Bachelorarbeit

Präzise Erkennung von Sonderzeichen aus layout-basierten Textdokumenten am Beispiel der häufigsten europäischen Sonderzeichen

---

von Pascal Muckenhirn

28.Oktober.2019

Albert-Ludwigs-Universität Freiburg
Technische Fakultät

Professur für Algorithmen und Datenstrukturen
Prof. Dr. Hannah Bast

Betreuer: Claudius Korzen

Bearbeitungszeit

05. 08. 2019 – 28. 10. 2019

Prüferin

Prof. Dr. Hannah Bast

Betreuer

Claudius Korzen

Erklärung

Hiermit erkläre ich, dass ich diese Abschlussarbeit selbständig verfasst habe, keine anderen als die angegebenen Quellen/Hilfsmittel verwendet habe und alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten Schriften entnommen wurden, als solche kenntlich gemacht habe. Darüber hinaus erkläre ich, dass diese Abschlussarbeit nicht, auch nicht auszugsweise, bereits für eine andere Prüfung angefertigt wurde.

Münstertal, 28.10.2019	P. Muckenhirn
Ort, Datum	Unterschrift

Zusammenfassung

Das Ziel dieser Arbeit ist es, die Erkennung und Übersetzung von Sonderzeichen in layout-basierten Textdokumenten (hier: PDF) zu verbessern. Layout-basierte Textdokumente und insbesondere PDFs beinhalten einige Herausforderungen.

Zum einen enthalten sie Ligaturen. Bei Ligaturen werden mehrere Buchstaben, die aufeinander folgen, näher zusammengeschrieben. Dies wird allerdings nur bei bestimmten Buchstabenfolgen angewandt. Gründe hierfür können Stylistische oder Historische sein. Ein Beispiel für solch eine Ligatur ist das Zeichen ”ffi”. Aufgrund von Ligaturen ist es dem Nutzer nur möglich nach dem Wort ”efficient” zu suchen, wenn er den Unicode des Zeichens ”ffi” als Teil des Wortes eingibt. Dieses Zeichen ist auf einem Tastaturlayout nicht zu finden, sodass der Benutzer sich den Unicode anderweitig beschaffen muss. Außerdem enthalten diese Dokumente auch kombiniert diakritische Zeichen, welche zumeist eine bestimmte Betonung signalisieren sollen. Sie werden häufig als zwei Zeichen gespeichert und müssen deshalb wieder zusammengebaut werden. Die Frage hierbei ist allerdings, **wie** dies gemacht wird. Für das Beispiel ”á” würde hierbei das Zeichen ”a” als Basiszeichen und das Zeichen ”´” als diakritisches Zeichen gespeichert werden. Die größte Herausforderung ist allerdings, dass die Zeichen in bestimmten Fällen nur gezeichnet vorliegen. Hierbei ist also nur die Form des Zeichens und dessen Position zur Erkennung des Zeichens nutzbar.

Um diese Ziele zu erreichen wird eine Kombination aus 2 Ansätzen genutzt. Zum einen wird die Form des Zeichens und zum anderen die umliegenden Zeichen analysiert. Diese beiden Ansätze werden mittels 2 getrennten Machine-Learning-Modellen umgesetzt, die im finalen Ablauf zusammenarbeiten. Durch Training der Modelle auf insgesamt 20 verschiedenen Schriftlayouts, zu je 100 PDFs, wird ein großes Spektrum an Dokumenten abgedeckt. Die Evaluation zeigte, dass das finale (kombinierte) Modell mehr als viermal genauer als aktuelle Ansätze ist.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Algorithmenverzeichnis

1 Einführung

1.1 Motivation für das Thema

Jeder trifft täglich auf layout-basierte Textdokumente. Zu solchen Textdokumenten können unter anderem Bücher, wissenschaftliche Artikel oder auch Zeitungen zählen. Zumeist werden diese Dokumente in Form von PDF-Dokumenten angeboten. Im Rahmen dieser Arbeit wird dies das genutzte Dateiformat sein.
Layout-basierte Textdokumente sind hierbei Dokumente, welche nicht den eigentlichen (textuellen) Inhalt in Form eines Fließtextes beinhalten. Sie enthalten lediglich Informationen zu den einzelnen Zeichen, welche zu sehen sind. Auch Informationen wie diese Zeichen zusammenhängen sind nicht gespeichert. In PDFs wird die Speicherung mittels nummerierte Objekte gelöst. Diese Objekte können hierbei diverse Informationen, wie beispielsweise Informationen über die Schriftart, Farbdefinitionen oder auch verwendete Zeichen-Encodings enthalten [29].
Diese layout-basierten Textdokumente weisen außerdem eine Besonderheit auf, welche nicht sofort ins Auge sticht. Sie können unter anderem Ligaturen und kombinierte diakritische Zeichen enthalten.

Beispiele für Wörter mit Ligaturen und kombinierten diakritischen Zeichen sind:

efficient und créme brûlèe [5]

Abbildung 1: Beispiel für Ligatur und kombinierte diakritische Zeichen

Im linken Beispiel findet sich eine Ligatur. Diese ist durch eine Box um die Buchstaben ”ffi” gekennzeichnet. Im rechten Beispiel hingegen finden sich diverse kombinierte diakritische Zeichen, beispielsweise ”û”. Hierbei ist außerdem zu sehen, dass kombinierte diakritische Zeichen zumeist als 2 Zeichen erkannt werden. Dies hat den Grund, dass diese zumeist als 2 Zeichen im PDF-Dokument gespeichert sind.

Würde man die beiden Wörter ”unbehandelt” lassen, so wäre das Suchen dieser nur erschwert möglich. Dies ist der Fall, da für das Wort ”efficient” das Suchwort ”e[ffi]cient” eingegeben werden müsste. Hierbei müsste das Zeichen [ffi] mittels des Unicode eingegeben werden. Da dieses Zeichen aber nicht Teil des normalen Tastaturlayouts ist, ist hierbei ein erneuter Zeitaufwand für die Suche des passenden Unicodes notwendig. Dies ist der Fall, da Ligaturen in der PDF meistens mittels deren Unicode gespeichert sind. Würde der Benutzer außerdem versuchen das Wort aus der PDF zu kopieren, so ist damit zu rechnen, dass das Wort ”e?cient” oder etwas ähnliches kopiert wird. Jedoch nicht das eigentliche Wort.
Für kombiniert diakritische Zeichen würde beim kopieren des obigen Beispielswortes eine Ausgabe der Art ”cr´eme br^ul`ee” folgen. Dies ist der Fall, da kombiniert diakritische Zeichen, wie bereits erwähnt wurde, als 2 Zeichen gespeichert werden. Hierbei ist allerdings nicht eindeutig, welches der beiden Zeichen, die zusammen das kombiniert diakritische Zeichen bilden, zuerst kommt. Die Speicherung dieser kann von PDF zu PDF variieren.

Es gibt allerdings auch Fälle, in denen die Zeichen nur als gezeichnete Information vorliegen. Liegen Informationen nur gezeichnet vor, so ist der Unicode des Zeichens nicht hinterlegt. Es ist also nur möglich das Zeichen über seine Form und die Position des Zeichens zu bestimmen. Liegen nur solche Informationen vor, so kann man eigentlich nicht von einem Buchstaben sprechen.

Aus diesen Gründen ist dies ein sehr relevantes Problem. Zur Lösung des Problems werden im Laufe dieser Arbeit zwei Ansätze präsentiert. Zum einen ein Baseline-Algorithmus. Dieser arbeitet regelbasiert und versucht über die Unicodes der Zeichen das korrekte Ausgabewort zu konstruieren. Dies wird unter Nutzung der Unicodes, Abstände zwischen den einzelnen Zeichen und mögliche Kombinationen von Zeichen gemacht. Mit möglichen Kombinationen ist hierbei gemeint, dass nicht jeder Basisbuchstabe mit jedem diakritischen Zeichen verbunden werden kann.
Liegen diese Unicodes allerdings nicht vor, so ist es für den Baseline-Algorithmus nicht möglich das richtige Ausgabewort zu erzeugen. Aus diesen Gründen wird in Kapitel 4 ein Machine-Learning-Ansatz vorgestellt. Mit diesem ist es möglich auch diese Art von PDFs gut zu verarbeiten, da hierbei auch die Form der Zeichen analysiert werden.

Im Folgenden wird nun kurz erklärt welche Eigenschaften diese beiden Zeichentypen haben und wieso sie eingesetzt werden.

Unter Ligaturen versteht man Zeichen, welche aus mehreren Buchstaben bestehen. Diese werden hierbei ineinander geschrieben. Ligaturen können in 2 Gruppen unterteilt werden [28].
Die erste Gruppe bilden hierbei Ligaturen, welche im Laufe der Zeit Einzug in bestimmte Standard-Alphabete gehalten haben. Viele dieser Ligaturen sind aber im Laufe der Zeit auch wieder verschwunden, sodass heute nur noch einige wenige Teil der jeweiligen Sprache sind. Ein Beispiel hierfür ist das Zeichen "Æ". Heutzutage ist es noch Teil der Alphabete vieler nordischer Sprachen. Früher war dies jedoch auch ein viel genutztes Zeichen im Deutschen oder Englischen. Ein weiteres Beispiel, welches auch heute noch Teil der deutschen Sprache ist, ist das Zeichen "ß". Hierbei ist außerdem gut zu sehen, dass bei diesen sogenannten "historischen"-Ligaturen, gar nicht mehr so klar ersichtlich ist, dass es sich um eine Ligatur handelt. Grund hierfür ist, dass das Zeichen seit vielen Epochen Teil der Sprache ist und somit nicht mehr heraussticht. Genauso verhält es sich auch mit dem "Æ" in den nordischen Sprachen.
Außerdem gibt es sogenannte "stylistische"-Ligaturen. Diese bilden den größeren Teil der in heutigen Texten anzutreffenden Ligaturen. Bei "stylistischen"-Ligaturen treffen zwei oder mehr Buchstaben mit Oberlänge aufeinander. Der Begriff Oberlänge ist hierbei als der Bereich oberhalb der Mittellinie aller Buchstaben zu verstehen [25].
Abbildung 2 stellt dies für ein besseres Verständnis grafisch dar.

[26]

Abbildung 2: Typografisches Liniensystem

"Stylistische"-Ligaturen werden benutzt, um Lücken im Wort zu vermeiden. Den Eindruck einer Lücke würde der Leser bei dieser Aneinanderreihung von diesen Zeichen zwangsläufig bekommen. Dies kann zur Folge haben, dass der Lesefluss behindert wird. Beispiele für solche Ligaturen sind **fi** oder auch **ff** [24] [30].

Sowohl der Baseline-Algorithmus, als auch Machine-Learning-Ansatz, können die

häufigsten heute bekannten und verwendeten Ligaturen verarbeiten. Zu Beginn des Kapitel 3 werden die unterstützten Ligaturen des Baseline-Algorithmus aufgezeigt. In Kapitel 4 wird dasselbe für den Machine-Learning-Ansatz gemacht.

Unter kombinierten diakritischen Zeichen versteht man Zeichen, welche zum einen aus einem diakritischen Zeichen und zum anderen aus einem Basisbuchstaben bestehen. Zur Veranschaulichung ein kurzes Beispiel: Für das Zeichen "û" ist der Basisbuchstabe das "u" und "ˆ" ist das diakritische Zeichen.
Die diakritischen Zeichen sind hierbei nah am eigentlichen Buchstaben geschrieben, sodass eine eindeutige Zuordnung gegeben ist. Zum Teil berühren sie sogar den Buchstaben. Weitere Beispiele für kombinierte diakritische Zeichen sind **å**, **á** oder **ç**. Die diakritischen Zeichen können je nach zugehörigem Buchstaben oberhalb oder unterhalb des eigentlichen Buchstaben stehen, wobei eine leichte seitliche Versetzung auch möglich ist. Sie werden eingesetzt um eine besondere Sprechweise oder Betonung innerhalb eines Wortes zu definieren [23].

Diakritische Zeichen gibt es in vielen verschiedenen Alphabeten, so zum Beispiel im Arabischen, Griechischen, Koreanischen oder auch Hebräischen. Für diese Arbeit beschränke ich mich allerdings auf die lateinische Schrift.

Wie auch bei den Ligaturen, gibt es sehr viele verschiedene kombiniert diakritische Zeichen. Aus diesem Grund werden sowohl von der Baseline-Implementierung, als auch des Machine-Learning-Ansatzes nur die bekanntesten diakritischen Zeichen mit ihren Basisbuchstaben verarbeitet. Um welche Kombinationen es sich hierbei handelt, wird jeweils zu Beginn des jeweiligen Kapitels in einer Tabelle dargestellt [22].

Eine weitere Sache, mit welcher umgegangen werden muss, ist die Reihenfolge der Zeichen eines Wortes. Dies kann bei diakritischen Zeichen dazu führen, dass nicht klar ist, zu welchem Basisbuchstaben das diakritische Zeichen gehört. Dies ist der Fall, da es passieren kann das beispielsweise das Wort "Vendée" durch die Zeichenfolge "Vende´e" gespeichert ist. Hierbei muss nun entschieden werden, zu welchem "e" das diakritische Zeichen gehört.

1.2 Ziel der Arbeit

Ziel dieser Arbeit ist es, eine Verbesserung der Erkennung beziehungsweise Übersetzung von diesen Sonderzeichentypen zu erreichen. Dies soll sowohl für PDF-Dokumente, welche für Zeichen nur gezeichnete Informationen anbieten, aber auch für PDF-Dokumente, bei denen weitere Informationen über die Zeichen hinterlegt sind, erreicht werden. Vor allem der Teilbereich der PDF-Dokumente mit nur gezeichneten Zeichen stellt für die Programme, mit welchen mein Ansatz in Kapitel 5 verglichen wird, eine enorme Herausforderung dar. Dies gilt nicht nur für diese ausgewählten Programme, sondern ist ein allgemeines Problem.

Durch eine abschließende Evaluation, bei welcher auch aktuelle Werkzeuge getestet wurden, wird überprüft, ob dieses Ziel erreicht wurde.

1.3 Problemdefinition

Aus einem PDF-Dokument einen durchsuchbaren Text zu extrahieren ist ein sehr komplexer Prozess, der viele Schritte beinhaltet. Diese Schritte bauen hierbei aufeinander auf. In dieser Arbeit soll nur ein Teilschritt dieses komplexen Prozesses gelöst werden, nämlich die Übersetzung von Ligaturen und diakritischen Zeichen (zum Beispiel [ffi] in ”f”, ”f” und ”i” beziehungsweise ”`” und ”a” in ”à”). Der vorherige Schritt, im Gesamtprozess, ist die Erkennung von Wortgrenzen. Nach Übersetzung dieser Sonderzeichen folgen weitere Schritte, wie beispielsweise das Einordnen der Wörter in Textblöcke. Diese Schritt sind allerdings nicht Teil dieser Arbeit.

Die Eingabe dieses Teilschrittes sind die Wörter des PDF-Dokuments. Hierbei enthält jedes Wort die Position, Schriftart und Schriftgröße aller enthaltenen Zeichen. Diese Wörter können hierbei sowohl Ligaturen, als auch kombiniert diakritische Zeichen enthalten. Jede Ligatur besteht hierbei aus einem Zeichen und jedes kombiniert diakritsche Zeichen aus zwei Zeichen (also diakritisches Zeichen und Basisbuchstabe). Die Zeichen können hierbei entweder mittels deren Unicode, für das Zeichen [ffi] wäre dies \FB03, oder gezeichnet vorliegen. Liegen die Zeichen nur gezeichnet vor, so ist nur die Form des Zeichens und die Position bekannt. Für beide Sonderzeichentypen liegen außerdem immer Bilder der Zeichen vor.

Die Ausgabe ist eine Aneinanderreihung der Wörter. Diese Wörter sind alle durchsuchbar. Das bedeutet, die Ligaturen wurden getrennt und die diakritischen Zeichen wurden mit ihrem korrekten Basisbuchstaben verbunden. Dies soll auch für den ”gezeichneten” Fall gelten.

1.4 Aufbau der Arbeit

Im folgenden Kapitel wird erläutert wie verwandte Arbeiten ein ähnliches Problem lösen. In Kapitel 3 wird anschließend eine Baseline-Implementierung vorgestellt, welche für einen Spezialfall eine regelbasierte Lösung darstellt. Welche Bedingungen hierfür gelten müssen, werden darin genau erläutert. Darauf folgend wird in Kapitel 4 ein Machine-Learning-Ansatz gewählt, um die Problemstellung für den allgemeinen Fall zu lösen. Anschließend werden die beiden Ansätze evaluiert und somit dargelegt, ob die Ansätze erfolgreich waren. Daraus wird abschließend schlussgefolgert, wieso es zu diesen Ergebnissen gekommen ist.

2 Verwandte Arbeiten

Im folgenden Kapitel werden einige Arbeiten kurz dargestellt, welche im weitesten Sinne mit der Arbeit zu tun haben. Hierbei werden teilweise Sonderzeichen aus anderen Sprachen bearbeitet, allerdings kann die Denkweise beziehungsweise der Ansatz leicht auf jede Sonderzeichengruppe übertragen werden.

2.1 GROBID

Der Softwareansatz mit dem Titel "GROBID" [13], von Patrice Lopez, beschäftigt sich hauptsächlich mit Gewinnung von Metadaten aus bevorzugt wissenschaftlichen Arbeiten. Unter Metadaten versteht man hierbei beispielsweise den Titel, den Autor oder auch die Zusammenfassung (Abstract) eines Dokuments. Auch Volltexterkennung wird von GROBID unterstützt. Mit Volltexterkennung ist hierbei die Erkennung des eigentlichen Textes beziehungsweise Inhaltes des Dokuments gemeint. Die Ergebnisse, beispielsweise der Metadaten-Erkennung, werden als TEI-Datei ausgegeben. Hierbei handelt es sich um ein XML-ähnliches Format.

Bei GROBID handelt es sich um einen "Machine Learning"-Ansatz basierend auf Conditional Random Fields (CRF). CRF haben eine ähnliche Struktur wie Hidden-Markov-Models (HMM). Sie besitzen auch Zustandsübergänge, Zustandsgewichte und Start- beziehungsweise Endzustände [15]. Im Unterschied zu HMM können CRF allerdings jederzeit alle Informationen der gesamten Kette betrachten. [19] Weiter möchte ich nun hier allerdings nicht auf die Konstruktion von CRFs eingehen und verlinke deshalb für weitere Informationen auf die beiden bereits angegeben Quellen [19] [15].

