Universität Leipzig
Institut für Informatik
Studiengang Informatik, B.Sc.

OCR-Post-Korrektur auf historischen Texten in deutscher Sprache

Bachelorarbeit

Ole David Borchardt

1. Gutachter: Prof. Dr. Martin Potthast

Datum der Abgabe: 28. Februar 2022

Erklärung

Hiermit versichere ich, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Leipzig, 28. Februar 2022

.....
Ole David Borchardt

Zusammenfassung

Im Zuge der immer weiter voranschreitenden Digitalisierung wird der digitale Zugang zu analog erstellten Dokumenten immer wichtiger. Diesen Zugang ermöglicht neben dem Einscannen von Dokumenten vor allem auch die optische Zeichenerkennung (OCR), denn nur so werden die Dokumente maschinell weiterverarbeitbar und durchsuchbar. OCR-Engines haben allerdings immer noch mit Fehlern zu kämpfen, die in vielen Anwendungsfällen eine aufwendige manuelle Korrektur erzwingen. Ein Lösungsansatz dafür ist die OCR-Post-Korrektur, mit deren Hilfe von OCR-Engines erkannte Texte im Nachhinein korrigiert werden. Dazu gibt es manuelle Ansätze mithilfe von Crowdsourcing oder automatische Ansätze, die in der Regel Sprachmodelle nutzen.

Im Rahmen dieser Arbeit sollen die Fehler auf einem Korpus von wissenschaftlichen Dokumenten um das Jahr 1900 korrigiert werden. Der Datensatz umfasst Scans von Dokumenten sowie die Ausgaben der OCR-Engine ABBYY FineReader 10, die als XML-Dokumente vorliegen. In der Arbeit beschränke ich mich auf Dokumente, die in deutscher Sprache vorliegen und die den Großteil des Korpus ausmacht.

In meiner Arbeit annotiere ich einen Teil der Dokumente mithilfe von Crowdsourcing. Dafür filtere ich den Datensatz und ziehe eine Stichprobe aus allen Dokumenten. Darüber hinaus entwerfe ich Webseiten, mithilfe derer die Dokumente annotiert werden. Danach gehe ich auf die OCR-Post-Korrektur mithilfe verschiedener automatischer Modelle ein. Dazu beschreibe ich Details des Datensatzes und die Grundlagen für die Sprachmodelle und das Crowdsourcing. Im weiteren Verlauf bespreche ich die Implementierung der Modelle und evaluiere sie.

Da mithilfe der manuellen Korrektur nur wenig Daten annotiert wurden, nutze ich stattdessen für das Training einiger Modelle den Datensatz eines Wettbewerbs zur OCR-Post-Korrektur. Ich evaluiere und vergleiche die Modelle sowohl auf dem Evaluierungsdatensatz des Wettbewerbs als auch auf den annotierten Daten des Archivs.

Die Evaluierung der Modelle zeigt, dass sie zwar Verbesserungen auf dem Evaluierungsdatensatz des Wettbewerbs erzielen können, allerdings nicht auf den eigentlich zu korrigierenden Korpus übertragbar sind.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Optische Zeichenerkennung (OCR) macht Schrift auf Bildern für Computer lesbar und ermöglicht damit die maschinelle Verarbeitung alter Bücher und Schriften. Daher spielt sie eine zentrale Rolle, um Vorteile, die eine Digitalisierung von Archiven mitbringen kann, beispielsweise eine bessere Durchsuchbarkeit, umzusetzen. Allerdings ist die OCR nicht perfekt und die Korrektheit der Ergebnisse hängt stark von den Ausgangsdaten ab. Das Layout eines Dokumentes, der Schrifttyp oder auch die Qualität der Scans können den OCR-Engines Probleme bereiten. Ein Beispiel dazu ist in Abbildung 1.1 zu sehen.

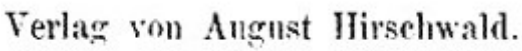


Abbildung 1.1: Erkannter Text: „Verlas von August Ilirschwald.“

Die Effekte solcher Fehler werden von Strien et al. (2020) untersucht, indem sie Algorithmen zu verschiedenen Aufgabenstellungen der natürlichen Sprachverarbeitung auf von OCR-Engines erkannte Texte anwenden. Unter den Aufgaben sind *Information Retrieval*, *Named Entity Recognition* und *Topic Modelling*. Die Texte entnehmen sie den Artikeln aus einem Datensatz historischer Zeitungen. Sie teilen die Artikel in vier Qualitätsstufen auf, indem sie für jeden Artikel die Levenshtein-Distanz mit der korrigierten Version des jeweiligen Artikels berechnen. Anschließend vergleichen sie die Abweichungen der Ergebnisse auf den Artikeln der verschiedenen Qualitätstufen und den korrigierten Artikeln. Sie kommen zu dem Schluss, dass die Qualität der erkannten Texte Auswirkungen in allen Aufgaben hat. Häufig nehmen die Abweichungen mit schlechterer OCR-Qualität zu. Daher ist die Nachbearbeitung der Ergebnisse nötig, gerade wenn es sich um alte Texte handelt.

Um erkannte Texte zu verbessern, gibt es eine Reihe von Ansätzen für die Nachbearbeitung (OCR-Post-Korrektur). Das Ziel ist dabei, zu einer Sequenz

von erkannten Token eine Sequenz von Token zu finden, die dem Text in dem jeweiligen Dokument entspricht. Die Anzahl der erkannten Token kann dabei von der Anzahl der tatsächlichen Token abweichen. Fehler werden „Real-Word-Error“ gennant, falls das betroffene Wort existiert, andernfalls werden sie „Non-Word-Error“ genannt (Nguyen et al. 2021, S.4). Die generelle Vorgehensweise bei der OCR-Post-Korrektur ist in Abbildung 1.2 zu sehen.

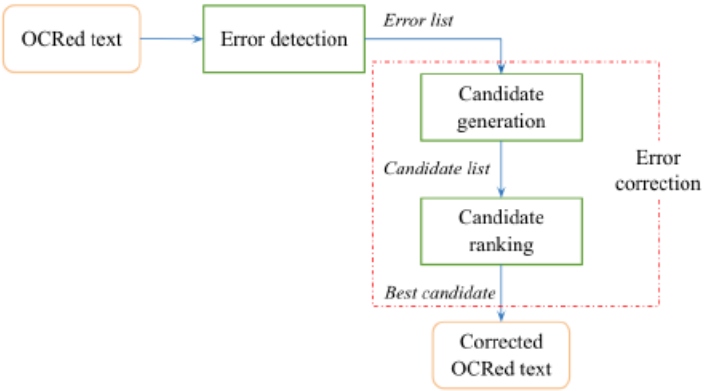


Abbildung 1.2: Generelles Verfahren der OCR-Post-Korrekur beschrieben von Nguyen et al. (2021)

Wie genau die einzelnen Schritte in Abbildung 1.2 funktionieren, ist stark von den verwendeten Verfahren und dem Automatisierungsgrad abhängig. Mögliche Ansätze reichen von manuellen über semi-automatische bis hin zu vollständig automatischen Systemen. Beispielsweise kann es sein, dass bei einem semi-automatischen Verfahren mehrere Kandidaten generiert werden und ein Mensch anschließend den besten auswählt. Das Ziel ist allerdings vollautomatische Systeme zu nutzen, da manuelle oder semi-automatische Ansätze teure menschliche Arbeit benötigen. Schätzungen zu den Kosten einer manuellen Korrektur liegen bei ungefähr einem Euro pro Seite und damit bei mehreren Hundert Euro pro Buch.[1]

Das Ziel meiner Arbeit ist die Entwicklung von Ansätzen zur Korrektur von OCR-Fehlern in den Dokumenten des „Virtual Laboratory: Essays and Resources on the Experimentalization of Life“. Das *Virtual Laboratory* ist ein Projekt, das ein Forum für Wissenschaftler und Interessierte an der Geschichte der Lebenswissenschaften bietet. Das zugrundeliegende Archiv besteht aus wissenschaftlichen Büchern, Zeitschriften und anderen Medien aus dem 19. und 20. Jahrhundert. Es enthält viele verschiedene Dokumente in unterschied-

[1] `http://www.impact-project.eu/about-the-project/concept/` (abgerufen am 14.01.2022)

lichen Formatierungen und Schriften. Außerdem enthält es zu den meisten Dokumenten hochauflösende Bilder.

Vergleichbare Datensätze, die von Nguyen et al. (2021) erwähnt werden, bieten nicht die gleiche Diversität. So fokussiert sich GT4HistOCR (Springmann et al. 2018) beispielsweise auf Frakturschrift, der TextBerg Corpus (Bubenhofer et al. 2015) enthält nur Jahrbücher des Schweizer Alpen-Clubs. Der Datensatz des Wettbewerbs zur OCR-Post-Korrektur der *International Conference on Document Analysis and Recognition* (ICDAR) und der RETAS-Datensatz[2] enthalten keine Bilder zu den von OCR-Engines erkannten Texten.

Im Rahmen meiner Arbeit implementiere und evaluiere ich einen manuellen Ansatz und weitere automatische Ansätze für die OCR-Post-Korrektur. In Kapitel 2 umreiße ich den Stand der Forschung und ordne meine Ansätze ein. In Kapitel 3 gehe ich auf Grundlagen der von mir gewählten Verfahren ein. In Kapitel 4 bespreche ich die Annotation der Dokumente aus dem *Virtual Laboratory*, um in Kapitel 5 die Implementierung und die Ergebnisse meiner automatischen Korrekturverfahren zu besprechen.

Als manuellen Ansatz lasse ich einen Teil des Archivs mithilfe von Amazon MTurk annotieren. Auf den annotierten Daten evaluiere ich automatische Modelle zur Erkennung und Korrektur von OCR-Fehlern. Für die automatischen Modelle verwende ich als Vergleichsmodell einen wörterbuchbasierten Algorithmus und trainiere eigene Sprachmodelle auf den Daten ICDAR-Wettbewerbs. Die Performanz der automatischen Modelle bewerte ich auf dem Evaluierungsdatensatz des Wettbewerbs und den in der manuellen Korrektur annotierten Daten.

Die Auswertung auf beiden Datensätzen zeigt, dass meine Sprachmodelle auf den Evaluierungsdaten des ICDAR-Wettbewerbs bessere Ergebnisse liefern als der wörterbuchbasierte Ansatz. Allerdings sind sie nicht einfach auf die Texte aus dem Archiv des *Virtual Laboratory* übertragbar, unter anderem, da sich die Datensätze bezüglich der Häufigkeit von Fehlern stark unterscheiden.

[2] `http://ciir.cs.umass.edu/downloads/ocr-evaluation/` (abgerufen am 24.01.2022)

Kapitel 2

Verwandte Arbeiten

Herangehensweisen an die OCR-Post-Korrektur reichen von der manuellen, über die semi-automatische bis zur automatischen Korrektur. Dabei entsprechen semi-automatische Verfahren in der Regel automatischen Verfahren mit einer weniger strengen Auswahl der Wörter, die für die Korrektur infrage kommen, weshalb auch Nguyen et al. (2021, S.5) sie letztendlich in einer Kategorie zusammenfassen.

Die manuelle Korrektur erfolgt in der Regel über *Crowdsourcing* und stützt sich auf Freiwillige. Beispiele sind Trove (Holley 2009), ReCAPTCHA (Von Ahn et al. 2008) oder Digitalkoot (Chrons und Sundell 2011). Damit *Crowdsourcing* Erfolg hat müssen Freiwillige motiviert werden, dem Projekt Zeit zu schenken und es muss eine möglichst automatische Kontrolle der Antworten geben um mangelhafte Antworten herauszufiltern und automatisiert eine große Menge an Wörtern korrigieren zu können.

Um Freiwillige zu motivieren hat Digitalkoot die Korrekturaufgaben in einfache Spiele integriert (*Gamification*). ReCAPTCHA gewährt einem Webseitenbesucher erst nach der Lösung einer Aufgabe Zugang zu der Webseite. Die Entwickler von reCAPTCHA führen außerdem als Motivation für Freiwillige an, dass diese mit ihren Antworten zur Digitalisierung von menschlichem Wissen und damit einem sinnvollen Projekt beitragen, insbesondere im Vergleich zu anderen CAPTCHAs (Von Ahn et al. 2008, S.3). Darüber hinaus profitierten beide Systeme von viel Aufmerksamkeit der Öffentlichkeit (Von Ahn et al. 2008, S.3; Chrons und Sundell 2011, S.6).

