



Universidade Estadual de Campinas  
Instituto de Computação



Carla Doris Cardoso Cusihualpa

## Improving Text Recognition Accuracy using Syntax-based Techniques

Melhorando a Precisão do Reconhecimento de Texto  
usando Técnicas Baseadas em Sintaxe

CAMPINAS  
2020

**Carla Doris Cardoso Cusihualpa**

**Improving Text Recognition Accuracy using Syntax-based  
Techniques**

**Melhorando a Precisão do Reconhecimento de Texto usando  
Técnicas Baseadas em Sintaxe**

Dissertação apresentada ao Instituto de Computação da Universidade Estadual de Campinas como parte dos requisitos para a obtenção do título de Mestra em Ciência da Computação.

Dissertation presented to the Institute of Computing of the University of Campinas in partial fulfillment of the requirements for the degree of Master in Computer Science.

**Supervisor/Orientador: Prof. Dr. Guido Costa Souza de Araújo**  
**Co-supervisor/Coorientador: Dr. Marcio Machado Pereira**

Este exemplar corresponde à versão final da  
Dissertação defendida por Carla Doris  
Cardoso Cusihualpa e orientada pelo Prof.  
Dr. Guido Costa Souza de Araújo.

**CAMPINAS**  
**2020**

Ficha catalográfica  
Universidade Estadual de Campinas  
Biblioteca do Instituto de Matemática, Estatística e Computação Científica  
Ana Regina Machado - CRB 8/5467

C179i Cardoso Cusihualpa, Carla Doris, 1995-  
Improving text recognition accuracy using syntax-based techniques / Carla Doris Cardoso Cusihualpa. – Campinas, SP : [s.n.], 2020.

Orientador: Guido Costa Souza de Araújo.

Coorientador: Marcio Machado Pereira.

Dissertação (mestrado) – Universidade Estadual de Campinas, Instituto de Computação.

1. Visão por computador. 2. Reconhecimento de texto. I. Araújo, Guido Costa Souza de, 1962-. II. Pereira, Marcio Machado, 1959-. III. Universidade Estadual de Campinas. Instituto de Computação. IV. Título.

Informações para Biblioteca Digital

**Título em outro idioma:** Melhorando a precisão do reconhecimento de texto usando técnicas baseadas em sintaxe

**Palavras-chave em inglês:**

Computer vision

Text recognition

**Área de concentração:** Ciência da Computação

**Titulação:** Mestra em Ciência da Computação

**Banca examinadora:**

Guido Costa Souza de Araújo [Orientador]

Fábio Augusto Menocci Cappabianco

Ricardo da Silva Torres

**Data de defesa:** 26-03-2020

**Programa de Pós-Graduação:** Ciência da Computação

**Identificação e informações acadêmicas do(a) aluno(a)**

- ORCID do autor: <https://orcid.org/0000-0002-8011-2365>

- Currículo Lattes do autor: <http://lattes.cnpq.br/1663107015123141>



Universidade Estadual de Campinas  
Instituto de Computação



**Carla Doris Cardoso Cusihualpa**

## **Improving Text Recognition Accuracy using Syntax-based Techniques**

### **Melhorando a Precisão do Reconhecimento de Texto usando Técnicas Baseadas em Sintaxe**

#### **Banca Examinadora:**

- Prof. Dr. Guido Costa Souza de Araújo  
IC/UNICAMP
- Prof. Dr. Fábio Augusto Menocci Cappabianco  
Universidade Federal De São Paulo (UNIFESP)
- Prof. Dr. Ricardo da Silva Torres  
Institute of Computing - UNICAMP

A ata da defesa, assinada pelos membros da Comissão Examinadora, consta no SIGA/Sistema de Fluxo de Dissertação/Tese e na Secretaria do Programa da Unidade.

Campinas, 26 de março de 2020

# Acknowledgements

First of all, I thank God and Saint Ignatius of Loyola for giving me this opportunity in my life to improve personally and professionally.

I am thankful to my parents Sergio and Doris for their advice, unconditional support, patience, love and always being with me. Moreover, I thank my siblings Carola, Ignacio, and Camila, and my aunt Candy for always been there for me and being my best friends. To my grandmother Eulogia for taking care of my family and my grandparents Nicanor and María Esther, my angels who always take care of me.

Especially, I would like to thank Prof. Guido and co-adviser Prof. Marcio for their advice, guidance, experience, and knowledge which enabled me to finish this stage of my life.

I thank all my friends that I made during these two years, especially those from the Computer Systems Laboratory (LSC), for sharing their knowledge. I would also like to thank my boyfriend for helping me at this stage of my life and always being there when I need it. And especially to the IC secretary, for helping me with the processes at UNICAMP.

This work was supported by Samsung.

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001.

# Resumo

Devido à grande quantidade de informações visuais disponíveis atualmente, a detecção e o reconhecimento de texto em imagens de cenas naturais começaram a ganhar importância nos últimos tempos. Seu objetivo é localizar regiões da imagem onde há texto e reconhecê-lo. Essas tarefas geralmente são divididas em duas partes: detecção de texto e reconhecimento de texto. Embora as técnicas para resolver esse problema tenham melhorado nos últimos anos, o uso excessivo de recursos de hardware e seus altos custos computacionais impactaram significativamente a execução de tais tarefas em sistemas integrados altamente restritos (por exemplo, celulares e TVs inteligentes). Embora existam métodos de detecção e reconhecimento de texto executados em tais sistemas, eles não apresentam bom desempenho quando comparados às soluções de ponta em outras plataformas de computação. Embora atualmente existam vários métodos de pós-correção que melhoram os resultados em documentos históricos digitalizados, há poucas explorações sobre o seu uso nos resultados de imagens de cenas naturais. Neste trabalho, exploramos um conjunto de métodos de pós-correção, bem como propusemos novas heurísticas para melhorar os resultados em imagens de cenas naturais, tendo como base de prototipação o software de reconhecimento de textos Tesseract. Realizamos uma análise com os principais métodos disponíveis na literatura para correção dos erros e encontramos a melhor combinação que incluiu os métodos de substituição, eliminação nos últimos caracteres e composição. Somado a isto, os resultados mostraram uma melhora quando introduzimos uma nova heurística baseada na frequência com que os possíveis resultados aparecem em bases de dados de magazines, jornais, textos de ficção, web, etc. Para localizar erros e evitar *overcorrection* foram consideradas diferentes restrições obtidas através do treinamento da base de dados do Tesseract. Selecionamos como melhor restrição a **incerteza** do melhor resultado obtido pelo Tesseract. Os experimentos foram realizados com sete banco de dados usados em sites de competição na área, considerando tanto banco de dados para desafio em reconhecimento de texto e aqueles com o desafio de detecção e reconhecimento de texto. Em todos os bancos de dados, tanto nos dados de treinamento como de testes, os resultados do Tesseract com o método proposto de pós-correção melhorou consideravelmente em comparação com os resultados obtidos somente com o Tesseract.

# Abstract

Due to a large amount of visual information available today, Text Detection and Recognition in scene images have begun to receive an increasing importance. The goal of this task is to locate regions of the image where there is text and recognize them. Such tasks are typically divided into two parts: Text Detection and Text Recognition. Although the techniques to solve this problem have improved in recent years, the excessive usage of hardware resources and its corresponding high computational costs have considerably impacted the execution of such tasks in highly constrained embedded systems (e.g., cell-phones and smart TVs). Although there are Text Detection and Recognition methods that run in such systems they do not have good performance when compared to state-of-the-art solutions in other computing platforms. Although there are currently various post-correction methods to improve the results of scanned documents, there is a little effort in applying them on scene images. In this work, we explored a set of post-correction methods, as well as proposed new heuristics to improve the results in scene images, using the Tesseract text recognition software as a prototyping base. We performed an analysis with the main methods available in the literature to correct errors and found the best combination that included the methods of substitution, elimination in the last characters, and compounder. In addition, results showed an improvement when we introduced a new heuristic based on the frequency with which the possible results appear in the frequency databases for categories such as magazines, newspapers, fiction texts, web, etc. In order to locate errors and avoid *overcorrection*, different restrictions were considered through Tesseract with the training database. We selected as the best restriction the **certainty** of the best result obtained by Tesseract. The experiments were carried out with seven databases used in Text Recognition and Text Detection/Recognition competitions. In all databases, for both training and testing, the results of Tesseract with the proposed post-correction method considerably improved when compared to the results obtained only with Tesseract.

# List of Figures

1.1	Examples of various types of noise in document images: (a) Regular shaped nontextual noise. (b) Textual and irregular shaped nontextual noise. (Extracted from [5]) . . . . .	19
2.1	<i>Scene Text Detection and Recognition</i> problem. All the possible texts are found, the outputs being the sequence that characters. . . . .	21
2.2	Tesseract OCR 3.4.x engine architecture [50]. . . . .	25
2.3	The Tesseract input is a binary image focused on the text. . . . .	25
2.4	Block diagram of Tesseract word recognition [50]. . . . .	26
2.5	(a) The Tesseract input image. (b) How blobs represent the word in the image (a). . . . .	26
2.6	Segmentation graph. . . . .	27
2.7	Shrinking features: (a) Nanofeatures (b) Picofeatures [51]. . . . .	27
2.8	The Viterbi algorithm executed in the segmentation graph of Figure 2.5. Created words are “ <i>Analysis</i> ” and “ <i>malyn</i> ”. . . . .	29
2.9	(a) WERD_CHOICEs that were formed with the Viterbi algorithm. (b) The best chosen WERD_CHOICE considering the word with the lowest rating word. (c) Remove all WERD_CHOICE that are not in a reasonable range. (d) The final results. . . . .	30
2.10	DAWG for all forms of the verbs <i>play</i> and <i>work</i> . . . . .	31
2.11	Trie for all forms of the verbs <i>play</i> and <i>work</i> . Trie that represents the Dawg in Figure 2.10. . . . .	31
2.12	(a) Distribution of the 64 bits that represent an edge in DAWG in an English dictionary. (b) Representation of the edge to node 3 of the Figure 2.10. . . . .	31
3.1	Correction using the method proposed in [3]. . . . .	34
4.1	Edit distance with images from the ICDAR 2013 Database. Blue characters indicate substitutions, red characters deletions, and green characters insertions. . . . .	41
4.2	Annotated code for <i>word_in_dawg</i> method of Tesseract [?], that determines if the word is in the dictionary and a Dawg that represents the words: <i>are</i> , <i>play</i> , and <i>plays</i> . . . . .	42
4.3	Annotated code for <i>spelling_correction</i> method. . . . .	43
4.4	An image where <i>Libccv</i> library does not achieve good Text Detection. . . . .	43

4.5	Engine architecture Tesseract OCR 3.4.x with the new methods Compounder and Spelling Correction. . . . .	44
4.6	Annotated code for <i>NodeChildVector</i> structure of Tesseract. . . . .	44
4.7	Annotated code for <i>substitution</i> method. . . . .	45
4.8	Annotated code for <i>find_last_unichar</i> method. A Dawg that represents the words: <i>play</i> , <i>plays</i> , <i>plan</i> , and <i>place</i> . . . . .	46
4.9	Annotated code for <i>find_unichar</i> method to find all characters considering the next character of the word. . . . .	47
4.10	How the <i>find_unichar</i> method finds potential characters, analyzing the following character. . . . .	47
4.11	(a) Annotated code for <i>find__unichar</i> method considering all characters after the node. (b) A Dawg that represents the words: <i>play</i> , <i>plays</i> , <i>plan</i> , and <i>place</i> . . . . .	48
4.12	Annotated code for <i>insertion</i> method. . . . .	48
4.13	How the <i>insertion</i> method finds potential characters using the <i>find_unichar</i> method with analyzing the following character. . . . .	49
4.14	Annotated code for <i>elimination</i> method. . . . .	50
4.15	Annotated code for <i>compounder</i> method. . . . .	50
4.16	New distribution of the 64 bits that represent the same edge of Figure 2.12	53
4.17	Representation of the decimal number 2.7 in a float variable. . . . .	53
4.18	Representation of the decimal number 2.7 that it will store in the structure Dawg. . . . .	53
4.19	The code how of a decimal number is obtained to store in the structure of Tesseract dictionary. . . . .	54
4.20	The code how of a decimal number is obtained to store in the structure of Tesseract dictionary. . . . .	55
4.21	The problem of using different frequencies for the same word with different letter cases. . . . .	55
4.22	Image from the ICDAR2013 Database, where the ground truth is “ <i>Trespassing</i> ” and the result of Tesseract is “ <i>trespasslng</i> ”. . . . .	56
4.23	Dictionary formed by words “ <i>actually</i> ”, “ <i>hungry</i> ”, and “ <i>unruly</i> ” with their respective frequencies 95, 75, and 30. The left side shows how the dictionary uses the Trie structure. The right side shows how the dictionary uses the Dawg structure. . . . .	56
4.24	Tesseract with the <i>spelling_correction</i> method gives an incorrect result because the recognized word is not in the dictionary. . . . .	57
4.25	Annotated code for the <i>AcceptableResult</i> method of Tesseract. . . . .	57
4.26	Decision Tree with training data from the section databases 5.1. The <i>sklearn</i> library with <i>python</i> was used to obtain the Decision Tree. . . . .	59
4.27	Annotated code for the <i>spelling_correction_final</i> method. . . . .	59
5.1	Images cloud of the 4 databases. From top to bottom, from right to left: ICDAR 2003, ICDAR 2013, INCIDENTAL SCENE TEXT 2015, and IIT 5K - WORD. . . . .	62
5.2	Images cloud of the three databases. From top to bottom, from right to left: ICDAR 2015, The Street View Text (SVT), and KAIST Scene Text. . . . .	64

5.3	Examples of Text Recognition metrics. . . . .	65
5.4	Results obtained Tesseract with the substitution method. . . . .	70
5.5	Results obtained Libccv and Tesseract with the substitution method. . .	71
5.6	Problems where insertion can be used to correct errors. . . . .	73
5.7	Problems where deletion can be used to correct errors. . . . .	74
5.8	Results obtained Tesseract with the spelling_correction method. . . . .	81
5.9	Results obtained Libccv and Tesseract with the spelling_correction method. . . . . .	82
5.10	Results obtained Tesseract with the frequency method. . . . .	86
5.11	Results obtained Libccv and Tesseract with the frequency method. . . .	87
5.12	Results obtained Tesseract with the combined method. . . . .	92
5.13	Results obtained Libccv and Tesseract with the combined method. . . .	93

# List of Tables

3.1	Similarities and differences between the proposed method and those described above. . . . .	37
4.1	Number of words that failed in Scene Text Recognition. . . . .	41
4.2	How various dictionary of Latin and Germanic origin languages provided by Tesseract use the 64 bits of every edge. . . . .	52
5.1	Four databases for the Scene Text Recognition task. . . . .	62
5.2	Three databases for the Scene Text Detection and Recognition task. . . .	63
5.3	Results of the substitution method considering the analysis of the next character. . . . .	67
5.4	Results of the substitution method considering the analysis of all the characters. . . . .	67
5.5	Results of the substitution method considering the analysis of the next character with Tesseract restriction. . . . .	68
5.6	Results of the substitution method considering the analysis of all the characters with Tesseract restriction. . . . .	68
5.7	Results of the substitution method considering the analysis of the next character with certainty restriction. . . . .	69
5.8	Results of the substitution method considering the analysis of all the characters with certainty restriction. . . . .	69
5.9	Words corrected by using an analysis of the next character or all characters. .	72
5.10	Words that cannot be corrected by the proposed substitutions method. . .	72
5.11	Results of the insertion method. . . . .	74
5.12	Results of the deletion method. . . . .	74
5.13	Example with the insertion method. . . . .	75
5.14	Example with the deletion method. . . . .	75
5.15	Results of the compounder method. . . . .	76
5.16	Example of coumponder method. . . . .	76
5.17	Results of the spelling_correction method considering the analysis of the next character. . . . .	77
5.18	Results of the spelling_correction method considering the analysis of all the characters. . . . .	78
5.19	Results of the spelling_correction method considering the analysis of the next character with Tesseract restriction. . . . .	78
5.20	Results of the spelling_correction method considering the analysis of all the characters with Tesseract restriction. . . . .	79

5.21	Results of the spelling_correction method considering the analysis of the next character with certainty restriction. . . . .	79
5.22	Results of the substitution method considering the analysis of all the characters with certainty restriction. . . . .	80
5.23	Results of the frequency in Tesseract. . . . .	83
5.24	Example of frequency method. . . . .	84
5.25	Results of the frequency method with Tesseract restriction. . . . .	84
5.26	Results of the frequency method with certainty restriction. . . . .	85
5.27	Results of the combination of spelling_correction and frequency methods considering the analysis of the next character. . . . .	88
5.28	Results of the combination of spelling_correction and frequency methods considering the analysis of all the characters. . . . .	88
5.29	Results of the combination of spelling_correction and frequency methods considering the analysis of the next character with Tesseract restriction. .	89
5.30	Results of the combination of spelling_correction and frequency methods considering the analysis of all the characters with Tesseract restriction. .	90
5.31	Results of the combination of spelling_correction and frequency methods considering the analysis of the next character with certainty restriction. .	90
5.32	Results of the combination of spelling_correction and frequency methods considering the analysis of all the characters with certainty restriction. .	91
5.33	Results with Testing Data considering the analysis of the next character.	94
5.34	Results with Testing Data considering the analysis of all the characters. .	94

# List of Abbreviations

API	Application Programming Interface
CC	Connected Components
COCA	Corpus of Contemporary American English
DAWG	Directed Acyclic Word Graph
DP	Distributed Proofreaders
EAST	Efficient accurate Scene Text Detector
ED	Edit Distance
EP	Edit Probability
ER	Extremal Regions
FSM	Weighted Finite-State
HMM	Hidden Markov Model
ICDAR	International Conference on Document Analysis and Recognition
IND	Incidental Scene Text
kNN	k-Nearest Neighbor
LSH	Locality Sensitive Hashing
LSMT	Long short-term memory
MSER	Maximally Stable Extremal Region
OCR	Optical Character Recognition
ROVER	Error voting technique
SMT	Statistical Machine Translation
STD	Scene Text Detection
STDR	Scene Text Detection and Recognition
STR	Scene Text Recognition
SVM	Support Vector Machine
SVT	Street View Text
SWT	Stroke Width Transform

VLMM	Variable Memory Length Markov Model
YOLO	You Only Look Once
WRA	Word recognition accuracy

# Contents

<b>List of Figures</b>	<b>8</b>
<b>List of Tables</b>	<b>11</b>
<b>1 Introduction</b>	<b>17</b>
1.1 Problem and Motivation . . . . .	17
1.2 Objectives . . . . .	19
1.3 Contributions . . . . .	20
1.4 Organization . . . . .	20
<b>2 Background</b>	<b>21</b>
2.1 Scene Text Detection and Recognition (STDR) . . . . .	21
2.1.1 Scene Text Detection . . . . .	22
2.1.2 Scene Text Recognition . . . . .	23
2.2 Tesseract . . . . .	23
2.2.1 A Brief History of Tesseract . . . . .	24
2.2.2 Tesseract Overview . . . . .	24
<b>3 Related Work</b>	<b>33</b>
<b>4 Syntax-Based Techniques</b>	<b>40</b>
4.1 Spelling correction . . . . .	40
4.1.1 Substitution . . . . .	44
4.1.2 Insertion . . . . .	48
4.1.3 Deletion . . . . .	49
4.1.4 Compounder . . . . .	50
4.2 Frequency . . . . .	51
4.3 Finding the error . . . . .	56
4.3.1 Tesseract decision . . . . .	57
4.3.2 Tree best variable . . . . .	58
<b>5 Experimental Evaluation</b>	<b>61</b>
5.1 Dataset . . . . .	62
5.2 Evaluation Metrics . . . . .	65
5.2.1 Scene Text Recognition . . . . .	65
5.3 Spelling correction . . . . .	66
5.3.1 Substitution . . . . .	66

5.3.2	Insertion and Deletion . . . . .	73
5.3.3	Compounder . . . . .	76
5.3.4	Best methods to correct errors . . . . .	77
5.4	Frequency . . . . .	83
5.5	Spelling correction and Frequency . . . . .	87
<b>6</b>	<b>Conclusions and Future Works</b>	<b>95</b>
	<b>Bibliography</b>	<b>96</b>

# Chapter 1

## Introduction

Each year a large amount of data is generated thanks to the various electronic devices, which encourages the development of methods for its analysis and subsequent use in other applications. Millions of images, videos, text, publications are produced per minute, feeding with data various areas such as Visual Computing, Machine Learning, Natural Language Processing. These applications solve problems such as the recognition or scanning of images, analysis of large volumes of data, facial recognition, recognition of abnormal events, among others.

One of the most complex challenges that have begun to take relevance in recent times is Text Detection and Recognition in natural scenes from images and videos. Scene Text Detection and Recognition aims to find all the regions in an image that a human being considers to have a text, framing that region into a bounding box and associating it with its corresponding characters. The result (a sequence of characters) can be easily processed by a computer. Since it has to face many challenges [57], methods that solve this problem divide it into two major tasks: Scene Text Detection (STD) and Scene Text Recognition (STR).

STD locates the text in an image. The result of this step are boxes that surround the region where there is a potential text. STR returns the sequence of characters of the text contained inside the region. In this work, we seek to bring contributions to improve the accuracy of STR.

### 1.1 Problem and Motivation

Methods to recognize text arose from the need to manipulate the scanned documents without the user having to transcribe them. The first methods known as OCR (Optical Character Recognize) were very precarious since they could only recognize the text in documents well scanned without problems of noise, lighting, or blur. The process consisted in recognizing the characters of the scanned document and returning the text as a result.

With the advancement of computing, other techniques started to be used, in addi-

tion to recognition, such as the formation of the word. However, the results were not as expected. To improve that, several techniques began to be used both in the input and output data to improve the results. Techniques used in the input data focused on improving the quality of the scanned document, while the techniques in the output data corrected the words that OCR recognized incorrectly.

It has reached a point where the results achieved by the OCR methods in scanned documents have become quite acceptable [19, 33, 39], moving the research efforts towards addressing other problems of text recognition, such as text in images from natural scenes.

Text Recognition solutions for natural scenarios (or Scene Text Recognition - STR), similarly as what happened early on for OCR, started producing very poor results. The first methods consisted of three basic steps to segment the region of text, recognize each character, and then form the word. Over time, several methods began to solve the STR problem. In order to know which was the most efficient method, several competitions arose, such as the ICDAR [32, 44, 26, 25, 8, 41].

The methods for STR began to test techniques that in other areas have given good results, such as neural networks. The use of neural networks improved the results, but not as expected, since they began to add new challenges to the problem such as orientation, multilingual, unfocused text, among others [10, 57, 59]. Currently, the best results for STR without challenges are recurrent neural networks, resulting in a 94.2% accuracy for the ICDAR 2013 dataset [7].

Currently, the state-of-the-art solutions that present the best results are sequence-based methods. For example, experiments carried out with Focusing Attention Network, proposed by Z. Cheng *et al.* [7] as a contribution over the state-of-the-art attention-mechanism for character recognition, and running on a work-station with one Intel Xeon(R) E5-2650 2.30GHz CPU, an NVIDIA Tesla M40 GPU, and 128GB RAM for the ICDAR2013 data set, resulted in an accuracy of up to 94.2%. Unfortunately, the excessive usage of hardware resources and its corresponding high computational costs have considerably impacted the execution of such tasks in highly constrained embedded systems (e.g. cellphones or smart TVs), demanding the design of faster methods which cannot achieve the same accuracy as in more powerful platforms. For example, experiments carried out with Tesseract v3.04 running on Artik 530 <sup>1</sup> hardware for the ICDAR 2013 data set resulted in an accuracy of only 43.48%.

Therefore, solutions that enable fast STD and STR in embedded systems need to use methods that do not consume much memory or energy such as those in the Libccv or OpenCV libraries for Text Detection, or traditional methods such as Tesseract for Text Recognition. Unfortunately the accuracy of the results produced by such methods is very low when compared to those obtained with neural networks.

Although researchers continue trying to solve the various challenges that STR presents, another text recognition problem has begun to emerge. The OCR methods that are considered good for scanned documents show deficiencies in scanned documents of historical

---

<sup>1</sup>The Artik 530 features are: High performance, 4-core, 32-bit ARM Cortex A-9 processor @ 1.2GHz; ARM Mali GPU for multimedia, 512MB RAM, and 4GB flash (eMMC).



Figure 1.1: Examples of various types of noise in document images: (a) Regular shaped nontextual noise. (b) Textual and irregular shaped nontextual noise. (Extracted from [5])

books, where the sheets are very worn, small pieces are missing, or do not exist (Figure 1.1).

G. Chiron *et al.* conducted an investigation where they show that the OCR methods available in the market are deficient for historical documents [9]. To solve the problem, attention begun to focus on methods that correct the results from OCR, known as post-correction. These methods are divided into two tasks: to find the wrong word and correct it. Although the methods proved to be efficient in the OCR area, there are not many post-correction methods developed for STR.

## 1.2 Objectives

This work aims to investigate and evaluate the existing post-correction methods as well as to propose new methods or heuristics to improve the accuracy of Tesseract when working on STR problems. The experiments were performed on several databases (Chapter 5). A comparative analysis of the results produced by the original Tesseract and Tesseract modified by the proposed method is presented and discussed.

In addition to the general objective, this work also aims at the following specific goals:

- Carry out a bibliographic investigation of the main post-correction methods in OCR and Scene Text Recognition.
- Conduct a study of the Tesseract Text Recognition tool, specifically of its language model.
- Explore the various post-correction methods and try out possible combinations.
- Implement the selected post-correction methods in Tesseract to improve its accuracy.

- Demonstrate that using these selected post-correction methods improve the accuracy in scene image text recognition.
- Conduct experiments on different databases for Text Recognition and for Text Detection and Recognition.

## 1.3 Contributions

In order to address the above described problem this work makes the following contributions:

- An analysis of the problems presented by the results of Tesseract v.3.4 in images of natural scenes. Tesseract is one of the most used tools to recognize text in scanned documents, but in Scenes images results are not good. The analysis of the results produced by Tesseract v.3.4 in scene images reveal many spelling errors, errors that are brought from the Text Detection stage, and errors in proper names. This work analyzes the most common errors and shows how to address them.
- Improvement of the accuracy of the Tesseract in scene images using post-correction methods. We have implemented several error correction methods in Tesseract to improve the results in scene images. The results of such methods show improvement with respect to the original Tesseract for all the tested databases.
- We show that post-correction methods can be used to improve the results in scenes images. Normally post-correction methods are used in scanned documents since context can be used to avoid overcorrection. However, we demonstrate that using other restrictions provided by the same text recognition methods overcorrection can be avoided, greatly improving the obtained results.

## 1.4 Organization

The rest of the work is organized as follows. Chapter 2 describes relevant concepts of Text Detection and Recognition, and Tesseract. Chapter 3 presents related work on Text Recognition. Chapter 4 describes the proposed approaches. Chapter 5 describes the testing databases and the experimental setup and analyses the results achieved by the proposed methods. Chapter 6 concludes with final remarks and directions for future work.

# Chapter 2

## Background

### 2.1 Scene Text Detection and Recognition (STDR)

Scene Text Detection and Recognition is a very complex image analysis problems in recent times. The task consisting of extracting text from scene images (Figure 2.1). The resulting text can be easily processed by a computer, thus allowing various applications such as urban navigation, help for people with visual impairment, automatic translation, indexing and search of image databases by text content, among others [34].



Figure 2.1: *Scene Text Detection and Recognition* problem. All the possible texts are found, the outputs being the sequence that characters.

STDR is known as a complex problem since it has to face many challenges [10, 57, 59]. The most common challenges are:

- Text occupies only a part of the image.
- Text is in a non-uniform background scene. In images and videos of natural scenes, objects such as signs, fences, bricks, buildings, symbols, among others, may look similar to a text, making it difficult to distinguish the text from such objects.
- To detect and recognize texts in images and videos of natural scenes one must consider light, noise, occlusion, blur, distortion, low resolution, and reflection.

