



Fachhochschul-Bachelorstudiengang

**SOFTWARE ENGINEERING**

A-4232 Hagenberg, Austria

# **Kategorisierung von lokalisierten Bilddateien mittels OCR**

Bachelorarbeit

zur Erlangung des akademischen Grades  
Bachelor of Science in Engineering

Eingereicht von

**Simon Gruber**

Begutachtet von Barbara Traxler MSc

Hagenberg, Oktober 2023

## **Erklärung**

Ich erkläre eidesstattlich, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen nicht benutzt und die den benutzten Quellen entnommenen Stellen als solche gekennzeichnet habe. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt.

Die vorliegende, gedruckte Bachelorarbeit ist identisch zu dem elektronisch übermittelten Textdokument.

Datum

Unterschrift

# Inhaltsverzeichnis

<b>1</b>	<b>Einleitung</b>	<b>1</b>
1.1	Motivation . . . . .	1
1.2	Ziele . . . . .	2
1.3	Fragestellung . . . . .	2
<b>2</b>	<b>Grundlagen</b>	<b>3</b>
2.1	Stand der Technik . . . . .	3
2.1.1	Texterkennungssysteme . . . . .	3
2.1.2	Filterung der Ergebnisdaten . . . . .	3
2.2	Verwendete Technologien . . . . .	4
2.2.1	Texterkennungssystem . . . . .	4
2.2.2	Bildbearbeitungswerkzeug . . . . .	4
<b>3</b>	<b>Konzept</b>	<b>5</b>
3.1	Annahmen . . . . .	5
3.2	Vergleich . . . . .	7
3.2.1	Metriken . . . . .	7
3.2.2	Testaufbau . . . . .	9
3.3	Verwendete Algorithmen . . . . .	10
3.3.1	Preprocessing . . . . .	10
3.3.2	Postprocessing . . . . .	16
<b>4</b>	<b>Durchführung</b>	<b>21</b>
4.1	Implementierung . . . . .	21
4.1.1	Vergleichsdaten . . . . .	21
4.1.2	Verwendete Bibliotheken . . . . .	22
4.1.3	Programmablauf . . . . .	27
4.2	Analyse . . . . .	29
<b>5</b>	<b>Zusammenfassung</b>	<b>30</b>
	<b>Quellenverzeichnis</b>	<b>31</b>
	Literatur . . . . .	31
	Medien . . . . .	32

# Kapitel 1

## Einleitung

### 1.1 Motivation

Die in Salzburg ansässige COPA-DATA GmbH bietet die Softwareplattform zenon an, die als umfassende Gesamtlösung Unternehmen in zahlreichen Anwendungsgebieten bei der Automatisierung ihrer Herstellungsprozesse unterstützt.

Die zenon-Plattform kann sowohl vom Kunden selbst, als auch durch das Professional Services Team individuell auf Kundenanforderungen zugeschnitten und in bestehende Prozesse und vor allem Software eingebunden werden. Den Grundstein für die hohe Anpassbarkeit bildet die Produktdokumentation, in der Schnittstellendokumentation, Anleitungen und Beispiele in verschiedensten Sprachen, Formaten und mit kundenspezifischen Erweiterungen umfassend sowohl für Mitarbeiter, als auch für Kunden festgehalten sind.

In der Produktdokumentation werden, besonders in Hinblick auf die grafischen Tools wie die zenon Engineering Studio Entwicklungsumgebung oder die zenon Service Engine, zahlreiche Grafiken verwendet, um Beispiele verständlicher zu machen und Anleitungen übersichtlicher zu gestalten. Um bei dem großen Funktionsumfang der zenon-Tools, den vielen Sprachen, Anpassungen und den unterschiedlichen Themengebieten innerhalb der Dokumentation nicht den Überblick zu verlieren, benötigt das interne „Technical Content and Translation“ Team unterstützend zu dem intern verwendeten CMS eine dedizierte Anwendung zur Verwaltung von sprachabhängigen Bilddateien.

Während das Programm auch die Basisfunktionalität, das effiziente Speichern, Bearbeiten, Löschen, Abrufen beziehungsweise das generelle Verwalten von Screenshots und der zugehörigen Metainformation abdecken soll, konzentriert sich diese Bachelorarbeit primär auf die Kategorisierungsfunktionalität.

Mithilfe von optischer Texterkennung (engl. optical character recognition, „OCR“) soll es den Mitarbeitern möglich gemacht werden, hochgeladene Screenshots und Grafiken innerhalb von kürzester Zeit aufgrund ihrer Inhalte zu verschlagworten, um sie später anhand dieser suchen zu können.

## 1.2 Ziele

Das Ziel dieser Bachelorarbeit ist das Ermitteln einer Vorgehensweise für Texterkennung in Screenshots von grafischen Oberflächen. Verschiedene Algorithmen zur Bildbearbeitung vor der Texterkennung oder Nachbearbeitung bzw. Filterung der Ergebnisdaten werden evaluiert und anhand von festgelegten Qualitätskriterien analysiert.

Die prototypische Implementierung dient als Basis für jegliche Tests und Analysen, anhand derer die Algorithmen automatisch verglichen werden. Die entwickelten Komponenten werden als Bibliotheken zur Verfügung gestellt, um die Texterkennung inklusive automatischer Bildverarbeitung und Filterung der erkannten Inhalte bzw. Schlagworte später in anderen Anwendungen weiterverwenden zu können.

## 1.3 Fragestellung

Ziel dieser Bachelorarbeit ist es, herauszufinden, welche Vorgehensweisen bei der Texterkennung zu den besten Ergebnissen führen. Dazu werden die Resultate der Bild- und Textverarbeitung anhand gängiger Fehlermetriken für Texterkennungssysteme verglichen.

## Kapitel 2

# Grundlagen

### 2.1 Stand der Technik

#### 2.1.1 Texterkennungssysteme

Optische Texterkennung wird in der Informationstechnik eingesetzt, um Textinhalte aus gedruckten oder digital reasterisierten Medien zu extrahieren. Dieses Verfahren kann für diverse Anwendungsgebiete genutzt werden, wie beispielsweise für Handschrifterkennung oder für das Ablesen von Nummernschildern eines Autos [1]. Auf dem Markt gibt es dafür bereits viele kommerzielle Komplettlösungen wie „IronOCR“, „Google Cloud Vision“, „Amazon Textract“ oder „Microsoft Azure Computer Vision“, die oftmals sehr gute Ergebnisse erzielen und sich gut in bestehende Prozesse oder Anwendungen integrieren lassen [5, 15].

Heutige Texterkennungssysteme arbeiten oft mit neuronalen Netzwerken sowie fortgeschrittenen Bildverarbeitungsalgorithmen, um Text in Bilddateien zu erkennen und zu extrahieren. Während es zahlreiche wissenschaftliche Werke zur grundlegenden Funktionsweise von optischen Texterkennungswerkzeugen gibt (beispielsweise „Optical character recognition“, Eikvil 1993 [4] oder „A survey on optical character recognition systems“, Islam 2017 [6]), werden die genauen Schritte zur richtigen Vorbereitung der Bilddaten – besonders in Bezug auf Screenshots – oftmals nur oberflächlich behandelt.

#### 2.1.2 Filterung der Ergebnisdaten

Das Themengebiet des Natural Language Processing beschäftigt sich mit der Interaktion zwischen menschlicher Sprache und Computern. Techniken aus der Informatik, Linguistik und dem maschinellen Lernen werden kombiniert, um mit menschlicher Sprache umzugehen und beispielsweise Textanalyse, Übersetzungen, Spracherkennung oder Dialogsysteme möglich zu machen [2]. Durch die große Aufmerksamkeit und die vielseitige Nutzung der Technologien – angefangen von automatischer Rechtschreibkontrolle bis hin zu digitalen Sprachassistenten – sowie dem Aufkommen von neuronalen Netzwerken wurden in diesem Forschungsgebiet in den letzten Jahren immer wieder Fortschritte

erzielt [3, 7].

Dadurch gibt es zahlreiche wissenschaftliche Ressourcen, die als Grundlage für die in dieser Bachelorarbeit verwendeten Vorgehensweise zur Interpretation und Extraktion relevanter Schlagworte aus den erkannten Freitextdaten dienen.

## 2.2 Verwendete Technologien

### 2.2.1 Texterkennungssystem

Die Nutzung der in Kapitel 1 erwähnten Anwendungen bzw. Dienstleistungen ist kostenpflichtig und die genaue innere Vorgehensweise dieser Programme ist nicht öffentlich bekannt [17–19].

Aufgrund dieser Tatsachen ist die Wahl des Texterkennungssystems für die prototypische Implementierung dieser Bachelorarbeit auf die seit 2005 unter der Freie-Software-Lizenz „Apache 2.0“ veröffentlichten „Tesseract Open Source OCR Engine“ (kurz: Tesseract) gefallen [12]. Diese basiert seit der Major-Version 4 auf einem neuronalen Netz, durch welches mithilfe von sprachspezifischen Trainingsdaten Texte in Bildern erkannt werden können. Außerdem stellt sie mit mittlerweile über 50.000 Sternen auf der Repository-Hosting-Plattform GitHub eines der beliebtesten Texterkennungssysteme dar [21; 22].

### 2.2.2 Bildbearbeitungswerkzeug

Als Werkzeug für die Durchführung der notwendigen Bildbearbeitungsschritte wurde die Softwarebibliothek „ImageMagick“ [20] gewählt. Sie ist umfassend dokumentiert, flexibel und lässt sich gut in gängige Programmiersprachen einbinden. Viele in der Bildverarbeitung genutzte Operationen sind außerdem bereits implementiert, was schnelles Prototyping vereinfacht und die Bibliothek zu einer idealen Wahl für die Realisierung von Bildbearbeitungsschritten in der prototypischen Implementierung macht.

## Kapitel 3

# Konzept

### 3.1 Annahmen

Um die Texterkennung mittels Tesseract und die anschließende Filterung der Ergebnisdaten zu verbessern, ist es sinnvoll, Anwendungsspezifische Annahmen für den Verarbeitungsablauf festzulegen.

#### Preprocessing

##### Eigenschaften von Screenshots

Im Falle dieser Bachelorarbeit handelt es sich bei den zu verarbeitenden Bildern ausschließlich um digitale Bildschirmaufnahmen von grafischen Benutzeroberflächen. Da die Es kann also angenommen werden, dass die Screenshots keine Transparenz aufweisen, die Perspektive der Aufnahme nicht verzerrt ist und der Kontrast in den meisten Fällen ausreicht, um die relevanten Inhalte zu erkennen. Weiters ist bei der Bildverarbeitung auf farbige Hintergrundflächen zu achten, mit deren Unterstützung Bildelemente in modernen grafischen Oberflächen oft gruppiert oder getrennt werden. Nach Sichtung des zu verarbeitenden Bilddatensatzes fällt zudem auf, dass die manche Screenshots durch das Selektieren mit der Maus sehr eng abgeschnitten wurden. Auch das ist bei der Vorverarbeitung zu berücksichtigen.

