低品質文字画像を用いた
高精細画像からの字形自動再切り出しの試み

守岡 知彦

1　はじめに

漢字字体規範史データセット (HNG dataset) [5] や平安時代漢字字書総合データベース (HDIC) [3] の古字書の掲出字画像のような歴史的な漢字字形の切り出し画像データは漢字字体の規範意識の変遷を知る上で重要なデータである。かつては高精細で自由に利用可能な全文画像を入手するのは簡単ではなかったため、例えば HNG の場合、初期のデータは画質の低いコピーをはさみで切り張りして作られた紙カードをスキャンして作成されたため、画質が低いものが少なくない。近年、IIIF を用いて比較的高精細な全文画像を公開する動きが進んでおり、HNG のソースとなった文献の内、敦煌写本の幾つかはフランス国立図書館の電子図書館 Gallica で公開されており、HNG に対応する文献を人手で再切り出しする試みも行われているが [4] 手作業では労力がかかるため大量の全文画像を全て再切り出しするのは現実的ではない。そこで、画像処理技術を用いてこの作業を自動化することを試みた。

2　方法

低品質な切り出し字形画像をテンプレート画像として高精細全文画像に対してテンプレートマッチングをかける。これにより切り出し字形画像にマッチする高精細全文画像中の矩形座標を得る。

具体的には OpenCV の Python バインディングである OpenCV-Python 上で、物体検出用の関数 matchTemplate [7] を用いて、テンプレートマッチングを行う。

このテンプレートマッチングは下記のように行うことができる：

```
# （全文）画像の読み込み
img = cv2.imread(image_filename)

# グレースケール化
img_gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
```

```
# テンプレート（切り出し字形）画像の読み込み
template = cv2.imread(template_filename)

# テンプレートファイルのグレースケール化
template_gray = cv2.cvtColor(template, cv2.COLOR_BGR2GRAY)

# 処理対象画像に対して、テンプレート画像との類似度を算出する
res = cv2.matchTemplate(img_gray, template_gray, cv2.TM_CCOEFF_NORMED)

# 類似度の高い部分を検出する
loc = np.where(res >= threshold)

# テンプレートマッチング画像の高さ・幅を取得する
h, w = template_gray.shape

# 検出した部分の表示
for pt in zip(*loc[::-1]):
    print ('%s\t%d,%d,%d,%d' %
           (template_filename,
            round(pt[0]*5619/img_width, 0), round(pt[1]*4693/img_height, 0),
            round(w*5619/img_width, 0), round(h*4693/img_height, 0)))
```

ここでは全文画像、切り出し画像ともグレースケール化し比較を行っている。

但し、テンプレート画像（低品質な切り出し字形画像）と画像（高精細全文画像）のテンプレートマッチングを行う際、両者の解像度が著しく異なるとうまく行かない。そこで、テンプレート画像と高精細全文画像中の該当領域のピクセル単位でのサイズが大体同じ大きさになるように、テンプレート画像を拡大するか画像を縮小するか、その両方を行ってからテンプレートマッチングを行う。

3　HDIC-KTB の篆書掲出字形の自動再切り出し

李媛氏から提供して頂いた篆隷万象名義の篆書掲出字形を国会図書館デジタルコレクションで公開されている全文画像を用いて自動際切り出しを試みた。この李媛氏から提供して頂いた篆書掲出字形画像は、一部に横幅が 140 ピクセルを越えるものもあるものの、横幅が 40〜60 ピクセル前後のものが多く、最小のものは横幅 29 ピクセルである。なお、全て色深度 8bit のモノクロ画像である。

一方、国会図書館の永続的識別子 info:ndljp/pid/1245837 など[*1]で公開されている画像は横幅 5619 ピクセル、縦幅 4693 ピクセルで、字形のサイズは 200〜300 ピクセル前後のものが多い。