Die Bewertung der Ergebnisse erfolgt in beiden Systemen durch das Vergleichen von Antworten der Freiwilligen und das Generieren von Verifikationsaufgaben aus gesicherten Antworten. So konnten beide Systeme eine sehr hohe Genauigkeit bei der Korrektur erreichen (Chrons und Sundell 2011, S.5; Von Ahn et al. 2008, S.3).

Nguyen et al. (2021) teilen die automatischen Korrekturansätze in verschiedene Kategorien ein. Sie unterscheiden zunächst zwischen Verfahren, die einzelne Wörter isoliert betrachten und Verfahren, die den Kontext eines Wortes bei der Fehlerkorrektur beachten (kontextbasiert). Erstere können „Real-Word-Errors“ nicht korrigieren.

Zu den Verfahren, die Wörter isoliert betrachten, gehören Programme, die verschiedene Ausgaben von OCR-Engines kombinieren. Lund, Kennard und Ringger (2013) kombinieren die Ausgaben einer OCR-Engine auf verschieden vorverarbeiteten Scans der gleichen Dokumente, Volk, Marek und Sennrich (2010) nutzen verschiedene OCR-Engines auf den gleichen Scans. Verfahren dieser Art sind in der Korrektur allerdings auf die Ergebnisse der OCR-Engines beschränkt, können also nur Korrekturen liefern, wenn das gesuchte Wort in mindestens einer Ausgabe vorhanden ist (Nguyen et al. 2021).

Außerdem gibt es lexikalische Verfahren, wie das von Estrella und Paliza (2014), die ein umfassendes und stark auf ihren Datensatz spezialisiertes Wörterbuch nutzen. Die Fehlerkorrektur erfolgt in der Regel über Distanzmaße wie die Levenshtein Distanz. Darüber hinaus gibt es Verfahren, die stochastische Modelle nutzen, häufig in Kombination mit Wörterbüchern und den Ansatz themenspezifischer Sprachmodelle, bei denen Wahrscheinlichkeiten von Token auf Basis eines Themas miteinbezogen werden.

In der Kategorie der kontextbasierten OCR-Post-Korrektur gibt es Sprachmodelle, merkmalbasierte Machine Learning Modelle und *Sequence-to-Sequence*-Modelle (Seq2Seq). Unter den Seq2Seq-Modellen gibt es sowohl statistische maschinelle Übersetzung (SMT) als auch neuronale maschinelle Übersetzung (NMT). SMT nutzt Systeme, die aus einem Sprach- und einem Übersetzungsmodell bestehen und wurde beispielsweise von dem Google Übersetzer genutzt, bevor er auf NMT-Systeme umgestellt wurde[1]. Ein bekannter Vetreter der SMT-Systeme ist Moses (Koehn et al. 2007). Moses wird von Afli, Barrault und Schwenk (2016) verwendet, um OCR-Fehler zu korrigieren. NMT Ansätze versuchen die Übersetzung mithilfe eines neuronalen Netzes durchzuführen. Gerade mit den Fortschritten durch neuronale Netze in verschiedenen Bereichen des *Natural Language Processing* (NLP) (Bahdanau, Cho und Bengio 2015; Vaswani et al. 2017) hat dieser Ansatz an Beliebtheit gewonnen.

Schaefer und Neudecker (2020) vergleichen die direkte Korrektur von erkannten Texten mit einem NMT-Modell mit einer vorherigen Fehlererkennung und einer anschließenden Korrektur der Fehler. Als NMT-Modell nutzen sie in beiden Fällen ein auf *Long-Short-Term-Memory*-Zellen (LSTM) (Hochreiter und Schmidhuber 1997) basierendes Seq2Seq-Modell. Sie konnten mit dem zweistufigen Verfahren deutlich bessere Ergebnisse erzielen.

[1] `https://blog.google/products/translate/found-translation-more-accurate-fluent-sentences-google-translate/` (abgerufen am 24.01.2022)

Im ICDAR-Wettbewerb zur OCR-Post-Korrektur konnte das Team „CCC“ von Clova AI in vielen Sprachen die größten Verbesserungen erzielen (Rigaud et al. 2019, S.5). Das Team nutzt für die Fehlererkennung ein vortrainiertes BERT-Modell (Devlin et al. 2018) mit *Convolutional-Layern* und einer Klassifizierungsschicht. Für die Fehlerkorrektur nutzt das Team ein Seq2Seq-Modell bestehend aus bidirektionalen LSTMs mit einem *Attention*-Mechanismus (Bahdanau, Cho und Bengio 2015).

Nguyen et al. (2020) vereinfachen das beschriebene BERT-Modell und nutzen keine *Convolutional-Layer*, sondern nur eine Klassifizierungsschicht. Sie erzielen damit geringfügig bessere Ergebnisse als „CCC“ bei der Fehlerkennung auf den Evaluierungsdaten des ICDAR-Wettbewerbs.

Zu guter Letzt gibt es Ansätze das Training unüberwacht zu gestalten. Dong und Smith (2018) präsentieren ein unüberwachtes Trainingsverfahren, das sich wiederholende Sequenzen aus großen Textkorpora nutzt. Außerdem stellen sie Ansätze vor, um *Attention* aus mehreren Ausgangssequenzen zu kombinieren, und so Informationen aus mehreren Textsequenzen zu nutzen um Kandidaten für die Korrektur zu erzeugen. Die Ergebnisse der unüberwacht trainierten Modelle kommen zwar nicht ganz an die der überwacht trainierten heran, können aber trotzdem Verbesserungen vorweisen. Hämäläinen und Hengchen (2019) konstruieren aus einem unannotierten OCR Datensatz einen parallelen Korpus, indem sie die enthaltenen Wörter mit einem Word2Vec-Modell (Mikolov et al. 2013) clustern. Anders als Dong und Smith nutzen sie zur Korrektur ein Modell auf Buchstabenebene.

In dieser Arbeit entwickle ich verschiedene Ansätze für die OCR-Post-Korrektur des *Virtual Laboratory*. Zunächst implementiere ich einen manuelles Verfahren mithilfe von Amazon MTurk. Außerdem entwickle ich automatische Verfahren und vergleiche sie miteinander. Zu diesen zählen sowohl einfache wörterbuchbasierte Verfahren und solche mit vortrainierten oder selbst trainierten neuronalen Netzen. Den Fokus lege ich auf die Entwicklung ähnlicher Verfahren wie Nguyen et al. (2020) und „CCC“ im ICDAR-Wettbewerb, da diese Verfahren die besten Ergebnisse erzielen konnten. Konkret bedeutet dies, ein BERT-Modell für die Fehlererkennung und ein Seq2Seq-Modell für die Fehlerkorrektur zu verwenden.

Kapitel 3

Grundlagen

In diesem Kapitel beschreibe ich das Archiv des *Virtual Laboratory* und lege Grundlagen für das Verständnis meiner manuellen und automatischen Korrekturverfahren. Dazu werde ich Grundbegriffe zu Amazon MTurk erklären und auf Grundlagen zu meinen automatischen Korrekturverfahren eingehen. Die konkrete Umsetzung beschreibe ich im nächsten Kapitel.

3.1 Struktur des Archivs des *Virtual Laboratory*

Das Archiv umfasst knapp 13.500 Dokumente, darunter Bücher, Auszüge von Büchern und Artikel, aber auch Audiodokumente und Filme. Die meisten Dokumente sind Artikel aus Journalen. Die Verteilung der verschiedenen Dokumenttypen ist in Abbildung 3.1 zu sehen.

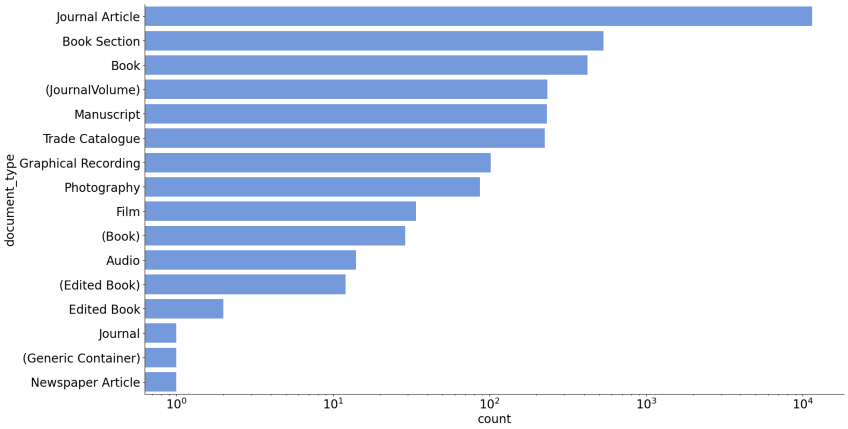


Abbildung 3.1: Verteilung der Dokumenttypen im Archiv des *Virtual Laboratory*

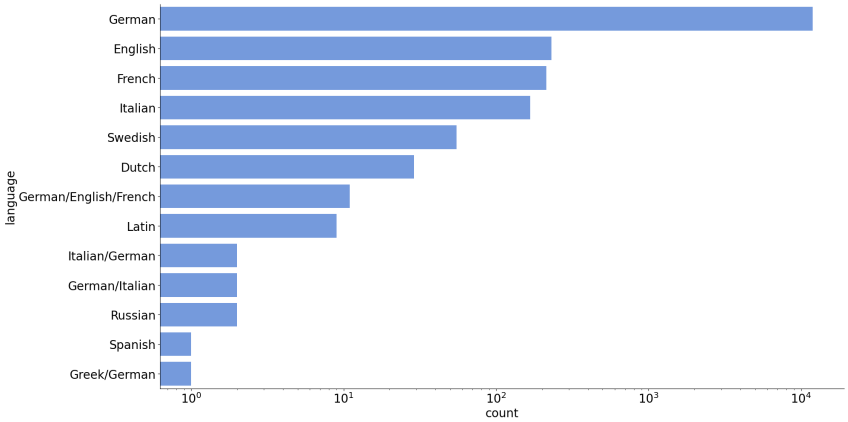


Abbildung 3.2: Verteilung der Sprachen im Archiv des *Virtual Laboratory*

Die Dokumente des Archivs umfassen 300.049 Textseiten und ungefähr 104 Millionen einzelne Wörter. Die einzelnen Seiten liegen in Form von Scans (häufig als TIFF-, manchmal auch als JPEG-Dateien) und Ausgabedateien der OCR-Engine ABBYY FineReader 10 vor. Eine solche Ausgabedatei ist ein XML-Dokument mit dem erkannten Text und zusätzlichen Informationen zu den einzelnen Bestandteilen der gescannten Seiten, beispielsweise wo sich die erkannten Buchstaben oder Absätze auf dem Bild befinden. Für mein manuelles Verfahren sind die Informationen auf Wort- und Buchstabenebene besonders interessant, z.B. Begrenzungen der Buchstaben auf den Scans oder ob ein Wort aus einem Wörterbuch kommt.

Die meisten der Dokumente des Archivs liegen in deutscher Sprache vor. Eine Verteilung der Sprachen in dem Archiv ist in Abbildung 3.2 zu finden.

Die Dokumente stammen aus den Jahren 1830 bis 1930 (Schmidgen und Evans 2003), allerdings gibt es vereinzelt Ausreißer wie den Buchauszug „Zur Entwicklung der deutschen Landwirtschaft im 19. Jahrhundert“ aus dem Jahr 1992.

3.2 Manuelle Fehlerkorrektur

Für das manuelle Korrekturverfahren nutze ich Amazon MTurk. Amazon MTurk ist eine *Crowdsourcing*-Infrastruktur, bei der eine Person oder Entität einer Gruppe von Personen Aufgaben bereitstellt. Die Personen, die Aufgaben bereitstellen werden *Requester* genannt. Personen, die die bereitgestellten Aufgaben bearbeiten, heißen *Worker*. Die Aufgaben sind *Human Intelligence Tasks* (HITs) (MTurk 2021). Die *Worker* bekommen für die Bearbeitung von HITs Geld.

Amazon MTurk empfiehlt, dass HITs die folgenden Kriterien erfüllen: Sie müssen in einem Webbrowser durchführbar sein, sie müssen kleinteilig sein und sie müssen klare Anweisungen enthalten (MTurk 2021).

Um möglichst geeignete *Worker* für einen HIT zu erhalten kann eine Reihe von Anforderungen für den HIT spezifiziert werden, zum Beispiel welche Qualifikationen ein *Worker* haben muss, um den HIT zu bearbeiten (MTurk 2021).

Wenn ein *Worker* einen HIT annimmt, entsteht eine Zuordnung zwischen *Worker* und HIT. Diese Zuordnung wird *Assignment* genannt. Es ist auch möglich, denselben HIT von mehreren *Workern* bearbeiten zu lassen, um verschiedene Ergebnisse zu vergleichen. Nach der Prüfung der Antwort kann ein *Assignment* entweder akzeptiert oder zurückgewiesen werden.