##### Optimieren von Daten für Tesseract

Für die Verwendung von Tesseract ist es wichtig, unabhängig von der Diversität der Ausgangsdaten möglichst einheitliche Bilder zu generieren, die den Trainingsdaten des neuronalen Netzes ähnlich sehen. Während störende Elemente wie Bildrauschen aus dem Bild entfernt werden sollen, sollen Texte unabhängig von der Hinter- bzw. Vordergrundfarbe gut zu erkennen und leicht von Formen oder grafischen Symbolen abzugrenzen sein [13] [9].



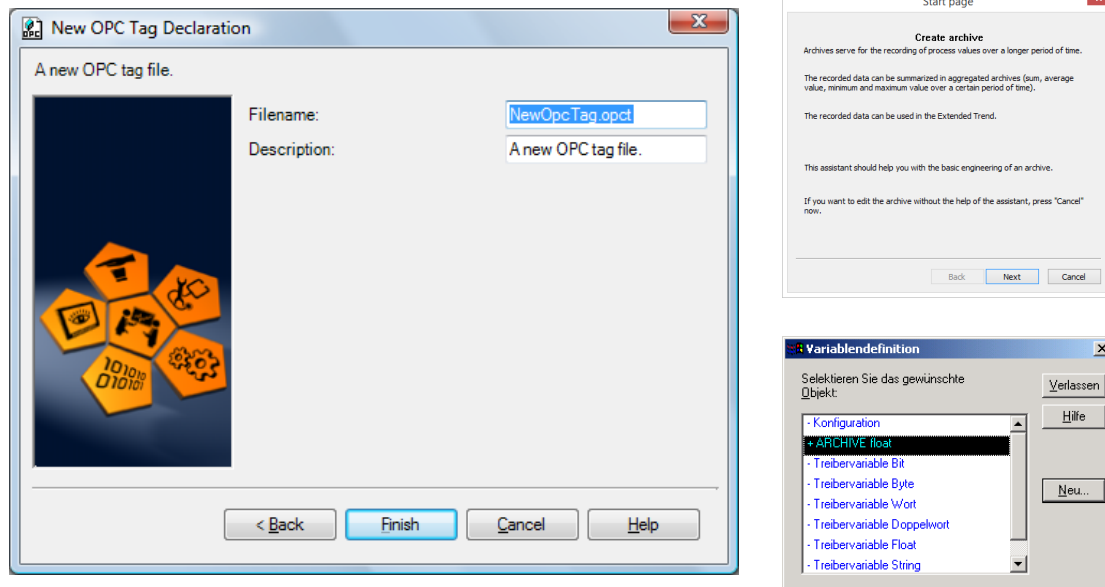


Abbildung 3.1: Beispielhafte Auswahl typischer Dialogscreenshots.

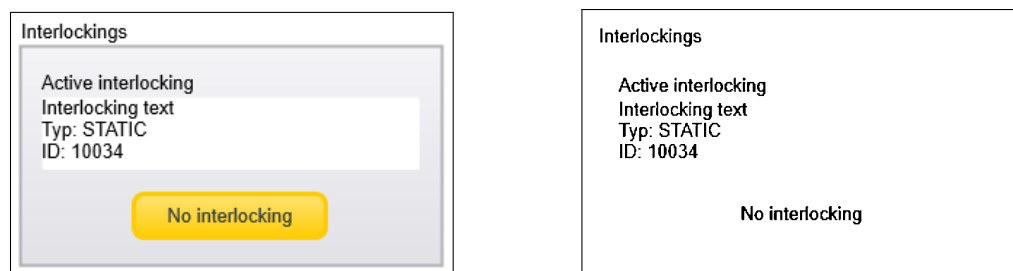


Abbildung 3.2: Ein optimales Ergebnisbild. Jegliche farblichen Flächen wurden durch die Bildverarbeitung entfernt. Übrig bleibt klar lesbarer Text mit einem hohen Kontrast zum Hintergrund.

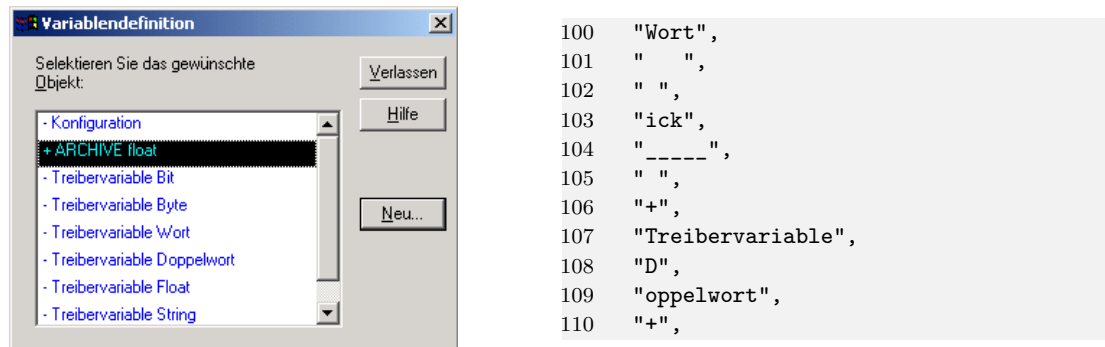
## Postprocessing

### Filtern von Symbolen

Bei der Texterkennung kommt es manchmal vor, dass grafische Elemente als Unicode-Symbole erkannt werden. Beispielsweise finden sich in den ungefilterten Ergebnisdaten oft Aufzählungszeichen „•“ oder diverse Varianten von Bindestrichen „–“. Diese Zeichen sind gemäß Anwendungsanforderungen nicht relevant für die Schlagwortsuche und können somit entfernt bzw. ignoriert werden.

### Mehrsprachigkeit

Eine weitere Anforderung an das Textverarbeitungssystem ist außerdem das Einlesen und Interpretieren mehrsprachiger Bilddateien. So sollen beispielsweise Bilder mit engli-



**Abbildung 3.3:** Auszug aus den ungefilterten Ergebnisdaten bei Durchführung der Texterkennung in dem gezeigten Screenshot.

schen, deutschen oder italienischen Inhalten zugeführt und die Ergebnisdaten richtig verarbeitet werden können. Um eine Filterung für verschiedene Zeichensätze zu ermöglichen und eine Unterstützung für Sprachen mit nicht-lateinischen Schriften zu gewährleisten, werden dynamische Sprachfilter verwendet, die individuell an die jeweilige Sprache angepasst werden können. Um die Ergebnisdaten nicht unnötig zu verkomplizieren, werden für die initialen Tests und die Beschreibung der generellen Vorgehensweise im Rahmen dieser Bachelorarbeit jedoch nur deutsche oder englische Inhalte verarbeitet.

### Schlagworte

Für die spätere Suche von Screenshots sollen relevante Schlagworte aus den erkannten Textdaten extrahiert werden. Ein Wort eignet sich dann als Schlagwort, wenn es in relevantem Bezug zum jeweiligen Bild steht und dabei idealerweise eine wichtige Aktion oder Information widerspiegelt. Inhalte, die direkt in der grafischen Benutzeroberfläche ersichtlich sind, eignen sich demnach besonders gut als Suchworte. Damit die Menge an Schlagwörtern allerdings nicht zu unspezifisch wird, sollte es vermieden werden, allgemeine sogenannte Stoppwörter (engl. "Stop words") ohne besondere Semantik, wie „und“, „oder“, in das Ergebnis mit aufzunehmen. Sie beinhalten keine spezifische Information und fördern aufgrund ihrer Häufigkeit das Auftreten von Verwechslungen.

## 3.2 Vergleich

### 3.2.1 Metriken

Um die erkannten Ergebnisse unter Verwendung der verschiedenen Pre- und Postprocessing Schritte mittels eines einheitlichen Systems vergleichen zu können, wird auf die in der optischen Texterkennung gängigen Metriken „Character Metric“, auch bekannt als „Character Error Rate“ und „Word metric“ oder „Word Error Rate“ (kurz: "WER") [8], basierend auf der Levenshtein-Distanz [10] zurückgegriffen.

Sowohl die Character- als auch die Word Error Rate sind beliebte Vergleichswerte, die ihren Ursprung in der computergestützten Sprachverarbeitung bzw. automatischen

Spracherkennung haben [16]. Da die optische Texterkennung und die automatische Spracherkennung jeweils darauf abzielen, maschinenlesbaren Text aus nicht-strukturierten Daten zu extrahieren, sind die Prinzipien dieser Metriken auch auf die optische Texterkennung anwendbar [14].

### 3.2.1.1 Word Error Rate

Die Word Error Rate (kurz: "WER") beschreibt den prozentualen Anteil der falsch erkannten oder fehlenden Wörter eines Textes im Vergleich zu einer Referenz, welche im Falle der folgenden Vergleiche immer alle sichtbaren Texte im Bild repräsentiert. Je niedriger die WER, desto genauer ist der OCR-Vorgang. Um die WER zu berechnen, bildet man die Summe aller notwendigen Ersetzungen, Entfernungen und Einfügungen, um aus dem erkannten Text den Referenztext bilden zu können und setzt sie mit der Gesamtwortanzahl im Referenztext in Verhältnis [11].

#### Berechnung

Die mathematische Formel für die Word Error Rate lautet somit wie folgt:

$$\text{WER} = \frac{S + D + I}{N}$$

wobei die einzelnen Komponenten folgende Größen darstellen:

- $S$  beschreibt die Anzahl der falsch erkannten Wörter (engl. "Substitutions")
- $D$  beschreibt die Anzahl der im Resultat fehlenden Wörter (engl. "Deletions")
- $I$  beschreibt die Anzahl der im Resultat fälschlicherweise eingefügte Wörter (engl. "Insertions")
- $N$  beschreibt die Gesamtanzahl der Wörter in der Referenz

#### Vorteile und Nachteile

Die WER spiegelt ohne großen Rechenaufwand direkt wider, wie stark die erkannten Texte der Referenz gleichen. Hierbei werden fehlerhafte Einsetzungen, Löschungen und falsch erkannte Wörter bzw. Teilwörter gleichermaßen gewichtet. Es ist jedoch nicht möglich, die korrekte Reihenfolge der erkannten Wörter darzustellen oder bestimmte wichtige Stellen im Text höher zu gewichten als andere. Auch werden fehlerhaft erkannte Wörter, auch wenn nur ein einzelner Buchstabe falsch ist, als vollwertige Ersetzung wahrgenommen, wodurch die WER selbst bei bis auf wenige Zeichen gut erkannte Texte stark beeinflusst werden kann.

