Erklärbarkeit und Usability oft unklar ist:

„Why doesn't the app list [the addresses] with an option of a map? Or easily allow one to choose that list? Could not quickly find this option - not user friendly."

Was benötigt der Absender hier? Er scheint eine Liste von Adressen nicht schnell gefunden zu haben, das ist sein Kritikpunkt. Demnach gibt es hier ein Problem mit der Benutzung der App, was definitiv unter Usability fällt. Würde ihm eine Erklärung (z.B. ein Tutorial) allerdings helfen, dass er die App besser benutzen kann? Man könnte annehmen, dass es so ist. In diesem Fall hat die Phrase „Could not quickly find this option" zum Schluss geführt, dass es sich um Erklärungsbedarf handelt.

Es wird deutlich: Wenn die Analyse schon für einen Menschen viele Annahmen und eine Kette von Schlüssen fordert, dann wird es auch für ein KI-System nicht einfach sein. Hinzu kommt die geringe Anzahl an Fällen von Erklärungsbedarf. Im Zuge einer Analyse von 300 Reviews konnte ich nur in 5 davon Erklärungsbedarf identifizieren (siehe Abschnitt 5.4.4).

4.4.2 Wahl des NLP-Ansatzes

Vor der manuellen Analyse bin ich davon überzeugt gewesen, das Problem mit einem Deep-Learning-Algorithmus lösen zu können. Ich hätte zwar mindestens 5000 Trainingsdaten 2.3.1 gebraucht, doch diese habe ich mir durch einen teilautomatisierten Ansatz erstellen zu können erhofft. Ich habe feststellen müssen, dass es aufgrund der Komplexität nötig wäre, alle 5000 Trainingsdaten manuell zu erstellen. Eine manuelle Analyse, bedingt durch die wortreichen Review-Texte, ist sehr zeitintensiv. So habe ich für 100 Reviews in etwa 1,5 Stunden gebraucht. Bei 5000 Reviews wären das 75 Stunden, also fast zwei komplette Arbeitswochen. Es wird mir deutlich, dass dieser Ansatz im Rahmen dieser Arbeit nicht umsetzbar ist.

Im Zuge meiner Analyse bin ich auf einen simpleren Machine-Learning-Ansatz namens „rule-based Matching" (mehr dazu siehe 5.4.2) gestoßen, den ich ursprünglich nur zum Erstellen der initialen Trainingsdaten habe verwenden wollen. Doch da dieser Ansatz trotz eines kleinen Datensatzes

an Trainingsdaten gute Ergebnisse hervorbringt, habe ich mich entschieden, gänzlich auf diesen Ansatz zu setzen.

Das rule-based Matching basiert darauf, vorgegebene Phrasen in Texten zu erkennen. Diese Phrasen können als Folge (ab hier „Pattern“ genannt) von Wort-Repräsentationen, beispielsweise als UPOS tags (siehe Abschnitt 2.3.3) oder in Lowercase-Form, angegeben werden. Man hat nun die Möglichkeit ein Pattern anzugeben und danach in Texten zu suchen. Beispielsweise kann die Phrase „Could not quickly find this option” mit folgendem Pattern erkannt werden:

„Could“	$\hat{=}$	UPOS: „VERB“
„not“	$\hat{=}$	Lowercase: „not“
„quickly“	$\hat{=}$	UPOS: „ADV“
„find“	$\hat{=}$	UPOS: „VERB“
„this“	$\hat{=}$	UPOS: „DET“
„option“	$\hat{=}$	UPOS: „NOUN“

Obwohl dieses Pattern die Phrase erkennt, ist ersichtlich, dass bei einer so allgemeinen Repräsentation auch viele ungewollte Phrasen erkannt werden. Der Recall ist somit hoch, die Precision aber niedrig (siehe zu den Metriken Abschnitt 2.3.2). Durch das Anwenden von genaueren Wort-Repräsentationen können detailliertere Patterns kreiert werden. Diese Patterns führen zu einer höheren Precision.

Mit einer Sammlung von gut angepassten Patterns können die Anforderungen erfüllt werden. Aufgrund dessen habe ich mich für den Ansatz des „rule-based Matching“ entschieden.

4.4.3 Iteratives Training

Wie kann nun ein System durch Nutzereingaben trainiert werden, welches durch eine fest vorgegebene Sammlung an Patterns funktioniert? Die Lösung besteht darin, solche Patterns automatisch zu erstellen. Im Auslieferungszustand der Software wird eine initiale Sammlung vorgegeben. Der Nutzer kann jedoch durch Kennzeichnung von Textsegmenten Patterns automatisch erstellen lassen, die zur Sammlung hinzugefügt und in die Analyse eingeschlossen werden. Das gilt sowohl für das Hinzufügen als auch das Ausschließen von Patterns. Somit kann ein iteratives Training umgesetzt werden.

Kapitel 5

Implementierung

In diesem Kapitel geht es um den Prozess der Implementierung. Ich zeige auf, warum ich welche Technologien verwendet und wie ich Probleme überwunden habe.

5.1 Architektur

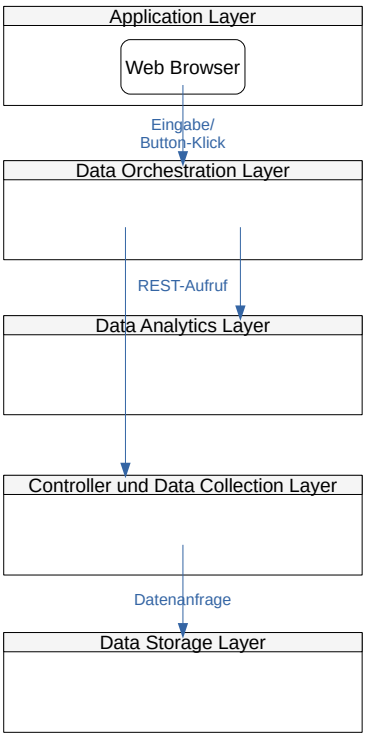

Abbildung 5.1: Schematische Architektur

Die getroffenen Konzeptentscheidungen 4 geben wiederholt Leitlinien vor. Schon bei der Festlegung der Architektur spielt ein bestimmtes Konzept eine große Rolle: Modularität (siehe Abschnitt 4.3). Wenn ich ein modulares System bauen will, dann sollten Teile auswechselbar sein. Bei einer monolithischen Architektur gibt es im Grunde nur ein großes Modul. In einer Microservice-Architektur hingegen kommunizieren viele unabhängige Layer (Microservices) miteinander [10], wodurch sie leicht austauschbar sind. Daher entscheide ich mich für letzteren Ansatz.

Abbildung 5.1 zeigt schematisch die Layer und wie sie kommunizieren. Dabei sind sie nach Abstraktion geordnet von oben nach unten. Ganz oben befindet sich der für den Nutzer sichtbare Application Layer. Ganz unten steht der Data Storage Layer, der für die Datenbank zuständig ist. Der Data Orchestration Layer bietet die Kommunikation zu den abstrakteren Layern darunter. Diese findet über eine REST-Schnittstelle [10] statt. Ich verwende REST, da es ein simples, weit verbreitetes und vielerorts unterstütztes Paradigma ist.

Im Folgenden wird die Implementierung der einzelnen Layer im Detail erläutert, in der Reihenfolge der Implementierung.

5.2 Controller, Data Collection und Data Storage Layer

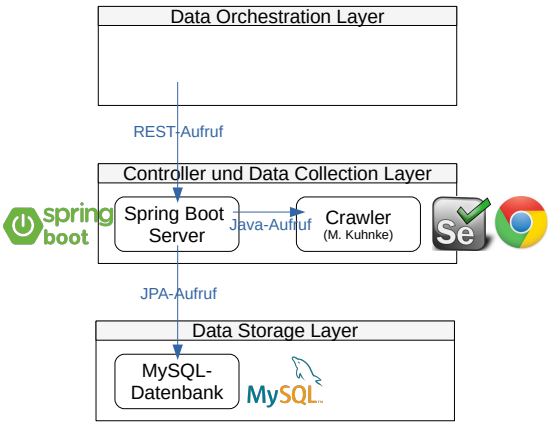

Abbildung 5.2: Controller, Data Collection und Data Storage Layer und Kommunikation mit anderen Layern

In diesem Abschnitt geht es um die Implementierung des Controllers und seinen Anschluss an den Data Storage und den Data Collection Layer.

Zunächst folgen die Überlegungen zur Implementierung des Control-

lers. Für einen schnellen und robusten Datenbankzugriff bietet sich ein Framework in Java an. Da Java eine kompilierte Sprache ist, bietet sie gegenüber interpretierten Sprachen wie Python einen Performancevorteil. Dies wirkt sich besonders beim Verarbeiten großer Datenmengen aus. Da das zu implementierenden System große Mengen an Daten verarbeiten soll, entscheide ich mich für ein Framework in Java. Spring Boot[1] bietet eine einfache Möglichkeit, in wenigen Schritten einen Server mit REST-API im populären Spring Framework aufzusetzen. Somit können REST-Anfragen an den Server (Controller) gestellt werden.

