1
Case Study – Book Cover Classification, Analysis, and Recognition

In this chapter, we will work on classifying an image cover into one of 30 categories. We will start by finding a publicly available book cover dataset, continue with analyzing the dataset, and then work on building a custom classifier using a combination of features extracted from a pretrained neural network and custom architecture. By the end of this chapter, you will also see how you can expose your solution to a REST web service.

The following topics will be covered in this chapter:

- Getting a dataset from a GitHub repository
- Analyzing the dataset
- Using pretrained neural networks, such as MobileNet V2 and Inception V3
- Building a custom multi-class classifier using MXNet
- Running text recognition (OCR) using Tesseract
- Evaluating the results

Technical requirements

Users are required to have the following setup:

- Julia 0.6.3 or higher
- The `Images.jl` package installed
- The `MXNet.jl` package installed
- The `JLD.jl` package installed
- Tesseract

In case of any issues, please refer to `Chapter 1`, *Getting Started with JuliaImages*, or `Chapter 5`, *Image Representation*, for troubleshooting. We will cover the installation procedure for Tesseract separately in this chapter.

Data preparation

Let's assume we have a book cover we would like to classify and find the book title of. How would you proceed in this case?

The process of preparing the dataset consists of two parts:

- Getting the images
- Getting the categories

Getting the images

There are two options to retrieve the dataset:

1. Download images directly from Amazon Marketplace
2. Download a ZIP archive containing both `train` and `test` datasets prepared using the links from step 1

Please be aware that the full dataset size is around 2 GB. You should ensure that you have sufficient disk space.

Downloading images directly from Amazon Marketplace

It is possible to download all of the required files manually from Amazon Marketplace. For this, you need to navigate to the `book dataset` repository on GitHub (`https://github.com/uchidalab/book-dataset/tree/master/Task1`) and use the listing files for the `train` and `test` datasets.

Please be aware that the next subsection provides access to a ZIP archive containing all of those files. It is highly recommended to proceed with the ZIP file instead of manually downloading each and every image. This solution can be used as a backup or as a reference.

You can run a simple Julia script to download the files. This script will iterate over data files placed in a root folder and download every single file in a CSV file and save it under the correct name, which can be used later so that you can map it with the category value:

```
PROJECT_IMG_FOLDER = joinpath("data", "book-covers")
DATASETS = ["test", "train"]

if ~isdir(PROJECT_IMG_FOLDER) mkdir(PROJECT_IMG_FOLDER) end

for dataset_name in DATASETS

    lines = nothing

    current_folder = joinpath(PROJECT_IMG_FOLDER, dataset_name)
    if ~isdir(current_folder) mkdir(current_folder) end

    file_name = joinpath(pwd(), "book30-listing-$dataset_name.csv")
    f = open(file_name); lines = readlines(f); close(f);
    for line in lines
        line_split = split(line, "\",\"")
        download(line_split[3], joinpath(PROJECT_IMG_FOLDER,
        dataset_name, line_split[2]))
    end
end
```

The download process can take time. There are over 50,000 images in total. You should have the `train` and `test` folders in the `data/book-covers` directory containing the images. We did not use any special package to process the CSV files and relied on simply reading all of the lines, splitting the columns using `","` as a separator.

Using a pre-downloaded ZIP archive

The other option would be to use another GitHub repository where the author has taken care of downloading all of those images and putting them into two ZIP archives—one for training the dataset and another for testing.

Please navigate to `https://github.com/leekyungmoon/Judging-a-Book-by-its-Cover` and find links to `train_images` and `test_images`. Both files are located on Google Drive and are safe to download.

Getting the categories

When you are done downloading the images, it is also important to get the file which includes categories corresponding to each image. They are stored in the `label` files in GitHub at `https://github.com/uchidalab/book-dataset/tree/master/Task1`.

These files consist of two columns—one corresponding to the filename and the other to the category.