Zum Verwalten der HITs nutze ich den MTurk Manager. Der MTurk Manager ersetzt das Interface, das MTurk *Requestern* bietet (Webis 2021). Über eine übersichtliche und gebündelte Oberfläche hinaus bietet der MTurk Manager eigene Funktionen, wie zum Beispiel das Erstellen sogenannter *Requester Templates*. Diese erleichtern dem *Requester* die Überprüfung der von Workern hochgeladenen Antworten, indem eigene Sichten auf die Ergebnisse als Webseiten definiert werden können.

3.3 Automatische Fehlerkorrektur

Die automatische Fehlerkorrektur trenne ich in die Fehlererkennung und die Fehlerkorrektur. Für die Fehlererkennung nutze ich *pyhunspell* und eine Variante von BERT. Für die Fehlerverbesserung nutze ich ebenfalls pyhunspell, eine andere Variante von BERT und ein *recurrent neural network* (RNN). Im folgenden Teil möchte ich die Grundlagen für das Verständnis der Verfahren legen.

Pyhunspell Pyhunspell (Latinier 2021), ist ein Python-Wrapper für Hunspell[1]. Hunspell ist ein beliebter wörterbuchbasierter *Spell-Checker*, der unter anderem von Firefox genutzt wird. Die Sprachen werden bei Hunspell in Wörterbuch- und Affix-Dateien definiert. Erstere enthält die Wörter einer Sprache und nennt für ein Wort anwendbare Regeln. Die Regeln werden in der Affix-Datei definiert und können beispielsweise zulässige Präfixe oder Suffixe enthalten. Hunspell kann anhand der definierten Wörter und Regeln bestimmen, ob ein Wort in der Sprache ist und bei Fehlern ähnliche Wörter aus der Sprache vorschlagen.

[1] `http://hunspell.github.io/` (abgerufen am 01.12.2021)

Recurrent Neural Network (RNN) RNNs sind neuronale Netze die eine Sequenz von Eingaben verarbeiten, indem sie jedes Element einzeln einlesen und einen internen Zustand, den *Hidden-State*, für die gesamte Sequenz mitführen. Der Zustand zum Zeitpunkt t wird aus der aktuellen Eingabe, dem vorherigen Zustand und einer Aktivierungsfunktion f gebildet und damit bei jeder verarbeiteten Eingabe verändert (Cho et al. 2014, S.2):

$$h_t = f(h_{t-1}, x_t)$$

So können RNNs theoretisch Informationen aus den einzelnen Eingaben über eine beliebig lange Sequenz transportieren. In der Praxis haben RNNs aber das Problem, dass Gradienten während des Trainings entweder dazu tendieren sehr klein oder sehr groß zu werden. Diese Probleme werden von Hochreiter (1991) im Detail erforscht.

Um dem Problem entgegenzuwirken, entwickeln Hochreiter und Schmidhuber (1997) die *Long-Short-Term-Memory*-Zelle (LSTM), welche die Aktivierungsfunktion eines RNNs durch eine komplexere Einheit aus *Forget-Gate*, *Input-Gate* und *Output-Gate* ersetzt. Außerdem gibt es, abgesehen von dem *Hidden-State*, noch einen *Cell-State*, der über die gesamte Sequenz mitgeführt und von den *Gates* verändert wird.

Eine vereinfachte Version der LSTMs sind *Gated-Recurrent-Units* (GRUs). Sie werden in Cho et al. (2014) vorgestellt, ursprünglich um Merkmale für SMT-Modelle zu erzeugen. GRUs enthalten nur ein *Reset-* und ein *Update-Gate*. Außerdem haben sie statt einem *Hidden-* und einem *Cell-State* nur noch einen *Hidden-State* (Olah 2015).

In meiner Arbeit nutze ich ein Seq2Seq-Modell bestehend aus GRUs. Hierbei gibt es einen *Encoder*, der eine Eingabesequenz auf einen internen Zustand abbildet und einen *Decoder*, der aus dem internen Zustand eine Ausgabesequenz bildet. Sowohl im *Encoder* als auch im *Decoder* verwende ich GRUs.

Außerdem nutze ich im *Decoder* einen *Attention*-Mechanismus. Bahdanau, Cho und Bengio (2015) stellen den *Attention*-Mechanismus erstmals vor. Ohne den Mechanismus müssen RNNs alle Informationen der Eingabesequenz in einem Vektor fester Länge, dem *Hidden-State*, festhalten. Der *Attention*-Mechanismus erlaubt es dem *Decoder* bei der Erzeugung einer Ausgabe auf alle *Hidden-States* zu den einzelnen Eingaben der Sequenz zuzugreifen und die Relevanz der *Hidden-States* für die neue Ausgabe individuell zu bewerten.

Dazu berechnet ein *Alignment*-Modell einen Wert, die sogenannte Energie, zwischen dem letzten *Hidden-State* des *Decoders* s_{i-1} und allen *Hidden-States* des *Encoders* h_j zu den einzelnen Eingaben:

$$e_{ij} = a(s_{i-1}, h_j)$$

Das *Alignment*-Modell ist selbst ein neuronales Netz, das mit dem gesamten RNN gemeinsam trainiert werden kann. Danach wird auf alle e_{ij} die *Softmax*-Funktion angewandt, sodass für jeden *Hidden-State* h_j ein Wert α_{ij} zwischen 0 und 1 generiert wird.

Danach wird als neuer *Encoder-Hidden-State* c_i eine gewichtete Summe über alle *Hidden-States* h_j gebildet:

$$c_i = \sum_{j=1}^{T_x} \alpha_{ij} h_j$$

Die Dimension von c_i ist damit weiterhin fest und der Vektor kann als Eingabe für den *Decoder* genutzt werden um auf dessen Basis die nächste Ausgabe zu generieren.

BERT BERT steht für *Bidirectional Encoder Representations from Transformers* und wird in Devlin et al. (2018) vorgestellt.

Transformer sind eine Klasse von neuronalen Netzen, die, ähnlich wie die oben beschriebene Architektur der RNNs, aus einem *Encoder* und einem *Decoder* bestehen. Allerdings nutzen *Transformer* im *Encoder* keine *Recurrence*, sondern verarbeiten die Eingabesequenz parallel. Informationen über die Reihenfolge der Token werden über *Positional-Encodings* erhalten. Durch die parallele Verarbeitung kann das Training beschleunigt werden. Außerdem nutzen *Transformer* einen anderen *Attention*-Mechanismus als den oben beschriebenen, nämlich die *Scaled-Dot-Product-Attention*. Die Funktion des *Encoders* bleibt dieselbe, er bildet die Eingabesequenz auf eine Sequenz von kontinuierlichen Werten ab.

Wie der Name sagt besteht BERT aus dem *Encoder* eines Transformers. Die erzeugte Repräsentation der Eingabesequenz aus kontinuierlichen Werten kann an andere neuronale Netze weitergegeben werden, beispielsweise an ein neuronales Netz, das dieser Repräsentation Klassen zuordnet.

Da BERT eine allgemeine Repräsentation der Eingabe erzeugt, die unabhängig von der konkreten Aufgabe ist, können vortrainierte BERT-Modelle für unterschiedliche Anwendungen genutzt werden. Um sie für eine spezielle Anwendung nutzbar zu machen wird das vortrainierte BERT gemeinsam mit dem angeschlossenen neuronalen Netz nachtrainiert. Durch das Vortraining hat BERT ein gutes Verständnis der Sprache und liefert in der Regel bessere Ergebnisse als nicht vortrainierte Modelle.

In meiner Arbeit nutze ich BERT für die Fehlererkennung in Verbindung mit einem neuronalen Netz zur Klassifizierung einzelner Token. Bei der Fehlerverbesserung nutze ich einen *Masked-Language-Modeling*-Kopf, ein angeschlossenes neuronales Netz mithilfe dessen ein maskiertes Token in einem Text vorhergesagt wird.

Kapitel 4

Datensatzerstellung

In diesem Kapitel beschreibe ich die manuelle Korrektur von Teilen des *Virtual Laboratory*-Archivs. Ich gehe zunächst auf die Erstellung von Aufgaben für MTurk ein, danach auf die Annotationsoberflächen und Schwierigkeiten im Annotationsprozess.

4.1 Generierung von HITs aus dem Archiv

Wie in Kapitel 3 erläutert, umfasst das Archiv viele unterschiedliche Dokumenttypen und Sprachen. Da nicht alle Dokumenttypen für die manuelle Korrektur geeignet sind und ich mich in dieser Arbeit auf deutsche Dokumente beschränke, filtere ich das Archiv zunächst. Dafür habe ich eine Postgresql-Datenbank angelegt, in der ich die Metadaten zu den Dokumenten, den Seiten und den einzelnen Wörtern verwalte.

Die Dokumente filtere ich nach folgenden Kriterien:

1. die Dokumente sollen Textdokumente sein,
2. die Dokumente sollen in deutscher Sprache vorliegen,
3. die Dokumente, die aufgrund ihrer Schriftart oder ihrer Qualität besonders schlecht erkannt wurden, sollen nicht genutzt werden.

Um Punkt 1 umzusetzen, verwerfe ich Film- und Audiodateien. Für Punkt 2 wähle ich nur Dokumente in deutscher Sprache aus. Dafür nutze ich eine csv-Datei, die zu vielen Dokumenten die Sprache angibt.

Einige Dokumente im Archiv wurden von der OCR-Engine nicht richtig bearbeitet. Beispielsweise gibt es Dokumente in Frakturschrift, bei der in der Ausgabedatei die Schrift „Times New Roman“ erkannt wurde. Dementsprechend hat die OCR-Engine bei solchen Dokumenten viele Fehler gemacht, die

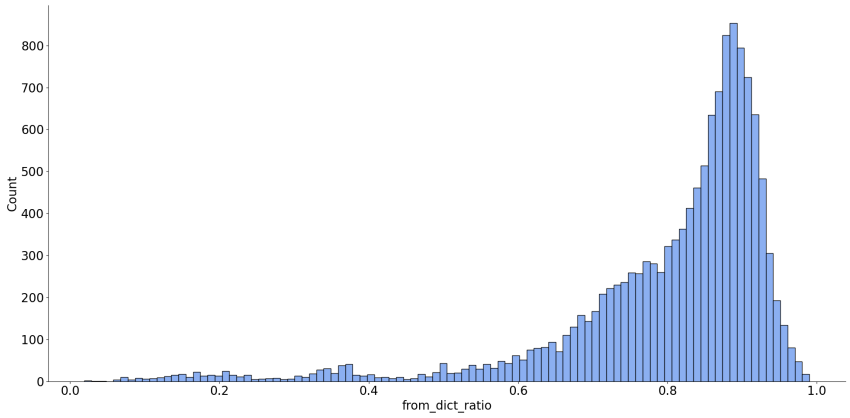


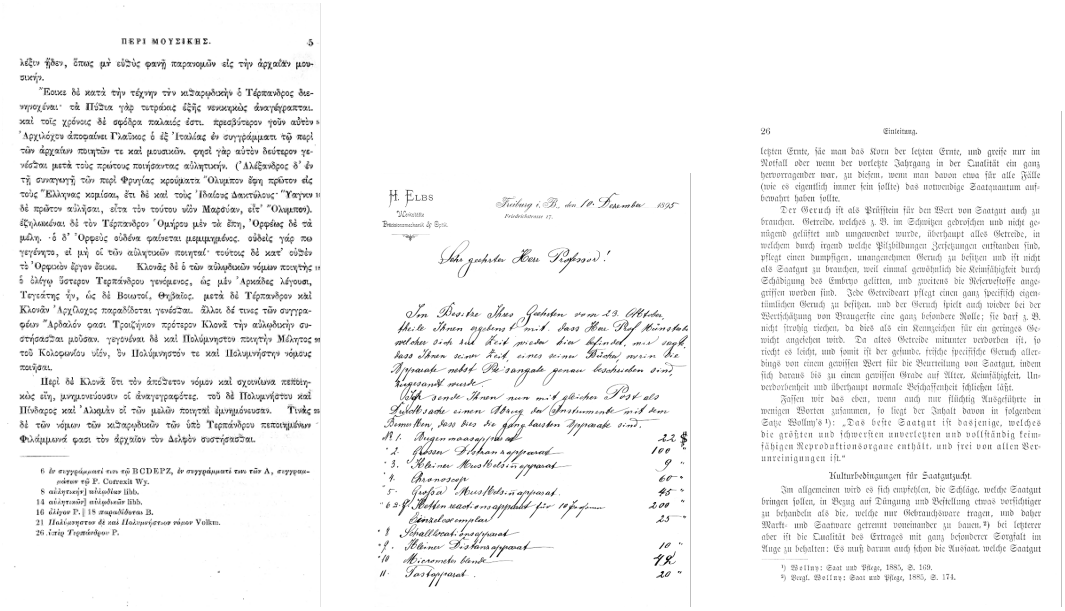
Abbildung 4.1: Verteilung des Verhältnisses von Wörtern aus dem Wörterbuch zur Gesamtanzahl der Wörter in den Dokumenten des Archivs

in anderen Dokumenten nicht in dem Maße auftreten und somit nicht repräsentativ für den Datensatz sind. Da die OCR-Engine bei diesen Dokumenten nicht richtig gearbeitet hat, sind sie auch für die OCR-Post-Korrektur nicht von Interesse und ich versuche gemäß Punkt 3 derartige Dokumente nicht manuell zu annotieren. Dafür verwerfe ich zunächst Dokumente des Typs „Manuskript“, bei denen es sich um handgeschriebene Dokumente handelt. Darüber hinaus wurden auch bei gescannten Seiten in schlechter Bildqualität keine sinnvollen Ergebnisse erzielt. Daher verwerfe ich Seiten mit weniger als 200 dpi und einer geringeren Auflösung als 1165x1654 Pixeln. Diese Auflösung entspricht 200 dpi bei einer A5 Seite. Außerdem verwerfe ich Seiten, die kein Bild haben, oder deren Bild keine TIFF-Datei ist.