Zur Anbindung an eine MySQL-Datenbank (Data Storage Layer) fungieren Aufrufe per Spring Data JPA (Java Persistence API)[2]. Somit werden SQL-Anfragen abstrahiert und vereinfacht.

Nun geht es um den Data Collection Layer. Er kümmert sich um das Einlesen von Review-Informationen. Ich darf dabei auf die Arbeit von Kuhnke [8] aufbauen. Seine Lösung ist ein Crawler, der auf Web-Scraping (mehr dazu siehe [8, S. 7-8]) mithilfe der Selenium-Bibliothek[3] basiert. Allerdings stoße ich beim Einbinden seines Codes auf Probleme. Ich finde heraus, dass sein System eine eigene Datenbank hat, die mit meiner kollidiert. Ich muss also seinen Code anpassen, indem ich seine Datenbank deaktiviere. Um die Modularität zu wahren, binde ich den Crawler über ein Java-Interface ein. Somit kann auch ein anderer Crawler eingesetzt werden. Er muss nur das Interface implementieren.

5.3 Data Orchestration Layer

In diesem Abschnitt geht es um die Implementierung des Data Orchestration Layers, der für die Anzeige im Web-Browser zuständig ist. Dieser Layer bildet mein Frontend und die Kommunikation mit dem Backend.

Da ich nach der Entscheidung in Abschnitt 4.2 eine Web-App baue, suche ich nach einem JavaScript-Framework und stoße auf React[4]. Gegenüber Alternativen (z.B. Vue oder Angular) bietet React eine sehr leichtgewichtige Lösung, die sich rein auf das Frontend konzentriert. Mithilfe des Frameworks ist es möglich, komplett auf Seitenaktualisierung zu verzichten. Außerdem werden geänderte Daten erkannt und automatisch im Frontend aktualisiert.

Da React von sich aus keine Unterstützung für REST bietet, verwende ich die Axios-Bibliothek[5]. Sie kann http-Aufrufe von React in REST-Aufrufe umwandeln.

Bei der Kommunikation zwischen Data Orchestration Layer und Controller stoße ich auf ein Problem: Da sie beide auf unterschiedlichen Ports

[1] `https://spring.io/projects/spring-boot`
[2] `https://spring.io/projects/spring-data-jpa`
[3] `https://www.selenium.dev/`
[4] `https://reactjs.org/`
[5] `https://github.com/axios/axios`

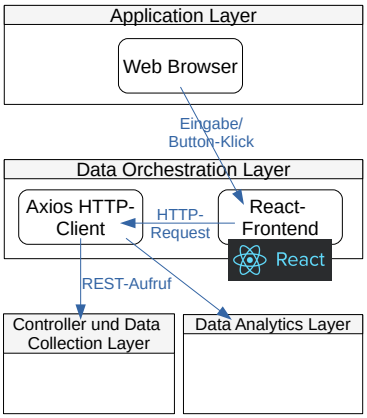

Abbildung 5.3: Data Orchestration Layer und Kommunikation mit anderen Layern

operieren, können sie nicht miteinander kommunizieren. Doch durch die Aktivierung von Cross-Origin Resource Sharing (kurz CORS)[6] können Ports freigeschaltet und das Problem gelöst werden.

Beim Design richte ich mich nach den Vorgaben des modernen und simplen Design-Systems Material Design[7] von Google. Dieses Design-System verhilft zu einer GUI mit hoher Usability, indem Elemente für Nutzer schnell ersichtlich und einfach verständlich sind. Durch den Einsatz einer Design-Bibliothek namens Material Design Bootstrap[8], die sich nach dem Material Design richtet, habe ich Zugriff auf vorimplementierte Elemente. Durch das Verbauen dieser Elemente kann schnell eine robuste GUI entworfen werden.

5.3.1 Hervorhebung und Bearbeitung von Erklärungsbedarf

Wie in Abschnitt 4.1.3 beschrieben, ziele ich darauf, Erklärungsbedarf durch Markierung im Text hervorzuheben. In diesem Abschnitt geht es um die Implementierung dieses Features.

Um eine Markierung und Hervorhebung von Textstellen zu ermöglichen, überlege ich zunächst, in welchem Format der zu markierende Text vorliegt. Da der Text von einem NLP-Tool analysiert wird, liegt er als Kette von Tokens vor (siehe Abschnitt 2.3.3). Um Hervorhebungen zu speichern, müssen diese Tokens gespeichert und identifiziert werden können. Diese Funktionalität setze ich zunächst im Controller um.

Danach suche ich nach Software-Bibliotheken, die zur Markierung von Token-basiertem Text verhelfen. Ich suche allerdings nicht nur nach der Möglichkeit, Tokens einmalig zu markieren. Da ein Token Teil mehrerer

[6] `https://fetch.spec.whatwg.org/#http-cors-protocol`
[7] `https://material.io/`
[8] `https://mdbootstrap.com/docs/react/`

Phrasen sein kann, die auf Erklärungsbedarf hindeuten, muss es möglich sein, einen Token mehrmals zu markieren. Außerdem muss es möglich sein, diese multiplen Markierungen einzeln zu löschen.

Leider lässt sich keine Bibliothek finden, die sich in mein System integrieren lässt und die gewünschten Funktionen bietet. Ich entscheide mich eine Bibliothek namens „react-text-annotate“[9] einzusetzen. Mithilfe dieser Bibliothek ist es möglich, Text als Tokens interaktiv zu markieren. Es wird außerdem eine kurze Benennung der Markierung daneben aufgeführt. Im vorliegenden Fall kann dies „explanation_need“ (englische Übersetzung für Erklärungsbedarf) sein. Es besteht die Möglichkeit, verschiedene Benennungen zu verwenden. So können in Zukunft weitere NFRs angezeigt und ihre Markierungen unterschieden werden, gemäß des Konzeptes der Modularität. Zusätzlich sind multiple Markierungen zu einem bestimmten Grad möglich. Liegen multiple Markierungen vor, so werden diese hintereinander angezeigt (siehe Abb 5.4). Allerdings können Teile einer Markierung nicht markiert werden und somit können nicht sicher alle Fälle markiert werden, bei denen Erklärungsbedarf vorliegen könnte. Außerdem ist die Funktion Markierungen durch linken Mausklick zu entfernen, für dieses System unpraktisch, da Nutzer versehentlich einen Mausklick tätigen und wichtige Information unwiderruflich löschen könnten.

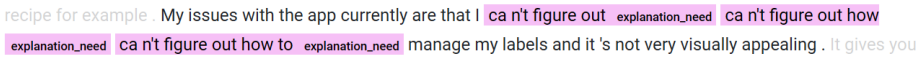

Abbildung 5.4: Hervorhebung mehrerer Fälle von Erklärungsbedarf auf einmal in Token-basiertem Text mithilfe von „react-text-annotate“

Die Bibliothek wird nun in den vorhandenen Code integriert und modifiziert. Nun sind Markierungen jeglicher Art möglich. Um das Löschen von Markierungen zu verbessern, wird ein Kontextmenü implementiert, das bei rechtem Mausklick auf eine Markierung erscheint. Durch Klick auf das Löschen-Symbol wird die Markierung gelöscht (siehe Abb 5.5).

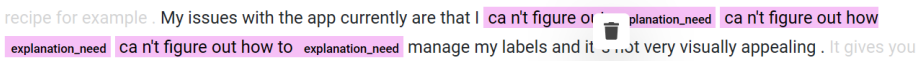

Abbildung 5.5: Löschen von Markierungen mithilfe eines Kontextmenüs

5.4 Data Analytics Layer

In diesem Abschnitt geht es um die Implementierung des Data Analytics Layers, der für die automatische Review-Analyse zuständig ist.

[9] `https://github.com/mcamac/react-text-annotate`

Mein Ziel ist es, ein System zu entwickeln, das Erklärungsbedarf in gegebenem Text erkennt. Das Ziel soll mithilfe einer Bibliothek mit Machine-Learning-Tools erreicht werden. Diese Bibliothek soll Tools zur Verarbeitung von Sprache mithilfe von NLP 4.4 enthalten. Durch Nachforschung stoße ich auf drei weit verbreitete Bibliotheken: Stanford Core NLP (in Java), NLTK und spaCy (beide in der Programmiersprache Python). Da spaCy[10] am einfachsten lernbar und am besten dokumentiert zu sein scheint, entscheide ich mich dafür.

5.4.1 Ansatz

In diesem Abschnitt geht es um den zu wählenden Ansatz zur Implementierung des KI-Systems.