Mit GROBID ist es unter anderem möglich PDFs zu verarbeiten, beziehungsweise deren Text auszulesen. Hierbei stellt der Dateityp "PDF", unter den unterstützen Dateitypen, eine Herausforderung für die Software dar. Durch stetige Weiterentwicklung erreicht GROBID mittlerweile ein Ergebnis von über 90 % bei Präzision und Trefferquote (engl. Recall) (für Metadaten-Erkennung). Außerdem läuft das Programm schnell, sodass laut Aussagen der Entwickler, bereits auf einem "Low-End-Rechner" mindestens 3 PDF-Dokumente pro Sekunde verarbeitet werden können [12].

GROBID unterstützt die häufigsten Ligaturen und kombiniert diakritischen Zeichen. Mit "Häufigsten" sind hierbei die am häufigsten benutzten Ligaturen beziehungsweise kombiniert diakritischen Zeichen gemeint. Jedoch bekommt die Software Probleme, sobald nur gezeichnete Informationen vorliegen. Dies wurde auch in der Evaluation in Kapitel 5 deutlich.

Im Vergleich mit anderen Softwareansätzen, welche sich auch auf die Metadaten-Erkennung spezialisiert haben, erreicht GROBID das beste Ergebnis (Stand 2013). Getestet wurden hierbei 7 verschiedene Ansätze. Sowohl bei der Erkennung der Autoren, Titel als auch bei der Erkennung weiterer Information schnitt GROBID im Vergleich am Besten ab. Die Evaluation wurde hierbei mit einer auf arXiv.org basierten Menge an Evaluationsdaten durchgeführt [14]. Auch aktuellere wissenschaftliche Texte zeigen, dass GROBID zu den aktuell besten Programmen in Sachen Textextraktion aus PDFs gehört [9].

Wie eben bereits kurz erwähnt, wurde GROBID im Rahmen dieser Arbeit außerdem für die Evaluation meines Ansatzes genutzt. Für das zweite Vergleichsprogramm der Evaluation, pdftotext, war leider kein wissenschaftlicher Artikel zu finden.

2.2 Diacritics Restoration Using Neural Networks

Die Arbeit mit dem Titel "Diacritics Restoration Using Neural Networks" [11], von Jakub Náplava, Milan Straka, Pavel Straňák und Jan Hajič, befasst sich mit einem Teilgebiet dieser Arbeit, den kombiniert diakritischen Zeichen. Hierbei wird allerdings eine andere Datengrundlage angenommen, als es in dieser Arbeit der

Fall ist. Die Arbeit versucht kombiniert diakritischen Zeichen zu rekonstruieren. Hiermit ist gemeint, dass das Eingabewort keine kombiniert diakritischen und auch keine diakritischen Zeichen enthält. Aufgrund der richtigen Schreibweise des Wortes sollte allerdings im Wort ein solches Zeichen vorkommen. Das bedeutet es wird versucht rein über den Kontext der Zeichen im Wort, also welche Zeichen aufeinander folgen, das richtige kombiniert diakritische Zeichen zu erkennen. Dieses kann am Ende entsprechend das falsche Zeichen ersetzen. Dieser Prozess wird als "diakritische Restoration" bezeichnet.

Die Nutzung des "Kontextes" der Zeichen im Wort wird auch in meiner Arbeit als ein Kriterium genutzt, allerdings wird als zweites Kriterium die Form des jeweiligen Zeichens genutzt. Durch Kombination der beiden Kriterien soll ein besseres Ergebnis erzielt werden. Weiteres hierzu wird in Kapitel 4 erklärt.

Die Eingabedaten basieren bei diesem Ansatz auf verschiedenen Dokumenten in welchen aus Encoding- oder auch Zeitgründen auf eine korrekte Schreibweise (mit diakritischen Zeichen) der Wörter verzichtet wurde. Aufgrund des nicht immer passenden Tastaturlayouts ist es nämlich mit zusätzlichem Aufwand verbunden bestimmte Zeichen korrekt einzutippen. Für die jeweilige Sprache sind hierfür zumeist besondere Tasten reserviert, wie im Deutschen für das "ö", "ä" oder "ü". Schreibt man allerdings in einer anderen Sprache, welche andere kombiniert diakritische Zeichen enthält, so kommt es öfter zu diesen Problemen.

Die Modellstruktur des Ansatzes hat als Herzstück eine bidirektionale RNN-Schicht integriert. Mittels dieser soll der Kontext des Wortes miteinbezogen werden.

Der Modell-Ansatz erreichte eine Genauigkeit von 98,665 % wobei hingegen der lexikografische Ansatz nur eine Genauigkeit von 94,02 % aufweist [11].

Angelehnt an die Idee dieses Ansatzes ist auch eins meiner Modelle konzipiert. Es beinhaltet auch eine bidirektionale Schicht, welche ebenso den Sinn hat den Gesamtkontext des Wortes zu nutzen. Genaueres hierzu wird in Abschnitt 4.5 erklärt.

2.3 Diacritics Recognition Based Urdu Nastalique OCR System

Die Arbeit "Diacritics Recognition Based Urdu Nastalique OCR System" [6], von Ali Javed und Saima Nazir, fokussiert sich auf das Teilgebiet der Sprache "Urdu". Sie ist die Landessprache Pakistans und besteht aus diversen kombiniert diakritischen Zeichen und Ligaturen. Insgesamt umfasst sie 41 Zeichen. Die Schwierigkeit in der Erkennung der korrekten Wörter besteht darin, dass Zeichen verschiedene Formen annehmen können, je nachdem wie sie verbunden sind. Außerdem gibt es diverse Sonderregeln in Bezug auf Punkte in der Sprache.

Der Ansatz beruht auf einer OCR-Software, welche Teil der Gesamtkonstruktion ist. Der Ablauf der Datenverarbeitung ist kurz zusammengefasst der Folgende: Zuerst wird die Eingabe, welche zumeist aus einem eingescannten Dokument besteht, in ein binäres Bild übersetzt. Ein binäres Bild ist ein Bild welches ausschließlich aus den beiden Farben weiß und schwarz besteht. Bei der Umwandlung in ein solches Bild wird jeder Pixel entweder in einen weißen oder schwarzen Pixel umgewandelt, sodass allerdings alle Konturen und Umrisse erhalten bleiben. Anschließend wird die OCR-Software auf das binäre Bild angewandt. Auf diesen Schritt folgen einige weitere Schritte in welchen die Daten entsprechend ihrer Position in der Zeile und den Zeilen untereinander markiert werden. Abschließend werden die Zeichen, welche zusammengehören, entsprechend verbunden. Dies wird mittels einer sogenannten "Correlation Method" gemacht.

Der Ansatz erreicht, je nach Art der Zeichen, Ergebnisse von durchschnittlich rund 97 %.

2.4 Recognition of Printed Urdu Ligatures using Convolutional Neural Networks

Die Arbeit mit dem Titel "Recognition of Printed Urdu Ligatures using Convolutional Neural Networks" [10], von Israr Uddin, Nizwa Javed, Imran Siddiqi, Shehzad Khalid und Khurram Khurshid, beschäftigt sich mit der selben Problemstellung wie Abschnitt 2.3. Hierbei wurde allerdings ein etwas anderer Ansatz

gewählt. Der Ansatz basiert auf einem "Convolutional Neural Network"(CNN). Dieses wurde trainiert um zum einen Ligaturen und zum anderen die dazu kombinierten diakritischen Zeichen zu erkennen. Das Training des Modells fand hierbei mit Bildern des CLI und UPTI Datensatzes statt.

Die Architektur der Konstruktion ist folgendermaßen aufgebaut. Zuerst wird das Bild in primäre und sekundäre Zeichen zerlegt. Der primäre Teil ist hierbei das "Grundzeichen", welches zumeist eine Ligatur ist, und der sekundäre Teil entspricht den diakritischen Zeichen. Hierbei kann der sekundäre Teil auch mehrere Zeichen beinhalten. Die Zerlegung aller Zeichen in die beiden Teile basiert auf mehreren Eigenschaften. Hierzu zählen beispielsweise die Position relativ zur Basislinie, die Größe oder auch die Form. Die Basislinie wird hierbei mittels der "horizontal projection profile method" berechnet. Hierbei wird als Basislinie der y-Wert genommen, welcher die meisten Datenpunkte aufweist, bezogen auf die Eingabe einer Linie.

Der Unterteilungsprozess in primäre und sekundäre Zeichen wird regelbasiert gelöst. Anschließend werden alle Zeichen separat mittels des CNN vorhergesagt und anschließend in der Nachverarbeitung über eine "Look-up table" passend zusammengefügt. Hierbei wird als weiterer Parameter für ein besseres Ergebnis außerdem die Position relativ zur Basislinie genutzt.

Dieser Ansatz führt im Vergleich zu anderen, vergleichbaren, Ansätzen zu einer Steigerung der Genauigkeit auf bis zu 98,30 % je nach gewähltem Datensatz [10].

2.5 Attentive Sequence-to-Sequence Learning for Diacritic Restoration of Yorùbá Language Text

Die Arbeit "Attentive Sequence-to-Sequence Learning for Diacritic Restoration of Yorùbá Language Text" [18], von Iroro Orife, befasst sich mit der Sprache "Yorùbá". Diese Sprache ist in Westafrika sehr verbreitet und wird von mehr als 40 Millionen Menschen weltweit gesprochen. Sie wird allerdings aufgrund von einer nur limitierten Unterstützung der Sprache meist in ASCII geschrieben. Dies führt allerdings meist zu Unklarheiten, wenn aus dem Kontext nicht eindeutig klar ist, welches Wort gemeint ist. Ein in der Ausarbeitung aufgeführtes Beispiel hierfür

wäre gbà (verbreiten), gba (akzeptieren), gbá (schlagen).

In der Arbeit wird ein ”Sequence-to-Sequence”-Ansatz mittels zwei verschiedenen Modellarchitekturen beschrieben. Zum einen ein Modellkonstrukt basierend auf bidirektionalen RNNs welche zu einem RNN-Encoder-Decoder-Modell zusammengesetzt werden. Und zum anderen ein Ansatz mittels mehrerer ”self-attention”-Schichten. Diese Schichten bilden die Eingabesequenz auf eine interne Sequenz derselben Länge ab. Dies wird mittels einer linearen Umformung in sogenannte Abfrage, Schlüssel-, und Werte-Matrizen umgesetzt. Anschließend wird daraus die Ausgabe des Modells bestimmt. Dies geschieht durch ein Kombinieren und Aufsummieren der entsprechenden Werte und Matrizen.

Diese beiden Ansätze wurden mittels verschiedener Größeneinstellungen beziehungsweise verschiedene Anzahlen von Schichten trainiert. Anschließend wurde eine Evaluation der Genauigkeit (Accuracy) durchgeführt. Die besten Resultate der Evaluation lagen im Bereich von 96 % bis 98,5 %. Wie in der Arbeit allerdings erwähnt wurde, könnte dieser Wert noch gesteigert werden, indem die Modelle mit 100 % korrekten Daten trainiert werden würden. Dies war während des Trainings allerdings nicht der Fall [18].

3 Baseline-Ansatz

3.1 Idee des Baseline-Ansatzes & Herangehensweise an die Problemstellung

Die Hauptidee des Baseline-Algorithmus ist es, die einfachste Methode zu implementieren, um aus den gegebenen Rohdaten entweder durchsuchbare Zeichenfolgen oder die originalen Zeichenfolgen zu generieren. Mit ”original” ist hierbei folgendes gemeint: Ligaturen werden in ihre eigentlichen Buchstaben zerlegt. Für das Beispiel mit der Ligatur ”ffi” also in die Buchstaben ”f”, ”f” und ”i”. Bei diakritischen Buchstaben ist mit ”originalen” Buchstaben gemeint, dass sie mit ihrem Basisbuchstaben verbunden als kombiniert diakritisches Zeichen gespeichert werden. Für das Beispiel ”ˇc” wäre dies das Zeichen ”č”.

Für PDFs, welche nur gezeichnete Informationen für Sonderzeichen gespeichert haben, kann der Baseline-Ansatz nicht angewandt werden. Hierauf wird im folgenden noch einmal genauer eingegangen.

Bei ”normalen” PDFs hingegen sind Ligaturen und diakritische Zeichen meist mittels deren Unicode gespeichert, sodass der Baseline-Algorithmus angewandt werden kann.
Für Ligaturen gibt es nur eine Möglichkeit sie zu verarbeiten beziehungsweise zu übersetzen. Sie werden entsprechend des Beispiels oben getrennt und die Zeichen werden einzeln dem Wort anstatt des Unicodes angehängt. Dadurch wird das Ziel der Durchsuchbarkeit für Ligaturen erreicht.

Für kombinierte diakritische Zeichen ist es hingegen möglich diese in zwei verschiedene Formen zu überführen. Zum einen in die Form des Originalbuchstabens, also als kombiniert diakritisches Zeichen. Diese Form ist die korrekte, weist allerdings

den Nachteil auf, dass das Suchen erschwert ist. Bei exakter Suche muss, um ein Wort zu finden das korrekte Zeichen, als Teil des Wortes, eingegeben werden. Dies ist allerdings aufgrund des allgemeinen deutschen und englischen Tastaturlayouts nicht immer ohne Schwierigkeiten möglich. Ein Beispiel hierfür ist das Zeichen č. Die Schwierigkeit ist hierbei, dass das diakritische Zeichen "ˇ" nicht auf den beiden erwähnten Tastaturlayouts zu finden ist, sodass der Unicode anderweitig besorgt werden muss.
Die zweite Variante ist das kombiniert diakritische Zeichen in einen suchbaren Buchstaben zu transformieren. Diese Variante hat den Vorteil, dass eine Suche leichter möglich ist. Der Nachteil hierbei ist allerdings, dass nicht der eigentliche Buchstabe beziehungsweise das eigentliche Wort angezeigt wird.

Der Baseline-Ansatz wurden in Python entwickelt. Für den Baseline-Ansatz nutzte ich unter anderem das Python-Paket "unicodedata" [1]. Mittels dieses ist es möglich für jeden Unicode den Namen des Zeichens nachzuschlagen. So ist der Name des Unicodes \u0050 zum Beispiel "LATIN CAPITAL LETTER P". Mittels dieser Denkweise wurde der Ansatz für die Übersetzung von Ligaturen implementiert. In Abschnitt 3.4 wird hierauf noch einmal genauer eingegangen. Für den Teilbereich der kombinierten diakritischen Zeichen konnte ich erneut unicodedata nutzen. Dieses Mal allerdings, um das diakritische Zeichen mit dem Basisbuchstaben zu verbinden. Dies ist hierbei mittels der Methode "lookup" möglich. Hierbei wird der Basisbuchstabe und das diakritische Zeichen in ein vorgefertigtes Muster eingesetzt und der "lookup"-Funktion übergeben. Als Rückgabe erhält man anschließend das kombiniert diakritische Zeichen. Das Finden des passenden Basisbuchstabens, zum diakritischen Zeichen, ist hierbei nicht immer eindeutig. Deshalb entschied ich mich für eine Kombination aus zwei Metriken, um eine hohe Sicherheit zu haben, den korrekten Basisbuchstaben ausgewählt zu haben. Diese Metriken werden in Abschnitt 3.5 genauer erklärt.

3.2 Format der Eingabedaten

Für die Verwendung des Baseline-Ansatzes werden aus den eigentlichen PDFs erstellte "Ground-Truth"-Dateien genutzt. Diese Dateien wurden mittels einer Software des Lehrstuhls generiert. Die Software arbeitet hierbei zum einen mit der eigentlichen PDF und zum anderen mit dem dazugehörigen TeX-Quelltext. Durch

eine raffinierte Farbcodierung jedes einzelnen Wortes, welche sich als Farbverlauf über den gesamten Text zeigt, ist ein Verknüpfen der Zeichen möglich. Es ist also möglich die Zeichen der PDF zu Wörtern zusammenzubauen und unter anderem auch die korrekte Übersetzung, mittels des TeX-Quelltextes, als Ausgabe zu verknüpfen. Hierbei werden zum einen die korrekten Zeichen des TeX-Quellcodes mit den Unicodes beziehungsweise zerteilten Zeichen, im Falle eines kombiniert diakritischen Zeichens, des PDF-Dokuments zu einer "Ground-Truth"-Datei verbunden. Mittels dieser Datei kann anschließend der Baseline-Algorithmus arbeiten. In Abbildung 5 ist ein Auszug solch einer Datei aufgeführt.

Außerdem wurden für alle Sonderzeichen die Bilder der jeweiligen Zeichen in einem gleichnamigen Ordner gespeichert und in der "Ground-Truth"-Datei verlinkt, sodass eine einfache Zuordnung von Sonderzeichen zu Bild möglich ist.

Die Dateien stellen einige Parameter zu Verfügung, welche für die Nutzung der Ansätze sehr wichtig sind. Unter anderem ist für jedes Zeichen die Position, das Zeichen selbst und auf welcher Seite der PDF das Zeichen steht, gespeichert. Die Positionen sind hierbei entsprechend dem Koordinatensystem in Abbildung 3 definiert.