Als einen weiteren Indikator für schlecht erkannte Dokumente berechne ich das Verhältnis von Wörtern aus einem Wörterbuch zu der Gesamtzahl der erkannten Wörter. Ob ein Wort aus einem Wörterbuch stammt entnehme ich der Ausgabedatei der OCR-Engine. In Abbildung 4.1 ist zu sehen, dass bei einem Großteil der Dokumente weit mehr als die Hälfte der erkannten Wörter in dem Wörterbuch enthalten waren.

Dokumente mit einem Wert von weniger als 0,5 weisen häufig ungewöhnliche Schriften oder Formatierungen vor. Daher beziehe ich diese bei der Korrektur nicht mit ein. So werden unter anderem Dokumente in Frakturschrift effektiv herausgefiltert. Drei Beispiele für Dokumente mit wenigen Wörtern aus dem Wörterbuch sind in Abbildung 4.2 zu sehen.

Abgesehen von den Dokumenten, filtere ich auch die einzelnen Wörter, die korrigiert werden sollen. Dafür stelle ich die Bedingung, dass unmittelbare Vorgänger- und Folgewörter aus dem Wörterbuch stammen sollen, um schlecht



(a) 9% aus Wörterbuch (b) 13% aus Wörterbuch (c) 32% aus Wörterbuch

Abbildung 4.2: Beispiele für Dokumente mit wenigen Wörtern aus dem Wörterbuch

erkannte Passagen der Scans auszulassen. Außerdem filtere ich nach leeren Wörtern.

Nach Anwendung der beschriebenen Filtermethoden enthält der gefilterte Datensatz 11.410 Dokumente, 166.905 Dokumentseiten und knapp 39 Millionen von anfangs über 100 Millionen Wörtern.

Da ich aufgrund der Größe des Datensatzes auch nach dem Filtern nicht jedes Wort manuell korrigieren lassen kann, versuche ich eine möglichst effektive Stichprobe aus den Daten zu ziehen. Dafür ziehe ich eine stratifizierte Zufallsstichprobe aus den Wörtern, die nach dem Filtern aller Tabellen übrig geblieben sind. Ich fasse gleiche Wörter in einer Schicht zusammen und ziehe aus jeder Schicht ein zufälliges Vorkommen. In Summe kommen so knapp 1,9 Millionen einzigartige Wörter zusammen, bei denen häufige Wörter nicht überrepräsentiert sind.

Die IDs der gezogenen Vorkommen, bestehend aus Dokument-ID, Seiten-ID und Wortnummer, schreibe ich in eine Tabelle mit Aufgaben. Aus dieser Tabelle ziehe ich zufällig IDs, um Aufgaben für die Fehlererkennung zu generieren. Zu den gezogenen IDs mische ich Verifikationsaufgaben, die ich für die Qualitätskontrolle der Ergebnisse nutze. Die Wörter dafür habe ich im Vorfeld selbst korrigiert.

Für die anschließende Fehlerkorrektur nutze ich Wörter, die bei der Fehlererkennung als falsch markiert wurden.

4.2 Annotationsoberflächen

Für die Fehlererkennung und die Fehlerkorrektur habe ich zwei Webseiten entworfen. Auf der Ersten wird erfragt ob eine gezeigte Transkription eines Wortes aus den Dokumenten dem zugehörigen Bildausschnitt entspricht (Fehlererkennung), auf der Zweiten wird nach einer Korrektur der Transkription gefragt (Fehlerkorrektur). Die Aufgaben aus beiden Oberflächen sind in Abbildung 4.3 zu sehen. Bilder der kompletten Oberflächen befinden sich im Anhang A.

Beim Entwurf der Annotationsoberflächen habe ich vor allem darauf geachtet, auch nach der Korrektur möglichst viele Daten der ursprünglichen XML-Dateien nutzen zu können. So lasse ich Buchstaben einzeln korrigieren, um eine möglichst genaue Zuordnung zu den Bilddateien und zu der Ausgabedatei der OCR-Engine zu ermöglichen.

Am Anfang der Webseiten, kann zwischen englischer und deutscher Sprache gewechselt werden, um einen möglichst niederschwelligen Zugang zu der Aufgabe zu ermöglichen. Außerdem ist eine Fortschrittsanzeige für den jeweiligen HIT zu sehen.

Im oberen Teil der Oberflächen stehen allgemeine Anweisungen für *Worker*. Diese umfassen die Beschreibung der Aufgabe, sowie Hinweise, wie Spezialfälle behandelt werden sollen. Das Ziel war, bei speziell formatierten Zeichen möglichst die Standardzeichen zu schreiben. So soll das „lange s“ als gewöhnliches „s“ und Kapitälchen als Kleinbuchstaben geschrieben werden. Außerdem sollen hoch- und tiefgestellte Zeichen wie gewöhnliche Zeichen geschrieben werden, da auch die OCR-Engine diese Fälle mithilfe der Buchstabenbegrenzungen auf dem Scan unterscheidet.

Der Textinhalt wird über einen Server geladen. Ich habe mich dafür entschieden, um schnell Änderungen an den Texten machen zu können, ohne die *Worker-Templates* im MTurk Manager ändern zu müssen. So können Fehler oder unklare Formulierungen auch bei veröffentlichten HITs geändert werden.

Nach den Anweisungen folgen einige Beispiele, die die Aufgabe anschaulich machen sollen. Bei der Fehlererkennung sind neben jedem Beispiel die Fehler und Erklärungen zu den Fehlern aufgelistet. Bei der Fehlerkorrektur steht neben dem Beispielbild der erkannte Text, die Korrektur und eine Erklärung. Insbesondere werden auch Beispiele zu den oben genannten Spezialfällen gegeben.

Unter den Beispielen ist ein Reiter, in dem die ganze Seite, auf der ein Wort annotiert werden soll, zu sehen ist. Das zu bearbeitende Wort ist auf der Seite hervorgehoben. Zusätzlich erscheint eine Lupe, wenn man mit der Maus über die Seite fährt. Ich habe mich dafür entschieden, die ganze Seite zu zeigen, da es sein kann, dass die Begrenzungen der Buchstaben nicht richtig gesetzt sind

oder Scans an manchen Stellen so schlecht sind, dass es sinnvoll ist, sich den gesamten Kontext des zu bearbeitenden Wortes anzuschauen.

Darunter stehen die eigentlichen Aufgaben, beispielhaft in Abbildung 4.3 dargestellt. In der Fehlererkennungsaufgabe ist hier das Wort, umrandet von einer roten Linie, zu sehen. Darunter befindet sich die die Transkription aus dem OCRten Text und die Frage, ob die Transkription mit dem Text im Bild übereinstimmt. Als Antworten können „ja“, „nein“ oder „nicht erkennbar“ ausgewählt werden. Über dem Bild befinden sich ein Handbuch und ein Frakturalphabet. Beide können bei Bedarf aufgeklappt werden. In dem Handbuch stehen Tipps zur Handhabung der Oberfläche, zum Beispiel Tastaturkürzel zum Auswählen von Antworten oder zum Wechseln der Seiten.

In der Korrekturoberfläche befindet sich, statt der Auswahlmöglichkeiten, für jeden erkannten Buchstaben des Wortes ein Texteingabefeld. Außerdem kann im Bild des Wortes ein Buchstabe angeklickt werden, wodurch das zugehörige Eingabefeld fokussiert wird. Darüber hinaus gibt es einen Reset-Button, der alle Eingabefelder auf die ursprüngliche Transkription zurücksetzt, und eine Eingabehilfe für Umlaute.

Bei beiden Aufgaben werden der Übersichtlichkeit halber Leerzeichen am Ende der Wörter nicht angezeigt.

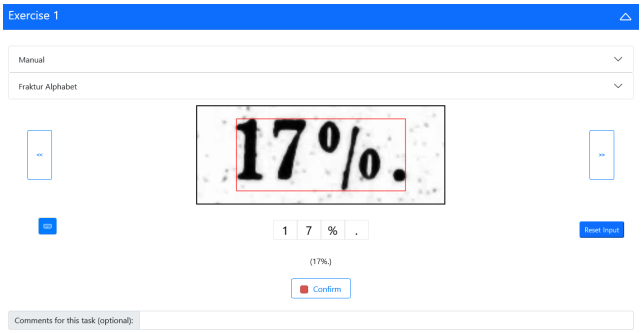
Am Ende jeder Aufgabe gibt es die Möglichkeit Kommentare zu der Aufgabe zu hinterlassen. Das wird in der MTurk Dokumentation empfohlen, um Rückmeldungen zu den Aufgaben und Verbesserungsvorschläge für die Oberflächen zu ermöglichen (MTurk 2021).

4.3 Requester Templates

Um die Antworten der Worker zu bewerten, habe ich im MTurk Manager für jede Aufgabe ein *Requester-Template* erstellt. Ausschnitte der *Requester-Templates* sind in Abbildung 4.4 zu sehen. Dort können einige Metriken für jedes *Assignment*, sowie die Antworten zu jedem Wort und ein Log, der zeigt, was ein *Worker* zu welchem Zeitpunkt getan hat, eingesehen werden. Metriken, die in beiden Aufgaben zu sehen sind, sind die Bearbeitungszeit und wie viele der Verifikationsaufgaben richtig gelöst wurden. Außerdem ist notiert, ob bei einem HIT ein Kommentar geschrieben wurde, um Anmerkungen schnell zu finden.

Bei der Fehlererkennungsaufgabe ist darüber hinaus zu sehen, wie viele Wörter jeweils als „korrekt“, „nicht korrekt“ oder „nicht erkennbar“ markiert wurden. Bei der Fehlerverbesserungsaufgabe ist der Anteil der veränderten Wörter zu sehen.

(a) Aufgabe Fehlererkennung



(b) Aufgabe Fehlerkorrektur

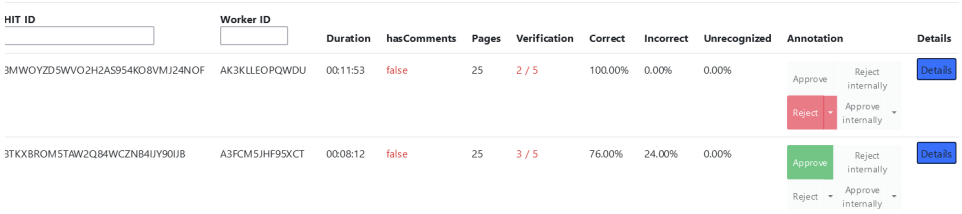
Abbildung 4.3: Interfaces für manuelle Fehlerkorrektur

Diese beiden Metriken sollen *Worker*, die keine sinnvollen Antworten geben, schnell erkennbar machen, da sie bei der Fehlererkennung oft für jedes Wort dieselbe Antwort geben und bei der Fehlerverbesserung die Wörter in der Regel nicht verändern.