Da momentan noch nicht mit Gewissheit gesagt werden kann, ob ein bestimmter KI-Algorithmus zu einem Ergebnis führt, womit eine gute Systemleistung erreicht werden kann, lege ich mich nicht auf einen Algorithmus fest. Stattdessen überlege ich mir einen Explorationsansatz, in dem immer wieder durch gewonnene Erkenntnisse Entscheidungen revidiert werden können. Durch die Flexibilität ist ein Erfolg wahrscheinlich. Der entwickelte Ansatz lautet folgendermaßen: Zunächst wird ein Datensatz zusammengestellt und klar definiert, welche konkreten Aufgaben das KI-System bewältigen muss. Danach geht man in einen Prozess des Labeling 2.3.3. Dort erfährt man viel über seine Daten und auch über Randfälle, die man bisher nicht bedacht hat, die aber wichtig für die Implementierung sind. Nun wird ein Machine-Learning-Algorithmus genutzt, um möglichst viele gelabelte Daten zu erzeugen. Mithilfe der bereits manuell gelabelten Daten kann die Performance des Algorithmus verifiziert werden. Durch die per Machine Learning mit Label versehenen Daten kann ein Deep-Learning-Algorithmus trainiert werden. Dieser Algorithmus ist nun in der Lage, das Problem zu einer gewissen Güte automatisch zu lösen.

5.4.2 Rule-based Matcher

Ich forsche nach, welchen Machine-Learning-Algorithmus ich verwenden kann, um gelabelte Daten zu erstellen. Dabei stoße ich auf den von spaCy angebotenen Ansatz des „rule-based Matching“. Ein sogenannter Matcher klassifiziert Text durch gegebene Patterns. Die Patterns spezifizieren Phrasen im Text, die mehrere Tokens 2.3.3 umfassen können, jedoch direkt verbunden sein müssen (keine Ausdrücke dazwischen erlaubt). Entdeckt nun ein Matcher, dass eine Stelle im Text ein Pattern erfüllt, so merkt er sich, wo im Text dieser Fall (im Folgenden „Match“ genannt) aufgetreten ist. Durch Tools lassen sich diese Matches visualisieren, beispielsweise durch Hervorhebung im

[10] `https://spacy.io/`

Text.

spaCy hat mithilfe eines großen Datensatzes an Sprachinformationen die Möglichkeit, Text mit linguistischen Angaben zu kennzeichnen. Darunter zählen Kennzeichnung von Satzteilen (z.B. UPOS tags 2.3.3, bei spaCy „POS“ genannt), Grundform-Repräsentation (bei spaCy „LEMMA“ genannt), Satzabhängigkeiten (z.B. Universal Dependency Relations 2.3.3, bei spaCy kurz „DEP“) und Lowercase-Repräsentation (bei spaCy kurz „LOWER“). In Abbildung 5.6 ist ein Beispiel gegeben für den Satz „Apple is looking at buying U.K. startup for $1 billion“. Eine ausgiebige Dokumentation aller linguistischen Kennzeichnungen bei spaCy findet sich auf der Internetseite[11].

Text	LEMMA	POS	DEP	LOWER
Apple	apple	PROPN	nsubj	apple
is	be	AUX	aux	is
looking	look	VERB	ROOT	looking
at	at	ADP	prep	at
buying	buy	VERB	pcomp	buying
U.K.	u.k.	PROPN	compound	u.k.
startup	startup	NOUN	dobj	startup
for	for	ADP	prep	for
$	$	SYM	quantmod	$
1	1	NUM	compound	1
billion	billion	NUM	pobj	billion

Abbildung 5.6: Linguistische Kennzeichnung bei spaCy (Beispiel angelehnt an Dokumentation von spaCy[13])

Der Matcher von spaCy gebraucht Patterns, um Matches zu finden. Diese Patterns bestehen aus einer Reihe von linguistischen Kennzeichnungen und speziellen Operatoren. Mithilfe der Operatoren ist es möglich, bestimmte Teile des Patterns explizit auszuschließen oder optional zu machen. Ein Beispiel ist folgendes Pattern:

```
[ {'LEMMA': 'what'}, {'LEMMA': 'be'}, {'POS': 'DET'},
{'LEMMA': 'about'}, {'POS': 'PUNCT', 'OP': '?'} ]
```

Es bezieht sich auf folgenden Satz: „What is that about?“. Durch die Verwendung des Operators (OP) „?“ ist ein Satzzeichen am Ende optional. Das bedeutet, dass auch die Sätze „What is that about“ oder „What is that about!“ erkannt werden. Eine ausgiebige Dokumentation der Operatoren findet sich unter[14].

[10] https://spacy.io/usage/linguistic-features
[11] https://spacy.io/api/annotation
[14] https://spacy.io/usage/rule-based-matching

5.4.3 Plan zum Erstellen eines KI-Systems

Nun möchte ich den gewählten Ansatz (siehe Abschnitt 5.4.1) Schritt für Schritt durchführen. Bevor ich ein KI-System aufsetzen kann, muss ich zunächst meine Daten besser verstehen. Ich muss mir ein Bild davon machen, wie Reviews aussehen, in denen Erklärungsbedarf zu finden ist. Außerdem brauche ich einen KI-Algorithmus, der durch steigende Anzahl an Trainingsdaten immer bessere Ergebnisse liefert. Um diese Ziele zu erreichen, teile ich den gewählten Ansatz in 7 Schritte auf.

1. Datensatz erstellen

Aus bereits funktionierender Anbindung an den Data Collection Layer erstelle ich einen großen Datensatz und speichere ihn als Datei im CSV-Format. Das Format besteht lediglich aus einer Spalte für den Review-Text. Durch das CSV-Format ist später ein einfacher Labeling-Prozess möglich, da das Labeling lediglich durch eine hinzugefügte Spalte dargestellt wird, die entweder einen Wert 1 (positive Ausgabe, Erklärungsbedarf vorhanden) oder 0 (negative Ausgabe, kein Erklärungsbedarf vorhanden) enthält.

2. Erste Patterns erstellen und Ergebnisse auswerten

Um das Problem näher kennen zu lernen und insbesondere seine Tücken, erstelle ich eine Reihe von Patterns und optimiere diese, damit sie zumindest zu einer bestimmten Zufriedenheit Erklärungsbedarf erkennen können.

Um einen Trend erkennen und nachregeln zu können, werte ich meine Ergebnisse aus. Ich verwende den Datensatz, um zu sehen, wie weit mich mein Ansatz gebracht hat und ob ich ihn wechseln sollte.

3. Patterns verbessern und einen Datensatz labeln

Anhand der Auswertung kann ich nun meine Patterns verbessern. Ich kann außerdem messen, wie viele Texte mit Erklärungsbedarf bisher gefunden werden. Anhand dessen kann ich entscheiden, wie ich den nächsten Schritt angehe: den Labeling-Prozess.

Ich nehme nun einen Teil des Datensatzes und labele diesen, indem ich eine Spalte “label“ einführe, in die ich entweder eine 0 oder eine 1 eintrage.

4. Matching und Labeling vergleichen

Anhand desselben Datensatzes vergleiche ich nun die Ergebnisse von Matching und Labeling. Besonders interessant sind hierbei die Fälle, wo sie nicht übereinstimmen. Danach führe ich gegebenenfalls einige Verbesserungen an den Patterns und den Labels durch.

5. Lernsystem implementieren

Durch den gut gelabelten Datensatz kann ich nun einen Deep-Learning-Algorithmus mit Trainingsdaten speisen. Um den Datensatz dafür verwenden zu können, muss ich ihn auf eine bestimmte Form bringen. Danach kann ich den Algorithmus trainieren.

6. Ergebnisse auswerten

Ich werte die Ergebnisse aus, indem ein Teil des Datensatzes sowohl vom Matching- als auch vom Deep-Learning-Ansatz verarbeitet wird. Wiederum sind hier die Unterschiede sehr interessant. Durch die Erkenntnisse kann ich meinen Ansatz überdenken und gegebenenfalls verbessern.

7. Modell speichern und iteratives Training

Im letzten Schritt finde ich einen Weg, den Fortschritt zu speichern und den Algorithmus durch Nutzereingaben trainierbar zu machen.

5.4.4 Durchführung des Plans

In diesem Abschnitt wird die Implementierung des KI-Systems beschrieben. Hierzu wird nach dem im letzten Abschnitt vorgestellten Plan vorgegangen.

1. Datensatz erstellen

Ich finde heraus, wie ich meine Daten aus Java ins CSV-Format konvertiere. Ich entschließe mich dazu, nicht nur eine Spalte für Review-Text, sondern auch eine für Review-Titel anzulegen. Diese Spalte gibt es zwar nur bei App Store Apps, aber sie enthält ebenso wie der Text wichtige Informationen, die auf Erklärungsbedarf hindeuten können. Außerdem füge ich zu Dokumentationszwecken die Spalten „Associated App“ und „App ID“ hinzu.

Nun erstelle ich meinen Datensatz. Da ich vermute, dass in der App-Kategorie „Navigation“ viele Vorkommnisse von Erklärungsbedarf sind, wähle ich Reviews von Apps dieser Kategorie aus. Mein Datensatz besteht aus 3200 Reviews aus dem Google Play Store von diversen Apps und aus dem Apple App Store von folgenden Apps: „Lyft“ mit 570 Reviews, „Google Maps“ mit 550 Reviews, „Waze Navigation & Live Traffic“ mit 570 Reviews und „ParkMobile - Find Parking“ mit 470 Reviews. Insgesamt besteht der Datensatz also aus 5360 Reviews.
Zunächst werden manche Zeichen falsch kodiert. So werden Sonderzeichen wie Emoticons im Datensatz als Fragezeichen dargestellt. Um das zu beheben, ändere ich das Encoding auf UTF-8.