Um also ein umfassendes Bild von der Genauigkeit des Texterkennungssystems zu erhalten, ist es sinnvoll, die Ergebnisse nicht nur anhand der WER, sondern auch noch mindestens anhand einer weiteren Fehlermetrik, wie beispielsweise der CER, zu vergleichen.

### 3.2.1.2 Character Error Rate

Die Character Error Rate (CER) beschreibt die Anzahl der falsch erkannten oder fehlenden Zeichen im Vergleich zu einem Referenzwort und basiert auf der Levenshtein-Distanz [10]. Je niedriger die CER, desto genauer ist der OCR-Vorgang. Ähnlich wie die WER wird die CER aus der Summe aller notwendigen Ersetzungen, Entfernungen und Einfügungen, um aus dem erkannten Wort die Referenz bilden zu können, geteilt durch die Zeichenanzahl des Referenzwortes, gebildet.

#### Berechnung

Die mathematische Formel für die Word Error Rate lautet somit wie folgt:

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

wobei die einzelnen Komponenten folgende Größen darstellen:

- $S$  beschreibt die Anzahl der falsch erkannten Wörter (Substitutions)
- $D$  beschreibt die Anzahl der im Resultat fehlenden Wörter (Deletions)
- $I$  beschreibt die Anzahl der im Resultat fälschlicherweise eingefügte Wörter (Insertions)
- $N$  beschreibt die Gesamtanzahl der Wörter in der Referenz

#### Vorteile und Nachteile

Die CER fasst in einem Wert zusammen, wie viele Änderungen auf Zeichenebene notwendig sind, um aus dem erkannten Wort das Referenzwort zu bilden. Es ist dabei wie bei der WER nicht relevant, in welcher Reihenfolge diese Zeichen auftreten. Ebenso gibt es keine gesonderte Gewichtung für Ersetzungen, Löschungen oder Einfügungen, wodurch besonders bei kurzen Wörtern auch kleinere Abweichungen bereits zu einer hohen CER führen können.

Durch den detaillierten Vergleich der einzelnen Wörter auf Zeichenebene stellt die CER jedenfalls ein ausreichend gutes Komplement zur WER dar, um in den folgenden Vergleichen genutzt werden zu können.

### 3.2.2 Testaufbau

Der Testaufbau im Rahmen der Implementierung, beschrieben in Abschnitt 4.1, erlaubt ein dynamisches Verketteten von verschiedenen Bildverarbeitungs- und Textfilterungsschritten. Für einen objektiven Vergleich zwischen den unterschiedlichen Vorgehensweisen und Algorithmen wird eine Grundabfolge der jeweiligen Schritte in einer „Processing-Pipeline“ definiert. Die Ergebnisse können schließlich anhand der in Unterabschnitt 3.2.1 beschriebenen Fehlermetriken mit einer durch den Menschen verschlagworteten Vergleichsmenge abgeglichen werden.

## 3.3 Verwendete Algorithmen

### 3.3.1 Preprocessing

Beim sogenannten „Preprocessing“ werden die zu verarbeitenden Bilder für die Texterkennung vorbereitet, um die Qualität der erkannten Textdaten zu verbessern.

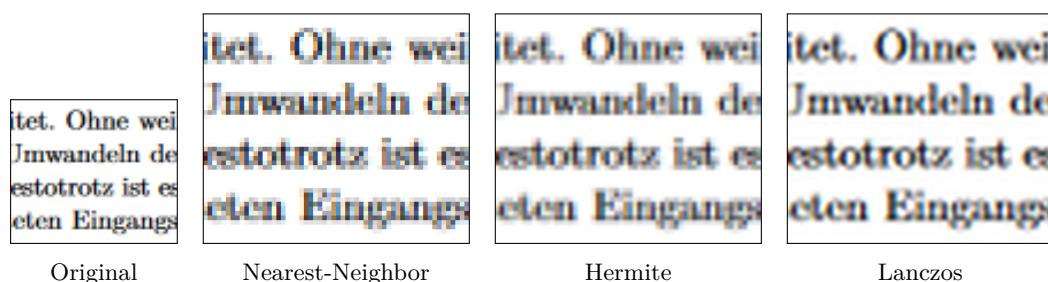
Verwendet man moderne Tesseract-Implementierungen, sind in diesen oft bereits rudimentäre Bildverarbeitungswerkzeuge verfügbar [23]. Mit diesen Werkzeugen werden die eingespeisten Bilder – sofern nicht bereits im richtigen Format – automatisch für die Texterkennung vorbereitet. Ohne weitere Einstellungen zu treffen, bewirkt diese Bildverarbeitung zwar ein Umwandeln der Eingangsgrafiken in ein meist gut für Tesseract geeignetes Bild, nichtsdestotrotz ist es jedoch sinnvoll, die Bildverarbeitungsschritte individuell auf die erwarteten Eingangsdaten anzupassen, um sich den in Abbildung 3.1 definierten optimalen Tesseract-Eingangsdaten anzunähern.

Die folgenden Preprocessing-Schritte basieren auf der empfohlenen Vorgehensweise zur Verbesserung der Output-Qualität laut Tesseract-Dokumentation [21]. Gemäß den obigen Annahmen werden jedoch weder perspektivische Fehler, noch ein eventuelles Rauschen korrigiert. Konkret werden folgende Bildverarbeitungsschritte verglichen:

#### 3.3.1.1 Resampling

Bei Resampling wird die Bildauflösung durch „Neuabtastung“ verändert. Um die für Tesseract optimale [21] Mindestauflösung von 300 dpi zu gewährleisten, muss das Eingangsbild, sofern es die Mindestauflösung unterschreitet, zunächst entsprechend vergrößert werden.

Da Tesseract auf klare und scharfe Kontraste angewiesen ist, um Text korrekt zu identifizieren, eignen sich nicht alle von ImageMagick zur Verfügung gestellten Skalierungsmethoden für die Weiterverarbeitung. Wie in Abbildung 3.4 zu sehen, neigen einige Filter besonders beim Hochskalieren dazu, Unschärfen oder Artefakte zu erzeugen, die die Genauigkeit der Texterkennung negativ beeinflussen können.



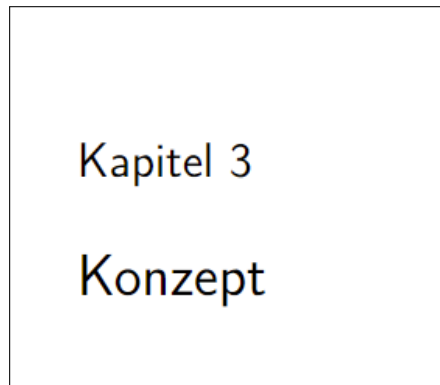
**Abbildung 3.4:** Ein Vergleich unterschiedlicher Resampling-Filter. Bei Anwendung des Lanczos-Filters bleiben Details und Konturen vergleichsweise gut erhalten und der Text ist gut lesbar.

Nach einigen Tests hat sich gezeigt, dass Bilder, die mittels des Spline-Verfahrens oder

der Hermite-Interpolation skaliert wurden, natürlich für das menschliche Auge ansprechender wirken. Tesseract jedoch profitiert stark von klaren Texten und hohen Kontrasten, weswegen diese Art des Resamplings keine ideale Basis für das Preprocessing bietet. Deswegen wird für die weiteren Schritte die Interpolation nach Lanczos für das Resampling verwendet.

#### 3.3.1.2 Rahmen

Befindet sich Text zu nah am Rand des Bildes, kommt es vor, dass dieser nicht richtig erkannt werden kann. Ebenso kann auch ein zu großer einfärbiger Rahmen am Rand des Bildes dazu führen, dass Bildsektionen fälschlicherweise als „leer“ erkannt und übersprungen werden, wodurch der zu erkennende Text nicht in die Ergebnisdaten mit aufgenommen wird.



**Abbildung 3.5:** Ein im Verhältnis zur Bildgröße zu großer einfärbiger Rahmen

#### 3.3.1.3 Binarisierung

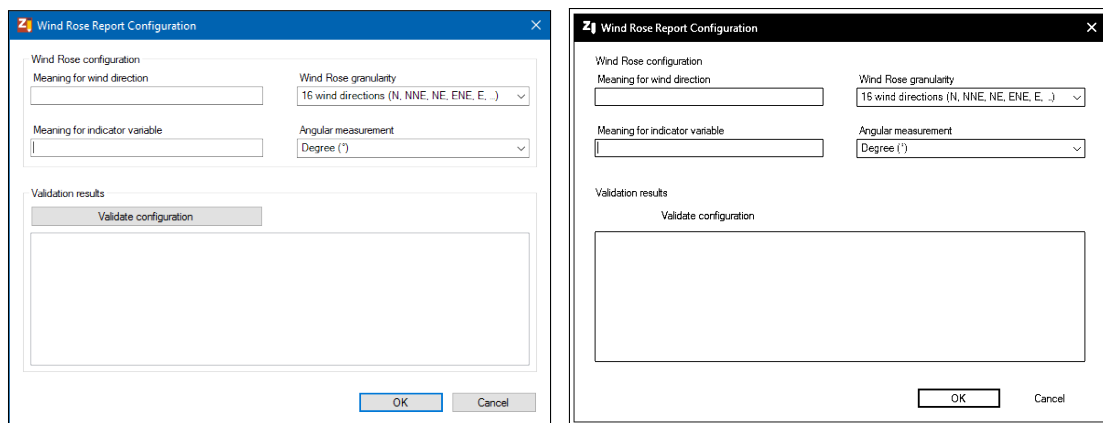
Das Erzeugen eines Binärbildes ist durch Anwendung von Segmentierungsverfahren möglich. Schwellenwertverfahren (engl. "Thresholding") bilden eine Untergruppe der Segmentierungsverfahren und werden genutzt, um Graustufenbilder Pixel für Pixel in binarisierte Ergebnisbilder mit zwei Segmenten, also einem Vordergrund und einem Hintergrund umzuwandeln. Der dazu notwendige Schwellenwert kann entweder fix definiert oder anhand von verschiedensten Algorithmen ermittelt werden. Ziel ist es, durch die Binarisierung textuelle Bildinhalte unabhängig von der eigentlichen Vorder- und Hintergrundfarbe mit ausreichendem Kontrast darzustellen. Somit ist das Texterkennungssystem in der Lage, die einzelnen Textelemente und deren Inhalte besser zu identifizieren und zu verarbeiten.