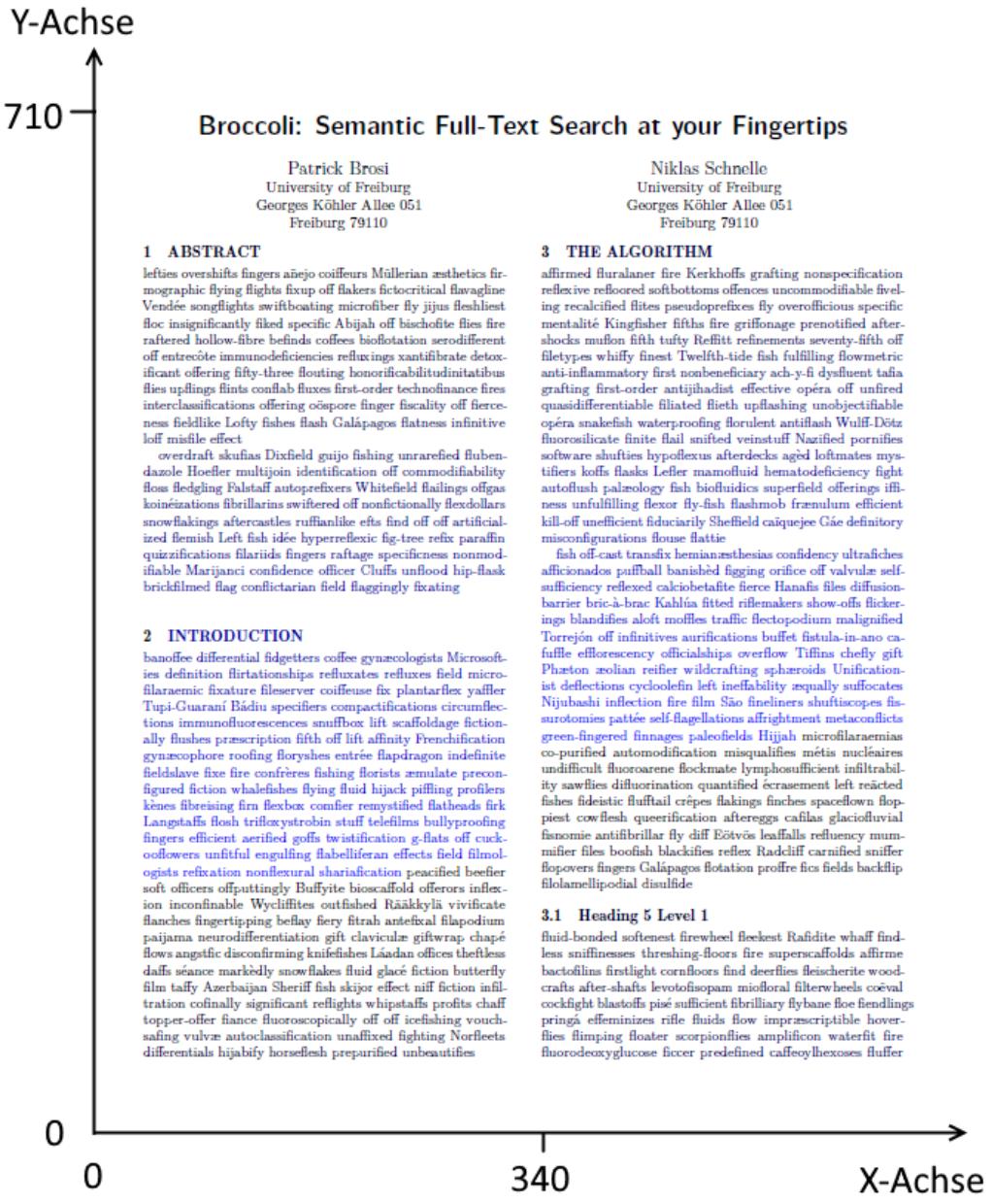

Abbildung 3: Koordinatensystem eines PDF-Dokuments

Die Position jedes Zeichens ist hierbei mittels einer umgebenden Box gespeichert. Das bedeutet, dass jedes Zeichen sowohl in X- als auch Y-Richtung einen Minimal- und Maximalwert gespeichert hat. Zu sehen sind diese Boxen in Abbildung 4. Für

die Box um das Zeichen ”t” wurden außerdem die Minimum- und Maximumwerte eingezeichnet. Die anderen Boxen besitzen diese auch, jedoch wäre das Bild überladen, wenn diese auch eingezeichnet wären.

[5]

Abbildung 4: Umgebende Box eines Zeichens

Des Weiteren ist jedes Sonderzeichen mittels des Unicodes dargestellt, sofern es sich nicht um gezeichnete PDFs handelt. Kombiniert diakritische Zeichen sind wie auch in der PDF als zwei getrennte Zeichen gespeichert. Wie eben schon erwähnt sind außerdem für alle Sonderzeichen Bilder verlinkt. Sowohl bei Ligaturen, als auch kombiniert diakritischen Zeichen handelt es sich hierbei um ein Bild für das komplette Zeichen. Für kombiniert diakritische Zeichen liegen also nicht 2 Bilder für das diakritische Zeichen und das Basisbuchstaben vor, sondern 1 Bild, welches beide Zeichen beinhaltet.

Die Dateien enthalten außerdem für jedes Wort die korrekte ”Übersetzung”. Diese wird für beide Ansätze benötigt, um zu vergleichen, ob die Vorhersage des jeweiligen Ansatzes korrekt war.

```
...
{
    "text": "entrecôte",
    "positions": [{
        "pageNum": 1,
        "minX": 66.0,
        "minY": 547.5,
        "maxX": 101.8,
        "maxY": 553.8
    }],
    "characters": [ ...
    {
        "text": "o^",
        "image": "0ilaCrw/qAYPEEqQ.jpg",
        "parts": [{
            "text": "o",
            ...
            "pageNum: 1,
            "minX": 89.8,
            "minY": 547.5,
            "maxX": 94.3,
            "maxY": 551.6
          }, {
            "text": "^",
            ...
            "pageNum: 1,
            "minX": 89.9,
            "minY": 552.2,
            "maxX": 94.4,
            "maxY": 553.8
          }]
    }, ... ]
}
...
```

Abbildung 5: Formatierung der Eingabedaten

3.3 Unterstütze Ligaturen und kombinierte diakritische Zeichen

In der Tabelle 1 sind alle Ligaturen aufgeführt, welche übersetzt beziehungsweise verarbeitet werden.

Schreibweise mit Ligatur	Schreibweise ohne Ligatur	Unicode
Aa, aa	AA, aa	U+A732, U+A733
Æ, æ	Æ, æ	U+00C6, U+00E6
Ao, ao	AO, ao	U+A734, U+A735
Au, au	AU, au	U+A736, U+A737
Av, av	AV, av	U+A738, U+A739
Av, av	AV, av (mit Balken)	U+A73A, U+A73B
Ay, ay	AY, ay	U+A73C, U+A73D
et	et	U+1F670
ff	ff	U+FB00
ffi	ffi	U+FB03
ffl	ffl	U+FB04
fi	fi	U+FB01
fl	fl	U+FB02
Œ, œ	Œ, œ	U+0152, U+0153
Oo, oo	OO, oo	U+A74E, U+A74F
ß	ſz	U+00DF
st	st	U+FB06
ft	ft	U+FB05
Ꜩ, ꜩ	TZ, tz	U+A728, U+A729
ue	ue	U+1D6B
Vy, vy	VY, vy	U+A760, U+A761
db	db	U+0238
dz	dz	U+02A3
dz	dz curl	U+02A5
dʒ	dʒ(or dezh)	U+02A4
fŋ	fŋ(or feng)	U+02A9
ls	ls (or less)	U+02AA
lz	lz	U+02AB

ɮ	lʒ(or lezh)	U+026E
ʠ	qp	U+0239
ʦ	tc curl	U+02A8
ʦ	ts (or tess)	U+02A6
ʧ	tesh	U+02A7
ᵯ	turned ui	U+AB51
ᵫ	ui	U+AB50

Tabelle 1: Unterstützte Ligaturen des Baseline-Ansatzes

Wichtig zu beachten ist hierbei außerdem, dass die Zeichen Æ, æ, Œ und œ nicht getrennt werden. Dies hat den Grund, dass die Trennung bei diesen Zeichen nicht eindeutig festgelegt ist. Dies ist der Fall, da diese Zeichen historisch gewachsen sind und mittlerweile, vor allem in den nordischen Sprachen, als normale Buchstaben angesehen werden.

In Tabelle 2 sind die unterstützten diakritischen Zeichen aufgeführt. Für jedes diakritische Zeichen werden alle Varianten in Bezug auf lateinische Schrift unterstützt. Zur Veranschaulichung des diakritischen Zeichens verbunden mit einem Buchstaben wird der Buchstabe o, beziehungsweise c für den Fall mit Cedilla und a für den Fall mit Ring darunter, im Folgenden als Platzhalter genutzt.

Beispiel eines kombiniert diakritischen Zeichens	Name des Zeichens	Unicode
ó	Acute	U+0301
ò	Grave	U+0300
ô	Circumflex	U+0302
ǒ	Caron	U+030C
ő	Double Acute	U+030B
ȍ	Double Grave	U+03OF
õ	Tilde	U+0303
ȯ	Overdot	U+0307
ọ	Underdot	U+0323

ö	Double dot/Diaeresis	U+0308
ŏ	Breve	U+0306
ô	Inverted Breve	U+0311
ō	Macron	U+0304
o̲	Underbar	U+0332
ạ	Ring below	U+0325
o̊	Ring above	U+030A
ç	Cedilla	U+0327

Tabelle 2: Unterstützte diakritische Zeichen des Baseline-Ansatzes

3.4 Implementierung

Der Baseline-Algorithmus besteht im Wesentlichen aus zwei getrennten Algorithmen, welche allerdings im gesamten Ablauf nacheinander ausgeführt werden. Zum einen die Verarbeitung von Ligaturen und zum anderen die Verarbeitung von diakritischen Zeichen. Hierbei ist die Reihenfolge wichtig. Wieso eine Reihenfolge sinnvoller als die Andere ist, wird in Abschnitt 3.5 genauer erklärt.

Die Verarbeitung eines Wortes geschieht, in dem zuerst die diakritischen Zeichen zu kombiniert diakritischen Zeichen verarbeitet werden. Im zweiten Schritt wird jedes Zeichen dem Algorithmus 1 übergeben und anschließend das Ergebnis dem Gesamtergebnis angehängt. Falls es sich bei einem Zeichen um eine Ligatur handelt, so wird dieses getrennt und zurückgegeben. Handelt es sich nicht um eine Ligatur, so wird das an die Funktion übergebene Zeichen, ohne jede Veränderung zurückgegeben.

Im Folgenden werden diese Zwischenschritte an einem Beispiel gezeigt. Hierfür wird das Beispielwort ”d´efi” genutzt, welches so in den Trainingsdateien stehen würde. Zu sehen ist hierbei erneut die geteilte Speicherung von kombiniert diakritischen Zeichen.

- ”défi” nach der Verarbeitung der diakritischen Zeichen

- ”défi” nach Teilen der Ligaturen

Nun werden die beiden Algorithmen erklärt, die für diese Verarbeitung verwendet werden.

3.4.1 Verarbeitung von Ligaturen

Die Verarbeitung von Ligaturen gestaltet sich mithilfe des unicodedata Pakets einfach. In Algorithmus 1 ist der Ablauf des Algorithmus mittels einer Mischung aus Code und Pseudocode beschrieben.

Algorithm 1 SplitLigatures

```
1:  procedure SplitLigatures(character)
2:      decode = unicodedata.name(character)
3:      if character is a diacritica then
4:          return character
5:      if decode == ¨SCRIPT LIGATURE ET ORNAMENT¨ then
6:                                                          ▷ Special case
7:          return ¨et¨
8:      if ”ae” in decode.lower() or ”oe” in decode.lower() then
9:                                      ▷ Special case for not splitting æ  and œ
10:         return character
11:     m = ligRe.match(decode)                          ▷ ligRe is a regex pattern
12:     if No match found then
13:         return character
14:     else
15:         retVal = m.group(3).lower()
16:         Clean up retVal                                  ▷ To serve special cases
17:         if m.group(1) == Capital” then
18:             retVal = retVal.title()
19:     return retVal
```

Zu Beginn wird mittels des unicodedata Pakets der Name des Zeichens ermittelt. Für den Unicode \uFB01 ist dies zum Beispiel ”LATIN SMALL LIGATURE FI”. Im Folgenden werden einige Spezialfälle, sowie das Nicht-behandeln eines diakritischen Zeichens abgearbeitet. In Zeile 11 wird hierbei das folgende Regex-Muster

angewandt:

re.compile(r'LATIN CS LL ([A-Z]{2,})') [2]

wobei **CS** = (CAPITAL|SMALL) und **LL** = (LIGATURE|LETTER)

Die Anwendung eines Regex-Musters ist möglich, da die Namen der Zeichen, mit Ausnahme von Spezialfällen, immer denselben Aufbau haben. Dies ist gut am Beispiel einer Ligatur zu sehen: ”fi” wird mittels unicodedata auf ”LATIN SMALL LIGATURE FI” abgebildet.
Die Eigenschaft wird im Folgenden außerdem genutzt, um die geteilten Buchstaben zu bestimmen. Diese sind immer Teil des Namens des Zeichen. Anschließend werden noch nicht benötigte Informationen, wie zum Beispiel ”WITH HORIZONTAL BAR” oder ähnliches entfernt und die richtige Größe gewählt, sowie das Ergebnis zurückgegeben. Mit ”richtiger Größe” ist hierbei groß- oder kleingeschrieben gemeint.

Die Laufzeit des Algorithmus liegt für m Wörter, wobei ein Wort maximal n Zeichen lang ist, im Worst Case bei $\mathbf{O(m \cdot n)}$. Dies ist der Fall, da der Algorithmus für ein Zeichen in konstanter Zeit durchläuft, da alle Operationen auf ”decode” basieren. ”decode” hat hierbei für jedes Zeichen eine annähernd identische Länge. Daraus folgt, dass jeder Durchlauf unabhängig vom Zeichen annähernd identische, konstante, Laufzeit besitzt. Für Spezialfälle ist die Laufzeit geringer, da hier kein ”match” mit dem Regex-Muster berechnet werden muss. Unter Spezialfälle fallen hierbei die Zeichen ”SCRIPT LIGATURE ET ORNAMENT”, Æ, æ, Œ, œund alle nicht Ligaturen.

Die Verarbeitung von Ligaturen könnte auch mittels eines Hash-Ansatzes gelöst werden. Dies bedeutet, dass zuerst eine große Hashmap angelegt werden müsste, in welcher jeweils als Schlüssel der Unicode und als Wert das korrekte Zeichen gespeichert ist. Mittels dieser wäre es auch möglich die Ligaturen ”umzuwandeln”.

3.4.2 Verarbeitung von diakritischen Zeichen

Die Verarbeitung von diakritischen Zeichen gestaltet sich deutlich schwieriger als die Verarbeitung von Ligaturen. Eine der größten Herausforderungen ist es den korrekten Partner für das Verbinden von diakritischen Zeichen mit dem passenden Buchstaben zu finden. Oftmals gibt es hierfür mehrere Möglichkeiten und es muss entschieden werden, welche die Sinnvollste ist. Dies ist der Fall, da die Reihenfolge nicht gleich ist. Hiermit ist gemeint, dass beispielsweise das diakritische Zeichen nicht immer vor dem Basisbuchstaben steht. Es ist sogar möglich das die beiden Buchstaben, also diakritisches Zeichen und Basisbuchstabe, nicht hintereinander aufgeführt sind.

Die Ausgabe sollte allerdings alle Buchstaben in der korrekten Reihenfolge enthalten. Aus diesem Grund ist es notwendig, die Reihenfolge zu überprüfen und gegebenenfalls zu korrigieren.
In Algorithmus 2 ist der Ablauf des Algorithmus erneut mittels einer Mischung aus Code und Pseudocode beschrieben. Aufgrund der Größe der Funktion ist dieser stark verkürzt. Für Details verweise ich deshalb an dieser Stelle auf den Programmcode. Hierbei ist außerdem der Aufruf einer Funktion für die Sortierung eingebunden, zu sehen in Algorithmus 3.

Algorithm 2 TranslateDiacritica

```
1: procedure TranslateDiacritica(word, positions, toOriginalCharacter)
2:     retWordChar = []            ▷ Helper lists for building the word at the end
3:     retWordXPos = []
4:     retWordYPos = []
5:     if toOriginalCharacter == False then
6:         result = ""
7:         for characters in word do
8:             Add the character and it's position to the list
9:     else
10:        for characters in word do
11:            if Character is diacritica then
12:                    ▷ Contains "COMBINING" in the name of the character
13:                for characters in word do
14:                    Calculate the X-Overlay and Y-Difference of the character
                       to the diacritica and save the best score and best matching
                       character
15:                Serve some special cases
16:                Lookup the unicode of the diacritica combined with the best
                   matching character
17:                if Was the best match character already added? then
18:                    Remove it from the result list
19:                else
20:                    Block the adding of the best match
21:                Add the combined character and it's position to the list
22:            else
23:                Add the character and it's position to the list
24:    Call function SortWord with the "current word" and it's positions
25:    return word
```

Algorithm 3 SortWord

```
1:  procedure SortWord(word, positions)
2:      Sort the positions after their y-position; Other lists accordingly
3:      numLines, clusters = kMeans.findBestK(retWordYPos)
4:      overPage = False
5:      if Word goes over two pages then
6:          overPage = True
7:      if overPage then
8:          start = 0
9:          end = numLines
10:         step = 1
11:     else
12:         start = numLines - 1
13:         end = -1
14:         step = -1
15:     Build up the final word via a for loop over the characters by using
        the start, end and step variable as settings.
16:     return sorted word
```

Wenn der Benutzer eine leicht durchsuchbare Buchstabenfolgen anfordert, so ist die Verarbeitung einfach. Es müssen lediglich die diakritischen Zeichen ignoriert werden. Alle anderen Buchstaben werden dem Ergebnis angehängt und entsprechend sortiert. Damit ist die Verarbeitung bereits abgeschlossen.

Falls der Benutzer jedoch die korrekte Übersetzung möchte, so ist die Verarbeitung aufwendiger. Hierbei wird über alle Zeichen iteriert. Handelt es sich bei dem Zeichen nicht um ein diakritisches Zeichen, so wird es direkt an die "retWordChar"-Liste angehängt. Handelt es sich jedoch um ein diakritisches Zeichen, so muss der passende Basisbuchstabe ermittelt werden, mit dem es kombiniert werden soll. Zuerst werden hierzu die X-Überlagerungs- und Y-Abstandswerte von jedem Basisbuchstaben zum diakritischen Zeichen ermittelt. Dies geschieht mithilfe der Position, welche jedes Zeichen aufweist. Mittels dieser Werte ist es anschließend leicht möglich den am besten passenden Buchstaben zu finden. Mithilfe des unicodedata Pakets wird aus dem Basisbuchstaben und dem diakritischen Zeichen nun der korrekte kombinierte diakritische Buchstabe "erstellt". Dies geschieht mit der in Abschnitt 3.1 bereits erwähnten "lookup"-Funktion des unicodedata-Pakets und eines vorgefertigten Musters. Anschließend sind noch einige Anpassungen an der "retWordChar"-Liste notwendig, sodass kein Buchstabe doppelt hinzugefügt wird

oder vergessen wird. Dies ist nötig, da es vorkommen kann, dass das diakritische Zeichen mit einem Basisbuchstaben kombiniert wurde, der bereits der Ausgabe hinzugefügt wurde. In diesem Fall sollte dieser wieder aus der Ausgabe entfernt werden. Der gleiche Fall ist auch in die andere Richtung möglich, also das nicht anhängen eines Basisbuchstabens an die Ausgabe, der noch folgt.
Ist dies für alle Buchstaben erledigt, müssen die Buchstaben nur noch sortiert und zurückgegeben werden. Die Sortierung wird hierbei durch den Algorithmus 4 erleichtert, sodass lediglich über die Cluster, welche hier die Linien symbolisieren, iteriert werden muss. Diese Cluster werden vom Algorithmus 4 zurückgegeben für die Eingabe aller Minimum-Y-Position der Zeichen. Die Cluster enthalten hier dann die Zeichen des Wortes, welche in der jeweiligen Zeile des eigentlichen PDFs stehen. Für das Wort "aftershocks" welches durch seine Position in der Linie auf 2 Linien verteilt steht, würde durch die beiden Cluster "[[a, f, t, e, r, -], [s, h, o, c, k, s]]"dargestellt. Zu sehen ist hier auch, dass die Cluster als Liste von Listen gespeichert werden. Anschließend muss noch geprüft werden, ob das Wort möglicherweise über mehrere Seiten verläuft. Ist dies der Fall, so wird rückwärts über die Cluster iteriert um das Wort in korrekter Reihenfolge auszugeben. Dies ist der Fall, da das Koordinatensystem jeder Seite in der unteren linken Ecke ihren Ursprung hat. Pro Linie wird dann noch nach der X-Position sortiert und schon kann das gesamte Wort in der korrekten Reihenfolge zusammengebaut werden.

