



Hussein Moghnieh, Ph.D.

[Follow](#)

Director of Engineering & Arch. at FastPay (Los Angeles), Founder of citation.io.

Feb 4 · 8 min read

Scanned Digits Recognition using k-Nearest Neighbor (k-NN)



scikit-image
image processing in python



machine learning in Python

Tags: Python, scikit-image, scikit-learn, Machine Learning, OpenCV, ImageMagick, Histogram of Oriented Gradients (HOG).

. . .

How to extract the numbers printed on 500 scanned images with noisy background (as shown below) into an excel file with **100%** accuracy in **2 minutes**?

The simple answer: you can't in 2 minutes, it takes 8 minutes to attain 100% accuracy.

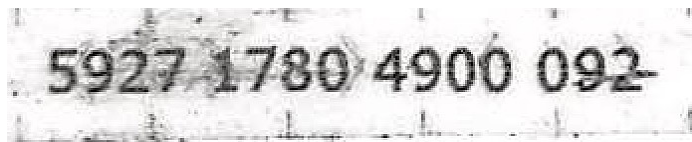


Figure 1. Original image

It takes **2 minutes** to pre-process the images and for a Machine Learning model to correctly predict 98% of the digits and **6 minutes** for a person to manually fix the 2% inaccurate prediction, albeit with minimal effort. The **6 minutes** was made possible by presenting to the

user the digits that the model was unable to classify with 100% confidence as shown in the “Presentation” section at the end of this blog.

Unsuccessful Approaches

Before explaining the k-NN solution, I'll briefly go over some of the unsuccessful methods I've explored to extract the digits.

1- Tesseract — Google's Optical Character Recognition (OCR)

Applying Google's Tesseract resulted in low accurate digits recognition despite using Tesseract's options to recognize an image as a single text line and to OCR digits only. Note that the images background noise were removed before applying Tesseract (more on the de-noising step later in this blog).

2- Image Template Matching

The second approach was to produce template images for each of the 9 digits and then detect each digit in an image and compare it to each of the 0 to 9 templates using openCV's *matchTemplate* function

```
import cv2

result = cv2.matchTemplate(roi, digitROI, cv2.TM_CCOEFF)
(_, score, _, _) = cv2.minMaxLoc(result)
```

This approach did not work for our problem due to noise. However, this blog <https://www.pyimagesearch.com/2017/07/17/credit-card-ocr-with-opencv-and-python/> successfully demonstrates the use of template matching to recognize printed digits on credit cards.

Successful Approach: Training and Predicting using Machine Learning

The last approach was to train my own Machine Learning model. This solution required the following:

- Singling out each digit from an image
- Choosing appropriate feature extraction to apply on each digit
- Choosing a multi-class classifier

Input/data pre-processing, feature engineering, and data preparation lie at the heart of any Machine Learning based solution. The choice of which Machine Learning classifier to use is an important step, however, its success lies in the above mentioned.

Outline:

1. Image Pre-Processing
2. Digits Extraction and Training / Testing Data Preperation
3. Feature Extraction
4. Training
5. Predicting
6. Presentation

1. Image Pre-Processing

TextCleaner script by Fred Weinhaus

(<http://www.fmwconcepts.com/imagemagick/textcleaner/>) has been used to remove the image background noise followed by an image sharpening step. Both these steps require ImageMagick library (<https://www.imagemagick.org>). Alternatively, I recommend using python's libraries such as **OpenCV** or **scikit-image** to pre-process the images.

```
# text cleaner
./textcleaner -g -e stretch -f 25 -o 10 -u -s 1 -T -p 10
input.jpg output_clean.jpg

# image sharpening
convert output_clean.jpg -sharpen 0x10 output_sharp.jpg
```

The above code resulted in the following image

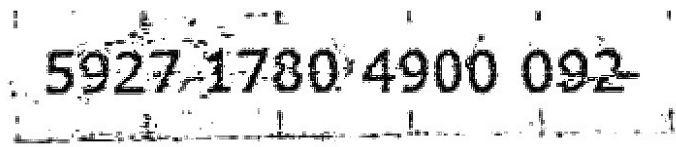


Image 2. De-noised image

2. Digits Extraction and Data Preperation

Singling out each digits from an image using OpenCV's **findContour** operation did not produce reliable results due to noise. For this specific problem, it was more robust to detect the “bounding box” around the digits (image cropping) and then “single out” each digit out of the cropped image. The latter step is easy after finding the bounding box since each digit will have a fixed coordinates relative to the upper-left corner of the cropped image.



