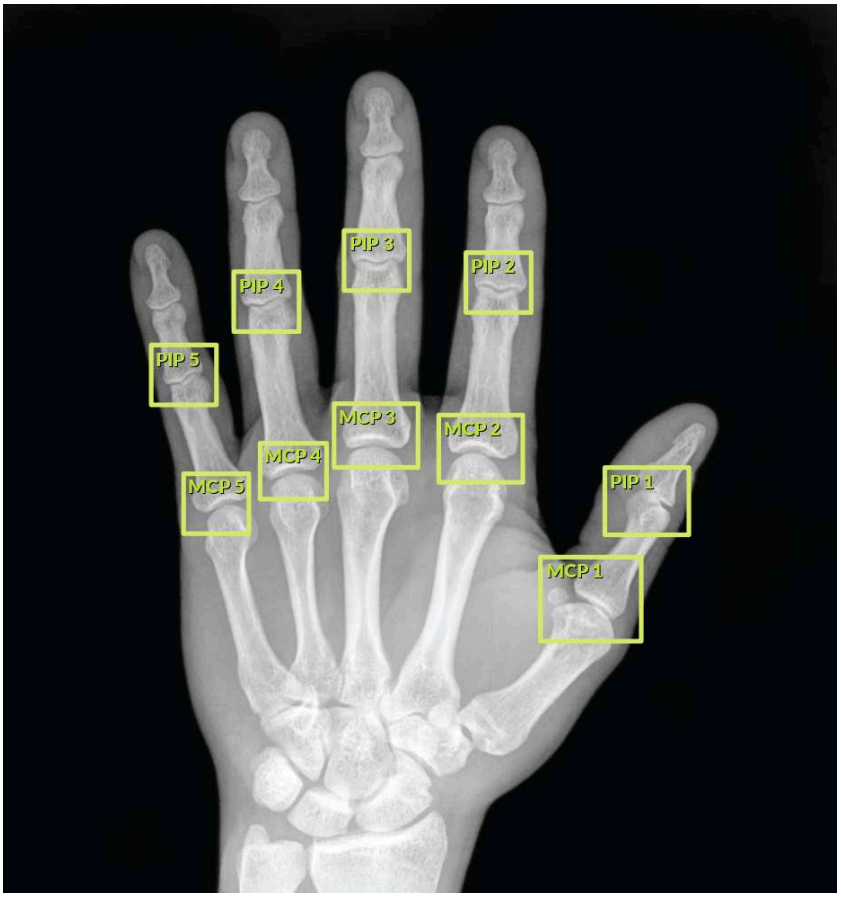
DeepXRay

Bachelorarbeit, Frühlingssemester 2020, HSLU – Informatik

Patrick Bucher

27. Mai 2020



Auftraggeber	Betreuer	Experte
Dr. Tobias Reinhard	Daniel Pfäffli	Jeremy Callner
Seantis GmbH	HSLU – Informatik	APG|SGA

Bachelorarbeit an der Hochschule Luzern – Informatik

Titel: DeepXRay

Student: Patrick Bucher

Studiengang: BSc Informatik

Jahr: 2020

Betreuungsperson: Daniel Pfäffli, HSLU – Informatik

Experte: Jeremy Callner, APG|SGA

Auftraggeber: Tobias Reinhard, Seantis GmbH

Codierung/Klassifizierung der Arbeit:

☒ A: Einsicht (Normalfall)
☐ B: Rücksprache (Dauer: ___ Jahr/Jahre)
☐ C: Sperre (Dauer: ___ Jahr/Jahre)

Eidesstattliche Erklärung Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig und ohne unerlaubte fremde Hilfe angefertigt habe, alle verwendeten Quellen, Literatur und andere Hilfsmittel angegeben habe, wörtlich oder inhaltlich entnommene Stellen als solche kenntlich gemacht habe, das Vertraulichkeitsinteresse des Auftraggebers wahre und die Urheberrechtsbestimmungen der Fachhochschule Zentralschweiz (siehe Merkblatt «Studentische Arbeiten» auf MyCampus) respektieren werde.

Ort/Datum, Unterschrift _____

Abgabe der Arbeit auf der Portfolio-Datenbank:

Bestätigungsvisum Student

Ich bestätige, dass ich die Bachelorarbeit korrekt gemäss Merkblatt auf der Portfolio-Datenbank abgelegt habe. Die Verantwortlichkeit sowie die Berechtigungen habe ich abgegeben, so dass ich keine Änderungen mehr vornehmen kann oder weitere Dateien hochladen kann.

Ort/Datum, Unterschrift _____

Verdankung

Ich bedanke mich bei meinem Arbeitgeber Seantis GmbH, insbesondere beim Auftraggeber Dr. Tobias Reinhard dafür, dass ich die Möglichkeit erhalten habe, eine interessante und herausfordernde Arbeit im Umfeld der Firma erstellen zu dürfen, und für die Hilfestellungen im Verlauf der Arbeit. Bei der SCQM bedanke ich mich für die zur Verfügung gestellten Röntgenbilder, die mir als Evaluationsdaten dienten. Bei Janick Rohrbach und Dr. Tobias Reinhard bedanke ich mich für ihre Vorarbeit im Bereich Machine Learning, ohne die das vorliegende Projekt nicht möglich gewesen wäre.

Ausschliesslich bei Abgabe in gedruckter Form:
Eingangsvisum durch das Sekretariat auszufüllen

Rotkreuz, den _____ Visum: _____

Hinweis: Die Bachelorarbeit wurde von keinem Dozierenden nachbearbeitet. Veröffentlichungen (auch auszugsweise) sind ohne das Einverständnis der Studiengangleitung der Hochschule Luzern – Informatik nicht erlaubt.

Abstract

Die rheumatoide Arthritis ist eine Autoimmunerkrankung, bei der Gelenke angegriffen und irreversibel geschädigt werden. Die Erosion des Gelenkgewebes wird durch medizinisches Fachpersonal anhand von Röntgenbildern ermittelt, was pro Patient mehrere Minuten in Anspruch nimmt.

In einer Vorarbeit haben Janick Rohrbach (ZHAW), Tobias Reinhard (Seantis GmbH), Beate Sick (Universität Zürich) und Oliver Dürr (HTWG Konstanz) diesen Vorgang mithilfe von *Deep Convolutional Neuronal Networks* automatisiert, wodurch eine gleichbleibende Bewertungsqualität gewährleistet wird. Das Machine-Learning-Modell wurde mit zehntausenden von klassifizierten Gelenken aus der SCQM-Datenbank (*Swiss Clinical Quality Management in Rheumatic Disease*) trainiert, validiert und getestet.

Die vorliegende Arbeit beschreibt, wie auf Basis dieses und zweier weiterer Machine-Learning-Modelle (Erkennung von Körperteilen und Extraktion von Gelenken auf Röntgenbildern) ein Webservice erstellt wird, womit das automatische Scoring von Gelenken anderen Anwendungen zur Verfügung gestellt werden kann. Der resultierende Prototyp ist eine verteilte Anwendung, welche die bestehenden Machine-Learning-Modelle mithilfe von Messaging zu einem lose gekoppelten System zusammenfügt. Diese Architektur erlaubt es, mehrere Instanzen pro Modell auszuführen, wodurch die Röntgenbilder schnell und zuverlässig verarbeitet werden können.

Die Qualität des Gesamtsystems wird mit verschiedenen Metriken evaluiert. Damit können Qualitätsverbesserungen des Gesamtsystems beurteilt werden, wenn einzelne Machine-Learning-Komponenten durch neuere, verbesserte Versionen ersetzt werden.

Im Ausblick wird behandelt, welche weiteren Schritte notwendig wären, um das System beim Auftraggeber in den Produktiveinsatz überführen zu können.

Inhaltsverzeichnis

1 Problemstellung

Bei der rheumatoiden Arthritis handelt es sich um eine Autoimmunerkrankung, bei der sich das Gelenkgewebe entzündet. Halten diese Entzündungen über längere Zeit an, werden die betroffenen Gelenke dadurch irreversibel geschädigt.

Die Erosion des Gelenkgewebes wird durch Fachpersonal anhand von Röntgenbildern eingestuft. Hierzu wird die Ratingen-Score verwendet, die das Fortschreiten der Erosion auf einer Skala von 0% (gesundes Gelenk) bis bis 100% (Gelenkgewebe vollständig erodiert) abbildet. Diese Einschätzung wurde bisher manuell vorgenommen, was pro Röntgenbild mehrere Minuten dauert. Dieser Vorgang wird als *Scoring* bezeichnet.

Im Rahmen einer Bachelorarbeit an der ZHAW (Rohrbach, 2017) und in einem Folgeprojekt im Rahmen einer Masterarbeit (nicht öffentlich verfügbar) haben sich Janick Rohrbach (Zürcher Hochschule für Angewandte Wissenschaften), Tobias Reinhard (Seantis GmbH), Beate Sick (Universität Zürich) und Oliver Dürr (HTWG Konstanz) mit der Automatisierung des Scorings befasst (Rohrbach, Reinhard, Sick & Dürr, 2019). Dadurch soll der Vorgang einerseits beschleunigt werden, andererseits soll auch eine gleichbleibende Bewertungsqualität gewährleistet werden. Diese Verbesserungen dürften sich gerade bei umfassenden und langfristigen Studien als hilfreich erweisen.

Die automatische Auswertung von Bildern mit klassischen Machine-Learning-Methoden gestaltete sich bisher als schwierig. Mithilfe von *Deep Convolutional Neuronal Networks*, die geeignete Merkmale von Bildern selbständig herausarbeiten, konnten auf diesem Gebiet in den letzten Jahren grosse Fortschritte erzielt werden.

Auf Basis von Röntgenbildern aus der SCQM[1]-Datenbank konnte ein entsprechendes Modell mit zehntausenden klassifizierten Gelenken trainiert, validiert und getestet werden. Hierbei stellte die Unausgeglichenheit der Testdaten – ca. zwei Drittel der Röntgenbilder zeigen gesunde Gelenke – eine grosse Herausforderung dar. Dennoch konnte ein Ergebnis erzielt werden, das sich sehen lässt: So stimmen die Bewertungen des automatischen Modells besser mit denjenigen von menschlichen Bewertungen überein, als die menschlichen Bewertungen untereinander.

Das Modell, bzw. die verschiedenen Modelle, die für den Scoring-Vorgang zum Einsatz kommen – Erkennung von Körperteilen, Extraktion von Gelenken, Scoring von Gelenken – beschränkt sich dabei auf linke Hände, bzw. auf zwei Arten von Gelenken: die Fingermittelgelenke zwischen dem ersten und zweiten Fingerglied jedes Fingers (proximale Interphalangealgelenke) und die Gelenke zwischen den Mittelhandknochen und dem ers-

[1] `https://www.scqm.ch/ueber-uns/` (abgerufen am 10.05.2020)

ten Fingerglied (Metacarpophalangealgelenke). Die Gelenke, jeweils von 1 (Daumen) bis 5 (kleiner Finger) nummeriert, sind auf Abbildung 1, Seite 10 eingezeichnet.

Mit dem trainierten, validierten und erfolgreich getesteten Modell und dem dazu publizierten Fachartikel *Bone Erosion Scoring for Rheumatoid Arthritis with Deep Convolutional Neural Networks*, (Rohrbach et al., 2019) ist der wissenschaftliche Grundstein für den produktiven Einsatz der automatischen Bewertung von Gelenken gemäss der Ratingen-Score gelegt. Bis dahin gibt es aber noch Einiges zu tun: Die erstellten Modelle müssen über einen Webservice zur Verfügung gestellt werden, damit sie in der Forschung zum Einsatz kommen können. Diese Aufgabe ist Gegenstand der vorliegenden Bachelorarbeit.

Geht es in der erwähnten Publikation (Rohrbach et al., 2019) v.a. um die Aspekte des Scorings im Bereich Machine Learning, liegt der Schwerpunkt in dieser Folgearbeit im Engineering-Bereich. Zunächst müssen die erstellten Modelle zu einer Software kombiniert werden, die als Web-Anwendung ansprechbar ist. Weiter muss sichergestellt werden, dass das automatische Scoring in nützlicher Frist durchgeführt werden kann – gerade wenn das System unter hoher Last steht. Hierzu empfiehlt sich die Parallelisierung des Vorgangs: Die Röntgenbilder werden auf verschiedene Ausführungseinheiten verteilt. Dort wird das dargestellte Körperteil erkannt, es werden die relevanten Gelenke extrahiert und auf der Ratingen-Skala gescored. Am Schluss werden die ermittelten Scores als Gesamtergebnis zurückgeliefert.

Zudem sollen die Machine-Learning-Modelle austauschbar gemacht werden, sodass Verbesserungen auf diesem Gebiet schnell und einfach in den Produktivbetrieb einfliessen können. Hierbei ist es wichtig, dass die Qualität und Leistungsfähigkeit verschiedener Versionen miteinander verglichen werden können.

Das Ergebnis der Arbeit soll ein Prototyp für einen Webservice sein, der in bestehende Software eingebunden werden kann, um so ein automatisiertes, schnelles und präzises Scoring von Gelenken zu ermöglichen. Dadurch soll der Krankheitsverlauf von rheumatoider Arthritis besser nachverfolgt werden können, was etwa bei der Durchführung von Medikamentenstudien hilfreich sein kann.

1.1 Projektauftrag

Ziel der Arbeit ist es, den *Proof of Concept*, der für die wissenschaftliche Publikation erstellt worden ist (Rohrbach et al., 2019), für die Anwendung in der Praxis tauglich zu machen. Auf Basis der bestehenden Modelle, bzw. auf Basis des Codes zum Erstellen derselben, ist ein Deployment zu entwerfen und umzusetzen, das den Anforderungen des produktiven Betriebs genügt.

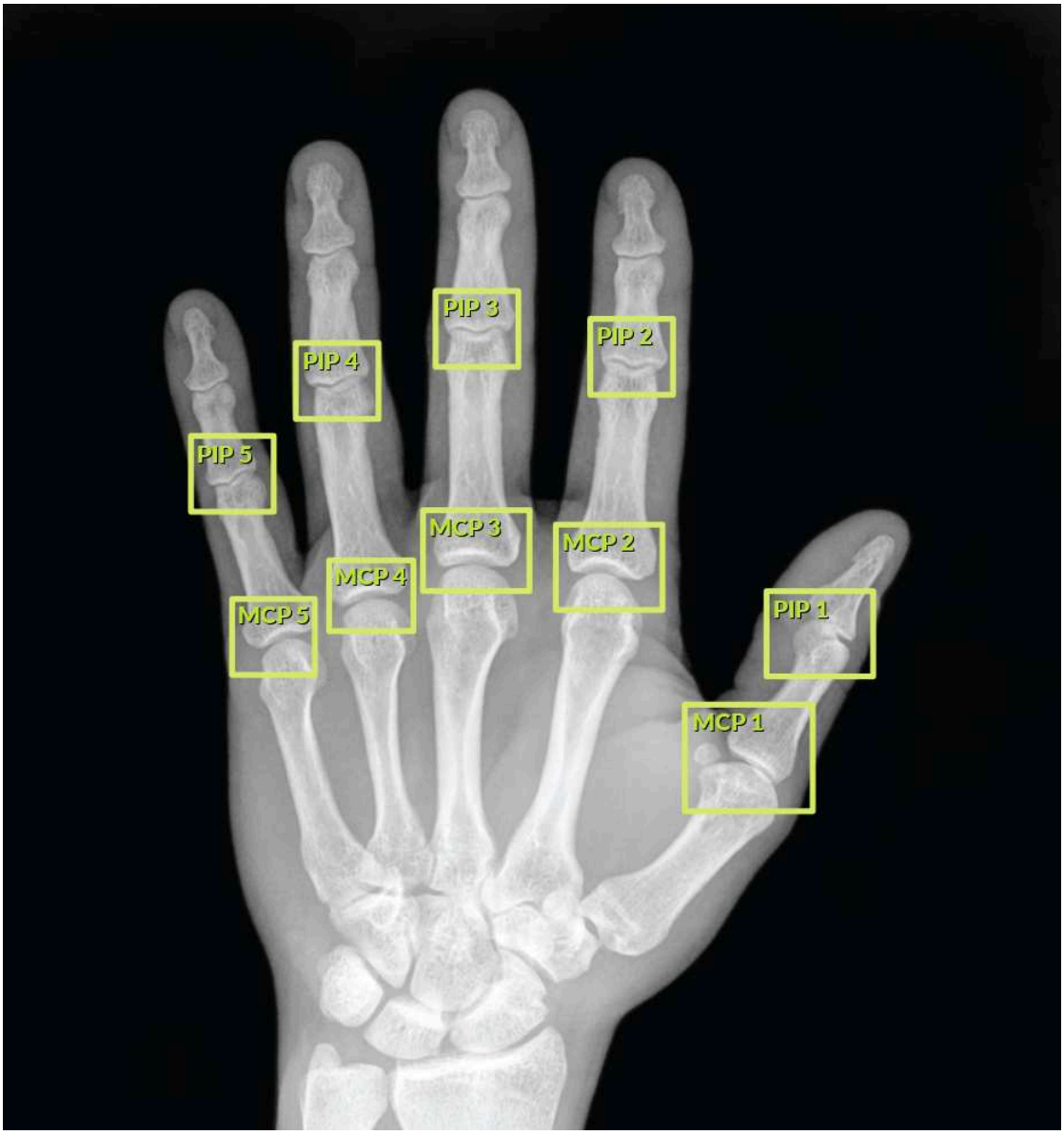


Abbildung 1: Röntgenbild einer linken Hand mit markierten Metacarpophalangealgelenken (MCP) und proximalen Interphalangealgelenken (PIP), Quelle: *OpenStax Anatomy and Physiology*, gefunden unter `https://commons.wikimedia.org/wiki/File:01_16_X-ray_of_Hand.jpg` (Lizenz: CC BY 4.0); Beschriftungen d.A.

Die bestehenden Modelle sind in verschiedenen Versionen von TensorFlow umgesetzt worden. Zwar sollen die Modelle im Rahmen der vorliegenden Arbeit nicht auf eine aktuelle Version von TensorFlow migriert werden, es soll aber Teil der Arbeit sein, aufzuzeigen, was hierzu nötig wäre.

1.1.1 Vorgaben

Sämtliche Modelle, die für das Scoring eines Röntgenbildes benötigt werden, sind gegeben und vortrainiert. Sie erfüllen Minimum-Standars was ihre Performance betrifft.

1.1.2 Erwartetes Resultat

Neben den für jede Bachelorarbeit zwingenden Resultate – Schlussbericht (vorliegendes Dokument), Zwischen- und Schlusspräsentation, Web-Abstract, Einführungsvideo – werden folgende Resultate vom Projekt erwartet:

1. Stand der Technik zur Industrialisierung (Evaluation, Testing) von ML-Modellen
2. Vorschlag für ein generisches Format von Modellen zum Exportieren aus ML-Frameworks und zum Importieren zur Ausführung
3. Vorschlag für geeignete Metriken zum Vergleich von Modellen
4. Aufzeigen, was für eine Vereinheitlichung der Codebasis gemacht werden muss
5. Lauffähiger Protoyp
    a) Automatisierte Evaluation und Testing von neuen Modellen mit Vergleich zu alten Modellen und Minimum-Standards
    b) Visualisierung und Reporting der Performanz neuer Modelle
    c) API zum Ausführen des Modelles (Verteilen über mehrere GPUs)
    d) Load Balancing, Messaging Systems
    e) Nahtloses Austauschen von alten Modellen mit neuen Versionen

1.1.3 Abgrenzung

Das Trainieren der Modelle ist nicht Teil der Arbeit. Die Modelle können als Blackbox betrachtet werden. Dementsprechend genügt es, sie aus einer High-Level-Sicht zu beschreiben.

Die effektive Integration in die Produktivumgebung von Seantis, sprich die Plattform *HealthData.ai*[2], ist nicht Teil der Arbeit. Die hierzu notwendigen Schritte sollen jedoch beschrieben werden.

1.2 Projektrisiken

Im Folgenden werden verschiedene Projektrisiken und mögiche Mitigationsmassnahmen für diese aufgelistet:

Keine lauffähigen Modelle Die verschiedenen Modelle wurden teils vor mehreren Jahren entwickelt und trainiert. Ob sie tatsächlich funktionieren, und ob die Modelldaten noch in einem konsistenten Zustand auffindbar sind, ist zu Beginn des Projekts nicht klar.

Risiko Eines oder mehrere Modelle sind nicht funktionstüchtig oder gar verloren.

Mitigation Anhand von bestehendem Quellcode kann versucht werden, die Modelle neu zu erstellen. Scheitert dies, müssten fehlende Modelle durch Platzhalter (Mocks) ersetzt werden, die Predictions und ein bestimmtes Laufzeitverhalten simulieren.

Aktualisierung der Modelle Die zugrundeliegenden Modelle sind teils schon etwas älter und in unterschiedlichen Versionen bzw. mit unterschiedlichen Frameworks entwickelt worden.

Risiko Die Modelle können nicht mit aktuellen Versionen der zugrundeliegenden Frameworks importiert werden.

Mitigation Die Modelle werden mittels geeigneter Massnahmen (Container, virtuelle Umgebung) isoliert, sodass sie in ihrer ursprünglichen «Trainigsumgebung» ausgeführt werden können.

Schlechte Performance (Prediction) Da verschiedene Modelle zum Einsatz kommen, die bisher nie dem produktiven Betrieb ausgesetzt gewesen sind, kann es sein, dass deren Performance im Bezug auf die Qualität der Predictions schlecht ist.

Risiko Eines oder mehrere Modelle weisen eine schlechte Prediction-Performance auf.

Mitigation Dieses Risiko muss getragen werden. Die Prediction-Performance des Gesamtsystems ist durch das schwächste Modell nach oben begrenzt.

[2] `https://www.healthdata.ai/en` (abgerufen am 27.05.2020)

Schlechte Performance (Laufzeit) Für ein Scoring eines Röntgenbildes sind verschiedene Modelle in mehreren Schritten involviert.

Risiko Die Laufzeit-Performance könnte so schwach ausfallen, dass ein produktiver Betrieb dadurch inpraktikabel wird.

Mitigation Mit geeigneten architektonischen Massnahmen (parallele Abarbeitung, Ausführung auf GPUs) kann die Laufzeit-Performance erhöht werden. Genügt dies nicht, muss auf ein synchrones Ansprechen des Gesamtsystems zugunsten einer anderen, d.h. asynchronen Lösung verzichtet werden.

Hoher Arbeitsspeicherverbrauch Da mehrere Modelle für das Scoring eines Röntgenbildes benötigt werden, kann dies zu einem hohen Arbeitsspeicherverbrauch führen.

Risiko Die Modelle und ihre umgebende Laufzeitumgebung können nicht gleichzeitig in den Arbeitsspeicher eines Systems geladen werden.

Mitigation Der Prototyp kann auf einem stärker bestückten System (etwa bei einem Cloud-Anbieter) ausgeführt oder über mehrere Systeme verteilt werden.

2 Stand der Praxis, Forschung und Technik

In diesem Kapitel geht es um den gegenwärtigen Stand der Praxis, der Forschung und der Technik.

Im Praxisteil werden die bereits bestehenden Machine-Learning-Modelle (siehe 2.1 *Bestehende Modelle*, Seite 14) betrachtet und in ihrer Funktionsweise beschrieben.

Was die Forschung betrifft, werden Metriken zur Evaluation von Machine-Learning-Modellen vorgestellt und dabei im Bezug auf ihre Eignung für das vorliegende Problem besprochen (siehe 2.4.3 *Mögliche Evaluationsmetriken*, Seite 25).

Der Technikteil umfasst für diese Arbeit im Engineering-Bereich mehrere Themen: Zunächst soll untersucht werden, was mit dem Begriff des Webservices genau gemeint ist (siehe 2.2 *Webservices*, Seite 18). Anschliessend wird auf verschiedene Varianten eingegangen, wie verschiedene Komponenten zu einem funktionierenden Gesamtsystem kombiniert werden können (siehe 2.3 *Integrationsvarianten*, Seite 19). Konkrete Vorschläge für eine Systemarchitektur werden auf dieser Basis erst im Folgekapitel (siehe 3.1 *Architekturvarianten*, Seite 31) besprochen. Zum Schluss werden verschiedene Formate für Machine-Learning-Modelle untersucht und bewertet (siehe 2.5 *Modellformate*, Seite 29).

2.1 Bestehende Modelle

Für das Ermitteln der Ratingen-Score anhand eines Röntgenbildes sind mehrere Schritte nötig.

Zunächst muss das auf dem Röntgenbild abgebildete Körperteil erkannt werden. Hierfür ist das Modell `body_part` zuständig.[3] Dieses Modell kann verschiedene Körperteile erkennen, u.a. Hände, Füsse (jeweils mit links/rechts-Unterscheidung) und Becken.

Im nächsten Schritt werden aus dem Röntgenbild die relevanten Gelenke extrahiert. Das Modell (bzw. die Modellsammlung) `joint_detection` kann neben den proximalen Interphalangealgelenken (PIP) und den Metacarpophalangealgelenken (MCP) auch das Karpalgelenk (vulgo: «Handgelenk») auf einem Röntgenbild erkennen und extrahieren.

Im dritten und letzten Schritt wird das Scoring für ein bestimmtes Gelenk auf einem entsprechenden Bildausschnitt vorgenommen. Hierfür ist das Modell `ratingen_score` zuständig.[4] Dieses Modell unterstützt nur das Scoring der PIP- und MCP-Gelenke der

[3] Die Namen der Modelle in diesem und in den weiteren Kapiteln wurden der Verzeichnisstruktur entnommen, in der die entsprechenden Daten abgelegt worden waren.

[4] Dieser Name wurde nicht anhand einer Verzeichnisstruktur, sondern anhand der Funktionalität des Modells ausgesucht.

linken Hand.

Die Menge der möglichen Inputs wird somit in dieser Kette von Modellen wie ein Trichter verengt: Können zu Beginn Hände, Füsse, Becken usw. erkannt werden, ist das Scoring am Schluss nur für zehn Gelenke (PIP 1-5 und MCP 1-5) der linken Hand möglich.

Dadurch verkommt das recht mächtige Modell `body_part` im Rahmen dieser Arbeit zu einer blossen Validierungsengine: Zeigt das Röntgenbild eine linke Hand? Wenn ja, kann der Prozess weitergeführt werden. Wenn nein, kann auf weitere Schritte verzichtet werden, denn ein Scoring wird damit nicht funktionieren.[5] Diese Restriktion kann möglicherweise zu einem späteren Zeitpunkt aufgehoben werden, sollten die Downstream-Modelle `joint_extraction` und `ratingen_score` entsprechend erweitert werden.

2.1.1 Erkennung von Körperteilen: `body_part`

Das Modell `body_part` kann, wie bereits beschrieben, auf Röntgenbildern verschiedene Körperteile erkennen. Im Kontext dieser Arbeit ist jedoch nur relevant, ob ein Röntgenbild (Input) eine linke Hand darstellt oder nicht.

Der Input für das Modell ist eine Liste monochromer Bilder von 144 mal 144 Pixeln. Technisch gesprochen ist dies ein NumPy-Array mit der Dimension `(n, 144, 144, 1)`, wobei `n` für die Anzahl der Bilder in der Liste steht.

Im Code zum Trainieren des Modells werden verschiedene Röntgenbilder zunächst auf die entsprechende Auflösung reduziert, d.h. gestaucht. Dieser Schritt ist für den produktiven Einsatz mit Sicherheit nötig, soll der Benutzer Bilder in verschiedenen Auflösungen verwenden können. Eine Seitenverhältnis von 1:1 (quadratische Form) dürfte dabei hilfreich sein, da das Bild ansonsten gestaucht werden muss.

Der Output des Modells ist eine Liste von Wahrscheinlichkeiten – bzw. eine Liste davon, denn für `n>1` werden mehrere Bilder in einem Prediction-Schritt verarbeitet. Die innere Liste ist ein NumPy-Array mit Fliesskommazahlen im Bereich $[0..1]$. Diese beschreiben, mit welcher Wahrscheinlichkeit ein Bild ein bestimmtes Körperteil zeigt.

Da im Rahmen der vorliegenden Arbeit nur linke Hände von Belang sind, kann das Element mit dem Index 6 (`left hand` in der Label Map) extrahiert werden. Ein bestimmter Threshold für die Wahrschienlichkeit könnte nachgelagert geprüft werden.

[5] Rein technisch könnten erkannte rechte Hände gespiegelt und an die Extraktion weitergeleitet werden. Auch wäre es möglich, etwa Röntgenbilder von Füssen an die Extraktion weiterzuleiten, in der Hoffnung, darauf entsprechende Gelenke zu erkennen. Solche Versuche sollen im Rahmen dieser Arbeit kategorisch unterlassen werden, da das Modell `ratingen_score` ausschliesslich mit Gelenken von linken Händen trainiert, evaluiert und getestet worden ist.

Das Modell wurde mit den Libraries `tensorflow` (Version 0.12.1) und `tflearn` (Version 0.2.1) erstellt und im `tflearn`-Format abgespeichert. Weiter kommt `scikit-image` in einer dazu kompatiblen Version zum Einsatz. Es ist mit einem Docker-Image basierend auf dem Image `python:3.6` lauffähig.

2.1.2 Extraktion von Gelenken: `joint_detection`

Wie in der Kapitelanleitung bereits erwähnt, handelt es sich bei `joint_detection` um eine ganze Reihe von Modellen, elf an der Zahl: für die zehn Gelenke MCP 1-5 und PIP 1-5, sowie für das Karpalgelenk (Handgelenk). Letzteres ist für die vorliegende Arbeit nicht relevant.

Das Modell extrahiert aus einem Röntgenbild (JPEG-Format), welches über das Dateisystem zur Verfügung gestellt wird, ein bestimmtes Gelenk. Der Gelenkname muss dabei als Parameter mitgegeben werden, z.B. `mcp3` oder `pip5`. Der Output ist eine JPEG-Datei der Grösse 150 mal 150 Pixel.[6]

Es wird somit pro Gelenk eine Datei extrahiert. Hierbei fällt auf, dass das Upstream-Modell `body_part` mit Bildern von 144 mal 144 Pixeln für das ganze Röntgenbild arbeitet, während in den `joint_detection`-Modellen jedes der zehn zu detektierenden Elemente mit einer Auflösung von 150 mal 150 Pixeln ausgegeben wird. Der initiale Bild-Input sollte also möglichst hochauflösend sein.

Die `joint_detection`-Modelle wurden ebenfalls mit `tensorflow` Version 0.12.1 erstellt und im `tflearn`-Format abgelegt. Als weitere Libraries kommen `Pillow` (Version 3.4.2) und `scipy` (Version 0.18.1) zum Einsatz. Das verwendete Docker-Image basiert wiederum auf dem Image `python:3.6`.

2.1.3 Scoring von Gelenken: `ratingen_score`

Das Modell `ratingen_score`, welches für das eigentliche Scoring zuständig ist, wurde nicht bei Seantis entwickelt, sondern im Rahmen einer Bachelorarbeit (Rohrbach, 2017). Es ordnet einem extrahierten Bild eines Gelenkes eine Score von 0 bis 5 zu. Ein gesundes

[6] Strenggenommen hat das Modell zwei Outputs: erstens eine Reihe sogenannter *Boxes*, die je durch vier Gleitkommazahlen definiert sind; zweitens eine Reihe von *Confidences*, bestehend aus je zwei Gleitkommazahlen. Auf die genaue Bedeutung dieser Werte soll hier nicht weiter eingegangen werden, zumal die Modelle nur aus einer High-Level-Sicht zu beschreiben sind (siehe 1.1.2 *Erwartetes Resultat*, Seite 11). Diese Boxes und Confidences könnten evtl. später auch dazu verwendet werden, die detektieren und gescorten Gelenke auf dem Bild grafisch hervorzuheben (siehe 7.3 *Weitere Ideen*, Seite 99).

Gelenk hat die Score 0. Ein Gelenk mit der Score 1 weist eine Gewebeerosion von maximal 20% auf. Jede höhere Score deckt einen weiteren Bereich von 20% ab, sodass eine Score von 5 einer Erosion von 80% bis 100% gleichkommt (Rohrbach, 2017, S. 10).[7]

Neben der Ratingen-Score gibt es noch eine weitere Score: die Rau-Score. Beschreibt die Ratingen-Score die Schädigung eines bestimmten Gelenkes, geht es bei der Rau-Score um eine umfassendere Einschätzung: Neben den zehn MCP- und PIP-Gelenken beider Hände werden auch die Karpalgelenke beider Hände sowie fünf Gelenke beider Füsse berücksichtigt. Janick Rohrbach hat das Berechnen dieser Score in seiner Bachelorarbeit beschrieben (Rohrbach, 2017, S. 10).

Als Input nimmt das Modell ein NumPy-Array bestehend aus einem oder mehreren RGB-Bildern mit der Auflösung 150 mal 150 Pixel entgegen. Technisch gesprochen sind dies NumPy-Arrays mit der Dimension `(n, 150, 150, 3)`, wobei `n` für die Anzahl Bilder steht.

Der Output ist ein NumPy-Array, bei dem jeder Eintrag die Wahrscheinlichkeit für eine bestimmte Score eines Gelenkbildes repräsentiert. So wäre beispielsweise `[0.68 0.21 0.08 0.02 0.01 0.00]` ein Eintrag mit der Score 0 (höchste Wahrscheinlichkeit), ein Output wie `[0. 0.12 0.73 0.15 0. 0.]` stünde für einen Eintrag mit der Score 2.

Das Modell wurde mit einer neueren Version von TensorFlow (1.4.0) erstellt. Weitere zur Ausführung benötigte Libraries sind `pandas`, `Pillow`, `h5py`, `scikit-learn` und `keras-utils`. Abgespeichert ist es im `.h5`-Format. Es basiert auf dem Inception-V3-Modell von François Chollet.[8]

Im Gegensatz zu den bisher beschriebenen Modellen musste das Scoring-Modell neu trainiert werden, da die Modelldaten weder bei Seantis noch auf dem GitHub-Repository[9] der Bachelorarbeit von Janick Rohrbach vorhanden waren. Da auf dem GPU-Server von Seantis (`perses.seantis.ch`) eine ältere Python-Version (3.5) im Einsatz ist, wurde diese Version auch für den Container verwendet, in dem das Modell ausführbar ist. Da das Scoring-Modell aus technischen Gründen[10] nicht auf dem Systsem von Seantis zu Ende

[7] Gelenke mit einer derart starken Schädigung machten einen extrem geringen Teil des Trainingssets aus. Das Problem ist nicht etwa, dass es keine solchen Fälle mit beinahe komplett erodiertem Gelenkgewebe gebe, sondern weil betreffende Patienten kaum ihre Hand flach ausbreiten können, sodass derart erodierte Gelenke auf den Röntgenbildern praktisch nicht auszumachen sind.

[8] `https://github.com/fchollet/deep-learning-models` (abgerufen am 01.05.2020)

[9] `https://github.com/janickrohrbach/arthritis-net` (abgerufen am 01.05.2020)

[10] CUDA war nach dem Aktualisieren des Grafikkartentreibers nicht mehr funktionstüchtig und erforderte einen Neustart des Servers. Aufgrund einer darauf eingerichteten Volume-Verschlüsselung kann ein solcher Neustart nur vor Ort durchgeführt werden, was aufgrund von verordnetem Home-Office (Covid-19-Krise) nicht sofort möglich war. Nach einem Neustart vor Ort funktionierte CUDA wieder einwandfrei.

trainiert werden konnte, wurde eine GPU-Instanz bei Exoscale verwendet, auf welcher die Linux-Distribution Debian Stretch installiert ist. Aus diesem Grund dient `python:3.5-stretch` als Base-Image für die Ausführung des Scoring-Modells.

2.2 Webservices

Der zu erstellende Prototyp soll als Webservice umgesetzt werden (siehe Kapitel 1 *Problemstellung*, Seite 8). Für den Begriff *Webservice* gibt es verschiedene Definitionen, wovon hier drei exemplarisch erwähnt sind – in der Reihenfolge, in der sie bei der Google-Suche nach «Definition Web Service» beim Autor erschienen sind:

1. «*A web service is an application or data source that is accessible via a standard web protocol (HTTP or HTTPS). Unlike web applications, web services are designed to communicate with other programs, rather than directly with users.*[11]» (TechTerms, 2017)

2. «*A Web service is a software service used to communicate between two devices on a network. More specifically, a Web service is a software application with a standardized way of providing interoperability between disparate applications. It does so over HTTP using technologies such as XML, SOAP, WSDL, and UDDI.*[12]» (Techopedia, 2017)

3. «*Il s'agit d'une technologie permettant à des applications de dialoguer à distance via Internet, et ceci indépendamment des plates-formes et des langages sur lesquelles elles reposent. Pour ce faire, les services Web s'appuient sur un ensemble de protocoles Internet très répandus (XML, HTTP), afin de communiquer. Cette communication est basée sur le principe de demandes et réponses, effectuées avec des messages XML.*[13]» (Institut d'électronique et d'informatique Gaspard-Monge (IGM), o. J.)