- Texts may contain more than one type of letter fonts, colors, scales or distance between words and characters. This makes the detection and recognition of characters difficult when the number of classes of characters is large [34].
- Contemplate different orientations of the text.
- Multilingual environments. Although the languages that come from Latin are very similar, other languages differ completely [28], like Japanese, Chinese or Korean which have many different types of characters. For example, in Arabic writing, characters are connected, changing their shape according to the context. In Hindu millions of shapes can be built from its alphabet, each representing a distinct syllable.

### 2.1.1 Scene Text Detection

Scene Text Detection (STD) identifies texts regions (regions of the image where there is a text) and marks its location in the image. The result of this step are polygonal bounding boxes that surround text regions. Depending on the solution, the bounding box can contain a line of a text or a word. STD has two primary goals: (a) to locate all possible candidates for texts in the image; and (b) to verify which of these candidates are a text or not.

The first techniques that appear in the literature detect all possible candidates for the text through two widely used methods: connected components (CC) and sliding windows [57]. CC-based methods extract regions of the image based on the characteristics (color, texture, strokes, borders) of the pixels. The most commonly used methods of this type are those based on MSER (Maximally Stable Extremal Region) [34] and ER (Extremal regions) [11]. In the sliding window methods, a window of different sizes passes along the image to find possible candidates for the text. The sliding window leads to a computational cost and inefficiency since the method looks for different rectangles in different sizes, radius, and rotation perspectives. To verify whether the detected candidates are texts or not, candidates must meet certain restrictions, such as: if a group of candidates has the same color, size, and width of the stroke, that group of candidates is considered as a word. Another way to verify that is to use classifiers such as Super Vector Machine [47] or Ada Boost [29] to determine if the candidate is a letter or not.

The most popular libraries for Text Detection that use CC and sliding-windows are OpenCV and Libccv. OpenCV is an open-source machine learning software library that includes over 2500 algorithms. OpenCV implements some algorithms to detect and recognize texts in images<sup>1</sup>. These algorithms are based on the work of L. Neumann *et al.* [35, 36] for images having only horizontal words. To deal with images with vertical, horizontal or inclined words, OpenCV implements the work of L. Gómez *et al.* [20]. Libccv uses the operator Stroke Width Transform (SWT) [14] for Text Detection. SWT uses the stroke widths of the pixels to form CC. SWT depends largely on the detection of edges.

---

<sup>1</sup>[https://github.com/opencv/opencv\\_contrib](https://github.com/opencv/opencv_contrib), last accessed on 12/12/2019

With the appearance of neural networks, some techniques started using YOLO (You Only Look Once) [40] approaches to detect text regions in a scene image. YOLO is a neural network that uses convolution layers to find objects. YOLO works similarly to sliding windows, but unlike sliding windows, it only goes through the image once. It finds all the possible candidate text, and then classifies the region as having a text or not.

The methods that use neural networks to detect texts became so popular that some famous libraries started using them as well. OpenCV implements EAST (Efficient accurate Scene Text Detector) [58] in its newer versions. EAST is a robust deep learning method that can find horizontal and rotated text.

### 2.1.2 Scene Text Recognition

Scene Text Recognition (STR) converts image regions to strings. Several STR methods have been proposed that they can be grouped into three types: traditional methods, deep neural-network-based methods, and sequence-based methods [1].

In traditional methods, handcrafted visual features are extracted to recognize individual characters one by one. Then the words are formulated using statistical models in terms of low-level features and high-level language [34]. Typically, a post-processing pass is included to correct syntactic errors, remove ambiguous recognition or segmentation errors so as to improve recognition accuracy.

With the emergence of deep neural networks, more robust characteristics began to be extracted, but post-processing was still necessary to give better results. As an evolution of this research line, the problem of STR has started to be viewed as a sequence of two machine learning problems: (1) first a deep neural network that encodes a text image into a sequence of characteristics; and (2) a sequence recognition solver based on a recurrent neural network that recovers the character sequence.

There is a number of library APIs that recognize texts in the scene image. The most famous is Google's APIs know as Tesseract. Although Tesseract was initially developed to recognize texts in scanned documents, it can currently be used to recognize scene images. Alchemy Visionn's APIs<sup>2</sup> is an IBM-owned company that uses machine learning and computer vision to recognize texts. Microsoft's Vision APIs<sup>3</sup> is used to extract printed and handwritten texts in the Microsoft Azure Cloud.

## 2.2 Tesseract

In this section, we perform a Tesseract analysis of version 3.04. Although the best and stable version is currently version 4, we use version 3.04, because it uses traditional methods to recognize text.

---

<sup>2</sup><https://www.ibm.com/blogs/cloud-archive/2016/05/alchemy-and-watson-visual-recognition-api/>, last accessed on 01/11/2019

<sup>3</sup><https://azure.microsoft.com/es-es/services/cognitive-services/computer-vision/>, last accessed on 03/11/2019

### 2.2.1 A Brief History of Tesseract

Tesseract is an open-source OCR engine, initially developed by HP between 1985 and 1995 [51]. Tesseract was developed to improve the quality of the HP scanners given the bad performance of the commercial OCR engines of those days while scanning poor quality documents.

The first time Tesseract appeared in a competition was at the annual UNLV OCR precision test of 1995. Although it showed superior results, it was set aside for 10 years [48]. In 2005 HP released Tesseract as open source tool and in 2006 Google started improving it.

From its first version (1.x) to the last (4.x), Tesseract has evolved to make it more robust and efficient. Among the improvements of each version are:

- Version 1.x focused on the problems of grayscale images.
- Version 2.x, languages born from Latin were added.
- Versions 3.0.x and 3.01.x, the analysis of page design was improved, that is, Tesseract begins to consider vertical texts and with slopes greater than 15 degrees; resources were added to recognize languages such as Arabic and Hindi; segmentation was improved to be able to recognize languages with a large number of characters such as Chinese and Japanese.
- Version 3.02.x and 3.03.x, the ability to recognize right-to-left written languages as Hebrew was added; the language model is added to improve accuracy.
- For versions 3.4.x and 3.5.x, Tesseract could recognize 100 languages simultaneously, though one still must indicate the language to be recognized.
- In version 4.x, Tesseract starts using a long short-term memory (LSMT) neural network, recognizing 123 languages with good results for scanned documents in the majority of languages.
- Version 5.x is in an alpha version, and it aims at eliminating all errors observed in version 4.x.

### 2.2.2 Tesseract Overview

In the beginning, Tesseract was designed for scanned documents. Therefore Tesseract OCR engine architecture is intended for scanned documents. Over time it has been modified to remain one of the most used text recognition frameworks. Currently, Tesseract 4.x uses a recurrent neural network LSMT to recognize the characters and form the final results, but most of the architecture is based on the Tesseract 3.4.x. Figure 2.2 shows the Tesseract OCR 3.4.x engine architecture that is currently used. In all that remains of the section, we describe the showed components in Figure 2.2.

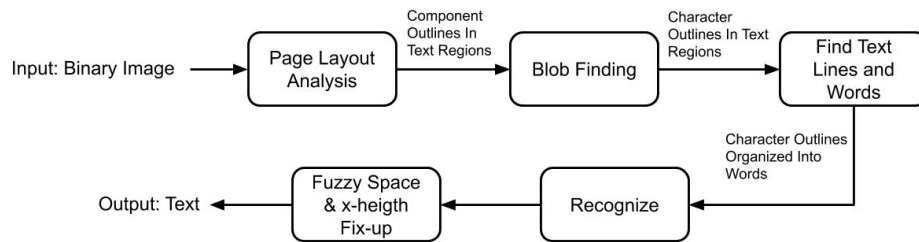


Figure 2.2: Tesseract OCR 3.4.x engine architecture [50].

Since it began to be developed at HP Labs, Tesseract does not perform STD. Therefore, Tesseract assumes that the input is a binary image of the text regions of the image to be analyzed (Figure 2.3).

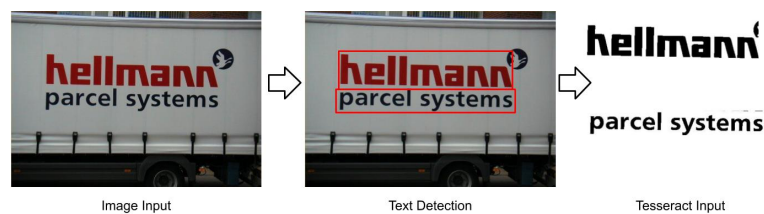


Figure 2.3: The Tesseract input is a binary image focused on the text.

## Layout Preprocessing

The first step of Tesseract 3.x is layout analysis. The layout analysis was designed so that in all the following steps the text is analyzed horizontally. As Tesseract is designed for scanned documents, it is considered that parts of the document can be horizontal or vertical [43, 54]. First, the automatic orientation detection finds all the orientations of the text regions. Then all vertical blocks are converted to horizontal, considering a rotation of 90 degrees, and the blocks of text regions are passed to the next step. For images in natural scenes, it works in the same way but the background complicates the detection of orientation.

When text regions are normalized, Tesseract recognizes connected components (CC). For recognizing all texts, CC is recognized in a white background and a black background. Then CC is joint to form blobs. Blobs are considered as a putative classifiable unit, which can be one or more CCs that overlap horizontally and their internal contours or nested holes. Later the text line finder detects lines of text by the vertical overlap of adjacent characters. With text lines, blobs in a line are organized into recognition units, to move on to the next step.

## Word Recognition

The next step is to recognize the characters and form the words. Figure 2.4 shows how Tesseract 3.x performs this step. Recognition begins with image segmentation (Character

Chopper stage). If the chopper cut too much the characters, Tesseract considers putting the components together (Character Associator stage). Then a static and an adaptive classifier classify them. Once classified, Tesseract forms the word and chooses the best result. If the result is not the desired one, Tesseract does the process again, changing the parameters.

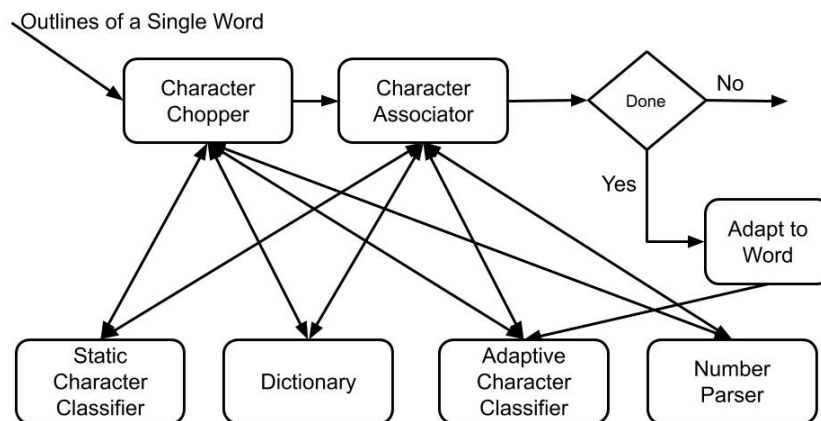


Figure 2.4: Block diagram of Tesseract word recognition [50].

Blobs do not always represent a character (Figure 2.5). A blob can contain more than one character, in which case they need to be separated, or it can not represent a character, in which case blobs are joined to form a character. Tesseract uses the best-first search strategy on the segmentation graph to find characters but grows exponentially according to the length of the blob sequence. A character width restriction is incorporated to avoid exponential growth, thus reducing the segmentation points to be evaluated.

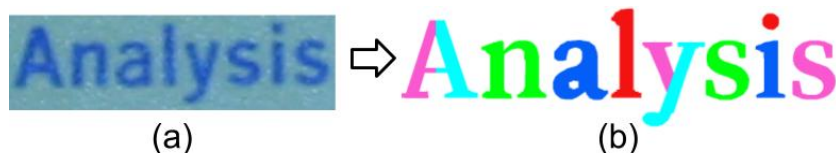


Figure 2.5: (a) The Tesseract input image. (b) How blobs represent the word in the image (a).

Figure 2.6 shows the segmentation graph of the image in Figure 2.5a, after classifying each blob or group of blobs representing a character. In Figure 2.5b, the nodes represent the spaces between the blobs, and the edges are all possible characters that the blob or a group of blobs represent. That is, the edge that goes from the first node to the second node represents all the characters that the classifier recognizes with blob number 1. The edge that goes from the first node to the third node represents all the characters that the classifier recognizes by joining blobs 1 and 2.

To recognize the characters, the first step is to extract features. Tesseract uses shrinking features. Shrinking features are segments of a polygonal contour approximation (Figure 2.7). There are two types of shrinking features: nanofeatures and picofeatures. They

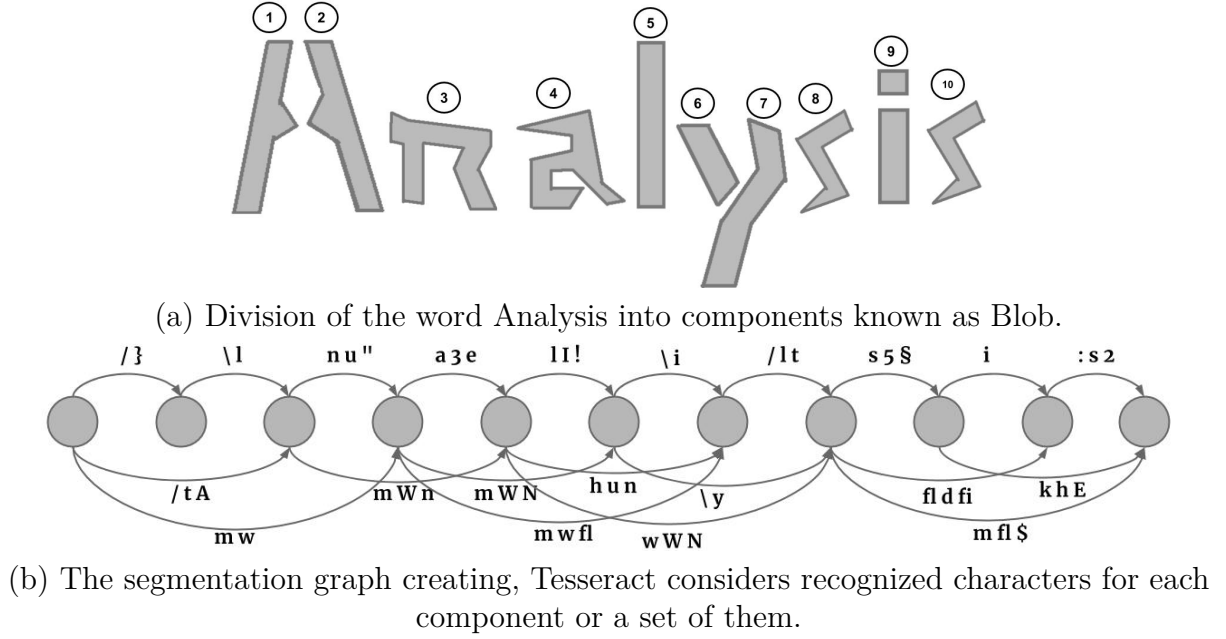


Figure 2.6: Segmentation graph.

can be nanofeatures if 4 dimensions are considered: x, y, direction, and length (Figure 2.7a). If the previous segmentations are of a fixed length, it is known as picofeatures (Figure 2.7b).

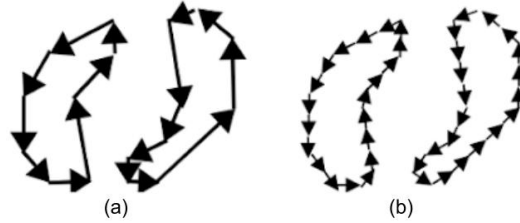


Figure 2.7: Shrinking features: (a) Nanofeatures (b) Picofeatures [51].

Tesseract version 3.x uses an optimized k-Nearest Neighbor (kNN) classifier to recognize characters. KNN is a supervised method of Machine Learning used to classify or predict values [27]. KNN searches the observations closest to the item that is trying to predict. Then KNN classifies the item depending on the majority of data around it. KNN classifies a new item as follows:

1. Calculate the distance between the new item and the rest of the data in the training dataset.
2. Select the k-nearest elements.
3. Make a majority vote between the k points, that is, the class that dominates between the k points will be the resulting class.

In Tesseract, to perform step 1, the Euclidean distance is used as shown in Equation 2.1.

$$\operatorname{argmin}(k) \frac{1}{M + J_k} \left( \sum_{l,i} (x_{il} - \mu_{ijk})^2 + \sum_{j,i} (x_{il} - \mu_{ijk})^2 \right) \quad (2.1)$$

Where:  $k$ : the number of neighboring points to perform the classification.  $\vec{X}_l = x_{i,l} : i \in [1, n], l \in [1, M]$ : the characteristics of the training data.  $\mu$ : characteristics of the item to be classified. The indexes:  $i$  = entity dimension,  $j$  = cluster,  $k$  = character class, and  $l$  = unknown entity index.

Equation 2.1 considers two Euclidean distances. The first summation calculates the characteristics that are allowed by the cluster while the second summation calculates the characteristics that are required. The second summation is added to prevent characters such as ‘c’ from being classified in class ‘e’. Since all the characteristics of ‘c’ are allowed in ‘e’, but ‘c’ does not have all the characteristics required to be classified as ‘e’ [51].

The complexity of calculating the distance between the new item and all training data is  $\mathcal{O}(J_k K M n)$ . For example for the English language, where each character has an average of 100 features, each feature has dimension 3, there are 3520 classes (110 characters-set \* 32 trained fonts), and each class has 50 training data, calculate distance is  $\mathcal{O}(10^{18})$ . The complexity increases more in languages where there are more classes such as in Chinese, Japanese or Thai.

An additional stage is considered before using KNN to reduce complexity. This stage is called class pruner, which reduces the character set to a small list of 1-10 characters. The class pruner is relatively fast since it uses a method related to Locality Sensitive Hashing (LSH) [12].

At the end of the recognition, each character has the following values: rating (the classifier distance weighted by the length of the outline in the blob), certainty (indicates the classifier certainty of the choice), script (in Tesseract, it is a group of languages that come from the same origin or share similarities as characters), a gap after or before character, and height of the character.

## Language Model

The Viterbi algorithm [17] is used in the segmentation graph to form the words. The objective of the Viterbi algorithm is to find the lowest cost path, that is, the path that best combines the recognized characters (Figure 2.8). The cost in the segmentation graph is determined by the language model components and the properties of the cut between the blobs of the path.

If results are not good enough, a list of pain points is constructed. Pain points are the points along the way of the character that do not seem to be consistent with the neighboring. With the new Pain points, Tesseract does the process again until the results are good enough.

The result of the word-formation process is a list of all possible words that the

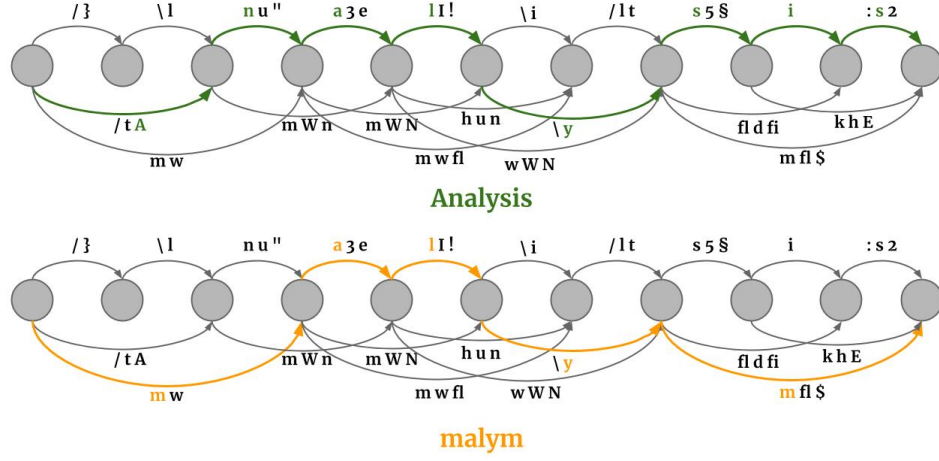


Figure 2.8: The Viterbi algorithm executed in the segmentation graph of Figure 2.5. Created words are “Analysis” and “malyn”.

Viterbi algorithm considers as reasonable good. These words in Tesseract are known as WERD\_CHOICE. Sometimes some WERD\_CHOICES are not good. The word rating that calculates the word defect is considered to filter these words.

Word Rating is calculated by considering the uncertainty of each character that forms the word and the inconsistencies that the word presents. Equation 2.2 shows how Tesseract calculates the rating for the word  $w$  of length  $l$ .

$$rating(w) = factor(w) \times \sum_{i=0}^{l-1} uncertainty(c_i) \quad (2.2)$$

Where:  $c_i \in w$ , the *uncertainty* method calculates the uncertainty of each character of the word, and the *factor* method calculates the general defect of the  $w$  word. Equation 2.3 shows the *factor* method.

$$factor(w) = 1 + frequency(w) + dictionary(w) + length(w) + shape(w) \quad (2.3)$$

The *frequency* method calculates if  $w$  is among the most frequent words of Tesseract. The *dictionary* method calculates if  $w$  is among the Tesseract dictionaries. The *length* method determines whether  $l$  does not exceed the common size of words of the language.

When Tesseract calculates the word rating for all WERD\_CHOICES, the best word is recognized, which is the one with the lowest word rating (Figure 2.9b). Then all the WERD\_CHOICES that are not in a reasonable range of the best option are removed (Figure 2.9c). Since all WERD\_CHOICES represent the same text of the image, they form a Tesseract variable known as WERD\_RES.

Finally, Tesseract calculates some attributes of each WERD\_RES, such as, *tess\_data*. *tess\_data* determines whether WERD\_RES is regarded as “reasonable good”. If the variable is true, Tesseract terminates, and the best WERD\_CHOICE is the final result. Otherwise, the same recognition procedure is performed again but with modified

(a)	(b)	(c)	(d)
Analysi2 -> 128.20	<b>Analysis -&gt; 86.25</b>	<b>Analysis -&gt; 86.25</b>	<b>Analysis -&gt; 86.25</b>
Analysi: -> 101.31	malvsi: -> 89.58	<del>malvsi: -&gt; 89.58</del>	tnalvsi: -> 96.68
malvsi: -> 89.58	tnalvsi: -> 96.68	tnalvsi: -> 96.68	Analysi: -> 101.31
AnalysiS -> 117.24	Analysi: -> 101.31	Analysi: -> 101.31	
Analysis -> 86.25	AnalysiS -> 117.24	<del>AnalysiS -&gt; 117.24</del>	
tnalv5i2 -> 126.32	tnalv5i2 -> 126.32	<del>tnalv5i2 -&gt; 126.32</del>	
tnalvsi: -> 96.68	Analysi2 -> 128.20	<del>Analysi2 -&gt; 128.20</del>	

Figure 2.9: (a) WERD\_CHOICES that were formed with the Viterbi algorithm. (b) The best chosen WERD\_CHOICE considering the word with the lowest rating word. (c) Remove all WERD\_CHOICE that are not in a reasonable range. (d) The final results.

parameters.

## Dictionary

Dictionaries are important in the process of Text Recognition in an image. Among the dictionaries that Tesseract uses are:

- word-dawg: made from dictionary words from the language.
- freq-dawg: made from the most frequency words.
- punc-dawg: made from punctuation patterns found around words.
- number-dawg: made from tokens which originally contained digits.
- bigram-dawg: A dawg of word bigrams where the words are separated by a space and each digit is replaced by a “?”.
- user-words: A list of additional words to add to the dictionary.

The dictionary is represented through a DAWG (Directed Acyclic Word Graph) structure. DAWG is compact, so it takes up little memory, and words are easy to find [50] (Figure 2.10). In the DAWG, each edge is labeled with a character. The characters along a path from the root to a node are the substring which the node represents. For example, node 6 in the Figure 2.10 represents “wor”. The initial node, node 0, represents an empty string, and the final node represents a word that exists in the dictionary.

Tesseract builds dictionaries from a list of words in alphabetical order. First, it forms a Trie structure with the words [18]. Trie is a tree-like structure that allows information retrieval (Figure 2.11). Unlike DAWG, Trie takes up more memory space since it has more nodes to store. After Tesseract constructs the Trie, some repeating nodes are reduced, thus forming the final DAWG. For example, the paths from node 7 to nodes 10, 13, and 14 is equal to the paths from node 8 to nodes 18, 19, and 20. Therefore it is reduced as shown in the paths from node 7 to node 12 of the DAWG in Figure 2.10.

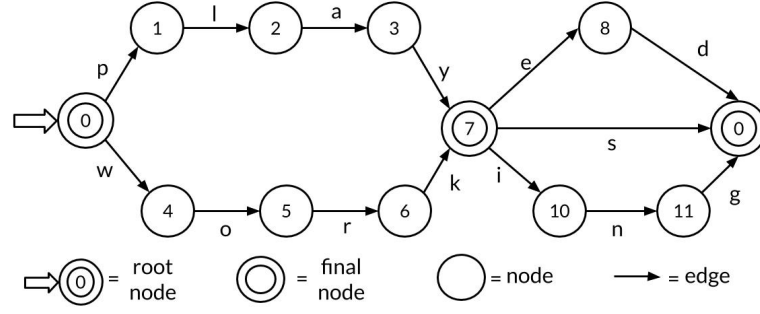


Figure 2.10: DAWG for all forms of the verbs *play* and *work*.

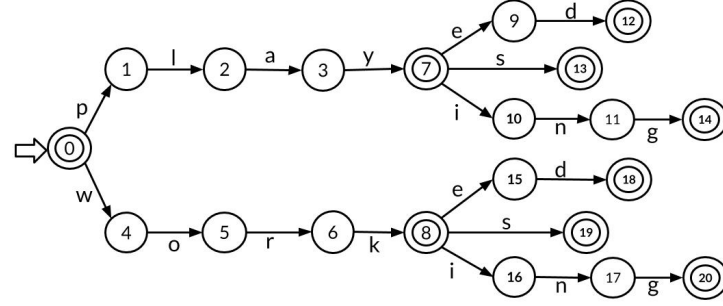


Figure 2.11: Trie for all forms of the verbs *play* and *work*. Trie that represents the Dawg in Figure 2.10.

Tesseract represents DAWG as an array of edges. Each edge is stored in a 64-bit variable. The first  $n$  bits on the right are reserved for the character identifier,  $n$  depends on how many characters exist in the language of the dictionary. The next 3 bits represent edge specifications. The first represents the marker flag of this edge, the second bit represents the direction flag of this edge, and the last bit represents if this edge marks the end of a word. The last bits represent the next node visited by following this edge. Figure 2.12a shows how the 64 bits are distributed for a dictionary in English.

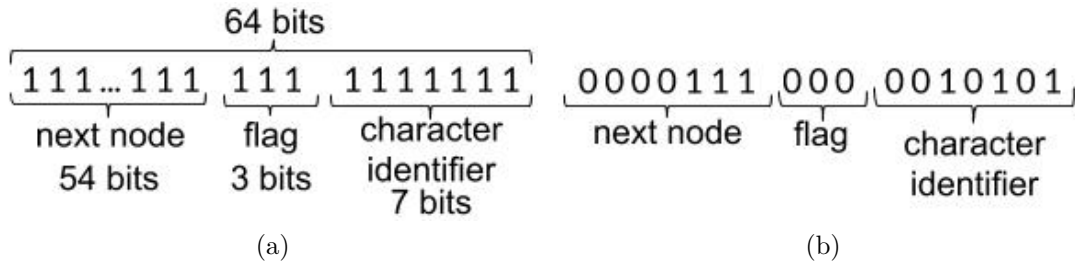


Figure 2.12: (a) Distribution of the 64 bits that represent an edge in DAWG in an English dictionary. (b) Representation of the edge to node 3 of the Figure 2.10.

Figure 2.12b represents the edge that goes to node 3. Given that the English language has  $111 \approx 2^7$  characters with lowercase, uppercase, digits, and punctuation marks, so character identifier = 7. 'a' is represented by  $21 = 0101$ . The next three bits represent specifications of the formed substring: the first bit is 0 since the next node is not marked;

the second bit is 0 since the word is forming from left to right and the third bit is 0 because the formed substring in node 3 is not a final word in the dictionary. The next node represents the node that the edge is going to, in this case,  $3 = 0111$ .

# Chapter 3

## Related Work

In this chapter, we briefly describe some relevant works related to post-correction methods in Optimal Character Recognition (OCR) and Scene Text Recognition (STR).