ImageMagick bietet eine Vielzahl an Thresholding-Algorithmen, deren Eignung in Abschnitt 3.2 verglichen wird.

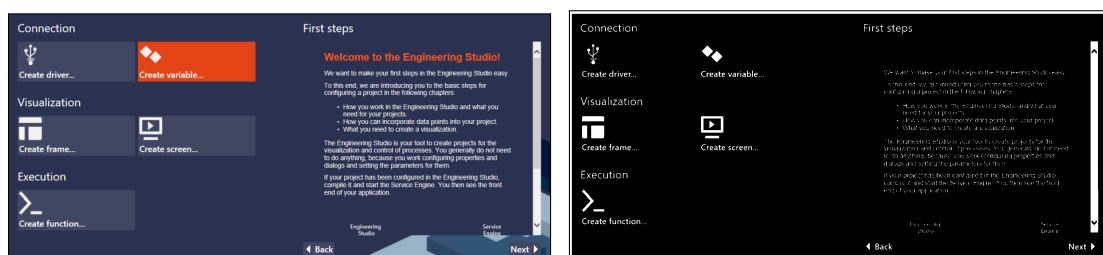
##### Feste Schwellenwertmethode

Ein häufig für die Bildsegmentierung genutztes Verfahren ist die feste Schwellenwertmethode, auf Englisch auch „Fixed Thresholding“ genannt. Bei diesem Bildverarbeitungsverfahren wird ein manuell vordefinierter Grenzwert auf einzelne Pixelwerte angewandt. Liegt der Pixelwert über dem festgelegten Schwellenwert, gilt er als Teil des Vordergrunds, andernfalls als Hintergrund. Somit können Objekte, also die einzelnen Buchstaben in den Grafikdateien, von ihrem Hintergrund getrennt werden.

Das fixe Thresholding benötigt durch seine Simplizität einen relativ geringen Berechnungsaufwand und weist daher eine hohe Performance auf. Allerdings ergibt es sich oft, dass die eigentlich bunten grafischen Elemente der Benutzeroberfläche aufgrund ihrer Helligkeit über dem Schwellenwert liegen. Dadurch werden sie, genau wie der Text, als Vordergrund wahrgenommen und die gesamte Fläche wird einfarbig. Somit kann jeglicher Text innerhalb dieser Fläche nicht vom Texterkennungssystem erkannt werden und die Qualität und Menge der erkannten Textdaten wird stark reduziert.



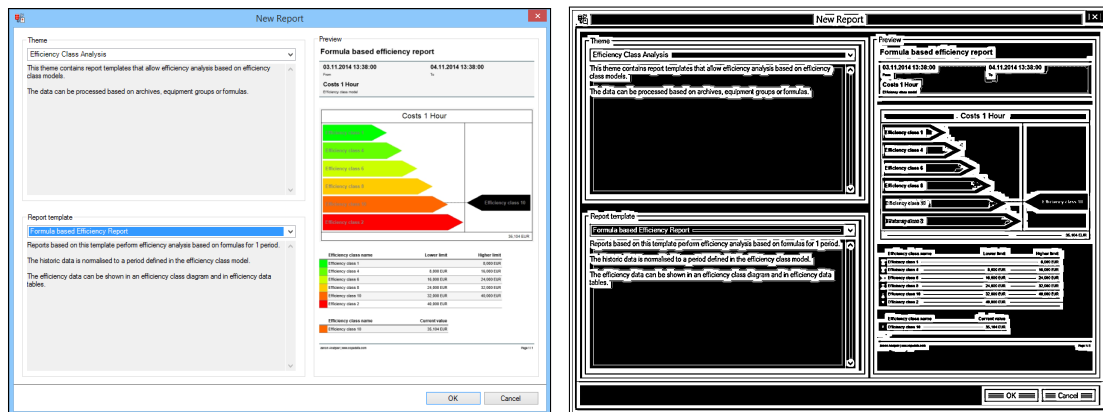
**Abbildung 3.6:** Anwendung des festen Schwellenwertverfahrens auf einen Beispielscreenshot. Bei einem passenden Schwellenwert und nur geringfügig verschiedenen Farbflächen ist der Textinhalt gut vom Hintergrund abgrenzbar. Der Schwellenwert im gezeigten Bild beträgt 60 %.



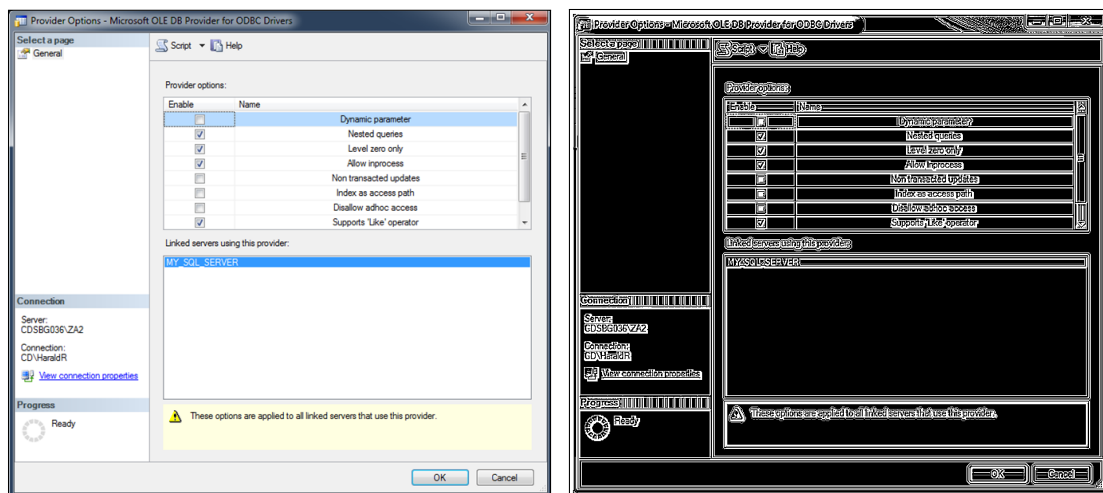
**Abbildung 3.7:** Anwendung des festen Schwellenwertverfahrens auf einen Beispielscreenshot. Bei einem falsch gewählten Schwellenwert oder komplexen UI-Elementstrukturen ist der Text nicht erkennbar. Der Schwellenwert im gezeigten Bild beträgt 80 %.

### Adaptive Schwellenwertmethode

Die adaptive Schwellenwertmethode gehört zu den halbautomatischen Schwellenwertalgorithmen. Bei diesem Verfahren wird der Schwellenwert auf Basis der lokalen Eigenschaften eines Bildbereichs angepasst, der durch die sogenannte „Blockgröße“ definiert wird. Innerhalb eines Blocks wird schließlich dynamisch ein fixer Schwellenwert ermittelt. Dadurch können im Gegensatz zur festen Schwellenwertmethode verschiedenfarbige Texte auf Hintergründen unterschiedlicher Helligkeit besser abgegrenzt werden und die Menge an erkanntem Text wird erhöht.



**Abbildung 3.8:** Anwendung der adaptiven Schwellenwertmethode auf einen Beispielscreenshot. Die Blockgröße ist gut an den Bildinhalt angepasst und alle Details bleiben erhalten. Dieses Verfahren punktet hier besonders bei den Farbigen „Energy Labels“, deren Textinhalte sonst mittels keinem anderen Verfahren komplett erkannt wurden.

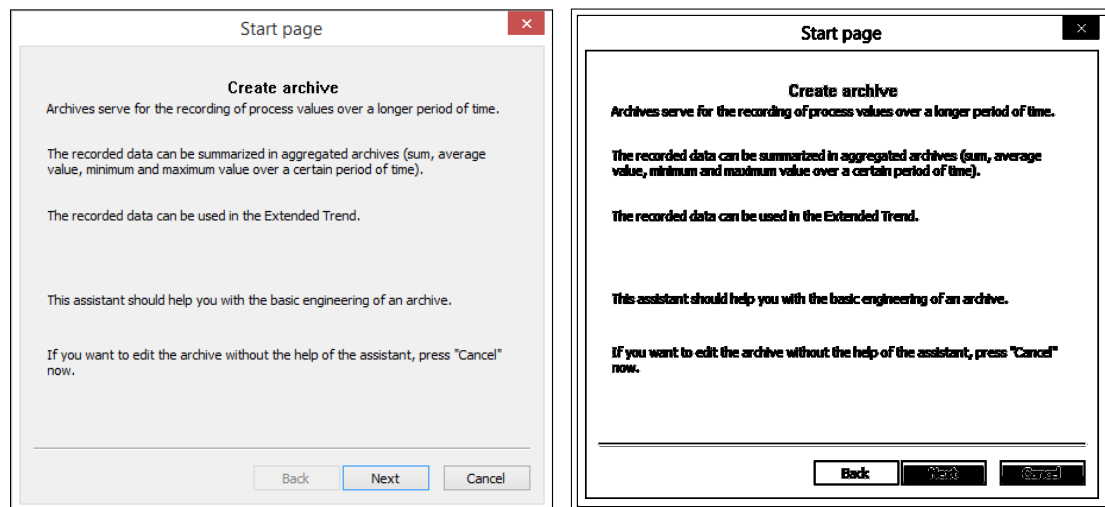


**Abbildung 3.9:** Anwendung der adaptiven Schwellenwertmethode auf einen Beispielscreenshot. Die bei unangepasster Blockgröße entstehenden Artefakte schränken die Funktionsweise des Texterkennungssystems deutlich ein.