Analyzing the dataset

When the dataset has been downloaded, it is important to put it in a `data/book-covers` folder:

- The `train` and `test` datasets should be put into the `train` and `test` folders accordingly. For example, the `train` dataset is located at `data/book-covers/train` and the `test` dataset is available at `data/book-covers/test`.
- `Label` files should be put in `data/book-covers` so that we can reference them when needed.

When the files are placed, we can proceed with verification and ensure that this is alright.

Verifying the images

Let's start by verifying the images that are available and can be loaded by Julia. We will also check what the resolutions of the images that have been selected are:

```
using Images, ImageView

PROJECT_IMG_FOLDER = joinpath("data", "book-covers")
```

```
DATASETS = ["test", "train"]

preview_count = 5
shape = (100, 78)
preview_img = zeros(typeof(RGB4{N0f8}(0.,0.,0.)), (100, 1));

original_shapes = []

for dataset_name in DATASETS

    current_folder = joinpath(PROJECT_IMG_FOLDER, dataset_name)
    files = filter(x -> contains(x, ".jpg"), readdir(current_folder))

    shuffle!(files)

    for i=1:preview_count
        img_original = load(joinpath(current_folder, files[i]))
        img_resized = imresize(img_original, shape)
        preview_img = hcat(preview_img, img_resized)
        push!(original_shapes, size(img_original))
    end
end

println(original_shapes)
imshow(preview_img)
```

The code will first output shapes for the images that are being processed. As you can see, the shapes are fixed in height but are different in width:

```
(Any[(500, 381), (500, 333), (500, 324), (500, 360), (500, 339)]...)
```

Finally, you will see a preview of the images. The books are very different—from the **National Geographic** to **French Grammar**:

Now that we have our first understanding of the images dataset, we are ready to have a quick look at categories.

Analyzing categories

If you look at the author's GitHub repository containing listing and `labels` files (`https://github.com/uchidalab/book-dataset/tree/master/Task1`), you will see that there should be a total of 30 classes with 1,710 and 190 images per class in the `train` and `test` datasets accordingly:

Label	Category Name	Training size	Test size
0	Arts and Photography	1,710	190
1	Biographies and Memoirs	1,710	190
2	Business and Money	1,1710	190
...	...	...	...

We can run a simple script to ensure that the numbers match. We will iterate over the test and training `label` files and count the number of occurrences for each category:

```
using StatsBase

PROJECT_IMG_FOLDER = joinpath("data", "book-covers")
DATASETS = ["test", "train"]

if ~isdir(PROJECT_IMG_FOLDER) mkdir(PROJECT_IMG_FOLDER) end

for dataset_name in DATASETS

    categories = Int[]

    current_folder = joinpath(PROJECT_IMG_FOLDER, dataset_name)
    if ~isdir(current_folder) mkdir(current_folder) end

    file_name = joinpath(PROJECT_IMG_FOLDER, "bookcover30-
    labels-$dataset_name.txt")
    f = open(file_name); lines = readlines(f); close(f);
    for line in lines
        value = parse(Int, split(line, " ")[2])
        push!(categories, value)
    end

    println(countmap(categories))
end
```

From the following result, we can see that everything is OK and that we can proceed. There are `190` examples in the `test` dataset and `1710` in the `train` dataset:

```
Dict(18=>190,2=>190,16=>190,11=>190,21=>190,0=>190,7=>190,9=>190,10=>190,25
=>190,26=>190,29=>190,19=>190,17=>190,8=>190,22=>190,6=>190,24=>190,4=>190,
3=>190,28=>190,5=>190,20=>190,23=>190,13=>190,14=>190,27=>190,15=>190,12=>1
90,1=>190)

Dict(18=>1710,2=>1710,16=>1710,11=>1710,21=>1710,0=>1710,7=>1710,9=>1710,10
=>1710,25=>1710,26=>1710,29=>1710,19=>1710,17=>1710,8=>1710,22=>1710,6=>171
0,24=>1710,4=>1710,3=>1710,28=>1710,5=>1710,20=>1710,23=>1710,13=>1710,14=>
1710,27=>1710,15=>1710,12=>1710,1=>1710)
```

Classification using MobileNet V2

In our first example, we will be using MobileNet V2 to extract image features and create another neural network that will act as a multiclass classifier.