2. Erste Patterns erstellen und Ergebnisse auswerten

Ich lese meine Daten mit Python ein und erstelle mithilfe der spaCy-Library zwei erste Patterns für die Ausdrücke „How can I“ und „I can't use“. Unter allen 5360 Reviews finde ich für diese ersten beiden Patterns bereits 38 Matches. Unter diesen 38 befinden sich nach Überprüfung aber nur 11 Fälle, in denen wirklich Erklärungsbedarf besteht. Bisher kann ich noch keine Aussage zum Recall treffen, da ich nicht weiß, wie viele Reviews mit Erklärungsbedarf es insgesamt gibt. Meine Precision allerdings ist noch nicht sehr hoch, da weniger als 30% meiner Matches überhaupt Erklärungsbedarf enthalten. Ich entscheide mich dazu, vorerst höheren Wert auf Precision zu legen.

Ich fahre fort mit anderen Ausdrücken wie „how does it“, „can you explain“, „hard to understand“ und „i don't understand“. Ich bemerke, dass nur beim Ausdruck „hard to understand“ eine annehmbare Precision vorherrscht. Außerdem gibt es unter den Matches in Review-Titeln eine deutlich höhere Precision als bei denen in Review-Texten.

3. Pattern verbessern und einen Datensatz labeln

Ich merke, dass ich so nicht weiterkomme. Ich beschließe mich zunächst von den Patterns abzuwenden und mit dem Labeling-Prozess zu beginnen. Daraus erhoffe ich mir konkretere Beispiele für Patterns.

Ich beginne Reviews zu labeln. Mir fällt auf, dass ein Großteil der Reviews sich nicht um die App selbst drehen, sondern um den Service des Unternehmens, welches hinter der App steht. Im vorliegenden Fall geht es um die App „Lyft“. Viele Reviews scheinen von Kunden zu stammen, die unzufrieden mit dem Service oder sogar frustriert darüber sind. Oft werden darin die negativen Erlebnisse der Kunden mit dem Service ausführlich geschildert. Dieses Feedback ist allerdings für die Entwicklung der App unerheblich und auch für Erklärbarkeit.

Ich bemerke außerdem, dass die meisten Kunden keine Fragen an die Entwickler stellen wie: „Wieso verhält sich die App so?“. Stattdessen behaupten sie eher, es liege ein Fehler in der App vor. In manchen Fällen schlagen sie auch eine Lösung vor, wie die Entwickler diesen Fehler beheben sollten. Impliziter Erklärungsbedarf (siehe Abschnitt 2.2.1) scheint also häufiger zu sein als expliziter.

Nach 1,5 Stunden Labeling habe ich 100 Reviews gelesen. Zwei Reviews davon enthalten nach meiner Einschätzung Erklärungsbedarf. Daraus schließe ich, dass der Labeling-Prozess aufgrund der hohen Wortanzahl pro Review viel Zeit in Anspruch nimmt. Obwohl ich nur zwei passende Reviews gefunden habe, sehe ich das Ergebnis positiv, denn ich habe endlich echte Anhaltspunkte, wie ich meine Patterns gestalten kann. Ich beschließe den Vorgang fortzuführen, allerdings für eine andere App, bei der es hoffentlich

weniger Feedback gibt, welches nichts mit der App zu tun hat.

Ich labele 100 Reviews der App „Waze“ (3 Fälle mit Erklärungsbedarf) und 100 der App „Google Maps“ (0 Fälle mit Erklärungsbedarf). Damit kann ich im Schnitt auf einen Anteil von Reviews mit Erklärungsbedarf von ca. 1,7% schließen. Aus den positiven Fällen erstelle ich nun einige Patterns.

5. Matching und Labeling vergleichen

Da ich nun meinen Ansatz gewechselt und aus den gelabelten Daten Patterns erstellt habe, die exakt auf die gefundenen Fälle von Erklärungsbedarf zielen, ergeben Matching und Labeling die gleichen Ergebnisse. Daher betragen Precision und Recall jeweils den Maximalwert 1. Das bedeutet, dass auch mein aktueller F1-Score 1 beträgt, also den Maximalwert. Dieser bezieht sich allerdings zunächst nur auf die gelabelten Daten. Bei allen anderen Daten kann ich noch keine Abschätzungen vornehmen.

6. Lernsystem implementieren und 7. Ergebnisse auswerten

Ursprünglich habe ich auf einen Deep-Learning-Ansatz gesetzt, um mein Problem gemäß der Anforderungen lösen zu können. Durch die Erkenntnisse aus den letzten Schritten komme ich allerdings zur Schlussfolgerung, dass Deep Learning mich nicht weit bringen wird. Durch den niedrigen Gesamtanteil an Reviews mit Erklärungsbedarf von ca. 1,7% wäre eine sehr große Menge an gelabelten Daten nötig, um angemessene Ergebnisse zu erzielen. Nun sehe ich allerdings, dass der Ansatz des Matching für einen sehr guten F1-Score sorgen kann. Es braucht lediglich manuelle Überprüfung. Ich lasse also vom Deep-Learning-Ansatz ab und konzentriere mich darauf, stattdessen den Matcher-Ansatz zu optimieren. Ich erhoffe mir, dass durch eine größere Anzahl an Patterns auch viele Daten korrekt erfasst werden können, die nicht manuell überprüft worden sind.

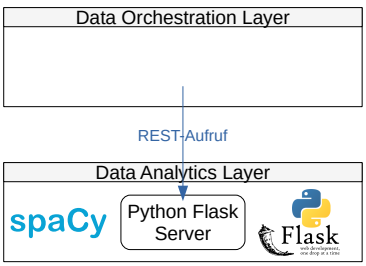

Abbildung 5.7: Data Analytics Layer und Kommunikation mit anderen Layern

Um mein KI-System als Data Analytics Layer an den Data Orchestration Layer anschließen zu können, brauche ich eine REST-API für Python. Ich

stoße schnell auf ein leichtgewichtiges Framework namens Flask[15], welches eine REST-API anbietet. Durch Aktivierung von CORS ist eine Kommunikation mit dem Data Orchestration Layer schnell implementiert (siehe Abb. 5.7). Nun ist es möglich, Review-Daten automatisch zu analysieren.

8. Modell speichern und iteratives Training

Wie kann ich nun mein System durch Nutzereingaben dazulernen lassen? Da der Lernfortschritt meines Systems stark von der Anzahl und Güte der Patterns abhängt, kann ich dies durch automatisches Generieren von Patterns erreichen. Die Idee ist, dass ein Nutzer selbst Patterns erstellen lassen kann, die danach in die Analyse einfließen.

Um dieses Ziel zu erreichen, unterscheide ich zunächst zwischen benutzerdefinierten Patterns und solchen, die vom Entwickler selbst erstellt wurden. Ich schreibe Python-Code, der gegebene Tokens 2.3.3 automatisch in Patterns umsetzen kann. Die benutzerdefinierten Patterns werden als Datei gespeichert, damit sie im Nachhinein von Administratoren gewartet werden können. Durch die Art und Weise, wie der Matcher von spaCy funktioniert, können nicht alle benutzerdefinierten Patterns gleich behandelt werden. Es wird unterschieden zwischen zwei Fällen:

1. Der Nutzer liest eine Textstelle, in der er Erklärungsbedarf sieht. Dieser Erklärungsbedarf wurde jedoch nicht automatisch erkannt. Er möchte also diese Textstelle als neues benutzerdefiniertes Pattern hinzufügen.

2. Der Nutzer sieht eine Textstelle, die fälschlicherweise als Erklärungsbedarf erkannt wurde. Er möchte, dass eine solche Stelle in Zukunft nicht mehr falsch erkannt wird. Er erzeugt also ein benutzerdefiniertes Pattern und schließt dieses von der Analyse aus.

Der Matcher besitzt nur die Möglichkeit, Patterns hinzuzufügen, nicht auszuschließen. Daher implementiere ich einen eigenen Mechanismus für Fälle wie Nummer 2. Dieser Mechanismus funktioniert wie folgt: Es werden zwei voneinander unabhängige Matcher-Objekte erzeugt, einer für das Hinzufügen (engl. include) und einer für das Ausschließen (engl. exlude) von Patterns. Der Text wird zunächst vom include-Matcher analysiert und besitzt nun Analyse-Daten. Das gleiche passiert danach für den exclude-Matcher mit seperaten Analyse-Daten. Die beiden Analyse-Daten werden nun verglichen. Wo der exclude-Matcher *und* der include-Matcher einen Match gefunden haben, wird dieser entfernt.

Es können nun beide Fälle gehandhabt werden. Das KI-System kann demnach durch Nutzereingaben dazulernen.