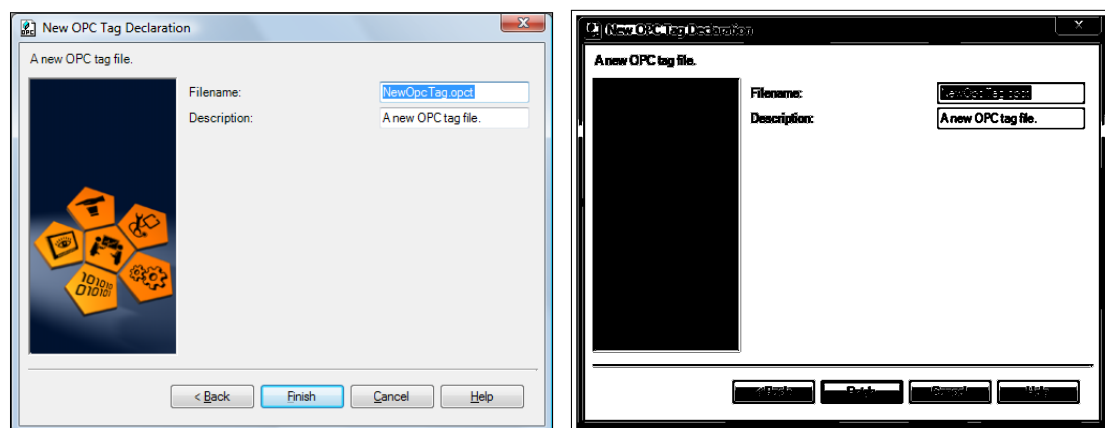


### Dreiecks-Schwellenwertmethode

Das Dreiecks-Schwellenwertverfahren verwendet das Histogramm eines Bildes, um einen globalen Schwellenwert zu ermitteln. Innerhalb des Histogramms wird eine Linie vom Höchstwert (engl. "Peak") zum Minimum gezeichnet und ermittelt die Normale mit der maximalen Länge. Dieses Verfahren erzielt die besten Ergebnisse, wenn die zu extrahierenden Elemente Intensitätswerte aufweisen, die an der Basis des ermittelten Peaks liegen. Für Screenshots von UI-Elementen mit komplexer Struktur und farblich stark variierenden Komponenten ist es eher nicht geeignet.



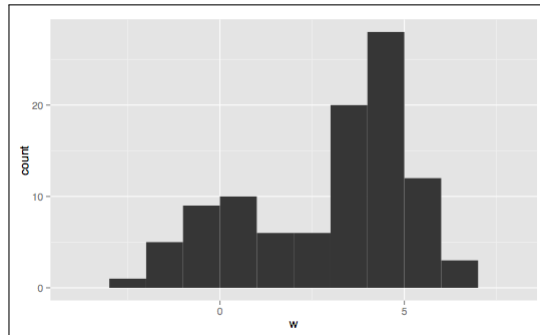
**Abbildung 3.10:** Anwendung der Dreiecks-Schwellenwertmethode auf einen Beispielscreenshot. Gleichen sich die Inhalte farblich, werden einige Details extrahiert. Durch kleinste Farbvariationen im Bild weicht der Schwellenwert jedoch vom Optimum ab und die Texte sind nur schwer zu erkennen.



**Abbildung 3.11:** Anwendung der Dreiecks-Schwellenwertmethode auf einen Beispielscreenshot. Bereits bei mäßiger Variation der Helligkeit verschwinden die ersten Details.

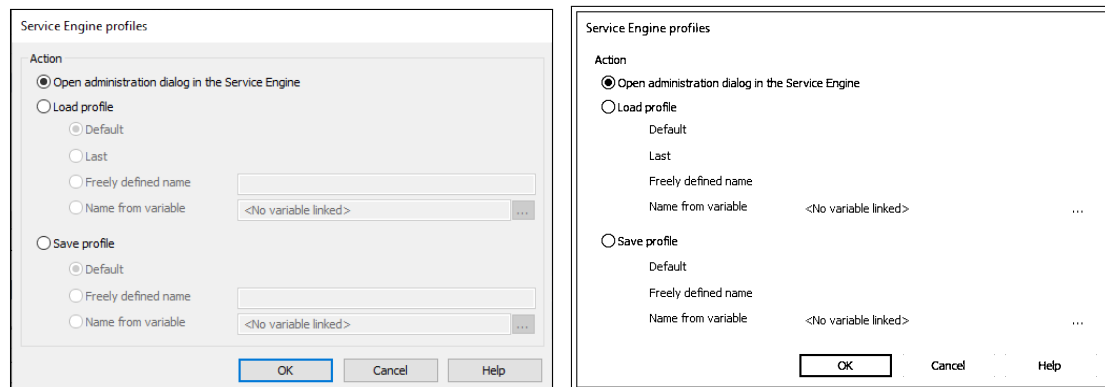
### Schwellenwertmethode nach Otsu

Bei dem Schwellenwertverfahren nach Otsu wird der globale Schwellenwert für ein Bild anhand des jeweiligen Histogramms ermittelt. Aufgrund dieser Eigenschaften funktioniert das Verfahren am besten, wenn das Histogramm des Bildes eine bimodale Verteilung aufweist, also zwei klare Spitzen hat.



**Abbildung 3.12:** Beispiel eines bimodalen Histogramms [24]

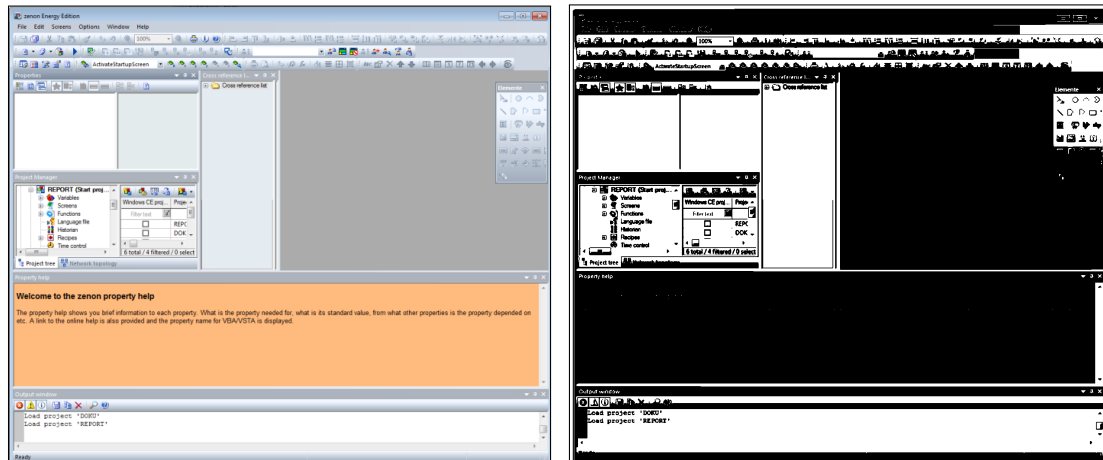
Enthält ein Bild jedoch starkes Hintergrundrauschen, funktioniert die Schwellenwertermittlung nur unzuverlässig. Weist es lokale Helligkeitsunterschiede auf, wie es bei grafischen Oberflächen mit ihren unterschiedlich eingefärbten Oberflächensektionen oft der Fall ist, entstehen Dank der Bestimmung eines einzelnen globalen Wertes für das gesamte Bild ähnliche Probleme wie bei der fixen Schwellenwertmethode.



**Abbildung 3.13:** Anwendung der Schwellenwertmethode nach Otsu auf einen Beispielscreenshot. Wird ein passender Schwellenwert ermittelt, lässt sich der Text gut vom Hintergrund trennen.

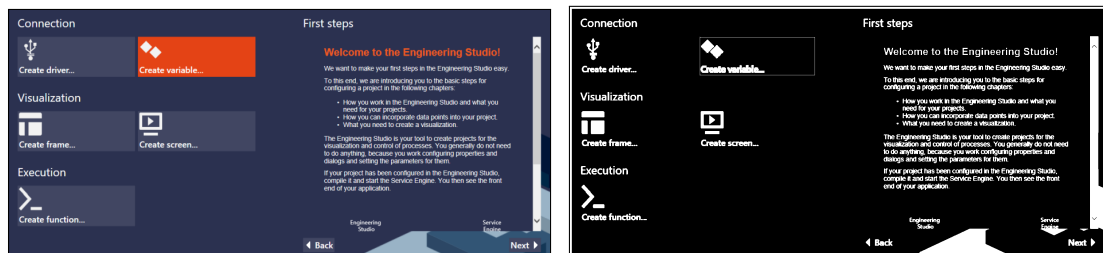
### Schwellenwertmethode nach Kapur

Die Schwellenwertmethode nach Kapur, Sahoo und Wong zielt darauf ab, einen Schwellenwert zu finden, der die Entropie zwischen den Vorder- und Hintergrundregionen maximiert.



**Abbildung 3.14:** Anwendung der Schwellenwertmethode nach Otsu auf einen Beispiel-screenshot. Bei komplexen Strukturen im User-Interface gehen aufgrund des globalen Schwellenwerts Details verloren.

Dieses Schwellenwertverfahren liefert gute Ergebnisse bei Bildern mit starker Varianz der Vorder- und Hintergrundkontraste bzw. einer breiten Helligkeitsverteilung.



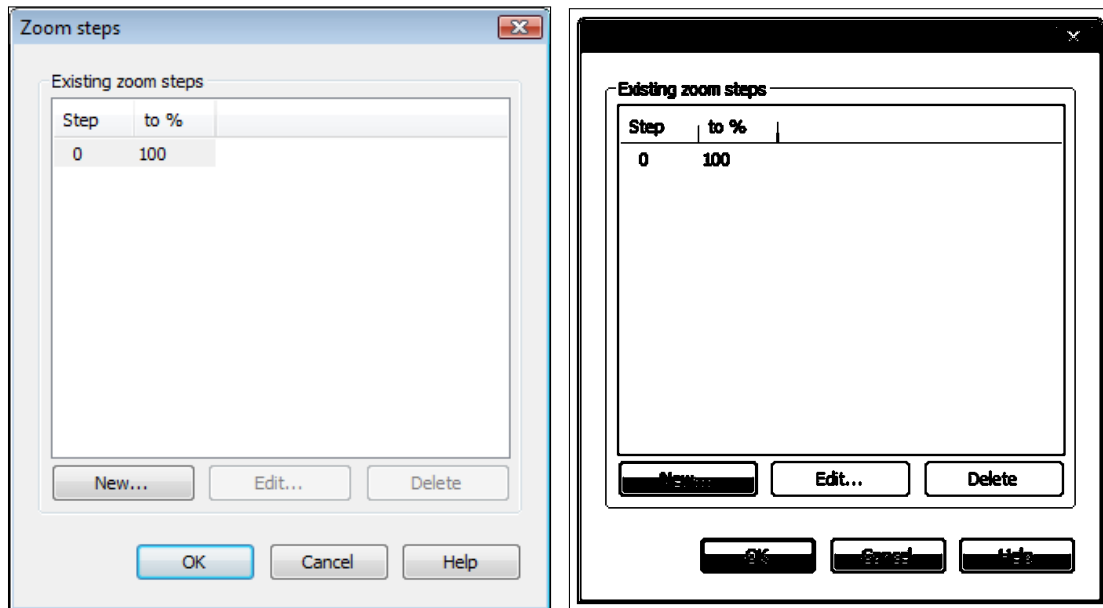
**Abbildung 3.15:** Anwendung der Schwellenwertmethode nach Kapur auf einen Beispiel-screenshot. Trotz der vielen verwendeten Farben wird der Inhalt gut dargestellt.

