Julius-Maximilians-Universität Würzburg
Institut für Informatik
Lehrstuhl für Informatik II

Masterarbeit

Analysen und Heuristiken zur Verbesserung von OCR-Ergebnissen bei Frakturtexten

Paul Vorbach

17. September 2014

Betreuer:
Prof. Dr. Jürgen Albert

Inhaltsverzeichnis

1 Einführung

1.1 Motivation und Zielsetzung

Optische Zeichenerkennung (engl. „Optical Character Recognition“, kurz: OCR) bezeichnet das maschinelle Erkennen von Buchstaben und anderen Zeichen in digitalem Bildmaterial, welche zu einem digitalen Text zusammengefügt werden. Die Zeichenerkennung hat bei gedruckten Antiquatexten in den letzten Jahrzehnten große Fortschritte gemacht, sodass sich neue Themenfelder für die Forschung ergeben haben. Aktuelle Forschungsgebiete sind unter anderem die Erkennung von Handschriften, von nicht-lateinischen Schriften sowie die Erkennung alter Hand- und Druckschriften, wo Erkennungsraten deutlich hinter denen von Antiquatexten zurückbleiben.

Diese Arbeit konzentriert sich auf die Verbesserung der Erkennung von Frakturtexten. Dafür wurde die Zeichenerkennungssoftware Tesseract am Beispiel eines zweibändigen Nachdrucks der Würzburger Bischofs-Chronik von Lorenz Fries von 1924 [21] evaluiert. Der Text, häufig auch Fries-Chronik genannt, wurde nach zwei handschriftlichen Abschriften verfasst, die etwa zeitgleich mit dem Original um das Jahr 1546 entstandenen. Er erschien in einer ersten Auflage 1848 im Verlag Bonitas-Bauer in Würzburg, wobei die „alten Sprachformen“ der Abschriften beseitigt wurden [34, S. 38, 46–47]. Die Schreibweise des Werks von 1848 wurde in der zweiten Auflage von 1924 jedoch beibehalten [21, S. VIII]. Die Fries-Chronik enthält die Geschichte der Bischöfe in Würzburg und beginnt mit dem heiligen Kilian[1] im siebten Jahrhundert. Das erste zugrunde liegende Manuskript endet 1496, während das zweite Manuskript bis zum Jahr 1582 fortgeführt wurde. Die weiteren Bischöfe bis zum Jahr 1804 wurden aus anderen Quellen ergänzt [21, S. VII]. Im ersten Band sind die Bischöfe bis 1496 enthalten. Die weiteren Bischöfe folgen in einem zweiten Band.

Das Ziel der Arbeit ist, die Erkennungsgenauigkeit von Tesseract auf diesem Werk zu analysieren und so zu optimieren, dass Tesseract auch zur Digitalisierung weiterer in Fraktur gesetzter Werke des gleichen Verlags verwendet werden kann. Diese enthalten häufig die gleichen Schriftarten und sollten somit ebenfalls gut erfasst werden können. Ein weiteres Ziel war, Empfehlungen für ein effizientes Training neuer Schriften zu sammeln und außerdem verschiedene Konfigurationsmöglichkeiten der Software hinsichtlich ihrer Effekte zu analysieren.

Die Fries-Chronik wurde gewählt, da Sie durch frühere Digitalisierungsprojekte gut erforscht ist und bereits hochwertige Scans vorliegen. Außerdem existiert noch eine weitere Fassung der Bände von Bonitas-Bauer aus dem Jahr 1961 [22], die vollständig in Antiqua gesetzt wurde, sonst aber unver-

[1] Kilian wird allgemein nicht als erster Bischof Würzburgs angesehen, da das Bistum erst später gegründet wurde [21, S. 1].

ändert blieb, sowie eine händisch nachkorrigierte Version einer OCR-Fassung der Ausgabe von 1924. Hierdurch konnten Erkennungsergebnisse einfach und effizient evaluiert werden. Trotzdem bleibt die Zeichenerkennung des Texts genügend schwierig und interessant, um allgemeine Aussagen über die Leistungsfähigkeit von Tesseract zur Erfassung von Frakturtexten machen zu können.

1.2 Aufbau der Arbeit

Im folgenden Kapitel werden einige Grundlagen zu Frakturschriften sowie zur optischen Zeichenerkennung im Allgemeinen behandelt. Anschließend folgt ein Überblick über den Markt für OCR-Software, welcher die Zielsetzung verschiedener OCR-Programme und deren Einsatzmöglichkeiten für Frakturtexte zeigt.

Zentraler Teil der Arbeit ist Kapitel 4, in dem zunächst die Software Tesseract vorgestellt wird. Die Funktionsweise der Komponenten der Texterkennungssoftware wird im Detail beleuchtet und mit einer Evaluation der existierenden Sprachpakete, die sich für Frakturtexte eignen, abgerundet. Die Evaluation erfolgt anhand einer Transkription der Fries-Chronik.

In Kapitel 5 werden zahlreiche Trainingsexperimente besprochen, die unternommen wurden, um die Erkennungsgenauigkeit der Software zu verbessern. Dazu wurden zunächst Hypothesen gebildet und durch zielgerichtete Trainingsexperimente widerlegt oder bestätigt. Später wurde die Texterkennung mit größeren Beispielen trainiert und bestehende Sprachpakete wurden verbessert. Daraus entstand ein Sprachpaket, das die verschiedenen Ansätze kombiniert und deutlich bessere Erkennungsergebnisse als die bestehenden Sprachpakete ermöglicht.

Anschließend werden in Kapitel 6 die im Rahmen der Arbeit entwickelten Softwarekomponenten vorgestellt. Diese erleichtern die Arbeit mit Tesseract und vereinfachen das Training neuer Schriftarten, indem sie dazu beitragen, typische Fehler zu vermeiden.

Im Fazit wird letztlich die Leistungsfähigkeit der angewandten Trainingsprozesse sowie der Implementierung betrachtet. Der Ausblick zeigt Verbesserungsmöglichkeiten der in dieser Arbeit erzielten Resultate sowie Erweiterungsmöglichkeiten für die Zukunft.

2 Grundlagen

2.1 Fraktur

Bei den Frakturschriften handelt es sich um eine Untergruppe der gebrochenen Schriften. In dieser Schriftklasse werden nach DIN 16518 die fünf Untergruppen Gotisch, Rundgotisch, Schwabacher, Fraktur und weitere Fraktur-Varianten unterschieden [71]. Sie haben gemeinsam, dass die Bögen ihrer Buchstaben durch einen abrupten Richtungswechsel während der Schreibbewegung „gebrochen" werden. Dies steht im Gegensatz zu Antiqua-Schriften, bei denen es keinen solchen Bogenbruch gibt. Abbildung 2.1 zeigt vier gebrochene Computerschriften, die nach historischem Vorbild erstellt wurden.

Die erste Schriftfamilie, die zu den gebrochenen Schriften gezählt wird, ist die der gotischen Schriften, welche seit dem elften Jahrhundert geschrieben wurden. Ihr bekanntester Vertreter ist die Textura, die zur Mitte des dreizehnten Jahrhunderts entstand. Aus den gotischen Schriften entwickelten sich Mitte des 14. Jahrhunderts die rundgotische Schrift und zum Ende des 15. Jahrhunderts die Schwabacher.

Die Fraktur wurde vor allem im deutschen Sprachraum seit dem Anfang des 16. Jahrhunderts verwendet. Seit Beginn des 20. Jahrhunderts wurde sie zunehmend durch Antiqua-Schriften verdrängt, fand zum Beginn des Nationalsozialismus einen kurzen Aufschwung und wurde am 3. Januar 1941 schließlich verboten [48]. Auch nach dem Ende des Zweiten Weltkriegs fand die Fraktur keine erneute Verbreitung.

Nach der Definition von Hendlmeier [26] lässt sich die Untergruppe der Fraktur-Schriften weiter in die Gruppen Renaissance-Fraktur, Barock-Fraktur, Klassizistische Fraktur, Biedermeier-Fraktur, Jugendstil-Fraktur, neuere geschriebene Künstler-Fraktur und neuere gezeichnete Künstler-Fraktur

𝔄𝔞 𝔅𝔟 ℭ𝔠 𝔇𝔡 𝔈𝔢 𝔉𝔣 𝔊𝔤 ℌ𝔥 ℑ𝔦 𝔍𝔧 𝔎𝔨 𝔏𝔩 𝔐𝔪 𝔑𝔫 𝔒𝔬 𝔓𝔭 𝔔𝔮 ℜ𝔯 𝔖𝔰 𝔗𝔱 𝔘𝔲 𝔙𝔳 𝔚𝔴 𝔛𝔵 𝔜𝔶 ℨ𝔷

(a) Gotisches Alphabet gesetzt aus „Manuskript-Gotisch" [50, S. 113]

𝕬𝖆 𝕭𝖇 𝕮𝖈 𝕯𝖉 𝕰𝖊 𝕱𝖋 𝕲𝖌 𝕳𝖍 𝕴𝖎 𝕵𝖏 𝕶𝖐 𝕷𝖑 𝕸𝖒 𝕹𝖓 𝕺𝖔 𝕻𝖕 𝕼𝖖 𝕽𝖗 𝕾𝖘 𝕿𝖙 𝖀𝖚 𝖁𝖛 𝖂𝖜 𝖃𝖝 𝖄𝖞 𝖅𝖟

(b) Rundgotisches Alphabet gesetzt aus „Weiß Rundgotisch" [60]

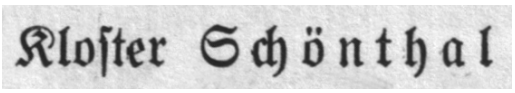
𝕬𝖆 𝕭𝖇 𝕮𝖈 𝕯𝖉 𝕰𝖊 𝕱𝖋 𝕲𝖌 𝕳𝖍 𝕴𝖎 𝕵𝖏 𝕶𝖐 𝕷𝖑 𝕸𝖒 𝕹𝖓 𝕺𝖔 𝕻𝖕 𝕼𝖖 𝕽𝖗 𝕾𝖘 𝕿𝖙 𝖀𝖚 𝖁𝖛 𝖂𝖜 𝖃𝖝 𝖄𝖞 𝖅𝖟

(c) Schwabacher Alphabet gesetzt aus der „Alten Schwabacher" [74]

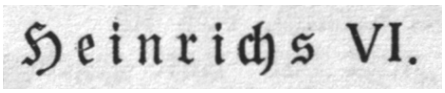
𝔄𝔞 𝔅𝔟 ℭ𝔠 𝔇𝔡 𝔈𝔢 𝔉𝔣 𝔊𝔤 ℌ𝔥 ℑ𝔦 𝔍𝔧 𝔎𝔨 𝔏𝔩 𝔐𝔪 𝔑𝔫 𝔒𝔬 𝔓𝔭 𝔔𝔮 ℜ𝔯 𝔖𝔰 𝔗𝔱 𝔘𝔲 𝔙𝔳 𝔚𝔴 𝔛𝔵 𝔜𝔶 ℨ𝔷

(d) Fraktur-Alphabet gesetzt aus der „Normalfraktur" [25]

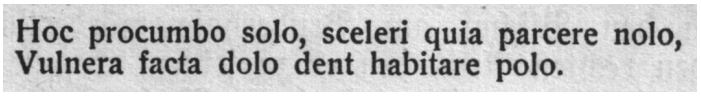
Abb. 2.1: Alphabete gesetzt aus vier gebrochenen Schriften



(a) Sperrsatz zur Hervorhebung eines Ortsnamens



(b) Römische Zahlen werden in Antiqua gesetzt …



(c) …ebenso wie Latein.

Abb. 2.2: Auszeichnungsmöglichkeiten in Frakturtexten

unterteilen. Die am häufigsten verwendeten Frakturschriften zählen zur Stilgruppe der Biedermeier-Fraktur und entstanden um 1830 [26]. Nachfolgend sind einige Besonderheiten von gebrochenen Schriften im Allgemeinen und der Fraktur im Speziellen aufgeführt.

Unterscheidung zwischen langem ſ und rundem s Im Fraktursatz existieren zwei Varianten des kleinen s, das lange ſ und das runde s. Beide Formen werden nach bestimmten Regeln verwendet. So steht das lange ſ im Anlaut oder innerhalb einer Silbe (ſagen, Rätſel, Manuſkript). „Das gilt auch dann, wenn ein sonst im Silbenanlaut stehender s-Laut durch den Ausfall eines unbetonten e in den Auslaut gerät. (auserleſne (*für:* auserleſene), ich preiſ (*für:* ich preiſe), […])“ [72, S. 100]

Das runde s (oft auch Schluss-s) findet sich hingegen im Auslaut einer Silbe (das, aus, Muskel) sowie „[…] für -sk in bestimmten Fremdwörtern. (brüsk, grotesk, Obelisk)“ [72, S. 100] Fremdwörter müssen jedoch in Antiqua gesetzt werden, sofern sie nicht als eingedeutscht erscheinen [72, S. 90].

„Das lange ſ steht in den Buchstabenverbindungen ſch, ſp, ſt. (ſchaden, […]; Knoſpe, […]; geſtern […]) Kein ſ steht aber, wenn in Zusammensetzungen s + ch, s + p und s + t zusammentreffen. (Zirkuschef, […] transparent, […] Dienstag, […])“ [72, S. 100]

Ligaturen Sowohl im Fraktur- als auch im Antiquasatz ist es üblich, für bestimmte Buchstabenkombinationen Ligaturen zu verwenden. Das heißt, diese Buchstaben werden zur besseren Lesbarkeit durch ein zusammengesetztes Zeichen, die Ligatur, ersetzt. Ligaturen zeichnen sich meist durch einen verringerten oder weggelassenen Zwischenraum sowie veränderte Details aus. Üblicherweise finden im Fraktursatz folgende Ligaturen Verwendung: ch, ck, ff, fi, fl, ft, ll, ſch, ſi, ſſ, ſt, ß [72, S. 97]. Obwohl das Eszett aus der Ligatur aus langem ſ und z entstand (ſz → ß), wird es laut Duden [72, S. 97] als ein einzelner Buchstabe behandelt.

Auszeichnungsmöglichkeiten Frakturschriften haben in der Regel keine kursiven und selten fette Schnitte. Daher wird zur Hervorhebung von Text im Fraktursatz meist auf das Sperren von Wörtern zurückgegriffen. Dabei werden die Abstände zwischen den Buchstaben durch Einfügen von Spatien vergrößert, wie in Abbildung 2.2a zu sehen ist. Hier fällt jedoch auf, dass die Ligatur ch nicht gesperrt

(a) Einfache Initiale

(b) Reich verzierte Initiale

Abb. 2.3: Initialen in Frakturtexten

Abb. 2.4: Die Abkürzung ꝛc.

wurde. Die Ligaturen ch, ck, und tz werden generell nicht gesperrt ([72, S. 97]; [20, S. 304]). Alle weiteren Ligaturen werden jedoch beim Sperren durch Leerraum getrennt.

Weitere Besonderheiten Eine weitere Besonderheit von Frakturtexten ist die Verwendung von Initialen zu Beginn eines Kapitels oder Abschnitts. Diese werden heute nur noch selten eingesetzt. Initialen können eher schlicht wie in Abbildung 2.3a aber auch reich verziert wie in Abbildung 2.3b ausfallen und stellen so eine Schwierigkeit für den Erkennungsprozess dar.

Neben den Buchstaben ſ und ß gibt es auch noch eine alternative Darstellung des kleinen r, nämlich das runde ꝛ, welches in mittelalterlichen Texten gebräuchlich war. In jüngeren Texten findet dieser Buchstabe jedoch keine Verwendung mehr. Eine dem runden ꝛ zum Verwechseln ähnliche tironische Note[ˆtiro] ist jedoch auch in Frakturtexten als Symbol für das lateinische Wort *et* erhalten geblieben. Abbildung 2.4 zeigt die Verwendung in der Abkürzung ꝛc., welche für *et cetera* steht.

Typisch für ältere Frakturtexte ist auch, dass die Majuskeln I und J als J geschrieben werden. Erst ab etwa 1900 entstanden auch Schriften, in denen zwischen J und J unterschieden wurde [20, S. 307]. Trotzdem kann die Unterscheidung beider Zeichen auch bei genauem Hinsehen, wie auch bei der hier verwendeten Normalfraktur, schwerfallen und muss meist durch den Kontext getroffen werden.

2.2 Optische Zeichenerkennung

Ein System zur optischen Zeichenerkennung ist in der Regel als Pipeline aus mehreren Komponenten aufgebaut. Zunächst erfolgt die Erfassung eines physikalisch vorliegenden Dokuments durch einen Scanner oder eine Digitalkamera. Anschließend erfolgt eine Vorverarbeitung, in der Schmutz und weitere störende Faktoren aus dem digitalen Bildmaterial entfernt werden. Häufig wird auch eine Reduktion des Farbraums hin zu Schwarz-Weiß oder Graustufen vorgenommen, um Muster leichter erkennen und verarbeiten zu können. Danach folgt eine Segmentierung des Bilds in Text- oder Bild-Regionen

sowie Zeilen, Wörter und Zeichen, was der weiteren Erkennung dient. Schließlich folgt die Erkennung der einzelnen Zeichen über Methoden der Mustererkennung.

Die folgenden Abschnitte stellen einige der gängigen Methoden zur optischen Zeichenerkennung vor, wobei die Digitalisierung durch Hardware nicht Teil der Übersicht sein soll. Einen umfassenderen Überblick über verbreitete Methoden und neuere Entwicklungen liefern Cheriet u. a. [18].

2.2.1 Vorverarbeitung

Für die Texterkennung ist es hilfreich, den Farbraum des Originals zu reduzieren. In der Regel wird die Texterkennung in OCR-Systemen auf graustufigen oder binären Schwarz-Weiß-Bildern durchgeführt. Viele Scanner können auch direkt Graustufenbilder erzeugen, meist liegen Bilder auf Computern aber im RGB-Farbraum vor.

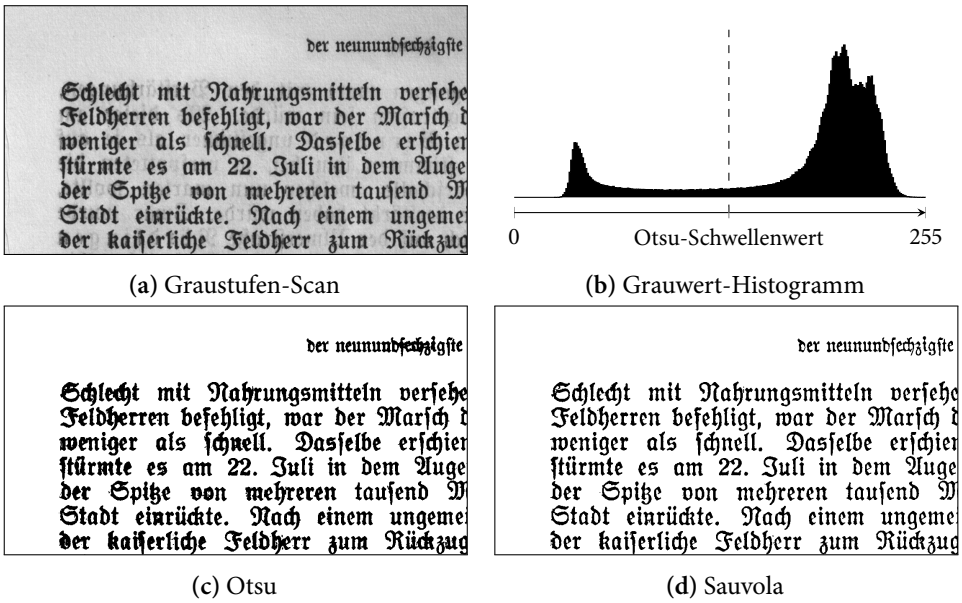
Bevor eine Binarisierung des Bilds vorgenommen werden kann, muss für die meisten Binarisierungsmethoden zunächst ein Graustufenbild vorliegen. Eine naive Berechnung des Graustufenwerts eines Bildpunkts als Mittelwert oder Maximum der Komponenten *Rot*, *Grün* und *Blau* führt in den meisten Fällen zu einer verfälschten Helligkeitsverteilung.

Das menschliche Auge nimmt grüne Farbtöne bei gleicher Lichtintensität heller wahr als rote und blaue Farbtöne. Diesen Wahrnehmungsunterschieden kann durch eine Konvertierung aus dem Farbraum *sRGB* in den Farbraum *CIE XYZ* über Formel 2.1 Rechnung getragen werden [61]. Im *XYZ*-Farbraum steht Y für die Luminanz. Somit ergibt sich der Helligkeitswert eines jeden Pixels durch Formel 2.2.

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} 0{,}4124 & 0{,}3576 & 0{,}1805 \\ 0{,}2126 & 0{,}7152 & 0{,}0722 \\ 0{,}0193 & 0{,}1192 & 0{,}9505 \end{bmatrix} \begin{bmatrix} R_{sRGB} \\ G_{sRGB} \\ B_{sRGB} \end{bmatrix} \tag{2.1}$$

$$Y = 0{,}2126 \cdot R_{sRGB} + 0{,}7152 \cdot G_{sRGB} + 0{,}0722 \cdot B_{sRGB} \tag{2.2}$$

Binarisierung Eine Möglichkeit, ein Graustufenbild mit L verschiedenen Grauwerten in ein Binärbild zu konvertieren, besteht darin, einen festen Schwellenwert $t \in \{0, 1, \ldots, L-1\}$ zu finden, der das Bild bestmöglich in schwarze und weiße Pixel unterteilt. Pixel, deren Grauwert kleiner oder gleich t ist, werden der Klasse der schwarzen Pixel C_0 zugeordnet. Alle weiteren Pixel gehören zur Klasse der weißen Pixel C_1. Einerseits lässt sich solch ein Schwellenwert durch Versuche manuell bestimmen. Andererseits gibt es jedoch auch zahlreiche Verfahren, um einen Schwellenwert durch statistische Methoden zu ermitteln. Eines der besten und schnellsten Verfahren [63] zur Ermittlung eines globalen Schwellenwerts ist die Methode von Otsu [44]. Das Verfahren ermittelt den optimalen Schwellenwert, indem es versucht, die Klassen C_0 und C_1 so zu wählen, dass die Varianzen der Grauwerte innerhalb der Klassen minimal sind.



(**a**) Graustufen-Scan (**b**) Grauwert-Histogramm

(**c**) Otsu (**d**) Sauvola

Abb. 2.5: Binarisierung eines Scan-Ausschnitts durch verschiedene Verfahren

Abbildung 2.5c zeigt das Ergebnis der Binarisierung des Graustufen-Scans in Abbildung 2.5a durch den Algorithmus. Das zum Scan gehörige Grauwert-Histogramm (s. Abbildung 2.5b) zeigt die Verteilung der Grauwerte im Scan, wobei 0 für schwarze und 255 für weiße Pixel steht. Die meisten Bildpunkte liegen im hellgrauen Bereich in der rechten Hälfte des Histogramms und stellen das Papier des Dokuments dar. Zudem gibt es ein kleineres Cluster von Pixeln nahe bei Schwarz. Alle weiteren Pixel liegen annähernd gleichmäßig verteilt zwischen diesen beiden Clustern. Der durch den Algorithmus ermittelte Schwellenwert liegt bei 133 und ist als gestrichelte Linie eingezeichnet. Die Stärke des Algorithmus liegt darin, dass er effizient implementiert werden kann [29] und gute Ergebnisse für ein Verfahren mit globalem Schwellenwert liefert [18, S. 10]. Er zeigt jedoch seine Schwäche bei Bildern, deren Histogramm kein eindeutiges Tal mit zwei Clustern aufweist. Das kann beispielsweise bei Texten mit starken Helligkeitsschwankungen im Hintergrund oder gleichmäßigen Farbverläufen der Fall sein.

Verbesserte Methoden zur Binarisierung stellen adaptive Verfahren wie das von Sauvola und Pietikäinen [49] dar. Das Verfahren liefert bessere Ergebnisse, indem für jeden Bildpunkt ein Fenster um den Bildpunkt in die Betrachtung mit einfließt. Für die Binarisierung von Textbereichen kommt dabei eine verbesserte Variante des Algorithmus von Niblack [40, S. 113–116] zum Einsatz, welche geringe Helligkeitsschwankungen im Hintergrund eher toleriert. Abbildung 2.5d zeigt das Ergebnis der Binarisierung durch den Algorithmus von Sauvola. Es ist deutlich zu erkennen, dass der Algorithmus im Vergleich zu Otsus Methode besser mit unsauberen Stellen des Scans zurecht kommt, er ist jedoch deutlich aufwändiger in der Berechnung.

Es existiert eine Vielzahl weiterer Verfahren zur Binarisierung von Bildern, die teilweise auch bessere Ergebnisse für bestimmte Kontexte liefern. Im Rahmen des regelmäßig stattfindenden *Document Image Binarization Contest (DIBCO)* werden neue Verfahren miteinander verglichen [23].