I expect that you will want to have the `MXNet` and `MXNet.jl` packages up and running before you proceed.

Getting the model

In order to proceed, you will need to download the MXNet MobileNet V2 model weights. Please navigate to the KeyKy Mobile Net repository (`https://github.com/KeyKy/mobilenet-mxnet`) and download the following two files:

- `mobilenet_v2-0000.params`
- `mobilenet_v2-symbol.json`

Place both files into the `weights/mobilenet-v2` folder.

Loading the network

We will start by loading the `MXNet` package and MobileNet V2 model and continue by removing the last `SoftmaxOutput` and `FullyConnected` layers.

The following code replicates the *Extracting features generated by MobileNet V2* section from `Chapter 7`, *Using Pre-Trained Neural Networks*. Please refer to it for more details:

```
using Images, MXNet

const MODEL_NAME = "weights/mobilenet-v2/mobilenet_v2"
const MODEL_CLASS_NAMES = "weights/mobilenet-v2/synset.txt"

nnet = mx.load_checkpoint(MODEL_NAME, 0, mx.FeedForward; context =
mx.gpu());
synset = readlines(MODEL_CLASS_NAMES);

layers = mx.get_internals(nnet.arch);
layers_flatten = nothing
layers_to_remove = Symbol[]

# We iterate over all layers until we find the one matching our
requirements
# and remove the ones to follow after
for i = 1:2000

    layer = layers[i];
    layer_name = mx.get_name(layer)
    if layers_flatten == nothing && layer_name == :pool6
        layers_flatten = layer
    elseif layers_flatten != nothing
        push!(layers_to_remove, layer_name)
        if layer_name in [:softmax, :label, :prob] break end
    end
end

nnet.arch = @mx.chain layers_flatten => Flatten()
map(x -> delete!(nnet.arg_params, x), layers_to_remove);
map(x -> delete!(nnet.aux_params, x), layers_to_remove);

mx.save_checkpoint(nnet, "weights/mobilenet-v2/MobiletNet-FE",
mx.OptimizationState(1, 0, 0, 0))
```

We also save the modified version of a model to disk so that we can reuse it in the future.

Extracting features

Our model is ready and we can proceed with the feature extraction process. The following processes will take place:

- Initialize all required packages
- Load the MobileNet V2 feature extractor model
- Configure the base variables, such as a folder with images and batch size
- Iterate through the training and testing datasets by loading the image names from `label` files
- Preprocess every single image and make a batch prediction
- Cache the end result to a `.jld` file

Consider the following code:

```
using Images, MXNet, JLD

nnet = mx.load_checkpoint("weights/mobilenet-v2/MobiletNet-FE", 0,
mx.FeedForward; context = mx.gpu());

PROJECT_IMG_FOLDER = joinpath("data", "book-covers");
DATASETS = ["train", "test"];
BATCH_SIZE = 100;

for dataset_name in DATASETS

    categories = Int[]

    current_folder = joinpath(PROJECT_IMG_FOLDER, dataset_name)
    file_name = joinpath(PROJECT_IMG_FOLDER, "bookcover30-
    labels-$dataset_name.txt")
    f = open(file_name); lines = readlines(f); close(f);
    results = zeros(Float16, (1280, size(lines, 1)))

    for i=1:BATCH_SIZE:size(lines, 1)
        println("BATCH: $i")

        mx_data = mx.zeros((224, 224, 3, BATCH_SIZE));

        for j=1:BATCH_SIZE

            if j % 50 == 0 println("\tIMG: $j") end

            img = imresize(load(joinpath(current_folder, split(lines[i
            + j - 1])[1])), (224, 224));
```

```
            img = permutedims(Float16.(channelview(RGB.(img))), (3, 2,
            1));

            img[:, :, :] *= 256.0;
            img[:, :, :] -= 128.;
            img[:, :, :] /= 128.;

            img = reshape(img, 224, 224, 3, 1);
            mx_data[j:j] = img;
        end

        data_provider = mx.ArrayDataProvider(:data => mx_data);
        results[:, i:i+(BATCH_SIZE-1)] = mx.predict(nnet,
        data_provider)
    end

    JLD.save(joinpath(PROJECT_IMG_FOLDER, "dump-$dataset_name.jld"),
    "data", results)
end
```