### 3.3.2 Postprocessing

Da die verarbeiteten Bilddaten bzw. deren extrahierte Textdaten später durch eine Schlagwort-basierte Suchfunktion durch den Nutzer auffindbar sein müssen, müssen die Ergebnisdaten im Rahmen des Postprocessings weiterverarbeitet werden. Ziel ist es, die Redundanz innerhalb des Datensets zu reduzieren. Ebenso sollen falsch erkannte Ergebnisdaten identifiziert und aus der Schlagwortmenge entfernt werden.

#### 3.3.2.1 Filterung anhand der Genauigkeit

Tesseract stellt im Rahmen der Texterkennung auch immer Metadaten zu den erkannten Texten zur Verfügung. Bei erkannten Wörtern wird beispielsweise immer eine Genauigkeit (engl. "Confidence") mit angegeben. Sie bestimmt, mit welcher Sicherheit ein Texterkennungssystem das jeweilige Wort erkannt hat, wobei Wörter mit hoher Confidence



**Abbildung 3.16:** Anwendung der Schwellenwertmethode nach Kapur auf einen Beispielscreenshot. Trotz der eigentlich einfach erscheinenden Oberfläche erzeugt das Verfahren Artefakte, die die Texterkennung deutlich erschweren.

eher richtig, mit niedriger Confidence eher falsch erkannt wurden.

Der Confidence-Filter prüft die jeweiligen Wörter auf ihre Metadaten und verwirft Wörter, deren Confidence unter einem fixen Schwellenwert liegt. Je nach Einstellung bzw. „Härte“ des Filters wird die Anzahl der falsch erkannten Inhalte innerhalb der Schlagwortmenge drastisch reduziert. Ist der Filter zu streng eingestellt, werden jedoch insgesamt weniger Worte in die Ergebnisse mit aufgenommen und es kann vorkommen, dass auch ursprünglich korrekt erkannte Wörter aufgrund eines niedrigen Confidence-Wertes verworfen werden.

```

2  {
3    "Text": "#:",
4    "Confidence": 68.34635162353516
5  },
6  {
7    "Text": "Variablendefinition",
8    "Confidence": 91.65243530273438
9  },
10 {
11   "Text": "BaER",
12   "Confidence": 0
13 },
14 {

```

```

2  {
3    "Text": "#:",
4    "Confidence": 68.34635162353516
5  },
6  {
7    "Text": "Variablendefinition",
8    "Confidence": 91.65243530273438
9  },
10 {
11   "Text": " ",
12   "Confidence": 95
13 },
14 {

```

**Abbildung 3.17:** Auszug aus den Ergebnisdaten der Texterkennung aus 3.3 nach der Confidence-Filterung. Alle Wörter unter dem Schwellenwert werer entfernt.

### 3.3.2.2 Normalisierung

Um die aus der Texterkennung gewonnenen Daten zunächst für die weitere Filterung vorzubereiten, ist es sinnvoll, die Redundanz der Daten möglichst zu reduzieren und die einzelnen Wörter zu normalisieren bzw. zu standardisieren. Beispielsweise kann durch das Umwandeln aller Textdaten in Kleinbuchstaben die Variation der Daten eingeschränkt werden, ohne jedoch für die Suche relevante Information zu verlieren.

2	"#:",	2	"#:",
3	"Variablendefinition",	3	"variablendefinition",
4	" ",	4	" ",
5	" ",	5	" ",
6	" ",	6	" ",
7	" ",	7	" ",
8	" ",	8	" ",
9	" ",	9	" ",
10	" ",	10	" ",
11	" ",	11	" ",
12	" ",	12	" ",
13	" ",	13	" ",
14	"Selektieren",	14	"selektieren",

**Abbildung 3.18:** Auszug aus den Ergebnisdaten der Texterkennung aus 3.3 nach der Normalisierung. Alle Wörter beinhalten nun ausschließlich Kleinbuchstaben.

### 3.3.2.3 Vermeidung von Duplikaten

Nach der Normalisierung werden Duplikate innerhalb der erkannten Textdaten entfernt. Dies verringert ebenfalls die Redundanz der Textdaten und steigert die Effizienz der nachfolgenden Filterverfahren.

2	"#:",	2	"bit",
3	"variablendefinition",	3	"float",
4	" ",	4	"sie",
5	" ",	5	"wort",
6	" ",	6	"byte",
7	" ",	7	"gewünschte",
8	" ",	8	"das",
9	" ",	9	"string",
10	" ",	10	" ",
11	" ",	11	" ",
12	" ",	12	" ",
13	" ",	13	" ",
14	"selektieren",	14	"doppelwort",

**Abbildung 3.19:** Auszug aus den Ergebnisdaten der Texterkennung aus 3.3 nach der Duplikatentfernung.

### 3.3.2.4 Filterung anhand der Wortlänge

Verarbeitet das Texterkennungssystem Texte mit unregelmäßigen Abständen oder grafischen Artefakten in der Schrift, werden statt des eigentlichen Wortes fälschlicherweise oft relativ kurze Symbolkombinationen erkannt. Um diese Kombinationen aus den Ergebnisdaten zu entfernen, können Zeichenketten mithilfe des Wortlängenfilters ungeachtet ihres Inhaltes verworfen werden.

Zusätzlich kann dieser Filter an die Anforderung des Zielsystems angepasst werden. So sind beispielsweise Wörter, die weniger als zwei Zeichen beinhalten, für die Schlagwortsuche grundsätzlich nicht relevant.

2	"#:",	2	"bit",
3	"variablendefinition",	3	"float",
4	" ",	4	"sie",
5	" ",	5	"wort",
6	" ",	6	"byte",
7	" ",	7	"gewünschte",
8	" ",	8	"das",
9	" ",	9	"string",
10	" ",	10	" ",
11	" ",	11	" ",
12	" ",	12	" ",
13	" ",	13	" ",
14	"selektieren",	14	"doppelwort",

**Abbildung 3.20:** Auszug aus den Ergebnisdaten der Texterkennung aus 3.3 nach der Wortlängenfilterung. Alle Wörter, die kürzer sind als der Schwellenwert, werden aus den Ergebnisdaten entfernt.

### 3.3.2.5 Sprachabhängige Filterung mittels Regular Expressions

Nachdem die zu filternden Textdaten durch vorherige Schritte weitestgehend vorverarbeitet wurden, werden die Ergebnisdaten ein letztes Mal mithilfe von regulären Ausdrücken (engl. "Regular Expressions") durchsucht. Aufgrund der guten Erweiterbarkeit der Regular Expressions ist es einfach, für jede Sprache einen individuellen Filter anzulegen, der den jeweiligen Zeichensatz beschriftet und unbekannte Sonderzeichen oder Symbole entfernt. So sind beispielsweise im Deutschen Umlaute erlaubt, während häufig auftretende, jedoch unerwünschte Symbole wie das phonetische Zeichen „æ“ oder mehrere hintereinandergereihte Leerzeichen explizit entfernt werden können.

2	"bit",	2	"bit",
3	"float",	3	"float",
4	"sie",	4	"sie",
5	"wort",	5	"wort",
6	"byte",	6	"byte",
7	"gewünschte",	7	"gewünschte",
8	"das",	8	"das",
9	"string",	9	"string",
10	" ",	10	"doppelwort",
11	"  ",	11	"treibervariable",
12	"   ",	12	"variablendefinition",
13	"    ",	13	"selektieren",
14	"doppelwort",	14	"archive",

**Abbildung 3.21:** Auszug aus den Ergebnisdaten der Texterkennung aus 3.3 nach der Filterung mit Regular Expressions. Findet der sprachabhängige Filter keine Treffer, wird das Wort aus den Ergebnisdaten entfernt.

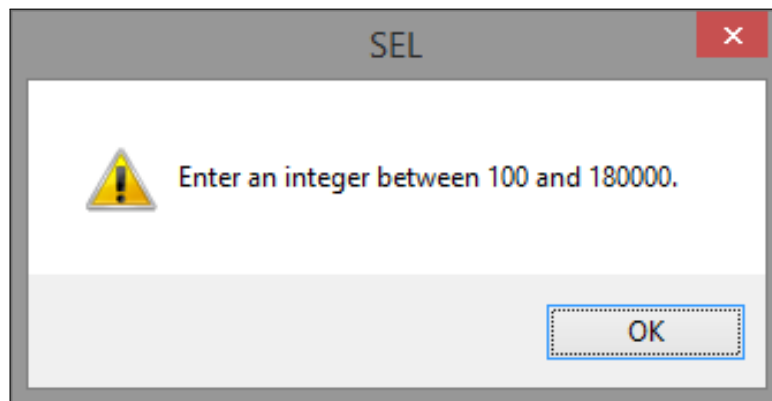
## Kapitel 4

# Durchführung

### 4.1 Implementierung

#### 4.1.1 Vergleichsdaten

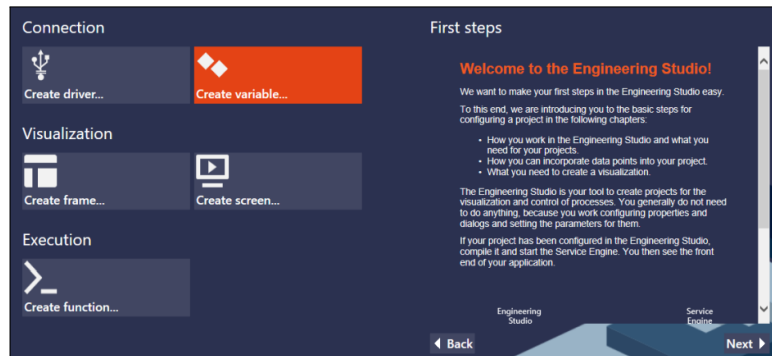
Als Ausgangsdaten für die Durchführung wurde eine zufällige Auswahl an Dokumentationsscreenshots getroffen. Zusätzlich wurden auch Bilder in die Stichprobe mit aufgenommen, die beispielsweise aufgrund ihrer Auflösung oder Kontrastverhältnisse schwer lesbar sind.



**Abbildung 4.1:** Ein gut für die Texterkennung geeigneter Screenshot. Die wesentlichen Inhalte weisen einen guten Kontrast zum Hintergrund auf und befinden sich in Bereichen mit gleichmäßiger Helligkeit.

Die textuellen Inhalte aller ausgewählten Bilder wurden anschließend manuell extrahiert und für den programmatischen Vergleich in einer Datei im selben Verzeichnis wie die Quellbilddatei festgehalten.





**Abbildung 4.2:** Ein schlecht lesbarer Screenshot. Aufgrund der vielen Symbole und der bunten Flächen stellt dieses Bild eine Herausforderung für das Texterkennungssystem dar.