According to Y. Bassil *et al.* [2], error correction methods are generally categorized into three main classes: (a) manual error correction; (b) dictionary-based error correction; and (c) context-based error correction.

### Manual error correction

The idea of manual correction, as the same name says, is that people correct the results manually. For example, in 2000, a project known as Distributed Proofreaders (DP) was launched on the web <sup>1</sup>, for the review of e-books in the Project Gutenberg<sup>2</sup>. The idea of the project DP was that volunteers around the world would compare scanned documents with their corresponding OCR results. As the first volunteers were not expected to correct all the errors, the corrected text would be corrected again by another volunteer. When volunteers observed that there were no more errors to correct, corrected texts were stored in the digital archives of the Project Gutenberg.

Errors can be easily corrected, but they are prone to human errors. For example, people may not notice some errors, especially if there are a large number of reviews to be done, as in scanned documents. Therefore, methods that automatically correct spelling errors began to be developed.

### Dictionary-based error correction

Dictionary-based methods, known as lexical error correction, are those in which errors can be corrected using a dictionary or lexicon.

Dictionary-based correction methods date back to the beginning of OCR methods in scanned documents. V. Levenshtein proposes a theoretical model to solve the insertion and deletion, considering them as binary problems [30]. The idea of from 0 to 1 is the

---


<sup>1</sup><https://www.pgdp.net/c/>, last accessed on 11/12/2019

<sup>2</sup>Project Gutenberg is a library of over 60,000 free eBooks. <https://www.gutenberg.org/>, last accessed on 15/01/2020.

insertion, and from 1 to 0 the deletion where 0 is a null string and 1 some character of the language. If there is more than one word that can replace the original, it is considered the word that has the lowest edit distance from the original.

The method of V. Levenshtein is not the only way to correct the words; other algorithms use finite-state automata theory. I. Guyon *et al.* propose a linguistic post-processor to correct characters that OCR recognizes incorrectly [21]. The system uses a Variable Memory Length Markov Model (VLMM) trained with a dictionary that generates the candidates with the help of the language syntactic properties. The approach passes characters to an automaton that predicts which are the possible next characters. If the next character of the word does not match, a new candidate is created. The authors also extended their VLMM to correct proper names by adding them into the dictionary. Unlike ours, this method only allows substitutions; deletion and insertion are not possible.

R. Beaufort *et al.* propose a method to make word corrections depending on character recognition [3]. The method performs the correction using a Finite-State Machine (FSM) and three lexicons: a normal dictionary, a confusion list, and an alphabetical mapping. FSM is created online, depending on the word, and weights are calculated from the lexicons. The method considers the best results as the route of the characters with the greatest weight. The method assumes that characters with which the word must be corrected must always be in the recognized character set, unlike our proposed method in which the characters with which it is corrected are not necessarily in the recognized character set. Figure 3.1 shows an example of how the explained method does not find the correct word. The correct word is “*Management*”, but the method using FSM does not correct since the ‘*t*’ character is not found among the characters recognized for the last character of the word.



Recognized Word	Correction using a FSM	Correction using proposed method
Managemen!	Characters recognized for the last character l = l,  , !	Using the dictionary
	Managemen!	Management

Figure 3.1: Correction using the method proposed in [3].

K. Taghva *et al.* propose an error correction system for documents recognized by OCR [53]. The correction is based on chain approximations and n-gram analysis in Bayesian systems based on statistics. The dictionary is stored in hash tables, at first, the system has a medium amount of words but more and more information is collected as the user interactively reviews the errors corrected by the method.

Methods that use a lexicon with the most common words of the language showed incorrect results in proper nouns or words that were not in the dictionary. J. Feild *et*

*al.* propose to eliminate these errors by incorporating language information using a large web-based lexicon (Web IT) [16]. When the method obtains the first initial label and the probabilities of bigram between characters, new words begin to be generated considering the words with distance edition of two and the frequency of the word obtaining with the most common websites. Then the method chooses the word with the highest probability (the sum of all the probabilities of characters by the frequency of the word). Unlike our proposed method, all words pass the post-processing, if the correct word is the initial label, a double effort is done to achieve the same result. Our proposed method tries to avoid the double-work.

F. Bai *et al.* propose Edit Probability (EP) to eliminate misalignment between ground truth and output caused by unrecognizable, missing, or superfluous characters to effectively train models based on attention [1]. EP estimates the probability of generating a string from the output sequence of the probability distribution conditioned on the input image. EP also predicts the possible word with the probability found analyzing two cases without lexicon or with the information of a lexicon (edit probability Trie). EP was designed only for methods that use attention-based models such as LMST since the probability is obtained using information from internal states.

N. Sharma *et al.* propose a text recognition method for multi-lingual videos (English, Hindi, and Bengali) [45]. The method uses Spatial Pyramid Matching to identify the script; SIFT to extract the characters; Support Vector Machine (SVM) to classify the characters; Hidden Markov Model (HMM) to form words from the characters and finally post-processing. The post-processing calculates the minimum edit distance between the words obtained and the entries in the lexicon, forming a list of the three main suggestions that are the final results. Unlike our proposed method, the lexicon is formed by the 2000 most common words of languages, that is, words that are not common to give errors. Also, post-processing is done for all words, even for those words which are not in the dictionary, and end up been recognized, thus resulting in false-positive recognition.

D. Chen *et al.* propose a method with post-processing to deal with low-resolution video frames [6]. Post-processing uses the error voting technique (ROVER) at the character level to find the final result. ROVER aligns the results of all the frames that make up the same shot. The final result is calculated as the concatenation of the characters of each position with the greatest frequency.

There are studies on whether post-correction methods should be performed, or it is better to improve the input or the process to obtain better results. L. Liu *et al.* propose a study on the benefits of correcting errors after performing OCR, also proposing a post-correction method [31]. The analyses showed that it is more efficient both in cost and in accuracy to correct errors after the OCR method, considering that if it is done in the beginning or middle, more problems will arise which will be more difficult to solve. The proposed post-correction method uses the idea of self-learning and knowledge acquisition during its process to correct errors. Techniques are the conservation map, character pair matrix, lists of error-prone characters, and set of characters that can be confused. At first, these techniques are trained with the dictionary, but when OCR engine starts correcting,

it learns from errors and stores them.

T. Novikova *et al.* try to show that the accumulation of errors in previous stages makes errors not manageable in post-processing [38]. Authors propose to introduce processing in word-formation, similar to the post-processing described in other methods. The processing uses a lexicon and an n-gram based on the occurrences of characters to correct word errors. Although they try to show that their method is better than methods that use post-processing, the two cases produce similar results.

R. Smith proposes a study on whether doing post-processing after performing Tesseract v.2.x on scanned documents improves their results [49]. Tesseract uses a dictionary to find words, but for post-processing, it adds a new variable where the frequency of the words is stored. The post-processing corrects ambiguities that the language model cannot correct. Results show that using the dictionary and the frequency can improve some results but in those specialized documents results worsened. Also, considering the use of memory to store the new frequency, it is not feasible to perform post-correction. It is better to strengthen the Tesseract language model to improve the results, which was done at Tesseract v.3.04. In the proposed method, the frequency is stored in the structure of the dictionary, also, that all words do not go through post-correction, only those which do not meet some restrictions.

Although many methods presented show an improvement in accuracy using a dictionary, vocabulary, or lexicon, all presented the same problem: the accuracy worsened when the word does not exist in the dictionary or the words are proper names. According to M. Strohmaier *et al.* [52], for a dictionary to be perfect for post-correction, it would have to meet these three requirements:

1. The dictionary must contain all words to be searched, ensuring that any incorrectly recognized words can be corrected.
2. The dictionary should only contain the words you want to search, avoiding inappropriate corrections.
3. The dictionary must store the frequency of each word, to disambiguate among several possible candidates.

Table 5.13 shows the similarities and differences between the proposed methods and the dictionary-based post-correction methods.

### Context-based error correction

Since dictionary-based methods were deficient for words that do not exist in the dictionary or proper names, researchers began to look for other ways to correct the errors efficiently. Thus, context-based methods began to develop. Beginning to be more relevant when the ICDAR 2017 [8] and 2019 [41] competitions started with the Post-OCR Text Correction challenge.

Table 3.1: Similarities and differences between the proposed method and those described above.

Methods	Similarity	Difference
V. Levenshtein [30]	Consider insertion and deletion to correct words. Use edit distance to choose the result.	It is a theoretical proposal. The proposed method considers substitution, compound, and frequency to correct the word.
I. Guyon <i>et al.</i> [21]	They also consider the incorrect word as a possible result.	They use a VLMM to store the dictionary. They only allow substitution.
R. Beaufort <i>et al.</i> [3]	They use more than one dictionary to correct the words.	Only the characters recognized by the classifier are candidates for correcting errors.
K. Taghva <i>et al.</i> [53]	They bases on chain approximations.	Human interaction helps to learn the errors of the system.
J. Feild <i>et al.</i> [16]	They use the frequency to choose the best result.	All recognized words pass the post-processing.
F. Bai <i>et al.</i> [1]	They use the Trie structure to correct errors.	They use information from internal states of their deep neural network.
N. Sharma <i>et al.</i> [45]	They calculate the minimum ED to form the suggestions.	They use a lexicon with the 2000 most common words of the main languages.

H. Niwa *et al.* propose a post OCR method for scanned documents [37]. The correction depends on the grammar, vocabulary, and content of the documents. First, random words are extracted from the document, thus obtaining the topic of the document. Then errors are corrected considering the vocabulary and grammar of the language. Finally, the most appropriate candidate is selected, depending on the topic of the document. As there is no control over choosing keywords, sometimes they result in generic words or misrecognized words, which do not help to choose a topic. Our proposed method cannot use the topic of the text to be corrected since there is not enough information infer it.

Methods to correct the scanned documents use information from them to improve results. M. Bokser proposes an OCR engine that they divide into four phases: segmentation, image recognition, document analysis, and ambiguity resolution [4]. The ambiguity resolution stage corrects various errors in words that are found by the classification stage. The OCR engine uses the dictionary, the specific knowledge of the language and properties of the document to correct the errors.

Y. Bassil *et al.* propose an OCR post-correction method based on the question “Did you mean ..?” to enable online error correction with Google [2]. The process consists of fragmenting text blocks of 5 words each, considering that if there are sentences of 4 or 6 words, the method groups those sentences into blocks of 4 or 6. These blocks are sent as a query to the Google search engine if the result returns “Did you mean ..?”, so the original block is considered misspelled, and replaced with the suggested block, otherwise,

the original block is retained. This method avoids using dictionaries since they are not dynamic elements, new words cannot be added, in addition to not supporting proper names.

For S. Schulz *et al.*, the post-correction OCR can solve errors as a problem of translating incorrect text to correct text [42]. They propose to use an integrated Statistical Machine Translation model (SMT) at the token and character level. The model tries to assign each word of the original text a corrected word that will be considered as its exact translation. To avoid overcorrection (avoid incorrect alternation of an initially correct word), the word studied must appear in the corpus (a collection of written or spoken material used to discover how language is used) in combination with the previous and subsequent ones. With this information and with other document information, SMT decides if the original word should not be translated. The same dynamic approach is also used to choose the best candidate.

H. Hammarström *et al.* propose a language-independent Post-correction system, without supervision or human intervention to improve OCR results in historical documents [22]. The idea of the method is based on the fact that two words are variants of each other if their distribution similarity exceeds what is expected by chance of their similarity of form. If two words are similar, the change is made. Unlike many methods, this does not use a dictionary but information obtained from each line of the document that is stored in a vocabulary. Results prove to be very good at avoiding overcorrection, but it also leaves many errors that language information can correct.

J. Evershed *et al.* describe an unsupervised OCR correction system [15]. The system analyzes every 3 words to detect and correct errors. When an error is detected, the confusion matrix generates possible words. The confusion matrix is filled with the errors found in the training data without any human interaction. The method must train the confusion matrix with the training data of each database to have good results. If it does not retrain the results get worse.

Despite using the information in the document to correct the errors instead of isolated words, most approach still suffer from the ability to handle proper names. Jean-Caurant *et al.* propose an approach to correct errors in entity names from scanned documents [23]. The method consists of constructing a bi-similar graph in which entities found in the training are saved with their respective variations, for example, “Washington” and “Washingtan”. Then, entities are related depending on the relationships found in training. Results show great improvement in the errors of entities, even in those in which the human could not recognize. This is due to the relationships that a graph builds between entities. Nevertheless, the quality of the results depends a lot on the training. If an incorrectly recognized entity is not in training, it cannot be corrected accordingly. The training is done with many databases and scanned documents to prevent this from happening.

Unlike dictionary-based methods, these methods can only be used when you have a lot of information about the text from the scanned documents. The best results are obtained with methods where not only the word but the text is analyzed. These methods

cannot be applied directly to STR since these problems need a lot of information to work properly.

# Chapter 4

## Syntax-Based Techniques

As explained in Section 2.2.2, Tesseract returns a set of characters that represent a word for every bounding box. Tesseract also returns information of the words and every character that form the word. In this work, we show that using information from Tesseract and the dictionary, the accuracy of Scene Text Recognition can be improved.

### 4.1 Spelling correction

Tesseract 3.4 has good results in scanned documents, whether the text is presented in tables, in the vertical orientation, or in languages such as Japanese, Chinese, Hindu. But in scene images, the results are not relatively good as those that currently exist in the literature [7, 1, 46]. A careful analysis shows that some incorrect results of Tesseract 3.4 can be corrected through spelling techniques since various incorrect results have an Edit Distance of one or two when compared to the ground truth.

ED is used to measure the dissimilarity of a pair of strings by counting the number of operations required to transform one string into the other [1]. Correction actions, like substitution, insertion, or deletion, can leverage on ED to fix words according to the analyzed problem. For example, in Figure 4.1a, the word “*FREEDUM*” and ground truth “*FREEDOM*” have an edit distance of 1 since we need to replace the ‘*U*’ for the ‘*O*’. In Figure 4.1b, the word “*WARNNIGI*” and ground truth “*WARNING!*” have an edit distance of 2 since we need to replace the ‘*I*’ for the ‘*!*’ and insert the ‘*I*’. In Figure 4.1c, the word “*CULCHESJ’ER*” and ground truth “*COLCHESTER*” have an edit distance of 3 since we need to replace the ‘*U*’ for the ‘*O*’ and the ‘*J*’ for the ‘*T*’ and delete the ‘*’*’.

Table 4.1 shows the number of words that failed, considering edit distance of one, two and three, after running Tesseract 3.4 on the databases used in this work (Section 5.1). In columns  $ED \leq 1$ ,  $ED \leq 2$ , and  $ED \leq 3$  of Table 4.1, we show the number of words with edit distance of columns 1, 2, and 3 respectively between the incorrect results of Tesseract and the ground truth. Substitution, insertion, and deletion actions are considered to calculate the edit distance. Also, the table shows the increase in accuracy



Figure 4.1: Edit distance with images from the ICDAR 2013 Database. Blue characters indicate substitutions, red characters deletions, and green characters insertions.

if the errors were corrected. Libccv library is used for STD and Tesseract is used for STR, as explained in the Chapter 5.

Table 4.1: Number of words that failed in Scene Text Recognition.

Data Base	Accuracy Text recognition	ED $\leq 1$	ED $\leq 2$	ED $\leq 3$
ICDAR 2003	50.11%	95(7.16%)	141(10.63%)	149(11.23%)
ICDAR 2013	55.66%	162(19.10%)	300(35.38%)	386(45.52%)
IIIT5K	26.1%	211(10.55%)	299(14.95%)	331(16.55%)
IND	7.9%	166(3.69%)	303(6.74%)	971(21.60%)
ICDAR 2015	59.22%	35(6.52%)	57(10.61%)	64(11.92%)
KAIST	60.05%	33(8.51%)	51(13.14%)	63(16.24%)
SVT	32.84%	4(5.97%)	10(14.93%)	15(22.39%)

The accuracy of STR could be increased by running spelling actions on the incorrect results, as shown in Table 4.1. The structure of the dictionary and the code provided by Tesseract (*word\_in\_dawg* method - Algorithm 4.19) was used to implement the spelling corrections.

Listing 4.2a shows the *word\_in\_dawg* method, which determines whether the word (WERD\_CHOICE) is in the dictionary or not. The *word\_in\_dawg* method relies on the structure of Tesseract dictionaries to search for the word. The algorithm starts by determining if the searched word is an empty string or not, considering the word length ( $l$ ) (line 2). If  $l > 0$ , the word is formed in the DAWG (lines 4 to 13). The method uses the *NODE\_REF* variable to determine in which node is currently being analyzed. For each character from *character<sub>1</sub>* until *character<sub>l-1</sub>*, the method *edge\_char\_of* calculates the next node to visit using the current node and *character<sub>i</sub>* (line 5). If the edge is *NO\_EDGE*, it returns false, because the edge that leaves the indicated node with the specified character does not exist (line 6). For example, in Figure 4.2b, the word “plot” does not exist, since considering  $i = 3$  in node 5 of the DAWG, no edge represents

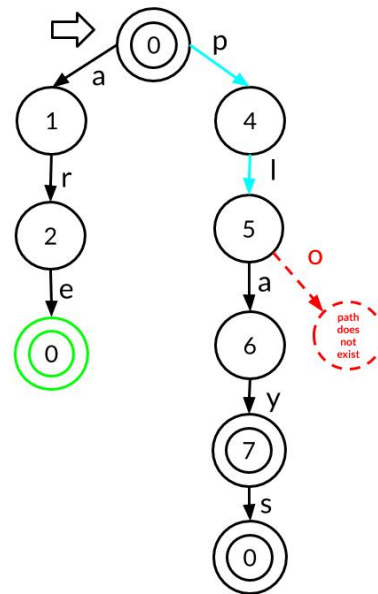
character “o”. Line 9 calculates the next node that the algorithm has to follow to form the searched word. If the node is 0, it returns false, because there are no words that can be formed since the previous node (line 10). For example, in DAWG 4.2b, the word “aren’t” does not exist since there is no path after node 0 painted in green. Finally, after the last character is consumed, the algorithm tests if it has reached an end node, line 14.

```

1  bool word_in_dawg(WERD_CHOICE &word) {
2      if (word.length() == 0) return false;
3      NODE_REF node = 0;
4      for (int i = 1; i < word.length(); i
5          ++){
6          EDGE_REF edge = edge_char_of(node,
7              word.unichar_id(i));
8          if (edge == NO_EDGE) {
9              return false;
10         }
11         node = next_node(edge);
12         if (node == 0) {
13             return false;
14         }
15     }
16     return edge_char_of(node, word.
17         unichar_id( word.length())) !=
18         NO_EDGE;
19 }

```

(a) Code



(b) Dawg

Figure 4.2: Annotated code for *word\_in\_dawg* method of Tesseract [?], that determines if the word is in the dictionary and a Dawg that represents the words: *are*, *play*, and *plays*.

The *spelling\_correction* method (described in Figure 4.3) uses the same idea as the one adopted by the method described above. The inputs are the words to be analyzed, which are stored into *word*. The algorithm starts at *character\_start* and goes until the *character\_start-1*, the node that represents the substring is stored into *node\_start*, and *edit\_dist* which means how many modifications have been made to the original word. We use the instruction *spelling\_correction(word,0,0,0)* to call the method for the first time. The output is the corrected word or the same word depending on the errors the method finds.

Unlike *word\_in\_dawg*, if the *spelling\_correction* method does not find a character (line 7), the algorithm must correct that character. Corrections may use the substitution method (Section 4.1.1), insertion method (Section 4.1.2), or the deletion method (Section 4.1.3). After correcting the character, these methods call the *spelling\_correction* method again to continue the process for the missing characters of the word. Every method returns a word with all the modifications and its edit distance (*edit\_dist* variable) (line 9). The *best\_replacement* method determines the best replacement which is the word

with the smallest edit distance (line 11). If the edit distance is greater than 40% of the word length, it will consider eliminating that process (line 3). All other parts of the algorithm are similar to the *word\_in\_dawg* method, considering that if the corrected word is not yet found in dictionaries, all the changes made must be discarded.

Line 3 in Figure 4.3 states that if the algorithm is allowed to modify more than half of the word's characters this would make the *spelling\_correction* method guess the word with few characters as clues. If more than 40% of the characters are modified, the method assumes that the word found by Tesseract is correct or Tesseract recognized symbols that are not words. In such case, the algorithm chooses to keep the word without making any modifications.

```

1 WERD_CHOICE spelling_correction (WERD_CHOICE word, int start, NODE_REF
  node_start, int &edit_dist){
2   if (word.length() == 0) return word;
3   if (edit_dist < word.length()*0.40) return word;
4   NODE_REF node = node_start;
5   for (int i = start; i < word.length() ; i++) {
6       ...
7       if ( edge == NO_EDGE ) {
8           int new_ed = edit_dist + 1;
9           WERD_CHOICE word_subs = substitution(word,node,i,new_ed);
10          ...
11          word = best_replacement(...);
12          return word;
13      }
14      ...
15  }
16  return word;
17 }

```

Figure 4.3: Annotated code for *spelling\_correction* method.



Figure 4.4: An image where *Libccv* library does not achieve good Text Detection.

We have noticed that problems show up when STD is used before Tesseract, as shown in Figure 4.4. These problems occur when characters in the word are irregularly separated,

causing bounding boxes to not correctly recognize the word limits. We have considered a new method to solve this problem called *compounder* (Section 4.1.4). Unlike *substitution*, *insertion*, and *deletion*, *compounder* is done at end of the Tesseract recognition phase, so it can leverage on the recognition information provided by it. Figure 4.5 shows the flowchart of Tesseract with the new methods described above.

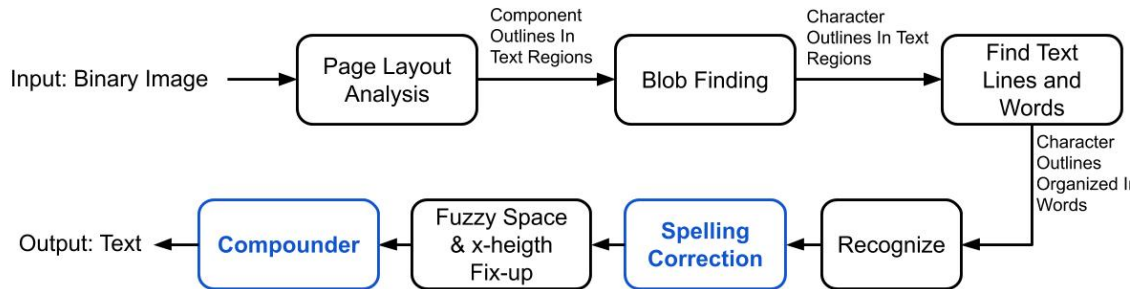


Figure 4.5: Engine architecture Tesseract OCR 3.4.x with the new methods Compounder and Spelling Correction.

### 4.1.1 Substitution

Substitution involves replacing characters so that the word gets corrected. Figure 4.1a shows an example where the “U” character is replaced by “O” so that *FREEDUM* becomes *FREEDOM*.

Several structures in Tesseract can help in the implementation of *substitution*. But to make it more efficient, the *NodeChildVector* structure will be used, since it has variables to store the necessary information. Figure 4.6 shows the *NodeChildVector* structure implemented by Tesseract [?]. *NodeChildVector* is a vector of *NodeChilds* (line 5). *NodeChild* is a structure that stores the character identifier (*UNICHAR\_ID*) and the edge that the character represents in the dawg (*EDGE\_REF*).

```

1 struct NodeChild {
2     UNICHAR_ID unichar_id;
3     EDGE_REF edge_ref;
4 };
5 typedef GenericVector<NodeChild> NodeChildVector;
  
```

Figure 4.6: Annotated code for *NodeChildVector* structure of Tesseract.

Figure 4.7 shows the implementation code of the *substitution* method. As input, the method receives the same data as the *spelling\_correction* method. The output is the modified word and the edit distance of the corrected word. To find the character to be replaced, it must verify if  $1 \leq i \leq l - 1$  or  $i = l$ , where  $i$  represents the position of the character to be modified and  $l$  the word length.

```

1  WERD_CHOICE substitution(WERD_CHOICE word, NODE_REF node, int i, int &
   edit_dist)
2  {
3      NodeChildVector new_character;
4      if (word.length() == i){
5          new_character=find_last_unichar(node);
6          if (new_character.size() > 0){
7              edit_dist = edit_dist + 1;
8              word.unichar_id(i) = new_character[0].unichar_id;
9              return word;
10         }
11     } else {
12         new_character = find_unichar(node, word.unichar_id(i+1));
13         if (new_character.size() > 0){
14             edit_dist = edit_dist + 1;
15             WERD_CHOICE word_replace, word_new = word;
16             int edit_dist_better = 0 , dist;
17             for (int j=0; j<new_character.size(); j++){
18                 dist=edit_dist;
19                 word_new.unichar_id(i) = new_character[j].unichar_id;
20                 word_new = spelling_correction(word_new, next_node(
                    new_character[j].edge_ref), i+1, dist);
21                 if (dist < edit_dist_better ){
22                     word_replace = word_new;
23                     edit_dist_better = dist;
24                 }
25             }
26             return word_replace;
27         }
28     }
29     edit_dist = word.length();
30     return word;
31 }

```

Figure 4.7: Annotated code for *substitution* method.

In the case of the last character of the word ( $i = l$ ), the method uses *find\_last\_unichar* to find all possible characters to correct the word (lines 4 to 10 from Figure 4.7). If the *find\_last\_unichar* method returns more than one character, as there is no way to tiebreaker two values (they have the same edit distance), the first character found is chosen (line 8 from Figure 4.7).

The *find\_last\_unichar* method explained in Listing 4.8a, receives as input the node representing the string formed from  $character_1$  to  $character_{l-1}$  and produces as output a NodeChildVector, where all the edges that can correct the word with their respective characters are stored. First, *find\_last\_unichar* finds all edges that leave the node using the *unichar\_ids\_of* method, storing all the resulting edges in the children variable (line 3). Given that *find\_last\_unichar* looks for the last character of the word, only the edges leading to a final node are stored (line 5). Figure 4.8b shows a DAWG that represents the words “play”, “plays”, “plan”, and “place”. The word found by Tesseract was “plad”, but

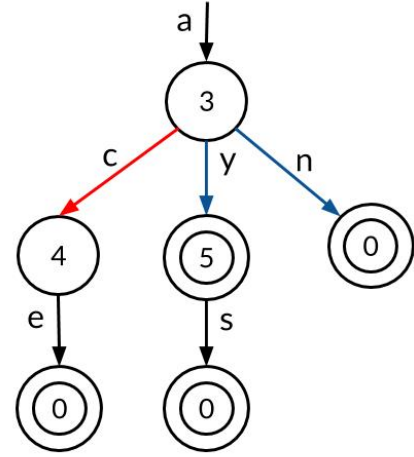
since it does not exist in the dictionary, the *spelling\_correction* method will correct it. Since the substring “pla” exists in the dictionary (node 3), only the last character needs to be corrected. The *unichar\_ids\_of* method gets 3 edges that leave node 3, which are “c”, “y”, and “n”. But only the edges represented by “y” and “n” are those that reach a final node, so they are the possible characters that can replace “d”.

```

1 NodeChildVector find_last_unichar(
2     NODE_REF node){
3     NodeChildVector children,
4     children_return;
5     unichar_ids_of(node,&children);
6     for (int i = 0; i < children.size();
7         i++) {
8         if (end_of_word(children[i].
9             edge_ref)){
10            children_return.push_back(
11                NodeChild(children[i].
12                    unichar_id,children[i].
13                    edge_ref));
14        }
15    }
16    return children_return;
17 }

```

(a) Code



(b) Dawg

Figure 4.8: Annotated code for *find\_last\_unichar* method. A Dawg that represents the words: *play*, *plays*, *plan*, and *place*.

If the character to be replaced is between positions 1 to  $l - 1$  (lines 10 to 27 from Figure 4.7), the substitution method uses the *find\_unichar* method to find all possible characters that can replace it. As there are still other characters to be analyzed after the replacement (line 18 from Figure 4.7), the substitution method calls *spelling\_correction* to analyze the missing characters ( $i + 1$  to  $l$ ). The substitution method uses edit distance to choose the best among the corrected words, which means the word with the lowest edit distance is the result. If the *substitution* method does not find characters, the edit distance takes the value of the word length and returns the word without any modification. The *find\_unichar* method can be implemented in two ways: (a) by analyzing the next character of the word; or (b) by considering all characters after the node.