[11] Ein Webservice ist eine Applikation oder Datenquelle, die über Standard-Web-Protokolle (HTTP oder HTTPS) erreichbar ist. Im Gegensaz zu Web-Anwendungen sind Webservices dafür ausgelegt mit anderen Programmen zu kommunizieren, nicht direkt mit den Benutzern. (Übersetzung des Autors)

[12] Ein Webservice ist ein Software-Service, der verwendet wird um zwischen zwei Geräten in einem Netzwerk zu kommunizieren. Genauer gesagt ist ein Webservice eine Software-Anwendung mit einer standardisierten Art um Interoperabilität zwischen verschiedenartigen Anwendungen anzubieten. Dies wird über HTTP mithilfe von Technologien wie XML, SOAP, WSDL und UDDI erreicht. (Übersetzung des Autors)

[13] Es handelt sich um eine Technologie, die es Anwendungen ermöglicht über das Internet zu kommunizieren, und zwar unabhängig von Plattformen und Sprachen, auf denen sie basieren. Um dies zu erreichen, stützen sich Webservices auf eine Menge weit verbreiteter Internet-Protokolle (XML, HTTP) für die Kommunikation. Diese Kommunikation basiert auf dem Anfrage-Antwort-Prinzip, was mithilfe von XML-Nachrichten bewerkstelligt wird. (Übersetzung des Autors)

Die Auflistung könnte beliebig weitergeführt werden. Allen Definitionen ist gemeinsam, dass HTTP(S) als Protokoll verwendet wird. Erwähnte Technologien wie XML und SOAP sind dabei schon länger auf dem Rückzug und werden zusehends von REST und JSON verdrängt (Mason, 2011).

Für den zu erstellenden Protoyp soll somit eine RESTful-API (via HTTP) angeboten werden. Genauer soll der Protoyp *gegen aussen* eine RESTful-API anbieten. Ob REST auch für die interne Kommunikation zwischen den Komponenten (siehe 2.1 *Bestehende Modelle*, Seite 14) eine gute Wahl ist, soll in der Architekturdiskussion im Folgekapitel (siehe 3.1 *Architekturvarianten*, Seite 31) untersucht werden.

2.3 Integrationsvarianten

Die gängigsten Methoden zum Integrieren von Anwendungen sind *File Transfer*, *Shared Database*, *Remote Procedure Invocation* und *Messaging* (Hohpe, Woolf, Brown & Fowler, 2004, Introduction, S. xxx) – und, ergänzend wie gerade besprochen, RESTful-APIs.[14] Aufgrund des synchronen Modus können REST und HTTP in diesem Kontext am ehesten mit Remote Procedure Invocation (RPC) verglichen werden, wobei HTTP an Ressourcen und RPC an Funktionen orientiert ist.

Die genannten Arten der Integration bauten aufeinander auf und stellten jeweils eine Verbesserung gegenüber ihrem Vorgänger dar, was jedoch jeweils eine erhöhte Komplexität zur Folge habe (Hohpe et al., 2004, S. 41-42). Die einfachste Variante sei der Verzicht auf eine Integration von Anwendungen, indem man sich auf eine einzelne, zusammenhängende (d.h. monolithische) Applikation beschränkt, die gar keine Integration erfordert (Hohpe et al., 2004, S. 39). Dies ist oftmals nicht möglich oder nicht sinnvoll, wie z.B. im vorliegenden Projekt, bei dem zueinander inkompatible Laufzeitumgebungen zum Einsatz kommen, die sich nicht in eine monolithische Applikation zwängen lassen.

Die Vor- und Nachteile der genannten Integrationsvarianten lassen sich folgendermassen zusammenfassen:

File Transfer Die Dateiübertragung sei die wohl einfachste Integrationsvariante, denn praktisch alle Betriebssysteme unterstützten Dateien und deren Austausch mit anderen Systemen oder Prozessen. Diese Einfachheit bezahle der Entwickler jedoch

[14] Der Begriff REST und das Konzept der RESTful-APIs sind zwar spätestens seit dem Jahr 2000 bekannt (R. T. Fielding & Taylor, 2000, Kapitel 6). Weite Verbreitung fand das Konzept jedoch erst später, gerade mit dem Siegeszug von *Microservices* – ein Begriff, der erst seit dem Jahr 2011 geläufig ist (Mauersberger, 2019). RESTful-APIs waren beim Erscheinen des erwähten Standardwerks im Jahr 2004 zwar schon bekannt, aber nicht weit verbreitet.

damit, dass er sich um viele Aspekte der Integration selber kümmern muss: Konventionen für Dateinamen und Verzeichnisse und das Sicherstellen deren Eindeutigkeit; das Löschen alter Dateien und das Feststellen für dessen Notwendigkeit; Sperrmechanismen zum Sicherstellen, dass nicht zwei Anwendungen gleichzeitig auf eine Datei zugreifen; Übertragung der Dateien über ein geeignetes Protokoll auf ein passendes Medium. Ein Hauptproblem stellten dabei die verschiedenen Bedürfnisse der beteiligten Anwendungen für die Synchronisierung der Dateien dar: Manche Applikationen benötigten Aktualisierungen in Sekundenschnelle, während für andere eine nächtliche, wöchentliche oder monatliche Aktualisierung ausreiche. Solche Unterschiede könnten oft zu Inkonsistenzen führen, die sich desto schwerer beheben liessen, je weiter ihr Auftreten von ihrer Kenntnissnahme entfernt sei. (Hohpe et al., 2004, S. 43-46)

Shared Database Gegenüber der Dateiübertragung biete eine gemeinsame Datenbank einige Vorteile. Zeitliche Lücken träten kaum mehr auf, denn jede Applikation habe stets Zugriff auf den aktuellen Stand. Mithilfe einer Datenbank sei es zudem einfacher, ein bestimmtes Datenformat sicherzustellen. Dank des Transaktionssystems von Datenbanksystemen seien die Daten stets konsistent. Zwar müsse man viel Aufwand betreiben, um bei den verschiedenen Applikationen mit gleicher Datenbank ein Schema auszuarbeiten, mit dem alle Applikationen umgehen können; es sei jedoch besser, diese Probleme direkt anzugehen, statt ihnen auszuweichen, um sich in der anschliessenden Integrationsphase wieder davon einholen zu lassen. Alle beteiligten Applikationen auf einen gemeinsamen Nenner zu bringen sei jedoch manchmal so schwierig, dass das Ergebnis dieser Übung unbefriedigend ausfalle, und in einem Datenschema resultiere, mit dem nur schwer umzugehen ist. Solche Konflikte könnten dazu führen, dass sich Trennungen aufdrängten.[15] Der Zugriff von mehreren Applikationen auf eine gemeinsame Datenbank mit dem damit einhergehenden wiederholten Lesen und Schreiben der gleichen Daten könne dazu führen, dass die Datenbank zum Flaschenhals werde, oder dabei gar Deadlocks aufträten. (Hohpe et al., 2004, S. 47-49)

Remote Procedure Invocation Die beiden genannten Ansätze *File Transfer* und *Shared Database* haben gemeinsam, dass sie Daten verschiedener Applikationen integrieren, nicht aber deren Funktionalität. (Hohpe et al., 2004, S. 49) Einer der mächtigsten

[15] Siehe auch Conway's Law: *«Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure.»* (Conway, 1968)

Mechanismen zur Strukturierung von Software sei die Kapselung mittels Funktionen. Mittels *Remote Prodecure Invocation* (RPI) oder *Remote Procedure Call* (RPC) könnten auch zwischen Applikationen Funktionen aufgerufen werden. Daten werden dabei als Funktionsparameter übertragen. Die internen Datenstrukturen der einzelnen Applikationen können dabei frei gestaltet und zu einem späteren Zeitpunkt verändert werden, ohne dass eine andere Applikation davon betroffen sei. Weiter könnten Anwendungen mittels RPI mehrere Schnittstellen auf die gleichen Daten anbieten.[16] Der Funktionsaufruf sei ein für alle Programmierer geläufiges Modell, was einerseits ein Vorteil ist, andererseits problematisch sein kann, zumal sich die Performance zwischen einem lokalen und einem entfernten Funktionsaufruf um mehrere Grössenordnungen unterscheiden kann. Ausserdem verleitet RPI dazu, eine verteilte Applikation wie eine einzelne, zusammenhängende zu gestalten, was zu einer engen Kopplung führe, sich aber angesichts der Performanceeinbussen nicht so anfühle. (Hohpe et al., 2004, S. 50-52)

Messaging Die grösste Herausforderung bei der Integration von Anwendungen bestehe oft darin, den Zeitverlust beim Austausch zwischen Systemen so gering wie möglich zu halten, ohne die Anwendungen dabei eng aneinander zu koppeln, oder dabei die Weiterentwicklung oder das Laufzeitverhalten der Anwendungen zu beeinträchtigen. Hierzu benötige es einen ähnlichen Mechanismus wie *File Transfer*, womit Datenpakete schnell produziert und einfach transferiert werden können – und darüber hinaus den Empfänger informiere, dass ein Datenpaket zur Verarbeitung anstehe, sowie einen Retry-Mechanismus für gescheiterte Übertragungen biete. Im Gegensatz zur *Shared Database* soll das interne Datenschema einer Anwendung veränderbar bleiben. Im Gegensatz zur *Remote Procedure Invocation* soll die Übertragung asynchron ablaufen, sodass der Aufrufer nicht blockierend auf die Antwort warten muss. Weiter sollen auch Nachrichten verschickt werden können, ohne dass die Gegenseite zu dieser Zeit bereit sein muss. *Messaging* erfülle alle diese Anforderungen, und biete darüber hinaus noch verschiedene Verteilungsmechanismen wie Broadcasting zu allen oder Routing zu verschiedenen Empfängern, oder verschiedene Topologien. Da Nachrichten zeitnah verarbeitet werden können, fallen die Synchronisationsprobleme von *File Transfer* weg. Diese Vorteile erkaufe sich der Entwickler mit einer steilen Lernkurve und einer Reihe neuer Herausforderungen, von denen der Rest des Buches grösstenteils handelt. (Hohpe et al., 2004, S. 53-59)

[16] Dies ist eine Ähnlichkeit zu HTTP, das verschiedene Methoden (`GET`, `POST` usw.) bietet, um auf die gleichen Ressourcen zuzugreifen.

Messaging ist somit die mächtigste, aber auch komplizierteste Variante. Wie sich die beschriebenen Machine-Learning Modelle (siehe 2.1 *Bestehende Modelle*, Seite 14) integrieren lassen, ist Thema des nächsten Kapitels (siehe 3.1 *Architekturvarianten*, Seite 31).

2.4 Evaluation von Modellen

Der Projektauftrag erwähnt den *Vorschlag für geeignete Metriken zu Vergleich von Modellen* als erwartetes Resultat (siehe 1.1.2 *Erwartetes Resultat*, Seite 11). Hierbei soll es nicht um die Evaluation der einzelnen verwendeten Modelle gehen, sondern um eine Evaluation des Gesamtsystems, das zu diesem Zweck wie ein grosses, zusammenhängendes Modell betrachtet wird. Zu einem späteren Zeitpunkt, an dem einzelne, verbesserte Modelle in das System einfliessen, können verschiedene Versionen des Gesamtsystems im Bezug auf ihre Performance miteinander verglichen werden.

2.4.1 Verschiedene Datentypen

Um eine passende Art der Evaluation zu finden, muss man sich zuerst vergegenwärtigen, mit welcher Art von Prediction (Output) man es zu tun hat. Hierbei fällt auf, dass in Wissenschaft und Praxis von verschiedenen Arten von Daten die Rede ist.

Einen guten Überblick über die Thematik bietet der Artikel *7 Data Types: A Better Way to Think about Data Types for Machine Learning* (Hale, 2018). So werde im Machine-Learning-Bereich teilweise nur zwischen numerischen und kategorischen Daten unterschieden. Diese Unterscheidung greife aber zu kurz und lasse sich weiter verfeinern und erweitern. Die elf [sic.[17]] Datentypen lassen sich folgendermassen zusammenfassen:

Useless Daten, oft zufällige und eindeutige wie z.B. eine Kontonummer, die zwar als Identifikation von Datensätzen, aber nicht zum Trainieren eines Modells verwendet werden können.

Nominal Diskrete Werte ohne numerische Beziehung zwischen den Werten, wie z.B. Arten von Tieren, auf die sich keine statistischen Operationen wie mean und median anwenden lassen.

Ordinal Diskrete Werte, die rangiert und sortiert werden können, wobei der Abstand zwischen zwei Zahlen unbekannt ist.

[17] Die ursprünglichen sieben Datentypen wurden in einer späteren Version des Artikels um vier weitere ergänzt.

Binary Diskrete Werte, welche in die beiden Kategorien null und eins eingeteilt werden können, und so einen Spezialfall von Nominal-, Ordinal- und/oder Intervalldaten darstellen.

Count Diskrete, positive Zahlen, die eine Anzahl wiedergeben, und auf die statistische Operationen wie mean und median sinnvollerweise angewendet werden können.

Time Zyklische, sich wiederholende, kontinuierliche Daten, die sich auf verschiedene Perioden (Jahr, Monat, Tag usw.) beziehen können.

Interval Daten, bei denen benachbarte Zahlen die gleichen Abstände voneinander haben, aber kein zeitliches Muster repräsentieren.

Image Zweidimensionale Bilder, wie z.B. Röntgenbilder.

Die drei weiteren vorgeschlagenen Kategorien, die für die vorliegende Arbeit nicht releant sind, lauten: Video, Audio und Text.

2.4.2 Datentypen der Modelle

Bei den verschiedenen Modellen kommen unterschiedliche Arten von Daten zum Einsatz. Die Einteilung nach der gerade aufgestellten Taxonomie ist dabei nicht immer klar und eindeutig:

`body_part` Für ein Röntgenbild wird eine Reihe von Wahrscheinlichkeiten erstellt, die beschreiben, welche Art von Körperteil es abbildet. Hierbei handelt es sich um Intervalldaten, denn eine Wahrscheinlichkeit von 0.8 kann als doppelt so wahrscheinlich wie eine Wahrscheinlichkeit von 0.4 interpretiert werden. Für die vorliegende Arbeit ist jedoch nur ein Körperteil (die linke Hand) und somit nur deren Wahrscheinlichkeit relevant (siehe 2.1.1 *Erkennung von Körperteilen:* `body_part`, Seite 15), welche mithilfe eines passenden Schwellenwertes zu einer Binärklassifikation regrediert. Ob die Interpretation der Prediction anhand dieses Schwellenwerts Teil der Modellkomponente oder zu den Aufgaben des Aufrufers gehört, oder gar weggelassen werden kann, ist eine Architekturfrage.

`joint_detection` Aus einem Röntgenbild einer linken Hand werden bis zu zehn Gelenke extrahiert. Sowohl Input wie Output sind Bilddaten. Für die Evaluation des Modells sind Bilddaten jedoch ungeeignet, da die Definition der Testdaten und der Vergleich mit den Predictions aufwändig (Vergleich von 2D-Matritzen) und fehleranfällig (Verschiebung um einzelne Pixel) ist. Für die Evaluation ist es sinnvoller, die

Frage zu stellen, ob ein betreffendes Gelenk (PIP 1, MCP 3, usw.) gefunden worden ist oder nicht, was mit einer Binärklassifikation pro Gelenk bewerkstelligt werden kann. Da pro Bild zehn Gelenke gefunden oder nicht gefunden werden können, ist eine Zählung (Count) im Bereich null bis zehn der geeignete Datentyp, sofern die zehn Extraktionsvorgänge konzeptuell als ein übergeordneter Vorgang betrachtet werden sollen.

`ratingen_score` Bei der Ratingen-Score handelt es sich um Intervalldaten von null bis hundert Prozent. In der SCQM-Datenbank werden die einzelnen Gelenkscores als Anteil des geschädigten Gelenkgewebes mit diskreten Prozentzahlen und einer Auflösung von 5% abgespeichert. Das entsprechende Modell teilt diese Score jedoch in sechs Kategorien ein, wobei die Kategorien 1 bis 5 jeweils für eine Bandbreite von 20% stehen. Ein Spezielfall ist die Kategorie 0, die für ein gesundes Gelenk mit 0% Erosion steht. Die Prediction hat damit sowohl Aspekte einer nominalen Kategorisierung als auch von Intervalldaten. Durch die Vermischung der Nominalzahl (Kategorie 0) mit Intervalldaten (Kategorien 1-5) liegen unterschiedlich grosse Abstände zwischen den Kategorien vor, wodurch die Kategorien als Ordinalzahlen betrachtet werden können. Für die Evaluation des Gesamtsystems ist weiter zu beachten, dass pro Vorgang nicht nur eines, sondern (bis zu) zehn Gelenke gescored werden. Die Anzahl der Übereinstimmungen von Prediction und tatsächlichem Wert könnten gezählt werden, also z.B. acht von zehn Gelenken wurden korrekt gescored. Besser ist eine Prozentangabe (Intervalldaten), welche die Anzahl der korrekt gescorten Gelenke der Anzahl vorhandener Gelenke gegenüberstellt, zumal diese Metrik auch bei fehlenden Gliedmassen oder unerkannten Gelenken eine sinnvolle Aussage macht.[18]

Betrachtet man die drei Modelle als Gesamtsystem, besteht der Output aus einer Reihe (null bis zehn) Scores im Intervall $[0..5]$. Zwar sind für die Performance des Gesamtsystems alle drei Modelle ausschlaggebend, der Evaluation stehen jedoch nur die Scores des letzten Modells zur Verfügung.

Für die Evaluation des Gesamtsystems kann pro Vorgang (Röntgenbild) eine Menge von Ordinalzahlen (Output, $[0..5]$) mit einer Reihe von Intervalldaten (Testdaten, $[0..100]$) verglichen werden (im Sinne von *Matching*), wobei auch die Grösse dieser Menge (Intervallzahl, $[0..10]$) aussagekräftig ist.

[18] Auf den Röntgenbildern sind statt Gelenke oftmals Metallgegenstände zu sehen, welche im fortgeschrittenen Krankheitsstadium eher als Ersatz denn als Reparatur eines Gelenks zu verstehen sind.

2.4.3 Mögliche Evaluationsmetriken

Für die ermittelten Datentypen gibt es verschiedene Evaluationsmetriken. Die ermittelten Scores für die Gelenke eines oder mehrerer Vorgänge könnten zusammengefasst und auf ihre Genauigkeit (Accuracy) geprüft werden (Zheng, 2015, S. 8):

$$\text{Global Accuracy} = \frac{\text{Total Correct Predictions}}{\text{Total Predictions}}$$

Formel 1: Global Accuracy

Eine solche Klassifikation behandle jedoch alle Klassen gleich, was gerade in medizinischen Anwendungen oft problematisch sein könne, zumal *false positives* (gesunden Patient als krank eingestuft) und *false negatives* (kranken Patient als gesund eingestuft) oft stark unterschiedliche Konsequenzen hätten.[19] Eine *Confusion Matrix* schaffe hier Abhilfe, und wäre für die binäre Klassifikation und somit für das Modell `body_part` eine sinnvolle Metrik.[20] (Zheng, 2015, S. 8)

Da bei der Evaluation des Gesamtsystems Scores von sechs verschiedenen Klassen relevant sind, und da die einzelnen Klassen mit stark unterschiedlicher Häufigkeit auftreten (Rohrbach, 2017, S. 16-17), ist die *Per-Class Accuracy* eine weitere mögliche Metrik. Für Klassen mit kleinen Datenbeständen könne die Varianz jedoch sehr hoch ausfallen, und die Evaluation darum wenig aussagekräftig werden (Zheng, 2015, S. 9):

$$\text{Per-Class Accuracy}_{\text{Class n}} = \frac{\text{Correct Predictions}_{\text{Class n}}}{\text{Total Predictions}_{\text{Class n}}}$$

Formel 2: Per-Class Accuracy

Die Kosten von *false positives* kommen mit der *Precision* zum Tragen. Die Precision ist definiert als (Shung, 2018):

$$\text{Precision} = \frac{\text{True Positive}}{\text{True Positive} + \text{False Positive}}$$

Formel 3: Precision (berücksichtigt die Kosten von *false positives*)

[19] Da der zu erstellende Prototyp nicht in der Diagnostik, sondern für die langfristige Erforschung einer Krankheit eingesetzt wird, ist das genannte Problem für die vorliegende Arbeit weniger relevant.

[20] Sofern die Interpretation eines Schwellenwerts Teil des Modells sein soll.

Möchte man die Kosten von *false negatives* abbilden, ist der *Recall* eine geeignete Metrik. Der Recall ist definiert als (Shung, 2018):

$$\text{Recall} = \frac{\text{True Positive}}{\text{True Positive} + \text{False Negative}}$$

Formel 4: Recall (berücksichtigt die Kosten von *false negatives*)

Die beiden Metriken Precision und Recall lassen sich mittels *F1 Score* zu einer einzigen Metrik kombinieren, die gegenüber der Accuracy den Vorteil hat, dass sie die «Kosten» von *false positives* und *false negatives* berücksichtigt (Shung, 2018):

$$\text{F1} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

Formel 5: F1 Score (bildet die Kosten von *false positives* und *false negatives* ab)

Log-Loss ist eine Metrik, die sich bei der Klassifikation mit Wahrscheinlichkeiten einsetzen lässt. Ihr Vorteil gegenüber der einfachen *Accuracy* bei der Binärklassifikation ist, dass sie *near misses* («knapp daneben») weniger stark bestraft als grobe Abweichungen der Predictions vom eigentlichen Wert (Zheng, 2015, S. 9-10). Diese Metrik liesse sich sinnvollerweise zur Evaluation des Modells `body_part` einsetzen (sofern dieses eine Wahrscheinlichkeit und keine interpretierte Binärklassifikation liefert), jedoch nicht für das Gesamtsystem.

Mithilfe einer *AUC*-Metrik (*Area under the curve*) liesse sich die Sensitivität eines Klassifikators grafisch aufzeigen, indem die Rate der *true positives* zu der Rate der *false positives* geplottet werde (*ROC*-Kurve, *receiver operating characteristic curve*). Dadurch liesse sich zeigen, wie viele korrekte Klassifikationen erzielt werden können, wenn man mehr *false positives* zulasse. *ROC*-Kurven lassen sich grafisch vergleichen. Mithilfe der berechneten *Area under the curve* lassen sich die verschiedenen Kurven auch einfach numerisch vergleichen. Diese Metrik ist v.a. für die Binärklassifikation geeignet. Dies gilt auch für *Precision Recall*, die nächste vorgeschlagene Metrik, die auch bei der Evaluation von Rankings hilfreich ist (Zheng, 2015, S. 10-14).

Für die Evaluation von Scores, die den Output des Gesamtsystems ausmachen, sind Regressionsmetriken sinnvoller. Diese unterscheiden nicht nur zwischen richtig und falsch, sondern gewichten Abweichungen anhand derer betraglichen Verfehlung, sodass grössere Abweichungen stärker bestraft werden als kleinere Abweichungen.

RMSE (*root-mean-square error*) bzw. *RMSD* (*root-mean-square deviation*) ist eine häufig verwendete Metrik, und ist definiert als die Quadratwurzel des Durchschnitts der quadrierten Distanzen zwischen dem tatsächlichen Wert y_i und der Prediction $\hat{y}_i$:

$$\text{RMSE} = \sqrt{\frac{\sum_i (y_i - \hat{y}_i)^2}{n}}$$

Formel 6: Root-Mean-Square Error (RMSE)/Root-Mean-Square Deviation (RMSD)

Da diese Metrik auf dem arithmetischen Mittel basiert, sei sie anfällig für sogenannte *Outliers* (Zheng, 2015, S. 15). Dies ist für das Scoring der Gelenke unproblematisch, zumal dieses in sechs Klassen erfolgt, und daher keine *Outliers* möglich sind. Es ist dennoch sinnvoll, eine Metrik beizuziehen, die auf dem Median basiert, um ein ausgewogeneres Gesamtbild zu erhalten. Die *median absolute percentage* (*MAPE*) wäre eine solche Metrik:

$$\text{MAPE} = \text{median}\left(\left|\frac{y_i - \hat{y}_i}{y_i}\right|\right)$$

Formel 7: Median Absolute Percentage (MAPE)

Diese Metrik gebe ein relatives Mass des typischen Fehlers an (Zheng, 2015, S. 15). Eine der einfachsten Metrik sei es, die relative Abweichung der Predictions vom eigentlichen Wert in Prozent anzugeben. Die akzeptable Abweichung in Prozent sei dabei von der Natur des jeweiligen Problems abhängig (Zheng, 2015, S. 15).

Beim Scoring von Gelenken ist es möglich, dass eine Einschätzung (z.B. 80% des Gelenkgewebes erodiert) gerade auf die Schwelle zwischen zwei Klassen (4 und 5) fällt. Da es sich um menschliche Einschätzungen, und nicht um präzise, maschinelle Messungen handelt, sei eine Abweichung um eine Klasse nicht als Fehler zu werten (Rohrbach et al., 2019, Abschnitt 3.3, S. 476 ff.). Bei insgesamt sechs Klassen können Predictions als korrekt gewertet werden, die innerhalb eines Fehlers von $16\frac{2}{3}$% liegen (Zheng, 2015, S. 16):

$$\left(\left|\frac{y_i - \hat{y}_i}{y_i}\right|\right) < 0.1\overline{6}$$

Formel 8: Almost Correct

Die genannten Metriken gewichten alle Klassen gleich. Dies sei problematisch, wenn

es ein Ungleichgewicht zwischen den einzelnen Klassen gäbe, weil die Metrik dann von der Klasse mit den meisten Datenpunkten dominiert werde (Zheng, 2015, S. 17), was im vorliegenden Kontext der Fall ist. Eine Kombination verschiedener Metriken sei darum angebracht, um ein möglichst genaues Bild von der Performance zu erhalten (Rohrbach et al., 2019, Abschnitt 3.3, S. 476 ff.).

In der genannten Quelle werden weitere problemspezifische Evaluationsmetriken besprochen, welche eine Prediction mit einer zuvor ermittelten Score vergleichen. Diese Metriken hätten allesamt kein befriedigendes Ergebnis geliefert. Stelle man jedoch mithilfe von *Cohen's Kappa* die Übereinstimmung des Modells mit den durch Menschen ermittelten Scores gegenüber, erhalte man dadurch eine aussagekräftige Metrik, welche zudem das Problem der Unausgewogenheit angemessen behandle (Rohrbach et al., 2019, Abschnitt 3.3.3, S. 477):

$$\kappa = \frac{p_{\text{obs}} - p_{\text{chance}}}{1 - p_{\text{chance}}} = 1 - \frac{q_{\text{obs}}}{q_{\text{chance}}}$$

Formel 9: Cohen's Kappa

Dabei steht p_{obs} für die beobachtete Übereinstimmung und p_{chance} für die Übereistimmung, die von einem Zufallsprozess zu erwarten ist. Diese Zahlen erhalte man von der Confusion Matrix, indem man die Zeilen- mit den Spaltenrändern multipliziere. *Cohen's Kappa* sei gerade dann von Vorteil, wenn die Performance der einzelnen Scorer fraglich ist (McHugh, 2012).

Da die zufällige Übereistimmung von der abgeschätzen Häfugkeit der Klasse abhängig ist, hängt *Cohen's Kappa* auch von der Unausgeglichenheit der Klassen ab.

Um zusätzlich noch grössere Abweichungen stärker zu bestrafen als kleinere, kann *Cohen's Quadratic Kappa* verwendet werden (Rohrbach et al., 2019, ebd.):

$$\kappa_{\text{squared}} = 1 - \frac{\sum_{i=1}^{k} \sum_{j=1}^{k} \left((i-j)^2 \cdot n_{\text{obs}_{ij}} \right)}{\sum_{i=1}^{k} \sum_{j=1}^{k} \left((i-j)^2 \cdot n_{\text{chance}_{ij}} \right)}$$

Formel 10: Cohen's Quadratic Kappa

Dadurch steige die Bestrafung für Fehler quadratisch mit dem Betrag der Verfehlung. Dabei sind n_{ij} Elemente in der Confusion Matrix, die *nicht* auf der Diagonalen liegen, d.h. die Fehler.

Mithilfe der *Interclass Correlation* könne man das feststellen, wie gross die Übereinstimmung zwischen verschiedenen Scorern ist. Da es sich bei der Ratingen-Score um eine menschliche Einschätzung und nicht um eine mit mathematisch reproduzierbarer Exaktheit ermittelte Grösse handelt, ist die Übereinstimmung zwischen der Modell-Vorhersage mit den menschlichen Scorern einerseits mit der Übereinstimmung der menschlichen Scorern untereinander andererseits ein gutes Mass für die Performance des Modells (Rohrbach et al., 2019, ebd.). Diese Art der Evaluierung benötige jedoch mehrere unabhängige menschlich ermittelte Scores, welche im Rahmen dieses Projekts nicht zur Verfügung stehen (siehe 6.1.1 *Evaluationsdaten*, Seite 81).

2.5 Modellformate

Für das Abspeichern von (trainierten) Machine-Learning-Modellen gibt es eine Vielzahl an Formaten. Im vorliegenden Projekt werden beispielsweise das `tflearn`-Format (für die Modelle `body_part` und `joint_detection`) und das `h5`-Format (für das Modell `ratingen_score`) verwendet.

Für neuere Versionen von TensorFlow hat sich das *SavedModel*-Format etabliert (Géron, 2019, Kapitel 19). Die Modelle lassen sich dabei nicht ohne weiteres von einem Format in ein anderes übertragen, zumal Modelle, die in älteren Versionen von TensorFlow erstellt worden sind, sich mit neueren Versionen nicht laden lassen.[21] Somit ist die Wahl des Modellformats eine wichtige Fragestellung, gerade im Hinblick auf Vorwärtskompatibilität für künftige Releases von TensorFlow, Keras oder anderer Frameworks und APIs.

Anstrengungen zur Vereinheitlichung von Modellformaten werden im Rahmen der ONNX-Initiative unternommen (*ONNX*, 2019). ONNX ist ein neuer, offener Standard für die Interoperabilität von Machine Learning. Mithilfe von `tf2onnx`[22] können bestehende TensorFlow-Modelle zu ONNX konvertiert werden. Die ONNX-Initiative wird u.a. von Microsoft, Amazon, Facebook, IBM und Intel getragen – sprich *«alle ausser Google»* (Gaskill, 2018). Ende 2019 ist das ONNX-Projekt zudem unter den Schirm der Linux Foundation gekommen (Parbel, 2019). Das Projekt ist auf GitHub jedoch weiterhin auf dem Microsoft-Account gehostet.[23] Seit Herbst 2019 liegt eine Runtime für ONNX von Microsoft in Version 1.0 vor (Xu, 2019). Mit dem *ONNX Model Zoo* werden Entwicklern fertige Modelle angeboten, auf denen diese ihre Projekte entwickeln können.[24]

[21] Zahlreiche Versuche sind für das Modell `body_part` unternommen worden – und gescheitert.
[22] `https://github.com/onnx/tensorflow-onnx` (abgerufen am 01.05.2020)
[23] `https://github.com/microsoft/onnxruntime` (abgerufen am 01.05.2020)
[24] `https://github.com/onnx/models` (abgerufen am 01.05.2020)

Ob sich ONNX als offener Standard durchsetzen – und sich gegen das SavedModel-Format von TensorFlow (bzw. Google) behaupten kann, ist Stand Frühling 2020 schwer abzuschätzen. Die Unterstützung von ONNX auf Azure, und die Tatsache, dass Google ONNX nicht unterstützt, dürfte eher als Kampfansage von Microsoft an Google verstanden werden, denn als Schritt hin zur Vereinheitlichung von Modellformaten. ONNX wird zwar mit der MIT-Lizenz als OpenSource-Projekt auf GitHub gehostet.[25] Bei der Hauptprogrammiersprache des Projekts (PureBasic[26]), handelt es sich jedoch um eine kommerzielle Programmiersprache. Diese ist plattformunabhängig (Windows, Linux, macOS) und *«begrenzt auf kleinere Programme»* frei verfügbar (*PureBasic Download*, 2020). Für OpenSource-Entwickler dürfte diese Einschränkung ein Hindernis darstellen, zumal das ONNX-Projekt mit seinem Umfang kaum unter die Kategorie *«kleinere Programme»* fallen dürfte.

Im Rahmen des vorliegenden Projekts sind die beiden Kandidaten – SavedModel und ONNX – von geringer Relevanz, da die Modelle in ihrem aktuellen Zustand übernommen werden sollen. Für die Weiterentwicklung der beschriebenen Modelle jenseits der Bachelorarbeit sollte die Entwicklung jedoch im Auge behalten werden (siehe auch 7.2.1 *Aktualisierung der Modelle*, Seite 96).

[25] `https://github.com/onnx/onnx` (abgerufen am 01.05.2020)
[26] 42.7% Anteil des Codes, Stand 29. April 2020

3 Ideen und Konzepte

In diesem Kapitel wird aufgezeigt, wie sich die im vorherigen Kapitel erarbeiteten technischen Grundlagen zu einem Design für einen Prototyp zusammenfügen lassen. Hierzu sollen verschiedene Architekturvarianten aufgezeigt und miteinander verglichen werden, von der dann die geeignetste für die Umsetzung vorgeschlagen wird.

3.1 Architekturvarianten

Soll für die Schnittstelle nach aussen HTTP verwendet werden (siehe 2.2 *Webservices*, Seite 18), gibt es für die interne Kommunikation zwischen den Komponenten keine expliziten Einschränkungen. Die äussere Schnittstelle gibt jedoch einen Rahmen vor.

Ein wichtiges Merkmal von HTTP ist, dass Anfragen atomar sind, d.h. nur als Ganzes beantwortet werden können: Zu jedem Request gibt es genau eine Response, und jede Response hat genau einen Status-Code, der über den Erfolg bzw. Misserfolg der Anfrage Auskunft gibt.[27]

Da HTTP ein synchrones Protokoll ist, sollten Anfragen schnell beantwortet werden, d.h. eher in Sekunden (besser Millisekunden) als in Minuten. Dauert eine Anfrage zu lange, wird diese von einem «ungeduldigen» Client abgebrochen. Zwar kann diese Zeitspanne mittels Keep-Alive-Header ausgedehnt werden (R. Fielding & Reschke, 2014, Anhang 1.2). Dies stellt jedoch Anforderungen an den Client, wessen Konfiguration sich der Hoheit des zu erstellenden Prototyps entzieht.

Eine Batch-Verarbeitung, bei der mehrere Bilder mit einer Anfrage zum Scoring in Auftrag gegeben werden, ist aus diesen beiden Grüden – Atomizität und synchrone Kommunikation – nicht sinnvoll, und soll deshalb nicht angeboten werden. Die RESTful-API soll darauf ausgelegt werden, nur ein Bild pro Request zu verarbeiten. Weiter soll der Prototyp den Anspruch haben, dass einzelne Bilder trotz der vielschichtigen Verarbeitung in einer für einen synchronen Client nützlichen Frist verarbeitet werden.

Die äussere Schnittstelle schliesst somit eine Batch-Verarbeitung aus. Der Protoyp erhält jeweils ein Bild pro Anfrage und soll damit mit einem möglichst vollständigen Scoring antworten, d.h. mit zehn Scores für die einzelnen Gelenke MCP 1-5 und PIP 1-5, sofern es sich beim Bild um eine Röntgenaufnahme einer linken Hand handelt.

[27] Auch wenn ein Status-Code wie `206 Partial Content` auf die Unvollständigkeit der Antwort hinweist, muss für den Zugriff auf die weiteren Antwortteile wieder ein neuer Request abgesetzt werden (R. Fielding, Lafon & Reschke, 2014, Kapitel 4.1).

Kann das Bild nicht oder nur teilweise verarbeitet werden – sei es aufgrund eines falschen Bildes (keine Röntgenaufnahme oder Röntgenaufnahme eines anderen Körperteils), aufgrund einer schlechten Bildqualität der Röntgenaufnahme, oder aufgrund einer mangelhaftgen Prediction-Performance der involvierten Modelle – sollen zumindest alle ermittelten Scores oder entsprechende Fehlermeldungen zurückgegeben werden.[28]

Auf Basis der vorgängig diskutierten Integrationsvarianten (siehe 2.3 *Integrationsvarianten*, Seite 19) werden im Folgenden verschiedene Architekturen für die Umsetzung des Prototyps vorgestellt. Dabei wird jeweils von einem Client ausgegangen, der synchron per HTTP mit dem zu erstellenden System (*DeepXRay*) kommuniziert.

3.1.1 Variante 1: HTTP, synchron

Die erste Variante (*HTTP, synchron*) verwendet vier verschiedene Komponenten. Dies sind einerseits Komponenten basierend auf den bereits beschriebenen Machine-Learning-Modellen `body_part`, `joint_detection` und `ratingen_score`.[29] Andererseits kommt der sogenannte `orchestrator` hinzu, der für die Kommunikation mit dem Client verantwortlich ist, und die Arbeit der Modellkomponenten koordiniert.

Die Komponentenarchitektur dieser Variante ist auf Abbildung 2, Seite 33 (Komponentendiagramm) ersichtlich. Die drei Modellkomponenten bieten jeweils eine interne HTTP-Schnittstelle an, die vom `orchestrator` angesprochen werden kann. Zwischen den Modellkomponenten findet keine Kommunikation statt.