Figure 3. De-noised inverted image

Note: Black / White pixels were inverted needed for feature extraction using Histogram of Oriented Gradient (HOG).

2.1 Detecting the Bounding Box

Using third party tools to crop the boundaries of the images did not work well on all images. Instead, I created a simple method to deterministically crop the images and detect the bounding box with 100% accuracy.

The method starts by counting the white pixels of a rectangle as shown in Figure 4. If the count of white pixels exceeds an empirically set value, then the coordinates of the rectangle are the upper boundary of the digits and will be used to crop the image.

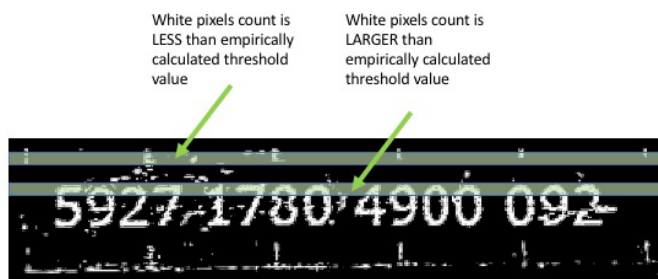


Figure 4. Top image cropping



Figure 5. Top image cropping

The same technique can be used to left crop the image as shown in Figure 6.

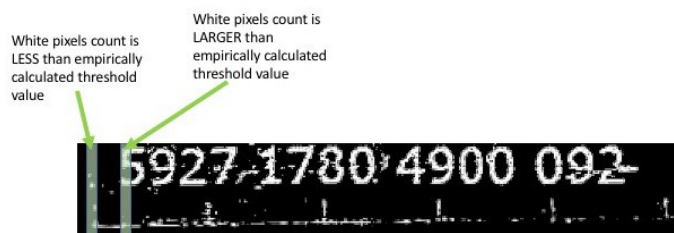


Figure 6. Left image cropping

The output of the above operations resulted in the following image:



Figure 7. Cropped image

```

1  from sklearn import datasets, svm, metrics
2  import scipy
3  import os
4
5
6  directory = '/testing'
7  output_directory = '/testing'
8
9  PRINT_SLICES = False
10 THRESHOLD_PIXELS_COUNT = 60000
11 MAX_BOUNDING_BOX_WIDTH = 675
12 MAX_BOUNDING_BOX_HEIGHT = 50
13
14 def detect_left_edge(image):
15     h,w = image.shape
16     max = 0
17     edge = 0
18     for x in range(0,100):
19
20         vertical_slice = image[0:h, x:x+15 ]
21         vertical_slice_pixels_count = vertical_slice.sum()
22
23         if( vertical_slice_pixels_count > THRESHOLD_PIXELS_
24             scipy.misc.imsave(output_directory + '/' + file
25             return 0
26
27         if (vertical_slice_pixels_count > max):
28             max = vertical_slice_pixels_count
29             edge = x * 2

```

2.2 Digit Extraction

Now that the bounding box is detected, it should be easy to single out each digit since each digit will have pre-fixed coordinates relative to the top-left corner of the cropped image.

I've applied the above code on a set of images and manually sorted the images of each digit into separate folders labeled from 0 to 9 as shown below to create my training / testing dataset.