[15] `https://flask.palletsprojects.com/en/1.1.x/`

Kapitel 6

Auswertung

In diesem Kapitel werden die Ergebnisse der Arbeit diskutiert. In Abschnitt 6.1 geht es um die Erkenntnisse, die ich über Analyse von Erklärungsbedarf in Reviews gewonnen habe. Abschnitt 6.2 diskutiert die Herangehensweise zur Wahl eines KI-Ansatzes. Im letzten Abschnitt 6.3 geht es darum, wie leistungsfähig das entwickelte System ist.

6.1 Review-Analyse - Erklärungsbedarf erfordert fallspezifische Betrachtung

Im Prozess des Labelings und des anschließenden Erstellens von Patterns habe ich einige Erkenntnisse gewinnen können.

So konnte ich sehen, dass eine Phrase, die in einem Fall auf Erklärungsbedarf hindeutet, in einem anderen Fall nicht darauf hindeuten muss. Im Folgenden geht es um die Phrase „don't know how to“. Das zugehörige Pattern ist so konfiguriert, dass es alle grammatikalischen Beugungen von „do“ erkennen kann. Daher wird sowohl „don't“ als auch „doesn't“ erkannt.

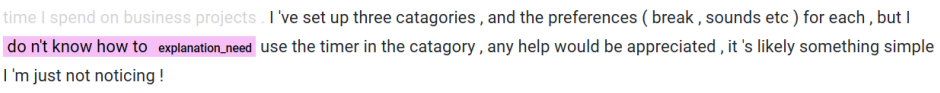

Abbildung 6.1: Beispiel eines Reviews zur App „Brain Focus Productivity Timer“ aus dem Google Play Store. Hier wird Erklärungsbedarf richtig erkannt.

Im ersten Fall (siehe Abb. 6.1) hat der Verfasser klaren Erklärungsbedarf. Er scheint etwas in der App nicht zu verstehen und fragt daher nach Hilfe. Er benutzt dort die Phrase „don't know how to“, durch die der Matcher hier Erklärungsbedarf erkennt. Im zweiten Fall (siehe Abb. 6.2) verwendet der Verfasser ebenfalls diese Phrase, wenn auch in einer anderen Beugung. Allerdings scheint es ihm um etwas gänzlich anderes zu gehen. Er scheint

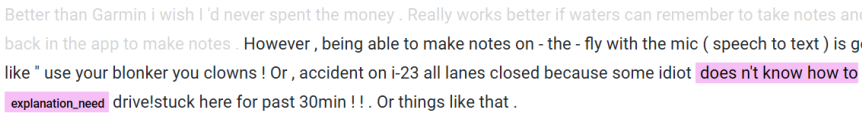

Abbildung 6.2: Beispiel eines Reviews zur App „Waze“ aus dem Google Play Store. Hier wird Erklärungsbedarf falsch erkannt.

unzufrieden mit dem Verhalten anderer im Straßenverkehr zu sein, was aber nichts mit der App zu tun hat.

Um diese Schlüsse ziehen zu können, braucht es fallspezifische Betrachtung des Kontextes. Es wird also deutlich, dass das Erkennen gewisser Phrasen nicht immer zum gewünschten Ziel führt. Ich ziehe daraus den Schluss, dass die automatische Analyse durch einen Matcher-Ansatz im Nachgang immer manuell überprüft werden muss, um verlässliche Ergebnisse zu erzielen. Meine Prognose ist, dass sich dieses Problem auch nicht durch weiteres Training gänzlich lösen lässt. Daraus lässt sich schlussfolgern, dass der Einsatz dieses Systems nur ein Hilfsmittel für Requirements Engineers sein kann, damit sie nicht alle Reviews händisch durchgehen müssen, sondern nur einige wenige. Es kann ihnen dagegen nicht die Arbeit der manuellen Überprüfung abnehmen.

6.2 KI-Ansatz - Deep Learning führt nicht immer zum Ziel

Anfangs bin ich davon überzeugt gewesen, dass Deep Learning mein Problem lösen kann. Durch die Erkenntnis, dass ich für eine solide Implementierung von Deep Learning zu wenige Trainingsdaten besitze, habe ich meinen Ansatz geändert. Stattdessen findet der Matcher-Ansatz von spaCy in dieser Arbeit Verwendung. Dieser baut auf einem durch Deep Learning vortrainierten Sprachmodell auf, funktioniert aber im Übrigen ohne den Einsatz von Deep Learning. Durch diesen Ansatz ist es mir möglich, trotz weniger Trainingsdaten eine annehmbare Leistung zu erzielen.

Goodfellow et al. bemerken, dass „der erfolgreiche Einsatz kleinerer Datensätze [...] ein wichtiges Forschungsgebiet [ist]“ [4, S. 24]. Auch mein vorliegendes Problem bedingt den Einsatz kleinerer Datensätze. Der Matcher-Ansatz scheint eine gute Lösung für Probleme dieser Art zu sein. Möglicherweise können die Erkenntnisse meiner Arbeit zu diesem Forschungsgebiet beitragen.

6.3 Systemleistung

Im Folgenden lege ich die Leistung des implementierten Systems dar und erkläre, wie sie zukünftig gesteigert werden kann.

Die Leistung der Feedback-Analyse ist stark abhängig von der Anzahl und Güte der konfigurierten Patterns. Im Zuge der Implementierung dieses Systems habe ich 300 Reviews manuell gelabelt. Davon habe ich in 5 Reviews Erklärungsbedarf gesehen. Aus den Erkenntnissen dieser 5 Reviews mit Erklärungsbedarf habe ich 9 Patterns entwickelt. Da Erklärungsbedarf aber sehr unterschiedlich aussehen kann, wird durch 9 Patterns kaum eine hohe Leistung erzielt werden können. Meine Annahme ist allerdings, dass durch ein wenig mehr Labeling bereits eine gute Leistungssteigerung erlangt werden kann.

6.3.1 Ergebnisse einer Labeling-Iteration

Um diese Annahme zu beweisen, führe ich eine weitere Labeling-Iteration durch, erstelle aus den Erkenntnissen Patterns und evaluiere im Nachhinein die Leistungssteigerung durch diese Iteration.

Labeling

Ich entscheide mich, 120 Reviews einer weiteren Navigations-App namens „HERE WeGo“ aus dem Apple App Store zu labeln. Ich finde 6 Reviews mit Erklärungsbedarf. Ohne bereits neue Patterns zu erstellen, lasse ich die 120 Reviews automatisch analysieren. Es werden 0 Matches gefunden. Dieses Ergebnis mag zunächst auf eine schlechte Leistung hindeuten, doch wie man später sehen kann, werden bei mehr Reviews auch mehr Matches erkannt. 120 Reviews sind bei dieser Anzahl an Patterns eine zu geringe Menge, um davon auf Leistung schließen zu können.

Patterns erstellen

Durch die Erkenntnisse aus den 6 Reviews mit Erklärungsbedarf kann ich neue Patterns erstellen. Allerdings kreiere ich nicht nur neue, sondern verfeinere auch bereits erstellte. Bisher habe ich immer auf eine hohe Precision gezielt, damit möglichst wenig falsch erkannte Matches entstehen. Um das zu erreichen, habe ich die Patterns sehr spezifisch gestaltet. Nun kann ich jedoch sehen, dass ich durch diesen Ansatz viele vermeintliche Matches knapp verpasse. So kann ich die Phrasen „couldn’t find any info“ oder „couldn’t figure out how to” nicht erkennen, wohl aber „couldn’t find how“, „couldn’t find how to“ oder „couldn’t find a way to”. So kann ich „don’t even know what it’s trying” nicht erkennen, wohl aber „don’t even know where to“.

Ich schließe daraus, dass es sich lohnen kann, die Patterns etwas genereller und nicht - wie bisher - sehr speziell zu konfigurieren. Dadurch kann es zwar sein, dass die Precision sinkt und die Ergebnisse weniger verlässlich sind, aber wie in Abschnitt 6.1 beschrieben, ist eine manuelle Überprüfung ohnehin nötig. Ich nehme an, dass es den Requirements Engineers mehr hilft, eine große Menge an Reviews automatisch aussortiert zu bekommen - selbst wenn einige wenige davon falsch erkannte sind - als nur eine geringe Menge. Es wird also darauf gezielt, einen hohen Recall zu erreichen, während ein geringes Sinken der Precision in Kauf genommen wird. Ein hohes Sinken der Precision muss allerdings vermieden werden, da sonst zu viele nicht relevante Reviews erkannt werden und das Überprüfen dieser Menge wiederum für Requirements Engineers zu zeitintensiv wäre.

Aufbauend auf diesen Erkenntnissen erstelle ich Patterns und habe nun eine Gesamtzahl von 23.

