Open in app Get started

Olga Kravchenko Follow
Jun 17, 2018 · 6 min read · Listen

Save

(Generalized) Procrustes analysis with Python/NumPy

One of the great discoveries that I came across when I got into stats is that people in the field sometimes have a great sense of humor. Granted, I assumed the opposite based solely on the field's stereotypical representation, so learn from my mistakes, kids! Procrustes analysis is named after a bandit from ancient Greek mythology that would hunt people down, and adjust their sizes to his iron bed by either cutting their limbs off or stretching them. The algorithm that implements it does something similar to a set of shapes. The general idea is to analyze the statistical shape variation, aka to find a degree of similarity between different shapes in a set. If there's a reference shape present, and all the other shapes are compared to it, we have ourselves what is referred to as **classical Procrustes analysis**. If what we're dealing with is just a bunch of shapes floating around, and there's no info suggesting a good candidate to be chosen for reference shape, the problem is referred to as **generalized Procrustes analysis**. In general, a term "shape" is used loosely, and the method can be interpolated from explicit shapes to abstract such as sets of numbers that need to be compared. However I came across the method in computer vision course, so I'll be mostly talking about it in that context.

Let's say we have ourselves five random triangles scattered across 2d plane, each vertex defined by an (x,y) coordinate. To compare them, they first need to be aligned to each other as closely as possible (hence Procrustes, get it? Get it?). In more practical terms, we need to find an optimal scale, rotation and translation that would make all of the triangles overlap w.r.t. either an explicitly selected reference landmark, or a mean landmark that is calculated in one way or the other. More formally, for classical Procrustes the steps are:

2. For each of the remaining shapes, calculate scale, rotation and translation that would align it to a reference shape as closely as possible

3. Calculate Procrustes distance by first calculating an SSD for each point w.r.t a reference point, then summing those and taking a square root of the sum

For generalized Procrustes analysis there's no reference shape to begin with, so what all the shaped are compared against is a mean shape that is chosen arbitrarily and then iteratively improved. The algorithm goes like:

1. Select a random shape as a mean shape (usually the first shape in the set is taken)

2. Align all the other shapes to it

3. Calculate a new mean shape, compare it to the old shape (e.g. by subtracting one from the other)

4. If the difference in mean shapes is above some margin, align the new mean to the old mean and return to step 2.

The second algorithm usually converges in a couple iterations.

Now to the practical part. For the examples here I will assume that each shape is represented by a 2n x 1 vector with with coordinates (x1,y1, x2, y2,…xn, yn). In principle it is not necessary and an 102 2 ill do, however there are some operations that make use of NumPy functionality for which this format optimal. All the visualizations will be done in OpenCV. Let's create and display five triangles, just to see what we will be working with.

Important: Procrustes analysis aligns shapes point by point, so it is necessary to preserve the same relative order of vertices for each triangle. Here, for the first four triangles I started with the right angle, went up the longer tangent, and went back down. For for the last triangle I went base left — up — base right.

The code below has some simple helper functions that I will be using throughout the post. The create_test_set function does not offer much flexibility, however should you test the algorithm for yourself, I assume you'll have a set of data to work with, so I did

```python
from random import randint

def show_image(img):
    '''
    Displays an image
    Args:
        img(a NumPy array of type uint 8) an image to be
        dsplayed
    '''

    cv2.imshow('', img)
    cv2.waitKey(5000)

 def generate_color():
    '''
    Generates a random combination
    of red, green and blue channels
    Returns:
        (r,g,b), a generated tuple
    '''
    col = []
    for i in range(3):
        col.append(randint(0, 255))

    return tuple(col)

def create_test_set():

    #create canvas on which the triangles will be visualized
    canvas = np.full([400,400], 255).astype('uint8')

    #convert to 3 channel RGB for fun colors!
    canvas = cv2.cvtColor(canvas,cv2.COLOR_GRAY2RGB)

    #initialize triangles as sets of vertex coordinates (x,y)
    triangles = []
    triangles.append(np.array([250,250, 250,150, 300,250]))
    #tr1 translated by 50 points on both axis
    triangles.append(triangles[0] - 50)
    #tr1 shrinked and consequently translated as well
    triangles.append((triangles[0] / 2).astype(np.int))
    #tr1 rotated by 90 defrees annd translated by 20 pixels
    triangles.append(np.array([250,250,150,250, 250, 200]) - 20)
```