4.4 Ergebnisse der manuellen Korrektur

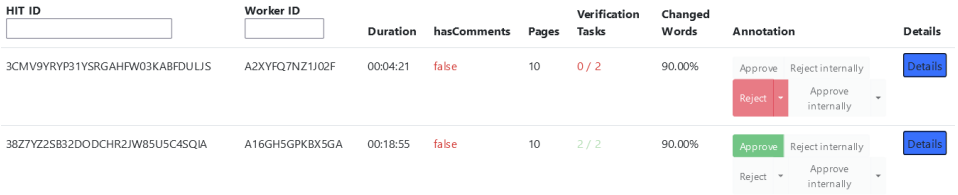
Im Rahmen der manuellen Korrektur wurden ungefähr 10.000 Wörter in 900 HITs annotiert. Sowohl der HIT für die Fehlererkennung als auch der HIT für die Fehlerkorrektur wurde mit 0.05$ vergütet, allerdings enthält der zur Fehlererkennung 25 Wörter und der zur Fehlerverbesserung 10 Wörter. Das Ziel war, dass die Bearbeitung eines HITs ungefähr eine Minute dauert. Darüber hinaus habe ich die zulässigen Regionen aus denen *Worker* angenommen werden auf Deutschland, Österreich und die Schweiz begrenzt.

Knapp 77% der HITs habe ich angenommen, den Rest habe ich größtenteils intern zurückgewiesen, da *Worker* sich bemüht haben, aber dennoch einige



HIT ID	Worker ID	Duration	hasComments	Pages	Verification	Correct	Incorrect	Unrecognized	Annotation	Details
3MWOYZD5WVO2H2AS954KO8VMJ24NOF	AK3KLLEOPQWDU	00:11:53	false	25	2 / 5	100.00%	0.00%	0.00%	Approve Reject internally Reject Approve internally	Details
3TKXBROM5TAW2Q84WCZNB4IJY90IJB	A3FCM5JHF95XCT	00:08:12	false	25	3 / 5	76.00%	24.00%	0.00%	Approve Reject internally Reject Approve internally	Details

(a) Requester Template Fehlererkennung



HIT ID	Worker ID	Duration	hasComments	Pages	Verification Tasks	Changed Words	Annotation	Details
3CMV9YRYP31YSRGAHFW03KABFDULJS	A2XYFQ7NZ1J02F	00:04:21	false	10	0 / 2	90.00%	Approve Reject internally Reject Approve internally	Details
38Z7YZ2SB32DODCHR2JW85U5C4SQIA	A16GH5GPKBX5GA	00:18:55	false	10	2 / 2	90.00%	Approve Reject internally Reject Approve internally	Details

(b) Requester Template Fehlerkorrektur

Abbildung 4.4: Interfaces zur Bewertung von Antworten

Fehler aufgetreten sind. Etwa 10% der HITs habe ich auch extern zurückgewiesen, weil *Worker* ausschließlich falsche Antworten gegeben haben.

Der *Worker*, der die meisten HITs bearbeitet hat, hat insgesamt 72 HITs bearbeitet von denen ich nur einen intern zurückgewiesen habe.

Die Kommentarfunktion wurde von den *Workern* in Einzelfällen genutzt. Die Kommentare betrafen häufig nur die jeweilige Aufgabe, allerdings wurde ich auch auf fehlende Anweisungen zu Groß- und Kleinschreibung hingewiesen, weshalb ich explizit die Kapitälchen in den Anweisungen erwähnt habe.

Aus den von den *Workern* annotierten Wörtern habe ich einen Datensatz zusammengestellt. Er umfasst eine Jsonlines-Datei, bei der jede Zeile ein Json-Objekt für das Wort enthält. Jedes Objekt enthält die Dokument-ID, die Seiten-ID und die Wortnummer auf der jeweiligen Seite, um es mit den Ausgabedateien der OCR-Engine in Beziehung setzen zu können. Darüber hinaus ist das Originalwort, die Angabe, ob es korrekt ist und gegebenenfalls eine Korrektur enthalten. Zusätzlich gibt es jeweils eine Liste von 19 Wörtern vor und hinter dem annotierten Wort, um Kontextinformationen bereitzustellen. Ein Beispiel eines Json-Objektes ist in Abbildung 4.5 zu sehen.

4.5 Probleme im Korrekturprozess

Im folgenden möchte ich auf einige Probleme in dem Korrekturprozess zu sprechen kommen. Ein Teil lässt sich darauf zurückführen, dass Ergebnisse der OCR-Engine miteinbezogen werden und Fehler in diesen Daten nicht korrigiert werden können. Allerdings vereinfacht die Nutzung dieser Daten den Annota-

```
{
    "document_id": "lit18293",
    "page_id": "p0030",
    "word_num": 183,
    "source_word": "Mercurinitrat.",
    "correct": "yes",
    "correction": null,
    "context_before": [
        ..., "von "
    ],
    "context_after": [
        "Bei ", ...
    ]
}
```


Abbildung 4.5: Beispiel eines annotierten Wortes aus dem exportierten Datensatz (die Kontext-Listen sind der Übersichtlichkeit halber gekürzt)

tionsprozess stark. Ein anderer Teil lässt sich auf die Antworten der *Worker* und deren Überprüfung zurückführen.

Probleme durch übernommene OCR-Engine-Informationen

Das erste Problem dieser Art ist, dass in dem Annotationsprozess die Begrenzungen der Buchstaben aus den Daten der OCR-Engine übernommen werden. Grundsätzlich funktioniert das gut, allerdings gibt es manchmal falsch gesetzte Begrenzungen der erkannten Buchstaben. Es passiert beispielsweise, dass mehrere falsche Buchstaben in einem einzigen Buchstaben erkannt werden, beispielsweise „ll“ statt einem „H“, oder dass Artefakte auf der gescannten Seite fälschlicherweise als Buchstaben erkannt werden. Wenn eines dieser Probleme auftritt, stimmt die Zahl der erkannten Buchstaben nicht mehr mit der Zahl der Buchstaben im Dokument überein.

Infolgedessen müssen in Eingabefeldern, die ursprünglich jeweils für einen Buchstaben vorgesehen waren, mehrere oder gar kein Buchstabe eingetragen werden. In den Fällen ist eine genaue Verknüpfung der Ergebnisse mit den Scans nicht möglich. Um das Problem zu lösen, müssten *Worker* eigenständig Buchstabenbegrenzungen einzeichnen und die Aufgabe würde sich verkomplizieren.

Ein weiteres Problem ist, dass die Wortgrenzen der OCR-Engine übernommen werden. Dass diese von OCR-Engines teilweise falsch gesetzt werden zeigen beispielsweise Nguyen et al. (2019). In ihren Datensätzen sind bis zu 20% der Wortgrenzen fehlerhaft. Dabei treten falsch getrennte Wörter häufiger auf, als falsch zusammengefügte. Da die Annotationsoberfläche der Übersichtlichkeit wegen Leerzeichen am Ende und am Anfang von Wörtern entfernt, kön-

nen falsch gesetzte Leerzeichen nicht gelöscht und Worttrennungsfehler nicht erkannt werden.

Alternativen um das Problem zu lösen, wären die Korrektur von Zeichenketten fester Länge, die Korrektur ganzer Zeilen oder aber die Leerzeichen stehen zu lassen. Allerdings würden diese Alternativen zulasten der Übersichtlichkeit gehen.

Probleme durch Antworten und deren Bewertung

Weitere Schwierigkeiten entstehen durch die Qualität der Antworten und die Mechanismen zur Überprüfung.

Zunächst wurden innerhalb eines HITs mehrere Wörter korrigiert. Da aber bei Amazon MTurk HITs die kleinstmöglichen Aufgaben sind, war es bei Fehlern bei einem Teil der Wörter nur möglich entweder den ganzen HIT zurückzuweisen oder auch die Fehler zu akzeptieren.

Weitere Probleme kommen direkt durch einige *Worker* zustande. Zunächst gab es *Worker*, die sich durch die Aufgaben schnell durchklicken, aber keine korrekten Antworten liefern. Sie können durch die Metriken in den *Requester-Templates* allerdings relativ schnell gefunden werden.

Der Großteil der Worker bemüht sich und liefert überwiegend gute Antworten, allerdings wurden bei ihren Fehlern ein paar Muster sichtbar. Zunächst hatten viele Worker Probleme mit der Unterscheidung zwischen den Buchstaben „f“ und „langes s“. Darüber hinaus waren auch Sonderzeichen ein Problem. Bei Umlauten wurde teilweise der Buchstabe ohne Punkte geschrieben (z.B. a statt ä) und ein „ß“ wurde teilweise mit einem „B“ verwechselt. Außerdem wurden als Buchstaben erkannte Artefakte oft nicht gelöscht und die Groß- und Kleinschreibung wurde auch nicht immer korrekt umgesetzt, insbesondere bei Kapitälchen.

Zudem kam es vor, dass Worker für die Korrektur einzelner Buchstaben nicht das Eingabefeld für den entsprechenden Buchstaben genutzt haben. Bei Wörtern, in denen einzelne Buchstaben hätten gelöscht werden müssen, wurde stattdessen von dem ersten Eingabefeld an ein Buchstabe in jedes weitere Feld geschrieben, bis das Wort vollständig war. Bei Wörtern, die mehr Buchstaben enthielten, als Eingabefelder vorhanden waren, wurden alle Eingabefelder mit einem Buchstaben gefüllt und danach gestoppt. Allerdings kamen diese Probleme nur selten vor.

Das größte Problem war, dass es mit meinem Verfahren unmöglich war Wörter korrigieren zu lassen, ohne die Ergebnisse nicht zumindest grob manuell zu überprüfen. Das liegt zum Teil daran, dass ich jedes Wort nur von einem *Worker* habe bearbeiten lassen. Daher kann ich keinen Wert für die Übereinstimmung der Antworten (*Inter-Annotator Agreement*) berechnen, mithilfe

dessen ich weitere Rückschlüsse auf die Korrektheit der Antworten hätte ziehen können.

Zum anderen wäre für eine umfassendere manuelle Korrektur des Archivs ein System, wie es zum Beispiel bei Chrons und Sundell (2011) umgesetzt wurde, nötig gewesen, das die Korrekturen der *Worker* automatisch vergleicht und Antworten von *Workern* mit vielen abweichenden Antworten automatisch ignoriert. Allerdings war es mir im Rahmen meiner Arbeit aus zeitlichen Gründen und weil der Fokus auf der Entwicklung automatischer Korrekturverfahren lag, nicht möglich ein solches System aufzubauen.

Kapitel 5

Automatische Korrektur und Ergebnisse

In diesem Kapitel beschreibe ich die Implementierung und das Training der automatischen Korrekturverfahren. Danach bespreche ich die Ergebnisse der Verfahren.

5.1 Implementierung der automatischen Korrekturverfahren

Für die automatische OCR-Post-Korrektur vergleiche ich die in Kapitel 3 genannten Fehlererkennungs- und Fehlerkorrekturmodelle. Im Rahmen der manuellen Korrektur wurden mit 10.000 Wörtern verhältnismäßig wenig Wörter annotiert. Der Datensatz des Wettbewerbs zur OCR-Post-Korrektur der ICDAR aus dem Jahr 2019 (ICDAR-Datensatz) enthält beispielsweise über 750.000 Token, von denen 413.703 aus deutschen Texten stammen (Rigaud et al. 2019, S.6). Daher habe ich für das Training der automatischen Modelle auf diesen Datensatz zurückgegriffen. Ausgewertet habe ich die Modelle sowohl auf den Evaluierungsdaten des ICDAR-Datensatzes als auch den annotierten Daten aus dem Archiv des *Virtual Laboratory*.

In dem Datensatz des ICDAR-Wettbewerbs sind die Dokumente in drei Ausführungen enthalten:

1. als rohe von der OCR-Engine erkannte Texte,
2. als von der OCR-Engine erkannte, aber an den korrigierten Texten ausgerichtete Texte
3. als korrigierte Texte.

```
[OCR_toInput] fÄbonncwcnt: .Vii Mn PostVaikäüxfr<wlo'/ tmti) di«
[OCR_aligned] @@@@@@@fÄbonncwcnt: .Vii @@@Mn PostVaikäüxfr<wlo'/ tmti) di«
[ GS_aligned] Nr. 151.Äbonnement: .Bei allen Postbureauxfranko@@ durch die
```

Abbildung 5.1: Ausschnitt eines Textes aus dem deutschsprachigen Teil des ICDAR-Datensatzes

Bei der Ausrichtung der Texte wird der Buchstabe „@“ als Füllzeichen genutzt (Rigaud et al. 2019). Ein Auszug eines Dokuments aus dem Datensatz ist in Abbildung 5.1 zu sehen. Die aneinander ausgerichteten Versionen werden für das Training der Modelle genutzt, der rohe erkannte Text für die Vorhersage.