**Abbildung 4.3:** Ein Screenshot und die daraus manuell extrahierten Schlagworte.

#### 4.1.2 Verwendete Bibliotheken

##### 4.1.2.1 Fremdbibliotheken

In der prototypischen Implementierung, geschrieben in der Programmiersprache C#, wurden in Referenz an die in Abschnitt 2.2 vorgestellten Technologien und Werkzeuge folgende NuGet-Bibliotheken als Basis für die Implementierung verwendet:

- **Magick.NET**  
Version: 13.1.0  
Lizenz: Apache-2.0

<https://www.nuget.org/packages/Magick.NET.Core>

- **Tesseract**

Version: 5.2.0

Lizenz: Apache-2.0

<https://www.nuget.org/packages/Tesseract>

#### 4.1.2.2 Verarbeitungsketten

Beim Entwurf des Verarbeitungssystems für die unterschiedlichen Bild- und Textverarbeitungsschritte wurde bewusst auf Flexibilität geachtet. Mithilfe von Interfaces und Builder-Methoden ist es möglich, Verarbeitungsschritte als Prozessoren (engl. "Processors") zu definieren. So stellt beispielsweise das Normalisieren eines durch Tesseract erkannten Wortes einen Verarbeitungsschritt dar.

```
public interface IProcessor
{
    IEnumerable Process(IEnumerable inputs);
}

public interface IProcessor<in TInput, out TOutput> : IProcessor
{
    IEnumerable<TOutput> Process(IEnumerable<TInput> inputs);

    new IEnumerable Process(IEnumerable inputs)
    {
        return Process((IEnumerable<TInput>)inputs);
    }
}
```

**Programm 4.1:** Auszug aus Datei „IProcessor.cs“: Schnittstelle eines Prozessors.

```
public class ToLowerProcessor
: Processor<ScanResult, ScanResult>
{
    public override IEnumerable<ScanResult> Process(
        IEnumerable<ScanResult> inputs
    )
    {
        foreach (var kv in inputs)
        {
            kv.Word.Text = kv.Word.Text.ToLower();
            yield return kv;
        }
    }
}
```

**Programm 4.2:** Auszug aus Datei „ToLowerProcessor.cs“: Normalisieren als einzelner Verarbeitungsschritt

Sollen mehrere Schritte verbunden werden, ist es möglich, bietet das Processing-Framework die Möglichkeit, eine sogenannte Verarbeitungskette aufzubauen. Hier können Delegates oder komplette Prozessoren dynamisch als einzelne Schritte aneinanderge-

reicht werden, wobei die Typensicherheit durch das generische Typensystem von C# stets gewahrt wird:

```
public interface IProcessorChainConfiguration<TInput, TOutput> : IProcessorChain<
    TInput, TOutput>
{
    IProcessorChainConfiguration<T, TOutput, TInput, TOutput> Use<T>(
        IProcessor<TInput, T> processor);

    IProcessorChainConfiguration<T, TOutput, TInput, TOutput> Use<T>(
        Func<IEnumerable<TInput>, IEnumerable<T>> processorFunc);

    IProcessorChainConfiguration<TInput, TOutput> Complete(
        IProcessor<TInput, TOutput> processor);

    IProcessorChainConfiguration<TInput, TOutput> Complete(
        Func<IEnumerable<TInput>, IEnumerable<TOutput>> processorFunc);
}

public interface IProcessorChainConfiguration<TInput, TOutput, TInChain, TOutChain>
{
    IProcessorChainConfiguration<T, TOutput, TInChain, TOutChain> Use<T>(
        IProcessor<TInput, T> processor);

    IProcessorChainConfiguration<T, TOutput, TInChain, TOutChain> Use<T>(
        Func<IEnumerable<TInput>, IEnumerable<T>> processorFunc);

    IProcessorChain<TInChain, TOutChain> Complete(
        IProcessor<TInput, TOutput> processor);

    IProcessorChain<TInChain, TOutChain> Complete(
        Func<IEnumerable<TInput>, IEnumerable<TOutput>> processorFunc);
}
```

**Programm 4.3:** Auszug aus Datei „IProcessorChainConfiguration.cs“: Schnittstelle zur Konfiguration einer Verarbeitungskette

Ist die Aufbauphase abgeschlossen, kann die Verarbeitungskette schlussendlich gestartet werden.

```
var postProcessor = new ProcessorChainConfiguration<ScanResult, ScanResult>()
    .Use(new ConfidenceFilter(50))
    .Use(new ToLowerProcessor())
    .Use(new DuplicateFilter())
    .Complete(new RegexFilter(WordRegex));

// ...

postProcessor.Process(data);
```

**Programm 4.4:** Auszug aus Datei „ImageViewModel.cs“: Konfiguration und Starten einer Verarbeitungskette

Abhängig von den verwendeten Prozessoren können also Eingangsdaten jeglichen Typs, in diesem Fall Bildobjekte der Magick.NET Bibliothek oder Ergebnisdaten des Texter-

kennungsvorgangs dynamisch verarbeitet werden.

#### Bildverarbeitungskette

Für den grundlegenden Ablauf der Bildverarbeitung und der anschließenden Ergebnisfilterung werden die Erkenntnisse aus Kapitel 3 mithilfe des in Unterunterabschnitt 4.1.2.2 beschriebenen Processing-Frameworks angewandt.

Angefangen mit einem Ausgangsbild, welches über die Softwarebibliothek Magick.NET geladen wurde, beginnt die Bildverarbeitung zunächst mit dem Resampling. Falls der geladene Screenshot die Mindestauflösung von 300dpi unterschreitet, wird es mittels Lanczos2-Verfahren, eine von Magick.NET mitgelieferte Implementierung des Lanczos2-Algorithmus mit leichter Schärfung [20], auf die Mindestauflösung vergrößert. Anschließend wird das Bild normalisiert, in Graustufen umgewandelt und jegliche Transparenz durch einen weißen Hintergrund ersetzt. Danach wird es mittels Schwellwertverfahren binarisiert und rund um das Bild wird ein Rahmen mit einer Dicke von 10px eingefügt. Um Texterkennungsfehler durch falsche Vorder- bzw. Hintergrundfarben auszuschließen, wird das Bild schlussendlich gemeinsam mit einer farblich invertierten Version an das Texterkennungssystem weitergegeben.

```
var preprocessing = new ProcessorChainConfiguration<MagickImage, MagickImage>()
    .Use(new CloneImageProcessor())
    .Use(new ResizeProcessor(FilterType.Lanczos2Sharp, PixelInterpolateMethod.Mesh))
    .Use(new NormalizeProcessor())
    .Use(_thresholdProcessor)
    .Use(new AddBorderProcessor(10))
    .Use(new BinarizeProcessor())
    .Use(new NegateCloneProcessor())
    .Complete(OnPreprocessed);
```

**Programm 4.5:** Auszug aus Datei „EvaluationProcessor.cs“: Definition der Preprocessing-Kette

Wurde der übergebene Screenshot vom Texterkennungssystem verarbeitet, müssen nun die Ergebnisse gefiltert werden. Dazu werden zunächst die Metadaten der einzelnen Wörter betrachtet und alle Elemente mit einer Confidence unter einem Schwellenwert von 50% verworfen. Danach werden die erkannten Texte mittels der C#-Funktion ToLower() normalisiert und anschließend auf Duplikate untersucht. Sind alle Duplikate verworfen, werden die Wörter schlussendlich mittels sprachabhängigen Regular Expressions – in diesem Fall gibt es gemäß den Annahmen in Abbildung 3.1 einen Sprachfilter für Englisch und Deutsch – gefiltert.

```
var postprocessing = new ProcessorChainConfiguration<ScanResult, ScanResult>()
    .Use(new ConfidenceFilter(50))
    .Use(new ToLowerProcessor())
    .Use(new DuplicateFilter())
    .Complete(new RegexFilter(wordRegex));
```

**Programm 4.6:** Auszug aus Datei „EvaluationProcessor.cs“: Definition der Postprocessing-Kette

#### 4.1.2.3 Lookup

Die „Lookup“ Bibliothek abstrahiert das Speichern von Schlüssel-Wert-Paaren. Dabei kann flexibel zwischen verschiedenen Speicherimplementierungen gewechselt werden. So ist es beispielsweise möglich, die Werte in einer Listenstruktur im Arbeitsspeicher, in einer Datei oder – mittels der EntityFramework-Bibliothek, welche von der .NET Foundation entwickelt wird – in einer Datenbank persistent abzulegen.

Unabhängig von der Ablagestruktur im Hintergrund können Lookups mittels einer gemeinsamen Schnittstelle manipuliert werden:

```
public interface ILookup<TKey, TValue>
: ILookup,
IDictionary<TKey, ICollection<TValue>>,
IDisposable
{
    ICollection<TValue> Add(TKey key);

    public void Add(TKey key, TValue value);

    public void AddRange(TKey key, IEnumerable<TValue> values);

    public bool Remove(TKey key, TValue value);

    public ICollection<TValue> GetOrAdd(TKey key);
}
```

**Programm 4.7:** Auszug aus Datei „ILookup.cs“: Definition der gemeinsamen Schnittstelle für Lookups

#### 4.1.2.4 OCR

Die „OCR“ Bibliothek enthält elementare Komponenten für die Texterkennung mittels der oben genannten Komponenten. Sie enthält Funktionen zur Bearbeitung von Bildern mittels Magick.NET inklusive anschließender Verarbeitung mittels Tesseract. Kernkomponenten wie das Texterkennungssystem werden automatisch auf Basis der Eingabeparameter konfiguriert und die Verwendung der Ergebnisdaten in externen Programteilen wird durch die Zurverfügungstellung von Datenmodellen für die Ergebnisdaten vereinfacht. Außerdem enthält die Bibliothek eine Reihe von vordefinierten Verarbeitungsketten bzw. Prozessoren für die Bild- und Textverarbeitung.

#### 4.1.2.5 Automatische Berichterstellung

Mithilfe des ReportGenerator-Frameworks wird die automatische Berichterstellung für unterschiedlichste Ausgabeformate abstrahiert. Durch die mitgelieferten Schnittstellendefinitionen ist es möglich, eigene Ausgabeformate zu definieren und den Funktionsumfang des ReportGenerators, wie beispielsweise das Erstellen von Tabellen oder das Anlegen und Überschriften, an die jeweilige Syntax und Dokumentstruktur anzupassen.

```
public interface IDocumentGenerator : IStreamWriter
{
```

```

IDocumentGenerator Append(string? text = default);

IDocumentGenerator AppendLine(string? text = default);

IDocumentGenerator AppendParagraph(string? text = default);

IDocumentGenerator AppendHeading(int level, string text);

IDocumentGenerator AppendTable(int columns, Action<ITableGenerator> table);

string FormatImage(string path, IBounds? bounds = default);
}

```

**Programm 4.8:** Auszug aus Datei „IDocumentGenerator.cs“: Hauptschnittstelle für den ReportGenerator

### 4.1.3 Programmablauf

Die prototypische Implementierung besteht neben den oben genannten Komponenten aus einem ausführbaren Kommandozeilenprogramm zur Texterkennung und einem Programm zum Vergleich der Ergebnisse mit den manuell verschlagworteten Soll-Daten. Alle relevanten Daten werden in entsprechenden Ausgabeverzeichnissen festgehalten und dadurch für den jeweiligen nächsten Schritt verfügbar gemacht.