Figure 4.9 shows the implementation of the *find\_unichar* method with the analysis of the next character of the word. The node that represents the formed substring until  $character_{i-1}$  and the *unichar\_id* of the  $character_{i+1}$  are the inputs. First, *find\_unichar* finds edges that leave the analyzed node (line 3). Then, for each such edges, *find\_unichar* traverses the reached nodes and finds again edges that leave them (line 5). Finally, *find\_unichar* compares the characters associated with the traversed edges are compared with the searched character, producing the matching edges as the method result (line 8).

Figure 4.10 explains how the *find\_unichar* method finds potential characters. The

```

1 NodeChildVector find_unichar(NODE_REF node, UNICHAR_ID unichar_id){
2   NodeChildVector children, children_return, children_in;
3   unichar_ids_of(node, &children);
4   for (int i = 0; i < children.size(); i++) {
5     unichar_ids_of(next_node(children[i].edge_ref), &children_in);
6     for (int j = 0; j < children_in.size(); j++)
7       if (unichar_id == children_in[j].unichar_id){
8         children_return.push_back(NodeChild(children[i].unichar_id,
9         children[i].edge_ref));
10        break;
11      }
12  }
13  return children_return;
14 }

```

Figure 4.9: Annotated code for *find\_unichar* method to find all characters considering the next character of the word.

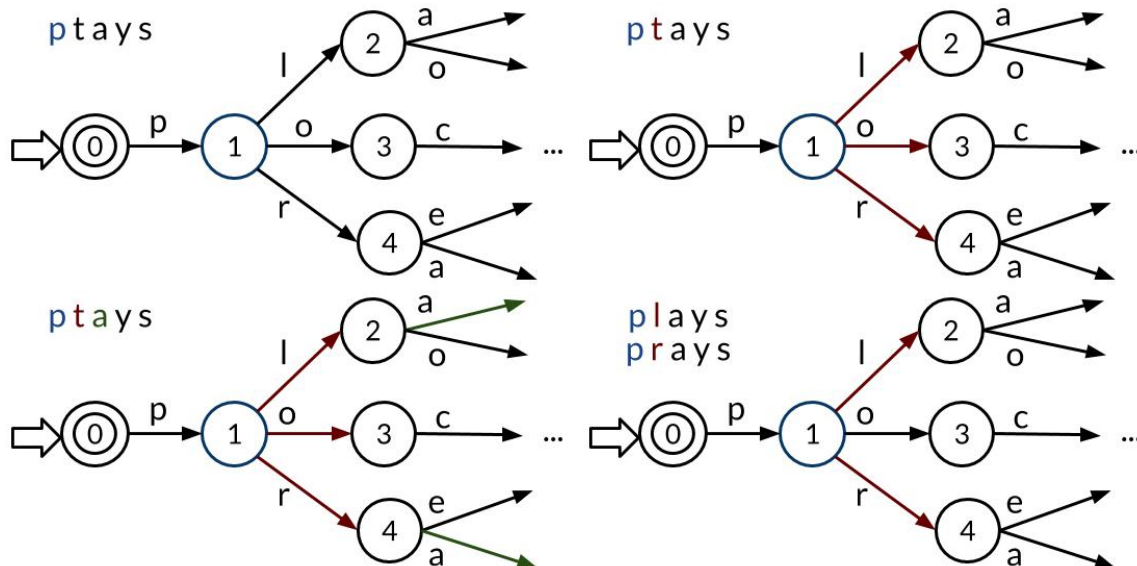


Figure 4.10: How the *find\_unichar* method finds potential characters, analyzing the following character.

top-left Dawg represents the words: “play”, “plot”, “pocket”, “pray”, and “precision”. The *substitution* method seeks to find all possible characters to replace the second character of the word “ptays” recognized by Tesseract, so that the method *find\_unichar*(1, “a”) is called. The top-right DAWG shows the edges that leave node 1. The bottom-left DAWG shows edges that match the “a” character. Finally, the bottom-right DAWG exposes edges that are potential replacements for the “t” character.

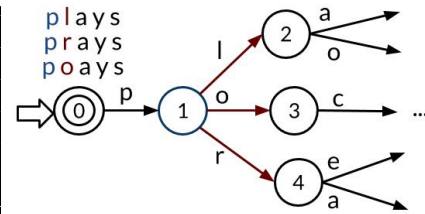
Listing 4.11a shows the implementation of the *find\_unichar* method, when considering all the characters after the node. Unlike the method that analyzes the following character, this method returns all the edges that follow the analyzed node without any restriction. If we continue with the example of Figure 4.10, the results are the edges of “l”, “o” and

```

1 NodeChildVector find_unichar(NODE_REF
  node){
2   NodeChildVector children;
3   unichar_ids_of(node, &children);
4   return children;
5 }

```

(a) Code



(b) Dawg

Figure 4.11: (a) Annotated code for *find\_unichar* method considering all characters after the node. (b) A Dawg that represents the words: *play*, *plays*, *plan*, and *place*.

“*r*”, as shown in Figure 4.11b. Notice that the edit distance is an important metric to determine the correct word when using this method.

### 4.1.2 Insertion

Insertion involves including a new character into the word to correct it. Figure 4.1b shows an example, where the “*I*” character is inserted between the “*N*” characters so that “*WARNNG*” becomes “*WARNING*”.

```

1 WERD_CHOICE substitution(WERD_CHOICE word, NODE_REF node, int i, int &
  edit_dist)
2 {
3   WERD_CHOICE word_res = word;
4   word_res.length() = word.length() + 1;
5   NodeChildVector new_character;
6   if (word.length() == i){
7     new_character=find_last_unichar(node);
8     if (new_character.size() > 0){
9       edit_dist = edit_dist + 1;
10      word_res.unichar_id(i+1) = new_character[0].unichar_id;
11      return word_res;
12    }
13  } else {
14    for (int k = i; k< word.length(); k++)
15      word_res.unichar_id(k+1) = word.unichar_id(k);
16    new_character = find_unichar(node, word_res.unichar_id(i));
17    if (new_character.size() > 0){
18      ...
19      return word_replace;
20    }
21  }
22  edit_dist = word.length();
23  return word;
24 }

```

Figure 4.12: Annotated code for *insertion* method.

The *insertion* algorithm (Figure 4.12) is similar to the one for the *substitution* method,

given its specific features. The input and output data are the same as for the *substitution* method. The first difference is in line 4. Given that the method inserts a new character the word length must be increased by one. If the inserted character is in the last position of the word, the only thing that would change is the position in which the method stores the new value, which would be  $i + 1$  and not  $i$  (line 10). If the character position is between 1 to  $l - 1$ , the characters after the position are moved to the right (line 14). The rest of the code is similar to the *substitution* method. In the insertion, you can also use two ways to find the missing character: (a) by analyzing the following character; or (b) by considering all the characters after the node.

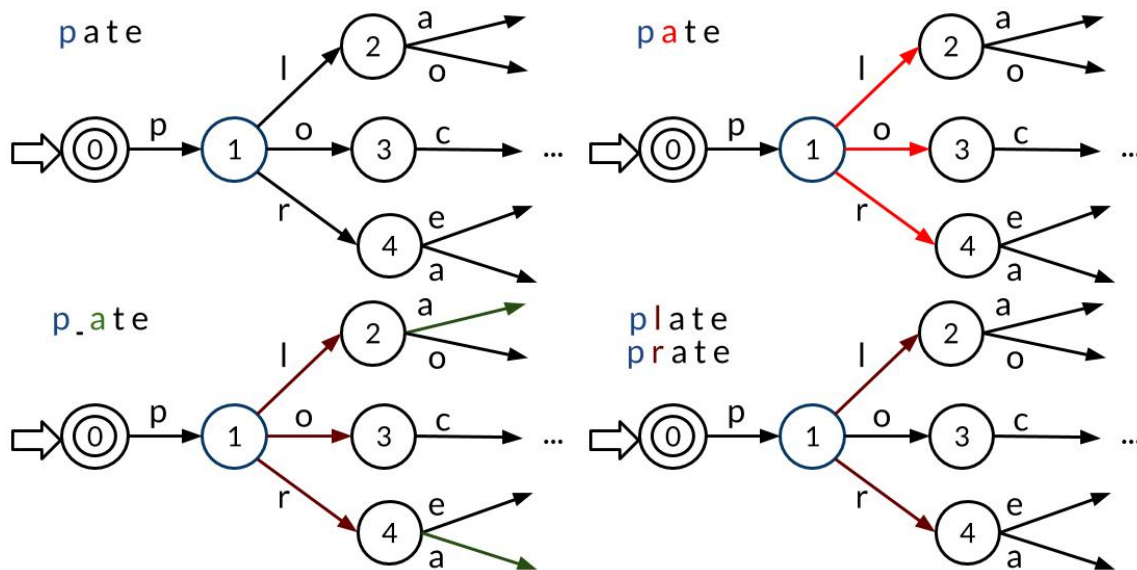


Figure 4.13: How the *insertion* method finds potential characters using the *find\_unichar* method with analyzing the following character.

Using the same example from Figure 4.10, Tesseract recognizes the word “pate” instead of “plate”. The *spelling\_correction* method recognizes that after node 1, there is no edge with the character ‘a’ (right-top Figure 4.13). Then *spelling\_correction* happens to make substitution, insertion, and elimination. Like the insertion is not in the last character, the insertion method moves all characters from “a” to “e”, one position to the right (left-bottom Figure 4.13). When the insertion method uses *find\_character* to analyze the following character, it only returns the edges “l” and “r” as a result, since they are the only ones that have character “a” labeled at its outgoing edges (right-bottom Figure 4.13). Finally, the spelling correction will analyze “plate” and “prate” words.

### 4.1.3 Deletion

Deletion involves removing the character from the word to correct it. Figure 4.1c shows an example, where the “ ’ ” character is removed and also two characters are replaced so that “culchesj’er” becomes “colchester”.

```

1 WERD_CHOICE elimination(WERD_CHOICE word, NODE_REF node, int i, int &
  edit_dist){
2     for (int k = i+1; k < word.length(); k++)
3         word.unichar_id(k-1) = word.unichar_id(k);
4     word.length() = word.length() - 1;
5     edit_dist = edit_dist + 1;
6     return spelling_correction(word, node, i, edit_dist);
7 }

```

Figure 4.14: Annotated code for *elimination* method.

Figure 4.14 shows the implementation code of the elimination method. The idea is that all characters from the position  $i + 1$  to  $l$  move one position to the left (line 2). Then to eliminate the remaining position, the word length is reduced by one (line 4). Finally, *spelling\_correction* is called again to analyze the missing characters and return the corrected word.

#### 4.1.4 Compounder

The *compounder* method aims at joining two string to form a single word. It is post-correction method that corrects the errors which were dragged from STD. Figure 4.4 shows an example. The idea is to use the values found in Text Detection and Text Recognition to join two strings. If they have any errors to correct, using *spelling\_correction* might correct the word.

```

1 void join(vector<string> &words, vector<Box> &boxes){
2     for (int i = 0; i < words.size(); i++){
3         for (int j = i + 1; j < words.size(); j++){
4             if (together(boxes[i], boxes[j])){
5                 string new_word = words[i] + words[j];
6                 if(word_in_dawg(new_word) == 0)
7                     new_word = spelling_correction(new_word, 0, 0, 0);
8                 if(word_in_dawg(new_word)){
9                     words[i] = new_word;
10                    boxes[i].w = boxes[i].w + boxes[j].w;
11                    boxes[i].h = boxes[i].h > boxes[j].h ? boxes[i].h :
                        boxes[j].h;
12                    words.erase(words.begin() + j);
13                    boxes.erase(boxes.begin() + j);
14                }
15            }
16        }
17    }
18 }

```

Figure 4.15: Annotated code for *compounder* method.

Figure 4.15 shows the implementation of the *compounder* method. As the method

tries to join words, it will go through the list of twice. The first *for* goes through all the words being considered (line 2). The second *for* goes through the words that can be joined with the word of the first *for* (line 3). For each pair of words, the *compounder* method determines whether the bounding boxes of the words are together or interposed through the *together* method (line 4). If the bounding boxes meet all the conditions of the *together* method, the *compounder* method creates the new word. If this new word is not in the dictionary, the *spelling\_correction* method corrects it (line 7). If the new corrected word is in the dictionary, it is stored (line 9). Also, the method modifies the width and height values of the new word (lines 10 and 11). If the new word is not in the dictionary or *spelling\_correction* cannot correct it, the method discards all the changes and continues to the next word.

## 4.2 Frequency

*Spelling\_correction* only uses edit distance to decide which is the best word among the corrected words. When two or more words have the same edit distance and they exist in the dictionary, there is no way to decide which one is the best. For example, in Figure 4.10, *spelling\_correction* can replace “*ptay*” by “*pray*” or “*play*” since both exist in the dictionary and have an edit distance of 1. One way to decide the tiebreaker between the words could be the frequency of the words. Considering also that Tesseract can generate more than one valid result in the dictionary, we can use the frequency in addition to the Tesseract rating to choose the best word. The frequencies of the words determine how many times a word appeared in a corpus. This work will use the Corpus of Contemporary American English (COCA) [13]. This corpus is probably the most widely-used corpus of English since it contains more than 560 million words of text. But how will Tesseract store the frequencies?

As other researches show, a direct acyclic graph can store other information in addition to the dictionary information, such as word frequency, cost of a function, among others. It depends on how the graph is implemented to store this new information. It can use another structure or modify the structure of the graph. For example, Tesseract uses the same DAWG structure to store additional information like, for example, if the direction of the word is stored inverted or not. Therefore to add a new information into Tesseract, it would be more convenient to store it into the DAWG structure, given that Tesseract’s code assumes that all the information that composes the dictionary is in a set of bit fields tagged at the DAWG edges. Moreover, adding a new structure would involve taking up more space in memory and increasing the time to retrieve the information from the new data structure.

The first versions of Tesseract stored information at the DAWG edge as 8 bits valued tags. This 8 bits tags worked fine for language dictionaries with few characters. But when Tesseract added languages with multi-character graphemes, 8 bits were insufficient to store a complete dictionary or relevant information to help with its handling. That’s

why in the latest versions, Tesseract stores such edge tags into 64 bits, even though in languages that have few characters such as Latin or Germanic script, several bits are not used.

Table 4.2: How various dictionary of Latin and Germanic origin languages provided by Tesseract use the 64 bits of every edge.

Languages	Number of characters	Next Node	Bits without use
English	$111 \approx 2^7$	$2,329,466 \approx 2^{22}$	32
Portuguese	$119 \approx 2^7$	$1,057,083 \approx 2^{20}$	34
Spanish	$110 \approx 2^7$	$1,402,329 \approx 2^{21}$	33
Spanish Old	$137 \approx 2^8$	$1,553,611 \approx 2^{21}$	32
French	$143 \approx 2^8$	$1,239,817 \approx 2^{21}$	32
Italian	$123 \approx 2^7$	$933,074 \approx 2^{19}$	35
Romanian	$106 \approx 2^7$	$468,943 \approx 2^{18}$	36
German	$118 \approx 2^7$	$1,376,864 \approx 2^{20}$	34
Dutch	$153 \approx 2^8$	$1,332,559 \approx 2^{20}$	33
Polish	$116 \approx 2^7$	$823,601 \approx 2^{19}$	35
Norwegian	$103 \approx 2^7$	$471,985 \approx 2^{18}$	36
Albanian	$107 \approx 2^7$	$350,225 \approx 2^{18}$	36
Danish	$103 \approx 2^7$	$355,498 \approx 2^{18}$	36
Bulgarian	$96 \approx 2^6$	$334,929 \approx 2^{18}$	37

Table 4.2 shows how several languages use the 64 bits at every edge. The second column represents how many characters the language has. For example, the English language has 111 characters meaning that 7 bits are used to represent all the characters. The third column shows how many nodes Tesseract are needed to represent the dictionary. The fourth column describes how many edge tag bits remain unused, considering those used to represent the characters, the 3 bits used as flags, and the bits used to store the next node.

Table 4.2 shows that there are 32 bits left at the edge that can be used to store new information in Tesseract dictionaries for languages of Latin origin (Portuguese, Spanish, French, among others) and languages of Germanic origin (English, Polish, German, among others). Acknowledging that the number of nodes is proportional to the number of words that exist in the dictionary, using 32 bits would exclude those dictionaries that have more words than the already formed Tesseract dictionaries. To avoid problems with dictionaries that have more words Tesseract, 23 bits will be taken after the flag to store the frequency. Figure 4.16 shows the new distribution of the 64 bits.

Since the new edge tag is a decimal number (single precision floating point), its values range from  $3.4 \times 10^{-38}$  to  $3.4 \times 10^{38}$ . A decimal stored into a floating variable considering IEEE 754, occupies a 32-bit memory space, which is divided into a sign (1 bit), exponent (8 bits), and mantissa (23 bits) [56]. Figure 4.17 shows the distribution of a decimal number. The sign bit determines if the decimal is positive (zero) or negative (one). The

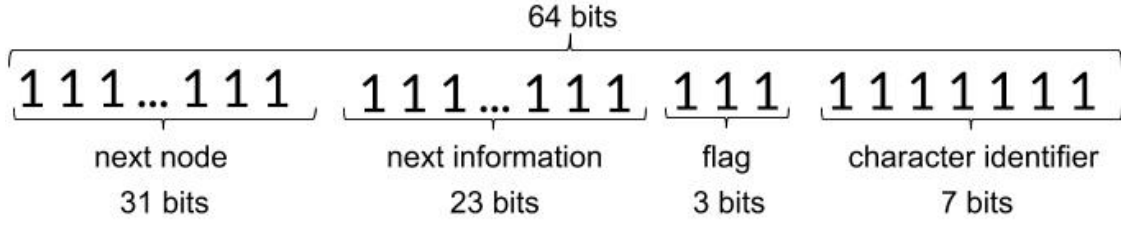
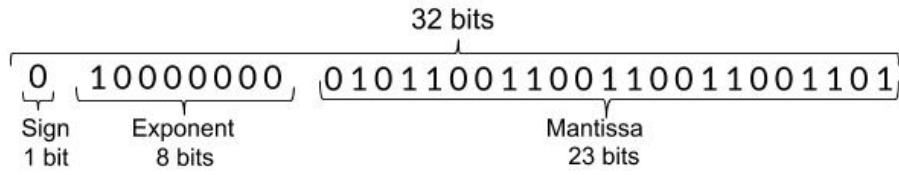


Figure 4.16: New distribution of the 64 bits that represent the same edge of Figure 2.12

exponent represents the exponential power. The mantissa is the actual number. The portion after the decimal point in the mantissa is obtained by adding each digit of the number by multiplying each digit by a power of two.



$$(-1)^{Sign} \times 1.(Mantissa) \times 2^{Exponent} = (-1)^0 \times 1.01011001100110011001101 \times 2^{10000000}$$

Figure 4.17: Representation of the decimal number 2.7 in a float variable.

Some bits must be removed, without changing the representation of the decimal number, so as to fit 32 bits into 23 bits. An accuracy analysis of the mantissa and exponent is required in order to evaluate the impact of fitting the 32 bits into 23. If the stored tag is always only positive or negative numbers, the sign bit can be eliminated. If any bit in the exponent is removed, it would completely modify the value. If the bits are removed at the end of the mantissa, some precision decimals are lost. Therefore, in our specific case, the sign bit could be discarded and 8 bits removed at the end of the mantissa. Figure 4.18 shows how the decimal number 2.7 would store a tag into an DAWG edge structure. As some precision bits of the mantissa are eliminated, the value represents  $2.69995 \approx 2.7$ .

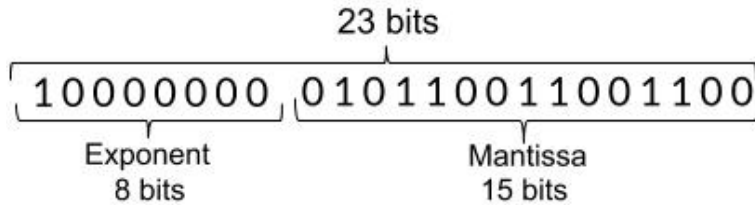


Figure 4.18: Representation of the decimal number 2.7 that it will store in the structure Dawg.

Figure 4.19 shows how the representation of a decimal number is transformed to be stored as a frequency tag into a DAWG's edge. The *set\_frequency\_in\_edge* method

receives as inputs the new decimal number and the respective edge identifier represented by the *EDGE\_RECORD* variable. The bit values of the *mantissa* and the *exponent* are determined (lines 5 and 6). The *mantissa* is reduced by 9 bits (line 8). The edge tag is created, and the *mantissa* and *exponent* variables are packed (line 11). Finally the frequency tag is stored into the corresponding edge in the graph (lines 13 and 14).

```

1 #define NUM_MANTISSA_BIT          23
2 #define NUM_REDUCE_MANTISSA      8
3 #define NUM_EXPONENT_BIT         8
4 void set_frequency_in_edge(EDGE_RECORD *edge_rec, float weight){
5     // Obtain mantissa and exponent of float
6     uint64 mantissa = get_mantissa(weight);
7     uint64 exponent = get_exponent(weight);
8     // Find new value of mantissa
9     mantissa = mantissa >> NUM_REDUCE_MANTISSA;
10    // Obtain new value
11    uint64 number_weight = mantissa | (exponent << (exponent_start_bit_
        - NUM_EXPONENT_BIT));
12    // Store the new value in the edge
13    *edge_rec &= (~weight_mask_);
14    *edge_rec |= ((static_cast<EDGE_RECORD>(number_weight) <<
        weight_start_bit_) & weight_mask_);
15 }

```

Figure 4.19: The code how of a decimal number is obtained to store in the structure of Tesseract dictionary.

Figure 4.20 shows the code used to store the edge data. Represent the split of a float variable into its *sign*, *exponent*, and *mantissa* (lines 5 through 13). The *get\_frequency* method receives as input the identifier of the edge from which we want to obtain the data. Retrieves the representation of the edge information (line 16). Retrieve the sign, exponent, and mantissa (lines 18 to 21). The sign is zero since the stored frequency tag is always positive. Eight leading zeros are added to the right to complete the mantissa to 23 bits. Finally, method returns the float (line 23).

Our analysis of Tesseract’s recognition phase revealed other problems besides storing the frequency tag at the DAWG edges. A recurrent one show up when different frequency values associated with the same word, which are written using different letter cases. For example, the frequency of “are” or “Are” is 2568218 while, the frequency of “ARE” is 181. Figure 4.21 shows a problem if Tesseract recognizes “ARE” as “ARW”. *spelling\_correction* corrects the word and considers ARC or ARE as possible replacements for “ARW”. Since “ARC” and “ARE” have an edit distance of one from “ARW”, *spelling\_correction* uses the frequency as a tiebreaker. As the frequencies of “ARE” and “ARC” are 181 and 5450 respectively, the final result is “ARC”. Method *spelling\_correction* should choose “ARe” since “are” is one of the most popular words in English, but given that “ARE” is in uppercase, *spelling\_correction* cannot choose it.

Using different letter cases in a dictionary formed through DAWG causes the words

```

1 #define MANTISSA_START_BIT      0
2 #define NUM_EXPONENT_BIT       8
3 int weight_mask_, weight_start_bit_, exponent_mask_, mantissa_mask_;
4 // To obtain mantissa. exponent and sign of a variable float
5 typedef union {
6     float f;
7     struct
8     {
9         unsigned int mantissa : 23;
10        unsigned int exponent : 8;
11        unsigned int sign : 1;
12    } val;
13 } myfloat;
14 float get_frequency(EDGE_RECORD &edge_rec){
15     // Get the value of the new information in the edge
16     uint64 num_weight=((edge_rec & weight_mask_) >> weight_start_bit_)
17     << NUM_EXPONENT_BIT;
18     // Get the sign, exponent and mantissa
19     myfloat var;
20     var.val.sign=0;
21     var.val.mantissa=(( num_weight & mantissa_mask_)>>
22     MANTISSA_START_BIT);
23     var.val.exponent=(( num_weight & exponent_mask_)>>
24     exponent_start_bit_);
25     // Return value
26     return var.f;
27 }

```

Figure 4.20: The code how of a decimal number is obtained to store in the structure of Tesseract dictionary.

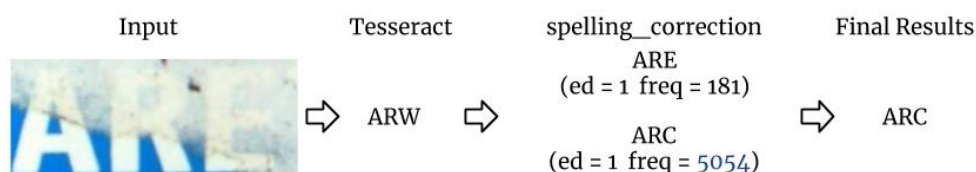


Figure 4.21: The problem of using different frequencies for the same word with different letter cases.

“ARE”, “Are”, and “are” to be considered by Tesseract as different. That means, if Tesseract recognizes “ArE”, “ARe”, or “aRe”, it keeps looking for a good result because the previous results are not in the dictionary so they are not good results. Also, this problem can result in errors in the *spelling\_correction* method. For example, Tesseract recognized the text in Figure 4.22 as “trespassing”. As recognition is wrong, *spelling\_correction* corrects it. But in the dictionary, there is no “trespassing” but “Trespassing”, so the final result of *spelling\_correction* is “trespass”, which is a wrong result.

In order to address this problem, we pre-process the dictionary accesses so that all

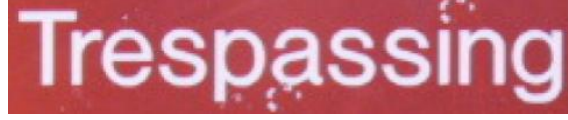


Figure 4.22: Image from the ICDAR2013 Database, where the ground truth is “*Trespassing*” and the result of Tesseract is “*trespassing*”.

words and their corresponding frequencies are stored in lower case. The final frequency of the word would be the result of Equation 4.1. If two words share the same final state (Figure 4.23), DAWG will add all frequencies and store in the graph. Tesseract code is modified so that each time the dictionary is accessed, all words are converted to lowercase. Also, recognized words are stored in lowercase to avoid the problem of recognizing the same word but with different cases.

$$Frequency_{total} = Frequency_{upper\_case} + Frequency_{lower\_case} \quad (4.1)$$

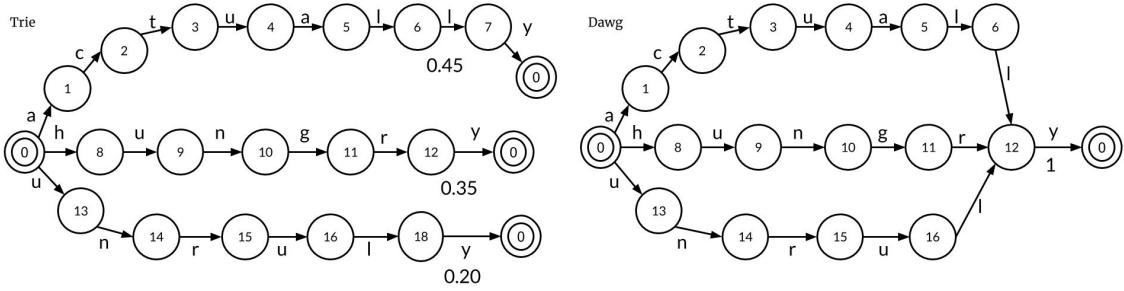


Figure 4.23: Dictionary formed by words “*actually*”, “*hungry*”, and “*unruly*” with their respective frequencies 95, 75, and 30. The left side shows how the dictionary uses the Trie structure. The right side shows how the dictionary uses the Dawg structure.

### 4.3 Finding the error

Tesseract does not perform a spelling correction of its word output, because the correction with a binary dictionary (if the word exists or does not exist in the dictionary), depends on the size of the dictionary [49]. Notice that for Tesseract, a recognized word may not necessarily be in the dictionary. For example, street names, store names, proper names, technical words, among others, are not in Tesseract’s dictionary. Consider, the example of Figure 4.24. It shows that Tesseract correctly recognizes the word NAD, but in the Tesseract dictionary for the English language, there is no NAD word. As it does not exist in the dictionary, the *spelling\_correction* method tries to correct the word and returns the word MAD. So Tesseract with *spelling\_correction* gives a final incorrect result. As a consequence, a new approach to deal with such error needs to be developed.