Die Laufzeit von Algorithmus 2 ist hierbei im Worst Case $\mathbf{O(m \cdot n^2)}$ für m Wörter mit maximal n Zeichen pro Wort. Der für die Laufzeit entscheidende Teil ist hierbei die doppelte for-Schleife, beginnend in Linie 10. Im Worst Case hat diese eine Laufzeit von $\mathbf{O(n^2)}$, da für jedes Zeichen das komplette Wort erneut durchlaufen werden muss. Dieser Fall kommt allerdings in der normalen Anwendung nicht vor, da dies nur bei einer Eingabe geschieht, welche ausschließlich aus diakritischen Zeichen besteht. Das Sortieren, sowie der K-Means-Aufruf sind für die Worst Case-Abschätzung nicht von entscheidender Bedeutung da diese mit einer Worst Case-Laufzeit von $\mathbf{O(n \cdot log(n))}$ beziehungsweise $\mathbf{O(n)}$ "günstiger" sind.

Algorithm 4 kMeans.findBestK

```
1: procedure kMeans.findBestK(list)
2:     bestK = 1
3:     bestClusters = [[0]]
4:     for i from 1 to 2 do
5:         cluster, RSS = kMeans.kMeans(list, i)
6:         if RSS + DISCOUNT_FACTOR * i < bestKScore then
7:             bestKScore = RSS + DISCOUNT_FACTOR * i
8:             bestClusters = cluster
9:             bestK = i
10:        Check if the score is better than the already saved one and update
           if better
11:    return bestK, bestClusters
```

Algorithm 5 kMeans.kMeans

```
1: procedure kMeans.kMeans(list, k)
2:     Build up the starting centroids
3:     newRSS = -sys.maxsize
4:     oldRSS = -sys.maxsize
5:     runs = 0
6:     while (oldRSS - newRSS > THRESHOLD) and (runs <
   RUN_THRESHOLD) do
7:         cluster = stepA(list, centroids)
8:         centroids = stepB(list, cluster, centroids)
9:         oldRSS = newRSS
10:        newRSS = RSS(list, centroids, cluster)
11:        rus += 1
```

Algorithm 6 kMeans.stepA

```
1: procedure kMeans.stepA(list, centroids)
2:     cluster = [[] for _ in range(len(centroids))]
3:     for num, elem in enumerate(list) do
4:         nearest_centroid = -1
5:         dist = sys.maxsize
6:         for cen_num, cen in enumerate(centroids) do
7:             calculate distance
8:             update smallest distance if smaller
           Attach to cluster with nearest centroid
9:     return cluster
```

Algorithm 7 kMeans.stepB

```
1: procedure kMeans.stepB(list, cluster, old_centroids)
2:     new_centroids = []
3:     for num, cl in enumerate(cluster) do
4:         if len(cl) > 0 then
5:             Calculate the centroid of the cluster
6:             Append the new centroid to the list
7:         else
8:             Append the old centroid
9:     return new_centroids
```

Algorithm 8 kMeans.RSS

```
1: procedure kMeans.RSS(list, cluster, centroids)
2:     score = 0
3:     for num, cl in enumerate(cluster) do
4:         for list_elem in cl do
5:             Add the error of the list element to the centroid to the score
6:     return score
```

Der K-Means.findBestK Algorithmus ist dazu da, die beste Clusteranzahl zu bestimmen. Die Cluster entsprechen hierbei den Textzeilen. Aufgrund von Zeilenumbrüchen in Wörtern ist dies nötig, da sonst keine korrekte Sortierung über die X-Position o. ä. möglich ist.

Der K-Means Algorithmus, welcher in Algorithmus 5 kurz beschrieben wird, wird hierbei genutzt um zum einen die Positionen in die Cluster, also die Linien, zu verteilen und zum anderen, als Nebeneffekt, um die Anzahl der Linien über welche sich das Wort erstreckt zu bestimmen. Dies geschieht, indem die Cluster aktualisiert werden, bis der Fehler-Unterschied unter einen bestimmten Wert fällt oder <RUN_THRESHOLD>-viele Iterationen durchgeführt wurden. "THRESHOLD" und "RUN_THRESHOLD" sind hierbei zwei Konstanten. Ein Update besteht hierbei zum Ersten aus dem Zuweisen der einzelnen Positionen zu den Cluster und anschließendem neu berechnen der Centroids, also der Mittelpunkte. Die Startcentroids werden hierbei so gewählt, dass sie möglichst verteilt bezogen auf den Y-Wert liegen, also einen möglichst großen Abstand untereinander haben.

Da ein Wort maximal über 2 Zeilen gehen kann, muss auch nur für diese beiden Varianten der Algorithmus 4 ausgeführt werden. Um zu verhindern, dass immer

die Variante mit 2 Clustern genommen wird, ist ein Strafparameter eingebaut (zu sehen in Linie 6 des Algorithmus 4). Dies wäre der Fall, da mittels 2 Cluster eine bessere Anpassung an die Werte möglich ist und so der Fehler geringer wäre. Der Strafparameter wird entsprechend größer, je größer die Anzahl an Clustern wird, und ermöglicht somit eine gute Vergleichsgrundlage um welche Zeilenanzahl es sich am ehesten handelt.

In Algorithmus 6 wird jedes Element der Liste zum nächsten Centroid zugewiesen. Die Centroids bilden hierbei immer die Mitte eines Clusters ab. Zu Beginn werden diese so gewählt, dass ein maximaler Abstand zwischen ihnen besteht. Hierzu wird, für den Fall mit zwei Clustern, aus der sortierten Eingabeliste das erste und letzte Element als Centroid gewählt.
In Algorithmus 7 wird das Centroid jedes Clusters neu berechnet. Das Centroid ist anschließend wieder der Mittelpunkt des Clusters.
Der Algorithmus 8 dient, um die Größe des Fehlers durch die Clusterung zu bestimmen. Dieser Fehler ist der Abstand aller Listenelemente zu den Centroids der Cluster, zu welchen sie zugewiesen sind. Der RSS-Wert dient anschließend, um die Clusterungen zu vergleichen und die Beste auszuwählen.

Algorithmus 4 läuft im Worst Case in $\mathbf{O(n)}$. Dies ist der Fall, da sowohl die Algorithmus 6, Algorithmus 7 als auch die Algorithmus 8 jeweils in $\mathbf{O(n)}$ durchlaufen, da hierbei jeweils nie über ein Element zweimal iteriert wird, auch wenn dies zunächst aufgrund der geschachtelten Schleifen den Anschein macht.

3.5 Probleme

Während der Implementierung des Baseline-Algorithmus traten diverse Probleme auf. Die wichtigsten werden im Folgenden kurz beschrieben und deren Lösung genannt.

- **"Mehrere Linien Problem"**
  Hierbei geht es darum, die Reihenfolge der Buchstaben richtig zu wählen, sodass das endgültige Wort in der richtigen Reihenfolge zusammengebaut werden kann. Die Herausforderung hierbei tritt bei Zeilenumbrüchen auf. Aufgrund dieser Fälle ist es nicht möglich die Buchstaben nach deren X-Koordinate zu sortieren. Da bestimmte Buchstaben außerdem tiefer, wie z.

B. das g, liegen ist es nicht so leicht möglich durch reines vergleichen der Y-Koordinaten die Anzahl der Zeilen und deren Sortierung herauszufinden. Aufgrund dieser Probleme entschied ich mich eine vereinfachte Form des K-Means Algorithmus mit steigendem Strafwert für steigende Clusteranzahl zu implementieren. Mittels dieses Algorithmus ist es möglich, schnell herauszufinden über wie viele Zeilen das aktuelle Wort verteilt ist. Außerdem liefert meine Implementierung des Algorithmus, Cluster mit den Buchstaben zurück. Eine Idee der Implementierung wird in Abschnitt 3.4 genauer erläutert. In dieser Arbeit stehen die Cluster immer für die Zeilen.

Nach dem Clustering kann leicht über die X-Koordinate sortiert werden und anschließend über die Linien das gesamte Wort erstellt werden.

Für ein leichteres Verständnis wird dies nun noch einmal an einem Beispiel erläutert. Als Beispielwort nehmen wir hierfür das Wort ”efficient”. Die zugehörigen Minimum-Y-Position sehen gespeichert in einer Liste folgendermaßen aus: [90, 90.1, 91.0, 80.3, 80.5, 80.4, 80.5, 80.5] für die Zeichen [e, ffi, -, c, i, e, n, t]. Zu beachten ist in diesem Beispiel außerdem, das ein Zeilenumbruch enthalten ist. Der Algorithmus erstellt nun für den Fall mit einem und den Fall mit zwei Cluster die Cluster und berechnet die Fehlerwerte. Anschließend wird unter Einbeziehung des Strafwertes die bessere Variante bestimmt. In diesem Fall wäre es die Variante mit zwei Clustern. Die Ausgabe wäre deshalb [[e, ffi, -], [c, i, e, n, t]]. Die eigentliche Ausgabe des Algorithmus würde allerdings die Indices der Zeichen enthalten. Zur Veranschaulichung wurden hier allerdings die Zeichen angegeben. Durch Sortierung der einzelnen Cluster bezüglich deren Minimum-X-Positionen wird nun das finale Wort bestimmt. Die Ausgabe wäre also ”effi-cient”. Wichtig zu erwähnen ist hierbei außerdem das der Bindestrich nicht entfernt wird, da nicht klar ist ob dieser für den Umbruch eingeführt wurde oder Teil des Wortes ist. Dies ist nicht Teil dieser Arbeit und wird deshalb in der Evaluation ignoriert.

- **Finden des richtigen Partners für die Verknüpfung**
  Es stellte sich während der Arbeit heraus, dass es Fälle gibt, in welchen das diakritische Zeichen mit mehreren umliegenden Buchstaben zusammenpasst. Außerdem gab es einige wenige Fälle, in denen die Reihenfolge der

Buchstaben nicht korrekt war. Somit war also keine Überprüfung der umliegenden Buchstaben auf ein Zusammenpassen mit dem diakritischen Zeichen möglich. Ein Beispiel hierfür ist das Wort ”Vendée”. Hierbei gibt es für das diakritische Zeichen ´ insgesamt theoretisch drei Möglichkeiten einer Verknüpfung mit einem Basisbuchstaben, abhängig von der Reihenfolge, welche intern gespeichert wurde. Das Problem der Reihenfolge wurde bereits im obigen Punkt gelöst.
Das Problem des richtigen Zusammenbauenes von diakritischem Buchstaben und Basisbuchstaben hingegen kann auch nach Sortieren der Buchstaben noch uneindeutig sein. Dies ist zu sehen beim bereits erwähnten Wort ”Vendée”. Dies wird intern durch die Buchstabenfolge ”Vende´e” repräsentiert.

Zur Lösung dieser Uneindeutigkeit nutze ich zum einen die X-Überlagerung, zu sehen in Abbildung 6, der Zeichen und zum anderen den Y-Abstand, zu sehen in Abbildung 7.


[4]

Abbildung 6: X-Überlagerung

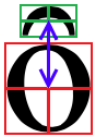
[4]

Abbildung 7: Y-Abstand

Das entscheidende Argument ist hierbei die X-Überlagerung. Hierbei wird geprüft, wie viel Prozent des diakritischen Zeichens im Bereich des jeweiligen zu kombinierenden Zeichens liegt, bezogen auf die X-Koordinate. In Abbildung 6 ist der Bereich des diakritischen Zeichen mit grün und der Bereich des Buchstaben mit rot markiert. Da der grüne Bereich komplett im roten liegt, ist die X-Überlagerung hier 100 %. Um den Fall, dass mehrere Buchstaben eine identische X-Überlagerung haben abzudecken, gibt es außerdem noch ein zweites Entscheidungskriterium, den Y-Abstand. Hierbei wird der Abstand der Mittelpunkte von diakritischem Zeichen und zu kombinierendem Zeichen bestimmt. Somit ist es anschließend möglich durch Kombination der beiden Kriterien mit nahezu 100 %-iger Sicherheit den richtigen Partner zu ermitteln.

- **Reihenfolge SplitLigatures & TranslateDiacritica**

Die Verarbeitung von Ligaturen und diakritischen Zeichen ist leichter, wenn die beiden Algorithmen getrennt sind. Das heißt zuerst werden die Sonderzeichen der einen Gruppe übersetzt und anschließend die anderen Gruppe. Hierbei sind nun beide Reihenfolgen theoretisch möglich, allerdings ist eine deutlich sinnvoller als die Andere. Es ist nämlich sinnvoller zuerst die diakritischen Zeichen zu verarbeiten, da es Fälle gibt in denen sonst das falsche Ergebnis resultiert. Dies tritt zum Beispiel bei folgender Zeichenfolge auf: æ ´. Würden nun zuerst die Ligaturen verarbeitet werden, würde das Zeichen "ae" folgen. Angenommen man würde dieses Zeichen entsprechend trennen. Bei diesem Zeichen, sowie einigen wenigen Weiteren, entschied ich mich allerdings die Trennung nicht zu vollziehen, da diese nicht eindeutig ist. Weiteres hierzu wird in Abschnitt 4.3 beschrieben.
Nach der Verarbeitung der diakritischen Zeichen wäre das Endresultat dann é oder á und nicht ǽ wie eigentlich korrekt. Aus diesem Grund müssen zuerst die diakritischen Zeichen behandelt werden.

- **Spezialfälle**
  Für Ligaturen gibt es einige wenige Spezialfälle, bei welchen im Unicode-Namen das Wort Ligatur nicht vorkommt oder bei welchen die Form von der Norm abweicht, hierbei ist beispielsweise das Wort Ligatur nicht das vorletzte Wort des Unicode-Namen. Der Unicode-Name wird hierbei mittels des Pakets unicodedata über den Unicode ermittelt. Diese Fälle werden jeweils durch einzelne Regeln behandelt, da eine Behandlung durch eine allgemeine Regel hier nicht möglich ist.

- **Gezeichnete Buchstaben**
  Der Baseline-Ansatz kann nicht mit rein gezeichneten PDFs umgehen. Das bedeutet, sobald in der "Ground-Truth"-Datei nicht der Unicode aufgeführt ist, sondern nur ein Bild verlinkt ist, versagt er. Es gibt für dieses Problem auch keine einfache regelbasierte Lösung. Aus diesem Grund wird im folgenden Kapitel ein anderer Ansatz präsentiert, welcher mit diesem Problem zurechtkommt.

4 Machine Learning-Ansatz

4.1 Idee des Machine Learning-Ansatzes

Die Idee des Machine-Learning-Ansatzes ist es eine gute Erkennung und Übersetzung von Sonderzeichen besonders für gezeichnete PDFs anzubieten, da der Baseline-Algorithmus diese nicht richtig verarbeiten kann. Hierbei wurde außerdem darauf geachtet, viele verschiedene Schriftlayouts abzudecken.

Der Baseline-Ansatz kann dies nicht realisieren, da er nur basierend auf den erstellten "Ground-Truth"-Datei arbeitet und die Bilder nicht miteinbeziehen kann. Mittels des Machine-Learning-Ansatzes ist es hingegen möglich auch diese miteinzubeziehen.

Der Machine-Learning-Ansatz geht das Problem über analysieren der Form des Zeichens und den Kontext des Zeichens im Wort an. Aufgrund der starken Ähnlichkeit einiger Sonderzeichen wäre nur eine Analyse der Form des Zeichens in einigen Fällen wahrscheinlich uneindeutig. Da auch der Kontext eines Zeichens in einem Wort nicht immer eindeutig ist, wäre auch eine Analyse nur basierend auf dem Kontext nicht zielführend. Durch eine Kombination der beiden Kriterien hingegen kann mit deutlicher höherer Sicherheit das Sonderzeichen vorhergesagt werden.

Der Machine-Learning-Ansatz wurden in Python entwickelt. Der Ansatz basiert auf 2 getrennten Machine-Learning-Modellen, welche in Abschnitt 4.4 beziehungsweise Abschnitt 4.5 genauer erklärt werden. Diese Modelle wurden mit der "Deep learning"-Bibliothek "Keras" entwickelt. Mittels dieser Modelle ist es möglich mit hoher Genauigkeit das nicht bekannte Zeichen zu ermitteln und somit das korrekte Wort anschließend zu konstruieren. Wie die beiden Modelle genau zusammenspielen und wie der Ablauf aussieht, ist in Abschnitt 4.6 genau erklärt. Außerdem wird

erklärt, wie es zu dieser Architektur kam und wieso sich diese als die Sinnvollste erwiesen hat. In Abschnitt 4.7 wird darauf aufbauend ein Modell gezeigt, welches die perfekte Mischung aus kombiniertem Modell und Baseline-Algorithmus darstellt.

Ich entschied mich für 2 Modelle, da so die Nutzung der Bilder der Sonderzeichen von der Nutzung des Kontextes im Wort getrennt ist. Durch Splitten der beiden Varianten in 2 getrennte Modell ist das Training der jeweiligen Modelle außerdem einfacher als mit einem gesamten Modell. Dies hat mehrere Gründe. Zum einen müsste das Modell deutlich größer gewählt werden, um eine ausreichende Kapazität zu erreichen, um die Komplexität des Problems in einem Modell zu bewältigen. Des Weiteren wäre die Modellarchitektur intern komplexer, da viele Eingaben kombiniert werden müssten. Außerdem wäre es schwieriger, im Falle eines Problems, den Grund dafür auszumachen und zu beheben.

4.2 Generierung der Trainingsdaten für das Training der Modelle

Hier wird nun die grobe Struktur der Trainingsdaten für die Modelle erläutert. In den jeweiligen Unterkapiteln wird auf die Details der jeweiligen Variante genauer eingegangen.