よって、この低解像度篆書掲出字形画像を 5 倍に拡大してテンプレート画像として用いればうまくテンプレートマッチングを行うことができるはずである。

しかしながら、国立国会図書館デジタルコレクションの IIIF マニフェストでは実際の解像度が記載され、IIIF Image API の info.json でも同様であるのに対し、IIIF Image API の画像リクエスト URI においてサイズとして full を指定して画像の取得を行った場合、実際には画像サイズの縦横いずれかの上限が 5000 ピクセルに縮小されてしまう。[6] このため、横幅を半分の 2809 ピクセルに縮小して画像を取得し、低解像度篆書掲出字形画像を 2.5 倍に拡大したものをテンプレート画像として用いるようにした。[*2]

この手法に基づき実験した結果、1042 字形中 1013 字形の座標が取得できた。

この結果は `https://github.com/chise/HDIC/blob/master/KTB_ndl_Seal.tsv` で公開している。

座標取得に失敗した 29 字形の内、28 字形は幅が 140 ピクセルを越えるものであり、1 字形は幅が 29 ピクセルであった。この 28 字形に関しては、テンプレート画像の拡大率を下げる必要があると考えられる。

4　本文字形の自動再切り出し

多くの物体検出の手法では与えられたテンプレート画像にマッチする複数の結果が返る。そして、その結果の中には似た（大部分が重なった）領域のものが多数含まれ得る。

古字書の掲出字形の場合、その字体に似た字形は本文画像に 1 回しか現れないため、テンプレートマッチングを行った際、最もスコアの高いものを選べば良いが、通常の本文テキストの場合、一般に同じ文字・字体が複数回現れるため、この方法では全ての出現字形の座標を得ることができない。一方、何もしなければ実際には 1 つの字形に対して複数の領域が対応してしまい問題である。

そこで、重複した領域を 1 つにまとめるための方法として Non-Maximum Suppression (NMS) という手法を用いる。

互いに重なった複数の領域に対し、重なり具合が閾値を越えたものを削ることで、重なり具合の大きい領域を抑制する方法である。この重なり具合は IoU (Intersection over Union) という値で定義される。これは互いに重なった 2 つの領域に対して、その重なり部分 (intersection) の面積を両者の和領域 (union) の面積（2 つの領域の面積の和から重なり部分の面積を引いたもの）で割ったものである。

なお、NMS は崩し字のように字形領域の重なりが大きい場合には向いていないと思われる。このようなケースには Soft-NMS (Soft Non-Maximum Suppression) [1] を用いた方が良いかも知

[*1] この詳細は [2] の `KTB_ndl.txt` に記載されている。

[*2] なお、OpenCV の関数 cv2.resize よりも ImageMagic の convert のリサイズ機能の方が結果が良かったため、こちらを用いている。

れない。これは NMS とほぼ同じ計算量のアルゴリズムで、IoU が閾値を越えたものを削る代わりにスコアを下げる手法である。NMS の場合、領域の重複が多いものは問答無用で削られてしまうが、Soft-NMS の場合、元のスコアが高ければ領域の重複が大きくてもそれなりのスコアが残るため敗者復活する可能性が生じる。

HNG のような楷書字体を対象にする場合、領域の重複はほとんどないため、通常の NMS でも問題ないと考えられる。

このように、NMS を使って後処理を行う方法を用いて、`img-geo.py` というコマンドラインツールと `https://image.chise.org/iiif-glyph-match.html` という Web サービスを試作した。

前者の使い方は次の通りである：

> img-geo.py 画像のファイル名 テンプレート画像のファイル名 拡大率（%）閾値（0〜1）

また、このソースコードを付録 付録 A に示す。なお、この Python プログラムにおける NMS の実装は `https://python-ai-learn.com/2021/02/14/nmsfast/` で紹介されている numpy を用いた高速な実装に基づき若干の修正を行ったものである。

また、後者も基本的に同様であるが、画像ファイル名の代わりに IIIF Image API の URL（例：`https://gallica.bnf.fr/iiif/ark:/12148/btv1b10099532k/f2`）、テンプレート画像のファイル名の代わりにテンプレート画像の URL を指定する。

HNG-MAM（摩訶摩耶經卷上 (P.2160)）の「念」の例示字形を 1.4 倍に拡大したものをテンプレート画像として用いて、Gallica の全文画像にテンプレートマッチングを行った結果は次の URL でアクセスできる：

```
https://image.chise.org/img-match?iiif-image=https%3A%2F%2Fgallica.
bnf.fr%2Fiiif%2Fark%3A%2F12148%2Fbtv1b10099532k%2Ff2&template=https%
3A%2F%2Fimage.hng-data.org%2Fglyphs%2FHNG%2F005%2F0620.png&scale=140&
threshold=0.65
```