Fehlererkennung Die Fehlererkennungsmodelle müssen jedem Wort eine von zwei Klassen zuordnen. Dabei steht eine „1“ für ein fehlerhaftes und eine „0“ für ein korrektes Wort. Um Trainingsdaten zu generieren, trenne ich die Texte an jedem Leerzeichen und weise jedem Token eine „0“ zu, wenn es mit dem korrigierten Text an der gleichen Position übereinstimmt, sonst eine „1“. Danach entferne ich die Füllzeichen und teile die Texte in Sequenzen mit einer Länge von 20 Token.

Als wörterbuchbasierten Ansatz nutze ich pyhunspell mit einem deutschen Standardwörterbuch. Für jedes Wort wird kontrolliert, ob es in dem Wörterbuch enthalten ist und dementsprechend als korrekt angesehen wird.

Als zweiten Ansatz nutze ich ein vortrainiertes BERT. Für die Implementierung nutze ich die *Transformers*-Bibliothek von *Huggingface*. Die Basis ist ein Modell, das im Rahmen des „Redewiedergabe“-Projekts (Brunner et al. 2020) entstanden ist. Es wurde auf deutschen Texten aus den Jahren 1840-1920 trainiert und auf dem *Huggingface-Model-Hub* bereitgestellt.

Auf Basis des vortrainierten Modells habe ich ein *BertForTokenClassification* auf den Trainingsdaten nachtrainiert. Das *BertForTokenClassification* besteht aus einem BERT und einer Klassifizierungsschicht, die jedem Token eine Klasse zuordnet.

BERT nutzt die *WordPiece*-Tokenisierung. Dabei werden seltene Wörter in Teilwörter aufgeteilt, um auch Wörter verarbeiten zu können, die nicht im Vokabular des Modells enthalten sind. Da ich auf Wortlevel entscheide ob ein Token fehlerhaft oder korrekt ist, müssen diese Labels während der *WordPiece*-Tokenisierung mit den Teilwörtern verknüpft werden. Ich habe mich dabei an einem *Huggingface*-Tutorial zu *Named Entity Recognition*[1] orientiert. Darin wird das Label eines Wortes für das erste Teilwort übernommen und für die restlichen Teilwörter das Label auf „-100“ gesetzt. Das ist ein durch PyTorch

[1] `https://huggingface.co/docs/transformers/custom_datasets#token-classification-with-wnut-emerging-entities` (abgerufen am 22.01.2022)

festgelegtes Label, mithilfe dessen der Vorhersagefehler auf diesen Teilwörtern während des Trainings ignoriert wird.

Für das Training des Modells nutze ich die Trainer-Klasse von *Huggingface* mit Standardparametern. Die Größe eines *Batches* habe ich auf 32 festgelegt. Alle 2000 Schritte evaluiere ich das Modell und erstelle einen Checkpoint. Darüber hinaus habe ich ein EarlyStoppingCallback hinzugefügt, sodass das Training nach fünf Evaluierungen ohne Verbesserung stoppt.

Fehlerkorrektur Die Fehlerkorrekturmodelle müssen zu einem fehlerhaften Wort das korrekte Wort finden. Mein erstes Verfahren nutzt pyhunspell und das deutsche Standardwörterbuch um Kandidaten für die Korrektur zu finden.

Darüber hinaus habe ich ein weiteres Verfahren mit BERT implementiert. In diesem maskiere ich fehlerhafte Wörter und lasse mit BERT Kandidaten für das maskierte Wort m generieren. Die 10 wahrscheinlichsten Kandidaten $x_1, ..., x_{10}$ gewichte ich nach ihren BERT-Logits $b_1, ..., b_{10}$ und nach der Ähnlichkeit mit dem Ausgangswort:

$$score(x_i) = s(b_i) \cdot (1 - L(x_i, m))$$

wobei L die mithilfe der Wortlängen normalisierte Levenshtein-Distanz und s die Softmax-Funktion ist. Als Korrektur wähle ich das Wort mit dem höchsten Gewicht.

Das letzte Verfahren zur Fehlerkorrektur nutzt das in Kapitel 3 beschriebene Seq2Seq-Modell. Der Code dafür stammt größtenteils von Trevett (2021). In der *Jupyter-Notebook*-Serie werden einige Paper zu verschiedenen Seq2Seq-Modellen besprochen und implementiert. Das dritte *Notebook*, das ich für mein Modell verwende, orientiert sich an Bahdanau, Cho und Bengio (2015) und nutzt GRUs mit dem darin vorgestellten *Attention*-Mechanismus.

Meine Änderungen beschränken sich größtenteils auf das Laden der Daten, da die Klassen zum Vorverarbeiten und *Batching* der Daten inzwischen veraltet sind.[2] Darüber hinaus erstelle ich ein Vokabular auf Buchstaben- und nicht auf Wortebene, wie es auch Nguyen et al. (2020) tun.

Um das Vokabular aufzubauen, führe ich, wie Nguyen et al. (2020), zwei Sonderzeichen ein: Für Worttrennungen, die nur im korrigierten Text vorkommen nutze ich „$“, für alle anderen „#“. Aus den vorverarbeiteten Daten baue ich ein Vokabular mittels *Torchtext*.

Die Trainingsdaten für das Modell generiere ich, indem ich um jedes falsche Wort aus dem ICDAR-Datensatz 5-Gramme um den Fehler bilde. Alle 5-

[2] `https://github.com/pytorch/text/releases/tag/v0.7.0-rc3` (abgerufen am 11.01.2022)

Gramme füge ich als einzelne Trainingsbeispiele dem Trainingsdatensatz hinzu. Auch dieses Vorgehen entspricht dem von Nguyen et al. (2020).

Die Hyperparameter des Modells habe ich aus dem *Jupyter-Notebook* übernommen. Ich nutze eine *Embedding*-Dimension von 256 im *Encoder* und im *Decoder* und eine *Hidden*-Dimension von 512 in beiden Teilen. Außerdem nutze ich während des Trainings eine *Teacher-Forcing-Ratio* von 0,5. Die *Teacher-Forcing-Ratio* gibt die Wahrscheinlichkeit an, mit der dem Modell während des Trainings zu einem Zeitpunkt anstatt der letzten eigenen Vorhersage die *Ground-Truth* des letzten Zeitpunktes als Eingabe gegeben wird. Dies kann den Trainingsprozess des Modells beschleunigen.[3]

5.2 Ergebnisse

In diesem Abschnitt bespreche ich die Ergebnisse der verschiedenen Verfahren, zuerst auf dem Datensatz des Wettbewerbs der ICDAR aus dem Jahr 2019, dann auf den annotierten Daten des *Virtual Laboratory*. Fehlererkennungs- und Fehlerverbesserungsmodelle werden separat evaluiert, das heißt Fehlerverbesserungsmodelle bekommen Label, die mithilfe des korrekten Textes generiert wurden. Außerdem habe ich 100 zufällige Wörter aus den annotierten Daten des *Virtual Laboratory* selbst überprüft und die Metriken dafür berechnet, um die Performanz der Modelle besser einschätzen zu können und auch die Qualität der Korrekturen zu bewerten.

Für die Fehlererkennung nutze ich die Metriken *Accuracy*, *Precision*, *Recall* und *F1-Score*. Für die Evaluierung der Fehlerverbesserungsmodelle berechne ich die *Word-Error-Rate* (WER) und die *Character-Error-Rate* (CER) zwischen Ausgangstexten bzw. den von den Modellen generierten Texten und den korrigierten Texten.

Die CER berechnet den Anteil von Ersetzungen S, Einsetzungen I und Löschungen D von Buchstaben an der Gesamtzahl der Buchstaben, der nötig ist, um von dem Ausgangstext auf den korrigierten Text zu kommen.

$$CER = \frac{S + I + D}{N}$$

N ist dabei die Länge des korrigierten Textes, daher kann die CER auch größer als 1 werden. Das gleiche ist bei der WER der Fall, allerdings berechnet sie S, I, D und N auf Wortebene.

[3] `https://machinelearningmastery.com/teacher-forcing-for-recurrent-neural-networks/` (abgerufen am 31.01.2022)

5.2.1 Evaluierungsdaten des ICDAR-Wettbewerbs

Die folgenden Ergebnisse habe ich auf den Evaluierungsdaten des ICDAR-Wettbewerbs berechnet. Pyhunspell hat dabei die einzelnen Token der rohen Texte für die Vorhersage bekommen, dem BertForTokenClassification habe ich Sequenzen fester Länge übergeben, damit Kontextinformationen miteinbezogen werden können.

Bei der Fehlererkennung können sowohl pyhunspell als auch das BertForTokenClassification gute Ergebnisse erzielen, das BERT-Modell übertrifft den wörterbuchbasierten Ansatz aber noch leicht. Insgesamt ist die Fehlerrate in dem Datensatz allerdings sehr hoch (siehe Tabelle 5.2), weshalb Modelle, die viele Wörter als falsch klassifizieren, gute Ergebnisse erzielen können. Die genauen Ergebnisse sind in Tabelle 5.1 zu sehen.

Verfahren	**Accuracy**	**Precision**	**Recall**	**F1**
BertForTokenClassification	**0.95**	**0.97**	**0.96**	**0.96**
Pyhunspell	0.87	0.9	0.92	0.91

Tabelle 5.1: Ergebnisse der Fehlererkennung über alle deutschen Texte des Datensatzes des ICDAR-Wettbewerbs

Bei der Fehlerkorrektur bin ich für das BERT-Modell und Pyhunspell wie bei der Fehlererkennung vorgegangen. Dem Seq2Seq-Modell habe ich auch Kontextinformationen mitgegeben, allerdings in diesem Fall nur zwei Token vor und hinter dem fehlerhaften Wort. Der Grund dafür ist, dass mit dem Seq2Seq-Modell nur ganze Sequenzen korrigiert werden können und ich verhindern will, dass viele korrekte Wörter verändert werden. Darüber hinaus habe ich die maximale Vorhersagelänge des Seq2Seq-Modells begrenzt, da es in seltenen Fällen passieren kann, das immer weitere Buchstaben vorhergesagt werden.

Die Ergebnisse bei der Fehlerkorrektur variieren stark. Der wörterbuchbasierte Ansatz mithilfe von pyhunspell erhöht bei den meisten Dokumenten die CER stark, die WER bleibt dieselbe oder verringert sich leicht. Ähnlich ist es bei dem BERT-Modell. Das Seq2Seq-Modell kann sowohl die CER als auch die WER auf den Dokumenten verringern. Genaue Ergebnisse sind Tabelle 5.2 zu entnehmen.

5.2.2 Archiv des *Virtual Laboratory*

Für die Evaluierung auf den annotierten Daten des *Virtual Laboratory* standen, anders als bei dem ICDAR-Datensatz, nur einzelne korrigierte Wörter zur Verfügung.

Verfahren	**CER**	**WER**
Seq2Seq	**0.21**	**0.61**
Roher Text	0.26	0.85
Pyhunspell	0.43	0.84
BertForMaskedLM	0.55	0.81

Tabelle 5.2: Ergebnisse der Fehlerkorrektur über alle deutschen Texte des Datensatzes des ICDAR-Wettbewerbs

Dazu kommt, dass sich *Worker* bei der Fehlererkennung und der Fehlerkorrektur nicht immer einig waren. So liegt die WER bei als falsch klassifizierten Wörtern bei 0.88 und nicht bei 1. Außerdem wurden noch nicht alle falschen Wörter korrigiert, weshalb der Anteil der falschen Wörter in dem Datensatz für die Fehlerkorrektur geringer ist, als er eigentlich sein müsste. Daher liegt die WER in Tabelle 5.4 unter der tatsächlichen WER. Bei der Fehlererkennung wurden 7141 Wörter als „korrekt“, 2766 Wörter als „falsch“ und 113 Wörter als „nicht erkennbar“ annotiert, was zu einer Fehlerrate von ungefähr 29% führt. 1333 der nicht korrekten Wörter wurden noch nicht korrigiert.

Insgesamt fällt die Evaluierung auf den annotierten Daten des Archivs deutlich schlechter aus, als die Evaluierung auf den Daten des ICDAR-Wettbewerbs. Für die Evaluation habe ich Sequenzen aus annotierten Wörter und, wenn möglich, 20 Wörtern vor und hinter dem jeweiligen Wort gebildet. Für die Evaluation der Fehlererkennung generiere ich Label indem ich das korrigierte Wort mit dem erkannten Wort abgleiche. Mit diesem Label vergleiche ich die Vorhersage des jeweiligen Modells. Auf den anderen Wörtern, die ich als Kontext übergebe, ignoriere ich die Vorhersage. Bei der Fehlerkorrektur übergebe ich je nach Modell entweder die ganze Sequenz oder nur das einzelne Wort, inklusive der Label, die ich schon für die Fehlerkorrektur generiert habe. Die Übergabe des Wortes ohne Kontext ist bei dem Seq2Seq-Modell nötig, da die Zuordnung der vorhergesagten Wörter zu den einzelnen Ursprungswörtern nicht möglich ist. Auch hier berechne ich die CER und die WER nur auf den annotierten Wörtern.