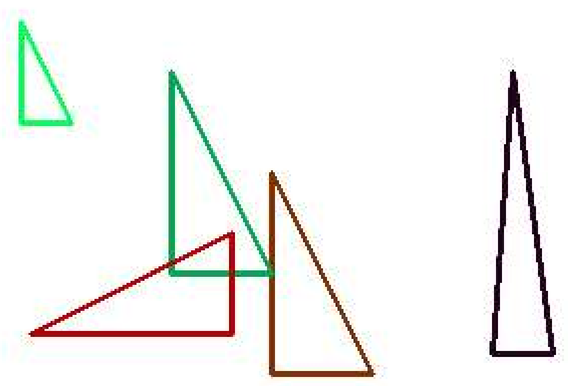
```
def draw_shapes(canvas, shapes):
    '''
    Draws shapes on canvas
    Args:
        canvas(a NumPy matrix), a background on which
        shapes are drawn
        shapes(list), shapes to be drawn
    '''

    for sh in shapes:
        pts = sh.reshape((-1,1,2))
        color = generate_color()
        cv2.polylines(canvas, [pts], True, color, 2)

    show_image(canvas)

if __name__ == '__main__':

    canvas, triangles = create_test_set()
    draw_shapes(canvas, triangles)
```



As you can see, what we're working with is four variants of the same shape, and one additional one that does not really line up with anything.

Exhibit A. Procrustes analysis.

First step is just translating all the points to be centered at the origin, which is done simply by subtracting the mean of the axis from every value of those axis. This will save a lot of pain when finding the rotation angle and the scale.

```python
import numpy as np

def get_translation(shape):
  '''
  Calculates a translation for x and y
  axis that centers shape around the
  origin
  Args:
    shape(2n x 1 NumPy array) an array
    containing x coodrinates of shape
    points as first column and y coords
    as second column
   Returns:
    translation([x,y]) a NumPy array with
    x and y translationcoordinates
  '''

  mean_x = np.mean(shape[::2]).astype(np.int)
  mean_y = np.mean(shape[1::2]).astype(np.int)

  return np.array([mean_x, mean_y])

def translate(shape):
  '''
  Translates shape to the origin
  Args:
    shape(2n x 1 NumPy array) an array
    containing x coodrinates of shape
    points as first column and y coords
    as second column
  '''
  mean_x, mean_y = get_translation(shape)
```

Now, to scale and rotation. In the manual that I've used, they give no interpretation for the formulas that are used to calculate scale factor and rotation angle. They work though, so I call this magic! Here's how you do it:

```python
from scipy.linalg import norm
import numpy as np
from math import atan

def get_rotation_scale(reference_shape, shape):
    '''
    Calculates rotation and scale
    that would optimally align shape
    with reference shape
    Args:
        reference_shape(2nx1 NumPy array), a shape that
        serves as reference for scaling and
        alignment

        shape(2nx1 NumPy array), a shape that is scaled
        and aligned

    Returns:
        scale(float), a scaling factor
        theta(float), a rotation angle in radians
    '''

    a = np.dot(shape, reference_shape) / norm(reference_shape)**2

    #separate x and y for the sake of convenience
    ref_x = reference_shape[::2]
    ref_y = reference_shape[1::2]

    x = shape[::2]
    y = shape[1::2]

    b = np.sum(x*ref_y - ref_x*y) / norm(reference_shape)**2

    scale = np.sqrt(a**2+b**2)
    theta = atan(b / max(a, 10**-10)) #avoid dividing by 0

    return round(scale,1), round(theta,2)
```

matrix:

$$R = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

Mind that the rotation is assumed to be performed around the origin. In case you want to rotate around an arbitrary point (x,y), just subtract it from the vector, perform rotation and then add it back.

```
from math import sin, cos
import numpy as np

def get_rotation_matrix(theta):

    return np.array([[cos(theta), -sin(theta)], [sin(theta), cos(theta)]])

def scale(shape, scale):

    return shape / scale

def rotate(shape, theta):
    '''
    Rotates a shape by angle theta
    Assumes a shape is centered around
    origin
    Args:
        shape(2nx1 NumPy array) an shape to be rotated
        theta(float) angle in radians
    Returns:
        rotated_shape(2nx1 NumPy array) a rotated shape
    '''

    matr = get_rotation_matrix(theta)

    #reshape so that dot product is eascily computed
    temp_shape = shape.reshape((-1,2)).T

    #rotate
    rotated_shape = np.dot(matr, temp_shape)
```

And it looks like we're done. Time to pile all of it together and see the beautiful triangular caterpillar emerge from a less beautiful chrysalis that also happens to be triangular!

```python
def procrustes_analysis(reference_shape, shape):
    '''
    Scales, and rotates a shape optimally to
    be aligned with a reference shape
    Args:
        reference_shape(2nx1 NumPy array), a shape that
        serves as reference alignment

        shape(2nx1 NumPy array), a shape that is aligned

    Returns:
        aligned_shape(2nx1 NumPy array), an aligned shape
        translated to the location of reference shape
    '''
    #copy both shapes in caseoriginals are needed later
    temp_ref = np.copy(reference_shape)
    temp_sh = np.copy(shape)

    translate(temp_ref)
    translate(temp_sh)

    #get scale and rotation
    scale, theta = get_rotation_scale(temp_ref, temp_sh)

    #scale, rotate both shapes
    temp_sh = temp_sh / scale
    aligned_shape = rotate(temp_sh, theta)

    return aligned_shape
```

Now, if this whole thing works, the triangles should be neatly aligned on top of each other. I will be translating all the shapes back to the position of reference landmark for visualization.

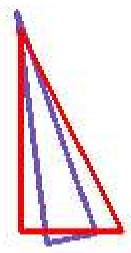
```python
if __name__ == '__main__':
    new_shapes = []
    new_shapes.append(triangles[0])

    #superimpose all shapes to reference shape
    for i sh in range(1,5):
        new_shape = procrustes_analysis(triangles[0], triangles[i])
        new_shape[::2] = new_shape[::2] + x
        new_shape[1::2] = new_shape[1::2] + y
        new_shapes.append(new_shape)

    draw_shapes(canvas, new_shapes)
```

procr_test.py hosted with ❤ by GitHub view raw

Aaand looks a-okay from here. The first 4 shapes are overimposed beautifully. The result on the last one is not that impressive, but this is what this algorithm is for: detecting deviations in shape.

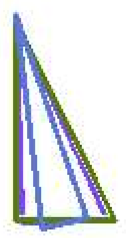


Spot the purple deviant

Now, as I've said before, Procrustes analysis is used to analyze the difference in shapes, and Procrustes distance is used as a measure. Here's a formula to calculate it:

Now that this is done, let's examine **exhibit B, generalized Procrustes analysis.** Which is basically the same thing we've just done with an extra for loop. The algorithm is outlined (way) above, so imma just leave you with the code:

I was lazy with the NumPy array initialization in the very beginning and used Python lists instead, and that came back to haunt me in this function. The code is a bit bumpy in places because I need to convert the lists to conveniently calculate means. I plotted the result to have fun with multicolored shapes one last time



So the odd triangle, the blue sheep, if you will, has now influenced (well, not terribly, but still) the shape of the rest of triangles. To see more dramatic changes, you can play around with the locations of vertices of triangles.

Wikipedia tells me that generalized Procrustes analysis is used to compare the results of surveys, interviews, and pretty much anything where there needs to be a "fair comparison" when accounting for some common factors. In fact, computer vision seems to have stolen it from statistics to use in a very narrow application domain.

So there. Hopefully, this was helpful and/or mildly entertaining. Either way, thanks for reading and happy shapes mutilation!