2.2.2 Segmentierung

Ziel der Segmentierung eines Dokuments ist es, Bereiche, die Text enthalten, von Bereichen mit Abbildungen zu unterscheiden. Zudem findet häufig in einer Layout-Analyse-Phase eine Abgrenzung von Textblöcken, Zeilen, Wörtern und Zeichen statt. Mögliche Vorgehensweisen zur Layout-Analyse sind *Bottom Up*- und *Top Down*-Ansätze. *Top Down*-Ansätze verwenden meist vortrainierte Muster für typische Seitenlayouts oder versuchen eine Seite rekursiv durch Schnitte in die relevanten Bereiche zu unterteilen [53]. Bei *Bottom Up*-Ansätzen wird ein Dokument zunächst in Zusammenhangskomponenten unterteilt. Diese werden dann in logische Einheiten wie Zeichen, Wörter, Zeilen usw. gruppiert.

Erster und wichtigster Schritt zur Segmentierung eines Dokuments nach dem *Bottom Up*-Ansatz ist die Erkennung von Zusammenhangskomponenten im binarisierten Bild. In der englischen Literatur werden diese gemeinhin als *Connected Components* oder als *Blobs* (kurz für „**B**inary **L**arge **Ob**jects“) bezeichnet. Um Blobs zu finden, wird ein sogenanntes Connected Component-Labeling durchgeführt, welches jedem Pixel eines Bilds ein Label in Form einer positiven Ganzzahl zuweist. Zusammenhängende Pixel gleicher Farbe erhalten das gleiche Label, wodurch zusammenhängende Komponenten eindeutig beschrieben sind.

Suzuki u. a. [62] listen eine Vielzahl verschiedener Connected Component-Labeling-Algorithmen auf. Meist sind diese auf mehrmaliges Durchlaufen des Binärbilds angewiesen. Als Verbesserung stellen Suzuki u. a. [62] jedoch auch einen effizienteren linearen Algorithmus vor. Ein weiterer, einfach nachzuvollziehender, linearer Algorithmus ist der Tracing-Algorithmus von Chang, Chen und Lu [17]. Der Algorithmus versucht, die Kontur einer Zusammenhangskomponente gezielt zu ermitteln, indem zu jedem gefundenen Randpixel einer Komponente der nächste Nachbar mit gleicher Farbe gesucht wird, bis wieder der Ausgangspunkt erreicht ist. Parallel weist er gefundenen Blobs ein eindeutiges Label zu. Der Vorteil dieses Vorgehens ist, dass die Pixel der Kontur eines Blobs als Liste zurückgeliefert werden können und somit die Implementierung der weiteren Erkennungskomponenten vereinfacht wird. Diese Zusammenhangskomponenten können je nach Sprache und abhängig von ihrer Lage und ihrem Abstand zueinander zu einzelnen Buchstaben, ganzen Wörtern, Zeilen und Textblöcken zusammengesetzt werden.

2.2.3 Merkmalsselektion und Klassifikation

Um eine Klassifikation eines gefundenen Zeichens durchführen zu können, müssen zunächst Merkmalsvektoren gefunden werden, die das Zeichen hinreichend genau beschreiben, es von anderen möglichen Zeichen abgrenzen und genügend nahe bei anderen Vertretern der gleichen Klasse liegen. Die Möglichkeiten zur Bestimmung der Merkmale eines Zeichens lassen sich grob den drei Bereichen *geometrische Merkmale*, *strukturelle Merkmale* und *Transformationen des Merkmalsraums* zuordnen [18, S. 54]. In der Praxis bestehen Verfahren zur Merkmalsselektion meist aus Kombinationen dieser Verfahren, um weniger anfällig für Fehler zu sein.

(a) Buchstabe 𝔅 **(b)** Degeneriertes 𝔅

Abb. 2.6: Verschiedene Buchstaben und deren Skelette

Geometrische und strukturelle Merkmale lassen sich auf unterschiedliche Arten beziehen. So gibt es Verfahren, die Merkmale aus den Momenten oder dem Histogramm eines Bildausschnitts berechnen [18, S. 55–59]. Häufig kommen auch Verfahren zum Einsatz, die Merkmale aus der Kontur eines Zeichens oder dessen Skelett berechnen, welche auch der Art und Weise nahekommen, wie der Mensch Buchstaben erkennt. Das Skelett eines Zeichens kann durch sogenanntes *Thinning* berechnet werden [18, S. 105].

Abbildung 2.6 zeigt die Kontur sowie das Skelett eines fehlerlosen 𝔅 sowie eines 𝔅 mit Unterbrechungen. Hierbei wird deutlich, dass die oben genannten Methoden anfällig gegenüber minderwertigem Bildmaterial sind. Durch die Unterbrechungen unterscheiden sich die Konturen der beiden Buchstaben deutlich. Skelette werden meist durch eine Reihe von Endpunkten, Gabelungspunkten und Kreuzungspunkten abstrahiert [18, S. 78], um so effizient Ähnlichkeiten zwischen Zeichen finden zu können. Das Skelett in Abbildung 2.6b teilt lediglich zwei Endpunkte mit dem fehlerfreien Skelett in Abbildung 2.6a. Kreuzungs- und Gabelungspunkte sind durch die Unterbrechungen verschwunden. Lösungen dieses Problems können beispielsweise die Zuhilfenahme anderer Merkmale oder die Betrachtung des Wort-Kontexts mit Hilfe des Sprachmodells sein und so den Buchstaben trotz schlechter Bildqualität korrekt klassifizieren.

Über Transformationen des Merkmalsraums kann nun versucht werden, dieselbe Information mit weniger Merkmalen darzustellen [18, S. 80]. Zwei Techniken zum Erreichen dieses Ziels sind *Principal Component Analysis* und *Linear Discriminant Analysis* sowie deren Varianten [18, S. 80–89].

Bei der Klassifikation kommen, abhängig von den gewählten Merkmalen, meist herkömmliche Klassifikatoren wie u. a. *Nächste-Nachbarn-Klassifikatoren* [18, S. 140–142], *Support Vector Machines* [18, S. 162–171] oder *künstliche neuronale Netze* [18, S. 142–162] zum Einsatz.

3 Übersicht über den Markt für OCR-Software

3.1 FineReader und Recognition Server

FineReader ist eine proprietäre, kommerzielle OCR-Software, die von der Firma ABBYY entwickelt und vertrieben wird [1]. Die erste Version erschien 1993 [5]. Die Software wird für die Betriebssysteme Microsoft Windows und Apple Mac OS X angeboten.

FineReader unterstützt mit Version 12 Professional die Erkennung von 190 Sprachen, wozu sowohl natürliche Sprachen als auch sechs Programmiersprachen wie C/C++ und Java gezählt werden. Jedoch existieren nur für 48 dieser Sprachen Wörterbücher [2]. Die Software verfügt zudem über grafische Werkzeuge zum Training neuer Zeichensätze, Fonts und Sprachen, die es Nutzern ermöglichen, spezielle Anwendungsfälle zu bearbeiten. Für die Erkennung von gebrochenen Schriften existiert mit *Historic OCR* bzw. *FineReader XIX* eine angepasste Version von *FineReader* [3]. Diese ist auf die Erkennung von gebrochenen Schriften und Fraktur im Speziellen ausgerichtet. Der Vertrieb dieser Version wurde jedoch 2010 eingestellt und die Funktionalität wurde in *Recognition Server* integriert, welcher im Gegensatz zu *FineReader* für den automatisierten Einsatz auf Server-Systemen entwickelt wurde [4].

Neben den offiziellen Produkten der Firma ABBYY existieren auch noch spezielle OCR-Programme, die von Dritt-Anbietern speziell für große Digitalisierungsprojekte angeboten werden, so beispielsweise die Software *HK-OCR* von der Firma Herrmann & Kraemer, welche auf ABBYY *FineReader Engine 9* basiert [19, S. 7].

ABBYY ist mit seinen Produkten Marktführer im Bereich für OCR-Software und wird daher auch in vielen Digitalisierungsprojekten der Universitätsbibliothek Würzburg sowie an einigen weiteren Universitäten eingesetzt [19, 28].

3.2 Tesseract

Tesseract wurde von der Firma Hewlett-Packard von 1984 bis 1994 entwickelt [52] und erstmals 1995 im Rahmen des vierten *Annual Test of OCR Accuracy* der University of Nevada, Las Vegas als „HP Labs OCR“ vorgestellt [47]. Bei diesem Wettbewerb war das System im Bezug auf die Erkennungsgenauigkeit unter den besten drei von insgesamt acht Kandidaten. Die Software wurde ab 1995 jedoch nicht mehr weiterentwickelt und kam auch nicht als Produkt auf den Markt. 2005 wurde das System schließlich von Hewlett-Packard unter dem Namen Tesseract veröffentlicht. Tesseract ist seither Open Source-Software und steht unter der Apache Software License 2.0. Seit 2006 wird die Entwicklung durch Google finanziert.

Ursprünglich wurde Tesseract ausschließlich für englischsprachige Texte entwickelt. Seit Version 2.00 aus dem Jahr 2007 werden offiziell auch die Sprachen Französisch, Italienisch, Deutsch, Spanisch und Niederländisch unterstützt [55]. Mit Version 2.02 wurde außerdem die Erkennung von Sprachen mit großen Zeichensätzen wie beispielsweise Kannada verbessert [55]. Tesseract bietet für die offizielle aktuelle Version 3.02 aus dem Jahr 2012 Sprachpakete für mehr als 60 Sprachen und steht für die Betriebssysteme Linux, Microsoft Windows und Mac OS X zur Verfügung. Die Veröffentlichung von Version 3.03 wurde nach einer langen Entwicklungsphase übersprungen, es wurde lediglich ein Release Candidate veröffentlicht. Aktuell arbeitet man an Version 3.04. Für diese Arbeit wurde Tesseract ausgewählt, da es sich dabei um kostenlose und lizenzfrei verfügbare Open Source-Software handelt. Die so entwickelten Programmkomponenten können deshalb problemlos in weiteren Projekten eingesetzt werden. Außerdem kann bei eventuellen Problemen der Texterkennung auch der Programmcode von Tesseract selbst analysiert oder sogar verbessert werden, was bei propietärer Software nicht möglich ist.

4 Analyse von Tesseract

Um die Erkennungsergebnisse besser nachvollziehen und beeinflussen zu können, ist es zunächst wichtig, die Funktionsweise von Tesseract zu analysieren. Eine gute Übersicht über Aufbau und Funktionsweise von Tesseract bietet Smith [52]. Er lässt dabei jedoch viele Details unerwähnt. Die Präsentation von Smith [53] zu Tesseract liefert zusätzlich viele Erläuterungen zu Implementierungsentscheidungen. In diesem Kapitel soll zunächst der Funktionsumfang und anschließend der Aufbau der Software sowie die Funktionsweise kritischer Komponenten nachvollzogen werden. Anschließend werden bestehende Sprach-Pakete auf ihre Tauglichkeit zur Erkennung von Frakturtexten untersucht, was die Grundlage für die Trainingsexperimente in Kapitel 5 bildet.

Für die Analyse wurde der Quellcode von Tesseract in einer vorläufigen Version von Release 3.04 herangezogen und für Microsoft Windows 7 kompiliert [66, 67]. Diese Version wurde gewählt, um auch neuere Entwicklungen des Projekts in die Evaluation mit einfließen zu lassen. Das vorhergehende Release (3.02.02) wurde vor etwas weniger als zwei Jahren veröffentlicht. Seither wurde eine Vielzahl neuer Funktionen eingeführt [55], beispielsweise die Erzeugung von durchsuchbaren PDF-Dateien des Scanmaterials. Es wurden aber auch einige Probleme der Texterkennung beseitigt.

Seit Version 3.00 baut Tesseract grundlegend auf der Bildverarbeitungs- und Bildanalyse-Software *Leptonica* [11] auf und ermöglicht damit die Verarbeitung der Bilddateiformate TIFF, PNG und JPEG sowie zahlreicher anderer Bildformate. Sogar TIFF-Dateien, die mehrere Seiten haben, werden unterstützt. Allerdings müssen die dafür notwendigen Software-Bibliotheken bei der Erstellung von *Leptonica* konfiguriert werden. Für die Analyse wurden jedoch ausschließlich einseitige TIFF-Dateien sowie PNG-Dateien verwendet.

Tesseract bietet neben einem Programm, über das die Texterkennung gesteuert werden kann, auch noch zahlreiche weitere Kommandozeilen-Programme für das Training neuer Sprachen und Schriftarten, die in Abschnitt 4.2 erläutert werden. Alle Programme müssen über die Kommandozeile des Systems aufgerufen werden. Eine grafische Benutzungsoberfläche ist nicht Teil des Projekts, was die Verwendung der Programme für unerfahrene Nutzer erheblich erschwert. Es existieren jedoch zahlreiche unabhängige Projekte [45], die versuchen, die Bedienung von Tesseract durch eine grafische Schnittstelle zu erleichtern. Zusätzlich gibt es zahlreiche grafische Werkzeuge, die das Training neuer Sprachen erleichtern. Diese Erweiterungen werden durch eine Programmierschnittstelle für die Programmiersprache C ermöglicht. Über diese ist es möglich, neben den herkömmlichen Dateiausgabeformaten auch noch weitere Informationen zur Texterkennung zu erhalten (vgl. Kapitel 6.1). Außerdem können repetitive Aufgaben über die Schnittstelle automatisiert werden.

Der größte Teil des Quellcodes von Tesseract wurde ursprünglich in C geschrieben, innerhalb der letzten Jahre jedoch in C++ für Objektorientierung umgeschrieben, um so die enthaltenen Prozesse leichter nachvollziehbar zu machen. Außerdem wurde damit die langfristige Wartung der Software erleichtert. Insgesamt umfasst das Projekt etwas weniger als 300 000 Zeilen Quellcode. Davon entfallen fast 200 000 Zeilen auf C++-Code, die weiteren Zeilen entfallen auf Konfigurations- und Automatisierungsskripte sowie ein visuelles Debugging-Tool, dessen grafische Oberfläche in Java geschrieben wurde.

Tesseract unterstützt in den aktuellen Entwicklungsversionen vier Ausgabeformate für die Ergebnisse der Zeichenerkennung. Zum einen sind dies reine Textdateien, die keinerlei Auszeichnungen neben originalgetreuen Zeilenumbrüchen sowie Leerzeilen bei Absätzen enthält. Erkannte Bilder werden im Text nicht ausgegeben. Die Ausgabe durch Textdateien ist standardmäßig aktiv.

Das zweite Ausgabeformat sind HTML-Dateien, die zusätzliche Auszeichnungen nach dem hOCR-Standard [13] enthalten. Unter anderem werden hierin Informationen über gefundene Textbereiche, Absätze, Zeilen und Wörter gespeichert. Zu jeder Einheit der Hierarchie werden die Koordinaten der zugehörigen umschließenden Box im title-Attribut des entsprechenden HTML-Elements kodiert. Bei Zeilen wird zusätzlich die ermittelte Grundlinie als Polynom mit zwei Koeffizienten gespeichert.[1] Das title-Attribut jedes Wort-Elements enthält zudem den von Tesseract bei der Zeichenklassifizierung ermittelten Konfidenzwert des Worts als Prozentangabe zwischen 0 und 100. Dieser gibt an, wie sicher Tesseract sich bei der Entscheidung für die Zeichen des Worts war. Sind die Zeichen deutlich zu erkennen und ist das Wort im Wörterbuch enthalten, liegt dieser Wert meist über 80 Prozent. Bei Unsicherheiten und unbekannten Wörtern kann der Wert auch deutlich darunter liegen. Um statt einer Textdatei eine HTML-Datei als Ausgabe zu erhalten, muss beim Aufruf von Tesseract die Konfigurationsdatei „hocr“ als zusätzliches Argument angegeben werden.

Weiter gibt es auch noch die Möglichkeit, ein PDF mit den Scans zu erzeugen, das anschließend nach Begriffen durchsucht werden kann. Dazu legt Tesseract einen transparenten Text über das PDF genau an die Stellen, wo Text gefunden wurde. Diese Datei erhält man durch Anhängen der Konfigurationsdatei „pdf“ als zusätzliches Argument.

Schließlich existiert auch noch ein weiteres Dateiformat für die Ausgabe, welches sich für Vergleiche mit den Ergebnissen des *Annual Test of OCR Accuracy* [47] eignet.

[1] Für Tesseract erfolgt die Ausgabe der Grundlinie im hOCR-Format als zwei Koeffizienten a und b eines Polynoms $f(x) = ax + b$. Die Steigung der Grundlinie wird also durch a beschrieben, b bezeichnet den negativen Abstand des unteren, linken Eckpunkts der umgebenden Box zur Grundlinie. Für nicht-lineare Grundlinien könnten nach dem Standard auch weitere Koeffizienten angegeben werden. Tesseract abstrahiert gefundene gebogene Grundlinien im hOCR-Format jedoch immer durch eine Gerade.

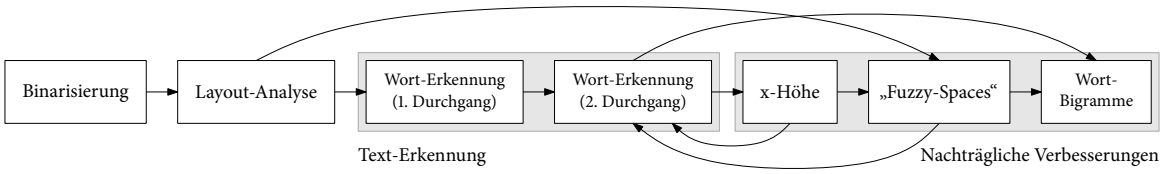


Abb. 4.1: Aufbau von Tesseract nach Smith [53]

4.1 Funktionsweise der Texterkennung

4.1.1 Aufbau

Wie viele OCR-Programme besteht Tesseract aus einer Pipeline von Komponenten, die einen weitestgehend linearen Ablauf der Texterkennung ermöglichen. Manche Komponenten können jedoch auch übersprungen oder wiederholt werden (vgl. Abbildung 4.1), sodass viele Entscheidungen nicht endgültig gefällt werden, sondern zu einem späteren Zeitpunkt und mit zusätzlichem Wissen erneut getroffen werden können [53].

Erster Schritt für die Erkennung von Text ist die Binarisierung eines Bildes durch das Verfahren von Otsu [44]. Liegt das zu analysierende Bild bereits als Binärbild vor, wird das binarisierte Bild unverändert übernommen. Unmittelbar nach der Binarisierung erfolgt eine Connected Component-Analyse, welche die Kontur einer jeden Zusammenhangskomponente für die weiteren Komponenten bereithält, wie es bereits in Abschnitt 2.2.2 beschrieben wurde. Nach der Layout-Analyse erfolgt die eigentliche Texterkennung der ermittelten Textbereiche. Sobald die einzelnen Zeichen klassifiziert wurden, folgen noch verschiedene Komponenten, die versuchen die Erkennungsergebnisse über das vorhandene Sprachmodell zu verbessern.

4.1.2 Layout-Analyse

Ziel der Layout-Analyse ist es, eine Seite in Bereiche mit und ohne Text zu unterteilen sowie mehrspaltigen Text zu erkennen und die Spalten voneinander zu trennen. Die Komponente zur Layout-Analyse versucht jedoch nicht, eine logische Unterteilung des Layouts in Überschriften, Textabsätze oder Kopf- und Fußzeilen vorzunehmen.

Im ersten Schritt der Layout-Analyse werden lange, vertikale Linien im Bild entfernt. Anschließend werden Initialen, Unterstreichungen und isolierte Bereiche herausgefiltert. Dabei ist nicht entscheidend, dass die Klassifizierung der Komponenten fehlerfrei erfolgt. Die herausgefilterten Komponenten werden nach der Zeilenerkennung wieder der jeweiligen Zeile zugeordnet und dienen lediglich zur Entscheidung, ob es sich um Text handelt oder nicht.

Anschließend wird der Median der Pixelhöhe der verbliebenen Zusammenhangskomponenten berechnet. Zusammenhangskomponenten, die weniger Pixel in der Höhe als ein Bruchteil des Medians haben, werden ignoriert. Anschließend werden die Komponenten der Höhe nach sortiert und die kleinsten zwanzig Prozent, die größten fünf Prozent sowie Komponenten, die sehr breit sind, wiederum

herausgefiltert. Sind keine klaren Zwischenräume zwischen mehreren Zeilen erkennbar, wird ein Bildbereich als Abbildung deklariert und von der Texterkennung ausgenommen. Die Unterteilung eines gefundenen Textblocks in Absätze erfolgt über die sogenannte Tab-Stop Detection [54].

Wörter und Zeichen werden in einem ersten Schritt ausschließlich durch die Abmessungen der ermittelten Blobs segmentiert [53]. Erst bei der eigentlichen Klassifikation wird versucht, die Erkennungswahrscheinlichkeiten durch eine Änderung der Segmentierung zu verbessern, indem Zeichen an schmalen Stellen aufgetrennt werden [52, 53].

4.1.3 Polygon-Approximation und Normalisierung

Als Zeichen segmentierte Blobs werden nach der Layout-Analyse durch ein Polygon approximiert. Dazu werden gerade Segmente des Umrisses eines Zeichens gesucht. So entsteht eine Folge von geraden Abschnitten und Eckpunkten, welche den Umriss des Zeichens approximiert. Die Normalisierung, Merkmalsextraktion und Klassifikation werden auf diesem approximierten Polygon durchgeführt.

Um Merkmale für jedes Zeichen bestimmen zu können, wird außerdem eine Normalisierung der Zeichen durchgeführt. Hierdurch werden verschiedene Schriftgrößen vereinheitlicht, sodass Tesseract nicht für jede Schriftgröße separat trainiert werden muss. Momentan werden in Tesseract zwei verschiedene Methoden zur Normalisierung eingesetzt.

Bei der ersten Methode, der sogenannten *Baseline Normalization*, wird ein Zeichen horizontal zentriert und vertikal an der Grundlinie und x-Linie orientiert, jedoch nicht skaliert [53]. Außerdem wird ein Zeichen so rotiert, dass es waagerecht auf der Grundlinie liegt. Der Mittelpunkt des Zeichens auf der Grundlinie bildet dabei das Rotationszentrum. Vorteil dieser Normalisierung ist, dass Groß- und Kleinbuchstaben besser voneinander unterschieden werden können, wenn sie ähnlich aussehen [52].

Die zweite Methode, die sogenannte *Character Normalization*, ist die Normalisierung anhand der zweiten Momente in x- und y-Richtung des Umrisses eines Zeichens [53]. Diese Momente werden durch die Formeln 4.1 und 4.2 berechnet. Dabei handelt es sich um die Standardabweichung der Koordinaten der n Randpixel eines Blobs in x- und y-Richtung. Das Zentrum der Normalisierung bildet der Mittelpunkt der x- und y-Koordinaten. Ein Vorteil dieser Normalisierung ist, dass auch in einer Richtung gestreckte oder gestauchte Zeichen besser erkannt werden können [52]. Daraus folgt aber auch der Nachteil, dass Zeichen wie „-“, „—“ und „|“ nach der Normalisierung gleich aussehen.

$$m_x = \sqrt{\frac{\sum_{i=0}^{n} x_i^2 - \frac{\sum_{i=0}^{n} x_i \cdot \sum_{i=0}^{n} x_i}{n}}{n}} \qquad m_y = \sqrt{\frac{\sum_{i=0}^{n} y_i^2 - \frac{\sum_{i=0}^{n} y_i \cdot \sum_{i=0}^{n} y_i}{n}}{n}} \tag{4.1, 4.2}$$

Es existiert noch eine weitere, nicht-lineare Normalisierung, die derzeit jedoch noch nicht zum Einsatz kommt [53].

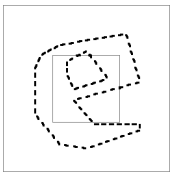


Abb. 4.2: Merkmale des Buchstaben e nach der Moment-basierten Normalisierung

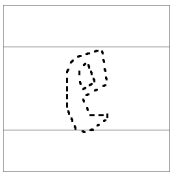


Abb. 4.3: Merkmale des Buchstaben e nach der Normalisierung anhand der x-Höhe

4.1.4 Merkmalsextraktion

Die Merkmalsextraktion eines Zeichens erfolgt auf der normalisierten Polygon-Approximation des Zeichenumrisses. Beim Training werden die geraden Segmente des Polygons in die Zeichenklasse (Prototyp) aufgenommen. Für jedes gerade Segment wird hierfür ein vierdimensionaler Vektor $\vec{p}$ berechnet. Dieser besteht aus den Koordinaten des Startpunkts (x, y), der Länge l des Segments sowie dem Neigungswinkel θ: $\vec{p} = (x, y, \theta, l)$. Jede dieser Komponenten wird auf den ganzzahligen Bereich zwischen 0 und 255 skaliert, um einen Merkmalsvektor als Folge von vier Bytes abspeichern zu können.