Evaluation der Iteration

Um die Ergebnisse zu evaluieren, muss zunächst eine Unterscheidung getroffen werden. Es gibt Reviews, die manuell gelabelt worden sind, und solche, die es nicht sind. Für erstere kennt man die Gesamtanzahl an Reviews, die tatsächlich Erklärungsbedarf enthalten (im Folgenden auch „relevante Fälle“ genannt), während man für letztere nur Vermutungen anstellen kann. Da die Metriken Recall und F1-Score darauf aufbauen, dass ein Wissen um die Gesamtanzahl an relevanten Fällen besteht (siehe Gleichung 6.1), können im letzteren Fall diese Metriken nichts aussagen. Von diesem Problem berichten auch Sajnani et al. [13]: Das Messen des Recalls stelle sich in vielen Fällen als schwierig heraus, da oft kein Wissen um die Gesamtanzahl der relevanten Elemente im zu messenden Datensatz besteht.

$$\begin{aligned} Recall &= \frac{\#_{relevante\ entnommene\ Elemente}}{\#_{relevante\ Elemente}} \\ &\overset{\text{Hier}}{=} \frac{\#_{richtig\ erkannte\ Reviews}}{\#_{alle\ Reviews\ mit\ Erklärungsbedarf}} \end{aligned} \tag{6.1}$$

Durch die genannten Schwierigkeiten lässt sich nur eine relative Aussage zum Recall treffen, das heißt, ob der Recall gestiegen oder gefallen ist.

Die Metrik Precision allerdings setzt kein Wissen über die Gesamtzahl der relevanten Fälle voraus, sondern nur über die Anzahl relevanter Fälle in der Menge der erkannten Reviews (siehe Gleichung 6.2). Dadurch ist es möglich, die Precision genau zu errechnen, wenn man die erkannten Reviews manuell analysiert.

$$\begin{aligned} Precision &= \frac{\#_{relevante\ entnommene\ Elemente}}{\#_{entnommene\ Elemente}} \\ &\overset{\text{Hier}}{=} \frac{\#_{richtig\ erkannte\ Reviews}}{\#_{erkannte\ Reviews}} \end{aligned} \tag{6.2}$$

Ich habe Reviews aus drei Kategorien von Apps aus dem Google Play Store analysieren lassen, jeweils vor und nach Erstellen der neuen Patterns. Ich habe die Kategorie „Navigation“ gewählt (dabei sind bereits gelabelte Reviews aus dieser Kategorie ausgeschlossen) mit 3200 Reviews, dann die Kategorie „Productivity“ mit 2544 und die Kategorie „Finance“ mit 2591. Die Ergebnisse habe ich manuell analysiert, um festzustellen, welche Fälle richtig und welche falsch erkannt worden sind. Im Folgenden werden die Ergebnisse dieser Analyse erläutert.

Ergebnisse der gelabelten Daten

Zunächst werden die Ergebnisse der Daten zur App „HERE WeGo“ betrachtet. Die Daten sind hier gänzlich gelabelt, daher ist es möglich, den Recall und den F1-Score zu berechnen.

Abbildung 6.3 zeigt den Stand vor der Iteration. Dort werden 0 Reviews gefunden. Somit betragen Precision, Recall und somit auch der F1-Score den Wert 0. Dagegen zeigt Abbildung 6.4 den Stand nach der Iteration. Dort ist zu erkennen, dass alle 6 Reviews, die auch im Labeling gefunden wurden, richtig erkannt werden. Die Precision, der Recall und der F1-Score betragen hiermit den Maximalwert 1.

Dieses gute Ergebnis ist nicht überraschend. Da die Patterns auf diesen Datensatz zugeschnitten sind, ist ein sehr hoher F1-Score zu erwarten. Anders verhält es sich dagegen bei nicht gelabelten Daten, wie im nächsten Abschnitt beschrieben wird.

Ergebnisse der nicht gelabelten Daten

Nun werden die Ergebnisse der Daten bezüglich der drei Kategorien betrachtet. In diesem Fall handelt es sich um nicht gelabelte Daten. Daher kann keine Berechnung des Recall oder des F1-Score vorgenommen werden. Es kann nur eine relative Aussage getroffen werden.

Abbildung 6.3 zeigt den Stand vor der Iteration. Dort werden vergleichsweise wenige Reviews als Match erkannt. Die Precision für die jeweiligen Kategorien beträgt $0,8$ für „Navigation“ und $0,67$ für „Productivity“ und „Finance“. Für alle drei Kategorien zusammen beträgt die Precision $0,71$. Dagegen zeigt Abbildung 6.4 den Stand nach der Iteration. Hier werden deutlich mehr Reviews erkannt. Die Precision für die jeweiligen Kategorien beträgt $0,64$ für „Navigation“, $0,69$ für „Productivity“ und $0,64$ für „Finance“. Für alle drei Kategorien zusammen beträgt die Precision $0,66$. Die Precision nach der Iteration ist also um 0,05 geringer als die Vorherige.

Zum Recall kann nur eine relative Aussage getroffen werden. Nimmt man alle drei Kategorien zusammen, kommt man vor der Iteration auf eine Zahl von 10 richtig erkannten Reviews und danach auf eine Zahl von 43. Da die Anzahl der erkannten relevanten Fälle gestiegen ist, kann gesagt werden, dass

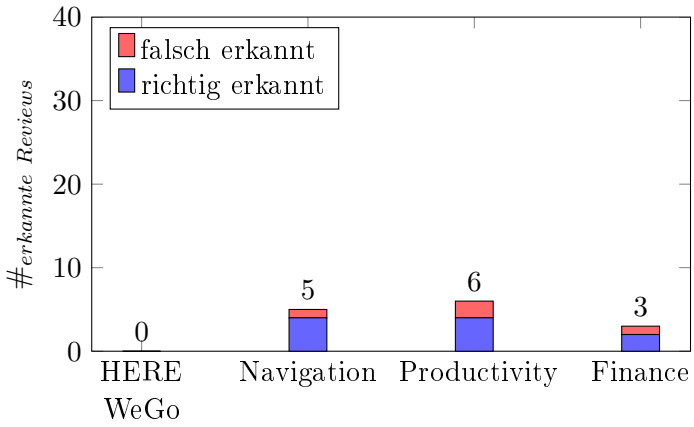

Abbildung 6.3: Auswertung von Matches vor der Iteration für die App HERE WeGo (0 richtig, 0 falsch) und die Kategorien „Navigation“ (4 richtig, 1 falsch), „Productivity“ (4 richtig, 2 falsch) und „Finance“ (2 richtig, 1 falsch).

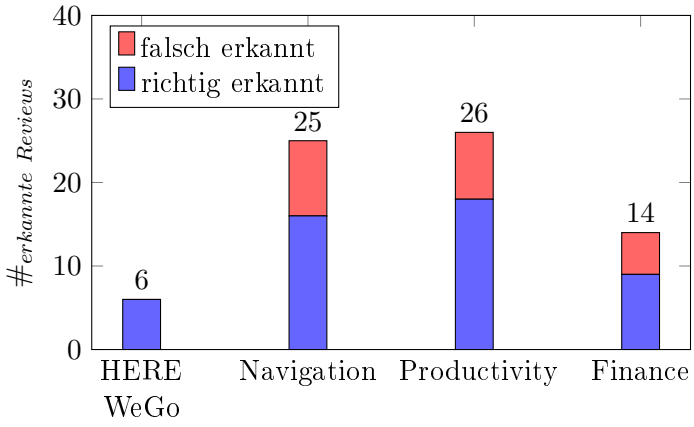

Abbildung 6.4: Auswertung von Matches nach der Iteration für die App HERE WeGo (6 richtig, 0 falsch) und die Kategorien „Navigation“ (16 richtig, 9 falsch), „Productivity“ (18 richtig, 8 falsch) und „Finance“ (9 richtig, 5 falsch).

der Recall gestiegen ist. Es kann auch gesagt werden, dass der Recall nach der Iteration um den Faktor $4,3$ höher ist als der Vorherige (siehe Gleichung 6.3). Dieser Wert kann errechnet werden, da durch die Betrachtung eines Verhältnisses kein Wissen um die Gesamtzahl der relevanten Fälle nötig ist.

$$\begin{aligned} Steigerung_{Recall} &= \frac{Recall_{nach\ Iteration}}{Recall_{vor\ Iteration}} \\ &= \frac{\#_{richtig\ erkannte\ Reviews\ nach\ Iteration}}{\#_{alle\ Reviews\ mit\ Erklärungsbedarf}} \times \\ &\quad \times \frac{\#_{alle\ Reviews\ mit\ Erklärungsbedarf}}{\#_{richtig\ erkannte\ Reviews\ vor\ Iteration}} \\ &= \frac{43}{10} = 4,3 \end{aligned} \tag{6.3}$$

Zum F1-Score kann keine Aussage getroffen werden, da nicht Precision *und* Recall gestiegen sind, sondern allein der Recall.

Zudem ist eine Besonderheit anzumerken: Obwohl in allen bisherigen Iterationen nur Reviews von Navigations-Apps gelabelt wurden, tritt die Kategorie „Navigation“ nicht etwa als besonders performant hervor. Es ist sogar so, dass die Kategorie „Productivity“ nach der Iteration mit einer Precision von $0,69$ bessere Ergebnisse liefert. Das lässt vermuten, dass sich das Aussehen von Erklärungsbedarf pro Kategorie kaum unterscheidet. Demnach wäre es nicht wichtig, beim Labeling möglichst viele App-Kategorien abzudecken.