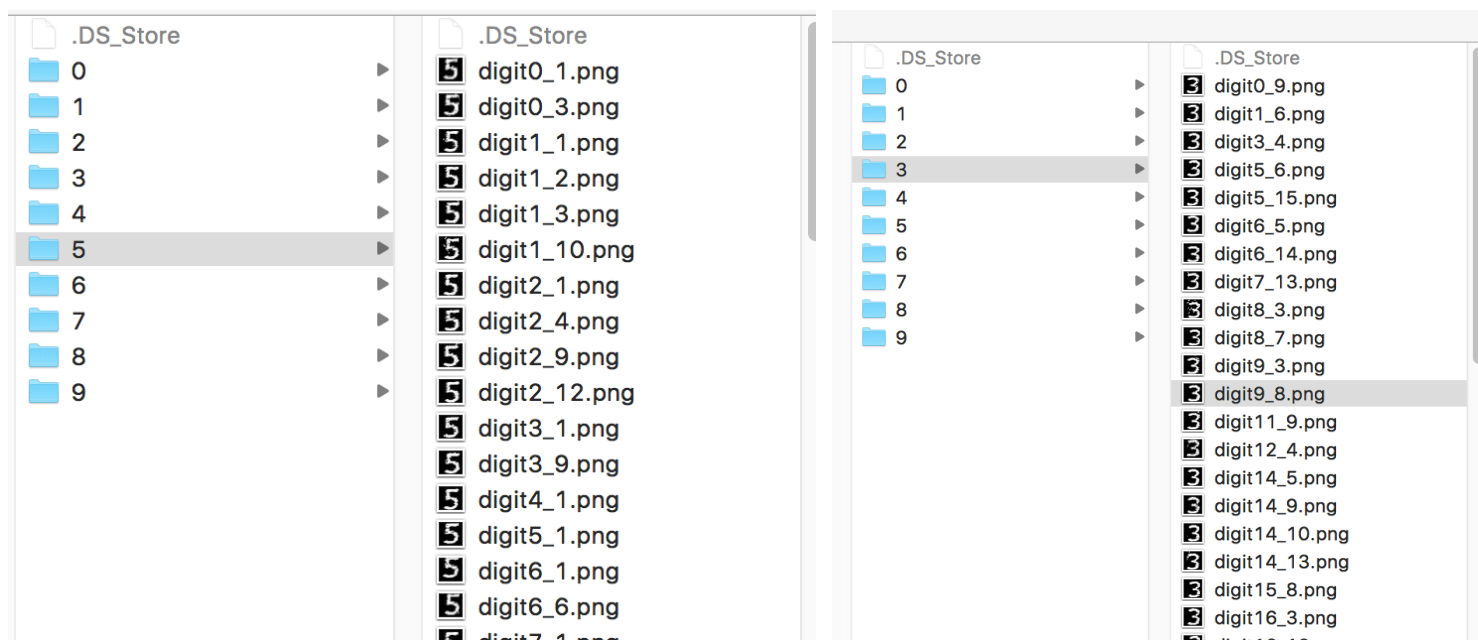


Figure 8. Manually labeling datasets

3. Feature Extraction

Feature extraction or feature engineering is the process of identifying the unique characteristics of an input (digit in our case) to enable a Machine Learning algorithm to work (in our case, to cluster similar digits). Of particular interest is the **Histogram of Oriented Gradients (HOG)** which has been successfully used in many OCR applications to extract handwritten text. The following code illustrates extracting HOG from an image using skimage's *hog* function.

```
from skimage.feature import hog

df= hog(training_digit_image, orientations=8,
pixels_per_cell=(10,10), cells_per_block=(5, 5))
```

In my case, the image is 50x50 pixels and hog's input parameters (i.e. *pixels_per_cell* and *cells_per_block*) were empirically set. The figure below illustrates applying HOG on an image producing a vector of 200 values (i.e. features).

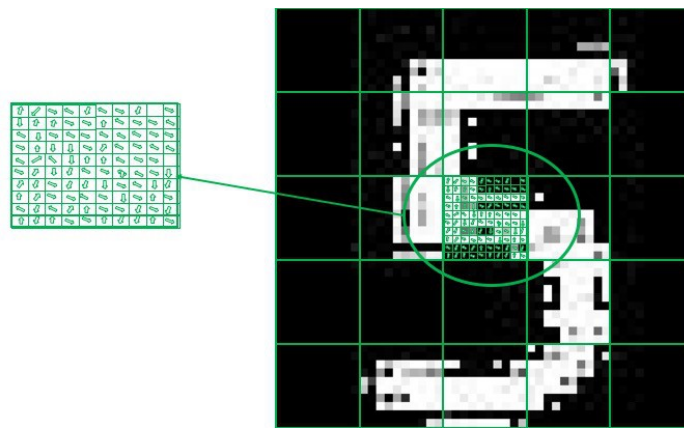


Figure 9. Illustration of Histogram of Oriented Gradients (HOG)

4. Training

In the previous steps, we extracted similar digits into folders to build our training dataset. The code below illustrates building our training / test dataset.

```

1  import numpy as np
2  import os
3  import scipy.ndimage
4  from skimage.feature import hog
5  from skimage import data, color, exposure
6  from sklearn.model_selection import train_test_split
7  from sklearn.neighbors import KNeighborsClassifier
8  from sklearn.externals import joblib
9
10 features_list = []
11 features_label = []
12 # load labeled training / test data
13 # loop over the 10 directories where each directory stores
14 for digit in range(0,10):
15     label = digit
16     training_directory = '/training_data_set/' + str(label)
17     for filename in os.listdir(training_directory):
18         if (filename.endswith('.png')):
```