図 1 にこの結果のスクリーンショットを示す。

また、HNG-DRT（大樓炭經卷三 (P2413)）の「念」の例示字形を 1.35 倍に拡大したものをテンプレート画像として用いた場合も同様の結果を得ることができる：

```
https://image.chise.org/img-match?iiif-image=https%3A%2F%2Fgallica.
bnf.fr%2Fiiif%2Fark%3A%2F12148%2Fbtv1b10099532k%2Ff2&template=https%
3A%2F%2Fimage.hng-data.org%2Fglyphs%2FHNG%2F006%2F0304.png&scale=135&
threshold=0.54
```

図 2 にこの結果のスクリーンショットを示す。

このように、長安宮廷写経の場合、字形が比較的均質であるため、別の文献の同じ字体の HNG の例示字形をテンプレートとして用いても、ある程度、字形を切り出せることが判る。

図 1　P.2160 に HNG-MAM (P.2160) の「念」の例示字形でテンプレートマッチングを行った例

図 2　P.2160 に HNG-DRT (P.2413) の「念」の例示字形でテンプレートマッチングを行った例

5　おわりに

OpenCV の物体検出用の関数 matchTemplate を用いてテンプレートマッチングを行うことで、低画質・低解像度の切り出し字形画像を使って高精細画像から字形画像の自動再切り出しを行うこ

とができた。但し、テンプレートマッチングは原理的にテンプレート画像とマッチング対象となる画像の該当領域のサイズがおおむね同じでないとうまくいかないが今回はテンプレート画像の拡大率を手動で指定することとしたが、この自動化は今後の課題である。

デジタル写真技術は時代とともに進歩するため、新しい機材を使って資料を撮影しなおすことは有効であるが、古い写真に基づいて作成された座標データなどはそのままでは利用できないという問題があり、その解決法を確立することは重要な課題のひとつであるといえるが、今回の試みはその解決法の一例になっていると考えられる。

また、漢字字体史研究の観点では OCR のようなテキストデータの自動認識とは別に、字体や字形の自動認識が望まれるが、今回用いたような物体検出技術に基づく方法はまさにこうしたグリフの自動認識につながるものといえる。

謝辞

篆隷万象名義の篆書掲出字形を提供して頂いた李媛氏に感謝する。

参考文献

[1] Navaneeth Bodla et al. “Soft-NMS – improving object Detection with one line of code”. In: *Proceedings of the 2017 IEEE international conference on computer vision (ICCV)*. 2017, pp. 5561–5569.

[2] 池田 証壽. *HDIC Database Project*. `https://github.com/shikeda/HDIC`. Mar. 2022.

[3] 平安時代漢字字書総合データベース編纂委員会. **平安時代漢字字書研究**. `https://hdic.jp/`. Mar. 2022.

[4] Masanao Saiki et al. *HNG-Kiridashi-data*. `https://gitlab.hng-data.org/HNG/hng-kiridashi-data`. Feb. 2020.

[5] 石塚晴通, 高田 智和, and 守岡 知彦. **漢字字体規範史データセット保存会**. `http://www.hng-data.org/`. July 2018.

[6] *IIIF* **に関するヘルプ（国立国会図書館 デジタルコレクション）**. `https://dl.ndl.go.jp/ja/help_iiif.html`. 2022.

[7] *Template Matching (OpenCV-Python Tutorials)*. `https://docs.opencv.org/4.x/d4/dc6/tutorial_py_template_matching.html`. 2022.