6.3.2 Schlussfolgerung

Zwar kann bezüglich der nicht gelabelten Daten keine Angabe zum F1-Score gegeben werden, aber man kann trotzdem durch die Ergebnisse die Leistung einschätzen.

Zunächst wird die Leistung des Systems eingeschätzt. Bezüglich Daten, die bereits gelabelt worden sind, steigt die Leistung pro Iteration sehr. Wie zu erkennen ist, kann der F1-Score sich vom niedrigsten zum höchsten Wert innerhalb einer Iteration entwickeln. Man kann also einschätzen, dass die Systemleistung bezüglich gelabelter Daten sehr gut ist.

Bezüglich nicht gelabelter Daten sinkt die Precision durch die Iteration geringfügig und der Recall steigt um einen hohen Faktor. Das Ziel der Analyse ist ein hoher Recall bei Inkaufnahme einer weniger hohen Precision (siehe Abschnitt 6.3.1). Daher kann ausgesagt werden, dass das geringfügige Sinken der Precision sich nicht negativ auswirkt. Die hohe Steigerung des Recalls wirkt sich dagegen positiv aus. Abschließend kann gesagt werden, dass die Leistung des Systems bezüglich nicht gelabelter Daten durch die Iteration gestiegen ist. Aufgrund fehlender Messdaten für den Recall kann allerdings keine Aussage über die Güte der Leistung getroffen werden.

6.3.3 Prognose

Im Folgenden wird eine Prognose für zukünftigen Fortschritt der Analyse durch weitere Labeling-Iterationen gegeben.

Bezüglich gelabelter Daten ist weiterhin eine sehr gute Leistung zu erwarten. Bei nicht gelabelten Daten ist anzunehmen, dass der Recall weiter ansteigt. Solange die Precision nicht zu sehr sinkt, führt dies weiterhin zu einer steigenden Systemleistung. Die tatsächliche Systemleistung kann hier allerdings erst eingeschätzt werden, wenn bei der verwendeten Datenmenge die Gesamtanzahl der relevanten Fälle bekannt ist. Dazu wird eine manuelle Analyse dieser Datenmenge benötigt.

Durch die Leistungssteigerung in beiden Fällen (gelabelte und nicht gelabelte Daten) und durch die Möglichkeit, dass Nutzer das System weiter trainieren können, sehe ich in diesem Tool eine gute Hilfe für Requirements Engineers, die tatsächlich Einsatz finden könnte.

Kapitel 7

Zusammenfassung und Ausblick

In diesem Kapitel werden die Erkenntnisse dieser Arbeit zusammengefasst und Empfehlungen für nachfolgende Arbeiten gegeben.

7.1 Zusammenfassung

Ungeklärter Erklärungsbedarf kann für Nutzer ein Hinderungsgrund sein, eine App zu verwenden. Um darauf reagieren zu können, ist es daher wichtig, Erklärungsbedarf in Nutzer-Feedback zu erkennen. Requirements Engineers haben die Aufgabe, Anforderungen von Nutzern in die Entwicklung einfließen zu lassen. Damit sie große Mengen an Nutzer-Feedback auf Erklärungsbedarf analysieren können, ist eine Automatisierung sinnvoll.

Um Requirements Engineers zu unterstützen, habe ich ein grafisches Tool zur Analyse von Erklärungsbedarf entwickelt. Dabei nutzt es Techniken aus dem Bereich der künstlichen Intelligenz, um den Analyse-Vorgang zu automatisieren. Allerdings ist die automatische Analyse von Erklärungsbedarf nicht trivial. Es zeigt sich, dass Erklärungsbedarf oft schwierig zu erkennen und selten ist. Beim Labeln von 420 Einheiten an Nutzer-Feedback habe ich in 11 davon Erklärungsbedarf identifizieren können. Dieser kleine Datensatz reicht nicht für das Aufsetzen eines Deep-Learning-Systems.

Durch den Ansatz, Erklärungsbedarf anhand wiederkehrender Textsegmente zu erkennen, ist eine Analyse auch ohne großen Datensatz automatisierbar. Dafür wird der Matching-Ansatz der NLP-Bibliothek spaCy verwendet. Hier werden sogenannte Patterns eingesetzt, um bestimmte Textsegmente zu erkennen. Diese Patterns können so fein konfiguriert werden, dass beliebig viele und beliebig spezielle Textsegmente durch sie erkannt werden. Im implementierten Tool haben Nutzer außerdem die Möglichkeit die Analyse zu verbessern, indem sie Textsegmente auswählen können, die automatisch als Pattern in das System eingespeist werden.

Identifizierter Erklärungsbedarf kann durch Hervorhebung im Text einfach wahrgenommen werden. Die Bedienung des Tools wird erleichtert durch eine GUI mit hoher Usability.

Die implementierte Analyse kann nun zum Ende der Arbeit nur zu einer bestimmten Güte Erklärungsbedarf erkennen. Allerdings ist gezeigt worden, dass innerhalb einer Labeling-Iteration eine gute Leistungssteigerung erreicht werden kann. Man kann annehmen, dass es nur wenige weitere Iterationen braucht, um eine annehmbare bis gute Analyse-Leistung zu erreichen.

Es konnte gezeigt werden, dass es durch Techniken der künstlichen Intelligenz möglich ist, viele Fälle von Erklärungsbedarf zu erkennen. Somit können Requirements Engineers nachhaltig dabei unterstützt werden, für Nutzer besser verständliche Systeme zu entwickeln.

7.2 Ausblick

Da das Lösen von Problemen mit KI an der Größe von Datensätzen scheitern kann, ist die Forschung an Machine-Learning-Ansätzen, die ohne große Datensätze auskommen, ein relevantes Thema. Mit dem Matcher von spaCy konnte hier ein vielversprechender Ansatz vorgestellt werden, der auch in Zukunft bei Problemen ähnlicher Art helfen könnte. Allerdings wäre es in zukünftigen Projekten auch möglich, ein System zur Analyse von Erklärbarkeit auf Deep-Learning-Basis zu implementieren, solange ein genügend großer Datensatz vorhanden ist.

Da mithilfe der Erkennung von wiederkehrenden Textsegmenten kein semantischer Kontext von Texten analysiert werden kann, bleibt hier die Analyse von implizitem Erklärungsbedarf zum großen Teil ungelöst. Es besteht also weiterhin ein Bedarf für derartige Forschung.

Außerdem kann in Zukunft auch das im Zuge dieser Arbeit implementierte Tool weiterentwickelt werden. Zum einen kann die Analyse von Erklärungsbedarf durch weitere Labeling-Iterationen auf eine bessere Leistung gehoben werden. Zum anderen ist durch den modularen Ansatz eine simple Weiterentwicklung oder Umgestaltung des Tools möglich. Beispielsweise kann die Analyse weiterer NFRs oder die Unterstützung anderer Sprachen hinzugefügt werden. Wenn eine große Anzahl an NFRs dort zur Verfügung steht, kann es zu einem sehr nützlichen Tool für Requirements Engineers werden, auch im Rahmen kommerzieller Nutzung.

Anhang A

Installation

In diesem Anhang wird die Installation des implementierten Tools „Feedback Visualizer“ erklärt.

A.1 Java-Projekte für Feedback Visualizer

Die Java-Projekte bilden den Controller und Data Collection Layer. Sie finden sich im Ordner „/ba-jacobsen/workspace/app“.

- Zunächst die Spring Tools Suite[1] installieren.
- Importiere Java-Projekte: Importiere „jacobsen-app-controller“ und „jacobsen-app-crawler-kuhnke“ als Maven-Projekte. Importiere „jacobsen-app-concepts“ als Java-Projekt
- In „jacobsen-app-crawler-kuhnke“ wird die Bibliothek „Lombok“ verwendet. Diese muss in die Spring Tool Suite integriert werden. Dazu Lombok herunterladen[2] und die Schritte der Installation der Dokumentation[3] ausführen (lombok.jar ausführen und in Eclipse integrieren). Ggf. noch einmal das Maven-Projekt updaten.
- Classpath hinzufügen: Damit die Projekte sich kennen, muss der Classpath angepasst werden. Dazu bei „jacobsen-app-crawler-kuhnke“ das Projekt „jacobsen-app-concepts“ zum Classpath hinzufügen. Bei „jacobsen-app-controller“ müssen die Projekte „jacobsen-app-concepts“ und das Untermodul von „jacobsen-app-crawler-kuhnke“ namens „crawler-application“ hinzugefügt werden.
- Auf Datenbank anpassen: Das implementierte Tool wurde mithilfe einer MySQL-Datenbank aufgesetzt. Es empfiehlt sich,

[1] `https://spring.io/tools`
[2] `https://projectlombok.org/download`
[3] `https://projectlombok.org/setup/eclipse`

ebenfalls eine MySQL-Datenbank zu hosten. Um sie an das Tool anzuschließen, navigiere im Projekt „jacobsen-app-controller“ unter „src/main/resources“ zur Datei „application.properties“. Dort können die Datenbank-Konfigurationen angepasst werden.