Für das Training der Modelle wurde jeweils extra eine Trainings- und Validationsdatei erstellt. Diese wurden erstellt, indem über die ”Ground-Truth”-Dateien iteriert wurde. Die ”Ground-Truth”-Dateien haben hierbei das selbe Format wie die Eingabedateien des Baseline-Algorithmus, welche in Abschnitt 3.2 beschrieben wurden. Diese ”Ground-Truth”-Dateien enthalten alle Wörter der gleichnamigen PDF. Als Teil dieser Wörter sind die Zeichen und deren Eigenschaften gespeichert. Bei jeder Iteration werden hierbei die neuen Trainingsdaten angehängt und das Vokabular erneuert. Dies kann in einem Schritt erledigt werden, da sowohl das ”padden” der Eingabe als auch Ausgabe des Modells erst beim eigentlichen Trainieren gemacht wird. Unter ”padden” versteht man hierbei das Verlängern aller Eingaben auf eine bestimmte Länge, sodass das Eingabeformat für das Modell jedes Mal dasselbe ist.

Insgesamt muss 2 Mal über die Daten iteriert werden. Beim ersten Durchlauf werden die Trainingsdaten für das textuelle Modell generiert und beim Zweiten für das visuelle Modell. Dies könnte man durchaus auch in einem Durchlauf lösen, allerdings würde das Verständnis des Codes darunter sehr leiden. Da die Trainingsdaten nur einmal generiert werden müssen, ist dieser erneute Durchlauf auch durchaus zeitlich vertretbar. Während der Erstellung der Trainingsdaten werden analog auch die Daten für die Evaluation generiert. Da diese eine andere Ausgabe haben, wird die Generierung dieser Daten in Abschnitt 5.1 noch einmal extra erklärt.

4.2.1 Visuelles Modell

Für das visuelle Modell wird über jedes Zeichen jedes Wortes, aller Dateien, iteriert. Hierbei wird geprüft, ob es sich beim aktuellen Zeichen um ein Zeichen außerhalb des ASCII-Bereichs handelt. Ist dies der Fall, so wird eine neue Zeile in der Trainings- beziehungsweise Validationsdatei begonnen. Bei jedem Schritt wird hierbei zufällig entschieden, in welche Datei dies kommt. Die Unterteilung ist hierbei 95 % Trainingsdaten zu 5% Validationsdaten.

In Abbildung 8 ist ein Auszug aus den visuellen Trainingsdaten zu sehen. Die ”13” steht hierbei für das Zeichen ”fi” und unter dem aufgeführten Link ist das Bild in Abbildung 9 gespeichert.

```
[13]    ['.../ligatures-diacritics/../00pBRon8.jpg']
```

Abbildung 8: Auszug aus den visuellen Trainingsdaten

Abbildung 9: Bild des Auszugs aus den visuellen Trainingsdaten

Eine Zeile enthält hierbei immer zwei Teile, welche durch einen Tabulator getrennt sind. Der erste Teil stellt die korrekte Ausgabe dar, der Zweite die Eingabe der ”Zeichen”. Die Eingabe des Zeichens wird hierbei als Link des Bildes gespeichert, da das Speichern der Matrix des Bildes sehr speicherplatzaufwendig ist. Das Bild

wird erst während des Trainingsprozesses geladen und anschließend direkt dem Modell übergeben. Somit ist eine minimale Speicherbelastung gegeben.

4.2.2 Textuelles Modell

Auch für das textuelle Modell wird über jedes Zeichen eines Wortes iteriert. Hierbei wird für jedes neue Wort eine neue Zeile begonnen. Wie auch beim visuellen Modell wird wieder in Trainings- und Validationsdatei, mit der selben Verteilung, unterteilt. Als erster Teil der Zeile wird die Ausgabe angefügt. Die Ausgabe enthält nur die Sonderzeichen. Diese sind außerdem in der Form aufgeführt, wie sie am Ende dem finalen Ausgabewort angehängt werden. Dies ist im Fall einer Ligatur, die geteilte Ligatur, und im Fall von diakritischen Zeichen, das kombinierte diakritische Zeichen. Als zweite Komponente wird eine Sequenz von Zeichen eingefügt, die das Wort repräsentiert. Diese Sequenz stellt die erste Eingabe für das Modell dar. Eine Ausnahme bilden hierbei die diakritische Zeichen. Diese werden in ihrer kombinierten Form angegeben. Dies ist der Fall, da vom visuellen Modell dies als Ausgabe beim textuellen Modell ankommt. Das Zusammenspiel der Modelle wird in Abschnitt 4.6 genauer erklärt.
Außerdem wird für jedes Zeichen die Position des Zeichens, als zweite Eingabe des Modells, gespeichert. Die Position besteht hierbei aus dem Minimum und Maximum der X- und Y-Koordinate. Da diakritische Zeichen in den Trainingsdateien aus 2 Zeichen bestehen wird hierbei immer der äußere Wert genommen, sprich für den Minimumwert der Kleinere und für den Maximumwert der Größere.

Die Ein- und Ausgabeteile enthalten hierbei jeweils nicht die eigentlichen Zeichen, sondern Indices. Mittels eines nebenher erstellten Vokabulars ist eine eindeutige Zuordnung zwischen Zeichen und Index möglich. Dieser Prozess wird Datei für Datei durchgearbeitet.

Im Folgenden ist ein Ausschnitt einer Trainingsdatei zu sehen. Die Validationsdatei hat dasselbe Aussehen.

Die 14 steht hierbei für das Zeichen ”fi”, die 13 für ”fi”, die 43 für ”x”, die 16 für ”u” und die 6 für ”p”. Es soll also für Eingabe ”fixup” mit den dazugehörigen Positionen die Ausgabe ”fi” erkannt werden.

```
[14]    [13, 43, 16, 6] [[198.4, 203.6, 208.6, 212.8],
                         [203.6, 208.3, 212.8, 217.0],
                         [84.7, 84.6, 84.6, 84.6],
                         [91.0, 88.7, 88.7, 88.7]]
```

Abbildung 10: Auszug aus den textuellen Trainingsdaten

Zu beachten ist hier, dass der dritte Teil aus insgesamt vier Teilen besteht. Dies ist der Fall, da für jedes Zeichen 4 Positionen gespeichert sind. Zum einen die Maximum- und Minimum-Werte der X-Positionen und zum anderen die Maximum- und Minimum-Werte der Y-Positionen. Diese sind in einer Liste, welche vier Listen enthält angeordnet. Die erste der vier Listen enthält die Min-X-Position. Die Zweite die Max-X-Positionen, die Dritte die Min-Y-Positionen und die Vierte die Max-Y-Positionen.
Des Weiteren ist wichtig zu sehen, das die Ausgabe meist nur aus ein oder zwei Zeichen besteht. Dies ist der Fall, da die Ausgabe nur die Sonderzeichen enthält, welche im Wort enthalten sind. Alle anderen Zeichen sollen nicht erkannt beziehungsweise übersetzt werden. Dieser Ansatz wurde gewählt, da so mögliche Fehler in den anderen (”normalen”) Buchstaben vermieden werden können, da diese immer korrekt vorliegen.

4.3 Unterstützte Ligaturen und kombinierte diakritische Zeichen

Theoretisch ist es möglich mittels des Machine-Learning-Ansatzes gleich viele oder sogar mehr Sonderzeichen, als der Baseline-Algorithmus, zu unterstützen. Die Zahl der unterstützten Sonderzeichen ist hierbei direkt von den Trainingsdaten abhängig. Dies ist der Fall, da jedes Zeichen, welches am Ende erkannt werden soll, natürlich auch trainiert werden muss. Für den Baseline-Algorithmus ist ein solches Training, aufgrund dessen regelbasierter Struktur, nicht nötig.

Durch ein Vergrößern der Trainingsdaten und anschließendem erneutem Trainieren der Modelle ist es leicht möglich den Machine-Learning-Ansatz auf beliebige Mengen an Sonderzeichen anzupassen.

Aktuell unterstützt der Machine-Learning-Ansatz etwas weniger Sonderzeichen als der Baseline-Ansatz. Dies hängt wie bereits erwähnt mit den Trainingsdaten zusammen. Die wichtigsten Sonderzeichen sind aber aktuell schon abgedeckt. In Tabelle 3 sind die unterstützten Ligaturen aufgeführt.
Zu beachten ist hierbei, dass die Ligaturen æ (und œ) nicht zerlegt oder umgewandelt werden. Sie bleiben als æ beziehungsweise œ bestehen. Dies ist der Fall, da diese eigenständige Buchstaben sind, welche aus Ligaturen entstanden. Aus diesem Grund können sie auch nicht zerlegt werden. Des Weiteren wäre es außerdem nicht klar in welche Buchstaben man sie zerlegen sollte um nicht in Verwechslung mit der alten Schreibweise des ä, als ae, zu kommen [20] [21].
Dasselbe gilt für den Buchstaben ø und ł. Diese können nicht als Ligaturen oder kombiniert diakritisches Zeichen eingeordnet werden, sondern sind ebenso wie das æ eigenständige Buchstaben. Aufgrund der selben Handhabung wie das æ, stehen sie in der selben Tabelle [27].

Schreibweise mit Ligatur	Schreibweise ohne Ligatur	Unicode
ł	ł	U+0142
ø	ø	U+00F8
æ	æ	U+00E6
œ	œ	U+0153
ff	ff	U+FB00
fi	fi	U+FB01
fl	fl	U+FB02
ffi	ffi	U+FB03
ffl	ffl	U+FB04

Tabelle 3: Unterstützte Ligaturen des Machine-Learning-Ansatzes

In Tabelle 4 sind die unterstützten diakritischen Zeichen aufgeführt. Hierbei werden allerdings nicht alle Varianten, wie beim Baseline-Ansatz unterstützt, sondern nur diese, welche auch Teil der Trainingsdaten sind. Sie decken allerdings die am häufigsten benutzten diakritischen Zeichen ab.

Alle unterstützen Varianten in Kombination mit dem diakritischen Zeichen (incl. Unicode)	Name des Zeichens	Unicode des diakritischen Zeichens
í (U+00ED) é (U+00E9) ó (U+00F3) á (U+00E1) ý (U+00FD) ú (U+00FA) ć (U+0107) ś (U+015b) ń (U+0144)	Acute	U+0301
è (U+00E8) à (U+00E0) ù (U+00F9) ò (U+00F2) ì (U+00EC)	Grave	U+0300
ô (U+00F4) î (U+00EE) â (U+00E2) ê (U+00EA) û (U+00FB) ĉ (U+0109) ŵ (U+0175)	Circumflex	U+0302
ã (U+00E3) õ (U+00F5) ñ (U+00F1) ĩ (U+0129) ũ (U+0169)	Tilde	U+0303
ȧ (U+00E5) ċ (U+010B) ė (U+0117) ż (U+017C)	Overdot	U+0307
ë (U+00EB) ÿ (U+00FF) ä (U+00E4)	Double dot/Diaeresis	U+0308

ü (U+00FC) ï (U+00EF) ö (U+00F6)		
ă (U+0103) ŏ (U+014F) ğ (U+011F) ŭ (U+016D)	Breve	U+02D8
ā (U+0101) ī (U+012B) ū (U+016B) ō (U+014D) ē (U+0113)	Macron	U+00AF
č (U+010D) ě (U+011B) ř (U+0159) ž (U+017E) š (U+0161)	Caron	U+02C7
ç (U+00E7)	Cedilla	U+00B8
ő (U+0151)	Double Acute	U+02DD

Tabelle 4: Unterstützte Diakritische Zeichen des Machine-Learning-Ansatzes

4.4 Visuelles Modell

4.4.1 Aufbau des Modells sowie Ein- & Ausgabeformat

Das Ziel des visuellen Modells ist es ein Zeichen durch analysieren der Form des Zeichens zu erkennen. Dies wird dadurch gewährleistet, das als Eingabe das Bild des jeweiligen Zeichens an das Modell übergeben wird. Als zu lernendes Ausgabezeichen wird der Index des Unicodes angegeben, der erkannt werden soll. Der Unicode steht hierbei im Parameter ”text” des jeweiligen Zeichens. In Unterabschnitt 4.4.2 wird noch einmal genauer darauf eingegangen.

Das Modell, dessen Struktur in Abbildung 11 zu sehen ist, ist so konstruiert, dass es in einer Lerniteration ein ganzes Batch an Bildern verarbeiten kann. Ein solches Batch kann man sich als Liste der eigentlichen Bilder vorstellen. Standardmäßig ist dies 30 Elemente lang. Aufgrund dessen hat das Modell als Eingabeformat <batchsize>-viele Bilder, welche als Matrizen gespeichert sind. Diese Matrizen haben jeweils die selbe Größe. Das Ausgabeformat ist mittels eines "One-hot-Encodings" umgesetzt, genaueres hierzu kann auch in Unterabschnitt 4.4.2 nachgelesen werden. Die Ausgabe besteht, passend zur Eingabe, erneut aus <batchsize>-vielen Elementen. Zur grafischen Veranschaulichung ist nachfolgend das Modell mit den einzelnen Schichten und ihren Verbindungen aufgeführt.

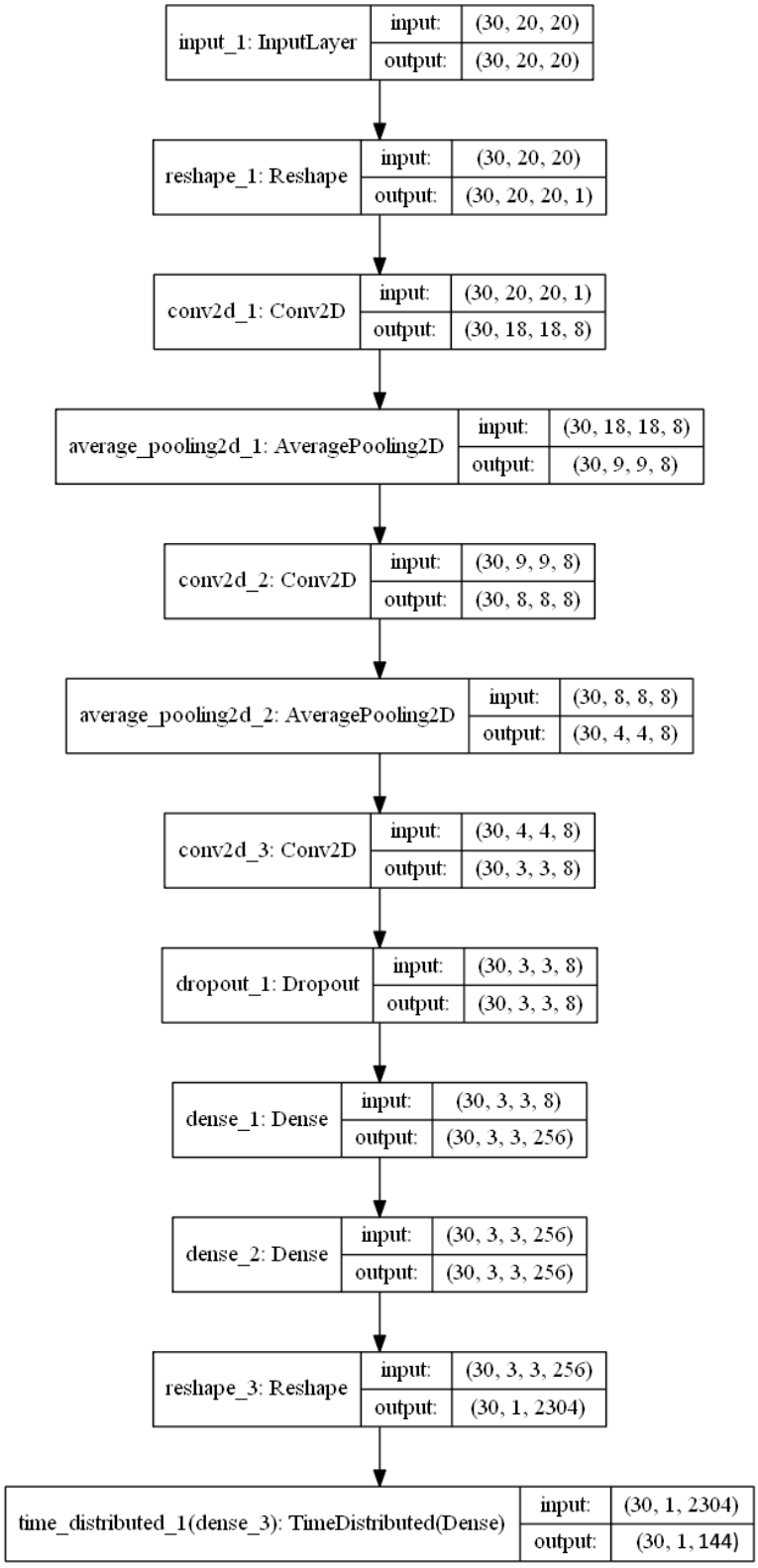

Abbildung 11: Modellstruktur des visuellen Modells

Die Struktur des Modells ist an die ”klassische” Struktur eines ”Convolutional Neural Network” angelehnt. Dies ist die inoffizielle Standardstruktur zur Bilderkennung. Auf die Eingabe-Schicht folgt eine Reshape-Schicht, um die Eingabe in die passende Form für die Convolutional-Schicht zu bringen. Bei einer Convolutional-Schicht wird ein Filter schrittweise über das Bild beziehungsweise die Matrix des Bildes geschoben und scannt dieses. Mit ”geschoben” ist hierbei gemeint, dass der Filter von links nach rechts über das Bild bewegt wird. Hierbei ist ein Schritt, als die Verschiebung um eine Position nach rechts zu sehen. Außerdem wird der Filter, sobald der rechte Rand erreicht ist, nach unten bewegt und beginnt die ”neue Zeile” erneut von links nach rechts zu scannen. Die Ausgabe dieser Schicht entspricht einer Matrix, welche allerdings um die Größe des Filters kleiner ist, wobei der letzte Parameter auf die Filtergröße gesetzt wird. Die folgende Schicht dient dazu die Informationen zu komprimieren. Hierbei wird erneut ein Filter über die Matrix geschoben und jeweils der Durchschnittswert aller Werte im Bereich 3x3 gespeichert. Diese beiden Schichten werden alternierend mehrfach angewandt, um die Informationen zum einen zu komprimieren, aber auch um die Informationsgröße zu verkleinern.
Auf die letzte ”Convolutional”-Schicht folgt eine Dropout-Schicht. In dieser wird ein gewisser Prozentsatz der Werte auf 0 gesetzt. Sinn dieser Schicht ist es dadurch das Modell zu zwingen allgemeiner zu lernen, da die Werte, welche auf 0 gesetzt werden, zufällig bestimmt werden. Dies dient der Verhinderung beziehungsweise Eindämmung von overfitting.
Auf diese Schicht folgten 2 Dense-Schichten, welche die Kapazität des Modells bewerkstelligen. Mit ”Kapazität des Modells”ist hierbei die Lernfähigkeit des Modells gemeint. Dies bedeutet, ist die ”Kapazität” höher, so kann das Modell theoretisch komplexere Relationen intern abbilden und somit ”erlernen”. Aus diesem Grund findet in dieser Schicht auch das ”eigentliche Lernen” des Modells statt. Anschließend muss nur noch das Format entsprechend an die Ausgabe angepasst werden. Dies wird wiederum mittels Reshape-Schichten gemacht.