Die bei der Klassifikation eines unbekannten Zeichens ermittelten Merkmale sind jedoch nur dreidimensional und bestehen aus den Koordinaten eines Punkts auf einem Polygonsegment (x, y) sowie der Neigung θ an diesem Punkt: $\vec{f} = (x, y, \theta)$. Auch diese Werte werden auf den Bereich zwischen 0 und 255 skaliert. Anstatt die Länge des Segments als vierte Komponente hinzuzunehmen, werden mehrere solche dreidimensionale Merkmalsvektoren für ein Segment bestimmt. Um eine zuverlässige Klassifikation von Zeichen auch bei Beschädigungen zu ermöglichen, wird der Abstand der gewählten Punkte hierbei gering gehalten. Dadurch passt auch bei Unterbrechungen des Zeichens immer noch die Mehrzahl der Merkmalsvektoren zum Prototyp. Die Abbildungen 4.2 und 4.3 zeigen die ermittelten, dreidimensionalen Merkmalsvektoren des Buchstaben e nach der Normalisierung des Eigenschaftsraums.

4.1.5 Klassifikation

Die eigentliche Klassifikation wird von einem k-nächste-Nachbarn-Klassifikator übernommen. Die Distanz zwischen mehreren Merkmalsvektoren $\vec{f}_i$ und den Prototyp-Vektoren $\vec{p}$ berechnet sich aus dem senkrechten Abstand von $\vec{f}_i$ zu $\vec{p}$ (vgl. Formel 4.3) sowie dem Winkel zwischen beiden Vektoren (vgl. Formel 4.4). Die kombinierte Distanz zwischen den Vektoren ist durch Formel 4.5 definiert. Die Komponenten dieser Distanzmaße sind in Abbildung 4.4 und Abbildung 4.5 dargestellt.

$$d(\vec{f}, \vec{p} = \vec{a} + t \cdot \vec{n}) = \left\|(\vec{a} - \vec{f}) - \left((\vec{a} - \vec{f}) \cdot \vec{n}\right) \cdot \vec{n}\right\| \qquad \alpha = \cos^{-1}\left(\frac{\vec{f} \cdot \vec{p}}{\|\vec{f}\| \cdot \|\vec{p}\|}\right) \qquad (4.3, 4.4)$$

$$d_{\mathrm{fp}} = \left(d(\vec{f}, \vec{p})\right)^2 + \alpha^2 \qquad (4.5)$$

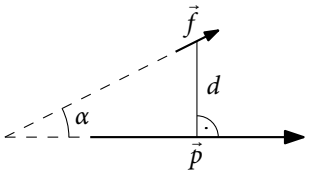


Abb. 4.4: Abstand und Winkel zwischen den Merkmalsvektoren $\vec{f}$ und $\vec{p}$

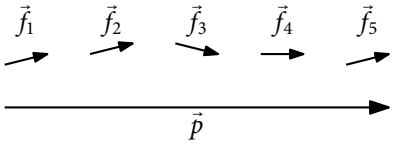


Abb. 4.5: Vergleich mehrerer Merkmalsvektoren $\vec{f}_i$ mit einem Prototypenvektor $\vec{p}$

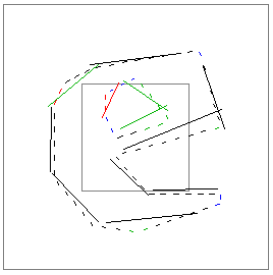


Abb. 4.6: Visualisierung der Klassifikation des Buchstaben e

Über diese Maße kann das Zeichen durch den Vergleich mehrerer Prototypen mit den ermittelten Merkmalsvektoren klassifiziert werden. Da hierdurch jedoch bei einem großen Zeichensatz und vielen Schriftarten sehr viele Berechnungen notwendig sind, werden die möglichen Klassen vor der Klassifikation durch einen sogenannten „Class Pruner“ eingeschränkt [53].

Abbildung 4.6 veranschaulicht die Klassifikation des Zeichens e. Durchgezogene Linien stellen dabei die Merkmale des Prototyps dar. Gestrichelte Linien stehen für die Merkmale des zu klassifizierenden Zeichens. Zusammengehörige Linien werden in gleichen Farben dargestellt.

4.1.6 Sprachmodell

Das durch Tesseract verwendete Sprachmodell ist vergleichsweise simpel. Es existieren lediglich zwei Listen von bekannten Wörtern. Eine dieser Listen enthält den gesamten Wortschatz. Die andere Liste enthält nur häufig vorkommende Begriffe. Ist das durch die Klassifizierung ermittelte Wort in einer Wortliste enthalten, wird es im Idealfall direkt akzeptiert. Andernfalls wird versucht eine bessere Segmentierung der Blobs zu Zeichen und Wörtern zu finden. Kann auch durch eine alternative Segmentierung kein Begriff aus dem Wörterbuch gefunden werden, so können einzelne Polygone auch an konkaven Stellen aufgetrennt werden. Das kann beispielsweise bei Buchstabenkombinationen wie „rn“ notwendig sein, die häufig zu einem Buchstaben verschmelzen und dadurch als „m“ klassifiziert werden.

4.2 Training neuer Sprachen und Schriftarten

Seit Version 2.00 bietet Tesseract die Möglichkeit, die Texterkennung für weitere Sprachen und Schriftarten zu trainieren. Dazu enthält Tesseract eine Reihe weiterer Werkzeuge. Die allgemeine Vorgehensweise

für das Training von Tesseract wird auf der Dokumentationsseite „Training Tesseract 3“ des Projekts beschrieben [58], soll hier aber zum besseren Verständnis der weiteren Kapitel erläutert werden.

Trainingsbeispiele Um Tesseract für neue Sprachen oder neue Schriftarten zu trainieren, benötigt man zunächst Bilder, die einen Text in der gewünschten Schriftart enthalten. Dafür eignen sich sowohl Scans, die später auch erfasst werden sollen, aber auch durch ein Bildbearbeitungs- oder Textverarbeitungsprogramm erstellte Texte. Letztere können einerseits ausgedruckt und wieder eingescannt werden, um eine Rastergrafik aus dem Text zu erzeugen. Andererseits existieren Konvertierungsprogramme wie beispielsweise *ImageMagick* [30], die PDF-Dateien direkt in Rastergrafiken umwandeln können. Bei Bildbearbeitungsprogrammen besteht dieses Problem nicht. Erzeugte Trainingsbeispiele können hier direkt in den von Tesseract unterstützten Grafikformaten gespeichert werden. PNG und TIFF eignen sich durch ihre Möglichkeit zur verlustfreien Komprimierung des Bildmaterials am besten für den Trainingsprozess.

Als Bildbearbeitungsprogramme eignen sich unter anderem die Programme *Adobe Photoshop* [6] und *The GIMP* [31], die beispielsweise auch die automatische Verwendung von Ligaturen oder des langen ſ bei bestimmten OpenType-Schriftarten unterstützen. Mit herkömmlichen Textverarbeitungsprogrammen wie *Microsoft Word* [35] oder *OpenOffice Writer* [8] aber auch *LaTeX* [32] lassen sich ebenfalls Trainingsbeispiele erzeugen. Die dritte Möglichkeit bietet das Programm *text2image*, welches ab Version 3.03 Teil von Tesseract ist. Das Kommandozeilen-Programm ermöglicht es, mit Hilfe einer Textdatei und einer Systemschriftart automatisch Trainingsbeispiele zu erstellen. Ähnliche Funktionen bietet auch *jTessBoxEditor* [39].

Damit sich Trainingsbeispiele synthetisch erzeugen lassen, müssen die zu erlernenden Schriftarten als Systemschriftarten auf dem System installiert sein. Oftmals existieren jedoch keine Computerschriftarten für die in historischen Texten verwendeten Schriften und es muss auf ähnliche Schriftarten oder bereits existierende Scans zurückgegriffen werden. In einer früheren Arbeit [65] wurden jedoch bereits Werkzeuge und Methoden vorgestellt, um mit Hilfe der Layoutanalyse und Zeichensegmentierung von Tesseract sowie dem Font-Editor *FontForge* [73] TrueType-Fonts aus historischen Texten zu erzeugen. Diese können dann wiederum für die Erzeugung von synthetischen Trainingsbeispielen verwendet werden.

Werden Trainingsbeispiele synthetisch erzeugt, müssen einige Regeln beachtet werden. Tesseract muss die Erkennung der Grundlinien und x-Höhen erleichtert werden, damit diese Informationen auch für jedes Zeichen erlernt werden können. Dafür sollten auf jeder Zeile des Trainingsbeispiels einige Zeichen enthalten sein, die Ober- und Unterlängen der Zeile abdecken. Daraus folgt auch, dass Satzzeichen, Zahlen und Sonderzeichen nicht für sich auf einer eigenen Zeile stehen sollten, da sonst die Zeilen- und x-Höhe nicht korrekt bestimmt werden kann. Stattdessen sollte man darauf achten, dass diese Zeichen gleichmäßig über das Trainingsbeispiel verteilt sind, um so einen natürlichen Text nachzubilden [58]. Der Inhalt des Trainingsbeispiels muss jedoch nicht zwingend Sinn ergeben. Das

Trainingsbeispiel hat keinen Einfluss auf das zur Erkennung verwendete Sprachmodell. Dieses wird in einem späteren Schritt hinzugefügt.

Weiterhin muss darauf geachtet werden, dass sich einzelne Zeichen nicht horizontal überlappen, da sonst Eigenschaften falsch erlernt werden können. Um dies zu verhindern, führen sich überlappende Zeichen zu einer Fehlermeldung im späteren Trainingsprozess [58], der aber schon im Vorhinein vermieden werden sollte, um nachträgliche Anpassungen und unnötige Wiederholungen zu vermeiden, da diese zeitintensiv sein können. Ligaturen sind von dieser Einschränkung jedoch ausgenommen. Sie können so trainiert werden, als handele es sich dabei um ein einzelnes Zeichen.

Beim Training von bereits existierenden Scans muss ebenfalls darauf geachtet werden, dass sich einzelne Zeichen nicht überlappen. Außerdem ist es wichtig, dass für jedes Trainingsbeispiel nur eine einzige Schriftart verwendet wird [58]. Dabei gilt auch jede Variante einer Schriftfamilie (fett, kursiv usw.) als eigene Schriftart. Für das Training einer Sprache lassen sich bis zu 64 unterschiedliche Varianten eines Symbols trainieren [58], wobei das Vorhandensein von 358 unterschiedlichen Schriftarten in der englischen Sprachdatei darauf hindeutet, dass diese Begrenzung durch neuere Trainingswerkzeuge umgangen werden kann.

Um alle Möglichkeiten des Trainingsprozesses ausschöpfen zu können, empfiehlt es sich, die Trainingsbeispiele nach einem festen Namensschema zu benennen [58]. Die Dateinamen setzen sich aus der Bezeichnung des resultierenden Sprachpakets, dem Namen der Schriftvariante sowie einer Zahl, durch die verschiedene Beispiele einer Schriftvariante unterschieden werden. Die Sprachbezeichnung sollte dem Standard ISO 639-3 folgen, also aus drei Buchstaben bestehen. Wahlweise kann der Sprachbezeichnung ein Suffix durch einen Binde- oder Unterstrich getrennt hinzugefügt werden, um das Paket von anderen abzugrenzen. Für das erste Beispiel eines deutschen Frakturtexts in der Schriftart Normalfraktur eignet sich beispielsweise der Dateiname „deu-fraktur.normalfraktur.exp0.tif“.

Box-Dateien Zweiter Trainingsschritt ist die Erzeugung einer sogenannten Box-Datei für jedes Trainingsbeispiel. Als Box wird dabei das Rechteck bezeichnet, das sich um ein einzelnes Zeichen oder eine Ligatur legen lässt, sodass alle Pixel des Zeichens enthalten sind. Die Box-Datei ist eine Textdatei, die in jeder Zeile die entsprechenden, UTF-8 kodierten Zeichen sowie die Koordinaten der zugehörigen Box und die Seitennummer angibt. Letztere ist nur für mehrseitige TIFF-Dateien relevant, sonst ist die Seitenangabe stets „0“. Alle Angaben werden jeweils durch ein einzelnes Leerzeichen getrennt. Der Ursprung des Koordinatensystems einer Box-Datei ist die untere linke Ecke des zugehörigen Trainingsbeispiels. Abbildung 4.7 zeigt einen kurzen Ausschnitt aus einer solchen Box-Datei.

Tesseract bietet die Möglichkeit, eine solche Box-Datei automatisch aus einem Trainingsbeispiel heraus zu erzeugen. Dazu müssen zusätzlich zum üblichen Aufruf über die Kommandozeile die Konfigurationsdateien „batch.nochop“ und „makebox“ angegeben werden. Diese sorgen dafür, dass von Tesseract neben der üblichen Ausgabe des erkannten Texts als einfache Textdatei auch eine Box-Datei mit allen erkannten Glyphen erstellt wird. Diese hat den gleichen Dateinamen wie die Ausgabedatei

```
f 1332 2290 1346 2347 0
ch 1348 2290 1385 2347 0
r 1392 2301 1411 2333 0
e 1415 2300 1433 2333 0
```

Abb. 4.7: Auszug aus einer Box-Datei

Abb. 4.8: Darstellung des gleichen Ausschnitts durch *jTessBoxEditor*

```
T 5 59,68,216,255,85,227,0,47,88,236 Latin 30 0 63 T      # T [54 ]A
c 3 58,64,192,200,80,153,0,36,88,178 Latin 18 0 74 c      # c [63 ]a
ft 3 0,68,206,255,70,353,0,43,113,353 Latin 69 0 33 st    # ft [17f 74 ]a
? 10 40,67,219,255,59,144,0,65,77,188 Common 82 10 82 ?   # ? [3f ]p
```

Abb. 4.9: Auszug aus einer „unicharset“-Datei

jedoch mit der Dateiendung „.box“. Das Ergebnis der automatischen Erzeugung ist häufig nicht fehlerfrei und muss für die weiterführenden Schritte des Trainings verbessert werden.

Die Qualität der Box-Datei hängt von der verwendeten Sprachdatei ab. Es empfiehlt sich, für das Training einer neuen Sprache oder Schriftart eine Sprachdatei zu verwenden, die einen ähnlichen Zeichensatz und ähnliche Schriftarten oder unter Umständen bereits die gleiche Sprache enthält. So kann der Aufwand bei der Korrektur der Box-Dateien gering gehalten werden. Ist eine solche Datei nicht vorhanden und sollen mehrere Trainingsbeispiele mit der gleichen Schriftart trainiert werden, kann es hilfreich sein, zunächst ein vollständiges Training für das erste Beispiel durchzuführen. Mit der so erstellten Trainingsdatei können anschließend Box-Dateien für alle weiteren Trainingsbeispiele mit höherer Genauigkeit generiert werden.

Das händische Bearbeiten der Box-Dateien mit einem herkömmlichen Texteditor ist nicht wirtschaftlich. Um die Koordinaten einer Box zu überprüfen, muss zuerst die entsprechende Stelle im Trainingsbeispiel gefunden und bei Bedarf verbessert werden, was sehr viel Zeit beanspruchen würde. Daher gibt es für diesen Zweck eine Vielzahl von Box-Editoren, die eine grafische Verbesserung der Box-Dateien ermöglichen. Hierzu werden alle Boxen als Rechtecke im Trainingsbeispiel eingezeichnet (vgl. Abb. 4.8). Falsch positionierte Boxen können verschoben, vergrößert und verkleinert werden. Außerdem können mehrere Boxen zu einer zusammengefasst werden. Auch das Einfügen neuer Boxen ist möglich. Darüber hinaus kann auch die Zuordnung zu Zeichen für jede Box verbessert werden. Ein solches Programm, welches auch mehrseitige TIFF-Dateien unterstützt, ist *jTessBoxEditor* [39].

Zeichenvorrat Aus den so entstandenen Box-Dateien kann nun der unterstützte Zeichenvorrat abgeleitet werden. Hierfür wird das Programm *unicharset_extractor* verwendet. Das Programm berechnet die Gesamtmenge aller in den Box-Dateien verwendeten Zeichen und listet diese in der Datei „unicharset“ auf. Neben den UTF-8-Zeichen werden hierin Informationen über die Art des Zeichens[2], verschiedene

[2] Der Zeichentyp wird durch eine fünfstellige Binärzahl kodiert, welche aber als Hexadezimalzahl notiert wird. Ist die erste Ziffer der Binärzahl gleich eins, so handelt es sich um ein Sonderzeichen. Die zweite Ziffer steht für Ziffernzeichen (0

maximale und minimale Abmessungen, die Art des Alphabets (bspw. Lateinisch, Kyrillisch usw.), die eventuell zugehörige Minuskel- oder Majuskelform sowie einige weitere Informationen zur Fehleranalyse abgelegt. Abbildung 4.9 zeigt einen Auszug einer solchen Datei.

Die Informationen über maximale und minimale Abmessungen eines Zeichens sind erst mit Version 3.02 eingeführt worden [43]. Sie werden initial durch *unicharset_extractor* auf die jeweiligen maximalen oder minimalen Grenzwerte gesetzt, um die Erkennung nicht zu beeinflussen. Anschließend können die Werte durch den Nutzer angepasst werden, um so die Erkennung zu verbessern. Die Anpassung ist jedoch optional. Da die manuelle Eingabe der Werte viel Zeit in Anspruch nehmen würde, gibt es in den Quellendateien von Tesseract das Verzeichnis „langdata", welches Standardwerte für die wichtigsten Alphabete und Zeichen vorgibt. Über das Programm *set_unicharset_properties* lassen sich die Werte aus diesem Verzeichnis automatisch in das zuvor erzeugte „unicharset" übernehmen. Danach können einzelne Werte je nach Bedarf angepasst werden. *set_unicharset_properties* wurde erstmals mit der Vorabversion von Release 3.03 veröffentlicht. Die Trainingsprogramme dieser Version lassen sich bisher aber nur unter Linux kompilieren und verwenden [57].

Font-Eigenschaften In der Datei „font_properties", welche mit Version 3.01 eingeführt wurde, können Tesseract allgemeine Merkmale der zu trainierenden Schriftvariante mitgeteilt werden. Jede Zeile der Textdatei steht dabei für eine Schriftart. Zunächst sind dies der Name einer Schriftart, wie beispielsweise „fraktur_normal" oder „times_italic". Diese Bezeichnung der Schriftart muss dem entsprechenden Segment aus dem Dateinamensschema des zugehörigen Trainingsbeispiels entsprechen, und darf keine Leerzeichen enthalten. Anschließend folgen, jeweils durch Leerzeichen getrennt, Informationen über die Eigenschaften der Schriftart wie *kursiv*, *fett*, *nichtproportional*, *Serifen* und *Fraktur*. Trifft eine Eigenschaft zu, so steht eine 1, andernfalls steht eine 0. Die so kodierten Informationen können später bei der Texterkennung über die Programmierschnittstelle von Tesseract abgefragt werden, um so neben dem bloßen Text auch Informationen über die Formatierung des Texts zu erhalten. Sie haben sonst jedoch keine Bedeutung für die Erkennung.

Training Im eigentlichen Trainingsschritt werden anhand der Box-Dateien und der zugehörigen Rastergrafiken die drei- und vierdimensionalen Merkmalsvektoren der Kontur eines jeden Zeichens ermittelt. Hierbei ist genau auf Fehlermeldungen zu achten, da manche Zeichen sonst später nicht oder nur falsch erkannt werden können [58]. Um das Training durchzuführen, muss Tesseract zusätzlich mit der Konfigurationsdatei „box.train" aufgerufen werden. Die ermittelten Merkmale sind nach der Ausführung in den Textdateien mit der Dateiendung „.tr" zu finden.

bis 9), Ziffer drei für Majuskeln und Ziffer vier für Minuskeln. Die fünfte Ziffer steht schließlich für Zeichen aus dem Alphabet [58]. Somit ergibt sich beispielsweise für das Zeichen T die Binärzahl 00101, da es sich um einen Großbuchstaben des Alphabets handelt. Dies entspricht in Hexadezimalschreibweise einer 5. Für das Fragezeichen ergibt sich die Binärzahl 10000, was einer hexadezimalen 10 entspricht (vgl. Abb. 4.9). Der Zeichentyp wird automatisch durch *unicharset_extractor* gesetzt, kann bei Bedarf jedoch angepasst werden.

Das Programm *shapeclustering* sollte derzeit nur bei indischen Schriften verwendet werden [58, 53]. Für die meisten anderen Sprachen, Fraktur eingeschlossen, können sich die Erkennungsergebnisse hierdurch deutlich verschlechtern.

Das Programm *mftraining* fügt die Merkmale jedes Glyphen anhand der Datei „unicharset“ durch ein Clustering zu Prototypen zusammen. Sich stark ähnelnde Glyphen werden durch einen Prototypen zusammengefasst. Sind die Unterschiede zwischen den Glyph-Varianten zu groß, können jedoch auch mehrere Prototypen für einen Unichar vergeben werden. Somit entsteht für jeden Unichar eine Klasse für die Klassifizierung während der Texterkennung, welche aus einem oder mehreren Prototypen besteht. Die so entstandene Datei „inttemp“ enthält die Informationen über die Prototypen für den k-nächste-Nachbarn-Klassifikator. Außerdem wird die Datei „pffmtable“ erstellt. In dieser Textdatei wird für jeden Unichar die Anzahl der Merkmale für den Class-Pruner festgehalten [53]. Hierdurch können bei der Klassifizierung die in Frage kommenden Klassen eingeschränkt werden, um so die Klassifikation zu beschleunigen.

Zum Schluss des Trainings wird über das Programm *cntraining* die Datei „normproto“ aus den TR-Dateien heraus erzeugt. Diese speichert Informationen über die Normalisierung der Prototypen [58] und die Position der Grundlinien im Prototyp, die ebenfalls für die Klassifikation wichtig sind.

Wörterbücher Tesseract verwendet innerhalb eines Sprachpakets bis zu sieben zusätzliche Dateien, die Informationen für das Sprachmodell enthalten [58]. Wichtigste Datei ist die allgemeine Wortliste. Hier sollten alle Wörter einer Sprache enthalten sein. Ist ein Wort in der Wortliste enthalten, ist es wahrscheinlicher, dass es bei der Texterkennung korrekt erkannt wird. Bei der Wortliste handelt es sich um eine reine Textdatei, die pro Zeile einen Begriff enthält. Daneben kann auch noch eine Datei mit sehr häufig auftretenden Wörtern einer Sprache erstellt werden. Diese sollte möglichst alle Stoppwörter einer Sprache enthalten und kann auch noch weitere häufig verwendete Wörter umfassen. Beispielsweise können auch Fachbegriffe in dieser Datei verwendet werden, wenn sich die Erkennung für bestimmte Fachbereiche eignen soll. Stehen bei der Klassifikation ein Wort aus der normalen Wortliste und der Wortliste mit häufigen Begriffen zur Wahl, wird in der Regel das Wort aus der Liste häufiger Begriffe gewählt. Durch die Anpassung der Konfigurationsvariablen „segment_penalty_dict_frequent_word“ und „segment_penalty_dict_nonword“ lässt sich die Auswahl der Begriffe aus den Wortlisten regulieren.

Neben den Wortlisten gibt es auch noch die Möglichkeit, Listen mit Mustern für Satzzeichen und Zahlen in jeweils eigenen Dateien zu definieren. In der Datei, die die Muster für Satzzeichen enthält, können häufig auftretende Kombinationen von Satzzeichen definiert werden, beispielsweise die Abfolge von Punkt und schließendem Anführungszeichen am Ende eines Zitats oder das Auslassungszeichen (…). Schließen Wörter direkt an die Satzzeichen an, wird dies durch ein einzelnes Leerzeichen signalisiert [58]. Ein beispielhaftes Muster für ein einzelnes, in Klammern gesetztes Wort sähe wie folgt aus: „( )“.

Die Datei mit Mustern für Zahlen ist für bestimmte Formatierungen von Zahlen und anderen Zeichen gedacht. Beispielsweise können Muster wie ISBN oder verschiedene Datumsformate in dieser Datei definiert werden, um Tesseract so die Erkennung besonderer Zahlen zu erleichtern. Jede Zeile der

```
2       , ,     1       „       1
2       r n     1       m       0
2       f z     1       ß       1
3       f c b   2       f ch    1
```

Abb. 4.10: Auszug aus einer „unicharambigs“-Datei

Datei „number-list“ enthält ein Muster, wobei jede mögliche Ziffer des Zeichens durch ein Leerzeichen ersetzt wird. Welche Zeichen Tesseract als mögliche Ziffern betrachtet, wurde zuvor in der Datei „unicharset“ festgelegt. Außerdem können die beiden Dateien „fixed-length-list“ und „unambig-list“ zum Einsatz kommen, sie eignen sich jedoch nur für die Erkennung der Sprachen Chinesisch, Japanisch und Koreanisch.

In der Datei „bigram-list“ können Kombinationen aus zwei Begriffen definiert werden, die häufig in Kombination auftreten. Kandidaten für diese Liste wären beispielsweise „et cetera“ oder „zum Beispiel“. Eventuelle Ziffern können durch Fragezeichen ersetzt werden, um so wiederum ein Muster zu erhalten [58]. Beispiele hierfür sind „??tes Jahrhundert“ oder „??er Jahre“.