Das Sequenzdiagramm Abbildung 3, Seite 34 veranschaulicht den Ablauf. Zunächst stellt der `client` eine HTTP-`POST`-Anfrage an den `orchestrator`, welche ein Röntgenbild (`xray`) enthält. Der `orchestrator` leitet das Bild weiter zu `body_part`. Diese Komponente antwortet mit einer Wahrscheinlichkeit, mit der es sich beim dargestellten Körperteil um eine linke Hand handelt.

Fällt diese Wahrscheinlichkeit genügend hoch aus, um eine Weiterverarbeitung zu rechtfertigen, wird das Röntgenbild weiter an die Komponente `joint_detection` gesendet.[30]

[28] Die Wahl der geeigneten HTTP-Status-Codes ist Teil der Implementierung und soll an dieser Stelle nicht weiter besprochen werden.

[29] Diese Bezeichnungen werden im Folgenden sowohl für die eigentlichen Modelle als auch für die *Modellkomponenten* verwendet, die das jeweilige Modell beinhalten, dieses aber um Integrationscode erweitern. Ob das Modell oder die Modellkomponente gemeint ist, ergibt sich dabei aus dem Kontext.

[30] Auf Negativfälle wird in diesem Kapitel nur eingegangen, wo sie für die Konzeption der Architektur ausschlaggebend sind. Ansonsten wird zugunsten der Übersichtlichkeit auf deren Besprechung verzichtet. Negativfälle werden im Kapitel 5 *Realisierung*, Seite 60 behandelt.

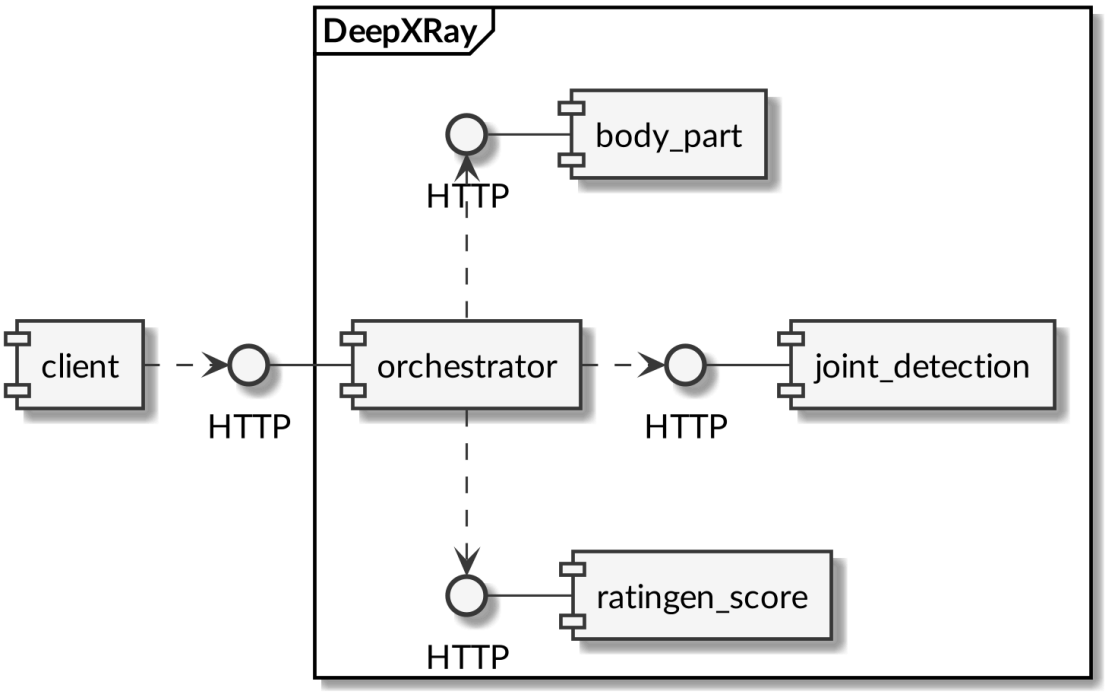


Abbildung 2: Komponentenarchitektur der Varianten 1 und 2 (HTTP, synchron und synchron/asynchron). Die Modellkomponenten bieten eine HTTP-Schnittstelle an, die vom `orchestrator` angesprochen wird. (Komponentendiagramm)

Dort werden die zehn relevanten Gelenke extrahiert und als Bildausschnitte in einem geeigneten Format[31] an den `orchestrator` zurückgeliefert.

Der `orchestrator` sendet diese extrahierten Bildausschnitte der Reihe nach an die Komponente `ratingen_score`, welche das Scoring der dargestellten Gelenke vornimmt. Die Score wird wiederum in einem geeigneten Format an den `orchestrator` zurückgesendet.

Am Schluss sammelt der `orchestrator` diese Scores, fasst sie in einer Datenstruktur zusammen, und sendet sie dem Client in einem geeigneten Format (JSON) zurück.

Diese Variante hat folgende Vor- und Nachteile:

Vorteile Die Variante ist sehr einfach umzusetzen. Da HTTP als externe Schnittstelle bereits gesetzt ist, können die internen Schnittstellen den gleichen Ansatz verwenden, wodurch nur eine Art von Schnittstelle benötigt wird.

[31] z.B. base64-codiert in einem JSON-Payload

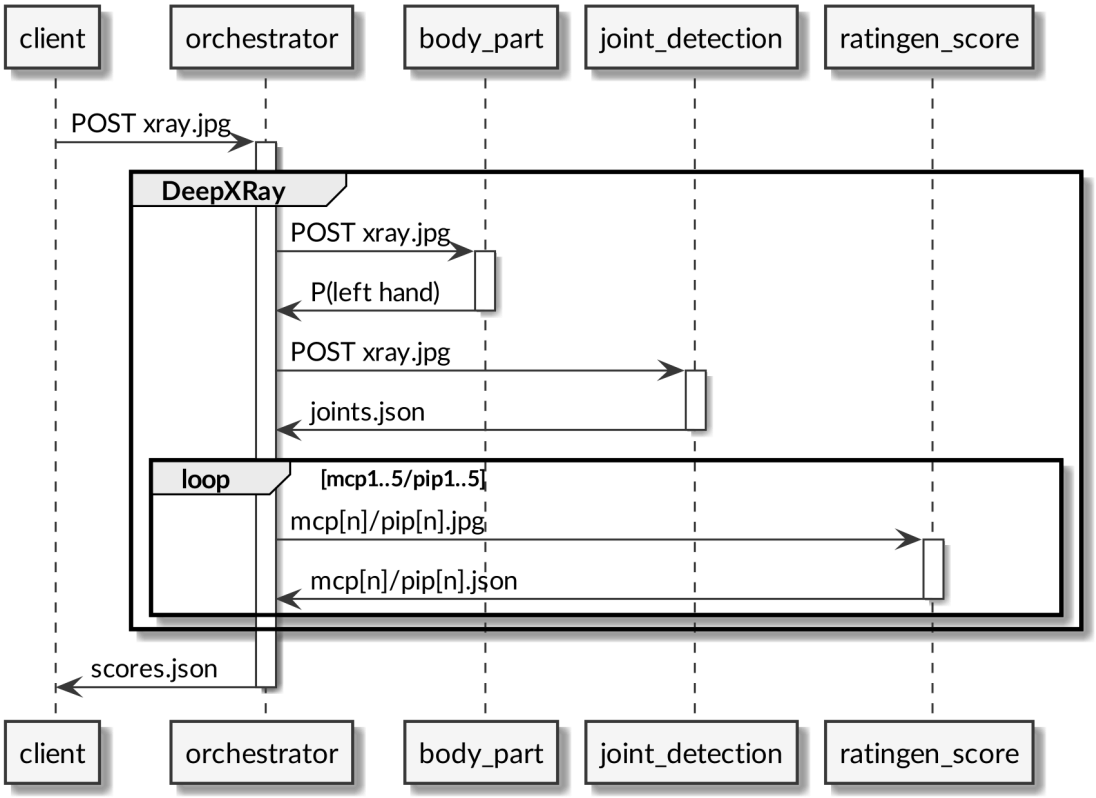


Abbildung 3: Datenfluss der Variante 1 (HTTP, synchron). Die einzelnen Modellkomponenten werden streng synchron aufgerufen und sequenziell abgearbeitet. (Sequenzdiagramm)

Nachteile Die Extraktion der Gelenke muss abgewartet werden, bis zum Scoring derselben übergegangen werden kann. Das Scoring findet streng sequenziell statt und ist daher langsam. Zu einem bestimmten Zeitpunkt kann jeweils nur eine Instanz der Modellkomponenten genutzt werden.

3.1.2 Variante 2: HTTP, synchron und asynchron

Die zweite Variante geht von der gleichen Komponentenarchitektur aus wie die erste Variante, siehe Abbildung 2, Seite 33 (Komponentendiagramm). Im Gegensatz zur ersten Variante soll die zweite Variante jedoch nicht streng synchron arbeiten. Hierzu ändert sich die interne Arbeitsweise der `orchestrator`-Komponente und die Schnittstelle zur Komponente `joint_detection`.

Der geänderte Ablauf ist auf Abbildung 4, Seite 35 ersichtlich. Die Erkennung des Kör-

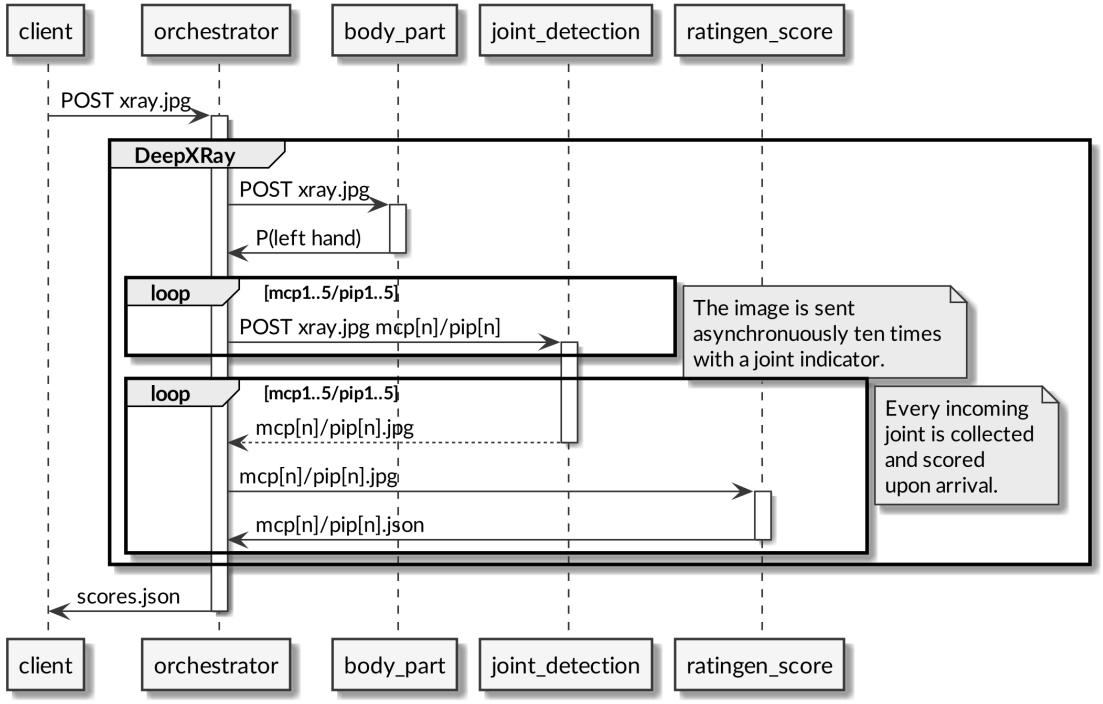


Abbildung 4: Datenfluss der Variante 2 (HTTP, synchron und asynchron). Es wird bereits mit dem Scoring begonnen, wenn das erste Gelenk extrahiert ist. (Sequenzdiagramm)

perteils auf dem Röntgenbild erfolgt wiederum über eine synchrone HTTP-`POST`-Anfrage an `body_part`.

Die Extraktion der Gelenke bei der `joint_detection`-Komponente erfolgt jedoch nicht mehr mit einer einzigen Anfrage, deren Antwort synchron abgewartet wird, sondern über eine Anfrage *pro Gelenk*, d.h. es werden zehn Anfragen an `joint_detection` gestellt: jeweils mit einem Röntgenbild und dem zu extrahierenden Gelenk. Diese Anfragen werden nebenläufig in einem Unterprozess pro Gelenk abgeschickt.

Sobald ein Unterprozess ein extrahiertes Gelenk von `joint_detection` als Antwort erhält, kann damit eine Anfrage für das Scoring an die Komponente `ratingen_score` gestellt werden, auf deren Antwort in jedem Unterprozess synchron gewartet wird.

Sind alle zehn Scores eingetroffen, können diese zu einer Antwort für den Client zusammengestellt und an diesen zurückgeschickt werden.

Diese hybride Variante (HTTP asynchron/synchron gemischt) hat Vorteile gegenüber der streng synchronen Variante, löst aber nicht alle Probleme:

Vorteile Es kann bereits mit dem Scoring der Gelenke begonnen werden, wenn das erste Gelenk extrahiert worden ist. So können `joint_detection` und `ratingen_score` gleichzeitig arbeiten, wodurch die Anfrage des Clients schneller bedient werden kann.

Nachteile Das Röntgenbild wird zehnmal statt nur einmal an `joint_detection` geschickt. Die Koordination nebenläufiger HTTP-Anfragen ist anspruchsvoller. Ohne zusätzlichen Load-Balancer ist weiterhin nur eine Instanz pro Komponente einsetzbar.

3.1.3 Variante 3: Messaging zwischen Modellkomponenten

Soll die Verarbeitung von Röntgenbildern mit mehreren Instanzen pro Komponente, d.h. parallel vonstatten gehen, wäre mit HTTP ein Load-Balancer pro Komponente nötig. Eine Alternative hierzu stellt das Messaging dar, welches in Abschnitt 2.3 *Integrationsvarianten*, Seite 19 vorgestellt worden ist.

Dabei wird die Arbeit nicht an die einzelnen Komponenten erteilt wie bei HTTP (Push), sondern von den einzelnen Komponenten (bzw. deren Instanzen) von einer Queue selbständig abgeholt (Pull). Dies ermöglicht eine parallele Abarbeitung, indem sich die unbeschäftigen Instanzen einer Komponente jeweils neue Arbeit nach dem Round-Robin-Verfahren von der Queue abholen.

Abbildung 5, Seite 37 zeigt eine mögliche Komponentenarchitektur, bei der intern nicht HTTP, sondern Messaging zum Einsatz kommt.[32] Bei dieser Variante werden die Komponenten der Reihe nach jeweils paarweise mit einer Queue verbunden, wodurch ein geschlossener Kreis entsteht. Über eine Error-Queue kann der Ablauf im Fehlerfall abgebrochen und als entsprechende Fehlermeldung an den `orchestrator` weitergegeben werden.

Gegenüber den ersten beiden Varianten (HTTP rein synchron und hybrid, d.h. asynchron/synchron) ändert sich der Ablauf grundlegend (siehe Abbildung 6, Seite 38). Der `orchestrator` empfängt wiederum das Röntgenbild per HTTP vom Client. Er reicht dieses über eine Queue weiter an die Komponente `body_part`. Hier gibt der `orchestrator` die Kontrolle über die Weiterverarbeitung ab.

[32] Hier fällt auf, dass die einzelnen Komponenten jeweils eine Queue *verwenden*, diese aber von keiner der involvierten Komponenten zur Verfügung gestellt wird. Tatsächlich werden die Queues in der Regel von einem *Message-Broker* zur Verfügung gestellt, der aus Gründen der Übersichtlichkeit nicht dargestellt ist. Andererseits gibt es auch Messaging-Lösungen, die ohne Message-Broker-Komponente auskommen, z.B. ZeroMQ.

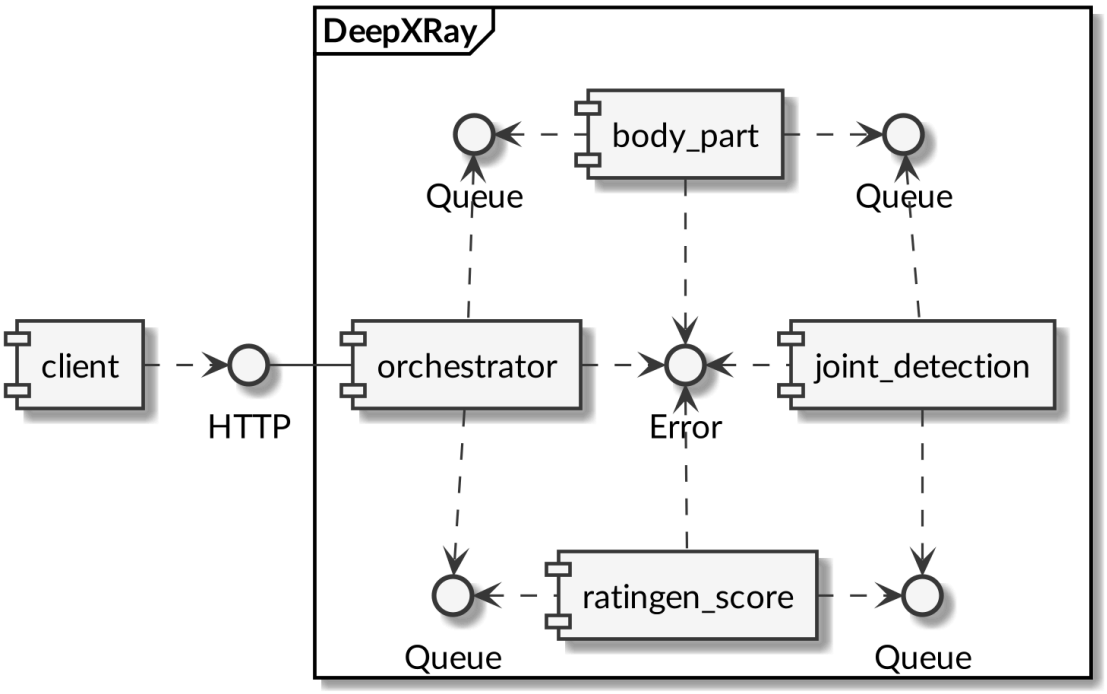


Abbildung 5: Komponentenarchitektur der Variante 3 (Messaging zwischen den Modellkomponenten). Mithilfe von Message-Queues werden die Komponenten paarweise miteinander verbunden, und bilden so einen Kreis.

Wird auf dem Röntgenbild eine linke Hand erkannt, reicht `body_part` das Röntgenbild zur Extraktion der Gelenke an `joint_detection` weiter, wiederum über eine Queue. Hier werden die zehn Gelenke der Reihe nach extrahiert. Jedes extrahierte Gelenk wird sogleich über eine weitere Queue an `ratingen_score` weitergegeben. Es kann wiederum mit dem Scoring begonnen werden, wenn die Extraktion der Gelenke noch in Gang ist.

Die Scores werden von `ratingen_score` über eine Queue an den `orchestrator` weitergereicht. Treten bei der Extraktion oder beim Scoring Fehler auf, werden diese über eine Error-Queue an den `orchestrator` gemeldet. Dieser braucht pro ursprünglich eingegangenem Request (Röntgenbild) zwei Queues zu beobachten: Die ermittelten Scores und die gemeldeten Fehler. Insgesamt müssen pro Vorgang zehn Nachrichten beim `orchestrator` eingehen, seien dies Scores oder Fehlermeldungen. Sind diese alle eingetroffen, kann der Client bedient werden.

Die asynchrone Verarbeitung über Message-Queues ermöglicht es, dass alle drei Modellkomponenten mit beliebig vielen Instanzen ausgeführt werden können, was die Ver-

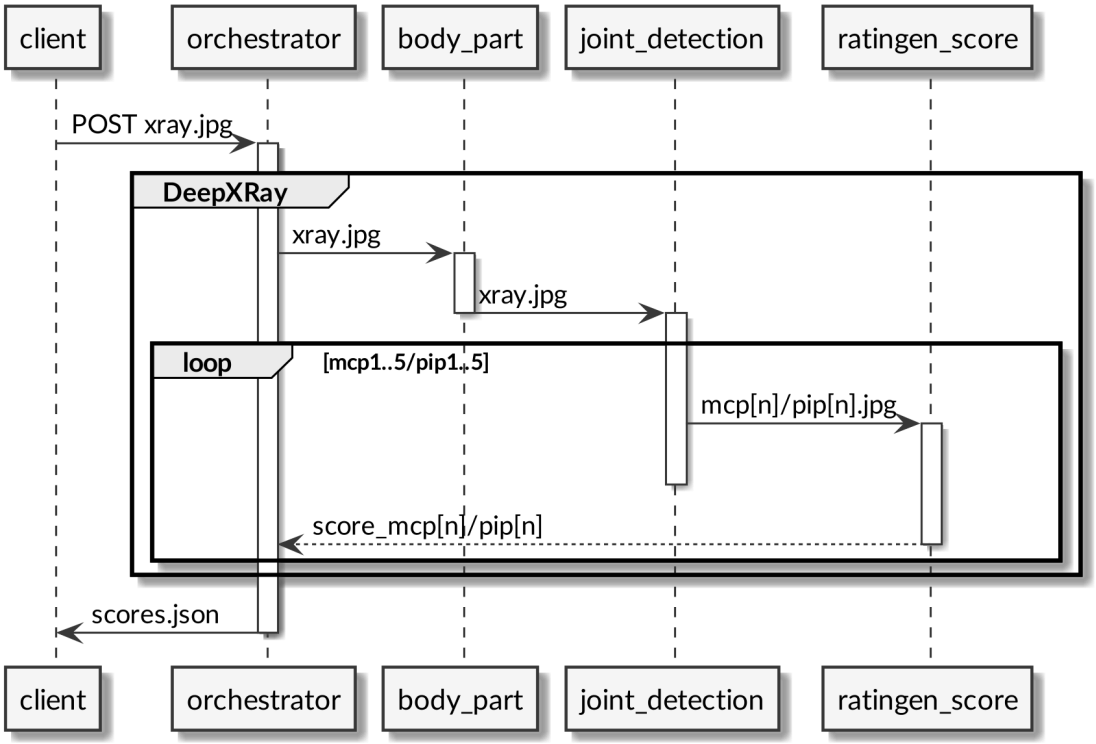


Abbildung 6: Datenfluss der Variante 3 (Messaging zwischen den Modellkomponenten). Die Anfragen werden via Message-Queue von Komponente zu Komponente weitergereit. (Sequenzdiagramm)

arbeitungsgeschwindigkeit bis zu einem gewissen Grad erhöht. Das «Einsammeln» der Ergebnisse wird über diese asynchrone Abarbeitung jedoch erschwert, zumal zu einem bestimmten Zeitpunkt die Nachrichten verschiedener Anfragen in den Queues hängen können. Der `orchestrator` benötigt darum einen Mechanismus, um die eintreffenden Nachrichten einer ursprünglichen HTTP-Anfrage zuordnen zu können. Andernfalls könnte der Client mit seiner Antwort Informationen bekommen, die zu anderen Anfragen gehören.

Dieses Problem kann mit einem *Correlation Identifier* gelöst werden (Hohpe et al., 2004, S. 163-169). Hierzu wird jeder Anfrage eine eindeutige Identifikation zugewiesen. Dies kann eine fortlaufende Nummerierung sein, oder aber ein generierter Zufallswert. Wichtig ist, dass dieser Wert im gegebenen Kontext eindeutig ist. Die Message wird mit diesem Correlation Identifier ausgestattet und für Folgenachrichten übernommen. Sammelt der `orchestrator` die eingehenden Nachrichten von den beiden Queues ein, kann er diese über den Correlation Identifier dur urpsprünglichen Anfrage zuordnen.

Diese auf Messaging basierte Architekturvariante stellt in vielerlei Hinsicht eine Verbesserung gegenüber den beiden HTTP-Varianten dar, bringt aber auch neue Probleme mit sich:

Vorteile Über den Messaging-Ansatz wird eine parallele Abarbeitung ohne zusätzliche Load-Balancer ermöglicht.

Nachteile Das asynchrone «Einsammeln» von Nachrichten und deren Zuordnung zu den ursprünglichen Client-Anfragen erfordert einen zusätzlichen Mechanismus (Correlation Identifier). Die dezentrale Kommunikation zwischen den Modellkomponenten ist komplizierter, da es mehr Kommunikationspaare gibt.

Zudem funktioniert der beschriebene Ansatz nur für das unmittelbar gegebene Problem: das Scoring von (linken) Händen, wobei immer genau zehn Gelenke von Interesse sind. Sollen in Zukunft auch die Röntgenbilder anderer Körperteile verarbeitet werden, etwa Becken mit zwei Hüftgelenken, funktioniert dieser Ansatz nicht mehr. Das Problem ist, dass der `orchestrator` nicht erfährt, welches Körperteil auf einem Röntgenbild dargestellt wird, und so nicht wissen kann, wie viele Scores (oder Fehlermeldungen) er einzusammeln hat.

3.1.4 Variante 4: Messaging, synchron und asynchon

Um für zukünftige Erweiterungen von anderen Körperteilen gerüstet zu sein, muss der `orchestrator` wissen, welches Körperteil auf einem Röntgenbild dargestellt ist, um so die richtige Anzahl an Scores bzw. Fehlermeldungen erwarten zu können. Darum wird `body_part` gegenüber der Variante 3 aus dem Kreis der vier Komponenten entfernt und synchron an den `orchestrator` gekoppelt. Die Komponentenarchitektur ist auf Abbildung 7, Seite 40 ersichtlich.

Obwohl die Kommunikation zwischen `body_part` und dem `orchestrator` synchron abläuft, soll dennoch eine Message-Queue als Schnittstelle zwischen diesen Komponenten dienen. Dies hat den Vorteil, dass `body_part` mit mehreren Instanzen gleichzeitig betrieben werden kann, die sich die Workloads abwechselnd von der Queue holen, die nicht nur als Übertragungsmedium, sondern auch als Load-Balancer fungiert.

Der Datenfluss ändert sich dabei gegenüber der vorherigen Messaging-Variante beträchtlich (siehe Abbildung 8, Seite 41). Das Körperteil wird – wie bei den beiden HTTP-Varianten – synchron detektiert. Wurde eine linke Hand erkannt, gibt der `orchestrator`

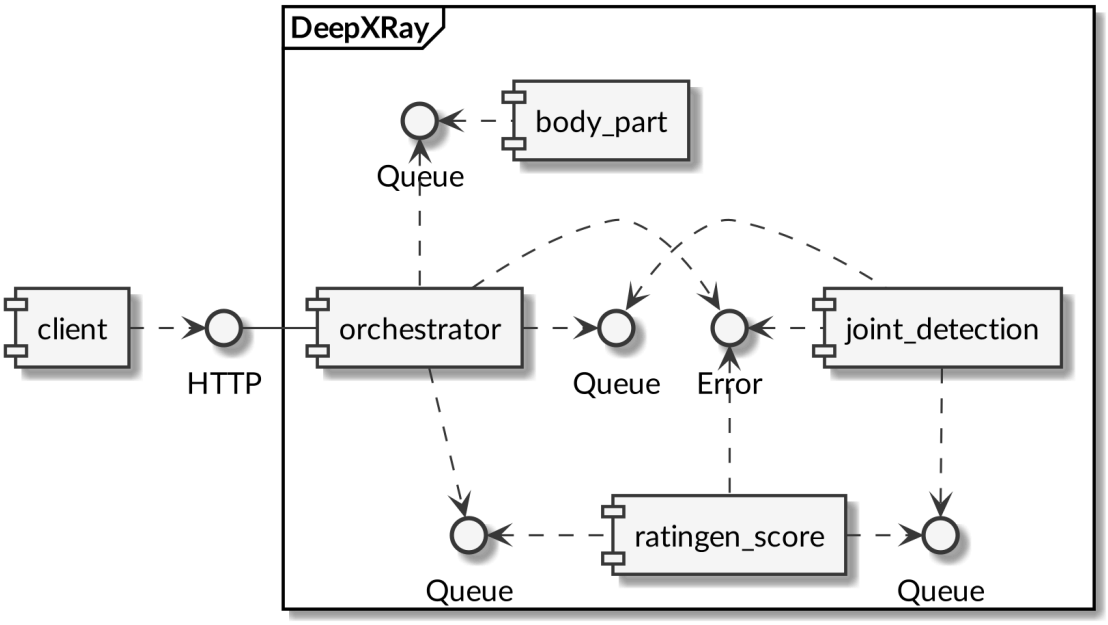


Abbildung 7: Komponentenarchitektur der Variante 4 (Messaging, synchron und asynchron). Die `body_part`-Komponente ist synchron, die anderen Modellkomponenten sind asynchron angebunden.

– und nicht `body_part`! – die Extraktionen in Auftrag. Da hier nun Messaging zum Einsatz kommt, können zehn entsprechende Nachrichten auf der Queue publiziert werden, von der `joint_detection` liest.

Extrahierte Gelenke werden über eine weitere Queue an `ratingen_score` weitergeleitet. Scheitert die Extraktion oder das Scoring, kann dies über eine Error-Queue an den `orchestrator` gemeldet werden. Da dieser die Extraktion der Gelenke in Auftrag gegeben hat, ist diesem die Anzahl der zu erwartenden Nachrichten bekannt. Sind diese Nachrichten eingetroffen, kann er den Client damit bedienen.

Diese Variante kombiniert die Vorteile der synchronen Erkennung des Körperteils mit den Vorteilen der nebenläufigen Extraktion und dem Scoring der Gelenke:

Vorteile Die Gelenke können nebenläufig extrahiert und gescored werden, was eine höhere Verarbeitungsgeschwindigkeit ermöglicht. Mithilfe von Messaging können die Komponenten mit mehreren Instanzen parallel betrieben werden. Zudem funktioniert die Architektur auch dann noch, wenn andere Körperteile als linke Hände (mit einer anderen Anzahl relevanter Gelenke) verarbeitet werden sollen.

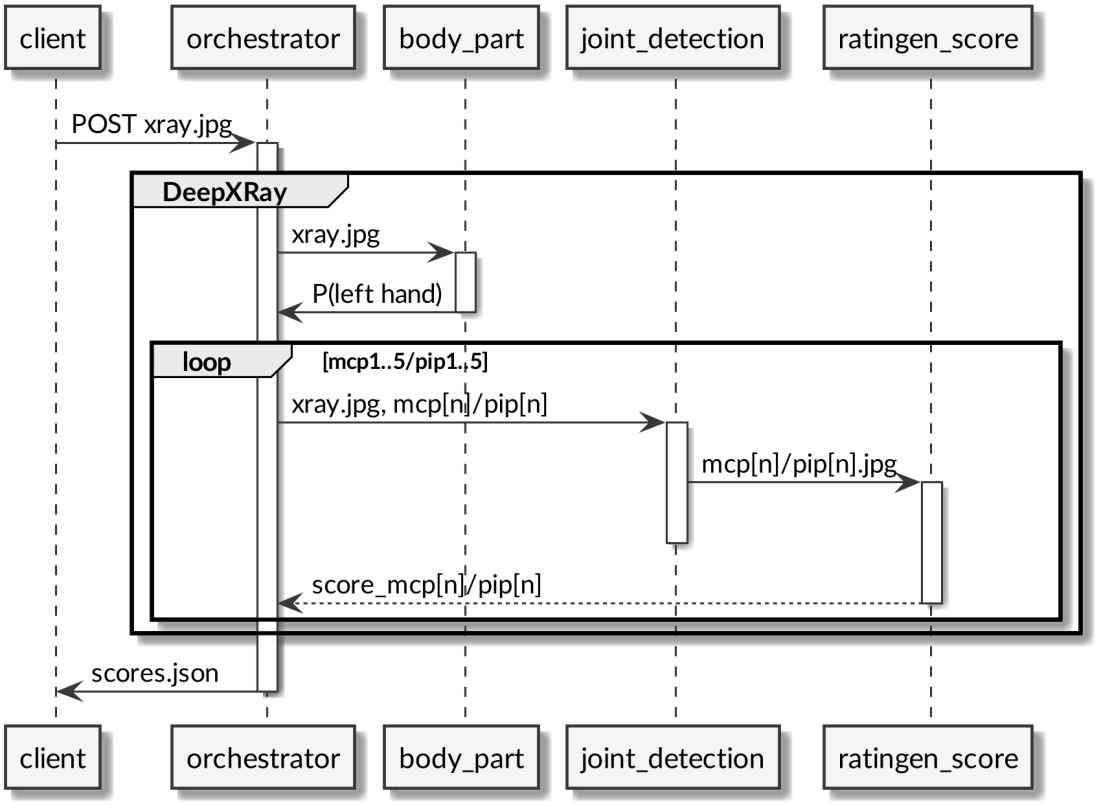


Abbildung 8: Datenfluss der Variane 4 (Messaging, synchron und asynchron). Die Modellkomponente `body_part` wird synchron, die Modellkomponenten `joint_detection` und `ratingen_score` werden asynchron angebunden. (Sequenzdiagramm)

Nachteile Das «Einsammeln» der Nachrichten erfordert – gleich wie die vorherige Messaging-Variante – einen Correlation Identifier. Der `orchestrator` hat noch mehr Kommunikationsaufgaben. Zudem wird das Röntgenbild einmal pro Gelenk publiziert, nicht einmal für alle zehn Gelenke.

3.1.5 Entscheidung

Jede Variante bietet Vorteile gegenüber ihrem Vorgänger: Variante 1 (HTTP, synchron) arbeitet streng sequenziell. Variante 2 (HTTP, asynchron/synchron) kann bereits mit dem Scoring beginnen, wenn die Extraktion noch nicht abgeschlossen ist. Variante 3 (Messaging zwischen Modellkomponenten) ermöglicht die Ausführung mehrerer Instanzen eines Modells. Variante 4 (Messaging, synchron und asynchron) lässt sich darüber hinaus

am besten erweitern. Trotz der Nachteile von Variante 4 – Notwendigkeit eines Correlation Identifiers, mehrfache Publikation des Röntgenbilds (einmal pro Gelenk), verschiedenartige Kommunikationsaufgaben (synchron und asynchron) – ist sie die flexibelste und am besten erweiterbare Variante.

Der Prototyp soll darum gemäss Variante 4 (Messaging, synchron und asynchron) umgesetzt werden.

3.2 Austauschbarkeit von Modellen

Die Modelle – oder Modellkomponenten – müssen gemäss Projektauftrag (siehe 1.1.2 *Erwartetes Resultat*, Seite 11) austauschbar sein. Für diese Anforderung gibt es verschiedene Lösungsansätze.

Ein Lösungsansatz wäre die Austauschbarkeit auf Modellebene. Hierzu wurden entsprechende Formate – TensorFlow SavedModel und ONNX – im Abschnitt 2.5 *Modellformate*, Seite 29 vorgestellt. Die bestehenden Modelle lassen sich jedoch nicht einfach in eines dieser Formate konvertieren. Zudem ist die Wahl eines dieser Formate noch keine Garantie für Vorwärtskompatibilität in einigen Jahren. Ändert sich die Modellspezifikation (Inputs, Outputs) nur geringfügig, ist die Austauschbarkeit auf Modellebene bereits nicht mehr gegeben. Die Austauschbarkeit von Modellen (oder Modellkomponenten) über die Modelldatei ist somit für das vorliegende Projekt keine Option.

Statt die Modelle selber austauschbar zu machen, und so Änderungen an ihren Inputs und Outputs für längere Zeit zu verunmöglichen, können die Modell*komponenten* austauschbar gemacht werden. Hierfür wird eine API spezifiziert, an die sich die Modellkomponenten halten müssen. Da für den Prototyp Messaging zum Einsatz kommen soll (siehe 3.1.4 *Variante 4: Messaging, synchron und asynchon*, Seite 39), ist diese API einerseits über das Nachrichtenformat, andererseits über die verwendete Messaging-Technologie (z.B. AMQP oder MQTT) zu definieren.

Da die verwendeten Modelle auf verschiedenen Versionen von TensorFlow basieren und mit unterschiedlichen High-Level-APIs (tflearn, Keras) erstellt worden sind, können diese nicht in der gleichen Laufzeitumgebung ausgeführt werden.

Mit virtuellen Python-Umgebungen ist es möglich, auf dem gleichen Rechner verschiedene Python-Versionen mit den dazugehörigen Libraries und Frameworks auszuführen. Mit dem Modul `venv` [33] können verschiedene virtuelle Python-Umgebungen auf einem Rechner erstellt werden, die auf der installierten Python-Version basieren.

[33] `https://docs.python.org/3/library/venv.html` (abgerufen am 11.05.2020)

Da die verwendeten Frameworks/Libraries jedoch nicht zu den neuesten Versionen von Python kompatibel sind, müssten auch verschiedene Versionen des Python-Interpreters installiert werden können. Das Modul `pyenv`[34] unterstützt dies. Diese Variante ist aber für den Produktiveinsatz riskant, zumal mehrere ältere Versionen des Python-Interpreters auf dem gleichen System laufen, und so den Angriffsvektor dieses Systems vergrössern.[35]

Container ermöglichen nicht nur die Installation praktisch beliebiger (älterer) Versionen von Python, sondern auch die Isolation dieser Umgebungen. So können ältere Versionen verwendet werden, ohne die Sicherheit des zugrundeliegenden Systems zu beeinträchtigen. Container-Runtimes wie Docker ermöglichen es zudem, Ports für die Kommunikation zwischen dem Container und dem zugrundeliegenden System bzw. Ports zwischen einzelnen Containern per Konfiguration zu definieren.