In order to work properly, method *spelling\_correction* depends on the fact that the ground truth exists in the dictionary and that it includes all words in the language. But

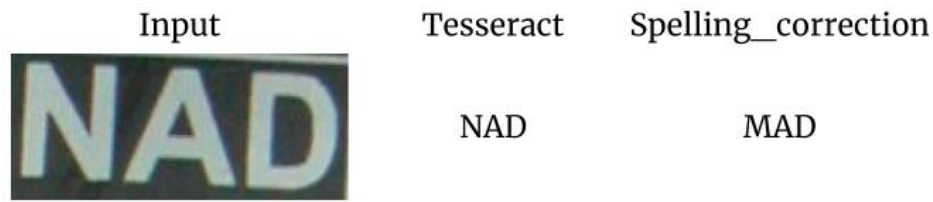


Figure 4.24: Tesseract with the *spelling\_correction* method gives an incorrect result because the recognized word is not in the dictionary.

since a dictionary cannot include all words, a way must be found to prevent all words recognized by Tesseract from being corrected by *spelling\_correction*.

### 4.3.1 Tesseract decision

As explained in the Section 2.2.2, the `WERD_RES` variable stores all the information of the text that Tesseract recognizes. `WERD_CHOICE` stores every possible text that represents the image. At the end, `WERD_RES` can have one or more `WERD_CHOICE`, given that Tesseract's classifier can recognize a symbol as more than one character, and the Language model can generate more than one possible word.

Tesseract has a way of knowing if a `WERD_RES` can be accepted as a good result or not. Figure 4.25 shows the *AcceptableResult* method where Tesseract decides if a `WERD_RES` is a good result. *AcceptableResult* uses variables such as *stopper\_nondict\_certainty\_base*, *reject\_offset\_*, *stopper\_smallword\_size*, *stopper\_certainty\_per\_char* which consolidate information obtained from its Machine Learning model training phase. Tesseract decides if `WERD_RES` is a good result if the best `WERD_CHOICE` of `WERD_RES` is greater than a threshold (*CertaintyThreshold*) (line 9). Finally, Tesseract stores this value into *tess\_accepted*, so it can be used in other methods.

```

1 bool AcceptableResult(WERD_RES* word) {
2     float CertaintyThreshold = stopper_nondict_certainty_base -
        reject_offset_;
3     if (valid_word(*word->best_choice) && case_ok(*word->best_choice)) {
4         int WordSize = LengthOfShortestAlphaRun(*word->best_choice);
5         WordSize -= stopper_smallword_size;
6         if (WordSize < 0)
7             CertaintyThreshold += WordSize * stopper_certainty_per_char;
8     }
9     if (word->best_choice->certainty() > CertaintyThreshold)
10         return true;
11     else
12         return false;
13 }
```

Figure 4.25: Annotated code for the *AcceptableResult* method of Tesseract.

Using the *tess\_accepted* variable, Tesseract can restrict the use of the *correction\_spelling*

method. If *tess\_accepted* == 1 (Tesseract considers WERD\_RES as good result), *spelling\_correction* does not correct WERD\_RES. If *tess\_accepted* == 0 (Tesseract considers WERD\_RES as bad result), *spelling\_correction* corrects WERD\_RES. Despite using *tess\_accepted*, to restrict the use of *spelling\_correction*, it still fixes words correctly recognized by Tesseract that do not exist in the dictionary, given that *tess\_accepted* depends on whether the word exists in the dictionary or not (*word\_in\_dawg*).

### 4.3.2 Tree best variable

Another way to decide if WERD\_RES is a good result or not is to use all the information that WERD\_RES stores. But since WERD\_RES does not store the information of the text that represents the image, it is also necessary to use the information of the best WERD\_CHOICE. Using the information provided by Tesseract, we can construct a model to determine which WERD\_RES is a correct result and which is not.

The information of WERD\_RES used to construct the model are: *tess\_accepted* (if the result is good), *x\_height* (the average height of the characters), *caps\_height* (the average height of the characters upper cases), *blob\_widths* (the average width of the characters), and *blob\_gaps* (the average gaps between the characters). The information of the best WERD\_CHOICE used to construct the model are: *rating* (the sum of the ratings of the individual blobs in the word), *certainty* (the certainty of the word), *certainties* (the sum of certainty of each character that forms the text), *adjust\_factor* (a factor that was used to adjust the rating), *min\_x\_height* (the minimum height of the characters), and *max\_x\_height* (the maximum height of the characters).

The models created to decide whether a WERD\_RES is a good result or not are logistic regression and Support Vector Machine (SVM). These models are trained with the training data of various databases shown in the Section 5.1. Other models were trained since the results of both logistic regression and SVM are similar to the results of the *tess\_accepted* variable.

A preliminary analysis of these variables revealed that *tess\_accepted* is the most important variable, but when we use the Decision Tree method to see which is the most important variable, it does not show the *tess\_accepted*. Figure 4.26 shows the most important condition to define whether a result is correct or not with the Decision Tree method using the same training data used to create the Machine Learning models.

We used all the training words from the databases to carry out the model. The word characteristics used are all the information Tesseract provides, Section 2.2 shows them. We classified manually of results of the training images between correctly recognized and incorrectly recognized to perform the model training.

After manual classification, we obtained 9365 data. 3314 data (35.39%) were correctly recognized, while 6051 data (64.61%) were incorrectly recognized. So the model will always favor that the recognized result is incorrect. Although the ideal would have been half correctly recognized words half wrong. Tesseract 3.04.1 does not have a good classifier, so increasing more databases would imply increasing more incorrectly recognized

words.

The characteristics of the model generated by Decision Tree are:

- The function to measure the quality of a division is “*gini*” [24].
- We do not consider any depth of the tree. The nodes expanded until the leaves were pure.
- No method of pruning is used.

The most representative variable to determine if a WERD\_RES result will be considered as good or not is the certainty variable of the best WERD\_CHOICE of the WERD\_RED. Using this information, we can create a condition to decide if WERD\_RES can be considered as the final result or has to go through the correction to get a good result.

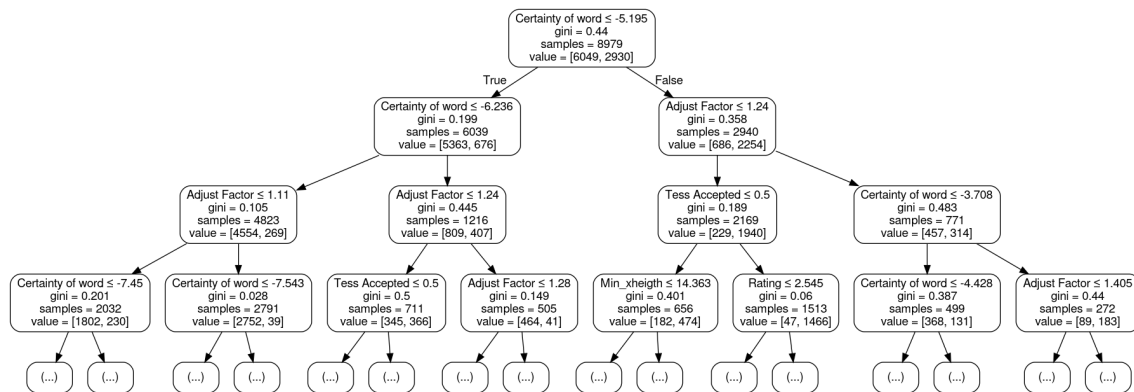


Figure 4.26: Decision Tree with training data from the section databases 5.1. The *sklearn* library with *python* was used to obtain the Decision Tree.

```

1 void spelling_correction_final (WERD_RES word){
2   WERD_CHOICE_IT choice_it(&word->best_choices);
3   if (word_in_dawg(word->best_choice))
4     return;
5   if (word->tess_accepted)
6     return;
7   if (word->best_choice->certainty() > -5.47)
8     return;
9   for (choice_it.mark_cycle_pt(); !choice_it.cycled_list(); choice_it.
      forward()) {
10    WERD_CHOICE text = choice_it.data();
11    text = spelling_correction(text,0,0,0);
12  }
13  dictionary_correction_pass(word);
14 }

```

Figure 4.27: Annotated code for the *spelling\_correction\_final* method.

Figure 4.27 shows the final implementation of *spelling\_correction\_final*, which is the method that other Tesseract methods will call. The input is `WERD_RES`. It has no output since the method corrects the `WERD_CHOICES` in place at `WERD_RES` itself. Line 2 shows variable `WERD_CHOICE_IT` that is used to iterate over all the `WERD_CHOICES` of `WERD_RES`. If the best `WERD_CHOICE` is in the dictionary, *spelling\_correction* does not correct the `WERD_RES` (line 3). Line 5 and 7 show the conditions that `WERD_RES` must meet so that the spelling corrects it. The restriction on line 5 shows that if `WERD_RES` is a good result, it will no longer make modifications. The restriction of line 7 is found by using the Decision Trees. If `WERD_RES` does not have any restrictions, the *spelling\_correction* method tries to correct all the `WERD_CHOICES` (lines between 9 and 12). Finally, Tesseract's *dictionary\_correction\_pass* method is used, which modifies the results of `WERD_RES` so that the best `WERD_CHOICE` is a word in the dictionary.

## Chapter 5

# Experimental Evaluation

This chapter describes and analyzes the experimental results achieved from the techniques proposed in the previous chapter. Seven databases are used to evaluate the effectiveness of the proposed spelling correction methods and frequency. The main findings from this evaluation are:

- Tesseract incorrectly recognizes some word's characters, due to various problems in the image such as noise, blur, lighting, or symbols close to characters. The substitution method can solve this problem, but using the substitution in all words that do not exist in the dictionary can causes overcorrection.
- Sometimes the Text Detection library does not locate all characters in the word, Tesseract does not find all blobs to form words, nor it recognizes symbols as characters of the word. These new or missing characters usually occur in the first or last part of the word. To correct it, we can use the insertion and deletion methods. We cannot use restrictions like the substitution method, because restrictions depend on the characters that exist in the word.
- Scene Text Detection sometimes finds a word separately, that is, two bounding boxes covering the same word. Sometimes Text Detection detects the first part of the word as a result and the remaining characters as another different result. It also happens with Tesseract that two WERD\_RES represent the same word: one that contains the first part of the word and the other the second. The compounder method connects these two parts of the word to correct the problem.
- Tesseract uses a metric known as rating to choose the best WORD CHOICE. The lower the rating the more suitable for correctness the word is. This rating uses the uncertainty of each character as well as an adjustment. This adjustment depends a lot on whether the word exists in the Tesseract dictionary or not. Sometimes when two words are in the dictionary, Tesseract rests exclusively upon the uncertainty of the characters to come up with a decision. In this work, when this problem occurs, we use the word frequency as a tie-breaker or even replace the rating as a decision metric.

## 5.1 Dataset

Four out of the seven used databases are focused on the Text Recognition task, that is, the images are already cropped texts. As images are only text, they do not need to go through the Scene Text Detection, so they are passed directly to Tesseract. Figure 5.1 shows the features of these four databases.



Figure 5.1: Images cloud of the 4 databases. From top to bottom, from right to left: ICDAR 2003, ICDAR 2013, INCIDENTAL SCENE TEXT 2015, and IIT 5K - WORD.

Table 5.1 reports a summary of the four databases for the Scene Text Recognition task.

Table 5.1: Four databases for the Scene Text Recognition task.

#	Database	# training	# testing	Accuracy (%)
1	ICDAR 2003 [32]	1327	529	50.11%
2	ICDAR 2013 [26]	848	1095	55.66%
3	INCIDENTAL SCENE [25]	4468	2077	7.90%
4	IIT 5K-WORD	2000	3000	26.10%

The ICDAR 2003 database was created for the competition with the same name. The images were captured from a wide variety of digital cameras, to have images with different resolutions. The ICDAR 2013 competition was developed for 3 tasks: Text Locating, Character Recognition, and Word Recognition. The objective of the word recognition is to find a system that allows reading a single word extracted from a captured scene. Two entries are given for this task, the images that a word contains to recognize and a dictionary with all the words in the database. The result with Tesseract was 50.11%, despite having good resolution images, there are also very low-resolution images that Tesseract cannot correctly recognize.

The ICDAR 2013 database, also known as Focused Scene Text, was developed for the competition with the same name. ICDAR 2013, like the ICDAR 2003 competition, has 3 challenges: Reading Text in Born-digital Images, Reading Text in Scene Images, and Reading Text in Videos. For this task, only the databases of the second challenge (Reading Text in Scene Images) are used. The images generally present reasonably focused text, and in most cases, they have a horizontal orientation. The result with Tesseract was 55.66%.

Incidental Scene Text 2015 (IND), is a database similar to ICDAR 2013. Unlike the ICDAR 2013 database, the images are text cut from real scene images where the user does not consider any previous action so that the text has good quality as the text centered on the photo. Therefore, images are low resolution, without good lighting and indistinguishable text. The result with Tesseract was 7.90%. Its low result was because the images are of very low resolution, some are so small that not even the human eye can recognize, and most of the images are proper names, where Tesseract does not recognize correctly.

The 5K words IIIT database was obtained from Google image search. This database focuses on collecting words like billboards, signboard, house numbers, house name plates, movie posters, and proper nouns that are found on the street. Words in images were manually annotated with bounding boxes and corresponding ground truth words. The result with Tesseract was 26.10%. Its low result was because the images are of very low resolution since most of the images are proper names. Also, they have a lot of noise, since they are from the street.

Three of the seven databases are specific to the Scene Text Detection and Recognition task. In order to use Tesseract in these databases, it is necessary to first use a library that does Scene Text Detection to obtain the bounding boxes that contain the text of the analyzed images. Figure 5.2 shows some examples of images from these databases for Scene Text Detection and Recognition.

Table 5.2 shows a summary of the three databases for the Scene Text Detection and Recognition task.

Table 5.2: Three databases for the Scene Text Detection and Recognition task.

#	Database	# Images training	# Images testing	Accuracy (%)
1	ICDAR 2015 [25]	233 (1080)	229 (1025)	59.22%
2	SVT [55]	100 (257)	249 (647)	32.84%
3	KAIST <sup>1</sup>	278 (704)	120 (303)	60.05%

ICDAR 2015 for the End-to-End challenge is a database developed for the ICDAR 2015 competition. Unlike other ICDAR competitions, this database is for End-to-End systems, that is, those that perform Scene Text Detection and Recognition. In addition to the images with their respective transcriptions and location ground truth, the database provides a generic vocabulary of 90k word. This vocabulary does not include all the



Figure 5.2: Images cloud of the three databases. From top to bottom, from right to left: ICDAR 2015, The Street View Text (SVT), and KAIST Scene Text.

words in the images since this vocabulary does not contain alphanumeric structures and punctuation marks. The result with Tesseract was 59.22%. Its result was because the images have a complex background.

The Street View Text (SVT) dataset consists of a group of images obtained from Google Street View. The images have three characteristics: all are from the street, all are from business signage, and have a low resolution. The objective of the database is: given a street view image, identify nearby business words. The result with Tesseract was 32.84%. Its result was because the images have a lot of noise since they are extracted from Google street. They have a complex background and also low-resolution.

KAIST scene text dataset contains images obtained from the streets and shops in Korea. KAIST is categorized into 3 languages: Korean, English, and mixed. The images were captured considering outdoors and indoors scenes, different lighting conditions (daylight, night, artificial light, among others), and with different resolutions (digital cameras with high resolution or cameras with low resolution). The result with Tesseract was 60.05%. Its result was because the images have different resolutions and lightings.

As Scene Text Detection does not always find all the texts in the image, especially in images that are low resolution. A manual cropped is done to all the images in the databases with the Scene Text Detection and Recognition tasks to generate only images focused on a word. This enables the use of all image texts to test the proposed methods.

In the following sections, the images present in the described datasets are used to conduct the experiments required to evaluate the methods proposed in this work. All experiments were implemented in the C++ programming language version 2.7.6 with

the following libraries: Tesseract v3.4 <sup>2</sup>, OpenCV <sup>3</sup>, and Libccv <sup>4</sup>. The experiments were conducted on a machine with  $\times 86\_64$  processor (8 cores, hyper-threading enabled), 31GB of RAM and clocked at 2.82GHZ. The machine runs a standard Ubuntu OS 16.04 LTS.

## 5.2 Evaluation Metrics

Benchmarks such as ICDAR make separate evaluations of Text Detection and Text Recognition, since both tasks have their challenges, results and metrics.

### 5.2.1 Scene Text Recognition

Metrics that are used to determine if the method has a good result for Scene Text Recognition are: (a) Total Edit Distance; (b) Word recognition accuracy; and (c) Correct Full Sequences [57]. Figure 5.3 shows examples of these metrics.

$$\begin{array}{l}
 \text{WRA} \\
 \left. \begin{array}{l} \text{gt: the sky is blue} \\ \text{p: the sky in blue} \end{array} \right\} \text{WRA} = \frac{3}{4} = 0.75 \\
 \\
 \text{CFS} \\
 \left. \begin{array}{l} \text{gt: the sky is blue} \\ \text{p: the sky in blue} \end{array} \right\} \text{CRW} = \frac{0}{1} = 0.00
 \end{array}$$

Figure 5.3: Examples of Text Recognition metrics.

*Word recognition accuracy (WRA)* is calculated by the formula:

$$WRA = \frac{|\text{correct recognized words}|}{|\text{ground truth number}|} \quad (5.1)$$

In the databases for Scene Text Detection and Recognition, STR results depend on how many words were found by Scene Text Detection. In these cases, *WRA* is calculated by the formula:

$$WRA = \frac{|\text{correct recognized words}|}{|\text{correct found word by Text Detection}|} \quad (5.2)$$

*Correctly Recognized Sequence (CFS)* is calculated by the formula:

$$CRS = \frac{|\text{correct recognized sequences}|}{|\text{number of sequences}|} \quad (5.3)$$

<sup>2</sup><https://github.com/tesseract-ocr>, last accessed on 15/01/2020

<sup>3</sup><https://opencv.org/>, last accessed on 15/01/2019

<sup>4</sup><http://libccv.org/>, last accessed on 01/12/2018

As the image texts of databases are recognized as individual words, the CRS metric cannot be used. If individual words were considered sentences, the same WRA metric results would be obtained. Therefore, this work only uses the WRA metric.

## 5.3 Spelling correction

In this section, we describe the results obtained using the spelling correction methods, as well as the combination of them. Experiments running on Scene Text Recognition databases and Scene Text Detection and Recognition databases are typically used in different ways. We only use Tesseract databases for the task of Text Recognition. In such case, for databases where Text Detection is necessary to recognize the text (e.g. SVT and KAIST), the original images are cut manually to produce text focused images in order to have more data for training and test.

In this work, we use Libccv and Tesseract on Text Detection and Recognition databases. In all cases, only the training data is used to perform the analysis.

### 5.3.1 Substitution

In this subsection, we present the results obtained using only the substitution method (Figure 4.7) to correct errors through spelling. (Figure 4.3). In our experiments, we consider the two methods to find the new character: analyzing the next character of the word and analyzing all the characters. Case sensitivity problems, the results of Tesseract and the ground truth are always converted to lowercase. Hence, words like “GLASS” or “GLAss” will be recognized as correct when compared to “GLASS”.

The aim of this experiment is to show that the Tesseract classification does not always correctly recognize the characters. Moreover, we show that when the Tesseract Language Model does not form correct words, replacing some incorrectly recognized characters can improve its accuracy.

Tables 5.3 and 5.4 present the results obtained after doing so. The first seven databases use only Tesseract, while the next three use the Libccv library to perform Text Detection. We do not show the results of Text Detection since the substitution method does not modify the formation of the bounding boxes.

In some databases, the accuracy improved, although, in other databases, this was not the case. For example, in the IND database, accuracy improved only by 0.29% (Table 5.3), while in the ICDAR 2013 database, accuracy got worse by 1.18% (Table 5.4). Improvements obtained depend on the number of images in the database. If there are more images in the database, the greater the probability that Tesseract will fail and that the proposed method can correct them.

Although results improved in most of the databases with training data, the results can improve more. For example, in the IIIT5K database, there were 56 words that substitution method correctly corrected, increasing accuracy by 2.8%. Nevertheless, it improved only 0.83% of accuracy (Table 5.4). This happened because in addition to the

Table 5.3: Results of the substitution method considering the analysis of the next character.

Database	WRA (%) original	WRA (%) substitution	Improvement	Enhanced Words	Worsened Words
ICDAR 2003	50.11%	50.49%	0.38%	15	10
ICDAR 2013	55.66%	54.72%	-0.94%	12	20
IND	7.90%	8.19%	0.29%	25	12
IIIT5K	26.10%	27.70%	1.60%	48	16
SVT	15.95%	16.73%	0.78%	4	2
KAIST	47.20%	47.35%	0.15%	11	10
ICDAR 2015	59.22%	61.08%	1.86%	14	4
SVT	32.84%	28.36%	-4.48%	0	3
KAIST	60.05%	60.82%	0.77%	9	6

Table 5.4: Results of the substitution method considering the analysis of all the characters.

Database	WRA (%) original	WRA (%) substitution	Improvement	Enhanced Words	Worsened Words
ICDAR 2003	50.11%	49.81%	-0.30%	16	20
ICDAR 2013	55.66%	54.36%	-1.30%	14	25
IND	7.90%	8.19%	0.29%	28	15
IIIT5K	26.10%	27.60 %	1.50%	45	15
SVT	15.96%	16.73%	0.78%	4	2
KAIST	47.20%	47.04%	-0.16%	12	13
ICDAR 2015	59.22%	60.89%	1.67%	17	8
SVT	32.84%	26.87%	-5.97%	0	4
KAIST	60.05%	60.31%	0.26%	9	8

words correctly corrected, the spelling correction method corrects words that Tesseract correctly recognized, but they do not exist in the dictionary. This problem also happens in the databases where the results got worse. For example, in ICDAR 2013, accuracy got worse by 0.94% since the words that did not need to be corrected exceeded the words that were corrected. For this reason, in the following experiments we use restrictions to avoid spelling overcorrection.

Tables 5.5 and 5.6 present the results with the training data using only the substitutions method to correct the errors. The substitution method with the Tesseract restriction only corrects results that Tesseract considers that they are not words to avoid avercorrection.

Compared to the results of Tables 5.3 and 5.4, in most databases the WRA improved considerably. As observed in Tables 5.3 and 5.4, many results with the substitution method got worse than the original Tesseract results. Using the restrictions from Tesseract the results improved when comparing to the original, except for the SVT database with Libccv. For example, in ICDAR 2013, the accuracy improved from  $-1.18\%$  to

Table 5.5: Results of the substitution method considering the analysis of the next character with Tesseract restriction.

Database	WRA (%) original	WRA (%) substitution	Improvement	Enhanced Words	Worsened Words
ICDAR 2003	50.11%	50.87%	0.76%	14	4
ICDAR 2013	55.66%	55.78%	0.12%	12	11
IND	7.90%	8.26%	0.36%	25	9
IIT5K	26.10%	27.45%	1.35%	41	14
SVT	15.95%	16.73%	0.78%	4	2
KAIST	47.20%	47.66%	0.46%	9	6
ICDAR 2015	59.22%	60.89%	1.67%	12	3
SVT	32.84%	29.85%	-2.99%	0	2
KAIST	60.05%	61.34%	1.29%	8	3

Table 5.6: Results of the substitution method considering the analysis of all the characters with Tesseract restriction.

Database	WRA (%) original	WRA (%) substitution	Improvement	Enhanced Words	Worsened Words
ICDAR 2003	50.11%	50.64%	0.53%	9	2
ICDAR 2013	55.66%	56.60%	0.94%	10	2
IND	7.90%	8.34%	0.44%	26	6
IIT5K	26.10%	27.55 %	1.45%	31	2
SVT	15.96%	16.73%	0.78%	2	0
KAIST	47.20%	47.82%	0.62%	7	3
ICDAR 2015	59.22%	61.08%	1.86%	15	5
SVT	32.84%	28.36%	-4.48%	0	3
KAIST	60.05%	61.08%	1.03%	8	4

0.94%. This occurs since the number of words that the substitution method incorrectly corrected decreased. But it also witnessed a decrease in some databases. For example, in ICDAR 2015, without any restriction, the method obtained an improvement of 1.86%, while with the Tesseract restriction, an increase of 1.67% was measured. Since, the 14 words that were corrected without any restrictions, only 12 words were corrected, with the restriction.

Although results improved with Tesseract decision, there were still several words that Tesseract recognized well, but the spelling correction corrected it. For example, the IIT5TK database had 14 words that the spelling correction method should not correct (Table 5.5). These 14 words were equivalent to 0.45% of accuracy.

To address that issue we looked for another restriction, since the Tesseract variable that determines whether a result is correct or not, independent of whether the word is in the Tesseract dictionaries or not. This restriction could reduce the correct results that the proposed method should not correct without reducing the number of words that the method needs to correct.

Tables 5.7 and 5.8 present the results of using Tesseract with the substitution method, considering the certainty restriction. The certainty restriction considers that words that exceed  $-5.47$  of certainty should not be corrected. Results with this restriction improved in many databases.

Table 5.7: Results of the substitution method considering the analysis of the next character with certainty restriction.

Database	WRA (%) original	WRA (%) substitution	Improvement	Enhanced Words	Worsened Words
ICDAR 2003	50.11%	50.64%	0.53%	9	2
ICDAR 2013	55.66%	56.37%	0.71%	7	1
IND	7.90%	8.28%	0.38%	21	4
IIIT5K	26.10%	27.50%	1.40%	30	2
SVT	15.95%	16.73%	0.78%	2	0
KAIST	47.20%	47.98%	0.78%	7	2
ICDAR 2015	59.22%	60.52%	1.30%	8	1
SVT	32.84%	29.85%	-2.99%	0	2
KAIST	60.05%	60.82%	0.77%	3	0

Database	WRA (%) original	WRA (%) substitution	Improvement	Enhanced Words	Worsened Words
ICDAR 2003	50.11%	50.64%	0.53%	9	2
ICDAR 2013	55.66%	56.60%	0.94%	10	2
IND	7.90%	8.34%	0.44%	26	6
IIIT5K	26.10%	27.55 %	1.45%	31	2
SVT	15.96%	16.73%	0.78%	2	0
KAIST	47.20%	47.82%	0.62%	7	3
ICDAR 2015	59.22%	61.08%	1.86%	11	1
SVT	32.84%	29.85%	-2.99%	0	2
KAIST	60.05%	60.31%	0.26%	3	2

Table 5.8: Results of the substitution method considering the analysis of all the characters with certainty restriction.

From the Tables, we noticed that both, correctly corrected words and words that should not be corrected, decreased. Although both have decreased, the number of words that should not be corrected decreased significantly, improving the results of six out of nine databases. For example, results of ICDAR 2003 improved from 0.38% without any restriction, to 0.76% when considering the Tesseract restriction, and to 0.83% when considering the certainty restriction. But there are also the databases in which using the certainty restriction the accuracy decreased when compared to the Tesseract restriction. As explained above, this happens because the restriction also decreases the number of correctly corrected words. For example, the results of ICDAR 2015 improve by 1.86%

without any restriction, 1.67% when considering the Tesseract restriction and 1.30% when considering the certainty restriction.