Um Wörterbücher platzsparend speichern zu können, verwendet Tesseract die Datenstruktur *Trie*, welche platzsparend in Form eines *Directed Acyclic Word Graph (DAWG)* [9] gespeichert werden. Daher muss aus jeder der erstellten Wortlisten und Muster-Dateien eine entsprechende DAWG-Datei erstellt werden. Dies erfolgt über das Programm *wordlist2dawg*. Durch das Programm *dawg2wordlist* kann eine DAWG-Datei auch wieder in eine Wortliste zurücktransformiert werden, wobei die vormalige Sortierung verloren geht.

Ersetzungsregeln Die letzte Datei namens „unicharambigs“ definiert Ersetzungsregeln für Zeichen, die häufig falsch klassifiziert werden [41]. So kann Tesseract manuell auf leicht zu verwechselnde Zeichen aufmerksam gemacht werden, wenn diese falsch klassifiziert werden. Typische Beispiele sind die falsche Erkennung eines öffnenden Anführungszeichens als zwei Kommata oder des Zeichens m als rn, wenn der Zwischenraum zwischen den beiden Zeichen zu gering ist. Durch die ersten beiden Zeilen des in Abbildung 4.10 dargestellten Auszugs aus einer solchen „unicharambigs“-Datei können diese Erkennungsfehler behoben werden.

Die Spalten der Datei werden durch ein Tabulatorzeichen voneinander getrennt. Die erste Spalte gibt an, wie viele Unichars in der zweiten Spalte folgen. In der zweiten Spalte folgen die zu ersetzenden Unichars. Mehrere Unichars werden durch Leerzeichen getrennt. In der dritten Spalte folgt die Anzahl der Unichars, die in der vierten Spalte folgen. Die vierte Spalte enthält wie in Spalte zwei eine Reihe von Unichars. Es ist zudem wichtig, zwischen Zeichen und Unichars zu unterscheiden. Wurden zwei Zeichen, beispielsweise bei einer Ligatur, als ein Unichar trainiert, werden diese auch nur als ein Unichar gezählt und nicht durch ein Leerzeichen getrennt. Das ist auch in der vierten Zeile des Auszugs für die ch-Ligatur zu sehen. Die letzte Spalte gibt an, ob es sich um eine zwingende (`1`) oder eine optionale Ersetzung (`0`) handelt. Optionale Ersetzungen werden nur beachtet, wenn auch ein Wörterbuch vorhanden ist.

Sowohl sämtliche Dateien des Sprachmodells als auch die Datei „unicharambigs“ sind für die Texterkennung nicht zwingend erforderlich. Tesseract kann auch ohne die Dateien ausgeführt werden, die Erkennungsgenauigkeit ist ohne Sprachmodell aber in der Regel beeinträchtigt.

Zusammenfügen der Trainingsdateien Schließlich müssen die Dateien über das Programm *combine_tessdata* noch zu einer einzelnen „traineddata“-Datei zusammengefügt werden. Diese enthält sowohl die Informationen über alle erzeugten Klassen als auch Informationen zum Sprachmodell und damit alles, was zur Texterkennung notwendig ist. Hierzu müssen alle erzeugten Dateien mit dem Sprachcode als Präfix versehen werden.

Damit Tesseract das fertige Sprachpaket finden kann, muss die Datei im Verzeichnis „tessdata“ abgelegt werden. Dieses befindet sich unter Windows im Installationsverzeichnis von Tesseract. Außerdem verweist die Umgebungsvariable „%TESSDATA_PREFIX%“ auf das darüber liegende Verzeichnis. In den meisten Linux-Distributionen ist das Verzeichnis unter dem Pfad „/usr/local/shared/tessdata“ zu finden.

4.3 Evaluation bestehender Sprachpakete

Tesseract bringt eine Vielzahl von fertigen Trainingsdateien für verschiedene Sprachen mit. Dazu zählt auch die deutsche Sprachdatei „deu-frak.traineddata“, welche extra für die Erkennung von Frakturtexten trainiert wurde. Die Datei ist bei einer normalen Installation nicht enthalten, sondern kann separat heruntergeladen werden [41]. Auf den Webseiten des Projekts werden zudem einige weitere, unabhängige Trainingsprojekte für Frakturschriften gelistet [56]. In diesem Abschnitt werden die genannten Trainingspakete bezüglich ihrer Erkennungs- bzw. Fehlerraten verglichen. Dafür wird zunächst näher auf die Methodik und die für die Evaluation verwendeten Werkzeuge eingegangen. Die Vorstellung der Ergebnisse folgt in Abschnitt 4.3.2.

4.3.1 Methodik

Für die Evaluation steht eine Transkription des größten Teils von Band eins der Fries-Chronik zu Verfügung. Die Transkription wurde bereits in einem früheren Projekt maschinell erfasst und anschließend händisch von Mitarbeitern der Universitätsbibliothek Würzburg korrigiert. Größere Passagen des Texts fehlen jedoch in der Transkription, sodass zu 559 von insgesamt 795 Scans, die Text enthalten, eine Transkription vorliegt.

Für die Evaluation wurden die Transkriptionsteile, welche ursprünglich als mehrere Rich-Text-Dokumente vorlagen, über *Microsoft Word* als einfache Text-Dateien mit UTF-8-Kodierung abgespeichert. Anschließend wurden einige Fehler der Transkription verbessert, vermutlich sind aber immer noch einige Fehler in der Transkription enthalten. In einem weiteren Schritt wurde Seite für Seite der Transkription als einzelne Datei abgespeichert, um so leichter einzelne Seiten mit der Ausgabe aus

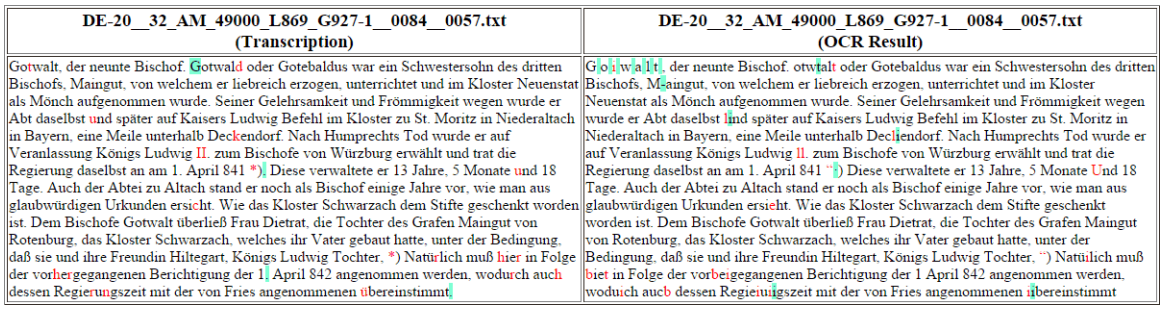
Tesseract vergleichen zu können. Außerdem wurde Tesseract für jeden Scan, zu dem auch eine Transkription vorlag, im einfachen Textmodus ausgeführt, sodass für jeden Scan zwei einfache Text-Dateien vorlagen, die miteinander verglichen werden konnten.

Für die Evaluation kamen entweder die händische Auswertung oder ein automatisierter Vergleich durch einen Distanz-Algorithmus in Frage. Da ein händischer Vergleich aller Scans als sehr zeitaufwändig eingeschätzt wurde, wurde diese Möglichkeit zugunsten eines automatisierten Ansatzes fallen gelassen. Für einen automatisierten Vergleich zweier Texte eignen sich Algorithmen für die Berechnung der Edit-Distanz zwischen zwei Zeichenketten oder Algorithmen für die Berechnung eines optimalen Alignments zweier Sequenzen. Durch erstere lässt sich der geringste Abstand zwischen zwei Zeichenketten bestimmen, um so beispielsweise fehlertolerante Suchen in Information-Retrieval-Systemen zu ermöglichen. Letztere werden häufig in der Bioinformatik für die Sequenzanalyse, beispielsweise zur Berechnung der Ähnlichkeit von DNA-Ketten, verwendet.

Eine häufig verwendete Variante der Edit-Distanz ist die sogenannte Levenshtein-Distanz [33], welche die drei Operationen „Einfügen“, „Ersetzen“ und „Löschen“ eines Zeichens kennt und diese gleich bewertet. Die Distanz repräsentiert die geringste Anzahl solcher Operationen die nötig sind, um eine Eingabe-Zeichenkette in die andere Eingabe-Zeichenkette zu transformieren. Damit bildet die Distanz ein sehr ähnliches Maß zu dem, was ein Mensch als Fehler bei der optischen Zeichenerkennung zählen würde. Mit der Damerau-Levenshtein-Distanz lässt sich die Definition der Levenshtein-Distanz so erweitern, dass auch das Vertauschen zweier Zeichen eine einzelne Operation darstellt. Während dies für die Erkennung von Tippfehlern sinnvoll erscheint, ist die Vertauschen-Operation bei der optischen Zeichenerkennung jedoch irreführend. OCR-Systemen ist bei der Erkennung die Reihenfolge der Zeichen durch ihre Lage in der Zeile fest vorgegeben. Stattdessen würden zwei Zeichen, die direkt aufeinander folgen und jeweils falsch erkannt wurden in bestimmten Situationen als einzelner Fehler gewertet, während ein Mensch hierfür zwei Fehler zählen würde.

Für das optimale Sequenz-Alignment werden in der Bioinformatik oft Algorithmen verwendet, die auf dem Needleman-Wunsch-Algorithmus [36] basieren. Dieser berechnet das Alignment ebenfalls wie bei der Levenshtein-Distanz als Folge von Editieroperationen. Der Ansatz hierbei ist jedoch die Ähnlichkeit zu maximieren anstatt den Abstand zu minimieren. Der Needleman-Wunsch-Algorithmus nimmt auch Einfüge- und Löschoperationen von mehreren, aufeinanderfolgenden Zeichen vor, während die Levenshtein-Distanz alle Operationen auf ein Zeichen beschränkt. Dadurch ist der Algorithmus in $O(\max(m,n)^3)$ für zwei Zeichenketten der Länge m bzw. n, während die meisten Implementierungen der Levenshtein-Distanz in $O(mn)$ sind. In der Praxis werden meist Varianten des Needleman-Wunsch-Algorithmus wie der Smith-Waterman-Algorithmus [59] verwendet, welche die Laufzeit durch Einschränkungen der Operationen verringern.

Der Speicherverbrauch des Needleman-Wunsch-Algorithmus und der meisten Implementierungen der Levenshtein-Distanz ist in $O(mn)$ und kann somit für lange Texte schnell zum Flaschenhals werden, da unter Umständen sogar mehrere Terabyte notwendig sein können, um mehrseitige Texte miteinander zu vergleichen. Mit dem Hirschberg-Algorithmus [27] existiert eine Variante des Needleman-Wunsch-



DE-20__32_AM_49000_L869_G927-1__0084__0057.txt (Transcription)	DE-20__32_AM_49000_L869_G927-1__0084__0057.txt (OCR Result)
Gotwalt, der neunte Bischof. Gotwald oder Gotebaldus war ein Schwestersohn des dritten Bischofs, Maingut, von welchem er liebreich erzogen, unterrichtet und im Kloster Neuenstat als Mönch aufgenommen wurde. Seiner Gelehrsamkeit und Frömmigkeit wegen wurde er Abt daselbst und später auf Kaisers Ludwig Befehl im Kloster zu St. Moritz in Niederaltach in Bayern, eine Meile unterhalb Deckendorf. Nach Humprechts Tod wurde er auf Veranlassung Königs Ludwig II. zum Bischofe von Würzburg erwählt und trat die Regierung daselbst an am 1. April 841 *)| Diese verwaltete er 13 Jahre, 5 Monate und 18 Tage. Auch der Abtei zu Altach stand er noch als Bischof einige Jahre vor, wie man aus glaubwürdigen Urkunden ersieht. Wie das Kloster Schwarzach dem Stifte geschenkt worden ist. Dem Bischofe Gotwalt überließ Frau Dietrat, die Tochter des Grafen Maingut von Rotenburg, das Kloster Schwarzach, welches ihr Vater gebaut hatte, unter der Bedingung, daß sie und ihre Freundin Hiltegart, Königs Ludwig Tochter. *) Natürlich muß hier in Folge der vorhergegangenen Berichtigung der 1| April 842 angenommen werden, wodurch auch dessen Regierungszeit mit der von Fries angenommenen übereinstimmt|	G|o|t|w|a|l|t, der neunte Bischof. otwtalt oder Gotebaldus war ein Schwestersohn des dritten Bischofs, M-aingut, von welchem er liebreich erzogen, unterrichtet und im Kloster Neuenstat als Mönch aufgenommen wurde. Seiner Gelehrsamkeit und Frömmigkeit wegen wurde er Abt daselbst |ind später auf Kaisers Ludwig Befehl im Kloster zu St. Moritz in Niederaltach in Bayern, eine Meile unterhalb Dec|endorf. Nach Humprechts Tod wurde er auf Veranlassung Königs Ludwig II. zum Bischofe von Würzburg erwählt und trat die Regierung daselbst an am 1. April 841 ``)| Diese verwaltete er 13 Jahre, 5 Monate Und 18 Tage. Auch der Abtei zu Altach stand er noch als Bischof einige Jahre vor, wie man aus glaubwürdigen Urkunden ersieht. Wie das Kloster Schwarzach dem Stifte geschenkt worden ist. Dem Bischofe Gotwalt überließ Frau Dietrat, die Tochter des Grafen Maingut von Rotenburg, das Kloster Schwarzach, welches ihr Vater gebaut hatte, unter der Bedingung, daß sie und ihre Freundin Hiltegart, Königs Ludwig Tochter, ``) Natüilich muß biet in Folge der vorbeigegangenen Berichtigung der 1 April 842 angenommen werden, wodurch aucb dessen Regieruligszeit mit der von Fries angenommenen iibereinstimmt

Abb. 4.11: Vergleichsansicht aus *ocrevalUAtion*

Algorithmus, der bei einer Laufzeit von $O(mn)$ nur einen Speicherverbrauch von $O(\min(m, n))$ hat. Damit liegt der Speicherverbrauch in der Größenordnung der kürzeren Zeichenkette und das Maß lässt sich auch für sehr lange Texte berechnen.

Die Auswertung erfolgte über die Software *ocrevalUAtion* [14], die im Rahmen des Impact-Projekts, eines durch die Europäische Union geförderten Digitalisierungsprojekts, und dessen Nachfolgeprojekt Succeed [64] entstand. Die Software ermöglicht die automatisierte Analyse zweier Textdateien durch die Berechnung eines Alignments zwischen den Dateien. Hierfür stehen Implementierungen für die Levenshtein-Distanz, die Damerau-Levenshtein-Distanz sowie eine Indel-Distanz zur Verfügung. Für die Evaluation wurde die standardmäßig eingestellte Levenshtein-Distanz gewählt.

Die Software erstellt einen Bericht, welcher die Zeichenfehlerrate, die Wortfehlerrate sowie eine Reihenfolgen-unabhängige Wortfehlerrate enthält. Zeichen- und Wortfehlerrate sind wichtige Kennzahlen für die Beurteilung der Erkennungsgenauigkeit eines OCR-Systems. Die Zeichenfehlerrate ist durch Gleichung 4.6 definiert. Gleichung 4.7 definiert die Wortfehlerrate. Dabei bezeichnen i, s und d bzw. i_w, s_w und d_w die Anzahl der Einfüge-, Ersetzungs- und Löschoperationen auf Zeichen- und Wortebene, n bezeichnet die Anzahl der Zeichen der Transkription, n_w die Anzahl der Wörter der Transkription. Durch den Umstand, dass ein einzelner Fehler innerhalb eines Worts einen Wortfehler verursacht, führt dazu, dass die Wortfehlerrate in der Regel höher ist als die Zeichenfehlerrate.

$$e = \frac{(i + s + d)}{n} \qquad e_\mathrm{w} = \frac{(i_\mathrm{w} + s_\mathrm{w} + d_\mathrm{w})}{n_\mathrm{w}} \tag{4.6, 4.7}$$

Die Erkennungsgenauigkeit (engl. *accuracy*), wie im Abschlussbericht des *Annual Test of OCR Accuracy* von Rice, Jenkins und Nartker [47] definiert, erhält man, indem man die Fehlerrate von 1 bzw. 100 Prozent abzieht. Neben den Werten für das gesamte Dokument ermittelt *ocrevalUAtion* auch noch die Gesamtzahl der Einfügungen, Ersetzungen und Löschungen für jedes Zeichen sowie die daraus resultierende Fehlerrate und stellt diese dar. So lassen sich schnell problematische Zeichen ausmachen. Für den Fall, dass die Anzahl der falschen Zuordnungen eines Zeichens im OCR-Text größer als die Anzahl der erwarteten Vorkommen des Zeichens ist, sind nach obiger Formel auch Fehlerraten größer als 100 Prozent und somit auch negative Accuracy-Werte möglich. Taucht ein Zeichen im OCR-Text auf, das nicht in der Transkription vorkommt, wird die Fehlerrate für das Zeichen als „Infinity“ angegeben.

Zudem stellt *ocrevalUAtion* den OCR-Text neben der Transkription dar und hebt die bei der Berechnung der Fehlerraten vorgenommenen Operationen farblich hervor, wie in Abbildung 4.11 zu sehen ist. Wird dabei die Maus über ein fehlendes Zeichen in einem Text bewegt, wird die entsprechende Stelle im anderen Text zusätzlich hervorgehoben. So wird eine schnelle Überprüfung der Fehlererkennung ermöglicht.

Bevor *ocrevalUAtion* zwei Texte miteinander vergleicht, führt es eine Normalisierung der beiden Eingabetexte durch. Dabei werden aufeinanderfolgende unsichtbare Zeichen wie Leerzeichen und Umbrüche durch ein einzelnes Leerzeichen ersetzt. Diese Ersetzung wird vorgenommen, um Ungenauigkeiten bei der Erstellung einer Transkription zu neutralisieren. Beispielsweise können sich beim Abschreiben eines Texts doppelte Leerzeichen einschleichen. Außerdem ist häufig nicht standardisiert, wie Einrückungen und sonstige Auszeichnungen durch Leerraum in einer Textdatei dargestellt werden. So kann beispielsweise ein Tabulatorzeichen für mehrere Leerzeichen stehen. Die Normalisierung des Weißraums führt dazu, dass lediglich Leerzeichen innerhalb eines Worts oder nicht erkannte Leerzeichen als Fehler gewertet werden.

Die Software bietet zusätzlich auch noch weitere Möglichkeiten zur Normalisierung, wie das Entfernen aller Satzzeichen, die Ersetzung von Umlauten durch Buchstaben des Grundalphabets oder das Ignorieren von Groß- und Kleinschreibung. Außerdem können in einer Konfigurationsdatei benutzerspezifische Ersetzungen von einem oder mehreren Zeichen definiert werden. Da diese Informationen aber für die Beurteilung der Erkennungsergebnisse wichtig sind, wurde hiervon zunächst kein Gebrauch gemacht.

Aufgrund einer Besonderheit der vorliegenden Transkription wurde eine Anpassung an *ocrevalUAtion* vorgenommen. Die Transkription ist nicht Zeilen-getreu erstellt worden und enthält somit auch keine der im Original enthaltenen Silbentrennungen am Zeilenende. Durch die Texterkennung von Tesseract werden Silbentrennungen jedoch in der Ausgabe belassen. Um diesem Umstand Rechnung zu tragen, wurde die Normalisierung so angepasst, dass ein Bindestrich am Zeilenende sowie der darauffolgende Zeilenumbruch ersatzlos entfernt werden. Diese Ersetzung verhindert, dass für jede Silbentrennung Fehler für den Bindestrich und das zusätzliche Leerzeichen gewertet werden, ist jedoch nicht immer korrekt. Erstreckt sich ein zusammengesetztes Wort über zwei Zeilen, so ist der Bindestrich auch in der Transkription enthalten. Liegt das zusammengesetzte Wort in der Transkription innerhalb einer Zeile, so führt die Ersetzung zu einem nicht erkannten Bindestrich. Aus praktischen Gründen wurde diese Fehlerquelle bei der Evaluation jedoch vernachlässigt. Weiter wurde die Tabelle mit den Fehlerraten für einzelne Zeichen um eine Spalte ergänzt, welche die Unicode-Bezeichnung des jeweiligen Zeichens enthält, sodass optisch schwierig voneinander unterscheidbare Zeichen wie beispielsweise die Ziffer 1, das kleine l und das große I, einfacher unterschieden werden können. In einer weiteren Spalte wurde die Anzahl der beibehaltenen Zeichen ergänzt, da diese zum Vergleich der Fehlerraten interessant sein kann.

4.3.2 Ergebnisse

„deu-frak“ Zunächst wurde das offizielle Sprachpaket für deutsche Frakturtexte „deu-frak“ [41] getestet. Leider stehen keine der für das Sprachpaket verwendeten Trainingsbeispiele zur Verfügung. Über das Programm *combine_tessdata* [42] konnte die Trainingsdatei jedoch in ihre Bestandteile zerlegt werden. Hierdurch konnten immerhin Aussagen über das verwendete Wörterbuch und die trainierten Zeichen getroffen werden. Es wurden insgesamt 135 Unichars trainiert. Dazu zählen neben dem Alphabet, Ziffern, Satz- und Sonderzeichen auch einige Ligaturen, die als eigene Unichars trainiert wurden.

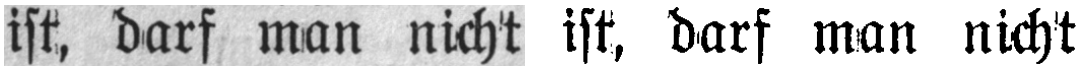
Unüblich für ein deutschsprachiges Unicharset sind hingegen zahlreiche Zeichen, die nur in skandinavischen Sprachen verbreitet sind, wie beispielsweise Ø oder å. Außerdem sind auch seltsame Kombinationen von Buchstaben enthalten, wie beispielsweise die Zeichenketten „MPO“ oder „rw“, die als eigene Unichars trainiert wurden. Das lange ſ wurde nicht als separates Unichar trainiert und somit steht in den OCR-Ergebnissen immer rundes s.

Im Sprachpaket ist ein Wörterbuch mit einer Liste häufig auftretender Wörter und einer Liste bekannter Wörter enthalten. Außerdem gibt es eine Liste mit möglichen Zahlen mit den Werten 0 bis 9, sowie eine Liste möglicher Satzzeichen. Die Listen der Ziffern und Satzzeichen entsprechen jedoch nicht den Empfehlungen der Trainingsanleitung [58]. Die Liste der bekannten Wörter umfasst etwas mehr als 40 000 Begriffe, während die Liste häufiger Wörter etwas weniger als 10 000 Begriffe enthält. Diese Verteilung ist jedoch unüblich. In der Liste häufiger Wörter des Sprachpakets „deu“ für deutsche Antiquatexte sind beispielsweise nur 99 der häufigsten Begriffe enthalten. Die gesamte Wortliste enthält aber etwas mehr als 100 000 Begriffe. Auch veraltete Schreibweisen wie beispielsweise „seyn“ statt „sein“ sind in den Wörterbüchern von „deu-frak“ enthalten. Allerdings wird auch hier nicht zwischen langem ſ und rundem s unterschieden. Es steht immer rundes s.

Das Paket erreicht auf den Testdaten eine Zeichenfehlerrate von durchschnittlich 2,50 Prozent sowie eine Wortfehlerrate von durchschnittlich 8,18 Prozent auf allen Scans. Die Fehlerraten der einzelnen Zeichen sind im Anhang in Tabelle A.1 aufgeführt. Nachfolgend soll nur auf einige interessante Werte eingegangen werden.

Häufige Fehler sind die Verwechslung von J mit I, M mit R, F mit I, B mit V, C mit E sowie ſ mit f. Außerdem wird die Majuskel I in römischen Zahlen, welche durchgängig in Antiqua gesetzt sind (bspw. in „Johann II.“), häufig mit dem kleinen l verwechselt. Daher ist die Fehlerrate von I/J mit 88,69 Prozent wesentlich höher als die von J mit nur 10,40 Prozent. Die Transkription zählt 2 069 Exemplare von I/J und 1 933 Exemplare von J. Mit einer Erkennungsgenauigkeit von je 99,72 Prozent werden die Buchstaben b und e am sichersten erkannt. Beim kleinen e werden zahlreiche Exemplare als kleines c klassifiziert. Trotzdem ist die Erkennungsgenauigkeit sehr hoch, da e nach dem Leerzeichen häufigstes auftretendes Zeichen ist.

Auffällig ist, dass bestimmte Zeichen durchgängig falsch erkannt werden. So werden anstatt der Anführungszeichen „ und “ immer die französischen Anführungszeichen » und « erkannt. Auch statt dem Sternchen (*) wird meist « erkannt. Von insgesamt 630 Exemplaren in der Transkription wurde lediglich ein einzelnes Sternchen korrekt klassifiziert.



(a) (b)

Abb. 4.12: Schmutz vor (a) und nach der Binarisierung durch den Algorithmus von Otsu (b)

Auch im Erkennungsergebnis sind teilweise skandinavische Schriftzeichen enthalten. Manche Sonderzeichen wie beispielsweise das Kreuzsymbol (†) waren im Unicharset nicht enthalten und konnten dadurch auch nicht erkannt werden.