Werden die Modellkomponenten als Container umgesetzt, können sie auf Basis ihrer Schnittstelle – Messaging-Format, Messaging-Protokoll und Port – austauschbar gemacht werden. Welche Libraries, Frameworks, Python-Versionen und Modellformate dabei zum Einsatz kommen, ist für die Umsysteme nicht relevant.

Die Austauschbarkeit von Modellkomponenten soll darum mithilfe von Containern gewährleistet werden.

3.3 Parallelisierung – Nebenläufigkeit

Der Projektauftrag nennt das «Verteilen über mehrere GPUs» als Anforderung (siehe 1.1.2 *Erwartetes Resultat*, Seite 11). Dieser ist nicht ohne Weiteres nachzukommen, da zur Ausführung eines Modells auf einer GPU dieses komplett in den Speicher derselben geladen werden muss. Da für die Gelenkextraktion nicht nur ein Modell, sondern zehn verschiedene Modelle in den Speicher geladen werden müssen, dürfte eine Vielzahl von GPUs nötig sein, um die Ausführung damit zu beschleunigen. Ein solches Setup steht für die Entwicklung nicht und für den Produktiveinsatz wohl kaum zur Verfügung.[36]

Auch wenn dieses optimale, auf GPUs basierende Setup vorerst keine Option ist, ist

[34] `https://github.com/pyenv/pyenv` (abgerufen am 11.05.2020)

[35] Das gestellte Problem ist ja, dass aufgrund von Inkompatibilitäten auf Ebene Libraries/Frameworks ältere Versionen von Python verwendet werden müssen. Das Modell `ratingen_score` wurde etwa mit Python 3.5 erstellt – eine Version, die nur noch bis am 13.09.2020 Updates erhalten soll (`https://devguide.python.org/#branchstatus`, abgerufen am 11.05.2020).

[36] Der Cloud-Anbieter Exoscale, der auch Hosting in der Schweiz anbietet, was für sensible Patientendaten wie Röntgenbilder auch erforderlich ist, verlangt für eine Instanz mit vier GPUs 2.28 € pro Stunde, d.h. bis zu 1696.32 € pro Monat.

die parallele Ausführung der verschiedenen Modelle für eine akzeptable End-to-End-Perfomance unbedingt erforderlich – und darum auch für die Architekturdiskussion (siehe 3.1 *Architekturvarianten*, Seite 31) eine grundlegende Anforderung.

Doch genügt eine geeignete Architektur, um Parallelisierung zu gewährleisten? Um diese Frage beantworten zu können, lohnt sich ein genauerer Blick auf die Begriffe *Nebenläufigkeit* (engl. *concurrency*) und *Parallelität* (engl. *parallelism*).

Die Begriffe werden oft synonym gebraucht, haben aber unterschiedliche Bedeutungen – die zugegebenermassen in einer Welt mit allgegenwärtigen Multicore-Systemen kaum noch erfahrbar sind.

Doch der Reihe nach: Rob Pike, einer der Schöpfer der Programmiersprache Go, definiert die beiden Begriffe folgendermassen: *«Concurrency and Parallelism are not the same thing. Concurrency is the composition of independently executing processes. Parallelism, on the other hand, ist the simultaneous execution of multiple things. One is really about structure (concurrency), and one is about execution (parallelism).»*[37] (Pike, 2013, ab 1:16).

Joe Armstrong, der 2019 verstorbene Schöpfer der Programmiersprache Erlang, geht genauer auf die Hardware-Aspekte dieser Konzepte ein: *«In everyday language, words like concurrent, simultaneous, and parallel mean almost the same thing. But in programming languages, we need to be more precise. In particular, we need to distinguish between concurrent and parallel programs. If we have only a single-core computer, then we can never run a parallel program on it. This is because we have one CPU, and it can do only one thing at a time. We can, however, run concurrent programs on a single-core computer. The computer time-shares between the different tasks, maintaining the illusion that the different tasks run in parallel.»*[38] (Armstrong, 2013, S. 8).

Parallelisierung erreicht man also dadurch, indem man ein nebenläufiges Programm auf Hardware mit mehreren CPU-Kernen bzw. mehreren CPUs laufen lässt. Ein nebenläufiges Programm muss so strukturiert sein, dass es potenziell mit mehreren Vorgängen gleichzeitig zurecht kommt. Eine solche Struktur wurde in der Architekturdiskussion er-

[37] Nebenläufigkeit und Parallelität sind nicht das Gleiche. Nebenläufigkeit ist die Komposition unabhängig voneinander ablaufender Prozesse. Parallelität hingegen ist die gleichzeitige Ausführung mehrerer Dinge. Bei einem geht es um die Struktur (Nebenläufigkeit), beim anderen geht es um die Ausführung (Parallelität). (Übersetzung des Autors)

[38] In der Alltagssprache bedeuten Wörter wie «nebenläufig», «gleichzeitig» und «parallel» fast das Gleiche. Bei Programmiersprachen muss man aber präziser sein. Besonders «nebenläufig» und «parallel» muss man voneinander unterscheiden. Hat man nur einen Single-Core-Computer, kann man niemals ein paralleles Programm auf ihm laufen lassen. Das liegt daran, dass man nur eine CPU hat, und diese kann nur eine Sache zur gleichen Zeit machen. Wir können jedoch ein nebenläufiges Programm auf einem Single-Core-Computer ausführen. Der Computer wechselt zwischen den verschiedenen Aufgaben hin und her, und hält die Illusion aufrecht, mehrere Aufgaben parallel abzuarbeiten. (Übersetzung des Autors)

arbeitet (siehe 3.1.4 *Variante 4: Messaging, synchron und asynchon*, Seite 39). Wird der Prototyp entsprechend umgesetzt, ist Parallelität praktisch automatisch gegeben, zumal heute kaum noch Rechner mit nur einem CPU-Kern im Einsatz sind.

4 Methoden

Das Projekt *DeepXRay* hat ein klares Ziel (siehe 1.1 *Projektauftrag*, Seite 9): Auf Basis bestehender Machine-Learning-Modelle soll ein Prototyp für einen Webservice erstellt werden, der sich später in andere Anwendungen integrieren lässt. Ein *rein* agiles Vorgehen ist somit für das Projekt nicht geeignet.

Die Ausgangslage des Projekts zu verstehen, und die zur Verfügung gestellten Artefakte verwenden zu können, ist ein wichtiger Teil des Projekts. An die Integration der einzelnen Teile kann erst gedacht werden, wenn deren Funktionsweise verstanden ist. Eine reine Top-Down-Planung wie im V-Modell oder im Wasserfallmodell ist somit nicht praktikabel.

Ein hybrides Modell, bei dem das übergeordnete Ziel (lauffähiger Prototyp für Webservice) von Anfang an bekannt ist, die einzelnen Aufgabenblöcke aber erst detailliert geplant werden, wenn deren Voraussetzungen geschaffen worden sind, ist somit besser geeignet.

4.1 Projektphasen

Das Projekt soll in den folgenden drei Phasen umgesetzt werden:

Erste Phase (Modelle) Die bestehenden Machine-Learning-Modelle – drei an der Zahl – sind schon etwas älter und basieren auf verschiedenen Technologien. Die Modelle müssen wieder lauffähig gemacht werden, sodass man sie für das Erstellen von Predictions verwenden kann. Die Inputs und Outputs der Modelle müssen dabei verstanden und dokumentiert werden, damit die nächste Phase in Angriff genommen werden kann.

Zweite Phase (Architektur) Der zu erstellende Webservice-Prototyp hat einerseits eine externe Schnittstelle, die zu einem späteren Zeitpunkt von anderen Applikationen angesteuert werden können muss, und andererseits interne Schnittstellen, um die Machine-Learning-Modelle zu koordinieren. Beide Schnittstellen müssen definiert werden. Hierbei kann die Gestaltung der internen Schnittstellen einerseits durch die bestehenden Artefakte und andererseits durch die gewählte externe Schnittstelle beeinflusst werden. Am Ende der zweiten Phase soll auf Basis der besprochenen Architekturansätze ein Architekturentscheid gefällt werden.

Dritte Phase (Orchestrierung) Auf Basis der lauffähigen Modelle und der entworfenen Architektur sollen nun die einzelnen Komponenten orchestriert werden, sodass sie

gegen aussen als ein System ansprechbar sind. Der Prototyp ist auf einer für Demo- und Testzwecke geeigneten Umgebung ausführbar. Für die Anwendung als ganzes sowie für deren einzelne Komponenten gibt es automatisierte Testfälle. Weiter wird eine weitgehend automatisierte Evaluation benötigt, womit die Qualität des Systems eingeschätzt – und später mit verbesserten Versionen des Systems verglichen werden kann.

Da die Aufgaben und die Struktur der einzelnen Phasen jeweils von den Ergebnissen der vorherigen Phasen abhängig sind, kann immer nur zum Ende der einen Phase die nächste Phase geplant werden. Dieses Kapitel wird deshalb im Projektverlauf laufend ergänzt. Das Ende einer jeden Phase ist sogleich ein Meilsenstein, für welchen Ziele, Artefakte und Termine zu definieren sind. Am Ende jeder Projektphase wird diese im Bezug auf Risiken und die Zielerreichung reflektiert.

Die Projektplanung beschränkt sich dabei auf das zu erstellende Software-Artefakt: den Webservice-Prototyp. Dokumentationsaufgaben fliessen dabei nur in die Projektplanung ein, wenn eine spätere Projektphase von deren Ergebnis abhängt. Dies ist etwa bei der Dokumentation der Modelle (Inputs/Outputs) und den Architekturvarianten der Fall. Die drei anderen verlangten Artefakte – Bericht, Web-Abstract und Video – werden im Verlauf des Projekts erstellt und höchstens in der Wochenplanung berücksichtigt, tauchen aber nicht im Abhängigkeitsbaum der einzelnen Phasenpläne auf.

Die einzelnen Aufgaben und Unteraufgaben sollen nicht weiter in Arbeitspakete heruntergebrochen oder zeitlich (Aufwand und Zeitpunkt) eingeplant werden, zumal der zeitliche Rahmen und die Grösse der Aufgabe für ein Projekt, das von einer einzelnen Person umgesetzt wird, problemlos überblickt werden kann. Die vorliegende Projektplanung soll v.a. zur Erkennung der Abhängigkeiten und zum Handhaben der Risiken dienen – nicht für eine detaillierte Arbeitsplanung.

4.1.1 Erste Phase: Modelle

- **Ziel:** ausführbare Modelle mit dokumentierter Schnittstelle
- **Artefakte**
  1. Modelldaten, Ausführungscode und Laufzeitumgebung für jedes Modell
  2. Dokumentation der Inputs und Outputs
  3. Planung Phase 2

- **Aufgaben**
  1. Modell `body_part`
     a) Laufzeitumgebung
     b) Code für Prediction
     c) Dokumentation
  2. Modell `joint_extraction`
     a) Laufzeitumgebung
     b) Code für Prediction
     c) Dokumentation
  3. Modell `ratingen_score`
     a) Modelldaten
     b) Laufzeitumgebung
     c) Code für Prediction
     d) Dokumentation
  4. Planung der zweiten Phase
- **Risiken und Mitigationen**
  1. Modell `ratingen_score` kann nicht rechtzeitig fertiggestellt werden.
     a) Die Arbeiten werden in Phase 2 fortgeführt.
     b) Das Modell wird durch ein Dummy-Modell simuliert.
  2. Die Modelle erstellen unbrauchbare Predictions.
     a) Das Risiko muss getragen werden, zumal die Performance der Modelle nicht Bestandteil der Arbeit ist.
  3. Die Modelle haben eine schlechte Laufzeitperformance.
     a) Architektonische Massnahmen (z.B. Parallelisierung) zur Erhöhung der Performance werden ergriffen.
     b) Eine geeignete Laufzeitumgebung wird gewählt.

Reflexion Das Modell `ratingen_score` konnte neu trainiert und erfolgreich exportiert werden. Alle drei Modelle sind in einer isolierten Umgebung (Docker-Container) lauffähig. Die Predictions der Modelle sind plausibel, wobei hierfür keine systematischen Tests vorgenommen worden sind. Die Laufzeitperformance erscheint als akzeptabel; die Predictions laufen in Sekundenschnelle ab.

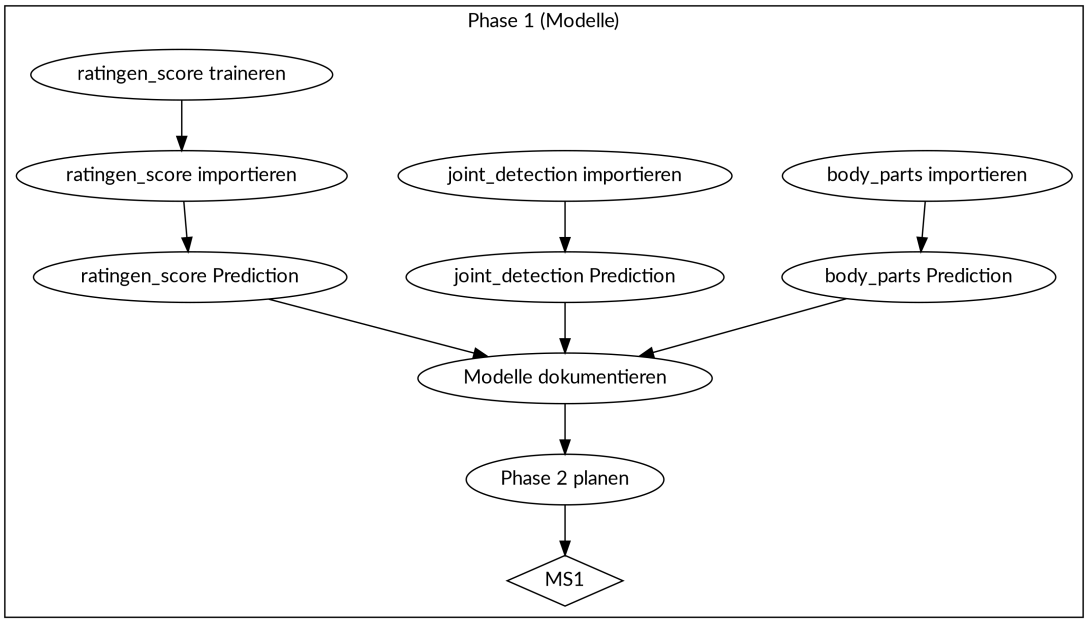


Abbildung 9: In der ersten Projektphase werden die drei Modelle lauffähig gemacht und dokumentiert.

4.1.2 Zweite Phase: Architektur

- **Ziel:** Architektur und Evaluationsmetriken festlegen
- **Artefakte**
  1. Architekturvorschläge und -diskussion
  2. begründeter Architekturentscheid
  3. Liste mit einzusetzenden Technologien
  4. Evaluationsmetriken
  5. Testkonzept
  6. Planung Phase 3
- **Aufgaben**
  1. Architektur
     a) Integrationsvarianten finden
     b) Architekturvorschläge aufstellen
     c) Architekturvariante auswählen
     d) Technologien finden

e) Prototyp implementieren

2. Evaluation

a) Datentypen finden
b) Datentypen der Modelle ermitteln
c) Evaluationsmetriken finden
d) Evaluationsmetriken bewerten
e) Geeignete Evaluationsmetriken auswählen

- **Risiken und Mitigationen**
  1. Es lassen sich keine geeigneten Technologien für einen Architekturvorschlag finden.
     a) Es wird auf eine andere Architekturvariante ausgewichen.
  2. Der Prototyp kann mit der vorgeschlagenen Architekturvariante nicht umgesetzt werden.
     a) Es wird auf eine andere Architekturvariante ausgewichen.
     b) Ein weiterer Prototyp wird implementiert, um die andere Variante zu prüfen.
  3. Es gibt keine geeigneten Evaluationsmetriken für das Gesamtsystem.
     a) Die Evaluation wird isoliert für die einzelnen Modelle vorgenommen.

Reflexion Das Auffinden verschiedener Integrationsvarianten, dazu passender Technologien und Evaluationsmetriken gestaltete sich sehr einfach. Hier war eher die Vielzahl der Möglichkeiten als deren Mangel ein Problem. Die Ausgestaltung und Auswahl der Architekturvarianten hat sehr viel Zeit in Anspruch genommen. Diese Investition hat sich jedoch gelohnt, zumal der Prototyp auf Anhieb funktioniert hat und als Vorlage für den zu erstellenden Webservice dienen kann.

4.1.3 Dritte Phase: Orchestrierung

- **Ziel:** lauffähiger Prototyp mit Evaluationsergebnissen
- **Artefakte**
  1. ausführbarer Programmcode

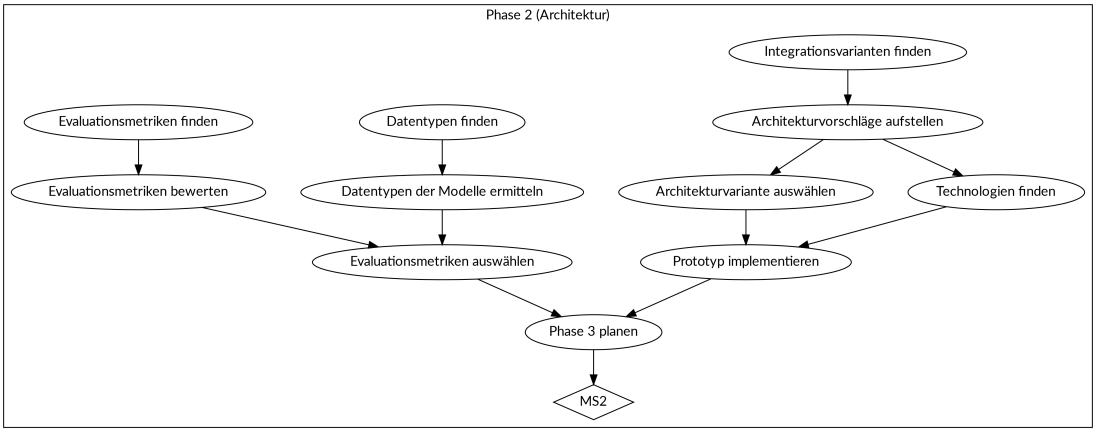


Abbildung 10: In der zweiten Projektphase sollen Architekturvarianten und Evaluationsmetriken gefunden werden. Mithilfe eines Prototyps wird der Architekturvorschlag validiert.

2. ausführbare Testfälle
3. gelabelte Evaluationsdaten
4. Evaluationsworkflow (Programmcode)
5. Evaluationsergebnisse

- **Aufgaben**
  1. Webservice-Prototyp
     a) `body_part` umsetzen
     b) `joint_detection` umsetzen
     c) `ratingen_score` umsetzen
     d) `orchestrator` umsetzen
     e) Web-UI umsetzen
  2. Evaluation
     a) Evaluationsdaten selektieren
     b) Evaluationsdaten aufbereiten
     c) Evaluationsdaten scoren
     d) Evaluationsworkflow umsetzen
     e) Ergebnisse evaluieren

- **Risiken und Mitigationen**
  1. Der Webservice läuft zu langsam.

   a) Es soll eine performantere Laufzeitumgebung verwendet werden.
   b) Die geringe Performance soll analysiert, Verbesserungsvorschläge sollen im Ausblick dokumentiert werden.
2. Die asynchrone Architektur führt zu inkonsistenten Ergebnissen (Vermischung der Resultate).
   a) Es sollen zuverlässige Synchronisationsmassnahmen eingesetzt werden.
   b) Es kann auf eine synchrone Architektur ausgewichen werden.
3. Es stehen keine oder zu wenige Evaluationsdaten zur Verfügung.
   a) Der Evaluationsworkflow wird nur mit Platzhalterdaten ausgeführt. Die Evaluationsergebnisse sollen zu einem späteren Zeitpunkt vom Auftraggeber ermittelt werden.
4. Die Evaluationsdaten lassen sich nicht in nützlicher Frist verarbeiten.
   a) Es soll nur eine Untermenge der Evaluationsdaten verarbeitet werden.
5. Die Evaluationsergebnisse sind unbefriedigend (schlechte Performance).
   a) Dieses Risiko ist für den Auftraggeber zu tragen, da die Modelle (und somit die Prediction-Performance) vorgegeben waren.

Reflexion Der Prototyp aus Phase 2 hat sich als Glücksfall erwiesen, zumal der Webservice analog zu ihm umgesetzt werden konnte. Mit den gewählten Technologien (Go, RabbitMQ, Docker) konnte der Webservice erstaunlich schnell umgesetzt werden – und lässt sich ohne Änderungen auf verschiedenen Umgebungen ausführen. Die Laufzeitperformance des Gesamtsystems ist für einen Prototyp akzeptabel, für den Produktiveinsatz jedoch verbesserungswürdig. Es konnten (in teilweise manueller Arbeit) genügend Evaluationsdaten gesammelt werden, um damit den Evaluationsworkflow umzusetzen und angemessen auf Plausibilität testen zu können. Für eine aussagekräftige Evaluation wären jedoch noch weitere ungesehene Daten nötig.

4.1.4 Meilensteinplanung

Die Meilensteinplanung (siehe Tabelle 1, Seite 54) deckt sich grösstenteils mit der Projektphasenplanung, zumal am Ende einer jeden Projektphase ein Meilenstein erreicht werden soll. Als vierter und entscheidender Meilenstein kommt die Schlussabgabe hinzu, bei der zusätzlich die Artefakte Bericht, Web-Abstract und Pitching-Video abzuliefern sind.

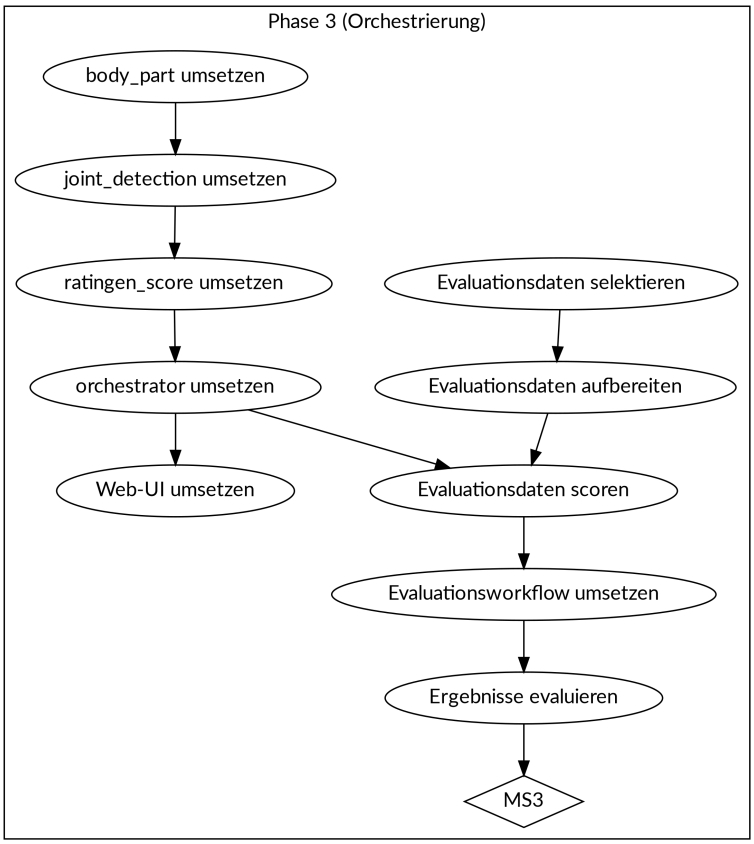


Abbildung 11: In der dritten Projektphase werden die einzelnen Teile zu einem lauffähigen System kombiniert. Dieses wird mit ausgewählten Metriken und entsprechenden Daten evaluiert.

4.1.5 Wochenplan

Obwohl in der Projektplanung (siehe 4.1 *Projektphasen*, Seite 46) auf eine detaillierte zeitliche Planung der einzelnen Aufgaben innerhalb der verschiedenen Projektphasen verzichtet worden ist, soll der Wochenplan (siehe Tabelle 2, Seite 54) eine zusätzliche zeitliche Orientierung ermöglichen. Im Gegensatz zum Meilensteinplan ist im Wochenplan auch die Erstellung der Schlusspräsentation berücksichtigt, welche in den Wochen nach der Abgabe erfolgen soll.

MS	SW	Datum	Beschreibung
1	6	31.03.2020	Phase 1 (Modelle) abgeschlossen
2	8	19.04.2020	Phase 2 (Architektur) abgeschlossen
3	13	24.05.2020	Phase 3 (Orchestrierung) abgeschlossen
4	15	05.06.2020	Schlussabgabe

Tabelle 1: In der Meilensteinplanung wird neben den drei detailliert geplanten Projektphasen auch die Schlussabgabe als vierter Meilenstein berücksichtigt.

SW	Von	Bis	Tätigkeiten	Ereignis/Meilenstein
0	12.02.	16.02.	Projektinitialisierung (Kick-Off)	Projektstart
1	17.02.	23.02.	Recherche, Bestandesaufnahme	
2	24.02.	01.03.	Recherche, Import bestehender Modelle	
3	02.03.	08.03.	Recherche, Import bestehender Modelle	
4	09.03.	15.03.	Recherche, Projektplanung	Erteilung def. Projektauftrag
5	16.03.	22.03.	Import/Ausführung bestehender Modelle	
6	23.03.	29.03.	Import/Ausführung bestehender Modelle, Dokumentation	
7	30.03.	05.04.	Dokumentation, Planung, Architektur	MS1
8	06.04.	12.04.	Architektur	
9	13.04.	19.04.	Architektur, Dokumentation, Planung	MS2
10	20.04.	26.04.	Umsetzung, Dokumentation	Zwischenpräsentation
11	27.04.	03.05.	Umsetzung, Dokumentation	
12	04.05.	10.05.	Umsetzung, Dokumentation	
13	11.05.	17.05.	Umsetzung, Dokumentation	
14	18.05.	24.05.	Deployment, Dokumentation	MS3, lauffähiger Prototyp
15	25.05.	31.05.	Dokumentation, Video	
16	01.06.	05.06.	Abschliessen Dokumentation	Schlussabgabe
	06.05.	30.06.	Abschlusspräsentation erstellen	Abschlusspräsentation

Tabelle 2: Der Wochenplan dient – als Ergänzung zu Phasen- und Meilensteinplan – als zeitliche Orientierungshilfe für den Projektablauf.

4.2 Teststrategie

Wie bereits im Modul *Software Testing* (Frühlingssemester 2019) eingeübt und im *Wirtschaftsprojekt* (Herbstsemester 2019) gewinnbringend eingesetzt, sollen die *Agile Test Quadrants* (Abbildung 12, Seite 55) wiederum als Grundlage für die Erarbeitung einer Teststrategie dienen.[39]

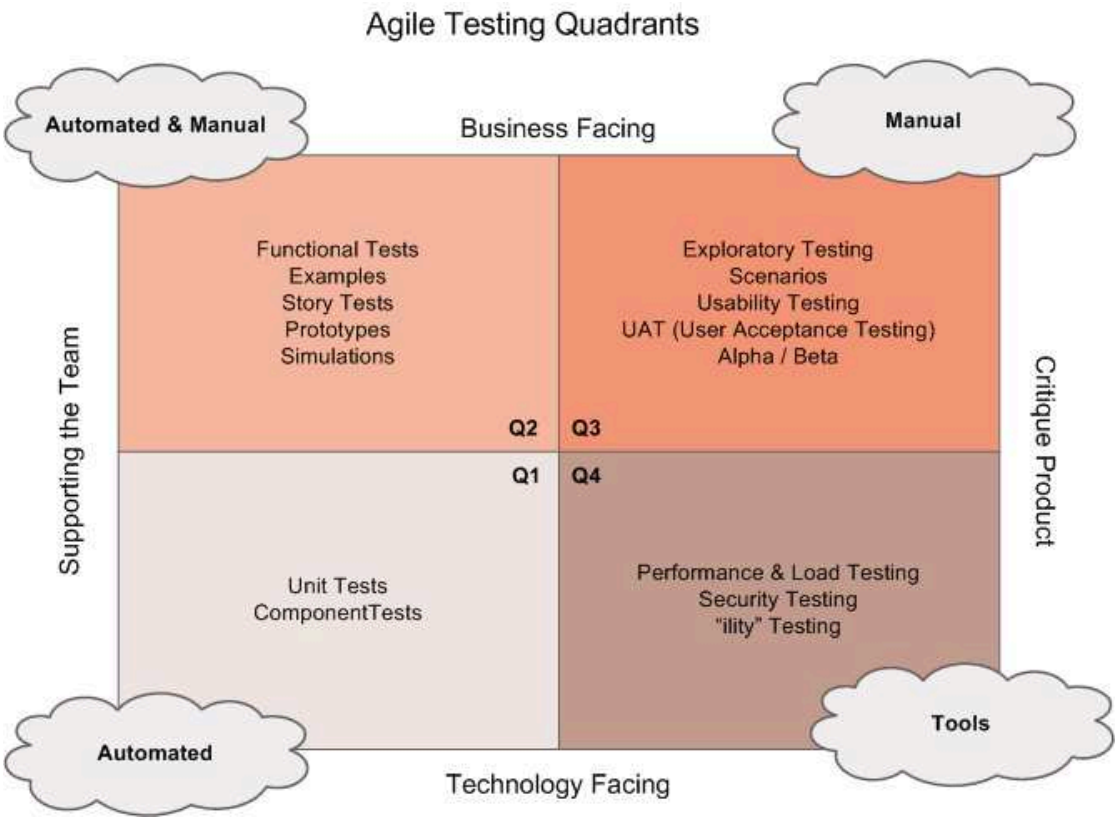


Abbildung 12: Die *Agile Testing Quadrants* (`https://lisacrispin.com/2011/11/08/using-the-agile-testing-quadrants`) sind eine bewährte Orientierungshilfe für das Erarbeiten einer Teststrategie.

Die verschiedenen Arten von Tests werden dabei als Werkzeuge betrachtet, die man für geeignete Aufgaben einsetzt – und für weniger geeignete Aufgaben wieder in die Werkzeugkiste zurücklegt. Die Wahl der Testwerkzeuge soll pragmatisch erfolgen, d.h. im Hinblick auf ihre Effektivität (ein Systemmerkmal wird sinnvoll geprüft) und Effizienz (mit wenig Testcode kann viel produktiver Code abgedeckt werden). Eindimensionale Metri-

[39] Auf die Umsetzung der Tests wird erst im Folgekapitel 5 *Realisierung*, Seite 60 eingegangen.

ken wie *«Es soll 95% des Codes mit Unittests abgedeckt werden»* lesen sich bei diesem Verständnis wie *«Der Bau der Hundehütte soll weitgehendst mit dem Hammer erfolgen»*. [40]

Die vier Quadranten, und ihre möglichen Anwendungen für das vorliegende Projekt *DeepXRay*, werden im Folgenden beschrieben.

4.2.1 Q1: automatisiert

Im ersten Quadranten geht es um Unittests und Komponententests. Diese sollen v.a. für den neu zu erstellenden Code geschrieben werden, der keine Abhängigkeit zu anderen Laufzeitkomponenten hat. Beim bestehenden Code (Modelle) ist dies nur teilweise sinnvoll, zumal es sich dabei um Legacy-Code handelt, der eher abgelöst als erweitert werden soll. Beim `orchestrator`, der viele Abhängigkeiten zu anderen Laufzeitkomponenten hat, wären solche Tests nur über umfassendes und aufwändiges *Mocking* möglich.[41] Der Evaluationsworkflow hingegen, bei dem verschiedene mathematische Metriken implementiert werden müssen, lässt sich hervorragend mittels Unittests prüfen.

Für den Evaluationscode soll auch die Testabdeckung gemessen werden. Absolute Ziele (z.B. *«95% der Codezeilen durch Testfälle abgedeckt»*) sind dabei wenig hilfreich, da man eine hohe Codeabdeckung auch erreichen kann, ohne die Ergebnisse zu überprüfen, die vom jeweiligen Code zurückgeliefert werden. Vielmehr geht es beim Messen der Testabdeckung darum, auf Codepassagen aufmerksam zu werden, die nicht automatisiert getestet werden.

4.2.2 Q2: automatisiert und manuell

Im zweiten Quadranten geht es um Tests, die manuell durchgeführt, bei Bedarf aber auch automatisiert werden können.

Das Vorgehen im Wirtschaftsprojekt im letzten Herbst hat sich dabei als sehr sinnvoll erwiesen: Tests werden zunächst manuell durchgeführt. Erst wenn man bemerkt, dass ein und derselbe Test immer wieder durchgeführt werden muss – und das Testen die Hän-

[40] Mit den Einfluss der Werkzeuge auf das Denken hat sich u.a. Edsger W. Dijkstra befasst (Dijkstra, 2001). Das Zitat, *«If your only tool is a hammer, then every problem looks like a nail.»*, dessen Ursprung umstritten ist (O'Toole, 2014), veranschaulicht das Problem pointiert.

[41] Beim Mocking wird häufig die interne Funktionsweise des Testcodes simuliert. Hierzu ist nicht nur Kenntnis der Schnittstelle, sondern auch der Implementierung vonnöten. Unittests, die Mocking verwenden, sind daher häufig fragil – und strenggenommen keine Black-Box-Tests. Mocking, bzw. dessen Notwendigkeit für das Schreiben von Unittests, gilt daher teilweise als *Code Smell* (Elliot, 2020, S. 214).

de stärker beansprucht als den Kopf –, wird der Test mithilfe eines Testskripts automatisiert. So leistet man nur dort Automatisierungsarbeit, wo diese wirklich sinnvoll ist.

Im vorliegenden Projekt könnten folgende Arten von Tests sinnvollerweise automatisiert werden:

Tests der einzelnen Modelle Da die Erstellung der Modelle nicht Bestandteil der Arbeit ist, werden die Modelle nicht auf Performance im Sinne «korrekter» Predictions geprüft. Es geht vielmehr darum zu prüfen, ob das Modell auch von aussen ansprechbar ist und soweit «korrekt» arbeitet, wie es eben erstellt worden ist. Hierzu können Sammlungen von Testdaten erstellt werden, die Inputs zu Outputs mappen. Dies könnte etwa folgendermassen aussehen:

- `body_part`: Ein Bild (Input) wird zu einem String (erkanntes Körperteil) gemappt.
- `joint_detection`: Ein Bild (Input) wird zu zehn anderen Bildern (extrahierte Gelenke) – oder zu einer Fehlermeldung, falls keine Gelenke erkannt worden sind, – gemappt.
- `ratingen_score`: Ein kleines Bild von einem Gelenk, das aus einer Röntgenaufnahme extrahiert worden ist, wird zu einer Score gemappt.

Tests über das ganze System Hierbei können die Testdefinitionen der einzelnen Modelle wiederverwendet werden, indem etwa ein ganzes Röntgenbild (Input) zu einer Reihe von Scores (oder zu einer Fehlermeldung) gemappt wird.

Für diese Testdefinitionen ist ein sinnvolles Format zu definieren. Eine Möglichkeit wäre JSON-Dateien, die einen Dateinamen zu einem Output mappen. Diese JSON-Dateien könnten statt Dateinamen auch die eigentlichen Bilddaten enthalten, indem diese base64-kodiert direkt eingefügt werden. Beim ersten Ansatz wird das manuelle Betrachten von Testinputs einfacher. Im zweiten Ansatz sind die Testdaten schöner abgekapselt.

In jedem Fall muss ein Testrunner entwickelt werden, der mit den definierten Testdaten umgehen kann. Je nach verwendeter Schnittstelle sieht ein solcher Testrunner anders aus. Mit dem Python-Testframework PyTest und der Testing-Library von Go konnten dabei schon viele positive Erfahrungen gesammelt werden.

Um die nebenläufige Verarbeitung von Röntgenbildern testen zu können, soll ein (nichtdeterministischer) Testfall erstellt werden, der mehrere Röntgenbilder schnell nacheinander zur Verarbeitung übergibt, ohne die eintreffenden Antworten synchron abzuwarten. Diese werden nebenläufig eingesammelt. Die Antworten sind darauf zu prüfen, dass

sie immer die gleichen Scores pro Röntgenbild enthalten. So kann (mit hoher Wahrscheinlichkeit) sichergestellt werden, dass der `orchestrator` die einzelnen Vorgänge intern korrekt auseinanderhält.

4.2.3 Q3: manuell

Der dritte Quadrant spielt im Rahmen der vorliegenden Arbeit nur eine sehr untergeordnete Rolle. Da die grafische Benutzeroberfläche, die für den Prototyp erstellt wird, nur zu Demozwecken dienen soll und kein eigentliches Lieferobjekt des vorliegenden Projekts ist, erübrigen sich formalisierte Tests desselben.

Manuelle Tests könnten sich aber an andere Stakeholder richten und sich auf die Umgebung des Prototyps beziehen.

- Wie kann die Umgebung in Betrieb genommen werden?
- Wie können die Testfälle ausgeführt werden?
- Wie finde ich Hilfe (Dokumentation), wenn etwas nicht (mehr) funktioniert?