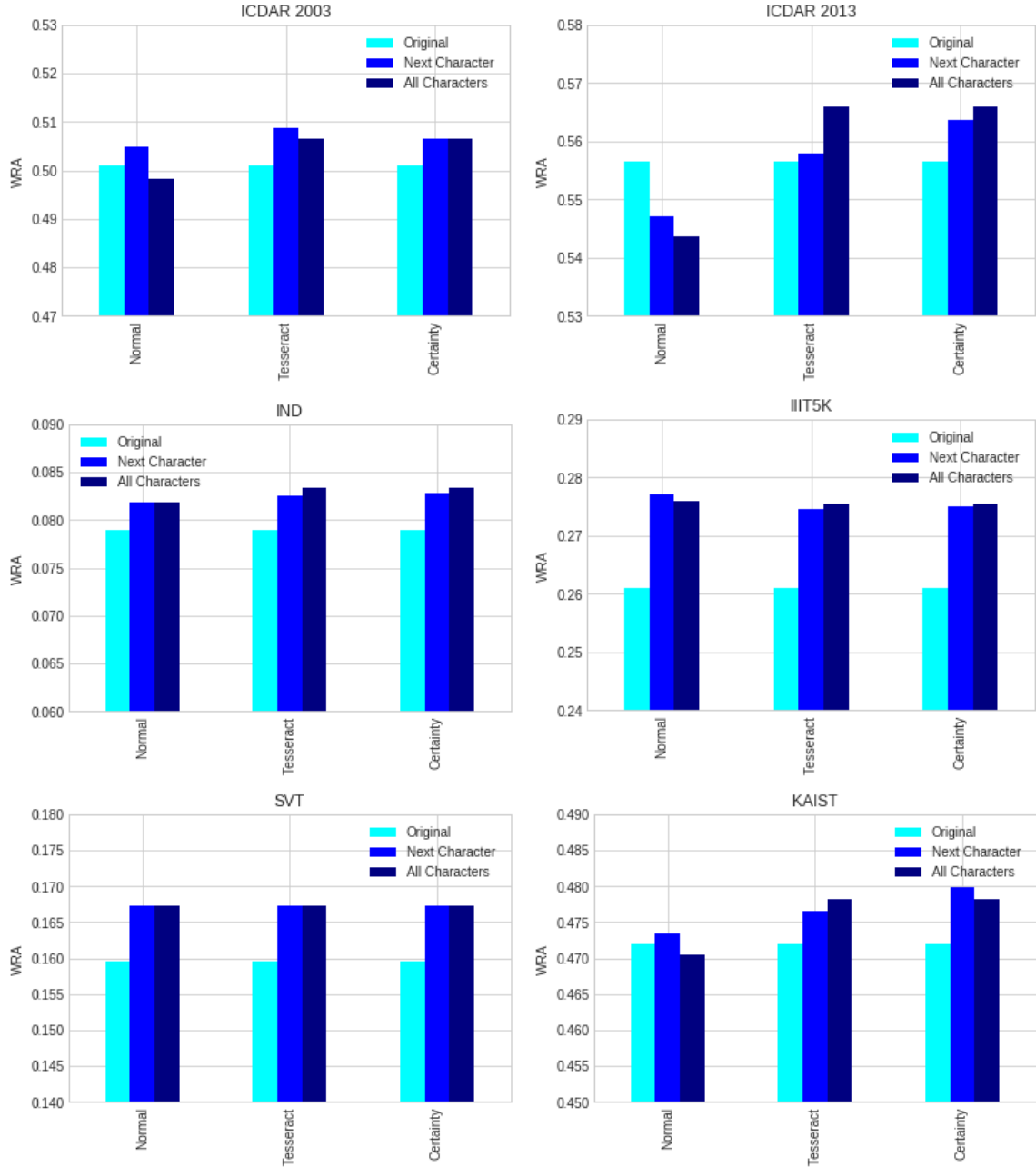


Figure 5.4: Results obtained Tesseract with the substitution method.

Figures 5.4 and 5.5 show a comparison between all the obtained results. We observe that in all databases, the proposed method with substitution improved the Tesseract results.

In Figure 5.4, the best results for ICDAR 2003, ICDAR 2013, IND and SVT (crop manually for Text Detection) databases are achieved when using the certainty restriction, since it considerably decreases the number of words that the proposed method should not

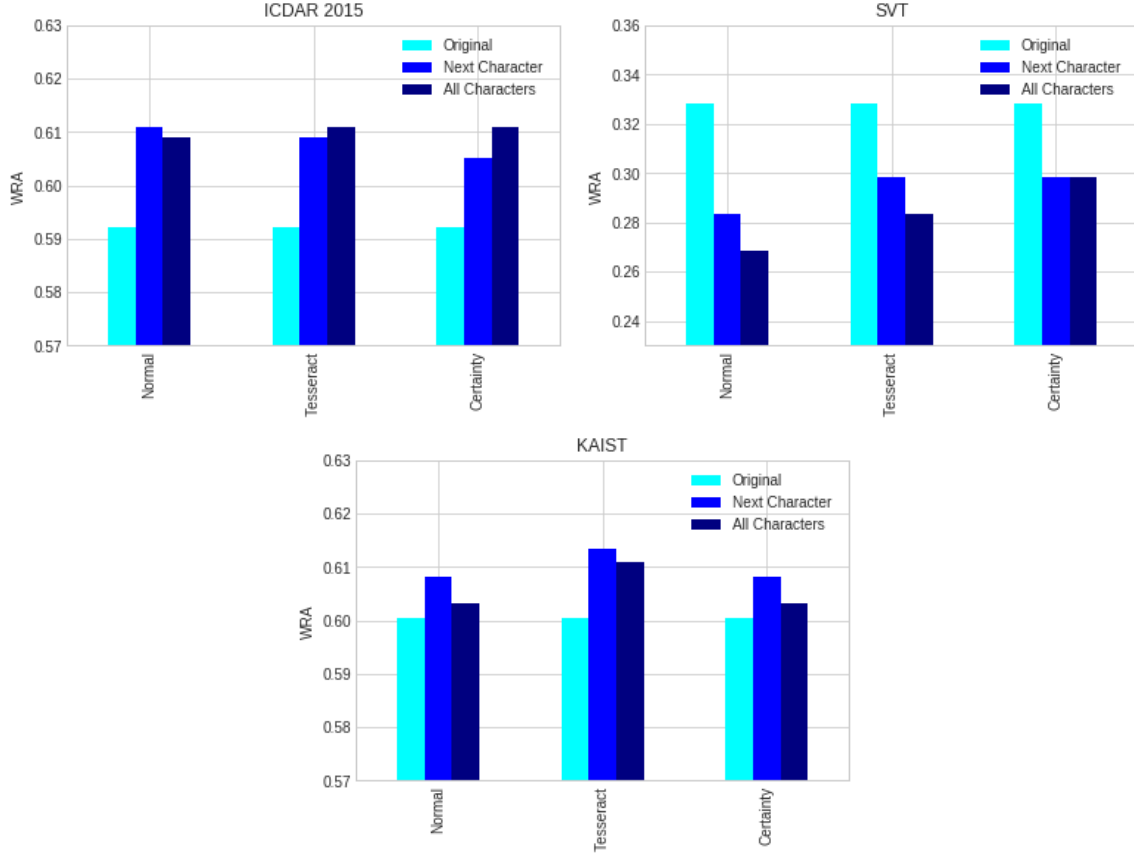


Figure 5.5: Results obtained Libccv and Tesseract with the substitution method.

correct. For the IIIT5K database, the best result is the substitution without any restriction. Due to the number of correctly corrected words decreased more than words with overcorrection with the Tesseract and certainty restrictions. For example, the proposed method without any restriction corrects forty-eight correct and sixteen incorrect words. With the Tesseract restriction, it improves forty-one and worsens fourteen, which means, the results decrease seven correct and only two incorrect words. For the SVT database, cropped manually for Text Detection, the results using any restriction or without using them do not vary, since the difference between the correct and incorrect word is the same in all of them.

In Figure 5.5, the best result for the ICDAR 2015 database is the proposed method without restriction, this case is the same explained with the IIIT5K database. For the KAIST database, the best result is using the method proposed with the Tesseract restriction. With this restriction, only words that the method should not correct decreased significantly, while correct words did not decrease. For the SVT database, results always worsen, since in this case there are no words that the substitution method can correct. However, there are words that it should not correct that neither the restrictions presented can prevent the method from correcting.

Figures 5.4 and 5.5 also show a comparison between the methods that were used to find the new character: analysis of the next character and analysis of all characters.

Table 5.9: Words corrected by using an analysis of the next character or all characters.

<b>Ground Truth</b>	<b>Original Results</b>	<b>Analysis of the next character</b>	<b>Analysis of all characters</b>
Manchester	Mancfiéiter	Mancfiéiter	Manchester
panasonic	panasom'c	panasom'c	panasonic
considering	consiantinl	consiantinl	considering
the	tfl	tfl	the
yellow	yellll	yellll	yellow
tesco	tesol	tesco	tesol
shafait	shafail	shafait	shaffer
steele	steetm	steele	steers
insead	insepd	insead	insert
indla	india	india	index

For the ICDAR 2013, IIIT5K, KAIST (cropped manually for Text Detection), ICDAR 2015, and KAIST databases, the best result is with the analysis of the next character. For example, in ICDAR 2013, the substitution method without any restriction with the analysis of the following character achieved a gain of 0.38% while using the other analyzes the results worsened by 0.60%, when compared to the original Tesseract results.

For the ICDAR 2013, IND, and SVT databases, the best result is obtained when using all character analysis. Although for some databases one analysis was better than the other, the difference between the use of one or the other is minimal. Since the majority of corrected words are the same they reached the same results. But there are words that only one of the methods could fix correctly. Table 5.9 shows some examples where only one analysis can correct the word.

Table 5.10: Words that cannot be corrected by the proposed substitutions method.

<b>Ground Truth</b>	<b>Tesseract recognition</b>
restaurant	fn'u'u'n'u'n'
wahlwiederh.	uflnruffliste
riverside	ri'oi-tered
superstar	nmmuvansnk
natwest	[iequeu

When we carried out experiments using the method of substitution with the analysis of all the characters, some words spent a lot of time to find the best result. We observed this problem more frequently if the method must replace the first character. This happens because, during this analysis, the algorithm must follow all paths to find the best word, unlike the analysis of the next character in which there are only one or two paths to follow. Images with this problem were removed to finish the experiments. Table 5.10 shows an example of some words that produce this problem. As observed, there are words that the distance is greater or equal to the ground truth word so they cannot be corrected even with the analysis of the following character.

Experiments using the substitution method show that there are many cases in which Tesseract does not correctly recognize characters that can be correctly replaced by other characters. But if the word does not exist in the dictionary, the substitution methods make overcorrection. The Tesseract and certainty restrictions can avoid it.

### 5.3.2 Insertion and Deletion

In this subsection, we present the results obtained using the insertion and deletion methods to correct the errors. The aim of this experiment in databases for Scene Text Recognition is to demonstrate if Tesseract recognizes all the characters, that is, if it finds all blobs. In databases for Scene Text Detection and Recognition, the aim of the experiment is to show how important Text Detection is for Tesseract to function properly.

We noticed that insertion only happens at the beginning or end of the word (Figure 5.6). Since there are problems with the image, such as lighting, Text Detection only detects, or Tesseract only recognizes a small part of the text. If Text Detection detects an initial and final part of the image, it will detect them as separate words. This problem is detailed in Section 5.3.3.

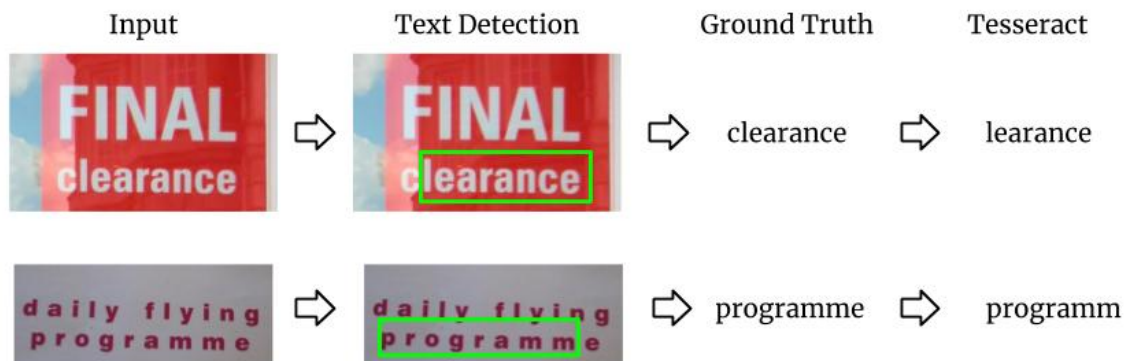


Figure 5.6: Problems where insertion can be used to correct errors.

We noticed that deletion only happens at the end of the word (Figure 5.7). The image has problems, such as noise. Text Detection detects symbols along with the word, and Tesseract recognizes these symbols as part of the word.

For these experiments, the insertion method uses only the analysis of the next character. Using the analysis with all characters produces infinite loops in errors that substitution method can correct, especially if the error is the first character. If the error is in the last character, there is no difference between using either method. As the deletion method does not seek a better character, it is not necessary to use some of the above analysis.

Table 5.11 and 5.12 present the results of using the insertion and deletion respectively in the spelling\_correction method.

Results of the insertion method (Table 5.11), show that all databases get worse since more words are incorrectly corrected, than correctly corrected words. For example, in the

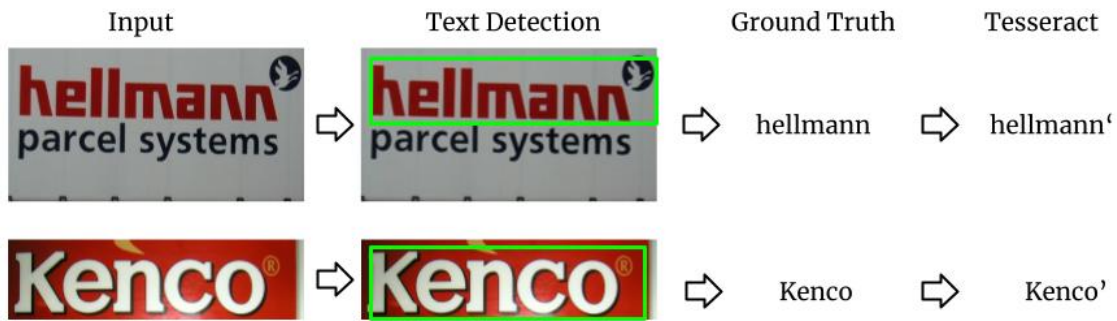


Figure 5.7: Problems where deletion can be used to correct errors.

Table 5.11: Results of the insertion method.

Database	WRA (%) original	WRA (%) insertion	Improvement	Enhanced Words	Worsened Words
ICDAR 2003	50.11%	49.81%	-0.30%	2	6
ICDAR 2013	55.66%	55.42%	-0.24%	3	5
IND	7.90%	7.63%	-0.27%	2	14
IIIT5K	26.10%	25.75 %	-0.35%	4	11
SVT	15.96%	15.96%	0.00%	0	0
KAIST	47.20%	46.42%	-0.78%	0	2
ICDAR 2015	59.22%	58.50%	-0.74%	5	13
SVT	32.84%	31.40%	-1.44%	0	4
KAIST	60.05%	59.27%	-0.79%	1	3

Table 5.12: Results of the deletion method.

Database	WRA (%) original	WRA (%) deletion	Improvement	Enhanced Words	Worsened Words
ICDAR 2003	50.11%	50.34%	0.23%	3	0
ICDAR 2013	55.66%	55.77%	0.11%	1	0
IND	7.90%	7.92%	0.02%	3	2
IIIT5K	26.10%	26.11 %	0.01%	5	3
SVT	15.96%	15.96%	0.00%	0	0
KAIST	47.20%	47.20%	0.00%	0	0
ICDAR 2015	59.22%	59.50%	0,28%	4	1
SVT	32.84%	32.84%	0.00%	0	0
KAIST	60.05%	60.05%	0.00%	0	0

IIIT5K database, the insertion method corrects four words correctly, but eleven words incorrectly. Results of the deletion method (Table 5.12) show that all databases improve minimally. For example, in the ICDAR 2013 database, there is only one word that is corrected.

The upper part of Table 5.13 presents some examples in which the insertion method corrected the results of Tesseract correctly, while the lower part presents examples in

which there was overcorrection with the insertion method. It can be seen that the problems of the insertion method occur in proper names and acronyms.

Table 5.13: Example with the insertion method.

<b>Original Results</b>	<b>Insertion</b>
futur	future
programm	programme
valv	valve
NAD	NADA
KENCO	KENTON
DE	DEC

The upper part of Table 5.14 presents some examples in which the deletion method corrected the results of Tesseract correctly, while the lower part presents examples in which there was overcorrection with the deletion method. Results that the deletion method improved are those that resulting from noise which led Tesseract recognize them as extra characters.

Table 5.14: Example with the deletion method.

<b>Original Results</b>	<b>Insertion</b>
chrisjian/ Checkmate)	christian Checkmate
Premium‘	Premium
Trespassing	Trespass
TUFFIN	TURF
Anfield	Annie

Results of Tables 5.11 and 5.12 cannot be improved using restricted substitution since Tesseract and certainty restrictions depend on the characters that exist in the word. It is not viable finding a restriction that attempts to decrease the number of words corrected erroneously without decreasing the number of words correctly corrected since both methods do not correct enough words like the substitution method.

Results of the insertion and deletion methods show that there are not many cases in which Scene Text Detection locate a word but not the first nor the last character. If there are a lot of errors in the Scene Text Detection step, it is because it does not find the word in the image or find a small piece of text that spelling\_correct method can not use to correct the word. In the Scene Text Recognition databases with the task, Tesseract does not always recognize all characters in the image, because texts in the image are not focused or there is noise of the image.

Based on the results, the insertion method will be discarded in the following experiments, and the deletion method will only be used if there are more characters at the end of the word.

### 5.3.3 Compounder

In this subsection, we present the results obtained using the compounder method to correct the errors. This experiment aims to verify if the method can join characters of the same word that Scene Text Detection found in different bounding boxes or if Tesseract recognized as different words, but it is the same word. Also, it also aims to demonstrate whether these errors occur frequently in databases.

The compounder method, as explained in subsection 4.1.4, is performed after Tesseract has finished recognizing and forming the word since the compounder method needs some data from both Text Detection and Tesseract. Also, the compounder method needs the `spelling_correction` method to be able to correct the words that it joins. In this case, the `spelling_correction` method includes the substitution and deletion methods to correct the errors.

Table 5.15 presents the final results obtained, in which we can observe an improvement in many of benchmarks while in others the results do not change. For example, in the ICDAR 2003 database, accuracy improved by 0.46%. While in the IIIT5K database, accuracy improved by 0.30%.

Table 5.15: Results of the compounder method.

Database	WRA (%) original	WRA (%) compounder	Improvement	Enhanced Words	Worsened Words
ICDAR 2003	50.11%	50.57%	0.46%	6	0
ICDAR 2013	55.66%	55.78%	0.12%	1	0
IND	7.90%	7.94%	0.04%	2	0
IIIT5K	26.10%	26.40%	0.30%	6	0
SVT	15.95%	15.95%	0.00%	0	0
KAIST	47.20%	47.66%	0.46%	3	0
ICDAR 2015	59.22%	60.15%	0.93%	5	0
SVT	32.84%	24.33%	1.49%	1	0
KAIST	60.05%	60.57%	0.52%	2	0

In all the studied databases, the compounder method does not correct any word that it should not, since the restrictions that must be met by both the characters to be joined and the formed word prevents this from happening. Cases where the formed word is in the dictionary, the bounding boxes are together or superimposed, among others. Table 5.16 shows some examples of the compounder method.

Table 5.16: Example of compounder method.

Results Original	Compounder
8401	950
Gat	ates
COMMONW	CON
EAALTH	DITIONED
COMMONWEALTH	CONDITIONED

The results of Table 5.15 are from the experiment that was carried out considering the substitution method without any restriction with the analysis of the following character. The experiments that were carried out considering restrictions and the analysis of all the characters for the substitution method, presented the same result as in Table 5.15.

Results of the compounder method show that it can successfully correct the errors that Scene Text Detection and Tesseract present when they find or recognize the same word in different bounding boxes or words, respectively. Based on the results, the compounder method will be used in the following experiments because this method only improves results without producing words that should not be corrected.

### 5.3.4 Best methods to correct errors

In this subsection, we present the results obtained by combining the best methods discussed so far, which are: (a) substitution; (b) deletion in the last character; and (c) compounder. This experiment aims to demonstrate if the combination of these methods improve the results of Tesseract than the methods applied separately.

Tables 5.17 and 5.18 present the results obtained with substitution, deletion in the last character, and compounder methods. The substitution method is implemented without any restrictions, with the analysis of the next character, and the analysis of all characters, respectively.

Table 5.17: Results of the spelling\_correction method considering the analysis of the next character.

Database	WRA (%) original	WRA (%) combined	Improvement	Enhanced Words	Worsened Words
ICDAR 2003	50.11%	50.94%	0.83%	21	10
ICDAR 2013	55.66%	54.83%	-0.83%	13	20
IND	7.90%	8.23%	0.33%	27	12
IIT5K	26.10%	28.00%	1.90%	54	16
SVT	15.95%	16.73%	0.78%	4	2
KAIST	47.20%	47.82%	0.62%	14	10
ICDAR 2015	59.22%	62.01%	2.79%	19	4
SVT	32.84%	29.85%	-2.99%	1	3
KAIST	60.05%	61.34%	1.29%	11	6

For example, in the IND database, only the substitution method improved the output by 0.29%, the compounder method alone improved it by 0.04%, and methods together generated an improvement of 0.33%. The only database that produced the same result with the substitution is the SVT.

Tables 5.19 and 5.20 present the results with the training data using substitution, deletion in the last character, and compounder methods. We implement the substitution method to correct only words that are not considered words by Tesseract. Table 5.19

Table 5.18: Results of the spelling\_correction method considering the analysis of all the characters.

Database	WRA (%) original	WRA (%) combined	Improvement	Enhanced Words	Worsened Words
ICDAR 2003	50.11%	50.26%	0.15%	22	20
ICDAR 2013	55.66%	54.48%	-1.18%	15	25
IND	7.90%	8.23%	0.33%	30	15
IIT5K	26.10%	27.90 %	1.80%	51	15
SVT	15.96%	16.73%	0.78%	4	2
KAIST	47.20%	47.51%	0.31%	15	13
ICDAR 2015	59.22%	61.82%	2.60%	22	8
SVT	32.84%	28.36%	-4.48%	1	4
KAIST	60.05%	60.57%	0.52%	10	8

presents the results of the substitution with the analysis of the next character and Table 5.20 presents the substitution with the analysis of all characters.

Table 5.19: Results of the spelling\_correction method considering the analysis of the next character with Tesseract restriction.

Database	WRA (%) original	WRA (%) combined	Improvement	Enhanced Words	Worsened Words
ICDAR 2003	50.11%	51.32%	1.21%	20	4
ICDAR 2013	55.66%	55.90%	0.24%	13	11
IND	7.90%	8.30%	0.40%	27	9
IIT5K	26.10%	27.75%	1.65%	47	14
SVT	15.95%	16.73%	0.78%	4	2
KAIST	47.20%	48.13%	0.93%	12	6
ICDAR 2015	59.22%	61.82%	2.60%	17	3
SVT	32.84%	31.34%	-1.50%	1	2
KAIST	60.05%	61.86%	1.81%	10	3

When comparing the results from Tables 5.3, 5.4, 5.17, and 5.18, in all databases, one can notice the WRA improved considerably. For example, in ICDAR 2015 databases, only the substitution method with the Tesseract restriction improves the result by 1.6%, the compounder method improves by 0.93%, the together methods without restrictions improve the results by 1.86%, and all together methods together with the Tesseract restriction improves the result by 2.60%. This means a 1.00% improvement when compared to only using the substitution method.

Similarly as in the usage of the substitution method in spelling\_correction, thanks to the Tesseract restriction, the number of words that the method should not correct decreased considerably, but also the number of correctly corrected words decreased. To improve that in the following we describe experiments that use the Tesseract certainty restriction.

Table 5.20: Results of the spelling\_correction method considering the analysis of all the characters with Tesseract restriction.

Database	WRA (%) original	WRA (%) combined	Improvement	Enhanced Words	Worsened Words
ICDAR 2003	50.11%	51.09%	0.98%	15	2
ICDAR 2013	55.66%	56.72%	1.06%	11	2
IND	7.90%	8.39%	0.49%	28	6
IIIT5K	26.10%	27.85 %	1.75%	37	2
SVT	15.96%	16.73%	0.78%	2	0
KAIST	47.20%	47.98%	0.78%	12	3
ICDAR 2015	59.22%	62.01%	2.79%	20	5
SVT	32.84%	29.85%	-2.99%	1	3
KAIST	60.05%	61.60%	1.55%	10	4

Tables 5.21 and 5.22 present the results using the substitution, deletion in the last character, and compounder methods to correct the errors, when considering the certainty restriction. Results with this restriction improved in many databases.

Table 5.21: Results of the spelling\_correction method considering the analysis of the next character with certainty restriction.

Database	WRA (%) original	WRA (%) combined	Improvement	Enhanced Words	Worsened Words
ICDAR 2003	50.11%	51.09%	0.98%	15	2
ICDAR 2013	55.66%	56.49%	0.83%	8	1
IND	7.90%	8.32%	0.42%	23	4
IIIT5K	26.10%	27.80%	1.70%	36	2
SVT	15.95%	16.73%	0.78%	2	0
KAIST	47.20%	48.44%	1.24%	10	2
ICDAR 2015	59.22%	61.45%	2.23%	13	1
SVT	32.84%	31.34%	-1.50%	1	2
KAIST	60.05%	61.34%	1.29%	5	0

Similar to the results obtained with the substitution method, some databases improved with the certainty restriction but others give better results with the Tesseract restriction. For example, KAIST (crop manually for Text Detection) database obtained an improvement of 1.24% with the certainty restriction, while ICDAR 2013 better result is 1.21% with the Tesseract restriction.

Figures 5.8 and 5.9 show a comparison between the original results of Tesseract (Original), substitution method with the analysis of the next character (S + Next Character), substitution method with the analysis of all characters (S + All Characters), the spelling\_correction method considering the substitution, deletion and compounder with the analysis of the next character (S + J + Next Character) and the spelling\_correction method considering the substitution, deletion and compounder with the analysis of all

Table 5.22: Results of the substitution method considering the analysis of all the characters with certainty restriction.

Database	WRA (%) original	WRA (%) combined	Improvement	Enhanced Words	Worsened Words
ICDAR 2003	50.11%	51.09%	0.98%	15	2
ICDAR 2013	55.66%	56.72%	1.06%	11	2
IND	7.90%	8.39%	0.49%	28	6
IIT5K	26.10%	27.85 %	1.75%	37	2
SVT	15.96%	16.73%	0.78%	2	0
KAIST	47.20%	48.29%	1.09%	3	0
ICDAR 2015	59.22%	62.01%	2.79%	16	1
SVT	32.84%	31.34%	-1.50%	1	2
KAIST	60.05%	60.82%	0.77%	5	2

the characters (S + J + All Character). We compare these methods when considering: no restrictions (Normal), Tesseract restriction (Tesseract), and certainty restriction (Certainty).

Results show that for some databases, the best results were with the Tesseract restriction and certainty restriction. All obtained a better result by combining the substitution and compounder methods than using the methods separately. Also, the compounder method attached to the substitution method does not increase the number of words that spelling\_correction should notcorrect, that is, only the correctly corrected words were increased, which improves the accuracy.

With Figures 5.8 and 5.9, we can conclude that each database there is a specific combination that leads to the best solution:

- The best result for the ICDAR 2003 database is obtained with the spelling\_correction method with substitution, deletion, and compounder methods. In this case the substitution method uses the analysis of the next character and with the Tesseract restriction (Tesseract - S + J + Next Character). The improvement is 1.21%.
- For the ICDAR 2013 database, the best result improves by 1.06%, by using Tesseract without certainty restrictions. The spelling\_correction method considers substitution, deletion, and compounder methods to correct errors, and substitution uses analysis of all characters. (Tesseract - S + J + All Character and Certainty - S + J + All Character).
- For the IND database, like in the ICDAR 2013 database case, Tesseract with certainty restrictions and spelling\_correction based on substitution, deletion, and compounder and analysis of all characters (Tesseract - S + J + All Character and Certainty - S + J + All Character) produce the best result. improvement of the best result is 0.49%.