In case you don't have enough computational power to run the feature extraction yourself, I have prepared a ZIP archive containing both the `train` and `test.jld` files. Extract the two files into the `data/book-covers` folder.

Here is the download link:

`https://drive.google.com/file/d/1v28ses_oNI3lT8R7hMJ_HGfXZVtT8e5r/view?usp=sharing`

Building the classifier

Let's move on to building a custom classifier. We will start by reloading the `train` and `test` datasets from a cache and extract the categories from the `labels` file:

```
using Images, MXNet, JLD

function get_labels_from_file(path)
  return map(x -> parse(Int8, split(x)[2]), readlines(path))
end

PROJECT_IMG_FOLDER = joinpath("data", "book-covers");
DUMP_IMG_TRAIN = joinpath(PROJECT_IMG_FOLDER, "dump-train.jld")
DUMP_IMG_TEST = joinpath(PROJECT_IMG_FOLDER, "dump-test.jld")
LABELS_TRAIN = joinpath(PROJECT_IMG_FOLDER, "bookcover30-labels-train.txt")
LABELS_TEST = joinpath(PROJECT_IMG_FOLDER, "bookcover30-labels-test.txt")
```

```
# get the data
train_img_data = JLD.load(DUMP_IMG_TRAIN, "data");
test_img_data = JLD.load(DUMP_IMG_TEST, "data");

train_labels = get_labels_from_file(LABELS_TRAIN);
test_labels = get_labels_from_file(LABELS_TEST);
```

As we are not 100% sure of the dataset and we need it to be random when training a new network, we will shuffle it manually:

```
shuffle_train_indices = shuffle(1:size(train_labels, 1))
shuffle_test_indices = shuffle(1:size(test_labels, 1))

train_img_data = train_img_data[:, shuffle_train_indices];
train_labels = train_labels[shuffle_train_indices];

test_img_data = test_img_data[:, shuffle_test_indices];
test_labels = test_labels[shuffle_test_indices];
```

We will keep the `test` dataset as a holdout set and split the `train` dataset into training and validation of `5000` in size, and then we can test the network quality during the training process. As per MXNet requirements, we will convert combined image features and labels in a single `ArrayDataProvider`:

```
valid_record_count = 5000
train_data_provider = mx.ArrayDataProvider(:data => train_img_data[:,
1:end-valid_record_count], :label => train_labels[1:end-
valid_record_count], shuffle = true, batch_size = 500);
valid_data_provider = mx.ArrayDataProvider(:data => train_img_data[:, end-
valid_record_count+1:end], :label => train_labels[end-
valid_record_count+1:end], shuffle = true, batch_size = 500);
```

Now, we are left with simply defining the network structure and running `mx.fit`:

```
arch = @mx.chain mx.Variable(:data) =>
  mx.FullyConnected(num_hidden=30) =>
  mx.SoftmaxOutput(mx.Variable(:label))

nnet = mx.FeedForward(arch, context = mx.cpu())
mx.fit(nnet, mx.ADAM(), train_data_provider, eval_data =
valid_data_provider, n_epoch = 15, eval_metric =
mx.MultiMetric([mx.Accuracy(), mx.ACE()]));
```

If you follow the training process, you will see that it reached 26% accuracy on a validation set. Not bad, considering that we had 30 classes in total. A random choice would give us 3% accuracy.