Eine weitere Auffälligkeit ergibt sich dadurch, dass Punkte häufig verloren gehen. Von insgesamt 11 691 Punkten wurden 1 805 Punkte ignoriert, was zu der hohen Fehlerquote von 20,83 Prozent beiträgt. Das Komma geht hingegen viel seltener verloren und hat nur eine Fehlerrate von 4,02 Prozent.

Sehr viele Erkennungsfehler entstehen auch durch Verschmutzungen vor oder nach Buchstaben. Diese sind wahrscheinlich durch zu hohen Anpressdruck beim Druckvorgang entstanden, der bei einigen der Bleilettern dazu geführt hat, dass zusätzliche Druckerschwärze auf das Papier gelangt ist. Vor allem vor den Buchstaben a, e, g und um t sind kleine Flecken oder kurze vertikale Linien zu finden, die weder durch die Binarisierung nach Otsu noch durch die Binarisierung nach Sauvola beseitigt werden können. Abbildung 4.12 zeigt eine kurze Textstelle, bei der es zu einigen Verschmutzungen vor a und um t gekommen ist. Die Erkennung liefert für diesen Textausschnitt „iſt!, darf mkan nicht“. Es gibt also Schmutzpartikel, die ignoriert werden, aber auch häufig Zeichen, die zwischen korrekt erkannten Zeichen eingefügt werden und so das Erkennungsergebnis verschlechtern. Meist werden Schmutzpartikel als Bindestriche, Punkte, Apostrophe oder rundes s klassifiziert, seltener werden auch f, l, i, « oder andere Satzzeichen erkannt. Verschmutzungen bilden nach Verwechslungen ähnlicher Zeichen die zweithäufigste Fehlerquelle.

„tesseract-dan-fraktur“ Als nächstes wurde das Paket „tesseract-dan-fraktur“ [7] getestet, welches Trainingsdateien für Frakturtexte in Dänisch, Deutsch und Schwedisch enthält. Ursprünglich wurde das Paket für dänische Frakturtexte konzipiert und anschließend mit einem deutschen und einem schwedischen Wörterbuch versehen.

Anders als bei „deu-frak“ liegen die Trainingsbeispiele für das Sprachpaket vor, sodass diese ebenfalls untersucht werden konnten. Für die Sprachdatei wurden 31 Trainingsbeispiele mit zahlreichen unterschiedlichen Frakturschriftarten trainiert. Drei der Trainingsbeispiele enthalten Texte, die in einer Variante der Schwabacher gesetzt sind. Zudem enthalten zwei Trainingsbeispiele auch Auszeichnungen in einer gotischen Schriftart. Sehr häufig ist mehr als ein Schriftstil innerhalb eines Traininsbeispiels enthalten, was den Empfehlungen der Trainingsanleitung widerspricht (vgl. Smith u. a. [58]).

Durch die Analyse der enthaltenen Box-Dateien konnten einige fehlerhafte Auszeichnungen gefunden werden. So sind ab und zu Groß- und Kleinschreibung verwechselt worden. Manchmal sind auch optisch unähnliche Glyphen falsch ausgezeichnet worden. In seltenen Fällen enthalten die Boxen nur einen

Teil des Zeichens. Dies kommt beispielsweise beim kleinen ꝵ vor, wo in wenigen Fällen nur die untere Hälfte des Buchstaben enthalten ist.

Die Zeichenfehlerrate des Pakets ist mit durchschnittlich 2,44 Prozent leicht besser als die des offiziellen Pakets für deutsche Frakturtexte. Die Wortfehlerrate ist mit 8,06 Prozent ebenfalls leicht besser.

„frk“ Außerdem wurde das Sprachpaket „frk“ [46] getestet, welches ebenfalls wie „deu-frak“ zu den offiziellen Sprachpaketen von Tesseract gehört und auf den Download-Seiten des Projekts mit „Frankish language data for Tesseract 3.02“ bezeichnet wird. Da der Verwendungszweck des Pakets unklar war, wurde es ebenfalls analysiert.

Das enthaltene Wörterbuch umfasst insgesamt knapp 100 000 Begriffe. Es handelt sich dabei aber ausschließlich um modernes Deutsch. Mittelalterliche Begriffe oder veraltete Schreibweisen sind nicht enthalten. Dafür ist im Gegensatz zu den beiden bisher getesteten Sprachpaketen auch eine Liste mit Wort-Bigrammen enthalten, die 722 420 Wortpaare umfasst. Der enthaltene Zeichensatz besteht aus Zeichen, die in deutschen Texten üblich sind.

Das Sprachpaket bleibt mit einer Zeichenfehlerrate von 6,27 Prozent und einer Wortfehlerrate von 29,50 Prozent deutlich hinter den Erkennungsraten der beiden anderen Sprachpakete zurück. Vor allem die Zahl der Einfügungen durch Schmutz ist wesentlich höher als bei den anderen Sprachpaketen. Zudem wird kein einziges langes ſ korrekt erkannt. Hierdurch liegt die Fehlerrate von ſ/s bei 37,04 Prozent. Auch die Fehlerraten der anderen Buchstaben liegen meist um wenige Prozentpunkte höher als die der anderen Sprachpakete. Da sich das Sprachpaket offenbar nicht für Frakturtexte eignet, wurde es von den weiteren Betrachtungen ausgeschlossen.

Erkennung ohne Wörterbuch Um den Einfluss des Wörterbuchs auf die Erkennungsgenauigkeit abschätzen zu können, wurden Varianten der Sprachpakete „deu-frak“ sowie von „tesseract-dan-fraktur“ ohne Wörterbücher erstellt. Dafür wurden die Sprachdateien mit dem Programm *combine_tessdata* in ihre Bestandteile zerlegt. Anschließend wurden alle Dateien, die Informationen über die Sprache enthalten, sowie die Datei „unicharambigs“ entfernt und die verbleibenden Dateien anschließend wieder mit *combine_tessdata* zusammengefügt.

Ohne Wörterbuch stieg die Zeichenfehlerrate des Pakets „deu-frak“ von 2,50 Prozent auf 5,65 Prozent an. Vor allem die Unterscheidung von ſ und langem ſ wurde erheblich erschwert. Die Fehlerrate von ſ/s stieg von zuvor 3,66 Prozent beinahe auf das Zehnfache, nämlich 31,33 Prozent. Auch die Anzahl der Fehler durch die Verwechslung von c und e stieg stark an. Interessanterweise stieg die Fehlerrate der schließenden Klammer sehr viel stärker als die der öffnenden Klammer. Häufig wurde ſ) anstatt ꝵ erkannt, da das kleine ꝵ in der Fraktur wie die Kombination aus einem kleinen ſ und einer nach unten versetzten, schließenden Klammer aussieht. Auch allgemein gesehen wurden viele Verwechslungen gemacht, die so vorher nicht aufgetreten waren.

Beim Paket „tesseract-dan-fraktur“ stieg die Zeichenfehlerrate ohne Wörterbuch sogar noch stärker auf 6,14 Prozent an. Die Fehlerrate von ſ/ꞩ stieg jedoch nicht ganz so stark von 3,64 Prozent auf 27,77 Prozent. Die Verwechslung von ꝛ und ſ) war ebenfalls zu verzeichnen und mit einer Fehlerrate von 4,97 Prozent deutlich höher als die von „deu-frak“ mit 3,25 Prozent. Die Effekte durch das Weglassen des Wörterbuchs ähneln sich auch bei den weiteren Zeichen. Zusätzlich haben beide Sprachpakete gemeinsam, dass Verschmutzungen sehr viel häufiger als eigenständige Zeichen klassifiziert werden.

4.4 Cube-Modus

Neben den vorgestellten Komponenten zur Texterkennung und zum Training existiert noch ein alternativer Modus von Tesseract namens Cube. Bei diesem kommt statt des erwähnten Nächste-Nachbarn-Klassifikators ein künstliches neuronales Netzwerk zum Einsatz [53]. Daher sind für diesen Modus andere Klassen und daraus folgend auch andere Methoden und Werkzeuge zum Training notwendig. Cube wurde erstmals mit Version 3.01 des Projekts für Arabisch vorgestellt. Leider ist der Erkennungsmodus seither nur sehr schlecht dokumentiert worden. In der Zwischenzeit wurden einige weitere Sprachen mit den notwendigen Trainingsdateien ausgestattet, es existieren jedoch keine öffentlich zugänglichen Werkzeuge für das Training neuer Sprachpakete für diesen Modus. Smith [53] beschreibt die Verbesserungen der Texterkennung, die durch Cube erzielt wurden, als „enttäuschend“. Daher wird der Cube-Modus derzeit auch nicht mehr aktiv weiterentwickelt. Außerdem existiert leider kein Sprachpaket für deutsche Frakturtexte.

Um trotzdem einen Eindruck darüber gewinnen zu können, ob durch Cube Verbesserungen für die Erkennung von Frakturtexten möglich sind, wurde ein Experiment mit dem englischen Sprachpaket durchgeführt. Hierfür wurde ein Teil des Anfangskapitels aus William Shakespeares „Hamlet“ in einer gotischen Schriftart gesetzt, da diese im Mittelalter auch im englischen Sprachraum gebräuchlich waren. Anschließend wurde die Texterkennung mit aktiviertem Cube-Modus durchgeführt. Im erzielten Ergebnis wurde jedoch kaum ein Zeichen korrekt erkannt, sodass die Zeichenfehlerrate bei 56,22 Prozent lag.

Um die Ergebnisse vergleichen zu können wurden auch noch zwei Testläufe mit den Paketen „eng“ ohne Cube-Modus und „deu-frak“ durchgeführt. Hierbei erzielte „deu-frak“ noch die besten Ergebnisse mit einer Zeichenfehlerrate von 10,27 Prozent. Die meisten Minuskeln wurden korrekt erkannt, jedoch unterscheiden sich die Majuskeln der gotischen Schrift stark von denen typischer Frakturschriften, sodass diese häufig falsch klassifiziert wurden. „eng“ erreichte eine Zeichenfehlerrate von 35,82 Prozent. Die Ergebnisse zeigen deutlich, dass die englische Sprachdatei weder im Cube-Modus, noch im normalen Modus für gebrochene Schriften trainiert wurde. Leider konnten durch das Experiment keine sinnvollen Aussagen zu den Möglichkeiten des Cube-Modus für Frakturtexte getroffen werden.

Zum Vergleich wurde das gleiche Textbeispiel noch in der Schriftart „Times New Roman“ gesetzt. Hierfür erreichte das englische Sprachpaket mit aktiviertem Cube-Modus eine Zeichenfehlerrate von 0,36 Prozent. Ohne Cube-Modus waren es nur 0,94 Prozent. Es gab also eine geringe Verbesserung

des Ergebnisses durch den Cube-Modus. Dadurch kann darauf geschlossen werden, dass sich der Cube-Modus potenziell für eine Verbesserung bei der Erkennung von Frakturschriften eignet. Durch die genannten Einschränkungen ist eine Verwendung jedoch leider nicht möglich.

5 Verbesserung der Erkennungsergebnisse

In diesem Kapitel werden Versuche und eigene Trainingsvorgänge beschrieben, die zur Verbesserung der Erkennungsergebnisse im Rahmen dieser Arbeit unternommen wurden. Die allgemeine Vorgehensweise für das Training neuer Schriftarten und Sprachen wurde bereits in Abschnitt 4.2 besprochen.

Um das Training eines neuen Sprachpakets zielführend durchführen zu können, wurden für das Sprachpaket „deu-frak“ zunächst noch Experimente mit einzelnen, schwierig zu unterscheidenden Zeichen durchgeführt. Durch gezieltes Training der Zeichen wurde anschließend überprüft, ob sich die ermittelten Schwächen beheben lassen.

In späteren Experimenten wurden Trainings mit Trainingsbeispielen für das gesamte Alphabet durchgeführt. Auch die Erkennung durch das Paket „tesseract-dan-fraktur“ wurde durch Anpassungen verbessert. Die besten Ergebnisse wurden zu einem Gesamtpaket zusammengefügt, welches anschließend auch mit einem Wörterbuch versehen wurde. So konnten die Erkennungsraten von Tesseract für die Fries-Chronik erheblich gesteigert werden.

5.1 Experimente mit einzelnen Zeichen

Unterscheidung von ſ und langem ſ Bei der Evaluation der Sprachpakete „deu-frak“ und „tesseract-dan-fraktur“ wurde festgestellt, dass die Zeichen ſ und langes ſ in vielen Fällen nicht korrekt klassifiziert werden und es immer wieder zu Verwechslungen kommt. Nachdem das Wörterbuch weggelassen wurde, verschlechterten sich die Erkennungsraten noch weiter. Diese Schwierigkeiten bei der Unterscheidung der beiden Buchstaben führte zur Aufstellung der Hypothese, dass der Querstrich, durch den sich ſ und ſ unterscheiden, als Merkmal für eine Unterscheidung der Zeichen nicht ausreicht.

Um die Unterscheidung von ſ und ſ gezielt überprüfen zu können, wurde ein kurzes Testbeispiel erstellt, das ausschließlich die Buchstaben fünf ſ und sieben ſ enthält, die aus den Scans der Fries-Chronik gesammelt wurden (vgl. Abbildung 5.1). Die Buchstaben hatten jeweils eine Höhe von 56 Pixeln, was den Scans der Fries-Chronik bei einer Auflösung von 300 DPI entspricht. Anschließend wurde die Texterkennung auf diesem Beispiel ausgeführt.

Die Erkennung mit „deu-frak“ war wie zu erwarten unbefriedigend. Lediglich die letzten beiden langen ſ wurden als solche erkannt. Alle weiteren Zeichen wurden als ſ klassifiziert. Die Erkennung durch „tesseract-dan-fraktur“ war hingegen deutlich besser. Bis auf ein einzelnes langes ſ wurden diese korrekt erkannt.

Um den Schwierigkeitsgrad weiter zu erhöhen, wurde die Erkennung auf kleiner skalierten Versionen des Beispiels durchgeführt. Bei einer Auflösung von 150 DPI wurden von „deu-frak“ sechs der sieben langen ſ auch als solche klassifiziert. Allerdings wurde ein langes ſ als kleines i erkannt. Ein ſ wurde zu einem ſ. Durch die Erstellung einer Box-Datei konnte die Ursache für die bessere Erkennung bei dieser geringen Auflösung ausgemacht werden. Durch die geringere Auflösung wurden einige aufeinanderfolgende lange ſ als Doppel-ſ-Ligatur klassiziert. Doppel-ſ-Ligaturen wurden in „deu-frak“ nicht trainiert. Bei 300 DPI war der Abstand jedoch so groß, dass keine Ligaturen erkannt wurden, sodass die Entscheidung auf ſ fiel. „deu-dan-fraktur“ dagegen erkannte bei 150 DPI bis auf drei ſ alle Zeichen als ſ.

Abbildung 5.2 zeigt die durch Tesseract ermittelten Umrisse und die dazugehörigen approximierten Polygone bei 300 DPI (links) und 150 DPI (rechts). Bei beiden Auflösungen ist der Querstrich, der das lange ſ vom ſ unterscheidet deutlich zu erkennen. Trotzdem können beide Sprachpakete beide Zeichen nicht zuverlässig unterscheiden. Aus diesem Grund wurde ein Vergleichstraining zu einem Trainingsbeispiel durchgeführt, das nur ſ und lange ſ enthält. Das Trainingsbeispiel wurde mit Schriftart Normalfraktur erstellt.

Nach dem Training wurden alle Buchstaben korrekt unterschieden. Auch bei 150 DPI und sogar bei 100 DPI, wo die einzelnen Zeichen nur noch 19 Pixel hoch und mit bloßem Auge kaum noch zu unterscheiden waren, wurden alle Buchstaben korrekt erkannt. Zusätzlich wurde ein Beispiel mit verschiedenen Fontgrößen und vertikal versetzten Buchstaben erstellt. Auch hier stellt die Unterscheidung von ſ und ſ kein Problem dar. Die anfänglich aufgestellte Hypothese konnte damit eindeutig widerlegt werden. Eine Möglichkeit, weshalb „deu-frak“ und „tesseract-dan-fraktur“ so schlecht bei der Unterscheidung von ſ und ſ abgeschnitten haben, wäre, dass beim Erstellen der Box-Dateien zu diesen Paketen eine falsche Zuordnung der Zeichen vorgenommen wurde, sodass sich die erzeugten Klassifikatoren zu wenig unterschieden.

Unterscheidung von 𝔅 und 𝔙 In einem zweiten Experiment erfolgte die Evaluation für 𝔅 (B) und 𝔙 (V) analog zum vorigen Experiment. Aus Beispielen der Scans der Fries-Chronik wurde ein Evaluationsbeispiel erstellt, das aus drei 𝔅 und drei 𝔙 bestand. Anschließend wurden die Texterkennungsergebnisse mit „deu-frak“ und „tesseract-dan-fraktur“ betrachtet.

Mit „deu-frak“ wurden alle Glyphen als 𝔅 klassifiziert, während mit „tesseract-dan-fraktur“ alle drei 𝔅 als 𝔙 klassifiziert sowie zwei der 𝔙-Glyphen als 𝔅 klassifiziert wurden. Durch das Training eines kurzen Beispiels in Normalfraktur konnte diese Erkennungsschwierigkeit jedoch nicht behoben

Abb. 5.1: Evaluationsbeispiel mit ſ und ſ

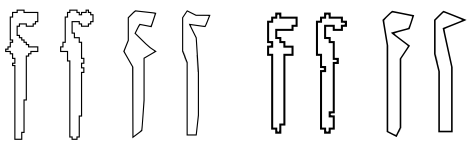


Abb. 5.2: Durch Tesseract ermittelte Umrisse und Polygone von ſ und ſ bei 300 und 150 DPI

werden. Alle Buchstaben wurden nach dem Training wie bei „deu-frak“ als 𝔅 klassifiziert. Ursache der Fehler war, dass sich die Glyphen der Großbuchstaben in der Fries-Chronik deutlich von denen der Normalfraktur unterscheiden. Deshalb wurde ein weiteres Trainingsbeispiel mit Glyphen aus der Fries-Chronik erstellt und trainiert. Durch das Ergebnis dieses Trainings konnten alle Glyphen korrekt klassifiziert werden.

5.2 Training ohne Wörterbuch

Nachdem die Experimente für einzelne Zeichen abgeschlossen waren, wurde damit begonnen, Trainings für das gesamte Alphabet und die wichtigsten Satz- und Sonderzeichen durchzuführen. Durch die Evaluation der bestehenden Sprachpakete mit und ohne Wörterbuch wurde gezeigt, dass durch den Einsatz eines Wörterbuchs viele potenzielle Fehler korrigiert werden können. Um jedoch die Erkennungsergebnisse nicht durch das Wörterbuch zu beeinflussen wurde es für die folgenden Experimente zunächst weggelassen.

Training mit Scans der Fries-Chronik Für die ersten Trainingsversuche wurde auf Scans der Fries-Chronik zurückgegriffen. Anhand von vier Scans wurde die Schriftart des Haupt-Textteils trainiert. Überschriften und Fußnoten sind in einer anderen Fraktur-Schriftart gesetzt und wurden daher durch ein Bildbearbeitungsprogramm entfernt. Auch Wörter, die in Antiqua gesetzt sind, wurden entfernt, um den Empfehlungen der Trainingsanleitung [58] zu entsprechen. Der Schmutz wurde jedoch vor dem Training nicht entfernt. Buchstaben wie e oder g, vor denen sich häufig Schmutz findet, wurden so trainiert, dass die Boxen den Schmutz enthalten.

Da nur eine einzelne Schriftart trainiert wurde und somit das Ergebnis der Evaluation auf dem Transkriptionstext nur wenig Aussagekraft gehabt hätte, wurde das so entstandene Sprachpaket auf denselben Scans evaluiert, auf denen es auch trainiert wurde. Die Zeichenfehlerrate lag bei 5,26 Prozent, die Wortfehlerrate bei 30,93 Prozent. Alle langen ſ wurden als ſ klassifiziert. Viele runde ſ wurden mit g und seltener auch mit ȝ (Z) oder ö verwechselt. Dies führte zu einer Fehlerrate der Klasse ſ/ſ von 87,41 Prozent. Hinzu kamen die häufige Verwechslung von ℑ (I) und ℑ (J) und Fehler durch Schmutz der als eigenes Zeichen klassifiziert wurde. Durch mehrmaliges Überprüfen der Box-Dateien wurde sichergestellt, dass darin keine Fehler enthalten waren.

Da die erzielten Erkennungsergebnisse nicht zufriedenstellend waren, wurde anschließend versucht, die Ergebnisse durch Änderungen an den Box-Dateien zu verbessern. Dafür wurde das Zeichen s in allen Boxen, die ein langes ſ enthielten, durch das eigens dafür vorgesehene Unicode-Zeichen („Latin small letter long s“; U+017F) ersetzt. Durch diese Änderung konnte die Zeichenfehlerrate auf 1,87 Prozent und die Wortfehlerrate auf 11,92 Prozent verringert werden. Die Fehlerrate von ſ/ſ sank auf 1,08 Prozent. Kein einziges langes ſ wurde mehr mit einem ſ verwechselt. Lediglich dreimal wurde Schmutz als ſ klassifiziert. Die weiteren Fehler blieben weitestgehend unverändert. Größte Fehlerquelle waren nun die Verwechslung von ℑ und ℑ mit Fehlerraten von 14,29 Prozent für ℑ und 52,17 Prozent für ℑ sowie

die Klassifizierung von Schmutz als eigene Zeichen. Dazu muss gesagt werden, dass ℑ und ℑ sich in der Frakturschrift der Fries-Chronik nicht voneinander unterscheiden. Zählt man die Falsch-Klassifikation von ℑ und ℑ nicht als Fehler, sinkt die Zeichenfehlerrate insgesamt auf 1,67 Prozent.

Da die Ergebnisse des Trainings vielversprechend waren, wurde das Training um weitere Trainingsbeispiele erweitert, sodass es auch auf der gesamten Transkription evaluiert werden konnte. Durch ein Bildbearbeitungsprogramm wurden zwei weitere Trainingsbeispiele aus Überschriften mehrerer Scans zusammengesetzt. Hierbei konnte jedoch nicht der gesamte Zeichensatz abgedeckt werden, da nicht jedes Zeichen in einer Überschrift vorkommt. Außerdem wurde ein weiteres Trainingsbeispiel für die in Fußnoten verwendete Schriftart erstellt.

Die Evaluation ergab eine Zeichenfehlerrate von 3,12 Prozent sowie eine Wortfehlerrate von 13,91 Prozent. Größtes Problem der Erkennung war jedoch wieder die falsche Klassifikation von Schmutz, was zu hohen Fehlerraten bei Satzzeichen führte. Die Texterkennung hatte auch Schwierigkeiten bei der Unterscheidung von ℑ und ℑ sowie ℜ und ℜ. Die Erkennung der Klasse ſ/s war mit einer Fehlerrate von 2,85 Prozent eher durchschnittlich. Eine Verwechslung von ſ und ſ trat selten auf. Buchstaben die selten waren hatten auch meist eine höhere Fehlerquote.

Training mit künstlich erzeugten Beispielen Nach diesen Experimenten wurden auch erstmals künstliche Trainingsbeispiele für das gesamte Alphabet und die wichtigsten Satz- und Sonderzeichen erstellt. Dafür wurden zahlreiche Fraktur-TrueType-Schriftarten gesammelt. Zunächst wurde ein einzelnes Trainingsbeispiel erstellt, das mit der Schriftart neue Fraktur [24] gesetzt wurde, da diese der Schriftart des größten Teils der Fries-Chronik von den getesteten Schriftarten am nächsten kommt. Der enthaltene Text besteht aus einer Auflistung des Alphabets und einiger Sonderzeichen, wobei Groß- und Kleinbuchstaben abwechselnd verwendet werden, um so sicherzustellen, dass Tesseract die Zeilen- und x-Höhe bestimmen kann. Außerdem wurden mehrere Pangramme verwendet, das heißt Sätze, die alle Buchstaben des Alphabets enthalten (z. B. „Falſches Üben von Xylophonmuſik quält jeden größeren Zwerg.“). Dazwischen wurden weitere Satz- und Sonderzeichen eingestreut und in einer Auflistung weiterer Wörter wurde dafür gesorgt, dass auch alle Großbuchstaben mehrfach vorkommen. Die Kantenglättung des Bildbearbeitungsprogramms wurde beim Erstellen des Trainingsbeispiels deaktiviert, sodass direkt ein Binärbild entstand. Um die Probleme aus dem vorigen Training zu vermeiden, wurde beim Erstellen der Box-Dateien darauf geachtet, dass alle langen ſ mit dem entsprechenden Unicode-Zeichen markiert wurden.