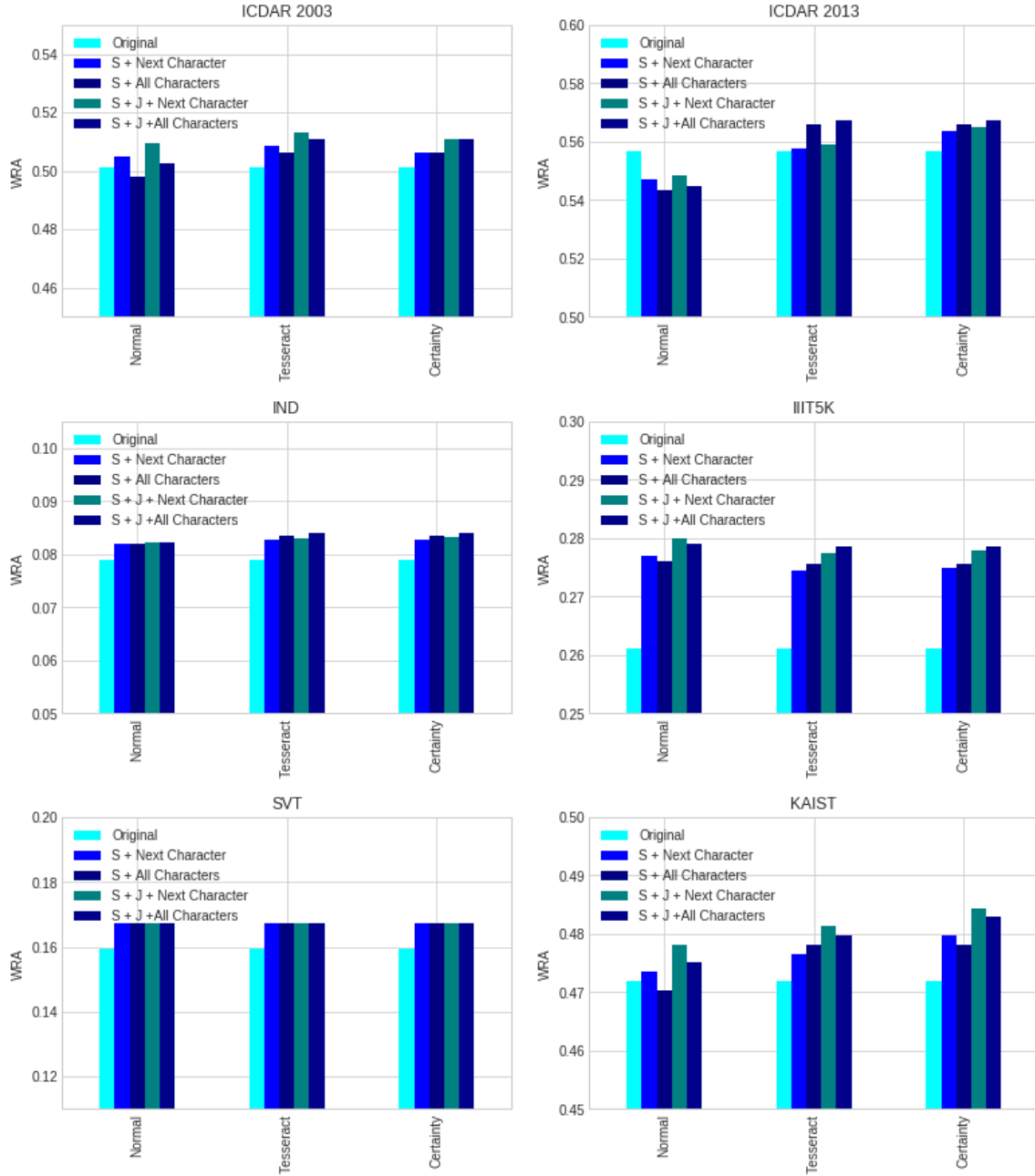


Figure 5.8: Results obtained Tesseract with the spelling\_correction method.

- For the IIIT5K database, the spelling\_correction method with substitution, deletion, and compounder methods, and analysis of the next character without any restriction, achieves the best result, an improvement of 1.9% (Normal - S + J + Next Character).
- For the SVT database where images were cut manually for Text Detection, all methods obtained the same improvement that is 0.78%.
- For the KAIST database where images were cut manually for Text Detection, the

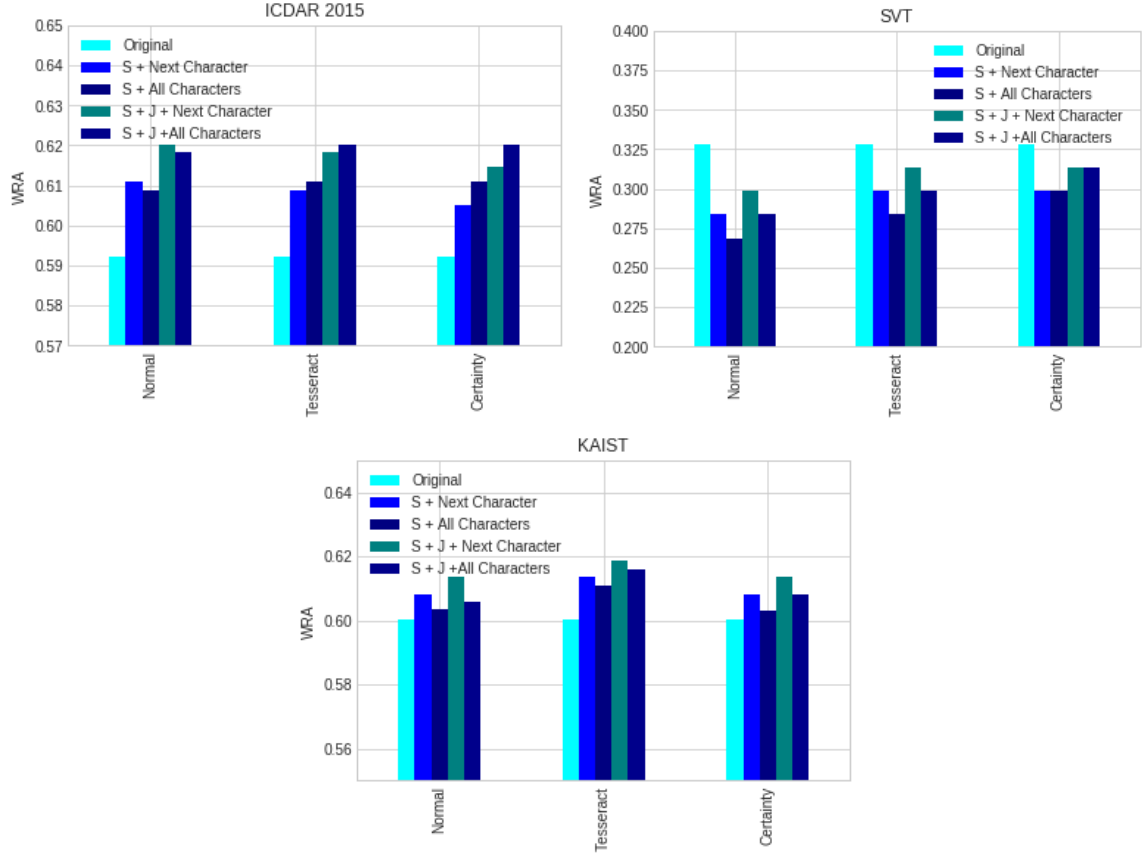


Figure 5.9: Results obtained Libccv and Tesseract with the spelling\_correction method.

best improvement is 2.23%. This improvement is obtained with certainty restriction with the analysis of the next character and the spelling\_correction method with the combination of the substitution, deletion and compounder methods to correct the errors (Certainty - S + J + Next Character).

- For the ICDAR 2015 database, there are three combinations where the improvement reaches 2.79%. (Normal - S + J + Next Character, Tesseract - S + J + All Character, and Certainty - S + J + All Character).
- For the SVT database, no combination managed to improve the original result. This is because there are only words that Tesseract recognized correctly, but given that they are not in the dictionary, the spelling\_correction method corrected them, making the result incorrect.
- For the KAIST database, the Tesseract restriction with the analysis of the next character and with the spelling\_correction considering substitution, deletion, and compounder achieved the best results, with an improvement of 1.81%.

The results showed that the best combination in general, considering all the databases, was the substitution, the deletion only in final characters and compounder, considering the Tesseract and certainty restrictions.

## 5.4 Frequency

In this section, we describe the results obtained when the representation of the dictionary is modified, and we use the frequency as a way to correct the errors described in Section 4.2. The same conditions as in the previous section were used in this section.

This experiment aims to demonstrate that using frequency as a way to choose the best WERD\_CHOICE is a better way than using the rating provided by Tesseract or a combination between them. Also, we want to demonstrate that using some bits that represent the edges formed by the DAWG to store information does not worsen the results.

Table 5.23 presents the results using the frequency method to choose the best among the WERD\_CHOICES of the WERD\_RES. That means the experiment uses the frequency as a way to choose the WERD\_CHOICE in all the words instead of the rating of Tesseract.

Table 5.23: Results of the frequency in Tesseract.

Database	WRA (%) original	WRA (%) frequency	Improvement	Enhanced Words	Worsened Words
ICDAR 2003	50.11%	51.77%	1.66%	33	11
ICDAR 2013	55.66%	56.25%	0.59%	19	14
IND	7.90%	8.48%	0.58%	53	27
IIIT5K	26.10%	27.55 %	1.45%	44	15
SVT	15.96%	17.51%	1.56%	5	1
KAIST	47.20%	47.51%	0.31%	13	11
ICDAR 2015	59.22%	60.34%	1.12%	12	6
SVT	32.84%	37.31%	4.47%	5	2
KAIST	60.05%	58.76%	-1.29%	6	11

Results in Table 5.23 show that in all databases, except KAIST, better results than the original were achieved. In some cases, results with the frequency produced a better accuracy than when using the spelling\_correction method. For example, in the IND database, with the frequency we achieved a accuracy of 8.48% which means an improvement of 0.58%. The best combination of the spelling correction method only obtained an improvement of 0.49%, showing that there was a improvement of 0.09% between the two methods.

The improvement resulting from using the frequency method is due to two reasons. The first reason is that the frequency method chooses the WERD\_CHOICE, that has the highest frequency among the possible words that Tesseract recognized for an image text. Table 5.24 shows some examples. The list in the column WERD\_CHOICE is words that Tesseract recognizes in the order in which it returns the words. The first WERD\_CHOICE is the result produced by Tesseract. The WERD\_CHOICE in red is the result given by original Tesseract, and the WERD\_CHOICE in green is the Tesseract result using the frequency method. The Rating column shows the value of the variable that Tesseract uses to decide which the best among the WERD\_CHOICE. The Frequency

column shows the frequency of the words recognized by Tesseract. In each group, the value that is in bold is the best.

Table 5.24: Example of frequency method.

Ground Truth	WERD_CHOICES	Rating	Frequency Tesseract
value	.value,	<b>68.21</b>	0.00
	.value	72.42	0.00
	value,	78.50	0.00
	value	82.78	<b>0.00013</b>
	valu5,	129.03	0.00
orange	ore/vga	<b>117.85</b>	0.00
	ora/vga	114.13	0.00
	orange	124.82	<b>0.000052</b>
	ora/ng:	133.46	0.00
up	uf	<b>28.89</b>	0.00
	up	29.85	<b>0.002393</b>
	vf	31.71	0.00000014
	nf	33.85	0.00000041
	hf	36.53	0.00000065

The second reason is that to use the frequency method, Tesseract needs a dictionary, in which all the characters are in a lower case. Hence, at the time of forming the word, Tesseract has more possibilities of not failing.

As in the substitution method, we can optimize the result if restrictions are used so that the frequency method does not modify the results that Tesseract correctly recognized.

Table 5.25 presents the results obtained when considering the Tesseract restriction and the frequency method to choose the best among the WERD\_CHOICES of the WERD\_RES. That means the experiment uses the frequency as a way to choose the WERD\_CHOICE in all the WERD\_RES that Tesseract considers as bad word.

Table 5.25: Results of the frequency method with Tesseract restriction.

Database	WRA (%) original	WRA (%) frequency	Improvement	Enhanced Words	Worsened Words
ICDAR 2003	50.11%	51.77%	1.66%	33	11
ICDAR 2013	55.66%	56.37%	0.71%	19	13
IND	7.90%	8.50%	0.60%	53	26
IIT5K	26.10%	27.50 %	1.40%	44	16
SVT	15.96%	17.51%	1.56%	5	1
KAIST	47.20%	47.35%	0.15%	13	12
ICDAR 2015	59.22%	60.15%	0.93%	12	7
SVT	32.84%	37.31%	4.47%	5	2
KAIST	60.05%	59.02%	-1.03%	6	10

Results in Table 5.25 show that for some databases, the result improves when compared to the results of the frequency method without restriction. But this improvement is a minimal amount when compared to the results that the restriction has with the substitution method. For example, the IND database had a gain of 0.02% compared to the frequency method without restriction, while the substitution method using the Tesseract restriction achieves an improvement of 0.25% when compared to the substitution method without any restriction. There are also databases in which the result worsens when compared to the results of the frequency method without restriction. For example, the IIIT5K database worsens 0.05% compared to the results of the frequency method without restriction.

Table 5.26 presents the results when considering the Tesseract restriction, for the frequency method to choose the best among the WERD\_CHOICES of the WERD\_RES. That means the experiment uses the frequency as a way to choose the WERD\_CHOICE in all the WERD\_RES that its best WERD\_CHOICE has a certainty less than  $-5.29$ .

Table 5.26: Results of the frequency method with certainty restriction.

Database	WRA (%) original	WRA (%) frequency	Improvement	Enhanced Words	Worsened Words
ICDAR 2003	50.11%	51.70%	1.59%	29	8
ICDAR 2013	55.66%	56.60%	0.94%	15	7
IND	7.90%	8.52%	0.62%	47	19
IIIT5K	26.10%	27.45 %	1.35%	33	6
SVT	15.96%	17.90%	1.95%	5	0
KAIST	47.20%	48.13%	0.93%	9	3
ICDAR 2015	59.22%	61.08%	1.86%	11	1
SVT	32.84%	37.31%	4.47%	4	1
KAIST	60.05%	61.34%	1.29%	6	1

Results in the Table 5.26 compare with the results of Table 5.25 show a great improvement in all databases like, although there are databases in which the result are worse , and in the SVT database, the result remains the same. For example, in the ICDAR 2015 database, the WERD\_RES that should not be changed is reduced to one word, thus achieving an improvement of 1.89%.

Figures 5.10 and 5.11 show a comparison between the original results of Tesseract (Original), the best result using the spelling\_correction method for every database (Best spelling\_correction), and the frequency method (Frequency). We compare them considering: without any restriction (Normal), Tesseract restriction (Tesseract), and certainty restriction (Certainty).

In the ICDAR 2003, IND, SVT (cut manually for Text Detection), and SVT databases, the results of the frequency method achieved better results than the better combinations of the spelling\_correction method. In the SVT database, the spelling\_correction method does not improve the results, but with the frequency method, we obtain an improvement. In the other databases, although the frequency method does not improve the results, it

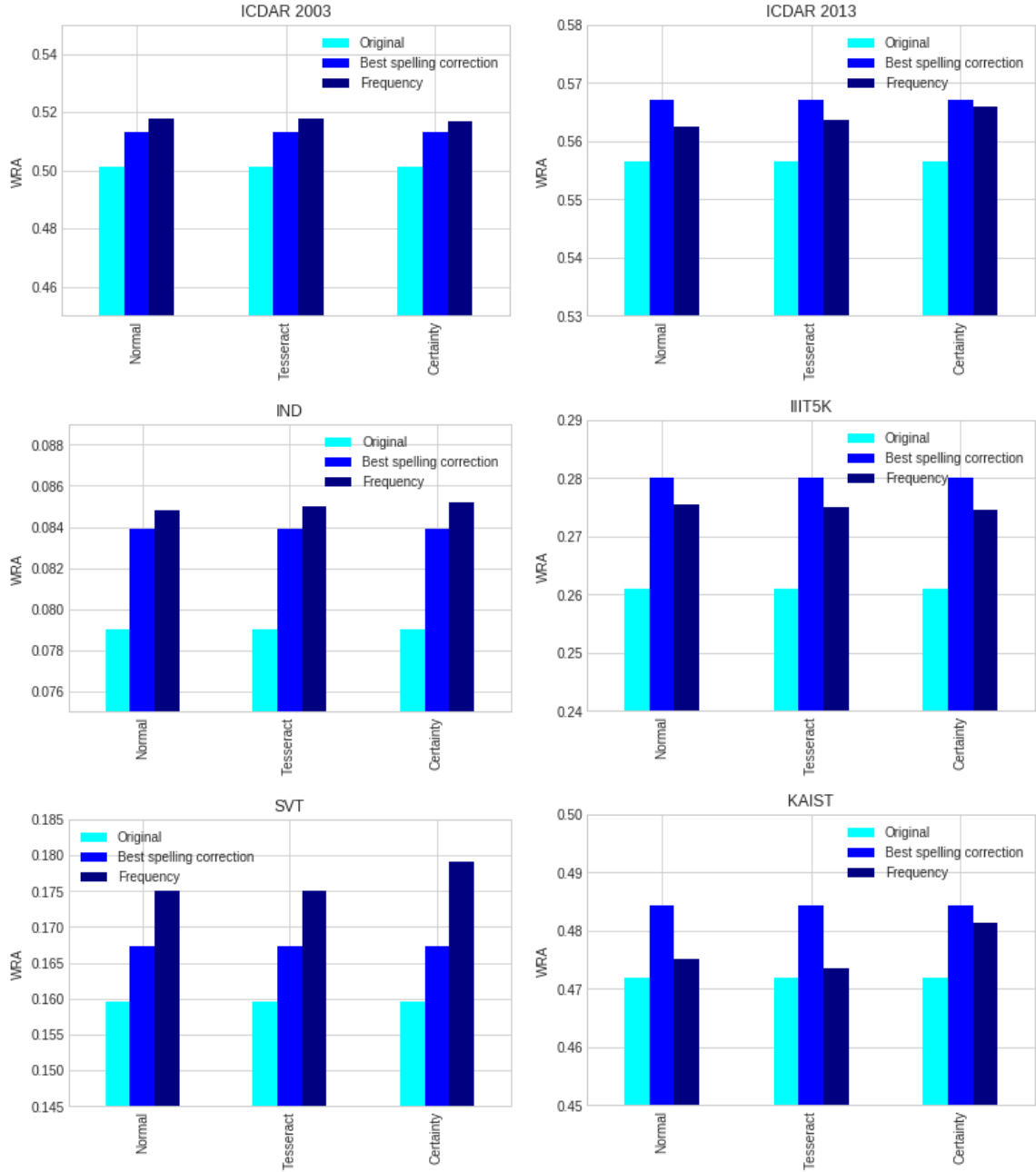


Figure 5.10: Results obtained Tesseract with the frequency method.

almost reaches the result of the spelling correction method. In the next section, we combined the results of the frequency and spelling correction method. With this combination, it is possible to increase the WRA of the databases.

Experiments show that the frequency method can replace the Tesseract rating, but using a combination between the Tesseract rating and the frequency method with the Tesseract and certainty restrictions, results improve considerably. Results also showed that the use of a dictionary with lowercase letters does not make the results worse.

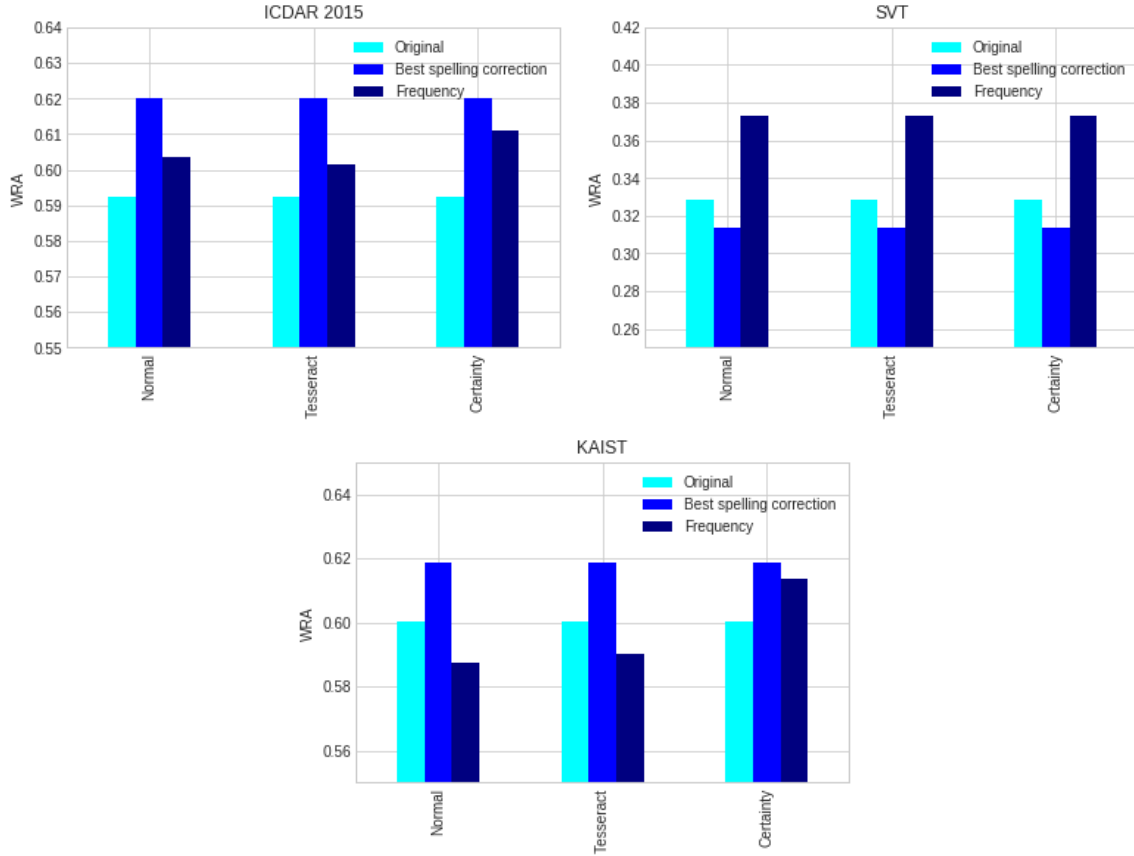


Figure 5.11: Results obtained Libccv and Tesseract with the frequency method.

## 5.5 Spelling correction and Frequency

In this section, we present the results obtained from combining the `spelling_correction` method and the frequency method. In addition, we present an analysis to determine which is the best combination to obtain the best results in training data and run the testing data with the best combination to have the final results for each database.

This experiment aims to demonstrate that the combination of the proposed methods can improve the results obtained by them separately. Also, a combination of them improves the original results of Tesseract using the testing data.

Based on the results described in the other sections, we carried out experiments considering the combination of the frequency method with the `spelling_correction` method using the substitution, deletion, and compounder methods to correct errors. As the experiments use the substitution method, we consider the two analyzes: analysis of the next character and analysis of all characters. In addition, we consider carrying out the experiments in the three cases: without restrictions, Tesseract restriction, and certainty restriction.

Tables 5.27 and 5.28 present the results of using the frequency method with the `spelling_correction` method without any restrictions. Table 5.27 shows the results with the analysis of the next character, and Table 5.28 presents the results with the analysis

of all characters.

Table 5.27: Results of the combination of spelling\_correction and frequency methods considering the analysis of the next character.

Database	WRA (%) original	WRA (%) combined	Improvement	Enhanced Words	Worsened Words
ICDAR 2003	50.11%	52.45%	2.34%	56	25
ICDAR 2013	55.66%	54.72%	-0.94%	31	39
IND	7.90%	8.57%	0.67%	71	41
IIT5K	26.10%	29.80%	3.70%	96	22
SVT	15.95%	17.51%	1.56%	6	2
KAIST	47.20%	47.66%	0.46%	28	25
ICDAR 2015	59.22%	62.94%	3.72%	30	10
SVT	32.84%	35.82%	2.98%	6	4
KAIST	60.05%	59.79%	-0.26%	20	21

Table 5.28: Results of the combination of spelling\_correction and frequency methods considering the analysis of all the characters.

Database	WRA (%) original	WRA (%) combined	Improvement	Enhanced Words	Worsened Words
ICDAR 2003	50.11%	51.24%	1.13%	53	38
ICDAR 2013	55.66%	53.42%	-2.24%	26	45
IND	7.90%	8.66%	0.79%	77	43
IIT5K	26.10%	29.30 %	3.20%	92	28
SVT	15.96%	17.12%	1.17%	6	3
KAIST	47.20%	47.04%	-0.16%	27	28
ICDAR 2015	59.22%	62.20%	2.98%	29	13
SVT	32.84%	35.82%	2.98%	6	4
KAIST	60.05%	59.02%	-1.03%	22	26

In the ICDAR 2003, IND, IIT5K, SVT (crop manually for Text Detection), and ICDAR 2015 databases, results improve compared to the frequency and spelling\_correction methods separately. In the KAIST (crop manually for Text Detection) and SVT databases, although the results are not better when compared to the methods separately, they are better when compared to the original Tesseract results. Finally, in the ICDAR 2013 and KAIST databases, the combined methods worsen the results, what does not occur with the methods separately. Making a comparison between the analysis of the next character and all characters depends on the database to decide which analysis is better. With all characters, the spelling\_correction can correct errors that the next character can not. But the problem with all characters is sometimes spent a lot of time trying to correct some words that spelling\_correction can not correct it like “*nmmuvansnk*” (ground truth “*superstar*”). While the next character finds quickly that spelling\_correction can not correct them

In these experiments, words that the substitution and frequency methods correctly correct them are considerably larger compared to the methods separately. For example, in the ITTT5K database, the method of substitution with the analysis of the following character corrects forty-eight words. The combination of the spelling\_correction and frequency methods with the analysis of the next character corrects ninety-six words. Words that the method should not correct go from sixteen words to twenty. As one can see, the corrected correctly words increase in relation to the words that get worse. Therefore, in many databases, the improvement increases considerably when compared to the frequency method or the spelling\_correction separately. Similar to the substitution or frequency method, results may improve even more if the proposed method is restricted to those words that Tesseract incorrectly recognized, considering the restrictions that are in the substitution method.

Tables 5.29 and 5.30 present the results of using the frequency method with the spelling\_correction method with the Tesseract restriction. Table 5.29 shows the results with the analysis of the next character, and Table 5.30 presents the results with the analysis of all characters.

Table 5.29: Results of the combination of spelling\_correction and frequency methods considering the analysis of the next character with Tesseract restriction.

Database	WRA (%) original	WRA (%) combined	Improvement	Enhanced Words	Worsened Words
ICDAR 2003	50.11%	52.52%	2.41%	55	23
ICDAR 2013	55.66%	54.83%	-0.83%	21	28
IND	7.90%	8.63%	0.73%	71	38
IIT5K	26.10%	29.60%	3.50%	90	20
SVT	15.95%	17.90%	1.95%	6	2
KAIST	47.20%	47.66%	0.46%	25	22
ICDAR 2015	59.22%	62.76%	3.54%	28	9
SVT	32.84%	35.82%	2.98%	6	4
KAIST	60.05%	60.57%	0.52%	19	17

In the ICDAR 2003, ICDAR 2013, IND, and KAIST databases, the results improve for both the following character analysis and all character analysis compared to the combined methods without any restrictions. Tesseract restriction reduces more the words that the method should not correct than words that it correctly correct. For example, in the IND databases, without any restriction with analysis of the next character, the proposed method corrects seventy-one but it makes overcorrection in forty-one incorrect word. While when the method uses the Tesseract restriction, it corrects seventy-one words and makes overcorrection in thirty-eighth. As we can observe in the previous example, the correct words remain the same while incorrect ones decrease by three.

In the SVT (crop manually for Text Detection) and SVT databases, results do not change in any of the analyses. This is because words that the combined method should not correct are minimal, that is, only three words.

Table 5.30: Results of the combination of spelling\_correction and frequency methods considering the analysis of all the characters with Tesseract restriction.

Database	WRA (%) original	WRA (%) combined	Improvement	Enhanced Words	Worsened Words
ICDAR 2003	50.11%	51.47%	1.36%	52	34
ICDAR 2013	55.66%	54.83%	-0.83%	26	33
IND	7.90%	8.72%	0.82%	77	40
IIIT5K	26.10%	29.15 %	3.05%	86	25
SVT	15.96%	17.12%	1.17%	6	3
KAIST	47.20%	47.20%	0.00%	24	24
ICDAR 2015	59.22%	62.57%	3.35%	19	2
SVT	32.84%	35.82%	2.98%	6	4
KAIST	60.05%	60.57%	0.52%	19	17

In the other IIIT5K, KAIST (crop manually for Text Detection), and ICDAR 2015 databases, results get worse with both analyzes compared to the results without any restrictions. Since words that the combined method must correct correctly decreased more than words that it should not correct. For example, the results of ICDAR 2015 without any restrictions were thirty-one correct and ten incorrect. Results with the Tesseract restriction are twenty-eight words correctly corrected and nine overcorrection. The proposed method with Tesseract restriction decreases three correct words and only one incorrect compare with dictionary restriction.