Testing the model

Now, let's get to testing the model. For this, we will need to run the predict function on a `test` dataset, select the most probable value, and compare it to the original value:

```
test_data_provider = mx.ArrayDataProvider(:data => test_img_data,
batch_size = 500);
predictions = mx.predict(nnet, test_data_provider)
results = map(x -> findmax(predictions[:, x])[2], 1:size(test_labels, 1))

println(round(mean((results - 1) .== test_labels), 2))
```

I got 26% when running the model on a `test` dataset. It matches the number from the validation process during the model's training.

Book cover classification using Inception V3

Inception V3 is more powerful than MobileNet V2, so we should expect better results. Let's follow the same procedure we used earlier and get the test score.

Getting the model

To start working with Inception V3, you first need to download the model files from the MXNet Model Zoo. Use the following links:

- URL: `http://data.dmlc.ml/models/imagenet/`
- ZIP: `inception-v3.tar.gz`

After you have downloaded the archive, please extract it to the `weights/inception-v3` folder.

Loading the network

We will start by loading the `MXNet` package and Inception V3 model and continue with removing the last `SoftmaxOutput` and `FullyConnected` layers.

The following code replicates the *Extracting features generated by Inception V3* section from `Chapter 7`, *Using Pre-Trained Neural Networks*. Please refer to it for more details on implementation:

```
using Images, MXNet

const MODEL_NAME = "weights/inception-v3/Inception-7"
nnet = mx.load_checkpoint(MODEL_NAME, 1, mx.FeedForward);

layers = mx.get_internals(nnet.arch);
layer_names = Symbol[]

layers_flatten = nothing
layers_to_remove = Symbol[]

for i = 1:2000

    layer = layers[i];
    layer_name = mx.get_name(layer)
    if layers_flatten == nothing && layer_name == :flatten
        layers_flatten = layer
    elseif layers_flatten != nothing
        push!(layers_to_remove, layer_name)
        if layer_name in [:softmax, :label] break end
    end
end

nnet.arch = layers_flatten
map(x -> delete!(nnet.arg_params, x), layers_to_remove);
map(x -> delete!(nnet.aux_params, x), layers_to_remove);

mx.save_checkpoint(nnet, "weights/inception-v3/InceptionV3-FE",
mx.OptimizationState(1, 0, 0, 0))
```

Now, we are ready to proceed with feature extraction!

Extracting features

The feature extraction process is very similar to the implementation we performed earlier when using MobileNet V2, except for small adjustments, such as image size, shapes, and cache file names. The following script will use the Inception V3 feature extractor and create two dump files corresponding to the `train` and `test` datasets:

```
using Images, MXNet, JLD
```

```
nnet = mx.load_checkpoint("weights/inception-v3/InceptionV3-FE", 0,
mx.FeedForward; context = mx.gpu());

PROJECT_IMG_FOLDER = joinpath("data", "book-covers");
DATASETS = ["train", "test"];
BATCH_SIZE = 100;

for dataset_name in DATASETS

    categories = Int[]

    current_folder = joinpath(PROJECT_IMG_FOLDER, dataset_name)
    file_name = joinpath(PROJECT_IMG_FOLDER, "bookcover30-
    labels-$dataset_name.txt")
    lines = readlines(file_name);
    results = zeros(Float16, (2048, size(lines, 1)))

    for i=1:BATCH_SIZE:size(lines, 1)
        println("BATCH: $i")

        mx_data = mx.zeros((299, 299, 3, BATCH_SIZE));

        for j=1:BATCH_SIZE

            if j % 50 == 0 println("\tIMG: $j") end

            img = imresize(load(joinpath(current_folder, split(lines[i
            + j - 1])[1])), (299, 299));
            img = permutedims(Float16.(channelview(RGB.(img))), (3, 2,
            1));

            img[:, :, :] *= 256.0;
            img[:, :, :] -= 128.;
            img[:, :, :] /= 128.;

            img = reshape(img, 299, 299, 3, 1);
            mx_data[j:j] = img;
        end

        data_provider = mx.ArrayDataProvider(:data => mx_data);
        results[:, i:i+(BATCH_SIZE-1)] = mx.predict(nnet,
        data_provider)
    end

    JLD.save(joinpath(PROJECT_IMG_FOLDER, "dump-$dataset_name-
    inception.jld"), "data", results)
end
```

This process will take even more time than MobileNet V2. You can wait for it to succeed or download the result from my Google Drive.