付録 A　img-geo.py のソースコード

```
#!/usr/bin/python3
```

```
import sys
import os
import cv2
import numpy as np

# 矩形 a と、複数の矩形 b の IoU を計算
def np_iou(a, b, a_area, b_area):
    # a は 1 つの矩形を表す shape=(4,) の numpy 配列
    # array([xmin, ymin, xmax, ymax])
    # b は任意の N 個の矩形を表す shape=(N, 4) の numpy 配列
    # 2 次元目の 4 は、array([xmin, ymin, xmax, ymax])

    # a_area は矩形 a の面積
    # b_area は b に含まれる矩形のそれぞれの面積
    # shape=(N,) の numpy 配列。N は矩形の数

    # a と b の矩形の共通部分 (intersection) の面積を計算するために、
    # N 個の b について、a との共通部分の xmin, ymin, xmax, ymax を一気に計算
    abx_mn = np.maximum(a[0], b[:,0]) # xmin
    aby_mn = np.maximum(a[1], b[:,1]) # ymin
    abx_mx = np.minimum(a[2], b[:,2]) # xmax
    aby_mx = np.minimum(a[3], b[:,3]) # ymax
    # 共通部分の幅を計算。共通部分が無ければ 0
    w = np.maximum(0, abx_mx - abx_mn + 1)
    # 共通部分の高さを計算。共通部分が無ければ 0
    h = np.maximum(0, aby_mx - aby_mn + 1)
    # 共通部分の面積を計算。共通部分が無ければ 0
    intersect = w*h

    # N 個の b について、a との IoU を一気に計算
    np_iou = intersect / (a_area + b_area - intersect)
    return np_iou

# NMS の計算
def non_max_suppression(bboxes, scores, iou_threshold=0.5):
    # bboxes は任意の N 個の矩形を格納した shape=(N, 4) の numpy 配列
    # 2 次元目の 4 要素は、array([xmin, ymin, xmax, ymax])
```

```
    # scores は任意の N 個の信頼度を格納した shape=(N,) の numpy 配列

    # bboxes の矩形の面積を一気に計算
    areas = (bboxes[:,2] - bboxes[:,0] + 1) \
          * (bboxes[:,3] - bboxes[:,1] + 1)

    # score の昇順（小さい順）の矩形インデックスのリストを取得
    sort_index = np.argsort(scores)

    i = -1 # 未処理の矩形の index
    while(len(sort_index) >= 2 - i):
        # score 最大の index を取得
        max_scr_ind = sort_index[i]
        # score 最大以外の index を取得
        ind_list = sort_index[:i]
        # score 最大の矩形それ以外の矩形の IoU を計算
        iou = np_iou(bboxes[max_scr_ind], bboxes[ind_list], \
                     areas[max_scr_ind], areas[ind_list])

        # IoU が閾値 iou_threshold 以上の矩形を計算
        del_index = np.where(iou >= iou_threshold)
        # IoU が閾値 iou_threshold 以上の矩形を削除
        sort_index = np.delete(sort_index, del_index)
        #print(len(sort_index), i, flush=True)
        i -= 1 # 未処理の矩形の index を 1 減らす

    # bboxes から削除されなかった矩形の index のみを抽出
    bboxes = bboxes[sort_index]

    return bboxes


args = sys.argv
image_filename = args[1]
template_filename = args[2]
scale = args[3]
threshold = float(args[4])
```

```
# （全文）画像の読み込み
img = cv2.imread(image_filename)

# グレースケール化
img_gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

# テンプレート画像のリサイズ（ImageMagic の convert コマンドを利用）
cmd = "convert -resize %s%% %s /tmp/template.png"%(scale,template_filename)
os.system(cmd)
template = cv2.imread('/tmp/template.png')
template_gray = cv2.cvtColor(template, cv2.COLOR_BGR2GRAY)

# 処理対象画像に対して、テンプレート画像との類似度を算出する
res = cv2.matchTemplate(img_gray, template_gray, cv2.TM_CCOEFF_NORMED)

# 類似度の高い部分を検出する
loc = np.where(res >= threshold)

scores = res[loc]

# テンプレートマッチング画像の高さ、幅を取得する
h, w = template_gray.shape

# 検出した部分を格納
boxes = []
for pt in zip(*loc[::-1]):
    boxes.append([pt[0], pt[1], pt[0] + w, pt[1] + h])
boxes = np.array(boxes)

# NMS の実行
boxes = non_max_suppression(boxes, scores, 0.6)

# 検出した部分を表示
for x1, y1, x2, y2 in boxes:
    print ("\t%d,%d,%d,%d" % (x1, y1, x2 -x1, y2 - y1))
```