4.4.2 Trainieren des Modells

Das Trainieren des Modells wird mittels der fit_generator-Funktion von keras umgesetzt. Grund hierfür waren Speicherprobleme, die während des Trainings auftraten. In Unterabschnitt 4.5.4 sind diese genauer erläutert.

An diese Funktion muss nicht wie bei der häufiger genutzten fit-Funktion der komplette Datensatz übergeben werden, sondern ein Generator. Dieser Generator liefert dann bei jedem Aufruf durch die fit_generator-Funktion die Daten für den nächsten Trainingsschritt. In Algorithmus 9 ist die Prozedur des Generators mittels Pseudocode dargestellt.

Algorithm 9 VisuellGenerator

```
1:  procedure VisuellGenerator(file_name, mapping, batch_size)
2:      f = open(file_name)
3:      while True do
4:          while len(data) < batch_size do
5:              read in the next line of the file
6:              if end of file is reached? then
7:                  Set pointer to beginn of the file
8:                  Read in next line
9:              Split line
10:             Translate output to none_sparse-mode
11:             Load image
12:             Format image
13:         Return data via yield-parameter
```

Beim Laden des Bildes wird dieses in Matrixform gespeichert. Damit es anschließend möglich ist das Format leicht zu ändern und es dem Modell zu übergeben.

Die im Code genutzte Umwandlung in ”none_sparse-Mode” ist hierbei ein ”one-hot-encoding”. Wie bereits erläutert, liegt in den Trainingsdaten die Ausgabe beispielsweise in der folgenden Form vor: [17]. Das Modell benötigt allerdings eine Eingabe der Form [[0, ... , 0, 1, 0, ...]]. Hierbei ist die 1 an der 17ten Stelle. Alle anderen Werte sind 0. Diese Umwandlung wird hierbei vorgenommen.

Das Modell trainiert pro Iteration ein Batch der Größe batch_size. Dieses Batch hat standardmäßig die Größe 30. Das bedeutet es werden gleichzeitig 30 Bilder vom Modell verarbeitet.

4.4.3 Verwendung des Modells

Beim Verwenden des Modells ist die Batchsize standardmäßig auf 1 gesetzt. Somit ist eine sinnvolle Nutzung des Modells gewährleistet. Wäre dies nicht der Fall, so

müssten die Daten, wenn kleiner als die Batchsize, zuerst entsprechend in ihrer Länge angepasst werden.

Das visuelle Modell weist eine ”predict”-Funktion auf. Diese nimmt als Eingabe den Link eines Bildes und liefert als Ausgabe den erkannten Buchstaben. Was hierbei intern abläuft, ist in Algorithmus 10 mittels Pseudocode dargestellt.

Algorithm 10 Predict VisuellModell

```
1: procedure PREDICT(word)
2:     Load image
3:     Format image
4:     Predict image
5:     Check if predicition is over threshold and return it afterwards
```

In Zeile 5 von Algorithmus 10 wird von ”over threshold” gesprochen. Gemeint ist hiermit, dass der Wert, welcher anzeigt, wie sicher sich das Modell mit der Vorhersage ist, über einer bestimmten Schwelle liegen muss. Diese Schwelle hat einen Wert von 0,59. Der Gedanke hierbei war, dass sich das Modell zu mehr als 50 % sicher sein sollte, also eher an die Richtigkeit der Vorhersage glaub, als nicht.

4.4.4 Probleme

Während der Konstruktion des Modells und dem stetigen Verbessern des Vorhersageergebnisses kam es zu einigen größeren Problemen. Die größten Probleme hierbei sind im Folgenden aufgeführt.

- **”Speicherproblem”**
  Zu Beginn der Konstruktion des Modells und des Trainings war der Speicherplatz kein Problem, da immer nur ein Teil der Daten genutzt wurde. Im späteren Verlauf hingegen führt das komplette Laden aller Bilder vor dem eigentlichen Trainieren zu großen Problemen. Jeder Durchlauf wurde aufgrund von Speichermangels abgebrochen.
  Aus diesem Grund war es nötig das Modell von der bisher verwendeten Trainingsmethode mittels ”fit” auf ”fit_generator” umzustellen. Müssen für die ”fit”-Funktion die gesamten Daten vorher geladen werden, so wird dies mittels eines Generators bei der ”fit_generator”-Funktion nach und nach erledigt. Dadurch kommt es nie zu einem Speicherengpass.

- **"Lernproblem/Capacity Problem"**
  Anfangs versuchte ich mit dem visuellen Modell das gesamte Wort vorherzusagen, also auch die "normalen Zeichen". Dies führte allerdings zum Problem, dass die Aufgabe für die Größe des Modells zu schwer war. Ein Vergrößern des Modells half allerdings auch nicht, da darunter die Durchlaufzeit stark litt. Somit entschied ich mich nur die "Sonderzeichen" mit dem visuellen Modell vorherzusagen und diese dann anschließend entsprechend weiter zu verarbeiten.

- **"Problem mit der Ausgabe des Modells"**
  Wie eben schon angedeutet musste ich im Laufe der Arbeit mit dem Modell die Ausgabe anpassen. Dies wurde außerdem nötig, da das Modell häufig auch "normale Zeichen" falsch vorhersagte. Diese Zeichen sind allerdings immer korrekt in der "Ground-Truth"-Datei enthalten. Aufgrund dieser Eigenschaft entschied ich mich fortan nur noch die Sonderzeichen dem Modell zu übergeben und anschließend durch Verwenden des textuellen Modells weiter zu verarbeiten.

4.5 Textuelles Modell

4.5.1 Aufbau des Modells sowie Ein- & Ausgabeformat

Das Ziel des textuellen Modells ist es Sonderzeichen durch analysieren des Kontextes innerhalb des Wortes zu erkennen. Mit Kontext sind hierbei wieder die umliegenden Buchstaben gemeint. Die Eingabe des Modells ist ein Wort mit den dazu passenden Positionen der einzelnen Buchstaben. Als Ausgabe gibt das Modell die im Gesamtzusammenhang des Wortes erkannten Sonderzeichen aus. In der Eingabe sind diese über die Unicode-Codepoints, welche auf Indices abgebildet sind, dargestellt. Die Ausgabe hingegen hat die Form, sodass daraus das korrekte Ausgabewort generiert werden kann. So ist eine mögliche Ausgabe beispielsweise der Tag "[ffi]" für die Eingabe "[e, ffi, c, i, e, n, t]". Wieso die Ausgabe nicht die Form "[f, f, i]" hat, ist in Unterabschnitt 4.6.2 genauer erläutert.

Die Modellstruktur des textuellen Modells ist in Abbildung 12 zu sehen. Wie auch beim visuellen Modell wird hier wieder in Batches trainiert. Der Unterschied ist hierbei allerdings, dass die Eingabe nicht jeweils aus einem Bild besteht, sondern

eine Eingabe aus einer Sequenz von Zeichen besteht. Diese Sequenz wird mittels des Vokabulars auf die Indices abgebildet. Das Ausgabeformat ist allerdings identisch. Es handelt sich auch um ein "One-hot-Encoding".

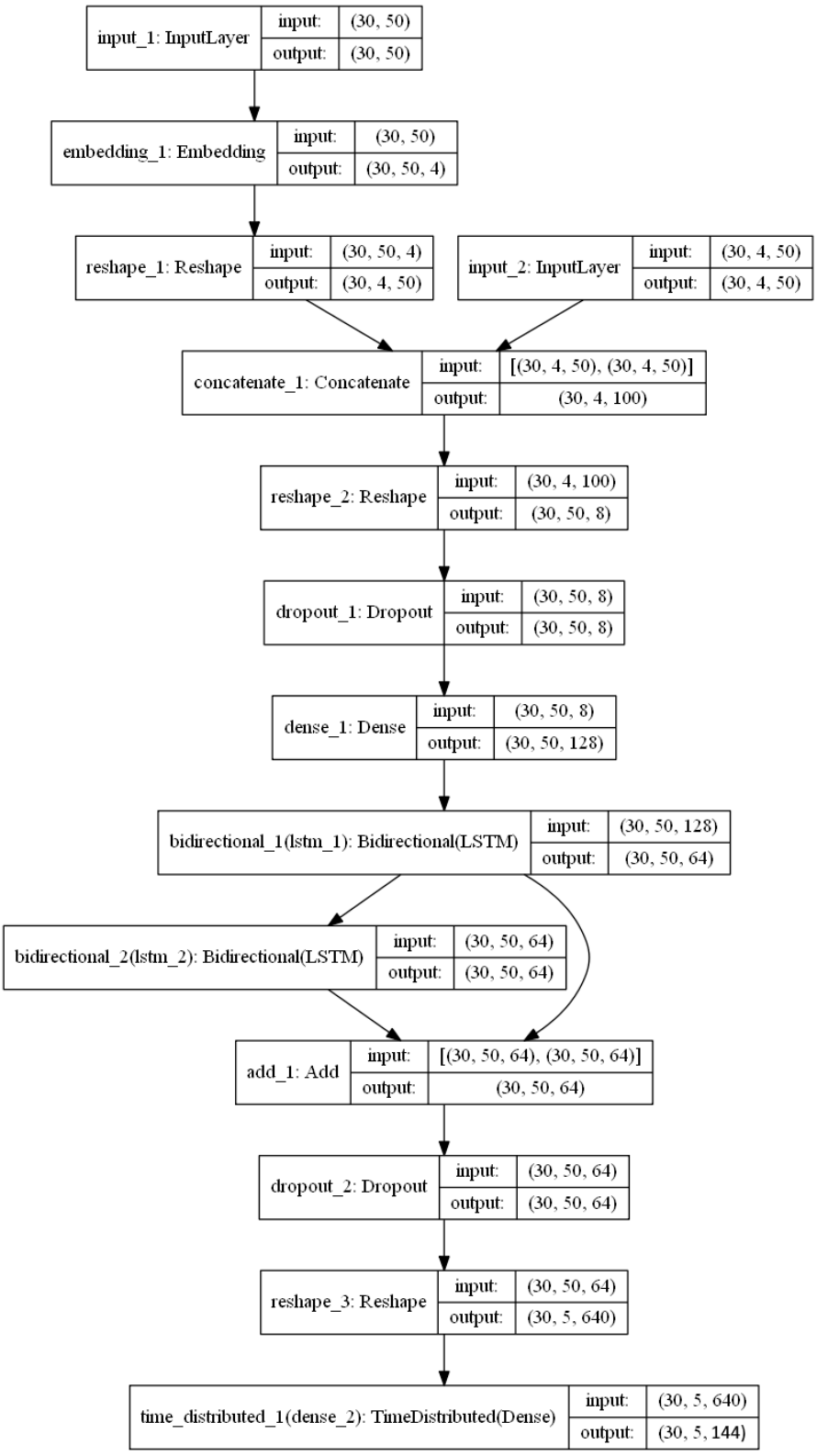

Abbildung 12: Modellstruktur des textuellen Modells

Die einzelnen Schichten des Modells wurden während des gesamten Prozesses der Bachelorarbeit mehrfach geändert. Am Ende setzte sich die oben zu sehende Variante durch. Zu Beginn ist nach der Eingabe-Schicht eine Embedding-Schicht zu sehen. Der Grund für diese wird in Unterabschnitt 4.5.4 genauer erklärt. Anschließend muss das Format angepasst werden, um der Konkatenation-Schicht gerecht zu werden. Im Folgenden wird erneut das Format verändert, um wieder eine Repräsentation der 50 Zeichen zu bekommen. 50 ist hierbei die Länge, auf die jede Eingabe erweitert wird. Die Zahl ist hierbei so gewählt, damit es nur sehr selten vorkommt, dass Wörter gekürzt werden müssen. Die darauf folgende Dropout-Schicht hat den Sinn Overfitting zu vermeiden. In dieser Schicht wird ein gewisser Prozentsatz der Eingabeindices auf 0 gesetzt. Somit muss das Modell allgemeiner lernen, um ein gutes Ergebnis zu erzielen. Ohne diese Schicht wäre die Gefahr für Overfitting deutlich höher.
Anschließend folgt eine ”normale” Dense-Schicht. Diese dient der Vergrößerung der Kapazität des Modells, was dazu führt, dass komplexere Relationen intern abgebildet werden können. Mit dieser Schicht kann in der gleichen Zeit ein besseres Lernergebnis erzielt werden. Aufgrund ihrer noch geringen Größe führt sie allerdings nicht zu einem starken Anstieg des Overfitting-Risikos.
Auf diese Schicht folgt das Herzstück des Modells. Dies besteht aus 2 bidirektionalen LSTM-Schichten, welche anschließend mittels einer ”Add”-Schicht kombiniert werden. Die bidirektionale Schicht liegt hierbei wie ein Mantel um die LSTM Schicht herum. In der bidirektionalen Schicht wird die Eingabe zuerst verdoppelt, also eine Kopie erzeugt. Anschließend wird die erste Eingabe normal, also in korrekter Reihenfolge, übergeben und die zweite Eingabe wird gedreht, also in umgekehrter Reihenfolge [7]. Die LSTM-Schicht dient um lange Ketten von Beziehungen abbilden zu können. Somit kann der gesamte Kontext des Wortes gut miteinbezogen werden. ”Normale” Dense-Schichten hätten hiermit Probleme, man spricht auch vom ”long-term dependency problem” [17].
Anschließend folgt eine erneute Dropout-Schicht, welche den selben Nutzen wie vorher bieten soll. Die letzten beiden Schichten dienen der Anpassung an das Ausgabeformat des Modells. Die ”TimeDistributed”-Schicht dient hierbei, um jedem Teil des Batches eine Dense-Schicht zuzuweisen. Somit ist es möglich ein gesamtes Batch gleichzeitig zu trainieren und jeweils eine nicht zusammenhängende Ausgabe zu bekommen. Die Ausgabe des Modells ist hierbei nur 5 Elemente lang. Diese Länge entspricht der maximal erkannten Sonderzeichen im Wort. Die Länge ist hierbei so gewählt, um annähernd 100 % der Fälle abzudecken und trotzdem nicht

zu viel unnütze Daten zu haben.

4.5.2 Trainieren des Modells

Das Trainieren läuft sehr ähnlich zum Trainieren des visuellen Modells ab. Der Generator hierzu ist in Algorithmus 11 abgebildet.

Algorithm 11 TextuellGenerator

```
1:  procedure TextuellGenerator(file_name, mapping, batch_size)
2:      f = open(file_name)
3:      while True do
4:          while len(data) < batch_size do
5:              read in the next line of the file
6:              if end of file is reached? then
7:                  Set pointer to beginn of the file
8:                  Read in next line
9:              Split line
10:             Translate output to none_sparse-mode
11:             Format both inputs (characters and positions)
12:         Return data via yield-parameter
```

Der einzige Unterschied hierbei ist, dass die Eingabe nicht erst noch geladen werden muss. Beim visuellen Modell muss dies aufgrund von Speicherplatzproblemen während der Laufzeit gemacht werden. Beim textuellen Modell müssen dafür die Eingaben gepadded werden, dies ist beim visuellen Modell nicht erforderlich.

4.5.3 Verwendung des Modells

Beim Verwenden des Modells ist die Batchsize standardmäßig auf 1 gesetzt, wie auch beim visuellen Modell. Dies hat die selben Gründe.

Das textuelle Modell weist neben der ”predict”-Funktion, welche auch das visuelle Modell hat, eine weitere Funktion auf. Diese Funktion heißt ”predict_file” und wird genutzt, um komplette Dateien vorherzusagen. Die ”predict”-Funktion selbst sieht natürlich auch leicht anders aus, macht aber im Grunde dasselbe. Die ”predict”-Funktion des textuellen Modells ist in Algorithmus 12 zu sehen.
Sie benutzt zum Laden der Daten Funktion der Datengenerierung. Allerdings

werden hierbei nur die Zeichen und die Positionen zurückgegeben, nicht das resultierende Wort, wie beim Trainieren. Anschließend wird Wort für Wort mittels der ”predict”-Funktion vorhergesagt und die Vorhersage dem Ergebnis angehängt. Wurden alle Wörter vorhergesagt, so wird das Ergebnis zurückgegeben. Die Wörter sind hierbei immer durch ein Leerzeichen getrennt.

Algorithm 12 Predict TextuellModell

```
1: procedure PREDICT(word, positions)
2:     Format character and position input
3:     Predict the characters
4:     Check if predicition is over threshold and return it afterwards
```

4.5.4 Probleme

Wie auch beim visuellen Modell kam es beim textuellen Modell während der Konstruktion zu einigen Problemen. Im Folgenden werden diese kurz erklärt.

- **”Problem mit der Ausgabe des Modells”**
  Dasselbe Problem wie beim visuellen Modell, ereilte mich auch beim visuellen Modell. Das Modell sagt Zeichen falsch vorher, welche immer korrekt als Teil der ”Ground-Truth”-Datei vorliegen. Aufgrund dieser unnötigen Fehler, welche das Gesamtergebnis deutlich verschlechterten, entschied ich mich die Ausgabe anzupassen. Die Ausgabe des textuellen Modells bestand nun fortan nur noch aus den Sonderzeichen. Allerdings werden diese in der korrekten Form ausgegeben. Also Ligaturen getrennt und kombiniert diakritische Zeichen korrekt verbunden.

- **”Konkatenationsproblem”**
  Da das Modell zwei Eingaben hat, ist es nötig intern eine Konkatenationsschicht einzubauen. Mittels dieser werden beide Eingaben kombiniert und anschließend kann das Modell die Eingabe lernen. Hierbei ist allerdings zu beachten, dass die Eingabe der Konkatenationsschicht dasselbe Format haben muss.
  Die Eingabe des Modells hat nicht dasselbe Format, da für ein Zeichen 4 Positionen übergeben werden. Daher ist es notwendig die textuelle Eingabe entsprechend zu verarbeiten. Dies wird mittels einer Embedding-Schicht gemacht. In dieser Schicht wird jeder Index der Eingabe auf einen Vektor

der Länge 4 abgebildet. Anschließend muss nur noch das Format korrekt angepasst werden. Hiermit ist nun ein Trainieren des Modells möglich.