Building the classifier

We will use the same classifier as in the MobileNet V2 implementation, which consists of a single `FullyConnected` layer and `SoftmaxOutput` layer:

```
using Images, MXNet, JLD

function get_labels_from_file(path)
  return map(x -> parse(Int8, split(x)[2]), readlines(path))
end

PROJECT_IMG_FOLDER = joinpath("data", "book-covers");
DUMP_IMG_TRAIN = joinpath(PROJECT_IMG_FOLDER, "dump-train-inception.jld")
DUMP_IMG_TEST = joinpath(PROJECT_IMG_FOLDER, "dump-test-inception.jld")
LABELS_TRAIN = joinpath(PROJECT_IMG_FOLDER, "bookcover30-labels-train.txt")
LABELS_TEST = joinpath(PROJECT_IMG_FOLDER, "bookcover30-labels-test.txt")

# get the data
train_img_data = JLD.load(DUMP_IMG_TRAIN, "data");
test_img_data = JLD.load(DUMP_IMG_TEST, "data");

train_labels = get_labels_from_file(LABELS_TRAIN);
test_labels = get_labels_from_file(LABELS_TEST);

# shuffle the dataset
shuffle_train_indices = shuffle(1:size(train_labels, 1));
shuffle_test_indices = shuffle(1:size(test_labels, 1));

train_img_data = train_img_data[:, shuffle_train_indices];
train_labels = train_labels[shuffle_train_indices];

test_img_data = test_img_data[:, shuffle_test_indices];
test_labels = test_labels[shuffle_test_indices];

# create data providers
valid_record_count = 5000
train_data_provider = mx.ArrayDataProvider(:data => train_img_data[:,
1:end-valid_record_count], :label => train_labels[1:end-
valid_record_count], shuffle = true, batch_size = 500);
valid_data_provider = mx.ArrayDataProvider(:data => train_img_data[:, end-
valid_record_count+1:end], :label => train_labels[end-
valid_record_count+1:end], shuffle = true, batch_size = 500);
```

```
arch = @mx.chain mx.Variable(:data) =>
  mx.FullyConnected(num_hidden=512) =>
  mx.Activation(name=:relu1, act_type=:relu) =>
  mx.FullyConnected(num_hidden=30) =>
  mx.SoftmaxOutput(mx.Variable(:label))

nnet = mx.FeedForward(arch, context = mx.cpu())
mx.fit(nnet, mx.ADAM(), train_data_provider, eval_data =
valid_data_provider, n_epoch = 10, eval_metric =
mx.MultiMetric([mx.Accuracy(), mx.ACE()]));

mx.save_checkpoint(nnet, "weights/book-cover-classifier",
mx.OptimizationState(1, 0, 0, 0))
```

This process shows a tiny improvement on the validation dataset, with the maximum accuracy reaching 28%. At the end of the process, we will save the new model to disk.

Validating the model

Let's verify if using a more complicated network structure helps us reach 28% on a `test` set:

```
test_data_provider = mx.ArrayDataProvider(:data => test_img_data,
batch_size = 500);
predictions = mx.predict(nnet, test_data_provider);
results = map(x -> findmax(predictions[:, x])[2], 1:size(test_labels, 1));

println(round(mean((results - 1) .== test_labels), 2))
```

Yes! The model reached 28% on a `test` set, which is good! It could be the case that we can get an even higher score by optimizing the model's learning rate or mixing two different models together, but that stays outside of the scope of this book.

Result comparison

The two models we have developed so far provided us with 26% and 28% accuracy. Compared to the results achieved in one of the repositories I referenced earlier, we did a great job.

According to the screenshot available in `Judging-a-Book-by-its-Cover`, the repository author reached 16-18% top accuracy on a `test` set, which is 10% lower than our results! Not bad at all!