Rückmeldungen in diesem Bereich sind gerade im Hinblick auf die Abgabe des Projekts sinnvoll, und könnten vom Auftraggeber und vom Betreuer eingeholt werden.

4.2.4 Q4: Tools

Im vierten Quadranten geht es v.a. um das Testen nicht-funktionaler Anforderungen, d.h. Performance, Security, Usability usw. Da es im vorliegenden Projekt nur um die Entwicklung eines Prototyps geht, können Überlegungen zu den Aspekten Security und Usability von Beginn weg weitgehend ausgeklammert worden.

Die Verarbeitung von Röntgenbildern kann mit einem Benchmark im Bezug auf die Laufzeitperformance untersucht werden. Go bietet hierfür passende Funktionalität in der Standardbibliothek (Donovan & Kernighan, 2015, Kapitel 11.4).

Flaschenhälse bei der Performance (CPU, Memory, I/O) könnten per Profiling ermittelt werden. Eine umfassende Analyse des Laufzeitverhaltens mittels Profiling ist jedoch aufwändig und soll nicht im Rahmen des vorliegenden Projekts erfolgen.

Für die Codequalität ist es sinnvoll, wenn statische Codeanalyse verwendet wird. Sogenannte Linter führen auf Basis von Regeln und Heuristiken Codeüberprüfungen durch und geben Warnungen bei potenziellen Problemen ab, die meist stilistischer Natur sind.

Mögliche Werkzeute sind `pylint` (Python) und `golint` (Go). Code-Auditing-Tools wie `pylama` (Python) und `go vet` (Go) warnen den Programmierer vor möglichen Problemen und Fehlern, und sollten darum auch eingesetzt werden, idealerweise automatisiert und in der Entwicklungsumgebung integriert.

Bei der Verwendung von Tools ist es wichtig, dass diese einfach eingesetzt werden können. Im Wirtschaftsprojekt hat sich ein einfaches `Makefile` als gute Variante erwiesen. Mit Befehlen wie `make cover` wurden etwa alle Unittests ausgeführt und die Codeabdeckung ausgegeben. Müsste man den ganzen Befehl dafür jeweils von Hand eintippen oder in der Shell-History suchen, wäre der Aufruf entsprechend umständlicher.

4.2.5 Testumgebung

Wie der zu erstellende Prototyp sollen die Tests zunächst nur auf einer Test- bzw. Demoumgebung ausführbar sein. Bei Seantis wird die CI-Umgebung Buildkite[42] verwendet, die sich nur mit grösserem Aufwand für neue Projekte einrichten lässt, zumal die erstellten Container für die Modellkomponenten dem CI-System über das interne Container-Buildtool `parts` zugänglich gemacht werden müssten.

Die Integration in die CI-Umgebung des Auftraggebers soll darum nicht Bestandteil des vorliegenden Projekts sein.

[42] `https://buildkite.com/` (abgerufen am 17.05.2020)

5 Realisierung

Beschreiben die bisherigen Kapitel, wie der Prototyp aufgebaut werden *soll*, geht es in diesem Kapitel darum, was tatsächlich umgesetzt worden *ist*. Der Prototyp basiert auf der Architekturvariante, die in 3.1.4 *Variante 4: Messaging, synchron und asynchon*, Seite 39 beschrieben worden ist.

Als Messaging-Lösung kommt RabbitMQ mit dem AMQP-Protokoll zum Einsatz (siehe 8.2 *Wahl der Message-Queue*, Seite 104).

Die Quellcodeartefakte und Modelldaten befinden sich im Abgabeordner (`Zusatz/`) im Unterverzeichnis `deepxray/`.

5.1 Modellkomponenten

Die bestehenden Machine-Learning-Modelle wurden im Abschnitt 2.1 *Bestehende Modelle*, Seite 14 in ihrer Funktionsweise grob beschrieben. Die eigentlichen Modelle werden um weiteren Code ergänzt, wodurch diese mit der Aussenwelt interagieren können. Die Kombination aus den eigentlichen Modellen und dem zusätzlichen Code zur Interaktion mit der Aussenwelt – der *Integrationscode* – wird als *Modellkomponente* bezeichnet. Für Komponente und Modell werden die gleichen Bezeichnungen verwendet; ob das Modell oder die Modellkomponente gemeint ist, ergibt sich dabei aus dem Kontext. Sämtliche Modellkomponenten sind in Python geschrieben (siehe 8.1.1 *Modellkomponenten* `body_part`, `joint_detection`, `ratingen_score`, Seite 102).

Die Modellkomponenten wurden alle nach dem gleichen Schema aufgebaut, wobei die Extraktion der Gelenke (`joint_detection`) aufgrund der zehn verschiedenen Modelle etwas aus der Reihe tanzt.

Für jede Modellkomponente gibt es eine *Hauptklasse*, die den jeweiligen Einstiegspunkt (Hauptprogramm) für die Komponente darstellt. Dazu kommt eine *Predictor*-Klasse, die das Machine-Learning-Modell kapselt und ein Interface für dieses anbietet. Die Hauptklasse kümmert sich um die Kommunikation mit der Message-Queue. Eingehende Nachrichten werden für den Predictor aufbereitet. Es folgt der Aufruf des Predictors, der das Ergebnis an die Hauptklasse zurückliefert. Hier wird die Antwort interpretiert (Erfolg oder Fehler) und an die entsprechende Message-Queue weitergeleitet.

Innerhalb der Modellkomponenten erfolgt die Abarbeitung synchron: Von der eingehenden Queue wird jeweils nur eine Message geholt und verarbeitet. Erst dann wird die nächste Message entgegengenommen und wiederum verarbeitet. Die parallele Verarbeitung erfordert somit, dass mehrere Instanzen pro Modellkomponente am Laufen sind.

5.1.1 Aufbau der Modellkomponenten

Diese verallgemeinerte Struktur wird im Klassendiagramm auf Abbildung 13, Seite 61 veranschaulicht. Die Hauptklasse (`Main`) verfügt jeweils über eine Verbindung mit dem Message-Broker (`connection`) und einen Kanal (`channel`), der über diese Verbindung kommuniziert. Eine `Predictor`-Instanz wird im Konstruktor der `Main`-Klasse erzeugt. Im Konstruktor (`__init__()`) der `Predictor`-Klasse wird das eigentliche Modell geladen. Hierzu wird ein Modellpfad benötigt, den die `Main`-Klasse aus einer Umgebungsvariablen liest.

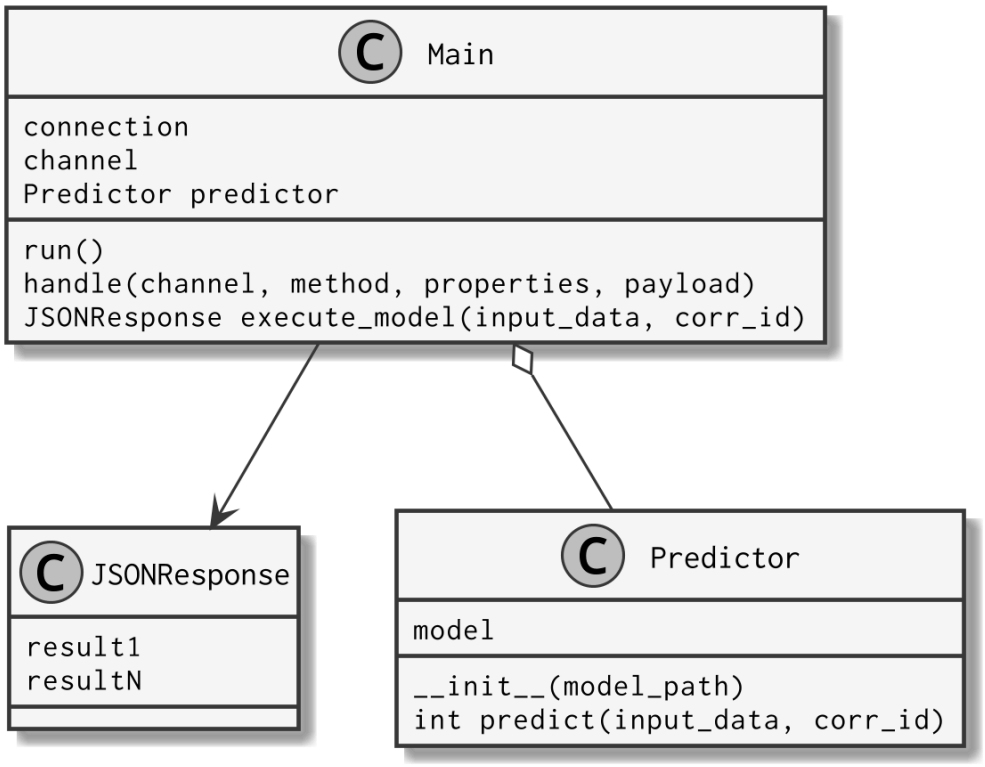


Abbildung 13: Die allgemeine Architektur der Modellkomponenten, bestehend aus einer Hauptklasse und einer Predictor-Klasse. (Klassendiagramm)

Die `run()`-Methode wird vom Hauptprogramm aufgerufen, sobald die Klasse instanziiert ist. Diese stellt sicher, dass alle benötigten Queues vorhanden sind, registriert die Callback-Methode für eingehende Nachrichten (`handle()`), und lässt anschliessend auf eingehende Nachrichten warten.

Die `handle()`-Methode wird automatisch aufgerufen, sobald eine Nachricht aus der Queue eintrifft. Diese hat per API-Definition vier Parameter: Der `channel` wird dazu be-

nötigt, um Nachrichten aus dem Callback heraus zu publizieren (etwa um Antworten zu senden). Der Parameter `method` ist ein AMQP-Delivery-Object, welches hier dazu dient, die Verarbeitung der Nachricht zu bestätigen, um darauf die nächste Nachricht zu erhalten. Der `properties`-Parameter enthält verschiedene Informationen, wovon hier nur der Correlation Identifier (`corr_id`) relevant ist. Dieser wird benötigt, um ausgehende Nachrichten dem gleichen Vorgang zuzuordnen wie die eingegangene Nachricht.[43] Der `body`-Parameter schliesslich enthält den Payload der Nachricht, sprich die eigentlichen Nutzdaten.

Beim Payload handelt es sich meistens um eine JSON-Datenstruktur. Diese enthält ein Bild (ganzes Röntgenbild oder nur einen einzelnen Gelenkausschnitt davon) und im Falle der Extraktion und des Scorings noch eine Gelenkbezeichnung. Das Bild ist jeweils als base64-kodierte Zeichenkette abgelegt.[44] Das Bild wird dekodiert und – wenn nötig mit Zusatzinformationen; zwecks Logging immer mit dem Correlation Identifier (`corr_id`) – an die `execute_model()`-Methode übergeben. Diese kapselt den Aufruf der `predict()`-Methode der `Predictor`-Instanz.

Die `predict()`-Methode führt die Prediction auf Basis des Modells aus, wovor teilweise noch einzelne Verarbeitungsschritte zu erfolgen haben (*Preprocessing*). Die ermittelte Prediction (der Output des Modells) wird von der Methode zurückgeliefert. Dieser wird von der `execute_model()`-Methode in eine JSON-Struktur (im Klassendiagramm als `JSONResponse` bezeichnet[45]) verpackt und an die `handle()`-Methode zurückgegeben, welche diese wiederum als Payload an die Ergebnis-Queue weitergibt. Tritt in diesem Prozess ein Fehler auf, wird von `predict()` eine entsprechender Exception geworfen, der erst von `handle()` abgefangen und als Fehlermeldung an die Fehler-Queue weitergereicht wird.

[43] Das Konzept des Correlation Identifiers wird in Abschnitt 3.1.3 *Variante 3: Messaging zwischen Modellkomponenten*, Seite 36 erklärt.

[44] Dieses Austauschformat wurde aufgrund früherer positiver Erfahrung in verschiedenen Projekten gewählt, ohne dass hierzu ausführlich Alternativen geprüft worden sind. *Protocol Buffers* dürften aufgrund des binären Übertragungsformats eine höhere Effizienz ermöglichen, jedoch auch eine höhere Komplexität – eine zusätzliche Notation und einen weiteren Build-Schritt – mit sich bringen (*Protocol Buffer Basics: Go*, 2020). Zudem erschien es dem Autor dieser Arbeit vernünftiger, die Anzahl der für ihn neuen Technologien und Konzepte – Machine Learning, Messaging via AMQP – innerhalb eines Projekts nicht weiter auszudehnen. Die Codierung, Übertragung und Dekodierung der Bilder stellt auch bei Weitem nicht den Flaschenhals in der Verarbeitung dar, wodurch der Einsatz von Protocol Buffers wohl kaum spürbare Performanceauswirkungen auf das Gesamtsystem hätte.

[45] Diese Klasse ist rein konzeptionell zu verstehen. Tatsächlich findet sich in keiner der drei Modellkomponenten eine Python-Klasse dieses Namens. Es handelt sich vielmehr jeweils um ein Dictionary (`dict`), das die aufgelisteten Eigenschaften enthält. Eine Implementierung in Java würde hier tendenziell eine Klasse verwenden, bei Go dürfte eine Struktur zum Einsatz kommen.

Dieser Prozess beschreibt die Arbeitsweise der Modellkomponenten im Allgemeinen. In den folgenden Abschnitten für die einzelnen Modellkomponenten wird nur noch auf Erweiterungen und Abweichungen dieses allgemeinen Schemas eingegangen. Auf weitere gemeinsame Aspekte, die eher technischer Natur sind, wird im Abschnitt 5.1.5 *Gemeinsame technische Aspekte*, Seite 67 eingegangen, der als Anhang zu diesem Unterkapitel zu lesen ist.

5.1.2 Modellkomponente `body_part`

Das Erkennen eines Körperteils ist der erste Schritt in der Verarbeitung eines Röntgenbildes, bei dem ein Machine-Learning-Modell zum Einsatz kommt. Das Klassendiagramm dieser Komponente (siehe Abbildung 14, Seite 63) entspricht dabei weitgehend demjenigen der generischen Modellkomponente.

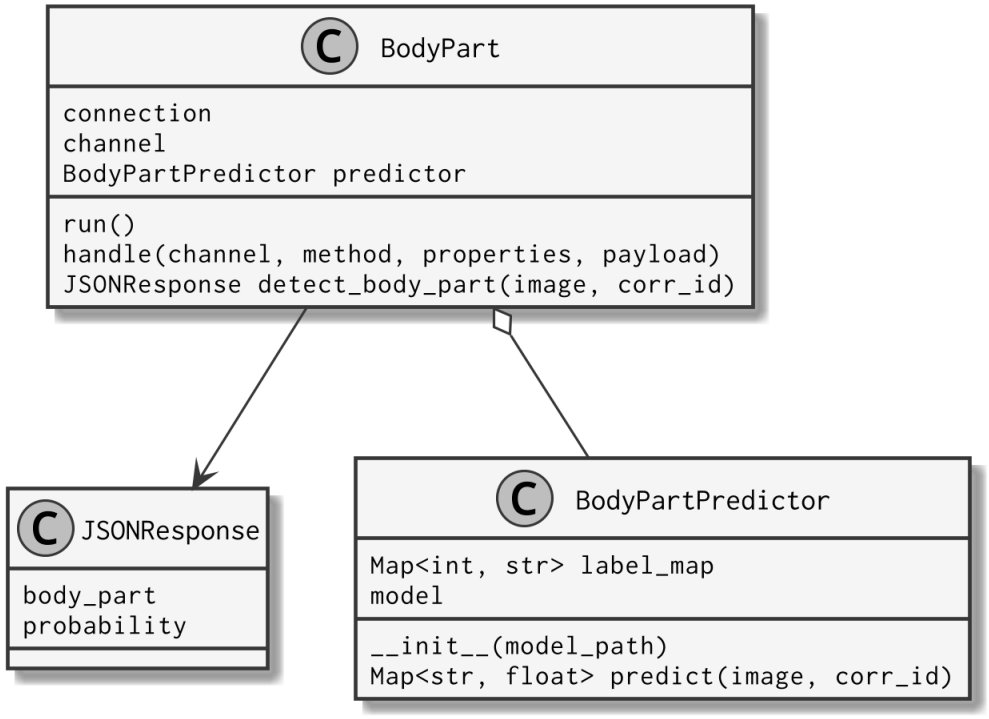


Abbildung 14: Die Modellkomponente `body_part` entspricht in ihrer Klassenstruktur weitgehend der generischen Modellkomponente. (Klassendiagramm)

Die Unterschiede zur generischen Komponente liegen bei Input und Output von `body_part`. Auf einem Röntgenbild (Input) wird erkannt, um welches Körperteil es sich mit

welcher Wahrscheinlichkeit handelt. Die Körperteile werden mithilfe der `label_map` in `BodyPartPredictor` zu den numerischen Outputs gemappt (das Label `hand left` hat den Output 6). Die `predict()`-Methode gibt eine Map von Körperteilen mit Wahrscheinlichkeiten zurück. In `detect_body_part()` wird dann das Körperteil mit der höchsten erzielten Wahrscheinlichkeit ausgewählt. Körperteil und Wahrscheinlichkeit werden zurückgegeben, damit letztere gegen eine vordefinierte Schwelle (z.B. mindestens 50% Wahrscheinlichkeit) geprüft werden könnte.

Beide Informationen – erkanntes Körperteil und Wahrscheinlichkeit – werden mit der `JSONResponse` an den Client zurückgeliefert, damit dieser eine allfällige Schwellenwertprüfung selber durchführen könnte.

Die `run()`-Methode der `BodyPart`-Klasse konsumiert Nachrichten von einer Queue namens `body_part`. Als Payload werden rohe JPEG-Daten (eine Reihe von Bytes) erwartet; eine Einbettung in JSON mit base64-Kodierung entfällt hier. Resultate werden auf einer Queue namens `body_part_response` publiziert.

Die Klasse `BodyPartPredictor` wird mit dem Integrationstest (`test_body_part_predictor.py`) im `tests/`-Unterverzeichnis getestet. Dieser prüft für verschiedene Röntgenbilder einer linken Hand, ob diese korrekt als das wahrscheinlichste Körperteil erkannt wird, und dass die Wahrscheinlichkeit in einem Bereich zwischen 0 und 1 liegt (Plausibilität). Für eine Röntgenaufnahme eines linken Fusses wird geprüft, dass auf dem Bild *nicht* eine linke Hand erkannt wird.

5.1.3 Modellkomponente `joint_detection`

Die Extraktion der Gelenke ist der aufwändigste Arbeitsschritt in der Verarbeitung von Röntgenbildern. Für jedes der zehn relevanten Gelenke (MCP 1-5, PIP 1-5) wird ein separates Modell benötigt. Eine `Predictor`-Klasse, die ein Modell lädt und verwendet, genügt bei der Modellkomponente `joint_detection` nicht den Anforderungen.

Die einzelnen Modelle werden daher nicht über die `Predictor`-Klasse (`JointDetectionPredictor`) abstrahiert, sondern über eine gesonderte `Model`-Klasse namens `JointDetectionModel`. Die `Predictor`-Klasse verfügt intern über eine Datentstruktur (ein Dictionary namens `models`), die eine Gelenkbezeichnung (`pip1`, `mcp3` usw.) einer Instanz von `JointDetectionModel` zuordnet. Für die eigentliche Prediction wird in der `predict()`-Methode von `JointDetectionPredictor` anhand des Gelenknamens die entsprechende `Model`-Klasse aus dem `models`-Dictionary gelesen, worauf anschliessend die `predict()`-Methode der jeweiligen `JointDetectionModel`-Klasse mit dem Röntgenbild aufgerufen werden kann. (Der Gelenkname ist durch den Kontext gegeben und wird über den Konstruktor festge-

legt.) Diese Klassenstruktur ist auf Abbildung 15, Seite 65 zu sehen.

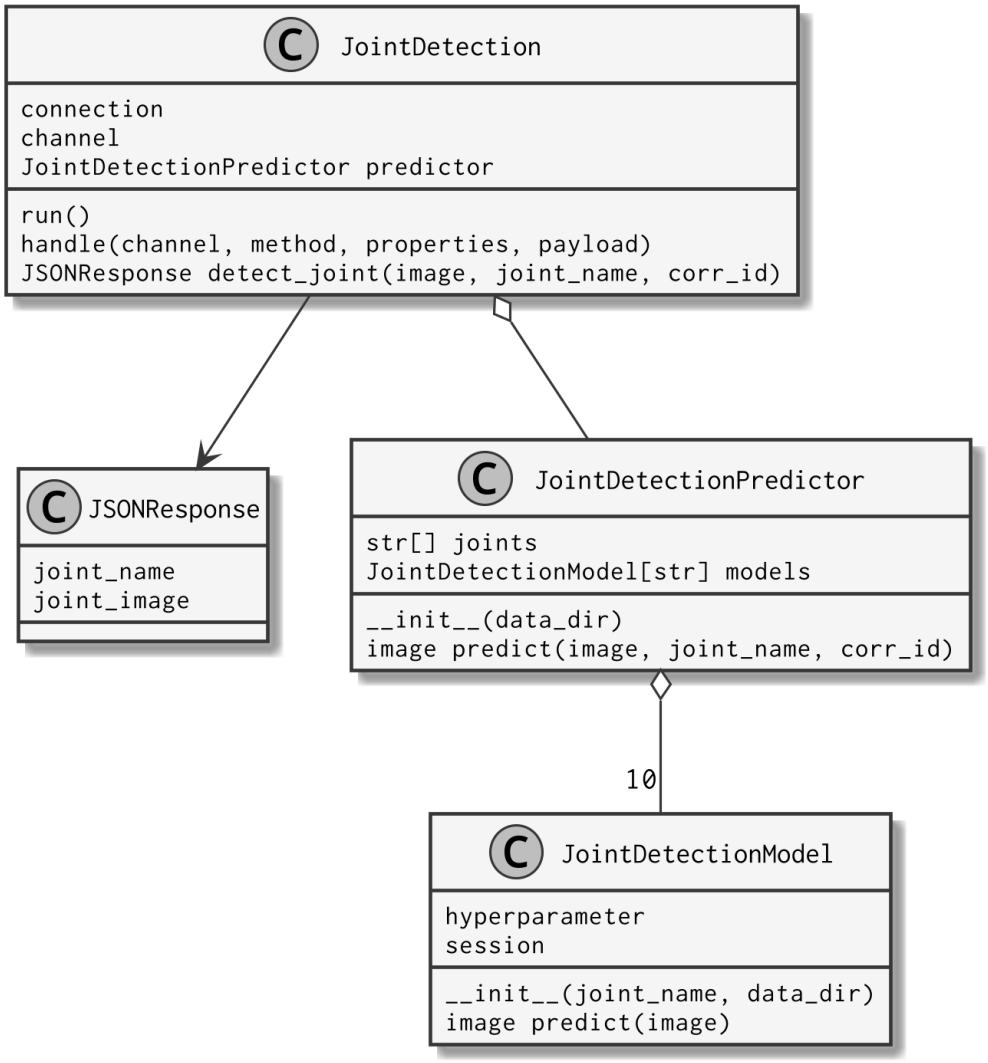


Abbildung 15: Die Modellkomponente `joint_detection` verwendet zehn verschiedene Modelle und abstrahiert diese über eine zusätzliche Klasse namens `JointDetectionModel`. (Klassendiagramm)

Die Modellkomponente `joint_detection` wird nicht nur einmal für alle zehn Gelenke aufgerufen (bzw. über eine entsprechende Message über die Queue `joint_detection` angestossen), sondern einmal pro Gelenk (siehe 3.1.4 *Variante 4: Messaging, synchron und asynchon*, Seite 39). Darum dient als Input nicht nur ein Röntgenbild, sondern auch die Bezeichnung des zu extrahierende Gelenks. Der Payload ist eine JSON-Datenstruktur, die

somit aus den Feldern `joint_name` (Zeichenkette der Länge 4) und `xray` (base64-kodiertes JPEG-Bild) besteht.

Der Output des Modells ist wiederum ein JPEG-Bild, das base64-kodiert in der `JSONResponse` im Feld `joint_image` abgelegt wird. Der Name des Gelenks (Feld `joint_name`) wird auch in diese Datenstruktur eingebettet, da diese Information im nächsten Verarbeitungsschritt benötigt wird. So wird das Ergebnis nicht an den Aufrufer zurückgeliefert, sondern an die Queue `ratingen_score` weitergeschickt. Allfällige Fehler werden in die Queue `joint_detection_error` geschrieben (JSON-Datenstruktur mit den Feldern `error` für die Fehlermeldung und `joint_name` für die Gelenkbezeichnung).

Die Gelenkbezeichnung muss in jedem Fall – Fehler oder Erfolg – weitergegeben werden, da der `orchestrator` seinen Client erst bedienen kann, wenn er für jedes Gelenk eine Nachricht erhalten hat (siehe 3.1.4 *Variante 4: Messaging, synchron und asynchon*, Seite 39).

Die Klasse `JointDetectionPredictor` wird mit dem Integrationstest (`test_joint_detection_predictor.py`) im `tests/`-Unterverzeichnis getestet, der alle zehn Modelle für verschiedene Röntgenbilder ausführt, und prüft, ob jeweils ein Bildausschnitt zurückgeliefert wird.

5.1.4 Modellkomponente `ratingen_score`

Das Scoring der Gelenke erfordert wiederum nur die Ausführung eines einzigen Machine-Learning-Modells, das mit allen zehn Gelenken gleichermassen umgehen kann. Der Parameter `joint_name` ist für die eigentliche Prediction irrelevant, wird aber zwecks Logging trotzdem durch verschiedene Methoden gereicht. Auch im Scoring-Ergebnis (`JSONResponse`) muss das Gelenk vermerkt sein, damit der `orchestrator` das betreffende Gelenk beim Eintreffen der Message für den jeweiligen Vorgang als erledigt markieren kann. Die Klassesnstruktur ist auf Abbildung 16, Seite 67 zu sehen.

Das Modell hinter dem `RatingenScorePredictor` erwartet das Bild eines Gelenks im JPEG-Format als Input, das im JSON-Payload über das Feld `joint_image` base64-kodiert mitgegeben wird. (Das Feld `joint_name` im JSON-Payload bezeichnet das Gelenk.) Der Output des Modells ist eine Score im Wertebereich von 0 bis 5 (siehe 2.1.3 *Scoring von Gelenken:* `ratingen_score`, Seite 16 zur Bedeutung dieser Scores).

Die Inputs werden von der Queue `ratingen_score` gelesen, die Ergebnisse auf die Queue `scores` (im Erfolgsfall, mit Gelenkname und Score) bzw. `ratingen_score_error` (im Fehlerfall, mit Gelenkname und Fehlermeldung) geschrieben.

Die Klasse `RatingenScorePredictor` wird mit dem Integrationstest (`test_ratingen_score_predictor.py`) im `tests/`-Unterverzeichnis getestet, der zehn zuvor eigenhändig

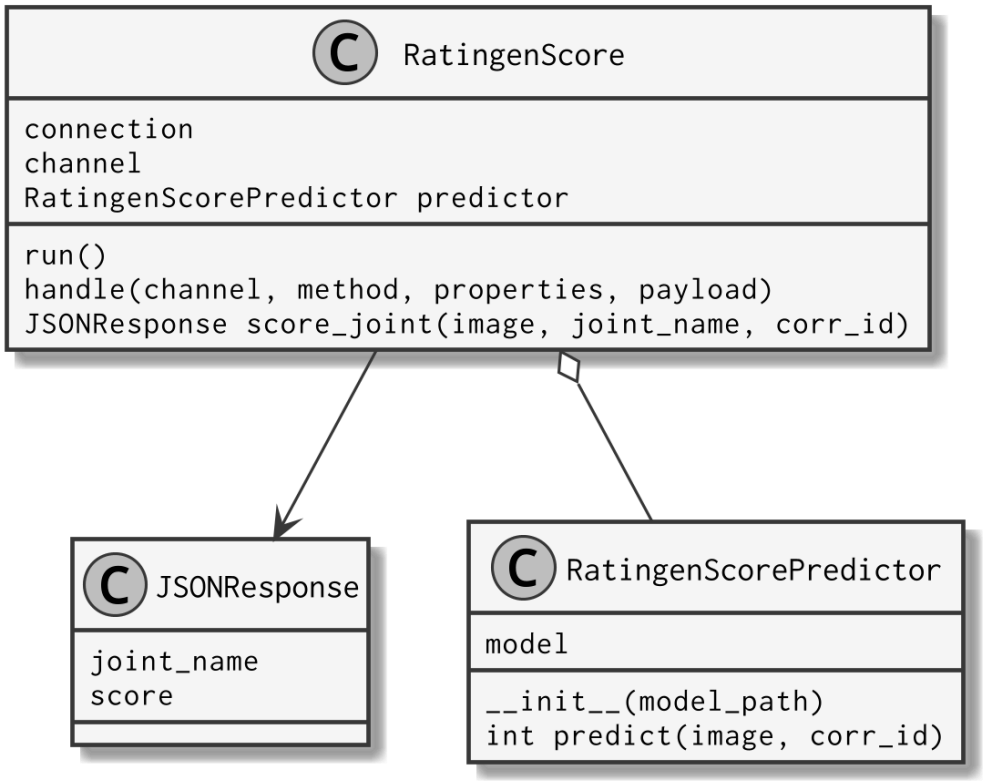


Abbildung 16: Die Modellkomponente `ratingen_score` entspricht (ähnlich wie `body_part` in ihrer Klassenstruktur weitgehend der generischen Modellkomponente. (Klassendiagramm)

ausgeschnittene Bilder von Gelenken einem Scoring unterzieht, und das Ergebnis auf Plausibilität (Wertebereich 0 bis 5) prüft.

5.1.5 Gemeinsame technische Aspekte

Die strukturellen Gemeinsamkeiten der drei Modellkomponenten wurden im Abschnitt 5.1.1 *Aufbau der Modellkomponenten*, Seite 61 erörtert. Hier soll es nachgelagert um die *technischen* Gemeinsamkeiten der erörterten Modellkomponenten gehen. Dies sind:

Container Die Modellkomponenten werden in Docker-Containern ausgeführt (siehe 3.2 *Austauschbarkeit von Modellen*, Seite 42). Die Images, die jeweils im `Dockerfile` der jeweiligen Modellkomponente definiert sind, unterscheiden sich dabei voneinander. Dies hat einerseits mit den unterschiedlichen Python-Versionen (siehe 8.1.1 *Modellkomponenten* `body_part`, `joint_detection`, `ratingen_score`, Seite 102), andererseits

mit den komponentenspezifischen Details (Modelldaten, Abhängigkeiten der verwendeten Libraries/Frameworks in `requirements.txt` und der Code-Struktur) zu tun. Gemeinsam haben alle Images, dass sie auf einem offiziellen Python-Image basieren, die Modelldaten und den Quellcode in zwei verschiedene Verzeichnisse kopieren, alle Abhängigkeiten automatisch aus der Datei `requirements.txt` installieren, und die Hauptklasse des jeweiligen Modells ausführen.

Testing Die verschiedenen Testfälle (Integrationstests) wurden bereits im Kontext jeder Modellkomponente kurz erläutert. Da sich die Modelle nur in einem Container ausführen lassen, müssen auch die Tests in einem Container laufen. Hierzu wird jeweils im `tests/`-Unterverzeichnis jeder Modellkomponente ein weiteres Image definiert, das auf dem Image der Modellkomponente basiert. Für dieses Image wird jeweils das Testing-Framework `pytest` installiert, der Testcode in den Container kopiert und anschliessend mit `pytest` ausgeführt. Das Erstellen des Images und Ausführen des Testcontainers erfolgt am einfachsten mit dem Shell-Skript `test.sh`, das sich im jeweiligen `tests/`-Verzeichnis befindet.

Logging Alle Modellkomponenten geben Log-Meldungen aus. Diese enthalten jeweils den Correlation Identifier am Ende jeder Logzeile in eckigen Klammern. Es wird auf dem Level `Debug` (Meldungen zum aktuellen Verarbeitungsschritt) und `Error` (Fehlermeldungen) geloggt. Als Ausgabe dient `stdout`, und nicht wie üblich `stderr`, da bei der Ausführung mit `docker-compose` standardmässig nur Meldungen von der Standardausgabe angezeigt werden. Der Python-Interpreter wird jeweils mit dem Flag `-u` (unbuffered) gestartet, sodass Log-Meldungen unverzüglich ausgegeben und nicht in einem Puffer zwischengespeichert werden. Auch die Art der Log-Meldungen ist bei allen Modellkomponenten ähnlich, so werden folgende Ereignisse geloggt: Verbindung mit dem Message-Broker (Erfolg/Misserfolg nach `n` Versuchen), Laden des Modells (mit der dazu benötigten Zeit), Eintreffen eines Payloads (mit dessen Grösse), abgeschlossene Prediction (mit Ergebnis und dazu benötigter Zeit); und allfällige Exceptions (als Fehler). Jede Log-Meldung enthält weiter den genauen Zeitpunkt, das Log-Level und den Namen der Klasse, welche die Log-Meldung ausgegeben hat.

Verbindungsaufbau Die Modellkomponenten sind erst voll einsatzfähig, wenn der von ihnen verwendete Message-Broker (RabbitMQ) aufgestartet ist, und sie sich mit diesem verbunden haben. Die Modellkomponenten sind somit vom Message-Broker abhängig, und ihre Container sollten idealerweise erst dann aufgestartet werden,

wenn RabbitMQ bereit ist. Leider verfügt `docker-compose` über keinen Mechanismus, der diese Abhängigkeit zuverlässig abbilden könnte.[46] Aus diesem Grund kann eine Modellkomponente nicht wissen, ob RabbitMQ schon bereit für eingehende Verbindungen ist. Eine einfache Lösung ist der wiederholte Versuch eines Verbindungsaufbaus mit einem *Backoff*, wobei nach jedem gescheiterten Versuch ein bestimmtes Zeitfenster abgewartet wird, bevor ein neuer Verbindungsaufbau versucht wird. Diese Versuche werden in einem Takt von zwei Sekunden unternommen, insgesamt maximal 60 mal. Kann während dieser Zeit keine Verbindung hergestellt werden, dürfte ein Fehler im Gesamtsystem vorliegen, der vor weiteren Verbindungsversuchen zu beheben wäre.

Gemeinsamer Code Obwohl die drei Modellkomponenten über gemeinsame Funktionalität verfügen (Verbindungsaufbau, Logging), wird zwischen ihnen keinerlei Code (zumindest kein selbstgeschriebener) geteilt. Stattdessen wurden die beiden Dateien `amqp.py` und `log.py` (zusammen ca. 50 Zeilen) für die Modellkomponenten jeweils kopiert und wenn nötig angepasst.[47] Dies mag zunächst nach einer schlechten Praxis und einem *Code Smell* klingen, dürfte aber der einzige Weg sein, die *unabhängige* Weiterentwicklung der Modellkomponenten zu ermöglichen. Basierten die Modellkomponenten auf einer gemeinsamen, selbstgeschriebenen Library (für Logging und Verbindungsaufbau), müsste diese für spezielle Anforderungen der Modellkomponenten jeweils angepasst und/oder erweitert werden, was die Vorwärtskompatibilität zu den anderen Modellkomponenten einschränken könnte. Zudem könnten künftige Versionen der Modellkomponenten in einer anderen Programmiersprache als Python geschrieben werden, wodurch die vormals gemeinsame Library nur noch unnötiger Ballast wäre.[48]

Legacy-Code Die Modellkomponenten `body_part` und `joint_detection` – sprich die beiden älteren Modelle – enthalten ein `legacy/`-Verzeichnis. Dieses enthält älteren Code, der grösstenteils ohne Änderung und Reflexion von den Vorprojekten übernommen worden ist. Der Code ist funktionstüchtig, dürfte aber für die Aktualisierung der Modelle auf neuere Frameworks grössere Anpassungen erfordern, und wird im Rahmen der vorliegenden Arbeit nicht analysiert oder dokumentiert.

[46] Mit `depends_on` kann höchstens geprüft werden, ob ein Container bereits *gestartet* worden ist, und nicht, ob die Container-Anwendung bereits fertig geladen und so etwa für eingehende Verbindungen *bereit* ist.

[47] In `ratingen_score` können aufgrund der Python-Version 3.5 keine f-Strings verwendet werden.

[48] Das Go-Proverb *«A little copying is better than a little dependency.»* bringt zum Ausdruck, dass die Wiederverwendung von Code in manchen Fällen nicht erstrebenswert, ja gar schädlich ist (Pike, 2015, 9:28).

Modelldaten Die Modelldaten werden beim Erstellen des Docker-Images jeweils in dieses hineinkopiert[49], genauer in das Verzeichnis `/usr/modeldata`.[50] Das Verzeichnis `/usr/modeldata` wird der Python-Anwendung via Umgebungsvariable `DATADIR` zugänglich gemacht, womit die Modelldaten in die Anwendung geladen werden können.[51]

5.2 Komponente `orchestrator`

Beim `orchestrator` handelt es sich um die zentrale Komponente des Prototyps. Der Prozess beginnt damit, dass ein Client ein Röntgenbild per HTTP an den `/score`-Endpoint (`POST`-Anfrage) sendet, der vom `orchestrator` angeboten wird, und endet damit, dass der `orchestrator` dem Client eine Reihe von Scores zurückliefert, idealerweise zehn an der Zahl. Dieser Vorgang ist in 3.1.4 *Variante 4: Messaging, synchron und asynchon*, Seite 39 beschrieben und auf Abbildung 8, Seite 41 grafisch dargestellt.