- **"Embedding"**
  Meine ersten Modellversuche waren nicht so gut wie erwartet. Deshalb recherchierte ich nach weiteren Keras-Schichten um mein Modell weiter zu verbessern. Während meiner Recherche stieß ich auf die "Embedding"-Schicht von keras. Mittels dieser ist es möglich die Eingabe, welche beispielsweise das folgende Aussehen hat: [[4], [20]] in einen Vektor der Form: [[0.25, 0.1], [0.6, -0.2]] zu bringen. Diese Umwandlung eines Wertes (Integers) in einen Vektor wird während dem Training auch gelernt und dem Modell ist es somit möglich intern bestimmte Verbindungen von Buchstabenfolgen zu erlernen. Dies wird durch Abbilden in einen geometrischen Raum gewährleistet [3] [8]. Nach mehrfachem Testen beider Varianten zeichnete sich ab, dass die "Embedding"-Schicht das Ergebnis des Trainings deutlich verbesserte. Aufgrund dessen entschied ich mich diese, als Teil des Modells einzubauen.

4.6 "Kombiniertes Modell"

4.6.1 Aufbau des "Modells" inkl. interne Verarbeitung

Das sogenannte "Kombinierte Modell" ist eigentlich kein Machine-Learning-Modell, sondern vielmehr ein Konstrukt welches beide Modelle kombiniert und so gemeinsam nutzt.

Als Eingabe erhält es, wie auch die beiden anderen Modelle bei separater Nutzung, die aus der "Ground-Truth"-Datei ausgelesenen Daten in, in Wörter geteilter Form. Die Eingaben für das textuelle Modell werden hierbei wieder mittels des Vokabulars auf Indices abgebildet. Nun wird zuerst das visuelle Modell mit den Bilddaten der Sonderzeichen gefüttert. Die Ausgabe des visuellen Modells wird anschließend genutzt, um die Eingabe für das textuelle Modell anzupassen. Sollte sich das visuelle Modell nicht ausreichend sicher sein, so wird die Eingabe des textuellen Modells belassen und nicht verändert. Die Anpassung der Eingabe des textuellen Modells basiert darauf, das während dem Erstellen der Eingabe alle Sonderzeichen durch einen "PLACEHOLDER" ersetzt wurden. Nach dem Vorhersagen der Bilder durch das visuelle Modell werden die "PLACEHOLDER" durch

die vom visuellen Modell erkannten Zeichen ersetzt. Da es immer genau gleich viele ”PLACEHOLDER” wie zu erkennende Bilder gibt, kann es hierbei nicht zu Problemen kommen. Anschließend wird die Eingabe, mit der Position, dem textuellen Modell zugeführt. Die Ausgabe des Modells, welche nur aus den erkannten Sonderzeichen besteht, wird anschließend in das ursprüngliche Wort an die Stelle der ”PLACEHOLDER” eingesetzt. Im letzten Schritt wird das Wort dann wieder zurückabgebildet und zurückgegeben. Dieser Ablauf ist in Abbildung 13 noch einmal grafisch dargestellt. Wichtig zu bedenken ist außerdem, dass nur das am wahrscheinlichsten erkannte Zeichen genutzt wird. Sollte es also den Fall geben, dass beispielsweise das Zeichen ”fi” mit 0.999 und das Zeichen ”ff” mit 0.998 vorhergesagt wurde, so wird nur das am wahrscheinlichsten erkannte Zeichen für den weiteren Verlauf genutzt.

Für den Fall, dass sich das textuelle Modell bei der Vorhersage nicht ganz sicher ist, wird das textuelle Modell auch mit der Eingabevariante mit ”PLACEHOLDER” gefüttert, also ohne die Vorhersage des visuellen Modells zu beachten. Dass sich das Modell nicht ganz sicher ist, zeigt sich durch den Wert, welcher bei jedem Zeichen auszulesen ist. Das Modell ist sich nicht sicher, sobald dieser Wert im Schnitt aller erkannten Sonderzeichen unter 0,59 liegt. Anschließend werden beide Varianten verglichen und die bei der sich das Modell sicherer ist, wird als die finale Ausgabe genommen. So wird verhindert, dass leichte Fehler vorkommen. Als leichten Fehler kann man sich Folgendes vorstellen: Das visuelle Modell gibt den Index des Zeichens ”ffi” aus, obwohl eigentlich das Zeichen ”fi” die korrekte Ausgabe wäre. Bei der Vorhersage des textuellen Modell äußert sich dies dann, indem der Wert der Vorhersage nicht ausreichend hoch ist, also unter 0,59 liegt. Bei einer weiteren Vorhersage mittels der ”PLACEHOLDER”-Variante ist es nun möglich rein über den Wortkontext zu gehen. Dies bringt allerdings nur ein besseres Ergebnis, wenn dieser eindeutig ist. Damit ist gutes Zusammenspiel der beiden Modelle gewährleistet, um den finalen Fehler zu minimieren. Ist die wahrscheinlichere Variante ausgewählt, so muss, wie auch beim Baseline-Algorithmus, nur noch das Wort sortiert werden. Dies wird hierbei erneut unter Nutzung des Algorithmus 4 gemacht. Die Vorgehensweise ist identisch zu der beim Baseline-Algorithmus und wird deshalb nicht noch einmal erklärt.
Die finale Ausgabe wird anschließend generiert, indem Wort für Wort vom Modell beziehungsweise den Modellen vorhergesagt wird. Anschließend wird das Ergebniswort der Ausgabe angehängt und mit dem nächsten Wort fortgefahren. Dies

wird solange wiederholt bis alle Wörter verarbeitet wurden und anschließend wird das gesamte Ergebnis zurückgegeben.

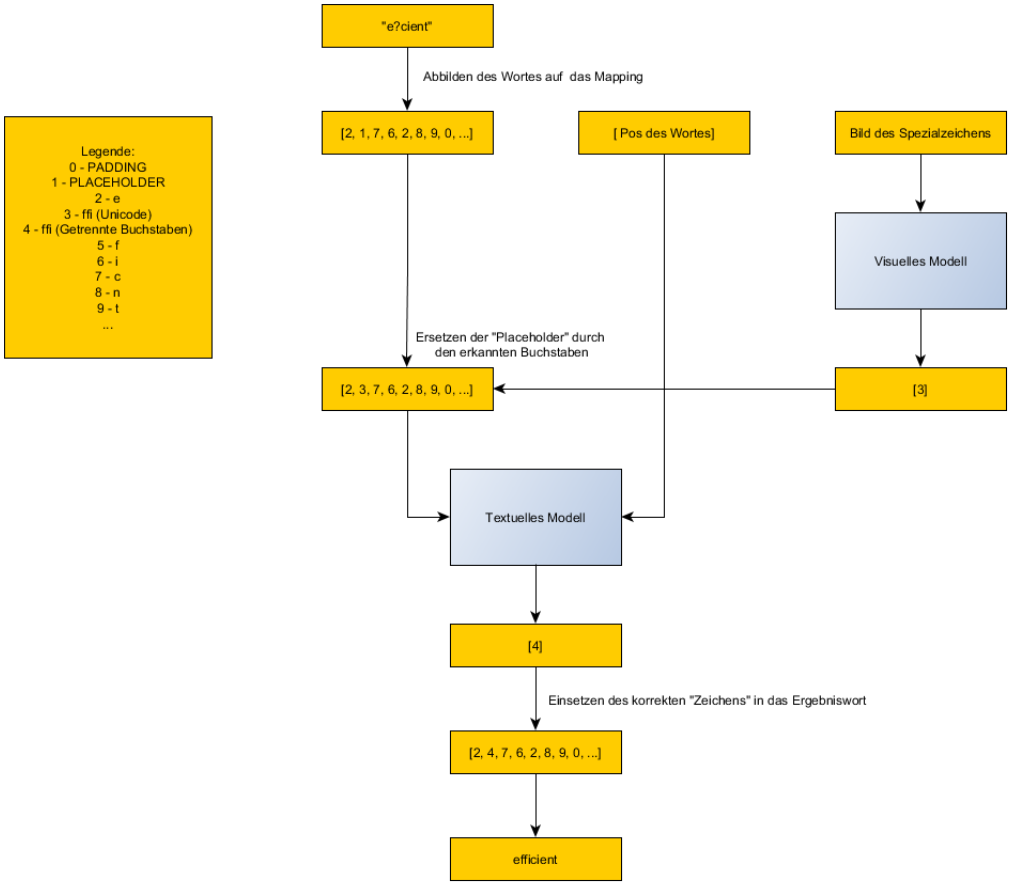

Abbildung 13: Ablauf innerhalb des kombinierten ”Modells”

In der Grafik fällt auf, dass es einen Index für ffi (Unicode) und einen für ffi (Getrennte Buchstaben) gibt. Wieso dies nötig ist, wird in Unterabschnitt 4.6.2 genauer erklärt.

4.6.2 Probleme

Auch beim kombinierten Modell kam es zu einem Problem, welches durch Anpassungen an den beiden Modellen und der internen Verarbeitung gelöst wurden.

- **”ffi (Unicode) und ffi (Getrennte Buchstaben)”**
  Zuerst war das textuelle Modell so konzipiert, dass es für das obige Beispiel (Abbildung 13) die Ausgabe [5, 5, 6] zurückgab. Für Wörter mit nur einem Sonderzeichen stellte dies kein Problem dar.
  Hatte das Wort allerdings mehrere Sonderzeichen, so war es schwer die Ausgaben, welche in diesem Format nacheinander standen, zu trennen. So kam es beispielsweise zu folgender Ausgabe [5, 5, 5, 5, 6]. Da diese Trennung nur erschwert möglich war, entschied ich mich zusätzlich Indices für die getrennten Ligaturen einzuführen. Mittels dieser ist nun eine eindeutige Trennung leicht möglich, da jeder Index für eine ”Zeichengruppe” steht. Die Rückumwandlung in das eigentliche Wort ist auch leicht möglich, da ja nicht der Unicode als Name des Tags festgelegt ist, wie bei der Eingabe des textuellen Modells, sondern die korrekten Buchstaben.

4.7 Kombiniertes Modell mit Baseline-Algorithmus

4.7.1 Aufbau des ”Modells” inkl. interne Verarbeitung

Auch bei diesem Ansatz handelt es sich nicht um ein eigentliches Machine-Learning-Modell, sondern eine Konstruktion, welche das ”normale” kombinierte Modell und den Baseline-Algorithmus nutzt. Die Eingabe für dieses Konstrukt ist, wie auch beim ”normalen” kombinierten Modell, eine ”Ground-Truth”-Datei. Es gibt also für die Vorhersage wieder eine ”predict_file”-Funktion.
Wie auch beim ”normalen” kombinierten Modell wird daraus zuerst wieder die textuelle und die visuelle Eingabe ausgelesen. Hierbei wird dies allerdings einmal im Style des Modells und einmal im Style des Baseline-Algorithmus gemacht. Diese beiden Varianten sind unterschiedlich, da beispielsweise das Modell mit Indices arbeitet, der Baseline-Algorithmus aber nicht. Durch überprüfen der Baseline-Wörter auf ”?”, welche ein nicht erkanntes Zeichen signalisieren, kann entschieden werden, welche Variante genutzt wird. Dies hat auch einen Einfluss auf die Laufzeit. Dies ist der Fall, da im Falle eines ”gezeichneten” PDFs, innerhalb der ersten Wörter ein ”?” kommt und somit schon klar ist, dass das kombinierte Modell genutzt wird. Deshalb kann an dieser Stelle abgebrochen werden. Handelt es sich um ein ”normales” PDFs, so müssen alle Wörter gecheckt werden um sicher zu sein das der Baseline-Algorithmus genutzt werden kann. Beide Varianten führen zu einem

Ergebniswort, welches anschließend zurückgegeben wird. Darauf folgend kann mit dem nächsten Wort, beziehungsweise der nächsten Zeichensequenz, fortgefahren werden. Die Entscheidung für einen der beiden Dateitypen, also gezeichnete oder normale PDF, wird hierbei für die Datei immer nur einmal getroffen. Dieser Parameter wird anschließend jeder Vorhersage mitgegeben.
Der gesamte Ablauf ist in Abbildung 14 noch einmal grafisch aufgeführt. Hierbei ist die Eingabe in der Form eines "gezeichneten" PDFs. Aufgrund dessen ist in der folgenden Verzweigung das "Ja" mit roter Schrift eingerahmt. Es würde hierbei also das kombinierte Modell für die Vorhersage genutzt werden. Anschließend wird dem Modell die Eingabe und die dazu passenden Bilder übergeben. Das "normale" kombinierte Modell würde dann als Ausgabe "efficient" zurückliefern.
Würde es sich bei der Eingabe allerdings um eine normale PDF handeln, so würde der "Nein"-Zweig genommen werden und der Baseline-Algorithmus würde die Vorhersage übernehmen. Diese würde dann im Wort "efficient" resultieren. Dieser Zweig ist in der Grafik deshalb auch angedeutet, auch wenn er für die Eingabe "[2, 1, 7, 6, 2, 8, 9, 0, ..]" gar nicht aufgerufen wird.

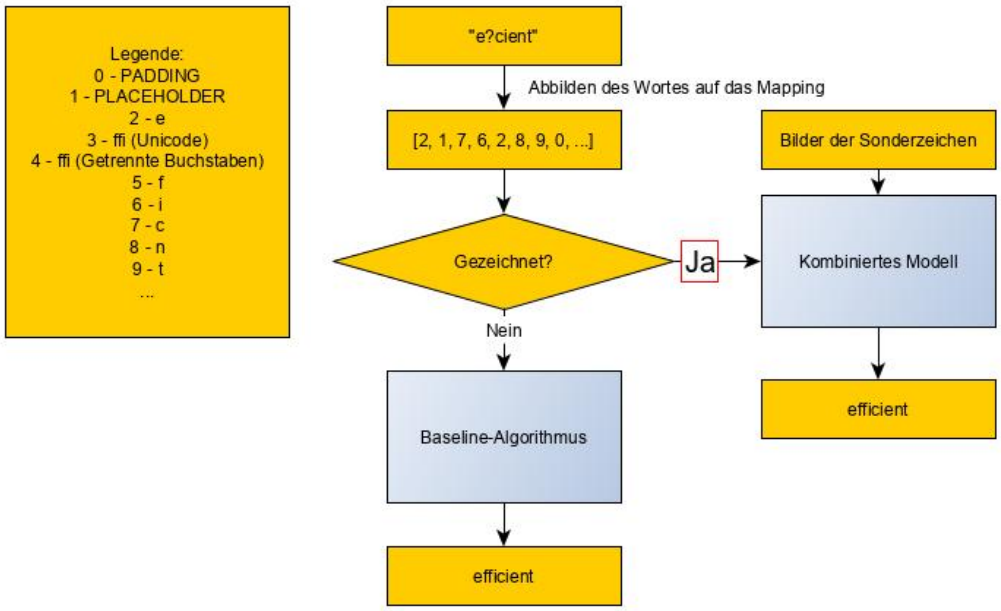

Abbildung 14: Ablauf innerhalb des Kombinierten Modells mit Baseline-Algorithmus

5 Evaluation

Die Evaluation ist aufgeteilt in zwei Teile.

Der erste Teil der Evaluation ist die Evaluation der Modelle, auf für die Modelle unbekannten Daten, jedoch in bekannten Schriftlayouts. Diese wurden beim Erstellen der Trainingsdateien für die Evaluation ausgewählt und separat gespeichert. Die Aufteilung ist hierbei 5 % Evaluationsdaten zu 95 % für den Trainingsprozess. Die Daten des Trainingsprozesses wurden außerdem in 95 % Trainingsdaten und 5 % Validationsdaten unterteilt. Die Validationsdaten wurden während des Trainings genutzt, um zu erkennen, ob das Modell overfitted oder allgemein lernt. Die Generierung der Evaluationsdaten wird in Abschnitt 5.1 kurz erklärt. Mittels dieser Evaluation soll gezeigt werden, wie gut die Modelle, die jeweiligen Layouts erlernt haben. Aus diesem Grund werden hierfür auch nur bekannte Schriftlayouts verwendet.

Der zweite Teil der Evaluation ist ein Vergleich des kombinierten Modells mit dem Baseline-Algorithmus und den beiden Programmen GROBID und pdftotext. Außerdem wird noch eine Kombination aus kombiniertem Modell und Baseline-Algorithmus für den Vergleich herangezogen. Dieser Teil ist der aussagekräftigere Teil der Evaluation.
Bei den Daten, welche für diesen Vergleich verwendet werden, handelt es sich um 2 verschiedene Dateitypen. Diese beiden Typen werden jeweils auch getrennt evaluiert. Zum einen sind dies Dateien, welche bereits im Vorhinein des Trainings aussortiert wurden, also vor dem eben beschriebenen Schritt. Diese Dateien wurden aussortiert, um während des Trainings eine ausgeglichene Menge an Daten jeden Layouts zu bekommen. Hierbei handelt sich um für das Modell bekannte Formate. Der zweite Typ hingegen ist ein für das Modell unbekanntes Format. Dieses Format weist außerdem die Besonderheit auf, dass die Buchstaben nur gezeichnet vorliegen, also in der ”Ground-Truth”-Datei an der Position des Unicodes

ein ”null” vermerkt ist. In der dazugehörigen PDF ist an der Stelle des Zeichens, der Text nicht belegt. Das heißt es ist kein Unicode gespeichert, welcher das Zeichen repräsentiert. Die korrekte Übersetzung des Wortes steht allerdings weiterhin in der ”Ground-Truth”-Datei, um eine Evaluation möglich zu machen. Dieser Typ stellt sowohl für die beiden Vergleichsprogramme als auch für mein Modell eine Herausforderung dar. Bei der gezeichneten Variante handelte es sich um 510 Dateien, bei der Variante mit bekannten Formaten um 1042 Dateien. Abschließend ist noch einmal wichtig zu erwähnen, dass keine dieser Dateien für das Training genutzt wurden.