Nach dem Training wurde das entstandene Sprachpaket auf den vier Scans des vorigen Trainings evaluiert. Dabei kam es zu einer Zeichenfehlerrate von 4,01 Prozent sowie einer Wortfehlerrate von 25,13 Prozent. Die Fehlerquellen waren weitestgehend ähnlich zu den vorausgegangenen Experimenten. Die meisten Fehler entstanden durch die falsche Klassifikation von Schmutz, einige weitere durch die Verwechslung von ℑ durch ℑ, ℜ durch ℜ und e durch c. Im Durchschnitt waren die Fehlerraten der einzelnen Zeichen jedoch etwas höher als im vorigen Experiment. Es wurde jedoch nur ein ſ als ſ klassifiziert, was zu einer Fehlerrate der Klasse ſ/s von 0,72 Prozent führte.

Auch dieses Sprachpaket wurde anschließend um weitere Schriftarten erweitert. So wurden zusätzliche Trainingsbeispiele für die Schriftart neue Fraktur und ein Beispiel für die Schriftart Normalfraktur erstellt. Trainingsbeispiele anderer Frakturschriftarten erwiesen sich jedoch als kontraproduktiv. Auch Trainingsbeispiele für eine Antiqua-Schriftart wurden hinzugefügt.

Die Evaluation erfolgte auf der gesamten Transkription. Die Texterkennung erreichte eine Zeichenfehlerrate von 3,70 Prozent und eine Wortfehlerrate von 18,02 Prozent und war damit leicht schlechter als die Erkennung des mit Scans trainierten Sprachpakets. Die Erkennungsergebnisse waren jedoch sehr ähnlich zueinander. Größtes Problem war wieder die falsche Klassifikation von Schmutz. Die Erkennung von in Antiqua gesetzten Wörtern war jedoch deutlich besser. Probleme bei der Erkennung von Antiqua waren vor allem die Unterscheidung von Groß- und Kleinschreibung mancher Zeichen sowie dass das große I in römischen Zahlen meist als kleines l erkannt wurde.

Verbesserungen am Paket „tesseract-dan-fraktur“ Als Alternative zur Erstellung eigener Trainingsbeispiele wurde versucht, das Sprachpaket „deu-dan-fraktur“ so anzupassen, dass es zu besseren Ergebnissen bei der Erkennung der Fries-Chronik kommt. In einem ersten Verbesserungsversuch wurden die Trainingsbeispiele des Pakets durch ein Bildbearbeitungsprogramm von skandinavischen Buchstaben bereinigt. Trainingsbeispiele, die keine Frakturschrift oder von der Fries-Chronik stark abweichende Schriften enthielten wurden entfernt. Außerdem wurden die Box-Dateien der einzelnen Trainingsbeispiele auf Fehler untersucht und diese anschließend behoben. Boxen für langes ſ wurden mit dem korrekten Unicode-Zeichen ersetzt, um so die Unterscheidung von ſ und ſ zu verbessern. Das enthaltene Wörterbuch wurde entfernt, um die Ergebnisse nicht zu beeinflussen.

Nach dem Training wurde das Sprachpaket auf dem gesamten transkribierten Teil der Chronik evaluiert. Es konnte jedoch nur ein leichter Rückgang der Zeichenfehlerrate von vormals 6,14 Prozent auf 5,09 Prozent verzeichnet werden. Die Fehlerrate der Klasse ſ/ſ lag vor der Anpassung bei 27,77 Prozent. Nach der Änderung waren es nur noch 11,92 Prozent, was eine deutliche Verbesserung darstellt. Leider konnte die Verbesserungrate der vorigen Experimente jedoch nicht erreicht werden. Die Fehlerraten für einige Großbuchstaben, darunter 𝔉, 𝔐 und 𝔚, stiegen durch die Anpassung deutlich an. Die Fehlerrate von 𝔑 hingegen sank deutlich.

Auch die Fehlerraten der meisten Kleinbuchstaben stiegen leicht an. Die Fehlerrate von ꝛ stieg sogar von 186,42 Prozent auf 402,47 Prozent. Dagegen sank beispielsweise die Fehlerrate von ꝙ von 97,22 Prozent auf 34,72 Prozent. Die Zeichenfehlerrate sank insgesamt trotzdem leicht, da die Fehler durch skandinavische Zeichen verschwanden. Außerdem verringerte sich die Fehleranzahl durch Satz- und Sonderzeichen deutlich. Auch durch eine weitere Beschränkung der Trainingsbeispiele konnten die Erkennungsergebnisse nur unwesentlich verbessert werden.

Zusammenführung der Ergebnisse Aus den ersten Erfahrungen durch das Training wurden die gesammelten Trainingsbeispiele anschließend zu einem Gesamtpaket zusammengeführt. Da die Erkennungsraten der verbesserten Version von „tesseract-dan-fraktur“ im Vergleich zu den anderen Paketen

schlechter waren, wurde dieses Paket jedoch nicht in das Gesamtpaket aufgenommen. Die resultierende Trainingsmenge bestand folglich aus den Trainingsbeispielen der Scans mit vier Schriftarten sowie aus den Trainingsbeispielen mit drei künstlich erstellten Schriftarten.

Die Evaluation auf der Transkription ergab eine Zeichenfehlerrate von 3,25 Prozent und eine Wortfehlerrate von 15,57 Prozent. Die Erkennungsraten für dieses Sprachpaket lagen somit zwischen den Erkennungsraten der Sprachpakete, die durch Scans und künstlich erzeugte Trainingsbeispiele erzeugt wurden.

5.3 Beseitigung der durch Schmutz verursachten Erkennungsfehler

Während der Trainingsversuche war immer wieder Schmutz zwischen Zeichen die Ursache vieler Erkennungsfehler. Vor allem Schmutz, der nicht mit Buchstaben verbunden ist, wird häufig als Zeichen klassifiziert. Meist reichen schon drei oder mehr schwarze Pixel zwischen zwei Buchstaben, um diese als eigenes Zeichen zu klassifizieren. Es wurde die Hypothese aufgestellt, dass die falsche Klassifikation von Schmutz durch die Normalisierung und damit die Vergrößerung von Schmutz hervorgerufen wird.

Konfigurationsparameter Zunächst wurde versucht, die Falscherkennungen von Schmutz durch verschiedene Konfigurationsparameter zu beheben. Dabei bot sich der Konfigurationsparameter „textord_heavy_nr“ an, was für starke Verringerung von Rauschen steht. Die hierdurch erzielten Ergebnisse waren jedoch eher ernüchternd. Zahlreiche erwünschte Zeichen wurden bei der Erkennung übergangen. Dadurch fehlte ein großer Teil der Satzzeichen und Punkte über i und Umlauten im OCR-Ergebnis. Die Änderung anderer Konfigurationsparameter wie „textord_max_noise_size“ oder „textord_noise_sizefraction“ hatte hingegen keine Änderung des ursprünglichen Erkennungsergebnisses zur Folge.

Änderungen am Quellcode von Tesseract Ein weiterer Ansatz für die Umgehung des Problems war, die Normalisierung von Tesseract abzustellen, um so zu verhindern, dass Schmutz auf die Größe eines Zeichens vergrößert werden kann. Nach kleineren Versuchen wurde diese Idee jedoch wieder fallen gelassen. Die Normalisierung ist so grundlegend in die Funktionsweise der Klassifikation verwoben, dass eine zielführende Behebung des Problems nicht möglich war.

Durch Debugging des Quellcodes von Tesseract anhand eines kleinen Beispiels, das problematischen Schmutz enthält, konnten Stellen im Programm ausgemacht werden, die für die Behandlung von Schmutz zuständig sind. Dort werden Zusammenhangskomponenten anhand ihrer Größe als „Noise“ (Rauschen/Schmutz), „Small“, „Normal“ oder „Large“ klassifiziert. Blobs, die zur Klasse „Noise“ gehören und sich außerhalb eines bei der Layout-Analyse ermittelten Textblocks befinden werden direkt von der Texterkennung ausgenommen. Elemente der Klasse, die sich innerhalb des Texts befinden, werden jedoch zum nächstgelegenen Wort dazugenommen. Anschließend führt die Normalisierung dazu, dass die Merkmale so stark vergrößert werden, dass ein Zeichen klassifiziert werden kann. Der von Tesseract ermittelte Konfidenzwert bleibt dabei jedoch meist gering.

(a) (b)

Abb. 5.3: Verschiedene Arten von Schmutz (a) und erwünschten Blobs (b)

Die meisten falsch klassifizierten Schmutzpartikel haben gemeinsam, dass sie nicht breiter als drei Pixel sind. Durch eine Änderung des Quelltexts wurde dafür gesorgt, dass kleine Blobs, deren Breite vier Pixel und deren Höhe nicht mehr als die Hälfte der Zeilenhöhe beträgt von der Texterkennung ignoriert werden. Es wurde außerdem ein Konfigurationsparameter namens „textord_blob_min_width“ eingeführt, über den die minimale Breite von nicht zu ignorierenden Blobs auch ohne eine Änderung des Quellcodes angepasst werden kann. Standardmäßig liegt dieser Wert bei vier Pixeln. Wird der Wert auf null gesetzt, so wird die Bedingung komplett ignoriert.

Anschließend erfolgte die Evaluation der Anpassung. Die Zeichenfehlerrate konnte auf 2,11 Prozent gesenkt werden. Die Wortfehlerrate sank auf 8,06 Prozent. Damit konnte die Fehlerrate der Texterkennung so stark reduziert werden, dass sie sogar ohne Wörterbuch besser war als die von „deu-frak“ und „tesseract-dan-fraktur“. Nach dieser Änderung lag die Fehlerrate für die meisten Zeichen unterhalb von fünf Prozent. Die Fehlerrate des Apostrophs, der am häufigsten bei Schmutz erkannt wurde, sank von vormals 1099,42 Prozent auf 164,35 Prozent. Die Fehlerrate des Bindestrichs sank von 1075,87 Prozent auf 208,25 Prozent. Die Falscherkennung von Schmutz konnte somit nicht vollständig verhindert werden. Insgesamt stellt dies jedoch eine enorme Verbesserung der Erkennungsergebnisse dar. Verbleibende Fehlerquellen waren die Verwechslung einzelner Zeichen und Sperrungen, für die jeder Buchstabe als eigenes Wort erkannt wurde und damit durch Leerzeichen vom Rest des Worts getrennt war.

5.4 Evaluation verschiedener Wörterbücher

Um die Texterkennung noch weiter zu verbessern, wurden verschiedene Wörterbücher getestet. Für einen ersten Test wurde das Wörterbuch vom Paket „deu-frak“ unverändert übernommen. Da in diesem Wörterbuch jedoch keine Wörter mit langem ſ enthalten sind und somit Wörter mit ſ nicht hätten verbessert werden können, wurde in die Datei „unicharambigs“ die verpflichtende Ersetzungsregel von ſ zu ȿ aufgenommen. Mit diesem Wörterbuch sank die Zeichenfehlerrate weiter auf 2,09 Prozent und die Wortfehlerrate auf 7,99 Prozent.

Als Alternative wurde ein Wörterbuch aus dem Korpus des Deutschen Textarchivs [10] generiert. Dieses enthält laut Projektbeschreibung 1 321 Werke des 17. bis 20. Jahrhunderts aus wissenschaftlicher, belletristischer und Gebrauchsliteratur. Für die Erstellung eines Wörterbuchs wurde der Kernkorpus verwendet, der zur Zeit 1 036 Werke im TEI-P5-Dateiformat enthält. Es wurde ein Parser für TEI-P5 geschrieben, der den enthaltenen Text jedes Dokuments in Wörter unterteilt und anschließend für jedes

Wort die Anzahl an Vorkommen zählt. Anschließend werden alle Wörter die mehr als fünf Mal im gesamten Korpus vorkommen in die Wortliste aufgenommen. Wörter, die mehr als 20 000 Mal auftraten wurden in die Datei „freq-list“ aufgenommen. Während die Zeichenfehlerrate mit diesem Wörterbuch bei 2,09 Prozent lag und damit genauso gut war wie die mit dem Wörterbuch aus „deu-frak“, war die Wortfehlerrate mit 7,88 Prozent leicht besser.

Dieses Ergebnis konnte durch den Einsatz der Datei „unicharambigs“ noch weiter verbessert werden. Dazu wurden die zuletzt gemachten Fehler analysiert und es wurden 58 Ersetzungsregeln abgeleitet. Außerdem wurde eine Liste mit Mustern für Satzzeichen sowie eine Liste mit Mustern für Zahlen erstellt. Durch diese Erweiterungen konnte die Zeichenfehlerrate zusätzlich auf 1,93 Prozent gesenkt werden. Die Wortfehlerrate sank sogar auf 6,85 Prozent.

Ein weiteres alternatives Wörterbuch wurde aus einem durch FineReader erstellten OCR-Text der Fries-Chronik von 1961 generiert. Dieser Text war zuvor von Mitarbeitern der Universitätsbibliothek Würzburg händisch nachkorrigiert worden. Das so generierte Wörterbuch umfasst 16 409 Begriffe. Es wurde dieselbe „unicharambigs“-Datei wie im vorigen Experiment verwendet. Mit diesem Wörterbuch konnte die Zeichenfehlerrate auf 1,90 Prozent gesenkt werden. Die Wortfehlerrate stieg jedoch leicht auf 6,90 Prozent.

5.5 Weitere Verbesserungsmöglichkeiten

Für eine weitere Möglichkeit zur Verbesserung der Erkennungsergebnisse bietet sich die Verwendung der Abmessungsdaten einzelner Zeichen an. Für eine Evaluation dieses Schritts wurden die Informationen unverändert mit dem Programm *set_unicharset_properties* aus dem Verzeichnis „langdata“ für den vorliegenden Zeichensatz übernommen. Die Evaluation wurde mit dem Wörterbuch durchgeführt, das aus dem Deutschen Textarchiv generiert wurde. Hierdurch konnten die Fehlerraten erneut gesenkt werden. Die Zeichenfehlerrate lag nun bei 1,88 Prozent, die Wortfehlerrate bei 6,82 Prozent.

Daneben wurde auch die Binarisierung nach der Methode von Sauvola mit einer Fenstergröße von 15 Pixeln für alle Scans ausprobiert. Hiermit ergab sich eine Zeichenfehlerrate von 1,89 Prozent. Die Wortfehlerrate lag anschließend bei 9,83 Prozent. Es konnte also keine weitere Verbesserung der Erkennungsrate erzielt werden. Allerdings gibt es im Gesamtwerk einzelne Scans, bei denen das Ergebnis durch die Binarisierung nach Sauvola deutlich besser als durch die Binarisierung nach Otsu ist. Für die Scans zu denen eine Transkription vorlag, war die Binarisierung nach Otsu jedoch ausreichend gut. Bei anderen Werken könnte also unter Umständen eine Anpassung der Binarisierung deutliche Verbesserungen bringen.

Danach wurde versucht, den Konfigurationsparameter „language_model_penalty_non_dict_word“ so zu erhöhen, dass die Erkennungsraten steigen. Über diesen Parameter kann der Einfluss der Wortliste auf das Erkennungsergebnis geregelt werden. Wird der Wert erhöht, schließt das Sprachmodell Wörter, die nicht im Wörterbuch enthalten sind, bei der Erkennung eher aus. Der Parameter wurde für die Evaluation von 0,15 auf 0,3 erhöht. Die Zeichenfehlerrate verschlechterte sich jedoch auf 1,91 Prozent.

Auch die Wortfehlerrate stieg auf 6,97 Prozent. Deshalb wurde von der Anpassung dieses Parameters abgesehen.

In einem abschließenden Experiment wurde ein zusätzlicher Verarbeitungsschritt eingefügt, der nach der Texterkennung verbliebene Sperrungen auflöst. Dazu wurden automatisch Abfolgen von Wörtern, die jeweils nur aus einem einzelnen Zeichen bestanden zu einem Wort zusammengefügt. So konnte die Fehlerrate des Leerzeichens von 2,27 Prozent auf 1,98 Prozent gesenkt werden. Insgesamt lag die Zeichenfehlerrate damit bei 1,85 Prozent im Vergleich zu 1,88 Prozent des bisher besten Ergebnisses. Die Wortfehlerrate erreichte mit 6,27 Prozent ebenfalls den besten Wert aller Ergebnisse.

Da es sich bei dieser Verbesserung um das letzte Experiment im Rahmen dieser Arbeit handelt, sind die genauen Fehlerraten für die einzelnen Zeichen in der Tabelle A.2 im Anhang aufgeführt. Damit lassen sich die durch die Arbeit erzielten Verbesserungen für jedes Zeichen im Detail vergleichen.

6 Implementierung eigener Werkzeuge

Um die Analyse und das Training der Texterkennungsergebnisse zu vereinfachen, wurden während der Durchführung immer wieder zusätzliche Softwarekomponenten geschrieben, die die Arbeit erleichtern. So konnten unnötige Wiederholungen von zeitaufwendigen Aufgaben maßgeblich verkürzt werden. Da diese Komponenten auch für zukünftige Projekte hilfreich sein können, werden sie in diesem Kapitel nun vorgestellt. Da für die Zukunft geplant ist, die Softwarekomponenten aus dieser Arbeit in einem Digitalisierungsprojekt einzusetzen, wurden alle Komponenten für die Java-Plattform geschrieben.

6.1 Grundlegende Komponenten

Um Tesseract aus Java-Programmen heraus steuern zu können, existiert bereits das Projekt Tess4j [37]. Das Projekt stellt Java-Komponenten zur Verfügung, die ermöglichen, dass Tesseract von Java aus aufgerufen und gesteuert werden kann. Dafür wird die C-Schnittstelle von Tesseract und die Java-Technologie *Java Native Access* eingesetzt. Leider wurde die aktuelle Entwicklungsversion von Tesseract 3.03 zum Zeitpunkt, als mit der Implementierung der Komponenten begonnen wurde, noch nicht unterstützt.[1] Auch die zugrunde liegende Bildverarbeitungs-Bibliothek *Leptonica* kann durch *Tess4j* nicht verwendet werden. Nach ersten Versuchen mit *Tess4j* wurde daher mit der Implementierung einer eigenen Komponente für die Zusammenarbeit zwischen Leptonica, Tesseract und Java begonnen.

Hierfür wurde das Programm *JNAerator* [16] eingesetzt. Damit ist es möglich, vollständig automatisch aus C- oder C++-Headerdateien eine entsprechende Schnittstelle für Java zu erzeugen. Um also Tesseract aus Java heraus aufrufen zu können, musste mit *JNAerator* lediglich die Header-Datei für die C-Schnittstelle in die entsprechenden Java-Klassen umgewandelt werden. Auch *Leptonica* wurde so für Java-Programme zugänglich gemacht, da für einige Funktionen der Schnittstelle von Tesseract die Verwendung von Klassen aus dieser Bibliothek notwendig ist. Ein weiterer Vorteil dieser Lösung war, dass durch den Einsatz von *JNAerator* die JNA-Schnittstelle problemlos durch eine Schnittstelle über die Software *BridJ* [15] ersetzt werden konnte. Der Vorteil von BridJ-Schnittstellen sind unter anderem Abstraktionen für Zeiger, die durch Java-Generics auch die Typinformationen über das referenzierte Objekt im Speicher enthalten. Hierdurch wird eine fehlerfreie Programmierung erheblich vereinfacht.

Durch diese Arbeiten entstanden die beiden Komponenten *jleptonica* [68] und *jtesseract* [69]. Diese lassen sich durch den Einsatz von *JNAerator* auch einfach und schnell an kommende Versionen von

[1] Mittlerweile werden auch die neueren Entwicklungen von Tesseract in einer Beta-Version unterstützt [38]. Eine Unterstützung der Funktionen von *Leptonica* existiert jedoch weiterhin nicht.

Tesseract anpassen. Die meisten der Komponenten, die in diesem Kapitel beschrieben werden, bauen grundlegend auf beiden Software-Bibliotheken auf.

6.2 Vergleichswerkzeug für die Texterkennung

Um die Erkennungsergebnisse der Layoutanalyse und der weiteren Segmentierungsschritte überprüfen zu können, kann es hilfreich sein, neben der reinen Textausgabe auch Informationen über die Lage von Textblöcken, Absätzen, Zeilen, Wörtern und Zeichen zu betrachten. Die entsprechenden Koordinaten sind im hOCR-Dateiformat zwar enthalten, lassen sich jedoch nur schwer überprüfen. Daher wurde zunächst ein Programm geschrieben, welches die Ergebnisse der Texterkennung in einem dynamisch erzeugten Bild neben dem Scan darstellen kann. Für eine erste Version wurde ein Parser für das hOCR-Format geschrieben. Dieser extrahiert die Informationen über die Koordinaten für die verschiedenen Hierarchiestufen sowie den jeweils enthaltenen Text und legt diese strukturiert im Speicher ab.

Zusätzlich zum Parser wurde eine grafische Komponente implementiert, die die Darstellung der Zeichen an den ermittelten Koordinaten ermöglicht. Dazu wird zunächst ein leeres Bild im Arbeitsspeicher erzeugt, das dieselben Abmessungen wie der Scan hat. In dieses Bild werden anschließend die erkannten Zeichen sowie umschließende Rechtecke eingezeichnet und zum Schluss am Bildschirm dargestellt. Die Darstellung der Rechtecke erfolgt genauso an den zugehörigen Stellen im Scan. Der Nutzer kann dabei bestimmen, ob Rechtecke um einzelne Zeichen oder ganze Wörter gelegt werden sollen. Außerdem können die erkannten Blöcke und Absätze als Rechtecke eingezeichnet werden. Die Darstellung des Texts erfolgt wahlweise in Fraktur oder Antiqua. Neben jeder Zeile kann auch die zugehörige Zeilennummer dargestellt werden. Zudem kann die Darstellung bis auf 200 Prozent vergrößert oder auf 10 Prozent verkleinert werden. Es wurde auch dafür gesorgt, dass beide Darstellungen beim Scrollen immer synchron bleiben. Fährt man mit der Maus über ein einzelnes Zeichen, wird der zugehörige, bei der Texterkennung ermittelte, Konfidenzwert in einem Overlay angezeigt.

Um auch die ermittelten Grundlinien jedes Worts anzeigen und dadurch überprüfen zu können, wurde Tesseract so angepasst, dass auch Informationen darüber in den hOCR-Dateien abgelegt werden. Diese Anpassungen wurden später auch offiziell durch Tesseract, jedoch nur für ganze Zeilen, übernommen und sind nun Teil des aktuellen Entwicklungszweigs.

Im weiteren Verlauf der Entwicklung wurde das Vergleichswerkzeug so angepasst, dass es statt vorher manuell erzeugter hOCR-Dateien nun die Programmierschnittstelle von Tesseract über *jtesseract* nutzt. Das Vergleichswerkzeug kann ganze Verzeichnisse von Scans einlesen und stellt jeden Scan in einer Liste mit Vorschaubildern und dem zugehörigen Dateinamen dar. Wählt man einen Scan aus, so wird die Zeichenerkennung durch Tesseract gestartet und anschließend wird das Ergebnis dargestellt. Über die Programmierschnittstelle konnte außerdem umgesetzt werden, dass Sprachpakete nachträglich ausgewählt werden können. Dazu wird beim Start das Verzeichnis „tessdata“ eingelesen und alle enthaltenen „traineddata“-Dateien werden in einer Liste zur Auswahl angeboten. Abbildung 6.1 zeigt das Vergleichswerkzeug in der aktuellen Version.

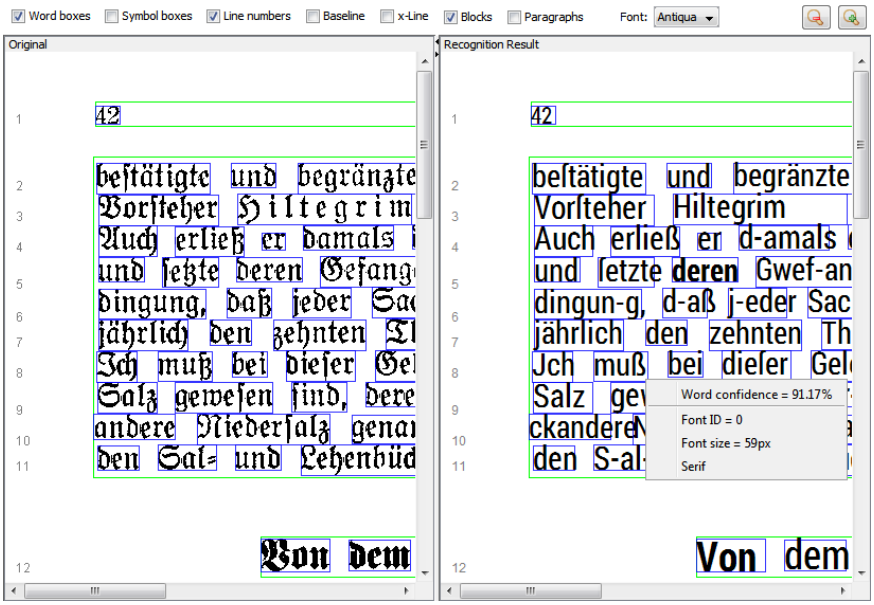


Abb. 6.1: Vergleichsansicht für das OCR-Ergebnis

Werden die Boxen der Wörter eingezeichnet, so kann durch einen Rechtsklick innerhalb einer Box ein Kontextmenü aufgerufen werden, das Informationen über den Konfidenzwert des Wortes sowie die ermittelten Informationen über die Schriftart, wie fett oder kursiv, angezeigt (vgl. Abbildung 6.1). Durch eine Erweiterung der Programmierschnittstelle von Tesseract können außerdem die alternativen Kandidaten für jedes erkannte Zeichen angezeigt werden. Dazu müssen die Rahmen um einzelne Zeichen aktiv sein. Anschließend zeigt das Kontextmenü nach einem Rechtsklick auf ein Zeichen dessen Alternativen sowie die zugehörigen Konfidenzen der Alternativen an. Für die Zukunft wäre denkbar, durch die Alternativen eine schnelle Korrektur der Erkennungsergebnisse zu ermöglichen. Die Erweiterung der Schnittstelle von Tesseract wurde ebenfalls in den Haupt-Entwicklungspfad des Projekts übernommen [70].