Dieser Vorgang dauert einige Sekunden. Währenddessen sollen auch andere Clients Röntgenbilder zur Verarbeitung in Auftrag geben können, ohne auf das Ende eines laufenden Vorgangs warten zu müssen.[52] Noch wichtiger ist, dass die Ergebnisse der beiden Vorgänge nicht durcheinandergebracht werden, sodass etwa Anfrager A, der das Röntgenbild einer gesunden Hand verarbeiten lässt, plötzlich die Score für ein geschädigtes Gelenk, das zu einem Bild von Anfrager B gehört, in seinem Ergebnis vorfinden würde.

Wie die nebenläufige Verarbeitung von Röntgenbildern korrekt und möglichst performant erfolgt, wird in Abschnitt 5.2.1 *Der `/score`-Endpoint*, Seite 74 erörtert. Zunächst soll hier aber ein Überblick über die `orchestrator`-Komponente geboten werden, der zum Verständnis des `/score`-Endpoints nötig ist.

Der `orchestrator` ist in Go (siehe 8.1.2 *Neue Komponente* `orchestrator`, Seite 102) geschrieben, genauer in der derzeit aktuellen Version 1.14 von Go. Das `orchestrator/`-Verzeichnis enthält die folgenden Artefakte:

[49] Das Einhängen (via `mount`) über ein lokales Verzeichnis wäre schneller, würde aber die Manipulation der Modelle zur Laufzeit ermöglichen, und bietet daher eine schlechtere Isolation.

[50] Beim Modell `ratingen_score` wird zusätzlich das generische Modell *Inception V3* nach `/root/.keras/models` kopiert, damit die Datei nicht beim Aufstarten des Containers vom Internet heruntergeladen wird, siehe `https://stackoverflow.com/q/60837438/6763074` (abgerufen am 23.05.2020).

[51] Bei `joint_detection` wird der Pfad zu den Modelldaten zusätzlich über den jeweiligen Gelenknamen ergänzt.

[52] In der Praxis verzögern sich die einzelnen Vorgänge unter starker Last des Systems. Dieses soll aber trotzdem *ansprechbar* bleiben, d.h. die Anfragen zumindest entgegennehmen und in die entsprechenden Queues zur Verarbeitung stellen.

`Dockerfile` Wie die Modellkomponenten läuft auch der `orchestrator` in einem Docker-Container. Aufgrund der kompilierten Natur von Go empfiehlt sich hier der Einsatz zwei verschiedener Container: einer für die Kompilierung, und einer für die Ausführung.[53] Das *Builder*-Image basiert auf einer aktuellen Version des Go-Docker-Images. Sämtliche Quellcodedateien, sowie die Moduldateien (`go.mod` und `go.sum`, siehe weiter unten), werden in das Image kopiert. Die Abhängigkeiten werden mit `go mod tidy` aufgelöst. Anschliessend wird das Programm kompiliert und als ausführbare Binärdatei unter `/app/orchestrator` abgelegt. Das *Runtime*-Image kann diese Datei mittels `COPY --from=builder` aus dem Builder-Image herauskopieren. Der Inhalt des `cert/`-Verzeichnisses (siehe unten) wird ebenfalls in das Runtime-Image kopiert um TLS-verschlüsselte Verbindungen via HTTPS zu unterstützen. Der Port 8080 wird gegen aussen freigegeben, und `/bin/orchestrator` als Standardeinstiegspunkt definiert.

`go.mod`/`go.sum` Bei diesen Dateien handelt es sich um die Moduldefinition. In `go.mod` werden der Name des Moduls (`code.seantis.ch/deepxray/orchestrator`), die verwendete Version von Go (1.14.2) und die verschiedenen Abhängigkeiten (`uuid` und `amqp`) definiert. In `go.sum` werden die Prüfsummen für die initial heruntergeladenen Abhängigkeiten festgehalten. Darum wird letztere Datei auch in den Build-Container kopiert, damit garantiert ist, dass immer die gleichen Abhängigkeiten verwendet werden. Weitere Details zur Verwendung von Go-Modulen finden sich im offiziellen Blog zu Go (Bui-Palsulich & Compton, 2019).

`orchestrator.go` Diese Datei enthält den Haupteinstiegspunkt, sprich die `main()`-Funktion der `orchestrator`-Komponente. Diese führt folgende Schritte aus: Zunächst wird die Verbindung mit dem Message-Broker RabbitMQ aufgenommen, wozu ein lineares Backoff verwendet wird (siehe Abschnitt *Verbindungsaufbau* in 5.1.5 *Gemeinsame technische Aspekte*, Seite 67). Als nächstes werden sämtliche Queues deklariert, die von RabbitMQ angeboten werden müssen[54]; auch diejenige, die nicht direkt vom `orchestrator` angesprochen wird (`ratingen_score`). Nun wird je ein Channels auf die vier Queues geöffnet, von denen der `orchestrator` Nachrichten empfangen soll: `bodyPartResponse` (Queue `body_part_response`), `jointDetectionError`

[53] Im Gegensatz zu Python, das einen Interpreter zur Ausführung benötigt, kommen kompilierte Go-Programme gänzlich ohne Laufzeitumgebung aus, da die statisch gelinkten Binärdateien sämtliche Abhängigkeiten beinhalten.

[54] Tabelle 6, Seite 106 listet diese in der entsprechenden Spalte auf.

(Queue `joint_detection_error`), `ratingenScoreError` (Queue `ratingen_score_error`) und `scoresResponse` (Queue `scores`). Die Handhabung dieser Channels ist weiter unten (`messaging/messaging.go`) erläutert. Schliesslich werden zwei Endpoints registriert: Erstens, der `/score`-Endpoint, der in Abschnitt 5.2.1 *Der* `/score`*-Endpoint*, Seite 74 erörtert wird; zweitens, ein `/canary`-Endpoint, womit geprüft werden kann, ob der Service läuft und via HTTP ansprechbar ist, und etwa zu Monitoring-Zwecken eingesetzt werden könnte. Die Endpoints werden vie TLS-Verbindung zugänglich gemacht, die über ein selbst signiertes Zertifikat verschlüsselt wird (siehe Abschnitt `cert/` weiter unten).

`httputils/httputils.go` Dieses Untermodul enthält eine Datenstruktur `HTTPErr`, die eine Fehlermeldung (Zeichenkette) mit einem HTTP-Status-Code kombiniert. Diese kann mit der Funktion `RespondHTTPErr()` formatiert an den Client übergeben werden und schreibt die Meldung zugleich mit dem Correlation Identifier als Log-Meldung. Zwei weitere Funktionen – `Respond()` und `RespondMessage()` – bieten eine vergleichbare Funktionalität über eine Schnittstelle, die nicht `HTTPErr` verwenden, sondern atomare Werte (Status-Code als Integer, Message als Zeichenkette). Alle Funktionen haben gemeinsam, dass sie zu einem HTTP-Status-Code die entsprechende textuelle Beschreibung ermitteln und ausgeben. Für den Produktiveinsatz wäre zu prüfen, ob interne Fehlermeldungen dem Client vorenthalten werden sollten.

`messaging/joints.go` Diese Datei definiert eine Map namens `BodyPartJoints`, welche einem Körperteil (`hand left`, siehe Beschreibung der `label_map` in Abschnitt 5.1.2 *Modellkomponente* `body_part`, Seite 63) den zu extrahierenden Gelenken (MCP 1-5 und PIP 1-5) zuordnet. Die Idee hinter dieser Map ist die Vorwärtskompatibilität: So soll ein erkanntes Körperteil nicht einfach statisch gegen `hand left` geprüft, sondern in `BodyPartJoints` nachgeschlagen werden. Um weitere Körperteile (und Gelenke) zu unterstützen, könnte die Map um weitere Definitionen ergänzt werden – sofern die Modellkomponenten entsprechend nachgerüstet werden. Weiter ist in `joints.go` die Funktion `CreateJointDetectionRequests` definiert, die für ein Röntgenbild und eine Körperteilbezeichnung eine Reihe von `JointDetectionRequest`-Strukturen (siehe `payloads.go` weiter unten) erzeugt; eine pro Gelenk. Diese Funktion schlägt die übergebene Körperteilbezeichnung in `BodyPartJoints` nach und reagiert mit einem Fehler, falls für dieses Körperteile keine Gelenke unterstützt werden. Im Erfolgsfall werden die erstellten Strukturen als Slice zurückgeliefert.

`messaging/messaging.go` Diese Datei enthält das Kernstück der `orchestrator`-Komponente: die `Handle()`-Funktion, welche für die eigentliche Orchestrierung zuständig ist. Diese Funktion erwartet die vier Channels auf die Queues, die oben (siehe `orchestrator.go`) deklariert worden sind. Weiter wird hier eine Map namens `connections` verwaltet, die einem Correlation Identifier einen Channel zuordnet. Hierbei handelt es sich um den Kanal, mit dem Nachrichten aus den Queues in den Anfragekontext zurückgeleitet werden können. (Der Mechanismus wird in 5.2.1 *Der `/score`-Endpoint*, Seite 74 aus einer anderen Perspektive erläutert.) Sämtliche asynchrone Kommunikation in der `orchestrator`-Komponente läuft in dieser Funktion zusammen, genauer in einem `for/select`-Loop, der für jeden Channel einen `case`-Branch hat (Donovan & Kernighan, 2015, Kapitel 8.7). Neben der Channels auf die vier Message-Queues kümmert sich `Handle()` im gleichen `for/select`-Loop auch um die Registrierung und Deregistrierung von Rückkanälen, die in der `connections`-Map verwaltet werden. Ein Anfragekontext (Aufruf des `/score`-Endpoints) kann sich über den Kanal `connectChan` registrieren. Dazu sendet er eine Nachricht (`ChannelConnection`) bestehend aus einem Correlation Identifier und einem eigens erstellten Channel an `connectChan`. (Der Kanal wird in der `connections`-Map abgelegt.) Ist der Anfragekontext am Ende seiner Lebensdauer angelangt, kann er sich über `disconnectChan` unter Angabe seines Correlation Identifiers wieder deregistrieren. (Der Kanal wird geschlossen und aus der `connections`-Map entfernt). Mithilfe dieses Konstrukts kann die Integrität der nebenläufigen Vorgänge ohne jegliche Locking-Mechanismen wie Mutexes gewährleistet werden.[55] Um Nachrichten aus den vier erwähnten Queues in den Anfragekontext weiterleiten zu können, wird deren Correlation Identifier verwendet. Mit diesem kann der Rückkanal in der `connections`-Map nachgeschlagen werden. Der Einfachheit halber ist der Rückkanal als `byte`-Slice typisiert, sodass darüber beliebige Payloads weitergereicht werden können.[56] Schliesslich enthält `messaging.go` die Funktionen `DeclareQueue()` und `ConsumeQueue()`, womit die Queues mit sinnvollen Einstellungen deklariert und konsumiert werden können.[57]

[55] Das Go-Proverb *«Channels orchestrate; mutexes serialize»* bringt den Vorteil dieses Ansatzes zum Ausdruck (Pike, 2015, 4:21).

[56] Die Art des Payloads ergibt sich dabei aus dem Kontext: So trifft die Nachricht von `body_part` immer früher ein als diejenigen von `ratingen_score`.

[57] Diese Einstellungen müssten für den Produktivbetrieb noch einmal gründlich geprüft werden. Optionen wie die persistente Speicherung von Messages ergeben aber in diesem Kontext keinen Sinn, da ein HTTP-Client kaum auf einen Neustart des Systems warten würde.

`messaging/payloads.go` Hier sind die verschiedenen Payloads als `struct` definiert, die einerseits zwischen den Komponenten untereinander und andererseits zwischen dem `orchestrator` und dem Client ausgetauscht werden. Diese entsprechen weitgehend den Strukturen, die in Tabelle 6, Seite 106 aufgelistet sind. Eine Ausnahme bildet die Struktur `ScoreResponse`, die sowohl für Fehlermeldungen als auch für Scoring-Ergebnisse verwendet werden kann, indem sie die Felder `score` *und* `error` enthält. Der Marshaling-Mechanismus von Go füllt dabei einfach die Felder ab, die im Payload zu finden sind, und lässt das jeweils andere Feld leer. Die Entscheidung, ob es sich bei der Datenstruktur eine Fehlermeldung oder um ein Scoring-Ergebnis handelt, ist der Interpretation des Aufrufers überlassen. Die einzelnen Felder der Strukturen sind zwecks Marshaling mit `json`-Tags annotiert (Donovan & Kernighan, 2015, Kapitel 4.5).

`cert/` Dieses Verzeichnis enthält ein selbst signiertes TLS-Zertifikat, womit die Verbindung zwischen dem `orchestrator` und dem HTTP-Client verschlüsselt wird. Da der Prototyp zu Test- und Demozwecken auf einem Cloud-Server ausgeführt werden soll, und diesem echte, d.h. schützenswerte Röntgenbilder übertragen werden, ist dieser einfache Schutzmechanismus angebracht, wenn auch für die Praxis untauglich.[58] Das `Makefile` im Verzeichnis dient dazu, ein neues Zertifikat über den Befehl `make` zu erstellen. Der Aufruf von `openssl` mit all den benötigten Parametern wird dadurch vereinfacht. Das Zertifikat wird beim Erstellen des Docker-Images in diesen integriert, und beim Aufstarten vom `orchestrator` geladen.

5.2.1 Der `/score`-Endpoint

Der Überblick über den `orchestrator` aus dem vorherigen Abschnitt sollte einen Rahmen schaffen, um die Rolle des `/score`-Endpoints besser verständlich zu machen. Dieser wird in der Funktion `createScoringEndpoint` als Closure (Donovan & Kernighan, 2015, S. 136) erzeugt und zurückgegeben.[59] Diese Closure benötigt als Kontext den AMQP-Channel (`amqpChan`), die Queue zum Zugriff auf die Antworten von `body_part` (`bodyPartResponseQueue`) und die beiden Channels zum Registrieren und Deregistrieren des Rückkanals mit einem Correlation Identifier (`connectChan` und `disconnectChan`).

Die Funktionsweise des `/score`-Endpunkts lässt sich am besten sequenziell erläutern:

[58] HTTP-Clients wie `curl` benötigen einen zusätzlichen Parameter, damit ein nicht verifiziertes TLS-Zertifikat für den Verbindungsaufbau akzeptiert wird.

[59] HTTP-Endpoints werden in Go als Funktionen gehandhabt, die dem Interface `http.Handler` genügen (Donovan & Kernighan, 2015, Kapitel 7.7).

1. Der Client ruft den Endpoint via HTTP (`POST`-Methode) auf. Wird eine andere Methode verwendet, erhält der Client als Antwort Status 405 (Method not Allowed) und der Vorgang wird abgebrochen.

2. Der Payload (JPEG-Röntgenbild als Form File `xray`) wird in der Funktion `loadXRay` ausgelesen. Fehlt dieser, erhält der Client als Antwort Status 400 (Bad Request); der Vorgang wird abgebrochen. Ist er vorhanden, wird das Bild in einen Puffer gelesen und zurückgegeben.

3. Es wird ein Correlation Identifier (eine zufällige UUID) erzeugt.[60] Sollte dies fehlschlagen, erhält der Client Status 500 (Internal Server Error) als Antwort.

4. Der Rückkanal wird als Channel vom Typ `[]byte` erzeugt und mit dem Correlation Identifier zusammen in einer `messaging.ChannelConnection` an den `connectChan` gesendet, um diesen zu registrieren. In einem *deferred call*, der unmittelbar danach definiert ist, wird der Correlation Identifier an den `disconnectChan` gesendet. Dadurch wird der Kanal in jedem Fall deregistriert (und geschlossen), egal über welchen Pfad die Funktion des `/score`-Endpoints verlassen wird (Donovan & Kernighan, 2015, S. 145).

5. Das Röntgenbild wird mit der Funktion `publishToBodyPart` in die Queue `body_part` gestellt. Sollte dies scheitern, wird dem Client Status 500 (Internal Server Error) zurückgeliefert. Andernfalls sollte nun die Erkennung des Körperteils angestossen worden sein.

6. Die Antwort von `body_part` wird synchron vom Rückkanal (`ch`) abgewartet. Trifft sie ein, wird sie in der Funktion `processBodyPartResponse` verarbeitet, indem die Bytes in die entsprechende JSON-Struktur umgewandelt werden (Unmarshaling). Ein Fehler würde wiederum mit Status 500 (Internal Server Error) an den Client geschickt.

7. Für das erkannte Körperteil werden mit `createJointDetectionRequests` die Nachrichten für die Gelenkextraktion erzeugt. Wie weiter oben für `joints.go` beschrieben worden ist, findet hier keine explizite Prüfung auf das Körperteil statt. Sollte es für das Körperteil keine extrahierbaren Gelenke geben, wird Status 501 (Not Implemented) an den Client zurückgeliefert, da das jeweilige Körperteil nicht zur Verarbeitung unterstützt wird. Wurde eine linke Hand auf dem Röntgenbild erkannt,

[60] `https://godoc.org/github.com/google/uuid#NewRandom` (abgerufen am 24.05.2020)

werden zehn Messages für die Gelenkextraktion erzeugt und (mit `toJSON`) in JSON umgewandelt. Das Röntgenbild wird hierzu base64-kodiert.

8. Die (zehn) Messages zur Extraktion werden mit `sendJointDetectionPayloads` auf der Queue `joint_detection` publiziert.

9. Der Payload für die Antwort an den Client wird initialisiert. Dieser enthält eine Map von Scores (`string` zu `int`). Nun beginnt das Warten auf die eintreffenden Nachrichten (Fehlermeldungen oder ermittelte Scoring-Ergebnisse).

10. Bis zu diesem Punkt dürfte der Client weniger als eine Sekunde gewartet haben. Bis alle Ergebnisse beim `orchestrator` eingetroffen sind, und diese an den Client weitergerreicht werden können, dürften nun einige Sekunden vergehen.[61] Bei hoher Last erhöht sich die Wartedauer. Diese soll limitiert werden, indem ein festes Timeout von 60 Sekunden definiert wird. Hierzu wird eine *goroutine* gestartet, die 60 Sekunden abwartet und dann über einen Channel meldet, dass sie abgelaufen ist.

11. Nun wird in einem Loop auf die eintreffenden Antworten gewartet (eine Iteration pro Gelenk). Hierzu dient ein `select`-Statement, das zwei `case`-Branches hat: Einerseits für die Ergebnisse aus dem Rückkanal (Channel `ch`), andererseits vom vorher beschriebenen Timeout-Kanal. Trifft eine Meldung über `ch` ein, wird diese ausgewertet (Fehlermeldung oder Score) und dem Payload für die Client-Response hinzugefügt. Läuft in diesem Loop das Timeout ab, und wird dies über den entsprechenden Kanal gemeldet, erhält der Client den Status 408 (Request Timeout) *ohne* partiellen Payload gemeldet. Der Vorgang wird damit abgebrochen.

12. Konnten hingegen alle (zehn) Antworten «eingesammelt» werden, ohne dass in der Zwischenzeit das Timeout abgelaufen ist, wird der Payload als JSON an den Client gesendet (Status: 200 OK) und geloggt.

Der Vorgang ist hiermit abgeschlossen. (Der Rückkanal wird, wie oben erwähnt, über einen *deferred call* deregistriert und geschlossen.) Während dieses Vorgangs können andere Anfragen auf den `/score`-Endpoint abgesetzt werden. Diese werden jeweils in einem gesonderten Funktionskontext abgearbeitet und haben ihren eigenen Correlation Identifier. Die verschiedenen Vorgänge kommen sich also nicht in die Quere.

[61] Benchmarks auf einem aktuellen Laptop und auf einem Cloud-Server (auf Arch bzw. Debian Linux) zeigen, dass der Client sich ca. acht bis zwölf Sekunden gedulden muss, bis er eine Antwort bekommt.

5.2.2 Testen des `/score`-Endpoints

Wie bereits in der Teststrategie skizziert (siehe 4.2 *Teststrategie*, Seite 55), kann der `/score`-Endpoint bedeutend effektiver von einem Systemtest (End-to-End) als von einer Reihe von Unit- oder Integrationstests abgedeckt werden. Im `test/`-Verzeichnis befinden sich mehrere Varianten dafür:

`test.sh` Hierbei handelt es sich um ein einfaches Bash-Skript, das den `/score`-Endpoint mit vier verschiedenen Röntgenbildern aufruft. Diese zeigen: je eine geschädigte Hand (`bad1.jpg` und `bad2.jpg`), eine gesunde Hand (`good.jpg`) und einen Fuss (`foot-.jpg`). Für die drei Aufrufe mit einer linken Hand wird der zurückgelieferte Payload mit `jq` formatiert und in JSON-Dateien abgelegt. Dieses Testskript nimmt keine weiteren Auswertungen vor und diente vorallem während der Entwicklung zum Testen.

`test.py` Dies ist ein Python-Skript, das mithilfe der `requests`-Library die gleichen Bilder verarbeiten lässt wie das zuvor genannte Bash-Skript. Im Gegensatz dazu wird hier auch der Status-Code der Antwort geprüft (200 bei den Händen, 501 beim Fuss). Das Skript diente dazu, die `requests`-Library auszuprobieren, die bei der systematischen Evaluation (siehe 6.1 *Technische Evaluation*, Seite 80) zum Einsatz kam.

`deepxray_test.go` Dieser Testfall wurde in Go geschrieben und ist der mächtigste der dreien. Die Testfälle sind in einer Map namens `tests` abgelegt und folgen dem Paradigma des *table-driven test design* (Donovan & Kernighan, 2015, S. 306). Als Key dient jeweils der Dateiname eines Röntgenbildes; als Value wird eine Struktur verwendet, die einerseits den erwarteten HTTP-Status-Code (200 oder 501), und andererseits eine Map von erwarteten Scores (`"pip1": 0, "pip2": 1`, usw.) enthält. Die Testfunktion `TestScoringAsync` führt diese Testfälle *asynchron* aus, d.h. startet für jeden Testfall eine eigene *goroutine*. Mithilfe einer `WaitGroup` (Donovan & Kernighan, 2015, S. 238) wird darauf gewartet, dass alle gesendeten Requests beantwortet worden sind. Die Auswertung der Antworten, d.h. der Vergleich zwischen den erwarteten und den eingetroffenen Scores, wird dabei in jeder Request-*goroutine* durchgefürt. So kann geprüft werden, dass bei mehreren nebenläufigen Vorgängen keine Scores durcheinandergeraten. Weiter enthält diese Datei einen Benchmark, der die gleiche Datei (`bad2.jpeg`) zehnmal an den `/score`-Endpoint schickt und anschliessend die durchschnittliche Verarbeitungszeit misst. Der Test kann mit `go test`, der Benchmark (inkl. Test) mit `go test -bench=.` gestartet werden.

5.3 Web-Oberfläche

Um den Prototyp anschaulich demonstrieren zu können, wurde im Verzeichnis `gui/` eine kleine Web-Oberfläche entwickelt. Diese wurde in Go umgesetzt und rendert das HTML serverseitig (Donovan & Kernighan, 2015, Kapitel 4.6). Die Anwendung wird analog zum `orchestrator` in einem Docker-Image kompiliert und in einem zweiten ausgeführt. Die Anwendung ist über Port 8888 via HTTPS erreichbar.

Der Benutzer kann ein Röntgenbild hochladen und mit Betätigung der Schaltfläche *Score* zur Verarbeitung abschicken. Nach einigen Sekunden erscheint links das hochgeladene Röntgenbild. In der mittleren Spalte werden die Scoring-Ergebnisse mit einem entsprechenden Farbcode[62] angezeigt. Rechts ist ein Röntgenbild mit markierten Gelenken als Legende dargestellt, damit der Benutzer Bezeichnungen wie *MCP 1* oder *PIP 3* einfacher dem hochgeladenen und links angezeigten Röntgenbild zuordnen kann.[63]

Auf die Funktionsweise der Web-Oberfläche soll hier nicht weiter eingegangen werden, da diese nur ein Nebenprodukt des Projekts ist.

Die Web-Oberfläche ist auf Abbildung 17, Seite 79 zu sehen.

[62] von grün wie gesund, über gelb wie mittelstark geschädigt, bis rot wie stark geschädigt

[63] Die Gelenke direkt mit farbigen Quadraten auf dem Originalbild zu markieren, wäre ein schönes Feature, das jedoch eine umfassende Erweiterung des Prototyps erfordern würde (siehe 7.3 *Weitere Ideen*, Seite 99).

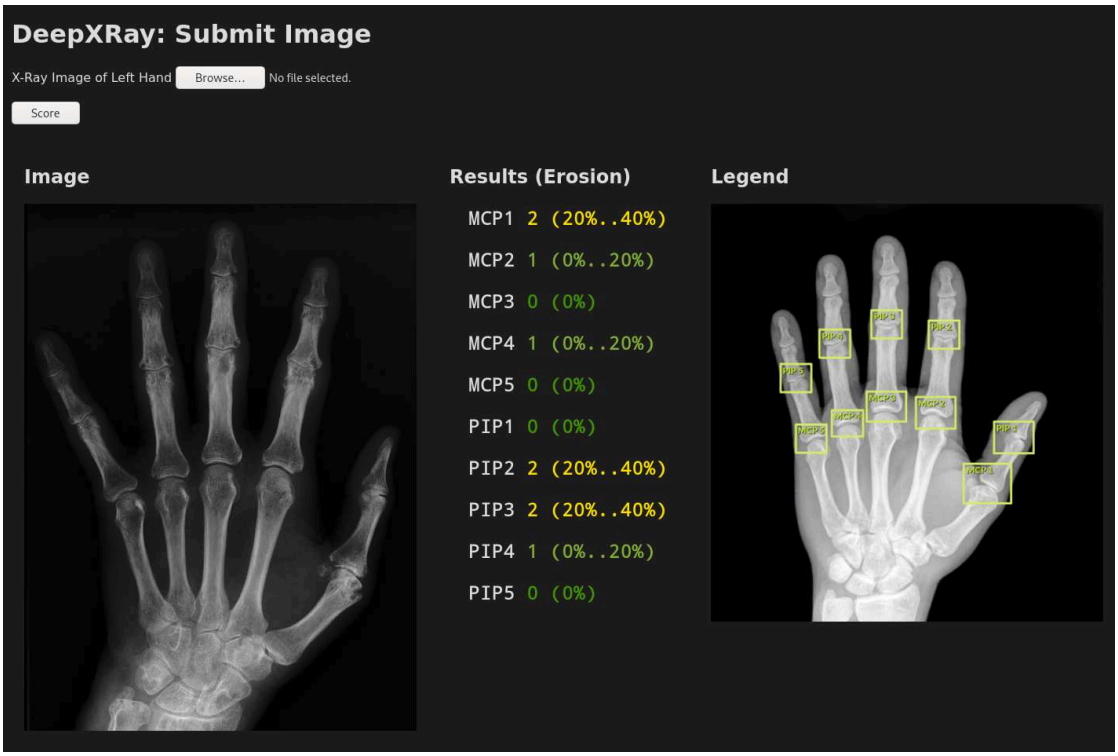


Abbildung 17: Die Web-Oberfläche, auf der links das verarbeitete Röntgenbild (*Case courtesy of Assoc Prof Frank Gaillard, Radiopaedia.org, rID: 3031*), in der Mitte die Scoring-Ergebnisse und rechts eine Legende mit beschrifteten Gelenken zu sehen sind. Die Schädigungen der Gelenke bewegen sich im tiefen bis mittleren Bereich.

6 Evaluation

In diesem Kapitel geht es um die Evaluation der Arbeit. Dies ist im doppelten Sinne zu verstehen: Einerseits soll der erstellte Prototyp anhand verschiedener Metrtiken (siehe 2.4.3 *Mögliche Evaluationsmetriken*, Seite 25) evaluiert werden (technische Evaluation); andererseits soll das Ergebnis der Arbeit in einem übergeordneten Kontext, der auch den vorliegenden Bericht umfasst, im Bezug auf die Zielerreichung (siehe 1.1 *Projektauftrag*, Seite 9) evaluiert und reflektiert werden (Evaluation des Zielerreichung).

6.1 Technische Evaluation

Der erstellte Protoyp kombiniert drei bestehende Modelle bzw. drei Arten von Modellen zu einem Gesamtsystem, das gegen aussen den Eindruck erweckt, als würde nur eine Art von Prediction erstellt: das Scoring der Gelenke. Das Erkennen des Körperteils und die Extraktion der Bildausschnitte, welche die Gelenke zeigen, ist für den Endanwender transparent.

Für die einzelnen Modelle (`body_part`, `joint_detection` und `ratingen_score`) gibt es teilweise bereits Evaluationsmechanismen. Diese wurden im Rahmen der Entwicklung der einzelnen Modelle erstellt. Soll ein einzelnes Modell verbessert werden, ist dieses isoliert im Rahmen dieses Vorgangs zu evaluieren.

Im Rahmen der vorliegenden Arbeit soll es nur um die Evaluation des Gesamtsystems gehen. Wird ein bestehendes Modell verbessert, hat dies Einfluss auf die Performance des Gesamtsystems. Diese Verbesserung soll mit einem Evaluationsworkflow ermittelt werden können. Dieser kann dabei behilflich sein, den Fokus für die Verbesserung bestehender Modelle richtig zu legen, sodass man Aufwand in diejenigen Modelle steckt, deren Verbesserung einen Performancegewinn für das Gesamtsystem bedeuten.[64]

Da die Modelle im laufenden Betrieb nicht weitertrainiert werden, bietet sich eine *Offline-Evaluation* an. Für die Offline-Evaluation eines Modells (bzw. eines Systems von Modellen) gibt es verschiedene Varianten: *Hold-Out Validation*, *Cross-Validation*, *Bootstrapping* (Zheng, 2015, S. 19). Die Hold-Out Validation ist eine einfache Variante, erfordert aber ein zusätzliches Evaluationsdatenset, das einerseits unabhängig von und andererseits gleich verteilt wie das Trainingsdatenset sein muss (i.i.d.: *independently and identically distributed*).[65]

[64] Mit *Performance* ist in diesem Kontext v.a. die Qualität der Predictions gemeint. Betreffend Laufzeitperformance wären Verbesserungen am Modell `joint_detection` jedoch auch sehr wünschenswert.

[65] Ob die *i.i.d.*-Hypothese für das Test- und das Evaluationsdatenset zutrifft, soll nicht im Rahmen der vorliegenden Arbeit geprüft werden. Die Problematik wird in 7.2 *Ungelöste Probleme*, Seite 94 erläutert.

6.1.1 Evaluationsdaten

Als Quelle für die Trainingsdaten aller involvierter Modelle diente die SCQM-Datenbank. Für das Trainieren der Modelle wurden umfassende Datensätze daraus verwendet. In der Zwischenzeit ist die SCQM-Datenbank weiter angewachsen, sodass es eine signifikante – und wichtiger: stetig wachsende – Anzahl neuer Testdaten gibt, die zum Zeitpunkt der Erstellung der Modelle noch nicht vorhanden waren. Da die Modelle auf das Jahr 2018 (und früher) zurückgehen, können die Daten ab dem 1. Januar 2019 für die Evaluation verwendet werden. Diese wurden noch von keinem der drei Modelle «gesehen».

Für die Auswahl der Evaluationsdaten wurden vom Auftraggeber zwei CSV-Dateien zur Verfügung gestellt[66]: Die eine enthält u.a. die eigentlichen Scores mit Datumsangaben und Referenzen auf die zugrundeliegende Studie. Die andere erhält Informationen zu den Studien, u.a. auch die Referenz auf den Patienten. Die Informationen aus den beiden Dateien müssen kombiniert werden[67], um so die Dateinamen ableiten zu können. Aufgrund dieser Selektion konnten vom Auftraggeber 1619 Röntgenbilder zur Verfügung gestellt werden, die jedoch verschiedene Körperteile zeigen.

Es musste somit eine Selektion vorgenommen werden, sodass im Evaluationsdatenset nur Röntgenbilder linker Hände zu finden sind. Anhand der Metadaten (erwähnte CSV-Dateien) ist dies leider nicht möglich. Eine Selektion anhand der Scores (mindestens eine Score von MCP1-5 oder PIP1-5 ungleich null) ist auch nicht möglich, da die SCQM-Datenbank auch Bilder gesunder Hände enthält. Die Selektion hatte darum manuell zu erfolgen. Um den mechanischen Vorgang der Kategorisierung von 1619 Röntgenbildern – zeigt es eine linke Hand oder ein anderes Körperteil? – komfortabler vornehmen zu können, wurde ein Python-Skript geschrieben, womit sich die Bilder interaktiv kategorisieren lassen.[68] Auf diese Weise konnten von den ursprünglichen 1619 Röntgenbildern 290 Aufnahmen linker Hände selektiert werden.

Die CSV-Dateien enthalten zu jedem Bild die Scores aller relevanter Gelenke (MCP 1-5, PIP 1-5). Diese Scores liegen in Prozentangaben vor. Da der Prototyp, bzw. das zugrundeliegende Modell `ratingen_score` nur mit sechs Klassen umgehen kann, müssen die-

[66] Diese Dateien befinden sich aus Gründen des Datenschutzes nicht im Abgabeordner.

[67] Dies wurde mit dem Python-Skript `deepxray/test_data_selection/extract.py` bewerkstelligt

[68] Siehe `deepxray/test_data_selection/categorize.py`. Das Skript arbeitet die CSV-Datei ab, die alle Scores und Dateinamen enthält. Jedes Bild wird in Firefox in einem neuen Tab geöffnet. Dabei wandert der Fokus *nicht* auf ein anderes Fenster, sondern bleibt auf der Kommandozeile. Zeigt das Bild eine linke Hand, kann `a` (wie «accept») und `[Return]` eingegeben werden, um die Zeile aus der CSV-Datei sowie das Bild zu übernehmen. Gibt man nur `[Return]` ein, werden Bild und CSV-Zeile verworfen.

se Prozentangaben in Scores umgerechnet werden.[69] Das Evaluationsdatenset steht nun zum Scoring bereit.

6.1.2 Scoring der Evaluationsdaten

Die Daten und Python-Skripts zum Vornehmen des Scorings und Auswerten der Ergebnisse (die eigentliche Evaluation) befinden sind im Abgabeordner im Unterverzeichnis `deepxray/evaluation/`. Das Scoring der Röntgenbilder erfolgt mit einem Python-Skript, das den Prototyp mithilfe der `requests`-Library anspricht.[70] Von den ursprünglich 290 Röntgenbildern konnten in ca. 55 Minuten 247 verarbeitet werden. Auf 42 Bildern wurde fälschlicherweise keine linke Hand erkannt, worauf der Vorgang abgebrochen worden ist (HTTP-Status 501: Not Implemented).[71] Ein weiteres Bild konnte aus bisher unbekannten Gründen nicht verarbeitet werden.[72]

Die Bilder, für die keine maschinelle Prediction erstellt werden konnte, werden für den weiteren Evaluationsworkflow nicht berücksichtigt.[73] Als Erkenntnis ist aber hier festzuhalten, dass `body_part` $\frac{42}{290} = 0.145$, d.h. fast 15% *false negatives* produzierte. Die folgenden weiteren Evaluationsmetriken sind daher in erster Linie von der Performance der Modelle `joint_detection` und `ratingen_score` abhängig.

6.1.3 Wahl der Evaluationsmetriken

In 2.4.3 *Mögliche Evaluationsmetriken*, Seite 25 wurden verschiedene Evaluationsmetriken vorgestellt und auf ihre Eignung für das vorliegende Problem (Bewertung von Röntgenbildern) geprüft. Von den vorgestellten Metriken sind folgende umgesetzt worden[74]:

[69] Dies wird mit dem Python-Skript `deepxray/test_data/extract_information.py` bewerkstelligt. Dieses Skript übernimmt nur eine Teilmenge der Informationen aus der urpsrünglichen CSV-Datei, sodass diese als `deepxray/test_data/part.csv` zur Verfügung gestellt werden kann.

[70] Siehe `deepxray/evaluation/scoring/scoring.py`. Zum Zeitpunkt des Scorings wurde noch HTTP ohne TLS verwendet.

[71] Aufgrund der Testdatenselektion, die nur linke Hände zum Ergebnis hatte, kann es nur *false negatives* (linke Hand nicht als solche erkannt) und keine *false positives* (anderes Körperteil als linke Hand erkannt) geben.

[72] In der Logdatei `deepxray/evaluation/scoring/scoring.log` lassen sich hierfür keine Hinweise finden.

[73] Die CSV-Datei `deepxray/evaluation/scoring/results.csv`, welche die Resultate enthält, umfasst darum nur 247 Datensätze.

[74] Siehe das Python-Modul `evaluation` im Verzeichnis `deepxray/evaluation/evaluation` [sic.] für deren Implementierung. Als Ergänzung zur Teststrategie (4.2 *Teststrategie*, Seite 55) ist anzumerken, dass der Code für die Metriken umfassend mit Unittests (`pytest`-Framework) getestet worden ist. Diese können mit `make test` ausgefürt und mit `make cover` zusätzlich auf die Codeabdeckung geprüft werden (Requirements: `pytest` und `pycover`).