Bei der manuellen Korrektur habe ich selbst 100 zufällige, von den *Workern* annotierte Wörter, selbst annotiert. Bei der Fehlerkorrektur habe ich die Ergebnisse so berechnet, dass das originale Wort genutzt wird, wenn das Wort von den *Workern* als korrekt angesehen wurde, ansonsten meine eigene Korrektur. Abweichungen bei Wörtern, die von den *Workern* als korrekt markiert wurden, fließen also, wie bei den Modellen auch, nicht ein.

In Tabelle 5.3 ist zu sehen, dass sowohl Pyhunspell als auch das BertForTokenClassification deutlich schlechter abschneiden als auf den Daten des

ICDAR-Wettbewerbs. Dennoch liefert das BertForTokenClassification wieder bessere Ergebnisse. Insgesamt bleiben die automatischen Modelle jedoch weit hinter der menschlichen *Performance*.

Verfahren	**Accuracy**	**Precision**	**Recall**	**F1**
BertForTokenClassification	**0.73**	**0.51**	0.83	**0.63**
Pyhunspell	0.37	0.3	**0.96**	0.46
Manuelle Korrektur (100 Wörter)	0.91	0.96	0.93	0.94

Tabelle 5.3: Ergebnisse der Fehlererkennung auf den annotierten Wörtern des Archivs des *Virtual Laboratory*

Bei der Fehlerkorrektur auf den Daten des Archivs schneidet das Seq2Seq-Modell am schlechtesten ab, doch die Ergebnisse der anderen Modelle sind ebenfalls nicht zufriedenstellend, wie in Tabelle 5.4 zu sehen ist. Wie bei der Fehlererkennung liefern die automatischen Modelle auch hier deutlich schlechtere Ergebnisse als die manuell von mir korrigierten. Sie erhöhen sogar sowohl die CER als auch die WER im Vergleich zum Original.

Verfahren	**CER**	**WER**
Roher Text	**0.04**	**0.16**
Pyhunspell	0.07	0.17
BertForMaskedLM	0.12	0.17
Seq2Seq	0.07	0.21
Manuelle Korrektur (100 Wörter)	0.01	0.08

Tabelle 5.4: Ergebnisse der Fehlerkorrektur auf den annotierten Wörtern des Archivs des *Virtual Laboratory*

5.3 Diskussion

Die Ergebnisse der von mir selbst korrigierten Wörter auf den annotierten Daten zeigen, dass die meisten Wörter richtig korrigiert wurden. Das liegt auch daran, dass die Antworten der *Worker* stichprobenartig von mir überprüft wurden. Dennoch sind die Ergebnisse etwas schlechter als bei anderen manuellen Korrekturverfahren wie Chrons und Sundell (2011), die angeben über 99% *Accuracy* zu erreichen. Allerdings musste ich, wie in Kapitel 4 beschrieben,

mit Amazon MTurk in einigen Fällen einzelne falsche Wörter akzeptieren, um nicht einen ganzen HIT zurückzuweisen.

Die automatischen Verfahren erzielen sehr unterschiedliche Ergebnisse. Die Modelle, auf die ich mich während meiner Arbeit konzentriert habe, also das *BertForTokenClassification* bei der Fehlererkennung und das Seq2Seq-Modell bei der Fehlerkorrektur, können auf dem Evaluierungsdatensatz des ICDAR-Wettbewerbs die besten Ergebnisse erzielen, da sie auf den zugehörigen Trainingsdaten trainiert wurden. Außerdem profitiert Pyhunspell bei der Fehlererkennung von der hohen Fehlerrate in dem Datensatz. Aufgrund des unangepassten Wörterbuchs werden viele Wörter als Fehler klassifiziert, was in dem Fall mit einer hohen Wahrscheinlichkeit richtig ist. Das *BertForTokenClassification* passt sich durch das *Finetuning* ebenfalls sehr gut an den Datensatz an.

Bei der Fehlerkorrektur kann der wörterbuchbasierte Ansatz mit Pyhunspell keine Verbesserungen erzielen. Das ist aufgrund des nicht angepassten Wörterbuches plausibel, vor allem, da die Texte in alter Sprache geschrieben sind und damit Wörter enthalten, die heutzutage nicht mehr genutzt oder aber anders geschrieben werden. Auch das *BertForMaskedLM* bringt keine Verbesserungen auf den Texten. Es basiert zwar auf demselben vortrainierten Modell, wie das *BertForTokenClassification*, wurde aber nicht nachtrainiert. Darüber hinaus bringt die Wortmaskierung das Problem mit sich, dass für ein maskiertes Token auch nur ein Token vorhergesagt wird. Das gesuchte Wort ist allerdings potenziell nicht komplett in dem Vokabular enthalten, was eine Vorhersage mehrerer Teilwörter nötig machen würde.

Das Seq2Seq-Modell kann auf den Daten des ICDAR-Wettbewerbs Verbesserungen erzielen. Ein Problem des Modells ist allerdings, dass es nicht nur ein einzelnes fehlerhaftes Wort vorhersagt sondern immer die gesamte ihm übergebene Sequenz übersetzt. Einerseits bringt das den Vorteil, dass es möglich ist aufgrund der Kontextinformationen *Real-Word-Errors* zu korrigieren, andererseits können durch die Übersetzung der ganzen Sequenz auch korrekte Wörter verfälscht werden. Außerdem kann es sein, dass das Seq2Seq-Modell mehr Wörter vorhersagt, als gesucht sind. Auch eine genaue Zuordnung des fehlerhaften Wortes zu einem Wort aus der korrigierten Sequenz ist deshalb nicht möglich.

Bei der Fehlererkennung auf den annotierten Daten schneiden sowohl Pyhunspell als auch das BERT-Modell deutlich schlechter ab. Auffällig ist bei beiden Modellen der hohe *Recall*. Bei Pyhunspell ist der Grund dafür, wie bei dem ICDAR-Datensatz auch, das Standardwörterbuch, das nicht für alte Sprache und Fachwörter ausgelegt ist. Auch das BERT-Modell mag damit seine Probleme haben, allerdings basiert es auf einem auf alten deutschen Tex-

ten vortrainierten Modell. Daher ist hier eventuell das *Finetuning* auf dem ICDAR-Datensatz, der eine deutlich höhere Fehlerrate hat, problematisch.

Die Ergebnisse des BertForMaskedLM und von Pyhunspell auf den annotierten Daten sind in etwa mit den Ergebnissen auf den ICDAR-Daten vergleichbar. Sie erhöhen die CER, die WER bleibt ähnlich wie im Original. Das Seq2Seq-Modell schneidet allerdings deutlich schlechter ab. Gründe dafür liegen einerseits bei dem Modell, da die Vorhersage von mehreren Wörtern bei der Korrektur einzelner Wörter stark ins Gewicht fallen. Zum anderen gibt es Unterschiede zwischen dem Trainingsdatensatz aus dem ICDAR-Wettbewerb und den annotierten Evaluierungsdaten des *Virtual Laboratory*. Beispielsweise wird das „lange s“ in der *Ground Truth* der Trainingsdaten nicht als normales „s“ geschrieben. Außerdem wurde während des Trainings immer Kontext mit übergeben, bei der Evaluierung nicht.

Insgesamt haben die Ergebnisse gezeigt, dass die auf den ICDAR-Daten trainierten Modelle nicht einfach auf die Texte des *Virtual Laboratory* übertragbar sind. Wie oben angedeutet ist die Verteilung von fehlerhaften Wörtern in den Datensätzen unterschiedlich. Darüber hinaus enthält der Datensatz des ICDAR-Wettbewerbs teilweise andere Zeichen. Außerdem stimmen die genutzten OCR-Engines nicht überein, was ebenfalls zu Diskrepanzen führt, da jede OCR-Engine unterschiedliche Ergebnisse und Fehler erzeugt.

Kapitel 6

Zusammenfassung und Ausblick

In meiner Arbeit liefere ich Ansätze, um die von der OCR-Engine erkannten Texte im Archiv des *Virtual Laboratory* zu korrigieren. Dafür habe ich einen manuellen Ansatz mit Amazon MTurk, sowie verschiedene automatische Ansätze implementiert.

Ziel des manuellen Ansatzes war, einen Teil des Archivs möglichst präzise und im Rahmen einer simplen Aufgabe korrigieren zu lassen. Um die Aufgabe einfach zu halten, wurden Informationen aus den Ausgabedateien der OCR-Engine genutzt. Außerdem sollte mit Tastaturkürzeln, Erläuterungen zu der Aufgabe und Hilfestellungen für alte Sprache und Schrift eine für die *Worker* angenehme Bearbeitung ermöglicht werden.

Mit dem manuellen Ansatz konnte ich einen kleinen Teil des Archivs korrigieren lassen, allerdings mussten die Antworten manuell überprüft werden. Dies hat den Umfang, in dem ich Teile des Archivs korrigieren konnte, stark begrenzt. Folglich ist der entstandene Datensatz, verglichen mit anderen Datensätzen für die OCR-Post-Korrektur, relativ klein.

Um automatische Korrekturverfahren implementieren zu können, habe ich den Datensatz des ICDAR-Wettbewerbs für das Training und die Evaluation verwendet. Das BERT-Modell für die Fehlererkennung und das Seq2Seq-Modell für die Fehlerkorrektur konnten hier gute Ergebnisse erzielen. Allerdings hat die Evaluation auf den manuell annotierten Datensatz gezeigt, dass die Modelle nicht auf das Archiv des *Virtual Laboratory* übertragbar sind.

Um automatische Korrekturverfahren auf Daten aus dem *Virtual Laboratory*-Archiv trainieren zu können, müsste ein größerer Teil des Archivs zunächst manuell korrigiert werden. Dazu wäre ein Ansatz nötig, der automatisch die Antworten bewertet. Aufgrund der vielseitigen Texte und den vorhandenen Scans könnte ein solcher Datensatz sowohl für die Entwicklung anderer Post-OCR-Korrekturansätze, als auch für die Prüfung der Eignung unterschiedlicher OCR-Engines für diese Art von Texten genutzt werden. Da eine manuelle Kor-

rektur mit Amazon MTurk in größerem Umfang teuer ist, gerade wenn mehrere *Worker* jedes Wort korrigieren, wäre ein System basierend auf freiwilligen Korrigierenden ideal.

Um die manuelle Korrektur zu umgehen, könnten unüberwachte Verfahren implementiert werden, ähnlich wie es Dong und Smith (2018) und Hämäläinen und Hengchen (2019) vorschlagen, mit der Einschränkung, dass die Ergebnisse womöglich nicht ganz an die überwachter Verfahren reichen.

Die von mir genutzten automatischen Verfahren könnten ebenfalls weiter ausgebaut werden. Beispielsweise könnte *Beam-Search* genutzt werden, um mit dem Seq2Seq-Modell wahrscheinlichere Kandidaten für die Fehlerkorrektur zu finden. Darüber hinaus wäre es interessant, auf Transformern basierende vortrainierte Seq2Seq-Modelle für die Korrektur zu nutzen.