6.3 Box-Editor mit Glyphen-Übersicht

Neben der Analyse der Erkennungsergebnisse wurde auch der Trainingsprozess durch eigene Werkzeuge vereinfacht. Der Trainingsprozess ist sehr anfällig gegenüber Fehlern. Größte Fehlerquelle sind dabei falsche Zuordnungen von Zeichen zu Boxen in den Box-Dateien zu jedem Beispiel. Eine einzige falsch deklarierte Box kann den Klassifikator für das Zeichen so verfälschen, dass die Erkennungsrate des zugewiesenen Unicode-Zeichens messbar sinkt (vgl. Kapitel 4.3.2 und 5.2). Gleichzeitig kann es sehr zeitaufwendig sein, einen Fehler in den Box-Dateien zu finden, wenn viele Box-Dateien für das Training verwendet werden.

Für das Editieren von Box-Dateien beim Training neuer Schriftarten wird ein sogenannter Box-Editor benötigt. Mit *jTessBoxEditor* existiert bereits ein sehr zuverlässiges Werkzeug, das die Verbesserung

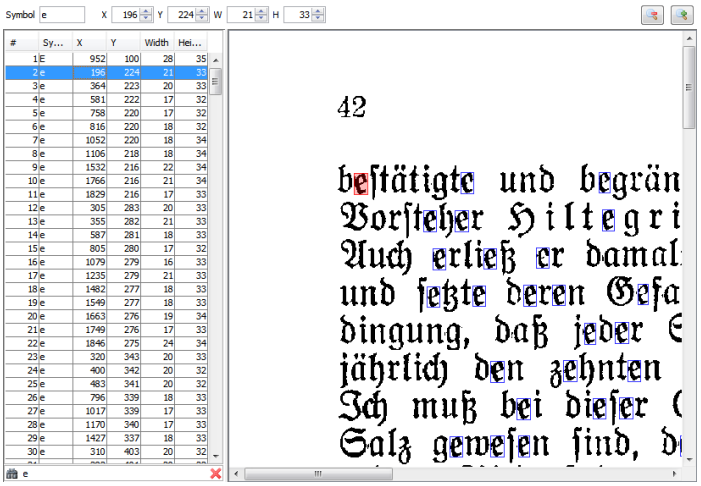


Abb. 6.2: Box-Editor mit aktivem Filter „e“

von Box-Dateien visuell unterstützt. In früheren Arbeiten zum Thema Tesseract wurde bereits ein Box-Editor mit erweiterter Funktionalität vorgestellt (vgl. [12, S. 55–59]; [65, S. 20–22]). Dieser wurde für die hier vorgestellten Werkzeuge angepasst und in die grafische Benutzungsoberfläche übernommen. Abbildung 6.2 zeigt den Box-Editor.

Der Funktionsumfang des Box-Editors ähnelt dem von *jTessBoxEditor* sehr. Dabei wird eine Tabelle mit Zeichen und zugehörigen Koordinaten angezeigt. Daneben wird der zur Box-Datei gehörende Scan angezeigt in dem die Boxen um jedes Zeichen als Rechtecke dargestellt werden. Einzelne Boxen können entweder im Scan oder in der Tabelle selektiert und anschließend über Kontrollelemente angepasst werden. Zusätzlich erleichtert ein Textfeld die Arbeit, über das die Einträge der Tabelle gefiltert werden können. Beim Filtern werden ausschließlich Boxen angezeigt, deren Text auch die eingegebenen Zeichen enthält. Dies gilt sowohl für die tabellarische Übersicht als auch für den Scan. So können häufig auftretende Probleme gezielt überprüft und bearbeitet werden. *jTessBoxEditor* bietet zwar auch eine Suchfunktion, dabei wird jedoch lediglich ein Eintrag in der Tabelle angesprungen. Einträge, auf die der Suchbegriff nicht passt, bleiben weiterhin sichtbar, was eine schnelle Bearbeitung erschwert.

Die eigentliche Innovation stellt aber die Glyphen-Übersicht dar. Über einen Reiter lässt sich auf diese Ansicht umschalten. In der Glyphen-Übersicht wird initial eine Liste der verschiedenen Zeichen, genauer gesagt der verschiedenen Unichars, angezeigt. Die Liste ist absteigend nach der Anzahl der Boxen für jedes Unichar sortiert. Wird ein Eintrag der Liste ausgewählt, werden daneben alle entsprechenden Ausschnitte für jedes Exemplar des Unichars in Form eines Tableaus angezeigt. In dieser Übersicht kann schnell festgestellt werden, ob eine Box als falsches Unichar deklariert wurde. Um die Situation noch weiter zu vereinfachen, können die Ausschnitte wahlweise nach dem Konfidenzwert, der Fläche des Ausschnitts (Breite × Höhe der Box) sowie dem Bildgewicht des Ausschnitts jeweils absteigend sortiert werden. Das Bildgewicht errechnet sich aus der Anzahl schwarzer Pixel im Ausschnitt. Hierdurch werden beispielsweise Boxen, die Schmutz enthalten, an das Ende des Tableaus gestellt und können so

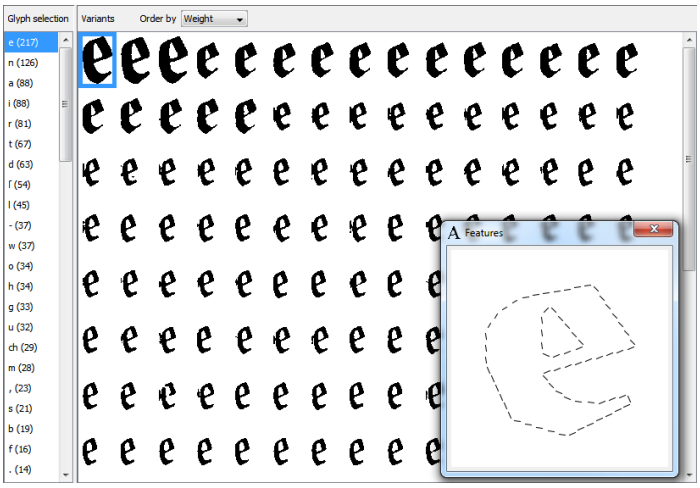


Abb. 6.3: Glyphen-Übersicht und durch Tesseract ermittelte Merkmale einer Glyphe

leicht identifiziert werden. Abbildung 6.3 zeigt die Glyphen-Übersicht mit Varianten des Buchstaben e. Über ein Kontextmenü kann zusätzlich für jede Glyphe ein Dialogfenster geöffnet werden, das die durch Tesseract ermittelten Merkmale der Glyphe anzeigt. Außerdem kann über einen weiteren Eintrag im Kontextmenü an die entsprechende Stelle des Box-Editors gesprungen werden, sodass beispielsweise ein fehlerhafter Eintrag, der in der Glyphen-Übersicht gefunden wurde, direkt verbessert werden kann.

6.4 Evaluationswerkzeug

Eine weitere Komponente der grafischen Oberfläche ist ein Werkzeug zur Evaluation, das die in den vorigen Kapiteln zur Evaluation der Texterkennungsergebnisse eingesetzte Software *ocrevalUAtion* in die Oberfläche integriert. Zu jedem Scan eines Projekts kann eine Transkription der Seite über ein Textfeld eingegeben und gespeichert werden. Wird der Scan später erneut geöffnet, wird die Transkription automatisch geladen. Außerdem kann die Transkription vom Texterkennungsergebnis übernommen werden, sodass diese nur noch verbessert werden muss. Über den Menüpunkt „File → Import Transcriptions…“ lässt sich zudem eine Textdatei einlesen, die die Transkriptionen für das gesamte Projekt enthält. Mehrere Seiten werden dabei durch die Zeichenkette „~~~~~“ auf einer eigenen Zeile voneinander getrennt. Diese kann jedoch bei Bedarf geändert werden. Abbildung 6.4 zeigt die Evaluationsansicht.

Nachdem die Transkription eingegeben wurde kann der Vergleich des OCR-Ergebnisses mit der Transkription über die Schaltfläche „Generate Report“ gestartet werden. *ocrevalUAtion* erstellt dann einen Vergleichsbericht, der anschließend automatisch durch den auf dem System installierten Browser geöffnet wird. Dieser enthält die in Kapitel 4.3.1 besprochenen Werte wie Zeichen- und Wortfehlerraten für das gesamte Dokument sowie für jedes Zeichen und auch die Vergleichsansicht beider Texte (vgl. Abbildung 4.11).

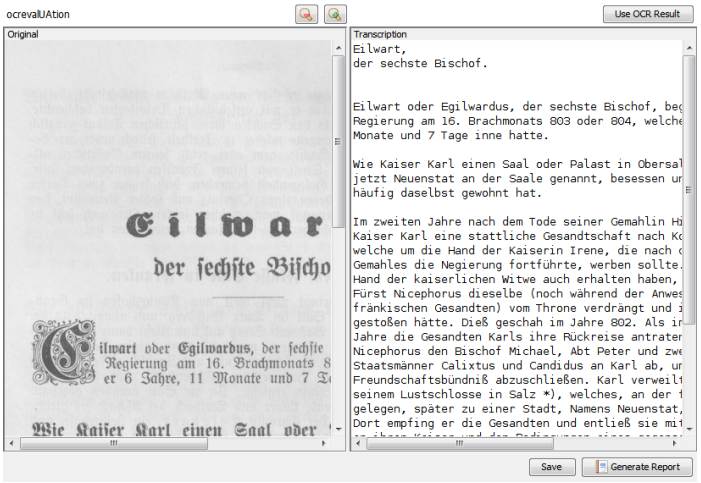


Abb. 6.4: Integration des Evaluationswerkzeugs *ocrevalUAtion*

6.5 Weitere Analysewerkzeuge

Ein weiteres Werkzeug zur Analyse ist ein Debugging-Tool für die minimalen und maximalen Abmessungen eines Unichars in der Datei „unicharset“. Hierdurch können die dort angegebenen Werte auf ihre Plausibilität überprüft werden. Der Debugger dient jedoch nur der Überprüfung der Werte. Werden Unstimmigkeiten entdeckt, so müssen die Werte über einen Texteditor in der „unicharambigs“-Datei selbst angepasst werden. Das Werkzeug lässt sich über das Menü der grafischen Oberfläche und den Eintrag „Tools → Debug Unicharset…“ aufrufen.

Außerdem wurde ein kleines Tool geschrieben, das die schnelle Überprüfung einer Textdatei bezüglich ihrer Abdeckung des Zeichensatzes ermöglicht. Das ist hilfreich, wenn neue Trainingsbeispiele erstellt werden. Sofern das Trainingsbeispiel auch als reine Textdatei vorliegt, kann diese über das Werkzeug geöffnet werden. Anschließend werden die Zeichen in einer Tabelle aufgelistet. Jeweils daneben steht die Anzahl der Vorkommen des Zeichens in der Datei. Das Tool kann über den Menüeintrag „Tools → Character Histogram…“ aufgerufen werden.

6.6 Automatisches Trainingswerkzeug

Da die Eingabe der Befehle, die zum Training über die Kommandozeile einzugeben sind, viel Zeit erfordert, wurde außerdem noch eine Komponente implementiert, die die Trainingsdurchführung nach der Erstellung von Trainingsbeispielen, zugehörigen Box-Dateien und der Datei „font_properties“ vollständig automatisiert (vgl. Abbildung 6.5). Über einen Datei-Dialog kann das Verzeichnis, welches die ausführbaren Dateien von Tesseract enthält, spezifiziert werden. Außerdem muss das Verzeichnis angegeben werden, welches die Trainingsbeispiele und Box-Dateien enthält. Optional können auch die Abmessungsdaten aus dem Verzeichnis „langdata“ übernommen werden. Dazu muss auch der Pfad dieses Verzeichnisses angegeben werden. Nach dem Training öffnet sich ein Fehlerbericht, der genaue

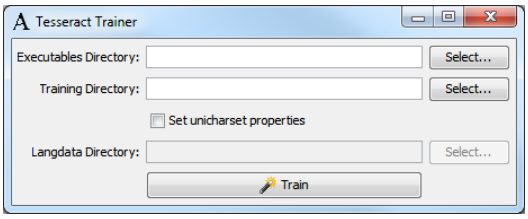


Abb. 6.5: Tesseract Trainer

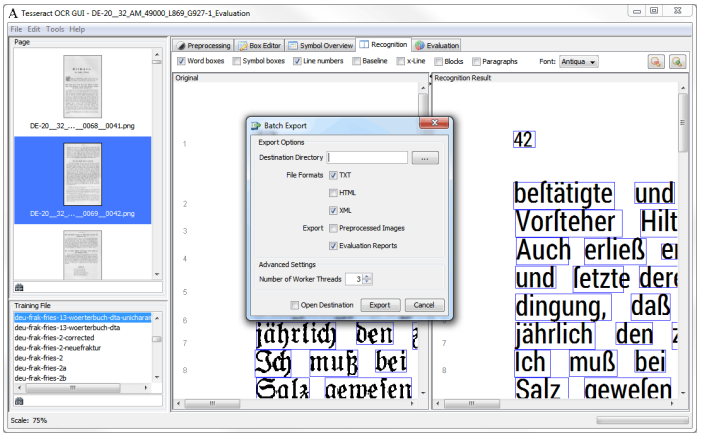


Abb. 6.6: Komplettansicht und Stapelverarbeitung

Informationen zu den beim Training aufgetretenen Problemen enthält. Das Ergebnis des Trainings liegt anschließend als Datei „traineddata“ im selben Verzeichnis wie die Trainingsbeispiele.

6.7 Stapelverarbeitung

Die letzte Komponente, die im Rahmen dieser Arbeit erstellt wurde, ermöglicht die Stapelverarbeitung eines gesamten Verzeichnisses. Die Stapelverarbeitung lässt sich über den Menüpunkt „File → Batch Export…“ starten. Dabei erscheint ein Dialog, über den das Zielverzeichnis ausgewählt werden kann. Außerdem können die erwünschten Ausgabe-Dateiformate ausgewählt werden. Auch die binarisierten Bilder und die Evaluationsberichte können bei Bedarf ausgegeben werden. Hierfür wird durch *ocrevalUAtion* zu jedem Scan des Verzeichnisses, zu dem auch eine Transkription vorliegt, ein eigener HTML-Bericht im Zielverzeichnis erstellt. Außerdem werden alle Berichte in einem Bericht, der alle Scans des Verzeichnisses umfasst, mit dem Dateinamen „project.report.html“ zusammengefasst. Während der Stapelverarbeitung wird eine Log-Datei erstellt, die eventuell auftretende Fehler enthält.

Die Stapelverarbeitung wurde in einer ersten Version so implementiert, dass alle Scans sequentiell abgearbeitet werden. Später wurde diese Version so angepasst, dass auch mehrere Kerne des Systems parallel verwendet werden können. Dafür lässt sich im Dialog der Stapelverarbeitung die Anzahl der zu verwendeten Kerne anpassen. Die Voreinstellung entspricht der Anzahl der logischen Kerne des Systems abzüglich eines Kerns. Diese Einstellung ist sinnvoll, wenn neben der Stapelverarbeitung

weiter gearbeitet werden möchte. Werden alle Kerne verwendet, so liegt die Auslastung des Systems durchgängig bei 100 Prozent. Ein komfortables Arbeiten ist daher nicht mehr möglich.

Durch die Verwendung mehrerer Kerne konnte die Dauer für das Verarbeiten des transkribierten Teils der Fries-Chronik auf einem System mit zwei Kernen und Hyperthreading (d. h. vier logischen Kernen) von anfänglich etwas mehr als dreißig Minuten auf etwa zehn Minuten reduziert werden. Dabei ist die erforderliche Dauer jedoch abhängig vom ausgewählten Sprachpaket. Sprachpakete mit sehr großen Wörterbüchern oder vielen möglichen Zeichen benötigen länger als Sprachpakete mit nur einer Schriftart und ohne Wörterbuch.

7 Fazit und Ausblick

Durch die verschiedenen Experimente, Trainings und Anpassungen des Quellcodes im Rahmen dieser Arbeit konnte gezeigt werden, dass Tesseract deutliche Verbesserungen von Erkennungsergebnissen für Frakturtexte ermöglicht. Das Sprachpaket „deu-frak“ liefert eine gute Ausgangsbasis. Durch gezieltes Training von Beispielen, die entweder aus Scans des zu erkennenden Texts bestehen, aber auch künstlich erzeugt werden können, lassen sich die Fehlerraten jedoch noch deutlich senken.

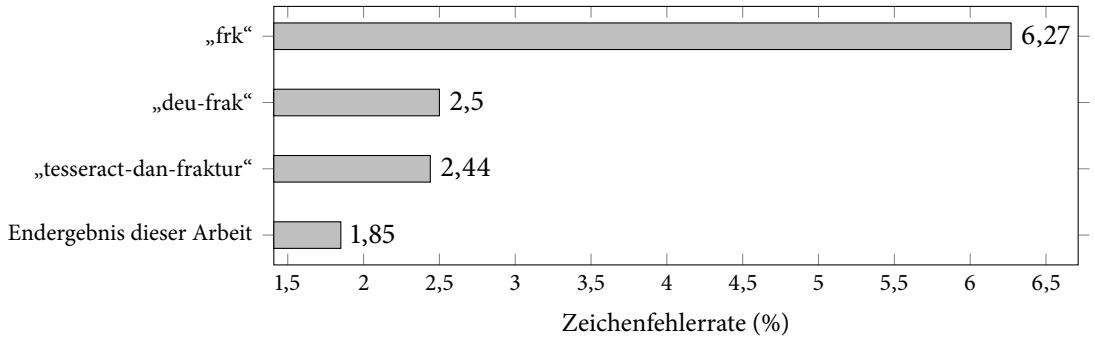


Abb. 7.1: Vergleich der Zeichenfehlerraten der verwendeten Sprachpakete

Durch die in dieser Arbeit vorgestellten Trainings und Anpassungen konnten die Zeichenfehlerraten im Vergleich zu den Paketen „deu-frak“ und „tesseract-dan-fraktur“ deutlich gesenkt werden (vgl. Abbildung 7.1). Vor allem die Unterscheidung optisch ähnlicher Zeichen konnte enorm verbessert werden. Nach dem Training von Box-Dateien, die das Unicode-Zeichen U+017F für Exemplare des langen ſ verwenden, wurden die Fehlerraten dieser Zeichen deutlich reduziert. Bei „deu-frak“ und „tesseract-dan-fraktur“ wurde die Unterscheidung zwischen langem ſ und ſ fast vollständig dem Wörterbuch überlassen.

Auch Fehler, welche durch falsch klassifizierten Schmutz hervorgerufen wurden, konnten zu einem großen Teil beseitigt werden. Hierfür wurde Tesseract an einer entscheidenden Stelle so angepasst, dass Schmutz bei der Texterkennung ignoriert wird. Durch die Erstellung von zielgerichteten Wörterbüchern konnten zudem weitere Fortschritte erzielt werden.

Während der zahlreichen Trainingsexperimente wurde festgestellt, dass es beim Erstellen und Bearbeiten von Box-Dateien häufig zu Fehlern kommt, die später nur noch schwer nachvollziehbar sein können. Es wurden Softwarekomponenten implementiert und vorgestellt, die die Arbeit mit Box-Dateien enorm erleichtern und dazu beitragen, Fehler schnell finden und beseitigen zu können. Die Glyphen-Übersicht

stellt jedes Exemplar eines Unichars in einem Tableau dar und ermöglicht die Sortierung der Exemplare nach verschiedenen Gesichtspunkten, sodass Fehler schnell auffallen.

Auch die weiteren vorgestellten Komponenten bieten Unterstützung beim Trainings- und Evaluationsprozess. Durch die Betrachtungswerkzeuge für Merkmale eines Zeichens und die Abmessungen einzelner Unichars kann das Training überprüft werden. Das automatische Trainingswerkzeug beschleunigt die Erstellung von Sprachpaketen enorm.

Für die Evaluation der Texterkennungsergebnisse eignen sich das visuelle Vergleichswerkzeug sowie das Evaluationswerkzeug, das automatisch Fehlerberichte erstellt. Mit dem Vergleichswerkzeug sind schnelle Überprüfungen bestimmter Textstellen leicht möglich. Außerdem können bei der Erkennung ermittelte Alternativen für jedes Zeichen angezeigt werden. Durch das Anzeigen der ermittelten Konfidenzwerte kann auch nachvollzogen werden, weshalb es zu bestimmten Entscheidungen kommt.

Die Stapelverarbeitung rundet die Softwarekomponenten ab. Hiermit kann die Texterkennung ohne großen Aufwand auf mehreren Scans durchgeführt werden. Nur dadurch konnten die zahlreichen Fehleranalysen für die gesamte Transkription angefertigt werden.

Um die Texterkennung von Tesseract für Frakturtexte noch weiter zu verbessern, bleiben für zukünftige Projekte verschiedene Punkte offen. So führt die durch Tesseract berechnete Polygon-Approximation sehr häufig zu Fehlern, da Details bei sehr ähnlichen Glyphen teilweise zu stark abstrahiert werden, um diese noch gut voneinander unterscheiden zu können. Einerseits bietet sich ein besserer Algorithmus zur Berechnung einer solchen Approximation an. Das Programm Potrace [51] enthält beispielsweise einen Algorithmus, der Formen sehr präzise durch ein Polygon abstrahiert. Smith [53] schlägt hingegen vor, die Approximation auf lange Sicht vollständig überflüssig zu machen.

Ein weiterer möglicher Ansatz zur Verbesserung der Erkennung von Fraktur wäre, Initialen in das Training mit aufzunehmen. Ein Experiment wurde hierzu noch nicht durchgeführt. Im Moment werden Initialen meist noch falsch klassifiziert. Da sie jedoch nur einen sehr geringen Teil des Texts ausmachen, fallen die hierdurch gemachten Fehler kaum ins Gewicht.

Eine weitere Verbesserungsmöglichkeit wäre, ein aufwendigeres Sprachmodell für Tesseract zu implementieren, das auch besser mit zusammengesetzten Wörtern zurecht kommt, wie sie in der deutschen Sprache üblich sind. Die Programmierschnittstelle von Tesseract bietet bereits Methoden zur Verwendung eines Bigramm-Modells, dieses wurde im Rahmen dieser Arbeit jedoch noch nicht evaluiert.

Um die schnelle Verbesserung von OCR-Ergebnissen zu ermöglichen, könnten zudem die vorgestellten Werkzeuge zur Verwendung und Analyse von Tesseract zu Korrektur-Werkzeugen erweitert werden. Das automatische Suchen und Ersetzen im OCR-Ergebnis oder die grafisch unterstützte Bearbeitung der Ergebnisse wären weitere mögliche Entwicklungszweige.

Die in dieser Arbeit getätigten Experimente und vorgestellten Softwarekomponenten können somit die Grundlage für die genannten Aspekte und weitere Forschung darstellen.

A Fehlerstatistiken

Es folgt die tabellarische Auflistung der Fehlerraten der Sprachpakete „deu-frak“ und „deu-frak-fries“. „Gesucht“ bezeichnet jeweils die Anzahl der Exemplare eines Zeichens in der Transkription. „Beibeh.“, „Einf.“, „Ers.“ und „Lösch.“ bezeichnen die Anzahl der für die Levenshtein-Distanz gezählten beibehaltenen, eingefügten, ersetzten und gelöschten Exemplare des jeweiligen Zeichens im OCR-Ergebnis. Zeichen, welchen bei der Fehlerrate kein Wert zugewiesen ist, kommen in der Transkription nicht vor, weswegen die Fehlerrate nicht berechnet werden kann.

Tab. A.1: Fehlerraten des Sprachpakets „deu-frak“

Zeichen	Gesucht	Beibeh.	Einf.	Ers.	Lösch.	Fehlerrate (%)
(Leerzeichen)	183144	181590	2654	0	1554	2,30
!	60	40	11	19	1	51,67
$	3	0	0	3	0	100,00
&	1	0	0	1	0	100,00
(	1047	1029	18	7	11	3,44
)	1631	1584	35	18	29	5,03
*	630	0	0	531	99	100,00
,	14642	14283	148	194	165	3,46
-	630	479	1912	33	118	327,46
.	11691	10212	399	611	868	16,06
/	6	0	0	6	0	100,00
0	1712	1682	9	17	13	2,28
1	2337	2266	46	37	34	5,01
2	1224	1179	8	16	29	4,33
3	1210	1119	12	57	34	8,51
4	1162	1125	17	10	27	4,65
5	790	753	13	12	25	6,33
6	638	611	6	11	16	5,17
7	622	597	16	8	17	6,59
8	546	529	8	10	7	4,58
9	590	566	7	12	12	5,25
:	485	462	25	10	13	9,90

Tab. A.1: Fehlerraten des Sprachpakets „deu-frak“ (Forts.)