Now that we created the training dataset and stored it into **features** and **features_label** arrays, we then divided our training sets into training and test sets using sklearn's function **train_test_split** and used the result to train a k-NN classifier and finally saved the model as illustrated in the code below.

```

# store features array into a numpy array
features = np.array(features_list, 'float64')

# split the labeled dataset into training / test sets
X_train, X_test, y_train, y_test =
train_test_split(features, features_label)
```

```
# train using K-NN
knn = KNeighborsClassifier(n_neighbors=3)
knn.fit(X_train, y_train)

# get the model accuracy
model_score = knn.score(X_test, y_test)

# save trained model
joblib.dump(knn, '/models/knn_model.pkl')
```

5. Predicting

The process of predicting digits on new images follows the same steps of singling out the digits illustrated in the training steps above and then simply applying k-NN's *predict* function as shown below.

```
knn = joblib.load('/models/knn_model.pkl')

def feature_extraction(image):
    return hog(color.rgb2gray(image), orientations=8,
               pixels_per_cell=(10, 10), cells_per_block=(5, 5))

def predict(df):
    predict = knn.predict(df.reshape(1,-1))[0]
    predict_proba = knn.predict_proba(df.reshape(1,-1))
    return predict, predict_proba[0][predict]

digits = []

# load your image from file

# extract features
hogs = list(map(lambda x: feature_extraction(x), digits))

# apply k-NN model created in previous
predictions = list(map(lambda x: predict(x), hogs))
```

k-NN's *predict* function returns a single digit value between 0 and 9 to denote the *prediction class* of the input image. K-NN's *predict_proba* function returns the accuracy associated with each predicted class.

For instance, assume that we applied prediction on an image containing the digit “5”. An example of an output would be

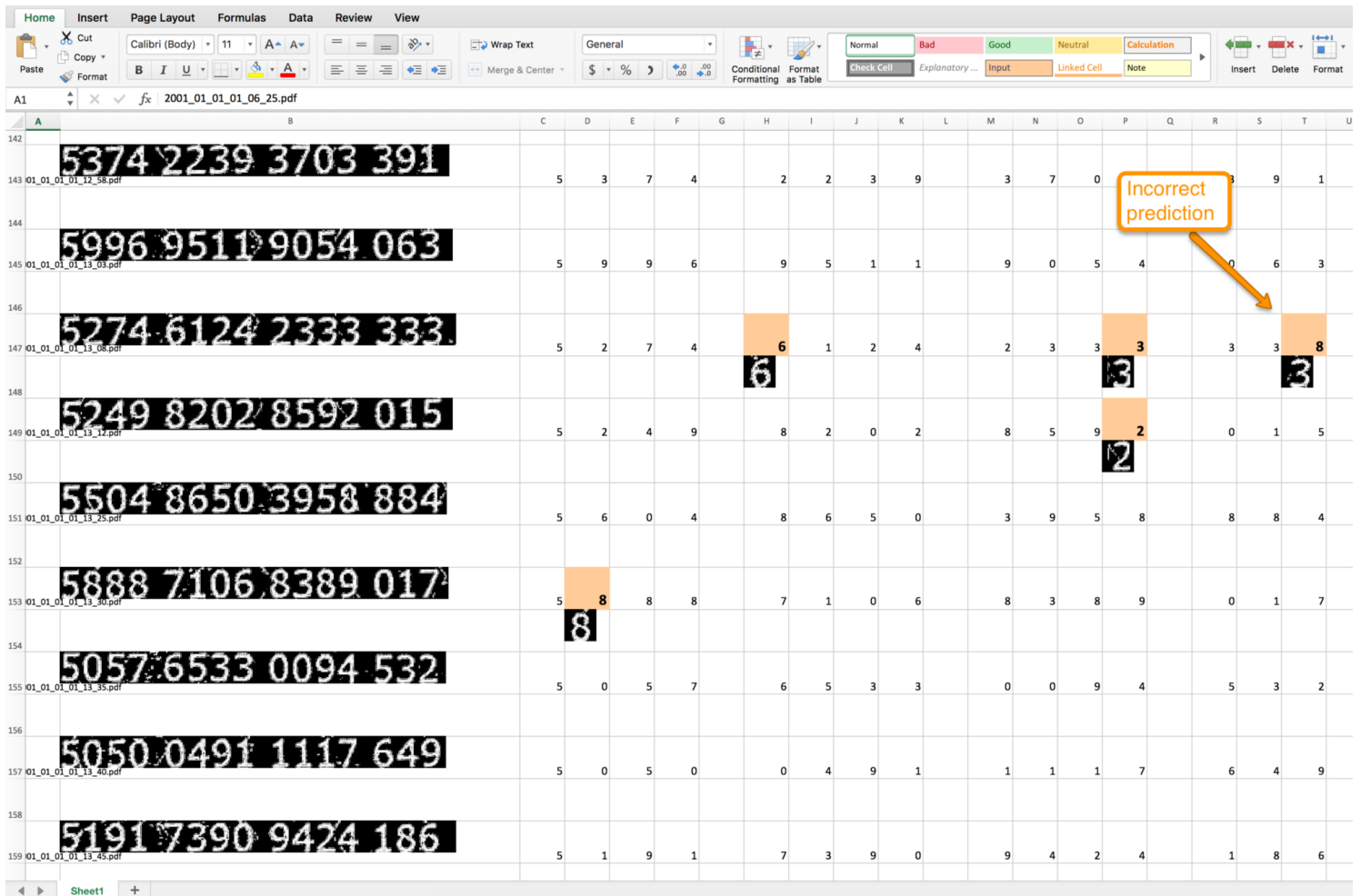
`prediction=5` and `predict_proba = [[0 0 0 0 0 .8 0 0 .2 0]]`. This means that k-NN classified the image as “5” with 80% confidence and as “8” with 20% confidence.