Text recognition using Tesseract

As one of the bonus sections in this chapter, I would like to give you a quick introduction to a text recognition process.

There are different tasks where you are actually required to retrieve the content from an image, and we will cover a scenario on how you can read text from a book cover.

Introduction to Tesseract

Tesseract is one of the best free solutions available. It provides **OCR** (**optical character recognition**) in nearly all languages, and depending on which version you use, supports rotation and skewness. It has been supported by Google for the past 12 years, which is good for a long-running project.

Installing Tesseract

Installing Tesseract is very simple:

- Ubuntu users, use `sudo apt-get install tesseract-ocr libtesseract-dev`
- Mac OS users, use `brew install tesseract` for those using Homebrew or `sudo port install tesseract` for those using MacPorts
- Windows users are recommended to download the installer from `https://digi.bib.uni-mannheim.de/tesseract/` and proceed with the installation guide described here at `https://github.com/UB-Mannheim/tesseract/wiki`

After the installation process is complete, please ensure that you can run Tesseract from a `terminal/ console/ cmd` by typing `tesseract`.

Running text recognition

Running `tesseract` from Julia is extremely easy. We will try running it on a single image to see if it works and returns results:

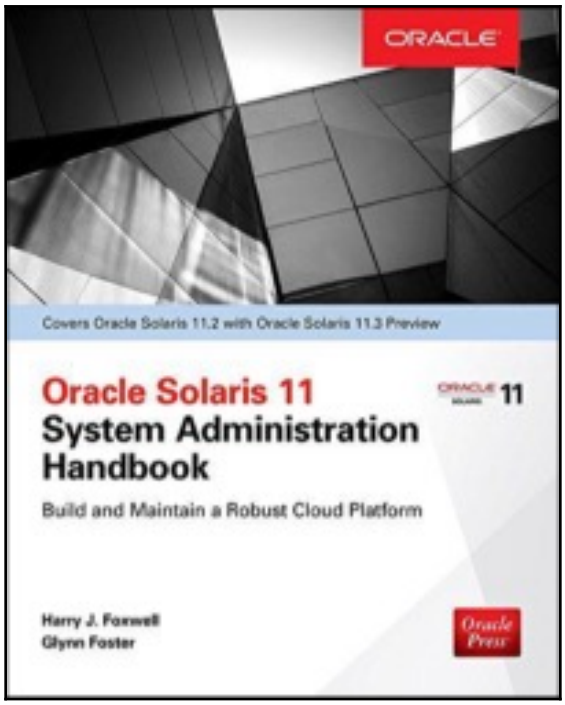
```
TESSERACT_PATH = "tesseract"
image_path = joinpath("data", "book-covers", "test", "007184418X.jpg")

text_str = readstring(`$TESSERACT_PATH $image_path stdout`)
```

```
text_array = readlines(`$TESSERACT_PATH $image_path stdout`)

println(text_str)
println(text_array)
```

It will output both texts as a single string and text as an array of lines. It is up to you to choose which is best:



Consider the following code:

```
Gown emu. Solaria nzwim Ovmk SM. n sPrwiew
System Administration
Handbook
Bmld and MamKain a Robusl C1oud Flaflolm
Harry 1mm"
Glynn Foslev
```

Running commands requires the use of the apostrophe. The use of single or double quotes won't execute the command, and it will fail.

Windows users are required to specify the full path to `tesseract.exe` unless they configured the PATH variable.

Tesseract is currently at Version 4. It is updated on a very frequent basis and the output you see might differ a lot from the preview we have given.

Improving the results

From the previous result, you can see that text recognition worked relatively well, but there were quite a few typos. Tesseract is very sensitive to the quality of an image and its color scheme. I suggest that you use the latest available version of Tesseract and try using a different combination of image enhancement and transformation techniques.

Summary

In this chapter, we built a book cover classifier and performed text recognition on the cover. Our solution managed to achieve better results than those found online. We also covered the basics of Tesseract and were able to recognize text from one of the book covers.