Zeichen	Gesucht	Beibeh.	Einf.	Ers.	Lösch.	Fehlerrate (%)
;	216	205	28	7	4	18,06
=	0	0	1	0	0	—
>	1	0	0	0	1	100,00
?	16	11	3	5	0	50,00
A	4154	4127	12	15	12	0,94
B	6864	6798	24	41	25	1,31
C	618	516	10	101	1	18,12
D	3551	3538	6	7	6	0,54
E	2536	2237	29	289	10	12,93
F	2621	2166	16	446	9	17,97
G	4412	4377	8	21	14	0,97
H	4483	4387	28	81	15	2,77
I	2069	518	78	1532	19	78,73
J	1933	1839	40	81	13	6,93
K	3936	3908	7	10	18	0,89
L	1844	1826	26	13	5	2,39
M	2977	2940	31	16	21	2,28
N	1412	1282	22	125	5	10,76
O	963	949	17	12	2	3,22
P	1788	1689	2	89	10	5,65
Q	39	37	2	2	0	10,26
R	2679	2456	8	212	11	8,62
S	7113	7039	17	48	26	1,28
T	1764	1754	22	3	7	1,81
U	1336	1329	10	3	4	1,27
V	2363	1757	14	596	10	26,24
W	3918	3885	45	18	15	1,99
X	96	77	33	18	1	54,17
Y	0	0	7	0	0	—
Z	1138	930	16	202	6	19,68
^	1	0	0	1	0	100,00
a	48780	48626	44	107	47	0,41
b	20122	19998	20	86	38	0,72
c	29398	29170	47	174	54	0,94
d	49235	49126	28	56	53	0,28

Tab. A.1: Fehlerraten des Sprachpakets „deu-frak“ (Forts.)

Zeichen	Gesucht	Beibeh.	Einf.	Ers.	Lösch.	Fehlerrate (%)
e	164113	163789	134	178	146	0,28
f	18102	17020	55	921	161	6,28
g	31731	31553	24	29	149	0,64
h	48991	48740	48	120	131	0,61
i	66935	66634	895	207	94	1,79
j	844	832	24	12	0	4,27
k	6310	6126	58	170	14	3,84
l	34217	34036	145	130	51	0,95
m	21762	21547	24	190	25	1,10
n	99505	98997	182	382	126	0,69
o	26843	26724	19	87	32	0,51
p	4020	3996	21	19	5	1,12
q	72	69	8	3	0	15,28
r	74297	74044	69	147	106	0,43
s	50558	49895	1189	579	84	3,66
t	52844	52611	95	167	66	0,62
u	38749	38094	33	569	86	1,78
v	8965	8789	37	166	10	2,38
w	11681	11632	18	36	13	0,57
x	81	77	40	3	1	54,32
y	652	632	6	18	2	3,99
z	12298	12190	99	60	48	1,68
¤	1	0	0	0	1	100,00
§	2	2	0	0	0	0,00
«	5	0	501	5	0	10120,00
°	1	0	0	1	0	100,00
·	0	0	261	0	0	—
»	1	0	255	0	1	25600,00
¼	4	0	0	0	4	100,00
Ã	5	0	0	5	0	100,00
Ä	0	0	3	0	0	—
Æ	0	0	1	0	0	—
Ö	2	0	0	2	0	100,00
Ø	0	0	2	0	0	—
Ü	5	0	0	4	1	100,00

Tab. A.1: Fehlerraten des Sprachpakets „deu-frak“ (Forts.)

Zeichen	Gesucht	Beibeh.	Einf.	Ers.	Lösch.	Fehlerrate (%)
ß	3259	3200	0	57	2	1,81
ä	4627	4471	9	150	6	3,57
å	0	0	2	0	0	—
ö	2874	2797	4	74	3	2,82
ü	7487	7123	5	354	10	4,93
—	67	65	89	2	0	135,82
’	345	318	38	6	21	18,84
“	200	1	72	194	5	135,50
„	292	0	0	247	45	100,00
†	33	0	0	29	4	100,00
(Breitenl. Leerz.)	1	0	0	0	1	100,00

Tab. A.2: Fehlerraten des Endergebnisses

Zeichen	Gesucht	Beibeh.	Einf.	Ers.	Lösch.	Fehlerrate (%)
(Leerzeichen)	183144	180851	1330	0	2293	1,98
!	60	48	5	10	2	28,33
$	3	0	0	3	0	100,00
&	1	0	2	1	0	300,00
(	1047	1032	22	4	11	3,53
)	1631	1589	54	12	30	5,89
*	630	581	127	32	17	27,94
,	14642	14224	180	277	141	4,08
-	630	482	631	20	128	123,65
.	11691	10294	433	505	892	15,65
/	6	0	0	6	0	100,00
0	1712	1693	19	8	11	2,22
1	2337	2229	35	70	38	6,12
2	1224	1177	7	18	29	4,41
3	1210	1148	14	21	41	6,28
4	1162	1127	21	9	26	4,82
5	790	752	10	11	27	6,08
6	638	605	9	18	15	6,58
7	622	592	17	11	19	7,56
8	546	532	7	9	5	3,85

Tab. A.2: Fehlerraten des Endergebnisses (Forts.)

Zeichen	Gesucht	Beibeh.	Einf.	Ers.	Lösch.	Fehlerrate (%)
9	590	565	7	10	15	5,42
:	485	469	124	3	13	28,87
;	216	179	22	32	5	27,31
>	1	0	0	0	1	100,00
?	16	11	42	5	0	293,75
A	4154	4128	38	9	17	1,54
B	6864	6791	17	46	27	1,31
C	618	602	6	15	1	3,56
D	3551	3535	6	8	8	0,62
E	2536	2509	7	17	10	1,34
F	2621	2594	34	15	12	2,33
G	4412	4380	8	19	13	0,91
H	4483	4384	41	81	18	3,12
I	2069	830	46	1192	47	62,11
J	1933	865	24	1051	17	56,49
K	3936	3906	15	13	17	1,14
L	1844	1825	11	14	5	1,63
M	2977	2942	40	14	21	2,52
N	1412	1370	29	35	7	5,03
O	963	951	13	8	4	2,60
P	1788	1773	4	7	8	1,06
Q	39	34	7	5	0	30,77
R	2679	2455	15	212	12	8,92
S	7113	7051	4	28	34	0,93
T	1764	1752	20	4	8	1,81
U	1336	1323	19	6	7	2,40
V	2363	2182	24	171	10	8,68
W	3918	3885	40	15	18	1,86
X	96	87	100	7	2	113,54
Y	0	0	5	0	0	—
Z	1138	1038	49	97	3	13,09
^	1	0	0	1	0	100,00
a	48780	48561	43	147	72	0,54
b	20122	19967	23	103	52	0,88
c	29398	29307	104	35	56	0,66

Tab. A.2: Fehlerraten des Endergebnisses (Forts.)

Zeichen	Gesucht	Beibeh.	Einf.	Ers.	Lösch.	Fehlerrate (%)
d	49235	49051	39	96	88	0,45
e	164113	163208	115	658	247	0,62
f	18102	17434	59	403	265	4,02
g	31731	31514	50	41	176	0,84
h	48991	48634	54	199	158	0,84
i	66935	65827	68	903	205	1,76
j	844	841	18	2	1	2,49
k	6310	6255	146	35	20	3,19
l	34217	34002	184	152	63	1,17
m	21762	21486	71	240	36	1,59
n	99505	98646	279	564	295	1,14
o	26843	26705	44	95	43	0,68
p	4020	4003	28	8	9	1,12
q	72	69	20	3	0	31,94
r	74297	73933	224	215	149	0,79
s	50558	50184	172	273	101	1,08
t	52844	52551	102	198	95	0,75
u	38749	38105	73	523	121	1,85
v	8965	8909	78	46	10	1,49
w	11681	11616	156	50	15	1,89
x	81	77	147	4	0	186,42
y	652	648	21	3	1	3,83
z	12298	12216	147	30	52	1,86
¤	1	0	0	0	1	100,00
§	2	0	0	2	0	100,00
«	5	0	0	5	0	100,00
°	1	0	0	1	0	100,00
·	0	0	27	0	0	—
»	1	0	0	0	1	100,00
¼	4	0	0	1	3	100,00
Ã	5	0	0	5	0	100,00
Ö	2	0	1	2	0	150,00
Ü	5	0	2	4	1	140,00
ß	3259	3250	13	5	4	0,68
ä	4627	4532	7	92	3	2,20

Tab. A.2: Fehlerraten des Endergebnisses (Forts.)

Zeichen	Gesucht	Beibeh.	Einf.	Ers.	Lösch.	Fehlerrate (%)
ö	2874	2834	7	39	1	1,64
ü	7487	7252	10	220	15	3,27
-	67	1	29	64	2	141,79
–	0	0	5	0	0	—
’	345	332	189	3	10	58,55
“	200	188	36	10	2	24,00
”	0	0	3	0	0	—
„	292	235	33	12	45	30,82
†	33	27	2	6	0	24,24
fi	0	0	1	0	0	—
(Breitenl. Leerz.)	1	0	0	0	1	100,00

Literaturverzeichnis

[1] ABBYY. *FineReader*. 1993–2014. URL: http://finereader.abbyy.de/ (abgerufen am 13. 08. 2014).

[2] ABBYY. *FineReader 12 Professional. Technische Spezifikationen*. 2014. URL: http://finereader.abbyy.de/professional/technische-spezifikationen/ (abgerufen am 13. 08. 2014).

[3] ABBYY. *FineReader XIX*. 2003–2010. URL: http://frakturschrift.de/ (abgerufen am 13. 08. 2014).

[4] ABBYY. *Recognition Server*. 2006–2014. URL: http://abbyyeu.com/rs/ (abgerufen am 13. 08. 2014).

[5] ABBYY. *Unternehmensübersicht. Geschichte*. 2014. URL: http://www.abbyy.de/unternehmen/geschichte/ (abgerufen am 29. 08. 2014).

[6] Adobe Systems. *Adobe Photoshop*. 1990–2014. URL: http://www.adobe.com/de/products/photoshop.html (abgerufen am 31. 08. 2014).

[7] P. A. Alberti. *tesseract-dan-fraktur. Tesseract ocr training data for Danish written in fraktur script and a few other languages*. 2011. URL: https://github.com/paalberti/tesseract-dan-fraktur (abgerufen am 07. 09. 2014).

[8] Apache Software Foundation. *Apache OpenOffice*. 2012–2014. URL: http://www.openoffice.org/ (abgerufen am 31. 08. 2014).

[9] A. W. Appel und G. J. Jacobson. „The world's fastest Scrabble program". In: *Communications of the ACM* 31.5 (1988), S. 572–578. DOI: 10.1145/42411.42420.

[10] Berlin-Brandenburgische Akademie der Wissenschaften. *Deutsches Textarchiv*. Hrsg. von Berlin-Brandenburgische Akademie der Wissenschaften. 2007–2014. URL: http://www.deutschestextarchiv.de/ (abgerufen am 10. 09. 2014).

[11] D. Bloomberg. *Leptonica*. 2001–2014. URL: http://www.leptonica.com/ (abgerufen am 29. 08. 2014).

[12] F. Blümm. *Halbautomatisches Training von OCR-Programmen für Frakturschriften*. Diplomarbeit. Julius-Maximilians-Universität Würzburg, 2008.

[13] T. Breuel. *The hOCR Embedded OCR Workflow and Output Format*. 2007. URL: https://docs.google.com/document/d/1QQnIQtvdAC_8n92-LhwPcjtAUFwBlzE8EWnKAxlgVf0/preview (abgerufen am 10. 09. 2014).

[14] R. C. Carrasco. *ocrevalUAtion*. Alicante, 2009–2014. URL: https://github.com/impactcentre/ocrevalUAtion (abgerufen am 23. 08. 2014).

[15] O. Chafik. *BridJ*. 2010–2014. URL: https://code.google.com/p/bridj/ (abgerufen am 10. 09. 2014).

[16] O. Chafik. *JNAerator*. 2008–2014. URL: https://code.google.com/p/jnaerator/ (abgerufen am 10. 09. 2014).

[17] F. Chang, C.-J. Chen und C.-J. Lu. „A linear-time component-labeling algorithm using contour tracing technique". In: *Computer Vision and Image Understanding* 93.2 (2004), S. 206–220. DOI: 10.1016/j.cviu.2003.09.002.

[18] M. Cheriet u. a. *Character recognition systems. A guide for students and practioners*. Wiley-Interscience, Hoboken, 2007.

[19] M. Federbusch und C. Polzin. *Volltext via OCR – Möglichkeiten und Grenzen. Testszenarien zu den Funeralschriften der Staatsbibliothek zu Berlin – Preußischer Kulturbesitz*. Bd. 43. Beiträge aus der Staatsbibliothek zu Berlin – Preußischer Kulturbesitz. Berlin, 2013.

[20] F. Forssman und R. de Jong. *Detailtypografie*. 3. Aufl. Hermann Schmidt, Mainz, 2008.

[21] L. Fries. *Würzburger Chronik. Geschichte, Namen, Geschlecht, Leben, Thaten und Absterben der Bischöfe von Würzburg und Herzoge zu Franken, auch was während der Regierung jedes Einzelnen derselben Merkwürdiges sich ereignet hat*. 2. Aufl. Bonitas-Bauer, Würzburg, 1924.

[22] L. Fries. *Würzburger Chronik. Geschichte, Namen, Geschlecht, Leben, Thaten und Absterben der Bischöfe von Würzburg und Herzoge zu Franken, auch was während der Regierung jedes Einzelnen derselben Merkwürdiges sich ereignet hat*. 3. Aufl. Bonitas-Bauer, Würzburg, 1961.

[23] B. Gatos, K. Ntirogiannis und I. Pratikakis. „ICDAR 2009 Document Image Binarization Contest (DIBCO 2009)". In: *10th International Conference on Document Analysis and Recognition*. 2009, S. 1375–1382. DOI: 10.1109/ICDAR.2009.246.

[24] P. Heidorn. *Blackletter Revival*. 2005. URL: http://www.moorstation.org/typoasis/blackletter/index.htm (abgerufen am 19. 08. 2014).

[25] G. Helzel. *Die „Normal-Fraktur"*. 1999. URL: http://www.romana-hamburg.de/Normalfraktur.pdf (abgerufen am 19. 08. 2014).

[26] W. Hendlmeier. *Schriftklassifikation. Die Stilgruppen der für die deutsche Sprache verwendeten Schriften*. 2013. URL: http://www.variatio-delectat.com/41Schriftenklassifikation.pdf (abgerufen am 19. 08. 2014).

[27] D. S. Hirschberg. „A linear space algorithm for computing maximal common subsequences". In: *Communications of the ACM* 18.6 (1975), S. 341–343. DOI: 10.1145/360825.360861.

[28] R. Holley. „How good can it get? Analysing and improving OCR accuracy in large scale historic newspaper digitisation programs". In: *D-Lib Magazine* 15.3/4 (2009).

[29] L.-K. Huang und M.-J. J. Wang. „Image thresholding by minimizing the measures of fuzziness". In: *Pattern Recognition* 28.1 (1995), S. 41–51. DOI: 10.1016/0031-3203(94)E0043-K.

[30] ImageMagick Studio. *ImageMagick*. 1990–2014. URL: http://www.imagemagick.org/ (abgerufen am 31. 08. 2014).

[31] S. Kimbal u. a. *The GIMP. The GNU Image Manipulation Program*. 1998–2014. URL: http://www.gimp.org/ (abgerufen am 31. 08. 2014).

[32] L. Lamport u. a. *LaTeX*. 1985–2014. URL: http://www.latex-project.org/ (abgerufen am 31. 08. 2014).

[33] V. I. Levenshtein. „Binary Codes Capable of Correcting Deletions, Insertions and Reversals". In: *Soviet Physics Doklady* 10.8 (1966), S. 707–710.

[34] G. Mälzer. „Die Würzburger Bischofs-Chronik des Lorenz Fries. Textzeugen und frühe Überlieferung". In: *Mainfränkische Hefte* 84 (1987), S. 1–64.

[35] Microsoft. *Microsoft Word*. 1983–2013. URL: http://office.microsoft.com/de-de/word/ (abgerufen am 31. 08. 2014).

[36] S. B. Needleman und C. D. Wunsch. „A general method applicable to the search for similarities in the amino acid sequence of two proteins". In: *Journal of Molecular Biology* 48.3 (1970), S. 443–453. DOI: 10.1016/0022-2836(70)90057-4.

[37] Q. Nguyen. *Tess4J*. 2010–2014. URL: http://tess4j.sourceforge.net/ (abgerufen am 10. 09. 2014).

[38] Q. Nguyen. *Tess4J Change Summary*. 2014. URL: http://tess4j.sourceforge.net/changelog.html (abgerufen am 10. 09. 2014).

[39] Q. Nguyen. *VietOCR. jTessBoxEditor*. 2008–2014. URL: http://vietocr.sourceforge.net/training.html (abgerufen am 31. 08. 2014).

[40] W. Niblack. *An introduction to digital image processing*. Prentice Hall, Englewood Cliffs, 1986.

[41] J. O'Regan. *deu-frak.traineddata.gz. German (Fraktur) language data for Tesseract (3.00 and up)*. 2010. URL: https://code.google.com/p/tesseract-ocr/downloads/detail?name=deu-frak.traineddata.gz (abgerufen am 15. 05. 2014).

[42] J. O'Regan und D. Eger. *COMBINE_TESSDATA(1) Manual Page*. 2010. URL: http://tesseract-ocr.googlecode.com/svn-history/r1147/trunk/doc/combine_tessdata.1.html (abgerufen am 08. 09. 2014).

[43] J. O'Regan und D. Eger. *UNICHARSET(5) Manual Page*. 2010. URL: http://tesseract-ocr.googlecode.com/svn-history/r1147/trunk/doc/unicharset.5.html (abgerufen am 07. 09. 2014).

[44] N. Otsu. „A Threshold Selection Method from Gray-Level Histograms". In: *IEEE Transactions on Systems, Man, and Cybernetics* 9.1 (1979), S. 62–66. DOI: 10.1109/TSMC.1979.4310076.

[45] Z. Podobný. *3rd Party. GUIs and Other Projects using Tesseract OCR*. 2012. URL: https://code.google.com/p/tesseract-ocr/wiki/3rdParty (abgerufen am 09. 09. 2014).

[46] Z. Podobný. *tesseract-ocr-3.02.frk.tar.gz. Frankish language data for Tesseract 3.02*. 2012. URL: https://code.google.com/p/tesseract-ocr/downloads/detail?name=tesseract-ocr-3.02.frk.tar.gz (abgerufen am 07. 09. 2014).

[47] S. V. Rice, F. R. Jenkins und T. A. Nartker. „The Fourth Annual Test of OCR Accuracy“. In: *Technical Report 95–04*. University of Nevada, Las Vegas, 1995, S. 1–39. URL: http://stephenvrice.com/images/AT-1995.pdf (abgerufen am 29. 08. 2014).

[48] P. Rück. „Paläographie und Ideologie. Die deutsche Schriftwissenschaft im Fraktur-Antiqua-Streit von 1871-1945“. In: *Signo. Revista de Historia de la Cultura Escrita* 1 (1994), S. 15–33. URL: http://hdl.handle.net/10017/7457 (abgerufen am 30. 08. 2014).

[49] J. Sauvola und M. Pietikäinen. „Adaptive document image binarization“. In: *Pattern Recognition* 33.2 (2000), S. 225–236. DOI: 10.1016/S0031-3203(99)00055-2.

[50] J. Schalansky. *Fraktur mon Amour*. 1. Aufl. Hermann Schmidt, Mainz, 2006.

[51] P. Selinger. *Potrace. A polygon-based tracing algorithm*. 2003. URL: http://potrace.sourceforge.net/potrace.pdf (abgerufen am 16. 08. 2014).

[52] R. Smith. „An Overview of the Tesseract OCR Engine“. In: *Ninth International Conference on Document Analysis and Recognition (ICDAR 2007) Vol 2*. 2007, S. 629–633. DOI: 10.1109/ICDAR.2007.4376991.

[53] R. Smith. *Everything you always wanted to know about Tesseract*. 2014. URL: https://drive.google.com/file/d/0B7l10Bj_LprhbUlIUFlCdGtDYkE (abgerufen am 06. 04. 2014).

[54] R. Smith. „Hybrid Page Layout Analysis via Tab-Stop Detection“. In: *Proceedings of the 10th international conference on document analysis and recognition*. 2009, S. 241–245. DOI: 10.1109/ICDAR.2009.257.

[55] R. Smith. *Tesseract Release Notes*. 2014. URL: https://code.google.com/p/tesseract-ocr/wiki/ReleaseNotes (abgerufen am 29. 08. 2014).

[56] R. Smith, Z. Podobný und N. White. *AddOns. External tools, wrappers and training projects for Tesseract*. 2008. URL: https://code.google.com/p/tesseract-ocr/wiki/AddOns (abgerufen am 06. 09. 2014).

[57] R. Smith, Z. Podobný und N. White. *Compiling. Compilation guide for various platforms*. 2012. URL: https://code.google.com/p/tesseract-ocr/wiki/Compiling (abgerufen am 08. 09. 2014).

[58] R. Smith u. a. *Training Tesseract 3. How to use the tools provided to train Tesseract 3 for a new language*. 2010. URL: https://code.google.com/p/tesseract-ocr/wiki/TrainingTesseract3 (abgerufen am 30. 08. 2014).

[59] T. F. Smith und M. S. Waterman. „Identification of common molecular subsequences“. In: *Journal of Molecular Biology* 147.1 (1981), S. 195–197. DOI: 10.1016/0022-2836(81)90087-5.

[60] D. Steffmann. *Weiß Rundgotisch*. 2004. URL: http://moorstation.org/typoasis/designers/steffmann/samples/w/weiss_rund.htm (abgerufen am 18. 08. 2014).

[61] M. Stokes u. a. *A Standard Default Color Space for the Internet – sRGB*. Hrsg. von World Wide Web Consortium. 1996. URL: http://www.w3.org/Graphics/Color/sRGB (abgerufen am 18. 08. 2014).

[62] M. Suzuki u. a. „INFTY“. In: *the 2003 ACM symposium*. Hrsg. von C. Roisin, E. V. Munson und C. Vanoirbeek. 2003, S. 95. DOI: 10.1145/958220.958239.

[63] Ø. D. Trier und A. K. Jain. „Goal-directed evaluation of binarization methods“. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 17.12 (1995), S. 1191–1201. DOI: 10.1109/34.476511.

[64] Universidad de Alicante. *Succeed. Support Action Centre of Competence in Digitisation*. 2013–2014. URL: http://www.succeed-project.eu/ (abgerufen am 06. 09. 2014).

[65] P. Vorbach. *Erstellung von TrueType-Fonts zu historischen Manuskripten*. Bachelorarbeit. Julius-Maximilians-Universität Würzburg, 2012.

[66] P. Vorbach. *How to build Tesseract 3.03 with Visual Studio 2013*. 2014. URL: http://vorba.ch/2014/tesseract-3.03-vs2013.html (abgerufen am 08. 09. 2014).

[67] P. Vorbach. *How to build Tesseract on Cygwin*. 2014. URL: http://vorba.ch/2014/tesseract-cygwin.html (abgerufen am 08. 09. 2014).

[68] P. Vorbach. *jleptonica*. 2014. URL: https://github.com/tesseract4java/jleptonica (abgerufen am 12. 09. 2014).

[69] P. Vorbach. *jtesseract*. 2014. URL: http://github.com/tesseract4java/jtesseract (abgerufen am 12. 09. 2014).

[70] P. Vorbach, Z. Podobný und N. White. *Add TessChoiceIterator + methods to C API*. 2014. URL: https://code.google.com/p/tesseract-ocr/issues/detail?id=1149 (abgerufen am 10. 09. 2014).

[71] K. H. Warkentin. „Klassifikation der Schriften nach DIN“. In: *Schrifttechnologie*. Hrsg. von P. Karow. Springer, Berlin und Heidelberg, 1992, S. 343–358. DOI: 10.1007/978-3-642-77160-6_16.

[72] M. Wermke u. a., Hrsg. *Duden. Die deutsche Rechtschreibung*. 22. Aufl. Bd. 1. Dudenverlag, Mannheim u. a., 2000.

[73] G. Williams. *FontForge*. 2000–2013. URL: http://fontforge.org/ (abgerufen am 31. 08. 2014).

[74] U. Zeidler. *Ligafaktur.de. Fraktur-/Ligatur-Programme und -Schriften*. 2011. URL: http://www.ligafaktur.de/Schriften.html (abgerufen am 07. 08. 2014).

Abbildungsverzeichnis

Tabellenverzeichnis