#### 4.1.3.1 Texterkennung

Zu Beginn der Ausführung des Kommandozeilenprogramms wird für jedes zu verarbeitende Bild abhängig von den definierten Schwellenwertverfahren eine Reihe von Prozessoren angelegt. Dazu wurde der statische Teil der Bildverarbeitungskette gemäß 4.4 innerhalb der „EvaluationProcessor“ Klasse definiert. Lediglich die zu evaluierenden Prozessoren für die jeweiligen Schwellenwertverfahren können außerhalb der Klasse dynamisch definiert werden. Der EvaluationProcessor legt die erzeugten Ergebnisdaten, bestehend aus den gefundenen Wörtern und zugehörigen Metadaten wie die Confidence, auf Dateiebene ab. Um überprüfen zu können, welches Bild schlussendlich an das Texterkennungssystem übergeben wurde, werden auch die verarbeiteten Bilder nach der Binarisierung gespeichert.

```

private static IEnumerable<EvaluationProcessor> MakeThresholdVariations()
{
    for (int i = 4; i <= 24; i += 4)
    {
        yield return new(new ThresholdAdaptiveProcessor(i));
    }

    for (int i = 20; i <= 80; i += 10)
    {
        yield return new(new ThresholdProcessor(i));
    }

    yield return new(new AutoThresholdProcessor(AutoThresholdMethod.Kapur));
    yield return new(new AutoThresholdProcessor(AutoThresholdMethod.OTSU));
    yield return new(new AutoThresholdProcessor(AutoThresholdMethod.Triangle));
}

```

```
}
```

**Programm 4.9:** Auszug aus Datei „Program.cs“: Definition der Thresholding Prozessoren

Ist die Erstellung der Bildbearbeitungsprozessoren abgeschlossen, wird jeder einzelne Prozessor über die Systembibliothek „System.Threading.Tasks“ als eigener Auftrag (engl. "Task") gestartet. In der Kommandozeile wird anschließend der aktuelle Status jedes Tasks angezeigt. Wurden alle Tasks abgeschlossen, wird das Programm beendet.

#### 4.1.3.2 Vergleich mit Soll-Daten

Wurden die in den jeweiligen Screenshots erkannten Textdaten abgelegt, werden diese Daten im zweiten Kommandozeilenprogramm „ReportGenerator“ nun mit den manuell verschlagworteten Daten verglichen und die Ergebnisse in einen Bericht (engl. "Report") gespeichert.

Als zentrale Komponente für den Vergleich spielt die Berechnung der in Unterabschnitt 3.2.1 erklärten Metriken eine wesentliche Rolle. Wie in Programm 4.1.3.2 ersichtlich, wird die Distanz zwischen zwei C#-Enumerables, seien es zwei Strings oder zwei Listen, über das Verfahren nach Levenshtein berechnet.

```
public static double GetDistance<T>(T reference, T? hypothesis)
where T : IEnumerable
{
    var refArr = reference.Cast<object>().ToArray();
    var hypArr = hypothesis?.Cast<object>().ToArray() ?? Array.Empty<object>();

    var distance = new int[refArr.Length + 1, hypArr.Length + 1];

    for (var x = 0; x <= refArr.Length; x++)
    {
        distance[x, 0] = x;
    }

    for (var y = 0; y <= hypArr.Length; y++)
    {
        distance[0, y] = y;
    }

    for (var x = 0; x < refArr.Length; x++)
    {
        for (var y = 0; y < hypArr.Length; y++)
        {
            var cost = Equals(refArr[x], hypArr[y]) ? 0 : 1;

            var c1 = distance[x, y] + cost; // Bottom left

            var c2 = distance[x, y + 1] + 1; // Top left
            var c3 = distance[x + 1, y] + 1; // Bottom right

            distance[x + 1, y + 1] = Min(c1, c2, c3); // Top right
        }
    }
}
```

```
        return distance[refArr.Length, hypArr.Length];  
    }
```

**Programm 4.10:** Auszug aus Datei „Calculator.cs“: Berechnung der Levenshtein-Distanz

Nach der Ermittlung der jeweiligen Distanzen auf Wort- bzw. Bildbasis werden sie mit den jeweiligen Ursprungsbildern, Prozessoren und den verwendeten Algorithmen in Bezug gesetzt. Die so aufbereiteten Ergebnisse werden anschließend an den ReportGenerator übergeben und in einen Bericht zusammengefasst.

## 4.2 Analyse

Zeigen des erstellten Reports

Ausarbeiten des erstellten Reports



## Kapitel 5

# Zusammenfassung

Hier werden die Ergebnisse der Bachelorarbeit zusammengefasst und besprochen.

# Quellenverzeichnis

## Literatur

- [1] AMAM Asif u. a. „An overview and applications of optical character recognition“. *Int. J. Adv. Res. Sci. Eng* 3.7 (2014), S. 261–274 (siehe S. 3).
- [2] KR1442 Chowdhary und KR Chowdhary. „Natural language processing“. *Fundamentals of artificial intelligence* (2020), S. 603–649 (siehe S. 3).
- [3] Kenneth W. Church und Lisa F. Rau. „Commercial Applications of Natural Language Processing“. *Commun. ACM* 38.11 (Nov. 1995), S. 71–79. DOI: 10.1145/219717.219778 (siehe S. 4).
- [4] Line Eikvil. „Optical character recognition“. *citeseer.ist.psu.edu/142042.html* 26 (1993) (siehe S. 3).
- [5] Urvashi Gupta und Rohit Sharma. „Comparison of Different Cloud Computing Platforms for Data Analytics“. In: Sep. 2023, S. 67–78. DOI: 10.1007/978-981-99-3716-5\_7 (siehe S. 3).
- [6] Noor Islam Islam. „A survey on optical character recognition systems“. *arXiv preprint arXiv:1710.05703* (2017) (siehe S. 3).
- [7] Krishna Prakash Kalyanathaya, D Akila und P Rajesh. „Advances in natural language processing—a survey of current research trends, development tools and industry applications“. *International Journal of Recent Technology and Engineering* 7.5C (2019), S. 199–202 (siehe S. 4).
- [8] Romain Karpinski, Devashish Lohani und Abdel Belaid. „Metrics for complete evaluation of ocr performance“. In: *IPCV’18-The 22nd Int’l Conf on Image Processing, Computer Vision, & Pattern Recognition*. 2018 (siehe S. 7).
- [9] Lily Rojabiyati Mursari und Antoni Wibowo. „The effectiveness of image pre-processing on digital handwritten scripts recognition with the implementation of OCR Tesseract“. *Computer Engineering and Applications Journal* 10.3 (2021), S. 177–186 (siehe S. 5).
- [10] and others. „Binary codes capable of correcting deletions, insertions, and reversals“. In: (Siehe S. 7, 9).
- [11] Youngja Park u. a. „An empirical analysis of word error rate and keyword error rate.“ In: *Interspeech*. Bd. 2008. 2008, S. 2070–2073 (siehe S. 8).
- [12] Smith R. „An Overview of the Tesseract OCR Engine“. In: 2007. URL: <https://ieeexplore.ieee.org/document/4376991> (besucht am 12.06.2023) (siehe S. 4).

- [13] Dan Sporici, Elena Cuşnir und Costin-Anton Boiană. „Improving the accuracy of Tesseract 4.0 OCR engine using convolution-based preprocessing“. *Symmetry* 12.5 (2020), S. 715 (siehe S. 5).
- [14] Xiang Tong und David A Evans. „A statistical approach to automatic OCR error correction in context“. In: *Fourth workshop on very large corpora*. 1996 (siehe S. 8).
- [15] William Ughetta und Brian W. Kernighan. „The Old Bailey and OCR: Benchmarking AWS, Azure, and GCP with 180,000 Page Images“. English (US). In: *Proceedings of the ACM Symposium on Document Engineering, DocEng 2020*. Proceedings of the ACM Symposium on Document Engineering, DocEng 2020. Association for Computing Machinery, Inc, Sep. 2020. DOI: 10.1145/3395027.3419595 (siehe S. 3).
- [16] Ye-Yi Wang, Alex Acero und Ciprian Chelba. „Is word error rate a good indicator for spoken language understanding accuracy“. In: *2003 IEEE workshop on automatic speech recognition and understanding (IEEE Cat. No. 03EX721)*. IEEE. 2003, S. 577–582 (siehe S. 8).
- [17] *Amazon Textract - Pricing*. eng. 23. Mai 2023. URL: <https://aws.amazon.com/textract/pricing/> (besucht am 12.06.2023) (siehe S. 4).
- [18] *Azure AI Vision - Pricing*. eng. 23. Mai 2023. URL: <https://azure.microsoft.com/en-gb/pricing/details/cognitive-services/computer-vision/> (besucht am 12.06.2023) (siehe S. 4).
- [19] *Google Cloud Vision - Pricing*. eng. 23. Mai 2023. URL: <https://cloud.google.com/vision/pricing> (besucht am 12.06.2023) (siehe S. 4).
- [20] *ImageMagick Homepage*. eng. 23. Mai 2023. URL: <https://www.imagemagick.org/> (besucht am 12.06.2023) (siehe S. 4, 25).
- [21] *Tesseract Documentation*. eng. 23. Mai 2023. URL: <https://tesseract-ocr.github.io/> (besucht am 12.06.2023) (siehe S. 4, 10).
- [22] *Tesseract Repository*. eng. 23. Mai 2023. URL: <https://github.com/tesseract-ocr/tesseract> (besucht am 04.01.2024) (siehe S. 4).
- [23] *TODO: MISSING SOURCE*. eng. 23. Mai 2023. URL: <https://example.com/todo> (besucht am 04.01.2024) (siehe S. 10).

## Medien

- [24] Wikimedia Commons. *Example of a histogram exhibiting bimodality*. 2014. URL: <https://commons.wikimedia.org/wiki/File:Bimodal-histogram.png> (besucht am 12.06.2023) (siehe S. 15).