Tables 5.31 and 5.32 present the results of using the frequency method with the spelling\_correction method with the certainty restriction. Table 5.31 shows the results with the analysis of the next character, and Table 5.32 presents the results with the analysis of all characters.

Table 5.31: Results of the combination of spelling\_correction and frequency methods considering the analysis of the next character with certainty restriction.

Database	WRA (%) original	WRA (%) substitution	Improvement	Enhanced Words	Worsened Words
ICDAR 2003	50.11%	52.45%	2.34%	46	15
ICDAR 2013	55.66%	56.96%	1.30%	23	12
IND	7.90%	8.81%	0.91%	67	26
IIIT5K	26.10%	29.35%	3.25%	73	8
SVT	15.95%	17.90%	1.95%	5	0
KAIST	47.20%	49.53%	2.33%	20	5
ICDAR 2015	59.22%	63.50%	4.28%	25	2
SVT	32.84%	38.81%	5.97%	6	2
KAIST	60.05%	62.63%	2.58%	14	4

With the certainty restriction, the majority of the database obtains their best results, except ICDAR 2003 and IIIT5K databases, since words corrected correctly decrease con-

Table 5.32: Results of the combination of spelling\_correction and frequency methods considering the analysis of all the characters with certainty restriction.

Database	WRA (%) original	WRA (%) substitution	Improvement	Enhanced Words	Worsened Words
ICDAR 2003	50.11%	52.15%	2.04%	44	17
ICDAR 2013	55.66%	56.37%	0.71%	20	14
IND	7.90%	8.92%	1.02%	72	26
IIIT5K	26.10%	29.10 %	3.00%	70	10
SVT	15.96%	17.90%	1.95%	5	0
KAIST	47.20%	49.22%	2.02%	19	6
ICDAR 2015	59.22%	63.50%	4.28%	25	2
SVT	32.84%	38.81%	5.97%	6	2
KAIST	60.05%	62.89%	2.84%	16	5

siderably, while word with overcorrection not decreased.

Figures 5.12 and 5.13 show a comparison between the original results of Tesseract (Original), the best result using the spelling\_correction method for every database (Best spelling correction), the frequency method (Frequency), the combined method with the analysis of the next character (F + M\_S + Next Charater), and the combined method with the analysis of the all character (F + M\_S + All Charater). We compare considering: without any restriction (Normal), Tesseract restriction (Tesseract), and certainty restriction (Certainty).

Figures 5.12 and 5.13 show that the combination method obtains the best results. But it depends on each database, what restriction or analysis is the best.

- In the ICDAR 2003 database, the best result is with the method combined with the Tesseract restriction with the analysis of the next character. The final result is 52.52%, which means an increase of 2.41%, 32 words of 1327.
- In the ICDAR 2013 database, the best result is with the method combined with the certainty restriction with the analysis of the next character. The final result is 56.96%, which means an increase of 1.30%, 11 words of 848.
- In the IND database, the best result is with the method combined with the certainty restriction with the analysis of the all characters. The final result is 8.92%, which means an increase of 1.02%, 46 words of 4494.
- In the IIIT5K database, the best result is with the method combined without the restriction with the analysis of the next character. The final result is 29.80%, which means an increase of 3.70%, 74 words of 2000.
- In the SVT database, crop manually for Text Detection, the best result is with the method combined with the certainty restriction with the analysis of the all characters. The final result is 18.29%, which means an increase of 2.34%, 6 words of 257.

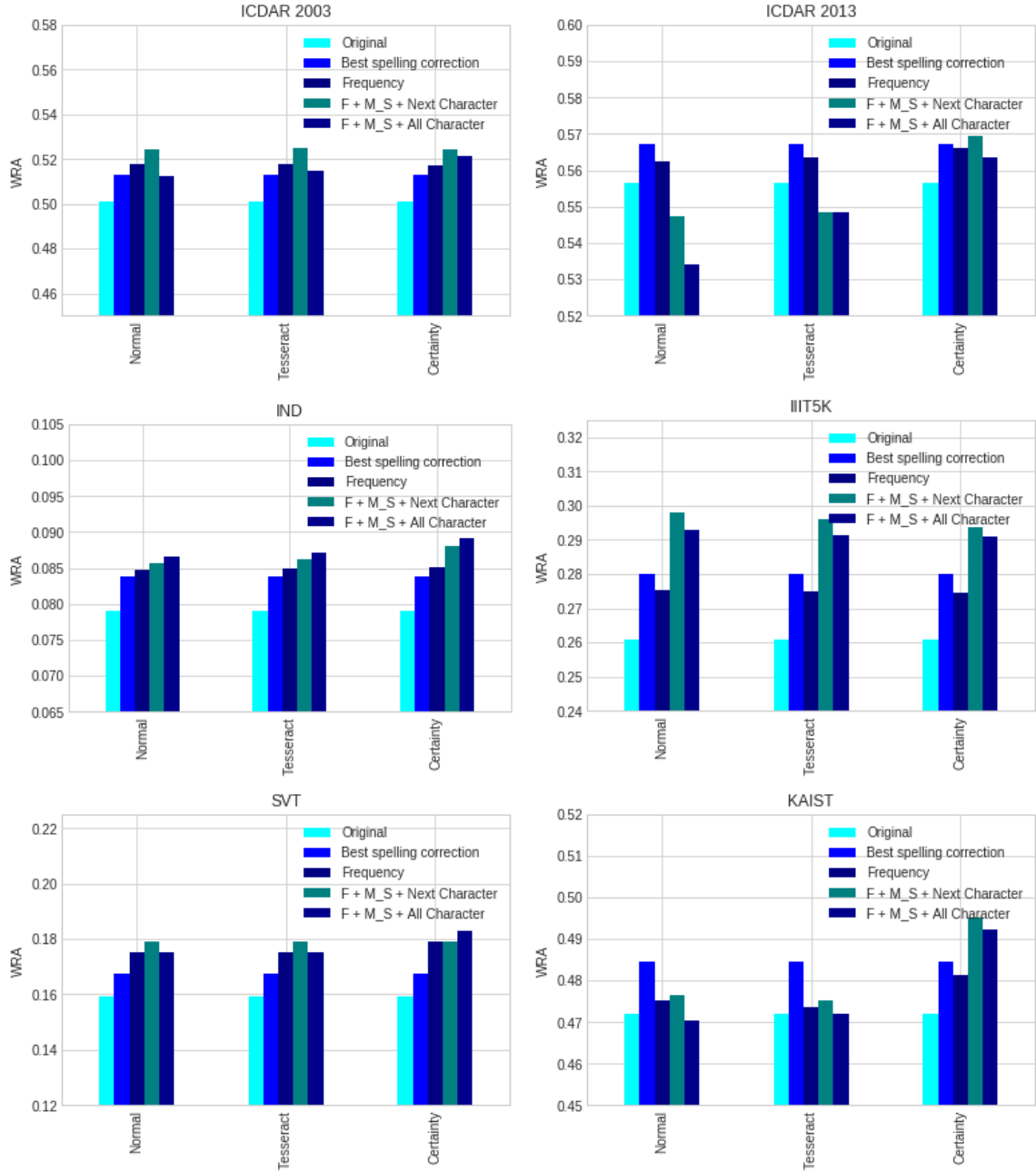


Figure 5.12: Results obtained Tesseract with the combined method.

- In the KAIST database, crop manually for Text Detection, the best result is with the method combined with the certainty restriction with the analysis of the next character. The final result is 49.53%, which means an increase of 2.33%, 15 words of 642.
- In the ICDAR 2015 database, the best result is with the method combined with the certainty restriction with the analysis of the next character and with the method combined with the certainty restriction with the analysis of all characters. The final result is 63.50%, which means an increase of 4.28%, 23 words of 537.

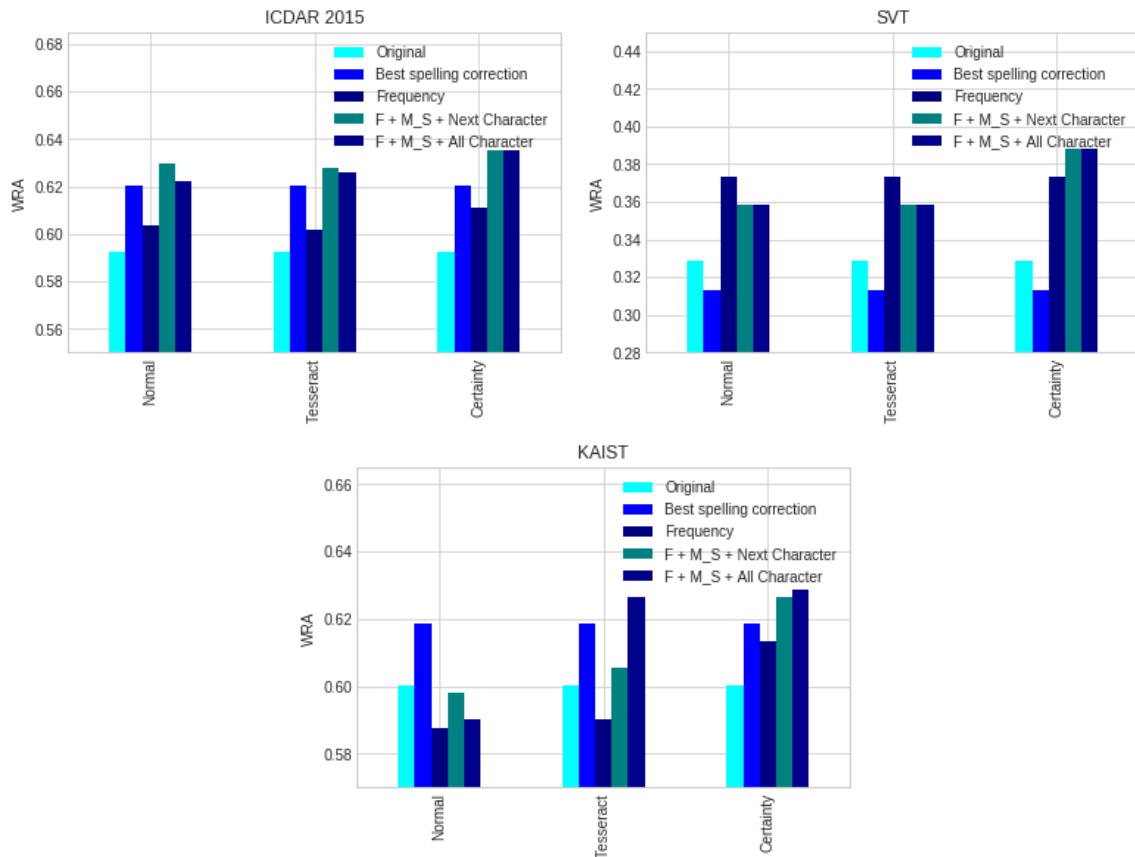


Figure 5.13: Results obtained Libccv and Tesseract with the combined method.

- In the SVT database, the best result is with the method combined with the certainty restriction with the analysis of the next character and with the method combined with the certainty restriction with the analysis of all characters. The final result is 38.81%, which means an increase of 5.97%, 18 words of 67.
- In the KAIST database, the best result is with the method combined with the certainty restriction with the analysis of all characters. The final result is 62.89%, which means an increase of 2.84%, 11 words of 388.

The best method is a combination of the spelling\_correction and frequency methods considering the substitution, deletion, and compounder methods to correct errors. The best restriction is with the certainty restriction. For the best analysis, there is a tie between the analysis of the next character and all characters. Therefore, to obtain the results with the testing data, we use the combined methods, with the certainty restriction, and the two analyzes for the substitution method.

Tables 5.33 and 5.34 present the results of the testing data with the best combination obtained with the training data. Table 5.33 presents results with the analysis of the next character and Table 5.34 with the analysis of all characters.

Tables 5.33 and 5.34 show that the method combined with the certainty restriction improved the original results of the Tesseract in all databases.

Table 5.33: Results with Testing Data considering the analysis of the next character.

Database	WRA (%) original	WRA (%) combined	Improvement	Enhanced Words	Worsened Words
ICDAR 2003	30.06%	30.81%	0.75%	10	6
ICDAR 2013	43.84%	46.48%	2.64%	39	10
IND	7.05%	8.06%	1.01%	30	9
IIIT5K	19.57%	22.20%	2.63%	96	17
SVT	21.48%	23.34%	1.86%	18	6
KAIST	42.14%	44.85%	2.71%	19	5
ICDAR 2015	62.46%	66.19%	3.73%	23	2
SVT	46.86%	49.71%	2.85%	6	1
KAIST	61.22%	63.78%	2.56%	6	1

Table 5.34: Results with Testing Data considering the analysis of all the characters.

Database	WRA (%) original	WRA (%) combined	Improvement	Enhanced Words	Worsened Words
ICDAR 2003	30.06%	30.81%	0.75%	11	7
ICDAR 2013	43.84%	46.21%	2.37%	36	10
IND	7.05%	8.06%	1.01%	30	9
IIIT5K	19.57%	22.20 %	2.63%	96	17
SVT	21.48%	22.72%	1.24%	15	7
KAIST	42.14%	44.66%	2.52%	19	6
ICDAR 2015	62.46%	66.19%	3.73%	23	2
SVT	46.89%	49.14%	2.28%	6	2
KAIST	61.22%	63.78%	2.05%	5	1

## Chapter 6

# Conclusions and Future Works

This work described and evaluated post-correction methods to improve the results of Tesseract in scene images. We performed an analysis of the results obtained from Tesseract to determine which post-correction methods were the most appropriate for Tesseract, allowing to obtain a better accuracy of the most common errors in Tesseract results.

Experiments carried out in different databases showed that post-correction methods in the final results of Tesseract improved its accuracy. Showing that pos-correction methods can be successfully implemented in Scene Text Recognition methods, where the results are not as expected.

We conclude that the best post-correction method for Tesseract is the combination of spelling correction techniques, frequency, and Tesseract restrictions. Among the spelling correction methods, the best combination is the substitution, which corrected the classification problems, the deletion in the last characters that corrected the noise in images, and the compounder that corrects Text Detection errors and errors produced by the space between characters. The word frequency proved to be an efficient way to choose the best result for the image among a group of candidates when all candidates are present in the dictionary.

Different restrictions were applied to the correction of errors in the results to avoid overcorrection. The restrictions were obtained by analyzing the training data as obtained from Tesseract. Restrictions improved results as it reduced the words that should not be corrected without decreasing the correctly corrected words.

With the restrictions implemented in techniques that correct the errors, the results showed that the post-correction methods can be used in results of scenes images where there is no more information than the word and the information provided by the method that recognizes the text.

As future work, we intend to extend the post-correction method presented with other text recognition methods, which use modern techniques to recognize the text. For example, neural networks. We could start by performing the experiments on the most current versions of Tesseract, where LSMT is used to recognize the text.

# Bibliography

- [1] Fan Bai, Zhazhan Cheng, Yi Niu, Shiliang Pu, and Shuigeng Zhou. Edit probability for scene text recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1508–1516, 2018.
- [2] Youssef Bassil and Mohammad Alwani. Ocr post-processing error correction algorithm using google online spelling suggestion. *arXiv preprint arXiv:1204.0191*, 2012.
- [3] Richard Beaufort and Céline Mancas-Thillou. A weighted finite-state framework for correcting errors in natural scene ocr. In *Ninth International Conference on Document Analysis and Recognition (ICDAR 2007)*, volume 2, pages 889–893. IEEE, 2007.
- [4] M. Bokser. Omnidocument technologies. *Proceedings of the IEEE*, 80(7):1066–1078, 1992.
- [5] Arpita Chakraborty and Michael Blumenstein. Marginal noise reduction in historical handwritten documents—a survey. In *2016 12th IAPR Workshop on Document Analysis Systems (DAS)*, pages 323–328. IEEE, 2016.
- [6] Datong Chen and Jean-Marc Odobez. Video text recognition using sequential monte carlo and error voting methods. *Pattern Recognition Letters*, 26(9):1386–1403, 2005.
- [7] Zhazhan Cheng, Fan Bai, Yunlu Xu, Gang Zheng, Shiliang Pu, and Shuigeng Zhou. Focusing attention: Towards accurate text recognition in natural images. In *Proceedings of the IEEE international conference on computer vision*, pages 5076–5084, 2017.
- [8] Guillaume Chiron, Antoine Doucet, Mickaël Coustaty, and Jean-Philippe Moreux. Icdar2017 competition on post-ocr text correction. In *2017 14th IAPR International Conference on Document Analysis and Recognition (ICDAR)*, volume 1, pages 1423–1428. IEEE, 2017.
- [9] Guillaume Chiron, Antoine Doucet, Mickaël Coustaty, Muriel Visani, and Jean-Philippe Moreux. Impact of ocr errors on the use of digital libraries: towards a better access to information. In *2017 ACM/IEEE Joint Conference on Digital Libraries (JCDL)*, pages 1–4. IEEE, 2017.

- [10] Chee Kheng Ch'ng and Chee Seng Chan. Total-text: A comprehensive dataset for scene text detection and recognition. In *2017 14th IAPR International Conference on Document Analysis and Recognition (ICDAR)*, volume 1, pages 935–942. IEEE, 2017.
- [11] Hojin Cho, Myungchul Sung, and Bongjin Jun. Canny text detector: Fast and robust scene text localization algorithm. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3566–3573, 2016.
- [12] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the twentieth annual symposium on Computational geometry*, pages 253–262, 2004.
- [13] Mark Davies. The 385+ million word corpus of contemporary american english (1990–2008+): Design, architecture, and linguistic insights. *International journal of corpus linguistics*, 14(2):159–190, 2009.
- [14] Boris Epshtein, Eyal Ofek, and Yonatan Wexler. Detecting text in natural scenes with stroke width transform. In *Computer Vision and Pattern Recognition (CVPR), 2010 IEEE Conference on*, pages 2963–2970. IEEE, 2010.
- [15] John Evershed and Kent Fitch. Correcting noisy ocr: Context beats confusion. In *Proceedings of the First International Conference on Digital Access to Textual Cultural Heritage*, pages 45–51, 2014.
- [16] Jacqueline L Feild and Erik G Learned-Miller. Improving open-vocabulary scene text recognition. In *2013 12th International Conference on Document Analysis and Recognition*, pages 604–608. IEEE, 2013.
- [17] G David Forney. The viterbi algorithm. *Proceedings of the IEEE*, 61(3):268–278, 1973.
- [18] Edward Fredkin. Trie memory. *Communications of the ACM*, 3(9):490–499, 1960.
- [19] Mehrdad J Gangeh, Sunil R Tiyyagura, Sridhar V Dasaratha, Hamid Motahari, and Nigel P Duffy. Document enhancement system using auto-encoders. 2019.
- [20] Lluís Gomez and Dimosthenis Karatzas. Multi-script text extraction from natural scenes. In *Proc. ICDAR*, pages 467–471, 2013.
- [21] Isabelle Guyon and Fernando Pereira. Design of a linguistic postprocessor using variable memory length markov models. In *Proceedings of 3rd International Conference on Document Analysis and Recognition*, volume 1, pages 454–457. IEEE, 1995.
- [22] Harald Hammarström, Shafqat Mumtaz Virk, and Markus Forsberg. Poor man's ocr post-correction: Unsupervised recognition of variant spelling applied to a multi-lingual document collection. In *Proceedings of the 2nd International Conference on Digital Access to Textual Cultural Heritage*, pages 71–75, 2017.

- [23] Axel Jean-Caurant, Nouredine Tamani, Vincent Courboulay, and Jean-Christophe Burie. Lexicographical-based order for post-ocr correction of named entities. In *2017 14th IAPR International Conference on Document Analysis and Recognition (ICDAR)*, volume 1, pages 1192–1197. IEEE, 2017.
- [24] Longquan Jiang, Bo Zhang, Qin Ni, Xuan Sun, and Pingping Dong. Prediction of snp sequences via gini impurity based gradient boosting method. *IEEE Access*, 7:12647–12657, 2019.
- [25] Dimosthenis Karatzas, Lluís Gomez-Bigorda, Angelos Nicolaou, Suman Ghosh, Andrew Bagdanov, Masakazu Iwamura, Jiri Matas, Lukas Neumann, Vijay Ramaseshan Chandrasekhar, Shijian Lu, et al. Icdar 2015 competition on robust reading. In *2015 13th International Conference on Document Analysis and Recognition (ICDAR)*, pages 1156–1160. IEEE, 2015.
- [26] Dimosthenis Karatzas, Faisal Shafait, Seiichi Uchida, Masakazu Iwamura, Lluís Gomez i Bigorda, Sergi Robles Mestre, Joan Mas, David Fernandez Mota, Jon Almazan Almazan, and Lluís Pere De Las Heras. Icdar 2013 robust reading competition. In *2013 12th International Conference on Document Analysis and Recognition*, pages 1484–1493. IEEE, 2013.
- [27] James M Keller, Michael R Gray, and James A Givens. A fuzzy k-nearest neighbor algorithm. *IEEE transactions on systems, man, and cybernetics*, (4):580–585, 1985.
- [28] Dar-Shyang Lee and Ray Smith. Improving book ocr by adaptive language and image models. In *2012 10th IAPR International Workshop on Document Analysis Systems*, pages 115–119. IEEE, 2012.
- [29] Jung-Jin Lee, Pyoung-Hean Lee, Seong-Whan Lee, Alan Yuille, and Christof Koch. Adaboost for text detection in natural scene. In *2011 International Conference on Document Analysis and Recognition*, pages 429–434. IEEE, 2011.
- [30] Vladimir I Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, volume 10, pages 707–710, 1966.
- [31] Lon-Mu Liu, Yair M Babad, Wei Sun, and Ki-Kan Chan. Adaptive post-processing of ocr text via knowledge acquisition. In *Proceedings of the 19th annual conference on Computer Science*, pages 558–569, 1991.
- [32] Simon M Lucas, Alex Panaretos, Luis Sosa, Anthony Tang, Shirley Wong, and Robert Young. Icdar 2003 robust reading competitions. In *Seventh International Conference on Document Analysis and Recognition, 2003. Proceedings.*, pages 682–687. Citeseer, 2003.
- [33] Shashank Mujumdar, Nitin Gupta, Abhinav Jain, and Douglas Burdick. Simultaneous optimisation of image quality improvement and text content extraction from

- scanned documents. In *2019 International Conference on Document Analysis and Recognition (ICDAR)*, pages 1169–1174. IEEE, 2019.
- [34] Lukáš Neumann. Scene text localization and recognition in images and videos. 2017.
  - [35] Lukas Neumann and Jiri Matas. Text localization in real-world images using efficiently pruned exhaustive search. In *Document Analysis and Recognition (ICDAR), 2011 International Conference on*, pages 687–691. IEEE, 2011.
  - [36] Lukáš Neumann and Jiří Matas. Real-time scene text localization and recognition. In *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on*, pages 3538–3545. IEEE, 2012.
  - [37] Hisao Niwa, Kazuhiro Kayashima, and Yasuharu Shimeki. Postprocessing for character recognition using keyword information. In *MVA*, pages 519–522, 1992.
  - [38] Tatiana Novikova, Olga Barinova, Pushmeet Kohli, and Victor Lempitsky. Large-lexicon attribute-consistent text recognition in natural images. In *European conference on computer vision*, pages 752–765. Springer, 2012.
  - [39] Remus Petrescu, Sergiu Manolache, Costin-Anton Boiangiu, Giorgiana Violeta Vlăsceanu, Cristian Avatavului, Marcel Prodan, and Ion Bucur. Combining tesseract and asprise results to improve ocr text detection accuracy. *Journal of Information Systems & Operations Management*, pages 57–64, 2019.
  - [40] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 779–788, 2016.
  - [41] Christophe Rigaud, Antoine Doucet, Mickaël Coustaty, and Jean-Philippe Moreux. Icdar 2019 competition on post-ocr text correction. 2019.
  - [42] Sarah Schulz and Jonas Kuhn. Multi-modular domain-tailored ocr post-correction. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 2716–2726, 2017.
  - [43] Faisal Shafait and Ray Smith. Table detection in heterogeneous documents. In *Proceedings of the 9th IAPR International Workshop on Document Analysis Systems*, pages 65–72. ACM, 2010.
  - [44] Asif Shahab, Faisal Shafait, and Andreas Dengel. Icdar 2011 robust reading competition challenge 2: Reading text in scene images. In *2011 international conference on document analysis and recognition*, pages 1491–1496. IEEE, 2011.
  - [45] Nabin Sharma, Ranju Mandal, Rabi Sharma, Partha P Roy, Umapada Pal, and Michael Blumenstein. Multi-lingual text recognition from video frames. In *2015*

*13th International Conference on Document Analysis and Recognition (ICDAR)*, pages 951–955. IEEE, 2015.

- [46] Baoguang Shi, Mingkun Yang, Xinggang Wang, Pengyuan Lyu, Cong Yao, and Xiang Bai. Aster: An attentional scene text recognizer with flexible rectification. *IEEE transactions on pattern analysis and machine intelligence*, 41(9):2035–2048, 2018.
- [47] CS Shin, KI Kim, MH Park, and Hang Joon Kim. Support vector machine-based text detection in digital video. In *Neural Networks for Signal Processing X. Proceedings of the 2000 IEEE Signal Processing Society Workshop (Cat. No. 00TH8501)*, volume 2, pages 634–641. IEEE, 2000.
- [48] Ray Smith. An overview of the tesseract ocr engine. In *Ninth International Conference on Document Analysis and Recognition (ICDAR 2007)*, volume 2, pages 629–633. IEEE, 2007.
- [49] Ray Smith. Limits on the application of frequency-based language models to ocr. In *2011 International Conference on Document Analysis and Recognition*, pages 538–542. IEEE, 2011.
- [50] Ray Smith, Daria Antonova, and Dar-Shyang Lee. Adapting the tesseract open source ocr engine for multilingual ocr. In *Proceedings of the International Workshop on Multilingual OCR*, page 1. ACM, 2009.
- [51] Ray W Smith. History of the tesseract ocr engine: what worked and what didn’t. In *Document Recognition and Retrieval XX*, volume 8658, page 865802. International Society for Optics and Photonics, 2013.
- [52] Christian M Strohmaier, Christoph Ringlstetter, Klaus U Schulz, and Stoyan Mihov. Lexical postcorrection of ocr-results: The web as a dynamic secondary dictionary? In *Seventh International Conference on Document Analysis and Recognition, 2003. Proceedings.*, pages 1133–1137. Citeseer, 2003.
- [53] Kazem Taghva and Eric Stofsky. Ocrspell: an interactive spelling correction system for ocr errors in text. *International Journal on Document Analysis and Recognition*, 3(3):125–137, 2001.
- [54] Ranjith Unnikrishnan and Ray Smith. Combined script and page orientation estimation using the tesseract ocr engine. In *Proceedings of the International Workshop on Multilingual OCR*, page 6. ACM, 2009.
- [55] Kai Wang and Serge Belongie. Word spotting in the wild. In *European Conference on Computer Vision*, pages 591–604. Springer, 2010.

- [56] Liang-Kai Wang, Charles Tsen, Michael J Schulte, and Divya Jhalani. Benchmarks and performance analysis of decimal floating-point applications. In *2007 25th International Conference on Computer Design*, pages 164–170. IEEE, 2007.
- [57] Qixiang Ye and David Doermann. Text detection and recognition in imagery: A survey. *IEEE transactions on pattern analysis and machine intelligence*, 37(7):1480–1500, 2015.
- [58] Xinyu Zhou, Cong Yao, He Wen, Yuzhi Wang, Shuchang Zhou, Weiran He, and Jiajun Liang. East: an efficient and accurate scene text detector. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, pages 5551–5560, 2017.
- [59] Yingying Zhu, Cong Yao, and Xiang Bai. Scene text detection and recognition: Recent advances and future trends. *Frontiers of Computer Science*, 10(1):19–36, 2016.