Anhang A

Annotationsoberflächen

Abbildung A.1: Annotationsoberfläche für die Fehlererkennung

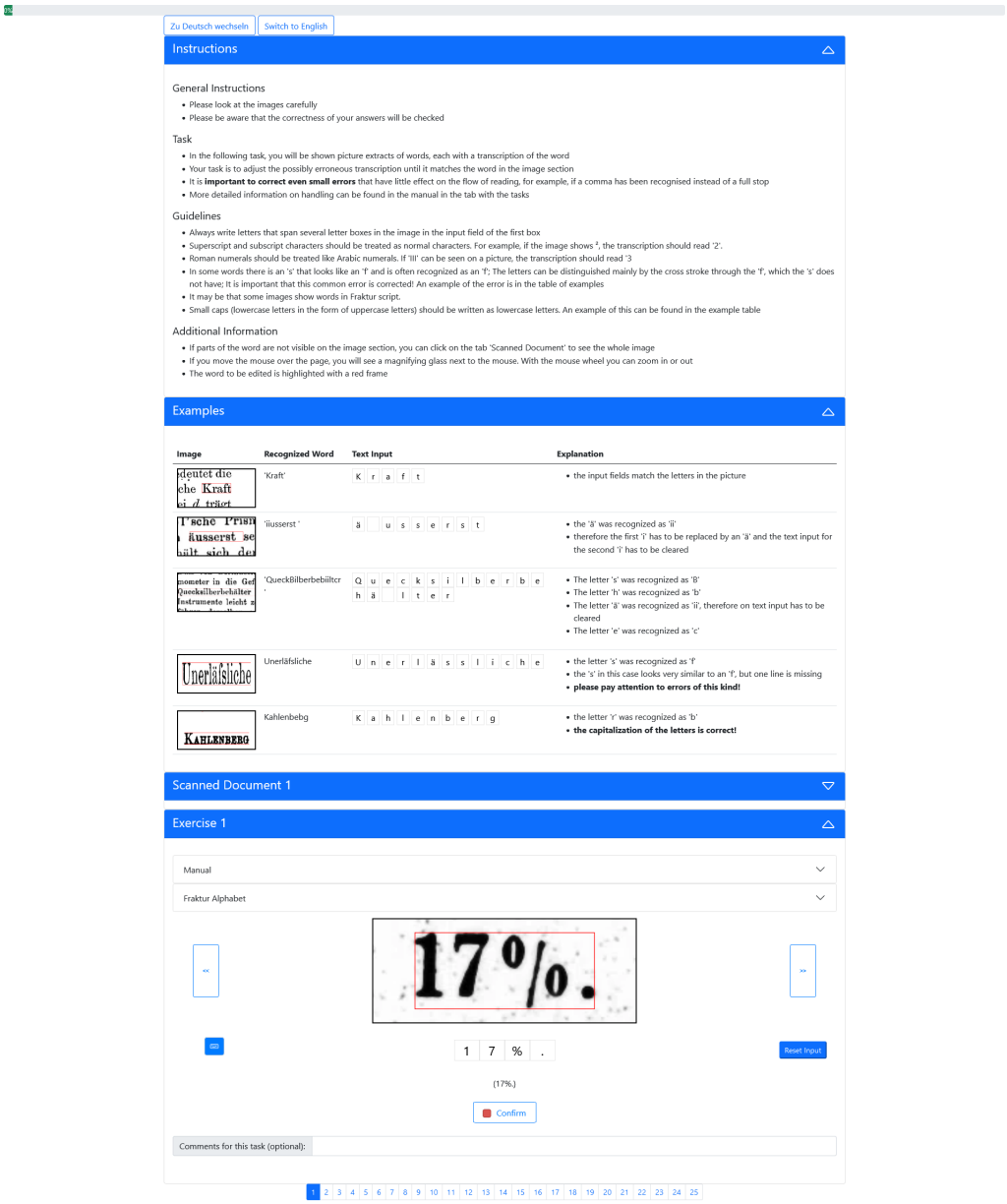


Abbildung A.2: Annotationsoberfläche für die Fehlerkorrektur

Literatur

Afli, Haithem, Loïc Barrault und Holger Schwenk (2016). „OCR Error Correction Using Statistical Machine Translation“. In: *Int. J. Comput. Linguistics Appl.* 7.1, S. 175–191. URL: `http://www.ijcla.bahripublications.com/2016-1/IJCLA-2016-1-pp-175-191-preprint.pdf`.

Bahdanau, Dzmitry, Kyunghyun Cho und Yoshua Bengio (2015). „Neural Machine Translation by Jointly Learning to Align and Translate“. In: *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*. Hrsg. von Yoshua Bengio und Yann LeCun. URL: `http://arxiv.org/abs/1409.0473`.

Brunner, Annelen et al. (2020). „To BERT or not to BERT - Comparing Contextual Embeddings in a Deep Learning Architecture for the Automatic Recognition of four Types of Speech, Thought and Writing Representation“. In: *Proceedings of the 5th Swiss Text Analytics Conference and the 16th Conference on Natural Language Processing, SwissText/KONVENS 2020, Zurich, Switzerland, June 23-25, 2020 [online only]*. Hrsg. von Sarah Ebling et al. Bd. 2624. CEUR Workshop Proceedings. CEUR-WS.org. URL: `http://ceur-ws.org/Vol-2624/paper5.pdf`.

Bubenhofer, Noah et al. (2015). *Text+Berg-Korpus (Release 151_v01)*. XML-Format. Digitale Edition des Jahrbuch des SAC 1864-1923, Echo des Alpes 1872-1924, Die Alpen, Les Alpes, Le Alpi 1925-2014, The Alpine Journal 1969-2008.

Cho, Kyunghyun et al. (2014). „Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation“. In: *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014, October 25-29, 2014, Doha, Qatar, A meeting of SIGDAT, a Special Interest Group of the ACL*. Hrsg. von Alessandro Moschitti, Bo Pang und Walter Daelemans. ACL, S. 1724–1734. DOI: `10.3115/v1/d14-1179`. URL: `https://doi.org/10.3115/v1/d14-1179`.

Chrons, Otto und Sami Sundell (2011). „Digitalkoot: Making Old Archives Accessible Using Crowdsourcing“. In: *Human Computation, Papers from the 2011 AAAI Workshop, San Francisco, California, USA, August 8, 2011*. Bd. WS-11-11. AAAI Workshops. AAAI. URL: http://www.aaai.org/ocs/index.php/WS/AAAIW11/paper/view/3813.

Devlin, Jacob et al. (2018). „BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding“. In: *CoRR* abs/1810.04805. arXiv: 1810.04805. URL: http://arxiv.org/abs/1810.04805.

Dong, Rui und David Smith (2018). „Multi-Input Attention for Unsupervised OCR Correction“. In: *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics, ACL 2018, Melbourne, Australia, July 15-20, 2018, Volume 1: Long Papers*. Hrsg. von Iryna Gurevych und Yusuke Miyao. Association for Computational Linguistics, S. 2363–2372. DOI: 10.18653/v1/P18-1220. URL: https://aclanthology.org/P18-1220/.

Estrella, Paula und Pablo Paliza (2014). „OCR correction of documents generated during Argentina's national reorganization process“. In: *Digital Access to Textual Cultural Heritage 2014, DATeCH 2014, Madrid, Spain, May 19-20, 2014*. Hrsg. von Apostolos Antonacopoulos und Klaus U. Schulz. ACM, S. 119–123. DOI: 10.1145/2595188.2595194. URL: https://doi.org/10.1145/2595188.2595194.

Hämäläinen, Mika und Simon Hengchen (2019). „From the Paft to the Fiiture: a Fully Automatic NMT and Word Embeddings Method for OCR Post-Correction“. In: Hrsg. von Ruslan Mitkov und Galia Angelova, S. 431–436. DOI: 10.26615/978-954-452-056-4_051. URL: https://doi.org/10.26615/978-954-452-056-4_051.

Hochreiter, Sepp (1991). „Untersuchungen zu dynamischen neuronalen Netzen“. In: *Diploma, Technische Universität München* 91.1.

Hochreiter, Sepp und Jürgen Schmidhuber (1997). „Long Short-Term Memory“. In: *Neural Comput.* 9.8, S. 1735–1780. DOI: 10.1162/neco.1997.9.8.1735. URL: https://doi.org/10.1162/neco.1997.9.8.1735.

Holley, Rose (2009). *Many hands make light work : public collaborative text correction in Australian historic newspapers*. Canberra : National Library

of Australia. ISBN: 978-0-642-27694-0. URL: http://www.nla.gov.au/ndp/project_details/documents/ANDP_ManyHands.pdf.

Koehn, Philipp et al. (2007). „Moses: Open Source Toolkit for Statistical Machine Translation“. In: *ACL 2007, Proceedings of the 45th Annual Meeting of the Association for Computational Linguistics, June 23-30, 2007, Prague, Czech Republic*. Hrsg. von John A. Carroll, Antal van den Bosch und Annie Zaenen. The Association for Computational Linguistics. URL: https://aclanthology.org/P07-2045/.

Latinier, Benoit (2021). *pyhunspell*. https://github.com/blatinier/pyhunspell. online; accessed: 2021-12-01.

Lund, William B., Douglas J. Kennard und Eric K. Ringger (2013). „Combining multiple thresholding binarization values to improve OCR output“. In: *Document Recognition and Retrieval XX, part of the IS&T-SPIE Electronic Imaging Symposium, Burlingame, California, USA, February 5-7, 2013, Proceedings*. Hrsg. von Richard Zanibbi und Bertrand Coüasnon. Bd. 8658. SPIE Proceedings. SPIE, 86580R. DOI: 10.1117/12.2006228. URL: https://doi.org/10.1117/12.2006228.

Mikolov, Tomás et al. (2013). „Efficient Estimation of Word Representations in Vector Space“. In: *1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013, Workshop Track Proceedings*. Hrsg. von Yoshua Bengio und Yann LeCun. URL: http://arxiv.org/abs/1301.3781.

MTurk, Amazon (2021). *Amazon MTurk Documentation*. https://docs.aws.amazon.com/mturk/index.html. online; accessed: 2021-12-01.

Nguyen, Thi-Tuyet-Hai et al. (2019). „Deep Statistical Analysis of OCR Errors for Effective Post-OCR Processing“. In: *19th ACM/IEEE Joint Conference on Digital Libraries, JCDL 2019, Champaign, IL, USA, June 2-6, 2019*. Hrsg. von Maria Bonn et al. IEEE, S. 29–38. DOI: 10.1109/JCDL.2019.00015. URL: https://doi.org/10.1109/JCDL.2019.00015.

Nguyen, Thi-Tuyet-Hai et al. (2020). „Neural Machine Translation with BERT for Post-OCR Error Detection and Correction“. In: *JCDL '20: Proceedings of the ACM/IEEE Joint Conference on Digital Libraries in 2020, Virtual Event, China, August 1-5, 2020*. Hrsg. von Ruhua Huang et al. ACM,

S. 333–336. DOI: 10.1145/3383583.3398605. URL: https://doi.org/10.1145/3383583.3398605.

Nguyen, Thi-Tuyet-Hai et al. (2021). „Survey of Post-OCR Processing Approaches“. In: *ACM Comput. Surv.* 54.6, 124:1–124:37. DOI: 10.1145/3453476. URL: https://doi.org/10.1145/3453476.

Olah, Christopher (2015). *Understanding LSTM Networks*. https://colah.github.io/posts/2015-08-Understanding-LSTMs/. online; accessed: 2021-12-02.

Rigaud, Christophe et al. (2019). „ICDAR 2019 Competition on Post-OCR Text Correction“. In: *2019 International Conference on Document Analysis and Recognition, ICDAR 2019, Sydney, Australia, September 20-25, 2019*. IEEE, S. 1588–1593. DOI: 10.1109/ICDAR.2019.00255. URL: https://doi.org/10.1109/ICDAR.2019.00255.

Schaefer, Robin und Clemens Neudecker (2020). „A Two-Step Approach for Automatic OCR Post-Correction“. In: *Proceedings of the The 4th Joint SIGHUM Workshop on Computational Linguistics for Cultural Heritage, Social Sciences, Humanities and Literature*, S. 52–57.

Schmidgen, Henning und Rand B. Evans (2003). „The Virtual Laboratory: A New On-Line Resource for the History of Psychology“. In: *History of Psychology* 6.2, S. 208–213. URL: https://epub.uni-regensburg.de/27142/.

Springmann, Uwe et al. (2018). „Ground Truth for training OCR engines on historical documents in German Fraktur and Early Modern Latin“. In: *CoRR* abs/1809.05501. arXiv: 1809.05501. URL: http://arxiv.org/abs/1809.05501.

Strien, Daniel van et al. (2020). „Assessing the Impact of OCR Quality on Downstream NLP Tasks“. In: *Proceedings of the 12th International Conference on Agents and Artificial Intelligence, ICAART 2020, Volume 1, Valletta, Malta, February 22-24, 2020*. Hrsg. von Ana Paula Rocha, Luc Steels und H. Jaap van den Herik. SCITEPRESS, S. 484–496. DOI: 10.5220/0009169004840496. URL: https://doi.org/10.5220/0009169004840496.

Trevett, Ben (2021). *PyTorch Seq2Seq*. https://github.com/bentrevett/pytorch-seq2seq. online; accessed: 2021-12-06.

Vaswani, Ashish et al. (2017). „Attention Is All You Need“. In: *CoRR* abs/1706.03762. arXiv: 1706.03762. URL: http://arxiv.org/abs/1706.03762.

Volk, Martin, Torsten Marek und Rico Sennrich (2010). „Reducing OCR errors by combining two OCR systems“. In.

Von Ahn, Luis et al. (2008). „recaptcha: Human-based character recognition via web security measures“. In: *Science* 321.5895, S. 1465–1468.

Webis (2021). *MTurk Manager Wiki*. https://github.com/webis-de/mturk-manager/wiki. online; accessed: 2021-12-01.