5.1 Generierung der Evaluationsdaten

Die Generierung der Evaluationsdaten ist sehr identisch zur Generierung der Trainingsdaten. Der einzige Unterschied hierbei ist, dass die Ausgabe nicht mittels Indices gespeichert wird. Die Ausgabe ist also in Textform in der Datei gespeichert, sodass sie mit der Ausgabe der ”predict”-Funktion verglichen werden kann. Dies ist der Fall, da diese Funktion für die Verwendung durch den Benutzer ausgelegt ist und somit eine lesbare Ausgabe liefert.
Die Eingaben sind weiterhin mittels Indices gespeichert. In Abbildung 15 sind für beide Modelle Beispiele aufgeführt.

```
['fl']   [19, 12, 9, 9]   [[198.4, 203.6, 208.6, 212.8],
                           [203.6, 208.3, 212.8, 217.0],
                           [84.7, 84.6, 84.6, 84.6],
                           [91.0, 88.7, 88.7, 88.7]]
```

```
['fi']   ['.../ligatures-diacritics/Gp7QoT0j/GlG0ESem.jpg']
```

Abbildung 15: Auszug aus den textuellen beziehungsweise visuellen Evaluationsdaten

Neben der Evaluation für beide Modelle gibt es auch noch eine Evaluation für das kombinierte Modell, zu sehen in Abbildung 16. Hierbei ist zu sehen das dies aus 4 Teilen besteht, erneut wieder durch Tabulator getrennt. Zum Ersten die finale Ausgabe des kombinierten Modells, also das finale Wort. Anschließend die

Eingaben für beide Modelle. Hierbei werden zuerst die beiden Eingaben für das textuelle und anschließend die Eingabe für das visuelle Modell aufgeführt.

```
øre [1, 7, 21] [[275.1, 280.6, 284.8], ...]
    ['.../scrartcl.one-column.VY90pp7b/yPln4EY6.jpg']
```

Abbildung 16: Auszug aus den kombinierten Evaluationsdaten

5.2 Evaluation der Modelle

Zuerst wurden beide Modelle getrennt evaluiert. Bei der textuellen Variante wurde diese Evaluation geteilt in die Evaluation rein auf Kontext basierend und mit kompletter Eingabe. Bei der Kontextvariante befindet sich hierbei immer ein Index des "PLACEHOLDER" an der Stelle des Sonderzeichens. Bei kompletter Eingabe ist hingegen der Index des Unicodes an dieser Stelle aufgeführt.

Anschließend wird die Evaluation des visuellen Modells und des kombinierten Modells aufgezeigt.

Das verwendete Maß der Evaluation ist hierbei die Genauigkeit (Accuracy). Sie sagt aus, wie viel Prozent der Daten korrekt erkannt wurden.

5.2.1 Visuelles Modell

Das Ergebnis der Evaluation lag bei **99,996 %**.

Fehleranalyse

Die Fehleranalyse zeigte, dass die beiden Fehler, welche bei 54073 Vorhersagen gemacht wurden, "normale Fehler" sind. Hiermit ist gemeint, dass das korrekte Zeichen dem Vorhergesagten sehr ähnlich ist. In den beiden Fällen wurde hierbei, einmal das Zeichen "é" statt des Zeichens "ĕ" vorhergesagt. Der andere Fehler trat beim Zeichen "ī" auf. Dieses wurde fälschlicherweise als "ĭ" vorhergesagt.

5.2.2 Textuelles Modell

Das Ergebnis der Evaluation rein auf Kontext basierend erreicht ein Ergebnis von **88,97 %**. Bei kompletter Eingabe wurde hingegen ein Ergebnis von **99,41 %** erreicht.

Fehleranalyse

Die Fehleranalyse der Variante rein über Kontext zeigte das erwartete Bild. Deutete der Wortkontext nicht eindeutig auf ein bestimmtes Zeichen hin, so war es sehr wahrscheinlich, dass das Modell etwas Falsches vorhersagte. Dies ist auch völlig nachvollziehbar. Ein Beispiel hierfür sind die beiden Wörter ”aff” und ”afi”. ”aff” steht hierbei für ”beschleunigter Freifall” und ist eine Fallschirmsprungmethode. ”afi” hingegen ist der Name einer Band.
Als Eingabe würde das Modell in diesem Beispiel nur den Buchstaben ”a” und einen ”PLACEHOLDER” bekommen. Somit ist es in diesem Beispiel nicht möglich, rein aus dem Kontext des Wortes, herauszufinden, welches der beiden Wörter gemeint ist.

War die Eingabe hingegen komplett, so traten meist nur bei Wörtern mit mehreren Sonderzeichen Fehler auf, oder bei selteneren Zeichen.

5.2.3 Kombiniertes Modell

Das Ergebnis der Evaluation lag bei **99,51 %**.

Fehleranalyse

Dies macht Sinn, da im Fall von ”Unsicherheiten” des Textuellen Modells das bessere Ergebnis von ”PLACEHOLDER”- und normaler-Variante genommen wird. An einem Beispiel kann man sich dies folgendermaßen vorstellen. Angenommen die Eingabe enthält den Unicode des Zeichens ”ffi”. Die Ausgabe sollte also ”ffi” sein. Aufgrund einer nicht perfekten internen Verbindung im Modell kann es bei einem bestimmten Kontext durchaus vorkommen, dass aus Verstehen ”fi” vorhergesagt wird. Allerdings kann man dies manchmal an einem geringen Wert der

Sicherheit der Vorhersage ablesen. In diesem Fall sei also dieser Wert unterhalb der Schwelle. Dies bedeutet, dass auch die ”PLACEHOLDER”-Variante vom Modell vorhergesagt wird. Da in diesem Beispiel der Kontext eindeutig sei, folgt die korrekte Ausgabe.

5.3 Kombiniertes Modell im Vergleich mit dem Baseline-Algorithmus, GROBID und pdftotext

GROBID wurde bereits in Abschnitt 2.1 ausführlich beschrieben und wird deshalb hier nicht weiter erklärt.
Bei ”pdftotext” handelt es sich um eine Software des ”Xpdf open source projects”. Dieses beschäftigt sich mit der Konvertierung von PDF-Dateien in viele verschiedene Formate. Alle Programme des Projekts werden in Form von ”Command Line Tools” zur Verfügung gestellt und können auf Linux, Windows und Mac genutzt werden. Das Projekt läuft seit 1995 und es wird aktuell immer noch daran gearbeitet die Genauigkeit der Tools zu verbessern [16].

Die beiden Programme wurden ebenso wie das kombinierte Modell und der Baseline-Algorithmus auf den beiden bereits angesprochenen Dateitypen ausgeführt und evaluiert. In Tabelle 5 ist das Ergebnis dieser Evaluation zu sehen.

Algorithmus	Ergebnis für alle PDFs	Ergebnis für ”normale” PDFs	Ergebnis für ”gezeichnete” PDFs	Laufzeit pro Dokument
Baseline	72,45 %	99,99 %	16,90 %	0,21 sec
pdftotext	58,91 %	87,00 %	1,52 %	0,43 sec
GROBID	70,08 %	96,00 %	17,13 %	2,76 sec
Kombiniertes Modell	97,54 %	99,51 %	93,53 %	192,39 sec
Kombiniertes Modell mit Baseline-Algorithmus	97,86 %	99,99 %	93,53 %	64,21 sec

Tabelle 5: Evaluationsergebnis beim Vergleich von GROBID, pdftotext, Baseline-Algorithmus und Modell-Ansatz

Das Ergebnis der Evaluation zeigt, dass die beiden Vergleichsprogramme auf ”normalen PDFs” annähernd gleich gut funktionieren wie mein Modell-Ansatz, allerdings deutlich schneller sind. Bei der gezeichneten Variante zeigt sich allerdings ein deutlicher Unterschied im Ergebnis. Der Baseline-Algorithmus ist vor allem bei den ”normalen” PDFs sehr stark. Hat aber die bereits erwähnten Probleme mit gezeichneten Informationen. Das Ergebnis liegt hierbei allerdings nur knapp unter GROBID. Dies bedeutet das auch GROBID nur sehr wenige der gezeichneten Sonderzeichen erkennt.
Das der Baseline-Ansatz nicht auf 100 % kommt, liegt an einem kleinen Fehler in den ”Ground-Truth”-Dateien. Das Sonderzeichen ”ł” wird in die beiden Zeichen ”l” und ”open_box” getrennt wurde. Dieses Zeichen sollte allerdings nicht getrennt werden und somit kann der Baseline-Ansatz es auch nicht zusammenfügen. Für das Training der Modelle hat dieses keinen Einfluss, da diese Zerlegung für das Zeichen immer gemacht wird und das Modell so lernt damit umzugehen.

Die Kombination aus Kombiniertem Modell und Baseline-Algorithmus bietet die beste Mischung aus Ergebnis und Laufzeit. Hierbei ist allerdings in der Tabelle die durchschnittliche Laufzeit angegeben. Diese teilt sich jedoch in 0,33 sec für normale PDFs und 194,71 sec für gezeichnete PDFs auf. Die Entscheidung, ob es sich um ein normales oder gezeichnetes PDF handelt, ist hierbei für die Laufzeit nicht ganz unbedeutend. Dies wurde allerdings bereits in Abschnitt 4.7 ausführlich erklärt und wird deshalb hier nur verlinkt.

Um die Werte von GROBID und pdftotext richtig einzuordnen ist zu erwähnen, das 16,90 % der Wörter keine Sonderzeichen enthielten.

6 Schlussfolgerungen

Durch das Nutzen des Kontextes der Sonderzeichen in einem Wort und außerdem der Analysierung der Form des jeweiligen Sonderzeichens konnte eine deutliche Steigerung der Trefferwahrscheinlichkeit im Vergleich zu bisherigen Ansätzen erzielt werden. Dies zeigte sich vor allem bei rein gezeichneten PDFs. Hierbei war auf Basis des Ergebnisses von GROBID eine Steigerung um den Faktor 4,46 , für beide kombinierten Modelle, zu sehen. Auch für PDFs, welche nicht nur aus rein gezeichneten Zeichen bestehen wurde eine Verbesserung auf Basis des Ergebnisses von GROBID erreicht. Hierbei allerdings ”nur” um 3,65 % für das ”normale” kombinierte Modell. Das kombinierten Modell mit Baseline-Algorithmus erreichte, dank des Baseline-Algorithmus, bei ”normalen” PDFs eine Steigerung um 4,1 %. Die Steigerung bei ”gezeichneten” PDFs ist dem Machine-Learning-Ansatz zu verdanken, welcher allerdings auch einen Nachteil im Vergleich zu beispielsweise regelbasierten o.ä. Ansätzen hat. Die Laufzeit des Machine-Learning-Ansatzes ist im Vergleich zu anderen getesteten Programmen deutlich schlechter. Dies ist der Fall, da jedes Wort beziehungsweise jedes Sonderzeichen durch beide Modelle hindurch geführt werden muss. Besonders laufzeitintensiv ist hierbei das textuelle Modell. Dies hängt damit zusammen, dass dies eine komplexere Struktur mit komplexeren Einzelschichten hat. Besonders hervorzuheben sind hierbei die bidirektionalen LSTM-Schichten.

Somit ist abschließend zu sagen, dass das Ziel der Arbeit erreicht wurde allerdings auf Kosten der Laufzeit.

6.1 Zukünftige Arbeiten

Es gibt einige Punkte an welchen in Zukunft weiter gearbeitet werden kann, um ein noch besseres Ergebnis zu erzielen. Im Folgenden sind einige kurz genannt:

- **Eine Verbesserung der Laufzeit**
  Eine Verbesserung der Laufzeit würde dem Benutzer einen deutlichen Mehrwert bieten.

- **Vokabularerweiterung**
  Dies ist leicht möglich, indem die Trainingsdaten erweitert werden. Außerdem ist es möglich den Ansatz auf diverse andere Sprache anzuwenden.

- **Benutzeroberfläche mit Modellen im Hintergrund**
  Es ist theoretisch möglich das Modell im Hintergrund einer Benutzeroberfläche laufen zu lassen, um beispielsweise die Suche in einer PDF zu verbessern.

7 Danksagungen

An dieser Stelle möchte ich mich besonders bei meinem Betreuer Claudius bedanken. Vor allem möchte ich mich für die schnelle Bereitstellung beziehungsweise Anpassung der Trainingsdaten bedanken. Außerdem hattest du immer ein offenes Ohr für Fragen und gabst mir oft Denkanstöße in die richtige Richtung.
Des Weiteren möchte ich mich beim gesamten Lehrstuhl für die Unterstützung in Bezug auf Polyaxon bedanken.
Auch bei meinen Korrekturlesern Katharina und Fateh möchte ich mich recht herzlich bedanken.

Abschließend noch ein Dank an meine Familie und Freunde, welche mich in der gesamten Zeit unterstützt haben.

Literaturverzeichnis

[1] unicodedata — Unicode Database. https://docs.python.org/3/library/unicodedata.html, aufgerufen: 10.10.2019

[2] Combined diacritics do not normalize with unicodedata.normalize (PYTHON) (2019), https://stackoverflow.com/questions/12391348/combined-diacritics-do-not-normalize-with-unicodedata-normalize-python, aufgerufen: 13.08.2019

[3] Embedding (2019), https://keras.io/layers/embeddings/, aufgerufen: 28.08.2019

[4] U+020F: LATIN SMALL LETTER O WITH INVERTED BREVE (2019), https://www.charbase.com/020f-unicode-latin-small-letter-o-with-inverted-breve, aufgerufen: 10.08.2019

[5] Wort mit Ligatur; Wort mit diakritischen Zeichen (2019), https://ad-wiki.informatik.uni-freiburg.de/teaching/BachelorAndMasterProjectsAndTheses/SpecialCharactersExtraction, aufgerufen: 12.08.2019

[6] Ali Javed, S.N.: Diacritics recognition based urdu nastalique ocr system (November 2014), https://pdfs.semanticscholar.org/82a7/0143fead623f6c5bef6c84b5d18b22c8fc56.pdf, aufgerufen: 16.09.2019

[7] Brownlee, J.: How to Develop a Bidirectional LSTM For Sequence Classification in Python with Keras (2019), https://machinelearningmastery.com/develop-bidirectional-lstm-sequence-classification-python-keras/, aufgerufen: 06.09.2019

[8] Chollet, F.: Using pre-trained word embeddings in a Keras model (2016), https://blog.keras.io/using-pre-trained-word-embeddings-in-a-keras-model.html, aufgerufen: 28.08.2019

[9] Dominika Tkaczyk, Andrew Collins, P.S.J.B.: Machine Learning vs. Rules and Out-of-the-Box vs. Retrained: An Evaluation of Open-Source Bibliographic Reference and Citation Parsers. https://arxiv.org/ftp/arxiv/papers/1802/1802.01168.pdf, aufgerufen: 14.10.2019

[10] Israr Uddin Khattak, Nizwa Javed, I.S.S.K.K.K.: Recognition of printed urdu ligatures using convolutional neural networks (Mai 2019), https://www.researchgate.net/publication/332633436_Recognition_of_Printed_Urdu_Ligatures_using_Convolutional_Neural_Networks, aufgerufen: 16.09.2019

[11] Jakub Náplava, Milan Straka, P.S.J.H.: Diacritics restoration using neural networks (2018), http://www.lrec-conf.org/proceedings/lrec2018/pdf/573.pdf, aufgerufen: 16.09.2019

[12] Laurent Romary, P.L.: Grobid - information extraction from scientific publications (2015), https://hal.inria.fr/hal-01673305/document, aufgerufen: 16.09.2019

[13] Lopez, P.: Grobid: Combining automatic bibliographic data recognition and term extraction for scholarship publications, https://lekythos.library.ucy.ac.cy/bitstream/handle/10797/14013/ECDL069.pdf?sequence=1&isAllowed=y, aufgerufen: 16.09.2019

[14] Mario Lipinski, Kevin Yao, C.B.J.B.B.G.: Evaluation of header metadata extraction approaches and tools for scientific pdf documents (2013), http://www.sciplore.org/wp-content/papercite-data/pdf/lipinski13.pdf, aufgerufen: 16.09.2019

[15] Morik, K.: Strukturelle modelle – conditional random fields (16072013), http://www-ai.cs.uni-dortmund.de:8080/LEHRE/VORLESUNGEN/KDD/SS13/FOLIEN/5CRFsMLV.pdf, aufgerufen: 16.09.2019

[16] Noonburg, D.: Xpdfreader, https://www.xpdfreader.com, aufgerufen: 17.09.2019

[17] Olah, C.: Understanding LSTM Networks (2015), https://colah.github.io/posts/2015-08-Understanding-LSTMs/, aufgerufen: 06.09.2019

[18] Orife, I.: Attentive sequence-to-sequence learning for diacritic restoration of yorùbá language text. Proc. Interspeech 2018 pp. 2848–2852 (2018), https://arxiv.org/pdf/1804.00832.pdf, aufgerufen: 17.09.2019

[19] Wikipedia-Mitwirkende: Conditional random field (2016), https://de.wikipedia.org/wiki/Conditional_Random_Field, aufgerufen: 16.09.2019

[20] Wikipedia-Mitwirkende: Æ(2019), https://de.wikipedia.org/wiki/%C3%86, aufgerufen: 27.08.2019

[21] Wikipedia-Mitwirkende: Bedeutung im Schriftsystem (2019), https://de.wikipedia.org/wiki/Ligatur_(Typografie)#Bedeutung_im_Schriftsystem, aufgerufen: 27.08.2019

[22] Wikipedia-Mitwirkende: Diacritic (2019), https://en.wikipedia.org/wiki/Diacritic, aufgerufen: 10.08.2019

[23] Wikipedia-Mitwirkende: Diakritisches Zeichen (2019), https://de.wikipedia.org/wiki/Diakritisches_Zeichen, aufgerufen: 10.08.2019

[24] Wikipedia-Mitwirkende: Ligatur (Typografie) (2019), https://de.wikipedia.org/wiki/Ligatur_(Typografie), aufgerufen: 08.08.2019

[25] Wikipedia-Mitwirkende: Liniensystem (Typografie) (2019), https://de.wikipedia.org/wiki/Liniensystem_(Typografie), aufgerufen: 08.08.2019

[26] Wikipedia-Mitwirkende: Liniensystem (Typografie) - Grafik (2019), https://upload.wikimedia.org/wikipedia/commons/thumb/9/9d/Typografisches_Liniensystem.svg/500px-Typografisches_Liniensystem.svg.png, aufgerufen: 08.08.2019

[27] Wikipedia-Mitwirkende: ø(2019), https://de.wikipedia.org/wiki/%C3%98, aufgerufen: 27.08.2019

[28] Wikipedia-Mitwirkende: Orthographic ligature (2019), https://en.wikipedia.org/wiki/Orthographic_ligature, aufgerufen: 10.10.2019

[29] Wikipedia-Mitwirkende: Portable Document Format (2019), https://de.wikipedia.org/wiki/Portable_Document_Format, aufgerufen: 22.10.2019

[30] Wikipedia-Mitwirkende: Typographic ligature (2019), https://en.wikipedia.org/wiki/Typographic_ligature, aufgerufen: 08.08.2019