1. Global Accuracy
2. Class Accuracy
3. Precision
4. Recall
5. F1-Score
6. Cohen's Kappa
7. Cohen's Quadratic Kappa

Der Goldstandard für das vorliegende Problem wäre wohl die *Interclass Correlation*, abgekürzt ICC (Rohrbach et al., 2019, Abschnitt 3.3, S. 5-6). Da für die Evaluation jedoch nur die Einschätzungen eines Scorers pro Datenpunkt (Gelenk) verfügbar ist (siehe 6.1.1 *Evaluationsdaten*, Seite 81), kann diese Evaluationsmetrik hier nicht angewendet werden.

6.1.4 Umsetzung der Evaluationsmetriken

Beim Modell `ratingen_score`, das den sichtbaren Output des Gesamtsystems ausmacht, bzw. beim weiterentwickelten Modell, das in der vorliegenden Arbeit nicht zur Verfügung steht, wurde bemerkt, dass die Übereinstimmung zwischen menschlichem Scorer und maschineller Prediction mit 47.5% auf den ersten Blick eher bescheiden ausfällt. Erlaubt man jedoch die Abweichung von einer Klasse nach oben und nach unten, liegt die Übereinstimmung bei beträchtlichen 83%. Aus diesem Grund sollen die aufgelisteten Metriken in zwei Ausprägungen berechnet werden: Einerseits basierend auf einem «harten» Vergleich (keine Abweichungen erlaubt), andererseits basierend auf einem «*Soft Match*», d.h. Abweichungen von ±1 werden als Übereinstimmung gewertet. So kann ein ausgewogeneres Gesamtbild der Performance geliefert werden.[75]

Die gewählten Evaluationsmetriken basieren grösstenteils auf einer Konfusionsmatrix (*Confusion Matrix*). Diese listet auf der x-Achse die menschlich ermittelten Scores (*Actual*) und auf der y-Achse die maschinell ermittelten Scores (*Prediction*) auf. Die «harte» Konfusionsmatrix, die auf exakten Übereinstimmungen basiert, wird mit der Funktion `create_confusion_matrix` erzeugt.[76] Für die Soft Matches wird eine «weiche» Konfusionsmatrix mit der Funktion `create_confusion_matrix` gemäss folgender Logik erzeugt:

1. Es wird ein NumPy-Array aus Nullen mit der Dimension 6x6 (Matrix) erzeugt.
2. Es wird über eine Liste iteriert, bei welcher jedes Gelenk durch eine Tupel bestehend aus zwei Werten – menschlich ermittelte Score (*actual*) und maschinell ermittelte

[75] In der Praxis kann es durchaus einen Unterschied machen, ob eine Einschätzung nur leicht oder um einen grossen Betrag danebenliegt.

[76] Siehe `deepxray/evaluation/evaluation/matrix.py`.

Score (*prediction*) – repräsentiert ist.[77]

3. Fehlt für ein Gelenk eine Score (*actual* oder *prediction*), wird der Eintrag ignoriert und taucht nicht in der Matrix auf.

4. Die menschlich ermittelte Score wird als Referenz genommen. Weicht die maschinell ermittelte Score maximal um den Betrag von 1 ab, wird die maschinelle Score auf diesen Wert angepasst. Fällt die Abweichung grösser aus, wird die maschinelle Score beibehalten.

5. Die Matrix wird an der entsprechenden Koordinate (Zeile der maschinellen, Spalte der menschlich ermittelten Score) um den Wert 1 erhöht.

Diese Konfusionsmatrizen enthalten jeweils absolute Werte, d.h. jedes Feld repräsentiert eine Anzahl von Gelenken. Summiert man die Werte der Matrix auf, und teilt den Wert jedes Feldes durch diese Summe, erhält man eine normalisierte Konfusionsmatrix. Die vier Kombinationen aus exakt/soft und absolut/normalisiert sind auf Abbildung 18, Seite 88 zu sehen.[78]

6.1.5 Ergebnisse der Evaluation

Die Evaluationsmetriken können mit einem Python-Skript berechnet und ausgegeben werden.[79] Dabei werden folgende Ergebnisse ermittelt:

Die *Global Accuracy* beträgt nach exaktem Vergleich 69.641%. Kommt Soft Matching zum Einsatz, beträgt sie 94.089%.

Die Ergebnisse der Metriken, die nach Klasse ermittelt werden, sind in Tabelle 3, Seite 85; diejenigen, die auf *Cohen's Kappa* basieren, in Tabelle 4, Seite 85 dargestellt.

6.1.6 Interpretation der Evaluationsergebnisse

Bei den Konfusionsmatrizen (Abbildung 18, Seite 88) fällt auf, dass sich die Scores auf die beiden Klassen 0 (ca. 72%) und 1 (ca. 25%) konzentrieren. Vor diesem Hintergrund ist eine

[77] Die Daten kommen über die Funktion `extract_human_machine_results()` aus der CSV-Datei `deepxray/evaluation/scoring/results.csv`.

[78] In `deepxray/evaluation/evaluation/visualizations.py` kann nachvollzogen werden, wie aus den NumPy-Matrizen Grafiken erstellt worden sind. Weiter wird ein Histogramm erzeugt, das jedoch nur wenig aussagekräftig ist.

[79] Siehe `deepxray/evaluation/evaluation/evaluation.py`. Die Ausgabe des Skripts ist in der Datei `evaluation/evaluation/evaluation.txt` zu finden.

Metrik	Ergebnis
Class Accuracy (exakt)	$[0 : 0.86, 1 : 0.43, 2 : 0.28, 3 : 0.29, 4 : 0.00, 5 : 0.00]$
Class Accuracy (soft)	$[0 : 0.99, 1 : 0.98, 2 : 0.69, 3 : 0.57, 4 : nan, 5 : 0.00]$
Precision (exakt)	$[0 : 0.86, 1 : 0.43, 2 : 0.28, 3 : 0.29, 4 : 0.00, 5 : 0.00]$
Precision (soft)	$[0 : 1.00, 1 : 1.00, 2 : 0.86, 3 : 0.50, 4 : 0.00, 5 : 0.00]$
Recall (exakt)	$[0 : 0.76, 1 : 0.55, 2 : 0.31, 3 : 0.29, 4 : nan, 5 : 0.00]$
Recall (soft)	$[0 : 0.99, 1 : 0.98, 2 : 0.69, 3 : 0.57, 4 : nan, 5 : 0.00]$
F1-Score (exakt)	$[0 : 0.80, 1 : 0.48, 2 : 0.29, 3 : 0.29, 4 : nan, 5 : nan]$
F1-Score (soft)	$[0 : 1.00, 1 : 0.99, 2 : 0.77, 3 : 0.53, 4 : nan, 5 : nan]$

Tabelle 3: Die Ergebnisse der klassenbasierten Evaluationsmetriken.

Metrik	Ergebnis	Vertrauensintervall (95%)
Cohen's Kappa (exakt)	0.324	$[0.283, 0.365]$
Cohen's Kappa (soft)	0.957	$[0.945, 0.970]$
Cohen's Quadratic Kappa (exakt)	0.449	$[0.443, 0.454]$
Cohen's Quadratic Kappa (soft)	0.797	$[0.794, 0.801]$

Tabelle 4: Die Ergebnisse der Evaluationsmetriken, die auf *Cohen's Kappa* basieren.

abschliessende Interpration der Ergebnisse nicht angebracht. Für diese wären weitere Evaluationsdaten zu sammeln und zu verarbeiten. Dennoch können aus den verschiedenen Metriken einige Erkenntnisse gewonnen werden.

Die *Global Accuracy* ist mit ca. 70% (exakt) bzw. ca 94% (soft) zwar hoch, aufgrund der unbalancierten Daten jedoch nicht aussagekräftig.[80]

Bei den klassenbasierten Metriken fällt auf, dass für die Scores 4 und 5 keine Aussagen möglich sind. Tatsächlich lassen sich im Ergebnisdatensatz des Scorings nur zwei Gelenke mit der Score 5 und kein einziges der Score 4 finden.[81] Bei den Klassen von 0-3 fällt auf, dass die Übereinstimmungen bei den *Soft Matches* durchgehend bedeutend höher ausfallen als diejenigen auf Basis exakter Vergleiche. Weiter gilt (mit wenigen geringfügigen Ausnahmen): Je höher die Score, desto unpräziser fallen die Predictions aus. Das könnte

[80] Hat die Mehrheit der Datenpunkte die Score 0, erreichte man mit einer statischen Prediction von 0 bereits eine Accuracy von über 50%.

[81] Dies ist nicht *nur* auf die *false negatives* von `body_part` zurückzuführen. So finden sich im ganzen Evaluationsdatensatz nur vier Gelenke der Score 4 und zwei der Score 5 (siehe Python-Skript `deepxray/test_data/high_scores.py`). Falsche Scores können auch auf eine fehlerhafte Gelenkextraktion hindeuten, zumal stark geschädigte Gelenke teilweise nicht mehr als solche zu erkennen sind.

damit zu tun haben, dass die wenigen höheren Scores (Klassen 2 und 3) eher zufällig als systematisch übereinstimmen. Die *Soft Matches* zeichnen hier ein etwas positiveres Bild, so beträgt die Übereinstimmung in den Klassen von 0 bis 3 zwischen 50% und 100% (gerundet).

Auch die Metriken *Cohen's Kappa* und *Cohen's Quadratic Kappa* hinterlassen einen gemischten Eindruck im Bezug auf die Performance. Diese fallen mit 0.324 bzw. 0.449 zu tief aus, dass der klinische Einsatz des Prototypen auch nur in Erwägung gezogen werden könnte.[82]

Dass *Cohen's Quadratic Kappa* grösser ausfällt als *Cohen's Kappa* (jeweils in der exakten Variante), mag zunächst überraschen. Dies ist jedoch ein bekanntes statistisches Phänomen bei sogenannten *tridiagonalen Tabellen*. Eine rechteckige Matrix (z.B. eine Konfusionsmatrix) ist tridiagonal, wenn nur in der Hauptdiagonalen (Übereinstimmungen) und in den beiden benachbarten Diagonalen (Sub- und Superdiagonalen) Werte auftauchen, die sich von null unterscheiden. Da es sich bei der Konfusionsmatrix in der vorliegenden Evaluation annähernd um eine tridiagonale Matrix handelt[83], ist der Unterschied zwischen den beiden Kappa-Metriken durchaus plausibel (Warrens, 2011).

Auf Basis der *Soft Matches* fallen die Kappas mit 0.957 bzw. 0.797 wesentlich höher aus. Dies könnte dahingehend interpretiert werden, dass der Prototyp zwar nicht sechs Klassen zuverlässig auseinanderhalten kann, jedoch in seinen Predictions nur geringfügige Abweichungen aufweist und immerhin für eine Unterscheidung zwischen tiefer, mittlerer und hoher Schädigung zuverlässig arbeitet. Aufgrund der Datenbasis mit sehr wenigen Daten in den höheren Klassen ist diese Interpretation jedoch voreilig.

6.1.7 Anmerkung zu Soft Matches

Die Soft Matches sind mit Vorsicht zu geniessen. Die menschlichen Scores liegen oft auf den Klassengrenzen der Predictions, z.B. liegt 20% gerade auf der Grenze zwischen Klasse 1 ($]0\%..20\%]$) und 2 ($]20\%..40\%]$), gehört per Definition aber zur Klasse 1 (Rohrbach, 2017, S. 10). Somit sollte eine maschinelle Prediction legitimerweise den Scores 1 oder 2 zugeordnet werden – zu 0 (0%) und 3 ($]40\%..60\%]$) jedoch besser nicht. Die Soft-Match-

[82] *«Any kappa below 0.60 indicates inadequate agreement among the ratorers and little confidence should be placed in the study results.»* (McHugh, 2012) Jedes Kappa unter 0.60 weist auf eine inadäquate Übereinstimmung unter den Bewertern hin, diesen Studienergebnissen sollte nur geringes Vertrauen entgegengebracht werden. (Übersetzung des Autors)

[83] Die Soft Matches, welche in den Haupt-, Sub- und Superdiagonalen zu liegen kommen, liegen ja weit über 90%.

Logik interpretiert jedoch nicht nur eine Score von 1 und 2, sondern auch von 0 als Übereinstimmung mit 20%. Was bei einer menschlichen Score von 10% durchaus sinnvoll wäre, schiesst bei Grenzfällen wie 20% übers Ziel hinaus. Die Soft Matches arbeiten somit tendenziell zu grosszügig.

Um solche Grenzfälle in der Evaluation präziser behandeln zu können, müsste der Evaluationsworkflow Gebrauch der exakten menschlichen Scores machen. So könnte die *Soft Confusion Matrix* beispielsweise für 10% die Klassen 0, 1 und 2 als Treffer werten, für 20% hingegen nur die Klassen 1 und 2. Dies dürfte zu präziseren – und wohl etwas schlechter ausfallenden – Soft-Metriken führen. Dieser erweiterte Evaluationsworkflow soll jedoch nicht im Rahmen der vorliegenden Arbeit umgesetzt werden.[84]

Die Toleranz der Abweichung könnte zudem empirisch bestimmt werden, indem die Abweichungen verschiedener menschlicher Scorer berücksichtigt werden. Hierfür stehen jedoch wissenschaftlich fundierte Metriken zur Verfügung (Stichwort *Interclass Correlation*, ICC), sodass man vom Soft-Matching-Ansatz wegkommen könnte, stünden die Ergebnisse mehrerer menschlicher Scorer zur Verfügung.

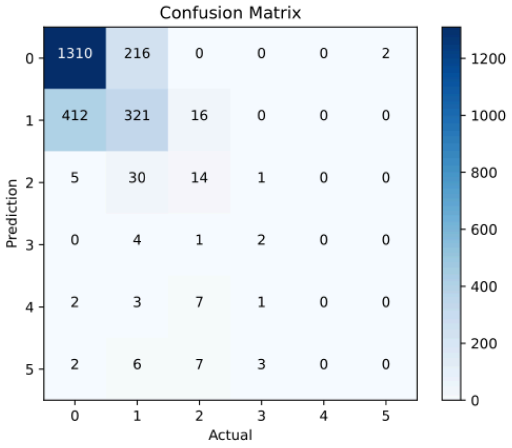
6.1.8 Fazit

Der Evaluationsdatensatz ist unzureichend, um eine verbindliche Evaluation des Prototypen vornehmen zu können. Zwar weisen die Predictions für die tieferen Scores eine hohe Übereinstimmung auf, und die Abweichungen fallen betraglich eher gering aus. Für die höheren Klassen gibt es aber nur vereinzelte Datenpunkte, die keine signifikante Aussage zur Performance zulassen.

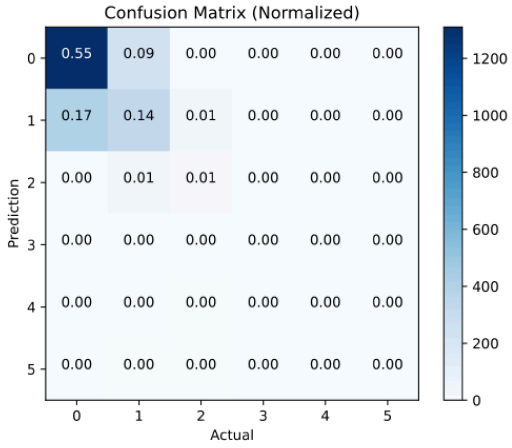
Wichtiger als ein solches Gesamturteil ist jedoch die Tatsache, dass ein weitgehend automatisierter Evaluationsworkflow mit verschiedenen Metriken umgesetzt worden ist, der in Zukunft mit weiteren Daten gefüttert und erneut ausgeführt werden kann. Schliesslich ist das Ziel der Evaluation nicht nur, eine einmalige Einschätzung des Systems vorzunehmen, sondern eine Möglichkeit zu haben, systematische Vergleiche zwischen verschiedenen Versionen des Systems anstellen zu können.

Für künftige Evaluationen ist zu beachten, dass neue und für Modellverbesserungen verwendete Trainingsdaten nicht mehr in der darauffolgenden Evaluation verwendet werden sollten.

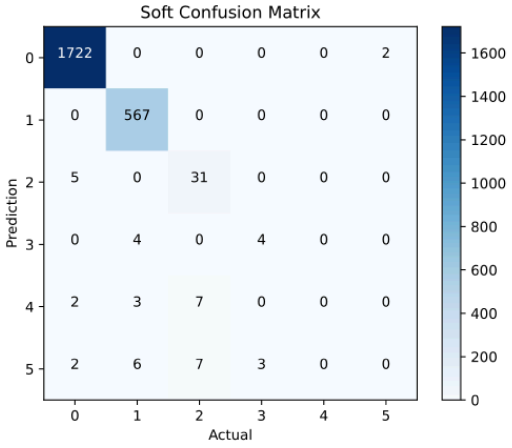
[84] Hierbei handelt es sich um einen klassischen Zielkonflikt: Der Data Miner möchte möglichst viele Daten zur Verfügung haben – der Datenschützer möchte nur diejenigen zur Verfügung stellen, die für den jeweiligen Zweck unbedingt nötig sind. Der Projektmanager hingegen drängt auf den Abschluss der Evaluation, da noch eine Dokumentation zu schreiben ist.



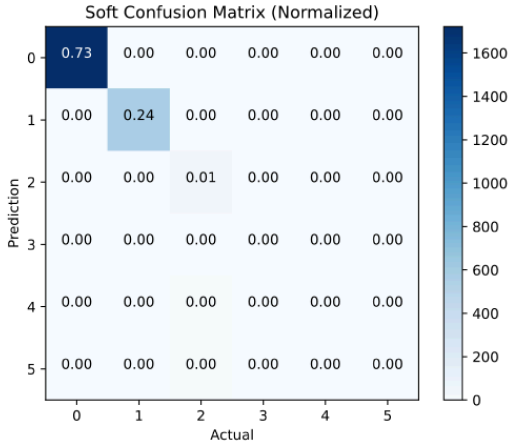
(a) Die exakte Konfusionsmatrix mit absoluten werten (Anzahl Gelenke).



(b) Die exakte Konfusionsmatrix normalisiert.



(c) Die «weiche» Konfusionsmatrix mit absoluten werten (Anzahl Gelenke).



(d) Die «weiche» Konfusionsmatrix normalisiert.

Abbildung 18: Die verschiedenen Konfusionsmatrizen im Überblick.

6.2 Evaluation der Zielerreichung

An dieser Stelle soll überprüft werden, ob und wie die Projektziele (siehe 1.1.2 *Erwartetes Resultat*, Seite 11) erreicht worden sind. Dies sind:

1. *Stand der Technik zur Industrialisierung (Evaluation, Testing) von ML-Modellen:* Dieses Thema wurde im Abschnitt 2.4 *Evaluation von Modellen*, Seite 22 auf Basis verschiedener Publikationen untersucht.

2. *Vorschlag für ein generisches Format von Modellen zum Exportieren aus ML-Frameworks und zum Importieren zur Ausführung:* Diese Problematik wurde in Abschnitt 2.5 *Modellformate*, Seite 29 behandelt. Mittelfristig dürfte das SavedModel-Format von TensorFlow besser geeignet sein als das Format der ONNX-Initiative.

3. *Vorschlag für geeignete Metriken zum Vergleich von Modellen:* Hierzu wurden allgemeine Metriken (Accuracy, Precision, Recall, F1-Score) als auch domänenspezifische (Cohen's Kappa) gefunden (siehe 2.4.3 *Mögliche Evaluationsmetriken*, Seite 25).

4. *Aufzeigen, was für eine Verheinheitlichung der Codebasis gemacht werden muss*: Diese Fragestellung wird im Ausblick behandelt (siehe 7.2.1 *Aktualisierung der Modelle*, Seite 96).

5. *Lauffähiger Prototyp*: Der Prototyp funktioniert und ist im Kapitel 5 *Realisierung*, Seite 60 ausführlich dokumentiert.

    a) *Automatisierte Evaluation und Testing von neuen Modellen mit Vergleich zu alten Modellen und Minimum-Standards*: Der Evaluationsworkflow wurde umgesetzt, mit einem Evaluationsdatenset durchgeführt und ist in Abschnitt 6.1 *Technische Evaluation*, Seite 80 dokumentiert.

    b) *Visualisierung und Reporting der Performanz neuer Modelle*: Die Performanz kann über den Evaluationsworkflow (siehe wiederum 6.1 *Technische Evaluation*, Seite 80) ausgewertet werden. Als Visualisierung dienen verschiedene Konfusionsmatrizen (siehe Abbildung 18, Seite 88). Ein zusätzlich erstelltes Histogramm ist leider nur wenig aussagekräftig.

    c) *API zum Ausführen des Modelles (Verteilen über mehrere GPUs)*: Die verschiedenen Modellkomponenten interagieren über eine Schnittstelle, die in Kapitel 5 *Realisierung*, Seite 60 dokumentiert und auf Tabelle 6, Seite 106 übersichtlich zusammengefasst ist. Die Verteilung auf mehrere GPUs wurde nicht unternommen, da bereits mit CPUs eine akzeptable Performance erreicht werden konnte.

d) *Load Balancing, Messaging Systems*: Diese Thematik wurde in der Architekturdiskussion (3.1 *Architekturvarianten*, Seite 31) ausführlich behandelt. Die verschiedenen Varianten für Messaging-Systeme werden kurz im Anhang evaluiert (siehe 8.2 *Wahl der Message-Queue*, Seite 104).

e) *Nahtloses Austauschen von alten Modellen mit neuen Versionen*: Die Austauschbarkeit kann nicht auf Modellebene, jedoch auf der Ebene der Modell*komponenten* gewährleistet werden. Die Schnittstellen sind in Tabelle 6, Seite 106 zu finden.

Fazit: Die wichtigsten Projektziele konnten erreicht werden. Dass die Austauschbarkeit der Machine-Learning-Modelle nicht auf Modellebene stattfinden konnte, zeichnete sich schon früh im Projektverlauf ab. Entsprechend wurde der Fokus früh auf die Austauschbarkeit von Modellkomponenten über definierte Schnittstellen gelegt. Im Bereich Visualisierung/Reporting wurde vergleichsweise wenig Aufwand investiert. Dafür musste zusätzliche Zeit investiert werden, um ein Modell neu zu trainieren, wofür die Modelldaten nicht mehr aufzufinden waren. Auch das Selektieren und Aufbereiten der Evaluationsdaten verursachte zusätzlichen Aufwand.

6.2.1 Rückblick auf Projektrisiken

Die zu Beginn des Projekts aufgestellten Projektrisiken (siehe 1.2 *Projektrisiken*, Seite 12) sollen an dieser Stelle wieder aufgegriffen werden. Sind Risiken eingetreten, und wenn ja, welchen Einfluss hatte dies auf den Projektverlauf? Haben sich die vorgeschlagenen Mitigationsmassnahmen als sinnvoll erwiesen?

Keine lauffähigen Modelle Dieses Risiko ist teilweise eingetreten, denn die Modelldaten zum `ratingen_score`-Modell waren nicht mehr auffindbar. Die Mitigationsmassnahme, das Modell neu zu trainieren, konnte den Schaden eingrenzen. Dabei ist zwar Mehraufwand entstanden, das Projekt wurde dadurch aber nur unwesentlich verzögert.

Aktualisierung der Modelle Dieses Risiko ist eingetreten, sodass die Aktualisierung der Modelle schon früh im Projektverlauf aufgegeben worden ist. Die Mitigationsmassnahmen (Isolation der bestehenden Modelle) haben sich als zielführend erwiesen. Der Prototyp konnte auf Basis der bestehenden Modelle umgesetzt werden.

Schlechte Performance (Prediction) Die technische Evaluation lässt hier (noch) kein eindeutiges Urteil zu. Die Predictions bewegen sich ganz klar nicht im reinen Zufalls-

bereich, dürften aber für einen Produktiveinsatz noch zu wenig präzise sein. Dieses Risiko wird vom Auftraggeber getragen.

Schlechte Performance (Laufzeit) Die Verarbeitungsgeschwindigkeit des Prototypen ist akzeptabel, wenn auch nicht berauschend. Die Anwendung ist skalierbar, sodass sich Hardwareerweiterungen (zusätzliche CPUs) und -verbesserungen (schnellere CPUs) positiv auf die Performance auswirken dürften.

Hoher Arbeitsspeicherverbrauch Das Gesamtsystem benötigt mindestens 2 GB Arbeitsspeicher (eine Instanz pro Modellkomponente). Empfohlen sind 6 GB oder besser 11 GB (für fünf bzw. zehn laufende Instanzen der Gelenkextraktion). Diese Anforderungen stellen angesichts gegenwärtiger Speicherpreise kein Problem dar. Der Speicherbedarf ist jedoch kritisch, wenn die Modelle auf einer oder mehreren GPUs ausgeführt werden sollen.

Auf die Risiken, die für die einzelnen Projektphasen ermittelt worden sind (siehe 4.1 *Projektphasen*, Seite 46), soll an dieser Stelle nicht weiter eingegangen werden. Diese wurden jeweils – falls nötig – in der Reflexion der einzelnen Projektphasen kurz behandelt.

Obwohl einige der ermittelten Projektrisiken eingetreten sind, wurde der Erfolg des Projekts dadurch nicht beeinträchtigt. Die Mitigationsmassnahmen haben sich als sinnvoll und ausreichend erwiesen.

7 Ausblick

Nach einem Semester der Recherche, Konzeption, Umsetzung und Dokumentation liegt als Projektergebnis ein lauffähiger, gut dokumentierter und skalierbarer Prototyp vor, der bei weitem noch nicht den Anforderungen für einen Produktiveinsatz genügt. Hierzu wäre noch einiges an Arbeit zu leisten, worauf im Abschnitt 7.2 *Ungelöste Probleme*, Seite 94 näher eingegangen wird. Im Abschnitt 7.3 *Weitere Ideen*, Seite 99 sollen weitere Möglichkeiten besprochen werden, wie der entwickelte Prototyp in Zukunft erweitert und verbessert werden könnte.

Zunächst soll aber das im Rahmen der Bachelorarbeit Geleistete reflektiert werden.

7.1 Reflexion der Arbeit

Zu Beginn der Arbeit traten einige Schwierigkeiten im Zusammenhang mit den gegebenen Modellen auf. Diese mussten zunächst wieder zum Laufen gebracht und teilweise sogar neu trainiert werden. Der Projekterfolg stand zu diesem Zeitpunkt in Frage. Der definitive Projektauftrag konnte erst zum spätmöglichsten Zeitpunkt erteilt werden.

Die Modelle konnten zudem nicht einfach für eine aktuelle Laufzeitumgebung konvertiert werden. Dies wäre sehr aufwändig gewesen und hätte den zeitlichen Rahmen des Projekts gesprengt. Stattdessen wurde die Entscheidung gefällt, die bestehenden Modelle in einer isolierten Laufzeitumgebung anzubieten. Die verschiedenen Modelle konnten so in der ersten Projektphase doch noch zum Laufen gebracht werden.

In der zweiten Projektphase wurde viel recherchiert und konzipiert. Der Projektfortschritt war in dieser Phase nicht offensichtlich, da die verschiedenen Architekturvarianten v.a. im Kopf des Autors und auf Notizpapier ausgearbeitet worden sind. Um bei der Zwischenpräsentation nicht nur ein reines Theoriekonstrukt demonstrieren zu müssen, wurde ein Prototyp mit simulierten Modellkomponenten entwickelt.[85]

Dieser «Fake-Prototyp» erwies sich als Glücksfall, zumal dessen Architektur grösstenteils für die Entwicklung des *eigentlichen* Prototypen übernommen werden konnte. Die Entwicklung desselben ging dann v.a. während zweier Wochenenden vonstatten, wonach nur noch geringfügige Verbesserungen vorgenommen worden sind.

Die Recherche, Wahl und Umsetzung der Evaluationsmetriken nahm mehr Zeit in Anspruch als erwartet. Auch mussten die Evaluationsdaten in verschiedenen Schritten aufbereitet und selektiert werden. Dank der schnellen Fortschritte beim Prototyp führten diese Aufwände jedoch zu keiner Verzögerung im Projekt.

[85] siehe `https://github.com/patrickbucher/fake-x-ray` (abgerufen am 26.05.2020)

Auf Basis einer pragmatischen und differenzierten Teststrategie wurden zwar eher wenige und kleine, dafür aber aussagekräftige automatisierte Tests umgesetzt. So kommen bei der Evaluation Unittests (mit hoher Codeabdeckung), bei den Modellkomponenten Integrationstests und für den `orchestrator` (bzw. für das Gesamtsystem) Systemtests (End-to-End) zum Einsatz.

Die Dokumentation hinkte dem Projektfortschritt über die meiste Zeit etwas hinterher, war aber zum Schluss des Projekts doch recht schnell geschrieben. Hier erwiesen sich die Erfahrungen des Wirtschaftsprojekt (sowie das damals erstellte LaTeX-Template) als sehr nützlich.

Das Projektergebnis ist für den Autor sehr zufriedenstellend ausgefallen und schafft für den Auftraggeber eine solide Basis für die künftige Weiterentwicklung.

7.1.1 Lessons Learned

1. Das systematische Nachdenken über Architekturentscheidungen mit verschiedenen Varianten mag in einer Zeit, die «Agilität» als höchstes Ideal preist, als überholt erscheinen und etwas nach V-Modell anmuten. Es kann jedoch wesentlich effektiver und effizienter sein, möglichen Problemen durch ungestörtes Nachdenken auf die Spur zu kommen, als diese erst nach dem Ende mehrerer Sprints (in die falsche Richtung) erkennen zu müssen.

2. Eine Software muss zwecks Qualitätsicherung systematisch getestet werden. Systematisch heisst jedoch nicht, dass jeder Aspekt der Software mit dem gleichen (Unittests), sondern mit einem auf das jeweilige Problem passenden Werkzeug (Unittests, Integrationstests, Systemtests, manuelle Tests usw.) Werkzeug getestet werden soll. Ein schlanker End-to-End-Test ist oft aussagekräftiger, stabiler und besser wartbar als eine Ansammlung von Unittests mit Mocks und Test-Doubles.

3. Obwohl es sich bei der vorliegenden Arbeit eher um ein Softwareentwicklungs- als um ein Machine-Learning-Projekt handelt, war ein grundlegendes Verständnis von Machine-Learning-Konzepten nötig, v.a. bei der Evaluation. Auch grundlegende Kenntnisse der Domäne (Scoring von Röntgenbildern, rheumatoide Arthritis) haben sich als extrem wichtig erwiesen, so konnte etwa eine Unstimmigkeit in der Struktur der Evaluationsdaten über die Sichtung einiger Röntgenbilder erkannt und behoben werden.

7.1.2 Verwendete Werkzeuge

Zur Entwicklung des Prototyps wurden die Programmiersprachen Go und Python verwendet. Als wichtige Libraries (die gegebenen Machine-Learning-Frameworks ausgenommen) kamen u.a. AMQP, UUID (Go) sowie NumPy, Pandas, SciPy Stats, Requests, PyTest und Pika (Python) zum Einsatz. Die Werkzeuge `make`, `bash` und `jq` erwiesen sich an verschiedenen Stellen als sehr hilfreich.

Der Prototyp macht regen Gebrauch von Docker und kann am einfachsten mit Docker Compose ausgeführt werden. RabbitMQ hat sich als sehr robuste und einfach zu handhabende Messaging-Lösung erwiesen.

Der Bericht und das Web-Abstract wurden mit LaTeX bzw. XeLaTeX gesetzt. Für das Arbeitsjournal kamen Markdown und Pandoc zum Einsatz. Die PDF/A-Konformität der Dokumente wurde mit Ghostscript sichergestellt. Die UML-Diagramme wurden mit PlantUML erstellt. Für weitere Grafiken kam Graphviz zum Einsatz.

Das Video wurde mit `ffmpeg` aufgenommen. Die Tonspur wurde mit Audacity aufgenommen, abgemischt und wiederum mit `ffmpeg` mit dem Video kombiniert.

Quellcode und Dokumentation wurden ausschliesslich in `vim` geschrieben. Plugins wie `vim-go` und `python-mode` machen diesen flexiblen Texteditor zu einer komfortablen Entwicklungsumgebung.

Sämtliche Quellcode- und Textartefakte wurden mit Git verwaltet und regelmässig zusammen mit Modell- und Testdaten mithilfe von `restic` auf einen externen Datenträger gesichert.

7.2 Ungelöste Probleme

Obwohl die Projektziele erreicht worden sind (siehe 6.2 *Evaluation der Zielerreichung*, Seite 89), bleibt noch einiges zu tun, damit der Prototyp dereinst produktiv eingesetzt werden kann. Im Verlauf der Arbeit wurden folgende ungelöste Probleme gesammelt und vom Rahmen der Bachelorarbeit abgegrenzt:

Modelle aktualisieren Die bestehenden Machine-Learning-Modelle sollen auf eine aktuelle Version von TensorFlow aktualisiert werden. Diese Problematik wird ausführlicher im Abschnitt 7.2.1 *Aktualisierung der Modelle*, Seite 96 behandelt.

Deployment der Container Beim Auftraggeber (Seantis GmbH) kommen zwar Container zum Einsatz, die auf Docker-Images basieren. Diese werden jedoch nicht mit

der Docker-Runtime, sondern mit `systemd-nspawn` ausgeführt.[86] Auch verfügt der Auftraggeber über ein eigenes Container-Buildsystem, in welches die erstellten Container integriert werden müssten. Der bisherige Ansatz mit Docker Compose ist nur für Test- und Demozwecke geeignet.

Continuous Integration Die verschiedenen Unit-, Integrations- und Systemtests wurden während der Entwicklung des Prototyps jeweils manuell ausgeführt. Für eine produktive Weiterentwicklung der Software wäre die Integration der Test-Pipeline in das CI-System des Auftraggebers angebracht. Dieses basiert auf Buildkite.[87] Um dieses System effektiv nutzen zu können, wäre wohl auch die Aufteilung des bisherigen Git-Repositories in mehrere Repositories (eines pro Komponente) sinnvoll.

Security Für eine produktive Anwendung, die mit schützenswerten Daten wie Röntgenbildern umgeht, gelten andere Sicherheitsanforderungen als für einen Prototyp. Für die Kommunikation zwischen dem Client und dem `orchestrator` dürfen keine selbst signierten, sondern nur von einer Zertifizierungsstelle ausgestellten TLS-Zertifikate verwendet werden. Die Message-Queues sind über geeignete Massnahmen zu schützen, indem diese nur intern zugänglich gemacht oder durch zureichende Authentifizierungs- und Autorisierungsmechanismen geschützt werden.

i.i.d.-Hypothese Die Trainings- und Evaluationsdaten müssen der i.i.d.-Hypothese (independent, identically distributed) genügen, das heisst unabhängig voneinander sein und der gleichen Verteilung folgen. Diese Hypothese konnte im Rahmen der Bachelorarbeit nicht geprüft werden. Es gibt mögliche Hinweise darauf, dass diese Hypothese für die verwendeten Röntgenbilder nicht zutreffen könnte. Zum einen sind schwere Krankheitsverläufe der rheumatoiden Arthritis aufgrund verbesserter Früherkennung und Therapiemassnahmen immer seltener. Die betroffenen Patienten sterben zusehends weg, was zu einer laufenden Verschiebung der Ratingen-Scores nach unten führen wird. Andererseits werden in der SCQM-Datenbank auch die Krankheitsverläufe der gleichen Patienten verfolgt, d.h. die stetig fortschreitenden (da irreversiblen) Schädigungen an Gelenken führen im Verlauf der Langzeitstudien zu Ratingen-Scores, die ansteigen oder bestenfalls stagnieren. Ob und inwiefern sich diese Verschiebungen mit der laufenden Aufname von Daten neuer Patienten in die SCQM-Datenbank ausgleicht, kann nur mit einer systematischen statistischen Untersuchung ermittelt werden.

[86] Die Gründe für und Vorteile dieses Ansatzes sind vielfältig (Walsh, 2016).
[87] siehe `https://buildkite.com/` (abgerufen am 26.05.2020)

Gleichbleibende Predictions Ein oft genannter Vorteil des automatischen Scorings von Röntgenbildern gegenüber menschlichen Scorern ist, dass der erste Ansatz eine konstante Qualität der Predictions gewährleistet. Dies gilt jedoch nur, solange die Modelle unverändert zum Einsatz kommen. Werden diese durch neuere, verbesserte Versionen ersetzt, würden für bestehende Daten wohl andere Scores resultieren. Bei diesem Zielkonflikt – konstante oder möglichst genaue Scores – gilt es aufseiten des Auftraggebers die Prioritäten festzulegen.

Evaluation In 2.4.1 *Verschiedene Datentypen*, Seite 22 wurde für das Modell `ratingen_score` erwähnt, dass die Evaluation nicht nur auf Basis einzelner Gelenke stattfinden soll, sondern dass auch die Anzahl erkannter Gelenke pro Röntgenbild ein mögliches Qualitätskriterium sei. Diese Evaluationsmetrik wurde nicht umgesetzt. Diese Metrik würde Rückschlüsse auf die Qualität des Modells `joint_detection` erlauben.