- App starten: Navigiere zum Projekt „jacobsen-app-controller“ und darin zum Package „de.jac.springboot“. Mache dort einen rechten Mausklick auf die Klasse „FeedbackVisualizerApplication“ und führe den Punkt „Run as“ und dann „Spring Boot App“ durch.

A.2 Data Orchestration für Feedback Visualizer

Im Ordner „/ba-jacobsen/workspace/app/jacobsen-app-orchestration-react“ findet sich der Code, der den Data Orchestration Layer des Tools Feedback Visualizer bildet und das Framework React verwendet. Für die Bearbeitung eignet sich beispielsweise der Editor „Visual Studio Code“[4].

- Typescript: Für die Textmarkierungen wurde „react-text-annotate“ verwendet, was auf Typescript basiert. Daher muss Typescript installiert werden:

  ```
  npm install -g typescript
  ```

- React-App starten: In einem Terminal zum genannten Ordner navigieren und den Befehl

  ```
  npm start
  ```

  ausführen. Es öffnet sich automatisch der als Standard konfigurierte Browser und zeigt die Website „http://localhost:3000/“. Das ist eine lokale Anzeige der React-Implementierung. Mithilfe dieser Anzeige kann nun entwickelt werden.

- Weitere Hilfe: Unter src/README.md finden sich viele nützliche Informationen, um die Ordnerstrukturen zu verstehen.

A.3 Python-Umgebungen

In dieser Sektion geht es um das Einrichten der Python-Umgebungen, die zur Analyse von Text verwendet werden.

[4] `https://code.visualstudio.com/Download`

A.3.1 KI-Bibliotheken für Feeback Vizualizer

Im Ordner „/ba-jacobsen/workspace/app/jacobsen-app-analyze-python“ findet sich der Code, der den Data Analytics Layer des Tools Feedback Visualizer bildet. Für die Bearbeitung eignet sich beispielsweise der Editor „Visual Studio Code“.

- Python: Installer herunterladen[5] und installieren.
- spaCy: Installation mit

  ```
  pip install -U spacy
  ```

  Wenn ein Fehler kommt, könnte es an einem fehlerhaften Paket liegen, dann Folgendes ausführen:

  ```
  pip uninstall numpy
  ```

  und

  ```
  pip install numpy==1.19.3
  ```

  Zum Installieren von spaCy-Sprachmodellen[6] (hier wird „en_core_web_md“ verwendet) diesen Befehl ausführen:

  ```
  python -m spacy download en_core_web_md
  ```

- JSON: Wichtig für den Zugriff auf JSON-Dateien:

  ```
  pip install jsonschema
  ```

- flask: Ein python-Framework, das zum Zugriff per REST-Requests genutzt wird. Virtuelle python-Umgebung aktivieren

  ```
  analyze-python-env\Scripts\activate
  ```

  Danach flask installieren:

  ```
  pip install flask
  ```

  Jetzt die flask-App starten:

  ```
  py app.py
  ```

- Um mit dem React-Framework zusammenarbeiten zu können, sind Cross-Origin-Anfragen nötig, die aufgrund der Same-Origin-Policy von den meisten Browsern geblockt werden. Daher wird Flask-CORS verwendet. Hierzu mit diesem Befehl installieren:

  ```
  pip install -U flask-cors
  ```

[5] `https://www.python.org/downloads/`
[6] `https://spacy.io/models/en`

A.3.2 KI-Exploration

Im Ordner „/ba-jacobsen/workspace/ki_exploration/Kapitel Implementierung - Datenexploration“ finden sich Ressourcen zum Experimentieren mit Patterns. Diese verwenden den Editor Jupyter-Lab. Zusätzlich zu den im letzten Abschnitt genannten Schritten sind Folgende auszuführen, um die Ressourcen zu verwenden.

- Installiere Jupyter-Lab:

  ```
  pip install jupyterlab
  ```

  Starte Jupyter-Lab durch Navigation zum genannten Ordner, dann Folgendes ausführen:

  ```
  jupyter-lab
  ```

- Installation der Pandas-Bibliothek für CSV-Interaktion mit diesem Befehl:

  ```
  pip install pandas
  ```

A.4 Erweiterung um weitere NFRs

Da das Konzept Modularität im Vordergrund der Arbeit gestanden hat, ist die Erweiterung in wenigen Schritten möglich. Um NFRs hinzuzufügen, muss lediglich eine Enum-Klasse im Java-Projekt „jacobsen-app-concepts“ bearbeitet werden. Ein Enum-Wert stellt dort ein NFR dar. Durch Hinzufügen weiterer Enum-Werte können NFRs hinzugefügt werden.

Um die Analyse weiterer NFRs zu ermöglichen, muss Python-Code im Data Analytics Layer hinzugefügt werden. In der Funktion „analyze_reviews()“ der Datei „/ba-jacobsen/workspace/app/jacobsen-app-analyze-python/app.py“ ist bislang nur die Analyse anhand von Erklärungsbedarf implementiert. Hier kann die Handhabung weiterer NFRs definiert werden.

A.5 Modifikation der Patterns

Die Dateien zur Erzeugung von Patterns finden sich unter „/ba-jacobsen/workspace/app/jacobsen-app-analyze-python“. Zum einen werden dort durch Entwickler vordefinierte Patterns durch Python-Code gespeichert. Diese finden sich in der Datei „explNeedPatterns.py“. Zum anderen gibt es durch Trainingsdaten der Nutzer automatisch generierte Patterns, die als JSON gespeichert werden. Es gibt eine JSON-Datei für hinzugefügte

(„customPatternsMatch.json“) und eine für ausgeschlossene Patterns („customPatternsExclude.json“).

Es empfiehlt sich, in weiteren Labeling-Iterationen nur die vordefinierten Patterns zu betrachten, sie weiterzuentwickeln und zu verfeinern. So erreicht man die höchste Initialleistung. Die JSON-Dateien sind eher für die Modifikation durch Nutzer des Programms gedacht. Allerdings können auch diese modifiziert und verfeinert werden.

Literaturverzeichnis

[1] L. Chazette and K. Schneider. Explainability as a non-functional requirement: challenges and recommendations. In *Requirements Engineering*, volume 25, pages 493–514. Springer, 2020.

[2] G. Fischer. Symmetry of ignorance, social creativity, and meta-design. *Knowledge-Based Systems*, 13(7):527–537, 2000.

[3] M. Glinz. On non-functional requirements. In *15th IEEE International Requirements Engineering Conference (RE 2007)*, pages 21–26, 2007.

[4] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning. Das umfassende Handbuch: Grundlagen, aktuelle Verfahren und Algorithmen, neue Forschungsansätze.* mitp, 2018.

[5] B. Goodman and S. Flaxman. European union regulations on algorithmic decision-making and a "right to explanation". *AI Magazine*, 38(3):50–57, Oct. 2017.

[6] E. C. Groen, S. Kopczyńska, M. P. Hauer, T. D. Krafft, and J. Doerr. Users — the hidden software product quality experts?: A study on how app users report quality aspects in online reviews. In *2017 IEEE 25th International Requirements Engineering Conference (RE)*, pages 80–89, 2017.

[7] S. Gärtner and K. Schneider. A method for prioritizing end-user feedback for requirements engineering. In *2012 5th International Workshop on Co-operative and Human Aspects of Software Engineering (CHASE)*, pages 47–49, 2012.

[8] M. Kuhnke. Identifizierung von Erklärungsbedarf via User-Feedback-Analyse. Masterarbeit, Leibniz Universität Hannover, Fachgebiet Software Engineering, 2020.

[9] W. Maalej and H. Nabil. Bug report, feature request, or simply praise? on automatically classifying app reviews. In *2015 IEEE 23rd International Requirements Engineering Conference (RE)*, pages 116–125, 2015.

[10] D. Malavalli and S. Sathappan. Scalable microservice based architecture for enabling dmtf profiles. In *2015 11th International Conference on Network and Service Management (CNSM)*, pages 428–432, 2015.

[11] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to Information Retrieval*. Cambridge University Press, Cambridge, United Kingdom, 2009.

[12] J. Nivre, M.-C. De Marneffe, F. Ginter, Y. Goldberg, J. Hajic, C. D. Manning, R. McDonald, S. Petrov, S. Pyysalo, N. Silveira, et al. Universal dependencies v1: A multilingual treebank collection. In *Proceedings of the Tenth International Conference on Language Resources and Evaluation (LREC'16)*, pages 1659–1666, 2016.

[13] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes. Sourcerercc: Scaling code clone detection to big code. *Proceedings of the 38th International Conference on Software Engineering*, May 2016.

[14] K. Schneider. Lecture: Grundlagen der Softwaretechnik. From slide set. Winter semester 2018/2019.

[15] C. Stanik. *Requirements Intelligence: On the Analysis of User Feedback*. PhD thesis, Universität Hamburg, 1995.

[16] R. Tomsett, D. Braines, D. Harborne, A. Preece, and S. Chakraborty. Interpretable to whom? A role-based model for analyzing interpretable machine learning systems, 2018.