Finally, `predictions = list(map(lambda x: predict(x), hogs))` results in the following vector of tuples where each tuple represents the predicted class of each of the digits on the image with its associated prediction confidence. Any prediction that does not classify an input with 100% confidence will be presented to the user for manual correction as illustrated in the next section.


```
[
  (5, 1.0), (1, 1.0), (9, 1.0), (2, 1.0), (1, 1.0), (2, 1.0),
  (4, 1.0), (7, 1.0), (2, 1.0), (3, 1.0), (4, 1.0), (3, 1.0),
  (4, 1.0),
  (4, 0.8), (0, 1.0)
]
```

6. Presentation

The last step was to present the result of the Machine Learning model in an excel file as shown below. For digits that were not predicted with 100% accuracy, I embedded the image of the expected digit below the actual prediction. This minor presentation tweak decreased the user's time to fix the non accurate prediction by 80%. Furthermore, this activity is not daunting as it does not require significant mental effort. A user can scroll over the file in few minutes and visually matches the actual result to the expected result. Many of the predictions were actually false negative, hence the user did not have to make many corrections.



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
142		5374 2239 3703 391																			
143	01_01_01_01_12_58.pdf		5	3	7	4		2	2	3	9		3	7	0				9	1	
144		5996 9511 9054 063																			
145	01_01_01_01_13_03.pdf		5	9	9	6		9	5	1	1		9	0	5	4			0	6	3
146		5274 6124 2333 333																			
147	01_01_01_01_13_08.pdf		5	2	7	4		6	1	2	4		2	3	3	3		3	3	8	
148		5249 8202 8592 015						6								3					
149	01_01_01_01_13_12.pdf		5	2	4	9		8	2	0	2		8	5	9	2		0	1	5	
150		5504 8650 3958 884																			
151	01_01_01_01_13_25.pdf		5	6	0	4		8	6	5	0		3	9	5	8		8	8	4	
152		5888 7106 8389 017																			
153	01_01_01_01_13_30.pdf		5	8	8	8		7	1	0	6		8	3	8	9		0	1	7	
154		5057 6533 0094 532																			
155	01_01_01_01_13_35.pdf		5	0	5	7		6	5	3	3		0	0	9	4		5	3	2	
156		5050 0491 1117 649																			
157	01_01_01_01_13_40.pdf		5	0	5	0		0	4	9	1		1	1	1	7		6	4	9	
158		5191 7390 9424 186																			
159	01_01_01_01_13_45.pdf		5	1	9	1		7	3	9	0		9	4	2	4		1	8	6	

Reading List

Hamid, N. A., & Sjarif, N. N. A. (2017). *Handwritten Recognition Using SVM, KNN and Neural Network*. *arXiv preprint arXiv:1702.00723*.

Adrian Rosebrock's blogs and books
(<https://www.pyimagesearch.com>). Great computer vision resources and many posts on digits recognition.

Patel, I., Jagtap, V., & Kale, O. (2014). A Survey on Feature Extraction Methods for Handwritten Digits Recognition. *International Journal of Computer Applications*, 107(12).