7.2.1 Aktualisierung der Modelle

Die verschiedenen Machine-Learning-Modelle basieren auf älteren Versionen von TensorFlow und anderen Libraries/Frameworks (hier als *Packages* zusammengefasst). Tabelle 5, Seite 96 bietet einen Überblick über die verwendeten und derzeit aktuellen Versionen.

Modell	Package	eingesetzt	aktuell
`body_part`	`tflearn`	`0.2.1`	`0.3.2`
`joint_detection`	`tensorflow`	`0.12.1`	`2.2.0`
`joint_detection`	`scipy`	`0.18.1`	`1.4.1`
`joint_detection`	`Pillow`	`3.4.2`	`7.1.2`
`ratingen_score`	`tensorflow`	`1.4.0`	`2.2.0`
`ratingen_score`	`scikit-learn`	`0.22.2.post1`	`0.23.1`

Tabelle 5: Die verwendeten Versionen verschiedener Machine-Learning-Packages sind grösstenteils veraltet.

Um das Ausmass der nötigen Änderungen für eine Aktualisierung auf neuere Packages besser abschätzen zu können wurde folgendermassen verfahren:

1. Die Modellkomponenten wurden in ein eigenes Arbeitsverzeichnis kopiert.
2. Die `Dockerfiles`, die auf älteren Versionen von Python basieren, wurden gelöscht. Schliesslich sollen die Modellkomponenten mit einer aktuellen Version von Python,

d.h. auch ausserhalb eines Containers, betrieben werden können.

3. Stattdessen wurde pro Modellkomponente eine neue virtuelle Python-Umgebung basierend auf Python 3.8.3 erstellt.
4. In den `requirements.txt`-Dateien wurden sämtliche Versionsangaben entfernt, sodass die jeweils neueste Version der Packages installiert wird.
5. Damit die Modellkomponenten überhaupt aufgestartet werden können, wurde eine Instanz von RabbitMQ im Hintergrund gestartet.
6. Es wurde versucht, die einzelnen Einstiegspunkte der Modellkomponenten (Hauptklasse) aufzustarten. Dabei aufgetretene Fehlermeldungen wurden nachgeschlagen.
7. Der Integrationstest im `tests/`-Unterverzeichnis wurde via `pytest` (nicht via Docker!) aufgestartet; Fehlermeldungen wiederum nachgeschlagen.

Für die aufgetretenen Fehlermeldungen werden möglichen Lösungsansätze skizziert. Hierzu sollen pro Modellkomponente höchstens eine Stunde aufgewendet werden. Dieser Ausblick kann nur eine Idee über das Ausmass der anstehenden Arbeiten, jedoch keine ausführlichen Arbeitsanleitungen liefern. Demnach gibt es für die Aktualisierung der Modelle folgendes zu tun:

`body_part` Das Unterpackage `contrib` wurde für TensorFlow Version 2 entfernt. Dieses kann nur zusammen mit TensorFlow 1 verwendet werden, was nicht im Sinne einer Aktualisierung ist. `tflearn` bietet derzeit keine Unterstützung für TensorFlow 2.0.[88] Da diese Situation nun schon seit über einem Jahr besteht, wäre die Umstellung von `tflearn` auf Keras wohl die bessere Variante als das Warten auf eine neue `tflearn`-Version mit Unterstützung für TensorFlow 2. Hierfür müsste der Code grösstenteils neu geschrieben und das Modell neu trainiert werden, was auch mit dem bekannten Problem der Modellformate (siehe 2.5 *Modellformate*, Seite 29) zu tun hat. Das zugrundeliegende Modell *Inception V3* kann auch für Keras verwendet werden.[89] Die Änderungen beschränken somit auf das Ansprechen der High-Level-API, da das Modell konzeptionell (Definition der Layer) übernommen werden kann.

`joint_detection` Hier kommt nicht `tflearn`, sondern die «rohe» API von TensorFlow zum Einsatz. Es muss somit nicht auf eine komplett andere, sondern bloss auf eine neuere API migriert werden. Rein technisch kommt Version 0.12.1 zum Einsatz.

[88] `https://github.com/tflearn/tflearn/issues/1121` (abgerufen am 26.05.2020)
[89] `https://keras.io/api/applications/inceptionv3/` (abgerufen am 26.05.2020)

Dies war die letzte Minor-Version vor dem 1.0-Release. Zur Migration von TensorFlow 1 auf 2 gibt es einen offiziellen und umfassenden Guide.[90]. Ob dieser auch für eine Migration von der Version 0.12.1 auf die aktuelle Version 2.2.0 funktioniert, müsste ausprobiert werden. Hier wäre ein Ausweichen auf die Keras-API wohl einfacher. Die `joint_detection`-Modelle basieren auf dem Modell *Inception V1*. Dieses arbeitet weniger genau und langsamer als die aktuelleren Varianten V2 und V3.[91] Dieses Basismodell wird nicht über ein Package, sondern direkt als Code (`joint_detection/legacy/utils`) zur Verfügung gestellt. Eine Umstellung auf Inception V3 würde diesen und weiteren Code im `utils/`-Unterverzeichnis (inklusive einigen C++-Code und die Library `stitch_wrapper.so`) obsolet machen. Das Umschreiben des TensorFlow-Codes würde sich so auf die Klasse `JointDetectionModel` und auf den Trainingscode beschränken. Alternativ kann Inception V3 auch über die Keras-API verwendet werden, was ganz im Sinne der Vereinheitlichung des Modell-Codes wäre.

`ratingen_score` Hierbei handelt es sich um das neueste Modell, das dementsprechend auch auf einer aktuelleren API von TensorFlow (Version 1.4.0) basiert. Hier wird TensorFlow über die Keras-API angesprochen, wodurch die Aktualisierung auf TensorFlow 2.2.0 keine grösseren Probleme verursachen sollte. Auch das Inception-Basismodell wird in der aktuellen Version 3 verwendet. Zudem ist diese Modellkomponente die einzige ohne Legacy-Code, wodurch keine zusätzliche Arbeit zum Portieren von Code anfällt. Ein erster Versuch der Aktualisierung auf TensorFlow 2.2.0 verlief vielversprechend. Die Laufzeitumgebung beschwerte sich einzig über den eingesetzten Adam-Optimizer. Die Modellkomponente konnte jedoch ohne diesen zum Laufen gebracht werden. Detaillierte Tests im Bezug auf die Qualität der Predictions und auf die Laufzeitperformance wurden jedoch keine unternommen. Solche könnten jedoch mit den Code-Artefakten aus dem GitHub-Repository[92] ohne grossen Aufwand unternommen werden.

Fazit: Die beiden Modellkomponenten `body_part` und `joint_detection` sollten über die Keras-API auf TensorFlow 2 aktualisiert werden. Bei letzterem sollte zusätzlich vom Basismodell Inception V1 auf Inception V3 umgestellt werden. Das Modell `ratingen_score` sollte sich problemlos auf TensorFlow 2 aktualisieren lassen, zumal es schon die

[90] `https://www.tensorflow.org/guide/migrate` (abgerufen am 26.05.2020)

[91] `https://towardsdatascience.com/a-simple-guide-to-the-versions-of-the-inception-network-7fc52b863202` (abgerufen am 26.05.2020) bietet einen Vergleich der verschiedenen Versionen.

[92] `https://github.com/janickrohrbach/arthritis-net` (abgerufen am 26.05.2020)

Keras-API verwendet. (Dieses Modell war schliesslich nicht der Grund dafür, dass die Aktualisierung der Modelle nicht in den Umfang der Bachelorarbeit aufgenommen worden ist.)

7.3 Weitere Ideen

An dieser Stelle sollen noch einige Ideen skizziert werden, die im Verlauf der Arbeit aufgekommen sind, jedoch klar ausserhalb des Projektscopes liegen und für diesen nie in Betracht gezogen worden sind.

Rau-Score Die Ratingen-Score bezieht sich auf die Schädigung eines einzelnen Gelenks. Mit der Rau-Score werden die Schädigungen verschiedener Gelenke (je elf an den beiden Händen, d.h. MCP 1-5, PIP 1-5 und das Handgelenk; sowie fünf Gelenke pro Fuss) zu einer einzigen Score zusammengefasst, die den Fortschritt der Erkrankung bei einem Patienten in einer einzigen Zahl zusammenfasst. Um diese Rau-Score berechnen zu können, müssten zusätzlich das Handgelenk sowie die Gelenke an den Füssen verarbeitet werden können. Die Berechnung der Rau-Score wäre dann trivial.

Optische Aufbereitung Bei der Web-Oberfläche wäre es sehr hilfreich, wenn die detektierten und gescorten Gelenke direkt mit einem farbigen Rahmen – von grün, keine Schädigung, bis rot, starke Schädigung – hervorgehoben werden könnten. Das Nachschauen an drei Orten – Röntgenbild, Liste der Scores und Legende – könnte so entfallen. Um dies umsetzen zu können, müsste die Modellkomponente `joint_detection` zusätzlich die ermittelten Bildkoordinaten der detektierten Gelenk zurückliefern, sodass diese anschliessend umrahmt werden könnten. Hierzu müsste die API erweitert werden. Dies könnte einerseits über einen weiteren Endpoint geschehen, oder mithilfe eines `Accept`-Headers, womit zwischen `image/jpeg` (für eine annotierte Grafik) und `application/json` (für den bestehenden JSON-Payload) unterschieden würde.

Implementierung in Erlang Recherchen zum Message-Broker RabbitMQ haben den Autor auf dessen zugrundeliegende Programmiersprache *Erlang* aufmerksam gemacht. Zwar konnte der `orchestrator` dank der hervorragenden Concurrency-Mechanismen von Go problemlos umgesetzt werden. Mit Erlang, das auf dem *Actor*-Modell basiert, würden sich derlei Orchestrierungsprobleme wohl noch eleganter lösen lassen. So könnten die verschiedenen Modellkomponenten mit ihrer Anbindung an

RabbitMQ als Actors abstrahiert werden. Der Nachrichtenfluss könnte dann mit einer deklarativen, auf Prolog basierenden Syntax einfach und übersichtlich zwischen diesen Actors definiert werden. Die Unterstützung dafür benötigter Technologien wie HTTP, JSON usw. wäre über die OTP gewährleistet.[93] Die Reimplementierung der `orchestrator`-Komponente böte eine schöne Fallstudie zum Vergleich der Concurrency-Modelle von Go (CSP) und Erlang (Actor-Modell) – und ein praxisnahes Erlang-Lernprojekt.

[93] Weitere Informationen zu Erlang finden sich auf der offiziellen Webseite (`https://erlang.org/doc/`, abgerufen am 26.05.2020), sowie in Joe Armstrongs Einführungsbuch (Armstrong, 2013) und Doktorarbeit (Armstrong, 2003).

8 Anhang

Im Anhang werden folgende Inhalte gesammelt:

Technologieevaluation Die Wahl der Programmiersprache – bzw. der Programmier*sprachen*, da verschiedene zum Einsatz kommen – wird anhand definierter Kriterien für die verschiedenen Einsatzbereiche (Komponenten) begründet. Auch die Wahl des Messaging-Protokolls und des Message-Brokers wird kurz begründet.

Schnittstellen Die Schnittstellen zwischen den verschiedenen Komponenten werden in einem kurzen, tabellarischen Überblick zusammengefasst.

Abgabe Abgegebene Artefakte wie Quellcode und Modelldaten, und weitere Inhalte, die nicht in den Bericht (dieses Dokument) eingefügt worden sind, werden hier aufgeführt.

8.1 Wahl der Programmiersprache(n)

Die Wahl der Programmiersprache ist in der vorliegenden Arbeit durch die vorgegebenen Modelle teilweise eingeschränkt. Bei anderen Komponenten, d.h. beim `orchestrator` und bei der Evaluation, bestehen grundsätzlich keine solchen Einschränkungen.

Wo eine Programmiersprache gewählt werden kann, soll dies nach den folgenden Kriterien geschehen:

Domäne Die Programmiersprache muss zum jeweiligen Einsatzbereich passen. [94]

Erfahrung Der Autor der vorliegenden Arbeit soll bereits Erfahrung mit der jeweiligen Programmiersprache haben, d.h. bereits nicht-triviale Software damit entwickelt haben. Diese Erfahrung sollte nicht zu weit zurückliegen.[95]

Kompatibilität Die Programmiersprache muss zu den bereits gesetzten und ausgewählten Technologien kompatibel sein.[96]

[94] Bei Projekten mit einem Hintergrund in der Data Science und im Machine Learning wären etwa Python und R sinnvolle Varianten. Bei der Entwicklung hochperformanter Serversysteme kommen eher C, C++, Rust und Go zum Einsatz.

[95] Kandidaten hierfür wären Java, Python, JavaScript, Go und R.

[96] Dies sind je nach Komponente u.a. TensorFlow, HTTP, JSON, Messaging.

Perspektive Es soll eine Programmiersprache gewählt werden, deren Alter und Popularität darauf hindeuten, dass sie in den nächsten fünf bis zehn Jahren noch gebräuchlich sein wird.[97] Dadurch kann die Anwendung über längere Zeit gewartet und muss nicht schon bald in grossen Teilen in einer anderen Programmiersprache neu entwickelt werden.

8.1.1 Modellkomponenten `body_part`, `joint_detection`, `ratingen_score`

Da die Modelle in älteren Versionen von TensorFlow implementiert worden sind, lassen sich die Modelldaten nicht einfach für die Ausführung in einer anderen Laufzeitumgebung exportierten. Darum sollen sie in einem Container ausgeführt werden, der ihre ursprüngliche Laufzeitumgebung abbildet (siehe 2.1 *Bestehende Modelle*, Seite 14 und 3.2 *Austauschbarkeit von Modellen*, Seite 42).

Der Code zur Beschreibung der Modelle soll darum ebenfalls beibehalten werden. Somit kommt für die Komponenten, welche die Machine-Learning-Modelle anbieten, Python zum Einsatz.[98] Somit ist Python 3.6 für die Komponenten `body_part` und `joint_detection`; und Python 3.5 für die Komponente `ratingen_score` gesetzt.

8.1.2 Neue Komponente `orchestrator`

Für den `orchestrator` gelten andere Anforderungen als für die Modellkomponenten. Da für diese Komopnente noch kein Code existiert, kann diese komplett von Grund auf und ohne Technologieeinschränkungen aufgrund geleisteter Vorarbeiten entwickelt werden. Als Schnittstelle muss nicht nur eine Message-Queue unterstützt, sondern auch HTTP angeboten werden. Die grösste Schwierigkeit beim `orchestrator` besteht darin, dass dieser mit mehreren Anfragen gleichzeitig umgehen können muss, ohne dass die nebenläufigen Vorgänge (Gelenkextraktion und -scoring) dabei durcheinander geraten dürfen. Da

[97] Der *Lindy Effect* (Goldman, 1964) besagt, dass sich die Lebenserwartung nicht-verderblicher Sachen wie z.B. Technologien proportional zu deren Alter verhält. Rust, das am 15. Mai 2015 in Version 1.0 erschienen ist, und seither an Popularität gewinnt, hätte demnach noch eine Mindestlebenserwartung von fünf Jahren. Bei Go, das am 28. März 2012 in Version 1.0 herausgegeben worden ist, dürften es somit noch mindestens acht Jahre sein. Bei Nim, das am 23. September 2019 in Version 1.0 erschienen ist, darf man hingegen nur mit einer Mindestlebenserwartung von neun Monaten rechnen.

[98] Die Modelle `body_part` und `joint_detection` sind in einer Umgebung mit Python 3.6 entwickelt worden. Das neuere Modell `ratingen_score` wurde auf einer Umgebung trainiert, welche die etwas ältere Version Python 3.5 erforderte. Python 3.6 unterscheidet sich v.a. durch neue Features von Python 3.5, wovon *PEP 498: formatted string literals* das einzige Feature ist, das der Autor dieser Zeilen regelmässig verwendet (siehe `https://docs.python.org/3/whatsnew/3.6.html` für detaillierte Release-Informationen).

der `orchestrator` ohne vorgelagerten Load-Balancer auskommen und nur mit einer einzigen Instanz laufen soll, muss sich dieser selber um die Nebenläufigkeit (Ausführung in mehreren Threads) kümmern – und sollte eine möglichst hohe Laufzeitperformance bieten.

Von den Programmiersprachen, welche die Kriterien von 8.1 *Wahl der Programmiersprache(n)*, Seite 101 erfüllen[99], hat Go im Bezug auf die Anforderungen für den `orchestrator` am meisten zu bieten. Zum einen verfügt Go über ein mächtiges HTTP-Package (für Client und Server) in der Standardbibliothek[100]. Viel wichtiger ist allerdings das Concurrency-Modell von Go, das auf dem Konzept von *Communicating Sequential Processes* (CSP) basiert und leichtgewichtige Threads (*goroutines*) als primäres Sprachkonstrukt mit dem Schlüsselwort `go` starten kann (Donovan & Kernighan, 2015, Kapitel 8).

CSP kann als prozessinterner Messaging-Mechanismus verstanden werden, bei dem verschiedene Unterprozesse (in Go: *goroutines*) über Queues (in Go: *channels*) Nachrichten untereinander austauschen. Die Vorteile der Messaging-Architektur, die im zweiten und dritten Kapitel besprochen worden sind (siehe 2.3 *Integrationsvarianten*, Seite 19; 3.1.3 *Variante 3: Messaging zwischen Modellkomponenten*, Seite 36 und 3.1.4 *Variante 4: Messaging, synchron und asynchon*, Seite 39), können somit auch innerhalb einer einzelnen Komponente genutzt werden. Die Probleme, mit denen man bei der Verwendung von Threads und geteiltem Zustand (*shared state*) zu kämpfen hat, treten beim korrekten Gebrauch von CSP nicht auf, da ein Datenobjekt zu einem bestimmten Zeitpunkt nur auf einer Seite des gemeinsamen Kanals – und damit nur in einem Thread – sein kann.[101]

Das `select/case`-Statement, das ähnlich wie `switch/case` aufgebaut ist, erlaubt es verschiedene Kanäle auf eine eingehende Nachricht zu prüfen, und mit entsprechendem Code darauf zu reagieren. Werden die externen Message-Queues intern mit Channels abgebildet, können die eingehenden Ergebnisse der verschiedenen Modellkomponenten so in den richtigen Kontext (HTTP-Anfrage) weitergeleitet werden.

Weiter bietet Go als kompilierte Programmiersprache eine ansprechende Laufzeitperformance, die derjenigen von Skriptsprachen weit überlegen ist.[102]

Aus diesen Gründen soll Go für die Umsetzung der `orchestrator`-Komponente verwendet werden.

---

[99] Java, Python, JavaScript, Go, und R

[100] siehe `https://golang.org/pkg/net/http/` (abgerufen am 23.05.2020)

[101] Rob Pike bringt diesen Ansatz mit seinem Go-Proverb *«Don't communicate by sharing memory, share memory by communicating.»* auf den Punkt (Pike, 2015, 2:45).

[102] Siehe etwa den Vergleich zwischen Go und Python 3 auf `https://benchmarksgame-team.pages.debian.net/benchmarksgame/fastest/go-python3.html` (abgerufen am 23.05.2020).

8.1.3 Evaluation

Bei der Evaluation werden Röntgenbilder mithilfe des Prototyps gescored. Auf die ermittelten Scores sollen anschliessend verschiedene statistische Metriken angewendet werden. Für solche Aufgaben kommen in der Regel Python oder R zum Einsatz. R hat den Vorteil, dass es eine Vielzahl statistischer Operationen ohne Zusatzpakete unterstützt. Für Python werden Zusatzpackages wie NumPy, Pandas und SciPy benötigt. Letztere haben im Kontext der vorliegenden Arbeit wiederum den Vorteil, dass der Autor schon gut mit ihnen vertraut ist.[103]

Da Python und NumPy bereits bei den Modellkomponenten zum Einsatz kommen, ist es durchaus angebracht auch die Evaluation mit diesen Werkzeugen zu bewerkstelligen, zumal so bereits von bestehendem Wissen profitiert werden kann.

Somit soll Python für die Evaluation verwendet werden.

8.2 Wahl der Message-Queue

Für Messaging bieten sich verschiedene Protokolle und fertige Softwarelösungen an. AMQP, MQTT, Apache Kafka und ZeroMQ sind allesamt populäre Lösungen, die Anbindungen für verschiedene Programmiersprachen unterstützen. Da eine umfassende Evaluation aller genannter Möglichkeiten den Rahmen der vorliegenden Arbeit sprengen würde, wird hier auf eine systematische Untersuchung verzichtet. Stattdessen soll nur zwischen den beiden (standardisierten) Protokollen AMQP und MQTT abgewägt werden. Die Wahl der spezifischen Implementierung für den Prototyp kann so vor dem Hintergrund erfolgen, dass diese für den Produktiveinsatz bei Bedarf ausgetauscht werden könnte, ohne dass die Schnittstellen dazu umfassend geändert werden müssten.

AMQP wurde als Messaging-Protokoll speziell für die Bedürfnisse der Finanzindustrie – Sicherheit, Zuverlässigkeit, Interoperabilität, Standardisierung, Offenheit (*AMQP is the Internet Protocol for Business Messaging*, o. J.) – entwickelt. Es ermöglicht Messaging auf Basis eines *Message Brokers*, der mehrere *Message Queues* anbietet. Der Broker empfängt Nachrichten von einem *Publisher* und leitet diese an einen oder mehrere *Consumer* weiter. Die Weiterleitung (*Routing*) wird mittels *Exchanges* bewerkstelligt (*AMQP 0-9-1 Model Explained*, o. J.).

MQTT ist ein leichtgewichtiges Messaging-Protokoll, das speziell für die Bedürfnisse in Anwendungen mit schwacher Hardware und unzuverlässigen Netzwerkverbindungen

[103] Im HSLU-Pflichtmodul *Statistics for Data Science* kommt Python mit den Libraries NumPy, Pandas und SciPy zum Einsatz. Die Verwendung von R ist nur als Alternative vorgesehen.

geschaffen worden ist – sprich für Internet-of-Things-Anwendungen. Im Gegensatz zu AMQP, das über Konstrukte auf Protokollebene wie Exchanges und Queues verfügt, beschränkt sich MQTT auf Publish/Subscribe-Mechanismen. (Roy, 2017, S. 178)

RabbitMQ unterstützt sowohl AMQP[104] als auch MQTT, und bietet somit maximale Flexibilität. Beschränkungen in der Grösse der Payloads sind weder in AMQP (16 Exabytes), RabbitMQ (2 Gigabytes) noch MQTT (256 Megabyte) für das vorliegende Projekt relevant (Roy, 2017, S. 180), zumal sich die Dateigrössen der verwendeten Röntgenbilder in einer Grössenordnung von etwa einem Megabyte bewegen. Wird MQTT im Kontext von RabbitMQ verwendet, können auch MQTT-Übertragungen von gewissen AMQP-Features profitieren, die von RabbitMQ unterstützt werden. Subtile Unterschiede zwischen den beiden Protokollen können aber dennoch zu Problemen in der Interoperabilität führen, wenn diese in der gleichen Messaging-Architektur eingesetzt werden (Roy, 2017, S. 179). Da der zu entwickelnde Prototyp eher in einem Rechenzentrum mit zuverlässigem Netzwerk und rechenstarken Servern zum Einsatz kommt als in einem IoT-Setting, dürfte die AMQP-Variante besser für dessen Einsatzbereich geeignet sein. Auch bei den Merkmalen im Bereich der Sicherheit hat AMQP mehr zu bieten als MQTT (*AMQP vs MQTT*, o. J.).

Für den Prototyp soll darum RabbitMQ zum Einsatz kommen und AMQP als Protokoll verwenden.[105]

8.3 Schnittstellen

In Tabelle 6, Seite 106 sind sämtliche Schnittstellen zwischen den einzelnen Komponenten aufgelistet.

[104] RabbitMQ unterstützt die Version 0.9.1 des AMQP-Protokolls standardmässig, die neuere Version 1.0.0 jedoch nur per Plugin. Auf die Unterschiede zwischen den beiden Protokollversionen soll hier nicht eingegangen werden. Diese scheinen jedoch beträchtlich und von einer Kontroverse geprägt zu sein (Hintjens, 2012), sodass AMQP 0.9.1 sich immer noch grosser Verbreitung erfreut.

[105] Mögliche AMQP-Alternativen zu RabbitMQ wären etwa OpenAMQ, StormMQ, Apache Qpid und Red Hat Enterprise MRG.

Sender/Empfänger	Channel/Queue	MIME-Type	Struktur
client → orchestrator	HTTP Port 8080 (POST)	image/jpeg	Form Field 'xray'
orchestrator → body_part	body_part	application/-octet-stream	Byte Stream
body_part → orchestrator	body_part_response	application/json	{"body_part": [string], "probability": [float]}
orchestrator → joint_detection	joint_detection	application/json	{"joint_name": [string], "xray": [jpeg_b64]}
joint_detection → orchestrator	joint_detection_error	application/json	{"error": [string], "joint_name": [string]}
joint_detection → ratingen_score	ratingen_score	application/json	{"joint_name": [string], "joint_image": [jpeg_b64]}
ratingen_score → orchestrator	ratingen_score_error	application/json	{"error": [string], "joint_name": [string]}
ratingen_score → orchestrator	scores	application/json	{"joint_name": [string], "score": [int] }
orchestrator → client	HTTP Port 8080	application/json	{"scores": { "mcp1": [int], "mcp2": [int], "mcp3": [int], "mcp4": [int], "mcp5": [int], "pip1": [int], "pip2": [int], "pip3": [int], "pip4": [int], "pip5": [int] } }

Tabelle 6: Die Schnittstellen zwischen den einzelnen Komponenten im Überlick. Der Datentyp jpeg_b64 bezeichnet ein base64-kodiertes JPEG-Bild, der technisch als Zeichenkette abgelegt wird.

8.4 Abgabe

Dieser Abschnitt gibt einen Überblick über die abgegebenen Artefakte, die nicht in den Bericht (vorliegendes Dokument) eingefügt worden sind.

Das Deckblatt (`Zusatz/Deckblatt.pdf`), bzw. die ersten drei Seiten des Berichts, wurden ausgedruckt, unterschrieben (Eidesstattliche Erklärung, Bestätigung der Abgabe) und wieder eingescannt. Auf eine Integration in das Hauptdokument wurde verzichtet, damit Index und Schriftbild erhalten bleiben.

Im Arbeitsjournal (`Zusatz/Arbeitsjournal.pdf`) sind die einzelnen Aufwände auf halbe Stunden gerundet nach Bereich – Projekt[administation], Recherche, Dokumentation, Umsetzung – rapportiert. Mithilfe eines `awk`-Skripts können die Aufwände nach Bereich ausgewertet werden. Die Aufwände sind in Tabelle 7, Seite 107 aufgelistet.

Bereich	Aufwand in Stunden
Umsetzung	118.5
Dokumentation	100.5
Recherche	41.0
Projekt	30.5
Präsentation	6.0
Video	4.0
Total	300.5

Tabelle 7: Die Aufwände aus dem Arbeitsjournal nach Bereich gegliedert und insgesamt.

Das Web-Abstract (`Web-Abstract.pdf`) stellt die vorliegende Arbeit zusammengefasst auf drei Seiten vor.

Im Pitching-Video (`Pitching-Video.mp4`) wird ebenfalls die vorliegende Arbeit vorgestellt, hier im Rahmen eines kurzen Screencasts (Demo des Prototypen über die Web-Oberfläche) mit Erläuterungen zum Projekt in der Tonspur.[106]

Das Verzeichnis `Zusatz/deepxray/` enthält Teile des Quellcoderepositories, das im Verlauf der Umsetzung angewachsen ist und komplett auf dem BitBucket-Server des Auftraggebers vorliegt. Evaluationsdaten (Röntgenbilder und Metadaten) wurden aus Gründen des Datenschutzes für die Abgabe (mit wenigen Ausnahmen) aus diesem entfernt. Das Verzeichnis enthält (in alphabetischer Reihenfolge):

[106] Der verwendete Soundtrack *Microchip* von Jason Farnham ist gemeinfrei. Dies ist auch in den Metadaten der Videodatei vermerkt.

`archive/` Code, der zum Kennenlernen, Testen und – im Falle von `ratingen_score` – Trainieren der Modelle verwendet worden und nur der Vollständigkeit halber hier abgelegt ist.

`body_part/` Die Modellkomponente `body_part` inklusive Modelldaten.

`demo_data/` In diesem Verzeichnis sind einige ausgesuchte Bilder enthalten, womit der Prototyp getestet werden kann.

`docker-compose.yaml` Diese Datei enthält die Servicedefinitionen für den Prototyp, der mit `docker-compose up` aufgestartet werden kann (siehe auch `README.md`).

`evaluation/` Hier ist der Code für das Scoring und die Evaluationsmetriken abgelegt. Die Ergebnisse des Scorings sind in einer CSV-Datei enthalten.

`gui/` Die Web-Oberfläche, die im Pitching-Video gezeigt wird.

`joint_detection/` Die Modellkomponente `joint_detection` inklusive Modelldaten (für zehn Modelle).

`orchestrator/` Der Code für die zentrale `orchestrator/`-Komponente.

`ratingen_score/` Die Modellkomponente `ratingen_score` inklusive Modelldaten.

`README.md` Anweisungen, wie der Prototyp gestartet werden kann (knapp gehalten).

`test/` End-to-End-Testfälle in Go, Python und Bash.

`test_data/` Python-Skripts zur Aufbereitung der Evaluationsdaten (eigentliche Testdaten entfernt).

`test_data_selection/` Python-Skripts zur Selektion der Evaluationsdaten (wiederum ohne Testdaten).

Die Struktur der Verzeichnisse, die Quellcode enthalten, sind in den Kapiteln 5 *Realisierung*, Seite 60 und 6.1 *Technische Evaluation*, Seite 80 erläutert (mit Ausnahme von `archive/`).

Literatur

AMQP 0-9-1 Model Explained. (o. J.). Pivotal. Zugriff auf https://www.rabbitmq.com/tutorials/amqp-concepts.html (Zugriff am 21.05.2020)

AMQP is the Internet Protocol for Business Messaging. (o. J.). AMQP.org. Zugriff auf http://www.amqp.org/about/what (Zugriff am 21.05.2020)

AMQP vs MQTT. (o. J.). EDUCBA. Zugriff auf https://www.educba.com/amqp-vs-mqtt/ (Zugriff am 21.05.2020)

Armstrong, J. (2003). *Making Reliable Distributed Systems in the Presence of Software Errors* (Unveröffentlichte Dissertation). Mikroelektronik och informationsteknik.

Armstrong, J. (2013). *Programming Erlang* (Second Aufl.). Pragmatic Bookshelf.

Bui-Palsulich, T. & Compton, E. (2019). *Using Go Modules.* The Go Blog. Zugriff auf https://blog.golang.org/using-go-modules (Zugriff am 24.05.2020)

Conway, M. E. (1968). How do committees invent. *Datamation*, *14* (4), 28-31.

Dijkstra, E. W. (2001). *To the members of the Budget Council.* University of Texas at Austin. Zugriff auf https://www.cs.utexas.edu/users/EWD/transcriptions/OtherDocs/Haskell.html (Zugriff am 17.05.2020)

Donovan, A. A. A. & Kernighan, B. W. (2015). *The Go Programming Language.* Addison-Wesley.

Elliot, E. (2020). *Composing Software.* Leanpub. Zugriff auf https://leanpub.com/composingsoftware

Fielding, R., Lafon, Y. & Reschke, J. (2014, June). *Hypertext Transfer Protocol (HTTP/1.1): Range Requests* (RFC Nr. 7233). RFC Editor. Internet Requests for Comments. Zugriff auf http://www.rfc-editor.org/rfc/rfc7233.txt (http://www.rfc-editor.org/rfc/rfc7233.txt)

Fielding, R. & Reschke, J. (2014, June). *Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing* (RFC Nr. 7230). RFC Editor. Internet Requests for Comments. Zugriff auf http://www.rfc-editor.org/rfc/rfc7230.txt (http://www.rfc-editor.org/rfc/rfc7230.txt)

Fielding, R. T. & Taylor, R. N. (2000). *Architectural styles and the design of network-based software architectures* (Bd. 7). University of California, Irvine Irvine.

Gaskill, B. (2018). *ONNX: the Open Neural Network Exchange Format.* Linux Journal. Zugriff auf https://www.linuxjournal.com/content/onnx-open-neural-network-exchange-format (Zugriff am 29.04.2020)

Goldman, A. (1964, June). Lindy's Law. *New Republic*, 34-35.

Géron, A. (2019). *Hands-On Machine Learning with Scikit-Learn, Keras & TensorFlow* (Zweite Aufl.). O'Reilly.

Hale, J. (2018). *7 data types: A better way to think about data types for machine learning.* Towards Data Science. Zugriff auf https://towardsdatascience.com/7-data-types-a-better-way-to-think-about-data-types-for-machine-learning-939fae99a689 (Zugriff am 13.04.2020)

Hintjens, P. (2012). *Burning Down the House.* Hintjens.com. Zugriff auf http://hintjens.com/blog:28 (Zugriff am 21.05.2020)

Hohpe, G., Woolf, B., Brown, K. & Fowler, M. (2004). *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley.

Institut d'électronique et d'informatique Gaspard-Monge (IGM). (o. J.). *Les services Web – Définition.* Autor. Zugriff auf http://www-igm.univ-mlv.fr/~dr/XPOSE2004/woollams/definition.html (Zugriff am 11.04.2020)

Mason, R. (2011). *How REST replaced SOAP on the Web: What it means to you.* InfoQ. Zugriff auf https://www.infoq.com/articles/rest-soap/ (Zugriff am 11.04.2020)

Mauersberger, L. (2019). *Microservices: What They Are and Why Use Them.* LeanIX. Zugriff auf https://www.leanix.net/en/blog/a-brief-history-of-microservices (Zugriff am 11.04.2020)

McHugh, M. L. (2012, 10). *Interrater reliability: the kappa statistic.* Biochemia Medica. Zugriff auf https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3900052/ (Zugriff am 03.05.2020)

ONNX. (2019). https://lfai.foundation/projects/onnx/. (Zugriff am 21.02.2020)

O'Toole, G. (2014). *If Your Only Tool Is a Hammer Then Every Problem Looks Like a Nail.* Tracing Quotations. Zugriff auf https://quoteinvestigator.com/2014/05/08/hammer-nail/ (Zugriff am 17.05.2020)

Parbel, M. (2019). *Machine Learning: Linux Foundation übernimmt das ONNX-Projekt.* Heise Online. Zugriff auf https://www.heise.de/developer/meldung/Machine-Learning-Linux-Foundation-uebernimmt-das-ONNX-Projekt-4586969.html (Zugriff am 29.04.2020)

Pike, R. (2013). *Concurrency is not Parallelism.* Heroku. Zugriff auf https://vimeo.com/49718712 (Zugriff am 03.05.2020)

Pike, R. (2015). *Go Proverbs.* The Go Programming Language. Zugriff auf https://www.youtube.com/watch?v=PAAkCSZUG1c (Zugriff am 24.05.2020)

Protocol Buffer Basics: Go. (2020). Google Developers. Zugriff auf https://developers.google.com/protocol-buffers/docs/gotutorial (Zugriff am 22.05.2020)

PureBasic Download. (2020). Fantaisie Software. Zugriff auf https://www.purebasic.com/german/download.php (Zugriff am 29.04.2020)

Rohrbach, J. (2017, 12). *Arthritis Net.* https://github.com/janickrohrbach/arthritis-net/blob/master/doc/project.pdf.

Rohrbach, J., Reinhard, T., Sick, B. & Dürr, O. (2019). Bone Erosion Scoring for Rheumatoid Arthritis with Deep Convolutional Neural Networks. *Computers & Electrical Engineering*, *78*, 472-481. Zugriff auf http://www.sciencedirect.com/science/article/pii/S0045790618329409

Roy, G. M. (2017). *RabbitMQ in Depth.* Manning.

Shung, K. P. (2018). *Accuracy, Precision, Recall or F1?* Zugriff auf https://towardsdatascience.com/accuracy-precision-recall-or-f1-331fb37c5cb9 (Zugriff am 16.05.2020)

Techopedia. (2017). *Web Service.* Autor. Zugriff auf https://www.techopedia.com/definition/25301/web-service (Zugriff am 11.04.2020)

TechTerms. (2017). *Web Service.* Autor. Zugriff auf https://techterms.com/definition/web_service (Zugriff am 11.04.2020)

Walsh, D. (2016). *Docker versus Systemd.* Red Hat Summit. Zugriff auf https://www.youtube.com/watch?v=93VPog3EKbs (Zugriff am 26.05.2020)

Warrens, M. J. (2011). Weighted kappa is higher than Cohen's kappa for tridiagonal agreement tables. *Statistical Methodology*, *8* (2), 268-272. Zugriff auf http://www.sciencedirect.com/science/article/pii/S1572312710000973

Xu, F. (2019). *Announcing ONNX Runtime 1.0.* Microsoft Open Source Blox. Zugriff auf https://cloudblogs.microsoft.com/opensource/2019/10/30/announcing-onnx-runtime-1-0/ (Zugriff am 29.04.2020)

Zheng, A. (2015). *Evaluating Machine Learning Models.* O'Reilly.

Abbildungsverzeichnis

Tabellenverzeichnis

Formelverzeichnis